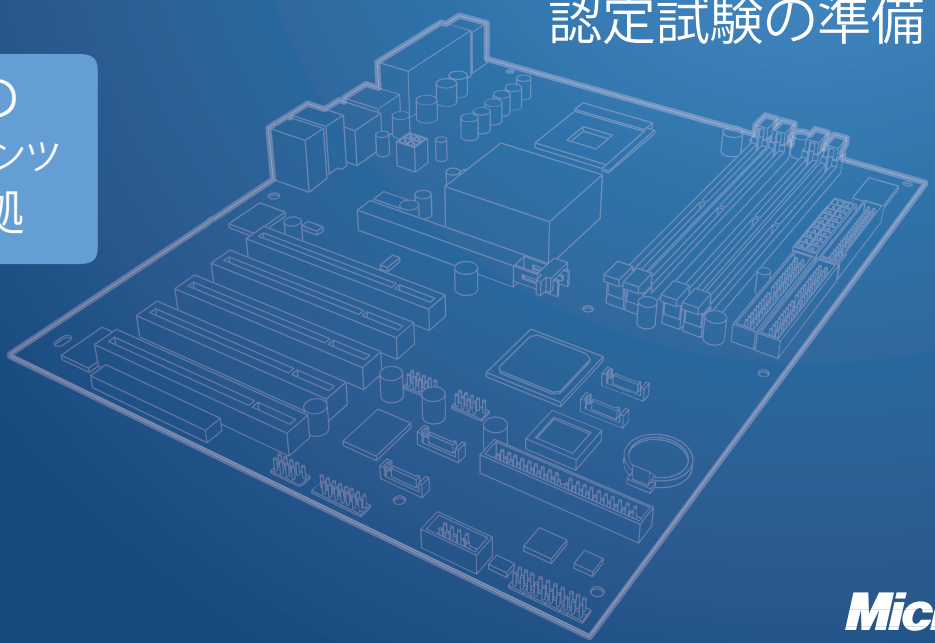
MCTS
Exam 70-571

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

目次

はじめに ..... **xi**
対象ユーザー xii
この本の特徴 xii
ハードウェアの必要条件 xiii
ソフトウェアの必要条件 xiii
表記規則 xiv
キーボード規則 xiv
注記 xv
付属 CD-ROM について xv
マイクロソフト認定プロフェッショナル プログラム xvi
技術サポート xvi

序文 ..... **xvii**

1 オペレーティング システム のカスタマイズ **1**
始める前に 2
レッスン 1: オペレーティング システム デザインの作成とカスタマイズ 3
オペレーティング システム デザイン 概要 3
OS デザインの作成 3
カタログ コンポーネント を使用した OS デザインのカスタマイズ 5
ビルド構成管理 6
OS デザイン プロパティ ページ 7
高度な OS デザイン構成 11
レッスン概要 13
レッスン 2: Windows Embedded CE サブプロジェクトの構成 14
Windows Embedded サブプロジェクト概要 14
サブプロジェクトを作成し、OS デザインに追加する方法 15
サブプロジェクトの構成 18
レッスン概要 19
レッスン 3: コンポーネントの複製 20
Public ツリーの編集とコンポーネントの複製 20
Public ソース コード 21
レッスン概要 22
レッスン 4: カタログ項目の管理 23
カタログ ファイル の概要 23
カタログのエントリの作成および編集 24

カタログ コンポーネントの依存関係 27
レッスン概要 27
レッスン 5: ソフトウェア開発キットの生成 28
ソフトウェア開発キットの概要 28
SDK の生成 28
SDK のインストール 30
レッスン概要 30
ラボ 1: OS デザインの作成、構成、およびビルド 31
第 1 章の復習 37
重要な用語 37
推奨する練習内容 37

2 ランタイム イメージのビルドおよび展開 39
始める前に 40
レッスン 1：ランタイム イメージのビルド 41
ビルド プロセスの概要 41
Visual Studio でのランタイム イメージのビルド 43
コマンド ラインからランタイム イメージをビルド 48
Windows Embedded CE ランタイム イメージ コンテンツ 49
レッスンの要約 59
レッスン 2：ビルド構成ファイルを編集 61
Dirs ファイル 61
Sources ファイル 63
Makefile ファイル 66
レッスン概要 66
レッスン 3：ビルド結果の分析 67
ビルド レポートを理解する 67
ビルドの問題のトラブルシューティング 69
レッスン概要 71
レッスン 4：ターゲット プラットフォームでのランタイム イメージの展開 72
展開方法の選択 72
デバイスへの接続 75
レッスン概要 75
演習 2 ランタイム イメージのビルドおよび展開 76
OS デザイン用にランタイム イメージをビルドする 76
接続オプションの構成 77
エミュレータの構成の変更 78
デバイス エミュレータでのランタイム イメージのテスト 79
本章のレビュー 81
用語 82

推奨される練習 82

3 システムのプログラミング 85
始める前に 86
レッスン 1: システム パフォーマンスの監視 と最適化 87
リアルタイム パフォーマンス 87
リアルタイム パフォーマンスの測定ツール 90
レッスン概要 96
レッスン 2: システム アプリケーションの実装 97
システム アプリケーションの概要 97
アプリケーションを起動時に開始する方法 97
Windows Embedded CE シェル 102
Windows Embedded CE コントロール パネル 104
キオスク モードの有効化 107
レッスン概要 108
レッスン 3：スレッドおよびスレッド同期の実装 109
プロセスとスレッド 109
Windows Embedded CE 上のスレッド スケジュール化 109
プロセス管理 API 110
スレッド管理 API 110
スレッド同期 116
スレッド同期に関するトラブルシューティング 122
レッスン概要 123
レッスン 4: 例外処理の実行 125
例外処理の概要 125
例外ハンドラ構文 127
終了ハンドラ構文 128
動的なメモリ割り当て 128
レッスン概要 131
レッスン 5: 電源管理の実行 132
Power Manager 概要 132
ドライバ電源状態 133
システム電源状態 134
アクティビティ タイマ 135
電源管理 API 137
電源状態構成 141
プロセッサのアイドル状態 143
レッスン概要 144
演習 3：キオスクモード、スレッド、電源管理 145
本章のレビュー 150

用語 150
おすすめの練習方法 151

4 システムのデバッグおよびテスト 153
始める前に 154
レッスン 1：ソフトウェア関連のエラーの検出 155
デバッグとターゲット デバイス コントロール 155
カーネル デバッガ 157
デバッグ メッセージ サービス 157
ターゲット コントロール コマンド 167
デバッガ拡張コマンド (CEDebugX) 168
詳細デバッガ ツール 170
アプリケーション検証ツール 171
CELog イベント追跡および処理 172
レッスン概要 176
レッスン 2：ランタイム イメージを構成してデバッグを有効にする 177
カーネル デバッガを有効にする 177
KITL 179
ターゲット デバイスのデバッグ 180
レッスン概要 184
レッスン 3：CETK を使用してシステムをテストする 185
Windows Embedded CE テスト キットの概要 185
CETK を使用する 187
カスタム CETK テスト ソリューションを作成する 192
CETK テスト結果を分析する 194
レッスン概要 195
レッスン 4：ブート ローダーをテストする 196
CE ブート ローダー アーキテクチャ 196
ブート ローダーのデバッグ テクニック 198
レッスン概要 199
演習 4：KITL、デバッグ領域、および CETK ツールに基づいたシステム デバッグおよびテスト 200
本章のレビュー 205
用語 206
おすすめの練習方法 206

5 ボード サポート パッケージのカスタマイズ 207
始める前に 208
レッスン 1：ボード サポート パッケージの適用と設定 209
ボード サポート パッケージ (BSP) の概要 209

ボード サポート パッケージ の適用 211
参照 BSP の複製 212
既存のライブラリからブート ローダーを実装する 215
OAL の適用 223
新しいデバイス ドライバの統合 227
設定ファイルの変更 228
レッスンの要約 228
レッスン 2：BSP のメモリ マッピングの構成 230
システム メモリ マッピング 230
メモリ マッピングと BSP 235
ドライバと OAL の間で共有されるリソースの有効化 236
レッスンの要約 237
レッスン 3：OAL への電源管理サポートの追加 238
電源状態の切り替え 238
アイドル モードでの電力消費の削減 239
システムの電源オフとサスペンド 240
クリティカル オフ状態のサポート 242
レッスンの要約 242
演習 5：ボード サポート パッケージ の適用 243
本章のレビュー 248
用語 249
おすすめの練習方法 249

6 デバイス ドライバを開発する 251
始める前に 252
レッスン 1：デバイス ドライバの基本を理解する 253
ネイティブおよびストリーム ドライバ 253
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 254
レッスン概要 256
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 257
デバイス マネージャ 257
ドライバ名前付け規則 258
ストリーム インターフェイス API 260
デバイス ドライバ コンテキスト 263
デバイス ドライバをビルドする 264
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 268
ドライバを動的にロードする 269
レッスン概要 270
レッスン 3：ドライバの構成とロード 272
デバイス ドライバ ロード手順 272

カーネル モデルおよびユーザー モデル ドライバ 279
レッスン概要 282
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 284
割り込み処理アーキテクチャ 284
割り込み識別子 (IRQ および SYSINTR) 288
ISR および IST 間の通信 291
インストール可能 ISR (IISR) 292
レッスン概要 294
レッスン 5：[デバイス ドライバ] 用電源管理を実装する 295
電源管理デバイス ドライバ インターフェイス 295
レッスン概要 298
レッスン 6：境界間のマーシャリング データ 300
基盤となるメモリ アクセス 300
物理メモリの割り当て 302
アプリケーション呼び出しバッファ 303
ポインタ パラメータの使用 304
埋め込みポインタの使用 304
バッファの取り扱い 305
レッスン概要 308
レッスン 7：ドライバ移植性の拡張 310
ドライバのレジストリ設定にアクセスする 310
割り込み関連レジストリ設定 311
メモリ関連レジストリ設定 312
PCI 関連レジストリ設定 312
バスを認識しないドライバの開発 313
レッスン概要 314
演習 6：デバイス ドライバの開発 315
本章のレビュー 320
用語 321
おすすめの練習方法 322

用語集 323

Index 327

著者について 347
Nicolas Besson 347
Ray Marcilla 347
Rajesh Kakde 348

はじめに

Microsoft Windows Embedded CE 6.0 R2 試験準備キットへようこそ。この準備キットの目的は、Windows Embedded CE 開発者にマイクロソフト認定テクノロジ スペシャリスト (MCTS) Windows Embedded CE 6.0 アプリケーション開発認定試験の準備をサポートすることです。

この準備キットを使用することによって、以下の試験目的において最大限の成績を収めることができます。

- オペレーティング システムのデザインをカスタマイズする。
- Windows Embedded CE コンポーネントを複製し、カタログ項目を管理する。
- ソフトウェア開発キット (SDK) を生成する。
- ランタイム イメージをビルドし、ビルド結果を分析する。
- ランタイム イメージの配置、監視、および最適化を行なう。
- マルチ スレッド システム アプリケーションを開発する。
- 例外操作を実装する。
- アプリケーション、デバイス ドライバ、および OEM アダプテーション層 (OAL) で電源管理をサポートする。
- ブート ローダーとメモリ マッピングへのカスタマイズを含むボード サポート パッケージ (BSP) の設定を行なう。
- 多機能ストリーム インターフェイス ドライバを開発する。
- 割り込みサービス ルーチン (ISR) と割り込みサービス スレッド (IST) を実装し、カーネル モードとユーザー モード コンポーネント間でデータを整理する。
- ソフトウェア関連のエラーを除外するためにカーネル モードとユーザー モード コンポーネントのデバッガを行なう。
- 開発ワーク ステーションとターゲット デバイスで標準およびユーザー定義テストを実行するために Windows Embedded CE テスト キット (CETK) を使用する。
- CETK ベースのテストでカスタム デバイス ドライバを含むために Tux 拡張コンポーネントを開発する。

対象ユーザー

この試験準備キットは、オペレーティング システム デザイン、システム コンポーネントのプログラミング、Windows Embedded CE プラットフォームでのデバッガについての基本レベルの知識を持つシステム開発者のためのものです。

特に、この準備キットは以下のスキルを持つ読者のために設計されています。

- Windows 開発と Windows Embedded CE 開発の基本的な知識を持つ。
- 少なくとも 2 年間の C または C++ プログラミング、および Win32 アプリケーション プログラミング インターフェイス (API) の経験がある。
- Microsoft Visual Studio 2005 と Windows Embedded CE 6.0 用 Platform Builder を熟知している。
- 標準 Windows デバッガ ツールを使用した基本的なデバッガ スキルを持つ。

詳細情報　試験 70-571 のためのユーザー プロファイル

認定試験に合格するための前提条件の情報については、http://www.microsoft.com/learning/exams/70-571.mspx にある 「the Preparation Guide for Exam 70-571」のユーザー プロファイル セクションを参照してください。

この本の特徴

各章の最初にはその章で扱われる試験目的のリストと 「始める前に」セクションがあり、その章を完了する準備をさせます。その後、章は各レッスンへと分けられます。各レッスンの最初には目的リストと予想学習時間の記載があります。レッスンのコンテンツは、トピックとレッスンの目的によってさらに分けられます。

各章の終わりには演習と章全体のレッスンの短い要約があります。その後に重要用語の短い確認と、その章の資料についての知識をテストしその章で提出されている試験目的を効果的にマスタするための推奨される練習があります。

その演習を用いて、特定の概念やスキルを実際に実行し、章のレッスンで学習したことをテストしてみることができます。すべての演習は、この段落の左にあるような箇条書き記号で識別されるステップ バイ ステップ手順を含みます。提示されている手順を効果的にマスタさせるため、各演習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があるワーク シートもこの本の付属物に含まれています。

演習を完了するために Microsoft Windows XP または Microsoft Windows Vista、Visual Studio 2005 Service Pack 1、および Windows Embedded CE 6.0 用 Platform Builder がインストールされた開発コンピュータが必要です。

ハードウェアの必要条件

開発コンピュータは以下の最低限の構成を満たしており、すべてのハードウェアは Windows XP または Windows Vista ハードウェア互換性リスト上にある必要があります。

- 1 GHz かそれ以上の 32 ビット (x86) または 64 ビット (x64) プロセッサ。
- 1 ギガバイト (GB) の RAM。
- 40 GB のハード ドライブと Visual Studio 2005 と Platform Builder で使用できる最低 20 GB のディスク スペース。
- DVD-ROM ドライブ。
- Microsoft マウスか互換性のあるポインティング デバイス。
- RAM 容量の 2 倍かそれ以上のページング ファイル セット。
- VGA 互換のディスプレイ。

ソフトウェアの必要条件

この過程でプロシージャを完了するために以下のソフトウェアが要求されています。

- Microsoft Windows XP SP2 または Windows Vista。
- Microsoft Visual Studio 2005 Professional Edition。
- Microsoft Windows Embedded CE 6.0。
- Microsoft Visual Studio 2005 Professional Edition SP1。
- Microsoft Windows Embedded CE 6.0 SP1。
- Microsoft Windows Embedded CE 6.0 R2。

注 Visual Studio 2005 と Windows Embedded CE 6.0 の試用版

Visual Studio 2005 と Windows Embedded CE 6.0 のインストール ガイドラインと評価版は、マイクロソフトのウェブサイト http://www.microsoft.com/japan/windows/embedded/eval/trial.mspx で入手できます。

表記規則

- ユーザーが入力する文字やコマンドは二重引用符 " " で囲まれます。
- 構文ステートメントでの山かっこ < > は変数情報のためのプレースホルダを示します。
- 本のタイトルはかぎかっこ「 」で示されます。
- ユーザーがファイル名を直接入力する場合を除き、ファイル名は < ファイル名 > または < ファイル名 > ファイルの形で、フォルダ名は < フォルダ名 > フォルダの形で示されます。ユーザーは別途記載されているときを除いて、ダイアログ ボックスやコマンド プロンプトにファイル名を入力するときにすべての小文字のアルファベットを使用することができます。
- ファイル名拡張子はすべて小文字です。
- 略語はすべて大文字です。
- 固定ピッチ フォントはコード サンプル、画面テキスト例、またはコマンド プロンプトや初期化ファイルにユーザーが入力しなければならないテキストを表しています。
- 構文ステートメントで使用される角かっこ [ ] は選択項目を囲むのに用いられます。例えば、コマンド構文での [ファイル名] はコマンドで入力するファイル名を選択できることを示しています。かっこ内の情報だけを入力し、かっこ自体は入力しないでください。
- 構文ステートメントで使用される中かっこ { } は必須項目を囲むのに用いられます。中かっこ内の情報だけを入力し、中かっこ自体は入力しないでください。

キーボード規則

- 2 つのキー名の間の正符号 (+) はこれらのキーを同時に押すことを意味します。例えば、"Alt + Tab キーを押します " は、Tab キーを押す間 Alt キーを押し続けることを意味します。
- 2 つまたはそれ以上のキー名の間の読点 (、) は各キーを同時にではなく順に押すことを意味します。例えば、"Alt、F、X キーを押します " は順に各キーを押して離すことを意味します。"Alt + W、L キーを押します " は、まず Alt キーと W キーを同時に押し、それらを離してから、L キーを押すことを意味します。
- キーボードでメニュー コマンドを選択することもできます。Alt キーを押してメニュー バーをアクティブにし、メニュー名とコマンド名でハイライ

ト表示されたり下線が引かれたりしている文字に対応するキーを順に押します。いくつかのコマンドでは、メニューで挙げられているキーの組み合わせを押すこともできます。

- またキーボードでダイアログ ボックスのチェック ボックスやオプション ボタンをオンやオフにすることもできます。Alt キーを押して、その後オプション名の下線が引かれている文字に対応したキーを押します。またはそのオプションがハイライト表示されるまで Tab キーを押し、その後チェック ボックスやオプション ボタンがオンやオフになるようスペース バーを押します。
- Esc キーを押すことによってダイアログ ボックスの画面をキャンセルすることができます。

注記

レッスン中にいくつかのタイプの注記が現れます。

- ヒントと書かれた注記は、発生する可能性のある結果や代替法の説明を含んでいます。
- 重要と書かれた注記は、タスクを完了するために必要不可欠な情報を含んでいます。
- 注と書かれた注記は、補足情報を含んでいます。
- 注意と書かれた注記は、起こる可能性のあるデータの損失についての警告を含んでいます。
- 試験のヒントと書かれた注記は、試験の指定事項や目的についての有用なヒントを含んでいます。

付属 CD-ROM について

付属 CD は、この本を通して使用される可能性のある、さまざまな情報補助を含んでいます。これには詳細なステップ バイ ステップ インストラクションのあるワークシートや実践練習で使用されるソース コード、無償の技術情報、マイクロソフト開発者からの記事が含まれています。

この本の電子版 (電子ブック) にはさまざまな入手可能な表示オプションが含まれています。付属 CD はまたこの公式自己ペース学習ガイドの印刷本を生産するためのポスト プレス ファイルの完全セットを含んでいます。圧縮ファイルは PDF (Portable Document Format) 形式で、専門的な印刷と製本のための必要なトンボが付いています。

マイクロソフト認定プロフェッショナル プログラム

マイクロソフト認定プロフェッショナル (MCP) プログラムは、受験者の現行のマイクロソフト製品と技術に対する理解力を証明する最高の方法を提供します。試験と対応する認定は、受験者のマイクロソフト製品と技術を用いてソリューションのデザインと開発をし、または実装とサポートをする重要な熟練度を証明するために開発されました。マイクロソフト認定となるコンピュータ プロフェッショナルは専門家として認識され、業界中で求められています。認定は個人、雇用者、そして組織にとってさまざまな益をもたらします。

詳細情報　すべてのマイクロソフト認定

マイクロソフト認定のすべてのリストについては、http://www.microsoft.com/japan/learning/mcp/default.mspx を参照してください。

技術サポート

この本と付属 CD のコンテンツの正確性を確実にするために最大の努力が払われました。Windows Embedded CE 開発に関する意見、質問や見解がありましたら、マイクロソフト製品サポート サービス (PSS) 、Microsoft Developer Network (MSDN) 、または以下のブログ サイトを通して、Windows Embedded CE 専門家と連絡を取ってください。

- Nicolas BESSON のウェブログ http://nicolasbesson.blogspot.com で Windows Embedded CE 6.0 試験準備キットの筆頭製作者と連絡を取り、これらの話題と関係づけられた新しい記事についての感想や話題意見をお寄せください。
- Windows Embedded ブログ http://blogs.msdn.com/mikehall/default.aspx で、Mike Hall の Windows Embedded でのコツ、ヒント、思いついた考えを読んでください。
- Windows CE ベース チーム ブログ http://blogs.msdn.com/ce_base/default.aspx でマイクロソフト開発者から直接 Windows Embedded CE のカーネルとデータ記憶技術、システム ツールについてのプロジェクト背景的情報を得てください。

詳細情報　Windows Embedded CE 製品サポート

入手できるすべての Windows Embedded CE 製品サポート オプションの詳細な情報については、http://www.microsoft.com/japan/windows/embedded/supportresources.mspx を参照してください。

序文

成功を収めた 12 年が過ぎ、多くのことが変わりましたが、私たちが Windows CE 1.0 を市場へと発売したのは昨日のことのようにも思われます。新しいテクノロジが現れ、他のものは消えていきましたが、私たちはパートナたちとともに新しいハードウェアとソフトウェアの新技術から最大の益を得るよう前進し続けてきました。Windows Embedded CE は進化し続けていますが、多数のプロセッサ アーキテクチャや、ロボット、携帯超音波画像システム、産業用制御装置、リモート センサとアラーム システム、POS 端末、メディア ストリーミング装置、ゲーム コンソール、シン クライアント、さらにはかつて私たちの誰もがマイクロソフト オペレーティング システムとは関連付けて考えなかったであろうデバイスを含むさまざまなデバイスとの組み合わせで実行される、軽量でリアル タイムな組み込みオペレーティング システムとして使われています。もしかしたら、いつか Windows Embedded CE は月面に置かれたデバイス上で実行するかもしれません。そうなっても驚かないでしょう。Windows Embedded CE はコンピュータ デバイスが生活をより楽により楽しくしているすべての場所に存在し得ます。

このプロジェクトが始まってすぐに、私たちは統合開発ツール セットを開発したり、Windows プログラミング インターフェイスとフレームワークをサポートしたりして、専門の組み込み開発者のニーズに焦点を合わせてきました。その後、私たちは開発者がオペレーティング システムをカスタマイズし、そのオペレーティング システムのためにアプリケーションを自由にビルドすることができるように、Windows Embedded CE 開発ツールを Visual Studio 2005 で統合しました。現在、Windows Embedded CE 6.0 は最高の x86、ARM、MIPS それに SH4 といったプロセッサをサポートしており、およそ 700 もの選択可能なオペレーティング システム コンポーネントを含んでいます。CE は構成、ビルド、ダウンロード、デバッグ、オペレーティング システム イメージとアプリケーションのテストに必要なツールを提供し、カーネルやデバイス ドライバ、また他の機能のためのソース コードとともに出荷され、アプリケーション開発者に Win32、MFC、または ATL ネイティブ コード アプリケーションや ..NET Compact Framework によるマネージ アプリケーションを作成するなどの選択肢を提供しています。マイクロソフト シェアード ソース イニシアティブの一部として、私たちは 250 万以上の CE ソース コードを出荷しました。これらのコードは開発者が見て、変更し、リビルドし、変更したソースを含む製品を販売することができるものです。そして最近、ホビイストの開発者が低価格なハードウェアと

CE ソフトウェア開発ツールを入手できるよう "Spark your Imagination" プログラムを開始しました。

2008 年 5 月に発表されたマイクロソフト認定テクノロジ スペシャリスト (TS) 試験 70-571「Windows Embedded CE 6.0 開発」のために、CE オペレーティング システム、開発ツール、またこの準備キットに関して多くの情報を見つけることができます。MCPにより、組み込みシステムの開発者は Windows Embedded テクノロジに基づく組み込みソリューションの開発に関するスキルを客観的に評価・実証することができるようになり、またその知識と熟練について認知されることが可能になりました。CE 6.0 に興味があるのであれば、この試験を受けることを考慮すべきです。私たちは、この本が皆さんのお役に立てることと確信しています。

マイクロソフト開発チームのすべてのメンバーより。

Mike Hall
Windows Embedded アーキテクト
Microsoft Corporation

第 1 章

オペレーティング システムのカスタマイズ

Windows ィ Embedded CE 6.0 R2 をターゲットデバイスに展開するときは、必要なオペレーティング システム (OS) コンポーネント、機能、ドライバ、構成の設定を含むランタイム イメージを使用しなければなりません。ランタイム イメージは OS デザインのバイナリ表示です。OS デザインを作成またはカスタマイズするか、対応ランタイム イメージを生成するには、Windows Embedded CE 6.0 用 Microsoft ィ Platform Builder を使用できます。OS デザイン毎に、Microsoft ィ Visual Studio ィ 2005 で新規の開発プロジェクトを通常に作成して自分用のターゲット デバイスとアプリケーションに必要なコンポーネントだけを含めます。これにより、オペレーティング システムのスペースを縮小し、ハードウェア用件を低減します。しかし、コンパクトで機能的なランタイム イメージを作成するには、Platform Builder（ユーザ インターフェイス (UI)、カタログ コンポーネント、ビルド処理の特性）をより良く理解する必要があります。本章では、OS デザインの作成方法と、Windows Embedded CE ランタイム イメージの新規生成方法を説明します。

本章の試験対象 :

- OS デザインの作成およびカスタマイズ
- Windows Embedded CE サブプロジェクトの構成
- コンポーネントの複製
- カタログ項目の管理
- ソフトウェア開発キット (SDK) の生成

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 :

- Windows Embedded CE でのソフトウェア開発の基本的な知識を有すること。
- Windows Embedded CE 6.0 R2 用 Platform Builder のディレクトリ構成とビルド処理の基本的な理解していること。
- Windows Embedded CE ランタイム イメージのバイナリ作成とターゲット デバイスへのダウンロードに精通していること。
- SDK を使用した、Windows Embedded CE 用アプリケーション開発の経験があること。
- 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について。

レッスン 1: オペレーティング システム デザインの作成とカスタマイズ

目的に適っている Windows Embedded CE 6.0 R2 の提供する機能をできるだけ多く（または少なく）使用して、Visual Studio 2005 の Platform Builder で OS デザインを作成できます。例えば、ポータブル マルチメディア デバイスなどの特定なターゲット デバイス用 OS デザインを作成する、またリモート プログラム可能なワイアレス デジタル サーモスタット用の OS デザインを作成することができます。この二つのターゲット デバイスは、同じハードウェアに依存する可能性がありますが、各々の目的が異なり、OS デザイン要件も異なります。

このレッスンを終了すると、以下をマスターできます :

- OS デザインの役割と特性を理解できる。
- OS デザインを作成、カスタマイズ、使用できる。

レッスン時間 (推定): 30 分

オペレーティング システム デザイン 概要

OS デザインは、ランタイム イメージに含まれるコンポーネントと機能を定義し、本質的に、Visual Studio の Windows Embedded CE 6.0 R2 用 Platform Builder プロジェクトに対応します。OS デザインは下記の要素のいずれか、またはすべてを含みます :

- カタログ項目 (ソフトウェアコンポーネントとドライバを含む)
- サブプロジェクト形式の追加ソフトウェア コンポーネント
- カスタム レジストリ設定
- ビルドオプション (ローカリゼーションまたは Kernel Independednt Tranport Layer (KITL) を基とするデバッグ)

さらに、各 OS デザインは少なくとも一つのデバイス ドライバのボード サポート パッケージ (BSP) への参照を含み、これには ハードウェア特定 ユーティリティ、OEM アダプテーション層 (OAL) を含まれます。

OS デザインの作成

Windows Embedded CE には、OS デザインを容易に作成できる OS デザイン ウィザードがあります。ウィザードを起動するには、Windows Embedded CE 6.0 R2 用 Platform Builder を組込み済みの Visual Studio 2005 を起動し、[ファイル]

メニューの [新規作成] をポイントして、[プロジェクト] をクリックすると、[新しいプロジェクト] ダイアログ ボックスが表示されます。ダイアログの [プロジェクト種類] の下にある [CE 6.0用Platform Builder] を選択し、[Visual Studio にインストールされたテンプレート] の下にある [OS デザイン] を選択し、OS デザイン名を [名前] フィールドに入力してから [OK] をクリックすると、Windows Embedded CE 6.0 OS デザイン ウィザードが起動します。

OS デザイン ウィザードから、よく使われるオプションとすでに選択された カタログ コンポーネントが付いた形で BSP とデザイン テンプレートを使用できます。ウィザードで定義したすべての設定はあとで変更できますので、今は個々の設定にこだわる必要はありません。デザイン テンプレート ページで選択したテンプレートにもよりますが、OS デザイン ウィザードは選択テンプレートに関連した特定のオプション デザイン テンプレートバリエーション ページを追加の形で表示する場合があります。例えば、Windows シン クライアント、Enterprise Terminal、および Windows Network Projector は、すべてリモート デスクトップ プロトコル (RDP) を使用するデバイスであり、また同じ シン クライアント デザイン テンプレートのバリエーションでもあります。選択したテンプレートとバリエーションにもよりますが、OS デザイン ウィザード は OS デザイン内の特定のコンポーネントを含めるために追加ページを表示する場合があります (例：ActiveSync ィ、WMV/MPEG-4 Video Codec、あるいは IPv6)。

OS デザイン テンプレート

CE 6.0 OS デザイン テンプレートは、ある特定の用途で Windows Embedded CE を使用するために必要な カタログ コンポーネントのサブセットです。OS デザインの新規作成時にテンプレートは必ずしも必要ではありませんが、テンプレートを使用するとかなりの時間を節約することができます。[カタログ項目ビュー] . で、カタログ コンポーネントを選択すると、あとでカタログ コンポーネントを簡単に変更できます。

適切なテンプレートを選択することで、開発時間と労力を節約できます。例えば、あなたが 展示会で新しい評価ボードをデモする必要があるとします。その場合、PDA デバイス やコンシューマ メディア デバイス デザイン テンプレートから始めて、OS デザイン ウィザードの中で必要なコンポーネントと共通 Windows アプリケーション (例：.NET Compact Framework 2.0、Internet Explorer ィ、および WordPad) を追加することをお勧めします。一方、コントローラ エリア ネットワーク (CAN) コントローラのドライバを開発する際は、スモール フットプリント デバイス デバイスのデザイン テンプレート から始めて必要不可欠なコンポーネントだけを追加することで、ランタイム イメージのサイズと起動時間を最小化できます。

OS デザイン ウィザードは柔軟性があり、カスタム デザイン テンプレートをサポートしています。テンプレートファイルは、拡張マークアップ言語 (XML) ドキュメントで %_WINCEROOT%\Public\CEBase\Catalog フォルダにあります。すでに存在する Platform Builder カタログ XML (PBCXML) ファイルのコピーから始めて、必要により PBCXML の構造を変更してください。Visual Studio を起動し、Visual Studio の [カタログ項目ビュー] を更新すると、Platform Builder は自動的にカタログ フォルダ内のすべての .pbcxml ファイルを列挙します。

カタログ コンポーネント を使用した OS デザインのカスタマイズ

OS デザイン ウィザードの終了後に OS デザインを簡単に変更することができます。カタログは、OS デザインに追加するすべてのコンポーネント用のリポジトリです。統合化開発環境 (IDE) 内から直接アクセスできます。ソリューション エクスプローラ ウィンドウ ペイン内の [カタログ項目ビュー] をクリックしてください。CE の機能のほとんどは、ActiveSync から TCP/IP まで ユーザが選択可能なカタログ コンポーネントに分割されています。これらのコンポーネントはユーザ インターフェイス (UI) 内から直接選択できます。各カタログ項目は、ランタイム イメージのビルドおよび統合に必要なコンポーネントの参照です。

他のカタログ項目に依存するカタログを追加するとき、暗黙的に OS デザインにも依存して追加します。これらの項目は、依存関係により OS デザインの一部を表し、カタログ項目ビュー内で緑色の四角のチェックボックスとして表示されます。一方、カタログ項目ビューは手動で選択された項目と、緑のチェックマーク付きのデザイン テンプレートに基づいて含まれた項目を表示します。

カタログ項目ビューで、すべてのカタログ コンポーネントまたはフィルタを使用して選択したカタログ項目のみを表示できます。. ソリューション エクスプローラ のカタログ項目ビューの左上隅にある [フィルタ] ボタン上の下向き矢印を押してフィルタを適用するか、カタログの [すべてのカタログ項目] オプションを選択してカタログ項目すべてを表示してください。

カタログ項目名か SYSGEN 変数コンポーネントセットを知っていると、手動で探すよりも追加または削除するカタログ項目を検索した方が便利な場合があります。項目名か SYSGEN 変数での検索では、カタログ項目ビューの一番上にあるテキストボックスに文字を入力してから緑の矢印をクリックしてください。

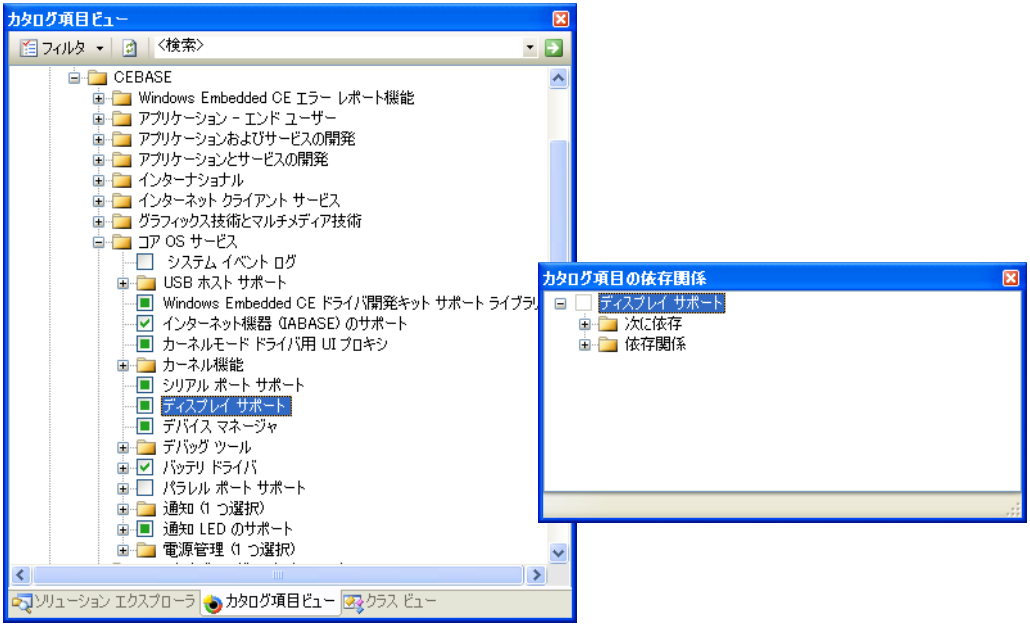


図 1-1　カタログ項目ビューの検索ボックスとカタログ項目の依存関係ウィンドウ

カタログ項目の依存関係を分析するには、項目を右クリックし [依存関係の表示] を選択すると カタログ項目依存ウィンドウが表示されます (図 1-1)。例えば、この機能を利用してある特定のカタログ項目が依存関係によって追加された理由を調べることができます。CE 6.0 R2 では、Platform Builder は動的にカタログをスキャンして選択項目に依存するすべてのコンポーネントだけでなく依存されるコンポーネントを列挙します。

ビルド構成管理

Windows Embedded CE は別々に変更可能な複数のビルド構成をサポートします。標準設定は、リリースとデバッグの 2 種類です。これらは OS デザイン作成時に自動的に利用可能になります。デバッグビルド構成では、コンパイラはデバッグ情報を生成し、デバッグを容易にする ソース コードへのリンクを プログラム データベース ファイル (.pdb) に保管し、段階的なコード実行はコードの最適化をしません。デバッグ構成でコンパイルされた Windows Embedded CE ランタイム イメージは、リリース設定でコンパイルしたイメージよりも 50 から 100 パーセント大きくなります。Visual Studio の [ビルド メニュー] を開き、[構成管理] をクリックして構成管理ダイアログ ボックス内の [アクティブ ソリューション構成] から、目的のビルド構成を選択します。標準のツールバーのプルダウン メニューを使用して、希望のビルド構成を選択することもできます。

OS デザイン プロパティ ページ

ビルド構成毎に、多くのプロジェクト プロパティ (例：ロケール、KITL を含める / 含めない、カスタム ビルド アクション、バイナリイメージのサブプロジェクト包含、カスタム SYSGEN 変数) を設定することができます。ソリューション エクスプローラ の [OS デザイン ノード] を右クリックして、[プロパティ ページ] ダイアログ ボックス を表示し [プロパティ] を選択すると、オプションにアクセスできます。OS デザイン ノードは、ソリューション最上位ノードの最初の子オブジェクトです。「OSDesign1」のように、プロジェクト名に対応します。ソリューション エクスプローラ が表示されないときは、[表示] メニューを開き [ソリューション エクスプローラ] をクリックします。カタログ項目ビューかクラス ビューが表示されているときは、[ソリューション エクスプローラ] タブをクリックしてソリューション ツリーを表示します。

ヒント　複数の構成のプロパティ設定

[プロパティ ページ] ダイアログ ボックスの左上隅にあるリスト ボックスからビルド構成を選択します。他のオプションは、いずれかまたはすべての構成を選択できます。度に複数のビルド構成のプロパティを設定するときに便利です。

ロケール オプション

[プロパティ ページ] ダイアログ ボックスの [構成プロパティ] に Windows Embedded CE イメージの言語設定を行う ロケール ノードがあります (図 1-2)。ロケール プロパティ ページは OS デザインのローカライズに必要な条件をカバーしますが、日本語のような東アジア言語は追加カタログ コンポーネントを必要とします。いくつかのインターナショナライゼーション (国際化) に関するカタログ コンポーネントを使用すると、ランタイム イメージのサイズがとても大きくなる場合があります。

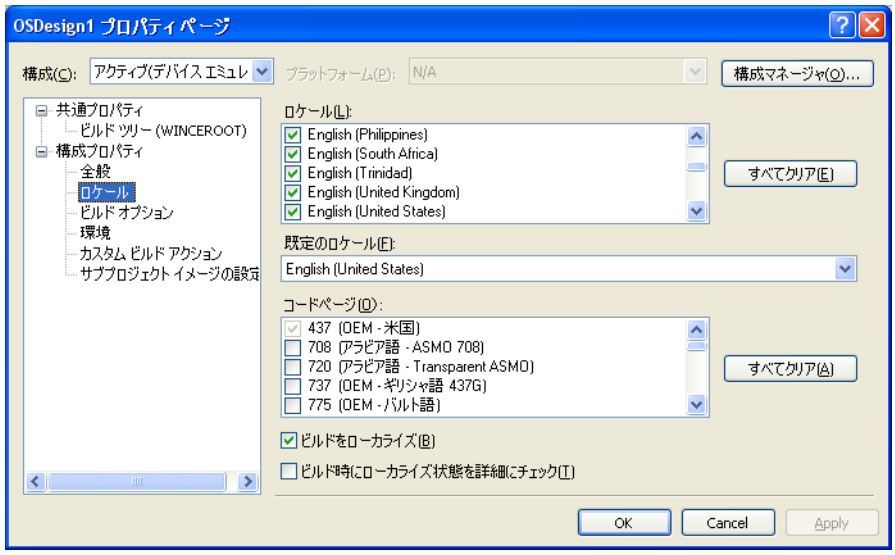


図 1-2　ロケール プロパティ ページ

ロケール プロパティ ページでランタイム イメージ用の以下のオプションを設定できます :

- ロケール　ランタイム イメージをローカライズする言語を選択します。選択言語にデフォルト ANSI と OEM コードページがある場合は、[コードページ] リストに表示されるとおり、対応するコードページが OS デザインに自動的に追加されます。
- デフォルト ロケール　OS デザイン用のデフォルトロケールを定義します。デフォルト言語は 英語 (米国)、コードページは 437 (OEM - 米国) です。
- コード ページ　OS デザインで利用可能な ANSI と OEM コードページを特定します。
- ビルドをローカライズ　ビルド プロセスにローカライズされた文字列とイメージ リソースを使用させます。Platform Builder は、OS デザイン ビルド プロセスのイメージ作成ステップ中に OS デザインのローカライズをします。ローカライズされたリソースファイルは、res2exe によって共通コンポーネント用バイナリファイルに統合されます。
- ビルド時にローカライズ状態を詳細にチェック　ローカライズ リソースが不足している場合、デフォルト ロケールのリソースを使用せずにビルド プロセスを中断します。

ビルド オプション

[プロパティ ページ] ダイアログ ボックスのロケール ノードのすぐ下に、イベント トラッキング、デバッグ、アクティブな OS デザインのビルドオプションを制御できる ビルド オプション ノードがあります (図 1-3)。

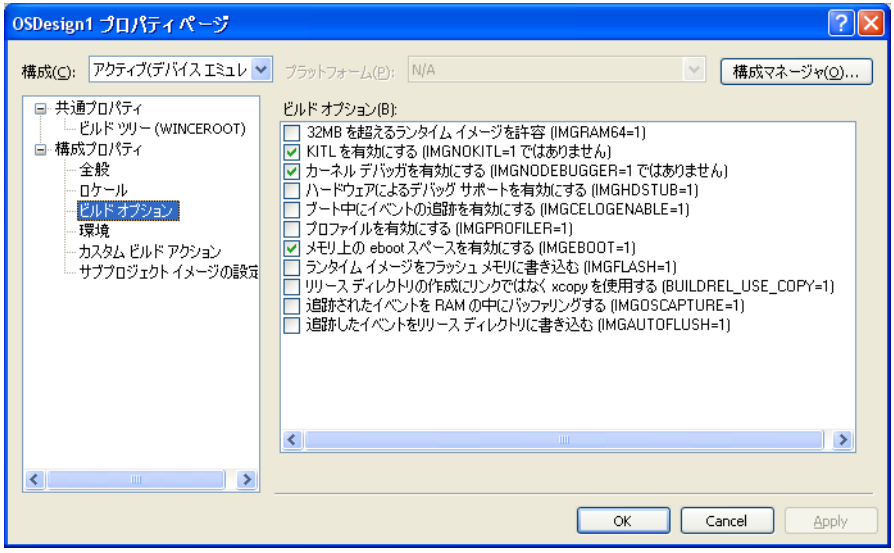


図 1-3 ビルド オプション プロパティ ページ

ビルド オプション プロパティ ページでランタイム イメージ用の以下のオプションを設定できます :

- 追跡されたイベントを RAM の中にバッファリングする　Platform Builder は CE イメージ内に OSCapture.exe を含めます。また OSCapture.exe によって追跡されたオペレーティング システム イベントを RAM にログすることで、ファイルにフラッシュしてあとで見ることができます。
- メモリ上の Eboot スペースを有効にする　起動時に Ethernet boot loader (EBOOT) はデータを Windows Embedded CE OS に渡せるようにします。
- ブート中にイベントの追跡を有効にする　起動時に通常よりも CE イベント ログ収集をもっと早く可能にします。If you activate this option, event tracking starts before most of the kernel and file system initialization is complete.
- ハードウェアによるデバッグ　サポートを有効にする　サードパーティのハードウェア デバッグ ツールに必要です (JTAG は exdi2 との準拠を調査します)。

- カーネル デバッガを有効にする　Windows Embedded CE デバッガを有効にしてランタイム イメージ内のステップ実行ができます。カーネル デバッグには、KITL と Platform Builder の実行時の通信が必要です。
- KITL を有効にする　KITL をランタイム イメージに追加します。KITL は便利なデバッグツールで、開発者がカーネル デバッガを使用したり、リモートデバイスのファイルシステム・レジストリ・他のコンポーネントと対話したりコードを実行できます。オペレーティング システムの最終ビルドに KITL を入れると、オーバーヘッドを引き起こし起動時にホスト コンピュータとの接続時間を浪費するので入れないでください。
- プロファイルを有効にする　ランタイム イメージ内にカーネル プロファイラを有効にし、タイミングとパフォーマンスデータの収集と表示を可能にします。ターゲット デバイス上の Windows Embedded CE のパフォーマンスを最適化するには カーネル プロファイラは便利なツールです。
- 追跡したイベントをリリース ディレクトリに書き込む　ランタイム イメージに CeLogFlush.exe を追加します。CeLogFlush.exe は OSCapture.exe によって収集したログ データを開発コンピュータ上のリリースディレクトリの Celog.clg ファイルに自動的に書き込みます。
- 32MB を超えるランタイム イメージを許容　32 MB を超えるイメージの作成を可能にします。しかし 64 MB を超えるイメージをビルドする場合はこのオプションを使用しないでください。その場合、環境変数を設定してください (例 : IMGRAM128)。
- リリース ディレクトリの作成にリンクではなく xcopy を使用する　リンクではなく xcopy を使用して実際にファイルをコピーします。Copylink を使用すると、開発コンピュータ上で NTFS が必要になる ハードリンクを作成する可能性があります。
- ランタイム イメージをフラッシュ メモリに書き込む　EBOOT はランタイム イメージをターゲット デバイスのフラッシュ メモリに書き込みます。

環境オプション

[プロパティ ページ] ダイアログ ボックスにある [環境 オプション] は、ビルド処理に使用される環境変数設定に使用します。Windows Embedded CE 6.0 R2 のほとんどの機能はカタログ コンポーネントで有効化できますが、いくつかのオプションは、Platform Builder がランタイム イメージで必要なコードをコンパイルするために SYSGEN 変数を設定する必要があります。BSP の開発ではビルド処理に影響する環境変数の設定は有効な場合があります。環境変数は、Windows Embedded CE ビルド処理中にコマンド ラインからアクセス可能です。ソース、

バイナリイメージ ビルダ (.bib)、レジストリファイル (.reg) の情報を特定する場合も環境変数を使用できます。

ヒント　デバッグ では動作するがリリースで動作しない
デバッグ構成でランタイム イメージをビルドできるのに、リリース構成でビルドできない場合は、[プロパティ ページ] ダイアログ ボックスを表示し、構成リストボックスの [すべての構成] から環境オプションを選択してデバッグとリリース両方に同じ環境変数を設定します。

高度な OS デザイン構成

このセクションでは OS デザインに関する高度なトピックをいくつか説明します。特に、同じ OS デザインを持つ複数のプラットフォームのサポート方法 と、OS デザインにある通常のファイル ロケーションとファイル タイプについて説明します。

OS デザインと複数プラットフォームの関連付け

OS デザイン ウィザードを使用して新しい OS デザインを作成するとき、ボード サポート パッケージ ウィザード ページでひとつ以上の BSP を選択できます。OS デザインを複数の BSP に関連付けると、, 複数のプラットフォーム用に同じコンテンツを使用して別のランタイム イメージを生成することができます。この場合、プロジェクトに複数の開発チームが関わっていて特にターゲット ハードウェアが利用できない時に便利です。例えば、エミュレータ ベースのプラットフォーム用ランタイム イメージを生成できるので、ハードウェアの最終版が出来上がる前にアプリケーション開発チームは作業を開始できます。OS 機能性に関しては、アプリケーション開発チームはターゲット プラットフォームの最終版が出来上がる前にアプリケーション プログラミング インターフェイス (API) を使用できます。2 つのランタイム イメージは同じコンポーネントと構成設定を共有するので、API は最後のターゲットに含まれます。

初期作成のあとに複数のプラットフォーム用のサポートをOS デザインに追加できます。ソリューション エクスプローラの [カタログ項目ビュー] 内の BSP 下にある対応するチェックボックスを選択します。BSP を選択するとリリースとデバッグ構成に追加のプラットフォームを自動的に追加します。Visual Studio の [ビルド] メニューにある [構成マネージャ] から、違うプラットフォームとビルド構成の切り替えを行えます。しかしプラットフォーム毎に時間のかかる SYSGEN フェーズを入れてすべてのビルド処理を実行する必要があります。

OS デザイン パスとファイル

OS デザインを使用して再頒布するには、構成ファイルと開発コンピュータ上のロケーションをを確実に知ることが必要です。OS デザインのデフォルト ロケーションは、「%_WINCEROOT%\OSDesigns」です。各プロジェクトはそれぞれ子ディレクトリに対応します。OS デザインは次のファイルとディレクトリ構成に対応します :

- <ソリューション名>　Visual Studio がプロジェクト用に作成した親ディレクトリ。
  - < ソリューション名 >.sln　OS デザインプロジェクトの設定を保存する Visual Studio ソリューションファイル (.sln)。通常、ファイル名は OS デザインと同じになります。
  - < ソリューション名 >.suo　ソリューション エクスプローラ ビューの状態などユーザ依存の情報を含む Visual Studio ユーザ オプション ファイル (.suo)。通常、ファイル名は OS デザインと同じになります。
  - <OS デザイン名 >　OS デザイン プロジェクトに含まれる残りのファイルの親ディレクトリ。
    - <OS デザイン名 >.pbxml　OS デザインのカタログ ファイル。選択したカタログ コンポーネントの参照と、OS デザインに関するすべての設定が含まれます。
    - サブプロジェクト　このディレクトリは OS デザインの一部として作成されたサブプロジェクトのサブフォルダをそれぞれ含みます。
    - SDK　このディレクトリは OS デザイン用に作成されたソフトウェア開発キット (SDKs) を含みます。
    - Reldir　リリースディレクトリ。Platform builder は、ターゲット デバイスにダウンロード可能なランタイム イメージの作成中に、ファイルをこのディレクトリにコピーします。
    - WinCE600　SYSGEN フェーズの完了後、現在の OS デザイン用のリソースファイルや構成ファイルがコピーされます。

ソース コントロール ソフトウェア の留意事項

基本的に OS デザインは Windows Embedded CE ランタイム イメージを生成するための Platform Builder 構成ファイルの集まりです。開発チームでソース コントロール ソフトウェアを使用する場合、これらの構成ファイルを ソース コントロール リポジトリに保管するだけで済みます。CE SYSGEN フォルダ (ランタイムイメージのビルド処理中に使用される) や Reldir ディレクトリからのファ

イルは、Platform Builder と BSP がインストールされたどのワークステーションでも再構成できるためリポジトリに含める必要はありません。また、拡張子が .user や .suo で終わるファイルは IDE 用のユーザ固有設定のため省略することができます。.ncb ファイルは IntelliSense ィ データを含むため省略できます。

レッスン概要

Windows Embedded CE 6.0 R2 用 Platform Builder にある OS デザイン ウィザードを使って OS デザインをすばやく簡単に作成することができます。一つまたは複数の BSP を選択して、ターゲット プラットフォーム用ハードウェア固有デバイス ドライバおよびユーティリティと、カタログ項目を追加するためのテンプレート バリエーションが付いたデザイン テンプレートを含めることができます。OS デザイン ウィザード終了後、OS デザインをさらにカスタマイズできます。不必要なカタログ項目を削除したり、コンポーネントを追加したり、デバッグおよびリリースビルドオプションなどのプロジェクト プロパティを設定できます。デバッグ ビルド構成で Platform Builder は、リリース ビルドに比較して 50 ～ 100 パーセント増のランタイム イメージ デバッグ情報を含みます。しかしデバッグ ビルドは開発中にコードのデバッグやステップ実行ができます。デバッグとリリース ビルド オプションを別々に構成できるため、OS デザインをデバッグ構成でコンパイルできるものの、リリース構成でコンパイルできないという状況に直面する場合があります。この場合、デバッグとリリースの構成に全く同じ環境変数をセットするとうまくいく場合があります。OS デザインを配布する場合、検索可能なデフォルトのディレクトリ %_WINCEROOT%\OSDesigns にソースファイルを配置する必要があります。開発チームで作業を調整するために、ソース 管理 ソフトウェアを使用することができます。

レッスン 2: Windows Embedded CE サブプロジェクトの構成

サブプロジェクトは、比較的独立したコンポーネントを全体ソリューションに含めるための親プロジェクトに加えられた Visual Studio プロジェクトです。この場合、親プロジェクトは OS デザインに対応します。サブプロジェクトは以下のフォームになります :

- アプリケーション (マネージまたはネイティブ)
- ダイナミック リンク ライブラリ (DLL)
- スタティック ライブラリ
- 構成設定のみを含む空のプロジェクト

サブプロジェクトは、特定のアプリケーションやデバイス ドライバまたは他のコードモジュールを OS デザインに含め、コードと OS デザインをひとつのソリューションとして保持する際に便利です。

このレッスンを終了すると、以下をマスターできます :

- サブプロジェクトを作成して設定ができる
- サブプロジェクトをビルドして使用できる

レッスン時間 (推定): 20 分

Windows Embedded サブプロジェクト概要

Windows Embedded CE 6.0 用 Platform Builder を使って OS デザインの一部のサブプロジェクトを作成できます。サブプロジェクトはモジュール式で再頒布可能のため、アプリケーション、ドライバ、その他のファイルを OS デザインに追加するときに BSP の一部として手動でビルドツリーに追加する必要がなく便利です。またテストアプリケーションおよび開発ツール用サブプロジェクトを作成してテストデバイス上でツールの作成と実行を簡単に行うことができます。

サブプロジェクトの種類

Windows Embedded CE は以下のサブプロジェクトをサポートします :

- アプリケーション　C　または　C++　言語でプログラムされたグラフィック ユーザ インターフェース (GUI) Win32 ィアプリケーション。

- コンソール アプリケーション　C または C++ 言語でプログラムされた GUI なしのアプリケーション。
- ダイナミック リンク ライブラリ (DLL)　ランタイムにロードして使用するドライバまたは他のコード ライブラリ。
- スタティック ライブラリ　ライブラリファイル (.lib) 型のコードモジュールで、他のサブプロジェクトにリンクしたり、OS デザインの SDK の一部としてエクスポートできます。
- TUX ダイナミック リンク ライブラリ　Microsoft Windows CE テストキット (CETK) 用 Windows Embedded CE カスタム テスト コンポーネント (第 4 章で説明)。

サブプロジェクトを作成し、OS デザインに追加する方法

簡単にサブプロジェクトを新規に作成し、既存のプロジェクトをサブプロジェクトとして OS デザインに追加することができます。ほとんどの場合、Windows Embedded CE サブプロジェクト ウィザード を使用して、これらの作業を行うことができます。ソリューション エクスプローラのサブプロジェクトのフォルダを右クリックし、[新しいサブプロジェクトの追加] または [既存のサブプロジェクトの追加] をクリックすると、ウィザードが起動します。ただし、各種のサブプロジェクトが果たす目的や CE サブプロジェクト ウィザードで作成したファイルおよび設定、ビルド プロセス、サブプロジェクトのカスタマイズ オプションなどの詳細を理解しておけば役に立ちます。

CE サブプロジェクト ウィザードは、必須の構成ファイルがすべて保存されている OS デザイン フォルダに、サブフォルダを 1 つ作成します。このフォルダには、次の必須ファイルを含みます :

- <名前>.pbpxml　サブプロジェクトに関するメタデータ情報を含む XML ベースのファイル。このファイルはサブプロジェクトをビルドするための .bib, .reg、SOURCES、および DIRS ファイルを参照します。
- < 名前 >.bib　ビルド処理中の makeimg で使用されるバイナリ イメージ ビルダ (BIB) ファイルで、バイナリ イメージに含めるファイルを指定します。
- < 名前 >.reg　最終ランタイム イメージに含める設定を使用したレジストリ ファイル。
- Sources　Windows Embedded CE のソース ファイル。Windows Embedded CE のビルド処理を制御するオプションを含むメイクファイルです。

- メイクファイル　Windows Embedded CE ビルド処理で SOURCES ファイルと共に使用されるファイル。

あとで使うためにサブプロジェクトのコピーを作成する場合は、OSDesigns フォルダ (%_WINCEROOT%\OSDesigns) を開いてから、OS デザイン用のソリューション ファイルを開きます。ソリューション フォルダには、通常 <OS デザイン名 >.sln ファイルおよび OS デザインに対応して名付けられたフォルダ 1 つがあります。. このフォルダには、OS デザインの定義ファイルである <OS デザイン名 >.pbxml といくつかのサブディレクトリがあります。これらのサブディレクトリの 1 つがサブプロジェクト フォルダになります (図 1-4)。このフォルダをバックアップしておくことをお勧めします。このフォルダは、ソリューション エクスプローラのサブプロジェクト コンテナを右クリックし [既存のサブプロジェクトの追加] をクリックすると、あとから任意の OS デザインに追加することができます。

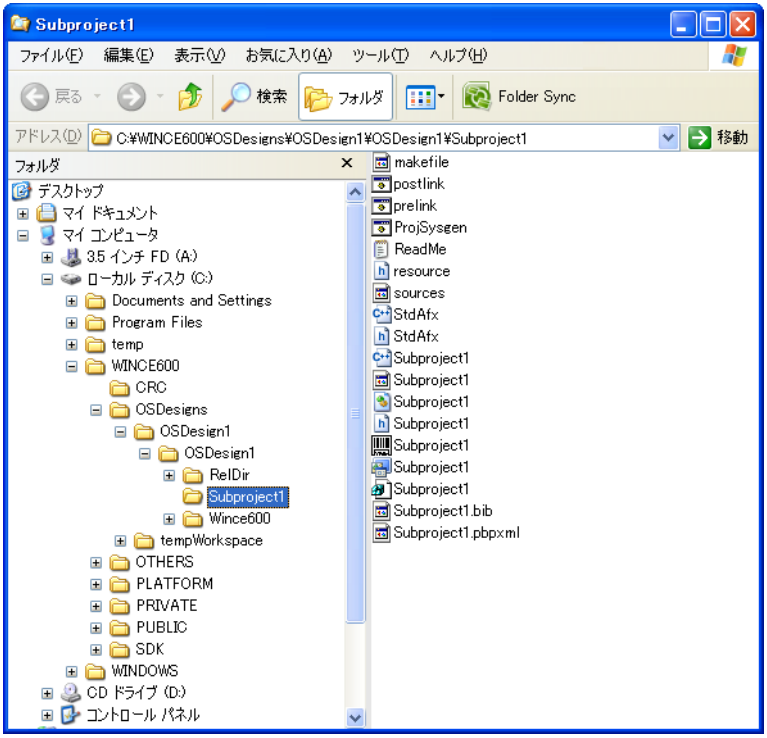


図 1-4　OS デザイン プロジェクトのサブプロジェクト フォルダ

Windows Embedded CE アプリケーションおよびダイナミック リンク ライブラリの作成方法

Windows Embedded CE アプリケーションまたは DLL を OS デザインに追加するには、CE サブプロジェクト ウィザードを使用して対応するサブプロジェクトを作成します。空のサブプロジェクトで最初から始めることもできますが、ベーシック コンソールまたは GUI アプリケーション テンプレートを使用し、必要に応じて後から独自のコードを追加する方が便利です。

スタティック ライブラリの作成

CE サブプロジェクト ウィザードにはスタティック ライブラリを作成するオプションもあり、あとから別のサブプロジェクトにリンクするか、SDK の一部として書き出すことができます。このオプションは複雑なサブプロジェクトに分割し、ハードウェアやファームウェア用のソリューションを開発するアプリケーション開発者により多くのオプションを提供するうえで有益な機能です。OS デザインのほかのサブプロジェクトがスタティック ライブラリに依存している場合、ライブラリを効果的に使用するためにサブプロジェクトのビルド順を調整する必要があるかもしれません。例えば、Windows Embedded CE アプリケーションがスタティック ライブラリを使用する場合、そのライブラリを最初にビルドしておけば、アプリケーションのビルド プロセスで最新のライブラリを使用できます。

ファイルや環境変数をランタイム イメージに追加するためのサブプロジェクトの作成

サブプロジェクトには、必ずしもソース コードが含まれているとは限りません。例えば、CE サブプロジェクト ウィザードを使用して、空のサブプロジェクトを作成して SOURCES ファイルを編集し、TARGETTYPE=NOTARGET を設定することで、バイナリ ターゲット ファイルを作成しないよう指定することができます。そのあとサブプロジェクトの .bib ファイルに対応する参照を追加し、ファイルをランタイム イメージに追加することができます。レジストリの設定をサブプロジェクトの .reg ファイルに追加し、サブプロジェクトの Projsysgen.bat ファイルを編集して SYSGEN 変数を追加することもできます。通常は .reg と .bib ファイル、および OS デザインのプロジェクト プロパティを直接変更したほうがより早く簡単にランタイム イメージに追加することができますが、サブプロジェクトを作成しておくと、今後複数の OS デザインのカスタマイズで再利用する予定がある場合、あとになって重宝することがあります。

サブプロジェクトの構成

Visual Studio はプロジェクトのプロパティを構成できるいくつかのオプションがあり、サブプロジェクトのビルド プロセスをカスタマイズすることができます。これらの設定を構成するには、この章の前半で説明した OS デザインのプロパティのページを表示します。[サブプロジェクト イメージの設定] でサブプロジェクトのプロパティを探します。現在の OS デザインで作成した、あるいは追加されたサブプロジェクトごとに、次のパラメータを構成することができます :

- ビルドから除く　このオプションを有効にすると、サブプロジェクトを OS デザインのビルド処理から排除します。つまり、ビルド エンジンは選択されたサブプロジェクトのソース ファイルを処理しません。
- イメージから除く　サブプロジェクトを変更した場合、ランタイム イメージの展開に時間がかかることがあります。サブプロジェクトを変更した場合、そのたびにターゲット プラットフォームから切断し、プロジェクトをリビルドして新しいイメージを作成し、ターゲット プラットフォームに再接続してから、最新のイメージをダウンロードしなければなりません。サブプロジェクトで作業をする際に時間と労力を節約するには、[イメージから除く] オプションでサブプロジェクトをランタイム イメージから排除しておきます。その場合、KITL、ActiveSync、またはその他の方法で、デバイスに転送し、デバイスのファイルを更新する方法を用意しておきます。
- 常にデバッグとしてビルドおよびリンクする　現在の OS デザイン ビルド処理がリリース設定を使用してる間に、デバッグ ビルド構成を使用してサブプロジェクトをビルドします。この方法を使用すると。オペレーティング システムがリリース設定を使用して実行している間、カーネル デバッガを使用して、サブプロジェクト コードをデバッグすることができます (このオプションをオンにしても、カーネル デバッガが自動的に有効になるわけではありません)。

ノート　ランタイム イメージから除く

サブプロジェクトからランタイム イメージを除く場合は、ターゲット デバイスにダウンロードした Nk.bin ファイルからサブプロジェクトのファイルを明示的に排除してください。その代わり Windows Embedded CE は、必要に応じて (KITL が有効であれば) KITL を使用して、リリース ディレクトリのサブプロジェクト ファイルに直接アクセスします。つまり、ランタイム イメージをいちいち再展開しなくても、ドライバまたはアプリケーションのサブプロジェクトのコードを変更することができます。リモート デバイスがコードが実行していないことを確認したら、コードをリビルドしてからもう一度実行します。

レッスン概要

Windows Embedded CE サブプロジェクトを使用して、アプリケーション、ドライバ、DLL、静的ライブラリを OS デザインに追加することができます。サブプロジェクトは、多数のアプリケーションとコンポーネントを含む複雑な Windows Embedded CE 展開プロジェクトを管理する際に有益な機能です。例えば、USB 周辺装置のカスタム シェル アプリケーションまたはデバイス ドライバをサブプロジェクトの形で OS デザインに含めておき、別の開発チームがこれらのコンポーネントを実装することができます。You can also use Windows Embedded CE サブプロジェクトを使用して、レジストリ設定、環境変数、または特定のファイルを、Core Connectivity (CoreCon) インターフェイスやテスト アプリケーションなどのさまざまな OS デザインに追加することもできます。サブプロジェクトを個別にバックアップしておき、既存のサブプロジェクトに追加して、のちの OS デザインに使用することもできます。

レッスン 3: コンポーネントの複製

Windows Embedded CE 6.0 R2 用 Platform Builder には、多彩な目的に応用できる再使用可能な Public ソース コード が用意されています。You can analyze and modify the source code for most of the components included in Windows Embedded CE に含まれている、シェルからシリアル ドライバのモデル デバイス ドライバ (MDD) 層 に及ぶほとんどのコンポーネントのソース コードは、分析および変更することができますが、Public ソース コードは直接変更しないでください。その代わり Public コードのコピーを作成し、それを機能用として自由に変更します。この方法なら、オリジナルの Windows Embedded CE 6.0 R2 コード ベースには影響を与えません。

このレッスンを終了すると、以下をマスターできます :

- 複製するコンポーネントを特定する。
- 既存のコンポーネントを複製する。

レッスン時間 (推定): 15 分

Public ツリーの編集とコンポーネントの複製

変更したいコードが %_WINCEROOT%\Public フォルダにあることがわかると、このコードを変更して別のフォルダに移動せずに、そのままビルドしたいと思うかもしれません。しかし Public ツリーを変更できない、いくつかの理由があります :

