Wired for Management Baseline

Version 1.1a

Specification to help reduce TCO for business PCs.

August 28, 1997
Intel Corporation

Revision History

Date	Description
August 28, 1997	Version 1.1a. Updated references; added missing values for DHCP Options in Appendix A; revised code samples in Appendixes C, D, E, F, and G.

Contents

NEED FOR MANAGEMENT BASELINE AND OVERVIEW

Need To Lower Total Cost of Ownership 1
The Solution: Making PCs Universally Manageable and Universally Managed 2
Baseline Component Technologies 3
Instrumentation *3*
Service Boot *3*
Remote Wake-up *4*
Power Management *4*
Baseline Evolution and the Proposed Future Baseline Technologies 4
WfM Baseline and Other Industry Standards/Guidelines 5
ACPI *5*
DMI *5*
Network PC Design Guidelines *5*
SNMP *5*
WBEM *5*
How This Document Is Organized 6

INSTRUMENTATION

Instrumentation Overview 7
Baseline Overview 7
DMI-Based Instrumentation 7
Detailed Specification 7
DMI 2.0 Service Provider *8*
DMI Standard Groups *8*
DMI Event Generation *10*
DMI Management Interfaces *10*
DMI Service Provider and Instrumentation Deployment *11*
Guide to DMI Instrumentation Development *11*
Proposed Baseline Extensions 11
Programming Interfaces *11*
DMI Standard Groups *12*

REMOTE NEW SYSTEM SETUP

Remote Boot Overview 13
Baseline Overview 13
Server Deployment *14*
Client Deployment *14*
Detailed Specification 15
Client Implementation of Remote New System Setup Protocol *15*
Client Environment for the Downloaded Image *15*
Proposed Baseline Extensions 15

REMOTE WAKEUP

Remote WakeUp Overview 16
Baseline Overview 16
Detailed Specification 17
Proposed Baseline Extensions 17
Windows NT 5.0 and Memphis Extensions 17

POWER MANAGEMENT

Power Management Overview 18
Baseline Overview 18
Detailed Specification 18
ACPI Description *19*
Power States *19*
Platform Power Management Characteristics *20*
Proposed Baseline Extensions 20

MOBILE COMPUTERS

Category Description 21
Motivation *21*
What Makes Mobiles Different *21*
Time Frame 22
Mobile Baseline Overview 22
Detailed Specification 23
Instrumentation *23*
Remote New System Setup *25*
Remote Wake-up *26*
Power Management *26*
Proposed Extensions 26

SERVER SYSTEMS

Category Description 28
Motivation *28*
What Makes Servers Different *29*
Reliability, Availability, Serviceability *29*
Remote Management *29*
Emergency and Preboot Management *30*
Server Baseline Overview 30
Detailed Specification 31
Instrumentation *31*
Dynamic Device Support *33*
Proposed Baseline Extensions *33*
Remote New System Setup *33*
Proposed Baseline Extensions *34*
Remote Wake-up *34*
Proposed Baseline Extensions *34*
Power Management *35*
Proposed Baseline Extensions *35*

INFORMATION AND RESURCE REFERENCES 36

TERMS AND ACRONYMS 37

APPENDICES

APPENDIX A: DHCP EXTENSIONS FOR NEW SYSTEM SETUP 39

Protocol Overview 39
Relationship to the Standard DHCP Protocol 41
Client Behavior 43
Initiation 43
Discovery Reply 44
Installation Service Request 46
Installation Service Reply 48
Executable Download and Execution 48
MTFTP Operation 48
Server Behavior 49
Redirection Service Behavior 49
Installation Service Behavior 50
Response to DHCPREQUEST 50
TFTP Service 50

APPENDIX B: PREBOOT EXECUTION ENVIRONMENT 51

Client State at Bootstrap Execution Time 51
Bootstrap Calling Convention 51
Memory Usage 52
Interrupt Vector Table 55
Preboot API Entry Point and Installation Check 55
Int 1Ah Function 5650h (Preboot API Installation Check) 56
Preboot API Entry Point Structure 56
Register Usage for Preboot APIs 57
Preboot Services API 58
UNLOAD PREBOOT STACK 58
GET BINL INFO 58
RESTART DHCP 58
RESTART TFTP 59
MODE SWITCH 59
TFTP API Service Descriptions 61
TFTP OPEN 61
TFTP CLOSE 61
TFTP READ 61
TFTP/MTFTP READ FILE 61
PROTECTED-MODE TFTP/MTFTP READ FILE 62
TFTP_GET_FILE_SIZE 62
UDP API Service Descriptions 64
UDP OPEN 64
UDP CLOSE 64
UDP READ 64

UNDI API Service Descriptions....65
UNDI STARTUP....*65*
UNDI CLEANUP....*65*
UNDI INITIALIZE....*65*
UNDI RESET ADAPTER....*66*
UNDI SHUTDOWN....*66*
UNDI OPEN....*67*
UNDI CLOSE....*67*
UNDI TRANSMIT PACKET....*67*
UNDI SET MULTICAST ADDRESS....*67*
UNDI SET STATION ADDRESS....*68*
UNDI SET PACKET FILTER....*68*
UNDI GET INFORMATION....*68*
UNDI GET STATISTICS....*68*
UNDI CLEAR STATISTICS....*69*
UNDI INITIATE DIAGS....*69*
UNDI FORCE INTERRUPT....*69*
UNDI GET MULTICAST ADDRESS....*69*
UNDI GET NIC TYPE....*70*

APPENDIX C: PREBOOT API COMMON TYPE DEFINITIONS....71

APPENDIX D: PARAMETER STRUCTURE AND TYPE DEFINITIONS....74

APPENDIX E: TFTP API PARAMETER STRUCTURE AND TYPE DEFINITIONS....78

APPENDIX F: UDP API CONSTANT AND TYPE DEFINITIONS....82

APPENDIX G: UNDI API CONSTANT AND TYPE DEFINITIONS....84

APPENDIX H DMI INSTRUMENTATION DETAILS....91

APPENDIX I: DHCP OPTIONS FOR HOST SYSTEM CHARACTERISTICS....94

APPENDIX J: UUIDS AND GUIDS....99

Executive Summary

Wired for Management (WfM) is an Intel initiative to improve the manageability of desktop, mobile, and server systems. Better manageability will add value by decreasing the Total Cost of Ownership (TCO) of business computing while taking advantage of the power and flexibility of high-performance PC computing.

The Wired for Management Baseline Specification continues the work of the Wired for Management initiative. This Baseline Specification is designed to help PC and server manufacturers produce systems that can be effectively managed over corporate networks to reduce customers' support costs. The interface specifications in this version of Wired for Management Baseline for instrumentation, remote wake-up, and remote new system setup are completely compatible with those described in the Network PC Guidelines. In addition, mobile- and server-specific requirements for these areas are gathered into the mobile and server sections. By outlining a recommended minimum level of management capabilities for all desktop, mobile, and server systems, the WfM Baseline lays a standards-based foundation on which PC manufacturers can build to provide even higher levels of system manageability.

Systems based on the WfM Baseline Specification will feature software agents and other capabilities that enhance networked operation and reduce support costs, while taking advantage of the PC's flexibility, performance, and compatibility with existing networks. The design guidelines specified in the Baseline will enable manufacturers to quickly deliver the integrated capability for central administration, remote network configuration, off-hours maintenance, and constant monitoring of system health.

Version 1.1 of the WfM Baseline defines the requirements for:

- Instrumentation to ensure that all hardware devices can be recognized and acted on by software, aiding in successful remote troubleshooting and repair.

- Standard interfaces and protocols to allow remote network access and interoperability between different vendor implementations of instrumentation and different management applications.

- Platform agents that allow remote configuration of all system software, from the operating system through drivers and applications, even without a formatted hard disk.

- Access and power management to allow maintenance of networked systems during off-hour periods.

Section 1

Need for Management Baseline and Overview

Need To Lower Total Cost of Ownership

The combination of personal computers and networks has proven potent for businesses, as connected PCs have come to play a central role in a wide range of mission-critical business applications. But while businesses have enjoyed the increased abilities that networked PCs bring to their organizations, their Information Technology (IT) departments have had to deal with the increasing administrative burden and costs of keeping things running smoothly. The proliferation of hardware and software choices, the expansion of PC capability and the explosion of the Internet have made the corporate computing environment increasingly complex — and expensive — to deploy and manage. With large environments of mixed computing devices, it becomes increasingly difficult for IT organizations to maintain machines for maximum availability, repair them quickly and manage them as a vital corporate asset.

The ongoing expense incurred in deploying and managing is a major factor in the Total Cost of Ownership (TCO) of personal computers. Studies by Gartner Group and others use different methods for computing TCO. Some include so-called "soft" costs, such as lost productivity when an employee interrupts the colleague in the next office for help with a system configuration question. Others focus only on "hard" costs, such as administrative tools and salaries of administrative personnel. A typical figure for "soft" TCO is around $8,000 per year; "hard" cost numbers are usually in the $2,000-$3,000 range. Whichever way they choose to compute TCO, IT departments agree that lower costs are needed to keep their businesses competitive.

The Wired for Management Baseline addresses some key factors that contribute to high TCO:

- **Ease of use problems**. The number of supported users and devices has been increasing rapidly. Understandably, many of these new users lack the desire and/or the skills to effectively manage their PCs. This causes frequent calls to technical support, because end-users cannot resolve even the most basic of questions about their system.

- **Growing demands for availability.** As PCs take on more of an organization's core business functions as well as its communications and productivity applications, the pressure for IT to keep the computing environment available rises. Costs of downtime range from thousands to millions of dollars per hour. Yet a growing number of situations require taking the system down, for example, virus identification and cleaning, installing upgrades, or providing other types of system maintenance. Downtime also occurs because IT may lack needed information on configurations, bandwidth or usage models.

- **Time-consuming repairs.** The complexity that comes with an open, mixed computing environment increases the time to diagnose, troubleshoot and fix a failure. Users who cannot accurately describe their system or problem increase the number of phone calls to technical support. Support personnel, who often lack the tools to remotely determine the problem or system configuration, must schedule on-site visits. Fixing a failure often

requires multiple trips. Most current help desk products do not provide problem resolution assistance. As a result, repairs are slow and labor-intensive; support personnel tend to burn out quickly and have high turnover, all of which cause IT costs to rise.

- **Difficulties of asset management.** Asset management is the process of maximizing the use of assets to produce revenue while minimizing their overall costs. When the computing investment is distributed, inventory and tracking information is difficult and time-consuming to gather. Currently it is handled as a manual process — one that is most often not accurate. Stories abound of otherwise well-managed companies that find after doing an inventory that they have thousands more systems than they thought — or thousands fewer. IT organizations need inventory and configuration data for issues such as server/license consolidation and rationalization, leasing considerations, analyzing effectiveness and costs of training, software upgrade analysis for volume purchasing plans, cost efficiency of outsourcing, warranty usage improvements and effective management of cascading technology.

The Solution: Making PCs Universally Manageable and Universally Managed

Fortunately, the connected PC has the capacity to lower the cost of PC management. The "intelligence" of the PC and its connection to the network can enable the PC to play an active role in self-management and make it easier for IT organizations to automate and centralize management activities. Today, however, fully managed PCs are the exception rather than the rule.

There are two major reasons for this situation:

- While many PCs contain features intended to make them manageable, there is no common set of features that are guaranteed to be found in *all* PCs.

- Manageability features are typically made available through proprietary means, not open industry specifications.

The result is a tight coupling between management applications and particular platforms: XYZ-brand PCs can be managed only by the XYZ management application. In a business with PCs from multiple suppliers, this leads to two choices: deploy an unwieldy, complicated collection of management applications that don't interoperate, or stick with labor-intensive manual procedures. Either choice drives up TCO.

The Wired for Management Baseline resolves these shortcomings by defining a targeted minimum set of management features for all PC platforms. The features are made available through well-defined, and in some cases, industry-standard interface specifications, giving management applications a consistent way to access those features. The goal is a mix-and-match capability among platforms and management applications: Any platform that meets the Baseline can be managed by any management application that uses the Baseline's interface specifications.

Examples of features defined in the Baseline that help efficiently manage PCs over networks include:

- A common set of PC instrumentation that provides remote management applications access to a common set of platform characteristics and settings in a standard format using standard interface specifications.
- Power management and remote wake-up features that enable off-hours maintenance and management by providing interface specifications for 'waking up' a 'sleeping' Manageable PC.
- Support for remote new system setup, which provides protocol and interface specifications that enable automation of initializing new or re-deployed Manageable PCs.

Note that the Baseline specifies a *minimum* set of management capabilities. Vendors of both platforms and management applications are encouraged to further increase the value of their products by incorporating additional manageability features.

Baseline Component Technologies

The Wired for Management Baseline focuses on four technologies that together enable a "Manageable PC" and make it possible to reduce the TCO of business computing: instrumentation, service boot, remote wake-up, power management and discovery.

Instrumentation

To begin reducing TCO, management applications must have complete programmatic access to the state, control and descriptive parameters of the PC and its component subsystems. A Manageable PC with instrumentation provides such access for both local and remote applications. In addition, instrumentation infrastructure provides the means by which the PC and its components can report state changes and unusual conditions autonomously (that is, without being specifically queried by a management applications).

To derive the full benefits of instrumentation, it must be possible to manage any PC from any management application. This is possible when the data is provided in standard format through standard interfaces.

The Wired for Management Baseline specifies a standard data format and standard remoteable interface for instrumentation, and defines the minimum set of data that a Managed PC must supply through its instrumentation. This ensures the availability of information for:

- Problem detection and notification of the appropriate support organization
- Problem diagnosis and correction
- Tracking or logging the failure information
- Computer equipment inventory

By streamlining these processes, instrumentation increases PCs' ease of use, reduces the labor costs involved in maintaining and supporting PCs, and increases the IT organization's ability to administer and manage its business computing assets.

Service Boot

Service boot is a means by which agents can be loaded remotely onto the PC to perform management tasks in the absence of a running OS. Service boot capability enables the automation of a number of PC management tasks. Examples include the following:

- Initial configuration of new machines
- Diagnosis of problems that prevent the operating system from functioning correctly
- Configuration updates prior to booting the operating system

Such automation can lower the costs of administration and technical support, thus decreasing TCO.

Remote Wake-up

To minimize the downtime end-users experience for system maintenance and upgrades, a Manageable PC must provide the ability to be automatically managed during off-hours, when the system is not otherwise in use. Remote wake-up lowers TCO by increasing availability for end-users and by minimizing time for system maintenance (which, in turn, reduces system failures and downtime).

The Wired for Management Baseline specifies that it must be possible to remotely wake-up a Managed PC from a soft-off state.

Power Management

Power management allows a PC to be consuming little power, but still be able to be fully operational in a short period of time. This enables low power consumption, automated off-hours maintenance, and an improved end-user experience when starting up (really 'waking up') a PC.

In conjunction with a remote wake-up, power management helps lower TCO by reducing end-user downtime and labor costs for routine system maintenance, and by reducing power consumption.

Baseline Evolution and the Proposed Future Baseline Technologies

The Wired for Management Baseline Specification cannot be static, and will evolve over time. Intel expects major revisions of the Baseline to occur on about a yearly basis as platform vendors consolidate the current feature set and additional features win widespread support.

This version of the Baseline focuses on desktop PCs, mobile, and server systems. Future versions will include WBEM interfaces and specifications as these are developed from the standards bodies.

Some manageability features that support TCO reduction have not been made required elements in the current Baseline because they lack agreed-upon interface specifications or may not be universally deployable in the timeframe covered by the current Baseline. These features are candidates for inclusion in future versions of the Baseline. To provide guidance for platform vendors in planning for future Baselines, this document describes these as proposed extensions to the Baseline. These proposed extensions include the use of WBEM/CIM and WMI interfaces for instrumentation, full support for ACPI, and a network protocol to enable the PC to discover and be discovered by services and other machines on the network.

WfM Baseline and Other Industry Standards/Guidelines

The WfM Baseline is designed for compatibility with current and emerging industry standards/guidelines.

ACPI

The Wired for Management Baseline specification for remote wake-up and power management promotes the use of the Advanced Configuration and Power Interface (ACPI).

DMI

Instrumentation requirements are specified in terms of the Desktop Management Task Force's (DMTF's) Desktop Management Interface (DMI) v2.00. For future platforms that will be using the new proposed Microsoft instrumentation model, WMI, will be compatible with DMI instrumentation. Intel and Microsoft are working together to ensure that there will be no loss of management functionality when systems are upgraded to NT 5.0 or 'Memphis' operating systems.

Network PC Design Guidelines

The interface specifications in the Wired for Management Baseline for instrumentation, remote wake-up, and remote new system setup are completely compatible with those described in the Network PC Design Guidelines. Thus, it will be possible for the same applications to manage Network PCs and PCs, mobile, and server systems conforming to the current Baseline.

SNMP

Managing PCs in a corporate enterprise might require that these devices integrate with current enterprise management applications. WfM Baseline recommends using SNMP support, in addition to DMI instrumentation, in these cases. It is recommended that if SNMP support is provided, the same information available through DMI or WMI interfaces should also be available through SNMP.

WBEM

The WfM Baseline is designed for compatibility with current and emerging industry standards. In the current Baseline, instrumentation requirements are specified in terms of the Desktop Management Task Force's (DMTF's) Desktop Management Interface (DMI), developed by the DMTF, specifically the DMI v2.00 Management Interface for management applications and DMI MIF files for component information definition. When Web-Based Enterprise Management (WBEM) technology and standards become available, the Baseline will specify the use of WBEM mechanisms.

How This Document Is Organized

The remainder of this document describes the Wired for Management Baseline. Sections 2 through 7 specify the Baseline functionality for each technology category and describe the interface specifications used to access that functionality. Section 8 lists documents that are referenced in this Baseline Specification and provides information about how to obtain them. Section 9 provides a brief glossary of terms and acronyms.

Throughout this Baseline, the following definitions apply:

- **Required:** Features that must be implemented to be compliant with this Baseline
- **Recommended:** Features that support or improve manageability. If a recommended feature is implemented, it must meet the requirements for that feature as defined in this Baseline. Some recommended features may become requirements in the future.
- **Optional:** Features that are neither required or recommended. If an optional feature is implemented, it must meet the requirements for the feature as defined in this Baseline

Section 2

Instrumentation

Instrumentation Overview

Instrumentation is a common methodology and syntax for defining the management features and capabilities of all hardware, software and attached peripherals of the PC. Instrumentation allows management applications to view and alter the state of a managed PC, and to monitor (be notified of changes in) the state. Industry-standard instrumentation makes it possible for any Baseline-managed PC to be managed remotely by any management application, regardless of the vendor or operating system.

Baseline Overview

DMI-Based Instrumentation

This Baseline requires that compliant platforms utilize the DMI v2.00 Management Interface (MI) and Component Interface (CI) application programming interfaces and host a DMI v2.00 Service Provider, as defined by the DMTF. This Baseline Instrumentation specification also identifies specific DMI standard groups, including event generation groups, that must be instrumented for a Baseline-compliant platform. In addition, it describes the deployment model for the Service Provider and instrumentation.

The Baseline Instrumentation specification does not define how instrumentation is implemented beneath the required application programming interfaces or what sources or methods are used to extract management information from the platform. The Baseline also does not prohibit vendors of platforms or platform components from extending the manageability of the platform by instrumenting additional DMI standard or proprietary groups, or by extending the management framework beyond DMI.

Detailed Specification

DMI Version 2.00 specifies four major elements that enable standard instrumentation to interact with management applications:

1. **Management Information Formats (MIF)**: a language that precisely defines the syntax and semantics of the management capabilities of system components. MIFs are expressed in terms of **components**, which have one or more named **attributes** which collectively define the information available to a management application. Attributes are collected into named **groups** for ease of reference. Groups have one or more attributes. The DMTF standardizes management information around the concept of **standard groups**. Component vendors create components by specifying and instrumenting those standard groups which best describe the management capabilities of a given component.

2. Two sets of application programming interfaces (APIs): the **Management Interface** (MI) used by management applications to interact with the Service Provider (described below) and the **Component Interface** (CI) used by managed components to interact with the Service Provider.

3. The **DMI Service Provider**[1] is an active, resident piece of code running on a computer system that mediates between the MI and CI and performs services on behalf of each. Each hardware or software product (component) in a PC has a MIF file to describe its manageable characteristics. When one of these products or components is initially installed into the system, the MIF file is added to the (implementation-dependent) *MIF database* managed by the Service Provider.

4. A set of services for facilitating **remote communications** at the Management Interface, using the Remote Procedure Calls (RPC) facilities deployed on all PC operating systems. This permits management applications to reside either locally on the managed platform, or remotely on another system. RPC allows the same APIs to be used transparently, regardless of whether the source and destination are co-located, and regardless of the [4] underlying communications media.

DMI 2.0 Service Provider

A system compliant with this specification must deploy a DMI Version 2.00 Service Provider, as specified in *Desktop Management Interface Specification, Version 2.00* [2][1]. (For information on DMTF documents referenced in this document, refer to Section 7.)

Compliance with this section of the Baseline Instrumentation is measured by *DMI Compliance Guidelines* [3][2], as specified by the DMTF.

DMI Standard Groups

Table 2-1 lists the standard groups, from the *System Standards Group Definition, Approved Version 1.0* [4][3], that must be instrumented and deployed on Baseline-compliant systems.

[1] The "Service Provider" was known as the "Service Layer" under DMI v1.x. The DMTF renamed the component for DMI v2.00.
[2] Desktop Management Interface Specification, Version 2.00. http://www.dmtf.org/tech/specs.html
[3] Desktop Management Interface Compliance Guidelines, Version 1.0, http://www.dmtf.org/tech/specs.html
[4] Systems Standard Group Definitions, Approved Version 1.0, http://www.dmtf.org/tech/apps.html

Table 2-1. Required DMI System Standard Groups

DMTF|ComponentID|001
DMTF|Disks Mapping Table|001
DMTF|Disks|002
DMTF|General Information|001
DMTF|Keyboard|003
DMTF|Mouse|003
DMTF|Operating System|001
DMTF|Partition|001
DMTF|Physical Container Global Table|001
DMTF|Processor|003
DMTF|System BIOS|001
DMTF|System Cache|002
DMTF|System Slots|003
DMTF|Video BIOS|001
DMTF|Video|002

Tables 2-2 and 2-3 contain standard groups related to system resource management and physical memory management (drawn from *System Standards Group Definition, Approved Version 1.0* [5][3]). All of these groups are valid standard groups, but the DMTF has designated some groups as 'Obsolete' and is recommending that instrumentation migrate to the designated 'Replacement' groups. To comply with this Baseline specification, a system must be instrumented with either **all** of the "Obsolete" groups or **all** of the corresponding "Replacement" groups. It is highly recommended that the "Replacement" groups be selected for newly implemented instrumentation.

