POST Memory Manager

Specification

Version 1.01

November 21, 1997

TABLE OF CONTENTS

1 Introduction

1.1 Revision History

Revision	Date	Changes
1.01	November 21, 1997	Included guidelines on using extended memory during POST. Clarified the processor mode and the state of Gate A20 during POST. Definitions for the terms: Big Real Mode, Extended Memory, and Gate A20 were added. Changed to not clear the contents of memory blocks when they are deallocated. Simplified the assembly language coding example by using 32-bit instructions and operands. Clarified the 'C' language code example by adding a function to find the PMM structure. Updated the contact information for the technical editors.
1.0	September 20, 1996	Approved for public release.

1.1.1 Technical Editors

Scott Townsend
Phoenix Technologies Ltd.
135 Technology Drive
Irvine, CA 92618
phone: (714) 790-2125
fax: (714) 790-2001
email: Scott_Townsend@Phoenix.com

Bob Hale
Intel Corporation
5200 N.E. Elam Young Parkway
Hillsboro, OR 97124-6497
phone: (503) 696-4249
fax: (503) 648-6705
email: robert_p_hale@ccm2.hf.intel.com

1.2 Related Documents

Title	Author	Version
BIOS Boot Specification	Phoenix, Intel, Compaq	1.01
Plug and Play BIOS Specification	Compaq, Phoenix, Intel,	1.0A
EISA Specification	BCPR Services, Inc.	3.12

1.3 Terms

Big Real Mode
Big Real Mode is a modified version of the processor's real mode with the segment limits changed from 1MB to 4GB. Big real mode allows the BIOS or an Option ROM to read and write extended memory without the overhead of protected mode. The BIOS puts the processor in big real mode during POST to allow simplified access to extended memory. The processor will be in big real mode while the PMM Services are callable.

BIOS
The *Basic Input Output System* is the system software embedded on a chip located on the computer's main board. The BIOS executes POST to test and initialize the system components and then boots the operating system. The BIOS also handles the low-level input/output to the various peripheral devices connected to the computer at run time. Additionally, most BIOSes have a Setup program that allows the user to configure the system.

Extended Memory
The *Extended Memory* area starts at memory address 1MB and ends at memory address 4GB. Extended memory is normally only accessible when the processor is in protected mode. One exception to this is big real mode (see above). Section 3.2.4 provides guidelines as to how a PMM client may access extended memory.

Gate A20
Gate A20 controls 1MB memory wrap-around. When Gate A20 is enabled, it forces memory accesses to wrap around and fall within the 0-1MB area by forcing address line 20 to be zero. This has the effect of not allowing access to extended memory. When Gate A20 is disabled, memory accesses beyond 1MB do not wrap around, thus allowing access to all of extended memory.

Option ROM
An *Option ROM (Read Only Memory)* is a software component located in a ROM chip on an add-in card or the system board. Its physical address is in system memory between addresses C0000h and DFFFFh. The BIOS may copy the component to shadow memory during POST. An Option ROM is characterized by the first two locations containing a two-byte signature of 55h AAh. Option ROMs are responsible for initializing their associated hardware, allowing it to be available to the rest of the system for booting or run time.

Paragraph
A *Paragraph* is 16 contiguous bytes of memory. Paragraph alignment of data means that the address of the data is of the form xxxx0h.

PMM
The *POST Memory Manager* is a software component of the BIOS that provides for the allocation of memory blocks during system POST.

POST
The *Power-On Self-Test* is the part of the BIOS that takes control immediately after the computer is turned on. POST initializes the computer hardware in preparation for loading the operating system.

Run-Time
Execution of run-time software takes place after the operating system has loaded. BIOS run-time services are available at POST and remain callable after the operating system has booted. Application programs and operating systems call BIOS run-time services for hardware-related functions. The PMM Services are not callable during run-time, and buffers allocated by the PMM during POST are not available at runtime.

2 Functionality

2.1 Overview

This specification describes the functionality and interface of the POST Memory Manager (PMM). The PMM provides memory allocation only during POST. Memory blocks allocated during POST will not be available during run time because it is assumed that the operating system gains ownership of all memory when it loads.

