PC CARD STANDARD

Volume 11

XIP Specification

Revision History

Date	XIP Specification Version	PC Card Standard Release	Revisions
11/90	1.0	PCMCIA 1.0	Initial definition
09/91	2.0	PCMCIA 2.0/JEIDA 4.1	Defined XIP Application Programming Interface (API)
11/92	2.01	PCMCIA 2.01	None
07/93	N/A	PCMCIA 2.1/JEIDA 4.2	None
02/95	5.0	February 1995 (5.0) Release	Extended and reformatted
03/95	N/A	March 1995 (5.01) Update	None
05/95	N/A	May 1995 (5.02) Update	None
11/95	N/A	November 1995 (5.1) Update	None
05/96	N/A	May 1996 (5.2) Update	None
03/97	6.0	6.0 Release	None
04/98	N/A	6.1 Update	None
02/99	7.0	7.0 Release	None
03/00	7.1	7.1 Update	None
11/00	7.2	7.2 Update	None
04/01	8.0	8.0 Release	Changed to PC Card Standard Volume Number 11 (was Volume Number 9)

CONTENTS

TABLES

1. INTRODUCTION

1.1 Purpose

In order to achieve savings of both RAM and ROM[1], it is beneficial to directly execute applications from ROM, without loading the image of the application into RAM prior to execution. A number of machines currently offer this capability; however, there is no application-level portability between these machines. This specification outlines a standard method, hereinafter referred to as eXecute In Place, or XIP, of achieving this.

This document shall describe the Metaformat tuples, data structures, driver architecture, and the Application Programming Interface (API) for eXecute In Place (XIP), as well as the architecture and load format of an XIP-compliant application.

1.2 Scope

This document is intended to provide enough information for software developers to design and implement XIP applications. It is also intended to provide sufficient information to allow an implementor to create an XIP implementation.

1.3 Related Documents

This section identifies documents related to the eXecute In Place Interface Specification. Information available in the following documents is generally not duplicated within this document.

The following documents which comprise the ***PC Card Standard***:

PC Card Standard Release 8.0 (April 2001), PCMCIA /JEITA

Volume 1. ***Overview and Glossary***
Volume 2. ***Electrical Specification***
Volume 3. ***Physical Specification***
Volume 4. ***Metaformat Specification***
Volume 5. ***Card Services Specification***
Volume 6. ***Socket Services Specification***
Volume 7. ***PC Card ATA Specification***
Volume 8. ***PC Card Host Systems Specification***
Volume 9. ***Guidelines***
Volume 10. ***Media Storage Formats Specification***
Volume 11. ***XIP Specification***

PCMCIA Card Services Interface Specification, Release 2.00, November 1992, PCMCIA

PCMCIA Socket Services Interface Specification, Release 1.01, September 1991, PCMCIA

PCMCIA Socket Services Interface Specification, Release 2.00, November 1992, PCMCIA

[1] It will be seen that the physical media an XIP application is built into generally has little impact upon the application. Thus, for the purposes of this document, unless otherwise noted, the term ROM will apply also to Flash memory, EEPROM, etc.

1.4 Data Sizes

The Version 1.0 XIP Specification was oriented primarily towards X86 architectures. The data structures in this document are an evolution of those specified in that earlier document. As such, the data sizes in the table below hold true within data structures. Data organization is "Little Endian."

Data type	**Bits**
char	8
short int	8
int	16
long int	32

Data sizes for passed parameters within API calls are functions of the particular Operating System Binding. In general, they tend to follow those in the above table.

2. OVERVIEW

Unlike other sections of the ***PC Card Standard***, this specification is, to a very large extent dependent upon the host processor class, and upon the host operating system. Thus, much of the detail to be expected in a document such as this must be relegated to application notes within appendices, each specific to a particular operating system and processor chip class.

2.1 SXIP, LXIP and EXIP

Three types of XIP support are defined in order to support three real-world architectures: LXIP, SXIP, and EXIP.

LXIP is for systems where demand-paging is required (i.e., pages not in memory must be explicitly paged in by software at some level). LXIP Applications are structured to operate in a 16KB paged-execution environment.

SXIP (Simple XIP) is for those systems which have only very limited paging mechanisms. SXIP applications are comprised of an execution image of at most 64K of code and / or read-only data, and are monolithic in nature. These applications do no overlaying of any sort. The SXIP concept has been added to this revision to allow execution of XIP on as wide a variety of systems as possible.

EXIP is for those systems with very large address spaces or with implicit paging (i.e., pages not in memory when accessed are placed into memory without intervention at a software level). EXIP applications are structured to operate in an environment where no paging is necessary, similar to an Intel 80386 extended-addressing-mode-execution environment.

These differences have no effect on the Metaformat, XIP data structures, or driver architecture. They are noticeable in the API described later in this document. There is significant difference in the hardware support required, and in the way applications are structured in the respective environments. There may also be a significant impact within particular Operating System Bindings.

2.1.1 Hardware support for EXIP

Hardware support for Card Services in protected mode is necessary and sufficient for EXIP support.

2.1.2 Hardware support for LXIP

Hardware support for Card Services in real mode is necessary and sufficient for LXIP support.In addition to support for real mode Card Services, an XIP-compliant platform must provide, at minimum, one contiguous window of at least 64K. This window must be capable of partitioning into 4 16K pages, each of which must be independently mappable. Additional windows and pages may be utilized if provided.

2.1.3 Hardware support for SXIP

Hardware support for Card Services in real mode is necessary and sufficient for SXIP support.

3. XIP PARTITIONS

XIP applications are stored, by definition, on XIP partitions. XIP partitions are used only to store XIP applications, and are entirely distinct from any other partitions, such as FAT or Flash file-system partitions. In order to ensure that XIP applications may be properly mapped into system memory, any XIP partition must begin on a 16K boundary, relative to the start of the card. An XIP partition is required to be an integral number of pages long, and each page of an XIP partition is also required to be 16K in length. If an XIP partition is not aligned on erase block or device boundaries, data outside of the XIP partition may be destroyed if the XIP_ERASE_PARTITION function is invoked.

3.1 XIP Partition Identification

There are two tuples relevant to XIP. The "Format Tuple (CISTPL_FORMAT)" defines the data-recording format for a card as well as the location and size of the associated memory region on the card. The "Organization Tuple (CISTPL_ORG)" defines the organization of the data in a specific partition. This tuple must follow a format tuple to be associated with it.

The Format Tuple for an XIP partition should have a TPLFMT_TYPE field value of TPLFMTTYPE_MEM. An error detection method may be specified. The TPLFMT_OFFSET and TPLFMT_NBYTES fields define the location and size of the XIP partition. The TPLFMT_FLAGS should be set to 0. The TPLFMT_ADDRESS field is not used, and should be set to 0.

The Organization Tuple for an XIP partition should have a TPLORG_TYPE field value of TPLORGTYPE_ROMCODE. This is somewhat misleading, as it suggests that the XIP partition is read-only; however, writing to areas of the partition is allowed when the card technology supports write access. The TPLORG_DESC field should have an appropriate value; the only value currently defined is "DOS_XIP" for DOS-compatible XIP partitions.

3.2 XIP Partition Structure

Within the XIP partition, a format for the XIP application images needs to be defined in order that the XIP manager can locate the XIP images it is to manage and map. This format is comprised of an XIP header and an XIP Directory. Based on the size of the data structure used to describe each entry of the XIP directory, and its offset within the first 16K block of the partition, up to 511 XIP applications could reside within a single XIP partition. The XIP directory is not required to be 16K in size, but it must be located wholly within the first 16K block.

3.2.1 XIP Partition Header Structure

An XIP header is located at the beginning of the XIP partition. The header is structured as:

```
struct XIP_partition_header {
    int          max_directory_entries;
    long int     xip_serial_number;
    int          data_structure_version_number;
    char         xip_header_reserved[24];
}
```

The max_directory_entries field contains the maximum number of possible entries within the XIP directory. The size of the directory structure may be inferred from this.

The XIP_serial_number shall be set on creation of an XIP partition only. It is analogous to a DOS volume serial number.

The data_structure_version_number field shall be set to 0002H for this release.

All bytes within the xip_header_reserved array shall be set to 0FFH.

The total length of the directory for an XIP partition can be determined by:

directory length = (max_directory_entries) * 32 bytes.

The XIP partition is divided into an integer multiple of 16K fixed-length pages, where each 16K page is aligned on 16K boundaries. Each 16K region is assigned a page number. The relationship between page n and the byte offsets, relative to the beginning of an n-byte-long XIP partition, is illustrated below.

16K Page Number	Byte Offset relative to the Beginning of an XIP partition	Page Usage
0	00000h...03FFFh	XIP Header and Directory Structure
1	04000h...07FFFh	XIP Applications
2	08000h...0BFFFh	" "
3	0C000h...0FFFFh	" "
4	10000h...13FFFh	" "
.	.	" "
.	.	" "
.	.	" "
((n/4000h)-1)	(n-4000h)...(n-1)	" "

3.2.2 XIP Directory Structure

XIP directory entries are stored as an array of fixed-length structures. The layout of an XIP directory entry is similar to that of a DOS FAT-file-system-directory entry.

The XIP driver uses the XIP directory to determine the number of applications which are present, their names and extensions, time and date of creation, and their locations within the partition. The XIP directory starts immediately after the XIP header.

An XIP application occupies some number of contiguous 16KB pages. All XIP application entries are allocated from the beginning of the XIP directory, i.e., all directory entries are allocated contiguously and all XIP pages are allocated from the beginning of the available XIP memory pages in the partition.

An XIP driver can determine the next unallocated memory page and directory entry by sequentially searching the XIP directory and finding the first entry that has never been allocated (i.e., XIP_STATUS = 0xxxxx111b). This directory entry is available for allocation, and the previous directory entry contains the information necessary to determine the next available XIP memory page.