- Public ディレクトリをバックアップし、OS デザイン プロジェクトごとに、WINCE600\PUBLIC_Company1、WINCE600\PUBLIC_Company2、および WINCE600\PUBLIC_Backupなどのように個別のディレクトリを作成して、管理する必要がある。
- Windows Embedded CE 向けの更新プログラム、QFE (quick fix engineering) が提供する修正プログラム、および Service Pack が編集を上書きしてしまったり、編集した内容と互換性を持たない可能性がある。
- コードの再頒布が困難なうえ、エラーが発生しやすくなる。
- Public ディレクトリ ツリーのコードを変更した場合、オペレーティング システムのビルドに最長 3 時間かかります。Public フォルダ全体をリビルドせずに、特定のコードだけをリビルドできるくらい CE のビルドプロセスについて熟知しているのであれば、コンポーネントの複製についても同様です。

注意 Public コードの変更
Public フォルダ ツリーの内容は、絶対に変更しないでください。

コンポーネントの複製は面倒な作業に思えるかもしれませんが、長い目でみると開発のために費やす時間と労力を節約してくれます。

Public ソース コード

Platform Builder は Windows Embedded CE コンポーネントのいくつかに対して、インスタント コピーをサポートしています。これらのコンポーネントを複製するには、ソリューション エクスプローラの [カタログ項目ビュー] 内のカタログ項目を右クリックし、[カタログ項目の複製] を選択します。Platform Builder は OS デザインで選択したコンポーネントのサブプロジェクトをコードのコピーを使用して自動的に作成します。SYSGEN キャプチャ ツール などのほかの方法を使用する前に、複製したいカタログ コンポーネントがカタログ項目の複製オプションをサポートしているかどうか調べてください。サポートしている場合、クリックを 2 回するだけで複製を完了することができます (図 1-5)。

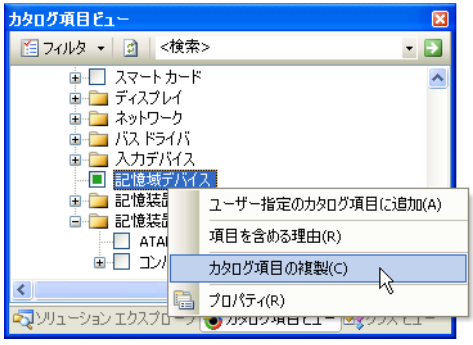


図 1-5 カタログ項目の複製

IDE で自動的にコンポーネントを複製できない場合は、手動で複製しなければなりません。しかし、Public ディレクトリ ツリーの .dll ファイルまたは .exe ファイルの Sources ファイルを調べると、このファイルはプラットフォームのディレクトリまたはサブプロジェクトのディレクトリにある Sources ファイルと違うことがわかります。これは、Public ディレクトリ ツリーのビルド処理が BSP のビルド処理と異なるためです。ビルド インストラクションはすべて、Sources ファイルと関連付けられたディレクトリと同じディレクトリにあるメイクファイルに定義されています。Public ディレクトリ ツリーは、必須のコンポーネントが相対的にリンクされる SYSGEN フェーズをサポートする必要があります。

Public ディレクトリ ツリーのコンポーネントを BSP コンポーネントまたはサブプロジェクトに変換するには、いくつかの手順を行う必要があります。詳しくは、Platform Builder for Microsoft Windows Embedded CE の付属ドキュメンテーションの「Using the Sysgen Capture Tool (Sysgen Capture Tool の使い方)」*http://msdn2.microsoft.com/en-us/library/aa924385.aspx* (英語) を参照してください。

基本的には、次の手順に従ってください :

1. Public コンポーネントのコードを、新しいディレクトリにコピーします。
2. 新しいディレクトリの Sources ファイルを編集します。RELEASETYPE=PLATFORM 行を追加するか、またはこの行が既に挿入されている場合は値を PLATFORM に変更し、ビルド エンジンが %_TARGETPLATROOT% フォルダに出力するように設定します。
3. WINCEOEM=1 を Sources ファイルに追加し、新しいディレクトリのコンポーネントをビルドします。ビルド エラーをすべて解消するために、さらに変更が必要な場合もあります。
4. Sysgen Capture tool を使用して、モジュラー Sources および Dirs ファイルを作成します。
5. Sysgen Capture Tool が作成したファイルの名前を変更し、このファイルとメイクファイルを一緒に使用して、新しく複製したモジュールをリビルドします。

必要なすべての変更を複製したコンポーネントに適用すると、他のコードと同じ要領で簡単い編集したり、再頒布することができます。

レッスン概要

Windows Embedded CE には、ほとんどの CE コンポーネントのソース コードを含む Public ディレクトリがありますが、この Public ディレクトリ ツリーのソース コードを直接変更しないようにします。その代わり、項目を自動または手動で複製します。複製による方法を使う理由を既に説明したように、Public ディレクトリ ツリーのソース コードをそのまま変更すると、問題を抱える原因になります。

レッスン 4: カタログ項目の管理

Windows Embedded CE の最も有益な機能の 1 つはカタログ システムです。カタログを使用すると、Windows Embedded CE ファームウェアを必要に応じてすばやく簡単にカスタマイズすることができます。各コンポーネントに対してカスタム カタログを作成すると、コンポーネントのインストールおよび構成が容易になります。これはアドホックとプロフェッショナルの Windows Embedded CE ソリューションを差別化する要因になります。アドホック ソリューションの場合、基本的なインストールに関するメモと必須の SYSGEN 変数を提供すれば十分かもしれませんが、プロフェッショナルなソフトウェアの場合には、カタログ項目と SYSGEN 変数と構成ための適切な値を提供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 :

- カタログのコンテンツをカスタマイズする。
- 新しいコンポーネント エントリを BSP カタログに追加する。

レッスン時間 (推定): 20 分

カタログ ファイル の概要

Windows Embedded CE カタログは、拡張マークアップ言語 (XML) を使用しており、ファイルには .pbcxml の拡張子が付けられます。カタログには多数の .pbcxml ファイルがあり、WINCEROOT ディレクトリにあります。Platform Builder はこれらのファイルを自動的に列挙し、ソリューション エクスプローラのカタログ項目ビューを生成します。

Platform Builder は次のディレクトリを解析し、カタログ項目を列挙します :

- Public カタログ ファイル %_WINCEROOT%\Public\<*any subdirectory*>\Catalog\
- BSP カタログ ファイル %_WINCEROOT%\Platform\<*any subdirectory*>\Catalog\
- サードパーティ カタログ ファイル %_WINCEROOT%\3rdParty\<*any subdirectory*>\Catalog\
- 汎用のシステムオンチップ (SOC) ファイル %_WINCEROOT%\Platform\Common\Src\soc\<*any subdirectory*>\Catalog\

ノート　サードパーティフォルダ

サードパーティフォルダ には、通常 OS デザインの一部として含めて配布することができる、スタンドアローン アプリケーションまたはソース アプリケーションがあります。サードパーティフォルダの .pbcxml ファイルを列挙することで、Platform Builder はこれらのコンポーネントのエントリをカタログ項目ビューに追加する方法を提供しています。

カタログのエントリの作成および編集

新しいカタログ項目を Windows Embedded CE カタログに追加するには、既存のカタログ ファイル (.pbcxml file) のコピーを作成してから、このファイルを Platform Builder と一緒に提供されているカタログ エディタを使用して編集します。Visual Studio の [ファイル] メニューにある [新規作成] をポイントして [ファイル] を選択して、新しいカタログ ファイルを作成することもできます。[新しいファイル] ダイアログ ボックスの Platform Builder for CE 6.0 R2 で、[Platform Builder カタログ ファイル] を選択してから [開く] をクリックします。

ノート　カタログ ファイルの編集

カタログ ファイルの編集は Platform Builder のカタログ エディタを使用します。メモ帳などのテキスト エディタの使用するための設定はできません。カタログ ファイルを Platform Builder 以外で開いたり編集すると、必要以上に時間がかかります。

カタログのエントリのプロパティ

それぞれのカタログのエントリには、Platform Builder で編集できるいくつかのプロパティがあります (図 1-6)。最も重要なプロパティには、次があります :

- 一意な ID　一意な ID 文字列。
- 名前　カタログ項目ビューに表示されるカタログ コンポーネントの名前。
- 説明　コンポーネントの説明で、ユーザがマウス ポインタをカタログ項目の上に数秒置くと表示されます。
- モジュール　このカタログ コンポーネントに付属するファイル一覧。
- SYSGEN　変数　カタログ項目の環境変数。カタログ　コンポーネントが SYSGEN 変数を設定する場合、ここで設定します。
- 追加変数　カタログ項目の追加の環境変数。このフィールドは　Sources、.bib,、および .reg ファイルで使用される環境変数を設定し、ビルド処理を制御することができるため、BSP のカタログ コンポーネントうちで最も重要な部分だと言えます。またこのフィールドを使用して、他のコンポーネントの依存関係を生成することもできます。

- プラットフォーム ディレクトリ　カタログ項目ファイルのある場所。新しい BSP の場合、このプロパティを BSP のディレクトリの名前に設定します。

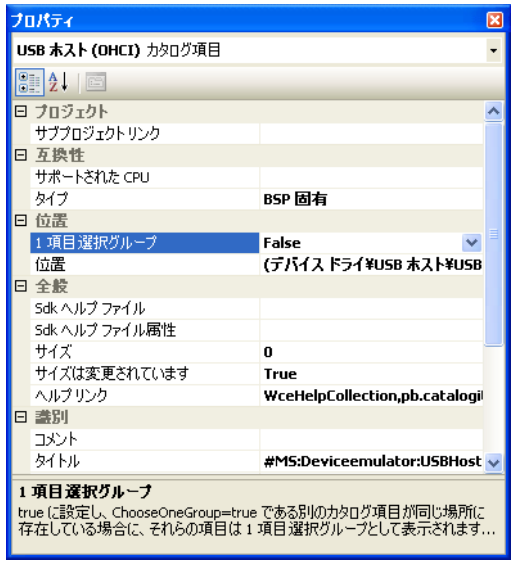


図 1-6　カタログ項目のプロパティ

ノート　一意の名前

各カタログ コンポーネントには、通常、ベンダとコンポーネントの名前で構成された一意の ID があります。カタログ項目の複製 機能を使用して、BSP を複製した場合、Platform Builder は複製したコンポーネントの一意の名前を自動的に作成します。カタログ ファイルを手動で編集する場合は、必ず一意の ID を使用してください。

新しいカタログ項目を OS デザインに追加する

新しいカタログ ファイルまたはカタログ項目を使用するには、対応する .pbcxml ファイルが サードパーティ ディレクトリまたは Platform ディレクトリのサブディレクトリにある Catalog という名前のサブフォルダにあることを確認してから、Visual Studio のカタログ項目ビューの [カタログツリーの更新] ボタンをクリックします。Platform Builder は サードパーティ ディレクトリおよび Platform ディレクトリをスキャンして、既存のカタログファイルを処理し、カタログを動的に再生成します。レッスン 1 で説明したように、カタログ項目ビューに一覧された新しいコンポーネントのチェック ボックスをオンにし、OS デザインに含めることができます。

カタログ項目を利用した BSP 開発

新しいカタログ項目を追加し、項目特有の環境変数の設定方法を学びました。このテクニックを使用して、コンポーネントを BSP に含め、C/C++ ビルド命令を設定し、ランタイム イメージのシステム レジストリの設定を変更することができます。ほかの開発者がこの BSP を使用する場合、OS デザイン プロジェクトのカタログ項目を選択すると、師弟した設定を無条件で使用します。カタログ コンポーネントを BSP に含めるには、BSP の Platform.bib ファイル を編集し、設定に基づいた条件付きステートメントを追加します。if-else ステートメントを使用して定義されているかどうかに関わらず、コンポーネントを含めることができます。.bib ファイルと .reg ファイルへの変更を反映させるために、Visual Studio の [ビルド] メニューの [詳細なビルド コマンド] の [現在の BSP およびサブプロジェクトのリビルド] コマンドを実行する必要があるかもしれません。[現在の BSP およびサブプロジェクトのリビルド] コマンドについては、第 2 章で詳しく説明します。

カタログ項目のプロパティで指定した環境変数に基づいて C/C++ 命令を設定するには、Sources ファイルで変数に基づいて条件付きステートメントを使用し、CDEFINES エントリを追加します。カタログ項目のプロパティに基づいて C/C++ ビルド命令を設定すると、将来における BSP のバイナリ バージョンの配布が困難になるため、通常は避けるようにしてください。

条件付きステートメントを使用して、システム レジストリのエントリを変更することもできます。新しいコンポーネントに関連した特定のレジストリ ファイルを含めたり、除いたりするには、.reg ファイルのみを編集します。

カタログからカタログ項目をエクスポートする

カタログ項目には直接複製できないものもあります。このようなコンポーネントを複製するには、新しいカタログ ファイルを作成しなければなりません。サードパーティ フォルダに新しいエントリを作成する場合、カタログ ファイルを新規に作成するか、または BSP の既存カタログ ファイルに新しいエントリを作成します。いずれの場合でも、すべての SYSGEN のオリジナルの値と追加変数が保持されているかどうかを確認するようにします。このレッスンで前述したように、カタログの各項目には一意の ID が必要ですので、ID を変更することを忘れないでください。

カタログ コンポーネントの依存関係

Windows Embedded CE 6.0 R2 用 Platform Builder のカタログは、コンポーネントの依存関係をサポートしています。コンポーネントが別のコンポーネントに依存していることを指定するには、カタログ項目のコンポーネントの SYSGEN または [追加変数] フィールド設定してから、その値を依存コンポーネントに追加環境変数の形で含めます。例えば、ディスプレイ用のディスプレイ ドライバとバックライト ドライバの両方のカタログ コンポーネントが BSP にある場合、ディスプレイ ドライバの [追加変数] フィールドを BSP_DISPLAY に設定し、バックライト ドライバの [追加変数] フィールドを BSP_BACKLIGHT に設定します。ディスプレイ ドライバをバックライト ドライバに依存させる場合、カタログ エディタで BSP_DISPLAY のカタログ エントリを編集し、BSP_BACKLIGHT を追加環境変数に追加します。ディスプレイ ドライバを OS デザインに含めると、Platform Builder は自動的にバックライト ドライバも含めます。カタログ項目ビューには、バックライト ドライバのチェック ボックスが表示され、緑の四角形はそのコンポーネントがディスプレイ ドライバに依存していることを示しています。

レッスン概要

Windows Embedded CE 6.0 R2 用 Platform Builder には、ファイル ベースのカタログ システムがあります。%_WINCEROOT% ディレクトリ ツリーの Platform ディレクトリまたは サードパーティ ディレクトリの個別のカタログ ファイルに、独自のカタログ項目を含めることができます。カタログ ファイルの形式は XML で、.pbcxml の拡張子が付いています。Visual Studio を起動し、ソリューション エクスプローラの [カタログ項目ビュー] を更新すると、Platform Builder は自動的に .pbcxml ファイルを列挙します。新しいカタログ項目を Windows Embedded CE カタログに追加するには、カタログ ファイルを新規に作成するか、または既存のカタログ項目のコピーを作成し、そのファイルの内容をカタログ エディタで編集します。すべての設定は Platform Builder で直接設定できるため、.pbcxml ファイルをメモ帳などのテキスト エディタで編集する必要はありません。条件付きの C/C++ ビルド命令、レジストリの変更、および依存関係の定義に対して、SYSGEN および追加環境変数を指定することができます。

レッスン 5: ソフトウェア開発キットの生成

ターゲット デバイス用のアプリケーションを作成するには、ソフトウェア開発キット (SDK) が必要です。SDK は自動的に OS デザインに対応するため、開発者は実際に使うことのできる機能だけを扱うことができます。SDK には OS デザインにある機能も含まれているため、アプリケーション開発者はサポートされていない API が原因で、ランタイム エラーを発生させるコードを誤って作成せずにすみます。

このレッスンを終了すると、以下をマスターできます :

- SDK の目的を特定する。
- SDK を生成する。
- ハード ドライブの SDK ファイルのローカライズする。
- SDK を使用する。

レッスン時間 (推定): 20 分

ソフトウェア開発キットの概要

OS デザインに対して有効なアプリケーションをコンパイルおよび作成するには、必須のヘッダー ファイルとリンクを含めて、開発プロジェクトの適切なライブラリにリンクする必要があります。OS デザインの SDK が、アプリケーション開発者に提供するカスタム コンポーネント用のヘッダー ファイルとライブラリを含む、すべての必須のヘッダー ファイルとライブラリを含んでいることを確認してください。Windows Embedded CE 6.0 R2 用 Platform Builder は、必須のヘッダー ファイルとライブラリをエクスポートすることで、OS デザイン用の SDK を作成することができます。

SDK の生成

カスタマイズした SDK の生成および配布は、通常 OS デザインの制作者の作業です。Platform Builder はこの目的のために SDK をエクスポートする機能を備えています。SDK エクスポート機能は、OS デザイン用にカスタマイズした SDK と SDK セットアップ ウィザードを含む .msi ファイルを作成します。

SDK の生成と構成

Platform Builder の SDK のエクスポート機能を使用して SDK を作成および構成するには、次の手順にしたがいます :

1. OS デザインを構成し、リリース構成で最低 1 回ビルドを行います。

2. ソリューション エクスプローラを表示し、SDK を右クリックして [新規追加] を選択して [SDK プロパティ ページ] ダイアログ ボックスを表示します。
3. [SDK プロパティ ページ] ダイアログ ボックスで、SDK の [全般] プロパティを構成し、[MSI フォルダ パス]、[MSI ファイル名]、および [ロケール] を定義します (図 1-7)。カスタム設定の名前もしていすることができます。
4. 追加のファイルを含めるには、[SDK プロパティのページ] ダイアログ ボックスの [追加フォルダ] ノードを選択します。
5. [OK] をクリックします。

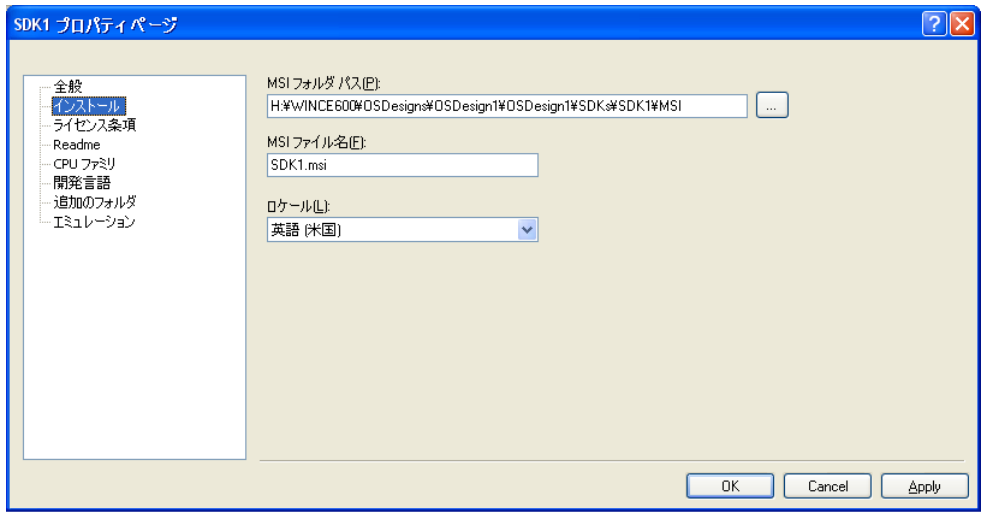


図 1-7 [SDK プロパティ ページ] ダイアログ ボックス

SDK への新しいファイルの追加

[SDK プロパティ ページ] の [追加フォルダ] オプションを使用するか、または OS デザインの SDK ディレクトリ (通常は \OSDesigns\< ソリューション名前 >\<OS デザイン名 >\WinCE600\< プラットフォーム名 >\SDK) にコピーし、ファイルを手動で SDK に追加することができます。ビルド エンジンがビルドのたびに最新バージョンのファイルを SDK にコピーするよう、.bat ファイルと Sources ファイルを使用して処理を自動化させることもできます。

ファイルは、必ず次の SDK サブディレクトリにコピーします :

- Inc　SDK のヘッダー ファイルを含みます。.
- Lib\<*Processor Type*>\<*Build Type*>　SDK のライブラリを含みます。

SDK のインストール

SDK ビルド プロセスを完了後、OS デザイン フォルダの SDK サブディレクトリの .msi ファイル (通常は %_WINCEROOT%\OSDesigns\< ソリューション名 >\<OS デザイン名 >\SDKs\SDK1\MSI\<SDK 名 >.msi) を探すことができます。この MSI は Platform Builder および該当する場合はサードパーティ コンポーネントのライセンス契約に従い、自由に再頒布することができます。

この MSI パッケージは Visual Studio 2005 がインストールされたどのコンピュータ にでもインストールすることができ、ターゲット デバイスの Windows Embedded CE アプリケーションの開発に使用できます。SDK がインストールされたコンピュータで %PROGRAMFILES%\Windows Embedded CE Tools\WCE600 のファイルを探します。

レッスン概要

Windows Embedded CE 6.0 R2 は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。そのため、ターゲット デバイスで動作するアプリケーションを開発するアプリケーション開発者は OS デザインに応じてカスタマイズされた SKD が必要になります。ファイルやライブラリが足りないためにビルトや実行時に発生する問題を回避するため、カスタム SDK には Windows Embedded CE コンポーネントのみでなく、OS デザインに含まれるカスタム コンポーネントのヘッダー とライブラリも含める必要があります。Platform Builder の SDK をエクスポートする機能を利用して、SDK の生成と MSI パッケージを作成し、SDK セットアップ ウィザードで、アプリケーション開発コンピュータで SDK を展開することができます。

ラボ 1: OS デザインの作成、構成、およびビルド

このラボでは OS デザインを作成したあと、カタログからコンポーネントを追加して、デザインをカスタマイズします。このラボの内容は、『Microsoft Windows Embedded CE 6.0 R2 受験対策キット』の他の章のレッスンで必要なため、ここに示されている手順は必ずすべて行ってください。

ノート　詳しい手順
このラボで紹介する手順を正しく習得するために、この本の付属教材の「ラボ 1 の詳しい手順」を参照してください。

OS デザインの作成

1. Windows Embedded CE 6.0 R2 用 Platform Builder をプラグインした Visual Studio 2005 で [ファイル] メニューの [新規作成] をポイントして、[プロジェクト] をクリックすると、新しい OS デザイン プロジェクトが作成されます。
2. 既定の OS デザイン名 (OSDesign1) を使用します。
3. Visual Studio の Windows Embedded CE 6.0 OS デザイン ウィザードが開始します。
4. BSP リストの [デバイス エミュレータ :ARMV4I] チェック ボックスをオンにして、[次へ] をクリックします。
5. [使用可能なデザイン テンプレート] から [PDA デバイス] を選択します。[使用可能なデザイン テンプレート] から [モバイル ハンドヘルド] を選択します。.
6. ウィザードの次のページで .NET Compact Framework 2.0 と ActiveSync のチェックをオフにします (図 1-8)。[インターネット ブラウザ] と [Quarter VGA リソース - 縦モード] のチェックはオンのままにしておきます。
7. Networking Communications (ネットワーク接続) ウィザード ページで [TCP/IPv6 サポート] と [パーソナル エリア ネットワーク (PAN)] のチェックをオフにし、Bluetooth と Infrared Data Association (IrDA) のサポートを排除します。[ローカル エリア ネットワーク (LAN)] はオンにしたままにしておきます。
8. [完了] をクリックして、Windows Embedded CE 6.0 OS デザイン ウィザードを閉じると、Visual Studio OS デザイン プロジェクトが開きます。[ソ

リューション エクスプローラ] タブが有効になり、[ソリューション コンテナ] に新しい OS デザイン プロジェクトが表示されます。

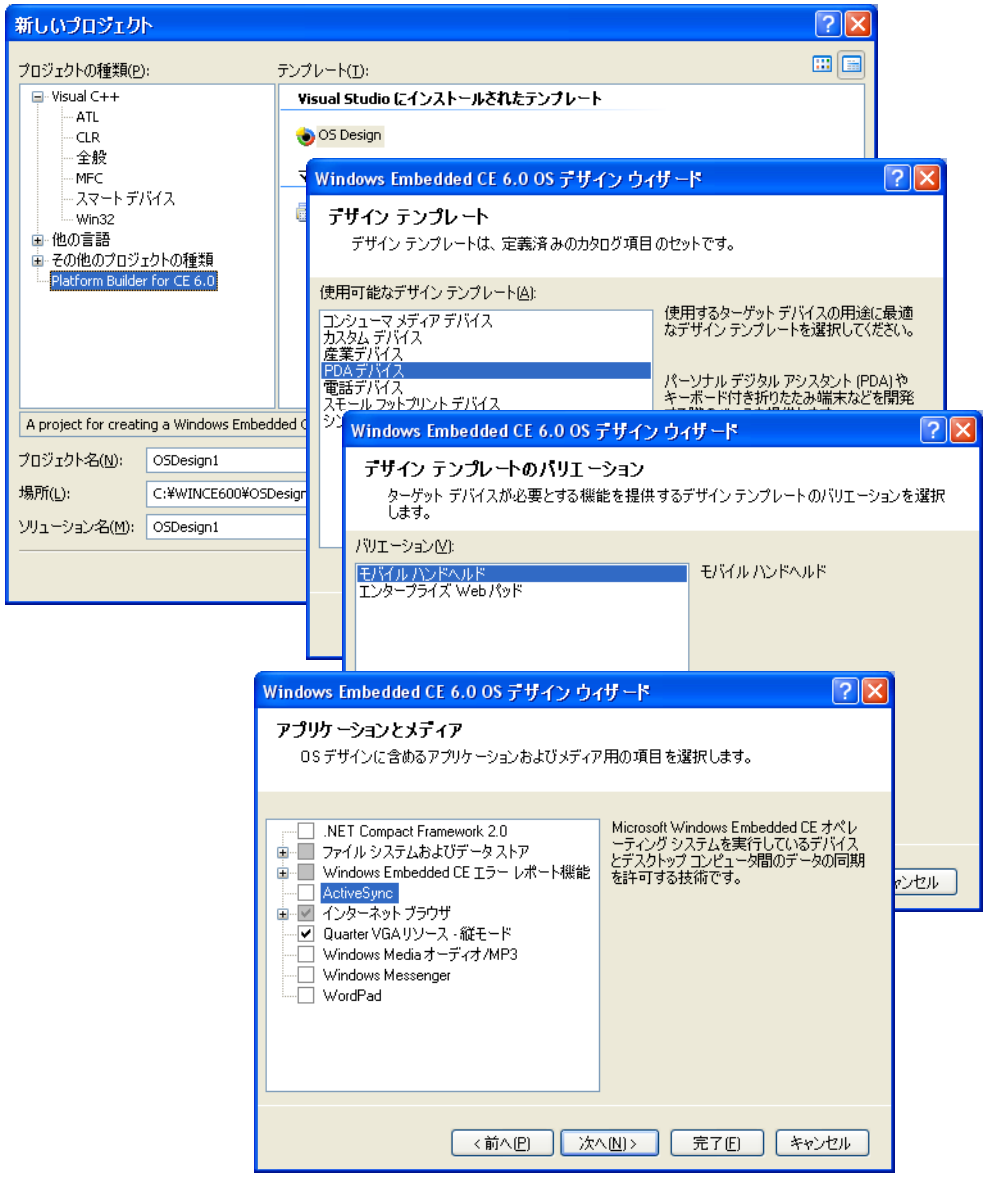


図 1-8　PDA デバイスの OS デザインを作成する

ノート　このあとの OS デザイン変更
OS デザイン ウィザードで作成される OS デザインの初期構成は、ウィザードの終了後に変更することができます。

- **OS カタログを調べる**

1. Visual Studio のソリューション エクスプローラで、[カタログ項目ビュー] をクリックします。
2. 個々のコンテナ　ノードを展開し、カタログのオンになっているチェック ボックスとアイコンを調べます。緑のチェック マークが表示された項目は、OS デザインの一部として特に追加されたことを示しています。緑の四角が表示された項目は、依存関係によって OS デザインに含まれていることを示しています。何も表示されていない項目はこの OS デザインには含まれていませんが、いつでも追加できることを示しています。
3. 緑の四角が表示されたカタログ項目を探します。
4. このカタログ項目を右クリックして、[項目を含める理由] を選択します。選択されたカタログ項目を OS デザインに含める理由になっているカタログ項目を示した [依存関係にあるカタログ項目を削除] ダイアログ ボックスが表示されます (図 1-9)。
5. カタログの Core OS | CEBASE | Applications ミ End User | Active Sync node を展開します。
6. ActiveSync システム CPL の項目のいずれかを右クリックして、[ソリューション ビューに表示] を選択します。ActiveSync コンポーネントを含むサブプロジェクトを示したソリューション エクスプローラのタブが表示されます。これは Windows Embedded CE 6.0 に用意されているソース コードをナビゲートする最適な方法です。

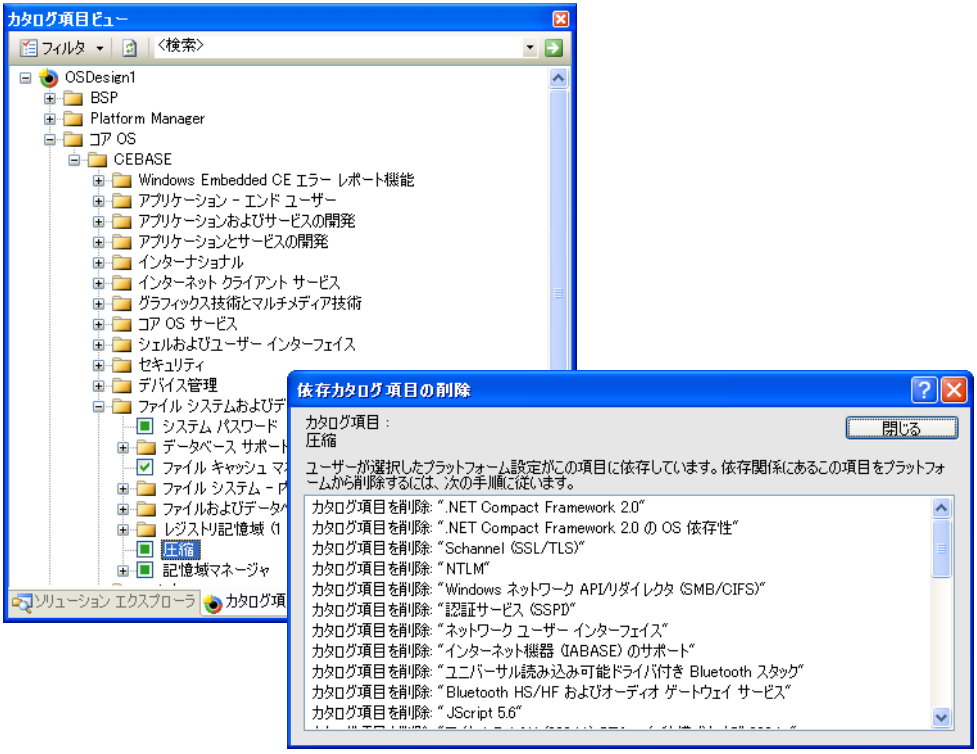


図 1-9　カタログ項目を依存した項目として含める理由

- **Internet Explorer 6.0 のサンプル ブラウザ カタログ項目のサポートを追加**

1. [カタログ項目ビュー] タブを選択して、OS デザイン カタログを表示します。フィルタ オプションが [カタログ内のすべてのカタログ項目] に設定されているかどうか確認します。
2. カタログ項目ビューの [フィルタ] ボタンの右側にある [検索] ボックスに、「Internet Explorer 6.0 のサンプル」と入力し、Enter キーを押すか、緑の矢印をクリックします。
3. 検索結果から Internet Explorer 6.0 の サンプル ブラウザ カタログ項目を探します。該当するチェック ボックスをオンにして、このカタログ項目を OS デザインに含めます (図 1-10)。

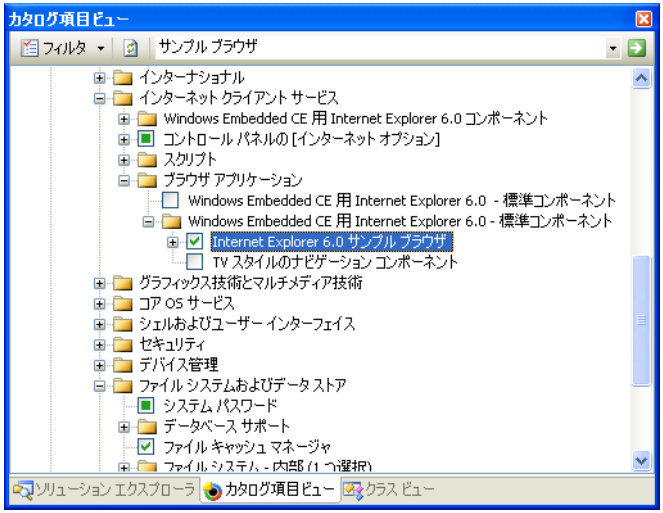


図 1-10 Internet Explorer 6.0 のサンプル ブラウザ カタログ項目を OS デザインに含める

- **OS デザインにおける管理コード開発のサポート**

1. [検索] ボックスに、「ipconfig」と入力して Enter キーを押します。
2. 検索結果からネットワーク ユーティリティ (IpConfig、Ping、Route) カタログ項目を探します。
3. ネットワーク ユーティリティ (IpConfig、Ping、Route) カタログ項目のチェック ボックスをオンにして、OS デザインに追加します。
4. [検索] ボックスに、「wceload 」と入力して Enter キーを押します。
5. 検索結果から CAB ファイル インストーラ/アンインストーラ カタログ項目を探します。このカタログ項目の SYSGEN 変数は wceload に設定されているため、検索結果に表示されるはずです。
6. CAB ファイル インストーラ / アンインストーラ カタログ項目を OS デザインに追加します。
7. 同じ要領で検索機能を使用し、.NET Compact Framework 1.0 への OS 依存関係を探します。.NET Compact Framework 1.0 への OS 依存関係カタログ項目が OS デザインに含まれているかどうか確認します (図 1-11)。

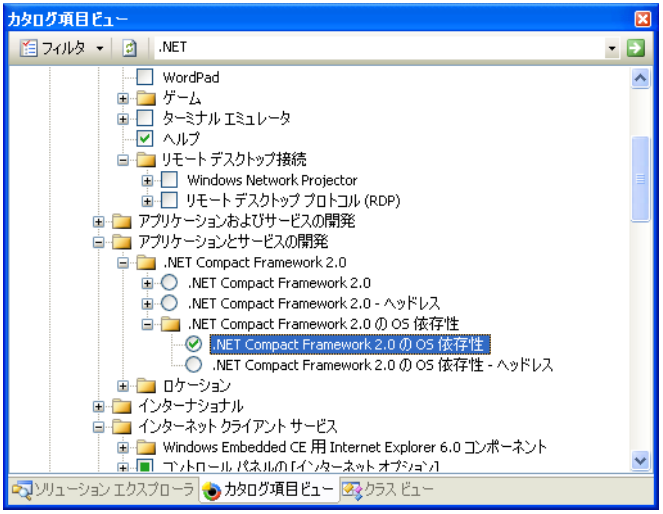


図 1-11　.NET Compact Framework 1.0 への OS 依存関係カタログ項目を OS デザインに含める

ノート　ヘッドレス .NET Compact Framework

このカテゴリには、2 つのコンポーネントがあります。ヘッドレス バージョンは、ディスプレイのないデバイス用ですので、記述にヘッドレス修飾子がないコンポーネントを必ず選択してください。

第 1 章の復習

Windows ィ Embedded CE 6.0 R2 をターゲットデバイスに展開するときは、必要なオペレーティング システム (OS) コンポーネント、機能、ドライバ、構成の設定を含む OS デザインを作成しなければなりません。Platform Builder を使用して、展開のために対応するランタイム イメージをビルドします。要件に応じてカスタマイズした OS デザインを作成する場合、次の最重要な作業を行う必要があります :

- Visual Studio の OS デザイン ウィザードを使用して、OS デザイン プロジェクトを作成する。
- 依存関係を使用して、手動で OS のコンポーネントを追加または削除する。
- カタログ エディタで環境変数と SYSGEN 変数を設定する。
- OS デザインをローカライズする地域の設定を構成する。
- カタログのコンポーネントを [カタログ項目の複製] をクリックして自動的に複製するか、または Sysgen Capture tool を使用して手動で複製する。
- OS デザインのカスタム SDK をエクスポートし、ターゲット デバイスのアプリケーションの依存関係を容易にする。

重要な用語

次の用語の意味を。巻末の用語集を調べ、回答の正誤を確認してみてください。

- OS デザイン
- コンポーネント
- SYSGEN 変数
- 環境変数
- ソフトウェア開発キット

推奨する練習内容

この章で扱っている試験範囲の内容を十分に理解するには、次の作業を完了してください :

カスタム OS デザインの作成

OS デザイン ウィザードを使用して、デバイス エミュレータ : ARMV4I BSP およびカスタム デバイス デザイン テンプレートに基づいて OS デザインを作成します。OS デザインが作成されたら、次の作業を行います :

- .Net Compact Framework 2.0 の追加　カタログ項目ビューの検索機能を使用して、このカタログ項目を追加します。
- ランタイム イメージのローカライズ　OS デザインのプロパティ ページを表示し、OS デザインをフランス語にローカライズします。

SDK の生成とテスト

ラボ 1 で生成した OS デザインを基に、次の作業を行います :

- バイナリ イメージのビルドと生成　リリース構成で生成した OS デザインのバイナリ イメージをビルドおよび生成します。
- SDK の作成とインストール　ビルド処理が正常に完了したかどうか確認してから、新しい SDK を作成してビルドし、アプリケーション開発用コンピュータにインストールします。
- SDK の使用　Visual Studio の別のインスタンスを使用して、Win32 Smart Device アプリケーションを作成します。カスタム SDK をプロジェクトの SDK 参照として使用し、アプリケーションをビルドします。

第 2 章

ランタイム イメージのビルドおよび展開

Microsoft ィ Windows ィ Embedded CE 6.0 R2 ビルド プロセスは非常に複雑です。このプロセスにはいくつかの段階が含まれており、多数のツールを使用して、Windows Embedded CE ビルド環境の初期化、ソース コードのコンパイル、モジュールおよびファイルの共通リリース ディレクトリへのコピー、およびランタイム イメージの作成を行います。[システム生成] ツール (Sysgen.bat) およびバイナリ イメージ作成ツール (Makeimg.exe) バッチ ファイルおよびビルド ツールによって、このプロセスを自動化します。これらのツールをコマンド プロンプトで直接実行するか、Windows Embedded CE 6.0 R2 の Microsoft Platform Builder でビルド プロセスを開始することができます。Platform Builder 統合開発環境 (IDE) は、同じプロセスおよびツールに依存しています。どちらの場合でも、ランタイム イメージを効率的に作成し、ビルド エラーのトラブルシューティングを行い、ボード サポート パッケージ (BSP) およびサブプロジェクトをランタイム イメージの一部としてターゲット デバイスに展開するには、ビルド プロセスおよび生成されたランタイム イメージを展開するのに必要な手順に関する深い理解が必要です。

本章の試験範囲：

- ランタイム イメージをビルドする
- ビルド結果およびビルド ファイルを分析する
- ランタイム イメージをターゲット デバイスに展開する

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。

- カタログ アイテムや環境変数や SYSGEN 変数の構成を含む、オペレーティング システム (OS) デザインの側面に関する理解 (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」)。
- ソースコード コンパイルおよびリンクを含む、Windows Embedded CE ソフトウェア開発に関する基本的な知識。
- 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：ランタイム イメージのビルド

Windows Embedded CE ビルド プロセスは、ランタイム イメージ開発サイクルの最後の段階です。OS デザインで定義された設定に基づき、Platform Builder はすべてのコンポーネント (サブプロジェクトと BSP を含む) をコンパイルしてから、ターゲット デバイスにダウンロード可能なランタイム イメージを作成します。ビルド プロセスには、いくつかのビルド フェーズが関係しており、バッチ ファイルによって自動化されます。ビルド オプションを正しく構成し、ランタイム イメージを効率的に作成し、ビルドに関連する問題を解決するには、これらのフェーズとビルド ツールを理解している必要があります。

このレッスンを終了すると、以下をマスターできます：

- ビルド プロセスを理解する。
- ビルドの問題を分析および修正する。
- ランタイム イメージをターゲット ハードウェアに展開する。

レッスン時間 (推定)：40 分

ビルド プロセスの概要

Windows Embedded CE ビルド プロセスには、図 2-1 に示すように、4 段階の主なフェーズが含まれています。ただし、各フェーズの目的や使用するツールを理解している場合は、これらを独立して実行することも可能です。ビルド ツールを選択的に実行することで、意図した方法で個別のビルド手順を実行することができ、ビルド時間を節約し、効率を大幅に向上することができます。

ビルド プロセスには、次の主なフェーズが含まれます。

- コンパイル フェーズ　コンパイラおよびリンカは、選択されたロケールに従って、ソース コードとリソース ファイルを使用して実行可能 (.exe) ファイル、静的 (.lib) ライブラリ、ダイナミック リンク ライブラリ (.dll) ファイル、およびバイナリ リソース (.res) ファイルを生成します。例えば、ビルド システムは、このフェーズ中に、プライベートおよびパブリック フォルダのソース コードを lib ファイルにコンパイルします。このプロセスを完了するには数時間かかることがありますが、バイナリが Microsoft によって提供されているため、幸いにもこれらのコンポーネントをリビルドする必要はほとんどありません。どの場合でも、プライベートおよびパブリック フォルダのソース コードを修正すべきではありません。

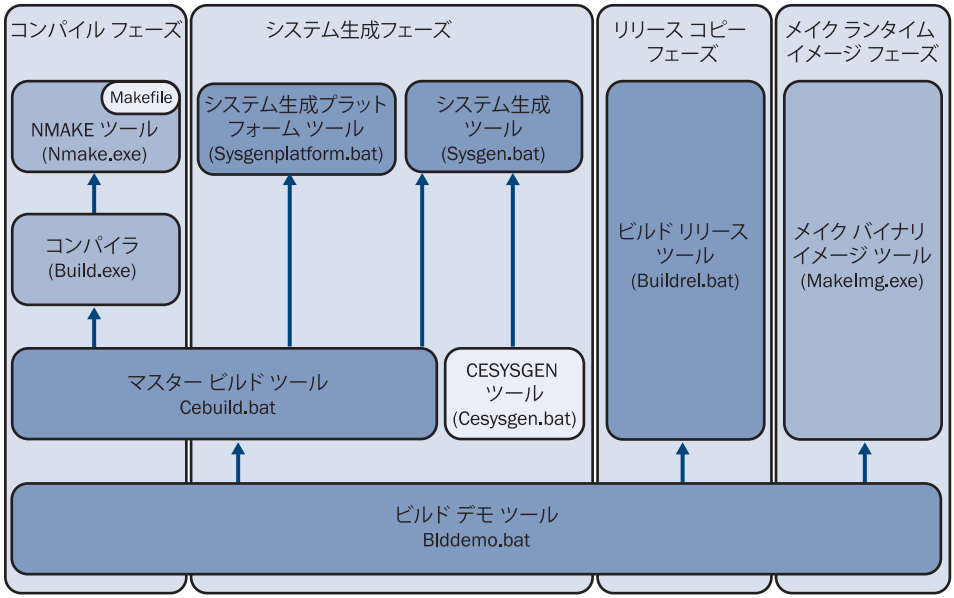


図 2-1　ビルド フェースおよびビルド ツール

- システム生成フェーズ　ビルド プロセスは、カタログ アイテムおよび OS デザインに含まれる従属ツリーに基づいて SYSGEN 変数を設定またはクリアし、ヘッダー ファイルにフィルタを掛け、OS デザインで定義されたソフトウェア開発キット (SDK) のインポート ライブラリを作成し、OS デザインのラインタイム イメージ構成ファイルのセットを作成し、およびプラットフォーム ディレクトリのソース ファイルに基づいて BSP をビルドします。
- ビルド フェーズ　ビルド システムは、システム生成フェーズ中に生成されたファイルを使用して、ボード サポート パッケージおよびアプリケーションのソース ファイルを処理します。この段階で、ハードウェア関連のドライバおよび OEM アダプテーション層 (OAL) がビルドされます。ビルド フェーズ中のプロセスは Sysgen フェーズで自動的に実行されますが、BSP およびサブプロジェクトのみ修正する場合、[システム生成] ツールを再度実行することなく BSP とサブプロジェクトをリビルドできることを理解しておくのは重要です。
- リリース コピー フェーズ　ビルド システムは、ランタイム イメージを作成するのに必要なすべてのファイルを OS デザインのリリース ディレクトリにコピーします。これには、コンパイルおよびシステム生成フェーズ中に作成された .lib、.dll、および .exe ファイル、およびバイナリ イメージ ビ

ルダ (.bib) およびレジストリ (.reg) ファイルが含まれます。ビルド システムでは、ヘッダーおよびライブラリが最新の場合、このフェーズをスキップすることがあります。

- ランタイム イメージの作成フェーズ　ビルド システムは、プロジェクト固有ファイル (Project.bib、Project.dat、Project.db、および Project.reg) をリリース ディレクトリにコピーし、リリース ディレクトリのすべてのファイルをランタイム イメージに結集させます。.reg および .bib ファイルで指定された環境変数に基づいたディレクティブは、ビルド システムが最終ランタイム イメージに含めるカタログ アイテムをコントロールします。通常、ランタイム イメージは Nk.bin という名前で、ターゲット デバイスにダウンロードして実行することが可能です。

Visual Studio でのランタイム イメージのビルド

開発ワークステーション上での Windows Embedded CE 6.0 R2 のインストール中に、Platform Builder は Visual Studio 2005 と統合し [ビルド] メニューを拡張することで、Visual Studio IDE でビルド プロセスを直接コントロールすることができます。図 2ミ2 に、[ソリューション エクスプローラ] で OS デザイン ノードを選択するときに、[ビルド] メニューで使用可能な Platform Builder を表示します。

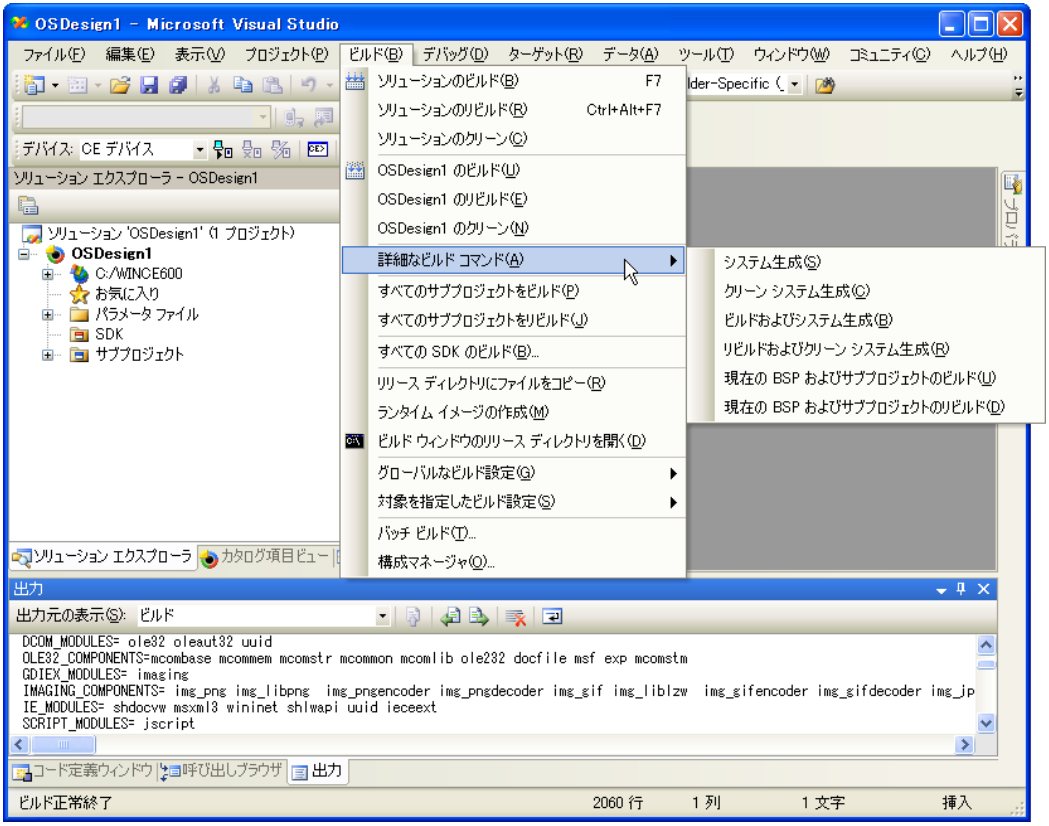


図 2-2　Visual Studio 2005 での Windows Embedded CE ビルド コマンド

[ビルド] メニューで Platform Builder コマンドを使用して、選択的なビルド ステップを実行したり、複数のビルド フェーズに及ぶ一連のビルド ステップを実行したりすることが可能です。例えば、[リリース ディレクトリにファイルをコピー] コマンドを使用して、ヘッダー ファイルおよびライブラリが変更されていない場合でも、ビルド システムが更新された .bib および .reg ファイルをリリース ディレクトリにコピーできるようにします。このコマンドを使用しないと、ビルド システムは、リリース コピー フェーズをスキップしてしまい、.bib ファイルや .reg ファイルの変更がランタイム イメージに適用されません。

表 2ミ1 は、Windows Embedded CE ビルド コマンドの目的を要約しています。

表 2ミ1 Windows Embedded CE ビルドおよびリビルド コマンド

メニュー オプション		説明
ソリューションのビルド		[詳細なビルド コマンド] サブメニューの [システム生成] コマンドと同等です。
ソリューションのリビルド		[詳細なビルド コマンド] サブメニューの [クリーン システム生成] コマンドと同等です。
ソリューションのクリーン		すべての中間ファイルを削除することで、リリース ディレクトリをクリーンします。
<OS デザイン名 > のビルド		複数の OS デザインを含むソリューションに役立ちます。単一 OS デザインのソリューションでは、これらのオプションは、[ソリューションのビルド]、[ソリューションのリビルド]、[ソリューションのクリーン] コマンドに相当します。
<OS デザイン名 > のリビルド		
<OS デザイン名 > のクリーン		
詳細なビルド コマンド	システム生成	[システム生成] ツールを実行し、パブリックおよびプライベート フォルダの .lib ファイルと関連付けて、ランタイム イメージ用ファイルを作成します。ファイルは、OS デザインの WinCE フォルダに保持されます。[グローバルなビルド設定] に従って、ビルド プロセスは、自動的に [リリース コピー]、次いで [ランタイム イメージの作成] フェーズに進むことができます。
	クリーン システム生成	前回のビルドで作成された中間ファイルを、[システム生成] ツールの実行前に消去します。前回の [システム生成] セッション後にファイルやカタログ アイテムを追加または削除した場合、このオプションを使用してビルド エラーの危険を低減します。
	ビルドおよびシステム生成	パブリックおよびプライベート フォルダのコンテンツ全体をコンパイルし、OS デザインの設定を使用してそれらのファイルを関連付けます。このプロセスには数時間かかり、パブリック フォルダのコンテンツを修正した場合のみ必要です。Windows Embedded CE コード ベースを変更しない限り (推奨しません)、このオプションを使用すべきではありません。

表 2 ミ 1 Windows Embedded CE ビルドおよびリビルド コマンド

メニュー オプション	説明
リビルドおよびシステム生成	パブリックおよびプライベート フォルダにある、前回のビルド時に作成された中間ファイルを消去してから、[ビルドおよびシステム生成] ステップを実行します。このオプションを使用すべきではありません。
現在の BSP およびサブプロジェクトのビルド	OS デザインの現在の BSP およびサブプロジェクト用ディレクトリにあるファイルをビルドし、[システム生成] ツールを実行します。このオプションは、現在の OS デザインによって使用されている以外の他の BSP もビルドすることに注意が必要です。そのため、BSP が互いに互換性があるか確認するか、使用していない BSP を削除する必要があります。
現在の BSP およびサブプロジェクトのリビルド	前回のビルドに作成された中間ファイルを消去し、[現在の BSP およびサブプロジェクトのビルド] ステップを実行します。
すべてのサブプロジェクトをビルド	すべてのサブプロジェクトをコンパイルおよびリンクしますが、最新のファイルはすべてスキップします。
すべてのサブプロジェクトをリビルド	すべてのサブプロジェクトをクリーン、コンパイル、およびリンクします。
すべての SDK のビルド	プロジェクトですべての SDK をビルドし、該当する Microsoft インストーラ (MSI) パッケージを作成します。通常、MSI パッケージのデバッグ バージョンを作成する必要はないため、このオプションは [リリース ビルド] 構成でのみ使用します。
リリース ディレクトリにファイルをコピー	BSP 用に生成されたファイルおよびコンポーネントを、コンパイルやシステム生成フェーズ中にリリース ディレクトリにコピーして、これらのファイルがランタイム イメージに含まれるようにします。
ランタイム イメージの作成	リリース ディレクトリのすべてのファイルを使用して、ランタイム イメージを作成します。このステップに従うことで、ランタイム イメージをターゲット デバイスにダウンロードできます。

表 2ミ1 Windows Embedded CE ビルドおよびリビルド コマンド

メニュー オプション		説明
ビルド ウィンドウのリリース ディレクトリを開く		[コマンド プロンプト] ウィンドウを開き、リリース ディレクリに変更し、すべての必要な環境変数を設定してバッチ ファイルの実行とツールの手動ビルドを行います。これを使用して、コマンド プロンプトでビルド ステップを実行します。標準 [コマンド プロンプト] ウィンドウは、ツールのビルドを成功させるために開発環境を初期化することはしません。
グローバルなビルド設定	ビルド後にリリース ディレクトリにファイルをコピー	すべてのコマンドで [リリース コピー] フェーズに自動的に進む機能を有効または無効にします。
	ビルド後のランタイム イメージの作成	ビルド操作後に、[ラインタイム イメージの作成] フェーズに自動的に進む機能を有効または無効にします。
対象を指定したビルド設定	ビルド後のランタイム イメージの作成	[ランタイム イメージの作成] フェーズを有効または無効にします。
バッチ ビルド		複数のビルドを順番に実行できるようにします。
構成マネージャ		ビルド構成を追加または削除できるようにします。

[詳細なビルド コマンド] サブメニューは、日常の作業で役に立つ、複数の Platform Builder の特定のビルド コマンドへのアクセスを提供します。例えば、OS デザインにカタログ コンポーネントを追加または削除してランタイム イメージのバイナリ バージョンを作成するとき、[システム生成] や [クリーン システム生成] コマンドを実行する必要があります。この規則の例外は、ThirdParty フォルダのコンポーネントなど、SYSGEN 変数を修正しないコンポーネントの場合です。これらのアイテムを選択または選択解除するときは、[システム生成] や [クリーン システム生成] を実行する必要はありません。[システム生成] フェーズに従って、Platform Builder は [現在の BSP およびサブプロジェクトのビルド] コマンドの実行と同様に、ビルド プロセスを続行します。

プラットフォーム ディレクトリまたは OS デザインの他のプロジェクトにあるソース コードをコンパイルおよびリンクして、コードを Platform\<BSP 名 >\Target および Platform\<BSP 名 >\Lib にあるターゲット ディレクトリに置きたい場合は、Visual Studio で [現在の BSP およびサブプロジェクトのビルド] または [現在の BSP およびサブプロジェクトのビルド] コマンドを選択できます。例えば、プラットフォーム ディレクトリのソース コードを修正したい場合は、これが必要です。[ビルド後にリリース ディレクトリにファイルをコピー] および [ビルド後にランタイム イメージを作成] オプションの設定に従って、Platform Builder はファイルをリリース ディレクトリにコピーし、ランタイム イメージを作成します。メニューを介して、またはコマンド プロンプトで Buildrel.exe および Makeimg.exe を実行して、これらのステップを個別に実行することもできます。

注意　[クリーン システム生成] は複数のビルド構成に影響する

[クリーン システム生成] コマンドをあるビルド構成で実行すると、後で、他のビルド構成の [システム生成] を実行する必要もあります。[クリーン システム生成] コマンドは、現在のビルド構成のファイルだけでなく、他のビルド構成用に生成されたすべてのファイルも削除することに留意してください。

コマンド ラインからランタイム イメージをビルド

Visual Studio 2005 の CE6 R2 プラグイン用 Platform Builder は、バッチ ファイルおよびビルド ツールへの便利なアクセスを提供しますが、これらのバッチ ファイルを実行して、コマンド プロンプトで直接ツールをビルドすることもできます。Platform Editor を使用する Visual Studio の各ビルド コマンドは、表 2 ミ 2 に列挙された特定のビルド コマンドと対応しています。ただし、Visual Studio の [ビルド ウィンドウを開く] コマンドを使用して、この目的で [コマンド プロンプト] ウィンドウを開くことに留意してください。標準コマンド プロンプトは、開発環境を初期化しません。ビルド プロセスは、必要な環境変数なしでは失敗します。

表 2 ミ 2　ビルド コマンドと対応するコマンド ライン

ビルド コマンド	対応するコマンド ライン
ビルド	blddemo -q
リビルド	blddemo clean -q
システム生成	blddemo -q

表 2-2 ビルド コマンドと対応するコマンド ライン

ビルド コマンド	対応するコマンド ライン
クリーン システム生成	blddemo clean -q
ビルドおよびシステム生成 *	Blddemo
リビルドおよびクリーン システム生成 *	blddemo clean cleanplat -c
現在の BSP およびサブプロジェクトのビルド	blddemo -qbsp
現在の BSP およびサブプロジェクトのリビルド	blddemo -c -qbsp

* 推奨せず

Windows Embedded CE ランタイム イメージ コンテンツ

図 2-3 に示すように、ランタイム イメージは、OS デザインの一部として、ターゲット デバイスで展開および実行するすべてのアイテムおよびコンポーネント (カーネル コンポーネント、アプリケーション、および構成ファイル) を含んでいます。開発者にとって最も重要な構成ファイルは、バイナリ イメージ ビルダ (.bib) ファイル、レジストリ (.reg)、データベース (.db)、およびファイル システム (.dat) ファイルです。これらのファイルは、メモリ レイアウトを決定し、Platform Builder がファイル システムおよびシステム レジストリを初期化する方法を指定します。これらのファイルの使用方法を理解しておくのは重要です。例えば、.reg および .bib ファイルを BSP 用に OS デザインで直接編集したり、サブプロジェクトを作成して、より構成要素化した方法で、カスタム設定をランタイム イメージに追加したりすることができます。第 1 章で説明したように、OS デザインの .reg ファイルや .bib ファイルを直接修正するのは、一般的により早く便利な方法ですが、サブプロジェクトは、複数の OS デザイン間でカスタム設定を再利用しやすくします。

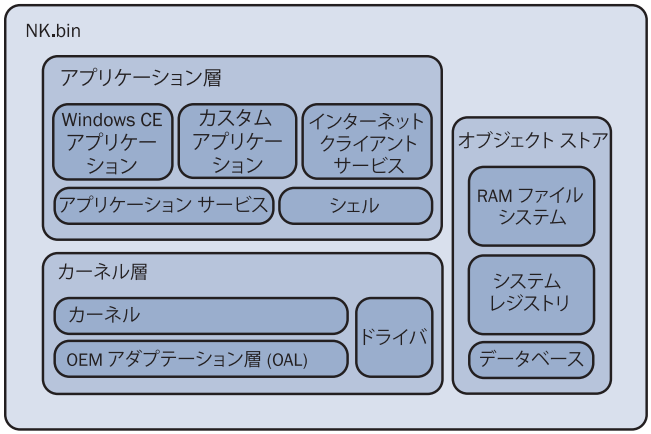


図 2-3　ランタイム イメージのコンテンツ

バイナリ イメージ ビルダ ファイル

Windows Embedded CE ビルド プロセスは、.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。ビルド プロセスの終了時に、[ランタイム イメージの作成] フェーズ中に、[バイナリ イメージの作成] ツール (Makeimg.exe) は、[ファイルのマージ] ツール (Fmerge.exe) を呼び出して、Platform\<BSP 名 >\Files フォルダからの Config.bib や Platform.bib、Project.bib、Common.bib、および他のサブプロジェクト .bib ファイルなどの、すべての該当する .bib ファイルを、リリース ディレクトリの Ce.bib という名前のファイルに結合します。次いで、[バイナリ イメージの作成] ツールは、[ROM イメージ ビルダ] ツール (Romimage.exe) を呼び出して、このファイルを処理し、ランタイム イメージに含むバイナリおよびファイルを決定します。

.bib ファイルには、次のセクションを含めることができます。

- MEMORY　メモリ レイアウトのパラメータを定義します。通常、このセクションは、C:\Wince600\Platform\DeviceEmulator\Files\Config.bib などの BSP の Config.bib ファイルにあります。
- CONFIG　Romimage.exe の構成オプションを定義し、バイナリ ランタイム イメージをカスタマイズします。通常、このセクションは、BSP の Config.bib ファイルにあります。このセクションはオプションです。
- MODULES　Romimage.exe が RAM かエクセキュート イン プレイス (XIP) にロードするとしてマークするファイルのリストを指定します。非圧縮オブジェクト モジュールのみが読み取り専用メモリから直接実行できます。このセクションでネイティブ実行ファイルを列挙できますが、管理された

バイナリはありません。共通言語ランタイム (CLR) は、Microsoft 中間言語 (MSIL) コンテンツを、ランタイム時にネイティブ機械コードに変換する必要があるためです。

- FILES　オペレーティング システムが実行のために RAM にロードする必要のある、実行可能ファイルおよび他のファイルを参照します。このセクションでは、管理されたコード モジュールを指定する必要があります。

.BIB ファイルの MEMORY セクション　Config.bib の MEMORY セクションは、予約済みのメモリ領域を定義し、各領域に名前、アドレス、サイズ、およびタイプを割り当てます。よい例は、デバイス エミュレータ BSP の Config.bib ファイルにある MEMORY セクションです。このデバイス エミュレータ BSP は、CE 6.0 R2 の Platform Builder を使用して、そのまま使用することができます。PLATFORM\<BSP 名>\FILES ディレクトリに Config.bib があります。図 2ミ4 は、Visual Studio 2005 の MEMORY セクションを示しています。

config.bib

```
MEMORY

;
; NK and RAM region definitions.
;
IF IMGFLASH !
#define    NKNAME    NK
#define    NKSTART   80070000
#define    NKLEN     02000000

#define    RAMNAME   RAM
#define    RAMSTART  82070000
#define    RAMLEN    01E7F000

ELSE

#define    NKNAME    NK
#define    NKSTART   88001000
#define    NKLEN     05fff000 // 96mb less 4k

#define    RAMNAME   RAM
#define    RAMSTART  80070000
#define    RAMLEN    03E7F000

ENDIF ; IMGFLASH

    PTS             80000000        00020000    RESERVED
    ARGS            80020000        00000800    RESERVED
    SLEEPSTATE      80020800        00000800    RESERVED
    EBOOT           80021000        00040000    RESERVED
    EBOOT_STACK     80061000        00004000    RESERVED
    EBOOT_RAM       80065000        00006000    RESERVED

    $(NKNAME)       $(NKSTART)      $(NKLEN)    RAMIMAGE
    $(RAMNAME)      $(RAMSTART)     $(RAMLEN)   RAM
```

図 2-4　.bib ファイルからの MEMORY セクション

MEMORY セクションのフィールドは、次のパラメータを定義します。

- 名前 MEMORY セクションの名前です。名前は一意である必要があります。
- アドレス この 16 進数値は、メモリ セクションの開始アドレスを示します。
- サイズ この 16 進数値は、メモリ セクションの全体の長さをバイト単位で定義します。
- タイプ このフィールドは、次のいずれかの値にすることができます。
  - *RESERVED* この領域が予約済みであることを示します。Romimage.exe は、イメージ作成中にこれらのセクションをスキップします。例えば、図 2-4 に示す Ce.bib ファイルには、いくつかの RESERVED セクションが含まれています。ARGS セクションなどは、ブート ローダー (EBOOT) の共有メモリ領域を提供して、スタートアップ (ARGS) 後にシステムと表示バッファの DISPLAY セクションにデータを渡します。他の OS デザインの Ce.bib ファイルは、カーネルがシステム メモリとして使用するようにサポートしていない、別の RESERVED セクションを含んでいます。
  - *RAMIMAGE* .bib ファイルの MODULES および FILES セクションで指定したカーネル イメージと他のモジュールをロードするために、システムが使用できるメモリ領域を定義します。ランタイム イメージは、1 つのみの RAMIMAGE セクションを含めることができ、アドレス範囲は連続している必要があります。
  - *RAM* RAM ファイル システムおよび実行アプリケーション用のメモリ領域を定義します。このメモリ セクションは連続している必要があります。デバイスにある拡張ダイナミック RAM (DRAM) 用などで、不連続なメモリ セクションが必要な場合、BSP の OAL の OEMGetExtensionDRAM 関数を実装することで、不連続メモリを割り当てることができます。Windows Embedded CE サポート物理不連続メモリの最大 2 つセクションまでサポートできます。

