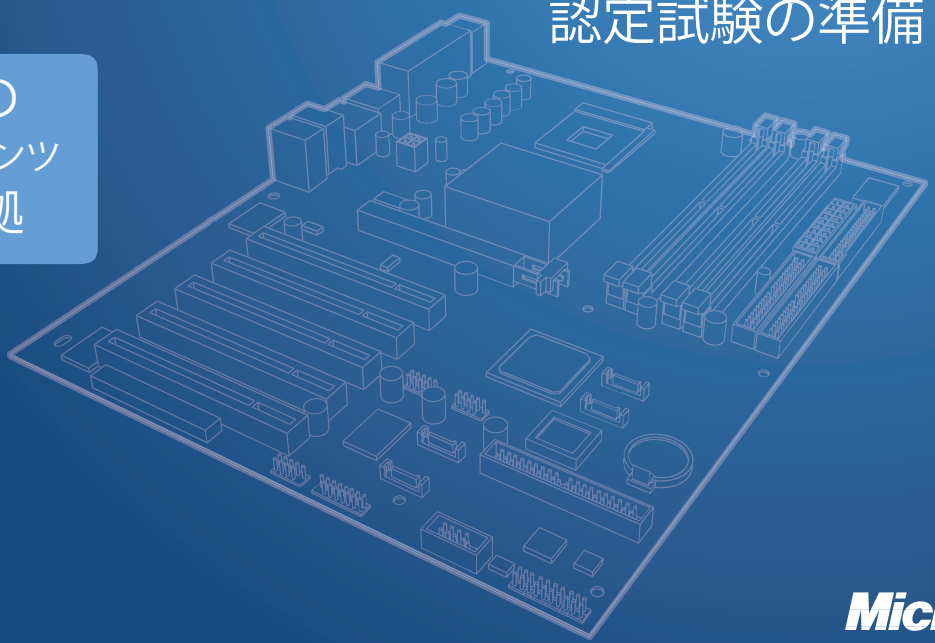
MCTS
Exam 70-571

Windows® Embedded CE 6.0

準備キット

認定試験の準備

最新の
R2 コンテンツ
に準拠



非売品

Microsoft®

出版元

Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

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

本体番号 3043-GA1

Body Part No. 098-109627

目次一覧

