(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2005-56364
(P2005-56364A)
(43)公開日　平成17年3月3日(2005.3.3)

| (51)Int.Cl.[7] | F I | | テーマコード（参考） |
|---|---|---|---|
| G06F 13/00 | G06F | 13/00　540A | 5C063 |
| H04N 7/08 | H04N | 7/08　Z | |
| H04N 7/081 | | | |

審査請求 未請求　請求項の数 21　ＯＬ　外国語出願　（全 60 頁）

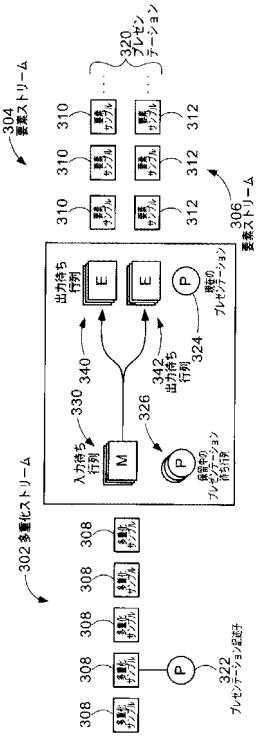
(21)出願番号　特願2003-299342(P2003-299342)
(22)出願日　平成15年8月22日(2003.8.22)
(31)優先権主張番号　10/635730
(32)優先日　平成15年8月6日(2003.8.6)
(33)優先権主張国　米国(US)

(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　カート　デビキュー
アメリカ合衆国　９８１０９　ワシントン州　シアトル　リー　ストリート　３５７

最終頁に続く

(54)【発明の名称】デマルチプレクサのアプリケーションプログラムインターフェイス

(57)【要約】
【課題】　マルチメディアデータの処理の制御を改善するためのマルチメディアデータのデマルチプレクサを代表する一連のインターフェイスとデータ構造（即ち、デマルチプレクサＡＰＩ）を提供する。
【解決手段】　このデータ構造は、各々がコマンドの要素を含むいくつかのフィールドを利用している。ひとつの実施例においては、デマルチプレクサの適切な動作のために少なくとも七つのコマンドが形成され、これらには、Initialize、SetPresentationDescriptor、GetPresentationDescriptor、GetPendingPresentationDescriptor、ProcessInput、ProcessOutput、Flushコマンドが含まれる。
【選択図】　図３



【特許請求の範囲】
【請求項１】
デマルチプレクサオブジェクトと通信するためのインターフェイスの公開に関するステップを実行するためのコンピュータ実行可能命令を有するコンピュータ読み取り可能媒体であって、前記インターフェイスは、
前記デマルチプレクサオブジェクトを構成するためのInitializeメソッドと、
前記デマルチプレクサオブジェクトに関してアクティブプレゼンテーション記述子を動的に設定するためのSetPresentationDescriptorメソッドと、
前記デマルチプレクサオブジェクトに対して新たな入力の多重化ストリームを提供するためのProcessInputメソッドと、
アクティブプレゼンテーションから少なくとも一つの要素ストリームを取り出すためのProcessOutputメソッドと、
現時点で待ち行列に入れられている入力および出力のサンプルを一括消去するためのFlushメソッドと
を備えたことを特徴とするコンピュータ読み取り可能媒体。
【請求項２】
前記インターフェイスは、前記デマルチプレクサオブジェクトに関して現時点でのアクティブプレゼンテーションのコピーを取り出すためのGetPresentationDescriptorメソッドをさらに備えたことを特徴とする請求項１に記載のコンピュータ読み取り可能媒体。
【請求項３】
前記GetPresentationDescriptorメソッドは、プレゼンテーション記述子を含むことを特徴とする請求項２に記載のコンピュータ読み取り可能媒体。
【請求項４】
前記インターフェイスは、次の保留中のプレゼンテーションを取り出すためのGetPendingPresentationDescriptorメソッドをさらに備えることを特徴とする請求項１に記載のコンピュータ読み取り可能媒体。
【請求項５】
前記GetPendingPresentationDescriptorメソッドは、保留中のプレゼンテーション記述子を含むことを特徴とする請求項４に記載のコンピュータ読み取り可能媒体。
【請求項６】
前記Initializeメソッドは、多重化ストリーム記述子、該多重化ストリーム記述子のための選択されたメディアタイプ、要素ストリームのメジャータイプの配列、および該メジャータイプの配列における前記メジャータイプの総数のパラメータを含むことを特徴する請求項１に記載のコンピュータ読み取り可能媒体。
【請求項７】
前記SetPresentationDescriptorメソッドは、プレゼンテーション記述子オブジェクトに対するポインタを含むことを特徴とする請求項１に記載のコンピュータ読み取り可能媒体。
【請求項８】
前記ProcessInputメソッドは、サンプルオブジェクトに対するポインタを含むことを特徴とする請求項１に記載のコンピュータ読み取り可能媒体。
【請求項９】
前記ProcessInputメソッドは、新たなプレゼンテーションのフラグを有する戻り値をさらに含むことを特徴する請求項８に記載のコンピュータ読み取り可能媒体。
【請求項１０】
新たなプレゼンテーションのフラグは、真の値を有する場合に、
次の保留中のプレゼンテーションを取り出すためのGetPendingPresntationDescriptorメソッドを呼び出すステップと、
所望のストリームを選択するステップと、
デマルチプレクサの入力待ち行列からのサンプルの処理を可能にするためのSetPresent

ationDescriptorメソッドを呼び出すステップと
を実行するためのコンピュータ実行可能命令をさらに備えたことを特徴とする請求項９に記載のコンピュータ読み取り可能媒体。
【請求項１１】
前記ProcessOutputメソッドは、ストリーム識別子と、サンプルオブジェクトに対するポインタに対するポインタを含むことを特徴とする請求項１に記載のコンピュータ読み取り可能媒体。
【請求項１２】
前記ProcessOutputメソッドは、出力戻り値をさらに含むことを特徴とする請求項１１に記載のコンピュータ読み取り可能媒体。
【請求項１３】
前記出力戻り値は、ストリームの終わりエラーコードおよびこれ以上のデータ無しのエラーコードのひとつを含むことを特徴とする請求項１２に記載のコンピュータ読み取り可能媒体。
【請求項１４】
前記インターフェイスは、データのインメモリバッファとして多重化されたデータを受け取ることを特徴とする請求項１に記載のコンピュータ読み取り可能媒体。
【請求項１５】
前記多重化されたデータは、デジタルビデオ（Ｄｉｇｉｔａｌ　Ｖｉｄｅｏ）、ＭＰＥＧ２、ＡＳＦの少なくとも一つを有するフォーマットを有することを特徴とする請求項１４に記載のコンピュータ読み取り可能媒体。
【請求項１６】
デマルチプレクサでの使用のためのInitializeのデータ構造をその上に保存したコンピュータ読み取り可能媒体であって、前記データ構造は、
ヘッダーを含む第１のフィールドと、
多重化ストリーム記述子を含む第２のフィールドと、
多重化ストリーム記述子の選択されたメディアタイプを含む第３のフィールドと、
要素ストリームのメジャータイプの配列を含む第４のフィールドと、
前記メジャータイプの配列における前記メジャータイプの総数を含む第５のフィールドと
を備えたことを特徴とするコンピュータ読み取り可能媒体。
【請求項１７】
デマルチプレクサでの使用のためのSetPresentationDescriptorのデータ構造をその上に保存したコンピュータ読み取り可能媒体であって、前記データ構造は、
ヘッダーを含む第１のフィールドと、
プレゼンテーション記述子を含む第２のフィールドと
を備えたことを特徴とするコンピュータ読み取り可能媒体。
【請求項１８】
デマルチプレクサでの使用のためのGetPresentationDescriptorのデータ構造をその上に保存したコンピュータ読み取り可能媒体あって、前記データ構造は、
ヘッダーを含む第１のフィールドと、
プレゼンテーション記述子を含む第２のフィールドと
を備えたことを特徴とするコンピュータ読み取り可能媒体。
【請求項１９】
デマルチプレクサでの使用のためのGetPendingPresentationDescriptorのデータ構造をその上に保存したコンピュータ読み取り可能媒体であって、前記データ構造は、
ヘッダーを含む第１のフィールドと、
保留中のプレゼンテーション記述子を含む第２のフィールドと
を備えたことを特徴とするコンピュータ読み取り可能媒体。
【請求項２０】

デマルチプレクサでの使用のためのProcessInputのデータ構造をその上に保存したコンピュータ読み取り可能媒体であって、前記データ構造は、
ヘッダーを含む第１のフィールドと、
サンプルオブジェクトに対するポインタを含む第２のフィールドと
を備えたことを特徴とするコンピュータ読み取り可能媒体。
【請求項２１】
デマルチプレクサでの使用のためのProcessOutputのデータ構造をその上に保存したコンピュータ読み取り可能媒体であって、前記データ構造は、
ヘッダーを含む第１のフィールドと、
ストリーム識別子を含む第２のフィールドと、
サンプルオブジェクトに対するポインタに対するポインタを含む第３のフィールドと
を備えたことを特徴とするコンピュータ読み取り可能媒体。
【発明の詳細な説明】
【技術分野】
【０００１】
本発明は、一般に電子的なデータ処理に関し、より詳細には、コンピューティング環境におけるマルチメディアデータの取り扱いに関する。
【背景技術】
【０００２】
デジタルに基づくマルチメディア、即ち、デジタル装置上で観賞するためのデジタル形式のビデオおよびオーディオの組み合わせは、能力および普及において急速に増大している。今日、製造されるほとんどすべての新しいパーソナルコンピュータは、何らかの形式のマルチメディアを含んでいる。カメラ、ビデオレコーダ、電話、テレビのようなデジタル製品の販売は、着実に増加している。マルチメディアはまた、インターネットの成長が着実かつ急速に持続するにつれて、インターネットの領域においてますます普及して一般的になりつつある。この成長と共に、その種のコンピュータ機器のユーザーによる、更なる性能の期待が生じるに至った。これら更なるユーザーの期待は、ハードウェアの能力に対してだけでなく、データそれ自身の処理能力に対しても拡大している。
【０００３】
これら増大する期待を満たすために、マルチメディアのアプリケーションに対して、ストリーミングとして知られる技術が開発された。ストリーミングによりデータを転送し、その結果、データを安定かつ連続的なストリームとして処理することがきる。これにより、ファイル全体を伝送する前に、データを表示したり、聴いたりしたりすることができるという利点があり、大きなマルチメディアファイルにとっては必須である。
【０００４】
当初は、このストリーミングの枠組みは、チェーンマネージャからの僅かな情報しか持たない一連のデータ処理モジュール（例、キャプチャリングフィルタ、変換フィルタ、そしてレンダリングフィルタ）から構成されていた。フィルタとも呼ばれるこれら複数のデータ処理モジュールは、どのように接続され、どのデータ形式を使用し、そしてどのように相互に制御するかに関して決定を行う。フィルタをチェーン状に接続している期間、プロトコルがデータフローの予め決められた固定のシーケンスを定義し、接続の折衝を制御する。典型的な折衝シーケンスは、インターフェイス、媒体、データ形式、アロケータ、そしてマスタークロックの順番で折衝する。データ処理チェーンは、コンピュータのシステム内で端から端までの解決策（ｅｎｄ－ｔｏ－ｅｎｄ　ｓｏｌｕｔｉｏｎ）を提供する。
【０００５】
ストリーミングの複雑性が増したことで、この業界では、ビデオおよびオーディオの処理チェーンのようにリアルタイム性の制約と共にデータを処理する処理チェーンを最適化することが必要になったという点を認識した。ひとつの解決策は、Ｍｉｃｒｏｓｏｆｔ（登録商標）ＣｏｒｐｏｒａｔｉｏｎによるＤｉｒｅｃｔＳｈｏｗ（登録商標）であり、こ

れはローカルファイルやインターネットサーバーからのマルチメディアストリームの再生、装置からのマルチメディアストリームのキャプチャ、そしてマルチメディアストリームのフォーマット変換を提供する。ＤｉｒｅｃｔＳｈｏｗ（登録商標）は、Ｗｉｎｄｏｗｓ（登録商標） Ｍｅｄｉａ Ａｕｄｉｏ、Ｗｉｎｄｏｗｓ（登録商標） Ｍｅｄｉａ Ｖｉｄｅｏ、ＭＰＥＧ、Ａｐｐｌｅ（登録商標）のＱｕｉｃｋＴｉｍｅ（登録商標）、ＡＶＩ（Ａｕｄｉｏ－Ｖｉｄｅｏ Ｉｎｔｅｒｌｅａｖｅｄ）、そしてＷＡＶ（Ｗｉｎｄｏｗｓ（登録商標） Ｗａｖｅ）のようなファイルタイプのビデオおよびオーディオのコンテンツの再生を可能にしている。ＤｉｒｅｃｔＳｈｏｗ（登録商標）は、プラグイン可能なフィルタコンポーネントのシステムを含んでいる。フィルタは、ＤｉｒｅｃｔＳｈｏｗ（登録商標）インターフェイスをサポートすると共に、データのストリームに関して、データのファイルに対する読み出し、コピー、変更、および書き込みによるオペレーションを可能とするオブジェクトである。フィルタの基本タイプは、ソースフィルタを含む。このソースフィルタは、何らかのソース、例えば、ディスク上のファイルや、衛星による送出、インターネットサーバー、あるいはＶＣＲ（ＶＴＲ）のような何らかのソースからデータを取り込み、そのデータを、フィルタの接続であるフィルタグラフの中に取り入れると共に複数のフィルタを接続する。フィルタグラフの中のこれらのフィルタは、変換フィルタ、同期ソースフィルタ、およびレンダリングフィルタ、即ち、データのフォーマットを変換する変換フィルタと、データを受信し、そしてそのデータを送信する同期ソースフィルタと、データを表示装置にレンダリングする場合のようにデータをレンダリングするレンダリングフィルタとを含む。データはまた、メディアを受け入れるいずれの場所に対してもレンダリングすることができる。ＤｉｒｅｃｔＳｈｏｗ（登録商標）に含まれるその他のタイプのフィルタには、データタイプを変更することなく効果（ｅｆｆｅｃｔ）を付加するイフェクトフィルタと、ソースデータのフォーマットを理解し、どのようにして正しいバイトを読み出すかを知っており、タイムスタンプを作成し、およびシークを実行するパーサーフィルタが含まれている。

【０００６】

動作中に、すべてのデータが多量の制御情報と共にフィルタからフィルタへと通過する。各フィルタは、他のフィルタに接続するために使われる「ピン」と呼ばれるオブジェクトを含んでいる。フィルタがピンを用いて接続されると、フィルタグラフが作成される。接続されたフィルタの集まりの概念である「フィルタグラフ」と、この接続されたフィルタの集まり、即ち、「フィルタグラフ」を制御するＤｉｒｅｃｔＳｈｏｗ（登録商標）の中で作成されるオブジェクト「Ｆｉｌｔｅｒ Ｇｒａｐｈ」との間に違いがあることに留意されたい。この「Ｆｉｌｔｅｒ Ｇｒａｐｈ」は、より正確にはフィルタグラフマネージャと呼ばれる。フィルタグラフにおけるデータの流れと接続を制御するために、ＤｉｒｅｃｔＳｈｏｗ（登録商標）はフィルタグラフマネージャを含んでいる。このフィルタグラフマネージャは、フィルタを適切な順番で接続することの保証を支援する。しかしながら、データと多くの制御情報はフィルタグラフマネージャを経由しない。フィルタは適切にリンクされなければならない。例えば、フィルタグラフマネージャはレンダリングの構成を検索し、利用可能なフィルタのタイプを決定し、与えられたデータタイプに対して適切な順番でフィルタをリンクし、適切なレンダリングフィルタを提供しなければならない。

