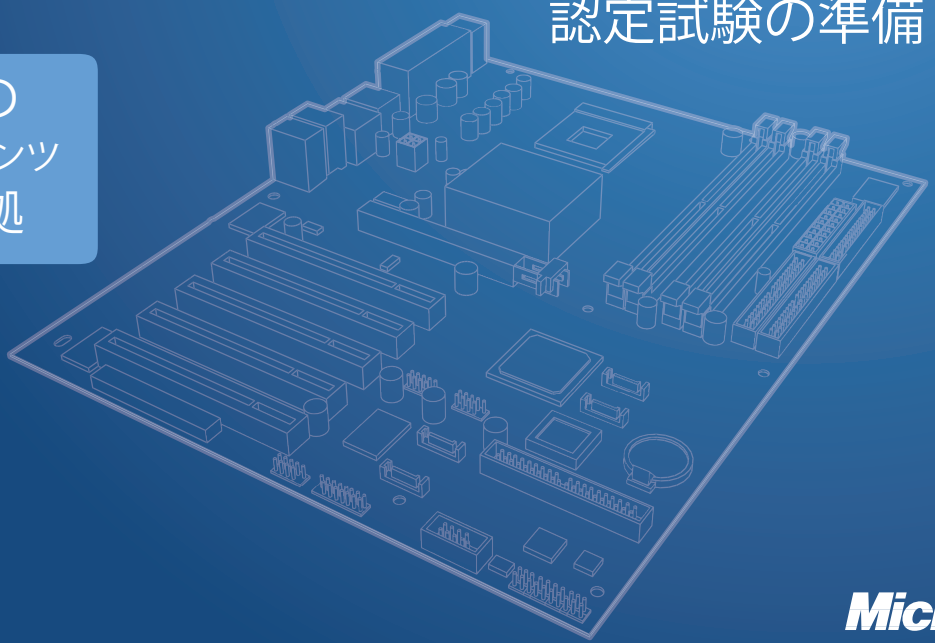
MCTS
Exam 70-571

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

はじめに xi

序文 xvii

1 オペレーティング システム のカスタマイズ 1

2 ランタイム イメージのビルドおよび展開 39

3 システムのプログラミング 85

4 システムのデバッグおよびテスト 153

5 ボード サポート パッケージのカスタマイズ 207

6 デバイス ドライバを開発する 251

用語集 323

索引 327

著者について 347

第 3 章

システムのプログラミング

システム パフォーマンスは、ユーザーの生産性に大きな影響を与えます。パフォーマンスの優劣は、そのままユーザーのデバイスに対する主観を左右します。実際にユーザーがデバイスの有益さを判断する場合、システムのパフォーマンスおよびユーザー インターフェイスの外観と使い心地が、その基準になっていることもまれではありません。インターフェイスが複雑すぎると、ユーザが戸惑ったり、セキュリティ リスクを高めてしまったり、またはユーザが意図せずに想定外の操作をしてしまう原因を作り出すこともあります。不適切な API やマルチスレッド環境のアプリケーション アーキテクチャを使用した場合、パフォーマンスに著しい影響が及びます。パフォーマンスの最適化とシステムのカスタマイズは、ファームウェアを提供する側にとって大きな課題です。この章では、ターゲット デバイスでシステム応答時間を最適化するためのさまざまなツールとベスト プラクティスについて説明します。

本章の試験範囲 :

- システム パフォーマンスの監視と最適化
- システム アプリケーションの実装
- スレッドを使用したプログラミングおよびスレッド同期オブジェクト
- ドライバおよびアプリケーションにおける例外処理の実装
- システム レベルでの電源管理のサポート

始める前に

この章のレッスンを完了するには、以下の予備知識が必要です :

- オペレーティング システムにおけるスケジューラの機能、割り込み、タイマなどのリアルタイムのシステム デザイン の概念を熟知していること。
- 同期オブジェクトを含むマルチスレッド プログラミングの基礎知識があること。
- Microsoft Visual Studio 2005 Service Pack 1 と Windows Embedded CE 6.0 用 Platform Builder がインストールされた開発用コンピュータ。

レッスン 1: システム パフォーマンスの監視 と最適化

パフォーマンスの監視と最適化は、スモール フットプリント デバイスの開発における重要な作業です。複雑なアプリケーションが今後もますます増えていくこと、使いやすさを追求した分リソースも多量に使用するユーザー インターフェイスを求めるユーザのニーズに応えていくために、最適化されたシステム パフォーマンスを実現することは、極めて重要な要件になります。パフォーマンスの最適化には、ファームウェアのアーキテクトとソフトウェア開発者が、それぞれの提供するシステム コンポーネントとアプリケーションでのリソースの使用量を制限し、他のコンポーネントやアプリケーションもリソースを使用できるようにしなければなりません。デバイスのドライバまたはユーザー アプリケーションのいずれの開発においても、処理アルゴリズムを最適化することで無駄なプロセッサ サイクルを減らすことができます。また、効果的なデータ構造はメモリを浪費せずにすみます。ツールはすべてのシステム レベルで用意されており、ドライバ、アプリケーション、その他のコンポーネントの内・外部で発生したパフォーマンスの問題を特定することができます。

このレッスンを終了すると、以下をマスターできます :

- 割り込みサービス ルーチン (ISR) の遅延の特定 .
- Windows Embedded CE システムのパフォーマンスの向上
- システム パフォーマンスのログの取得と分析

レッスン時間 (推定): 20 分

リアルタイム パフォーマンス

ドライバ、アプリケーション、および OEM アダプテーション層 (OAL) のコードは、システムとリアルタイム性能に影響を与えます。Windows Embedded CE はリアルタイムの有無に関わらず使用することができますが、リアルタイム オペレーティング システム (OS) の設定で非リアルタイムのコンポーネントやアプリケーションを使用した場合、システム パフォーマンスが低下することにご注意ください。例えば、デマンド ページング、デバイスの入出力 (I/O)、電源管理などは、リアルタイム デバイス用にデザインされていることに留意してこれらの機能を慎重に使用するようにしてください。

デマンド ページング

デマンド ページングは、RAM の容量に制約があるデバイスに搭載されている複数のプロセス間におけるメモリの共有 を支援します。Windows Embedded CE は、デマンド ページングが有効になっていると、メモリが不足しているアクティ

ブなプロセスから、メモリ ページを破棄および削除します。メモリにすべてのアクティブ プロセスのコードを保持しておくには、オペレーティング システム全体のデマンドページングを無効にするか、またはダイナミック リンクライブラリ (DLL)、デバイス ドライバなどの特定のモジュールを無効にします。

デマンド ページングを無効にするには、次の方法で行います :

- オペレーティング システム Config.bib ファイル を編集し、CONFIG section の ROMFLAGS オプションを設定します。
- DLL LoadLibrary 関数 ではなく、LoadDriver 関数を使用して、DLL をメモリに読み込みます。
- デバイス ドライバ ドライバの DEVFLAGS_LOADLIBRARY フラグ を Flags レジストリ エントリに追加します。このフラグを使用すると、デバイス ドライバ は LoadDriver 関数ではなく、LoadLibrary 関数を使用してドライバを読み込みます。

Windows Embedded CE は、デマンド ページングが無効になっていると、普通にメモリを割り当てて使用しますが、自動的に破棄しません。

システム タイマ

システム タイマは ハードウェア タイマであり 、1 ミリ秒あたり 1 Windows Embedded CE ティックの頻度で、システムティックを生成します。システム スケジューラは、このタイマを使用して、システムで実行すべきスレッドと時間を判断します。スレッド とは、オペレーティング システムの命令を実行するために、プロセッサタイマに割り当てられたプロセス中の最小の実行可能単位のことです。Sleep 関数を使用すると、指定した間隔でスレッドを停止させることができます。Sleep 関数に渡せる最小値は 1 (Sleep(1)) で、スレッドを約 1 ミリ秒停止させます。しかしスリープ時間には、システム タイマの最新のティックとその前のティックの残りも含まれるため、厳密には 1 ミリ秒ではありません。また、スリープ時間は、スレッドの優先順位にも関連付けられています。オペレーティングシステムは、スレッド優先順位に基づいて、プロセッサがスレッドを実行する順位をスケジュールします。そのため、リアルタイムアプリケーション用に正確なタイマが必要な場合は、Sleep関数は使用しないようにします。リアルタイム用には、割り込みを使用した専用のタイマまたはマルチメディアタイマを使用します。

電源管理

電源管理はシステムのパフォーマンスに影響を及ぼします。プロセッサが Idle 電源状態に入状すると、周辺装置またはシステムスケジューラの生成した割り

込みによって、プロセッサは Idle 状態から出状し、以前のコンテキストを復元し、スケジューラを呼び出します。電源コンテキストの切り替え は、時間のかかるプロセスです。Windows Embedded CE の電源管理機能については、Microsoft MSDN ィ Web サイト (*http://msdn2.microsoft.com/en-us/library/aa923906.aspx*) の Windows Embedded CE 6.0 ドキュメントの『Power Management (電源管理)』のセクションを参照してください。

システム メモリ

カーネルはヒープ、プロセス、クリティカル セクション、ミューテックス、イベント、およびセマフォに対するシステム メモリの割り当ておよび管理を行います。しかし、カーネルはこれらのカーネル オブジェクトが解放されても、システム メモリを完全に解放するわけではありません。カーネルはシステム メモリを保持し、次に割り当てる際に再使用します。割り当て済みのメモリを再使用する方が速いため、カーネルは起動プロセス中にシステム メモリ プールを初期化し、プールの使用可能メモリが不足した時にのみ追加メモリを割り当てます。システム パフォーマンスは、プロセスの仮想メモリ、ヒープ、オブジェクト、スタックの使用方法に応じて低下することがあります。

非リアルタイム API

システム API または Graphical Windows Event System (GWES) API を呼び出す場合、API の中にはウィンドウの描画などの非リアルタイム関数に依存しているものもありますので留意してください。非リアルタイム API にコールを転送すると、システム パフォーマンスが著しく低下する場合があります。そのため、リアルタイム アプリケーションの API が、リアルタイムに準拠していることを確認するようにします。また、ファイル システムやハードウェアへのアクセスに使用する API などが、ブロッキング メカニズムとして、ミューテックスやクリティカル セクションを使用してリソースを保護する場合があるため、他の API のパフォーマンスに影響を与えます。

ノート　非リアルタイム API

非リアルタイム API はリアルタイム パフォーマンスに相当な影響を与えますが、残念ながら Win32 ィの API ドキュメントには、このリアルタイム問題について、あまり詳しい情報は記載されていません。実際にパフォーマンス テストを行い、経験を積むことで、適切な関数を選べるようになります。

リアルタイム パフォーマンスの測定ツール

Windows Embedded CE には、パフォーマンスを監視および改善するためのツールが多く含まれており、これらのツールを使用して Win32API がシステムパフォーマンスに及ぼす影響を測定することができます。割り当てられたシステムメモリを解放しないといった、メモリーの非効率な使用を特定する際に有効です。

以下の Windows Embedded CE ツールは、特にシステム コンポーネントとアプリケーションのリアルタイム パフォーマンスを測定する場合に役立ちます :

- ILTiming　割り込みサービス ルーチン (ISR) および割り込みサービス スレッド (IST) の遅延を測定します。
- OSBench　カーネルがカーネル　オブジェクトの管理に費やす時間を追跡し、システム パフォーマンスを測定します。
- リモート パフォーマンス モニタ　メモリ使用量、ネットワーク スループット、その他の面からシステム パフォーマンスを測定します。

割り込み遅延タイミング (ILTiming)

ILTiming は、特に ISR と IST の遅延を測定したい相手先ブランド製造者 (OEM) にとって有益なツールです。特に ILTiming は割り込みの発生後に ISR を呼び出すまでの所要時間 (ISR の遅延) および ISR の終了後から IST が実際に開始するまでの時間 (IST 遅延) を測定することができます。既定の設定により、このツールはシステムハードウェアのティックタイマを使用しますが、他のタイマ (ハイパフォーマンスカウンタ) を使用することもできます。

ノート　ハードウェア タイマの制限
すべてのハードウェア プラットフォームが、ILTiming ツールに必要なタイマ サポートを提供しているわけではありません。.

ILTiming ツールは OAL の OALTimerIntrHandler 関数に依存して、システム ティック割り込みを管理する ISR を実装します。タイマ割り込み処理は、現在の時間を保存しておき、受信のために待機している ILTiming アプリケーション スレッドに SYSINTR_TIMING 割り込みイベントを返します。このスレッドが IST です。ISR が割り込みを受信してから、IST の SYSINTR_TIMING イベントの受信までに経過した時間が IST 遅延であり、ILTiming ツールはこの遅延を測定します。

Windows Embedded CE 6.0 R2 用の Microsoft Platform Builder がインストールされている場合、ILTiming ツールのソースコードは、開発用コンピュータの %_WINCEROOT%\Public\Common\Oak\Utils フォルダにあります。IL Timing ツールは IST 優先順位と型の設定に使用できる複数のコマンドラインパラメータをサポートしており、次の構文で使用します :

iltiming [-i0] [-i1] [-i2] [-i3] [-i4] [-p priority] [-ni] [-t interval] [-n interrupt] [-all] [-o file_name] [-h]

表 3ミ1 は ILTiming コマンドライン パラメータの詳細です。

表 3ミ1 ILTiming パラメータ

コマンドライン パラメータ	説明
-i0	アイドル スレッドなし。このパラメータは -ni パラメータを使用した場合と同じ効果があります。
-i1	スレッド 1 つ分で、実際の処理は行いません。.
-i2	スレッド 1 つ分で、SetThreadPriority (THREAD_PRIORITY_IDLE) 関数を呼び出します。
-i3	スレッド 2 つ分で、SetEvent および WaitForSingleObject を 10 秒間のタイムアウトで交代させます。
-i4	スレッド 2 つ分で、SetEvent および WaitForSingleObject を無限のタイムアウトで交代させます。
-i5	スレッド 1 つ分で、VirtualAlloc (64 KB) または VirtualFree のいずれか、または両方を呼び出します。キャッシュとトランスレーション ルックアサイド バッファ (TLB) をフラッシュします。
-p *priority*	IST プライオリティ (0 から 255) を指定します。既定の設定は 0 (ゼロ) で、プライオリティは最高です。
-ni	非アイドル状態の優先スレッドを指定します。既定の設定は 0 (ゼロ) で、優先順位は最高です。これは -i0 パラメータを使用した場合と同じ効果があります。
-t *interval*	SYSINTR_TIMING タイミングの間隔を、ミリ秒のクロック ティックで指定します。既定の設定は 5 です。
-n *interrupt*	割り込み数を指定します。このパラメータを使用すると、テストの実行期間を指定することができます。既定の設定は １０ です。

表 3 ミ 1 ILTiming パラメータ

コマンドライン パラメータ	説明
-all	すべてのデータを出力します。既定の設定は、サマリのみを出力します。
-o *file_name*	ファイルに出力します。既定の設定は、デバッガのメッセージング ウィンドウに出力します。

ノート アイドル スレッド
ILTiming はアイドル スレッド (コマンドライン パラメータの - i1、-i2、-i3、-i4) を作成し、システム上のアクティビティを生成します。これにより、カーネルは IST を処理する前に終了する必要のある、優先権のない カーネル コールを呼び出します。これはバックグラウンド タスク中のアイドル スレッドにとって有効です。

オペレーティング システム ベンチマーク (OSBench)

OSBench ツールは、カーネルがカーネル オブジェクトの管理に費やす時間を特定することで、システム パフォーマンスを測定することができます。スケジューラに応じて OSBench はスケジューラのパフォーマンス タイミング テストによる方法で、タイミングを測定します。パフォーマンス タイミング テストでは、スレッドの同期などの基本的なカーネル処理にかかる時間を測定します。

OSBench は、次のカーネル処理のタイミング情報を追跡することができます :

- クリティカル セクションを取得または解放する。
- イベントの発生を待機または通知する。
- セマフォまたはミューテックスを作成する。
- スレッドを放棄する。
- システム API を呼び出す。

ノート OSBench によるテスト
異なるシステム構成でのパフォーマンス問題を特定するには、Microsoft Windows CE Test Kit (CETK) などのストレス テスト スイートと OSBench を併用します。

OSBench ツールは、カーネル処理のタイミング サンプルを収集する複数のコマンドライン パラメータをサポートしており、次の構文を使用します :

osbench [-all] [-t test_case] [-list] [-v] [-n number] [-m address] [-o file_name] [-h]

