

[19] 中华人民共和国国家知识产权局

[51] Int. Cl.
G06F 9/445 (2006.01)



[12] 发明专利说明书

专利号 ZL 200410102015.5

[45] 授权公告日 2009 年 6 月 10 日

[11] 授权公告号 CN 100498703C

[22] 申请日 2004.12.16

[21] 申请号 200410102015.5

[30] 优先权

[32] 2003.12.16 [33] US [31] 60/530,135

[32] 2004.5.1 [33] US [31] 10/837,150

[73] 专利权人 微软公司

地址 美国华盛顿州

[72] 发明人 A·罗杰斯 J·格劳姆

M·同科洛维茨

[56] 参考文献

CN1266225A 2000.9.13

CN1375082A 2002.10.16

EP1164475A2 2001.12.19

WO01/44934A1 2001.6.21

US2002/0049966A1 2002.4.25

审查员 唐 嫣

[74] 专利代理机构 上海专利商标事务所有限公司

代理人 陈 斌

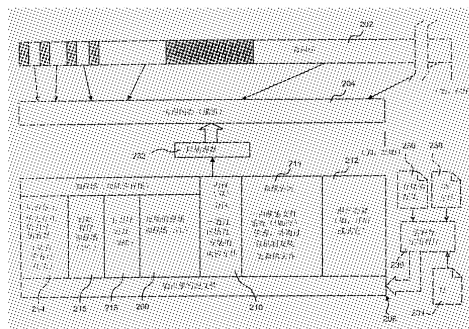
权利要求书 4 页 说明书 34 页 附图 10 页

[54] 发明名称

以存储技术抽象方式在文件内创建文件系统

[57] 摘要

所描述的是一种用于制造计算机设备的系统和
方法，其中，单个制造映象文件包含可向其写入个
别的操作系统组件包的内部分区和文件系统布局。
该映象文件独立于任何特定的存储技术。为生成该
文件，盘映象实用程序输入一存储器配置文件和一
包一分区映射文件，以在该映象文件内创建该分区/
文件系统。操作系统组件(包)被写入分区中。为
在一特定存储设备上储存(如，闪存)该映象文件的
数据，后处理过程该映象文件以定制用于特定存储
设备的数据。



1. 一种在计算环境中用于在包括分区和文件系统的映象文件内创建文件系统、使得所述映象文件能被更容易地更新并且所述文件系统能被容易地适应于各种存储设备的方法，其特征在于，所述方法包括：

访问包含要安装到第一存储媒质上的映象数据的包；

访问将向其写入所述包内容的分区的描述；以及

生成一向第二存储媒质的输出文件，所述输出文件包含使每一所述包的内容基于所述描述储存在其中之一的分区。

2. 如权利要求 1 所述的方法，其特征在于，所述方法还包括访问一存储器配置文件以确定所述输出文件内所述分区的至少一个的位置。

3. 如权利要求 1 所述的方法，其特征在于，至少一个所述包的内容对应于一操作系统组件，其中，所述分区之一包括一系统分区，且所述方法还包括基于所述描述将所述操作系统组件写入所述系统分区中。

4. 如权利要求 1 所述的方法，其特征在于，至少一个所述包的内容对应于一内核组件，其中，所述分区之一包括一基于 RAM 的映象分区，且所述方法还包括基于所述描述将所述内核组件写入所述基于 RAM 的映象分区中。

5. 如权利要求 1 所述的方法，其特征在于，至少一个所述包的内容对应于一内核组件，其中，所述分区之一包括一基于 ROM 的映象分区，并且所述方法还包括基于所述描述将所述内核组件写入所述基于 ROM 的映象分区中。

6. 如权利要求 1 所述的方法，其特征在于，至少一个所述包的内容对应于一更新加载器组件，其中，所述分区之一包括一二进制映象分区，且所述方法还包括基于所述描述将所述更新加载器组件写入所述二进制映象分区中。

7. 如权利要求 1 所述的方法，其特征在于，生成所述输出文件包括向扩展到所述输出文件的末端的分区写入数据、以及基于所写入的实际数据量调整所述分区的大小。

8. 如权利要求 1 所述的方法，其特征在于，生成所述输出文件包括向扩展到所述输出文件的末端的分区写入数据、在所写入的数据的末端之后添加一空闲空间缓冲区、以及基于所写入的实际数据量和所述空闲空间缓冲区调整所述分区的大小。

9. 如权利要求 1 所述的方法, 其特征在于, 所述方法还包括向所述输出文件中的一主引导记录写入数据以定义分区。

10. 如权利要求 9 所述的方法, 其特征在于, 向主引导记录写入数据包括定义一用户存储分区。

11. 如权利要求 1 所述的方法, 其特征在于, 所述方法还包括向所述输出文件写入额外的数据。

12. 如权利要求 1 所述的方法, 其特征在于, 所述方法还包括执行一后处理过程以准备将数据从所述输出文件传输到所述第一存储媒质中。

13. 如权利要求 12 所述的方法, 其特征在于, 所述第一存储媒质包括闪存, 其中的代码可原地执行, 且所述方法还包括访问一存储器配置文件以修正所述分区的至少一个, 使得其中的代码将在原地执行。

14. 如权利要求 1 所述的方法, 其特征在于, 所述方法还包括访问一存储器配置文件以确定所述第一存储媒质的哪些区段被保留。

15. 一种在计算环境中用于在包括分区和文件系统的映象文件内创建文件系统、使得所述映象文件能被更容易地更新并且所述文件系统能被容易地适应于各种存储设备的方法, 其特征在于, 所述方法包括:

访问一包含分区的文件, 所述分区包含要安装到一存储媒质的映象数据, 其中所述映象数据对应于包;

根据一分区—包映射文件和一存储器配置文件生成所述文件; 以及

处理所述文件以在所述分区和文件系统布局中引入一特定存储技术所要求的任何修改。

16. 如权利要求 15 所述的方法, 其特征在于, 所述方法还包括将数据从所述文件传输到对应于所述特定存储技术的存储媒质中。

17. 如权利要求 15 所述的方法, 其特征在于, 所述方法还包括访问所述存储器配置文件以确定所述文件内所述分区的至少一个的位置。

18. 一种在计算环境中用于在包括分区和文件系统的映象文件内创建文件系统、使得所述映象文件能被更容易地更新并且所述文件系统能被容易地适应于各种存储设备的系统, 其特征在于, 所述系统包括:

一个盘映象实用程序组件, 所述盘映象实用程序组件输入一包含分区信息和包信息的分区—包映射文件, 并基于所述分区—包映射文件中的数据, 使用所述包

信息中标识的包输出一映象文件,所述映象文件包括所述分区信息中标识的多个分区,使每一所述包的内容基于所述分区信息储存在所述分区之一中;以及

一后处理器,所述后处理器处理所述映象文件以在所述映象文件中引入一特定存储技术所要求的任何修改。

19. 如权利要求 18 所述的系统,其特征在于,所述盘映象实用程序组件还输入一包含指示如何配置一存储设备的存储器的数据的存储器配置文件。

20. 如权利要求 19 所述的系统,其特征在于,所述存储设备用其代码可在原地执行的存储器来配置,并且其中,访问所述存储器配置文件以修正所述分区的至少一个,使得其中的代码将在原地执行。

21. 如权利要求 19 所述的系统,其特征在于,访问所述存储器配置文件以确定所述存储设备的存储器的哪些区段被保留。

22. 如权利要求 18 所述的系统,其特征在于,至少一个所述包的内容对应于一操作系统组件,其中,所述分区之一包括一系统分区,并且其中,所述盘映象实用程序组件基于所述分区信息将所述操作系统组件写入所述系统分区中。

23. 如权利要求 18 所述的系统,其特征在于,至少一个所述包的内容对应于一内核组件,其中,所述分区之一包括一基于 RAM 的映象分区,并且其中,所述盘映象实用程序组件基于所述分区信息将所述内核组件写入所述基于 RAM 的映象分区中。

24. 如权利要求 18 所述的系统,其特征在于,至少一个所述包的内容对应于一内核组件,其中,所述分区之一包括一基于 ROM 的映象分区,并且其中,所述盘映象实用程序组件基于所述分区信息将所述内核组件写入所述基于 ROM 的映象分区中。

25. 如权利要求 18 所述的系统,其特征在于,至少一个所述包的内容对应于一更新加载器组件,其中,所述分区之一包括一二进制映象分区,并且其中,所述盘映象实用程序组件基于所述分区信息将所述更新加载器组件写入所述二进制映象分区中。

26. 如权利要求 18 所述的系统,其特征在于,所述盘映象实用程序组件向扩展到所述映象文件的末端的分区写入数据,并基于所写入的实际数据量调整所述分区的大小。

27. 如权利要求 18 所述的系统,其特征在于,所述盘映象实用程序组件向扩

展到所述映象文件的末端的分区写入数据、在所写入的数据的末端之后添加一空闲空间缓冲区、并基于所写入的实际数据量和所述空闲空间缓冲区调整所述分区的大小。

28. 如权利要求 18 所述的系统，其特征在于，所述盘映象实用程序组件向所述映象文件中的一主引导记录写入数据，以定义分区。

29. 如权利要求 28 所述的系统，其特征在于，所述盘映象实用程序组件通过向所述主引导记录写入数据定义一用户存储分区。

以存储技术抽象方式在文件内创建文件系统

相关申请的参照

本发明要求 2003 年 12 月 16 日提交的美国临时专利申请序列号 60/530,135 的优先权，该申请整体结合于此。

本申请涉及与其同时提交的以下美国专利申请，这些申请整体结合于此：

摘要号 4271/307,649，“以故障保险方式向非易失存储应用自定义软件映象更新（Applying Custom Software Image Updates To Non-Volatile Storage in a Failsafe Manner）”；

摘要号 4281/307,650，“确定对安装有效的最大依赖软件更新组（Determining the Maximal Set of Dependent Software Updates Valid for Installation）”；

摘要号 4291/307,651，“确保软件更新仅在特定的设备或设备类上安装或运行（Ensuring that a Software Update may be Installed or Run only on a Specific Device or Class of Devices）”；以及

摘要号 4301/307,652，“自描述软件映象更新组件（Self-Describing Software Image Update Components）”。

技术领域

本发明一般涉及诸如具有嵌入式操作系统的那些设备等计算设备，尤其涉及配置计算设备的非易失存储。

背景技术

诸如个人数字助理、当代移动电话和手持式及袖珍计算机等移动计算设备正在变为重要且流行的用户工具。一般而言，它们变得足够小，使得它们极度方便，而消耗较少的电池功率，且在同时变得能够运行更强大的应用程序。

在制造这类设备的过程中，嵌入式操作系统映象通常被内建到每一设备的单块映象文件中，并储存在非易失存储中（如，NAND 或 NOR 闪存、硬盘等等）。作为结果，必须从构成操作系统的各种部件预配置单块映象文件。此外，更新这一

设备有时是必需或期望的，并需要对操作系统作出改变。

然而，当处理单块映象时，有许多缺点，包括为安装任一更新，需要替换整个映象（或可能其预定的子集），这需要大量的资源，包括临时存储和带宽。迄今为止，使用了单块替换用于更新这类设备，因为更新操作系统的单独组件是一项困难的任务，这是由于各种冲突和依赖性。此外，任一这样的组件化会导致另一问题，因为仍需要初始映象用于制造，但是迄今为止初始映象本质上只是传输到设备上的比特的单块组。需要一种将操作系统映象组件转换成适合用作初始映象的基于文件系统的制造映象，而仍被设计成方便设备的组件化更新的机制。

发明内容

简言之，本发明提供了一种一般针对创建包含分区和文件系统布局的单个制造映象文件的方法和系统，在构建时，将个别的操作系统组件包安装到该映象文件中。该过程以一种存储技术抽象方式完成，使得该系统和方法无需关注底层存储（如，闪存）的类型，和/或底层存储强加在映象布局上的任何要求。相反，在单独的步骤中，对结果映象文件进行后处理（post-process），包括定制该映象文件用于存在的实际类型的存储设备。

在一个实现中，在文件内创建各种类型的分区，每一分区对应于一个文件系统。将操作系统映象组件的集合（称为包）转换成基于文件系统的制造映象的分区。从该文件，可在制造过程中以使得映象更新技术稍后可利用设备上的同样的分区和文件模型的方式在设备上建立初始操作系统映象。

为将各种包转换成初始制造映象，创建单个制造映象文件，其中，按分区和文件系统布局排列映象。然后在需要时对该文件进行后处理，以在需要时添加用于在制造时安装其内容的元数据。

为构建最终被写入虚拟闪存中的该文件，创建各种分区。注意，总闪存的某些可由设备制造商为各种目的而保留，而让剩余的存储器可用，作为由操作系统组件和可任选用户存储使用的虚拟存储器。每一分区具有某一目的，并可被认为是其自己的文件系统。例如，可以有诸如用于更新加载器的二进制分区，以及用于内核（NK）分区的 RAMIMAGE/ROMIMAGE 分区。IMGFS 分区可包含操作系统文件，用户存储分区可被指定为用于用户数据存储。在文件中创建主引导记录以指定分区。可在文件中包括另外的数据。

在一个实现中，盘映象实用程序创建该文件，并且映象后处理器添加元数据来准备用于写入期望的存储媒质中的文件。盘映象实用程序负责取出操作系统包，并基于如何在存储中排列分区的存储器描述（存储器配置文件）以及包应当如何被映射到该分区结构的描述（分区映射文件），生成包含各种分区的输出文件，使每一包的内容储存在适当的分区中。此外，将可执行代码修正到适当的基于虚拟地址空间的地址。存储器配置文件为非易失存储和系统 RAM 都提供了操作系统运行时虚拟地址空间存储器映射。分区映射文件包含唯一地标识的包的列表，并用于将包映射到指定的分区。

