JUNGO

WinDriver

ユーザーズ ガイド

エクセルソフト株式会社

COPYRIGHT

Jungo Ltd.

POB 8493 Netanya Zip - 42504 Israel
Phone (USA) 1-877-514-0537 (WorldWide) +972-9-8859365
Fax (USA) 1-877-514-0538 (WorldWide) +972-9-8859366

ご注意

- このソフトウェアの著作権はイスラエル国 Jungo Ltd. 社にあります。
- このマニュアルに記載されている事項は、予告なしに変更されることがあります。
- このソフトウェアおよびマニュアルは、本製品のソフトウェア ライセンス契約に基づき、登録者の管理下でのみ使用することができます。
- このソフトウェアの仕様は予告なしに変更することがあります。
- このマニュアルの一部または全部を、エクセルソフト株式会社の文書による承諾なく、無断で複写、複製、転載、文書化することを禁じます。

エクセルソフト株式会社

〒108-0014 東京都港区芝 5-1-9 ブゼンヤビル 4F
TEL 03-5440-7875　FAX 03-5440-7876
E-MAIL: xlsoftkk@xlsoft.com
Home Page: http://www.xlsoft.com/

Rev. 9.0 – 6/2007

目次

図表

第 1 章
WinDriver の概要

この章では、WinDriver の使い方を紹介し、ドライバ作成の基本的なステップを学習します。

1.1 はじめに

WinDriver はデバイスドライバを短期間に作成することを目的に設計された、開発ツールキットです。WinDriver は、自動的にハードウェアを検出し、アプリケーションからハードウェアにアクセスするドライバを生成するウィザードおよびコード生成機能を持っています。WinDriver を使用して開発されたドライバは、サポートされているすべてのオペレーティング システムでソース コード互換になります。また、ドライバは Windows / 98 / Me / 2000 / XP / Server 2003 / Vista ではバイナリ互換になります。バス アーキテクチャのサポートは、PCI / PCMCIA / CardBus / ISA / EISA / CompactPCI / PCI Express (PCMCIA は Windows 2000 / XP / Server 2003 / Vista でのみサポートされています) および USB です。WinDriver はハイパフォーマンスなドライバ作成のソリューションを提供します。

WinDriver を使用すれば、デバイスドライバの開発に数ヶ月要していたものが、数時間で簡単に行えます。このマニュアルは、上級者ユーザー向けの機能を多く紹介しています。しかし、多くの開発者は、この章を読み、DriverWizard の章と別冊 PDF の関数リファレンスのを参照すれば、ドライバの記述に成功できるでしょう。

WinDriver は、USB と PCI ブリッジをサポートします。また、**PLX**、**Altera**、**AMCC**、**QuickLogic**、**Xilinx**、**Cypress**、**Microchip**、**Philips**、**Agere**、**Texas Instruments**、**Silicon Laboratories**、および **National Semiconductors** に関してはより詳細なサポートを行っています。各チップセットに関する詳細は第 8 章 を参照してください。第 11 章 では WinDriver の Kernel PlugIn 機能を使用して、ドライバ コードを最適化する方法を説明しています。ここで WinDriver の Kernel PlugIn 機能が詳しく説明されています。この機能を使用すると、開発者はすべてのコードをユーザー モードで開発し、後でパフォーマンスに関わる部分をカーネル モードに移動できます。Kernel PlugIn の概要は第 11 章 および第 12 章 を参照してください。

WinDriver およびその他の開発ツールに関する最新情報を入手するには、エクセルソフト(株) のホームページ (http://www.xlsoft.com/) および開発元の Jungo 社のホームページ (http://www.jungo.com/) を定期的に参照することを推奨します。

1.2 背景

1.2.1 チャレンジ

保護されたオペレーティング システム (Windows、Linux および Solaris) では、通常開発が行われるアプリケーション レベル (ユーザー モード) から直接ハードウェアにアクセスできません。ハードウェアへのアクセスは、オペレーティング システムが「デバイスドライバ」と呼ばれるソフトウェア モジュールを使ってアクセスする必要があります (カーネル モード または Ring 0)。アプリケーション レベルからカスタム ハードウェア デバイスにアクセスするには、プログラマは次の事柄を行う必要があります:

1. オペレーティング システムの内部情報を学習する。
2. デバイスドライバの記述方法を習得する。
3. カーネル モードでの開発、デバッグに使用するツール (DDK、ETK、DDI、DKI など) を習得する。
4. ハードウェアの基本的な入出力を行うカーネル モードのデバイスドライバを記述する。
5. カーネル モードで記述したデバイスドライバでハードウェアにアクセスする、ユーザー モードでアプリケーションを記述する。
6. コードを実行するオペレーティング システムに対して、それぞれステップ 1 から 4 を繰り返す。

1.2.2 WinDriver の特長

容易な開発: WinDriver は、短時間で **PCI / PCMCIA / CardBus / ISA / EISA / CompactPCI / PCI Express** および **USB** ベースのデバイスドライバを開発できるように設計された、デバイスドライバ開発用ツールキットです。WinDriver を利用すると MSDEV、Visual C/C++、MSDEV .NET、Borland C++ Builder、Borland Delphi、Visual Basic 6.0、MS eMbedded Visual C++、MS Platform Builder C++、GCC などの 32 ビット コンパイラを使って「ユーザー モード」でドライバを作成できます。WinDriver を使用することにより、オペレーティング システムの内部、カーネル プログラミングなどの知識を必要とせずにデバイスドライバを作成できます。

クロス プラットフォーム: WinDriver で作成されたドライバは **Windows 98 / Me / 2000 / XP / Server 2003 / Vista、Windows CE.NET、Windows Embedded CE v6.00、Windows Mobile 5.0 / 6.0、Linux および Solaris** で動作します。そのため、一度コードを記述すれば他のプラットフォームでも動作します。

ユーザー フレンドリーなウィザード: DriverWizard は、対象のハードウェアのデバイスドライバを開発する前に、デバイスのリソースを表示または定義したり、ハードウェアとの通信をテストするためのグラフィカルな診断プログラムです。ハードウェアのメモリ範囲、レジスタ、割り込みなどが確認されます。デバイスが完全に動作していることを確認した後、DriverWizard はハードウェアのすべてのリソースにアクセス可能なデバイスドライバの雛形を作成します。

カーネル モードのパフォーマンス: WinDriver の API はパフォーマンス向上のため、最適化されています。ユーザー モードでは達成できないパフォーマンスの向上を図る場合、WinDriver の「WinDriver Kernel PlugIn」を利用します。WinDriver Kernel PlugIn を利用するには、まず通常の WinDriver ツールを利用してドライバをユーザー モードで作成します。次にパフォーマンスに大きく関わるコード (割り込みハンドラ、I/O にマップされたメモリ領域へのアクセスなど) をWinDriver の Kernel PlugIn に移動します。Kernel PlugIn に移動したモジュールはカーネル モードで実行するので、実行までのオーバーヘッドがなくなります。この機能を利用することにより、開発が容易なユーザー モードで開発を行い、必要な箇所のパフォーマンスを向上することができます。速度を向上させる箇所だけをカーネル モードに移動できるため、開発期間を短縮できるほか、作成するデバイスドライバのパフォーマンスを犠牲にすることもありません。この機能に関する詳細は第 10 章 を参照してください。

このユニークな機能により、開発者はカーネルの動作を習得する必要もなく OS カーネル内でユーザー モード コードを実行できます。Windows CE の場合、ユーザー モードとカーネル モードの境界がないため、Kernel PlugIn を使用する必要はありません。そのため、ユーザー モードから最適なパフォーマンスを達成できます。セクション [9.2.3] では、Windows CE における割り込み処理率を改良する方法を説明します。

1.3 WinDriver の処理速度

PCI ドライバの場合、WinDriver Kernel PlugIn は、カスタム カーネル ドライバと同程度の処理速度を期待できます。その処理速度は、オペレーティング システムとハードウェアの制限によって異なります。大雑把に見積もって、Kernel PlugIn を使って毎秒約 100,000 回の割り込み処理ができます。USB ドライバの場合、カスタム カーネル ドライバと同程度の転送速度を期待できます。USB 1.1 または USB 2.0 の最大限の性能を引き出します。

1.4 最後に

WinDriver を使用して、カスタム ハードウェアにアクセスするアプリケーションを作成するために必要な手順をまとめます:

- DriverWizard を実行し、ハードウェアとそのリソースを検出します。
- DriverWizard を使って、デバイス ドライバのコードを自動生成します。または、WinDriver のサンプルの 1 つをアプリケーションの基礎として使用します。各 PCI チップセットへの拡張サポートに関する詳細および各 USB チップセットへの拡張サポートに関する詳細は第 8 章 を参照してください。
- アプリケーションに実装する機能を適用するために、生成された関数またはサンプルの関数を使用して、ユーザー モード アプリケーションを必要に応じて修正してください。

これで、すべての対応するプラットフォームから新しいハードウェアにアクセスするアプリケーションを作成できます。(コードは Windows 98 / Me / 2000 / XP / Server 2003 / Vista プラットフォームでバイナリ互換性があります。そのため、これらの OS 間でドライバを移植する場合は再ビルドする必要はありません。)

1.5 WinDriver の利点

- ユーザー モードで容易にドライバを開発。
- Kernel PlugIn で高性能なドライバを開発。
- ユーザー フレンドリーな DriverWizard はコードを記述する前に、ハードウェアの診断を行い、ドライバ コードの大部分を DriverWizard が自動的に生成します。
- DriverWizard で C、C#、Delphi (Pascal) または Visual Basic のドライバ コードを自動的に生成します。
- PCI / PCMCIA / CardBus / ISA / EISA / CompactPCI / PCI Express および USB デバイスを製造元に関わらずサポートします。
- PLX / Altera / AMCC / Xilinx などの PCI チップをサポートします。そのため、開発者は PCI チップの詳細を特に知る必要はありません。
- USB 実装の詳細が分かりづらい、Cypress、Microchip、Philips、Texas Instruments、Agere、Silicon Laboratories などの USB コントローラを拡張サポートします。
- 作成されるアプリケーションは Windows 98 / Me / 2000 / XP / Server 2003 / Vista でバイナリ互換です。

- 作成されるアプリケーションは Windows 98 / Me / 2000 / XP / Server 2003 / Vista、Windows CE.NET、Windows Embedded CE v6.00、Windows Mobile 5.0 / 6.0、Linux および Solaris でソース コード互換です。
- MSDEV、Visual C/C++、MSDEV .NET、Borland C++ Builder、Borland Delphi、Visual Basic 6.0、MS eMbedded Visual C++、MS Platform Builder C++、GCC などのコンパイラを含む一般的な開発環境で使用可能です。
- DDK、ETK、DDI などのシステム レベル プログラムに関する知識を必要としません。
- I/O、DMA、割り込み処理、メモリ マップされた カードへのアクセスをサポートしています。
- マルチ CPU、マルチ PCI バス プラットフォーム (PCI / PCMCIA / CardBus / ISA / EISA / CompactPCI / PCI Express) をサポートします。
- 64 ビット PCI データ転送をサポートします。
- ダイナミック ドライバ ローダーを含んでいます。
- 詳細なマニュアルとヘルプ ファイルが用意されています。
- C、C#、Delphi、Visual Basic 6.0 の詳細なサンプルが用意されています。
- WHQL 認証ドライバ (Windows)。
- 2 ヶ月間の無料テクニカルサポート (インストール、ライセンス、配布に関する質問)。
- 作成したドライバを無料で使用、配布できます。

1.6 WinDriver のアーキテクチャ

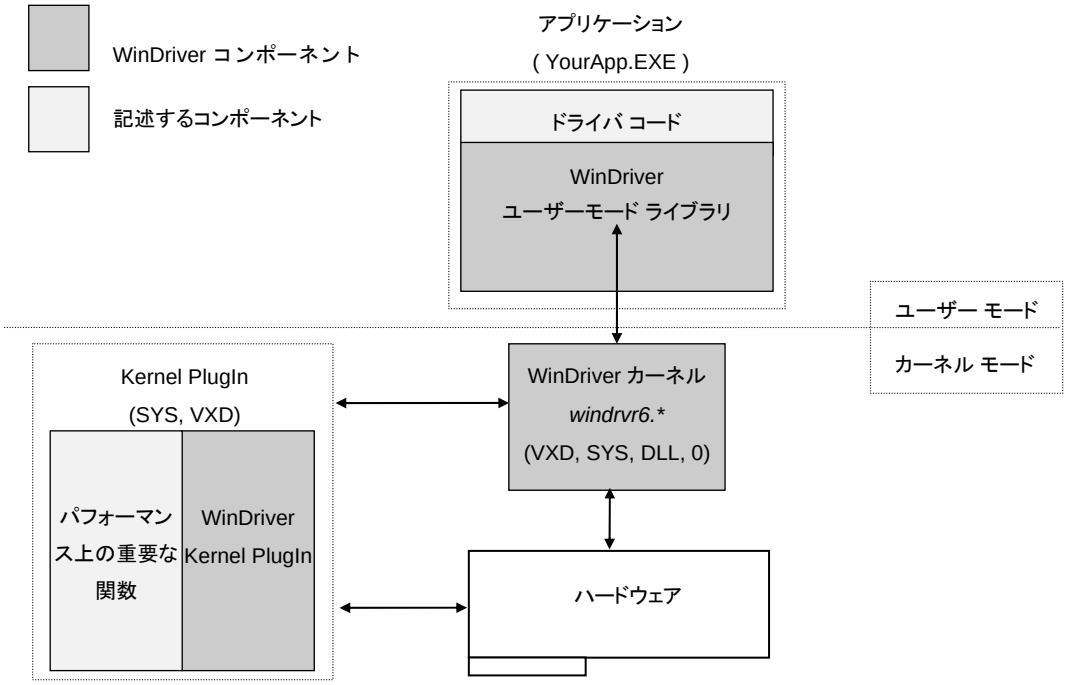


図 1.1: WinDriver アーキテクチャ

ハードウェアにアクセスする場合、アプリケーションは WinDriver ユーザー モード ライブラリ (windrvr.h) から WinDriver 関数を呼び出します。ユーザー モード ライブラリがハードウェアにネイティブ コールでアクセスする WinDriver カーネルを呼び出します。