2.1.1 Why a PMM?

The need for a PMM arises out of the increased usage of memory by internal BIOS modules, as well as the burgeoning size of Option ROMs. As the system BIOS has become more complex, it has been split up into separate components, each with their own requirements for execution. Since PCI Option ROMs are shadowed and have a mechanism for reducing their final run-time footprint, they often utilize a memory block for storage of initialization data. In addition, many newer PCI Option ROMs are now using compression and need a memory block to decompress their code into before finalizing their run-time image.

The capability to organize the usage of memory during POST is necessary in order to keep various software components from over-writing each other's data areas. Option ROMs can utilize the PMM to allocate memory blocks for usage during initialization without danger of corrupting the data area of a system BIOS component or another Option ROM.

2.1.2 PMM Model

The PMM utilizes a client/server model, in which the server is the PMM Services and the clients are internal BIOS code and Option ROMs. Clients allocate memory blocks from the PMM utilizing the PMM Services. The PMM allows clients to allocate either conventional or extended memory blocks to aid in their initialization. A buffer may be used for a scratch data area or for execution of code.

As the name implies, the PMM is available only during POST. The duration of availability of the PMM to software components within the BIOS is implementation specific. Typically, BIOS clients can take advantage of the PMM as soon as memory initialization and testing is completed. For Option ROMs, the PMM is available when their initialization vector is called. The PMM is not available after the system BIOS has invoked INT 19h. All memory allocated by the PMM will be automatically zeroed and freed just before INT 19h is executed. The well-behaved Option ROM deallocates what it knows will not be used again.

The PMM will be available when the Boot Connection Vector (BCV) is called for Option ROMs that have a $PnP header and contain a valid BCV. The PMM will not be available when the Bootstrap Entry Vector (BEV) is called, since this occurs after INT 19h is executed. Details of $PnP Option ROM initialization and booting may be found in the BIOS Boot Specification.

2.2 Clients

Clients of the PMM are both internal BIOS code modules and Option ROMs. Clients allocate memory blocks in paragraph (16-byte) chunks and free these blocks when they are done using them. The execution environment requires that the CPU be in big real mode. The PMM Services provide a real mode far call interface only. Protected mode access is not currently defined. Option ROM clients should not use BIOS interrupt 15h services to access extended memory. A client must not call interrupt 15h functions AH=87h or AH=89h, or modify Gate A20, as this may cause a catastrophic system failure.

3 PMM Services Interface

Clients that access the PMM do so through a set of function calls collectively referred to as PMM Services. PMM Services reside within the system BIOS and are available to both internal BIOS clients as well as Option ROM clients once the presence of PMM Services has been detected.

In a system BIOS implementation of the POST Memory Manager Specification, all three PMM Services function calls are required to be present.

3.1 Detecting PMM Services

A data structure exists within the BIOS for PMM presence detection. The PMM Structure is located in the system BIOS address space on a paragraph boundary between segment addresses E000h and FFFFh. There will be only one PMM Structure in the system BIOS. The presence of the PMM Structure indicates that the PMM Services are present and available for calling.

3.1.1 PMM Structure

Offset	Name	Size	Value	Description
00h	*SignatureByteOne*	BYTE	'$'	Signature byte 1.
01h	*SignatureByteTwo*	BYTE	'P'	Signature byte 2.
02h	*SignatureByteThree*	BYTE	'M'	Signature byte 3.
03h	*SignatureByteFour*	BYTE	'M'	Signature byte 4.
04h	*StructureRevision*	BYTE	01h	Structure revision.
05h	*StructureLength*	BYTE	Varies	Length of this structure in bytes.
06h	*StructureChecksum*	BYTE	Varies	Checksum update field.
07h	*EntryPoint*	FAR PTR	Varies	Segment:Offset of PMM Services entry point.
0Bh	*Reserved*	5 BYTEs	0	Reserved

The StructureRevision field will be incremented if the PMM Structure changes. All fields up to and including the EntryPoint field are guaranteed not to change location within the PMM Structure. Any change to the PMM Structure will be at or beyond the Reserved field only.