.BIB ファイル CONFIG セクション CONFIG セクションは、次のオプションを含めて、ランタイム イメージ用に追加パラメータを定義します。

- AUTOSIZE 自動的に RAMIMAGE および RAM セクションを結合し、RAMIMAGE セクションの未使用のメモリを RAM に割り当てます。必要に

応じて、RAM セクションのメモリを取り分けて、RAMIMAGE に提供します。

- BOOTJUMP　指定されると、ブート ジャンプ ページを、既定の領域ではなく、RAMINAGE セクション内の特定の領域に移動します。
- COMPRESSION　イメージの書き込み可能メモリ セクションを自動的に圧縮します。このオプションの既定値は ON です。
- FIXUPVAR　[バイナリ イメージの作成] フェーズで、カーネル グローバル変数を初期化します。
- FSRAMPERCENT　RAM ファイル システム用に使用する RAM の割り合いをパーセントで設定します。
- KERNELFIXUPS　Romimage.exe がカーネルによって書き込み可能メモリに再割り当てするようにします。このオプションは、通常有効 (ON) です。
- OUTPUT　Romimage.exe が Nk.bin ファイルの出力ディレクトリとして使用するディレクトリを変更します。
- PROFILE　イメージがプロファイラを含めるかどうかを指定します。
- RAM_AUTOSIZE　RAM のサイズを XIP セクションの最後まで拡張します。
- RESETVECTOR　ジャンプ ページを指定された場所に再配置します。これは、MIPS プロセッサが 9FC00000 からブートするのに必要です。
- ROM_AUTOSIZE　ROMSIZE_AUTOGAP 設定を考慮する XIP 領域をリサイズします。
- ROMFLAGS　カーネルの次のオプションを構成します。
  - *デマンド ページング*　実行または一部をページングする前に、ファイルを完全に RAM にコピーします。
  - *フル カーネル モード*　すべての OS スレッドをカーネル モードで実行します。これは、攻撃に対するシステムの脆弱性を残しますが、パフォーマンスは向上します。
  - *ROM モジュールのみ信頼*　ROM のファイルのみを信頼できるファイルとしてマークします。
  - *X86 システム上で X86 TLB をフラッシュ*　パフォーマンスを向上しますが、セキュリティ リストが増大します。
  - */base リンカ設定を重視*　DLL の /base リンカ設定を使用するかどうかを定義します。

- ROMOFFSET 保存場所とは別のメモリ場所のランタイム イメージを実行できるようにします。例えば、ランタイム イメージをフラッシュ メモリに保存して、RAM からコピーおよび実行することができます。
- ROMSIZE ROM のサイズをバイト単位で指定します。
- ROMSTART ROM の開始アドレスを指定します。
- ROMWIDTH データ ビットの量を指定し、Romimage.exe がランタイム イメージを分割する方法を指定します。Romimage.exe は、ランタイム イメージ全体を 1 つのファイルに置いたり、ランタイム イメージを奇数や偶数の 16 ビット文字の 2 つのファイルに分割したり、または偶数や奇数の 8 ビット バイトの 4 つのファイルを作成したりすることができます。
- SRE Romimage.exe が .sre ファイルを生成するかどうかを決定します。Motorola S-record (SRE) は、ほとんどの ROM 作成プログラムで認識されるファイル形式です。
- X86BOOT x86 リセット ベクトル アドレスで、JUMP 命令を追加するかどうかを指定します。
- XIPSCHAIN Chain.bin および Chain.lst ファイルの作成を可能にし、XIP チェーンを設定して、イメージを複数のファイルに分割できるようにします。

.BIB ファイルの MODULES および FILES セクション BSP および OS デザイン開発者は、.bib ファイルの MODULES および FILES セクションをしばしは編集して、新しいコンポーネントをランタイム イメージに追加できるようにする必要があります。MODULES および FILES セクションの形式はほとんど同一ですが、MODULES セクションは、より多くの構成オプションをサポートします。主な違いは、MODULES セクションはメモリで圧縮されていないファイルを列挙して XIP をサポートするのに対し、FILES セクションは、圧縮されたファイルを列挙します。オペレーティング システムは、ファイルにアクセスするときにデータを解凍します。

次のリストは、Platform.bib ファイルからの 2 つの小さなセクション MODULES および FILES を示しています。完全な例については、デバイス エミュレータ BSP の Platform.bib ファイルを確認してください。

```
MODULES
;  名前                    パス                                  メモリ タイプ
;  --------------          -----------------------------          -----------
; @CESYSGEN IF CE_MODULES_DISPLAY
IF BSP_NODISPLAY !
  DeviceEmulator_lcd.dll   $(_FLATRELEASEDIR)\DeviceEmulator_lcd.dll   NK SHK
IF BSP_NOBACKLIGHT !
  backlight.dll            $(_FLATRELEASEDIR)\backlight.dll            NK SHK
ENDIF BSP_NOBACKLIGHT !
ENDIF BSP_NODISPLAY !
; @CESYSGEN ENDIF CE_MODULES_DISPLAY

FILES

;  名前                 パス                                  メモリ タイプ
;  --------------       --------------------------------       -----------
; @CESYSGEN IF CE_MODULES_PPP
dmacnect.lnk            $(_FLATRELEASEDIR)\dmacnect.lnk             NK  SH
; @CESYSGEN ENDIF CE_MODULES_PPP
```

MODULES および FILES セクションのファイル参照で、次のオプションを定義できます。

- 名前　メモリ　テーブルに表示されるモジュールまたはファイルの名前です。通常、この名前はランタイム イメージのファイル名と同一です。
- パス　Romimage.exe がランタイム イメージと結合するファイルへの完全パスです。
- メモリ　Config.bib ファイルの MEMORY セクションのメモリ領域の名前を参照します。このファイルに Romimage.exe はモジュールやファイルをロードします。通常、NK と設定して、MEMORY セクションで定義する NK 領域のファイルを統合します。
- オーバーライド　セクション　FILE　セクションのモジュールおよび MODULES セクションの FILES を指定できるようにします。基本的に、Romimage.exe は、エントリが存在するセクションを無視し、エントリを指定されたセクションのメンバとして取り扱います。このパラメータはオプションです。
- タイプ　ファイル タイプを指定します。表 2-3 に示すように、フラグを組み合わせることもできます。

表 2 ミ 3　MODULES および FILES セクションのファイル タイプ定義

MODULES および FILES セクション	MODULES セクションのみ
■ S　ファイルはシステムファイルです。 ■ H　酵しファイルです。 ■ U　非咎縮ファイルです。(ファイルの持定の設定は、咎縮です) ■ N　信鋸されないモジュ　ルです。 ■ D　デバッグできなモジュ　ルです。	■ K　Ro mi ma g e . e x e　が修正した仮想アドレスを　DL L　のパブリック　エクスポ　トに割り屹て、ユ　ザ　　モ　ドではなく、カ　ネルモ　ドでモジュ　ルを墾行します。ドライバは、カ　ネル　モ　ドで墾行して、基盤となるハ　ドウェアへの直接アクセスを確立する必要があります。 ■ R　リソ　ス　　ファイルを咎縮します。 ■ C　ファイルのすべてのデ　タを咎縮します。ファイルがすでに　RA M　にある場合、ファイルを再度解凍して　RA M　の新しいセクションに置きます。これにより、RA M　をかなり消費することになります。 ■ P　モジュ　ルごとに　CP U　タイプを確認しません。 ■ X　モジュ　ルに署名し、RO M　に署名を含めます。 ■ M　モジュ　ルをオンデマンドでペ　ジングすべきでないカ　ネルを通知します。(デマンド　ペ　ジングの僚果の詳細については、第　3　章を津照してください) ■ L　Ro mi ma g e . e x e　が　RO M　DL L　を分割しないように指示します。

条件付き .bib ファイル処理　.bib ファイルが、環境変数および SYSGEN 変数に基づいた、条件式をサポートしていることは重要です。カタログ項目を介して環境変数を設定し、これらの変数を .bib ファイルの IF 式で確認して、特定のモジュールや他のファイルを含めたり、除外したりすることができます。SYSGEN 変数では、@CESYSGEN IF 式を代わりに使用します。

前のセクションに列挙されている MODULES および FILES は、@CESYSGEN IF および IF 式を使用して、SYSGEN 変数および環境変数に基づいた条件を処理するようにします。例えば、MODULES　セクションの　@CESYSGEN　IF

CE_MODULES_DISPLAY 式は、OS デザインが表示コンポーネントを含む場合に、BSP が表示ドライバを自動的に含めるように指定します。図 2-5 に示すように、表示を使用している OS デザイン用に Visual Studio で [カタログ項目ビュー] を表示することで、Platform Builder が表示コンポーネントを BSP に自動的に追加しているかを確認できます。

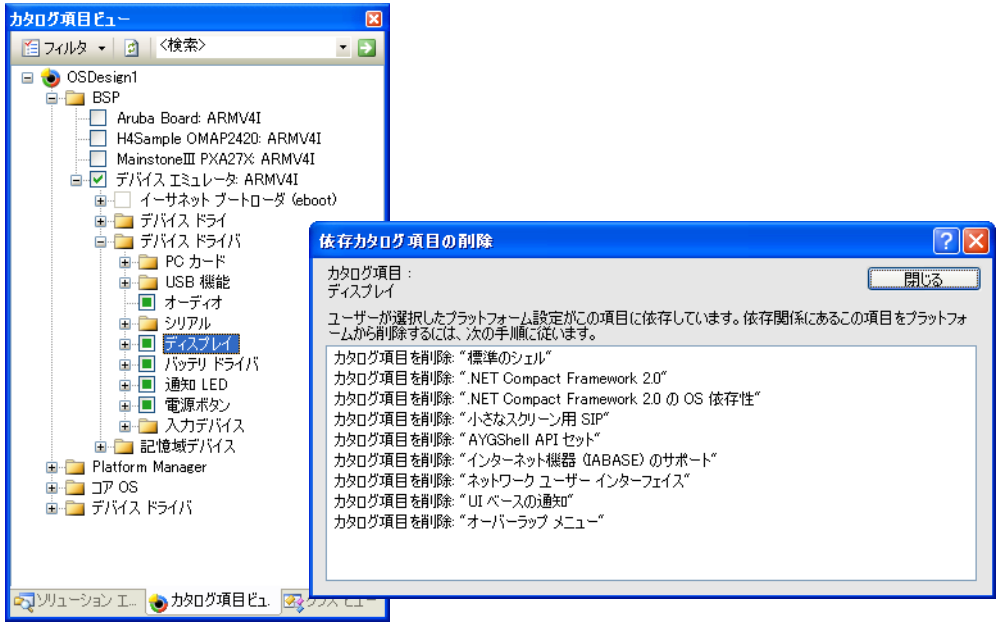


図 2-5 表示項目に基づくコア OS コンポーネント

レジストリ ファイル

レジストリ (.reg) ファイルは、リモート デバイスでシステム レジストリを初期化するために使用されます。これらのファイルは、CE .reg ファイルがヘッダーおよびバージョン情報で開始されないことを除くと、Windows デスクトップ オペレーティング システムのレジストリ ファイルとほとんど同一です。開発コンピュータで誤って CE .reg ファイルをダブルクリックして、デスクトップ レジストリに設定を追加することに同意してしまった場合、ダイアログ ボックスが表示されて、.reg ファイルがレジストリ スクリプトで有効でないことが通知されます。別の相違点は、CE .reg ファイルは .bib ファイルと同様の条件式を含めることができ、それにより選択されたカタログ項目に応じてレジストリ設定をインポートできる点です。デバイス エミュレータ BSP の Platform.reg ファイルからの次のスニペットは、前処理条件の使用法を示しています。

```
; Our variables
#define BUILTIN_ROOT HKEY_LOCAL_MACHINE\Drivers\BuiltIn
;#define PCI_BUS_ROOT $(BUILTIN_ROOT)\PCI
#define DRIVERS_DIR $(_PUBLICROOT)\common\oak\drivers

; @CESYSGEN IF CE_MODULES_RAMFMD
; @CESYSGEN IF FILESYS_FSREGHIVE
; HIVE BOOT SECTION
[HKEY_LOCAL_MACHINE\init\BootVars]
   "Flags"=dword:1          ; common.reg のコメントを参照
; END HIVE BOOT SECTION
; @CESYSGEN ENDIF FILESYS_FSREGHIVE
; @CESYSGEN IF CE_MODULES_PCCARD
; @XIPREGION IF DEFAULT_DEVICEEMULATOR_REG

IF BSP_NOPCCARD !
#include "$(_TARGETPLATROOT)\src\drivers\pccard\pcc_smdk2410.reg"
#include "$(DRIVERS_DIR)\pccard\mdd\pcc_serv.reg"
[HKEY_LOCAL_MACHINE\Drivers\PCCARD\PCMCIA\TEMPLATE\PCMCIA]
   "Dll"="pcmcia.dll"
   "NoConfig"=dword:1
   "NoISR"=dword:1 ; ISR を全くロードしない。
   "IClass"=multi_sz:"{6BEAB08A-8914-42fd-B33F-61968B9AAB32}=
                                                PCMCIA Card Services"
ENDIF ; BSP_NOPCCARD !
; @XIPREGION ENDIF DEFAULT_DEVICEEMULATOR_REG
; @CESYSGEN ENDIF CE_MODULES_PCCARD
```

データベース ファイル

Windows Embedded CE は、データベース (.db) ファイルに依存しており、既定のオブジェクト ストアを定義します。オブジェクト ストアは、トランザクション ベース保存機構です。言い換えると、オペレーティング システムおよびアプリケーションが不揮発性データ 記憶用に使用する RAM のデータベースのリポジトリです。例えば、オペレーティング システムがオブジェクト ストアを使用して、スタックやメモリ ヒープの管理、ファイルの圧縮および解凍、および ROM ベース アプリケーションや RAM ベース データの統合を行います。記憶機構のトランザクション指向の特性により、データをオブジェクト ストアに書き込んでいる間に突然電源遮断が発生した場合でも、データ整合性を保証します。システムの再起動時に、Windows Embedded CE は中断されたトランザクションの完了か、割り込みが発生する前の最新の正常な構成に回復します。システム ファイルの場合、最新の正常な構成とは、Windows Embedded CE が ROM から初期設定を再読み込みが必要であることを意味しています。

システム ファイルを保存する

システム (.dat) ファイル、特に RAM ファイル システムを初期化するための設定を含む Platform.dat および Project.dat を保存します。ターゲット デバイス上でランタイム イメージをコールド スタートすると、Filesys.exe は、これらの .dat ファイルを処理して、RAM ファイル システム ディレクトリ、ファイル、およびリンクをターゲット デバイスに作成します。Platform.dat ファイルは、通常ハードウェア関連のエントリに使用され、Project.dat ファイルは OS デザインに適用されますが、ビルド システムはすべての .dat ファイルを Initobj.dat という名前の 1 つのファイルに最終的にマージするため、既存の .dat ファイルを使用してファイル システム設定を定義することができます。

例えば、Project.dat ファイルをカスタマイズすることで、ランタイム イメージの Windows ディレクトリに加えて、ルート ディレクトリを定義することができます。既定では、ROM イメージに置かれたアイテムは Windows ディレクトリに表示されますが、.dat ファイルを使用することで、ファイルを Windows ディレクトリの外部でも表示されるようにすることもできます。ROM Windows ディレクトリのファイルをコピーしたり、リンクを付けたりすることもできます。デスクトップにショートカットを作成する場合または [スタート] メニュー アプリケーションへのリンクを追加する場合に特に役立ちます。.reg および .bib ファイルと同様に、.dat ファイルで IF および IF ! 条件ブロックを使用できます。

次のリストは、Project.dat ファイルを使用して、「Program Files」および「My Documents」という名前の 2 つの新しいルート ディレクトリを作成し、「My Projects」サブディレクトリを「Program Files」の下に作成し、Myfile.doc ファイルを「Windows」ディレクトリから「My Documents」ディレクトリにマップする方法を示しています。

```
Root:-Directory("Program Files")
Root:-Directory("My Documents")
Directory("\Program Files"):-Directory("My Projects")
Directory("\My Documents"):-File("MyFile.doc", "\Windows\Myfile.doc")
```

レッスンの要約

ビルド システムについての深い理解は、開発時間を短縮し、プロジェクト コストを低減します。ソース コードの変更を素早くテストして、不必要なコンパイル サイクルを無くしたい場合は、ビルド プロセスの各フェーズ中に実行するステップを理解している必要があります。また、.reg、.bib、.db、および .dat ファイルなどの、ランタイム イメージ構成ファイルの目的と場所も理解し、OS デザインを効率的に作成および保守できるようにする必要があります。

Windows Embedded CE ビルド システムは、[ランタイム イメージの作成] フェーズ中に、多様な .reg、.bib、.db、および .dat ファイルを統合ファイルに結合します。ビルド システムは、これを使用して最終的なランタイム イメージを構成します。ターゲット デバイスにランタイム イメージをロードすることなく、特定の設定やファイルが最終イメージを構成することを検証するため、これらのファイルを確認するのは適切です。OS デザインのリリース ディレクトリに、多様なランタイム イメージ構成ファイルがあります。必要なエントリが抜けていることが分かった場合、カタログ アイテムで定義された環境変数や SYSGEN 変数に加えて、条件式も確認します。

ビルド システムは、次のランタイム イメージ構成ファイルを [ランタイム イメージの作成] フェーズで作成します。

- Reginit.ini　Platform.reg、Project.reg、Common.reg、IE.reg、Wceapps.reg、および Wceshell.reg ファイルを結合しています。
- Ce.bib　Config.bib、Platform.bib、Project.bib、および subproject bib ファイルを結合しています。
- Initdb.ini　Common.db、Platform.db、および Project.db ファイルを結合しています。
- Initobj.dat　Common.dat、Platform.dat、および Project.dat ファイルを結合しています。

レッスン 2：ビルド構成ファイルを編集

ランタイム イメージ構成ファイルに加えて、Windows Embedded CE は、ビルド構成ファイルも使用して、ソース コードのコンパイルと、機能バイナリ コンポーネントとの関連付けを行います。特に、ビルド システムは、Dirs、Sources、および Makefile という 3 つのタイプのソース コード構成ファイルに依存しています。これらのファイルは、[ビルド] ツール (Build.exe) およびコンパイラとリンカ (Nmake.exe) をスキャンするソース コード ディレクトリ、およびビルドするバイナリ コンポーネントのタイプに関する情報と共に提供します。CE 開発者として、第 1 章で説明されている手順に従い、パブリック カタログ項目を複製するときなど、これらのファイルを頻繁に編集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：

- ビルド プロセス中に使用するソース コード構成ファイルを識別する。
- ビルド構成ファイルを編集してアプリケーション、DLL、および静的ライブラリを生成する。

レッスン時間 (推定)：25 分

Dirs ファイル

Dirs ファイルは、ソース コード ファイルを含むディレクトリを識別子、ビルド システムに含めます。Build.exe が Dirs ファイルを実行中のフォルダで検出すると、Dirs ファイルで参照されているサブディレクトリをスキャンし、これらのサブディレクトリのソース コードをビルドします。この他に、この機構によってランタイム イメージの一部を選択的に更新することができます。Subproject1 のソース コードに変更を加える場合、Subproject1 ディレクトリで Build.exe を実行することで、このサブプロジェクトを選択的にリビルドできます。該当するディレクトリ参照を Dirs ファイルから削除するか、条件式を使用することで、ソース コード ツリーのディレクトリをビルド プロセスから除外することもできます。

Dirs ファイルは、直接的なコンテンツ構造を持つテキスト ファイルです。DIRS、DIRS_CE、または OPTIONAL_DIRS を使用してから、単一行または複数行 (各行を円記号 (\) で終了して次の行に続くことを示す) でサブディレクトリのリストを指定します。DIRS キーワードを使用して参照されたディレクトリは、ビルド システムに常に含まれます。DIRS_CE キーワードを代わりに使用する場合、ソース コードがとりわけ Windows Embedded CE ランタイム イメージ用に記述されていれば、Build.exe はソース コードのみをビルドします。OPTIONAL_DIRS キーワードは、オプションのディレクトリを指定します。ただし、Dirs ファイルは

DIRS 指示子のみを含むことができることに留意してください。Build.exe は、リストされている順番でディレクトリを処理するため、必須のディレクトリを最初にリストするようにしてください。ワイルドカード「*」を使用して、すべてのディレクトリを含めることもできます。

既定の Windows Embedded CE コンポーネントから取った次のリストでは、Dirs ファイルを使用して、ソース コード ディレクトリをビルド プロセスに含める方法を示します。

```
# C:\WINCE600\PLATFORM\DEVICEEMULATOR\SRC\Dirs
------------------------------------------------
DIRS=common     \
     drivers    \
     apps       \
     kitl       \
     oal        \
     bootloader

# C:\WINCE600\PLATFORM\H4SAMPLE\SRC\DRIVERS\Dirs
------------------------------------------------------
DIRS=        \
# @CESYSGEN IF CE_MODULES_DEVICE
    buses \
    dma \
    triton \
# @CESYSGEN IF CE_MODULES_KEYBD
    keypad \
# @CESYSGEN ENDIF CE_MODULES_KEYBD
# @CESYSGEN IF CE_MODULES_WAVEAPI
    wavedev \
# @CESYSGEN ENDIF CE_MODULES_WAVEAPI
# @CESYSGEN IF CE_MODULES_POINTER
    touch \
    tpagent \
# @CESYSGEN ENDIF CE_MODULES_POINTER
# @CESYSGEN IF CE_MODULES_FSDMGR
    nandflsh \
# @CESYSGEN ENDIF CE_MODULES_FSDMGR
# @CESYSGEN IF CE_MODULES_SDBUS
    sdhc \
# @CESYSGEN ENDIF CE_MODULES_SDBUS
# @CESYSGEN IF CE_MODULES_DISPLAY
    backlight \
# @CESYSGEN ENDIF CE_MODULES_DISPLAY
# @CESYSGEN IF CE_MODULES_USBFN
    usbd \
# @CESYSGEN ENDIF CE_MODULES_USBFN
# @CESYSGEN ENDIF CE_MODULES_DEVICE
# @CESYSGEN IF CE_MODULES_DISPLAY
    display \
# @CESYSGEN ENDIF CE_MODULES_DISPLAY
```

ノート [ソリューション エクスプローラ] での Dirs ファイルの編集

Windows Embedded CE 6.0 R2 の Platform Builder を使用する Visual Studio の [ソリューション エクスプローラ] は、Dirs ファイルを使用して OS デザイン プロジェクトの Windows Embedded CE ディレクトリ構造の動的ビューを生成します。ただし、[ソリューション エクスプローラ] で Dirs ファイルを編集するとビルド順序が変わり、再度ビルドを行って解決する必要があるビルド エラーが発生する可能性があるため、[ソリューション エクスプローラ] のディレクトリを追加または削除すべきではありません。

Sources ファイル

C:\Wince600\OSDesigns\OSDesign1 などの、標準 OS デザインのフォルダおよびファイルを確認する場合、プロジェクトが既定で Dirs ファイルを含んでいないことを確認します。カスタム コンポーネントおよびアプリケーション用にサブプロジェクトを含める場合、Sources ファイルを各サブプロジェクトのルート フォルダで代わりに検索します。Sources ファイルは、ビルド指示子を含む、ソース コード ファイルに関する詳細情報を提供します。これは Dirs ファイルは提供できません。ただし、ソース コード ディレクトリは、1 つの Dirs ファイルまたは 1 つの Sources ファイルのみを含めることができ、両方を含めることはできません。これは、Sources ファイルのあるディレクトリは、他のコードを含むサブディレクトリを含めることができないことを示しています。ビルド プロセス中に、Nmake.exe は Sources ファイルを使用して、ビルドするファイル タイプ (.lib、.dll、または .exe) およびビルド方法を定義します。Dirs ファイルと同様に、Sources ファイルでは、各行末に円記号 (\) を置いて次の行に継続させない限り、宣言を 1 行で指定する必要があります。

次のリストは、デバイス エミュレータ BSP の Sources ファイルのコンテンツを示しています。既定では、このファイルは C:\Wince600\Platform\DeviceEmulator\Src\Drivers\Pccard フォルダにあります。

```
WINCEOEM=1

TARGETNAME=pcc_smdk2410
TARGETTYPE=DYNLINK
RELEASETYPE=PLATFORM
TARGETLIBS=$(_COMMONSDKROOT)\lib\$(_CPUINDPATH)\coredll.lib \
           $(_SYSGENOAKROOT)\lib\$(_CPUINDPATH)\ceddk.lib

SOURCELIBS=$(_SYSGENOAKROOT)\lib\$(_CPUINDPATH)\pcc_com.lib

DEFFILE=pcc_smdk2410.def
DLLENTRY=_DllEntryCRTStartup

INCLUDES=$(_PUBLICROOT)\common\oak\drivers\pccard\common;$(INCLUDES)
```

```
SOURCES= \
    Init.cpp \
    PDSocket.cpp \
    PcmSock.cpp  \
    PcmWin.cpp

#xref VIGUID {549CAC8D_8AF0_4789_9ACF_2BB92599470D}
#xref VSGUID {0601CE65_BF4D_453A_966B_E20250AD2E8E}
```

Sources ファイルでは、次の指示子を定義できます。

- TARGETNAME　ターゲット　ファイル名で、ファイル名拡張子なしで指定します。
- TARGETTYPE　次のような、ビルドするファイルのタイプを定義します。
- DYNLINK　ダイナミック リンク ライブラリ (DLL)。
- LIBRARY　静的リンク ライブラリ (LIB)。
- PROGRAM　実行可能プログラム (EXE)。
- NOTARGET　なにもファイルをビルドしません。
- RELEASETYPE　次のように、Nmake.exe がターゲット ファイルを置くディレクトリを指定します。
  - *PLATFORM*　PLATFORM\(BSP 名)\< *ターゲット* >。
  - *OAK*、*SDK*、*DDK*　%_PROJECTROOT%\Oak\< *ターゲット* >。
  - *LOCAL*　現在のディレクトリ。
  - *CUSTOM*　TARGETPATH で指定されたディレクトリ。
  - *MANAGED*　%_PROJECTROOT%\Oak\< ターゲット >\Managed。
- TARGETPATH　RELEASETYPE=CUSTOM のファイル パスを定義します。
- SOURCELIBS　TARGETNAME で指定されたターゲット ファイルとリンクされたライブラリを指定して、最終バイナリ出力を作成します。このオプションは、通常、.lib ファイルを作成するのに使用され、.dll や .exe ファイルの作成には使用されません。
- TARGETLIBS　追加ライブラリおよびオブジェクト　ファイルを指定して、最終バイナリ出力にリンクします。特に、.dll や .exe ファイルの作成に使用され、.lib ファイルの作成には使用されません。
- INCLUDES　追加ディレクトリをリスト表示して、含めるファイルを検索します。

- SOURCES　この特有のコンポーネントに対して使用するソース　コードを定義します。
- ADEFINES　アセンブラのパラメータを指定します。
- CDEFINES　コンパイラのパラメータを指定します。IFDEF 式での使用のために追加の DEFINE 式として使用することができます。
- LDEFINES　リンカ定義を設定します。
- RDEFINES　リソース コンパイラの DEFINE 式を指定します。
- DLLENTRY　DLL のエントリ ポイントを定義します。
- DEFFILE　DLL のエクスポートされたシンボルを含む .def ファイルを定義します。
- EXEENTRY　実行可能ファイルのエントリ ポイントを設定します。
- SKIPBUILD　ターゲットのビルドを、実際にターゲットをビルドせずに、成功したとしてマークします。
- WINCETARGETFILE0　現在のディレクトリをビルドする前にビルドされる必要のある非標準ファイルを指定します。
- WINCETARGETFILES　このマクロ定義は、Build.exe が現在のディレクトリにある他のターゲットをリンクした後に、Build.exe がビルドする必要のある非標準ターゲット ファイルを指定します。
- WINCE_OVERRIDE_CFLAGS　コンパイラ フラグを定義して、既定の設定をオーバーライドします。
- WINCECPU　コードに特定の CPU タイプが必要であり、特定の CPU に対してのみビルドされる必要があることを指定します。

ノート　ビルド前後に特定のアクションを実行する

標準の指示子以外に、Windows Embedded CE の Sources ファイルは、PRELINK_PASS_CMD および POSTLINK_PASS_CMD 指示子をサポートします。これらの指示子を使用して、PRELINK_PASS_CMD=pre_action.bat および POSTLINK_PASS_CMD=post_action.bat などのビルド プロセスの前後に、コマンド ライン ツールやバッチ ファイルに基づいてカスタム アクションを実行することができます。これは、カスタム アプリケーションの開発時に、追加ファイルをリリース ディレクトリにコピーしたい場合などに役立ちます。

Makefile ファイル

サブプロジェクト フォルダのコンテンツを詳しく見ると Makefile が既定の前処理指示子、コマンド、マクロ、および他の式を Nmake.exe に提供していることを理解できます。ただし、Windows Embedded CE では、この Makefile には、%_MAKEENVROOT%\Makefile.def を参照する単一行のみが含まれています。既定では、環境変数 %_MAKEENVROOT% は C:\Wince600\Public\Common\Oak\Misc フォルダを指定しており、この場所の Makefile.def ファイルは、すべての CE コンポーネント用の標準 Makefile となっているため、このファイルを修正すべきではありません。他にも、Makefile.def ファイルには、!INCLUDE $(MAKEDIR)\sources などの Sources ファイルにプルする include 式が含まれており、サブプロジェクト フォルダから Sources ファイルを指定します。サブプロジェクト フォルダで Sources ファイルを編集して、Nmake.exe がターゲット ファイルをビルドする方法を調整する必要があります。

レッスン概要

Windows Embedded CE 6.0 R2 開発環境は、Makefile、Sources、および Dirs ファイルに依存しており、Build.exe および Nmake.exe がソース コードをコンパイルして、ランタイム イメージ用に機能バイナリ コンポーネントに関連付ける方法をコントロールします。Dirs ファイルを使用してビルド プロセスに含まれるソース コード ディレクトリを定義したり、Sources ファイルを使用して詳細にビルド指示子をコンパイルおよびビルドするよう指定したりできます。対照的に、Makefile はカスタマイズが全く必要ありません。これは、一般的な事前処理指示子、コマンド、マクロ、およびビルド システムの他の処理命令を使用して、既定の Makefile.def ファイルを参照するだけです。パブリック カタログ項目を複製や新しいコンポーネントの作成を効率的に行うには、ファイルの目的およびビルド プロセスをコントロールする方法をよく理解しておく必要があります。

レッスン 3：ビルド結果の分析

ソフトウェア開発サイクル中に、ビルド エラーに遭遇することは必ずあります。実際、コンパイル エラーをソース コードの構文チェックとして使用するのは一般的ではありませんが、Visual Studio 2005 で使用可能な IntelliSense ィ および他のコード支援を使用してタイプミスや他の構文エラーを提言することはできます。構文エラーは、[出力] ウィンドウで該当するエラーメッセージをダブルクリックしてソース コード ファイルの問題のある重要な行にジャンプできるため、修正は比較的単純です。ただし、コンパイラ エラーは、発生する可能性のあるビルド エラーの 1 つに過ぎません。他の一般的なエラーとしては、数式エラー、式計算エラー、リンカ エラー、およびランタイム イメージ構成ファイルに関連したエラーなどがあります。エラー メッセージに加えて、ビルド システムは、ステータス メッセージおよび警告を生成して、ビルドの問題を分析および診断できるようにします。ビルド プロセス中に生成される情報は、膨大な量に及ぶことがあります。ビルド エラーを効率的に識別、特定、および解決したい場合は、ビルド メッセージの異なるタイプと一般的な形式を理解しておくことが必要です。

このレッスンを終了すると、以下をマスターできます：

- ビルド レポートを特定し、分析する。
- ビルドの問題を診断し、解決する。

レッスン時間 (推定)：15 分

ビルド レポートを理解する

Visual Studio IDE またはコマンド プロンプトでビルドを実行すると、ビルド プロセスは大量のビルド情報を出力します。ビルド システムは、Build.log ファイル のこの情報を追跡します。コンパイルやリンカ警告およびエラーに関する詳細は、Build.wrn および Build.err ファイルにもあります。Visual Studio の [ビルド] メニューで該当するコマンドの 1 つを使用して OS デザイン用に完全なビルドやシステム生成操作を開始する場合、ビルド システムは、これらのファイルを %_WINCEROOT% フォルダ (既定では、C:\Wince600) に書き込みます。対照的に、[ソリューション エクスプローラ] でサブプロジェクト フォルダを右クリックしたり、コンテキスト メニューで [ビルド] コマンドをクリックしたりして、特定のコンポーネントに対してのみビルドを実行する場合、ビルド システムはこれらのファイルを特定のディレクトリに書き込みます。どちらの場合でも、ビルド プロセス中に警告やエラーが発生した場合は、Build.wrn および

Build.err のみが存在します。ただし、これらのファイルをメモ帳やテキスト エディタを使用して開き解析する必要はありません。CE 6.0 R2 の Platfrom Builder を使用する Visual Studio 2005 は、ビルド プロセス中にこの情報を [出力] ウィンドウに表示します。また、[表示] メニューの [他のウィンドウ] にある、[エラー一覧] をクリックして表示される [エラー一覧] ウィンドウのステータス メッセージ、警告、およびエラーを調べることもできます。

図 2ミ6 は、ドッキング解除して表示した [出力] ウィンドウおよび [エラー一覧] を示しています。[出力] ウィンドウは、[出力元の表示] から [ビルド] を選択した場合、Build.log コンテンツを表示します。[エラー一覧] ウィンドウは、Build.wrn および Build.err ファイルのコンテンツを表示します。

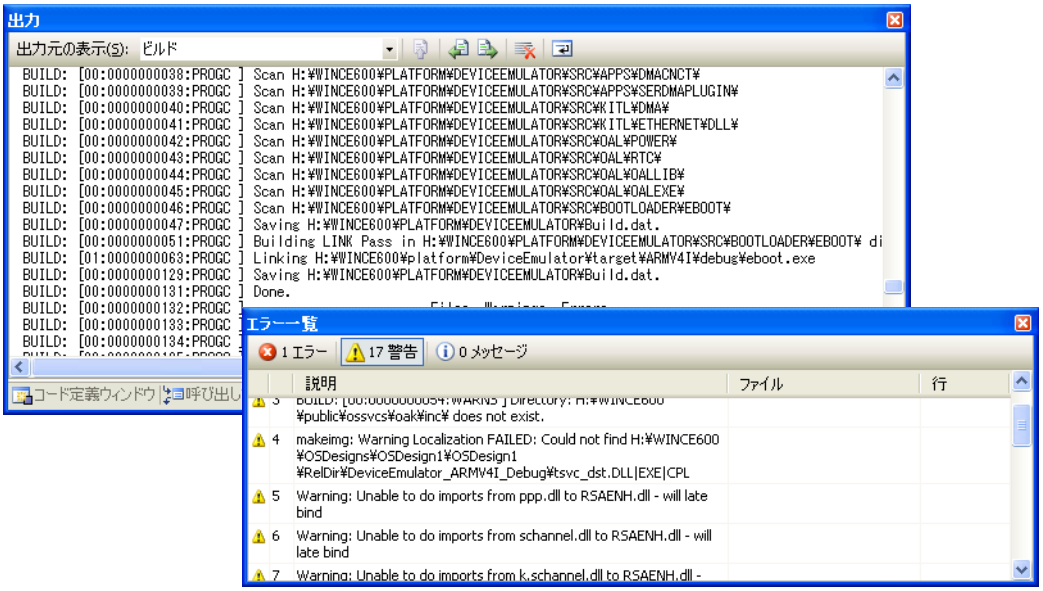


図 2-6　Visual Studio の [出力] ウィンドウおよび [エラー一覧] ウィンドウとビルド情報

特に、ビルド ログ ファイルで次の情報を確認できます。

- Build.log　ビルド プロセス中の各フェーズで生成された個別コマンドに関する情報が含まれます。この情報は、一般的なビルド プロセスおよび特殊なビルド エラーの両方を分析するのに役立ちます。
- Build.wrn　ビルド　プロセス中に生成された警告に関する情報が含まれます。可能な場合、警告の原因を除去するか、少なくとも警告の理由を特定します。Build.wrn の情報は、Build.log にも含まれます。

- Build.err　ビルド プロセス中に遭遇したビルド エラーに関する特定の情報が含まれます。この情報は、Build.log の追加の詳細情報とともに使用可能です。このファイルはエラーが発生したときにのみ作成されます。

ノート　ビルド ステップを識別する
ビルド システムは、スキップされた、および含められたビルド フェーズを Build.log ファイルで追跡します。例えば、「CEBUILD: システム生成フェーズに直接スキップ」エントリは、ビルド システムがコンポーネントのコンパイル フェーズをスキップしたことを示します。システム生成フェーズが開始される場所、ビルド プロセスが SYSGEN から BUILD に移行する方法、および BUILD が MAKEIMG に移行する方法を決定できます。

ビルドの問題のトラブルシューティング

ビルド ログ ファイルの分析により、一般的なビルド プロセスに関する深い理解を得られますが、これは、ビルド エラー のトラブルシューティング時に特に役立ちます。エラー メッセージがソース コード ファイルに関連する場合、[エラー一覧] ウィンドウのメモリ エントリをダブルクリックすることで、コードの該当行にジャンプすることができます。ただし、すべてのビルド エラーがソース コードに関係するわけではありません。ライブラリ参照が欠落しているとリンカ エラーが発生し、コンポーネント ファイルが欠落しているとシステム生成エラーが発生し、ディスク容量を使い切るとコピー エラーが発生し、ランタイム イメージ構成ファイルの設定が正しくないとランタイム イメージの作成エラーが発生し、ビルド プロセスの失敗の原因となります。

システム生成フェーズのエラー

システム生成エラーは、通常、ファイルが欠落していることが原因です。Build.log ファイルは、理由に関する詳細情報を提供します。OS デザインに最近追加したコンポーネントや削除したコンポーネントは、必要な依存関係が使用できなかった場合に、このタイプのエラーの原因となることがあります。システム生成エラーを診断するには、カタログ項目およびその依存関係に関連したすべての変更を検証するのがよい方法です。コンポーネントの中には、通常のシステム生成サイクルではなく、クリーン システム生成の実行する必要のあるものもあります。通常、リリースやデバッグ ビルド構成で [クリーン システム生成] を実行すると、別のビルド構成でも通常の [システム生成] を実行しなければならなくなるため、[クリーン システム生成] コマンドを使用すべきではありません。ただし、カタログ項目を追加または削除した後に、システム生成ビルド エラーが発生した場合は、次の通常のシステム生成時に、[クリーン システム生成] を実行して問題を解決する必要があるかもしれません。

ビルド フェーズ中のエラー

ビルド エラーは、通常、コンパイラ エラーかリンカ エラーが原因となります。コンパイラ エラーは構文エラーで、関数呼び出しにパラメータがないか不正なパラメータがある、ゼロで除算されているというような問題のためにコンパイラが有効なバイナリ コードを生成できないことを示します。コンパイラ エラーをダブルクリックすることで、コードの問題のある重要な行にジャンプできます。ただし、コンパイラ エラーは、他のコンパイラ エラーが原因で発生することもあることに留意してください。例えば、不正確な変数宣言は、変数が多くの場所で使用されていれば、多数のコンパイラ エラーを引き起こすことがあります。一般に、エラー リストの先頭から初めて、コードを修正し、再コンパイルするのがよい方法です。小さなコード変更であっても、リストから多数のエラーを除去できることがしばしばあります。

リンカ エラーは、コンパイラ エラーよりもトラブルシューティングが難しいです。一般に、ライブラリの欠落か互換性のないライブラリが原因となります。不正確に実装された API は、リンカがエクスポートされた DLL 関数への外部参照を解決できなかった場合に、リンカ エラーを引き起こすことがあります。他の一般的な原因としては、ルートで環境変数が不正確に初期化されていることがあります。ビルド ファイル、特に Sources ファイルは、参照ライブラリを指定するのに、ハードコードされたディレクトリ名ではなく、環境変数を使用します。これらの環境変数が設定されていないと、リンカはライブラリの場所を特定できません。例えば、Windows Embedded CE を既定の構成でインストールした場合、%_WINCEROOT% は C:\Wince600 を指定している必要があり、%_FLATRELEASEDIR% は、現在のリリース ディレクトリを指定している必要があります。環境変数の値を検証するには、Visual Studio で [ビルド] メニューを開き、[ビルド ウィンドウのリリース ディレクトリを開く] を選択してから、コマンド プロンプトで、set _winceroot などの環境変数を使用するか使用しないで、set コマンドを実行します。パラメータを使用せずに set コマンドを実行すると、すべての環境変数が表示されますが、リストが非常に長くなります。

リリース コピー フェーズでのエラー

[リリース コピー] フェーズ中に発生した Buildrel エラーは、通常、ハード ドライブ容量が不十分な場合に発生します。[リリース コピー] フェーズ中に、ビルド システムはファイルをリリース ディレクトリにコピーします。この場合、ハード ドライブ容量を空けるか、OS デザイン フォルダを別のドライブに置く必要があるかもしれません。パスや OS デザイン名にスペースがあると、ビルド プロセス中にエラーが発生するため、OS デザイン フォルダへの新しいパスにスペースがないことを確認してください。

ランタイム イメージの作成フェーズのエラー

ビルド プロセスの最終フェーズで発生するエラーは、通常、ファイルの欠落が原因です。これは、前のステップでコンポーネントのビルドに失敗したが、それでもビルド プロセスで [ランタイム イメージの作成] フェーズに進んでしまった場合に発生することがあります。ビルド システムが Reginit.ini ファイルや Ce.bib ファイルを作成できないときに、.reg ファイルや .bib ファイルの構文エラーによってこの状況が発生することがあります。ビルド プロセス中に、これらのファイルの作成のために、Makeimg.exe によって FMerge ツール (FMerge.exe) が呼び出され、不正な条件式などのために失敗すると、イメージ作成エラーが発生します。他のエラーの可能性としては「Error: Image Exceeds (X)」で、これは、イメージが Config.bib で指定された最大可能サイズよりも大きいことを意味しています。

レッスン概要

Windows Embedded CE 6.0 R2 の Platform Builder は、Visual Studio 2005 のビルド ログ システムと統合して、ビルド プロセス中に生成され、Build.log、Build.wrn、および Build.err で追跡されているステータス情報、警告およびエラー メッセージへの便利なアクセスを提供します。Visual Studio でビルド プロセスを開始する方法に応じて、これらのファイルは %_WINCEROOT% フォルダかサブプロジェクト フォルダに存在しますが、これらのファイルのコンテンツを [出力] ウィンドウや [エラー一覧] ウィンドウから直接分析できるため、ファイルの実際の場所は重要ではありません。メモ帳かテキスト エディタでこれらのファイルを開く必要はありません。

ビルド ログ ファイルを分析することで、一般的なビルド プロセスおよび特殊なビルドの問題についての深い理解が得られます。時折発生する一般的なビルドの問題は、コンパイラ エラー、リンカ エラー、システム生成エラー、ビルド エラー、および [リリース コピー] および [ランタイム イメージの生成] フェーズ中に発生する他のエラーです。ビルド エラーがソース コード ファイルの行に直接関連している場合、[エラー一覧] ウィンドウでメッセージ エントリをダブルクリックして、Visual Studio で自動的にソース コード ファイルを開いて重要な問題の発生している行にジャンプすることができます。ハード ドライブ容量が不十分なために発生する buildrel エラーなどの他のエラーは、Visual Studio IDE 外部でトラブルシューティング手順を実行する必要があります。

レッスン 4：ターゲット プラットフォームでのランタイム イメージの展開

すべてのビルドの問題を解決し、ランタイム イメージの生成に成功すると、ターゲット デバイスに Windows Embedded CE を展開する準備が整います。このタスクを達成するためのいくつかの方法があります。選択する方法は、ターゲット デバイスで Windows Embedded CE をロードするのに使用するスタートアップ プロセスに依存しています。いくつかの方法で、Windows Embedded CE 6.0 ランタイム イメージを開始することができます。ランタイム イメージをターゲット デバイスに ROM ツールを使用して展開する必要がある場合、イメージを ROM から直接開始することができます。また、ブート ローダーを使用して、デバイスが起動するたびにランタイム イメージをダウンロードするか、再利用のために不揮発性メモリにイメージを保存することができます。Windows Embedded CE 6.0 R2 は、特定の要件に応じてカスタマイズ可能な汎用ブート ローダー コードとともに提供されます。また、サードパーティ ブート ローダーを実装するのも直接的な方法です。基本的に、Windows Embedded CE は、ほとんどすべての起動環境に適用させることができ、開発サイクル中に素早くかつ便利にランタイム イメージをダウンロードし、エンド ユーザーにリリースできるようにします。

このレッスンを終了すると、以下をマスターできます：

- ランタイム イメージをターゲット デバイスに実装する方法を決定する。
- Platform Builder を構成して、正しい展開層を選択する。

レッスン時間 (推定)：15 分

展開方法の選択

ランタイム イメージを展開するには、ターゲット デバイスへの接続を確立する必要があります。これには、Platform Builder がデバイスと通信する方法を決定する、いくつかの通信パラメータを構成する必要があります。

Windows Embedded CE の [コア接続] インフラストラクチャは、多様なダウンロード方式およびトランスポート機構を備えており、異なる通信機能のハードウェア プラットフォームに適応します。通信パラメータをターゲット デバイス用に定義するには、Visual Studio で [ターゲット] メニューを開き、[接続オプション] を選択します。これで、[ターゲット デバイスの接続オプション] ダイアログ ボックス が表示されます。既定では、Platform Builder は、[ターゲット デバイス] リスト ボックスの [CE デバイス] という名前のターゲット デバイス

を提供しますが、[デバイスの追加] リンクをクリックすることで一意な名前で追加デバイスを作成することもできます。

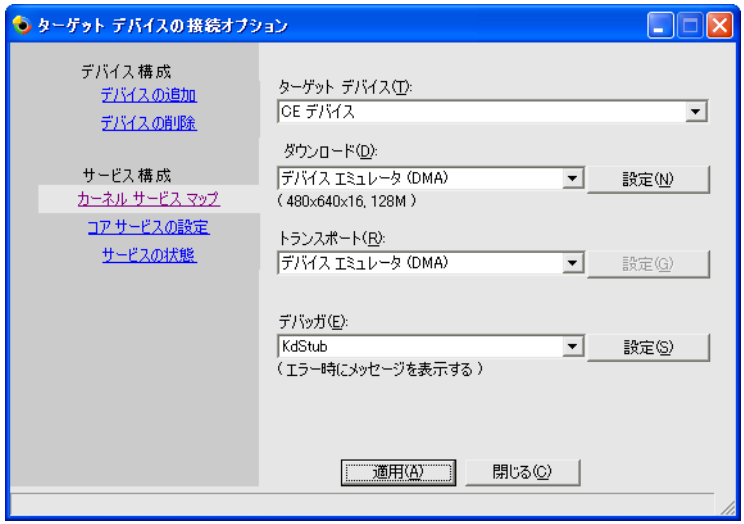


図 2-7　[ターゲット デバイスの接続オプション] ウィンドウ

レイヤのダウンロード

[ダウンロード] リスト ボックスおよび関連する [設定] ボタンにより、ランタイム イメージをターゲット デバイスにダウンロードするのに使用するダウンロード サービスを構成できます。[コア接続] インフラストラクチャは、ランタイム イメージの展開用に次のダウンロード層をサポートします。

- イーサネット　イーサネット接続を介して、ランタイム イメージをダウンロードします。[設定] ボタンを使用して、イーサネット ダウンロード サービスを構成します。開発ワークステーションおよびターゲット デバイスは、同一のサブネットにある必要があります。そうでないと、ターゲット デバイスに接続することができません。
- シリアル　RS232 接続を介して、ランタイム イメージをダウンロードします。[設定] ボタンを使用して、ポート、ボー レート、および他のシリアル通信パラメータを構成します。
- デバイス エミュレータ (DMA)　直接メモリ アクセス (DMA) を介して、ランタイム イメージをデバイス エミュレータにダウンロードします。[設定] ボタンを使用して、デバイス エミュレータを構成します。
- USB　ユニバーサル シリアル バス (USB) 接続を介して、ランタイム イメージをダウンロードします。構成の必要な設定はありません。

- イメージの更新　デバイスのフラッシュ　メモリのイメージを更新します。構成の必要な設定はありません。
- なし　ランタイム　イメージをダウンロードまたは更新したくない場合に、このオプションを選択します。

トランスポート層

ランタイム イメージをリモート デバイスに転送した後、OS デザインの KITL (Kernel Independent Transport Layer) を有効にしたい場合に、デバイスに接続します。一般に、選択されたカーネル トランスポート サービスは、[ダウンロード] リスト ボックスで選択されたダウンロード サービスと一致している必要があります。[コア接続] インフラストラクチャは、次のトランスポート層のオプションをサポートしています。

- イーサネット　イーサネット接続を介して、ターゲット デバイスと通信します。この接続は、ダウンロード デバイスと同じ設定を使用します。
- シリアル　RS232 接続を介して、ターゲット デバイスと通信します。この接続は、ダウンロード デバイスと同じ設定を使用します。
- デバイス エミュレータ (DMA)　直接メモリ アクセス (DMA) を介して、デバイス エミュレータと通信します。
- USB　USB 接続を介して、ターゲット デバイスと通信します。
- なし　ターゲット デバイスとの通信を無効にします。

デバッガ オプション

OS デザインで 1 つまたは複数のデバッガのサポートを有効にする場合、デバッガ名が [デバッガ] リスト ボックスにオプションとして表示されます。既定では、次のデバッガ オプションが使用可能です。

- サンプル　デバイス　エミュレータ　eXDI2　ドライバ　これは、Windows Embedded CE 6.0 R2 に含まれるサンプルの XRI (Extensible Resource Identifier) データ交換 (XDI) ドライバです。XDI は、標準ハードウェア デバッグ インターフェイスです。
- KdStub　カーネル デバッガです。KdStub は、カーネル デバッガ Stub の略で、Platform Builder と Visual Studio がソフトウェア デバッガを使用するように指示します。
- CE ダンプ ファイル リーダー　[エラー レポート生成プログラム] カタログ項目 を OS デザインに追加すると、このオプションをエラー発生後のデバッグに使用できます。
- なし　デバッガを使用したくない場合に、このオプションを選択します。

デバイスへの接続

デバイス接続を構成すると、[コア接続] インフラストラクチャを使用して、ランタイム イメージをターゲット デバイスやデバイス エミュレータに転送する準備ができます。これは、Visual Studio 2005 では、[ターゲット] メニューで使用可能な [デバイスの接続] コマンドを使用して実行します。KITL や [コア接続] インフラストラクチャをデバッグに使用する予定がない場合でも、デバイスに接続して、Platform Builder がランタイム イメージをダウンロードできるようにしておく必要があります。

イメージのダウンロード後、開始プロセスが開始し、ターゲットデバイスで有効になっていれば KITL がアクティブになり、開始プロセスに従ってカーネル デバッガを使用することができ、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 プロセスをデバッグします。KITL を使用することで、Platform Builder を使用する Visual Studio で使用可能なリモート ツールを [ターゲット] メニューで利用することができます。例えば、[ファイル ビューア] でデバイスのファイル システムと対話したり、[レジストリ エディタ] でデバイスのレジストリ設定にアクセスしたり、[パフォーマンス モニタ] でリソースの使用率と応答時間を分析したり、[カーネル トラッカ] および他のリモート ツールで実行中のシステムの詳細情報を表示したりします。システム デバッグに関する詳細情報については、第 4 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。

レッスン概要

Windows Embedded CE は、多様なデバイス接続を介してランタイム イメージ展開をサポートして、(イーサネット接続、シリアル接続、DMA、および USB 接続を含む) 多様な要件や機能のハードウェア プラットフォームに適応します。例えば、デバイス エミュレータ上で CE 6.0 R2 を展開したい場合、DMA は正しい選択です。通信パートナーを構成するだけで、Platform Builder を使用する Visual Studio 2005 の [ターゲット] メニューで [デバイスの接続] をクリックして、Windows Embedded CE の展開の準備ができます。

ヒント
認定試験に合格するには、Windows Embedded CE ランタイム イメージを展開する多様な方式に通じている必要があります。特に、デバイス エミュレータのランタイム イメージを展開する方法に通じている必要があります。

演習 2 ランタイム イメージのビルドおよび展開

この演習では、デバイス エミュレータ BSP に基づいて、OS デザインのビルドおよび展開を行い、Visual Studio の [出力] ウィンドウのビルド情報を分析して多様なビルド フェーズの開始を識別してから、ランタイム イメージをダウンロードするためにターゲット デバイスへの接続を構成します。ターゲット デバイスをカスタマイズする方法を実演するため、[デバイス エミュレータ] 構成を修正して、より大きい画面解像度をサポートし、ネットワーク接続を有効にします。最後のステップでは、ランタイム イメージをダウンロードし、カーネル デバッガのあるターゲット デバイスと接続することで、Windows Embedded CE 開始プロセスを詳細に調べることができます。Visual Studio で初期 OS デザインを作成するには、演習 1 「OS デザインの作成、構成、およびビルド」に概略されている手順に従ってください。

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するために、この本の付属物中のドキュメント「演習 2 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。

OS デザイン用にランタイム イメージをビルドする

1. 演習 1 を終了した後、図 2-8 に示すように、Visual Studio の [ビルド] メニューにある [詳細なビルド コマンド] から [システム生成] を選択します。別の方法として、[ビルド] メニューから [ビルド ソリューション] を選択することもできます。これにより、システム生成ステップで開始するビルドを実行されます。

ヒント　システム生成操作

システム生成操作は、完了まで最大 30 分要します。時間を節約するため、OS デザインを変更するたびに毎回 [システム生成] を実行しないでください。その代わり、すべての必要なコンポーネントの追加および削除後に [システム生成] を実行します。

2. [出力] ウィンドウのビルド プロセスに従います。ビルド情報を調べて、SYSGEN、BUILD、BUILDREL、および MAKEIMG ステップを確認します。Ctrl+F を押して [検索と置換] ダイアログ ボックスを表示してから、次のテキストを検索して、これらのフェーズの開始を確認します。
   a. *Starting Sysgen Phase For Project*　SYSGEN ステップを開始します。
   b. *Build Started With Parameters*　BUILD ステップを開始します。

c. *C:\WINCE600\Build.log* BUILDREL ステップを開始します。

d. *BLDDEMO: Calling Makeimg-Please Wait* MAKEIMG ステップを開始します。

3. Windows エクスプローラで、C:\Wince600 フォルダを開きます。Build.* ファイルが存在していることを確認します。

4. メモ帳などのテキスト エディタで Build.* ファイルを開き、コンテンツを確認します。

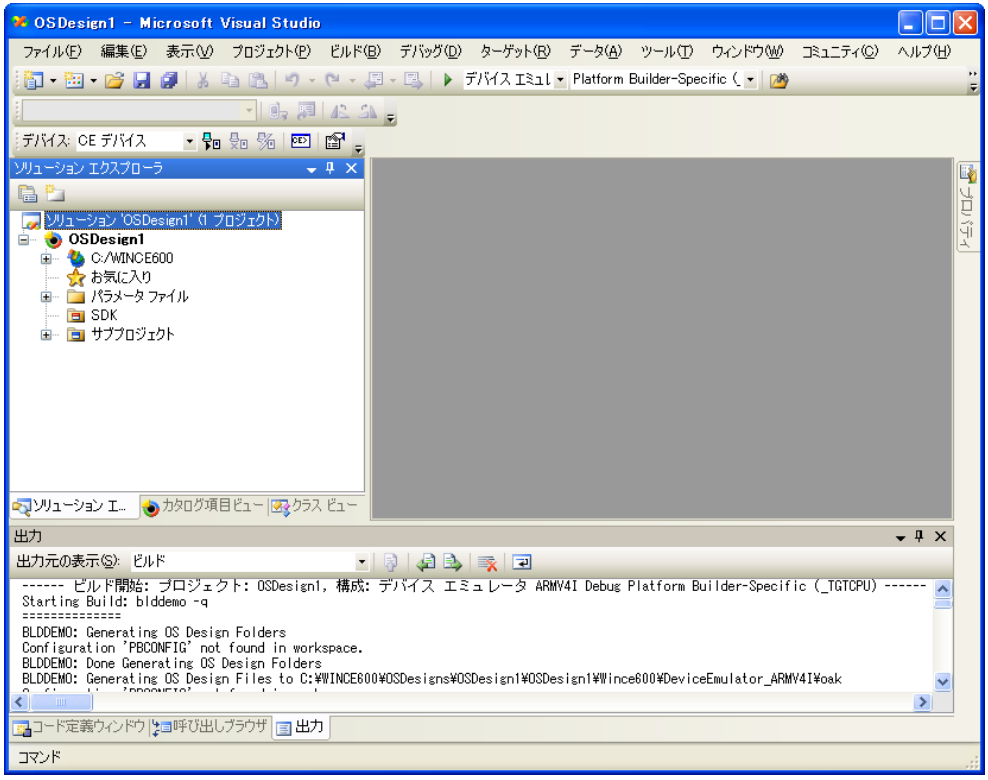


図 2-8 OS デザインのビルド

接続オプションの構成

1. Visual Studio で、[ターゲット] メニューから [接続オプション] をクリックして、[ターゲット デバイスの接続オプション] ダイアログ ボックスを表示します。

2. [デバイスの追加] をクリックし、[新しいターゲット デバイス名] ボックスに "CE Device" と入力し、[追加] をクリックします。[ターゲット デバイス] ボックスの一覧から [CE Device] をクリックします。
3. [ダウンロード] リスト ボックスから、[デバイス エミュレータ (DMA)] を選択します。
4. [トランスポート] リスト ボックスから、[デバイス エミュレータ (DMA)] を選択します。
5. [デバッガ] リスト ボックスから、図 2-9 に示すように、KdStub を選択します。

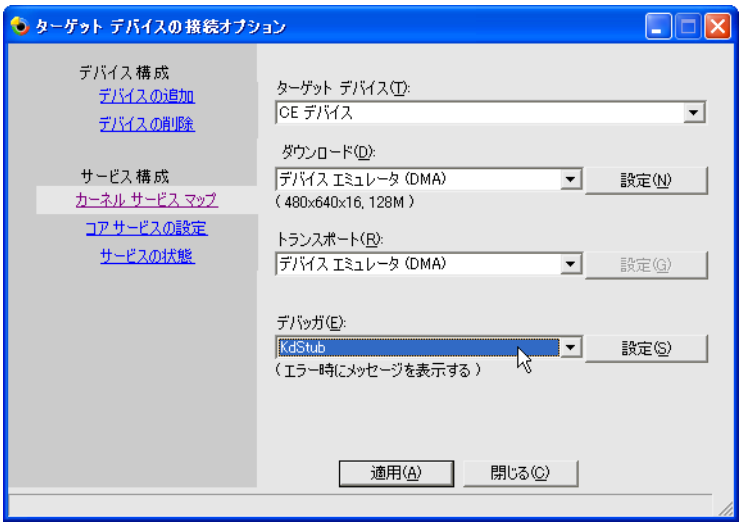


図 2-9　[ターゲット デバイスの接続オプション] の設定

エミュレータの構成の変更

1. [ダウンロード] ボックスのとなりの、[設定] をクリックします。
2. [エミュレータのプロパティ] ダイアログ ボックスで、[表示] タブをクリックします。
3. [画面の幅] を [640] ピクセルに、[画面の高さ] を [480] ピクセルに変更します。
4. [ネットワーク] タブをクリックします。
5. 図 2-10 に示すように、[NE2000 PCMCIA ネットワーク アダプタを有効にし、次の項目にバインドする] チェック ボックスを洗濯してから、リスト

ボックスから [接続されたネットワーク カード] オプションを選択し、[OK] をクリックします。

6. [適用] をクリックして、新しいデバイス設定を保存します。
7. [閉じる] をクリックして、[ターゲット デバイスの接続オプション] ボックスを閉じます。

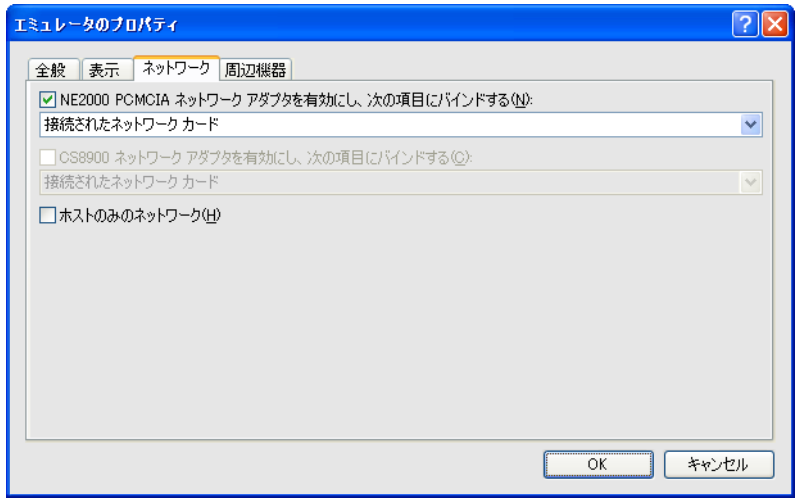


図 2-10 デバイス エミュレータ ネットワーク オプション

デバイス エミュレータでのランタイム イメージのテスト

1. Visual Studio で、[ターゲット] メニューから [デバイスの接続] をクリックします。
2. Visual Studio がランタイム イメージをターゲット デバイスにダウンロードすることを確認します。ダウンロードを完了するには、数分かかることがあります。
3. 開始プロセス中、Visual Studio の [出力] ウィンドウでデバッグ メッセージを目で追います。
4. 図 2-11 に示すように、Windows Embedded CE がブート プロセスを終了するまで待ち、デバイス エミュレータで抽出し、OS デザインの機能をテストします。

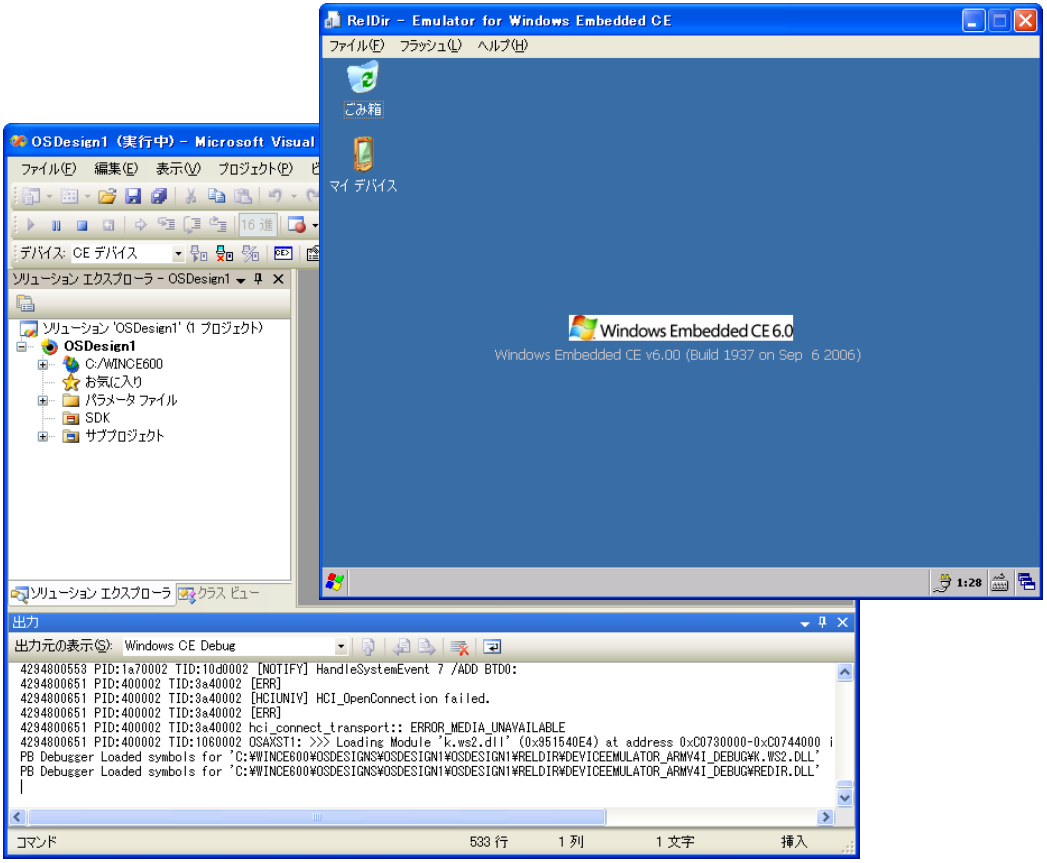


図 2-11　Windows Embedded CE デバイス エミュレータ

本章のレビュー

Windows Embedded CE ビルド プロセスには、いくつかのフェーズが含まれており、ビルドのタイプとランタイム イメージ構成ファイルに依存して、ソース コードのコンパイルとランタイム イメージの作成を行います。これには、BSP およびサブプロジェクト用の .exe ファイル、静的ライブラリ、DLL、およびバイナリ リソース (.res) ファイルを生成する [コンパイル] フェーズ、OS デザインに選択されたカタログ項目用にパブリック フォルダからの SYSGEN 変数に基づいてソース コードのフィルタおよびコピーを行って、一連のランタイム構成ファイルの作成行う [システム生成] フェーズ、ランタイム イメージのビルドに必要なファイルを BSP およびサブプロジェクトからのリリース ディレクトリにコピーする [リリース コピー] フェーズ、最後に、.bib、.reg、.db、および .dat ファイルに指定された設定に従って、リリース ディレクトリのコンテンツからランタイム イメージを作成する [ランタイム イメージの作成] フェーズが含まれます。