WinDriver は、ユーザー モードで実行されてもパフォーマンスにあまり影響しないように設計されています。しかし、ハードウェアによってはユーザー モードでは得られないほど高いパフォーマンスを必要とする場合があります。このような場合、ユーザー モードで開発したコードからパフォーマンスが必要なモジュール (割り込みハンドラ等) のコードを変更せずに WinDriver の Kernel PlugIn に移動します。これにより、WinDriver カーネルがカーネル モードでこのモジュールを呼び出し、パフォーマンスを向上させます。そのため、ユーザー モードで容易にドライバの開発、デバッグを行い、必要な部分のパフォーマンスを向上できます。Kernel PlugIn に関する詳細は第 11 章 を参照してください。 Windows CE の場合、ユーザー モードとカーネル モードの境界がないため、Kernel PlugIn を使用する必要はありません。そのため、ユーザー モードから簡単に最適なパフォーマンスを達成できます。

1.7 WinDriver がサポートするプラットフォーム

WinDriver は以下のオペレーティング システムをサポートします:

- Windows 98 / Me / 2000 / XP / Server 2003 / Vista – これ以降、"**Windows**" と呼びます。
- Windows CE 4.x – 5.x (Windows CE.NET)、Windows Embedded CE v6.00、Windows Mobile 5.0 / 6.0 – これ以降、"**Windows CE**" と呼びます。
- Linux
- Solaris

Windows NT と VxWorks は以前のバージョンでサポートしてます。

同じソース コードがサポートするすべてのプラットフォーム上で実行できます。また、作成した実行ファイルは、Windows 98 / Me / 2000 / XP / Server 2003 / Vista で動作します。これらの中の 1 つのオペレーティング システム用に作成したドライバであっても、WinDriver を使用することにより、コードの変更を行わずに他のオペレーティング システムに移行できます。

1.8 評価版 (Evaluation Version) の制限

すべての評価版は、フル機能を装備しています。制限される機能はありません。以下に登録版と評価版の違いを記述します。

- 毎回 WinDriver を起動すると評価版であることを示すメッセージが表示されます。
- DriverWizard を使用しているとき、評価版が実行していることを知らせるダイアログ ボックスが、ハードウェアと相互作用するたびに表示されます。
- Linux、Solaris および CE 版では、60 分間動作した後、停止します。再度評価するには、再ロードする必要があります。
- Windows の評価版はインストール後、30 日間使用できます。
- 詳細は、別冊 PDFの「評価版 (Evaluation Version) の制限」の章を参照してください。

1.9 WinDriver を使用してドライバを開発するには

1.9.1 Windows、Linux および Solaris

1. DriverWizard を起動し、デバイスを診断します。詳細は第 5 章「DriverWizard」を参照してください。
2. 雛型となるコードを生成するか、または WinDriver のサンプルをドライバ アプリケーションの雛型とします。各チップセット特有の拡張サポートに関する詳細は第 8 章 を参照してください。
3. DriverWizard が生成するコードを修正してアプリケーションに必要な機能を作成してください。
4. ユーザー モードでドライバのテストやデバッグを行います。
5. コードにパフォーマンス的に重要な部分が含まれている場合、第 10 章「パフォーマンスの向上」を参考にパフォーマンスを向上することもできます。

注意: DriverWizard で作成したコードは、検出または定義したリソースへの read および write を行う関数を持つ診断プログラムで、対象のカードの割り込み、Listen、USB パイプのアクセスなどが行えます。

1.9.2 Windows CE

1. Windows ホスト マシンにあなたのハードウェアを装着します。
2. DriverWizard でハードウェアを診断します。
3. ドライバ コードの雛形を DriverWizard で生成します。
4. ドライバ コードの雛形を DriverWizard で生成します。
5. ハードウェアの仕様にあわせて、Visual C++ でこのコードを修正します。Platform Builder を使用している場合、ワークスペースへ 生成された *.pbp を挿入します。
6. ホスト マシン上で実行する CE エミュレーションでコードとハードウェアをテストおよびデバッグします。

ヒント: Windows のホスト マシンにハードウェアを装着できない場合、DriverWizard を使用してすべてのリソースを手動で入力する必要があります。DriverWizard でコードを生成し、ハードウェアをシリアル接続でテストします。生成したコードが正しく動作することを確認したら、ハードウェアの仕様にあわせて修正します。また、サンプルのファイルを雛形として使用することもできます。

1.10 WinDriver ツールキットの内容

- WinDriver CD
  - ユーティリティ
  - サポートする API チップセット
  - サンプル ファイル
- 印刷マニュアル

- 2 ヶ月間のインストール、ライセンスおよび配布に関する質問 (FAX、電子メール)
- WinDriver モジュール

1.10.1 WinDriver のモジュール

- WinDriver (**WinDriver/include/**): 汎用ハードウェア アクセス ツールキット。
  - **windrvr.h:** WinDriver API の宣言および定義します。
  - **wdu_lib.h:** ラッパー USB API を提供する WinDriver USB (WDU) ライブラリの宣言および定義します。
  - **wdc_lib.h and wdc_defs.h**: PCI/PCMCIA/CardBus/ISA/ EISA/CompactPCI/PCI Express デバイスへアクセスするラッパー API を提供する WinDriver Card (WDC) ライブラリの宣言および定義します。詳細は別冊 PDF を参照してください。
  - **windrvr_int_thread.h:** 割り込み処理を簡略化するラッパー 関数の定義をしています。
  - **windrvr_events.h**: イベント処理および PnP 通知を実装する関数を含みます。
  - **utils.h:** 一般的なユーティリティ関数を宣言します。
  - **status_strings.h:** WinDriver のステータス コードをエラー メッセージに変換する API を宣言します。
- DriverWizard ([スタート] メニュー - [プログラム] - [WinDriver] - [Wizard] - [DriverWizard] からアクセスできます): ハードウェアを診断し、ドライバを簡単にコード化するグラフィカルなツール (第 5 章「DriverWizard」を参照してください)。
- Graphical Debugger ([スタート] メニュー - [プログラム] - [WinDriver] - [util] - [wddebug_gui] からアクセスできます): ドライバの実行中にデバッグ情報を収集するグラフィカルなデバッグ ツール。WinDriver は、Windows CE または VxWorks などの GUI サポートが無いプラットフォームで使用可能なプログラム (**WinDriver/util/wddebug**) のコンソール版を含んでいます。Debug Monitor に関する詳細はセクション [7.2] を参照してください。
- WinDriver 配布用パッケージ (**WinDriver/redist** – Windows、Windows CE、Linux および Solaris; **WinDriver/redist_win98_compat** – Windows 98 / Me / 2000 / XP / Server 2003 / Vista): ユーザーに配布するファイル。
- WinDriver Kernel PlugIn: Kernel PlugIn を作成するためのファイルとサンプル。詳細は第 11 章 を参照してください。
- 本書: さまざまな形式の WinDriver マニュアル。これらは **WinDriver/docs/** ディレクトリに保存されています。

1.10.2 ユーティリティ

- **pci_dump.exe** (**WinDriver/util/pci_dump.exe**): インストールされている PCI カードの PCI 設定レジスタのダンプを取得するためのユーティリティ。

- **pci_diag.exe** (**WinDriver/util/pci_diag.exe**): PCI 設定レジスタの入出力、PCI I/O 領域とメモリ領域へのアクセス、および PCI 割り込み処理を行うためのユーティリティ。
- **pci_scan.exe** (**WinDriver/util/pci_scan.exe**): インストールされている PCI カードのリストおよび各カードに割り当てられたリソースを取得するためのユーティリティ。
- **pcmcia_diag.exe** (**WinDriver/util/pcmcia_diag.exe**): PCMCIA 属性空間の入出力、PCMCIA I/O 領域とメモリ領域へのアクセス、PCMCIA 割り込み処理を行うためのユーティリティ。
- pcmcia_scan.exe (WinDriver/util/pcmcia_scan.exe): インストールされている PCMCIA カードのリストおよび各カードに割り当てられたリソースを取得するためのユーティリティ。
- **usb_diag.exe** (**WinDriver/util/usb_diag.exe**): インストールされている USB デバイスのリスト、各デバイスに割り当てられたリソースの取得、USB デバイスのアクセスを行うユーティリティ。

CE バージョンに添付

- **\REDIST\...\X86EMU\WINDRVR_CE_EMU.DLL**: Windows CE の X86 HPC エミュレーションモード 用 Windriver カーネル と通信する DLL。
- **\REDIST\...\X86EMU\WINDRVR_CE_EMU.LIB**: Windows CE の X86 HPC エミュレーションモードでコンパイルした WinDriver アプリケーションをリンクするためのインポート ライブラリ。

1.10.3 特定チップセットのサポート

WinDriver はカスタム ラッパー API と以下のチップセットを含む主要な PCI チップセット用 (第 8 章 を参照) のサンプルコードを提供します。

- PLX 9030、9050、9052、9054、9080、9056、および 9656 – これらは **WinDriver/plx** ディレクトリに保管されています。
- AMCC S5933 - **WinDriver/amcc** に保管されています。
- Altera pci_dev_kit - **WinDriver/altera/pci_dev_kit/** に保管されています。
- Xilinx VirtexII - **WinDriver/xilinx/VirtexII** に保管されています。

WinDriver はカスタム ラッパー API と以下のコントローラを含む主要な USB コントローラ用 (第 8 章 を参照) のサンプルコードを提供します。

- Cypress EZ-USB - **WinDriver/cypress/** に保管されています。
- Microchip PIC18F4550 - **WinDriver/microchip/pic18f4550/** に保管されています。
- Philips PDIUSBD12 - **WinDriver/pdiusbd12/** に保管されています。
- Texas Instruments TUSB3410、TUSB3210、TUSB2136、TUSB5052 - **WinDriver/ti/** に保管されています。
- Agre USS2828 - **WinDriver/agere/** に保管されています。
- Silicon Laboratories C8051F320 USB - **WinDriver/silabs/** に保管されています。

1.10.4 サンプル

特定のチップセット用のサンプルに加え、WinDriver にはデバイスと通信したり、さまざまなタスクを実行する WinDriver API の使用方法のデモンストレーション用のサンプルが含まれています。

- **`WinDriver/samples/`** - C のサンプル。

  このサンプルには、[1.10.2] で紹介したユーティリティのソースコードも含まれています。

- **`WinDriver/csharp.net`** および **`WinDriver/vb.net`** - .NET C# のサンプル (Windows)
- **`WinDriver/delphi/samples/`** - Delphi (Pascal) のサンプル (Windows)
- **`WinDriver/vb/samples/`** - Visual Basic のサンプル (Windows)

1.11 WinDriver で作成したドライバを配布できますか

はい、可能です。WinDriver 開発用ツールキットとして購入されている WinDriver を使用して作成されたデバイスドライバはロイヤリティ フリーでコピーを無制限に配布することができます。詳細については契約同意書 (**`WinDriver/docs/license.pdf`**) を参照してください。

第 2 章
デバイスドライバの理解

この章では、一般的なデバイス ドライバの手引き紹介し、デバイス ドライバの構造的な要素を説明します。

注意: WinDriver の簡単な API を使用するだけで、ドライバやカーネル開発の知識なしで、ハードウェアと通信したり、ユーザー モードでデバイスドライバを作成できます。

2.1 デバイスドライバの概要

デバイスドライバは、端末、ディスク、テープ ドライブ、ビデオ カードおよびネットワーク メディアなどの特定のハードウェア デバイスと OS 間のインターフェイスを提供するソフトウェアの一種です。デバイスドライバは、デバイスへサービスを提供し、ハードウェア引数を設定し、カーネルからデバイスへデータを転送し、カーネルへ戻ってきたデータを渡し、デバイスのエラーを処理したりします。ドライバは、デバイスとプログラム間の翻訳機のような役割をします。各デバイスは、そのドライバのみが理解できるような特別なコマンドのセットを持っています。対照的に、多くのプログラムは、汎用的なコマンドを使用してデバイスにアクセスします。よって、ドライバはプログラムから汎用的なコマンドを受信し、それをデバイスが理解できる特別なコマンドに翻訳します。