第 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。

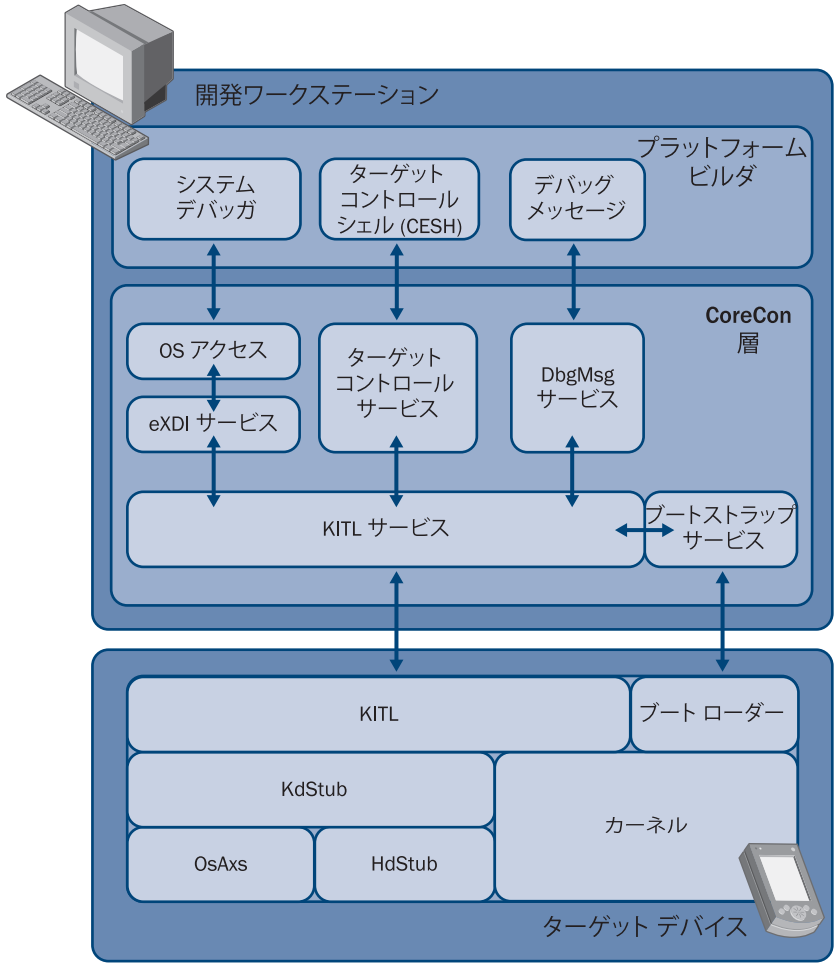


図 4-1　CE デバッグおよびターゲット コントロール アーキテクチャ

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。

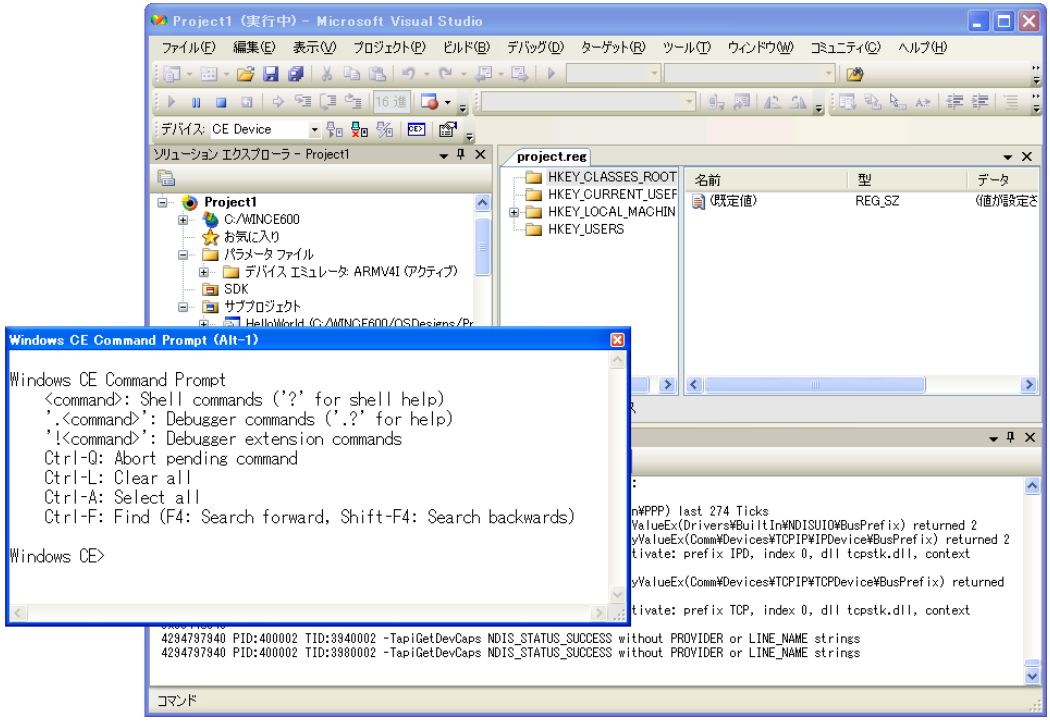


図 4-3　ターゲット コントロール シェル

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。