Platform Builder が生成し、追跡する Build.log、Build.wrn、および Build.err の情報を分析すると、ビルド プロセスを検証することができます。Build.log ファイルには、各ビルド フェーズで発行される詳細情報が含まれます。Build.wrn および Build.err には、同様の情報が含まれますが、ビルド プロセス中に発生した警告およびエラーがフィルタされています。これらのテキスト ファイルを直接メモ帳で開く必要はありません。Visual Studio でビルド ステータス情報およびエラー メッセージを扱うほうがより便利です。[出力] ウィンドウおよび [エラー一覧] ウィンドウは、便利なアクセスを提供しています。

ビルド エラーは、いろいろな理由で発生することがあります。最も一般的な理由は、コンパイラおよびリンカ エラーです。例えば、ビルド コマンドを不適切に初期化されたビルド環境で実行すると、無効な場所を指定した Sources ファイルのライブラリ ディレクトリを識別する環境変数で、リンカ エラーが発生します。Dirs ファイルおよび Makefile.def などの、他の重要なビルド構成ファイルは、条件式やディレクトリ パスの SYSGEN 変数や環境変数に依存させることもできます。

ランタイム イメージの生成に成功すると、ターゲット デバイスで Windows Embedded CE を展開できます。これには、[コア接続] インフラストラクチャに基づいたデバイス接続を構成する必要があります。最後の開発ステップは、Windows Embedded CE 6.0 R2 の Platform Builder を使用する Visual Studio の [ターゲット] メニューで、[デバイスの接続] コマンドをクリックするだけです。

Windows Embedded CE ビルド プロセスをコントロールする、次の構成ファイルに通じておくことは重要です。

- バイナリ イメージ ビルダ (.bib) ファイル　メモリ レイアウトを構成し、ランタイム イメージに含むファイルを決定します。
- レジストリ (.reg) ファイル　ターゲット デバイスでシステム レジストリを初期化します。
- データベース (.db) ファイル　既定のオブジェクト ストアを設定します。
- ファイル システム (.dat) ファイル　開始時に RAM ファイル システム レイアウトを初期化します。
- Dirs ファイル　ビルド プロセスを含めるディレクトリを決定します。
- ソース　ファイル　コンパイラおよびリンカの事前処理指示子、コマンド、マクロ、および他のプロセス命令を定義します。Platform Builder IDE を使用する Visual Studio の Makefile ファイルで行います。
- Makefile ファイル　既定の Makefile.def ファイルの参照、編集すべきではありません。

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。

- システム生成
- ビルド参照
- フラット リリース ディレクトリ
- 接続オプション
- KITL

推奨される練習

この章で提出されている試験目標を効果的にマスタできるよう、以下のタスクを完了してください。

コマンド ラインからビルド プロセスを起動

Windows Embedded CE ビルド プロセスの理解を深めるため、次のステップを実行します。

1. システム生成プロセス　カタログ項目の環境変数の設定後にコマンド ラインから sysgen -q を実行します。
2. ビルド プロセス　[コマンド プロンプト] ウィンドウを開き、現在の BSP フォルダ (%_TARGETPLATROOT%) に移動し、build-c を WINCEREL 環境変数を 1 に設定するパラメータ (set WINCEREL=1) を使用して、または使用せずに実行します。ビルドの前後に、%_FLATRELEASEDIR% フォルダのコンテンツを確認します。

ランタイム イメージの展開

Platform Builder のターゲット デバイス用の異なるダウンロード、トランスポート、およびデバッグ設定 Windows Embedded CE ランタイム イメージを [デバイス エミュレータ] に展開します。

パブリック カタログ コンポーネントを手動で複製する

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」で説明されているように、%_WINCEROOT%\Public フォルダからのコンポーネントを、ソース ファイルを BSP にコピーすることで複製します。次に、sysgen_capture を実行して、コンポーネントの依存関係を定義する Sources ファイルを作成します。新しい Sources ファイルを編集して、BSP の一部としてコンポーネントをビルドします。この詳細開発タスクを成功させるためのステップごとの詳細情報については、*http://msdn2.microsoft.com/en-us/library/aa924385.aspx* の Microsoft MSDN ィ ウェブサイトにある、Microsoft Windows Embedded CE 用 Platform Builder 製品ドキュメントの「Using the Sysgen Capture Tool」セクションを参照してください。

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。

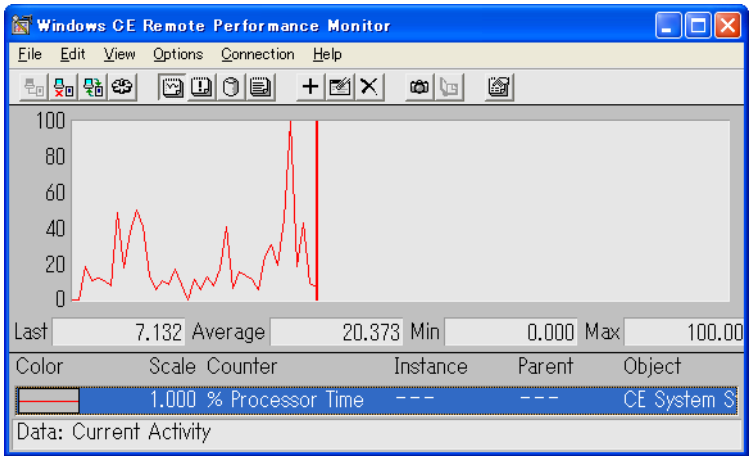


図 3ミ1 リモート パフォーマンス モニタのパフォーマンス チャート

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。

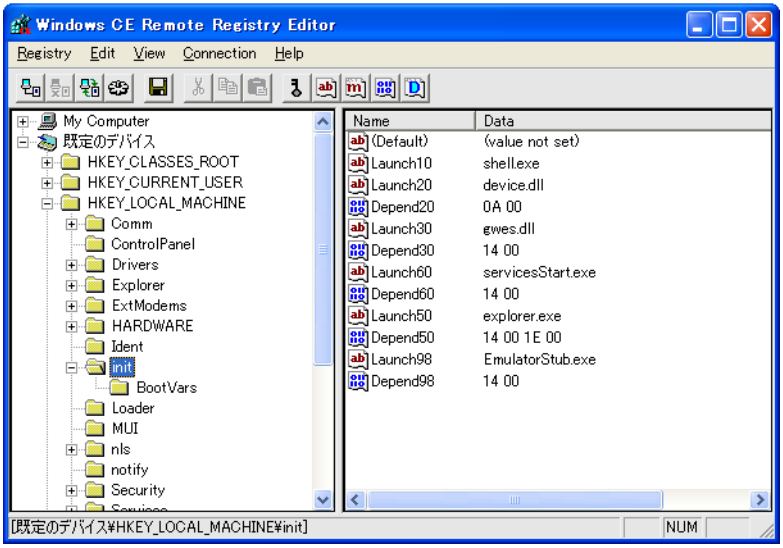


図 3ミ2 HKEY_LOCAL_MACHINE\INIT レジストリ キー

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

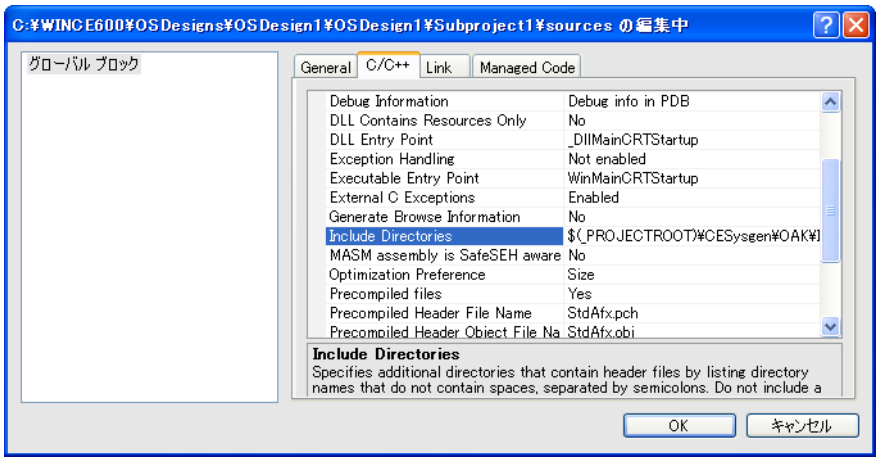


図 3-3 コントロール パネル アプレットのインクルード ディレクトリ

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。

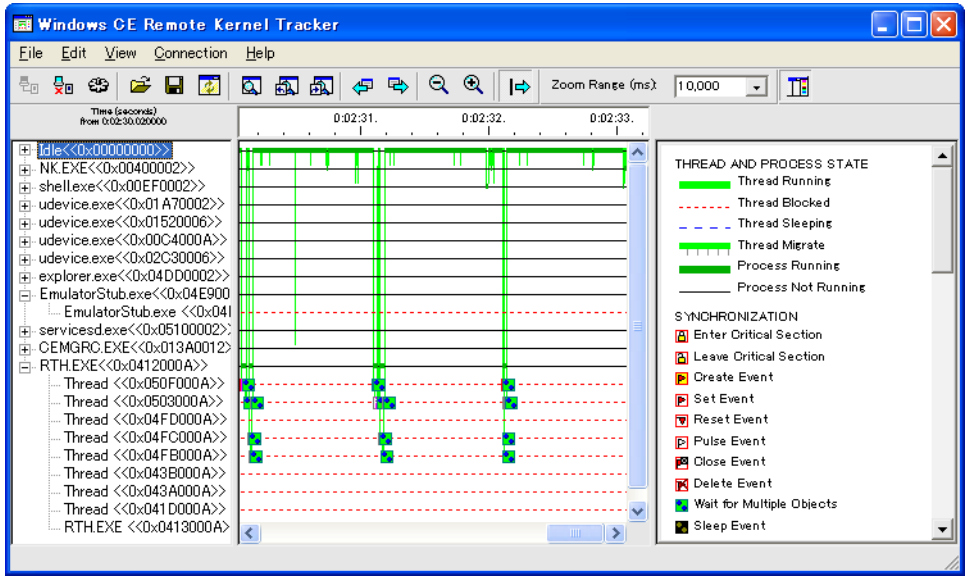


図 3ミ4　Remote Kernel Tracker ツール

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。

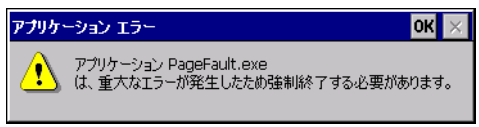


図 3ミ5　処理されていないページ フォールトの例外。ユーザーの視点から

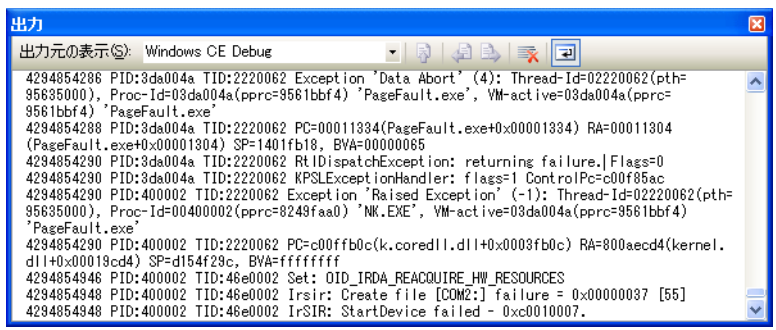


図 3ミ6　Visual Studio 2005 における KITL に関する未処理ページ フォールト例外のデバッグ出力

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。

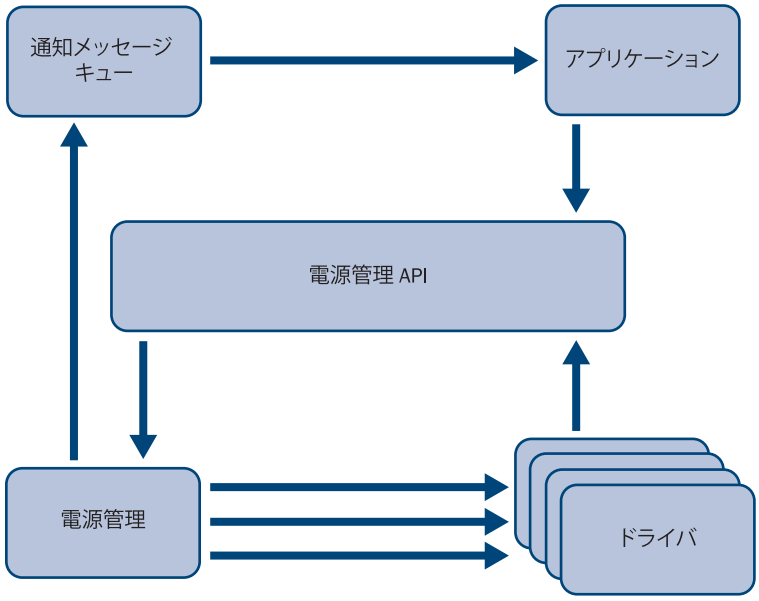


図 3ミ7　Power Manager と Power Management の インタラクション

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。

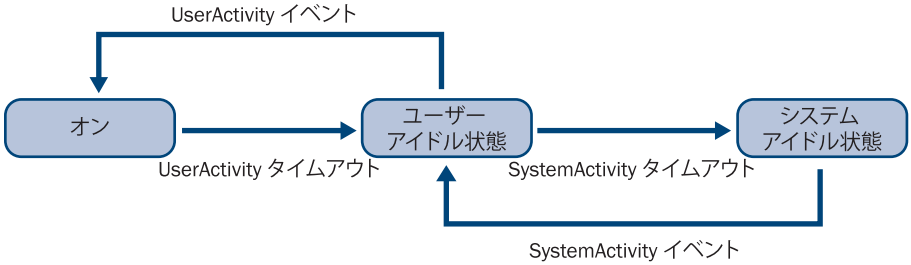


図 3ミ9　アクティビティ タイマ、イベント、およびシステム電源状態移行

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嚀わる ■ PBT_POWERINFOCHANGE　バッテリのステ　タスが嚀わる。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。

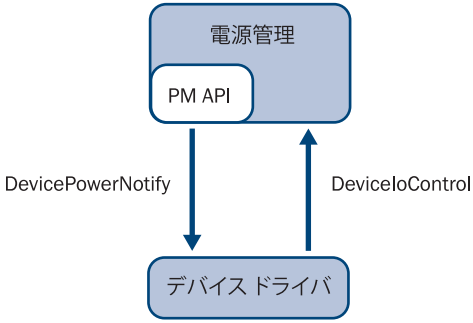


図 3ミ10　Power Manager およびデバイスドライバインタラクション

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。

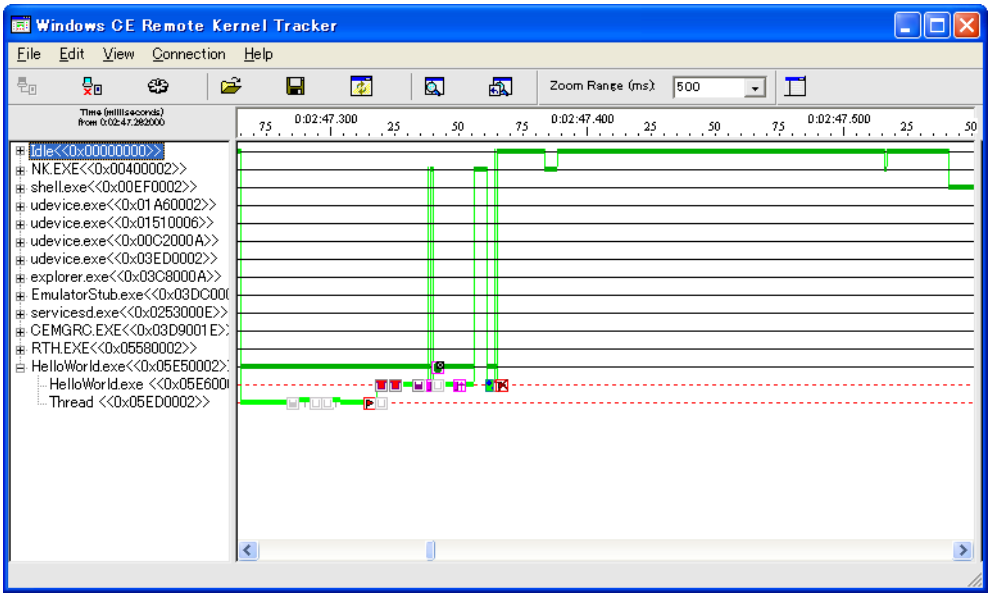


図 3 ミ 11　Remote Kernel Tracker ツールでスレッド実行をトラッキング

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。

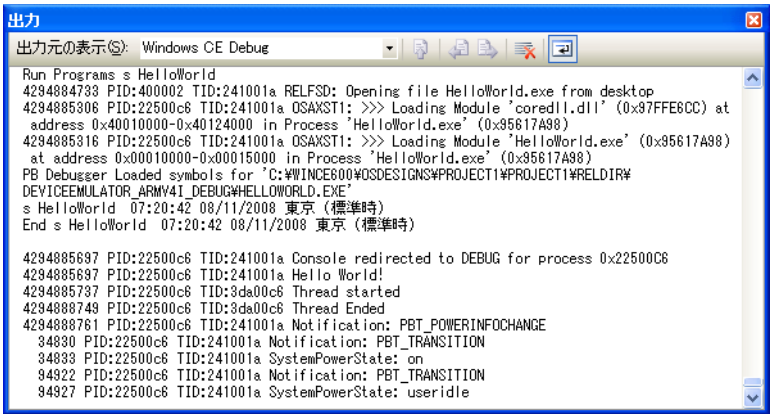


図 3ミ12　受領された電源管理通知

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。

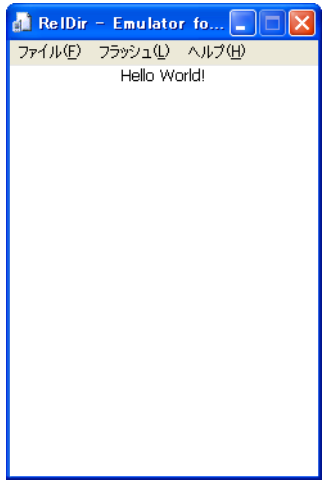


図 3 ミ 13　標準のシェルを Subproject_Shell アプリケーションに置き換え

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。

第 4 章

システムのデバッグおよびテスト

デバッグおよびシステム テストは、ソフトウェア開発サイクルにおける重要なタスクで、ターゲット デバイス上のソフトウェア関連およびハードウェア関連の欠陥を特定し解決するという最も重要な目標があります。一般的に、デバックとはコードをステップごとに実行するプロセスを指し、エラーの根本原因を診断するためにコードの実行中に発生したデバッグ メッセージを分析します。また、これは一般的なシステム コンポーネントおよびアプリケーションの実装を学ぶための効果的なツールです。一方、システム テストは品質保証アクティビティで、一般的な使用シナリオ、性能、信頼性、セキュリティおよびその他の関連要素に関して、最終構成におけるシステムの検証を行います。システム テストの全般的な目的は、メモリ リーク、デッドロック、またはハードウェア衝突などの製品の欠陥や障害を検出することですが、デバッグはこれらの問題の原因を特定し、それらを除去することです。小さなフットプリントのデバイスやコンシューマー機器の開発者の多くにとって、システム障害の特定と除去は、ソフトウェア開発のもっとも困難なプロセスで、生産性にかなり重要な影響を与えます。この章では、障害と特定と除去の自動化と高速化して、バグを低減し、システムのリリースを早めるのに役立つ Microsoft ィ Visual Studio ィ 2005 と Microsoft Windows ィ Embedded CE 6.0 R2 用 Platform Builder、および Windows Embedded CE Test Kit (CETK) で使用可能なデバッグおよびテスト ツールを取り上げます。これらのコードにより習熟すると、コードの修正ではなく、コードの記述により多くの時間を割けるようになります。

本章の試験範囲：

- ランタイム イメージのデバッグ用の要件を識別する
- デバッガ機能を使用してコード実行を分析する
- デバッグ領域を理解し、デバッグ メッセージの出力を管理する
- CETK ツールを使用して、既定およびユーザー定義のテストを実行する
- ブート ローダーおよびオペレーティング システム (OS) をデバッグする

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。

- Windows Embedded CE ソフトウェア開発およびデバッグ概念に関する基本的な知識。
- Windows Embedded CE でサポートされるドライバ アーキテクチャに関する基本的な理解。
- OS デザインおよびシステム構成概念の熟知。
- Microsoft Visual Studio 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：ソフトウェア関連のエラーの検出

ソフトウェア関連エラーの範囲には、タイプミス、初期化されてない変数、無限ループなどの単純なものから、重大な競合条件および他のスレッド同期の問題などの、より複雑で難解なものがあります。幸いなことに、ほとんどのエラーは、検出してから簡単に修正できます。これらのエラーを特定する最もコスト効率の良い方法は、コード分析を行うことです。Windows Embedded CE デバイス上で多様なツールを使用して、オペレーティング システムのデバッグおよびドライバおよびアプリケーションをステップごとに確認できます。これらのデバッグ ツールをよく理解しておくと、コード分析を高速化でき、ソフトウェア エラーの修正をできる限り効率的に行うことができます。

このレッスンを終了すると、以下をマスターできます：

- Windows Embedded CE 用の重要なデバッグ ツールを識別。
- ドライバやアプリケーションにおいて、デバッグ領域を介してデバッグ メッセージをコントロール。
- ターゲット コントロール シェルを使用してメモリの問題を識別。

レッスン時間 (推定)：90 分

デバッグとターゲット デバイス コントロール

Windows Embedded CE ターゲット デバイスをデバッグおよびコントロールする主要なツールは Platform Builder で、図 4-1 で図示されているように、開発ワークステーション上で実行します。Platform Builder 統合開発環境 (IDE) には、これを目的とした多様なツールが含まれており、システム デバッガ、CE ターゲット コントロール シェル (CESH)、およびデバッグ メッセージ (DbgMsg) 機能が含まれ、ブレークポイントに達した後にコードのステップ実行を行ったり、メモリ、変数、およびプロセスに関する情報を表示したりできます。さらに、Platform Builder IDE には、リモート ツール群も含まれており、これには Heap Walker、Process Viewer、および Kernel Tracker などが含まれ、ランタイム時にターゲット デバイスの状態を分析できます。

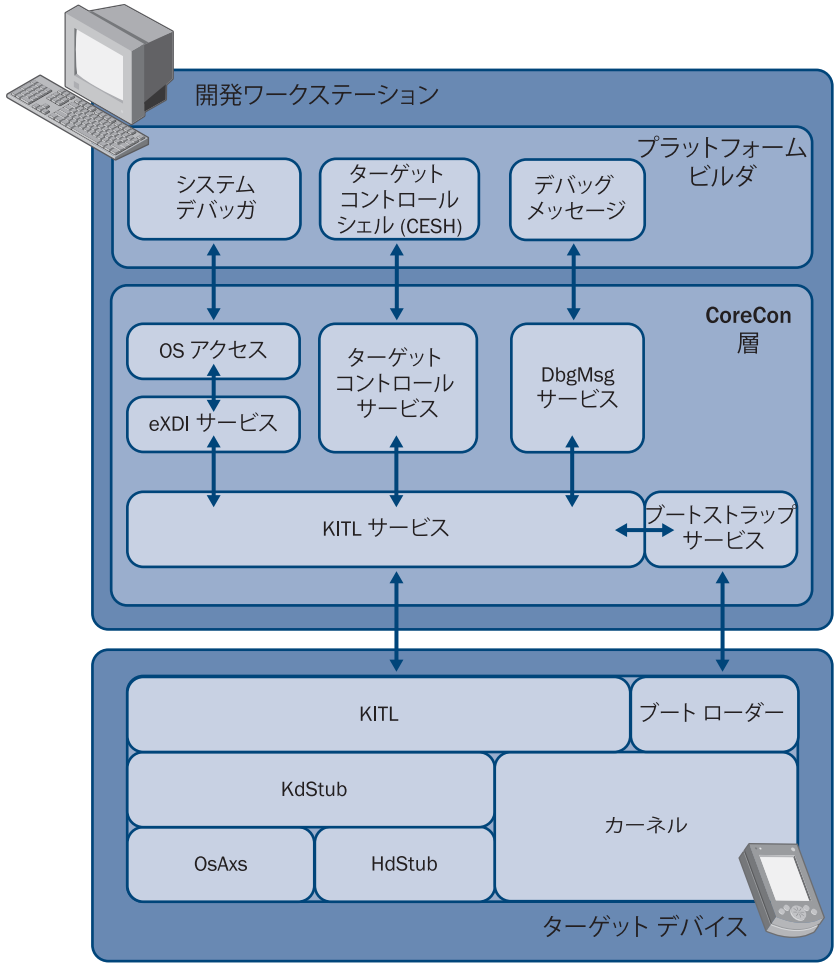


図 4-1　CE デバッグおよびターゲット コントロール アーキテクチャ

ターゲット デバイスと通信するには、Platform Builder は、ランタイム イメージの一部としてターゲット デバイスに展開された Core Connectivity (CoreCon) インフラストラクチャおよびデバッグ コンポーネントに依存しています。CoreCon インフラストラクチャは、OS Access (OsAxS)、ターゲット コントロール、および DbgMsg サービスを Platform Builder の一方に提供し、Kernel Independent Transport Layer (KITL) およびブートストラップ サービスを介したターゲット デバイスとのインターフェイスを他方に提供します。ターゲット デバイス自体では、デバッグおよびターゲット コントロール アーキテクチャは、KITL および通信目的にブート ローダーに依存しています。ランタイム イメージに、Kernel Debugger スタブ (KdStub)、ハードウェア デバッガ スタブ (HdStub)、

および OsAxS ライブラリなどのデバッグ コンポーネントが含まれている場合は、Platform Builder を使用してカーネル ランタイム情報を取得し、ジャストインタイム (JIT) デバッガを実行します。Platform Builder は、Extended Debugging Interface (eXDI) を介したハードウェア補助デバッグもサポートしており、カーネルをロードする前にターゲット デバイスを日常的にデバッグできます。

カーネル デバッガ

カーネル デバッガは、CE ソフトウェア デバッグ エンジンで、カーネル コンポーネントと CE アプリケーションをデバッグします。開発ワークステーションでは、直接 Platform Builder で作業し、ソースコードでのブレークポイントの挿入と削除およびアプリケーションの実行などを行うことができますが、ランタイム イメージの KITL およびデバッグ ライブラリ (KdStub および OsAxS) のサポートを含めることで、Platform Builder がデバッグ情報を取得しターゲット デバイスをコントロールできるようにしておく必要があります。この章の、後続のレッスン 2 「ランタイム イメージを構成してデバッグを有効にする」では、カーネル デバッグのシステム構成に関する詳細な情報を提供します。

次のターゲット側コンポーネントは、カーネル デバッグに不可欠です。

- KdStub　例外およびブレークポイントの収集、カーネル情報の取得、およびカーネル操作の実行を行います。
- OsAxS　メモリ割り当て、アクティブなプロセスおよびスレッド、プロキシ、およびロードされたダイナミック リンクライブラリ (DLL) に関する情報など、オペレーティング システムの状態に関する情報を取得します。

ノート　Windows Embedded CE でのアプリケーションのデバッグ

カーネル デバッガを使用することで、個別アプリケーションだけでなく、ランタイム イメージ全体をコントロールできます。ただし、KdStub は、ファーストチャンス例外およびセカンドチャンス例外を受け取るカーネルコンポーネントです。第 3 章「システム プログラミングの実行」で説明されています。ターゲット側の KdStub モジュールを最初に停止させずに、セッション中でカーネル デバッガを停止させた状態で例外が発生すると、カーネル デバッガが例外を処理してターゲット デバイスを継続して実行できるようにするため、デバッガが再接続されるまでランタイム イメージは応答を停止します。

デバッグ メッセージ サービス

Platform Builder では、KITL (が有効、および KdStub) が有効なターゲット デバイスに接続すると、Microsoft Visual Studio 2005 の出力ウィンドウでデバッグ情報を検査することができます。デバッグ情報は、Platform Builder が CoreCon

インフラストラクチャで DbgMsg サービスを使用することでターゲット デバイスから取得します。

デバッグ メッセージは、実行中のプロセス、無効な入力などの信号の潜在的に重要な問題に関する詳細な情報を提供し、コード中の障害のある場所に関するヒントも提供します。このヒントを使用して、ブレークポイントを設定したり、カーネル デバッガでコードをステップ実行したりしてさらに調査できます。カーネル デバッガ スタブの機能の 1 つは、デバッグ メッセージの動的管理のサポートです。これにより、ソース コードを修正することなく、デバッグ詳細を構成できます。他にも、Visual Studio の [ターゲット] メニューからアクセスできる [デバッグ メッセージ オプション] を表示している場合、タイムスタンプ、プロセス ID、またはスレッド ID を除外できます。また、別のツールでの分析のためにデバッグ出力をファイルに送信することもできます。ターゲット デバイスでは、デバッグ メッセージはすべて、NKDbgPrintf 関数 を介して処理される既定の出力ストリームに直接送信できます。

ノート　KITL あり、および KITL なしのデバッグ メッセージ

カーネル デバッガおよび KITL の両方が有効な場合、デバッグ メッセージは Visual Studio の出力ウィンドウに表示されます。KITL が使用可能でない場合、デバック情報は、構成されたシリアルポートを介してターゲット デバイスから開発コンピュータに転送され、OEM アダプテーション層 (OAL) によって使用されます。

デバッグ メッセージ用のマクロ

デバッグ情報を生成するため、Windows Embedded CE は、一般的にデバッグ マクロおよびリテール マクロの 2 つのカテゴリに分類されるいくつかのデバッグ マクロを提供します。デバッグ マクロは、デバッグ ビルド構成 (環境変数 WINCEDEBUG=debug) でコードがコンパイルされている場合に、情報を出力します。それに対し、リテール マクロは、シップ構成 (WINCESHIP=1) でランタイム イメージがビルドされていない限り、デバッグ ビルド構成およびリテール ビルド構成 (WINCEDEBUG=retail) の両方で 情報が生成されます。シップ構成では、すべてのデバッグ マクロが無効になります。

表 4-1 は、コードに挿入してデバッグ情報を生成できる、デバッグ マクロを要約しています。

表 4-1 デバッグ メッセージを出力する Windows Embedded CE マクロ

マクロ	説明
DEBUGMSG	ランタイム イメージがデバッグ ビルド構成でコンパイルされた場合は、条件付きで printf スタイルのデバッグ メッセージを既定の出力ストリーム (つまり、Visual Studio の出力ウィンドウまたは指定されたファイル) に出力します。
RETAILMSG	ランタイム イメージが、シップ ビルド構成ではなく、デバッグまたはリリース ビルド構成でコンパイルされた場合は、条件付きで printf スタイルのデバッグ メッセージを既定の出力ストリーム (つまり、Visual Studio の出力ウィンドウまたは指定されたファイル) に出力します。
ERRORMSG	ランタイム イメージが、シップ ビルド構成ではなく、デバッグまたはリリース ビルド構成でコンパイルされた場合は、条件付きで付加的な printf スタイルのデバッグ情報を既定の出力ストリーム (つまり、Visual Studio の出力ウィンドウまたは指定されたファイル) に出力します。このエラー情報には、ソース コード ファイルの名前、行番号が含まれ、メッセージを生成したコードの行をすぐに特定するのに役立ちます。
ASSERTMSG	ランタイム イメージがデバッグ ビルド構成でコンパイルされた場合は、条件付きで printf スタイルのデバッグ メッセージを既定の出力ストリーム (つまり、Visual Studio の出力ウィンドウまたは指定されたファイル) に出力してから、デバッガに割り込みます。実際のところ、ASSERTMSG は DEBUGMSG を呼び出した後に DBGCHK を呼び出します。
DEBUGLED	ランタイム イメージがデバッグ ビルド構成でコンパイルされた場合は、条件付きで WORD 値を WriteDebugLED 関数に渡します。このマクロは、発光ダイオード (LED) のみに出力してシステム ステータスを示すデバイスに役立ちます。OAL で OEMWriteDebugLED 関数 を実装する必要があります。
RETAILLED	ランタイム イメージがデバッグまたはリリース ビルド構成でコンパイルされた場合は、条件付きで WORD 値を WriteDebugLED 関数に渡します。このマクロは、LED のみに出力してシステム ステータスを示すデバイスに役立ちます。OAL で OEMWriteDebugLED 関数を実装する必要があります。

デバッグ領域

デバッグ メッセージは､マルチスレッド プロセスの分析に特に便利なツールで、とりわけ、コードのステップ実行では検出しにくい、同期や他のタイミングの

問題の分析に適しています。ただし、コード内でデバッグ マクロを使いすぎると、ターゲット デバイス上で生成されるデバッグ メッセージの数が多くなりすぎて処理できなくなることがあります。生成される情報の量をコントロールするため、デバッグ マクロで条件式を指定できるようになっています。例えば、次のコードは、dwCurrentIteration 値が最大許容値よりも大きい場合に、エラーメッセージを出力します。

```
ERRORMSG(dwCurrentIteration > dwMaxIteration,
         (TEXT("Iteration error: the counter reached %u, when max allowed is %u\r\n"),
          dwCurrentIteration, dwMaxIteration));
```

上記の例では、ERRORMSG は、dwCurrentIteration が dwMaxIteration より大きい場合にはいつでもデバッグ情報を出力しますが、条件式でデバッグ領域を使用することでデバッグ メッセージをコントロールすることができます。これは、DEBUGMSG マクロを使用して、ソース コードを毎回変更して再コンパイルすることなく、モジュール (つまり、実行可能ファイルや DLL) のコード実行を異なるレベルで試験したい場合に特に役立ちます。最初に、実行可能ファイルまたは DLL でデバッグ領域を有効にし、グローバル DBGPARAM 変数をデバッグ メッセージ サービスに登録してどの領域を有効にするか指定する必要があります。すると、プログラムに従って、または開発ワークステーションやターゲット デバイスのレジストリ設定によって、現在の既定の領域を指定することができます。また、Platform Builder の [ターゲット] メニューまたは [ターゲット コントロール] ウィンドウにある [CE デバッグ領域] から、モジュールのデバッグ領域を動的にコントロールすることもできます。

ヒント　デバッグ領域のバイパス

ランタイム イメージのリビルド時に TRUE または FALSE に設定可能なブール変数を DEBUGMSG および RETAILMSG マクロに渡している場合、ドライバおよびアプリケーションでデバッグ領域をバイパスできます。

領域登録

デバッグ領域を使用するには、3 つのフィールドのあるグローバル DBGPARAM 変数を定義する必要があります。表 4-2 で要約されているように、これらのフィールドで、モジュール名、登録したいデバッグ領域の名前、および現在の領域マスクのフィールドを指定します。

表 4-2　DBGPARAM 要素

フィールド	説明	例
lpszName	モジュールの名前を、最大 32 文字で定義します。	`TEXT("ModuleName")`

rglpszZones	デバッグ領域の 16 個の名前の配列を定義します。それぞれの名前は、最大 32 文字の長さにすることができます。Platform Builder は、モジュールで有効な領域を選択するときに、ユーザーにこの情報を表示します。	`{`
		`TEXT("Init"),`
		`TEXT("Deinit"),`
		`TEXT("On"),`
		`TEXT("Undefined"),`
		`TEXT("Undefined"),`
		`TEXT("Undefined"),`
		`TEXT("Undefined"),`
		`TEXT("Undefined"),`
		`TEXT("Undefined"),`
		`TEXT("Undefined"),`
		`TEXT("Undefined"),`
		`TEXT("Undefined"),`
		`TEXT("Failure"),`
		`TEXT("Warning"),`
		`TEXT("Error")`
		`}`

ulZoneMask	DEBUGZONE マクロで使用されている現在の領域マスクにより、現在選択されているデバッグ領域を定義します。	`MASK_INIT | MASK_ON | MASK_ERROR`

ノート　デバッグ領域

Windows Embedded CE は、合計 16 個の名前付けされたデバッグ領域をサポートしていますが、モジュールで必要なければすべてを定義する必要はありません。各モジュールには、実装されている各領域の目的を明確に表す、別個の一連の領域名を使用します。

Dbgapi.h ヘッダー ファイルは、DBGPARAM 構造体およびデバッグ マクロを定義します。これらのマクロは、dpCurSettings と名前付けされた事前定義された DBGPARAM 変数を使用するため、次のコード スニペットに示すように、ユーザーのソース コードでも同じ名前を使用することは重要です。

```
#include <DBGAPI.H>

// 領域マスク定義の読みやすさを向上するマクロ
#define DEBUGMASK(n)  (0x00000001<<n)
```

```
// このモジュールでサポートされる領域マスクの定義
#define MASK_INIT      DEBUGMASK(0)
#define MASK_DEINIT    DEBUGMASK(1)
#define MASK_ON        DEBUGMASK(2)
#define MASK_FAILURE   DEBUGMASK(13)
#define MASK_WARNING   DEBUGMASK(14)
#define MASK_ERROR         DEBUGMASK(15)

// 失敗、警告、およびエラーに設定される初期デバッグ領域状態
// のある dpCurSettings 変数
DBGPARAM dpCurSettings =
{
  TEXT("ModuleName"), // 明快にする ?¾ め実際のモ ÉWÉÖール名を éwíË
 {
   TEXT("Init"), TEXT("Deinit"), TEXT("On"), TEXT("Undefined"),
    TEXT("Undefined"), TEXT("Undefined"), TEXT("Undefined"),
    TEXT("Undefined"), TEXT("Undefined"), TEXT("Undefined"),
    TEXT("Undefined"), TEXT("Undefined"), TEXT("Undefined"),
    TEXT("Failure"), TEXT("Warning"), TEXT("Error")
  },
  MASK_INIT | MASK_ON | MASK_ERROR
};

// DLL へのメイン エントリ ポイント
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    if(ul_reason_for_call == DLL_PROCESS_ATTACH)
    {
// DLL がプロセスのアドレス領域にロードされるたびに
    // デバッグ メッセージ サービスに登録
    DEBUGREGISTER((HMODULE)hModule);
  }
  return TRUE;
}
```

領域定義

上記のサンプル コードでは、モジュールに 6 つのデバッグ領域を登録しており、これで、デバッグ領域をデバッグ マクロの条件式と併用することができます。次のコードの行は、これを行う方法の 1 つを示しています。

```
DEBUGMSG(dpCurSettings.ulZoneMask & (0x00000001<<(15)),
        (TEXT("Error Information\r\n")));
```

デバッグ領域が現在 MASK_ERROR に設定されていると、条件式は TRUE を出力し、DEBUGMSG は情報をデバッグ出力ストリームに送信します。ただし、コードの読みやすさを向上するため、次のコード スニペットで示すように、Dbgapi.h で定義される DEBUGZONE マクロを使用して、領域のフラグを定義する必要があります。他にも、このアプローチでは、論理 AND および OR 演算子によって、デバッグ領域の組み合わせを簡単化できます。

```
#include <DBGAPI.H>

// 領域フラグの定義：選択されたデバッグ領域によって、TRUE または FALSE
#define ZONE_INIT        DEBUGZONE(0)
#define ZONE_DEINIT      DEBUGZONE(1)
#define ZONE_ON          DEBUGZONE(2)
#define ZONE_FAILURE     DEBUGZONE(13)
#define ZONE_WARNING     DEBUGZONE(14)
#define ZONE_ERROR       DEBUGZONE(15)

DEBUGMSG(ZONE_FAILURE, (TEXT("Failure debug zone enabled.\r\n")));
DEBUGMSG(ZONE_FAILURE && ZONE_ WARNING,
        (TEXT("Failure and Warning debug zones enabled.\r\n")));
DEBUGMSG(ZONE_FAILURE || ZONE_ ERROR,
        (TEXT("Failure or Error debug zone enabled.\r\n")));
```

デバッグ領域の有効化と無効化

DBGPARAM フィールド ulZoneMask は、モジュールの現在のデバッグ領域設定において重要です。これは、グローバル dpCurSettings 変数の ulZoneMask 値を直接変更することにより、モジュールで計画的に実現できます。別の方法としては、[ウォッチ] ウィンドウ内のブレークポイントで、デバッガの ulZoneMask 値を変更する方法があります。SetDbgZone 関数を呼び出すことで、他のアプリケーションからデバッグ領域をコントロールすることもできます。ランタイム時に有効な方法としては、図 4-2 に示すように、[デバッグ領域] ダイアログ ボックスを使用する方法です。このダイアログ ボックスは、Platform Builder を使用して [ターゲット] メニューの [CE デバッグ領域] コマンドから Visual Studio で表示できます。

図 4-2　Platform Builder でデバッグ領域の設定

[名前] リストは、デバッグ領域をサポートするターゲット デバイス上で実行されているモジュールを表示します。選択されたモジュールがデバッグ メッセージ サービスに登録されていると、[デバッグ領域] の下に表示される 16 個の領域のリストを確認できます。それらの名前は、選択されたモジュールの dpCurSettings 定義に対応しています。領域を選択または選択解除して、モジュールを選択または選択解除できます。既定では、dpCurSettings 変数で定義された領域は、[デバッグ領域] リストで有効およびチェックされています。デバッグ メッセージ サービスに登録されていないモジュールについては、[デバッグ領域] リストは無効であり使用できません。

起動時にデバッグ領域のオーバーライド

アプリケーションを起動するか、DLL をプロセスにロードしたときに、Windows Embedded CE は、dpCurSettings 変数で指定した領域を有効にします。この時点では、ブレークポイントを設定して、[ウォッチ] ウィンドウで ulZoneMask 値を変更するまで、デバッグ領域を変更することができません。ただし、CE はレジストリ設定を使用する、さらに便利な方法をサポートしています。異なるアクティブなデバッグ領域を使用してモジュールを容易にロードするには、dpCurSettings 変数の lpszName フィールドに指定されたモジュール名に対応する名前で REG_DWORD 値を作成し、アクティブにしたいデバッグ領域の複合値に設定します。この値は、開発ワークステーションまたはターゲット デバイスで構成できます。開発ワークステーションでこの値を構成するのが一般的に望ましいです。ターゲット デバイス レジストリ エントリを変更するとランタイム

イメージの再ビルドが必要ですが、開発ワークステーション上のレジストリ エントリの変更は、関係するモジュールを再起動するだけで済みます。

表 4 ミ 3 は、*ModuleName* と呼ばれるサンプル モジュールの構成を示しています。このプレースホルダ名を、実行可能ファイルか DLL の実際の名前に置き換えていることを確認してください。

表 4-3 スタートアップ レジストリ パラメータ例

場所	開発ワークステーション	ターゲット デバイス
レジストリ キー	HKEY_CURRENT_USER \Pegasus\Zones	HKEY_LOCAL_MACHINE \DebugZones
エントリ名	ModuleName	ModuleName
タイプ	REG_DWORD	REG_DWORD
値	0x00000001 - 0x7FFFFFFF	0x00000001 - 0x7FFFFFFF
コメント	デバッグ メッセージ システムは、開発ワークステーションが使用できないか、開発側のレジストリにモジュールの値が含まれていない場合のみ、モジュールにターゲット側の値を使用します。	

ノート　すべてのデバッグ領域の有効化

Windows Embedded CE は、REG_DWORD 値の下位 16 ビットを使用して、アプリケーションのデバッグを目的として名前付けされたデバッグ領域を定義します。カーネル用に取って置かれている最高位ビットを除いて、残りのビットは、名前指定していないデバッグ領域を定義するために使用可能です。そのため、モジュールのデバッグ領域値を 0xFFFFFFFF にすべきではありません。最大値は 0x7FFFFFFF で、名前指定されたデバッグ領域と名前指定されていないデバッグ領域をすべて有効にできます。

詳細情報　ペガサス レジストリ キー

ペガサスという名前は、Microsoft がパーソナル コンピュータおよびその他家庭用電化製品向けに 1996 年にリリースした最初の Windows CE のコードネームです。

ベスト プラクティス

デバッグ メッセージを取り扱うとき、デバッグ メッセージを使いすぎるとコード実行が遅くなることに留意してください。さらに重要なこととして、システ

ムは、不意のスレッド同期機構を提供することのある、デバッグ出力操作を直列化します。例えば、リリース ビルドで複数のスレッドが同期化されずに実行していると、デバッグ ビルドでは結果として発生する問題は顕著なものとはなりません。

デバッグ メッセージおよびデバッグ領域を扱うときは、次のベスト プラクティスを考慮してください。

- 条件式　デバッグ領域に基づいて、条件式とともにデバッグ マクロを使用します。DEBUGMSG(TRUE) は使用しないでください。また、モデル デバイス ドライバ (MDD) / プラットフォーム依存ドライバ (PDD) の中には、RETAILMSG(TRUE) などのリテール マクロを条件式なしで使用する手法がありますが、この場合は使用しないでください。
- リリース ビルドからデバッグ コードを除外する　デバッグ ビルドでデバッグ領域のみを使用する場合、グローバル変数 dpCurSettings およびゾーン マスク定義を #ifdef DEBUG #endif 保護に含め、デバッグ領域の使用をデバッグ マクロ (DEBUGMSG など) に限定する必要があります。
- リリース ビルドでリテール マクロを使用する　リリース ビルドでもデバッグ領域を使用したい場合、グローバル変数 dpCurSettings およびゾーン マスク定義を #ifndef SHIP_BUILD #endif 保護に含め、DEBUGREGISTER への呼び出しを RETAILREGISTERZONES への呼び出しに置き換える必要があります。
- モジュール名を明快に識別する　可能な場合には、dpCurSettings.lpszName 値をモジュールのファイル名に設定します。
- 既定で詳細度を制限する　ドライバの既定領域を ZONE_ERROR および ZONE_WARNING のみに設定します。新しいプラットフォームを構築するときは、ZONE_INIT を有効にします。
- エラー デバッグ領域を回復不能な問題に制限する　モジュールまたは重要な機能が、不正確な構成または他の問題のために失敗した場合のみ ZONE_ERROR を使用します。回復可能な問題には、ZONE_WARNING を使用します。
- 可能な限りエラーおよび警告を除去する　モジュールは、ZONE_ERROR または ZONE_WARNING メッセージなしでロードできる必要があります。

ターゲット コントロール コマンド

[ターゲット コントロール] サービスは、デバッガのコマンド シェルへのアクセスを提供し、ファイルをターゲット デバイスやデバッグ アプリケーションに転送します。図 4-3 に表示されている、このターゲット コントロール シェルは、Visual Studio 内で Platform Builder を使用して、[ターゲット] メニューの [ターゲット コントロール] オプションを介してアクセスできます。ただし、ターゲット コントロール シェルは、Platform Builder インスタンスが KITL を介してデバイスに接続されている場合のみ使用可能です。

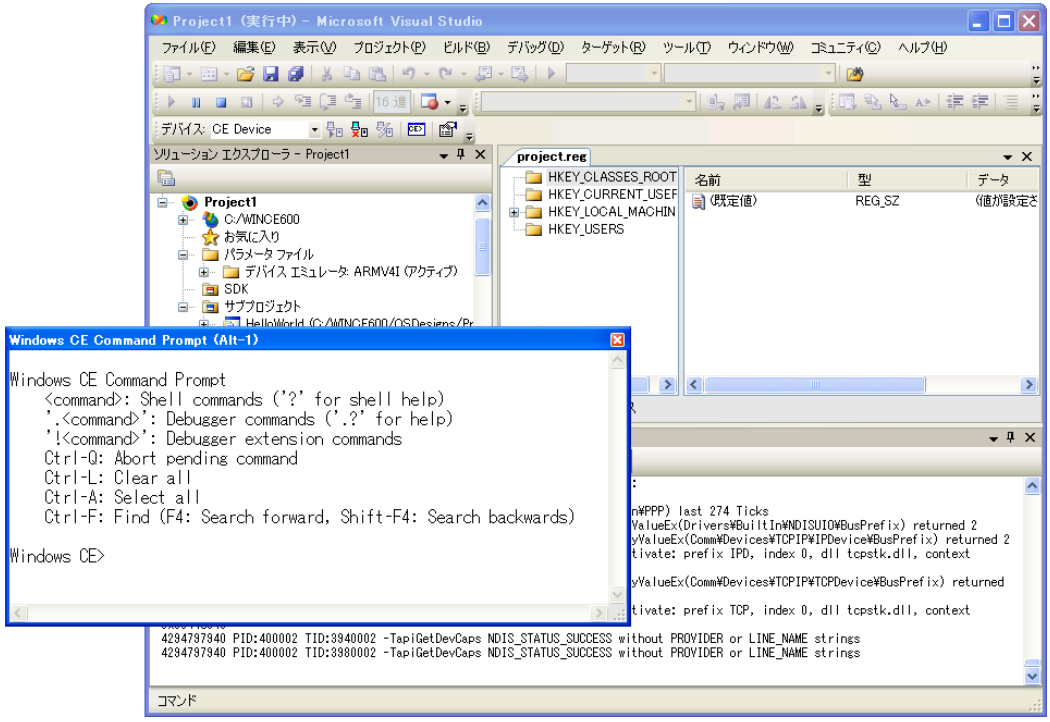


図 4-3　ターゲット コントロール シェル

その他にも、ターゲット コントロール シェルは、次のデバッグ操作を実行します。

- カーネル デバッガへの割り込み (break コマンド)。
- メモリ ダンプをデバッグ出力 (dd コマンド) またはファイル (df コマンド) に送信。

- カーネル (mi kernel コマンド) またはシステム全体 (mi full コマンド) のメモリ使用率を分析。
- プロセス (gi proc コマンド)、スレッド (gi thrd コマンド)、スレッド プロパティ (tp コマンド)、およびシステムにロードされているモジュール (gi mod コマンド) のリスト表示。
- プロセスの起動 (s コマンド) およびプロセスの終了 (kp コマンド)。
- プロセス ヒープのダンプ (hp コマンド)。
- システム プロファイラの有効化および無効化 (prof コマンド)。

ノート　ターゲット コントロール コマンド

ターゲット コントロール コマンドの完全なリストについては、http://msdn2.microsoft.com/en-us/library/aa936032.aspx の Microsoft MSDN ィ Web サイトにある、Windows Embedded CE 6.0 ドキュメントの「Target Control Debugging Commands」セクションを参照してください。

デバッガ拡張コマンド (CEDebugX)

通常のデバッガ コマンドに加えて、[ターゲット コントロール] サービスは、コマンド拡張 (CEDebugX) を使用したデバッガを提供しており、カーネルとアプリケーションのデバッグの効率を向上します。この拡張は、メモリ リークおよびデッドロックを検出したり、システムの全体的な健全性を診断したりするための付加的な機能を提供します。付加的なコマンドは、ターゲット コントロール シェルを介してアクセス可能で、感嘆符 (!) で始まります。

デバッガ拡張コマンドを使用するには、ターゲット コントロール シェルで break コマンドを使用するか、Visual Studio の [ターゲット] メニューで [すべて中断] コマンドを使用して、カーネル デバッガに割り込む必要があります。他にも、!diagnose all コマンドを入力して、ヒープの破損、デッドロック、またはメモリ スタベーションなどの、障害の潜在的な理由を識別することができます。健全なシステムでは、図 4-4 に示すように、CEDebugX はどんな問題も検出しません。

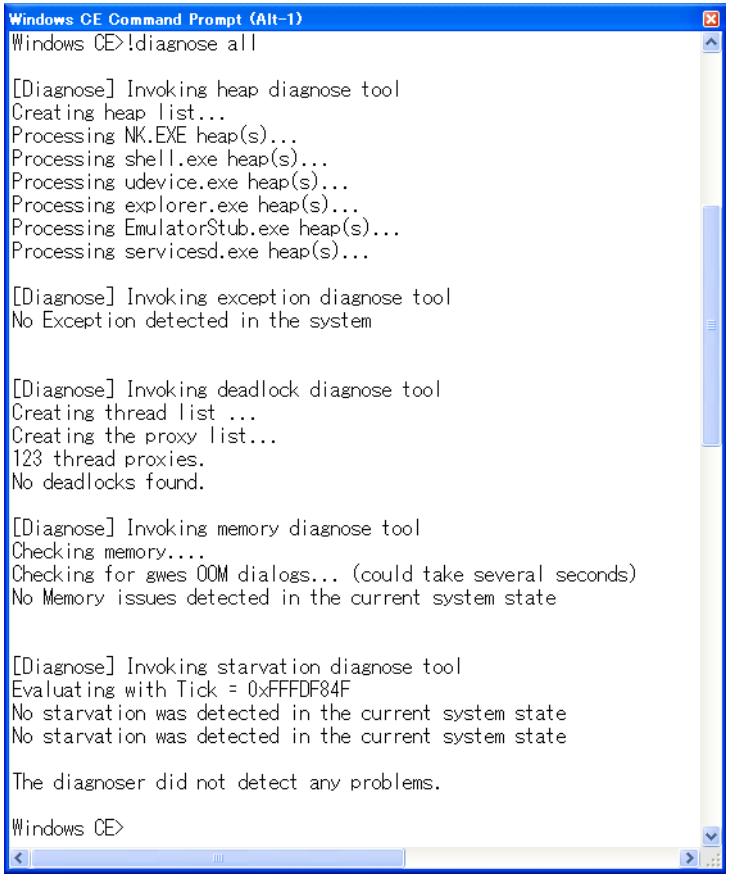


図 4-4　CEDebugX でランタイム イメージを診断

!diagnose all コマンドは、次の診断を実行します。

- ヒープ　システムのプロセスすべてのヒープ　オブジェクトをすべて診断し、潜在的なコンテンツの損傷を識別します。
- 例外　システムで発生した例外を診断し、例外発生時のプロセス　ID、スレッド ID、PC アドレスなどの、例外の詳細を提供します。
- メモリ　システム メモリを診断して、潜在的なメモリの損傷およびメモリ低下状況を識別します。
- デッドロック　スレッド状態とシステム　オブジェクトを診断します（スレッド同期の詳細については、第 3 章を参照）。デッドロックを生成した、システム オブジェクトおよびスレッド ID をリスト表示します。

- スタベーション　スレッドおよびシステム オブジェクトを診断し、潜在的なスレッド スタベーションを識別します。スタベーションは、スケジューラがより優先度の高いスレッドのためにビジーになっているために、スケジューラによってスレッドが全くスケジュールされなかった場合に発生します。

詳細デバッガ ツール

ターゲット コントロール シェルおよび CEDebugX コマンドは実行中のシステムまたは CE ダンプ ファイル (デバッガとして CE ダンプ ファイル リーダーを選択してエラー発生後のデバッグを実行する場合) の詳細な分析を実行できるようにしますが、コマンド ライン インターフェイスへの制限はありません。Platform Builder には、デバッグ効率の向上を目的としたいくつかのグラフィカル ツールが含まれています。これらの詳細デバッガ ツールには、[ウィンドウ] サブメニューを開いているときに [デバッグ] メニューを介して Visutal Studio でアクセスすることができます。

Platform Builder IDE には、次の詳細デバッガ ツールが含まれています。

- ブレークポイント　システムで有効なブレークポイントをリスト表示し、ブレークポイントのプロパティへのアクセスを提供します。
- ウォッチ　ローカルおよびグローバル変数に読み取りおよび書き込みアクセスを提供します。
- 自動変数　[ウォッチ] ウィンドウに似た変数へのアクセスを提供します。変数のこのリストをデバッガが動的に作成するのに対し、[ウォッチ] ウィンドウは、アクセス可能かどうかにかかわりなく、手動で追加された変数をすべてリスト表示します。[自動変数] ウィンドウは、関数に渡されたパラメータ値を確認するのに役立ちます。
- 呼び出し履歴　システムがブレーク状態にあるときにのみアクセス可能です (コード例外がブレークポイントで中断された)。このウィンドウは、システムで有効なすべてのプロセスのリスト、およびホストされたスレッドのリストを提供します。
- スレッド　システムのプロセスで実行中のスレッドのリストを提供します。この情報は、動的に取得され、いつでも更新できます。
- モジュール　システムにロードおよびアンロードされたモジュールをリスト表示し、モジュールがロードされたメモリ アドレスを提供します。この機能は、ドライバ DLL が実際にロードされたかどうかを識別するのに役立ちます。

- プロセス　[スレッド] ウィンドウと同様に、システムで実行中のプロセスのリストを提供します。他にも、必要なときにプロセスを終了させることができます。
- メモリ　デバイス メモリへの直接アクセスを提供します。メモリ アドレスまたは変数名を使用して、必要なメモリ コンテンツを特定できます。
- 逆アセンブリ　システムで実行されている現在のコード行のアセンブリ コードを表示します。
- レジスタ　コードの特定の行を実行しているときに、CPU レジスタ値へのアクセスを提供します。
- 詳細メモリ　デバイス メモリの検索、メモリの一部の別のセクションへの移動、およびコンテンツ パターンを使用したメモリ範囲の充填を行うために使用できます。
- 最近値シンボル一覧　バイナリで使用可能な、最近値シンボル用の特定のメモリ アドレスを決定します。シンボルを含むファイルへの完全なパスも提供します。このツールは、例外を生成した関数の名前を特定するのに役立ちます。

注意　メモリ破壊

メモリおよび詳細メモリ ツールによって、メモリ コンテンツを変更することができます。これらのツールを正しく使用しないと、システム エラーの原因になったり、ターゲット デバイスのオペレーティング システムを損傷したりすることがあります。

アプリケーション検証ツール

潜在的なアプリケーションの互換性や安定性の問題、および必要なソース コード レベルの修正を識別する便利な別のツールは、アプリケーション検証ツールで、CETK に含まれています。このツールは、アプリケーションや DLL に接続して、スタンドアロン デバイス上では追跡するのが難しい問題を診断することができます。アプリケーション検証ツールは、開発ワークステーションへのデバイス接続を必要とせず、システム起動時に起動させて、ドライバやシステム アプリケーションを確認および検証させることができます。このツールは、CETK ユーザー インターフェイスから起動したり、ターゲット デバイスから手動で起動したりすることもできます。CETK の外でアプリケーション検証ツールを使用したい場合、Getappverif_cetk.bat f ファイルを使用して、すべての必要なファイルをリリース ディレクトリにコピーする必要があります。

ノート　アプリケーション検証ツールのドキュメント

シム拡張 DLL を使用してカスタム テスト コードを実行する方法やアプリケーション テスト中の関数の動作を変更する方法を含む、アプリケーション検証ツールの詳細については、http://msdn2.microsoft.com/en-us/library/aa934321.aspx にある Windows Embedded CE 6.0 ドキュメントの「Application Verifier Tool」セクションを参照してください。

CELog イベント追跡および処理

Windows Embedded CE には、ランタイム イメージに含めてパフォーマンス問題を診断できる、拡張性のあるイベント追跡システムが含まれています。CELog イベント追跡システムは、ミューテックス、イベント、メモリ割り当て、および他のカーネル オブジェクトに関連する、事前定義された一連のカーネルおよび coredll イベントをログ記録します。CELog イベント追跡システムの拡張可能なアーキテクチャにより、カスタム フィルタを実装してユーザー定義イベントを追跡できます。KITL を介して開発ワークステーションに接続されたプラットフォームについては、CELog イベント追跡システムは、表 4-4 で要約されているように、ZoneCE レジストリ エントリで指定されたゾーンに基づいて、選択的にイベントをログ記録できます。

表 4-4　イベント ログ記録ゾーン用 CELog レジストリ パラメータ

場所	HKEY_LOCAL_MACHINE\System\CELog
レジストリ エントリ	ZoneCE
エントリ タイプ	REG_DWORD
値	< *ゾーン ID*>
説明	既定では、すべてのゾーンがログ記録されます。すべての使用可能なゾーン ID 値のリストについては、http://msdn2.microsoft.com/en-us/library/aa909194.aspx の Microsoft MSDN Web サイトにある、Windows Embedded CE 6.0 ドキュメントの「CELog Zones」セクションを参照してください。

CELog イベント追跡システムを使用することで、CELog がターゲット デバイスの RAM のバッファに保存していたデータを収集することができます。さらに、Remote Kernel Tracker や Readlog などのパフォーマンス ツールで、収集されたデータを処理することができます。また、CELogFlush ツールを使用して、周期的にデータをファイルにフラッシュすることもできます。

ノート　CELog およびシップ ビルド

CELog の動作によるパフォーマンスやメモリ障害を避けるため、またシステムを損なおうとする悪意のあるユーザーからの攻撃範囲を狭めるため、CELog イベント追跡システムを最終ビルドに含めないようにする必要があります。

Remote Kernel Tracker

Remote Kernel Tracker ツールにより、プロセスやスレッドに基づいて、システム アクティビティの監視を行うことができます。このツールは、KITL を介してリアルタイムにターゲット デバイスからの情報を表示できますが、CELog データファイルに基づいて、Remote Kernel Tracker をオフラインで使用することも可能です。第 3 章「システム プログラミングの実行」で、Remote Kernel Tracker ツールに関する詳細情報を確認できます。

図 4 ミ 5 は、スレッド動作に関する情報を収集するターゲット デバイス上の Kernel Tracker を示しています。

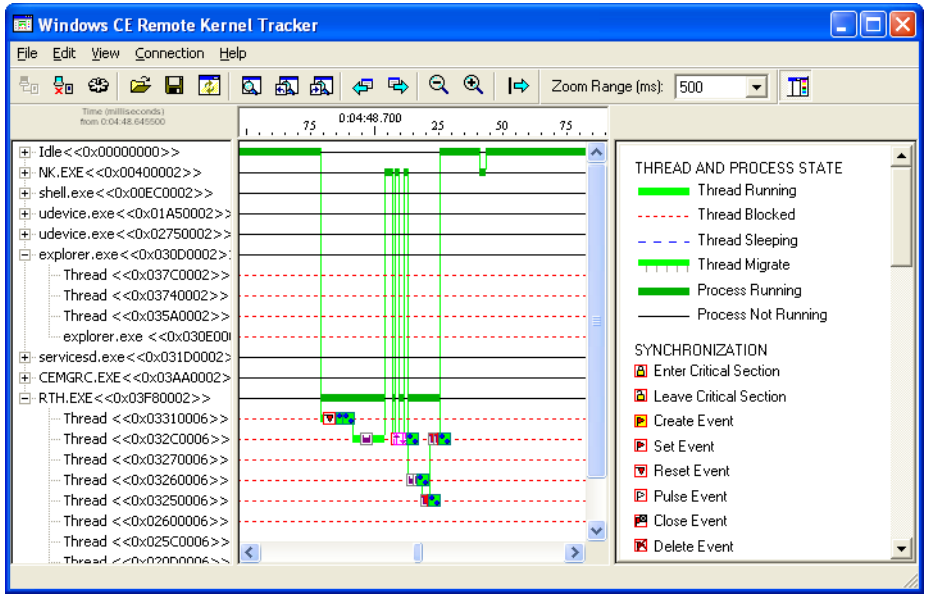


図 4-5　Kernel Tracker のスレッド情報

CELogFlush ツール

CELog データ ファイルを作成するには、CELogFlush ツールを使用し、RAM にバッファされた CELog イベント データを .clg ファイルに保存します。このファイルは、開発ワークステーションの RAM ファイル システム、不揮発性記憶媒

体またはリリース ファイル システムにあります。バッファ オーバーランによるデータ損失を最小限に抑えるため、より大きな RAM バッファを指定して、CELog がバッファをフラッシュする頻度を高めることができます。繰り返しファイルを開いたり閉じたりする操作を避けるためにファイルを開いたままにしたり、ファイルを応答の遅い不揮発性記憶媒体ではなく、RAM ファイル システムに保存したりすることで、パフォーマンスを最適化することができます。