3.1.2 Detection Algorithm

A client follows this procedure to locate and access PMM Services:

1. Search for the four-byte "$PMM" string on paragraph boundaries starting at E000h, and ending, if not found, at FFFFh.
2. Verify that the PMM Structure data is valid by performing a checksum. The checksum is calculated by doing a byte-wise sum of the entire PMM Structure and comparing this sum with zero. If the checksum is not zero, then the PMM Structure data is not valid and the EntryPoint field should not be called.
3. Optionally inspect the StructureRevision field to determine the appropriate structure map. The StructureRevision field changes if previously reserved fields in the PMM Structure are redefined to be valid fields.
4. Make calls to the EntryPoint field in the PMM Structure to allocate and free memory as desired.

3.2 PMM Services

The functions described below define PMM Services. The calls are defined as if called from the C programming language. The PMM interface uses the standard C large model calling convention. In particular, return values of type unsigned long are returned in the DX:AX register pair.

3.2.1 Interface Style

The EntryPoint field in the PMM Structure points to the PMM Services interface. Clients call the PMM Services by executing a far call to the address specified in the EntryPoint field. PMM services return to the client via a far return. Values returned to the caller are placed in the DX:AX register pair. The flags and all registers, other than DX and AX, are preserved across calls to PMM services.

3.2.2 Stack Requirements

PMM Services requires a minimum stack size of 256 bytes. BIOSes that support the PMM must provide a stack size of at least 1KB when calling an initialization vector, a Boot Connection Vector, or a Bootstrap Entry Vector of a PCI or PnP ISA Option ROM.

3.2.3 Memory Block Alignment

The default alignment of memory blocks is on a paragraph boundary. Alignment of a memory block on a particular memory boundary (larger than a paragraph) is supported by specifying the desired alignment in the *length* parameter of the pmmAllocate function.

3.2.4 Accessing Extended Memory

This section specifies how clients should access extended memory blocks allocated by the PMM. When control is passed to an option ROM from a BIOS that supports PMM, the processor will be in big real mode, and Gate A20 will be disabled (segment wrap turned off). This allows access to extended memory blocks using real mode addressing.

In big real mode, access to memory above 1MB can be accomplished by using a 32-bit extended index register (EDI, etc.) and setting the segment register to 0000h. The following code example assumes that the pmmAllocate function was just called to allocate a block of extended memory, and DX:AX returned the 32-bit buffer address.

```
; Assume here that DX:AX contains the 32-bit address of our allocated buffer.

; Clear the DS segment register.
push      0000h
pop       ds

; Put the DX:AX 32-bit buffer address into EDI.
mov       di, dx                ; Get the upper word.
shl       edi, 16               ; Shift it to upper EDI.
mov       di, ax                ; Get the lower word.

; Example: clear the first four bytes of the extended memory buffer.
mov       [edi], 00000000h      ; DS:EDI is used as the memory pointer.
```

In a similar way, the other segment registers and 32-bit index registers can be used for extended memory accessing.

WARNING! Clients must not modify the state of Gate A20, put the processor in protected mode, or call BIOS Interrupt 15h, functions 87h or 89h. Any of these actions would alter the big real mode of the processor and could lead to a catastrophic system failure.

3.3 pmmAllocate - Function 0

3.3.1 Description

The pmmAllocate function attempts to allocate a memory block of the specified type and size, and returns the address of the memory block to the caller. The memory block is a contiguous array of paragraphs whose size is specified by the length parameter. The contents of the allocated memory block are undefined and must be initialized by the client.

The entryPoint identifier is of type function pointer, and is a far address. It must be assigned the value of the EntryPoint field in the PMM structure before it can be called.

The return value is of type unsigned long, and represents the 32-bit physical memory address of the memory block that the PMM has allocated. A return value of 0x00000000 indicates that an error occurred and no memory has been allocated. A return value of 0xFFFFFFFF indicates that the client called the PMM Services with an invalid function number.

3.3.2 Function Prototype

```
unsigned long (*entryPoint)(
      unsigned int function,        // 0 for pmmAllocate
      unsigned long length,         // in paragraphs
      unsigned long handle,         // handle to assign to memory block
      unsigned int flags            // bit flags specifying options
);
```

3.3.3 Parameters

`function`

0 for pmmAllocate. Invalid values for the function parameter (0x0003…0xFFFF) cause an error value of 0xFFFFFFFF to be returned, signaling that the function is not supported.