Table 2-2. Required System Resource Management Standard Groups

DMI Standard Group	***Implementation Guidelines***
DMTF|System Resources 2|001	For new instrumentation
DMTF|System Resource Device Info|001	For new instrumentation
DMTF|System Resource DMA Info|001	For new instrumentation
DMTF|System Resource I/O Info|001	For new instrumentation
DMTF|System Resource IRQ Info|001	For new instrumentation
DMTF|System Resource Memory Info|001	For new instrumentation
DMTF|System Resources|001	Obsolete, recommended for legacy instrumentation only
DMTF|System Resources Description|001	Obsolete, recommended for legacy instrumentation only

[5] Systems Standard Group Definitions, Approved Version 1.0, http://www.dmtf.org/tech/apps.html

Table 2-3. Required Physical Memory Management Standard Groups

DMI Standard Group	*Implementation Guidelines*
DMTF|Memory Device|001	For new instrumentation
DMTF|Memory Array Mapped Addresses|001	For new instrumentation
DMTF|Memory Device Mapped Addresses|001	For new instrumentation
DMTF|Physical Memory Array|001	For new instrumentation
DMTF|Physical Memory|002	Obsolete, recommended for legacy instrumentation only

The following standard groups from the *LAN Adapter Standard Groups Definition, Release Version 1.0* [6][4], must be instrumented and deployed on Network Interface Cards (NICs) in systems compliant with this specification:

DMTF|Network Adapter 802 Port|001
DMTF|Network Adapter Driver|001

In addition to supplying instrumentation for and installation of the above standard groups, compliance with this section of the Baseline Instrumentation is measured by *DMI Compliance Guidelines, Version 1.0* [7][2].

DMI Event Generation

Some of the standard groups listed above are associated with event generation groups. Event generation for these groups is optional. However, if the instrumentation generates events associated with a required group, event generation must be compliant with the event model specification defined in *Desktop Management Interface Specification, Version 2.00*[8] [1].

DMI Management Interfaces

Management applications for WfM Baseline-compliant systems must implement the procedural version of the Management Interface (MI) as specified for DMI Version 2.00. This includes compliance with for the *Service Provider API for Management Applications* and the *Management Provider API*. The new procedural interface introduced with DMI Version 2.00 is an interface which supports remote access, using Remote Procedure Call.

Hardware and software vendors who want to supply components or products that comply with Baseline Instrumentation are required to be compliant with the Component Interface (CI) as specified for DMI Version 2.00. This interface includes compliance with the *Service Provider API for Components* and the *Component Provider API*. The CI is a local interface, to be used within a single system.

Hardware and software vendors are strongly encouraged to implement to the procedural CI defined in DMI Version 2.00. See the DMTF Compliance Guidelines regarding backwards compatibility for existing instrumentation implemented using the DMI Version 1.x block interface.

[6] LAN Adapter Standard Groups Definition, Release Version 1.0. http://www.dmtf.org/tech/apps.html
[7] Desktop Management Interface Compliance Guidelines, Version 1.0, http://www.dmtf.org/tech/specs.html
[8] Desktop Management Interface Specification, Version 2.00. http://www.dmtf.org/tech/specs.html

For detailed specification of the required Management Interfaces, refer to *Desktop Management Interface Specification, Version 2.00* [9][1].

Compliance with this section of the Baseline Instrumentation is measured by *DMI Compliance Guidelines, Version 1.0* [10][2].

DMI Service Provider and Instrumentation Deployment

The DMI 2.0 Service Provider and the required instrumentation is required to be deployed with the platform. This enables any compliant management application to access and manage the system via its DMI instrumentation as soon as the system is up and running.

The Service Provider implementation is OS-dependent. The Service Provider can be deployed by the operating system vendor as part of the operating system or by the platform vendor as a separate software component installed in conjunction with the operating system.

The instrumentation and associated platform-specific drivers are both platform- and OS-dependent. They should be deployed by the platform vendor along with the platform, and installed with the operating system.

Guide to DMI Instrumentation Development

The DMTF supplies a set of guidelines to help hardware and software vendors with the task of adding instrumentation to their products. *Desktop Management Task Force: Enabling your product for manageability with MIF files, Revision 1.0* [11][5] (also known as *MIF Guidelines*) is available from the DMTF.

Proposed Baseline Extensions

Programming Interfaces

When Web Based Enterprise Management (WBEM) standards are defined, the Baseline Specification will reference the WBEM standards and employ WBEM mechanisms to define the minimal collection of information about managed components that must be available from a compliant PC. The WBEM standards will support use of the DMI Management Interface (MI) by management applications. Currently, the IETF and DMTF are working to define WBEM standards, the first is expected to be published in early 1998.

Microsoft Corporation is developing the Windows* Management Interface (WMI) extensions to the Win32 Driver Model (WDM). When WMI is available (starting with Windows NT* 5.0 and Windows “Memphis”), the WfM Baseline will specify WMI as an alternative interface through which instrumented components provide information and notifications on PCs that run a Microsoft Windows Memphis or NT 5.0 operating systems. Intel and Microsoft are working together to ensure that there will be no loss of management functionality when PCs with DMI-compliant component instrumentation are migrated to using WMI-enabled operating systems.

[9] Desktop Management Interface Specification, Version 2.00. http://www.dmtf.org/tech/specs.html
[10] Desktop Management Interface Compliance Guidelines, Version 1.0, http://www.dmtf.org/tech/specs.html

[11] MIF Guidelines. http://www.dmtf.org/tech/apps.html

On PCs running operating systems other than Microsoft Windows, instrumented components are required to provide information and notifications through the DMI v2.00 Component Interface. As WBEM technology becomes available on these systems, the WfM Baseline will specify the use of the WBEM mechanisms for these systems.

DMI Standard Groups

The Baseline defines both an initial baseline and proposed extensions. The proposed extensions list additional DMI standard groups that are recommended for inclusion in subsequent versions of the Baseline. It also describes proposed group definitions that outline new platform functions not yet standardized by the DMTF, but that are candidates for standardization and subsequent inclusion in a future version of this Baseline.

Table 2-4 lists standard groups from the *System Standards Group Definition, Approved Version 1.0* [12][3] that are proposed as candidates for inclusion in the next version of this Baseline. They are not required in the current Baseline, since it may be difficult to deploy instrumentation for these groups in the timeframe covered by the current Baseline.

Table 2-4. Proposed Baseline Extensions: DMTF Standard Groups

DMTF|BIOS Characteristics|002
DMTF|Cooling Device|001
DMTF|FRU|002
DMTF|Logical Drives|001
DMTF|Operational State|002
DMTF|Temperature Probe|001
DMTF|Voltage Probe|001

Future versions of the Baseline will include as appropriate new DMI standard groups (or their WBEM equivalents).

[12] Systems Standard Group Definitions, Approved Version 1.0, http://www.dmtf.org/tech/apps.html

Section 3

Remote New System Setup

Remote Boot Overview

PCs generally boot (load an operating system, or other executable image) from a local floppy or hard disk. However, there are several situations in which the PC's manageability can be enhanced by having the capability to load the executable from a server. For example:

- **Remote new system setup**: If the PC does not yet have an OS installed on its hard disk, downloading an executable from a server can help automate the OS installation and other configuration steps. This is accomplished by loading from the server an executable image that will install the OS.

- **Remote emergency boot**: If the PC fails to boot its installed OS due to a hardware or software failure, downloading an executable from a server can provide the PC with an executable that enables remote problem notification and diagnosis.

- **"Classic" remote boot**: The PC executes a centrally maintained OS image as its normal mode of operation. Typically this has been used to enable diskless client PCs connected to a server over a LAN.

The client's request for the download of an executable image may be conditioned on its configuration, the state of its hardware or software, or both, but the interaction is always initiated by the client (not the server)[13].

Baseline Overview

The purpose of the Baseline specification for remote new system setup is to enable interoperation between clients and servers supplied by independent vendors, so the main focus is on the messages exchanged between the client and the server. However, since it is also important that the meaning of the client-server interaction be standard as well as its form, some aspects of the independent states and behaviors of the client and server are also specified. The client-server interaction is based on two IETF standard protocols: Dynamic Host Configuration Protocol (DHCP, RFC 1541) and the Trivial File Transfer Protocol (TFTP, RFC 1350)[14].

This version of the Baseline covers only the case of a client requesting remote new system setup. This section describes the intended usage of the features enabled by the remote new system setup specification. In general, it assumes new client PCs are being deployed into a LAN environment.

[13] That is, the protocol sequence per se is always initiated by the client. However, some higher-level mechanism might implement a "push"-style management operation by causing the client to initiate the protocol.

The operations of standard DHCP and/or BOOTP servers (which serve up IP addresses and/or OS boot images) will not be disrupted by the use of the extended protocol. Clients and servers that know about these extensions will recognize and use this information, and those that don't recognize these extensions will ignore it.

In summary, the Baseline remote new system setup protocol operates as follows. The client initiates the protocol by broadcasting a DHCP request containing an extension that identifies the request as coming from a client that implements the remote new system setup protocol. Assuming that a server implementing this protocol is available, after several intermediate steps, the server sends the client the name of an executable file on the server. The client uses TFTP to download the executable. 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.

Server Deployment

A *WfM configuration server* is a server that has been configured to recognize and respond to the DHCP extensions specified in this section of the Baseline. The specification supports two ways of deploying this functionality:

1. **Combined standard DHCP and configuration 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 serve up new system setup executables to WfM clients that request them.

2. **Separate standard DHCP and configuration services.** WfM configuration servers are added to the environment. They respond only to WfM clients, and provide only new system setup executables.

Each WfM configuration server must have one or more new system setup executables appropriate to the clients that it serves.

Client Deployment

This Baseline does not specify the operational details and functionality of the new system setup executable 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.

This Baseline specifies the protocols by which a client requests and downloads an executable image from a server and the minimum requirements on the client execution environment when the downloaded image is executed. It does not specify the implementation details of preboot code on the client or the provisions for security (authorization, privacy, and data integrity) during the DHCP and TFTP exchanges between the client and the server.

[14] IETF RFCs are available at http://www.ietf.cnri.reston.va.us/home.html

Detailed Specification

Client Implementation of Remote New System Setup Protocol

A Baseline-compliant PC must be capable of using DHCP and TFTP as described in Appendix A to effect the installation of an OS from a server. The conditions under which the PC initiates remote OS installation are implementation dependent.

Client Environment for the Downloaded Image

To facilitate the development of the interoperable downloadable images, certain aspects of the client environment at the time the downloaded image is executed must be guaranteed. These aspects include:

- How execution of the downloaded image is initiated.
- Use of client memory, including which locations the downloaded image can utilize and which it must not modify.
- Relevant interrupt vector settings.
- An implementation of the Preboot Services API, which the downloaded image can use for basic network I/O. This has the additional benefit that a new type of network adapter can be deployed without the necessity of deploying a new adapter-specific download image.

The details of the client preboot execution environment requirements appear in Appendices B through G.

Proposed Baseline Extensions

Future Baseline versions will include specifications for remote emergency boot, “classic” remote boot, and other services. The following features are likely to be included in the next revision of the Baseline Specification:

- Security provisions for the DHCP and TFTP interactions.
- Specification of support for other manageability features such as an emergency boot.

Section 4

Remote Wake-Up

Remote Wake-Up Overview

Being able to wake up a platform remotely is important to the ability to manage the PC and perform routine maintenance tasks when the system is not being used. Since PCs must be capable of switching to a power-reduced mode when not being used, remote-wake up must be effective even with systems that are in a reduced-power state.

This section of the Baseline describes the mechanisms by which this remote wake-up can be implemented. Supported power management modes are discussed in the Section 5.

Baseline Overview

If a PC supports a reduced power state, it must be possible to bring the system to a fully powered state in which all management interfaces are available. Typically, the LAN adapter recognizes a special packet as a signal to wake up the system. The current Baseline recommends Advanced Micro Devices, Inc. Magic Packet™ for this functionality. Remote wakeup can also be accomplished by rebooting the system from its low-power state. For hardware implementations that don't support a transition from the system's current low-power state to its full-power state while preserving context, rebooting is the only viable solution.

Upon shipment of Windows NT 5.0 and 'Memphis", wakeup capabilities are required to be based on matching patterns specified by the local networking software, as described in "Network Wake-up Frames" and "Network Wake-up Frame Details" in *Network Device Class Power Management Reference Specification, Version 1.0* or higher.

This requirement applies specifically to Ethernet and token ring adapters. *Network Device Class Power Management Reference Specification* does not support ATM and ISDN adapters.

For additional implementation guidelines, see items #34-35 in the "Network Communications" chapter of *PC 97 Hardware Design Guide*[15] from Microsoft. Implementation details are described in "Network Wake-up Frames" and "Network Wake-up Frame Details" *in Network Device Class Power Management Reference Specification, Version 1.0* or higher, also from Microsoft.

[15] PC 97 Hardware Design Guide: http://www.microsoft.com/hwdev/pc97.htm

Detailed Specification

This Baseline is compatible with use of the Magic Packet™ technology from. AMD's, for remote wake-up. For a description of this technology, see http://www.amd.com/products/npd/overview/20212d.html.

For this Baseline it is required that systems be in a state so they are available for service and management from the network. This system state has the following characteristics:

- The platform's power state is full ON.
- Peripherals such as the monitor that are attached to the platform but are not required for the external management activity need not be fully powered.
- All management interfaces provided by the loaded environment are available.

Proposed Baseline Extensions

Future Baseline revisions will provide more recommendations and details on remote wake-up technologies.

ACPI

It is recommended that as Advanced Configuration and Power Management Interface (ACPI) technology becomes available, PC system manufacturers include as much hardware and BIOS ACPI support as possible in Managed PCs.

Windows NT 5.0 and Memphis Extensions

A system that runs future releases of Microsoft operating systems (starting with Windows NT 5.0 or Memphis) and has hardware and BIOS support for ACPI will wake up and resume execution from the point where it was suspended.

Section
5

Power Management

Power Management Overview

Power management contributes to lower TCO while enhancing user experience. The need for power management is underscored by the requirement for remote wake-up where remote off-hours management of PCs must not decrease system life. In addition, new regulations and labeling programs are mandating tighter restrictions on platform power consumption. For example, Energy Star requires that PCs automatically power down to 30 watts or less when not in use. Other programs call for lower power consumption than the current 30W limit, with some specifying 5W or less.

The Advanced Configuration and Power Management Interface (ACPI) is an abstract interface between the OS and the hardware designed to achieve independence between the hardware and the software. The ACPI Tables, which describe a particular platform's hardware, are at heart of the ACPI implementation and the role of the ACPI BIOS is primarily to supply the ACPI Tables (rather than an API). ACPI provides the opportunity to integrate the interface for controlling power management and Plug-n-Play features on system devices.

Baseline Overview

This Baseline strongly recommends using ACPI-compliant components where possible. At time of publication, it is recognized that ACPI technology is not available for all components. As ACPI technology becomes available, it is recommended that PC system manufacturers include as much hardware and BIOS ACPI support as possible in Network PCs. ACPI is currently described in the Proposed Extensions section. Future versions of WfM Baseline will require ACPI-compliant components.

Portions of the ACPI specification have been included; however, for complete details the reader should refer to the actual specification. A copy of the ACPI specification may be obtained from the web site: http://www.teleport.com/~acpi.

Detailed Specification

ACPI defines the interfaces between the operating system software, the hardware and BIOS software. It also defines the semantics of these interfaces. The term operating system power management (OSPM) is used to refer to an operating system that uses the ACPI interfaces to implement power management policies.

Figure 5-1 shows the software and hardware components relevant to ACPI and how they relate to each other. ACPI describes the *interfaces between* components, the contents of the ACPI Tables, and the related semantics of the other ACPI components. Note that the ACPI

Tables, which describe a particular platform's hardware, are at heart of the ACPI implementation and the role of the ACPI BIOS is primarily to supply the ACPI Tables (rather than an API).

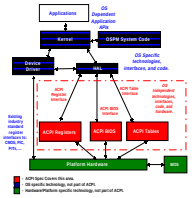


Figure 5-1 OSPM/ACPI Global System

ACPI Description

ACPI has three runtime components:

1. ACPI Tables - These tables describe the interfaces to the hardware. These descriptions allow the hardware to be built in flexible ways, and can describe arbitrary operation sequences needed to make the hardware function. ACPI Tables may contain p-code language, the interpretation of which is performed by the OS.

2. ACPI Registers - The constrained part of the hardware interface, described (at least in location) by the ACPI Tables.

3. ACPI BIOS - Refers to the portion of the firmware that is compatible with the ACPI specification requirements. Typically, this is the code that boots the machine (as legacy BIOSs have done) and implements interfaces for suspend, resume and some restart operations. The ACPI Description Tables are also provided by the ACPI BIOS. Note that in Figure 5-1 the boxes labeled "BIOS" and "ACPI BIOS" refer to the same component on a platform; the box labeled "ACPI BIOS" is broken out to emphasize that a portion of the BIOS is compatible with the ACPI specification requirements.

Power States

Baseline-capable systems have four distinct power states: Working, Sleeping, Soft Off, and Mechanical Off. In the Working state, user mode application threads are dispatched and running. Individual devices and processors may be in low-power states if they are not being used. Any device the system turns off because it is not actively in use can be turned on with short latency. (What "short" means depends on the device. An LCD display needs to come on in sub-second times, while it is generally acceptable to wait a few seconds for a printer to wake up.) When the computer is idle or the user has pressed the power button, the OS will put the computer into one of the sleeping states (Sleep or Soft Off). No user-visible computation occurs in a sleeping state. The sleeping states differ in what events can bring the system to a Working state, and how long this takes. For example pressing a key will cause a wake-up from the Sleeping state whereas a Remote Wake-up event (Magic Packet) or power switch is needed to wake from the Soft Off state. When the machine must awaken to all possible events and/or do so very quickly, it can enter only the Sleep state and achieves only a partial reduction of system power consumption. However, if the only events of interest are: 1) a user pushing on a switch or 2) receipt of a Remote Wake-up event then the system can be placed in the Soft Off state (For a discussion of Remote Wake-up refer to Section 4).

In the Soft Off state, the machine draws almost zero power and retains system context for an extended period of time; however the wake-up latency from Soft Off to the Working state is higher than the latency for the Sleeping state to the Working state..

New Meanings for the "On/Off" Button

The power button on the front of a Baseline capable machine is redefined to be a "soft" button, which will normally *not* turn the machine physically off, but will put it in a soft off or sleeping state instead. Unlike today's on/off button, the soft on/off button sends a request to the system. What the system does with this request depends on the policy issues derived from user preferences, user function requests, and application data.

A second "override" switch in some less obvious place (perhaps accessible through a pin hole or placed on the back of the machine) stops current flow forcing the machine into the Mechanical Off state without OS consent. This is required for legal reasons in some jurisdictions (for example, in some European countries) and to deal with various rare, but problematic, situations.

Platform Power Management Characteristics

Systems conforming to the Wired For Management Baseline will have the capability to be managed during off hours by having different modes for Day and Night.

- *Day Mode* - In day mode, clients will stay in the Working state all the time, and put unused devices into low power states whenever possible. OS-driven power management allows careful tuning of when to do this, thus making it workable.

- *Night Mode* - In night mode, clients sleep as deeply as they can sleep but are still able to wake up and answer service requests coming in over the network, phone links, etc., within required latencies. So, for example, a print server might go into deep sleep until it receives a print job at 3 A.M., at which point it wakes up in perhaps less than 30 seconds, prints the job, and then goes back to sleep.

Proposed Baseline Extensions

Future versions of WfM Baseline will require systems to support ACPI in both the platform and OS. This applies to all classes of computers, including desktop, mobile, home and server machines. Legacy support for APM BIOS in the platform is optional.

Section
6

Mobile Computers

Category Description

Mobile computers differ from desktop business computers in several ways. These differences require that the Wired for Management Baseline capabilities be implemented in a distinct manner on mobile computers. This section describes the changes needed to accommodate mobile computers and outlines the extensions needed in the future to keep mobile computers universally manageable and universally managed.

Motivation

Notebook computers make up a significant portion of PCs in businesses that have professional nomadic workers. Managing a mobile asset is a difficult task for corporations, resulting in a higher TCO for notebook PCs. IT organizations will continue to demand that manageability solutions extend to mobile PCs. This section specifies the Baseline requirements for mobile PCs that will help reduce TCO.

What Makes Mobiles Different

Mobile computers have many characteristics that make them different from desktop PCs. Among these are size, weight, power constraints, and roaming (i.e. occasionally connected, pull model communications). Dynamically attached devices (i.e. PC Card slots, drive swap bays), while not unique to mobile computers,[16] have long been a distinguishing characteristic of them and are covered in this specification. Several of these characteristics have major impacts on the way the Wired for Management Baseline is implemented on mobile computers.:

Mobile clients have an *occasionally connected* communications model, meaning they roam and are not always reachable by management applications. The connection type also varies based on what is available at the traveler's destination (LAN or dial-up). This affects the bandwidth and latency available for remote management. Additionally, the connection is a *pull model,* meaning it is made and broken at the discretion of the mobile worker -- not IT. All these differences profoundly impact the way the Baseline is implemented on a mobile computer.

Mobile platforms are designed to be dynamically expandable. Instrumentation must be "smarter" because devices can dynamically appear/disappear in a mobile system (e.g. PC cards, hot docking). This is especially true of dynamic communications adapters. Key parts of the Baseline rely on communications that occur when the OS has not yet booted or is inactive (i.e. remote Wake-up, remote new system setup). That necessitates the platform have intimate knowledge of the communications adapter. Most mobile computers have communications adapters that can be dynamically added/removed at any moment (i.e. PC Card NIC, hot docking

[16] USB, 1394, and device bay technologies bring dynamic device attachment to desktop PCs.

to a station that contains a NIC). The ease with which these adapters can be swapped means that a new and different adapter, unplanned by the PC manufacturer, could be inserted by the user. This must be taken into account when applying the Baseline to mobile platforms.

This Mobile section of the WfM Baseline specifies the minimum level of manageability that all mobiles must meet regardless of how they are used. The objective is to ensure that mobile computers can be managed by any management application that uses the Baseline's open interface specifications the same as any other PC that conforms to the Baseline.

Time Frame

Market forces will decide how rapidly the Baseline is adopted on mobiles. Some Baseline features will begin appearing on notebooks as early as the end of 2H'97. The goal is to have the vast majority of new notebook platforms offered for sale by 1H'98 Baseline-capable.

Mobile Baseline Overview

The Wired for Management Baseline for mobile PCs relies on the same technologies as for other Baseline-capable PCs, with some implementation differences due to the nature of mobile computers. These technologies enable a 'Manageable" mobile PC and satisfy the needs of IT organizations for inventory, configuration, and problem diagnosis and resolution, which in turn reduces TCO of their computing environment. The requirements for a mobile PC are:

- **Instrumentation** -- ***Required***. Same requirement as other Baseline-capable PCs with the addition of: 1) the DMTF Mobile Supplement to the System Standard Group definition, and 2) instrumentation support for hot pluggable devices that dynamically appear/disappear in mobile systems (e.g. PC cards, hot docking) without requiring a system reboot.

- **Remote New System Setup** -- ***Recommended***. Recommended for Baseline-capable mobile PCs with the understanding that a LAN connection may not always be available. May be provided via boot diskette, docking station, PC Card NIC, or LAN on motherboard.

- **Remote Wake-up** -- ***Recommended***. Recommended for notebooks connected to the LAN.

- **Power Management** —***ACPI compliance required***. Mobile computers must comply with the ACPI specification.

Detailed technical information on the requirements is provided in Sections 2 through 5.

Detailed Specification

Instrumentation

Instrumentation provides the means by which management applications can 'get' and 'set' the PC subsystem management data. A mobile PC typically has different subsystems that instrumentation must provide a way to 'get' and 'set'. Some examples of mobile subsystems are: docking stations, port replicators, power management, batteries, device swap bays, IR ports, and LCD displays.. The DMTF has defined a mobile supplement to the DMI System Standards Group definition that describes these subsystems. The Baseline specification requires that mobile computers be in compliance with the *Mobile Supplement to the DMI Systems Standard Group Definitions*.

Table 6-1 lists the standard groups and mobile groups, from the *System Standards Group Definition, Approved Version 1.0* [17] [3] that must be instrumented on Baseline-compliant mobile systems.