后处理过程在输出文件上行动，以如特定存储技术所需的引入对分区和文件系统布局的任何修改。例如，可需要作出调整以处理不同的闪存部分管理闪存中的扇区级信息的不同方式。

当结合附图阅读以下详细描述时，可以清楚其它优点。

附图说明

图 1 是一般表示可结合本发明的计算机系统的框图；

图 2 所示是依照本发明的一个方面，用文件中用于维护作为映象写入嵌入式设备或其它媒质中的数据的相异分区构建的输出文件的框图；

图 3 所示是依照本发明的一个方面用于配置输出文件的各种组件的框图；

图 4 所示是依照本发明的一个方面配置输出文件的盘映象实用程序的总体流程的流程图；

图 5 所示是依照本发明的一个方面确认存储器配置文件的逻辑的流程图；

图 6 所示是依照本发明的一个方面负责核实包一分区映射文件并负责将指定的分区与包匹配的过程的流程图；

图 7 所示是依照本发明的一个方面处理闪存元件的过程的流程图；

图 8 所示是依照本发明的一个方面用于构造基于 ROM/RAM 的分区的过程的流程图；

图 9 所示是依照本发明的一个方面用于写系统分区的过程的流程图；以及

图 10A 和 10B 所示是依照本发明的一个方面，如由后处理过程所处理的在闪存中排列扇区和块的不同方法的框图。

具体实施方式

示例性操作环境

图 1 示出了一个这样的手持式计算设备 120 的功能组件，包括处理器 122、存储器 124、显示屏 126 和键盘 128（可以是物理或虚拟键盘，或表示两者）。可存在麦克风 129 以接收音频输入。存储器 124 一般包括易失存储器（如，RAM）和非易失存储器（如，ROM、PCMCIA 卡等等）。操作系统 130 驻留在存储器 124 中，并在处理器 122 上执行，如微软公司的 Windows®操作系统或另一操作系统。

一个或多个应用程序 132 被加载到存储器 124 中，并在操作系统 130 上运行。应用程序的示例包括电子邮件程序、调度程序、PIM（个人信息管理）程序、文字处理程序、电子表格程序、因特网浏览器程序等等。手持式个人计算机 120 也可包括加载到存储器 124 中的通知管理器 134，它在处理器 122 上执行。通知管理器 134 处理如来自应用程序 132 的通知请求。同样，如下所述，手持式个人计算机 120 包括适用于将手持式个人计算机 120 连接到网络（包括作出电话呼叫）的网络软件 136（如，硬件驱动程序等）和网络组件 138（如，无线电和天线）。

手持式个人计算机 120 具有电源 140，它被实现为一个或多个电池。电源 140 还可包括忽略内置电池或对其重新充电的外部电源，如 AC 适配器或加电对接托架。

图 1 所示的示例性手持式个人计算机 120 被示出为具有三种类型的外部通知机制：一个或多个发光二极管（LED）142 和音频生成器 144。这些设备可直接耦合至电源 140，使得当被激活时，即使手持式个人计算机处理器 122 或其它组件被关闭以保存电池能量时，它们也保留一段由通知机制指示的持续时间。LED 142 较佳地不限时间地持亮，直到用户采取行动。注意，音频生成器 144 的当代版本使用当今手持式个人计算机电池的太多能量，因此它被配置成当系统的剩余部分被关闭时，或者在激活后的一段确定的持续时间之后被关闭。

注意，尽管示出了基本手持式个人计算机，然而，为实现本发明的目的，实际上能够以可由程序使用的某一方式接收数据通信和处理数据的任何设备都是等效的。

在文件内创建文件系统

本发明一般针对安装和/或更新储存在基于微软 Windows®CE .NET 的便携式

设备等小型移动计算设备上的软件,这些设备包括在其中将初始软件或软件更新写入诸如闪存等嵌入式设备的非易失存储器的那些设备。尽管如此,本发明提供了在总体上计算的益处,并由此可应用到其它计算设备和其它类型的存储,包括各种类型的存储器和/或其它类型的存储媒质,如硬盘驱动器。为简化目的,术语“闪存”在后文参考设备的可更新存储来使用,尽管可以理解,任一存储机制都是等效的。此外,术语“映象”一般包括初始软件安装映象以及对该映象的随后的软件更新的概念,即使仅更新该映象的一部分。

包含可执行代码和数据两者的映象可被应用到存储。可执行代码在安装时被定制到嵌入式设备的虚拟地址空间。依照本发明的一个方面,一般而言,映象更新技术将操作系统映象分裂成可单独更新的可更新组件,而维持任何跨组件的依赖性。如可以理解的,以方便对设备的初始安装以及对其随后的更新的方式排列初始映象。

依照本发明的一个方面,提供了一种将操作系统映象组件的集合(称为包)转换成基于文件系统的制造映象的系统和方法。这通过分区和文件系统模型来完成,使得能够创建一输出文件。从该文件,可在制造过程中以使得映象更新技术可稍后利用设备上同样的分区和文件系统模型的方式在设备上建立初始操作系统映象。这方便了对个别组件、对整个分区或如需要时对整个映象的安全且故障保险的更新。

为将各种包转换成初始制造映象,提供了一种一般针对创建单个制造映象文件的方法和系统,该映象文件进而包含按分区和文件系统布局排列的映象。然后如需要对该文件进行后处理,来准备用于在构建时安装的内容。由此,个别的操作系统组件包被写入这一制造文件。整个文件构造过程以存储(如,闪存)技术抽象的方式完成,使得该系统和方法无需关注底层存储的类型,和/或底层存储在映象的布局上强加的要求。相反,在单独的步骤中,对结果映象文件进行后处理,包括定制映象文件用于所存在的实际类型的存储设备,使得随后可如期望的将初始映象应用到任何设备。

图2提供了设备存储(如,闪存)的一个实例,在该存储中,以由本发明的各个方面方便的方式布局制造映象,如后文所描述的。在图2中,总闪存202的某一些(如,总共64兆字节中的32兆字节)由设备制造商为各种目的而保留,(如,用于涉及设备无线电的代码)。剩余部分(如,32兆字节)是可作为由操作系统

和（可任选的）用户存储使用的虚拟存储器 204 的闪存。如后文所描述的，该文件将考虑保留的分段。

依照本发明的一个方面，如后文所描述的，在文件 206 中创建各种分区，文件 206 是要写入虚拟闪存 204 的文件。每一分区具有某一目的，并且（或可被认为是）包含其自己的文件系统。例如，在图 2 的示例中，有用于更新加载器的分区 209、内核（NK）分区 210、系统分区 211 和用户存储分区 212（如，为用户数据存储格式化的）。在文件中创建主引导记录（MBR）213 来指定分区，如后文所描述的。此外，如图 2 所示，为完整性目的，可如通过预先向其挂起数据在文件 206 中包括另外的数据，如预先存在的引导加载器 214（用于旧设备，尽管可任选地用于新设备），以及初始程序加载器（IPL）215。如可容易地理解的，这一另外的数据的某些或全部可被闪存到独立于其它分区的设备中，由此，可将文件 206 闪存到剩余的虚拟闪存中，然而，在制造文件 206 中包括这些比特消除了这一额外的制造步骤。注意，图 2 的分区并非按比例示出的，并且还应当注意，总存储器和/或可用存储器的量仅为示例。

如可以理解的，本发明提供了在单个文件中使用标准文件系统概念且以存储抽象的方式创建制造映象的能力。结果，可以直接的方式并在对核心过程只有极少/没有影响的情况下，令嵌入式或其它解决方案在变为可用时适用于任何新存储技术。例如，通过后处理，闪存文件系统映象可适用于变为硬盘映象。

另外，能够在构建时在个别的文件内创建文件系统意味着不需要在制造时环境中实现复杂的分区、格式化和其它文件系统逻辑。相反，当今用于将映象写入存储的标准方法（如，闪存群程序员、JTAG 或字节流复制）仍能够起作用，即使底层映象可能是复杂的分区以及稍后可由操作系统映象在构建时使用的基于文件系统的模式。

为实现本发明的各方面，在一个示例实现中（参考图 2 在上文一般描述的），提供了两个主要组件，即盘映象实用程序 230 和映象后处理器 232。盘映象实用程序 230 负责取出包装的操作系统组件的集合（或包）234，以及用户（如，制造商）希望分区/文件系统如何在存储中布局的描述（文件 236），和应当如何将包 234 映射到该分区结构的描述（文件 238，也称为包—分区映射文件和/或映象清单文件）。根据这一信息，盘映象实用程序 230 生成包含各种分区 209-212 的单个输出文件 206，使得每一包的内容储存在适当的分区中，并且使可执行代码被修正到适

当的基于虚拟地址空间的地址上。

一旦在这一输出文件状态中，后处理器 232 在这一输出文件上行动，以引入特定的存储技术所需的对分区和文件系统布局的任何修改。例如，可需要作出调整以处理不同的闪存部分管理闪存中的扇区级信息的不同方式，如后文参考后处理过程一般描述的。

在图 2 的示例中，由此，盘映象实用程序 230 由两个输入文件 236 和 238 驱动，即，存储器配置文件 236（如，memory.cfg.xml）和包一分区映射文件 238（如，*.sku.xml，其中，*表示有效文件名）。一般而言，分区映射文件 238 包含将每一特定操作系统组件（如，包）映射到存储器配置文件 236 中指示的特定分区的信息。

分区 209-212 可以是（并通常是）不同的类型。例如，在一个实现中，可以有 BINARY 分区（BINARY 分区具有简单地照原样复制到其上的比特），如压缩的更新加载器分区 209；一个或多个 RAMIMAGE 或 ROMIMAGE 分区，如 NK 分区 210；一个或多个 IMGFS（映象更新文件系统）分区，如系统分区 211；和/或 USERSTORE 分区，如 TFAT 或其它分区 212。尽管对分区和/或分区类型可以有任意的数量，然而一个当前的实现为保持该机制容易配置的目的将总数限制到四，如在图 2 中所示的更新加载器（BINARY）、NK 分区（RAMIMAGE 或 ROMIMAGE）、系统分区（IMGFS）和用户分区（USERSTORE）。注意，即使采用了这一任意的限制，也可以有重复的类型，如，采用四个分区的限制，三个二进制类型和一个 IMGFS 也是可接受的，并且还应当注意，其它类型的分区是可行的。

存储器配置文件 236 表示用于非易失存储和系统 RAM 的操作系统运行时虚拟地址空间存储器映射。数据表示实用程序的用户希望如何通过指定可在其上储存信息的一个或多个分区，并向储存在分区中的数据分配特征来利用指定的存储资源。

在一个实现中，存储器配置文件 228 是对照以下 XSD 确认的 XML 格式的文件：

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema targetNamespace="http://tempuri.org/MemoryCFG.xsd"
  elementFormDefault="qualified"
  xmlns="http://tempuri.org/MemoryCFG.xsd"
  xmlns:mstns="http://tempuri.org/MemoryCFG.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="MEMORYCFG">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="HARDWARE">
          <xs:complexType>
```

```

        <xs:sequence>
            <xs:element name="RAM">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="RAM_RESERVE"
minOccurs="0" maxOccurs="unbounded">
                            <xs:complexType>
                                <xs:sequence></xs:sequence>
                                <xs:attribute name="START"
type="xs:string" use="required" />
                                <xs:attribute name="LENGTH"
type="xs:string" use="required" />
                                <xs:attribute name="ID"
type="xs:string" use="required" />
                            </xs:complexType>
                        </xs:element>
                    </xs:sequence>
                    <xs:attribute name="START"
type="xs:string" use="required" />
                    <xs:attribute name="LENGTH"
type="xs:string" use="required" />
                </xs:complexType>
            </xs:element>
            <xs:element name="NOR" minOccurs="0"
maxOccurs="unbounded">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="NOR_RESERVE"
minOccurs="0" maxOccurs="unbounded">
                            <xs:complexType>
                                <xs:sequence />
                                <xs:attribute name="VASTART"
type="xs:string" use="required" />
                                <xs:attribute name="LENGTH"
type="xs:string" use="required" />
                                <xs:attribute name="ID"
type="xs:string" use="required" />
                            </xs:complexType>
                        </xs:element>
                    </xs:sequence>
                    <xs:attribute name="VASTART"
type="xs:string" use="required" />
                    <xs:attribute name="SECTORSIZE"
type="xs:string" use="required" />
                    <xs:attribute name="BLOCKSIZE"
type="xs:string" use="required" />
                    <xs:attribute name="LENGTH"
type="xs:string" use="required" />
                    <xs:attribute name="ID" type="xs:string"
use="required" />
                </xs:complexType>
            </xs:element>
            <xs:element name="NAND" minOccurs="0"
maxOccurs="unbounded">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="NAND_RESERVE"

```

```

maxOccurs="1" minOccurs="0">
    <xs:complexType>
        <xs:sequence />
        <xs:attribute name="LENGTH"
type="xs:string" use="required" />
        <xs:attribute name="ID"
type="xs:string" use="required" />
    </xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="SECTORSIZE"
type="xs:string" use="required" />
<xs:attribute name="BLOCKSIZE"
type="xs:string" use="required" />
<xs:attribute name="LENGTH"
type="xs:string" use="required" />
<xs:attribute name="ID" type="xs:string"
use="required" />
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="PARTITIONS">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="BINARY" minOccurs="0"
maxOccurs="unbounded">
                <xs:complexType>
                    <xs:sequence></xs:sequence>
                    <xs:attribute name="STORAGE_ID"
type="xs:string" use="required" />
                    <xs:attribute name="DATAFILE"
type="xs:string" use="required" />
                    <xs:attribute name="ID" type="xs:string"
use="required" />
                    <xs:attribute name="COMPRESS"
type="xs:boolean" />
                </xs:complexType>
            </xs:element>
            <xs:element name="RAMIMAGE" minOccurs="0"
maxOccurs="1">
                <xs:complexType>
                    <xs:sequence />
                    <xs:attribute name="STORAGE_ID"
type="xs:string" use="required" />
                    <xs:attribute name="FREE_SPACE_BUFFER"
type="xs:string" />
                    <xs:attribute name="FIXUP_ADDRESS"
type="xs:string" use="required" />
                    <xs:attribute name="COMPRESS"
type="xs:boolean" />
                    <xs:attribute name="LENGTH"
type="xs:string" />
                    <xs:attribute name="ID" type="xs:string"
use="required" />
                    <xs:attribute name="FSRAMPERCENT"

```

```

type="xs:string" use="required" />
                                <xs:attribute name="ROMFLAGS"
type="xs:string" use="required" />
                                </xs:complexType>
                                </xs:element>
                                <xs:element name="ROMIMAGE" minOccurs="0"
maxOccurs="1">
                                <xs:complexType>
                                <xs:sequence />
                                <xs:attribute name="FREE_SPACE_BUFFER"
type="xs:string" />
                                <xs:attribute name="STORAGE_ID"
type="xs:string" use="required" />
                                <xs:attribute name="LENGTH"
type="xs:string" />
                                <xs:attribute name="ID" type="xs:string"
use="required" />
                                <xs:attribute name="FSRAMPERCENT"
type="xs:string" use="required" />
                                <xs:attribute name="ROMFLAGS"
type="xs:string" use="required" />
                                </xs:complexType>
                                </xs:element>
                                <xs:element name="IMGFS" maxOccurs="1"
minOccurs="0">
                                <xs:complexType>
                                <xs:sequence />
                                <xs:attribute name="FREE_SPACE_BUFFER"
type="xs:string" />
                                <xs:attribute name="STORAGE_ID"
type="xs:string" use="required" />
                                <xs:attribute name="LENGTH"
type="xs:string" />
                                <xs:attribute name="ID" type="xs:string"
use="required" />
                                </xs:complexType>
                                </xs:element>
                                <xs:element name="USERSTORE" minOccurs="0"
maxOccurs="unbounded">
                                <xs:complexType>
                                <xs:sequence></xs:sequence>
                                <xs:attribute name="STORAGE_ID"
type="xs:string" use="required" />
                                <xs:attribute name="ID" type="xs:string"
use="required" />
                                <xs:attribute name="PART_TYPE"
type="xs:string" use="required" />
                                </xs:complexType>
                                </xs:element>
                                </xs:sequence>
                                </xs:complexType>
                                </xs:element>
                                </xs:sequence>
                                </xs:complexType>
                                </xs:element>
                                </xs:schema>

```

一个示例存储器配置描述如下：