`length`

The size of the requested memory block in paragraphs, or if length is 0x00000000, no memory is allocated and the value returned is the size of the largest memory block available for the memory type specified in the flags parameter. The alignment bit in the flags register is ignored when calculating the largest memory block available.

`handle`

A client-specified identifier to be associated with the allocated memory block. A handle of 0xFFFFFFFF indicates that no identifier should be associated with the block. Such a memory block is known as an "anonymous" memory block and cannot be found using the pmmFind function (see below). If a specified handle for a requested memory block is already used in a currently allocated memory block, the error value of 0x00000000 is returned.

`flags`

A bitmap used by the client to designate options regarding memory allocation.

Bits	Field	Value	Description
1..0	*MemoryType*	1..3	0 = Invalid 1 = Requesting conventional memory block (0 to 1MB). 2 = Requesting extended memory block (1MB to 4GB). 3 = Requesting either a conventional or an extended memory block, whichever is available.
2	*Alignment*	0..1	0 = No alignment. 1 = Use alignment from the length parameter.
15..3	*Reserved*	0	Reserved for future expansion, must all be zero.

`flags.MemoryType`

Specifies whether the requested memory block should be allocated from conventional memory, extended memory, or from either conventional or extended memory. At least one of the bits in this field must be set. If bit 0 is set, the PMM will attempt to allocate only from conventional memory. If bit 1 is set, the PMM will attempt to allocate only from extended memory. If both bits 0 and 1 are set,

the PMM will first try to allocate from conventional memory, and if that fails, the PMM will then attempt to allocate from extended memory.

`flags.Alignment`
Specifies whether the requested memory block should be aligned on a specific memory address boundary or not. The alignment is defined as the least significant set bit of the length parameter. For example, if the length parameter was 0x00000500, then the alignment would be on a 0x100-paragraph boundary. This is equivalent to a 20KB buffer that is aligned on a 4KB boundary. If the length parameter is 0x00000000, the alignment bit is ignored.

`flags.Reserved`
Must all be zero.

3.4 pmmFind - Function 1

3.4.1 Description

The pmmFind function returns the address of the memory block associated with the specified handle.

The entryPoint identifier is of type function pointer, and is a far address. It must be assigned the value of the EntryPoint field in the PMM structure before it can be called.

The return value is of type unsigned long, and represents the 32-bit physical memory address of the memory block that is associated with the handle. A return value of 0x00000000 indicates that the handle does not correspond to a currently allocated memory block.

3.4.2 Function Prototype

```
unsigned long (*entryPoint)(
      unsigned int function,        // 1 for pmmFind
      unsigned long handle          // handle assigned to memory block
);
```

3.4.3 Parameters

`function`
1 for pmmFind. Invalid values for the function parameter (0x0003…0xFFFF) cause an error value of 0xFFFFFFFF to be returned, signaling that the function is not supported.

`handle`
An identifier specified by the client that was assigned to a memory block when the pmmAllocate function was called. If called with an anonymous handle (0xFFFFFFFF) or a currently undefined handle, a value of 0x00000000 will be returned indicating that no associated memory block was found.

3.5 pmmDeallocate - Function 2

3.5.1 Description

The pmmDeallocate function frees the specified memory block that was previously allocated by pmmAllocate., Memory blocks are ***not*** cleared to all zeros by the PMM before being deallocated. Memory below 1MB is cleared by the BIOS before OS boot, but deallocated extended memory blocks in particular will not explicitly be cleared. Clients of the PMM should not rely on the contents of deallocated memory blocks.

The entryPoint identifier is of type far function pointer. It must be assigned the value of the EntryPoint field in the PMM structure before it can be called.

The return value is of type unsigned long. If the memory block was deallocated correctly, the return value is 0x00000000. If there was an error, the return value is non-zero. The error return values are implementation dependent, so long as they are non-zero.

3.5.2 Function Prototype

```
unsigned long (*entryPoint) (
     unsigned int function,        // 2 for pmmDeallocate
     unsigned long buffer          // value returned by pmmAllocate
);
```

3.5.3 Parameters

`function`

2 for pmmDeallocate. Invalid values for the function parameter (0x0003…0xFFFF) cause an error value of 0xFFFFFFFF to be returned, signaling that the function is not supported.

