(19)日本国特許庁(JP)　(12)公開特許公報(A)　(11)特許出願公開番号
特開2005-182809
(P2005-182809A)
(43)公開日　平成17年7月7日(2005.7.7)

(51)Int.Cl.[7]	F I		テーマコード（参考）
G06F 11/00	G06F	9/06 630A	5B076

審査請求 未請求　請求項の数 41　ＯＬ　外国語出願　（全 44 頁）

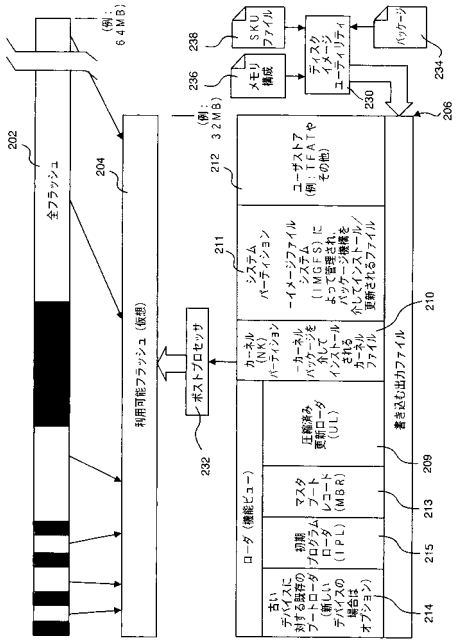
(21)出願番号　特願2004-363716 (P2004-363716)
(22)出願日　平成16年12月15日 (2004.12.15)
(31)優先権主張番号　60/530,135
(32)優先日　平成15年12月16日 (2003.12.16)
(33)優先権主張国　米国 (US)
(31)優先権主張番号　10/837,150
(32)優先日　平成16年5月1日 (2004.5.1)
(33)優先権主張国　米国 (US)

(71)出願人　500046438
マイクロソフト　コーポレーション
アメリカ合衆国　ワシントン州　９８０５２－６３９９　レッドモンド　ワン　マイクロソフト　ウェイ
(74)代理人　100077481
弁理士　谷　義一
(74)代理人　100088915
弁理士　阿部　和夫
(72)発明者　アンドリュー　エム．ロジャース
アメリカ合衆国　９８０５２　ワシントン州　レッドモンド　ワン　マイクロソフト　ウェイ　マイクロソフト　コーポレーション内

最終頁に続く

(54)【発明の名称】ストレージ技術抽象化方式におけるファイル内でのファイルシステムの生成

(57)【要約】
【課題】　単一の製造イメージファイルが、個々のオペレーティングシステムコンポーネントパッケージの書き込み先である内部パーティション／ファイルシステムレイアウトを含んでいる、コンピュータデバイスの製造で使用されるシステムおよび方法を提供する。
【解決手段】　イメージファイルは、どんな特定のストレージ技術からも独立している。ファイルを生成するために、ディスクイメージユーティリティツールが、メモリ構成ファイルおよびパッケージ－パーティションマッピングファイルを入力して、イメージファイル内にパーティション／ファイルシステムを生み出す。オペレーティングシステムコンポーネント（パッケージ）をパーティションに書き込む。イメージファイルのデータを特定のストレージデバイスに書き込む（フラッシュする）ために、イメージファイルを後処理して、特定のストレージデバイスに合わせてデータをカスタマイズする。
【選択図】　図２



【特許請求の範囲】
【請求項１】
コンピューティング環境で、
第１の記憶媒体にインストールすべきイメージデータを含むパッケージにアクセスするステップと、
前記パッケージの内容の書き込み先となるパーティションの記述にアクセスするステップと、
第２の記憶媒体への出力ファイルを生成するステップであって、前記出力ファイルはパーティションを含み、各パッケージの内容は前記記述に基づいて前記パーティションのうちの１つに記憶されるステップと
を備えることを特徴とする方法。
【請求項２】
メモリ構成ファイルにアクセスして、前記出力ファイル内の前記パーティションのうちの少なくとも１つの位置を決定するステップをさらに備えることを特徴とする請求項１に記載の方法。
【請求項３】
少なくとも１つのパッケージの内容はオペレーティングシステムコンポーネントに対応し、前記パーティションのうちの１つはシステムパーティションを含み、前記記述に基づいて前記オペレーティングシステムコンポーネントを前記システムパーティションに書き込むステップをさらに備えることを特徴とする請求項１に記載の方法。
【請求項４】
少なくとも１つのパッケージの内容はカーネルコンポーネントに対応し、前記パーティションのうちの１つはＲＡＭベースのイメージパーティションを含み、前記記述に基づいて前記カーネルコンポーネントを前記ＲＡＭベースのイメージパーティションに書き込むステップをさらに備えることを特徴とする請求項１に記載の方法。
【請求項５】
少なくとも１つのパッケージの内容はカーネルコンポーネントに対応し、前記パーティションのうちの１つはＲＯＭベースのイメージパーティションを含み、前記記述に基づいて前記カーネルコンポーネントを前記ＲＯＭベースのイメージパーティションに書き込むステップをさらに備えることを特徴とする請求項１に記載の方法。
【請求項６】
少なくとも１つのパッケージの内容は更新ローダコンポーネントに対応し、前記パーティションのうちの１つはバイナリイメージパーティションを含み、前記記述に基づいて前記更新ローダを前記バイナリイメージパーティションに書き込むステップをさらに備えることを特徴とする請求項１に記載の方法。
【請求項７】
前記出力ファイルを生成するステップは、前記ファイルの最後まで延びるパーティションにデータを書き込むステップと、書き込まれた実際のデータ量に基づいて前記パーティションのサイズを調整するステップとを含むことを特徴とする請求項１に記載の方法。
【請求項８】
前記出力ファイルを生成するステップは、前記ファイルの最後まで延びるパーティションにデータを書き込むステップと、前記書き込まれたデータの最後の後に空き空間バッファを追加するステップと、書き込まれた実際のデータ量および前記空き空間バッファに基づいて前記パーティションのサイズを調整するステップとを含むことを特徴とする請求項１に記載の方法。
【請求項９】
前記ファイル中のマスタブートレコードにデータを書き込んでパーティションを定義するステップをさらに備えることを特徴とする請求項１に記載の方法。
【請求項１０】
マスタブートレコードにデータを書き込むステップはユーザストアパーティションを定

義するステップを含むことを特徴とする請求項７に記載の方法。
【請求項１１】
前記出力ファイルに追加データを書き込むステップをさらに備えることを特徴とする請求項１に記載の方法。
【請求項１２】
後処理を実施して、前記出力ファイルから前記第１の記憶媒体にデータを転送するために準備するステップをさらに備えることを特徴とする請求項１に記載の方法。
【請求項１３】
前記第１の記憶媒体は内部のコードをインプレース実行できるフラッシュメモリを構成し、メモリ構成ファイルにアクセスして、内部のコードがインプレース実行されるように前記パーティションのうちの少なくとも１つをフィックスアップするステップをさらに備えることを特徴とする請求項１２に記載の方法。
【請求項１４】
メモリ構成ファイルにアクセスして、前記第１の記憶媒体のどのセクションが予約済みかを決定するステップをさらに備えることを特徴とする請求項１に記載の方法。
【請求項１５】
実行されたときに請求項１に記載の方法を実施するコンピュータ実行可能命令を有することを特徴とする１つまたは複数のコンピュータ読み取り可能な記録媒体。
【請求項１６】
コンピューティング環境で、
記憶媒体にインストールすべきイメージデータを含むパーティションを備えたファイルにアクセスするステップと、
前記ファイルを処理して、特定のストレージ技術によって必要とされる修正があれば前記パーティションおよびファイルシステムレイアウトに修正を導入するステップと
を備えることを特徴とする方法。
【請求項１７】
前記ファイルから前記特定のストレージ技術に対応する前記記憶媒体にデータを転送するステップをさらに備えることを特徴とする請求項１６に記載の方法。
【請求項１８】
メモリ構成ファイルにアクセスして、前記出力ファイル内の前記パーティションのうちの少なくとも１つの位置を決定するステップをさらに備えることを特徴とする請求項１６に記載の方法。
【請求項１９】
前記イメージデータはパッケージに対応し、パーティション－パッケージマッピングファイルおよびメモリ構成ファイルから前記ファイルを生成するステップをさらに備えることを特徴とする請求項１６に記載の方法。
【請求項２０】
実行されたときに請求項１７に記載の方法を実施するコンピュータ実行可能命令を有することを特徴とする１つまたは複数のコンピュータ読み取り可能な記録媒体。
【請求項２１】
コンピューティング環境で、
パーティション情報およびパッケージ情報を含むパーティション－パッケージマッピングファイルを入力し、前記ファイル中のデータに基づいて、前記パッケージ情報中で識別されるパッケージを使用してイメージファイルを出力するディスクイメージユーティリティプロセスであって、前記イメージファイルは、前記パーティション情報中で識別される複数のパーティションを含み、各パッケージの内容は前記記述に基づいて前記パーティションのうちの１つに記憶されるディスクイメージユーティリティプロセスと、
前記イメージファイルを処理して、特定のストレージ技術によって必要とされる修正があれば前記イメージに修正を導入するポストプロセス
を備えたことを特徴とするシステム。

【請求項２２】
前記ディスクイメージユーティリティプロセスはさらに、デバイスのためのストレージがどのように構成されるかを示すデータを含むメモリ構成ファイルを入力することを特徴とする請求項２１に記載のシステム。
【請求項２３】
前記デバイスは、内部のコードをインプレース実行できるメモリを備えて構成され、前記メモリ構成ファイルがアクセスを受けて、内部のコードがインプレース実行されるように前記パーティションのうちの少なくとも１つがフィックスアップされることを特徴とする請求項２２に記載のシステム。
【請求項２４】
前記メモリ構成ファイルがアクセスを受けて、デバイスメモリのどのセクションが予約済みかが決定されることを特徴とする請求項２２に記載のシステム。
【請求項２５】
少なくとも１つのパッケージの内容はオペレーティングシステムコンポーネントに対応し、前記パーティションのうちの１つはシステムパーティションを含み、前記ディスクイメージユーティリティは前記記述に基づいて前記オペレーティングシステムコンポーネントを前記システムパーティションに書き込むことを特徴とする請求項２１に記載のシステム。
【請求項２６】
少なくとも１つのパッケージの内容はカーネルコンポーネントに対応し、前記パーティションのうちの１つはＲＡＭベースのイメージパーティションを含み、前記ディスクイメージユーティリティは前記記述に基づいて前記カーネルコンポーネントを前記ＲＡＭベースのイメージパーティションに書き込むことを特徴とする請求項２１に記載のシステム。
【請求項２７】
少なくとも１つのパッケージの内容はカーネルコンポーネントに対応し、前記パーティションのうちの１つはＲＯＭベースのイメージパーティションを含み、前記ディスクイメージユーティリティは前記記述に基づいて前記カーネルコンポーネントを前記ＲＯＭベースのイメージパーティションに書き込むことを特徴とする請求項２１に記載のシステム。
【請求項２８】
少なくとも１つのパッケージの内容は更新ローダコンポーネントに対応し、前記パーティションのうちの１つはバイナリイメージパーティションを含み、前記ディスクイメージユーティリティは前記記述に基づいて前記更新ローダを前記バイナリイメージパーティションに書き込むことを特徴とする請求項２１に記載のシステム。
【請求項２９】
前記ディスクイメージユーティリティは、前記ファイルの最後まで延びるパーティションにデータを書き込み、書き込んだ実際のデータ量に基づいて前記パーティションのサイズを調整することを特徴とする請求項２１に記載のシステム。
【請求項３０】
前記ディスクイメージユーティリティは、前記ファイルの最後まで延びるパーティションにデータを書き込み、前記書き込んだデータの最後の後に空き空間バッファを追加し、書き込んだ実際のデータ量および前記空き空間バッファに基づいて前記パーティションのサイズを調整することを特徴とする請求項２１に記載のシステム。
【請求項３１】
前記ディスクイメージユーティリティは前記ファイル中のマスタブートレコードにデータを書き込んでパーティションを定義することを特徴とする請求項２１に記載のシステム。
【請求項３２】
前記ディスクイメージユーティリティは前記マスタブートレコードにデータを書き込むことによってユーザストアパーティションを定義することを特徴とする請求項３１に記載のシステム。

【請求項３３】
データ構造を記憶した少なくとも１つのコンピュータ読み取り可能な記録媒体であって、前記データ構造は、
前記データ構造中のパーティションを定義するマスタブートレコードを含む第１のデータセットであって、各パーティションがファイルシステムに対応する第１のデータセットと、
バイナリパーティションを含む第２のデータセットと、
ストレージベースのパーティションを含む第３のデータセットと、
システムパーティションを含む第４のデータセットとを備え、
前記データ構造が処理されて記憶媒体上にオペレーティングシステムイメージが生み出されることを特徴とするコンピュータ読み取り可能な記録媒体。
【請求項３４】
前記バイナリパーティションは更新ローダコードを含むことを特徴とする請求項３３に記載のデータ構造を記憶したコンピュータ読み取り可能な記録媒体。
【請求項３５】
前記更新ローダコードは前記バイナリパーティション中で圧縮されていることを特徴とする請求項３４に記載のデータ構造を記憶したコンピュータ読み取り可能な記録媒体。
【請求項３６】
前記ストレージベースのパーティションは、カーネルコードを含むＲＡＭイメージパーティションを構成することを特徴とする請求項３３に記載のデータ構造を記憶したコンピュータ読み取り可能な記録媒体。
【請求項３７】
前記カーネルコードは前記ＲＡＭイメージパーティション中で圧縮されていることを特徴とする請求項３６に記載のデータ構造を記憶したコンピュータ読み取り可能な記録媒体。
【請求項３８】
前記ストレージベースのパーティションは、カーネルコードを含むＲＯＭイメージパーティションを構成することを特徴とする請求項３３に記載のデータ構造を記憶したコンピュータ読み取り可能な記録媒体。
【請求項３９】
前記システムパーティションはオペレーティングシステムコンポーネントを含むことを特徴とする請求項３３に記載のデータ構造を記憶したコンピュータ読み取り可能な記録媒体。
【請求項４０】
前記第１のデータセット中の前記マスタブートレコードは、前記定義済みパーティションのうちの１つとしてユーザストアパーティションを定義することを特徴とする請求項３３に記載のデータ構造を記憶したコンピュータ読み取り可能な記録媒体。
【請求項４１】
初期プログラムローダコードを含む第５のデータセットをさらに備えることを特徴とする請求項３３に記載のデータ構造を記憶したコンピュータ読み取り可能な記録媒体。
【発明の詳細な説明】
【技術分野】
【０００１】
本発明は一般に、組込みオペレーティングシステムを有するデバイスなどのコンピューティングデバイスに関し、より詳細には、コンピューティングデバイスの不揮発性ストレージを構成することに関する。
【背景技術】
【０００２】
パーソナルディジタルアシスタント、最新の携帯電話機、ハンドヘルドおよびポケットサイズのコンピュータなど、携帯コンピューティングデバイスは重要かつ一般的なユーザ