Table 6-1. DMI System Standard Groups

DMI Standard Group	***Implementation Guidelines***
DMTF|ComponentID|001	Required
DMTF|Disk Mapping Table|001	Required
DMTF|Disks|002	Required
DMTF|General Information|001	Required
DMTF|Keyboard|003	Required
DMTF|Mouse|003	Obsolete, Use mobile supplement DMTF|Pointing Devices|001
DMTF|Operating System|001	Required
DMTF|Partition|001	Required
DMTF|Physical Container Global Table|001	Optional
DMTF|Processor|003	Required
DMTF|System BIOS|001	Required
DMTF|System Cache|002	Required
DMTF|System Slots|003	Required if slots are present (i.e. PC Card slots)
DMTF|Video BIOS|001	Required
DMTF|Video|002	Required

Tables 6-2 and 6-3 contain standard groups related to system resource management and physical memory management (drawn from *System Standards Group Definition, Approved Version 1.0* [3]). All of these groups are valid standard groups, but the DMTF has designated some groups as 'Obsolete' and is recommending that instrumentation migrate to the designated 'Replacement' groups. To comply with this Baseline specification, a system must be instrumented with either **all** of the "Obsolete" groups or **all** of the corresponding "Replacement" groups. It is highly recommended that the "Replacement" groups be selected for newly implemented instrumentation.

[17] Systems Standard Group Definitions, Approved Version 1.0, http://www.dmtf.org/tech/apps.html

Table 6-2. Required System Resource Management Standard Groups

DMI Standard Group	*Implementation Guidelines*
DMTF|System Resource 2|001	For new instrumentation
DMTF|System Resource Device Info|001	
DMTF|System Resource DMA Info|001	
DMTF|System Resource I/O Info|001	
DMTF|System Resource IRQ Info|001	
DMTF|System Resource Memory Info|001	
DMTF|System Resources|001	Obsolete, recommended for legacy instrumentation only
DMTF|System Resources Description|001	

Table 6-3. Required Physical Memory Management Standard Groups

DMI Standard Group	*Implementation Guidelines*
DMTF|Memory Device|001	For new instrumentation
DMTF|Memory Array Mapped Addresses|001	
DMTF|Memory Device Mapped Addresses|001	
DMTF|Physical Memory Array|001	
DMTF|Physical Memory|002	Obsolete, recommended for legacy instrumentation only

Table 6-4 contains standard groups related to mobile computer systems (drawn from *DMTF Mobile Supplement to Standard Groups, version 1.0*[18]).

Table 6-4. Mobile Supplement Standard Groups

DMI Mobile Supplement Standard Group	*Implementation Guidelines*
DMTF|Portable Battery|001	Required
DMTF|Dynamic States|001	Required
DMTF|Video Output Device|001	Optional
DMTF|Infrared Port|001	Optional
DMTF|Pointing Device|001	Required
DMTF|System Power Management|001	Recommended
DMTF|Power Management Table|001	Recommended
DMTF|Power Management Association Table|001	Recommended
DMTF|Device Bay|001	Optional

If a Network Interface Card (NIC) is provided as part of the mobile platform the following standard groups from the *LAN Adapter Standard Groups Definition, Release Version 1.0* [19], must be instrumented and deployed on systems compliant with this specification:

[18] Mobile Supplement to Standard Groups, approved version 1.0, ftp.dmtf.org/upload/notebook/mobi10.doc
[19] LAN Adapter Standard Groups Definition, Release Version 1.0. http://www.dmtf.org/tech/apps.html

Table 6-5. Network Interface Card Standard Groups

DMTF|Network Adapter 802 Port|001
DMTF|Network Adapter Driver|001

Dynamic Device Support

Instrumentation for mobile platforms must support dynamic peripherals that appear/disappear (e.g. hot insertion/removal of PC cards, hot docking, drive swap bays). Re-enumeration of dynamic peripherals must be done without requiring the system to reboot. If a peripheral is provided as part of the mobile platform and that peripheral is encompassed in the DMI groups listed in Tables 6-1 through 6-5, then instrumentation must be present and active when the peripheral is attached.

Remote New System Setup

This Baseline recommends remote new system setup for mobile computers. The remote new system setup capability provides a standard means to boot a PC from the network to perform new machine configuration or diagnostic and repair tasks that are difficult or impossible otherwise. Remote new system setup is supported while the mobile client is connected to the LAN.

The client side of remote new system setup on mobile computers may be provided via any one (or more) of the following ways:

- Special boot diskette (or other removable bootable media)
- System BIOS code
- NIC Adapter ROM code

Since remote new system setup interacts with the NIC (network interface card) prior to OS boot the code is of necessity specific to a particular NIC. Depending on how the NIC is attached to the notebook (dynamic attachment such as CardBus and docking station or soldered down to the notebook motherboard) further code dependencies may exist. For example, if the NIC is attached via PC Card slot and remote new system setup is supported in system BIOS, the code should:

1. Power ON the slot
2. Initialize the slot controller
3. Read the tuples to determine that the correct NIC is inserted
4. Handle the case where the NIC is in any slot of a multi-slot implementation
5. If the correct NIC is not found or the NIC is not cabled to the LAN then a message would be displayed to prompt the user to correct the situation and retry

Remote Wake-up

Remote wake-up enables client PCs to be awakened from a low power sleep state so that system maintenance can be performed across a wire.

Remote wake-up assumes that managed PCs are always on the desktop connected to the LAN day and night, which is not the normal case with most mobile PCs [20]. Therefore, remote wake-up is recommended, not required, for mobile PCs. Mobile professionals typically take their notebooks with them (e.g. home, airplane, hotel room) or safely lock them away as opposed to leaving them on their desktops overnight and on weekends connected to the LAN. For this reason remote wake-up is recommended for mobile PCs.

If remote wake-up is supported on a mobile computer the mobile usage pattern must be temporarily altered (i.e. by requesting that the mobile user leave their notebook at the office connected to the LAN). Changing the mobile usage pattern is a policy task that IT organizations must undertake. Note that notifying mobile professionals when off-hours maintenance is scheduled does not guarantee that all notebooks will be maintained since some will always be out of the office on road trips.

It is recommended that mobile PC vendors provide a way to secure the notebook when it is left on the desktop for manageability after hours. Restraint cables and docking stations with password lock are sufficient for this purpose.

Depending on the attachment type of the NIC used (e.g. PC Card), powering it may require special attention. For example, it may be helpful to require the user to configure the PC Card slot to remain powered when a remote wake-up is expected and the notebook is being suspended rather then assuming so and running down the battery.

Power Management

Mobile systems conforming to the Wired for Management Baseline must have the capability to transition in and out of low power states as required to perform useful work and conserve power. The WfM Baseline requires that mobile computers conform to the ACPI (Advanced Configuration and Power Interface) specification.

Detailed technical information on Power Management requirements is given in the Power Management section of the WfM Baseline.

Proposed Extensions

Alert Queuing

Instrumentation alerts provide the means to communicate state changes or unusual conditions to higher level software for further processing. Because mobile PCs are occasionally disconnected, alerts can be lost. A future extension to the Baseline will recommend that if the mobile is not connected to the network at the instant an alert occurs that the alert be queued until such time as the connection is re-established. Then the alert will be forwarded to the

[20] The notable exception is where due to their small footprint and low power consumption, notebooks are used on the desktop in a stationary state and never moved..

managing application. This future revision of WfM Baseline will define which alerts to queue, how to forward them, and when to remove them from the queue.

Remote New System Setup & Remote Wake-up Security

BIOS passwords present a problem when remotely managing PCs. Some notebooks incorporate a password protected hard disk drive (HDD) that requires the BIOS password to be entered before data can be read or written. Since the mechanism for BIOS password entry requires physical keystrokes to be entered at the client's keyboard, this presents a problem for remote access. The current workaround is to disable the BIOS password when remote service is expected. Future versions of the Baseline will define a mechanism for remote authentication which allows the client to remain protected while awaiting management and enable the password protected HDD to be unlocked across a wire.

Non-LAN Transports

Futures versions of the Baseline will also address extending remote new system setup and remote wake-up to non-LAN transports which are typically used for mobile connectivity when out of the office.

Section
7

Server Systems

Category Description

The Wired for Management Baseline defines a targeted minimum set of management features for desktop PC platforms. This section outlines how the technologies addressed by the Wired for Management 1.1 baseline apply to server systems. A large array of additional management technologies are in wide spread use on server systems. These technologies are designed to assure that servers meet the reliability, serviceability and availability requirements appropriate to the business environments where servers are deployed.

Server systems differ from desktop business computers in a number of important ways. These differences require that the Wired for Management Baseline capabilities be implemented in a distinct manner on servers, and also presents the need for additional categories of requirements unique to servers. This section of the Baseline describes the manageability requirements for server systems and outlines the extensions needed in the future to keep server systems universally manageable and universally managed.

It is not the intent of this release to address all of the management requirements of server systems. Future versions of this Baseline will specify requirements which are more specific to the management of servers, including capabilities to support reliability, availability and serviceability goals, and integration into large enterprise environments.

Motivation

In client/server environments, the ability of the users of client systems to accomplish their tasks is directly impacted by the availability of critical services exported by servers in the computing environment: examples include file and print services, electronic mail and messaging servers, Internet gateways, routing services, and business-critical application servers. Servers have traditionally supported advanced manageability features, focusing on remote management from centralized administrative consoles, failure detection and critical event notification, and diagnostic tools. In addition, servers frequently are configured with redundant subsystems, so that the server can tolerate the failure of critical components such as fans, power supplies, and disks. Redundant subsystems may support hot-plug (hot-swap) connectors, so that repairs can be performed without bringing the server off-line. Servers are also configured for scaleable performance and capacity; multiple processors, high performance I/O peripherals, multi-chassis configurations, intelligent I/O processors, and rack-mounted chassis all present additional management requirements.

This chapter outlines how the technologies addressed by the Wired for Management 1.1 Baseline apply to server systems. An array of additional management technologies are in wide spread use on server systems. The technologies in this baseline are adjunctive to these

technologies, and are intended to create a common minimum infrastructure shared by desktop, mobile and server platforms when each is the object of management activities.

What Makes Servers Different

The priorities for Server Management are quite different from those for Client (desktop or mobile) Management. While the major goal for Client Management is to reduce per-client maintenance and administration costs, the top priority for servers tends to be minimizing down time. The failure of a server, or of a service hosted by a server, can impact many clients simultaneously.

The management technologies which are required to support many of the server characteristics described in the following three subsections will be addressed in future versions of this Baseline.

Reliability, Availability, Serviceability

The Total Cost of Ownership of the server is measured by metrics associated with Reliability, Availability and Serviceability (RAS). Server management mechanisms focus on providing fault prediction, fault detection and fault resilience. If a failure does occur, Server Management strives to provide rapid servicing of the problem, since the Mean Time to Repair directly affects the overall availability of the server. Rapid restoration of the server to full service usually requires remote notification when the problem occurs, information that identifies the FRU (Field Replaceable Unit) that failed, and, if possible, remote mechanisms for reconfiguration and recovery. In addition, servers frequently support automatic recovery mechanisms which allow the server to return to operation after a failure without external intervention; for example, servers may support an automatic server restart capability. Many of these services are accessed and controlled via instrumentation interfaces.

Servers frequently support redundant and hot-swap hardware subsystems. Instrumentation needs to be smart enough to handle individual devices which dynamically appear and disappear.

Remote Management

High quality remote management is also a high priority. Most client systems will have a user who, if necessary, can intervene to take local actions such as typing at the keyboard, warm or cold reset, cycling system power, etc., whereas a Server will usually be isolated from local user intervention - both logically and physically. Therefore, servers are usually managed using a *push model*, where management sessions are invoked by the remote administrator.

Remote Management provides the interfaces for retrieving system health information, delivering problem alerts, and driving system recovery functions (such as system power cycling, resetting, and reconfiguration). Remote Management can also provide interfaces by which additional management software, such as remote diagnostics, can be loaded.

The primary communication channel for remote server management may be WAN, LAN or modem connections. For example, modem connections are common in small office or branch office environments. Servers therefore share some characteristics with mobile clients, in that their connection type can vary, and the bandwidth and latency available for remote management can vary greatly.

Emergency and Preboot Management

Emergency Management is also a high priority in Servers. Emergency Management in this case refers to mechanisms that allow Remote Management to occur under conditions where the normal remote communications links have failed. This may be because the server is powered down, or because a hardware or software failure is preventing the Operating System from running or making a remote connection.

Emergency Management is often linked with a technology referred to as *Pre-boot Management.* Pre-boot Management is a set of interfaces that provide communication interfaces for system configuration and management without requiring an operational Operating System. These functions include running system configuration software, accessing system management information logs, and downloading and re-installing the Operating System and platform firmware/BIOS.

Emergency Management will often provide a secondary communication channel, used for remote management communications when the normal communication interfaces are inoperative. The secondary channel can be implemented using an alternate network connection, telephone line, or other communication media. The robustness of the implementation is judged on its degree of independence from the primary communication physical interfaces and software.

Server Baseline Overview

When applying the Wired for Management Baseline v1.1 capabilities to servers, the requirements are:

- **Instrumentation** - ***Required***. Servers must be instrumented to be well managed, supporting the retrieval of configuration information, the monitoring of key subsystems and the proactive communication of critical events. While a single standard instrumentation interface is desirable (to be addressed in future versions of this standard), currently deployed systems use both DMI and SNMP instrumentation. Version 1.1 WfM-compliant servers must use either DMI or SNMP to provide server instrumentation.

  - DMI Requirements: The set of DMI standard groups required for compliant servers are listed in the detailed specification section below. Servers incorporate the same requirements as desktop PCs with the addition of standard groups relevant to server hardware configurations. Additionally, since dynamic devices are an important facet of many server configurations, any instrumentation which is provided for those devices should accommodate their dynamic nature and be available whenever the device is present.

  - SNMP Requirements: The server system should support SNMPv1 or above. If SNMP is supported, the standard MIB-II (IETF RFC 1213) and Host Resources MIBs (IETF RFC 1514) are required. As with DMI instrumentation, if hot pluggable devices are instrumented, then that instrumentation should be available whenever the device is present.

- **Remote New System Setup** - ***Recommended.*** Support may be provided via system BIOS code, option ROM on NIC adapter card, or special boot diskette.

- **Remote Wake-up** - ***Optional.***. Servers rarely require or attain power managed sleep states. However, remote wake-up combined with remote power down can be useful at sites where power conservation dictates administrative power down during off-hours.

- **Power Management** - ***Recommended***. ACPI compliant hardware and BIOS strongly recommended.

Detailed technical information on the requirements is in Chapters 2 through 5 and the next section of this chapter.

Future revisions of this Baseline will specify requirements which are more specific to the management of servers, including capabilities to support reliability, availability and serviceability goals.

Detailed Specification

Instrumentation

Servers must comply with the instrumentation requirements outlined in Chapter 2 of this document. However, the DMI Standard Groups subsection under the heading **Detailed Specification** in Section 2 should be replaced by the groups and implementation guidelines listed in Table 7-1 through Table 7-5 below. Table 7-1 through Table 7-4 contain server guidelines for groups described in Section 2. Table 7-5 contains groups specifically related to servers.

Table 7-1. DMI System Standards Groups

DMI Standard Group	***Implementation Guidelines***
DMTF|ComponentID|001	Required
DMTF|Disk Mapping Table|001	Recommended (see Note)
DMTF|Disks|002	Recommended (see Note)
DMTF|General Information|001	Required
DMTF|Keyboard|003	Optional
DMTF|Mouse|003	Optional
DMTF|Operating System|001	Required
DMTF|Partition|001	Recommended (see Note)
DMTF|Physical Container Global Table|001	Required
DMTF|Processor|003	Required
DMTF|System BIOS|001	Required
DMTF|System Cache|002	Required
DMTF|System Slots|003	Required
DMTF|Video BIOS|001	Optional
DMTF|Video|002	Optional

Note for Table 7-1. Mass storage management is very important for servers. The Mass Storage MIF (to be published by the DMTF shortly) addresses the management of server-class storage. The Mass Storage MIF supersedes the Disk Mapping Table, Disks and Partition groups, therefore these groups are recommended rather than required for servers.

Table 7-2 and Table 7-3 contain standard groups related to system resource management and physical memory management (drawn from *System Standards Group Definition, Approved Version 1.0* [3]). All of these groups are valid standard groups, but the DMTF has designated

some groups as 'Obsolete' and is recommending that instrumentation migrate to the designated 'Replacement' groups. To comply with this Baseline specification, a system must be instrumented with either **all** of the "Obsolete" groups or **all** of the corresponding "Replacement" groups. It is highly recommended that the "Replacement" groups be selected for newly implemented instrumentation.

Table 7-2. Required System Resource Management Standard Groups

DMI Standard Group	*Implementation Guidelines*
DMTF|System Resource 2|001	For new instrumentation
DMTF|System Resource Device Info|001	
DMTF|System Resource DMA Info|001	
DMTF|System Resource I/O Info|001	
DMTF|System Resource IRQ Info|001	
DMTF|System Resource Memory Info|001	
DMTF|System Resources|001	Obsolete, for legacy instrumentation only
DMTF|System Resources Description|001	

Table 7-3. Required Physical Memory Management Standard Groups

DMI Standard Group	*Implementation Guidelines*
DMTF|Memory Device|001	For new instrumentation
DMTF|Memory Array Mapped Addresses|001	
DMTF|Memory Device Mapped Addresses|001	
DMTF|Physical Memory Array|001	
DMTF|Physical Memory|002	Obsolete, for legacy instrumentation only

The following standard groups from the *LAN Adapter Standard Groups Definition, Release Version 1.0* [21][4], must be instrumented and deployed on systems compliant with this specification:

Table7-4. LAN Adapter Standard Groups

DMI Standard Group	*Implementation Guidelines*
DMTF|Network Adapter 802 Port|001	Required
DMTF|Network Adapter Driver|001	Required

Table 7-5 contains standard groups related to server systems (drawn from *System Standards Group Definition, Approved Version 1.0* [3]). Groups marked *Required if supported by platform* should be instrumented if the hardware capabilities of the platform support management of the respective devices.

[21] LAN Adapter Standard Groups Definition, Release Version 1.0. http://www.dmtf.org/tech/apps.html

Table 7-5. Server-Related Standard Groups

DMI Standard Group	*Implementation Guidelines*
DMTF|Cooling Device|002	Required if supported by platform
DMTF|FRU|002	Required
DMTF|Operational State|002	Required
DMTF|Power Supply|002	Required
DMTF|Power Unit Global Table|001	Required
DMTF|System Hardware Security|001	Required
DMTF|System Power Controls|001	Required if supported by platform
DMTF|Temperature Probe|001	Required
DMTF|Voltage Probe|001	Required if supported by platform

Dynamic Device Support

Instrumentation for Server platforms should support dynamic peripherals that appear/disappear (e.g. hot insertion/removal of fans/blowers, power supplies, drive swap bays). If dynamic instrumentation is provided, then re-enumeration of dynamic peripherals must be done without requiring the system to reboot. If a peripheral is provided as part of the server platform and that peripheral is encompassed in the standard groups listed in Table 7-1 through Table 7-5 then instrumentation must be present and active when the peripheral is attached.

Proposed Baseline Extensions

Future revisions of this baseline will address migration to the DMTF Mass Storage MIF.

Remote New System Setup

The remote new system setup capability is recommended on server platforms compliant with this Baseline.

The remote new system setup capability provides a standard means for a remote system setup client to request boot services from the network to perform new machine configuration, operating system install, and diagnostic and repair tasks. Note that during the new system setup process, the target server system is in the role of client to the remote new system setup provider.

Remote new system setup on a server may be accomplished in one or more of these ways:

- System BIOS code
- NIC Adapter option ROM for add-in NIC cards
- Special boot diskette

Since remote new system setup interacts with the NIC (network interface chip or card) prior to OS, the code is of necessity specific to a particular NIC. Servers may support multiple NIC add-in cards, or a combination of a NIC on the motherboard and add-in cards. It is preferable that the implementation be robust enough to account for the presence of multiple NICs. If the correct NIC is not found by the remote new system setup code or the NIC is not cabled to the LAN, a message should be displayed to prompt the user to correct the situation and retry.

In addition to OS loads, Service Boot may be most useful on servers for loading and booting diagnostics and repair utilities. The methods employed on the server to give control to the Remote New System Setup code under *push* conditions are not specified in this baseline.

"Classic" Remote Boot is not a requirement for servers. It is commonly associated with enabling diskless client PCs connected to a server over a LAN and is not feasible for servers.

Proposed Baseline Extensions

The Remote New System Setup capability will be required for server platforms in future versions of this Baseline.

Remote new system setup assumes new systems are being deployed into a LAN environment, therefore setup while unconnected to a LAN is not supported. It is very common for server systems to support serial ports and/or modems as secondary management access paths, for preboot management of the server.

The use of the new system setup protocol to perform remote emergency boots on servers is very desirable. When a server fails to boot its installed OS due a hardware or software failure, the protocol can be used to download an alternate runtime environment. The environment can in turn support remote problem diagnosis, system reconfiguration, and a rapid return to operational status. This version of the Baseline covers only the case of remote new system setup occurring in response to a client request (*pull model*). Since servers are "user-less", frequently locked in labs or closets, and sometimes even "headless" (configured without keyboard or monitor), a *emergency push* model is required for remote server management. Security issues must be addressed as part of the emergency push solution.

Future versions of this Baseline will address these requirements.

Detailed information on Remote New System Setup requirements is in Chapter 3.

Remote Wake-up

The Remote Wakeup capability is optional on server platforms compliant with this Baseline.

Due the need for servers to be continuously available to client systems, servers rarely require or attain power-managed sleep states. An exception is sites where power conservation dictates the administrative power down of all systems during off-hours. For these sites, power management combined with remote wakeup can be very useful.

Proposed Baseline Extensions

The *emergency push model*, described in the Remote New System Setup section above, requires the system setup provider to force the targeted system into the preboot execution state. Emergency push support requires new remote commands or packets to reboot or reset the server, analogous to the remote wakeup packets. These potentially destructive commands in turn require secure authentication and authorization to prevent unauthorized access to the server. In addition, care must be taken to coordinate the remote reboot/reset requests with the server's operating system, so that graceful shutdown of the server occurs whenever possible. Non-LAN (e.g. modem) transports for these capabilities is also important.

Future versions of this Baseline will address these requirements.

Detailed technical information on Remote Wakeup requirements is in Chapter 4.

Power Management

Server requirements for power management are as the same as those for desktop PC platforms. Advanced Configuration and Power Interface (ACPI) compliant hardware and BIOS components are strongly recommended on server systems where possible.

Servers are not required to provide power management beyond the minimum required for ACPI compliance; see Chapter 1 of the ACPI specification for a list of minimum requirements. Note that server operating systems may choose to implement different power management policies for servers than those implemented for desktop or mobile platforms.

Proposed Baseline Extensions

ACPI compliance for both server platforms will be required in future versions of this Baseline.