The XIP driver may need to read the directory at driver initialization time and whenever a card with an XIP partition is installed in a slot in order to determine any information it needs to operate properly. The XIP driver will also read the "directory" at driver run time in order to support XIP functions that access the XIP directory. If the type of memory making up the XIP partition permits it, the XIP driver may be able to add new XIP applications into the partition. ROM XIP partitions cannot support adding new XIP applications; Flash or EEPROM XIP partitions could support this functionality.

Byte	Name
0...7	XIP_NAME
8...10	XIP_EXT
11	XIP_STATUS
12...13	XIP_APP_BEGIN
14...15	XIP_APP_OFFSET
16	XIP_APP_TYPE
17	XIP_VERSION_REQD
18..20	XIP_RESERVED
21	XIP_HEADER_CKSUM
22...23	XIP_CREATION_TIME
24...25	XIP_CREATION_DATE
26...27	XIP_FIRST_APP_PAGE
28...31	XIP_SIZE

Bytes 0-7 (XIP_NAME):

Specifies an ASCII string that is the primary name of this XIP application. The format of this name, in combination with the next field, XIP_EXT, is the same as the DOS primary-file name: 8 characters followed by three characters. If the name is less than eight (8) characters, it must be padded on the right with blanks (20h).

Bytes 8-10 (XIP_EXT):

Specifies an ASCII string that is the name extension of this XIP application. The format of this name is the same as the DOS file-name extension: 0 to 3 characters. If the extension is less than three (3) characters it must be padded on the right with blanks (20h).

Byte 11 (XIP_STATUS):

Specifies the status of this XIP directory entry (see table below). Values for the status bits are chosen so that an XIP directory may be updated in media like flash memory, without first erasing and then re-recording it. Flash memory has unique constraints placed on how bit values in it may change.

If XIP_STATUS has a value of 0FFh, the previous XIP directory entry was the last one in the XIP partition. New entries in the XIP directory may be made at this point in the directory. If the first byte in the first directory entry has a value of 0FFh, the XIP directory is completely empty. If the XIP directory is completely empty, the first page occupied by the directory may be partially available, the last page may be partially available, and all remaining pages of the partition are available for allocation. Whether there are any partially-available pages is determined by the total size and location of the partition, and the size of the XIP directory. The size and physical location of the XIP partition within the card's physical-address space, is determined by the tuple data structures.

If XIP_STATUS indicates a deleted entry, the XIP_FIRST_APP_PAGE and XIP_SIZE are still necessary for managing which pages may be allocated to new XIP applications that are being copied into this partition. Also note that the XIP pages that have been deleted are not reusable because they have not been erased. When an entry is marked deleted, the name and extension fields of the directory entry must be ignored, since they are not required to be cleared.

If bits 0...2 of the XIP_STATUS are set to 001, than bits 3 and 4 will indicate the structure of the application. A value of 10 within bits 3 and 4 is invalid.

Bit	Definition
0...2	XIP Allocation Status: 111 XIP Directory entry is free to be used to store the next XIP directory entry. The previous XIP directory entry is the last valid entry in they XIP directory. 011 XIP directory entry and the contents of its associated XIP application are in the process of being created. Since this XIP directory entry has not yet been completely processed, the application may not be loaded, as the XIP application code has not been completely copied. However, the XIP_FIRST_APP_PAGE and XIP_SIZE fields are valid for this directory entry. An XIP utility program must "close" the XIP directory entry before the application may be loaded. 001 XIP directory entry is allocated and contains a valid XIP application. 000 XIP directory entry had been allocated, but is now deleted. It no longer contains a valid XIP application. The fields XIP_FIRST_APP_PAGE and XIP_SIZE contain valid data that must be sued when searching for the next free page within the XIP partition.
3...4	LXIP/EXIP Application 00 The application is structured for LXIP 01 The application is structured for EXIP 11 The application is structured for SXIP
5...7	Reserved for future use. Must be all ones.

Bytes 12-13 (XIP_APP_BEGIN):

Specifies the offset in the first page for the application's entry point. The offset shall be located on a 16-byte boundary.

Bytes 14-15 (XIP_APP_OFFSET):

Specifies the offset in the first page of the application for the application's first byte. The offset shall be located on a 16-byte boundary.

Byte 16 (XIP_APP_TYPE):

Specifies the XIP application type. A list of application types are to be found in the table below; however, future operating system bindings will cause this list to expand.

XIP_APP_TYPE	Name	Description
0	XIP_TYPE_NON_EXEC	Non-executable. Examples of XIP Type 0 include pseudo-volume labels, XIP V1.00 applications, etc.
1	XIP_TYPE_SXIP_DOS	Simple executable image for execution under DOS. An XIP Type 1 application does no overlaying, and can run under an SXIP driver.
2	XIP_TYPE_V2_OVERLAY_DOS	Overlaid executable image for execution under DOS. An XIP Type 2 application is suitable for overlaid execution under DOS. Executable image formats are specified in the appropriate Operating System Binding Appendix.

Byte 17 (XIP_VERSION_REQD):

Specifies the version of XIP API Services required for execution.

Bytes 18-20 (XIP_RESERVED):

These bytes are reserved for future use. All reserved bytes must be set to a value of 0FFh.

Byte 21 (XIP_DIREC_CKSUM):

Specifies an 8-bit additive checksum of the 32 bytes of this XIP directory entry, exclusive of this byte. Note that, in order to maintain compatibility with XIP Version 1.0, especially in cases where bytes 16 or 17 are 0FFh, 0FFh must be considered a valid entry, even when a different value is reached via calculation.

Bytes 22-23 (XIP_CREATION_TIME):

Specifies the time that this XIP directory entry was created, or added, to the XIP partition. For media such as flash memory, it is not possible to "update" an XIP directory entry without first erasing the entire XIP partition. Therefore, the time only refers to the time that the XIP application was added to the XIP directory. The format of the create time is DOS compatible and is described in the following table.

Bytes 24-25 (XIP_CREATION_DATE):

Specifies the date that this XIP directory entry was created, or added, to the XIP partition. For media such as flash memory, it is not possible to "update" an XIP directory entry without first erasing the entire XIP partition. Therefore, the date only refers to the time that the XIP application was added to the XIP directory. The format of the create date is DOS compatible and is described in the following table.

Bits	Definition
0...4	Day of month (1-31)
5...8	Month (1-12)
9...15	Year (relative to 1980)

Bytes 26-27 (XIP_FIRST_APP_PAGE):

Specifies the first 16K page within an XIP partition that is allocated to this XIP application. Pages are allocated contiguously within an XIP partition. There is nothing which precludes the possibility of having multiple XIP applications within a single page.

Bytes 28-31 (XIP_SIZE):

Specifies the size, in bytes, of this XIP application. This size must include any padding required to achieve any application-specific alignment. The size may be zero to allow an entry to simply be a "label."

4. XIP DEVICE DRIVER ARCHITECTURE

XIP functions are to be provided by a high-level device driver. The intent of this specification is that this driver be reliant upon Card Services for lower level functionality. Card Services is in turn dependent upon Socket Services. The XIP device driver simply manages the data in the XIP partition, and provides the required services. Hardware-related services, such as those needed to manipulate mapping hardware, are intended to be provided by Card Services. However, especially for systems with limited resources, there is no intent to prohibit an XIP driver which does not rest upon any underlying level of software.

XIP Device Drivers must be able to provide all functionality listed below as Common, and are assured of being able to execute SXIP applications. Additionally, they may provide services listed below as elements of the Write, LXIP, or EXIP functionality sets.

4.1 Migration Path of Drivers

Implementations of this architecture are expected to tend to migrate into BIOS functions over time. The layered architecture will initially be implemented as loadable drivers. Some manufacturers may initially provide all three drivers (XIP, Card Services, and Socket Services), or combinations of two of the three, as a single unit. In any case, the location and partitioning of the drivers should have no impact on XIP applications, as the applications should be unable to determine the location and/or type (loaded or BIOS-resident) of the drivers, nor will the applications be able to distinguish between a joint driver and completely separate drivers.

4.2 Sharing the Hardware Interface Between Device Drivers

It is possible that other device drivers or embedded software, distinct from the XIP driver, will need to access the memory-mapping-interface hardware that maps memory on the card into the system's address space. It should be obvious that such a memory-mapping interface is required for XIP. However, a memory-mapping interface would be perfectly acceptable for a disk device-driver as well. The same hardware interface could be used for XIP driver, a DOS FAT-file-system driver, and a DOS Flash-file-system driver.

Consequently, it is very important for system designers to provide the ability to read, as well as write, the state of their memory-mapping hardware. Write-only memory-mapping-hardware registers are not acceptable. The reason for this requirement is that a disk-device driver, for instance, would necessarily have to save and restore the state of the memory-mapping hardware, before and after a disk access, as the mapping hardware may be in use by an XIP driver and XIP application, as well as other drivers using the memory-mapping resource.

Note: An XIP driver shall not do saves/restores of its mapping hardware between XIP function calls!

The virtualization that Card Services provides should alleviate much of the concern that multiple and separate device drivers accessing the same hardware might cause.

4.2.1 Device Driver Load Order

The XIP and Card Services device drivers are to be loaded like any other device driver during initial start-up processing.

For the XIP drivers, any Socket Services and Card Services drivers are to be loaded before the XIP driver. During initialization of the XIP device driver, information about the system's mapping capabilities, which can only be provided by these drivers, is required. There is no requirement that Memory Technology Drivers be loaded before the XIP driver.

4.3 XIP Loader

The XIP loader is responsible for preparing the system for the execution of an XIP application. The function of the XIP loader is to look up, map, and start the XIP application execution. Naturally, the XIP loader will perform different functions based on the XIP_APPLICATION_TYPE of the application, as well as the operating system environment.

4.4 XIP IOCTL References

The XIP device driver requires that applications use general I/O Control calls (IOCTL) to get and/or set the entry point for the XIP device driver. This allows an arbitrary chain of applications to monitor or patch the device driver. The entry point of the XIP device driver is actually the procedure that an application calls whenever it needs to call the XIP API (Application Programming Interface). This being the case, it is required that all XIP device drivers support IOCTL open, read, and write, and close calls, if the operating system supports such. No other IOCTL calls should be assumed.