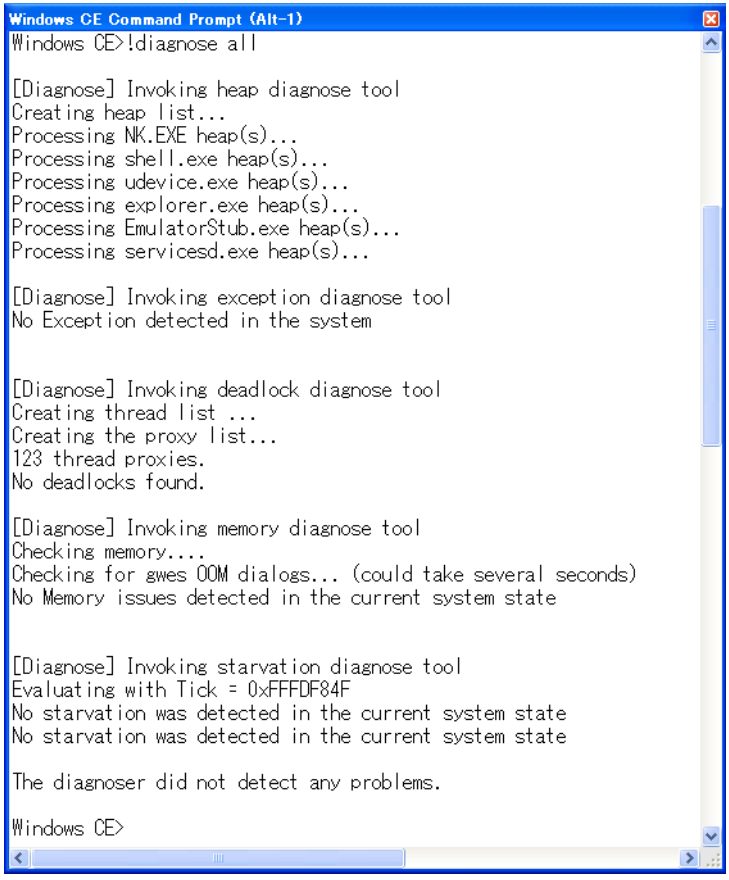


図 4-4　CEDebugX でランタイム イメージを診断

!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。

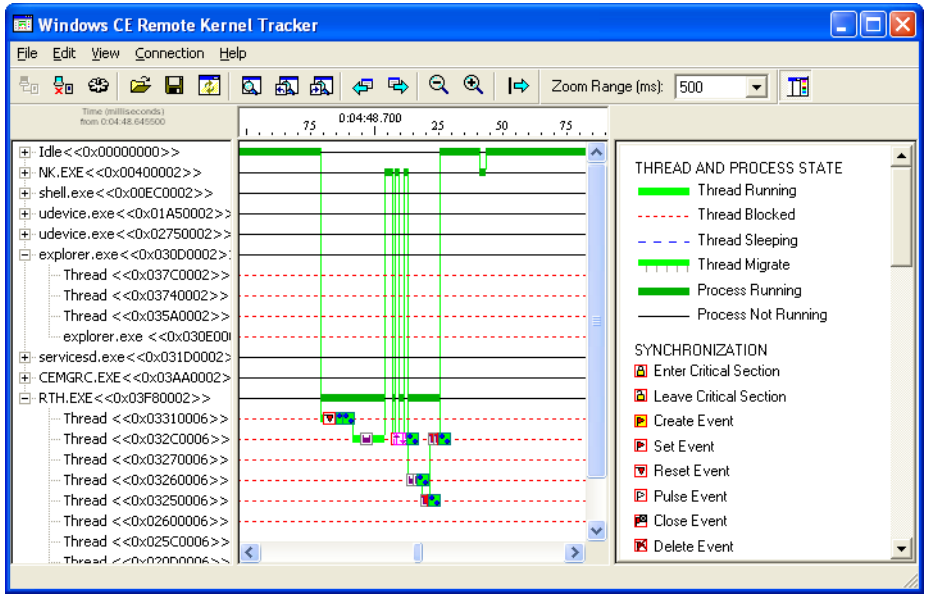


図 4-5　Kernel Tracker のスレッド情報

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。

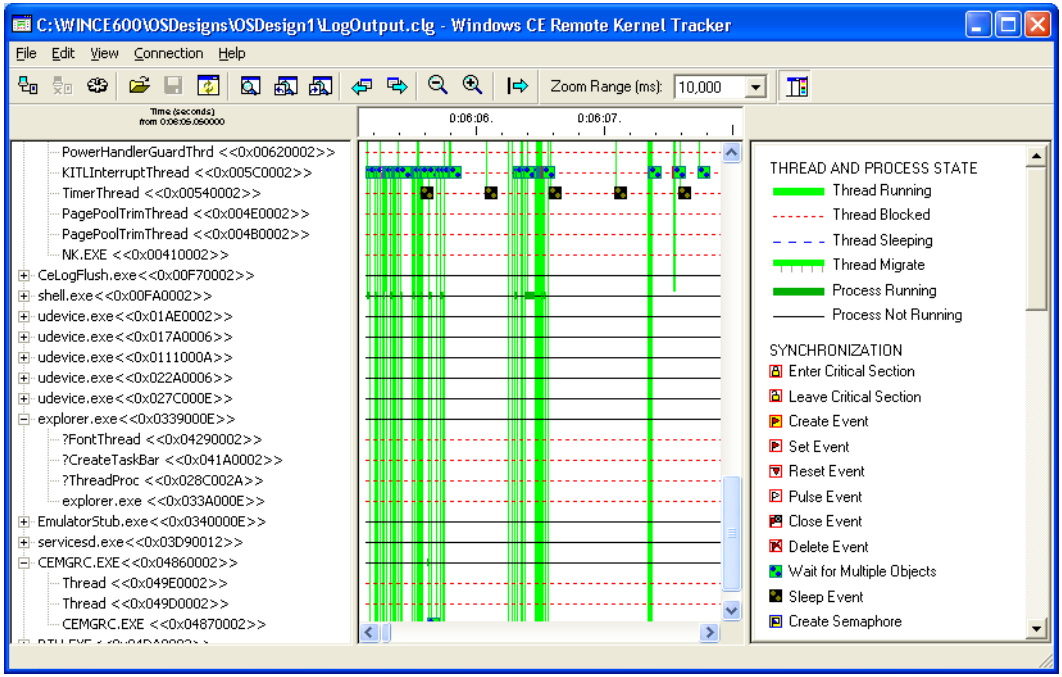


