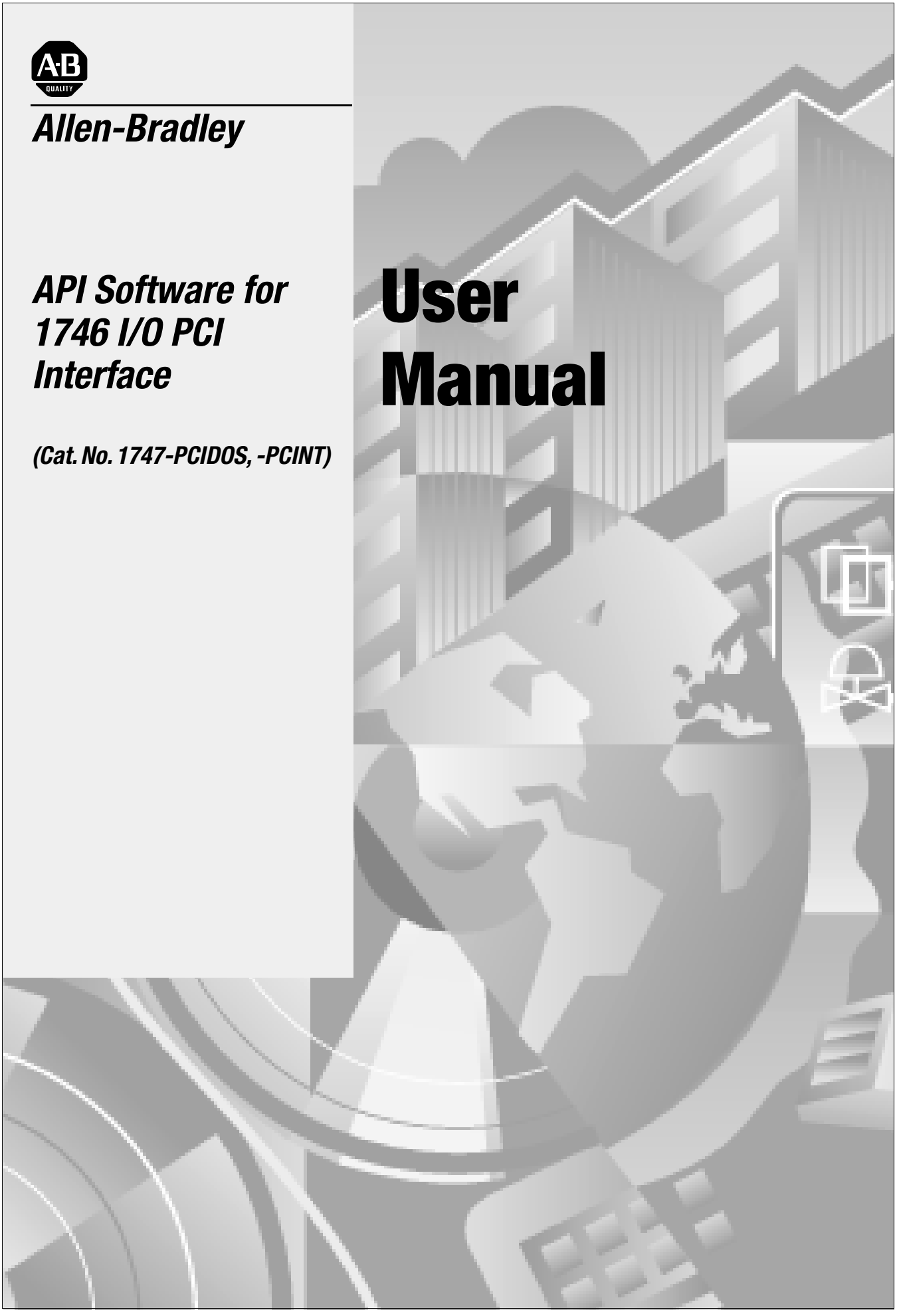


Allen-Bradley

API Software for 1746 I/O PCI Interface

(Cat. No. 1747-PCIDOS, -PCINT)

User Manual

Important User Information

Because of the variety of uses for the products described in this publication, those responsible for the application and use of this control equipment must satisfy themselves that all necessary steps have been taken to assure that each application and use meets all performance and safety requirements, including any applicable laws, regulations, codes and standards.

The illustrations, charts, sample programs and layout examples shown in this guide are intended solely for example. Since there are many variables and requirements associated with any particular installation, Allen-Bradley does not assume responsibility or liability (to include intellectual property liability) for actual use based upon the examples shown in this publication.

Allen-Bradley publication SGI-1.1, *Safety Guidelines For The Application, Installation and Maintenance of Solid State Control* (available from your local Allen-Bradley office) describes some important differences between solid-state equipment and electromechanical devices which should be taken into consideration when applying products such as those described in this publication.

Throughout this manual we use notes to make you aware of safety considerations:

ATTENTION: Identifies information about practices or circumstances that can lead to personal injury or death, property damage or economic loss.

Attention helps you to:

- identify a hazard
- avoid the hazard
- recognize the consequences

Important: Identifies information that is critical for successful application and understanding of the product.

Using This Manual

Who Should Use this Manual

Use this manual if you are responsible for developing control applications using the 1746 I/O PCI Interface API (application programming interface) software in an MS-DOS or Windows NT environment.

This manual documents the 1746 I/O PCI Interface API software for DOS and the API software for Windows NT. The APIs use most of the same calls. Differences are noted as appropriate.

Terminology

Throughout the language of this document, we refer to the 1746 I/O PCI Interface card (1747-PCIS) as the scanner and the 1747-PCIL chassis interface module as the adapter.

There are two versions of the PCI Bus Interface Card. 1747-PCIS has a 256k memory capacity and the 1747-PCIS2 has a 1M capacity.

Reference Material1746 I/O PCI Interface

The following books might be useful as you develop your 1746 I/O PCI Interface applications:

This document:	By:	Has this ISBN number:
PC System Architecture Series PCI System Architecture	MindShare, Inc. Addison-Wesley Publishing Company	ISBN: 0-201-40993-3
PCI Hardware and Software Architecture and Design	Edward Solari and George Willse	ISBN: 0-929392-28-0

Support

Due to the PC-based architecture of the 1746 I/O PCI Interface, the telephone support provided with the purchase price of the 1746 I/O PCI Interface consists primarily of determining if the system software and hardware is operating within documented specifications. The tools for this support are:

- diagnostic utility disk that ships with the 1746 I/O PCI Interface
- 1746 I/O PCI Interface system diagnostic LEDs

The diagnostic utility disk uses the DOS API as its programming interface, which provides examples of how to use the API. The diagnostic utility disk is a good tool to help diagnose your API application software. See appendix B for more information.

When you purchase a 1746 I/O PCI Interface system, you also receive firmware upgrades during the 12-month warranty period.

You can purchase extended support in blocks of 5 hours by ordering support contracts (1747-OCTS).

Table of Contents

Configuring I/O Modules

Chapter 5

Library of Routines

Chapter 6

Overview

Introduction

This chapter provides an overview of the 1746 I/O PCI Interface and the API software. This chapter also describes how to install the API.

You should have one of the following APIs:

- API for DOS (catalog number 1747-PCIDOS)
- API for Windows NT (catalog number 1747-PCINT)

The API software license agreement allows you to freely distribute the executable.

Relationship to the Open Controller

The API software for the 1746 I/O PCI Interface is compatible with the API for the 1747 Open Controller. The sample code and header files contain comments and functions that are supported by the Open Controller but not supported by the 1746 I/O PCI Interface. The following table lists the differences between the Open Controller and the 1746 I/O PCI Interface.

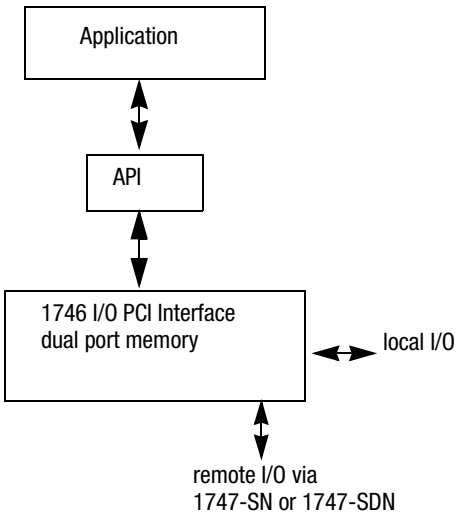
Open Controller	1746 I/O PCI Interface
User assigns interrupts and memory allocation.	1747-PCIS(2) is a plug-and-play card.
Watchdog can reset the entire system.	Watchdog cannot reset entire system.
Contains OC_ReadRtcSRAM.	Function not supported.
Contains OC_WriteRtcSRAM.	Function not supported.
Does not provide access to user SRAM.	Provides access to user SRAM.

Important: All references to Open Controller in the example code or header files apply to the 1746 I/O PCI Interface.

The 1746 I/O PCI Interface API

Use the 1746 I/O PCI Interface API to develop the software interface between your application and the 1746 I/O PCI Interface scanner to control local I/O and to control remote I/O via the 1747-SN or 1747-SDN scanners. The API provides calls for typical control functions, such as:

- configuring I/O files
- initializing the scanner
- accessing the user LEDs, user jumper, and 3-position switch
- reading 1746 I/O PCI Interface status
- enabling/disabling forces



API Software for DOS

The DOS API software provides a library of C function calls for DOS application programs to interface with the dual port memory. The DOS API supports any language compiler that uses the Pascal calling convention.

API Software for Windows NT

The Windows NT API supports any programming languages that use the Win32 _stdcall calling convention for application interface functions. The Windows NT API function names are exported from a DLL in undecorated format to simplify access from other programming languages.

The Windows NT API software consists of two main components:

- the 1746 I/O PCI Interface device driver (OCdriver)
- the API library, which is a DLL (dynamically-linked library)

The Windows NT API library is a DLL and must be installed on the system in order to run an application which uses it. The Windows NT API accesses the scanner via the driver created for the bus interface The driver:

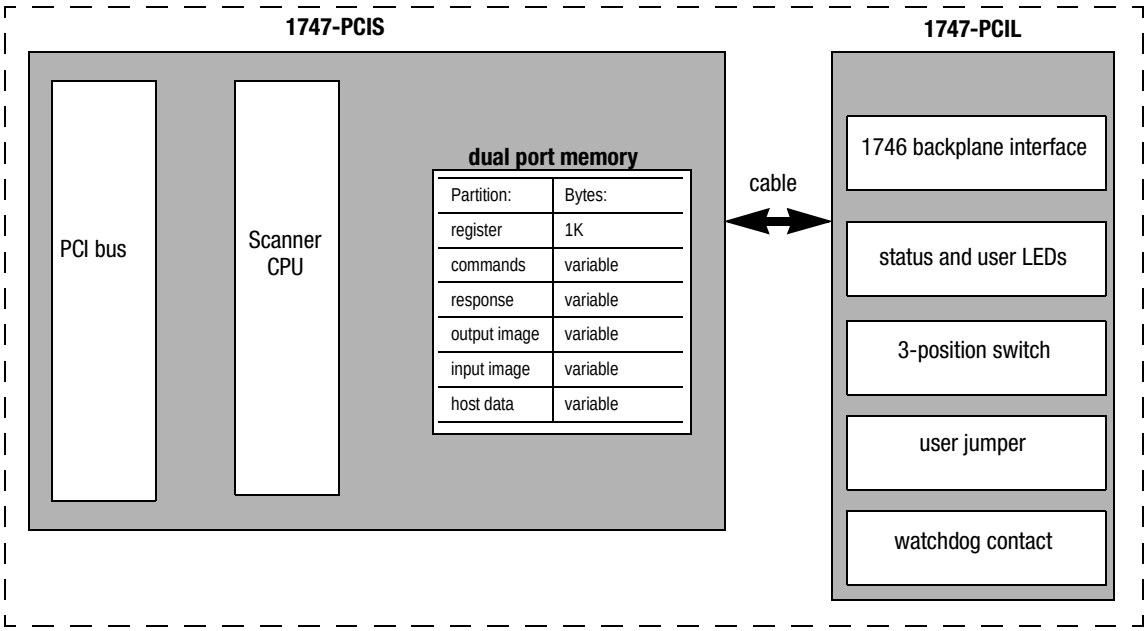
- allocates resources (interrupt and memory window)
- initializes scanner hardware
- provides access to the scanner's dual port RAM
- services interrupts from the scanner (priority messages)

Important: Only access the OCdriver through the API library functions.

When the OCdriver is loaded it tries to allocate an interrupt and a memory window for the memory and interrupt that were allocated using the settings by the PCI bus at power-up for the dual port RAM.

Understanding the 1746 I/O PCI Interface Architecture

The 1746 I/O PCI Interface architecture consists of a PCI card that plugs into a PC and cables to a 1746 I/O chassis. The scanner scans the 1746 local I/O bus and reads/writes inputs and outputs to/from the dual port registers.



The dual port is an 8K byte memory partition (optionally battery-backed) that provides an interface between the integrated scanner and your application software that resides on the host.

Important: The jumper for the battery-backed dual port memory is disabled by default. You must switch the jumper in order to enable the dual port memory battery-backed function. By enabling the battery-backed dual port memory, you will decrease the life of the battery.

Your application (the code you develop using the API) uses the dual port memory to communicate with the scanner to handle control functions on the 1746 backplane, such as:

- scanner commands and responses
- battery and scanner status
- scan rate frequency and timing
- I/O image counters
- priority messages and interrupts
- semaphores to ensure data integrity
- software-generated watchdogs
- control of the 4 user-definable LEDs, the 2-position jumper, and the 3-position switch

The scanner functionality of the dual port supports I/O control functions, such as:

- synchronizing scans to the application
- forcing I/O
- defining discrete-input interrupts
- defining I/O module-driven interrupts (such as for the 1746-BAS module)
- enabling and disabling I/O slots
- resetting I/O

In addition to providing access to the control scanner, the dual port memory also provides non-volatile (optional battery-backed) storage for:

- I/O values
- application parameters (timers, counters, presets)

Scanner Modes

The scanner CPU operates in six basic modes:

- performing POST (power-on self test)
- waiting for host initialization
- Idle
- Scan
- Faulted
- non-recoverable fault

After the scanner successfully completes the POST, the scanner waits for an initialization complete command from the application. Once the scanner receives this command, the scanner enters Idle mode.

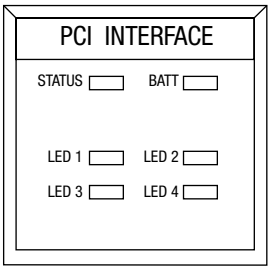
Before the scanner can enter Scan mode, the application must download a valid I/O configuration to the scanner.

If a recoverable fault occurs, the scanner enters Faulted mode. Use the OC_ClearFault API function to clear the fault before the scanner can resume operating in Scan mode.

If a non-recoverable fault occurs, reset the scanner (cycle power). Some possible non-recoverable faults include:

- POST failure
- background diagnostic failure
- internal watchdog timeout

Checking LED Indicators



STATUS

The STATUS indicator reports the status of the scanner. The following table lists the LED states for STATUS:

This state:	Means:	Take this action:
Yellow	The scanner is running POST.	None
Flashing green	The scanner is in idle mode and is not scanning I/O.	None
Solid green	The scanner is scanning I/O.	None
Flashing red	An I/O fault has occurred.	Check software to identify fault condition.
Solid red	A scanner internal fault has occurred.	Power system off and back on. If the problem persists, service may be required.
Off	The adapter is not powered up.	Turn on power.

Installing the DOS API

To install the DOS API, copy the following files to a directory you create. The sample code files are optional, but they show how to use the API functions.

This file:	Contains:
ocapil.lib	API functions that you link to your application
ocapi.h	API header file that contains API-referenced structures
sample.c	Sample application program calling the API functions
sampleb.mak	Sample make file for the Borland C compiler
samplem.mak	Sample make file for the Microsoft C compiler

Installing the Windows NT API

To install the Windows NT API, use the setup utility:

1. Insert the API diskette into a diskette drive.

   It is recommended that you exit all applications before starting the setup process.

2. Select Run from the startup menu, then select the `setup.exe` program from the API diskette.

3. Click on OK to execute the setup utility. Follow the displayed instructions. Click on Next.

4. The next dialog lets you choose whether to install the API development and executable files (Complete) or the API executable files (Runtime), or just the API development files (Development). To develop applications with the API, you need the development files. To only run applications, only the runtime files are necessary. The development files consist of an include file, import library, and sample code. The runtime files consist of a device driver and a dynamically-linked library.

   Important: Runtime files may only be installed on a Windows NT system. However, the development files may be installed on either Windows NT or Windows 95 systems.

   Choose the appropriate installation option and click on Next.

5. Specify the destination directory and click on Next.

6. The necessary files are copied to the disk, and the system registry is updated to include the OCdriver information.

7. Choose whether to reboot the system now or later and click on Finish.

Important: You must shutdown and reboot the scanner before using the API. (The setup utility sets the registry Start parameter for OCdriver to Automatic; therefore, the service manager starts the OCdriver when the system is booted.)

The Windows NT API uses these files:

This file:	Contains:
ocapi.lib	Import library in Microsoft COFF format
ocapi.h	API header file that contains API-referenced structures
ocapi.dll	API DLL
sample.c	Sample application program calling the API functions
sampleb.mak	Sample make file for the Borland C compiler
samplem.mak	Sample make file for the Microsoft C compiler

Installation Details

This section describes the actions the setup utility performs to install the API and OCdriver.

If you select Runtime (Complete), the setup utility:

1. copies the device driver file, `OCdriver`, to the system device driver directory (`%SystemRoot%\system32\drivers`).
2. adds this key and these values to the system registry:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\OCdriver
ErrorControl: REG_DWORD 0x1
Group: REG_SZ Extended base
Start: REG_DWORD 0x2
Type: REG_DWORD 0x1
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Drivers\ OCdriver
EventMessageFile= REG_EXPAND_SZ%SystemRoot%\System32\Drivers\OCdriver.sys
TypesSupported= REG_DWORD 0X00000007

3. copies the library file, `OCapi.dll`, to the `%SystemRoot%\system32` directory.

If you select Runtime & Development, the setup utility performs the same steps as for Runtime only and the setup utility copies `ocapi.lib` , `ocapi.h`, and the sample source files to a development directory.

Uninstalling the Windows NT API

To uninstall Windows NT API, use the following instructions.

1. From the Control Panel, select Add/Remove Programs.
2. From the list of installed programs, select Open Control API.
3. Click on Add/Remove.
4. Click on Yes.

   All of the API files and registry keys will be deleted.

Notes:

Using the API

Introduction

This chapter describes the API and how to use its components. For more information about developing applications using the API, see chapter 3.

Getting Started

To use the API, make sure you have copied the following files to your development directories. The sample files are optional.

This file:	Contains:
ocapil.lib	API functions that you link to your application (DOS only)
ocapi.lib	Import library in Microsoft COFF format (Windows NT only)
ocapi.h	API header file that contains API-referenced structures
ocapi.dll	API DLL (Windows NT only)
sample.c	Sample application program calling the API functions
sampleb.mak	Sample MAKE file for the Borland C compiler
samplem.mak	Sample MAKE file for the Microsoft C compiler

Your application must link to the appropriate library (`ocapil.lib` for DOS or `ocapi.lib` for Windows NT) and include `ocapi.h`. You can copy the sample files and adapt them for your application.

Programming Conventions

The API is supplied as an object code library file (`ocapil.lib`) or a DLL (`ocapi.dll`) that you link with the host application's object code using commercially-available tools.

This convention:	Considerations:
calling convention	The DOS API functions are specified using the C programming language syntax. To allow you to develop control applications in other programming languages, the API functions use the standard Pascal calling convention. The Windows NT API supports any programming languages that use the Win32 _stdcall calling convention for application interface functions. The Windows NT API function names are exported from the DLL in undecorated format to simplify access from other programming languages.
header files	The API includes a header file (`ocapi.h`) that contains API function declarations, data structure definitions, and other constant definitions. The header file is in standard C format.
sample code	The API comes with sample files to provide an example application that communicates with the scanner. The sample files include all source files and MAKE files required to build the sample application.
compiler support	The DOS API is supplied in the large memory model, compatible with both Microsoft and Borland compilers. The DOS library (`ocapil.lib`) is compiled as a 16-bit MS-DOS library using the 80386 instruction set. The Windows NT library (`ocapi.dll`) is compiled for use with Microsoft Visual C++ or Borland C++.

DOS Considerations

The DOS API is as consistent as possible with APIs for other operating system platforms. This makes it easier for you to migrate applications and it simplifies support.

To create a consistent API, careful consideration was given to these requirements:

This requirement:	Considerations:
memory mapping	The dual port RAM, or shared memory, is mapped automatically at power-up by the PCI bus in the host processor's memory address space on any **even** 8K boundary between 0xC0000 and 0xDFFFF. For MS-DOS, it is important that any installed memory managers (such as EMM386) or other TSR software avoid accessing the 8K dual port memory window. Place the base memory select jumper in 1M position to allow the PCI BIOS to assign a base memory address.
DOS interrupts	An interrupt is automatically assigned to the scanner by the PCI bus at power-up.
control-break handler	Because communication with the scanner requires memory and interrupt resources (as described above), improper termination of the host application can leave these resources allocated by the scanner and unusable by other applications. For this reason the API includes a default control-break handler. The default control-break handler is installed by OC_OpenScanner. If you press a `[Ctrl-C]` or `[Ctrl-Break]` key sequence, the following prompt is displayed: Terminate Application? (Y/N) _ A response of Y properly exits the application; a response of N causes the application to continue. If you need a different control-break handler, you must install it after calling the OC_OpenScanner function. Always call the OC_CloseScanner function before exiting an application.

Windows NT Considerations

During development, the application must be linked with an import library that provides information about the functions contained within the DLL. The API import library is compatible with the Microsoft linker. You can generate import libraries for other programming languages from the DLL.

The Windows NT API can only be accessed by one process at a time. The API is designed to be multi-thread safe, so that multi-threaded control applications can be developed. Where necessary, the API functions acquire a mutex before accessing the scanner interface. In this way, access to the scanner device is serialized. If the mutex is in use by another thread, a thread will be blocked until it is freed.