【発明の開示】

【発明が解決しようとする課題】

【０００７】

フィルタは、大量のプログラムの再利用を可能にした一方で、これらフィルタの使用によっていくつかの予期しない問題も生じた。フィルタによって生じた問題のひとつは、誕生したフィルタの数多くのＡＰＩである。各フィルタは本質的に別個のＡＰＩを有している。結果として、ある特定のフィルタは、このフィルタが付随することになるあらゆるフィルタのＡＰＩにインターフェイス接続する能力がなければならない。また、フィルタの使用によって、ある特定のフィルタを、問題のある状態でシャットダウンするという問題

が生じる。グラフの中のある特定のフィルタがシャットダウンされると、シャットダウンされたフィルタとインターフェイスしているどのフィルタも別の関連のインターフェイスを必要とする。一般に、インターフェイスの喪失を上品に取り扱うようにフィルタをプログラミングすることは、インターフェイスが喪失した時のフィルタの状態を知り得ないので、困難である。インターフェイスの喪失は、それゆえ、フィルタにおける予測しない振る舞いにつながり、究極的には異常な振る舞いをするプログラムにつながる傾向がある。さらに、ＤｉｒｅｃｔＳｈｏｗ（登録商標）における全体的な制御は、二つのブロック間に分散されている。フィルタマネージャがフィルタの生成と除去（ｉｎｓｔａｎｔｉａｔｉｏｎ ａｎｄ ｒｅｍｏｖａｌ）を制御している間、フィルタ間のインターフェイスはデータの流れを制御する。このように制御を分散することは、必然的にブロック間の境界を横断する何らかの制御機能が存在するので、ソフトウェア設計を厄介なものにする。ＤｉｒｅｃｔＳｈｏｗに伴うもうひとつの問題は、フィルタがメディアフォーマットの折衝とバッファ管理機能の責務を担っていることである。フィルタはこのタスクを遂行するために他のフィルタと通信を行う。フィルタに関するこの依存性により、ＤｉｒｅｃｔＳｈｏｗ上で構築しているアプリケーションが、フィルタの中にプログラムされている可能性のあるバグや非効率性に対して影響を受けやすくなる。このようにして、下手に書かれたフィルタは、フィルタグラフとフィルタグラフに関連付けられたアプリケーションを簡単に停止させる可能性がある。

【０００８】

このため、ＤｉｒｅｃｔＳｈｏｗ（登録商標）のアーキテクチャを改善する必要性がある。より詳細には、マルチメディアデータの処理の制御を改善する必要性がある。

【課題を解決するための手段】

【０００９】

本発明は、マルチメディアデータのデマルチプレクサを表すインターフェイス、データ構造、イベントの一式を含む。データ構造はいくつかのフィールドを利用し、各々のフィールドはコマンドの要素を含んでいる。ひとつの実施例においては、デマルチプレクサの適切な動作のために、少なくとも７つのコマンドが形成され、これらにはInitialize、SetPresentationDescriptor、GetPresentationDescriptor、GetPendingPresentationDescriptor、ProcessInput、ProcessOutput、Flushコマンドが含まれている。これらのインターフェイスは、集合的に、デマルチプレクサアプリケーションプログラミングインターフェイス（Ｄｅｍｕｘ ＡＰＩ）として知られている。このＤｅｍｕｘ ＡＰＩは、オーディオおよびビデオ（圧縮あるいは非圧縮）のような要素ストリームデータを生成するために、統一的な方法で消費者がＤＶのような多重化ストリームデータを利用できる。

【００１０】

Initializeメソッドは、デマルチプレクサオブジェクトを初期化し、構成するために使われ、このデマルチプレクサを構成するために使われるパラメータを有している。これらのパラメータは、以下のパラメータ、即ち、多重化ストリームを記述する多重化ストリーム記述子オブジェクトと、この多重化ストリーム記述子に関する選択のメディアタイプと、ユーザーがデマルチプレクサからの出力として取り出すことに関心のある要素ストリームのメジャータイプの配列と、このメジャータイプの配列におけるメジャータイプの総数を含む。

【００１１】

SetPresentationDescriptorメソッドは、デマルチプレクサオブジェクトのアクティブプレゼンテーション記述子を動的に設定するために使われる。SetPresentationDescriptorメソッドは、プレゼンテーション記述子オブジェクトに対するポインタを有している。ProcessInputメソッドは、新規入力の多重化ストリームをデマルチプレクサオブジェクトに提供するために使われ、サンプルオブジェクトに対するポインタを含む。ProcessInputメソッドは、新たなプレゼンテーションのフラグを有する戻り値を有している。この新たなプレゼンテーションのフラグが真（TRUE）の値である場合、多重化サンプルのプレゼンテーション記述子に基づいて、このプレゼンテーションは変更される。ユーザーは、次に

保留中のプレゼンテーションを取り出すためのGetPendingPresentationDescriptorメソッドを呼び出し、所望のストリームを選択し、デマルチプレクサの入力待ち行列（「キュー」とも呼ばれる）からのサンプルの処理を可能にするSetPresentationDescriptorを呼び出さなければならない。

【００１２】

ProcessOutputメソッドは、アクティブプレゼンテーションから少なくとも一つの要素ストリームを取り出すために使われる。ProcessOutputメソッドは、ストリーム識別子とサンプルオブジェクトに対するポインタへのポインタを含む。ProcessOutputメソッドは、出力戻り値をさらに含む。出力戻り値には、ストリームの終わりというエラーコードと、これ以上データがないというエラーコードを含む。

【００１３】

Flushメソッドは、現在、待ち行列中の入力および出力サンプルを一括消去するために使われる。Flushメソッドにはパラメータは必要ない。

【００１４】

GetPresentationDescriptorメソッドは、デマルチプレクサオブジェクトの現在のアクティブプレゼンテーション記述子のコピーを取り出すために使われる。GetPresentationDescriptorメソッドは、プレゼンテーション記述子を含む。

【００１５】

GetPendingPresentationDescriptorメソッドは、次に保留中のプレゼンテーションを取り出すために使われる。GetPendingPresentationDescriptorメソッドは、保留中のプレゼンテーション記述子を含む。

【００１６】

本発明の更なる特徴および利点は、添付の図面を参照して説明する例示の実施形態の詳細な説明から明らかとなるであろう。

【００１７】

特許請求の範囲は、個々に本発明の特徴を述べているけれども、本発明は、その目的および利点と共に、添付の図面を参照して行われる詳細な説明から最も良く理解されよう。

【発明を実施するための最良の形態】

【００１８】

本発明は、マルチメディアデータのデマルチプレクサを表すインターフェイス、データ構造、イベントの一式、即ち、集合的にデマルチプレクサＡＰＩと呼ばれる一式を提供する。このＡＰＩにより、オーディオおよびビデオ（圧縮あるいは非圧縮）のような要素ストリームデータを生成し、統一的な方法で消費者はＤＶや、ＭＰＥＧ２、ＡＳＦなどのような多重化ストリームデータを利用することができる。このデマルチプレクサＡＰＩは、多重化ストリームの初期化データに基づく新たなストリーム記述子の動的な生成をサポートし、多重化ストリームサンプルに基づく新たなストリーム記述子の動的な生成をサポートする。さらに、このデマルチプレクサＡＰＩは、多重化ストリームについてのメタデータを経由しての帯域外の初期化、あるいは多重化ストリームサンプルを経由しての帯域内の初期化のいずれか一方をサポートし、複数のバッファにまたがることのある、デマルチプレクスされたサンプルの生成をサポートする。Ｍｉｃｒｏｓｏｆｔ（登録商標）Ｃｏｒｐｏｒａｔｉｏｎによるメディアファンデーション（Ｍｅｄｉａ　Ｆｏｕｎｄａｔｉｏｎ）アーキテクチャと関連（ｃｏｎｔｅｘｔ）して、このデマルチプレクサＡＰＩは、以下に説明するメディアプロセッサによって、主に制御され、メディアプロセッサに出力を受け渡す。デマルチプレクサＡＰＩは、いずれのマルチメディアアーキテクチャがデマルチプレクサを明確な方法で利用することできるように設計されている。

【００１９】

図面を参照すると、これらの図面では同様の番号が同様の要素を示すが、本発明が、適切なコンピューティング環境において実装されているように図示されている。要求されるものではないが、本発明は、パーソナルコンピュータによって実行される、プログラムモジュールのような、コンピュータで実行可能な命令という一般的な状況（ｃｏｎｔｅｘｔ

）において説明されるであろう。一般に、プログラムモジュールは、特定のタスクを実行したり、特定の抽象データタイプを実装する、ルーチン、プログラム、オブジェクト、コンポーネント、データ構造を含む。さらに、当業者は、本発明が、携帯型の端末、マルチプロセッサシステム、マイクロプロセッサベースあるいはプログラム可能な家電製品、ネットワークＰＣ、ミニコンピュータ、メインフレームコンピュータ、その他同種類のものを含む、他のコンピュータシステムの構成で実施可能であることを認めるであろう。本発明は、また、通信ネットワークを通してリンクされている遠隔処理装置によってタスクが実行される分散型コンピューティング環境においても実施することができる。分散型コンピューティング環境においては、プログラムモジュールは、ローカルおよびリモートの両方のメモリ記憶装置に配置されていることもある。

【００２０】

図１は、本発明がその上で実施することができる適切なコンピューティングシステム環境１００の例を図示している。このコンピューティングシステム環境１００は、適切なコンピューティング環境１００の単なるひとつの例であり、本発明の使用の範囲または機能について何らの限定を示唆するように意図されたものではない。コンピューティング環境１００は、例示の動作環境１００で図示された構成要素のどのひとつまたは組み合わせに関連してどのような依存性または要求を有するものとして解釈されるべきではない。

【００２１】

本発明は、数多くのその他の汎用あるいは特定目的のコンピューティングシステム環境あるいは構成で動作することができる。よく知られたコンピューティングシステム、環境および／または構成の例、即ち、本発明と共に使用することに適する可能性のある例には、これらに限定されるものではないが、パーソナルコンピュータ、サーバーコンピュータ、携帯型またはラップトップ式の装置、タブレット型装置、マルチプロセッサシステム、マイクロプロセッサベースのシステム、セットトップボックス、プログラム可能な家電製品、ネットワークＰＣ、ミニコンピュータ、メインフレームコンピュータ、任意の上記システムまたは装置およびその他同種類のものを含む分散型コンピュータ環境が含まれる。

【００２２】

本発明は、コンピュータによって実行される、プログラムモジュールのような、コンピュータで実行可能な命令という一般的な状況（ｃｏｎｔｅｘｔ）において説明されることがある。一般に、プログラムモジュールには、特定のタスクを実行したり、特定の抽象データタイプを実装したりする、ルーチン、プログラム、オブジェクト、コンポーネント、データ構造などが含まれる。本発明は、また、通信ネットワークを通してリンクされている遠隔処理装置によってタスクが実行される分散型コンピューティング環境において実施することができる。分散型コンピューティング環境においては、プログラムモジュールは、メモリ記憶装置を含むローカルおよび／または遠隔のコンピュータ記憶媒体に配置されていることもある。

【００２３】

図１を参照して、本発明を実施するための例示的なシステムは、コンピュータ１１０の形態で汎用コンピューティング装置を含む。コンピュータ１１０の構成要素は、これらに限定されないが、処理ユニット１２０、システムメモリ１３０、そしてシステムメモリから処理ユニット１２０を含む様々なシステム構成要素を連結するシステムバス１２１を含むことができる。システムバス１２１は、メモリバスまたはメモリコントローラ、周辺バス、および様々なバスアーキテクチャのいずれかを用いたローカルバスを含む何種類かのバス構造のいずれかとすることができる。限定ではなく、例示として、この種のアーキテクチャには、ＩＳＡ（Ｉｎｄｕｓｔｒｙ Ｓｔａｎｄａｒｄ Ａｒｃｈｉｔｅｃｔｕｒｅ）バス、ＭＣＡ（Ｍｉｃｒｏ Ｃｈａｎｎｅｌ Ａｒｃｈｉｔｅｃｔｕｒｅ）バス、拡張ＩＳＡ（ＥＩＳＡ）バス、ＶＥＳＡ（Ｖｉｄｅｏ Ｅｌｅｃｔｒｏｎｉｃｓ Ｓｔａｎｄａｒｄｓ Ａｓｓｏｃｉａｔｉｏｎ）ローカルバス、そしてメザニンバスとしても知られるＰＣＩ（Ｐｅｒｉｐｈｅｒａｌ Ｃｏｍｐｏｎｅｎｔ Ｉｎｔｅｒｃｏｎｎｅｃｔ）バスが含まれる。

【００２４】
コンピュータ１１０は、典型的には種々のコンピュータ読み取り可能媒体を含む。コンピュータ読み取り可能媒体は、コンピュータ１１０でアクセスでき、そして揮発性および不揮発性媒体と、取り外し可能および取り外し不可の媒体の双方とを含む、利用可能ないずれの媒体とすることができる。限定ではなく、例示として、コンピュータ読み取り可能媒体は、コンピュータ記憶媒体および通信媒体を有してもよい。コンピュータ記憶媒体には、コンピュータ読み取り可能命令、データ構造、プログラムモジュールまたはその他のデータのような情報の保存のためのいずれかの方法または技術で実装される揮発性および不揮発性、取り外し可能および取り外し不可の媒体が含まれる。コンピュータ記憶媒体には、これらに限定されないが、ＲＡＭ、ＲＯＭ、ＥＥＰＲＯＭ、フラッシュメモリまたはその他のメモリ技術、ＣＤ－ＲＯＭ、デジタル多用途ディスク（ＤＶＤ）またはその他の光ディスク記憶装置、磁気カセット、磁気テープ、磁気ディスク記憶装置またはその他の磁気記憶装置、あるいは所望の情報を保存するために使用することができ、コンピュータ１１０によってアクセスできるいずれかのその他の媒体が含まれる。通信媒体は、コンピュータ読み取り可能命令、データ構造、プログラムモジュールあるいはその他のデータを搬送波または他のトランスポート機構のような変調されたデータ信号中に組み込むのが通常であり、およびいずれの情報配信媒体を含む。この「変調されたデータ信号」という用語は、一つ以上の特徴量（ｉｔｓ　ｃｈａｒａｃｔｅｒｉｓｔｉｃｓ　ｓｅｔ）を有する信号、あるいは情報を信号中で符号化するような方法で変更された信号を意味する。限定ではなく、例示として、通信媒体には、有線ネットワークまたは直接の有線接続（ｄｉｒｅｃｔ－ｗｉｒｅｄ　ｃｏｎｎｅｃｔｉｏｎ）のような有線媒体、そして音響、ＲＦ、赤外線およびその他の無線媒体のような無線媒体が含まれる。上記のいずれの組み合わせも、コンピュータ読み取り可能媒体の範囲内に含まれるべきである。

【００２５】
システムメモリ１３０は、読み出し専用メモリ（ＲＯＭ）１３１およびランダムアクセスメモリ（ＲＡＭ）１３２のような揮発性および／または不揮発性メモリの形態のコンピュータ記憶媒体を含む。スタートアップの期間などに、コンピュータ１１０内で要素間の情報を転送する助けとなる基本ルーチンを格納している基本入出力システム１３３（ＢＩＯＳ）は、通常、ＲＯＭ１３１に保存されている。ＲＡＭ１３２は、通常、即座にアクセスでき、そして／あるいは、処理ユニット１２０によって現時点で処理させられているデータおよび／またはプログラムモジュールを格納している。限定ではなく、例示として、図１は、オペレーティングシステム１３４、アプリケーションプログラム１３５、その他のプログラムモジュール１３６、そしてプログラムデータ１３７を図示している。