ツールになりつつある。一般にこれらは、十分に小型化して非常に便利になり、電池電力の消費がより少なくなると同時に、より強力なアプリケーションを実行することができるようになった。

【０００３】

このようなデバイスの製造過程の間には通常、組込みオペレーティングシステムのイメージがモノリシックなイメージファイル中に構築され、各デバイスの不揮発性ストレージ（例えばＮＡＮＤまたはＮＯＲフラッシュメモリ、ハードディスクなど）に記憶される。このため、オペレーティングシステムを構成する様々な部分からモノリシックなイメージファイルを事前に構成しなければならない。さらに、このようなデバイスを時々更新することが必要であるかまたは望ましく、これはオペレーティングシステムに変更を加えることを要する。

【発明の開示】

【発明が解決しようとする課題】

【０００４】

しかし、モノリシックなイメージを扱うときには、いくつかの不都合がある。これらの不都合には、いずれかの更新をインストールするのにイメージ全体（または場合によってはその何らかの事前定義済みサブセット）を置換する必要があることが含まれ、これは一時ストレージおよび処理能力を含めた多量のリソースを要する。オペレーティングシステムの別々のコンポーネントを更新することは様々な競合および依存関係のせいで困難な作業であるため、従来、このようなデバイスを更新するにはモノリシックな置換が用いられてきた。さらに、このようなコンポーネント化は別の問題にもつながる。すなわち、初期イメージは引き続き製造に必要とされるが、従来、初期イメージは本質的に、デバイスに転送されるモノリシックなビットグループでしかなかった。オペレーティングシステムイメージコンポーネントを変換して、初期イメージとして使用するのに適し、かつコンポーネント化方式のデバイス更新を容易にするように設計もされたファイルシステムベースの製造イメージにするための機構が必要とされている。

【０００５】

本発明は、このような状況に鑑みてなされたもので、その目的とするところは、単一の製造イメージファイルが、個々のオペレーティングシステムコンポーネントパッケージの書き込み先である内部パーティション／ファイルシステムレイアウトを含んでいるシステムおよび方法を提供することにある。

【課題を解決するための手段】

【０００６】

簡単に言えば、本発明は、個々のオペレーティングシステムコンポーネントパッケージの構築時のインストール先であるパーティション／ファイルシステムレイアウトを含む、単一の製造イメージファイルを作成することを一般に対象とする方法およびシステムを提供するものである。このプロセスはストレージ技術抽象方式で実施され、したがってこのシステムおよび方法は、基礎をなすストレージ（例えばフラッシュ）のタイプも、かつ／または基礎をなすストレージがイメージのレイアウトに課す場合のあるどんな要件も考慮しない。そうではなく、別個のステップで、得られたイメージファイルを後処理する。後処理には、存在するストレージデバイスの実際のタイプに合わせてイメージファイルをカスタマイズすることが含まれる。

【０００７】

一実装形態では、様々なタイプのパーティションをファイル内に作成する。各パーティションはファイルシステムに対応する。オペレーティングシステムイメージコンポーネント（パッケージと呼ばれる）の集合を、ファイルシステムベースの製造イメージのパーティションに変換する。後でイメージ更新技術がこのデバイス上で同じパーティション／ファイルシステムモデルを利用することができるような形で、このファイルから、製造過程の間に、初期オペレーティングシステムイメージをこのデバイス上で確立することができる。

【０００８】
様々なパッケージを初期製造イメージに変換するために、単一の製造イメージファイルを作成するが、このファイル中で、イメージはパーティション／ファイルシステムレイアウトに構成される。次いで、このファイルの内容を製造時にインストールするために、このファイルを必要に応じて後処理して、必要なメタデータを追加する。
【０００９】
最終的には仮想フラッシュに書き込まれるファイルであるこのファイルを構築するために、様々なパーティションを作成する。デバイス製造業者はフラッシュ全体のうちのいくらかを様々な目的で予約し、残りのメモリをオペレーティングシステムコンポーネントおよびオプションのユーザストアによって使用される仮想メモリとして利用可能にすることができることに留意されたい。各パーティションは何らかの目的を有し、独自のファイルシステムとみなすことができる。例えば、更新ローダなどのためのバイナリパーティションや、カーネル（ＮＫ）パーティションのためのＲＡＭＩＭＡＧＥ／ＲＯＭＩＭＡＧＥパーティションがあるものとすることができる。ＩＭＧＦＳパーティションはオペレーティングシステムファイルを含むことができ、ユーザストアパーティションはユーザデータ記憶のために指定することができる。パーティションを指定するために、マスタブートレコードをファイル中に作成する。追加のデータをファイルに含めることもできる。
【００１０】
一実装形態では、ディスクイメージユーティリティがファイルを作成し、イメージポストプロセッサがメタデータを追加して、ファイルを所望の記憶媒体への書込みに向けて準備する。ディスクイメージユーティリティは、オペレーティングシステムパッケージを取り込み、どのようにパーティションをストレージ中でレイアウトすべきかについてのメモリ記述（メモリ構成ファイル）と、どのようにパッケージをパーティション構造にマッピングすべきかについての記述（パーティションマッピングファイル）とに基づいて、様々なパーティションを含む出力ファイルを作成することを担い、この出力ファイル中では、各パッケージの内容が適切なパーティションに記憶されている。さらに、どんな実行可能コードも、適切な仮想アドレス空間ベースのアドレスにフィックスアップ（fix up）されている。メモリ構成ファイルは、不揮発性ストレージとシステムＲＡＭの両方に対する、オペレーティングシステムの実行時仮想アドレス空間メモリマップを提供する。パーティションマッピングファイルは、一意に識別されるパッケージのリストを含み、パッケージを指定のパーティションにマッピングするのに使用される。
【００１１】
ポストプロセッサは、特定のストレージ技術によって必要とされる修正があれば、出力ファイルに作用してパーティション／ファイルシステムレイアウトに修正を導入する。例えば、フラッシュ中で異なるフラッシュ部分によってセクタレベルの情報が管理される際の異なる方式を扱うために、調整を行う必要がある場合がある。
【００１２】
その他の利点は、以下の詳細な説明を図面と共に読めば明らかになるであろう。
【発明を実施するための最良の形態】
【００１３】
例示的な動作環境
図１に、このような１つのハンドヘルドコンピューティングデバイス１２０の機能コンポーネントを示す。ハンドヘルドコンピューティングデバイス１２０は、プロセッサ１２２、メモリ１２４、表示装置１２６、キーボード１２８（物理的または仮想キーボードとしてもよく、この両方を表すもとしてもよい）を備える。オーディオ入力を受け取るためのマイクロホン１２９があってもよい。メモリ１２４は、揮発性メモリ（例えばＲＡＭ（random access memory））と不揮発性メモリ（例えばＲＯＭ（read only memory）、ＰＣＭＣＩＡ（Personal Computer Memory Card International Association）カードなど）の両方を一般に含む。Ｍｉｃｒｏｓｏｆｔ　Ｃｏｒｐｏｒａｔｉｏｎから出ているＷｉｎｄｏｗｓ（登録商標）オペレーティングシステムや別のオペレーティングシステムなど、

オペレーティングシステム１３０がメモリ１２４中にあり、プロセッサ１２２上で実行される。

【００１４】

１つまたは複数のアプリケーションプログラム１３２が、メモリ１２４にロードされ、オペレーティングシステム１３０上で稼動する。アプリケーションの例には、ｅメールプログラム、スケジューリングプログラム、ＰＩＭ（個人情報管理）プログラム、ワードプロセッシングプログラム、表計算プログラム、インターネットブラウザプログラムなどが含まれる。ハンドヘルドパーソナルコンピュータ１２０はまた、メモリ１２４にロードされた通知マネージャ１３４を備えることもでき、通知マネージャ１３４はプロセッサ１２２上で実行される。通知マネージャ１３４は、例えばアプリケーションプログラム１３２からの通知要求を扱う。また後述するが、ハンドヘルドパーソナルコンピュータ１２０は、ハンドヘルドパーソナルコンピュータ１２０をネットワークに接続するのに適した、ネットワーキングソフトウェア１３６（例えばハードウェアドライバなど）およびネットワークコンポーネント１３８（例えば無線機およびアンテナ）も備え、接続には電話を掛けることが含まれる場合がある。

【００１５】

ハンドヘルドパーソナルコンピュータ１２０は、１つまたは複数の電池として実現される電源１４０を有する。電源１４０はさらに、ＡＣアダプタや電力ドッキングクレードルなど、内蔵電池をオーバーライドするかまたは再充電する外部電源を備えることもできる。

【００１６】

図１に表す例示的なハンドヘルドパーソナルコンピュータ１２０は、３つのタイプの外部通知機構と共に示されている。すなわち、１つまたは複数の発光ダイオード（ＬＥＤ）１４２と、オーディオ発生器１４４である。これらのデバイスは電源１４０に直接に結合させることができ、したがってこれらがアクティブのときは、ハンドヘルドパーソナルコンピュータのプロセッサ１２２および他のコンポーネントが電池電力の節約のためにシャットダウンすることがあっても、これらのデバイスは通知機構によって指定された期間にわたってオンのままでいる。ＬＥＤ１４２は、ユーザが動作を起こすまで無期限にオンのままでいることが好ましい。最新バージョンのオーディオ発生器１４４は、今日のハンドヘルドパーソナルコンピュータ電池に対して電力使用が過多であり、したがって、システムの残りの部分がオフになったときは、または起動後のいくらかの有限期間にわたって、オフになるように設計されることに留意されたい。

【００１７】

基本的なハンドヘルドパーソナルコンピュータを示したが、携帯電話機など、データ通信を受信し、プログラムによって使用されるようにデータを何らかの方式で処理することができるものなら、ほぼどんなデバイスも本発明の実施に等価であることに留意されたい。

【００１８】

ファイル内にファイルシステムを作成する

本発明は一般に、初期ソフトウェアまたはソフトウェア更新が組込みデバイスの不揮発性メモリ、例えばフラッシュメモリに書き込まれるデバイスを含めて、Ｍｉｃｒｏｓｏｆｔ　Ｗｉｎｄｏｗｓ（登録商標）ＣＥ　．ＮＥＴベースのポータブルデバイスなどの小型携帯コンピューティングデバイスに記憶されるソフトウェアをインストールおよび／または更新することを対象とする。ただし本発明は、コンピューティング一般に利益もたらし、したがってその他のコンピューティングデバイスにも適用することができ、また、様々なタイプのメモリ、および／またはハードディスクドライブなどその他のタイプの記憶媒体を含めて、その他のタイプのストレージにも適用することができる。簡単にするために、以下ではデバイスの更新可能ストレージについて用語「フラッシュ」を使用する場合があるが、どんなストレージ機構も等価であることを理解されたい。さらに、用語「イメージ」は一般に、初期ソフトウェアインストールイメージの概念を含み、さらに、既存のイ

メージの一部だけが更新されるときであっても、イメージに対する後続のソフトウェア更新の概念を含む。

【００１９】

実行可能コードとデータの両方を含むイメージをストレージに適用することができる。実行可能コードは、インストール時に、組込みデバイスの仮想アドレス空間環境に合わせてカスタマイズされる。本発明の一態様によれば、イメージ更新技術は一般に、オペレーティングシステムイメージを、別々に更新できる更新可能コンポーネントに分割し、かつ、コンポーネント間のどんな依存性も維持する。理解されるであろうが、初期イメージは、デバイスへの初期インストールならびにその後の更新を容易にするような方式で構成される。

【００２０】

本発明の一態様によれば、オペレーティングシステムイメージコンポーネント（パッケージと呼ばれる）の集合をファイルシステムベースの製造イメージに変換するシステムおよび方法が提供される。これは、パーティション／ファイルシステムモデルを介して実施することができ、それにより出力ファイルが作成される。後でイメージ更新技術がこのデバイス上で同じパーティション／ファイルシステムモデルを利用することができるような形で、このファイルから、製造過程の間に、初期オペレーティングシステムイメージをこのデバイス上で確立することができる。これにより、個々のコンポーネント、パーティション全体、または必要ならイメージ全体の、安全でフェイルセーフな更新が容易になる。

【００２１】

様々なパッケージを初期製造イメージに変換するために、単一の製造イメージファイルを作成することを一般に対象とする方法およびシステムが提供される。この製造イメージファイルは、パーティション／ファイルシステムレイアウトに構成されたイメージを含む。次いで、このファイルを必要に応じて後処理し、その内容を構築時のインストールに向けて準備する。したがって、個々のオペレーティングシステムコンポーネントパッケージは、この製造ファイルに書き込まれる。このファイル構築プロセス全体は、ストレージ（例えばフラッシュ）技術抽象方式で実施され、したがってこのシステムおよび方法は、基礎をなすストレージのタイプも、かつ／または基礎をなすストレージがイメージのレイアウトに課す場合のあるどんな要件も考慮しない。そうではなく別個のステップで、存在するストレージデバイスの実際のタイプに合わせてイメージファイルをカスタマイズすることを含めて、得られたイメージファイルを後処理し、それにより、次いで初期イメージを任意のデバイスに望むように適用することができる。

【００２２】

図２に、後述するように本発明の様々な態様によって容易になる方式で製造イメージがレイアウトされる際のレイアウト先であるデバイスのストレージ（例えばフラッシュ）の例を提供する。図２では、全フラッシュ２０２のうちのいくらか（例えば全６４メガバイトのうちの３２メガバイト）が、様々な目的で（例えばデバイスの無線機に関係するコードのために）、デバイス製造業者によって予約されている。残りの部分（例えば３２メガバイト）は、オペレーティングシステムおよび（任意選択で）ユーザストアによって使用される仮想メモリ２０４として利用可能なフラッシュメモリである。後述するように、ファイルは予約済みセクションを考慮に入れる。

【００２３】

本発明の一態様により、かつ後述するように、仮想フラッシュ２０４に書き込まれるファイルであるファイル２０６中に、様々なパーティションを作成する。各パーティションは何らかの目的を有し、独自のファイルシステムを含む（またはそのようにみなすことができる）。例えば図２の実施形態では、更新ローダのためのパーティション２０９、カーネル（ＮＫ）パーティション２１０、システムパーティション２１１、ユーザストアパーティション２１２（例えばユーザデータ記憶のためにフォーマットされたもの）がある。ファイル中には、後述するようにパーティションを指定するためのマスタブートレコード（ＭＢＲ）２１３を作成する。さらに、完全にするために図２に表すように、例えばこれ