図 4-6　readlog -fixthreads によって準備され、Remote Kernel Tracker で開かれた CELog データファイル

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。

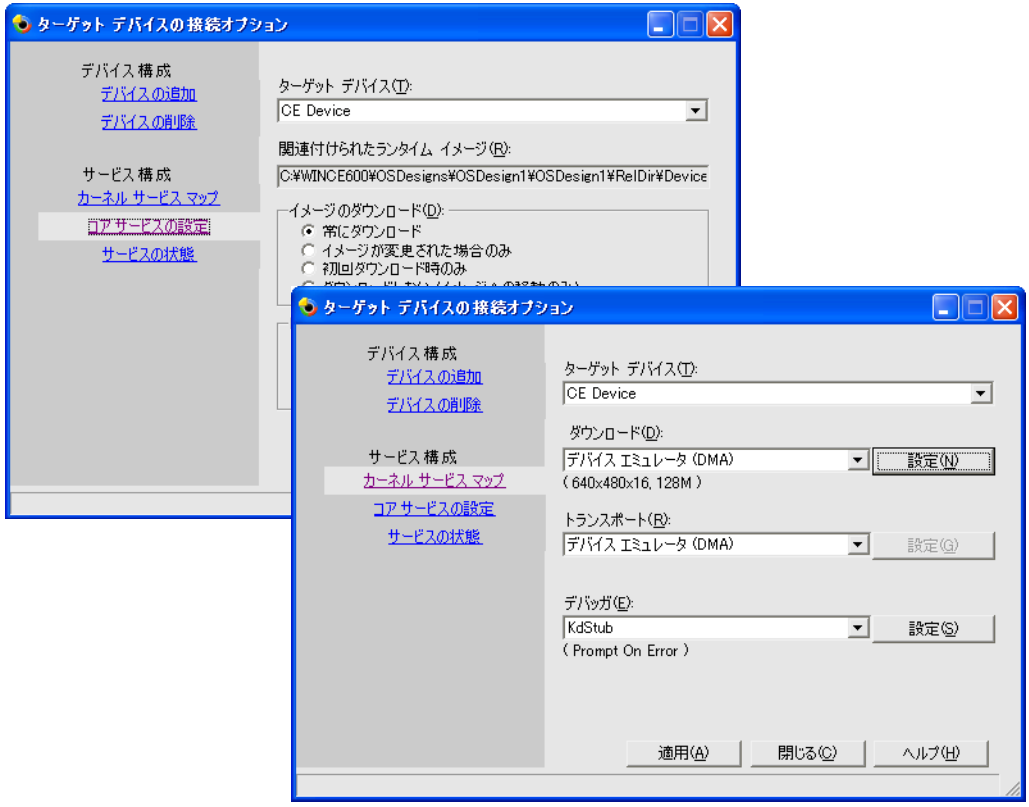


図 4-7 KITL 通信インターフェイスの構成

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。

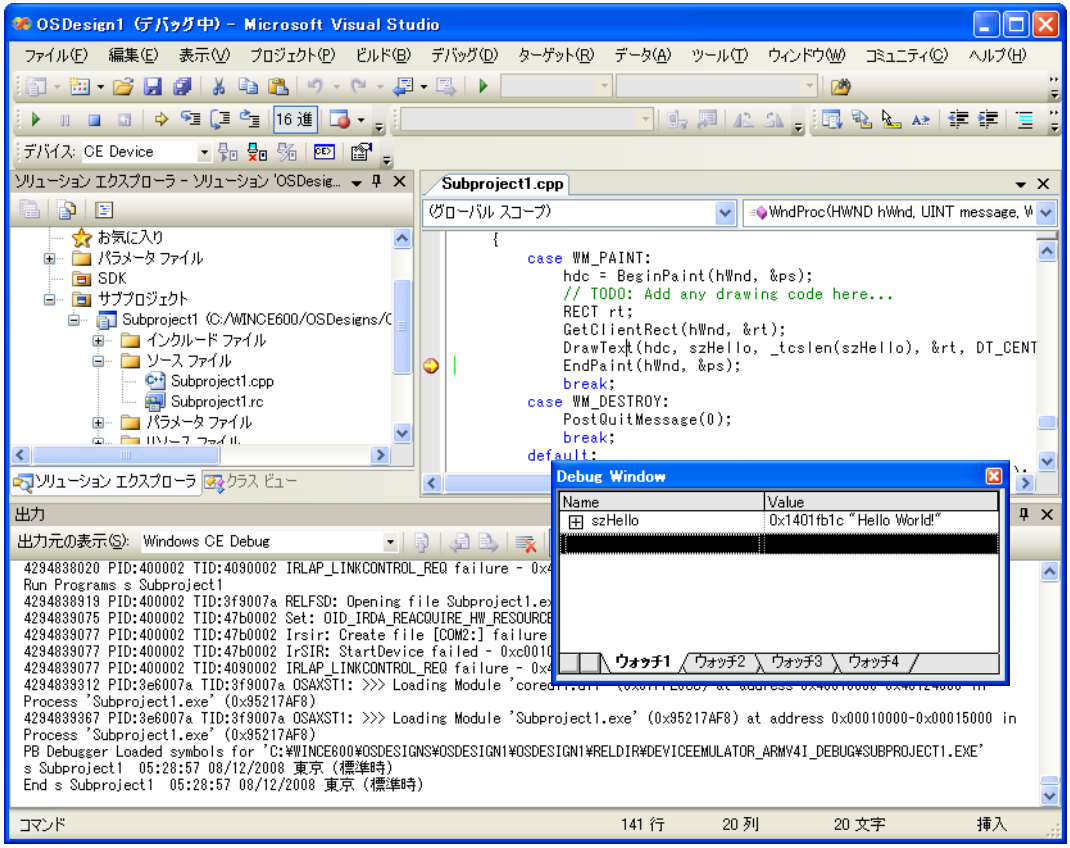