```

<?xml version="1.0" encoding="utf-8" ?>
<MEMORYCFG xmlns="http://tempuri.org/MemoryCFG.xsd">
  <HARDWARE>
    <RAM START="0x840A0000" LENGTH="0x00F48000"></RAM>
    <NOR VASTART="0x80000000" SECTORSIZE="0x200" BLOCKSIZE="0x20000"
    LENGTH="0x3800000" ID="AR11FLASH">
      <NOR_RESERVE VASTART="0x80000000" LENGTH="0x80000"
      ID="GSM1"></NOR_RESERVE>
      <NOR_RESERVE VASTART="0x80080000" LENGTH="0x20000"
      ID="IPL"></NOR_RESERVE>
      <NOR_RESERVE VASTART="0x80100000" LENGTH="0x80000"
      ID="GSM2"></NOR_RESERVE>
      <NOR_RESERVE VASTART="0x80200000" LENGTH="0x80000"
      ID="GSM3"></NOR_RESERVE>
      <NOR_RESERVE VASTART="0x80300000" LENGTH="0x80000"
      ID="GSM4"></NOR_RESERVE>
      <NOR_RESERVE VASTART="0x80800000" LENGTH="0x1800000"
      ID="BIGRESRV"></NOR_RESERVE>
      <NOR_RESERVE VASTART="0x82000000" LENGTH="0x0040000"
      ID="EBOOT"></NOR_RESERVE>
    </NOR>
  </HARDWARE>
  <PARTITIONS>
    <BINARY STORAGE_ID="AR11FLASH" ID="ULDR" DATAFILE="uldr.nb0" />
    <ROMIMAGE STORAGE_ID="AR11FLASH" ID="NK" FSRAMPERCENT="0x40"
    ROMFLAGS="2" FREE_SPACE_BUFFER="0x10000" />
    <IMGFS STORAGE_ID="AR11FLASH" ID="OS" />
    <USERSTORE STORAGE_ID="AR11FLASH" ID="Storage" PART_TYPE="0x04" />
  </PARTITIONS>
</MEMORYCFG>

```

如可从本示例见到的，存储器配置文件的硬件段提供了 RAM 位置和所保留的每一闪存部分的完整描述，并也用于要由盘映象实用程序作为存储来管理的每一闪存部分。NOR 和 NAND 标签一般用于分别指线性（类似 RAM）或基于块的存储。向每一存储部分提供唯一的标识符/名字。分区段表示应当如何在基于分区的抽象中使用指定的存储部分。

例如，ROMIMAGE 意味着分区的内容应当被修正/重定位（如后文所描述的），以从存储部分实际执行，（原地执行，或 XIP），这是线性类型存储设备的特征。注意，如果使用 ROMIMAGE，则盘映象实用程序确保任何原地执行代码不被压缩，还确保模块的各个代码段在物理空间中是连续的，即，原地执行代码的这些代码段将不跨越保留的区域。RAMIMAGE 标签指示内容应当被修正/重定位，以用尽 RAM（假定加载器在适当的时刻将代码从存储移至 RAM）。除映象存储之外，可表示额外的分区类型；在上述示例中，将用户存储定义为用于数据存储的分区。

分区映射文件（或 SKU）238 包含唯一地标识的包的列表，并用于将包映射

到指定的分区中。分区映射文件是对照以下 XSD 确认的 XML 文件：

```
<?xml version="1.0" encoding="utf-8" ?>
- <xs:schema id="SKUFile" targetNamespace="http://tempuri.org/SKUFile.xsd"
  elementFormDefault="qualified" xmlns="http://tempuri.org/SKUFile.xsd"
  xmlns:mstns="http://tempuri.org/SKUFile.xsd"
```

```
xmlns:xs="http://www.w3.org/2001/XMLSchema">
- <xs:element name="SKU">
  - <xs:complexType>
    - <xs:sequence>
      - <xs:element name="PACKAGE_LIST" maxOccurs="unbounded"
minOccurs="0">
        - <xs:complexType>
          - <xs:sequence>
            - <xs:element name="PACKAGE_FILE" maxOccurs="unbounded"
minOccurs="0">
              - <xs:complexType mixed="true">
                <xs:sequence />
                <xs:attribute name="COMPRESSION" type="xs:string" />
              </xs:complexType>
            </xs:element>
          </xs:sequence>
          <xs:attribute name="PARTITION_ID" type="xs:string"
use="required" />
          <xs:attribute name="COMPRESSION" type="xs:string" />
        </xs:complexType>
      </xs:element>
      <xs:element name="RESERVE_PACKAGE" type="xs:string"
maxOccurs="unbounded" minOccurs="0" />
    </xs:sequence>
    <xs:attribute name="DEFAULT_COMPRESSION" type="xs:string" />
  </xs:complexType>
</xs:element>
</xs:schema>
```

一个示例分区映射（SKU）文件 238 如以下 XML 格式的数据：

```
<?xml version="1.0" encoding="utf-8" ?>
<SKU xmlns="http://tempuri.org/SKUFile.xsd">
  <PACKAGE_LIST PARTITION_ID="NK">
    <PACKAGE_FILE>oem</PACKAGE_FILE>
  </PACKAGE_LIST>
  <PACKAGE_LIST PARTITION_ID="OS">
    <PACKAGE_FILE>lang</PACKAGE_FILE>
  </PACKAGE_LIST>
</SKU>
```

在这一特定的示例中，一个示例包文件（“oem”）被映射到名为“NK”的分区。在以上存储器配置文件 236 中，它被定义为具有修正/重定位的代码以在闪存本身上原地执行的 ROMIMAGE 分区，这与 RAM 相反。如从本示例可见到的，指定的分区在硬盘扇区中记录的单个 NOR 闪存部分上存在，它在虚拟地址 0x8000.0000 处，并且长度为 0x380.0000 字节。如同样可从本示例中见到的，名为“lang”的包被映射到 IMGFS 分区（由称为“IMGFS”文件系统驱动程序管理的分区）中的被称为“OS”的第二分区，并也驻留在硬件扇区中指定的单个 NOR 部分上（但是不与前一 NK 分区重叠）。

基于两个输入文件 236 和 238 中提供的配置信息，盘映象实用程序 230 处理记录的包的内容。位于给定包中的每一可执行码通过修正或重定位过程被适当地定位到一唯一的虚拟地址空间范围。根据分区内容是意味着在 RAM 外执行还是在闪存中原地执行，盘映象实用程序 230 使用该地址空间信息以及关于在总体的虚拟地址空间上可将可执行文件的子段置于何处的已知的限制（如，基于 CPU 体系结构等），来处理每一可执行文件并将其内容定位到非重叠的虚拟地址空间范围。

如在图 3 中一般表示的，修正过程由称为 Romimage 的组件（如，链接到盘映象实用程序 230 或在其之内，或与其相关联）302 处理。在示例实现中，Romimage 302 是 DLL（动态链接库），它也在包安装过程中在设备上使用以在安装时将代码修正/重定位到设备上。在被构建时，使用运行时在设备上使用的相同的文件系统代码，将结果文件 206 储存在构建文件的台式机上所储存的文件内的适当的分区中。注意，该代码对设备以及主机构建系统都是可构建的，以将不兼容问题最小化。

在构建文件之前，盘映象实用程序 230 请求某一大小的文件的创建，如大小为 64 兆字节。在 FSDMGR 304 创建了文件之后，盘映象实用程序 230 处理存储器配置文件 236，以令 FSDMGR 304 划分出保留的分段，使得它们保持完整。剩余的存储器，如 32MB 现在通过 FSDMGR 304 可用。在这一时刻，该文件也准备好用需要的分区来构建。

如可从以上示例存储器配置文件中所见到的，期望一 BINARY 分区。更具体地，图 3 示出了盘映象实用工具 230 如何与 romimage.dll 功能以及文件系统设备驱动程序管理器 304（FSDMGR.dll）一起工作来构建文件 206 的示例。为在文件中创建分区，盘映象实用程序 230 结合 FSDMGR 304 一起工作，以创建一主引导记录（因为还没有该记录）。将数据添加到主引导记录以指示之后为 BINARY 分区，

即，数据创建该分区。在这一实现中，不指定大小，由此，对于现在，剩余的闪存的整体大小用于该 BINARY 分区。

在本示例中，盘映象实用程序 230 将通过 Romimage 302 请求 FSDMGR 304 将更新加载器写入 BINARY 分区中，由图 3 中通过用带圈的一（1）标签的箭头所一般表示的。如上述名为“以故障保险方式向非易失存储应用自定义软件映象更新”的相关专利申请中所描述的，更新加载器本质上是操作系统代码的一个特殊分段，当设备被更新时，设备启动到该分段（而非正常的操作系统代码）。由于从盘映象实用程序的观点来看，更新加载器是比特的二进制大数据块（binary blob），因此安装该分区，并且 FSDMGR 304 使用 rawFS dll 308 将这些二进制比特写入 BINARY 分区 209 中（写入主引导记录 213 之后的空间中）。

一旦被写入，通过调用 FSDMGR 304 获取实际写入的数据量。在自动确定大小的操作中，下一分区的起点的新偏移基于该实际大小，由此，更新加载器分区 209 本质上仅消耗文件中它所需要的空间量。

盘映象实用程序 230 再次以类似的方式调用以在文件中创建 NK 分区（即，通过将数据写入主引导记录 213 中），然后通过以将数据写入 NK 分区的请求调用 romimage.dll 写 NK 分区。注意，romimage.dll 用于构建 NK 分区所需的从盘映象实用程序 230 发送的参数包括要构建的文件列表以及用于分配闪存空间和虚拟地址空间的分配器，如后文所描述的。romimage.dll 302 将该数据修正到一组比特中，然后将它们提供给 FSDMGR 304，FSDMGR 304 然后通过 rawFS dll 308 写 NK 分区，如在图 3 中由用带圈的数字二（2）标签的箭头所表示的。注意，该物理修正以及虚拟修正可为任何原地执行代码所需。

再次，不指定大小，因此整个剩余空间用于该 NK 分区，直到被调整大小。然而，在本示例中，并非基于对 NK 分区所写的确切大小将偏移从文件的末端移回，而是可将 NK 比特之后的某些额外空间量（缓冲区）留在 NK 分区中，如由图 3 中的阴影区所表示的。这一空闲空间缓冲区在存储器 XML 文件中指定，并本质上将偏移稍微移至实际所写入的字节量之前，并由此令对 NK 分区的未来更新能够消耗比当前 NK 分区所消耗的更多的空间，即，允许随后的更新的增长。

如在图 3 中由用带圈的数字三（3）标签的箭头所表示的，通过 romimage.dll，下一步创建并写 IMGFS 分区（再次延伸到文件的末尾）。注意，在这一时刻，这是 IMGFS 分区，而非二进制分区，如由 FSDMGR 304 所识别的。由此，使用 imgfs.dll

308 来写入系统分区，因为这是在运行时在设备上处理该特定分区的驱动程序。此外，注意，通过 `imgfs.dll` 308 抽象任何物理修正，由此，对于系统分区，`romimage` 过程 302 仅如指定的执行虚拟地址修正。然后如上所述地执行自动调整大小，以将下一分区的偏移移至写入的数据量（加上任一期望等空闲空间缓冲区，尽管在当前的示例中没有空闲空间缓冲区）。

下一分区是 `USERSTORE`，它可以是任何类型的分区，在本示例中为类型 4 的分区。`USERSTORE` 分区通过写入主引导记录中来创建，并被扩展到文件的末端。然而，注意，在这一时刻不写入任何数据，由此，该分区未被安装。当用户稍后希望访问该分区时，`FSDMGR` 将通过适当的驱动程序来完成这一过程，如，如果分区对应于该特定的文件系统格式，则通过 `TFAT.dll` 310 来完成。由此，不需要 `TFAT.dll` 310 来构建初始文件，但是在图 3 中作为可用于在运行时访问用户分区数据的驱动程序的示例示出。

转向本发明的操作的解释，作为概述，盘映象实用程序 230 是采用平台存储器配置文件 236（如 `memory.cfg.xml`）和映象清单文件 238（如，`platform.sku.xml`）作为输入，并输出表示设备的完整 ROM 映象的数据文件 206 的台式机实用程序。由于盘映象实用程序 230 负责重定位模块数据，一个设计目标是能够与设备侧更新应用程序共享代码。

除其它操作之外，盘映象实用程序 230 分析定义设备的存储器布局的存储器配置文件 236、任何保留的区域、以及包含预构建 `BIN/NB0` 文件（其中，`NB0` 文件是所有 `.bin` 文件如它们在 ROM 中所出现的布局）的一个或多个闪存分区的位置和大小。盘映象实用程序 230 也分析声明基于包的分区及其内容的映象清单文件 238。

在图 3 中一般表示的实现中，盘映象实用程序 230 能够构造各种类型的分区，包括 `BINARY`、`RAMIMAGE`、`ROMIMAGE`、`IMGFS` 和 `USERSTORE` 分区。`BINARY` 分区包括包含任意数据，但不包含包的预构建 `NB0` 文件。如由名字所暗示的，`RAMIMAGE` 分区被修正到 `RAM`，而 `ROMIMAGE` 分区被修正以从储存它们的 `NOR` 部分原地执行。`IMGFS` 分区包括由盘映象实用程序 230 支持的分区类型，并可简单地通过作出对 `IMGFS.dll` 308 的合适版本的调用（通过 `FSDMGR` 304）来构造。`USERSTORE` 分区可以是任何自定义分区类型，并且当前用作主引导记录 213 中用于在启动时创建的分区的占位符。

下表概述了盘映象实用程序 230 的输入文件和输出文件：

术语	描述
CFG 文件	输入到盘映象实用程序的存储器配置文件，它指定了硬件和分区信息。此处也称为 Memory.CFG.XML，它是盘映象实用程序所需的输入。
SKU 文件	输入到盘映象实用程序的分区映射文件/映象清单文件，它指定了基于包的分区内容。
RAMIMAGE	被修正以从 RAM 运行的二进制分区。可包含包。
ROMIMAGE	被修正以从闪存运行的二进制分区。可包含包。
IMGFS	盘映象实用程序支持的一种分区类型。可包含包，如系统分区包。
USERSTORE	用户定义的分区类型。其期望的使用是创建 FAT 或扩展分区的 MBR 条目，但是可以是任一分区类型。
BINARY	盘映象实用程序支持的一种分区类型。可包含任意数据。不能包含包。

盘映象实用程序 230 负责在基于包的分区的任一个中重定位模块。为实现这一功能，盘映象实用程序 230 采用一虚拟地址 (VA) 分配器。盘映象实用程序 230 也能够为在设备的每一闪存部分中包含分区的每一闪存部分输出 BIN/NB0 文件（后文描述），以及主引导记录 (MBR)。输出文件可通过使用 MSPart 台式机组件来创建。此外，盘映象实用程序 230 中的大多数代码（如，VA 分配器、IMGFS 交互、模块重定位）也可用于设备侧更新应用程序，由此，盘映象实用程序 230 考虑到设备限制和代码可移植性。

图 4 表示盘映象实用程序 230 的整体流程。注意，盘映象实用程序 230 可包括台式机组件，也可包括用于执行各种操作的单独组件。例如，盘映象实用程序可通过命令行来运行，由此，为如块 400 所示地处理命令行自变量，盘映象实用程序 230 可包括一命令行处理器。单独的 CFG 文件分析器可处理存储器配置文件（块 402），单独的 sku/包分析器可如块 404 所示地处理分区映象 (SKU) 文件。组件及其功能在图 4 中由 BINARY 分区构建器/过程（块 406）、RAMIMAGE/ROMIMAGE 分区构建器/过程（块 408）、IMGFS 分区构建器/过程（块 412）和后处理器（块 414）表示。盘映象实用程序 230 使用的后处理器和数据结构在后文单独的小节中描述。

如如上所述，盘映象实用程序 230（如，dskimage.exe）可通过命令行自变量来调用，如命令行：

