THIS SHEET HAS BEEN REPLACED

A METHOD FOR BASIC INPUT-OUTPUT SYSTEM (BIOS) CODE COMPRESSION/DECOMPRESSION DISPATCHER

BACKGROUND OF THE INVENTION

Field of the Invention

The present invention is related to personal computers and more particularly to an object-oriented basic input/output system using data compression.

Background Art

In IBM compatible personal computers a set of programs called basic input/output system (BIOS) are encoded in read-only memory (ROM). The BIOS facilitates the transfer of data and instructions between a central processing unit (CPU) and peripheral devices such as disk drives. Computer systems are designed to perform functional tests of the BIOS every time the computer is turned on. When the computer is turned on, BIOS is copied to an area of random access memory (RAM) set aside for it. Since a RAM is much faster acting than a ROM, accessing the BIOS code from RAM results in much faster initialization of the computer.

As more new features are designed into computers, more code has to be stored into the BIOS stored on the ROM. The size of the ROM must be increased to accommodate the additional code, resulting in additional manufacturing costs. It is desirable to decreases product cost by reducing the size of a ROM required to store the BIOS and all of its components. This can be done by decompressing the code as it is stored on the ROM and then decompressing the code at the time it is copied to the RAM.

SUMMARY OF THE INVENTION

Briefly, the present invention provides a method in which memory controller related components, a power on self test (POST) object, and a decompression dispatcher are stored in a read only memory (ROM). The method tests and initializes the memory controller related components;

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

initializes the decompression dispatcher; dispatches the POST object resulting in uncompressed code, and, executes the uncompressed code.

More specifically, different basic input/output system (BIOS) functions are split into components, or objects, and are only decompressed and executed when needed. A decompression dispatcher software takes a request from currently executing BIOS code to decompress another portion of BIOS code that may be needed. The requester can specify whether or not to just decompress the portion of code into memory or to decompress and initialize the decompressed BIOS. The decompression dispatcher can be used to dispatch different portions of BIOS code to different processors in a mulit-processor system.

The BIOS stored in a read only memory (ROM) is organized into a set of components, each being dispatched as an independent executable object after being copied to a shadow memory portion of a random access memory (RAM). Three types of components are defined, initialization components, runtime components and functional components. The initialization components are dispatched during a Power On Self Test (POST) cycle such that the initialization components are active during initialization. The initialization components are terminated prior to loading an operating system. The runtime components are maintained decompressed and executable after loading of the operating system. The functional components are dispatched on an as needed basis. The components can be actively or passively dispatched. Control over a dispatched object is returned to a requester by the decompression dispatcher upon a condition that the component is to be passively dispatched and the passively dispatched component is recorded in a memory manager. Control over a dispatched object is passed to the decompressed component upon a condition that the component is to be actively dispatched. When dispatching components the decompression dispatcher works from a packing list provided in the ROM image by a BIOS build process. To optimize device usage, the binary images that comprise the different components are packed into the final ROM image.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN REPLACED

The packing list provides a detailed description of the packing as well as other important pieces of information regarding the types of components that have been packed into the ROM.

An advantage of this invention is that the decompression dispatcher can be used to embed option ROMs of add-in cards into the BIOS and, by virtue of auto-detection, associate those ROMs with the installed add-in card.

An advantage of this invention is that it provides a way to put more features into existing products and reduce the cost of future products.

A further advantage of this invention is that it provides a solution that is more comprehensive than the prior art by allowing all possible code and text to be compressed.

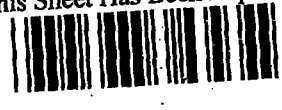
A further advantage of this invention is that it provides a way of customizing BIOSs without having to change individual binary images.

A further advantage of this invention is that it provides a way of distributing BIOS execution to other area of memory besides the standard 0F0000h, and onto multiple processors.

A further advantage of this invention is that it provides a mechanism for selling low-cost PCI add-in cards without having to ship a ROM on each card, because the ROM can be embedded into the BIOS of the mother board.

The foregoing and other objects, features, and advantages of the invention will be apparent from the following more particular description of a preferred embodiment of the invention as illustrated in the accompanying drawings.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN REPLACED

BRIEF DESCRIPTION OF THE DRAWINGS

FIGURE 1 is a block diagram of a computer system in which the present invention is embodied;

FIGURE 2 is a flow diagram of a basic input/output system (BIOS) program in accordance with the present invention;

FIGURE 3 is a flow diagram of the decompression dispatcher of **FIGURE 2**;

FIGURE 4 is a flow diagram of how the decompression dispatcher of **FIGURE 2** is invoked;

FIGURE 5 is a flow diagram of how components are dispatched;

FIGURE 6 is a flow chart of the execution of the dispatched component;

FIGURE 7 is a flow diagram of the get component subclass identifier block (506) of **FIGURE 5**;

FIGURE 8 is a flow diagram of dispatching of components of a class that is protected having 4 gig limits;

FIGURE 9 is a flow diagram of dispatching of components of a class that is protected having 64 K limits;

FIGURE 10 is a flow diagram of removing a component from memory;

FIGURE 11 is a flow diagram of a function returns the size and address of any dispatched component;

FIGURE 12 is a flow diagram of the procedure for getting a specified component transferred and stored in RAM in an uncompressed and executable state; and,

This Sheet Has Been Replaced



FIGURE 13 is a flow diagram of a BIOS build process that automates compressing and packing components into a final ROM binary image.