`buffer`

The 32-bit physical address of the memory block. This is the value that was returned by the pmmAllocate function.

3.6 C Language Coding Example

The following C language code example illustrates how to use PMM Services. It finds the PMM entry point, allocates a 16KB conventional memory block, verifies the block is found by using its handle, and deallocates the memory block. No error checking is performed.

```
// C library include files:
#include <stdlib.h>

//
// Structure templates and type definitions:
//
typedef unsigned char UCHAR;
typedef unsigned int UINT;
typedef unsigned long ULONG;
typedef struct {                              // PMM Structure
        ULONG signature;
        UCHAR revision;
        UCHAR length;
        UCHAR checkSum;
        UCHAR *entryPoint;
        UCHAR reserved[5];
} PMM_STRUCTURE;

//
// Equate definitions:
//
#define PMM_ALLOCATE          0             // PMM function numbers.
#define PMM_FIND              1
#define PMM_DEALLOCATE        2
#define BUFFER_SIZE           1024          // number of paragraphs for 16KB buffer.
#define HANDLE                0x12345678    // Handle to test PMM.
#define PMM_SIGNATURE         0x4D4D5024    // "$PMM" string in DWORD format.
#define NULL_FUNCTION_PTR     ((ULONG) 0)

//
// Function declarations:
//
void main(void);
ULONG findPMMEntry(void);

//
// Start of main program.
//
void main(void)
{
        ULONG (*pmmEntry)(), findPMMEntry(), result;
        UCHAR *buffer, *buffer_alias;

        //
        // Get PMM Services entry point, exit if not found.
        //
        if((pmmEntry = (ULONG (*)()) findPMMEntry()) == NULL_FUNCTION_PTR);
                exit(1);

        //
        // Allocate a 16KB buffer and assign our own handle to it.
        //
        buffer = (UCHAR *) (*pmmEntry)(PMM_ALLOCATE, BUFFER_SIZE, HANDLE, 1);

        //
        // Look up the buffer address based on our known handle.
        //
        buffer_alias = (UCHAR *) (*pmmEntry)(PMM_FIND, HANDLE);

        //
        // Deallocate the buffer.
        //
        result = (*pmmEntry)(PMM_DEALLOCATE, buffer);
}
```

```
//
// Searches for the existence of PMM Services and returns the entry point if found.
//
ULONG findPMMEntry(void)
{       ULONG searchAddress;
        ULONG *memPointer;
        UCHAR *checksumPointer;
        PMM_STRUCTURE *pmmStruct;
        UINT i;
        UCHAR sum;
        //
        // Search the UMB region on paragraph boundaries from E0000h to FFFF0h
        // for "$PMM".  If found, verify the checksum and return the entry point.
        // Else, return null.
        //
        for(searchAddress = 0x0000E000; searchAddress < 0x00010000; ++searchAddress)
        {
                //
                // Convert to an x86-sytle far pointer
                //
                memPointer = (ULONG *) (searchAddress << 16);
                if (*memPointer == (ULONG) PMM_SIGNATURE)           {

                        //
                        // Found "$PMM" signature, calculate the structure check sum.
                        //
                        checksumPointer = (UCHAR *) memPointer;
                        pmmStruct = (PMM_STRUCTURE *) memPointer;
                        for(i = 0, sum = 0; i < sizeof PMM_STRUCTURE; ++i)
                        {
                                sum += *checksumPointer;
                                ++checksumPointer;
                        }

                        //
                        // Did the structure check sum verify?
                        //
                        if(sum == 0){
                                //
                                // Yep, return the entry point.
                                //
                                return((ULONG) pmmStruct->entryPoint);
                        }
                }
        }

        //
        // Structure not found or didn't checksum correctly.
        //
        return(NULL_FUNCTION_PTR);
}
```