To create a consistent API, careful consideration was given to these requirements::

This requirement:	Considerations:
	The NT API device driver detects the Scanner Adapter and automatically configures the memory window address and interrupt assignment. The base memory address jumper must be positioned to choose 32 bit addressing. The API and device driver must be installed on the system.
memory mapping	Place the base memory select jumper in 32-bit position to allow the PCI BIOS to assign a base memory address anywhere in 32-bit memory for protected-mode applications (WinNT). NT device drivers (1747-PCINT) use the PCI BIOS or OS services to determine the memory window base address and provide access to the dual port memory. • To determine the allocated memory base address and interrupt, run the Windows NT diagnostic found in Administrative Tools.
NT interrupts	An interrupt is automatically assigned to the scanner by the PCI bus at power-up • To determine the allocated memory base address and interrupt, run the Windows NT diagnostic found in Administrative Tools.

A group of thread-blocking functions are provided to aid multi-threaded application development. The functions are:

- OC_WaitForDII
- OC_WaitForEos
- OC_WaitForEosDmdOut
- OC_WaitForIoInt
- OC_WaitForDmdIn
- OC_WaitForExtError

For more information, see chapter 6.

Tools to Use

The API functions support both Microsoft and Borland C compilers. The API disk includes sample MAKE files for each compiler.

When you use the DOS API and link to `ocapil.lib`, use the appropriate command-line switch to select the large memory model. For more information, see your user manual for your compiler.

If you plan to use a programming language other than C, make sure the programming language follows the appropriate calling convention (Pascal for the DOS API; Win32 _stdcall for Windows NT). After you write your application, use your compiler to link to `ocapil.lib` (DOS) or `ocapi.lib` (Windows NT).

Sample DOS MAKE file for Borland compilers

The following sample file shows how to use a Borland MAKE file. The bold headings indicate the statements you need to modify for your environment.

```
#*************************************************************************
#
# Title: Makefile for Open Controller API Sample
#
#   Abstract:
#       This file is used by the Borland MAKE utility to build the
#       sample application.
#
#   Environment:
#               1747-OCE Open Controller
#               MS-DOS
#       Borland C/C++ Compiler (16-bit)#
#*************************************************************************
#
# Paths to Tools
#
# Note: Modify the following paths to
# correspond to your environment.
#
#-------------------------------------------
CPATH   = D:\BC5                    # Location of Borland tools
CC      = $(CPATH)\bin\Bcc          # compiler
LINK    = $(CPATH)\bin\TLink        # linker
MAKE    = $(CPATH)\bin\Make         # make utility

#-------------------------------------------
# Path to API Library and Include file
#
# Note: Modify the following path to
# correspond to your environment.
#
#-------------------------------------------
APILIB  = ..\ocapil.lib         # Path to Open Controller API library
APIINC  = ..                    # Path to Open Controller API include file

#-------------------------------------------
# Options
#-------------------------------------------
CFLAGS  = -c -v- -w -ml -I$(APIINC)
LFLAGS  = -v- -Tde -d -c

sample.exe : sample.obj $(APILIB) sampleb.mak
    $(LINK) $(LFLAGS) c0l sample.obj, $*.exe, $*.map, $(APILIB) cl

clean:
    del *.exe
    del *.obj
    del *.map

rebuild:
    $(MAKE) clean
    $(MAKE)
.c.obj:
    $(CC) $(CFLAGS) $*.c

sample.obj: sample.c $(APIINC)\ocapi.h
sampleb.mak
```

Sample DOS MAKE file for Microsoft compilers

The following sample file shows how to use a Microsoft MAKE file. The bold headings indicate the statements you need to modify for your environment.

```
#*************************************************************************
#  Title: Makefile for Open Controller API Sample
#
#   Abstract:
#       This file is used by the Microsoft NMake utility to build the
#       sample application.
#
#   Environment:
#               1747-OCE Open Controller
#               MS-DOS
#               Microsoft C/C++ Compiler (16-bit)
#*************************************************************************

#-----------------------------------------------
# Note: The environment variables LIB and
# INCLUDE must be set to the path to the
# Microsoft C library and include files.
# For example:
#
#    set LIB=D:\MSVC15\LIB
#    set INCLUDE=D:\MSVC15\INCLUDE
#
#-----------------------------------------------
# Paths to Tools
#
# Note: Modify the following paths to
# correspond to your environment.
#
#-----------------------------------------------
CPATH   = D:\MSVC15                # Location of Microsoft tools
CC      = $(CPATH)\bin\cl          # compiler
LINK    = $(CPATH)\bin\link        # linker
MAKE    = $(CPATH)\bin\nmake       # make utility

#-----------------------------------------------
# Path to API Library and Include file
#
# Note: Modify the following path to
# correspond to your environment.
#
#-----------------------------------------------
APILIB  = ..ocapil.lib          # Path to Open Controller API library
APIINC  = ..                    # Path to Open Controller API include file

#-----------------------------------------------
# Options
#-----------------------------------------------
CFLAGS  = /c /nologo /G3 /W3 /AL /Oi /D /Gx- /I $(APIINC)
LFLAGS  = /MAP:A /NOI /PACKC

sample.exe : sample.obj $(APILIB) samplem.mak
    $(LINK) $(LFLAGS) sample.obj, $*.exe, $*.map, $(APILIB), nul.def

clean:
    del *.exe
    del *.obj
    del *.map

rebuild:
    $(MAKE) -f samplem.mak clean
    $(MAKE) -f samplem.mak

.c.obj:
    $(CC) $(CFLAGS) $*.c

sample.obj: sample.c $(APIINC)\ocapi.h
samplem.mak
```

Sample Windows NT MAKE file for Microsoft compilers

The following sample file shows how to use a Microsoft MAKE file. The bold headings indicate the statements you need to modify for your environment.

```
#*************************************************************************
#   Title: Makefile for Open Controller NT API Sample
#
#   Abstract:
#       This file is used by the Microsoft NMake utility to build the
#       sample application.
#
#   Environment:
#               1747-OCE Open Controller
#               Microsoft Windows NT 4.0
#               Microsoft Visual C++
#
#               (c)Copyright Allen-Bradley
#
#*************************************************************************

#-------------------------------------------
# Note: The environment variable LIB
# must be set to the path to the
# Microsoft C library files.
# For example:
#
#    set LIB=D:\MSDEV\LIB
#
#-------------------------------------------
# Paths to Tools
#
# Note: Modify the following paths to
# correspond to your environment.
#
#-------------------------------------------
CPATH   = D:\MSDEV                  # Location of Microsoft tools
CC      = $(CPATH)\bin\cl           # compiler
LINK    = $(CPATH)\bin\link         # linker
MAKE    = $(CPATH)\bin\nmake        # make utility

#-------------------------------------------
# Path to API Library and Include file
#
# Note: Modify the following paths to
# correspond to your environment.
#
#-------------------------------------------
APILIB  = ..\api\lib\ocapi.lib      # Path to Open Controller API library
APIINC  = ..\api\include            # Path to Open Controller API include file

#---------------------------------
# Compiler/Linker Debugging Options
# (Define DEBUG for debugging.)
#---------------------------------
!ifdef DEBUG
CDEBUG  = -Zi -Od
LDEBUG  = -debug:full -debugtype:cv
!else
CDEBUG  = -Ox
LDEBUG  = /RELEASE
```

```
!endif

#----------------------------------------
# Compiler Options
#
# -W3   Turn on warnings
# -GB   Optimize code for 80486/Pentium
# -MT   Use Multithreaded runtime lib
#
#----------------------------------------
CFLAGS  =   -W3 -GB -MT \
            -I$(APIINC) -I$(CPATH)\include

#---------------------------------
# Linker Options
#
#---------------------------------
LFLAGS  =   /NODEFAULTLIB /SUBSYSTEM:CONSOLE \
            /INCREMENTAL:NO /PDB:NONE

#---------------------------------
# Libraries
#
# libcmt    Multithreaded C runtime
# kernel32  Base system lib
#
#---------------------------------
LIBS    =   libcmt.lib kernel32.lib

#---------------------------------
# Final target
#---------------------------------
sample.exe : sample.obj $(APILIB)
    $(LINK) @<<
$(LDEBUG) $(LFLAGS) $(LIBS) $**
<<
    @echo Finished

clean:
    del *.exe
    del *.obj
    del *.map

rebuild:
    $(MAKE) -f samplem.mak clean
    $(MAKE) -f samplem.mak

#--------------------------
# Intermediate target rules
#--------------------------
.c.obj:
  $(CC) @<<
/c $(CDEBUG) $(CFLAGS) $*.c
<<

#---------------------------------
# Intermediate target dependancies
#---------------------------------

sample.obj: sample.c $(APIINC)\ocapi.h
samplem.mak
```

Sample Windows NT MAKE file for Borland compilers

The following sample file shows how to use a Borland MAKE file. The bold headings indicate the statements you need to modify for your environment.

```
#************************************************************************
#
#   Title: Makefile for Open Controller API Sample
#
#   Abstract:
#       This file is used by the Borland MAKE utility to build the
#       sample application.
#
#   Environment:
#               1747-OCE Open Controller
#               Microsoft Windows NT 4.0
#               Borland C++ Compiler
#
#               (c)Copyright Allen-Bradley
#
#************************************************************************
#-----------------------------------------------
# Paths to Tools
#
# Note: Modify the following paths to
# correspond to your environment.
#
#-----------------------------------------------
CPATH   = D:\BC5                    # Location of Borland tools
CC      = $(CPATH)\bin\Bcc32        # compiler
LINK    = $(CPATH)\bin\TLink32      # linker
MAKE    = $(CPATH)\bin\Make         # make utility

#-----------------------------------------------
# Path to API Library and Include file
#
# Note: Modify the following path to
# correspond to your environment.
#
#-----------------------------------------------
APIDLL  = ..\api\lib\ocapi.dll      # Path to Open Controller API library
APIINC  = ..\api\include            # Path to Open Controller API include file
APILIB  = .\ocapi.lib               # Borland compatible import library

#-----------------------------------------------
# Options
#-----------------------------------------------
CFLAGS  = -c -v -4 -tWM -w -I$(APIINC)
LFLAGS  = -v -Tpe -d -c -ap -r

#-----------------------------------------------
# Final Target
#-----------------------------------------------
sample.exe : sample.obj $(APILIB) sampleb.mak
    $(LINK) @&&|
$(LFLAGS) +
D:\BC5\LIB\c0x32.obj +
$*.obj, $*.exe, $*.map
D:\BC5\LIB\import32.lib +
D:\BC5\LIB\cw32mt.lib +
$(APILIB)
```

```
|

clean:
    del *.exe
    del *.obj
    del *.map
    del *.lib

rebuild:
    $(MAKE) -f sampleb.mak clean
    $(MAKE) -f sampleb.mak

.c.obj:
    $(CC) $(CFLAGS) $*.c

#-----------------------------------------------
# Create a Borland-compatible import library
#-----------------------------------------------
$(APILIB): $(APIDLL)
    implib $(APILIB) $(APIDLL)
```

sample.obj: sample.c $(APIINC)\ocapi.h sampleb.mak

Developing Applications

Introduction

This chapter describes the proper programming sequence for your application. This chapter also describes how to partition the dual port memory in the 1746 I/O PCI Interface.

How the API Functions Are Organized

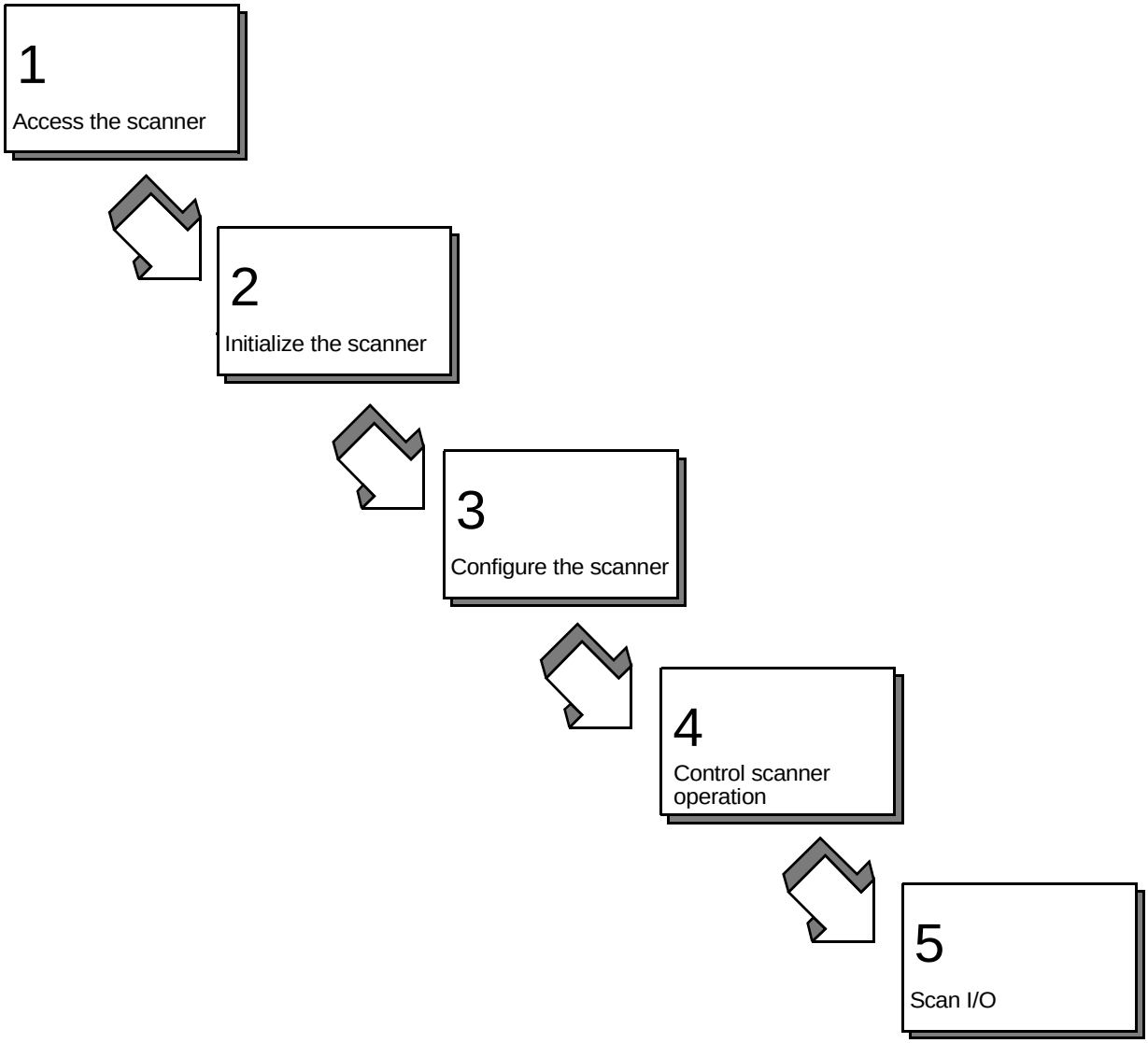
Each of the API functions falls into one of these four categories:

- scanner initialization
- scanner I/O configuration
- data input/output
- user interface/miscellaneous

Chapter 6 describes each API function and identifies its functionality group.

Programming Sequence

Follow this programming sequence when you develop your application.



Access the scanner

The host application must first call OC_OpenScanner to gain access to the scanner. Once an application has access, no other application can gain access to the scanner. When the application no longer requires access to the scanner, it must call OC_CloseScanner to release access of the scanner to other applications.

Once the scanner is opened, you must call OC_CloseScanner before exiting the application.

Initialize the scanner

After the scanner is reset and performs its POST, the scanner waits for initialization. In this state, the scanner can't be configured or control I/O. The only operational function is that which controls the LEDs. Any call to a function that requires the scanner to be initialized returns an error. You must initialize the scanner by sending it partitioning information before the host application can communicate with the scanner.

Initialize the scanner by calling the OC_InitScanner function to send the scanner partitioning information, which defines in bytes the size of the output image, the input image, and the host retentive data. Each of these memory segments must be at least large enough to hold their respective data, and must be an even number. If the input or output partition is initialized smaller than the actual size of the input or output image for a configuration, the OC_DownloadIOConfiguration function returns an error. The host retentive data size is optional and can be 0.

To determine the input image and output image sizes, use the OC_CreateIOConfiguration function to create an I/O configuration. OC_CreateIOConfiguration returns an I/O configuration with the number of bytes of inputs and outputs for each module. If a configuration already exists, you can use OC_GetIOConfig to return the current I/O configuration. The application can then calculate the minimum size of the segments required to hold the input and output images. For more information, see page 3-18.

The API has a defined constant specifying the total number of bytes available for the three segmenters This constant is specified as:
`OCSEGMENTSIZELIMIT`

Once the scanner has been initialized properly it cannot be re-initialized unless it is reset with the OC_ResetScanner function. Once the scanner is reset, scanner communications are disabled again until the scanner is initialized. The host application can call OC_GetScannerStatus to determine if the scanner has been initialized.

If the scanner was previously initialized, the host application can retrieve the initialization partition information with the OC_GetScannerInitInfo function instead of resetting and re-initializing the scanner.

Configure the scanner

To access I/O modules in a rack, you must define the rack sizes and installed module types for the scanner. You can either create a specific configuration or read the current configuration. The scanner cannot be set to Scan mode until it has been configured (received a valid scanner configuration).

If the scanner is in Scan mode and the host application has not downloaded a scanner configuration, the scanner has already been configured. To control I/O, use OC_GetIOConfiguration to retrieve the current scanner configuration.

The application can read the current I/O configuration with the OC_GetIOConfiguration function. If the scanner is not in Scan mode, this function returns the current scanner configuration which can be downloaded to the scanner using OC_DownloadIOConfiguration.

If the application requires a specific I/O configuration, the application can define the I/O configuration structure with the rack sizes and module types installed in each slot. The application passes this configuration structure to OC_CreateIOConfiguration. OC_CreateIOConfiguration returns a scanner configuration that can be downloaded to the scanner. For more information, see chapter 5.

Once a valid scanner configuration is successfully downloaded to the scanner via OC_DownloadIOConfiguration, the application can set the scanner to Scan mode and control I/O.

Both OC_CreateIOConfiguration and OC_GetIOConfiguration build the configuration data from an internal database of supported I/O modules.

Control scanner operation

Once the scanner has been configured, the application can control scanner operation. The host application can:

- set the scanner to Idle or Scan mode
- control the scan time
- control I/O
- read or write module files
- clear faults
- enable/disable slots
- set I/O Idle state
- install/remove forces
- handle module interrupts and discrete input interrupts

The API uses messages to communicate with the scanner. The scan time settings affect the time allowed by the scanner to process a message. OC_SetScanTime adjusts the scan time of the application.

The scanner processes messages during any available time that it is not scanning I/O. If the scan time is set too small, some API functions might take a relatively long time to complete. If some functions seem to be taking too long to complete, increase the scan time to provide more time for the scanner to process messages. If the scan time is set too large, I/O won't update fast enough. For information about estimating scan time, see *PCIS Bus Card for 1746 Local I/O Installation Instructions, publication 1747-5.31*.

Scan I/O

The scanner provides two basic methods for scanning I/O: timed scans and on-demand scans. The host application can use either, or a combination of both.

Typically, the scanner reads inputs from modules and writes outputs to modules once every scan time. To read inputs and write outputs, the application calls OC_ReadInputImage and OC_WriteOutputImage independently from the scanner's scan sequence.

The application can change the behavior of the input and output scans by allowing the application to have more control over I/O scanning. The application can prevent the scanner from doing any output scans and allow the application to read inputs and initialize outputs before the scanner begins to write outputs. This mode allows the application to pre-scan the inputs before the outputs are written.

The application can set the scanner to a conditional scan mode where the scanner writes outputs at the next scan time after the application writes data to the output image. In this mode, the scanner only writes outputs each time the application writes data to the output image.

The application can also prevent output scans by the scanner and have the scanner send a message after every input scan. The application can detect an end-of-scan message and then read the input image, perform logic, and write outputs using OC_DemandOutputScan to force the scanner to write outputs immediately. This lets the application synchronize its control loop with the input and output scans.

The application can also disable both input and output scans. In this mode, the scanner is a slave and input or output scans only take place when requested by the host application.

Programming Example for DOS

The following DOS example (`sample.c` on your API disk) shows how to program the above steps. Callouts on the right margin identify the code for each step.

```
/*************************************************************************
*
*    FILE:sample.c
*
*    PURPOSE:Sample application code for 1746 I/O PCI Interface API
*
*    SUMMARY:This program,
*             -  Resets and initializes the scanner.
*             -  Displays the scanner firmware and hardware versions.
*             -  Autconfigures the I/O in chassis.
*             -  Reads the front panel switch position and lights LED 1.
*             -  Reads first discrete input module data word.
*             -  Writes incremental data to first output module data word.
*             -  Closes connection to scanner and exits.
*
*    ENVIRONMENT:1747-PCIS 1746 I/O PCI Interface
*                     MS-DOS
*                     Borland/Microsoft C/C++ Compiler (16-bit)
*
*************************************************************************/

/*=======================================================================
=                           INCLUDE FILES                               =
=======================================================================*/

#include ”ocapi.h”
#include <stdio.h>
#include <dos.h>
#include <time.h>
#include <conio.h>
#include <string.h>

/*=======================================================================
=                     MODULE WIDE GLOBAL VARIABLES                      =
=======================================================================*/

HANDLE   Handle;           /* Software ID to scanner device */
OCIOCFG  OCcfg;            /* Chassis I/O config. data structure */

/*=======================================================================
=                         FUNCTION PROTOTYPES                           =
=======================================================================*/

void Ioexit( int );

/*=======================================================================
=                             MAIN PROGRAM                              =
=======================================================================*/

void main()
{
    int               retcode;     /* Return code from API calls */
    int               i;
    int               slots;
    int               input_slot, input_found = 0;
    int               output_slot, output_found = 0;
    OCINIT            ocpart;
    BYTE              status;
    OCVERSIONINFO     verinfo;
    BYTE              swpos;
    WORD              wData;
```

```
/*
**  Open the scanner
*/
retcode = OC_OpenScanner( &Handle, 0, 0);
if ( retcode != SUCCESS )
{
     printf( "\nERROR: OC_OpenScanner failed: %d\n", retcode );
     Ioexit( 1 );
}


/*
**  Reset the scanner
*/
printf( "\n\n  Going to reset OC, takes 6 seconds to complete...\n" );

retcode = OC_ResetScanner( Handle, OCWAIT );
if ( retcode != SUCCESS )
{
     printf( "\nERROR: OC_ResetScanner failed: %d\n", retcode );
     Ioexit( 1 );
}

/*
**  Check scanner status register
*/
retcode = OC_GetScannerStatus( Handle, &status );
if ( retcode != SUCCESS )
{
     printf( "\nERROR: OC_GetScannerStatus failed: %d\n", retcode );
     Ioexit( 1 );
}

if ( status != SCANSTS_INIT)
{
     printf("\nERROR: POST failure detected: %d\n", status);
     Ioexit(1);
}

/*
**  Initialize the DPR partitions
**  You can use OC_CreateIOConfiguration to determine the I/O image table
**  sizes before paritioning the DPR
*/
ocpart.OutputImageSize = 0x800;
ocpart.InputImageSize = 0x800;
ocpart.HostRetentiveDataSize = 0;

retcode = OC_InitScanner( Handle, &ocpart );
if ( retcode != SUCCESS )
{
     printf(" \nERROR: OC_InitScanner failed: %d\n", retcode );
     Ioexit( 1 );
}

/*
**  Display software/hardware versions
*/
retcode = OC_GetVersionInfo( Handle, &verinfo );
if ( retcode != SUCCESS )
{
     printf( "\nERROR: OC_GetVersionInfo failed: %d\n", retcode );
     Ioexit( 1 );
}
printf( "\n\n  Scanner Firmware Series: %02d Revision: %02d  ",
     verinfo.ScannerFirmwareSeries, verinfo.ScannerFirmwareRevision );
printf( "\n  Hardware Series: %02d Revision: %02d",
     verinfo.OCHardwareSeries, verinfo.OCHardwareRevision );
delay( 3000 );
```

Access the scanner
See page 6-48.