4.5 XIP Driver Chain

In the case that the XIP Device Driver does not supply a particular functionality set, additional device drivers may chain in to provide such sets. An application should not be required to differentiate in any way which driver provides which services; in fact, the combination of XIP device drivers should appear monolithic to the application.

For a device driver to chain in, it must first determine the existing entry point in the same manner as any client. Having saved the current XIP entry point, it may perform an IOCTL write to set up the new entry point.

When a chained device driver is called, it should inspect the function parameters to determine if it needs to handle the call. If not, it should pass control to the older XIP device driver.

5. XIP APPLICATIONS PROGRAMMING INTERFACE

In order for an application to use the XIP device driver, the device driver must obviously be present and installed, and the XIP application must be able to determine both the presence and entry point of the driver. Furthermore, additional device drivers should be allowed to chain into the current driver to provide functionality not granted by the original driver. The specific mechanics of interacting with the XIP device driver are, of course, dependent on the Operating System Binding; however, the following generalities should be observed.

5.1 XIP API Calling Interface

The XIP device driver uses a procedure-call interface rather than a software-interrupt interface. The callback interface, being private, means that other software, not directly related to the operation of the XIP device, will not be chained into the execution path. The benefit of this interface is that there are no software interrupt-chaining-compatibility problems.

Whether passed arguments are register-based or stack-based is a detail of the Operating System Binding.

5.1.1 Initializing the XIP Interface

In order for an XIP client to use the XIP device driver, the client needs to know the entry point for XIP services within the device driver. Generally, a client should request that the operating system open a device with the name "XIP$$$$$." The client should then request that the operating system read the XIP entry point from that device, and then close that device. If any errors occur in the course of the above steps, it will probably be necessary to install or re-install the XIP device driver before further processing.

5.1.2 Calling the XIP API

Having read the entry point of the XIP device driver, calling the XIP API from a client involves a simple two-step process. First, load the entry point previously read from the XIP device. Second, call the address pointed to by that address. This secondarily indirect address is the XIP API entry point. Operating System Bindings should probably contain examples to make this process clear.

The second level of indirection adds flexibility in being able to both chain into and unchain from an XIP device driver. Furthermore, all future clients are immediately affected by changes in the device chain, with no special effort on their part.

5.2 XIP API Functions

The following functions comprise the entire set of XIP functions that are required to be present within a fully compliant XIP version 1.10 driver.

Each function is listed below in the following form:

```
XIP (*FUNCTION, parameters)
```

The details of how the parameters map to the calling sequence (i.e., parameter order, register based parameters, et. al.) is an Operating System Binding and implementation issue.

Each function implicitly returns it's result as a binary success(true) / fail(false) indicator. Other return values, and the meanings thereof, hinge on the function result. If the function fails (returns FALSE), the FUNCTION returns a status code, and the other variables are undefined, unless otherwise noted. If the function is successful (return TRUE), the FUNCTION variable is undefined.

Also note that variables may "overlay" one another; i.e., a variable with one meaning on input may have another meaning on output, or a variable may have different meanings or structure dependent upon the contents of other variables.

5.2.1 Get XIP Version (Common)

```
XIP (*function, *version, *functionality)
```

Purpose:

This function returns the version of the XIP driver installed in the system, as well as a bit flag indicating the capabilities of the system. An application uses this function to determine if the set of XIP functions it requires are supported by this XIP driver.

Calling Parameters:

function	XIP_GET_VERSION

Results if successful:

version	Contains the XIP driver's version number in binary coded decimal (BCD) format. The upper four bits contain the integer digit of the version number. The lower four bits contain the fractional digit of version number. For example, version 1.10 is represented as 010Ah. When checking for a version number, an application should check for a version number or greater. Vendors may use the fractional digit to indicate enhancements or corrections to their XIP drivers. Therefore, to allow for future versions of XIP drivers, an application shall not depend on an exact version number.
functionality	A bitfield indicating which XIP functionality set(s) are supported by this driver. Values are as in the table below.

Bits	Description
7...3	Reserved (Set to 0)
2	Supports Write functionality set (if 1)
1	Supports EXIP functionality set (if 1)
0	Supports LXIP functionality set (if 1)

Status Codes:

XIP_STAT_UNKNOWN_ERROR

5.2.2 Get XIP Mappable Segments (SXIP, LXIP)

```
XIP (*function, *XIP_mappable_array, *mappable_array_length)
```

Purpose:

This function returns an array of addresses for each mappable XIP page in a system. The array is sorted in ascending segment order.

Calling Parameters:

function	XIP_GET_MAP_SEGS
XIP_mappable_array	An un-initialized array. The first element of the array gives the acceptable length of the array, not including the first element (thus a first word of 0, although legal, would cause no change to the contents of this array). The remainder of the array will be filled in by the function.

Results if successful:

XIP_mappable_array	The array will have been filled in with the segment address of each mappable page in the system, up to the number of entries allowed in the array. The array will be sorted in ascending order, with the length of the array first, the lowest-valued page first, etc. SXIP environments will have only one entry in the array.
mappable_array_length	Actual number of mappable pages in the system. If an application wished to be precise, it could call this function with *XIP_mappable_array set to 0, allocate mappable_array_length+1 words of memory, then call this function again with XIP_mappable_array set to the newly allocated memory, and *XIP_mappable_array set to mappable_array_length.

Status codes

XIP_STAT_UNKNOWN_ERROR

5.2.3 Get XIP Partition IDs (Common)

```
XIP (*function, partition_ids_array, *partition_array_length)
```

Purpose:

This function returns an array of XIP partition IDs currently accessible in the system. XIP partition IDs are tokens used to uniquely identify each XIP partition. Token values are implementation specific, and are valid only after this function is called.

Partition IDs are assigned at initialization time by the XIP driver. These IDs are valid only during a given system boot. Thus, the ID for a specific partition may change from boot to boot. Normally, IDs will not be reassigned to a different partition once they have been assigned to a specific partition, however, the Disable Partition ID function can be used to allow a partition ID to be reassigned.

This function will not return a partition ID that has been disabled. However, it may return new partition IDs that correspond to XIP partitions on a newly inserted card.

Calling Parameters:

function	XIP_GET_PARTITIONS
partition_ids_array	An un-initialized array. The first element of the array gives the acceptable length of the array, not including the first element (thus a first word of 0, although legal, would cause no change to the contents of this array). The remainder of the array will be filled in by the function.

Results if successful:

partition_ids_array	The array will have been filled in with the partition IDs of each XIP partition present in the system. No order to the resulting array may be assumed, other than that the length of the array will be first.
partition_array_length	Actual number of partitions in the system. If an application wished to be precise, it could determine the length of the full array, and then allocate and fill the array, analogous to the method given in Get XIP Mappable Segments.

Status codes

XIP_STAT_UNKNOWN_ERROR

5.2.4 Get XIP Handle Range (Common)

```
XIP (*function, *XIP_first_handle, *XIP_last_handle, *total_handles)
```

Purpose:

This function returns the range of handle values that the XIP driver manages. The maximum range of handle values is a detail of any particular XIP Operating System Binding.

The XIP driver assigns a unique handle value, in the specified range, to every XIP application present in the XIP directory. An XIP application uses a handle as a parameter whenever it calls the XIP driver to perform a map, unmap, or copy function.

Calling Parameters:

function	XIP_GET_HANDLE_RANGE

Results if successful:

XIP_first_handle	This is the value of the first handle managed by the XIP driver. Shall be in the range 8000H through 8FFFh, inclusive.
XIP_last_handle	This is the value of the last handle managed by the XIP driver. This value must obviously be greater than that value returned as XIP_first_handle. Shall be in the range 8000H through 8FFFH, inclusive.
total_handles	The maximum number of handles that this XIP driver is capable of supporting.

Status Codes:

XIP_STAT_UNKNOWN_ERROR

5.2.5 Map/Unmap an XIP Handle's Pages (SXIP, LXIP)

```
XIP (*function, handle, partition, *map_count, seg_map_array)
```

Purpose:

This function maps or unmaps one, several, all, or none of the logical pages associated with a handle into as many mappable segments as the system supports. Both mapping and unmapping pages may be done in the same invocation. Mapping or unmapping no pages is not considered an error. If a request to map or unmap zero pages is made, nothing is done and no error is returned. The segment map array passed to this function does not have to have any special order with respect to mappable_segment elements.

The function should allow mapping a specific logical page at more than one mappable segment address, provided that the hardware supports this. This same "feature" is also a part of the LIM specification.

Calling Parameters:

function	XIP_MAP_HANDLE
handle	Handle of the XIP application owning the pages to be mapped in. This handle is obtained through any of the search functions listed below.
partition	Contains the ID of the partition containing the indicated XIP application
map_count	Contains the number of entries in the seg_map_array. This argument is ignored and assumed 1 in SXIP environments.
seg_map_array	An array of seg_map structures, defined as:

```
struct seg_map_struct {
    int  log_page_number   ;
    int  mappable_segment  ;
};
```

log_page_number contains the number of the logical page to be mapped. Logical pages are numbered zero relative with respect to the beginning of the XIP application with which they are associated. Therefore, the number for a logical page can range from zero to one less than the number of logical pages allocated to the handle.

If the logical page number is set to -1, the associated mappable_segment is unmapped. Unmapping a page makes it inaccessible.

mappable_segment contains the segment address within the system memory address space at which the logical page is to be mapped. This segment address must correspond exactly to a mappable segment address, as returned via the XIP_GET_MAP_SEGS function.

Results:

map_count	Contains the number of entries actually mapped, regardless of the function status return. If the function fails, this number of pages has been mapped, and may need to be unmapped by the application. The XIP driver will map the pages sequentially, so if only n pages have been mapped, it may be assumed that the first n pages have been mapped.