図 4-8　Hello World アプリケーションのデバッグ

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。

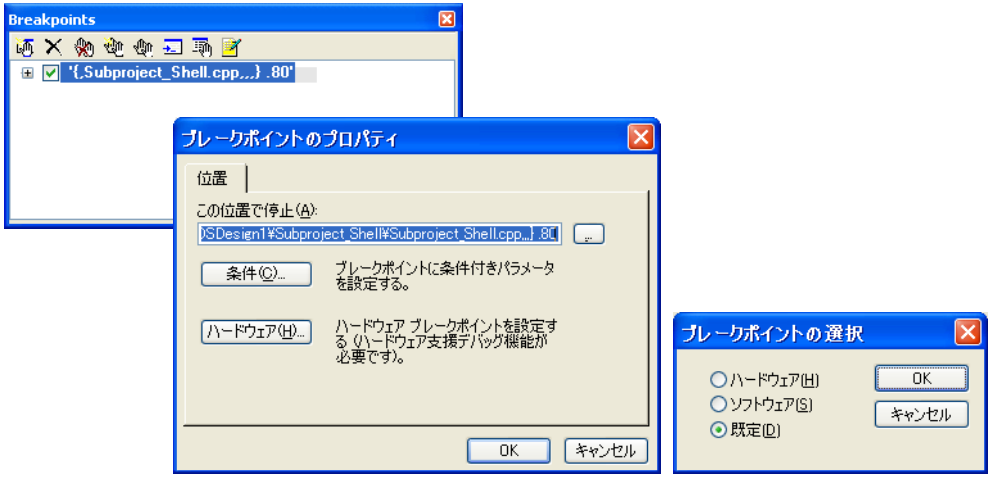


図 4-9　ハードウェア ブレークポイントを設定する

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。

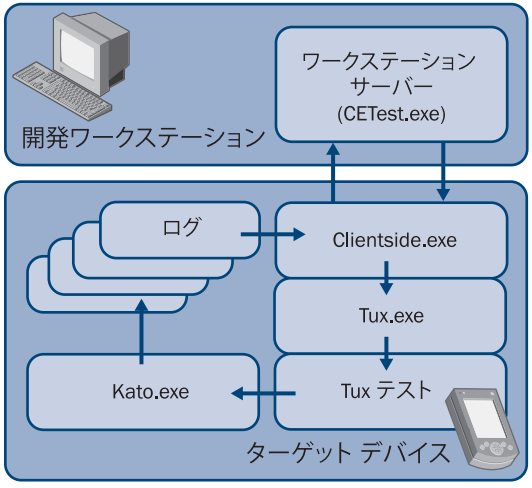


図 4-10　CETK クライアント / サーバー アーキテクチャ