Initialize the scanner
See pages 6-63, 6-33, and 6-7.

```
/*
**  Read switch position
*/
retcode = OC_GetSwitchPosition( Handle, &swpos );
if ( retcode != SUCCESS )
{
    printf( "\nERROR: OC_GetSwitchPosition failed: %d\n", retcode );
    Ioexit( 1 );
}
printf( "\n\n  Switch position: " );
switch( swpos )
{
    case SWITCH_TOP:
        printf( "Top \n" );
        break;
    case SWITCH_BOTTOM:
        printf( "Bottom \n" );
        break;
    case SWITCH_MIDDLE:
        printf( "Middle \n" );
        break;
}
delay( 3000 );

/*
**  Read auto-config
*/
retcode = OC_GetIOConfiguration( Handle, &OCcfg );
if ( retcode != SUCCESS )
{
    printf( "\nERROR: OC_GetIOConfiguration failed: %d\n", retcode );
    Ioexit( 1 );
}

/*
**  Display rack configuration
*/
slots = OCcfg.Rack1Size + OCcfg.Rack2Size + OCcfg.Rack3Size;
if ( slots > 31 )
    slots = 31;
printf( "\n\n         Chassis configuration " );
for ( i=1; i < slots; i++ )
{
    if ( OCcfg.SlotCfg[i].type != 0xff )
        printf( "\n  Slot %2d: Type %2d, Mix %3d  %s",
            i, OCcfg.SlotCfg[i].type, OCcfg.SlotCfg[i].mix,
            OCcfg.SlotCfg[i].Name );
    else
        printf( "\n  Slot %2d: %s", i, OCcfg.SlotCfg[i].Name );

    /* Find digital input card */
    if ( OCcfg.SlotCfg[i].mix < 8 && !input_found )
    {
        input_found = 1;
        input_slot = i;
    }
    /* Find digital output card */
    if ( (OCcfg.SlotCfg[i].mix > 7) && (OCcfg.SlotCfg[i].mix < 32) && !output_found )
    {
        output_found = 1;
        output_slot = i;
    }
}
delay( 3000 );
```

```
/*
**  Download the configuration to the scanner
*/
retcode = OC_DownloadIOConfiguration( Handle, &OCcfg );
if ( retcode != SUCCESS )
{
     printf( "\nERROR: OC_DownloadIOConfiguration failed: %d\n", retcode );
     Ioexit( 1 );
}
```

Configure the scanner See page 6-11.

```
/*
**  Set output update mode to always
*/
retcode = OC_SetOutputUpdateMode( Handle, OUTUPD_ALWAYS );
if ( retcode != SUCCESS )
{
     printf( "\nERROR: OC_SetOutputUpdateMode failed: %d\n", retcode );
     Ioexit( 1 );
}

/*
**  Set scan time to 5ms, periodic scan mode
*/
retcode = OC_SetScanTime( Handle, SCAN_PERIODIC, 20 );
if ( retcode != SUCCESS )
{
     printf( "\nERROR: OC_SetScanTime failed: %d\n", retcode );
     Ioexit( 1 );
}

/*
**  Goto Scan Mode
*/
retcode = OC_SetScanMode( Handle, SCAN_RUN );
if ( retcode != SUCCESS )
{
     printf( "\nERROR: OC_SetScanMode failed: %d\n", retcode );
     Ioexit( 1 );
}

/*
**  Turn on User LED 1
*/
retcode = OC_SetUserLEDState( Handle, 1, LED_GREEN_SOLID );
if ( retcode != SUCCESS )
{
     printf( "\nERROR: OC_SetUserLEDState failed: %d\n", retcode );
     Ioexit( 1 );
}
printf( "\n\n  LED1 is on solid green now. \n" );
delay( 3000 );
```

Control scanner operation See pages 6-70 and 6-73.

```
    /*
    **  Read first Input word
    */
    retcode = OC_ReadInputImage( Handle, NULL, input_slot, 0, 1, &wData );
    if ( retcode != SUCCESS )
    {
        printf( "\nERROR: OC_ReadInputImage failed: %d\n", retcode );
        Ioexit( 1 );
    }
    printf( "\n\n  First input image data word --> 0x%04x \n", wData );
    delay( 3000 );

    /*
    **  Write to the first Output word
    */

    printf( "\n\n  Incrementing first discrete output word. \n" );
    for ( wData=0; wData < 256; wData++)
    {
        retcode = OC_WriteOutputImage( Handle, NULL, output_slot, 0, 1, &wData );
        if ( retcode != SUCCESS )
        {
            printf("\nERROR: OC_WriteOutputImage failed: %d\n", retcode);
            Ioexit(1);
        }
        delay ( 10 );
    }

    /*
    **  Must always close the scanner before exiting
    */
    OC_CloseScanner( Handle );

    printf( "\n\n  Program is done! \n\n" );

}       /* end main() */
```

Scan I/O
See page 6-54 and 6-90.

```
/************************************************************************
*
*    Name:  Ioexit
*
*    Description:
*
*         Common error handling routine.  This routine displays any
*         extended error and exits the program.
*
*    Arguments:
*         retcode : int( input )
*         This error code is passed to the exit() routine.
*
*    External effects:
*         The program is terminated.
*
*    Return value:
*         none
*
************************************************************************/
void Ioexit( int retcode )
{
     OCEXTERR exterr;
     char *msg;
     if (OC_GetExtendedError(Handle, &exterr) == SUCCESS)
     {
          if ( exterr.ErrorCode != 0 )
          {
               OC_ExtendedErrorMsg(Handle, &exterr, &msg);
               printf("\nERROR: %s\n", msg);
          }
     }

     OC_CloseScanner(Handle);

     exit(retcode);

}         /* end Ioexit() */
```

Programming Example for Windows NT

The following Windows NT example (`sample.c` on your Windows NT API disk) shows how to program the above steps. Callouts on the right margin identify the code for each step.

```
/*********************************************************************
*    Title: Simple application sample code for 1746 I/O PCI Interface API
*
*    Abstract:
*
*        This file contains a simple application using the PCI
*        bus interface API.
*
*    Environment:
*               1747-PCIS 1746 I/O PCI Interface
*               Microsoft Windows NT 4.0
*               Microsoft Visual C++ / Borland C++
*                (c)Copyright Allen-Bradley *
*********************************************************************/

/*====================================================================
=                           INCLUDE FILES                            =
====================================================================*/

#include <windows.h>
#include <process.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>
#include <string.h>
#include "ocapi.h"

/*====================================================================
=                  MODULE WIDE GLOBAL VARIABLES                      =
====================================================================*/

HANDLE      OChandle;
OCIOCFG     OCcfg;

/*********************************************************************
*      Entry point:
*        Ioexit
*
*      Description:
*       Common error handling routine.  This routine displays any
*       extended error and exits the program.
*
*      Arguments:
*       rc        : int             ( input )
*       This error code is passed to the exit() routine.
*
*      External effects:
*        The program is terminated.
*
*      Return value:
*        none
*
*      Access: Public
* -------------------------------------------------------------------
*     Notes:
*
*********************************************************************/
void Ioexit
     int rc
 ) {
     OCEXTERR exterr;
     char *msg;
```

```
    if (OC_GetExtendedError(OChandle, &exterr) == SUCCESS)
    {
         if ( exterr.ErrorCode != 0 )
         {
             OC_ExtendedErrorMsg(OChandle, &exterr, &msg);
             printf("\n\nERROR: %d %s\n", msg, exterr.ErrorCode);
         }
     }
     OC_CloseScanner(OChandle);

     exit(rc);

} /* end Ioexit() */

/*************************************************************************
*     Entry point:
*              tErrorEvent
*
*     Description:
*       Thread to handle errors.
*
*     Arguments:
*       none
*
*     External effects:
*       none
*
*     Return value:
*       none
*
*     Access: Public
*
*-------------------------------------------------------------------------
*     Notes:
*
*************************************************************************/
 void tErrorEvent( void *dummy )
 {

    while(1)
    {
         /* Sleep until the scanner reports an error */
             OC_WaitForExtError(OChandle, INFINITE);

         /* An error has occurred.  Perform whatever error handling */
         /* that is necessary.  In this case, we just print a message */
         /* and exit the process. */

        Ioexit(1);
            }

} /* end tErrorEvent() */
```

```
/*********************************************************************** *
*     Entry point:
*        main
*
*     Description:
*        Entry point of the PCI I/O bus interface API sample application.
*
*        This program resets, initializes, and autoconfigures the scanner.
*        It displays the scanner firmware and hardware versions, and
*        the front panel switch position.
*        It lights User LED 1, reads inputs from a 32 pt discrete
*        input module, and writes data to the M0 file on a BAS module.
*
*     Arguments:
*        none
*
*     External effects:
*        none
*
*     Return value:
*        0 if no errors were encountered
*        1 if errors
*
*     Access: Public
*
*---------------------------------------------------------------------
*     Notes:
*
************************************************************************/
main()
{
     int rc;
     int i;
     int slots;
     int BASslot;
     int IB32slot;
     int fRecreateIOcfg;
     OCINIT ocpart;
     BYTE status;
     OCVERSIONINFO verinfo;
     BYTE swpos;
     WORD wData,wLen;
     BYTE temp;

     BASslot = IB32slot = 0;
     fRecreateIOcfg = 0;

    /* Open the scanner */
     if (SUCCESS != (rc = OC_OpenScanner(&OChandle)))
     {
         printf("\nERROR: OC_OpenScanner failed: %d\n", rc);
         Ioexit(1);
     }

    /* Create an error-handling thread */
     if (-1 == (long) _beginthread(tErrorEvent, 0, NULL))
         printf("\nERROR: _beginthread tErrorEvent failed.\n");
```

Access the scanner
See page 6-48.

```
/* Reset the scanner */
 printf("\nResetting the scanner...");
 if (SUCCESS != (rc = OC_ResetScanner(OChandle, OCWAIT)))
 {
     printf("\nERROR: OC_ResetScanner failed: %d\n", rc);
     Ioexit(1);
 }

/* Check scanner status register */
 if (SUCCESS != (rc = OC_GetScannerStatus(OChandle, &status)))
 {
     printf("\nERROR: OC_GetScannerStatus failed: %d\n", rc);
     Ioexit(1);
 }

if ( status != SCANSTS_INIT)
{
     printf("\nERROR: POST failure detected: %d\n", status);
     Ioexit(1);
 }

/* Initialize the DPR partitions */
 ocpart.OutputImageSize = 0x800;
 ocpart.InputImageSize = 0x800;
 ocpart.HostRetentiveDataSize = 0;
 if (SUCCESS != (rc = OC_InitScanner(OChandle, &ocpart)))
 {
     printf("\nERROR: OC_InitScanner failed: %d\n", rc);
     Ioexit(1);
 }

/* Display software/hardware versions */
 if (SUCCESS != (rc = OC_GetVersionInfo(OChandle, &verinfo)))
 {
     printf("\nERROR: OC_GetVersionInfo failed: %d\n", rc);
     Ioexit(1);
 }
 printf("\nOC API Series: %02d Revision: %02d  ",
     verinfo.APISeries,verinfo.APIRevision);
 printf("\nOCdriver Series: %02d Revision: %02d  ",
     verinfo.OCdriverSeries, verinfo.OCdriverRevision);
 printf("\nOC Scanner Firmware Series: %02d Revision: %02d  ",
     verinfo.ScannerFirmwareSeries, verinfo.ScannerFirmwareRevision);
 printf("\nOC Hardware Series: %02d Revision: %02d\n",
     verinfo.OCHardwareSeries, verinfo.OCHardwareRevision);

 /* Read switch position */
 if (SUCCESS != (rc = OC_GetSwitchPosition(OChandle, &swpos)))
 {
     printf("\nERROR: OC_GetSwitchPosition failed: %d\n", rc);
     Ioexit(1);
 }
 printf("\nSwitch position: ");
 switch(swpos)
 {
     case SWITCH_TOP:
         printf("Top");
         break;
     case SWITCH_BOTTOM:
         printf("Bottom");
         break;
     case SWITCH_MIDDLE:
         printf("Middle");
          break;     }

/* Read temperature */
 if (SUCCESS != (rc = OC_GetTemperature(OChandle, &temp)))
 {
     printf("\nERROR: OC_GetTemperature failed: %d\n", rc);
     Ioexit(1);
 }
 printf("\nTemperature: %dC  ", temp);
```

Initialize the scanner
See pages 6-63, 6-33, and 6-46.

```
    /* Read auto-config */
     if (SUCCESS != (rc = OC_GetIOConfiguration(OChandle, &OCcfg)))
     {
         printf("\nERROR: OC_GetIOConfiguration failed: %d\n", rc);
         Ioexit(1);
     }

    /* Display rack configuration */
     slots = OCcfg.Rack1Size + OCcfg.Rack2Size + OCcfg.Rack3Size;
     if ( slots > 31 )   slots = 31;

    printf("\n\nRack configuration:");
    for (i=1; i<slots; i++)
    {
         if (OCcfg.SlotCfg[i].type != 0xff)
         {
             printf("\nSlot %2d: Type %2d, Mix %3d  %s",
                 i, OCcfg.SlotCfg[i].type, OCcfg.SlotCfg[i].mix,
                 OCcfg.SlotCfg[i].Name);
         }
         else
         {
             printf("\nSlot %2d: %s", i, OCcfg.SlotCfg[i].Name);
         }

        /* check for BAS modules class 1 or 4 */
         if ( ((OCcfg.SlotCfg[i].mix == 35) || (OCcfg.SlotCfg[i].mix == 131))
 && (OCcfg.SlotCfg[i].type == 6))
         {
             if ( OCcfg.SlotCfg[i].mix == 35 )
             {                                  /* if Class 1 BAS module, then ...
                OCcfg.SlotCfg[i].mix = 131;     /* ...make it class 4 */
               OCcfg.SlotCfg[i].Name = NULL;     /* remove name so that OC_CreateIOConfiguration
will key off mix/type */
                 fRecreateIOcfg = 1;
             }
             BASslot = i;
         }

        /* check for IB32 modules */
         if ( OCcfg.SlotCfg[i].mix == 7 )
         {
             IB32slot = i;
         }
     }

    /* if we converted a Class 1 BAS module to Class 4, recreate the IO configuration */
    /* to insure we get the M0 and M1 file sizes */
     if (fRecreateIOcfg == 1)
     {
         if (SUCCESS != (rc = OC_CreateIOConfiguration(&OCcfg)))
         {
             printf("\nERROR: OC_CreateIOConfiguration failed: %d\n", rc);
             Ioexit(1);
         }
     }

    /* Download the configuration to the scanner */
     if (SUCCESS != (rc = OC_DownloadIOConfiguration(OChandle, &OCcfg)))
     {
         printf("\nERROR: OC_DownloadIOConfiguration failed: %d\n", rc);
         Ioexit(1);
     }
```

Configure the scanner See page 6-11.

```
    /* Set output update mode to always */
     if (SUCCESS != (rc = OC_SetOutputUpdateMode(OChandle, OUTUPD_ALWAYS)))
     {
         printf("\nERROR: OC_SetOutputUpdateMode failed: %d\n", rc);
         Ioexit(1);
     }

    /* Set scan time to 5ms, periodic scan mode */
     if (SUCCESS != (rc = OC_SetScanTime(OChandle, SCAN_PERIODIC, 20)))
     {
         printf("\nERROR: OC_SetScanTime failed: %d\n", rc);
         Ioexit(1);
     }

    /* Goto Scan Mode */
     if (SUCCESS != (rc = OC_SetScanMode(OChandle, SCAN_RUN)))
     {
         printf("\nERROR: OC_SetScanMode failed: %d\n", rc);
         Ioexit(1);
    }

    /* Turn on User LED 1 */
     if (SUCCESS != (rc = OC_SetUserLEDState(OChandle, 1, LED_GREEN_SOLID)))
     {
         printf("\nERROR: OC_SetUserLEDState failed: %d\n", rc);
         Ioexit(1);
     }

    /* Read word 0 of IB32 module */
     if ( IB32slot != 0 )
   {  if (SUCCESS != (rc = OC_ReadInputImage(OChandle, NULL, IB32slot, 0, 1, &wData)))
         {
             printf("\nERROR: OC_ReadInputImage failed: %d\n", rc);
             Ioexit(1);
          }     }

    /* Write the data read to word 2 of BAS module M0 file  */
     wLen = 1;     if ( BASslot != 0 )
     {
      if (SUCCESS != (rc = OC_WriteModuleFile(OChandle, FILTYP_M0, &wData, BASslot,
2, wLen)))
         {
             printf("\nERROR: OC_WriteModuleFile failed: %d\n", rc);
             Ioexit(1);
         }
     }

    /* Close the scanner before exiting */
     OC_CloseScanner(OChandle);

    return(0);

} /* end main()*/
```

Control scanner operation
See pages 6-70 and 6-73.

Scan I/O
See pages 6-54 and 6-88.

Handling Interrupt Messages

Modules that communicate via discrete input interrupts or module interrupts require special attention. The API buffers these interrupts internally until they are extracted via OC_PollScanner. The internal message buffer can hold as many as 5 messages. If the message buffer is full, the oldest message in the buffer is overwritten by the next message. Interrupts will be missed if OC_PollScanner is not called by the application more often than interrupts are received.

For Windows NT, use the OC_WaitFor*xxx* functions.

Handling Errors

Every function call returns a status code for the function. Check this status code before using the data returned by the function. The scanner reports faults and other errors via messages. The API library buffers these errors internally and reports their existence as an Extended Error. The application must periodically call OC_GetExtendedError to determine if an extended error message exists.

The library buffers extended errors in a queue. The queue can hold as many as 5 extended errors at one time. If the queue is full when a new extended error is received from the scanner, the oldest extended error is lost and ERR_OCOVERRUN is returned. The host application must call OC_GetExtendedError periodically to remove existing extended errors from the buffer.

Extended Errors cause the scanner to fault. Once the scanner is faulted, it is forced to Idle mode and cannot go to Scan mode until the Extended Errors are extracted via OC_GetExtendedError and the fault is cleared via OC_ClearFault. For Windows NT, use the OC_WaitForExtError function.

Determining Partition Sizes for Shared Memory

The host application initializes the scanner by providing partitioning information, which contains the size of memory to be reserved in the shared memory for the input and output images. The size of memory to be reserved for each of the images must be greater than or equal to the number of input and output words required by each module. The host application can't communicate with the scanner until it has been initialized.

The partitioning information is passed to OC_InitScanner in the OCINIT structure, which is defined as:

```
typedef struct tagOCINIT {
         WORD OutputImageSize;        /* size in bytes */
         WORD InputImageSize;         /* size in bytes */
         WORD HostRetentiveDataSize;  /* size in bytes */
} OCINIT;
```

To determine the input and output image sizes, call OC_CreateIOConfiguration with a configuration that contains the I/O modules to be installed. OC_CreateIOConfiguration returns the number of bytes of I/O required by each module. Or you can use OC_GetIOConfig to use the current configuration, if one exists. The input and output sizes are based on the number of words of I/O required by each module. As an estimate, take the total number of input and output words for all the modules in the system and multiply by two to get the number of required bytes. The following code fragment calculates the number of bytes required by the input and output images:

```
OCINIT   initinfo;
OCIOCFG  iocfg;
int           i,numslots;

/* assuming application has filled iocfg with I/O configuration */
OC_CreateIOConfiguration(&iocfg);

numslots = iocfg.Rack1Size + iocfg.Rack2Size + iocfg.Rack3Size;
if ( numslots > 31 )   numslots = 31;
initinfo.OutputImageSize = initinfo.InputImageSize = 0;
for ( i=1 ; i<numslots ; i++) {
         initinfo.OutputImageSize += ((iocfg.SlotCfg[i].OutputSize+1) / 2) * 2;
         initinfo.InputImageSize += ((iocfg.SlotCfg[i].InputSize+1) / 2) * 2;
}
```

Any remaining shared memory can be allocated for host retentive data, which is the portion of the dual port RAM that you can use to store data in case power fails. If the application doesn't need host retentive data, set its size to 0. If the application needs host retentive data, the application can determine the amount of memory available by using the OCSEGMENTSIZELIMIT constant.

This constant specifies the total number of bytes available for the three segment sizes. To calculate the maximum memory available for the host retentive data, use this formula:

```
initinfo.HostRetentiveDataSize =
     OCSEGMENTSIZELIMIt - initinfo.OutputImageSize - initinfo.InputImageSize;
```

If the I/O configuration changes and causes the image sizes to change, the maximum memory available for Host Retentive Data will change accordingly, and information stored in the Host Retentive Data memory may be overwritten.

Notes:

Using the API Structures

Introduction

This chapter describes the structures the API uses. These structures are declared in `ocapi.h`.