Status Codes:

XIP_STAT_HANDLE_NFOUND	The specified XIP handle could not be found, and is probably corrupt. No pages have been mapped.
XIP_STAT_BAD_PAGE	One or more of the logical pages to be mapped is out of the range of logical pages allocated to the given XIP handle.
XIP_STAT_BAD_SEGMENT	One or more of the segment addresses is invalid, or, the map_count specified exceeds the number of mappable segments in the system.
XIP_STAT_BAD_PARTITION	The partition specified does not exist.
XIP_STAT_PAGE_MAPPED	One or more of the logical pages specified is already mapped at a mappable segment. The new logical page specified is not mapped.
XIP_STAT_CARD_CHANGED	
XIP_STAT_UNKNOWN_ERROR	

5.2.6 Get XIP Mapping Context Size (Common)

```
XIP (*function, *map_context_size)
```

Purpose:

The Get XIP Mapping Context Size function returns the storage requirements for the array passed to the XIP_GET_CONTEXT and XIP_SET_CONTEXT functions. The size returned is implementation-dependent, but it is explicitly fixed (i.e., the return value of this function will never change). Clearly, this implies that supplemental XIP device drivers should not intercept this function, or the XIP_GET_CONTEXT and XIP_SET_CONTEXT functions.

Calling Parameters:

function	XIP_GET_CONTEXT_SIZE

Results if Successful:

map_context_size	Contains the size of the block that will be transferred to or from the memory area which an application will supply to the XIP_SET_CONTEXT or XIP_GET_CONTEXT functions.

Status Codes:

XIP_STAT_UNKNOWN_ERROR

5.2.7 Get XIP Mapping Context (Common)

```
XIP (*function, *xip_context)
```

Purpose:

This function saves the mapping context to the specified buffer. The format and content of the resultant buffer is strictly implementation dependent.

Calling Parameters:

function	XIP_GET_CONTEXT
xip_context	A buffer of at least the size given by the XIP_GET_CONTEXT_SIZE function.

Results if successful:

xip_context	The buffer contains the state of the XIP mapping hardware. It also contains any additional information necessary to restore the XIP mapping hardware to the current state when this function is invoked. The content and size of this information is vendor specific.

Status Codes:

XIP_STAT_CARD_CHANGED

XIP_STAT_UNKNOWN_ERROR

5.2.8 Set XIP Mapping Context (Common)

```
XIP (*function, *xip_context)
```

Purpose:

This function restores the physical and logical mapping context for all XIP mappable regions to the state it was in when the specified buffer was filled in via the XIP_GET_CONTEXT function. The format and content of the buffer is strictly implementation dependent.

Calling Parameters:

function	XIP_SET_CONTEXT
xip_context	A buffer that has previously been passed to the XIP_GET_CONTEXT function. The buffer is of at least the size given by the XIP_GET_CONTEXT_SIZE function.

Results if successful:

none

Status Codes:

XIP_STAT_BAD_MAP_ARRAY	The contents of the xip_context buffer have been corrupted, or an invalid pointer has been passed to the function.
XIP_STAT_CARD_CHANGED	
XIP_STAT_UNKNOWN_ERROR	

5.2.9 Search for XIP Directory Entry (Common)

```
XIP (*function, partition, *application_name, *handle, *page_count)
```

Purpose:

This function searches the XIP directory for a specific XIP application. If the name is found, the function returns only the handle associated with the name and the number of logical pages allocated to it.

The XIP handle returned by the XIP driver always specifies the same XIP application within the XIP partition. For example, if you searched the XIP directory six times for an existing XIP application named "WORDPROC.XIP", the XIP driver would return the same handle value each time because the position of the XIP application within the XIP directory structure had not changed.

An XIP handle is somewhat analogous to an index into an array of fixed length structures. The handle returned may be used by as many processes as need access to the XIP application.

Calling Parameters:

function	XIP_SEARCH
partition	The partition ID of the XIP directory to be searched
application_name	The name of the application to be searched for. The name must be in 8.3 format, i.e., an 8-letter name, followed by the 3 letter extension. If either the name is shorter than 8 characters, or the extension is shorter than 3 characters, they must be padded with spaces on the right to the appropriate size.

Results if successful:

handle	The value of the handle assigned to the specified XIP application.
page_count	The number of logical pages allocated to the XIP application. This value can be zero.

Status Codes:

XIP_STAT_APP_NOT_FOUND
XIP_STAT_BAD_APP_NAME
XIP_STAT_BAD_PARTITION
XIP_STAT_CARD_CHANGED
XIP_STAT_UNKNOWN_ERROR

5.2.10 Search for full XIP Directory (Common)

```
XIP (*function, partition, *XIP_dir_entry, *handle, *page_count)
```

Purpose:

This function searches the XIP directory for a specific XIP application. If the name is found, the function returns the full XIP directory information associated with the name, as well as the handle and number of logical pages allocated to the application.

This function is very similar to the XIP_SEARCH function, save that it fills in the entire XIP directory entry if the specified application is found.

Calling Parameters:

function	XIP_SEARCH_FULL
partition	The partition ID of the XIP directory to be searched
XIP_dir_entry	A structure of the form:

```
   struct xip_dir_entry_struct {
       char xip_name[8];
       char xip_ext[3];
       short int  xip_status;
       int  xip_app_begin;
       int  xip_app_offset;
       short int  xip_app_type;
       short int  xip_version_reqd;
       char xip_reserved[3];
       short int  xip_header_cksum;
       int  xip_creation_time;
       int  xip_creation_date;
       int  xip_first_app_page;
       long int   xip_size;
   }
```

The xip_name and xip_ext field must be filled in, and padded on the right with spaces. All other fields will be ignored.

Results if successful:

XIP_dir_entry	The entire structure will be filled in with the appropriate values, copied from the XIP directory.
handle	The value of the handle assigned to the specified XIP application.
page_count	The number of logical pages allocated to the XIP application. This value may be zero.

Status Codes:

XIP_STAT_APP_NOT_FOUND

XIP_STAT_BAD_PARTITION

XIP_STAT_CARD_CHANGED

XIP_STAT_UNKNOWN_ERRO
R

5.2.11 Get First XIP Directory Entry (Common)

```
XIP (*function, partition, *XIP_dir_entry, *handle, *page_count)
```

Purpose:

This function returns the full XIP directory information associated with the first active entry in the XIP directory in the specified partition, as well as the handle and number of logical pages for that XIP application.

Calling Parameters

function	XIP_SEARCH_FIRST
partition	Contains the ID of the partition containing the XIP directory
XIP_dir_entry	An XIP_dir_entry_struct structure. The structure need not be initialized

Results if successful:

XIP_dir_entry	The entire structure will be filled in with the appropriate values, copied from the XIP directory.
handle	The value of the handle assigned to the specified XIP application.
page_count	The number of logical pages allocated to the XIP application. This value may be zero.

Status Codes:

XIP_STAT_APP_NOT_FOUND

XIP_STAT_BAD_PARTITION

XIP_STAT_CARD_CHANGED

XIP_STAT_UNKNOWN_ERROR

5.2.12 Get Next XIP Directory Entry (Common)

```
XIP (*function, partition, *XIP_dir_entry, *handle, *page_count)
```

Purpose:

This function returns the full XIP directory information associated with the next (relative to the given handle) active entry in the XIP directory in the specified partition, as well as the handle and number of logical pages for that XIP application.

Calling Parameters

function	XIP_SEARCH_NEXT
partition	Contains the ID of the partition containing the XIP directory.
XIP_dir_entry	An XIP_dir_entry_struct structure. The structure need not be initialized.
handle	Contains the value of the handle returned from a XIP_SEARCH_FIRST or XIP_SEARCH_NEXT function call. The XIP driver uses this handle value to begin the search for the next active entry in the XIP directory.

Results if successful:

XIP_dir_entry	The entire structure will be filled in with the appropriate values, copied from the XIP directory.
handle	The value of the handle assigned to the specified XIP application
page_count	The number of logical pages allocated to the XIP application. This value may be zero.

Status Codes:

XIP_STAT_HANDLE_NFOUND	The handle passed to the function is invalid. The application has probably corrupted the XIP handle previously obtained.
XIP_STAT_APP_NOT_FOUND	No more active entries were found in the XIP directory.
XIP_STAT_BAD_PARTITION	
XIP_STAT_CARD_CHANGED	
XIP_STAT_UNKNOWN_ERROR	

5.2.13 Add XIP Directory Entry (Write)

```
XIP(*function, partition, *XIP_dir_entry, *handle)
```

Purpose:

This function allows one to create a new XIP directory entry, most probably in the process of adding a new XIP application to the XIP partition. One would use this function to name the XIP application, and allocate space for it within the XIP partition. One would subsequently map the allocated pages into memory, and then copy the new XIP application into them.

Calling Parameters:

function	XIP_ADD_ENTRY
partition	Contains the ID of the partition containing the XIP directory.
XIP_dir_entry	An XIP_dir_entry_struct structure. The structure must be initialized, with the exception of the XIP_status field.

Results if successful:

handle	The value of the handle assigned to the newly added XIP application.

Status:

XIP_STAT_NO_HANDLES
XIP_STAT_NO_PAGES
XIP_STAT_NO_FREE_PAGES
XIP_STAT_NAME_EXISTS
XIP_STAT_BAD_APP_NAME
XIP_STAT_NO_WRITE
XIP_STAT_BAD_PARTITION
XIP_STAT_NO_DIR_SPACE
XIP_STAT_CARD_CHANGED
XIP_STAT_COPY_ERROR
XIP_STAT_NO_WRITE_MEDIA
XIP_STAT_UNKNOWN_ERROR

5.2.14 Copy XIP Page (Write)

```
XIP (*function, partition, logical_page_number, write_count, handle,
              *char_buffer)
```

Purpose:

This function allows an application to copy data from a buffer into one page allocated to an XIP application. The XIP driver should do the copying because the type of programmable memory which the XIP application is being copied into may not respond to simple memory writes issued by an application program.

Updates to existing memory pages may be made if the card technology and system supports write access.