2.2 機能によるドライバの分類

機能に応じて、さまざまなドライバの種類が存在します。このセクションでは、最も一般的な 3 つのドライバの種類を簡単に紹介します。

2.2.1 モノリシック ドライバ

モノリシックドライバは、ハードウェア デバイスをサポートするのに必要なすべての機能を持ったデバイスドライバです。モノリシックドライバは 1 つ、または複数のユーザー アプリケーションによりアクセスされ、ハードウェア デバイスを直接制御します。ドライバは IO コントロール コマンド (IOCTL) を通してアプリケーションと通信し、DDK、ETK、DDI/DKI 関数を使用してハードウェアを制御します。

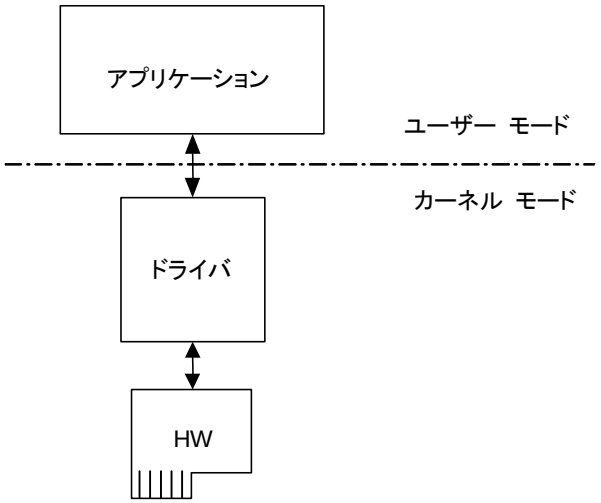


図 2.1: モノリシック ドライバ

モノリシック ドライバは、すべての Windows プラットフォームおよび UNIX プラットフォームを含むオペレーティング システムに存在します。

2.2.2 レイヤード ドライバ

レイヤード ドライバは、IO 要求を他のデバイス ドライバと一緒に処理するデバイス ドライバのスタックの一部です。たとえば、レイヤード ドライバは、ディスクへの呼び出しを横取りし、ディスクへ / から転送されるすべてのデータを暗号 / 復号化するドライバです。このようなドライバは既存のドライバの上位に位置し、暗号化 / 復号化のみを行います。

レイヤード ドライバはフィルタ ドライバとしても知られています。これらは、Windows プラットフォームおよび UNIX プラットフォームを含むすべての OS でサポートされています。

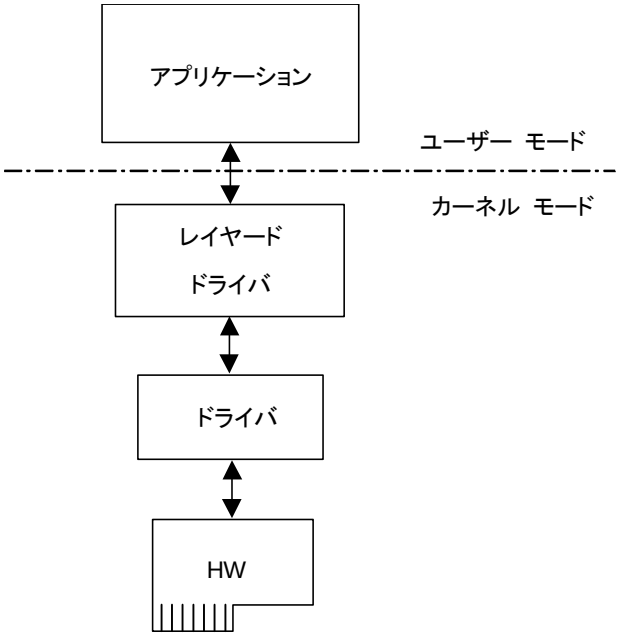


図 2.2: レイヤード ドライバ

2.2.3 ミニポート ドライバ

ミニポートドライバは、ミニポートドライバをサポートするクラスドライバへの add-on です。そのクラス用のドライバが必要とするすべての関数をミニポートドライバによって実装しなくても済むように使用します。クラスドライバは、ミニポートドライバの基本的なクラスの機能を提供します。クラスドライバは、すべての HID デバイスまたはネットワーク デバイスなどの共通的な機能のデバイスのグループをサポートするドライバです。

ミニポートドライバは、ミニクラスドライバまたはミニドライバとも呼ばれ、Windows NT (2000) ファミリ、Windows NT / 2000 / XP / Server 2003 / Vista でサポートされています。

Windows NT / 2000 / XP / Server 2003 / Vista は、その他にもクラスの共通的な機能をハンドルするドライバクラス (ポートと呼ばれる) を提供します。ユーザーに応じて、特定のハードウェアの内部的な動作を行う必要がある機能のみを追加します。

NDIS ミニポートドライバはそれらのクラスの一例です。NDIS ミニポート フレームワークを使用して、NT の通信スタックに接続するネットワークドライバを作成します。よって、そのネットワークドライバは、アプリケーションで使用する共通的な通信の呼び出しにアクセスできます。Windows NT のカーネルは、さまざまな通信スタック用のドライバと一般的な通信カードのコードを提供します。NDIS フレームワークによって、ネットワーク カードの開発者は、このコードをすべて記述する必要はありません。開発を行うネットワーク カードの独自のコードのみを記述します。

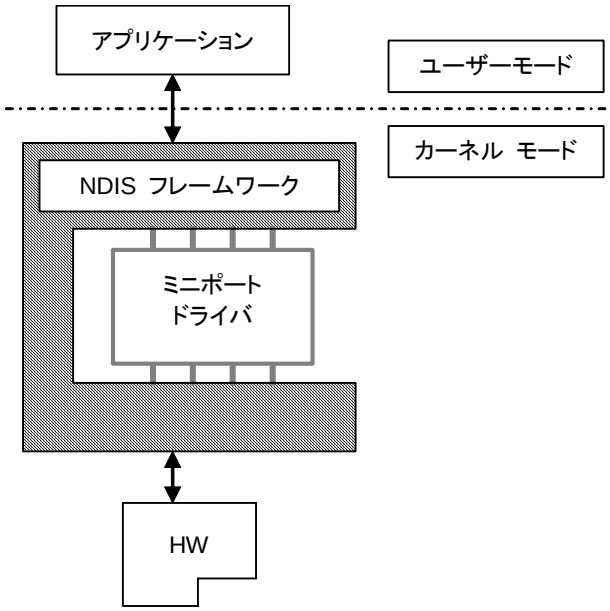


図 2.3: ミニポート ドライバ

2.3 OS によるドライバの分類

2.3.1 WDM ドライバ

WDM (Windows Driver Model) ドライバは、Windows NT および Windows 98 OS ファミリのカーネル モード ドライバです。Windows NT ファミリとは、Windows NT / 2000 / XP / Server 2003 / Vista で、Windows 98 ファミリとは、Windows 98 と Windows Me を指します。WDM は、OS に統合されるコードの一部としてデバイスドライバの動作をチャネリングすることによって、動作します。これらのコードの一部は、DMA および

Plug-and-Play (Pnp) デバイスのエミュレーションを含む、低レベルなバッファ管理を行います。WDM ドライバは、電源管理プロトコルをサポートし、モノリシック ドライバ、レイヤード ドライバおよびミニポート ドライバを持つ PnP ドライバです。

2.3.2 VxD ドライバ

VxD ドライバは、Windows 95 / 98 / Me の Virtual Device Drivers で、ファイル名の終わりが .vxd 拡張子なので VxDs を呼ばれています。VxD ドライバは、典型的なモノリシックです。VxD ドライバは、ハードウェアへの直接アクセスと権限を持った OS の機能を提供します。VxD ドライバをあらゆる種類にスタックまたはレイヤとすることがでますが、ドライバの構造自体は、レイヤ化しません。

2.3.3 デバイス ドライバ

クラシックな Unix ドライバ モデルでは、デバイスは次の 3 つのカテゴリのうちの 1 つに属します: キャラクタ (Char) デバイス、ブロック デバイスおよびネットワーク デバイス。これらのデバイスを実行するドライバは同様にキャラクタ ドライバ、ブロック ドライバまたはネットワーク ドライバとして知られています。Unix では、ドライバはカーネルにリンクしているコード ユニットで、特権を持つ カーネル モードで実行します。一般的に、ドライバ コードはユーザー モード アプリケーションに代わって実行されます。ユーザー モード アプリケーションから Unix ドライバへのアクセスは、ファイル システムを経由して提供されます。つまり、デバイスは開くことが可能な特別なデバイス ファイルとしてアプリケーションから見えます。

Unix デバイス ドライバは、レイヤードまたはモノリシック ドライバのいずれかです。モノリシック ドライバは、1 レイヤのレイヤード ドライバとして知られています。

2.3.4 Linux デバイス ドライバ

Linux デバイス ドライバは、クラシックな Unix デバイス ドライバ モデルが基となっています。さらに、Linux は独自の特長を持っています。

Linux では、ブロック デバイスはキャラクタ デバイスのようにアクセスすることができますが、ユーザーやアプリケーションに対して見えないブロック指向インターフェイスを持っています。

通常、Unix では、デバイス ドライバはカーネルにリンクされ、また、新しいデバイスをインストールした後にシステムを停止させ、再起動します。Linux はモジュールと呼ばれる動的にロードすることができるドライバの概念を持っています。Linux モジュールは、システムをシャットダウンすることなくモジュールを動的にロードしたり削除することができます。すべての Linux ドライバは書き込み可能なため、静的にリンクさせたり、モジュラー フォームに書き込むことができ、これにより動的にロード可能となります。これは、モジュールが検索しているハードウェアが見つからない場合、モジュールはハードウェアを検索して、モジュール自体をアンロードするように記述されることができるので、Linux のメモリの使用を効果的にします。

Unix のデバイス ドライバのように、Linux デバイス ドライバは、レイヤードまたはモノリシック ドライバのいずれかです。

2.3.5 Solaris デバイス ドライバ

Solaris デバイス ドライバも Linux ドライバのようにクラシックな Unix デバイス ドライバ モデルが基となっています。Linux ドライバのように、Solaris ドライバをカーネルに静的にリンクするか、カーネルから動的にロードまたは削除する場合があります。

Unix と Linux デバイス ドライバのように、Solaris デバイス ドライバは、レイヤードまたはモノリシック ドライバのいずれかです。

2.4 ドライバのエントリー ポイント

すべてのデバイスドライバは、C コンソール アプリケーションの main() 関数のような main のエントリー ポイントを 1 つ持っています。このエントリー ポイントを Windows では、DriverEntry() と呼び、Linux では、init_module() と呼びます。OS がデバイスドライバをロードする際に、このドライバのエントリー処理を呼びます。

初めてドライバをロードする際に、すべてのドライバが一度のみ実行する必要があるグローバルな初期化があります。このグローバルな初期化が DriverEntry() / init_module() ルーティンの役割をします。エントリ関数はまた、OS がどのドライバ コールバックを呼ぶかを登録します。これらのドライバ コールバックは、ドライバからのサービスで、OS の要求です。Windows の場合、これらのコールバックを dispatch routines と呼び、Linux の場合、file operations と呼びます。たとえば、ハードウェアの切断など、ある規定の結果として、各登録されたコールバックを OS が呼びます。

2.5 ハードウェアとドライバの連結

OS がデバイスをそのドライバにどのようにリンクさせるかは、OS によって異なります。Windows の場合、INF ファイルによって、そのリンクを行います。INF ファイルが、デバイスをドライバと動作するように登録します。この連結を DriverEntry() を呼ぶ前に実行します。OS がデバイスを認識し、デバイスと関連付けている INF ファイル内のデータベースを探し、INF ファイルによって、ドライバのエントリー ポイントを呼びます。

Linux の場合、デバイスとドライバ間のリンクを init_module() ルーティンで定義します。Init_module() ルーティンは、指定したドライバがどのハードウェア処理する示すコールバックを持っています。コードの定義を基にして、OS はドライバのエントリー ポイントを呼びます。

2.6 ドライバとの通信

ドライバはインスタンスを作成できるので、アプリケーションがドライバと通信をできるように、アプリケーションでドライバへのハンドルを開くことができます。アプリケーションは、ファイル アクセス API (Application Program Interface) を使用するドライバと通信します。アプリケーションは、ファイル名としてデバイスの名前を持った、CreateFile() (Windows の場合) の呼び出し、または open() (Linux の場合) の呼び出しを使用するドライバへのハンドルを開きます。デバイスからの read およびデバイスへの write を行うために、アプリケーションは ReadFile() および WriteFile() (Windows の場合) または read() および write() (Linux の場合) を呼びます。送信する要求をDeviceIoControl() (Windows の場合) および ioctl() (Linux の場合) と呼ばれる I/O コントロールの呼び出しを使用して実現します。この I/O コントロールの呼び出しで、アプリケーションは以下の内容を指定します。