API Structures

Structure Name:	Syntax:

DII_CFG
Passed to OC_ConfigureDII. Configures a discrete input interrupt for a module.
	`typedef struct tagDII_CFG`
	`{`
	`  BYTE  SlotNum;  /* slot number */`
	`  BYTE  IOIncludeMask; /* declare which Discrete Inputs can cause interrupts */`
	`  BYTE  IOEdgeType;  /* select required transition of each discrete input */`
	`  WORD  PresetCount;  /* set the number of transitions required to cause interrupt */`
	`} DII_CFG;`

FORCEDATA
Passed to OC_SetForces. Configures input and output forces.
	`typedef struct tagFORCEDATA`
	`{`
	`  BYTE  SlotNum;  /* slot number */`
	`  WORD  WordOffset;  /* offset to word to force */`
	`  BYTE  IOType;  /* selects force inputs or outputs */`
	`  WORD  ForceMask;  /* bits set to 1 are forced, 0 removes forces */`
	`  WORD  ForceVal;  /* selects force state of bits set to 1 in ForceMask */`
	`} FORCEDATA;`

MSGBUF
Returned by OC_PollScanner. MsgID identifies the message type. Type-specific data is contained in `MsgData[ ]`.
	`#define OCMSGDATASIZE4  /* number of bytes of message data */`
	`typedef struct tagMSGBUF`
	`{`
	`  BYTE  MsgID;  /* Message type  */`
	`  BYTE  MsgData[OCMSGDATASIZE];  /* Type-specific data  */`
	`} MSGBUF;`

OCEXTERR
Returned by OC_GetExtendedError. I/O error report from scanner.
	`#define OCERRDATASIZE3  /* number of bytes of error data */`
	`typedef struct tagOCEXTERR`
	`{`
	`  BYTE  ErrorCode;  /* Extended error code  */`
	`  BYTE  SlotNum;  /* Associated slot number */`
	`  BYTE  ErrorData[OCERRDATASIZE];/* Error code data  */`
	`} OCEXTERR;`

OCINIT
Passed to OC_InitScanner function to specify dual port RAM partition sizes for output image, input image, and host retentive data.
	`typedef struct tagOCINIT`
	`{`
	`  WORD  OutputImageSize;  /* size in bytes */`
	`  WORD  InputImageSize;  /* size in bytes */`
	`  WORD  HostRetentiveDataSize;/* size in bytes */`
	`} OCINIT;`

OCIOCFG
Used by OC_CreateIOConfiguration, OC_GetIOConfiguration, and OC_DownloadIOConfiguration. Configuration information for a system. 1, 2, or 3 racks may be configured for up to 30 I/O modules. (Slot 0 is reserved for the 1746 I/O PCI Interface.)
	`typedef struct tagOCIOCFG`
	`{`
	`  BYTE  Rack1Size;  /* number of slots in Rack 1 */`
	`  BYTE  Rack2Size;  /* number of slots in Rack 2 */`
	`  BYTE  Rack3Size;  /* number of slots in Rack 3 */`
	`  OCSLOTCFGSlotCfg[OCMAXSLOT];/* configuration for each slot */`
	`} OCIOCFG;`

Structure Name:	Syntax:

OCSLOTCFG
Configuration information for a module. The mix and type codes together form a unique identification for each module.
	`typedef struct tagOCSLOTCFG`
	`{`
	`  BYTE  mix;  /* mix code */`
	`  BYTE  type;  /* type code */`
	`  BYTE  InputSize;  /* number of inputs in bytes */`
	`  BYTE  OutputSize;  /* number of outputs in bytes */`
	`  WORD  M0Size;  /* size of M0 file in words */`
	`  WORD  M1Size;  /* size of M1 file in words */`
	`  WORD  GSize;  /* size of G file in words */`
	`  WORD  *GData;  /* pointer to array of length GSize words */`
	`  char  *Name;  /* pointer to module name string */`
	`} OCSLOTCFG;`

OCVERSIONINFO
Returned by OC_GetVersionInfo. Software and hardware version numbers.
	`typedef struct tagOCVERSIONINFO`
	`{`
	`  WORD  APISeries;  /* API series */`
	`  WORD  APIRevision;  /* API revision */`
	`  WORD  ScannerFirmwareSeries;  /* Scanner firmware series */`
	`  WORD  ScannerFirmwareRevision;/* Scanner firmware revision */`
	`  WORD  OCHardwareSeries;  /* Hardware series */`
	`  WORD  OCHardwareRevision;  /* Hardware revision */`
	`  WORD  OCdriverSeries  /* OCdriver series - Windows NT only */`
	`  WORD  OCdriverRevision  /* Ocdriver reviwion - Windows NT only */`
	`} OCVERSIONINFO;`

STSFILE
Scanner status file.
	`typedef struct tagSTSFILE`
	`{`
	`  WORD  wWordNum[OCSTSFILEWSIZE];`
	`} STSFILE;`

Configuring I/O Modules

Introduction

This chapter explains how to configure the I/O modules for your 1746 I/O PCI Interface system. You can either use the autoconfigure (OC_GetIOConfiguration) function or build your own configuration (OC_CreateIOConfiguration).

A separate I/O configuration utility is available for the PCI SLC I/O bus interface to simplify this process. The utility is on the 1746 I/O PCI Interface utilities disk that ships with the 1746 I/O PCI Interface (1747-PCIS[2]). The I/O configuration utility (ioconfig.exe) allows an I/O configuration data file to be created and saved to disk.

Configuring I/O

The application configures the scanner by downloading information about the installed rack sizes and module types. Call the OC_GetIOConfiguration function to get the current I/O configuration or use OC_CreateIOConfiguration to build an I/O configuration. Both of these functions return a valid I/O configuration that can be downloaded to the scanner.

The scanner will not go to Scan mode until the OC_DownloadIOConfiguration function sends the configuration information. The scanner checks the downloaded I/O configuration against the installed modules when the application attempts to set the scanner to Scan mode. The scanner returns an extended error if the I/O configuration is not valid and the scanner will fault.

The OC_CreateIOConfiguration function requires a structure containing rack sizes and module types or module names. The structure is:

```
struct {
   BYTE     Rack1Size;  /* number of slots in rack1 (4,7,10, or 13) */
   BYTE     Rack2Size;  /* number of slots in rack2 (0,4,7,10, or 13) */
   BYTE     Rack3Size;  /* number of slots in rack3 (0,4,7,10, or 13) */
   OCSLOTCFGSlotCfg[31];/* slot information */
} OCIOCFG;
```

Initialize the three rack size variables with the total number of slots in each rack. If rack 2 or rack 3 is not installed, set the size to 0.

`SlotCfg` contains information about each slot in the racks. The 1746 I/O PCI Interface supports as many as 31 slots, numbered 0 to 30. Slot 0 is the adapter slot (left slot of rack 1) and is invalid for scanner functions. Each slot is described by the structure `OCSLOTCFG`:

```
struct {
   BYTE     mix;        /* Module I/O Mix value */
   BYTE     type;       /* Module Type */
   BYTE     InputSize;  /* number of inputs in bytes */
   BYTE     OutputSize; /* number of outputs in bytes */
   WORD     M0Size;     /* size of M0 file in words */
   WORD     M1Size;     /* size of M1 file in words */
   WORD     GSize;      /* size of G file in words */
   WORD     *GData;     /* pointer to array of length GSize words */
   char     *Name;      /* pointer to module name string */
} OCSLOTCFG;
```

You can specify a module by name or by mix and type. You only specify G data if the module uses G files (such as the 1747-SN). If the `Name` pointer is NULL, OC_CreateIOConfiguration uses `mix` and `type` to identify the module. See page 4 for the `mix` and `type` values. OC_CreateIOConfiguration supplies the `InputSize`, `OutputSize`, `M0Size`, `M1Size`, `Gsize`, and `Name` fields.

If `Name` points to a string containing a valid module name, the module name identifies the module. OC_CreateIOConfiguration supplies the `mix`, `type`, `InputSize`, `OutputSize`, `M0Size`, `M1Size`, and `Gsize` fields.

Initialize empty slots and slot 0 with a `mix` value of 0xFF and a `type` value of 0xFF.

If the module is not in the internal database, OC_CreateIOConfiguration doesn't alter the OCSLOTCFG.

To support modules not included in the internal database of modules, the host application can initialize the `mix`, `type`, `InputSize`, `OutputSize`, `M0Size`, `M1Size`, and `GSize` before downloading the I/O configuration to the scanner. See the I/O module's user manual to determine the proper configuration information.

After the OC_CreateIOConfiguration and OCGetIOConfiguration functions return, the I/O configuration structure must be checked for installed modules with G files. If the `Gsize` field of a non-empty slot configuration is not zero, then the module contains a G file. If the module contains a G file, initialize `GData` to point to an array of `Gsize` words to be loaded into the module during scanner configuration. See the I/O module's user manual to determine the proper G file data.

Using M0-M1 Files and G Files

The 1746 I/O PCI Interface uses M0-M1 files and G files to download configuration information to specialty I/O modules. The following descriptions describe the basics of M0-M1 and G files. For detailed information, see the user manual for the specialty I/O module you are configuring.

M0-M1 files

M0 and M1 files are data files that reside in specialty I/O modules only. There is no image for these files in the dual port memory (like the discrete input and output image files). The application of these files depends on the function of the particular specialty I/O module. Your application program initiates the transfer of these files. Each transfer is a single request or an API call. With respect to the 1746 I/O PCI Interface, the M0 file is a module output file (a write-only file) and the M1 file is a module input file (a read-only file).

You can address M0 and M1 files in your application and they can also be acted upon by the specialty I/O module - independent of the processor scan.

G files

Some specialty modules (such as the 1747-SN) use configuration files, which act as the software equivalent of DIP switches.

The data you enter into the G file is automatically passed to the specialty I/O module when you enter Scan mode.

Supported I/O Modules

Module Name:[a]	Description:	Class:	Mix:[b]	Type:
AMCI-1561		1	35	14
1203-SM1 Class1		1	35	16
1203-SM1 Class 4		4	136	17
1394-SJT		4	136	17
1746-IA4	4-Input 100/120 V ac	0	1	0
1746-IA8	8-Input 100/120 V ac	0	3	0
1746-IA16	16-Input 100/120 V ac	0	5	0
1746-IB8	8-Input (SINK) 24 V dc	0	3	6
1746-IB16	16-Input (SINK) 24 V dc	0	5	6
1746-IB32	32-Input (SINK) 24 V dc	0	7	6
1746-IC16	16-Input dc	0	5	9
1746-IH16	16-Input ac	0	5	7
1746-IG16	16-Input [TTL](SOURCE) 5 V dc	0	5	15
1746-IM4	4-Input 200/240 V ac	0	1	1
1746-IM8	8-Input 200/240 V ac	0	3	1
1746-IM16	16-Input 200/240 V ac	0	5	1
1746-IN16	16-Input 24 V ac/V dc	0	5	10
1746-ITB16	16-Input [FAST](SINK) 24V dc	0	5	19
1746-ITV16	16-Input [FAST](SOURCE) 24V dc	0	5	18
1746-IV8	8-Input (SOURCE) 24 V dc	0	3	20
1746-IV16	16-Input (SOURCE) 24 V dc	0	5	20
1746-IV32	32-Input (SOURCE) 24 V dc	0	7	20
1746-OA8	8-Output(TRIAC) 100/240 V ac	0	27	3
1746-OA16	16-Output(TRIAC) 100/240 V ac	0	29	3
1746-OAP12	Enhanced ac	0	28	3
1746-OB8	8-Output [TRANS](SOURCE)10/50 V dc	0	27	13
1746-OB16	16-Output [TRANS](SOURCE)10/50 V dc	0	29	13
1746-OB16E	16-Output dc	0	29	20
1746-OB32	32-Output [TRANS](SOURCE) 10/50 V dc	0	31	13
1746-OBP8	8-Output dc	0	27	21
1746-OBP16	16-Output [TRANS 1 amp](SRC) 24V dc	0	29	21
1746-OG16	16-Output [TTL](SINK) 5 V dc	0	29	15
1746-OV8	8-Output [TRANS](SINK)10/50 V dc	0	27	14
1746-OV16	16-Output [TRANS](SINK)10/50 V dc	0	29	14
1746-OV32	32-Output [TRANS](SINK) 10/50 V dc	0	31	14
1746-OW4	4-Output [RELAY] V ac/V dc	0	25	0
1746-OW8	8-Output [RELAY] V ac/V dc	0	27	0
1746-OW16	16-Output [RELAY] V ac/V dc	0	29	0
1746-OX8	8-Output [ISOLATED RELAY] V ac/V dc	0	27	1
1746-OVP16	16-Output [TRANS 1 amp] (SINK) 24V dc	0	29	22

a.The module names shown in this table correspond to those used by the OC_GetIOConfiguration and OC_CreateIOConfiguration functions.

b.The mix code for a module is composed of one byte field. The upper 3 bits represent the class of the module, and the lower 5 bits represent the I/O mix of the module.

Module Name:[a]	Description:	Class:	Mix:[b]	Type:
1746-IO4	2-Input 100/120 V ac 2-Output [RLY]	0	8	0
1746-IO8	4-Input 100/120 V ac 4-Output [RLY]	0	11	0
1746-IO12	6-Input 100/120 V ac 6-Output [RLY]	0	15	0
1746-INT4	4 thermocouples, isolated	1	35	15
1746sc-INO4VI	Spectrum Controls, 4 Analog Outputs	1	35	19
1746sc-INI4VI	Spectrum Controls, 4 Analog Inputs	1	35	20
1746sc-INO4I	Spectrum Controls, 4 Analog Outputs	1	35	21
1746sc-INI4I	Spectrum Controls, 4 Analog Inputs	1	35	22
1746-NI4	4 Channel Analog Input	1	44	1
1746-NI8	8 Analog Inputs	1	35	26
1746-NI8	8 Analog Inputs	3	127	26
1746-NIO4I	Analog Comb. 2 In & 2 Current Out	1	32	1
1746-NIO4V	Analog Comb. 2 In & 2 Voltage Out	1	32	2
1746-FIO4I	Fast Analog Comb 2 In & 2 Current Out	1	32	24
1746-FIO4V	Fast Analog Comb 2 In & 2 Voltage Out	1	32	18
1746-NO4I	4 Channel Analog Current Output	1	54	1
1746-NO4V	4 Channel Analog Voltage Output	1	54	2
1746-NT4	4 Channel Thermocouple Input Module	1	35	10
1746sc-NT8	Spectrum Controls, 4 Analog inputs isolated	1	35	33
1746-NR4	4 Channel RTD/Resistance Input Module	1	35	13
1746-HSCE	High Speed Counter/Encoder	3	127	5
1746-HS	Single Axis Motion Controller	1	33	3
1746-HSRV	SLC Servo Single AX MC	3	101	14
1746-HSTP1	Stepper Controller Module	1	35	12
1746-BAS1[c]	BASIC Module - 5/01 Configuration	1	35	6
1746-BAS2[c]	BASIC Module - 5/02 Configuration	4	131	6
1746-QS	Synchronized Axes	4	136	27
1746-QV	Open Loop Velocity	4	131	15
1747-DCM1[c]	Direct Commun. Module (1/4 RACK)	1	32	25
1747-DCM2[c]	Direct Commun. Module (1/2 RACK)	1	33	25
1747-DCM3[c]	Direct Commun. Module (3/4 RACK)	1	34	25
1747-DCM4[c]	Direct Commun. Module (FULL RACK)	1	35	25
1747-MNET	Module Interface	4	158	11
1747-SDN	DeviceNet Scanner	4	136	6
1747-SN	Remote I/O Scanner	4	136	8
1747-DSN1[c]	Distributed I/O Scanner - 7 Blks	1	35	7
1747-DSN2[c]	Distributed I/O Scanner - 30 Blks	4	136	7
1747-KE1[c]	Interface Module, Series A	1	42	9
1747-KE2[c]	Interface Module, Series B	1	35	9

a.The module names shown in this table correspond to those used by the OC_GetIOConfiguration and OC_CreateIOConfiguration functions.

b.The mix code for a module is composed of one byte field. The upper 3 bits represent the class of the module, and the lower 5 bits represent the I/O mix of the module.

c.Some modules can have multiple configurations. To distinguish between different configurations of the same module, a single digit is appended to the module name.

Notes:

Library of Routines

Introduction

The MS-DOS API is a run-time library that can be linked with most industry standard programming language compilers using the Pascal calling convention.

The Windows NT API is a 32-bit DLL that can be linked with most industry-standard programming language compilers.

This chapter provides the programming information for each routine and identifies which operating system supports the routine. The calling convention for each API function is shown in C format.

OC_CalculateCRC

OC_CalculateCRC calculates a 16-bit CRC.

Syntax:

```
void   OC_CalculateCRC(BYTE *bufPtr, WORD bLen, WORD *Crc);
```

Parameters:

Parameter:	Description:
bufPtr	Points to the buffer that contains the bytes for the CRC calculation
bLen	Number of bytes for which to calculate the CRC
Crc	A word that returns the calculated CRC

Description:

This function is useful for verifying data integrity. For example, a CRC might be appended to data stored in the host retentive data partition. When the data is later retrieved, a new CRC can be calculated and compared to the old CRC to ensure the data is valid.

Return Value:

none

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
BYTE   buffer[100];
WORD   buffer_crc;
int    retcode;
   retcode = OC_CalculateCRC( buffer, 100, &buffer_crc );
```

OC_ClearFault

OC_ClearFault clears any fault condition of the scanner.

Syntax:

```
int    OC_ClearFault(HANDLE handle);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner

Description:

All extended error information must be retrieved before the fault can be cleared.

If the scanner encounters an error condition, it generates an extended error and faults. The fault forces the scanner into Idle mode. The scanner cannot be placed into Scan mode until the fault is cleared.

Return Value:

Name:	Value:	Description:
SUCCESS	0	fault was cleared successfully
ERR_OCACCESS	2	handle does not have access to the scanner
ERR_OCEXTERR	11	scanner extended error message, see OC_GetExtendedError
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner
ERR_OCRESPONSE	10	scanner did not respond to request

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
int       retcode;
   retcode = OC_ClearFault( Handle );
```

OC_CloseScanner

This function must always be called before exiting the application.

Syntax:

```
int    OC_CloseScanner(HANDLE handle);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner

Description:

This function releases control of the scanner device, releases the interrupt assigned by OC_OpenScanner, and disables the segment address assignment.

ATTENTION: The system might become unstable if you don't call OC_CloseScanner before exiting the application.

Return Value:

Name:	Value:	Description:
SUCCESS	0	scanner was closed successfully
ERR_OCACCESS	2	handle does not have access to the scanner

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
int       retcode;
   retcode = OC_CloseScanner( Handle );
```

OC_ConfigureDII

OC_ConfigureDII allows an application to receive a message from the scanner when an input bit pattern of a discrete I/O module matches a compare value.

Syntax:

```
int    OC_ConfigureDII(HANDLE handle, DII_CFG *diicfg);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
diicfg	Points to the DII configuration

Description:

The application configures the compare value using this function and when the comparison completes, the scanner generates a message to the application. The application must then call OC_PollScanner to retrieve the message.

The DII_CFG structure is defined as:

```
typedef struct {
   BYTE   SlotNum;  /* slot number 1-30*/
   BYTE   IOIncludeMask;/* bits allowed mask */
   BYTE   IOEdgeType;/* bit pattern to compare */
   WORD   PresetCount;/* number of matches */
} DII_CFG;
```

This value:	Means:
Slotnum	Must contain the slot number of a Class 0 Discrete Input module. An I/O error report is generated if the scanner determines the slot does not contain a valid discrete input module.
IOIncludeMask	Should contain the bits in the discrete input module to include in the comparison. Only bits 0 - 7 of word 0 of the module can be configured for DII's. IOIncludeMask is a bit-mapped mask. Any bit set to 1 in this mask includes the corresponding bit of the discrete input module in the comparison. Any bit set to 0 is ignored.
IOEdgeType	Defines the bit pattern to compare. Only bits that correspond to bits set to 1 in IOIncludeMask are used. Only bits 0 - 7 are valid. IOEdgeType is a bit-mapped value. If a bit is set to 1, the comparison for the bit matches when its corresponding discrete input bit changes from 0 to 1. If a bit is set to 0, the comparison for the bit matches when its corresponding discrete input bit changes from 1 to 0.
PresetCount	When PresetCount is 0 or 1, the scanner generates a message each time the comparison matches. When it is between 2 and 65535, the message is generated when the number of comparison matches meets PresetCount.

The scanner recognizes a match when every bit in the IOIncludeMask has finished transitioning. After a message is generated, another message will be generated as soon as the next specified number of matches occurs.

To disable DII's, set IOIncludeMask to 0 with a valid SlotNum. DII's are disabled by default on reset.

Return Value:

Name:	Value:	Description:
SUCCESS	0	discrete input interrupt (DII) was configured successfully
ERR_OCACCESS	2	`handle` does not have access to the scanner
ERR_OCRESPONSE	10	scanner did not respond to request
ERR_OCSCANCFG	14	scanner has not been configured
ERR_OCSLOT	12	slot number is invalid

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
DII_CFG   diicfg;
int       retcode;
   diicfg.Slotnum = 6;/* Slot 6 has discrete input module */
   diicfg.IOIncludeMask = 1;/* bit 0 is the input trigger */
   diicfg.IOEdgeType = 1;/* bit 0 must toggle from low to high */
   diicfg.PresetCount = 3;/* bit 0 must toggle 3 times */
   retcode = OC_ConfigureDII( Handle, &diicfg );

   /* Use OC_PollScanner() to check for DII messages */
```

OC_CreateIO Configuration

OC_CreateIOConfiguration creates a scanner configuration from an application-specific installation of rack sizes and installed modules. See chapter 5 for more information.

Syntax:

```
int    OC_CreateIOConfiguration(OCIOCFG *iocfg);
```

Parameters:

Parameter:	Description:
iocfg	Specifies the rack sizes and installed modules

Description:

Modules can be specified by name or by mix and type. The function automatically fills in the rest of the required information in the OCIOCFG structure.

This function returns in iocfg the scanner configuration information obtained from the rack sizes and installed module types specified in iocfg. The scanner configuration can then be downloaded to the scanner with OC_DownloadIOConfiguration, which allows the application to control the number of racks and their sizes and the position and type of modules installed in the racks.

The OCIOCFG structure is defined as:

```
typedef struct tagOCIOCFG
{
   BYTE   Rack1Size;    /* number of slots in Rack 1 */
   BYTE   Rack2Size;    /* number of slots in Rack 2 */
   BYTE   Rack3Size;    /* number of slots in Rack 3 */
   OCSLOTCFG SlotCfg[OCMAXSLOT];/* configuration for each slot */
} OCIOCFG;
```

Return Value:

Name:	Value:	Description:
SUCCESS	0	I/O configuration was read successfully
ERR_OCUNKNOWN	18	at least one module was not found in the internal database The SlotCfg data for the unknown module is not altered; the remaining modules are configured.

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
OCIOCFG   iocfg;
int       retcode, numslots, i;
char      module_name[] = ”1746-BAS”;

   iocfg.Rack1Size = 10;                  /* 10 slot chassis */
   iocfg.Rack2Size = 7;                   /* 7 slot chassis */
   iocfg.Rack3Size = 0;                   /* Only 2 chassis */

   numslots = iocfg.Rack1Size + iocfg.Rack2Size + iocfg.Rack3Size;

   for ( i=1; i<numslots; i++ ){
      iocfg.SlotCfg[i].mix = OCEMPTYMIX;
      iocfg.SlotCfg[i].type = OCEMPTYTYPE; /* Empty all slots */
   }
   iocfg.SlotCfg[6].mix = 35;
   iocfg.SlotCfg[6].type = 6;             /* Slot 6 has 1746-BAS module */
      or
   iocfg.SlotCfg[6].name = module_name;   /* Use name instead */
      .
      .         /* Add additional module information to */
      .         /* match the physical I/O configuration */
      .

   retcode = OC_CreateIOConfiguration( &iocfg );

   /* Use OC_DownloadIOConfiguration() to download the information */
```

OC_DemandInputScan

OC_DemandInputScan forces the scanner to immediately perform an input scan.