Detailed technical information on Power Management requirements is in C

Section 8

Information and Resource References

For background on the **Wired for Management Initiative**: **http://www.intel.com/managedpc**

For **Web Based Enterprise Management (WBEM)**: **http://wbem.freerange.com**
The specifications upon which the WBEM schema are based can be found on the DMTF site under the heading of the Common Information Model:
http://www.dmtf.org/work/cim.html

Section 2 contains references to the following **DMTF documents on DMI**:

[1] Desktop Management Interface Specification, Version 2.00, March 27, 1996, Desktop Management Task Force, Inc. **http://www.dmtf.org/tech/specs.html**

[2] Desktop Management Interface Compliance Guidelines, Version 1.0, September 11, 1995, Desktop Management Task Force, Inc. **http://www.dmtf.org/tech/specs.html**

[3] Systems Standard Group Definitions, Approved Version 1.0, May 1, 1996, Desktop Management Task Force, Inc. **http://www.dmtf.org/tech/apps.html**

[4] LAN Adapter Standard Groups Definition, Release Version 1.0, October 4, 1994, Desktop Management Task Force, Inc. **http://www.dmtf.org/tech/apps.html**

[5] Desktop Management Task Force: Enabling your product for manageability with MIF files, Revision 1.0, November 1994, Desktop Management Task Force, Inc. **http://www.dmtf.org/tech/apps.html**

To contact the DMTF:

Desktop Management Task Force
M/S JF2-53
2111 N.E. 25th Avenue
Hillsboro. OR 97124
Phone: (503) 264-9300
Fax: (503) 264-9027
Email: dmtf-info@dmtf.org

For a copy of the **ACPI specification**: **http://www.teleport.com\~acpi**

For more information on **AMD's Magic Packet™** technology:**http://www.amd.com/products/npd/overview/20212d.html.**

Section 9

Terms and Acronyms

This section describes briefly the platform management information technologies referenced in these guidelines and gives references to full definitions and descriptions of these technologies.

Common Information Model (CIM). CIM is the management information schema for WBEM. It is an object-oriented schema that is being defined by a subcommittee of the Desktop Management Task Force (DMTF). CIM is designed to be extended for each operating environment in which it is used. For CIM specifications, see http://www.dmtf.org/work/cim.html.

Desktop Management Interface (DMI). DMI is a platform management information framework created by the Desktop Management Task Force (DMTF). DMTF specifications define industry-standard interfaces for management of desktop and server computing platforms.

For specifications on DMI Management Information Interfaces, see *Desktop Management Interface Specification, Version 2.00*. For specification of DMI Component Interfaces, see *Desktop Management Interface Specification, Version 2.00*. Compliance of PC platforms to the DMI specifications referenced in these Guidelines is measured by *DMI Compliance Guidelines, Version 1.0*.

Instrumentation. A mechanism for reporting information about the state of PC hardware and software to enable management applications to understand and change the state of a PC and to be notified of state changes.

Simple Network Management Protocol (SNMP). SNMP is used widely throughout the industry as the standard under which servers, routers, hubs, and other network-based devices are managed. Enterprise-level management applications have long used SNMP as their manageability protocol because of its stability, flexibility, and wide-spread adoption.

Web-Based Enterprise Management (WBEM). WBEM is a set of platform management information technologies originally proposed by BMC Software, Inc., Cisco Systems, Inc., Compaq Computer Corporation, Intel Corporation, and Microsoft Corporation, based on standards being developed in a number of industry bodies, including the Desktop Management Task Force (DMTF) and the Internet Engineering Task Force (IETF). WBEM is being designed to provide uniform access for management applications to management information from a variety of sources, such as DMI, SNMP, and operating system-specific component instrumentation. For specifications on WBEM, see http://wbem.freerange.com.

Win32 Driver Model (WDM). A driver model based on the Windows NT driver model that is designed to provide a common set of I/O services and binary-compatible device drivers for both Windows NT and future Windows operating systems for specific classes of drivers. These driver classes include USB and IEEE 1394 buses, audio, still-image capture, video capture, and HID-compliant devices such as USB mice, keyboards, and joysticks.

Windows Management Instrumentation (WMI). WMI is an extension to WDM and is a new component instrumentation approach for Microsoft operating systems, Windows NT 5.0 and Windows "Memphis". WMI drivers have their schema built into the driver image as a resource, which enables dynamic "import" of specific driver schema data into the CIM schema.

"Wired for Management" (WfM). Intel's broad based initiative to reduce the TCO of business computing while maintaining the power and flexibility of high-performance computing.

Appendices

Appendix A: DHCP Extensions for New System Setup

This description assumes a knowledge of the standard DHCP/BOOTP protocols.

Protocol Overview

The protocol is a combination of a straightforward extension of DHCP (through the use of several new DHCP Option tags) and the definition of simple packet transactions which 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 DHCP request packets sent by a client as part of this extended protocol from other DHCP request packets that the installation server may receive.
- Distinguish DHCP reply packets sent by a server as part of this extended protocol from other DHCP reply packets that the client may receive.
- Convey client network adapter type.
- Convey client System ID.
- Convey client system architecture type.

Based on the client network adapter type and system architecture type, the 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. How this executable accomplishes the setup of the system is not specified by these guidelines.

This section presents an informal, step-by-step description of the remote new system setup protocol. A detailed description of packet formats and client and server actions appears later in this attachment.

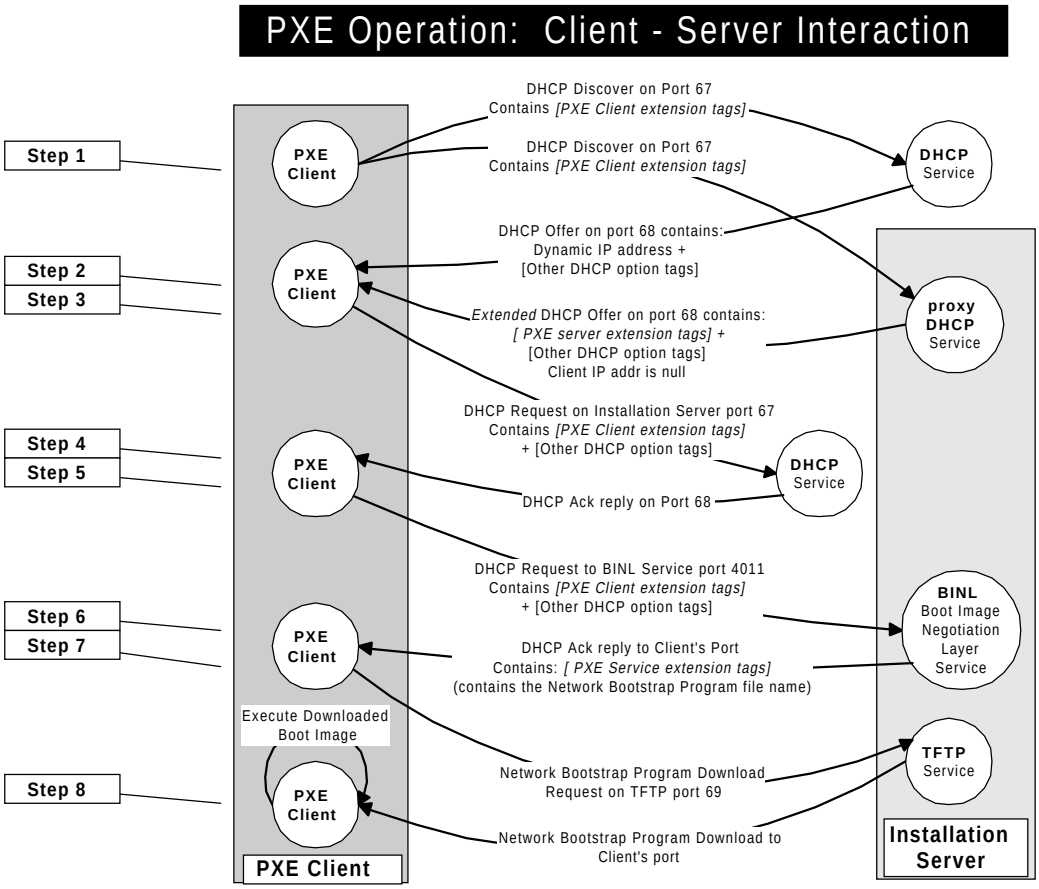


Figure 1 – PXE Operation – Client Server Interaction (updated in version 1.1a)

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

- A tag for client identifier (if the client identifier is known).
- A tag for the client Network Interface Identifier.
- A tag for the client system architecture.

Step 2. The PXE PROXY DHCP Service responds by sending a PXE PROXY DHCPOFFER message to the client on the standard DHCP reply port (68). This packet contains the address of the PXE PROXY DHCP Service. The client IP address field is null.

At this point, other DHCP Services and BOOTP Services also respond with DHCP offers or BOOTP reply messages on 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 Service. If the Installation Server is also functioning as a standard DHCP Service, then the DHCP Service reply from the Installation Server will also contain standard DHCP parameters (in particular, an IP address for the client)

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 OFFER with PXE extensions. The PXE timeout for rebroadcast is:

4, 8, 16 seconds, yielding three broadcasts and a timeout after 28 seconds.

The PXE timeout for rebroadcast is 4 seconds after receiving an OFFER without PXE extensions but with a valid “bootfile name” option.

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 Server IP address of the BINL (Boot Image Negotiation Layer) Service from the “siaddr” field in the PXE proxy DHCP offer.

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 go ahead and use the address.

Step 5. The client sends a DHCP Request packet to the BINL Service on port 4011. This packet is exactly the same as the initial DHCP Discover in Step 1, except that it is coded as a DHCP Request and now contains the following:

- Contains the IP address assigned to the client from a DHCP Service.
- Contains all the PXE options fields received from the selected DHCP Offer which contained the PXE options.

Step 6. The BINL Service on the Installation Server sends a DHCP Acknowledge packet back to the client, also on port 4011. This reply packet contains:

- Boot file name and location.
- The Client UUID/GUID option in the PXE proxy DHCP offer.
- MTFTP[22] configuration parameters.

Step 7. The client downloads the executable file using either standard TFTP or MTFTP. The file downloaded and the placement of the downloaded code in memory is dependent on the client’s CPU architecture. (For Intel architecture Network PCs, see Attachment B.)

Step 8. Finally, the PXE Client initiates execution of the downloaded code. The way in which this is done is dependent on the client’s CPU type. For Intel architecture Network PC systems, the client code executes a far call to the first location in the code.

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 an installation server, that is, one that delivers executables for new system setup. The client *can* 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 1541, even though not all possible messages and states of that protocol

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

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 an installation 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.

PXE DHCP Options

Tag Name	*Tag Number*	*Length*	*Type*	*Data Field*
Client UUID/GUID	97	17	20	*per Attachment K*
Client Network Interface Identifier	94	3-9	1 = UNDI 2 = PCI 3 = PNP	Type 1 = Major ver(1), Minor Ver(1) Type 2 = Vendor ID(2), Device ID(2), Class Code(3), Rev(1) Type 3 = EISA Device ID(4), Class Code(3)
Tag Name	*Tag Number*	*Length Field*		*Data Field*
Client System Architecture	93	2		0 = Intel Architecture PC 1 = NEC/PC98 2 = etc.
Class Identifier	60	9		"PXEClient"

Encapsulated Vendor Options (DHCP Option #43)

Tag Name	*Tag Number*	*Length Field*	*Data Field*
DHCP_VENDOR	43	varies	Encapsulated options below
(Multiple DHCP_VENDOR options can be used)			

"PXEClient" Encapsulated Options for DHCP Option #43

Tag Name	*Tag Number*	*Length Field*	*Data Field*
PXE_PAD	0	None	None
MTFP IP Addr	1	4	a0, a1, a2, a3
MTFTP Client UDP	2	2	Port Number
MTFTP Server UDP	3	2	Port Number
MTFTP Start Delay	4	1	
MTFTP Timeout Delay	5	1	
Reserved	6-63	1	
Loader Options	64-127	1	(vendor specific)
Vendor Options	128-254	1	(vendor specific)
PXE_END	255	None	

The *Client UUID/GUID* field specifies a globally unique ID (GUID), retrieved from the client system. Client UUID/GUID must be generated per Attachment K. If the Client does not have a GUID, the DHCP service (or Installation Service) may supply one by returning the option with a valid value. The client must store this value if it can, and if so, must use it in all subsequent DHCP transactions.

The *Client Network Interface Identifier* specifies either the physical Network Interface Adapter or indicates the presence of an API (UNDI, described below) that will support a universal boot loader. The UNDI interface should be supported.

The UNDI type field must have a major version of 2 and a minor version of 0 for this version of the protocol. (Future versions may recognize more tags based on this version number.)

If neither PCI nor PNP information is available then the UNDI interface should be supported. Otherwise, the vendor must create an ad hoc PNP or PCI entry and assume the responsibility of distributing the appropriate NIC driver to PXE Servers.

The *Client System Architecture* identifier specifies the system architecture of the client.

The *Class Identifier* (Option 60) of "PXEClient" is required to assure unambiguous identification of clients meeting this specification.

Encapsulated Options (Option 43) are provided to allow configuration for MTFTP boot file transfers. MTFTP should be implemented in the Client. If provided by the DHCP, proxyDHCP, or BINL service, these options should be used.

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.

Client Behavior

This section summarizes client behavior for initiation, discovery reply, installation service request, installation service reply, and executable download and execution.

Sending a PXE (Preboot eXecution Environment) Client message requires the use of DHCP Option fields. All PXE Client packets provide the same extended DHCP information in these options. This includes DHCP Request 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.

Initiation

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 1541 for a DHCPDISCOVER message, with the addition of PXE Client option fields: The format of these options is specified below:

DHCP Header		
Field (length)	*Value*	*Comment*
op (1)	1	Code for BOOTP BOOTREQUEST
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)	blank	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		

DHCP Options				
Tag Name	*Tag Number*	*Length*	*Type*	*Data Field*
Client UUID/GUID	97	17	20	*per Attachment K*
Client Network Device Interface Type	94	3-9	1 = UNDI 2 = PCI 3 = PNP	Type 1 = Major ver(1), Minor Ver(1) Type 2 = Vendor ID(2), Device ID(2), Class Code(3), Rev(1) Type 3 = EISA Device ID(4), Class Code(3)

Tag Name	*Tag Number*	*Length Field*	*Data Field*
DHCP Message Type	53	1	1 = DHCPDISCOVER
Class Identifier	60	9	"PXEClient"
Client System Architecture	93	2	0 = Intel Architecture PC, 1 = NEC/PC98

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

Discovery Reply

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 are as follows:

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	PXE client 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 Number*	*Length*	*Type*	*Data Field*
Client UUID/GUID	97	17	20	*per Attachment K*

Tag Name	*Tag Number*	*Length Field*	*Data Field*
DHCP Message Type	53	1	2 = DHCPOFFER
Server Identifier	54	4	a1, a2, a3, a4
Class Identifier	60	9	"PXEClient"

Encapsulated Vendor Options (DHCP Option #43)

Tag Name	*Tag Number*	*Length Field*	*Data Field*
DHCP_VENDOR	43	varies	Encapsulated options below
(Multiple DHCP_VENDOR options can be used)			

"PXEClient" Encapsulated Options for DHCP Option #43

Tag Name	*Tag Number*	*Length Field*	*Data Field*
PXE_PAD	0	None	None
MTFP IP Addr	1	4	a0, a1, a2, a3
MTFTP Client UDP	2	2	Port Number
MTFTP Server UDP	3	2	Port Number
MTFTP Start Delay	4	1	
MTFTP Timeout Delay	5	1	
PXE_END	255	None	

If the responding server does not have an installation capability, it will provide a valid address in *siaddr* to redirect the client to an installation server.

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 1541. Each *extended* DHCPOFFER message can also contain configuration information as specified in RFC 1541. The presence of such information in an *extended* DHCPOFFER message is indicated by a nonzero value in the client IP address field. Which, if any, of these configurations 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 1541.

To move to the installation server request state, the client must have received at least one *extended* DHCPOFFER message. Beyond this, the criteria for the client exiting this state are not defined by this specification.

Installation Service Request

To enter this state, the client must have an IP address. Also, the client must have received one or more *extended* DHCPOFFER messages and therefore know the IP address of one or more installation servers. The client selects one of these installation servers and sends a DHCPREQUEST message to the server on port 4011. Otherwise the format of this message is the same as an *extended* DHCPDISCOVER. The following table lists the required values in the fields of this message; fields marked with an asterisk contain unspecified values.

DHCP Header		
Field (length)	*Value*	*Comment*
op (1)	1	Code for BOOTP BOOTREQUEST
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)	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 Number*	*Length*	*Type*	*Data Field*
Client UUID/GUID	97	17	20	*per Attachment K*
Client Network Device Interface Type	94	3-9	1 = UNDI 2 = PCI 3 = PNP	Type 1 = Major ver(1), Minor Ver(1) Type 2 = Vendor ID(2), Device ID(2), Class Code(3), Rev(1) Type 3 = EISA Device ID(4), Class Code(3)

Tag Name	*Tag Number*	*Length Field*	*Data Field*
DHCP Message Type	53	1	3 = DHCPREQUEST
Server Identifier	54	4	a1, a2, a3, a4
Class Identifier	60	9	"PXEClient"
Client System Architecture	93	2	0 = Intel Architecture PC 1 = NEC/PC98.

Installation Service Reply

In this state, the client must be prepared to receive an *extended* DHCPACKNOWLEDGE message from the installation server. The following table lists the required values in the fields of this message:

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	PXE client 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 Number*	*Length*	*Type*	*Data Field*
Client UUID/GUID	97	17	20	*per Attachment K*
Tag Name	*Tag Number*	*Length Field*		*Data Field*
DHCP Message Type	53	1		4 = DHCPACKNOWLEDGE
Server Identifier	54	4		a1, a2, a3, a4
Encapsulated Vendor Options (DHCP Option #43)				
Tag Name	*Tag Number*	*Length Field*		*Data Field*
DHCP_VENDOR	43	varies		Encapsulated options below
(Multiple DHCP_VENDOR options can be used)				
"PXEClient" Encapsulated Options for DHCP Option #43				
Tag Name	*Tag Number*	*Length Field*		*Data Field*
PXE_PAD	0	None		None
MTFP IP Addr	1	4		a0, a1, a2, a3
MTFTP Client UDP	2	2		Port Number
MTFTP Server UDP	3	2		Port Number
MTFTP Start Delay	4	1		
MTFTP Timeout Delay	5	1		
PXE_END	255	None		

The options fields in this message must include the following:

- The Installation Server must direct the client to a TFTP server by responding with *siaddr* filled in. (Usually, the TFTP Server resides on the same machine, so *siaddr* would be set to null.)
- Client UUID/GUID if received from the client
- Server Identifier (address of the responding Installation server)
- Bootfile Name if Option 52 (Option Overload) is used.
- *System architecture* indicating architecture the *bootfile* supports. *System architecture* value must be the same as received by the client to insure proper operation of the *bootfile*.

After receiving this message, the client moves to the executable download state.

Executable Download and Execution

In this state, the client is to download all or some portion of the executable file 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 Network PC systems based on Intel Architecture, the entire bootstrap image (up to 32K in size) is downloaded into the client PC starting at location 07C00h. The TFTP/MTFTP session that was used to download the bootstrap image is terminated and the logical network connection to the TFTP server is closed.

After the bootstrap image is downloaded, the TFTP connection is closed and control is passed to the bootstrap image. The way in which this is done is dependent on the client's CPU type. For Network PC systems based on Intel Architecture, the boot ROM code is to execute a far call to location 0:7C00h.

MTFTP Operation

Implementation of MTFTP in the client is strongly recommended. If the server sends MTFTP parameters, then the client should proceed as described in this section. In this case the client goes through three phases: an open, a receive and a close, with an error recovery phase that can be entered at any point.

MTFTP open

1. The network client acquires at least the following information from the BINL reply:

   - Client bootfilename
   - MTFTP Server UDP port number
   - MTFTP Client UDP port number
   - MTFTP multicast IP address
   - MTFTP transmission start delay
   - MTFTP transmission time-out delay

2. Client binds to the MTFTP UDP port and waits for the MTFTP transmission start delay to receive packets. No network traffic is generated.

3. If there is a response, MTFTP packets are collected from the network. The client keeps track of received packets in an internal list.

If no packet is received, the client initiates an MTFTP open to the server.

MTFTP receive

1. In order to find out if a client needs to acknowledge or not, the server sends a unicast TFTP packet to that client. The first packet of a MTFTP transmission is always sent both as unicast and multicast UDP/IP. This instructs the network client that it is the acnowledging client.

2. A server always transmits the complete file. Therefore, clients that start listening to a conversation part way through can wait and then get the rest on the next MTFTP transmission to make up for what was missed the first time.

3. The acknowledging client must ACK all packets even if the client has received the entire file.

MTFTP close

1. An MTFTP transmission is finished when the acknowledging client has received all packets and disconnects from the network. Clients who did not receive all packets can initiate a new transmission, if one has not already started.

2. Before a new transmission is started there is a calculated delay. The default delay is modified by an algorithm based on the number of packets received. Clients who received fewer packets will wait for a shorter time than those who received more. This algorithm ensures that:

   - Slow clients define the transmission speed.
   - Clients with a large number of received packets can disconnect from MTFTP after they received all missing packets.
   - Clients who hook into an ongoing MTFTP transmission and therefore only receive the tail of the transmission can disconnect from MTFTP after they received the missing head of the transmission.
   - Clients with a small number of received packets are more likely to become the acknowledging client.

Server Behavior

The server behavior needed for the extended protocol comprises two pieces of functionality: a redirection service, and an installation 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 the location (IP address) of the installation service.
- The installation service receives extended DHCPREQUEST messages (generated by the client Installation Service Request step) on UDP port 4011 and responds with DHCPACK messages containing the location (IP address) of the TFTP service and the file name of a new system setup executable appropriate to the client.

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

Redirection Service Behavior

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

Response to DHCPDISCOVER

The redirection service will always be prepared to receive on UDP port 67, an extended DHCPDISCOVER message with contents as described earlier in the “Initiation” section. The redirection service will only respond to messages which include DHCP Option 60 with the value of “PXEClient”.

If the redirection service responds to a message, it will respond by sending to the initiating client a DHCPOFFER message containing options as described earlier in the “Discovery Reply” section:

The “siaddr” field in the reply, if filled in, will be the location of an installation service. If the “siaddr” field is not filled in then the installation service is at the same address as the redirection service.

The client IP address field of the message will be 0.0.0.0.

If the redirection service is also a standard DHCP configuration service, then the DHCPOFFER message sent to the client will be as specified in RFC 1541.

Installation Service Behavior

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

Response to DHCPREQUEST

The installation service will always be prepared to receive a DHCPREQUEST message with contents as described earlier in the “Installation Service Request” section. The installation service will respond by sending to the initiating client a DHCPACKNOWLEDGE message as described earlier in the “Installation Service Reply” section. The file name in this message will be the complete path name of a new system setup executable appropriate to the client that is accessible using TFTP from the installation server’s IP address.

TFTP Service

The server running the installation service will provide TFTP service, as described in the previous section.

Appendix B: Preboot Execution Environment

To enable the interoperability of clients and downloaded bootstrap programs, the client preboot 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. The services provided by the client for use by the bootstrap are as follows:

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

The aspects of the client's state to be ensured by the client preboot code at the point in time that execution of the downloaded bootstrap is initiated are as follows:

- The use of certain portions of the client's main memory
- The settings of certain portions of the client's interrupt vector
- The settings of certain of the client's CPU registers

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.

Client State at Bootstrap Execution Time

This section describes the client state, including information about the bootstrap calling convention, memory usage, and interrupt vector table.

Bootstrap Calling Convention

The entire bootstrap image is downloaded into memory starting at location 07C00h. The preboot code transfers control to the bootstrap by executing a far call to the beginning of the bootstrap code. At this point the following must be true:

- CS:IP is to contain the value 0:7C00h

- ES:BX is to contain the address of the PXENV Entry Point structure described in the “Preboot API Entry Point and Installation Check” section later in this attachment
- EDX is to contain the physical address of the PXENV Entry Point structure
- SS:SP is to contain the address of the beginning of the unused portion of the preboot services stack

Note that the bootstrap code can determine how much free stack space is available by examining the contents of SP and by having knowledge of the memory usage conventions described in the following section.

