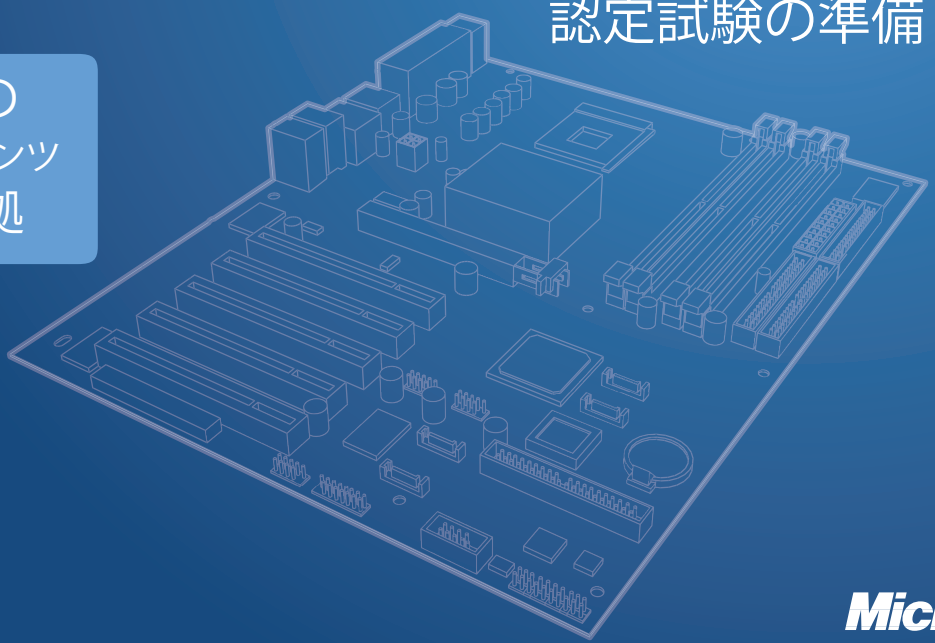
MCTS
Exam 70-571

Windows® Embedded CE 6.0

準備キット

認定試験の準備

最新の
R2 コンテンツ
に準拠



非売品

Microsoft®

出版元

Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

データ取得編集者 :	Sondra Webber、Microsoft Corporation
筆者 :	Nicolas Besson、Adeneo Corporation Ray Marcilla、Adeneo Corporation Rajesh Kakde、Adeneo Corporation
著作指導 :	Warren Lubow、Adeneo Corporation
技術レビューア :	Brigette Huang、Microsoft Corporation
編集出版 :	Biblioso Corporation

本体番号 3043-GA1

Body Part No. 098-109627

目次一覧

第 6 章

デバイス ドライバを開発する

デバイス ドライバは、ターゲット デバイスに統合または接続された周辺ハードウェアとオペレーティング システムやユーザー アプリケーションとの相互作用を可能にするコンポーネントです。周辺機器には、PCI (Peripheral Component Interconnect (PCI) バス、キーボード、マウス、シリアル ポート、ディスプレイ、ネットワーク アダプタ、および記憶デバイスが含まれます。ハードウェアに直接アクセスするのではなく、オペレーティング システム (OS) は、対応するデバイス ドライバをロードしてから、それらのドライバが提供する機能や入出力 (I/O) サービスを使用して、デバイスの動作を実行します。この方法で、Microsoft ィ Windows ィ Embedded CE 6.0 R2 アーキテクチャは、基盤となるハードウェア詳細の柔軟性、拡張性、および独立性を維持しています。ハードウェア固有のコードを含むハードウェア ドライバは、CE に同梱されている標準ドライバに加えて、カスタム ドライバを実装して、追加の付属機器をサポートさせることができます。実際、デバイス ドライバは、OS デザインのボード サポート パッケージ (BSP) の重要な部分を占めています。ただし、粗末に実装されたドライバは、信頼性のあるシステムを損傷することがあることに留意するのは重要です。デバイス ドライバを開発するとき、厳密なコーディング プラクティスに従い、多様なシステム構成でコンポーネントを徹底的にテストすることは必須です。この章では、適切なコード構造でデバイス ドライバを記述し、セキュアで適切に策定された構成のユーザー インターフェイスを開発し、長期間の使用にも耐える信頼性を確保し、複数の電源管理機能をサポートするためのベスト プラクティスについて紹介します。

本章の試験範囲：

- Windows Embedded CE でデバイス ドライバをロードおよび使用する
- システムで割り込みを管理する
- メモリ アクセスとメモリ処理を理解する
- ドライバの移植可能性とシステム統合を拡張する

始める前に

- この章のレッスンを完了するには、次が必要です。
- I/O コントロール (IOCTL) および直接メモリ アクセス (DMA) などの、ドライバ開発に関連する基本概念を含む、Windows Embedded CE ソフトウェア開発に関する基本的な知識。
- 割り込み処理およびデバイス　ドライバの割り込みに対する応答方法の理解。
- C および C++ のメモリ管理に精通していること、およびメモリ リークを回避する方法に関する知識。
- Microsoft Visual Studio ィ 2005 Service Pack 1 および Windows Embedded CE 6.0 R2 用 Platform Builder がインストールされている開発コンピュータ。

レッスン 1：デバイス ドライバの基本を理解する

Windows Embedded CE では、デバイス ドライバは、基盤となるハードウェアやオペレーティング システムとターゲット デバイスで実行しているアプリケーションの間の抽象的なレイヤを提供するダイナミック リンク ライブラリ (DLL) です。ドライバは、一式の既知の機能を表示し、初期化とハードウェアとの通信を行うロジックを提供します。ソフトウェア開発者は、ドライバの機能をアプリケーションで呼び出し、ハードウェアと相互にやり取りします。デバイス ドライバ インターフェイス (DDI) などの、既知のアプリケーション プログラミング インターフェイス (API) にデバイス ドライバが関連付けられている場合、ディスプレイ ドライバや記憶デバイスのドライバのように、ドライバをオペレーティング システムの一部としてロードすることができます。物理ハードウェアに関する詳細を必要とせずに、アプリケーションは、ReadFile や WriteFile などの標準 Windows API 関数を呼び出して、周辺デバイスを使用できます。異なるドライバを OS デザインに追加することにより、アプリケーションを再プログラムすることなく、異なるタイプの周辺機器をサポートさせることができます。

このレッスンを終了すると、以下をマスターできます :。

- ネイティブおよびストリーム ドライバの違いの理解。
- モノシリック ドライバおよび複数層ドライバ アーキテクチャの利点および不利な点を理解。

レッスン時間 (推定) : 15 分

ネイティブおよびストリーム ドライバ

Windows Embedded CE デバイス ドライバは、標準 DllMain 関数をエントリ ポイントとして表示する DLL であるため、親プロセスは、LoadLibrary または LoadDriver を呼び出すことでドライバをロードすることができます。LoadLibrary によってロードされたドライバをページ アウトできますが、オペレーティング システムは LoadDriver によってロードされたドライバをページ アウトしません。

すべてのドライバは D l l Main エントリ ポイントを表示しますが、Windows Embedded CE は、ネイティブ ドライバおよびストリーム ドライバという、2 つの異なるタイプのドライバをサポートします。通常、ネイティブ CE ドライバは、ディスプレイ ドライバ、キーボード ドライバ、およびタッチスクリーン ドライバなどの、入出力周辺機器をサポートします。グラフィックス、ウィンド

ウ、およびイベント サブシステム (GWES) は、これらのドライバを直接ロードおよび管理します。ネイティブ ドライバは、目的によって特定の関数を実装し、GWES は GetProcAddress API を呼び出すことによって決定できます。GetProcAddress は、ドライバが関数をサポートしない場合、ポインタを希望する関数か NULL に返します。

それに対し、ストリーム ドライバは、[デバイス マネージャ] でこれらのドライバをロードおよび管理できるようにする、既知の一連の関数を提供します。ストリーム ドライバと相互にやり取りをする [デバイス ドライバ] については、ドライバは、Init、Deinit、Open、Close、Read、Write、Seek、および IOControl 関数を実装する必要があります。多くのストリーム ドライバでは、Read、Write、および Seek 関数は、ストリーム コンテンツへのアクセスを提供しますが、すべての周辺機器がストリーム デバイスであるわけではありません。デバイスに、Read、Write、および Seek 以外の特別な要件がある場合、IOControl 関数を使用して、必要な関数を実装することができます。IOControl 関数は、ストリーム デバイス ドライバの特殊なすべての要件に適応できるようにする、ユニバーサル関数です。例えば、カスタム IOCTL コマンド コードおよび入力および出力バッファを渡すことで、ドライバの機能を拡張できます。

ノート　ネイティブ ドライバ インターフェイス

ネイティブ ドライバは、ドライバの種類に応じて、異なるタイプのインターフェイスを実装する必要があります。サポートされるドライバ タイプに関する完全な情報については、http://msdn2.microsoft.com/en-us/library/aa930800.aspx の Microsoft MSDN? Web サイトにある、Windows Embedded CE 6.0 ドキュメントの「Windows Embedded CE Drivers」セクションを参照してください。

モノシリック ドライバと複数層ドライバ アーキテクチャ

ネイティブ ドライバとストリーム ドライバは、表示する API について異なるのみです。システム起動時とオンデマンドの 2 つのタイプのドライバをロードすることができ、図 6-1 に示すように、両方ともモノシリック デザインか複数層デザインを使用することができます。

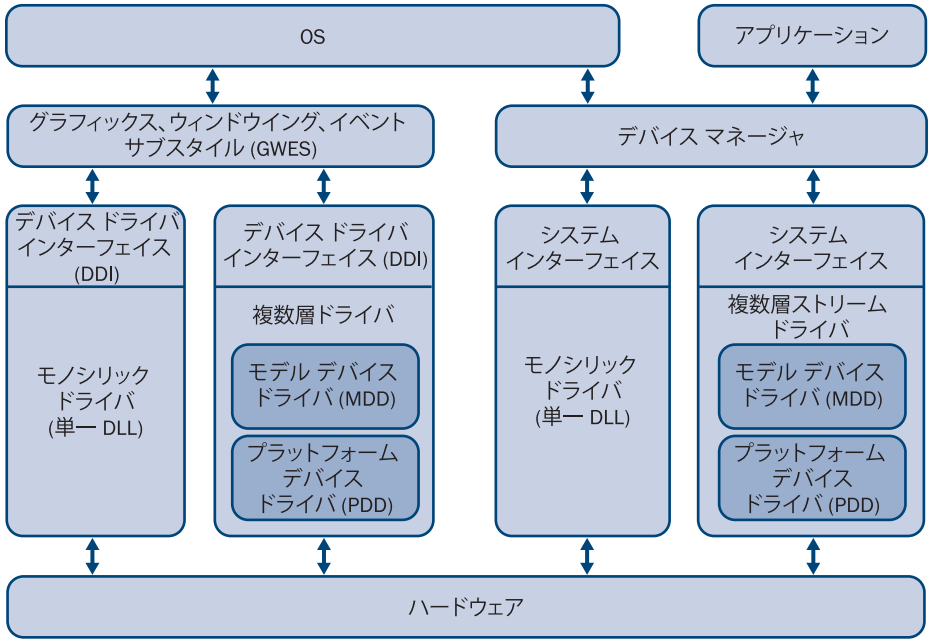


図 6-1　モノシリック ドライバおよび複数層ドライバ アーキテクチャ

モノシリック ドライバ

モノシリック ドライバは単一の DLL に依存しており、オペレーティング システムとアプリケーションへのインターフェイス、およびハードウェアへのロジックの両方を実装します。モノシリック ドライバの開発コストは、一般的に複数層ドライバより高いですが、この欠点にもかかわらず、モノシリック ドライバには多数の利点があります。主要な利点は、ドライバ アーキテクチャでの別個の層間を呼び出す追加関数を回避することにより、パフォーマンスを向上することができます。メモリ要件は、複数層ドライバに比べ、若干低くなります。モノシリック ドライバは、非共通カスタム ハードウェアの場合に適切な選択ともなります。再使用できる複数層ドライバ コードが存在しない場合、および固有のドライバ プロジェクトである場合、モノシリック アーキテクチャでドライバを実装する利点を実感できます。再使用可能なモノシリック ソース コードが使用可能な場合には特にそう言えます。

複数層ドライバ

コードの再利用を容易にし、開発経費やコストを軽減させるため、Windows Embedded CE は、モデル デバイス ドライバ (MDD) およびぷラットフォーム デバイス ドライバ (PDD) に基づく、複数層ドライバ アーキテクチャをサポートします。MDD および PDD は、ドライバ更新の追加抽象層を提供し、新しいハー

ドウェアのデバイス ドライバの開発を高速化します。MDD 層には、オペレーティング システムおよびアプリケーションへのインターフェイスが含まれています。一方、MDD は PDD 層のインターフェイスとなります。PDD 層は、ハードウェアと通信するための実際の機能を実装します。

複数層ドライバを新しいハードウェアに移行するとき、通常、MDD 層のコードを修正する必要はありません。最初から新しいドライバを作成するよりは、既存の複数層ドライバを複製してから機能を追加または削除するほうが複雑さを低減できます。Windows Embedded CE に含まれるドライバの多くは、複数層ドライバ アーキテクチャの利点を活用しています。

ノート　MDD/PDD アーキテクチャおよびドライバ更新

MDD/PDD アーキテクチャは、QFE (Quick Fix Engineering) 修正をカスタマに提供するなど、ドライバ開発者がドライバ更新の開発中の時間を節約する助けになります。PDD 層への修正を制限することは、開発の労力を増加させることになります。

レッスン概要

Windows Embedded CE は、ネイティブおよびストリーム ドライバをサポートします。ネイティブ ドライバは、ストリーム デバイスでないすべてのデバイスに対しては、最善の選択となります。例えば、ディスプレイ デバイス ドライバは独自のパターンでデータを処理できる必要があるため、ネイティブ ドライバの適切な候補となります。記憶ハードウェアおよびシリアル ポートなどの他のデバイスでは、データを、ファイルのように、バイトの指定されたストリーム形式で処理するため、ストリーム ドライバは最適な候補となります。ネイティブおよびストリーム ドライバは両方とも、モノシリックまたは複数層ドライバ デザインにすることができます。一般に、MDD および PDD に基づいて複数層アーキテクチャを使用するのには利点があります。コードの再利用やドライバ更新の開発が容易になるためです。モノシリック ドライバは、パフォーマンス上の理由で MDD および PDD 間の追加関数呼び出しを回避したい場合に、最適な選択となります。

レッスン 2：ストリーム インターフェイス ドライバを実装する

Windows Embedded CE では、ストリーム ドライバは、ストリーム インターフェイス API を実装するデバイス ドライバです。ハードウェア仕様に関係なく、すべての CE ストリーム ドライバは、ストリーム インターフェイス機能をオペレーティング システムに提供するため、Windows Embedded CE の [デバイス マネージャ] はこれらのドライバをロードおよび管理できます。名前が示しているように、ストリーム ドライバは、統合されたハードウェア コンポーネントや周辺機器などのデータ ストリームのソースやシンクとして動作する I/O デバイスに適しています。ただし、ストリーム ドライバで他のドライバにアクセスして、アプリケーションに基盤となるハードウェアへのより便利なアクセスを提供することもできます。どのような場合にも、完全に機能し信頼性のあるストリーム ドライバを開発したい場合は、ストリーム インターフェイス機能と実装方法を理解しておく必要があります。

このレッスンを終了すると、以下をマスターできます :。

- [デバイス マネージャ] の目的を理解する。
- ドライバ要件を識別する。
- ストリーム ドライバを実装および使用する。

レッスン時間 (推定) : 40 分

デバイス マネージャ

Windows Embeddded CE の [デバイス マネージャ] は、システムのストリーム デバイス ドライバを管理する OS コンポーネントです。ブート プロセス中に、OAL (Oal.exe) はカーネル (Kernel.dll) をロードし、カーネルは [デバイス ドライバ] をロードします。もっと正確に言えば、図 6-2 に示すように、カーネルはデバイス ドライバ シェル (Device.dll) をロードし、そしてこれが実際のコア デバイス ドライバ コード (Devmgr.dll) をロードし、これがロード、アンロード、およびストリーム ドライバとのインターフェイスを担当します。

ストリーム ドライバは、ブート時またはプラグ アンド プレイで該当するハードウェアが接続された場合はオンデマンドで、オペレーティング システムの一部としてロードされます。これで、ユーザー アプリケーションは、ReadFile および WriteFile などのファイル システム API を介して、または DeviceControl 呼び出しによって、ストリーム ドライバを使用することができます。[デバイス マ

ネージャ] がファイル システムを介して提供するストリーム ドライバは、特定のファイル名で通常のファイル リソースとしてアプリケーションに表示されます。一方、DeviceIoControl 関数は、アプリケーションが直接入出力操作を実行できるようにします。ただし、両方の条件で、アプリケーションは、[デバイス マネージャ] を介して、ストリーム ドライバと間接的にやり取りを行うことがあります。

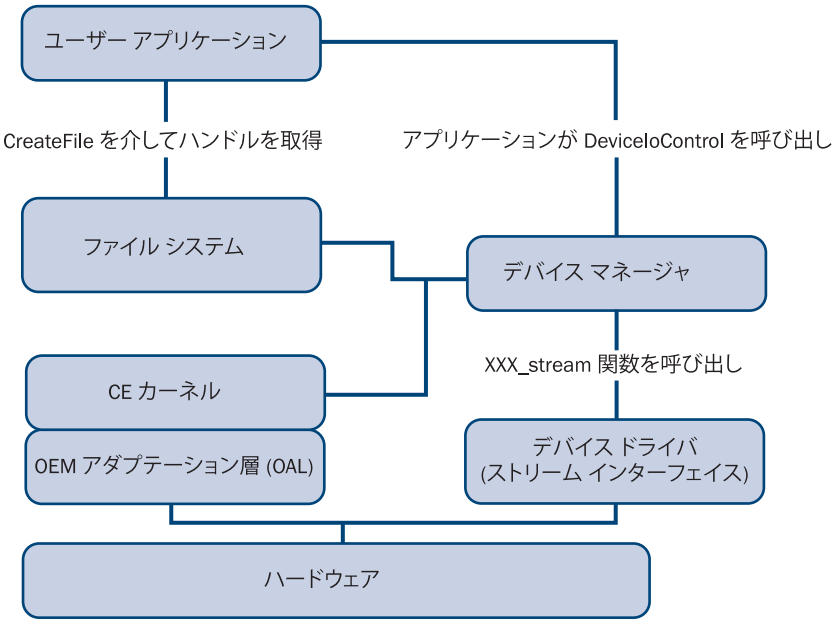


図 6-2　Windows Embedded CE 6.0 の [デバイス マネージャ]

ドライバ名前付け規則

ファイル システムを介してストリーム ドライバを使用するアプリケーションの場合、ストリーム ドライバはファイル リソースとして存在している必要があるため、アプリケーションはデバイス ファイルを CreateFile 呼び出しで指定してデバイスのハンドルを取得する必要があります。ハンドルを取得すると、アプリケーションは ReadFile や WriteFile を使用して入出力操作を行うことができます。この操作は、[デバイス マネージャ] が該当するストリーム インターフェイス関数への呼び出しに翻訳し、必要な読み込みおよび書き込み動作を実行します。Windows Embedded CE 6.0 では、ストリーム デバイス リソースの認識と適切なストリーム デバイスへのリダイレクト ファイル入出力処理が行われるため、ストリーム ドライバは、これらのリソースを通常のファイルと見分けるための、特別な名前付け規則に従う必要があります。