表 3ミ2 は、OSBench コマンドライン パラメータについて説明しています。

表 3ミ2 OSBench パラメータ

コマンドライン パラメータ	説明
-all	すべてのテストを実行 (既定 : -t オプションで指定したテストだけを実行する) : TestId 0: クリティカル セクション TestId 1: イベント のセットとウェイクアップ TestId 2: セマフォの解放と取得 TestId 3: ミューテックス TestId 4: ボランタリー イールド TestId 5:PSL API コール オーバーヘッド TestId 6: インタロック API (decrement、increment、testexchange、exchange)
-t *test_case*	実行するテストの ID (各テスト毎に -t オプションが必要)
-list	テスト ID と説明を一覧表示する
-v	Verbose: 測定に関する追加詳細を表示する
-n *number*	テストあたりのサンプル数 (既定 =100)
-m *address*	マーカーの値の書き込み先仮想アドレス (既定 = <none>)
-o *file_name*	カンマ区切り値 (CSV) ファイルへの出力 (既定 : デバッグ用のみの出力)

OSBench ソースコードをチェックアウトし、テストの内容を特定します。ソースコードは次の場所にあります :

- %_WINCEROOT%\Public\Common\Oak\Utils\Osbench
- %_WINCEROOT%\Public\Common\Oak\Utils\Ob_load

デフォルトで結果はデバッグ出力に送信されますが、カンマ区切り値 (CSV) ファイルにリダイレクトすることもできます。

ノート OSBench の要件

OSBench ツールはシステム タイマを使用します。そのため OAL は OEMInit 関数での QueryPerformanceCounter 関数および QueryPerformanceFrequency 関数の初期化をサポートしなければなりません。

リモート パフォーマンス モニタ

リモート パフォーマンス モニタ アプリケーションは、オペレーティング システム、メモリ使用量、ネットワーク遅延、その他の要素のリアルタイム パフォーマンスを追跡することができます。各システム要素は、使用量、キューの長さ、遅延に関する情報を提供する一連のインジケータと関連付けられています。リモート パフォーマンス モニタはターゲット デバイスで生成されたログ ファイルを分析することができます。

リモート パフォーマンス モニタ アプリケーションはリモートツールです。開発中および既に出荷済みのデバイスとそれをモニタするアプリケーションは、デバイスに接続し、展開する方法があれば動作します。

リモート パフォーマンス モニタは次のオブジェクトを監視します :

- リモート アクセス サーバー (RAS)
- インターネット制御メッセージ プロトコル (ICMP)
- Transport Control Protocol (TCP)
- インターネット プロトコル (IP)
- ユーザー データグラム プロトコル (UDP)
- メモリ
- バッテリ
- システム
- プロセス
- スレッド

このリストは、使用するリモート パフォーマンス モニタ拡張 DLL の実装に応じて増えます。サンプルコードは、% COMMONPROGRAMFILES%\Microsoft Shared\Windows CE Tools\Platman\Sdk\WCE600\Samples\CEPerf フォルダにあります。

Windows ワークステーションのパフォーマンス ツールと同様に、リモート パフォーマンス モニタは、ターゲット デバイスで使用可能なパフォーマンス オブ

ジェクトに基づいて、パフォーマンス チャート、特定のしきい値で発生するアラートの構成、生ログファイルの書き込み、パフォーマンス レポートのコンパイルができます。図 3ミ1 は、パフォーマンス チャートの一例です。

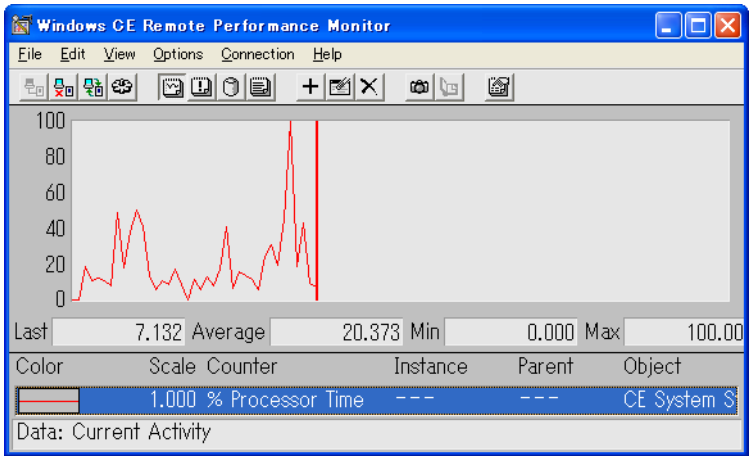


図 3ミ1 リモート パフォーマンス モニタのパフォーマンス チャート

ハードウェアの検証

ILTiming ツール、OSBench、およびリモート パフォーマンス モニタは、パフォーマンスの監視におけるニーズのほとんどに対応していますが、別の方法でシステム パフォーマンス情報を収集しなければならない場合もあります。たとえば、正確な割り込み遅延タイミングを取得したい場合、または使用中のハードウェア プラットフォームがILTimingツールに必要なタイマサポートを提供していない場合などは、プロセッサおよび波形発生器の General Purpose Input/Output (GPIO) インターフェイスに基づいたハードウェア ベースのパフォーマンス測定を行わなければなりません。

波形発生器を GPIO で使用すると、ISR および IST が処理する割り込みが生成される可能性があります。これらの ISR および IST は別の GPIO を使用して、受信した割り込みに応答する波形を生成します。これら 2 つの波形ム発生器から生成された入力波形と ISR または IST が生成した出力波形 ム の間で経過した時間が、割り込みの遅延時間になります。

レッスン概要

Windows Embedded CE は、開発環境から使用することのできるシステム パフォーマンスの測定やリアルタイム デバイス パフォーマンスを検証できる多くのツールを提供しています。ILTiming は割り込み遅延の測定に有益なツールです。OSBench ツールは、カーネルがシステム オブジェクトを管理する方法を分析することができます。リモート パフォーマンス モニタは、開発中あるいは既に出荷済みのデバイスについてのパフォーマンスと統計的なデータを、チャート、ログ、レポートの形で収集する方法を提供します。リモート パフォーマンス モニタには、予め設定したパフォーマンスのしきい値を超えるとアラートを生成する機能があります。これらのツール以外に、遅延とパフォーマンスを測定するためのハードウェア監視機能を使用する他の方法もあります。

レッスン 2: システム アプリケーションの実装

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』で説明したように、Windows Embedded CE　は、広範なスモールフットプリント デバイスのコンポーネント化されたオペレーティング システムおよび開発プラットフォームとして動作します。適用範囲は、ミッションクリティカルな産業用コントローラなどの特定の作業のために制限されたアクセス権を持つデバイスから、パーソナル デジタル アシスタント (PDA) などのすべての設定とアプリケーションを含む、オペレーティング システムすべてへのアクセスを提供するオープン プラットフォームを含んでいます。しかし、実際には、Windows Embedded CE デバイスがユーザーにインターフェイスを提供するには、システム アプリケーションが必要になります。

このレッスンを終了すると、以下をマスターできます :

- 起動時にアプリケーションを開始する。
- 既定のシェルを置き換える。
- シェルをカスタマイズする。

レッスン時間 (推定): 25 分

システム アプリケーションの概要

開発者はシステム アプリケーションとユーザー アプリケーションを区別し、これらのアプリケーションが異なる目的を持っていることを強調します。Windows Embedded CE デバイスの場合、「システム アプリケーション」とは、通常ユーザーとシステムとの間のインターフェイスを提供するアプリケーションを指しています。一方、「ユーザー アプリケーション」とは、ユーザーとアプリケーション固有のロジックとデータとの間のインターフェイスを提供するプログラムを意味します。ユーザー アプリケーション同様、システム アプリケーションもグラフィック インターフェイスまたはコマンドライン インターフェイスを実装することができますが、通常システム アプリケーションはオペレーティング システムの一部として自動的に開始します。

アプリケーションを起動時に開始する方法

Windows Embedded CE の初期化プロセスの一部として、アプリケーションが自動的に開始するよう構成することができます。この機能を設定する場合、アプリケーションをいつ実行するか、つまり Windows Embedded CE がシェル ユーザー インターフェイス (UI) をロードする前か後かによって、いくつかの方法が

あります。例えば、アプリケーションの起動方法を制御するレジストリの設定をいくつか変える方法です。別の一般的な方法は、スタートアップ フォルダにアプリケーションのショートカットを置くことで、標準のシェルがアプリケーションを開始できるようにする方法です。

HKEY_LOCAL_MACHINE\INIT レジストリ キー

Windows Embedded CE レジストリには、デバイス マネージャやグラフィック Windows イベント システム(GWES)などのオペレーティング システム コンポーネントとアプリケーションを起動時に開始できるレジストリ エントリがいくつかあります。これらのレジストリ　　エントリは、図　　3-2　　に示す HKEY_LOCAL_MACHINE\INIT レジストリ キーにあります。ここにランタイム イメージにある独自のアプリケーションを実行させるエントリを作成すると、ターゲット デバイスにわざわざ手動でロードと実行をしなくても、アプリケーションを開始することができます。そのほかにも、アプリケーションを自動的に開始することでソフトウェア開発中のデバッグを容易にすることもできます。

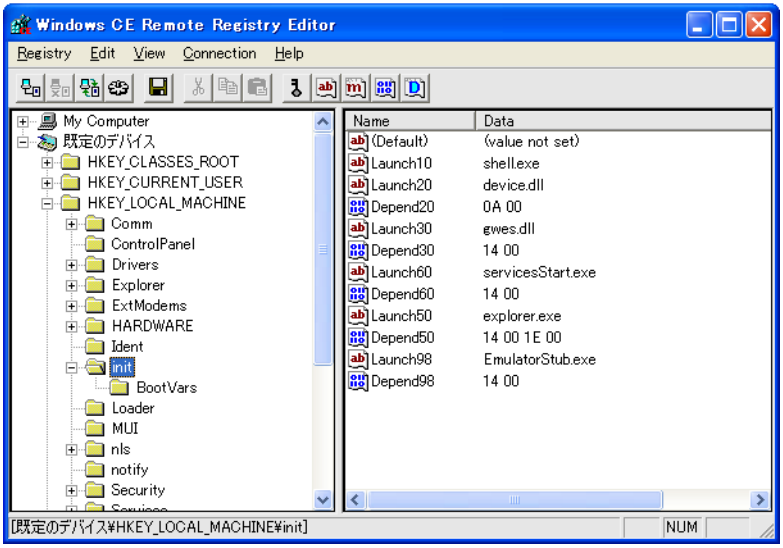


図 3ミ2 HKEY_LOCAL_MACHINE\INIT レジストリ キー

表 3ミ3 は、ランタイム イメージの起動時に、Windows Embedded CE の典型的なコンポーネントを開始するレジストリ エントリの例を 3 つ挙げています。

表 3 ミ 3 スタートアップ レジストリ パラメータ の具体例

場所	HKEY_LOCAL_MACHINE\INIT		
コンポーネント	デバイス マネージャ	GWES	エクスプローラ
バイナリ	Launch20= "Device.dll"	Launch20= "Device.dll"	Launch50= "Explorer.exe"
依存関係	Depend20= hex:0a,00	Depend30= hex:14,00	Depend50= hex:14,00, 1e,00
説明	LaunchXX レジストリ エントリはアプリケーションのバイナリ ファイルを指定し、DependXX レジストリ エントリはアプリケーション間の依存関係を定義します。		

表 3-3 の Launch50 レジストリ エントリを調べると、Windows Embedded CE 標準シェル (Explorer.exe) は、デバイス マネージャ と GWES のプロセス 0x14 (20) とプロセス 0x1E (30) が正常に開始するまでは実行しないことがわかります。DependXX エントリの 16 進法の値は、LaunchXX エントリで指定された 10 進法の起動番号 *XX* を参照しています。

SignalStarted API を実装すると、カーネル マネージが HKEY_LOCAL_MACHINE \INIT レジストリ キーに登録されたすべてのアプリケーション間の依存関係を処理しやすくなります。次のコード スニペットに示したように、その後アプリケーションは SignalStarted 関数を使用して、アプリケーションが起動し、初期化が終了したことをカーネルに知らせます。

```
int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPTSTR lpCmdLine,
    int nCmdShow)
    {
        // Perform initialization here...

        // Initialization complete,
        // call SignalStarted...
        SignalStarted(_wtol(lpCmdLine));

        // Perform application work and eventually exit.
        return 0;
    }
```

依存関係の処理は簡単です。カーネルが Launch レジストリ エントリの起動番号を判断し、シーケンス ID として使用し、lpCmdLine のスタートアップ パラ

メータとして WinMain エントリ ポイントに渡します。アプリケーションは必要な初期化を行ったあと、SignalStarted 関数を呼び出して、カーネルに初期化が終了したことを通知します。SignalStarted 関数は DWORD パラメータを使用するため、SignalStarted コードの _wtol 関数を呼び出し、起動番号を文字列から長正数型に変換します。例えば、デバイス マネージャは SignalStarted の値 20 を、GWES は値 30 をカーネルに渡してからでなければ、Explorer.exe は開始しません。

スタートアップ フォルダ

ターゲット デバイスで標準のシェルを使用している場合、アプリケーションまたはショートカットをデバイスの Windows\Startup フォルダにドロップすることができます。Explorer.exe はこのフォルダを調べ、見つかったすべてのアプリケーションを開始します。

ノート　StartupProcessFolder 関数

Windows\Startup フォルダは、ターゲット デバイスが Windows Embedded CE 標準シェルを実行している場合にのみ使用します。標準シェルを使用していない場合、スタートアップ フォルダと同じように、HKEY_LOCAL_MACHINE\INIT レジストリ キーに登録されたエントリに基づいて起動時に開始するカスタム起動アプリケーションを作成します。スタートアップ フォルダを調べて、その中にあるアプリケーションを開始する方法を示したサンプル コードは、%_WINCEROOT%\Public\Shell\OAK\HPC\Explorer\Main フォルダの Explorer.cpp ファイルを参照してください。StartupProcessFolder 関数を探し、独自のアプリケーションを実装する足がかりにしてください。

Windows Embedded CE 標準シェルは、実行可能ファイルとショートカット ファイルを処理することができます。Windows Embedded CE ショートカット ファイルと Windows XP のショートカット ファイルは、どちらも同じような機能を提供しますが、内容は違います。CE のショートカット ファイルは、.lnk のファイル名拡張子を持つテキスト ファイルで、リンクされたターゲットへのコマンドライン パラメータを含んでおり、次の構文を使用します :

nn# command [optional parameters]

プレース ホルダ *nn* は、27#\Windows\iexplore.exe ミ home のようにシャープ記号 (#) の前に使用される文字の数を示しています。このコマンドの例は Internet Explorer ィ を開始し、ホームページを開きます。希望の .lnk ファイルを作成し、ランタイム イメージに追加したら、次に示す .dat ファイル エントリの要領で、Platform.dat ファイルまたは project.dat ファイルを編集して、.lnk ファイルをスタートアップ フォルダにマップします :

```
Directory("\Windows\Startup"):-File("Home Page.lnk", "\Windows\homepage.lnk")
```

第 2 章はこれらの構成作業について、詳しく説明しています。

ノート　スタートアップ フォルダの制限

スタートアップ フォルダの重要な利点は、このフォルダに置かれたアプリケーションが、初期化と起動のプロセスが正常に終了したことをカーネルに知らせる SignalStarted API を実装する必要がないことです。しかし、これは同時にオペレーティング システムがアプリケーション間の依存関係を管理できない、または特定のスタートアップ シーケンスを強制できないということでもあります。オペレーティング システムは、スタートアップ フォルダのアプリケーションすべてを同時に開始します。

遅延スタートアップ

アプリケーションを自動的に開始する別のオプションとして、サービス ホスト プロセス (Services.exe) を活用する方法があります。Windows Embedded CE には多機能なサービス コントロール マネージャ (SCM) は含まれていませんが、組込みサービスが含まれており、アプリケーションの開始に使用できる Svcstart という名前のサンプル サービスもあります。

Svcstart は、起動プロセスの終了後、すぐに使用可能にならないシステムコンポーネントとサービスに対して依存関係を持つアプリケーションにとって、特に有益なサービスです。例えば、DHCP サーバーからネットワーク インターフェイス カード (NIC) の IP アドレスの取得やファイル システムの初期化に数秒かかるとします。このような状況に対応するため、Svcstart サービスは、アプリケーションを開始するまでの待ち時間を指定する Delay パラメータをサポートしています。Svcstart のサンプルコードは、%_WINCEROOT%\Public\Servers\SDK\Samples\Services\Svcstart フォルダにあります。サンプル コードを Svcstart.dll にコンパイルし、この DLL をランタイム イメージに追加してから、sysgen -p servers svcstart コマンドを実行して、Svcstart サービスをオペレーティング システムに登録します。Services.exe を使用して、Svcstart サービスをロードします。