【００２６】
コンピュータ１１０は、また、その他の取り外し可能／取り外し不可の、揮発性／不揮発性のコンピュータ記憶媒体を有することもある。単に例示のために、図１には、取り外し不可の、不揮発性の磁気媒体から読み出したり、書き込みをするハードディスクドライブ１４１や、取り外し可能な、不揮発性の磁気ディスク１５２から読み出したり、書き込みをする磁気ディスクドライブ１５１や、ＣＤ－ＲＯＭやその他の光媒体のような取り外し可能な、不揮発性の光ディスク１５６から読み出したり、書き込みをする光ディスクドライブ１５５を図示している。例示的なオペレーティング環境において使用することができる他の取り外し可能／取り外し不可の、揮発性／不揮発性のコンピュータ記憶媒体には、これらに限定されないが、磁気テープカセット、フラッシュメモリカード、デジタル多用途ディスク、デジタルビデオテープ、固体（ＩＣ）のＲＡＭ、固体（ＩＣ）のＲＯＭ、その他同種類のものが含まれる。ハードディスクドライブ１４１は、通常、インターフェイス１４０のような取り外し不可のメモリインターフェイスを通してシステムバスに接続され、磁気ディスクドライブ１５１および光ディスクドライブ１５５は、通常、インターフェイス１５０のような取り外し可能なメモリインターフェイスによってシステムバス１２１に接続される。

【００２７】

上記で説明され、図１に図示された、これらドライブおよびそれらに付属するコンピュータ記憶媒体は、コンピュータ読み取り可能命令、データ構造、プログラムモジュール、およびコンピュータ１１０のその他のデータの保管をする。図１において、例えば、ハードディスクドライブ１４１は、オペレーティングシステム１４４、アプリケーションプログラム１４５、その他のプログラムモジュール１４６、およびプログラムデータ１４７を格納するように図示されている。これらの構成要素は、オペレーティングシステム１３４、アプリケーションプログラム１３５、その他のプログラムモジュール１３６、およびプログラムデータ１３７と同一であっても、異なっていても良いということに留意されたい。オペレーティングシステム１４４、アプリケーションプログラム１４５、その他のプログラムモジュール１４６、およびプログラムデータ１４７には、これらが少なくとも異なるコピーであることをここに図示するために異なる数字が与えられている。ユーザーは、キーボード１６２、一般にマウスと呼ばれるポインティング装置１６１、トラックボールやタッチパッド、マイクロフォン１６３、そしてタブレットや電子デジタイザ１６４のような入力装置を通してコンピュータ１１０に対してコマンドおよび情報を入力することができる。その他の入力装置（図示せず）には、ジョイスティック、ゲームパッド、衛星受信アンテナ、スキャナ、またはその種の他のものを含んでもよい。これらおよびその他の入力装置は、システムバスに結合されるユーザー入力インターフェイス１６０を通して処理ユニット１２０に接続されることが多いが、パラレルポートや、ゲームポート、ユニバーサルシリアルバス（ＵＳＢ）のようなその他のインターフェイスおよびバス構造によって接続されることもある。モニター１９１やその他の種類の表示装置もまた、ビデオインターフェイス１９０のようなインターフェイス経由でシステムバス１２１に接続される。モニター１９１はまた、タッチスクリーンパネルまたはその種の他のものと一体化されていることもある。モニターおよび／またはタッチスクリーンパネルは、タブレット型のパーソナルコンピュータのような、コンピューティング装置１１０がその中に組み込まれている筐体に物理的に結合可能であることに留意されたい。さらに、コンピューティング装置１１０のようなコンピュータは、出力周辺装置インターフェイス１９５またはその種の他のものを通して接続されることのあるスピーカー１９７およびプリンタ１９６のようなその他の周辺出力装置を含んでもよい。

【００２８】

コンピュータ１１０は、遠隔コンピュータ１８０のような、一つ以上の遠隔コンピュータに論理コネクションを用いてネットワーク化された環境において動作することができる。遠隔コンピュータ１８０は、パーソナルコンピュータ、サーバー、ルータ、ネットワークＰＣ、ピア装置あるいはその他の共通ネットワークノードとすることができ、単にメモリ記憶装置１８１だけが図１に図示されているが、コンピュータ１１０に関連して上述した要素の多くあるいはすべてを含むのが通常である。図１に示された論理コネクションには、ローカルエリアネットワーク（ＬＡＮ）１７１およびワイドエリアネットワーク（ＷＡＮ）１７３が含まれているが、その他のネットワークを含んでもよい。この種のネットワーキング環境は、オフィスや、企業全体のコンピュータネットワーク、イントラネットおよびインターネットにおいて、一般的である。例えば、コンピュータシステム１１０は、データがそこから移転されるソースマシーンを含むことができ、遠隔コンピュータ１８０は、デスティネーションマシーンを有することができる。ソースおよびデスティネーションマシーンは、しかしながら、ネットワークまたは何らかのその他の手段によって接続されている必要はなく、その代わりに、データをソースのプラットフォームによって書き込むことができて、一つのあるいは複数のデスティネーションのプラットフォームによって読み出すことができる任意の媒体を経由してデータを移転することもできることに留意されたい。

【００２９】

ＬＡＮネットワーキング環境で使用される場合、コンピュータ１１０は、ネットワークインターフェイスあるいはアダプタ１７０を通してＬＡＮ１７１に接続される。ＷＡＮネットワーキング環境で使用される場合、コンピュータ１１０は、インターネットのような

ＷＡＮ１７３に渡って通信を確立するためにモデム１７２またはその他の手段を含むのが典型的である。モデム１７２は、内蔵、または、外付けとすることができ、ユーザー入力インターフェイス１６０経由で、あるいはその他の適切な機構で、システムバス１２１に接続することができる。ネットワーク化された環境において、プログラムモジュールは、コンピュータ１１０に関連して示すプログラムモジュール、あるいはそれらの部分は、遠隔メモリ記憶装置に保管することができる。限定ではなく、例示として、図１は、リモートアプリケーションプログラムをメモリ装置１８１上に常駐するものとして図示している。示されたネットワーク接続は例示であり、これらコンピュータ間の通信リンクを確立するその他の手段を使用することができることが理解されるであろう。

【００３０】

以下の説明においては、本発明は、別途明示されない限り、一つ以上のコンピュータによって実行される動作およびオペレーションの記号表示を参照して説明されるであろう。従って、時にコンピュータによって実行されているとみなされるそのような動作およびオペレーションには、構造化された形態でデータを表す電気的信号についてのコンピュータの処理ユニットによる操作が含まれる点は理解されよう。この操作はデータを変換するか、または、データをコンピュータのメモリシステムの場所に保持し、これにより、当業者によってよく理解されている方法でコンピュータの動作を再構成し、そうでなければ変更する。データを保持するデータ構造は、データのフォーマットによって規定された特定の属性を有するメモリの物理的な場所である。しかしながら、本発明は、前述の文脈において説明されている一方で、当業者は以降に説明される各種の動作およびオペレーションがハードウェアにおいても実施されうるということ認めるであろうから、本発明が限定的であるということを意味しない。

【００３１】

図２を参照すると、本発明のデマルチプレクサは、マイクロソフトのマルチメディアアーキテクチャのひとつの実装である、メディアファンデーション（Ｍｅｄｉａ　Ｆｏｕｎｄａｔｉｏｎ）アーキテクチャにおいて動作することができる。図２はメディアファンデーションアーキテクチャにおけるデマルチプレクサを示しているけれども、本発明のデマルチプレクサＡＰＩは、その他のマルチメディアアーキテクチャにおいても使用可能である。デマルチプレクサを説明する前に、メディアファンデーションについて説明する。メディアファンデーションは、コンポーネント化されたアーキテクチャである。図のように、メディアファンデーションは、メディアファンデーションにおける機能の基本ユニットのいくつかを担うコアレイヤ２１１のコンポーネントおよび、下にあるコアコンポーネントを用いて、より汎用的なタスクの実行を担う制御レイヤ２０１を含む。

【００３２】

コアレイヤ２１１のコンポーネントには、メディアソース２１０とストリームソース２１４が含まれ、これらは、汎用的で明確に定義されたインターフェイスを通してマルチメディアデータを提供する。メディアソース２１０は、アクセスされるストリームを含む、プレゼンテーションを記述する。異なるマルチメディアのファイルタイプまたは装置からマルチメディアデータを提供するために、多くのメディアソースが実装される。コアレイヤ２１１はさらに、ブロック２０８に示された変換部を含み、これは、汎用的で明確に定義されたインターフェイスを通してマルチメディアデータに関しある種の変換オペレーションを実行する。変換の例には、コーデック、ビデオサイズ変更、オーディオ再サンプラ、統計処理装置、カラー再サンプラ、その他がある。入力としてインターリーブされたマルチメディアデータを取り込み、そしてこのデータを個々に有用なマルチメディアデータのメディアストリームに分離する、本発明のデマルチプレクサが、図２のアーキテクチャにおける変換部である。ブロック２０８はさらにマルチプレクサを含み、これは、個々のメディアストリームを取り込み、それらをインターリーブされたマルチメディアデータに結合する。マルチプレクサは、共通の明確に定義されたインターフェイスを共有し、異なるタイプのマルチメディアデータに多重化するための多くの実施例がある。コアレイヤ２１１はさらに、ストリームシンク２１２とメディアシンク２３０を含む。メディアシンク

２３０は、入力として汎用的で明確に定義されたインターフェイスを通してマルチメディアデータを受け入れる。マルチメディアデータと共に異なる機能を実行するためのメディアシンクには多くの実施例がある。例としては、マルチメディアデータをある特定のファイルタイプに書き込んだり、ビデオカードを用いてモニター上にマルチメディアデータを表示したりすることである。

【００３３】

制御レイヤ２０１のコンポーネントは、より高いレベルのタスクをより簡単な方法で実行するためにコアレイヤ２１１のコンポーネントを用いる。典型的には、制御レイヤのコンポーネントは、ある与えられたタスクに対して多くの異なるコアレイヤのコンポーネントを使うであろう。例としては、マルチメディアファイルを再生するには、メディアソースがディスクからファイルを読み出し、データを解析し、一つ以上の変換処理により圧縮のマルチメディアデータを展開し、そして一つ以上のメディアシンクがマルチメディアデータを表示することを含む。制御レイヤ２０１は、メディアストリームを受け取ったり、送ったりするためにアプリケーション２０２とやり取りを行うメディアエンジン２６０と、メディアセッション２４０と、メディアプロセッサ２２０と、メディアセッション２４０内に示されているトポロジーローダー２５０を含む。トポロジーローダー２５０は、コアレイヤのコンポーネント間でのデータフローの記述を担う、制御レイヤのコンポーネントである。制御レイヤのコンポーネントを、制御レイヤによって使用されるより原始的なコアレベルのコンポーネントへのアクセスを回避するように構成することができる。データは、システムを通して流れ、メディアソース２１０から始まり、メディアセッション２４０を通してメディアプロセッサ２２０へ、そしてメディアシンク２３０の出力に流れる。メディアプロセッサ２２０は、トポロジー中のメディアソースとその他のコンポーネントのパイプラインを駆動する。メディアセッション２４０は、いつトポロジー中のイベントが起こるかをガイドし、そしてトポロジーローダー２５０は、トポロジー中で指定されたイベントが起こることを保証する。メディアセッション２４０はまた、メディアプロセッサ２２０を構成し、メディアプロセッサによって戻されたサンプルを処理（ｃｏｎｓｕｍｅ）する。これは、メディアエンジン２６０の関連で行われ、メディアセッション２４０がメディアエンジン２６０の呼び出し元（例、アプリケーション２０２）と折衝したサンプルをメディアプロセッサ２２０からメディアシンクへ送る。トポロジー中のコンポーネントには、メディアソース２１０のコンポーネントおよびメディアシンク２３０のコンポーネントだけでなくその他のノードも含まれる。メディアファンデーションシステム２００は、ストリーミングメディアオブジェクトを接続するためのインターフェイスとレイアウトを提供する。このシステムにより、ユーザーはトポロジーのコンセプトを使用して記号による抽象化を経由して、汎用または指定のソース、変換、およびシンクオブジェクト間の接続を指定することができる。

【００３４】

図３を参照して、デマルチプレクサ３００の概要をこれから説明する。以下の説明において、コマンドが説明の中で参照される。これらのコマンドは、後述する。デマルチプレクサＡＰＩは、データのソースからデータのフォーマットを分離する。デマルチプレクサＡＰＩは、多重化されたデータをデータのインメモリバッファとして取り込み、およびデマルチプレクスのオペレーションを行う。これには、この動作を行うために多くの異なるデータソースが同一の実装のデマルチプレクサを用いることができるという望ましい効果がある。例えば、ＤＶデータを、ＤＶカメラから直接もたらすこともできるが、ハードディスク上のファイルとして保存させることもできる。この場合、多重化されたＤＶデータを生成するための二つのコードがあるが（例、カメラと通信するものと、ファイルシステムと通信するもの）、同一の実装のデマルチプレクサを使うことができる。デマルチプレクサ３００は、IMFDemultiplexerインターフェイスをサポートし、多重化ストリームをその要素ストリームの部分に分割する責任を担っている。デマルチプレクサ３００は、同期の形態で動作し（ＤＭＯと類似）、および多重化ストリーム中の変化により生じる、利用可能な要素ストリーム中の動的な変化を取り扱うことができる。デマルチプレクサは、IM

FSampleインターフェイス経由で送られるサンプルを受け入れ、複数のバッファにまたがって生成サンプルを収容する。

【００３５】

多重化ストリーム３０２は、一つ以上の要素ストリームを含む単一ストリームである。要素ストリーム３０４、３０６は、同種の要素（例、オーディオ、ビデオなど）のストリームである。多重化サンプル３０８と要素サンプル３１０、３１２の間には、必ずしも何らかの種類の一対一の対応があるわけではない。例えば、多重化サンプルごとの要素ストリームごとに完全な要素サンプルがあることもあれば、ないこともある。加えて、要素サンプルが多重化ストリーム中で正しい順番になるという特別な必要性もない。ストリームから出てきた要素サンプルが同一のタイムスタンプを共有しないこともある。ある要素ストリームは、その他の要素ストリームからオフセットがあるかもしれない。いくつかの多重化ストリームは、そのストリームの期間中に単一系列（ｓｉｎｇｌｅ　ｓｅｔ）の要素ストリームのみからなることがある。いくつかの多重化ストリームは、異なった時間に異なった系列の要素ストリームを有することもある。

【００３６】

整合された要素ストリームの各系列は、プレゼンテーション３２０と呼ばれる。各系列は、対応するプレゼンテーション記述子３２２を有している。このプレゼンテーション記述子３２２は、二つの主な目的を有する。第一に、それは各要素ストリームのメディアタイプを記述する。第二に、それはどの利用可能なストリームをデマルチプレクサ３００によって抽出するかを選択するためのメカニズムを提供する。現在のプレゼンテーション３２４（Ｔｈｅ　Ｃｕｒｒｅｎｔ　Ｐｒｅｓｅｎｔａｔｉｏｎ）は、常に、選択されたストリームと現在の出力ストリームのデータタイプを記述する。