Syntax:

```
int    OC_DemandInputScan(HANDLE handle, int mode);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
mode	If mode is: OCWAIT OC_DemandInputScan waits for the input scan to be completed before returning to the caller. OCNOWAITOC_DemandInputScan returns immediately.

Description:

If an I/O scan is in progress when this function is called, the input scan is performed after the current scan has completed.

The scanner updates the input image with data read from the modules. Use OC_ReadInputImage to read data from the input image.

Return Value:

Name:	Value:	Description:
SUCCESS	0	demand input request was successful
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCRESPONSE	10	scanner did not respond to request
ERR_OCSCANCFG	14	scanner has not been configured

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
int       retcode;
   retcode = OC_DemandInputScan( Handle, OCWAIT );
```

OC_DemandOutputScan

OC_DemandOutputScan forces the scanner to immediately perform an output scan.

Syntax:

```
int    OC_DemandOutputScan(HANDLE handle, int mode);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
mode	If mode is: OCWAIT OC_DemandOutputScan waits for the output scan to be completed before returning to the caller. OCNOWAITOC_DemandOutputScan returns immediately.

Description:

The scanner updates module outputs from the data in the output image. Use OC_WriteOutputImage to write data to the output image.

Return Value:

Name:	Value:	Description:
SUCCESS	0	demand output request was successful
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCRESPONSE	10	scanner did not respond to request
ERR_OCSCANCFG	14	scanner has not been configured

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
int       retcode;
   retcode = OC_DemandOutputScan( Handle, OCWAIT );
```

OC_DownloadIO Configuration

OC_DownloadIOConfiguration downloads an I/O configuration to the scanner.

Syntax:

```
int    OC_DownloadIOConfiguration(HANDLE handle, OCIOCFG *iocfg);
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`iocfg`	Specifies the rack sizes and installed modules

Description:

The scanner must be in Idle mode to receive an I/O configuration. This function forces the scanner to Idle mode to download the configuration.

The scanner checks the downloaded I/O configuration for validity, and if there are any errors, an extended error might be generated. If an error is generated, the scanner will fault.

The `OCIOCFG` structure is defined as:

```
typedef struct tagOCIOCFG
{
   BYTE      Rack1Size;    /* number of slots in Rack 1 */
   BYTE      Rack2Size;    /* number of slots in Rack 2 */
   BYTE      Rack3Size;    /* number of slots in Rack 3 */
   OCSLOTCFG SlotCfg[OCMAXSLOT];/* configuration for each slot */
} OCIOCFG;
```

Return Value:

Name:	Value:	Description:
SUCCESS	0	I/O configuration was downloaded successfully
ERR_OCACCESS	2	`handle` does not have access to scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner
ERR_OCOUTOFMEM	17	unable to allocate memory for configuration data
ERR_OCRESPONSE	10	scanner did not respond to request

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
OCIOCFG   iocfg;
int       retcode;
   /* Either OC_CreateIOConfiguration() or OC_GetIOConfiguration() were */
   /* called previously to fill in 'iocfg' structure */
   retcode = OC_DownloadIOConfiguration( Handle, &iocfg );
```

OC_EnableEOSNotify

OC_EnableEOSNotify controls whether end-of-scan notification messages are generated by the scanner.

Syntax:

```
int    OC_EnableEOSNotify(HANDLE handle, int mode);
```

Parameters:

Parameter:	Description:	
handle	Must be a valid handle returned from OC_OpenScanner	
mode	If mode is: EOSMSG_ENABLE message function EOSMSG_DISABLE scan disabled	the scanner generates an end-of-scan after each scan. Use the OC_PollScanner to retrieve end-of-scan messages. the scanner does not generate End-of- messages. End-of-scan messages are after the scanner has been reset.

Description:

There are three types of end-of-scan messages:

This type of message:	Is generated after:
OCMSG_EOS_DMDIN	a OC_DemandInputScan command has completed
OCMSG_EOS_DMDOUT	a OC_DemandOutputScan command has completed
OCMSG_EOS	each timed I/O scan

End-of-scan messages are generated from the scanner to the API via interrupts after each scan. The scan rate is controlled by the OC_SetScanTime function and end-of-scan interrupts are generated at the scan rate. Enabling end-of-scan messages can affect the performance of the application due to the overhead incurred in processing these interrupts. An alternative method to synchronize with the scanner's I/O scan is to monitor the scanner watchdog register, which is incremented at the end of each timed I/O scan. See OC_GetScannerWatchdogCount.

Return Value:

Name:	Value:	Description:
SUCCESS	0	notification was generated successfully
ERR_OCACCESS	2	`handle` does not have access to scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner
ERR_OCPARAM	8	parameter contains invalid value

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE Handle;
int       retcode;
   retcode = OC_EnableEOSNotify( Handle, EOSMSG_ENABLE );
   /* Use OC_PollScanner() to check EOS messages */
```

OC_EnableForces

OC_EnableForces enables/disables forces for all inputs and outputs with entries in the force files on the scanner.

ATTENTION: Enabling forces will potentially change the output data values that your application was previously controlling.

Syntax:

```
int    OC_EnableForces(HANDLE handle, int mode);
```

Parameters:

Parameter:	Description:	
handle	Must be a valid handle returned from OC_OpenScanner	
mode	If mode is: FORCE_ENABLE FORCE_DISABLE FORCE_CLEAR output forces	 forces are enabled forces are disabled forces are disabled and all input and are cleared from the force files.

Description:

If no I/O forces are in the force files, OC_EnableForces does not enable forces and instead returns an error. All forces are disabled by default.

Return Value:

Name:	Value:	Description:
SUCCESS	0	forces were updated successfully
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCNOFORCES	15	no forces installed, scanner cannot enable forces
ERR_OCPARAM	8	parameter contains invalid value
ERR_OCRESPONSE	10	scanner did not respond to request
ERR_OCSCANCFG	14	scanner has not been configured

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
int       retcode;

   /* Use OC_SetForces() to configure forcing information first */
   retcode = OC_EnableForces( Handle, FORCE_ENABLE );
```

OC_EnableSlot

OC_EnableSlot enables fine tuning of the I/O scanning process.

Syntax:

```
int    OC_EnableSlot(HANDLE handle, int slotnum, int state);
```

Parameters:

Parameter:	Description:	
handle	Must be a valid handle returned from OC_OpenScanner	
slotnum	Must contain a valid slot number.	
state	If state is:	
	SLOT_ENABLE	the module is released from its reset state and is included in the I/O scan
	SLOT_DISABLE	the module is no longer included in the I/O scan and any outputs remain at their last state

Description:

This function enables or disables the scanner from scanning the module in a specific slotnum. This applies to both the input and output scan. Slots that are disabled are not included in the I/O scan. By default, all slots are enabled.

Return Value:

Name:	Value:	Description:
SUCCESS	0	module was updated successfully
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCNOFORCES	15	no forces installed, scanner cannot enable forces
ERR_OCPARAM	8	parameter contains invalid value
ERR_OCRESPONSE	10	scanner did not respond to request
ERR_OCSCANCFG	14	scanner has not been configured

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
int       retcode;
   retcode = OC_EnableSlot( Handle, 6, SLOT_DISABLE );    /* Exclude slot 6 */
```

OC_ErrorMsg

OC_ErrorMsg returns a descriptive text message associated with the API return value `errcode`.

Syntax:

```
int OC_ErrorMsg(int errcode, char **msg);
```

Description:

The null-terminated message string is placed in a static buffer that is reused each time this function is called. A pointer to this buffer is returned in `msg`.

Return Value:

Name:	Description:
SUCCESS	`errcode` was valid. `msg` points to corresponding error description.
ERR_OCPARAM	`errcode` was invalid. `msg` points to unknown error code string.

Considerations:

Supported in the DOS API library and the Windows NT API library.

Example:

```
HANDLE    Handle;
char      *msg;
int       rc;

if (SUCCESS != (rc = OC_OpenScanner(&Handle)))
{
   /* Open failed - display error message */
   OCErrorMsg(rc, &msg);
   printf("Error: %s\n", msg);
}
```

OC_ExtendedErrorMsg

OC_ExtendedErrorMsg returns a descriptive text message associated with an extended error.

Syntax:

```
int    OC_ExtendedErrorMsg(HANDLE handle, OCEXTERR *exterr, char **msg);
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`exterr`	Points to an extended error
`msg`	Points to a static buffer that contains a null-terminated message string for the associated extended error

Description:

This function is useful when displaying an error message. You should use OC_GetExtendedError to obtain the message before using OC_ExtendedErrorMsg to display the message. If you don't use OC_GetExtendedError first, OC_ExtendedErrorMsg displays a null message.

The `OCEXTERR` structure is defined as:

```
#define OCERRDATASIZE  3           /* number of bytes of error data */
typedef struct tagOCEXTERR
{
   BYTE   ErrorCode;               /* Extended error code       */
   BYTE   SlotNum;                 /* Associated slot number */
   BYTE   ErrorData[OCERRDATASIZE]; /* Error code data   */
} OCEXTERR;
```

See appendix A for error codes.

Return Value:

Name:	Value:	Description:
SUCCESS	0	extended error information was read successfully
ERR_OCACCESS	2	`handle` does not have access to scanner

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
OCEXTERR  exterr;
char      *msg;
int       retcode;
   /* Should already have called OC_GetExtendedError() to obtain exterr */
   retcode = OC_ExtendedErrorMsg( Handle, &exterr, &msg );
   printf("ERROR:%s\n", msg);
```

OC_GetBatteryStatus

OC_GetBatteryStatus gets the current state of the battery of the scanner.

Syntax:

```
int    OC_GetBatteryStatus(HANDLE handle, BYTE *batstate);
```

Parameters:

Parameter:	Description:	
handle	Must be a valid handle returned from OC_OpenScanner	
batstate	If batstate is:	
	BATTERY_GOOD	battery voltage is good
	BATTERY_LOW	battery voltage has dropped below a reliable level

Description:

The battery provides backup power for the host retentive data (dual port RAM).

Return Value:

Name:	Value:	Description:
SUCCESS	0	battery state was read successfully
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
BYTE      batt_sts;
int       retcode;
   retcode = OC_GetBatteryStatus( Handle, &batt_sts );
```

OC_GetDeviceInfo OC_GeDeviceInfo returns information about the scanner device.

Syntax:

```
int OC_GetDeviceInfo(HANDLE handle, OCDEVICEINFO *devinfo);
```

Description:

The OCDEVICEINFO is defined as:

```
{
   WORD      ScannerType;        /* scanner device type */
   WORD      ScannerIrq;         /* allocated interrupt */
   WORD      ScannerMemory;      /* dual-port memory access */
   WORD      ControlIo;          /* PCIS control registers address */
   WORD      SRAM_Size           /* size of available SRAM in bytes */
} OCDEVICEINFO;
```

`handle` must be a valid handle returned from OC_OpenScanner.

Return Value:

Name:	Description:
SUCCESS	The extended error information was read successfully.
ERR_OCACCESS	`handle` does not have access to scanner

Considerations:

Supported in the DOS API library and the Windows NT API library.

Description:

```
HANDLE    Handle;
OCDEVICEINFO     devinfo;

/* display size of available SRAM */
OC_GetDeviceInfo(Handle, &devinfo);
printf("SRAM Size is %ld bytes\n", devinfo.SRAM_Size);
```

OC_GetExtendedError

OC_GetExtendedError reads extended error information from the scanner.

Syntax:

```
int    OC_GetExtendedError(HANDLE handle, OCEXTERR *buf);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
buf	Contains the extended error information If no extended error information is available, the error code field of buf will be 0.

Description:

The extended error information is written during Scan mode or its configuration. An API function that determines that the scanner has responded with an error returns an error code of ERR_OCEXTERR. OC_GetExtendedError retrieves the extended error information written by the scanner and removes the error from the scanner.

The library buffers extended errors in a queue. The queue can hold as many as 5 extended errors at one time. If the queue is full when a new extended error is received from the scanner, the oldest extended error is lost and ERR_OCOVERRUN is returned. The host application must call this function periodically to remove existing extended errors from the buffer.

The OCEXTERR structure is defined as:

```
#define OCERRDATASIZE  3     /* number of bytes of error data */
typedef struct tagOCEXTERR
{
   BYTE   ErrorCode;               /* Extended error code       */
   BYTE   SlotNum;                 /* Associated slot number */
   BYTE   ErrorData[OCERRDATASIZE]; /* Error code data   */
} OCEXTERR;
```

See appendix A for error codes.

Return Value:

Name:	Value:	Description:
SUCCESS	0	extended error information was read successfully
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCOVERRUN	16	an error message has been discarded

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE Handle;
OCEXTERRexterr;
int       retcode;
   retcode = OC_GetExtendedError( Handle, &exterr );
```

OC_GetInputImage UpdateCounter

OC_GetInputImageUpdateCounter reads the value of the input image update counter from the scanner and places it into `count`.

Syntax:

```
int    OC_GetInputImageUpdateCounter(HANDLE handle, BYTE *count);
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`count`	Contains the value of the input image update counter

Description:

The input image update counter is incremented by the scanner after each input scan.

The input image update counter is only incremented if the scanner is in Scan mode, input scans are enabled, and inputs are present. Use the counter to determine whether a change occurred; the value of the counter is not important. It is possible to configure a system with no inputs; in this case, the input image update counter would not be incremented.

Return Value:

Name:	Value:	Description:
SUCCESS	0	input image update counter was read successfully
ERR_OCACCESS	2	`handle` does not have access to scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
BYTE      count;
int       retcode;
   retcode = OC_GetInputImageUpdateCounter( Handle, &count );
```

OC_GetIOConfiguration

OC_GetIOConfiguration queries the I/O rack about the installed rack sizes and I/O modules in each 1746 chassis.

ATTENTION: OC_GetIOConfiguration can take several milliseconds to complete, depending upon the rack configuration. While it is executing, I/O scanning and DII's are disabled.

Syntax:

```
int    OC_GetIOConfiguration(HANDLE handle, OCIOCFG *iocfg);
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`iocfg`	Specifies the rack sizes and installed modules Use the information in `iocfg` as input to OC_DownloadIOConfiguration to configure the scanner.

Description:

If the scanner is in Scan mode and OC_GetIOConfiguration returns successfully, OC_GetIOConfiguration enables the host application to access I/O. The scanner must have previously received a valid configuration prior to going to Scan mode.

The `OCIOCFG` structure is defined as:

```
typedef struct tagOCIOCFG
{
   BYTE      Rack1Size;    /* number of slots in Rack 1 */
   BYTE      Rack2Size;    /* number of slots in Rack 2 */
   BYTE      Rack3Size;    /* number of slots in Rack 3 */
   OCSLOTCFG SlotCfg[OCMAXSLOT];/* configuration for each slot */
} OCIOCFG;
```

Return Value:

Name:	Value:	Description:
SUCCESS	0	I/O configuration was read successfully
ERR_OCACCESS	2	`handle` does not have access to scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner
ERR_OCRESPONSE	10	scanner did not respond to request

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
OCIOCFG   iocfg;
int       retcode;
   retcode = OC_GetIOConfiguration( Handle, &iocfg );
   /* Use OC_DownloadIOConfiguration() to download the information */
```

OC_GetLastFaultCause

OC_GetLastFaultCause retrieves the cause of the last fault.

Syntax:

```
int    OC_GetLastFaultCause(HANDLE handle, BYTE *FaultCode, int *SlotNum);
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`FaultCode`	Points to the address that contains the fault cause If the value returned in `FaultCode` is 0, the scanner has not received any faults since it has been reset.
`SlotNum`	Slot number that caused the fault

Description:

When the scanner faults, an extended error is generated. The error code and slot number of the most recent fault is retained and returned by this function. The fault cause is a duplicate of the most recent extended error.

The OC_ClearFault function clears the fault in the scanner but does not clear the cause of the last fault.

See Appendix A for error codes.

Return Value:

Name:	Value:	Description:
SUCCESS	0	fault was cleared successfully
ERR_OCACCESS	2	`handle` does not have access to scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
BYTE      status, FaultCause;
int       FaultSlot;
int       retcode;
   OC_GetScannerStatus ( Handle, &status );
   if ( status = SCANSTS_FAULT )
   {
      retcode = OC_GetLastFaultCause ( Handle, &FaultCause, &FaultSlot );
   }
```

OC_GetMeasuredScan Time

OC_GetMeasuredScanTime returns the maximum and last observed I/O scan times.

Syntax:

```
int    OC_GetMeasuredScanTime(HANDLE Handle, WORD *maxtime, WORD *lasttime);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
maxtime	Returns the maximum scan time
lasttime	Returns the last scan time

Description:

The scanner calculates these values at the end of each I/O scan. The values are represented in units of 250 microseconds.

The scan times are reset to zero when changing to Scan mode, and are not valid until the end of the second I/O scan. Only the timed I/O scans are measured; the demand input or output scans are not.

Return Value:

Name:	Value:	Description:
SUCCESS	0	measured scan time was read successfully
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
WORD      max_time, last_time;
int       retcode;
   retcode = OC_GetMeasuredScanTime( Handle, &max_time, &last_time );
```

OC_GetScannerInitInfo This function retrieves current information about the shared memory partitioning.

Syntax:

```
int    OC_GetScannerInitInfo(HANDLE handle, OCINIT *scaninit);
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`scaninit`	Points to the structure that contains the initialization information this function returns

Description:

If the scanner has not been initialized, OC_GetScannerInitInfo returns an error.

If the scanner has been previously initialized, an application can retrieve the current scanner partitioning information with this function instead of resetting and re-initializing the scanner.

The `OCINIT` structure us defined as:

```
typedef struct tagOCINIT
{
   WORD   OutputImageSize;       /* size in bytes */
   WORD   InputImageSize;        /* size in bytes */
   WORD   HostRetentiveDataSize; /* size in bytes */
} OCINIT;
```

Return Value:

Name:	Value:	Description:
SUCCESS	0	scanner initialization information was retrieved successfully
ERR_OCACCESS	2	`handle` does not have access to the scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner
ERR_OCPOST	7	scanner POST failed, see OC_GetScannerStatus

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
OCINIT    scaninit;
int       retcode;

   retcode = OC_GetScannerInitInfo( Handle, &scaninit );
   if ( retcode == SUCCESS )
   {
      printf( ”Input Image Size = %d bytes \n”, scaninit.InputImageSize );
      printf( ”Output Image Size = %d bytes \n”, scaninit.OutputImageSize );
      printf( ”Host Retentive Data Size = %d bytes \n”,
      scaninit.HostRetentiveDataSize );
   }
   else
      /* handle error */
```

OC_GetScannerStatus

OC_GetScannerStatus reads the current status of the scanner.

Syntax:

```
int    OC_GetScannerStatus(HANDLE handle, BYTE *scansts);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
scansts	Status of the scanner

Description:

If OC_GetScannerStatus returns SUCCESS, scansts has one of these values:

This value:	Has this hex value:	Means the:
SCANSTS_BPIC	4	POST backplane IC test failed; scanner is halted
SCANSTS_CRC	2	software CRC checksum failed
SCANSTS_DPR	5	POST dual port RAM test failed; scanner is halted
SCANSTS_FAULT	13	scanner faulted; scanner is in Scan mode
SCANSTS_IDLE	11	scanner initialized; scanner is in Idle mode
SCANSTS_INIT	10	POST passed; waiting for OC_InitScanner from host
SCANSTS_INT	8	POST interrupt test failed; scanner is halted
SCANSTS_POST	1	power-on self test (POST) is in progress
SCANSTS_RAM	3	POST RAM test failed; scanner is halted
SCANSTS_SCAN	20	scanner initialized; scanner in Scan mode
SCANSTS_THERM	6	POST thermometer test failed; scanner is halted
SCANASTS_TIMER	7	POST timer test failed; scanner is halted
SCANSTS_WDOG	12	scanner watchdog timeout; scanner is halted

Return Value:

Name:	Value:	Description:
SUCCESS	0	scanner status was read successfully
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCEXTERR	11	scanner extended error message (scansts is returned)

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
BYTE      scansts;
int       retcode;
   retcode = OC_GetScannerStatus( Handle, &scansts );
```

OC_GetScanner WatchdogCount

OC_GetScannerWatchdogCount reads the contents of the watchdog register of the scanner.

Syntax:

```
int    OC_GetScannerWatchdogCount(HANDLE handle, BYTE *count);
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`count`	Returns the watchdog register contents

Description:

The watchdog register is incremented by the scanner after every timed I/O scan.

This register is incremented in both Scan and Idle modes, and is incremented even if both output and input scans are disabled. The control application can monitor this register to ensure that the scanner is functioning normally. It is also useful for synchronizing with the I/O scan.

Return Value:

Name:	Value:	Description:
SUCCESS	0	watchdog was read successfully
ERR_OCACCESS	2	`handle` does not have access to scanner

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
BYTE      wdog_count;
int       retcode;
   retcode = OC_GetScannerWatchdogCount( Handle, &wdog_count );
```

OC_GetStatusFile

OC_GetStatusFile reads a copy of the current scanner system status file into the STSFILE structure pointed to by stsfil on the scanner.