THIS SHEET HAS BEEN
REPLACED

DESCRIPTION OF THE PREFERRED EMBODIMENT

Refer to **FIGURE 1** which is a block diagram of a computer system in which the present invention is embodied. The computer includes a read only memory, or ROM, (10), a dynamic random access memory, or DRAM, (22), a system memory cache (24), a cache/DRAM controller (20), and central processing units (28, 30). A set of programs called basic input/output system (BIOS) are encoded in ROM (10). The BIOS facilitates the transfer of data and instructions between the central processing units (28, 30) and peripheral devices. In the present invention the BIOS is organized into a set of BIOS components (32 through 54), each of which is capable of being dispatched as an independent executable object. Each BIOS component is a self-contained (link independent) binary image that performs a certain task or function. The BIOS components fall into three major types: initialization components, runtime components and functional components.

The ROM (10) is a non-volatile device used to store the BIOS components, some in a compressed state and some in an uncompressed state. The DRAM (22) stores a shadow RAM binary image of selected components stored in the ROM which have been decompressed and stored in the shadow RAM. In Intel 80386 and 80486 computers, shadow RAM is a portion of upper memory area set aside for programs retrieved from ROM.

Compressed code is the code that resides in the ROM in a compressed state. Decompressed code is code that has been decompressed from its compressed state inside the ROM and now resides in DRAM (22) shadowed memory, in an uncompressed state. Uncompressed code is code that resides inside the ROM in an uncompressed state. This is code that was never compressed.

A component that has been decompressed and activated is an object that is manipulated with modular programming techniques.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

Stored in the ROM (10) are the following components: power-on self-test (POST) object (32); advanced power management (APM) object (34); video BIOS option ROM (36); runtime system BIOS object (38); peripheral component interconnect (PCI) BIOS object (40); small computer system interface (SCSI) option ROM (42); configuration utility (44); server management software object (46); miscellaneous optional ROM(48); component packing list (50); decompression dispatcher (52); and power-on self-test (POST) loader module (54).

The ROM (10) is connected to PCI to ISA Bridge Device (14) via Industry Standard Architecture (ISA) Bus (12). The PCI to ISA Bridge Device (14) is connected to the Peripheral Component Interconnect (PCI) Bus (16) which is connected to the host to PCI bridge device (18).

System random access memory, RAM, (22) and system memory cache (24) are connected to the Cache/DRAM Controller (20) via a local memory bus. The CPU 1 (28) and the CPU 2 (30) is connected to the host to PCI bridge device (18) via the Host CPU Bus (26).

FIGURE 13 is a flow diagram of a BIOS build process that automates compressing and packing components into a final ROM binary image. A list of binary images and associated attributes to include in the final ROM image (1300) and a number of binary image files (1302) are provided to a ROM Packing Utility (1304). The ROM Packing Utility (1304) creates a packed ROM image (1306) and a packing list (1308) that are merged by a merge utility (1310) into a final ROM image (1312) and a packing map (1314) that are stored in the ROM (10) shown in **FIGURE 1**.

Terminology

Compressed Code- Compressed code is the code that resides in the BIOS storage device in a compressed state.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN REPLACED

Decompressed Code- Decompressed code is code that has been decompressed from its compressed state inside the BIOS storage device and now resides in DRAM or Shadowed Memory in an uncompressed state.

Uncompressed Code- Uncompressed code is code that resides inside the BIOS storage device in an uncompressed state, code that never has been compressed.

Storage Device- The non-volatile device used to store the BIOS code (compressed or uncompressed). This may be a FLASH or EEPROM.

Component- A BIOS component is a self-contained (link independent) binary image that performs a certain task or function. A BIOS component could be something as simple as the decompression dispatcher itself or as complex as the entirety of Power On Self Test (POST) software.

Object- An object is a component that has been decompressed and activated.

Direct Dependency- A Direct Dependency is a dependency upon another object or component that is satisfied by a far call or absolute jump instruction.

Indirect Dependency- An Indirect Dependency is a dependency upon another object or component that is satisfied by an API such as an interrupt function or the decompression dispatcher.

Packing Utility- The Packing Utility is the utility used by the BIOS build process to automate compressing and packing Components into the final read only memory (ROM) image.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN REPLACED

Packing Script- The Packing Script is the script used by the Packing Utility to describe the packing order of the ROM image. The Packing Script facilitates the automated build process.

Packing List- The Packing List is generated by the Packing Utility and describes all the attributes of the packed ROM image. This list is used by the decompression dispatcher to dispatch components.

Decompression dispatcher- The decompression dispatcher is the component responsible for all decompression and dispatching activities.

Component Types

The BIOS is organized into a set of components, each being dispatched as an independent executable object. Components fall into three major types: initialization components, runtime components and functional components.

Initialization components are dispatched during what is commonly referred to as the Power On Self Test (POST) cycle. These components are active during initialization and terminated prior to loading the Operating System. Initialization components may contain direct runtime dependencies but these dependencies are limited to objects of the same Component Class. All initialization components must be segment relocatable.

Runtime components are objects that must remain decompressed and executable after loading of the Operating System. Typically, runtime components are combined to form the 64K System BIOS image located in the F0000h segment. Runtime components may not contain any direct dependencies to any other component regardless of its type or class, and indirect dependencies are allowed only to other runtime components.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

Functional components are dispatched on an as needed basis. A Functional component may not contain any direct dependencies to any other component regardless of type or class. An example of a Functional component is a Configuration Utility or a BIOS Setup program.

Component Dispatching

Components are dispatched through a utility called the decompression dispatcher. There two types of dispatching supported by the decompression dispatcher, active and passive.

Components are said to be passively dispatched when the binary image is decompressed into memory and control is returned to the requester by the decompression dispatcher. Passively dispatched components are recorded in the Memory Manager and must be disposed of prior to loading of the boot strap.

Components are said to be actively dispatched when the binary image is decompressed into memory and control is given to the decompressed object. Components must support Real mode execution to be actively dispatched. Control is given directly to the actively dispatched component by the decompression dispatcher.