らの前にデータを付加することによって、既存のブートローダ２１４（古いデバイスの場合。新しいデバイスではオプション）や初期プログラムローダ（ＩＰＬ）２１５など、追加のデータをファイル２０６中に含めることもできる。容易に理解できるように、この追加データの全部またはいくらかを、他のパーティションとは別にデバイスにフラッシュする（flash）こともでき、その場合、ファイル２０６は残りの仮想フラッシュにフラッシュされることになる。しかし、これらのビットを製造ファイル２０６に含めることで、そのような追加の製造ステップがなくなる。図２のパーティションは正確な縮尺で示したものではなく、さらに全メモリ量および／または利用可能メモリ量は例にすぎないことに留意されたい。

【００２４】

理解されるであろうが、本発明は、標準的なファイルシステム概念を用いて、製造イメージを単一ファイル中にストレージ抽象化方式で作成する能力を提供する。この結果、任意の新しいストレージ技術が利用可能になるのに伴い、単純な方式で、かつコアプロシージャにほとんど／まったく影響を及ぼさずに、組込みおよびその他の解決をその新しいストレージ技術に適合させることができる。例えば、後処理によって、フラッシュファイルシステムイメージをハードディスクイメージになるように適合させることができる。

【００２５】

加えて、ファイルシステムを構築時に個別ファイル内に作成できることは、複雑な区分化、フォーマット化、およびその他のファイルシステム論理を製造時環境で実施する必要がないことを意味する。そうではなく、場合によっては複雑であるパーティション／ファイルシステムベースの方式がイメージの基礎をなし、これが後で実行時にオペレーティングシステムによって使用されるかもしれないにもかかわらず、ストレージにイメージを書き込むための今日の標準的な手段（例えばフラッシュギャングプログラマ、ＪＴＡＧ、またはバイトストリームコピー）が依然として機能することになる。

【００２６】

本発明の様々な態様を達成するために、例示的な一実装形態（上で図２に関して一般に述べたもの）では、２つの主要コンポーネント、すなわちディスクイメージユーティリティ２３０およびイメージポストプロセッサ２３２が提供される。ディスクイメージユーティリティ２３０は、ユーザ（例えば製造業者）がどのようにパーティション／ファイルシステムをストレージ中でレイアウトさせたいかについての記述（ファイル２３６）と、パッケージ２３４をどのようにそのパーティション構造にマッピングすべきかについての記述（ファイル２３８、パッケージ－パーティションマッピングファイルおよび／またはイメージマニフェストファイルとも呼ばれる）と共に、パッケージされたオペレーティングシステムコンポーネント（またはパッケージ）の集合２３４を取り込むことを担う。この情報から、ディスクイメージユーティリティ２３０は、様々なパーティション２０９～２１２を含む単一の出力ファイル２０６を生成する。この出力ファイル２０６中では、各パッケージの内容が適切なパーティションに記憶され、実行可能コードが適切な仮想アドレス空間ベースのアドレスにフィックスアップされている。

【００２７】

この出力ファイル状態になった後、ポストプロセッサ２３２は、特定のストレージ技術によって必要とされる修正があれば、この出力ファイルに作用してパーティション／ファイルシステムレイアウトに修正を導入する。例えば、フラッシュ中で異なるフラッシュ部分によってセクタレベルの情報が管理される際の異なる方式を扱うために、調整を行う必要がある場合がある。これについては後で後処理に関して一般に述べる。

【００２８】

図２の例示的な実装形態では、ディスクイメージユーティリティ２３０はしたがって、２つの入力ファイル２３６および２３８、すなわちメモリ構成ファイル２３６（例えばｍｅｍｏｒｙ．ｃｆｇ．ｘｍｌ）およびパッケージ－パーティションマッピングファイル２３８（例えば＊ｓｋｕ．ｘｍｌ、＊は有効ファイル名を表す）によって駆動される。一般に、パーティションマッピングファイル２３８は、特定のオペレーティングシステムコン

ポーネント（例えばパッケージ）それぞれを、メモリ構成ファイル２３６中で示される特定のパーティションにマッピングする情報を含む。

【００２９】

パーティション２０９～２１２は、異なるタイプのものとすることができる（通常はそうである）。例えば一実装形態では、圧縮済み更新ローダパーティション２０９などのＢＩＮＡＲＹパーティション（ＢＩＮＡＲＹパーティションではビットが単にそのままコピーされる）、ＮＫパーティション２１０など１つまたは複数のＲＡＭＩＭＡＧＥまたはＲＯＭＩＭＡＧＥパーティション、システムパーティション２１１など１つまたは複数のＩＭＧＦＳ（Ｉｍａｇｅ　Ｕｐｄａｔｅ　Ｆｉｌｅ　Ｓｙｓｔｅｍ）パーティション、および／あるいはＴＦＡＴやその他のパーティション２１２などＵＳＥＲＳＴＯＲＥパーティションがあるものとすることができる。パーティションおよび／またはパーティションタイプの総数はいくつでもよいが、現在の一実装形態では、機構を構成しやすいものに維持するために、総数は４つに制限される。これらは例えば、図２に表すような更新ローダ（ＢＩＮＡＲＹ）、ＮＫパーティション（ＲＡＭＩＭＡＧＥまたはＲＯＭＩＭＡＧＥ）、システムパーティション（ＩＭＧＦＳ）、ユーザパーティション（ＵＳＥＲＳＴＯＲＥ）である。このような任意の制限はあるものの、タイプの繰返しがあってもよいことに留意されたい。例えば、４つのパーティションの制限で、３つのバイナリタイプと１つのＩＭＧＦＳが許容されることになり、さらに他のタイプのパーティションも可能である。

【００３０】

メモリ構成ファイル２３６は、不揮発性ストレージとシステムＲＡＭの両方に対する、オペレーティングシステムの実行時仮想アドレス空間メモリマップを示す。このデータは、情報を記憶できる１つまたは複数のパーティションを指定して、パーティションに記憶されるデータに特性を割り当てることにより、ユーティリティのユーザがどのように指定のストレージリソースを利用したいと思っているかを示す。

【００３１】

一実装形態では、メモリ構成ファイル２３６は、以下のＸＳＤ（XMLスキーマ定義用語）に対して妥当性検査されるＸＭＬ（Extensible Markup Language）フォーマットのファイルである。

【００３２】

【数１－１】

```
<?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>
```

【００３３】

【数１－２】

```
                <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>
```

【００３４】

【数１－３】

```
                                    <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"
```

【００３５】

【数１－４】

```
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>
```

【００３６】

【数１－５】

```
</xs:schema>
```

【００３７】

例示的なメモリ構成を以下に示す。

【００３８】

【数２】

```
<?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>
```

【００３９】

この例からわかるように、メモリ構成ファイルのハードウェアセクションは、ＲＡＭの位置と予約済みの各フラッシュ部分の位置との完全な記述を提供し、また、ディスクイメージユーティリティによって記憶域として管理されることになる各フラッシュ部分の完全な記述も提供する。ＮＯＲおよびＮＡＮＤタグを総称的に使用して、線形（ＲＡＭのような）またはブロックベースの記憶域をそれぞれ参照する。各記憶部分には固有の識別子／名前が提供される。パーティションセクションは、指定の記憶部分をどのようにパーティションベースの抽象化において使用すべきかを示す。

【００４０】

例えばＲＯＭＩＭＡＧＥは、パーティションの内容をフィックスアップ／再配置して（後述する）、ストレージ部分から実際に実行すべきであること（インプレース実行すなわちＸＩＰ）を意味する。これは線形タイプのストレージデバイスの特性である。ＲＯＭＩＭＡＧＥが使用される場合、ディスクイメージユーティリティは、どんなインプレース実行コードも圧縮されないようにし、さらに、モジュールの個々のコードセクションが物理空間で隣接するようにすることに留意されたい。すなわち、インプレース実行コードのこのようなコードセクションは、予約済み領域には広がらないことになる。ＲＡＭＩＭＡＧＥタグは、内容をフィックスアップ／再配置してＲＡＭから実行すべきであることを示す（ローダが適時にコードをストレージからＲＡＭに移動すると仮定して）。イメージ記憶に加えて、追加のパーティションタイプを示すこともできる。上の例では、ユーザストアがデータ記憶のためのパーティションとして定義されている。

【００４１】

パーティションマッピングファイル（またはＳＫＵ）２３８は、一意に識別されるパッケージのリストを含み、パッケージを指定パーティションにマッピングするのに使用される。パーティションマッピングファイルは、以下のＸＳＤに対して妥当性検査されるＸＭＬファイルである。

【００４２】
【数３】

```
  <?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>
```

【００４３】
例示的なパーティションマッピング（ＳＫＵ）ファイル２３８は、以下のＸＭＬフォーマットのデータのように見えることになる。
【００４４】
【数４】

```
<?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>
```

【００４５】
この特定の例では、例示的なパッケージファイル（「ｏｅｍ」）が、「ＮＫ」という名前のパーティションにマッピングされ、「ＮＫ」は、上のメモリ構成ファイル２３６中で、コードがＲＡＭ中ではなくフラッシュ自体の上でインプレース実行されるようにフィックスアップ／再配置されるＲＯＭＩＭＡＧＥパーティションとして定義されている。また、この例からわかるように、この指定パーティションは、ハードウェアセクション中で示された、仮想アドレス０×８０００．００００にあり長さ０×３８０．００００バイトであ

る単一のＮＯＲフラッシュ部分にある。また、この例からわかるように、「ｌａｎｇ」という名前のパッケージが、「ＯＳ」として参照される第２のパーティションにマッピングされ、「ＯＳ」は、ＩＭＧＦＳパーティション（「ＩＭＧＦＳ」と呼ばれるファイルシステムドライバによって管理されるパーティション）にあり、また、ハードウェアセクション中で指定された単一のＮＯＲ部分にある（ただし前のＮＫパーティションとは重複しない）。

【００４６】

２つの入力ファイル２３６および２３８中で提供された構成情報に基づいて、ディスクイメージユーティリティ２３０は、示されたパッケージの内容を処理する。所与のパッケージに位置する各実行可能コードは、フィックスアップまたは再配置プロセスを通して、固有の仮想アドレス空間範囲に適切に配置される。パーティション内容がＲＡＭから実行されるように意図されているかフラッシュ中でインプレース実行されるように意図されているかに応じて、ディスクイメージユーティリティ２３０は、実行可能ファイルのサブセクションを仮想アドレス空間全体のどこに配置できるかについての既知の制限（例えばＣＰＵ（central processing unit）アーキテクチャなどに基づく）と共にアドレス空間情報を使用して、各実行可能ファイルを処理し、重複しない仮想アドレス空間範囲に内容を配置する。

【００４７】

図３に一般に表すように、フィックスアッププロセスは、Ｒｏｍｉｍａｇｅ３０２と呼ばれるコンポーネントによって扱われる（このコンポーネントは例えばディスクイメージユーティリティ２３０にリンクされているか、その内部にある。そうでなければそれに関連付けられている）。この例示的な実装形態では、Ｒｏｍｉｍａｇｅ３０２はＤＬＬ（ダイナミックリンクライブラリ）であり、これはまた、コードをインストール時にデバイス上でフィックスアップ／再配置するために、パッケージインストール過程の間にもデバイス上で使用される。構築中に、得られたファイル２０６は、実行時にデバイス上で使用されるのと同じファイルシステムコードを使用して、ファイルを構築しているデスクトップコンピュータに記憶されたファイル内の適切なパーティションに記憶される。非互換性の問題を最小限にするために、コードはデバイスならびにホスト構築システムに対して構築可能であることに留意されたい。

【００４８】

ファイルを構築する前に、ディスクイメージユーティリティ２３０は、特定のサイズ、例えば６４メガバイトのファイルの作成を要求する。ＦＳＤＭＧＲ３０４がファイルを作成した後、ディスクイメージユーティリティ２３０は、メモリ構成ファイル２３６を処理して、ＦＳＤＭＧＲ３０４に予約済みセクションを区画させ、それにより予約済みセクションがそのままの状態になるようにする。残りのメモリ、例えば３２ＭＢは、今やＦＳＤＭＧＲ３０４を介して利用可能である。この時点で、必要なパーティションを備えたファイルを構築する準備ができる。

【００４９】

上の例示的なメモリ構成ファイルからわかるように、ＢＩＮＡＲＹパーティションが望まれている。より具体的には、図３は、ディスクイメージユーティリティ２３０がどのようにｒｏｍｉｍａｇｅ．ｄｌｌ機能およびファイルシステムデバイスドライバマネージャ３０４（ＦＳＤＭＧＲ．ＤＬＬ）と協働してファイル２０６を構築するかについての例を示す。ファイル中にパーティションを作成するために、ディスクイメージユーティリティ２３０は、ＦＳＤＭＧＲ３０４と協働してマスタブートレコードを作成する（マスタブートレコードがまだ存在しないので）。ＢＩＮＡＲＹパーティションが後に続くことを示すデータを、マスタブートレコードに追加する。すなわち、このデータはパーティションを生み出す。この実装形態ではサイズは指定されず、したがって、さしあたりは、残りのフラッシュのサイズ全体がこのＢＩＮＡＲＹパーティションに使用される。

【００５０】

この例で、ディスクイメージユーティリティ２３０は、図３に一般に示すように丸付き

数字１で符号を付けた矢印を介してＦＳＤＭＧＲ３０４が更新ローダをＢＩＮＡＲＹパーティションに書き込むよう、Ｒｏｍｉｍａｇｅ３０２を介して要求する。「Applying Custom Software Image Updates To Non-Volatile Storage in a Failsafe Manner」という名称の関連特許出願に記載されているように、更新ローダは本質的に、デバイス更新時のデバイスのブート先であるオペレーティングシステムコードの特別セクションである（通常のオペレーティングシステムコードではなく）。更新ローダはディスクイメージユーティリティから見るとビットのバイナリＢＬＯＢ（Binary Large OBject）なので、パーティションはマウントされ、ＦＳＤＭＧＲ３０４は、ｒａｗＦＳ　ｄｌｌ３０６を使用してバイナリビットをＢＩＮＡＲＹパーティション２０９に（マスタブートレコード２１３の後に続く空間に）書き込む。

【００５１】

書込みが終わると、ＦＳＤＭＧＲ３０４を呼び出すことによって、書き込まれたデータの実際の量が得られる。自動サイズ変更動作では、次のパーティションの開始に対する新しいオフセットはこの実際のサイズに基づき、したがって、更新ローダパーティション２０９は本質的に、それが必要とするファイル中の空間量だけを消費する。

