Preboot Execution Environment (PXE) Specification

Version 2.1

September 20, 1999

Intel Corporation

with special contributions from

SYSTEMSOFT™

This document is for informational purposes only. **INTEL MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS DOCUMENT.**

Intel Corporation may have patents or pending patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. The furnishing of this document does not give you any license to the patents, trademarks, copyrights, or other intellectual property rights except as expressly provided in any written license agreement from Intel Corporation.

Intel does not make any representation or warranty regarding specifications in this document or any product or item developed based on these specifications. INTEL DISCLAIMS ALL EXPRESS AND IMPLIED WARRANTIES, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OR MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND FREEDOM FROM INFRINGEMENT. Without limiting the generality of the foregoing, Intel does not make any warranty of any kind that any item developed based on these specifications, or any portion of a specification, will not infringe any copyright, patent, trade secret or other intellectual property right of any person or entity in any country. It is your responsibility to seek licenses for such intellectual property rights where appropriate. Intel shall not be liable for any damages arising out of or in connection with the use of these specifications, including liability for lost profit, business interruption, or any other damages whatsoever. Some states do not allow the exclusion or limitation of liability or consequential or incidental damages; the above limitation may not apply to you.

† Other product and corporate names may be trademarks of other companies and are used only for explanation and to the owners' benefit, without intent to infringe.

Table of Contents

List of Tables

List of Figures

1. Introduction

A common problem faced by IT managers is to ensure that client systems in their enterprises can boot appropriate software images using appropriate configuration parameters. These selected boot images and configuration parameters must be acquired from selected servers in the enterprise as dictated by the needs of the particular environment, the capabilities or mission of the user, the resources available within the client, etc. Furthermore, these clients should boot consistently and in an interoperable manner regardless of the sources or vendors of the software and the hardware of both client and server machines.

This goal can be accomplished only through a uniform and consistent set of pre-boot protocol services within the client that ensure that network-based booting is accomplished through industry standard protocols used to communicate with the server. In addition, to ensure interoperability, the downloaded Network Bootstrap Program (NBP) must be presented with a uniform and consistent pre-boot operating environment within the booting client, so it can accomplish its task independent of, for example, the type of network adapter implemented in the system. This capability is useful in enhancing the manageability of the client machine in several situations; for example:

- **Remote new system setup**. If the client does not have an OS installed on its hard disk, or the client has no hard disk at all, downloading an NBP from a server can help automate the OS installation and other configuration steps.
- **Remote emergency boot**. If the client machine fails to boot due to a hardware or software failure, downloading an executable image from a server can provide the client with a specific executable that enables remote problem notification and diagnosis.
- **Remote network boot**. In instances where the client machine has no local storage, it can download its system software image from the server in the course of normal operation.

This document specifies the **Preboot Execution Environment (PXE)**. PXE embodies three technologies that will establish a common and consistent set of pre-boot services within the boot firmware of Intel Architecture systems:

- A uniform protocol for the client to request the allocation of a network address and subsequently request the download of an NBP from a network boot server.
- A set of APIs available in the machine's pre-boot firmware environment that constitutes a consistent set of services that can be employed by the NBP or the BIOS.
- A standard method of initiating the pre-boot firmware to execute the PXE protocol on the client machine.

In summary, using the capabilities described above, a newly installed networked client machine should be able to enter a heterogeneous network, acquire for itself a network address from a DHCP server, and then download an NBP to set itself up. This sets the stage to enable IT managers to customize the manner in which their network client machines go through a network-based booting process.

1.1 Structure of this Document

This document is organized in a top down manner from the point of view of the PXE client "boot behavior". This section contains an overview. The next two sections specify the external behavior of PXE in platform architecture independent terms, so both sections specify the functionality PXE provides, and are of interest to all system providers regardless of platform or BIOS type. The last two sections specify implementation details and required platform support for PXE in the Intel Architecture PC platform.

Section 2 begins with a description of the network protocol used by the booting PXE client and the Redirection and Boot servers that provide the client with boot information and files. This section covers the network visible behavior of PXE and is defined in terms of network protocol.

Section 3 describes the PXE APIs available to the boot program(s) downloaded from the Boot Server. This section specifies the standard interface provided by PXE to the downloaded boot program. With the exception of finding the API entry point, this section is platform architecture independent.

Section 4 specifies the procedure a standard Intel Architecture PC BIOS uses to find and load the boot ROM code (the PXE Initial Program Load) and the PXE loader behavior in this environment.

Section 5 PXE BIOS Support, specifies the BIOS support required to support PXE in a standard Intel® Architecture PC.

Three new capabilities, described for the first time in this document, have been added to the PXE specification:

- Boot Server Discovery.
- Protected Remote Boot.
- The ability to split PXE Base Code and UNDI code into separate ROMs.

1.2 Related Documents

After referring to a related specification the first time, this document uses the [TAG] reference from this section to refer to related specifications.

1.2.1 Wired for Management

Wired for Management (WfM) Baseline Version 2.0, December 23, 1998 *[WFM]*
http://developer.intel.com/ial/wfm/wfmspecs.htm

PXE PDK
http://www.intel.com/ial/wfm/tools/pxe/index.htm

PXE Powerpoint Presentation
http://www.intel.com/ial/wfm/class/index.htm

Boot Integrity Services API Specification ***[BIS]***
Version 1.0

http://www.intel.com/ial/wfm/wfmspecs.htm

1.2.2 BIOS Specifications

System Management BIOS Reference Specification ***[SM BIOS]***
Version 2.2, March 16, 1998
ftp://download.intel.com/ial/wfm/smbios.pdf
http://www.phoenix.com/techs/specs.html

BIOS Boot Specification ***[BBS]***
Version 1.01, January 11, 1996
http://www.phoenix.com/techs/specs.html

POST Memory Manager Specification ***[PMM]***
Version 1.01, January 8, 1998
http://www.phoenix.com/techs/specs.html

Plug and Play BIOS Specification ***[PnP BIOS]***
Version 1.0A, May 5, 1994
http://www.phoenix.com/techs/specs.html

1.2.3 UUID Documents

CAE Specification ***[UUID]***
DCE 1.1: Remote Procedure Call
Document Number: C706
Universal Unique Identifier Appendix
Copyright (c) 1997 The Open Group
http://www.opengroup.org/onlinepubs/9629399/toc.htm

1.2.4 Other PC System Documents

PC 9x System Design Guide, v1.0 ***[PC98]***
http://developer.intel.com/design/pc98

Network PC Design Guidelines, v1.0b ***[NETPC]***
http://developer.intel.com/design/netpc/netovr.htm
ftp://download.intel.com/ial/wfm/netpc.pdf

1.3 Data Types and Terms Used in This Guide

The following conventions and terms are used in this specification:

!PXE	Acronym for the !PXE structure. This structure is used by protocol drivers that need to locate and use PXE services.
BAID	Acronym for a BIOS Aware IPL Device. The BIOS contains all code required to IPL from the device.
Base Memory	The first 640K bytes of memory in the system.
BCV	Acronym for Boot Connection Vector. A field in the PnP header for a device with an associated option ROM

BEV	Acronym for Boot Entry Vector. A field in the Plug and Play (PnP) Header of a device with an associated option ROM. PXE is implemented as a BEV option ROM.
BIOS	Acronym for Basic Input/Output System, also known as ROM BIOS when resident in Read Only Memory or ROM.
BOOTP	This is an earlier IETF-defined booting protocol that is much less flexible than DHCP. However, DHCP has been defined to be upwardly compatible with BOOTP and both these protocols can co-exist and function simultaneously in the same network. RFC 1534 defines how DHCP and BOOTP must be implemented to ensure they can co-exist in the same network and inter-operate
Bootstrap	Also known as the Initial Program Load (IPL). The initial code loaded by the BIOS to initiate a client operating environment.
BUSD	Bus/Device. A BUSD option ROM may contain code to locate and initialize devices on a bus that is not supported in the BIOS Core. The BUSD API calls are used by the UNDI IPL routine and NBPs to enable and disable bus components and devices.
Client	In this document, the Client is usually the machine receiving the NBP. The client machine hosts the PXE boot ROM.
DDIM	Device Driver Initialization Model. Under this model, all Option ROMs installed in a Plug and Play system which indicate that they support DDIM will be copied into RAM by the System BIOS. Documented in the [PnP] specification
DHCP	Dynamic Host Configuration Protocol. An industry standard Internet protocol defined by the IETF. DHCP was defined to dynamically provide communications-related configuration values such as network addresses to network client computers at boot time. DHCP is specified by IETF RFCs 1534, 2131, and 2132
Extended Memory	Typically used to describe memory on an Intel architecture system above 1 MB.
GUID	Globally unique identifier; a synonym for UUID.
IETF	Internet Engineering Task Force. The open industry body that owns the technical specifications for Internet standards (protocols, APIs, etc.)
IPL	Acronym for Initial Program Load, also known as the bootstrap or boot process
MTFTP	Multicast Trivial File Transfer Protocol. PXE implements a proprietary implementation of MTFTP.
NBP	Acronym for Network Bootstrap Program. The remote boot image downloaded by the PXE client via TFTP or MTFTP.
Option ROM	ROM associated with a plug and play device. May be located on the device or in non-volatile storage on a system.

POST	Acronym for Power On Self-Test. POST processing in the BIOS is responsible for initializing the system hardware and starting IPL.
PXE	Acronym for Pre-boot Execution Environment
PXENV+	Acronym for the PXENV+ structure. This structure was used by protocol drivers that need to locate and use PXE services. New protocol drivers must be written to use !PXE.
RFC	Request for Comment. This is a class of document used by the IETF for proposing technologies for adoption by the IETF and setting these technologies on a standards track. Each RFC is assigned a unique integer document number. When a technology is adopted by the IETF as a standard, the corresponding RFC becomes the document that formally specifies the technology.
ROM	Acronym for Read-Only Memory
Shadow	A technique for mapping RAM into UMB space, potentially on top of ROM already occupying this space. Shadow memory may be write protected after initialization.
System	The host computer
TFTP	Trivial File Transfer Protocol. An industry standard Internet protocol defined by the IETF to enable the transmission of files across the Internet. Trivial File Transfer Protocol (TFTP, Revision 2) to support NBP download is specified by IETF RFC 1350.
UDP	User Datagram Protocol.
UNDI	Universal Network Device Interface.
Upper Memory	An area of system memory between the video buffers and the system ROM BIOS. Typically between real mode segments C000 and F000.
UUID	Universally Unique ID. This is a 128-bit identifier generated via a specific algorithm that is extremely unlikely to be generated by the algorithm in another place or at another time. Thus UUIDs can safely be used to uniquely name entities in computer systems (e.g. software images, APIs, machines, sessions, etc.). It is specified in [UUID].

Table 1-1 Data Type Definitions

Data Type	*Description*
ADDR32	**Physical 32-bit address.** Typedef UINT32 ADDR32;
IP4	**Network address.** #define IP_ADDR_LEN 4 Typedef union u_IP4 { UINT32 num; UINT8 array[IP_ADDR_LEN]; } IP4;
MAC_ADDR	**Hardware address**. #define MAC_ADDR_LEN 16 Typedef UINT8 MAC_ADDR[MAC_ADDR_LEN];
OFF16	**Unsigned 16-bit offset.** Typedef UINT16 OFF16;
PXENV_EXIT	**Unsigned 16-bit PXE exit code.** Typedef UINT16 PXENV_EXIT;
PXENV_STATUS	**Unsigned 16-bit PXE status code.** Typedef UINT16 PXENV_STATUS;
SEGDESC	**Protected mode segment descriptor.** Typedef struct s_SEGDESC { UINT16 segment_address; UINT32 Physical_address; UINT16 Seg_Size; } t_SEGDESC;
SEGOFF16	**Segment/Selector and 16-bit offset.** Typedef struct s_SEGOFF16 { OFF16 offset; SEGSEL segment; } SEGOFF16;
SEGSEL	**Unsigned 16-bit segment address or protected mode selector**. Typedef UINT16 SEGSEL;
UDP_PORT	**Communication port/socket number.** Typedef UINT16 UDP_PORT;
UINT8	**Unsigned 8-bit integer.** Typedef unsigned char UINT8;
UINT16	**Unsigned 16-bit integer.** Typedef unsigned short UINT16;
UINT32	**Unsigned 32-bit integer.** Typedef unsigned long UINT32;

1.4 Required vs. Recommended Features

Required, *Recommended*, and *Optional* are used in these guidelines to define the disposition of PXE features. The terms are described below:

Required — These are the basic features that must be implemented.

Recommended — These features provide improved capabilities to the end-user, improve

	manageability, or add functionality supported by the operating systems or software layers below the operating system, such as the BIOS. They are not required, but it is strongly suggested that they be implemented. It is generally expected that "Recommended" features may become "Required" in future revisions of this specification.
Optional	These features are not required.
Must	Required
Should	Recommended
May	Optional

1.5 Overview

1.5.1 PXE Protocol

PXE is defined on a foundation of industry-standard Internet protocols and services that are widely deployed in the industry, namely TCP/IP, DHCP, and TFTP. These standardize the *form* of the interactions between clients and servers. To ensure that the *meaning* of the client-server interaction is standardized as well, certain vendor option fields in DHCP protocol are used, which are allowed by the DHCP standard. The operations of standard DHCP and/or BOOTP servers (that serve up IP addresses and/or NBPs) will not be disrupted by the use of the extended protocol. Clients and servers that are aware of these extensions will recognize and use this information, and those that do not recognize the extensions will ignore them.

In brief, the PXE protocol operates as follows. The client initiates the protocol by broadcasting a DHCPDISCOVER containing an extension that identifies the request as coming from a client that implements the PXE protocol. Assuming that a DHCP server or a Proxy DHCP server implementing this extended protocol is available, after several intermediate steps, the server sends the client a list of appropriate Boot Servers. The client then discovers a Boot Server of the type selected and receives the name of an executable file on the chosen Boot Server. The client uses TFTP to download the executable from the Boot Server. Finally, the client initiates execution of the downloaded image. At this point, the client's state must meet certain requirements that provide a predictable execution environment for the image. Important aspects of this environment include the availability of certain areas of the client's main memory, and the availability of basic network I/O services.

1.5.1.1 Deployment of servers

On the server end of the client-server interaction there must be available services that are responsible for providing redirection of the client to an appropriate Boot Server. These redirection services may be deployed in two ways:

1. **Combined standard DHCP and redirection services.** The DHCP servers that are supplying IP addresses to clients are modified to become, or are replaced by servers that serve up IP addresses for all clients and redirect PXE-enabled clients to Boot Servers as requested.
2. **Separate standard DHCP and redirection services.** PXE redirection servers (Proxy DHCP servers) are added to the existing network environment. They respond only to PXE-enabled clients, and provide only redirection to Boot Servers.

Each PXE Boot Server must have one or more executables appropriate to the clients that it serves.

1.5.1.2 Deployment of Clients

PXE does not specify the operational details and functionality of the NBP that the client receives from the server. However, the intent is that running this executable will result in the system's being ready for use by its user. At a minimum, this means installing an operating system, drivers, and software appropriate to the client's hardware configuration. It might also include user-specific system configuration and application installation.

PXE specifies the protocols by which a client requests and downloads an executable image from a Boot Server and the minimum requirements on the client execution environment when the downloaded image is executed.

1.5.2 PXE APIs

To enable the interoperability of clients and downloaded bootstrap programs, the client PXE code provides a set of services for use by the BIOS or a downloaded NBP.

The API services provided by PXE for use by the BIOS or NBP are:

- **Preboot Services API.** Contains several global control and information functions.
- **Trivial File Transport Protocol (TFTP) API.** Enables opening and closing of TFTP connections, and reading packets from and writing packets to a TFTP connection.
- **User Datagram Protocol (UDP) API.** Enables opening and closing UDP connections, and reading packets from and writing packets to a UDP connection.
- **Universal Network Driver Interface (UNDI) API.** Enables basic control of and I/O through the client's network interface device. This allows the use of universal protocol drivers such that the same universal driver can be used on any network interface that implements this API.

The following diagram illustrates the relationship between the NBP (the remote boot program) and the PXE APIs.

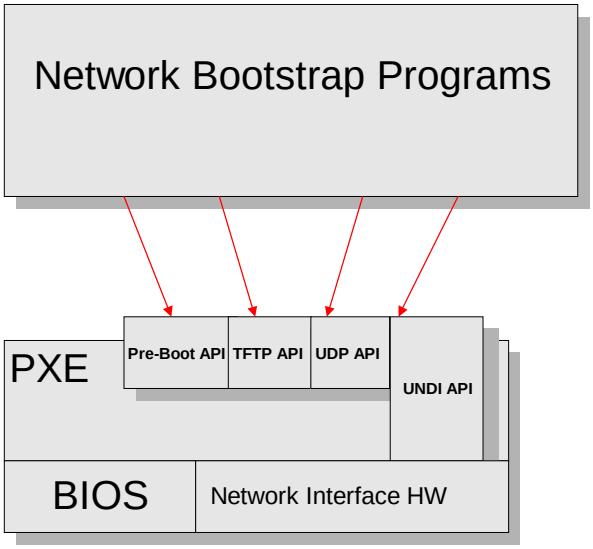

Figure 1-1 PXE APIs

2. PXE Client / Server Protocol

The description of PXE Client / Server Protocol assumes knowledge of the standard DHCP/BOOTP protocols.

2.1 Relationship to the Standard DHCP Protocol

The initial phase of this protocol piggybacks on a subset of the DHCP protocol messages to enable the client to discover a Boot Server, that is, a server that delivers executables for new system setup. The client *may* use the opportunity to obtain an IP address, which is the expected behavior, but it is not required. Clients that do obtain an IP address using DHCP or BOOTP must implement the protocol as specified in RFC 2131, even though not all possible messages and states of that protocol are described or mentioned in this protocol specification. The points at which this protocol piggybacks or otherwise interacts with the standard DHCP protocol are also noted.

The second phase of this protocol takes place between the client and a Boot Server, and uses the DHCP message format simply as a convenient format for communication. This second phase of the protocol is otherwise unrelated to the standard DHCP services.

2.2 Protocol Details

The protocol is a combination of an extension of DHCP (through the use of several new DHCP Option tags) and the definition of simple packet transactions that use the DHCP packet format and options to pass additional information between the client and server. This added complexity is introduced by the requirement to operate without disturbing existing DHCP services.

In this protocol, DHCP options fields are used to do the following:

- Distinguish between DHCPDISCOVER and DHCPREQUEST packets sent by a client as part of this extended protocol from other packets that the DHCP server or Boot Server might receive.
- Distinguish between DHCPOFFER and DHCPACK packets sent by a DHCP or Proxy DHCP server as part of this extended protocol from other packets that the client may receive.
- Convey the client system's ID to the DHCP and Boot Server (in other words, UUID).
- Convey the client system's architecture type to the DHCP and Boot Server.
- Convey the Boot Server type from which the client is requesting a response.

Based on any or all of the client network adapter type, system architecture type, and client system ID, the Boot Server returns to the client the file name (on the server) of an appropriate executable. The client downloads the specified executable into memory and executes it. The function of this executable is not specified by these guidelines.

2.2.1 PXE Boot

This section gives a step-by-step synopsis of the PXE protocol. A detailed description of packet formats and client and server actions appears later in this section. Note that this version of the PXE specification introduces remote-boot authentication. PXE remote-boot authentication relies on the presence of platform security capabilities as described in the [BIS] specification.

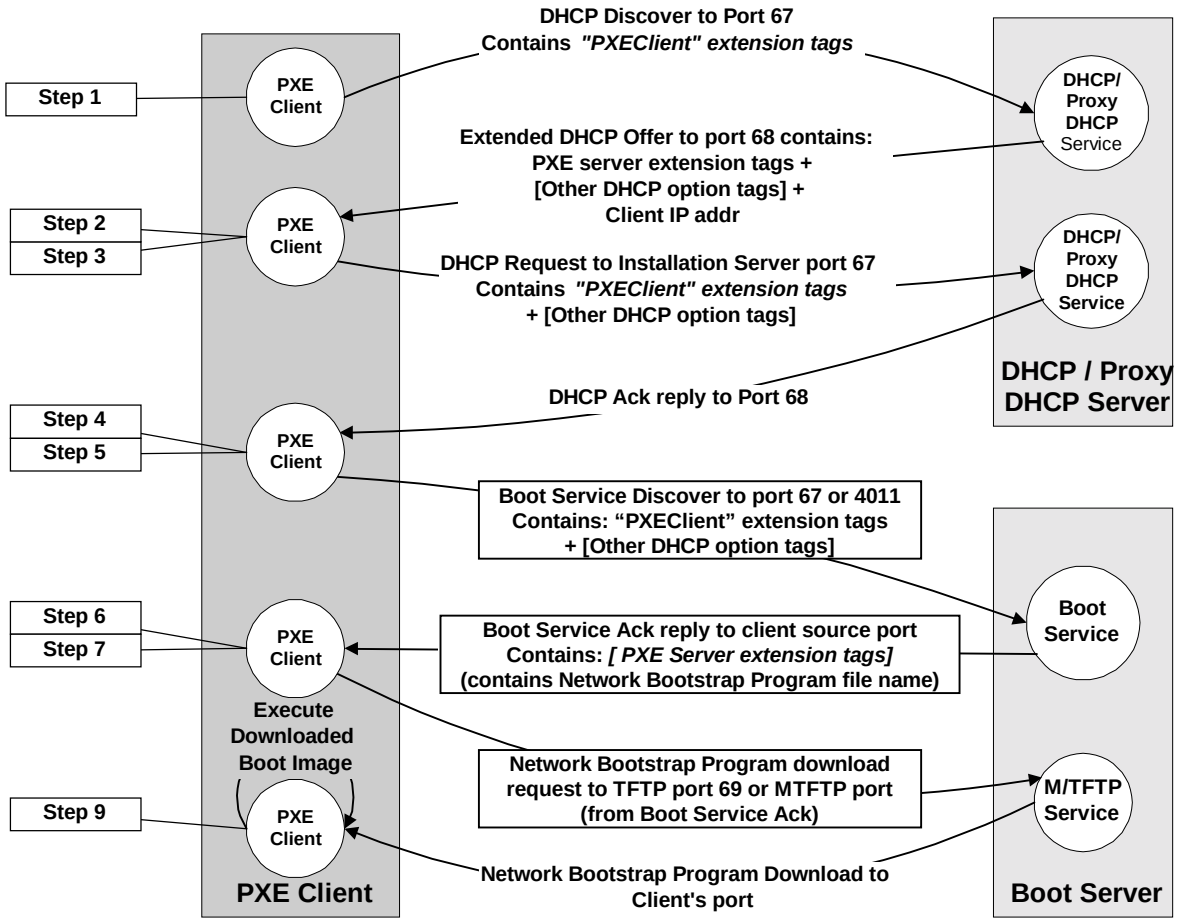

Figure 2-1 PXE Boot

Step 1. The client broadcasts a DHCPDISCOVER message to the standard DHCP port (67). An option field in this packet contains the following:

- A tag for client identifier (UUID).
- A tag for the client UNDI version.
- A tag for the client system architecture.
- A DHCP option 60, Class ID, set to "PXEClient:Arch:xxxxx:UNDI:yyyzzz".

Step 2. The DHCP or Proxy DHCP Service responds by sending a DHCPOFFER message to the client on the standard DHCP reply port (68). If this is a Proxy DHCP Service, then the client IP address field is null (0.0.0.0). If this is a DHCP Service, then the returned client IP address field is valid.

At this point, other DHCP Services and BOOTP Services also respond with DHCP offers or BOOTP reply messages to port (68). Each message contains standard DHCP parameters: an IP address for the

client and any other parameters that the administrator might have configured on the DHCP or Proxy DHCP Service.

The timeout for a reply from a DHCP server is standard. The timeout for re-broadcasting to receive a DHCPOFFER with PXE extensions, or a Proxy DHCPOFFER is based on the standard DHCP timeout but is substantially shorter to allow reasonable operation of the client in standard BOOTP or DHCP environments that do not provide a DHCPOFFER with PXE extensions. (See below.)

Step 3. From the DHCPOFFER(s) that it receives, the client records the following:

- The Client IP address (and other parameters) offered by a standard DHCP or BOOTP Service.
- The Boot Server list from the Boot Server field in the PXE tags from the DHCPOFFER.
- The Discovery Control Options (if provided).
- The Multicast Discovery IP address (if provided).

Step 4. If the client selects an IP address offered by a DHCP Service, then it must complete the standard DHCP protocol by sending a request for the address back to the Service and then waiting for an acknowledgment from the Service. If the client selects an IP address from a BOOTP reply, it can simply use the address.

Step 5. The client selects and discovers a Boot Server. This packet may be sent broadcast (port 67), multicast (port 4011), or unicast (port 4011) depending on discovery control options included in the previous DHCPOFFER containing the PXE service extension tags. This packet is the same as the initial DHCPDISCOVER in Step 1, except that it is coded as a DHCPREQUEST and now contains the following:

- The IP address assigned to the client from a DHCP Service.
- A tag for client identifier (UUID)
- A tag for the client UNDI version.
- A tag for the client system architecture.
- A DHCP option 60, Class ID, set to “PXEClient:Arch:xxxxx:UNDI:yyyzzz”.
- The Boot Server type in a PXE option field

Step 6. The Boot Server unicasts a DHCPACK packet back to the client on the client source port. This reply packet contains:

- Boot file name.
- MTFTP[1] configuration parameters.
- Any other options the NBP requires before it can be successfully executed.

Step 7. The client downloads the executable file using either standard TFTP (port69) or MTFTP (port assigned in Boot Server Ack packet). The file downloaded and the placement of the downloaded code in memory is dependent on the client’s CPU architecture.

Step 8. The PXE client determines whether an authenticity test on the downloaded file is required. If the test is required, the client sends another DHCPREQUEST message to the boot server requesting a credentials file for the previously downloaded boot file, downloads the credentials via TFTP or MTFTP, and performs the authenticity test.

Step 9. Finally, if the authenticity test succeeded or was not required, then the PXE client initiates execution of the downloaded code

[1] ***Multicast Trivial File Transfer Protocol as defined by this document through the use of DHCP encapsulated vendor options.***

2.2.2 Protocol Timeouts

The following flow chart specifies the required timeouts at various stages of the remote boot process. The DHCP timeouts are specified in RFC 2131 and are noted for reference in the diagram. The timeouts specific to the PXE boot process must be implemented as specified in Figure 2-2 PXE Client Timeouts.

The purpose of the timeouts is to ensure the PXE client gives precedence to servers supplying "PXEClient" specific configuration tags. The PXE boot ROM must function as a normal DHCP boot ROM in the absence of a PXE specific response. However, the PXE boot ROM must wait for specified times to see if a PXE response is available before using a non-PXE configuration.

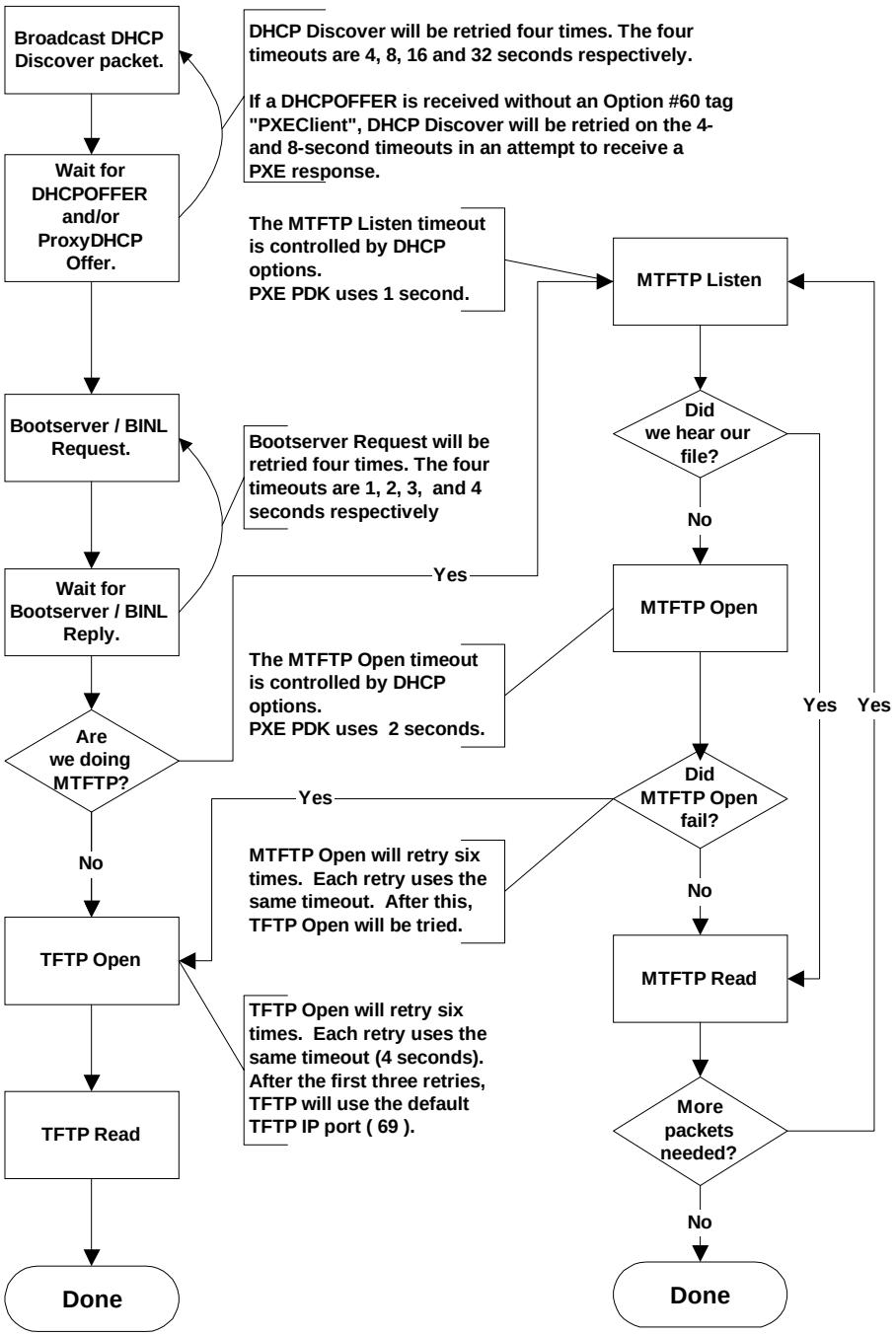

Figure 2-2 PXE Client Timeouts

2.2.3 Proxy DHCP

The PXE DHCP options may be supplied by the DHCP service or a Proxy DHCP service. This Proxy DHCP service may reside on the same server as the DHCP service, or it may be located on a separate server. A Proxy DHCP service on the same server as the DHCP service is illustrated in Figure 2-3. In this case, the Proxy DHCP service is listening to UDP port (4011), and communication with the

Proxy DHCP service occurs after completing the standard DHCP protocol. Proxy DHCP uses port (4011) because it cannot share port (67) with the DHCP service. The PXE client knows to interrogate the Proxy DHCP service because the DHCPOFFER from the DHCP service contains an Option #60 “PXEClient” tag without corresponding Option #43 tags or a boot file name.