【００３７】

デマルチプレクサ３００によってサンプルを処理する前に、多重化されたサンプルを要素サンプルに変換するために使われる分割アルゴリズムは、どのストリームが抽出されることになるかを知る必要がある。この情報は、プレゼンテーション記述子３２２の中に含まれている。ストリームが選択される前には、プレゼンテーション記述子は「保留中」である。一度、ストリームが選択されると、プレゼンテーション記述子は「アクティブ（動作中）」となる。プレゼンテーションをアクティブにさせるためには、GetPendingPresentationDescriptorを呼び出すことによって保留中プレゼンテーション待ち行列３２６（「キュー」とも呼ばれる）からプレゼンテーションを取り出す。適切なストリームを選択し、その後、呼び出し元がSetPresentationDescriptorメソッドを起動する。この時点で（ある条件が合えば）、このプレゼンテーションはアクティブのプレゼンテーション３２４になり、保留中プレゼンテーション待ち行列３２６から取り除かれる。保留中のプレゼンテーションは、前のアクティブのプレゼンテーションからすべて出力された場合にだけアクティブ可能となる。

【００３８】

デマルチプレクサ３００は、少なくとも二組の待ち行列を含んでいる。これらの待ち行列は、入力待ち行列３３０と、出力待ち行列３４０、３４２である。ProcessInput()がデマルチプレクサ３００に関して呼び出されると、この入力はすぐに処理されるか、あるいは入力待ち行列３３０に入れることができる。一旦、このデータが処理されると、それは出力待ち行列３４０、３４２に入れられる。出力待ち行列にある利用可能なデータタイプとストリームは、常に、現在のアクティブなプレゼンテーション３２４に対応している。

【００３９】

デマルチプレクサ３００の全般的な説明について述べられた今、デマルチプレクサ３００の状態と遷移について説明されるべきであろう。次に続く説明において、コマンドが説明の中で参照されるであろう。これらのコマンドを以下に説明する。ここで図４を参照する。デマルチプレクサ３００が作成されたが、Initializeがまだ呼び出されていないときに、デマルチプレクサ３００は非初期化の状態にある。Initializeは、この状態におけるデマルチプレクサオブジェクト３００に関して唯一の有効なオペレーションである。Init

ialize()を呼び出すことにより、デマルチプレクサ３００が保留中の状態４０２に遷移する。

【００４０】

保留中の状態４０２は、有効なアクティブプレゼンテーションがないことを示している。ProcessOutputに対する呼び出しは、失敗するであろう。アクティブプレゼンテーションを設定するには、メディアプロセッサ２２０がGetPendingPresentationDescriptor()を呼び出し、適切なストリームを選択し、SetPresentationDescriptor()を呼び出す。GetPendingPresentationDescriptor()に対する呼び出しが失敗に終わる場合、次にProcessInput（）が呼び出される。その後、プレゼンテーション記述子３２２を得ることができるまで、GetPendingPresentationDescriptor()が再び呼び出される。一旦、PresentationDescriptor（プレゼンテーション記述子）がデマルチプレクサ３００上で設定されると、デマルチプレクサ３００は中立（Ｎｅｕｔｒａｌ）の状態４０４に遷移する。Flush()を呼び出すことにより、すべての待ち行列中の入力および出力データを破棄し、デマルチプレクサ３００は保留中の状態４０２に遷移する。

【００４１】

中立の状態４０４において、すべての関数の呼び出しが（初期化を除いて）有効である。ProcessInputの呼び出しに関して発見された新たなプレゼンテーションがあり、現在のプレゼンテーションの最後のサンプルが出力待ち行列から出力（Ｓｅｒｖｉｃｅ）されるとき、デマルチプレクサ３００は保留中の状態４０２に遷移する。

【００４２】

いくつかのストリームは、ストリームのコンテンツに基づいて検出することができる、固定の限られた期間を有している。この条件が検出され、すべてが出力されたとき、デマルチプレクサは、end_of_streamの状態４０６に遷移する。その後のすべての呼び出しに対しては、適切なエラーコードが返される。

【００４３】

回復不能なエラーがデマルチプレクサ３００に起こると、エラー状態４０８に遷移する。エラー状態４０８へは、他のどの状態からでも達する。デマルチプレクサ３００が除去されることになると、Release()が呼び出され、デマルチプレクサ３００がシステムから除去されるのに先立って、デマルチプレクサ３００が終わりの状態４１０に遷移する。デマルチプレクサ３００が最後の参照から解放されると、その後デマルチプレクサ３００がどの状態にあるかを顧慮することなく、デマルチプレクサ３００はメモリから除去される。Release()は、非初期化状態を含む、どの状態からでも呼び出すことができる。

【００４４】

デマルチプレクサ３００の状態と遷移が説明された今、上記で参照されたコマンドをこれから説明する。これらのコマンドには、Initialize（）、SetPresentationDescriptor()、GetPresentationDescriptor()、GetPendingPresentationDescriptor()、ProcessInput()、ProcessOutput()、そしてFlush()が含まれる。

【００４５】

図５は例示のデータ構造図である。データ構造図は、本発明のデマルチプレクサＡＰＩの７つのメッセージを構成するために使われる基本メッセージデータ構造４６０を示す。明らかなように、メッセージデータ構造４６０は、いくつかのフィールド$462_{1-N}$を有している。好ましい実施例では、最初のフィールド$462_1$は、ヘッダーのために確保される。残りのフィールドは、パラメータである。

【００４６】

本発明のデータ構造に従って、Initializeコマンドが構築される。図６のデータ構造図から明らかなように、Initializeコマンド４８０は、いくつかのフィールド４８２～４９０で構築されている。これらのフィールドは、ヘッダーフィールド４８２、ストリーム記述子オブジェクトフィールド４８４、メディアタイプフィールド４８６、メジャータイプカウントフィールド４８８、そしてメジャータイプアレイフィールド４９０である。それぞれの各コマンドは、図５に図示されたものと同様の態様で構築される。各々を以下で説

明する。

【００４７】

Initialize（）メソッドは、デマルチプレクサオブジェクト３００を構成し、および初期化する。多重化ストリーム記述子は、（任意のヘッダーデータなどを含む）デマルチプレクサの状態を初期化するのに適切なメタデータを含んでもよい。このコマンドのシンタックスは、

【００４８】

【数１】

```
HRESULT Initialize(
  IMFStreamDescriptor* pMuxedStreamDescriptor,
  IMFMediaType* pSelectedMediaType,
  DWORD cMajorTypes,
  GUID* aMajorTypes,
);
```

【００４９】

である。

【００５０】

pMuxedStreamDescriptorパラメータは、入力パラメータであり、多重化ストリームを記述するストリーム記述子オブジェクトに対するポインタである。このパラメータの主目的は、ストリーム記述子上にあるかもしれない任意のメタデータをデマルチプレクサ３００が使用できるようにするためである。pSelectedMediatypeパラメータは、pMuxedStreamDescriptorのために選択されたメディアタイプを指定する入力パラメータである。これは、ProcessInput()に対する呼び出しにおいてデマルチプレクサ３００に渡されるサンプルに対応しているメディアタイプである。cMajorTypesパラメータは、aMajorType配列におけるメジャータイプの総数である入力パラメータである。このパラメータはゼロであってもよい。aMajorTypesパラメータは、入力パラメータであり、呼び出し元がデマルチプレクサ３００からの出力として取り出そうとしている要素ストリームのメジャータイプの配列である。このパラメータは、cMajorTypesがゼロに等しければ、ヌル（ＮＵＬＬ）であっても良い。この配列に見られる各々のメジャータイプのデフォルトストリームは、GetPendingPresentationDescriptor()から返されるプレゼンテーション記述子において選択される。このメソッドが成功すると、メソッドはS_OKの値を返す。このメソッドが失敗すると、エラーコードを返す。プレゼンテーションが利用可能であれば、そのときそのプレゼンテーションは、GetPendingPresentationDescriptor経由で取り出すことができる。プレゼンテーションが初期化の後に利用可能でない場合は、次にデータがProcessInput経由でデマルチプレクサに送り込まれた後に、プレゼンテーションが利用可能になる。

【００５１】

SetPresentationDescriptorメソッドは、新たなストリームの選択、即ち、呼び出し元が関心のある新たなストリームの選択を指示しているデマルチプレクサ３００についてアクティブプレゼンテーション記述子を設定する。このプレゼンテーション記述子は、GetPresentationDescriptorメソッドあるいはGetPendingPresentationDescriptorメソッド経由で生成された記述子でなければならない。このコマンドのシンタックスは、

【００５２】

【数２】

```
HRESULT SetPresentationDescriptor(
  IMFPresentationDescriptor* pPresentationDescriptor
);
```

【００５３】

である。

【００５４】

pPresentationDescriptorパラメータは、プレゼンテーション記述子オブジェクトに対するポインタである。このメソッドが成功すると、S_OKが返される。プレゼンテーション記述子が無効であると、それはMF_E_INVALID_PRESENTATIONで失敗することがある。プレゼンテーション記述子が保留中のプレゼンテーションであり、および現在のアクティブプレゼンテーションからまだ保留中の出力がある場合、それはMF_E_OUTPUT_PENDINGで失敗することがある。現在のアクティブプレゼンテーションから保留中の出力がある場合、SetPresentationDescriptor()を保留中のプレゼンテーションと共に呼び出さないかもしれない。ストリームを選択したり、非選択したりするために、SetPresentationDescriptor()をアクティブプレゼンテーションと共に呼び出してもよい。あるストリームが選択されない場合は、そのとき出力待ち行列にあるそのストリームのすべてのサンプルが失われる。新たなストリームが選択されると、そのストリームのサンプルは、いずれ利用可能になるであろう。ストリームが異なれば、入力および出力のバッファリング要求も異なるので、新たなストリームのサンプルがどの時点で利用可能になるかは個々のデマルチプレクサによるであろう。

【００５５】

GetPresentationDescriptorメソッドは、デマルチプレクサ３００の現在アクティブなプレゼンテーション記述子のコピーを取り出す。このコマンドのシンタックスは、

【００５６】

【数３】

```
HRESULT GetPresentationDescriptor(
  IMFPresentationDescriptor* ppPresentationDescriptor
);
```

である。

【００５７】

ppPresentationDescriptorは、プレゼンテーション記述子オブジェクトに対するポインタに対するポインタである。このメソッドが成功すると、S_OKが返される。失敗すると、エラーコードが返される。保留中の出力があると、そのときGetPresentationDescriptorは、その出力に対応するプレゼンテーション記述子を返す。

【００５８】

GetPendingPresentationDescriptorメソッドは、次の保留中のプレゼンテーションを取り出す。このコマンドのシンタックスは、

【００５９】

【数４】

```
HRESULT GetPendingPresentationDescriptor(
  IMFPresentationDescriptor* ppPendingPresentationDescriptor
);
```

【００６０】

である。

【００６１】

ppPendingPresentationDescriptorパラメータは、プレゼンテーション記述子オブジェクトに対するポインタに対するポインタである。このメソッドが成功すると、S_OKが返される。保留中のプレゼンテーションがなければ、メソッドは、MF_E_PRESENTATION_NOT_AVAILABLEを返す。ProcessInputが何回か呼び出されると、いくつかの保留中のプレゼンテーションが待ち行列に並んでいることがある。このメソッドは、単に次の保留中のプレゼンテーションを返すだけであろう。現在のアクティブなプレゼンテーションのすべての出力が処理されると、保留中のプレゼンテーションがSetPresentationを呼び出すことによってアクティブにされる。

【００６２】

ProcessInputメソッドは、呼び出し元が新たな入力の多重化ストリームサンプルをデマルチプレクサに提供可能にする。デマルチプレクサが、プレゼンテーション中の新たなストリームの存在を検出すると、*pfNewPresentationAvailableが真に設定される。その呼び出し元は、::GetPendingPresentationDescriptor経由で保留中のプレゼンテーション記述子を取り出すことができる。このコマンドのシンタックスは、

【００６３】

【数５】

```
HRESULT ProcessInput(
 IMFSample* pSample
 BOOL* pfNewPresentationAvailable
);
```

【００６４】

である。

【００６５】

pSampleパラメータは、サンプルオブジェクトに対するポインタである。このメソッドが成功すると、S_OKを返す。pfNewPresentationAvailableパラメータは、ProcessInputに対する呼び出しによって、新たなプレゼンテーション記述子が保留中のプレゼンテーション待ち行列に加えられることになると、真を返す。このデマルチプレクシングのオペレーションは、デマルチプレクサのProcessInputあるいはProcessOutputに対する呼び出しのどちらかによって、行うことができる。どちらの場合でも、ユーザーがProcessOutputを呼び出すか、Flush()に対する呼び出しでデータを破棄するまで、ProcessInputに対する呼び出しに基づいて、データが待ち行列に入れられる。一つのプレゼンテーションだけが、同時にアクティブ可能となる。新たなプレゼンテーションが利用可能になると、呼び出し元は、ProcessOutputを呼び出して、すべての保留中の出力をクリアし、GetPendingPresentationDescriptorを呼び出してこの新たなプレゼンテーションを取り出し、それからSetPresentationを呼び出してアクティブプレゼンテーションを設定しなければならない。

【００６６】

ProcessInputメソッドは、呼び出し元がアクティブプレゼンテーションの要素ストリームや複数のストリームを取り出し可能にする。このコマンドのシンタックスは、

【００６７】

【数６】

```
HRESULT ProcessOutput(
 DWORD dwStreamIdentifier,
 IMFSample* ppSample
);
```

【００６８】

である。

【００６９】

dwStreamIdentifierパラメータは、要求されたサンプルのアクティブプレゼンテーションのストリーム識別子を含む３２ビット値である。ppSampleパラメータは、サンプルオブジェクトに対するポインタに対するポインタである。このメソッドが成功すると、S_OKを返す。失敗すると、エラーコードを返す。ストリームの終わりに達すると、MF_E_ENDOFSTREAMエラーコードが返される。E_NO_MORE_DATAが返される場合は、処理するために利用可能なデータがない。ProcessInput()を別の多重化されたデータサンプルと共に呼び出すことにより、このエラーを軽減することができる。利用可能なデータの処理が、アクティブプレゼンテーションがもはや有効ではないという理由でブロックされると、このメソッドはMF_E_NEW_PRESENTATIONを返す。プレゼンテーションが変わるときには、利用可能な新たな系列のストリームが存在する。ユーザーは、どのストリームを抽出するかをデマルチ

プレクサ３００に指示しなければならない。GetPendingPresentationDescriptorを呼び出し、所望のストリームを選択し、そしてSetPresentationDescriptor()を呼び出すことにより、入力待ち行列の更なるサンプルの処理が可能になる。このデマルチプレクサオブジェクト３００は、必要に応じてこれらサンプルの空間を割り当てる。このデマルチプレクシングのオペレーション、デマルチプレクサのProcessInputあるいはProcessOutputに対する呼び出しのどちらかによって、行うことができる。どちらの場合でも、データは、ユーザーがProcessOutputを呼び出すか、Flush()に対する呼び出しでデータが破棄されるまで、ProcessInputに対する呼び出しに基づいて待ち行列に入れられる。

【００７０】

Flushメソッドにより、呼び出し元がデマルチプレクサ３００のすべての現在の待ち行列中の入力および出力サンプルを一括消去可能となる。Flushは、また、ActivePresentationDescriptorもクリアする。これは、アップストリームのデータが新たな場所でシークされているか、呼び出し元が単純にすべてのバッファされたデータを破棄したいときに行われる。このコマンドのシンタックスは、