ノート　CELogFlush 構成
レジストリ設定からこのツールを構成する方法などの、CELogFlush ツールに関する詳細情報については、http://msdn2.microsoft.com/en-us/library/aa935267.aspx の Microsoft MSDN Web サイトにある、Windows Embedded CE 6.0 ドキュメントの「CELogFlush Registry Settings」セクションを参照してください。

Readlog ツール

グラフィカルな Remote Kernel Tracker アプリケーションに加えて、%_WINCEROOT%\Public\Common\Oak\Bin\i386 フォルダにある Readlog ツールを使用して CELog データを処理することができます。Readlog はコマンド ライン ツールで、デバッグ メッセージやブート イベントなど、Remote Kernel Tracker によって明らかにされていない情報を処理および表示します。最初に Remote Kernel Tracker でシステム動作を分析してから、Readlog ツールを使用して識別されたプロセスやスレッドに集中するのが便利な方法です。CELogFlush ツールによって .clg ファイルに書き込まれる生データは、ゾーンによって指定され、特定の情報を特定し抽出するために使用されます。データにフィルタをかけたり、拡張 DLL に基づいてフィルタ機能を拡張して、カスタム イベント コレクタによって取得されたカスタム データを処理できます。

最も便利な Readlog シナリオは、CELog データ ファイルのスレッド開始アドレス (CreateThead 呼び出しに渡された関数) を実際のスレッド関数の名前に置き換えて、Remote Kernel Tracker のシステム分析を促進することです。このタスクを実現するには、fixthreads パラメータ (readlog -fixthreads) を使用して Readlog を開始する必要があります。Readlog は、リリース ディレクトリのシンボル .map ファイルを検索して、開始アドレスに基づいてスレッド関数を識別し、対応する参照を使用して新しいログを生成します。

図 4ミ6 は、Remote Kernel Tracker の CELog データを示しており、CELog イベント追跡システムによって取得され、CELogFlush ツールによって .clg ファイルにフラッシュされ、Readlog アプリケーションを -fixthreads パラメータで使用することで、情報の見易さを向上する準備をします。

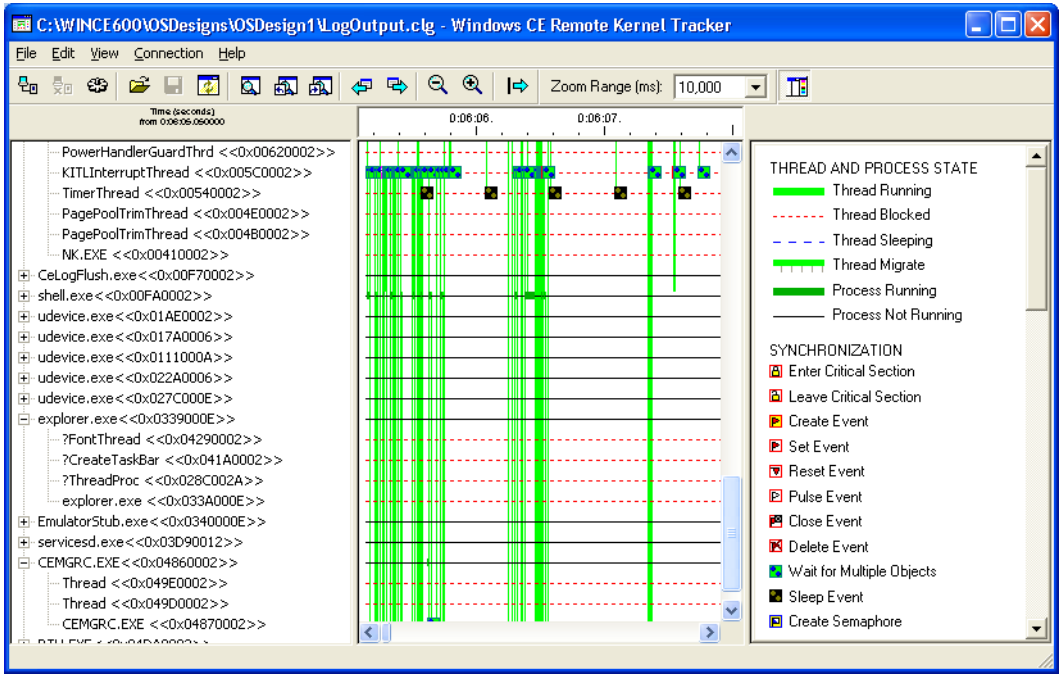


図 4-6　readlog -fixthreads によって準備され、Remote Kernel Tracker で開かれた CELog データファイル

ノート　参照名マッチングの向上

CELog イベント追跡システムは、IMGPROFILER 環境変数セットを使用してイメージのリビルドを行うことで、明示的にカーネル プロファイラを有効にし、プロファイル シンボルをランタイム イメージに追加している場合、カーネル プロファイラを使用して、CreateThread イベントのキャプチャ時に開始アドレスに基づいてスレッド関数名を検索できます。ただし、CELog は、ランタイム イメージにビルドされたプロファイル シンボルのみ検索できます。ソフトウェア開発キット (SDK) に基づいて開発されたアプリケーションのシンボルは、通常、CELog イベント追跡システムでは使用できません。

レッスン概要

オペレーティング システムおよびアプリケーションのデバッグのためには、CE システムおよび Platform Builder や CETK を含む、デバッグ ツールに精通している必要があります。最も重要なデバッグ ツールは、システム デバッガ、デバッグ メッセージ機能、および CE ターゲット コントロール シェルです。システム デバッガにより、ブレークポイントの設定、カーネルやアプリケーション コードのステップ実行ができるのに対し、デバッグ メッセージ機能は、コード実行を中断することなくシステム コンポーネントやアプリケーションの分析を行うオプションを提供します。多様なデバッグおよびリテール マクロは、表示コンポーネントのある / ないターゲット デバイスからデバッグ情報を出力するのに使用できます。システムおよびアプリケーションは、潜在的に大量のデバッグ メッセージを生成できるため、デバッグ情報の出力をコントロールするためにデバッグ ゾーンを使用する必要があります。デバッグ ゾーンの主要な利点は、ランタイム イメージをリビルドすることなく、デバッグ情報の詳細度を動的に変更できることです。それに対し、ターゲット コントロール シェルによって、コマンドをターゲット デバイスに送信することができます。例えば、break コマンドに続けて !diagnose all コマンドを使用することで、デバッガに割り込み、CEDebugX を実行して、メモリ リーク、例外およびデッドロックなどの、全体的なシステムの健全性を確認できます。

これらのコア デバッグ ツール以外に、特有の CE 構成ツールやトラブルシューティング ツールを使用できます。例えば、アプリケーション検証ツールによって、潜在的なアプリケーションの互換性や安定性問題を識別したり、Remote Kernel Tracker を使用してプロセス、スレッド、およびシステム性能を分析したりできます。Remote Kernel Tracker は、CELog イベント追跡システム、特にターゲット デバイスのメモリのログ記録されたデータに依存しています。また、このデータを CELogFlush ツールを使用してファイルにフラッシュすることもできます。シンボル ファイルが分析したいモジュールに対して使用可能な場合、Readlog ツールを使用してスレッド開始アドレスを実際の関数名に置き換えたり、Remote Kernel Tracker でのより便利なオフライン分析用に CELog データ ファイルを生成したりすることもできます。

レッスン 2：ランタイム イメージを構成してデバッグを有効にする

Windows Embedded CE のデバッグ機能は、開発ワークステーション コンポーネントおよびターゲット デバイスに依存しており、特定の設定とハードウェア サポートが必要です。開発ワークステーションとターゲット デバイス間の接続なしでは、デバッグ情報や他の要求を交換することはできません。例えば、この通信リンクが切断された場合、最初にターゲット側のデバッグ スタブをアンロードすることなく開発ワークステーションのデバッガを停止しているため、例外発生後にデバッガがコード実行を再開するのを待っている間、ランタイム イメージはユーザー入力に対する応答を停止することがあります。

このレッスンの後、次のことができるようになります。

- ランタイム イメージのカーネル デバッガを有効にする
- KITL の要件を識別する
- デバッグ コンテキストでカーネル デバッガを使用する

レッスン時間 (推定) : 20 分

カーネル デバッガを有効にする

レッスン 1 で説明したように、Windows Embedded CE 6.0 の開発環境には、CE ターゲット デバイス上で開発者がコードのステップ実行を対話的に行うことを可能にする、カーネル デバッガが含まれています。このデバッガは、カーネル オプションおよびターゲット デバイスと開発コンピュータ間の通信層の設定が必要です。

OS デザイン設定

デバッグ用に OS デザインを有効にするには、環境変数 IMGNODEBUGGER および IMGNOKITL を設定解除して、Platform Builder に KdStub ライブラリを含め、ランタイム イメージをビルドするときに、ボード サポート パッケージ (BSP) で KITL 通信層を有効にする必要があります。Platform Builder は、このタスクを完了するための便利な手法を提供します。Visual Studio で、OS デザイン プロジェクトを右クリックし、[プロパティ] を選択して [OS デザイン] プロパティ ページを表示します。[ビルド オプション] ペインに切り替えてから、[カーネル デバッガを有効にする] と [KITL を有効にする] を選択します。第 1 章「オペレーティング システム設計のカスタマイズ」で、[OS デザイン] プロパティ ページのダイアログ ボックスの詳細を説明しています。

デバッガの選択

ランタイム イメージ用に KbStub および KITL を有効にしていれば、デバッガを選択してターゲット デバイスの通信パラメータを使用してターゲット デバイス上でシステムお分析をすることができます。パラメータを構成するには、第 2 章「ランタイム イメージのビルドおよび展開」で説明されているように、Visual Studio で [ターゲット] メニューを開いてから [接続オプション] を選択することで、[ターゲット デバイスの接続オプション] ダイアログ ボックスを表示します。

既定では、接続オプションにデバッガは選択されていません。次のデバッガの選択肢があります。

- KdStub　これは、カーネルおよびアプリケーションのソフトウェア デバッガで、システム コンポーネント、ドライバ、およびターゲット デバイス上で実行されるアプリケーションのデバッグを行います。KdStub は、Platform Builder と通信するためには KITL が必要です。
- CE ダンプ ファイル リーダー　Platform Builder は、ダンプ ファイルをキャプチャするオプションを提供し、CE ダンプ ファイル リーダーを使用してダンプ ファイルを開くことができます。ダンプ ファイルによって、特定の時点におけるシステムの状態を確認することができ、参考にするのに便利です。
- サンプル デバイス エミュレータ eXDI 2 ドライバ　KdStub は、カーネルのロード前にシステムが実行するルーチンをデバッグしたり、割り込みサービス ルーチン (ISR) をデバッグすることはできません。これは、デバッグ ライブラリがソフトウェア ブレークポイントに依存しているためです。ハードウェア補助デバッグでは、Platform Builder に、JTAG (Joint Test Action Group) プローブと併用可能なサンプル eXDI ドライバを含めます。JTAG プローブにより、プロセッサによって処理されるハードウェア ブレークポイントを設定できます。

ノート　ハードウェア補助デバッグ

ハードウェア補助デバッグの詳細については、http://msdn2.microsoft.com/en-us/library/aa935824.aspx の Microsoft MSDN Web サイトにある、Windows Embedded CE 6.0 ドキュメントの「Hardware-assisted Debugging」セクションを参照してください。

KITL

この章の初めで、図 4ミ1 に示したように KITL は開発コンピュータとターゲット デバイス間の必要不可欠な通信層であり、カーネル デバッガ サポートが有効にされている必要があります。名前が示しているように、KITL は完全にハードウェアと独立しており、ネットワーク接続、シリアル ケーブル、USB (ユニバーサル シリアル バス)、または DMA (直接メモリ アクセス) などの他のサポートされている通信機構を介して動作します。唯一の要件は、両側 (開発コンピュータとターゲット デバイス) で同一のインターフェイスがサポートされ、使用されていることです。図 4-7 に示すように、デバイス エミュレータ用の最も一般的で高速な KITL インターフェイスは DMA です。イーサネット チップをサポートするターゲット デバイスについては、通常、ネットワーク インターフェイスを使用するのが最適です。

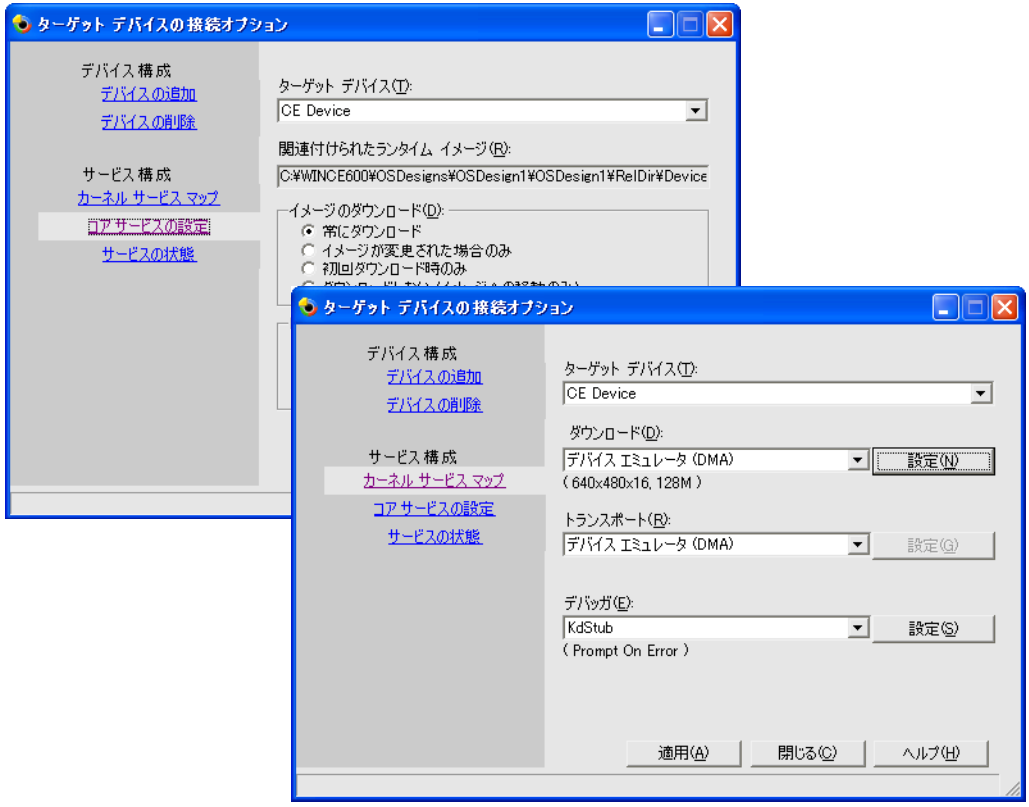


図 4-7 KITL 通信インターフェイスの構成

KITL は、次の 2 つの操作方法をサポートしています。

- アクティブ モード　既定では、Platform Builder は KITL を構成して、起動プロセス中に開発コンピュータと接続します。この設定は、ソフトウェア開発サイクル中のカーネルおよびアプリケーション デバッグに最も便利です。
- 受動モード　[デバイスの起動時に KITL を有効にする] チェックボックスの選択を解除することで、KITL を受動モードで構成できます。つまり、Windows Embedded CE は KITL インターフェイスを初期化しますが、KITL は起動プロセス中に接続を確立しません。例外が発生した場合、KITL は、開発コンピュータへの接続の確立を試みるため、JIT デバッグの実行が可能です。受動モードは、起動時に開発コンピュータへの物理的な接続がないモバイル デバイスで作業するのに最も適しています。

ノート　KITL モードおよびブート引数

[デバイスの起動時に KITL を有効にする] 設定は、ブート ローダー用に Platform Builder を構成するブート引数 (BootArgs) です。ブート ローダーおよび BSP 開発プロセス中の利点の詳細については、http://msdn2.microsoft.com/en-us/library/aa917791.aspx の Microsoft MSDN Web サイトにある、Windows Embedded CE 6.0 ドキュメントの「Boot Loaders」セクションを参照してください。

ターゲット デバイスのデバッグ

開発側デバッガ コンポーネントとターゲット側デバッガ コンポーネントは、互いに独立して実行されていることに留意するのは重要です。例えば、アクティブ ターゲット デバイスを使用せずに、Platform Builder を使用して Visual Studio 2005 でカーネル デバッガを実行することが可能です。[デバッグ] メニューを開いてから [開始] をクリックするか、F5 キーを押した場合、カーネル デバッガが開始し、[出力] ウィンドウにターゲット デバイスへの接続を待機していることを示す情報が表示されます。それに対し、デバッガへのアクティブな KITL 接続なしでデバッグ可能なランタイム イメージを開始して、例外が発生した場合、この章で前述したように、システムが停止するため、停止のランタイム イメージが表示され、デバッガからのコントロール要求を待機します。この理由で、デバッグ可能ターゲット デバイスを接続するときは、通常、デバッガが自動的に開始します。F5 を押してデバッグ セッションを開始する代わりに、[ターゲット] メニューで [デバイスの接続] を使用することもできます。

有効化および管理ブレークポイント

Platform Builder のデバッグ機能は、Windows デスクトップ アプリケーションの他のデバッガにある機能をほとんど提供します。図 4-8 で示すように、ブレークポイントの設定、行ごとのコードのステップ実行、および [ウォッチ] ウィンドウを使用した変数値やオブジェクト プロパティの表示や変更を行うことができます。ただし、ブレークポイントを使用できるかどうかは、ランタイム イメージに KdStub ライブラリが存在しているかどうかに依存していることに留意してください。

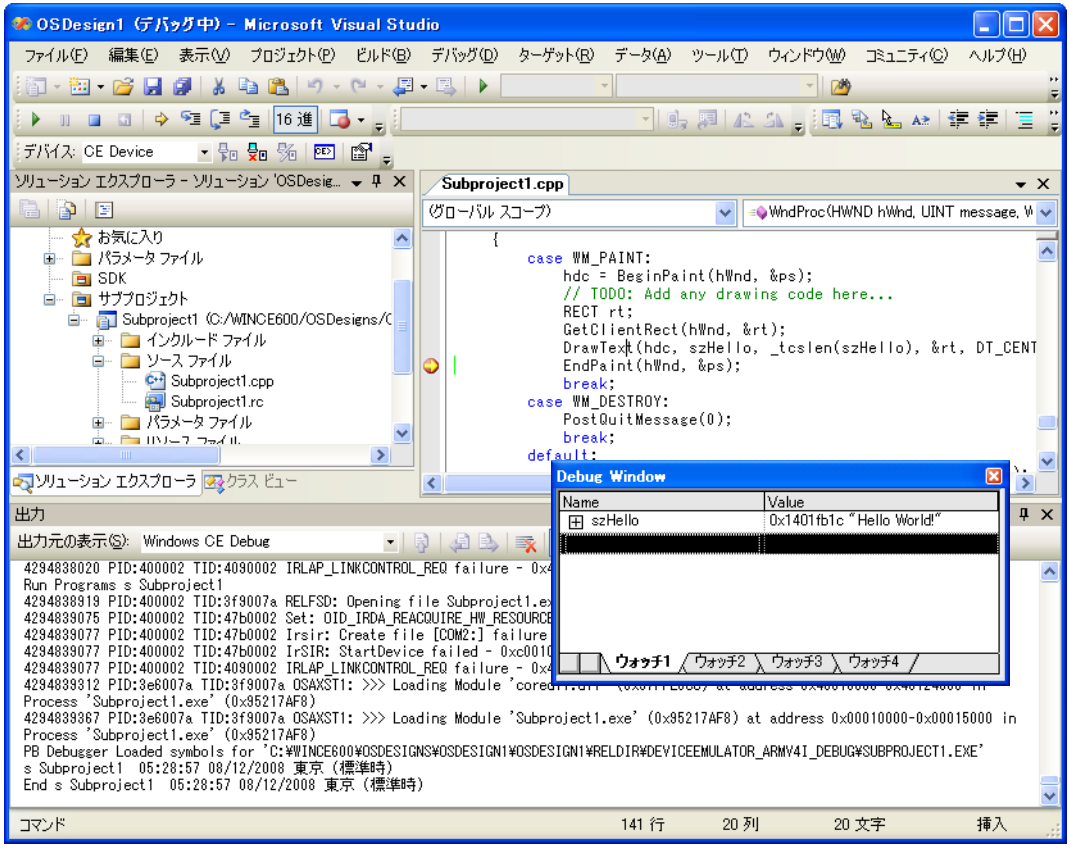


図 4-8　Hello World アプリケーションのデバッグ

ブレークポイントを設定するには、Visual Studio の [デバッグ] メニューで [ブレークポイントの設定 / 解除] を使用します。他の方法として、F9 を押して現在の行にブレークポイントを設定するか、コード行の左側の空白部分をクリックします。選択に従って、デバッガがブレークポイントをインスタンス化でき

るかに応じて、Platform Builder はブレークポイントを赤い点か赤い丸印で表示します。赤い丸印は、ブレークポイントがインスタンス化されていないことを示します。インスタンス化されていないブレークポイントが発生するのは、Visual Studio インスタンスがターゲット コードにリンクされていない場合、ブレークポイントが設定されているがまだロードされていない場合、デバッガが有効になっていない場合、またはデバッガが実行中だがコード実行がまだ中断されていない場合です。デバッガが実行中のときにブレークポイントを設定すると、デバッグがブレークポイントをインスタンス化できるようにする前に、まずデバイスがデバッガに割り込む必要があります。

Visual Studio で Platform Builder を使用してブレークポイントを管理するには、次のようなオプションがあります。

- [ソース コード]、[呼び出し履歴]、[逆アセンブリ] ウィンドウ　F9 を押すか、[デバッグ] メニューから [ブレークポイントの設定 / 解除] を選択する、またはコンテキスト メニューから [ブレークポイントの挿入 / 削除] を選択することで、ブレークポイントを設定、削除、有効化、または無効化することができます。
- [新規ブレークポイント] ダイアログ ボックス　[デバッグ] メニューの [新規ブレークポイント] にあるサブメニューからこのダイアログ ボックスを表示することができます。[新規ブレークポイント] ダイアログ ボックスにより、ブレークポイントの場所および条件を設定することができます。ループ カウンタや他の変数が特定の値になったなど、指定された条件が TRUE になった場合にのみ、デバッガは条件付きブレークポイントで停止します。
- [ブレークポイント] ウィンドウ　[デバッグ] メニューの [ウィンドウ] サブメニューから [ブレークポイント] をクリックするか、Alt+F9 を押すことで、[ブレークポイント] ウィンドウを表示することができます。[ブレークポイント] ウィンドウによって、設定されたブレークポイントがすべて一覧表示され、ブレークポイントのプロパティを構成できます。例えば、手動で場所情報を指定する必要のある [新規ブレークポイント] ダイアログ ボックスを使用する代わりに、ソース コードで直接必要なブレークポイントを直接設定してから、[ブレークポイント] ウィンドウでこのブレークポイントをのプロパティを表示して、条件パラメータを定義することもできます。

ヒント　ブレークポイントが多すぎる

ブレークポイントは控えめに使用します。ブレークポイントの設定数が多すぎると、頻繁に [再開] を選択する必要があり、デバッグの効率に影響を与え、一度にシステムの 1 つの側面に集中するのが難しくなります。必要に応じて、ブレークポイントを無効にしたり、解除したりすることを考慮します。

ブレークポイントの制限

[新規ブレークポイント] ダイアログ ボックスまたは [ブレークポイント] ウィンドウでブレークポイントのプロパティを構成するとき、[ハードウェア] ボタンが表示されます。これを使用して、ブレークポイントをハードウェア ブレークポイントまたはソフトウェア ブレークポイントとして構成することができます。OAL コードや割り込みハンドラでソフトウェア ブレークポイントを使用することはできません。ブレークポイントは、システムの実行を完全に中断する必要があるためです。他のシステム プロセスでは、KITL 接続はアクティブであり続ける必要があります。KITL 接続が開発ワークステーションでデバッガと通信する唯一の手段であるためです。KITL は OAL のインターフェイスとなり、カーネルの割り込みベース通信機構を使用します。ブレークポイントを割り込みハンドラ関数で設定する場合、ブレークポイントに達するとシステムはそれ以上通信できなくなります。割り込みハンドラは単一スレッドで、割り込み可能な関数ではないためです。

割り込みハンドラをデバッグする必要がある場合、デバッグ メッセージかハードウェア ブレークポイントを使用できます。ただし、ハードウェア ブレークポイントには、プロセッサのデバッグ レジスタで割り込みを登録するために eXDI 互換デバッガ (JTAG プローブなど) が必要です。JTAG は複数のデバッガを管理できますが、通常、一度に 1 つのプロセッサで 1 つのハードウェア デバッガを有効にすることができます。ハードウェア補助デバッグには、KdStub ライブラリを使用することはできません。

ハードウェア ブレークポイントを構成するには、次の手順に従います。

1. [デバッグ] メニューを開いてから、[ブレークポイント] をクリックして、[ブレークポイント] ウィンドウを開きます。
2. ブレークポイント リストでブレークポイントを選択し、右クリックします。
3. [ブレークポイントのプロパティ] をクリックして [ブレークポイントのプロパティ] ダイアログ ボックスを表示し、[ハードウェア] ボタンをクリックします。

4. [ハードウェア] ラジオ ボタンを選択してから、[OK] を 2 回クリックしてすべてのダイアログ ボックスを閉じます。

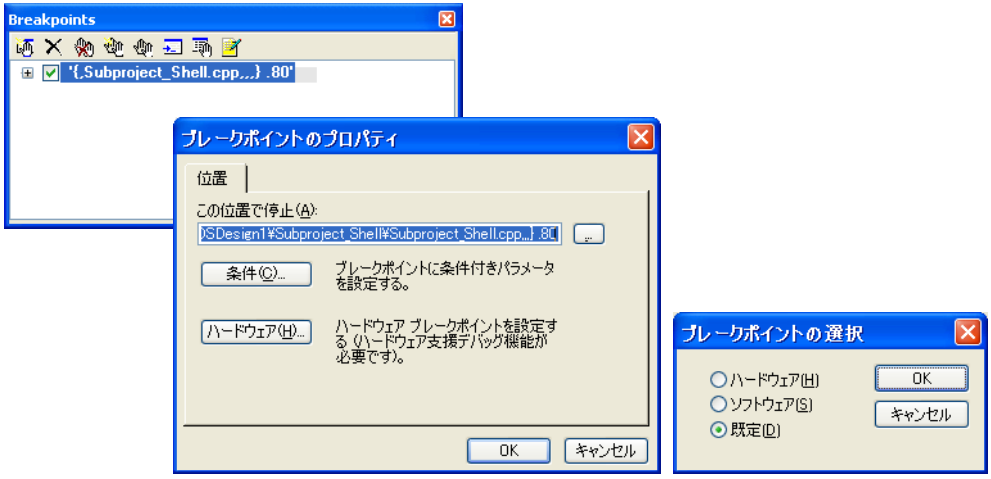


図 4-9　ハードウェア ブレークポイントを設定する

レッスン概要

デバッガの有効化は、ランタイム イメージに KITL とデバッガ ライブラリが含まれていれば、Platform Builder IDE で直接的に構成プロセスを実行できます。次いで、[ターゲット デバイスの接続オプション] ダイアログ ボックスを表示してから、適切なトランスポートとデバッガを選択します。通常、トランスポートは DMA またはイーサネットですが、USB やシリアル ケーブルを使用して開発ワークステーションをターゲット デバイスと接続することも可能です。

Platform Builder のデバッグ機能は、Windows デスクトップ アプリケーションの他のデバッガにある機能をほとんど提供します。ブレークポイントの設定、行ごとのコードのステップ実行、および [ウォッチ] ウィンドウを使用した変数値やオブジェクト プロパティの表示および変更が可能です。また、Platform Builder は指定された基準に基づいて、コード実行を中断する条件付きブレークポイントもサポートしています。JTAG プローブに基づくハードウェア補助デバッグや他のハードウェア デバッガには、eXDI ドライバを Platform Builder とともに使用することもできますが、ソフトウェア デバッグではデバッガとして KdStub を選択します。ハードウェア補助デバッグにより、ソフトウェア ブレークポイントを使用できない、カーネル、OAL コンポーネント、および割り込みハンドラ関数がロードされる前に実行されるシステム ルーチンの分析を行うことができます。

レッスン 3：CETK を使用してシステムをテストする

ソフトウェア テストは、開発コストおよびサポート コストを低減しながら製品品質を向上する重要な要素です。これは、ターゲット デバイス用にカスタム BSP を作成した場合、新しいデバイス ドライバを追加した場合、およびカスタム OAL コードを実装した場合に特に重要です。システムの新しいシリーズを生産用にリリースする前に、機能テスト、ユニット テスト、ストレス テスト、および他のタイプのテストを実行して、システムの各部分を検証し、ターゲット デバイスが通常の条件下で高い信頼性で動作することを確認するのは重要です。通常、製品を市場に出してから欠陥を修正するのは、テスト ツールとスクリプトを作成して、ターゲット デバイスのユーザー操作のシミュレーションを行い、システムの開発中に欠陥を修正するのに比べ、非常に多くのコストがかかります。システム テストは、後回しにすべきではありません。システム テストを効率的にソフトウェア開発サイクル全体で実行するには、CETK を使用できます。

このレッスンの後、次のことができるようになります。

- CETK テスト ツールの通常の使用法を理解する。
- ユーザー定義の CETK テストを作成する。
- ターゲット デバイスで CETK テストを実行する。

レッスン時間 (推定) : 30 分

Windows Embedded CE テスト キットの概要

CETK は、Windows Embedded CE の Platform Builder に含まれる、個別のテスト アプリケーションであり、CE テスト カタログで適切に計画された一連の自動化テストに基づいて、アプリケーションおよびデバイス ドライバの安定性を検証します。CETK には、付属デバイスのいくつかのドライバ カテゴリ用の多数の既定のテストが含まれています。また、特定の必要に合わせてカスタム テストを作成することも可能です。

ノート　CETK テスト

CETK に含まれる既定のテストの完全なリストについては、http://msdn2.microsoft.com/en-us/library/aa917791.aspx の Microsoft MSDN Web サイトにある、Windows Embedded CE 6.0 ドキュメントの「CETK Tests」セクションを参照してください。

CETK アーキテクチャ

図 4-10 に示されているように、CETK アプリケーションは、開発コンピュータおよびターゲット デバイス上で実行するコンポーネントを使用する、クライアント / サーバー ソリューションです。開発コンピュータはワークステーション サーバー アプリケーション (CETest.exe) を実行するのに対し、ターゲット デバイスはクライアント側アプリケーション (Clientside.exe)、テスト エンジン (Tux.exe)、およびテスト結果ロガー (Kato.exe) を実行します。他にも、このアーキテクチャにより、同一開発アプリケーションからの複数の異なるデバイスの同時テストを実行できます。ワークステーション サーバーおよびクライアント側アプリケーションは、KITL、ActiveSync ィまたは Windows Sockets (Winsock) 接続を介して通信を行うことができます。

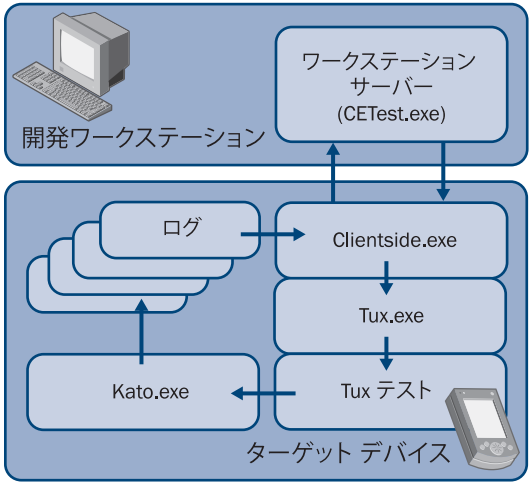


図 4-10　CETK クライアント / サーバー アーキテクチャ

CETK アプリケーションには、次のコンポーネントが含まれます。

- 開発ワークステーション サーバー　CETest.exe は、グラフィカル ユーザー インターフェイス (GUI) を提供し、CETK テストを実行および管理します。また、このアプリケーションにより、サーバー設定および接続パラメータの構成、およびターゲット デバイスへの接続が可能になります。デバイス接続を確立することで、ワークステーション サーバーは自動的にクライアント側アプリケーションをダウンロードおよび開始し、テスト要求を送信し、取得されたログに基づいてテスト結果を編集してリアルタイムに表示します。

- クライアント側アプリケーション　Clientside.exe は、ワークステーション サーバー アプリケーションへのインターフェイスを提供し、テスト エンジンをコントロールし、テスト結果をサーバー アプリケーションに返します。Clientside.exe がターゲット デバイスで使用可能でない場合、ワークステーション サーバーはターゲット デバイスへの通信ストリームを確立できません。
- テスト エンジン　CETK テストは、Tux.exe がターゲット デバイス上でロードおよび実行する DLL に実装されます。通常、ワークステーション サーバーおよびクライアント側アプリケーションを介してリモートでテスト エンジンを起動しますが、Tux.exe をローカルで起動して、ワークステーション サーバー要件なしでスタンドアロンで実行することも可能です。
- テスト結果ロガー　Kato.exe は、ログ ファイルで CETK テストの結果を追跡します。Tux DLL は、このロガーを使用して、テストが成功または失敗したかに関する追加情報を提供したり、複数のユーザー定義出力デバイスに出力を送信したりすることができます。すべての CETK テストで、同一のロガーおよびフォーマットを使用するため、特定の要件に応じて、自動結果処理用に既定のファイル パーサーを使用したり、カスタム ログ ファイル パーサーを実装したりすることができます。

ノート　マネージ コード用の CETK

CETK の管理バージョンは、ネイティブおよびマネージ コードで使用可能です。マネージ バージョンの詳細については、http://msdn2.microsoft.com/en-us/library/aa934705.aspx の Microsoft MSDN Web サイトにある、Windows Embedded CE 6.0 ドキュメントの「Tux.Net Test Harness」セクションを参照してください。

CETK を使用する

ターゲット デバイスでサポートされている接続オプションに応じて、多様な方法で CETK テストを実行することができます。KITL、Microsoft ActiveSync、または TCP/IP 接続を使用してターゲット デバイスに接続し、ターゲット側 CETK コンポーネントのダウンロード、必要なテストの実行、および結果の開発ワークステーション上のグラフィカル ユーザー インターフェイスでの表示が可能です。それに対し、ターゲット デバイスが接続オプションをサポートしない場合、適切なコマンド ライン オプションを使用して、テストをローカルで実行する必要があります。

CETK ワークステーション サーバー アプリケーションを使用する

ワークステーション サーバー アプリケーションを使用して作業するには、開発コンピュータで Windows Embedded CE 6.0 プログラム グループの [Windows Embedded CE 6.0 テスト キット] をクリックして、[接続] メニューを開き、[クライアントの開始] コマンドを選択します。次いで、[設定] ボタンをクリックして、トランスポートを構成できます。ターゲット デバイスのスイッチがオンになっていて、開発ワークステーションに接続されている場合、[接続] をクリックし、希望のターゲット デバイスを選択してから、[OK] をクリックして通信チャネルの確立および必要なバイナリの展開を行います。これで、CETK アプリケーションをターゲット デバイス上で実行する準備が整いました。

図 4-11 に示されているように、CETK アプリケーションは、ターゲットで使用可能なデバイス ドライバを検出し、テストを実行するための便利な手法を提供します。1 つの方法は、[テスト] メニューの [テストの起動 / 停止] の下にあるデバイス名をクリックする方法です。これにより、CETK は検出されたコンポーネントをすべてテストします。もう 1 つの方法は、[テスト カタログ] ノードを右クリックしてから、[テストの開始] を選択することです。また、個別のコンテナを拡張子、各デバイス テストを右クリックし、[クイック スタート] をクリックして単一コンポーネントのみをテストすることもできます。ワークステーション サーバー アプリケーションでは、デバイス ノードを右クリックして、[ツール] サブメニューを開いたときに、アプリケーション検証、CPU モニタ、リソース消費、および Windows Embedded CE ストレス ツールへのアクセスが提供されます。

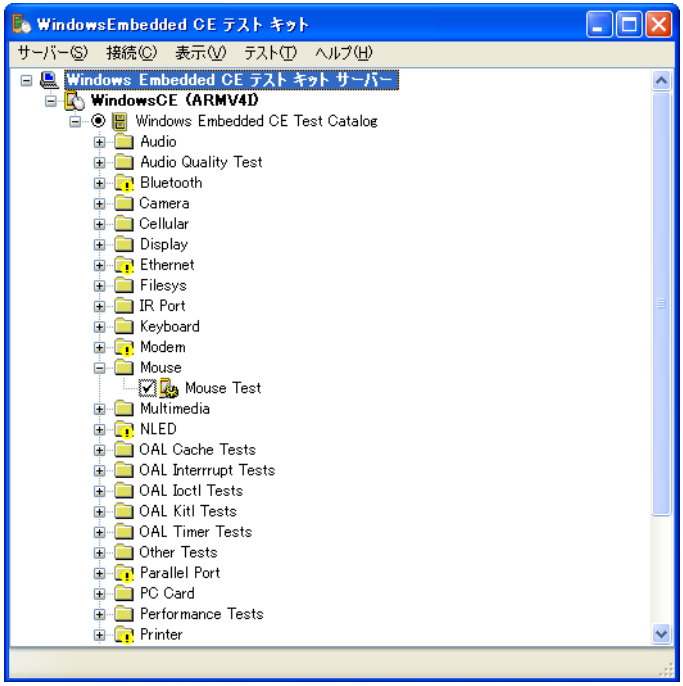


図 4-11　CETK アプリケーションのグラフィカル ユーザー インターフェイス

テスト スイートの作成

一度にすべてのテストを実行したり、個別にクイック テストを実行したりする以外に、ソフトウェア開発サイクル全体にわたって繰り返し実行したい一連のカスタム テストを含めた、テスト スイートを作成することができます。新規のテスト スイートを作成するには、ワークステーション サーバー アプリケーションの [テスト] メニューで使用可能な [テスト スイート エディタ] を使用します。[テスト スイート エディタ] は、グラフィカル ツールで、スイート含めるテストを簡単に選択できます。テスト スイート定義をテスト キット スイート (.tks) ファイルの形式でエクスポートしたり、これらのファイルを追加の開発コンピュータにインポートして、すべてのワークステーション サーバーで確実に同一の一連のテストを実行させることができます。これらの .tks ファイルは、テスト定義アーカイブの基礎を提供することもできます。

既定のテストをカスタマイズする

グラフィカル ユーザー インターフェイスによって、ワークステーション サーバー アプリケーションがテストの実行用にテスト エンジン (Tux.exe) に送信するコマンド ラインをカスタマイズすることができます。テストのパラメータを変更するには、[テスト カタログ] でテストを右クリックして、[コマンド ライ

ンの編集] オプションを選択します。例えば、[Storage Device Block Driver Benchmark Test] は、デバイスの各セクタへのデータを読み込みまたは書き込むことで、記憶デバイスのパフォーマンスを分析します。これは、記憶デバイス上のすべての既存データが破壊されることを意味しています。不意のデータ損失から保護するため、既定では、[Storage Device Block Driver Benchmark Test] はスキップされます。[Storage Device Block Driver Benchmark Test] を確実に実行するには、コマンド ラインを編集して、-zorch によって呼び出される特殊パラメータを明示的に追加する必要があります。

サポートされるコマンド ライン パラメータは、各個別の CETK テスト実装に依存しています。テストには、テストするデバイス ドライバを識別するインデックス番号などの、多様な構成パラメータ、またはテストの実行用の提供される必要のある追加情報などがサポートおよび必要とされます。

ノート　CETK テスト用コマンドライン パラメータ

追加情報へのリンクを含む既定の　CETK　の完全なリストについては、http://msdn2.microsoft.com/en-us/library/ms893193.aspx. の Microsoft MSDN Web サイトにある、Windows Embedded CE 6.0 ドキュメントの「CETK Tests」セクションを参照してください。

Clientside.exe を手動で実行する

Windows Embedded CE テスト キット カタログ アイテムをランタイム イメージに含めている、CETK コンポーネントをワークステーション サーバー アプリケーションと共にダウンロードしている、またはファイル ビューア リモート ツールを使用してコンポーネントを開発ワークステーションからターゲット デバイスにエクスポートしている場合、ターゲット デバイス上で Clientside.exe を実行し、ワークステーション サーバーへの接続を手動で確立できます。ターゲット デバイスが、この目的で [実行] ダイアログ ボックスが提供されない場合、[Platform Builder IDE] で [ターゲット] メニューを開き、[プログラムの実行] を選択し、[Clientside.exe] を選択してから、[実行] を選択します。

Clientside.exe は、特定のワークステーション サーバー アプリケーションに接続し、インストールされたドライバを検出し、自動的にテストを実行するために指定できる、次のコマンドライン パラメータをサポートしています。

```
Clientside.exe [/i=<サーバー IP アドレス> | /n=<サーバー名>] [/p=<サーバー ポート番号>] [/a] [/s] [/d] [/x]
```

これらのパラメータ ファイルは、ターゲット デバイスの Wcetk.txt file や HKEY_LOCAL_MACHINE/Software/Microsoft/CETT レジストリ キーでも定義

することができ、Clientside.exe をコマンドライン パラメータなしで起動できることも注意すべき重要な点です。この場合、Clientside.exe は、ルート ディレクトリで Wcetk.txt を検索してから、ターゲット デバイスの Windows ディレクトリを検索し、次いで開発ワークステーションのリリース ディレクトリを検索します。Wcetk.txt がどこにも存在しない場合、CETT レジストリ キーが確認されます。表 4ミ5 は Clientside.exe パラメータを要約しています。

表 4-5 Clientside.exe 開始パラメータ

コマンド ライン	Wcetk.txt	CETT レジストリ キー	説明
/n	SERVERNAME	ServerName (REG_SZ)	ホスト サーバー名を指定します。/i と併用することはできず、名前解決には、ドメイン ネーム システム (DNS) が必要です。
/i	SERVERIP	ServerIP (REG_SZ)	ホスト IP アドレスを指定します。/n と併用することはできません。
/p	PORTNUMBER	PortNumber (REG_DWORD)	ワークステーション サーバー インターフェイスから行誠意可能なサーバー ポート番号を指定します。
/a	AUTORUN	Autorun (REG_SZ)	1 に設定すると、接続が確立された後に、自動的にテストを開始します。
/s	DEFAULTSUITE	DefaultSuite (REG_SZ)	実行する既定のテスト スイートの名前を指定します。
/x	AUTOEXIT	Autoexit (REG_SZ)	1 に設定すると、テストが完了したときに、自動的にアプリケーションを終了します。
/d	DRIVERDETECT	DriverDetect (REG_SZ)	0 に設定すると、デバイス ドライバの検出が無効になります。

スタンドアロン モードで CETK テストを実行する

開発ワークステーション上で Clientside.exe は CETest.exe に接続しますが、接続なしでも CETK テストを実行することは可能で、接続の可能性のないデバイスを扱う際に特に役に立ちます。Windows Embedded CE テスト キット カタログ アイテムをランタイム イメージに含めると、テスト エンジン (Tux.exe) を直接開始でき、暗黙的に Kato ロギング エンジン (Kato.exe) を開始してログ ファイルのテスト結果を追跡します。例えば、マウス テスト (mousetest.dll) を実行し test_results.log と呼ばれるファイル内の結果を追跡するには、次のコマンドラインを使用できます。

```
Tux.exe -o -d mousetest -f test_results.log
```

ノート　Tux コマンドライン パラメータ

Tux.exe コマンドライン パラメータの完全なリストについては、http://msdn2.microsoft.com/en-us/library/aa934656.aspx の Microsoft MSDN Web サイトにある、Windows Embedded CE 6.0 ドキュメントの「Tux Command-Line Parameters」セクションを参照してください。

カスタム CETK テスト ソリューションを作成する

CETK には大量のテストが含まれますが、既定のテストはすべてのテスト要件を満たすことはできません。独自のカスタム デバイス ドライバを BSP に追加した場合には特にそうです。カスタム ドライバ向けにユーザー定義テストを実装するオプションを提供するには、CETK は Tux フレームワークが必要です。Platform Builder には、数回のマウスのクリックで、スケルトン Tux モジュールを作成する、WCE TUX DLL テンプレートが含まれます。ロジックをドライバを動作させるために実装すると、既存のテスト実装のソース コードを確認するのに便利です。CETK にはソース コードが含まれており、Windows Embedded CE のセットアップ ウィザードで Windows Embedded CE 用共有ソースの一部としてインストールすることができます。既定の場所は、%_WINCEROOT%\Private\Test です。

カスタム Tux モジュールを作成する

Tux フレームワークと互換性のあるカスタム テスト ライブラリを作成するため、サブプロジェクトをランタイム イメージの OS デザインに追加することで Windows Embedded CE サブプロジェクト ウィザードを開始し、WCE TUX DLL テンプレートを選択します。これにより、Tux ウィザードは、ドライバ要件に応じてカスタマイズ可能なスケルトンを作成します。

サブプロジェクトで次のファイルを編集して、スケルトン Tux モジュールをカスタマイズする必要があります。

- ヘッダー ファイル Ft.h　関数テーブル ヘッダーおよび関数テーブル エントリが含まれ、TUX 関数テーブル (TFT) を定義します。関数テーブル エントリは、テスト ID をテスト ロジックを含む関数と関連付けます。
- ソース コード ファイル Test.cpp　テスト関数を含みます。スケルトン Tux モジュールは、参照として使用して Tux DLL へのカスタム テストを追加するのに使用可能な、単一の TextProc 関数を含みます。テストが完了したら、サンプル コードを置き換えてカスタム ドライバのロードおよび動作、Kato を介したアクティビティのログ記録、Tux テスト エンジンへの適切なステータス コードの返送を行うことができます。

CETK テスト アプリケーションでカスタム テストを定義する

スケルトン Tux モジュールは完全に機能することができ、コードを変更することなく、ソリューションのコンパイルおよびランタイム イメージのビルドが可能です。新しいテスト関数をターゲット デバイスで実行するには、ユーザー定義テストを CETK ワークステーション サーバー アプリケーションで構成する必要があります。この目的で、CETK には、[テスト] メニューで [ユーザー定義] コマンドをクリックすることで開始可能な [ユーザー定義テスト ウィザード] を含めます。図 4ミ12 は、[ユーザー定義テスト ウィザード] と構成パラメータを示しており、スケルトン Tux モジュールを実行します。

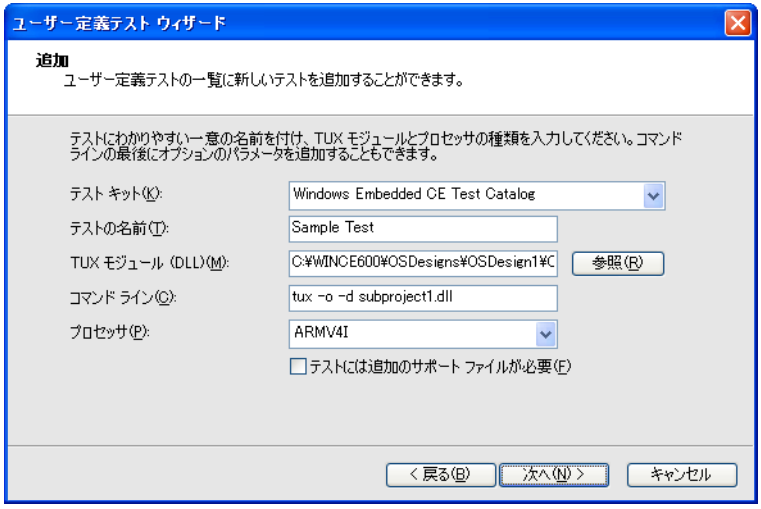


図 4-12　[ユーザー定義テスト ウィザード] でカスタム テストを構成する

カスタム テストをデバッグする

Tux テストは、Tux DLL に実装されたコードとロジックに依存しているため、テスト コードのデバッグに必要です。言及する必要のある 1 つの点としては、テスト ルーチンでブレークポイントを設定できますが、それらのブレークポイントでコード実行が中断されると、クライアント側アプリケーション (Clientside.exe) とワークステーション サーバー アプリケーション (CEText.exe) の間の接続が失われることです。ブレークポイントではなく、デバッグ メッセージを使用することを検討してください。徹底的なデバッグのためにブレークポイントを使用する必要がある場合、このレッスンで前述したように、ターゲット デバイス上でスタンドアロン モードで Tux.exe を直接実行します。テストを右クリックして、[コマンドラインの編集] を選択したときに、ワークステーション サーバー アプリケーションの必要なコマンド ラインを表示できます。

CETK テスト結果を分析する

CETK テストでは、スケルトン Tux モジュールの使用例で示されているように、テスト結果のログ記録に Kato を使用する必要があります。

```
g_pKato->Log(LOG_COMMENT, TEXT("This test is not yet implemented."));
```

ワークステーション　サーバー　アプリケーションは、これらのログを Clientside.exe を介して自動的に取得し、開発ワークステーションに保存します。他のツールを使用してこれらのログ ファイルにアクセスすることもできます。例えば、スタンドアロン方式で CETK を使用する場合、ファイル ビューア リモート ツールを使用して、ログ ファイルを開発ワークステーションにインポートできます。

CETK には、C:\Program Files\Microsoft Platform Builder\6.00\Cepb\Wcetk フォルダに、一般的な CETK パーサー (Cetkpar.exe) が含まれており、図 4-13 に示されているように、インポートされたログ ファイルを表示するのに便利です。通常、ワークステーション サーバー アプリケーションで完了したテストを右クリックし、[結果の表示] を選択することでこのパーサーを開始できます。また、直接 Cetkpar.exe を開始することもできます。特に PerfLog.dll に基づくパフォーマンス テストなどの、いくつかのテストは、カンマ区切り (CSV) 形式に解析し、スプレッドシートで開いてパフォーマンス データを確認することができます。この目的で、CETK には PerfToCsv パーサー ツールが含まれており、特殊な分析条件においてカスタム パーサーを開発できます。Kato ログ ファイルは、プレーン テキスト形式を使用します。

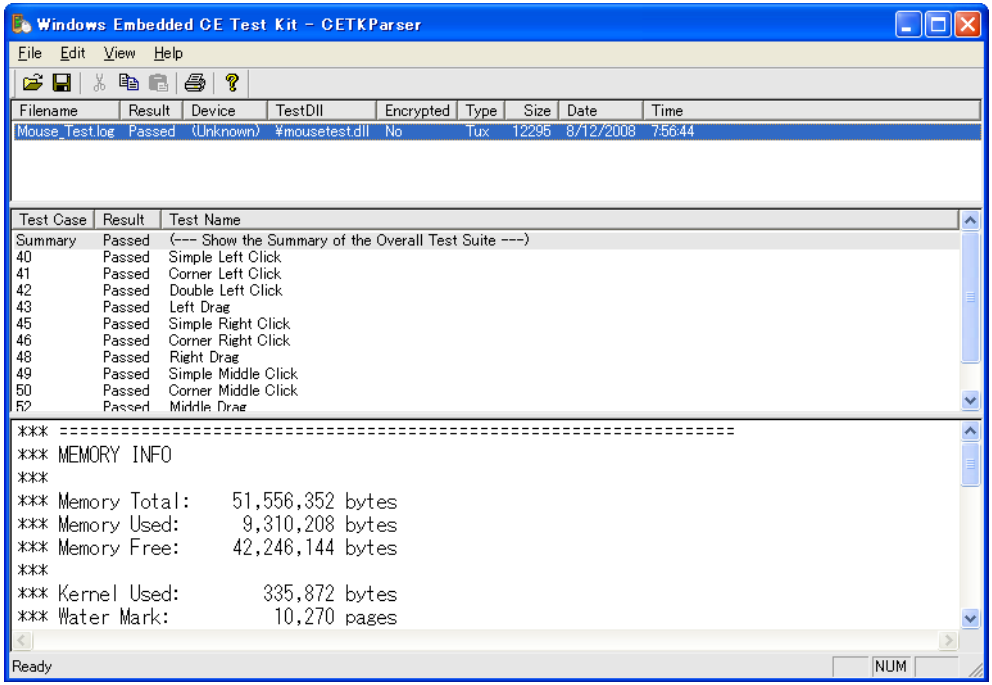


図 4-13 CETK テスト結果を分析する

レッスン概要

Windows Embedded CE テスト キットは、ターゲット デバイス上でドライバおよびアプリケーションを接続モードおよびスタンドアロン モードでテストできるようにする、拡張可能ツールです。ターゲット デバイスが KITL、ActiveSync、または TCP/IP を介した接続をサポートしない場合に、スタンドアロン モードで CETK ツールを実行するが便利です。最も一般的には、開発者は CETK を使用して、ターゲット デバイスの BSP に追加されたデバイス ドライバをテストします。

CETK は、すべてのテスト DLL の共通フレームワークを提供する、Tux テスト エンジンに依存しています。Tux DLL には実際のテスト ロジックが含まれており、ターゲットデバイスで実行してドライバをロードおよび動作させます。Tux DLL は、ログ ファイルのテスト結果を追跡するための Kato のインターフェイスも提供しています。カスタム パーサーやスプレッドシートなどの別のツールで CETK テスト アプリケーションやプロセスに直接アクセスすることができます。

レッスン 4：ブート ローダーをテストする

ブート ローダーの一般的なタスクは、デバイスに電源を供給した後に、カーネルをメモリにロードしてから、OS スタートアップ ルーチンを呼び出すことです。Windows Embedded CE では明確に、ブート ローダーはボード サポート パッケージ (BSP) の一部であり、コア ハードウェア プラットフォームの初期化、ランタイム イメージのダウンロードおよびカーネルの開始を担当します。ブート ローダーを最終製品に含めるつもりがなく、直接ブートストラップをランタイム イメージに含めるつもりでも、ブート ローダーは開発サイクルで大いに役立つはずです。他にも、ブート ローダーはラインタイム イメージ展開の複雑さを簡単化します。ランタイム イメージをイーサネット接続、シリアル ケーブル、DMA、または USB 接続を介して、開発コンピュータからダウンロードするのは、開発時間を接続するのに役立つ便利な機能です。Windows Embedded CE 6.0 の Platform Builder に含まれるソース コードに基づいて、カスタム ブート ローダーを開発して、新しいハードウェアや機能をサポートするようにできます。例えば、ブート ローダーを使用して RAM からランタイム イメージをフラッシュ メモリにコピーして、別個のフラッシュ メモリ プログラマまたは IEEE (Institute of Electrical and Electronic Engineers) 1149.1 互換テスト アクセス ポートや境界スキャン テクノロジを除去することができます。ただし、ブート ローダーのデバッグおよびテストは、カーネルがロードされる前に実行されるコードに対して作業しているため、複雑な作業となります。

このレッスンの後、次のことができるようになります。

- CE ブート ローダー アーキテクチャを説明する
- ブート ローダーの一連の共通デバッグ テクニックを理解する

レッスン時間 (推定) : 15 分

CE ブート ローダー アーキテクチャ

ブート ローダーの根本的な考え方は、線形で、非揮発性の CPU アクセス可能メモリにある小さなブートストラップ プログラムをブート前のルーチンで起動することです。初期ブート ローダー イメージを、ターゲット デバイスのボード製造業者または JTAG プローブによって提供された組み込み監視プログラムを介して CPU がコードの取得を開始するメモリ アドレスに置くと、ブート ローダーはシステムの起動またはリセット時に実行されます。通常のブート ローダー タスクは、この段階で実行されます。これには、中央処理装置 (CPU)、メモリ コントローラ、システム クロック、UART (Universal Asynchronous Receiver/

Transmitter)、イーサネット コントローラ、および他のハードウェアデバイスなどの初期化、ランタイム イメージのダウンロードとバイナリ イメージ ビルダー (BIB) レイアウトに基づいた RAM へのコピー、StartUp 関数へのジャンプなどが含まれます。ランタイム イメージの最新の記録には、この関数の開始アドレスが含まれます。StartUp 関数は、カーネル初期化ルーチンを呼び出すことにより、ブート プロセスを続行します。

多様なブート ローダー実装はその複雑さや実行するタスクが異なりますが、図 4-14 に示すように、Windows Embedded CE が提供する共通の特徴は、静的ライブラリによってブート ローダー開発を利用することです。結果のブート ローダー アーキテクチャは、ブート ローダー コードをデバッグする方法に影響します。ブート ローダー開発の詳細については、第 5 章「ボード サポート パッケージのカスタマイズ」を参照してください。

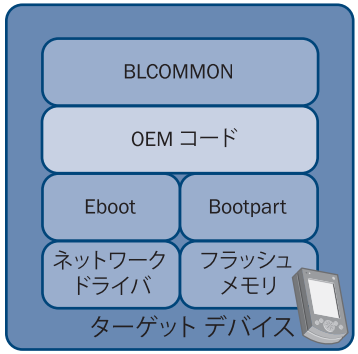


図 4-14　Windows Embedded CE ブート ローダー アーキテクチャ

Windows Embedded CE ブート ローダー アーキテクチャは、次のコード部分およびライブラリに基づいています。

- BLCOMMON　基本ブート ローダー フレームワークを実装して、より高速な実行のためのフラッシュ メモリから RAM へのブート ローダーのコピー、イメージ ファイルのコンテンツのデコード、チェックサムの検証、およびロード進捗の追跡の持続を行います。BLCOMMON は、明確な OEM 関数をプロセス全体で呼び出し、ハードウェア固有のカスタマイズを扱います。
- OEM コード　OEM が BLCOMMON ライブラリをサポートするために、ハードウェア プラットフォームに実装する必要のあるコードです。

- Eboot　動的ホスト構成プロトコル (DHCP)、簡易ファイル転送プロトコル (TFTP)、およびユーザー データグラム プロトコル (UDP) サービスを提供して、イーサネット接続を介してランタイム イメージをダウンロードします。
- Bootpart　メモリ分割ルーチンを提供して、ブート　ローダーがバイナリ ROM イメージ ファイル システム (BinFS) パーティションおよび同一記憶デバイス上の他のファイル システムが使用されているセカンド パーティションを作成します。また、Bootpart でブート パーティションを作成して、ブート パラメータを保存できます。
- ネットワーク ドライバ　多様な共通ネットワーク コントローラ デバイス用の基本初期化およびアクセス基本要素をカプセル化します。ライブラリのインターフェイスは汎用的であるため、ブート ローダーおよび OS は両方ともそのインターフェイスを使用できます。ブート ローダーは、インターフェイスを使用してランタイム イメージのダウンロードを行い、OS は、インターフェイスを使用して Platform Builder への KITL の接続を確立します。

ブート ローダーのデバッグ テクニック

ブート ローダーの設計は、通常、2 つの別個の部分で構成されています。最初の部分は、アセンブリ言語で記述され、C 言語で記述されている 2 番目の部分にジャンプする前にシステムを初期化します。図　4-14　に示すように、BLCOMMON ベースのアーキテクチャを使用している場合、アセンブリ コードをデバッグする必要はありません。デバイスに UART がある場合、C コードで RETAILMSG マクロを使用して、データをシリアル出力インターフェイスを介してユーザーに表示できます。

アセンブリ コードをデバッグする必要があるか、C コードをデバッグする必要があるかに応じて、次のデバッグ テクニックを使用できます。

- アセンブリ コード　7 セグメント LED のあるデバッグ ボードなどの、LED やシリアル通信インターフェイスの UART に依存した初期スタートアップ コード用の共通デバッグ テクニックを使用します。これは、汎用 I/O (General Purpose Input/Output: GPIO) レジスタにアクセスして入出力ラインの状態を変更するほうが比較的分かりやすいためです。
- C コード　C コード レベルでは、詳細通信インターフェイスおよびデバッグ マクロにアクセスできるため、デバッグはより容易です。

- アセンブリおよび C コード　ハードウェア デバッガ (JTAG プローブ) が使用可能な場合、Platform Builder を eXDI ドライバと併用して、ブート リーダーのデバッグを行うことができます。

試験のヒント
認証試験に合格するには、ブート ローダー、カーネル、デバイス ドライバ、およびアプリケーションをデバッグするための異なるテクニックを熟知しているかが重要です。

レッスン概要

ブート ローダーのデバッグは複雑なタスクで、ハードウェア プラットフォームの十分な理解が必要です。ハードウェア デバッガが使用可能な場合、Platform Builder を eXDI ドライバと併用して、ハードウェア補助デバッグを行います。ハードウェア デバッガが使用可能でない場合、アセンブリ コードおよび C 形式マクロのデバッグに LED ボードを使用して、シリアル通信インターフェイスを介してデバッグ メッセージを C コードで出力します。

演習 4：KITL、デバッグ領域、および CETK ツールに基づいたシステム デバッグおよびテスト

本演習では、デバイス エミュレータ BSP に基づいて OS デザインにサブプロジェクトとして追加されたコンソール アプリケーションのデバッグを行います。デバッグを有効にするには、KdStub および KITL をランタイム イメージに含め、対応するターゲット デバイス接続オプションを構成します。次いで、コンソール アプリケーションのソース コードを変更してデバッグ領域のサポートを実装し、ペガソス レジストリ キーで初期のアクティブなデバッグ領域を指定し、ターゲット デバイスをカーネル デバッガに接続して、Visual Studio の [出力] ウィンドウでデバッグ メッセージを検査します。その後、CETK を使用して、ランタイム イメージに含まれているマウス ドライバをテストします。Visual Studio で初期 OS デザインを作成するには、演習 1「OS デザインの作成、構成、およびビルド」に概略されている手順に従ってください。

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するために、この本の付属物中のドキュメント「演習 4 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。

- KITL の有効化およびデバッグ領域の使用

1. Visual Studio で、演習 1 で作成された OS デザイン プロジェクトを開き、OS デザイン名を右クリックしてから、OS デザイン プロパティを編集するための [プロパティ] を選択し、[構成プロパティ] を選択してから、[ビルド オプション] を選択し、ラインタイム イメージ用に [KITL を有効にする] チェック ボックスを選択します。
2. OS デザインのプロパティ ページのダイアログ ボックスで、[カーネル デバッガ] 機能を有効にし、変更を適用し、ダイアログ ボックスを閉じます。
3. 前の手順で有効にした KITL およびカーネル デバッガ コンポーネントを含むイメージをビルドするため、現在作業しているデバッグ ビルド構成を確認します。
4. [ビルド] メニューの [詳細なビルド コマンド] にある [現在の BSP およびサブプロジェクトのリビルド] を選択し、OS デザインをビルドします (後続の手順でエラーが発生した場合は、[クリーン システム生成] を実行します)。

5. [ターゲット] メニューを開き、[接続オプション] をクリックして、[ターゲット デバイスの接続オプション] ダイアログ ボックスを表示します。次の設定を確認し、[OK] をクリックします。

構成パラメータ	設定
ダウンロード	デバイス エミュレータ (DMA)
トランスポート	デバイス エミュレータ (DMA)
デバッガ	KdStub

6. サブプロジェクトを OS デザインに追加し、[WCE コンソール アプリケーション] テンプレートを選択します。プロジェクトに TestDbgZones という名前を付け、CE サブプロジェクト ウィザードで [標準的な "Hello World" アプリケーション] オプションを選択します。
7. DbgZone.h という名前の新しいヘッダー ファイルをサブプロジェクトに追加し、次の領域を定義します。

```
#include <DBGAPI.H>

#define DEBUGMASK(n)        (0x00000001<<n)
#define MASK_INIT           DEBUGMASK(0)
#define MASK_DEINIT         DEBUGMASK(1)
#define MASK_ON             DEBUGMASK(2)
#define MASK_ZONE3          DEBUGMASK(3)
#define MASK_ZONE4          DEBUGMASK(4)
#define MASK_ZONE5          DEBUGMASK(5)
#define MASK_ZONE6          DEBUGMASK(6)
#define MASK_ZONE7          DEBUGMASK(7)
#define MASK_ZONE8          DEBUGMASK(8)
#define MASK_ZONE9          DEBUGMASK(9)
#define MASK_ZONE10         DEBUGMASK(10)
#define MASK_ZONE11         DEBUGMASK(11)
#define MASK_ZONE12         DEBUGMASK(12)
#define MASK_FAILURE        DEBUGMASK(13)
#define MASK_WARNING        DEBUGMASK(14)
#define MASK_ERROR          DEBUGMASK(15)

#define ZONE_INIT           DEBUGZONE(0)
#define ZONE_DEINIT         DEBUGZONE(1)
#define ZONE_ON             DEBUGZONE(2)
#define ZONE_3              DEBUGZONE(3)
#define ZONE_4              DEBUGZONE(4)
#define ZONE_5              DEBUGZONE(5)
#define ZONE_6              DEBUGZONE(6)
#define ZONE_7              DEBUGZONE(7)
#define ZONE_8              DEBUGZONE(8)
#define ZONE_9              DEBUGZONE(9)
```

```
#define ZONE_10             DEBUGZONE(10)
#define ZONE_11             DEBUGZONE(11)
#define ZONE_12             DEBUGZONE(12)
#define ZONE_FAILURE        DEBUGZONE(13)
#define ZONE_WARNING        DEBUGZONE(14)
#define ZONE_ERROR          DEBUGZONE(15)
```