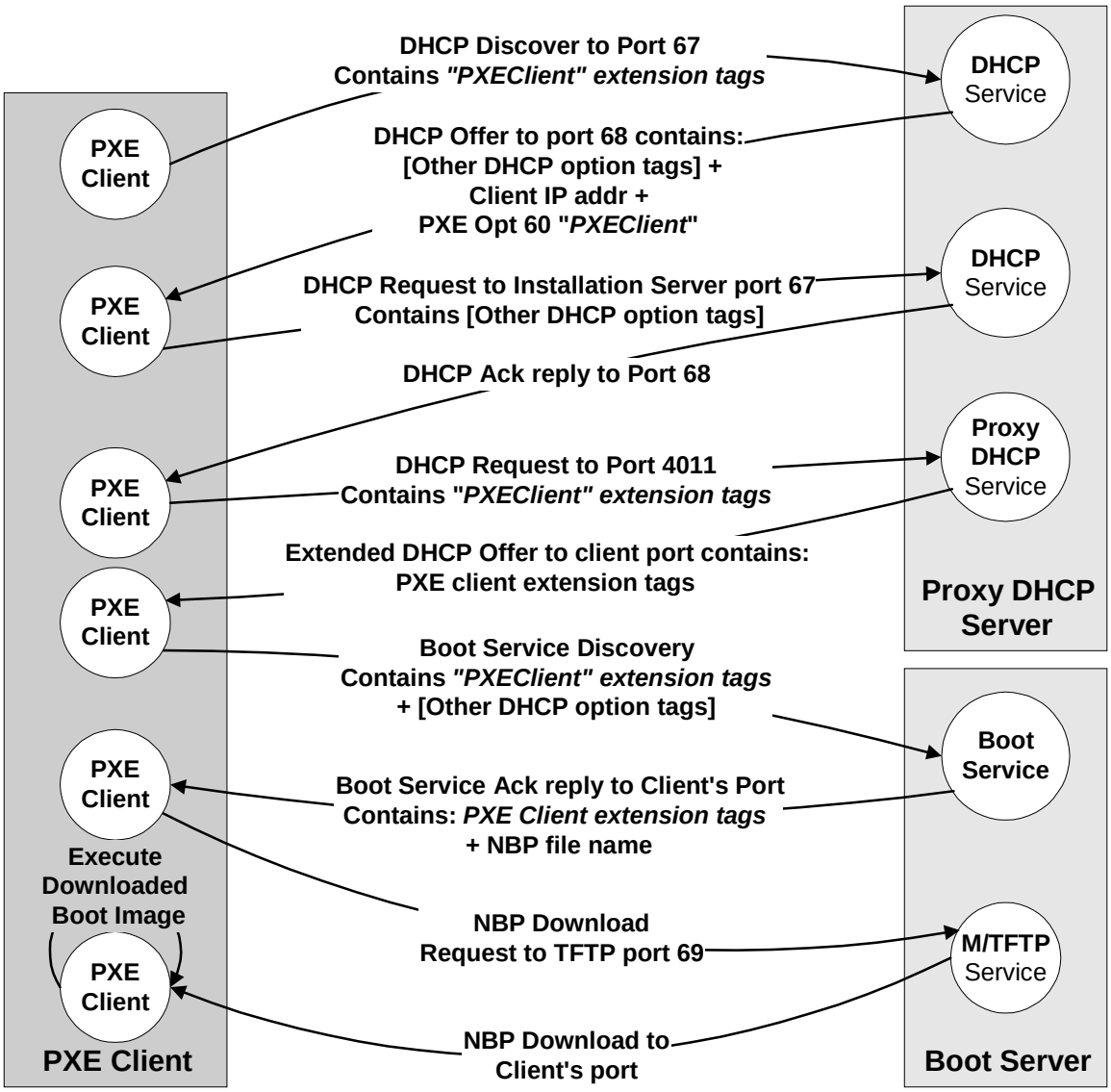

Figure 2-3 PXE Client Response to DHCP Server Containing a Proxy DHCP Service

Figure 2-4 illustrates the case of a Proxy DHCP service and the DHCP service on different servers. In this case, the Proxy DHCP service listens to UDP port (67) and responds in parallel with the DHCP service.

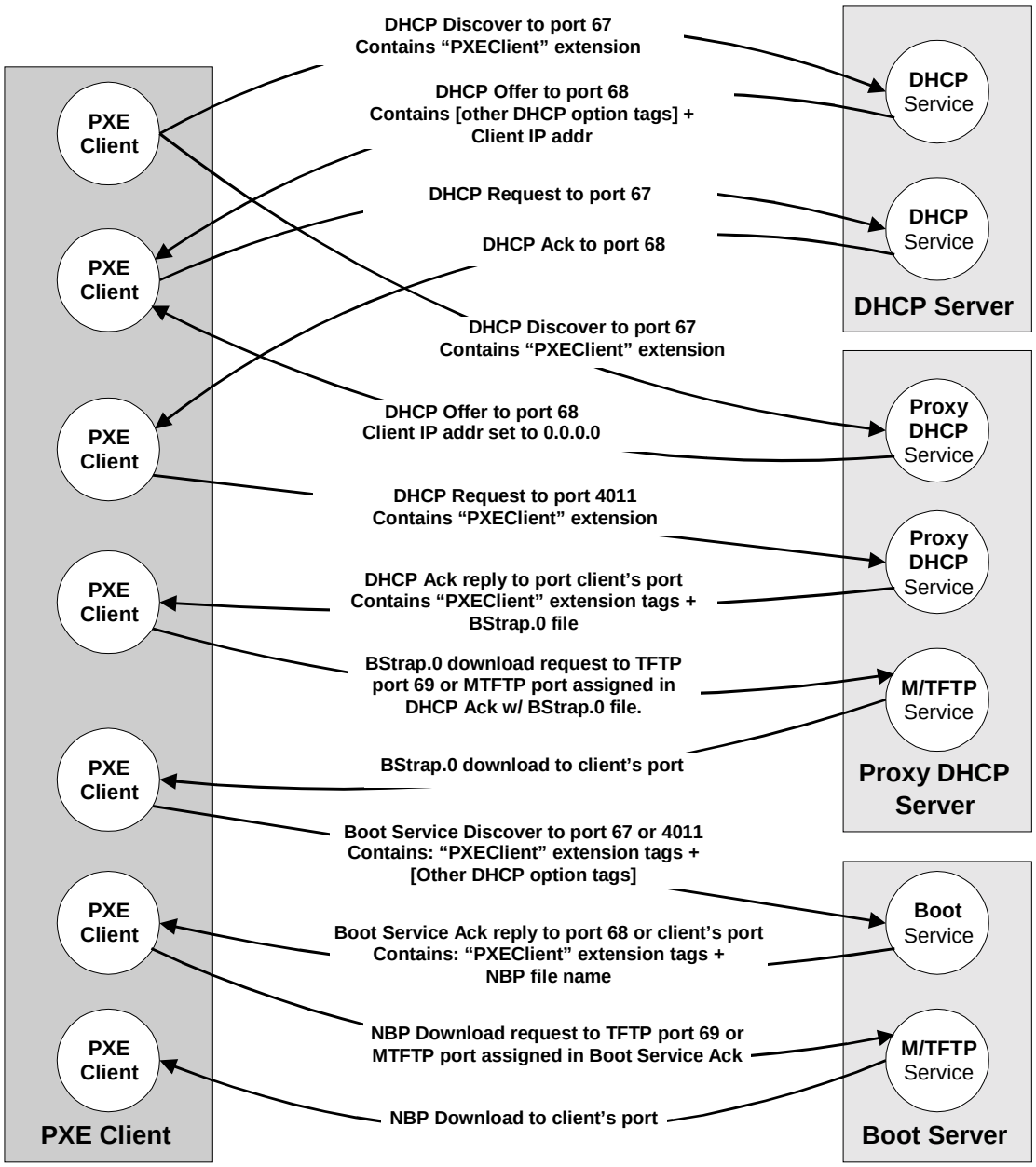

Figure 2-4 PXE Client Response to DHCP Server Supplying Boot Service Discovery Code

2.3 DHCP Tags used for PXE Protocol

Table 2-1 lists all the PXE DHCP tags used by the Client, the Boot Server or the DHCP or Proxy DHCP service. Subsequent sections break out the tag use for each participant.

Table 2-1 PXE DHCP Options (Full List)

Tag Name	*Tag #*	*Length*	*Type*	*Data Field*	
Client machine identifier (UUID).	97 & 61	17	Type (1) = 0	UUID(16)	Required Note #1
Client network interface identifier.	94	3	Type (1) = 1	UNDI – Major ver(1), Minor ver(1)	
Client system architecture.	93	2	0 = IA x86 PC(2) 1 = NEC/PC98(2) 2 = IA64 PC.(2) 3 = DEC Alpha (2) 4 = ArcX86 (2) 5 = Intel Lean Client (2)		
Parameter Request List	55	Varies	This parameter request list is the minimum that must be implemented by PXE Base-Code option ROMs. subnet(1), router(3), vendor(43), class(60) vendor options (128 through 135).		Required
Class Identifier	60	32	"PXEClient:Arch:xxxxx:UNDI:yyyzzz" – used for transactions between client and server. "PXEServer" – used for transactions between servers. (These strings are case sensitive. This field must not be null terminated.) The information from tags 93 and 94 is embedded in the Class Identifier string xxxxx = Client Sys Architecture 0 – 65535 yyy = UNDI Major version 0 – 255 zzz = UNDI Minor version 0 – 255 Delimiter is ":" (colon)		Required
Vendor Options	43	Varies	Encapsulated options below. Multiple DHCP_VENDOR options can be used.		
Message Type	53	1	1=DHCPDISCOVER, 2=DHCPOFFER, 3=DHCPREQUEST, 4=DHCPDECLINE, 5=DHCPACK, 6=DHCPNAK, 7=DHCPRELEASE, 8=DHCPINFORM		Required
Server ID	54	4	a1, a2, a3, a4		
Message Length	57	2	Max DHCP message length(2)		Required
End of Options	255	None	None		Required
PXE Options	**1-63**	**varies**	**Reserved for PXE use**		
PXE_MTFTP_IP	1	4	**Multicast IP Address(4)** Multicast IP address of boot file.		Recom-mended Note #2
PXE_MTFTP_ CPORT	2	2	**UDP Port Number Intel order(2)** UDP port that client should monitor for MTFTP responses.		
PXE_MTFTP_ SPORT	3	2	**UDP Port Number Intel order(2)** UDP port that MTFTP servers are using to listen for MTFTP open requests.		
PXE_MTFTP_ TMOUT	4	1	**Open Timeout(1)** Number of seconds a client must listen for activity before trying to start a new MTFTP transfer.		

Tag Name	*Tag #*	*Length*	*Type*	*Data Field*	
PXE_MTFTP_ DELAY	5	1	**Reopen Timeout(1)** Number of seconds a client must listen before trying to restart a MTFTP transfer.		
PXE_DISCOVERY_ CONTROL	6	1	**(Bit field. Bit 0 is the least significant bit.)(1)** bit 0 = If set, disable broadcast discovery. bit 1 = If set, disable multicast discovery. bit 2 = If set, only use/accept servers in PXE_BOOT_SERVERS. bit 3 = If set, and a boot file name is present in the initial DHCP or ProxyDHCP offer packet, download the boot file (do not prompt/menu/discover). bit 4-7 = Must be 0. If this tag is not supplied all bits assumed to be 0		Required Note #3
DISCOVERY_ MCAST_ADDR	7	4	**Multicast IP-addr(4)** Boot Server discovery multicast IP address. Boot Servers capable of multicast discovery must listen on this multicast address. This option is required if the multicast discovery disable bit (bit 1) in the PXE_DISCOVERY_CONTROL option is not set.		Recom-mended
PXE_BOOT _SERVERS	8	varies	**Boot Server type(2)** Type 0 = PXE bootstrap server Type 1 = Microsoft Windows† NT† Boot Server Type 2 = Intel LCM Boot Server Type 3 = DOS/UNDI Boot Server Type 4 = NEC ESMPRO† Boot Server Type 5 = IBM WSoD Boot Server Type 6 = IBM LCCM† Boot Server Type 7 = CA Unicenter† TNG Boot Server Type 8 = HP OpenView† Boot Server Type 9 through 32767 = reserved Type 32768 through 65534 = vendor use Type 65535 = PXE API Test server	**IPcnt(1), IP-addr-list(IPcnt*4), type(2)….** If IPcnt is zero for a server type, the client may accept offers from any boot server of that type. Boot Servers must not respond to discovery requests of types they do not support.	Required for PXE client. Note #3
PXE_BOOT_MENU	9	varies	**Boot Server type(2)** Type 0 = Local boot (remove base-code and UNDI from RAM, if possible)	**desclen(1), “description”, Boot Server type(2)….** Boot “order” is implicit in the menu order. “desclen” is length of “description” “desclen” cannot be 0.	Required Note #4

Tag Name	*Tag #*	*Length*	*Type* *Data Field*	
PXE_MENU_ PROMPT	10	varies	**timeout(1), "prompt"** The timeout is the number of seconds to wait before auto-selecting the first boot menu item. The prompt is displayed followed by the number of seconds remaining before the first item in the boot menu is auto-selected. If <F8> is pressed, the menu must be displayed. If this option is not provided, the menu must be displayed without prompt and timeout. If the timeout is 0, the first item in the menu must be auto-selected. If the timeout is 255, the menu and prompt must be displayed without auto-selecting or timeout.	
PXE_MCAST_ ADDRS_ALLOC	11	8	**McastIPbase(4), MIPblock(2), MIPrange(2)** McastIPBase is the starting address for multicast addresses. MIPblock = total size of multicast block. MIPrange = max number of multicast addresses available to any one Boot Server. MIPrange may equal MIPblock. The Boot service must randomly pick an address within the range defined by McastIPBase through McastIPBase+MIPblock-MIPrange as the base number for their MIPrange number of IP addresses. The Boot service manages base addresses for each boot tree internally.	Recom-mended for boot servers, not used by clients.
PXE_CREDENTIAL_ TYPES	12	varies	**Credentials type(4), …** The credential types retrieved from the security sub-system are to be stored in this option field in network order. This option is required for security requests and acknowledgments between the client and the server. There is no default or assumed value for this option.	Required for security. Note #5
Loader Options	**64-127**	**varies**	**(Boot Server specific)**	
PXE_BOOT_ITEM	71	4	**Boot Server type(2), layer(2)** Layer 0 = First file of selected Boot Server type. MSbit of layer field indicates credentials for the of selected file (for example, layer 8000h is credentials for layer 0000h.) If this tag is missing, type 0 and layer 0 is assumed.	Required Note #6
Vendor Options	**128-254**	**varies**	**(Vendor NBP specific)**	
PXE_END	255	None		Required

Note #1

The Client UUID field specifies a universally unique ID (UUID), retrieved from the client system. The client must have a UUID and must report it in tag #97 and #61.

See UUID programming notes in Section 5.2.1 UUID Support.

The *Client Network Interface Identifier* specifies the version of the UNDI API (described below) that will support a universal network driver. The UNDI interface must be supported and its version reported in tag #94.

The UNDI type field must have a major version of 2 and a minor version of 1 for this version of the protocol. (Future versions may recognize more tags based on this version number.)

The *Client System Architecture* identifier specifies the system architecture of the client. This identifier is required and must be reported in tag #93.

Note #2

MTFTP is recommended. These options define the client/server port numbers and open/re-open timeouts that must be used in MTFTP open/read requests.

MTFTP IP Addr is the multicast IP address the client must use to receive the image file.

MTFTP Client UDP is the port the client must listen on to receive the image file.

MTFTP Server UDP is the port the client must use to communicate with the MTFTP service. The client binds to the MTFTP UDP port and waits for the duration of the MTFTP transmission start delay to receive packets.

MTFTP Start Delay is the timeout to begin receiving image file packets before attempting to become the MTFTP acknowledging client (master client) upon initial connection to the MTFTP service.

MTFTP Timeout Delay is the delay multiplied by the percentage of the file received, the client must wait before attempting to become the MTFTP acknowledging client (master client) upon cessation of packet transmissions during an ongoing MTFTP transfer.

Note #3

These options control the type of boot server discovery mechanisms used by clients. Clients must use discovery methods in this order:

1. **Multicast**. If the client supports multicast discovery and multicast discovery is enabled (PXE_DISCOVERY_CONTROL, option #43 tag #6 does exist OR it does exist and bit1 is not set.) and a multicast discovery IP address is available. (DISCOVERY_MCAST_ADDR exists.)
2. **Broadcast**. If broadcast discovery is enabled, (PXE_DISCOVERY_CONTROL, option #43 tag #6 doesn't exist OR it does exist and bit 0 is not set).
3. **Unicast**. If a Boot Server list is available, (PXE_BOOT_SERVERS, Option #43 tag #8).

If PXE_DISCOVERY_CONTROL bit 2 is set, the client may still use multicast and broadcast discovery (if it is permitted by bits 0 and 1); but the client may only accept replies from servers that are identified in the PXE_BOOT_SERVERS option.

Note #4

These options define the information, if any, displayed by the client during a network boot.

Note #5

The client must fill in this option when requesting credentials (MSbit in layer number is set). Boot servers must not respond if they do not support the requested credential type. Clients requesting credentials must ignore any server response that does not have the credential option. Clients must include all the supported credential types when doing a layer zero discovery. Clients must use the same credential type, selected in the layer zero discovery, for all subsequent layers. If the client lists more than one credential type in the discover request, the boot server must respond with the one credential type that will be used.

Note #6

This option is required to discover Boot Servers. Only the client may change the type field; either the client or the server may change the layer field. Layer 0 always indicates the first boot file for a particular Boot Server type. Boot Servers capable of providing the boot file requested in the PXE_BOOT_ITEM must respond. Boot Servers not capable of providing the boot file requested must not respond.

2.4 Client Behavior

Client behavior for initiation, discovery reply, Boot Service request, Boot Service reply, and NBP download and execution are summarized in this section.

Sending a PXE Client message requires the use of DHCP Option fields. All PXE Client packets provide the same extended DHCP information in these options. This includes DHCPREQUEST messages used to communicate with the server to which the PXE Client has been redirected. Other fields and options may be different between the packets, based on the standard DHCP protocol.

2.4.1 PXE Option Precedence

Depending on the configuration of the system it is possible for the client to receive the same PXE DHCP option type from multiple sources. For example, the client could receive a PXE DHCP Offer from both a DHCP server and a Proxy server. The precedence from high to low that the client must apply is:

DHCP – takes precedence over all other sources of an option.

Proxy – takes precedence over Boot Servers.

Within a level the client may choose any one of the replies it receives, but must select all options from the same reply (e.g. When receiving three proxy replies, the client is free to use any of them, buy it may not select options from more than one of them.).

Boot Servers are the lowest precedence. As such, options in the DHCP and Proxy may be used to override any options a boot server may send. Further, a boot server is unable to override any options set by the DHCP and Proxy.

2.4.2 DHCPDISCOVER

To initiate the interchange between the client and server, the client broadcasts a DHCPDISCOVER packet to the standard DHCP server UDP port (67). The contents of this message must be as described in RFC 2131 for a DHCPDISCOVER message, with the addition of PXE Client option fields. The format of these options is specified in Table 2-2; fields marked with an asterisk contain unspecified values.

Table 2-2 DHCPDISCOVER Packet to DHCP/Proxy DHCP Server

DHCP Header				
Field (length)	**Value**	**Comment**		
op (1)	1	Code for BOOTP REQUEST		
htype (1)	*			
hlen (1)	*			
hops (1)	*			
xid (4)	*			
secs (2)	*			
flags (2)	*			
ciaddr (4)	0.0.0.0	PXE client always sets this value to 0.0.0.0		
yiaddr (4)	*	Client's IP address. Provided by server		
siaddr (4)	*	Next bootstrap server IP address		
giaddr (4)	*			
chaddr (16)	xx-xx-xx-xx-xx-xx-xx-xx	Client's MAC address		
sname (64)	*	Can be overloaded if using Opt 66		
bootfile (128)	*	Can be overloaded if using Opt 67		
99.130.83.99 (Magic Cookie)				
DHCP Options				
Tag Name	**Tag #**	**Length**	**Data Field**	
Client UUID/GUID	97 & 61	17	Type(1) 0 = UUID	UUID(16)
Client Network Type	94	3	1 = UNDI	
Client System Architecture	93	2	Architecture Type(2)	
Parameter Request List	9	Varies	This parameter request list is the minimum that must be implemented by PXE BC option ROMs. subnet(1), router(3), vendor(43), class(60)	
DHCP Message Type	53	1	1 = DHCPDISCOVER	
Message Length	57	2	Max DHCP message length(2)	
Class Identifier	60	32	"PXEClient:Arch:xxxxx:UNDI:yyyzzz"	

After sending the DHCPDISCOVER message, the client must be prepared to receive replies as described in the following section.

2.4.3 DHCPOFFER

In this state, the client is prepared to receive one or more *extended* DHCPOFFER replies from servers on the standard DHCP client UDP port (68). Sending a PXE Server message requires the use of DHCP Options. The format of these options is shown in Table 2-3; fields marked with an asterisk contain unspecified values.

Table 2-3 DHCPOFFER Packet from DHCP/Proxy DHCP Server

DHCP Header				
Field (length)	**Value**	**Comment**		
op (1)	2	Code for BOOTP REPLY		
htype (1)	*			
hlen (1)	*			
hops (1)	*			
xid (4)	*			
secs (2)	*			
flags (2)	*			
ciaddr (4)	0.0.0.0	Server always sets this value to 0.0.0.0		
yiaddr (4)	a0, a1, a2, a3	Client's IP address. Provided by server		
siaddr (4)	a0, a1, a2, a3	Next bootstrap server IP address		
giaddr (4)	*			
chaddr (16)	*	Client's MAC address		
sname (64)	*	Can be overloaded if using Opt 66		
bootfile (128)	*	Can be overloaded if using Opt 67		
99.130.83.99				
DHCP Options				
Tag Name	**Tag #**	**Length**	***Data Field***	
DHCP Message Type	53	1	2= DCHPOFFER	
Server Identifier	54	4	a1, a2, a3, a4	
Client Machine Identifier	97	17	Type(1) 0 = UUID	UUID(16)
Class Identifier	60	9	"PXEClient"	
Vendor Options	43	Varies	*Encapsulated options below.*	
PXE_DISCOVERY_CONTROL	6	1		
DISCOVERY_MCAST_ADDR	7	4	**Multicast IP-addr(4)**	
PXE_BOOT_SERVERS	8	varies	Boot Server type(2), IPcnt(1), IP-addr-list(IPcnt*4), Boot Server type(2)…	
PXE_BOOT_MENU	9	varies	Boot Server type(2), desclen(1), "description", Boot Server type(2)….	
PXE_MENU_PROMPT	10	varies	timeout(1), "prompt"	
PXE_END	255	None	None	

In this state, the client must also be prepared to receive one or more *standard* DHCPOFFER messages from servers. Each of these messages will contain configuration information as specified in RFC 2131. Each *extended* DHCPOFFER message can also contain configuration information as specified in RFC 2132. Which, of these configurations, if any, is used by the client is not defined by this specification. If the client decides to accept one of the configurations offered, then it must engage in further communications with the server as specified in RFC 2132.

2.4.4 Boot Server Discovery

To enter the Boot Server Discovery state, the client must have an IP address. Also, the client must have received one or more *extended* DHCPOFFER messages, and therefore, know the type of one or

more Boot Servers. The client discovers one of these Boot Servers by sending a DHCPREQUEST message to the boot server using either unicast, multicast, or broadcast per the discovery instructions in Option #43, Tag #6 (PXE_DISCOVERY_CONTROL). Table 2-4 lists the required values in the fields of this message; fields marked with an asterisk contain unspecified values.

Table 2-4 Boot Server Request Packet

DHCP Header				
Field (length)	**Value**	**Comment**		
op (1)	1	Code for BOOTP REQUEST		
htype (1)	*			
hlen (1)	*			
hops (1)	*			
xid (4)	*			
secs (2)	*			
flags (2)	*			
ciaddr (4)	a0, a1, a2, a3	PXE client sets this value to the clients IP address if the client has an IP address		
yiaddr (4)	0.0.0.0	Must be 0.		
siaddr (4)	0.0.0.0	Must be 0.		
giaddr (4)	0.0.0.0			
chaddr (16)	*xx-xx-xx-xx-xx-xx-xx-xx	Client's MAC address		
sname (64)	*	Can be overloaded if using Opt 66		
bootfile (128)	*	Can be overloaded if using Opt 67		
99.130.83.99				
DHCP Options				
Tag Name	**Tag #**	**Length**	**Type**	**Data Field**
Client UUID/GUID	97 & 61	17	Type (1) 0 = UUID	UUID(16)
Client Network Device Interface Type	94	3	Type (1) = 1	UNDI – Major Ver(1), Minor Ver(1)
Message Type	53	1	3 = DHCPREQUEST 8 = DHCPINFORM	
Client System Architecture	93	2		
Class Identifier	60	32	"PXEClient Arch:xxxxx:UNDI:yyyzzz"	
Vendor Options	43	varies	Encapsulated options below	
PXE_BOOT_ITEM	71	4	Boot Server type(2), layer(2)	
PXE_END	255	None		

2.4.5 Boot Server Reply

In the Boot Server Reply state, the client must be prepared to receive an *extended* DHCPACK message from the Boot Service. Table 2-5 lists the values the boot server may send to the client:

Table 2-5 Boot Server ACK Packet

DHCP Header				
Field (length)	**Value**	**Comment**		
op (1)	2	Code for BOOTP REPLY		
htype (1)	*			
hlen (1)	*			
hops (1)	*			
xid (4)	*			
secs (2)	*			
flags (2)	*			
ciaddr (4)	0.0.0.0	Server always sets this value to 0.0.0.0		
yiaddr (4)	a0, a1, a2, a3	Client's IP address. Provided by server		
siaddr (4)	a0, a1, a2, a3	Next bootstrap server IP address		
giaddr (4)	*			
chaddr (16)	*	Client's MAC address		
sname (64)	*	Can be overloaded if using Opt 66		
bootfile (128)	*	Can be overloaded if using Opt 67		
99.130.83.99				
DHCP Options				
Tag Name	**Tag #**	**Length**	**Type**	**Data Field**
DHCP Message Type	53	1	5 = DHCPACK	
Server Identifier	54	4	a1, a2, a3, a4	
Client UUID/GUID	97 & 61	17	Type (1) 0 = UUID	UUID(16)
Class Identifier	60	9	"PXEClient"	
Vendor Options	43	varies	Encapsulated options below	
MTFP IP Addr	1	4	a0, a1, a2, a3	
MTFTP Client UDP	2	2	Port Number (Intel order)	
MTFTP Server UDP	3	2	Port Number (Intel order)	
MTFTP Start Delay	4	1		
MTFTP Timeout Delay	5	1		
PXE_BOOT_ITEM	71	4	Boot Server type(2), layer(2)	
PXE_END	255	None		

The options fields in this message must, at a minimum, include the following:

- The *siaddr* field should be null. (The Boot Server should include the MTFTP service. If not, it is the responsibility of the boot server to insure the availability of the MTFTP server to which the client has been redirected. In general it is strongly recommended that the MTFTP service reside on the bootserver to insure that a response to a client will inherently include the guarantee of boot file availability.)
- The boot file name (PXE_BOOT_ITEM) must be included.
- DHCP message type (DHCPACK).
- The Server Identifier (address of the responding Boot Server) must be included.
- Class Identifier ("PXEClient").
- Boot Server Type and Layer the server is providing to the client.

The MTFTP options must be included if the client is to perform a multicast file transfer. After receiving this message, the client moves to the executable download state.

2.4.6 Network Bootstrap Program (NBP) Download

In the NBP download state, the client is to download all or some portion of the NBP using the standard TFTP. The portion of the file downloaded and the placement of the downloaded code in memory is dependent on the client's CPU architecture.

For systems based on Intel Architecture, the entire NBP is downloaded into the client PC starting at location 07C00h. The TFTP/MTFTP session that was used to download the NBP is terminated and the logical network connection to the TFTP server is closed.

2.4.7 NBP Authentication

After downloading the NBP, the client must perform the NBP authentication procedure. PXE remote-boot authentication relies on the presence of platform security capabilities as described in the [BIS] specification. The PXE client must perform the following procedure to determine whether authentication of the NBP (Network Bootstrap Program, i.e. the remote boot file) is required, and to perform the authenticity test.

1. Determine whether an implementation of the Boot Integrity Services (BIS) API is present by searching for an SMBIOS structure of the appropriate type as described in [BIS]. If the structure is not present, then the authentication test is not required and the remaining steps in this procedure are not required.
2. Use the appropriate BIS function to determine the current setting of the BIS Boot Object Authorization Check Flag. If the flag is set to FALSE, then the authentication test is not required and the remaining steps in this procedure are not required. If the flag is set to TRUE, then the client must perform steps 3 and 4.
3. Obtain credentials for the NBP from the boot server.
4. Call the BIS function *VerifyBootObject*, supplying the downloaded NBP and credentials as input. If the output from *VerifyBootObject* indicates that an error occurred or that verification failed, then the NBP authentication fails and the PXE client must not initiate execution of the NBP. If the output from *VerifyBootObject* indicates that the verification succeeded, then NBP authentication succeeds.