- 呼び出し (デバイスのハンドルを提供することによって) を作成するデバイス
- デバイスが実行すべき関数を記述する IOCTL コード
- 実行される要求のデータを持ったバッファ

IOCTL コードは、ドライバとリクエスタが共通のタスクとして同意する数です。

ドライバとアプリケーション間で渡されるデータを構造体でカプセル化します。Windows の場合、この構造体を I/O Request Packet (IRP) と呼び、I/O Manager がカプセル化します。この構造体をデバイスドライバへ渡します。デバイスドライバはそれを編集し、他のデバイスドライバへ渡す場合もあります。

第 3 章
WinDriver USB の概要

この章では、USB バスの基本的な特徴や WinDriver USB の特徴およびアーキテクチャを説明します。

注意: この章の WinDriver USB ツールキットのリファレンスは、USB ホストドライバ開発用のスタンダード WinDriver USB ツールキットと関連しています。

USB デバイス ファームウェア開発用の WinDriver USB Device ツールキットに関する詳細は第 16 章 を参照してください。

3.1 USB の概要

USB (Universal Serial Bus) は、周辺機器をコンピュータに接続することを想定して PC アーキテクチャに追加された規格です。ユニバーサル シリアル バスは、Intel、Compaq、Microsoft、NEC などの PC 業界、テレコミュニケーションのリーダーにより 1995 年に開発されました。USB の開発時には、一般的な周辺機器の安価な接続方法を提供すること、PC の構成を簡単に変更できること、多くの周辺機器を接続可能なことなどがその目標として掲げられました。

USB 規格は、以上の必要性をすべてクリアしています。USB ポートには、最大で 127 個 (ハブを含む)の周辺デバイスを接続可能です。USB はまた、Plug-and-Play やホット スワップをサポートしており、USB 1.1 規格では等時性データ転送や非同期データ転送、倍速データ転送をサポートしています。低速の USB デバイスでは 1.5Mbps (メガビット毎秒)、高速 USB デバイスでは 12Mbps を達成しています (これもオリジナルのシリアル ポートよりも大幅に速度が向上しています)。デバイスと PC を接続するケーブルの長さは、最長で 5m です。USB はバスに接続された低電力デバイスに対して電力供給することが可能です (最大 500mA)。

USB2.0 規格は、USB 1.1 (フル) の転送速度よりも 40 倍高速な 480Mbs (メガビット毎秒) を達成します。USB 2.0 は USB 1.1 と完全に互換性を保っているため、同じケーブルやコネクタ、ソフトウェアを使用することが可能です。

USB2.0 はより高性能な帯域幅、PC 周辺機器の機能とのコネクションをサポートします。また、同時進行している周辺機器との互換性を保ちます。

USB2.0 は、対話式ゲーム、広帯域インターネット アクセス、デスクトップおよび Web パブリッシング、インターネット サービスおよびインターネット会議など、多くのアプリケーションの使用が可能となります。以上の利点により、USB は現在さまざまなマーケットで活用されています。

3.2 WinDriver USB の利点

このセクションでは、USB 規格および USB 規格をサポートする WinDriver USB ツールキットの主な利点について説明します。

- 最大限に簡単に使用できる外部接続方法。
- デバイスのドライバを自動的にマッピングし、自動設定を行います。
- コンピュータの動作中にデバイスを接続しても周辺機器を再設定します。
- データ転送率が Kb/s から Mb/s のデバイスに適しています。
- 同じケーブルで等時性転送と非同期転送をサポートします。
- 複数のデバイスの同時処理をサポートします (複数接続可能)。
- USB 2.0 (高速) を公式にサポートしている OS では最大 480 Mb/s 、USB 1.1 (高速) では最大 12 Mb/s のデータ転送速度をサポートします。
- 転送率や短い待ち時間が保証されます (等時性転送は、転送率のほとんどを使用します)。
- 柔軟性: 幅広い範囲のパケット サイズや、データ転送速度をサポートします。
- 強固性: プロトコルにエラー処理機能が組み込まれているため、動的にデバイスを追加したり、取り外したりしてもリアルタイムでデバイスの状況が監視されます。
- PC 業界の標準です。
- 周辺機器とホスト ハードウェアの統合に最適化されています。
- 実装にかかるコストが小さいため、安価な周辺機器の開発に適しています。
- ケーブルやコネクタも安価です。
- 電源管理や電源供給機能が組み込まれています。

3.3 USB のコンポーネント

USB の主なコンポーネントは次のとおりです:

USB ホスト: USB ホスト コントローラがインストールされていて、クライアント ソフトウェアやデバイス ドライバが動作する USB ホスト プラットフォームです。USB ホスト コントローラは、ホストと USB 機器のインターフェイスです。ホストは、USB 機器の検出や、ホストとデバイスのコントロール、データ フローの管理を行います。また、電力を USB 機器に供給するなどの機能もあります。

USB ハブ: USB ホストの 1 つの USB ポートに複数の USB デバイスを接続する際に使用する USB デバイスです。ホストに搭載されたハブを特にルート ハブと呼びます。これ以外のハブは、外部ハブです。

USB 機能: データの送受信や、バスの情報をコントロールして機能を提供する USB デバイスです。通常、USB 機能は、ケーブルによってハブに接続される個別の周辺機器として実装されます。しかし、1 つの USB ケーブルで複数の機能と埋め込み型ハブを実装する複合デバイスを作成することも可能です。ホストには、複合デバイスは、外部デバイスとの接続用ポートを持っている可能性のある、取り外し不可能な 1 つまたは複数の USB デバイスを備えたハブのように見えます。

3.4 USB デバイスのデータ フロー

USB デバイスの操作を行う際、ホストは、クライアント ソフトウェアとデバイスの間のデータ フローを開始することができます。

データは、一度にホストと 1 つのデバイス間でのみ転送することができます (ピアツーピア通信)。ただし、2 つのホストまたは 2 つの USB デバイスは直接通信できません (1 つのデバイスがマスタ (ホスト) となり、別のデバイスがスレーブとなる On-The-Go (OTG) デバイスはこの限りではありません)。

USB バスのデータは、ホストで動作しているソフトウェアのメモリ バッファとデバイスのエンドポイント間を動作するパイプを使って転送されます。

USB バスのデータフローは半二重なので、一度に一方向にのみ送信することが可能です。

エンドポイントは、USB デバイスのユニークな識別が可能なものであり、デバイスとのデータ フローの始点と終点を識別する目的で使用されます。各 USB デバイスには、論理的、または物理的なエンドポイントが複数存在します。3 つの USB 速度 (低速、フル、高速) はすべて、1 つの双方向コントロール エンドポイント (エンドポイント 0) と 15 個の一方向エンドポイントをサポートしています。各一方向エンドポイントは、IN 転送または OUT 転送として使用できるため、理論上は 30 個のエンドポイントをサポートしていることになります。各エンドポイントの属性には、バス アクセスの周波数、必要な転送率、エンドポイント番号、エラー処理機構、エンドポイントが送受信可能な最大パケット サイズ、転送タイプ、転送方向などが存在します。

パイプとは、USB デバイスのエンドポイントとホストのソフトウェアの関連を表す論理的なコンポーネントです。デバイスとのデータのやり取りは、パイプを通して行われます。パイプには、パイプで転送されるデータの種類によってストリーム パイプとメッセージ パイプの 2 種類が存在します。割り込み、バルク、等時性のデータを送信するパイプは、ストリーム パイプです。これに対し、コントロール転送タイプはメッセージ パイプでサポートされます。これらの USB 転送タイプは、次に説明します。

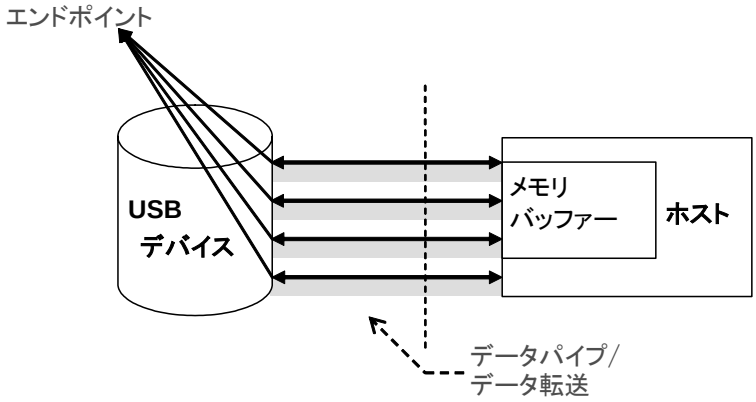


図 3.1: USB エンドポイント

3.5 USB データ交換

USB の標準ではホストとデバイスの間で機能的データ交換とコントロール交換の 2 種類のデータ交換をサポートしています。

- 機能的データ交換はデバイスからまたはデバイスへのデータの移動に使用されます。バルク転送、割り込み転送、等時性転送の 3 種類のデータ転送があります。

- コントロール交換は、デバイスを識別し、設定条件を決定して、デバイスを設定するのに使用されます。デバイス上の他のパイプのコントロールを含む、その他のデバイス特有の目的にも使用することができます。コントロール交換はコントロール パイプ (一般的にはデフォルトで、Pipe 0 です) を経由して転送されます。コントロール交換は、セットアップ ステージ (セットアップ パケットはホストからデバイスに送られます)、オプショナル データ ステージ、およびステータス ステージから構成されます。

図 3.2 は、WinDriver の DriverWizard ユーティリティ (第 5 章 を参照) により識別された両方向のコントロール (エンドポイント) と 6 つの機能的データ転送パイプ (エンドポイント) を持つ USB デバイスを示しています。

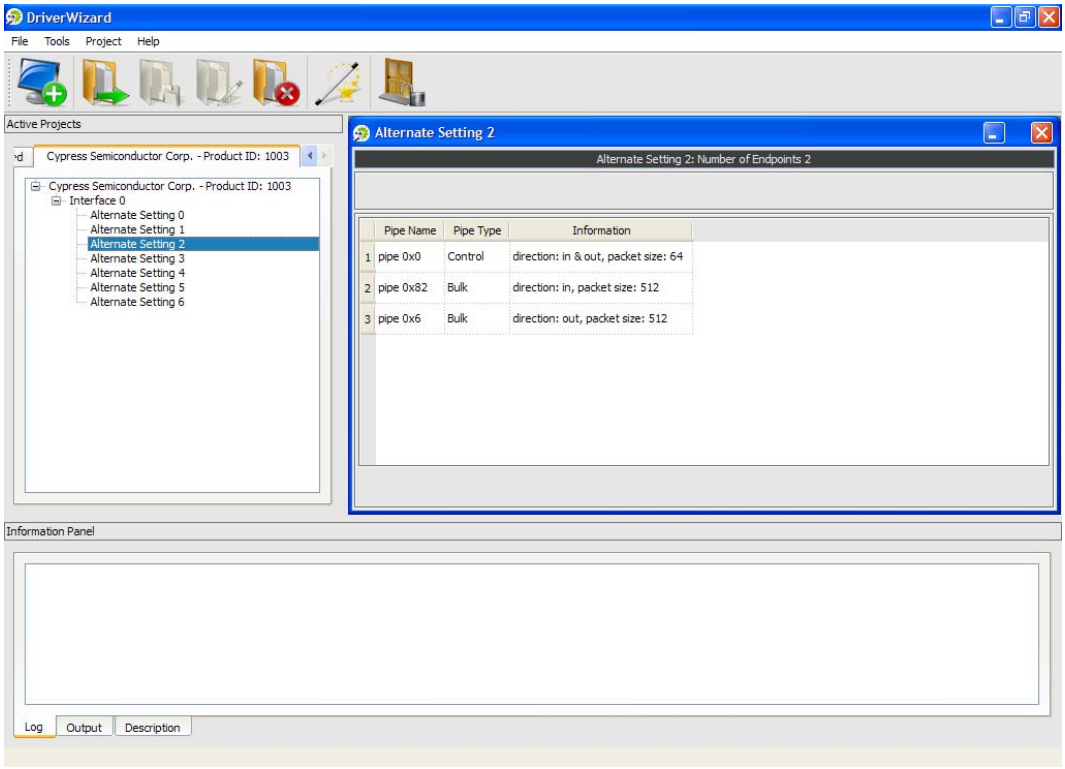


図 3.2: USB パイプ

セットアップ パケットの送信によりコントロール転送を実行する方法についての詳細は、第 9 章 の「実行に当たっての問題」を参照してください。

3.6 USB データ転送タイプ

USB デバイス (機能) は、ホストのメモリ バッファとデバイスのエンド ポイントの間をパイプを通して通信を行います。USB はデバイスとソフトウェアの使用目的にあわせて 4 つの転送タイプを用意しています。エンドポイントの転送タイプは、エンドタイプ記述子により決定されます。

USB の仕様では、4 種類のデータ転送が定義されています。

3.6.1 コントロール転送 (Control Transfer)

コントロール転送は、ホストのソフトウェアとデバイスとの間で主に設定操作、コマンド操作、ステータス操作をサポートするために使用されます。この転送タイプは、低速、フル、および高速デバイスで使用されます。