8. DbgZone.h ヘッダー ファイルの include 式を TestDbgZones.c ファイルに追加します。

```
#include "DbgZone.h"
```

9. 次のように、_tmain 関数の上に、デバッグ領域の dpCurSettings 変数を定義します。

```
DBGPARAM dpCurSettings =
{
  TEXT("TestDbgZone"),
  {
    TEXT("Init"), TEXT("Deinit"), TEXT("On"), TEXT("n/a"),
    TEXT("n/a"), TEXT("n/a"), TEXT("n/a"), TEXT("n/a"),
    TEXT("n/a"), TEXT("n/a"), TEXT("n/a"), TEXT("n/a"),
    TEXT("n/a"), TEXT("Failure"), TEXT("Warning"), TEXT("Error")
  },
  MASK_INIT | MASK_ON | MASK_ERROR
};
```

10. _tmain 関数の先頭行に、モジュールのデバッグ領域を登録します。

```
DEBUGREGISTER(NULL);
```

11. RETAILMSG および DEBUGMSG マクロを使用して、デバッグ メッセージを表示し、それを次のようにデバッグ領域に関連付けます。

```
DEBUGMSG(ZONE_INIT,
        (TEXT("Message :ZONE_INIT")));
RETAILMSG(ZONE_FAILURE || ZONE_WARNING,
         (TEXT("Message :ZONE_FAILURE || ZONE_WARNING")));
DEBUGMSG(ZONE_DEINIT && ZONE_ON,
        (TEXT("Message :ZONE_DEINIT && ZONE_ON")));
```

12. アプリケーションをビルドし、ターゲット デバイスに接続してから、[ターゲット コントロール] ウィンドウを使用してアプリケーションを開始します。

13. 最初のメッセージのみがデバッガ [出力] ウィンドウに表示されます。

```
4294890680 PID:3c50002 TID:3c60002 Message : ZONE_INIT
```

14. 開発コンピュータでレジストリ エディタ (Regedit.exe) を開くと、既定では、残りのデバッグ領域が有効になります。

15. HKEY_CURRENT_USER\Pegasus\Zones キーを開き、TestDbgZone と呼ばれる REG_DWORD 値を作成します (dpCurSettings 変数で定義されたモジュールの名前に依存)。
16. すべての 16 個の領域を有効にするには、値を 0xFFFF に設定します。これは 32 ビット DWORD 値では、下位 16 ビットに相当します (図 4-15 参照)。
17. Visual Studio で、アプリケーションを再度起動し、次の出力に注意します。

```
4294911331 PID:2270006 TID:2280006 Message : ZONE_INIT
4294911336 PID:2270006 TID:2280006 Message : ZONE_FAILURE || ZONE_WARNING
4294911336 PID:2270006 TID:2280006 Message : ZONE_DEINIT && ZONE_ON
```

18. レジストリで TestDbgZone 値を変更して、異なるデバッグ領域の有効化および無効化を行い、Visual Studio の [出力] ウィンドウで結果の検証を行います。

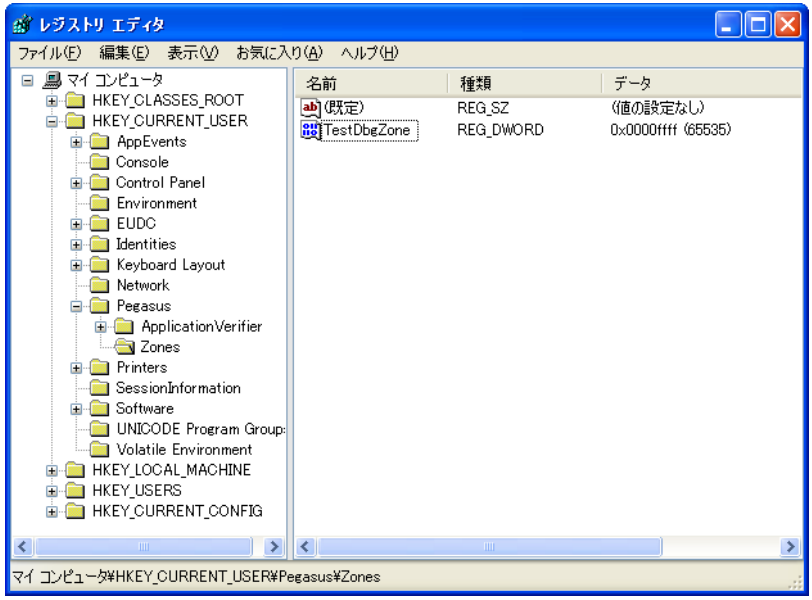


図 4-15 HKEY_CURRENT_USER\Pegasus\Zones: "TestDbgZone"=dword:FFFF

ノート Platform Builder でデバッグ領域を有効化および無効化する

Platform Builder で TestDbgZone モジュールのデバッグ領域をコントロールすることはできません。このモジュールの有効なゾーンを開いて変更できるようにする前に、そのアプリケーション プロセスが終了しているためです。Platform Builder では、グラフィカル アプリケーションや DLL など、ロードされたモジュールのデバッグ領域のみ管理できます。

- CETK を使用してマウス ドライバ テストを実行する
  1. 開発コンピュータの [スタート] メニューから、Windows CE テスト キット アプリケーションを開きます (Windows Embedded CE 6.0 メニューを開き、[Windows Embedded CE テスト キット] をクリックします)。
  2. [Windows Embedded CE テスト キット] ウィンドウで、[接続] メニューを開き、[クライアントの開始] をクリックして、ターゲット デバイスへの接続を確立します。
  3. [接続] をクリックし、[接続マネージャ] ウィンドウでデバイスを選択します。
  4. 図 4-16 に示すように、ワークステーション サーバー アプリケーションの、デバイスへの接続の成功、必要な CETK バイナリの展開、使用可能なデバイス ドライバの検出、すべてのコンポーネント一覧の階層ツリーによる表示が行われていることを確認します。
  5. [Windows CE テスト カタログ] を右クリックして、[すべてのテストの選択を解除] をクリックします。
  6. リストで各ノードを開き、[マウス テスト] チェック ボックスを選択します。
  7. [テスト] メニューを開き、[テストの開始 / 停止] をクリックして、マウス テストを実行します。
  8. ターゲット デバイス上で、必要なマウス動作が実行されます。
  9. テストを完了すると、テスト エントリを右クリックして [結果の表示] を選択することで、テスト レポートにアクセスできます。
  10. CETK パーサーで結果を調べて、成功、スキップ、および失敗したテスト手順を確認します。

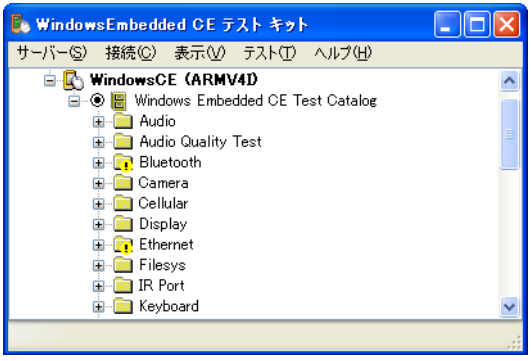


図 4-16　[Windows Embedded CE テスト キット] ウィンドウのデバイス カテゴリ

本章のレビュー

Windows Embedded CE 用 Platform Builder は、包括的な一連のデバッグおよびテスト ツールを提供しており、エラーの根本的な原因の診断および除去、製品のリリース前に最終構成でのシステムの検証を実行できるようにします。デバッグ ツールは、Visual Studio とともに統合されており、KITL 接続を介してターゲット デバイスと通信します。他にも、メモリ ダンプを作成し、[CE ダンプ ファイル リーダー] を使用してオフライン モードでシステムのデバッグを行うことができます。これは、エラー発生後のデバッグに特に役立ちます。デバッグ環境は、eXDI ドライバによって拡張することも可能で、標準カーネル デバッガの機能を超えて、ハードウェア補助デバッグを実行することができます。

カーネル デバッガは、カーネル コンポーネントおよびアプリケーション用のハイブリッド デバッガです。ターゲット デバイスに KdStub および KITL を有効にして接続している場合、デバッグは自動的に開始します。[ターゲット コントロール] ウィンドウを使用して、デバッグ用のアプリケーションを開始し、CEDebugX コマンドに基づいた、詳細なシステム テストを実行できます。ただし、ブレークポイントを割り込みハンドラや OAL モジュールに設定できないことに留意することは重要です。これは、これらのレベルでは、カーネルは単一スレッド モードで動作しており、コード実行が中断されると開発ワークステーションとの通信が停止されてしまうためです。割り込みハンドラをデバッグするには、ハードウェア デバッガかデバッグ メッセージを使用します。デバッグ メッセージ機能はデバッグ領域をサポートしており、ランタイム イメージをリビルドすることなく、情報出力をコントロールします。デバッグ メッセージを使用して、ブート ローダーの C コード部分をデバッグしますが、アセンブリ コード部分はハードウェア デバッガか LED パネルを使用する必要があります。

CETK テストをスタンドアロン モードで実行することもできますが、CETK テスト アプリケーションに基づいてシステム テストを中央集中化したい場合、KITL は必須となります。ターゲット デバイス用にカスタム BSP を開発する場合、CETK を使用して自動または半自動コンポーネント テストをカスタム Tux DLL に基づいて実行できます。Platform Builder には、WCE TUX ダイナミック リンク ライブラリ テンプレートが含まれており、特定のテスト要件に合致するように拡張可能なスケルトン Tux モジュールを作成できます。カスタム Tux DLL を CETK テスト アプリケーションで統合して、個別またはより大規模なテスト スイートの一部としてテストを実行できます。すべての CETK テストは同一のログ記録エンジンとログ ファイル形式を使用するため、同一のパーサー ツールを使用して、既定およびユーザー定義テストの結果の分析を行うことができます。

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。

- デバッグ領域
- KITL
- ハードウェア デバッガ
- dpCurSettings
- DebugX
- ターゲット コントロール
- Tux
- Kato

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。

メモリ リークの検出

メモリ ブロックを割り当て、決してそれらを開放しないことにより、メモリ リークを生成するコンソール アプリケーション用にサブプロジェクトを OS デザインに追加します。この章で取り上げたツールを使用することで、問題を特定し修正します。

カスタム CETK テスト

WCE TUX ダイナミック リンク ライブラリ用にサブジェクトを OS デザインに追加します。Tux DLL をビルドし、Windows Embedded CE テスト キット アプリケーションに登録します。CETK テストを実行し、テスト結果を検証します。自身の Tux DLL でブレークポイントを設定し、スタンドアロン モードで CETK テストを実行することでコードをデバッグします。

第 5 章

ボード サポート パッケージのカスタマイズ

アプリケーションの開発者はいつもボード サポート パッケージ (BSP) を作成する必要はありません。しかし、OEM が Microsoft ィ Windows ィ Embedded CE 6.0 R2 を新しいハードウェア プラットフォームに移植するときに必要です。OEM がこの作業を効果的に行うために、Windows Embedded CE は「製品品質」OEM アダプテーション層 (PQOAL) アーキテクチャの機能を有しています。これはプロセッサ モデルと OAL 関数によって整理された OAL ライブラリに基づいて、コードの再利用を促進するものです。マイクロソフトは OEM 開発者にそれらの特定の要求を満たし、電源管理、パフォーマンスの最適化、入出力制御 (IOCTL) のテストされ、実証された製品の機能を十分に生かすために、既存の BSP を複製しカスタマイズすることを推奨しています。この章では PQOAL アーキテクチャを扱い、BSP の複製の方法を説明し、OEM 開発者が Windows Embedded CE を新しいハードウェア アーキテクチャやモデルに適用させるために実装する必要のある関数を列挙します。BSP を開発するつもりがないとしても、BSP をカスタマイズするうえでのさまざまな面を理解することには利点があります。この章では、BSP カスタマイズのいくつかの面の概要を示し、スタートアップ プロセスの変更とカーネル初期化ルーチンの実装からデバイス ドライバ、電源管理機能、またパフォーマンス最適化のサポートの追加までを網羅します。

本章の試験範囲：

- Windows Embedded CE の BSP アーキテクチャを理解する
- 特定のターゲット デバイスのために BSP とブート ローダーを変更し適用する
- メモリ管理とレイアウトを理解する
- BSP で電源管理を有効にする

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。

- Windows Embedded CE ソフトウェア開発に関する基本的な知識。
- 埋め込みデバイスのハードウェア　アーキテクチャについての徹底的な理解。
- 電源管理とそれをドライバとアプリケーションで実装することに関する基本的な知識。
- Microsoft Visual Studio 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：ボード サポート パッケージの適用と設定

一般的に新しいハードウェア プラットフォームの BSP 開発プロセスは、ROM モニタを使い、適切な参照 BSP を複製することによってハードウェアの機能テストを実行し、カーネルをサポートするためにブート ローダーとコア OAL 関数を実装した後に始めます。目標は、可能な限り少ないカスタム コードを使ってブート可能なシステムを作成することです。その後統合されたハードウェアや周辺機器のハードウェアをサポートするために BSP にデバイス ドライバを追加したり、ターゲット デバイスの性能に応じて電源管理や他の高度なオペレーティング システム (OS) の機能を実装することによってシステムを拡張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：

- PQOAL ベースのボード サポート パッケージのコンテンツを識別し検索する。
- ハードウェア固有また共通のコード ライブラリを識別する。
- BSP の複製の方法を理解する。
- ブート ローダ、OAL、デバイス ドライバを適用する。

レッスン時間 (推定)：40 分

ボード サポート パッケージ (BSP) の概要

BSP は与えられたプラットフォームのブート ローダー、OAL、またデバイス ドライバのためのすべてのソース コードを含んでいます。これらのコンポーネントに加えて、BSP は図 5-1 に示されているようにビルドまたシステム設定のファイルをも含んでいます。設定ファイルは実際のランタイム イメージには含まれていませんが、第 2 章「ランタイム イメージのビルドおよび展開」で説明されているようにソース コード ファイル、メモリ レイアウト、レジストリ設定や、ランタイム イメージをコンパイルしビルドするためのその他の面を特定するのに必要な BSP パッケージの一部です。

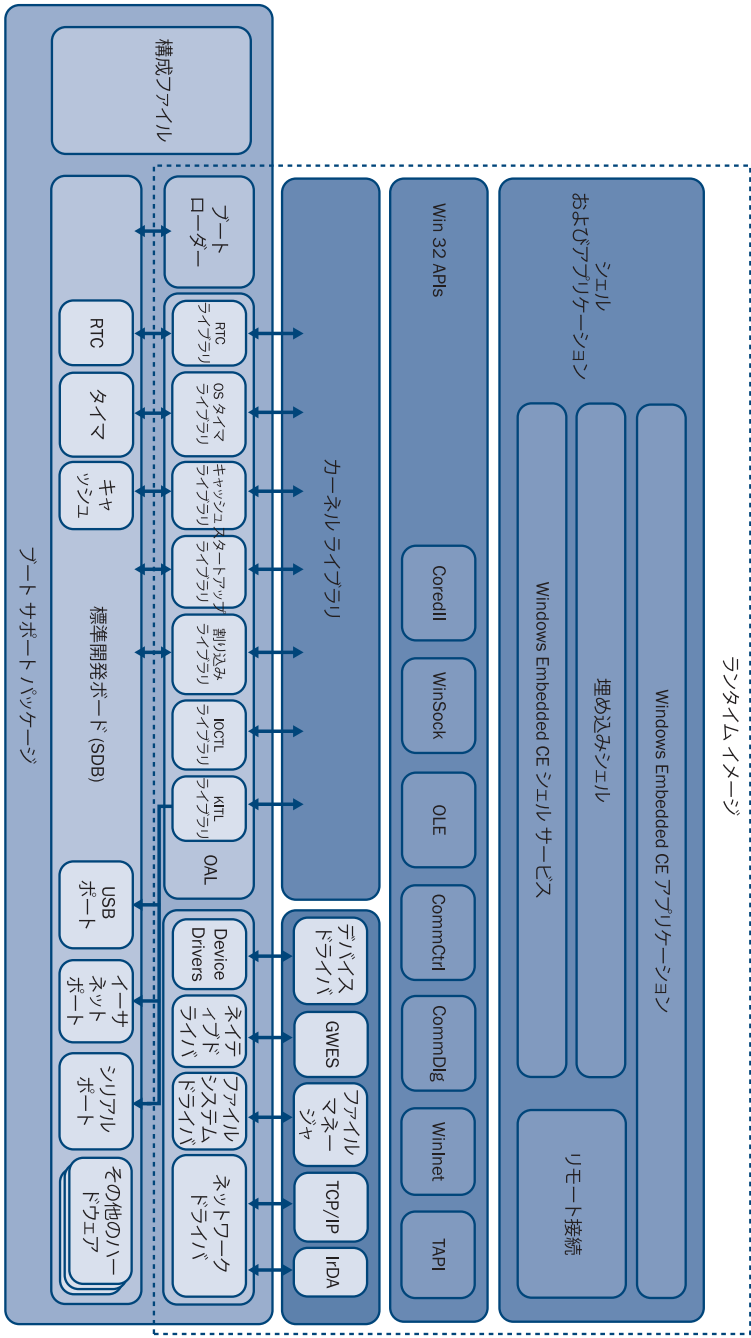


図 5-1　Windows Embedded CE 6.0 の残存成分との関係における BSP のコンポーネント

図 5-1 に示されるように、BSP 開発には以下の主要コンポーネントが含まれます。

- ブート　ローダー　デバイスの起動および再起動のときに実行されます。ブート ローダはハードウェア プラットフォームの初期化やオペレーティング システムへの実行の受け渡しを担当します。
- OEM アダプテーション層 (OAL)　BSP のコアに相当し、カーネルとハードウェアの間のインターフェイスとなります。それはカーネルと直接リンクしているため、CE ランタイム イメージでカーネルの一部となります。いくつかのコア カーネル コンポーネントは、スレッド スケジューラのための割り込み処理やタイマ処理のようなハードウェア初期化のために OAL に直接依存しています。
- デバイス ドライバ　特定の周辺機器の機能を管理し、デバイス ハードウェアとオペレーティング システムとの間のインターフェイスを提供します。第 6 章「デバイス ドライバの開発」で説明されているように、Windows Embedded CE はそれらが置かれる各々のインターフェイスに基づくさまざまなドライバ アーキテクチャをサポートします。
- 設定ファイル　ビルド　プロセスを制御するのに必要な情報を提供し、プラットフォーム オペレーティング システムのデザインにおいて重要な役割を果たします。BSP に含まれる典型的な設定ファイルは、Sources ファイル、Dirs ファイル、Config.bib、Platform.bib、Platform.reg、Platform.db、Platform.dat と、catalog ファイル (*.pbcxml) です。

ボード サポート パッケージの適用

一般的に、最初から BSP を作成するかわりに既存の参照 BSP を複製することによって BSP 開発プロセスを省略して開始するのはよいことです。もしまったく新しい CPU とまったく新しいプラットフォームのために BSP を開発しなければならないとしても、類似のプロセッサ アーキテクチャに基づいて BSP を複製することが依然として勧められています。このようにして、既存の BSP からハードウェア依存コードを再利用することによって BSP 開発時間を短縮でき、またそれらがマーケットで購入可能になることから将来的に新しい Windows Embedded バージョンへの移行周期を短縮できます。登録商標を持つ BSP はマイクロソフトが新しいオペレーティング システム バージョンの一部として暗黙のうちに移行しテストした PQOAL コード部分から利点を得られないため、登録商標を持つ BSP デザインの移行を行うのは一般的に PQOAL ベースのデザインを移行するよりだいぶ困難になります。

ボード サポート パッケージ の適用には以下のステップの手順が含まれます。

1. 参照 BSP の複製。
2. ブート ローダーの実装。
3. OAL 関数の適用。
4. ランタイム イメージ設定ファイルの変更。
5. デバイス ドライバの開発。

参照 BSP の複製

プラットフォーム ビルダは参照 BSP の複製を容易にするウィザードを含みます。このウィザードは選択された参照 BSP のすべてのソース コードを新しいフォルダ構造へとコピーし、ユーザーが参照 BSP や %_WINCEROOT% フォルダ階層の他の BSP に影響を及ぼさずに、新しいハードウェアのための BSP をカスタマイズできるようにします。図 5-2 は Microsoft Visual Studio 2005 で Platform Builder for Windows Embedded CE 6.0 を用いて BSP 複製ウィザードを開始する方法を示しています。

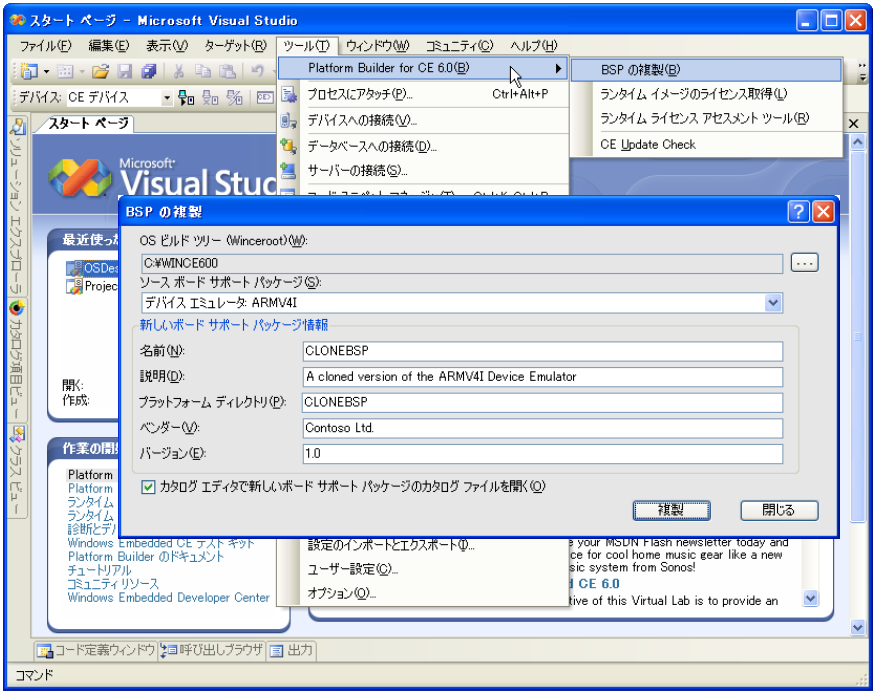


図 5-2　Visual Studio 2005 での BSP の複製

ノート　BSP 名

BSP を複製するときには、この新しい一連のファイルに新しい名前を選択する必要があります。プラットフォームのために選択した名前はハード ドライブのフォルダの名前と一致する必要があります。前述の章で記載されたように、ビルド エンジンはコマンド ラインの文字列に基づいており、フォルダ名にスペースがあるものとは互換性がありません。そのため BSP 名にはスペースを含んではなりません。代わりにアンダスコア (_) を使用することができます。

BSP フォルダ構造

コードの再利用性を向上するために、PQOAL ベースの BSP はプロセッサ ファミリで一致した共通アーキテクチャと対応フォルダ構造を機能として有しています。この共通アーキテクチャによりソース コードの大部分はハードウェア固有の BSP 要求条件にかかわらず再利用できます。図 5-3 は典型的 BSP フォルダ構造を示し、表 5-1 は最も重要な BSP フォルダを要約しています。

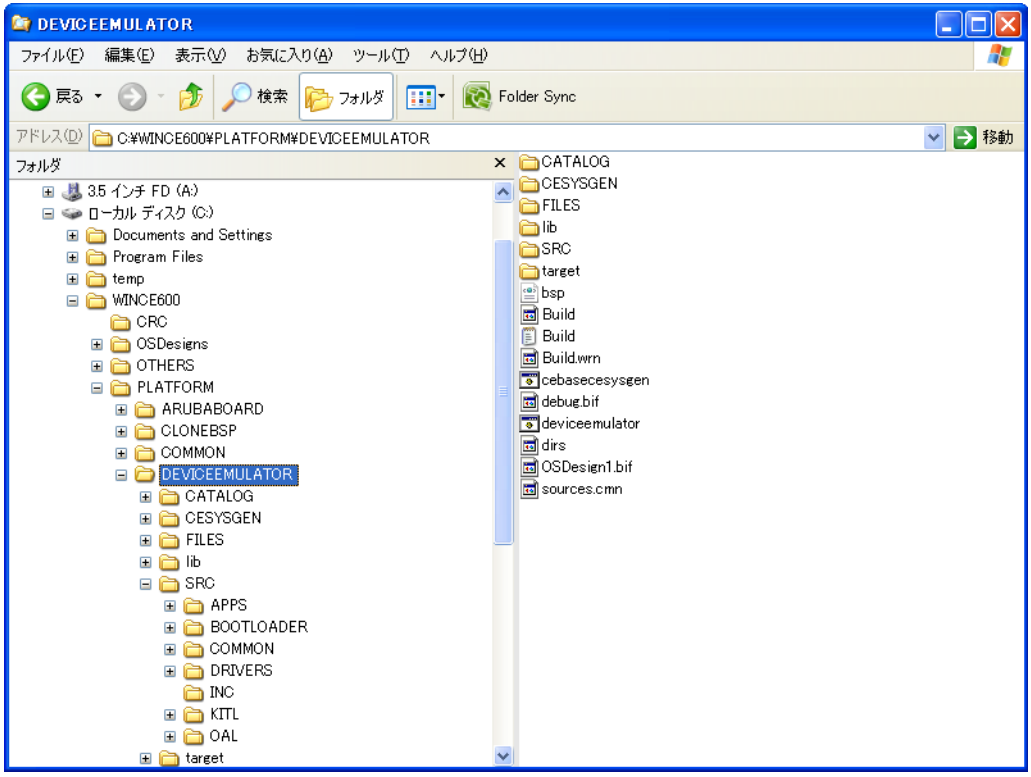


図 5-3　典型的な BSP のフォルダ構造

ヒント　%_TARGETPLATROOT%

BSP のパスを検索するためにビルド ウィンドウで環境変数 %_TARGETPLATROOT% を、現在の OS デザインで使用されていながら使用することができます (Visual Studio で [ビルド] メニューから [ビルド ウィンドウ オプション] をポイントし、[リリース ディレクトリ] をクリックします) 。

表 5-1　重要な BSP フォルダ

フォルダ	説明
ルート フォルダ	設定とバッチ ファイルを含みます。開発者にとって最も重要な 2 つのファイルは以下のとおりです : ■ Sources.cmn　全 BSP にわたって共通なマクロ定義を含みます。 ■ <*BSP Name*>.bat　デフォルト BSP 環境変数を設定します。
CATALOG	BSP のすべてのコンポーネントが定義される BSP カタログ ファイルを含みます。このファイルは OS デザイン段階で BSP の機能を加えたり取り除いたりするのに使われます。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」でカタログ項目を管理する方法を論じています。
CESYSGEN	sysgen ツールのメイク ファイルを含みます。BSP の設定ではこのディレクトリを変更しないよう要求します。
FILES	.bib、.reg、.db や、.dat ファイルなどのビルド設定ファイルを含みます。
SRC	PQOAL モデルによって適用しなければならないプラットフォーム固有のソース コードを含みます。これは CPU タイプによってコードをプラットフォーム固有と共通コンポーネントに分けるものです。
COMMON	プラットフォーム ディレクトリの下位階層にあり、ほとんどの BSP ソースコードを含みます。これは共通するひとそろいのプロセッサ固有のコンポーネントから成っています。BSP はこのフォルダ内でライブラリとリンクし、ビルド プロセスの間に生成されます。これらはプロセッサ ベースの周辺機器とプロセッサ固有の OAL パートのライブラリです。もしハードウェアがサポートされているプロセッサのファミリからの CPU を使用しているなら、これらのライブラリのほとんどは変更なしに再利用することができるでしょう。

プラットフォーム固有のソース コード

ユーザーが BSP の一部として適用しなければならない最も重要なプラットフォーム固有のソース コードは、ブート ローダー、OAL、デバイスドライバのためのものです。対応するソース コードは Src フォルダの下位、下記のサブディレクトリの中にあります。

- Src\Boot loader はブート ローダー コードを含みます。しかしながら、ブート ローダーが BLCOMMON や関係するライブラリに依存している場合、ブート ローダーの基本的なハードウェア固有部分のみがこのディレクトリに位置します。再利用可能なブート ローダー コードはパブリック フォルダ (%_WINCEROOT%\Public\Common\Oak\Drivers\Ethdbg) 内で入手可能で、BSP パートとライブラリとしてリンクしています。第 4 章「システムのデバッガおよびテスト」でブート ローダー開発を容易にする静的ライブラリを紹介しています。
- Src\Oal ハードウェア プラットフォームに固有の最小限のコードを含んでいます。大部分の OAL コードは %_WINCEROOT%\Platform\Common に位置し、ハードウェア依存、プロセッサ ファミリ関連、チップ セット固有、プラットフォーム固有のグループに分けられています。これらのコード グループはほとんどの OAL 機能を提供し、ライブラリとしてプラットフォーム固有部分にリンクされています。
- Src\Common and Src\Drivers ドライバ ソース コードを含み、メンテナンスと移植性を容易にするための、いくつかの異なったカテゴリに整理されています。これらのカテゴリは、通常、プロセッサ固有とプラットフォーム固有です。プロセッサ固有コンポーネントは Src\Common ディレクトリに位置し、同じプロセッサ ファミリに基づく新しいハードウェアに適用するときには変更を必要としません。

既存のライブラリからブート ローダーを実装する

ブート ローダーを新しいプラットフォームのために適用するときには、次のようないくつかの面を考慮する必要があります。

- プロセッサ アーキテクチャの変化。
- ターゲット デバイスでのブート ローダー コードの位置。
- プラットフォームのメモリ アーキテクチャ。
- ブート プロセス中に実行するタスク。

- ランタイム イメージのダウンロードのためのサポートされたトランスポート。
- サポートされなければならない追加機能。

メモリ マッピング

最も大切な適用タスクはブート ローダーのためのメモリのマッピングの定義に関するものです。Windows Embedded CE に含まれている標準 BSP はメモリ設定を .bib ファイルで定義しています。このファイルは %_WINCEROOT%\Platform\Arubaboard\Src\Boot loader\Eboot\Eboot.bib. のようにブート ローダー サブディレクトリに位置しています。以下のリストはユーザー固有の要求を満たすためにカスタマイズできる Eboot.bib ファイルの例を示します。

```
MEMORY

;   名前   スタート   サイズ   タイプ
;   -------  --------  --------  ----

    ; Eboot の前に RAM をいくらか保持する。
    ; このメモリは後で使用する。

DRV_GLB  A0008000  00001000  RESERVED   ; ドライバ グローバル。4 KB で十分。

    EBOOT    A0030000  00020000  RAMIMAGE   ; ローダーのために 128 KB を取り分ける。後で終了処理する。
    RAM      A0050000  00010000  RAM        ; フリー RAM。後で終了処理する。

CONFIG
    COMPRESSION=OFF
    PROFILE=OFF
    KERNELFIXUPS=ON

    ; これらの構成オプションが、.nb0 ファイルを作成させる。
    ; .nb0 ファイルは、直接フラッシュ メモリに書き込まれ、ブートされる
    ; 可能性あり。ローダーが RAM から実行するためにリンクされているので、
    ; 以下の設定オプションは RAMIMAGE
    ; セクションと一致する必要がある。
    ROMSTART=A0030000
    ROMWIDTH=32
    ROMSIZE=20000

MODULES
;   名前    パス    メモリ タイプ
;   -----------  ---------------------------------------- -----------
    nk.exe       $(_TARGETPLATROOT)\target\$(_TGTCPU)\$(WINCEDEBUG)\EBOOT.exe   EBOOT
```

ドライバ グローバル

また、ブート ローダーがセットアップ プロセスの間オペレーティング システムに情報を渡すためのメモリ セクションを取っておくために、Eboot.bib ファイルを使うこともできます。この情報は初期化されたハードウェアの現在の状況や、またもしブート ローダーがイーサネット ダウンロードやカーネル依存トランスポート層 (KITL) を有効にするようなオペレーティング システムのためのユーザーおよびシステム フラッグをサポートしているならば、そのネットワーク通信能力などが反映されることがあります。この通信を有効にするために、ブート ローダーとオペレーティング システムはドライバ グローバル (DRV_GLB) と呼ばれる物理メモリの共通領域を共有しなければなりません。上記の Eboot.bib リストは DRV_GLB マッピングをも含みます。DRV_GLB 領域でブート ローダーがオペレーティング システムに渡すデータは、ユーザー固有の要求によって定義された BOOT_ARGS 構造を守らなければなりません。

以下のプロシージャはイーサネットと IP の設定情報をブート ローダーから DRV_GLB 領域を通してオペレーティング システムに渡す方法を図解します。これを行うために、%_WINCEROOT%\Platform\<BSP 名 >\Src\Inc フォルダに Drv_glob.h などのヘッダ ファイルを作成し、以下の内容を書き込んでください。

```
#include <halether.h>

// デバッグ イーサネット パラメータ。
typedef struct _ETH_HARDWARE_SETTINGS
{
    EDBG_ADAPTER    Adapter;            // Platform Builder と通信するための NIC。
    UCHAR           ucEdbgAdapterType;  // デバッグ イーサネット アダプタのタイプ。
    UCHAR           ucEdbgIRQ;          // デバッグ イーサネット アダプタで使用する IRQ 行。
    DWORD           dwEdbgBaseAddr;     // デバッグ イーサネット アダプタのベース I/O アドレス。
    DWORD           dwEdbgDebugZone;    // 有効にする EDBG デバッグ ゾーン。

    // デバイス名を作成するためのベース。
    // これが Platform Builder にデバイスを認識させるために
    // 一意のデバイス名を生成するよう
    // EDBG MAC アドレスと結合される。
    char szPlatformString[EDBG_MAX_DEV_NAMELEN];

    UCHAR           ucCpuId;            // CPU のタイプ。
} ETH_HARDWARE_SETTINGS, *PETH_HARDWARE_SETTINGS;


// BootArgs - ブート ローダーから OS に渡されるパラメータ。
#define BOOTARG_SIG  0x544F4F42 // "BOOT"

typedef struct BOOT_ARGS
{
    DWORD   dwSig;
```

```
    DWORD   dwLen;              // BootArgs 構造の全長。
    UCHAR   ucLoaderFlags;      // ブート ローダーによって設定されたフラグ。
    UCHAR   ucEshellFlags;      // Eshell からのフラグ。
    DWORD   dwEdbgDebugZone;    // どのデバッグ メッセージが有効か。

    // 以下のアドレスは LDRFL_JUMPIMG が設定され、
    // ucEshellFlags 中の対応するビットが設定されている (Eshell によって設定され、
    // ビット定義が Ethdbg.h にある ) ときにだけ有効である。
    EDBG_ADDR EshellHostAddr;   // IP とイーサネットのアドレス、
                                // および Eshell を実行しているホストの UDP ポート。
    EDBG_ADDR DbgHostAddr;      // IP とイーサネットのアドレス、
                                // およびデバッガ メッセージを受信しているホストの UDP ポート。
    EDBG_ADDR CeshHostAddr;     // IP とイーサネットのアドレス、
                                // およびイーサネット テキスト シェルを実行しているホストの UDP ポート。
    EDBG_ADDR KdbgHostAddr;     // IP とイーサネットのアドレス、
                                // およびカーネル デバッガを実行しているホストの UDP ポート。

    ETH_HARDWARE_SETTINGS Edbg; // デバッグ イーサネット コントローラ。

} BOOT_ARGS, *PBOOT_ARGS;

// ブート ローダーによって設定されたフラグの定義。
#define    LDRFL_USE_EDBG    0x0001  // デバッグ イーサネットを使用するために設定。

// LDRFL_USE_EDBG が設定されている場合は、次の 2 つのフラグのみが参照される。
#define    LDRFL_ADDR_VALID  0x0002  // EdbgAddr メンバが有効な場合に設定。
#define    LDRFL_JUMPIMG     0x0004  // 設定されると、構成情報を取得するために
                                     // Eshell を使用しない。
                                     // 代わりに、ucEshellFlags メンバを使用。
typedef struct _DRIVER_GLOBALS
{
    //
    // TODO: 後でドライバと OS の間で共有されている情報を
    // この部分に入力する。
    //
    BOOT_ARGS        bootargs;
} DRIVER_GLOBALS, *PDRIVER_GLOBALS;
```

スタートアップ エントリ ポイントとメイン関数

このルーチンがハードウェアの初期化を行うため、ブート ローダーのスタートアップ エントリ ポイントは、CPU が実行のための取得コードを開始するアドレスにある線形メモリに位置しているべきです。もし適用が同じプロセッサ チップセットのための参照 BSP に基づいて行われるのであれば、ほとんどの CPU 関連やメモリ制御関連のコードは変更せずにおくことができます。反対に、もし CPU アーキテクチャが違うのなら、以下のタスクを実行することによってスタートアップ ルーチンを適用しなければなりません。

1. CPU を適切なモードにします。

2. すべての割り込みを無効にします。
3. メモリ コントローラを初期化します。
4. キャッシュ、トランスレーション ルックアサイド バッファ (TLB) 、メモリ管理ユニット (MMU) をセットアップします。
5. 実行の高速化のためにブート ローダーをフラッシュ メモリから RAM にコピーします。
6. メイン関数で C コードへ飛びます。

スタートアップ ルーチンは最後にブート ローダーのメイン関数を呼び出します。もしブート ローダーが BLCOMMON に基づいているならば、この関数が続いて BootLoaderMain を呼び出し、これが OEM プラットフォーム関数を呼び出すことによってダウンロード トランスポートを初期化します。マイクロソフトにより供給される標準ライブラリを使用することの利点は、新しいハードウェア プラットフォームへ BSP を適用するために要求される変更がコンポーネント化、単離化、最小化されていることにあります。

シリアル デバッガ出力

ブート ローダー適用の次のステップはシリアル デバッガ出力の初期化です。これはブート プロセスの重要な部分です。なぜならこれによって第 4 章「システムのデバッガおよびテスト」説明されているように、デバッガ メッセージを分析するためにブート ローダーと開発者の間の対話操作が有効になるからです。

図 5-2 は、ブート ローダーでシリアル デバッガ出力をサポートするために要求される OEM プラットフォーム関数を列挙しています。

図 5-2 シリアル デバッガ出力関数

関数	説明
OEMDebugInit	プラットフォームで UART を初期化します。
OEMWriteDebugString	デバッガ UART へ文字列を書き込みます。
OEMWriteDebugByte	デバッガ UART にバイトを書き込み、OEMWriteDebugString で使われます。
OEMReadDebugByte	デバッガ UART からのバイトを読み取ります。

プラットフォームの初期化

CPU とデバッガ シリアル出力が初期化されたなら、残っているハードウェア初期化タスクに注意を向けてください。OEMPlatformInit ルーチンがこれらの残っている次のようなタスクを実行します。

- リアル タイム クロックの初期化。
- 外部メモリと特定のフラッシュ メモリのセットアップ。
- ネットワーク コントローラの初期化。

イーサネットを介したダウンロード

もしハードウェア プラットフォームがネットワーク コントローラを含んでいるなら、ブート ローダーはイーサネットを経てランタイム イメージをダウンロードできます。図 5-3 はイーサネット ベースの通信をサポートするために実装しなければならない関数を列挙しています。

図 5-3　イーサネット サポート関数

関数	説明
OEMReadData	ダウンロードのためにトランスポートからデータを読み取ります。
OEMEthGetFrame	関数ポインタ pfnEDbgGetFrame を用いて、NIC からデータを読み取ります。
OEMEthSendFrame	関数ポインタ pfnEDbgSendFrame を用いて、NIC へデータを書き込みます。
OEMEthGetSecs	固定した時間と比較して経過した秒数を返します。

イーサネット サポート関数はネットワーク コントローラ固有のルーチンへのコールバックを使用します。これは、もし違ったネットワーク コントローラをサポートしたいならば、追加のルーチンを実装し、下記のサンプル コードに示されるように OEMPlatformInit 関数で適切な関数 ポインタをセットアップしなければならないことを意味します。

```
cAdaptType=pBootArgs->ucEdbgAdapterType;

// EDBG ドライバ コールバックをイーサネット
// コントローラ タイプに基づいて設定。
switch (cAdaptType)
{
case EDBG_ADAPTER_NE2000:
    pfnEDbgInit           = NE2000Init;
    pfnEDbgInitDMABuffer  = NULL;
```

```
    pfnEDbgGetFrame       = NE2000GetFrame;
    pfnEDbgSendFrame      = NE2000SendFrame;
    break;

case EDBG_ADAPTER_DP83815:
    pfnEDbgInit           = DP83815Init;
    pfnEDbgInitDMABuffer  = DP83815InitDMABuffer;
    pfnEDbgGetFrame       = DP83815GetFrame;
    ...
}
```

フラッシュ メモリ サポート

ネットワーク通信機能が実装されたので、ブート ローダーは新しいハードウェア プラットフォームにランタイム イメージをダウンロードして、それに制御を渡すことができます。または、ランタイム イメージをフラッシュ メモリに保存することもできます。図 5-4 はもし参照 BSP のブート ローダーがあらかじめそれらの機能をサポートしていないならば、これらを実行するために実装しなければならないダウンロードやフラッシュ メモリ サポート関数を列挙しています。

図 5-4 ダウンロードとフラッシュ メモリをサポートするための関数

関数	説明
OEMPreDownload	プラットフォーム ビルダによってサポートされる必要なダウンロード プロトコルをセットアップします。
OEMIsFlashAddr	イメージがフラッシュ用か RAM 用かを確認します。
OEMMapMemAddr	イメージの RAM への一時的な再マッピングを実行します。
OEMStartEraseFlash	OS イメージに合わせるために十分なサイズのフラッシュを削除する準備をします。
OEMContinueEraseFlash	ダウンロードの進行に基づいてフラッシュの削除を継続します。
OEMFinishEraseFlash	ダウンロードが終了したときにフラッシュの削除を完了します。
OEMWriteFlash	OS イメージをフラッシュへ書き込みます。

ユーザーの操作

ブート ローダーはユーザーにプラットフォームを開始するためのいくつかの選択肢を提供するメニューに基づくユーザーの操作をサポートでき、これは開発プロセス中や後にメンテナンスやソフトウェア アップデートの際に有用になります。図 5-4 は標準ブート ローダー メニューを示しています。サンプル ソース コードは参照 BSP の Src\Boot loader\Eboot ディレクトリや %_WINCEROOT%\Platform\Common\Src\Common\Boot\Blmenu フォルダにある Menu.c ファイルを確認してください。

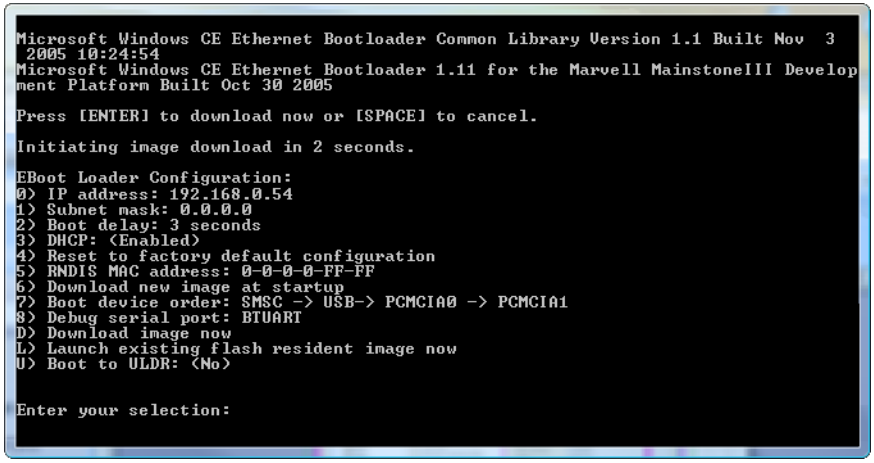


図 5-4　ブート ローダー メニューの例

追加機能

コア関数の他に、ダウンロード進行状況表示、同じダウンロード セッションの間に複数の .bin ファイルをダウンロードするためのサポート (マルチビン イメージ通知) 、信頼できるイメージのみのダウンロードなどの便利機能も加えることができます。さらに、プラットフォーム ビルダから直接ランタイム イメージのダウンロードのためのサポートを実装することもできます。このタスクを行うには、ブート ローダーが BOOTME パケットをターゲット デバイスに関する詳細とともに準備し、それを基になるトランスポートを通して送信しなければなりません。もしトランスポートがイーサネットであれば、このパケットはネットワークを通して一斉送信されます。マイクロソフトが提供するライブラリはこれらの機能をサポートしており、ユーザーは必要に合わせてそれらをカスタマイズできます。

ノート　OEM ブート ローダー関数

必須または任意のブート ローダー関数およびブート ローダー構造の詳細については、http://msdn2.microsoft.com/en-us/library/aa908395.aspx にある Microsoft MSDN? Web サイトの Windows Embedded CE 6.0 ドキュメントの「Boot Loader reference」セクションを参照してください。

OAL の適用

BSP 適用の一つの重要な部分は OAL のプラットフォーム固有部分に関わるものです。新しいプラットフォームが現在サポートされていない CPU を使っていたとしたら、OAL 適用の際、新しいプロセッサ アーキテクチャをサポートするためにほとんどの OAL コードを変更する必要があります。反対に、新しいハードウェアが参照 BSP プラットフォームによく似ているのであれば、ほとんどの既存コード ベースを再利用できるかもしれません。

OEM アドレス テーブル

カーネルは仮想メモリを初期化するといった特殊なタスクを実行します。カーネルは完全に自己制御でなければならないため、これらのタスクをブート ローダーに依存して行うことはできません。さもなければ、オペレーティング システムはブート ローダーの存在に依存することになり、ランタイム イメージを直接ブートストラップすることは不可能になってしまいます。しかし、メモリ管理ユニット (MMU) 全体にわたる仮想物理アドレス マッピングを構築するために、カーネルはハードウェア プラットフォームの基になるメモリ レイアウトの情報がなければなりません。この情報を得るために、カーネルは静的仮想メモリ領域を定義する OEMAddressTable (または g_oalAddressTable) と呼ばれる重要なテーブルを使います。OAL は読み取り専用セクションとしての OEMAddressTable の宣言を含んでおり、カーネルが取る最初の行動の一つがこのセクションを読み取り、対応する仮想メモリ マッピング テーブルをセットアップし、その後カーネルがコードを実行できる仮想アドレスに移行することです。カーネルはランタイム イメージで入手可能なアドレス情報に基づいてリニア メモリ中の OEMAddressTable の物理アドレスを決定することができます。

OEMAddressTable を変更することによって新しいハードウェア プラットフォームのメモリ設定のいかなる違いをも表示することができます。以下のサンプル コードは OEMAddressTable セクションの宣言の方法を示します。

```
;------------------------------------------------------------------
public      _OEMAddressTable

    _OEMAddressTable:

    ; OEMAddressTable は物理また仮想アドレスの間のマッピングを定義します。
    ; o は読み取り専用セクションにあるべきです。
    ; o 1 の最初の入力は RAM であるべきで、0x80000000 -> 0x00000000 からマッピングされます。
    ; o 各入力はフォーマットです ( VA, PA, cbSize )。
    ; o cb サイズ は 4 M の複数であるべきです。
    ; o の最後の入力は (0, 0, 0) であるべきです。
    ; o は少なくとも一つのゼロではない入力があるべきです。
    ; RAM 0x80000000 -> 0x00000000, サイズ 64 M
     dd   80000000h,     0,    04000000h
    ; フラッシュとその他のメモリ、
    ; dd フラッシュ VA,    フラッシュ PA,   フラッシュサイズ
    ; 最後の入力、すべてゼロ
     dd   0     0     0
```

StartUp エントリ ポイント

ブート ローダーと同様に、OAL は StartUp ポイントを含んでいて、ブート ローダーやシステムはカーネル実行を開始し、システムを初期化するためにそこに飛びます。例えば、プロセッサを正しい状態にするためのアッセンブリ コードは通常ブート ローダーで使われるのと同じコードです。実際、ブート ローダーと OAL の間でコードを共有することは BSP でコードの重複を最低限にするための一般的な方法です。しかしすべてのコードが 2 回実行されるわけではありません。例えば、ブート ローダーから開始されるハードウェア プラットフォームで、ブート ローダーは初期化の基礎作業を行なっているのに対して、StartUp は直接 KernelStart 関数に飛びます。

KernelStart 関数は前述のセクションで論じたようにメモリ マッピング テーブルを初期化し、マイクロソフト カーネル コードを実行するためにカーネル ライブラリを読み込みます。マイクロソフト カーネル コードは OEMInitGlobals 関数と呼ばれるようになり、静的 NKGLOBALS 構造へのポインタを OAL へと渡し、OEMGLOBALS 構造へのポインタを OAL からの戻り値の形で回収します。NKGLOBALS は KITL とマイクロソフト カーネル コードで使われるすべての関数と変数へのポインタを含んでいます。OEMGLOBALS は BSP のために OAL で実装されたすべての関数と変数へのポインタを有しています。ポインタをこれらのグローバル構造へ置換することによって、Oal.exe と Kernel.dll はお互いの関数とデータにアクセスできるようになり、構造属性のまたプラットフォーム固有のスタートアップ タスクを継続できるようになります。

アーキテクチャの一般的なタスクには、ページ テーブルとキャッシュ情報のセットアップ、トランジション ルックアサイド バッファ (TLB) の消去、アーキ

テクチャ固有バスとコンポーネントの初期化、インタロックされた API コードのセットアップ、デバッガ目的のためにカーネル通信をサポートするよう KITL の読み込み、カーネル デバッガ出力の初期化が含まれます。カーネルはその後、プラットフォーム固有の初期化を実行するための OEMGLOBALS 構造での、関数ポインタを介した OEMInit 関数の呼び出しに進みます。

表 5-5 は Kernel.dll が呼び出し、ユーザーが新しいハードウェア プラットフォームで Windows Embedded CE を実行するために BSP で変更しなければならないプラットフォーム固有の関数を列挙しています

図 5-5 カーネル スタートアップ サポート関数

関数	説明
OEMInitGlobals	Oal.exe と Kernel.dll の間でグローバル ポインタを交換します
OEMInit	プラットフォームのハードウェア インターフェイスを初期化します
OEMGetExtensionDRAM	入手可能であれば追加 RAM に関する情報を提供します。
OEMGetRealTime	RTC から時間を取得します。
OEMSetAlarmTime	RTC アラームを設定します。
OEMSetRealTime	RTC で時間を設定します。
OEMIdle	スレッドが実行されていないときに CPU をアイドル状態にします。
OEMInterruptDisable	特定のハードウェア割り込みを無効にします。
OEMInterruptEnable	特定のハードウェア割り込みを有効にします。
OEMInterruptDone	割り込みプロセス完了の信号を出します。
OEMInterruptHandler	割込みを処理します (SHx プロセッサとは異なります) 。
OEMInterruptHandler	FIQ を処理します (ARM プロセッサに特定) 。
OEMIoControl	OEM 情報のための IO 制御コード。
OEMNMI	マスク不可能な割り込みをサポートします (SHx プロセッサに固有) 。
OEMNMIHandler	マスク不可能な割り込みを処理します (SHx プロセッサに固有) 。

図 5-5 カーネル スタートアップ サポート関数

関数	説明
OEMPowerOff	CPU をサスペンド状態にし、最終の電源停止処理を管理します。

カーネル依存トランスポート レイヤ

OEMInit 関数は、ボード固有の周辺機器を初期化し、カーネルの変数を設定し、KITL IOCTL をカーネルに渡すことによって KITL を開始する主要 OAL ルーチンです。もしランタイム イメージで KITL を追加し有効にしてあるのならば、第 4 章「システムのデバッガおよびテスト」で説明されているように、カーネルは種々のトランスポート レイヤでのデバッガのために KITL を開始します。

図 5-6 は新しいプラットフォームで KITL サポートを有効にするために OAL が含むべき関数を列挙しています。

図 5-6 KITL サポート関数

関数	説明
OEMKitlInit	KITL を初期化します。
OEMKitlGetSecs	現在の時間を秒単位で返します。
TransportDecode	受信したフレームをデコードします。
TransportEnableInt	KITL 割込みが基礎にされた割込みであるなら有効化あるいは無効化します。
TransportEncode	トランスポートの要求されたフレーム構造によって、データをエンコードします。
TransportGetDevCfg	デバイスの KITL トランスポート設定を取得します。
TransportReceive	トランスポートからフレームを受信します。
TransportSend	トランスポートを使用してフレームを送信します。
KitlInit	KITL システムを初期化します。
KitlSendRawData	トランスポートを使ってプロトコルをバイパスしながら生データを送信します。
KitlSetTimerCallback	指定された総時間の後に呼び出されたコールバックを登録します。
KitlStopTimerCallback	上記のルーチンで使われたタイマを無効にします。

プロファイル タイマ サポート

オペレーティング システムのコアに位置し、OAL はシステムのパフォーマンスを測定するため、またパフォーマンスの最適化をサポートするためにメカニズムとして完璧な選択です。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、割り込み待機時間測定 (IL 測定) ツールを用いて、割り込みが生じた後に割り込みサービス ルーチン (ISR) が起動するまでに要する時間 (ISR 待機時間) や、ISR が生じたときから割り込みサービス スレッド (IST) が実際に開始したときまでの時間 (IST 待機時間) を計測することができます。しかしながら、このツールはすべてのハードウェア プラットフォームで入手可能ではない、システム ハードウェア ティック タイマや代替の高性能タイマを必要とします。もし新しいハードウェア プラットフォームが高性能ハードウェア タイマをサポートしているなら、図 5-7 で列挙されている関数を実装することによって IL Timing や類似したツールをサポートできます。

図 5-7 プロファイル タイマ サポート関数

関数	説明
OEMProfileTimerEnable	プロファイラ タイマを有効にします。
OEMProfileTimerDisable	プロファイラ タイマを無効にします。

ノート スレッドのスケジューリングと割り込み処理

OAL はまた割り込み処理とカーネル スケジューラをサポートしなければなりません。スケジューラはプロセッサのタイプに依存しますが、割り込み処理はさまざまなタイプのプロセッサにとって最適化されていなければなりません。

新しいデバイス ドライバの統合

コア システム関数とは別に BSP も周辺機器のためのデバイス ドライバを含んでいます。これらの周辺機器 デバイスはプロセッサ チップ上のコンポーネントや外部コンポーネントとなり得ます。プロセッサと分離されているときでさえ、それらはハードウェア プラットフォームの構成部分として残ります。

デバイス ドライバ コードの位置

表 5-8 は PQOAL モデルによるデバイス ドライバのためのソース コード位置を列挙しています。ユーザーの BSP が参照 BSP と同じプロセッサに基づいているなら、デバイス ドライバの適用には主に %TGTPLATROOT% フォルダのソース コードへの変更が求められています。新しいプラットフォームに参照プラットフォームには存在しない周辺機器が含まれているなら、BSP に新しいドライバ

を追加することも可能です。デバイス ドライバの開発に関するさらに多くの情報については、第 6 章「デバイス ドライバの開発」を参照してください。

表 5-8　デバイス ドライバのためのソース コード フォルダ

フォルダ	説明
%_WINCEROOT%\Platform\%_TGTPLAT%	プラットフォーム依存ドライバを含みます。
%_WINCEROOT%\Platform\Common\Src\Soc	プロセッサ ネイティブの周辺機器のためのドライバを含みます。
%_WINCEROOT%\Public\Common\Oak\Drivers	外部コントローラを含むネイティブではないの周辺機器のためのドライバを含みます。

設定ファイルの変更

既存の BSP から BSP を複製した場合、すべての設定ファイルはすでに揃っています。しかしながら、レッスン 2 で詳細に説明されているように Config.bib ファイルのメモリ レイアウトを再調査するのは大切なことです。第 2 章「ランタイム イメージのビルドおよび展開」で説明されたように、BSP に新しいドライバや変更されたコンポーネントを追加したときだけ、他の設定ファイルに変更が求められます。

レッスンの要約

適切な参照 BSP を複製することによって BSP 開発プロセスを開始することには、多くの利点があります。多くのテストされ証明された機能を引き出すことができるので、理想的には BSP は同じまたは類似したハードウェア プラットフォームに基づくものがよいでしょう。Windows Embedded CE は複製プロセスを容易にする PQOAL アーキテクチャとプラットフォーム ビルダ ツールを採用しています。目標は最小限のカスタマイズでブート可能なシステムを作成し、必要に応じて付加的な機能を追加し、周辺機器デバイスをサポートすることです。

最初に適用しなければならない最初の BSP コンポーネントはブート ローダーです。ブート ローダーはハードウェア プラットフォームの初期化と実行のカーネルへの受け渡しを担当します。2 つめのコンポーネントは OAL です。OAL はハードウェアの初期化、割り込み処理、スレッド スケジューラのためのタイマ処理、KITL、カーネル デバッガ出力のためにカーネルが必要とするプラットフォーム固有のコードを含んでいます。適用しなければならない 3 番目の BSP

部分は周辺機器 デバイスのためのデバイス ドライバです。適応が求められる 4 番目の BSP 部分はビルド プロセッサを制御し、メモリ レイアウトを決定し、システム設定内容を指定する、設定ファイルです。BSP 適用が同じプロセッサ アーキテクチャのための参照 BSP に基づいて行われるのであれば、ほとんどの CPU 関連やメモリ制御関連の BSP コードは変更せずにおくことができます。BSP のための必要なセットアップを作成する必要はなく、ハードウェアを立ち上げるのに重点を置いたプラットフォーム固有コード部分をアドレスするだけでよいのです。

レッスン 2：BSP のメモリ マッピングの構成

Windows Embedded CE でのメモリ管理は以前のバージョンと比べて明確に変化しています。過去のバージョンでは、すべてのプロセスは同じ 4 GB アドレス領域を共有していました。CE 6.0 では、各プロセスは固有のアドレス領域を有しています。この仮想メモリ管理の新システムは、以前のバージョンの限界が 32 プロセスであった実行能力を CE 6.0 で 32,000 プロセスまで引き上げました。このレッスンでは新しいメモリ アーキテクチャと管理の詳細を網羅し、プラットフォームで仮想メモリ領域を正確な物理メモリアドレスへとマップできるようにします。

このレッスンを終了すると、以下をマスターできます：

- Windows Embedded CE が仮想メモリを管理する方法を記述する。
- ハードウェア プラットフォームのための静的メモリ マッピングを設定する。
- システムで連続していない物理メモリを仮想メモリにマップする。
- OAL とデバイス ドライバの間でリソースを共有する。

レッスン時間（推定）：15 分

システム メモリ マッピング

Windows Embedded CE は MMU を使用して物理メモリにマップされた、32 ビット仮想アドレス領域を持ったページ メモリ管理システムを使用しています。32 ビットを用いて、システムは合計 4 GB の仮想メモリにアドレスでき、CE 6.0 はそれを以下の 2 つの領域に分けています (図 5-5 を参照)。

- カーネル領域　仮想メモリの上部の 2 GB に位置しており、ターゲット デバイス上で実行されているすべてのアプリケーション プロセスの間で共有されています。
- ユーザー領域　仮想メモリの下部の 2 GB に位置しており、各個別のプロセスに排他的に使用されます。各プロセスはその独自のアドレス領域があります。カーネルはプロセス切り替えが生じたときにこのプロセス アドレス領域のマッピングを管理します。プロセスは直接カーネル アドレス領域にアクセスすることはできません。

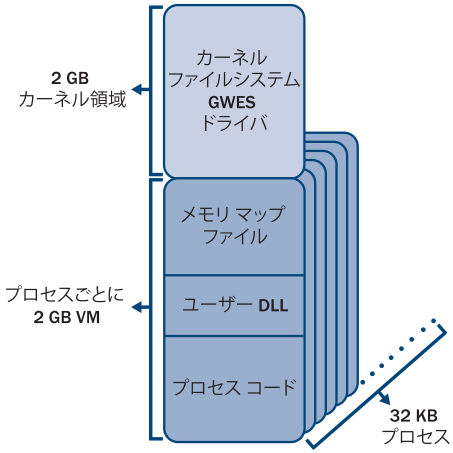


図 5-5　Windows Embedded CE 6.0 での仮想メモリ領域

カーネル アドレス領域

図 5-6 に示されているように、Windows Embedded CE 6.0 はカーネル アドレス領域をさらに特定の目的によっていくつかの領域に分けています。512 MB の下部の 2 つの領域は物理メモリをそれぞれキャッシュ済みまた未キャッシュの仮想メモリへ静的にマップします。カーネル エクセキュート イン プレイス (XIP) DLL とオブジェクト ストアである中央の 2 つの領域は OS デザインにとって重要です。残りの領域は、カーネル モジュールと CPU 固有の目的のためのものです。

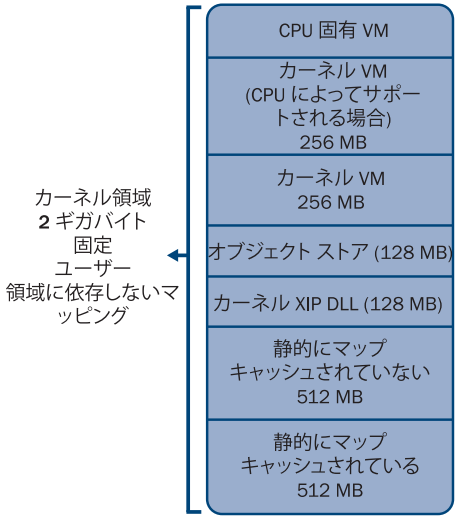


図 5-6　Windows Embedded CE 6.0 でのカーネル領域

表 5-9 は開始と終了のアドレスでのカーネル仮想メモリ領域を要約しています。

表 5-9　カーネル メモリ領域

開始アドレス	終了アドレス	説明
0xF0000000	0xFFFFFFFF	CPU 固有のシステム トラップとカーネル データ ページに使われます。
0xE0000000	0xEFFFFFFF	カーネル仮想マシン、CPU 依存である、例えばこのスペースは SHx にとって利用可能ではありません。
0xD0000000	0xDFFFFFFF	OS のすべてのカーネル モード モジュールに使用されます。
0xC8000000	0xCFFFFFFF	RAM ファイル システム、データベースやレジストリに使われるオブジェクト ストア
0xC0000000	0xC7FFFFFF	XIP DLL。
0xA0000000	0xBFFFFFFF	物理メモリの未キャッシュ マッピング。
0x80000000	0x9FFFFFFF	物理メモリのキャッシュ済みマッピング。

プロセス アドレス領域

プロセス アドレス領域は 0x00000000 から 0x7FFFFFFF までの範囲があり、CPU 依存カーネル データ セクション、4 つのメイン プロセス領域、ユーザーとカーネル領域の間の 1 MB のバッファに分けられます。図 5-7 はメイン領域を示しています。1 GB の最初のプロセス領域はアプリケーション コードとデータのためのものです。次の 512 MB のプロセス領域は DLL と読み取り専用 データのためのものです。次の 256 MB と 255 MB の 2 つの領域はメモリ マップ オブジェクトと共有システム ヒープのためのものです。共有システム ヒープはアプリケーション プロセスにとっては読み取り専用ですが、カーネルにとっては読み取りおよび書き込み可能です。

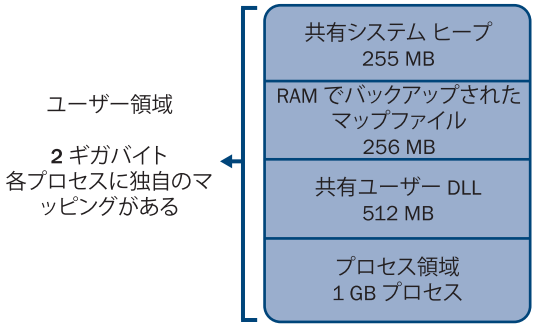


図 5-7　Windows Embedded CE 6.0 でのプロセス領域

表 5-10 は開始と終了のアドレスでのユーザー領域の仮想メモリ領域を要約しています。

表 5-10　プロセス メモリ領域

開始アドレス	終了アドレス	説明
0x7FF00000	0x7FFFFFFF	ユーザーとカーネル領域の間での保護のための未マップのバッファ。
0x70000000	0x7FEFFFFF	カーネルとプロセスの間での共有ヒープ。
0x60000000	0x6FFFFFFF	実際の物理ファイルに対応しないメモリマップ オブジェクト、主にプロセス間通信のために RAM バック マップ ファイルで使用されるアプリケーションとの下位互換性のため。
0x40000000	0x5FFFFFFF	プロセスと読み取り専用データの中に読み込まれた DLL。
0x00010000	0x3FFFFFFF	アプリケーション コードとデータ。
0x00000000	0x00010000	CPU 依存ユーザー カーネル データ (ユーザー プロセスにとっては読み取り専用)。