To obtain credentials for the NBP from the boot server during this procedure, the client sends a *unicast* DHCPREQUEST message to the boot server from which the NBP was downloaded. (Note: The client *must not* send this request as a multicast or broadcast.) The following table lists the required values in the fields of this message; fields marked with an asterisk contain unspecified values. After sending this message, the client moves to the Boot Server Credentials Reply state.

Table 2-6 Boot Server Credentials Request Packet

DHCP Header				
Field (length)	**Value**	**Comment**		
op (1)	1	Code for BOOTP REQUEST		
htype (1)	*			
hlen (1)	*			
hops (1)	*			
xid (4)	*			
secs (2)	*			
flags (2)	*			
ciaddr (4)	a0.a1.a2.a3	PXE client sets this value to the client's IP address if the client has an IP address		
yiaddr (4)	a0, a1, a2, a3	Client's IP address. Provided by DHCP server		
siaddr (4)	a0, a1, a2, a3	Server's IP address		
giaddr (4)	0.0.0.0			
chaddr (16)	*xx-xx-xx-xx-xx-xx-xx-xx	Client's MAC address		
sname (64)	*	Can be overloaded if using Opt 66		
bootfile (128)	*bootfile name*	NBP name sent by boot server		
99.130.83.99				
DHCP Options				
Tag Name	**Tag #**	**Length**	**Type**	**Data Field**
Client UUID/GUID	97	17	Type (1) 0 = UUID	UUID(16)
Client Network Device Interface Type	94	3	1 = UNDI	Type 1 = Major Ver(1), Minor Ver(1)
DHCP Message Type	53	1	3 = DHCPREQUEST	
Client System Architecture	93	2		
Class Identifier	60	9	"PXEClient:Arch:xxxxx:UNDI:yyyzzz"	
DHCP_VENDOR	43	Varies	Encapsulated options below	
PXE_PAD	0	None	None	
PXE_BOOT_ITEM	71	4	Boot Server type(2), layer(2)	
PXE_CREDENTIAL_TYPES	12	Varies	Credential types (4)	
PXE_END	255	None		

Note: In the PXE_BOOT_ITEM option in this message, the MSB of the *layer* data field is 1. This distinguishes a request for credentials from a request for the boot file itself.

Note: The *bootfile* field of this message must contain the boot file name exactly as the client received it from the boot server.

2.4.8 Boot Server Credentials Reply

The client enters the Boot Server Credentials Reply state only if it determines that a boot file authenticity test is needed and sends a credentials request to the boot server. In the Boot Server

Credentials Reply state, the client must be prepared to receive an *extended* DHCPACK message from the Boot Service. The following table lists the required values in the fields of this message:

Table 2-7 Boot Server Credentials ACK Packet

DHCP Header				
Field (length)	**Value**	**Comment**		
op (1)	2	Code for BOOTP REPLY		
htype (1)	*			
hlen (1)	*			
hops (1)	*			
xid (4)	*			
secs (2)	*			
flags (2)	*			
ciaddr (4)	0.0.0.0	Server always sets this value to 0.0.0.0		
yiaddr (4)	a0, a1, a2, a3	Client's IP address. Provided by server		
siaddr (4)	a0, a1, a2, a3	Next bootstrap server IP address		
giaddr (4)	*			
chaddr (16)	*	Client's MAC address		
sname (64)	*	Can be overloaded if using Opt 66		
bootfile (128)	*	Can be overloaded if using Opt 67		
99.130.83.99				
DHCP Options				
Tag Name	**Tag #**	**Length**	**Type**	**Data Field**
DHCP Message Type	53	1	5 = DHCPACK	
Server Identifier	54	4	a1, a2, a3, a4	
Client UUID/GUID	97 & 61	17	Type (1) 0 = UUID	UUID(16)
Class Identifier	60	9	"PXEClient"	
DHCP_VENDOR	43	Varies	Encapsulated options below	
PXE_PAD	0	None	None	
MTFP IP Addr	1	4	a0, a1, a2, a3	
MTFTP Client UDP	2	2	Port Number (Intel order)	
MTFTP Server UDP	3	2	Port Number (Intel order)	
MTFTP Start Delay	4	1		
MTFTP Timeout Delay	5	1		
PXE_BOOT_ITEM	71	4	Boot Server type(2), layer(2)	
PXE_CREDENTIAL_TYPES	12	4	Credential types (4)	
PXE_END	255	None		

The options fields in this message must include the following:

- The *siaddr* field must be null. (The Boot Server must include the MTFTP service.)
- *Server Identifier* (address of the responding Boot Server).
- The name of the credentials file in the *bootfile* field.

After receiving this message, the client downloads the credentials file and performs the authenticity test on the boot file as described previously. If the client is unable to obtain a credentials file, the authenticity test is deemed to have failed.

2.4.9 NBP Execution

If the authenticity test succeeds or is not required, then control is passed to the NBP. The way this is done is dependent on the client's CPU type.

2.4.9.1 NBP Execution for x86 PC/AT

For systems based on x86 PC/AT Intel Architecture, the NBP image is downloaded to 0:7C00h. The PXE ROM code executes a far call to location 0:7C00h.

2.4.10 MTFTP Operation

Implementation of MTFTP in the client is strongly recommended. If the server sends MTFTP parameters, and the client supports MTFTP, then the client must proceed as described in this section. In this case the client goes through four phases: listen, open, receive, and close, with an error recovery phase that can be entered at any point. Because there is no standard IETF implementation of MTFTP; this is a PXE proprietary implementation.

To enable multicast file transfers, the boot server provides the client with the following information. This information is provided in the boot server reply packet.:

- Client boot file name (Read from DHCP option 67 (bootfile name), if present. Otherwise, read from DHCP bootfile field.)
- Boot server IP address (Read from the DHCP option 54 (server identifier), if not found, use the siaddr field.)
- MTFTP Server UDP port number (Read from PXE option 3 (PXE_MTFTP_CPORT).)
- MTFTP Client UDP port number (Read from PXE option 2 (PXE_MTFTP_SPORT).)
- MTFTP multicast IP address (Read from PXE option 1 (PXE_MTFTP_IP).)
- MTFTP transmission listen delay (Read from PXE option 5 (PXE_MTFTP_DELAY).)
- MTFTP transmission time-out delay (Read from PXE option 4 (PXE_MTFTP_TMOUT).)

2.4.10.1 MTFTP listen

MTFTP sessions always begin with the client listening for a matching MTFTP session already in progress. If a matching MTFTP session is in progress, the client must collect all valid packets without acknowledging them. If the client collects all of the packets in the session, it is done and does not need to go on to the MTFTP open step.

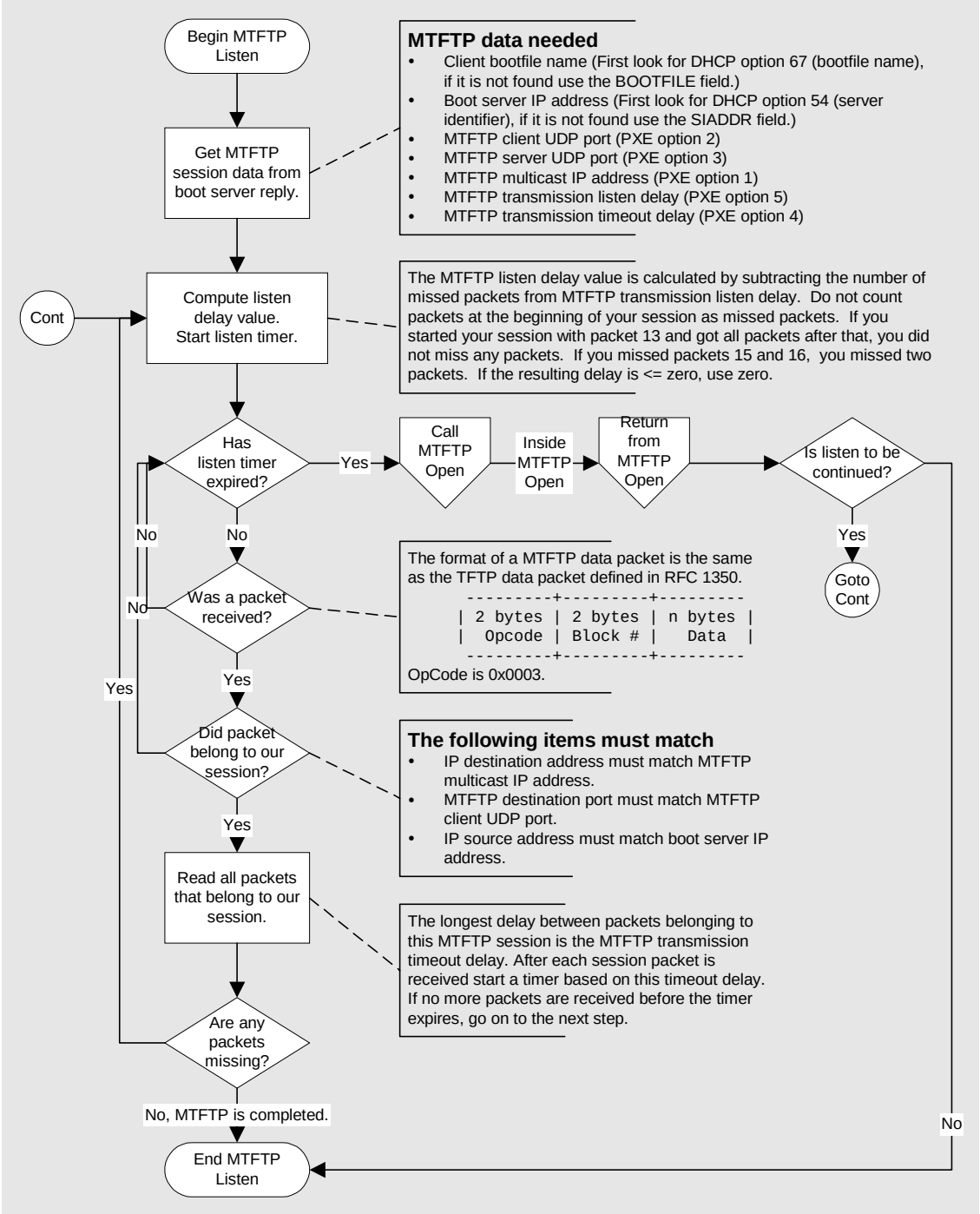

Figure 2-5 MTFTP Listen

2.4.10.2 MTFTP open

The MTFTP open state is transferred to if the client does not collect any packets during the listen phase AND the client receives a UNICAST MTFTP data packet from the MTFTP server. If the client receives a MULTICAST MTFTP data packet, the client transitions back to the MTFTP listen state.

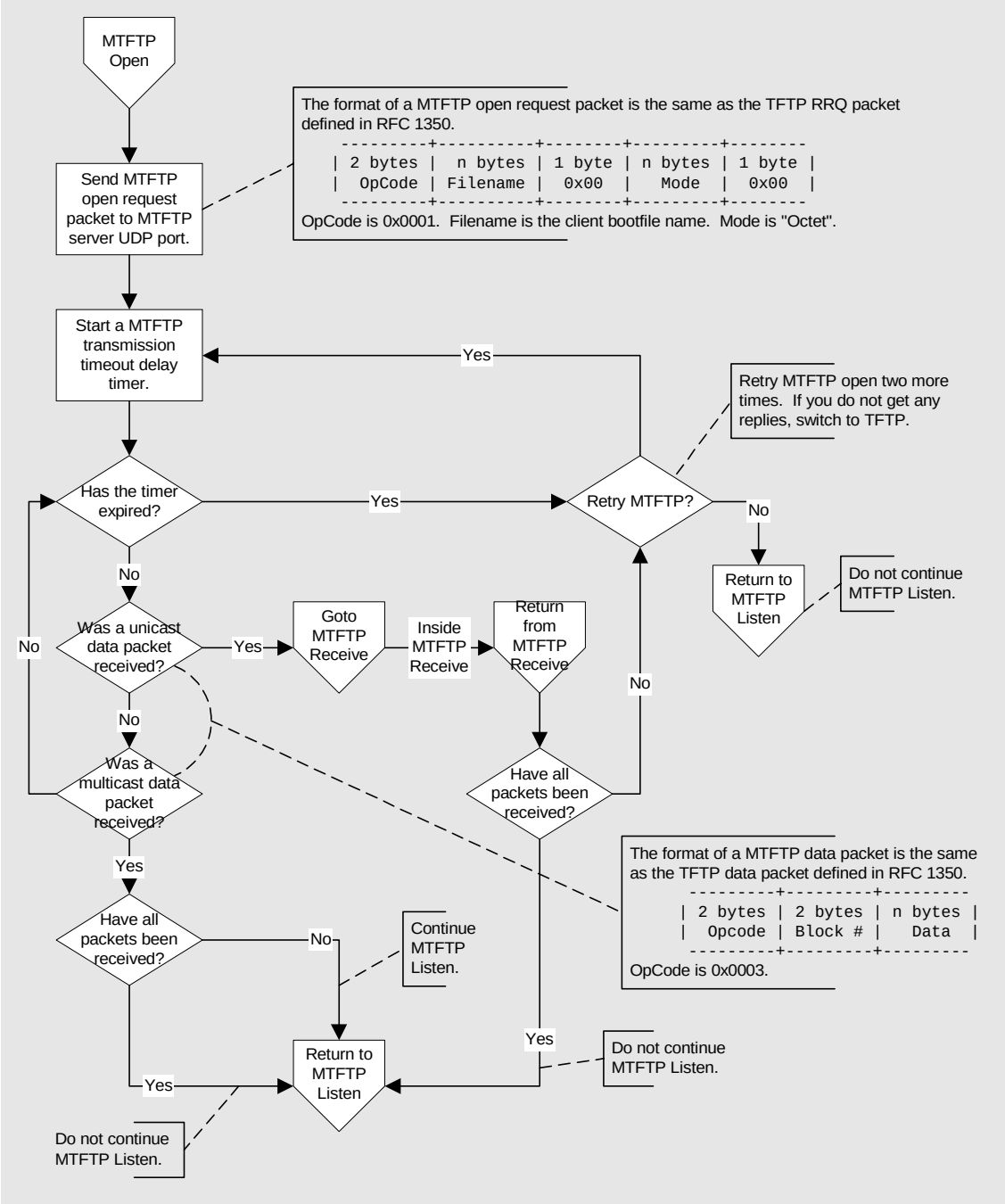

Figure 2-6 MTFTP Open

2.4.10.3 MTFTP receive

When the client enters the MTFTP receive state, the client must ACKnowledge all of the MTFTP data packets, including all packets that the client received during the MTFTP listen state. If an error occurs and the client must terminate the connection, the client must try to send an MTFTP error packet to the server (an MTFTP error packet uses the same format and error codes as the TFTP error packets defined in RFC 1350).

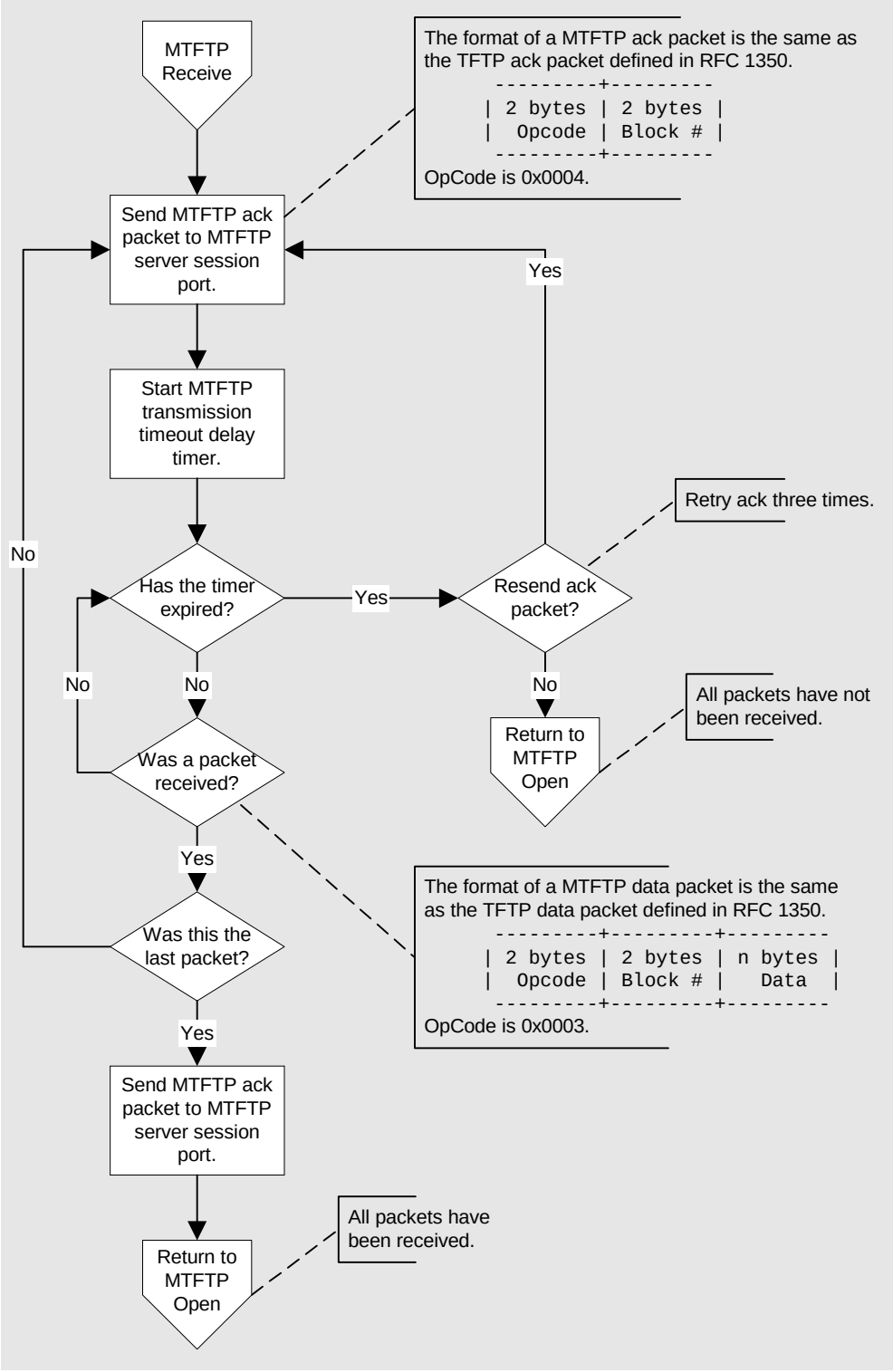

Figure 2-7 MTFTP Receive

2.4.10.4 MTFTP close

The MTFTP session ends when a listening client has received all of the packets it needs OR a receiving client ACKnowledges the last packet in the session. When either of these two cases occurs, the client drops out of the session, and no further action is required by the client.

2.5 Server Behavior

The server behavior needed for the extended protocol comprises two pieces of functionality: a Redirection service, and a Boot Service.

- The Redirection Service receives extended DHCPDISCOVER messages (generated by the client Initiation step) on the standard DHCP server port (67) and responds with DHCPOFFER messages containing a list of the Boot Server types and Boot Server discovery configuration tags.
- The Boot Service receives extended DHCPREQUEST and DHCPINFORM messages (generated by the client Boot Service Discovery step) and responds with DHCPACK messages containing the file name of an executable appropriate to the client.

A standard DHCP service may be extended to include the functionality of either the redirection service and/or the Boot Service. In this case, this extended DHCP service must implement all behaviors specified for the service included.

2.5.1 Redirection Service Behavior

This section summarizes the behavior of the redirection service to the DHCPDISCOVER message and other DHCP messages.

2.5.1.1 Response to DHCPDISCOVER

The redirection service must always be prepared to receive extended DHCPDISCOVER on UDP port (67) or an extended DHCPREQUEST on UDP port(4011). The format of these messages are described earlier in the DHCPDISCOVER section. The redirection service must only respond to messages that include DHCP Option #60 with the value of “PXEClient”.

If the redirection service responds to a message, it must respond by sending to the initiating client a DHCPOFFER message containing options as described earlier in the DHCPOFFER section.

If a Proxy DHCP server responds, the client IP address field of the message must be null (0.0.0.0).

If the redirection service is also a standard DHCP configuration service, then the DHCPOFFER message sent to the client must be as specified in RFC 2131.

2.5.2 Boot Service Behavior

This section summarizes the behavior of the Boot Service to the DHCPREQUEST message and TFTP service messages.

2.5.3 Response to DHCPREQUEST

The Boot Service must always be prepared to receive either a DHCPREQUEST or DHCPINFORM message with contents as described in the Boot Server Discovery section or as described in the NBP Authentication section. The Boot Service must distinguish between these cases on the basis of the MSB of the *layer* data field in the PXE_BOOT_ITEM option in the received message.

If the MSB of the *layer* data field is 0, then the Boot Service must respond by sending to the initiating client a DHCPACKNOWLEDGE message as described in the Boot Server Reply section. The file name in this message must be the complete path name of an executable appropriate to the client. The client must download the file from the Boot Server using MTFTP.

If the MSB of the *layer* data field is 1, then the Boot Service must respond by sending to the initiating client a DHCPACKNOWLEDGE message as described in the Boot Server Reply section. The file name in this message must be the complete path name of a file containing credentials for the boot file previously downloaded by the client. The format of the credentials file must be as described in [BIS]. The server must select the credentials file on the basis of

- The boot file name sent by the client in the *bootfile* field of the received message; and
- The appropriate credentials type. The Boot Service must determine the appropriate credentials type by examining the received message for a PXE_CREDENTIALS_TYPES option. If a PXE_CREDENTIALS_TYPES option is found, then Boot Service must examine the list of types in the data field of the option. In this case, the appropriate credentials type is the first type in the list that is among the types supported by the Boot Service. If no PXE_CREDENTIALS_TYPES option is found, then the appropriate credentials type is 1024-bit DSA / SHA-1, as described in [BIS].

The Boot Service must support at least the following credentials types as described in [BIS]:

- 1024-bit DSA / SHA-1; and
- 512-bit RSA / MD5.

The client must download the file credentials from the Boot Server using MTFTP

The Boot Service must be prepared to receive unicast, multicast, or broadcast messages. Multicast should be supported if a Multicast Boot Server Discovery address is provided.

The Boot Server may attempt to configure itself with a Multicast Boot Server Discovery address by executing a DHCPREQUEST or DHCPINFORM with an Option tag #60 set to “PXEServer”. The boot server should expect to receive an Option tag #60 “PXEServer” followed by tag #43 and subtag #7 (DISCOVERY_MCAST_ADDR) and subtag #11 (PXE_MCAST_ADDRS_ALLOC).

Table 2-8 DHCP/Proxy DHCPACK to Boot Service

DHCP Header				
Field (length)	**Value**	**Comment**		
op (1)	2	Code for BOOTP REPLY		
htype (1)	*			
hlen (1)	*			
hops (1)	*			
xid (4)	*			
secs (2)	*			
flags (2)	*			
ciaddr (4)	0.0.0.0	Server always sets this value to 0.0.0.0		
yiaddr (4)	a0, a1, a2, a3	Client's IP address. Provided by DHCP server		
siaddr (4)	a0, a1, a2, a3	Server IP address		
giaddr (4)	0.0.0.0			
chaddr (16)	*xx-xx-xx-xx-xx-xx-xx-xx	Client's MAC address		
sname (64)	*	Can be overloaded if using Opt 66		
bootfile (128)	*	Can be overloaded if using Opt 67		
99.130.83.99				
DHCP Options				
Tag Name	**Tag #**	**Length**	**Type**	**Data Field**
DHCP Message Type	53	1	5 = DHCPACK	
Client System Architecture	93	2		
Class Identifier	60	9	"PXEServer"	
DHCP_VENDOR	43	varies	Encapsulated options below	
PXE_PAD	0	None	None	
DISCOVERY_ MCAST_ADDR	7	4	**Multicast IP-addr** Boot Server discovery multicast IP address. Boot Servers capable of multicast discovery must listen on this multicast address.	
PXE_MCAST_ ADDRS_ALLOC	11	8	**McastIPbase(4), MIPblock(2), MIPrange(2)** McastIPBase is the starting address for multicast addresses. MIPblock = total size of multicast block. MIPrange = max number of multicast addresses available to any one Boot Server. MIPrange MAY equal MIPblock. Boot service must randomly pick an address within the range defined by McastIPBase through McastIPBase+MIPblock-MIPrange as the base number for their MIPrange number of IP addresses. The Boot service manages base addresses for each boot tree internally.	
PXE_END	255	None		

2.5.3.1 TFTP Service

The Boot Server must provide the TFTP service and should provide the MTFTP service, as described in the previous section. Redirection by the Boot Service to a TFTP service on a remote server should not be done as it is not reasonably possible for the redirecting server to know for certain that the TFTP server being redirected to is truly available.

3. PXE APIs

To enable the interoperability of clients and downloaded bootstrap programs, the client PXE code must provide a set of services for use by a downloaded bootstrap. (It also must ensure certain aspects of the client state at the point in time when the bootstrap begins executing, which is discussed in Section 4 PXE Initial Program Load (IPL).) The API services that must be provided by PXE for use by the bootstrap program are as follows:

- **Preboot API.** Contains several global control and information functions.
- **Trivial File Transport Protocol (TFTP) API.** Enables opening and closing of TFTP connections, and reading packets from and writing packets to a TFTP connection.
- **User Datagram Protocol (UDP) API.** Enables opening and closing UDP connections, and reading packets from and writing packets to a UDP connection.
- **Universal Network Driver Interface (UNDI) API.** Enables basic control of and I/O through the client's network interface device.

In a PXE Split ROM implementation, the Preboot, TFTP, and UDP APIs are provided in the BC (Base-Code) ROM. The UNDI API is provided in the UNDI ROM. (The BUSD ROM provides the BUSD API. These APIs are covered in Section 4.4.1.2 CardBus ROM Scan & Init, and section 4.4.2.2 Enable BUSD.

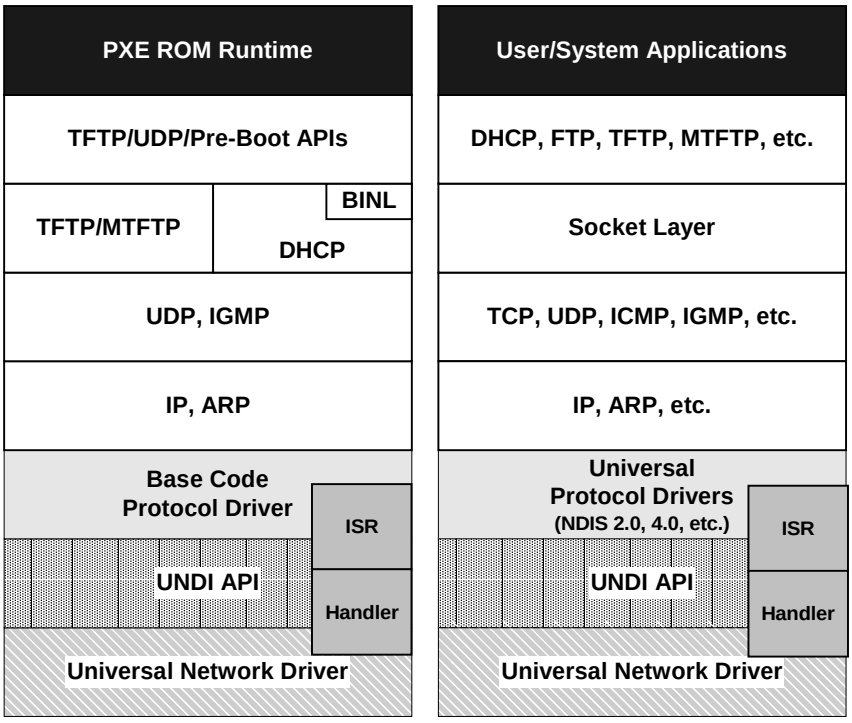

Figure 3-1 PXE Stack–Before and After Remote Boot

The PXE APIs are available to the bootstrap only if the Option #60 "PXEClient" is present in the DHCPOFFER message

Note: The descriptions in subsequent sections are specific to Intel-architecture PCs. A processor architecture-independent description of these interface and state specifications is probably possible, but has not been attempted.

3.1 PXE Installation Check

PXE installation check procedures are architecture-dependent. The methods described in this section are for PC/AT x86 clients.

In general, a PXE installation can be discovered using either of two methods. The first method (which can only be used in real mode) is to use the installation check interrupt, Int 1Ah. The second is to scan base memory for the !PXE or PXENV+ structure.

The primary users of these installation check methods are protocol drivers designed to use the PXE APIs. NBPs do not need to use these installation check procedures because the address of the !PXE structure is passed to them on the stack.

Note: For backward compatibility with existing applications, the address of the PXENV+ structure must also be passed in the ES:BX register pair. Since support for the PXENV+ structure is not planned for future versions, new applications (for example, universal protocol drivers) should be written to !PXE.

3.1.1 Real mode (Int 1Ah Function 5650h)

```
Enter:
       AX := 5650h (VP)
Exit:
       AX := 564Eh (VN)
       ES := 16-bit segment address of the PXENV+ structure.
       BX := 16-bit offset of the PXENV+.
       EDX := may be trashed by the UNDI INT 1Ah handler.
       All other register contents are preserved.
       CF is cleared.
       IF is preserved.
       All other flags are undefined.
```

3.1.2 PXENV+ Structure

This structure is supported for backward compatibility with NBPs. New NBPs must be written to use the !PXE structure. The !PXE structure can be found using the far pointer at the end of the PXENV+ structure.

The PXENV+ structure must be paragraph-aligned and reside in the UNDI code segment.