```
dskimage CFGfile SKUfile
```

如应当容易地清楚的，CFGfile 参数是到存储器配置文件 236 的路径，如上所述，该输入文件详细描述当前平台的 RAM 和闪存布局，并定义分区。SKUfile 参数是到分区映射/映象清单文件 238 的路径，如上所述，该文件列出了包的集合并将它们分配到分区。在一个实现中，盘映象实用程序 230 在分析输入文件时不考虑文件扩展名，但是在被调用时（如，通过脚本）期望具有以下名字的输入：

```
CFGFile=Memory.cfg.xml
```

```
SKUFile=%_TGTPLAT%.sku.xml
```

当前，命令行处理器仅检查 CFGfile 和 SKUfile 命令行自变量的存在，并将它们传递到其各自的（如，XSD）确认器。如果用相对路径指定了自变量，则盘映象实用程序 230 相对于从其调用盘映象实用程序 230 的目录查找文件。如果用绝对路径指定了自变量，则盘映象实用程序 230 使用绝对路径查找该文件。在盘映象实用程序 230 中，path1 是 Environment.CurrentDirectory，path2 是 CFGfile 或 SKUfile。如果自变量不存在，则命令行处理器采取适当的行动（如，在 C#中抛出异常、打印使用消息和退出）。

memory.cfg.xml 分析器/过程被设计成给予制造商描述硬件和向闪存部分分配分区的很大的灵活性。在一个当前的实现中，存储器配置分析器/过程分析器（图 4 的块 402）在 memory.cfg.xml 文件上采用两级确认，如图 5 中一般表示的。首先，通过 C# XMLValidatingReader 类，核实 memory.cfg.xml 文件正确句法；如由步骤 500 所示的，XMLValidatingReader 确保 momory.cfg.xml 包含有效的 XML 和需要的标签，否则步骤 502 分支到输出错误条件。第二级确认由盘映象实用程序 230 执行。在第二级确认中，创建 RAM、闪存部分和分区的内部表示。使用 XML 串行化来创建内部表示，然后在该内部结构上执行确认（如在图 5 中由步骤 504 和 506 所示的），以检测无效的配置，如将分区映射到不存在的闪存部分。

在一个实现中，（如，基于 Windows® CE 的实现），为使硬件配置有效，文件中的 RAM START 属性需要引用有效的高速缓存内核虚拟地址（0x80000000-0x9FFFFFFF）；START+LENGTH 也需要有效。任意 RAM_RESERVE 分段需要在由父 RAM 元素的 START 和 LENGTH 属性所指定的内核虚拟地址范围内开始和

结束。RAM_RESERVE 元素无法重叠（START 到 START+LENGTH 应当对每一 RAM_RESERVE 是唯一的），并需要具有唯一的、非空的 ID 串。

在本示例实现中，可实施各种规则，包括 NOR/NAND（被总称为 FLASH）需要具有唯一的“ID”属性，并不能被命名为“RAM”。FLASH RESERVE 元素需要唯一的“ID”，并不能具有长于 8 个字符的名字。FLASH 元素 LENGTH 属性需要被 BLOCKSIZE 均匀可分，BLOCKSIZE 进而需要被 SECTORSIZE 均匀可分。对于 NOR 元素，VASTART 需要是块对齐的。FLASH RESERVE 元素 LENGTH 属性需要被父 FLASH BLOCKSIZE 均匀可分。对于 NOR_RESERVE 元素，VASTART 是块对齐的，NOR VASTART 和 VASTART+LENGTH 应当不与 RAM 或任意其它 NOR 元素重叠，并且应当是有效的高速缓存内核虚拟地址。

此外，为有效，NOR_RESERVE 元素需要在由父 NOR 元素的 VASTART 和 LENGTH 属性指定的高速缓存内核虚拟地址范围内开始和结束。可使用 Allocator（分配器）类分层结构来核实 HARDWARE 条件，如后文所描述的。为有效的高速缓存内核地址范围创建一全局分配器，将其用于检测 RAM 和 NOR 标签的有效地址，以及 RAM 和 NOR 部分之间的任何冲突。类似地，对每一 RAM 和 FLASH 部分创建分配器，以检测有效的 RESERVE 区域。分配器可储存在关联的 RAM/FLASH 对象中，用于容易的检索。

分区数据也储存在 MemoryCFG 分层结构中。PARTITONS 确认的规则包括分区具有唯一 ID 属性、PARTITION 的 STORAGE_ID 属性需要匹配 FLASH 部分的 ID 属性、以及 ROMIMAGE 分区的 STORAGE_ID 属性不能引用 NAND 闪存。对于 RAMIMAGE/BINARY，COMPRESS 属性是布尔类型，并由此只能是以下之一：{0, 1, true, false}。

也确认 RAMIMAGE/ROMIMAGE/IMGFS（总称为 PACKAGE 分区），并且在当前的实现中，只能有一个 RAMIMAGE/ROMIMAGE 分区。如果指定了两个，则确认失败。FSRAMPERCENT 和 ROMFLAGS 属性对应于 RAMIMAGE/ROMIMAG 分区的目录中的字段（如，在基于 Windows® CE 的实现中）。RAMIMAGE 分区的 FIXUP_ADDRESS 涉及该分区应当在 RAM 中何处开始。该属性需要指向 RAM 中的有效位置，其后紧跟至少空闲的 0x1000 字节。当前，每一 FLASH 部分上不能有一个以上 USERSTORE 分区。

如上所述，映象清单（包一分区映射或 SKU）文件 238 包含由分区组织的包

的列表。XML 模式（上文示出）相对直接。映象清单文件分析器/过程负责核实 SKU 文件包含有效 XML，并负责将 PACKAGE_LIST 中指定的分区与 memory.cfg.xml 中的 PACKAGE 分区 ID 进行匹配。PACKAGE_LIST 标签的 PARTITION_ID 属性需要与 PACKAGE 分区元素的 ID 属性相匹配；如果无法找到匹配，则 SKU 分析器将抛出异常。图 6 的步骤 600、602 和 604 概述了这些各种操作。

每一 PACKAGE_LIST 标签具有需要的 PARTITION_ID 属性。将包与分区进行匹配包括取出 PARTITION_ID 并使用它作为对 C# 散列表的查找。如果找到匹配，则将子 PACKAGE_FILE 标签转化成数组列表，并与 PACKAGE 分区的现有包列表合并。提取包可以以下方式完成：

```
///对于每一 PackagePartition
///{
///    创建目录.\DskImage\PartitionID
///    对于 PackagePartition 中的每一 Package
///    {
///        打开文件.\Packages\PackageName
///        如果未找到文件，则错误
///        将内容提取到.\DskImage\PartitionID
///        如果提取失败，则错误
///    }
///}
```

注意，盘映象实用程序 230 检查 _FLATRELEASEDIR 环境变量的存在，并且如果找到，则使用它作为当前的工作目录。这与传统的构建系统的行为相一致，它允许用户从任一目录执行“makeimg（制造映象）”，而仅在 _FLATRELEASEDIR 中处理文件。

分区由 MSPart 的台式机（构建系统）版本创建并管理，（如在传统的 Windows® CE 构建系统中）。MSPart 接口允许盘映象实用程序 230 从分区的实际构造中抽象关于底层硬件的细节。对由 MSPart 创建的 NBO 文件的任何硬件专用调整在后处理过程中完成。MSPart 可由盘映象实用程序 230 工具的剩余部分使用，如果是这样，则在创建任何分区之前，对每一 FLASH 部分创建一 MSPart 卷，并且划分出任何

RESERVE 分段, 如后文所描述的。盘映象实用程序 230 维护一单独的类 `fsdmgr.cs`, 它包装所需要的未管理的功能。该过程维护闪存部分的一全局散列表, 使得其每一个可由名字访问并通过其重复。上述过程通过在每一 FLASH 部分的 FLASHREXERVE 元素上重复并调用 `fsdmgr.dll` 来完成, 如图 7 中一般表示的。RESERVE 区域在实际被创建时用数据来填充。为此, `FSDMgr` 类将来自盘映象的数据汇集到 `fsdmgr_nt.dll` 以填充 RESERVE 区域。

构建 NBO 的第一个步骤是向 `MSPart` 安装的卷添加 BINARY 分区。首先处理 BINARY 分区, 因为它们是完全自包含的 (即, 它们不需要修正, 也不需要 NOR 上的 `SectorData` 元数据)。如上文一般描述的, 构建 BINARY 分区相对较直接; 该过程重复通过 `BinaryPartitions` (二进制分区) 数组, 并调用其 `Create()` 方法。由于每一分区对象指向其父闪存部分, 检索正确的卷句柄也是非常直接的。

在一个实现中, `PartitionInfo` (分区信息) 类 (`MemoryCFG` 分层结构的一部分) 包含名为 `CreateAllPartitions` (创建所有分区) 的函数, 它通过在 `BinaryPartition` 对象上重复, 并调用其 `Create()` 方法来开始。`BinaryPartition.Create()` 函数在指定的闪存块上创建并安装新的分区、打开 `memory.cfg.xml` 的 `DATAFILE` 属性中指定的文件、并调用 `FSDMgr` 以将数据写入该分区。这一特征的伪代码描述如下 (使用某些 `MSPart` API) :

```
///创建最大可用大小的分区 (如果创建失败, 错误)
///打开 DATAFILE (如果未找到, 错误)
///将数据从 DATAFILE 复制到分区 (如果用尽空间, 错误)
///关闭分区和 DATAFILE
///查询使用的空间并适当地调整分区的大小
```

如上所述, 盘映象实用程序提供了用于分别创建 ROMIMAGE 和 RAMIMAGE 类型的分区—从 ROM 和 RAM 原地执行的分区的 “romimage”。这些分区通常包含构成 IMGFS 文件系统所需的系统组件的最小组: `nk`、`coredll`、`filesystem`、`fsdmgr`、`msspart`、`imgfs` 和任何需要的块驱动程序。

ROMIMAGE/RAMIMAGE 过程一般可被分裂成各种步骤 (其中八个在当前实现中列出, 如也由图 8 的步骤 801-808 所示的) :

构建并排序模块和文件的列表 (步骤 801) ;

向所有模块分配虚拟地址空间（步骤 802）；

执行模块修正（步骤 803）；

压缩所有数据段和文件（步骤 804）；

执行物理布局（步骤 805）；

分配复制分段（步骤 806）；

执行内核模块修正（步骤 807）；以及

输出实际分区（步骤 808）。

上述每一步骤可通过对 romimage.dll 作出调用来执行，允许容易地访问各种（如，C 和 C#）应用程序的分配和重定位功能。盘映象实用程序可通过称为 ROMImage.cs 的包装类与 romimage.dll 接口。

romimage.dll 包含用于创建并管理多个分配器的 Allocator 类分层结构和功能、用于创建并管理文件列表的 File（文件）类和功能、以及用于执行 ROMIMAGE/RAMIMAGE 分区构建过程的步骤的函数。

Allocator 类分层结构用于管理 RAM 和 FLASH 中的可用物理空间。它展现作出任意分配和固定保留的功能。Allocator 类（和子 AllocNode）定义如下：

```
class AllocNode
{
    public:
        AllocNode(DWORD s, DWORD l) { dwStart = s; dwLength = l;
dwFlags = 0;}
        AllocNode(DWORD s, DWORD l, DWORD f) { dwStart = s; dwLength =
l; dwFlags = f;}
        HANDLE GetHandle() { return (HANDLE) this; }
        DWORD GetStart() { return dwStart; }
        DWORD GetLength() { return dwLength; }
        DWORD GetEnd() { return dwStart + dwLength; }
        DWORD GetFlags() { return dwFlags; }
        BOOL IsReserveAlloc() { return (dwFlags & RESERVE_ALLOC); }

    private:
        DWORD dwStart;
        DWORD dwLength;
        DWORD dwFlags;
};

class Allocator
{
    public:
        Allocator(DWORD s, DWORD l);
        HANDLE GetHandle() { return (HANDLE) this; }
        DWORD GetTotalFreeSpace();
        HRESULT DoReserve(DWORD dwReserveStart, DWORD dwReserveLength,
DWORD dwFlags,
                        list<AllocNode>::iterator *pAllocNode, BOOL
```

```

fSimulate);
    HRESULT Allocate(DWORD dwLength, DWORD dwAlignment, DWORD
dwFlags,
                    list<AllocNode>::iterator *pAllocNode, BOOL
fSimulate);
    HRESULT SetWindowSize(DWORD s, DWORD l, BOOL fIgnoreReserves);
    HRESULT GetMaxFreeNode(list<AllocNode>::iterator *pFreeNode);
    BOOL GetFirstWindowReserve(list<AllocNode>::iterator
*pAllocNode);
    BOOL GetNextWindowReserve(list<AllocNode>::iterator pCurNode,
list<AllocNode>::iterator *pAllocNode);
    void PrintFreeList();