各 USB デバイスには、設定情報、ステータス情報、コントロール情報にアクセスするために最低 1 つのパイプ (デフォルト パイプ) が用意されています。コントロール転送は、非定期的な転送に使用されます。コントロール パイプは双方向のパイプで、データは両方向に流れることができます。コントロール転送にはまた、頑強なエラー検出、エラー リカバリ、再発信する機能が実装されており、これはドライバと独立してリトライを行います。コントロール エンドポイントの最大パケット サイズは、低速デバイスでは 8 バイトのみ、フル デバイスでは 8、16、32、または 64 バイト、高速デバイスでは 64 バイトのみです。

3.6.2 等時性転送 (Isochronous Transfer)

マルチメディアのストリームやテレフォニーなど、時間に依存する情報を扱う転送タイプです。この転送タイプは、フルおよび高速デバイスで使用され、低速デバイスでは使用されません。転送は定期的に連続的に行われます。等時性パイプは単方向であり、エンドポイントは情報の送信か受信のどちらかしかできません。双方向の等時性通信では、各方向ごとに等時性パイプを使用する必要があります。USB の等時性転送は決まった待ち時間の範囲内で USB の転送率を保証します。また、少ないデータが転送される場合を除いてデータ転送レートが守られることを保証します。この種類の転送ではデータの正当性よりも時間の方が重要なため、データ転送中にバスでエラーが発生してもリトライは行われません。

3.6.3 割り込み転送 (Interrupt Transfer)

割り込み転送は、少量のデータを送受信したり、非同期のタイム フレームで情報をやり取りするデバイスに使用されます。この転送タイプは、低速、フル、高速デバイスで使用されます。割り込み転送タイプは、最大のサービス ピリオドと、バスにエラーがあった場合、次のピリオドで転送が再試行されることが保証されています。割り込みパイプは、等時性パイプと同じ単方向です。割り込みエンドポイントの最大パケット サイズは、低速デバイスでは 8 バイト以下、フル デバイスでは 64 バイト以下、高速デバイスでは 1,024 バイト以下です。

3.6.4 バルク転送 (Bulk Transfer)

バルク転送は、非定期的に大きなパケットを通信する転送タイプです。バルク転送は、一般的に大量の時間に依存しないデータを転送するデバイスで使用されます。この際、使用可能な転送率をすべて使用するため、プリンタやスキャナなどのデバイスに使用されます。この転送タイプは、フルおよび高速デバイスで使用され、低速デバイスでは使用されません。バルク転送は、使用可能なバスを使用するため、データ転送は保証しますが待ち時間は保証しません。エラー検出機能が組み込まれているので再試行も行われます。他の転送に USB の転送率が使われていない場合は、システムはそれをバルク転送に使用します。ストリーム パイプ (等時性、割り込み) と同じように、バルク パイプは単方向です。このため、双方向転送の場合はエンドポイントが 2 つ必要になります。バルク エンドポイントの最大パケット サイズは、フル デバイスでは 8、16、32、または 64 バイト、高速デバイスでは 512 バイトです。

3.7 USB 設定

USB 機能 (またはデバイスの機能) を操作する前に、デバイスを設定する必要があります。ホストが USB デバイスから設定情報を取得して設定を行います。USB デバイスは記述子で属性をレポートします。USB 記述子の詳細は USB の仕様の第 9 章 を参照してください (完全な仕様書は、http://www.usb.org を参照してください)。

USB 記述子は 4 レベルの階層構造として説明できます:

- デバイス レベル

- 設定レベル
- インターフェイス レベル (このレベルには代替レベルというサブレベルを使用できます)。
- エンドポイント レベル

図 3.3 に示すように、各 USB デバイスのデバイス記述子はひとつしかありません。各デバイスにはひとつ以上の設定があり、各設定にはひとつ以上のインターフェイスがあり、各インターフェイスにはエンドポイントが存在します (存在しない場合もあります)。

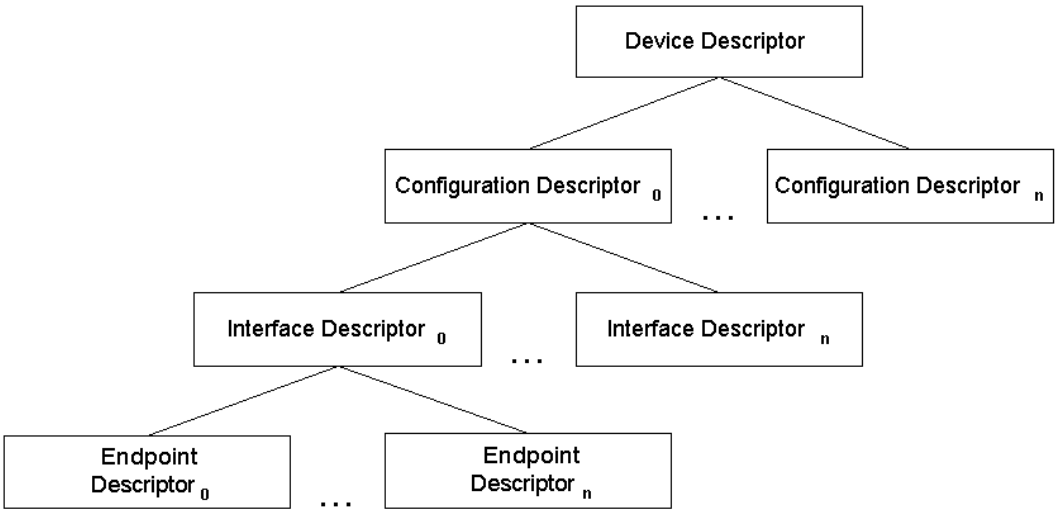


図 3.3: デバイス記述子

デバイス レベル: デバイス記述子には、すべてのデバイス設定のグローバル情報である一般的な情報が存在します。デバイス記述子には、デバイス クラス (HID デバイス、ハブ、ロケータ デバイスなど)、サブクラス、プロトコル コード、ベンダー ID、デバイス ID などの情報が含まれています。各 USB デバイスには、必ずデバイス記述子が存在します。

設定レベル: USB デバイスにはひとつ以上の設定記述子が存在します。各設定記述子は、設定のインターフェイスと電源属性を表します (セルフパワー、リモート Wakeup、最大電力消費値など)。設定は、ひとつずつしか使用できません。ISDN アダプタはひとつの 128Kbps インターフェイスとふたつの 64Kbps インターフェイスを同じデバイスで設定可能な例になります。

インターフェイス レベル: インターフェイスとは関連するエンドポイントの集合であり、デバイスの特定機能を表します。各インターフェイスは独立して動作する場合もあります。インターフェイス記述子は、インターフェイスの数、このインターフェイスが使用するエンドポイントの数、インターフェイス特有のクラス、サブクラスと、インターフェイスが単独で動作した場合のプロトコルの値を表します。さらに、インターフェイスには代替設定が可能です。代替設定は、デバイスを設定した後にエンドポイントやエンドポイントの特徴を変化できます。

エンドポイント レベル: エンドポイント記述子が一番ロー レベルになります。エンドポイント記述子はホストにこのエンドポイントのデータ転送タイプ、最大パケット サイズを表します。等時性エンドポイントは、この値を使ってデータ転送に必要なバス時間の予約を行います。他のエンドポイントの属性は、バス アクセスの周波数、エンドポイントの数、エラー処理の仕組みや、転送の方向を表します。同じエンドポイントは、異なる代替設定で、異なるプロパティ (または用途) を持つことができます。

WinDriver は USB の設定を自動化します。DriverWizard と USB 診断アプリケーションが USB バスをスキャンして、すべての USB デバイスを検出し、各デバイスの設定、インターフェイス、代替設定、エンドポ

イントを検出します。開発者はドライバの開発を開始する前に必要な設定を選択できます。WinDriver は、エンドポイント記述子に定義されているエンドポイント転送タイプを識別します。WinDriver が作成したドライバには、この段階で取得した設定情報がすべて含まれます。

3.8 WinDriver USB

WinDriver USB を使用すると、USB の仕様や OS の内部を把握しなくても、高性能な USB ベースのデバイスのドライバを簡単に開発できます。WinDriver USB を使用することにより DDK (Microsoft Driver Development Kit) や WDM (Win32 ドライバ モジュール) の知識がなくても USB ドライバの開発が可能になります。

WinDriver USB で開発したドライバ コードは、WinDriver がサポートする Windows プラットフォーム - Windows 98 / Me /2000 /XP / Server 2003 / Vista - でバイナリ互換があります。ソース コードは、WinDriver USB がサポートしているすべてのオペレーティング システム - Windows 98 / Me /2000 /XP / Server 2003 / Vista 、Windows CE.NET、Windows Embedded CE v6.00、Windows Mobile 5.0 / 6.0 および Linux - で互換性があります。WinDriver USB がサポートするオペレーティング システムの最新情報に関しては、エクセルソフト社の Web サイトを参照してください (http://www.xlsoft.com/)。

WinDriver USB は、すべてのベンダーの USB デバイスをサポートするツールキットです。WinDriver USB は、USB の仕様やアーキテクチャをカプセル化して、アプリケーションのロジックの開発に集中できるように設計されています。WinDriver USB の DriverWizard でハードウェアを検出、設定、テストなどをコードを記述する前に行えます。DriverWizard は必要な設定、インターフェイス、代替設定をグラフィカル ユーザー インターフェイスでまず設定可能にします。USB デバイスを検出し、設定を行ったらテストを行います。パイプ上でデータ転送を行ったり、制御要求を送信したり、パイプをリセットしたりして、ハードウェアのリソースが正しく動作しているかを確認できます。

ハードウェアの診断を終了したら、DriverWizard は C、C#、Visual Basic .NET、Delphi または Visual Basic でデバイス ドライバのソース コードを自動的に生成します。WinDriver USB では、アプリケーションから呼び出せるユーザー モード API を用意しています。WinDriver USB API は対象の USB デバイス特有のもので、パイプのリセットやデバイスのリセット など、USB 特有の操作を行えます。DriverWizard で生成されるコードは診断プログラムを実装し、特定のドライバで WinDriver USB API を使用する方法を示します。アプリケーションは、コンパイルして実行してください。このアプリケーションをドライバの雛型として使用して開発サイクルを開始し、ドライバを開発することも可能です。

DriverWizard は、対象のデバイスを WinDriver と動作するように登録する .INF ファイルも自動的に生成します。.INF ファイルは、WinDriver を使用して、正確に USB デバイスを識別し処理するために必要です。なぜ .INF ファイルを作成する必要があるのかについては、セクション 15.1.1 を参照してください。
DriverWizard を使って .INF ファイルを作成する方法の詳細は、セクション 5.2 の手順 3 を参照してください。

WinDriver USB を使用すると、すべての開発をユーザー モード、使い慣れた開発環境、デバッグ ツールおよびコンパイラ (MSDEV、Visual C/C++、MSDEV .NET、Borland C++ Builder、Borland Delphi、Visual Basic 6.0、MS eMbedded Visual C++、MS Platform Builder C++、GCC など) を使用して行えます。