Calling Parameters:

```
XIP (*function, partition, logical_page_number, write_count, handle,
              *char_buffer)
```

function	XIP_COPY_PAGE
partition	Contains the ID of the partition which is to have a page copied into it.
logical_page_number	The logical page number of the given XIP application to write. If logical page zero is specified, the buffer is copied to the offset in the XIP page specified in the XIP_OFFSET field of the directory entry for this application.
write_count	Number of bytes to write to the XIP page.
handle	The value of the handle assigned to the newly added XIP application. This is the handle value returned by the "Add XIP Directory Entry" function.
char_buffer	Contains a far pointer to an area of RAM memory which contains the XIP application code or data to be copied into the logical page specified.

Results:

None

Status Codes:

XIP_STAT_HANDLE_NFOUND	
XIP_STAT_BAD_PAGE	
XIP_STAT_NO_WRITE	
XIP_STAT_BAD_PARTITION	
XIP_STAT_TOO_MANY_BYTES	
XIP_STAT_CARD_CHANGED	
XIP_STAT_COPY_ERROR	The function failed due to an error in copying. The media supports this function but the logical page to which the buffer is being copied to is not blank, and the data cannot be correctly copied.
XIP_STAT_NO_WRITE_MEDIA	
XIP_STAT_UNKNOWN_ERROR	

5.2.15 Delete XIP Directory Entry (Write)

```
XIP (*function, partition, *application_name)
```

Purpose:

This function deletes an XIP application from the XIP directory.

Calling Parameters:

function	XIP_DELETE_ENTRY
partition	Contains the ID of the partition containing the XIP application to be deleted.
application_name	The name of the application to be searched for. The name must be in 8.3 format, i.e., an 8-letter name, followed by the 3 letter extension. If either the name is shorter than 8 characters, or the extension is shorter than 3 characters, they must be padded on the right with spaces to the appropriate size.

Results:

None

Status Codes:

XIP_STAT_APP_NOT_FOUND
XIP_STAT_BAD_APP_NAME
XIP_STAT_NO_WRITE
XIP_STAT_BAD_PARTITION
XIP_STAT_CARD_CHANGED
XIP_STAT_NO_WRITE_MEDIA
XIP_STAT_UNKNOWN_ERROR

5.2.16 Erase XIP Partition (Write)

```
XIP (*function, partition)
```

Purpose:

This function allows an XIP utility to erase the entire contents of an XIP partition in preparation for reuse. The entire XIP partition must be aligned on erase block or device boundaries for this function to erase only the XIP partition, and nothing else.

Calling Parameters:

function	XIP_ERASE_PARTITION
partition	Contains the ID of the partition to be erased.

Results:

None

Status Codes:

XIP_NO_WRITE
XIP_STAT_BAD_PARTITION
XIP_STAT_CARD_CHANGED
XIP_STAT_NO_WRITE_MEDIA
XIP_STAT_UNKNOWN_ERROR

5.2.17 Close XIP Directory Entry (Write)

```
XIP (*function, partition, handle)
```

Purpose:

This function ends the process of creating a new XIP entry. Until this function is executed, a newly-created XIP application cannot be executed. The function "activates" the newly added XIP application by setting the status bits of the corresponding XIP directory entry to a value indicating that it may be loaded and executed.

Calling Parameters:

function	XIP_CLOSE_ENTRY
partition	Contains the ID of the partition containing the XIP directory entry which is to be closed.
handle	The value of the handle assigned to the newly created XIP application.

Results:

None

Status Codes:

XIP_STAT_HANDLE_NFOUND
XIP_STAT_NO_WRITE
XIP_STAT_BAD_PARTITION
XIP_STAT_CARD_CHANGED
XIP_STAT_NO_WRITE_MEDIA
XIP_STAT_UNKNOWN_ERROR

5.2.18 Execute XIP Application (Common)

```
XIP (*function, partition, app_name, *return_code)
```

Purpose:

This function causes the specified XIP application to be mapped in and executed.

Calling Parameters:

function	XIP_EXEC
partition	Contains the ID of the partition containing the XIP directory entry which is to be mapped.
app_name	The ASCIIZ name of the application to be executed.

Results if successful:

return_code	Exit code set by the specified application.

Status Codes:

XIP_STAT_APP_NOT_EXEC
XIP_STAT_APP_NOT_FOUND
XIP_STAT_MAP_HWARE_BUSY
XIP_STAT_BAD_PARTITION
XIP_STAT_CARD_CHANGED
XIP_STAT_UNKNOWN_ERROR

5.2.19 Map Extended Segment (EXIP)

```
XIP (*function, partition, handle, *system_address, *map_count)
```

Purpose:

This function maps memory PC Cards into the system's extended address space. The specified application on the memory PC Card will appear at the address range returned by this function. A system may support simultaneous mapping of multiple segments.

Calling Parameters:

function	XIP_MAP_EXTENDED
partition	Contains the ID of the partition containing the XIP directory entry which is to be mapped.
handle	The value of the handle assigned to the XIP application.

Results if successful:

system_address	System memory address corresponding to the specified application.
map_count	The size, in bytes, of the block mapped system memory.

Status Codes:

XIP_STAT_MAP_HWARE_BUSY

XIP_STAT_NO_EXIP_DRIVER

XIP_STAT_NO_EXIP

XIP_STAT_BAD_PARTITION

XIP_STAT_CARD_CHANGED

XIP_STAT_UNKNOWN_ERROR

5.2.20 Unmap Extended Segment (EXIP)

```
XIP(*function, partition, handle)
```

Purpose:

This function removes the mapping of the specified application. The specified application will no longer appear in the address space.

Calling Parameters:

function	XIP_UNMAP_EXTENDED
partition	Contains the ID of the partition containing the XIP directory entry which is to be mapped.
handle	The value of the handle assigned to the XIP application.

Results:

None

Status Codes:

XIP_STAT_HANDLE_NFOUND

XIP_STAT_NO_EXIP_DRIVER

XIP_STAT_NO_EXIP

XIP_STAT_BAD_PARTITION

XIP_STAT_CARD_CHANGED

XIP_STAT_UNKNOWN_ERROR

5.2.21 Get Partition ID from Address (Common)

```
XIP(*function, *partition, system_address)
```

Purpose:

This function returns the ID of the partition onto which the specified system address is currently mapped.

Calling Parameters:

function	XIP_XLATE_PARTITION
system_address	The system address to be translated to a partition ID.

Results if successful:

partition	partition which is mapped to the specified system address.

Status Codes:

XIP_STAT_ADDR_NOT_MAPPED
XIP_STAT_CARD_CHANGED
XIP_STAT_UNKNOWN_ERROR

5.2.22 Get Slot Number (Common)

```
XIP (*function, partition, *slot)
```

Purpose:

This function returns the number of the slot that contains the specified partition.

Calling Parameters:

function	XIP_GET_SLOT
partition	The ID of the partition to be investigated.

Results if successful:

slot	The slot number that holds the specified partition. Note that the slot number is vendor specific.

Status Codes:

XIP_STAT_BAD_PARTITION
XIP_STAT_CARD_CHANGED
XIP_STAT_UNKNOWN_ERROR

5.2.23 Disable Partition ID (Common)

```
XIP (*function, partition)
```

Purpose:

This function marks a partition ID as invalid and frees it for future reassignment to an XIP partition.

Calling Parameters:

function	XIP_DISABLE_PARTITION
partition	Partition ID of the partition to be disabled.

Results:

None

Status Codes:

XIP_STAT_BAD_PARTITION

XIP_STAT_UNKNOWN_ERROR

6. APPENDICES

6.1 XIP Equate Values

6.1.1 Summary of XIP Function Codes

The following functions are provided by a compliant XIP driver.

Table 6-1: XIP API Functions

Name	Description	Attributes
XIP_GET_VERSION	Get XIP Version	Common
XIP_GET_MAP_SEGS	Get XIP Mappable Segments	LXIP
XIP_GET_PARTITIONS	Get XIP Partition IDs	Common
XIP_GET_HANDLE_RANGE	Get XIP Handle Range	Common
XIP_MAP_HANDLE	Map/Unmap an XIP Handle's Pages	LXIP
XIP_GET_CONTEXT_SIZE	Get XIP Mapping Context Size	Common
XIP_GET_CONTEXT	Get XIP Mapping Context	Common
XIP_SET_CONTEXT	Set XIP Mapping Context	Common
XIP_SEARCH	Search for XIP Directory Entry	Common
XIP_SEARCH_FIRST	Get First XIP Directory Entry	Common
XIP_SEARCH_NEXT	Get Next Directory Entry	Common
XIP_ADD_ENTRY	Add XIP Directory Entry	Write
XIP_COPY_PAGE	Copy XIP Page	Write
XIP_DELETE_ENTRY	Delete XIP Directory Entry	Write
XIP_ERASE_PARTITION	Erase XIP Partition	Write
XIP_CLOSE_ENTRY	Close XIP Directory Entry	Write
XIP_MAP_EXTENDED	Map Extended Segment	EXIP
XIP_UNMAP_EXTENDED	Unmap Extended Segment	EXIP
XIP_XLATE_PARTITION	Get Partition ID from Address	Common
XIP_GET_SLOT	Get Slot Number	Common
XIP_DISABLE_PARTITION	Disable Partition ID	Common
XIP_EXEC	Execute XIP Application	Common
XIP_SEARCH_FULL	Search for specific XIP Directory Entry	Common

6.1.2 Summary of XIP Status Codes

The following status codes are returned by a compliant XIP driver.