Table 3-1 PXENV+ Structure

Offset	*Type (bytes)*	*Name*	*Contents*
0x00	UINT8	Signature	"PXENV+"
0x06	UNIT16	Version	API version number. MSB=major LSB=minor. NBPs and OS drivers must check for this version number. If the API version number is 0x0201 or higher, use the !PXE structure. If the API version number is less than 0x0201, then use the PXENV+ structure.
0x08	UNIT8	Length	Length of this structure in bytes. This length must be used when computing the checksum of this structure.
0x09	UNIT8	Checksum	Used to make 8-bit checksum of this structure equal zero.
0x0A	SEGOFF16	RMEntry	Far pointer to real-mode PXE/UNDI API entry point. May be CS:0000h.
0x0E	UNIT32	PMOffset	32-bit offset to protected-mode PXE/UNDI API entry point. Do not use this entry point. For protected-mode API services, use the !PXE structure.
0x12	SEGSEL	PMSelector	Protected-mode selector of protected-mode PXE/UNDI API entry point. Do not use this entry point. For protected-mode API services, use the !PXE structure.
0x014	SEGSEL	StackSeg	Stack segment address. Must be set to 0 when removed from memory.
0x016	UNIT16	StackSize	Stack segment size in bytes.
0x018	SEGSEL	BC_CodeSeg	BC code segment address. Must be set to 0 when removed from memory.
0x1A	UNIT16	BC_CodeSize	BC code segment size. Must be set to 0 when removed from memory.
0x1C	SEGSEL	BC_DataSeg	BC data segment address. Must be set to 0 when removed from memory.
0x1E	UNIT16	BC_DataSize	BC data segment size. Must be set to 0 when removed from memory.
0x20	SEGSEL	UNDIDataSeg	UNDI data segment address. Must be set to 0 when removed from memory.
0x22	UNIT16	UNDIDataSize	UNDI data segment size. Must be set to 0 when removed from memory.
0x24	SEGSEL	UNDICodeSeg	UNDI code segment address. Must be set to 0 when removed from memory.
0x26	UNIT16	UNDICodeSize	UNDI code segment size. Must be set to 0 when removed from memory.
0x28	SEGOFF16	PXEPtr	Real mode segment offset pointer to !PXE structure. This field is only present if the API version number is 2.1 or greater.

3.1.3 Protected mode (Scanning base memory)

The !PXE structure must be placed on a paragraph boundary in the UNDI code segment. Scan base memory between the top of free base memory (FBM) and 0A0000h (640 Kbytes). If the top of FBM cannot be determined, start scanning paragraph boundaries from the top of base memory, 0A0000h (640K), down to 10000h (64K). Base-code runtime and UNDI drivers will almost always end up between 80000h (512K) and 0A0000h (640K).

3.1.4 !PXE Structure

The !PXE structure defines the location and size of the PXE and UNDI runtime segments. This structure must not be considered valid, and its contents must not be used, if the signature and checksum are not correct.

Table 3-2 !PXE Structure

Offset	*Type (bytes)*	*Name*	*Description*
0x00	UINT8(0x04)	Signature	'!PXE'
0x04	UINT8	StructLength	Length of this structure in bytes. This length must be used when computing the checksum of this structure.
0x05	UINT8	StructCksum	Used to make structure byte checksum equal zero.
0x06	UINT8	StructRev	Revision of this structure is zero. (0x00)
0x07	UINT8	reserved	Must be zero.
0x08	SEGOFF16	UNDIROMID	Real mode segment:offset of UNDI ROM ID structure. Check this structure if you need to know the UNDI API revision level. Filled in by UNDI loader module.
0x0C	SEGOFF16	BaseROMID	Real mode segment:offset of BC ROM ID structure. Must be set to zero if BC is removed from memory. Check this structure if you need to know the BC API revision level. Filled in by base-code loader module.
0x10	SEGOFF16	EntryPointSP	PXE API entry point for 16-bit stack segment. This API entry point is in the UNDI code segment and must not be CS:0000h. Filled in by UNDI loader module.
0x14	SEGOFF16	EntryPointESP	PXE API entry point for 32-bit stack segment. May be zero. This API entry point is in the UNDI code segment and must not be CS:0000h. Filled in by UNDI loader module.
0x18	SEGOFF16	StatusCallout	Far pointer to DHCP/TFTP status call-out procedure. If this field is –1, DHCP/TFTP will not make status calls. If this field is zero, DHCP/TFTP will use the internal status call-out procedure. StatusCallout defaults to zero. **Note:** The internal status call-out procedure uses BIOS I/O interrupts and will only work in real mode. This field must be updated before making any base-code API calls in protected mode.
0x1C	UINT8	reserved	Must be zero.
0x1D	UINT8	SegDescCnt	Number of segment descriptors needed in protected mode and defined in this table. UNDI requires four descriptors. UNDI plus BC requires seven.
0x1E	SEGSEL	FirstSelector	First protected mode selector assigned to PXE. Protected mode selectors assigned to PXE must be consecutive. Not used in real mode. Filled in by application before switching to protected mode.
0x20 0x28 0x30 0x38 0x40 0x48 0x50	SEGDESC(0x08) SEGDESC(0x08) SEGDESC(0x08) SEGDESC(0x08) SEGDESC(0x08) SEGDESC(0x08) SEGDESC(0x08)	Stack UNDIData UNDICode UNDICodeWrite BC_Data BC_Code BC_CodeWrite	The first two bytes of these segment descriptors contain the real mode segment or the protected mode selector. The next four bytes contain the physical address of the segment, and the last two bytes contain the size of the segment. Some implementations may need more selectors. The first seven are required to be implemented in this order. **Note:** These descriptors always contain the physical addresses of the segments and the protected mode driver must not overwrite them with the virtual addresses. Filled in by UNDI and base-code loader modules before any API calls are made.

3.2 PXE API Calling Convention

The !PXE APIs use the Microsoft 16-bit C/C++ __cdecl parameter format. This allows the PXE APIs to be used from most commercial 16-bit C/C++ compilers without using assembly language, or special compiler extensions.

The first two examples below show how a !PXE API is called from C and assembly. The third example shows how a PXENV+ API was called from assembly.

Both API styles are supported in this version of the specification for backwards compatibility. PXENV+ support will be dropped in a future version of this specification.

Network bootstrap programs (NBPs), UNDI OS drivers and application programs must use the !PXE APIs if they are available.

Example-1: !PXE API call from Microsoft 16-bit C/C++ v1.52c, using __cdecl parameter convention (16-bit stack segment):

```
(far __cdecl * PXE->EntryPointSP)(PXENV_TFTP_OPEN, &tftp_open_param);
```

Example-2: !PXE API call from Microsoft 16-bit Assembler v6.12, using __cdecl parameter convention (16- or 32-bit stack segments):

Note: When using a 32-bit stack segment do not push 32-bit words onto the stack. The PXE API services will not work, unless there are three 16-bit parameters pushed onto the stack.

```
push   DS                          ;Far pointer to parameter structure
push   offset tftp_open_param      ;is pushed onto stack.
push   PXENV_TFTP_OPEN             ;UINT16 is pushed onto stack.
call   dword ptr (s_PXE ptr es:[di]).EntryPointSP
add    sp, 6                       ;Caller cleans up stack.
```

Example-3: PXENV+ call from Microsoft 16-bit Assembler v6.12:

```
mov    word ptr PXENV_API, 0
mov    word ptr PXENV_API + 2, UNDI-CS
...
mov    bx, PXENV_TFTP_OPEN
les    di, tftp_open_param
call   dword ptr PXENV_API
```

Figure 3-2 shows an example Base-Code (BC) API call being made from an NBP into the UNDI API dispatch routine and then being passed into the BC API dispatch routine.

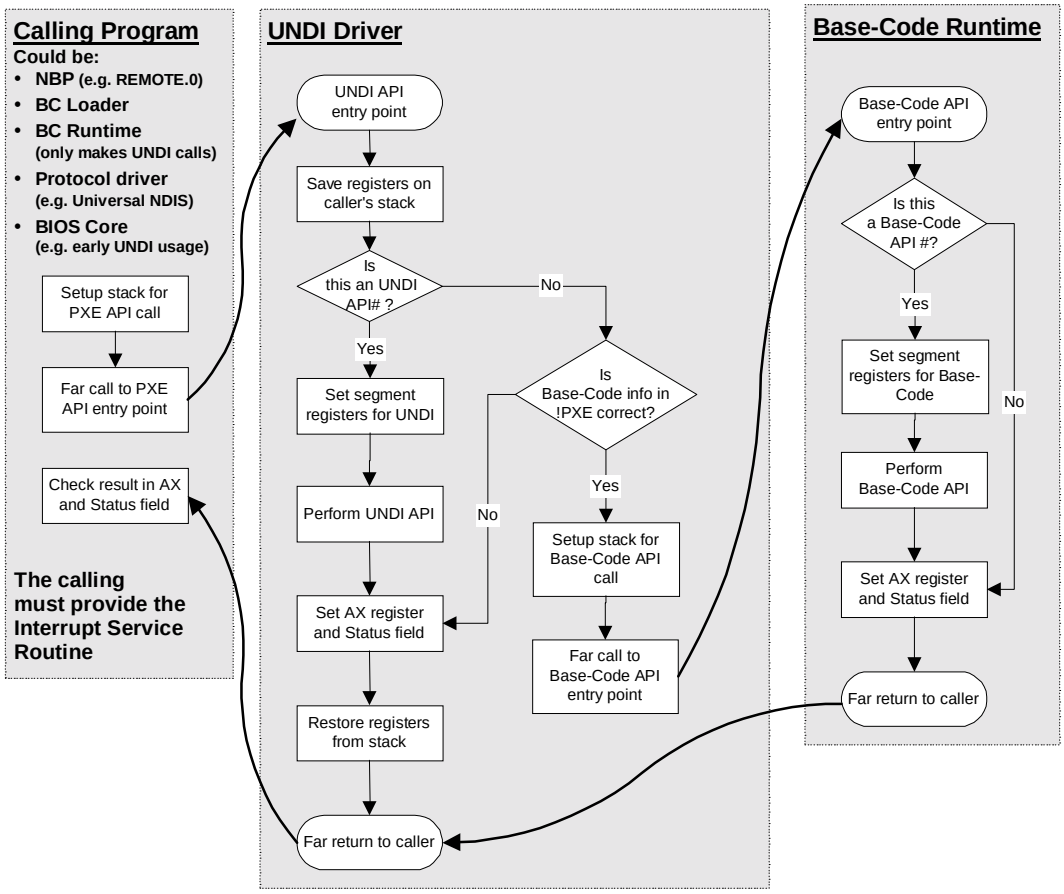

Figure 3-2 PXE API Calling Sequence

Example-4: API call being transferred from the UNDI driver to the BC runtime using Microsoft 16-bit C/C++ v1.52c, using __cdecl parameter convention:

```
PXEptr = MK_FP(sreg.cs, PXEoffset);
(far __cdecl * PXE->EntryPointSP)(PXENV_TFTP_OPEN, (void far
*)&tftp_open_param, PXEptr);
```

Example-5: API call being transferred from the UNDI driver to the BC runtime using Microsoft 16-bit Assembler v6.12, using __cdecl parameter convention:

```
push    CS                                  ;UNDI code segment
push    offset cs:PXE                       ;!PXE offset
push    DS                                  ;UNDI data segment
push    offset ds:tftp_open_param           ;Param offset
push    PXENV_TFTP_OPEN                     ;UINT16 is pushed onto stack.
call    dword ptr (s_PXE ptr es:[di])       ;EntryPointSP
add     sp, 10  ;Caller cleans up stack.
```

3.3 Early UNDI API Usage

The BIOS Core may make use of the UNDI option ROM to gain network access for the BIOS. In general, the BIOS is responsible for establishing a suitable environment for the UNDI API. The 'Early UNDI API Usage' flowchart below outlines the sequence the BIOS uses to load and use UNDI outside of the normal IPL sequence defined in Section 4.

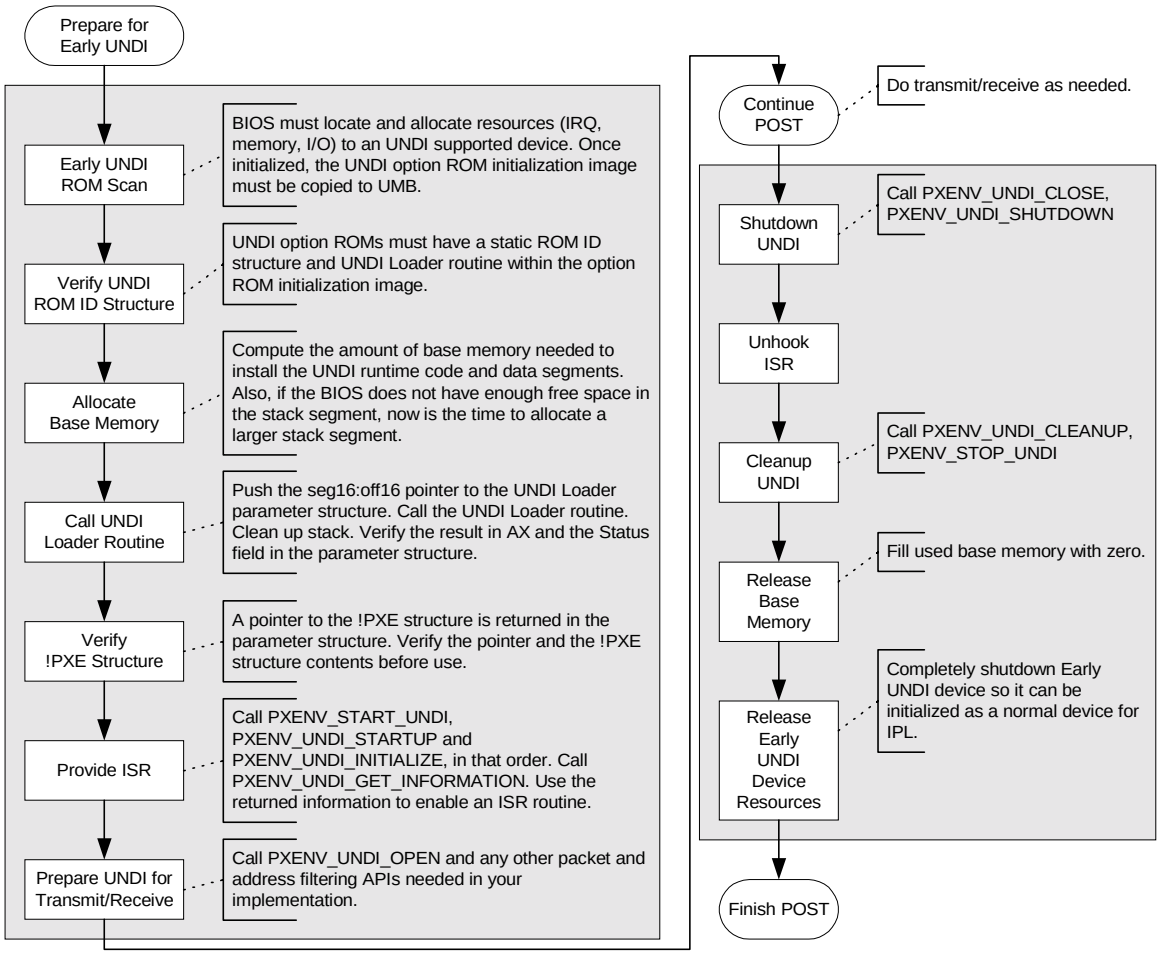

Figure 3-3 Early UNDI API Usage

3.4 PXE API Service Descriptions

3.4.1 Preboot API Service Descriptions

All the fields in the Preboot API Service parameter structures are to be stored in little endian (Intel) format unless otherwise specified.

UNLOAD BASE CODE STACK

Op-Code:	PXENV_UNLOAD_STACK (0070h)
Input:	Far pointer to a t_PXENV_UNLOAD_STACK parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This API prepares the base code for unloading by unhooking the IRQ and filling the base code and CPU stack segment entries in the PXENV+ and !PXE structures with zero. The original interrupt vector will be restored. This API does not change the amount of free base memory.
Note:	If PXENV_STATUS_UNDI_KEEP_ALL is returned, the !PXE and PXENV+ structures are NOT changed. The base code cannot be removed because the NIC interrupt vector has been changed. The base code interrupt service routine has been disabled and will no longer call the PXENV_UNDI_ISR API. If PXENV_STATUS_SUCCESS or PXENV_STATUS_FAILURE are returned, the !PXE and PXENV+ structures are changed. The CPU stack and base code segments can be re-used. Do not adjust the size of free base memory if PXENV_STATUS_FAILURE is returned. Increasing the size of free base memory: The size of free base memory is stored in a 16-bit word at 0x40:0x13. This word contains the size of free base memory in Kbytes. To increase the size of free base memory, increase the value stored in this memory location. This can only be done before an OS has been started. To remove only the base code and keep the UNDI, compare the UNDI code and data segment addresses. Keep the smaller of the two segment addresses. Shift this address right six bits. This is the new size of free base memory. To remove both the base code and UNDI, compare the UNDI code and data segment addresses. Keep the larger of the two segment addresses. Add to this the length of the kept segment in paragraphs (shift right four bits). Shift the sum right six bits, this is the new size of free base memory. Inserting the unused memory back into the memory chain: This can only be done after an OS has already started. There is no memory chain in base memory before an OS is started. How this is done is OS-dependent and is outside the scope of this document.

```
typedef struct s_PXENV_UNLOAD_STACK {
PXENV_STATUS Status;
UINT8 reserved[10];
} t_PXENV_UNLOAD_STACK;
```

Set before calling API service

N/A

Returned from API Service

Status:

Possible status codes:

PXENV_STATUS_SUCCESS: base-code is ready to be removed.

PXENV_STATUS_FAILURE: the size of free base memory has been changed.

PXENV_STATUS_UNDI_KEEP_ALL: the NIC interrupt vector has been changed.

The following flowchart shows how to unload to the base code.

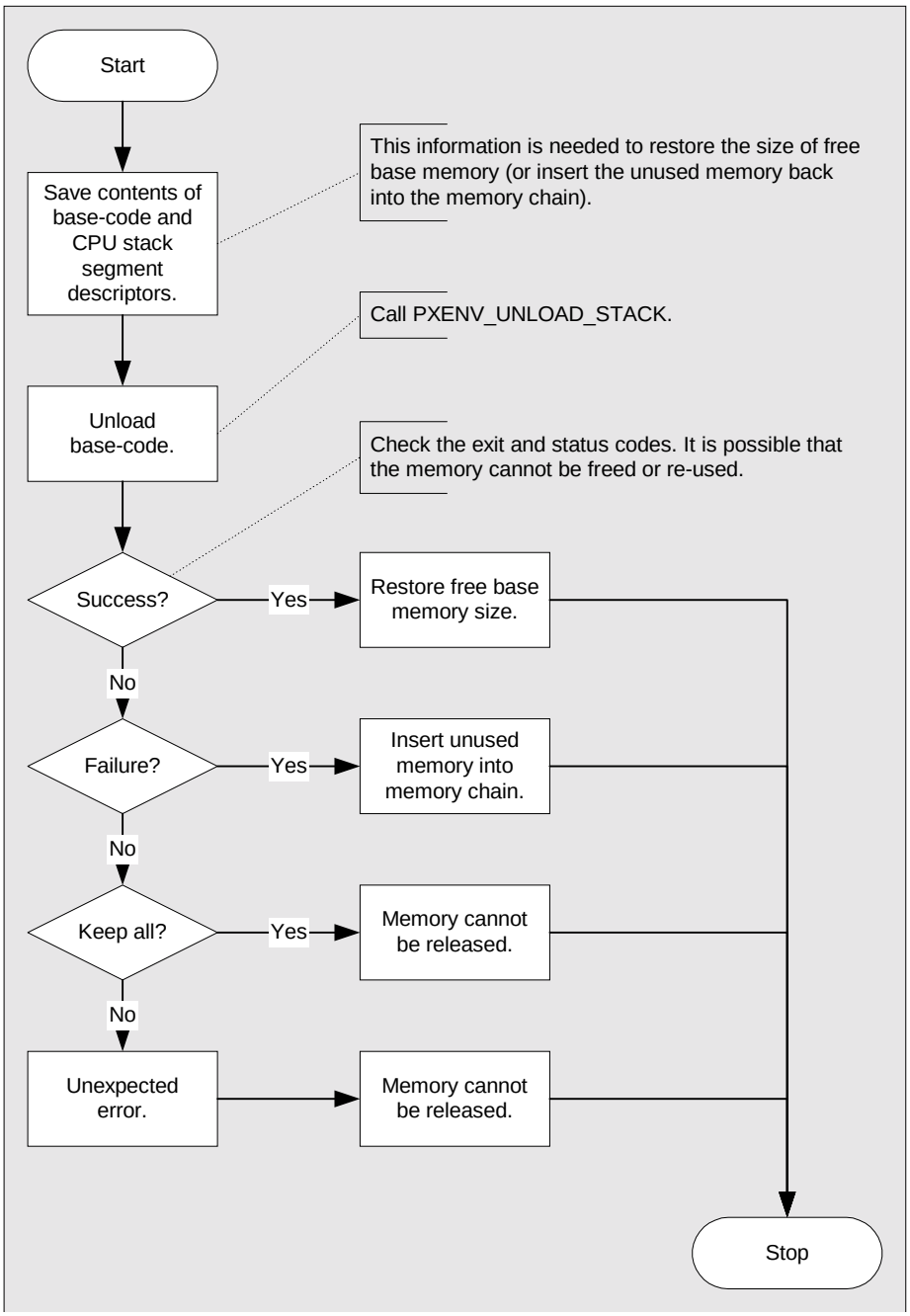

Figure 3-4 Unloading the base code

GET CACHED INFO

Op-Code:	PXENV_GET_CACHED_INFO (0071h)
Input:	Far pointer to a t_PXENV_GET_CACHED_INFO parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants. The buffer specified in the parameter structure must be filled with the requested information.

Description:	This service returns one of three buffers: 1. The client's DHCPDISCOVER packet 2. The DHCP server's DHCPACK packet 3. The Boot Server's Discover Reply packet, which contains Option #60 set to "PXEClient", a valid boot file name, and may contain MTFTP options In the downloaded NBP, the information that is returned by this service is used to configure client INI and CFG files. These files are then used to complete a valid network connection back to the configuration server. When an NBP does a Discover Request and gets a Discover Reply, it must replace the cached Discover Reply packet
Note:	This service cannot be used with a 32-bit stack segment.

```
typedef struct s_PXENV_GET_CACHED_INFO
{
PXENV_STATUS Status;
UINT16 PacketType;
#define PXENV_PACKET_TYPE_DHCP_DISCOVER  1
#define PXENV_PACKET_TYPE_DHCP_ACK       2
#define PXENV_PACKET_TYPE_CACHED_REPLY   3

UINT16 BufferSize;
SEGOFF16 Buffer;
UINT16 BufferLimit
} t_PXENV_GET_CACHED_INFO;
```

Set before calling API service

PacketType: Type of cached packet being requested.

BufferSize: Maximum number of bytes of data that can be copied into Buffer.

Buffer: Segment:Offset address of storage to be filled in by API service

Returned from API Service

Status: See the PXENV_STATUS_xxx constants.

BufferSize: Number of bytes of data that have been copied into Buffer. If BufferSize and Buffer were both set to zero, this field will contain the amount of data stored in Buffer in the BC data segment.

Buffer: If BufferSize and Buffer were both set to zero, this field will contain the segment:offset address of the Buffer in the BC data segment.

BufferLimit: Maximum size of the Buffer in the BC data segment.

```
typedef struct bootph
{
UINT8 opcode;
#define BOOTP_REQ       1
#define BOOTP_REP       2

UINT8 Hardware;
UINT8 Hardlen;
UINT8 Gatehops;
UINT32 ident;
UINT16 seconds;
UINT16 Flags;
#define BOOTP_BCAST     0x8000

IP4 cip;
IP4 yip;
IP4 sip;
IP4 gip;
MAC_ADDR CAddr;
UINT8 Sname[64];
UINT8 bootfile[128];
Union
  {
    UINT8 d[BOOTP_DHCPVEND];
    Struct
    {
      UINT8 magic[4];
#define VM_RFC1048      0x63825363L

      UINT32 flags;
      UINT8 pad[56];
    } v;
  } vendor;
} BOOTPLAYER;
```

Cached packet format

Opcode: Message opcode.

Hardware: Hardware type. See ARP section in "Assigned Numbers" RFC.

Hardlen: Hardware address length.

Gatehops: Client sets to zero. Optionally used by relay agent when booting via a relay agent.

Ident: Transaction ID. Random number chosen by client.

Seconds: Filled in by client. Seconds elapsed since client began address acquisition/renewal process.

Flags: BOOTP/DHCP broadcast flags.

CIP: Client IP address.

YIP: 'Your' IP address.

SIP: IP address of next server in boot process.

GIP: Relay agent IP address.

CAddr: Client hardware address.

SName: Optional server host name. Null terminated string.

Bootfile: Boot file name. Null terminated string.

Vendor: Vendor/DHCP parameter options.

Vendor.d: Array of bytes encompassing all of the Vendor/DHCP options.

Vendor.v.magic: DHCP magic cookie.

Vendor.v.flags: BOOTP flags/opcodes.

Vendor.v.pad: End of BOOTP vendor extensions

RESTART TFTP

Op-Code:	PXENV_RESTART_TFTP (0073h)
Input:	Far pointer to a t_PXENV_RESTART_TFTP parameter structure that has been initialized by the caller. The t_PXENV_RESTART_TFTP parameter structure is identical to the t_PXENV_TFTP_OPEN parameter structure.
Output:	If TFTP cannot be restarted, PXENV_EXIT_FAILURE must be returned and CF must be set. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants. If TFTP is restarted, control is never returned to the caller.
Description:	This service tries to establish a new TFTP connection with the server and to start a download of a new NBP. The NBP to be downloaded will be determined by the previously downloaded NBP. Once the NBP is downloaded into memory, control is passed to the NBP entry point. The download address and entry point for x86 PC/AT architecture is 0:7C00h. No other system memory is changed or initialized.
Note:	It is the responsibility of the caller to make sure the network connection is in a valid state before trying to restart TFTP. The existing network connection with the server needs to be maintained or restored. The existing TFTP connection needs to be closed. Service cannot be used in protected mode.

typedef struct s_PXENV_TFTP_READ_FILE { PXENV_STATUS Status; UINT8 FileName[128]; UINT32 BufferSize; ADDR32 Buffer; IP4 ServerIPAddress; IP4 GatewayIPAddress; IP4 McastIPAddress; UDP_PORT TFTPClntPort; UDP_PORT TFTPSrvPort; UINT16 TFTPOpenTimeOut; UINT16 TFTPReopenDelay; } t_PXENV_TFTP_READ_FILE;	**Set before calling API Service** **FileName**: Name of file to be downloaded. Null terminated. **BufferSize**: Size of the receive buffer in bytes. **Buffer**: Physical address of receive buffer. **ServerIPAddress**: IP address of TFTP server in network order. **GatewayIPAddress**: IP address of relay agent in network order. **McastIPAddress**: File multicast IP address in network order. **TFTPClntPort**: Client multicast listening port. **TFTPSrvPort**: Server multicast listening port. **TFTPOpenTimeOut**: Timeout value, in seconds, to be used for receiving data or ACK packets. If zero, default TFTP timeout is used. **TFTPReopenDelay**: Maximum time, in seconds, between ACK of last packet and new MTFTP open request. **Returned from API service** **Status**: See PXENV_STATUS_xxx constants. **BufferSize**: Number of bytes written into the receive buffer.

START UNDI

Op-Code:	PXENV_START_UNDI (0000h)
Input:	Far pointer to a t_PXENV_START_UNDI parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This service is used to pass the BIOS parameter registers to the UNDI driver. The UNDI driver is responsible for saving the information it needs to communicate with the hardware. This service is also responsible for hooking the Int 1Ah service routine
Note:	This API service must be called only once during UNDI Option ROM boot. The UNDI driver is responsible for saving this information and using it every time PXENV_UNDI_STARTUP is called. Service cannot be used in protected mode.

typedef struct s_PXENV_START_UNDI { PXENV_STATUS Status; UINT16 AX; UINT16 BX; UINT16 DX; UINT16 DI; UINT16 ES; } t_PXENV_START_UNDI;	**Set before calling API service** **AX, BX, DX, DI, ES:** BIOS initialization parameter registers. These fields should contain the same information passed to the option ROM initialization routine by the Host System BIOS. Information about the contents of these registers can be found in the [PnP], [PCI] and [BBS] specifications. **Returned from API service** **Status**: See the PXENV_STATUS_xxx constants.

STOP UNDI

Op-Code:	PXENV_STOP_UNDI (0015h)
Input:	Far pointer to a t_PXENV_STOP_UNDI parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This routine is responsible for unhooking the Int 1Ah service routine.
Note:	This API service must be called only once at the end of UNDI Option ROM boot. One of the valid status codes is PXENV_STATUS_KEEP. If this status is returned, UNDI must not be removed from base memory. Also, UNDI must not be removed from base memory if BC is not removed from base memory. Service cannot be used in protected mode.

`typedef struct s_PXENV_STOP_UNDI`
`{`
`PXENV_STATUS Status;`
`} t_PXENV_STOP_UNDI;`
	Set before calling API service
	N/A
	Returned from API service
	Status: See the PXENV_STATUS_xxx constants.

START BASE

Op-Code:	PXENV_START_BASE (0075h)
Input:	Far pointer to a t_PXENV_START_BASE parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This API service is used to start the BC. The BC will make calls to UNDI as needed to implement an IP stack and start the DHCP client software. Service cannot be used in protected mode.
Note:	The UNDI driver must be started first. Refer to the START UNDI API call.

`typedef struct s_PXENV_START_BASE`
`{`
`PXENV_STATUS Status;`
`} t_PXENV_START_BASE;`
	Set before calling API service
	N/A
	Returned from API service
	Status: See the PXENV_STATUS_xxx constants.

STOP BASE

Op-Code:	PXENV_STOP_BASE (0076h)
Input:	Far pointer to a t_PXENV_STOP_BASE parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This API service is used to stop the BC. The BC will make calls to UNDI as needed.
Note:	The BC must be started first. Refer to the START_BASE API call. One of the valid status codes is PXENV_STATUS_KEEP. If this status is returned, BC must not be removed from base memory. Service cannot be used in protected mode.

`typedef struct s_PXENV_STOP_BASE`
`{`
`PXENV_STATUS Status;`
`} t_PXENV_STOP_BASE;`
	Set before calling API service
	N/A
	Returned from API service
	Status: See the PXENV_STATUS_xxx constants.

3.4.2 TFTP API Service Descriptions

All the fields in the TFTP API parameter structures are to be stored in little endian (Intel) format unless otherwise specified.