3.7 Assembly Language Coding Example

The following assembly language coding example performs the equivalent of the C language example above. Note the use of 386 instructions (pushd, etc.) and 32-bit instruction operands.

```
    ;
    ; Find the PMM Services entry point and checksum the structure.
    ;
    call      findPMMEntry          ; DS:SI will point to PMM Structure.

    ;
    ; Allocate a 16KB buffer and assign our own handle to it.
    ;
    push      0001h                 ; Specify conventional memory.
    pushd     12345678h             ; Our handle is 12345678h.
    pushd     00000400h             ; Buffer size is 16kb (400h paragraphs).
    push      0000h                 ; Specify allocate - function 0.
    call      DWORD PTR [si+7]      ; Call the entry point in the structure.
    add       sp, 12                ; Clean up stack after call - C style.
    cmp       dx, 0000h             ; Buffer address is in DX:AX (32-bit)
    jne       allocSuccess
    cmp       ax, 0000h
    jne       allocSuccess
    jmp       failed                ; Return value of 00000000h is an error.

allocSuccess:
    ;
    ; Save our buffer address from DX:AX into EDI.
    ;
    mov       di, dx                ; Put the upper word into DI.
    shl       edi, 16               ; Shift it to upper EDI.
    mov       di, ax                ; Put the lower word into EDI.

    ;
    ; Look up the buffer address based on our known handle.
    ;
    pushd     12345678h             ; Our handle is 12345678h.
    push      0001h                 ; Specify find - function 1.
    call      DWORD PTR [si+7]      ; Call the entry point in the structure.
    add       sp, 6                 ; Clean up stack after call - C style.
    cmp       dx, 0000h             ; Buffer address is in DX:AX (32-bit)
    jne       findSuccess
    cmp       ax, 0000h
    jne       findSuccess
    jmp       failed                ; Return value of 00000000h is an error.

    ;
    ; Deallocate the buffer.
    ;
findSuccess:
    push      edi                   ; Retrieve buffer address from EDI.
    push      0002h                 ; Specify deallocate - function 2.
    call      DWORD PTR [si+7]      ; Call the entry point in the structure.
    add       sp, 6                 ; Clean up stack after call - C style.
    cmp       dx, 0000h             ; Return value is in DX:AX (32-bit)
    jne       passed
    cmp       ax, 0000h
    jne       passed
failed:                             ; Return value of 00000000h is an error.

passed:                             ; No error occurred.
```

4. Conventions for Creating Memory Block Handles

Although most clients of the PMM will typically use an anonymous handle (0xFFFFFFFF), the PMM supports named memory regions (via handles) which persist until deallocated or until POST completes. Since these services are available for use by Option ROMs, the possibility occurs that a handle allocated by the BIOS or by one vendor's Option ROM may be the same as the handle another vendor's Option ROM is designed to use.

This section describes standard mechanisms which this can avoid "handle collision". PMM implementations will not (and, in some cases, cannot) validate handles requested by Option ROMs for compliance with these conventions. Compliance with these conventions supports the best interests of the vendors and industry-wide interoperability.

4.1 Name Selection

4.1.1 EISA Product Identifiers

The convention defined here relies on the use of Plug and Play / EISA Product Identifiers ("PnP ID"s).

These IDs consist of two parts. The first part is a manufacturer identifier of 3 uppercase English letters (A through Z) compressed into 16 bits, using 5 bits per character with 1 for "A" up to 26 for "Z". The second part is a 16-bit value defined by the manufacturer. These two parts are concatenated together to form a 32-bit value:

First	second	third	fourth	byte
0				most significant bit is zero
Aaaaa				compressed first character
bb	bbb			compressed second character
	ccccc			compressed third character
		nnnnnnnn	nnnnnnnn	manufacturer defined

The values are generally written in documentation as the 3 ASCII characters followed by 4 hexadecimal digits, e. g. PNP0C03. The manufacturer identifiers are unique and must be requested from BCPR Services, Inc. as described in the "EISA Specification Version 3.12", BCPR Services, Inc., 1992.

4.1.2 Convention for Selecting Handle Values

This specification requires that a card's Option ROM must use handles that comply with the card manufacturer's PnP ID. That is to say, the upper 16 bits of a handle are the manufacturer's vendor identifier, whereas the vendor defines the lower 16 bits.

The handles used by an Option ROM are not required to correspond to any IDs of the cards on which the Option ROMs making the requests reside. For example, if a PnP ISA card has a PnP ID of "XYZ0000", its Option ROM may validly make a request for memory with the handle "XYZ0000" or "XYZ0123" but not "ABC0000".