表 3ミ4 は、アプリケーションを開始するために、Svcstart サービスがサポートしているレジストリ設定を示しています。

表 3ミ4　Svcstart レジストリ パラメータ

場所	HKEY_LOCAL_MACHINE\Software\Microsoft\Svcstart\1
アプリケーション パス	@="iexplore.exe"
コマンドライン パラメータ	Args="-home"

表 3ミ4　Svcstart レジストリ パラメータ

場所	HKEY_LOCAL_MACHINE\Software\Microsoft\Svcstart\1
遅延時間	Delay=dword:4000
説明	ミリ秒で定義された遅延時間が経過すると、指定したコマンドライン パラメータを使用してアプリケーションを開始します。詳しくは、Svcstart.cpp ファイルを参照してください。

Windows Embedded CE シェル

デフォルトで Platform Builder はコマンド プロセッサ シェル、標準シェル、およびシン クライアント シェルの 3 つのシェルを提供して、ターゲット デバイスとユーザー間のインターフェイスを実装しています。これらのシェルはターゲット デバイスとの相互作用のために、それぞれ異なる機能をサポートしています。

コマンド プロセッサ シェル

コマンド プロセッサ シェルは、コマンドが限られたコンソール入出力を提供します。このシェルは、ディスプレイ対応デバイスと、キーボードとディスプレイ画面のないヘッドレス デバイスの両方に対して使用することができます。ディスプレイ対応デバイスの場合、ConsoleWindow コンポーネント (Cmd.exe) を含めておき、コマンド プロセッサ シェルがコマンド プロンプト ウィンドウを使用した入出力を処理できるようにします。一方、ヘッドレス デバイスの入出力は、通常シリアルポートを使用して処理します。

表 3ミ5 は、シリアルポートをコマンド プロセッサ シェルと一緒に使用できるようにするため、ターゲット デバイスで構成する必要のあるレジストリ設定を示しています。

表 3ミ5　コンソール レジストリ パラメータ

場所	HKEY_LOCAL_MACHINE\Drivers\Console	
レジストリ エントリ	OutputTo	COMSpeed
型	REG_DWORD	REG_DWORD
既定値	なし	19600

表 3 ミ 5 コンソール レジストリ パラメータ

場所	HKEY_LOCAL_MACHINE\Drivers\Console	
説明	コマンド プロセッサ シェルが、入出力に使用するシリアルポートを定義します。 ■ この値に - 1 を設定すると、入出力をデバッグ ポートにリダイレクトします。 ■ この値に 0 を設定すると、リダイレクトしません。 ■ 0 以上 10 以下の値に設定すると、入出力はシリアル ポートにリダイレクトされます。	シリアル ポートのデータ転送速度を、bps (1 秒あたりのビット数) で指定します。

Windows Embedded CE 標準シェル

標準シェル は、Windows XP デスクトップと似たようなグラフィック ユーザー インターフェイス (GUI) を提供します。標準シェルの主な目的は、ターゲット デバイスでユーザー アプリケーションを開始し、実行することです。このシェルには、ユーザーがアプリケーションウィンドウを切り替えることができるスタート メニューとタスク バーを備えたデスクトップを含んでいます。標準シェルには、ネットワーク インターフェイスやシステムの現在時間などの追加情報を表示するシステム通知領域があります。

Visual Studio の OS デザイン ウィザードを使用して OS プロジェクトを作成する際に、Enterprise Terminal デザイン テンプレートを選択すると、Windows Embedded CE 標準シェルは必須のカタログ項目になります。このシェルのソースコードが %_WINCEROOT\Public\Shell\OAK\HPC フォルダにありますので、シェルを複製し、カスタマイズすることができます。カタログ項目を複製し、OS デザインに追加する方法については、第 1 章で説明しています。

シン クライアント シェル

製品ドキュメントで Windows ベースのターミナル (WBT) シェルとも呼ばれているシン クライアント シェルは、ユーザー アプリケーションをローカルで実行しないシン クライアント デバイスの GUI シェルです。Internet Explorer をシン クライアント OS デザインに追加することができますが、ネットワークのほかのユーザー アプリケーションをすべてターミナル サーバーで実行しなければなりません。シン クライアント シェルは、リモート デスクトップ プロトコル (RDP)

を使用して、サーバーに接続し、リモートの Windows デスクトップを表示します。既定により、シン クライアント シェルはリモート デスクトップを全画面表示で表示します。

Taskman

Windows タスク マネージャ (TaskMan) シェル アプリケーションを複製およびカスタマイズすることで、独自のシェルを実装することもできます。%_WINCEROOT%\Public\Wceshellfe\Oak\Taskman フォルダにあるソース コードは、その手掛かりとなるでしょう。

Windows Embedded CE コントロール パネル

コントロール パネルは、システムおよびアプリケーションの構成ツールにアクセスを提供する特殊なリポジトリです。製品ドキュメントでは、これらの構成ツールをアプレットと称し、コントロール パネルに組み込まれていることを示しています。各アプレットは他のアプレットには依存せず、対象にしている特定の目的を果たします。Windows Embedded CE の既存のコントロール パネル アプレットを削除し、独自のアプレットを追加することで、コントロール パネルのコンテンツをカスタマイズすることができます。

コントロール パネルのコンポーネント

コントロール パネルは、次の 3 つのコンポーネントに依存する構成システムです :

- フロントエンド (Control.exe)　このアプリケーションは、ユーザー インターフェイスを表示し、コントロール パネルにあるアプレットの開始を支援します。
- ホスト アプリケーション (Ctlpnl.exe)　コントロール パネルのアプリケーションをロードおよび実行します。
- アプレット　個別の構成ツールで .cpl ファイル形式です。コントロール パネルのユーザー インターフェイスには、アイコンと名前が表示されます。

Windows Embedded CE コントロール パネルの実装については、%_WINCEROOT%\Public\Wceshellfe\Oak\Ctlpnl フォルダのソース コードを参照してください。コントロール パネルのコードを複製し、カスタマイズして独自のバージョンのコントロール パネルを実装することができます。

コントロール パネル アプレットの実装

既に説明したように、コントロール パネル アプレットは、システム コンポーネントまたはユーザー アプリケーションの構成ツールであり、ターゲット デバイ

スの Windows フォルダにある .cpl ファイルです。本質的に .cpl ファイルは、CplApplet API. を実装する DLL です。1 つの .cpl ファイルに複数のコントロール パネル アプリケーションを含めることができますが、2 つのアプレットを複数の .cpl ファイルにまたがらせることはできません。すべての .cpl ファイルが CPlAppletAPI を実装することから、ユーザー インターフェイスに使用可能なアプレットを表示するために、Control.exe が実装したアプリケーションの詳細情報を起動時に取得するプロセスは簡単です。Control.exe は Windows フォルダにあるすべての .cpl ファイルを列挙し、各ファイルの CPlApplet 関数を呼び出す必要があるだけです。

DLL の性質と CPlApplet API の要件によると、.cpl ファイルは次の 2 つのパブリック エントリ ポイントを実装しなければなりません :

- BOOL APIENTRY DllMain(HANDLE hModule, DWORD ul_reason_for_call, LPVOID lpReserved)　DLL の初期化に使用します。システムは DllMain を呼び出し DLL をロードします。初期化が正常に行われると DLL は true を返し、失敗した場合は false を返します。
- LONG CALLBACK CPlApplet(HWND hwndCPL, UINT message, LPARAM lParam1, LPARAM lParam2)　コールバック関数です。コントロール パネルのアプレットに対し、アクションを実行するためのエントリ ポイントとしての役目を果たします。

ノート　DLL エントリ ポイント

DllMain および CPlApplet エントリ ポイントをエクスポートし、コントロール パネル アプリケーションがこれらの関数にアクセスできるようにする必要があります。エクスポートしない関数は、DLL にとってプライベート エントリ ポイントになります。C インターフェイスをサポートするには、関数の定義が export "C" { } block にあることを確認してください。

コントロール パネルは CPlApplet 関数を呼び出してアプレットを初期化し、情報を取得し、ユーザーのアクションについての情報を提供し、アプレットをアップロードします。全機能を装備した CPlApplet インターフェイスを実装するには、アプレットは表 3-6 に示す複数のコントロール パネル メッセージをサポートしなければなりません :

表 3 ミ 6 コントロール パネル メッセージ

コントロール パネル メッセージ	説明
CPL_INIT	コントロール パネルはこのメッセージを送信して、アプレットのグローバルな初期化を行います。メモリの初期化は、この時点で行われる典型的なタスクの 1 つです。
CPL_GETCOUNT	コントロール パネルはこのメッセージを送信して、.cpl ファイルに実装されたコントロール パネル アプリケーションの数を判断します。
CPL_NEWINQUIRE	コントロール パネルは、CPL_GETCOUNT が指定したコントロール パネル アプリケーションすべてに対して、このメッセージを送信します。この時点で、コントロール パネル アプリケーションは NEWCPLINFO 構造を返して、コントロール パネルのユーザー インターフェイスに表示されるアイコンと名前を指定します。
CPL_DBLCLK	ユーザーがコントロール パネル ユーザー インターフェイスのアプレットをダブルクリックすると、このメッセージが送信されます。
CPL_STOP	コントロール パネルは、CPL_GETCOUNT で指定された各インスタンスに対して 1 度、このメッセージを送信します。
CPL_EXIT	コントロール パネルは、システムが DLL を解放する前にアプレットに CPL_GETCOUNT が指定したコントロール パネル アプリケーションすべてに対して 1 度、このメッセージを送信します。

ノート NEWCPLINFO 情報

.cpl ファイルに組み込まれたリソースには、実装するコントロール パネル アプレットそれぞれの NEWCPLINFO 情報 があり、CPL_NEWINQUIRE メッセージの応答として返されたアプレットのアイコン、名前、および説明のローカライズを容易にしています。

コントロール パネル アプレットの作成

コントロール パネル アプレットを作成し、対応する .cpl ファイルを生成するには、アプレット サブプロジェクトのソース コード フォルダを探し、次の CPL ビルド ディレクティブを、ソース ファイルの最後尾に追加します :

```
CPL=1
```

さらに図 3-3 に示すように、Control Panel ヘッダー ファイルへのパスを、Visual Studio のアプレット サブプロジェクト設定の [C/C++] タブのインクルード ディレクトリ エントリに追加しなければなりません :

$(_PROJECTROOT)\CESysgen\Oak\Inc

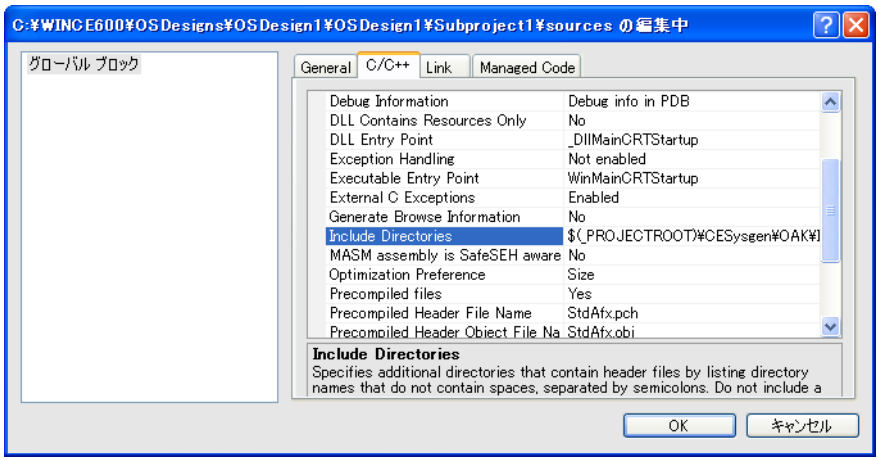


図 3-3 コントロール パネル アプレットのインクルード ディレクトリ

キオスク モードの有効化

医療モニタ機器、自動預金受払機 (ATM)、または産業用制御装置 などの Windows Embedded CE デバイスの多くは、1 つのタスク専用デバイスです。標準のグラフィック シェルは、これらのデバイスには有益ではありません。標準シェルを削除すると、コントロール パネルの構成設定へのアクセスが制限されるだけでなく、ユーザーが追加のアプリケーションを開始できないよう保護します。その結果、デバイスはシェルアクセスを使用せず、ターゲット デバイスの特別な目的に従ってアプリケーションを直接開くキオスク モードで動作します。

Windows Embedded CE のキオスク アプリケーションは、ネイティブ コードまたはマネージ コードで開発されます。唯一の要件は、このアプリケーションを標準のシェル (Explorer.exe) の代わりに開始することです。その後システムはブラック シェルを開始します。つまり、デバイスで実行しているシェル アプリケーションはありません。この構成を実装するには、HKEY_LOCAL_MACHINE\Init キーのレジストリ エントリを構成する必要があるだけです。この章の前半で説明したように、Explorer.exe の LaunchXX エントリは Launch50 です。表 3-7 に示すように、Explorer.exe をカスタム キオス

ク アプリケーションと置き換えると終了したことになりますが、アプリケーションの依存関係を適切に管理するには、カスタム キオスク アプリケーションは、カーネルの SignalStarted API を実装する必要があることに留意してください。.

表 3-7　スタートアップ レジストリ パラメータ の具体例

場所	HKEY_LOCAL_MACHINE\INIT
コンポーネント	カスタム キオスク アプリケーション
バイナリ	Launch50="myKioskApp.exe"
依存関係	Depend50=hex:14,00, 1e,00
説明	キオスク モードを有効にするには、デバイス レジストリにある Explorer.exe の Launch50 エントリを、カスタム キオスク アプリケーションをポイントするエントリに置き換えます。

ノート　マネージ アプリケーションのキオスク モード

標準シェルの代わりにマネージ アプリケーションを実行するには、バイナリ ファイルをランタイム イメージに含め、マネージ アプリケーションの .bib ファイルを編集します。特に、システムの FILES セクションにあるバイナリ ファイルを定義して、共通言語ランタイム (CLR) の内部にあるアプリケーションをロードします。

レッスン概要

Windows Embedded CE は、アイテムとカスタマイズ可能な関数を幅広く備えた、コンポーネント化されたオペレーティング システムです。その機能の 1 つは、起動時にアプリケーションを自動的に開始するよう構成でき、特にインストールや構成ツールにとって有益です。コントロール パネルに独自のアプレットを追加し、カスタマイズすることができます。CPlApplet API に準拠した DLL であるカスタム .cpl ファイルを実装し、コントロール パネルがアプレットを呼び出すことができます。ATM、券売機、医療モニタ機器、空港のチェックイン端末、または産業用制御装置などの特殊な目的のデバイスは、標準シェルを独自のキオスク アプリケーションで置換することで、ユーザー環境を更にカスタマイズすることができます。Windows Embedded CE オペレーティング システムのコード ベースまたは開始プロセスを、カスタマイズする必要はありません。単にタスクを既定の Launch50 レジストリ エントリを、標準またはマネージ コード アプリケーションをポイントするカスタムの Launch50 エントリに交換するだけで、キオスク モードを有効にできます。

レッスン 3：スレッドおよびスレッド同期の実装

Windows Embedded CE はマルチスレッドの OS です。この処理モデルはプロセスの中に複数のスレッドを含めることができるため、UNIX ベースの埋め込み OS とは異なります。マルチスレッドのアプリケーションやドライバを実装・デバッグし、対象デバイスにおいて最適なシステム パフォーマンスを達成するため、単一のプロセス内およびプロセス間でこれらのスレッドをいかにして管理し、スケジュール化し、同期するか、把握する必要があります。

このレッスンを終了すると、以下をマスターできます：

- スレッドを作成・停止する。
- スレッド優先度を管理する。
- 複数のスレッドを同期する。
- スレッド同期に関する問題をデバッグする。

レッスン時間 (推定): 45 分

プロセスとスレッド

プロセスはアプリケーションの一つの例です。そこには処理コンテキストがあり、その中には仮想アドレス スペース、実行可能コード、システム オブジェクトへのオープン ハンドル、セキュリティ コンテキスト、一意なプロセス ID、および環境変数が含まれます。また、実行のプライマリ スレッドもあります。Windows プロセスでは、スレッドは追加スレッドを作成できます。プロセスごとにハードコードされた最大スレッド数はありません。各スレッドはメモリを使用し、物理メモリはプラットフォーム上に限定されるので、最大スレッド数は利用可能なメモリリソースに左右されます。Windows Embedded CE 上の最大プロセス数は 32,000 に限定されます。

Windows Embedded CE 上のスレッド スケジュール化

Windows Embedded CE は、さまざまなプロセスから複数のスレッドを同時に実行するプリエンプティブ マルチタスクをサポートします。Windows Embedded CE は優先度に従い、スレッドのスケジュール化を実行します。システム上の各スレッドは 0 から 255 までの優先度を付けられます。優先度 0 が最も高くなります。スケジューラは優先度リストを管理し、スレッドを選択し、ラウンドロビン配置におけるスレッド優先度に従い、次を実行します。優先度の同じスレッドはランダムに連続して実行します。スレッドのスケジュール化はタイムスライス アルゴリズムによる点に留意する必要があります。各スレッドは限られた

時間に対してのみ実行できます。スレッドが実行可能な最大タイムスライスは、*クォンタム (quantum)* と呼ばれます。クォンタムが経過した場合、スケジューラはスレッドを中断し、リスト内の次のスレッドを再開します。

アプリケーションはそのニーズに従ってスレッドのスケジューリングを行う場合、スレッドごとにクォンタムを設定することができます。しかし、スケジュールは、最初に実行する優先度の高いスレッドを選択するので、スレッドに対しクォンタムを変更しても、優先度の高いスレッドに影響を及ぼすことはありません。優先度の高いスレッドが実行可能になった場合、スケジューラは優先度の低いスレッドに関し、それらのタイムスライス内で中断します。

プロセス管理 API

Windows Embedded CE は、コア Win32 API の一部として、いくつかのプロセス管理機能を含みます。プロセスの作成および終了に役立つ 3 つの重要な機能が表 3ミ8 に記載されています。

表 3ミ8　プロセス管理機能

機能	説明
CreateProcess	新しいシステムを開始する
ExitProcess	クリーンアップおよびアンロード DLL を使ってプロセスを終了する
TerminateProcess	クリーンアップおよびアンロード DLL を使わずにプロセスを終了する

ノート　プロセス管理 API
プロセス管理機能および API ドキュメント全文に関する詳細は、Core OS Reference for Windows Mobile ィ 6 および Windows Embedded CE 6.0 を参照してください。次の Microsoft MSDN サイトでご覧になれます。http://msdn2.microsoft.com/en-us/library/aa910709.aspx.