TFTP OPEN

Op-Code:	PXENV_TFTP_OPEN (0020h)
Input:	Far pointer to a t_PXENV_TFTP_OPEN parameter structure that has been initialized by the caller. The IP addresses and port numbers in this structure are to be stored in big endian (Motorola) format.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	Opens a TFTP connection for reading/writing. At any one time there can be only one open connection. The connection must be closed before another can be opened.

Note:	Service cannot be used if an MTFTP or UDP connection is active. Service cannot be used in protected mode if the StatusCallout field in the !PXE structure is set to zero. Service cannot be used with a 32-bit stack segment.

```
typedef struct s_PXENV_TFTP_OPEN
{
PXENV_STATUS Status;
IP4 ServerIPAddress;
IP4 GatewayIPAddress;
UINT8 FileName[128];
UDP_PORT TFTPPort;
UINT16 PacketSize;
} t_PXENV_TFTP_OPEN;
```

Set before calling API service

ServerIPAddress: TFTP server IP address in network order.

GatewayIPAddress: Relay agent IP address in network order. If this address is set to zero, the IP layer will resolve this using its own routing table. The IP layer should provide space for a minimum of four routing entries obtained from default router and static route DHCP option tags in the DHCPackr message, plus any non-zero GIADDR field from the DHCPOffer message(s) accepted by the client.

Filename: Name of file to be downloaded. Null terminated.

TFTPPort: UDP port TFTP server is listening to requests on.

PacketSize: Requested size of TFTP packet, in bytes; with a minimum of 512 bytes.

Returned from API service

PacketSize: Negotiated size of TFTP packet, in bytes; less than or equal to the requested size

Status: See PXENV_STATUS_xxx constants.

TFTP CLOSE

Op-Code:	PXENV_TFTP_CLOSE (0021h)
Input:	Far pointer to a t_PXENV_TFTP_CLOSE parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	Closes the previously opened TFTP connection.
Note:	Service cannot be used if there is not an active MTFTP connection. Service cannot be used in protected mode if the StatusCallout field in the !PXE structure is set to zero. Service cannot be used with a 32-bit stack segment.

```
typedef struct s_PXENV_TFTP_CLOSE
{
PXENV_STATUS Status;
} t_PXENV_TFTP_CLOSE;
```

Set before calling API service

N/A

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

TFTP READ

Op-Code:	PXENV_TFTP_READ (0022h)
Input:	Far pointer to a t_PXENV_TFTP_READ parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants. When a read is successful, the PacketNumber and Packet Lengths must also be filled in.
Description:	Reads one packet from the open TFTP connection.
Note:	Service cannot be used if there is not an active MTFTP connection. Service cannot be used in protected mode if the StatusCallout field in the !PXE structure is set to zero. Service cannot be used with a 32-bit stack segment.

`typedef struct s_PXENV_TFTP_READ`
`{`
`PXENV_STATUS Status;`
`UINT16 PacketNumber;`
`UINT16 BufferSize;`
`SEGOFF16 Buffer;`
`} t_PXENV_TFTP_READ;`
	Set before calling API service
	Buffer: Segment:Offset address of packet buffer.
	Returned from API service
	Status: See the PXENV_STATUS_xxx constants.
	PacketNumber: Packet number (1-65535) sent from the TFTP server.
	BufferSize: Number of bytes written to the packet buffer. Last packet if this is less thanthe size negotiated in TFTP_OPEN. Zero is valid.

TFTP/MTFTP READ FILE

Op-Code:	PXENV_TFTP_READ_FILE (0023h)
Input:	Far pointer to a t_PXENV_TFTP_READ_FILE parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants. When a read is successful, the PacketCount and Packet Lengths must also be filled in.
Description:	This service will open a TFTP, or MTFTP, connection, download the entire file and close the connection. It is up to the caller to make sure that there is enough free memory to download the file into. For example, you cannot download a 2 MB file into base memory (below 640K).
Note:	UDP open must be called before UDP read or write can be used after transferring a file with this service. Service cannot be used if a MTFTP or UDP connection is active. Service cannot be used in protected mode if the StatusCallout field in the !PXE structure is set to zero. Service cannot be used with a 32-bit stack segment.

`typedef struct s_PXENV_TFTP_READ_FILE`
`{`
`PXENV_STATUS Status;`
`UINT8 FileName[128];`
`UINT32 BufferSize;`
`ADDR32 Buffer;`
`IP4 ServerIPAddress;`
`IP4 GatewayIPAddress;`
`IP4 McastIPAddress;`
`UDP_PORT TFTPClntPort;`
`UDP_PORT TFTPSrvPort;`
`UINT16 TFTPOpenTimeOut;`
`UINT16 TFTPReopenDelay;`
`} t_PXENV_TFTP_READ_FILE;`
	Set before calling API service
	FileName: Name of file to be downloaded. Null terminated.
	BufferSize: Size of the receive buffer in bytes.
	Buffer: Physical address of receive buffer.
	ServerIPAddress: IP address of TFTP server in network order.
	GatewayIPAddress: IP address of relay agent in network order. If this address is set to zero, the IP layer will resolve this using its own routing table.
	McastIPAddress: File multicast IP address in network order.
	TFTPClntPort: Client multicast listening port.
	TFTPSrvPort: Server multicast listening port.
	TFTPOpenTimeOut: Timeout value, in seconds, to be used for receiving data or ACK packets. If zero, default TFTP timeout is used.
	TFTPReopenDelay: Maximum time, in seconds, between ACK of last packet and new MTFTP open request.
	Returned from API service
	Status: See PXENV_STATUS_xxx constants.
	BufferSize: Number of bytes written into the receive buffer.

TFTP_GET_FILE_SIZE

Op-Code:	PXENV_TFTP_GET_FSIZE (0025h)
Input:	Far pointer to a t_PXENV_TFTP_GET_FSIZE parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants. When the call is successful, the FileSize field must be filled in.

Description	This service will query the server for the size of the given file using TFTP option extension protocol. This service will not and hence must not be used to open a TFTP connection for the given file.
Note:	Service cannot be used if a MTFTP or UDP connection is active. Service cannot be used in protected mode if the StatusCallout field in the !PXE structure is set to zero. Service cannot be used with a 32-bit stack segment.

```
typedef struct s_PXENV_TFTP_GET_FSIZE
{
PXENV_STATUS Status;
IP4 ServerIPAddress;
IP4 GatewayIPAddress;
UINT8 FileName[128];
UINT32 FileSize;
} t_PXENV_TFTP_GET_FSIZE;
```

Set before calling API service

ServerIPAddress: IP address of TFTP server.

GatewayIPAddress: IP address of relay agent. If this address is set to zero, the IP layer will resolve this using its own routing table.

Filename: Name of file on TFTP server. Null terminated.

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

Filesize: Size of the file in bytes.

3.4.3 UDP API Service Descriptions

UDP OPEN

Op-Code:	PXENV_UDP_OPEN (0030h)
Input:	Far pointer to a t_PXENV_UDP_OPEN parameter.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	Opens a UDP connection for reading and writing. There can only be one open connection at a time.
Note:	Service cannot be used if a MTFTP or UDP connection is active. Service cannot be used in protected mode if the StatusCallout field in the !PXE structure is set to zero. Service cannot be used with a 32-bit stack segment.

```
typedef struct s_PXENV_UDP_OPEN
{
PXENV_STATUS status;
IP4 src_ip;
} t_PXENV_UDP_OPEN;
```

Set before calling API service

SrcIP: IP address of this station.

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

UDP CLOSE

Op-Code:	PXENV_UDP_CLOSE (0031h)
Input:	Far pointer to a t_PXENV_UDP_CLOSE parameter.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	Closes the previously opened UDP connection.
Note:	Service cannot be used if a MTFTP is active, or if there is no active UDP connection. Service cannot be used in protected mode if the StatusCallout field in the !PXE structure is set to zero. Service cannot be used with a 32-bit stack segment.

```
typedef struct s_PXENV_UDP_CLOSE
{
PXENV_STATUS status;
} t_PXENV_UDP_CLOSE;
```

Set before calling API service

N/A

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

UDP WRITE

Op-Code:	PXENV_UDP_WRITE (0033h)
Input:	Far pointer to a t_PXENV_UDP_WRITE parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	Writes one packet to the specified IP address on the open UDP connection.
Note:	Service cannot be used if a MTFTP is active, or if there is no active UDP connection. Service cannot be used in protected mode if the StatusCallout field in the !PXE structure is set to zero. Service cannot be used with a 32-bit stack segment.

```
typedef struct s_PXENV_UDP_WRITE
{
PXENV_STATUS status;
IP4 ip;
IP4 gw;
UDP_PORT src_port;
UDP_PORT dst port;
UINT16 buffer_size;
SEGOFF16 buffer;
} t_PXENV_UDP_WRITE;
```

Set before calling API service

IP: Destination IP address.

GW: IP address of relay agent. If this address is set to zero, the IP layer will resolve this using its own routing table.

SrcPort: Source UDP port. Assigned 2069 if set to zero.

DstPort: Destination UDP port.

BufferSize: Length of the packet in bytes.

Buffer: Segment:Offset of the packet buffer.

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

UDP READ

Op-Code:	PXENV_UDP_READ (0032h)
Input:	Far pointer to a t_PXENV_UDP_READ parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants. This API function is non-blocking and returns the same values as PXENV_UNDI_TRANSMIT_PACKET. PXENV_EXIT_SUCCESS/PXENV_STATUS_SUCCESS is returned if a packet has been transferred into the caller's buffer. PXENV_EXIT_FAILURE/PXENV_STATUS_FAILURE is returned if no packet is available to transfer.
Description:	Reads one packet from the opened UDP connection.
Note:	Service cannot be used if a MTFTP is active, or if there is no active UDP connection. Service cannot be used in protected mode if the StatusCallout field in the !PXE structure is set to zero. Service cannot be used with a 32-bit stack segment.

`typedef struct s_PXENV_UDP_READ`
`{`
`PXENV_STATUS status;`
`IP4 src_ip;`
`IP4 dest_ip;`
`UDP_PORT s_port;`
`UDP_PORT d_port;`
`UINT16 buffer_size;`
`SEGOFF16 buffer;`
`} t_PXENV_UDP_READ;`
	Set before calling API service
	DestIP: Only accept packets sent to this IP address. If this is zero, packets sent to any IP address are accepted.
	DestPort: Only accept packets sent to this UDP port. If this is zero, all packets sent to this station are accepted.
	BufferSize: Size of the packet buffer in bytes.
	Buffer: Segment:Offset of the packet buffer.
	Returned from API service
	Status: See the PXENV_STATUS_xxx constants.
	SrcIP: IP address of the sender.
	SrcPort: UDP source port of the sender.
	BufferSize: Number of bytes written into the packet buffer.

3.4.4 UNDI API Service Descriptions

UNDI STARTUP

Op-Code:	PXENV_UNDI_STARTUP (0001h)
Input:	Far pointer to a t_PXENV_UNDI_STARTUP parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This API is responsible for initializing the contents of the UNDI code & data segment for proper operation. Information from the !PXE structure and the first PXENV_START_UNDI API call is used to complete this initialization. The rest of the UNDI APIs will not be available until this call has been completed.
Note:	PXENV_UNDI_STARTUP must not be called again without first calling PXENV_UNDI_SHUTDOWN. PXENV_UNDI_STARTUP and PXENV_UNDI_SHUTDOWN are no longer responsible for chaining interrupt 1Ah. This must be done by the PXENV_START_UNDI and PXENV_STOP_UNDI API calls. This service cannot be used in protected mode.

`typedef struct s_PXENV_UNDI_STARTUP`
`{`
`PXENV_STATUS Status;`
`} t_PXENV_UNDI_STARTUP;`
	Set before calling API service
	N/A
	Returned from API service
	Status: See the PXENV_STATUS_xxx constants.

UNDI CLEANUP

Op-Code:	PXENV_UNDI_CLEANUP (0002h)
Input:	Far pointer to a t_PXENV_UNDI_CLEANUP parameter structure.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This call will prepare the network adapter driver to be unloaded from memory. This call must be made just before unloading the Universal NIC Driver. The rest of the API will not be available after this call executes. This service cannot be used in protected mode.

`typedef struct s_PXENV_UNDI_CLEANUP {`
`PXENX_STATUS Status;`
`} t_PXENV_UNDI_CLEANUP;`
	Set before calling API service
	N/A
	Returned from API service
	Status: See the PXENV_STATUS_xxx constants.

UNDI INITIALIZE

Op-Code:	PXENV_UNDI_INITIALIZE (0003h)
Input:	Far pointer to a t_PXENV_UNDI_INITIALIZE parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This call resets the adapter and programs it with default parameters. The default parameters used are those supplied to the most recent UNDI_STARTUP call. This routine does not enable the receive and transmit units of the network adapter to readily receive or transmit packets. The application must call PXENV_UNDI_OPEN to logically connect the network adapter to the network. This call must be made by an application to establish an interface to the network adapter driver.
Note:	When the PXE code makes this call to initialize the network adapter, it passes a NULL pointer for the Protocol field in the parameter structure.

```
typedef struct
s_PXENV_UNDI_INITIALIZE {
PXENV_STATUS Status;
ADDR32 ProtocolIni;
UINT8 reserved[8];
} t_PXENV_UNDI_INITIALIZE;
```

Set before calling API service

ProtocolIni: Physical address of a memory copy of the driver module from the 'protocol.ini' file obtained from the protocol manager driver (refer to the NDIS 2.0 specification). This parameter is supported for the universal NDIS driver to pass the information contained in the 'protocol.ini' file to the NIC driver for any specific configuration of the NIC. (Note that the module identification in the 'protocol.ini' file was done by NDIS.) This value can be NULL for any other application interfacing to the universal NIC driver

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

UNDI RESET ADAPTER

Op-Code:	PXENV_UNDI_RESET_ADAPTER (0004h)
Input:	Far pointer to a t_PXENV_UNDI_RESET_ADAPTER parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This call resets and reinitializes the network adapter with the same set of parameters supplied to Initialize Routine. Unlike Initialize, this call opens the adapter that is, it connects logically to the network. This routine cannot be used to replace Initialize or Shutdown calls.

```
typedef struct s_PXENV_UNDI_RESET
{
PXENV_STATUS Status;
t_PXENV_UNDI_MCAST_ADDRESS
R_Mcast_Buf;
} t_PXENV_UNDI_RESET;

#define MAXNUM_MCADDR       8

typedef struct
s_PXENV_UNDI_MCAST_ADDRESS
{
UINT16 MCastAddrCount;
MAC_ADDR McastAddr[MAXNUM_MCADDR];
} t_PXENV_UNDI_MCAST_ADDRESS;
```

Set before calling API service

R_Mcast_Buf: This is a structure of MCastAddrCount and McastAddr.

MCastAddrCount: Number of multicast MAC addresses in the buffer.

McastAddr: List of up to MAXNUM_MCADDR multicast MAC addresses.

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

UNDI SHUTDOWN

Op-Code:	PXENV_UNDI_SHUTDOWN (0005h)

Input:	Far pointer to a t_PXENV_UNDI_SHUTDOWN parameter.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This call resets the network adapter and leaves it in a safe state for another driver to program it.
Note:	The contents of the PXENV_UNDI_STARTUP parameter structure need to be saved by the Universal NIC Driver in case PXENV_UNDI_INITIALIZE is called again.

```
typedef struct s_PXENV_UNDI_SHUTDOWN
{
PXENV_STATUS Status;
} t_PXENV_UNDI_SHUTDOWN;
```

Set before calling API service

N/A

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

UNDI OPEN

Op-Code:	PXENV_UNDI_OPEN (0006h)
Input:	Far pointer to a t_PXENV_UNDI_OPEN parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This call activates the adapter's network connection and sets the adapter ready to accept packets for transmit and receive.

```
typedef struct s_PXENV_UNDI_OPEN
{
PXENV_STATUS Status;
UINT16 OpenFlag;
UINT16 PktFilter;
#define FLTR_DIRECTED  0x0001
#define FLTR_BRDCST    0x0002
#define FLTR_PRMSCS    0x0004
#define FLTR_SRC_RTG   0x0008

t_PXENV_UNDI_MCAST_ADDRESS
R_Mcast_Buf;
} t_PXENV_UNDI_OPEN;
```

Set before calling API service

OpenFlag: This is an adapter specific input parameter. This is supported for the universal NDIS 2.0 driver to pass in the open flags provided by the protocol driver. (See the NDIS 2.0 specification.) This can be zero.

PktFilter: Filter for receiving packets. This can be one, or more, of the FLTR_xxx constants. Multiple values are arithmetically or-ed together.

"Directed" packets are packets that may come to your MAC address or the multicast MAC address.

R_Mcast_Buf: See definition in UNDI RESET ADAPTER (0004h).

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

UNDI CLOSE

Op-Code:	PXENV_UNDI_CLOSE (0007h)
Input:	Far pointer to a t_PXENV_UNDI_CLOSE parameter.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This call disconnects the network adapter from the network. Packets cannot be transmitted or received until the network adapter is open again.

```
typedef struct s_PXENV_UNDI_CLOSE
{
PXENV_STATUS Status;
} t_PXENV_UNDI_CLOSE;
```

Set before calling API service

N/A

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

UNDI TRANSMIT PACKET

Op-Code:	PXENV_UNDI_TRANSMIT (0008h)
Input:	Far pointer to a t_PXENV_UNDI_TRANSMIT parameter structure that has been initialized by the caller.

Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status code must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This call transmits a buffer to the network. The media header for the packet can be filled by the calling protocol, but it might not be. The network adapter driver will fill it if required by the values in the parameter block. The packet is buffered for transmission provided there is an available buffer, and the function returns PXENV_EXIT_SUCCESS. If no buffer is available the function returns PXENV_EXIT_FAILURE with a status code of PXE_UNDI_STATUS__OUT OF_RESOURCE. The number of buffers is implementation-dependent. An interrupt is generated on completion of the transmission of one or more packets. A call to PXENV_UNDI_TRANSMIT is permitted in the context of a transmit complete interrupt.

```
typedef struct s_PXENV_UNDI_TRANSMIT
{
PXENV_STATUS Status;
UINT8 Protocol;
#define P_UNKNOWN  0
#define P_IP       1
#define P_ARP      2
#define P_RARP     3

UINT8 XmitFlag;
#define XMT_DESTADDR   0x0000
#define XMT_BROADCAST  0x0001

SEGOFF16 DestAddr;
SEGOFF16 TBD;
UINT32 Reserved[2];
} t_PXENV_UNDI_TRANSMIT;

#define MAX_DATA_BLKS  8

typedef struct s_PXENV_UNDI_TBD
{
UINT16 ImmedLength;
SEGOFF16 Xmit;
UINT16 DataBlkCount;
  struct DataBlk
  {
    UINT8 TDPtrType;
    UINT8 TDRsvdByte;
    UINT16 TDDataLen;
    SEGOFF16 TDDataPtr;
  } DataBlock[MAX_DATA_BLKS];
 t_PXENV_UNDI_TBD;
```

Set before calling API service

Protocol: This is the protocol of the upper layer that is calling UNDI TRANSMIT call. If the upper layer has filled the media header, this field must be P_UNKNOWN.

XmitFlag: If this flag is XMT_DESTADDR, the NIC driver expects a pointer to the destination media address in the field DestAddr. If XMT_BROADCAST, the NIC driver fills the broadcast address for the destination.

TBD: Segment:Offset address of the transmit buffer descriptor.

ImmedLength: Length of the immediate transmit buffer: Xmit.

Xmit: Segment:Offset of the immediate transmit buffer.

DataBlkCount: Number of blocks in this transmit buffer.

TDPtrType:

0 => 32-bit physical address in TDDataPtr (not supported in this version of PXE)

1 => segment:offset in TDDataPtr which can be a real mode or 16-bit protected mode pointer

TDRsvdByte: Reserved must be zero.

TDDatalen: Data block length in bytes.

TDDataPtr: Segment:Offset of the transmit block.

DataBlock: Array of transmit data blocks.

Returned from API service

Status: See the PXENV_STATUS_xxx constants

UNDI SET MULTICAST ADDRESS

Op-Code:	PXENV_UNDI_SET_MCAST_ADDRESS (0009h)
Input:	Far pointer to a t_PXENV_TFTP_SET_MCAST_ADDRESS parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This call changes the current list of multicast addresses to the input list and resets the network adapter to accept it. If the number of multicast addresses is zero, multicast is disabled.

typedef struct s_PXENV_UNDI_SET_MCAST_ADDRESS { PXENV_STATUS Status; t_PXENV_UNDI_MCAST_ADDRESS R_Mcast_Buf; } t_PXENV_UNDI_SET_MCAST_ADDR;	**Set before calling API service** **R_Mcast_Buf:** See description in the UNDI RESET ADAPTER (0004h) API. **Returned from API service** **Status**: See the PXENV_STATUS_xxx constants

UNDI SET STATION ADDRESS

Op-Code:	PXENV_UNDI_SET_STATION_ADDRESS (000Ah)
Input:	Far pointer to a t_PXENV_UNDI_SET_STATION_ADDRESS parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This call sets the MAC address to be the input value and is called before opening the network adapter. Later, the open call uses this variable as a temporary MAC address to program the adapter's individual address registers.

typedef struct s_PXENV_UNDI_SET_STATION_ADDRESS { PXENV_STATUS Status; MAC_ADDR StationAddress; } t_PXENV_UNDI_SET_STATION_ADDR;	**Set before calling API service** **StationAddress:** Temporary MAC address to be used for transmit and receive**.** **Returned from API service** **Status**: See the PXENV_STATUS_xxx constants.

UNDI SET PACKET FILTER

Op-Code:	PXENV_UNDI_SET_PACKET_FILTER (000Bh)
Input:	Far pointer to a t_PXENV_UNDI_SET_PACKET_FILTER parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This call resets the adapter's receive unit to accept a new filter, different from the one provided with the open call.

typedef struct s_PXENV_UNDI_SET_PACKET_FILTER { PXENV_STATUS Status; UINT8 filter; } t_PXENV_UNDI_SET_PACKET_FILTER;	**Set before calling API service** **Filter**: See the receive filter values in the UNDI OPEN (0006h) API description. **Returned from API service** **Status**: See the PXENV_STATUS_xxx constants.

UNDI GET INFORMATION

Op-Code:	PXENV_UNDI_GET_INFORMATION (000Ch)
Input:	Far pointer to a t_PXENV_UNDI_GET_INFORMATION parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This call copies the network adapter variables, including the MAC address, into the input buffer.
Note:	The PermNodeAddress field must be valid after PXENV_START_UNDI and PXENV_UNDI_STARTUP have been issued. All other fields must be valid after PXENV_START_UNDI, PXENV_UNDI_STARTUP and PXENV_UNDI_INITIALIZE have been called.

```
typedef struct s_PXENV_UNDI_GET_INFORMATION
{
PXENV_STATUS Status;
UINT16 BaseIo;
UINT16 IntNumber;
UINT16 MaxTranUnit;
UINT16 HwType;
#define ETHER_TYPE      1
#define EXP_ETHER_TYPE  2
#define IEEE_TYPE       6
#define ARCNET_TYPE     7

UINT16 HwAddrLen;
MAC_ADDR CurrentNodeAddress;
MAC_ADDR PermNodeAddress;
SEGSEL ROMAddress;
UINT16 RxBufCt;
UINT16 TxBufCt;
} t_PXENV_UNDI_GET_INFORMATION;
```

Set before calling API service

N/A

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

BaseIO: Adapter base I/O address.

IntNumber: Adapter IRQ number.

MaxTranUnit: Adapter maximum transmit unit.

HWType: Type of protocol at the hardware level.

HWAddrLen: Length of the hardware address.

CurrentNodeAddress: Current hardware address.

PermNodeAddress: Permanent hardware address.

ROMAddress: Real mode ROM segment address.

RxBufCnt: Receive queue length.

TxBufCnt: Transmit queue length.

UNDI GET STATISTICS

Op-Code:	PXENV_UNDI_GET_STATISTICS (000Dh)
Input:	Far pointer to a t_PXENV_UNDI_GET_STATISTICS parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This call reads statistical information from the network adapter, and returns.

```
typedef struct s_PXENV_UNDI_GET_STATISTICS
{
PXENV_STATUS Status;
UINT32 XmtGoodFrames;
UINT32 RcvGoodFrames;
UINT32 RcvCRCErrors;
UINT32 RcvResourceErrors;
} t_PXENV_UNDI_GET_STATISTICS;
```

Set before calling API service

N/A

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

XmtGoodFrames: Number of successful transmissions.

RcvGoodFrames: Number of good frames received.

RcvCRCErrors: Number of frames received with CRC error.

RcvResourceErrors: Number of frames discarded because receive queue was full.

UNDI CLEAR STATISTICS

Op-Code:	PXENV_UNDI_CLEAR_STATISTICS (000Eh)
Input:	Far pointer to a t_PXENV_UNDI_CLEAR_STATISTICS parameter.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This call clears the statistical information from the network adapter.

```
typedef struct s_PXENV_UNDI_CLEAR_STATISTICS
{
PXENV_STATUS Status;
} t_PXENV_UNDI_CLEAR_STATISTICS;
```

Set before calling API service

N/A

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

UNDI INITIATE DIAGS

Op-Code:	PXENV_UNDI_INITIATE_DIAGS (000Fh)

Input:	Far pointer to a t_PXENV_UNDI_INITIATE_DIAGS parameter.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This call can be used to initiate the run-time diagnostics. It causes the network adapter to run hardware diagnostics and to update its status information.

```
typedef struct s_PXENV_UNDI_INITIATE_DIAGS
{
PXENV_STATUS Status;
} t_PXENV_UNDI_INITIATE_DIAGS;
```

Set before calling API service

N/A

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

UNDI FORCE INTERRUPT

Op-Code:	PXENV_UNDI_FORCE_INTERRUPT (0010h)
Input:	Far pointer to a t_PXENV_UNDI_FORCE_INTERRUPT parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This call forces the network adapter to generate an interrupt. When a receive interrupt occurs, the network adapter driver usually queues the packet and calls the application's callback receive routine with a pointer to the packet received. Then, the callback routine either can copy the packet to its buffer or can decide to delay the copy to a later time. If the packet is not immediately copied, the network adapter driver does not remove it from the input queue. When the application wants to copy the packet, it can call the PXENV_UNDI_FORCE_INTERRUPT routine to simulate the receive interrupt.

```
typedef struct s_PXENV_UNDI_FORCE_INTERRUPT
{
PXENV_STATUS Status;
} t_PXENV_UNDI_FORCE_INTERRUPT;
```

Set before calling API service

N/A

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

UNDI GET MULTICAST ADDRESS

Op-Code:	PXENV_UNDI_GET_MCAST_ADDRESS (0011h)
Input:	Far pointer to a t_PXENV_GET_MCAST_ADDRESS parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This call converts the given IP multicast address to a hardware multicast address.

```
typedef struct s_PXENV_UNDI_GET_MCAST_ADDRESS
{
PXENV_STATUS Status;
IP4 InetAddr;
MAC_ADDR MediaAddr;
} t_PXENV_UNDI_GET_MCAST_ADDR;
```

Set before calling API service

InetAddr: IP multicast address.

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

MediaAddr: MAC multicast address.

UNDI GET NIC TYPE

Op-Code:	PXENV_UNDI_GET_NIC_TYPE (0012h)
Input:	Far pointer to a t_ PXENV_UNDI_GET_NIC_TYPE parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants. If the PXENV_EXIT_SUCCESS is returned the parameter structure must contain the NIC information.

Description:	This call, if successful, provides the NIC-specific information necessary to identify the network adapter that is used to boot the system.
Note:	The application first gets the DHCPDISCOVER packet using GET_CACHED_INFO and checks if the UNDI is supported before making this call. If the UNDI is not supported, the NIC-specific information can be obtained from the DHCPDISCOVER packet itself. PXENV_START_UNDI, PXENV_UNDI_STARTUP and PXENV_UNDI_INITIALIZE must be called before the information provided is valid.

```
typedef s_PXENV_UNDI_GET_NIC_TYPE
{
PXENV_STATUS Status;
UINT8 NicType;
#define PCI_NIC        2
#define PnP_NIC        3
#define CardBus_NIC    4

  Union {
    Struct {
      UINT16 Vendor_ID;
      UINT16 Dev_ID;
      UINT8 Base_Class;
      UINT8 Sub_Class;
      UINT8 Prog_Intf;
      UINT8 Rev;
      UINT16 BusDevFunc;
      UINT16 SubVendor_ID;
      UINT16 SubDevice_ID;
    } pci, cardbus;
    struct {
      UINT32 EISA_Dev_ID;
      UINT8 Base_Class;
      UINT8 Sub_Class;
      UINT8 Prog_Intf;
      UINT16 CardSelNum;
    } pnp;
  } info;
} t_PXENV_UNDI_GET_NIC_TYPE;
```

Set before calling API service

N/A

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

NICType: Type of NIC information stored in the parameter structure.

Info: Information about the fields in this union can be found in the [PnP] and [PCI] specifications

UNDI GET IFACE INFO

Op-Code:	PXENV_UNDI_GET_IFACE_INFO (0013h)
Input:	Far pointer to a t_ PXENV_UNDI_GET_IFACE_INFO parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants. If the PXENV_EXIT_SUCCESS is returned, the parameter structure must contain the interface specific information.
Description:	This call, if successful, provides the network interface specific information such as the interface type at the link layer (Ethernet, Tokenring) and the link speed. This information can be used in the universal drivers such as NDIS or Miniport to communicate to the upper protocol modules.
Note:	UNDI follows the NDIS2 specification in giving this information. It is the responsibility of the universal driver to translate/convert this information into a format that is required in its specification or to suit the expectation of the upper level protocol modules. PXENV_START_UNDI, PXENV_UNDI_STARTUP and PXENV_UNDI_INITIALIZE must be called before the information provided is valid.

`typedef struct s_PXENV_UNDI_GET_IFACE_INFO`
`{`
`PXENV_STATUS Status`
`UINT8 IfaceType[16];`
`UINT32 LinkSpeed;`
`UINT32 ServiceFlags;`
`UINT32 Reserved[4];`
`} t_PXENV_UNDI_GET_NDIS_INFO;`
	Set before calling API service
	N/A
	Returned from API service
	Status: See the PXENV_STATUS_xxx constants.
	IfaceType: Name of MAC type in ASCIIZ format. This is used by the universal NDIS driver to specify its driver type to the protocol driver.
	LinkSpeed: Defined in the NDIS 2.0 specification.
	ServiceFlags: Defined in the NDIS 2.0 specification.
	Reserved: Must be zero.