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。

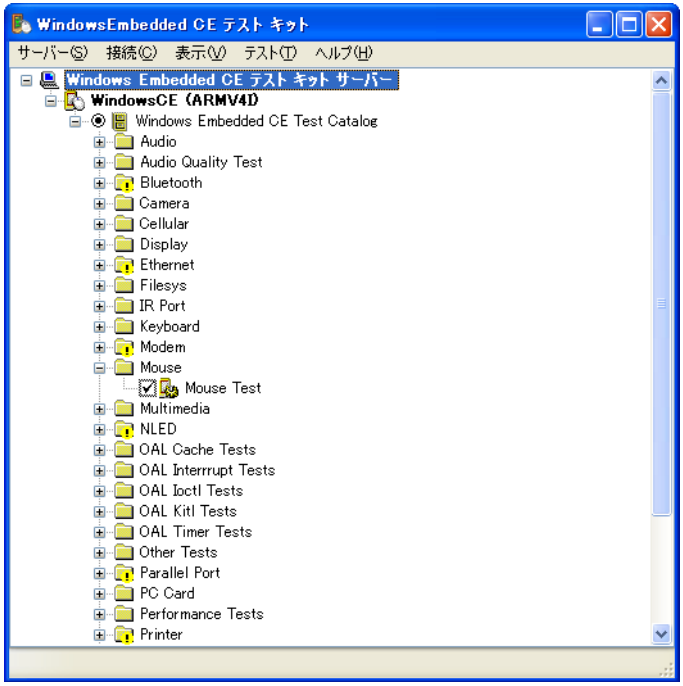


図 4-11　CETK アプリケーションのグラフィカル ユーザー インターフェイス

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。

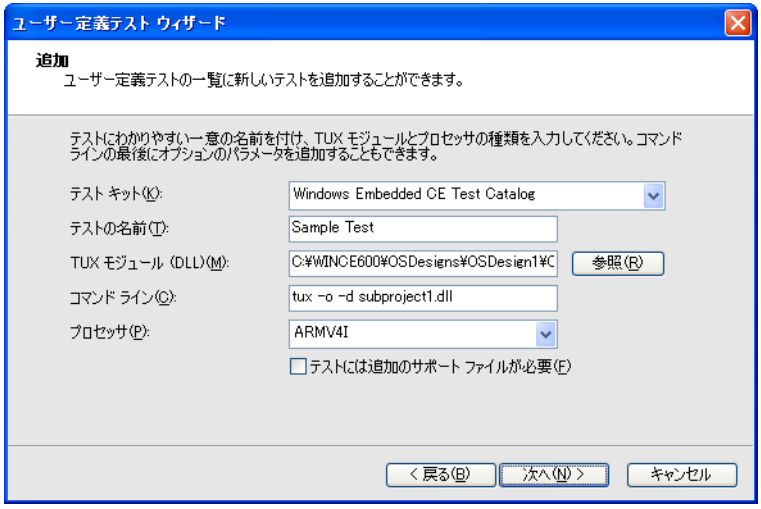


図 4-12　[ユーザー定義テスト ウィザード] でカスタム テストを構成する

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。