スレッド管理 API

各プロセスには少なくとも、プライマリ スレッドと呼ばれるスレッドが一つあります。これはプロセスのメイン スレッドで、このスレッドを終了させることで、プロセスも終了させることを意味します。また、プライマリ スレッドは、ワーカー スレッドのように、同時計算を行い、他の処理タスクを行うためのス

レッドを追加で作成できます。これらの追加スレッドは必要に応じ、コア Win32 API を使うことでさらにスレッドを作成できます。表 3 ミ 9 は、Windows Embedded CE 上のスレッドを使って動くアプリケーションで使用する最も重要な機能について記載しています。

表 3 ミ 9　スレッド管理機能

機能	説明
CreateThread	新しいスレッドを作成する
ExitThread	スレッドを終了する
TerminateThread	クリーンアップその他のコードを実行せずに特定のスレッドを停止する。スレッドを終了させることでメモリ オブジェクトが放置され、メモリ リークの原因となるため、極端なケースでのみ本機能を使用する
GetExitCodeThread	スレッドの終了コードを戻す
CeSetThreadPriority	スレッド優先度を設定する
CeGetThreadPriority	現在のスレッド優先度を入手する
SuspendThread	スレッドを中断する
ResumeThread	中断したスレッドを再開する
Sleep	指定された時間、スレッドを中断する
SleepTillTick	次のシステムティックまでスレッドを中断する

ノート　スレッド管理 API
スレッド管理機能および API ドキュメント全文に関する詳細は、Core OS Reference for Windows Mobile 6 および Windows Embedded CE 6.0 を参照してください。次の Microsoft MSDN サイトでご覧になれます。http://msdn2.microsoft.com/en-us/library/aa910709.aspx.

スレッドの作成および終了

新たにスレッドを作成する際に使用する CreateThread 機能は、システムがスレッドを作成する方法を管理するいくつかのパラメータや、スレッドが実行する命令を予測します。これらのパラメータの大部分を null またはゼロに設定することは可能ですが、少なくとも、スレッドが実行すべきアプリケーション定義機能にポインタを提供する必要があります。本機能の中から他の機能を呼び

出すことも可能ですが、本機能は典型的にはスレッドに対するコアな処理の命令を定義します。リンカはコンパイル時にコア機能の開始アドレスを決定できなければなりませんので、スタティック リファレンスとしてのコア機能を CreateThread に移すことが重要です。非スタティック機能ポインタを移すことはできません。

次のコードリストは、%_WINCEROOT%\Public\Shell\OAK\HPC\Explorer\Main フォルダ内の Explorer.cpp ファイルからコピーできます。そこにはスレッド作成方法が記載されています。

```
void DoStartupTasks()
{
    HANDLE hThread = NULL;

    // Spin off the thread which registers and watches the font dirs
    hThread = CreateThread(NULL, NULL, FontThread, NULL, 0, NULL);
    if hThread)
    {
        CloseHandle(hThread);
    }

    // Launch all applications in the startup folder
    ProcessStartupFolder();
}
```

本コードは新しいスレッドのコア機能として FontThread を指定します。現在のスレッドはスレッド ハンドルを必要としていませんので、本コードは戻されたスレッド ハンドルをすぐに閉じます。新しいスレッドは現在のスレッドと並行して実行され、コア機能から戻る際に暗黙的に終了します。これは C++ 機能クリーンアップを可能にするため、望ましいスレッド終了方法と言えます。ExitThread を明示的に呼び出す必要はありません。

しかし、コア機能終了前に処理を終了させるスレッド ルーチン内で ExitThread 機能を明示的に呼び出すことは可能です。ExitThread は、現在のスレッドが接続解除していることを示す値を使って、接続された全ての DLL のエントリ ポイントを呼び出した後、現在のスレッドのスタックの割り当てを解除し、現在のスレッドを終了させます。現在のスレッドがプライマリ スレッドである場合、アプリケーション プロセスは終了します。ExitThread は現在のスレッドに作用しますので、スレッド ハンドルを指定する必要はありません。しかし、他のスレッドが GetExitCodeThread 機能を使って取得できる数値終了コードを渡さなければなりません。本プロセスは、終了しようとするスレッドに関し、エラーと理由を特定する際に役立ちます。ExitThread が明示的に呼び出されない場合、終了コードはスレッド機能の戻り値に対応します。GetExitCodeThread が

STILL_ACTIVE という値を戻した場合、スレッドはまだアクティブで実行中です。

実際には避けるべきことですが、TerminateThread 機能を呼び出す以外にスレッドを終了させる方法が無いケースがまれに発生する場合があります。ファイル レコードを破壊する故障中のスレッドにこの機能が必要な場合があります。ファイル システムをフォーマットすることで、コードを開発中にデバッグセッションで TerminateThread を呼び出すことが必要な場合があります。ハンドルを終了させるスレッドと終了コードに渡す必要がありますが、これは GetExitCodeThread 機能を使うことにより、後で取得できます。TerminateThread の呼び出しは、通常の処理の一部であってはなりません。スレッドがスタックし、接続された DLL が放置され、終了したスレッドが持つクリティカル セクションとミューテックスが放棄され、メモリ リークと不安定の原因となります。プロセス シャットダウン手順の一部として TerminateThread を使用しないでください。プロセス内のスレッドは、ExitThread 機能を使うことで暗黙的または明示的に終了できます。

スレッド優先度の管理

各スレッドは 0 から 255 までの優先値があり、プロセス内およびプロセス間の他の全てのスレッドに関連し、どのようにシステムがスレッドを実行するスケジュールを立てるか決定します。Windows Embedded CE では、コア Win32 API は、下記のようにスレッドの優先度を設定する 4 つのスレッド管理機能を含みます。

- 基本優先度レベル　SetThreadPriority および SetThreadPriority 機能を使い、Windows Embedded CE の初期バージョン (0 ミ 7) と互換性のあるレベルでスレッド優先度を管理します。
- 全優先度レベル　CeSetThreadPriority および CeGetThreadPriority 機能を使い、全レベル (0 ミ 255) でスレッド優先度を管理します。

ノート　基本優先度レベル

Windows Embedded CE 初期バージョンの基本優先度レベル 0 から 7 は、CeSetThreadPriority 機能のうち、優先度の最も低い 8 つのレベル、248 から 255 に対しマッピングされます。

スレッド優先度がスレッド間の関係を定義することを念頭に置く必要があります。他の重要なスレッドが低い優先度で実行されている場合、高い優先度を割り当てると、システムに不利益が生じる可能性があります。低めの優先度を割り当てることで、アプリケーションの動作を向上できる可能性があります。異

なる優先度を使ったパフォーマンス テストは、アプリケーションまたはドライバにおけるスレッドに対する最高の優先度レベルを特定する、信頼できる方法です。しかし、256 もの異なる優先度をテストすることは効率的とは言えません。表 3ミ10 に記載されたドライバあるいはアプリケーションの目的によって、スレッドに対する適当な優先度範囲を選択してください。

表 3ミ10　スレッド優先度範囲

範囲	説明
0 ～ 96	リアルタイム ドライバに対して予約済み
97 ～ 152	デフォルト デバイス ドライバが使用
153 ～ 247	リアルタイムで優先度の低いドライバに対し予約済み
248 ～ 255	アプリケーションへの非リアルタイム優先度に対しマッピング

スレッドの中断および再開

システム パフォーマンスについて、時間のかかる初期化ルーチンまたは他の要因に依存する特定の条件付タスクを遅らせることができます。結局、ループを受け取り、要求されるコンポーネントが最終的に使える状態にあるか 1 万回確認するようなことは効率的ではありません。ワーカー スレッドを適切な時間、例えば 10 ミリ秒休止させ、その後依存関係の状態をチェックし、再度 10 ミリ秒休止させるか、条件が許せば処理を継続するアプローチの方が良いです。スレッド内部からの休止機能を使い、スレッドを中断・再開します。また、SuspendThread および ResumeThread 機能を使用し、他のスレッドを通じてスレッドを管理することも可能です。

休止機能は、ミリ秒単位で休止間隔を指定する数値を反映します。実際の休止間隔がこの数値を上回る可能性が高いことを留意する必要があります。休止機能は現在のスレッドのクォンタムの残りを解除し、スケジューラは、指定した間隔の時間が経過し、他に優先度の高いスレッドがなくなるまで本スレッドを他のタイムスライスに渡しません。例えば、機能呼び出し Sleep(0) は 0 ミリ秒の休止間隔を意味するものではありません。Sleep(0) は現在のクォンタムの残りを他のスレッドに対し解除します。スケジューラがスレッドリスト上に優先度が同じか、高いスレッドを他に持っていない場合、現在のスレッドが引き続き実行されるのみです。

Sleep(0) 呼び出しと同様、SleepTillTick 機能は現在のスレッドのクォンタムの残りを解除し、次のシステムティックまでスレッドを中断します。システムティックベースでタスクを同期したい場合、この方法が役に立ちます。

WaitForSingleObject または WaitForMultipleObjects 機能は、別のスレッドあるいは同期オブジェクトにシグナルが送られるまで、スレッドを中断します。例えば、WaitForSingleObject 機能が代わりに有効になった場合、スレッドは、Sleep および GetExitCodeThread 呼び出しを繰り返してループを入力することなく、他のスレッドが終了するのを待つことができます。本アプローチの結果、リソースをより有効に活用し、コードの読みやすさを改善することができます。タイムアウト値をミリ秒単位で WaitForSingleObject あるいは WaitForMultipleObjects 機能に渡すことが可能です。

スレッド管理サンプル コード

下記のコード スニペットは、どのように中断モードでスレッドを作成し、スレッド機能やパラメータを指定し、スレッド優先度を変更し、スレッドを再開し、スレッドがその処理を終了するのを待つか、示しています。最後のステップとして、次のコード スニペットは、エラー コードがスレッド機能から戻ってきたかをチェックする方法について示しています。

```
// Structure used to pass parameters to the thread.
typedef struct
{
    BOOL bStop;
} THREAD_PARAM_T

// Thread function
DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
    // Perform thread actions...

    // Exit the thread.
    return ERROR_SUCCESS;
}

BOOL bRet = FALSE;
THREAD_PARAM_T threadParams;
threadParams.bStop = FALSE;
DWORD dwExitCodeValue = 0;

// Create the thread in suspended mode.
HANDLE hThread = CreateThread(NULL, 0, ThreadProc,
                  (LPVOID) &threadParams,
                  CREATE_SUSPENDED, NULL);
if (hThread == NULL)
{
    // Manage the error...
}
else
{
    // Change the Thread priority.
```

```
    CeSetThreadPriority(hThread, 200);

    // Resume the thread, the new thread will run now.
    ResumeThread(hThread);

    // Perform parallel actions with the current thread...

    // Wait until the new thread exits.
    WaitForSingleObject(hThread, INFINITE);

    // Get the thread exit code
    // to identify the reason for the thread exiting
    // and potentially detect errors
    // if the return value is an error code value.
    bRet = GetExitCodeThread(hThread, &dwExitCodeValue);

    if (bRet && (ERROR_SUCCESS == dwExitCodeValue))
    {
       // Thread exited without errors.
    }
    else
    {
       // Thread exited with an error.
    }

    // Don't forget to close the thread handle
    CloseHandle(hThread);
}
```

スレッド同期

マルチスレッド プログラミングの真骨頂は、デッドロックを避け、リソースに対するアクセスを保護し、スレッド同期を保証することにあります。Windows Embedded CE は、クリティカル セクション、ミューテックス、セマフォ、イベントおよびインターロック機能等、ドライバもしくはアプリケーションのスレッドに対するリソース アクセスを同期するカーネル オブジェクトをいくつか提供します。しかし、オブジェクトの選択は、実行したいタスクによって異なります。

クリティカル セクション

クリティカル セクションは、単一プロセス内でスレッドを同期し、リソースへのアクセスを保護するオブジェクトです。クリティカル セクションによって保護されたリソースへアクセスする際、スレッドは EnterCriticalSection 機能を呼び出します。本機能は、クリティカル セクションが利用可能になるまでスレッドをブロックします。

状況によっては、スレッド実行をブロックすることは効果的でない場合があります。例えば、利用不可能だと思われるオプションのリソースを使いたい場合、EnterCriticalSection 機能を呼び出すことで、オプションのリソースに対し何ら処理を行うことなくスレッドをブロックし、カーネル リソースを使用します。この場合、TryEnterCriticalSection 機能を呼び出すことにより、ブロックすることなくクリティカル セクションを使用した方が効果的です。本機能はクリティカル セクションを取り込もうとし、また、クリティカル セクションが使えない場合はすぐに戻します。そして、スレッドは、ユーザーに入力を促し、あるいは不明なデバイスをプラグインするなど、代替コードパスを継続できます。

EnterCriticalSectionまたはTryEnterCriticalSectionを通じクリティカル セクションを入手した後、スレッドはリソースへの排他的アクセスを持つことになります。現在のスレッドがクリティカル セクション オブジェクトをリリースするための LeaveCriticalSection 機能を呼び出すまで、他のスレッドはこのリソースにアクセスできません。他のメカニズムの間でも、本メカニズムは、スレッドを終了させる際になぜ TerminateThread 機能を使ってはならないかを明らかにしています。TerminateThread 機能はクリーンアップを行いません。終了したスレッドがクリティカル セクションを持っていた場合、ユーザーがアプリケーションを再開するまで、保護されたリソースは使用不可になります。

表 3ミ11 は、スレッド同期目的のクリティカル セクション オブジェクトを使用して作業を行える最も重要な機能について記載しています。

表 3ミ11 クリティカル セクション API

機能	説明
InitializeCriticalSection	クリティカル セクション オブジェクトを作成・開始する
DeleteCriticalSection	クリティカル セクション オブジェクトを破棄する
EnterCriticalSection	クリティカル セクションオブジェクトを取り込む
TryEnterCriticalSection	クリティカル セクション オブジェクトを取り込もうとする
LeaveCriticalSection	クリティカル セクション オブジェクトをリリースする

ミューテックス

クリティカル セクションが単一プロセスに限られる一方、ミューテックスは複数のプロセスの間で共有されるリソースへの排他的アクセスを相互にコーディネートできます。ミューテックスはプロセス間の同期を容易にするカーネル オ

ブジェクトです。CreateMutex 機能を呼び出し、ミューテックスを作成します。名称の無いミューテックスを作成することも可能ですが、スレッドを作成することで、作成時にミューテックス オブジェクトに対する名称を指定できます。他のプロセスにおけるスレッドも CreateMutex 機能を呼び出し、同じ名称を指定できます。しかし、これらを次々と呼び出しても新しいカーネル オブジェクトを作成できませんが、ハンドルを既存のミューテックスに戻すことは可能です。この時点で、別々のプロセスにおけるスレッドは、保護された共有リソースへのアクセスを同期するためのミューテックス オブジェクトを使うことができます。