Windows Embedded CE 6.0 は、ストリーム ドライバの次の名前付け規則をサポートします。

- レガシ名　ストリーム ドライバの従来の名前付け規則で、3 つの大文字、1 つの数字、および 1 つのコロンで構成されます。形式は *XXX*[0 ミ 9]: です。ここで、*XXX* は 3 文字のドライバ名で、[*0 ミ 9*] は、ドライバのレジストリ設定で指定されたドライバのインデックスです (レッスン 3、「ドライバの構成とロード」を参照)。ドライバ インデックスは、1 つのみの数字であるため、レガシ名は最大 10 個のみのストリーム ドライバのインスタンスをサポートしています。最初のインスタンスはインデックス 1 に対応し、9 番目のインスタンスはインデックス 9 を使用し、10 番目のインスタンスはインデックス 0 となります。例えば、CreateFile(L"COM1:" ノ) は、レガシ名 COM1: を使用して最初のシリアル ポートのストリーム ドライバにアクセスします。

ノート　レガシ名の制限
従来の名前付け規則では、ストリーム ドライバごとに、10 個以上のインスタンスはサポートされていません。

- デバイス名　10 以上のインデックスのストリーム ドライバにアクセスするため、レガシ名の代わりにデバイス名を使用することができます。デバイス名は、\$device\XXX[インデックス] の形式に従います。ここで、\$device\ はデバイス名であることを示す名前空間で、*XXX* は 3 文字のドライバ名で、[*インデックス*] はドライバのインデックスです。インデックスは、複数の桁にすることができます。例えば、CreateFile(L"\$device\COM11" ノ) は、11 番目のシリアル ポートのストリーム ドライバにアクセスします。CreateFile(L"\$device\COM1" ノ) などの、レガシ名のストリーム ドライバにもアクセスできます。

ノート　レガシ名およびデバイス名アクセス
レガシ名およびデバイス名の形式は異なり、異なるドライバ インスタンスの範囲をサポートしていますが、両方の場合に、CreateFile は同一ストリーム ドライバへのアクセスを持つ同一のハンドルを返します。

- バス名　PCMCIA (Personal Computer Memory Card International Association) や USB (Universal Serial Bus) などのバス上のデバイス用のストリーム ドライバで、バス上で利用可能なドライバを列挙したときに、対応するバス ドライバが [デバイス ドライバ] に渡すバス名に対応していま

す。バス名は、基盤となるバス構造に関連しています。通常の形式は、\$bus\BUSNAME_[バス番号]_[デバイス番号]_[関数番号] で、ここで \$bus\ はバス名であることを示す名前空間、*BUSNAME* はバスの名前つまりタイプ、[*バス番号*]、[*デバイス番号*]、および [関数番号] は、バス固有の識別子を表しています。例えば、CreateFile(L"\$ bus\PCMCIA_0_0_0"ノ) は、デバイス 0、関数 0、PCMCIA バス 0 にアクセスします。これはシリアル ポートに該当します。

ノート　バス名アクセス

バス名は主に、バス ドライバをアンロードおよびリロードするため、および電源管理のためのハンドルを取得するために使用され、データ読み込みおよび書き込み操作には使用されません。

ストリーム インターフェイス API

[デバイス マネージャ] でストリーム ドライバのロードと管理を確実に行うため、ストリーム ドライバは、一般的にストリーム インターフェイスとして参照される、共通インターフェイスをエクスポートする必要があります。12 の関数で構成されるストリーム インターフェイスは、表 6-1 で要約されているように、デバイスの初期化およびデバイスのオープン、データの読み取りと書き込み、デバイスの電源オンおよびオフ、およびデバイスの初期化解除を行います。

表 6-1　ストリーム インターフェイス関数

関数名	説明
XXX_Init	[デバイス マネージャ] はこの関数を呼び出して、ブート プロセス中または ActivateDeviceEx の呼び出しの応答として、ドライバのロードを行うことで、ハードウェアおよびデバイスによって使用されているすべてのメモリ構造を初期化します。
XXX_PreDeinit	[デバイス マネージャ] は、XXX_Deinit を呼び出す前にこの関数を呼び出すことで、ドライバが休止中のスレッドを再開し、初期化解除プロセスを高速化するためにオープン ハンドルを無効にできるようにします。アプリケーションはこの関数を呼び出しません。
XXX_Deinit	[デバイス マネージャ] は、この関数を呼び出して、ドライバの無効化とアンロード後に DeActivateDevice 呼び出しの応答として、メモリ構造や他のリソースの初期化解除および割り当て解除を行います。

表 6-1 ストリーム インターフェイス関数

関数名	説明
XXX_Open	読み取り、書き込み、または両方の操作のために CreateFile を呼び出すことで、アプリケーション リクエストがデバイスにアクセスしたときに、[デバイス マネージャ] はこの関数を呼び出します。
XXX_PreClose	[デバイス マネージャ] は、この関数を呼び出すことで、アンロード プロセスを高速化するために、ドライバがハンドルを無効にし、休止中のスレッドを再開できるようにします。アプリケーションはこの関数を呼び出しません。
XXX_Close	[デバイス マネージャ] は、CloseHandle 関数を呼び出すなどによって、アプリケーションがドライバのオープン インスタンスを閉じたときに、この関数を呼び出します。ストリーム ドライバは、前回の XXX_Open 呼び出し中に割り当てられたすべてのメモリおよびリソースの割り当て解除を行う必要があります。
XXX_Read	[デバイス マネージャ] は、ReadFile 呼び出しの応答としてこの関数を呼び出し、デバイスからデータを読み取り、それを呼び出し元に渡します。デバイスは読み取り用のデータを提供しませんが、ストリーム デバイス ドライバはこの関数を実装して、[デバイス マネージャ] との互換性を確立する必要があります。
XXX_Write	[デバイス マネージャ] は、WriteFile 呼び出しの応答としてこの関数を呼び出し、呼び出し元からのデータをデバイスに渡します。XXX_Read と同様に、XXX_Write は必須ではありますが、入力専用通信ポートなどのように、基盤となるデバイスが書き込み操作をサポートしていない場合は、空にしておくことも可能です。
XXX_Seek	[デバイス マネージャ] は、SetFilePointer 呼び出しの応答としてこの関数を呼び出し、読み取りまたは書き込み用に、データ ストリームでデータ ポインタを特定のポイントに移動します。XXX_Read および XXX_Write と同様に、この関数は空にできますが、[デバイス マネージャ] との互換性の確立のためにエクスポートされる必要があります。

表 6-1 ストリーム インターフェイス関数

関数名	説明
XXX_IOControl	[デバイス マネージャ] は、DeviceIoControl 呼び出しの応答としてこの関数を呼び出し、デバイス固有のコントロール タスクを実行します。例えば、電源機能のクエリや電源ステータスの管理を IOCTL の IOCTL_POWER_CAPABILITIES、IOCTL_POWER_QUERY、および IOCTL_POWER_SET によって行う場合など、ドライバが電源管理インターフェイスのアドバタイズを行う場合、電源管理機能は DeviceIoControl 呼び出しに依存します。アプリケーションも DeviceIoControl 関数を使用して、XXX_Write や XXX_Read を使用しないデバイス ドライバでデータの読み取りおよび書き込みを行います。これは、多くのストリーム ドライバにおける共通のアプローチです。
XXX_PowerUp	[デバイス マネージャ] は、オペレーティング システムが低電力モードから戻ったときにこの関数を呼び出します。アプリケーションはこの関数を呼び出しません。この関数はカーネル モードで実行するため、外部 API を呼び出すことはできません。また、オペレーティング システムが単一スレッド モード、非ページ モードで実行されるため、ページ アウトすることはできません。Microsoft は、ドライバの機能を中断および再開するために、[電源管理] および電源管理 IOCTL に基づいて、ドライバに電源管理を実装させることを推奨しています。
XXX_PowerDown	[デバイス マネージャ] は、オペレーティング システムが休止モードに切り替わるときにこの関数を呼び出します。XXX_PowerUp と同様に、この関数はカーネル モードで実行するため、外部 API を実行できず、ページ アウトすることもできません。アプリケーションはこの関数を呼び出しません。Microsoft は、[電源管理] および電源管理 IOCTL に基づいて、ドライバに電源管理を実装させることを推奨しています。

ノート XXX_ プレフィックス

関数名では、プレフィックス XXX は、3 文字のデバイス名を参照するプレースホルダです。このプレフィックスをデバイス コードでの実際の名前に置き換える必要があります。例えば、COM と呼ばれるドライバの場合は COM_Init、SPI (Serial Peripheral Interface) ドライバの場合には SPI_Init となります。

デバイス ドライバ コンテキスト

[デバイス マネージャ] は、デバイス コンテキストおよびオープン コンテキスト パラメータに基づくコンテキスト管理をサポートしています。[デバイス マネージャ] は、パラメータを DWORD 値として、各関数呼び出しのあるストリーム ドライバに渡します。メモリ ブロックのように、ドライバがインスタンス固有のリソースの割り当ておよび割り当て解除する必要がある場合は、コンテキスト管理は不可欠な要素になります。デバイス ドライバが DLL であり、すべてのドライバ インスタンスによって共有されるドライバによって定義や割り当てが行われる、グローバル変数および他のメモリ構造を示唆していることに留意するのは重要です。XXX_Close または XXX_Deinit 呼び出しの応答として誤ったリソースの割り当て解除を行うと、メモリ リーク、アプリケーション障害、および一般的なシステムの不安定性の原因になることがあります。

ストリーム ドライバは、次の 2 つのレベルに基づいて、デバイス ドライバ インスタンスごとにコンテキスト情報を管理できます。

1. デバイス コンテキスト　ドライバは、XXX_Init 関数でこのコンテキストの初期化を行います。そのため、このコンテキストは初期化コンテキストとも呼ばれます。この主な目的は、ドライバがハードウェア アクセスに関連するリソースの管理を行えるようにサポートすることです。[デバイス マネージャ]　はこのコンテキスト情報を　XXX_Init、XXX_Open、XXX_PowerUp、XXX_PowerDown、XXX_PreDeinit および XXX_Deinit 関数に渡します。
2. オープン　コンテキスト　ドライバは、この　2　番目のコンテキストを XXX_Open 関数で初期化します。アプリケーションが CreateFile をストリーム ドライバ用に呼び出すたびに、ストリーム ドライバは新しいオープン コンテキストを作成します。オープン コンテキストは、ストリーム ドライバを有効にし、データ ポインタおよび他のリソースをそれぞれの開かれているドライバ インスタンスに関連付けます。[デバイス マネージャ] は XXX_Open 関数でデバイス コンテキストをストリーム ドライバに渡します。それにより、ドライバはデバイス コンテキストへの参照をオープン コンテキストに保存することができます。この方法で、ドライバは、XXX_Read、XXX_Write、XXX_Seek、XXX_IOControl、XXX_PreClose および XXX_Close などの後続の呼び出しで、デバイス コンテキスト情報へのアクセスを保持することができます。[デバイス マネージャ] は、オープン コンテキストのみをこれらの関数に DWORD パラメータの形式で渡します。

次のコード リストは、ドライバ名 SMP (例えば SMP1:) のサンプル ドライバ用にデバイス コンテキストを初期化する方法を示しています。

```
DWORD SMP_Init(LPCTSTR pContext, LPCVOID lpvBusContext)
{
    T_DRIVERINIT_STRUCTURE *pDeviceContext = (T_DRIVERINIT_STRUCTURE *)
        LocalAlloc(LMEM_ZEROINIT|LMEM_FIXED, sizeof(T_DRIVERINIT_STRUCTURE));

    if (pDeviceContext == NULL)
    {
        DEBUGMSG(ZONE_ERROR,(L" SMP: ERROR: Cannot allocate memory "

+ "for sample driver's device context.\r\n"));

        // ドライバが初期化に失敗した場合は 0 を返す。

return 0;
    }

    // システムの初期化を実行 ...

    pDeviceContext->dwOpenCount = 0;

    DEBUGMSG(ZONE_INIT,(L"SMP: Sample driver initialized.\r\n"));

    return (DWORD)pDeviceContext;
}
```

デバイス ドライバをビルドする

デバイス ドライバを作成するため、Windows Embedded CE DLL 用のサブプロジェクトを OS デザインに追加することができますが、これを行う最も一般的な方法は、デバイス ドライバのソース ファイルをボード サポート パッケージ (BSP) の [ドライバ] フォルダ内に追加することです。Windows Embedded CE サブプロジェクトの構成に関する詳細情報については、第 1 章「オペレーティング システム デザインのカスタマイズ」を参照してください。

デバイス ドライバの適切な開始点としては、Simple Windows Embedded CE DLL サブプロジェクトがあります。これにより、Windows Embedded CE サブプロジェクト ウィザードで [自動生成されたサブプロジェクト ファイル] ページを選択できます。これは、DLL、および空のモジュール定義 (.def) やレジストリ (.reg) ファイルなどの多様なパラメータ ファイル用の DllMain エントリ ポイントの定義を使用して、ソース コード ファイルを自動的に作成します。また、ソース ファイルを事前構成してターゲット DLL をビルドします。パラメータ ファイルおよびソース ファイルに関する詳細情報については、第 2 章「ランタイム イメージのビルドおよび展開」を参照してください。

ストリーム関数を実装する

DLL サブプロジェクトを作成すると、ソース コード ファイルを Visual Studio で開いて、必要な関数を追加してストリーム インターフェイスや必要なドライバ機能を実装することができます。次のコード リストに、全く作業を実行しないストリーム インターフェイス関数の定義を示します。

```
// SampleDriver.cpp : DLL アプリケーション用にエントリ ポイントを定義する。
//

#include "stdafx.h"

BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD  ul_reason_for_call,
                      LPVOID lpReserved)
{
    return TRUE;
}

DWORD SMP_Init(LPCTSTR pContext, LPCVOID lpvBusContext)
{
    // ここにデバイス コンテキスト初期化コードを実装する。
    return 0x1;
}

BOOL SMP_Deinit(DWORD hDeviceContext)
{
    // ここにデバイス コンテキストを閉じるためのコードを実装する。
    return TRUE;
}

DWORD SMP_Open(DWORD hDeviceContext, DWORD AccessCode, DWORD ShareMode)
{
    //ここにオープン コンテキスト初期化コードを実装する。
    return 0x2;
}

BOOL SMP_Close(DWORD hOpenContext)
{
    // ここにオープン コンテキストを閉じるためのコードを実装する。
    return TRUE;
}

DWORD SMP_Write(DWORD hOpenContext, LPCVOID pBuffer, DWORD Count)
{
    // ここにストリーム デバイスに書き込むためのコードを実装する。
    return Count;
}

DWORD SMP_Read(DWORD hOpenContext, LPVOID pBuffer, DWORD Count)
{
    // ここにストリーム デバイスから読み取るためのコードを実装する。
```

```
    return Count;
}

BOOL SMP_IOControl(DWORD hOpenContext, DWORD dwCode,
                   PBYTE pBufIn, DWORD dwLenIn, PBYTE pBufOut,
DWORD dwLenOut, PDWORD pdwActualOut)
{
    // ここに詳細ドライバ動作を処理するコードを実装する。
    return TRUE;
}

void SMP_PowerUp(DWORD hDeviceContext)
{
    // ここに電源管理コードを実装するか IO コントロールを使用する。
    return;
}

void SMP_PowerDown(DWORD hDeviceContext)
{
    // ここに電源管理コードを実装するか IO コントロールを使用する。
    return;
}
```

ストリーム関数をエクスポートする

ドライバ DLL でストリーム関数を外部アプリケーションにアクセス可能にするには、ビルド プロセス中に関数をエクスポートするリンカが必要です。C++ ではこのためのオプションがいくつか提供されていますが、ドライバ DLL と [デバイス マネージャ] と n 互換性のため、関数を DLL サブプロジェクトの .def ファイルで定義することで関数をエクスポートする必要があります。リンカは .def ファイルを使用し、どのファンクションでエクスポートするか、どのようにエクスポートするかを定義します。標準ストリーム ドライバの場合、ドライバのソース コードおよびレジストリ設定で指定したプレフィックスを使用してストリーム インターフェイス関数をエクスポートする必要があります。図 6ミ3 は、前述のセクションでリスト表示した、ストリーム インターフェイス スケルトンのサンプル .def ファイルを示しています。

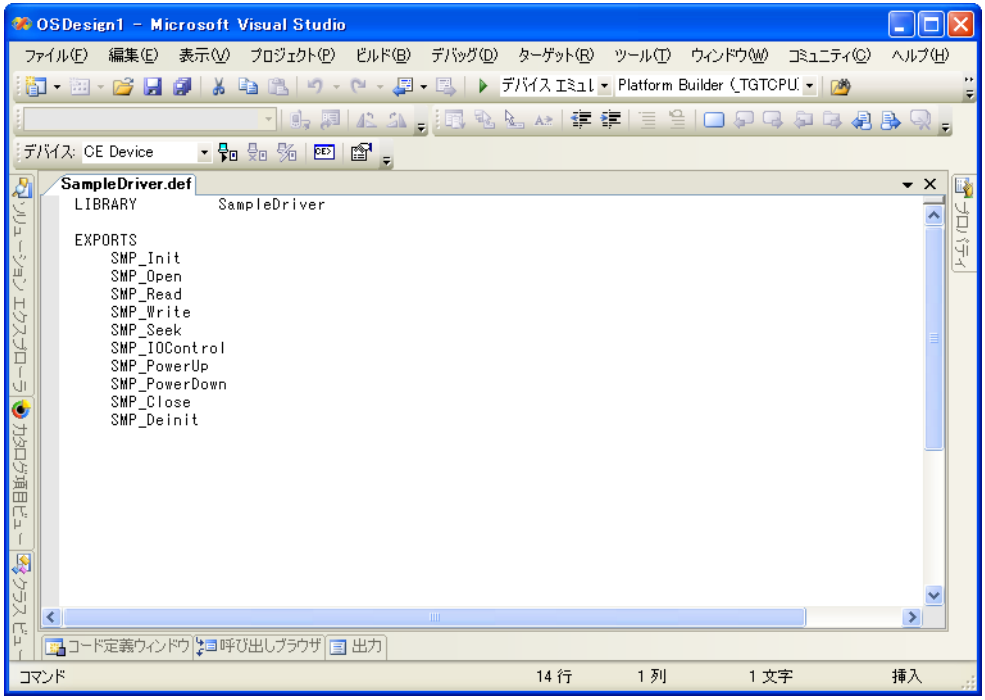


図 6-3 ストリーム ドライバのサンプル .def ファイル

ソース ファイル

新たに作成されたストリーム ドライバをビルドする前に、DLL サブプロジェクトのルート フォルダのソース ファイルを確認して、ビルド プロセスに必要なすべてのファイルが確実に含まれるようにする必要があります。第 2 章で説明したように、ソース ファイルは、コンパイラおよびリンカで構成され、必要なバイナリ ファイルをビルドします。表 6-2 は、デバイス ドライバ用の最も重要なソース ファイル指示子を列挙しています。