【００７１】

【数７】

```
HRESULT Flush();
```

【００７２】

である。

【００７３】

このFlushコマンドには、パラメータはない。このメソッドが成功すると、S_OKを返す。失敗すると、エラーコードを返す。Flushを呼び出すことにより、デマルチプレクサ３００のすべての待ち行列中のデータのクリアはもとより、アクティブプレゼンテーション記述子もクリアする。特定のデマルチプレクサに依存して、保留中のプレゼンテーションは、GetPendingPresentationを呼び出すことによってすぐに利用可能になることもあるし、保留中のプレゼンテーションが利用可能になるまで、呼び出し元が繰り返しProcessInputを呼び出す必要があることもある。

【００７４】

デマルチプレクサ３００のコマンドが説明された今、ソース（例、アプリケーション２０２）が、多重化ストリームを受け入れるシンクについて折衝しない場合の典型的なオペレーションについて説明する。ここで図７を参照して、ソースがアプリケーションによる選択のために多重化ストリームを公開する（例、ビデオキャプチャのソースがＤＶを公開するか、または、ＴＶのソースがＭＰＥＧ２プログラムストリームを公開する）（ステップ５００）。制御レイヤ２０１は、アプリケーションがこの多重化ストリームに対するシンクについて折衝したかどうかを判断する（ステップ５０２）。アプリケーション２０２がその多重化ストリームに対するシンクについて折衝していなければ、メディアセッション２４０は、適切なデマルチプレクサをロードし、それを多重化ストリーム記述子と共に初期化する（ステップ５０４）。

【００７５】

プレゼンテーションがデマルチプレクサから利用可能であれば（ステップ５０６）、メディアセッション２４０は、要素ストリーム記述子を用いて再折衝を試みる（ステップ５０８）。トポロジーの残りの部分が構築される（ステップ５１０）。公開された要素ストリームは、ＤＭＯ（例、デコーダ）を用いてメディアプロセッサ２２０によって処理される。

【００７６】

プレゼンテーションが利用できなければ、そのときメディアセッションは、更なる情報が利用できるようになるまで、多重化ストリームに対するトポロジーを終了するためにヌルのメディアシンクを使用する（ステップ５１２）。セッションが開始され、サンプルが流れているときはいつでも、このデマルチプレクサ３００には、プレゼンテーション記述

子が利用可能になるまで、サンプルが与えられる（ステップ５１４）。これは、多重化ストリームを描写するメディアプロセッサノード上で呼び出されるIMFMediaStream::ProcessSampleを呼び出すことによって行われる。メディアプロセッサ２２０は、基礎を成している多重化ストリームに関してIMFMediaStream::ProcessSampleを呼び出す（例、ＡＶＩソースがＤＶストリームを公開）。この多重化ストリームサンプルが取り出されると、メディアプロセッサがデマルチプレクサ３００上でIMFDemultiplexer::ProcessInputを呼び出す。このProcessSampleおよびProcessInputの呼び出しは、新たなプレゼンテーションのフラグがIMFDemultiplexer::ProcessInputからの戻り値に関して真になるまで続く。メディアプロセッサ２２０は、その後、現在のトポロジーをデマルチプレクサの変更に起因して更新する必要があるというイベントを経由してメディアセッション２４０に信号を送る。メディアセッション２４０は、新たに利用可能になった要素ストリーム記述子にアクセスできるようにIMFDemultiplexer::GetCurrentPresentationを呼び出す。このプレゼンテーション記述子が利用可能になると、メディアセッションは、これらの要素ストリーム記述子を用いて再折衝することができる（ステップ５１６）。

【００７７】

マルチメディアデータストリームのデマルチプレクサＡＰＩについて説明した。デマルチプレクサＡＰＩは、マルチメディアデータのデマルチプレクサを代表するための一連のインターフェイス、データ構造およびイベントを具備している。このＡＰＩは、オーディオおよびビデオ（圧縮あるいは非圧縮）のような要素ストリームデータを生成するために、統一的な方法で消費者がＤＶのような多重化ストリームデータを使用できるようにする。このＡＰＩは、デマルチプレクサを独立したコンポーネントとして利用可能にする。このＡＰＩにより、フィルタのための数多くのＡＰＩに対する必要性を低減し、ある特定のフィルタについて、遺物となったフィルタがサポートされていないシステムにおいてそのフィルタが付随する可能性のあるあらゆるフィルタのＡＰＩにインターフェイスできるようにする必要はもはやなくなる。また、メディアプロセッサが、フィルタグラフ中のフィルタではなく、デマルチプレクサを制御しているので、このデマルチプレクサは上品にシャットダウンされうる。

【００７８】

特許、特許出願、および出版物を含む、本文に引用されたすべての参考文献は、参照により、そのすべてが本明細書に包含される。

【００７９】

本発明の原理を適用することができる多くの可能な実施例があることを考慮すると、図面と関連して本明細書で説明された実施例は、例示のためだけに意図されたものであり、発明の範囲を限定するものとして受け取られるべきではないということを認識すべきである。例えば、当業者には、ソフトウェアで示されたこの例示としての実施例の要素をハードウェアで実装し、そしてその逆も同様であり、あるいはこの例示された実施例が本発明の精神から逸脱することなく構成および細部において変更することができることが認識できよう。そのため、本明細書に記載されているような発明は、特許請求の範囲およびその均等物の範囲内に入るようなこの種のあらゆる実施態様を考慮している。

【図面の簡単な説明】

【００８０】

【図１】本発明がその上に備わっている例示のコンピュータシステムを一般的に示すブロック図である。

【図２】本発明がその中で動作する例示のコンピュータ環境を一般的に示すブロック図である。

【図３】本発明の教示に従うデマルチプレクサを示すブロック図である。

【図４】本発明のデマルチプレクサの状態遷移図である。

【図５】本発明の呼び出しの構成を可能にするデータ構造を示すデータ構造図である。

【図６】本発明の例示のメッセージの構成を示す簡易化したデータ構造図である。

【図７】アプリケーションがマルチメディアのシンクと折衝していない場合の本発明の教

示に従うデマルチプレクサをロードし、動作するために取られるステップを示すフローチャートである。

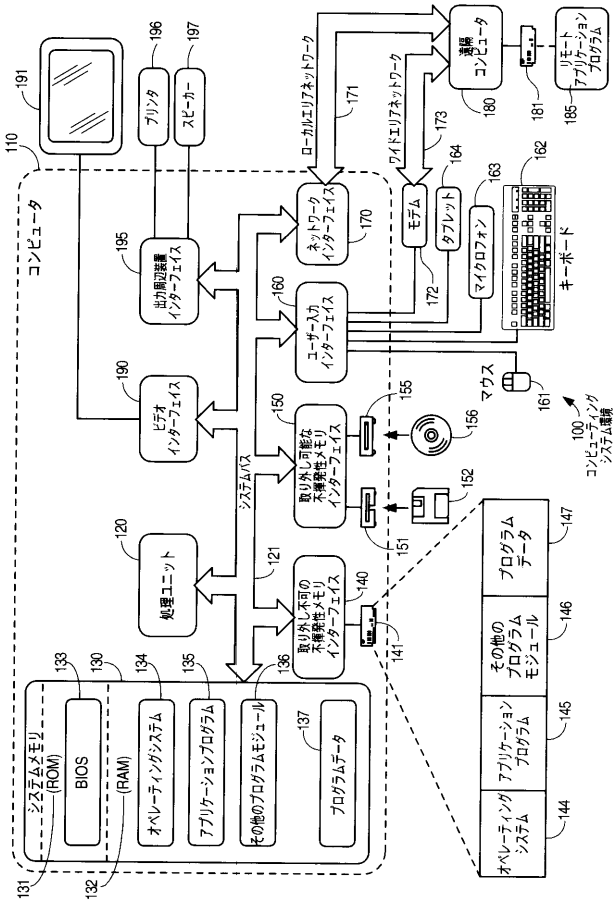
【符号の説明】

【００８１】

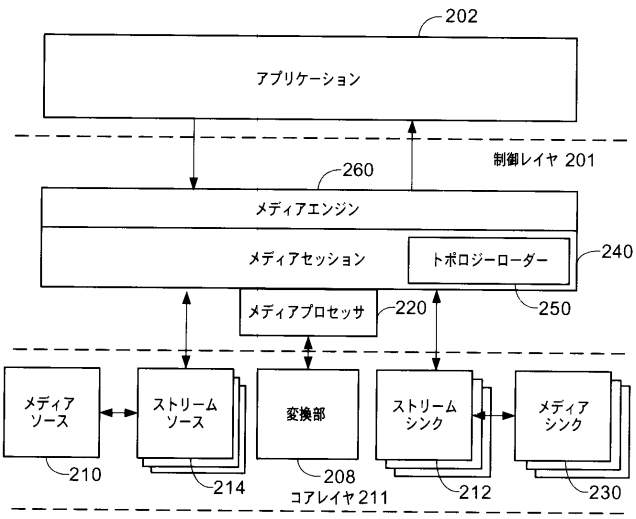
１００　コンピューティングシステム環境
１１０　コンピュータ
１２０　処理ユニット
１２１　システムバス
１３０　システムメモリ
１３１　ＲＯＭ
１３２　ＲＡＭ
１３３　ＢＩＯＳ
１３４　オペレーティングシステム
１３５　アプリケーションプログラム
１３６　その他のプログラムモジュール
１３７　プログラムデータ
１４０　取り外し不可の不揮発性メモリインターフェイス
１４１　ハードディスクドライブ
１４４　オペレーティングシステム
１４５　アプリケーションプログラム
１４６　その他のプログラムモジュール
１４７　プログラムデータ
１５０　取り外し可能な不揮発性メモリインターフェイス
１５１　磁気ディスクドライブ
１５２　取り外し可能な不揮発性磁気ディスク
１５５　光ディスクドライブ
１５６　取り外し可能な不揮発性光ディスク
１６０　ユーザー入力インターフェイス
１６１　マウス
１６２　キーボード
１６３　マイクロフォン
１６４　タブレット
１７０　ネットワークインターフェイス
１７１　ローカルエリアネットワーク
１７２　モデム
１７３　ワイドエリアネットワーク
１８０　遠隔コンピュータ
１８１　メモリ記憶装置
１８５　リモートアプリケーションプログラム
１９０　ビデオインターフェイス
１９１　モニター
１９５　出力周辺装置インターフェイス
１９６　プリンタ
１９７　スピーカー
３００　デマルチプレクサ
３０２　多重化ストリーム
３０４　要素ストリーム
３０６　要素ストリーム
３０８　多重化サンプル
３１０　要素サンプル

３１２　　要素サンプル
３２０　　プレゼンテーション
３２２　　プレゼンテーション記述子
３２４　　現在のプレゼンテーション
３２６　　保留中のプレゼンテーション待ち行列
３３０　　入力待ち行列
３４０　　出力待ち行列
３４２　　出力待ち行列

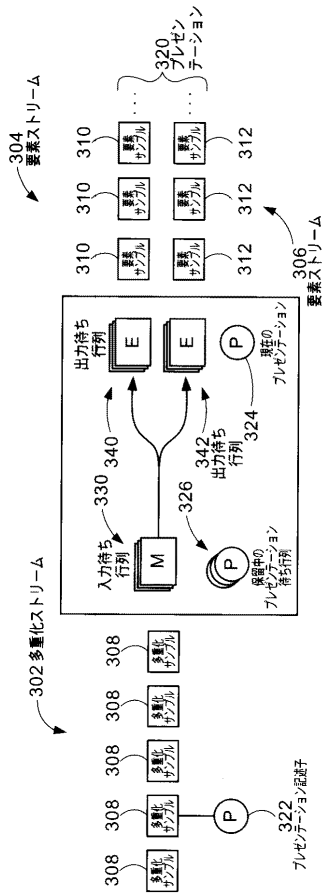
【図１】



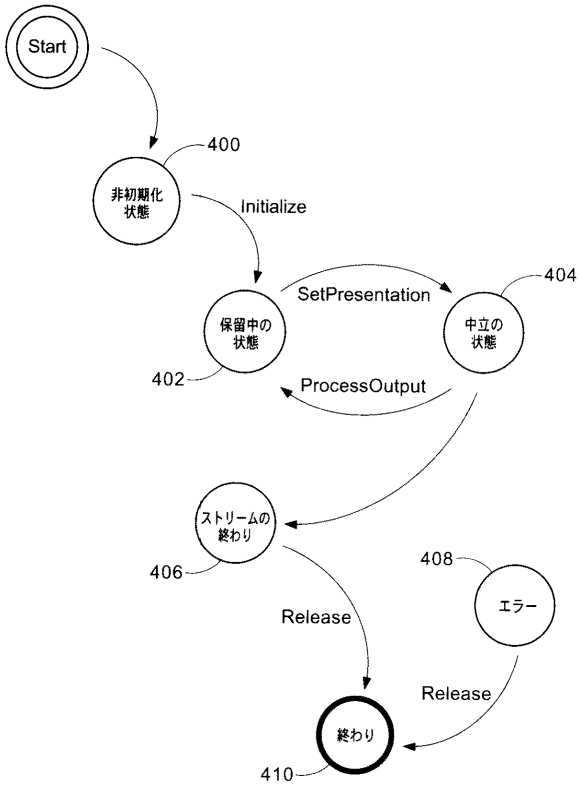

【図２】



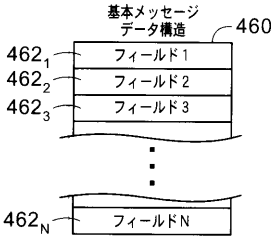
【図３】


304 要素ストリーム
320 プレゼンテーション
310
312
306 要素ストリーム
出力待ち行列
340
342 出力待ち行列
324
現在のプレゼンテーション
330
326
入力待ち行列
保留中のプレゼンテーション待ち行列
302 多重化ストリーム
308
322 プレゼンテーション記述子

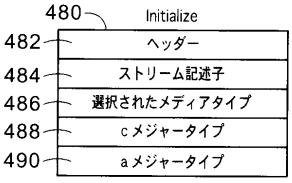
【図４】


Start
400
非初期化状態
Initialize
保留中の状態
402
SetPresentation
404
中立の状態
ProcessOutput
ストリームの終わり
406
Release
408
エラー
Release
終わり
410

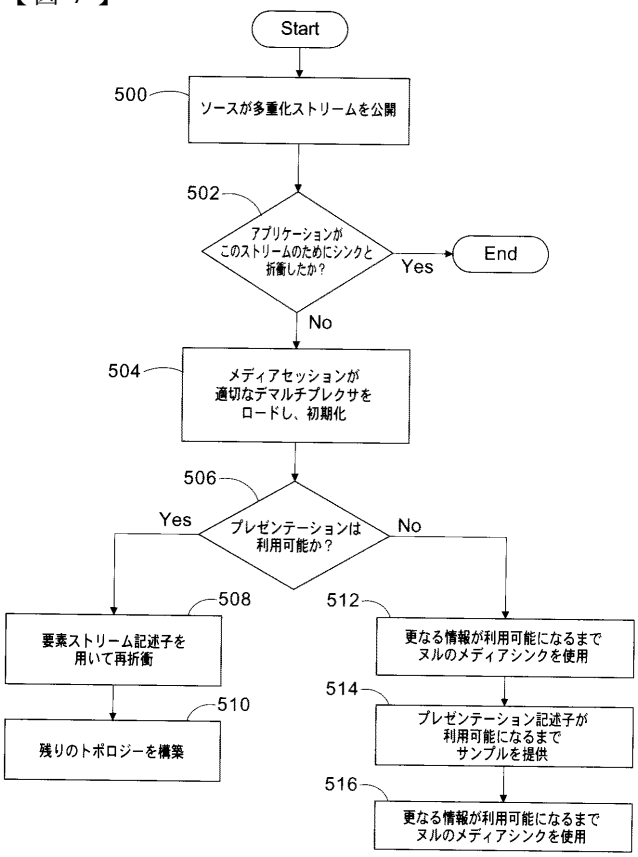
【図５】


基本メッセージデータ構造
460
462₁ フィールド1
462₂ フィールド2
462₃ フィールド3
462N フィールドN

【図６】


480
Initialize
482 ヘッダー
484 ストリーム記述子
486 選択されたメディアタイプ
488 cメジャータイプ
490 aメジャータイプ

【図７】


Start
500
ソースが多重化ストリームを公開
502
アプリケーションがこのストリームのためにシンクと折衝したか?
Yes
End
No
504
メディアセッションが適切なデマルチプレクサをロードし、初期化
506
プレゼンテーションは利用可能か?
Yes
No
508
要素ストリーム記述子を用いて再折衝
510
残りのトポロジーを構築
512
更なる情報が利用可能になるまでヌルのメディアシンクを使用
514
プレゼンテーション記述子が利用可能になるまでサンプルを提供
516
更なる情報が利用可能になるまでヌルのメディアシンクを使用

フロントページの続き

(72)発明者　ロビン　シー．ビー．スピード
イギリス　エスオー２３　９エスエヌ　ハンプシャー　ウィンチェスター　セント　ジェームズ　ビラズ　２
(72)発明者　シー．アラン　ラドウィグ
アメリカ合衆国　９８０５８　ワシントン州　レントン　サウスイースト　１６１　プレイス　１３５４７
(72)発明者　ジェフリー　ティー．ダンバー
アメリカ合衆国　９８０３３　ワシントン州　カークランド　８　アベニュー　３１８

Ｆターム(参考) 5C063 AB03 AB05 AC01 AC10 DA07 DA13

【外国語明細書】

# DEMULTIPLEXER APPLICATION PROGRAMMING INTERFACE

## FIELD OF THE INVENTION

This invention relates generally to electronic data processing and, more particularly, relates to handling multimedia data in a computing environment.