    DWORD GetStart() { return dwStart; }
    DWORD GetLength() { return dwLength; }
    DWORD GetEnd() { return dwStart + dwLength; }
    DWORD GetWindowStart() { return dwWindowStart; }
    DWORD GetWindowLength() { return dwWindowLength; }
    DWORD GetWindowEnd() { return dwWindowStart + dwWindowLength; }
    DWORD GetFirstWindowAlloc();
    DWORD GetLastWindowAlloc();

#ifdef _DEBUG
    list<AllocNode> &GetFreeList() { return FreeList; }
    list<AllocNode> &GetAllocList() { return AllocList; }
#endif // _DEBUG

private:
    DWORD dwStart;
    DWORD dwLength;
    DWORD dwWindowStart;
    DWORD dwWindowLength;
    list<AllocNode> FreeList;
    list<AllocNode> AllocList;

    // private functions
    HRESULT TopDownFirstFit(DWORD dwLength, DWORD dwAlignment,
PDWORD pdwAllocStart);
    HRESULT BottomUpFirstFit(DWORD dwLength, DWORD dwAlignment,
PDWORD pdwAllocStart);
    HRESULT BottomUpBestFit(DWORD dwLength, DWORD dwAlignment,
PDWORD pdwAllocStart);
};

```

romimage.dll 导出用于操纵分配器的以下功能:

```

HRESULT CreateAllocator(DWORD dwStart, DWORD dwLength, PHANDLE
phAlloc);
HRESULT DestroyAllocator(HANDLE hAllocator);
HRESULT CanReserveRange(HANDLE hAllocator, DWORD dwStart, DWORD
dwLength);
HRESULT ReserveRange(HANDLE hAllocator, DWORD dwStart, DWORD

```

```
dwLength, PHANDLE pNode);
HRESULT SetAllocatorWindow(HANDLE hAllocator, DWORD dwStart,
DWORD dwLength, BOOL fIgnoreReserves);
HRESULT Allocate(HANDLE hAllocator, DWORD dwLength, DWORD
dwAlignment, DWORD dwFlags, PHANDLE pNode);
```

Allocate 函数的有效标志包括 BOTTOMUP_ALLOC 和 TOPDOWN_ALLOC。如名字所暗示的,这些都是分别查找从当前分配窗的底部和顶部开始的空闲空间的首次满足 (first-fit) 算法。它们需要线性搜索 (并由此可在请求大量的分配时产生性能瓶颈)。

File 类分层结构用于储存关于文件或模块的所有元数据, 并定义如下:

```
class File
{
public:
    File(wchar_t *FileName, DWORD Attributes);
    ~File();
    HANDLE GetHandle() { return (HANDLE) this; }
    DWORD GetLength() { return dwDataLength; }
    DWORD GetFlags() { return dwFlags; }
    wstring GetFileName() { return szFileName; }
    FILESEntry GetFilesEntry() { return filesEntry; }
    DWORD GetFileAttributes();
    virtual HRESULT Load(wchar_t *szFilePath);
    virtual HRESULT DoLayout(Allocator &Alloc, LayoutType ft);
    virtual HRESULT WriteToIMGFS(HVOL hVolume);
    virtual HRESULT WriteToNK(PBYTE pbBuffer, DWORD cbSize, DWORD
dwOffset, PBYTE pbEntryLoc);
    virtual HRESULT DoCompression();

#ifdef _DEBUG
    virtual HRESULT DumpFile();
#endif

protected:
    wstring szFileName;
    HANDLE hFileNameAlloc;
    HANDLE hPhysAlloc;
```

```

    DWORD dwFlags;
    DWORD dwDataLength;
    BYTE *data;

private:
    CMappedFile *pMappedFile;
    FILESEntry filesEntry;
};

class Section
{
public:
    Section() { hPhysAlloc = hRAMAlloc = INVALID_HANDLE_VALUE; }
    HANDLE GetHandle() { return (HANDLE) this; }
    BOOL IsDataSection();
    BOOL IsCodeSection();
    BOOL IsKDATA();

    // these can be made private...
    o32_rom *o32Header;
    HANDLE hPhysAlloc;
    HANDLE hRAMAlloc;
    DWORD dwDataLength;
    BYTE *data;
};

class FileList;

class Module : public File
{
public:
    Module(wchar_t *FileName, DWORD Attributes);
    ~Module();

    // virtual functions
    virtual HRESULT Load(wchar_t *szFilePath);
    virtual HRESULT DoLayout(Allocator &Alloc, LayoutType ft);
    virtual HRESULT WriteToIMGFS(HVOL hVolume);
    virtual HRESULT WriteToNK(PBYTE pbBuffer, DWORD cbSize, DWORD
dwOffset, PBYTE pbEntryLoc);
    virtual HRESULT DoCompression();

    // useful query functions
    BOOL IsKernel() { return (szFileName == KernelName); }
    BOOL IsKernelModule() { return (dwFlags & TYPEFLAG_K) ? TRUE :
FALSE; }
    BOOL IsSlot0Only() { return fIsSlot0Only; }
    BOOL IsDLL() { return (e32Header->e32_imageflags &
IMAGE_FILE_DLL) ? TRUE : FALSE; }

    // get/set
    DWORD GetCopySectionCount();
    TOCentry GetTocEntry() { return tocEntry; }
    void SetSlot0Only(BOOL value) { fIsSlot0Only = value; }
    void SetCrellocPtr(Section *value) { crelocPtr = value; }
    e32_rom *GetE32Ptr() { return e32Header; }

```

```

// API functions
HRESULT DoSplitAlloc(Allocator &Slot0Alloc, Allocator &Slot1Alloc);
HRESULT DoSlot0OnlyAlloc(Allocator &Slot0Alloc);
HRESULT DoKernelVAAAlloc(Allocator &RAMAlloc, list<COPYentry>
&CopySections);
HRESULT DoKernelFixups(BOOL fReverse, FileList &ParentList);
HRESULT DoFixups(BOOL fReverse, FileList &ParentList);
HRESULT DoFixups(BOOL fReverse, char *szRelFile);
HRESULT FixupBlob(BYTE *data, DWORD dwDataLen, DWORD dwStartRVA,
BYTE srcSection, BOOL fReverse, FileList &ParentList);
HRESULT AddSection(PHANDLE phSection);

private:
    e32_rom *e32Header;
    DWORD dwOriginalVBase;
    Section *crelocPtr;
    list<Section> o32Headers;
    BOOL fIsSlot0Only;
    TOcentry tocEntry;

    // allocator handles
    HANDLE hSlot0Alloc;
    HANDLE hSlot1Alloc;
    HANDLE hE32Alloc;
    HANDLE hO32BlockAlloc;

    // private functions
    HRESULT ApplyFixup(WORD wRelocType, PDWORD pdwData, DWORD dwOffset,
WORD wRelocLowAddr);
    HRESULT DoCodeLayout(Allocator &Alloc);
    HRESULT MungerRVAs();
    HRESULT UnMungerRVAs();
};

```

在构建文件的台式机中，每一文件的内容由 File 和 Section 类的数据成员作出存储器映射和指向它。定义以下数据结构用于操作 File 类分层结构：

```

class FileList
{
public:
    FileList();
    HANDLE GetHandle() { return (HANDLE) this; }
    HRESULT AddFile(wchar_t *szFileName,
        DWORD dwFlags,

```

```

        wchar_t *szPathToFile,
        BOOL fLoadData,
        PHANDLE phFile);
HRESULT AddModule(wchar_t *szFileName,
        DWORD dwFlags,
        wchar_t *szPathToFile,
        BOOL fLoadData,
        PHANDLE phFile);
HRESULT DoVAAlloc(Allocator &Slot0Alloc, Allocator &Slot1Alloc);
HRESULT DoVAFixups(BOOL fReverse);
HRESULT DoKernelVAFixups(BOOL fReverse);
HRESULT DoPhysicalLayout(Allocator &Alloc);
HRESULT DoCopySectionAlloc(Allocator &Alloc);
HRESULT DoWriteNKPartition(HVOL hVolume, Allocator &Alloc);
HRESULT DoWriteIMGFSPartition(HVOL hVolume);
HRESULT DoParseMiscNKInfo(MiscNKInfo *pData);
HRESULT DoCompression();

BOOL HasKernelModules() { return (KernelModules.size() > 0); }
BOOL HasKernel() { return (KernelModules.begin()->GetFileName() ==
KernelName); }

// debug spew
void PrintKernelModules();

// get/set
DWORD GetTOCLoc() { return dwTocLoc; }
void SetTOCExtensions(DWORD dwRomExtLoc) { pToc.pExtensions =
(PVOID) dwRomExtLoc; }

private:
    list<File> Files;
    list<Module> KernelModules;
    list<Module> SplitModules;
    list<Module> Slot0Modules;
    list<COPYentry> CopySections;
    HANDLE hCopySectionsAlloc;
    ROMHDR pToc;
    DWORD dwTocLoc;
    HANDLE hTocAlloc;
    DWORD dwSlot1Base;
    BOOL fCompressPartition;
    HANDLE hPrologAlloc;

    // private functions
    HRESULT DoLayout(Allocator &Alloc, LayoutType lt);
    HRESULT WriteROMInfo(HVOL hVolume);
    HRESULT WriteProlog(PBYTE pBuffer, DWORD cbSize);
    HRESULT WriteCopySections(PBYTE pBuffer, DWORD cbSize);
    DWORD GetCopySectionCount();
};

extern list<FileList> FileLists;

```

提供以下函数用于构建 RAMIMAGE/ROMIMAGE 分区：

```

HRESULT BuildNKPartition(HANDLE hFileList,
        HANDLE hVolume,
        HANDLE hSlot0Alloc,

```

```

HANDLE hSlot1Alloc,
HANDLE hPhysAlloc,
DWORD dwReserved);

```

其中：

hFileList 是到 FileList 对象的句柄--FileList 对象的创建在下一节中讨论；

hVolume 是到 MSPart 卷的句柄，由 DskImage 在分区创建过程中创建；

hSlot0Alloc 是到起点为 0x600000，长度为 0x1A00000 的分配器的句柄；

hSlot1Alloc 是到起点为 0x2100000，长度为 0x1F00000 的分配器的句柄；

hPhysAlloc 是物理分配器（对 ROMIMAGE 分区为闪存部分，对 RAMIMAGE 分区为 RAM）；

hRAMAlloc 是 RAM 分配器，它对应于 memory.cfg.xml 中的 RAM 元素；以及

dwReserved 是到 MiscNKInfo 结构的指针，定义如下：

```

typedef struct _MiscNKInfo
{
    USHORT cbSize;
    USHORT usCPUType;
    DWORD dwROMFlags;
    DWORD dwFSRAMPercent;
    BOOL fCompressPartition;
} MiscNKInfo;

```

为构建并排序模块和文件的列表，（图 8 的步骤 801），romimage.dll 管理文件和模块的多个链接的列表。为构建列表，romiimage.dll 提供以下函数：

```

HRESULT CreateFileList(PHANDLE phFileList);
HRESULT DestroyFileList(HANDLE hFileList);
HRESULT AddFileToList(Handle hFileList,
    wchar_t *szFileName,
    DWORD dwFlags,
    wchar_t *szPathToFile,
    BOOL fLoadData,

```

```
PHANDLE phFile);
```

也可以有将 File 对象从一个文件对象移动到另一个的 API。这一函数可为以下形式：

```
HRESULT SpliceFile(HANDLE hSrcList,  
HANDLE hDstList,  
HANDLE hFile);
```

为向模块分配虚拟地址空间，（图 8 的步骤 802），插槽 0/1 虚拟地址分配需要文件列表、用于插槽 0 的分配器和用于插槽 1 的分配器。文件列表已以正确的顺序排序—内部 DoVAAlloc 方法然后需要依照以下逻辑执行分配：

```
///找出由传递到 DoVAAlloc 的句柄引用的文件列表和分配器  
///如果未找到这些组件的任一个，错误  
///对于文件列表中的每一文件  
///{  
///    文件是内核或内核模块？  
///    {  
///        保存由稍后使用一到下一文件  
///    }  
///    文件是插槽 1 DLL？  
///    {  
///        执行插槽 1 代码分配  
///        如果分配失败，移动到插槽 0  
///        执行插槽 0 数据分配  
///        如果分配失败，错误  
///    }  
///    文件是插槽 0 DLL？  
///    {  
///        执行插槽 0 代码分配  
///        如果分配失败，错误  
///        执行插槽 1 数据分配  
///        如果分配失败，错误
```

```
/// }  
/// 文件仅是文件?  
/// {  
///     不做任何事情一到下一文件  
/// }  
///}
```

对除内核模块之外的每样东西执行模块修正，如图 8 的步骤 803 所示的。为此，提供了 `FileList` 类中的以下函数：

`HRESULT DoVAFixups(BOOL fReverse);`

`fReverse` 自变量指定了模块应当是被修正还是返回重新基于其原始值。这一功能为设备侧更新应用程序所需，但是在构建该文件的台式机上，`fReverse` 总为 `FALSE`（调用者仅需指定文件列表）。注意，较早的 `romimage.exe` 重复通过 `.rel` 文件，并立即修正整个模块，然而，采用组件化的更新，需要仅修正分段的指定页的能力。`Module`（模块）类中的 `FixupBlob` 函数支持取出到模块数据的任意指针，并通过重复通过 `creloc` 分段和仅找出应用到该数据的修正来修正它。盘映象实用程序 230 在每一模块分段上仅调用一次该函数。设备侧更新应用程序将在它所重建的每一二进制差页之后调用它。注意，在这一点上，模块的重定位信息储存在模块内（在 `creloc` 分段中）。结果，对于设备侧更新，不再需要重复通过重定位文件，因为需要的信息在模块中自包含。

在图 8 的步骤 808 执行物理布局之前，如图 8 的步骤 804 所示的压缩模块数据段。注意，盘映象实用程序 230 不支持模块的代码段的压缩，因为 `ROMIMAGE` 和 `RAMIMAGE` 分区打算原地执行。然而，如有需要，可容易地添加代码压缩。

提供了 `FileList`（文件列表）类中的以下函数用于压缩目的：

`HRESULT DoCompression();`

`DoCompression` 在指定的文件列表上重复，并压缩除内核模块之外的所有东西的数据段。更新储存在 `Section`（分段）类中的 `o32rom` 头部的 `psize` 成员以反映压缩的数据长度。可使用任一合适的压缩算法，如一个当前的 `romimage.exe` 实现使用被优化用于解压的算法。然而，由于 `romimage.dll` 由设备侧更新应用程序使用，可使用其它算法作为替代。

一旦压缩了数据段，过程具有执行物理布局所需的信息。为此，提供了 FileList 类中的以下函数：