メモリ管理ユニット（MMU）

Windows Embedded CE 6.0 はプロセッサに仮想メモリと最大 512 MB のマップ済み物理メモリで物理メモリを関連付けるメモリ マッピング機構を提供するよう要求します。図 5-8 はカーネルのキャッシュ済みおよび未キャッシュの静的マッピング領域へマップされた 32 MB のフラッシュ メモリと 64 MB の RAM を使用した例を示しています。ARM ベースと x86 ベースのプラットフォームではメモリ マッピングはユーザー定義の OEMAddressTabl e を使用するのに対し

て、SHx ベースと MIPS ベースのプラットフォームではマッピングは CPU によって直接定義されます。メモリ管理ユニット (MMU) は物理仮想 アドレス マッピングの管理を担当します。

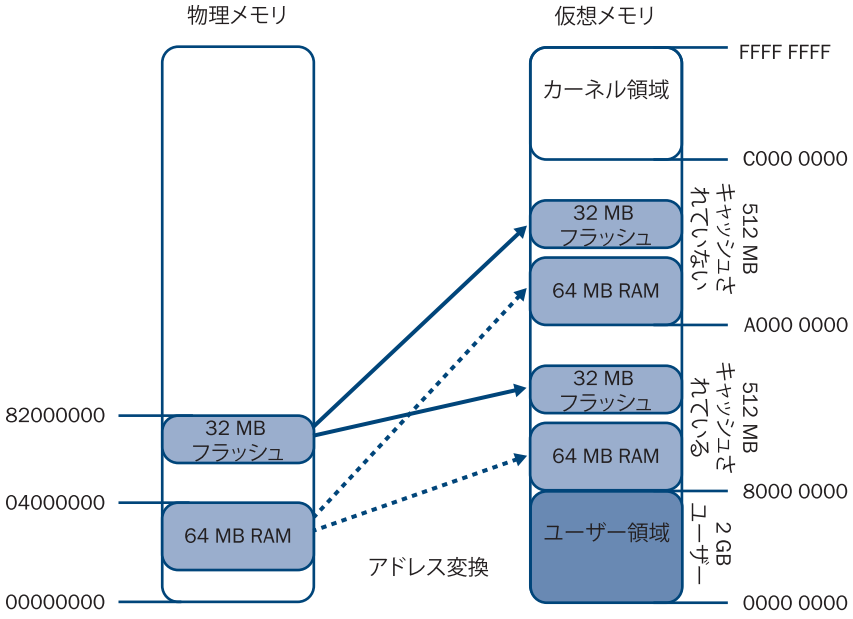


図 5-8　物理仮想メモリ マッピングの例

ノート　MMU の初期化

カーネルはシステム セットアップの間に MMU を初期化し、必要なページ テーブルを作成します。このカーネルのプロセッサ固有部分はハードウェア プラットフォームのアーキテクチャに依存します。実装の詳細については %_PRIVATEROOT%\Winceos\Coreos\Kernel の下位にあるプロセッサ タイプごとのサブディレクトリにある Windows Embedded CE のプライベート コードを参照してください。

静的にマップされた仮想アドレス

図 5-8 で表された仮想メモリ領域はそれらがスタートアップ時に定義されマッピングが変化しない事実を強調した静的にマップされた仮想アドレスです。静的にマップされた仮想アドレスは常にカーネル モードで入手可能でまた直接アクセス可能です。しかしながら、Windows Embedded CE が CreateStaticMapping と NKCreateStaticMapping API によってランタイムで静的マッピングをもサポートしていることは注目に値します。 これらの関数は指定された物理アドレスへマップされた未キャッシュ仮想アドレスを返します。

動的にマップされた仮想アドレス

カーネルはまた動的にも物理仮想 アドレスのマッピングを管理しており、これはすべての非静的マッピングに使用される手法です。LoadKernelLibrary を通してカーネルへ読み込まれるドライバや DLL は、VirtualAlloc を呼び出すことによってカーネル アドレス領域へ仮想メモリの領域を取り置き、VirtualCopy への呼び出しを通して新しいページ テーブル エントリを作成することにより物理アドレスへ仮想アドレスをマップすることができます。これは入出力処理を実行するために仮想アドレスをレジスタや周辺機器デバイスのフレーム バッファへマップするための一般的な手段です。もしマップされたバッファが必要なくなると、デバイス ドライバや DLL はページ テーブル エントリを削除し割り当てられた仮想メモリを解放するために VirtualFree を呼び出すことができます。

メモリ マッピングと BSP

BSP で静的メモリ マッピングについての情報を含めるために 2 つの要素をカスタマイズしなければなりません。

- Config.bib ファイル　システムがプラットフォーム上の異なったメモリ領域を使用する方法についての情報を提供します。例えば、どれだけのメモリが OS にとって利用可能か、どれだけがフリー RAM や特定の必要のためのリザーブ メモリとして使用できるかを明記できます。
- OEMAddressTable　レッスン 1 で説明されているように、基になるプラットフォームのメモリ レイアウトについての情報を提供します。Config.bib で指定されたメモリは OEMAddressTable でもマップされなければなりません。

不連続物理メモリのマッピング

第 2 章「ランタイムイメージのビルドおよび展開」で説明されたように、Config.bib ファイルの MEMORY セクションでオペレーティング システムのために RAMIMAGE メモリ領域での一つの連続領域を定義しなければなりません。システムはカーネル イメージやユーザーが MODULES や FILES セクションで指定したすべてのモジュールを読み込むためにこの定義を使用します。ユーザーは複数の RAMIMAGE 領域を定義できませんが、OAL は RAMIMAGE 領域を拡張し、ランタイムに追加的な非連続メモリ セクションを提供することができます。

図 5-11 は RAM 領域を拡張するのに重要な変数と関数を要約しています。

図 5-11 RAM 領域を拡張するための変数と関数

変数と関数	説明
MainMemoryEndAddress	この変数は RAM 領域の終わりを表示します。カーネルは Config.bib ファイルでオペレーティング システムのために取り分けられたサイズによって最初にこの変数を設定します。OAL OEMInit 関数は付加的な連続メモリが使用可能になった時にこの変数を更新します。
OEMGetExtensionDRAM	OAL はカーネルに非連続メモリの追加的領域の存在をレポートするためにこの関数を使用できます。OEMGetExtensionDRAM は第二のメモリ領域の開始アドレスと長さを返します。
pNKEnumExtensionDRAM	OAL はカーネルに一つ以上のメモリの追加的領域の存在をレポートするためにこの関数ポインタを使用できます。このメカニズムは 15 箇所までの異なった非連続メモリ領域をサポートすることができます。pNKEnumExtensionDRAM 関数ポインタを実装するならば、スタートアップ プロセスの間 OEMGetExtensionDRAM は呼び出されません。

ドライバと OAL の間で共有されるリソースの有効化

デバイス ドライバはしばしばメモリ マップ レジスタや DMA バッファといった物理リソースにアクセスする必要が生じるが、システムは仮想 アドレスとのみ作動するため、ドライバは物理メモリに直接アクセスできません。デバイス ドライバが物理メモリへアクセスできるようになるために、物理アドレスは仮想アドレスへマップされなければならない。

動的な物理メモリへのアクセス

もしバッファが DMA 処理を要求されたときのように、ドライバが物理的に連続したメモリを要求するなら、ドライバは AllocPhysMem 関数を使用することによって連続する物理メモリを割り当てることができます。もし割り当てが成功するなら、AllocPhysMem は指定された物理アドレスに対応する仮想アドレスへのポインタを返します。システムはメモリを割り当てるため、後に必要がなくなったときに FreePhysMem を呼び出すことによってそのメモリを開放するのは重要なことです。

反対に、もしドライバが Config.bib で定義された物理メモリ領域への非ページアクセスを要求するなら、MmMapIoSpace 関数を使用することができます。

MmMapIoSpace は指定された物理アドレスへ直接マップする非ページ仮想アドレスを返します。この関数は典型的にデバイス レジスタにアクセスするのに使用されます。

物理メモリの静的な取り置き

場合によって、ドライバ間であるいはドライバと OAL 間で (IST と ISR の間でのように) 物理メモリの共通領域を共有することが必要になるかもしれません。ブート ローダーとカーネルの間のブート引数のためにメモリ領域を共有するのと同じように、Config.bib ファイルでドライバ通信目的のために共有メモリ領域を取り置くことができます。一つの標準的な方法は、レッスン 1 で記載されたように Drv_glob.h で定義された DRIVER_GLOBALS 構造を使用することです。

ドライバと OAL 間の通信

カーネルによって要求される IOCTL の標準的セットに加えて、ドライバは OEMIoControl で実装されるカスタム IOCTL を通して OAL と通信できます。カーネル モード ドライバはカスタム IOCTL を介して、KernelIoControl を使用して間接的に OEMIoControl を呼び出します。カーネルは OEMIoControl へパラメータを通過させるだけで、プロセスは行いません。しかしながら、ユーザー モード ドライバは規定値では直接カスタム OAL IOCTL を呼び出せません。ユーザー モード ドライバやプロセッサからの KernelIOControl の呼び出しがカーネル モード コンポーネント (Oalioctl.dll) を通って OEMIoControl へ渡されます。これがユーザー アクセス可能な OAL IOCTL コードのリストを保ちます。もし要求された IOCTL コードがこのモジュールのリストになければ呼び出しは拒否されます。しかし %_WINCEROOT%\Public\Common\Oak\Oalioctl フォルダにある Oalioctl.cpp ファイルを変更することによってこのリストをカスタマイズできます。

レッスンの要約

Windows Embedded CE 6.0 メモリ アーキテクチャの十分な理解はすべての CE 開発者にとって必須です。特に BSP 開発者にとって、CE 6.0 が利用可能な物理メモリを仮想メモリ アドレス領域にマップする方法を知るのは重要です。OAL 、カーネル モード モジュール、またユーザー モード ドライバとアプリケーションからメモリにアクセスするには、カーネル モードやユーザー モードで利用可能な静的、動的マッピング手法の詳細な理解が要求されます。カーネル モードとユーザー モードの間の通信に関するさらに多くの情報については、第 6 章「デバイス ドライバの開発」を参照してください。

レッスン 3：OAL への電源管理サポートの追加

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、Windows Embedded CE 6.0 は OEM 開発者がハードウェア プラットフォームに適切になるようにシステム電源状態定義を実装するようカスタマイズできる電源管理コンポーネントに基づく電源管理機能の総合セットを提供します。OAL との関係で、電源管理機能の実装は 2 要素のタスクになります。ハードウェア コンポーネントの電源状態を制御するためにオペレーティング システムを有効にし、電源状態の変化についてオペレーティング システムに通知するためハードウェア プラットフォームも有効にする必要があるのです。ほとんどの埋め込みデバイスは電力消費を減らし、バッテリ寿命を延ばすため、少なくとも基本的な電源管理を必要とします。

このレッスンを終了すると、以下をマスターできます：

- プロセッサの電気消費を減らす方法を記述します。
- システムをサスペンドにしたり再開したりする切り替えパスを識別します。

レッスン時間（推定）：15 分

電源状態の切り替え

携帯情報端末 (PDA) のような恒常的に使用されていない埋め込みデバイスは延長時間をアイドル状態で操作し、通常の電力モードから省電力モードやサスペンド状態に切り替えることによって電力消費を抑える機会を提供します。現在市販されているほとんどの埋め込みプロセッサは、これら図 5-9 で図解されているような切り替えをサポートしています。

Windows Embedded CE は以下の方法で電源関連のイベントに対応することができます。

- バッテリの著しい低下　システムはボード上の電圧比較演算子 (NMI) が起動したマスク不可能割り込みに対する対応としてクリティカル オフ状態に切り替えます。
- アイドル　システムは CPU に実行するワーカー スレッドがないとき CPU を省電力モードに切り替え、割り込みが生じたときに再開させます。
- サスペンド　システムはユーザーがオフ ボタンを押したときや無通信のタイムアウトへの対応としてデバイスをサスペンド状態に切り替え、ユーザーが電源ボタンを再び押すなどの再開イベントの対応として再開します。いくつかの埋め込みデバイスでサスペンド状態は真の電源オフ状態に相当しており、この場合システムはコールド ブートで再開します。

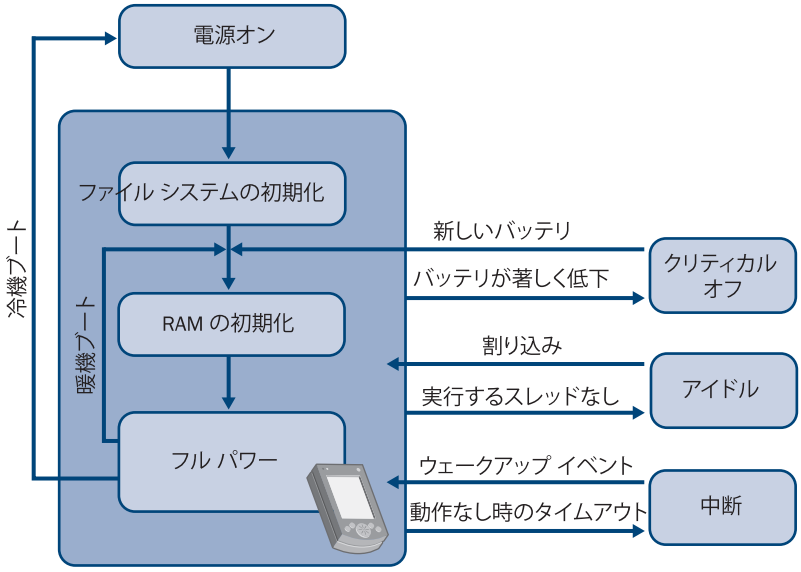


図 5-9　電源状態の切り替え

アイドル モードでの電力消費の削減

デバイスを省電力モードに切り替えるために、Windows Embedded CE は OEMIdle 関数を使用しています。これはスケジューラに実行するスレッドがないときにカーネルを呼び出すものです。OEMIdle 関数は、プラットフォームの機能に依存するハードウェア固有のルーチンです。例えば、システム タイマが一定の間隔で作動するならば、タイマ割り込みが起こるたびにシステムが再開するために OEMIdle 関数は期待される節電機能を実際に供給できないことになります。反対に、プロセッサがプログラム可能な間隔タイマをサポートするならば、省電力モードで経過した時間の合計を指定するためにカーネルの dwReschedTime 変数を使用できます。

省電力モードから再開する際、システムはスケジューラで使用されたカーネル グローバル変数を変更しなければなりません。これは特に CurMSec 変数にとって重要です。この変数はシステムが最後のシステム ブートからのミリ秒数の追跡記録を保つのに使用します。再開ソースはシステム タイマにもその他の割り込みにもなり得ます。再開ソースがシステム タイマなら、実行が OEMIdle 関数に戻される前に CurMSec 変数は変更されています。他の場合には CurMSec に変更値が含まれていません。OEMIdle 実装の詳細に関してさらに学習するためには、%_WINCEROOT%\Platform\Common\Src\Common\Timer\Idle フォルダにある Idle.c ソース コード ファイルを参照してください。

ノート　カーネル グローバル変数

カーネルがスケジュールのためにエクスポートするグローバル変数の詳細については、http://msdn.microsoft.com/en-us/library/aa915099.aspx の Microsoft MSDN Web サイトにある、Windows Embedded CE 6.0 ドキュメントの「Kernel Global Variables for Scheduling」セクションを参照してください。

システムの電源オフとサスペンド

Windows Embedded CE デバイスがサポートできる最大の省電力状態は電源オフまたはサスペンド状態です。システムは直接 GwesPowerOffSystem directly or SetSystemPowerState を呼び出すか SetSystemPowerState を呼び出すことによって、デバイスがサスペンド状態に入ることを要求できます。どちらの機能も最終的に OEMPowerOff ルーチンを呼び出します。

OEMPowerOff ルーチンは OAL の一部で、CPU をサスペンド状態に切り替えることを担当します。もしプロセッサがサスペンド状態に入ったとき自動的に RAM を自動更新モードにしないならば、OEMPowerOff がそれを行います。デバイスを再開するために割り込みを設定することもできます。ハンドヘルド デバイスでこれは典型的に電源ボタンの割り込みですが、ターゲット プラットフォームに適切な他の再開イベント ソースを使用することもできます。

サスペンド状態へ入る

サスペンド状態に入るとき、Windows Embedded CE は以下のステップの手順を実行します。

1. GWES が電源オフ イベントについてのタスク バーを通知します。
2. システムがもし調整画面ならば調整を中止します。
3. システムがウィンドウズ メッセージ キューを停止します。ステップ 3 の後、システムはシングル スレッド モードに入ります。これはブロッキング操作を実行する関数の呼び出しを防ぐためです。
4. システムが再開の際スタートアップ ユーザー インターフェイス (UI) が現れるかどうかチェックします。
5. システムがビデオ メモリを RAM へ保存します。
6. システムが SetSystemPowerState (NULL、POWER_STATE_SUSPEND、POWER_FORCE) を呼び出します。
7. 電源管理
   a. ファイル システムに関連するドライバを電源オフにするために FileSystemPowerFunction を呼び出します。

b. カーネルに最終電源オフを行うことを通知するために PowerOffSystem を呼び出します。
c. スケジューラを起動するために Sleep(0) を呼び出します。

ノート FileSystemPowerFunction および PowerOffSystem
OS デザインが電源管理や GWES を含んでいない場合、OEM は明示的に FileSystemPowerFunction と PowerOffSystem を呼び出さなければなりません。

8. カーネル :
   a. GWES プロセスの読み込みを解除します。
   b. Filesys.exe の読み込みを解除します。
   c. OEMPowerOff を呼び出します。
9. OEMPowerOff が割り込みを設定し、CPU をサスペンド状態にします。

サスペンド状態からの再開

事前に設定された割り込みがシステムを再開したとき、関連付けられた ISR が実行し OEMPowerOff ルーチンへ返します。この関数から戻るときに、システムは再開手順を実行します。これには以下のステップが含まれます。

1. OEMPowerOff が元の状態へ割り込みを再設定し、返します。
2. カーネル :
   a. システム タイマを再初期化するために InitClock を呼び出します。
   b. 電源オン通知とともに Filesys.exe を開始します。
   c. 電源オン通知とともに GWES を開始します。
   d. KITL 割込みが使用中であったなら、それを再初期化します。
3. 電源オン通知とともに電源管理が FileSystemPowerFunction を呼び出します。
4. GWES
   a. RAM からビデオ メモリを復元します。
   b. ウィンドウズ マネージャの電源をオンにします。
   c. 画面のコントラストを設定します。
   d. 要求されているならばスタートアップ UI を表示します。
   e. 再開のタスクバーを通知します。
   f. ユーザー サブシステムを通知します。
   g. 要求されているのであればアプリケーションを起動させます。

ノート　ウェイクアップ ソースの登録

OAL がカーネル IOCTL_HAL_ENABLE_WAKE をサポートしているなら、アプリケーションはウェイクアップ ソースを登録できます。詳細については、http://msdn2.microsoft.com/en-us/library/aa914884.aspx の Microsoft MSDN Web サイトにある、Windows Embedded CE 6.0 ドキュメントの「IOCTL_HAL_ENABLE_WAKE」セクションを参照してください。

クリティカル オフ状態のサポート

NMI を起動させる電圧比較演算子を機能としてもつハードウェア プラットフォームでは、バッテリ低下状態でデータを失わないよう保護するためクリティカル オフ状態のサポートを実装することができます。x86 ハードウェアではカーネルがシステムでの重大なイベントをキャプチャするために OEMNMIHandler 関数をエクスポートします。その他のシステムでは、電源管理と連携してデータ損失なくシステムの電源をオフにするために SetSystemPowerState を呼び出すカスタム IST を実装しなければなりません。クリティカル オフ状態は典型的に有効な動的 RAM 更新を伴うサスペンド状態に対応します。

ノート　バッテリ レベルがゼロになる

クリティカル オフ状態のサポートを実装しているとき、システムがまだ周辺機器の電源をオフにする、RAM を自動更新にする、またおそらくは再開状態を設定する、CPU をサスペンドにするといったすべての電源オフ タスクを実行する時間がある時点で NMI を起動させるようにしてください。

レッスンの要約

電源管理はターゲット デバイスでの効率的な電源消費を確実にする重要な Windows Embedded CE 機能です。OEM はバッテリ電源を使用するデバイスにとって通常の電力モードからアイドルまたはサスペンド モードまたはクリティカル オフ状態への切り替えを可能にするために OAL で電源管理機能を実装しなければなりません。電源管理サポートを実装することには、タイマ関連のカーネル変数を再同期すること、周辺機器の電源をオフにすること、RAM を自動更新モードにすること、再開状態を設定すること、CPU をサスペンドにすることが関わってきます。これらの低レベル ルーチンを実装することは簡単なことではありませんが、マイクロソフトは実装の詳細についてのより良い理解を得させるためにサンプル BSP で十分な参照コードを提供しています。

演習 5：ボード サポート パッケージ の適用

この演習では Platform Builder を用いて Visual Studio 2005 で参照 BSP を複製し、それをランタイム イメージをビルドするために使用します。Windows Embedded CE 開発コンピュータ上で実行できるため、この演習では基になるプラットフォームとしてデバイス エミュレータを使用します。マイクロソフトは参照 BSP として Platform Builder にデバイス エミュレータ BSP を含んでいます。

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するために、この本の付属物中のドキュメント「演習 5 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。

x BSP の複製

1. Visual Stadio 2005 で、[ツール] メニューから、[Platform Builder for CE 6.0] をポイントし、[BSP の複製] をクリックします。
2. [BSP の複製] ダイアログ ボックスで [ソース ボード サポート パッケージ] ボックスの一覧から [デバイス エミュレータ : ARMV41] をクリックします。
3. [新しい BSP 情報] ボックスの一覧で表 5-12 に示されている情報を入力します (図 5-10 も参照してください)。

表 5-12　新しい BSP の詳細

パラメータ	値
名前	DeviceEmulatorClone
説明	デバイス エミュレータ BSP の複製
プラットフォーム ディレクトリ	DeviceEmulatorClone
ベンダ	Contoso Ltd.
バージョン	0.0

4. [カタログ エディタで新しい BSP カタログ ファイルを開く] チェック ボックスをオンにして、[複製] をクリックします。

5. プラットフォーム ビルダがデバイス エミュレータ BSP の複製に成功したことを確認し、対応する [BSP の複製] ダイアログ ボックスで [OK] をクリックします。
6. Visual Studio が自動的に DeviceEmulatorClone.pbcxml カタログ ファイルを開くことを確認します。カタログ エディタを何の変更もせずに閉じます。

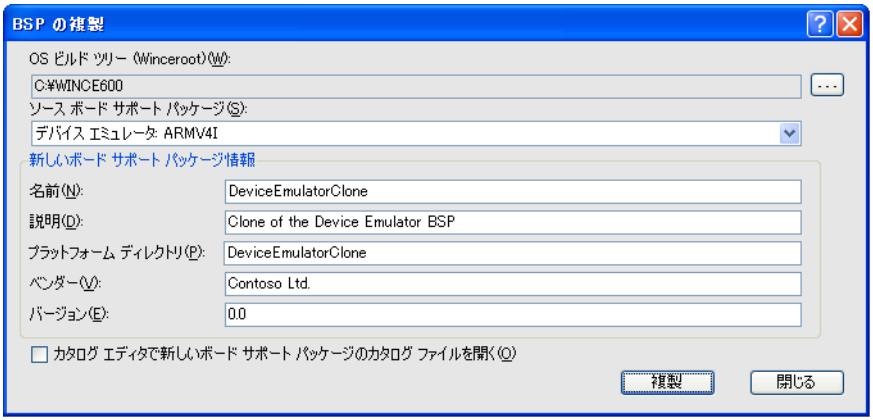


図 5-10　BSP 複製情報

ランタイム イメージの作成

1. 複製した BSP を検証するために、DeviceEmulatorClone BSP に基づく新しい OS デザインを作成します。図 5-11 に図解されているように、OS デザイン DeviceEmulatorCloneTest を呼び出します (このステップを完了する方法に関する詳細は第 1 章の演習 1 も参照してください) 。
2. [デザイン テンプレート] ボックスの一覧で [産業デバイス] を、[デザイン テンプレート変数] ボックスの一覧で [産業コントローラ] をクリックします。ウィザードの続くステップで既定のオプションを利用します。
3. Platform Builder が DeviceEmulatorCloneTest プロジェクトを生成した後、[カタログ項目ビュー] でカタログ項目を検査することによって OS デザインを確認します。
4. [ビルド］メニューから [構成マネージャ] をクリックし、[アクティブ ソリューション構成］ボックスの一覧に [DeviceEmulatorClone ARMV4I デバッガ] が表示されていることを確かめ、デバッガ ビルド構成が有効であることを確認します。
5. [ビルド］メニューから [ソリューションのビルド] をクリックします。

6. ビルドが完了したのち、デバイス エミュレータを使用するための接続オプションを設定します。
7. ランタイム　イメージをデバイス　エミュレータにダウンロードし、Windows Embedded CE を開始するために、[ターゲット] メニューから [デバイスの接続] をクリックします。Visual Studio 2005 の出力ウィンドウのデバッガ メッセージに注意します。デバイスが完全に開始するまで待ちます。

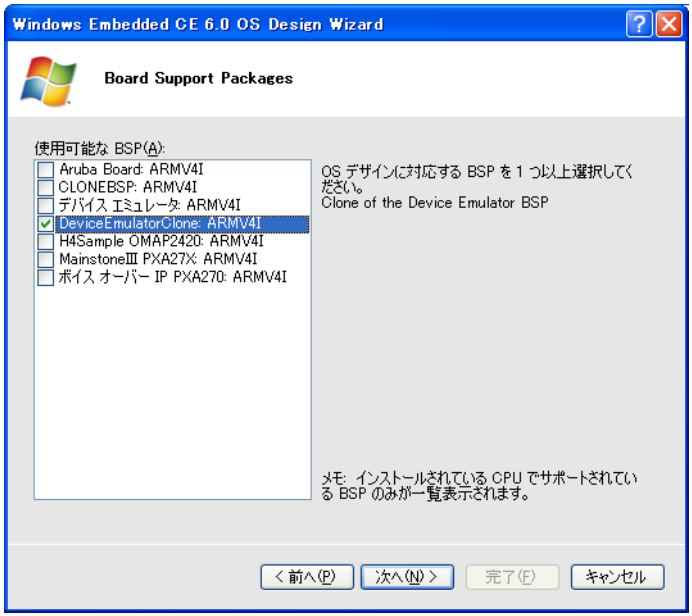


図 5-11　DeviceEmulatorClone BSP に基づく新しい OS デザイン

ノート　BSP 適用

デバイス エミュレータは参照 BSP と複製 BSP の両方のために同じハードウェア プラットフォームをエミュレートします。このため、新しいランタイム イメージはそれ以上の適用なしでデバイス エミュレータ上で実行します。しかしながら、実際には基にあるハードウェアは多くの場合に異なっており、CE を正常に開始するには BSP 適応が必要となります。

x BSP のカスタマイズ

1. ターゲット デバイスから接続を解除しデバイス エミュレータを閉じます。
1. 図 5-12 で図解されているように、Visual Studio で %_PLATFORMROOT%\DeviceEmulatorClone\Src\Oal\Oallib フォルダにある init.c ソース コード ファイルを開きます。
2. OAL 関数 OEMGetExtensionDRAM を検索し、システム セットアップの間 Visual Studio で [出力] ウィンドウでデバッガ メッセージがプリントされるように以下の行のコードを追加します。

```
BOOL
OEMGetExtensionDRAM(
   LPDWORD lpMemStart,
   LPDWORD lpMemLen
   )
{

...

   OALMSG(OAL_FUNC, (L"++OEMGetExtensionDRAM\r\n"));

   //変更がランタイム イメージの一部であることを確認するためのテスト メッセージ。
   OALMSG(1,(TEXT("This modification is part of the run-time image.\r\n")));

...

}
```

3. 変更を含ませるためにランタイム イメージをリビルドし、デバイス エミュレータで新しいランタイム イメージをダウンロードして開始するためにデバイスへ再び接続します。Windows Embedded CE が [出力] ウィンドウでデバッガ メッセージを出力することを確認してください。

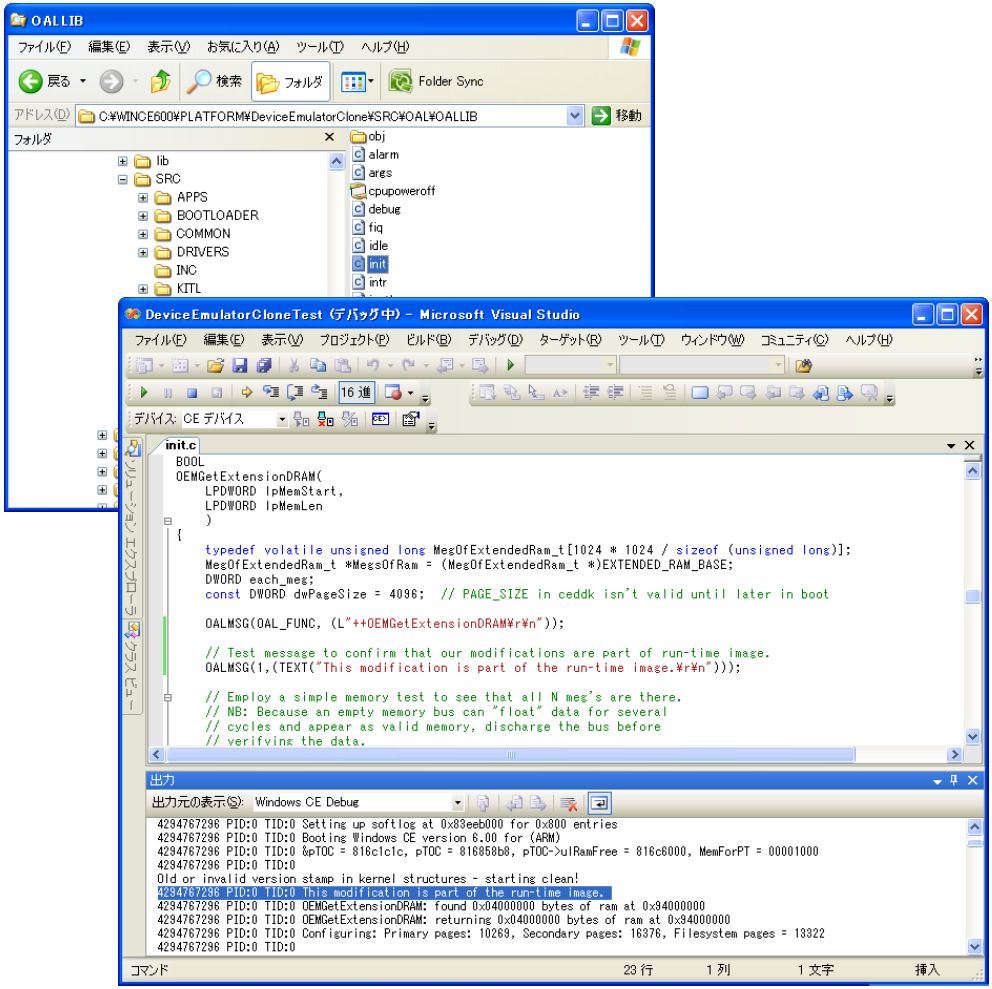


図 5-12　DeviceEmulatorClone BSP のカスタマイズ

本章のレビュー

ボード サポート パッケージの適用は OEM が Windows Embedded CE 6.0 を新しいハードウェア プラットフォームにポートするときに直面する最も複雑で重要な開発タスクの一つです。この仕事を容易にするために、マイクロソフトは Platform Builder とともに参照 BSP を提供し、OEM に最も適切な BSP を複製することによって開発プロセスを開始するように勧めています。PQOAL ベースの BSP はプロセッサ タイプと OAL 関数によってプラットフォーム不明とプラットフォーム固有コードを分けるためによく整理されたフォルダとファイル構造を採用しており、OEM はカーネルやオペレーティング システムの一般的な面にサイドトラックされずに、プラットフォーム固有の実装の詳細に集中することができます。

OEM 開発者は BSP の効果的な適用を確実にするために以下の推奨を考慮するべきです。

- Windows Embedded CE の参照 BSP の研究　Windows Embedded CE の BSP はカーネルへの密接な関連性とともによく定義されたアーキテクチャを採用しています。これによりカーネルがオペレーティング システムを実行するのに要求する数々の API を実装することが必要になります。これらの API とそれらの目的を知ることはとても重要です。PQOAL ベースのアーキテクチャが依然として関係してきます。
- BSP の複製　最初から完全に新しい BSP を書き込むことは避けてください。その代りに、適用プロセスを省略して開始するために BSP を複製してください。参照 BSP から可能な限り多くのコードを再利用することで、開発時間を短縮するだけでなく、ソリューションの品質を向上させ、将来のアップグレードの効果的な処理のための確実な土台を提供することになります。
- ブート ローダーと BLCOMMON　BLCOMMON と関係するライブラリはランタイム イメージをダウンロードし、ユーザーがスタートアップ プロセスの間ターゲット デバイスと対話操作することを有効にするための有用なハードウェア固有の機能を提供するため、ブート ローダーを実装しているときにはこれらのライブラリを使用してください。
- メモリと BSP　Windows Embedded CE 6.0 が物理また仮想メモリを扱う方法を徹底的に理解してください。必要であれば、入手可能なメモリに関する現在の情報をオペレーティング システムに提供し OEMAddressTable でのエントリを調整するために <ブート ローダー>.bib と Config.bib ファイルを設定してください。Windows Embedded CE では直接物理メモリにア

クセスできないことを念頭に置いておいてください。物理メモリ アドレスを仮想メモリ アドレスにマップするために正確なメモリ マッピング API を使用してください。

- 電源管理の実装　システムが CPU をアイドル モードに切り替えることができるようにするため OEMIdle 関数を実装してください。もしプラットフォームがユーザー動作や著しいバッテリ レベル低下に対応するサスペンド モードへの電源状態の切り替えをサポートしているなら、OEMPowerOff の実装も考慮してください。

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。

- PQOAL
- ブート ローダー
- KernelIoControl
- ドライバ グローバル

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。

周辺機器デバイスのハードウェア レジスタへのアクセス

周辺機器ハードウェアのためにデバイス ドライバを実装し、デバイスと対話操作するために MmMapIoSpace API を使用してハードウェア レジスタにアクセスしてください。アプリケーションから MmMapIoSpace を呼び出すことは可能でないことに注意してください。

ノート　エミュレータ制限
デバイス エミュレータはソフトウェアで ARM をエミュレートしているため、ハードウェア デバイスにはアクセスできません。この提案されている練習を実行するために正規のハードウェア プラットフォームを使用するべきです。

プラットフォームのメモリ マッピングの再整理

複製されたデバイス エミュレータ BSB の Config.bib ファイルを変更することにより、システム上の利用可能な RAM をさらに削減し、メモリ情報 API やPlatform Builder ツールを使用することによってシステム上の利用可能なメモリの点での影響を研究してください。

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。

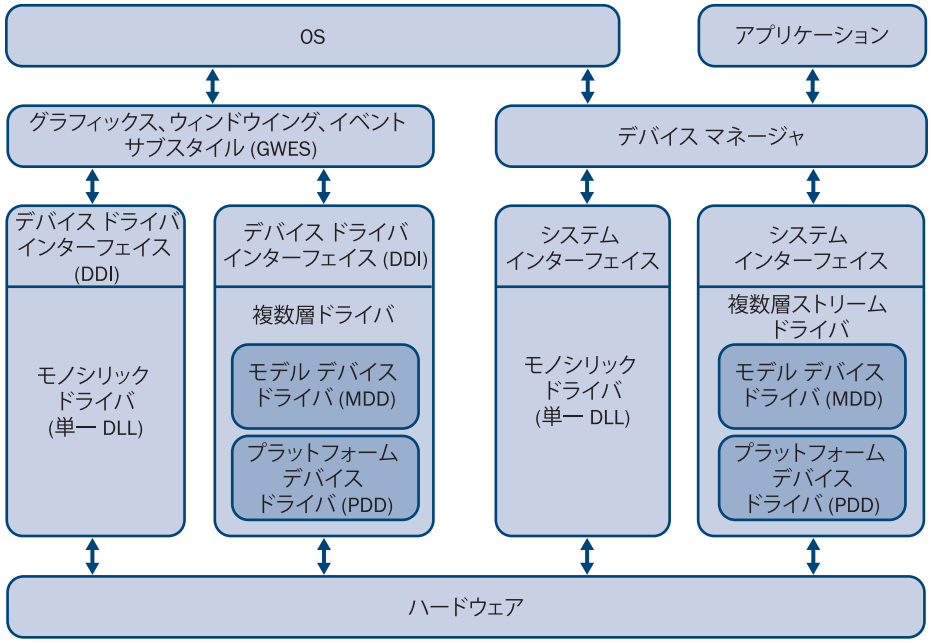


図 6-1　モノシリック ドライバおよび複数層ドライバ アーキテクチャ

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。

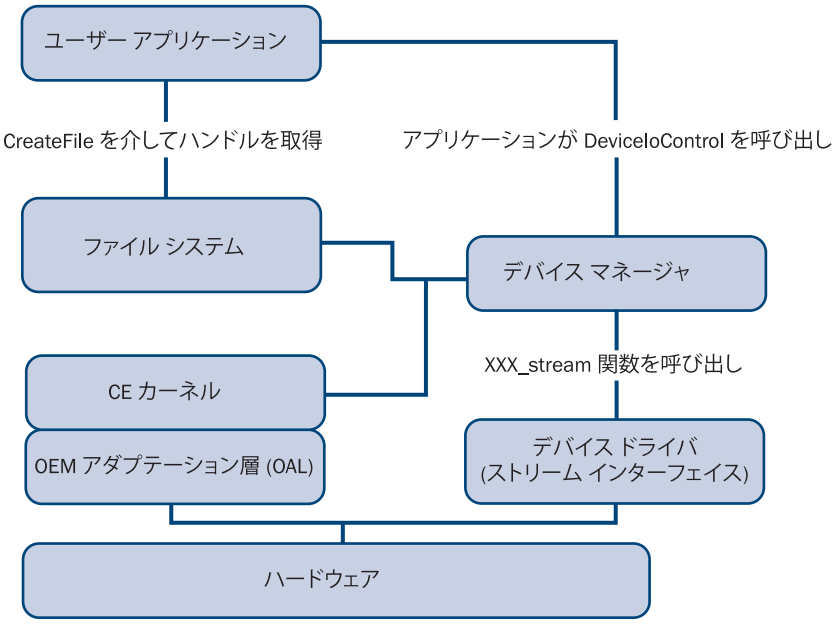


図 6-2　Windows Embedded CE 6.0 の [デバイス マネージャ]

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。

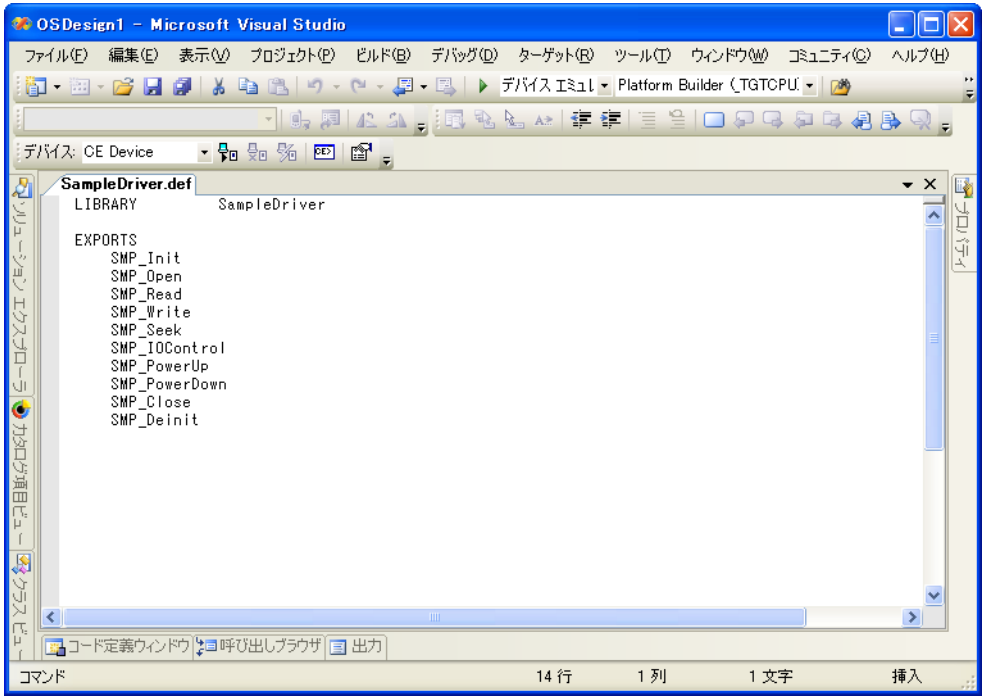


図 6-3 ストリーム ドライバのサンプル .def ファイル

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。

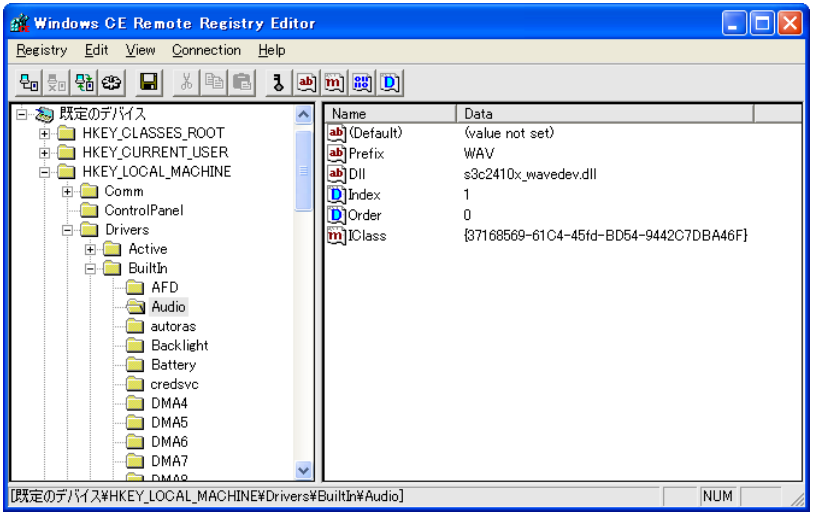


図 6-4　オーディオ デバイス ドライバ レジストリ

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、複拠のブート 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拠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誂みリクエスト (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。

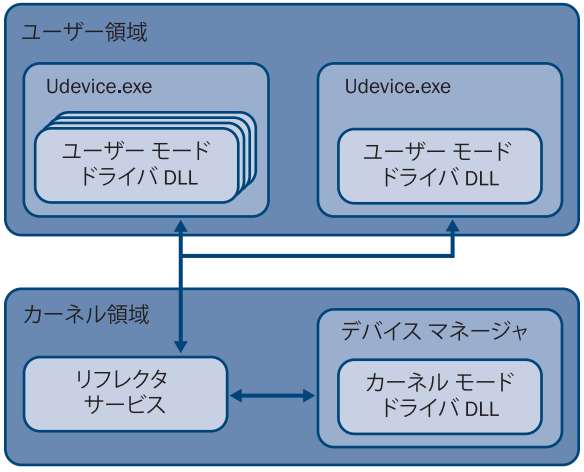


図 6-5　ユーザー モード ドライバ、カーネル モード ドライバ、およびリフレクタ サービス

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。

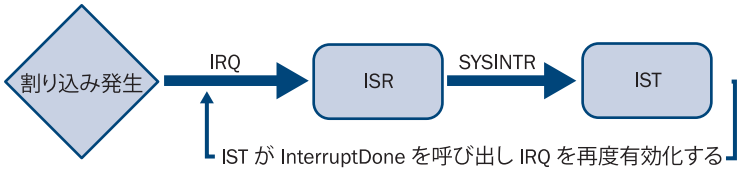


図 6-6　IRQ、ISRs、SYSINTR、および IST

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)。

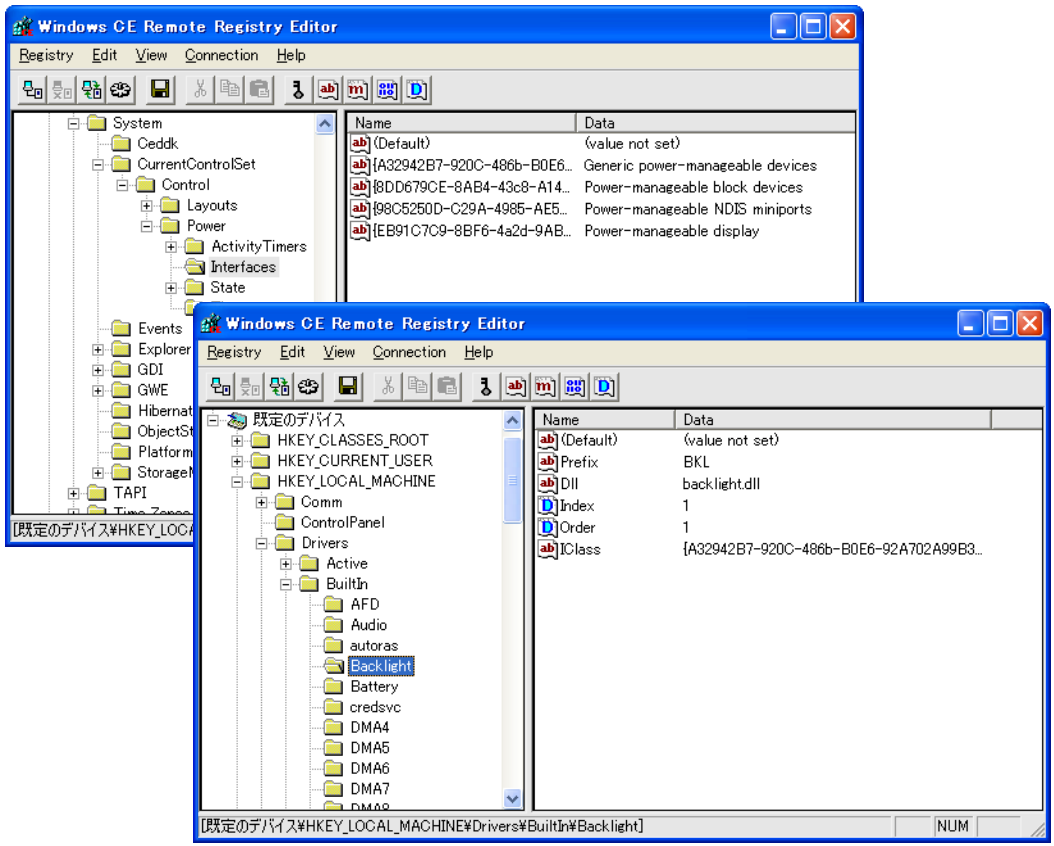


図 6-7　IClass レジストリ エントリを構成して、電源管理通知を受け取る。

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。

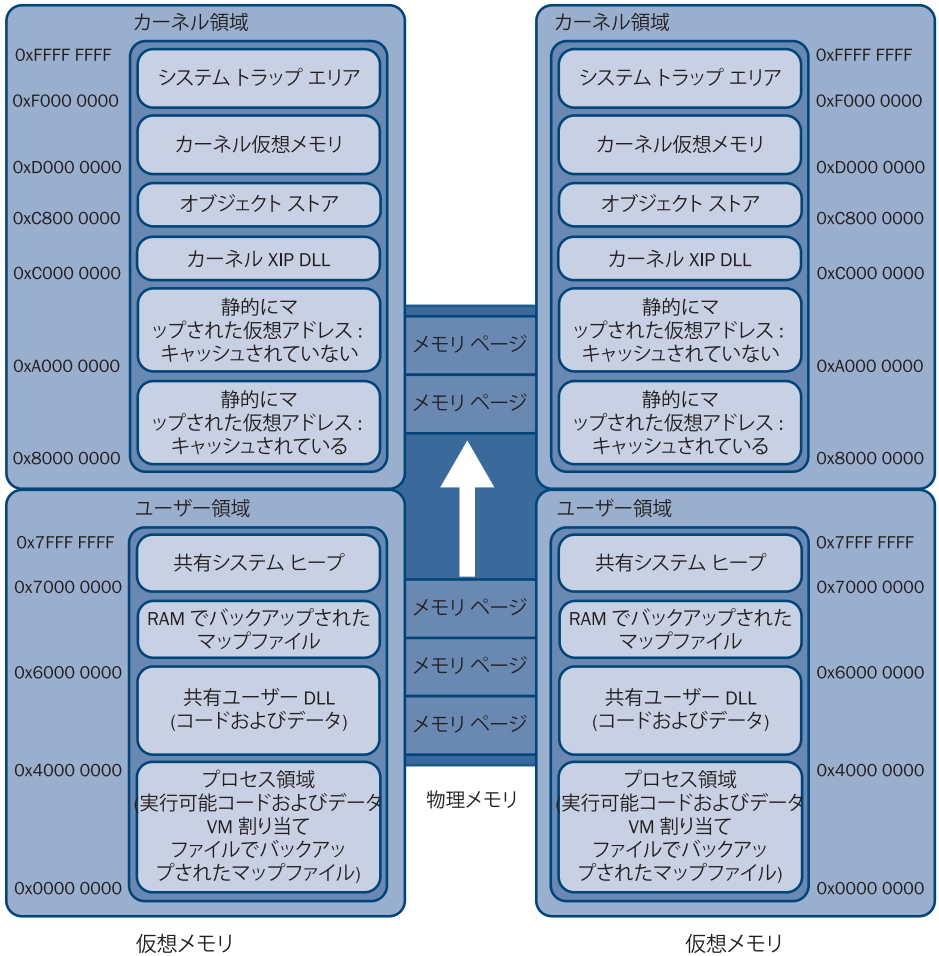


図 6-8　カーネル領域およびユーザー領域の仮想メモリ領域

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。

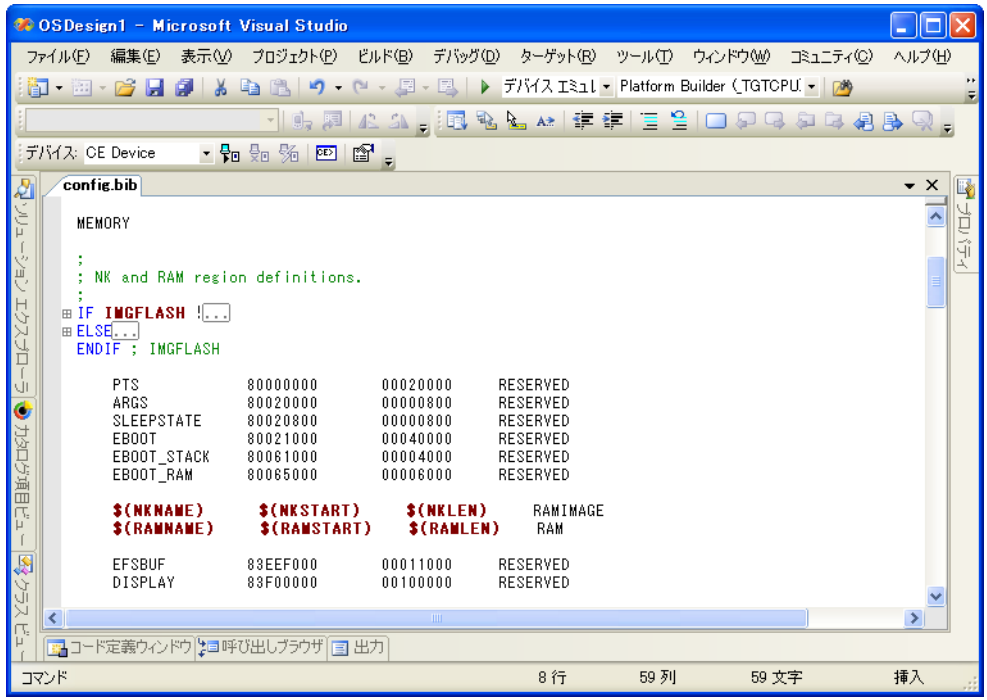


図 6-9　Config.bib ファイルでの予約メモリ領域の定義

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。

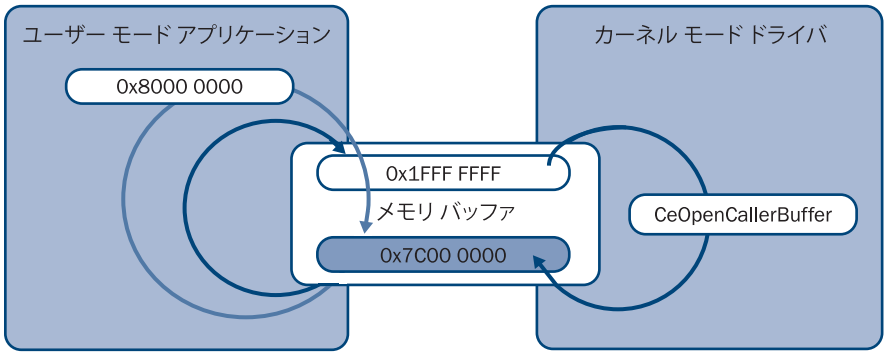


図 6-10　共有バッファでのマーシャルされたポインタの操作

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。

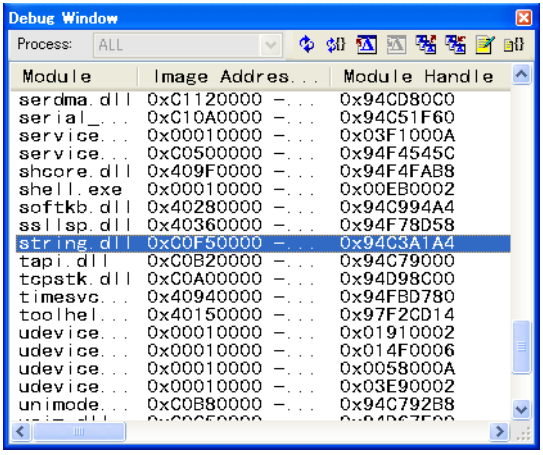


図 6-11　ロードされた文字列ドライバと [モジュール] ウィンドウ

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)。

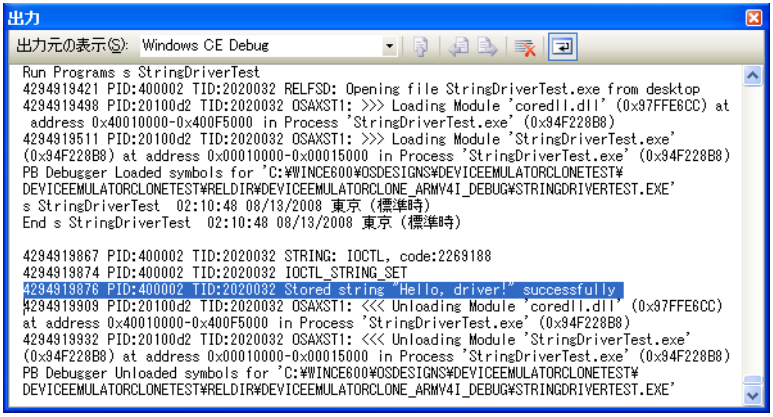


図 6-12　文字列ドライバからのデバッグ メッセージ

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)。

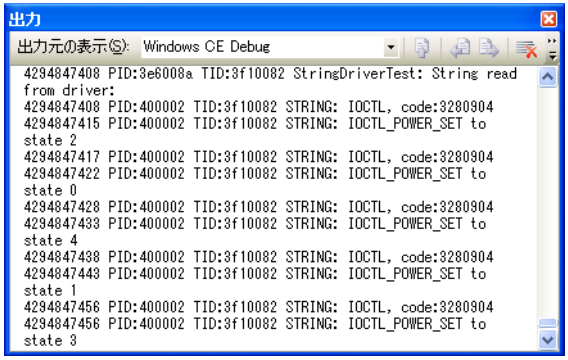


図 6-13 文字列ドライバからの電源管理関連のデバッグ メッセージ

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。

用語集

アプリケーション プログラミング インターフェイス (API)　API はオペレーティング システムやライブラリがアプリケーション プログラムからの要求をサポートするために提供する機能インターフェイス。

アプリケーション検証 (AppVerifier)
AppVerifier は、開発者が通常のアプリケーション テストでは識別しにくいヒープの破損や正しくないハンドル使用法など、微妙なプログラミング エラーを発見するのに役立つ。

同期アクセス　2 つかそれ以上のスレッドが、同時に同じバッファにアクセスしたとき。

バイナリ イメージ ビルダ (.bib)　bib ファイルはランタイム イメージに含まれるモジュールとファイルを定義。

ブート ローダー　プロセッサ起動時にプロセッサを初期化してその後オペレーティング システムを起動させるために実行されるコードの構成要素。

ボード サポート パッケージ (BSP)　BSP はすべてのボード ハードウェア特有コードの一般名。典型的にはブート ローダー、OEM アダプテーション層 (OAL)、またボード特有デバイス ドライバを含みます。

カタログ　ユーザーに OS デザインのために選択可能な機能を提出するコンポーネントのコンテナ。

デバッガ拡張子コマンド (CEDebugX)
CeDebugX は Platform Builder デバッガのための拡張子。休止時のシステムの状態について詳細な情報を提出し、クラッシュ、ハングやデッドロックを診断しようとします。

Windows Embedded CE 試験キット (CETK)
CETK は Windows Embedded CE オペレーティング システムのために開発したデバイス ドライバを試験することのできるツール。

複製　複製することによってファイルの精密なコピーを生成し、変更を実行する前にそれらの安全なコピーを保管。[PUBLIC] フォルダ中のコードは常に変更を行なう前に複製される必要があります。

コンポーネント　カタログを使用しながら OS デザインに加えたりそこから除いたりすることができる CE 機能。

コア接続 (CoreCon)　Windows CE はダウンロードやデバッガのための多機能接続を有効にする、コア接続と呼ばれる統合接続インフラストラクチャをサポート。

クリティカル セクション　ミューテックス オブジェクトと同様の同期プロセスを持つオブジェクト。違いはクリティカル セクションは単一プロセスのスレッドによってのみアクセスされることです。

データ マーシャリング　異なったプロセスでのデータのアクセス権と妥当性をチェックするデータ上で行なわれるプロセス。

デバッグ領域　特定の機能やドライバのモードに関連してデバッガ メッセージを有効または無効にするフラグ。

デバイス ドライバ　デバイス ドライバは物理または仮想デバイスの機能を抽象することによってデバイス操作を管理するソフトウェア。

Dirs ファイル　Dirs ファイルはビルドされるソース コードを含むサブディレクトリを特定するテキスト ファイル。

埋め込みポインタ メモリ構造に埋め込まれたポインタ。

環境変数 機能を有効または無効にすることのできる Windows 環境変数。これは一般的にカタログからビルド システムや OS デザインを設定するのに使用されます。

イベント システム中の他のスレッドを通知するためにスレッドやカーネルで使用される同期オブジェクト。

例外 例外はプログラムが実行している間に起こる異常な状況。

Iltiming Iltiming は Windows Embedded CE システムで割り込みサービス ルーチン (ISR) や割り込みサービス スレッド (IST) 待ち時間を決定する。

割り込み プロセスを要求する何かが発生したことを表示するために一時的にシステムを中断する (システムに割り込む) トリガ。システム上の各割り込みは特定の割り込み要求 (IRQ) 値と関連付けられ、この IRQ 値は 1 つかそれ以上の割り込みサービス ルーチン (ISR) と関連付けられています。

割り込みサービス ルーチン (ISR) ISR はハードウェアが割り込みに応答して起動するソフトウェアのルーチン。ISR は割り込みを検査して SYSINTR 値を返すことによってそれを処理する方法を決定します。この SYSINTR 値がその後割り込みサービス スレッド (IST) と関連付けられます。

割り込みサービス スレッド (IST) IST は割り込みプロセスのほとんどを行なうスレッド。OS が処理するための割り込みを受けると OS は IST を起動します。各 IST が SYSINTR 値と関連付けられた後、SYSINTR 値は割り込みサービス ルーチン (ISR) から返されることがあり、その後関連付けられた IST が実行します。

IRQ (割り込み要求) IRQ 値は、割り込みでハードウェアと関連付けられる。各 IRQ 値は 1 つかそれ以上の、それが起動させられたときに関連付けられた割り込みを処理するためにシステムが実行する割り込みサービス ルーチン (ISR) と関連付けることができます。

カーネルデバッガ カーネル デバッガはターゲット デバイスへの接続を設定したり、ターゲット デバイスにランタイム イメージをダウンロードしたりするために要求される機能を統合。それはオペレーティング システム、ドライバ、またアプリケーションのデバッガを許可します。

Kernel Independent Transport Layer (KITL) KITL は、デバッガ サービスをサポートするための簡便な方法を提供するために設計されました。

カーネル モード ドライバ カーネルのメモリ領域で実行するドライバ。

カーネル トラッカ このツールは OS の開発ワーク ステーション上や Windows Embedded CE ベースのデバイスで生じているアプリケーション イベントの視覚表現を提供。

複数層ドライバ 後日のメンテナンスや再利用を容易にするためにいくつかの層に分けられたドライバ。

モデル デバイス ドライバ (MDD) 複数層ドライバの MDD 層は OS とプラットフォーム デバイス ドライバ (PDD) 層に規格化されたインターフェイスを持ち、ドライバに関連したすべてのハードウェア依存プロセスを実行。

モノリシック ドライバ 異なった層に分けられていないドライバ。それはまた、そのドライバが独自の層デザインを持っているとしても、CE の標準モデル デバイス ドライバ (MDD) やプラットフォーム デバイス ドライバ (PDD) 層アーキテクチャに合致しないドライバを意味することもあります。

ミューテックス ミューテックス オブジェクトはそれが何らかのスレッドに所有されていないときは信号がオンに、所有されているときには信号がオフになるように状態が設定されてい

る同期オブジェクト。ミューテックスは、同時に 1 つのスレッドにのみ所有されることができます。それはグローバル変数やハードウェア デバイスといった、ある時点で 1 つのスレッドにのみアクセスされるべきリソースを表現するのに使用されます。

ネイティブ ドライバ タッチ スクリーン、キーボード、またディスプレイ ドライバのみが Windows Embedded CE でネイティブ ドライバとして存在し、デバイス マネージャではなく GWES によって管理される。

OEM アダプテーション層 (OAL) OAL は Windows Embedded CE カーネルとターゲット デバイスのハードウェアの間に理論的に存在するコードの層。物理的に OAL はカーネルの実行可能ファイルを作成するためにカーネル ライブラリとリンクしています。

オペレーティング システム ベンチマーク (OSBench) スケジューラの性能を測るために使用されるツール。

OS デザイン Windows Embedded CE6 R2 オペレーティング システムのカスタマイズされたバイナリ ランタイム イメージを生成する Windows Embedded CE6 R2 用 Platform Builder プロジェクト。

プラットフォーム デバイス ドライバ (PDD) 複数層ドライバの PDD 層はハードウェアと直接インターフェイスを持ち、ハードウェア特有のプロセスを実行。

電源管理 システム全体また各ドライバに D0 (完全にオン) から D4 (完全にオフ) までの電源状態を割り当てることによって、システムの電力消費を制御する。それはユーザーとシステムの活動や特定の要求に基づいてこれらの状態間の切り替えを調整します。

プロセス プロセスは Windows Embedded CE のプログラム。各プロセスは複数のスレッドを持つことが可能です。プロセスはユーザー領域でもカーネル領域でも実行することができます。

生産品質 OAL (PQOAL) PQOAL は OAL 開発のプロセスを単純化し短縮する規格化された OAL 構造。それはコード ライブラリを通して改良されたレベルの OAL コンポーネント化、コード再利用をサポートするディレクトリ構造、集中設定ファイル、またプロセッサ ファミリとハードウェア プラットフォームで一貫したアーキテクチャを提供します。

Quick Fix Engineering (QFE) マイクロソフトのウェブサイトから入手可能な Windows Embedded CE のパッチ。それらはバグを修正し新しい機能を提供します。

リフレクタ サービス 代わりに要求を実行することでユーザー モード ドライバがカーネルとハードウェアにアクセスできるようにするサービス。

レジストリ ハードウェアとソフトウェア コンポーネントのための設定情報を含む Windows Embedded CE のための情報ストア。

リモート パフォーマンス モニタ このアプリケーションはオペレーティング システムのリアル タイム性能を追跡可能。それはまたメモリ使用、ネットワーク待ち時間、またその他の要素も追跡できます。

ランタイム イメージ ハードウェア デバイス上で配置されるバイナリ ファイル。それはまたアプリケーションとドライバのための完全なオペレーティング システムの要求されるファイルも含んでいます。