## BACKGROUND OF THE INVENTION

Digitally based multimedia, the combination of video and audio in a digital format for viewing on a digital device is rapidly increasing in capacity and proliferation. Nearly every new personal computer manufactured today includes some form of multimedia. Sales of digital products such as cameras, video recorders, phones and televisions are steadily increasing. Multimedia is also becoming increasingly prevalent in the Internet realm as the growth of the Internet steadily and rapidly continues. Along with this growth have come increased performance expectations by the users of such computer equipment. These increased user expectations extend not only to hardware capability, but also to the processing capability of the data itself.

A technique known as streaming has been developed for multimedia applications to satisfy these increasing expectations. Streaming allows data to be transferred so that it can be processed as a steady and continuous stream. This has the benefit that data can be displayed or listened to before the entire file has been transmitted, a must for large multimedia files.

Initially, the streaming framework consisted of a chain of data processing modules (e.g., capturing filters, transformation filters, and rendering filters) with very little intelligence from the chain manager. The data processing modules, also called filters, make the decisions on how to get connected, what data formats to use, and how to control each other. During connection of filters in a chain, protocols define a predefined fixed sequence of data flow and control connection negotiations. A typical negotiation sequence is to negotiate the following in order: the interface, the medium, the data format, the allocators, and the master clock. The data processing chain provides an end-to-end solution within a computer system.

As the complexity of streaming increased, the industry recognized that it became necessary to optimize the processing chains that were processing data with real-time constraints such as video and audio processing chains. One solution is DirectShow® by Microsoft® Corporation, which provides playback of multimedia streams from local files or Internet servers, capture of multimedia streams from devices, and format conversion of multimedia streams. DirectShow® enables playback of video and audio content of file types such as Windows Media Audio, Windows Media Video, MPEG, Apple® QuickTime®, Audio-Video Interleaved (AVI), and WAV (Windows Wave). DirectShow® includes a system of pluggable filter components. Filters are objects that support DirectShow® interfaces and that are capable of operating on streams of data by reading, copying, modifying and writing data to a file. The basic types of filters include a source filter, which takes the data from some source, such as a file on disk, a satellite feed, an Internet server, or a VCR, and introduces it into the filter graph,

which is a connection of filters. The filters in a filter graph include a transform filter, which converts the format of the data, a sync and source filter that receives data and transmits the data; and a rendering filter, which renders the data, such as rendering the data to a display device. The data can also be rendered to any location that accepts media. Other types of filters included in DirectShow® include effect filters, which add effects without changing the data type, and parser filters, which understand the format of the source data and know how to read the correct bytes, create times stamps, and perform seeks.

During operation, all data passes from filter to filter along with a great amount of control information. Each filter contains objects called "pins" that are used to connect to other filters. When filters are connected using the pins, a filter graph is created. Note there is a distinction between a "filter graph" which is the concept of a group of connected filters, and a "Filter Graph" which is the object you create in DirectShow® that controls the group of connected filters, the "filter graph". The "Filter Graph" is more correctly called a filter graph manager. To control the data flow and connections in a filter graph, DirectShow® includes a filter graph manager. The filter graph manager assists in assuring that filters are connected in the proper order. However, the data and much of the control do not pass through the filter graph manager. Filters must be linked appropriately. For example, the filter graph manager must search for a rendering configuration, determine the types of filters available, link the filters in the appropriate order for a given data type and provide an appropriate rendering filter.

While filters allowed a great deal of reuse of programs, the use of these filters also created some unanticipated problems. One of the problems created by the filters is the large number of API's for the filters that came into being. Each filter essentially has a separate API. As a result, a given filter must be capable of interfacing to the API for every filter to which it might attach. Also, the use of filters creates the problem of shutting down a given filter problematic. When a given filter in a graph is shut down, any filter that interfaces with the shut down filter requires a different associated interface. In general, programming a filter to gracefully handle the loss of an interface is difficult, as the state of the filter can be unknown when the interface is lost. The loss of interfaces, therefore, tends to lead to unpredicted behavior in the filters and ultimately to ill behaved programs. Further, the overall control in DirectShow® is distributed between two blocks. The interface between the filters controls the data flow while the filter manager controls the instantiation and removal of filters. Distributing the control in this manner makes software design cumbersome, as there are inevitably some control functions that cross the boundary between the blocks. Another problem with DirectShow is that the filters shoulder the responsibility of media format negotiation and buffer management functionality. Filters communicate with other filters to accomplish this task. The dependency on filters causes applications building on DirectShow to be susceptible to bugs and inefficiencies that could be programmed into a filter. Thus, a badly written filter could easily bring down the filter graph and an application associated with the filter graph.

There is a need to improve the DirectShow® architecture. More particularly, there is a need to improve control of processing of multimedia data.

## BRIEF SUMMARY OF THE INVENTION

The invention includes a set of interfaces, data structures and events for representing a demultiplexer of multimedia data. The data structure utilizes a number of fields, each containing an element of a command. In one embodiment, at least seven commands are formed for proper operation of the demultiplexer, including Initialize, SetPresentationDescriptor, GetPresentationDescriptor, GetPendingPresentationDescriptor, ProcessInput, ProcessOutput, and Flush commands. The interfaces are collectively known as the Demultiplexer Application Programming Interface (Demux API). The Demux API allows the consumer to use muxed stream data such as DV in a uniform manner to generate elementary stream data such as audio and video (compressed or uncompressed).

The Initialize method is used to initialize and configure the demultiplexer object and has parameters that are used to configure the demultiplexer. The parameters include a muxed stream descriptor object that describes the muxed stream, a selected media type for the muxed stream descriptor, an array of major types of elementary streams that the user is interested in retrieving as output from the demultiplexer, and a count of the major types in the array of major types.

The SetPresentationDescriptor method is used to dynamically set the active presentation descriptor on the demultiplexer object. The SetPresentationDescriptor method includes a pointer to a presentation descriptor object. The ProcessInput method is used to provide a new input muxed stream to the demultiplexer object and includes a pointer to a sample object. The ProcessInput method has a return value that has a new presentation flag. If the new presentation flag has a TRUE value, the presentation has changed based on the presentation descriptor in the muxed sample. The user must call GetPendingPresentationDescriptor method to retrieve the next pending presentation, select desired streams, and call the SetPresentationDescriptor method to enable processing of samples from the demultiplexer's input queue.

The ProcessOutput method is used to retrieve at least one elementary stream from an active presentation. The ProcessOutput method includes a stream identifier and a pointer to a pointer to a sample object. The ProcessOutput method further includes an output return value. The output return value includes an end of stream error code and a no more data error code.

The Flush method is used to flush currently queued input and output samples. No parameters are needed for the Flush method.

The GetPresentationDescriptor method is used to retrieve a clone of the currently active presentation descriptor on the demultiplexer object. The GetPresentationDescriptor method includes a presentation descriptor.

The GetPendingPresentationDescriptor method is used to retrieve the next pending presentation. The GetPendingPresentationDescriptor method includes a pending presentation descriptor.

Additional features and advantages of the invention will be made apparent from the following detailed description of illustrative embodiments which proceeds with reference to the accompanying figures.

## BRIEF DESCRIPTION OF THE DRAWINGS

While the appended claims set forth the features of the present invention with particularity, the invention, together with its objects and advantages, may be best understood from the following detailed description taken in conjunction with the accompanying drawings of which:

Figure 1 is a block diagram generally illustrating an exemplary computer system on which the present invention resides;

Figure 2 is a block diagram generally illustrating an exemplary computer environment in which the present invention operates;

Figure 3 is a block diagram illustrating a demultiplexer in accordance with the teachings of the present invention;

Figure 4 is a state transition diagram of the demultiplexer of the present invention;

Figure 5 is a data structure diagram illustrating the data structure model that allows for construction of calls of the present invention;

Figure 6 is a simplified data structure diagram illustrating the construction of an exemplary message of the present invention; and

Figure 7 is a flow chart illustrating the steps taken to load and operate the demultiplexer in accordance with the teachings of the present invention when an application does not negotiate a multi-media sink.

## DETAILED DESCRIPTION OF THE INVENTION

The invention provides a set of interfaces, data structures and events for representing a demultiplexer of multimedia data that are collectively called a demultiplexer API. The API allows the consumer to use muxed stream data such as DV, MPEG2, ASF, etc. in a uniform manner to generate elementary stream data such as audio and video (compressed or uncompressed). The demultiplexer API supports dynamic generation of new stream descriptors based on the muxed stream initialization data and supports dynamic generation of new stream descriptors based on the muxed stream samples. Additionally, the demultiplexer API supports either out of band initialization via metadata about the muxed stream or in band initialization via muxed stream samples and supports generation of demultiplexed samples which can span multiple buffers. In the context of the Media Foundation architecture by Microsoft® Corporation, the demultiplexer API is primarily controlled by, and output delivered to, a media processor as described herein below. The demultiplexer API is designed such that any multimedia architecture should be able to use a demultiplexer in a well-defined manner.

Turning to the drawings, wherein like reference numerals refer to like elements, the invention is illustrated as being implemented in a suitable computing environment. Although not required, the invention will be described in the general context of computer-executable instructions, such as program modules, being executed by a personal computer. Generally, program modules include routines, programs, objects, components, data structures, etc. that perform particular tasks or implement particular abstract data types. Moreover, those skilled in the art will appreciate that the invention may be practiced with other computer system configurations, including hand-held devices, multi-processor systems, microprocessor based or programmable consumer electronics, network PCs, minicomputers, mainframe computers, and the like. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

Figure 1 illustrates an example of a suitable computing system environment 100 on which the invention may be implemented. The computing system environment 100 is only one example of a suitable computing environment and is not intended to suggest any limitation as to the scope of use or functionality of the invention. Neither should the computing environment 100 be interpreted as having any dependency or requirement relating to any one or combination of components illustrated in the exemplary operating environment 100.

The invention is operational with numerous other general purpose or special purpose computing system environments or configurations. Examples of well known computing systems, environments, and/or configurations that may be suitable for use with the invention include, but are not limited to: personal computers, server computers, hand-held or laptop devices, tablet devices, multiprocessor systems, microprocessor-based systems, set top boxes, programmable consumer electronics, network PCs, minicomputers, mainframe computers, distributed computing environments that include any of the above systems or devices, and the like.

The invention may be described in the general context of computer-executable instructions, such as program modules, being executed by a computer. Generally, program modules include routines, programs, objects, components, data structures, etc. that perform particular tasks or implement particular abstract data types. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in local and/or remote computer storage media including memory storage devices.

With reference to Figure 1, an exemplary system for implementing the invention includes a general purpose computing device in the form of a computer 110. Components of computer 110 may include, but are not limited to, a processing unit 120, a system memory 130, and a system bus 121 that couples various system components

including the system memory to the processing unit 120. The system bus 121 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. By way of example, and not limitation, such architectures include Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA) bus, Enhanced ISA (EISA) bus, Video Electronics Standards Association (VESA) local bus, and Peripheral Component Interconnect (PCI) bus also known as Mezzanine bus.

Computer 110 typically includes a variety of computer readable media. Computer readable media can be any available media that can be accessed by computer 110 and includes both volatile and nonvolatile media, and removable and non-removable media. By way of example, and not limitation, computer readable media may comprise computer storage media and communication media. Computer storage media includes volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical disk storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and which can be accessed by computer 110. Communication media typically embodies computer readable instructions, data structures, program modules or other data in a modulated data signal such as a carrier wave or other transport mechanism and includes any information

delivery media. The term "modulated data signal" means a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, RF, infrared and other wireless media. Combinations of the any of the above should also be included within the scope of computer readable media.

The system memory 130 includes computer storage media in the form of volatile and/or nonvolatile memory such as read only memory (ROM) 131 and random access memory (RAM) 132. A basic input/output system 133 (BIOS), containing the basic routines that help to transfer information between elements within computer 110, such as during start-up, is typically stored in ROM 131. RAM 132 typically contains data and/or program modules that are immediately accessible to and/or presently being operated on by processing unit 120. By way of example, and not limitation, Figure 1 illustrates operating system 134, application programs 135, other program modules 136, and program data 137.

The computer 110 may also include other removable/non-removable, volatile/nonvolatile computer storage media. By way of example only, Figure 1 illustrates a hard disk drive 141 that reads from or writes to non-removable, nonvolatile magnetic media, a magnetic disk drive 151 that reads from or writes to a removable, nonvolatile magnetic disk 152, and an optical disk drive 155 that reads from or writes to a removable, nonvolatile optical disk 156 such as a CD ROM or other optical media. Other

removable/non-removable, volatile/nonvolatile computer storage media that can be used in the exemplary operating environment include, but are not limited to, magnetic tape cassettes, flash memory cards, digital versatile disks, digital video tape, solid state RAM, solid state ROM, and the like. The hard disk drive 141 is typically connected to the system bus 121 through a non-removable memory interface such as interface 140, and magnetic disk drive 151 and optical disk drive 155 are typically connected to the system bus 121 by a removable memory interface, such as interface 150.