```
HRESULT DoPhysicalLayout(Allocator &Alloc);
```

图 8 的步骤 805 表示输出分区。对于 ROMIMAGE 分区，使用关联的 FLASH 部分的分配器。该分区的起始位置通过查询 MSPart 来确定。对于 RAMIMAGE 分区，使用物理布局的 RAM 分配器。该分区的起始位置由分区的 RAMIMAGE 元素的 FIXUP_ADDRESS 属性来确定。

DoPhysicalLayout 使用首次满足算法并以以下顺序重复通过文件列表的内容：

1. 代码段（页对齐的）
2. 数据段（包括.creloc 和所有文件）
3. TOC（目录，包括所有 TOC 条目）
4. 所有 e32 头部
5. 所有 o32 头部块
6. 复制段块
7. 文件名

在这一阶段，生成目录（TOC）的大部分内容。这对储存模块头部的物理位置的地方是必需的。通过在布局模块头部时生成 TOC，避免了创建另一数据结构来保留该元数据的需要。

为向 kdata 分配复制段（图 8 的步骤 807）和空间，使用 RAM 分配器。注意，对于 ROMIMAGE 分区，该步骤可较早执行，但是不顺序执行，以保持 ROMIMAGE 和 RAMIMAGE 过程尽可能地相似。类似于插槽 0/1 分配，在 FileList 类中提供了一个函数：

```
HRESULT DoCopySectionAlloc(Allocator &Alloc);
```

一旦该步骤完成，过程可填满 pToc 的 RAMStart、RAMFree 和 RAMEnd 属性。

一旦分配了复制段，修正内核和内核模块，如图 8 的步骤 806 所示的。类似于插槽 0/1 修正，在 FileList 类中提供了以下函数：

```
HRESULT DoKernelVAFixups(BOOL fReverse);
```

DoKernelVAFixups 重复通过内核模块的列表，并通过调用每一 Module（模块）对象的 DoKernelFixups 方法来执行必需的修正。

在这一点上，已修正了所有东西，并且已执行了物理分配。盘映象实用程序

230 已与 MSPart 通信，并具有到分区搜寻数据的句柄。为输出最终的映象，提供了用于写分区的函数：

```
HRESULT DoWriteNKPartition(HVOL hVolume, Allocator &Alloc);
```

对于 RAMIMAGE 和 ROMIMAGE 分区，DoWriteNKPartition 创建 RAM 中分区的表示，然后将该存储器块作为单个文件输出，类似于盘映象实用程序 230 本身处理 BINARY 分区的方法。Alloc 自变量对 ROMIMAGE 分区是必需的，它需要在写出时跳过 RESERVE 区域。

在 ROMIMAGE 和 RAMIMAGE 分区之间只有相对较小的差异，包括 RAMIMAGE 分区的物理布局使用 RAM 分配器，而 ROMIMAGE 的物理布局使用父闪存部分的分配器。同样，对于 RAMIMAGE 分区，DoWriteNKPartition 也可能失败，因为在闪存部分上可能没有足够的空间。由于 ROMIMAGE 分区使用父闪存部分的分配器，对于 ROMIMAGE 分区，DoWriteNKPartition 应当从不失败。可以用 C 和 C#来实现，令管理的 dskimage.exe 作出对未管理的 romimage.dll 的调用。

盘映象实用程序 230 负责重复通过_FLATRELEASEDIR\DskImage\Partition 中的清单 (DSM) 文件，并将 DSM 文件条目 (和 DSM 文件本身) 添加到文件列表，以由 romimage.dll 处理。对于 ROMIMAGE 分区，盘映象实用程序 230 调用 MSPart 以在尝试物理布局之前创建新分区；(对于 RAMIMAGE，它可在任何时刻创建分区)。在输出分区之后，盘映象实用程序 230 如上所述地调整其大小。其它操作由 romimage.dll 处理。

如上所述，构建的另一包分区类型是 IMGFS。鉴于多种原因，这在其它分区之后构建，包括由于 Windows CE 内核强加的限制，对 RAMIMAGE/ROMIMAGE 区域的分配需要首先出现，以及 IMGFS 分区需要 NOR 上的 SectorInfo。样本 NOR 闪存媒体驱动程序假定组织闪存使得包含 SectorInfo 的分区在没有 SectorInfo 的分区之后到来。构建 IMGFS 分区的步骤类似于 ROMIMAGE/RAMIMAGE 分区的步骤。以下步骤本质上从 ROMIMAGE/RAMIMAGE 分区构建器中重复利用：

- 构建并排序模块和文件的列表；

- 向所有模块分配虚拟地址空间；

- 执行模块修正；以及

- 输出实际分区。

IMGFS 的物理布局由 IMGFS 和 MSPart 处理。类似于

RAMIMAGE/ROMIMAGE 分区，存在用于写 IMGFS 分区的 BuildIMGFSPartition API:

```
HRESULT BuildIMGFSPartition(  
    HANDLE hFileList,  
    HANDLE hVolume,  
    HANDLE hSlot0Alloc,  
    HANDLE hSlot1Alloc,  
    DWORD dwReserved);
```

期望指定的插槽 0 和插槽 1 分配器与早先用于构建 RAMIMAGE/ROMIMAGE 分区的分配器相同。

IMGFS 负责压缩。默认地，压缩文件和模块数据段。

注意，IMGFS 分区服务不能包含内核模块（在 ce.bib 中具有 K 标志的模块）。由此，在执行 VA 分配之前，BuildIMGFSPartition 核实指定的 FileList 不包含任何内核模块。

接下来，调用以下 FileList 成员函数：

1. DoVAAlloc
2. DoVAFixups
3. DoWriteIMGFSPartition（图 9）

如图 9 中一般表示的，在执行模块修正之后，过程重复（步骤 900 和 918）通过指定的文件列表，为每一条目创建 IMGFS 中的新文件。注意，如果当前列表项目是模块（步骤 902），则它在各个分段上依次处理（步骤 904 到 912）。否则，创建文件（步骤 914），并写数据（步骤 916）。

构建的另一分区类型是 USERSTORE，它本质上是用户期望的任一类型的文件系统。它理想地被设计成用于 FAT 或扩展分区（PART_TYPE 0x04 和 0x05）。由于在一个当前实现中，memorycfg.xsd 不允许用户为 USERSTORE 指定长度，该分区跨越剩余的闪存。为构建 USERSTORE，过程调用 FSDMGR 来创建指定 PART_TYPE 的分区，并指令 FSDMGR 使用闪存部分上的剩余空间，即，通过指定它应当被自动确定大小（由此，在一个实现中，有每一闪存部分一个 USERSTORE 的限制）。

后处理过程

依照本发明的一个方面，当完成时，盘映象实用程序已在构建系统（如，台式机）上生成了单个文件 206，它包含对应于不同文件系统的一个或多个分区。在那些分区内，将安装包内容，在适当时在虚拟地址空间修正可执行模块。注意，在这一点上，用户仅指示硬件的某些细节，即 RAM 和存储位置（地址）、其大小和该存储是线性还是基于块的。尚未指定的是闪存是 NOR 闪存还是 NAND 闪存，或硬盘驱动器或某些其它类型的存储的细节。存储技术可在映象上施加影响，取决于它如何被管理。

例如，闪存通常被划分成块，并进一步被细分成页或扇区，如图 10A 和 10B 中一般表示的。这些子组件的每一个的大小和存储技术标识其每一个的方式（如数字地），以及管理每一页或扇区的坏/保留/只读状态的方式是存储技术专用的。

后处理过程步骤的目的是将这一管理信息以不违反映象布局的要求的方式引入到映象中，如在盘映象实用程序的输入文件中所指定的。例如，如果分区之一要在 NOR 闪存外执行，则有指定它能够以 CPU 页增量被映射到 CPU 的 CPU 要求，并且由于存储管理要求，这无法被改变。