表 6-2 デバイス ドライバ用の重要なソース ファイル指示子

指示子	説明
WINCEOEM=1	追加ヘッダー ファイルおよび %_WINCEROOT%\Public ツリーからのインポート ライブラリを含むようにして、ドライバが KernelIoControl、InterruptInitialize、および InterruptDone などの、プラットフォーム依存関数呼び出しを実行できるようにします。
TARGETTYPE=DYNLINK	ビルド ツールが DLL を作成するように指示します。

表 6-2　デバイス ドライバ用の重要なソース ファイル指示子

指示子	説明
DEFFILE=*< ドライバ定義ファイル名 >*.def	エクスポートされた DLL 関数を定義する、モジュール定義ファイルを参照します。
DLLENTRY=*<DLL メイン エントリ ポイント >*	プロセスやスレッドのドライバ DLL へのアタッチ、およびプロセスやスレッドのドライバ DLL からのデタッチが行われるときに呼び出される関数を指定します (プロセス アタッチ、プロセス デタッチ、スレッド アタッチ、スレッド デタッチ)。

ファイル API を使用して、ストリーム ドライバを開くおよび閉じる

ストリーム ドライバにアクセスするには、アプリケーションは CreateFile 関数を使用でき、希望するデバイス名を指定する必要があります。次の例は、SMP1: という名前のドライバを読み取りおよび書き込み用に開く方法を示しています。ただし、重要な注意事項として、ブート プロセス中などに、[デバイス マネージャ] がドライバをすでにロードしている必要があります。この章の後述のレッスン 3 で、デバイス ドライバの構成およびロード方法に関する詳細を説明します。

```
// ドライバを開く。これにより、SMP_Open 関数の呼び出しが実行される。
hSampleDriver = CreateFile(L"SMP1:",
    GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    NULL);

if (hSampleDriver == INVALID_HANDLE_VALUE )
{
    ERRORMSG(1,(TEXT("Unable to open the driver.\r\n"));
    return FALSE;
}

// ドライバにアクセスし、必要に応じて、読み取り、
// 書き込み、および検索操作が実行される。

// ドライバを閉じる。
CloseHandle(hSampleDriver);
```

ドライバを動的にロードする

このレッスンで前述したように、アプリケーションは ActivateDevice または ActivateDeviceEx 関数 呼び出しの後に、ストリーム デバイス ドライバとの通信を行うことができます。ActivateDeviceEx は ActivateDevice よりも柔軟性がありますが、これらの関数は両方とも [デバイス マネージャ] にストリーム ドライバをロードさせ、ドライバの XXX_Init 関数を呼び出させます。実際、ActivateDevice は ActivateDeviceEx を呼び出します。ただし、ActivateDeviceEx はすでにロードされたドライバへのアクセスは提供しないことに注意してください。ActivateDeviceEx 関数の主な目的は、関数呼び出しで指定されたドライバ固有レジストリ キーを読み取って、DLL 名、デバイス プレフィックス、インデックス、および他の値を特定し、対応する値をアクティブ デバイス リストに追加してから、デバイス ドライバを [デバイス マネージャ] プロセス領域にロードします。関数呼び出しは、アプリケーションが後で DeactivateDevice 関数の呼び出しでドライバをアンロードするために使用可能なハンドルを返します。

ActivateDeviceEx は、次のコード サンプルで示すように、古い RegisterDevice 関数をメソッドとして置き換えてオンデマンドでドライバをロードします。

```
// [ デバイス ドライバ ] に、定義がレジストリの HKLM\Drivers\ サンプル
// にあるドライバをロードさせる。
hActiveDriver = ActivateDeviceEx(L"\\Drivers\\Sample", NULL, 0, NULL);
if (hActiveDriver == INVALID_HANDLE_VALUE)
{
    ERRORMSG(1, (L"Unable to load driver"));
    return -1;
}

// ドライバがロードされると、アプリケーションでドライバを開くことが可能。
hDriver = CreateFile (L"SMP1:",
                      GENERIC_READ| GENERIC_WRITE,
                      FILE_SHARE_READ | FILE_SHARE_WRITE,
                      NULL,
                      OPEN_EXISTING,
                      FILE_ATTRIBUTE_NORMAL,
                      NULL);

if (hDriver == INVALID_HANDLE_VALUE)
{
    ERRORMSG(1, (TEXT("Unable to open Sample (SMP) driver")));
    return 0;
}

// ここにドライバを使用するコードを挿入。

// アクセスが不要になったら、ドライバを閉じる。
if (hDriver != INVALID_HANDLE_VALUE)
```

```
{
    bRet = CloseHandle(hDriver);
    if (bRet == FALSE)
    {
        ERRORMSG(1, (TEXT("Unable to close SMP driver")));
    }
}

// [ デバイス マネージャ ] を使用して、手動でドライバをシステムからアンロードする。
if (hActiveDriver != INVALID_HANDLE_VALUE)
{
    bRet = DeactivateDevice(hActiveDriver);
    if (bRet == FALSE)
    {
        ERRORMSG(1, (TEXT("Unable to unload SMP driver ")));
    }
}
```

ノート　自動および動的にドライバをロードする

ActivateDeviceEx　を呼び出してドライバをロードすると、ブート　プロセス中に HKEY_LOCAL_MACHINE\Drivers\BuiltIn キーで定義されたパラメータを介して自動でドライバをロードしたときと同じ結果になります。BuiltIn レジストリ キーは、この章で後述するレッスン 3 でより詳しく説明します。

レッスン概要

ストリーム ドライバは、ストリーム インターフェイス API を実装する Windows Embedded CE ドライバです。ストリーム インターフェイスにより、[デバイス マネージャ] でこれらのドライバをロードおよび管理でき、アプリケーションは標準ファイル システム関数を使用してこれらのドライバへのアクセスと I/O 操作の実行を行います。ストリーム ドライバを CreateFile 呼び出しによってアクセス可能なファイル リソースとするには、ストリーム ドライバの名前は、デバイス リソースを通常のファイルから区別する次の特殊な名前付け規則に従う必要があります。レガシ名 (COM1: など) では、1 桁のインスタンス識別子のみを含めるため、ドライバごとに 10 個のインスタンスまでという制限があります。10 個以上のドライバ インスタンスをサポートする必要がある場合、デバイス名 (\$device\COM1 など) を代わりに使用します。

[デバイス マネージャ] は単一のドライバを複数回ロードして、異なるプロセスおよびスレッドからのリクエストに応えることができるため、ストリーム ドライバはコンテキスト管理を実装する必要があります。Windows Embedded CE は、デバイス ドライバ、デバイス コンテキストおよびオープン コンテキストの 2 つのコンテキスト レベルを認識します。オペレーティング システムはこれら

をドライバへの適切な各関数呼び出しに渡すことで、ドライバは内部リソースおよび割り当てられたメモリ領域を各呼び出し元に関連付けできます。

ストリーム インターフェイスは、次の 12 の関数で構成されています。XXX_Init、XXX_Open、XXX_Read、XXX_Write、XXX_Seek、XXX_IOControl、XXX_PowerUp、XXX_PowerDown、XXX_PreClose、XXX_Close、XXX_PreDeinit、および XXX_Deinit です。すべての関数が必須であるわけではありませんが (XXX_PreClose や XXX_PreDeinit など)、ストリーム デバイス ドライバが実装するすべての関数は、ドライバ DLL から [デバイス マネージャ] に公開する必要があります。これらの関数をエクスポートするため、それらを DLL サブプロジェクトの .def ファイルで定義する必要があります。DLL サブプロジェクトのソース ファイルを調整して、ドライバ DLL がプラットフォーム依存関数呼び出しを実行できるようにする必要もあります。

レッスン 3：ドライバの構成とロード

一般的に、Windows Embedded CE 6.0 でストリーム ドライバをロードする 2 つのオプションがあります。HKEY_LOCAL_MACHINE\Drivers\BuiltIn レジストリ キーでドライバ設定を構成することで、デバイス マネージャ] がブート シーケンス中にドライバを自動的にロードするように指定できます。または、ActivateDeviceEx を直接呼び出すことでドライバを動的にロードすることもできます。どちらの方法でも、[デバイス マネージャ] はデバイス ドライバを同一のレジストリ フラグおよび設定を使用してロードすることができます。キーの違いは、ActivateDeviceEx を使用するときにドライバへのハンドルを受け取ることです。これを後で DeactivateDevice への呼び出しで使用できます。開発段階では特に、ActivateDeviceEx によってドライバを動的にロードできる利点があります。それにより、ドライバのアンロード、更新バージョンのインストール、およびオペレーティング システムの再起動なしにドライバをリロードすることが可能になります。DeactivateDevice を使用して、BuiltIn レジストリ キーのエントリに基づいて自動的にロードされたドライバをアンロードすることができますが、ActivateDeviceEx を直接呼び出すことなくリロードすることはできません。

このレッスンを終了すると、以下をマスターできます：

- デバイス ドライバの必須レジストリ設定を識別する。
- ドライバ内からレジストリ設定にアクセスする。
- アプリケーションで、起動時やオンデマンドでドライバをロードする。
- ドライバをユーザー領域またはカーネル領域にロードする。

レッスン時間 (推定)：25 分

デバイス ドライバ ロード手順

デバイス ドライバを静的または動的にロードする場合でも、ActivateDeviceEx 関数は常に関係します。バス列挙子 (BusEnum) と呼ばれる専用ドライバは、ActivateDeviceEx を直接呼び出すことができるのと同様に、HKEY_LOCAL_MACHINE\Drivers\BuiltIn で登録されたすべてのドライバ用に ActivateDeviceEx を呼び出し、lpszDevKey パラメータでドライバ設定用の代替レジストリ パスに渡します。

[デバイス マネージャ] は次の手順を使用して、ブート時にデバイス ドライバをロードします。

3. [デバイス マネージャ] は HKEY_LOCAL_MACHINE\Drivers\RootKey エントリを読み取り、レジストリのデバイス ドライバ エントリの場所を特定します。RootKey エントリの既定値は Drivers\BuiltIn です。
4. [デバイス マネージャ] は、RootKey の場所 (HKEY_LOCAL_MACHINE\Drivers\BuiltIn) で指定された場所の Dll レジストリ値を読み取り、ロードする列挙子 DLL を特定します。既定では、これはバス列挙子 (BusEnum.dll) です。バス列挙子は、Init および Deinit 関数をエクスポートするストリーム ドライバです。
5. バス列挙子は起動時に実行し、RootKey レジストリの場所をスキャンして、追加のバスやデバイスを参照するサブキーを検索します。後で別の Roo t Key 使用して実行することで、さらに多くのドライバをロードすることができます。バス列挙子は、各サブキーの [順序] 値を調査してロード順序を決定します。
6. 最下位の [順序] 値から始めて、バス列挙子はサブキーを繰り返して確認し、現在のドライバのレジストリ パス (つまり、HKEY_LOCAL_MACHINE\Drivers\BuiltIn*<ドライバ名>*) に渡して ActivateDeviceEx を呼び出します。
7. ActivateDeviceEx は、ドライバのサブキーにある DLL 値に登録されているドライバ DLL をロードしてから、HKEY_LOCAL_MACHINE\Drivers\Active レジストリ キーでドライバのサブキーを作成して、現在のロードされたすべてのドライバの追跡を続けます。

図 6 ミ 4 は、オーディオ デバイス ドライバの HKEY_LOCAL_MACHINE\Drivers\BuiltIn レジストリ キーに登録されている一般的なレジストリを示しています。

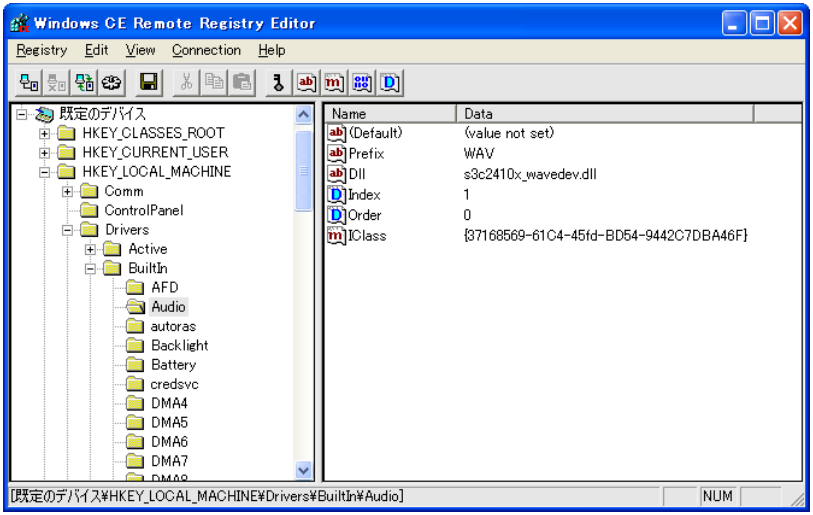


図 6-4　オーディオ デバイス ドライバ レジストリ

デバイス ドライバをロードするためのレジストリ設定

ActivateDeviceEx を使用してドライバを動的にロードする場合、ドライバのレジストリ設定を HKEY_LOCAL_MACHINE\Drivers\BuiltIn のサブキーにおく必要はありません。HKEY_LOCAL_MACHINE\SampleDriver などの、任意のパスを使用することができます。ただし、ドライバのレジストリ値はどちらの場合も同一です。表 6ミ3 はドライバのレジストリ サブキーでデバイス ドライバ用に指定できる一般的なレジストリ エントリの一覧を示しています (サンプル値については図 6ミ4 を参照)。

表 6-3　デバイス ドライバの一般的なレジストリ エントリ

レジストリ エントリ	タイプ	説明
プレフィックス	REG_SZ	ドライバの 3 文字の名前を含む文字列値。これは、ストリーム インターフェイス関数では、*XXX* と置き換える値になります。また、アプリケーションは、このプレフィックスを使用して CreateFile を介してドライバのコンテキストを開きます。
Dll	REG_SZ	これは、[デバイス マネージャ] がロードしてドライバをロードする DLL の名前です。 これは、ドライバの唯一の必須レジストリ エントリであることに注意してください。

表 6-3 デバイス ドライバの一般的なレジストリ エントリ

レジストリ エントリ	タイプ	説明
インデックス	REG_DWORD	これは、ドライバ プレフィックスに追加される数字で、ドライバ n ファイル名を作成します。例えば、この値が 1 の場合、アプリケーションは CreateFile(L"XXX1:"…) または CreateFile(L"\$device\XXX1"…) への呼び出しを介してこのドライバにアクセスできます。 この値はオプションであることに注意してください。定義しない場合、[デバイス マネージャ] は次の使用可能なインデックス値をドライバに割り当てます。
順序	REG_DWORD	これは、[デバイス マネージャ] がドライバをロードする順序です。この値が指定されず、他のドライバにも順序が指定されていない場合、、ドライバは最後にロードされます。同一の [順序] 値のドライバは同時に開始されます。 個の値は、連続的なロード順序を強制したい場合のみに使用します。例えば、GPS (Global Positioning System) ドライバは、UART (Universal Asynchronous Receiver/Transmitter) ドライバがシリアル ポートを介して GPS データにアクセスする場合に必要になります。この場合、UART ドライバに、GPS ドライバより低い [順序] 値を割り当てて、UART ドライバが最初に開始されるようにします。これにより、初期化中に、GPS ドライバが UART ドライバにアクセスできるようにします。
IClass	REG_MULTI_SZ	この値は、あらかじめ定義されたデバイス インターフェイスのグローバル一意識別子 (GUID) を指定することができます。プラグ アンド プレイ通知システムや電源管理機能をサポートするなどのために、インターフェイスを [デバイス マネージャ] にアドバタイズするには、IClass 値への次の該当するインターフェイス GUID を追加するか、AdvertiseInterface をドライバで呼び出す必要があります。

表 6-3 デバイス ドライバの一般的なレジストリ エントリ

レジストリ エントリ	タイプ	説明
フラグ	REG_DWORD	この値は、次のフラグを含めることができます。 ■ DEVFLAGS_UNLOAD (0x0000 0001) ドライバは、XXX_Init への呼び出し後にアンロードします。 ■ DEVFLAGS_NOLOAD (0x0000 0004) ドライバをロードできません。 ■ DEVFLAGS_NAKEDENTRIES (0x0000 0008) ドライバのエントリ ポイントは、Init、Open、IOControl などで、プレフィックスはありません。 ■ DEVFLAGS_BOOTPHASE_1 (0x0000 1000) ドライバは、システム フェーズ 1 で、複数のブート フェーズのあるシステム用にロードされます。これは、ブートプロセス中にドライバが複数回ロードされるのを回避します。 ■ DEVFLAGS_IRQ_EXCLUSIVE (0x0000 0100) バス ドライバは、IRQ 値によって指定された割り込みリクエスト (IRQ) への排他的アクセスがある場合にのみ、このドライバをロードします。 ■ DEVFLAGS_LOAD_AS_USERPROC (0x0000 0010) ユーザー モードでドライバをロードします。

ノート フラグ
フラグ レジストリ値の詳細については、http://msdn2.microsoft.com/en-us/library/aa929596.aspx の Microsoft MSDN Web サイトにある、Windows Embedded CE 6.0 ドキュメントの「ActivateDeviceEx」セクションを参照してください。

表 6-3 デバイス ドライバの一般的なレジストリ エントリ

レジストリ エントリ	タイプ	説明
UserProcGroup	REG_DWORD	ユーザー モードでロードする DEVFLAGS_LOAD_AS_USERPROC (0x0000 0010) フラグが付いたドライバを、ユーザー モードのドライバ ホスト プロセス グループに関連付けます。同一グループに属するユーザー モード ドライバは、同一ホスト プロセス インスタンスで [デバイス マネージャ] によってロードされます。このレジストリ エントリが存在しない場合、[デバイス マネージャ] はユーザー モード ドライバを新しいホスト プロセス インスタンスにロードします。

ロードされたデバイス ドライバに関連するレジストリ キー

ドライバ固有サブキーの構成可能なレジストリ エントリを除けば、[デバイス マネージャ] は HKEY_LOCAL_MACHINE\Drivers\Active キーでロードされたドライバ用のサブキーでの動的レジストリ情報を保持することもできます。サブキーは、オペレーティング システムが動的に割り当て、システムが再起動されるまで各ドライバに増分される数値に該当します。数値は、特定のドライバを指し示すわけではありません。例えば、デバイス ドライバをアンロードおよびリロードする場合、オペレーティング システムは次の番号をドライバに割り当て、以前のサブキーを再利用しません。サブキー番号と特定のデバイス ドライバ間の信頼できる関係を確証できないため、HKEY_LOCAL_MACHINE\Drivers\Active キーのドライバ エントリを手動で編集すべきではありません。ただし、ドライバの XXX_Init 関数で、ロード時のドライバ固有のレジストリ キーを作成、読み取り、および書き込みすることができます。これは、[デバイス マネージャ] が現在の Drivers\Active サブキーへのパスを、最初のパラメータとしてストリーム ドライバに渡すためです。ドライバは、OpenDeviceKey を使用してこのレジストリ キーを開くことができます。