3.9 WinDriver USB のアーキテクチャ

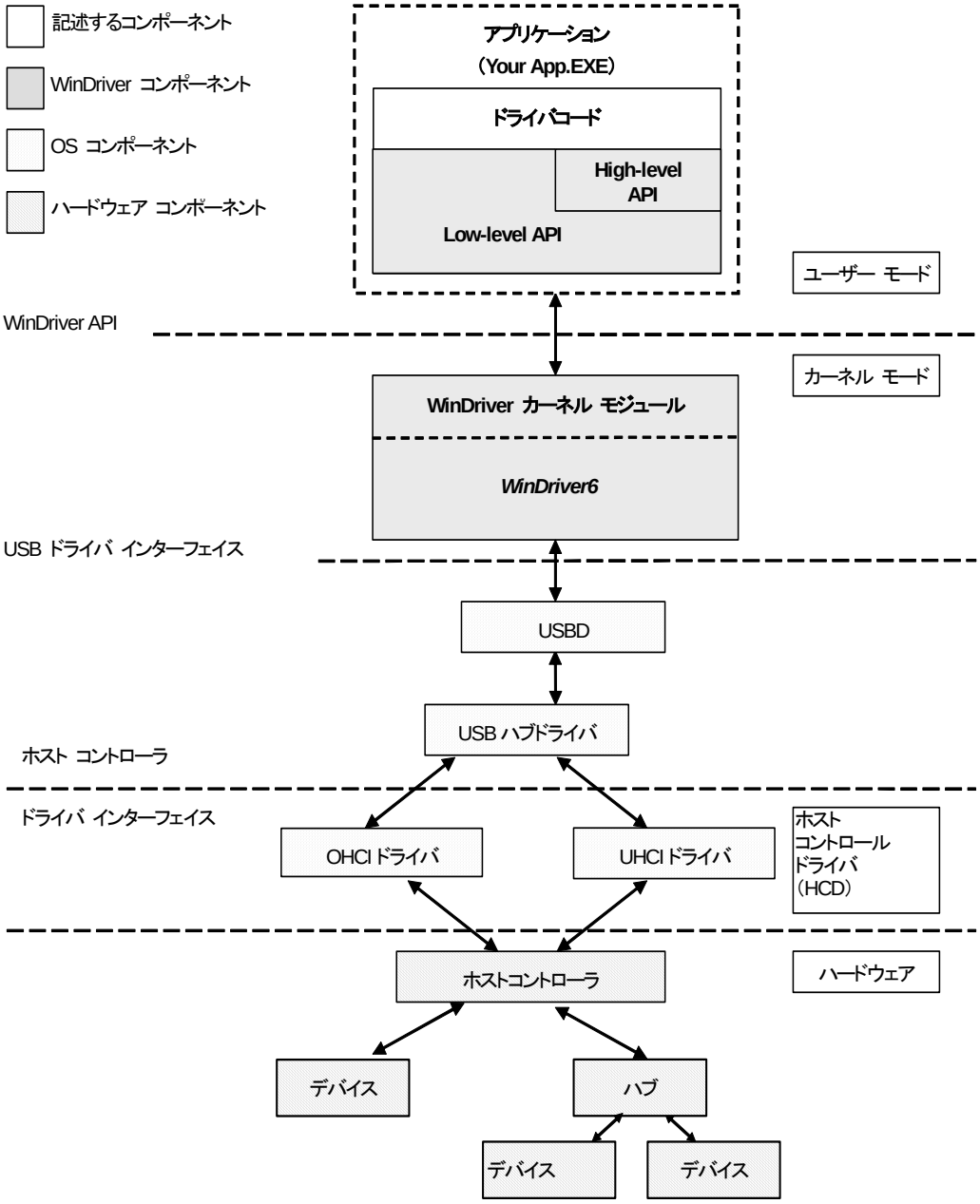


図 3.4: WinDriver USB アーキテクチャ

ハードウェアをアクセスするには、アプリケーションが WinDriver USB API に含まれる関数を使用する WinDriver カーネル モジュールを呼び出します。高水準関数は低水準関数を利用します。そして、WinDriver カーネル モジュールとユーザー モード アプリケーション間の通信を可能にする IOCTL を使用します。WinDriver カーネル モジュールは、USB デバイスのリソースをネイティブなオペレーティング システム コールでアクセスします。

USB デバイスと USB デバイスドライバを抽象化するため、2 つのレイヤーがあります。上位のものが USB ドライバ レイヤー (USB ドライバ (USBD) と USB ハブ ドライバ)、下位のものがホスト コントローラ ドライバ レイヤー HCD になります。HCD と USBD の境界線は、オペレーティング システムに依存するため定義されてい

ません。HCD と USBD の両方とも、ソフトウェア インターフェイスであり、オペレーティング システムのコンポーネントですが、HCD レイヤーが抽象化すると下位に表されます。

HCD は、ホスト コントローラ ハードウェアの抽象化を行うソフトウェア レイヤーです。また USBD は、USB デバイス自体とホスト ソフトウェアと USB デバイスの機能を抽象化します。

USBD はクライアント (特定のデバイス ドライバなど) と、USB ドライバ インターフェイス (USBDI) を使ってコミュニケートします。よりロー レベルでは USBD と USB ハブ ドライバが、ホスト コントローラ ドライバ インターフェイス (HCDI) を使って HCD とコミュニケートして、ハードウェアのアクセスとデータの転送を行います。

USB ハブ ドライバは、特定のハブに追加したり取り外したデバイスを検知する機能を持っています。ハブ ドライバがデバイスを追加、または取り外したことを伝える信号を受け取ると、ホスト ソフトウェアと USBD を追加して、デバイスを検出、設定します。設定を行うソフトウェアは、ハブ ドライバ、デバイス ドライバなどのソフトウェアに実装します。

WinDriver USB は、以上に説明した設定手順とハードウェア アクセスを抽象化します。WinDriver USB API を使用することにより、説明した手順をマスターすることなく、ハードウェア関連の操作を行うことが可能です。

3.10 WinDriver USB を使って作成できるドライバ

WinDriver USB を使用すると、ほとんどのモノリシック ドライバ (特定の USB デバイスにアクセスするドライバ) を作成可能です。NDIS ドライバ、SCSI ドライバ、ディスプレイ ドライバ、USB-シリアル ポート変換ドライバ、USB レイヤード ドライバなどの標準的なドライバを作成する場合は、KernelDriver USB をご利用ください。

開発時間を短縮したい場合は、KernelDriver USB よりも WinDriver USB を推奨します。

第 4 章
WinDriver のインストール

この章では、WinDriver のインストール手順や正常にインストールされたかどうかを確認する方法を紹介します。この章の最後では、アンインストールの方法も記述しています。

4.1 動作環境

4.1.1 Windows 98 / Me

- 32 ビットの x86 プロセッサ
- C、Visual Basic、または Delphi をサポートする 32 ビット開発環境

4.1.2 Windows 2000 / XP / Server 2003 / Vista

- 32 ビットまたは 64 ビット (x64: AMD64 または インテル EM64T) の x86 プロセッサ
- C、.NET, Visual Basic、または Delphi をサポートする開発環境
- Windows 2000: Service Pack 4
- Windows XP: Service Pack 2

4.1.3 Windows CE

- An x86 / MIPS / ARM Windows Embedded CE v6.00 または Windows CE 4.x - 5.0 (.NET) ターゲット プラットフォーム
  または
  ARMV4I Windows Mobile 5.0 / 6.0 ターゲット プラットフォーム
- Windows 2000 / XP / Server 2003 / Vista ホスト開発プラットフォーム
- **`Windows CE 4.x – 5.0`**: Microsoft eMbedded Visual C++ と対応するターゲット SDK または Microsoft Platform Builder とターゲット プラットフォーム用の対応する BSP (Board Support Package)

  `Windows Embedded CE 6.0`: Microsoft Visual Studio (MSDEV) .NET と Windows CE 6.0 Plugin

  `Windows Mobile`: Microsoft Visual Studio (MSDEV) .NET 2005

4.1.4 Linux

- Linux カーネル 2.2.x、2.4.x、または 2.6.x に対応する 32 ビットの x86 プロセッサ

または

Linux カーネル 2.4.x または 2.6.x に対応する 64 ビットの x86 プロセッサ (AMD64 またはインテル EM64T (**x86_64**))、またはインテル Itanium / Itanium 2 (**IA64**) プロセッサ

または

Linux カーネル 2.4.x または 2.6.x に対応する 32 ビットの PowerPC プロセッサ

- GCC コンパイラ

  注意: カーネルと同じバージョンの GCC コンパイラをご使用ください。

- C をサポートする 32 ビット または 64- ビット (どちらを使用するかはターゲットに依存) の開発環境 – ユーザー モード用

- 開発用 PC: **glibc2.3.x**

- WinDriver GUI アプリケーション (例: DriverWizard [第 5 章]、Debug Monitor [7.2]) を実行するのに必要な **libstdc++.so.5**

4.1.5 Solaris

- Solaris 8 / 9 / 10 / OpenSolaris

  注意: Solaris 8 には、アップデート 3 以降 (http://www.sun.com) をご使用ください。

- Sparc プラットフォームの 64 ビットまたは 32 ビット カーネル

  または

  Intel x86 プラットフォームの 32 ビット カーネル

- C (GCC など) をサポートする 32 ビット開発環境

- Intel x86 プラットフォームの Solaris 2.6 / 7.0 32 ビット (WinDriver v5.22 で対応)

注意: GCC 以外の開発環境を選択した場合、ご使用のコンピュータに **libgcc** がインストールされていることをご確認ください。次のサイトからダウンロード可能です http://www.sunfreeware.com。

以下のように、libgcc の場所を **LD_LIBRARY_PATH** へ設定します。

```
LD_LIBRARY_PATH= /usr/local/lib:/usr/local/lib/sparcv9
```

4.2 WinDriver のインストール

WinDriver CD には、各オペレーティング システム用の WinDriver が収録されています。CD のルート ディレクトリには、Windows 98 / Me / 2000 / XP / Server 2003 / Vista および Windows CE 用の WinDriver が収められており、CD ドライブに CD を 挿入すると自動的にインストールが開始されます。他の WinDriver バージョンは、サブ ディレクトリ (**Linux**、**Wince** など) に含まれています。

4.2.1 Windows にインストールするには

注意: WinDriver を Windows 98 / Me / 2000 / XP / Server 2003 / Vista にインストールするには、システムの管理者権限のあるユーザーで行う必要があります。

1. WinDriver CD を CD-ROM ドライブに挿入します (WinDriver CD からインストールせずに、ダウンロードした WinDriver をインストールする場合は、ダウンロードしたインストール ファイル (**WDxxx.EXE**、**xxx** はバージョン番号。例: WD900.EXE) をダブルクリックして、手順 3 に進んでください)。

2. インストール プログラムが自動的に起動します。自動的に起動しない場合は、**WDxxx.EXE** ファイルをダブルクリックしてください。[Install WinDriver] ボタンをクリックします。

3. 画面に表示されるライセンス同意書をお読みください。[Yes] を選択してライセンスに同意してください。

4. WinDriver をインストールする場所を選択します。

5. [Setup Type] ダイアログボックスで、次のいずれかを選択します。

   - **Typical** – すべての WinDriver モジュール(WinDriver ツールキットと特定チップセット用の API) をインストールします。
   - **Compact** – WinDriver ツールキットだけをインストールします。
   - **Custom** – インストールする WinDriver のモジュールを選択します。

6. インストーラがファイルのコピーを完了後、チュートリアルを開始するか選択します。

7. セットアップを完了したら、コンピュータを再起動してください。

注意: WinDriver のインストールは、**WD_BASEDIR** 環境変数にインストール時に指定された WinDriver ディレクトリを定義します。この変数は DriverWizard [第 5 章] コードを生成する際に、デフォルトの保存先を決定し、生成される project ファイルまたは make ファイルのinclude パスで使用されます。また、サンプル Kernel PlugIn のプロジェクトおよび makefile からも使用されます。このため、インストール後にWinDriver ディレクトリの名前や場所を変更する場合は、WD_BASEDIR 環境変数の値を編集し、新しいディレクトリを指定する必要があります。次の手順で WD_BASEDIR 環境変数の値を変更することができます。

1. [スタート] メニューから、[プログラム] - [設定] - [コントロール パネル] - [システム] を選択して、[システムのプロパティ] ダイアログ ボックスを開きます。

2. [詳細設定] タブで、[環境変数] ボタンをクリックします。

3. [システム環境変数] から WD_BASEDIR 変数を選択して、[編集] ボタンをクリックするか、ダブルクリックします。

4. [システム変数の編集] ダイアログ ボックスで、[変数値] を新しい WinDriver ディレクトリのフルパスに指定し、[OK] をクリックします。[環境変数] ダイアログ ボックス、[システムのプロパティ] ダイアログ ボックスでも [OK] をクリックします。

登録版ユーザーの場合

次の手順で、エクセルソフト株式会社から受け取ったライセンス コードを入力して WinDriver を登録します。

1. [スタート] メニューから、[プログラム] - [WinDriver] - [DriverWizard] の順に選択して、DriverWizard を起動します。

2. [File] メニューから [Register WinDriver] を選択して、[License Information] ダイアログボックスを表示します。