The drives and their associated computer storage media, discussed above and illustrated in Figure 1, provide storage of computer readable instructions, data structures, program modules and other data for the computer 110. In Figure 1, for example, hard disk drive 141 is illustrated as storing operating system 144, application programs 145, other program modules 146, and program data 147. Note that these components can either be the same as or different from operating system 134, application programs 135, other program modules 136, and program data 137. Operating system 144, application programs 145, other program modules 146, and program data 147 are given different numbers hereto illustrate that, at a minimum, they are different copies. A user may enter commands and information into the computer 110 through input devices such as a keyboard 162, a pointing device 161, commonly referred to as a mouse, trackball or touch pad, a microphone 163, and a tablet or electronic digitizer 164. Other input devices (not shown) may include a joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 120 through a user input interface 160 that is coupled to the system bus, but may be connected by other

interface and bus structures, such as a parallel port, game port or a universal serial bus (USB). A monitor 191 or other type of display device is also connected to the system bus 121 via an interface, such as a video interface 190. The monitor 191 may also be integrated with a touch-screen panel or the like. Note that the monitor and/or touch screen panel can be physically coupled to a housing in which the computing device 110 is incorporated, such as in a tablet-type personal computer. In addition, computers such as the computing device 110 may also include other peripheral output devices such as speakers 197 and printer 196, which may be connected through an output peripheral interface 194 or the like.

The computer 110 may operate in a networked environment using logical connections to one or more remote computers, such as a remote computer 180. The remote computer 180 may be a personal computer, a server, a router, a network PC, a peer device or other common network node, and typically includes many or all of the elements described above relative to the computer 110, although only a memory storage device 181 has been illustrated in Figure 1. The logical connections depicted in Figure 1 include a local area network (LAN) 171 and a wide area network (WAN) 173, but may also include other networks. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets and the Internet. For example, the computer system 110 may comprise the source machine from which data is being migrated, and the remote computer 180 may comprise the destination machine. Note however that source and destination machines need not be connected by a network or any

other means, but instead, data may be migrated via any media capable of being written by the source platform and read by the destination platform or platforms.

When used in a LAN networking environment, the computer 110 is connected to the LAN 171 through a network interface or adapter 170. When used in a WAN networking environment, the computer 110 typically includes a modem 172 or other means for establishing communications over the WAN 173, such as the Internet. The modem 172, which may be internal or external, may be connected to the system bus 121 via the user input interface 160, or other appropriate mechanism. In a networked environment, program modules depicted relative to the computer 110, or portions thereof, may be stored in the remote memory storage device. By way of example, and not limitation, Figure 1 illustrates remote application programs 185 as residing on memory device 181. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

In the description that follows, the invention will be described with reference to acts and symbolic representations of operations that are performed by one or more computers, unless indicated otherwise. As such, it will be understood that such acts and operations, which are at times referred to as being computer-executed, include the manipulation by the processing unit of the computer of electrical signals representing data in a structured form. This manipulation transforms the data or maintains it at locations in the memory system of the computer, which reconfigures or otherwise alters the operation of the computer in a manner well understood by those skilled in the art. The

data structures where data is maintained are physical locations of the memory that have particular properties defined by the format of the data. However, while the invention is being described in the foregoing context, it is not meant to be limiting as those of skill in the art will appreciate that various of the acts and operation described hereinafter may also be implemented in hardware.

Turning now to Figure 2, the demultiplexer of the present invention may operate in the Media Foundation architecture, which is an implementation of one of Microsoft's multimedia architectures. While figure 2 shows the demultiplexer in the Media Foundation architecture, it is recognized that the demultiplexer API of the invention is usable in other multimedia architectures. Prior to describing the demultiplexer, the Media Foundation shall be described. Media Foundation is a componentized architecture. As shown, Media Foundation includes core layer 211 components that are responsible for some basic unit of functionality in Media Foundation, and Control layer 201 components, responsible for performing more general tasks using the underlying Core component.

Core layer 211 components include media sources 210 and stream sources 214, which provide multimedia data through a generic, well-defined interface. The media sources 210 describe the presentation, including streams to be accessed. There are many implementations of media sources, for providing multimedia data from different multimedia file types or devices. Core layer 211 further includes transforms shown in block 208, which perform some sort of transformation operation on multimedia data

through a generic, well-defined interface. Transform examples are codecs, video resizers, audio resamplers, statistical processors, color resamplers, and others. The demultiplexer of the present invention, which takes interleaved multimedia data as an input, and separates the data into individually useful media streams of multimedia data is a transform in the architecture of Figure 2. Block 208 further includes multiplexers, which take individual media streams and combine them into interleaved multimedia data. Multiplexers share a common, well-defined interface, and there are many implementations for multiplexing into different types of multimedia data. Core layer 211 further includes stream sinks 212 and media sinks 230. Media Sinks 230 accept multimedia data as input through a generic, well-defined interface. There are many implementations of media sinks for performing different functions with multimedia data. For instance, writing the multimedia data to a given file type, or displaying the multimedia data on a monitor using a video card.

Control layer 201 components use the core layer 211 components to perform higher-level tasks in a simpler way. Typically a control layer component will use many different core layer components for a given task. For instance, playing back a multimedia file will involve a media source to read the file from disk and parse the data, one or more transforms to decompress the compressed multimedia data, and one or more media sinks to display the multimedia data. Control layer 201 includes media engine 260, which interacts with application 202 to receive and send media streams, media session 240, media processor 220 and topology loader 250, shown within media session 240. Topology loader 250 is a control layer component responsible for describing the data

flow between the core layer components. A control layer components can be configured to avoid access to the more primitive core-level components used by the control layer. Data flows through system beginning with a media source 210 flowing through the media session 240 to media processor 220 and an output at media sink 230. Media processor 220 runs a pipeline of media sources and other components in the topology. Media session 240 guides when events in a topology occur, and the topology loader 250 ensures that events prescribed in a topology occur. The media session 240 also configures the media processor 220 and consumes the samples returned by the media processor 210. It does this in the context of the media engine 260 and sends the samples from the media processor 220 to the media sinks it has negotiated with the caller of the media engine 260 (e.g., application 202). The Components in a topology include media source 210 components and media sink 230 components as well as other nodes. The media foundation system 200 provides interfaces and a layout for connecting streaming media objects. The system allows a user to specify connections between generic or specified sources, transforms, and sink objects via a symbolic abstraction using a topology concept.

Turning now to Figure 3, an overview of the demultiplexer 300 shall now be described. In the description that follows, commands will be referenced in the description. These commands are described herein below. The demultiplexer API separates the format of the data from the source of the data. The demultiplexer API takes multiplexed data as an in-memory buffer of data, and performs the demultiplex operation. This has the desired effect that many different sources of data can use the same implementation of the demultiplexer to perform this operation. For example, DV data

can come directly from a DV camera, but can also be stored as a file on a hard drive. In this case, there are two pieces of code for generating the multiplexed DV data (e.g., one that talks to the camera, another that talks to the file system), but the same demultiplexer implementation can be used. The demultiplexer 300 supports the IMFDemultiplexer interface and is responsible for splitting a muxed stream into its elementary stream parts. The demultiplexer 300 operates in a synchronous fashion (similar to a DMO), and can handle dynamic changes in the available elementary streams caused by changes in the muxed stream. It accepts and generates samples which are expressed via the IMFSample interface, and accommodates generating samples which span multiple buffers.

A muxed stream 302 is a single stream that contains more than one elementary stream. An elementary stream 304, 306 is a stream of like elements (e.g., audio, video, etc.). There isn't necessarily any kind of one-to-one correspondence between muxed samples 308 and elementary samples 310, 312. For example, there may or may not be a complete elementary sample for every elementary stream in every muxed sample. Additionally, there is no particular requirement that the elementary samples be in the correct order in the muxed stream either. The elementary samples coming out of the stream may not share the same timestamps. One elementary stream may be at an offset from the other. Some muxed streams can contain only a single set of elementary streams for the duration of the stream. Some muxed streams may have different sets of elementary streams at different times.

Each set of coordinated elementary streams is called a presentation 320. Each set has a corresponding presentation descriptor 322. This presentation descriptor 322 has two main purposes. First, it describes the media types of each elementary stream. Second, it provides a mechanism to select which of the available streams are to be extracted by the demultiplexer 300. The Current Presentation 324 always describes the selected streams and data-types of the current output streams.

Before a sample can be processed by the demultiplexer 300, a splitting algorithm used to transform multiplexed samples to elementary samples needs to know what streams are to be extracted. This information is contained in the presentation descriptor 322. Before the streams are selected, the presentation descriptor is "pending". Once the streams are selected the presentation descriptor is "active". To make a presentation active, it is retrieved from the pending presentation queue 326 by calling GetPendingPresentationDescriptor. The appropriate streams are selected and then the caller invokes the SetPresentationDescriptor method. At this point (if certain conditions are met), the presentation becomes the active presentation 324 and is removed from the pending presentation queue 326. A pending presentation can only become active if all output from the previous active presentation has been serviced.

The demultiplexer 300 contains at least two sets of queues. The queues are input queues 330 and output queues 340, 342. When ProcessInput() is called on the demultiplexer 300, the input may be immediately processed or put in the input queue 330. Once the data is processed, it is put into an output queue 340, 342. The data types and

streams available in the output queues always correspond to the current active presentation 324.

Now that the overall description of the demultiplexer 300 has been described, the states and transitions of the demultiplexer 300 shall be described. In the description that follows, commands will be referenced in the description. These commands are described herein below. Turning now to Figure 4, the demultiplexer 300 is in the uninitialized state 400 when the demultiplexer 300 has been created but Initialize has not yet been called. Initialize is the only valid operation on the demultiplexer object 300 in this state. Calling Initialize() will transition the demultiplexer 300 to the pending state 402.

The pending state 402 indicates that there is no valid active presentation. Calls to ProcessOutput will fail. To set the active presentation, the media processor 220 calls GetPendingPresentationDescriptor(), selects the appropriate streams and calls SetPresentationDescriptor(). If the call to GetPendingPresentationDescriptor() fails, then ProcessInput() is called. Then GetPendingPresentationDescriptor() is called again until a presentation descriptor 322 can be obtained. Once a PresentationDescriptor has been set on the demultiplexer 300, it will transition to the Neutral state 404. Calling Flush() will discard all queued input and output data and transition the demultiplexer 300 to the pending state 402.

In the neutral state 404, all function calls (except initialize) are valid. When there is a new presentation discovered on a ProcessInput Call and the last sample from the current presentation is serviced from the output queue, the demultiplexer 300 will transition to the pending state 402.

Some streams have fixed and finite durations that can be detected based on the stream content. When this condition is detected and all output has been serviced, the demultiplexer will transition to the end_of_stream state 406. All future calls will return an appropriate error code.

When an unrecoverable error occurs in the demultiplexer 300, it will transition to the error state 408. The error state 408 can be reached from any other state. When the demultiplexer 300 is to be removed, Release() is called and the demultiplexer 300 transitions to the end state 410 prior to the demultiplexer 300 being removed from the system. When the last reference to the demultiplexer 300 is released, then the demultiplexer 300 is removed from memory without respect to what state the demultiplexer 300 is in. Release() can be called from any state, including the uninitialized state.

Now that the states and transitions of the demultiplexer 300 have been described, the commands that have been referenced above will now be described. These commands include Initialize(), SetPresentationDescriptor(), GetPresentationDescriptor(), GetPendingPresentationDescriptor(), ProcessInput(), ProcessOutput(), and Flush().

Illustrated in Figure 5 is an exemplary data structure diagram that illustrates the basic message data structure 460 used to construct the seven messages of the demultiplexer API of the present invention. As may be seen, the message data structure 460 comprises a number of fields $462_{1-N}$. In a preferred embodiment, the first field $462_1$ is reserved for the Header. The remaining fields are parameters.

In accordance with the data structure of the present invention, the Initialize command is constructed. As may be seen from the data structure diagram of Figure 6, the Initialize command 480 is constructed from a number of fields 482-490. These fields are a header field 482, a stream descriptor object field 484, a media type field 486, a major type count field 488, and a major type array field 490. Each of the different commands are constructed in a similar fashion as illustrated in Figure 5, and descriptions of each will be provided below.

The Initialize() method configures and initializes the demultiplexer object 300. The muxed stream descriptor may contain metadata appropriate to initialize the demultiplexer state (including any header data etc.). The syntax of the command is

```
HRESULT Initialize(
 IMFStreamDescriptor* pMuxedStreamDescriptor,
 IMFMediaType* pSelectedMediaType,
 DWORD cMajorTypes,
 GUID* aMajorTypes,
);
```

The *pMuxedStreamDescriptor* parameter is an input parameter and it is a pointer to a Stream Descriptor object that describes the muxed stream. The primary purpose for this

parameter is to allow any metadata that may be on the stream descriptor to be used by the Demultiplexer 300. The *pSelectedMediatype* parameter is an input parameter that specifies the selected media type for the pMuxedStreamDescriptor. This is the media type corresponding to the samples that are passed to the Demultiplexer 300 in calls to ProcessInput(). The *cMajorTypes* parameter is an input parameter that is the count of major types in the aMajorType array. This parameter can be zero. The *aMajorTypes* parameter is an input parameter and it is an array of major types of the elementary streams that the caller is interested in retrieving as output from the demultiplexer 300. This parameter can be NULL if cMajorTypes is equal to zero. The default stream of each major type found in the array will be selected in the presentation descriptor that is returned from GetPendingPresentationDescriptor(). The method returns a S_OK value if the method succeeds. If the method fails, it returns an error code. If a presentation is available, then it can be retrieved via the GetPendingPresentationDescriptor method. If the presentation is not available after initialization, then after data has been fed to the demultiplexer via ProcessInput, a presentation may become available.

The SetPresentationDescriptor method sets the active presentation descriptor on the demultiplexer 300 indicating the new stream selection that the caller is interested in. The presentation descriptor must be a descriptor generated via the GetPresentationDescriptor method or GetPendingPresentationDescriptor method. The syntax of the command is

```
HRESULT SetPresentationDescriptor(
  IMFPresentationDescriptor* pPresentationDescriptor
);
```

The *pPresentationDescriptor* parameter is a pointer to a presentation descriptor object. If the method succeeds, it returns S_OK. It may fail with MF_E_INVALID_PRESENTATION if the presentation descriptor is invalid. It may fail with MF_E_OUTPUT_PENDING if the presentation descriptor is a pending presentation and there is still pending output from the current active presentation. SetPresentationDescriptor() may not be called with a pending presentation if there is pending output from the current active presentation. SetPresentationDescriptor() may be called with the active presentation to select or deselect streams. If a stream is deselected, then all samples from that stream that are in the output queue are lost. If a new stream is selected, then samples from that stream will become available at some future time. Because different streams have different input and output buffering requirements it will depend on the individual demultiplexer at what point the new stream's samples will be available.

The GetPresentationDescriptor method retrieves a clone of the currently active presentation descriptor on the demultiplexer 300. The syntax of the command is

```
HRESULT GetPresentationDescriptor(
  IMFPresentationDescriptor* ppPresentationDescriptor
);
```

The *ppPresentationDescriptor* is a pointer to a pointer to a presentation descriptor object. If the method succeeds, it returns S_OK. If it fails, it returns an error code. If there is pending output then GetPresentationDescriptor returns the presentation descriptor that corresponds to that output.

The GetPendingPresentationDescriptor method retrieves the next pending presentation. The syntax of the command is

**HRESULT GetPendingPresentationDescriptor(**
**IMFPresentationDescriptor*** *ppPendingPresentationDescriptor*
**);**

The *ppPendingPresentationDescriptor* parameter is a pointer to a pointer to a presentation descriptor object. If the method succeeds, it returns S_OK. If there are no pending presentations, then the method returns MF_E_PRESENTATION_NOT_AVAILABLE. If ProcessInput is called several times, there may be several pending presentations queued. This method will only return the next pending presentation. Once all output for the currently active presentation has been processed, then the pending presentation is made active by calling SetPresentation.