盘映象实用程序 230 工具的后处理过程阶段用于对 MSPart 生成的 NB0 文件作出存储硬件专用调整。例如，如图 10A 中所示的，NAND 闪存具有在单独的空间内伴随扇区的扇区信息（每一扇区的元数据 *m*）。然而，NOR 硬件需要修改 IMGFS 分区以在每一扇区的块内包括 SectorInfo，如在图 10B 中由在每一块的剩余扇区中维护的四扇区块的三个扇区的元数据（*m*）所一般表示的。注意，为进行块对齐，某些空间未使用，如由图 10B 的阴影区所表示的。具有成对闪存的设备制造商也可希望将 NB0 分离成用于每一闪存部分的单独的文件。

后处理器 232（图 2，如 postproc.exe）可由盘映象实用程序 230 在 memory.cfg.xml 的任一 NOR 闪存元素上自动执行。给定 NOR 元素的 ID、VASTART、BLOCKSIZE 和 SECTORSIZE 属性，后处理器输出 ID.nb0.postproc 和 ID.bin。为进一步后处理盘映象实用程序 230 的输出，用户（如，OEM）可创建一个 postdiskimage.bat 文件，它由 buildpkg 脚本在盘映象实用程序 230 过程完成之后调用。

在操作中，后处理器 232 打开 NB0，并找出主引导记录，并使用其中的数据来查找 IMGFS 分区。后处理器 232 然后以（例如，NOR）块驱动程序能够理解的

格式向每一扇区添加扇区数据。后处理器 232 也移动 USERSTORE(如果存在的话)的起始,并保存修改的 NB0,因为在向 IMGFS 添加了扇区数据之后,它不再对应于闪存块的起始。后处理器 232 还可将修改的 NB0 转换成用于传统引导加载器的二进制文件。

如可从以上详细描述所见到的,提供了一种将操作系统映象组件转换成基于文件系统的制造映象的机制。该映象文件独立于任一特定的存储技术,并适合用作初始映象,同时方便了设备的组件化更新,

尽管本发明易受各种修改和替换构造的影响,仍在附图中示出了某些说明的实施例并在上文详细描述了它们。然而应当理解,这并非将本发明限于所揭示的具体形式,而是相反,本发明覆盖落入本发明的精神和范围之内内的所有修改、替换构造和等效技术方案。

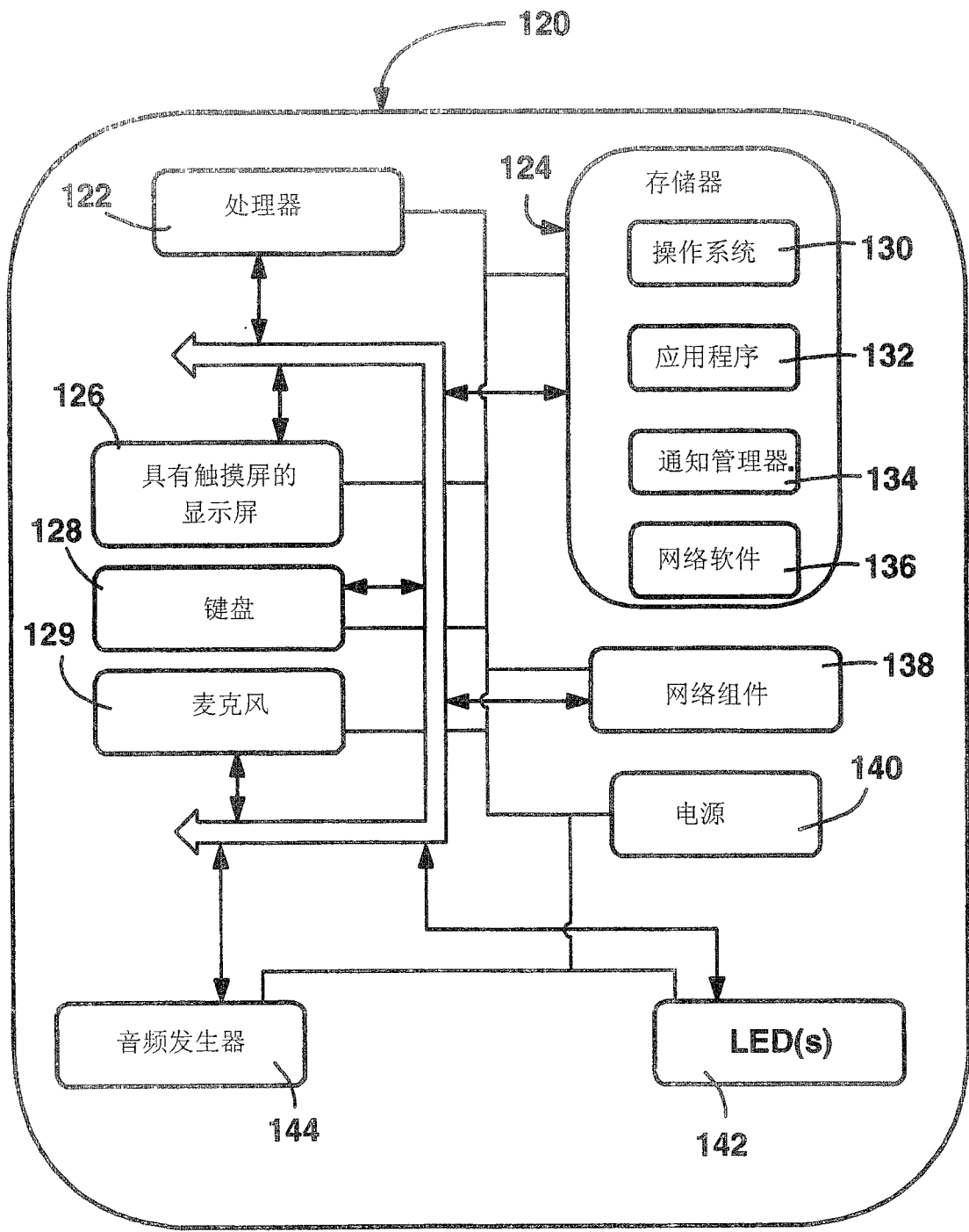


图 1

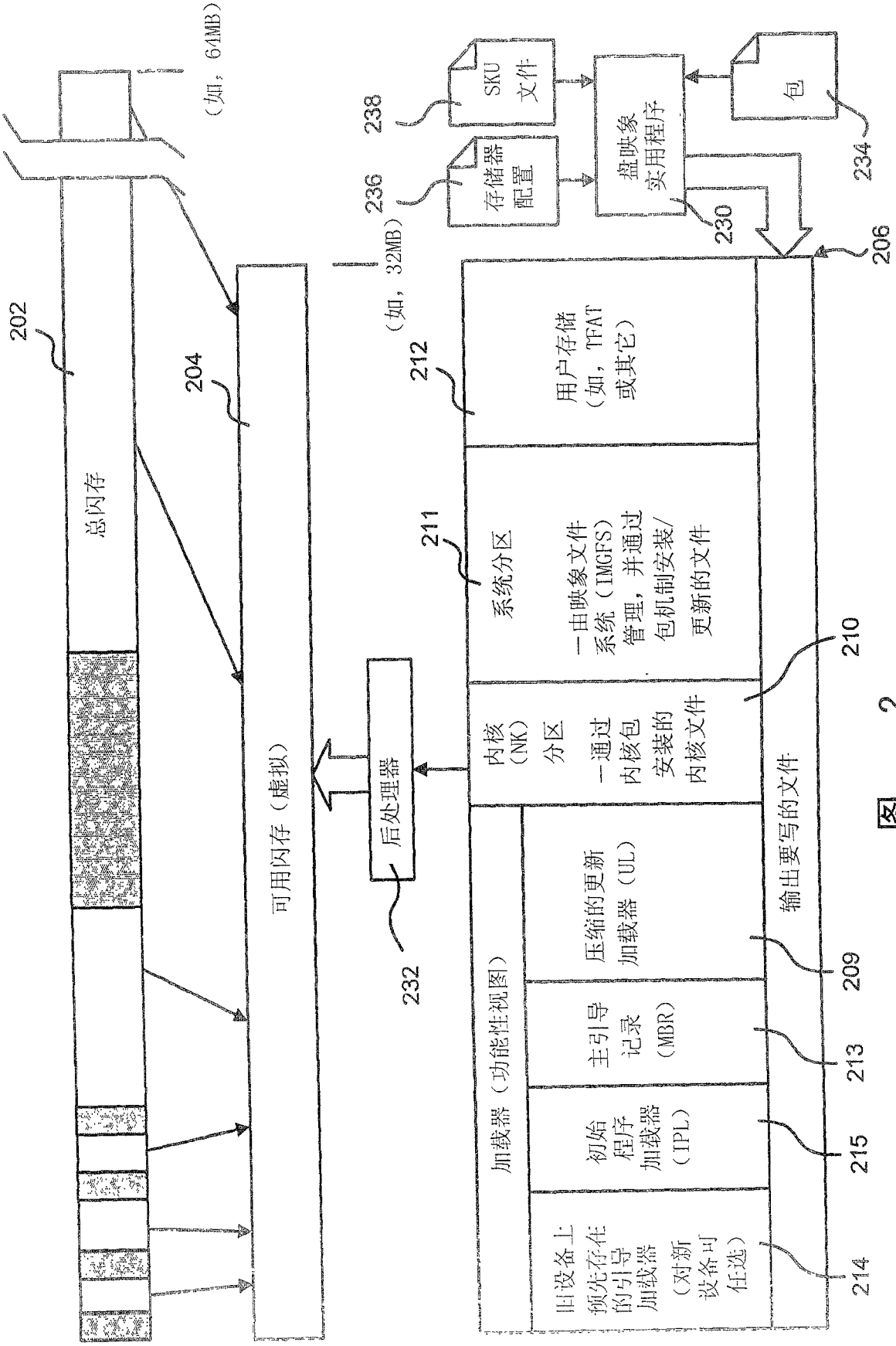
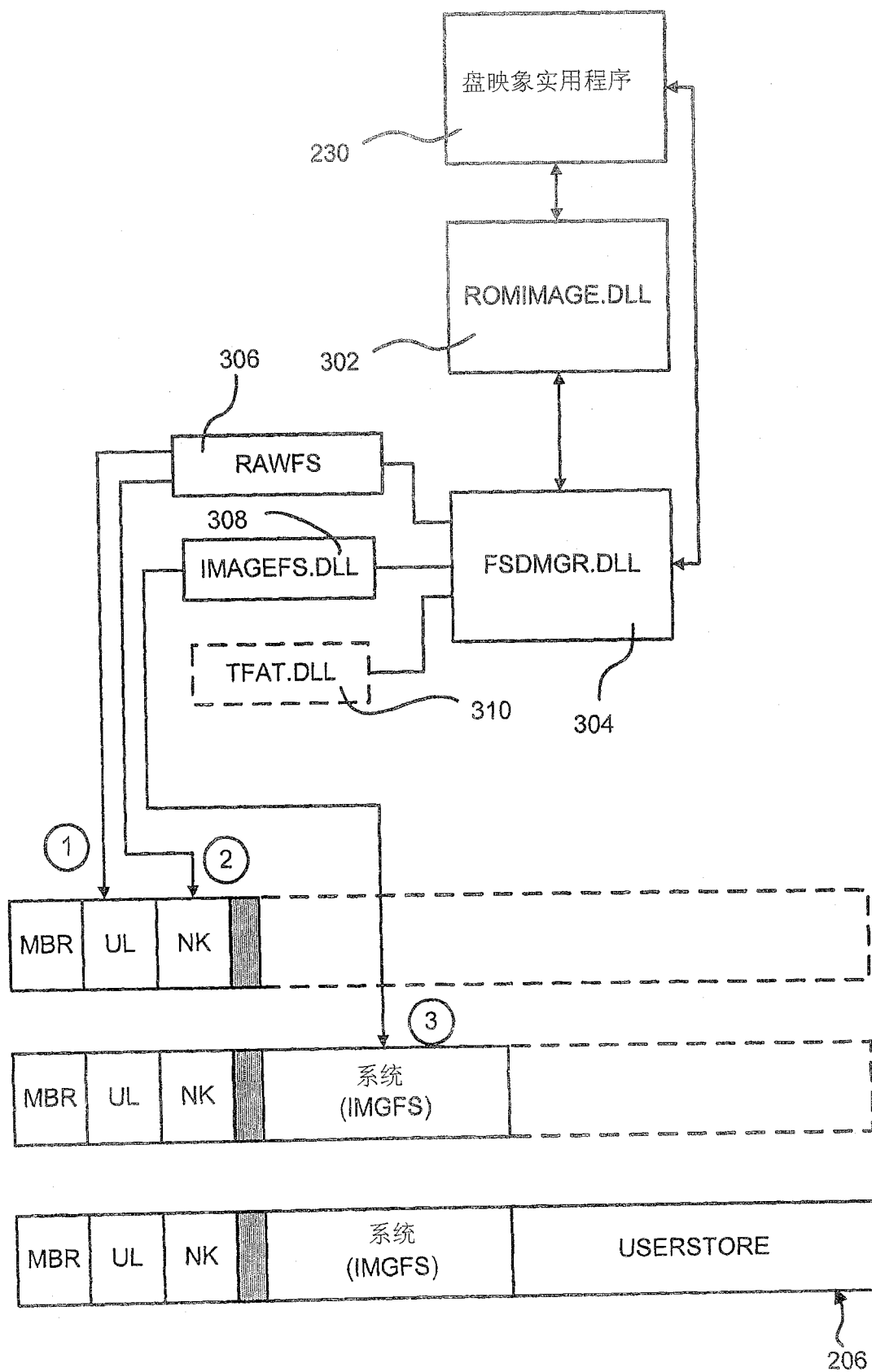


图 2



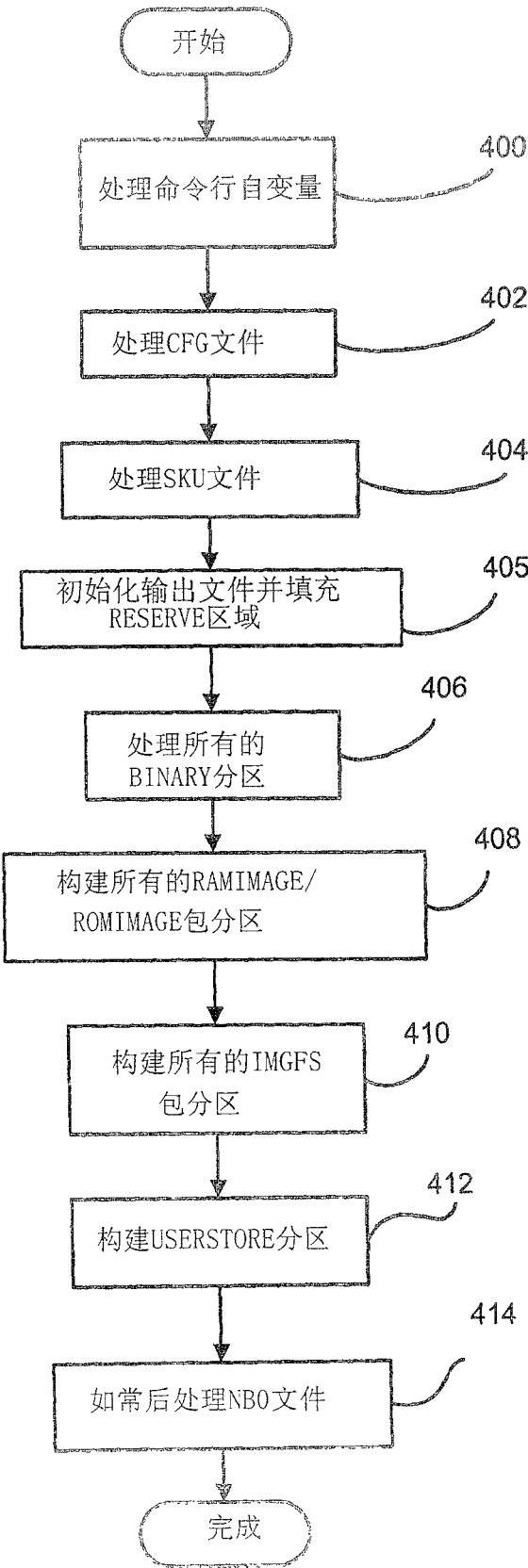


图 4

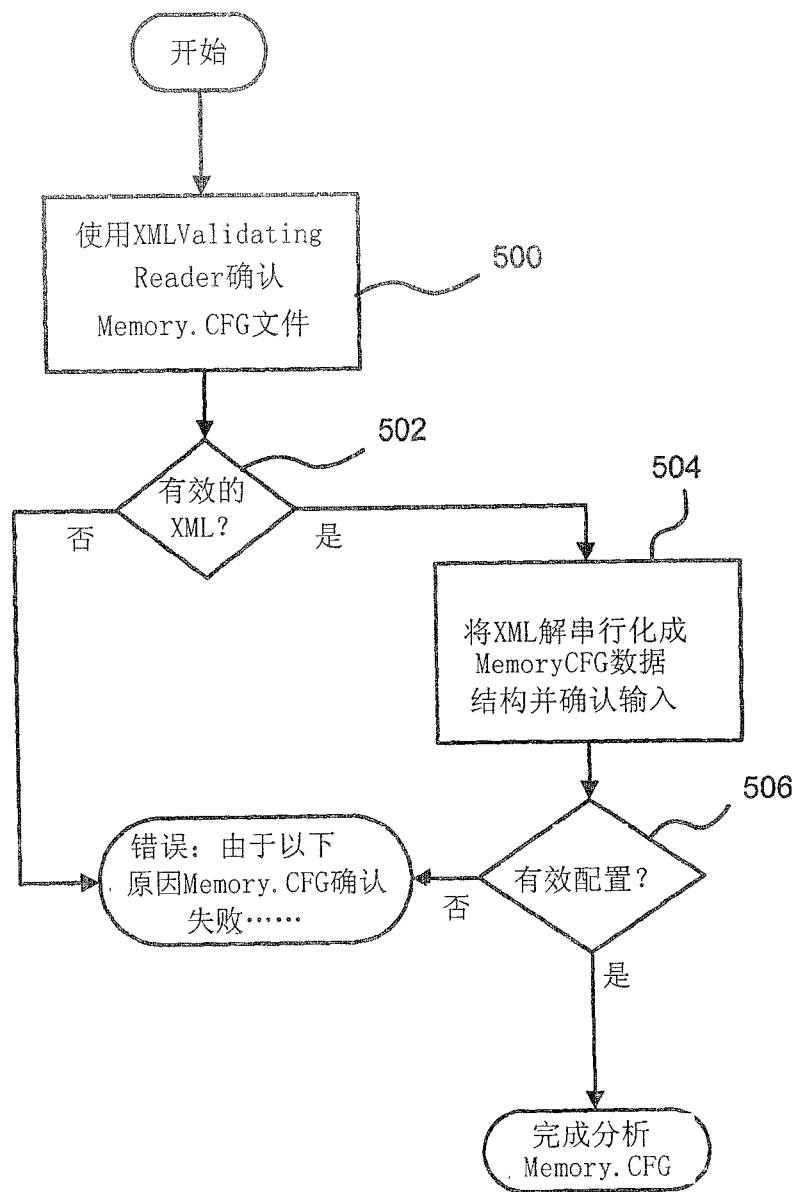


图 5

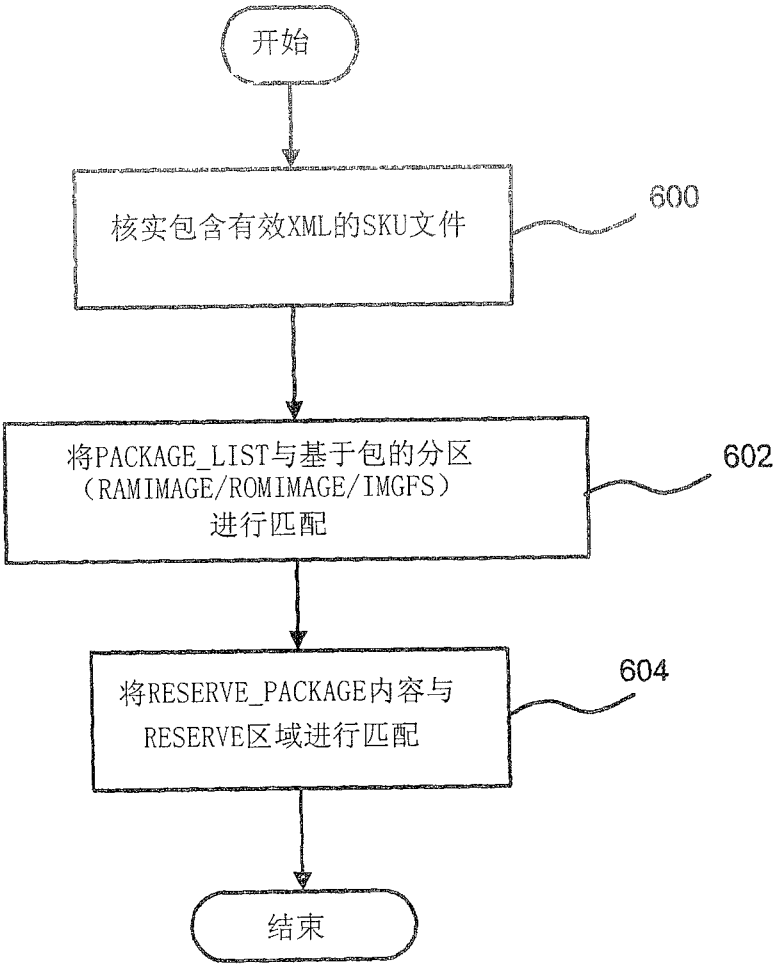
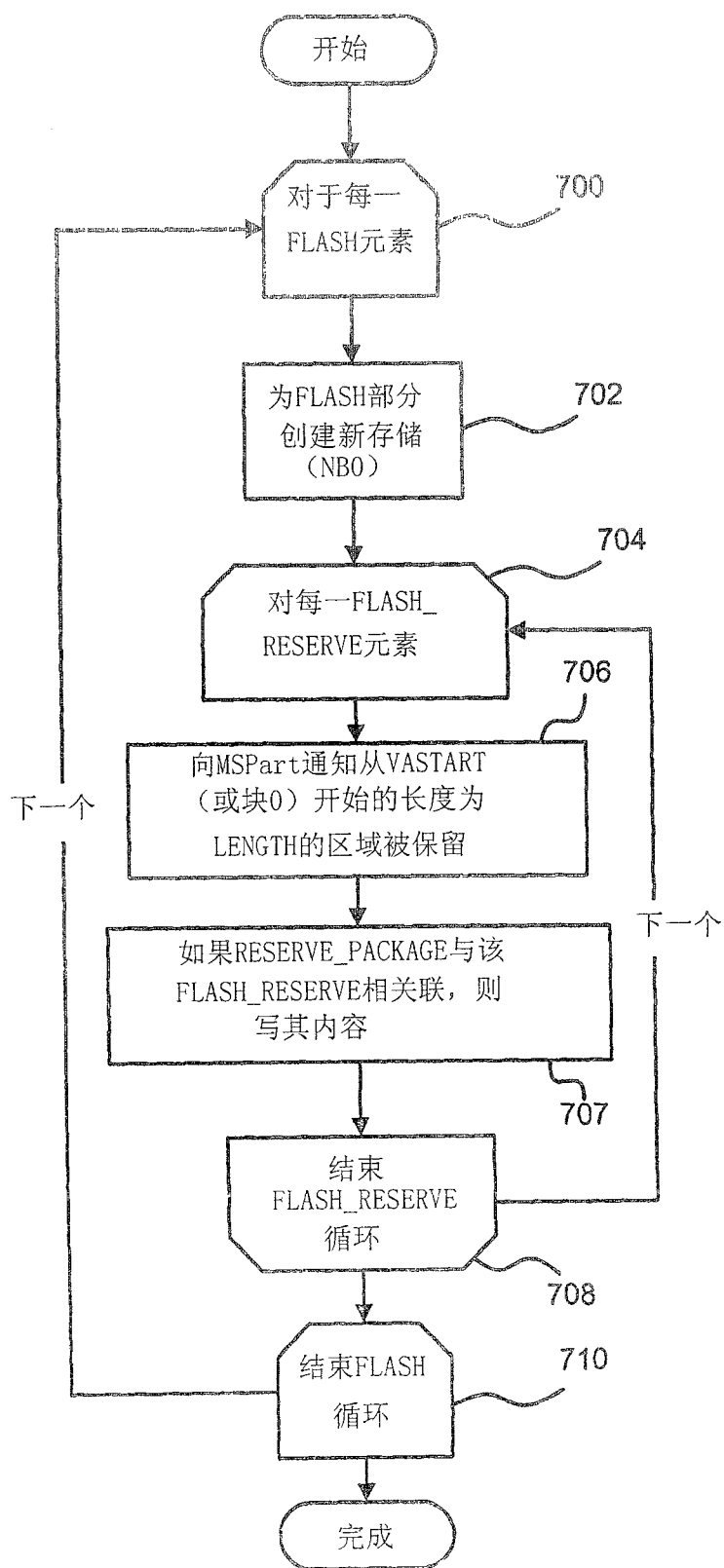
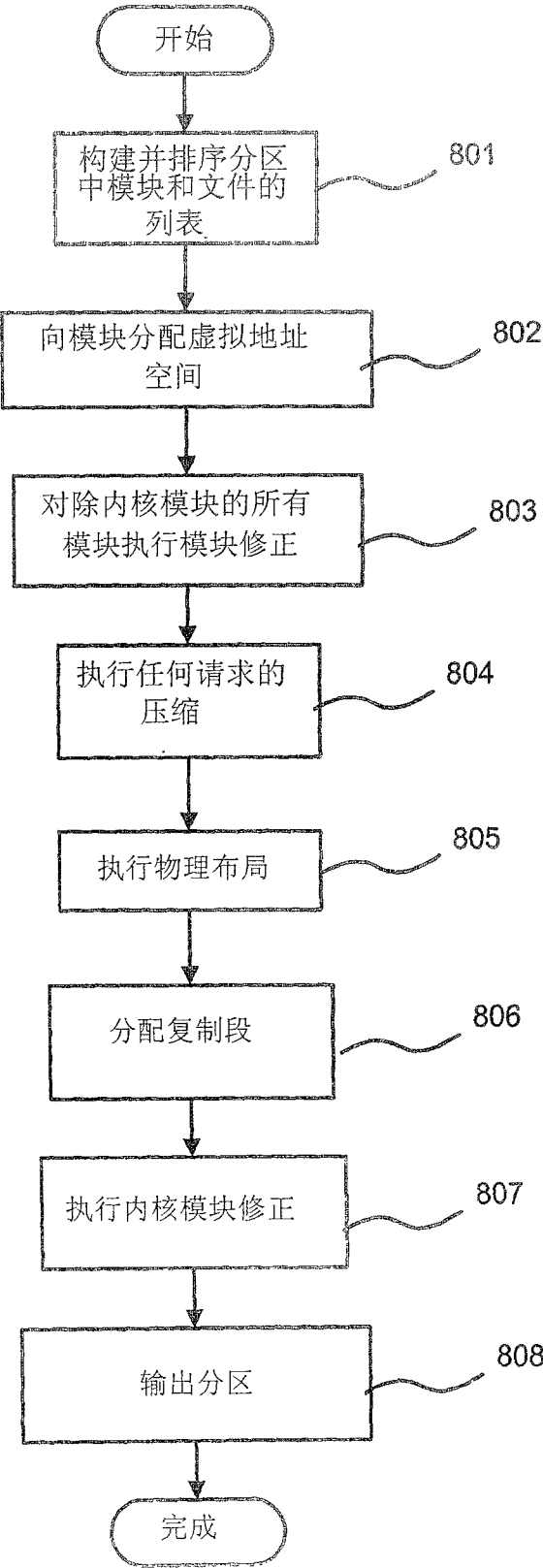


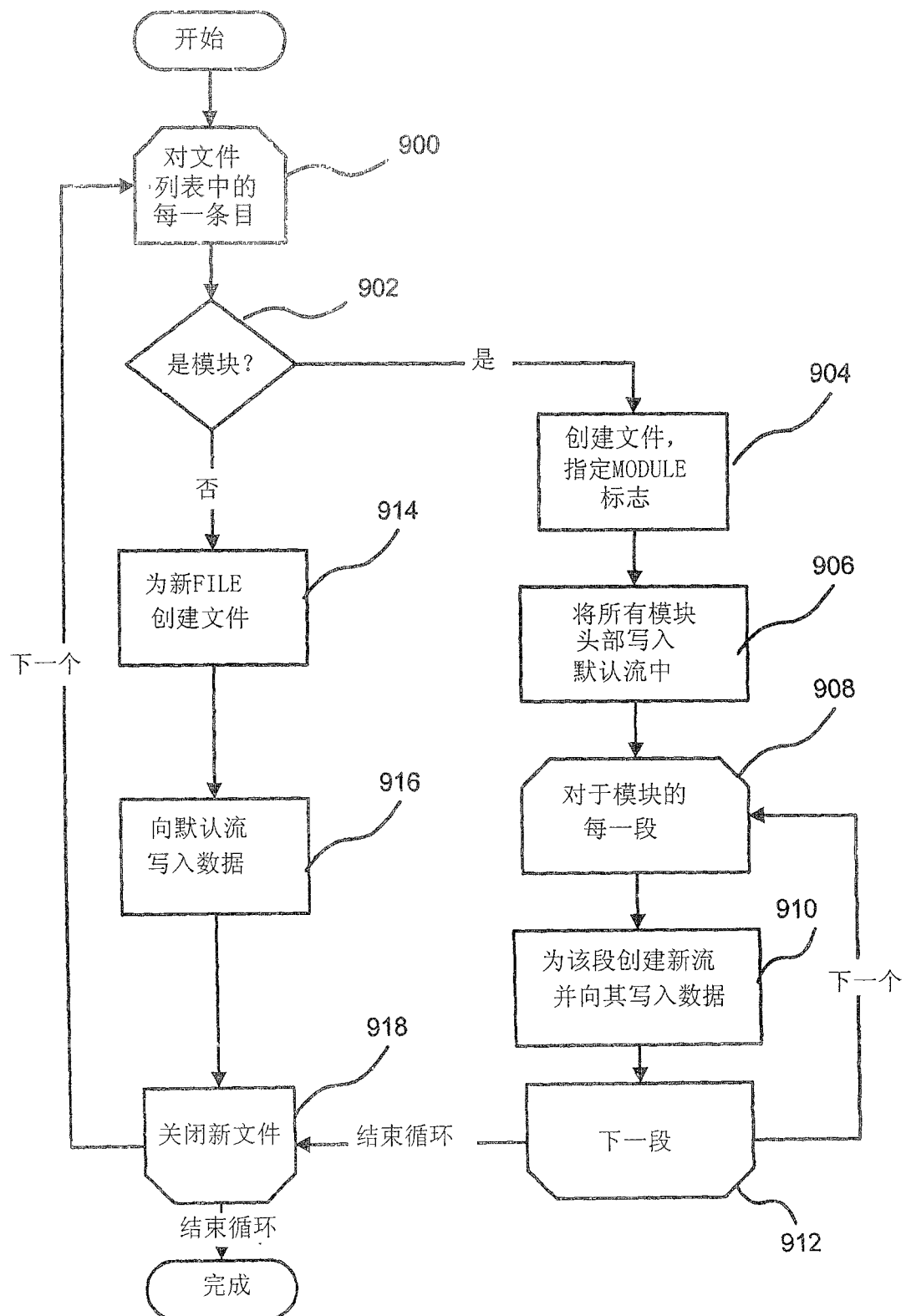
图 6





图

8



图

9

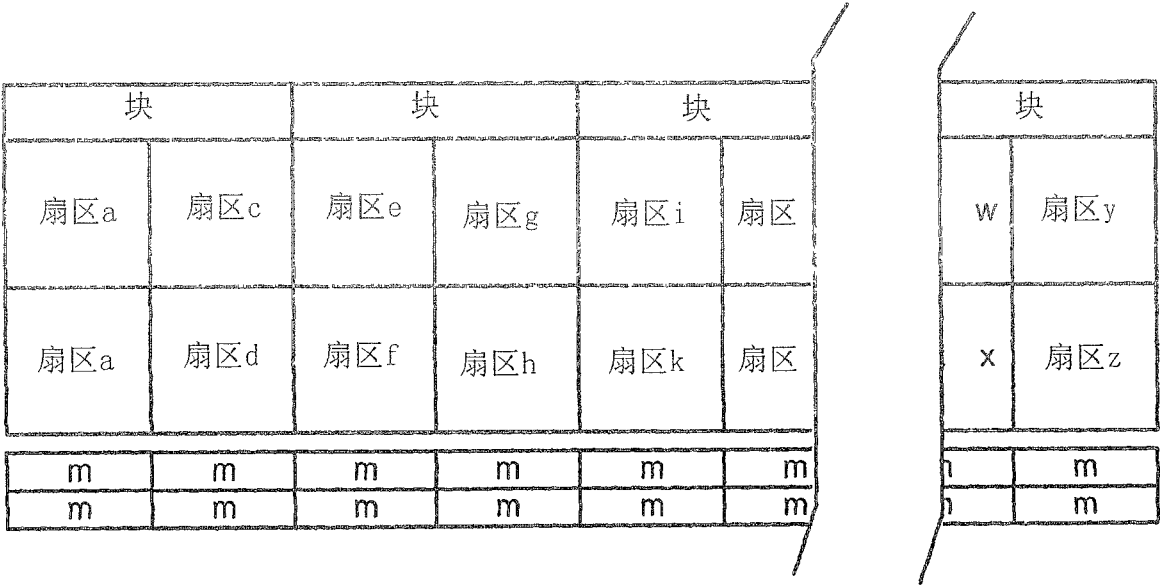


图 10A

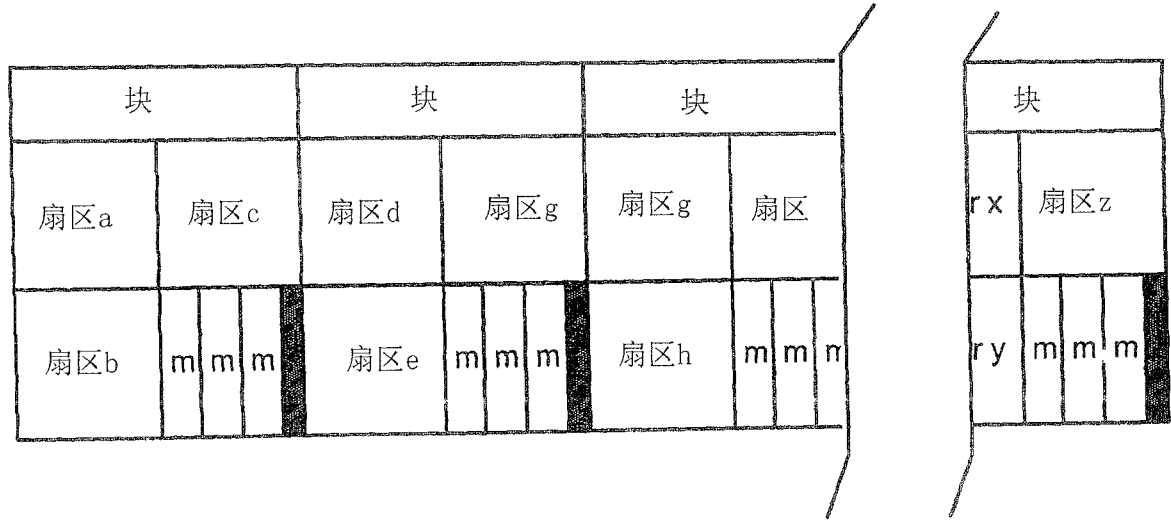


图 10B