Syntax:

```
int    OC_GetStatusFile(HANDLE handle, STSFILE *stsfil);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
stsfil	Points to the STSFILE structure that contains scanner system status

Description:

The status file is updated by the scanner at the end of each I/O scan.

The STSFILE structure is defined as:

```
typedef struct tagSTSFILE
{
   WORD   wWordNum[OCSTSFILEWSIZE];
} STSFILE;
```

The status file is organized by words. The status file uses these classifications to define the data each word contains:

This classification:	Means the data:
status	is used primarily to monitor scanner options or status. This information is usually not written by the application, except to clear a function such as a minor error bit.
dynamic configuration	can be written by application to select scanner options while in Scan mode.

The status file contains:

Word/Bit:	Classification:	Description:
0/0 to 0/4	status	Scanner mode/status **bit 4 bit 3 bit 2bit 1bit 0** 1 0 000= (16) download in progress 1 0 001= (17) Idle mode (program) 1 1 110= (30) Scan mode (run) All other values for bits 0-4 are reserved.
0/5	status	Forces enabled bit This bit is set if forces have been enabled.

Word/Bit:	Classification:	Description:
0/6	status	Forces installed bit This bit is set is forces have been installed.
0/7 to 0/12	reserved	
0/13	dynamic configuration	Major error halted bit This bit is set by the scanner when a major error is encountered. The scanner enters a fault condition. Word 2, Fault Code will contain a code which can be used to diagnose the fault condition. When bit 0/13 is set, the scanner places all outputs in a safe state and sets the Status LED to the fault state (flashing red). Once a major fault state exists, the condition must be corrected and bit 0/13 cleared before the scanner will accept a mode change request.
0/14	reserved	
0/15	status	First pass bit The bit is set by the scanner to indicate that the first I/O scan following entry into Scan mode is in progress. The scanner clears this bit following the first scan.
1/0 to 1/10	reserved	
1/11	status	Battery low bit This bit is set by the scanner when the Battery Low LED is on. It is cleared when the Battery Low LED is off.
1/12	status	DII overflow bit This bit is set by the scanner when a DII interrupt occurs and the scanner is unable to successfully transmit the DII Received priority message to the host.
1/13 to 1/15	reserved	
2	status	Major error fault code A code is written to this word by the scanner when a major error occurs. See word S:0/13. The code defines the type of fault. If not zero, the upper byte indicates the slot associated with the error. This word is not cleared by the scanner.
3 to 4	dynamic configuration	I/O slot enables These two words are bit mapped to represent the 30 possible I/O slots in an SLC 500 system. Bits 3/0 through 4/14 represent slots 0-30 (slot 0 is reserved for the 1746 I/O PCI Interface). Bit 4/15 is unused. When a bit is set (default condition), it allows the I/O module in the referenced slot to be updated in the I/O scan. When a bit is cleared, the corresponding I/O module will no longer be included in the I/O scan. Changes to the I/O slot enable bits will take affect at the end of the next I/O scan.
5	status	Maximum observed scan time This word indicates the maximum observed interval between consecutive I/O scans. The interval time is reported in units of 250 ms. Resolution of the observed scan time is +0 to -250 ms. For example, the value 10 indicates that 2.25-2.5 ms was the longest scan time.
6	dynamic configuration	Index register This word indicates the element offset used in indexed addressing.
7 to 8	status	I/O interrupt pending These two words are bit-mapped to the 30 I/O slots. Bits 7/1 through 8/14 refer to slots 1-30. Bits 7/0 and 8/15 are not used. The pending bit associated with a slot is set when an interrupt request is received from that slot. This bit is set regardless of the state of the I/O interrupt enabled bit (wee word 9 and 10).

Word/Bit:	Classification:	Description:
9 to 10	status	I/O interrupt enabled These two words are bit-mapped to the 30 I/O slots. Bits 9/1 through 10/14 refer to slots 1-30. Bits 9/0 and 10/15 are not used. The corresponding enable bit must be set in order for an I/O interrupt received priority message to be generated when a module issues an interrupt request.
11/0 to 11/8	reserved	
11/9	status	I/O scan toggle bit This bit is cleared upon entry into Scan mode and is toggled (changes state) at the end of every I/O scan.
11/10	dynamic configuration	DII reconfiguration bit If the bit is set by the host, the DII function will reconfigure itself at the end of the next I/O scan.
11/11 to 11/15	reserved	
12	status	Last I/O scan time This word indicates the current observed interval between consecutive I/O scans. The interval time is reported in units of 250 ms. Resolution of the last scan time is +0 to -250 ms. For example, the value 10 indicates that 2.25-2.5 ms was the last scan time.
13	dynamic configuration	DII function enable A value of zero written to this word will disable the discrete input interrupt function. Any non-zero value will enable the function.
14	dynamic configuration	DII slot number This word is used to configure the DII function. The slot number (1-30) that contains the discrete I/O module should be written to this word. The scanner will fault if the slot is empty or contains a non-discrete I/O module. This word is applied upon detection of the DII reconfigure bit 11/10 or upon entry to Scan mode.
15	dynamic configuration	DII bit mask This word contains a bit-mapped value that corresponds to the bits to monitor on the discrete I/O module. Only bits 0-7 are used in the DII function. Setting a bit indicates that it is to be included in the comparison of the discrete I/O module's bit transition to the DII compare value (word 16). Clearing a bit indicates that the transition state of that bit is a "don't care." This word is applied upon detection of the DII reconfigure bit 11/10 and at the end of each I/O scan.
16	dynamic configuration	DII compare value This word contains a bit-mapped value that corresponds to the bit transitions that must occur in the discrete I/O module for a count or interrupt to occur. Only bits 0-7 are used in the DII function. Setting a bit indicates that the bit must transition from a 0 to a 1 to satisfy the compare condition for that bit. Clearing a bit indicates that the bit must transition from a 1 to a 0 in order to satisfy the compare condition for that bit. An interrupt or count will be generated upon the last bit transition of the compare value. This word is applied upon detection of the DII reconfigure bit 11/10 and at the end of each I/O scan.

Word/Bit:	Classification:	Description:
17	dynamic configuration	DII preset When this value is 0 or 1, an interrupt is generated each time the bit transition comparison is satisfied (see words 15 and 16). When this value is 2-65535, a count occurs each time the bit transition comparison is satisfied. When the total number of counts equals the DII preset value, an interrupt will be generated. This word is applied upon detection of the DII reconfigure bit 11/10 and at the end of each I/O scan.
18	status	DII accumulator The DII accumulator contains the number of count transitions that have occurred (see word 17). When a count occurs and the accumulator is greater than or equal to the preset value, a DII interrupt is generated.
19	status	Scanner firmware series This word indicates the scanner firmware series number. The series and revision numbers are used to identify versions of firmware.
20	status	Scanner firmware revision This word indicates the scanner firmware revision number. The series and revision numbers are used to identify versions of firmware.
21	status	1746 I/O PCI Interface hardware series This word indicates the 1746 I/O PCI Interface hardware series number. The series and revision numbers are used to identify versions of firmware.
22	status	1746 I/O PCI Interface hardware revision This word indicates the 1746 I/O PCI Interface hardware revision number. The series and revision numbers are used to identify versions of firmware.
23	status	Scanner RAM size This word indicates the size of RAM in 16-bit K words. For example, a value of 64 indicates 64K words, or 128K bytes.
24	status	Scanner flash ROM size This word indicates the size of flash ROM in 16-bit K words. For example, a value of 64 indicates 64K words, or 128K bytes.

Return Value:

Name:	Value:	Description:
SUCCESS	0	system status file was read successfully
ERR_OCACCESS	2	`handle` does not have access to scanner
ERR_OCSCANCFG	14	scanner has not been configured

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
STSFILE   stsfile;
int       retcode;
   retcode = OC_GetStatusFile( Handle, &stsfile );
```

OC_GetSwitchPosition

OC_GetSwitchPosition reads the current position of the three-position front-panel switch from the scanner.

Syntax:

```
int    OC_GetSwitchPosition(HANDLE handle, BYTE *swpos);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
swpos	If swpos is: SWITCH_TOP switch is in the top position SWITCH_MIDDLE switch is in the middle position SWITCH_BOTTOM switch is in the bottom position

Description:

The switch position has no effect on the scanner. The application can use this switch for any purpose.

The scanner must be initialized before you can monitor the switch position.

Return Value:

Name:	Value:	Description:
SUCCESS	0	switch position was read successfully
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
BYTE      sw_pos;
int       retcode;
   retcode = OC_GetSwitchPosition( Handle, &sw_pos );
   if ( sw_pos == SWITCH_BOTTOM )
      OC_SetScanMode ( Handle, SCAN_IDLE );
```

OC_GetTemperature

OC_GetTemperature reads the current temperature of the 1746 I/O PCI Interface's built-in thermometer.

Syntax:

```
int    OC_GetTemperature(HANDLE handle, BYTE*temp);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
temp	Returns the temperature in degrees Celsius

Description:

The temperature is updated every 10 seconds by the scanner.

The optimal operating temperature range for the 1746 I/O PCI Interface is 0° to 60°C. When OC_GetTemperature returns a value of 75_ C or higher, the 1746 I/O PCI Interface is operating beyond its optimal operating temperature range and you need to correct the situation.

The scanner must be initialized before you can monitor the temperature.

Return Value:

Name:	Value:	Description:
SUCCESS	0	temperature was read successfully
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
BYTE      temp;
int       retcode;
   retcode = OC_GetTemperature( Handle, &temp );
```

OC_GetUserJumper State

OC_GetUserJumperState reads the state of the user selectable jumper.

Syntax:

```
int    OC_GetUserJumperState(HANDLE handle, BYTE *jmpr);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
jmpr	If jmpr is: JUMPER_PRESENT jumper is installed JUMPER_ABSENT jumper is not installed

Description:

The scanner reads the state of the jumper once during its POST and does not continually monitor the state of the jumper.

The scanner must be initialized before you can monitor the jumper position.

Return Value:

Name:	Value:	Description:
SUCCESS	0	switch position was read successfully
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
BYTE      jmpr;
int       retcode;
   retcode = OC_GetUserJumperState( Handle, &jmpr );
```

OC_GetUserLEDState

OC_GetUserLEDState reads the status of one of the four user-defined LEDs.

Syntax:

```
int    OC_GetUserLEDState(HANDLE handle, int lednum, int *state);
```

Parameters:

Parameter:	Description:	
handle	Must be a valid handle returned from OC_OpenScanner	
lednum	Must be a value from 1 to 4, which corresponds to LED 1, LED 2, LED 3, and LED 4	
state	If state is:	
	LED_OFF	LED is off
	LED_RED_SOLID	LED is on, red solid
	LED_GREEN_SOLID	LED is on, green solid
	LED_RED_FLASH	LED is on, red flashing (LED1 and LED2 only)
	LED_GREEN_FLASH	LED is on, green flashing (LED1 and LED2 only)

Description:

The application can use this function to determine the current state of the LEDs.

Return Value:

Name:	Value:	Description:
SUCCESS	0	LED was read successfully
ERR_OCACCESS	2	handle does not have access to scanner

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
int       led_state;
int       retcode;
   retcode = OC_GetUserLEDState( Handle, 1, &led_state );
```

OC_GetVersionInfo

OC_GetVersionInfo retrieves the current version of the API library, 1746 I/O PCI Interface hardware, and scanner firmware.

Syntax:

```
int    OC_GetVersionInfo(HANDLE handle, OCVERSIONINFO *verinfo);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
verinfo	Returns the current version of the API library, 1746 I/O PCI Interface hardware, and scanner firmware

Description:

The scanner must be initialized before this function returns valid version information.

The OCVERSIONINFO structure is defined as:

```
typedef struct tagOCVERSIONINFO
{
   WORD   APISeries;                /* API series                */
   WORD   APIRevision;              /* API revision              */
   WORD   ScannerFirmwareSeries;    /* Scanner firmware series   */
   WORD   ScannerFirmwareRevision;  /* Scanner firmware revision */
   WORD   OCHardwareSeries;         /* Hardware series           */
   WORD   OCHardwareRevision;       /* Hardware revision         */
} OCVERSIONINFO;
```

The Windows NT version uses the above structure with these additional members:

```
   WORD   OCDriverSeries;           /* Device driver series */
   WORD   OCDriverRevision          /* Device driver series revision */
```

Return Value:

Name:	Value:	Description:
SUCCESS	0	version information was read successfully
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE          Handle;
OCVERSIONINFO   verinfo;
int             retcode;
   retcode = OC_GetVersionInfo( Handle, &verinfo );
```

OC_InitScanner

This function initializes the shared memory interface between the host and scanner and this function configures the shared memory partitioning.

Syntax:

```
int    OC_InitScanner(HANDLE handle, OCINIT *scaninit);
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`scaninit`	Points to the structure that contains the initialization information passed from the application

Description:

If the scanner is executing POST when this function is called, ERR_OCPOST is returned.

If the scanner has been previously initialized and the partition information in `scaninit` is identical to the current scanner partitioning, OC_InitScanner returns successfully.

If the scanner has been previously initialized and the partition information in `scaninit` is different from the current scanner partitioning, OC_InitScanner returns an error value that indicates that the scanner was previously initialized. The scanner must be reset via OC_ResetScanner before the initialization information can be changed. If the scanner has already been initialized, you can call OC_GetScannerInitInfo to retrieve current partition information.

The `OCINIT` structure is defined as:

```
typedef struct tagOCINIT
{
   WORD   OutputImageSize;       /* size in bytes */
   WORD   InputImageSize;        /* size in bytes */
   WORD   HostRetentiveDataSize; /* size in bytes */
} OCINIT;
```

Return Value:

Name:	Value:	Description:
SUCCESS	0	scanner was initialized successfully
ERR_OCACCESS	2	`handle` does not have access to the scanner
ERR_OCMEM	3	shared memory not found
ERR_OCPAR	6	initialization failed due to invalid partition information
ERR_OCPOST	7	POST in progress or scanner POST failed, see OC_GetScannerStatus
ERR_OCREINIT	4	scanner has already been initialized
ERR_OCRESPONSE	10	scanner did not respond to request

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
OCINIT    scaninit;
int       retcode;
   scaninit.InputImageSize = 128;          /* 64 words for input image */
   scaninit.OutputImageSize = 128;         /* 64 words for output image */
   scaninit.HostRetentiveDataSize = 500;   /* 256 words for host data area */
   retcode = OC_InitScanner( Handle, &scaninit );
```

OC_OpenScanner

OC_OpenScanner acquires access to the scanner device and sets a unique ID that the application uses to access the scanner in subsequent functions.

Syntax:

```
DOS int    OC_OpenScanner(HANDLE *handle, 0, 0);
NT  int    OC_OpenScanner(HANDLE *handle);
```

Important: The two argument values of zero are ignored by the DOS API function.They are a carryover from the Open Controller API.

Description:

This function must be called before any of the other scanner access functions can be used.

ATTENTION: After OC_OpenScanner has been called, OC_CloseScanner **must** be called before exiting the application.

Return Value:

Name:	Value:	Description:
SUCCESS	0	scanner was opened successfully
ERR_OCOPEN	1	scanner is already open
ERR_OCMEM	3	shared memory not found

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
int       retcode;
   retcode = OC_OpenScanner( &Handle, 0, 0 );
```

OC_PetHostWatchdog

OC_PetHostWatchdog increments the host-to-scanner watchdog register of the scanner.

Syntax:

```
void   OC_PetHostWatchdog(HANDLE handle);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner

Description:

OC_PetHostWatchdog must be called at time intervals less than the timeout value specified in the OC_SetHostWatchdog function.

Return Value:

Name:	Value:	Description:
SUCCESS	0	watchdog was updated successfully
ERR_OCACCESS	2	handle does not have access to scanner

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
   retcode = OC_PetHostWatchdog( Handle );
```

OC_PollScanner

OC_PollScanner reads module I/O interrupt, discrete input interrupt, and end-of-scan notification messages from the scanner.

Syntax:

```
int    OC_PollScanner(HANDLE handle, int MsgFilter, MSGBUF *msgbuf);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
MsgFilter	If MsgFilter is: OCMSG_DIINT OCMSG_IOINT, OCMSG_EOS, OCMSG_EOS_DMDIN or OCMSG_EOS_DMDOUT OCMSG_ANY message, in OC_PollScanner returns a message only of the corresponding type, if any OC_PollScanner searches for a the order OCMSG_DIINT, OCMSG_IOINT, OCMSG_EOS_DMDIN, OCMSG_EOS_DMDOUT, then OCMSG_EOS, from any of the sources.
msgbuf	A structure that contains scanner messages

Description:

The MSGBUF structure is defined as:

```
typedef struct {
   BYTE   MsgID;
   BYTE   MsgData[4];
} MSGBUF;
```

The MsgID member of the msgbuf structure will be one of the following values:

This value:	Means:
OCMSG_NONE	No message available
OCMSG_IOINIT	I/O module interrupt message, see OC_SetModuleInterrupt
OCMSG_DIINT	Discrete input interrupt message, see OC_ConfigureDII
OCMSG_EOS_DMDIN	End-of-scan notification message from OC_DemandInputScan command, see OC_EnableEOSNotify
OCMSG_EOS_DMDOUT	End-of-scan notification message from OC_DemandOutputScan command, see OC_EnableEOSNotify
OCMSG_EOS	End-of-scan notification message for timed I/O scan, see OC_EnableEOSNotify

The data returned in the MsgData member array depends upon the value in MsgID:

This MsgID:	Returns:
MsgID=OCMSG_IOINT	MsgData[0] slot number that generated the interrupt
MsgID=OCMSG_DIINT	MsgData[0] mask of last bit transition that generated the interrupt MsgData[2] lowbyte of count of matches that generated the interrupt MsgData[3] highbyte of count of matches that generated the interrupt
MsgID=OCMSG_EOS_DMDIN MsgID=OCMSG_EOS_DMDOUT MsgID=OC_EOS	MsgData[0] number of scans that occurred since the last end-of-scan message of this type was read

Separate queues hold messages received from each source. The host application must call OC_PollScanner periodically to read messages from each enabled source to prevent messages from being discarded. If a message queue is full when a new message is received, the oldest message is discarded and the next call to OC_PollScanner results in a return value of ERR_OCOVERRUN. The queue can hold as many as five messages.

Return Value:

Name:	Value:	Description:
SUCCESS	0	scanner was polled successfully
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCOVERRUN	16	a message has been discarded

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
MSGBUF    msgbuf;
int       retcode;
   retcode = OC_PollScanner( Handle, OCMSG_ANY, &msgbuf );
   /* Check msgbuf.MsgID for what message is available */
```

OC_ReadHostRetentive Data

OC_ReadHostRetentiveData reads the host-retentive-data partition of the scanner.

Syntax:

```
int    OC_ReadHostRetentiveData(HANDLE handle, BYTE *buf, WORD offset, WORD len);
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`buf`	Contains the data that is read
`offset`	The data is read `offset` bytes from the beginning of the data partition If the number of bytes from `offset` to the end of the partition is smaller than `len`, no bytes will be read and ERR_OCPARAM is returned.
`len`	Defines how many bytes to read

Description:

Data is not read past the end of the host-retentive-data partition.

It is recommended that you verify the integrity of the data stored in the host-retentive-data partition. You can use the OC_CalculateCRC function to generate a 16-bit CRC.

Important: The jumper for the battery-backup dual-port memory is disabled by default. You must switch the jumper to enable the battery-backup feature.

Return Value:

Name:	Value:	Description:
SUCCESS	0	host retentive data was written successfully
ERR_OCACCESS	2	`handle` does not have access to scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner
ERR_OCPARAM	8	parameter contains invalid value

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
BYTE      retent_data[500];
int       retcode;
   retcode = OC_ReadHostRetentiveData( Handle, retent_data, 0, 500 );
```

OC_ReadInputImage

OC_ReadInputImage reads the current input image from the scanner.

Syntax:

```
int    OC_ReadInputImage(HANDLE handle, WORD *inpimgcpy, int slotnum, WORD offset, WORD
len, WORD *imagebuf);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
inpimgcpy	If inpimgcpy is: NULL OC_ReadInputImage reads data directly from the input image in the scanner's shared memory. not NULL OC_ReadInputImage reads data from inpimgcpy which contains a copy of the scanner's input image.
slotnum offset len	If slotnum is: positive input data for that slot is read into the array pointed to by imagebuf. Then imagebuf contains len words of input data starting at word offset of the module in the slot. Data will not be read past the end of the input image for the slot. -1 the entire input image is read into the array pointed to by imagebuf, and offset and len are ignored.
imagebuf	Must point to an array that is large enough to accept the amount of data in the requested input image

Description:

To guarantee that a series of calls to OC_ReadInputImage read data from a single input scan, OC_ReadInputImage can first be called to read the entire input image into a local buffer pointed to by imagebuf, with the inpimgcpy pointer set to NULL and slotnum set to -1. The imagebuf buffer can then be passed as inpimgcpy in subsequent OC_ReadInputImage calls to retrieve the slot data from the copy of the input image. This preserves input image file integrity across multiple calls to OC_ReadInputImage.

If file integrity is not necessary, the host application can set inpimgcpy to NULL and access data directly from shared memory.

Return Value:

Name:	Value:	Description:
SUCCESS	0	input image was read successfully
ERR_OCACCESS	2	`handle` does not have access to scanner
ERR_OCPARAM	8	parameter contains invalid value
ERR_OCSCANCFG	14	scanner has not been configured
ERR_OCSLOT	12	slot number is invalid

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
WORD      inputdata[2];
int       retcode;

   retcode = OC_ReadInputImage( Handle, NULL, 6, 0, 2, inputdata );

   /* Read slot 6 data, first 2 words, directly from the input image */
   /* table to inputdata buffer */
```

OC_ReadIOConfigFile

OC_ReadIOConfigFile reads the configuration data that is already stored in the DOS file created using OC_WriteIOConfigFile.

Syntax:

```
int    OC_ReadIOConfigFile(OCIOCFG *iocfg, char *filename);
```

Parameters:

Parameter:	Description:
`iocfg`	A structure that contains a copy of the configuration data that is in `filename`
`filename`	References a file that was created using OC_WriteIOConfigFile

Description:

If G file data is included in the configuration file, OC_ReadIOConfigFile allocates memory for the data and initializes the G data values to point to the allocated memory. The host application should release the allocated memory via the `free()` function when it is no longer needed.

Return Value:

Name:	Value:	Description:
SUCCESS	0	I/O configuration file was read successfully
ERR_OCFILERROR	19	error encountered while opening or reading the file

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
OCIOCFG   iocfg;
int       retcode;
   /* DOS file named RACK1.CFG, is storing the rack configuration info. */
   retcode = OC_ReadIOConfigFile( &iocfg, ”RACK1.CFG” );
   /* Use OC_DownloadIOConfiguration() to download the information */
```

OC_ReadModuleFile

OC_ReadModuleFile reads a data file from a module.

Syntax:

```
int    OC_ReadModuleFile(HANDLE handle, BYTE ftype, WORD *mfile, int slotnum, WORD offset,
WORD len);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
ftype	Defines the module type as: FILTYP_M0 M0 file FILTYP_M1 M1 file FILTYP_G G file
mfile	Buffer file that contains data read from the module at slotnum The data in mfile is read from the module starting at word offset. This function does not read data past the end of the module file for the slot.
slotnum	Must be a valid slot number
offset	Must be valid word number within module file
len	Number of words read from the module located at slotnum on the scanner into the buffer mfile

Description:

This function accesses an internal data file of the selected module. I/O scanning is blocked while this access takes place.