Control may or may not be returned to the requester depending upon the Component being dispatched. The code segment (CS), instruction pointer (IP) and Flags of the requester remain on the stack when control is given to a actively dispatched component. That component may execute a ADD SP, 6 if it chooses to maintain control or execute an IRET instruction to return to the requester.

When dispatching components the decompression dispatcher works from a Packing List provided in the ROM image by the BIOS build process. To optimize device usage, the binary images that comprise the different components are packed into the final ROM image. The Packing List

This Sheet Has Been Replaced



THIS SHEET HAS BEEN REPLACED

provides a detailed description of the packing as well as other important pieces of information regarding the types of components that have been packed into the ROM.

Class Identification

Each Component is identified by its Class and Sub-Class code. The Class / Sub-Class combination must be unique for each component and there must not be two components of the same Sub-Class within a given Class.

Packing Address

The Packing Address field specifies the offset inside the ROM image where the corresponding Component has been located. The Packing Address is typically provided by the Packing Utility unless specified in the Packing Script.

Unitialized Data components will have a Packing Address of zero (00000000h).

Compressed Size

The Compressed Size field specifies the amount of space (in bytes) consumed by a component when it is in the ROM image.

Unitialized Data components use this field to specify the maximum amount of memory required and be aligned on a 64K boundary.

Decompressed Size

The Decompressed Size field specifies the amount of space (in bytes) consumed by a component after it has been dispatched. Components that are placed in the ROM uncompressed have identical Compressed and Decompressed sizes.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

Unitialized Data components specify this field to be identical to the Compressed Size field.

Load Address

The Load Address field specifies the 32 bit physical address of where the component is to be dispatched. This field must be specified even if the component is intended to be relocatable. Actively dispatched components must specify a Load Address that is below 1Mb and aligned on a paragraph (16 byte) boundary.

Unitialized Data components must specify a load address that is aligned on a 64K boundary. If the Load Address is arbitrary then a value of all FF's (FFFFFFFFh) may be specified. Components referencing Unitialized Data with an arbitrary Load Address must use the Get Unitialized Data Address function to determine the actual Load Address of the Unitialized Data. This function can only be called after the Unitialized Data has been dispatched. Refer to the Memory Management section described below for restrictions on fixed Load Address for Unitialized Data components.

Execution Offset

The Execution Offset field specifies a 16 bit offset into the component where execution is to begin. This offset must be specified for all components that satisfy direct dependencies of other components regardless of whether or not they are to be actively dispatched. Components that do not satisfy direct dependencies of other components should specify this field as FFFFh.

Unitialized Data components must specify an Execution Offset of FFFFh.

Relocation Type

The Relocation Type field specifies what type, if any, of component relocation is supported. Types are either: not relocatable; relocatable below 1MB only; relocatable above 1MB only; or relocatable anywhere.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

Dispatch Type

The Dispatch Type field specifies what type of dispatching is supported by the corresponding component. Types are either: passive dispatching; active dispatching; or both active and passive dispatching.

Component Organization

Though the flexibility of Component Type, Class, and Sub-Class allow unique customization of a particular BIOS, there are five components that must be supported in every BIOS. These five components are: decompression dispatcher, decompression dispatcher unitialized data, POST Loader, power on self test (POST), and system BIOS runtime executable. These five components are described in the following paragraphs.

Decompression Dispatcher

The decompression dispatcher component contains all the functions that comprise the decompression dispatcher. Access is gained through the interrupt interface defined in the decompression dispatcher Functions section described below. This component resides uncompressed in the ROM image.

Decompression Dispatcher Unitialized Data

This component is 128K in length and provides the memory buffers required by the decompression dispatcher.

POST Loader

The POST Loader component handles all hardware generated resets (i.e. power on, front panel, etc.) and software generated resets after being processed by the Shutdown Handler. This component resides uncompressed in the ROM image and may not be dispatched into DRAM.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

The primary responsibility of this module is to configure enough of the system to gain 2Mb of DRAM, activate the decompression dispatcher, and dispatch the main Power On Self Test.

Power On Self Test

The Power On Self Test component (24) is the main POST executable, and is responsible for dispatching all components required for initialization of the system.

System BIOS Runtime Executable

The System BIOS Runtime Executable component is responsible for all compatible System Services including Interrupt and Device Service Routines. This component must also contain the Shutdown Handler responsible for handling all software related resets and dispatching the POST Loader as needed.

Decompression Dispatcher Functions

Dispatch Component ()

The decompression dispatcher component (44) dispatches a Component into memory (12). Each Component is identified by a unique Class / Sub-Class code combination. Dispatched Components are recorded in an Allocation Table kept by the Memory Manager. The actual base address of a Component may be retrieved at any time by calling the Get Physical Address( ) function.

It is the responsibility of the caller to ensure that the Enter Flat Mod( ) has been called and 4G Data Segments are active (this implies that Gate A20 is enabled) before calling this function. Also, if dispatching is being done into shadowed memory (between C0000h-FFFFFh), then the caller must ensure that the appropriate range has been Read/Write enabled in DRAM.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN REPLACED

Unitialized Data Components are also dispatched using this function. All possible requests for Unitialized Data must be satisfied in the Packing List at build time in order for them to be dispatched by this function. Dynamic assigning of Unitialized Data Components is not supported by the decompression dispatcher.

Input:

AH 00h
AL 00h - Passive Dispatch
01h - Active Dispatch
BX Class Code
CX Sub-Class Code
EDI Relocation Address (00000000h if Relocation not Requested.)
ESI Unitialized Data Size (00000000h if Default)

Output - Success:

CF Clear (0)
AH 00h

Output - Failure:

CF Set (1)
AH Error Code
01h - Invalid Function.
02h - Component Packing List entry not found.
03h - Component does not support relocation
04h - Component cannot relocate above 1Mb
05h - Component does not support Active Dispatching
06h - Cannot Actively Dispatch above 1Mb
07h - Active Dispatch aborted because of invalid segment alignment
08h - Cannot allocate memory for component
09h - Decompression failed, dispatch aborted.