表 6-4 に、Drivers\Active でサブキーに含めることのできる一般的なエントリのリストを示します。

表 6-4　HKEY_LOCAL_MACHINE\Drivers\Active キーのデバイス ドライバのレジストリ エントリ

レジストリ エントリ	タイプ	説明
Hnd	REG_DWORD	ロードされたデバイス ドライバのハンドル値です。この DWORD 値をレジストリから取得して、ドライバのアンロードのために DeactivateDevice への呼び出しに渡すことができます。
BusDriver	REG_SZ	ドライバのバスの名前です。
BusName	REG_SZ	デバイスのバスの名前です。
DevID		[デバイス マネージャ] からの一意のデバイス識別子です。
FullName	REG_SZ	$device 名前と共に使用する場合、デバイスの名前です。
Name	REG_SZ	プレフィックスが指定されている場合、インデックスを含むドライバのレガシ デバイス ファイル名です (プレフィックスを指定していないドライバには表示されません)。
Order	REG_DWORD	ドライバのレジストリ キーと同じ順序の値です。
Key	REG_SZ	ドライバのレジストリ キーへのレジストリ パスです。
PnpId	REG_SZ	PCMCIA ドライバ用プラグ アンド プレイ識別子です。
Sckt	REG_DWORD	PCMCIA ドライバの場合、PC カードの現在のソケットと機能状態を示します。

ノート　アクティブ キーを確認する

RequestDeviceNotifications 関数を DEVCLASS_STREAM_GUID のデバイス インターフェイス GUID と一緒に呼び出すことにより、アプリケーションは、[デバイス マネージャ] からメッセージを受け取って、機械的にロードされたストリーム ドライバを識別させることができます。RequestDeviceNotifications は、EnumDevices 関数と取って代わるものです。

カーネル モデルおよびユーザー モデル ドライバ

ドライバは、カーネルのメモリ領域かユーザー メモリ領域で実行することができます。カーネル モードでは、ドライバはハードウェアとカーネル メモリに対する完全なアクセスが可能ですが、関数呼び出しは通常、カーネル API への呼び出しに制限されます。既定では、Windows Embedded CE 6.0 はドライバをカーネル モードで実行します。それに対し、ユーザー モードのドライバはカーネル メモリに直接アクセスすることはできず、ユーザー モードで実行するとパフォーマンス面で不利な点がいくらかあります。ただし、ユーザー モードで発生したドライバ障害は現在のプロセスのみに影響すると言う利点があります。カーネル モードのドライバの障害は、オペレーティング システム全体に影響を与えます。システムは、通常、ユーザー モード ドライバの障害からは、より円滑に回復させることができます。

ノート　カーネル ドライバの制約

カーネル ドライバは、CE 6.0 R2 で直接ユーザー インターフェイスを表示することはできません。任意のユーザー インターフェイス要素を使用する場合、ユーザー モードにロードするコンパニオン DLL を作成してから、CeCallUserProc を使用してこの DLL を呼び出す必要があります。CeCallUserProc に関する詳細情報については、http://msdn2.microsoft.com/en-us/library/aa915093.aspx の MSDN Web ページを参照してください。

ユーザー モード ドライバおよびリフレクタ サービス

基盤となるハードウェアと通信したり、役に立つタスクを実行したりするには、ユーザー モード ドライバは、標準ユーザー モード プロセスでは使用できないシステム メモリおよび特権 API にアクセス可能である必要があります。これを容易にするには、Windows Embedded CE 6.0 はリフレクタ サービス機能を備えています。これは、カーネル モードで実行し、バッファ マーシャリングの実行、およびユーザー モード ドライバのために特権メモリ管理 API の呼び出しを行います。リフレクタ サービスは透過的であるため、ユーザー モード ドライバは、変更なしでカーネル モード ドライバとほとんど同じ方法で動作することができます。この方法に関する例外は、カーネル API を使用するドライバはユーザー モードでは使用できないということです。この種のカーネル モード ドライバはユーザー モードでは実行できません。

アプリケーションが ActivateDeviceEx を呼び出すと、[デバイス マネージャ] は、カーネル領域で直接ドライバをロードするか、リフレクタ サービスにリクエストを渡し、CreateProcess 呼び出しを使用して、ユーザー モードでドライバ ホスト プロセス (Udevice.exe) を開始します。ドライバのレジストリ キーのフラグ　レジストリ　エントリは、ドライバが　ユーザー　モード

(DEVFLAGS_LOAD_AS_USERPROC フラグ) で実行するかどうかを決定します。必要な Udevice.exe のインスタンスとユーザー モード ドライバを開始するとリフレクタ サービスは [デバイス マネージャ] から XXX_Init 呼び出しをユーザー モード ドライバに渡し、ユーザー モード ドライバからのリターン コードを [デバイス マネージャ] に返します (図 6-5 参照)。同一のプロキシ原則は他のすべてのストリーム関数に適用されます。

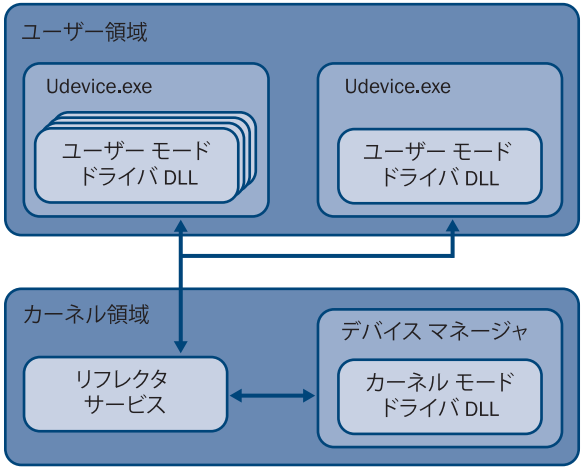


図 6-5　ユーザー モード ドライバ、カーネル モード ドライバ、およびリフレクタ サービス

ユーザー モード ドライバ レジストリ設定

Windows Embedded CE 6.0 では、単一のホスト プロセスで複数のユーザー モード ドライバを実行したり、システムで複数のホスト プロセスを有効にすることができます。単一 Udevice.exe インスタンスでグループ化されたドライバは、同一のプロセス領域を共有します。これは互いに依存するドライバで特に役立ちます。ただし、同一のプロセス領域のドライバは、互いの安定性に影響を与えることがあります。例えば、1 つのユーザー モード ドライバがホスト プロセスの失敗の原因となった場合、そのホスト プロセスのすべてのドライバが失敗します。影響を受けたドライバやこれらのドライバにアクセスするアプリケーションを除き、システムは動作し続けますが、アプリケーションがサポートする場合は、ドライバをリロードすることでこの状況から回復することができます。個別のユーザー モードのドライバ ホスト プロセスの重要なドライバを隔離すると、全体的なシステムの安定性を向上できます。表 6-5 に一覧表示されているレジストリ エントリを使用することで、個別のホスト プロセス グループを定義できます。

表 6-5 ユーザー モード ドライバ ホスト プロセス用レジストリ エントリ

レジストリ エントリ	タイプ	説明
HKEY_LOCAL_MACHINE\Drivers\ProcGroup_###	REG_KEY	ユーザー モード ドライバ ホスト プロセス用の 3 文字のグループ ID (###) を定義します。例えば、ProcGroup_003 で、ドライバのレジストリ キーの UserProcGroup エントリで UserProcGroup =3 などのように指定することができます。
ProcName	REG_SZ	ユーザー モード ドライバをホストするためにリフレクタ ドライバが開始するプロセスです。例えば、ProcName= Udevice.exe とします。
ProcVolPrefix	REG_SZ	ユーザー モード ドライバ ホスト プロセス用にリフレクタ サービスがマウントするためのファイル システム ボリュームを指定します。例えば、ProcVolPrefix = $udevice とします。指定された ProcVolPrefix は、ドライバ デバイス名の $device ボリュームと置き換えられます。

必要なホスト プロセス グループを定義すると、UserProcGroup レジストリ エントリをデバイス ドライバのレジストリ サブキーに追加することで (このレッスンで前述した表 6ミ3 を参照)、各ユーザー モード ドライバを特定のグループと関連付けることができます . 既定では、UserProcGroup レジストリ エントリは存在せず、すべてのユーザー モード ドライバを個別のホスト プロセス インスタンスにロードする、[デバイス マネージャ] の構成に対応します。

バイナリ イメージ ビルダ構成

第 2 章「ランタイム イメージのビルドおよび展開」で説明したように、Windows Embedded CE ビルド プロセスは、バイナリ イメージ ビルダ (.bib) ファイルに依存して、ランタイム イメージのコンテンツの生成およびデバイスの最終メモリ レイアウトの定義を行います。他にも、ドライバのモジュール定義用にフラグの組み合わせを指定することもできます。.bib ファイル設定およびレジストリ エントリがデバイス ドライバと合致しない場合に、問題が発生することがあります。例えば、.bib ファイルのデバイス ドライバ モジュール用に K フラグを指定して、ドライバのレジストリ サブキーの DEVFLAGS_LOAD_AS_USERPROC を

設定して、ドライバをユーザー モード ドライバ ホスト プロセスにロードするようにした場合、K フラグが Romimage.exe にメモリ アドレス 0x80000000 以上のカーネル領域にあるモジュールをロードするように命令するため、ドライバのロードに失敗します。ユーザー モードでドライバをロードするには、モジュールを 0x80000000 以下のユーザー領域にロードするようにする必要があります。例えば、BSP の Config.bib で定義された NK メモリ領域にロードします。

次の .bib ファイル エントリは、ユーザー モード ドライバを NK メモリ領域にロードする方法を示しています。

```
driver.dll      $(_FLATRELEASEDIR)\driver.dll    NK    SHQ
```

S および H フラグは、Driver.dll がシステム ファイルおよび隠しファイルの両方であり、フラット リリース ディレクトリにあることを示しています。Q フラグは、システムがこのモジュールを同時にカーネル領域およびユーザー領域の両方にロードできることを示しています。これは、DLL の 2 つのコピーをランタイム イメージに作成し、1 つを K フラグあり、もう 1 つを K フラグなしにします。このようにすると、ドライバの ROM および RAM 領域要件は 2 倍になります。Q フラグは控え目に使用します。

上記の例を拡張すると、Q フラグは次に対応します。

```
driver.dll      $(_FLATRELEASEDIR)\driver.dll    NK    SH
driver.dll      $(_FLATRELEASEDIR)\driver.dll    NK    SHK
```

レッスン概要

Windows Embedded CE はドライバをカーネル領域とユーザー領域にロードできます。カーネル領域で実行しているドライバは、システム API およびカーネル メモリにアクセスすることができ、障害が発生した場合のシステムの安定性に影響することがあります。ただし、適切に実装されたカーネル モード ドライバは、カーネル モードとユーザー モード間のコンテキスト スイッチを低減することで、ユーザー モード ドライバよりも良いパフォーマンスを示します。それに対し、ユーザー モード ドライバの利点は、障害が主に現在のユーザー モード プロセスのみに影響することです。ユーザー モード ドライバの権限は限られますが、サードパーティ ベンダからの信頼されていないドライバの場合には、重要な機能となりえます。

ユーザー モードで実行しているドライバをカーネル モードで実行している [デバイス マネージャ] と統合するには、[デバイス マネージャ] はユーザー モード ドライバ ホスト プロセスのドライバをロードするリフレクタ サービスを使

用し、ストリーム関数呼び出しおよび戻り値をドライバと [デバイス マネージャ] 間で転送します。このようにして、アプリケーションは使い慣れたファイル システム API を引き続き使用してドライバにアクセスできるため、ドライバは [デバイス マネージャ] との互換性を維持するために、ストリーム インターフェイス API に関してコードを変更する必要がありません。既定では、ユーザー モード ドライバは別個のホスト プロセスで実行しますが、ホスト プロセス グループを構成し、対応する UserProcGroup レジストリ エントリをドライバのレジストリ サブキーに追加することで、ドライバをそれらのグループと関連付けることができます。ドライバ サブキーは任意のレジストリの場所に置くことができますが、ブート時に自動的にドライバをロードしたい場合、サブキーを [デバイス マネージャ] の、既定で HKEY_LOCAL_MACHINE\Drivers\BuiltIn にある RootKey におく必要があります。サブキーが別の場所にあるドライバは、ActivateDeviceEx 関数を呼び出すことで、オンデマンドでロードすることができます。

レッスン 4：デバイス ドライバに割り込み機構を実装する

割り込みはハードウェアやソフトウェアで生成される通知で、タイマ イベントやキーボード イベントなど、CPU に迅速な対応が必要なイベントが発生したことを知らせます。割り込みに対応して、CPU は現在のスレッドの実行を停止し、カーネルのトラップ ハンドラにジャンプしてイベントに応答してから、割り込みを処理した後に元のスレッドの実行を再開します。このようにして、統合された周辺ハードウェア コンポーネント (システム クロック、シリアル ポート、ネットワーク アダプタ、キーボード、マウス、タッチスクリーン、および他のデバイスなど) が CPU によって検知させ、カーネル例外ハンドラがカーネルまたは関連デバイス ドライバ内の割り込みサービス ルーチン (ISR) で適切なコードを実行するようにできます。デバイス ドライバで割り込みプロセスを効率的に実装するため、カーネルでの ISR の登録や [デバイス マネージャ] プロセスでの割り込みサービス スレッド (IST) の実行を含む、Windows Embedded CE 6.0 割り込み処理機構に関する詳しい理解が必要です。

このレッスンを終了すると、以下をマスターできます :

- OEM アダプテーション層 (OAL) で割り込みハンドラを実装する。
- デバイス ドライバの割り込みサービス スレッド (IST) で割り込みを登録および処理する。

レッスン時間 (推定) : 40 分

割り込み処理アーキテクチャ

Windows Embedded CE 6.0 は、柔軟性のある割り込み処理アーキテクチャを実装することにより、さまざまな割り込みスキームを使用した異なる CPU タイプをサポートするポータブル オペレーティング システムです。最も重要なこととして、割り込み処理アーキテクチャは、Windows Embedded CE の OEM アダプテーション層 (OAL) の割り込み同期機能およびスレッド同期機能の利点を活用することで、割り込み処理を ISR と IST に分割します (図 6-6 を参照)。

Windows Embedded CE 6.0 割り込み処理は、次の概念に基づいています。

1. ブート プロセス中に、カーネルは OAL で OEMInit 関数を呼び出し、割り込みリクエスト (IRQ) 値に基づいて対応するハードウェア割り込みを使用して、すべての使用可能な ISR ビルトをカーネルに登録します。IRQ 値は、プロセッサ割り込みコントローラ レジスタで割り込みの原因を識別する数値です。

2. デバイス ドライバは、LoadIntChainHandler 関数を呼び出すことによって、ISR DLL に実装された ISR を動的にインストールすることができます。LoadIntChainHandler は、ISR DLL をカーネル メモリ領域にロードし、指定された ISR ルーチンを指定された IRQ 値と一緒にカーネルの割り込みディスパッチ テーブルに登録します。
3. イベントが現在のスレッドの実行を停止し、コントロールを別のルーチンに転送することが必要であることを CPU に通知するために、割り込みが発生します。
4. 割り込みの応答として、CPU は現在のスレッドの実行を停止し、すべての割り込みの主なターゲットとしてカーネル例外ハンドラにジャンプします。
5. 例外ハンドラは、優先度が同等または低いすべての割り込みをマスク オフし、現在の割り込みを処理するために登録されている適切な ISR を呼び出します。ほとんどのハードウェア プラットフォームは、割り込みマスクと割り込み優先度を使用して、ハードウェア ベースの割り込み同期機構を実装しています。
6. ISR は、現在の割り込みのマスクなどのすべての必要なタスクを実行することで、現在のハードウェア デバイスがそれ以上の割り込みをトリガしないようにして現在の処理を妨害しないようにしてから、SYSINTR 値を例外ハンドラに返します。SYSINTR 値は、論理割り込み識別子です。
7. 例外ハンドラは、SYSINTR 値をカーネルの例外サポート ハンドラに渡し、SYSINTR 値のイベントを定義し、(見つかった場合には) 割り込みを待機する IST のイベントについての信号を発信します。
8. 割り込みサポート ハンドラは、すべての割り込みのマスクを解除し、現在処理中の割り込みの例外を使用します。現在の例外のマスク オフを明示的に維持することにより、現在のハードウェア デバイスが IST の実行中に別の割り込みをトリガすることを回避できます。
9. 信号が発信されたイベントの応答として IST を実行することで、システムの他のデバイスをブロックすることなく割り込み処理を実行および完了します。
10. IST は、InterruptDone 関数を実行して、カーネルの割り込みサポート ハンドラに IST が処理を完了し、別の割り込みイベントの準備ができていることを知らせます。
11. 割り込みサポート ハンドラは、OAL の OEMInterruptDone 関数を呼び出すことで、割り込み処理プロセスを完了し、割り込みを再利用可能にします。

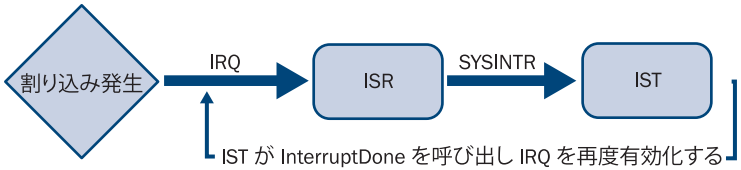


図 6-6　IRQ、ISRs、SYSINTR、および IST

割り込みサービス ルーチン

一般的に、ISR はハードウェア割り込みの応答として実行するコードの小さなブロックです。この ISR が実行中に、カーネル例外ハンドラは優先度が同じまたは低いすべての例外をマスク オフするため、ISR を完了し、SYSINTR 値をできるだけ迅速に返すことにより、カーネルがすべての IRQ を最小の遅延で再度有効 (アンマスク) にできるようにすることは重要です (現在処理されている割り込みは除く)。ISR に長い時間を費やしすぎると、システム パフォーマンスを著しく損なって、いくつかのデバイス上で、割り込みの失敗やオーバーラン バッファをもたらすことがあります。他の重要な側面としては、ISR はカーネル モードで実行し、より高いレベルのオペレーティング システム API にアクセスできません。これらの理由で、ISR は通常、ハードウェア レジスタからメモリ バッファへの高速データ コピーなどの、最も基本的なタスク以外は実行しません。Windows Embedded CE では、時間を要する割り込み処理は通常 IST で実行されます。

ISR の主なタスクは、割り込みの原因を特定し、デバイスでその割り込みをマスク オフまたはクリアしてから、割り込み用に SYSINTR 値を返してカーネルに実行する IST について通知することです。単純な状況では、ISR は SYSINTR_NOP を返して、必要な処理がこれ以上ないことを示します。状況に応じて、カーネルは、割り込みを処理する IST のイベントへの通知を行いません。それに対し、デバイス ドライバが IST を使用して割り込みを処理している場合、ISR は論理割り込み識別子をカーネルに渡し、カーネルは割り込みイベントの決定と通知を行い、IST は通常通り WaitForSingleObject 呼び出しから再開してループで割り込み処理命令を実行します。ISR と IST 間の待ち時間は、第 3 章「システム プログラミングの実行」で説明されているように、システムで実行されているそのスレッドおよび他のスレッドの優先度に依存しています。通常、IST は高い優先度のスレッドとともに実行されます。

割り込みサービス スレッド

IST は、ISR の完了後に、割り込みの応答として追加の処理を実行する正規のスレッドです。IST 関数には、通常、ループおよび WaitForSingleObject 呼び出しが含まれ、カーネルが指定された IST イベントを通知するまでスレッドを無制