Table 6-2: XIP Status Codes

Number	Name	Description
082h	XIP_STAT_APP_NOT_EXEC	XIP Application is marked as non-executable, or is of a type not executable by this driver.
083h	XIP_STAT_HANDLE_NFOUND	XIP handle not found.
085h	XIP_STAT_NO_HANDLES	Insufficient XIP handles to complete the operation.
087h	XIP_STAT_NO_PAGES	Insufficient total logical pages within the XIP partition to complete the operation.
088h	XIP_STAT_NO_FREE_PAGES	Insufficient unallocated logical pages within the XIP directory to complete the operation.
08Ah	XIP_STAT_BAD_PAGE	Logical XIP page out of the range of logical pages allocated to the XIP application.
08Bh	XIP_STAT_BAD_SEGMENT	Mappable segment address is not mappable by the XIP driver.
0A0h	XIP_STAT_APP_NOT_FOUND	XIP application not found in the XIP directory, or there are no more XIP applications present in this XIP directory.
0A1h	XIP_STAT_NAME_EXISTS	XIP application name already exists.
0A2h	XIP_STAT_BAD_APP_NAME	XIP application name is invalid.
0A3h	XIP_STAT_BAD_MAP_ARRAY	XIP mapping array contents are invalid.
0F3h	XIP_STAT_ADDR_NOT_MAPPED	Address not currently mapped.
0F4h	XIP_STAT_MAP_HWARE_BUSY	All mapping hardware currently in use.
0F5h	XIP_STAT_NO_EXIP_DRIVER	Function not available. Hardware extended memory mapping not available.
0F6h	XIP_STAT_NO_EXIP	Function not available. Processor does not support extended memory addressing.
0F7h	XIP_STAT_NO_WRITE	The driver or system does not support Write functionality. The media may or may not support Write functionality.
0F8h	XIP_STAT_BAD_PARTITION	The specified partition does not exist.
0F9h	XIP_STAT_NO_DIR_SPACE	Insufficient space in the XIP directory to complete the operation.
0FAh	XIP_STAT_TOO_MANY_BYTES	The number of bytes requested is too large to write.
0FBh	XIP_STAT_PAGE_MAPPED	Page already mapped into system.
0FCh	XIP_STAT_CARD_CHANGED	The card containing the XIP partition has changed since the last XIP API call.
0FDh	XIP_STAT_COPY_ERROR	An error occurred in copying the XIP application into the XIP partition.
0FEh	XIP_STAT_NO_WRITE_MEDIA	This media does not support write functionality (ROM).
0FFh	XIP_STAT_UNKNOWN_ERROR	The function failed. Cause unknown.

6.2 DOS Operating System Binding

6.2.1 Introduction

This appendix gives details for implementation and utilization of a XIP Operating System Binding (OSB) for DOS. The scope of this implementation is limited to SXIP and LXIP only.

In general, the XIP Operating System Binding for DOS is quite straightforward. One point of confusion may arise with parameter passing; as the XIP OSB for DOS is a register-based API, distinctions between passing by value and passing by reference may become somewhat blurred, and there is naturally some overlaying of parameters within API functional definitions (i.e., a particular register is used as one variable on entry to the API function, and another on exit). Context should resolve the confusion.

6.2.1.1 Related Documents

This section identifies documents related to the XIP DOS Operating System Binding Specification. Information available in the following documents, or in documents listed in the Related documents section of the XIP Specification, is not duplicated within this document.

DR DOS 6.0 System and Programmer's Guide, Second Edition, August 1991, Novell, Inc.

IBM-AT Technical Reference Manual, First Edition, March 1984, International Business Machines.

Schulman, Andrew, et al ***Undocumented DOS***, First Edition, October 1990, Addison-Wesley Publishing Company, Inc.

6.2.1.2 Data Sizes

Data sizes are as defined within the main specification for all data structures. For API parameters, pointers and long ints are 32-bits, ints are sixteen bits, and chars and short ints are 8 bits.

6.2.1.3 Included Code

All code fragments included within this document presuppose the availability of previously defined entry points and routines (i.e., definition of a routine in one fragment allows calling that routine in later fragments with no further explanation).

6.2.2 XIP Loader and Execution

The DOS XIP Device driver includes a loader. SXIP device drivers support only XIP Type 1 applications. LXIP device drivers accept both type 1 and 2 applications.

6.2.2.1 Termination of XIP Execution

It is important to note that XIP applications must terminate with either an Int 21H, Service 4CH (EXIT) request, or an Int 21H, Service 31H (KEEP) request. None of Int 20H, Int 21H, Service 0, or Int 27H are acceptable, as all require that CS be equal to the segment of the active PSP, which, when an XIP terminates, is not the case.

6.2.2.2 SXIP Execution

SXIP (Type 1) applications are similar in form to .COM programs. The executable image is less than 64K in size, and is not overlaid. These programs are loaded in the following way:

1. The entire image of the program is mapped into memory.
2. A block of memory large enough to hold the environment and program name is allocated, and the current environment is copied to that block.
3. The name of the XIP image (including an ascii number representing the partition number) is appended to the new environment.
4. The rest of memory is allocated, and a PSP created at the beginning of the newly allocated memory.
5. The newly created environment is marked as belonging to the newly created PSP.
6. DS, ES, and SS are set to the segment of PSP. SP and all other registers are set to 0.
7. The entry point specified in the first page of the application is jumped to.

As an example, if the XIP application WORDPROC.XIP were to be executed from partition 10, the ASCIIZ string appended to the end of the environment would be "10:WORDPROC.XIP."

Note that, in contrast to the Version 1.0 specification, the above method of loading is not a suggested method; it is now specified, and any other loading methods must be compatible.

6.2.2.3 SXIP Image Format

SXIP applications are direct execution images. No header of any type is used. No code or data is relocated into RAM at startup.

6.2.2.4 LXIP Execution

LXIP (Type 2) applications are, in general, much more complex than SXIP applications. However, the majority of the loading sequence is identical. The primary difference in the load sequence is that only the first 16K page of the application is mapped into memory. Steps 2 through 7, above, are identical.

Note again that, in contrast to the Version 1.0 specification, the above method of loading is not a suggested method; it is now specified, and any other loading methods must be compatible.

6.2.2.5 LXIP Image format

LXIP (Type 2) applications are responsible for managing their own overlays, as well as data initialization. Thus, the Image format of an LXIP application is strictly the responsibility of the developer.

6.2.3 XIP API Details

The details of each API entry calling format are listed below. One should refer to the XIP specification for additional information. In all cases, the return values are based upon the assumption that the operation succeeded, and thus, that the carry flag is clear. If such is not the case, AH will return with the appropriate error code.

6.2.3.1 Get XIP Version (Common)

```
XIP (*function, *version, *functionality)
```

function	AH	XIP_GET_VERSION	80h
version	AL		
functionality	AH		

6.2.3.2 Get XIP Mappable Segments (LXIP)

```
XIP (*function, *XIP_mappable_array, *mappable_array_length)
```

function	AH	XIP_GET_MAP_SEGS	81h
XIP_mappable_array	ES:DI		
mappable_array_length	CX		

6.2.3.3 Get XIP Partition IDs (Common)

```
XIP (*function, partition_ids_array, *partition_array_length)
```

function	AH	XIP_GET_PARTITIONS	82h
partition_ids_array	ES:DI		
partition_array_length	CX		

6.2.3.4 Get XIP Handle Range (Common)

```
XIP (*function, *XIP_first_handle, *XIP_last_handle, *total_handles)
```

function	AH	XIP_GET_HANDLE_RANGE	83h
XIP_first_handle	BX		
XIP_last_handle	DX		
total_handles	CX		

6.2.3.5 Map/Unmap an XIP Handle's Pages (LXIP)

```
XIP (*function, handle, partition, *map_count, seg_map_array)
```

function	AH	XIP_MAP_HANDLE	84h
handle	DX		
partition	AL		
map_count	CX		
seg_map_array	DS:SI		

6.2.3.6 Get XIP Mapping Context Size (Common)

```
XIP (*function, *map_context_size)
```

function	AH	XIP_GET_CONTEXT_SIZE	85h
map_context_size	AL		

6.2.3.7 Get XIP Mapping Context (Common)

```
XIP (*function, *xip_context)
```

function	AH	XIP_GET_CONTEXT	86h
xip_context	ES:DI		

6.2.3.8 Set XIP Mapping Context (Common)

```
XIP (*function, *xip_context)
```

function	AH	XIP_SET_CONTEXT	87h
xip_context	DS:SI		

6.2.3.9 Search for XIP Directory Entry (Common)

```
XIP (*function, partition, *application_name, *handle, *page_count)
```

function	AH	XIP_SEARCH	88h
partition	AL		
application_name	DS:SI		
handle	DX		
page_count	CX		

6.2.3.10 Get First XIP Directory Entry (Common)

```
XIP (*function, partition, *XIP_dir_entry, *handle, *page_count)
```

function	AH	XIP_SEARCH_FIRST	89h
partition	AL		
XIP_dir_entry	ES:DI		
handle	DX		
page_count	CX		

6.2.3.11 Get Next XIP Directory Entry (Common)

```
XIP (*function, partition, *XIP_dir_entry, *handle, *page_count)
```

function	AH	XIP_SEARCH_NEXT	8Ah
partition	AL		
XIP_dir_entry	ES:DI		
handle	DX		
page_count	CX		

6.2.3.12 Add XIP Directory Entry (Write)

```
XIP(*function, partition, *XIP_dir_entry, *handle)
```

function	AH	XIP_ADD_ENTRY	8Bh
partition	AL		
XIP_dir_entry	DS:SI		
handle	DX		

6.2.3.13 Copy XIP Page (Write)

```
XIP (*function, partition, logical_page_number, write_count, handle,
             *char_buffer)
```

function	AH	XIP_COPY_PAGE	8Ch
partition	AL		
logical_page_number	BX		
write_count	CX		
handle	DX		

6.2.3.14 char_buffer DS:SI Delete XIP Directory Entry (Write)

```
XIP (*function, partition, *application_name)
```

function	AH	XIP_DELETE_ENTRY	8Dh
partition	AL		
application_name	DS:SI		

6.2.3.15 Erase XIP Partition (Write)

```
XIP (*function, partition)
```

function	AH	XIP_ERASE_PARTITION	8Eh
partition	AL		

6.2.3.16 Close XIP Directory Entry (Write)

```
XIP (*function, partition, handle)
```

function	AH	XIP_CLOSE_ENTRY	8Fh
partition	AL		
handle	DX		

6.2.3.17 Map Extended Segment (EXIP)

```
XIP (*function, partition, handle, *system_address, *map_count)
```

function	AH	XIP_MAP_EXTENDED	90h
partition	AL		
handle	DX		
system_address	EDX		
map_count	ECX		