UNDI GET STATE

Op-Code:	PXENV_UNDI_GET_STATE (0015h)
Input:	Far pointer to a t_PXENV_UNDI_GET_STATE parameter.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants. The UNDI_STATE field in the parameter structure must be set to one of the valid state constants
Description:	This call can be used to obtain state of the UNDI engine in order to avoid issuing adverse call sequences

`typedef struct s_PXENV_UNDI_GET_STATE`
`{`
`#define PXE_UNDI_GET_STATE_STARTED 1`
`#define PXE_UNDI_GET_STATE_INITIALIZED 2`
`#define PXE_UNDI_GET_STATE_OPENED 3`
`PXENV_STATUS Status;`
`UINT8 UNDIstate;`
`} t_PXENV_UNDI_GET_STATE;`
	Set before calling API service
	N/A
	Returned from API service
	Status: See the PXENV_STATUS_xxx constants.
	State: See definitions of the state constants.
	Note. UNDI implementation is responsible for maintaining internal state machine.

UNDI ISR

Op-Code:	PXENV_UNDI_ISR (0014h)
Input:	Far pointer to a t_PXENV_UNDI_ISR parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This API function will be called at different levels of processing the interrupt. The FuncFlag field in the parameter block indicates the operation to be performed for the call. This field is filled with the status of that operation on return.

Note:	Interrupt Service Routine Operation: In this design the UNDI does not hook the interrupt for the Network Interface. Instead, the application or the protocol driver hooks the interrupt and calls UNDI with the PXENV_UNDI_ISR API call for interrupt verification (PXENV_UNDI_ISR_IN_START) and processing (PXENV_UNDI_ISR_IN_PROCESS and PXENV_UNDI_ISR_GET_NEXT). When the Network Interface HW generates an interrupt the protocol driver's interrupt service routine (ISR) gets control and takes care of the interrupt processing at the PIC level. The ISR then calls the UNDI using the PXENV_UNDI_ISR API with the value PXENV_UNDI_ISR_IN_START for the FuncFlag parameter. At this time UNDI must disable the interrupts at the Network Interface level and read any status values required to further process the interrupt. UNDI must return as quickly as possible with one of the two values, PXENV_UNDI_ISR_OUT_OURS or PXENV_UNDI_ISR_OUT_NOT_OURS, for the parameter FuncFlag depending on whether the interrupt was generated by this particular Network Interface or not. If the value returned in FuncFlag is PXENV_UNDI_ISR_OUT_NOT_OURS, then the interrupt was not generated by our NIC, and interrupt processing is complete. If the value returned in FuncFlag is PXENV_UNDI_ISR_OUT_OURS, the protocol driver must start a handler thread and send an end-of-interrupt (EOI) command to the PIC. Interrupt processing is now complete. The protocol driver strategy routine will call UNDI using this same API with FuncFlag equal to PXENV_UNDI_ISR_IN_PROCESS. At this time UNDI must find the cause of this interrupt and return the status in the FuncFlag. It first checks if there is a frame received and if so it returns the first buffer pointer of that frame in the parameter block. The protocol driver calls UNDI repeatedly with the FuncFlag equal to PXENV_UNDI_ISR_IN_GET_NEXT to get all the buffers in a frame and also all the received frames in the queue. On this call, UNDI must remember the previous buffer given to the protoco,l remove it from the receive queue and recycle it. In case of a multi-buffered frame, if the previous buffer is not the last buffer in the frame it must return the next buffer in the frame in the parameter block. Otherwise it must return the first buffer in the next frame. If there is no received frame pending to be processed, UNDI processes the transmit completes and if there is no other interrupt status to be processed, UNDI re-enables the interrupt at the NETWORK INTERFACE level and returns PXENV_UNDI_ISR_OUT_DONE in the FuncFlag. **IMPORTANT: It is possible for the protocol driver to be interrupted again while in the strategy routine when the UNDI re-enables interrupts.**

```
typedef struct s_PXENV_UNDI_ISR
{
PXENV_STATUS Status;
UINT16 FuncFlag;
UINT16 BufferLength;
UINT16 FrameLength;
UINT16 FrameHeaderLength;
SEGOFF16 Frame;
UINT8 ProtType;
UINT8 PktType;
} t_PXENV_UNDI_ISR;

#define PXENV_UNDI_ISR_IN_START      1
#define PXENV_UNDI_ISR_IN_PROCESS    2
#define PXENV_UNDI_ISR_IN_GET_NEXT    3

/* One of these will be returned for
PXENV_UNDI_ISR_IN_START */
#define PXENV_UNDI_ISR_OUT_OURS      0
#define PXENV_UNDI_USR_OUT_NOT_OURS  1

/* One of these will be returned for
PXENV_UNDI_ISR_IN_PROCESS and
PXENV_UNDI_ISR_IN_GET_NEXT */
#define PXENV_UNDI_ISR_OUT_DONE      0
#define PXENV_UNDI_ISR_OUT_TRANSMIT  2
#define PXENV_UNDI_ISR_OUT_RECEIVE   3
#define PXENV_UNDI_ISR_OUT_BUSY      4
```

Set before calling API service

FuncFlag: One of the PXENV_UNDI_ISR_IN_xxx constants.

Returned from API service

Status: See the PXENV_STATUS_xxx constants.

FuncFlag: One of the PXENV_UNDI_ISR_OUT_xxx constants.

BufferLength: This parameter contains the length of the data in the buffer given by Frame.

FrameLength: This parameter contains the total length of the receive frame. A receive frame may contain more than one data buffer. If FrameLength is not the same as BufferLength, the application will have to call PXENV_UNDI_ISR several times to receive the complete frame. In the case of the multi buffered frame, all buffers (except the last one) must contain at least 512 bytes of data (this does not include the media header). In other words, the minimum buffer size is 512 bytes plus the size of the media header.

FrameHeaderLength: This field contains the length of the media header in the buffer given in Frame. This field must be zero if the buffer is not the first one in a multi buffered frame.

Frame: This field defines the pointer to a buffer in the receive frame. In protected mode, the selector must be same as the UNDI data segment. This is required to obtain the virtual address in the protected mode protocol drivers.

ProtType: This field contains the protocol identifier (1=IP, 2=ARP, 3=RARP and 4=OTHER) for the received packet.

PktType: This field contains the type of frame received (0-directed/promiscuous, 1-broadcast and 2-multicast).

The following flowchart shows how a protocol driver (for example Base-Code and NDIS.DOS) interfaces with UNDI when servicing interrupts.

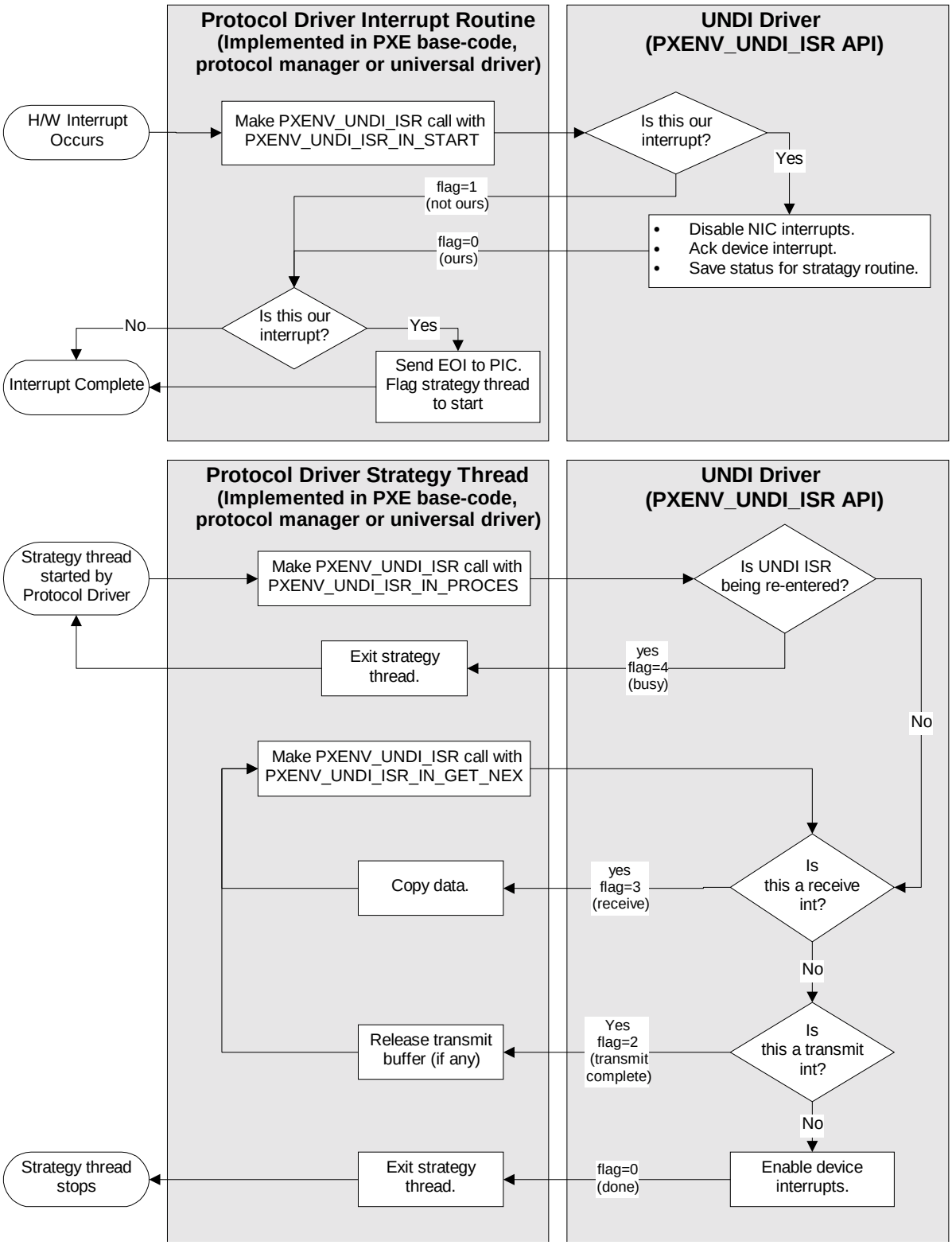

Figure 3-5 Interrupt Service Routine Operation

3.5 PXE Return Status Definitions

Important: The code provided in this section is provided for information purposes only.

```
/*
 *
* Copyright(c) 1997/1998 by Intel Corporation. All Rights Reserved.
 *
/* Parameter/Result structure storage types.
 */
typedef unsigned char UINT8;
typedef unsigned short UINT16;
typedef unsigned long UINT32;
typedef signed char INT8;
typedef signed short INT16;
typedef signed long INT32;

/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =*/
/* Exit codes returned in AX by a PXENV API service.
 */
#define    PXENV_EXIT_SUCCESS            0x0000
#define    PXENV_EXIT_FAILURE            0x0001

/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = */
/* Status codes returned in the status word of PXENV API parameter
 * structures. Some of these codes are also used to return status
 * from an NBP to the boot ROM. */

/* Generic API status & error codes that are reported by the loader */

#define    PXENV_STATUS_SUCCESS                                    0x00
#define    PXENV_STATUS_FAILURE                                    0x01
/* General failure. */

#define    PXENV_STATUS_BAD_FUNC                                   0x02
/* Invalid function number. */

#define    PXENV_STATUS_UNSUPPORTED                                0x03
/* Function is not yet supported. */

#define    PXENV_STATUS_KEEP_UNDI                                  0x04
/* UNDI must not be unloaded from base memory. */

#define    PXENV_STATUS_KEEP_ALL                                   0x05

#define    PXENV_STATUS_OUT_OF_RESOURCES                           0x06
/* Base-code and UNDI must not be unloaded from base memory. */


/* ARP errors (0x10 to 0x1F) */
#define    PXENV_STATUS_ARP_TIMEOUT                                0x11

/* Base-Code state errors */
#define    PXENV_STATUS_UDP_CLOSED                                 0x18
#define    PXENV_STATUS_UDP_OPEN                                   0x19
#define    PXENV_STATUS_TFTP_CLOSED                                0x1A
#define    PXENV_STATUS_TFTP_OPEN                                  0x1B


/* BIOS/system errors (0x20 to 0x2F) */
#define    PXENV_STATUS_MCOPY_PROBLEM                              0x20
#define    PXENV_STATUS_BIS_INTEGRITY_FAILURE                      0x21
#define    PXENV_STATUS_BIS_VALIDATE_FAILURE                       0x22
#define    PXENV_STATUS_BIS_INIT_FAILURE                           0x23
#define    PXENV_STATUS_BIS_SHUTDOWN_FAILURE                       0x24
#define    PXENV_STATUS_BIS_GBOA_FAILURE                           0x25
#define    PXENV_STATUS_BIS_FREE_FAILURE                           0x26
#define    PXENV_STATUS_BIS_GSI_FAILURE                            0x27
#define    PXENV_STATUS_BIS_BAD_CKSUM                              0x28


/* TFTP/MTFTP errors (0x30 to 0x3F) */
#define    PXENV_STATUS_TFTP_CANNOT_ARP_ADDRESS                    0x30
#define    PXENV_STATUS_TFTP_OPEN_TIMEOUT                          0x32
```

```
#define    PXENV_STATUS_TFTP_UNKNOWN_OPCODE                    0x33
#define    PXENV_STATUS_TFTP_READ_TIMEOUT                      0x35
#define    PXENV_STATUS_TFTP_ERROR_OPCODE                      0x36
#define    PXENV_STATUS_TFTP_CANNOT_OPEN_CONNECTION            0x38
#define    PXENV_STATUS_TFTP_CANNOT_READ_FROM_CONNECTION       0x39
#define    PXENV_STATUS_TFTP_TOO_MANY_PACKAGES                 0x3A
#define    PXENV_STATUS_TFTP_FILE_NOT_FOUND                    0x3B
#define    PXENV_STATUS_TFTP_ACCESS_VIOLATION                  0x3C
#define    PXENV_STATUS_TFTP_NO_MCAST_ADDRESS                  0x3D
#define    PXENV_STATUS_TFTP_NO_FILESIZE                       0x3E
#define    PXENV_STATUS_TFTP_INVALID_PACKET_SIZE               0x3F

/* Reserved errors 0x40 to 0x4F) */


/* DHCP/BOOTP errors (0x50 to 0x5F) */
#define    PXENV_STATUS_DHCP_TIMEOUT                           0x51
#define    PXENV_STATUS_DHCP_NO_IP_ADDRESS                     0x52
#define    PXENV_STATUS_DHCP_NO_BOOTFILE_NAME                  0x53
#define    PXENV_STATUS_DHCP_BAD_IP_ADDRESS                    0x54

/* Driver errors (0x60 to 0x6F) */
/* These errors are for UNDI compatible NIC drivers. */
#define    PXENV_STATUS_UNDI_INVALID_FUNCTION                  0x60
#define    PXENV_STATUS_UNDI_MEDIATEST_FAILED                  0x61
#define    PXENV_STATUS_UNDI_CANNOT_INIT_NIC_FOR_MCAST         0x62
#define    PXENV_STATUS_UNDI_CANNOT_INITIALIZE_NIC             0x63
#define    PXENV_STATUS_UNDI_CANNOT_INITIALIZE_PHY             0x64
#define    PXENV_STATUS_UNDI_CANNOT_READ_CONFIG_DATA           0x65
#define    PXENV_STATUS_UNDI_CANNOT_READ_INIT_DATA             0x66
#define    PXENV_STATUS_UNDI_BAD_MAC_ADDRESS                   0x67
#define    PXENV_STATUS_UNDI_BAD_EEPROM_CHECKSUM               0x68
#define    PXENV_STATUS_UNDI_ERROR_SETTING_ISR                 0x69
#define    PXENV_STATUS_UNDI_INVALID_STATE                     0x6A
#define    PXENV_STATUS_UNDI_TRANSMIT_ERROR                    0x6B
#define    PXENV_STATUS_UNDI_INVALID_PARAMETER                 0x6C

/* ROM and NBP Bootstrap errors (0x70 to 0x7F) */
#define    PXENV_STATUS_BSTRAP_PROMPT_MENU                     0x74
#define    PXENV_STATUS_BSTRAP_MCAST_ADDR                      0x76
#define    PXENV_STATUS_BSTRAP_MISSING_LIST                    0x77
#define    PXENV_STATUS_BSTRAP_NO_RESPONSE                     0x78
#define    PXENV_STATUS_BSTRAP_FILE_TOO_BIG                    0x79

/* Environment NBP errors (0x80 to 0x8F) */


/* Reserved errors (0x90 to 0x9F) */

/* Misc. errors (0xA0 to 0xAF) */
#define    PXENV_STATUS_BINL_CANCELED_BY_KEYSTROKE             0xA0
#define    PXENV_STATUS_BINL_NO_PXE_SERVER                     0xA1
#define    PXENV_STATUS_NOT_AVAILABLE_IN_PMODE                 0xA2
#define    PXENV_STATUS_NOT_AVAILABLE_IN_RMODE                 0xA3

/* BUSD errors (0xB0 to 0xBF) */
#define    PXENV_STATUS_BUSD_DEVICE_NOT_SUPPORTED              0xB0


/* Loader errors (0xC0 to 0xCF) */
#define    PXENV_STATUS_LOADER_NO_FREE_BASE_MEMORY             0xC0
#define    PXENV_STATUS_LOADER_NO_BC_ROMID                     0xC1
#define    PXENV_STATUS_LOADER_BAD_BC_ROMID                    0xC2
#define    PXENV_STATUS_LOADER_BAD_BC_RUNTIME_IMAGE            0xC3
#define    PXENV_STATUS_LOADER_NO_UNDI_ROMID                   0xC4
#define    PXENV_STATUS_LOADER_BAD_UNDI_ROMID                  0xC5
#define    PXENV_STATUS_LOADER_BAD_UNDI_DRIVER_IMAGE           0xC6
#define    PXENV_STATUS_LOADER_NO_PXE_STRUCT                   0xC8
#define    PXENV_STATUS_LOADER_NO_PXENV_STRUCT                 0xC9
#define    PXENV_STATUS_LOADER_UNDI_START                      0xCA
#define    PXENV_STATUS_LOADER_BC_START                        0xCB


/* Vendor errors (0xD0 to 0xFF) */
```

4. PXE Initial Program Load (IPL)

4.1 Overview

Understanding the Initial Program Load (IPL) process requires some background on the evolution of Intel Processor Architecture because the IPL process has evolved and become more complex over time. This section identifies the initial system architecture and the evolutionary steps related to IPL that have occurred since the first IBM PC was introduced in 1981.

The original IBM PC used an Intel 8088 processor with a 20-bit addressable memory space or 1MB. This processor uses a segmented architecture where a segment may begin on any 16-byte boundary. To reach a particular memory location requires two 16-bit registers within the processor: a segment register and an offset register. The segment register is set to the upper 16-bits of a 20-bit address and the offset register is used to access anywhere within a 64KB address range from the base address set by the segment register.

The 1MB address space is also subdivided into several functional areas. When an x86 processor is reset, it begins execution 16 bytes from the top of memory. To handle reset vectoring, a ROM-based Basic Input Output System or BIOS is placed at the top of memory. Since the BIOS is typically in Read-Only Memory it is often referred to as ROM BIOS. The BIOS provides a standard software interface to system hardware and Power On Self Test or POST processing. Initially, the top 64KB were reserved for the BIOS.

The first 1KB of address space is used as vectors for the 256 levels of interrupts supported by Intel x86 processors. Several of these vectors are used for traditional hardware event notification. The x86 processor also supports the concept of a software interrupt invoked explicitly by the software INT instruction.

The software interrupt feature permits a process in one area of memory to invoke a second process in another area without having to know the address of the second process. In addition, if the second process performs a well-defined function, another process may replace the interrupt vector with the address of a local routine to filter or completely replace the first process. The ROM BIOS uses software interrupts as the method for entering most BIOS-provided services.

After power-on or reset, the ROM BIOS POST process first initializes BIOS-aware system hardware to a known state. POST also establishes a number of system interrupt vectors, both hardware and software. The software vectors serve as a means of invoking BIOS services using the INT instruction.

After the interrupt vectors, the rest of the lower 640KBof address space is typically used for BIOS scratch pad storage, the operating system and temporary program space. The next 128KB are set aside for memory mapped video buffers. The remaining address space below the ROM BIOS and above the video buffers was initially reserved for the use of Option ROMs. Over time this area has come to be called Upper Memory.

An Option ROM is used to extend the services or capabilities of the BIOS prior to IPL. It is the only way, other than directly modifying the BIOS, that new devices may be added to the IPL process. During POST, the BIOS scans the Upper Memory area for Option ROMs that have been mapped into

this space by adapter cards plugged into a system expansion bus. A valid Option ROM begins on a 2KB boundary and contains a data structure with a signature, the length of the Option ROM and an entry point for initialization code.

If a valid Option ROM is located by the BIOS, the ROM's initialization code is invoked. Option ROMs replace or filter standard BIOS services by replacing the BIOS initialized interrupt vectors.

The original IPL sources for the first Intel architecture system (the IBM PC) included cassette tape and floppy drives. Fixed disks were added to the standard IPL process with the introduction of the IBM XT, though several vendors had previously provided proprietary approaches. The IBM XT used an Option ROM on the fixed disk controller to re-vector the diskette support routine in the BIOS and install additional routines for supporting partitioned fixed disk drives. Within the IBM definition of IPL, only the first floppy drive or the first fixed drive was bootable.

The IBM AT followed the IBM XT. The IBM AT introduced a new Intel processor, the 80286. This processor was backward compatible with the 8086 when operated in what was called real mode. Real mode was the default mode when the processor was reset or first powered-on. The 80286 also offered an additional mode of operation called protected mode. Protected mode expanded the addressable memory space to 24 bits or 16MB. Support for protected mode on the 80286 was limited, but it did allow simple storage of data above 1 megabyte in what was termed extended memory. Routines added to the ROM BIOS and still available today provide simple methods to access Extended Memory.

Subsequent Intel Architecture systems have added more powerful processors with even larger address spaces of 32 bits or 4GB and new modes of operation. However, for backward compatibility, real mode continues to be used for IPL.

Until the widespread acceptance of Windows, most operating systems also continued to use real mode. This meant most applications were limited to the amount of system memory left after loading the operating system and any additional device drivers into System Memory. As the operating system and associated device drivers became more complex they required more System Memory. To relieve what came to be called RAM cram, the Upper Memory area became the location where hardware "windows" were created to bank switch Expansion Memory, really an entirely separate address space, into the 1 megabyte address space of real mode. Initially, this required special hardware and worked best for data storage. As the technology evolved, Expansion Memory began to be used to store executable programs, especially device drivers that had been taking up precious system memory.

At some point, system vendors realized that copying Option ROMs into RAM improved overall system performance. This was due to the fact that the access speed for RAM devices was much faster than ROM devices. So ROM BIOS was again extended, this time to copy Option ROMs into special Shadow RAM that was then mapped over the Option ROM. After copying, the Shadow RAM was write-protected to make sure it exhibited the same characteristics as the original Option ROM.

It became apparent that Shadow RAM could also be used to provide some of the same benefits of Expansion Memory. In particular, if Shadow RAM was mapped into a vacant Upper Memory space, the same device drivers previously moved into Expansion Memory could be moved into Shadow RAM without requiring the specialized Expansion Memory hardware.

At the same time systems were increasing in memory size and processor speed, more and faster peripherals were being added and the expansion bus was becoming a limiting factor for system performance. The speed of the initial Industry Standard Architecture or ISA bus was increased from 4.77 MHz to 6 MHz and beyond and then replaced or augmented by a range of buses known as Micro

Channel, EISA and VESA. Though these expansion busses may have been electrically different, the IPL process remained the same, until the Peripheral Component Interconnect or PCI bus.

The PCI bus affected IPL in two ways. First, Option ROMs were no longer hardware-mapped into the Upper Memory area. Instead, Option ROM images on PCI cards were copied into Shadow RAM for initialization. This allowed the image copied into Shadow RAM to shrink by discarding initialization code after initialization. The initialization code simply updates the length in the Option ROM header also copied into Shadow RAM. The BIOS POST also provided bus specific information about the device to the initialization code. This allowed multiple devices of the same type to be resident in the system, each with their own Option ROM support.

The [PnP] Specification also defines a method for Option ROMs on non-PCI devices to indicate they support being copied into Shadow RAM in Upper Memory. The Device Driver Interface Module (DDIM) is a single page at the end of the [PnP] specification that describes how an Option ROM can identify this capability.

The second way the PCI bus affected IPL was the corresponding [PnP] Specification. This specification defined Plug and Play headers in the Option ROM. These headers provide additional information that includes IPL-related information. In particular, there are two fields that contain pointers to special extended initialization routines:

- Boot Connection Vector (BCV)
- Bootstrap Entry Vector (BEV)

The BCV is used for devices that replace or supplement the BIOS disk IPL services to be a bootable device. The BEV is used for devices that have an alternative IPL method not related to the BIOS disk services. Thus, PCI and the Plug and Play BIOS specification provided some of the initial framework required to organize bootable devices. In 1996, Compaq, Intel and Phoenix introduced the [BBS] Specification. The [BBS] uses information from the multiple specifications related to IPL and attempts to present a unified definition for BIOS Boot technology. The [BBS] identifies different categories of boot devices:

- BIOS Aware IPL Devices (BAID)
- Boot Connection Vector (BCV) devices
- Bootstrap Entry Vector (BEV) devices

The BIOS maintains an IPL Table listing all of the possible IPL sources. BIOS Aware IPL Devices have all the code necessary to perform IPL resident in the system BIOS. BAID devices include floppy or fixed drives.

BCV and BEV devices use Option ROMs associated with the IPL device to extend the IPL Table. These Option ROMs may be resident on the associated device or in non-volatile storage on the motherboard. PXE is implemented as a BEV device.

The BIOS adds IPL devices to the BBS IPL Table already containing the BAIDs. These devices are identified during the BIOS Option ROM scan process. If the device hardware is detected and the rest of Option ROM initialization is successful (in other words, the Option ROM initialization code and loader code are placed in upper memory), control is returned to the BIOS indicating the Option ROM manages an IPL device. Within the ROM, a Plug and Play Expansion header will be present for each bootable device supported by the Option ROM. Each PnP Expansion header for a PXE IPL device will have a non-zero BEV.

Once Option ROM scan is complete, the BIOS builds a list of bootable devices using the information obtained during the scan. According to the [BBS], the priority list for these devices is established by the enduser through BIOS setup.

4.2 PXE Split ROM Architecture

Prior to this specification, all PXE Option ROMs were implemented as a monolithic Option ROM with an option ROM header that encapsulates three components (see Figure 4-1). However, there are a number of advantages to implementing each component separately and providing a method for the components to find each other and bind at run time. (This type of implementation is an implementation of the PXE Split ROM Architecture.)

The first advantage of separating the ROM components is that through re-use of common components it is possible for a platform to support more than one PXE device without linearly increasing the code storage requirements for each new device.

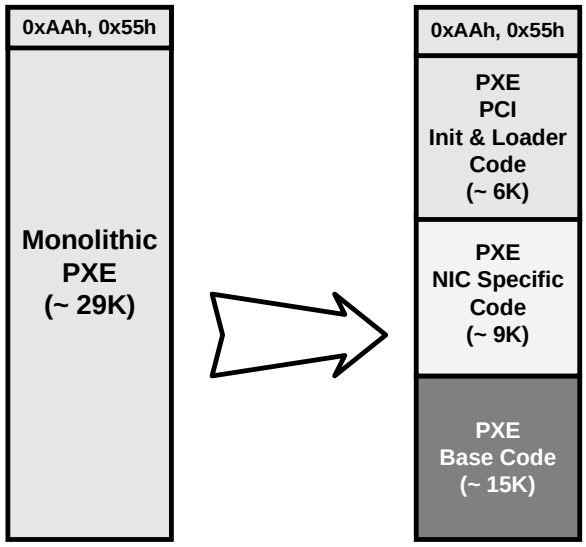

Figure 4-1 Pre-Split ROM PXE Architecture

Second, by providing access to the NIC specific code (the UNDI driver) by breaking out the code and allowing it to be loaded independently from the other components, the BIOS may make direct use of the Network Interface for sending alerts.

Finally, implementing the components separately allows the UNDI driver to be carried on the NIC, while the BIOS can supply the Base-Code. Figure 4-2 illustrates this implementation.

The PXE Split ROM Architecture specifies three different Option ROMs that work cooperatively to create a working PXE that supports one or more network interfaces. The three Option ROMs are, the Base-Code ROM (BC ROM), the UNDI ROM, and the BUSD ROM.

The Base-Code Option ROM is common code to support one or more instances of the other ROMs. The BC ROM provides the protocol stack in addition to various loader and initialization code. A Base-Code ROM is required.

The UNDI ROM provides the network interface specific code. An UNDI ROM is specific to particular network interface hardware. UNDI provides a set of APIs that allow the remotely booted program to use universal protocol drivers. (A universal protocol driver is a driver written to interface to the UNDI API.) A system must contain at least one (but may contain several) UNDI ROMs, where each ROM would support one network interface.

The BUSD ROM provides bus support for CardBus-based network interface cards. BUSD is only required if supporting CardBus. Unlike the Base-Code and UNDI ROMs, BUSD's call mechanism must be added to the BIOS.