This distinction between the card's ID and handle values (while still retaining association with the vendor via the vendor identifier) has several consequences. For example, it provides the flexibility for this convention to support Option ROMs for cards that do not have PnP IDs (for example PCI cards). Also, a card may allocate multiple named blocks without having multiple PnP IDs.

The BIOS reserves for itself all ids with the high order bit set and any ID with the high-order 6 bits equal to zero. The BIOS may or may not flag attempts by Option ROMs to use these ids as an error.

Disclaimer: In cases where add-in cards and Option ROM code is purchased from one company for distribution by a different company, the exact definition of *who* is the manufacturer of a particular peripheral may become murky. It is up to the various parties involved to establish their own practices within the guidelines of this document. A case where this may arise is a vendor of Option ROMs selling those to several card manufacturers.

Note that Option ROMs may allocate anonymous handles without fear of colliding with previously allocated handles.

4.2 Recommended Method for the Use of Named Blocks

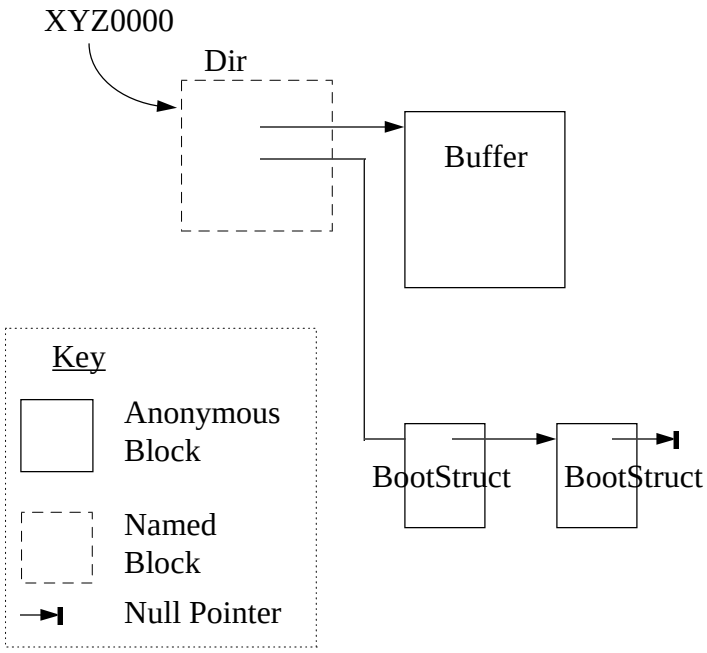
In general, only one named region is needed per card. The named block is allocated and used as a list of pointers to unnamed blocks, functioning somewhat like a directory in a file system.

As an example, assume that the hard disk controller card marketed by XYZ has an Option ROM. During its initialization process, the card's Option ROM needs to allocate a buffer for reading from the hard disks it controls and a linked list of blocks describing each possible bootable hard disk. The following pseudo-code would implement this scenario. (Error recovery, while important, is not shown for the sake of clarity.)

```
#define NoId 0xFFFFFFFF   // The ID passed in for anon blocks.
#define XYZ0000 ( (('X'-'A'+1) << 26) | \
                  (('Y'-'A'+1) << 21) | \
                  (('Z'-'A'+1) << 16) )
#define ConvMem 1         // allocate blocks between 0 and 640k
struct dir {
  byte *buffer;
  struct BootStruct *boot;
};
...
struct dir *base, *t;
...
// readability function
unsigned long pmmAllocate (unsigned long handle, size) {
  entryPoint = findPMMEntry();
  return (*entryPoint)(PMM_ALLOCATE, size, handle, ConvMem);
}
...
// (n+15)/16 calculates the required number of paragraphs given
// n equal to the structure's length in bytes.
base = pmmAllocate(XYZ0000, (sizeof(dir)+15)/16);
...
// allocate the buffer.
base->buffer = pmmAllocate(NoId, (BufLen+15)/16);
...
base->boot = NULL;
for each device {
  // do stuff
  if (device is bootable) {
    t = pmmAllocate(NoId, (sizeof(BootStruct)+15)/16);
    t->next = base->boot;
    base->boot = t;
  }
}
```

This means that, in general, only one named block is required per Option ROM. (The value of XYZ0000 is 0x633A0000.) Visually, the resulting data structure is:

Support of Multiple Structures With One Named Block



4.3 Finding Other Cards in the System

The use of PnP IDs for named handles has another consequence: it makes it possible to perform a simplified search for other similar cards in the system.