限にブロックします。次のコード スニペットを参照してください。ただし、IST イベントを使用できるようにする前に、InterruptInitialize を SYSINTR 値を使用して、およびイベント ハンドラをパラメータとして呼び出すことで、CE カーネルが ISR が SYSINTR 値を返すときにはいつでもこのイベントを通知できるようにする必要があります。第 3 章では、マルチ スレッド プログラミングおよびイベントや他のカーネル オブジェクトに基づくスレッド同期について詳しい情報が提供されています。

```
CeSetThreadPriority(GetCurrentThread(), 200);

   // 停止の指示があるまでループ
   while(!pIst->stop)
   {
       // IST イベントを待機する。
       WaitForSingleObject(pIst->hevIrq, INFINITE)

       // 割り込みを処理する。
       InterruptDone(pIst->sysIntr);
   }
```

IST が IRQ の処理を完了すると、InterruptDone を呼び出して、割り込みが処理され、IST が次の IRQ を処理する準備ができており、OEMInterruptDone 呼び出しによって割り込みを再度有効にできることをシステムに通知します。表 6-6 に、システムが使用して割り込みコントローラと相互作用して割り込みの管理を行う、OAL 関数のリストを示します。

表 6-6 割り込み管理用 OAL 関数

関数	説明
OEMInterruptEnable	この関数は、カーネルによって InterruptInitialize の応答として呼び出され、指定された割り込みを割り込みコントローラ内で有効にします。
OEMInterruptDone	この関数は、カーネルによって InterruptDone の応答として呼び出され、割り込みをマスク解除し、割り込みコントローラの割り込みを承認します。
OEMInterruptDisable	この関数は、割り込みコントローラの割り込みを無効にし、InterruptDisable 関数の応答として呼び出されます。
OEMInterruptHandler	ARM プロセッサのみでは、この関数は割り込みコントローラのステータスの確認によって発生した SYSINTR 割り込みを識別します。

表 6-6　割り込み管理用 OAL 関数

関数	説明
HookInterrupt	ARM 以外のプロセッサでは、この関数は、callback 関数を指定された割り込み ID で登録します。この関数は、OEMInit 関数で呼び出して必須割り込みを登録する必要があります。
OEMInterruptHandlerFIQ	ARM プロセッサでは、高速割り込み (FIQ) 行の割り込みを処理するために使用されます。

注意　WaitForMultipleObjects 登録

WaitForMultipleObjects 関数を使用して、割り込みイベントの待機を行わないでください。複数の割り込みイベントを待機する必要がある場合は、IST を各割り込み用に作成する必要があります。

割り込み識別子 (IRQ および SYSINTR)

各ハードウェア割り込み行は、割り込みコントローラ レジスタの IRQ 値に該当します。各 IRQ 値は、1 つのみの ISR と関連付けることができますが、ISR は複数の IRQ をマップすることができます。カーネルは IRQ を保持する必要はありません。カーネルは、IRQ の応答として ISR から返された SYSINTR 値を特定し通知するのみです。ISR からのさまざまな SYSINTR 値を返す機能により、同一の共有割り込みを使用する複数のデバイスをサポートするための基盤が提供されます。

ノート　OEMInterruptHandler および HookInterrupt

ARM ベース システムのように、単一の IRQ のみをサポートするターゲット デバイスは、OEMInterruptHandler 関数を ISR として使用して、割り込みをトリガする埋め込み周辺機器を識別します。OEM（相手先ブランド供給）では、OAL の一部としてこの関数を実装する必要があります。Intel 86 ベース システムなどの、複数の IRQ をサポートするプラットフォームでは、HookInterrupt を呼び出すことで、複数の IRQ を個別の ISR と関連付けることができます。

静的割り込みマッピング

正しい SYSINTR 戻り値を特定する ISR の場合、IRQ および SYSINTR 間にマッピングが必要で、OAL にハードコードすることができます。デバイス エミュレータ BSP 用 Bsp_cfg.h ファイル は、SYSINTR_FIRMWARE 値に関連するターゲット デバイス用の OAL の SYSINTR 値を定義する方法を示します。自身の OAL で

カスタム ターゲット デバイス用に追加の識別子を定義したい場合、カーネルは SYSINTR_FIRMWARE より低いすべての値を将来の使用のために保持すること、および最大値は SYSINTR_MAXIMUM 以下であるべきことに留意する必要があります。

静的 SYSINTR 値のマッピングをターゲット デバイスの IRQ に追加するため、システム初期化中に OALIntrStaticTranslate 関数を呼び出すことができます。例えば、デバイス エミュレータ BSP は、BSPIntrInit 関数で OALIntrStaticTranslate を呼び出して、組み込み OHCI （Open Host Controller Interface) 用にカーネルの割り込みマッピング配列 (g_oalSysIntr2Irq および g_oalIrq2SysIntr) でカスタム SYSINTR 値を登録します。ただし、静的 SYSINTR 値およびマッピングは、IRQ を SYSINTR と関連付ける一般的な方法ではありません。それは、難解でカスタム割り込み処理を実装するために OAL コード変更が必要なためです。静的 SYSINTR 値は、一般に、明示的なデバイス ドライバがなく、OAL に ISR が存在するターゲット デバイスのコア ハードウェア コンポーネントに使用されます。

動的割り込みマッピング

IOCTL_HAL_REQUEST_SYSINTR の IO コントロール コードを使用してデバイス ドライバの KernelIoControl を呼び出して IRQ/SYSINTR マッピングを登録する場合、SYSINTR 値をハードコードする必要がないのが利点です。結果的に、呼び出しは OALIntrRequestSysIntr 関数で終了して、指定された IRQ 用の新しい SYSINTR を動的に割り当ててから、カーネルの割り込みマッピング配列で IRQ および SYSINTR マッピングを登録します。最大 SYSINTR_MAXUMUM まで自由に SYSINTR 値を配置することは、静的な SYSINTR 割り当てよりも柔軟性があります。これは、この機能では、新しいドライバを BSP に追加するときに OAL に修正を加える必要がないためです。

IOCTL_HAL_REQUEST_SYSINTR を使用して KernelIoControl を呼び出すと、IRQ と SYSINTR 間で 1:1 の関係を確立します。IRQ-SYSINTR マッピング テーブルに、すでに指定された IRQ のエントリがある場合、OALIntrRequestSysIntr は 2 番目のエントリを作成します。割り込みマッピング テーブルからのエントリを削除するには、KernelIoControl を IOCTL_HAL_REQUEST_SYSINTR の IO コントロール コードを使用して呼び出します。IOCTL_HAL_RELEASE_SYSINTR は、IRQ を SYSINTR 値から関係を解除します。

次のコード サンプルは、IOCTL_HAL_REQUEST_SYSINTR および IOCTL_HAL_RELEASE_SYSINTR の使用方法を示しています。カスタム値 (dwLogintr) を取得し、この値を OAL に渡して SYSINTR 値に変換してから、この SYSINTR を IST イベントと関連付けます。

```
DWORD dwLogintr = IRQ_VALUE;
DWORD dwSysintr = 0;
HANDLE hEvent   = NULL;
BOOL bResult    = TRUE;

// 割り当てと関連付けるイベントを作成する
m_hEvent = CreateEvent(NULL,FALSE,FALSE,NULL);
if (m_hDetectionEvent == NULL)
{
    return ERROR_VALUE;
}

// カーネル (OAL) に SYSINTR 値を IRQ に関連付けるように依頼する。
bResult = KernelIoControl(IOCTL_HAL_REQUEST_SYSINTR,
                          &dwLogintr, sizeof(dwLogintr),
                          &dwSysintr, sizeof(dwSysintr),
                          0);
if (bResult == FALSE)
{
    return ERROR_VALUE;
}

// 割り込みを初期化し、SYSINTR 値をイベントと関連付ける。
bResult = InterruptInitialize(dwSysintr, hEvent,0,0);

if (bResult == FALSE)
{
    return ERROR_VALUE;
}

// 割り込み管理ループ
while(!m_bTerminateDetectionThread)
{
    // 割り込みに関連付けられたイベントを待機
    WaitForSingleObject(hEvent,INFINITE);

    // ここで実際の IST 処理を追加

    // 割り込みを承認
    InterruptDone(m_dwSysintr);
}


// 割り込みの初期化解除により割り込みをマスク
bResult = InterruptDisable(dwSysintr);

// SYSINTR の登録を解除
bResult = KernelIoControl(IOCTL_HAL_RELEASE_SYSINTR,
                          &dwSysintr, sizeof(dwSysintr),
                          NULL,0,
                          0);

// イベント オブジェクトを閉じる
CloseHandle(hEvent);
```

共有割り込みマッピング

IRQ と SYSINTR 間の 1:1 の関係は、割り込み共有のために、IRQ 用に直接 複数の ISR を登録できないことを意味します。ただし、複数の ISR を間接的にマップできます。割り込みマッピング テーブルは 1 つの IRQ を 1 つの静的 ISR にマップしますが、この ISR 内で NKCallIntChain 関数を呼び出して、LoadIntChainHandler を介して動的に登録された一連の ISR を繰り返すことできます。NKCallIntChain は、共有割り込み用に登録された ISR によって、SYSINTR_CHAIN とは異なる、最初の SYSINTR 値を返します。現在の割り込み原因の適切な SYSINTR を特定すると、静的 ISR はこの論理割り込み識別子をカーネルに渡して、該当する IST イベントの通知を行うことができます。LoadIntChainHandler 関数およびインストール可能 ISR の詳細は、このレッスンで後述します。

ISR および IST 間の通信

ISR および IST は、別の時間に、別のコンテキストで実行されるため、ISR がデータを IST に渡す場合に、物理および仮想メモリ マッピングに特別の注意を払う必要があります。例えば、ISR は個別のバイトを周辺機器から入力バッファにコピーすると、バッファが一杯になるまで SYSINTR_NOP が返されます。ISR は、入力バッファが IST 用に準備できた場合のみ、実際の SYSINTR 値を返します。カーネルは、該当する IST イベントに通知し、IST を実行して、データを処理バッファにコピーします。

このデータ転送を実現する方法としては、.bib ファイルに物理メモリ セクションを保持することです。Config.bib には、シリアルおよびデバッグ ドライバ用のいくつかの例が含まれます。これで、ISR が OALPAtoVA 関数を呼び出して、保持されたメモリ セクションの物理アドレスを仮想アドレスに変換することができます。ISR はカーネル モードで実行されているため、ISR は保持されたメモリにアクセスして、周辺機器からデータをバッファすることができます。それに対し、IST では、MmMapIoSpace をカーネルの外部で呼び出し、物理メモリをプロセス固有仮想アドレスにマップします。MmMapIoSpace は、VirtualAlloc および VirtualCopy を使用して物理メモリを仮想メモリにマップしますが、アドレス マッピング処理をより詳細にコントロールしたい場合は、VirtualAlloc および VirtualCopy を直接呼び出すこともできます。

データを ISR から IST に渡す別の方法としては、デバイス ドライバで AllocPhysMem 関数を使用することで、SDRAM で動的に物理メモリを割り当てる方法です。これは、オンデマンドでカーネルにロードされたインストール可能 ISR に特に役に立ちます。AllocPhysMem は、物理的に連続したメモリ領域を

割り当て物理アドレスを返します (または、割り当てサイズが使用可能でない場合は失敗します)。デバイス ドライバは、ユーザー定義 IO コントロール コードに基づく KernelIoControl への呼び出しで、ISR への物理アドレスに通知します。これで、ISR は、OALPAtoVA 関数を使用して物理アドレスを仮想アドレスに変換し、静的に保持されたメモリ領域について説明したように、IST は MmMapIoSpace や VirtualAlloc および VirtualCopy 関数を使用します。

インストール可能 ISR (IISR)

Windows Embedded CE をカスタム ターゲット デバイスに導入するとき、OAL を可能な限り汎用的に保つことが重要です。そのようにしないと、システムに新しいコンポーネントを追加するたびにコードを変更することが必要になります。柔軟性と適応性を実現するため、Windows Embedded CE はインストール可能 ISR (IISR) をサポートしています。IISR は、新しい周辺機器がプラグ アンド プレイで接続されたときなどに、デバイス ドライバがカーネル領域にオンデマンドでロードできるようにするものです。インストール可能 ISR はまた、複数のハードウェア デバイスが同一の割り込み行を共有したときに、割り込みを処理するソリューションも提供します。ISR アーキテクチャは、インストール可能 ISR のコードを含み、表 6-7 で要約されているエントリ ポイントをエクスポートする、リーン DLL に依存しています。

表 6-7　エクスポートされたインストール可能 ISR DLL 関数

関数	説明
ISRHandler	この関数には、インストール可能割り込みハンドラが含まれています。戻り値は IST の SYSINTR 値で、LoadIntChainHandler 関数への呼び出しでインストール可能 ISR 用に登録された、IRQ への応答として実行します。OAL は、最低でもその IRQ へのチェーンをサポートしている必要があります。これは、処理されていない割り込みが、割り込みが発生したときに別のハンドラ (この場合はインストール可能 ISR) に関連付けられるようにすることを意味しています。
CreateInstance	この関数は、インストール可能 ISR が LoadIntChainHandler 関数を使用してロードされるときに呼び出されます。ISR のインスタンス識別子を返します。
DestroyInstance	この関数は、インストール可能 ISR が FreeIntChainHandler 関数を使用してアンロードされるときに呼び出されます。
IOControl	この関数は、IST から ISR への通信をサポートします。

ノート　汎用インストール可能 ISR (GIISR)
インストール可能 ISR の実装を助けるため、Microsoft は、多くのデバイスの一般的なほとんどすべてのデバイスをカバーする、汎用インストール可能 IIS のサンプルを提供しています。ソース コードは、次のフォルダ %_WINCEROOT%\Public\Common\Oak\Drivers\Giisr にあります。

IISR を登録する

LoadIntChainHandler 関数には、インストール可能 ISR をロードおよび登録するために指定する必要のある 3 のパラメータが必要です。最初のパラメータ (lpFilename) は、ロードする ISR DLL のファイル名を指定します。2 番目のパラメータ (lpszFunctionName) は、割り込みハンドラ関数の名前を指定します。また、3 番目のパラメータ (bIRQ) は、インストール可能 ISR を登録する IRQ 番号を定義します。ハードウェアの切断に対応して、デバイス　ドライバは、FreeIntChainHandler 関数を呼び出すことで、インストール可能 ISR をアンロードすることもできます。

外部依存関係およびインストール可能 ISR

LoadIntChainHandler が ISR DLL をカーネル領域にロードすることに留意するのは重要です。これは、インストール可能 ISR が高レベルのオペレーティング システム API を呼び出すことができず、他の DLL をインポートしたり、暗黙的なリンクを設定することができないことを意味しています。DLL に明示的または暗黙的な他の DLL へのリンクがある場合、または C ランタイム ライブラリを使用している場合、DLL をロードすることはできません。インストール可能 ISR は、完全に自己完結型である必要があります。

インストール可能 ISR が C ランタイム ライブラリや他の DLL に関連付けられないようにするために、次の行を DLL サブプロジェクトのソース ファイルに追加する必要があります。

```
NOMIPS16CODE=1
NOLIBC=1
```

NOLIBC=1 指示子は、C ランタイム ライブラリに関連付けられないようにし、NOMIPS16CODE=1 オプションは、コンパイラ オプション /QRimplicit-import を有効にして、他の DLL への暗黙的なリンクを回避します。この指示子は、パイプライン　ステージがインターロックされないマイクロプロセッサ (Microprocessor without Interlocked Pipeline Stages：MIPS) CPU とは全く関係がないことに注意してください。

レッスン概要

Windows Embedded CE は ISR および IST に依存しており、通常コード実行パスの外部で CPU の検知が必要な、内部および外部ハードウェア コンポーネントによってトリガされる割り込みリクエストに応答します。ISR は、通常、カーネルに直接コンパイルされるか、ブート時にロードされたデバイス ドライバで実装されて、HookInterrupt 呼び出しを介して該当する IRQ に登録されますが、デバイス ドライバがオンデマンドでロードでき、LoadIntChainHandler 呼び出しによって IRQ と関連付けることができる、インストール可能 ISR を ISR DLL に実装することもできます。インストール可能 ISR によって、割り込み共有をサポートすることもできます。例えば、ARM ベース デバイスなど、単一 IRQ のみを持つシステムでは、OEMInterruptHandler 関数を修正することができます。これは、割り込みをトリガしたハードウェア コンポーネントによっては、さらにインストール可能 ISR をロードすることができる静的な ISR です。

ISR DLL が外部コードとの依存関係を持つべきでない以外にも、ISR および インストール可能 ISR にも同様の点が多くあります。割り込みハンドラの主なタスクは、割り込みの原因を特定し、デバイスでその割り込みをマスク オフまたはクリアしてから、割り込み用に IRQ の SYSINTR 値を返して実行する IST についてカーネルに通知することです。Windows Embedded CE は、IRQ を SYSINTR 値と関連付ける割り込みマッピング テーブルを保持します。静的 SYSINTR 値をソースコードで定義したり、ランタイム時にカーネルからリクエスト可能な動的 SYSINTR 値を使用したりすることができます。動的 SYSINTR 値を使用することで、ソリューションの移植性を向上することができます。

SYSINTR 値に従って、カーネルは、該当する割り込みサービス スレッドが WaitForSingleObject 呼び出しから再開できるようにする、IST イベントに通知することができます。ISR の代わりに IST で IRQ を処理するための作業のほとんどを実行することにより、最適なシステム パフォーマンスを実現することができます。これは、システムが、低いまたは同等の優先度の割り込み原因を ISR 実行中のみブロックするためです。ISR が終了すると、カーネルは、現在処理中の割り込みを除く、すべての割り込みをアンマスクします。現在の割り込み原因はブロックされ続けることで、同一デバイスからの新しい割り込みが現在の割り込み処理手順の影響を受けないようにすることができます。IST がその動作を終了すると、IST は InterruptDone を呼び出して新しい割り込み用に準備完了したカーネルに通知する必要があります。これでカーネルの割り込みサポート ハンドラが割り込みコントローラで IRQ を再度有効にできるようになります。

レッスン 5：[デバイス ドライバ] 用電源管理を実装する

第 3 章で説明したように、電源管理は Windows Embedded CE デバイスの重要な要素です。オペレーティング システムには電源管理 (PM.dll) が含まれています。電源管理は、[デバイス マネージャ] と統合されたカーネル コンポーネントで、デバイスが自身の電源状態およびアプリケーションを管理して、特定のデバイスの電源要件を設定できるようにするものです。電源管理の主な目的は、電源消費を最適化し、電源通知やコントロール用に API をシステム コンポーネント、ドライバ、およびアプリケーションに提供することです。電源管理は、どんな特定の電源状態でも電源消費や機能に対する厳密な要件を強制するわけではありませんが、電源管理機能をデバイス ドライバに追加することによって、ターゲット デバイスの電源状態に合致する方式でハードウェア コンポーネントの状態を管理できるようにするのは非常に役立ちます。電源管理、デバイスおよびシステム電源状態、および Windows Embedded CE 6.0 でサポートされる電源管理機能の詳細については、第 3 章「システム プログラミングの実行」を参照してください。