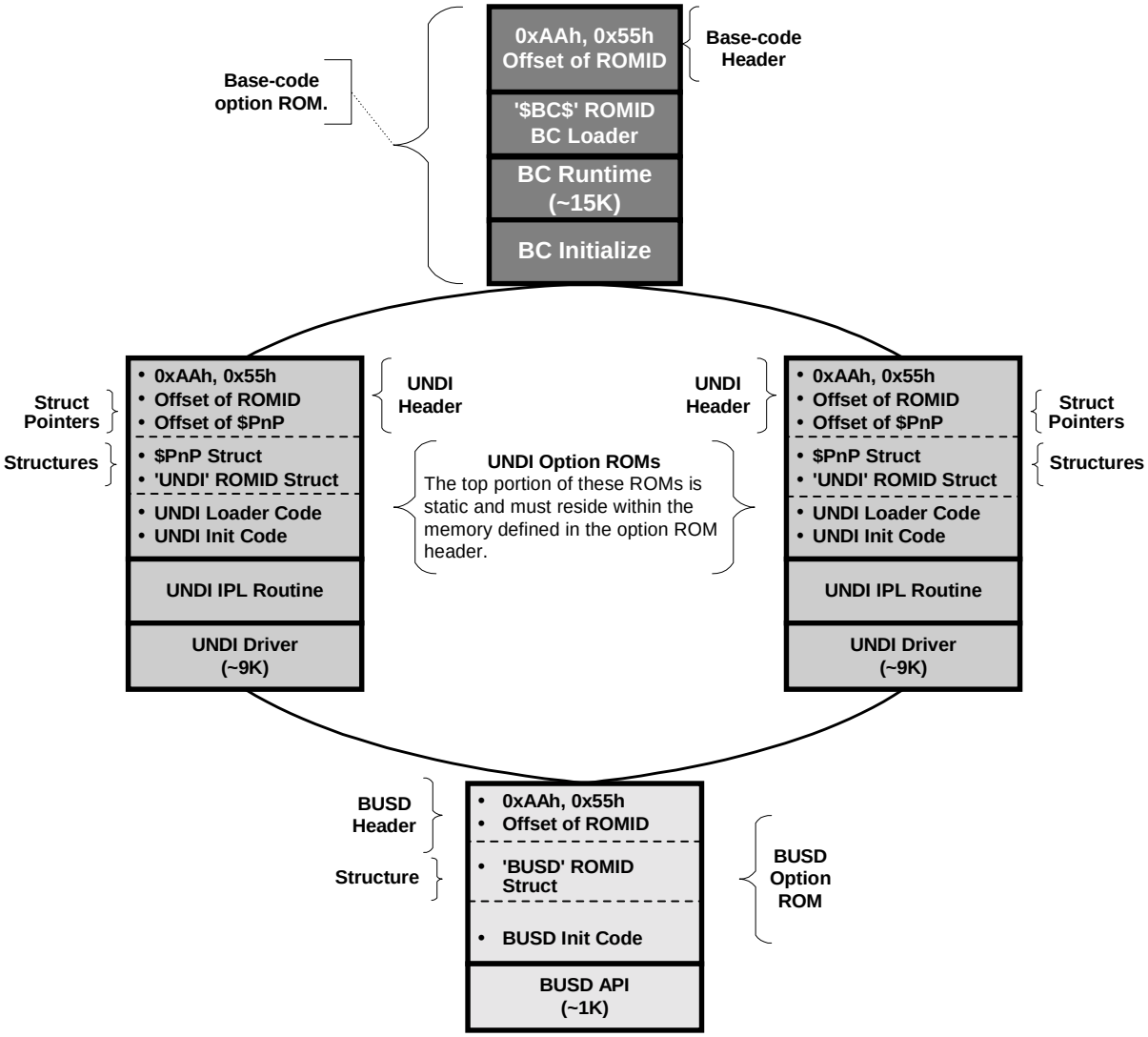

Figure 4-2 Split Base Code and UNDI Code

4.3 PXE Option ROM Components

There are several required ROM components that a compliant PXE Option ROM must have to guarantee the ability to boot in the PXE environment.

4.3.1 Option ROM header

A compliant PXE Option ROM header contains a pointer to a PXE ROM ID structure at offset 16h as depicted in Table 4-1.

Table 4-1 Option ROM Header for PXE ROMs

Offset	*Size*	*Name*	*Contents*
00h	02h	Signature	PC/AT option ROM signature. 55h, 0AAh
02h	01h	ROMLength	Size of this Option ROM is 512-byte blocks
03h	04h	InitEntryPointer	Initialization entry point for all implementations. This is usually a jump to the initialization routine.
07h	15h	Reserved	Varies
16h	02h	PXEROMID	Offset of the PXE ROM ID Structure for this PXE Split Option ROM. BC, UNDI, BUSD ROM structures are defined in the specification. If this is not a split ROM implementation then this field is zero.
18h	02h	PCIRHeader	Offset of the PCI Expansion Header
1Ah	02h	PnPHeader	Offset of the Plug and Play Expansion Header.

4.3.2 Initialization Routine

For option ROMs that are also boot ROMs, the initialization routine is responsible for registering the IPL routine with the system BIOS. The initialization routine is also used to prepare the UMB and save initialization parameters for later use.

4.3.3 IPL Routine

UNDI option ROMs have an IPL routine. The IPL routine is registered with the PXE-compliant BIOS by an offset in the PnP expansion header. Information about the PnP expansion header contents can be found in [PnP] and [BBS].

4.3.4 Loader Routine

UNDI split ROMs must have a loader routine that will install the UNDI driver into base memory so it can be used.

Base-Code split ROMs must have a loader routine that will allocate base memory, call the UNDI loader routine, install the base-code runtime into base memory and start the UNDI driver and base-code runtime.

4.3.5 UNDI Driver

The UNDI driver may be included within the memory defined by the option ROM header.

An UNDI driver must be capable of executing in real mode and 16:16 protected mode with 16-bit (SP) or 32-bit (ESP) stack segments. Separate entry points for 16-bit and 32-bit stack segments are defined in the !PXE structure.

UNDI driver API specifications, parameter passing and status/result codes are covered in other sections in this document

4.4 PXE Boot Sequence

Figure 4-3 provides a flowchart outlining the system boot timeline. This is followed by detailed information about the flowchart.

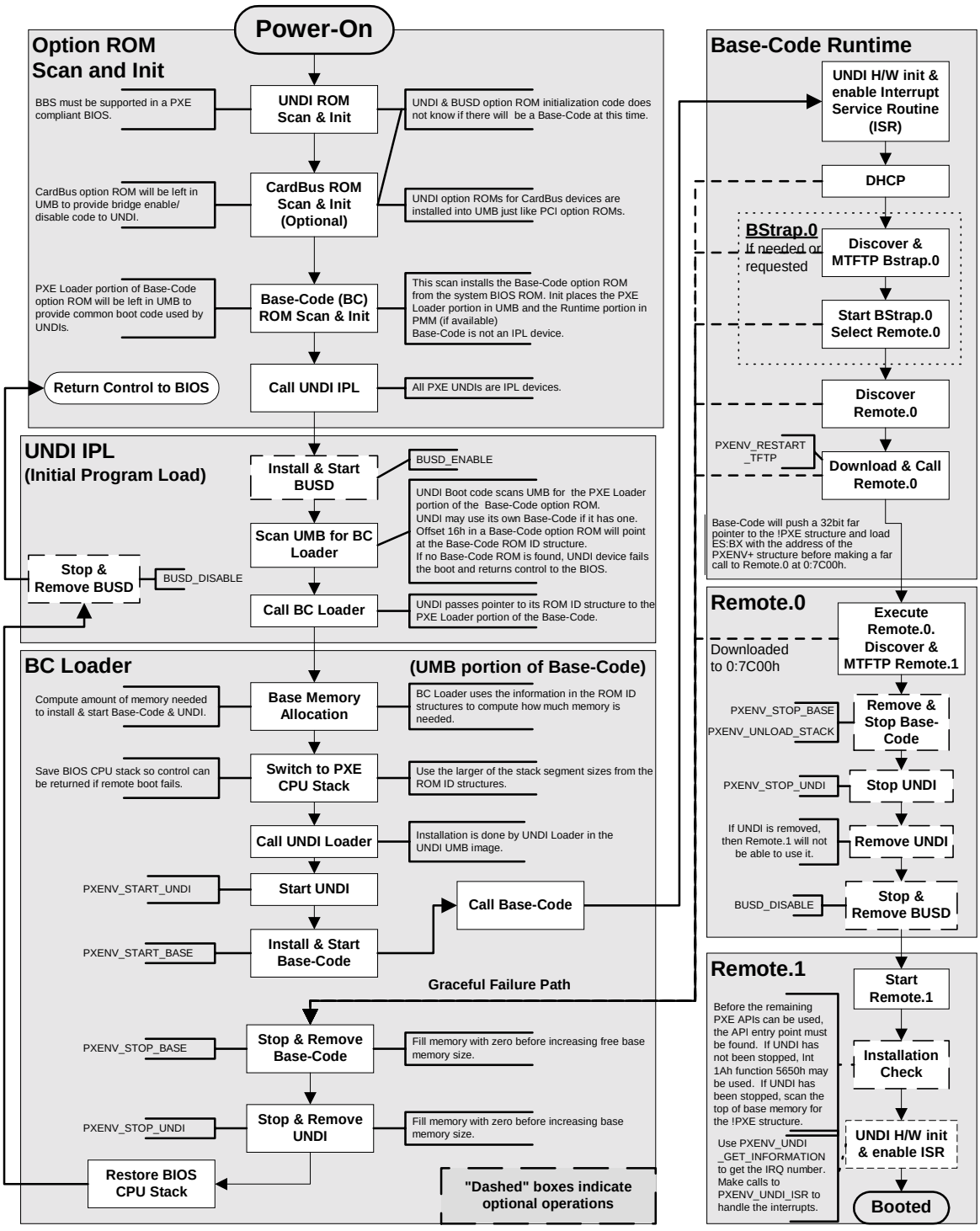

Figure 4-3 PXE IPL

4.4.1 Option ROM Scan and Initialization

Remembering that PXE option ROMs require the video subsystem to be initialized first, the system memory map is shown in Table 4-2 at this point.

Table 4-2 Memory Map after video initialization

Base Memory Address	Length	Description
0h	400h	Interrupt vector table
400h	100h	System BIOS data segment (FBM is in number of K and stored in a 16-bit word in the BIOS data segment at 40:13h)
500h	Variable	Free base memory (portions may be used by system BIOS)
9F800h	Variable	Extended BIOS data area (Not all BIOSes have this. Those that do, have 1K or 2K of data)
Upper Memory Address	**Length**	**Description**
A0000h	20000h	Video RAM (typical)
C0000h	8000h	Video ROM (typical)
C8000h	Variable	Option ROMs and upper memory
E0000h	10000h	System BIOS (Some BIOSes have 128K of code/data)
F0000h	10000h	System BIOS
Extended Memory Address	**Length**	**Description**
100000h	Variable	Free extended memory (portions may be used by system BIOS)

4.4.1.1 UNDI ROM Scan & Init

The BIOS must call the UNDI option ROM initialization routine (offset 03h of the ROM) after the ROM has been discovered in, or transferred to, upper memory (UMB). The BIOS must pass initialization parameters (AX, BX, DX, ES:DI) describing the location of the device in the host system. Refer to [BBS], [PnP] & [PCI] for detailed information on these parameter registers.

The following flowchart provides a guideline for writing an UNDI option ROM initialization routine. It deals with how the initialization routine should try to detect installed BC runtime images and which portions of the UNDI should be removed from upper memory before returning control to the system BIOS.

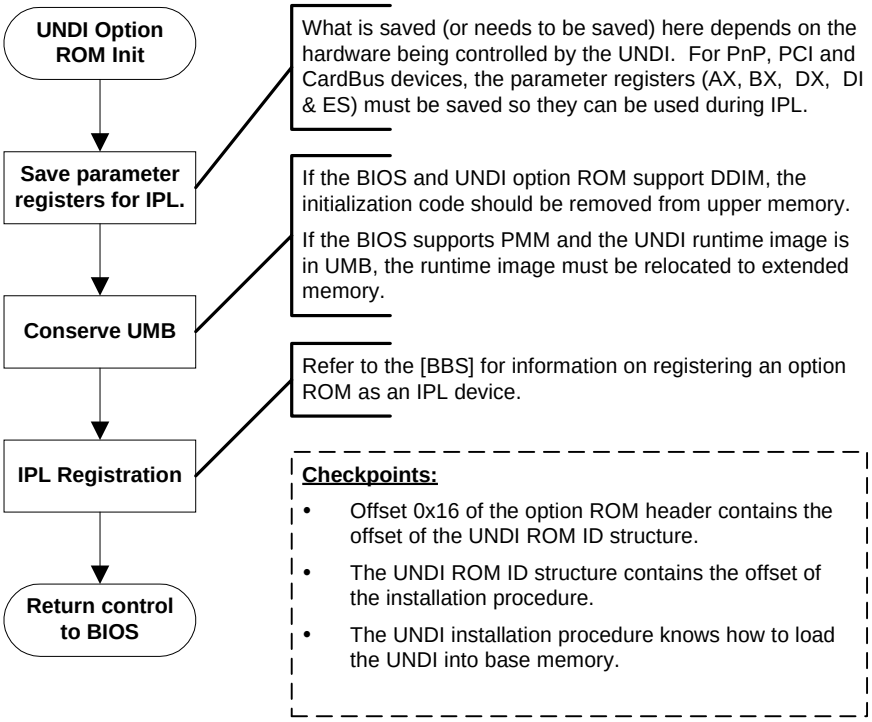

Figure 4-4 UNDI Option ROM Initialization

Table 4-3 Memory Map after UNDI ROM Transferred to UMB from BIOS ROM

Base Memory Address	Length	Description
0h	400h	Interrupt vector table
400h	100h	System BIOS data segment (FBM is in number of K and stored in a 16-bit word at 04:13h)
500h	Variable	Free base memory (portions may be used by system BIOS)
9F800h	Variable	Extended BIOS data area (Not all BIOSes have this. Those that do, have 1K or 2K of data)
Upper Memory Address	**Length**	**Description**
A0000h	20000h	Video RAM (typical)
C0000h	8000h	Video ROM (typical)
C8000h	4000h	UNDI ROM (Init, IPL, Loader, Driver)
CC000h	Variable	Option ROMs and upper memory
E0000h	10000h	System BIOS (Some BIOSes have 128K of code/data)
F0000h	10000h	System BIOS
Extended Memory Address	**Length**	**Description**
100000h	Variable	Free extended memory (portions may be used by system BIOS)

Table 4-4 Memory Map after UNDI ROM Initialized

Base Memory Address	Length	Description
0h	400h	Interrupt vector table
400h	100h	System BIOS data segment (FBM is in number of K and stored in a 16-bit word at 04:13h)
500h	Variable	Free base memory (portions may be used by system BIOS)
9F800h	Variable	Extended BIOS data area (Not all BIOSes have this. Those that do, have 1K or 2K of data)
Upper Memory Address	**Length**	**Description**
A0000h	20000h	Video RAM (typical)
C0000h	8000h	Video ROM (typical)
C8000h	800h	UNDI ROM (IPL, Loader)
C8800h	Variable	Option ROMs and upper memory
E0000h	10000h	System BIOS (Some BIOSes have 128K of code/data)
F0000h	10000h	System BIOS
Extended Memory Address	**Length**	**Description**
PMM+x	4000h	UNDI (Driver)
100000h	Variable	Free extended memory (portions may be used by system BIOS)

4.4.1.2 CardBus ROM Scan & Init

During CardBus option ROM scan, all CardBus bridges are configured and initialized. After the bridges are configured, boot devices with UNDI option ROMs are discovered and initialized. These UNDI ROMs should be installed into upper memory and initialized. If no CardBus devices with UNDI ROMs are discovered during the scan, the CardBus bridge configuration must be returned to its normal BIOS initialization state.

If the Host System has native support for the BUSD devices and at least one device is discovered, a BUSD ROM ID structure must be placed in UMB space to publish the BUSD EntryPoint address. The BUSD ROM ID structure is only needed if the bus containing the network devices needs to be enabled for the UNDI (and/or base-code) to remote boot. If the UNDI and/or base-code can not find the BUSD ROM ID structure after scanning the UMB, it should assume the bus is enabled by the system BIOS. Note that in this case the BUSD ROM APIs (BUSD_ENABLE/BUSD_DISABLE) are unavailable.

If the Host System utilizes a BUSD driver to initialize and discover BUSD PXE capable devices, the BUSD driver may be copied to UMB space. The Host System will call the Initialization Vector at offset [3] to perform initialization and discovery of devices. Following successful discovery of at least one device, the BUSD initialization code should be discarded, leaving the BUSD ROM ID structure with BUSD API entry point in UMB space.

Table 4-5 Memory Map after BUSD ROM Transferred to UMB from BIOS ROM

Base Memory Address	Length	Description
0h	400h	Interrupt vector table
400h	100h	System BIOS data segment (FBM is in number of K and stored in a 16-bit word at 40:13h)
500h	Variable	Free base memory (portions may be used by system BIOS)
9F800h	Variable	Extended BIOS data area (Not all BIOSes have this. Those that do, have 1K or 2K of data)
Upper Memory Address	**Length**	**Description**
A0000h	20000h	Video RAM (typical)
C0000h	8000h	Video ROM (typical)
C8000h	800h	UNDI ROM (IPL, Loader)
C8800h	8000h	BUSD ROM (Initialize,API)
D0800h	Variable	Option ROMs and upper memory
E0000h	10000h	System BIOS (Some BIOSes have 128K of code/data)
F0000h	10000h	System BIOS
Extended Memory Address	**Length**	**Description**
PMM+x	4000h	UNDI (Driver)
100000h	Variable	Free extended memory (portions may be used by system BIOS)

Table 4-6 Memory Map after BUSD ROM Initialized

Base Memory Address	Length	Description
0h	400h	Interrupt vector table
400h	100h	System BIOS data segment (FBM is in number of K and stored in a 16-bit word at 04:13h)
500h	Variable	Free base memory (portions may be used by system BIOS)
9F800h	Variable	Extended BIOS data area (Not all BIOSes have this. Those that do, have 1K or 2K of data)
Upper Memory Address	**Length**	**Description**
A0000h	20000h	Video RAM (typical)
C0000h	8000h	Video ROM (typical)
C8800h	800h	BUSD ROM (API)
C9000h	Variable	Option ROMs and upper memory
E0000h	10000h	System BIOS (Some BIOSes have 128K of code/data)
F0000h	10000h	System BIOS
Extended Memory Address	**Length**	**Description**
PMM+x	4000h	UNDI (Driver)
100000h	Variable	Free extended memory (portions may be used by system BIOS)

4.4.1.3 BC ROM Scan & Init

PXE-compliant BIOSes, with built in NICs, contain an embedded BC option ROM in the BIOS ROM (similar to built in video and SCSI option ROMs). This BC option ROM is not associated with any

hardware. It is to be transferred to UMB and initialized. The BC option ROM requires DDIM support.

Table 4-7 Memory Map after BC ROM Transferred to UMB from BIOS ROM

Base Memory Address	Length	Description
0h	400h	Interrupt vector table
400h	100h	System BIOS data segment (FBM is in number of K and stored in a 16-bit word at 40:13h)
500h	Variable	Free base memory (portions may be used by system BIOS)
9F800h	Variable	Extended BIOS data area (Not all BIOSes have this. Those that do, have 1K or 2K of data)
Upper Memory Address	**Length**	**Description**
A0000h	20000h	Video RAM (typical)
C0000h	8000h	Video ROM (typical)
C8000h	800h	UNDI ROM (IPL, Loader)
C8800h	800h	BUSD ROM (API)
C9000h	8000h	BC ROM (Initialize, Loader, Runtime)
D1000h	Variable	Option ROMs and upper memory
E0000h	10000h	System BIOS (Some BIOSes have 128K of code/data)
F0000h	10000h	System BIOS
Extended Memory Address	**Length**	**Description**
PMM+x	4000h	UNDI (Driver)
100000h	Variable	Free extended memory (portions may be used by system BIOS)

The flowchart in Figure 4-5 illustrates how a BC option ROM should initialize and install the BC runtime image and Loader code.

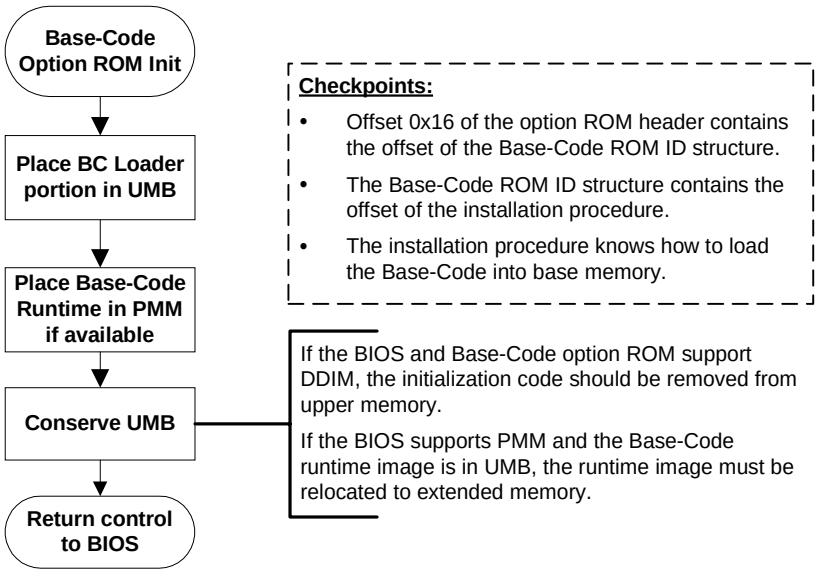

Figure 4-5 Base-Code Option ROM Initialization

Table 4-8 Memory Map after BC Option ROM Initialized

Base Memory Address	Length	Description
0h	400h	Interrupt vector table
400h	100h	System BIOS data segment (FBM is in number of K and stored in a 16-bit word at 04:13h)
500h	Variable	Free base memory (portions may be used by system BIOS)
9F800h	Variable	Extended BIOS data area (Not all BIOSes have this. Those that do, have 1K or 2K of data)
Upper Memory Address	**Length**	**Description**
A0000h	20000h	Video RAM (typical)
C0000h	8000h	Video ROM (typical)
C8000h	800h	UNDI ROM (IPL, Loader)
C8800h	800h	BUSD ROM (API)
C9000h	800h	BC ROM (Loader)
C9800h	Variable	Option ROMs and upper memory
E0000h	10000h	System BIOS (Some BIOSes have 128K of code/data)
F0000h	10000h	System BIOS
Extended Memory Address	**Length**	**Description**
PMM+x	4000h	UNDI (Driver)
PMM+y	8000h	BC (Runtime)
100000h	Variable	Free extended memory (portions may be used by system BIOS)

4.4.1.4 IPL Selection

Please refer to [BBS], [PnP] and [PCI] specifications for information on IPL selection.

4.4.2 UNDI Initial Program Load (IPL)

4.4.2.1 Scan UMB for BC Loader Routine

When an UNDI option ROM is selected as a Boot device, and it does not have a BC runtime of its own, it scans UMB for a Base-Code option ROM. Checking the 16-bit word at offset 16h of valid option ROMs does this. In BC option ROM this 16-bit word is an offset to the BC ROM ID structure.

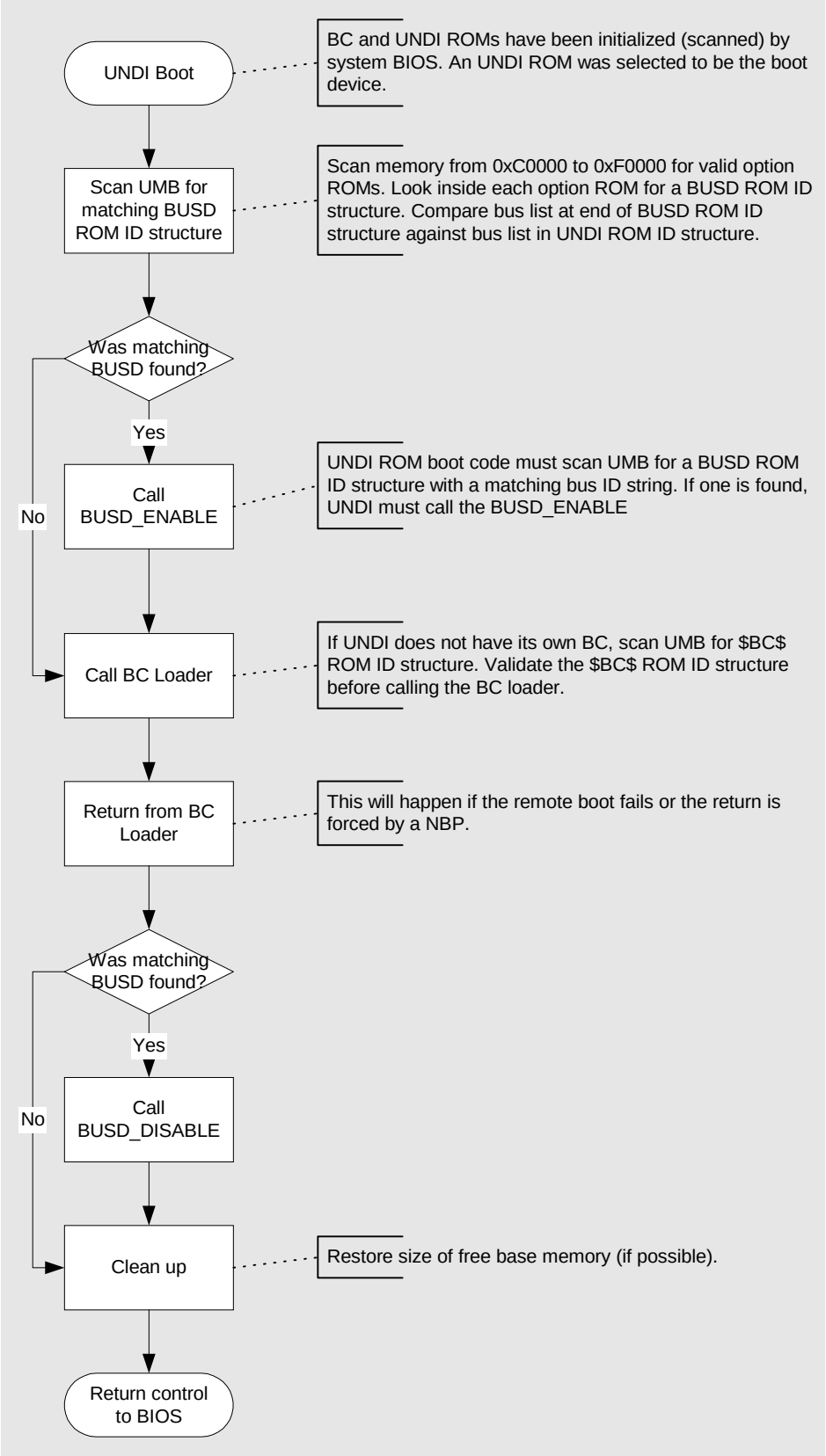

Figure 4-6 UNDI Option ROM Boot

4.4.2.2 Enable BUSD

If the IPL device requires BUSD support, it scans UMB for the BUSD ROM ID Structure and calls the Enable functionality.

4.4.2.3 Call BC Loader Routine

When a valid BC ROM ID structure is found, the UNDI IPL routine will push the segment:offset address of its ROM ID structure and the initialization parameter registers onto the stack and make a far call to the BC loader routine.

4.4.2.4 Disable BUSD

If the IPL device requires BUSD support, it scans UMB for the BUSD ROM ID Structure and calls the Disable functionality.

4.4.3 BC Loader Routine

The BC ROM ID structure is in the Base-Code option ROM in UMB. The offset of this structure is stored in the 16-bit word at offset 0x16 of the BC option ROM header. Before information in this structure is used by the UNDI IPL routine, the signature, length, revision and checksum must be validated.

4.4.3.1 Base Memory Allocation

Using the runtime size information from the BC and UNDI ROM ID structures, the BC loader routine computes how much memory is needed for the runtime code, data and stack segments. It then allocates enough free base memory by reducing the free base memory size (FBMS). (FBMS is a 16-bit word, at 40:13h, in the BIOS data segment, that contains the amount of free base memory in Kbytes.)

4.4.3.2 Switch to Runtime CPU Stack

When base memory is allocated and partitioned, BC switches to the runtime CPU stack, saving the location of the BIOS CPU stack.

4.4.3.3 Install & Start UNDI Driver

When the runtime CPU stack is ready, the segment addresses of the UNDI code and data segments are pushed onto the stack and the UNDI loader routine is called. If UNDI load completes successfully (AX == zero == success), PXENV_START_UNDI is called.

4.4.3.4 Install & Start BC

After UNDI is loaded and started, the Base-Code loader routine loads the Base-Code code and data segments and calls PXENV_START_BASE.

4.4.4 BC Runtime

Table 4-9 Memory Map after PXE BC Runtime Loaded

Base Memory Address	Length	Description
0h	400h	Interrupt vector table
400h	100h	System BIOS data segment (FBM is in number of K and stored in a 16-bit word at 04:13h)
500h	Variable	Free base memory (portions may be used by system BIOS)
8D000h	800h	PXE CPU Stack
8D800h	4000h	BC Data Segment
91800h	6000h	BC Code Segment
97800h	4000h	UNDI Data Segment
9B800h	4000h	UNDI Code Segment
9F800h	Variable	Extended BIOS data area (Not all BIOSes have this. Those that do, have 1K or 2K of data)
Upper Memory Address	**Length**	**Description**
A0000h	20000h	Video RAM (typical)
C0000h	8000h	Video ROM (typical)
C8000h	800h	UNDI ROM (IPL, Loader)
C8800h	800h	Base-Code ROM (Loader)
C9000h	Variable	Option ROMs and upper memory
E0000h	10000h	System BIOS (Some BIOSes have 128K of code/data)
F0000h	10000h	System BIOS
Extended Memory Address	**Length**	**Description**
PMM+x	4000h	UNDI (Driver)
PMM+y	8000h	BC (Runtime)
100000h	Variable	Free extended memory (portions may be used by system BIOS)

4.4.5 Client State at Bootstrap Execution Time (Remote.0)

The entire remote boot NBP is downloaded into base memory starting at location 0:7C00h. The PXE ROM code must then transfer control to the NBP by executing a far call to the beginning of the NBP. On entry to the NBP:

- CS:IP must contain the value 0:7C00h.
- ES:BX must contain the address of the PXENV+ structure.
- SS:[SP+4] must contain the segment:offset address of the !PXE structure.
- EDX is no longer used.
- SS:SP is to contain the address of the beginning of the unused portion of the PXE services stack.
- There must be at least 1.5KB of free stack space for the NBP.

The NBP may abort the network boot by returning control to the boot ROM. The boot ROM may remove all, some, or none of the PXE modules that have been loaded into base memory.

Listed below are three scenarios for API availability after the network boot is aborted. In each case, control is passed back to the BIOS which attempts to boot the next device in the BIOS boot order list. The next bootstrap program may make use of whichever PXE APIs have been left in memory.