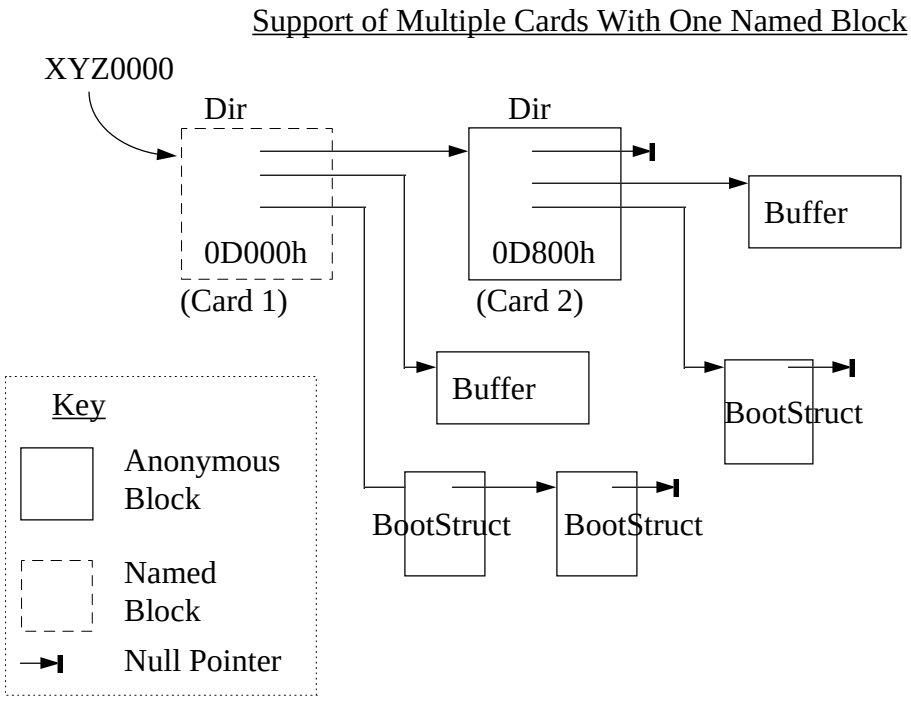
Assume that XYZ's Option ROMs always attempt to allocate a block of memory named "XYZ0000". If two of XYZ's Option ROMs exist in the same system, the second request to allocate will fail. If the company's Option ROMs instead do a pmmFind for the handle XYZ0000 prior to attempting to allocate memory with the same handle, the Option ROM can use the previously initialized allocation.

In companies with more complex product lines, each set of products may need to use a different "family" ID (e. g. XYZ0000 for one family, XYZ0100 for another, etc.). Pseudocode implementing multiple card support is shown below.

```
#define NoId 0xFFFFFFFF
#define XYZ0000 ( (('X'-'A'+1) << 26) | \
                  (('Y'-'A'+1) << 21) | \
                  (('Z'-'A'+1) << 16) )
struct dir {
  struct dir *nextcard;   // point to next directory
  byte *buffer;
  struct BootStruct *boot;
  int cardsCS;
}
...
struct dir *base, *t;
...
// pmmAllocate is assumed to be defined as above.
...
if ((base = (*entryPoint)(PMM_FIND, XYZ0000)) != 0) {
  t = pmmAllocate(NoId, (sizeof(dir)+15)/16);
  t->nextcard = base->nextcard;
  base = t;
```

```
} else {
  base = pmmAllocate(XYZ000, (sizeof(dir)+15)/16);
  base->nextcard = NULL;
}
...
base->buffer = pmmAllocate(NoId, (BufLen+15)/16);
base->cardsCS = GetOurCodeSegment();
...
base->boot = NULL;
for each device {
  // do stuff
  if (device is bootable) {
    t = pmmAllocate(NoId, (sizeof(BootStruct)+15)/16);
    t->next = base->boot;
    base->boot = t;
  }
}
```

Visually, the resulting data structure looks like:



[End of Specification]