【００５２】

ディスクイメージユーティリティ２３０は再び、同様にしてファイル中にＮＫパーティションを（すなわちマスタブートレコード２１３にデータを書き込むことによって）作成するために呼出しを行い、次いでＮＫパーティションには、ＮＫパーティションにデータを書き込む要求でｒｏｍｉｍａｇｅ．ｄｌｌを呼び出すことによって書込みが行われる。ＮＫパーティションを構築するためにディスクイメージユーティリティ２３０から送られる、ｒｏｍｉｍａｇｅ．ｄｌｌによって必要とされるパラメータには、構築するファイルのリストと、後述するフラッシュ空間および仮想アドレス空間を割り振るためのアロケータとが含まれることに留意されたい。ｒｏｍｉｍａｇｅ．ｄｌｌ３０２は、このデータをビットのセットにフィックスアップし、次いでこれらをＦＳＤＭＧＲ３０４に提供する。次いでＦＳＤＭＧＲ３０４は、図３に丸付き数字２で符号を付けた矢印で表すように、ｒａｗＦＳ　ｄｌｌ３０６を介してＮＫパーティションを書き込む。例えばいずれかのインプレース実行コードのために、物理フィックスアップならびに仮想フィックスアップが必要な場合があることに留意されたい。

【００５３】

この場合もサイズは指定されておらず、したがって、サイズ変更されるまでは、残りの空間全体がこのＮＫパーティションに使用される。ただしこの例では、ＮＫパーティションに対して書き込まれた正確なサイズに基づいてオフセットをファイルの最後から後退させるのではなく、図３の影付き領域で表すように、ＮＫビットの後に続くいくらかの量の追加空間（バッファ）をＮＫパーティション中に残すことができる。この空き空間バッファは、メモリＸＭＬファイル中で指定され、本質的にオフセットを実際の書込みバイト量よりもわずかに前に移動し、それにより、ＮＫパーティションの将来の更新が現在のＮＫパーティションよりも多くの空間を消費できるようにする。すなわち、後続の更新での増大を見込んでおく。

【００５４】

次に、図３に丸付き数字３で符号を付けた矢印で表すように、ｒｏｍｉｍａｇｅ．ｄｌｌを介してＩＭＧＦＳパーティション（やはりファイルの最後まで延びる）を作成し、書き込む。今回は、ＦＳＤＭＧＲ３０４によって認識されるように、これはバイナリパーティションではなくＩＭＧＦＳパーティションであることに留意されたい。したがって、これは特定のパーティションを実行時にデバイス上で扱うドライバなので、ｉｍｇｆｓ．ｄｌｌ３０８を使用してシステムパーティションに書き込む。さらに、どんな物理フィックスアップもｉｍｇｆｓ．ｄｌｌ３０８を介して抽象化され、したがって、このシステムパーティションに対してｒｏｍｉｍａｇｅプロセス３０２は指定どおりに仮想アドレスフィックスアップを実施するだけであることにも留意されたい。次いで、前述のように自動サイズ変更を実施して、次のパーティションに対するオフセットを書込みデータ量（これに

所望の空き空間バッファを足した量。ただしこの例では空き空間バッファはない）に移動する。

【００５５】

次のパーティションはＵＳＥＲＳＴＯＲＥであり、これは任意のタイプのパーティションとすることができる。この例ではタイプ４のパーティションである。ＵＳＥＲＳＴＯＲＥパーティションは、マスタブートレコードに書込みを行うことによって作成され、ファイルの最後まで延びる。ただし、この時点ではデータは書き込まれず、したがってこのパーティションはマウントされないことに留意されたい。後でユーザがこのパーティションにアクセスしたいと思ったとき、ＦＳＤＭＧＲは、このパーティションがその特定のファイルシステムフォーマットに対応する場合には、例えばＴＦＡＴ．ｄｌｌ３１０などの適切なドライバを介してそうすることになる。したがって、ＴＦＡＴ．ｄｌｌ３１０は、初期ファイルの構築には必要ないが、図３では、実行時にユーザパーティションのデータにアクセスするのに使用できるドライバの例として（破線の四角で）示してある。

【００５６】

本発明の動作の説明に移るが、要約すれば、ディスクイメージユーティリティ２３０は、プラットフォームメモリ構成ファイル２３６（例えばｍｅｍｏｒｙ．ｃｆｇ．ｘｍｌ）およびイメージマニフェストファイル２３８（例えばｐｌａｔｆｏｒｍ．ｓｋｕ．ｘｍｌ）を入力とし、デバイスに対する完全なＲＯＭイメージを表すデータファイル２０６を出力する、デスクトップユーティリティである。ディスクイメージユーティリティ２３０はモジュールデータの再配置を担うので、設計目標の１つは、コードをデバイス側の更新アプリケーションと共有できることである。

【００５７】

動作の中でもとりわけ、ディスクイメージユーティリティ２３０は、デバイスに対するメモリレイアウトと、任意の予約済み領域と、事前構築済みＢＩＮ／ＮＢ０ファイルを含む１つまたは複数のフラッシュパーティションの位置およびサイズとを定義するメモリ構成ファイル２３６を解析する（ＮＢ０ファイルはＲＯＭ中に現れなければならないすべての．ｂｉｎファイルのレイアウトである）。ディスクイメージユーティリティ２３０はまた、パッケージベースのパーティションおよびそれらの内容を宣言するイメージマニフェストファイル２３８も解析する。

【００５８】

図３に一般に表した実装形態では、ディスクイメージユーティリティ２３０は、ＢＩＮＡＲＹ、ＲＡＭＩＭＡＧＥ、ＲＯＭＩＭＡＧＥ、ＩＭＧＦＳ、ＵＳＥＲＳＴＯＲＥパーティションを含めて、様々なタイプのパーティションを構築することができる。ＢＩＮＡＲＹパーティションは、任意のデータを収めるがパッケージは収めない事前構築済みＮＢ０ファイルを含む。名前が示唆するように、ＲＡＭＩＭＡＧＥパーティションはＲＡＭにフィックスアップされ、一方、ＲＯＭＩＭＡＧＥパーティションは、それらが記憶されているＮＯＲ部分からインプレース実行されるようにフィックスアップされる。ｉｍａｇｅＦＳパーティションは、ディスクイメージユーティリティ２３０によってサポートされるパーティションタイプを含み、適したバージョンのＩＭＧＦＳ．ｄｌｌ３０８への呼出しを（ＦＳＤＭＧＲ３０４を介して）単に行うだけで構築することができる。ＵＳＥＲＳＴＯＲＥパーティションは、任意のカスタムパーティションタイプとすることができ、現在は、ブート時に作成されるパーティションのためのマスタブートレコード２１３中のプレースホルダとして使用される。

【００５９】

以下の表に、ディスクイメージユーティリティ２３０の入力ファイルおよび出力ファイルを要約する。

【００６０】

【数５】

用語	説明
CFGファイル	ディスクイメージユーティリティに入力される、ハードウェアおよびパーティションの情報を指定するメモリ構成ファイル。本明細書ではMemory.CFG.XMLとも呼び、ディスクイメージユーティリティへの必須入力である。
SKUファイル	ディスクイメージユーティリティに入力される、パッケージベースのパーティション内容を指定するパーティションマッピングファイル／イメージマニフェストファイル。
RAMIMAGE	RAMから実行されるようにフィックスアップされるバイナリパーティション。パッケージを含むことができる。
ROMIMAGE	フラッシュから実行されるようにフィックスアップされるバイナリパーティション。パッケージを含むことができる。
IMGFS	ディスクイメージユーティリティによってサポートされるパーティションタイプ。システムパーティションのパッケージなど、パッケージを含むことができる。
USERSTORE	ユーザ定義のパーティションタイプ。予想される使用法はFATまたはExtendedパーティションのためのMBRエントリを作成することだが、どんなパーティションタイプとすることもできる。
BINARY	ディスクイメージユーティリティによってサポートされるパーティションタイプ。任意のデータを含む。パッケージを含むことはできない。

【００６１】

ディスクイメージユーティリティ２３０は、いずれかのパッケージベースのパーティション中でモジュールを再配置することを担う。これを達成するために、ディスクイメージユーティリティ２３０は仮想アドレス（ＶＡ）アロケータを利用する。ディスクイメージユーティリティ２３０はまた、デバイスの各フラッシュ部分のパーティションを含む各フラッシュ部分ごとのＢＩＮ／ＮＢ０ファイル（前述のもの）と、Ｍａｓｔｅｒ Ｂｏｏｔ Ｒｅｃｏｒｄ（ＭＢＲ）とを出力することもできる。出力ファイルは、ＭＳＰａｒｔデスクトップコンポーネントを使用して作成することができる。さらに、ディスクイメージユーティリティ２３０中のコードの多く（例えばＶＡアロケータ、ＩＭＧＦＳ対話、モジュール再配置）はまた、デバイス側更新アプリケーションにも有用であり、したがって、ディスクイメージユーティリティ２３０はデバイス制限およびコード移植性を考慮に入れる。

【００６２】

図４に、ディスクイメージユーティリティ２３０の全体的な流れを表す。ディスクイメージユーティリティ２３０は、デスクトップコンポーネントを含むことができ、また、様々な動作を実施するための別々のコンポーネントを含むこともできることに留意されたい。例えば、ディスクイメージユーティリティ２３０はコマンドラインを介して実行することができ、したがってディスクイメージユーティリティ２３０は、ブロック４００で表すようにコマンドライン引数を処理するために、コマンドラインプロセッサを備えることができる。別個のＣＦＧファイルパーサがメモリ構成ファイルを処理することができ（ブロ

ック４０２）、別個のｓｋｕ／パッケージパーサが、ブロック４０４で表すようにパーティションマッピング（ＳＫＵ）ファイルを処理することができる。図４では、コンポーネントおよびそれらの機能が、ＢＩＮＡＲＹパーティションビルダ／プロセス（ブロック４０６）、ＲＡＭＩＭＡＧＥ／ＲＯＭＩＭＡＧＥパーティションビルダ／プロセス（ブロック４０８）、ＩＭＧＦＳパーティションビルダ／プロセス（ブロック４１０）、ＵＳＥＲＳＴＯＲＥパーティションビルダ／プロセス（ブロック４１２）、ポストプロセッサ（ブロック４１４）で表されている。ディスクイメージユーティリティ２３０によって使用されるポストプロセッサおよびデータ構造については、後で別の章で述べる。

【００６３】

前述のように、ディスクイメージユーティリティ２３０（例えばｄｉｓｋｉｍａｇｅ．ｅｘｅ）は、以下のコマンドのようなコマンドライン引数を介して呼び出すことができる。

dskimage CFGfile SKUfile

容易に明らかなはずだが、ＣＦＧｆｉｌｅパラメータはメモリ構成ファイル２３６へのパスであり、メモリ構成ファイル２３６は、前述のように、現プラットフォームに関するＲＡＭおよびフラッシュレイアウトを詳述しパーティションを定義する入力ファイルである。ＳＫＵｆｉｌｅパラメータはパーティションマッピング／イメージマニフェストファイル２３８へのパスであり、パーティションマッピング／イメージマニフェストファイル２３８は、前述のように、パッケージの集合をリストし、それらにパーティションを割り当てるものである。一実装形態では、ディスクイメージユーティリティ２３０は、入力ファイルを解析するときにファイル拡張子を見ないが、呼び出されたとき（例えばスクリプトを介して）は、以下の名前で入力を予想する。

ＣＦＧファイル＝Ｍｅｍｏｒｙ．ｃｆｇ．ｘｍｌ

ＳＫＵファイル＝％＿ＴＧＴＰＬＡＴ％．ｓｋｕ．ｘｍｌ

【００６４】

現在、コマンドラインプロセッサは、ＣＦＧｆｉｌｅおよびＳＫＵｆｉｌｅコマンドライン引数が存在するかどうかをチェックし、それらをそれぞれの（例えばＸＳＤ）バリデータ（validator）に渡すだけである。引数が相対パスで指定された場合は、ディスクイメージユーティリティ２３０は、ディスクイメージユーティリティ２３０の呼出し元であるディレクトリを基準としてファイルを探す。引数が絶対パスで指定された場合は、ディスクイメージユーティリティ２３０は絶対パスを使用してファイルを探す。ディスクイメージユーティリティ２３０中では、パス１はＥｎｖｉｒｏｎｍｅｎｔ．ＣｕｒｒｅｎｔＤｉｒｅｃｔｏｒｙであり、パス２はＣＦＧｆｉｌｅまたはＳＫＵｆｉｌｅである。引数が存在しない場合は、コマンドラインプロセッサは適切な動作を行う（Ｃ＃では、例外を投げ、使用法メッセージをプリントし、終了するなど）。

【００６５】

ｍｅｍｏｒｙ．ｃｆｇ．ｘｍｌパーサ／プロセスは、ハードウェアを記述する際およびフラッシュ部分にパーティションを割り当てる際に多くのフレキシビリティを製造業者にもたらすように設計されている。現在の一実装形態では、メモリ構成パーサ／プロセスパーサ（図４のブロック４０２）は、図５に全体的に示すように２つのレベルの妥当性検査をｍｅｍｏｒｙ．ｃｆｇ．ｘｍｌファイルに対して利用する。第１に、Ｃ＃　ＸＭＬＶａｌｉｄａｔｉｎｇＲｅａｄｅｒクラスを介して、ｍｅｍｏｒｙ．ｃｆｇ．ｘｍｌファイルの構文が正しいかどうか検証する。ステップ５００で表すように、ＸＭＬＶａｌｉｄａｔｉｎｇＲｅａｄｅｒは、ｍｅｍｏｒｙ．ｃｆｇ．ｘｍｌが有効なＸＭＬおよび必要なタグを含むことを確認する。そうでない場合は、ステップ５０２は分岐してエラー状態を出力する。第２レベルの妥当性検査は、ディスクイメージユーティリティ２３０によって実施される。第２レベルの妥当性検査中に、ＲＡＭおよびフラッシュ部分およびパーティションの内部表現を作成する。ＸＭＬシリアライズを用いて内部表現を作成し、次いでこの内部構造に対して妥当性検査を実施（図５のステップ５０４および５０６で表すように）して、無効な構成、例えば存在しないフラッシュ部分に対するパーティションマッピングを

検出する。

【００６６】

例示的な一実装形態（例えばＷｉｎｄｏｗｓ（登録商標）ＣＥベースの実装形態）では、ハードウェア構成が有効であるためには、ファイル中のＲＡＭ　ＳＴＡＲＴ属性が、有効なキャッシュ済みカーネル仮想アドレス（０×８０００００００～０×９ＦＦＦＦＦＦＦ）を参照する必要がある。ＳＴＡＲＴ＋ＬＥＮＧＴＨも有効である必要がある。どんなＲＡＭ＿ＲＥＳＥＲＶＥセクションも、親のＲＡＭ要素のＳＴＡＲＴおよびＬＥＮＧＴＨ属性によって指定されたカーネル仮想アドレス範囲内で開始および終了する必要がある。ＲＡＭ＿ＲＥＳＥＲＶＥ要素は重複することはできず（ＳＴＡＲＴからＳＴＡＲＴ＋ＬＥＮＧＴＨまでは各ＲＡＭ＿ＲＥＳＥＲＶＥに固有であるべきである）、空でない固有のＩＤ（identification）ストリングを有する必要がある。