6.2.3.18 Unmap Extended Segment (EXIP)

```
XIP(*function, partition, handle)
```

function	AH	XIP_UNMAP_EXTENDED	91h
partition	AL		
handle	DX		

6.2.3.19 Get Partition ID from Address (Common)

```
XIP(*function, *partition, system_address)
```

function	AH	XIP_XLATE_PARTITION	92h
system_address	ES:DI		
partition	AL		

6.2.3.20 Get Slot Number (Common)

```
XIP (*function, partition, *slot)
```

function	AH	XIP_GET_SLOT	93h
partition	AL		
slot	AL		

6.2.3.21 Disable Partition ID (Common)

```
XIP (*function, partition)
```

function	AH	XIP_DISABLE_PARTITION	94h
partition	AL		

6.2.3.22 Execute XIP Application (Common)

```
XIP (*function, partition, app_name, *return_code)
```

function	AH	XIP_EXEC	95h
partition	AL		
app_name	ES:DI		

6.2.3.23 Search for full XIP Directory Entry (Common)

```
XIP (*function, partition, *XIP_dir_entry, *handle, *page_count)
```

function	AH	XIP_SEARCH_FULL	96h
partition	AL		
XIP_dir_entry	ES:DI		

6.2.3.24 Secondary Map/Unmap an XIP Handle's Pages (LXIP)

```
XIP (*function, handle, partition, *map_count, seg_map_array)
```

function	AH	SECOND_XIP_MAP_HANDLE	97h
handle	DX		
partition	AL		
map_count	CX		
seg_map_array	ES:DI		

6.2.3.25 Secondary Set XIP Mapping Context (Common)

```
XIP (*function, *xip_context)
```

function	AH	SECOND_XIP_SET_CONTEXT	98h
xip_context	ES:DI		

6.2.3.26 Secondary Search for XIP Directory Entry (Common)

```
XIP (*function, partition, *application_name, *handle, *page_count)
```

function	AH	SECOND_XIP_SEARCH	99h
partition	AL		
application_name	ES:DI		
handle	DX		
page_count	CX		

6.2.3.27 Secondary Add XIP Directory Entry (Write)

```
XIP(*function, partition, *XIP_dir_entry, *handle)
```

function	AH	SECOND_XIP_ADD_ENTRY	9Ah
partition	AL		
XIP_dir_entry	ES:DI		
handle	DX		

6.2.3.28 Secondary Copy XIP Page (Write)

```
XIP (*function, partition, logical_page_number, write_count, handle,
             *char_buffer)
```

function	AH	SECOND_XIP_COPY_PAGE	9Bh
partition	AL		
logical_page_number	BX		
write_count	CX		
handle	DX		
char_buffer	ES:DI		

6.2.3.29 Delete XIP Directory Entry (Write)

```
XIP (*function, partition, *application_name)
```

function	AH	SECOND_XIP_DELETE_ENTRY	9Ch
partition	AL		
application_name	ES:DI		

6.2.4 XIP Applications Programming Interface

6.2.4.1 Initializing the XIP API

1. Issue a DOS "open read-only mode" request (INT 21h, Service 3D00h). This function requires a far pointer to the ASCIIZ string containing the device name to open. In this case, the device name is actually the internal name found in the XIP device driver's header. The pointed-to ASCIIZ string should have the following format:

   ```
   XIP_device_name   DB   "XIP$$$$$", 0
   ```

If DOS does not return an error status, one can assume that either a device with the name "XIP$$$$$" is installed, or a file with this name is on the current disk drive. Proceed to step 4.1.4, otherwise, proceed to the next step.

1. If DOS returned a "too many open files" status, one can modify one's application so that it opens the XIP device before it opens any other files. The XIP handle is not used after the entry point is obtained. If this was not the error one's application received, proceed to the next step.
2. If DOS returned a "file / path not found", the XIP-device driver is not installed. If one's application requires the XIP-device driver, there is only one way to correct the problem. The user must install the XIP-device driver, modify the CONFIG.SYS file to reflect the installation, and reboot the system before proceeding. One's application cannot proceed further.
3. Issue a DOS IOCTL "get device data" using the handle obtained in step 1. This function returns device data that allows one to determine whether XIP is a device or a file.
4. If DOS returns any error status, one may assume that the XIP device driver is not installed. The user will have to follow the procedure outlined in step 3 to correct the problem.
5. Check that "XIP$$$$$" is a device and not a file with the same name. The device data returned by the previous DOS function contains the ISDEV bit (DX bit 7). If the ISDEV bit is a 1 then "XIP$$$$$" is a character device and not a file. If ISDEV is bit is a 0 then "XIP$$$$$" is a file and there is no XIP-device driver installed. The user will have to follow the procedure outlined in step 3 to correct the problem. Also, the file named XIP shall be renamed so that the user may access it after the XIP driver is installed. This should be an extremely rare situation.
6. Issue a DOS "IOCTL read" using the handle obtained in step 1 for a maximum of 4 bytes.
7. If DOS returns any error status, or the driver does not transfer the specified number of bytes, one may assume that the XIP-device driver is not a compliant driver. The user will have to follow the procedure outlined in step 3 to correct this problem.
8. Save the device driver entry-point address returned by the “read.”
9. Issue a DOS "close" command using the handle obtained in step 1. Doing so frees up the handle allocated by the original "open.” The handle is not used again.

The following procedure is an example of the technique outlined in this section.

```
;-------------------------------------------------------------------;
;    open_XIP_driver                                               ;
; The procedure verifies that the XIP driver is installed in the ;
; system and returns a handle so that driver IOCTLs may be done  ;
; if it is present.                                              ;
; If XIP driver is installed                                     ;
;    CARRY CLEAR                                                 ;
;    (bx) = handle for XIP device driver get/set calls           ;
; else                                                           ;
;    CARRY SET                                                   ;
;-------------------------------------------------------------------;

open_XIP_driver  proc
;-------------------------------------------------------------------;
; Open the XIP device.                                           ;
;-------------------------------------------------------------------;
     mov   dx, offset XIP_device_name
                       ; (ds:dx) = far ptr to
                       ; device name string
     mov   ax, 3D00h   ; (ax) = open read-only function
     int   21h         ; issue device read-only open
     jc    oXd_02            ; error during device open

;-------------------------------------------------------------------;
; Get the info flags for the XIP handle.                         ;
;-------------------------------------------------------------------;
     mov   bx, ax            ; (bx) = handle returned by open
     mov   ax, 4400h   ; (ax) = IOCTL get device data function
     int   21h         ; issue get device data IOCTL
     jc    oXd_01            ; error during IOCTL get device info

;-------------------------------------------------------------------;
; Test the ISDEV bit in the device info flags.                   ;
;-------------------------------------------------------------------;
     test  dx, 0080h   ; (dx) = file or device data
     jz    oXd_01            ; XIP is a file, NOT a device

;-------------------------------------------------------------------;
; XIP driver is installed in this system.                        ;
; Return:                                                        ;
;   (bx) = XIP driver handle.                                    ;
;   (CARRY CLEAR) to indicate that the XIP device is             ;
;   installed.                                                   ;
;-------------------------------------------------------------------;
     clc
     ret

;-------------------------------------------------------------------;
; XIP driver is not installed in this system.                    ;
; Close the file named XIP$$$$$.                                 ;
; (bx) = handle returned by open call                            ;
;-------------------------------------------------------------------;
oxd_01:    mov   ah, 3Eh           ; (ah) = close function
     int   21h         ; close XIP$$$$$ file/driver

;-------------------------------------------------------------------;
; XIP driver is not installed in this system.                    ;
; Return:                                                        ;
;   (CARRY SET) to indicate that the XIP device is not           ;
;   installed.                                                   ;
;-------------------------------------------------------------------;
oXd_02:    stc
```

```
        ret

open_XIP_driver  endp

;-------------------------------------------------------------------;
; XIP driver name.                                                  ;
;-------------------------------------------------------------------;
XIP_device_name  db    "XIP$$$$$", 0
```

6.2.5 IOCTL Read (Get Current XIP API Entry Point)

The DOS "IOCTL read" function (INT 21h, function 4402h) is used to obtain the XIP API entry point. This function will read, into a buffer supplied by the application, a dword pointer supplied by the XIP driver. The dword pointer in the buffer is a far pointer to a far pointer to the XIP API. All applications needing to use the XIP API must obtain this entry point before they can make an XIP API call.

The following example builds on the previous example and demonstrates how an application obtains the XIP API entry point.

```
;-----------------------------------------------------------------;
; Get the current XIP API entry point.                           ;
; If XIP API services are available                              ;
;   CARRY CLEAR                                                  ;
;     (bx)  = handle for future XIP device driver get/set calls  ;
;   (XIP_callback) = far pointer to far pointer to the XIP API   ;
; else                                                           ;
;   CARRY SET                                                    ;
;-----------------------------------------------------------------;
get_XIP_callback proc
     call  open_XIP_driver        ; check for the XIP driver & open it
     jc    gXc_02                 ; XIP driver not installed

;-----------------------------------------------------------------;
; Get the XIP API entry point.                                   ;
; (bx) = XIP driver handle returned by open                      ;
;-----------------------------------------------------------------;
     mov   dx, offset XIP_callback; (ds:dx) = far ptr to XIP callback
buffer
     mov   cx, 4            ; (cx) = # bytes to transfer (dword size)
     mov   ax, 4402h        ; (ax) = IOCTL read device data
     int   21h              ; issue IOCTL read device data
     jc    gXc_01                 ; error during IOCTL read device data

;-----------------------------------------------------------------;
; Verify that only the XIP API entry point was transferred.      ;
;-----------------------------------------------------------------;
     cmp   ax, 4            ; (ax) = # bytes actually transferred
     jne   gXc_01                 ; driver did not transfer the
specified
                            ; # of bytes
;-----------------------------------------------------------------;
; XIP API services are available.                                ;
; Return:                                                        ;
;   (XIP_callback) = far pointer to far pointer to the XIP API.  ;
;   (CARRY CLEAR) to indicate that the XIP API services          ;
;   are available.                                               ;
;-----------------------------------------------------------------;
     clc
     ret

;-----------------------------------------------------------------;
; Close the XIP device.                                          ;
; (bx) = handle returned by open_XIP_driver call                 ;
;-----------------------------------------------------------------;
gXc_01:    mov   ah, 3Eh                ; (ah) = close function
     int   21h              ; close XIP device

;-----------------------------------------------------------------;
; XIP API services are not available.                            ;
; Return:                                                        ;
;   (CARRY SET) to indicate that the XIP API services are        ;
;   are not available.                                              ;
;                                                                   ;
;-------------------------------------------------------------------;
gXc_02:     stc
      ret
get_XIP_callback endp

;-------------------------------------------------------------------;
; XIP callback storage.                                             ;
;-------------------------------------------------------------------;
XIP_callback     dd    ?
```

6.2.6 IOCTL Write (Set New XIP API Entry Point)

The DOS "IOCTL write" function (INT 21h, function 4403h) is used to set a new XIP API entry point. This function will write a dword pointer, supplied by the application, to the XIP driver. This dword pointer is a far pointer to the new XIP entry procedure. The function provides an XIP utility, or another device driver that needs to trap XIP API accesses, with the ability to chain into the XIP API's path of execution.