Caution: A bootstrap should not exceed 32 KB in length. The memory between 07C00h and 10000h is free for use by the bootstrap.

Memory Usage

The following table describes the usage of the first megabyte of the client’s main memory when execution of the downloaded bootstrap is initiated.

Memory Usage During Execution of Downloaded Bootstrap

Address	Status	Preboot services usage	Conventional usage
0 3FF	RESERVED, except for Vector 1Ah at 0:68h.	Vector 1Ah is used to export the preboot services API.	Interrupt Vector Table
400 4FF	RESERVED, except for the 16-bit word at 40:13h	The 16-bit word at 40:13h is the size of free base memory in KB (SFBM/400h).	BIOS Data Area
500 6FF	RESERVED		DOS Data Area
700 7BFF	RESERVED		IO.SYS Load Area
7C00 10000 10000 10000+SFBM-1 10000+SFBM (SS:SP)		Downloaded Bootstrap Free base memory Preboot Services CPU Stack (unused)	
(SS:SP)+1 9FFFF	RESERVED	Preboot Services CPU Stack (used by Preboot Services) Preboot Services Code and Data Extended BIOS Data Area (possibly)	
A0000 BFFFF	RESERVED		Video Memory
C0000 C7FFF	RESERVED		Video BIOS
C8000 DFFFF	RESERVED		Other BIOS / Upper Memory
E0000 EFFFF	RESERVED	Contains a unique system ID structure.	Other BIOS / Upper Memory / System BIOS
F0000 FFFFF	RESERVED	Contains a unique system ID structure.	System BIOS

Free Memory 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.

Preboot Services 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 preboot services CPU stack. The downloaded image should not modify the used portion of the preboot services CPU stack prior to the time in the boot sequence when it is certain that the preboot services will not be needed again.

Preboot Services Code and Data. This memory area is reserved for the code and data that implement the preboot services. These locations should not be modified by the downloaded image prior to the time in the boot sequence, when it is certain that the preboot services will not be needed again.

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

PXENV Unique System ID (SYSID Bios Area). When execution of the download bootstrap begins, the client's main memory must contain a PXENV unique system ID structure. This structure must meet the following conditions:

- Entry Point Structure - This will be found in the 000E0000h to 000FFFFFh physical address area of Memory/RAM. *The Entry Point Structure is PARAGRAPH Aligned.*

Entry Point Structure

Element	Length	Description
Header/Type	7 Bytes	_SYSID_
Checksum	1 Byte	Checksum of 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 beginning of 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).

UUID Structure Format

Element	Length	Description
Header/Type	6 Bytes	_UUID_
Checksum	1 Byte	Checksum of 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).

1. Header/Type - This is a fixed size for all SYSID BIOS Structure Types. It will always be 6 bytes long. The first and last byte will always be the underscore ascii characters. The middle four bytes are the ASCII characters of UUID.

2. Checksum - This value is a two's complement based checksum which will cause the addition of all bytes defined for this table interface to be equal to 00h. Please note that this is a 8-bit addition calculation (byte wide addition).

3. Length - This value is a Total length of the entire UUID BIOS Structure type. In other words, this value is the addition of all the bytes in this structure from the first byte of the Header/Type field to the last byte in the Variable Data Portion field. The value for this field (for 16 bytes in the Variable Data Portion) is 019h.

4. Variable Data Portion - This value is the 16 byte long (10h) UUID.

Interrupt Vector Table

When execution of the downloaded bootstrap begins, interrupt 1Ah is chained to export the preboot services, TFTP, UDP, and UNDI APIs.

Preboot API Entry Point and Installation Check

Procedures for finding the preboot API entry point structure are architecture dependent. The methods described in this section work for PC/AT x86 clients. In general, the API entry point can be discovered using either of two methods. The first method is to use the installation check interrupt, Int 1Ah. The second is to scan base memory for the preboot API entry point structure. In addition, as described earlier in the "Bootstrap Calling Convention" section, certain registers contain the address of the entry point structure when the downloaded bootstrap is executed.

The preboot API supports only 16-bit real-mode or virtual-86 mode calls. Application programs must make far calls (CALL xxxxh:yyyyh) to the functions in the preboot APIs.

Int 1Ah Function 5650h
(Preboot API Installation Check)

Enter:

AX := 5650h (VP)

Exit:

AX := 564Eh (VN)

ES := 16-bit real-mode segment of the preboot API entry point structure.

BX := 16-bit real-mode offset of the preboot API entry point structure.

EDX := 32-bit physical address of the preboot API entry point structure.

All other register contents are preserved.

CF is cleared.

IF is preserved.

All other flags are undefined.

Preboot API Entry Point Structure

The preboot API entry point structure will be paragraph aligned and placed between the top of free base memory and A0000h (640k). The top of free base memory can be calculated using the size of free base memory word. This word is located in the BIOS data segment at 40:13h.

```
typedef struct s_PXENV_ENTRY {
UINT8 signature[6];  /* "PXENV+" */
UINT16 version;      /* MSB=major, LSB=minor */
UINT8 length;        /* sizeof(struct s_PXENV_ENTRY) */
UINT8 checksum;      /* 8-bit checksum off structure, */
                     /* including this bytes should be 0. */
UINT16 rm_entry_off;/* 16-bit real-mode offset and segment */
UINT16 rm_entry_seg;/* of the PXENV API entry point. */
UINT16 pm_entry_off;/* 16-bit protected-mode offset and */
UINT32 pm_entry_base;   /* segment base address of the */
                     /* PXENV API entry point. */
/* The PROM stack, base code and data segment selectors are only */
     /* required until the base code (TFTP API) layer is removed from */
     /* memory (this can only be done in real mode). */
        UINT16 stack_sel;         /* PROM stack segment.  Will be set */
        UINT16 stack_size;        /* to 0 when removed from memory. */
        UINT16 base_cs_sel;       /* Base code segment.  Will be set */
        UINT16 base_cs_size;      /* to 0 when removed from memory. */
        UINT16 base_ds_sel;       /* Base data segment.  Will be set */
        UINT16 base_ds_size;      /* to 0 when removed from memory. */
     /* The MLID code and data segment selectors are always required */
     /* when running the boot PROM in protected mode. */
        UINT16 mlid_ds_sel;       /* MLID data segment. */
        UINT16 mlid_ds_size;
        UINT16 mlid_cs_sel;       /* MLID code segment. */
        UINT16 mlid_cs_size;
} t_PXENV_ENTRY;
```

Register Usage for Preboot APIs

All API services use the following register settings:

Enter:

BX := PXENV function number

ES := Segment or selector of parameter structure

DI := Offset of parameter structure

Exit:

AX := EXIT_SUCCESS or EXIT_FAILURE

All other register contents are preserved.

CF := Cleared on success, set on error

IF is preserved.

All other flags are undefined.

Preboot Services API

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

UNLOAD PREBOOT STACK

Op-Code:	PXENV_UNLOAD_STACK
Input:	ES:DI points to a t_PXENV_UNLOAD_STACK parameter structure.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will be set to one of the values represented by the PXENV_STATUS_xxx constants. If successful, the address of the preboot entry point structure will also be filled in.
Description:	The preboot services implementation, except for the Universal NIC Driver, will be removed from base memory. UNDI API calls will still be available.
Note:	Service cannot be used in protected mode.
Warning!	The contents of the preboot entry point structure will be changed by this service. The old preboot entry point structure and contents are invalid and should no longer be used. The CPU stack used by the preboot services will be discarded. The caller must switch to a local CPU stack before making this call. This service should not be used after transferring control to a downloaded OS image.

GET BINL INFO

Op-Code:	PXENV_GET_BINL_INFO
Input:	ES:DI points to a t_PXENV_GET_BINL_INFO parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will be set to one of the values represented by the PXENV_STATUS_xxx constants. The buffer specified in the parameter structure will be filled with the requested information.
Description:	This service will return one of three buffers: The client's DHCPDISCOVER packet The DHCP server's DHCPACK packet The DHCPOFFER packet, which contains Option 60 set to "PXEClient" and a valid bootfile name. In the downloaded image, 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.

RESTART DHCP

Op-Code:	PXENV_RESTART_DHCP
Input:	ES:DI points to a t_PXENV_RESTART_DHCP parameter.
Output:	If DHCP cannot be restarted, PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will be set to one of the values represented by the PXENV_STATUS_xxx constants. If DHCP is restarted, control is never returned to the caller.
Description:	This service will try to establish a new DHCP connection with the server and try to start a download of a new image. The image to be downloaded will be determined by the server.
Note:	It is the responsibility of the caller to make sure the network connection is in a valid state before trying to restart DHCP. Any existing network connection should be closed, and the network adapter must be shutdown using the UNDI API service PXENV_UNDI_SHUTDOWN. Service cannot be used in protected mode.

RESTART TFTP

Op-Code:	PXENV_RESTART_TFTP
Input:	ES:DI points 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 will be returned and CF will be set. The status field in the parameter structure will 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 will try to establish a new TFTP connection with the server and to start a download of a new image. The image to be downloaded will be determined by the previously downloaded image.
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.

MODE SWITCH

Op-Code:	PXENV_MODE_SWITCH
Input:	ES:DI points to a t_PXENV_MODE_SWITCH parameter structure that has been initialized by the caller.
Output:	The status field in the parameter structure will be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	This service *must* be used when changing the processor between real mode and protected mode. The caller *must* initialize the stack, base code, base data, MLID code, and MLID data selectors and recompute the structure checksum before running this service.
Note:	This service can only be called from real mode (before entering, and after leaving, protected mode.) Interrupts need to be disabled when changing the PXENV entry point structure and when calling this service.
Warning!	This service can only be used with the default base code interrupt call backs.

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

Input: ES:DI points 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 will be returned in AX, with the CF set accordingly. The status field in the parameter structure will 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.

TFTP CLOSE

Op-Code: PXENV_TFTP_CLOSE

Input: ES:DI points to a t_PXENV_TFTP_CLOSE parameter structure that has been initialized by the caller.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will be set to one of the values represented by the PXENV_STATUS_xxx constants.

Description: Closes the previously opened TFTP connection.

TFTP READ

Op-Code: PXENV_TFTP_READ

Input: ES:DI points to a t_PXENV_TFTP_READ parameter structure that has been initialized by the caller.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will be set to one of the values represented by the PXENV_STATUS_xxx constants. When a read is successful, the PacketNumber and PacketLength fields will also be filled in.

Description: Reads one packet from the open TFTP connection.

TFTP/MTFTP READ FILE

Op-Code: PXENV_TFTP_READ_FILE

Input: ES:DI points to a t_PXENV_TFTP_READ_FILE parameter structure that has

been initialized by the caller.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will be set to one of the values represented by the PXENV_STATUS_xxx constants. When a read is successful, the PacketCount and PacketLength fields will 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.

This service cannot be call while in protected mode.

PROTECTED-MODE TFTP/MTFTP READ FILE

Op-Code: PXENV_TFTP_READ_FILE_PMODE

Input: ES:DI points to a t_PXENV_TFTP_READ_FILE_PMODE parameter structure that has been initialized by the caller.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will be set to one of the values represented by the PXENV_STATUS_xxx constants. When a read is successful, the PacketCount and PacketLength fields will 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.

This service cannot be called while in real mode.

TFTP_GET_FILE_SIZE

Op-Code: PXENV_TFTP_GET_FSIZE

Input: ES:DI points to a t_PXENV_TFTP_Get_FSIZE parameter structure that has been initialized by the caller.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will be set to one of the values represented by the PXENV_STATUS_xxx constants. When the call is successful, the FileSize field will 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: This service must not be called when there is an outstanding open tftp connection on the file

UDP API Service Descriptions

UDP OPEN

Op-Code:	PXENV_UDP_OPEN
Input:	ES:DI points to a t_PXENV_UDP_OPEN parameter.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will 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.

UDP CLOSE

Op-Code:	PXENV_UDP_CLOSE
Input:	ES:DI points to a t_PXENV_UDP_CLOSE parameter.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	Closes the previously opened UDP connection.

UDP WRITE

Op-Code:	PXENV_UDP_WRITE
Input:	ES:DI points to a t_PXENV_UDP_WRITE parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will 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.

UDP READ

Op-Code:	PXENV_UDP_READ
Input:	ES:DI points to a t_PXENV_UDP_READ parameter structure that has been initialized by the caller.
Output:	PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will be set to one of the values represented by the PXENV_STATUS_xxx constants.
Description:	Reads one packet from the opened UDP connection.

UNDI API Service Descriptions

UNDI STARTUP

Op-Code: PXENV_UNDI_STARTUP

Input: ES:DI points to a t_PXENV_UNDI_STARTUP parameter structure that has been initialized by the caller.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will be set to one of the values represented by the PXENV_STATUS_xxx constants.

Description: This call provides the Universal NIC Driver with necessary startup parameters, such as the data segment and network adapter identification variables. This call hooks Interrupt 1Ah to export the UNDI API. The rest of the API will not be available until this call has been completed. The data segment must be zero-filled before this API call is made.

Note: The entry point of the UNDI API must be at offset 0 of the UNDI code segment. The preboot code will install the UNDI API by making a far call to the API entry point, with ES:DI and BX setup for UNDI STARTUP.

This service cannot be used in protected mode.

UNDI CLEANUP

Op-Code: PXENV_UNDI_CLEANUP

Input: ES:DI points to a t_PXENV_UNDI_CLEANUP parameter structure.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will be set to one of the values represented by the PXENV_STATUS_xxx constants.

Description: This call will uninstall the Interrupt 1Ah hook and 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.

UNDI INITIALIZE

Op-Code: PXENV_UNDI_INITIALIZE

Input: ES:DI points to a t_PXENV_UNDI_INITIALIZE parameter structure that has been initialized by the caller.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will 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. The parameter block to this call contains the pointer to the call-back routines that will be called when a packet is received or when any other interrupt occurs.

Note: When a receive interrupt occurs, the network adapter driver queues the packet and calls the application's callback receive routine with a pointer to the packet received. Then, the callback routine can either copy the packet into its buffer or decide to delay the copy to a later time. The callback receive routine always gets the pointer to the first packet in the receive queue and not to the currently received packet that generated the interrupt.

If the call-back routine decides not to copy the data from the buffer at this time, the packet will remain in the receive queue and, as a result, the later packets might be dropped when the receive queue is full. At a later time, when the application wants to copy the packet, it can call the PXENV_UNDI_FORCE_INTERRUPT routine to simulate the receive interrupt.

When the preboot code makes this call to initialize the network adapter, it passes a NULL pointer for the ProtocolIni field in the parameter structure.

UNDI RESET ADAPTER

Op-Code: PXENV_UNDI_RESET_ADAPTER

Input: ES:DI points to a t_PXENV_UNDI_RESET_ADAPTER parameter structure that has been initialized by the caller.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will 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.

UNDI SHUTDOWN

Op-Code: PXENV_UNDI_SHUTDOWN

Input: ES:DI points to a t_PXENV_UNDI_SHUTDOWN parameter.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will 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.

UNDI OPEN

Op-Code: PXENV_UNDI_OPEN

Input: ES:DI points to a t_PXENV_UNDI_OPEN parameter structure that has been initialized by the caller.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will 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.

UNDI CLOSE

Op-Code: PXENV_UNDI_CLOSE

Input: ES:DI points to a t_PXENV_UNDI_CLOSE parameter.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will 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.

UNDI TRANSMIT PACKET

Op-Code: PXENV_UNDI_TRANSMIT

Input: ES:DI points to a t_PXENV_UNDI_TRANSMIT parameter structure that has been initialized by the caller.

Output: PXENV_EXIT_SUCCESS or the error code will be returned in AX, with the CF set accordingly. The error code will 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 transmission is always synchronous and blocks until the network adapter has placed the packet on the network.

UNDI SET MULTICAST ADDRESS

Op-Code: PXENV_UNDI_SET_MCAST_ADDRESS

Input: ES:DI points 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 will be returned in AX, with the CF set accordingly. The status field in the parameter structure will 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.

UNDI SET STATION ADDRESS

Op-Code: PXENV_UNDI_SET_STATION_ADDRESS

Input: ES:DI points 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 will be returned in AX, with the CF set accordingly. The status field in the parameter structure will 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.

UNDI SET PACKET FILTER

Op-Code: PXENV_UNDI_SET_PACKET_FILTER

Input: ES:DI points 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 will be returned in AX, with the CF set accordingly. The status field in the parameter structure will 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.

UNDI GET INFORMATION

Op-Code: PXENV_UNDI_GET_INFORMATION

Input: ES:DI points to a t_PXENV_UNDI_GET_INFORMATION parameter structure that has been initialized by the caller.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will 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.

UNDI GET STATISTICS

Op-Code: PXENV_UNDI_GET_STATISTICS

Input: ES:DI points to a t_PXENV_UNDI_GET_STATISTICS parameter structure that has been initialized by the caller.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with

the CF set accordingly. The status field in the parameter structure will 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.

UNDI CLEAR STATISTICS

Op-Code: PXENV_UNDI_CLEAR_STATISTICS

Input: ES:DI points to a t_PXENV_UNDI_CLEAR_STATISTICS parameter.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will 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.

UNDI INITIATE DIAGS

Op-Code: PXENV_UNDI_INITIATE_DIAGS

Input: ES:DI points to a t_PXENV_UNDI_INITIATE_DIAGS parameter.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will 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.

UNDI FORCE INTERRUPT

Op-Code: PXENV_UNDI_FORCE_INTERRUPT

Input: ES:DI points to a t_PXENV_UNDI_FORCE_INTERRUPT parameter structure that has been initialized by the caller.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will 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.

UNDI GET MULTICAST ADDRESS

Op-Code: PXENV_UNDI_GET_MCAST_ADDRESS

Input: ES:DI points to a t_PXENV_GET_MCAST_ADDRESS parameter structure that has

been initialized by the caller.

Output: PXENV_EXIT_SUCCESS or PXENV_EXIT_FAILURE will be returned in AX, with the CF set accordingly. The status field in the parameter structure will 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.

UNDI GET NIC TYPE

Op-Code: PXENV_UNDI_GET_NIC_TYPE

Input: ES:DI points 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 will be returned in AX, with the Carry Flag set accordingly. The status field in the parameter structure will be set to one of the values represented by the PXENV_STATUS_xxx constants. If the PXENV_EXIT_SUCCESS is returned the parameter structure will 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 DHCP discover packet using GET_BINL_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 DHCP discover packet itself.

Appendix C: Preboot API Common Type Definitions

Important: The code provided in this appendix is provided for informational purposes only.

```
/*
 *
* Copyright(c) 1997 by Intel Corporation.  All Rights Reserved.
 *
 */

#ifndef _PXENV_CMN_H
#define _PXENV_CMN_H

/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =*/
/* PXENV.H - PXENV/TFTP/UNDI API common, Version 2.x, 97-Jan-17
 *
 * Constant and type definitions used in other PXENV API header files.
 */


/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =*/
/* 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;


/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =*/
/* Result codes returned in AX by a PXENV API service.
 */
#define PXENV_EXIT_SUCCESS   0x0000
#define PXENV_EXIT_FAILURE   0x0001
#define PXENV_EXIT_CHAIN     0xFFFF  /* used internally */


/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =*/
/* CPU types
 */
#define PXENV_CPU_X86        0
#define PXENV_CPU_ALPHA      1
#define PXENV_CPU_PPC        2


/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =*/
/* Bus types
 */
#define PXENV_BUS_ISA        0
#define PXENV_BUS_EISA       1
#define PXENV_BUS_MCA        2
#define PXENV_BUS_PCI        3
#define PXENV_BUS_VESA       4
#define PXENV_BUS_PCMCIA     5
```

```
/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = */
/* Status codes returned in the status word of PXENV API parameter structures.
 */

/* Generic API errors 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_1A_HOOKED                              0x04
/* Int 1Ah cannot be unhooked. */

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

/* BIOS/system errors (0x20 to 0x2F) */
#define PXENV_STATUS_MCOPY_PROBLEM                          0x20

/* TFTP errors (0x30 to 0x3F) */
#define PXENV_STATUS_TFTP_CANNOT_ARP_ADDRESS                0x30
#define PXENV_STATUS_TFTP_OPEN_CANCELED_BY_KEYSTROKE        0x31
#define PXENV_STATUS_TFTP_OPEN_TIMEOUT                      0x32
#define PXENV_STATUS_TFTP_UNKNOWN_OPCODE                    0x33
#define PXENV_STATUS_TFTP_ERROR_OPCODE                      0x34
#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

/* BOOTP errors 0x40 to 0x4F) */
#define PXENV_STATUS_BOOTP_CANCELED_BY_KEYSTROKE            0x40
#define PXENV_STATUS_BOOTP_TIMEOUT                          0x41
#define PXENV_STATUS_BOOTP_NO_BOOTFILE_NAME                 0x43

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

/* Driver errors (0x60 to 0x6F) */
/* These errors are for UNDI compatible NIC drivers. */
#define PXENV_STATUS_UNDI_MEDIATEST_FAILED                  0x61
#define PXENV_STATUS_UNDI_CANNOT_INIT_NIC_FOR_MCAST         0x62

/* Bootstrap (.1) errors (0x70 to 0x7F) */
/* These errors are for the LSA/LCM bootstrap layer. */
```

```
/* Environment (.2) errors (0x80 to 0x8F) */
/* These errors are for LSA/LCM environment layers. */

/* MTFTP errors */
#define PXENV_STATUS_MTFTP_OPEN_CANCELED_BY_KEYSTROKE 0x91
#define PXENV_STATUS_MTFTP_OPEN_TIMEOUT                  0x92
#define PXENV_STATUS_MTFTP_UNKNOWN_OPCODE            0x93
#define PXENV_STATUS_MTFTP_READ_CANCELED_BY_KEYSTROKE 0x94
#define PXENV_STATUS_MTFTP_READ_TIMEOUT                  0x95
#define PXENV_STATUS_MTFTP_ERROR_OPCODE                  0x96
#define PXENV_STATUS_MTFTP_CANNOT_OPEN_CONNECTION    0x98
#define PXENV_STATUS_MTFTP_CANNOT_READ_FROM_CONNECTION   0x99
#define PXENV_STATUS_MTFTP_TOO_MANY_PACKAGES             0x9A

/* 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
/* Reserved errors (0xB0 to 0xCF) */

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

#endif /* _PXENV_CMN_H */

/* EOF - $Workfile:   pxe_cmn.h  $ */
```