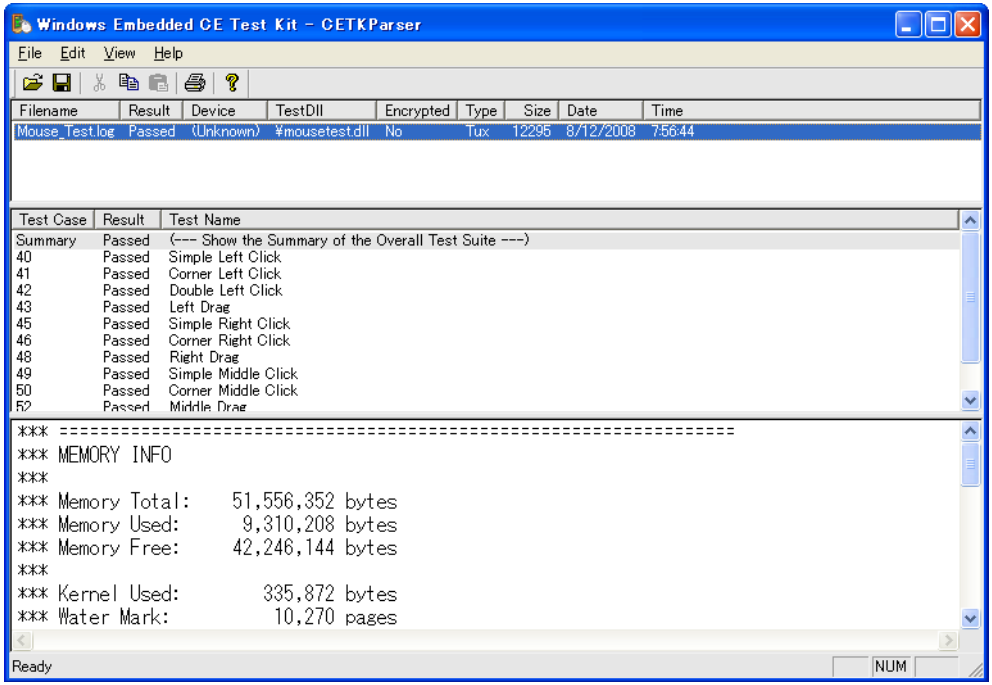


図 4-13 CETK テスト結果を分析する

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。

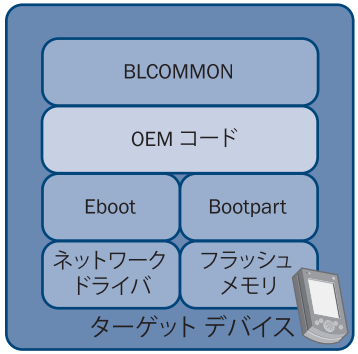


図 4-14　Windows Embedded CE ブート ローダー アーキテクチャ

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。