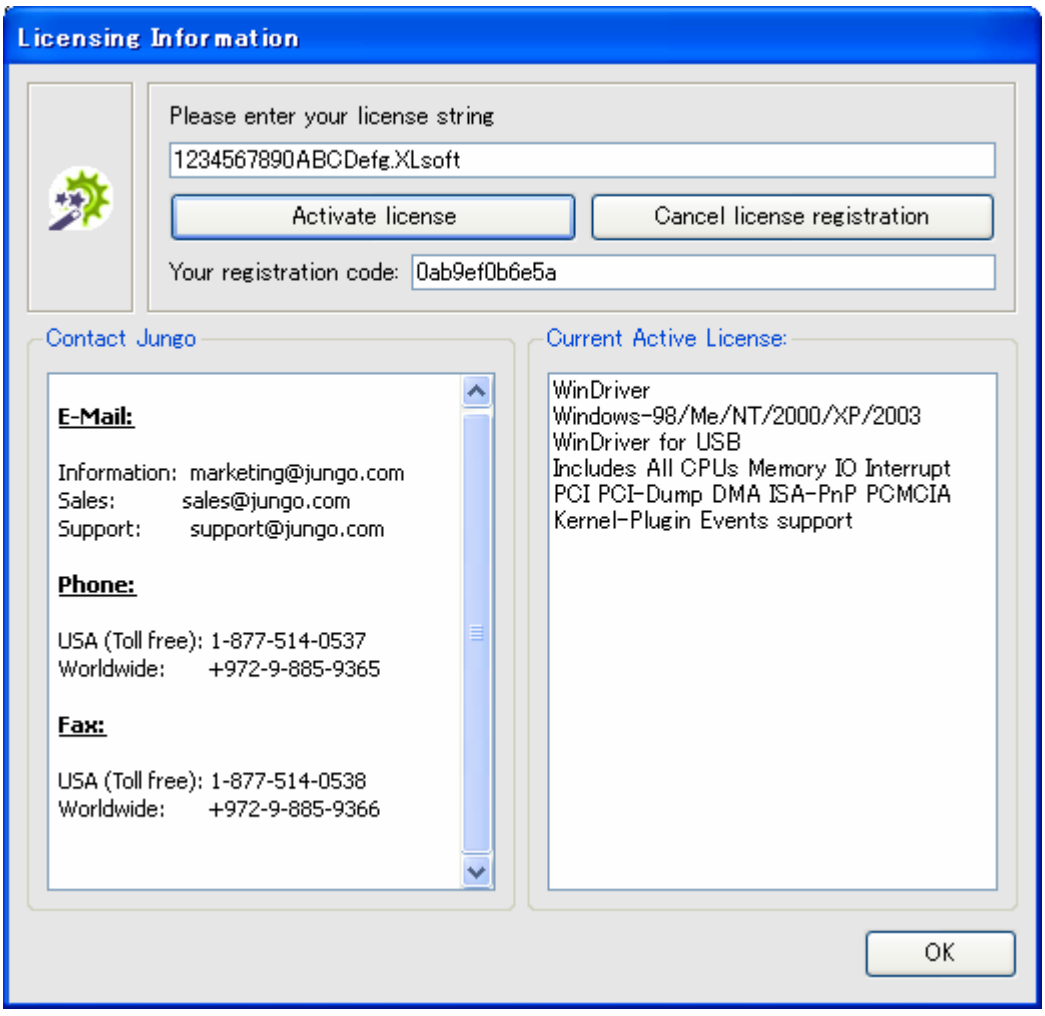
3. 以前のバージョンのライセンス コードが登録されている場合、[Cancel license registration] ボタンをクリックして、以前のバージョンのライセンス コードを解除します。

4. [Please enter your license string] 入力ボックスにエクセルソフト株式会社から受け取ったライセンス コードを入力して、[Activate license] をクリックし、ライセンス コードを登録します。

5. 試用期間中に開発したソース コードを有効にするには、次のセクションを参照してください:

   - `WDC_DriverOpen()` 関数
   - `WD_License()` 関数 (デフォルトで使用される `WDC_xxx` API の代わりに低水準の `WD_xxx` API を使用している場合)
   - `WDU_Init()` 関数

   エクセルソフト株式会社から受け取ったライセンス コードを使用して上記の関数で登録することによって、評価版で作成したサンプルを有効にします。



注意: カーネル上の現在のライセンスをチェックするには、[WinDriver Wizard] を実行して、[File] メニューから [Register WinDriver] を選択してください。現在、カーネルに設定されている有効なライセンスが表示されます。

ライセンス コードには、スペースおよびピリオドなども含まれますのでライセンス登録の際には、電子メールで受け取ったこの文字列を「コピー＆貼り付け」し、手入力によるミスを防いでください。

WinDriver をあなたのプログラムで使用するために

WinDriver ウィザードを使用しないで、あなたのプログラムから直接 WinDriver カーネルをアクセスするには、**WD_License 関数**、**WDC_DriverOpen 関数**、**WDU_Init 関数**を使用してライセンスをプログラム内で登録します。

例: 以下の RegisterWinDriver() をソースに組み込み、main() または WinMain() の先頭でコールしてください。

```
void RegisterWinDriver()
{
        HANDLE hWD;
        WD_LICENSE lic;

        hWD = WD_Open();
        if (hWD!=INVALID_HANDLE_VALUE)
        {
                // 以下の文字列にあなたのライセンスコードを入れてください。
                strcpy(lic.cLicense, "12345abcde12345.Company Name");
                WD_License(hWD, &lic);
                WD_Close(hWD);
        }
}
```

評価版で作成したソース コードを製品版として使用する場合にもこの記述を追加してください。

WinDriver のライセンス取得について

WinDriver 正式登録版を使用するには、「ライセンス コード」が必要です。「ライセンス コード」を取得していないお客様および代理店から購入されたお客様はパッケージに同封されている「ユーザー登録のご案内とライセンス コードの申請方法について」の要旨の説明に従い、Web サイトからライセンス コードの申請を行ってください。正式登録に必要なライセンス コードを発行致します。

注意: 現在、ライセンスコードには、ソフトウェア保護のため開発者の会社名 (必要に応じて部門名/ 開発者名) が登録されます。このライセンスコードはインストールするマシンの情報をもとに開発元の Jungo 社から発行されますので、ライセンス コード申請時にマシン情報 (DriverWizard の Your registration code) と社名の英語表記もあわせてお知らせください。

連絡先:

〒108 – 0014
東京都港区芝 5 - 1 - 9 ブゼンヤビル 4F
エクセルソフト株式会社
電話: 03 - 5440 - 7875 Fax: 03 - 5440 – 7876
E-mail: xlsoftkk@xlsoft.com

4.2.2 WinDriver CE のインストール

4.2.2.1 新規の CE ベース プラットフォームを開発する際に WinDriver CE をインストールする場合

次の説明は、Windows CE Platform Builder を使用して WinCE カーネル イメージをビルドするプラットフォーム開発者向けです。

注意:
以下の手順は、Windows CE Platform Builder、または MSDEV 2005 と Windows CE 6.0 plugin を使用して Windows CE カーネル イメージをビルドするプラットフォーム開発者向けです。本手順では、これらのプラットフォームの参照を "**Windows CD IDE**" の表記を使用します。

インストール前に Windows CE と デバイスドライバ の統合について Microsoft のドキュメントをよくお読みください。

1. ターゲット ハードウェアに一致したプロジェクト レジストリ ファイルを編集します。ステップ 2 で、WinDriver コンポーネントを使用するように選択した場合、編集するレジストリ ファイルは、**WinDriver\samples\wince_install\<TARGET_CPU>\WinDriver.reg** (例えば、**WinDriver\samples\wince_install\ARMV4I\WinDriver.reg**) となります。もしくは、**WinDriver\samples\wince_install\project_wd.reg** ファイルを編集します。

2. Sysgen プラットフォームのコンパイル ステージの前に、このステップで記述されている手順に従って Windows CE プラットフォームにドライバを簡単に統合できます。
   注意:

   - このステップに記載されている手順は、Windows CE 4.x - 5.x with Platform Builder を使用する開発者のみに関連します。
     Windows CE 6.x with MSDEV 2005 を使用する開発者は次のステップ 3 に進んでください。

   - この手順では、対象の Windows CE プラットフォームに WinDriver を統合する便利な方法を紹介します。この方法を使用しない場合、Sysgen ステージの後で、ステップ 4 で記述されている手動の統合ステップを実行する必要があります。

   - このステップで記述されている手順で、WinDriver のカーネル モジュール (**windrvr6.dll**) を対象の OS イメージに追加します。WinDriver CE カーネル ファイル (**windrvr6.dll**) を永続的に Windows CE イメージ (**NK.BIN**) の一部とする場合にのみこのステップが必要です。例えば、フロッピーディスクを使用してターゲット プラットフォームにカーネル ファイルを移す場合などがこれに該当します。オン デマンドで CESH/PPSH サービスを通して **windrvr6.dll** をロードする場合、このステップで記述されている手順を実行しないで、ステップ 4 で記述されている手動による統合の方法を実行する必要があります。

     a. Windows CD IDE を実行してプラットフォームを開きます。

     b. **File** メニューから **Manage Catalog Items**.... を選択し、**Import**… ボタンをクリックし、関連する **WinDriver\samples\wince_install\<TARGET_CPU>** ディレクトリ (例えば、**WinDriver\samples\wince_install\ARMV4I**) から **WinDriver.cec** を選択します。
        これで WinDriver のコンポーネントを Platform Builder Catalog へ追加します。

     c. **Catalog** ビューで、**Third Party** ツリーの **WinDriver Component** ノードをマウスの右クリックし、**Add to OS design** を選択します。

3. 対象の Windows CE プラットフォームをコンパイルします (Sysgen ステージ)。

4. 上記のステップ 2 で記述された手順を実行しなかった場合、対象のプラットフォームに手動でドライバを統合するために、Sysgen ステージの後で、以下のステップを実行してください。
   注意: 上記のステップ 2 で記述された手順を実行した場合には、このステップをスキップし、直接ステップ 5 へ進んでください。

   a. Windows CD IDE を実行してプラットフォームを開きます。

   b. **Build** メニューから **Open Build Release Directory** を選択します。

   c. WinDriver CE カーネル ファイル - **WinDriver\redist\<TARGET_CPU>\windrvr6.dll** - を開発プラットフォーム上の **%_FLATRELEASEDIR%** サブディレクトリにコピーします。

   d. **WinDriver\samples\wince_install** ディレクトリの **project_wd.reg** ファイルの内容を **%_FLATRELEASEDIR%** サブディレクトリの **project.reg** ファイルに追加します。

      WinDriver CE カーネル ファイル (**windrvr6.dll**) を永続的に Windows CE イメージ (**NK.BIN**) の一部とする場合にのみこのステップが必要です。例えば、フロッピーディスクを使用してターゲット プラットフォームにカーネル ファイルを移す場合などがこれに該当します。オン デマンドで CESH/PPSH サービスを通して **windrvr6.dll** をロードする場合、永続カーネルをビルドするまでこのステップを実行する必要はありません。

5. **Build** メニューより **Make Image** を選択し、新しいイメージ **NK.BIN** の名前をつけます。

6. ターゲット プラットフォームに新しいカーネルをダウンロードし、**Target** メニューより **Download** / **Initialize** を選択するか、またはフロッピー ディスクを使用して初期化します。

7. ターゲット CE プラットフォームを再起動します。WinDriver CE カーネルは自動的にロードします。

8. サンプル プログラムをコンパイルして起動し、WinDriver CE がロードされ、正常に動作するのを確認してください。

4.2.2.2 Windows CE ベース コンピュータ用のアプリケーションを開発する際に WinDriver CE をインストールする場合

注意: 指定がない限り、このセクションの "Windows CE" の記述は、Windows Mobile を含む、対応するすべての Windows CE プラットフォームを表します。

この手順は、WinCE カーネルをビルドするのではなく、ドライバのダウンロードのみ行うドライバ開発者、または既成の WinCE プラットフォームに Microsoft eMbedded Visual C++ (Windows CE 4.x – 5.x) または MSDEV .NET 2005 (Windows Mobile または Windows CE 6.x) を使用してビルドするドライバ開発者向けです。

9. WinDriver CD を Windows ホスト マシンの CD ドライブにセットします。

10. 自動インストールを終了します。

11. CD の **WINCE** ディレクトリにある **WDxxxCE.EXE** をダブル クリックします。このプログラムは必要な WinDriver のファイルをホスト開発プラットフォームにコピーします。

12. WinDriver CE カーネル ファイル **WinDriver\redist\TARGET_CPU\windrvr6.dll** をターゲットの CE コンピュータの **WINDOWS** サブディレクトリにコピーします。

13. 起動時に Windows CE がロードするデバイスドライバのリストに WinDriver を追加します:

    - **\WinDriver\samples\wince_install\PROJECT_WD.REG** ファイルに記載されたエントリに従って、レジストリを編集します。ハンドヘルド CE コンピュータの Windows CE Pocket Registry を使用するか、または MS eMbedded Visual C++ (Windows CE 4.x - 5.x) / MSDEV .NET 2005 (Windows Mobile または Windows CE 6.x) で提供される Remote CE Registry Editor Tool を使用して実行します。Remote CE Registry Editor ツールを使用するには、対象の Windows ホスト プラットフォームに Windows CE Services がインストールされている必要があります。

    - Windows Mobile では、起動時に OS のセキュリティ スキーマが署名されていないドライバのロードを防ぎます。従って、起動後に、WinDriver のカーネル モジュールを再ロードする必要があります。ターゲットの Windows Mobile プラットフォームで、OS の起動時に毎回、WinDriver をロードするには、**WinDriver\redist\Windows_Mobile_5_ARMV4I\wdreg.exe** ユーティリティをターゲットの Windows\StartUp\ ディレクトリにコピーします。

14. ターゲット CE コンピュータを再起動します。WinDriver CE カーネルは自動的にロードします。suspend/resume ではなく、システムの再起動を行ってください (ターゲット CE コンピュータのリセットまたは電源ボタンを使用します) 。

15. サンプル プログラムをコンパイルして起動し、WinDriver CE がロードされ、正常に動作するのを確認してください。

4.2.2.3 Windows CE のインストールにおける注意事項

Windows 2000 / XP / Server 2003 / Visata ホスト PC での WinDriver のインストールでは、**WD_BASEDIR** 環境変数を定義します (インストール中に選択した WinDriver のディレクトリの場所を示します)。WinDriver の DriverWizard でコードを生成する際には、この変数を使用します - 生成したコードを保存するデフォルトのディレクトリで、生成された project / make ファイルの include パスに使用します。サンプルの Kernel PlugIn プロジェクトおよび makefile でも、この変数を使用します。