Appendix D: Preboot API Parameter Structure and Type Definitions

Parameter Structure and Type Definitions

Important: The code provided in this appendix is provided for informational purposes only.

```
/*
 *
* Copyright(c) 1997 by Intel Corporation.  All Rights Reserved.
 *
 */

#ifndef _PXENV_API_H
#define _PXENV_API_H

/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =*/
/*      Parameter structure and type definitions for PXENV API version 2.x
 *
 *      PXENV.H needs to be #included before this file.
 *
 *      None of the PXENV API services are available after the stack
 *      has been unloaded.
 */

#include "bootp.h"          /* Defines BOOTPLAYER */


/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =*/
/* Format of PXENV entry point structure.
 */
typedef struct s_PXENV_ENTRY {
       UINT8 signature[6];      /* 'PXENV+' */
       UINT16 version;          /* MSB=major, LSB=minor */
       UINT8 length;            /* sizeof(struct s_PXENV_ENTRY) */
       UINT8 checksum;          /* 8-bit checksum off structure, */
                    /* including this bytes should */
                    /* be 0. */
       UINT16 rm_entry_off;     /* 16-bit real-mode offset and */
       UINT16 rm_entry_seg;     /* segment of the PXENV API entry */
                    /* point. */
       UINT16 pm_entry_off;     /* 16-bit protected-mode offset */
       UINT32 pm_entry_seg;     /* and segment base address of */
                           /* the PXENV API entry point. */
       UINT16 stack_sel;        /* PROM stack segment.  Will be set */
       UINT16 stack_size;       /* to 0 when removed from memory. */

       UINT16 base_cs_sel;      /* Base code segment.  Will be set */
       UINT16 base_cs_size;     /* to 0 when removed from memory. */

       UINT16 base_ds_sel;      /* Base data segment.  Will be set */
       UINT16 base_ds_size;     /* to 0 when removed from memory. */
```

```
        /* The MLID code and data segment selectors are always required */
        /* when running the boot PROM in protected mode. */

        UINT16 mlid_ds_sel;      /* MLID data segment. */
        UINT16 mlid_ds_size;

        UINT16 mlid_cs_sel;      /* MLID code segment. */
        UINT16 mlid_cs_size;

} t_PXENV_ENTRY;

#define PXENV_ENTRY_SIG          "PXENV+"


/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = */
/* One of the following command op-codes needs to be loaded into the
 * op-code register (BX) before making a call a PXENV API service.
 */
#define PXENV_UNLOAD_STACK       0x70
#define PXENV_GET_BINL_INFO      0x71
#define PXENV_RESTART_DHCP       0x72
#define PXENV_RESTART_TFTP       0x73
#define PXENV_MODE_SWITCH        0x74


/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =*/
/* PXENV API parameter structure typedefs.
 */

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
typedef struct s_PXENV_UNLOAD_STACK {
        UINT16 status;           /* Out: See PXENV_STATUS_xxx */
                        /*      constants. */
        UINT16 rm_entry_off;     /* Out: 16-bit real-mode segment and */
        UINT16 rm_entry_seg;     /*      offset of PXENV Entry Point */
                        /*      structure. */
        UINT16 pm_entry_off;     /* Out: 16-bit protected-mode offset */
        UINT32 pm_entry_base;        /*      and segment base address of */
                        /*      PXENV Entry Point structure. */
} t_PXENV_UNLOAD_STACK;


/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/* Packet types that can be requested in the s_PXENV_GET_BINL_INFO structure. */
#define PXENV_PACKET_TYPE_DHCP_DISCOVER  1
#define PXENV_PACKET_TYPE_DHCP_ACK       2
#define PXENV_PACKET_TYPE_BINL_REPLY     3

/* Three packets are preserved and available through this interface: 1) The
 * DHCP Discover packet sent by the client, 2) the DHCP acknowledgement
 * packet returned by the DHCP server, and 3) the reply packet from the BINL
 * server.  If the DHCP server provided the image bootfile name, the
 * DHCP_ACK and BINL_REPLY packets will identical.
 */

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
typedef struct s_PXENV_GET_BINL_INFO {
```

```
        UINT16 status;           /* Out: See PXENV_STATUS_xxx */
                    /*      constants. */
        UINT16 packet_type;      /* In: See PXENV_PACKET_TYPE_xxx */
                    /*     constants */
        UINT16 buffer_size;      /* In: Size of the buffer in */
                    /*     bytes.  Specifies the maximum */
                    /*     amount of data that will be */
                    /*     copied by the service.  A size */
                    /*     of zero is valid. */
                    /* Out: Amount of BINL data, in */
                    /*      bytes, that was copied into */
                    /*      the buffer.  For an input */
                    /*      size of zero, no data will be */
                    /*      copied and buffer_size will */
                    /*      be set to the maximum amount */
                    /*      of data available to be */
                    /*      copied. */
        UINT16 buffer_offset;        /* In: 16-bit offset and segment */
        UINT16 buffer_segment;       /*     selector of a buffer where the */
                    /*     requested packet will be */
                    /*     copied. */
                    /*Out: If buffer_size, buffer_offset and */
                    /*     buffer_segment are all zero;  */
                    /*     buffer_offset and buffer_segment */
                    /*     will be changed to point at the */
                    /*     packet buffers in the base code. */
} t_PXENV_GET_BINL_INFO;


/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
typedef struct s_PXENV_RESTART_DHCP {
        UINT16 status;           /* Out: See PXENV_STATUS_xxx */
                    /*      constants. */
} t_PXENV_RESTART_DHCP;

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
#define s_PXENV_RESTART_TFTP s_PXENV_TFTP_READ_FILE
#define t_PXENV_RESTART_TFTP t_PXENV_TFTP_READ_FILE

typedef struct s_PXENV_MODE_SWITCH {
        UINT16 status;   /* Out: See PXENV_STATUS_xxx constants*/
        UINT16 pxenv_entry_off;   /* In: Offset of PXENV entry point */
                            /*     structure. */
    UINT16 pxenv_entry_seg;   /* In: Real-mode segment or protected- */
                            /*     mode selector of the PXENV */
                            /*     entry point structure. */

        /* Protected-mode status call-back API is documented below. */

    UINT16 pmode_status_off;      /* In: Offset of 16-bit protected */
                                  /*     mode status call-back. */
    UINT16 pmode_status_sel;      /* In: Selector of 16-bit protected */
                                  /*     mode status call-back. */
} t_PXENV_MODE_SWITCH;

/*
 * The protected-mode call back will be used by the base code when the
```

```
 * client PC is in protected-mode and pmode_status_sel is non-zero.
 *
 * The base code will call the status call-back
 * with the following registers:
 *      AX = 0 (Inside a time-out loop.)
 *      AX = 1 - n (Packet number of received TFTP packet.)
 *      All other registers and flags are undefined.
 *
 * The call-back will return a continue/cancel flag
 * in the following registers:
 *      AX = 0 (continue)
 *      AX = 1 (cancel)
 * All other AX values are undefined, and will be treated as cancel.
 * All other registers and flags must be unchanged.
 */} t_PXENV_MODE_SWITCH;

#endif /* _PXENV_API_H */

/* EOF - $Workfile:   pxe_api.h  $ */
```

Appendix E: TFTP API Parameter Structure and Type Definitions

Important: The code provided in this appendix is provided for informational purposes only.

```
/*
* Copyright(c) 1997 by Intel Corporation.  All Rights Reserved.
 *
 */

/* TFTP_API.H
 *      Parameter structure and type definitions for TFTP API version 2.x
 *
 *      PXENV.H needs to be #included before this file.
 *
 *      None of the TFTP API services are available after the stack
 *      has been unloaded.
 */

#ifndef _TFTP_API_H
#define _TFTP_API_H


#include "pxe_cmn.h"


/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =*/
/* #defines and constants
 */

/* One of the following command op-codes needs to be loaded into the
 * op-code register (BX) before making a call a TFTP API service.
 */
#define PXENV_TFTP_OPEN             0x20
#define PXENV_TFTP_CLOSE            0x21
#define PXENV_TFTP_READ             0x22
#define PXENV_TFTP_READ_FILE        0x23
#define PXENV_TFTP_READ_FILE_PMODE  0x24
#define PXENV_TFTP_GET_FSIZE    0x25


/* Definitions of TFTP API parameter structures.
 */

typedef struct s_PXENV_TFTP_OPEN {
       UINT16 Status;           /* Out: See PXENV_STATUS_xxx */
                              /*      constants. */
       UINT8 ServerIPAddress[4];    /* In: 32-bit server IP */
                                /*     address. Big-endian. */
       UINT8 GatewayIPAddress[4];   /* In: 32-bit gateway IP */
                                /*     address. Big-endian. */
       UINT8 FileName[128];
```

```
        UINT16 TFTPPort;         /* In: Socket endpoint at */
                                 /*     which the TFTP server is */
                                 /*     listening to requests. */
                                 /*     Big-endian. */
} t_PXENV_TFTP_OPEN;


typedef struct s_PXENV_TFTP_GET_FSIZE {
        UINT16 Status;               /* Out: See PXENV_STATUS_xxx */
                                 /*      constants. */
        UINT8 ServerIPAddress[4];    /* In: 32-bit server IP */
                                 /*     address. Big-endian. */
        UINT8 GatewayIPAddress[4];   /* In: 32-bit gateway IP */
                                 /*     address. Big-endian. */
        UINT8 FileName[128];
        UINT32 FileSize;             /* Out: File Size */
} t_PXENV_TFTP_GET_FSIZE;


typedef struct s_PXENV_TFTP_CLOSE {
        UINT16 Status;           /* Out: See PXENV_STATUS_xxx */
                     /*      constants. */
} t_PXENV_TFTP_CLOSE;


typedef struct s_PXENV_TFTP_READ {
        UINT16 Status;           /* Out: See PXENV_STATUS_xxx */
                     /*      constants. */
        UINT16 PacketNumber;     /* Out: 16-bit packet number. */
        UINT16 BufferSize;       /* In: Size of the receive */
                     /*     buffer in bytes. */
                     /* Out: Size of the packet */
                     /*      written into the buffer. */
        UINT16 BufferOffset;     /* In: Segment/Selector and */
        UINT16 BufferSegment;        /*     offset of the receive buffer. */
                     /* Out: Unchanged */
} t_PXENV_TFTP_READ;


typedef struct s_PXENV_TFTP_READ_FILE {
        UINT16 Status;           /* Out: See PXENV_STATUS_xxx */
                     /*      constants. */
        UINT8 FileName[128];     /* In: file to be read */
        UINT32 BufferSize;       /* In: Size of the receive */
                     /*     buffer in bytes. */
                     /* Out: Size of the file */
                     /*      written into the buffer. */
        UINT32 BufferOffset;     /* In: 32-bit physical address of the */
                     /*     buffer to load the file into. */
        UINT8 ServerIPAddress[4];    /* In: 32-bit server IP */
                     /*     address. Big-endian. */
        UINT8 GatewayIPAddress[4];   /* In: 32-bit gateway IP */
                     /*     address. Big-endian. */
        UINT8 McastIPAddress[4];     /* In: 32-bit multicast IP address */
                     /*     on which file can be received */
                     /*     can be null for unicast */
        UINT16 TFTPClntPort;         /* In: Socket endpoint on the Client */
```

```
                    /*     at which the file can be */
                    /*     received in case of Multicast */
        UINT16 TFTPSrvPort;      /* In: Socket endpoint at which */
                    /*     server listens for requests. */
        UINT16 TFTPOpenTimeOut;      /* In: Timeout value in seconds to be */
                    /*     used for receiving data or ACK */
                    /*     packets.  If zero, default */
                    /*     TFTP-timeout is used. */
        UINT16 TFTPReopenDelay;      /* In: wait time in seconds to delay */
                    /*     a reopen request in case of */
                    /*     multicast. */
} t_PXENV_TFTP_READ_FILE;

typedef struct s_PXENV_TFTP_READ_FILE_PMODE {
    UINT16 Status;            /* Out: See PXENV_STATUS_xxx */
        /*      constants. */
    UINT8 FileName[128];      /* In: file to be read */
    UINT32 BufferSize;        /* In: Size of the receive */
        /*     buffer in bytes. */
        /* Out: Size of the file */
        /*      written into the buffer. */
    UINT32 BufferOffset;      /* In: 32-bit physical address of the */
        /*     buffer to load the file into. */
    UINT16 BufferSelector;    /* In: This field must be set to 0 in */
                              /*     real-mode, and to a valid data */
                              /*     selector in protected-mode. */
    UINT8 ServerIPAddress[4]; /* In: 32-bit server IP */
        /*     address. Big-endian. */
    UINT8 GatewayIPAddress[4]; /* In: 32-bit gateway IP */
        /*     address. Big-endian. */
    UINT8 McastIPAddress[4];  /* In: 32-bit multicast IP address */
        /*     on which file can be received */
        /*     can be null for unicast */
    UINT16 TFTPClntPort;      /* In: Socket endpoint on the Client */
        /*     at which the file can be */
        /*     received in case of Multicast */
    UINT16 TFTPSrvPort;       /* In: Socket endpoint at which */
        /*     server listens for requests. */
    UINT16 TFTPOpenTimeOut;   /* In: Timeout value in seconds to be */
        /*     used for receiving data or ACK */
        /*     packets.  If zero, default */
        /*     TFTP-timeout is used. */
    UINT16 TFTPReopenDelay;   /* In: wait time in seconds to delay */
        /*     a reopen request in case of */
        /*     multicast. */
} t_PXENV_TFTP_READ_FILE_PMODE;

/* Note:
        If the McastIPAddress specifies a non-zero value, the TFTP_ReadFile call
        tries to listen for multicast packets on the TFTPClntPort before
        opening a TFTP/MTFTP connection to the server.
        If it receives any packets (and not all) or if does not receive any,
        it waits for specified time and tries to reopen a multicast connection
        to the server.
        If the server supports multicast, it notifies the acknowledging client
        with a unicast and starts sending (multicast) the file.
        If the multicast open request times out, the client tries to connect to
```

```
        the server at TFTP server port for a unicast transfer.
*/


#endif /* _TFTP_API_H */

/* EOF - $Workfile:   tftp_api.h  $ */
```

Appendix F:UDP API Constant and Type Definitions

Important: The code provided in this appendix is provided for informational purposes only.

```
/*
 *
* Copyright(c) 1997 by Intel Corporation.  All Rights Reserved.
 *
 */

#ifndef _UDP_API_H
#define _UDP_API_H


#include "pxe_cmn.h"


/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =*/
/* #defines and constants
 */

#define PXENV_UDP_OPEN   0x30
#define PXENV_UDP_CLOSE  0x31
#define PXENV_UDP_READ   0x32
#define PXENV_UDP_WRITE  0x33


/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =*/
/* Typedefs
 */

typedef struct s_PXENV_UDP_OPEN {
       UINT16 status;           /* Out: See PXENV_STATUS_xxx #defines. */
       UINT8 src_ip[4];     /* Out: 32-bit IP address of this station */

} t_PXENV_UDP_OPEN;


typedef struct s_PXENV_UDP_CLOSE {
       UINT16 status;           /* Out: See PXENV_STATUS_xxx #defines. */
} t_PXENV_UDP_CLOSE;


typedef struct s_PXENV_UDP_READ {
       UINT16 status;           /* Out: See PXENV_STATUS_xxx #defines. */
       UINT8 src_ip[4];     /* Out: See description below */
       UINT8 dest_ip[4];        /* In/Out: See description below */
       UINT16 s_port;           /* Out: See description below */
       UINT16 d_port;           /* In/Out: See description below */
       UINT16 buffer_size;      /* In: Size of receive buffer. */
                        /* Out: Length of packet written into */
                        /*     receive buffer. */
       UINT16 buffer_off;       /* In: Segment/Selector and offset */
```

```
        UINT16 buffer_seg;       /*     of receive buffer. */
} t_PXENV_UDP_READ;

/*
src_ip: (Output)
======
UDP_READ fills this value on return with the 32-bit IP address
of the sender.

dest_ip: (Input/Output)
=======
If this field is non-zero then UDP_READ will filter the incoming
packets and accept those that are sent to this IP address.

If this field is zero then UDP_READ will accept any incoming
packet and return it's destination IP address in this field.

s_port: (Output)
=======
UDP_READ fills this value on return with the UDP port number
of the sender.

d_port: (Input/Output)
=======
If this field is non-zero then UDP_READ will filter the incoming
packets and accept those that are sent to this UDP port.

If this field is zero then UDP_READ will accept any incoming
packet and return it's destination UDP port in this field.

*/

#define UDP_READ_ANY_IP      0x0000  /* Accept packets sent to any IP. */
#define UDP_READ_CHECK_IP    0x0001  /* Only accept packets sent to a */
                     /* specific IP address. */


typedef struct s_PXENV_UDP_WRITE {
        UINT16 status;           /* Out: See PXENV_STATUS_xxx #defines. */
        UINT8 ip[4];         /* In: 32-bit destination IP address. */
        UINT8 gw[4];         /* In: 32-bit Gateway IP address. */
        UINT16 src_port; /* In: Source UDP port, assigned 2069 if given 0 */
        UINT16 dst port;         /* In: Destination UDP port */
        UINT16 buffer_size;      /* In: Length of packet in buffer. */
        UINT16 buffer_off;   /* In: Segment/Selector and offset */
        UINT16 buffer_seg;   /*     of transmit buffer. */
} t_PXENV_UDP_WRITE;


#endif /* _UDP_API_H */

/* EOF - $Workfile:   udp_api.h  $ */
```