ミューテックス オブジェクトの状態は、それを所有するスレッドがない場合はシグナルが送られますが、ある場合は送られません。所有権をリクエストする際は、待機機能 WaitForSingleObject または WaitForMultipleObjects のどちらかをスレッドが使わなければなりません。待機間隔でミューテックスが使用不可能になった場合、タイムアウト値を指定し、代替コードパスに沿ってスレッド処理を再開できます。一方、ミューテックスが使用可能になり、現在のスレッドに所有権が許可された場合、他のスレッドに対しミューテックス オブジェクトをリリースするためミューテックスが待機を満たす際には、必ず ReleaseMutex 機能を呼び出してください。スレッドの実行をブロックすることなく、ループ等の場合にスレッドが待機機能を複数回呼び出せますので、この機能は重要です。システムはデッドロック状況を避けるため、所有するスレッドをブロックしませんが、スレッドはミューテックスをリリースするため、待機機能と同じ回数だけ ReleaseMutex 機能を呼び出さなければなりません。

表 3ミ12 は、スレッド同期目的にミューテックス オブジェクトを使用して作業を行える最も重要な機能について記載しています。

表 3ミ12　ミューテックス API

機能	説明
CreateMutex	名称の付いた、あるいは付かないミューテックス オブジェクトを作成し、開始する。プロセス間で共有されるリソースを保護するため、名称の付いたオブジェクトを使わなければならない
CloseHandle	ミューテックス ハンドルを閉じ、ミューテックス オブジェクトへのリファレンスを削除する。カーネルがミューテックス オブジェクトを削除する前に、ミューテックスへのリファレンスを全て個々に削除しなければならない

表 3ミ12 ミューテックス API

機能	説明
WaitForSingleObject	単一のミューテックス オブジェクトの所有権が許可されるまで待機する
WaitForMultipleObjects	単一または複数のミューテックス オブジェクトの所有権が許可されるまで待機する
ReleaseMutex	ミューテックス オブジェクトをリリースする

セマフォ

プロセス内およびプロセス間のリソースへの排他的アクセスを相互に提供できるカーネル オブジェクトに加え、Windows Embedded CE は、単一あるいは複数のスレッドによるリソースへの同時アクセスを可能にするセマフォ オブジェクトも提供します。これらのセマフォ オブジェクトはカウンタをゼロと最大値との間に維持し、リソースにアクセスするスレッドの数をコントロールします。最大値は CreateSemaphore 機能呼び出しに指定されています。

セマフォ カウンタは、同期オブジェクトへ同時にアクセスできるスレッドの数を制限します。システムは、カウンタがゼロになり、非シグナル状態になるまで、スレッドがセマフォ オブジェクトに対し待機を完了させるごとにカウンタの数字を下げ続けます。カウンタはマイナスの値まで数字を下げることはできません（最大 0 まで）。ReleaseSemaphore 機能を呼び出すことで、カウンタの数字を特定の値の分上げ、セマフォ オブジェクトを再度シグナル状態に切り替え戻すセマフォを、所有権を持つスレッドがリリースするまで、いかなる追加スレッドもリソースへのアクセスを取得することはできません。

ミューテックスと同様、複数のオブジェクトが同じセマフォ オブジェクトのハンドルを開き、プロセス間で共有されるリソースにアクセスできます。最初に CreateSemaphore 機能を呼び出すことで、特定の名称の付いたセマフォ オブジェクトを作成できます。また、名称の無いセマフォを作成することもできますが、このようなオブジェクトはプロセス間の同期に対しては使用できません。同じセマフォ名で引き続き CreateSemaphore 機能を呼び出した場合、新しいオブジェクトは作成されませんが、同じセマフォの新しいハンドルを開くことができます。

表 3ミ13 は、スレッド同期目的にセマフォ オブジェクトを使用して作業を行える最も重要な機能について記載しています。

表 3ミ13 セマフォ API

機能	説明
CreateSemaphore	カウンタの数字を使って名称の付いた、あるいは付かないセマフォ オブジェクトを作成し、開始する。プロセス間で共有されるリソースを保護するため、名称の付いたオブジェクトを使う
CloseHandle	セマフォ ハンドルを閉じ、セマフォ オブジェクトへのリファレンスを削除する。カーネルがセマフォ オブジェクトを削除する前に、セマフォへのリファレンスを全て個々に削除しなければならない
WaitForSingleObject	単一のセマフォ オブジェクトの所有権が許可されるまで待機する
WaitForMultipleObjects	単一または複数のセマフォ オブジェクトの所有権が許可されるまで待機する
ReleaseSemaphore	セマフォ オブジェクトをリリースする

イベント

イベント オブジェクトは、スレッドを同期するもう一つのカーネル オブジェクトです。タスクが完了した場合、またはデータが閲覧できる場合、本オブジェクトによって、アプリケーションが他のスレッドをシグナル状態にできるようになります。各イベントは、その状態を特定する、API が使用するシグナル状態・非シグナル状態の情報を持ちます。2 つのタイプのイベント、すなわち手動のイベントおよび自動リセットのイベントは、イベントが予想する動作に従って作成されます。

名称の無いイベントを作成することも可能ですが、スレッドを作成することで、作成時にイベント オブジェクトに対する名称が指定されます。他のプロセスにおけるスレッドが CreateMutex 機能を呼び出し、同じ名称を指定することは可能ですが、連続して呼び出すことで新たにカーネル オブジェクトが作成されるわけではありません。

表 3ミ14 は、スレッド同期目的のイベント オブジェクトに対する最も重要な機能について記載しています。

表 3ミ14 イベント API

機能	説明
CreateEvent	名称の付いた、あるいは付かないイベント オブジェクトを作成し、開始する
SetEvent	イベントをシグナル状態にする（下記参照）
PulseEvent	イベントをパルスおよびシグナル状態にする（下記参照）
ResetEvent	シグナル状態のイベントをリセットする
WaitForSingleObject	イベントがシグナル状態になるまで待機する
WaitForMultipleObjects	単一または複数のイベント オブジェクトによりシグナル状態になるまで待機する
CloseHandle	イベント オブジェクトをリリースする

イベント API の動作は、それが適用されるイベントのタイプによって異なります。手動イベント オブジェクト上で SetEvent を使う際、ResetEvent が明示的に呼び出されるまで、イベントはシグナル状態になります。自動リセットのイベントは、単一の待機スレッドがリリースされるまで、ひたすらシグナル状態になります。単一の待機スレッドについては、非シグナル状態にすぐに戻るまで自動リセットイベント上の PulseEvent 機能を使う場合、リリースされます。手動のスレッドの場合、待機スレッドはリリースされ、すぐに非シグナル状態に戻ります。

インターロック機能

マルチスレッド環境では、スレッドはいつでも割り込むことができ、その後スケジューラによって再開できます。コードやアプリケーションのリソースの部分は、セマフォやイベント、あるいはクリティカル セクションを使って保護できます。アプリケーションによっては、このようなコード 1 行のみを保護するためにこうしたシステム オブジェクトを使った場合、時間がかかることがあります。

```
// Increment variable
dwMyVariable = dwMyVariable + 1;
```

C にある上記サンプル ソースコードは単一の指示ですが、アセンブリ内ではそれ以上の場合があります。この特定の例では、スレッドは操作の途中で中断され、後に再開されますが、同じ変数を使ったスレッドの場合、エラーの出る可

能性があります。操作はアトミックではありません。幸い、Windows Embedded CE 6.0 R2 では、同期オブジェクトを使用せずに値を上下させ、マルチスレッドが安全なアトミック操作において値を加えることが可能です。これはインターロック機能を使って行います。

表 3ミ15 は、アトミックに変数を操作する際に使う最も重要なインターロック機能について記載しています。

表 3ミ15　インターロック API

機能	説明
InterlockedIncrement	32 ビットの変数を加える
InterlockedDecrement	32 ビットの変数を引く
InterlockedExchangeAdd	値にアトミックを加える

スレッド同期に関するトラブルシューティング

マルチスレッド プログラミングにより、ユーザー インターフェイス インタラクションおよびバックグラウンド タスクに対する別々のコード実行単位に基づきソフトウェア ソリューションを構築できます。これはスレッド同期メカニズムを慎重に実行することが必要な、詳細な開発技法です。特にループやサブルーチンで複数の同期オブジェクトを使う場合、デッドロックが起こり得ます。例えば、スレッド One はミューテックス A を所有し、A をリリースする前にミューテックス B を待機します。一方、スレッド Two はミューテックス B をリリースする前にミューテックス A を待機します。それぞれのスレッドはもう一方がリリースするリソースに依存していますので、この状況ではどちらのスレッドも継続できません。特に複数のプロセスからのスレッドが共有リソースにアクセスしている場合、こうした状況に関する検索やトラブルシューティングは困難です。Remote Kernel Tracker ツールは、どのようにスレッドがシステム上でスケジュール化され、デッドロックの検索を可能にするか、特定しています。

Remote Kernel Tracker ツールにより、対象デバイスに関する全てのプロセス、スレッド、スレッド相互作用、および他のシステム アクティビティのモニタが可能になります。本ツールは、%_FLATRELEASEDIR% ディレクトリにおける Celog.clg という名称のファイルでカーネルその他のシステム イベントのログを取る Celog イベント トラッキング システムに依存しています。システム イベントはゾーンによって分類されます。Celog イベント トラッキング システムは、データのログに対する特定のゾーンに重点を置くよう、設計できます。

対象デバイス上で Kernel Independent Transport Layer (KITL) を有効にする場合、Remote Kernel Tracker は CeLog データを視覚化し、スレッドとプロセスの間のインタラクションを分析します。これは図 3ミ4 に記載されています。KITL がデータを直接 Remote Kernel Tracker ツールへ送ると同時に、収集されたデータをオフラインで分析することも可能です。

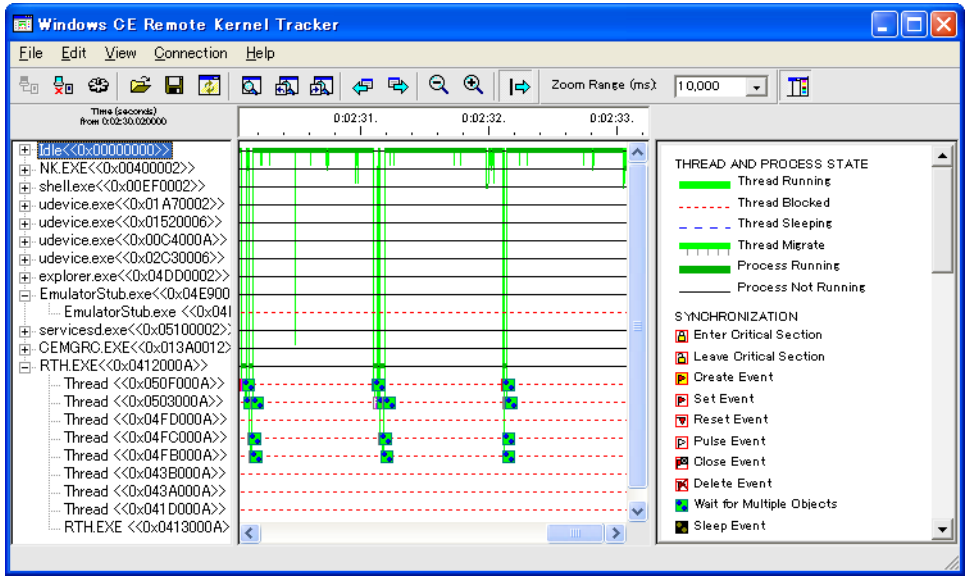


図 3ミ4　Remote Kernel Tracker ツール

ノート　イベント トラッキングおよびフィルタリング

CeLog イベント トラッキングおよび CeLog イベント フィルタリングに関する詳細は、Windows Embedded CE 6.0 の「CeLog Event Tracking Overview（イベントトラッキング概要）」を参照してください。次の Microsoft MSDN サイトでご覧になれます。http://msdn2.microsoft.com/en-us/library/aa935693.aspx.

レッスン概要

Windows Embedded CE は、プロセスやスレッドを作成し、スレッド優先度を 0 から 255 まで割り当て、スレッドを中断し、再開するプロセス管理機能をいくつか提供する、マルチスレッド OS です。特定の時間スレッドを中断する際は Sleep 機能が役立ちますが、別のスレッドまたは同期オブジェクトにシグナルが送られるまで、WaitForSingleObject または WaitForMultipleObjects 機能もスレッドを中断する際に使うことが可能です。2 つの方法、すなわち、クリーン

アップを使って、また、クリーンアップを使わずにプロセスとスレッドを終了させます。原則として、常に ExitProcess と ExitThread を使用し、システムがクリーンアップを実行できるようにします。他に全く方法がない場合にのみ、TerminateProcess と TerminateThread を使います。

複数のスレッドを使って作業する場合、プロセス内およびプロセス間で共有されたリソースに対するアクセスをコーディネートするため、スレッド同期を実行することが望ましいです。Windows Embedded CE は本目的、具体的にはクリティカル セクション、ミューテックスおよびセマフォに対しカーネル オブジェクトをいくつか提供します。クリティカル セクションは、単一のプロセス内のリソースに対するアクセスを保護します。ミューテックスは、複数のプロセスの間で共有されるリソースに対する排他的アクセスを相互にコーディネートします。セマフォは、プロセス内およびプロセス間のリソースに対する、複数のスレッドによる同時アクセスを実行します。イベントは他のスレッドに通知し、スレッドが安全なアトミック方法で変数を操作するためのインターロック機能を通知する際に使用します。開発フェーズの際にスレッド同期についてデッドロック等の問題に直面した場合は、イベントトラッキング システムの CeLog、および Remote Kernel Tracker を使い、対象デバイス上のスレッド インタラクションを分析してください。

試験に関するアドバイス

認定試験に合格するには、Windows Embedded CE 6.0 R2 に記載されたさまざまな同期オブジェクトの使用方法を理解する必要があります。

レッスン 4:　例外処理の実行

Windows Embedded CE を実行する対象デバイスには、システムおよびアプリケーション処理の一部としての例外が含まれます。課題は、適切な方法で例外に対応することです。例外処理により、安定した OS が可能になり、ユーザーは有意義な体験ができます。例えば、予期せぬ形でビデオ アプリケーションを終了させる代わりに、カメラに現在接続されていない場合は、ユーザーに対し、ユニバーサル シリアル バス （USB）カメラを接続するよう勧めた方が有益なことがおわかりでしょう。しかし、普遍的なソリューションとして例外処理を行ってはなりません。予期せぬアプリケーションの動作の結果、実行可能なファイル、DLL、メモリ構造およびデータに対し、悪意のあるコードによる不正行為が起こる可能性があります。この場合、故障したコンポーネントまたはアプリケーションを終了させることが、データやシステムを保護する最善の方策です。

このレッスンを終了すると、以下をマスターできます :

- 例外処理の理由を理解する
- 例外処理をキャッチし、スローする

レッスン時間 (推定): 30 分

例外処理の概要

例外処理はエラー条件の結果発生するイベントです。プロセッサ、OS、アプリケーションがカーネル モードやユーザー モードでの通常の制御フローの外で指示を実行している際に、こうした条件が発生する可能性があります。例外をキャッチし、例外処理を行うことで、アプリケーションの信頼性を高め、ユーザーに有意義な体験を提供することができます。しかし、厳密には、構造化された例外処理は Windows Embedded CE の不可欠な部分ですので、例外ハンドラを実行する必要はありません。

OS はあらゆる例外をキャッチし、それらをイベントの原因となったアプリケーション プロセスに転送します。プロセスがその例外イベントを処理しない場合、システムは例外をポストモーテム デバッガに転送し、その後、システムを故障したハードウェアまたはソフトウェアから守るための動作の中でプロセスを終了させます。Dr. Watson は Windows Embedded CE 用メモリ ダンプ ファイルを作成する共通のポストモーテム デバッガです。

例外処理とカーネル デバッグ

例外処理はまた、カーネル デバッグの基準でもあります。OS の設計でカーネル デバッグを有効にした場合、Platform Builder はランタイム イメージにカーネル デバッグ スタブ（KdStub）を含み、例外処理を上げてデバッガに入り込むコンポーネントを有効にします。そして、状況を分析し、コードを通し、処理を再開し、あるいは手動でアプリケーション プロセスを終了させることができます。しかし、対象デバイスとインタラクションするには、開発ワークステーションへの KITL 接続が必要です。KITL 接続が無ければ、デバッガは例外処理を無視し、アプリケーションが実行され続けますので、デバッガがアクティブでないかのように、OS が別の例外ハンドラを使うことができます。アプリケーションが例外処理を行わない場合、OS はカーネル デバッガに対し、ポストモーテム デバッグを実行する 2 回目のチャンスを与えます。この流れの中では、デバッグはしばしばジャストインタイム（JIT）デバッグと呼ばれます。そして、デバッガは例外処理を受け入れなければならず、KITL 接続がデバッグ出力に対し利用可能になるのを待ちます。Windows Embedded CE は、KITL 接続が確立され、対象デバイスのデバッグを開始するまで待機します。このシナリオではカーネル デバッガは例外処理を行うチャンスが 2 回ありますので、開発者ドキュメントはしばしば例外処理（初回）および例外処理（2 回目）という用語を使っていますが、実際には同じ例外イベントについて言及しています。デバッグおよびシステム テストに関しては、詳しくは第 5 章「システムのデバッグおよびテスト」を参照してください。