If one is creating an XIP utility that absolutely must chain into the XIP API, remember to restore the original XIP entry point before one's utility exits back to DOS. If one does not, and one's code exits, the next application that attempts to use the XIP API will probably hang the users system.

The following example builds on the previous examples and demonstrates how an application sets a new XIP API entry point.

```
;------------------------------------------------------------------;
; Get the current XIP API entry point and set a new XIP          ;
; API entry point.                                               ;
; If XIP API services are available                              ;
;   CARRY CLEAR                                                  ;
;   (bx)             = handle for XIP device driver get/set calls;
;   (XIP_callback)   = far pointer to far pointer to the XIP API ;
;   (old_XIP_ent_pt) = address of the current XIP API entry point.
     ;
;   (new_XIP_ent_pt) = address of the new XIP API entry point.   ;
; else                                                           ;
;   CARRY SET                                                    ;
;------------------------------------------------------------------;

set_XIP_callback proc
;------------------------------------------------------------------;
; Open the XIP driver and get the XIP callback.                  ;
;------------------------------------------------------------------;
      call  get_XIP_callback ; get XIP callback
      jc    sXc_01                 ; could not get the XIP callback

;------------------------------------------------------------------;
; Save the address of the current XIP API entry point so that    ;
; it can be restored later.  The example assumes that the old    ;
; XIP entry point is accessible via the example code segment.    ;
;------------------------------------------------------------------;
      les   di, XIP_callback             ; (es:di) = far ptr to far ptr
                                   ; to XIP API entry point
      les   di, dword ptr es:[di]        ; (es:di) = far ptr XIP entry
                                   ; point address
      mov   word ptr cs:old_XIP_ent_pt[0], di
      mov   word ptr cs:old_XIP_ent_pt[2], es
      ; (old_XIP_ent_pt) = current XIP entry point address

;------------------------------------------------------------------;
; Initialize a far pointer in a buffer so that it points to      ;
; the new XIP API entry point.                                   ;
;------------------------------------------------------------------;
      mov   word ptr new_XIP_ent_pt[0], offset XIP_trap
      mov   word ptr new_XIP_ent_pt[2], cs
      ; (new_XIP_ent_pt) = new XIP entry point address

;------------------------------------------------------------------;
; Send the new XIP API entry point to the XIP driver.            ;
; (bx) = handle returned by get_XIP_callback                     ;
;-----------------------------------------------------------------;
     mov   dx, offset new_XIP_ent_pt
     ; (ds:dx) = far ptr to new XIP entry point buffer
     mov   cx, 4             ; (cx) = # bytes to transfer (dword size)
     mov   ax, 4403h         ; (ax) = IOCTL write device data
     int   21h               ; issue IOCTL write device data
     jc    gXc_01                 ; error during IOCTL read device data

;-----------------------------------------------------------------;
; New XIP entry point has been set.                               ;
; Return:                                                         ;
;   (bx) = handle for future XIP device driver get/set calls     ;
;   (CARRY CLEAR) to indicate that the XIP API services are      ;
;   available.                                                    ;
;-----------------------------------------------------------------;
     clc
     ret

;-----------------------------------------------------------------;
; XIP API services are not available.                             ;
; Return:                                                         ;
;   (CARRY SET) to indicate that the XIP API services are        ;
;   not available.                                                ;
;-----------------------------------------------------------------;
sXc_01:    stc
     ret
set_XIP_callback endp

;-----------------------------------------------------------------;
; New XIP entry point storage.                                    ;
;-----------------------------------------------------------------;
new_XIP_ent_pt   dd    ?

;-----------------------------------------------------------------;
; Old XIP entry point storage.  This example assumes that this   ;
; pointer resides in the same CODE segment as do the             ;
; set_XIP_callback and XIP_trap procedures.                      ;
;-----------------------------------------------------------------;
old_XIP_ent_pt   dd    ?
```

6.2.7 Chaining into the XIP API

The following example builds on the previous examples and demonstrates how an XIP utility would properly chain into the XIP API. The hypothetical example provided assumes that the original XIP driver cannot either write or erase the special devices the XIP_trap code supports. However, the original driver is capable of doing all other functions. Therefore, the example inspects the function codes passed to the XIP API and traps only erase and write functions rather than permitting the original XIP driver to do them. All other function will be passed on through.

The example also assumes that set_XIP_callback has been called and has completed successfully.

```
;-------------------------------------------------------------------;
;This example code simply checks to see if the XIP API code passed;
;to the XIP driver performs writes or erases.  If it does, it    ;
;services the erase or write.  If it doesn't, it chains through  ;
;to the original XIP API entry point.                            ;
;-------------------------------------------------------------------;

XIP_trap   proc  far
;-------------------------------------------------------------------;
;Trap the three XIP functions that perform writes or erases.     ;
;-------------------------------------------------------------------;
      cmp   ah, XIP_Add      ; (ah) = XIP function code
      je    Xt_Add_XIP_Dir         ; trap Add XIP Directory Entry

      cmp   ah, XIP_Copy
      je    Xt_Copy_XIP_Dir        ; trap Copy XIP Directory Entry

      cmp   ah, XIP_Erase
      je    Xt_Erase_Partn         ; trap Erase XIP Partition

;-------------------------------------------------------------------;
;Chain into the original XIP API entry point.                    ;
;-------------------------------------------------------------------;
      jmp   dword ptr cs:old_XIP_ent_pt

;-------------------------------------------------------------------;
;Your special trap code continues here.                          ;
;-------------------------------------------------------------------;
Xt_Add_XIP_Dir:       .
                 .
Xt_Copy_XIP_Dir: .
                 .
Xt_Erase_Partn:       .
                 .

;-------------------------------------------------------------------;
;Your trap code has finished its work.                           ;
;-------------------------------------------------------------------;
XIP_trap_OK:
      clc
      ret                    ; CARRY CLEAR indicates operation passed

XIP_trap_err:
      ; (ah) = error status of operation
      stc                    ; CARRY SET indicates operation failed
      ret
XIP_trap   endp
```

6.2.8 Example of XIP API Use

An example is included to illustrate some of the more complex processes involved in using an XIP driver and the companion XIP directory structure managed by the driver.

Suppose one wishes to install a new XIP application named "WORDPROC.XIP" to an XIP partition within a user's PC Memory Card. Further assume that the new XIP application is 109 Kbytes long. Assume further that the XIP partition on this hypothetical PC Memory Card already has some XIP applications in it, but absolute-page 17 through absolute-page n within this partition are free. The following steps illustrate how one might copy this new XIP application into the partition. This example assumes an XIP installation tool that adds applications on page boundaries.

1. An XIP-copy-utility would search the existing XIP directory structure, using the "Search for XIP Directory Entry" function, for an existing XIP application with the same name.
2. If an XIP application with the same name already existed in the XIP directory, the XIP-copy-utility might inform the user of this condition and, if instructed to do so by the user, delete the old XIP application by using the "Delete XIP Directory Entry" function.
3. The XIP-copy-utility, knowing the size of the new XIP application, and that no XIP application is in the current XIP directory with the same name as the new one being added, would create for the new application using the "Add XIP Directory Entry" function.

   The data structure for adding this XIP application would look like:

```
XIP_dir_struct STRUC
name  DB "WORDPROC"
ext   DB "XIP"       ; if necessary
status   DB    8h          ; this is an EXIP application
begin DW xxxxh       ; entry point offset
offset   DW    0           ; beginning of page
reserved DW    0,0,0       ; reserved words
creation_time  DW    xxxxh       ; DOS formatting of time bits
creation_date  DW    xxxxh       ; DOS formatting of date bits
first_page     DW    xxxxh       ; based on previous entry
size  DD (109*1024)  ; size in Kbytes
XIP_dir_struct ENDS
```

4. After this structure is written into the XIP partition, pages 17, 18, 19...23 are now used by the XIP application named "WORDPROC.XIP."
5. The XIP-copy-utility would then copy the first 16-Kbyte portion of the new XIP application, from whatever media it was contained on, into a RAM buffer . The XIP-copy-utility then copies this buffer into the first logical page allocated to the new XIP application by using the "Copy XIP Page" function. Realize that the type of memory that XIP applications are stored will typically not be normal RAM. It is necessary to have the XIP driver do the copying because it is aware of the nature of the memory on the card. The call may even fail if the memory type happens to be ROM, which is not writable. The call may also fail if a defect is discovered in the memory in the XIP partition. The important point to remember is that the status of every operation must always be checked.
6. The process begun in step 4 is repeated for logical pages 1, 2, 3, 4, 5, and 6.
7. The last step required is to use the "Close XIP Directory Entry" function. This essentially makes the XIP directory entry and its corresponding application "active" so that it can be loaded and executed.

8. Once all logical pages of the new XIP application have been initialized and the entry has been closed, the process of adding a new XIP application to the partition is complete. At this point, the XIP-copy-utility is done and the user would have a new XIP application in their XIP partition.