このレッスンを終了すると、以下をマスターできます：

- デバイス ドライバの電源管理インターフェイスを識別する。
- デバイス ドライバで電源管理を実装する。

レッスン時間 (推定)：30 分

電源管理デバイス ドライバ インターフェイス

[電源管理] は電源管理との相互のやり取りを行います。XXX_PowerUp、XXX_PowerDown、および XXX_IOControl 関数によってドライバを有効にします。例えば、デバイス ドライバ自身は、DevicePowerNotify 関数を呼び出すことによって、[電源管理] のデバイス ドライバ レベルの変更をリクエストすることができます。応答として、[電源管理] は XXX_IOControl を IOCTL_POWER_SET の IO コントロール コードとともに呼び出し、リクエストされたデバイス電源状態を渡します。デバイス ドライバにとって、[電源管理] からデバイス自身の電源状態を変更するのは非常に複雑ではありますが、この手順によって一貫した動作と適切なエンド ユーザー エクスペリエンスを確保することができます。デバイスに対してアプリケーションが特定の電源レベルをリクエストすると、[電源管理] は IOCTL_POWER_SET ハンドラを DevicePowerNotify の応答として呼び出しません。それに応じて、デバイス ドライバは DevicePowerNotify への成功した呼び出しの結果 IOCTL_POWER_SET

ハンドラが呼び出されたり、または DevicePowerNotify の任意の呼び出しの結果 IOCTL_POWER_SET が呼び出されたとは見なしません。[電源管理] は、システム電源状態変更中など、多くの状況でデバイス ドライバに通知を送信します。電源管理通知を受け取るため、デバイス ドライバは電源管理が、ドライバのレジストリ サブキーで IClass レジストリ エントリが静的に、または AdvertiseInterface 関数を使用して動的に有効になっていることをアドバタイズする必要があります。

XXX_PowerUp および XXX_PowerDown

XXX_PowerUp および XXX_PowerDown ストリーム インターフェイス関数を使用して、機能の中断および再開を実装することができます。カーネルは、XXX_PowerDown を CPU の電源オフ直前に呼び出し、XXX_PowerUp を電源オン直後に呼び出します。ほとんどのシステム呼び出しが無効になっているこのような状態では、システムは単一スレッド モードで動作することに留意するのは重要です。この理由で、Microsoft は、XXX_PowerUp や XXX_PowerDown の代わりに XXX_IOControl 関数を使用して、機能の中断および再開を含む、電源管理機能の実装を行うことを推奨しています。

注意　電源管理の制約

XXX_PowerUp および XXX_PowerDown 関数によって機能の中断および再開を実装する場合、システム API (WaitForSingleObject など、特にスレッド ブロック API) の呼び出しを回避します。単一スレッド モードでアクティブ スレッドをブロックすると、修復不能なロックアップの原因になります。

IOControl

ストリーム ドライバで電源管理を実装する最適な方法は、電源管理 I/O コントロール コードをドライバの IOControl 関数に追加することです。ドライバの電源管理機能に関する [電源管理] を IClass レジストリ エントリや AdvertiseInterface 関数を介して通知すると、ドライバは該当する通知メッセージを受け取ります。

表 6ミ8 は、[電源管理] がデバイス ドライバに送信して、電源管理関連タスクを実行できるようにする IOCTL の一覧を示しています。.

表 6-8 電源管理 IOCTL

関数	説明
IOCTL_POWER_CAPABILITIES	ドライバがサポートする電源状態に関する情報をリクエストします。ドライバは、依然として、他の電源状態 (D0 から D4) に設定できることに注意してください。
IOCTL_POWER_GET	ドライバの現在の電源状態をリクエストします。
IOCTL_POWER_SET	ドライバの電源状態を設定します。ドライバは、受け取った電源状態番号を実際の設定にマップし、デバイス状態を変更します。新しいデバイス ドライバ電源状態は、出力バッファで [電源管理] に返す必要があります。
IOCTL_POWER_QUERY	[電源管理] は、ドライバがデバイスの状態を変更できるかどうかを確認します。この関数は、あまり重要ではありません。
IOCTL_REGISTER_POWER_RELATIONSHIP	デバイス ドライバが別のデバイス ドライバのプロキシとして登録できるようにし、これにより [電源管理] がすべての電源リクエストをこのデバイス ドライバに渡すようにします。

IClass 電源管理インターフェイス

[電源管理] は、ドライバのレジストリ サブキーで構成してドライバを 1 つまたは複数の デバイス クラス値と関連付けることができる、IClass レジストリ エントリをサポートしています。IClass 値は、グローバル一意識別子 (GUID) で、HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Power\Interfaces レジストリ キーで定義されたインターフェイスを参照します。ドライバ開発者にとって最も重要なインターフェイスは、汎用電源管理のインターフェイスで、GUID {A32942B7-920C-486b-B0E6-92A702A99B35} と関連付けられたデバイスによって有効になります。この GUID をデバイス ドライバの IClass レジストリ エントリに追加することで、電源管理通知のドライバ IOCTL を送信するよう [電源管理] に情報を提供できます (図 6-7 参照)。

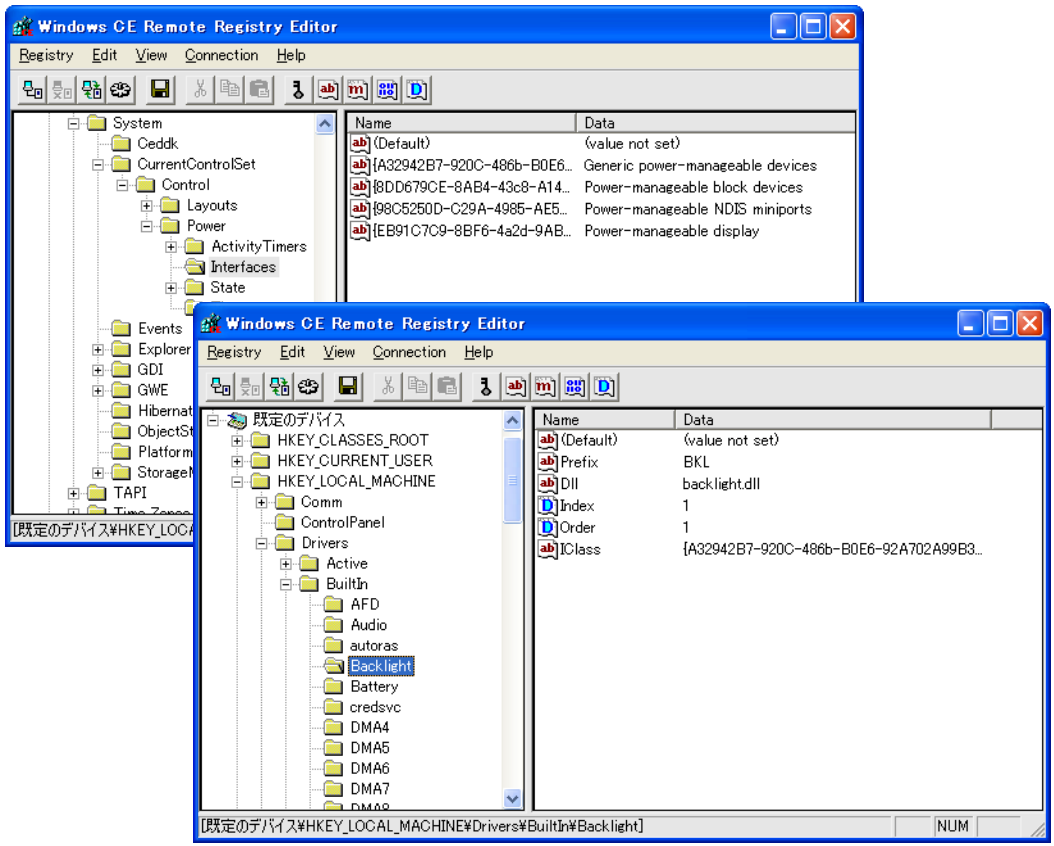


図 6-7　IClass レジストリ エントリを構成して、電源管理通知を受け取る。

詳細情報　電源管理のレジストリ設定

第 3 章「電源管理の実装」のレッスン 5 で説明されているように、レジストリ設定とデバイス クラスを使用して、デバイスの既定の電源状態を構成することもできます。

レッスン概要

Windows Embedded CE で信頼性のある電源管理を確保するため、デバイス ドライバは [電源管理] を使わずに自身の内部電源状態を変更すべきではありません。オペレーティング システム コンポーネント、ドライバ、およびアプリケーションは、DevicePowerNotify 関数を呼び出して、電源状態の変更をリクエストします。それに応じて、[電源管理] は、電源状態変更が現在のシステムの状態と一致した場合に、電源状態変更リクエストをドライバに送信します。電源管

理機能をストリーム ドライバに追加するための推奨方法は、電源管理 IOCTL のサポートを XXX_IOControl 関数に追加することです。[電源管理] は XXX_PowerUp および XXX_PowerDown 関数の呼び出しをシステムが単一スレッド モードのみで動作しているときに呼び出すため、これらの関数は制限された機能のみを提供します。デバイスが IClass レジストリ エントリを介して電源管理インターフェイスのアドバタイズを行うか、AdvertiseInterface を呼び出してサポートされている IOCTL インターフェイスを動的に通知する場合、[電源管理] は IOCTL をデバイス ドライバに電源関連イベントの応答として送信します。

レッスン 6：境界間のマーシャリング データ

Windows Embedded CE 6.0 では、各プロセスには個別の仮想メモリ領域およびメモリ コンテキストがあります。それに応じて、あるプロセスから別のプロセスへのデータのマーシャリングには、プロセスのコピーまたは物理メモリ セクションのマッピングが必要です。Windows Embedded CE 6.0 はほとんどの詳細を処理し、OALPAtoVA および MmMapIoSpace などのシステム関数を提供して、比較的直接的な方法で、物理メモリ アドレスを仮想メモリ アドレスにマップします。ただし、ドライバ開発者は、データ マーシャリングの詳細を理解して、システムの信頼性と安全性を確保する必要があります。埋め込みポインタを有効にし、適切に非同期バッファ アクセスを処理することで、ユーザー アプリケーションがカーネル モード ドライバを利用して、アプリケーションがアクセス可能であるべきではないメモリ領域に処理することができないようにしておくことは重要です。不適切に実装されているカーネル モード ドライバは、悪意のあるアプリケーションにバック ドアを開いて、システム全体を乗っ取らせてしまうことがあります。

このレッスンを終了すると、以下をマスターできます：

- デバイス ドライバでバッファを割り当て、使用する。
- アプリケーションで埋め込みポインタを使用する。
- デバイス ドライバで埋め込みポインタの種類を確認する。

レッスン時間 (推定)：30 分

基盤となるメモリ アクセス

Windows Embedded CE は、図 6-8 に示すように、仮想メモリ コンテキストで動作し、物理メモリを隠します。オペレーティング システムは、仮想アドレスの物理アドレスへの変換および他のメモリ アクセス管理タスクを仮想メモリ マネージャ (VMM) およびプロセッサのメモリ管理ユニット (MMU) に依存しています。

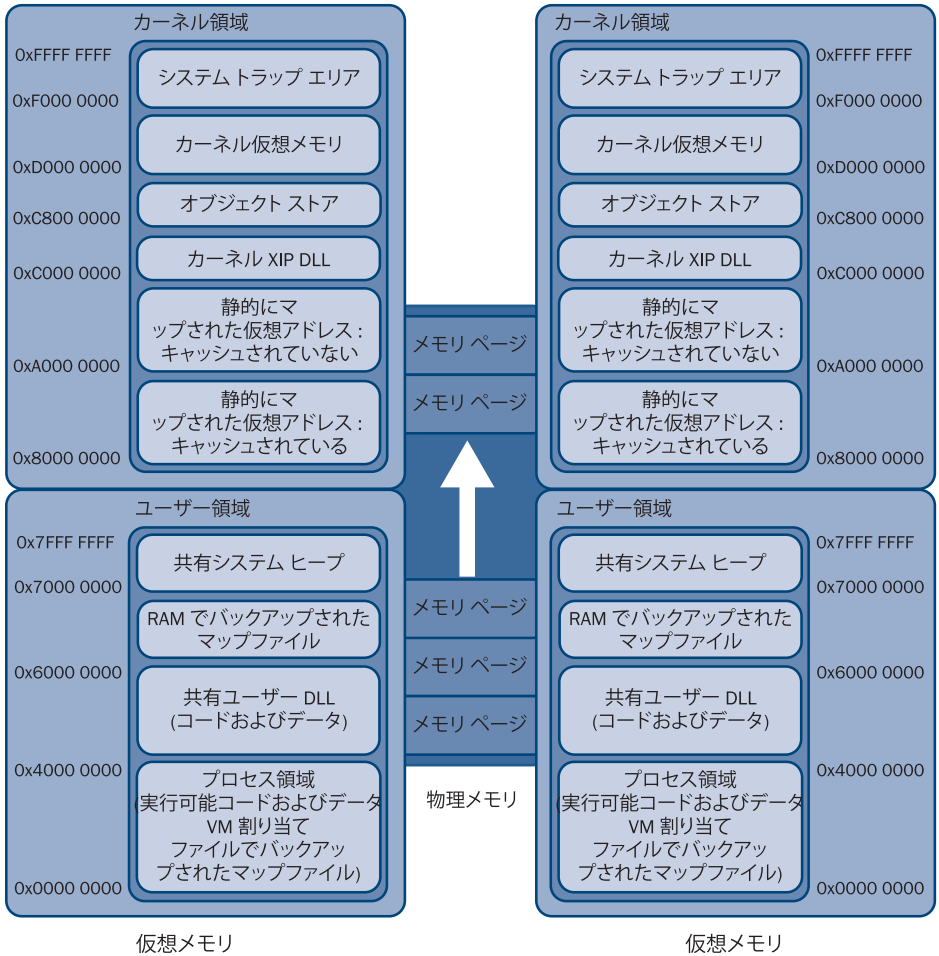


図 6-8　カーネル領域およびユーザー領域の仮想メモリ領域

物理アドレスは、カーネルが MMU を有効にする前の初期化時を除き、CPU によって直接アドレス指定することはできませんが、これは物理メモリにもはやアクセスできないことを意味するわけではありません。実際、完全に割り当てられた仮想メモリ ページは、ターゲット デバイスの実際の物理ページにいくらかマップする必要があります。別個の仮想アドレス領域のプロセスは、同一の物理メモリ領域を使用可能な仮想メモリ領域にマップしてデータを共有するために、1 つの機構のみを必要とします。物理アドレスは、システムで実行されているシステム全体で同一です。仮想アドレスのみが異なります。プロセスごとに物理アドレスを仮想アドレスに変換することで、プロセスは同一の物理メモリ領域にアクセスでき、プロセス境界内でデータを共有できます。

この章で前述したように、ISR などのカーネル モード ルーチンは OALPAtoVA を呼び出して物理アドレス (PA) をキャッシュ済みまたはキャッシュされていない仮想アドレス (VA) にマップできます。OALPAtoVA は、カーネル領域内で物理アドレスを仮想アドレスにマップするため、IST などのユーザー モード プロセスはこの関数を使用できません。カーネル領域は、ユーザー モードではアクセスできません。ただし、IST などの、ユーザー モード プロセスのスレッドは、MmMapIoSpace 関数を呼び出して、カーネル領域で、物理アドレスをページングされていない、キャッシュされた、またはキャッシュされていない仮想アドレスにマップできます。何も一致しなかったか、既存のマッピングが返された場合は、MmMapIoSpace 呼び出しによって、MMU テーブル (TBL) で新しいエントリが作成されます。MmUnmapIoSpace 関数を呼び出すことで、ユーザー モード プロセスはメモリを再度解放することができます。

ノート　物理メモリ アクセスの制約

アプリケーションおよびユーザー モード ドライバは、物理デバイス メモリに直接アクセスすることはできません。ユーザー モード プロセスは、HalTranslateBusAddress を呼び出して、MmMapIoSpace の呼び出し前に、物理システム メモリ アドレスへのバス用の物理デバイス メモリ範囲をマップする必要があります。単一の関数呼び出しでバス アドレスを仮想アドレスに変換するには、TransBusAddrToVirtual　関数を使用します。これは、HalTranslateBusAddress および MmMapIoSpace を呼び出します。

物理メモリの割り当て

メモリの一部を割り当てて、ドライバまたはカーネルで使用することができます。これを行うには、2 つの方法があります。

- 動的、AllocPhysMem 関数を呼び出す　AllocPhysMem は、1 または複数ページの連続物理メモリを割り当てます。ページは、コードがユーザー モードかカーネル モードで実行されているかによって MmMapIoSpace または OALPAtoVA を呼び出して、ユーザー領域で仮想メモリにマップできます。物理メモリはメモリ ページの単位内に割り当てられているため、物理メモリのページよりも少ないページを割り当てることはできません。メモリ ページのサイズは、ハードウェア プラットフォームに依存しています。一般的なページ サイズは 64 KB です。
- 静的、Config.bib ファイルで RESERVED セクションを作成する　BSP フォルダの Config.bib などで、ランタイム イメージの BIB ファイルの MEMORY セクションを使用して、静的に物理メモリを保持することができます。図 6-9 はこのアプローチを示しています。メモリ領域の名前は情報提供目的

で、システムで定義された別のメモリ領域を識別するためだけに使用されます。情報の重要な部分は、アドレス定義および RESERVED キーワードです。これらの設定に基づいて、Windows Embedded CE は、予約領域をシステム メモリから除外することで、周辺機器やデータ転送で DMA に使用できるようにします。システムは予約済みのメモリ領域を使用しないため、アクセス衝突の危険がありません。

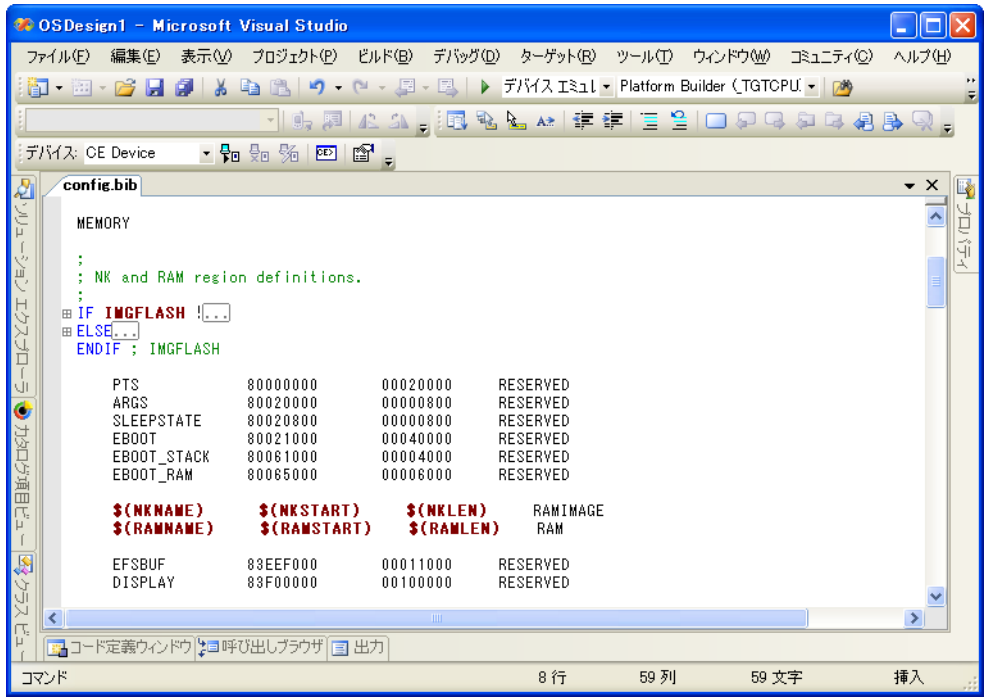


図 6-9　Config.bib ファイルでの予約メモリ領域の定義

アプリケーション呼び出しバッファ

Windows Embedded CE 6.0 では、アプリケーションおよびデバイス ドライバは異なるプロセス領域で実行されます。例えば、[デバイス マネージャ] はストリーム ドライバをカーネル プロセス (Nk.exe) またはユーザー モード ドライバ ホスト プロセス (Udevice.exe) にロードするのに対し、各アプリケーションは自身の個別のプロセス領域で実行されます。1 つのプロセス領域の仮想メモリ アドレスへのポインタは他のプロセス領域では無効であるため、プロセス境界間での通信やデータ転送用に物理メモリの同一バッファ領域へのアクセスを別個のプロセスでサポートさせるには、ポインタ パラメータをマップまたはマーシャルする必要があります。