ハードウェアとソフトウェアの例外処理

Windows Embedded CE は全てのハードウェアとソフトウェアの例外処理に関し、同じ構造化例外処理（SEH）アプローチを使います。中央処理装置（CPU）は、ゼロでの除算または無効なメモリアドレスへのアクセスを試みることによるアクセス違反のような無効な命令シーケンスに対応して、ハードウェアの例外処理を上げることができます。一方、ドライバやシステム アプリケーション、ユーザー アプリケーションは、RaiseException 機能を使うことにより、OS の SEH メカニズムを呼び出すためのソフトウェア例外処理を上げることができます。例えば、要求されたデバイス（USB カメラ、データベース接続等）にアクセスできない場合、ユーザーが無効なコマンドライン パラメータを指定した場合、あるいは通常のコードパス外で特別な命令を実行するよう求められるといった理由がある場合、例外処理を上げることができます。RaiseException 機能呼び出しの際、いくつかのパラメータを指定し、例外処理について記載した情報を指定できます。そして、本仕様は例外ハンドラのフィルタ式で使用できます。

例外ハンドラ構文

Windows Embedded CE はフレームベースの構造化例外処理をサポートします。微妙なコード シーケンスを中括弧で囲み、__ try キーワードを使ってマークし、本コード実行中のいかなる例外処理も、__ except キーワードを使ってマークされるセクションで続く例外ハンドラを呼び出す必要があることを表示します。Microsoft Visual Studio に含まれる C/C++ コンパイラは、システムが例外処理の発生したポイントでコンピュータの状態を回復させ、スレッドの実行を続けることを可能にする、もしくは例外ハンドラの置かれたコールスタック フレームにおいてコントロールを例外ハンドラに移行し、スレッド例外処理を続けることを可能にする追加命令を使い、これらのキーワードをサポートし、コードブロックを編集します。

下記のコードは、構造化例外処理に対し、どのように __try キーワードと __except キーワードを使うかを示しています。

```
__try
{
    // Place guarded code here.
}
__except (filter-expression)
{
    // Place exception-handler code here.
}
```

__except キーワードは、簡易式またはフィルタ機能となるフィルタ式をサポートします。フィルタ式は下記の値のうちの一つを評価できます。

- EXCEPTION_CONTINUE_EXECUTION　システムは例外処理が発生したポイントで例外が解決され、スレッドの実行が続いていると仮定します。フィルタ機能は、典型的には例外処理後この値を戻し、通常通り処理を続けます。
- EXCEPTION_CONTINUE_SEARCH　システムは適切な例外ハンドラを検索し続けます。
- EXCEPTION_EXECUTE_HANDLER　システムスレッドは例外ポイントよりもむしろ例外ハンドラから順番に実行し続けます。

ノート　例外処理サポート
例外処理は C 言語の延長ですが、ネイティブには C++ でサポートされています。

終了ハンドラ構文

Windows Embedded CE は終了処理をサポートします。Microsoft の C および C++ 言語への延長として、制御フローが保護されたコードブロックを変えない場合でも、システムが常に特定のコードブロックを実行することを保証します。このコードセクションは終了ハンドラと呼ばれ、例外その他のエラーが保護されたコードで発生してもクリーンアップ タスクを実行する際に使用します。例えば、もう必要でないスレッドハンドラを閉じる際に終了ハンドラを使用できます。

次のコードは、構造化された例外処理に対し、どのように __try および __finally キーワードを使うかを示しています。

```
__try
{
    // Place guarded code here.
}
__ finally
{
    // Place termination code here.
}
```

終了処理は保護されたセクション内で __leave キーワードをサポートします。このキーワードは、保護されたセクションの現在の位置でスレッド実行を終了させ、呼び出し履歴をアンワインドせずに、終了ハンドラにおける最初のステートメントでのスレッド実行を再開します。

ノート　__try、__except および __finally ブロックの使用

単一の __try ブロックは例外ハンドラと終了ハンドラの両方を持つことはできません。最終的に __except と __finally の両方を使う必要がある場合、外側の try-except ステートメントと try-finally ステートメントを使用してください。

動的なメモリ割り当て

動的なメモリ割り当ては、システム上の合計関連メモリページ数を最小限にするための構造化例外処理に依存する割り当て技法です。これは大容量のメモリ割り当てを行わなければならない場合にとりわけ有用です。全体の割り当てを事前にコミットさせた場合、システムからコミット可能なページが無くなり、その結果、仮想メモリ割り当てエラーとなる場合があります。

動的なメモリ割り当て技法は次の通りです。

1. ベース アドレス NULL を使って VirtualAlloc を呼び出し、メモリ ブロックを確保します。システムはページをコミットさせずにこのメモリ ブロックを確保します。
2. メモリ ページへのアクセスを試みます。これによって、非コミット ページからの読み出しや同ページへの書き込みができなくなりますので、例外が上げられます。この違法な操作の結果、ページ フォールトの例外が上げられます。図 3ミ5 および 3ミ6 は、PageFault.exe というアプリケーションにおいて処理されないページ フォールトの結果を示しています。
3. フィルタ機能に基づき例外ハンドラを実行します。確保した領域からのフィルタ機能にページをコミットさせます。正常に実行された場合、EXCEPTION_CONTINUE_EXECUTION を戻し、例外が発生したポイントで __try ブロックにおいてスレッドを実行し続けます。ページ割り当てが失敗した場合、EXCEPTION_EXECUTE_HANDLER を戻し、__except ブロックに例外ハンドラを呼び出し、確保し、コミットしたページの領域全体をリリースします。

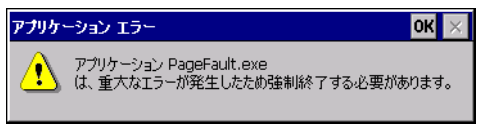


図 3ミ5　処理されていないページ フォールトの例外。ユーザーの視点から

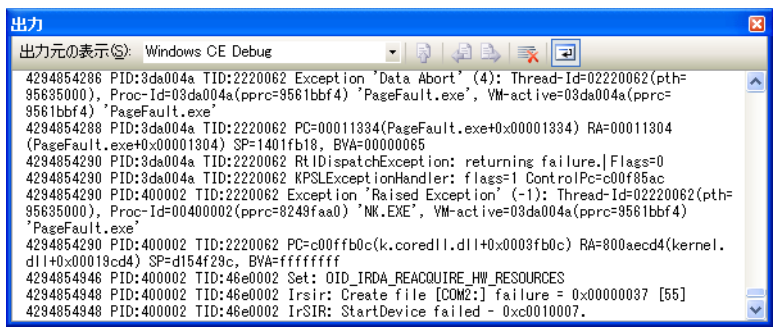


図 3ミ6　Visual Studio 2005 における KITL に関する未処理ページ フォールト例外のデバッグ出力

次のコード スニペットはページ フォールト例外処理に基づく動的メモリ割り当て技法について示しています。

```
#define PAGESTOTAL 42    // Max. number of pages

LPTSTR lpPage;           // Page to commit
DWORD dwPageSize;        // Page size, in bytes

INT ExceptionFilter(DWORD dwCode)
{
    LPVOID lpvPage;

    if (EXCEPTION_ACCESS_VIOLATION != dwCode)
    {
        // This is an unexpected exception!
        // Do not return EXCEPTION_EXECUTE_HANDLER
        // to handle this in the application process.
        // Instead, let the operating system handle it.
        return EXCEPTION_CONTINUE_SEARCH;
    }

    // Allocate page for read/write access.
    lpvPage = VirtualAlloc((LPVOID) lpPage,
                           dwPageSize, MEM_COMMIT,
                           PAGE_READWRITE);
    if (NULL == lpvPage)
    {
        // Continue thread execution
        // in __except block.
        return EXCEPTION_EXECUTE_HANDLER;
    }

    // Set lpPage to the next page.
    lpPage = (LPTSTR) ((PCHAR) lpPage + dwPageSize);

    // Continue thread execution in __try block.
    return EXCEPTION_CONTINUE_EXECUTION;
}

VOID DynamicVirtualAlloc()
{
    LPVOID lpvMem;
    LPTSTR lpPtr;
    DWORD i;
    BOOL bRet;

    // Get page size on computer.
    SYSTEM_INFO sSysInfo;
    GetSystemInfo(&sSysInfo);
    dwPageSize = sSysInfo.dwPageSize;

    // Reserve memory pages without committing.
    lpvMem = VirtualAlloc(NULL, PAGESTOTAL*dwPageSize,
                          MEM_RESERVE, PAGE_NOACCESS);
```

```
    lpPtr = lpPage = (LPTSTR) lpvMem;

    // Use structured exception handling when accessing the pages.
    for (i=0; i < PAGESTOTAL*dwPageSize; i++)
    {
        __try
        {   // Write to memory.
            lpPtr[i] = 'x';
        }
        __except (ExceptionFilter(GetExceptionCode()))
        {   // Filter function unsuccessful. Abort mission.
            ExitProcess( GetLastError() );
        }
    }

    // Release the memory.
    bRet = VirtualFree(lpvMem, 0, MEM_RELEASE);
}
```

レッスン概要

Windows Embedded CE はネイティブに例外処理と終了処理をサポートします。プロセッサ、OS およびアプリケーションでの例外は不適切な命令シーケンス、利用不可能なメモリ アドレスへのアクセスの試み、アクセス不可能なデバイス リソース、無効なパラメータ、あるいはその他、動的メモリ割り当て等の特別な処理を必要とする操作に対して上がります。try ミ except ステートメントを使い、通常の制御フロー外のエラー条件に対応でき、また、try ミ finally ステートメントを使い、制御フローが保護された __try コード ブロックを変えない場合でもコードを実行できます。

例外処理はフィルタ式とフィルタ機能をサポートします。これらにより、上げられたイベントにどう対応するか、コントロールできるようになります。悪意のあるコードの結果、予期せぬアプリケーションの動作が起こる可能性がありますので、全ての例外を捕捉することはおすすめできません。アプリケーションの動作を信頼できるものにするため、直接対応する必要のある例外のみを処理してください。OS はいかなる未処理の例外もポストモーテム デバッガに転送し、メモリダンプを作成し、アプリケーションを終了することができます。

試験に関するアドバイス

認定試験に合格するには、Windows Embedded CE 6.0 R2 に記載された例外処理および終了処理方法を理解する必要があります。

レッスン 5:　電源管理の実行

Windows Embedded CE デバイスにとって、電源管理は不可欠です。電力消費を抑えることで、バッテリの寿命を延ばし、ユーザーに対しては長期にわたる有意義な体験を保証します。これはポータブル デバイスに関する電源管理の究極的な目標です。固定デバイスも電源管理の恩恵を受けます。装置のサイズにかかわらず、非アクティブ後デバイスを低電力状態にする場合、運用コストや熱放散、機械的磨耗や破損を減らすことができます。そしてもちろん、効果的な電源管理機能を実行することで、私たちの環境への負荷を減らす一助になります。

このレッスンを終了すると、以下をマスターできます :

- 対象デバイスに対する電源管理ができる
- アプリケーションにおける電源管理機能を実行する

レッスン時間 (推定): 40 分

Power Manager 概要

Windows Embedded CE では、Power Manager (PM.dll) は、電源管理機能を実行する Device Manager (Device.exe) と統合するカーネル コンポーネントです。本質的に、Power Manager はカーネル、OEM アダプテーション層（OAL)、周辺デバイスおよびアプリケーション用ドライバとの間のメディエイタとして機能します。カーネルと OAL をドライバやアプリケーションから分けることで、ドライバやアプリケーションはその電源状態をシステム状態とは別に管理できます。ドライバやアプリケーションは Power Manager とインターフェイスし、電源イベントに関する通知を受け取り、電源管理機能を実行することができます。Power Manager はイベントやタイマに対応してシステムの電源状態を設定し、ドライバの電源状態を制御し、バッテリ電源状態が危機的な場合等、電源状態を変えるよう求められる OAL イベントに対応する能力を持ちます。

Power Manager のコンポーネントおよびアーキテクチャ

Power Manager はそのタスクに従い、通知インターフェイス、アプリケーション インターフェイス、そしてデバイス インターフェイスを表示します。通知インターフェイスにより、システム状態またはデバイス電源状態が変わる等、電源管理イベントに関する情報をアプリケーションが受領できるようになります。一方、デバイス インターフェイスは、デバイス ドライバの電源レベルを制御するメカニズムを提供します。Power Manager は、システムの電源状態とは別にデバイスの電源状態を設定できます。アプリケーションと同様、デバイス

マネージャはその電源要求を Power Manager へ戻すためのドライバ インターフェイスを使用できます。重要なのは、図 3ミ7 に示されているように、Power Manager と Power Manager API が Windows Embedded CE 上で電源管理を集約化することです。

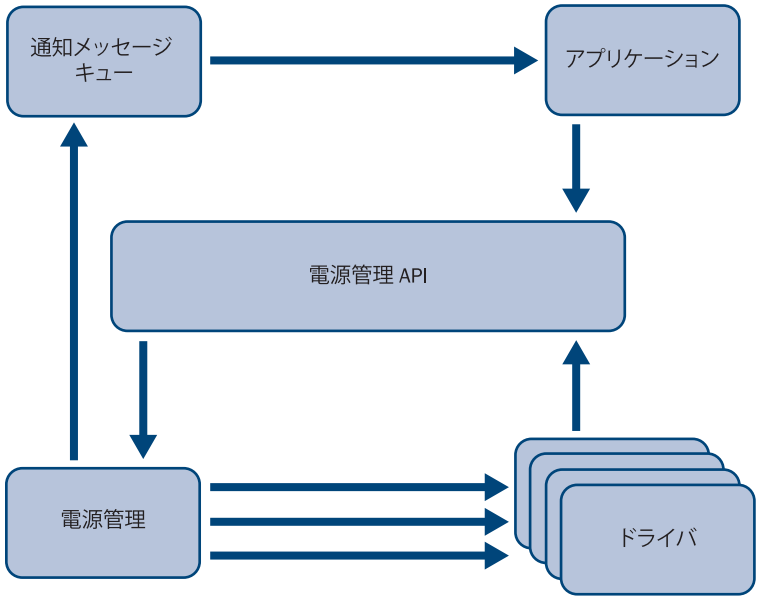


図 3ミ7　Power Manager と Power Management の インタラクション

Power Manager ソースコード

Windows Embedded CE には、開発コンピュータ上の %_WINCEROOT%\Public\Common\Oak\Drivers\Pm フォルダにある Power Manager 用ソースコードが付いています。このコードをカスタマイズすることで、対象デバイス上にカスタマイズされた電源処理メカニズムを提供できます。例えば、OEM メーカーは、PowerOffSystem 機能を呼び出す前に特別なコンポーネントを閉じる追加ロジックを実行できます。標準 Windows Embedded CE コンポーネントを複製し、カスタマイズする技法については、第 1 章「OS 設計のカスタマイズ」を参照してください。

ドライバ電源状態

アプリケーションやデバイス マネージャは周辺デバイスの電源状態を制御する際、DevicePowerNotify 機能を使用できます。例えば、DevicePowerNotify を呼び出し、BLK1 等、バックライト ドライバの電源レベルを変更したい旨を Power Manager に通知することができます。Power Manager は、ハードウェア デバイ

ス能力に従い、次の 5 つの異なる電源レベルで必要な電源状態を指定するよう、要求します。

- D0　Full On; デバイスは正常に機能
- D1　Low On; デバイスは機能しているが、処理能力が低い
- D2　Standby; デバイスは一部電源が入っており、要求があればスリープ解除
- D3　Sleep; デバイスは一部電源が入っている。この状態ではデバイスにはまだ電源が入っており、CPU のスリープを解除（デバイスからのスリープ解除）する割り込みを実行できる
- D4　Off; デバイスには電源が入っていない。この状態ではデバイスは電力をほとんど消費しない

ノート　CE デバイス電源状態レベル

デバイスの電源状態（D0 から D4）は、OEM メーカーがそのプラットフォームで電源管理機能を実行できるようにするガイドラインです。Power Manager はいかなる状態でもデバイスの電力消費や応答、あるいは能力に対し制限を課すことはありません。原則として、大きな番号の状態は小さな番号の状態よりも電力消費が少なく、また、電源状態 D0 と D1 はユーザーの視点からは稼動中と見られるデバイスに関するものだと考えられます。粒度の低い物理デバイスの電源レベルを管理するデバイスドライバは、原電状態のサブセットを実行できます。D0 は唯一求められる電源状態です。