The ProcessInput method allows the caller to provide a new input muxed stream sample to the demultiplexer. If the demultiplexer detects the presence of new streams in the presentation then the *pfNewPresentationAvailable will be set to TRUE, and the caller can retrieve the pending presentation descriptor via ::GetPendingPresentationDescriptor. The syntax of the command is

**HRESULT ProcessInput(**
**IMFSample*** *pSample*
**BOOL*** *pfNewPresentationAvailable*
**);**

The *pSample* parameter is a pointer to a sample object. If the method succeeds, it returns S_OK. The *pfNewPresentationAvailable* parameter returns TRUE when a call to ProcessInput results in a new presentation descriptor being added to the pending presentation queue. The Demultiplexing operation may be done by the demultiplexer

either on the call to ProcessInput or ProcessOutput. In either case the data will be queued on the call to ProcessInput until the user calls ProcessOutput or the data is discarded with a call to Flush(). Only one presentation may be active at one time. When a new presentation is available the caller must clear all pending output by calling ProcessOutput, retrieve the new presentation by calling GetPendingPresentationDescriptor and then set the active presentation by calling SetPresentation.

The ProcessInput method allows the caller to retrieve an elementary stream or streams from the active presentation. The syntax of the command is

```
HRESULT ProcessOutput(
  DWORD dwStreamIdentifier,
  IMFSample* ppSample
);
```

The *dwStreamIdentifier* parameter is a 32-bit value containing the stream identifier from the active presentation for the sample requested. The *ppSample* parameter is a pointer to a pointer to a sample object. If the method succeeds, it returns S_OK. If it fails, it returns an error code. If the end of the stream has been reached, then the MF_E_ENDOFSTREAM error code will be returned. If E_NO_MORE_DATA is returned, there is no data available to process. Calling ProcessInput() with another multiplexed data sample may alleviate the error. When processing of available data is blocked because the active presentation is no longer valid, the method will return MF_E_NEW_PRESENTATION. When the presentation changes, there is a new set of streams available. The user must indicate to the demultiplexer 300 what streams are to be extracted. Calling GetPendingPresentationDescriptor, selecting the desired streams, and

calling SetPresentationDescriptor() will enable the processing of more samples from the input queue. The Demultiplexer object 300 will allocate space for the samples if required. The Demultiplexing operation may be done by the demultiplexer either on the call to ProcessInput or ProcessOutput. In either case, the data will be queued on the call to ProcessInput until the user calls ProcessOutput or the data is discarded with a call to Flush().

The Flush method allows the caller to flush all currently queued input and output samples in the demultiplexer 300. Flush also clears the ActivePresentationDescriptor. This can be done when the upstream data is seeked to a new location or the caller simply wants to discard all buffered data. The syntax of the command is

**HRESULT Flush();**

There are no parameters to the Flush command. If the method succeeds, it returns S_OK. If it fails, it returns an error code. Calling Flush clears all queued data from the demultiplexer 300 as well as clearing the active presentation descriptor. Depending on the specific demultiplexer a pending presentation may be immediately available by calling GetPendingPresentation or the caller may need to repeatedly call ProcessInput until a pending presentation becomes available.

Now that the demultiplexer 300 commands have been described, a typical operation shall be described where the source (e.g., application 202) does not negotiate a sink that accepts the muxed stream. Turning now to Figure 7, the source exposes the muxed stream for selection by the application (e.g. a video capture source exposes DV or

a TV source exposes an MPEG2 program stream) (step 500). The control layer 201 determines if the application has negotiated a sink for the muxed stream (step 502). If the application 202 does not negotiate a sink for the muxed stream, the media session 240 loads the appropriate demultiplexer and initializes it with a muxed stream descriptor (step 504).

If a presentation is available from the demultiplexer (step 506), the media session 240 tries to renegotiate using the elementary stream descriptors (step 508). The rest of the topology is built (step 510). The exposed elementary streams are processed by media processor 220 using DMOs (e.g., decoders).

If a presentation is not available, then the media session uses a NULL media sink to terminate the topology for the muxed stream until more information is available (step 512). Whenever the session is started and samples are flowing, the demultiplexer 300 is given samples until a presentation descriptor becomes available (step 514). This is done by calling the IMFMediaStream::ProcessSample is called on the media processor node that represents the muxed stream. The media processor 220 calls IMFMediaStream::ProcessSample on the underlying muxed stream e.g. AVI source exposing DV stream. When the muxed stream sample is retrieved, the Media Processor calls IMFDemultiplexer::ProcessInput on the demultiplexer 300. The ProcessSample and ProcessInput calls continue until the new presentation flag is TRUE on Return from IMFDemultiplexer::ProcessInput. The media processor 220 then signals the media session 240 via an event that the current topology needs to be updated due to the

demultiplexer change. The media session 240 calls IMFDemultiplexer::GetCurrentPresentation to get access to the newly available elementary stream descriptors. When the presentation descriptor becomes available, the media session can renegotiate using the elementary stream descriptors (step 516).

A demultiplexer API for multimedia data streams has been described. The demultiplexer API includes a set of interfaces, data structures and events for representing a demultiplexer of multimedia data. The API allows the consumer to use muxed stream data such as DV in a uniform manner to generate elementary stream data such as audio and video (compressed or uncompressed). The API allows demultiplexers to be used as an independent component. This API reduces the need for the large number of API's for filters and a given filter no longer needs to be capable of interfacing to the API for every filter to which it might attach in systems where legacy filters are not supported. Also, the demultiplexer may be shut down gracefully as the media processor is controlling the demultiplexer and not filters in the filter graph.

All of the references cited herein, including patents, patent applications, and publications, are hereby incorporated in their entireties by reference.

In view of the many possible embodiments to which the principles of this invention may be applied, it should be recognized that the embodiment described herein with respect to the drawing figures is meant to be illustrative only and should not be taken as limiting the scope of invention. For example, those of skill in the art will recognize that the elements of the illustrated embodiment shown in software may be

implemented in hardware and vice versa or that the illustrated embodiment can be modified in arrangement and detail without departing from the spirit of the invention. Therefore, the invention as described herein contemplates all such embodiments as may come within the scope of the following claims and equivalents thereof.

1. A computer-readable medium having computer-executable instructions for performing the step of exposing an interface for providing communication with a demultiplexer object, the interface including:

an Initialize method to configure the demultiplexer object;

a SetPresentationDescriptor method to dynamically set an active presentation descriptor on the demultiplexer object;

a ProcessInput method to provide a new input muxed stream to the demultiplexer object;

a ProcessOutput method to retrieve at least one elementary stream from an active presentation; and

a Flush method to flush currently queued input and output samples.

2. The computer-readable medium of claim 1 wherein the interface further comprises a GetPresentationDescriptor method to retrieve a clone of the currently active presentation descriptor on the demultiplexer object.

3. The computer-readable medium of claim 2 wherein the GetPresentationDescriptor method includes a presentation descriptor.

4. The computer-readable medium of claim 1 wherein the interface further comprises a GetPendingPresentationDescriptor method to retrieve the next pending presentation.

5. The computer-readable medium of claim 4 wherein the GetPendingPresentationDescriptor method includes a pending presentation descriptor.

6. The computer-readable medium of claim 1 wherein the Initialize method includes parameters, the parameters comprising:

a muxed stream descriptor;

a selected media type for the muxed stream descriptor;

an array of major types of elementary streams; and

a count of major types in the array of major types.

7. The computer-readable medium of claim 1 wherein the SetPresentationDescriptor method includes a pointer to a presentation descriptor object.

8. The computer-readable medium of claim 1 wherein the ProcessInput method includes a pointer to a sample object.

9. The computer-readable medium of claim 8 wherein the ProcessInput method further includes a return value having a new presentation flag.

10. The computer-readable medium of claim 9 having further computer executable instructions for performing the steps comprising:

if the new presentation flag has a TRUE value:

calling a GetPendingPresentationDescriptor method to retrieve the next pending presentation;

selecting desired streams; and

calling the SetPresentationDescriptor method to enable processing of samples from the demultiplexer's input queue.

11. The computer-readable medium of claim 1 wherein the ProcessOutput method includes a stream identifier and a pointer to a pointer to a sample object.

12. The computer-readable medium of claim 11 wherein the ProcessOutput method further includes an output return value.

13. The computer-readable medium of claim 12 wherein the output return value includes one of an end of stream error code and a no more data error code.

14. The computer-readable medium of claim 1 wherein the interface takes multiplexed data as an in-memory buffer of data.

15. The computer-readable medium of claim 14 wherein the multiplexed data has a format comprising at least one of Digital Video, MPEG2, and ASF.

16. A computer-readable medium having stored thereon an Initialize data structure for use in a demultiplexer, comprising:

a first field containing a header;

a second field containing a muxed stream descriptor;

a third field containing a selected media type of the muxed stream descriptor;

a fourth field containing an array of major types of elementary streams; and

a fifth field containing a count of major types in the array of major types.

17. A computer-readable medium having stored thereon a SetPresentationDescriptor data structure for use in a demultiplexer, comprising:

a first field containing a header; and

a second field containing a presentation descriptor.

18. A computer-readable medium having stored thereon a GetPresentationDescriptor data structure for use in a demultiplexer, comprising:

a first field containing a header; and

a second field containing a presentation descriptor.

19. A computer-readable medium having stored thereon a GetPendingPresentationDescriptor data structure for use in a demultiplexer, comprising:

a first field containing a header; and

a second field containing a pending presentation descriptor.

20. A computer-readable medium having stored thereon a ProcessInput data structure for use in a demultiplexer, comprising:

a first field containing a header; and

a second field containing a pointer to a sample object.

21. A computer-readable medium having stored thereon a ProcessOutput data structure for use in a demultiplexer, comprising:

a first field containing a header;

a second field containing a stream identifier; and

a third field containing a pointer to a point to a sample object.

## 1. Abstract

A set of interfaces and data structures (i.e., a demultiplexer API) for representing a demultiplexer of multimedia data is presented. The data structure utilizes a number of fields, each containing an element of a command. In one embodiment, at least seven commands are formed for proper operation of the demultiplexer, including Initialize, SetPresentationDescriptor, GetPresentationDescriptor, GetPendingPresentationDescriptor, ProcessInput, ProcessOutput, and Flush commands. The demultiplexer API allows the consumer to use muxed stream data such as DV in a uniform manner to generate elementary stream data such as audio and video (compressed or uncompressed) and allows demultiplexers to be used as an independent component.

## 2. Representative Drawing

FIG. 3

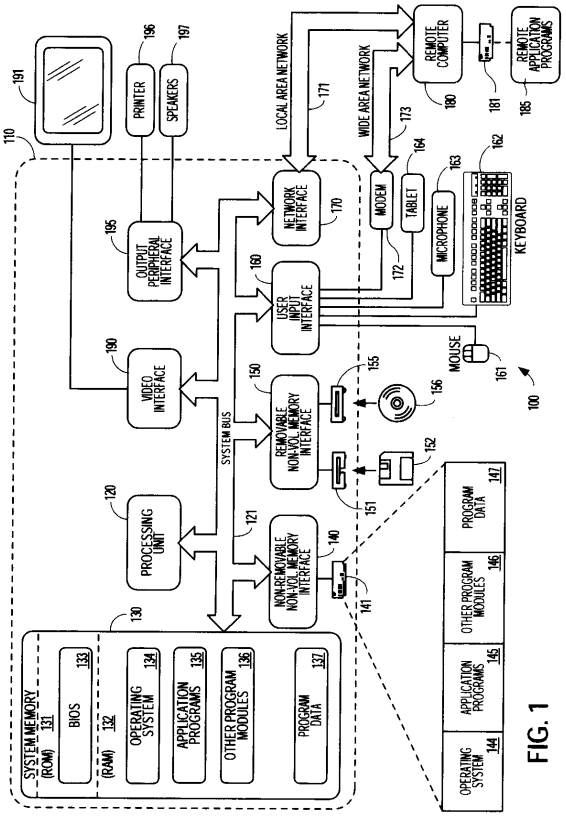



FIG. 1

FIG. 2

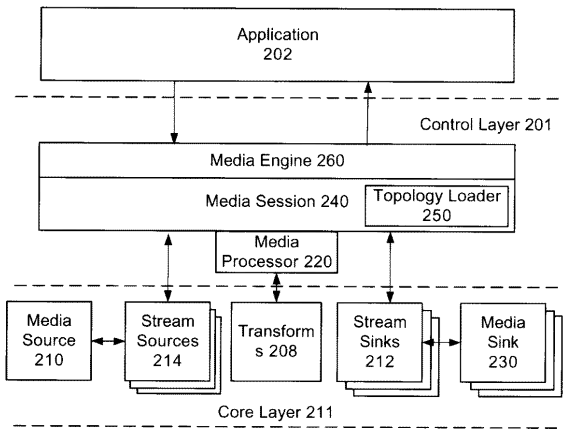


FIG. 3

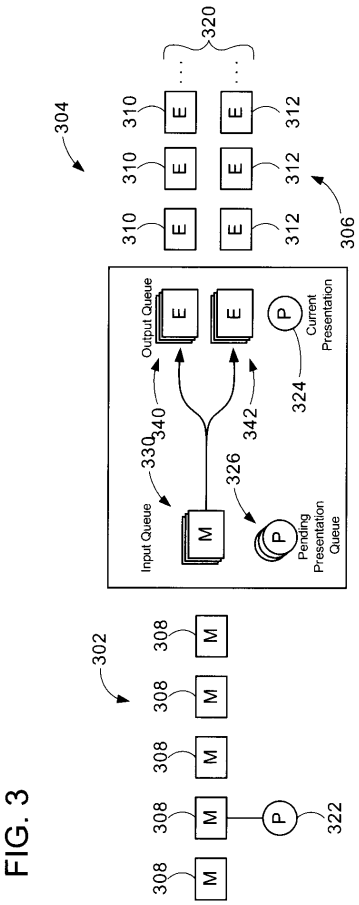



FIG. 4

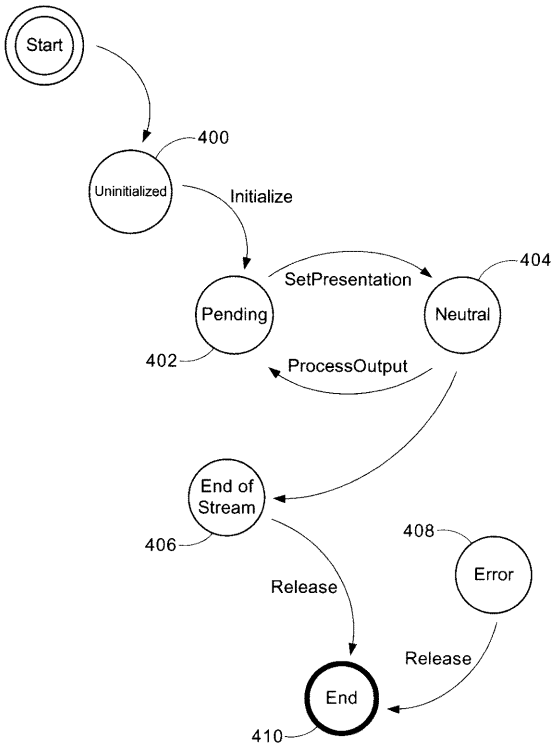



FIG. 5

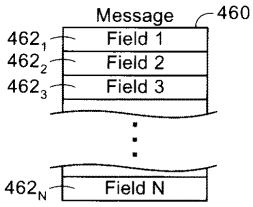



FIG. 6

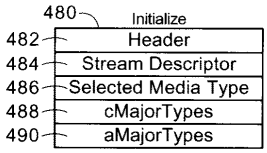



FIG. 7