This Sheet Has Been Replaced



Get Component Attributes( )

This function returns the information from the packing list corresponding to a given component.

Input:

AH	01h
BX	Class Code
CX	Sub-Class Code

Output - Success:

CF	Clear (0)
AH	Relocation Type
AL	Dispatch Type
EDX	Decompressed Size
SI	Execution Offset
EDI	Default Load Address

Output - Failure:

CF	Set (1)
AH	Error Code
	01h - Invalid Function.
	02h - Component Packing List entry not found.

Get Component SubClass( )

Return the Sub-Class code for entry ***n*** in the Packing List that corresponds to a given Component Class code. To find all components related to a given Class code call this function repeatedly incrementing the Entry Number until a failure is returned.

Input:

AH	02h
AL	Entry Number

BX Class Code

Output - Success:

CF Clear (0)
CX Sub-Class Code

Output - Failure:

CF Set (1)
AH Error Code
01h - Invalid Function.
02h - Component Packing List entry not found.

Enter Flat Mode( )

Enter Flat Mode execution. This function is provided for general component usage and places the system in a mode where the full 4G address range can be accessed through the general segment registers. Gate A20 must be set by the caller prior to using this function. CS and SS are unchanged and have 64K access.

Input:

AH 03h
Gate A20 Must be Enabled.

Output - Success:

CF Clear (0)
DS 0000h (4G Access)
ES 0000h (4G Access)
FS 0000h (4G Access)
GS 0000h (4G Access)

Output - Failure:

CF Set (1)
AH 01h - Invalid Function.

Leave Flat Mode( )

Leave Flat Mode execution. This function turns off the flat mode execution. It is the responsibility of the caller to track and change the state of the Gate A20. No A20 switching is done by this function.

Input:

AH 04h

Output - Success:

CF Clear (0)
DS 0000h (64K Access)
ES 0000h (64K Access)
FS 0000h (64K Access)
GS 0000h (64K Access)

Output - Failure:

CF Set (1)
AH 01h - Invalid Function.

Dispose Component( )

This function removes a component from memory based upon the Class and Sub-Class identifier provided by the caller. An error is returned to the caller if the given component does not have an entry in the Allocation Table. Components that have been disposed of by this function are removed from memory and the memory re-initialized to all zero's (0000h)

Input:

AH 05h
BX Class Code
CX Sub-Class Code

Output - Success:

CF Clear (0)
AH 00h

THIS SHEET HAS BEEN REPLACED

Output - Failure:

CF Set (1)
AH 01h - Invalid Function.
02h - Component Allocation Table entry not found.

Get physical Address( )

Return to the caller the base address and size of any dispatched Component identified by the given Class and Sub-Class code. An error is returned to the caller if the given Component does not have an entry in the Allocation Table.

Input:

AH 06h
BX Class Code
CX Sub-Class Code

Output - Success:

CF Clear (0)
AH 00h
EDI 32 bit Base Address of Component
EDX Size of Component in bytes (1K aligned)

Output - Failure:

CF Set (1)
AH 01h - Invalid Function.
02h - Component Allocation Table entry not found.

Memory Management

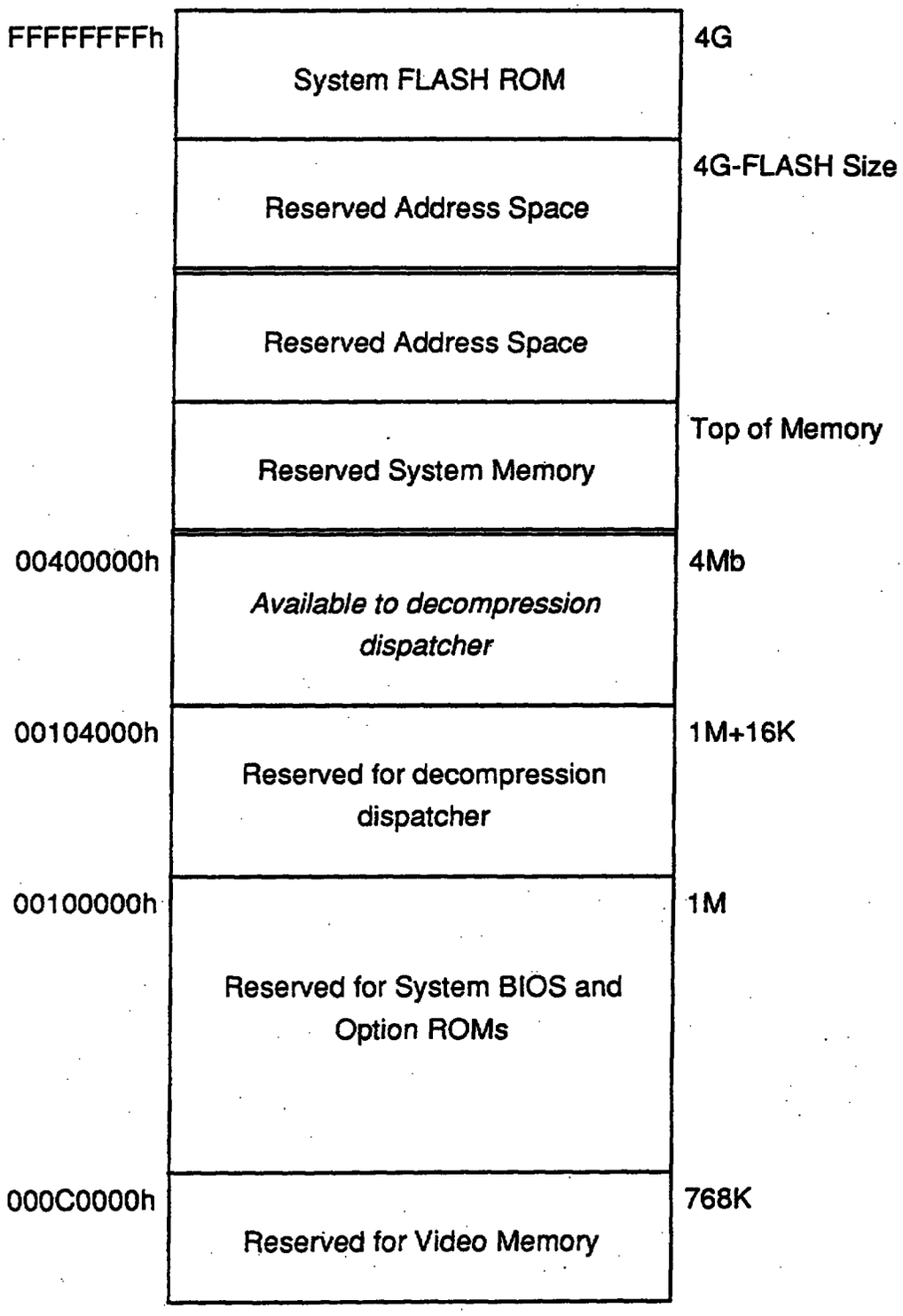
Memory testing and subsequent update of status to a video monitor are all done in 'Real Big' mode as opposed to constantly switching between Real and Protected Mode. 'Real Big' mode is a mode where the data segment registers of the processor are loaded with Protected Mode type