Return Value:

Name:	Value:	Description:
SUCCESS	0	file was read successfully
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCPARAM	8	parameter contains invalid value
ERR_OCRESPONSE	10	scanner did not respond to request
ERR_OCSCANCFG	14	scanner has not been configured
ERR_OCSLOT	12	slot number is invalid

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
WORD      filedata[2];
int       retcode;
   retcode = OC_ReadModuleFile( Handle, FILTYP_M1, filedata, 6, 3, 2 );
   /*  Reads words 3 and 4 from module in slot 6. */
```

OC_ReadOutputImage

OC_ReadOutputImage reads the current output image from the scanner.

Syntax:

```
int    OC_ReadOutputImage(HANDLE handle, WORD *outimgcpy, int slotnum,
WORD offset, WORD len, WORD *imagebuf);
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`outimgcpy`	If `outimgcpy` is: NULL OC_ReadOutputImage reads data directly from the output image in the scanner's shared memory. not NULL OC_ReadInputImage reads data from `outimgcpy` which contains a copy of the scanner's output image.

`slotnum`
`offset`
`len`
	If `slotnum` is positive, the output image for that slot is read into the array pointed to by `imagebuf`. Then `imagebuf` is filled with `len` words of output data starting at word `offset` of the module in the slot.
	If `slotnum` is -1, the entire output image is read into the array pointed to by `imagebuf`, and `offset` and `len` are ignored.

`imagebuf`	Must point to an array that is large enough to accept the amount of data in the requested output image

Description:

Since the scanner never changes data in the output image, it is not necessary to copy the image, as with the OC_ReadInputImage function, to preserve file integrity. It is supported, however, to provide a consistent interface.

Return Value:

Name:	Value:	Description:
SUCCESS	0	output image was read successfully
ERR_OCACCESS	2	`handle` does not have access to scanner
ERR_OCPARAM	8	parameter contains invalid value
ERR_OCSCANCFG	14	scanner has not been configured
ERR_OCSLOT	12	slot number is invalid

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
WORD      outputdata[2];
int       retcode;

   retcode = OC_ReadOutputImage( Handle, NULL, 6, 0, 2, outputdata );

   /* Read slot 6 data, first 2 words, directly from the output image */
   /* table to outputdata buffer */
```

OC_ReadSRAM

OC_ReadSRAM reads data from the battery-backed user memory.

Syntax:

```
int OC_ReadSRAM(HANDLE Handle, BYTE *bufptr, DWORD offset, DWORD length);
```

Description:

The battery-backed memory may be used to store important data that needs to be preserved in the event of a power failure. The size of the available memory in bytes may be obtained using OC_GetDeviceInfo.

Important: It is recommended that the integrity of data stored in the user memory be verified by some means. The OC_CalculateCRC function may be used to generate a 16-bit CRC that may be used for this purpose.

`handle` must be a valid handle returned from OC_OpenScanner. `bufptr` points to a buffer to be receive the data. `offset` specifies the offset within the memroy to begin reading. `length` specifiies the number of bytes to be read.

If *`offset + length`* points past the end of the memory, no bytes will be written and ERR_OCPRAM will be returned.

Return Value:

Name:	Description:
SUCCESS	data was read successfully
ERR_OCACCESS	`handle` does not have access to scanner
ERR_OCPARAM	`offset+length` points past the end of the memory

Considerations:

Supported in the Windows NT API library only.

Example:

```
Byte   buf[100];                /*buffer of 100 bytes of important data*/
HANDLE handle;
Word crc, crc_saved;

/* Read 100 bytes of data from offset 0 in SRAM */
OC_ReadSRAM(handle, buf, 0, 100);
/* Calculate CRC */
OC_CalculateCRC(buf, 100, &crc);
/* Read saved CRC at offset 100 in SRAM */
OC_ReadSRAM(handle, &crc_saved, 100, 2);
/* Check CRC */
if (crc != saved_crc)
   printf(“ERROR: Data is corrupted.\n”);
```

OC_ResetScanner

OC_ResetScanner generates a temporary hard reset to the scanner.

ATTENTION: This call stops scanning and resets outputs.

Syntax:

```
int    OC_ResetScanner(HANDLE handle, int mode);
```

Parameters:

Parameter:	Description:	
handle	Must be a valid handle returned from OC_OpenScanner	
mode	If mode is:	
	OCNOWAIT	OC_ResetScanner returns after releasing the reset signal to the scanner.
	OCWAIT	OC_ResetScanner returns after POST is completed.

Description:

After the reset is generated, the scanner begins to execute its POST.

Return Value:

Name:	Value:	Description:
SUCCESS	0	scanner was reset successfully
ERR_OCACCESS	2	handle does not have access to the scanner
ERR_OCPOST	7	scanner POST failed

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
int       retcode;
   retcode = OC_ResetScanner( Handle, OCWAIT );
```

OC_SetForces

OC_SetForces installs and removes input and output forces to the scanner.

Syntax:

```
int    OC_SetForces(HANDLE handle, FORCEDATA *forcedata)
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
forcedata	Defines the inputs and outputs to force.

Description:

If the result of OC_SetForces removes all I/O forces, the scanner disables forces. If any I/O forces are later installed, OC_EnableForces must be called to re-enable forces.

The FORCEDATA structure is defined as:

```
typedef struct tagFORCEDATA {
   BYTE   SlotNum;      /* Slot Number of local I/O or 1747-SN module (1-30)*/
   WORD   WordOffset;   /* Word Offset in I/O image */
   BYTE   IOType;       /* Selects Input or Output Image */
   WORD   ForceMask;    /* Install/Remove bitmask */
   WORD   ForceVal;     /* Install value bitmask */
} FORCEDATA;
```

This value:	Means:
SlotNum WordOffset	SlotNum and WordOffset select the word of I/O that contains the bits to be forced
IOType	IOType must be FORCE_INPUTS or FORCE_OUTPUTS
ForceMask ForceVal	All 16 bits of the word are installed/removed according to ForceMask and ForceVal Each bit in ForceMask that is set to 0 will have its force removed. Each bit in ForceMask that is set to 1 will have its force installed. For each bit that has its force installed, the corresponding bit in ForceVal determines the state of the force. For bits that have their force removed, the corresponding bit in ForceVal is ignored.

Return Value:

Name:	Value:	Description:
SUCCESS	0	I/O forces were configured successfully
ERR_OCACCESS	2	`handle` does not have access to scanner
ERR_OCPARAM	8	parameter contains invalid value
ERR_OCRESPONSE	10	scanner did not respond to request
ERR_OCSCANCFG	14	scanner has not been configured
ERR_OCSLOT	12	slot number is invalid

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    handle;
FORCEDATA forces;
int       retcode;
   /* Force low byte of input word 1 of slot 6 to 0x5A */
   forces.SlotNum = 6;
   forces.WordOffset = 1;
   forces.IOType = FORCE_INPUTS;
   forces.ForceMask = 0x00FF;
   forces.ForceVal = 0x005A;
   retcode = OC_SetForces( handle, &forces );
   /* Must call OC_EnableForces() to actually apply the force data */
```

OC_SetHostWatchdog

OC_SetHostWatchdog sets the host-to-scanner watchdog delay and mode of the scanner.

Syntax:

```
int    OC_SetHostWatchdog(HANDLE handle, int mode, WORD delay);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
mode	If mode is: WATCHDOG_IGNORE the host-to-scanner watchdog is disabled (default) WATCHDOG_IDLE a watchdog timeout causes the scanner to fault. The status LED is set to flashing red, the I/O is reset, I/O scanning stops, and internal scanner error of 0x0B is set, and the major error code is set to 0x40. Use OC_ClearFault before the scanner can be set to Scan mode
delay	Specifies the watchdog timeout in multiples of 100ms and can have any value from 1 (100ms) to 65535 (6553.5s).

Description

Once the Host Watchdog is enabled, the host application must call OC_PetHostWatchdog more often than the time specified as the watchdog timeout. If the host application does not call OC_PetHostWatchdog for a time longer than the watchdog timeout, then the action specified by mode is performed.Return Value

Return Value:

Name:	Value:	Description:
SUCCESS	0	host watchdog was set successfully
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner
ERR_OCPARAM	8	parameter contains invalid value

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
int       retcode;
   retcode = OC_SetHostWatchdog(Handle, WATCHDOG_IDLE, 10 );
   /* Watchdog times out in 1 second and places scanner in idle mode */
```

OC_SetInputUpdate Mode

OC_SetInputUpdateMode controls how the scanner updates inputs.

Syntax:

```
int    OC_SetInputUpdateMode(HANDLE handle, int mode);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
mode	If mode is: INUPD_NEVER the scanner does not scan inputs unless explicitly requested by the OC_DemandInputScan function. INUPD_ALWAYS the scanner continuously updates inputs on every scan. By default, the input update mode is INUPD_ALWAYS. A change in status of the input update mode takes effect at the start of the next scan.

Description:

This function does not affect output image scanning.

Return Value:

Name:	Value:	Description:
SUCCESS	0	conditional scan was set successfully
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCINIT	5	scanner has not been initialized; see OC_InitScanner
ERR_OCPARAM	8	parameter contains invalid value

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
int       retcode;
   retcode = OC_SetInputUpdateMode( Handle, INUPD_NEVER );
   /* Must use OC_DemandInputScan() function to obtain new input data */
```

OC_SetIOIdleState

OC_SetIOIdleState controls the state of I/O when the scanner goes from Scan mode to Idle mode.

Syntax:

```
int    OC_SetIOIdleState(HANDLE handle, int mode);
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`mode`	If `mode` is: IDLESTATE_HOLD module I/O's maintain their last state. IDLESTATE_RESET module I/O's are reset by the scanner. The default I/O idle state is IDLESTATE_RESET.

Description:

The I/O will always be reset in the case of a fault.

Return Value:

Name:	Value:	Description:
SUCCESS	0	I/O state was changed successfully
ERR_OCACCESS	2	`handle` does not have access to scanner
ERR_OCINIT	5	scanner has not been initialized; see OC_InitScanner
ERR_OCPARAM	8	parameter contains invalid value

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
int       retcode;
   retcode = OC_SetIOIdleState( Handle, IDLESTATE_HOLD );

   /* Outputs will remain in last state when scanner goes to idle mode */
```

OC_SetModuleInterrupt

OC_SetModuleInterrupt enables, disables, or acknowledges the module interrupt for the slot `slotnum` on the scanner.

Syntax:

```
int    OC_SetModuleInterrupt(HANDLE handle, int slotnum, int mode);
```

Parameters:

Parameter:	Description:	
`handle`	Must be a valid handle returned from OC_OpenScanner	
`slotnum`	Must be a valid slot number	
`mode`	If `mode` is:	
	IOINT_DISABLE	the module interrupt is disabled (default)
	IOINT_ENABLE	the module interrupt is enabled
	IOINT_ACK	the module interrupt is acknowledged.

Description:

When a module interrupt is received, the scanner generates a module interrupt message that the host application can read by calling the OC_PollScanner function. After retrieving the module interrupt message, the host application should immediately acknowledge the module interrupt and then service the module interrupt message. The module interrupt must be acknowledged before another can be received from that module.

Return Value:

Name:	Value:	Description:
SUCCESS	0	module interrupt was processed successfully
ERR_OCACCESS	2	`handle` does not have access to scanner
ERR_OCPARAM	8	parameter contains invalid value
ERR_OCRESPONSE	10	scanner did not respond to request
ERR_OCSCANCFG	14	scanner has not been configured
ERR_OCSLOT	12	slot number is invalid

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
int       retcode;
   retcode = OC_SetModuleInterrupt( Handle, 6, IOINT_ENABLE );
   /* Slot 6 module now enabled to generate module interrupts. */
   /* Use OC_PollScanner() to check for Module Interrupt messages. */
```

OC_SetOutputUpdate Mode

OC_SetOutputUpdateMode controls how the scanner updates real outputs from the Output Image.

Syntax:

```
int    OC_SetOutputUpdateMode(HANDLE handle, int mode);
```

Parameters:

Parameter:	Description:	
handle	Must be a valid handle returned from OC_OpenScanner	
mode	If mode is:	
	OUTUPD_NEVER	the scanner does not write outputs from the output image unless explicitly requested by the OC_DemandOutputScan function
	OUTUPD_CHANGE	the scanner writes outputs only when the scanner output image is written via OC_WriteOutputImage, but does not continuously update outputs from the output image every scan
	OUTUPD_ALWAYS	the scanner continuously updates outputs from the output image on every scan.
	By default, the output update mode is OUTUPD_NEVER. A change in status of the output update mode will take effect at the start of the next scan.	

Description:

This function does not affect input image scanning.

Setting the output update mode to OUTUPD_NEVER allows the host application to read the input image and perform logic to determine the initial state of outputs. Once the output image is written with the initial state data, the output update mode can be changed to allow updating of the outputs.

Setting the output update mode to OUTUPD_CHANGE allows the host application to signal the scanner to write outputs by calling OC_WriteOutputImage. This setting allows the scanner's minimum scan time to be reduced (since it is only scanning inputs most of the time), and is provided as a performance enhancement.

Setting the output update mode to OUTUPD_ALWAYS forces the scanner to write outputs from the output image on every scan.

Return Value:

Name:	Value:	Description:
SUCCESS	0	conditional scan was set successfully
ERR_OCACCESS	2	`handle` does not have access to the scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner
ERR_OCPARAM	8	parameter contains invalid value

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
int       retcode;
   retcode = OC_SetOutputUpdateMode( Handle, OUTUPD_NEVER );
   /* Must use OC_DemandOutputScan() function to send new output data */
```

OC_SetScanMode

OC_SetScanMode changes the scan mode of the scanner.

Syntax:

```
int    OC_SetScanMode(HANDLE handle, int mode);
```

Parameters:

Parameter:	Description:	
handle	Must be a valid handle returned from OC_OpenScanner	
mode	If mode is:	
	SCAN_IDLE	the scanner changes to Idle mode and stops scanning I/O
	SCAN_RUN	the scanner changes to Scan mode and begins scanning I/O.

Description:

The scanner must be properly configured before going to Scan mode.

OC_SetScanMode will fail if there are any unread extended errors or faults. Call OC_GetExtendedError to extract all extended errors and call OC_ClearFault to clear any faults before calling OC_SetScanMode.

Before the scanner changes to Scan mode, it compares the downloaded I/O configuration to the racks and I/O modules actually installed. If there are any problems, an extended error is generated and OC_SetScanMode returns an error. If the scanner finds no problems, the scanner is in Scan mode when this function returns.

Return Value:

Name:	Value:	Description:
SUCCESS	0	scan mode was set successfully
ERR_OCACCESS	2	handle does not have access to the scanner
ERR_OCFAULT	13	scanner is faulted
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner
ERR_OCPARAM	8	parameter contains invalid value
ERR_OCRESPONSE	10	scanner did not respond to request

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
int       retcode;
   retcode = OC_SetScanMode( Handle, SCAN_RUN );             /* Scan I/O */
```

OC_SetScanTime

OC_SetScanTime sets the I/O scan time and I/O scan interval of the scanner.

Syntax:

```
int    OC_SetScanTime(HANDLE handle, int mode, int time);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
mode time	If mode is: SCAN_PERIODIC time determines the frequency of I/O scans in multiples of 250us and must contain a value between 1 and 255. SCAN_DELAYED time determines the delay between I/O scans in multiples of 10us and must contain a value between 1 and 256. The default mode is SCAN_PERIODIC and the default time is 10 if OC_SetScanTime is not used to change the scan time.

Description:

A scan time change will take effect when the scanner transitions from Idle mode to Scan mode.

Return Value:

Name:	Value:	Description:
SUCCESS	0	scan time was set successfully
ERR_OCACCESS	2	handle does not have access to the scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner
ERR_OCPARAM	8	parameter contains invalid value

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
int       retcode;
   retcode = OC_SetScanTime( Handle, SCAN_PERIODIC, 20 );
   /* Scan set to start every 5 msec. */
```

OC_SetUserLEDState

OC_SetUserLEDState sets the state of a user-defined LED

Syntax:

```
int    OC_SetUserLEDState(HANDLE handle, int lednum, int state);
```

Parameters:

Parameter:	Description:	
`handle`	Must be a valid handle returned from OC_OpenScanner	
`lednum`	Must be a value from 1 to 4, which corresponds to LED1, LED2, LED3, and LED4	
`ledstate`	If `ledstate` is:	
	LED_OFF	LED is off
	LED_RED_SOLID	LED is on, red solid
	LED_GREEN_SOLID	LED is on, green solid
	LED_RED_FLASH	LED is on, red flashing (LED1 and LED2 only)
	LED_GREEN_FLASH	LED is on, green flashing (LED1 and LED2 only)

Description:

The application can use the four user LEDs for any purpose.

Return Value:

Name:	Value:	Description:
SUCCESS	0	LED was updated successfully
ERR_OCACCESS	2	`handle` does not have access to scanner
ERR_OCPARAM	8	parameter contains invalid value

Considerations:

Supported in the DOS API library and the Windows NT API library.

Example:

```
HANDLE    Handle;
int       retcode;
   retcode = OC_SetUserLEDState( Handle, 1, LED_GREEN_SOLID );
```

OC_SetupPowerFail Action

OC_SetupPowerFailAction registers the action to be taken when a power fail interrupt is received from the scanner.

Syntax:

```
int OC_SetupPowerFailAction(HANDLE handle, BYTE *bufptr, WORD offset, WORD length, void
(*callback)() );
```

Parameters:

Parameter:	Description:	
`handle`	Must be a valid handle returned from OC_OpenScanner	
`bufptr`	If `bufptr` is: NULL Not NULL	 no data is copied (default) at power fail `length` bytes of data are copied from `bufptr` to the host retentive data partition starting at `offset`.
`offset`	Specifies the offset within the host retentive data partition to begin copying	
`length`	Number of bytes to copy If `length` points beyond the end of the host retentive data partition, it is truncated	
`callback`	If `callback` is: NULL Not NULL	 no callback function is executed (default) the power fail interrupt routine calls this function after copying the data to the host retentive data partition (if configured)

Description:

You can configure the power failure action in four ways:

- No action (default); the power failure interrupt is ignored (`bufptr` is NULL; `callback` is NULL)
- Copy a block of data to the host retentive data partition in dual port RAM (`bufptr` points to the data to save; `callback` is NULL)
- Execute a user callback function (`bufptr` is NULL; `callback` points to the routine to call)
- Copy a block of data and execute a callback function (`bufptr` points to the data to save; `callback` points to the routine to call)

A system typically has at least 10 milliseconds, and possibly as much as 50 milliseconds or more between the power fail interrupt and the loss of power. The duration of this interval is a function of the power supply and system configuration, and it varies from system to system. You might need to experiment to determine the typical value for a particular system.

Data is copied to the host retentive data partition at the rate of approximately 1K bytes per millisecond.

Declare `bufptr` as static if the OC_SetupPowerFail function is used other than in `main( )`; otherwise random data will be sent to the host retentive data area.

Return Value:

Name:	Value:	Description:
SUCCESS	0	power fail action was registered successfully
ERR_OCACCESS	2	`handle` does not have access to scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner
ERR_OCPARAM	8	parameter contains invalid value

Considerations:

Supported in the DOS API library only.

Example:

This example is for DOS only.

```
HANDLE    Handle;
BYTE      buffer1[100];
BYTE      buffer2[100];
int       retcode;
   void PowerFailSave( void )                /* power fail callback routine */
   {
       len = 100;
       OC_WriteRtcSRAM( buffer2, 0, &len );    /* Put data in protected area */
   }
   retcode = OC_SetupPowerFailAction( Handle, buffer1, 0, 100, PowerFailSave );
```

OC_WaitForDII

Blocks the calling thread until a DII interrupt is received from the scanner or `msTimeout` milliseconds have elapsed.

Syntax:

```
int    OC_WaitForDII(HANDLE handle, DWORD msTimeout)
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`msTimeout`	Specifies the number of milliseconds to wait Set to INFINITE to wait forever

Description:

If a DII has been received since the last OCMSG_DIINT message was retrieved with the OC_PollScanner function, OC_WaitForDII returns SUCCESS immediately.

Return Value:

Name:	Value:	Description:
SUCCESS	0	a DII was received
ERR_RESPONSE	10	`msTimeout` milliseconds elapsed without a DII
ERR_OCACCESS	2	`handle` does not have access to scanner

Considerations:

Supported in the Windows NT API library only.

Example:

```
HANDLE handle;
intrc;
MSGBUF diiMSG
/* Wait for 10 seconds for a DII */
rc = OC_WaitForDII(handle, 10000);
switch(rc) {
   case SUCCESS:/* got a DII */
/* fetch the DII message */
      OC_PollScanner (handle,OCMSG_DIINT,&diiMSG);
   break;
   case ERR_OCRESPONSE:/* timed out */
      printf("\nTimed out waiting for DII\n");
   break;
   default;
      printf("\nError!\n");
   break;
}
```

OC_WaitForEos

Blocks the calling thread until an end-of-scan (EOS) notification is received from the scanner or `msTimeout` milliseconds have elapsed.

Syntax:

```
int    OC_WaitForEos(HANDLE handle, DWORD msTimeout)
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`msTimeout`	Specifies the number of milliseconds to wait Set to INFINITE to wait forever

Description:

If an EOS message has been received since the last OCMSG_EOS message was retrieved with the OC_PollScanner function, OC_WaitForEos returns SUCCESS immediately.

You can use this function to synchronize a control application with the I/O scan. See the OC_EnableEOSNotify function.

Return Value:

Name:	Value:	Description:
SUCCESS	0	an EOS message was received
ERR_RESPONSE	10	`msTimeout` milliseconds elapsed without an EOS
ERR_OCACCESS	2	`handle` does not have access to scanner

Considerations:

Supported in the Windows NT API library only.

Example:

```
HANDLE handle;
intrc;
MSGBUF eosMSG

/* Wait for 10 seconds for the EOS */
rc = OC_WaitForEos(handle, 10000);
switch(rc) {
   case SUCCESS:/* got EOS*/
/* reset the EOS event*/
      OC_PollScanner (handle,OCMSG_EOS,&eosMSG);
   break;
   case ERR_OCRESPONSE:/* timed out */
      printf("\nTimed out waiting for EOS\n");
   break;
   default;
      printf("\nError!\n");
   break;
}
```

OC_WaitForEosDmdIn

Blocks the calling thread until a demand input end-of-scan (EOS) notification is received from the scanner or `msTimeout` milliseconds have elapsed.

Syntax:

```
int    OC_WaitForEosDmdIn(HANDLE handle, DWORD msTimeout)
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`msTimeout`	Specifies the number of milliseconds to wait Set to INFINITE to wait forever

Description:

If a demand input EOS message has been received since the last OCMSG_EOS_DMDIN message was retrieved with the OC_PollScanner function, OC_WaitForEosDmdIn returns SUCCESS immediately.