【００６７】

この例示的な実装形態では、以下の規則を含めた様々な規則を施行することができる。すなわち、ＮＯＲ／ＮＡＮＤ（ＦＬＡＳＨと総称される）は、固有の「ＩＤ」属性を有する必要があり、「ＲＡＭ」という名前を付けることはできない。ＦＬＡＳＨ　ＲＥＳＥＲＶＥ要素は、固有の「ＩＤ」属性を必要とし、８文字よりも長い名前を有することはできない。ＦＬＡＳＨ要素のＬＥＮＧＨ属性はＢＬＯＣＫＳＩＺＥで均等に分割可能である必要があり、ＢＬＯＣＫＳＩＺＥはＳＥＣＴＯＲＳＩＺＥで均等に分割可能である必要がある。ＮＯＲ要素の場合、ＶＡＳＴＡＲＴはブロック整合される必要がある。ＦＬＡＳＨ　ＲＥＳＥＲＶＥ要素のＬＥＮＧＴＨ属性は、親のＦＬＡＳＨ　ＢＬＯＣＫＳＩＺＥで均等に分割可能である必要がある。ＮＯＲ＿ＲＥＳＥＲＶＥ要素の場合、ＶＡＳＴＡＲＴはブロック整合され、ＮＯＲ　ＶＡＳＴＡＲＴおよびＶＡＳＴＡＲＴ＋ＬＥＮＧＨは、ＲＡＭとも他のどんなＮＯＲ要素とも重複すべきではなく、有効なキャッシュ済みカーネル仮想アドレスであるべきである。

【００６８】

さらに、有効であるためには、ＮＯＲ＿ＲＥＳＥＲＶＥ要素が、親のＮＯＲ要素のＶＡＳＴＡＲＴおよびＬＥＮＧＴＨ属性によって指定されたキャッシュ済みカーネル仮想アドレス範囲内で開始および終了する必要がある。ＨＡＲＤＷＡＲＥ条件は、後述するＡｌｌｏｃａｔｏｒクラス階層を使用して検証することができる。有効なキャッシュ済みカーネルアドレス範囲に対してグローバルＡｌｌｏｃａｔｏｒを作成し、これを使用して、ＲＡＭおよびＮＯＲタグに対する有効アドレス、ならびにＲＡＭ部分とＮＯＲ部分の間の衝突を検出する。同様に、各ＲＡＭおよびＦＬＡＳＨ部分についてもＡｌｌｏｃａｔｏｒを作成して、有効なＲＥＳＥＲＶＥ領域を検出する。Ａｌｌｏｃａｔｏｒは、取り出しやすいように、関連するＲＡＭ／ＦＬＡＳＨオブジェクトに記憶することができる。

【００６９】

パーティションデータもまたＭｅｍｏｒｙＣＦＧ階層に記憶される。ＰＡＲＴＩＴＩＯＮＳの妥当性検査に関する規則には、パーティションが固有のＩＤ属性を有すること、ＰＡＲＴＩＴＩＯＮのＳＴＯＲＡＧＥ＿ＩＤ属性がＦＬＡＳＨ部分のＩＤ属性と一致する必要があること、ＲＯＭＩＭＡＧＥパーティションのＳＴＯＲＡＧＥ＿ＩＤ属性がＮＡＮＤフラッシュを参照することはできないことが含まれる。ＲＡＭＩＭＡＧＥ／ＢＩＮＡＲＹの場合、ＣＯＭＰＲＥＳＳ属性はブールタイプであり、したがって｛０，１，ｔｒｕｅ，ｆａｌｓｅ｝のうちの１つをとることしかできない。

【００７０】

ＲＡＭＩＭＡＧＥ／ＲＯＭＩＭＡＧＥ／ＩＭＧＦＳ（ＰＡＣＫＡＧＥパーティションと総称される）もまた妥当性検査され、現在の一実装形態では、ＲＡＭＩＭＡＧＥ／ＲＯＭＩＭＡＧＥパーティションは１つしか存在できない。両方が指定された場合は、妥当性検査は失敗する。ＦＳＲＡＭＰＥＲＣＥＮＴおよびＲＯＭＦＬＡＧＳ属性は、ＲＡＭＩＭＡＧＥ／ＲＯＭＩＭＡＧＥパーティションのテーブルオブコンテンツ中のフィールドに対応する（例えばＷｉｎｄｏｗｓ（登録商標）ＣＥベースの実装形態で）。ＲＡＭＩＭＡＧＥパーティションのＦＩＸＵＰ＿ＡＤＤＲＥＳＳは、ＲＡＭ中のどこでパーティションが開

始すべきかを参照する。この属性は、ＲＡＭ中で少なくとも０×１０００バイトの空きが直後に続く有効な位置をポイントする必要がある。現在、１つのＦＬＡＳＨ部分につき複数のＵＳＥＲＳＴＯＲＥパーティションが存在することはできない。

【００７１】

前述のように、イメージマニフェストファイル（パッケージ－パーティションマッピングまたはＳＫＵファイル）２３８は、パーティションによって構成されるパッケージのリストを含む。このＸＭＬスキーマ（先に示したもの）は、比較的単純である。イメージマニフェストファイルパーサ／プロセスは、ＳＫＵファイルが有効なＸＭＬを含むことを検証することを担い、また、ＰＡＣＫＡＧＥ＿ＬＩＳＴタグ中で指定されたパーティションをｍｅｍｏｒｙ．ｃｆｇ．ｘｍｌ中のＰＡＣＫＡＧＥパーティションＩＤと照合することも担う。ＰＡＣＫＡＧＥ＿ＬＩＳＴタグのＰＡＲＴＩＴＩＯＮ＿ＩＤ属性は、ＰＡＣＫＡＧＥパーティション要素のＩＤ属性と一致する必要がある。ＳＫＵパーサは、一致を見つけることができない場合は例外を投げる。図６のステップ６００、６０２、６０４に、これらの様々な動作を要約する。

【００７２】

各ＰＡＣＫＡＧＥ＿ＬＩＳＴタグは、必要とされるＰＡＲＴＩＴＩＯＮ＿ＩＤ属性を有する。パッケージをパーティションと照合することは、ＰＡＲＴＩＴＩＯＮ＿ＩＤ属性をとり、これをＣ＃　ＨａｓｈＴａｂｌｅへのルックアップとして使用することを含む。一致が見つかった場合は、子のＰＡＣＫＡＧＥ＿ＦＩＬＥタグをＡｒｒａｙＬｉｓｔに変換し、ＰＡＣＫＡＧＥパーティションの既存のパッケージリストとマージする。パッケージの抽出は以下のようにして行うことができる。

【００７３】

【数６】

```
/// 各PackagePartitionにつき
/// {
///     PackagePartition中の各Packageごとにディレクトリ.¥DskImage¥Pa
rtitionIDを作成する
///     {
///         ファイル.¥Packages¥PackageNameを開く
///         ファイルが見つからない場合はエラー
///         内容情報.¥DskImage¥PartitionIDを抽出する
///         抽出が失敗した場合はエラー
///     }
/// }
```

【００７４】

ディスクイメージユーティリティ２３０は、＿ＦＬＡＴＲＥＬＥＡＳＥＤＩＲ環境変数が存在するかどうかチェックして、見つかればこれをカレント作業ディレクトリとして使用することに留意されたい。これは、従来の構築システムの挙動と一致する。従来の構築システムでは、ユーザが＿ＦＬＡＴＲＥＬＥＡＳＥＤＩＲ中のファイルを操作するだけで任意のディレクトリから「ｍａｋｅｉｍｇ」を実行することができる。

【００７５】

パーティションは、ＭＳＰａｒｔのデスクトップ（構築システム）バージョンによって作成され管理される（従来のＷｉｎｄｏｗｓ（登録商標）ＣＥ構築システムの場合と同様に）。ＭＳＰａｒｔインタフェースにより、ディスクイメージユーティリティ２３０は、基礎をなすハードウェアに関する詳細を実際のパーティション構築から抽象化することができる。ＭＳＰａｒｔによって作成されたＮＢ０ファイルに対するどんなハードウェア特有の調整も、後処理で実施される。ＭＳＰａｒｔは、他のディスクイメージユーティリテ

ィ２３０ツールが使用することもでき、その場合、後述するように、いずれかのパーティションを作成する前に、ＭＳＰａｒｔボリュームを各ＦＬＡＳＨ部分ごとに作成し、ＲＥＳＥＲＶＥセクションがあれば区画する。ディスクイメージユーティリティ２３０は、必要とされるアンマネージド関数をラップする別個のクラスｆｓｄｍｇｒ．ｃｓを維持する。プロセスは、フラッシュ部分のグローバルＨａｓｈＴａｂｌｅを維持し、それにより、各フラッシュ部分へ名前でアクセスできるように、かつ各フラッシュ部分を反復できるようにする。上のプロセスは、図７に一般に表すように、各Ｆｌａｓｈ部分のＦｌａｓｈＲｅｓｅｒｖｅ要素を反復して、ｆｓｄｍｇｒ．ｄｌｌへの呼出しを行うことによって達成される。ＲＥＳＥＲＶＥ領域には、この領域が実際に作成されたときにデータが投入される。このために、ＦＳＤＭｇｒクラスが、データをディスクイメージからｆｓｄｍｇｒ＿ｎｔ．ｄｌｌに導いて、ＲＥＳＥＲＶＥ領域を埋める。

【００７６】

ＮＢ０を構築する際の第１のステップは、ＭＳＰａｒｔによってマウントされたＶｏｌｕｍｅにＢＩＮＡＲＹパーティションを加えることである。ＢＩＮＡＲＹパーティションは完全に自立型なので（すなわちフィックスアップを必要とせず、ＮＯＲ上のＳｅｃｔｏｒＤａｔａメタデータも必要としない）、ＢＩＮＡＲＹパーティションを最初に処理する。先に一般に述べたように、ＢＩＮＡＲＹパーティションの構築は比較的単純である。プロセスは、ＢｉｎａｒｙＰａｒｔｉｔｉｏｎｓのアレイを反復し、それらのＣｒｅａｔｅ（）メソッドを呼び出す。各Ｐａｒｔｉｏｔｉｏｎオブジェクトはその親のフラッシュ部分をポイントするので、正しいボリュームハンドルを取り出すのも非常に単純である。

【００７７】

一実装形態では、ＰａｒｔｉｔｉｏｎＩｎｆｏクラス（ＭｅｍｏｒｙＣＦＧ階層の一部）は、ＣｒｅａｔｅＡｌｌＰａｒｔｉｔｉｏｎｓと呼ばれる関数を含み、この関数は、ＢｉｎａｒｙＰａｒｔｉｔｉｏｎオブジェクトを反復してそれらのＣｒｅａｔｅ（）メソッドを呼び出すことによって開始する。ＢｉｎａｒｙＰａｒｔｉｔｉｏｎ．Ｃｒｅａｔｅ（）関数は、新しいパーティションを作成し、これを指定のフラッシュブロックにマウントし、ｍｅｍｏｒｙ．ｃｆｇ．ｘｍｌからのＤＡＴＡＦＩＬＥ属性中で指定されたファイルを開き、ＦＳＤＭｇｒを呼び出してデータをパーティションに書き込む。以下に、この機能に関する擬似コードを示す（いくつかのＭＳＰａｒｔ　ＡＰＩを使用して）。

【００７８】

【数７】

```
/// 利用可能な最大サイズのパーティションを作成する（作成が失敗した場合はエラー）
/// DATAFILEを開く（ファイルが見つからない場合はエラー）
/// データをDATAFILEからパーティションにコピーする（空間が足りなくなる場合はエラー）
/// パーティションおよびDATAFILEを閉じる
/// 使用空間を照会してパーティションのサイズを適切に変更する
```

【００７９】

前述のように、ディスクイメージユーティリティは、ＲＯＭＩＭＡＧＥおよびＲＡＭＩＭＡＧＥタイプのパーティションを作成するのに使用される「ｒｏｍｉｍａｇｅ」を提供する。これらのパーティションはそれぞれ、ＲＯＭおよびＲＡＭからインプレース実行されるパーティションである。これらのパーティションは通常、ＩＭＧＦＳファイルシステムを稼動させるのに必要なシステムコンポーネントの最小限のセットを含む。すなわち、ｎｋ、ｃｏｒｅｄｌｌ、ｆｉｌｅｓｙｓ、ｆｓｄｍｇｒ、ｍｓｐａｒｔ、ｉｍｇｆｓ、および任意の必要なブロックドライバである。

【００８０】

ＲＯＭＩＭＡＧＥ／ＲＡＭＩＭＡＧＥプロセスは一般に、様々なステップに分解することができる（現在の一実装形態におけるこれらのうちの８つを以下にリストする。また図

８のステップ８０１～８０８にも表す）。
モジュールおよびファイルのリストを構築してソートする（ステップ８０１）
すべてのモジュールに仮想アドレス空間を割り振る（ステップ８０２）
モジュールフィックスアップを実施する（ステップ８０３）
すべてのデータセクションおよびファイルを圧縮する（ステップ８０４）
物理レイアウトを実施する（ステップ８０５）
コピーセクションを割り振る（ステップ８０６）
カーネルモジュールフィックスアップを実施する（ステップ８０７）
実際のパーティションを出力する（ステップ８０８）

【００８１】

上の各ステップは、ｒｏｍｉｍａｇｅ．ｄｌｌへの呼出しを行うことによって実施することができ、それにより様々な（例えばＣおよびＣ＃）アプリケーションから割振りおよび再配置の機能に容易にアクセスすることができる。ディスクイメージユーティリティは、ＲＯＭＩｍａｇｅ．ｃｓ．と呼ばれるラッパクラスを介してｒｏｍｉｍａｇｅ．ｄｌｌとインタフェースすることができる。

【００８２】

ｒｏｍｉｍａｇｅ．ｄｌｌは、複数のＡｌｌｏｃａｔｏｒを作成および管理するためのＡｌｌｏｃａｔｏｒクラス階層および機能と、ファイルのリストを作成および管理するためのＦｉｌｅクラス階層および機能と、ＲＯＭＩＭＡＧＥ／ＲＡＭＩＭＡＧＥパーティション構築プロセスの各ステップを実施するための関数とを含む。