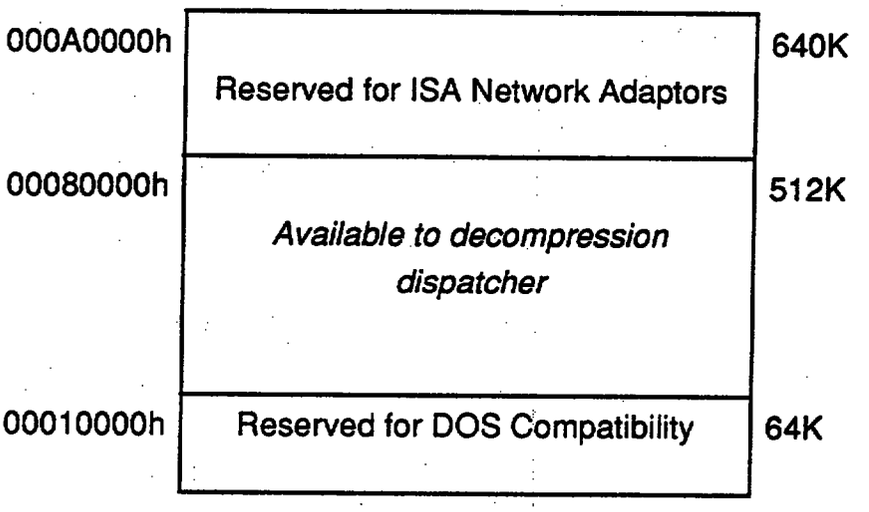
This Sheet Has Been Replaced



characteristics while the code and stack segment registers retain their Real Mode characteristics. The processor essentially runs in Real Mode while allowing a full 4 GB access to memory.

Memory Map


FFFFFFFFh
System FLASH ROM
4G
Reserved Address Space
4G-FLASH Size
Reserved Address Space
Reserved System Memory
Top of Memory
00400000h
Available to decompression dispatcher
4Mb
00104000h
Reserved for decompression dispatcher
1M+16K
00100000h
Reserved for System BIOS and Option ROMs
1M
000C0000h
Reserved for Video Memory
768K



The Memory Manager tracks allocated memory blocks using a 4K byte map and a 6K allocation table. The Memory Manager uses the 4K byte map to specify the allocated blocks of memory. Each byte in the byte map represents a 1K block of memory starting at 0Mb and ending at 4Mb. A set byte (FFh) in a byte location means the represented 1K block of memory has already been assigned to a requester. A clear byte (00h) in a byte location means the represented 1K block of memory is available.

The Memory Manger uses the 6K Allocation Table to identify the assignee of each memory block. There is enough room in the Allocation Table to assign 512 blocks of memory to unique Components. Each entry in the Allocation Table has the following format:

Field	Offset	Length	Description
Class	0h	WORD	Class code of component using memory.
Sub-Class	2h	WORD	Sub-Class code of component using memory.
Base Address	4h	DWORD	32 bit Physical address of memory.
Size	8h	DWORD	32 bit Length of memory block.

When a request is being made for memory the Memory Manager searches the byte map for a block of memory large enough to satisfy the request. This search is conducted on a first fit algorithm. When a location is found, the corresponding byte is set in the byte map and then logged in the first available entry in the Allocation Table.

Interface Functions

Init Memory Manager( )

Alloc Memory Block( )

Reserve a block of memory for a given Component.

Input:

BX	Class Code
CX	Sub-Class Code
EDI	Physical Address (00000000h if not specified.)
ESI	Size of Block to allocate.

Output - Success:

CF	Clear (0)
AH	00h
EDI	Physical Address of allocated memory block

Output - Failure:

CF	Set (1)
AH	07h - Cannot allocate memory for component

Dealloc Memory Block( )

Un-Reserve a block of memory for a given Component.

Input:

BX	Class Code

THIS SHEET HAS BEEN REPLACED

CX Sub-Class Code

Output:
None

Get Memory Block( )
Where am I function.