システム電源状態

Power Manager は、アプリケーションおよびデバイス ドライバのリクエストに対応してデバイス ドライバに電源状態変更通知を送ることに加え、ハードウェア関連のイベントやソフトウェアのリクエストに対応してシステム全体の電源状態を移行させることもできます。ハードウェアのイベントにより、Power Manager は低い、危機的なバッテリ レベルや、バッテリ電源から AC 電源への移行に対応できます。ソフトウェアのリクエストにより、アプリケーションは Power Manager の SetSystemPowerState 機能を呼び出す際、システムの電源状態を変更するよう、リクエストできます。

デフォルト Power Manager を実行することにより、次の 4 つの電源状態をサポートします。

- On　システムは正常に機能し、通常の電力で実行
- UserIdle　ユーザーはデバイスを受動的に使用。構成可能な時間については、ユーザーのインプット無し

- SystemIdle　ユーザーはデバイスを使用していない。構成可能な時間については、システムアクティビティ無し
- Suspend　デバイスの電源は切れているが、デバイスによるスリープ解除をサポート

システムの電源状態は対象デバイスの要求および能力に依存していることを留意する必要があります。OEM メーカーは InCradle や OutOfCradle 等、自社の、あるいは追加のシステム電源状態を定義できます。Windows Embedded CE は、定義できるシステム電源状態の数に制限は設けませんが、本レッスンで既述の通り、システム電源状態は全てデバイス電源状態の一つに変換されます。

図 3ミ8 はデフォルトのシステム電源状態とデバイス電源状態との関係を示しています。

図 3ミ8　デフォルトのシステム電源状態と、関連するデバイス電源状態

アクティビティ タイマ

システム状態の移行はアクティビティ タイマと、対応するイベントに基づいて行われます。ユーザーがデバイスを使用しない場合、タイマは時間切れとなり、非アクティブのイベントを上げ、それに対応し、Power Manager に対してはシステムを電源状態 Suspend に移行させます。ユーザーが入力を行ってシステムを戻し、システムとインタラクションする場合、アクティビティ イベントにより Power Manager がシステムを電源状態 On に戻します。しかし、この簡素化モデルは、携帯情報端末（PDA）上のビデオクリップを閲覧するユーザーのように、入力を行わないユーザーのアクティビティの時間が長引くことを考慮していません。また、表示パネルの場合と同様、ユーザーが直接入力を行う方法の無い対象デバイスも考慮していません。こうしたシナリオをサポートするため、Power Manager をデフォルトで実行することにより、図 3ミ9 に示されているように、ユーザー アクティビティとシステム アクティビティが区別され、それに従ってシステム電源状態が移行されます。

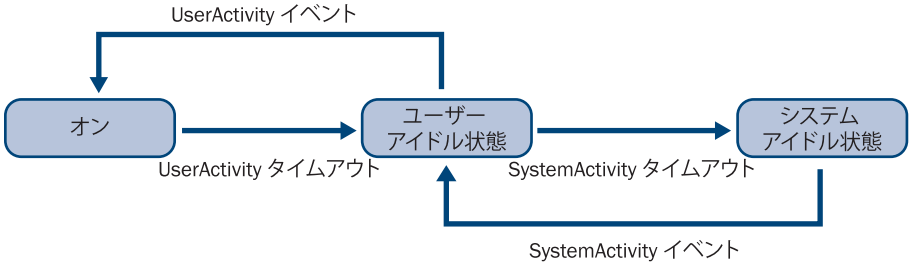


図 3ミ9　アクティビティ タイマ、イベント、およびシステム電源状態移行

システム アクティビティおよびユーザー アクティビティのタイムアウトを構成するには、Power Control Panel アプレットを使用します。また、レジストリを直接編集することによって、追加のタイマを実行し、そのタイムアウトを設定することも可能です。Windows Embedded CE は作成できるタイマの数を制限しません。起動時に Power Manager はレジストリ キーを読み、アクティビティ タイマを列挙し、関連イベントを作成します。表 3ミ16 は SystemActivity タイマに対するレジストリ設定について記載しています。OEM メーカーは同様のレジストリ キーを追加し、追加タイマに対し、これらの値を構成できます。

表 3ミ16　アクティビティ タイマに対するレジストリ設定

場所	**HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Power\ActivityTimers\SystemActivity**	
入力	Timeout	WakeSources
タイプ	REG_DWORD	REG_MULTI_SZ
値	A （10 分間）	0x20
説明	Timeout レジストリ入力はタイマのしきい値を分単位で定義	WakeSources レジストリ入力はオプションで、使用可能なスリープ状態の解除元に対する識別子のリストを定義。デバイスからのスリープ状態の間、Power Manager は IOCTL_HAL_GET_WAKE_SOURCE 入出力制御（IOCTL）コードを使ってスリープ状態の解除元を決定し、関連アクティビティ タイマをアクティブに設定

ノート　アクティビティ タイマ

アクティビティ タイマの定義により、Power Manager はタイマを再設定し、アクティビティ ステータスを入手する、名称の付いたイベントを構成します。詳細は、Windows Embedded CE 6.0 Documentation の「アクティビティ タイマ」を参照してください。次の Microsoft MSDN サイトでご覧になれます。http://msdn2.microsoft.com/en-us/library/aa923909.aspx.

電源管理 API

本レッスンで既述のように、Power Manager は電源管理についてアプリケーションやドライブを有効にする 3 つのインターフェイス、すなわち通知インターフェイス、ドライバ インターフェイス、およびアプリケーション インターフェイスを表示します。

通知インターフェイス

表 3ミ17 に記載されているように、通知インターフェイスは、アプリケーションがメッセージ キューを通して電源通知に関し登録および再登録する際に使える 2 つの機能を提供します。電源通知はマルチキャスト メッセージであり、それはすなわち Power Manager が通知メッセージを登録されたプロセスにのみ送るということを留意する必要があります。このように、電源管理が有効なアプリケーションは Windows Embedded CE 上で、Power Management API を実行しないアプリケーションとシームレスに共存できます。

表 3ミ17　電源管理通知インターフェイス

機能	説明
RequestPowerNotifications	Power Manager を使ってアプリケーションプロセスを登録し、電源通知を受け取る。その後、Power Manager は次の通知メッセージを送る。 ■ PBT_RESUME　システムを Suspend（中拍）溝態から再開 ■ PBT_POWERSTATUSCHANGE　AC　電源とバッテリ電源との間でシステムを移行 ■ PBT_TRANSITION　システムが新しい電源溝態に噂わる ■ PBT_POWERINFOCHANGE　バッテリのステータスが噂わる。本メッセージはバッテリドライバが蚣み誂まれる場合にのみ有傻
StopPowerNotifications	電源通知を受け取らないよう、アプリケーションプロセスを登録解除

次のサンプルコードは電源通知の使い方を表しています。

```
// Size of a POWER_BROADCAST message.
DWORD cbPowerMsgSize =
    sizeof POWER_BROADCAST + (MAX_PATH * sizeof TCHAR);

// Initialize a MSGQUEUEOPTIONS structure.
MSGQUEUEOPTIONS mqo;
mqo.dwSize = sizeof(MSGQUEUEOPTIONS);
mqo.dwFlags = MSGQUEUE_NOPRECOMMIT;
mqo.dwMaxMessages = 4;
mqo.cbMaxMessage = cbPowerMsgSize;
mqo.bReadAccess = TRUE;

//Create a message queue to receive power notifications.
HANDLE hPowerMsgQ = CreateMsgQueue(NULL, &mqo);
if (NULL == hPowerMsgQ)
{
    RETAILMSG(1, (L"CreateMsgQueue failed: %x\n", GetLastError()));
    return ERROR;
}

// Request power notifications.
HANDLE hPowerNotifications = RequestPowerNotifications(hPowerMsgQ,
                                        PBT_TRANSITION |
```

```
                                        PBT_RESUME |
                                        PBT_POWERINFOCHANGE);

    // Wait for a power notification or for the app to exit.
    while(WaitForSingleObject(hPowerMsgQ, FALSE, INFINITE) == WAIT_OBJECT_0)
    {
        DWORD cbRead;
        DWORD dwFlags;
        POWER_BROADCAST *ppb = (POWER_BROADCAST*) new BYTE[cbPowerMsgSize];

        // Loop through in case there is more than 1 msg.
        while(ReadMsgQueue(hPowerMsgQ, ppb, cbPowerMsgSize, &cbRead,
                           0, &dwFlags))
        {
          \\ Perform action according to the message type.
        }
    }
```

デバイス ドライバ インターフェイス

Power Manager と統合するため、デバイス ドライバは I/O コントロール (IOCTL) をサポートする必要があります。Power Manager はこれらを使い、図 3ミ10 に示されているように、デバイスの電源状態を設定・変更するとともに、デバイス固有の電源能力のクエリを行います。Power Manager IOCTL に基づき、デバイス ドライバはハードウェア デバイスを対応する電源構成に入れ込む必要があります。

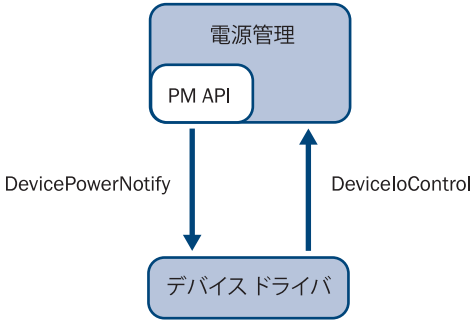


図 3ミ10　Power Manager およびデバイスドライバインタラクション

Power Manager は次の IOCTL を使い、デバイス ドライバと対話します。

- IOCTL_POWER_CAPABILITIES　Power Manager はデバイス ドライバの電源管理能力をチェックします。戻された情報はハードウェアと、ハードウェア デバイスを管理するドライバの能力を反映する必要があります。ドライバはサポートされた Dx 状態のみ戻さなければなりません。

- IOCTL_POWER_SET　Power Manager はドライバに対し、指定された Dx 状態へスイッチさせます。ドライバは電源の切り替えを行わなければなりません。
- IOCTL_POWER_QUERY　Power Manger はドライバがデバイスの状態を変えることができるかどうかチェックします。
- IOCTL_POWER_GET　Power Manger はデバイスの現在の電源状態を確定するよう求めます。
- IOCTL_REGISTER_POWER_RELATIONSHIP　Power Manger は親ドライバに対し、制御している全ての子デバイスを登録するよう、通知します。Power Manger は、POWER_CAPABILITIES 構造の Flags メンバーにおいて POWER_CAP_PARENT フラグを含むデバイスにのみ、この IOCTL を送ります。

ノート　内部電源状態切り替え

電源管理を信頼できるものにするため、デバイス ドライバは Power Manager の関与無しにその内部電源状態を変えてはなりません。ドライバが電源状態の切り替えを必要とする場合、ドライバは DevicePowerNotify 機能を使い、電源状態を変えるようリクエストします。そして、Power Manager がドライバへ電源状態変更リクエストを送り戻す際、ドライバはその内部電源状態を変えることができます。

アプリケーション インターフェイス

アプリケーション インターフェイスは Power Manager を通じ、システムおよび個々のデバイスの電源状態を管理する際にアプリケーションが使用できる機能を提供します。表 3ミ18 はこうした電源管理機能の概要を示しています。

表 3ミ18　アプリケーション インターフェイス

機能	説明
GetSystemPowerState	現在のシステム電源状態を取得
SetSystemPowerState	電源状態の変更をリクエスト。Suspend モードに切り替える場合、Suspend はシステムに対し透明なので、機能は再開後戻ることになる。再開後、通知メッセージを分析し、システムが Suspend から再開したことを確認できる
SetPowerRequirement	デバイスに対し最小限の電源状態をリクエストする

表 3ミ18　アプリケーション インターフェイス

機能	説明
ReleasePowerRequirement	SetPowerRequirement 機能を使ってあらかじめ設定した電源要求をリリースし、元のデバイス電源状態を回復
GetDevicePower	指定したデバイスの現在の電源状況を取得
SetDevicePower	デバイスに対し電源状態を変えるようリクエスト

電源状態構成

図 3ミ8 に示したように、Power Manager はシステム電源状態をデバイス電源状態に関連付け、システムおよびデバイスを同期し続けます。特に断りの無い限り、Power Manager は次のシステム状態対デバイス状態のマッピング、On = D0、UserIdle = D1、SystemIdle = D2 および Suspend = D3 をデフォルトで適用します。個々のデバイスおよびデバイスクラスに対する関連付けは明示的なレジストリ設定により優先できます。

個々のデバイスに対する電源状態構成の優先

Power Manager をデフォルトで実装することにより、レジストリにおけるシステム状態対デバイス状態のマッピングを HKEY_LOCAL_MACHINE\System\CurrentControlSet\State キーのもとで管理します。それぞれのシステム電源状態は別々のサブキーに対応し、OEM 固有の電源状態に対しては追加のサブキーを作成できます。

表 3ミ19 はシステム電源状態が On の場合のサンプル構成を表しています。この構成によって、Power Manager はバックライト ドライバ BLK1: を除く全てのデバイスを D0 デバイス電源状態に切り替えます。バックライト ドライバ BLK1 は D2 状態にのみ進むことができます。

表 3ミ19　システム電源状態 On に対するデフォルトおよびドライバ固有の電源状態定義

場所	HKEY_LOCAL_MACHINE\System\CurrentControlSet\State\On		
入力	Flags	Default	BKL1:
タイプ	REG_DWORD	REG_DWORD	REG_DWORD
値	0x00010000 (POWER_STATE_ON)	0 (D0)	2 (D2)

表 3ミ19　システム電源状態 On に対するデフォルトおよびドライバ固有の電源状態定義

場所	HKEY_LOCAL_MACHINE\System\CurrentControlSet\State\On		
説明	本レジストリキーに関連したシステム電源状態を特定。可能なフラグのリストについては Public\Common\Sdk\Inc フォルダの Pm.h ヘッダー ファイルを参照	システム電源状態が On の場合、ドライバに対する電源状態をデフォルトで D0 状態に設定	システム電源状態が On の場合、バックライトドライバ BLK1: を D2 状態に設定

デバイス クラスに対する電源状態構成の優先

複数のシステム電源状態に対しデバイス電源状態をそれぞれ定義することは退屈な作業となりがちです。Power Manager は、値 IClass に基づきデバイス クラスをサポートすることによって、電源管理ルールの定義に使用できる構成を容易にします。HKEY_LOCAL_MACHINE \System\CurrentControlSet\Control \Power\Interfaces レジストリ キーには次の 3 つのデフォルト クラス定義があります。

- {A3292B7-920C-486b-B0E6-92A702A99B35}　汎用の電源管理によるデバイス
- {8DD679CE-8AB4-43c8-A14A-EA4963FAA715}　電源管理によるブロック デバイス
- {98C5250D-C29A-4985-AE5F-AFE5367E5006}　電源管理による Network Driver Interface Specification (NDIS) ミニポート ドライバ

表 3ミ20 は NDIS デバイス クラスに対するサンプル構成を表しており、NDIS ドライバの進めるのが最大でも D4 状態であることを示しています。

表 3ミ20　NDIS デバイス クラスに対するサンプル電源状態定義

場所	HKEY_LOCAL_MACHINE\System\ CurrentControlSet \Control \Power\State\On\{98C5250D-C29A-4985-AE5F-AFE5367E5006}
入力	Default
タイプ	REG_DWORD
値	4 (D4)

表 3ミ20　NDIS デバイス クラスに対するサンプル電源状態定義

場所	HKEY_LOCAL_MACHINE\System\ CurrentControlSet \Control \Power\State\On\{98C5250D-C29A-4985-AE5F-AFE5367E5006}
説明	システム電源状態が On の場合、NDIS ドライバに対するデバイス電源状態を D4 に設定

プロセッサのアイドル状態

電源管理によるアプリケーションとデバイス ドライバに加え、カーネルも電源管理に寄与します。スレッドを実行しようとしない場合、カーネルは OAL の一部である OEMIdle 機能を呼び出します。このアクションは、現在のコンテキストを保存し、メモリを更新状態に置き、クロックを停止させることを含むアイドル状態にプロセッサを切り替えます。プロセッサのアイドル状態は、アイドル状態から即座に戻る能力を保持すると同時に、電力消費を最小限に抑えます。