【００８３】

Ａｌｌｏｃａｔｏｒクラス階層は、ＲＡＭおよびＦＬＡＳＨ中の利用可能な物理空間を管理するのに使用される。これは、任意割振りと固定予約の両方を行うための機能を公開する。Ａｌｌｏｃａｔｏｒクラス（および子のＡｌｌｏｃＮｏｄｅ）は以下のように定義される。

【００８４】

【数８－１】

```
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
```

【００８５】

【数８－２】

```
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);
};
```

【００８６】

ｒｏｍｉｍａｇｅ．ｄｌｌは、Ａｌｌｏｃａｔｏｒを操作するために以下の関数をエクスポートする。

【００８７】

【数９】

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

【００８８】

Ａｌｌｏｃａｔｅ関数についての有効フラグには、ＢＯＴＴＯＭＵＰ＿ＡＬＬＯＣおよびＴＯＰＤＯＷＮ＿ＡＬＬＯＣが含まれる。名前が示唆するように、これらは両方とも、それぞれカレント割振りウィンドウの最下部および最上部から空き空間を探すファーストフィット（ｆｉｒｓｔ－ｆｉｔ）アルゴリズムである。これらは線形探索を必要とする（したがって、多数の割振りが要求されたときは性能ボトルネックを生じる場合がある）。

【００８９】

Ｆｉｌｅクラス階層は、ファイルまたはモジュールに関するすべてのメタデータを記憶するのに使用され、以下のように定義される。

【００９０】

【数１０－１】

```
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;
```

【００９１】

【数１０－２】

```
    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 SetCrelocPtr(Section *value) { crelocPtr = value; }
    e32_rom *GetE32Ptr() { return e32Header; }
```

【００９２】

【数１０－３】

```
    // API functions
    HRESULT DoSplitAlloc(Allocator &Slot0Alloc, Allocator &Slot1Alloc);
    HRESULT DoSlot0OnlyAlloc(Allocator &Slot0Alloc);
    HRESULT DoKernelVAAlloc(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 MungeRVAs();
    HRESULT UnMungeRVAs();
};
```

【００９３】

ファイルを構築しているデスクトップコンピュータ上で、各ファイルの内容は、ＦｉｌｅおよびＳｅｃｔｉｏｎクラスのデータメンバによってメモリマッピングされポイントされる。Ｆｉｌｅクラス階層を操作するために、以下のデータ構造が定義される。

【００９４】

【数１１－１】

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

【００９５】

【数１１－２】

```
        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 pbBuffer, DWORD cbSize);
    HRESULT WriteCopySections(PBYTE pbBuffer, DWORD cbSize);
    DWORD GetCopySectionCount();
};

extern list<FileList> FileLists;
```

【００９６】

ＲＡＭＩＭＡＧＥ／ＲＯＭＩＭＡＧＥパーティションを構築するために、以下の関数が提供される。

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

```
        HANDLE hSlot1Alloc,
        HANDLE hPhysAlloc,
        HANDLE hRAMAlloc,
        DWORD dwReserved);
```

【００９７】

上の関数で、

ｈＦｉｌｅＬｉｓｔは、ＦｉｌｅＬｉｓｔオブジェクトへのハンドルである。ＦｉｌｅＬｉｓｔオブジェクトの作成については次の章で論じる。

【００９８】

ｈＶｏｌｕｍｅは、ＭＳＰａｒｔボリュームへのハンドルであり、パーティション作成中にＤｓｋＩｍａｇｅによって作成される。

【００９９】

ｈＳｌｏｔ０Ａｌｌｏｃは、開始０×６０００００、長さ０×１Ａ０００００のアロケータへのハンドルである。

【０１００】

ｈＳｌｏｔ１Ａｌｌｏｃは、開始０×２１０００００、長さ０×１Ｆ０００００のアロケータへのハンドルである。

【０１０１】

ｈＰｈｙｓＡｌｌｏｃは、物理アロケータである（ＲＯＭＩＭＡＧＥパーティションの場合はフラッシュ部分、ＲＡＭＩＭＡＧＥパーティションの場合はＲＡＭ）。

【０１０２】

ｈＲＡＭＡｌｌｏｃは、ｍｅｍｏｒｙ．ｃｆｇ．ｘｍｌ中のＲＡＭ要素に対応するＲＡＭアロケータである。

【０１０３】

ｄｗＲｅｓｅｒｖｅｄは、以下のように定義されるＭｉｓｃＮＫＩｎｆｏ構造体へのポインタである。

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

【０１０４】

モジュールおよびファイルのリストを構築してソートするために（図８のステップ８０１）、Ｒｏｍｉｍａｇｅ．ｄｌｌは、ファイルおよびモジュールの複数のリンクリストを管理する。リストを構築するために、Ｒｏｍｉｍａｇｅ．ｄｌｌは以下の関数を提供する。

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

【０１０５】

また、Ｆｉｌｅオブジェクトをあるファイルリストから別のファイルリストに移動するためのＡＰＩがあってもよい。このような関数は以下の形式とすることができる。

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

【０１０６】

モジュールに仮想アドレス空間を割り振るために（図８のステップ８０２）、Ｓｌｏｔ０／１仮想アドレス割振りは、ＦｉｌｅＬｉｓｔと、Ｓｌｏｔ０に対するＡｌｌｏｃａｔｏｒと、Ｓｌｏｔ１に対するＡｌｌｏｃａｔｏｒとを必要とする。ＦｉｌｅＬｉｓｔはすでに正しい順序でソートされていることになり、次いで内部ＤｏＶＡＡｌｌｏｃメソッドは、以下の論理に従って割振りを実施する必要がある。

【０１０７】

【数１２】

```
/// DoVAAllocに渡されたハンドルによって参照されるFileListおよびAlloc
atorを見つける
///これらのコンポーネントが見つからない場合はエラー
/// FileList中の各Fileにつき
/// {
///     Fileはカーネルまたはカーネルモジュールか?
///     {
///         後のために保存する　次のFileに移る
///     }
///     Fileはslot1 DLLか?
///     {
///         slot1ｺｰﾄﾞ割振りを実施する
///         割振りが失敗した場合はslot0に移る
///         slot0データ割振りを実施する
///         割振りが失敗した場合はエラー
///     }
///     Fileはslot0 DLLか?
///     {
///         slot0コード割振りを実施する
///         割振りが失敗した場合はエラー
///         slot0データ割振りを実施する
///         割振りが失敗した場合はエラー
///     }
///     Fileは単にFileか?
///     {
///         何もしない　次のFileに移る
///     {
/// }
```

【０１０８】

図８のステップ８０３で表すように、カーネルモジュール以外のすべてに対してモジュールフィックスアップを実施する。このために、ＦｉｌｅＬｉｓｔクラス中の以下の関数が提供される。

```
HRESULT DoVAFixups(BOOL fReverse);
```

ｆＲｅｖｅｒｓｅ引数は、モジュールをフィックスアップすべきか、あるいはその元の値にリベース（rebase）すべきかを指定する。このような機能はデバイス側更新アプリケーションによって必要とされるが、ファイルを構築しているデスクトップコンピュータ上では、ｆＲｅｖｅｒｓｅは常にＦＡＬＳＥになる（呼出し元はファイルリストを指定するだけでよい）。以前のｒｏｍｉｍａｇｅ．ｅｘｅは、．ｒｅｌファイルを反復してモジュール全体を一度にフィックスアップしていたが、コンポーネント化による更新では、セクションの指定ページだけをフィックスアップする機能が必要であることに留意されたい。Ｍｏｄｕｌｅクラス中のＦｉｘｕｐＢｌｏｂ関数は、モジュールデータへの任意のポインタをとり、ｃｒｅｌｏｃセクションを反復してこのデータに適用されるフィックスアップだけを見つけることによってこれをフィックスアップすることをサポートする。ディスクイメージユーティリティ２３０は、１つのモジュールセクションにつき１度これを呼び出す。デバイス側更新アプリケーションは、構築する各ｂｉｎｄｉｆｆページの後でこれを呼び出すことになる。この時点で、モジュールについての再配置情報がモジュール内（．ｃｒｅｌｏｃセクション中）に記憶されていることに留意されたい。この結果、必要な情報がモジュール自体に含まれているので、デバイス側更新はもはや再配置ファイルを反復する必要はない。

【０１０９】

図８のステップ８０５で物理レイアウトを実施する前に、図８のステップ８０４で表すように、モジュールのデータセクションを圧縮する。ＲＯＭＩＭＡＧＥおよびＲＡＭＩＭＡＧＥパーティションはインプレース実行されるように意図されているので、ディスクイメージユーティリティ２３０は、モジュールのコードセクションの圧縮はサポートしないことに留意されたい。ただし望むなら、コード圧縮は容易に追加することができる。

【０１１０】

圧縮のために、ＦｉｌｅＬｉｓｔクラス中の以下の関数が提供される。

```
HRESULT DoCompression();
```

ＤｏＣｏｍｐｒｅｓｓｉｏｎは、指定のＦｉｌｅＬｉｓｔを反復し、カーネルモジュール以外のすべてのデータセクションを圧縮する。Ｓｅｃｔｉｏｎクラスに記憶されたｏ３２ｒｏｍヘッダのｐｓｉｚｅメンバは、圧縮データ長を反映するように更新される。適した任意の圧縮アルゴリズムを使用することになる。例えば、現在のｒｏｍｉｍａｇｅ．ｅｘｅ実装形態の１つでは、圧縮解除に最適化されたアルゴリズムを使用する。しかし、デバイス側更新アプリケーションによってｒｏｍｉｍａｇｅ．ｄｌｌが使用されるので、他のアルゴリズムを代わりに使用することもできる。

【０１１１】

データセクションを圧縮した後には、プロセスは、物理レイアウトを実施するのに必要な情報を有する。このために、ＦｉｌｅＬｉｓｔクラス中の以下の関数が提供される。

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

図８のステップ８０５に、物理レイアウトの実施を表す。ＲＯＭＩＭＡＧＥパーティションの場合は、関連するＦＬＡＳＨ部分のＡｌｌｏｃａｔｏｒを使用する。パーティションの開始位置は、ＭＳＰａｒｔに照会することによって決定する。ＲＡＭＩＭＡＧＥパーティションの場合は、物理レイアウトのためのＲＡＭ　Ａｌｌｏｃａｔｏｒを使用する。パーティションの開始位置は、パーティションのＲＡＭＩＭＡＧＥ要素のＦＩＸＵＰ＿ＡＤＤＲＥＳＳ属性によって決定する。

【０１１２】

ＤｏＰｈｙｓｉｃａｌＬａｙｏｕｔは、ファーストフィットアルゴリズムを使用し、ＦｉｌｅＬｉｓｔの内容を以下の順序で反復する。

１．コードセクション（ページ整合されている）
２．データセクション（．ｃｒｅｌｏｃおよびすべてのファイルを含む）
３．ＴＯＣ（テーブルオブコンテンツ。すべてのＴＯＣエントリを含む）
４．すべてのｅ３２ヘッダ
５．すべてのｏ３２ヘッダブロック

６．コピーセクションブロック
７．ファイル名

【０１１３】
この段階の間に、ほとんどのテーブルオブコンテンツ（ＴＯＣ）を生成する。これは、モジュールヘッダの物理位置を記憶するための場所に必要である。モジュールヘッダをレイアウトするときにＴＯＣを生成することにより、このメタデータを保持するためにさらに別のデータ構造を生み出す必要が回避される。

【０１１４】
コピーセクション（図８のステップ８０６）およびｋｄａｔａ用空間を割り振るために、ＲＡＭ　Ａｌｌｏｃａｔｏｒを使用する。このステップは、ＲＯＭＩＭＡＧＥパーティションについてはもっと早く実施することもできるが、ＲＯＭＩＭＡＧＥおよびＲＡＭＩＭＡＧＥプロセスをできるだけ同様に保つために、そうしないことに留意されたい。ｓｌｏｔ０／１の割振りと同様、以下の１つの関数がＦｉｌｅＬｉｓｔクラス中で提供される。

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

このステップが完了すると、プロセスは、ｐＴｏｃのＲＡＭＳｔａｒｔ、ＲＡＭＦｒｅ、ＲＡＭＥｎｄ属性を埋めることができる。

【０１１５】
コピーセクションを割り振った後、図８のステップ８０７で表すように、カーネルおよびカーネルモジュールをフィックスアップする。ｓｌｏｔ０／１のフィックスアップと同様、以下の関数がＦｉｌｅＬｉｓｔクラス中で提供される。

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

ＤｏＫｅｒｎｅｌＶＡＦｉｘｕｐｓは、カーネルモジュールのリストを反復し、各ＭｏｄｕｌｅオブジェクトのＤｏＫｅｒｎｅｌＦｉｘｕｐｓメソッドを呼び出すことによって必要なフィックスアップを実施する。

【０１１６】
この時点で、すべてがフィックスアップされ、物理割振りが実施されている。ディスクイメージユーティリティ２３０は、すでにＭＳＰａｒｔと通信しており、データを求めるパーティションへのハンドルを有する。最終的なイメージを出力するために、パーティションを書き込むための以下の関数が提供される。

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

ＲＡＭＩＭＡＧＥおよびＲＯＭＩＭＡＧＥパーティションの場合、ＤｏＷｒｉｔｅＮＫＰａｒｔｉｔｉｏｎは、ディスクイメージユーティリティ２３０自体がＢＩＮＡＲＹパーティションを扱うのと同様にして、ＲＡＭ中にパーティションの表現を作成し、次いでこのメモリブロックを単一のファイルとして出力する。Ａｌｌｏｃ引数は、書き出すときにＲＥＳＥＲＶＥ領域をスキップする必要のあるＲＯＭＩＭＡＧＥパーティションに必要である。

【０１１７】
ＲＯＭＩＭＡＧＥパーティションとＲＡＭＩＭＡＧＥパーティションの間には、比較的小さいいくつかの違いがあるだけである。これらの違いとしては、ＲＡＭＩＭＡＧＥパーティションに対する物理レイアウトはＲＡＭ　Ａｌｌｏｃａｔｏｒを使用するが、ＲＯＭＩＭＡＧＥパーティションに対する物理レイアウトは親フラッシュ部分のＡｌｌｏｃａｔｏｒを使用することが含まれる。また、フラッシュ部分に十分な空間がない場合があるため、ＲＡＭＩＭＡＧＥパーティションに対するＤｏＷｒｉｔｅＮＫＰａｒｔｉｔｉｏｎが失敗することもあり得る。ＲＯＭＩＭＡＧＥパーティションは親フラッシュ部分のＡｌｌｏｃａｔｏｒを使用するので、ＲＯＭＩＭＡＧＥパーティションではＤｏＷｒｉｔｅＮＫＰａｒｔｉｔｉｏｎは決して失敗しないはずである。実装はＣとＣ＃の両方で行うことができ、マネージドｄｓｋｉｍａｇｅ．ｅｘｅがアンマネージドｒｏｍｉｍａｇｅ．ｄｌｌへの呼出しを行う。