Input:
BX Class Code
CX Sub-Class Code

Output - Success:
CF Clear (0)
AH 00h
EDI Physical Base Address
EDX Size of Block

Output - Failure:
CF Set (1)
AH 07h - Cannot allocate memory for component

Description of Flow Charts
Refer to **FIGURE 2** which is a flow diagram of a basic input/output system (BIOS) program in accordance with the present invention. The program starts (200), and tests and initializes the memory controller related components (202). Next the program initializes the power on self test (POST) decompression dispatcher (204) shown in more detail in the flow diagram of **FIGURE 3**. Finally, the program dispatches and executes the POST (206) as shown in **FIGURES 4** and **5**, and stops (208).

Refer to **FIGURE 3** which is a flow diagram of a decompression dispatcher of **FIGURE 2**. The dispatcher data and executable code is copied (302) from the ROM to the RAM. The dispatcher interrupt vector is

This Sheet Has Been Replaced

installed (304), the memory allocation table is cleared (306) and the component handle table is cleared (308). The compressed BIOS component information in the packing list is fetched from RAM (310). The relocation type field is examined to find what type of relocation is supported, below 1 meg.(312, 314) , above 1 meg. (316, 318), or anywhere (320). If the required size is not available (322) allocation error flags are set (324). If the specified load address is not available (326) allocation error flags are set (328).

If the required size is available (322), then the memory allocation table is updated (330). If the end of the component list in the packing list has not been reached (332), then the flow returns to block (310). If the end of the component list in the packing list has been reached (332), then the flow ends (334).

Refer to **FIGURE 4** which is a flow diagram of how the decompression dispatcher is invoked. The function to be dispatched requires the passing of and returning of parameters through a stack, which is an area of RAM set aside for temporary storage of the parameters. The portion of the stack holding local variables and parameters of a procedure is called a stack frame. The stack frame is set up (402) and a check is made to determine if the requested function is valid (404). If not, the procedure is halted (406). If yes, then the dispatcher function is executed (408). When the flow returns, the stack frame is restored (410) and the flow ends (412). **FIGURES 5** through **12** are flows for execution of the dispatcher function block (408) of **FIGURE 4**.

Refer to **FIGURE 5** which is a flow diagram of how components are dispatched. The dispatcher assumes only one component from a class is being loaded, and hence the count of components is set to 1. The counter is used as an index into the packing list. The component of a class might contain a number of components referenced by a subclass identifier. A check (504) is made to determine if all components of the class are to be loaded. If yes, then the first subclass identifier is fetched (506) as

THIS SHEET HAS BEEN REPLACED

described in **FIGURE 7**. If the subclass is found in the packing list (508) then the count of the subclass index counter is incremented (510). If the subclass is not found in the packing list (508), then the count of the subclass index counter is not incremented and the flow proceeds to get packing information for the component indexed by the count of the component counter (512). The procedure next uncompresses the component at the index count and places the uncompressed code in RAM (514). The details of block (514) are shown in more detail in **FIGURE 12**. At the end of the uncompressing procedure, a check (516) is made to determine if there are more components in this class to uncompress. If yes, the component index counter is decremented (518). If no, a check (520) is made to determine if this is the last component actively dispatched or if an option ROM is present. If not, the flow ends (524). If yes, then the return address is adjusted to execute the dispatched component (522).

Refer to **FIGURE 6** which is a flow chart of the execution of the dispatched component. The list pointer for the packing list entries is fetched (602). The packing list pointer is checked (604) to determine if the end of the packing list is pointed to. If yes, a search failure flag is set (610). If no, a check (606) is made to determine if the pointer is pointing to the desired entry. If no, the list pointer is advanced (608). If yes, the flow proceeds to block (612). If the search of the packing list is successful (612), then the component decompression information is fetched (614). and this part of the procedure ends (616).). If the search of the packing list is not successful (612), an error is displayed (618) and the procedure is halted (620).

Refer to **FIGURE 7** which is a flow diagram of the get component subclass identifier block (506) of **FIGURE 5**. The sub class index is checked (702) to determine if it is valid. If not valid, an error is displayed (722) and the procedure is halted (724). If yes, the packing list pointer is checked (704) to determine if the end of the packing list is pointed to. If yes, a search failure flag is set (710). If no, a check (706) is made to

This Sheet Has Been Replaced



THIS SHEET HAS BEEN REPLACED

determine if the packing list pointer is pointing to an entry from the desired class. If no, the packing list pointer is advanced (708). If yes, a check (712) is made to determine if the packing list pointer is referencing a desired sub component.. if no, the packing list pointer is advanced (708). If yes, the component code is fetched from the ROM (714) and this portion of the procedure stops (718).

Refer to **FIGURE 8** which is a flow diagram of dispatching of components of a class that is protected having 4 gig limits. The procedure loads the GDTR global descriptor table register (802) with the global descriptor table and then enters protected mode (804). The data segment (DS), extra segment (ES) , global segment (GS) , and general-purpose file segment (FS), registers are configured with 4 gig limits and protected mode is exited (808). The registers DS, ES, FS, and GS are loaded with a 0 base address (810) and the procedure stops (812).

Refer to **FIGURE 9** which is a flow diagram of dispatching of components of a class that is protected having 64 K limits. The procedure loads the GDTR global descriptor table register (902) with the global descriptor table and then enters protected mode (904). The data segment (DS), extra segment (ES) , global segment (GS) , and general-purpose file segment (FS), registers are configured with 64K limits and protected mode is exited (908). The registers DS, ES, FS, and GS are loaded with a 0 base address (910) and the procedure stops (912).

Refer to **FIGURE 10** which is a flow diagram of removing a component from memory. The procedure loads the pointer to the handle table (1002), and checks to see if it is pointing to the correct entry (1004). If not, it checks to see if it is at the end of the table (1006) and if so sets the not found flag (1010) and flow proceeds to block (1014). If it is not at the end of the table the pointer is incremented to the next entry (1008) and the flow returns to block (1004). If the handle pointer is at the correct entry, the not found flag is cleared (1012) and flow proceeds to block (1014). If the not found flag is set the procedure is exited (1022), otherwise the size

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

and address arc retrieved (1018) and the corresponding entry is cleared from the memory allocation table (1018) and the handle entry is cleared from the handle table (1020).