OEMIdle 機能が Power Manager を含まないことを留意する必要があります。カーネルは OEMIdle 機能を直接呼び出し、そして、ハードウェアをアイドルまたは休止状態に切り替えるのは OAL 次第です。カーネルは DWORD 値（dwReschedTime）を OEMIdle に渡し、最大アイドル時間を表示します。この時間が過ぎたとき、あるいはハードウェア タイマがサポートする最大遅延時間に達したとき、プロセッサは非アイドル モードに切り替えを戻し、前の状態を復元し、スケジューラが呼び出されます。実行しようとするスレッドがまだ無い場合、カーネルはすぐに OEMIdle を再度呼び出します。キーボードまたはスタイラスによるユーザー入力への対応と同様、ドライバ イベントはいつでも発生し、システム タイマ作動前にシステムにアイドリングを停止させる可能性があります。

スケジューラはデフォルト設定では 1 ミリ秒の間隔で静的タイマおよびシステム ティックに基づいて作動します。しかし、ダイナミック タイマを使い、スケジューラ テーブルの内容を使うことで特定される次のタイムアウトにシステム タイマを設定することにより電力消費を最適化できます。その後、プロセッサはティックごとにアイドル モードから切り替え戻すことはしません。代わりに、dwReschedTime の定義したタイムアウトが期限切れとなるか、割り込みが発生した後、プロセッサは非アイドル モードにのみ切り替えます。

レッスン概要

Windows Embedded CE 6.0 R2 は、Power Manager をデフォルトで実行する場合、システムおよびそのデバイスの電源状態を管理する際に使用できる電源管理 API を提供します。また、OEMIdle 機能も提供しますが、これは OEM メーカーに対し、特定の時間、システムを低電力アイドル状態に置く機会を与えるためスケジューリングされたスレッドをシステムが持たない場合に実行するものです。

Power Manager は通知インターフェイス、アプリケーション インターフェイスおよびデバイス インターフェイスを表示するカーネル コンポーネントです。一方ではカーネルと OAL との間のメディエイタとして、そしてもう一方ではデバイス ドライバとアプリケーションとの間のメディエイタとして機能します。アプリケーションとデバイス ドライバは DevicePowerNotify 機能を使い、5 つの異なる電源レベルで周辺デバイスの電源状態を制御します。デバイス電源状態は、デフォルトとカスタマイズのシステム電源状態と関連し、システムとデバイスを同期させ続けることもできます。アクティビティの回数と対応するイベントに基づき、Power Manger は自動的にシステム状態を移行させることができます。デフォルトのシステム電源状態は、On、UserIdle、SystemIdle および Suspend の 4 種類あります。次のシステム状態対デバイス状態のマッピングに対するカスタマイズは、個々のデバイスおよびデバイスクラスのレジストリ設定の中で行います。

Power Manager に加え、カーネルは OEMIdle 機能によって電源管理をサポートします。プロセッサをアイドル状態に切り替えることで、アイドル状態からすぐに戻る能力を保持する一方、電力消費を可能な限り抑えます。プロセッサは定期的に、または割り込みが発生した場合、ユーザー入力に対応する場合やデバイスがデータ転送用メモリへのアクセスをリクエストする場合と同様、非アイドル状態に戻ります。

Power Manager や OEMIdle を使って電源管理を適切に行えば、デバイスの電力消費を大幅に減らすことができ、それによってバッテリの寿命を延ばし、運用コストを削減し、デバイスの寿命を延ばすことができます。

演習 3：キオスクモード、スレッド、電源管理

本演習では、キオスクアプリケーションを開発し、標準のシェルの代わりに本アプリケーションを実行する対象デバイスを構成します。それから、本アプリケーションを延長し、Remote Kernel Tracker ツールを使い、アプリケーションプロセスで並行して複数のスレッドを実行し、スレッド実行を分析します。その後、本アプリケーションを電源管理に使うことができます。

ノート　詳細かつ段階的な指示書

本演習で示された手順をしっかりマスターできるよう、本書の付属資料の「Detailed Step-by-Step Instructions for Lab 3（演習 3 に関する詳細かつ段階的な指示書）」を参照してください。

x スレッドの作成

1. New Project Wizard を使用し、HelloWorld という新しい WCE Console Application を作成します。その際、Typical Hello_World Application オプションを使ってください。
2. _tmain 機能の前に、ThreadProc というスレッド機能を実行します。

```
DWORD WINAPI ThreadProc( LPVOID lpParameter)
{
    RETAILMSG(1,(TEXT("Thread started")));

    // Suspend Thread execution for 3 seconds
    Sleep(3000);

    RETAILMSG(1,(TEXT("Thread Ended")));

    // Return code of the thread 0,
    // usually used to indicate no errors.
    return 0;
}
```

3. CreateThread 機能を使い、スレッドを開始します。

```
HANDLE hThread = CreateThread( NULL, 0, ThreadProc, NULL, 0, NULL);
```

4. CreateThread の戻り値をチェックし、スレッドが正常に作成されたことを確認します。
5. スレッドがスレッド機能の終端に達するまで待機し、終了します。

```
WaitForSingleObject(hThread, INFINITE);
```

6. ランタイム イメージを作成し、対象デバイスにダウンロードします。

7. Remote Kernel Tracker を開始し、システムでどのようにスレッドが管理されているか分析します。
8. 図　3 ミ 11　に示されたように、HelloWorld　アプリケーションを開始し、Remote Kernel Tracker 画面に示されたスレッド実行方法に従って操作します。

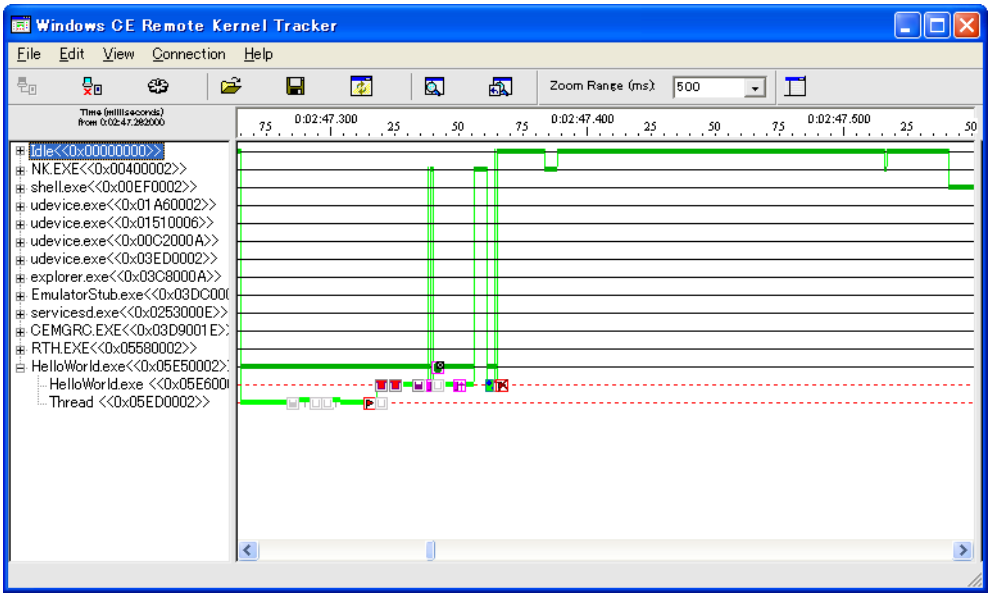


図 3 ミ 11　Remote Kernel Tracker ツールでスレッド実行をトラッキング

x 電源管理通知メッセージを有効にする

1. Visual Studio で HelloWorld アプリケーションを引き続き使用します。
2. サブプロジェクト レジストリ設定に行き、AC 電源モード（ACUserIdle）における UserIdle タイムアウトに対するレジストリ入力を 5 秒間に設定することで、より頻繁に電源管理通知を行います。

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\Timeouts]
    "ACUserIdle"=dword:5                ; in seconds
```

3. ThreadProc 機能でメッセージキュー オブジェクトを作成します。

```
// Size of a POWER_BROADCAST message.
DWORD cbPowerMsgSize =
    sizeof POWER_BROADCAST + (MAX_PATH * sizeof TCHAR);

// Initialize our MSGQUEUEOPTIONS structure.
MSGQUEUEOPTIONS mqo;
```

```
mqo.dwSize = sizeof(MSGQUEUEOPTIONS);
mqo.dwFlags = MSGQUEUE_NOPRECOMMIT;
mqo.dwMaxMessages = 4;
mqo.cbMaxMessage = cbPowerMsgSize;
mqo.bReadAccess = TRUE;

//Create a message queue to receive power notifications.
HANDLE hPowerMsgQ = CreateMsgQueue(NULL, &mqo);
if (NULL == hPowerMsgQ)
{
    RETAILMSG(1, (L"CreateMsgQueue failed: %x\n", GetLastError()));
    return -1;
}
```

4. Power Manager からの通知受領をリクエストし、受領された通知をチェックします。

```
// Request power notifications
HANDLE hPowerNotifications = RequestPowerNotifications(hPowerMsgQ,
                                        PBT_TRANSITION |
                                        PBT_RESUME |
                                        PBT_POWERINFOCHANGE);
DWORD dwCounter = 20;

// Wait for a power notification or for the app to exit
while(dwCounter-- &&
      WaitForSinglObject(hPowerMsgQ, INFINITE) == WAIT_OBJECT_0)
{
    DWORD cbRead;
    DWORD dwFlags;
    POWER_BROADCAST *ppb =
      (POWER_BROADCAST*) new BYTE[cbPowerMsgSize];

    // loop through in case there is more than 1 msg.
    while(ReadMsgQueue(hPowerMsgQ, ppb, cbPowerMsgSize,
                       &cbRead, 0, &dwFlags))
    {
        switch(ppb->Message)
        {
            case PBT_TRANSITION:
            {
                RETAILMSG(1,(L"Notification: PBT_TRANSITION\n"));
                if(ppb->Length)
                {
                    RETAILMSG(1,(L"SystemPowerState: %s\n",
                                 ppb->SystemPowerState));
                }
                break;
            }
            case PBT_RESUME:
            {
                RETAILMSG(1,(L"Notification: PBT_RESUME\n"));
```

```
                    break;
                }
                case PBT_POWERINFOCHANGE:
                {
                    RETAILMSG(1,(L"Notification: PBT_POWERINFOCHANGE\n"));
                    break;
                }
                default:
                    break;
            }
        }
        delete[] ppb;
    }
```

5. アプリケーションをビルドし、ランタイム イメージをリビルドします。
6. ランタイム イメージを開始します。
7. マウスのカーソルを移動させてユーザー アクティビティを作成します。非アクティブの状態が 5 秒間続いた後、図 3ミ12 に示されたように、Power Manager はアプリケーションに通知する必要があります。

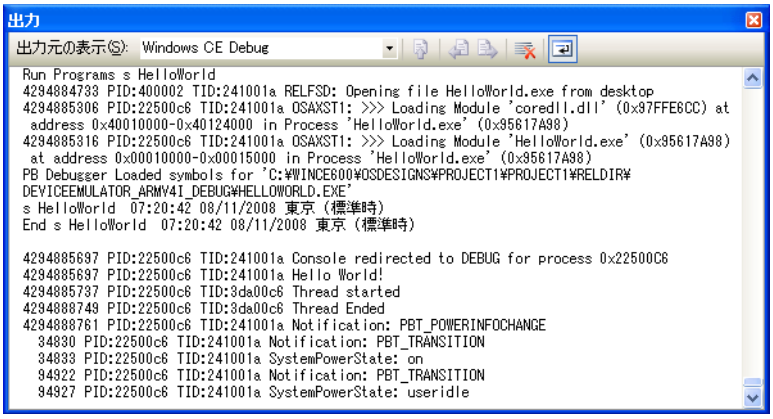


図 3ミ12　受領された電源管理通知

x キオスク モードを有効にする

1. Subproject Wizard を使って Subproject_Shell という WCE アプリケーションを作成します。その際、Typical Hello_World Application オプションを使用します。
2. 最初の LoadString ラインの前に SignalStarted 命令を追加します。

```
// Initialization complete,
// call SignalStarted...
SignalStarted(_wtol(lpCmdLine));
```

3. アプリケーションをビルドします。
4. サブプロジェクト .reg ファイルでレジストリ キーを追加し、起動時にアプリケーションを起動させます。次のラインを追加し、対応する Launch99 と Depend99 の入力を作成します。

```
[HKEY_LOCAL_MACHINE\INIT]
"Launch99"="Subproject_Shell.exe"
"Depend99"=hex:14,00, 1e,00
```

5. ランタイム イメージをビルドし、開始します。
6. Subproject_Shell アプリケーションが自動的に開始することを確認します。
7. Launch50 レジストリにおける Explorer.exe へのリファレンスを、次のように Subproject_Shell アプリケーションへのリファレンスに置き換えます。

```
[HKEY_LOCAL_MACHINE\INIT]
"Launch50"="Subproject_Shell.exe"
"Depend50"=hex:14,00, 1e,00
```

8. ランタイム イメージをビルドし、開始します。
9. 図　3 ミ 13　に示されたように、標準のシェルの部分で対象デバイスが Subproject_Shell アプリケーションを実行することを確認します。

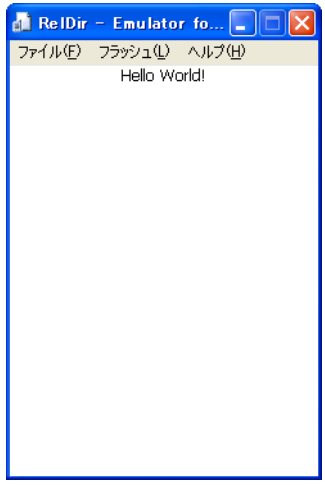


図 3 ミ 13　標準のシェルを Subproject_Shell アプリケーションに置き換え

本章のレビュー

Windows Embedded CE は、対象デバイスで最適なシステムパフォーマンスおよび電力消費を保証できるさまざまなツールや特徴、そして API を提供します。ILTiming、OSBench、Remote Performance Monitor といったパフォーマンス ツールは、ドライバやアプリケーション、そして OAL コードの中で、およびこれらの間で、デッドロックやその他スレッド同期に関連した問題等、パフォーマンスの問題を特定するのに役立ちます。Remote Kernel Tracker を使用した場合、Windows Embedded CE がネイティブにサポートする構造化例外処理に依存すると同時に、プロセスやスレッド実行を詳細に検証できます。

Windows Embedded CE はコンポーネント化された OS です。オプション コンポーネントを含めたり除外したり、標準のシェルをカスタム アプリケーションに置き換えたりすることもできます。標準のシェルを自動的に開始するよう構成されたアプリケーションに置き換えることで、キオスク構成を有効にする基盤が確立されます。Windows Embedded CE はキオスク構成でブラック シェルを使って実行されますが、その際、ユーザーはそのデバイス上で操作を開始したり、他のアプリケーションに切り替えたりすることはできません。

シェルが何かにかかわらず、デバイス ドライバやアプリケーションで電源管理機能を実行し、エネルギー利用を制御できます。Power Manager をデフォルトで実行することにより、典型的なニーズがカバーされますが、特別な要件を持つ OEM メーカーはカスタムロジックを追加します。Power Manager ソースコードを含む場合、Windows Embedded CE を使います。電源管理フレームワークは柔軟で、レジストリ設定によりデバイス電源状態をマッピングできるカスタマイズシステム電源状態がいくつであっても、それらをサポートします。

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。

- ILTiming
- Kiosk Mode
- Synchronization Objects
- Power Manager
- RequestDeviceNotifications

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。

ILTiming および OSBench ツールを使用

エミュレータ デバイス上で Iltiming と OSBench を使用し、エミュレートされた ARMV4 プロセッサのパフォーマンスを検証します。

カスタムシェルの実装

Task Manager を使い、ソースコードを使用して Windows Embedded CE に含まれた対象デバイスの外観をカスタマイズし、シェルを置き換えます。

マルチスレッド アプリケーションとクリティカル セクションを使った演習

グローバル変数へのアクセスを保護するため、マルチスレッド アプリケーションでクリティカル セクションのオブジェクトを使用します。

1. アプリケーションのメイン コードで 2 つのスレッドを作成し、スレッド機能で 2 秒間（Sleep(2000)）、無限ループで 3 秒間（Sleep(3000)）待機します。アプリケーションのプライマリ スレッドは、WaitForMultipleObjects 機能を使って両方のスレッドか終了するまで待機する必要があります。
2. グローバル変数を作成し、それに対し両方のスレッドからアクセスします。一本のスレッドは変数に書き込み、もう一本は変数を読み出す必要があります。最初の Sleep の前後で変数へアクセスし、その値を表示することで、同時アクセスを視覚化できます。
3. 両スレッド間で共有される CriticalSection オブジェクトを使用することで、変数へのアクセスを保護します。ループの最初にクリティカル セクションを取り込み、ループの終了時にリリースします。その結果を前回の出力と比較します。