ポインタ パラメータの使用

ポインタ パラメータは、呼び出し元がパラメータとして関数に渡すことのできるポインタです。DeviceIoControl の lpInBuf および lpOutBuf パラメータはよい例です。アプリケーションは、DeviceIoControl を使用して、直接入力および出力操作を実行できます。入力バッファ (lpInBuf) へのポインタおよび出力バッファ (lpOutBuf) へのポインタは、アプリケーションおよびドライバ間のデータ転送を有効にします。DeviceIoControl は、Winbase.h で次のように宣言されています。

```
WINBASEAPI BOOL WINAPI DeviceIoControl (HANDLE hDevice,
                              DWORD dwIoControlCode,
                              __inout_bcount_opt(nInBufSize)LPVOID lpInBuf,
                              DWORD nInBufSize,
                              __inout_bcount_opt(nOutBufSize) LPVOID lpOutBuf,
                              DWORD nOutBufSize,
                              __out_opt LPDWORD lpBytesReturned,
                              __reserved LPOVERLAPPED lpOverlapped);
```

ポインタ パラメータは、カーネルが自動的にこれらのパラメータの完全アクセス チェックとマーシャリングを実行するため、Windows Embedded CE 6.0 で使用するのに便利です。上記の DeviceIoControl 宣言では、バッファ パラメータ lpInBuf および lpOutBuf は指定されたサイズの入出力 (in/out) パラメータとして定義されているのに対して、lpBytesReturned は出力専用 (out) パラメータとして定義されます。これらの宣言に基づき、カーネルは、アプリケーションがアドレスを読み取り専用メモリ (共有ヒープなど。ユーザー モード プロセスには読み取り専用ですが、カーネルには書き込み可能) に入出力または出力専用バッファ ポインタとして渡さないようにするか、例外をトリガするかを保証することができます。この方法で、Windows Embedded CE 6.0 は、アプリケーションが、カーネル モード ドライバを介して、メモリ領域への昇格されたアクセス権を取得できないことを保証します。これに応じて、ドライバ側では、XXX_IOControl ストリーム インターフェイス関数 (pBufIn および pBufOut) を介して渡されるポインタのどんなアクセス チェックも実行する必要はありません。

埋め込みポインタの使用

埋め込みポインタとは、呼び出し元がメモリ バッファを介して間接的に関数に渡すポインタです。例えば、アプリケーションはポインタを入力バッファ内に保管し、パラメータ ポインタ lpInBuf を介して DeviceIoControl に渡します。カーネルは、自動的にパラメータ ポインタ lpInBuf をチェックおよびマーシャルしますが、システムには、入力バッファ内の埋め込みポインタを識別する方

法はありません。カーネルについて考慮する限り、メモリ バッファはバイナリ データを含むのみです。Windows Embedded CE 6.0 は、ポインタを含むメモリのこのブロックを明示的に指定する機構を提供していません。

埋め込みポインタは、カーネルのアクセス チェックおよびマーシャル ヘルパーをバイパスするため、アクセス チェックと埋め込みポインタのマーシャリングを、それらを使用する前にデバイス ドライバで手動で実行する必要があります。そのようにしないと、悪意のあるユーザー コードが利用して不正なアクションを実行したり、システム全体を損傷したりする可能性のある脆弱性作成を作り出してしまうことがあります。カーネル モード ドライバは、高レベルの特権を有しており、ユーザー モードがアクセスできないシステム メモリにアクセスできます。

呼び出し元プロセスが必要なアクセス権、ポインタのマーシャル機能、およびバッファへのアクセス権を持っていることを確証するには、CeOpenCallerBuffer 関数を呼び出す必要があります。CeOpenCallerBuffer は、呼び出し元がカーネル モードかユーザー モードで実行しているかに基づいてアクセス権を確認し、呼び出し元のバッファの物理メモリに対する新しい仮想アドレスを作成し、オプションで一時ヒープ バッファを割り当てて呼び出しのバッファのコピーを作成します。物理メモリのマッピングには、ドライバ内での新しい仮想アドレス範囲の割り当てが関係するため、ドライバが処理を終えたときに、CeCloseCallerBuffer を呼び出すことを忘れないようにしてください。

バッファの取り扱い

暗黙的 (パラメータ ポインタ) または 明示的 (埋め込みポインタ) なアクセス チェックおよびポインタ マーシャリングを実行すると、デバイス ドライバはバッファにアクセスできるようになります。ただし、バッファへのアクセスは排他的ではありません。デバイス ドライバはバッファからデータを読み取り、データを書き込みますが、図 6-10 で示すように、呼び出し元も同時に読み取りおよび書き込みを行います。例えば、デバイス ドライバがマーシャルされたポインタを呼び出し元のバッファに保持している場合、セキュリティの問題が発生することがあります。アプリケーションの 2 番目のスレッドは、ポインタを操作して、ドライバを介して保護されたメモリ領域にアクセスできます。この理由で、ドライバは、呼び出し元から受け取ったポインタのコピーおよびバッファ サイズ値を常に確認し、埋め込みポインタをローカル変数にコピーして、同期しない修正が発生しないようにする必要があります。

重要　非同期バッファ処理

マーシャル後に、呼び出し元のバッファのポインタを使用すべきではありません。また、呼び出し元のバッファを使用してマーシャルされたポインタまたはドライバ処理に必要な他の変数を保存しないようにします。例えば、バッファ サイズ値をローカル変数にコピーして、呼び出し元がこれらの値を操作してバッファ オーバーランを引き起こすことができないようにします。呼び出し元による非同期修正を回避する 1 つの方法は、CeOpenCallerBuffer を TRUE に設定された ForceDuplicate パラメータとともに呼び出して、呼び出し元のバッファからのデータを一時ヒープ バッファにコピーします。

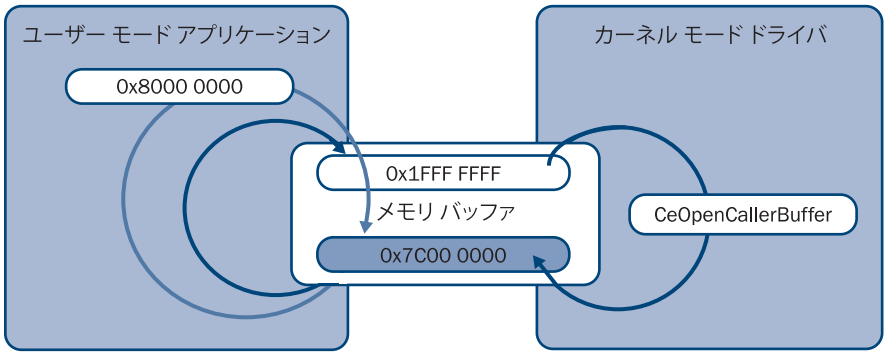


図 6-10　共有バッファでのマーシャルされたポインタの操作

同期アクセス

同期メモリ アクセスは、非同時バッファ アクセスの同意語です。呼び出し元のスレッドは、関数呼び出しが返され (DeviceIoControl など)、ドライバがそのプロセス タスクを実行する間に、バッファがアクセスする呼び出し元プロセスの他のスレッドがなくなるまで待機します。このシナリオでは、デバイス ドライバはパラメータ ポインタおよび埋め込みポインタを、付加的な注意を必要とせずに使用できます (CeOpenCallerBuffer の呼出し後)。

次は、アプリケーションから同期的にバッファにアクセスする例です。このサンプル ソース コードは、ストリーム ドライバの XXX_IOControl 関数からの抜粋です。

```
BOOL SMP_IOControl(DWORD hOpenContext, DWORD dwCode,
                   PBYTE pBufIn, DWORD dwLenIn,
                   PBYTE pBufOut, DWORD dwLenOut,
                   PDWORD pdwActualOut)
{
    BYTE *lpBuff = NULL;

    ...
```

```
    if (dwCode == IOCTL_A_WRITE_FUNCTION)
    {
        // パラメータを確認
        if ( pBufIn == NULL || dwLenIn != sizeof(AN_INPUT_STRUCTURE))
        {
            DEBUGMSG(ZONE_IOCTL, (TEXT("Bad parameters\r\n")));
            return FALSE;
        }

        // 入力バッファにアクセス
        hrMemAccessVal = CeOpenCallerBuffer((PVOID) &lpBuff,
                                            (PVOID) pBufIn,
                                             dwLenIn,
                                             ARG_I_PTR,
                                             FALSE);

        // hrMemAccessVal 値を確認
        // lpBuff を介して pBufIn にアクセス

        ...

        // 必要なくなったときに、バッファを閉じる
        CeCloseCallerBuffer((PVOID)lpBuff, (PVOID)pBufOut,
                            dwLenOut, ARG_I_PTR);

    }

...

}
```

非同期アクセス

非同期バッファ アクセスは、複数の呼び出し元およびドライバ スレッドが、バッファに逐次的または同時にアクセスするとみなします。いずれのアクセス方法も課題があります。逐次アクセス シナリオでは、呼び出し元スレッドはドライバ スレッドがその処理を完了する前に終了してしまうことがあります。マーシャリング ヘルパー関数 CeAllocAsynchronousBuffer を呼び出すことにより、CeOpenCallerBuffer でマーシャルされた後にバッファを再度マーシャルして、ドライバで呼び出し元のアドレス領域が使用できなくなってもバッファを継続して使用できるようにする必要があります。ドライバの処理終了後に、CeFreeAsynchronousBuffer を忘れずに呼び出してください。

デバイス ドライバがカーネルおよびユーザー モードで動作することを確実にするには、次のアプローチを使用して非同期バッファ アクセスをサポートします。

- ポインタ パラメータ　ポインタ パラメータをスカラー DWORD 値として渡してから、CeOpenCallerBuffer および CeAllocAsynchronousBuffer を呼

び出して、アクセス チェックおよびマーシャリングを実行するようにします。ユーザー　モードでポインタ　パラメータに対して CeAllocAsynchronousBuffer を呼び出せないこと、O_PTR or IO_PTR 値の非同期書き込みを実行できないことに注意してください。

- 埋め込みポインタ　埋め込みポインタを　CeOpenCallerBuffer　および CeAllocAsynchronousBuffer に渡して、アクセス チェックおよびマーシャリングを実行します。

同時アクセスの 2 番目のシナリオを実行するには、前述のように、マーシャリング後にバッファの安全なコピーを作成する必要があります。CeOpenCallerBuffer を TRUE に設定された ForceDuplicate パラメータを使用して呼び出し、CeCloseCallerBuffer を呼び出すのが 1 つ目の方法です。もう 1 つの方法は、パラメータ　ポインタによって参照されるバッファ用に、CeAllocDuplicateBuffer および CeFreeDuplicateBuffer を呼び出すことです。ポインタまたはバッファをスタック変数にコピーするか、VirtualAlloc を使用してヒープ メモリを割り当ててから、memcpy を使用して呼び出し元のバッファをコピーする必要があります。安全なコピーを作成していない場合、脆弱性を残してしまい、悪意のあるアプリケーションが利用してシステムを操作する可能性があることに留意してください。

例外処理

非同期バッファ アクセス シナリオで無視すべきでない別の重要な要素は、埋め込みポインタが有効なメモリ アドレスを指定しない可能性があるということです。例えば、アプリケーションは、割り当てられていないまたは予約されたメモリ領域を参照するドライバにポインタを渡すことができるか、非同期にバッファを解放することができます。信頼性のあるシステムを保証し、メモリ リークを回避するため、バッファ アクセス コードを __try フレームおよび任意のクリーンアップ コードで囲んで、__finally ブロックまたは例外ハンドラのメモリ割り当てを解放する必要があります。例外処理の詳細情報については、第 3 章「システム プログラミングの実行」を参照してください。

レッスン概要

Windows Embedded CE 6.0 は、カーネル機能およびドライバ開発者の作業の複雑さを大幅に解消するマーシャリング ヘルパー関数を使用して、アプリケーションとデバイス ドライバ間の内部プロセス通信を促進します。パラメータ ポインタの場合、カーネルは自動的にすべてのチェックとポインタ マーシャリングを実行します。カーネルはドライバに渡されたアプリケーション バッファのコンテンツの評価ができないため、埋め込みポインタのみ特別の注意が必要で

す。同期アクセス シナリオで埋め込みポインタを検証およびマーシャリングすることには、CeOpenCallerBuffer を直接的に呼び出すことも含まれています。ただし、非同期アクセス シナリオでは、ポインタを再度マーシャルするために、CeAllocAsynchronousBuffer への追加呼び出しが必要です。ドライバによってシステムに脆弱性を作成することがないようにするため、バッファを正しく処理し、バッファ コンテンツの安全なコピーを作成して、呼び出し元が値を操作できないようにし、マーシャル後に呼び出し元バッファのポインタまたはバッファ サイズ値を使用しないようにする必要があります。マーシャルされたポインタやドライバ処理に必要な他の変数を決して呼び出し元のバッファに保管しないでください。

レッスン 7：ドライバ移植性の拡張

デバイス ドライバは、オペレーティング システムの柔軟性および移植性を向上するのに役立ちます。理想的には、異なるターゲット デバイスで多様な通信要件を使用して実行するのに、コードの変更を全く必要としません。比較的直接的ないくつかの方法を使用することで、ドライバの移植性や再利用性を実現できます。一般的な手法の 1 つは、パラメータを OAL やドライバにハードコードする代わりに、レジストリに構成設定を保持することです。Windows Embedded CE は、デバイス ドライバ設計を強化できる MDD や PDD に基づく複数層アーキテクチャもサポートしています。また、バスを認識しない方式でドライバを実装して、接続するバス タイプにかかわりなく周辺機器をサポートする別の方法もあります。

このレッスンを終了すると、以下をマスターできます：

- デバイス ドライバの移植性と再利用性を向上するための、レジストリ設定の使用方法を説明する。
- バスを認識しない方法でデバイス ドライバを実装する。

レッスン時間 (推定)：15 分

ドライバのレジストリ設定にアクセスする

デバイス ドライバの移植性および再利用性を向上するため、ドライバのレジストリ サブキーに追加する必要のある、レジストリ エントリを構成することができます。例えば、I/O マップ メモリ アドレスや デバイス ドライバが動的にロードするインストール可能 ISR の設定を定義することができます。デバイス ドライバのレジストリ キーのエントリにアクセスするには、ドライバは、自身の設定が置かれている場所を識別する必要があります。これには、HKEY_LOCAL_MACHINE\Drivers\BuiltIn キーは必要ではありません。ただし、ロードされたドライバのサブキーの HKEY_LOCAL_MACHINE\Drivers\Active キーにあるキー値で、正しいパス情報が使用可能です。[デバイス マネージャ] は、パスをドライバの、LPCTSTR pContext パラメータにある XXX_Init 関数への Drivers\Active サブキーに渡します。次いで、デバイス ドライバは、この LPCTSTR 値を OpenDeviceKey の呼び出しで使用し、デバイスのレジストリ キーへのハンドルを取得します。キー値をドライバの Drivers\Active サブキーから直接読み取ることは必要ではありません。OpenDeviceKey から返されたハンドルは、ドライバのレジストリ キーを指定し、他のレジストリ ハンドルと同様に

使用することができます。最も重要なこととして、もはや必要なくなったときに、ハンドルを閉じることを忘れないでください。

ヒント　XXX_Init 関数およびドライバ設定

XXX_Init 関数は、レジストリで定義されたドライバのすべての構成設定を定義するのに最適です。後続のストリーム関数呼び出しで繰り返しレジストリにアクセスするのではなく、構成情報を XXX_Init 呼び出しの応答として [デバイス マネージャ] によって作成され、返された、デバイス コンテキストに保存するのはよい方法です。

割り込み関連レジストリ設定

デバイス ドライバがインストール可能 ISR をデバイスに用にロードする必要があり、コードの移植性を向上したい場合、レジストリ キーの ISR ハンドラ、IRQ、および SYSINTR 値を登録し、ドライバの初期化時にレジストリからの値を読み取り、IRQ および SYSINTR 値を有効にしてから、LoadIntChainHandler 関数を使用して指定された ISR をインストールすることができます。

表 6-9 に、この目的で構成できるレジストリ エントリを列挙します。DDKReg_GetIsrInfo 関数を呼び出すことで、これらの値を読み出し、LoadIntChainHandler 関数に動的に渡すことができます。デバイス ドライバの割り込み処理に関する詳細は、この章で前述した、レッスン 4「デバイス ドライバに割り込み機構を実装する」を参照してください。

表 6-9　デバイス ドライバ用割り込み関連のレジストリ エントリ

レジストリ エントリ	タイプ	説明
IRQ	REG_DWORD	ドライバ内での IST の設定用に SYSINTR をリクエストするために使用する IRQ を指定します。
SYSINTR	REG_DWORD	SYSINTR 値を指定して、ドライバ内で IST を設定するのに使用します。
IsrDll	REG_SZ	インストール可能 ISR を含む DLL のファイル名です。
IsrHandler	REG_SZ	指定された DLL が提供するインストール可能 ISR のエントリ ポイントを指定します。

メモリ関連レジストリ設定

メモリ関連レジストリ値によって、レジストリを介してデバイスを構成することができます。表 6 ミ 10 に、ドライバが DDKWINDOWINFO 構造で DDKReg_GetWindowInfo を呼び出すことによって取得可能な、メモリ関連レジストリ情報を列挙します。BusTransBusAddrToVirtual 関数を使用することで、メモリ マップ ウィンドウのバス アドレスを物理システム アドレスにマップすることができ、それを MnMapIoSpace を使用して仮想アドレスに変換できます。

表 6-10　デバイス ドライバ用メモリ関連のレジストリ エントリ

レジストリ エントリ	タイプ	説明
IoBase	REG_DWORD	デバイスによって使用される単一のメモリ マップされたウィンドウのバス関連ベースです。
IoLen	REG_DWORD	IoBase で定義されたメモリ マップされたウィンドウの長さを指定します。
MemBase	REG_MULTI_SZ	デバイスによって使用される複数のメモリ マップされたウィンドウのバス関連ベースです。
MemLen	REG_MULTI_SZ	MemBase で定義されたメモリ マップされたメモリ ウィンドウの長さを指定します。