Refer to **FIGURE 11** which is a flow diagram of a function which returns the size and address of any dispatched component. This function loads the pointer to the handle table (1102), and checks to see if it is pointing to the correct entry (1104). If not, it checks to see if it is at the end of the table (1106) and if so sets the not found flag(1110)and flow proceeds to block (1114). If it is not at the end of the table the pointer is incremented to the next entry (1108) and the flow returns to block (1104). If the handle pointer is at the correct entry, the not found flag is cleared(1112)and flow proceeds to block (l114). If the not found flag is set, an error is displayed (1120) and the function is halted (1122), otherwise the size and address are retrieved (l116) and the function is exited (1118).

Refer to **FIGURE 12** which is a flow diagram of the procedure for getting a specified component transferred and stored in RAM in an uncompressed and executable state.

A RAM memory block large enough to hold the specified component is found (1202) and allocated. After the memory space is allocated (1204), the size field in the packing list is accessed to get the size information (1206). The compressed and decompressed size information is obtained from the packing list entry (1206). The Compressed Size field specifies the amount of space (in bytes) consumed by a component when it is in the ROM image. Unitialized Data components use this field to specify the maximum amount of memory required and that they be aligned on a 64K boundary. The Decompressed Size field specifies the amount of space (in bytes) consumed by a component after it has been dispatched. Components that are placed in the ROM uncompressed have identical Compressed and Decompressed sizes. Unitialized Data components specify this field to be identical to the Compressed Size field.

This Sheet Has Been Replaced



If the component is not compressed, then the component is copied (1212) to RAM from the non-volatile storage device used to store the compressed or uncompressed BIOS code. If the component is compressed, then the component is decompressed (1210), stored in an uncompressed state in the shadowed memory portion of RAM, and the procedure stops (1216).

In the event that memory cannot be allocated (1204) an error is displayed (1224) and the procedure halts (1226).

While the invention has been particularly shown and described with reference to preferred embodiments thereof, it will be understood by those skilled in the art that the foregoing and other changes in form and detail may be made therein without departing from the scope of the invention.

THIS SHEET HAS BEEN REPLACED

* C L M S *

What is claimed is:

1. In a data processing system in which are stored in a read only memory (ROM), memory controller related components, a power on self test (POST) object, and a decompression dispatcher,

a method comprising steps of:

A. testing and initializing said memory controller related components;

B. initializing said decompression dispatcher;

C. dispatching said POST object resulting in uncompressed code, and,

D. executing said uncompressed code.

2. The method in accordance with claim 1 wherein

said data processing system includes a random access memory (RAM) having stored therein a dispatcher interrupt vector, and a memory allocation table, and,

said ROM has stored therein dispatcher data, executable dispatcher code, and compressed basic input/output system (BIOS) component information,

said step of initializing said decompression dispatcher further comprising steps of:

A. copying said dispatcher data and executable dispatcher code from said ROM to said RAM;

B. installing said dispatcher interrupt vector;

C. clearing said memory allocation table;

D. getting said compressed BIOS component information from said ROM;

E. updating said memory allocation table;

F. setting up a stack frame;

G. executing said dispatcher code; and,

H. restoring said stack frame upon completion of execution of said dispatcher function.

This Sheet Has Been Replaced

THIS SHEET HAS BEEN REPLACED

* C L M S *

3. The method in accordance with claim 2 wherein

said computer system includes a component index counter,

said dispatcher information includes a packing list of component subclass identifiers, and,

said step of executing said uncompressed code includes steps of:

I. setting a default count of said component index counter to one;

J. getting one component subclass identifier of said component subclass identifiers;

K. searching said packing list for an instance of said one component subclass identifier;

L. incrementing said count of subclass components upon a condition that said sub class is found in said packing list;

M. getting packing information for an indexed component indexed by said count of components;

N. uncompressing said indexed component resulting in an uncompressed component;

O. storing said uncompressed component in said RAM;

P. decrementing said component index counter upon a condition that more components are to be uncompressed in said class.

4. The method in accordance with claim 3 wherein

said computer system includes a list pointer.

said step of getting one component subclass identifier of said component subclass identifiers includes steps of:

Q. advancing said list pointer until said list pointer points to an entry in said packing list from said one class; and,

R. getting a component code corresponding to said entry in said packing list.

5. The method in accordance with claim 4 comprising further steps of:

S. finding a RAM memory block large enough to hold said component code.

6. The method in accordance with claim 5 comprising further steps of:

This Sheet Has Been Replaced

THIS SHEET HAS BEEN REPLACED

* C L M S *

T. allocating said memory block;

U. getting size information from said entry in said packing list;

V. decompressing said component code resulting in decompressed code upon a condition that said component code is compressed; and,

W. storing said decompressed code in said RAM.

7. The method in accordance with claim 5 comprising further steps of:

T. allocating said memory block;

U. getting size information from said entry in said packing list; and,

V. copying said component code from said ROM to said RAM upon a condition that said component code is uncompressed;

W. storing said component code in said RAM.

8. In a data processing system in which,

basic input/output system (BIOS) is organized into three types of components, initialization components, runtime components and functional components,

each of said three types of components capable of being dispatched as an independent executable object,

initialization of said processor being controlled by a power on self test (POST),

general operation of said processor being controlled by an operating system,

a method comprising steps of:

A. dispatching said initialization components during said power on self test (POST) cycle such that said dispatching initialization components are active during said initialization;