【０１１８】

ディスクイメージユーティリティ２３０は、＿ＦＬＡＴＲＥＬＥＡＳＥＤＩＲ＼ＤｓｋＩｍａｇｅ＼Ｐａｒｔｉｔｉｏｎ中のマニフェスト（ＤＳＭ）ファイルを反復し、ＤＳＭファイルエントリ（およびＤＳＭファイル自体）をｒｏｍｉｍａｇｅ．ｄｌｌについてのＦｉｌｅＬｉｓｔに追加して処理することを担う。ＲＯＭＩＭＡＧＥパーティションの場合、ディスクイメージユーティリティ２３０は、物理レイアウトを試みる前に、ＭＳＰａｒｔを呼び出して新しいパーティションを作成する（ＲＡＭＩＭＡＧＥの場合は、いつでもパーティションを作成することができる）。パーティションを出力した後、ディスクイメージユーティリティ２３０は前述のようにこれをサイズ変更する。その他の操作もｒｏｍｉｍａｇｅ．ｄｌｌによって扱われる。

【０１１９】

前述のように、構築される別のパッケージパーティションタイプはＩＭＧＦＳである。これは、いくつかの理由で他のタイプの後に構築される。これらの理由としては、Ｗｉｎｄｏｗｓ（登録商標）　ＣＥカーネルによって課される制限によりＲＡＭＩＭＡＧＥ／ＲＯＭＩＭＡＧＥ領域に対する割振りを先に行う必要があること、およびＩＭＧＦＳパーティションがＮＯＲ上のＳｅｃｔｏｒＩｎｆｏを必要とすることが含まれる。サンプルＮＯＲフラッシュメディアドライバは、ＳｅｃｔｏｒＩｎｆｏを含むパーティションがＳｅｃｔｏｒＩｎｆｏを含まないパーティションの後にくるようにフラッシュが構成されているものとみなす。ＩＭＧＦＳパーティションを構築するステップは、ＲＯＭＩＭＡＧＥ／ＲＡＭＩＭＡＧＥパーティションの場合と同様である。以下のステップは本質的に、ＲＯＭＩＭＡＧＥ／ＲＡＭＩＭＡＧＥパーティションビルダから再利用される。

モジュールおよびファイルのリストを構築してソートする

すべてのモジュールに仮想アドレス空間を割り振る

モジュールフィックスアップを実施する

実際のパーティションを出力する

【０１２０】

ＩＭＧＦＳに対する物理レイアウトは、ＩＭＧＦＳおよびＭＳＰａｒｔによって扱われる。ＲＡＭＩＭＡＧＥ／ＲＯＭＩＭＡＧＥと同様、ＩＭＧＦＳパーティションを書き込むためのＢｕｉｌｄＩＭＧＦＳＰａｒｔｉｔｉｏｎ　ＡＰＩが存在する。

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

指定されるＳｌｏｔ０およびＳｌｏｔ１アロケータは、前にＲＡＭＩＭＡＧＥ／ＲＯＭＩＭＡＧＥパーティションを構築するために使用されたのと同じものになることが予想される。

【０１２１】

ＩＭＧＦＳは圧縮を担う。デフォルトでは、ＦｉｌｅｓおよびＭｏｄｕｌｅデータセクションを圧縮する。

【０１２２】

ＩＭＧＦＳパーティションはカーネルモジュール（ｃｅ．ｂｉｂ中にＫフラグを有するモジュール）を含むことはできないことに留意されたい。したがって、ＶＡ割振りを実施する前に、ＢｕｉｌｄＩＭＧＦＳＰａｒｔｉｔｉｏｎは、指定されたＦｉｌｅＬｉｓｔがどんなカーネルモジュールも含まないことを検証する。

【０１２３】

次に、以下のＦｉｌｅＬｉｓｔメンバ関数を呼び出す。

１．ＤｏＶＡＡｌｌｏｃ

２．ＤｏＶＡＦｉｘｕｐｓ

３．ＤｏＷｒｉｔｅＩＭＧＦＳＰａｒｔｉｔｉｏｎ（図９）

【０１２４】
図９に一般に表すように、モジュールフィックスアップを実施した後、プロセスは指定されたＦｉｌｅＬｉｓｔを反復し（ステップ９００および９１８）、各エントリにつき新しいファイルをＩＭＧＦＳ中に作成する。カレントリストアイテムがＭｏｄｕｌｅである場合は（ステップ９０２）、セクションごとに処理することに留意されたい（ステップ９０４から９１２）。そうでない場合は、ファイルを作成し（ステップ９１４）、データを書き込む（ステップ９１６）。
【０１２５】
構築される別のパーティションタイプはＵＳＥＲＳＴＯＲＥであり、これは本質的に、ユーザが望む任意のタイプのファイルシステムである。これは、ＦＡＴまたはＥｘｔｅｎｄｅｄパーティション（ＰＡＲＴ＿ＴＹＰＥ０×０４および０×０５）に向けて設計されるのが理想的である。現在の一実装形態では、ｍｅｍｏｒｙｃｆｇ．ｘｓｄはユーザがＵＳＥＲＳＴＯＲＥの長さを指定することを許容しないので、このパーティションは残りのフラッシュに広がる。ＵＳＥＲＳＴＯＲＥを構築するために、プロセスはＦＳＤＭＧＲを呼び出して、指定されたＰＡＲＴ＿ＴＹＰＥのパーティションを作成し、ＦＳＤＭＧＲにフラッシュ部分の残りの空間を使用するよう命令する。すなわち、自動サイズ変更されるべきであることを指定することによって命令する（一実装形態では、１つのフラッシュ部分につき１つのＵＳＥＲＳＴＯＲＥという制限がある）。
【０１２６】
後処理
本発明の一態様によれば、完了時、ディスクイメージユーティリティは、構築（例えばデスクトップ）システム上で単一のファイル２０６を生成し終えており、このファイルは、種々のファイルシステムに対応する１つまたは複数のパーティションを含む。これらのパーティション内にパッケージの内容がインストールされることになり、実行可能モジュールが仮想アドレス空間中で適切にフィックスアップされる。この時点では、ユーザはハードウェアの詳細のうちのいくつかだけしか指示していないことに留意されたい。すなわち、ＲＡＭおよびストレージの位置（アドレス）、これらのサイズ、ストレージが線形かブロックベースかに関する詳細である。まだ指定されていない詳細は、フラッシュがＮＯＲフラッシュであるかＮＡＮＤフラッシュであるか、あるいはハードディスクドライブであるか他の何らかのタイプのストレージであるかに関する詳細である。ストレージ技術は、それがどのように管理されるかに応じてイメージに影響を及ぼす場合がある。
【０１２７】
例えば、図１０Ａおよび１０Ｂに一般に表すように、フラッシュストレージは通常、ブロックに分割され、さらにページまたはセクタに細分される。これらの各サブコンポーネントのサイズと、ストレージ技術がそれぞれを識別する方式（例えば数値による方式）と、各ページまたはセクタごとの不良／予約済み／読取り専用ステータスを管理する方式とは、ストレージ技術に特有である。
【０１２８】
後処理ステップの目的の１つは、この管理情報を、ディスクイメージユーティリティへの入力ファイル中で指定されたイメージレイアウトの要件に反しないような方式で、イメージに導入することである。例えば、パーティションの１つがＮＯＲフラッシュから実行されることになる場合は、これがＣＰＵページインクリメントしながらＣＰＵにマッピング可能であることを指定するＣＰＵ要件があり、これはストレージ管理要件により変更できない。
【０１２９】
ディスクイメージユーティリティ２３０ツールの後処理段階を用いて、ＭＳＰａｒｔによって生成されたＮＢ０ファイルに対してストレージハードウェア特有の調整を行う。例えば図１０Ａに表すように、ＮＡＮＤフラッシュは、セクタに付随してそのセクタ情報（各セクタごとのメタデータｍ）を別の空間に有する。しかし、ＮＯＲハードウェアは、ＩＭＧＦＳパーティションを修正して、各セクタのブロック内にＳｅｃｔｏｒＩｎｆｏを含

める必要がある。これを図１０Ｂに一般に表すが、４つのセクタを含むブロックのうち３つのセクタに関するメタデータ（ｍ）が、各ブロックの残りのセクタ中に保持されている。図１０Ｂの影付き領域で表すように、ブロックを整合させるためにいくらかの空間は使用されない場合もあることに留意されたい。ペアフラッシュ（paired flash）を有するデバイス製造業者はまた、ＮＢ０を各フラッシュ部分ごとに別々のファイルに分離したいと思う場合もある。

【０１３０】

ポストプロセッサ２３２（図２、例えばｐｏｓｔｐｒｏｃ．ｅｘｅ）は、ディスクイメージユーティリティ２３０によって、ｍｅｍｏｒｙ．ｃｆｇ．ｘｍｌ中の任意のＮＯＲフラッシュ要素上で自動的に実行されるものとすることができる。ＮＯＲ要素のＩＤ、ＶＡＳＴＡＲＴ、ＢＬＯＣＫＳＩＺＥ、ＳＥＣＴＯＲＳＩＺＥ属性が与えられれば、ポストプロセッサはＩＤ．ｎｂ０．ｐｏｓｔｐｒｏｃおよびＩＤ．ｂｉｎを出力する。ディスクイメージユーティリティ２３０の出力をさらに後処理するには、ユーザ（例えばＯＥＭ（original equipment manufacturer））がｐｏｓｔｄｉｓｋｉｍａｇｅ．ｂａｔファイルを作成することができ、このファイルは、ディスクイメージユーティリティ２３０の処理が完了したときにｂｕｉｌｄｐｋｇスクリプトによって呼び出されることになる。

【０１３１】

動作時、ポストプロセッサ２３２はＮＢ０を開き、マスタブートレコードを見つけ、その中のデータを使用してＩＭＧＦＳパーティションを突き止める。次いでポストプロセッサ２３２は、（例えばＮＯＲ）ブロックドライバが理解するフォーマットで、各セクタにセクタデータを追加する。セクタデータをＩＭＧＦＳに追加した後にはもはやＵＳＥＲＳＴＯＲＥがフラッシュブロックの開始に対応しない場合があるので、ポストプロセッサ２３２はまた、ＵＳＥＲＳＴＯＲＥが存在すればその開始を移動し、修正済みＮＢ０を保存する。ポストプロセッサ２３２はさらに、レガシーブートローダのために修正済みＮＢ０をバイナリファイルに変換することもできる。

【０１３２】

以上の詳細な記述からわかるように、オペレーティングシステムイメージコンポーネントをファイルシステムベースの製造イメージに変換する機構が提供される。このイメージファイルは、どんな特定のストレージ技術からも独立しており、初期イメージとして使用するのに適するとともに、デバイスをコンポーネント化方式で更新することを容易にする。

【０１３３】

本発明は様々な修正および代替構造が可能だが、本発明の例示的ないくつかの実施形態を図面に示し、上に詳細に述べた。ただし、開示した特定の形式に本発明を限定する意図はなく、反対に、本発明の趣旨および範囲に含まれるすべての修正、代替構造、均等物をカバーするものとすることを理解されたい。

【図面の簡単な説明】

【０１３４】

【図１】本発明を組み込むことのできるコンピュータシステムを一般に表すブロック図である。

【図２】本発明の一態様による、組込みデバイスまたはその他の媒体にイメージとして書き込まれるデータを保持するための別々のパーティションを内部に設けて構築された出力ファイルを表すブロック図である。

【図３】本発明の一態様による、出力ファイルを構成するのに使用される様々なコンポーネントを表すブロック図である。

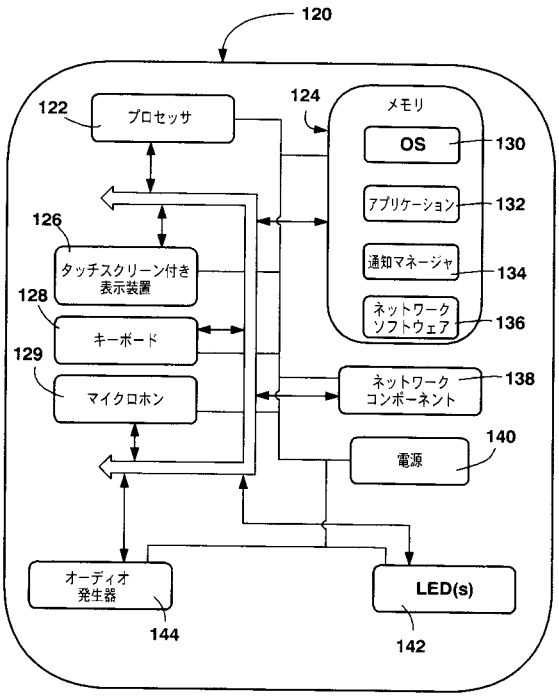
【図４】本発明の一態様による、出力ファイルを構成するディスクイメージユーティリティの全体的な流れを表す流れ図である。

【図５】本発明の一態様による、メモリ構成ファイルを妥当性検査するための論理を表す流れ図である。

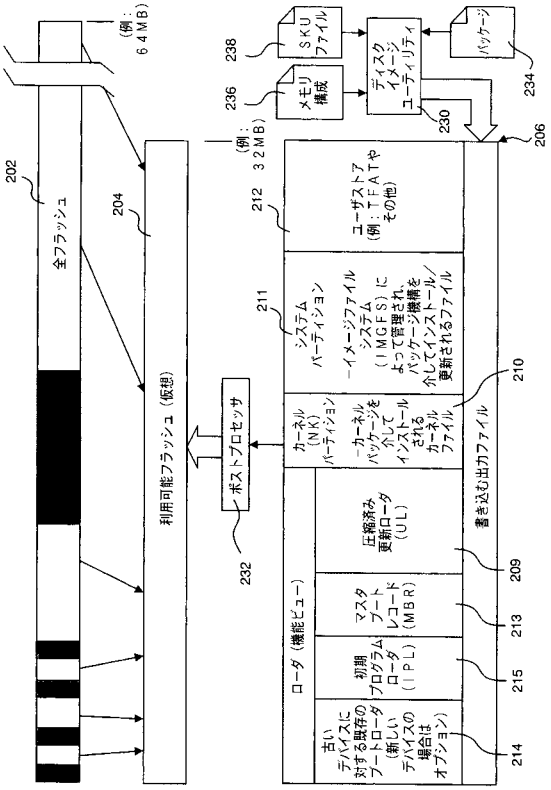
【図６】本発明の一態様による、パッケージをパーティションマッピングファイルに対し