セマフォ セマフォ オブジェクトは決まった数の同時発生スレッドのみがそれにアクセスできるようにすることで、ハードウェアやソフトウェア リソースへのアクセスを保護する同期オブジェクト。セマフォはゼロから特定の最大値までの間のカウントを保ちます。カウントはスレッドがセマフォ オブジェクトの待ちを完了するたびに減少し、スレッドがセマフォを放出するたびに増加します。カウントがゼロになると、スレッドはセマフォによって保護されたリソースにアクセスできなくなります。セマ

フォの状態はそのカウントがゼロより大きくなったとき信号がオンに、そのカウントがゼロになったときに信号がオフになるように設定されています。

シェル シェルはユーザーの操作をデバイスと解釈するためのソフトウェア。デバイスが起動したときにそれは起動します。既定のシェルは AYGShell と呼ばれ、Windows のデスクトップ バージョンのものと同じようにデスクトップ、[スタート] メニュー、そしてタスク バーを含みます。

ソフトウェア開発キット (SDK) 第三者開発者がカスタマイズされた Windows Embedded CE6 R2 ランタイム イメージのためのアプリケーションを作ることを可能にするために使用される。

ソース ファイル ソース ファイルはサブディレクトリのソース コードのためのマクロ定義を設定するテキスト ファイル。Build.exe がソース コードのコンパイルとリンクの方法を決定するためにこれらのマクロ定義を使用します。

ストリーム インターフェイス ドライバ ストリーム インターフェイス ドライバは、ドライバにより制御されるデバイスのタイプに係わりなく、ストリーム インターフェイス機能に関係するすべてのドライバを指します。GWES によって管理されるネイティブ ドライバ以外のすべてのドライバはストリーム インターフェイスをエクスポートします。

サブプロジェクト OSDesign で容易に集積し、削除し、また再利用できるファイルのセット。

同期オブジェクト 同期オブジェクトは複数のスレッドの実行を調整するためにそのハンドルが待機機能の 1 つに特定され得るオブジェクト。

同期アクセス 2 つかそれ以上の別のスレッドが同じバッファを使用して動作しているとき。ある時点では 1 つのスレッドのみがバッファにアクセスでき、その他のスレッドは現在のスレッドのアクセスが完了するまでアクセスできません。

システム生成 システム生成フェーズはパブリックおよび BSP フォルダをフィルタ処理するために行なわれるビルド プロセスの最初の段階です。それは OS デザインで選択されたコンポーネントと関連付けられたファイルを識別します。このフェーズの間、OS デザインで選択されたコンポーネントは実行可能にリンクされ、OS デザインのフォルダにコピーされます。

システム生成変数 選択された CE 機能が一緒にリンクされる CE ビルド プロセスのシステム生成フェーズへの命令。

Sysintr IRQ に対応する値。関連付けられたイベントに信号を送るのに使用されます。この値は割り込みの応答として割り込みサービス ルーチン (ISR) によって返されます。

ターゲット制御シェル デバッガ コマンドへのアクセスを提供する Platform Builder for Visual Studio でのシェル。ターゲット制御シェルは KITL を通してターゲット システムに接続されたとき入手可能になります。

スレッド スケジューラがオペレーティング システムで管理できる最小のソフトウェア ユニット。1 つのドライバやアプリケーションに複数のスレッドが存在することができます。

ユーザー モード ユーザー モードで読み込まれたドライバとすべてのアプリケーションはユーザー メモリ領域で実行する。それらがこのモードにあるとき、ドライバとアプリケーションはハードウェア メモリに直接のアクセスを持たず、特定の API とカーネルに制限されたアクセスを持ちます。

仮想メモリ 仮想メモリはそれを使用するプロセスへ、連続しているように見せるためにシステムの物理メモリを抽象する方法。Windows Embedded CE 6.0 R2 の各プロセスは利用可能な 2 ギガバイトの仮想メモリ スペースを持ち、プロセスから物理メモリにアクセスするため、このメモリは MmMapIoSpace か OALPAtoVA を使用してプロセスの仮想アドレス スペースにマップされる必要があります。

索引

.bib ファイルのセクション 50
  CONFIG セクション 52
  MEMORY 51
.NET Compact Framework 2.0 4, 31
.NET Compact Framework 2.0 要素 4
.pbcxml ファイル 23
.tks ファイル。テスト キット スイート (.tks) ファイルを参照
/base リンカ設定を重視 53
32 プロセス制限 230
  10
3rdParty フォルダ 24
4 GB アドレス領域 230

A

ActivateDevice 関数 269
ActivateDeviceEx 関数 269
ActiveSync 4, 31, 187
ADEFINES 指示子 65
AdvertiseInterface 関数 275, 296
AllocPhysMem 関数 236, 291, 302
API。アプリケーション プログラミング インターフェイス (API) を参照
ARM ベース プラットフォーム 233
ASSERTMSG マクロ 159
ATM 自動預金受払機 (ATM) を参照
Autoexit パラメータ 191
AUTOSIZE パラメータ 52

B

BinFS。バイナリ ROM イメージ ファイル システム (BinFS) を参照
black シェル 107
BLCOMMON フレームワーク 197
Bluetooth 31
BootArgs。ブート引数 (BootArgs) を参照
BOOTJUMP パラメータ 53
BootLoaderMain 関数 219
BOOTME パケット 222
Bootpart 198
BSP 開発 26
BSP 開発時間の短縮 211
Bsp_cfg.h ファイル 288
BSPIntrInit 関数 289
BSP。ボード サポート パッケージ (BSP) を参照
Build ツール (Build.exe) 61
Build.err ファイル 67, 69
Build.log ファイル 67
Build.wrn ファイル 67
Buildrel エラー 70
BuiltIn レジストリ キー 272
BusEnum。バス列挙子 (BusEnum) を参照
BusTransBusAddrToVirtual 関数 312

C

C インターフェイス 105
CAN。コントローラ エリア ネットワーク (CAN) を参照
CDEFINES エントリ 26
CDEFINES 指示子 65
CE 6.0 OS デザイン テンプレート。デザイン テンプレートを参照
CE システム生成フォルダ 12
CE ストレス ツール 188
CE ターゲット コントロール シェル (CESH) 155
CE ダンプ ファイル リーダー 74, 170, 178
Ce.bib ファイル 50, 60
CeAllocAsynchronousBuffer 関数 307
CeAllocDuplicateBuffer 関数 308
CeCallUserProc 関数 279
CeCloseCallerBuffer 関数 305, 308
CEDebugX。デバッガ拡張コマンド (CEDebugX) を参照
CeFreeAsynchronousBuffer 関数 307
CeFreeDuplicateBuffer 関数 308
CeLog イベント追跡システム 172
  Remote Kernel Tracker ツール 173
  参照名マッチング 175
  シップ ビルド 173
CELogFlush ツール 173
CeOpenCallerBuffer 関数 305, 308
CESH。CE ターゲット コントロール シェル (CESH) を参照
CETest.exe。開発サーバー アプリケーション (CETest.exe) を参照
CETK テスト結果を分析 194
CETK パーサー (Cetkpar.exe) 194
CETK。Microsoft Windows CE テスト キット (CETK) 用 Windows Embedded CE カスタム テスト コンポーネントを参照
Chain.bin ファイル 54
Chain.lst ファイル 54
CLR。共通言語ランタイム (CLR) を参照
Common.bib ファイル 50
COMPRESSION パラメータ 53

CONFIG セクション 52, 88
Config.bib ファイル 88, 235, 282, 302
Console レジストリ パラメータ 102
copylink 10
CoreCon。コア接続 (CoreCon) を参照
CPlApplet API 105
CPU アクセス可能メモリ 196
CPU モニタ 188
CPU 依存ユーザー カーネル データ 233
CreateFile 関数 268
CreateInstance 関数 292
CreateProcess 関数 279
CSV。カンマ区切り値 (CSV) を参照

D

DbgMsg 機能。デバッグ メッセージ (DbgMsg) 機能を参照
DBGPARAM 変数 160
DDI。デバイス ドライバ インターフェイス (DDI) を参照
DDKPCIINFO 構造 312
DDKReg_GetPciInfo 関数 312
DDKReg_GetWindowInfo 関数 312
DDKWINDOWINFO 構造 312
DeactivateDevice 関数 269
DEBUGLED マクロ 159
DEBUGMSG マクロ 159
DefaultSuite パラメータ 191
DEFFILE 指示子 65
DependXX エントリ 99
DestroyInstance 関数 292
DEVFLAGS_LOADLIBRARY フラグ 88
DeviceIoControl 関数 258, 304
DevicePowerNotify 関数 295
DHCP。動的ホスト構成プロトコル (DHCP) を参照
DIRS キーワード 61
DIRS ファイル 61
DIRS_CE キーワード 61
DLLENTRY 指示子 65
DllMain 関数 253
DLL。ダイナミック リンク ライブラリ (DLL) を参照
dpCurSettings 変数 164
DRIVER_GLOBALS 構造 237
DriverDetect パラメータ 191
DriversBuiltIn レジストリ キー 272
DRV_GLB。ドライバ グローバル (DRV_GLB) を参照
DYNLINK 指示子 64

E

Eboot 198
Eboot.bib ファイル 216
Enterprise Terminal 4
Enterprise Terminal デザイン テンプレート 103
EnumDevices 関数 278
ERRORMSG マクロ 159
Ethdbg ブート ローダー 215
eXDI。Extended Debugging Interface (eXDI) を参照
EXEENTRY 指示子 65
export ”C” { } ブロック 105
Extended Debugging Interface (eXDI) 157
Extensible Data Interchange (XDI) 74
Extensible Resource Identifier (XRI) 74

F

FILES セクション 54
Filesys.exe 59, 241
FileSystemPowerFunction 241
FIQ。高速割り込み (FIQ) 行を参照
FIXUPVAR パラメータ 53
FMerge ツール (FMerge.exe) 71
FMerge.exe。FMerge ツール (FMerge.exe) を参照
ForceDuplicate パラメータ 306
FreeIntChainHandler 関数 292
FreePhysMem 関数 236
FSRAMPERCENT パラメータ 53

G

General Purpose Input/Output (GPIO) 95
Getappverif_cetk.bat ファイル 171
GetProcAddress API 254
GIISR。汎用インストール可能 ISR (GIISR) を参照
GPIO。General Purpose Input/Output (GPIO) を参照
Graphical Windows Event System (GWES) 89, 98
GUID。グローバル一意識別子 (GUID) を参照
GUI。グラフィック ユーザー インターフェイス (GUI) を参照
GwesPowerOffSystem 関数 240
GWES。Graphical Windows Event System (GWES) を参照

H

H フラグ 282
HalTranslateBusAddress 関数 302
HdStub。ハードウェア デバッガ スタブ (HdStub) を参照
Heap Walker 155
HookInterrupt 関数 288

I

IClass 値 275, 296
IDE。統合化開発環境 (IDE) を参照
Idle 電源状態 88

IEEE。Institute of Electrical and Electronic Engineers (IEEE) を参照
IISR。インストール可能 ISR (IISR) を参照
ILTiming ツール 90
  パラメータ 91
IL 測定。割り込み待機時間測定 (ILTiming) ツールを参照
IMGNODEBUGGER 環境変数 177
IMGNOKITL 環境変数 177
INCLUDES 指示子 64
INIT レジストリ キー 98
Initdb.ini ファイル 60
Initobj.dat ファイル 59
Institute of Electrical and Electronic Engineers (IEEE) 196
IntelliSense 67
Internet Explorer 4
  サンプル ブラウザ カタログ項目 34
  シン クライアント シェル 103
InterruptDone 関数 285
InterruptInitialize 関数 287
IOControl 関数 254, 292, 296
IP アドレスの構成 101
IPv6 4
ISR の遅延 90
ISR 待機時間 227
ISRHandler 関数 292
ISR。割り込みサービス ルーチン (ISR) を参照
IST 待機時間 227
IST 遅延 90
IST。割り込みサービス スレッド (IST) を参照

J

Joint Test Action Group (JTAG) プローブ 178, 196
JTAG プローブ。Joint Test Action Group (JTAG) プローブを参照

K

K フラグ 282
Kato ロギング エンジン 192
Kato.exe。テスト結果ロガー (Kato.exe) を参照
KdStub 74, 156, 178
Kernel Independent Transport Layer (KITL)
  操作方法 180
  ターゲット コントロール アーキテクチャ 156
  通信インターフェイス 179
  ブート引数 180
  有効 179
Kernel Tracker 155
KERNELFIXUPS パラメータ 53
KernelIoControl 関数 237, 289
KernelStart 関数 224
KITL
  有効にする 10
KITL (Kernel Independent Transport Layer)
  トランスポート機構 74
KITL。カーネル独立トランスポート層 (KITL) を参照

L

LAN。ローカル エリア ネットワーク (LAN) を参照
LaunchXX エントリ 99
LDEFINES 指示子 65
LIBRARY 指示子 64
LoadDriver 関数 88, 253
LoadIntChainHandler 関数 285, 291
LoadKernelLibrary 関数 235
LoadLibrary 関数 88, 253

M

MainMemoryEndAddress 関数 236
Makefile ファイル 66
Makeimg.exe。バイナリ イメージ作成ツール (Makeimg.exe) を参照
MDD。モデル デバイス ドライバ (MDD) を参照
memcpy 308
MEMORY セクション 51
Microsoft Visual Studio 2005 3
  IntelliSense 67
  ウォッチ ウィンドウ 163
  エラー一覧ウィンドウ 68
  カタログ項目ビュー 4
  構成管理 6
  出力ウィンドウ 68
  出力ウィンドウのデバッグ情報 157
  接続オプション 72
  ソリューション エクスプローラ 5
  ターゲット デバイスのデバッグ 180
  ビルド ウィンドウを開くコマンド 48
  ビルド メニュー 43
  ランタイム イメージのビルド 43
  ランタイム イメージをビルド 48
Microsoft Windows CE テスト キット (CETK) 用 Windows Embedded CE カスタム テスト コンポーネント 15
Microsoft カーネル コード 224
MIPS ベース プラットフォーム 233
MIPS。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 (MIPS) を参照
MmMapIoSpace 関数 236, 291, 302
MmUnmapIoSpace 関数 302
MMU。メモリ管理ユニット (MMU) を参照
MODULES および FILES セクションのファイル タイプ定義 56

MODULES セクション 54
My Documents ディレクトリ 59

N

NEWCPLINFO 情報 106
NK メモリ領域 282
Nk.bin ファイル 43
NKCallIntChain 関数 291
NKDbgPrintf 関数 158
NKGLOBALS 構造 224
Nmake.exe。コンパイラとリンカ (Nmake.exe) を参照
NOLIBC=1 指示子 293
NOTARGET 指示子 64

O

OALIntrRequestSysIntr 関数 289
OALIntrStaticTranslate 関数 289
Oalioctl.dll 237
OALPAtoVA 関数 291, 302
OALTimerIntrHandler 関数 90
OAL。OEM アダプテーション層 (OAL) を参照
OEM アダプテーション層 (OAL) 3, 211
  IOCTL コード 237
  OEMInit 関数 226
  アーキテクチャの一般的なタスク 224
  スタートアップ エントリ ポイント 224
  電源管理サポートおよび 238
  ドライバとの間で共有されるリソース 236
  ブート ローダーとの間でのコード共有 224
  プロファイル タイマ サポート関数 227
  割り込み管理関数 287
  割り込み同期機能 284
OEM アドレス テーブル 223
OEMAddressTable テーブル 223, 235
OEMEthGetFrame 関数 220
OEMEthGetSecs 関数 220
OEMEthSendFrame 関数 220
OEMGetExtensionDRAM 関数 236
OEMGLOBALS 構造 224
OEMIdle 関数 239
OEMInit 関数 226, 284
OEMInitGlobals 関数 224
OEMInterruptDisable 関数 287
OEMInterruptDone 関数 285, 287
OEMInterruptEnable 関数 287
OEMInterruptHandler 関数 287
OEMInterruptHandlerFIQ 関数 288
OEMIoControl 関数 237
OEMNMIHandler 関数 242
OEMPlatformInit ルーチン 220
OEMPowerOff ルーチン 240
OEMReadData 関数 220
OEMWriteDebugLED 関数 159
OEM。相手先ブランド供給 (OEM) を参照
OHCI。Open Host Controller Interface (OHCI) を参照
Open Host Controller Interface (OHCI) 289
OpenDeviceKey 関数 277
OPTIONAL_DIRS キーワード 61
OS Access (OsAxS) 156
OS デザイン ウィザード 3, 5, 13, 31
  ボード サポート パッケージ ウィザード ページ 11
  標準シェル 103
  複数のプラットフォームのサポート 11
OS デザイン。オペレーティング システム (OS) デザインを参照
OS デザインのファイルとディレクトリの構造 12
OS デザインのローカライズ 7
OS デザインを複数の BSP と関連付ける 11
OsAxS。OS Access (OsAxS) を参照
OSBench ツール 90, 92
  ソース コード 93
  パラメータ 93
OUTPUT パラメータ 53

P

PAN。パーソナル エリア ネットワーク (PAN) を参照
PBCXML。Platform Builder カタログ XML (PBCXML) を参照
PCI。Peripheral Component Interconnect (PCI) を参照
PCMCIA。Personal Computer Memory Card International Association (PCMCIA) を参照
PDA デバイス デザイン テンプレート 4, 31
PDA。携帯情報端末 (PDA) を参照
PDD。プラットフォーム デバイス ドライバ (PDD) を参照
PerfToCsv パーサー ツール 194
Peripheral Component Interconnect (PCI) 251
Personal Computer Memory Card International Association (PCMCIA) 259
Platform Builder の構成ファイル 12
Platform Builder カタログ XML (PBCXML) 5
Platform Builder 固有のビルド コマンド 47
Platform Builder。Windows Embedded CE 6.0 用 Microsoft Platform Builder を参照
Platform Builder 用設定ファイル 209
Platform.bib ファイル 26, 54
Platform.dat ファイル 59
Platform.reg ファイル 57
pNKEnumExtensionDRAM 関数 236
PortNumber パラメータ 191
POSTLINK_PASS_CMD 指示子 65
PowerOffSystem 関数 241

PQOAL。生産品質 OEM アダプテーション層 (PQOAL) を参照
PRELINK_PASS_CMD 指示子 65
Process Viewer 155
PROFILE パラメータ 53
Program Files ディレクトリ 59
PROGRAM 指示子 64
Project.bib ファイル 50
Project.dat ファイル 59
Projsysgen.bat ファイル 17
Public ソース コード 20
  編集 21
Public ツリーの編集 20

Q

Q フラグ 282
QRimplicit-import 293
QueryPerformanceCounter 関数 94
QueryPerformanceFrequency 関数 94

R

RAM バック マップ ファイル 233
RAM ファイル システム 53, 59
RAM_AUTOSIZE パラメータ 53
RAMIMAGE パラメータ 52
RDEFINES 指示子 65
RDP。リモート デスクトップ プロトコル (RDP) を参照
RDP。リモート デスクトップ プロトコル (RDP) を参照
Readlog ツール 174
Reginit.ini ファイル 60
  HKEY_LOCAL_MACHINE\Drivers\Active 277
RegisterDevice 関数 269
Reldir ディレクトリ 12
ReleaseSemaphore 関数 120
RELEASETYPE 指示子 64
Remote Kernel Tracker ツール 173
RequestDeviceNotifications 関数 278
RESERVED キーワード 303
RESETVECTOR パラメータ 53
RETAILLED マクロ 159
RETAILMSG マクロ 159
ROM Windows ディレクトリ 59
ROM イメージ ビルダ ツール (Romimage.exe) 50
ROM イメージ ファイル システム 198
ROM ベース アプリケーション 58
ROM モジュールのみ信頼 53
ROM_AUTOSIZE パラメータ 53
ROMFLAGS オプション 88
ROMFLAGS パラメータ 53
Romimage.exe。ROM イメージ ビルダ ツール (Romimage.exe) を参照
ROMOFFSET パラメータ 54
ROMSIZE パラメータ 54
ROMSTART パラメータ 54
ROMWIDTH パラメータ 54
RS232 接続 73

S

S フラグ 282
SCM。サービス コントロール マネージャ (SCM) を参照
SDK。ソフトウェア開発キット (SDK) を参照
Serial Peripheral Interface (SPI) 262
ServerIP パラメータ 191
ServerName パラメータ 191
Services.exe。サービス ホスト プロセス (Services.exe) を参照
SetDbgZone 関数 163
SetSystemPowerState 関数 240
SHx ベース プラットフォーム 233
SignalStarted API 99, 108
Simple Windows Embedded CE DLL サブプロジェクト 264
SKIPBUILD 指示子 65
Sleep 関数 88
SOURCELIBS 指示子 64
Sources ファイル 26
  ADEFINES 指示子 65
  CDEFINES エントリ 26
  CDEFINES 指示子 65
  DEFFILE 指示子 65
  DLLENTRY 指示子 65
  DYNLINK 指示子 64
  EXEENTRY 指示子 65
  INCLUDES 指示子 64
  LDEFINES 指示子 65
  LIBRARY 指示子 64
  NOTARGET 指示子 64
  POSTLINK_PASS_CMD 指示子 65
  PRELINK_PASS_CMD 指示子 65
  PROGRAM 指示子 64
  RDEFINES 指示子 65
  RELEASETYPE 指示子 64
  SKIPBUILD 指示子 65
  SOURCELIBS 指示子 64
  SOURCES 指示子 65
  TARGETLIBS 指示子 64
  TARGETNAME 指示子 64
  TARGETPATH 指示子 64
  TARGETTYPE 指示子 64
  WINCE_OVERRIDE_CFLAGS 指示子 65
  WINCECPU 指示子 65

WINCETARGETFILE0 指示子 65
WINCETARGETFILES 指示子 65
Sources ファイルの指示子 65
Sources ファイルの標準指示子 65
SOURCES 指示子 65
SPI。Serial Peripheral Interface (SPI) を参照
SRE パラメータ 54
StartUp 関数 197
StartupProcessFolder 関数 100
Storage Device Block Driver Benchmark Test 190
SuspendThread 関数 114
Svcstart サンプル サービス 101
レジストリ パラメータ 101
SYSGEN 変数
条件式の基づく 56
Sysgen.bat 39
SYSINTR 値 285, 288
SYSINTR_NOP 値 286
SYSINTR_TIMING 割り込みイベント 90

T

TARGETLIBS 指示子 64
TARGETNAME 指示子 64
TARGETPATH 指示子 64
TARGETTYPE 指示子 64
TARGETTYPE=NOTARGET 17
TCP/IPv6 サポート 31
TFTP。簡易ファイル転送プロトコル (TFTP) を参照
TLB。トランジション ルックアサイド バッファ (TLB) を参照
TransBusAddrToVirtual 関数 302
TUX DLL テンプレート 192
Tux.exe。テスト エンジン (Tux.exe) を参照

U

UART。Universal Asynchronous Receiver/Transmitter (UART) を参照
Udevice.exe。ユーザー モード ドライバ ホスト プロセス (Udevice.exe) を参照
UDP。ユーザー データグラム プロトコル (UDP) を参照
Universal Asynchronous Receiver/Transmitter (UART) 196
USB。ユニバーサル シリアル バス (USB) を参照
UserProcGroup レジストリ エントリ 281

V

VirtualAlloc 関数 235, 291, 308
VirtualCopy 関数 235, 291
VirtualFree 関数 235
Visual Studio 2005。Microsoft Visual Studio 2005 を参照
VMM。仮想メモリ マネージャ (VMM) を参照

W

WaitForMultipleObjects 関数 288
WaitForSingleObject 関数 115, 286
WCE TUX DLL テンプレート 192
Win32 API 90
WINCE_OVERRIDE_CFLAGS 指示子 65
WINCECPU 指示子 65
WINCEDEBUG 関数変数 158
WINCETARGETFILE0 指示子 65
WINCETARGETFILES 指示子 65
Windows Embedded CE 6.0 R2 用 Platform Builder ソフトウェア開発キット (SDK) 28
Windows Embedded CE 6.0 用 Microsoft Platform Builder 1, 39
Windows Embedded CE 6.0 用 Microsoft Platform Builder
BSP 複製ウィザード 212
Heap Walker 155
Kernel Tracker 155
OS デザイン ウィザード 3
Process Viewer 155
構成ファイル 12
サブプロジェクト ウィザード 15, 264
詳細デバッガ ツール 170
ターゲット コントロール オプション 167
ターゲット デバイスの接続オプション ダイアログ ボックス 72, 178
デバッグ メッセージ オプション 158
デバッグ領域ダイアログ ボックス 163
ビルド結果の分析 67
Windows Embedded CE Test Kit (CETK) 185
CETK パーサー (Cetkpar.exe) 194
PerfToCsv パーサー ツール 194
zorch パラメータ 190
アーキテクチャ 186
アプリケーション検証ツール 171
カスタム テスト 189
概要 185
クライアント側アプリケーション (Clientside.exe) 186
コマンド ライン パラメータ 190
スケルトン Tux モジュール 193
スタンドアロン モード 192
テスト エンジン (Tux.exe) 186
テスト キット スイート (.tks) ファイル 189
テスト スイート エディタ 189
テスト結果を分析 194
テスト結果ロガー (Kato.exe) 186
マネージ コード 187
ユーザー定義テスト ウィザード 193

ワークステーション サーバー アプリケーション (CETest.exe) 188
Windows Embedded CE サブプロジェクト ウィザード 15, 264
Windows Embedded CE シェル 102
Windows Embedded CE 標準シェル 103
Windows Network Projector 4
Windows Sockets (Winsock) 186
Windows シン クライアント 4
Windows タスク マネージャ (TaskMan) 104
Windows ディレクトリ 59
Windows ベースの Terminal (WBT) シェル 103
Winsock。Windows Sockets (Winsock) を参照
WMV/MPEG-4 Video Codec 4
WordPad 4
WriteDebugLED 関数 159

X

X86 システム上で X86 TLB をフラッシュ 53
x86 ベース プラットフォーム 233
X86BOOT パラメータ 54
XDI。Extensible Data Interchange (XDI) を参照
XIP チェーン 54
XIPSCHAIN パラメータ 54
XIP。エクセキュート イン プレイス (XIP) を参照
XML。拡張マークアップ言語 (XML) を参照
XRI。Extensible Resource Identifier (XRI) を参照
XXX_ プレフィックス 262
XXX_Init 関数 277
XXX_IOControl 関数 295, 304
XXX_PowerDown 関数 295
XXX_PowerUp 関数 295

Z

zorch パラメータ 190

あ

アーキテクチャの一般的なタスク 224
相手先ブランド供給 (OEM) 207
アイドル スレッド 92
アイドル モード 239
アイドル イベント 238
アクセス チェック 305
アクティビティ タイマ 135
アセンブリ言語 198
アドホック ソリューション 23
アドレス テーブル 223
仮想 - 物理 223
アドレス マッピング
アプリケーション デバッグ 157
アプリケーション プログラミング インターフェイス (API) 11
CPlApplet API 105
GetProcAddress API 254
SignalStarted API 99
Win32 API 90
イベント API 121
インターロック API 121
クリティカル セクション API 117
ストリーム インターフェイス API 257, 260
スレッド管理 API 110
非リアルタイム 89
ファイル システム API 257
プロセス管理 API 110, 137
ミューテックス API 118
アプリケーション検証ツール 171, 188
アプリケーション呼び出しバッファ 303
アプレット 104
アラート 95

い

イーサネット サポート関数 220
イーサネット ダウンロード サービス 73
依存関係の処理 99
一般的なレジストリ エントリ、デバイス ドライバ 274
新しいシステム 230
カーネル領域 230
周辺機器のフレーム バッファおよび 235
初期化 223
静的にマップされたアドレス 234
動的にマップされたアドレス 235
入出力処理および 235
非連続物理メモリおよび 235
マッピング テーブル 223
未キャッシュ 234
ユーザー領域 230
イベント API 121
イベントの追跡 9
イベントログ記録ゾーン 172
イメージ構成ファイル 60
医療モニタ機器 107
インスタンス固有リソース 263
インストール可能 ISR (IISR) 292
DLL 関数 292
アーキテクチャ 292
外部依存関係 293
登録 293
プラグ アンド プレイ 292
インターナショナライゼーション (国際化) 7
既定のロケール 8

コード ページ 8
ロケール 8
インターフェイス GUID 275
動的 289
インターフェイス マッピング
インターロック API 121
インターロックされた API 225

う

ウィンドウズ マネージャ 241
ウィンドウの描画 89
ウォッチ ウィンドウ 163, 170
埋め込みポインタ 300, 304

え

エクセキュート イン プレイス (XIP) 231
エラー レポート生成プログラム カタログ項目 74
エラー一覧ウィンドウ 68
エラー発生後のデバッガ 74

お

オーディオ デバイス ドライバ レジストリ 274
オーバーラン バッファ 286
オープン コンテキスト 263
オブジェクト ストア 58
オプション 10
black シェル 107
Windows タスク マネージャ (TaskMan) 104
Windows ベースの Terminal (WBT) シェル 103
依存関係の処理 99
インターナショナライゼーション (国際化) 7
カーネル オブジェクト 89
カスタマイズ 5
環境変数 10
管理コード開発 35
キオスク モード” 107
言語設定 7
高度な構成 11
コマンド プロセッサ シェル 102
コンポーネント化 97
再頒布と OS デザイン 12
作成とカスタマイズ 3
シェル 102
システム アプリケーション 97
シン クライアント シェル 103
ソース コード 20
デザイン 1
デバイス マネージャ 88
電源管理 88
パフォーマンスの最適化 10
標準シェル 103
ビルド オプション 3
フットプリント 1
要素 3
ランタイムイメージ 1
リアルタイム パフォーマンス 94
オペレーティング システム (OS)
オペレーティング システム ベンチマーク (OSBench)。
OSBench を参照
オペレーティング システムのフットプリント 1

か

カーネル アクセス チェック 305
カーネル アドレス領域 231
カーネル オブジェクト 89
スレッド同期および 116
カーネル スタートアップ サポート関数 225
カーネル デバッガ 10, 157, 177
KdStub 74, 156
アプリケーション デバッグ 157
ランタイム情報を取得 156
例外処理および 126
カーネル ドライバの制約 279
カーネル プロセス (Nk.exe) 303
カーネル プロファイラ 10
カーネル メモリ領域 232
カーネル モード ドライバ 279
カーネル依存トランスポート層 (KITL)
カーネル初期化ルーチン 197
カーネル独立トランスポート層 (KITL) 3
カーネルの静的マッピング領域 233
サポート関数 226
カーネル領域 230
カーネル割り込みマッピング配列 289
開始時間の短縮 4
開発サーバー アプリケーション (CETest.exe) 186
開発サイクル 153
埋め込みポインタのマーシャリング 305
開発ボード機能の実演 4
拡張マークアップ言語 (XML) 5
カスタム CETK テスト 192
カスタム デザインテンプレート 5
エラー レポート生成 74
仮想アドレス領域
仮想アドレス領域
仮想 - 物理アドレス マッピング 223
仮想メモリ
仮想メモリ マネージャ (VMM) 300
仮想メモリの管理の新しいシステム 230
仮想メモリの初期化 223

カタログ エディタ
カタログ システム 23
カタログ ファイル 23
カタログからのカタログ項目のエクスポート 26
カタログ項目 3
.pbcxml ファイル 23
3rdParty フォルダ 24
BSP 開発 26
ID 25
Internet Explorer 6.0 サンプル ブラウザ カタログ項目 34
OS デザインで追加または削除 47
Public ディレクトリ ツリーから BSP コンポーネントへの変換 22
Windows Embedded CE 標準シェル 103
依存関係 27
エクスポート 26
カタログ項目複製オプション 21
管理 23
作成と編集 24
条件付き処理の基づく 56
バックライト ドライバ 27
東アジア言語 7
複製 20
プロパティ 24
カタログ項目の依存関係 5
カタログ項目の依存関係ウィンドウ 6
カタログ項目の検索 5
カタログ項目ビュー 4, 33
カタログ項目の検索 5
カタログ項目複製オプション 21
フィルタ項目 5
カタログ項目ビュー ソリューション エクスプローラ 5
表示項目の依存関係 57
カタログ項目表示
カタログのエントリのプロパティ 24
簡易ファイル転送プロトコル (TFTP) 198
環境オプション 10
環境変数 10
_TARGETPLATROOT 214
IMGNODEBUGGER 177
IMGNOKITL 177
WINCEDEBUG 158
条件式の基づく 56
環境変数に基づく指示子 43
カンマ区切り値 (CSV) 194
管理コード開発 35

き

キーボード イベント 284
キオスク モード 107
マネージ アプリケーション 108
基幹セクション 89
既存のスレッド 111
既定のロケール 8
起動時にデバッグ領域をオーバーライド 164
起動の構成 97
基盤となるハードウェアとオペレーティング システム間の抽象的なレイヤ 253
逆アセンブリ ツール 171
境界間のマーシャリング データ 300
競合条件 155
共通言語ランタイム (CLR) 108
共通リリース ディレクトリ 39
共有割り込みマッピング 291

く

クライアント側アプリケーション (Clientside.exe) 186, 190
スタンドアロン モード 192
開始パラメータ 191
グラフィック ユーザー インターフェイス (GUI) 103
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) 253
クリーン システム生成コマンド 45
クリティカル オフ状態 238, 242
クリティカル セクション 116
クリティカル セクション API 117
グローバル一意識別子 (GUID) 275

け

携帯情報端末 (PDA) 238
言語設定 7
現在の BSP とサブプロジェクトの再構築 26

こ

コア デバッグ ツール 176
コア接続 (CoreCon) 19
インフラストラクチャ 72
ターゲット コントロール アーキテクチャ 156
トランスポート機構 74
レイヤのダウンロード 73
構成管理 6
構成管理プログラム データベース (.pdb) ファイル 6
構成マネージャ 11
高速割り込み (FIQ) 行 288
高パフォーマンス カウンタ 90
コード ページ 8
コード再利用 207
コードの再利用 207
コードの再利用性の向上 213

コマンド プロセッサ シェル 102
コマンド ライン ツールに基づくカスタム ビルド アクション 65
コンシューマ メディア デバイス デザイン テンプレート 4
コンテキスト管理 263
  オープン コンテキスト 263
  デバイス コンテキスト 263
コントローラ エリア ネットワーク (CAN) 4
コントロール パネル 104
  CPlApplet API 105
  NEWCPLINFO 情報 106
  コンポーネント 104
  ソース ファイル 106
  メッセージ 106
コンパイラ エラー 67
コンパイラとリンカ (Nmake.exe) 61
コンパイル フェーズ 41
コンポーネント化したオペレーティング システム 97
コンポーネントの複製 20
  Public ツリーの編集 20
  カタログ項目複製オプション 21, 25
  ボード サポート パッケージ (BSP) および 211

さ

サービス コントロール マネージャ (SCM) 101
サービス ホスト プロセス (Services.exe) 101
再開ソース 242
最近値シンボル一覧 171
最終構成におけるシステムの検証 153
最新の正常な構成 58
サスペンド状態 238, 240
サスペンド状態からの再開 241
サブプロジェクト 3
  CreateFile 関数 268
  Dirs ファイル 61
  IOCTL_HAL_REQUEST_SYSINTR および IOCTL_HAL_RELEASE_SYSINTR 289
  OEMAddressTable テーブル 223
  OEMPlatformInit 関数 220
  Projsysgen.bat ファイル 17
  TARGETTYPE=NOTARGET 17
  イメージの設定 18
  カスタム設定を再利用 49
  構成 14, 18
  構成ファイル 15
  作成と追加 15
  サブプロジェクト ウィザード 15
  システム生成変数 17
  種類 14
  ストリーム関数の実装 265
  スレッド管理 115
  静的ライブラリ 17
  ソースコードなし 17
  ダイナミック リンク ライブラリ (DLL) 17
  デバイス コンテキストの初期化 264
  ドライバを動的にロードする 269
  非同時バッファ アクセス 306
  ランタイム イメージから除く 18
  レジストリ設定 17
  割り込みサービス スレッド (IST) 287
産業用制御装置 107
参照名マッチング 175
サンプル コード
サンプル デバイス エミュレータ eXDI2 ドライバ 74, 178

し

シェル 102
  black シェル 107
  Windows タスク マネージャ (TaskMan) 104
  Windows ベースの Terminal (WBT) 103
  コマンド プロセッサ セル 102
  シン クライアント シェル 103
  標準シェル 103
システム アプリケーション 97
システム スケジューラ 88
システム タイマ 88
システム テスト 153, 185
  監視 87
  最適化 87
  リアルタイム オペレーティング システム 87
システム パフォーマンス
システム メモリ プール 89
システム メモリ マッピング 230
システム メモリの再利用 89
システム生成キャプチャ ツール 21
システム生成フェーズ 42
  エラー 69
システム生成変数 10
  サブプロジェクト 17
システム電源状態 134
システムの全体的な健全性診断 168
システムのテスト 153
  自動化 185
システムのプログラミング 85
シップ ビルド 173
自動ソフトウェア テスト 185
自動的なドライバのロード 270
自動的に開始 97
自動変数ツール 170
自動預金受払機 (ATM) 107
ジャストインタイム (JIT) デバッグ 157
修復不能なロックアップ 296

周辺機器のフレームバッファ 235
出力ウィンドウ 68
条件式およびデバッグ 166
条件付きファイル処理 56
詳細デバッガ ツール 170
詳細なビルド コマンド 26, 45
現在の BSP とサブプロジェクトの再構築 26
詳細メモリ ツール 171
ショートカット ファイル 100
初期化されていない変数 155
シリアル デバッグ出力関数 219
シリアル通信パラメータ 73
シン クライアント シェル 103
シン クライアント デザイン テンプレート 4
シンボル 171
信頼できるイメージ 222

す

スケルトン Tux モジュール 193
スタート メニュー 59
スタートアップ フォルダ 100
制限 101
スタートアップ レジストリ パラメータ 99
遅延スタートアップ 101
スタートアップの構成
スタベーション 170
スタンドアロン モード 192
ストリーム インターフェイス API 260
ストリーム関数のエクスポート 266
ストリーム インターフェイス ドライバ 257
ストリーム ドライバ 253
CreateFile 関数 268
XXX_ プレフィックス 262
インスタンス固有リソース 263
カーネル モードの制約 279
コンテキスト管理 263
ストリーム関数のエクスポート 266
ソース ファイル指示子 267
デバイス名 259
名前付け規則 258
プラグ アンド プレイ 257
レガシ名 259
ロードおよびアンロード 257, 269
ロード手順 272
ストリーム ドライバ。ストリーム インターフェイス ドライバも参照
ストリーム ドライバの従来の名前付け規則 259
ストリーム関数のエクスポート 266
すべてのデバッグ領域を有効化 165
スモール フットプリント デバイス 87
スモール フットプリント デバイス デザイン テンプレート 4
スレッド 88
アイドル 92
管理機能 111
既存 111
再開 114
作成 111
終了 111
スタベーション 170
中断 114
同期 109, 116
不意の同期 166
優先度 113
優先度レベル 113
スレッド ツール 170
スレッド管理 API 110
不意 166
割り込み処理 284
スレッド同期
スレッドの再開 114
スレッドの作成 111
スレッドの終了 111
スレッドの中断 114
スレッド優先順位 88

せ

生産品質 OEM アダプテーション層 (PQOAL) 207
高度なデバッガ ツール 211
脆弱性 305
静的にマップされた仮想アドレス 234
静的ライブラリ 17
制約、電源管理 296
接続オプション 72
セマフォ 119
ReleaseSemaphore 関数 120

そ

ソース コード 20
Eboot.bib ファイル 216
Windows タスク マネージャ (TaskMan) 104
コントロール パネル 104
サンプル コードの管理 115
デバイス ドライバのためのフォルダ 228
ドライバ グローバル 217
ソース コードの構文チェック 67
ソース コントロール ソフトウェア 12
ソース ファイル 63
コントロール パネル 106
ソース ファイル指示子、デバイス ドライバ 267

ソフトウェア開発キット (SDK) 28
新しいファイルの追加 29
インストール 30
構成と生成 28
生成とテスト 38
ビルド プロセスおよび 42
ソフトウェア開発サイクル 153
ソフトウェア関連エラー 155
ソフトウェア例外 126
ソリューション エクスプローラ 5, 43
Dirs ファイル 63
カタログ項目の依存関係ウィンドウ 6
カタログ項目ビュー 5
サブプロジェクトウィザード 15
プロパティ ページ ダイアログ ボックス 7

た

ターゲット コントロール アーキテクチャ 156
ターゲット コントロール コマンド 168
ターゲット コントロール サービス 167
Windows Embedded CE のロード 72
接続 75
通信パラメータの定義 72
デバッガ オプション 74
ファイル システムおよびシステム レジストリの初期化 49
ターゲット コントロール シェル。CE ターゲット コントロール シェル (CESH) を参照
ターゲット デバイス
ターゲット デバイス コントロール 155
ターゲット デバイスの欠陥 153
ターゲット デバイスの接続オプション ダイアログ ボックス 72, 178
ターゲット デバイスへの接続 75
ターミナル サーバー 103
OALTimerIntrHandler 関数 90
SYSINTR_TIMING 割り込みイベント 90
システム タイマ 88
電源管理および 135
ハードウェア タイマ 88
ダイナミック リンク ライブラリ (DLL) 17
C インターフェイス 105
デバイス ドライバ 253
タイマ
タイマ イベント 284
ダウンロード進行状況表示 222
ダウンロード方法 72, 196
Build.exe 61
CE ストレス ツール 188
CELogFlush ツール 173
CETest.exe 186
Cetkpar.exe 194
Clientside.exe 186, 190
CPU モニタ 188
Filesys.exe 59
FMerge (FMerge.exe) 71
Heap Walker 155
IL 測定 227
ILTiming 90
Kato.exe 186
Kernel Tracker 155
Nmake.exe 61
OSBench 90
PerfToCsv パーサー 194
Process Viewer 155
Readlog ツール 174
Remote Kernel Tracker 173
ROM イメージ ビルダ (Romimage.exe) 50
Sysgen.bat 39
Tux.exe 186
Windows タスク マネージャ (TaskMan) 104
アプリケーション検証ツール 171, 188
ウォッチ ウィンドウ 170
逆アセンブリ ツール 171
コントロール パネル 104
最近値シンボル一覧 171
システム生成キャプチャ ツール 21
自動変数ツール 170
詳細デバッガ ツール 170
詳細メモリ ツール 171
スレッド ツール 170
デバッグおよびテスト 153
バイナリ イメージの作成 (Makeimg.exe) 50
バイナリ イメージ作成 (Makeimg.exe) 39
ブレークポイント 170
プロセス ツール 171
メモリ ツール 171
モジュール ツール 170
呼び出し履歴ツール 170
リアルタイム パフォーマンスの測定 90
リソース消費ツール 188
リモート パフォーマンス モニタ 90, 94
レジスタ ツール 171
単一スレッド モード 296

ち

遅延 90
ISR と IST 90
遅延スタートアップ 101
Svcstart サンプル 101
逐次アクセス シナリオ 307

て

ティック タイマ 90
データ整合性 58
  OS デザインの概要 3
  インターナショナライゼーション (国際化) 7
  オペレーティング システム (OS) 1
  カタログ項目 3
  環境変数 10
  言語設定 7
  高度な構成 11
  サブプロジェクト 3
  ファイルとディレクトリの構造 12
  ブルド オプション 3
データベース (.db) ファイル 49, 58
デザイン
デザイン テンプレート 4

    ARMV4I 31
  Enterprise Terminal 103
  PBCXML 構造 5
  PDA デバイス 4, 31
  カスタム 5
  コンシューマ メディア デバイス 4
  シン クライアント 4
  スモール フットプリント デバイス 4
デザイン テンプレート バリエーション 4
デザイン再頒布 12
デスクトップのアプリケーション ショートカット 59
デスクトップのショートカット 59
テスト アクセス ポートおよび境界スキャン テクノロジ 196
テスト エンジン (Tux.exe) 186
  コマンド ライン パラメータ 192
テスト キット スイート (.tks) ファイル 189
テスト スイート 189
テスト結果ロガー (Kato.exe) 186
デッドロック 153, 168
  ARMV4I 31
デバイス エミュレータ
デバイス エミュレータ (DMA) 73
デバイス クラス 142
デバイス コンテキスト 263
デバイス コンテキストの初期化 264
デバイス ドライバ 14
  DllMain 関数 253
  IClass 値 275
  IOControl 関数 254
  OAL との間で共有されるリソース 236
  アプリケーション呼び出しバッファ 303
  カーネル モードの制約 279
  開発 251
  コンテキスト管理 263
  ストリーム ドライバ 253
  ソース コード フォルダ 228
  ソースファイル指示子 267
  通信用有メモリ領域 237
  デバイス レジスタ アクセス 237
  電源管理 295
  電源状態 133
  名前付け規則 258
  ネイティブ ドライバ 253
  バスを認識しない 313
  ビルド 264
  複数層ドライバ アーキテクチャ 254
  ページング 253
  ボード サポート パッケージ (BSP) および 211
  モノシリック ドライバ 254
  リフレクタ サービス 279
  レガシ名 259
  レジストリ エントリ 274
  ロードおよびアンロード 257, 269
  ロード手順 272
  割り込みハンドラ 284
デバイス ドライバ インターフェイス (DDI) 253
デバイス ドライバの移植性 310
デバイス マネージャ 88
  概要 257
  シェル 257
  ストリーム ドライバ相互作用 254
  ブート時にデバイス ドライバをロード 272
  レジストリの設定 99
デバイス レジスタ アクセス 237
デバイス名 259
デバッガ オプション 74
デバッガ拡張コマンド (CEDebugX) 168
デバッグ 6, 153
  CE ダンプ ファイル リーダー 170
  Tux DLL 194
  アセンブリ言語 198
  エラー発生後のデバッガ 74
  カーネル デバッガ 74
  条件式 166
  シリアル デバッグ出力関数 219
  詳細 158, 166
  ターゲット コントロール コマンド 168
  デバッグ ゾーン 159
  デバッグ メッセージのマクロ 158
  ハードウェア デバッグ インターフェイス 74
  ハードウェア補助 178
  不可欠なコンポーネント 157
  ブート ローダー 198
  ブレークポイント 157
  ボード サポート パッケージ (BSP) 177
  有効 177
  リテール マクロ 158

リリース ビルドからデバッグ コードを除外 166
割り込みハンドラ 183
デバッグ ゾーン 159
デバッグ メッセージ (DbgMsg) 機能 155
デバッグ メッセージ オプション 158
デバッグ メッセージ サービス 157, 164
デバッグ メッセージの動的管理 158
デバッグ メッセージのマクロ 158
ASSERTMSG 159
DBGPARAM 変数 160
DEBUGLED 159
DEBUGMSG 159
ERRORMSG 159
RETAILLED 159
RETAILMSG 159
デバッグ ゾーン 159
デバッグのリテール マクロ 158
DBGPARAM 変数 160
dpCurSettings 変数 164
SetDbgZone 関数 163
Tux DLL 194
すべて有効化 165
ウォッチ ウィンドウ 163
起動時にオーバーライド 164
ダイアログ ボックス 163
定義 162
登録 160
バイパス 160
ベスト プラクティス 165
有効化と無効化 163
レジストリ設定 164
デバッグ領域
デバッグ領域のベスト プラクティス 165
デマンド ページング 53, 87
電源オフ状態 238
電源管理 88, 132
電源状態
I/O コントロール (IOCTL) 297
Idle 電源状態 88
InCradle 135
OEM アダプテーション層 (OAL) 238
アイドル イベント 238
アクティビティ タイマおよび 135
アクティビティ タイマおよび 135
アプリケーション インターフェイス 140
アプリケーション インターフェイス 140
切り替え 238
クリティカル オフ状態 238, 242
構成 141
コンテキストの切り替え 89
再開ソースおよび 242
サスペンド状態 238, 240
サスペンド状態からの再開 241
システム 134
システム電源状態 134
制約 296
単一スレッド モード 296
通知インターフェイス 297
デバイス クラスおよび 142
デバイス ドライバ 295
デバイス ドライバ 295
電源オフ状態 238
ドライバ電源状態 133
電源管理 (PM.dll)
テンプレート バリエーション 4

と

同期
同期メモリ アクセス 306
統合化開発環境 (IDE) 5
動的にマップされた仮想アドレス 235
動的ホスト構成プロトコル (DHCP) 101, 198
動的メモリ割り当て 128
スレッド 109
不意 166
ドライバ グローバル (DRV_GLB) 217
ドライバ通信用共有メモリ領域 237
ドライバ電源状態 133
ドライバと OAL 間で共有されるリソース 236
ドライバを動的にロードする 269
トラップ ハンドラ 284
ビルドの問題 69
トラブルシューティング
トランザクション ベース保存機構 58
トランジション ルックアサイド バッファ (TLB) 224
トランスポート機構 72, 74

な

内部テスト アプリケーション 14
名前付け規則、ドライバ 258
Internet Explorer 4
OS デザイン 3
WordPad 4
カタログ項目 3

に

入出力処理 235, 258

ね

ネイティブ ドライバ 253

は

パーソナル エリア ネットワーク (PAN) 31
ハードウェア タイマ 88, 90
ハードウェア デバッガ スタブ (HdStub) 156
ハードウェア デバッグ インターフェイス 74
ハードウェア ブレークポイント 183
ハードウェア依存コード 211
ハードウェア衝突 153
ハードウェア初期化タスク 220
ハードウェアの検証 95
ハードウェア補助デバッグ 178
バイナリ ROM イメージ ファイル システム (BinFS) 198
バイナリ イメージ ビルダ (.bib) ファイル 49
AUTOSIZE パラメータ 52
BOOTJUMP パラメータ 53
COMPRESSION パラメータ 53
CONFIG セクション 52
FILES セクション 54
FIXUPVAR パラメータ 53
FSRAMPERCENT パラメータ 53
H フラグ 282
K フラグ 282
KERNELFIXUPS パラメータ 53
MEMORY セクション 51
MODULES セクション 54
NK メモリ領域 282
OUTPUT パラメータ 53
PROFILE パラメータ 53
Q フラグ 282
RAM_AUTOSIZE パラメータ 53
RAMIMAGE パラメータ 52
RESETVECTOR パラメータ 53
ROM_AUTOSIZE パラメータ 53
ROMFLAGS パラメータ 53
ROMOFFSET パラメータ 54
ROMSIZE パラメータ 54
ROMSTART パラメータ 54
ROMWIDTH パラメータ 54
S フラグ 282
SRE パラメータ 54
X86BOOT パラメータ 54
XIPSCHAIN パラメータ 54
セクション 50
ファイル タイプ定義 56
不連続メモリおよび 52
自動スタートアップ 50
条件付き処理 56
バイナリ イメージの作成ツール (Makeimg.exe) 39, 50
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 (MIPS) 293
波形発生器 95
バス ドライバ 260
バス名アクセス 260
バス列挙子 (BusEnum) 272
バスを認識しないドライバ 313
バックライト ドライバ 27
バッテリの著しい低下状態 238
バッテリ レベルがゼロになる 242
バッテリ寿命
バッファ マージャリング 279
バッファ処理 306
パフォーマンスの監視 87
アラート 95
チャート 95
波形発生器 95
レポート 95
割り込み遅延タイミング 90, 95
パフォーマンスの最適化 10, 87
汎用インストール可能 ISR (GIISR) 293

ひ

ヒープ 89
東アジア言語 7
ビデオ メモリ 241
非同期バッファ アクセス 300, 306
表示項目依存関係 57
標準コマンド プロンプト 48
標準シェル 103
削除 107
非リアルタイム API 89
非リアルタイム コンポーネント 87
ビルド ウィンドウを開くコマンド 48
ビルド オプション 3
ビルドオプション
32MB を超えるランタイム イメージを許容 10
KITL を有効にする 10, 177
アクティブな OS デザイン 9
カーネル デバッガを有効にする 10, 177
追跡されたイベントを RAM の中にバッファリングする 9
追跡したイベントをリリース ディレクトリに書き込む 10
ハードウェアによるデバッグ サポートを有効にする 9
ブート中にイベントの追跡を有効にする 9
プロファイルを有効にする 10
メモリ上の Eboot スペースを有効にする 9
ランタイム イメージをフラッシュ メモリに書き込む 10
ビルド コマンド 45
対応するコマンド ライン 48
ビルド フェーズ 42
エラー 70
ビルド プロセス 39, 41
Platform Builder および 39

Visual Studio 43
エラー 67
環境変数に基づく指示子および 43
コマンド ライン ツールに基づくカスタム アクション 65
コンパイル フェーズ 41
システム生成フェーズ 42
詳細なビルド コマンド 45
標準コマンド プロンプトおよび 48
ソフトウェア開発キット (SDK) および 42
バッチ ファイルおよび 41
ビルド フェーズ 42
ビルド ログ ファイル 68
ビルド結果の分析 67
フェーズ 41
ランタイム イメージの作成フェーズ 43
リリース コピー フェーズ 42
リリース コピー フェーズのスキップ 44
リリース ディレクトリにファイルをコピー コマンド 44
ビルド プロセス中のエラー 67
ビルド プロセスのコントロール 43
ビルド メニュー 43
ビルド レポート 67
リリース ディレクトリの作成にリンクではなく xcopy を使用する 10
ビルド結果の分析 67
ビルド構成管理 6
.bib ファイル 49
.dat ファイル 49, 59
.db ファイル 49, 58
.pbcxml ファイル 23
.reg ファイル 49, 57
.tks ファイル 189
Bsp_cfg.h 288
Build.err 67
Build.log 67
Build.wrn 67
Ce.bib 50, 60
Chain.bin 54
Chain.lst 54
Common.bib 50
Config.bib ファイル 88, 235, 302
Device.dll 257
Devmgr.dll 257
Dirs ファイル 61
Eboot.bib 216
Initdb.ini 60
Initobj.dat 59
Makefile ファイル 66
Nk.bin ファイル 43
Oalioctl.dll 237
Platform.bib ファイル 26, 54
Platform.dat 59
Platform.reg 57
Project.bib 50
Project.dat 59
Reginit.ini 60
Sources ファイル 26
Sysgen.bat 39
Udevice.exe 279
イメージ構成ファイル 60
環境オプション 10
クリーン システム生成コマンドおよび 48
構成ファイル 15
サブプロジェクト イメージの設定 18
ショートカット ファイル 100
詳細なビルドコマンド 26
ソース コントロール ソフトウェア 12
ソース ファイル 63
ビルド オプション 9
ビルド構成ファイル 61
ビルド指示子 63
プロジェクトのプロパティ 7
ビルド構成のデバッグ 11
ビルド構成のデバッグ 6
ビルド構成ファイル 61
非連続物理メモリ 235
品質保証 153

ふ

ファイル
ファイル システム (.dat) ファイル 49, 59
ファイル システム API 257
ファイル入出力処理 258
不意のスレッド同期 166
ブート ローダー
ブート ローダーと OAL 間のコード共有 224
ブート ローダーのスタートアップ エントリ ポイント 218
ブート ローダーのメニュー 222
ドライバ グローバルおよび 217
ブートストラップ サービス 156
ブートの引数 (BootArgs)
ブート引数 (BootArgs) 180
ブート前のルーチン 196
不揮発性データ記憶 58
複数層ドライバ 255
複数層ドライバ アーキテクチャ 254
複数のプラットフォームのサポート 11
複数のプラットフォームをサポートするデザイン 11
複製ウィザード 212
物理メモリ アクセスの制約 302
物理メモリの割り当て 302
プラグ アンド プレイ 257, 292
フラグ レジストリ値 276

フラッシュ メモリ サポート 221
プラットフォーム デバイス ドライバ (PDD) 255
プラットフォーム固有ソース コード 215
フル カーネル モード 53
  BLCOMMON フレームワーク 197
  BootLoaderMain 関数 219
  BOOTME パケット 222
  Bootpart 198
  Eboot 198
  Ethdbg 215
  OAL との間でのコード共有 224
  アーキテクチャ 196
  アセンブリ言語 198
  イーサネット サポート関数 220
  イーサネット経由のランタイム イメージのダウンロード 220
  一般的なタスク 196
  カーネル初期化ルーチン 197
  シリアル デバッガ出力関数および 219
  スタートアップ エントリ ポイント 218
  テスト 196
  デバッグ テクニック 198
  ドライバ グローバルおよび 217
  ネットワーク ドライバ 198
  ハードウェア初期化タスク 220
  バイナリ ROM イメージ ファイル システム (BinFS) 198
  フラッシュ メモリ サポート 221
  ボード サポート パッケージ (BSP) および 211
  メニュー 222
  メモリ マッピング 216
ブレークポイント 157, 170
  Tux DLL 194
  制限 183
  設定数が多すぎる 183
  ハードウェア 183
  有効化および管理 181
  割り込みハンドラ 183
ブレークポイントを設定 181
不連続メモリ 52
プログラム データベース (.pdb) ファイル 6
プロセス アドレス領域 232
プロセス ツール 171
プロセスおよびスレッド 109
プロセス間通信 233
プロセス管理 API 110
プロファイル タイマ サポート関数 227
プロファイルを有効にする 10
プロフェッショナルの Windows Embedded CE ソリューション 23

へ

ペガソス レジストリ キー 165

ほ

ポインタ パラメータ 304
ポインタ マーシャリング 305
ボード サポート パッケージ (BSP) 3, 177, 207
    fobK c[
  OEM アダプテーション層 (OAL) および 211
  開発時間の短縮 211
  既存の参照 BSP の複製 211
  コンポーネント 210
  シリアル デバッガ出力関数および 219
  設定ファイル 209, 211
  適応と設定 209
  デバイス ドライバおよび 211
  デバイス ドライバのためのソース コード フォルダ 228
  ハードウェア依存コードおよび 211
  複製ウィザード 212
  フォルダ構造 213
  ブート ローダーおよび 211
  プラットフォーム特有のソース コード 215
  メモリ マッピング 230
ボード サポート パッケージ (BSP) のコンポーネント 210
ボード サポート パッケージ (BSP) のフォルダ構造 213
ボード サポート パッケージ ウィザード ページ 11
ホスト プロセス グループ 281

ま

マーシャル ヘルパー 305
マウス テスト 192
前処理条件 57
マッピング テーブル 223
  Windows Embedded CE Test Kit (CETK) 187
  キオスク モード 108
マネージ アプリケーション
マルチビン イメージ通知 222

み

未キャッシュの仮想アドレス 234
ミューテックス 89, 117
  ミューテックス API 118

む

無限ループ 155
無通信のタイムアウト 238

め

メモリ アクセス 300
同期 306
非同期 306
例外処理 308
メモリ ツール 171
メモリ マッピング 216
メモリ リーク 153, 168
メモリ レイアウト 49
BSP のメモリ マッピング 230
カーネル領域 232
システム メモリからの予約領域 303
プロセス領域 233
メモリ管理
メモリ管理ユニット (MMU) 223, 233, 300
メモリ分割ルーチン 198
メモリマップされたファイル 233
ARM ベース プラットフォーム 233
DEVFLAGS_LOADLIBRARY フラグ 88
LoadDriver 関数 88
LoadLibrary 関数 88
MIPS ベース プラットフォーム 233
ROMFLAGS オプション 88
SHx ベース プラットフォーム 233
x86 ベース プラットフォーム 233
基幹セクション 89
システム メモリ プール 89
システム メモリの再利用 89
静的にマップされた仮想アドレス 234
デマンド ページング 87
動的にマップされた仮想アドレス 235
動的割り当て 128
ヒープ 89
非連続物理メモリおよび 235
プロセス 89
ミューテックス 89
メモリ共有 87
メモリ領域 232

も

モジュール ツール 170
モデル デバイス ドライバ (MDD) 20, 255
モノシリック ドライバ 255
モノシリック ドライバ アーキテクチャ 254

ゆ

ユーザー アプリケーション 97
ターミナル サーバー 103
ユーザー データグラム プロトコル (UDP) 198
ユーザー モード ドライバ 279
ユーザー モード ドライバ ホスト プロセス (Udevice.exe) 279
アプリケーション呼び出しバッファ 303
レジストリ エントリ 281
ユーザー定義テスト ウィザード 193
ユーザー領域 230
ユニバーサル シリアル バス (USB) 73

よ

要素
呼び出し履歴ツール 170

ら

ランタイム イメージ 1
カスタム 設定を追加 49
構成ファイル 60
コマンド ラインからビルドおよび展開 48
コンテンツ 49
サブプロジェクトを除く 18
ダウンロード方法 196, 220
展開 72
ビルドおよび展開 39
ランタイム イメージから除く 18
ランタイム イメージの作成フェーズ 43
エラー 71
ランタイム イメージの展開 72
ランタイム イメージをフラッシュ メモリに書き込む 10

り

リアルタイム システムのデザイン 86
リアルタイム パフォーマンス 87, 94
測定 90
リソース消費ツール 188
リビルド コマンド 45
リフレクタ サービス 279
リモート デスクトップ プロトコル (RDP) 4, 103
リモート パフォーマンス モニタ 90
拡張 DLL 94
監視されるオブジェクト 94
領域定義 162
領域登録 160
リリース コピー フェーズ 42
エラー 70
スキップ 44
リリース コピー フェーズのスキップ 44
リリース ディレクトリ 43
リリース ディレクトリにファイルをコピー コマンド 44
リリース ビルドからデバッグ コードを除外 166
リリース構成 6, 11

リンカ警告およびエラー 67

れ

例外処理 125
カーネル デバッガおよび 126
メモリ アクセス 308
構文 127
レガシ名 259
レジスタ ツール 171
レジストリ (.reg) ファイル 49, 57
レジストリ キー 98
CELog レジストリ パラメータ 172
CELogFlush ツール 174
Clientside.exe 開始パラメータ 191
Console キー 102
DependXX エントリ 99
HKEY_LOCAL_MACHINE\Drivers\Active 273, 310
HKEY_LOCAL_MACHINE\Drivers\BuiltIn 272, 310
HKEY_LOCAL_MACHINE\INIT 98
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Co
ntrolPowerInterfaces 297
LaunchXX エントリ 99
PCI 関連 312
Svcstart サンプル サービス 101
UserProcGroup レジストリ エントリ 281
イベント ログ記録ゾーン 172
コマンド プロセッサ シェル 102
サブプロジェクト 17
スタートアップ パラメータ 99
デバイス クラスおよび 142
デバイス ドライバ 274
デバッグ領域 164
フラグ レジストリ値 276
ペガソス レジストリ キー 165
メモリ関連 312
ユーザー モード ドライバ ホスト プロセス
(Udevice.exe) 281
レジストリ設定

ろ

ローカル エリア ネットワーク (LAN) 31
ロケール 7

わ

ワークステーション サーバー アプリケーション
(CETest.exe) 188
割り込み 284
OAL の同期機能 284
割り込みサービス スレッド (IST) 227, 285
割り込みサービス ルーチン (ISR) 227, 285
IST および IST 間の通信 291
WaitForMultipleObjects 関数 288
アーキテクチャ 284
静的 288
デバイス ドライバ 284
ブレークポイント 183
カーネル配列 289
割り込み待機時間測定 227
割り込み待機時間測定 (ILTiming) ツール 227
割り込み遅延タイミング 90, 95
割り込みハンドラ

著者について

Nicolas Besson

Nicolas Besson は Windows Embedded CE テクノロジの 7 年以上の徹底した技術経験があります。彼は現在、Windows Embedded CE　技術に焦点を当て、世界規模の存在感を持つ重要な Microsoft Gold Embedded Partner である Adeneo でソフトウェア開発とプロジェクト管理を専門にしています。Nicolas は最近 2 年間マイクロソフト eMVP となっています。彼は世界中の企業と人々にトレーニングを提供することによってその知識を共有しています。Windows Embedded CE テクノロジに対する彼の熱意については、http://nicolasbesson.blogspot.com の彼のブログをご覧ください。

Ray Marcilla

Ray Marcilla は Washington の Bellevue にある Adeneo のアメリカ支社に勤務する Embedded ソフトウェアの開発者です。Ray はネイティブとマネージ コードでのアプリケーション開発、また CE ドライバ開発の意味深い経験を持っています。彼はまた CE 関連の技術プレゼンテーションやセミナにも参加しています。Ray は数々の ARM や x86 開発プラットフォームのための興味深いプロジェクトで働いてきました。余暇の時間は外国語を学んで過ごします。現在日本語が流暢で、いくらかの韓国語やフランス語も話すことができます。

Rajesh Kakde

Rajesh は 2001 年から Windows Embedded CE に関わっています。彼は世界のいろいろな場所で Consumer Electronics & Industrial Real-time Devices を含むさまざまな産業部門で働いてきました。彼は BSP とドライバ開発、アプリケーション開発に広範囲の経験があります。

現在、彼はシニア Windows Embedded コンサルタントとして Adeneo Corp. チームに所属しており、BSP やドライバ、数々の OEM プロジェクトの管理において高度の技術的専門知識を提供しています。彼はまた Adeno と Windows Embedded CE のセミナと訓練も配布しており、この技術についての意見を交換し熱意を分かち合っています。