PCI 関連レジストリ設定

標準 PCI デバイス インスタンス情報を使用した DDKPCIINFO 構造を操作するために使用可能な別のレジストリ ヘルパー関数は、DDKReg_GetPciInfo です。表 6ミ11 は、ドライバのレジストリ サブキーで構成可能な PCI 関連設定を列挙します。

表 6-11　デバイス ドライバ用 PCI 関連のレジストリ エントリ

レジストリ エントリ	タイプ	説明
DeviceNumber	REG_DWORD	PCI デバイス番号です。
FunctionNumber	REG_DWORD	デバイスの PCI 機能番号です。多機能 PCI カードの単一機能デバイスを示します。
InstanceIndex	REG_DWORD	デバイスのインスタンス番号です。
DeviceID	REG_DWORD	デバイスのタイプです。

表 6-11　デバイス ドライバ用 PCI 関連のレジストリ エントリ

レジストリ エントリ	タイプ	説明
ProgIF	REG_DWORD	USB OHCI や UHCI などの、レジスタ固有のプログラミング インターフェイスです。
RevisionId	REG_DWORD	デバイスの改訂番号です。
Subclass	REG_DWORD	IDE コントローラなどの、デバイスの基本機能です。
SubSystemId	REG_DWORD	デバイスを使用するカードやサブシステムのタイプです。
SubVendorId	REG_DWORD	デバイスを使用するカードやサブシステムのベンダです。
VendorId	REG_MULTI_SZ	デバイスの製造業者です。

バスを認識しないドライバの開発

インストール可能 ISR、メモリ マップされたウィンドウ、および PCI デバイス インスタンス情報の設定と同様に、GPIO 番号やタイミング構成をレジストリに保持して、バスを認識しないドライバ設計を実現できます。バスを認識しないドライバの基盤となる考え方は、PCI や PCMCIA などの同一のハードウェア チップセットに対して、コード修正することなしに、複数のバスの実装をサポートすることです。

バスを認識しないドライバを実装するには、次の方法があります。

1. ドライバのレジストリ　サブキーですべての必要な構成パラメータを保持し、Windows Embedded CE レジストリ ヘルパー関数 DDKReg_GetIsrInfo、DDKReg_GetWindowInfo、および DDKReg_GetPciInfo を使用して、ドライバの初期化中にこれらの設定を取得します。
2. HalTranslateBusAddress を呼び出して、バス固有アドレスをシステムの物理アドレスに変換してから、MmMapIoSpace を呼び出して、物理アドレスを仮想アドレスにマップします。
3. LoadIntChainHandler 関数を DDKReg_GetIsrInfo から取得された情報を使用して呼び出すことで、ハードウェアのリセット、割り込みのマスク、およびインストール可能 ISR のロードを実行します。

4. RegQueryValueEx を使用することで、レジストリからのインストール可能 ISR の初期化設定をロードし、ユーザー定義 IOCTL のある KernelLibIoControl への呼び出しでインストール可能 ISR に値を渡します。例えば、Windows Embedded CE に含まれている GIISR (Generic Installable Interrupt Service Routine) は、IOCTL_GIISR_INFO ハンドラを使用して GIISR を有効にするインスタンス情報を初期化し、デバイスの割り込みビットが設定されたタイミングを認識して、該当する SYSINTR 値を返します。ソース コードを C:\Wince600\Public\Common\Oak\Drivers\Giisr フォルダで確認できます。
5. CreateThread 関数を呼び出して IST を開始し、割り込みをマスク解除します。

レッスン概要

デバイス ドライバの移植性を向上するため、ドライバのレジストリ サブキーのレジストリ エントリを構成できます。Windows Embedded CE は、DDKReg_GetIsrInfo、DDKReg_GetWindowInfo、および DDKReg_GetPciInfo などの、これらの設定を取得するために使用可能ないくつかのレジストリ ヘルパー関数を提供しています。これらのヘルパー関数は、インストール可能 ISR、メモリ マップされたウィンドウ、および PCI デバイス インスタンス情報の固有の情報を要求しますが、RegQueryValueEx を呼び出して他のレジストリ エントリからの値を取得することもできます。ただし、これらのレジストリ関数を使用するには、OpenDeviceKey を呼び出して、最初にドライバのレジストリ サブキーへのハンドルを取得する必要があります。OpenDeviceKey は、レジストリ パスを待機します。これは、[デバイス マネージャ] が XXX_Init 関数呼び出しでドライバに渡すものです。もはや必要なくなったときに、レジストリ ハンドルを閉じることを忘れないでください。

演習 6：デバイス ドライバの開発

この演習では、メモリに 128 Unicode 文字の文字列を保存し、取得したストリームドライバを実装します。このドライバの基本バージョンはこの本の付属物の中で入手可能です。コードをサブプロジェクトとして OS デザインに追加することのみが必要です。ついで、.bib ファイルおよびレジストリ設定を構成して、ブート時にドライバを自動的にロードし、WCE コンソール アプリケーションを作成してドライバの機能のテストを行います。最後のステップでは、文字列ドライバに電源管理サポートを追加します。

ノート　詳細なステップごとの指示

この演習で提示されているプロシージャを効果的にマスターするために、この本の付属物中のドキュメント「演習 6 のための詳細なステップ バイ ステップ インストラクション」を参照してください。

ランタイムへのストリーム インターフェイス ドライバの追加

1. デバイス エミュレータ BSP を複製し、演習 2「ランタイムイメージのビルドおよび展開」で概説されているように、この BSP に基づいて OS デザインを作成します。
2. 付属 CD の \Labs\StringDriver\String フォルダにある文字列ドライバ ソース コードを %_WINCEROOT%\Platform\<BSPName>\Src\Drivers のパスにある BSP フォルダ内にコピーします。「Drivers」フォルダのお使いのプラットフォームに「String」という名前があります。このフォルダの直下に付属 CD のドライバからの “sources”、“string.c”、“string.def” といったファイルがあることを確認します。スクラッチ領域からドライバを書き込むことも可能ですが、作業例から起動するほうが処理は速くなります。
3. 新しい「String」フォルダの上にある「Drivers」フォルダの Dirs ファイルにエントリを追加し、文字列ドライバをビルド プロセスに含めます。

注意　ビルド オプションに含める

[ソリューション エクスプローラ] で [ビルドに含める] オプションを使用して、文字列ドライバをビルド プロセスに含めないでください。[ソリューション エクスプローラ] は、重要な CESYSGEN 指示子を Dirs ファイルから削除してしまいます。

4. Platform.bib へのエントリを追加して、ラインタイム イメージに、$(_FLATRELEASEDIR) に含まれるビルド文字列ドライバを追加します。ドライバ モジュールに隠しシステム ファイルとしてマークを付けます。

5. 以下の行を Platform.reg に追加し、文字列ドライバの .reg ファイルがランタイム イメージのレジストリに含まれるようにします。

```
#include "$(_TARGETPLATROOT)\SRC\DRIVERS\String\string.reg"
```

6. [ソリューション エクスプローラ] でそのフォルダを右クリックし、[ビルド] をクリックして、文字列ドライバをビルドします。
7. デバッガ モードで新規ランタイム イメージを作成します。

ノート　リリース モードでランタイム イメージをビルドする
ランタイム イメージのリリース バージョンを使用して作業したい場合、ドライバ コードの DEBUGMSG 式を RETAILMSG 式に変更して、ドライバ メッセージを出力する必要があります。

8. フラット　リリース　ディレクトリの生成された　Nk.bin　を開き、HKEY_LOCAL_MACHINE\Drivers\BuiltIn\String サブキーに String.dll およびレジストリ エントリを含めていることを確認して、スタートアップ時にドライバをロードするようにします。
9. [デバイス エミュレータ] で生成されたイメージをロードします。
10. CTRL+ALT+U を押してイメージの開始後に [モジュール] ウィンドウを開くか、Visual Studio で [デバッグ] メニューを開き、[ウィンドウ] をポイントし、[モジュール] を選択します。図 6-11 に示すように、システムですでに string.dll がロードされていることを確認します。

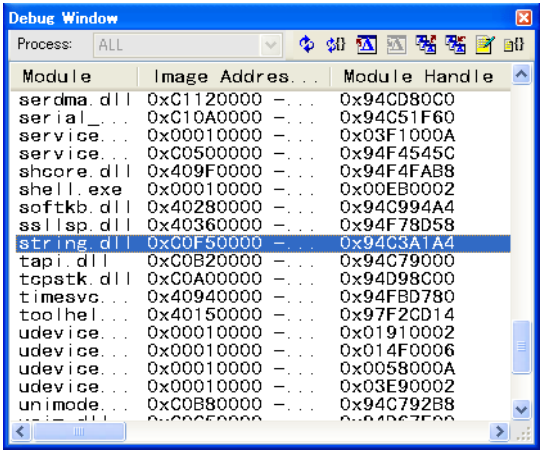


図 6-11　ロードされた文字列ドライバと [モジュール] ウィンドウ

x アプリケーションからドライバにアクセスする

1. 新しい WCE コンソール アプリケーション サブプロジェクトを、Windows Embedded CE サブプロジェクト ウィザードを使用して、OS デザインの一部として作成します。[WCE コンソール アプリケーション] を選択し、テンプレート [シンプルな Windows Embedded CE コンソール アプリケーション] を選択します。
2. ソリューション表示で OS デザイン名を右クリックし、[プロパティ] を選択することで、サブプロジェクト イメージ設定を修正して、イメージからサブプロジェクトを除外します。
3. <windows.h> および <winioctl.h> を含めます。
4. アプリケーションにコードを追加して、CreateFile を使用してドライバのインスタンスを開きます。2 番目の CreateFile パラメータ (dwDesiredAccess) については、GENERIC_READ|GENERIC_WRITE で渡します。4 番目のパラメータ (dwCreationDisposition) については、OPEN_EXISTING で渡します。$device 名前付け規則のあるドライバを開く場合、スラッシュを使用せず、コロンをファイル名の最後に含めないようにしてください。

```
HANDLE hDrv = CreateFile(L"\\$device\\STR1",
                         GENERIC_READ|GENERIC_WRITE,
                         0, 0, OPEN_EXISTING, 0, 0);
```

5. IOCTL ヘッダー ファイル (String_ioctl.h) を文字列ドライバ フォルダから新しいアプリケーションのフォルダにコピーし、ソース コード ファイルに含めます。
6. String_iocontrol.h で定義され、残りのサンプル文字列ドライバに含まれている、PARMS_STRING 構造のインスタンスを宣言し、アプリケーションが文字列をドライバに保存できるようにします。次のコードを使用します。

```
PARMS_STRING stringToStore;
wcscpy_s(stringToStore.szString,
         STR_MAX_STRING_LENGTH,
         L"Hello, driver!");
```

7. DeviceIoControl 呼び出しを、IOCTL_STRING_SET の I/O コントロール コードとともに使用して、この文字列をドライバに格納します。
8. ビルドし、[ターゲット] メニューから [プログラムを実行] を選択して、アプリケーションを実行します。
9. アプリケーションを実行すると、[デバッグ] ウィンドウに [Stored String "Hello, driver!" Successfully] というメッセージが表示されます (図 6-12 参照)。

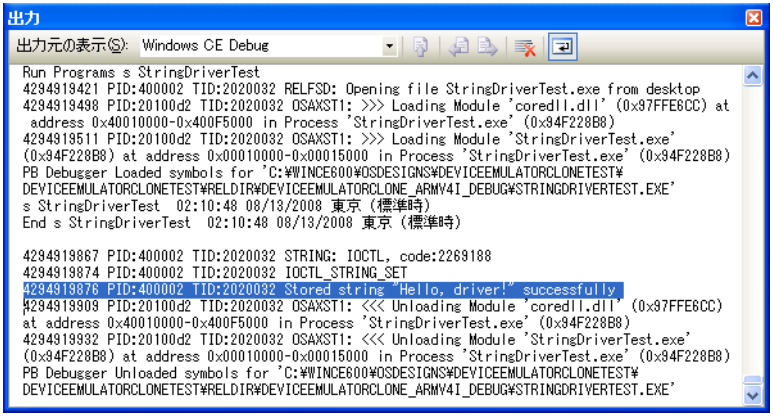


図 6-12　文字列ドライバからのデバッグ メッセージ

x 電源管理サポートの追加

1. デバイス エミュレータをオフにし、接続を切断します。
2. 下記の行を使って、汎用電源管理デバイスの IClass を String.reg の文字列ドライバのレジストリ キーに追加します。

```
"IClass"=multi_sz:"{A32942B7-920C-486b-B0E6-92A702A99B35}"
```

3. 付属 CD の \Labs\StringDriver\Power の下位の StringDriverPowerCode.txt ファイルにある電源管理コードを、文字列ドライバの IOControl 関数に追加して、IOCTL_POWER_GET、IOCTL_POWER_SET、IOCTL_POWER_CAPABILITIES をサポートさせます。
4. 文字列ドライバのデバイス コンテキストにコードを追加し、それが現在の電源状態を保存します。

```
CEDEVICE_POWER_STATE CurrentDx;
```

5. ヘッダー <pm.h> をアプリケーションに追加し、文字列ドライバの名前および異なる電源状態を使用して、SetDevicePower への呼び出しを次のように追加します。

```
SetDevicePower(L"STR1:", POWER_NAME, D2);
```

6. アプリケーションを再度実行して、[電源管理] が文字列ドライバの電源状態を変更したときに、電源状態に関連したデバッグ メッセージを観察します (図 6-13 参照)。

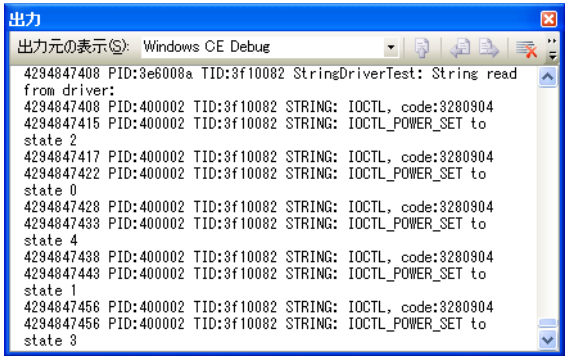


図 6-13 文字列ドライバからの電源管理関連のデバッグ メッセージ

本章のレビュー

Windows Embedded CE 6.0 は、そのデザイン、および ARM-、MIPS-、SH4-、および x86 ベースのボードを、多様なハードウェア構成でサポートする、特殊なモジュラーです。CE カーネルには、コア OS コードおよび OAL やデバイス ドライバに存在するプラットフォーム固有コードが含まれています。実際、デバイス ドライバーは、OS デザインの BSP の重要な部分を占めています。ハードウェアに直接アクセスするよりも、オペレーティング システムが該当するデバイス ドライバをロードしてから、ドライバの提供する機能や I/O サービスを使用するようにします。

Windows Embedded CE デバイス ドライバは、よく知られた API を固守する DLL であるため、オペレーティング システムによってロードすることができます。ネイティブ CE ドライバは CWES とのインターフェイスを提供し、ストリーム ドライバは [デバイス マネージャ] とのインターフェイスを提供します。ストリーム ドライバはストリーム インターフェイス API を実装するため、リソースを特殊ファイル システム リソースとして提供することができます。アプリケーションは、標準ファイル システム API を使用して、これらのドライバと対話することができます。ストリーム インターフェイス API はまた、IOCTL ハンドラのサポートも含んでおり、ドライバを [電源管理] と統合したいときに便利です。例えば、[電源管理] は XXX_IOControl を IOCTL_POWER_SET の IOControl コードとともに呼び出し、リクエストされたデバイス電源状態を渡します。

ネイティブおよびストリーム ドライバは、モノリシックまたは複数層デザインを特徴としています。複数層デザインは、デバイス ドライバのロジックを MDD と PDD 部分に分け、コードの再利用性を向上します。複数層デザインは、ドライバの更新も容易にします。Windows Embedded CE は、ISR および IST に基づく、柔軟性のある割り込み処理アーキテクチャの機能も提供しています。ISR の主なタスクは、割り込みの原因を識別し、実行する IST に関する SYNTINR 値をカーネルに通知します。IST は、時間を要するバッファ コピー プロセスなど、多数の処理を実行します。

一般的に、Windows Embedded CE 6.0 でドライバをロードする 2 つのオプションがあります。ドライバのレジストリ設定を BuiltIn レジストリ キーに追加して、ブート プロセス中にドライバが自動的に起動するようにするか、ActivateDeviceEx への呼び出しで自動的にドライバをロードするようにします。ドライバのレジストリ エントリによっては、カーネル モードまたはユーザー モードでドライバを実行できます。Windows Embedded CE 6.0 には、ユーザー モード ドライバ ホスト プロセスおよびほとんどのカーネル モード ドライバを

コード変更なしでユーザー モードで実行できるようにする、リフレクタ サービスが含まれています。デバイス ドライバは、Windows Embedded CE 6.0 上のアプリケーションとは異なるプロセス領域で実行されるため、データを物理メモリ セクションのマッピングか、通信を可能にするためのコピー プロセスで、データをマーシャルすることが必要です。CeOpenCallerBuffer および CeAllocAsynchronousBuffer を呼び出すことで埋め込みポインタを検証およびマーシャルし、非同期バッファ アクセスを適切に処理することで、ユーザー アプリケーションがカーネル モード ドライバを利用してシステムを操作することができないようにしておくことは必須です。

用語

これらの用語がどういう意味かわかりますか？本書の終わりにある用語集の用語を調べれば、答えをチェックできます。

- IRQ
- SYSINTR
- IST
- ISR
- ユーザー モード
- マーシャリング
- ストリーム インターフェイス
- ネイティブ インターフェイス
- PDD
- MDD
- モノリシック
- バスを認識しない

おすすめの練習方法

本章で示した試験範囲を確実にマスターできるよう、次のタスクを完了させます。

電源管理機能の拡張

文字列ドライバの電源管理コードの開発を続けます。

- 文字列バッファのクリア　デバイス ドライバの電源状態 D3 または D4 にスイッチしたときに、文字列ドライバを修正して、文字列バッファのコンテンツを削除します。
- 電源機能の変更　異なる POWER_CAPABILITIES 値を [電源管理] に返したときに発生する現象を確認します。

IOCTL の増加

さらに IOCTL ハンドラを追加することで、文字列ドライバの機能を拡張します。

- 保存された文字列を反転　IOCTL を追加して、バッファの文字列のコンテンツを反転します。
- 文字列を連結　2 番目の文字列と連結する IOCTL を、バッファのオーバーランなしで保存されている文字列に追加します。
- 埋め込みポインタ　文字列パラメータを文字列へのポインタに置き換え、CeOpenCallerBuffer を使用してアクセスします。

インストール可能 ISR

製品マニュアルを読んで、インストール可能 ISR について学んでください。

- インストール可能 ISR について学ぶ　インストール可能 ISR の詳細については、*http://msdn2.microsoft.com/en-us/library/aa929596.aspx* の Microsoft MSDN Web サイトにある、Windows Embedded CE 6.0 ドキュメントで「Installable ISRs and Device Drivers」のセクションをお読みください。
- インストール可能 ISR の例　インストール可能 ISR の例を確認し、構造を学びます。%_WINCEROOT%\Public\Common\Oak\Drivers\Giisr フォルダにある GIISR コードは、学習に役立ちます。