The three possibilities for PXE API combinations left in memory are:

- Base-Code and UNDI are instructed to be removed from base memory. In this case, the NBP has returned AX == PXENV_STATUS_SUCCESS.
- Base-Code is instructed to be removed, but UNDI is to be kept in base memory. In this case, the NBP returned AX == PXENV_STATUS_KEEP_UNDI.
- Base-Code and UNDI are instructed to be kept in base memory. In this case, the NBP returned AX == PXENV_STATUS_KEEP_ALL.
- Any other value in AX is the same as PXENV_STATUS_SUCCESS.

The initial Network Bootstrap Program (NBP) size should not exceed 32KB. While it is possible to download a larger program as the initial NBP, the 32K size limit provides the advantage of being able to clean up and fail gracefully if it is determined the system is not adequate for the intended application or OS image. For example, when remote booting an operating system, it is strongly recommended that the load be broken into at least two files, where the first executable is not larger than 32K. It should be the purpose of this executable to determine whether the client system has adequate resources to successfully boot the OS in question. If not, this executable should fail the boot.

This memory map shows the NBP “Remote.0” being loaded into memory (and some memory being reserved for a subsequent remote DOS boot [“Remote.1”]).

Table 4-10 Memory Map after REMOTE.0 Downloaded

Base Memory Address	Length	Description
0h	400h	Interrupt vector table
400h	100h	System BIOS data segment (FBM is in number of K and stored in a 16-bit word at 04:13h)
500h	7700h	Reserved
7C00h	8400h	REMOTE.0 (code & data)
10000h	Variable	Free base memory (portions may be used by system BIOS)
8D000h	800h	PXE CPU Stack
8D800h	4000h	BC Data Segment
91800h	6000h	BC Code Segment
97800h	4000h	UNDI Data Segment
9B800h	4000h	UNDI Code Segment
9F800h	Variable	Extended BIOS data area (Not all BIOSes have this. Those that do, have 1K or 2K of data)
Upper Memory Address	**Length**	**Description**
A0000h	20000h	Video RAM (typical)
C0000h	8000h	Video ROM (typical)
C8000h	800h	UNDI ROM (IPL, Loader)
C8800h	800h	BUSD ROM (API)
C9000h	800h	BC ROM (Loader)
C9800h	Variable	Option ROMs and upper memory
E0000h	10000h	System BIOS (Some BIOSes have 128K of code/data)
F0000h	10000h	System BIOS
Extended Memory Address	**Length**	**Description**
PMM+x	4000h	UNDI (Driver)
PMM+y	8000h	Base-Code (Runtime)
100000h	Variable	Free extended memory (portions may be used by system BIOS)

4.4.6 Client State at Bootstrap Execution Time (Remote.1)

Table 4-11 Memory Map after REMOTE.1 Downloaded

Base Memory Address	Length	Description
0h	400h	Interrupt vector table
400h	100h	System BIOS data segment (FBM is in number of K and stored in a 16-bit word at 04:13h)
500h	7700h	Reserved
7C00h	8400h	REMOTE.0 (code & data)
10000h	Variable	Free base memory (portions may be used by system BIOS)
8D000h	800h	PXE CPU Stack
8D800h	4000h	BC Data Segment
91800h	6000h	BC Code Segment
97800h	4000h	UNDI Data Segment
9B800h	4000h	UNDI Code Segment
9F800h	Variable	Extended BIOS data area (Not all BIOSes have this. Those that do, have 1K or 2K of data)
Upper Memory Address	**Length**	**Description**
A0000h	20000h	Video RAM (typical)
C0000h	8000h	Video ROM (typical)
C8000h	800h	UNDI ROM (IPL, Loader)
C8800h	800h	BUSD ROM (API)
C900h	800h	BC ROM (Loader)
C9800h	Variable	Option ROMs and upper memory
E0000h	10000h	System BIOS (Some BIOSes have 128K of code/data)
F0000h	10000h	System BIOS
Extended Memory Address	**Length**	**Description**
PMM+x	4000h	UNDI (Driver)
PMM+y	8000h	BC (Runtime)
100000h	Variable	Free extended memory (portions may be used by system BIOS)
E98000h	168000h	REMOTE.1 (DOS boot diskette image)

Free Base Memory (FBM) Size (BIOS Data Area). When execution of the downloaded bootstrap begins, the 16-bit word at memory address 40:13h must contain the amount of free base memory in KB.

PXE CPU Stack. When execution of the downloaded bootstrap is begun, SS:SP is to contain the address of the top of the unused portion of the PXE services CPU stack. The downloaded NBP should not modify the used portion of the PXE services CPU stack prior to the time in the boot sequence when it is certain that the PXE services will not be needed again.

Base and UNDI Code and Data Segments. This memory area is reserved for the code and data that implement the PXE services. These locations should not be modified by the downloaded NBP prior to the time in the boot sequence when it is certain that the PXE services will not be needed again.

Extended BIOS Data. If EBDA has been allocated, the downloaded NBP should not modify memory in the EBDA.

Table 4-12 Memory Map after REMOTE.1 Started

Base Memory Address	Length	Description
0h	400h	Interrupt vector table
400h	100h	System BIOS data segment (FBM is in number of K and stored in a 16-bit word at 04:13h)
500h	97300h	DOS does memory
97800h	4000h	UNDI Data Segment
9B800h	4000h	UNDI Code Segment
9F800h	Variable	Extended BIOS data area (Not all BIOSes have this. Those that do, have 1K or 2K of data)
Upper Memory Address	**Length**	**Description**
A0000h	20000h	Video RAM (typical)
C0000h	8000h	Video ROM (typical)
C8000h	800h	UNDI ROM (IPL, Loader)
C8800h	800h	Base-Code ROM (Loader)
C9000h	Variable	Option ROMs and upper memory
E0000h	10000h	System BIOS (Some BIOSes have 128K of code/data)
F0000h	10000h	System BIOS
Extended Memory Address	**Length**	**Description**
100000h	Variable	Free extended memory
E98000h	168000h	REMOTE.1 (DOS boot diskette image)

4.4.6.1 Stop & Remove BC Runtime

If control is returned from PXENV_START_BASE, PXENV_STOP_BASE is called. BC runtime can be removed from memory if an exit code of PXENV_EXIT_SUCCESS (in AX) is returned and there is not a status code of (PXENV_STATUS_KEEP) in the parameter structure.

4.4.6.2 Stop & Remove UNDI

If BC runtime has been stopped, PXENV_STOP_UNDI is called. UNDI can be removed from memory if an exit code of PXENV_EXIT_SUCCESS (in AX) returned and there is not a status code of (PXENV_STATUS_KEEP) in the parameter structure.

4.4.6.3 Restore BIOS CPU Stack

Restore the original BIOS CPU stack and increase the size of free base memory.

4.5 Requirements on individual PXE participants

4.5.1 UNDI Option ROM

An UNDI Option ROM image is required for each boot device. This Option ROM image may be kept on a boot device's local storage (for example, FLASH on a NIC) or built in to the Host System BIOS.

4.5.1.1 UNDI ROM ID Structure

The UNDI ROM ID structure (shown below) must be contained within the memory defined by the Option ROM header. This structure provides information about the UNDI driver revision, memory requirements and the address of the UNDI loader. This structure must be static.

Table 4-13 UNDI ROM ID Structure

Offset	*Type(bytes)*	*Name*	*Description*
0x00	UINT8(0x04)	Signature	'UNDI'
0x04	UINT8	StructLength	Length of this structure in bytes. (0x12 + 0x04 * #bus)
0x05	UINT8	StructCksum	Used to make structure byte checksum equal zero.
0x06	UINT8	StructRev	Revision of this structure is zero. (0x00)
0x07	UINT8(0x03)	UNDIRev	UNDI API revision number implemented in the driver. The least significant byte of the revision number is stored in the first byte of the field. For UNDI revision 2.1.0, this field contains (0x00, 0x01, 0x02).
0x0A	UINT16	UNDILoader	Offset of the UNDI loader in this option ROM. See UNDI loader description below for more details. This is also the UNDI loader routine entry point.
0x0C	UINT16	StackSize	Minimum stack segment size, in bytes, needed for UNDI driver operation.
0x0E	UINT16	DataSize	Minimum data segment size, in bytes, needed for UNDI driver operation.
0x10	UINT16	CodeSize	Minimum code segment size, in bytes, needed for UNDI driver operation.
0x12	UINT8(0x04 * bus-count)	BusType	Type of bus the UNDI driver is written for. There may be more than one bus type in this field. 'ISAR', 'EISA', 'VESA', 'PCCR' 'PCCR' = PC CardBus

4.5.1.2 UNDI Initialization Routine

An UNDI Option ROM must contain an UNDI initialization routine. The UNDI initialization routine must be contained within the memory defined by the Option ROM header.

The UNDI initialization routine must perform/provide the following:

- Contain a $PnP expansion header structure.
- Contain an UNDI ROM ID structure. Offset 16h of the option ROM header contains the 16-bit offset of the UNDI ROM ID structure.
- Contain an UNDI loader routine.
- Save the initialization parameters (AX, BX, DX, ES:DI) from the BIOS. These parameters will be used by the UNDI loader routine, UNDI IPL routine and UNDI driver.
- Register the UNDI IPL routine with the Host System BIOS.

The UNDI initialization routine should:

- Conserve as much UMB as possible by removing the initialization routine and using PMM, if available/needed to store the UNDI driver.
- Verify that the initialization parameters identify the correct boot device.

4.5.1.3 UNDI Loader Routine

The UNDI loader routine must be contained within the memory defined by the Option ROM header.

The UNDI loader may be called by the PXE loader or Host System BIOS.

The UNDI loader must perform/provide the following:

- Install the UNDI code and data images into base memory.
- Create and/or fill the !PXE structure in the UNDI code segment.
- Set the !PXE structure checksum to zero.
- Fill in the status field in the UNDI Loader parameter structure. (See PXENV_STATUS_LOADER_xxx #defines)
- Set the exit code in AX to PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE.

The BC (Base-Code) loader or Host System BIOS may call the UNDI loader. The UNDI loader routine requires:

- The caller to call the BUSD enable API, if needed.
- At least UNDIROMID->StackSize bytes of CPU stack space.
- At least UNDIROMID->DataSize bytes of base memory for the UNDI data segment.
- At least UNDIROMID->CodeSize bytes of base memory for the UNDI code segment.
- A filled-in UNDI Loader parameter structure.
- The caller push a 32-bit far pointer to the UNDI Loader parameter structure onto the CPU stack.
- The caller make a far call to the UNDI loader entry point.
- The caller clean up the CPU stack after control is returned from the UNDI loader routine.
- A check of the exit code returned in AX.

UNDI Loader Parameter Structure

`Typedef struct s_UNDI_LOADER {`
`  PXENV_STATUS Status;`
`  UINT16 AX;`
`  UINT16 BX;`
`  UINT16 DX;`
`  UINT16 DI;`
`  UINT16 ES;`
`  UINT16 UNDI_DS;`
`  UINT16 UNDI_CS;`
`  SEGOFF16 PXEptr;`
`  SEGOFF16 PXENVptr;}`
`t_UNDI_LOADER;`
	Pass/Fail status: See PXENV_STATUS_xxx #defines
	In: AX passed to UNDI initialization routine
	In: BX passed to UNDI initialization routine
	In: DX passed to UNDI initialization routine
	In: DI passed to UNDI initialization routine
	In: ES passed to UNDI initialization routine
	In: Address of UNDI data segment to fill in
	In: Address of UNDI code segment to fill in
	Out: Far pointer to !PXE structure.
	Out: Far pointer to PXENV+ structure.

4.5.1.4 UNDI IPL Routine

The UNDI IPL routine may be included within the memory defined by the Option ROM header.

The UNDI IPL routine must perform the following:

- Scan UMB for a Base-Code option ROM.
- Verify the BC ROM ID structure.
- These BUSD-related items are only performed if needed:
  - Scan UMB for a BUSD option ROM.
  - Verify the BUSD ROM ID structure.
  - Create a BUSD_ENABLE parameter structure.
  - Push a 32-bit far pointer to the BUSD parameter structure.
  - Push a 16-bit constant: PXENV_BUSD_ENABLE.
  - Make a far call to the BUSD API entry point.
  - Clean up the CPU stack (pop 6 bytes) after control is returned from BUSD.
- Create a BC Loader parameter structure.
- Push a 32-bit far pointer to the BC Loader parameter structure.
- Make a far call to the BC loader routine.

- Clean up the CPU stack after control is returned from the BC loader routine.
- Return control to the Host System BIOS.

4.5.1.5 UNDI Driver

The UNDI driver may be included within the memory defined by the option ROM header. An UNDI driver must be able to:

- Operate in real mode.
- Operate in 16:16 protected mode with a 16-bit (SP) stack segment.
- Operate in 16:16 protected mode with a 32-bit (ESP) stack segment.

An UNDI driver must:

- Always set AX and Status field for all UNDI APIs on success or failure.
- Call the Base-Code API entry point for all non-UNDI APIs.
  - Before making this far call, the UNDI driver must push the following parameters (in this order) onto the stack:
    - !PXE segment (16-bit)
    - !PXE offset (16-bit)
    - Param segment (16-bit)
    - Param offset (16-bit)
    - Function (16-bit)
- Clean up the stack.
- Return Base-Code AX/Status without changes to caller.

UNDI driver API specifications, parameter passing and status/result codes are covered in other sections in this document.

4.5.1.6 UNDI Driver Carrying its own Base Code

A PXE implementation provided on a NIC that carries its own Base Code may use it in lieu of the Base Code provided by the system BIOS. However, it is recommended that NIC PXEs use the Base Code provide by the system BIOS if it is available, even if the on-NIC PXE implementation includes a Base Code. The NIC implementation may only provide its Base Code for itself, and may not replace the Base Code provided by the system BIOS for other network interface devices. If a NIC provides its own Base Code, the NIC must provide a user-settable mechanism to disable the Base Code on the NIC.

4.5.2 BUSD Option ROM

Unlike UNDI and BC Option ROMs, BUSD Option ROMs are not a drop-in component. BUSD ROMs need to interact with Host System BIOS during POST. Bridge components and boot devices need to be configured during option ROM scan and disabled after option ROMs are transferred to UMB. How information is transferred between the BUSD ROM and the Host System BIOS is implementation dependent.

Note: If a BIOS uses an UNDI driver early in the POST process that requires bridge components to be enabled, the BIOS must initialize the bridge components.

4.5.2.1 BUSD ROM ID Structure

The BUSD ROM ID structure (shown below) must be contained within the memory defined by the option ROM header. This ROM ID structure is used by the UNDI IPL routine and contains information about the revision, memory requirements and entry point of the BUSD API.

Table 4-14 BUSD ROM ID Structure

Offset	*Type (bytes)*	*Name*	*Description*
0x00	UINT8(0x04)	Signature	'BUSD'
0x04	UINT8	StructLength	Length of this structure in bytes. (0x0E + 0x04 * #bus)
0x05	UINT8	StructCksum	Used to make structure byte checksum equal zero.
0x06	UINT8	StructRev	Revision of this structure is zero. (0x00)
0x07	UINT8(0x03)	BUSDRev	BUSD API revision number implemented in the driver. The least significant byte of the revision number is stored in the first byte of the field. For BUSD revision 2.1.0, this field contains (0x00, 0x01, 0x02).
0x0A	UINT16	EntryPoint	Offset of the BUSD API entry point. **Note:** This is not the same as the PXE runtime API entry point
0x0C	UINT16	StackSize	Minimum stack segment size, in bytes, needed for API operation.
0x0E	UINT8(0x04 * bus-count)	Bus Type	Type of bus supported by this BUSD option ROM. There may be more than one bus type included in this field. 'PCCR' = CardBus

4.5.2.2 BUSD Initialization Routine

Here are three examples of BUSD initialization routines.

1. The initialization routine only leaves a BUSD option ROM image in UMB. All of the bridge configuration and device detection code is built into the BIOS.
2. The initialization routine configures and initializes the bridge components. This configuration information needs to be communicated back to the BIOS. Code to scan for devices with UNDI option ROM support across these bridges is built into the BIOS.
3. The initialization routine configures and initializes the bridge components and scans for devices with UNDI option ROM support across these bridges. The configuration and detected device information needs to be communicated back to the BIOS.

BUSD initialization routines should try to conserve as much UMB as possible. This can be done by removing initialization code after use and storing any configuration information in PMM allocated extended memory.

4.5.2.3 BUSD APIs

BUSD API Entry Point

The BUSD API entry point is located in the BUSD option ROM image in UMB. Using the segment address of the BUSD Option ROM image creates the address of this entry point and the 16-bit offset supplied in the BUSD ROM ID structure.

The BUSD API entry point should only be used by an UNDI IPL routine to initialize bridge components, or by an NBP that is going to start an OS that needs to have the bridge components disabled.

The UNDI IPL routine would call BUSD enable before scanning for and calling the BC loader. If the remote boot is canceled, control is returned to UNDI IPL and BUSD disable is called before returning control to the BIOS IPL selection routine.

If needed, an NBP would call BUSD disable before starting a downloaded OS image.

BUSD API Functions

If an invalid BUSD API op-code is given to the BUSD API entry point, a status of PXENV_STATUS_BAD_FUNC will be returned.

If invalid register contents are passed to the BUSD API, a status of PXENV_STATUS_FAILURE will be returned.

If the UNDI ROM ID structure is not present or valid, a status of PXENV_STATUS_LOADER_NO_UNDI_ROMID or PXENV_STATUS_LOADER_BAD_UNDI_ROMID will be returned.

BUSD Disable

Op-Code:	BUSD_DISABLE (0000h)
Input:	Far pointer to a t_BUSD_DISABLE parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This service is used to disable a configured bridge. This is done byte the UNDI IPL before returning control to the BIOS IPL selection routine when a remote boot has been aborted. NBPs that need to enable a driver that expects to have the bridge disabled can also do this.

```
Typedef struct s_BUSD_DISABLE {
    PXENV_STATUS Status;
    UINT16 AX;
    UINT16 BX;
    UINT16 DX;
    UINT16 DI;
    UINT16 ES;
    SEGOFF16 UNDI_ROMID;
} t_BUSD_DISABLE;
```

Set before calling API service

AX, BX, DX, DI, ES: Register values passed by the BIOS to the UNDI option ROM initialization entry point.

UNDI_ROMID: Segment and offset of the UNDI ROMID structure.

Returned from API service

Status: PXENV_STATUS_SUCCESS or one of the PXE error codes.

BUSD Enable

Op-Code:	BUSD_ENABLE (0001h)
Input:	Far pointer to a t_PXENV_BUSD_ENABLE parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE must be returned in AX. The status field in the parameter structure must be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This service is used by the UNDI_IPL to reconfigure a bridge at the beginning of the UNDI IPL. The UNDI loader routine and the UNDI driver expect the bridge to be configured before they are called.

```
Typedef struct s_BUSD_ENABLE {
    PXENV_STATUS Status;
    UINT16 AX;
    UINT16 BX;
    UINT16 DX;
    UINT16 DI;
    UINT16 ES;
    SEGOFF16 UNDI_ROMID;
    } t_BUSD_ENABLE;
```

Set before calling API service

AX, BX, DX, DI, ES: Register values passed by the BIOS to the UNDI option ROM initialization entry point.

UNDI_ROMID: Segment and offset of the UNDI ROMID structure.

Returned from API service

Status: `PXENV_STATUS_SUCCESS or one of the PXE error codes.`

4.5.3 Base-Code (BC) Option ROM

A Base-Code Option ROM must be supplied with PXE-compliant BIOSes in machines that have NIC(s) built into the motherboard. It is recommended that the Base-Code be supplied with machines supporting transient NICs without the ability to carry the Base-Code themselves.

When a PXE Base-Code (BC) Option ROM is included in the BIOS, it must be scanned/transferred into UMB and initialized/called before IPL. After initialization, the option ROM image left in UMB must contain a valid BC ROM ID structure and the BC loader. The actual BC runtime image may be relocated to [PMM] allocated extended memory or reside in some other local storage.

4.5.3.1 Base-Code ROM ID Structure

The Base-Code ROM ID structure (shown below) provides information about the BC runtime revision, memory requirements and the address of the BC loader. The BC ROM ID structure must be present before IPL.

Table 4-15 BC ROM ID Structure

Offset	Type(bytes)	Name	Description
0x00	UINT8(0x04)	Signature	'BC'
0x04	UINT8	StructLength	Length of this structure in bytes. (0x12)
0x05	UINT8	StructCksum	Used to make structure byte checksum equal zero.
0x06	UINT8	StructRev	Revision of this structure is zero. (0x00)
0x07	UINT8(0x03)	BC_Rev	BC API revision number implemented in the driver. The least significant byte of the revision number is stored in the first byte of the field. For BC revision 2.1.0, this field contains (0x00, 0x01, 0x02).
0x0A	UINT16	BC_Loader	Offset of the BC loader routine in this option ROM. This routine will be called by UNDI IPL routines.
0x0C	UINT16	StackSize	Minimum stack segment size, in bytes, needed for BC runtime operation.
0x0E	UINT16	DataSize	Minimum data segment size, in bytes, needed for BC runtime operation.
0x10	UINT16	CodeSize	Minimum code segment size, in bytes, needed for BC runtime operation.

4.5.3.2 Base-Code Initialization Routine

The Base-Code initialization routine is an optional component. If implemented, it must be contained within the memory defined by the option ROM header.

The BC initialization routine must:

- Not register the BC option ROM as an IPL device.
- Compute amount of base memory needed for BC runtime and UNDI driver.
- Create and/or fill in the BC ROM ID structure, if needed. Offset 16h of the option ROM header contains the 16-bit offset of the BC ROM ID structure.
- Leave the BC loader routine in UMB.

The BC initialization routine should:

- Conserve as much UMB as possible by removing the initialization routine and using PMM, if available/needed to store the BC runtime.

4.5.3.3 Base-Code Loader Routine

The Base-Code loader routine must be present before IPL.

The Base-Code loader must:

- Verify the UNDI ROM ID structure contents and checksum.
- Allocate base memory for the BC runtime and UNDI driver code and data segments.
- Allocate base memory for a CPU stack, if needed.
- Create an UNDI loader parameter structure.
- Push a 32-bit far pointer to the UNDI loader parameter structure
- Make a far call to the UNDI loader entry point.
- Clean up the CPU stack after control is returned from the UNDI loader.
- Install the BC runtime code and data into base memory.
- Fill in the BC fields in the !PXE and PXENV+ structures in the UNDI code segment.
- Update the checksums in the !PXE and PXENV+ structures.

- Call the following PXE APIs, with the required parameters, in this order: PXENV_START_UNDI, PXENV_START_BASE, PXENV_STOP_BASE, PXENV_STOP_UNDI.
- Clean up, and release allocated base memory, if required.

The Base-Code loader requires:

- The UNDI IPL routine to make a far call to the BC loader routine passing a far pointer to the BC loader parameter structure.
- The UNDI IPL routine is also responsible for removing the parameters from the stack.

BC Loader Parameter Structure

`Typedef struct s_BC_LOADER {`
`  PXENV_STATUS Status;`
`  UINT16 AX;`
`  UINT16 BX;`
`  UINT16 DX;`
`  UINT16 DI;`
`  UINT16 ES;`
`  SEGOFF16 UNDI_ROMID;`
`} t_BC_LOADER;`
	
	`Pass/Fail status: See PXENV_STATUS_xxx #defines`
	`In: AX passed to UNDI initialization routine`
	`In: BX passed to UNDI initialization routine`
	`In: DX passed to UNDI initialization routine`
	`In: DI passed to UNDI initialization routine`
	`In: ES passed to UNDI initialization routine`
	`In: Offset of UNDI ROMID structure`

4.5.3.4 BC Runtime

The BC runtime must be capable of executing in real mode and 16:16 protected mode with a 16-bit (SP) stack segment. The BC runtime does not need to support a 32-bit (ESP) stack segment.

The BC runtime starts when the PXENV_START_BASE API is called from the BC loader. Control is not returned to the BC loader unless remote boot fails, at which time PXENV_STOP_BASE and PXENV_STOP_UNDI will be called.

The BC runtime must:

- Call the PXENV_UNDI_GET_INFORMATION API.
- Provide an ISR that calls PXENV_UNDI_ISR.
- Implement DHCP and TFTP client protocols, as defined in this specification.
- Provide the Preboot, UDP and TFTP APIs.

4.5.4 Network Bootstrap Program

Network bootstrap programs (NBPs) are binary images that will be downloaded by the BC runtime to 0:7C00h. The BC runtime will then make a far call to 0:7C00h after pushing a 32-bit far pointer to the !PXE structure onto the stack.

Note: For backwards compatibility the BC runtime must also load the address of the PXENV+ structure into the ES:BX register pair.

NBPs can use the PXE APIs to:

- Download other NBPs, applications, or OS images (Pre-boot, UDP and TFTP APIs)
- Communicate on the network (UNDI APIs)
- Shut down and stop the BC runtime and/or the UNDI driver. Once the BC and UNDI are stopped, one or both may be removed from base memory (UNDI cannot be removed if BC cannot be removed).

5. PXE BIOS Support

This section discusses host system BIOS support required for PXE compliance and how PXE boot devices (ROMs) and PXE Network Boot Programs (NBPs) use it.

5.1 BIOS Support

5.1.1 BIOS Requirements

PXE-compliant BIOS's implementations *must*:

- Locate and configure all PXE-capable boot devices (UNDI Option ROMs) in the system, both built-in and add-ins.
- Supply a PXE per this specification, if the system includes a built-in network device.
- Implement the following specifications:
  - *Plug-and-Play BIOS Specification v1.0a* or later.
  - *System Management BIOS (SMBIOS) Reference Specification v2.2* or later.
  - The requirements defined in Sections 3 and 4 of the *BIOS Boot Specification (BBS) v1.01* or later, to support network adapters as boot devices.
- Supply a valid UUID and Wake-up Source value for the system via the [SMBIOS] structure table.
  Note: An implementation might also choose to supply the UUID via the _SYSID_ structure (see Section 5.2.1) because [PC98] requires the interface.

5.1.2 BIOS Recommendations

PXE-compliant BIOSes *should* implement:

- PXE support for CardBus via the PXE BUSD option ROM if the host system supports CardBus.
  PXE support for non-industry standard boot devices, such as CardBus, requires Host System support for locating and initializing PXE Boot Devices and loading expansion/option ROMs for those devices.
- *POST Memory Manager Specification v1.01* or later

  PMM is strongly recommended. PMM provides a straightforward way for LAN on Motherboard PXE implementations to move their ROM image from UMB to extended memory. While methods to do this exist outside of PMM, their use is undefined and unreliable. Placing PXE ROM images into UMB space reduces the available UMB space by approximately 32 KB. This is sufficient to compromise or even prevent successful operation of some downloaded programs
- *Boot Integrity Services (BIS) API Specification v1.0* or later

5.2 PXE Support

5.2.1 UUID Support

PXE-compliant Boot ROMs *must* support UUID detection by reading table-based SMBIOS structures, see [SMBIOS] for details.

PXE-compliant Boot ROMs *should* support UUID detection for legacy system support by reading:

- Table-based _SYSID_ structures, see below for details.
- SMBIOS structures through the PnP function interface, see [SMBIOS] for details.

5.2.1.1 Reading table-based SYSID structures

The SYSID Entry Point structure, described below, can be located by application software by searching for the anchor string on paragraph (16-byte) boundaries within the physical memory address range 000E0000h to 000FFFFFh.

The UUID BIOS structure can be found by walking the list of SYSID BIOS structures in the SYSID BIOS Structure Table.

Table 5-1 Format of SYSID Entry Point Structure

Element	*Length*	*Description*
Header/Type	7 Bytes	_SYSID_
Checksum	1 Byte	Checksum of the SYSID BIOS Entry Point Structure
Length	2 Bytes	Total length of SYSID BIOS Structure Table (Set to 011h).
SYSID BIOS Structure Table Address	4 Bytes	32 bit physical address of the beginning of the SYSID BIOS Structure Table. ***This value is BYTE Aligned!!***
Number of SYSID BIOS Structures	2 Bytes	Total number of structures within the SYSID BIOS Structure Table.
SYSID BIOS Revision	1 Byte	Revision of the SYSID BIOS Extensions (Set to 00h).

Table 5-2 Format of the SYSID BIOS structures

Element	*Length*	*Description*
Header/Type	6 Bytes	_????_
Checksum	1 Byte	Checksum of the SYSID BIOS Structure
Length	2 Bytes	Total length of SYSID BIOS Structure
Variable Data Portion	?? Bytes	Depends on SYSID BIOS Structure Header/Type Field

Table 5-3 Format of the UUID BIOS structure

Element	*Length*	*Description*
Header/Type	6 Bytes	_UUID_
Checksum	1 Byte	Checksum of the UUID BIOS Structure
Length	2 Bytes	Total length of UUID BIOS Structure (Set to 0019h).
Variable Data Portion	16 Bytes	Actual UUID data (Initially set all bytes to 0FFh).

5.2.2 Remote Wake Up Source

5.2.2.1 Detection

PXE-compliant NBPs should implement two methods of Remote Wake-Up source detection to enable legacy system support.

- Reading table-based SMBIOS structures
- Reading SMBIOS structures through the PnP function interface

5.2.3 Bootstraps

PXE-compliant boot ROMs must support the PnP/BBS bootstrap mechanism. If the PXE implementation is to reside on a NIC, it should also support Int 18h and Int 19h bootstrap:

Note: Bootstrap interrupts 18h and 19h are mutually exclusive.

5.2.4 Memory Management

5.2.4.1 POST Memory Manager

If PMM is available, it must be used by PXE Option ROMs if using PMM will conserve UMB.

How PMM is detected and used is covered in the [PMM] specification. The PMM functions are only available prior to INT 19. This means that PMM is available to the Option ROM Initialization (Scan) process, but not available to the Option ROM Boot (IPL) process.

5.2.5 Boot Integrity Services

PXE Option ROMs must support Remote Boot Authentication if the Platform Boot Integrity Services are present in the Host system BIOS.

Remote boot authentication requires the BIOS to provide a set of platform security functions. Generally, the level of detail required to compile code that calls the interface is TBD, and may be deferred to header files distributed in a future Software Development Kit.

A detailed description of the platform security capabilities may be found in the [BIS] specification.