You can use this function to synchronize a control application with the I/O scan. See the OC_EnableEOSNotify and OC_DemandInputScan functions.

Return Value:

Name:	Value:	Description:
SUCCESS	0	demand input EOS message was received
ERR_RESPONSE	10	`msTimeout` milliseconds elapsed without an EOS
ERR_OCACCESS	2	`handle` does not have access to scanner

Considerations:

Supported in the Windows NT API library only.

Example:

```
HANDLE handle;
intrc;
MSGBUF eosMSG

/* Request an input scan, but don't wait */
OC_DemandInputScan(handle, OCNOWAIT);

/* Could have other code here */


/* Wait 1 second for the EOS */
rc = OC_WaitForEosDmdIn(handle, 1000);
switch(rc) {
   case SUCCESS:/* got EOS*/
/* reset the EOS_DMDIN event*/
      OC_PollScanner (handle,OCMSG_EOS_DMDIN,&eosMSG);
      /* do logic, etc. synchronized with the I/O scan */
   break;
   case ERR_OCRESPONSE:/* timed out */
      printf("\nTimed out waiting for EOS\n");
   break;
   default;
      printf("\nError!\n");
   break;
}
```

OC_WaitForEosDmdOut

Blocks the calling thread until a demand output end-of-scan (EOS) notification is received from the scanner or `msTimeout` milliseconds have elapsed.

Syntax:

```
int    OC_WaitForEosDmdOut(HANDLE handle, DWORD msTimeout)
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`msTimeout`	Specifies the number of milliseconds to wait Set to INFINITE to wait forever

Description:

If a demand output EOS message has been received since the last OCMSG_EOS_DMDOUT message was retrieved with the OC_PollScanner function, OC_WaitForEosDmdOut returns SUCCESS immediately.

You can use this function to synchronize a control application with the I/O scan. See the OC_EnableEOSNotify and OC_DemandOutputScan functions.

Return Value:

Name:	Value:	Description:
SUCCESS	0	demand output EOS message was received
ERR_RESPONSE	10	`msTimeout` milliseconds elapsed without an EOS
ERR_OCACCESS	2	`handle` does not have access to scanner

Considerations:

Supported in the Windows NT API library only.

Example:

```
HANDLE handle;
intrc;
MSGBUF eosMSG

/* Request an output scan, but don't wait */
OC_DemandOutputScan(handle, OCNOWAIT);

/* Could have other code here */


/* Wait 1 second for the EOS */
rc = OC_WaitForEosDmdOut(handle, 1000);
switch(rc) {
   case SUCCESS:/* got EOS*/
/* reset the EOS_DMDOUT event*/
      OC_PollScanner (handle,OCMSG_EOS_DMDOUT,&eosMSG);
      /* do logic, etc. sychronized with the I/O scan */
   break;
   case ERR_OCRESPONSE:/* timed out */
      printf("\nTimed out waiting for EOS\n");
   break;
   default;
      printf("\nError!\n");
   break;
}
```

OC_WaitForExtError

Blocks the calling thread until an extended error is received from the scanner or `msTimeout` milliseconds have elapsed.

Syntax:

```
int    OC_WaitForExtError(HANDLE handle, DWORD msTimeout)
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`msTimeout`	Specifies the number of milliseconds to wait Set to INFINITE to wait forever

Description:

If an extended error has been received since the last extended error message was retrieved with the OC_GetExtendedError function, OC_WaitForExtError returns SUCCESS immediately.

Return Value:

Name:	Value:	Description:
SUCCESS	0	an extended error occurred
ERR_RESPONSE	10	`msTimeout` milliseconds elapsed without an extended error
ERR_OCACCESS	2	`handle` does not have access to scanner

Considerations:

Supported in the Windows NT API library only.

Example:

```
HANDLE handle;
intrc;
OCEXTERRexterr

/* Error handler thread */
while(1) {/* loop forever */
   OC_WaitForExtError(handle, INFINITE);
   /* fetch the error data */
   OC_GetExtendedError(handle, &exterr);
   /*handle the error */
}
```

OC_WaitForIoInt

Blocks the calling thread until a module interrupt is received from the scanner or `msTimeout` milliseconds have elapsed.

Syntax:

```
int    OC_WaitForIoInt(HANDLE handle, DWORD msTimeout)
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`msTimeout`	Specifies the number of milliseconds to wait Set to INFINITE to wait forever

Description:

If a module interrupt has been received since the last OCMSG_IOINT message was retrieved with the OC_PollScanner function, OC_WaitForIoInt returns SUCCESS immediately.

Return Value:

Name:	Value:	Description:
SUCCESS	0	a module interrupt was received
ERR_RESPONSE	10	`msTimeout` milliseconds elapsed without a module interrupt
ERR_OCACCESS	2	`handle` does not have access to scanner

Considerations:

Supported in the Windows NT API library only.

Example:

```
HANDLE handle;
intrc;
MSGBUF mintMSG
/* Wait for 10 seconds for a module interrupt*/
rc = OC_WaitForIoInt(handle, 10000);
switch(rc) {
   case SUCCESS:/* got a module interrupt*/
/* fetch the module interrupt message */
      OC_PollScanner (handle,OCMSG_IOINT,&mintMSG);
/* handle the module interrupt */
   break;
   case ERR_OCRESPONSE:/* timed out */
      printf("\nTimed out waiting for module interrupt\n");
   break;
   default;
      printf("\nError!\n");
   break;
}
```

OC_WriteHostRetentive Data

OC_WriteHostRetentiveData writes data to the host-retentive-data partition of the scanner.

Syntax:

```
int    OC_WriteHostRetentiveData(HANDLE handle, BYTE *buf, WORD offset, WORD len);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
buf	Contains the data that is read
offset	The data is written offset bytes from the beginning of the data partition If the number of bytes from offset to the end of the partition is smaller than len, no bytes are written and ERR_OCPARAM is returned.
len	Defines how many bytes to write

Description:

Data is not written past the end of the host-retentive-data partition.

It is recommended that you verify the integrity of the data stored in the host-retentive-data partition. You can use the OC_CalculateCRC function to generate a 16-bit CRC.

Data written to the host retentive data partition of the shared memory is battery-backed, and will be retained if power is removed from the rack, as long as the battery voltage is good.

Return Value:

Name:	Value:	Description:
SUCCESS	0	host retentive data was written successfully
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCINIT	5	scanner has not been initialized, see OC_InitScanner
ERR_OCPARAM	8	parameter contains an invalid value

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
BYTE      retent_data[500];
int       retcode;
   retcode = OC_WriteHostRetentiveData( Handle, retent_data, 0, 500 );
```

OC_WriteIOConfigFile

OC_WriteIOConfigFile writes the configuration data contained in the `iocfg` structure to the file named `filename`.

Syntax:

```
int    OC_WriteIOConfigFile(OCIOCFG *iocfg, char *filename);
```

Parameters:

Parameter:	Description:
`iocfg`	A structure that contains the configuration data that is to be written to `filename`
`filename`	References the file to write If `filename` does not exist, it is created.

Description:

Configuration files created by OC_WriteIOConfigFile can be read by OC_ReadIOConfigFile.

The `OCIOCFG` structure is defined as:

```
typedef struct tagOCIOCFG
{
   BYTE   Rack1Size;    /* number of slots in Rack 1 */
   BYTE   Rack2Size;    /* number of slots in Rack 2 */
   BYTE   Rack3Size;    /* number of slots in Rack 3 */
   OCSLOTCFGSlotCfg[OCMAXSLOT];/* configuration for each slot */
} OCIOCFG;
```

Return Value:

Name:	Value:	Description:
SUCCESS	0	I/O configuration file was written successfully
ERR_OCFILERROR	19	error encountered while opening or writing the file

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
OCIOCFG‘  iocfg
int       retcode;
   /* Either OC_CreateIOConfiguration() or OC_GetIOConfiguration() were */
   /* called previously to fill in ’iocfg’ structure */
   retcode = OC_WriteIOConfigFile( &iocfg, ”RACK1.CFG” );
```

OC_WriteModuleFile

OC_WriteModuleFile writes a data file to a module.

Syntax:

```
int    OC_WriteModuleFile(HANDLE handle, BYTE ftype, WORD *mfile, int slotnum, WORD
offset, WORD len);
```

Parameters:

Parameter:	Description:
handle	Must be a valid handle returned from OC_OpenScanner
ftype	Defines the module type as: FILTYP_M0 M0 file FILTYP_M1 M1 file FILTYP_G G file
mfile	Buffer file that contains data read from the module at slotnum The data in mfile is written to the module starting at word offset. This function does not write data past the end of the module file for the slot.
slotnum	Must be a valid slot number
offset	Must be valid word number within module file
len	Number of words written from the module located at slotnum on the scanner into the buffer mfile

Description:

This function accesses an internal data file of the selected module. I/O scanning is blocked while this access takes place.

Return Value:

Name:	Value:	Description:
SUCCESS	0	file was read successfully
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCPARAM	8	parameter contains invalid value
ERR_OCRESPONSE	10	scanner did not respond to request
ERR_OCSCANCFG	14	scanner has not been configured
ERR_OCSLOT	12	slot number is invalid

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
WORD      filedata[2];
int       retcode;

   filedata[0] = 0x55AA;
   filedata[1] = 0xAA55;

   retcode = OC_WriteModuleFile( Handle, FILTYP_M0, filedata, 6, 3, 2 );

   /*  Writes words 3 and 4 from module in slot 6  */
```

OC_WriteOutputImage OC_WriteOutputImage updates the output image on the scanner.

Syntax:

```
int    OC_WriteOutputImage(HANDLE handle, WORD *outimgcpy, int slotnum,
WORD offset, WORD len, WORD *imagebuf);
```

Parameters:

Parameter:	Description:
`handle`	Must be a valid handle returned from OC_OpenScanner
`outimgcpy`	If `outimgcpy` is: NULL OC_WriteOutputImage writes data directly to the output image in the scanner's shared memory; if the update mode is OUTUPD_CHANGE, the scanner is signalled to update the outputs not NULL OC_WriteOutputImage writes data to `outimgcpy`; the output image data is not affected If file integrity is not necessary, the host application can set `outimgcpy` to NULL.

`slotnum`
`offset`
`len`
	If `slotnum` is positive, the output image for that slot is written from the array pointed to by `imagebuf`. Then `len` words of output data starting at word `offset` are written to the module in the slot, and will not write past the end of the output image for the slot.
	If `slotnum` is -1, the entire output image is written from the array pointed to by `imagebuf`, and `offset` and `len` are ignored.

`imagebuf`	Must point to an array that is large enough to accept the amount of data from the requested output image

Description:

To guarantee that changes to the output image resulting from a series of calls to OC_WriteOutputImage are posted to the I/O modules in a single output scan, OC_WriteOutputImage can be called to modify a local copy of the output image, then finally called to write the entire copy of the output image to the scanner's shared memory. This preserves output image file integrity across multiple calls to OC_WriteOutputImage.

Return Value:

Name:	Value:	Description:
SUCCESS	0	output image was written successfully
ERR_OCACCESS	2	`handle` does not have access to scanner
ERR_OCPARAM	8	parameter contains invalid value
ERR_OCSCANCFG	14	scanner has not been configured
ERR_OCSLOT	12	slot number is invalid

Considerations:

Supported in the DOS API library and the Windows NT API library

Example:

```
HANDLE    Handle;
WORD      outputdata[2];
int       retcode;

   outputdata[0] = 0x55AA;
   outputdata[1] = 0xAA55;

   retcode = OC_WriteOutputImage( Handle, NULL, 6, 0, 2, outputdata );
```

OC_WriteSRAM

OC_WriteSRAM wires data to the battery-backed user memory

Syntax:

```
int OC_WriteSRAM(HANDLE Handle, BYTE*bufptr, DWORD offset, DWORD length);
```

Description:

The battery-backed memory may be used to store important data that needs to be preserved in the event of a power failure. The size of the available memory in bytes may be obtained using OC_GetDeviceInfo.

Important: It is recommended that the integrity of data stored in the user memory be verified by some means. The OC_CalculateCRC function may be used to generate a 16-bit CRC that may be used for this purpose.

`handle` must be a valid handle returned from OC_OpenScanner. `bufptr` points to the data to be written. `offset` specifies the offset within the memroy to begin writing. `length` specifiies the number of bytes to be written.

If *`offset`* + *`length`* points past the end of the memory, no bytes will be written and ERR_OCPRAM will be returned.

Return Value:

Name:	Description:
SUCCESS	data was written successfully
ERR_OCACCESS	`handle` does not have access to scanner
ERR_OCPARAM	`offset+length` points past the end of the memory

Considerations:

Supported in the Windows NT API library only.

Example:

```
Byte   buf[100];                /*buffer of important data*/
Word crc;
HANDLE handle;

/* Save 100 bytes of data to offset 0 in SRAM */
OC_WriteSRAM(handle, buf, 0, 100);
/* Calculate CRC */
OC_CalculateCRC(buf, 100, &crc);
/* Write data CRC to offset 100 in SRAM */
OC_WriteSRAM(handle, &crc, 100, 2);
```

Error Codes

Introduction

This appendix describes the error code data.

Error Code Returned by API Functions

Most of the API functions return values (see chapter 6). These are the values returned by the API functions to indicate success or possible error conditions (but not all are returned by each function):

Name:	Return Value:	Description:
SUCCESS	0	function returned successfully
ERR_OCACCESS	2	handle does not have access to scanner
ERR_OCEXTERR	11	scanner responded with an extended error message
ERR_OCFAULT	13	request denied because scanner faulted
ERR_OCFILERROR	19	error occurred while reading/writing disk file
ERR_OCINIT	5	scanner has not been initialized
ERR_OCIOCFG	9	I/O configuration is invalid
ERR_OCMEM	3	shared memory cannot be found
ERR_OCNOFORCES	15	no forces installed, cannot enable forces
ERR_OCOPEN	1	scanner already open
ERR_OCOUTOFMEM	17	memory allocate failed
ERR_OCOVERRUN	16	DII, I/O initialization, or error report message overrun
ERR_OCPAR	6	initialization failed due to invalid partition information
ERR_OCPARAM	8	parameter contains invalid value
ERR_OCPOST	7	scanner POST failure
ERR_OCREINIT	4	scanner has already been initialized
ERR_OCRESPONSE	10	scanner did not respond to request successfully
ERR_OCSCANCFG	14	scanner I/O configuration not downloaded
ERR_OCSLOT	12	slot number is invalid
ERR_OCUNKNOWN	18	unknown module type
ERR_OCNOTSUPP	20	function not supported on this platform

Extended Error Codes

The OC_GetExtendedError function returns error information in a structure of type OCEXTERR. This structure is five bytes in length and contains this information:

Table A.1
OCEXTERR Structure

Buffer Offset:	Bytes:	Description:
0	1	extended error code
1	1	associated slot or file number
2	3	error code data

When the scanner encounters an error, the extended error code and associated slot (if any) is written to the extended error code and slot number fields. Error-dependent information is written to the remaining field and the scanner goes to Idle mode. The extended error codes that can be reported by the scanner are:

Table A.2
Extended Error Codes

Extended Error Code:	Description:
0x00	No errors present
0x01	Downloaded directory file is invalid
0x08	Internal software error
0x12	Downloaded configuration is corrupted
0x21	Power fail on expansion rack occurred
0x2E	Invalid DII input slot
0x40	Host Watchdog Timeout
0x50	Data error while accessing module
0x51	Stuck PINT error
0x52	Module is missing
0x53	Module detected in unused slot
0x54	Module type does not match downloaded configuration
0x55	Module I/O mix does not match downloaded configuration
0x56	Rack type does not match downloaded configuration
0x57	Specialty I/O module lock memory command timeout
0x58	Specialty I/O module fault
0x59	Specialty I/O module command timeout
0x5A	Module interrupt problem
0x5B	G file configuration error
0x5C	M0/M1 file configuration error
0x5D	Unsupported interrupt service requested by module
0x5E	I/O driver error
0x60 - 0x8F	Module specific errors (see I/O module documentation)
0x90	MINT occurred on disabled slot
0x91	PINT occurred on disabled slot
0x93	Unsupported module error
0x94	Module has been inserted or reset
0xF0	Internal scanner error

The error code data provides information specific to the cause of all extended errors, except 0xF0 (see Table A.3). The first byte of the error code data contains a subsystem identifier.

Table A.3
First Byte of Error Code Data - Scanner Communications Subsystem Error Codes

Subsystem ID:	Description:
0x06	Scanner Communications
0x08	Backplane Interface (module I/O)

The second byte of the error code data provides details about the subsystem identifier.

Table A.4
Second Byte of Error Code Data - Scanner Communications Subsystem Error Codes

Error Code:	Description:
0x00	Download error - invalid directory file pointer block
0x01	Download error - change to Idle mode failed
0x02	Download error - configuration file CRC failed (byte 1 contains the file number)
0x03	Download error - directory CRC failed
0x04	Download error - input or output image exceeds partition allocation (byte 1 contains 0 for output image, 1 for input)

The third byte of the Error Code Data provides details about the Subsystem Error Codes.

Table A.5
Third Byte of Error Code Data - Backplane Interface Subsystem Error Codes

Error Code:	Description:
0x01	Rack configuration verify error
0x02	DII configuration error
0x03	I/O error occurred while updating slot enables
0x06	I/O error occurred while polling for PINT
0x07	Bad module mode change attempted
0x08	Slot error occurred during mode change
0x09	Source of module interrupt not found
0x0A	Corrupted directory file detected when going to run mode
0x0C	I/O error occurred while servicing module interrupt
0x0D	Module interrupt from disabled slot
0x0E	Interrupting module requested unsupported service
0x0F	I/O error while performing input scan
0x10	I/O error while performing output scan
0x11	Verified read or write error

If the extended error code is an internal scanner error (0xF0), the slot number is set to 0 and the first byte of the Error Code Data contains the source of the error. Table 9 lists the internal scanner extended error sources.

Table A.6
Internal Scanner Errors

Error Code:	Description:
0x03	Scanner message queue full
0x06	Internal scanner watchdog timeout
0x07	CRC checksum failure
0x09	Invalid message
0x0A	RAM failure
0x0B	Host watchdog timeout

Testing Function Calls

Introduction

Both the DOS API and the Windows NT API come with a utility program called `api_test.exe`. This interactive program lets you execute, from the keyboard, every API call for the 1746 I/O PCI Interface. Use the source code of the utility program, along with the executable program, to test different argument values for each function call and to verify correct scanner operation.

Another DOS utility program called `ocdiag.exe` comes with the 1746 I/O PCI Interface hardware. Use this utility program to verify:

- hardware operation
- scanner functionality
- I/O control

The Windows NT utility program is `oc_nt.exe`. To get a copy of this utility:

- contact A-B Technical Support Services at 440-646-6800

  or

- download a copy from the Technical Support Services bulletin board at 440-646-5441

Use the `-d` option when executing the self-unzipping file so that you preserve subdirectory structures.

Notes:

Allen-Bradley
Publication Problem Report

If you find a problem with our documentation, please complete and return this form.

Pub. Name **API Software for 1746 I/O PCI Interface**

Cat. No. **1747-PCIDOS, -PCINT** Pub. No. **1747-6.5.3** Pub. Date **June 1998** Part No. **955132-74**

Check Problem(s) Type:	Describe Problem(s):			Internal Use Only
☐ Technical Accuracy	☐ text	☐ illustration		
☐ Completeness What information is missing?	☐ procedure/step ☐ example ☐ explanation	☐ illustration ☐ guideline ☐ other	☐ definition ☐ feature	☐ info in manual (accessibility) ☐ info not in manual
☐ Clarity What is unclear?				
☐ Sequence What is not in the right order?				
☐ Other Comments Use back for more comments.				

Your Name _____ Location/Phone _____

Return to: Marketing Communications, Allen-Bradley Co., 1 Allen-Bradley Drive, Mayfield Hts., OH 44124-6118 Phone: (440) 646-3176
FAX: (440) 646-4320

Publication ICCG-5.21-August1995 PN955107-82

PLEASE FASTEN HERE (DO NOT STAPLE)

Other Comments

PLEASE FOLD HERE

PLEASE REMOVE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 18235 CLEVELAND OH

POSTAGE WILL BE PAID BY THE ADDRESSEE

Rockwell Automation

Allen-Bradley

1 ALLEN BRADLEY DR
MAYFIELD HEIGHTS OH 44124-9705

Rockwell Automation

Allen-Bradley

Allen-Bradley, a Rockwell Automation Business, has been helping its customers improve productivity and quality for more than 90 years. We design, manufacture and support a broad range of automation products worldwide. They include logic processors, power and motion control devices, operator interfaces, sensors and a variety of software. Rockwell is one of the world's leading technology companies.

Worldwide representation.

Argentina • Australia • Austria • Bahrain • Belgium • Brazil • Bulgaria • Canada • Chile • China, PRC • Colombia • Costa Rica • Croatia • Cyprus • Czech Republic • Denmark • Ecuador • Egypt • El Salvador • Finland • France • Germany • Greece • Guatemala • Honduras • Hong Kong • Hungary • Iceland • India • Indonesia • Ireland • Israel • Italy • Jamaica • Japan • Jordan • Korea • Kuwait • Lebanon • Malaysia • Mexico • Netherlands • New Zealand • Norway • Pakistan • Peru • Philippines • Poland • Portugal • Puerto Rico • Qatar • Romania • Russia-CIS • Saudi Arabia • Singapore • Slovakia • Slovenia • South Africa, Republic • Spain • Sweden • Switzerland • Taiwan • Thailand • Turkey • United Arab Emirates • United Kingdom • United States • Uruguay • Venezuela • Yugoslavia

Allen-Bradley Headquarters, 1201 South Second Street, Milwaukee, WI 53204 USA, Tel: (1) 414 382-2000 Fax: (1) 414 382-4444

Publication 1747-6.5.3 – June 1998

PN 955132-74