Appendix G: UNDI API Constant and Type Definitions

Important: The code provided in this appendix is provided for informational purposes only.

```
/*
 *
* Copyright(c) 1997 by Intel Corporation.  All Rights Reserved.
 *
 */

#ifndef _UNDI_API_H
#define _UNDI_API_H

/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = */
/* UNDI_API.H
 *      Parameter structure and type definitions for TFTP API version 2.x
 *
 *      PXENV.H needs to be #included before this file.
 *
 *      All of the UNDI API services are still available after the stack
 *      has been unloaded.
 */


/* One of the following command op-codes needs to be loaded into the
 * op-code register (BX) before making a call a TFTP API service.
 */

#include "pxe_cmn.h"

#define PXENV_UNDI_STARTUP      0x0001
#define PXENV_UNDI_CLEANUP      0x0002
#define PXENV_UNDI_INITIALIZE       0x0003
#define PXENV_UNDI_RESET_NIC    0x0004
#define PXENV_UNDI_SHUTDOWN     0x0005
#define PXENV_UNDI_OPEN         0x0006
#define PXENV_UNDI_CLOSE        0x0007
#define PXENV_UNDI_TRANSMIT     0x0008
#define PXENV_UNDI_SET_MCAST_ADDR       0x0009
#define PXENV_UNDI_SET_STATION_ADDR 0x000A
#define PXENV_UNDI_SET_PACKET_FILTER 0x000B
#define PXENV_UNDI_GET_INFORMATION  0x000C
#define PXENV_UNDI_GET_STATISTICS       0x000D
#define PXENV_UNDI_CLEAR_STATISTICS 0x000E
#define PXENV_UNDI_INITIATE_DIAGS       0x000F
#define PXENV_UNDI_FORCE_INTERRUPT  0x0010
#define PXENV_UNDI_GET_MCAST_ADDR       0x0011

#define ADDR_LEN        16
#define MAXNUM_MCADDR           8


/* Definitions of TFTP API parameter structures.
```

```
 */

typedef struct s_PXENV_UNDI_MCAST_ADDR {
       UINT16 MCastAddrCount;       /* In: Number of multi-cast */
                    /* addresses. */
       UINT8 MCastAddr[MAXNUM_MCADDR][ADDR_LEN]; /* In: */
                    /* list of multi-cast addresses. */
                    /* Each address can take up to */
                    /* ADDR_LEN bytes and a maximum */
                    /* of MAXNUM_MCADDR address can */
                    /* be provided*/
} t_PXENV_UNDI_MCAST_ADDR;


typedef struct s_PXENV_UNDI_STARTUP {
       UINT16 Status;           /* Out: See PXENV_STATUS_xxx constants. */
       UINT8 BusType;           /* In: NIC bus type. */
       UINT8 AddrType;          /* 0 means DataSeg contains segment */
                    /* addess for DS; 1 means DataSegAddr */
                    /* contains 32-bit physical addr for */
                    /* the data segment. */
       UINT16 DataSeg;          /* Segment address for DS */
       UINT32 DataSegAddr;      /* In: 32-bit physical address */
                    /* of Universal NIC Driver */
                    /* data segment. */
       UINT16 DataSegSize;      /* In: Size of data segment in bytes. */
       UINT16 CodeSegSize;  /* In: Size of Code segment in bytes. */
       struct {
        UINT16 BusDevFunc; /* In: Bus, device and function numbers */
                           /* of this NIC. -1 if not PCI NIC */
        UINT16 PCI_ds_off;      /* Far pointer to PCI data structure */
        UINT16 PCI_ds_seg;
       } pci;
       struct {
        UINT16 CardSelNum;  /* In: Card select number.  */
                           /* -1 for non-PnP BBS device */
        UINT16 PnP_eh_off;  /* Far pointer to PnP expansion header */
        UINT16 PnP_eh_seg;
       } pnp;
} t_PXENV_UNDI_STARTUP;


typedef struct s_PXENV_UNDI_CLEANUP {
       UINT16 Status;           /* Out: See PXENV_STATUS_xxx constants. */
} t_PXENV_UNDI_CLEANUP;


typedef struct s_PXENV_UNDI_INITIALIZE {
       UINT16 Status;           /* Out: See PXENV_STATUS_xxx constants. */
       UINT32 ProtocolIni;      /* In: See description below */
       UINT16 ReceiveOffset;        /* In: See description below */
       UINT16 ReceiveSegment;       /* In: See description below */
       UINT16 GeneralIntOffset; /* In: See description below */
       UINT16 GeneralIntSegment;    /* In: See description below */
} t_PXENV_UNDI_INITIALIZE;
```

```
/* ProtocolIni :

    This is an input parameter and is a 32-bit physical address of
    a memory  copy of the  driver module in  the protocol.ini file
    obtained from the  Protocol Manager  driver(refer to  NDIS 2.0
    specifications).   This parameter  is basically  supported for
    the universal NDIS driver to pass the information contained in
    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
    itself.)  This value can be NULL for any other application
    interfacing to the Universal NIC Driver.

    ReceiveOffset,  ReceiveSegment:

    This is a pointer to the receive call-back routine and must be
    a non NULL pointer.   This  routine  will  be  called  in  the
    context  of  the  receive  interrupt  after  switching  to  an
    interrupt stack.  The parameters for the routine are passed in
    the registers which  are -  pointer to  the receive  buffer in
    ES:DI and the length of data in CX.  AX contains the length of
    the media  header starting at ES:DI,  BL contains the protocol
    id (0-unknown,  1-IP,  2-ARP,  3-RARP  and  4-others)  and  BH
    contains receive flag (0-directed/promiscuous, 1-broadcast and
    2-multicast).  It is the call-back routine's responsibility to
    initialize it's own  data segment  before starting  to execute
    and to preserve the contents of all the registers except AX.

    The  call-back can either  process the packet  or postpone the
    processing  to a later time.   It must return  a SUCCESS if it
    either copied the  packet into  its own  buffer or  decided to
    reject the  packet after  examining the  packet contents.   In
    this  case the NIC driver removes  the packet from the receive
    queue and recycles the buffer.  If the call-back does not want
    to look at the packet at this time it can return DELAY and the
    NIC  driver keeps the packet in the queue and will always give
    the  first packet's pointer to the call-back in the subsequent
    interrupts.  This  delay  may  however  cause  the  subsequent
    packets  to be dropped if  the receive queue is  full.

    If the application  decides  to  process  the  packet  it  had
    delayed  it can force the NIC driver to start the call-back by
    calling ForceInterrupt routine.

       GeneralIntOffset, GeneralIntSegment:

    This is also a pointer to a call back routine and will also be
    called  in  the  context  of  an  interrupt.    However,  this
    interrupt is  not  a  receive  interrupt  and  may  be  for  a
    1)transmit  complete, 2)post processing for a previous receive
    interrupt after releasing the interrupt stack or 3)it may be a
    software  interrupt.   The AX  register contains  the function
    code 1,  2 or 3 accordingly.  If  this routine is called for a
    transmit  complete indication, CX register contains the length
    of the packet transmitted and BX register contains the type of
    transmission  0, 1 or 2 according to 0)if the transmit was for
    a directed packet  (i.e.    neither  a  broadcast  and  nor  a
```

```
    multicast),  1)if  it  was  a  broadcast  or  2)if  it  was  a
    multicast.


    Note: This call-back  pointer  must  not  be  NULL.    If  the
    application  does not want to process any of these interrupts,
    a pointer to the routine which just returns the status must be
    provided.
*/


typedef struct s_PXENV_UNDI_SHUTDOWN {
       UINT16 Status;          /* Out: See PXENV_STATUS_xxx constants. */
} t_PXENV_UNDI_SHUTDOWN;


typedef struct s_PXENV_UNDI_RESET {
       UINT16 Status;          /* Out: See PXENV_STATUS_xxx constants. */
       t_PXENV_UNDI_MCAST_ADDR R_Mcast_Buf; /* multicast address list */
                   /* see note below  */
} t_PXENV_UNDI_RESET;

  /* Note: The  NIC  driver  does  not  remember  the  multicast
    addresses provided in any  call.    So  the  application  must
    provide the multicast address  list with all  the calls that
    reset the receive unit of the adapter.
  */


typedef struct s_PXENV_UNDI_OPEN {
       UINT16 Status;          /* Out: See PXENV_STATUS_xxx constants. */
       UINT16 OpenFlag;     /* In: See description below */
       UINT16 PktFilter;       /* In: Filter for receiving */
                   /* packet. It takes the following */
                   /* values, multiple values can be */
                   /* ORed together. */
#define FLTR_DIRECTED   0x0001  /* directed/multicast */
#define FLTR_BRDCST 0x0002  /* broadcast packets */
#define FLTR_PRMSCS 0x0004  /* any packet on LAN */
#define FLTR_SRC_RTG 0x0008  /* source routing packet */

       t_PXENV_UNDI_MCAST_ADDR McastBuffer; /* In: */
                   /* See t_PXENV_UNDI_MCAST_ADDR. */
} t_PXENV_UNDI_OPEN;

/*  OpenFlag:
    This is  an input parameter and is  adapter specific.  This is
    supported  for Universal NDIS 2.0 driver to pass down the Open
    flags  provided  by   the  protocol   driver  (See   NDIS  2.0
    specifications).  This can be zero.
*/


typedef struct s_PXENV_UNDI_CLOSE {
       UINT16 Status;       /* Out: See PXENV_STATUS_xxx constants. */
} t_PXENV_UNDI_CLOSE;
```

```
#define MAX_DATA_BLKS        8

typedef struct s_PXENV_UNDI_TBD{
        UINT16 ImmedLength;      /* In: Data buffer length in */
                    /* bytes. */
        UINT16 XmitOffset;       /* 16-bit segment & offset of the */
        UINT16 XmitSegment;      /* immediate data buffer. */
        UINT16 DataBlkCount;     /* In: Number of data blocks. */
        struct DataBlk {
         UINT8 TDPtrType;/*  /* 0 => 32 bit Phys pointer in TDDataPtr
                             /*  not supported in this version of LSA */
                             /*  1 => seg:offser in TDDataPtr which can
                             /*  be a real mode or 16-bit protected mode
                             /*  pointer */
         UINT8 TDRsvdByte;   /*   Reserved, must be zero. */
         UINT16 TDDataLen;   /*   Data block length in bytes. */
         UINT32 TDDataPtr;   /*   Far pointer to data buffer. */
        } DataBlock[MAX_DATA_BLKS];
} t_PXENV_UNDI_TBD;

typedef struct s_PXENV_UNDI_TRANSMIT {
        UINT16 Status;           /* Out: See PXENV_STATUS_xxx constants. */
        UINT8    Protocol;       /* See description below */
#define P_UNKNOWN    0
#define P_IP    1
#define P_ARP       2
#define P_RARP      3

        UINT8    XmitFlag;       /* See description below */
#define XMT_DESTADDR 0x0000      /* destination address given */
#define XMT_BROADCAST    0x0001      /* use broadcast address */
        UINT16 DestAddrOffset;       /* 16-bit segment & offset of the */
        UINT16 DestAddrSegment;      /* destination media address */
                    /*  See description below */
        UINT16 TBDOffset;        /* 16-bit segment & offset of the */
        UINT16 TBDSegment;       /* transmit buffer descriptor of type */
                    /* XmitBufferDesc   */
        UINT32 Reserved[2];      /* for future use   */
} t_PXENV_UNDI_TRANSMIT;

/*
    Protocol:

    This is the protocol  of  the  upper  layer  that  is  calling
    NICTransmit call.   If the  upper layer  has filled  the media
    header this field must be 0.

    XmitFlag:

    If  this flag is  0, the NIC  driver expects a  pointer to the
    destination media  address in the field  DestMediaAddr.  If 1,
    the   NIC  driver   fills  the   broadcast  address   for  the
    destination.

    DestAddrOffset & DestAddrSegment:
```

```
    This  is a pointer to the  hardware address of the destination
    media.  It  can be  null if  the destination  is not  known in
    which case the XmitFlag contains 1 for broadcast.  Destination
    media address  must be  obtained by  the upper  level protocol
    (with  Address Resolution Protocol) and NIC driver does not do
    any address resolution.

*/

typedef struct s_PXENV_UNDI_SET_MCAST_ADDR {
       UINT16 Status;           /* Out: See PXENV_STATUS_xxx constants. */
       t_PXENV_UNDI_MCAST_ADDR McastBuffer; /* In: */
                    /* See t_PXENV_UNDI_MCAST_ADDR. */
} t_PXENV_UNDI_SET_MCAST_ADDR;


typedef struct s_PXENV_UNDI_SET_STATION_ADDR {
       UINT16 Status;           /* Out: See PXENV_STATUS_xxx constants. */
       UINT8  StationAddress[ADDR_LEN]; /* new address to be set */
} t_PXENV_UNDI_SET_STATION_ADDR;


typedef struct s_PXENV_UNDI_SET_PACKET_FILTER {
       UINT16 Status;           /* Out: See PXENV_STATUS_xxx constants. */
       UINT8 filter;            /* In: Receive filter value. */
                   /* see t_PXENV_UNDI_OPEN for values */
} t_PXENV_UNDI_SET_PACKET_FILTER;

typedef struct s_PXENV_UNDI_GET_INFORMATION {
       UINT16 Status;           /* Out: See PXENV_STATUS_xxx constants. */
       UINT16 BaseIo;           /* Out: Adapter's Base IO */
       UINT16 IntNumber;        /* Out: IRQ number */
       UINT16 MaxTranUnit;      /* Out: MTU */
       UINT16  HwType;          /* Out: type of protocol at hardware level */

#define ETHER_TYPE 1
#define EXP_ETHER_TYPE 2
#define IEEE_TYPE   6
#define ARCNET_TYPE    7

    /*  other numbers can  be obtained from  rfc1010 for "Assigned
    Numbers".  This number may not be validated by the application
    and hence adding new numbers to the list should be fine at any
    time.  */

       UINT16 HwAddrLen;        /* Out: actual length of hardware address */
       UINT8 CurrentNodeAddress[ADDR_LEN]; /* Out: Current hardware address*/
       UINT8 PermNodeAddress[ADDR_LEN]; /* Out: Permanent hardware address*/
       UINT16 ROMAddress;       /* Out: ROM address */
       UINT16 RxBufCt;          /* Out: receive Queue length */
       UINT16 TxBufCt;          /* Out: Transmit Queue length */
} t_PXENV_UNDI_GET_INFORMATION;

typedef struct s_PXENV_UNDI_GET_STATISTICS {
       UINT16 Status;           /* Out: See PXENV_STATUS_xxx constants. */
       UINT32 XmtGoodFrames;        /* Out: No. of successful transmissions*/
       UINT32 RcvGoodFrames;        /* Out: No. of good frames received */
```

```
        UINT32 RcvCRCErrors;         /* Out: No. of frames with CRC error */
        UINT32 RcvResourceErrors;    /* Out: no. of frames discarded - */
                        /* Out: receive Queue full */
} t_PXENV_UNDI_GET_STATISTICS;

typedef struct s_PXENV_UNDI_CLEAR_STATISTICS {
        UINT16 Status;           /* Out: See PXENV_STATUS_xxx constants. */
} t_PXENV_UNDI_CLEAR_STATISTICS;

typedef struct s_PXENV_UNDI_INITIATE_DIAGS {
        UINT16 Status;           /* Out: See PXENV_STATUS_xxx constants. */
} t_PXENV_UNDI_INITIATE_DIAGS;

typedef struct s_PXENV_UNDI_FORCE_INTERRUPT {
        UINT16 Status;           /* Out: See PXENV_STATUS_xxx constants. */
} t_PXENV_UNDI_FORCE_INTERRUPT;

typedef struct s_PXENV_UNDI_GET_MCAST_ADDR {
        UINT16 Status;           /* Out: See PXENV_STATUS_xxx constants. */
        UINT32 InetAddr;     /* In: IP Multicast Address */
        UINT8  MediaAddr[ADDR_LEN];  /* Out: corresponding hardware */
                        /*      multicast address */
} t_PXENV_UNDI_GET_MCAST_ADDR;

#define PXENV_UNDI_GET_NIC_TYPE  0x12

typedef s_PXENV_UNDI_GET_NIC_TYPE{
UINT16 Status;  /* OUT: See PXENV_STATUS_xxx constants */
UINT8 NicType;  /* OUT: 2=PCI, 3=PnP */
        struct{
         UINT16 Vendor_ID;    /* OUT:  */
         UINT16 Dev_ID;   /* OUT:  */
         UINT8 Base_Class;    /* OUT: */
         UINT8 Sub_Class;/* OUT: */
         UINT8 Prog_Intf;/* OUT: program interface */
         UINT8 Rev;  /* OUT: Revision number */
         UINT16 BusDevFunc;   /* OUT: Bus, Device */
                 /* & Function numbers */
        }pci;
         struct{
             UINT32 EISA_Dev_ID; /* Out:  */
             UINT8 Base_Class;   /* OUT: */
             UINT8 Sub_Class;/* OUT: */
             UINT8 Prog_Intf;/* OUT: program interface */
             UINT16 CardSelNum;  /* OUT: Card Selector Number */
             }pnp;
        }pci_pnp_info;
}t_PXENV_UNDI_GET_NIC_TYPE;

#endif /* _UNDI_API_H */

/* EOF - $Workfile:   undi_api.h  $ */
```

Appendix H: DMI Instrumentation Details

The following standard groups from the *System Standards Group Definition, Approved Version 1.0,* must be instrumented and deployed on DMI-instrumented systems compliant with these guidelines:

- DMTF|ComponentID|001
- DMTF|Disk Mapping Table|001
- DMTF|Disks|002
- DMTF|General Information|001
- DMTF|Keyboard|003
- DMTF|Mouse|003
- DMTF|Operating System|001
- DMTF|Partition|001
- DMTF|Physical Container Global Table|001
- DMTF|Processor|003
- DMTF|System BIOS|001
- DMTF|System Cache|002
- DMTF|System Slots|003
- DMTF|Video BIOS|001
- DMTF|Video|002

The following table contains standard groups related to system resource management from the *System Standards Group Definition, Approved Version 1.0*. All these groups are valid standard groups, but the groups designated as “Replacement” groups are designed to replace the two groups marked “Original.” The DMTF recommends that instrumentation migrate to the Replacement groups.

To be compliant with these guidelines, a DMI-instrumented system provides either all of the Original groups or all of the corresponding Replacement groups. It is highly recommended that the Replacement groups be selected for newly implemented instrumentation.

DMI Standard Group	Implementation Guidelines
DMTF|System Resource 2|001	Replacement, recommended for new instrumentation
DMTF|System Resource Device Info|001	Replacement, recommended for new instrumentation
DMTF|System Resource DMA Info|001	Replacement, recommended for new instrumentation
DMTF|System Resource I/O Info|001	Replacement, recommended for new instrumentation
DMTF|System Resource IRQ Info|001	Replacement, recommended for new instrumentation
DMTF|System Resource Memory Info|001	Replacement, recommended for new instrumentation
DMTF|System Resources|001	Original, recommended for legacy instrumentation only
DMTF|System Resources Description|001	Original, recommended for legacy instrumentation only

The following table contains standard groups related to physical memory management from the *System Standards Group Definition, Approved Version 1.0*. All of these groups are valid standard groups, but the groups designated as “Replacement” groups are designed to replace the group marked “Original.” The DMTF recommends that instrumentation migrate to the Replacement groups. To be compliant with these guidelines, a DMI-instrumented system should be instrumented with either the Original group or the corresponding Replacement groups. It is highly recommended that the Replacement groups be selected for newly implemented instrumentation.

DMI Standard Group	Implementation Guidelines
DMTF|Memory Device|001	Replacement, recommended for new instrumentation
DMTF|Memory Array Mapped Addresses|001	Replacement, recommended for new instrumentation
DMTF|Memory Device Mapped Addresses|001	Replacement, recommended for new instrumentation
DMTF|Physical Memory Array|001	Replacement, recommended for new instrumentation
DMTF|Physical Memory|002	Original, recommended for legacy instrumentation only

The following standard groups from the *LAN Adapter Standard Groups Definition, Release Version 1.0,* must be instrumented and deployed on DMI-instrumented systems compliant with these guidelines:

- DMTF|Network Adapter 802 Port|001
- DMTF|Network Adapter Driver|001

Appendix I: DHCP Options For Host System Characteristics

INTERNET DRAFT

Mike Henry
Eric Dittert
Intel Corp.
March 26, 1997

DHCP Options For Host System Characteristics
<draft-dittert-host-sys-char-01.txt>

Status of this Memo

This document is an Internet Draft. Internet Drafts are working documents of the Internet Engineering Task Force (IETF), its Areas, and its Working Groups. Note that other groups may also distribute working documents as Internet Drafts.

Internet Drafts are draft documents valid for a maximum of six months. Internet Drafts may be updated, replaced, or obsoleted by other documents at any time. It is not appropriate to use Internet Drafts as reference material or to cite them other than as a "working draft" or "work in progress."

Please check the "1id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on ftp.is.co.za (Africa), nic.nordu.net (Europe), munnari.oz.au (Pacific Rim), ds.internic.net (US East Coast), or ftp.isi.edu (US West Coast).

Notice
All product and company names mentioned herein might be trademarks
of their respective owners.

Abstract

The interoperability of configuration services based on the Dynamic Host Configuration Protocol (DHCP) [1] in an environment of heterogeneous clients depends on clients accurately identifying themselves and their relevant characteristics to configuration servers. The class identifier provided through DHCP option 60 [2] helps in this regard, but such identifiers essentially only enable clients and servers that are "good friends" to find each other. This draft proposes the definition of two options that convey particular, generally useful information about the client system. This enables all servers to recognize this information, and is a step toward a richer form of interoperability for configuration services.

Expires September 1997 [Page 1]
March 26, 1997

The proposed options are

* Client System Architecture
* Client Network Device Interface

This draft also proposes a new type of client identifier based on generated UUID/GUIDs to be used in conjunction with the DHCP client identifier option (61).

1.0 Introduction

The use of DHCP to provide clients with configuration information in general, and boot images in particular can be complicated by several circumstances. Among these are

1) clients in the same service domain with different system architectures or hardware configurations
2) clients in the same service domain for which different software configurations are desired
3) the desire to have clients and servers provided by different vendors successfully interact

(By "clients in the same service domain" we mean clients, requests from which can reach the same server.) A key element in enabling the successful use of DHCP in such circumstances is the provision of mechanisms by which clients can accurately identify themselves and their relevant characteristics to a server.

For identifying characteristics of the client that are relevant to the selection of a boot image, the currently available mechanisms are the DHCP class identifier option (code 60) and the DHCP vendor specific information option (code 43). By definition, the vendor specific information option does not address the problem of enabling interoperability of clients and servers provided by different vendors. Information conveyed by the class identifier option could enable interoperability, provided that a sufficiently specific and complete set of class identifiers were defined and agreed to.

We suggest using an alternate approach, in which new, specific options are used to convey the characteristics of the client that determine which boot image(s) could run on the client, and the class identifier is used as a (site-specific) designation of the desired software configuration for the client. Section 2 defines two new options that are useful for conveying the client's hardware configuration.

For identifying the client as a unique entity, the currently available mechanisms is the DHCP client identifier option (code 61) [2]. Section 3 of this draft defines for use in this option an identifier type based on generated GUIDs - identifiers that are

guaranteed to be, or are very, very likely to be unique across time and all clients.

2.0 Client Characteristics Options

The options defined in this section provide the server with explicit knowledge about the client system that is generally useful in selecting an executable that the client can use as a boot image.

2.1 Client System Architecture Option

DHCP clients SHOULD include this option in DHCPDISCOVER and DHCPREQUEST messages. Doing so provides the server with explicit knowledge of the client's system architecture.

DHCP servers that use this option SHOULD include the option in responses that contain a bootfile name. If included, the value of the option MUST denote a system architecture for which the bootfile named is valid. DHCP servers MUST NOT include this option in responses that do not contain a bootfile name.

The format for this option is as follows:

```
 Code  Len   System Arch Code
+-----+-----+-----+-----+
| TBD |  2  | s1  | s2  |
+-----+-----+-----+-----+
```

The currently defined types and their codes are

System Architecture	Code
Intel Architecture PC	1
NEC PC-9800	2

2.2 Client Network Device Interface Option

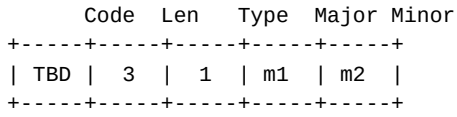
DHCP clients SHOULD include this option in DHCPDISCOVER and DHCPREQUEST messages. Doing so provides the server with explicit
knowledge of the client's network device.

DHCP servers that use this option SHOULD include the option in responses that contain a bootfile name. If included, the value of the option MUST denote a network device for which the bootfile named is valid. DHCP servers MUST NOT include this option in responses that do not contain a bootfile name.

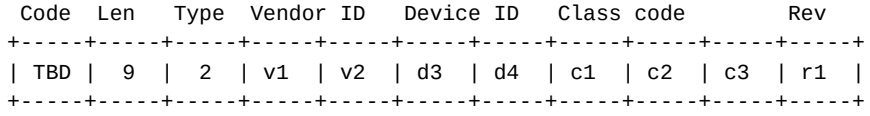
Three types of network device specifications are defined for use with this option:

* devices that support the Universal Network Driver Interface (UNDI), as described in the Network PC design guidelines [3]
* Plug-and-Play devices [4]
* PCI devices [5]

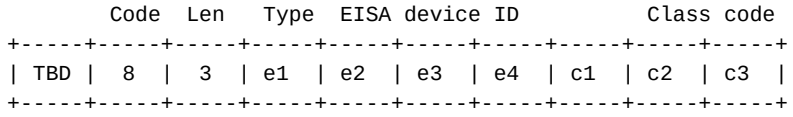
Each devices that supports (UNDI) SHOULD be specified as an UNDI device, regardless of whether it is also a Plug-and-Play device or a PCI device. To specify an UNDI device, the option contains a type code of 1 and the major and minor UNDI version numbers:



To specify a PCI network device, a type code of 2 is used, and the vendor ID, device ID, class code, and revision are included:

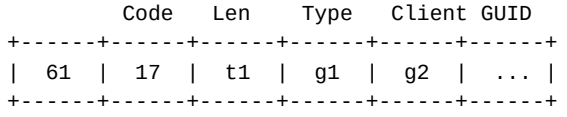


To specify a Plug-and-Play network device, a type code of 3 is used, and the EISA device ID and the class code are included:



3.0 UUID/GUID-based Client Identifiers

Whenever a client identifier option is included in a DHCP message, it MAY contain an identifier in UUID/GUID format. A client identifier option containing a type code of <TBD> MUST contain a 128-bit GUID as follows:



The format of the GUID MUST be as specified in the design guidelines for Network PCs [3].

4.0 References

[1] Droms, R. "Dynamic Host Configuration Protocol", RFC 1531

[2] Alexander,S. and Droms, R., "DHCP Options and BOOTP Vendor Extension" RFC 1533.

[3] Design Guidelines for a Network PC, reference to be provided

[4] Plug-and-Play specification, reference to be provided

[5] PCI specification, reference to be provided

5.0 Authors' Addresses

Mike Henry
Intel Corporation, MS JF3-408
5200 NE Elam Young Pkwy
Hillsboro, OR 97124

Phone: (503) 264-9689
Email: Mike_Henry@ccm.jf.intel.com

Eric Dittert
Intel Corporation, MS JF3-206
5200 NE Elam Young Pkwy
Hillsboro, OR 97124

Phone: (503) 264-8461
Email: Eric_Dittert@ccm.jf.intel.com

Appendix J: UUIDs and GUIDs

Network Working Group Paul J. Leach, Microsoft
INTERNET-DRAFT Rich Salz, Open Group
<draft-leach-uuids-guids-00.txt>
Category: Informational
Expires August 24, 1997 February 24, 1997

STATUS OF THIS MEMO

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress".

To learn the current status of any Internet-Draft, please check the "1id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on ftp.is.co.za (Africa), nic.nordu.net (Europe), munnari.oz.au (Pacific Rim), ds.internic.net (US East Coast), or ftp.isi.edu (US West Coast).

Distribution of this document is unlimited. Please send comments to the authors or the CIFS mailing list at cifs@listserv.msn.com. Discussions of the mailing list are archived at http://microsoft.ease.lsoft.com/archives/CIFS.html.

ABSTRACT

This specification defines the format of UUIDs (Universally Unique IDentifier), also known as GUIDs (Globally Unique IDentifier). A UUID is 128 bits long, and if generated according to the one of the mechanisms in this document, is either guaranteed to be different from all other UUIDs/GUIDs generated until 3400 A.D. or extremely likely to be different (depending on the mechanism chosen). UUIDs were originally used in the Network Computing System (NCS) [1] and later in the Open Software Foundation’s (OSF) Distributed Computing Environment [2].

This specification is derived from the latter specification with the kind permission of the OSF.

Introduction

This specification defines the format of UUIDs (Universally Unique IDentifiers), also known as GUIDs (Globally Unique IDentifiers). A UUID is 128 bits long, and if generated according to the one of the mechanisms in this document, is either guaranteed to be different from all other UUIDs/GUIDs generated until 3400 A.D. or extremely likely to be different (depending on the mechanism chosen).

Motivation

One of the main reasons for using UUIDs is that no centralized authority is required to administer them (beyond the one that allocates IEEE 802.1 node identifiers). As a result, generation on demand can be completely automated, and they can be used for a wide variety of purposes. The UUID generation algorithm described here supports very high allocation rates: 10 million per second per machine if you need it, so that they could even be used as transaction IDs.

UUIDs are fixed-size (128-bits) which is reasonably small relative to other alternatives. This fixed, relatively small size lends itself well to sorting, ordering, and hashing of all sorts, storing in databases, simple allocation, and ease of programming in general.

Specification

A UUID is an identifier that is unique across both space and time, with respect to the space of all UUIDs. To be precise, the UUID consists of a finite bit space. Thus the time value used for constructing a UUID is limited and will roll over in the future (approximately at A.D. 3400, based on the specified algorithm). A UUID can be used for multiple purposes, from tagging objects with an extremely short lifetime, to reliably identifying very persistent objects across a network.

The generation of UUIDs does not require that a registration authority be contacted for each identifier. Instead, it requires a unique value over space for each UUID generator. This spatially unique value is specified as an IEEE 802 address, which is usually already available to network-connected systems. This 48-bit address can be assigned based on an address block obtained through the IEEE registration authority. This section of the UUID specification assumes the availability of an IEEE 802 address to a system desiring to generate a UUID, but if one is not available section 4 specifies a way to generate a probabilistically unique one that can not conflict with any properly assigned IEEE 802 address.

Format

The following table gives the format of a UUID. The UUID consists of a record of 16 octets. The fields are in order of significance for comparison purposes, with "time_low" the most significant, and "node" the least significant.