て検証すること、および指定されたパーティションをパッケージと照合することを担うプロセスを表す流れ図である。
【図７】本発明の一態様による、フラッシュ要素の処理を表す流れ図である。
【図８】本発明の一態様による、ＲＯＭ／ＲＡＭベースのパーティションを構築するためのプロセスを表す流れ図である。
【図９】本発明の一態様による、システムパーティションを書き込むプロセスを表す流れ図である。
【図１０Ａ】本発明の一態様による、後処理によって扱われるフラッシュメモリ中でセクタおよびブロックが構成される方式を表すブロック図である。
【図１０Ｂ】本発明の一態様による、後処理によって扱われるフラッシュメモリ中でセクタおよびブロックが構成される方式を表すブロック図である。
【符号の説明】
【０１３５】
２０２　全フラッシュ
２０４　仮想メモリ
２０６　ファイル
２０９　パーティション
２１０　カーネル（ＮＫ）パーティション
２１１　システムパーティション
２１２　ユーザストアパーティション
２１３　マスタブートレコード（ＭＢＲ）
２１４　ブートローダ
２１５　初期プログラムローダ（ＩＰＬ）
２３０　ディスクイメージユーティリティ
２３２　イメージポストプロセッサ
２３６　ファイル
２３４　パッケージ
２３８　ファイル

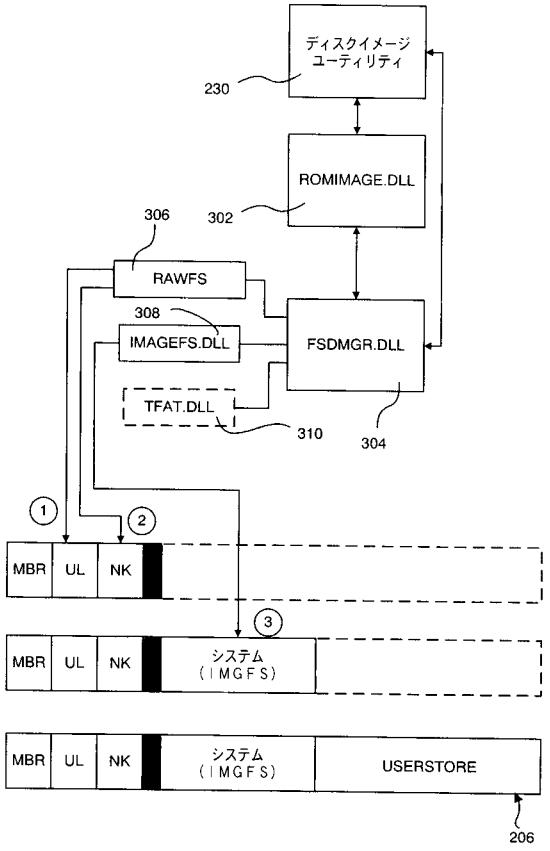
【図１】


120
プロセッサ
122
メモリ
124
OS
130
アプリケーション
132
通知マネージャ
134
ネットワークソフトウェア
136
126
タッチスクリーン付き表示装置
128
キーボード
129
マイクロホン
ネットワークコンポーネント
138
電源
140
オーディオ発生器
144
LED(s)
142

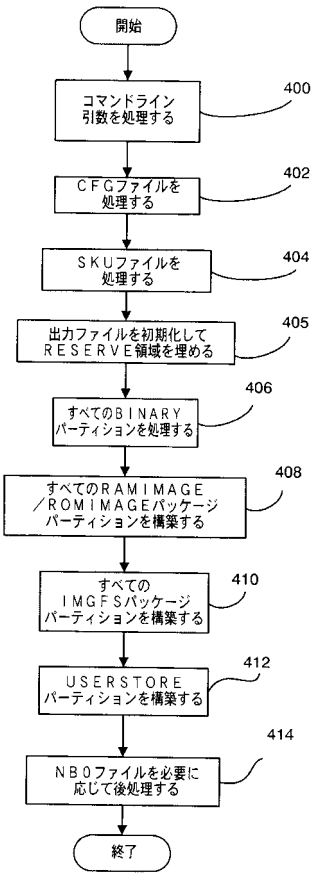
【図２】


(例：64MB)
202
全フラッシュ
204
利用可能フラッシュ（仮想）
ポストプロセッサ
232
(例：32MB)
SKUファイル
238
メモリ構成
236
ディスクイメージユーティリティ
230
パッケージ
234
206
ユーザストア（例：TFATやその他）
212
システムパーティション
211
イメージファイルシステム（IMGFS）によって管理されパッケージ機構を介してインストール／更新されるファイル
カーネル（NK）パーティション
カーネルパッケージを介してインストールされるカーネルファイル
210
ローダ（機能ビュー）
圧縮済み更新ローダ（UL）
209
マスタブートレコード（MBR）
213
初期プログラムローダ（IPL）
215
古いデバイスに対する既存のブートローダ（新しいデバイスの場合はオプション）
214
書き込む出力ファイル

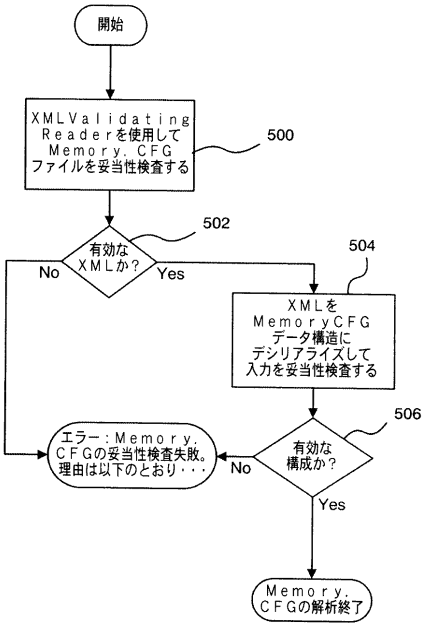
【図３】


ディスクイメージユーティリティ
230
ROMIMAGE.DLL
302
306
RAWFS
308
IMAGEFS.DLL
FSDMGR.DLL
TFAT.DLL
310
304
1
2
3
MBR
UL
NK
システム（IMGFS）
USERSTORE
206

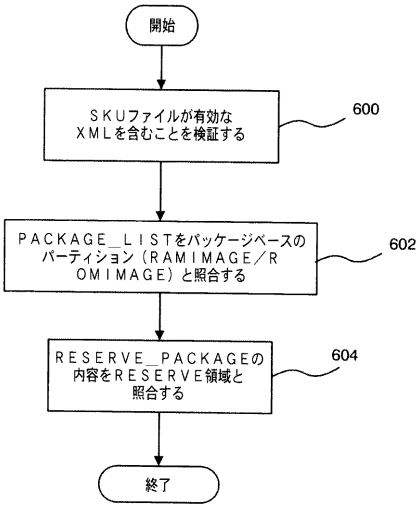
【図４】


開始
コマンドライン引数を処理する
400
CFGファイルを処理する
402
SKUファイルを処理する
404
出力ファイルを初期化してRESERVE領域を埋める
405
すべてのBINARYパーティションを処理する
406
すべてのRAMIMAGE／ROMIMAGEパッケージパーティションを構築する
408
すべてのIMGFSパッケージパーティションを構築する
410
USERSTOREパーティションを構築する
412
NBOファイルを必要に応じて後処理する
414
終了

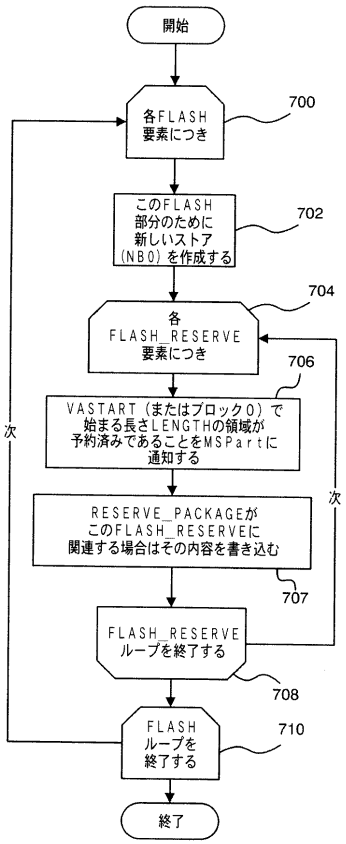
【図５】


開始
XMLValidating Readerを使用して Memory.CFG ファイルを妥当性検査する
500
有効な XMLか？
502
No
Yes
XMLを MemoryCFG データ構造に デシリアライズして 入力を妥当性検査する
504
有効な 構成か？
506
エラー：Memory.CFGの妥当性検査失敗。理由は以下のとおり・・・
No
Yes
Memory.CFGの解析終了

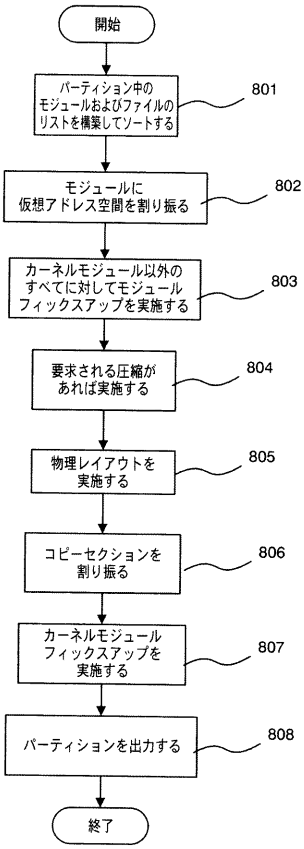
【図６】


開始
SKUファイルが有効な XMLを含むことを検証する
600
PACKAGE_LISTをパッケージベースの パーティション（RAMIMAGE／ROMIMAGE）と照合する
602
RESERVE_PACKAGEの 内容をRESERVE領域と 照合する
604
終了

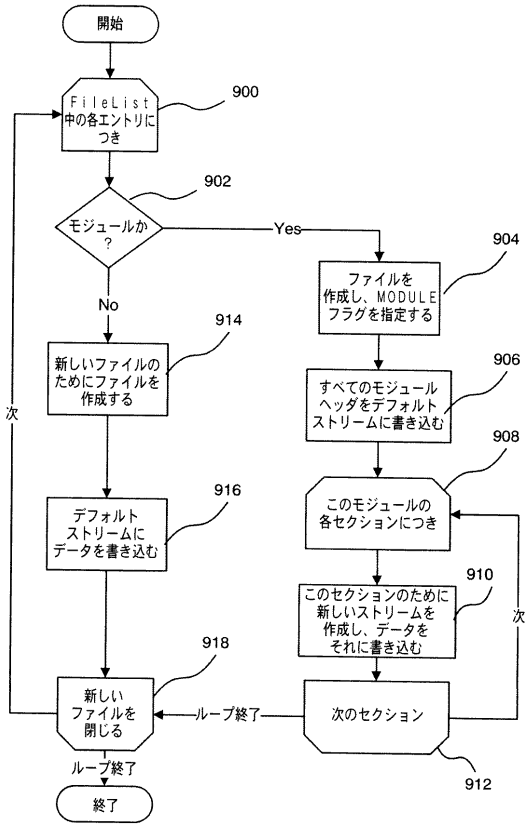
【図７】


開始
各FLASH 要素につき
700
このFLASH 部分のために 新しいストア (NBO) を作成する
702
各 FLASH_RESERVE 要素につき
704
VASTART（またはブロック0）で 始まる長さLENGTHの領域が 予約済みであることをMSPartに 通知する
706
RESERVE_PACKAGEが このFLASH_RESERVEに 関連する場合はその内容を書き込む
707
FLASH_RESERVE ループを終了する
708
FLASH ループを 終了する
710
終了
次
次

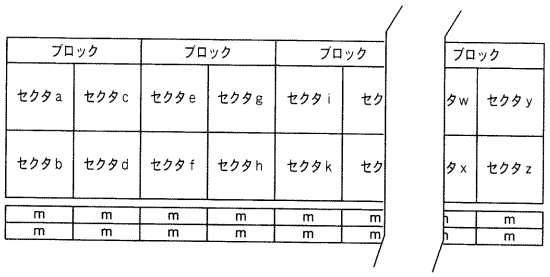
【図８】


開始
パーティション中の モジュールおよびファイルの リストを構築してソートする
801
モジュールに 仮想アドレス空間を割り振る
802
カーネルモジュール以外の すべてに対してモジュール フィックスアップを実施する
803
要求される圧縮が あれば実施する
804
物理レイアウトを 実施する
805
コピーセクションを 割り振る
806
カーネルモジュール フィックスアップを 実施する
807
パーティションを出力する
808
終了

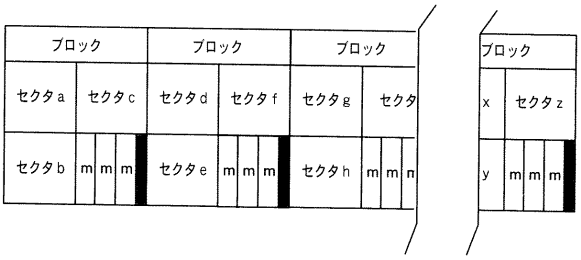
【図９】


開始
FileList中の各エントリにつき
900
902
モジュールか？
Yes
No
904
ファイルを作成し、MODULEフラグを指定する
906
すべてのモジュールヘッダをデフォルトストリームに書き込む
908
このモジュールの各セクションにつき
910
このセクションのために新しいストリームを作成し、データをそれに書き込む
912
次のセクション
914
新しいファイルのためにファイルを作成する
916
デフォルトストリームにデータを書き込む
918
新しいファイルを閉じる
ループ終了
ループ終了
次
次
終了

【図１０Ａ】


ブロック
ブロック
ブロック
ブロック
セクタa
セクタc
セクタe
セクタg
セクタi
セク
セクタb
セクタd
セクタf
セクタh
セクタk
セク
タw
セクタy
タx
セクタz
m

【図１０Ｂ】


ブロック
ブロック
ブロック
ブロック
セクタa
セクタc
セクタd
セクタf
セクタg
セクタ
x
セクタz
セクタb
セクタe
セクタh
y
m

フロントページの続き

(72)発明者　ジェフリー　ディー．グラウム
アメリカ合衆国　９８０５２　ワシントン州　レッドモンド　ワン　マイクロソフト　ウェイ　マイクロソフト　コーポレーション内
(72)発明者　マーク　スコット　トンケロヴィッツ
アメリカ合衆国　９８０５２　ワシントン州　レッドモンド　ワン　マイクロソフト　ウェイ　マイクロソフト　コーポレーション内

Ｆターム(参考) 5B076 EB01

【外国語明細書】
2005182809000001.pdf