B. terminating Initialization prior to loading said operating system;

C. loading said operating system;

E. maintaining said runtime components decompressed and executable after loading of said operating system;

F. dispatching said functional components on an as needed basis;

This Sheet Has Been Replaced

THIS SHEET HAS BEEN REPLACED

* C L M S *

G. decompressing a binary image of a component into memory thereby creating a decompressed object;

H. returning control to said requester by said decompression dispatcher upon a condition that said component is to be passively dispatched;

I. recording a passively dispatched component in a memory manager; and,

J. passing control to said decompressed component upon a condition that said component is to be actively dispatched;

9. The method in accordance with claim 8 wherein

said data processing system includes a random access memory (RAM) having stored therein a dispatcher interrupt vector, and a memory allocation table, and,

said ROM has stored therein dispatcher data, executable dispatcher code, and compressed basic input/output system (BIOS) component information,

said step of initializing said decompression dispatcher further comprising steps of:

A. copying said dispatcher data and executable dispatcher code from said ROM to said RAM;

B. installing said dispatcher interrupt vector;

C. clearing said memory allocation table;

D. getting said compressed BIOS component information from said ROM;

E. updating said memory allocation table;

F. setting up a stack frame;

G. executing said dispatcher code; and,

H. restoring said stack frame upon completion of execution of said dispatcher function.

10. The method in accordance with claim 9 wherein

said computer system includes a component index counter,

This Sheet Has Been Replaced

THIS SHEET HAS BEEN REPLACED

* C L M S *

said dispatcher information includes a packing list of component subclass identifiers, and,

said step of executing said uncompressed code includes steps of:

I. setting a default count of said component index counter to one;

J. getting one component subclass identifier of said component subclass identifiers;

K. searching said packing list for an instance of said one component subclass identifier;

L. incrementing said count of subclass components upon a condition that said sub class is found in said packing list;

M. getting packing information for an indexed component indexed by said count of components;

N. uncompressing said indexed component resulting in an uncompressed component;

O. storing said uncompressed component in said RAM;

P. decrementing said component index counter upon a condition that more components are to be uncompressed in said class.

11. The method in accordance with claim 10 wherein

said computer system includes a list pointer,

said step of getting one component subclass identifier of said component subclass identifiers includes steps of:

Q. advancing said list pointer until said list pointer points to an entry in said packing list from said one class; and,

R. getting a component code corresponding to said entry in said packing list.

12. The method in accordance with claim 11 comprising further steps of:

S. finding a RAM memory block large enough to hold said component code.

13. The method in accordance with claim 12 comprising further steps of:

T. allocating said memory block;

U. getting size information from said entry in said packing list;

This Sheet Has Been Replaced

THIS SHEET HAS BEEN REPLACED

* C L M S *

V. decompressing said component code resulting in decompressed code upon a condition that said component code is compressed; and,

W. storing said decompressed code in said RAM.

14. The method in accordance with claim 12 comprising further steps of:

T. allocating said memory block;

U. getting size information from said entry in said packing list; and,

V. copying said component code from said ROM to said RAM upon a condition that said component code is uncompressed;

W. storing said component code in said RAM.

15. In a data processing system in which are stored in a read only memory (ROM),

a plurality of basic input/output system (BIOS) components,

each one of said BIOS components being a self-contained, link independent, binary image of code that upon execution performs a certain task,

said BIOS components including a decompression dispatcher component a power on self test (POST) component,

a component packing list, said packing list describing all attributes of said packed ROM image; said packing list being used by said decompression dispatcher to dispatch said components;

compressed code in said non-volatile storage device in a compressed state, said compressed code being code that has been compressed,

said method comprising steps of:

A. executing said decompression dispatcher, said decompression dispatcher decompressing said compressed code resulting in decompressed code, said decompressed code being code that has been decompressed from its compressed state inside said non-volatile storage device;

B. storing said decompressed code in a shadowed memory in an uncompressed state as decompressed code; and,

This Sheet Has Been Replaced

THIS SHEET HAS BEEN REPLACED

* C L M S *

C. activating said decompressed code resulting in an object, said object being a component that has been decompressed and activated.

16. A BIOS build process comprising steps of:

A. providing a list of binary images and associated attributes to include in a final ROM image and a number of binary image files to a ROM Packing Utility;

B. invoking said ROM Packing Utility to create a packed ROM image and a packing list;

C. merging said packed ROM image and said packing list into a final ROM image and a packing map; and,

D. storing said final ROM image and said packing map in a ROM.

17. In a data processing system in which are stored in a read only memory (ROM),

a plurality of basic input/output system (BIOS) components,

each one of said BIOS components being a self-contained, link independent, binary image of code that upon execution performs a certain task,

said BIOS components including a decompression dispatcher component a power on self test (POST) component and a BIOS build process,

said build process including a packing utility used by said build process to automate compressing and packing components into a read only memory (ROM) binary image,

said packing utility including a packing script,

a method comprising steps of:

A. invoking said BIOS build process;

B. executing said packing script, said packing script being used by said packing utility to describe a packing order of components packed into said read only memory (ROM) image;

C. invoking said packing utility to generate a packing list, said packing list describing all attributes of said packed ROM image; said

This Sheet Has Been Replaced

THIS SHEET HAS BEEN REPLACED

* C L M S *

packing list being used by said decompression dispatcher to dispatch components;

D. storing uncompressed code in a non-volatile storage device in an uncompressed state, said uncompressed code being code that never has been compressed; and,

E. storing compressed code in said non-volatile storage device in a compressed state, said compressed code being code that has been compressed.

18. The method in accordance with claim 17 comprising further steps of:

F. executing said decompression dispatcher, said decompression dispatcher decompressing said compressed code resulting in decompressed code, said decompressed code being code that has been decompressed from its compressed state inside said non-volatile storage device;

G. storing said decompressed code in a shadowed memory in an uncompressed state as decompressed code; and,

H. activating said decompressed code resulting in an object, said object being a component that has been decompressed and activated.

This Sheet Has Been Replaced