Field	Data Type	Octet #	Note
time_low	unsigned 32 bit integer	0-3	The low field of the timestamp.
time_mid	unsigned 16 bit integer	4-5	The middle field of the timestamp.
time_hi_and_version	unsigned 16 bit integer	6-7	The high field of the timestamp multiplexed with the version number.
clock_seq_hi_and_reserved	unsigned 8 bit integer	8	The high field of the clock sequence multiplexed with the variant.
clock_seq_low	unsigned 8 bit integer	9	The low field of the clock sequence.
node	unsigned 48 bit integer	10-15	The spatially unique node identifier.

To minimize confusion about bit assignments within octets, the UUID record definition is defined only in terms of fields that are integral numbers of octets. The version number is in the most significant 4 bits of the time stamp (*time_hi*), and the variant field is in the most significant 3 bits of the clock sequence (*clock_seq_high*).

The timestamp is a 60 bit value. For UUID version 1, this is represented by Coordinated Universal Time (UTC) as a count of 100-nanosecond intervals since 00:00:00.00, 15 October 1582 (the date of Gregorian reform to the Christian calendar).

The following table lists currently defined versions of the UUID.

MSB0	MSB1	MSB2	Msb3	VERSION	DESCRIPTION
0	**0**	**0**	1	**1**	**THE VERSION SPECIFIED IN THIS DOCUMENT.**
0	**0**	**1**	0	**2**	**RESERVED FOR DCE SECURITY VERSION, WITH EMBEDDED POSIX UIDS.**

The variant field determines the layout of the UUID. The structure of UUIDs is fixed across different versions within a variant, but not across variants; hence, other UUID variants may not interoperate with the UUID variant specified in this document. Interoperability of UUIDs is defined as the applicability of operations such as string conversion, comparison, and lexical ordering across different systems. The *variant* field consists of a variable number of the msbs of the *clock_seq_hi_and_reserved* field.

The following table lists the contents of the variant field.

MSB0	MSB1	MSB2	DESCRIPTION
0	**-**	**-**	**RESERVED, NCS BACKWARD COMPATIBILITY.**
1	**0**	**-**	**THE VARIANT SPECIFIED IN THIS DOCUMENT.**
1	**1**	**0**	**RESERVED, MICROSOFT CORPORATION GUID.**
1	**1**	**1**	**RESERVED FOR FUTURE DEFINITION.**

The clock sequence is required to detect potential losses of monotonicity of the clock. Thus, this value marks discontinuities and prevents duplicates. An algorithm for generating this value is outlined in the “Clock Sequence” section below.

The clock sequence is encoded in the 6 least significant bits of the *clock_seq_hi_and_reserved* field and in the *clock_seq_low* field.

The *node* field consists of the IEEE address, usually the host address. For systems with multiple IEEE 802 nodes, any available node address can be used. The lowest addressed octet (octet number 10) contains the global/local bit and the unicast/multicast bit, and is the first octet of the address transmitted on an 802.3 LAN.

Depending on the network data representation, the multi-octet unsigned integer fields are subject to byte swapping when communicated between different endian machines.

The nil UUID is special form of UUID that is specified to have all 128 bits set to 0 (zero).

Algorithms for Creating a UUID

Various aspects of the algorithm for creating a UUID are discussed in the following sections. UUID generation requires a guarantee of uniqueness within the node ID for a given variant and version. Interoperability is provided by complying with the specified data structure. To prevent possible UUID collisions, which could be caused by different implementations on the same node, compliance with the algorithm specified here is required.

Clock Sequence

The clock sequence value must be changed whenever:

- the UUID generator detects that the local value of UTC has gone backward.

- the UUID generator has lost its state of the last value of UTC used, indicating that time *may* have gone backward; this is typically the case on reboot.

While a node is operational, the UUID service always saves the last UTC used to create a UUID. Each time a new UUID is created, the current *UTC* is compared to the saved value and if either the current value is less (the non-monotonic clock case) or the saved value was lost, then the *clock sequence* is incremented modulo 16,384, thus avoiding production of duplicate UUIDs.

The *clock sequence* must be initialized to a random number to minimize the correlation across systems. This provides maximum protection against *node* identifiers that may move or switch from system to system rapidly. The initial value MUST NOT be correlated to the node identifier.

The rule of initializing the *clock sequence* to a random value is waived if, and only if all of the following are true:

- The *clock sequence* value is stored in non-volatile storage.

- The system is manufactured such that the IEEE address ROM is designed to be inseparable from the system by either the user or field service, so that it cannot be moved to another system.

- The manufacturing process guarantees that only new IEEE address ROMs are used.

- Any field service, remanufacturing or rebuilding process that could change the value of the clock sequence must reinitialise it to a random value.

In other words, the system constraints prevent duplicates caused by possible migration of the IEEE address, while the operational system itself can protect against non-monotonic clocks, except in the case of field service intervention. At manufacturing time, such a system may initialise the clock sequence to any convenient value.

System Reboot

There are two possibilities when rebooting a system:

- the UUID generator state - the last UTC, adjustment, and clock sequence - of the UUID service has been restored from non-volatile store

- the state of the last UTC or adjustment has been lost.

If the state variables have been restored, the UUID generator just continues as normal. Alternatively, if the state variables cannot be restored, they are reinitialised, and the clock sequence is changed.

If the clock sequence is stored in non-volatile store, it is incremented; otherwise, it is reinitialised to a new random value.

Clock Adjustment

UUIDs may be created at a rate greater than the system clock resolution. Therefore, the system must also maintain an adjustment value to be added to the lower-order bits of the time. Logically, each time the system clock ticks, the adjustment value is cleared. Every time a UUID is generated, the current adjustment value is read and incremented atomically, then added to the UTC time field of the UUID.

Clock Overrun

The 100 nanosecond granularity of time should prove sufficient even for bursts of UUID creation in high-performance multiprocessors. If a system overruns the clock adjustment by requesting too many UUIDs within a single system clock tick, the UUID service may raise an exception, handled in a system or process-dependent manner either by:

- terminating the requester

- reissuing the request until it succeeds

- stalling the UUID generator until the system clock catches up.

If the processors overrun the UUID generation frequently, additional node identifiers and clocks may need to be added.

UUID Generation

UUIDs are generated according to the following algorithm:

- Determine the values for the UTC-based timestamp and clock sequence to be used in the UUID, as described above.

- For the purposes of this algorithm, consider the timestamp to be a 60-bit unsigned integer and the clock sequence to be a 14-bit unsigned integer. Sequentially number the bits in a field, starting from 0 (zero) for the least significant bit.

- Set the *time_low* field equal to the least significant 32-bits (bits numbered 0 to 31 inclusive) of the time stamp in the same order of significance.

- Set the *time_mid* field equal to the bits numbered 32 to 47 inclusive of the time stamp in the same order of significance.

- Set the 12 least significant bits (bits numbered 0 to 11 inclusive) of the *time_hi_and_version* field equal to the bits numbered 48 to 59 inclusive of the time stamp in the same order of significance.

- Set the 4 most significant bits (bits numbered 12 to 15 inclusive) of the *time_hi_and_version* field to the 4-bit version number corresponding to the UUID version being created, as shown in the table above.

- Set the *clock_seq_low* field to the 8 least significant bits (bits numbered 0 to 7 inclusive) of the *clock sequence* in the same order of significance.

- Set the 6 least significant bits (bits numbered 0 to 5 inclusive) of the *clock_seq_hi_and_reserved* field to the 6 most significant bits (bits numbered 8 to 13 inclusive) of the *clock sequence* in the same order of significance.

- Set the 2 most significant bits (bits numbered 6 and 7) of the *clock_seq_hi_and_reserved* to 0 and 1, respectively.

- Set the *node* field to the 48-bit IEEE address in the same order of significance as the address.

String Representation of UUIDs

For use in human readable text, a UUID string representation is specified as a sequence of fields, some of which are separated by single dashes.

Each field is treated as an integer and has its value printed as a zero-filled hexadecimal digit string with the most significant digit first. The hexadecimal values a to f inclusive are output as lower case characters, and are case insensitive on input. The sequence is the same as the UUID constructed type.

The formal definition of the UUID string representation is provided by the following extended BNF:

```
UUID                    = <time_low> "-" <time_mid> "-"
                          <time_high_and_version> "-"
                          <clock_seq_and_reserved>
                          <clock_seq_low> "-" <node>
time_low                = 4*<hexOctet>
time_mid                = 2*<hexOctet>
time_high_and_version   = 2*<hexOctet>
clock_seq_and_reserved = <hexOctet>
clock_seq_low           = <hexOctet>
node                    = 6*<hexOctet
hexOctet                = <hexDigit> <hexDigit>
hexDigit =
     "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
     | "a" | "b" | "c" | "d" | "e" | "f"
     | "A" | "B" | "C" | "D" | "E" | "F"
```

The following is an example of the string representation of a UUID:

```
f81d4fae-7dec-11d0-a765-00a0c91e6bf6
```

Comparing UUIDs

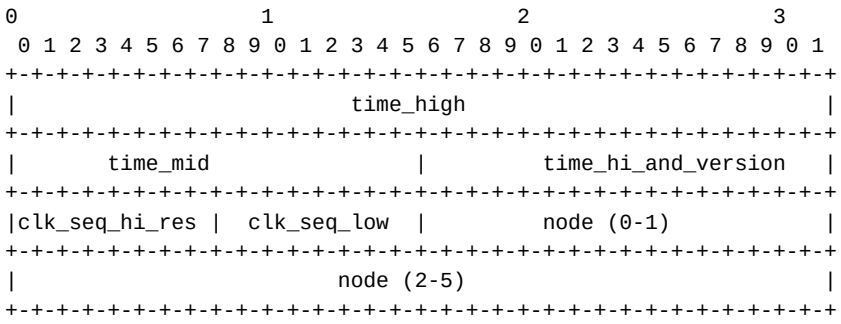
Consider each field of the UUID to be an unsigned integer as shown in the table in section 3.1. Then, to compare a pair of UUIDs, arithmetically compare the corresponding fields from each UUID in order of significance and according to their data type. Two UUIDs are equal if and only if all the corresponding fields are equal. The first of two UUIDs follows the second if the most significant field in which the UUIDs differ is greater for the first UUID. The first of a pair of UUIDs precedes the second if the most significant field in which the UUIDs differ is greater for the second UUID.

Byte order of UUIDs

UUIDs may be transmitted in many different forms, some of which may be dependent on the presentation or application protocol where the UUID may be used. In such cases, the order, sizes and byte orders of the UUIDs fields on the wire will depend on the relevant presentation or application protocol. However, it is strongly RECOMMENDED that the order of the fields conform with ordering set out in section 3.1 above. Furthermore, the payload size of each field in the

application or presentation protocol MUST be large enough that no information lost in the process of encoding them for transmission.

In the absence of explicit application or presentation protocol specification to the contrary, a UUID is encoded as a 128-bit object, as follows: the fields are encoded as 16 octets, with the sizes and order of the fields defined in section 3.1, and with each field encoded with the Most Significant Byte first (also known as network byte order).



Node IDs when no IEEE 802 network card is available

If a system wants to generate UUIDs but has no IEE 802 compliant network card or other source of IEEE 802 addresses, then this section describes how to generate one.

The ideal solution is to obtain a 47 bit cryptographic quality random number, and use it as the low 47 bits of the node ID, with the most significant bit of the first octet of the node ID set to 1. This bit is the unicast/multicast bit, which will never be set in IEEE 802 addresses obtained from network cards; hence, there can never be a conflict between UUIDs generated by machines with and without network cards.

If a system does not have a primitive to generate cryptographic quality random numbers, then in most systems there are usually a fairly large number of sources of randomness available from which one can be generated. Such sources are system specific, but often include:

- the percent of memory in use

- the size of main memory in bytes

- the amount of free main memory in bytes

- the size of the paging or swap file in bytes

- free bytes of paging or swap file

- the total size of user virtual address space in bytes

- the total available user address space bytes

- the size of boot disk drive in bytes

- the free disk space on boot drive in bytes

- the current time

- the amount of time since the system booted

- the individual sizes of files in various system directories

- the creation, last read, and modification times of files in various system directories

- the utilization factors of various system resources (heap, etc.)

- current mouse cursor position

- current caret position

- current number of running processes, threads

- handles or IDs of the desktop window and the active window

- the value of stack pointer of the caller

- the process and thread ID of caller

various processor architecture specific performance counters (instructions executed, cache misses, TLB misses)

(Note that it precisely the above kinds of sources of randomness that are used to seed cryptographic quality random number generators on systems without special hardware for their construction.)

In addition, items such as the computer's name and the name of the operating system, while not strictly speaking random, will help differentiate the results from those obtained by other systems.

The exact algorithm to generate a node ID using these data is system specific, because both the data available and the functions to obtain them are often very system specific. However, assuming that one can concatenate all the values from the randomness sources into a buffer, and that a cryptographic hash function such as MD5 [3] is available, the following code will compute a node ID:

```
#include <md5.h>
#define HASHLEN 16

void GenNodeID(
 unsigned char * pDataBuf,    // concatenated "randomness values"
 long cData,          // size of randomness values
 unsigned char NodeID[6]      // node ID
)
{
  int i, j, k;
  unsigned char Hash[HASHLEN];
  MD_CTX context;

  MDInit (&context);
  MDUpdate (&context, pDataBuf, cData);
  MDFinal (Hash, &context);

  for (j = 0; j<6; j++) NodeId[j]=0;
  for (i = 0,j = 0; i < HASHLEN; i++) {
     NodeID[j++] ^= Hash[i];
     if (j == 6) j = 0;
 };
 NodeID[0] |= 0x80;      // set the multicast bit
};
```

Other hash functions, such as SHA-1 [4], can also be used (in which case HASHLEN will be 20). The only requirement is that the result be suitably random – in the sense that the outputs from a set uniformly distributed inputs are themselves uniformly distributed, and that a single bit change in the input can be expected to cause half of the output bits to change.

Obtaining IEEE 802 addresses

The following URL

http://stdsbbs.ieee.org/products/oui/forms/index.html

contains information on how to obtain an IEEE 802 address block. Cost is $1000 US.

Security Considerations

It should not be assumed that UUIDs are hard to guess; they should not be used as capabilities.

Acknowledgements

This document draws heavily on the OSF DCE specification for UUIDs. Ted Ts'o provided helpful comments, especially on the byte ordering section which we mostly plagiarized from a proposed wording he supplied (all errors in that section are our responsibility, however).

References

[1] Lisa Zahn, et. al., Network Computing Architecture, Prentice Hall, Englewood Cliffs, NJ, 1990

[2] DCE: Remote Procedure Call, Open Group CAE Specification C309 ISBN 1-85912-041-5 28cm. 674p. pbk. 1,655g. 8/94

[3] R. Rivest, RFC 1321, "The MD5 Message-Digest Algorithm", 04/16/1992.

[4] SHA Spec - TBD

Authors' addresses

Paul J. Leach
Microsoft
1 Microsoft Way
Redmond, WA, 98052, U.S.A.
Email: paulle@microsoft.co

Rich Salz
The Open Group
11 Cambridge Center
Cambridge, MA 02142, U.S.A.
Email r.salz@opengroup.org

Appendix A – UUID Reference Implementation

```
/*
** Copyright (c) 1990- 1993, 1996 Open Software Foundation, Inc.
** Copyright (c) 1989 by Hewlett-Packard Company, Palo Alto, Ca. &
** Digital Equipment Corporation, Maynard, Mass.
** To anyone who acknowledges that this file is provided "AS IS"
** without any express or implied warranty: permission to use, copy,
** modify, and distribute this file for any purpose is hereby
** granted without fee, provided that the above copyright notices and
** this notice appears in all source code copies, and that none of
** the names of Open Software Foundation, Inc., Hewlett-Packard
** Company, or Digital Equipment Corporation be used in advertising
** or publicity pertaining to distribution of the software without
** specific, written prior permission.  Neither Open Software
** Foundation, Inc., Hewlett-Packard Company, nor Digital Equipment
** Corporation makes any representations about the suitability of
** this software for any purpose.
*/
#include <sys/types.h>
#include <sys/time.h>

typedef unsigned long   unsigned32;
typedef unsigned short  unsigned16;
typedef unsigned char   unsigned8;
typedef unsigned char   byte;

#define CLOCK_SEQ_LAST              0x3FFF
#define RAND_MASK                   CLOCK_SEQ_LAST

typedef struct _uuid_t {
    unsigned32          time_low;
    unsigned16          time_mid;
    unsigned16          time_hi_and_version;
    unsigned8           clock_seq_hi_and_reserved;
    unsigned8           clock_seq_low;
    byte                node[6];
} uuid_t;

typedef struct _unsigned64_t {
    unsigned32          lo;
    unsigned32          hi;
} unsigned64_t;

/*
**  Add two unsigned 64-bit long integers.
*/
#define ADD_64b_2_64b(A, B, sum) \
    { \
        if (!(((A)->lo & 0x80000000UL) ^ ((B)->lo & 0x80000000UL))) { \
            if (((A)->lo&0x80000000UL)) { \
                (sum)->lo = (A)->lo + (B)->lo; \
                (sum)->hi = (A)->hi + (B)->hi + 1; \
            } \
            else { \
                (sum)->lo  = (A)->lo + (B)->lo; \
                (sum)->hi = (A)->hi + (B)->hi; \
```

```
            } \
        } \
        else { \
            (sum)->lo = (A)->lo + (B)->lo; \
            (sum)->hi = (A)->hi + (B)->hi; \
            if (!((sum)->lo&0x80000000UL)) (sum)->hi++; \
        } \
    }

/*
**  Add a 16-bit unsigned integer to a 64-bit unsigned integer.
*/
#define ADD_16b_2_64b(A, B, sum) \
    { \
        (sum)->hi = (B)->hi; \
        if ((B)->lo & 0x80000000UL) { \
            (sum)->lo = (*A) + (B)->lo; \
            if (!((sum)->lo & 0x80000000UL)) (sum)->hi++; \
        } \
        else \
            (sum)->lo = (*A) + (B)->lo; \
    }

/*
**  Global variables.
*/
static unsigned64_t     time_last;
static unsigned16       clock_seq;

static void
mult32(unsigned32 u, unsigned32 v, unsigned64_t *result)
{
    /* Following the notation in Knuth, Vol. 2. */
    unsigned32 uuid1, uuid2, v1, v2, temp;

    uuid1 = u >> 16;
    uuid2 = u & 0xFFFF;
    v1 = v >> 16;
    v2 = v & 0xFFFF;
    temp = uuid2 * v2;
    result->lo = temp & 0xFFFF;
    temp = uuid1 * v2 + (temp >> 16);
    result->hi = temp >> 16;
    temp = uuid2 * v1 + (temp & 0xFFFF);
    result->lo += (temp & 0xFFFF) << 16;
    result->hi += uuid1 * v1 + (temp >> 16);
}

static void
get_system_time(unsigned64_t *uuid_time)
{
    struct timeval tp;
    unsigned64_t utc, usecs, os_basetime_diff;

    gettimeofday(&tp, (struct timezone *)0);
    mult32((long)tp.tv_sec, 10000000, &utc);
    mult32((long)tp.tv_usec, 10, &usecs);
```

```
    ADD_64b_2_64b(&usecs, &utc, &utc);

    /* Offset between UUID formatted times and Unix formatted times.
     * UUID UTC base time is October 15, 1582.
     * Unix base time is January 1, 1970. */
    os_basetime_diff.lo = 0x13814000;
    os_basetime_diff.hi = 0x01B21DD2;
    ADD_64b_2_64b(&utc, &os_basetime_diff, uuid_time);
}

/*
** See "The Multiple Prime Random Number Generator" by Alexander
** Hass pp. 368-381, ACM Transactions on Mathematical Software,
** 12/87.
*/
static unsigned32 rand_m;
static unsigned32 rand_ia;
static unsigned32 rand_ib;
static unsigned32 rand_irand;

static void
true_random_init(void)
{
    unsigned64_t t;
    unsigned16 seed;

 /* Generating our 'seed' value Start with the current time, but,
  * since the resolution of clocks is system hardware dependent and
  * most likely coarser than our resolution (10 usec) we 'mixup' the
  * bits by xor'ing all the bits together.  This will have the effect
  * of involving all of the bits in the determination of the seed
  * value while remaining system independent.  Then for good measure
  * to ensure a unique seed when there are multiple processes
  * creating UUIDs on a system, we add in the PID.
  */
    rand_m = 971;
    rand_ia = 11113;
    rand_ib = 104322;
    rand_irand = 4181;
    get_system_time(&t);
    seed  =  t.lo        & 0xFFFF;
    seed ^= (t.lo >> 16) & 0xFFFF;
    seed ^=  t.hi        & 0xFFFF;
    seed ^= (t.hi >> 16) & 0xFFFF;
    rand_irand += seed + getpid();
}

static unsigned16
true_random(void)
{
    if ((rand_m += 7) >= 9973)
        rand_m -= 9871;
    if ((rand_ia += 1907) >= 99991)
        rand_ia -= 89989;
    if ((rand_ib += 73939) >= 224729)
        rand_ib -= 96233;
    rand_irand = (rand_irand * rand_m) + rand_ia + rand_ib;
```

```
    return (rand_irand >> 16) ^ (rand_irand & RAND_MASK);
}

/*
**  Startup initialization routine for the UUID module.
*/
void
uuid_init(void)
{
    true_random_init();
    get_system_time(&time_last);
#ifdef NONVOLATILE_CLOCK
    clock_seq = read_clock();
#else
    clock_seq = true_random();
#endif
}

static int
time_cmp(unsigned64_t *time1, unsigned64_t *time2)
{
    if (time1->hi < time2->hi) return -1;
    if (time1->hi > time2->hi) return 1;
    if (time1->lo < time2->lo) return -1;
    if (time1->lo > time2->lo) return 1;
    return 0;
}

static void new_clock_seq(void)
{
    clock_seq = (clock_seq + 1) % (CLOCK_SEQ_LAST + 1);
    if (clock_seq == 0) clock_seq = 1;
#ifdef NONVOLATILE_CLOCK
    write_clock(clock_seq);
#endif
}

void uuid_create(uuid_t *uuid)
{
    static unsigned64_t     time_now;
    static unsigned16       time_adjust;
    byte                    eaddr[6];
    int                     got_no_time = 0;

    get_ieee_node_identifier(&eaddr);      /* TO BE PROVIDED */

    do {
        get_system_time(&time_now);
        switch (time_cmp(&time_now, &time_last)) {
        case -1:
            /* Time went backwards. */
            new_clock_seq();
            time_adjust = 0;
            break;
        case 1:
            time_adjust = 0;
            break;
```

```
            default:
                if (time_adjust == 0x7FFF)
                    /* We're going too fast for our clock; spin. */
                    got_no_time = 1;
                else
                    time_adjust++;
                break;
            }
        } while (got_no_time);

        time_last.lo = time_now.lo;
        time_last.hi = time_now.hi;

        if (time_adjust != 0) {
            ADD_16b_2_64b(&time_adjust, &time_now, &time_now);
        }

        /* Construct a uuid with the information we've gathered
         * plus a few constants. */
        uuid->time_low = time_now.lo;
        uuid->time_mid = time_now.hi & 0x0000FFFF;
        uuid->time_hi_and_version = (time_now.hi & 0x0FFF0000) >> 16;
        uuid->time_hi_and_version |= (1 << 12);
        uuid->clock_seq_low = clock_seq & 0xFF;
        uuid->clock_seq_hi_and_reserved = (clock_seq & 0x3F00) >> 8;
        uuid->clock_seq_hi_and_reserved |= 0x80;
        memcpy(uuid->node, &eaddr, sizeof uuid->node);
    }
```