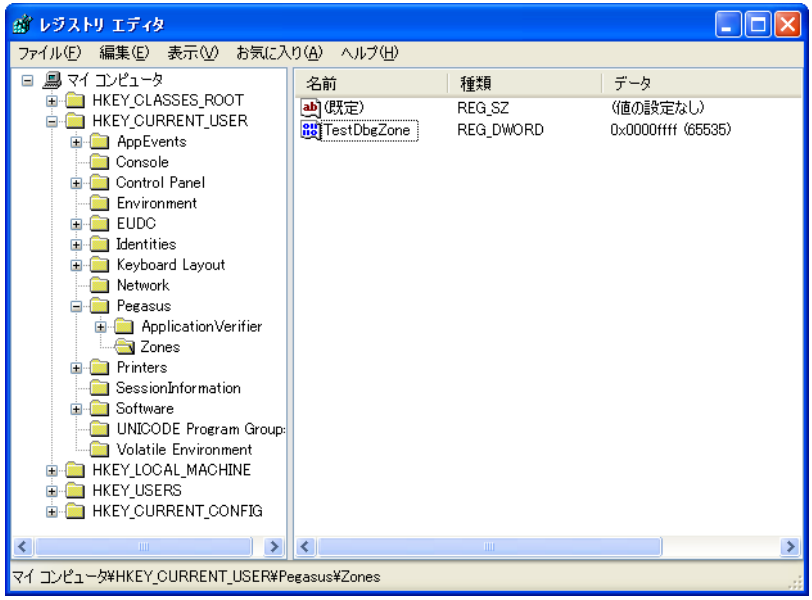


図 4-15 HKEY_CURRENT_USER\Pegasus\Zones: "TestDbgZone"=dword:FFFF

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。

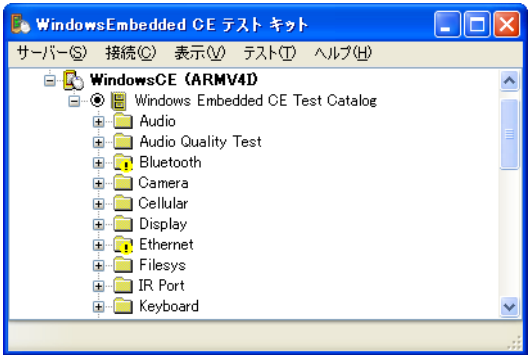


図 4-16　[Windows Embedded CE テスト キット] ウィンドウのデバイス カテゴリ

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。