従って、WinDriver のインストール後、WinDriver のディレクトリの名前 / 場所を変更する場合、WD_BASEDIR 環境変数の値を変更し、新しい WinDriver のディレクトリの場所を指すように設定する必要があります。以下の手順で、WD_BASEDIR の値を変更できます:

1. [スタート] メニューから、[プログラム] - [設定] - [コントロール パネル] - [システム] を選択して、[システムのプロパティ] ダイアログ ボックスを開きます。

2. [詳細設定] タブで、[環境変数] ボタンをクリックします。

3. [システム環境変数] から WD_BASEDIR 変数を選択して、[編集] ボタンをクリックするか、ダブルクリックします。

4. [システム変数の編集] ダイアログ ボックスで、[変数値] を新しい WinDriver ディレクトリのフルパスに指定し、[OK] をクリックします。[環境変数] ダイアログ ボックス、[システムのプロパティ] ダイアログ ボックスでも [OK] をクリックします。

注意: WinDriver Windows 2000 / XP / Server 2003 / Vista ツールキットを同じホスト PC にインストールすると、Windows CE のインストールで設定された WD_BASEDIR 変数の値が上書きされます。

4.2.3 Linux に WinDriver をインストールするには

4.2.3.1 インストールするシステムの用意

Linux では、カーネル自身をコンパイルしたのと同じヘッダー ファイルでカーネル モジュールをコンパイルする必要があります。WinDriver は、カーネル モジュール **windrvr6.o/.ko** をインストールするため、インストールの際に **windrvr6.o/.ko** を Linux カーネルのヘッダー ファイルでコンパイルする必要があります。

そのため、WinDriver for Linux をインストールする前に、Linux ソース コードおよび **versions.h** ファイルがご使用のマシンにインストールされていることを確認してください:

Linux カーネル ソース コードのインストール

- Linux をインストールする際に [Custom] を選択してインストールしてからソース コードのインストールを選択します。
- Linux がコンピュータにインストールされている場合、Linux ソース コードがインストールされているか確認します。**/usr/src** ディレクトリの 'linux' をご確認ください。ソース コードがインストールされていない場合、ソースコードをインストールするか、Linux をソース コードつきで再インストールします。

version.h のインストール

- **version.h** ファイルは、Linux カーネル ソース コードを最初にコンパイルしたときに作成されます。提供されるコンパイル済みカーネルに **version.h** が含まれていない場合があります。このファイルを確認するには **/usr/src/linux/include/linux/** を参照します。このファイルがない場合は次の操作を行ってください。

  ① **'make xconfig'**と入力します。

  ② **'Save and Exit'** を選択して設定情報を保存します。

  ③ **'make dep'** と入力します。

WinDriver GUI アプリケーション (例: DriverWizard [第 5 章]、Debug Monitor [7.2]) を実行するには、バージョン 5 の **libstdc++** (**libstdc++.so.5**) が必要です。このファイルがインストールされていない場合は、適切な RPM (例: **compat-libstdc++**) を利用してインストールしてください。

インストールを行う前に、'linux' シンボリック リンクがあることを確認します。ない場合は作成します。
ln -s <target kernel>/ linux

たとえば、Linux 2.4 カーネルの場合、次を入力します。
ln -s linux-2.4/ linux

4.2.3.2 インストール

1. WinDriver CD を Linux マシン CD ドライブに挿入するか、またはダウンロードしたファイルを適当なディレクトリに保存します。

2. インストール を行うディレクトリに移動します (例 /home/username/)。
   $ cd /home/username

3. WD800LN.tgz ファイルを解凍します。
   $ tar xvzf /<file location>/WD900LN.tgz

   例:

   - CD の場合:
     $ tar xvzf /mnt/cdrom/LINUX/WD900LN.tgz
   - ダウンロードファイルの場合:
     $ tar xvzf /home/username/WD900LN.tgz

4. **WinDriver/redist/** ディレクトリに移動します (このディレクトリは tar によって作成されたディレクトリです)。
   $ cd <WinDriver directory path>/redist/

5. WinDriver をインストールします。

   - <WinDriver directory>/redist$ **./configure**

     注意: configure スクリプトは、特定の実行中のカーネルを基に **makefile** を作成します。**-with-kernel-source=<path>** フラグを configure スクリプトへ追加して、インストールされたその他のカーネルを基に **configure** スクリプトを実行することもできます。<path> はカーネル ソース ディレクトリへのフルパス (例: **/usr/src/linux**) です。

   - <WinDriver directory>/redist$ **make**

   - 'スーパー ユーザー' になります。
     <WinDriver directory>/redist$ **su**

   - ドライバをインストールします。
     <WinDriver directory>/redist# **make install**

6. シンボリック リンクを作成し、DriverWizard GUI を簡単に起動できるようにします。
   ln -s <WinDriver の絶対パス>/WinDriver/wizard/wdwizard/usr/bin/wdwizard

7. **wdwizard** ファイルに read (読み取り) / excute (実行) の権限を設定し、ほかのユーザーがプログラムにアクセスできるようにします。

8. ユーザーおよびグループ ID を変更します。必要に応じて read (読み取り) / write (書き込み) 権限をデバイス ファイル **/dev/windrvr** に与え、ユーザーにデバイスを介してハードウェアにアクセスできるようにします。

   udev ファイル システムを使用する Linux カーネル 2.6.x を使用している場合は、**/etc/udev/permissions.d/50-udev.permissions** ファイルを編集して、権限を変更します。たとえば、次の行を追加して、read (読み取り) および write (書き込み) 権限を与えます。
   windrvr6:root:root:0666

   次のように chmod コマンドを使用することもできます。
   chmod /dev/windrvr6 666

9. **WD_BASEDIR** 環境変数にインストール時に指定された WinDriver ディレクトリを定義します。この変数は WinDriver のサンプルと DriverWizard [第 5 章] で生成されるコードの make ファイルおよびソースファイルで使用されます。また、DriverWizard で生成されるプロジェクトのデフォルトの保存先を決定するのにも使用されます。この変数を定義せずに、WinDriver の makefile を使用してサンプル コードまたは生成されたコードをビルドしようとすると、変数を定義するように指示されます。

   注意: インストール後にWinDriver ディレクトリの名前や場所を変更する場合は、WD_BASEDIR 環境変数の値を編集し、新しいディレクトリを指定する必要があります。

10. WinDriver を使用して、ハードウェアにアクセスを開始し、ドライバ コードを生成します。

ヒント: Linux の **/etc/rc.d/rc.local** ファイルに次の行を追加して **wdreg** スクリプトを起動すると、システムを再起動するごとにドライバ モジュール (**windrvr6.o/.ko**) を自動的にロードします。

```
wdreg windrvr6
```

登録版ユーザーの場合

次の手順で、エクセルソフト株式会社から受け取ったライセンス コードを入力して WinDriver を登録します。

1. DriverWizard GUI を起動します。
   <path to WinDriver>/wizard/wdwizard

2. [File] メニューから [Register WinDriver] オプションを選択して、[License Information] ダイアログ ボックスを表示します。

3. 以前のバージョンのライセンス コードが登録されている場合、[Cancel license registration] ボタンをクリックして、以前のバージョンのライセンス コードを解除します。

4. [Please enter your license string] 入力ボックスにエクセルソフト株式会社から受け取ったライセンス コードを入力して、[Activate license] をクリックし、ライセンス コードを登録します。

5. 評価版 WinDriver を使用して作成したソースコードを登録するには、WDC_DriverOpen() 関数 (PCI の場合) または WDU_Init () 関数 (USB の場合) を参照してください。

   デフォルトで使用される WDC_xxx API の代わりに低水準の WD_xxx API を使用している場合は、WD_License() 関数を参照してください。

4.2.3.3 Linux でハードウェアへのアクセスを制限するには

注意: /dev/windrvr6 は、ユーザー プログラムへの直接的なハードウェア アクセスを与えるため、マルチ ユーザー Linux システムの安定性に影響する可能性があります。DriverWizard へのアクセスおよびデバイス ファイル **/dev/windrvr6** へのアクセスを信頼できるユーザーのみに制限してください。

セキュリティのため、WinDriver インストール スクリプトは、**/dev/windrvr6** および DriverWizard 実行ファイル (**wdwizard**) への権限の変更を自動的に行いません。

4.2.4 Solaris に WinDriver をインストールするには

WinDriver のインストールは、カーネル モジュール **windrvr6.o** をインストールするため、スーパー ユーザーまたはルート権限を持つユーザーによってインストールする必要があります。

1. WinDriver CD を Solaris マシンの CD ドライブに挿入するか、適当なディレクトリ (例 /home/username/) にダウンロードファイルをコピーします。

2. インストレーションを行うディレクトリに移動します (例 /home/username)。
   `$ cd /home/username`

3. WinDriver の配布用ファイル (**`WD900SL.tgz`** (x86) / **`WD900SLS32.tgz`** (SPARC 32 ビット) / **`WD900SLS64.tgz`** (SPARC 64 ビット)) を現在のディレクトリにコピーします。
   `$ cp /home/username/WD800SL.tgz`

4. ファイルを展開します。
   `$ gunzip -c WD900SL.tgz | tar xvf -`

5. WinDriver ディレクトリに移動します。

6. インストール スクリプト **`WinDriver/install_windrvr`** を使用して、WinDriver をインストールします。
   ~/WinDriver# **`./install_windrvr`**

   インストールの際に、デバイスの Vendor ID と Device ID (16 進数) を定義して、どの PCI カードを対象にするか決定する必要があります。二つの方法があります (**`<vid>`** は Vendor ID、**`<did>`** は Device ID)。
   `~/WinDriver# ./install_windrvr <vid>,<did> [<vid>,<did> ...]`

   たとえば、PLX 9030 および PLX 9054 を使用する場合は、次を実行します。
   `~/WinDriver# ./install_windrvr 10b5,9030 10b5,9054`

7. **`libgcc`** パッケージをインストールします (http://www.sunfreeware.com/ よりダウンロードすることができます)。

8. 環境変数を追加します。

   - SPARC 32 ビット および x86 プラットフォームの場合:
     `LD_LIBRARY_PATH=/usr/local/bin`
   - SPARC 64 ビット プラットフォームの場合:
     LD_LIBRARY_PATH=/usr/local/lib:/usr/local/lib/sparcv9

以下の手順はオプションです

1. シンボリック リンクを作成し、DriverWizard GUI を簡単に起動できるようにします。
   ~/WinDriver# **`ln -s ~/WinDriver/wizard/wdwizard/ usr/bin/wdwizard`**

2. **`wdwizard`** ファイル の read (読み取り) および excute (実行) の権限を変更し、一般ユーザーがプログラムにアクセスできるようにします。

3. ユーザーおよびグループ ID を変更し、さらに必要に応じて read (読み取り) / write (書き込み) 権限をデバイス ファイル **`/dev/windrvr6`** に与え、ユーザーにデバイスを通じハードウェアにアクセスできるようにします。

4. WinDriver を使用してハードウェアにアクセスしてドライバ コードを生成できます。

以下のステップは登録版ユーザー用です

次の手順で、エクセルソフト株式会社から受け取ったライセンス コードを入力して WinDriver を登録します。

1. DriverWizard GUI を起動します。
   ```
   ~/WinDriver/wizard$ ./wdwizard
   ```

2. [File] メニューから [Register WinDriver] を選択して、[License Information] ダイアログボックスを表示します。

3. 以前のバージョンのライセンス コードが登録されている場合、[Cancel license registration] ボタンをクリックして、以前のバージョンのライセンス コードを解除します。

4. [Please enter your license string] 入力ボックスにエクセルソフト株式会社から受け取ったライセンス コードを入力して、[Activate license] をクリックし、ライセンス コードを登録します。

5. 評価版 WinDriver を使用中に作成したソースコードを登録するには、`WDC_DriverOpen()` 関数 (PCI の場合) を参照してください。

   `WD_License()` 関数(デフォルトで使用される `WDC_xxx` API の代わりに低水準の `WD_xxx` API を使用している場合)

4.2.4.1 Solaris でハードウェアへのアクセスを制限する

注意: `/dev/windrvr6` は、ユーザー プログラムへの直接的なアクセスを与えるため、マルチ ユーザー Solaris システムの安定性に影響する可能性があります。DriverWizard およびデバイス ファイル **`/dev/windrvr6`** へのアクセスを信頼するユーザーのみに制限してください。

セキュリティのため、WinDriver インストレーション スクリプトは、**`/dev/windrvr6`** および DriverWizard 実行ファイル (**`wdwizard`**) への権限の変更を自動的に行いません。

4.3 アップグレード版のインストール

Windows 版 WinDriver を新しいバージョンにアップグレードするには、Windows 2000 / XP / Server 2003 / Vista に WinDriver をインストールする手順が説明されているセクション 4.2.1 の「Windows 98 / Me / 2000 / XP / Server 2003 / Vista にインストールするには」にあるステップを実行します。既存のインストールを上書きするか、別のディレクトリにインストールすることができます。

インストール後、DriverWizard を起動し、(ライセンスをお持ちの場合) ライセンス文字列を入力します。これで WinDriver のアップグレードは終了です。

ソース コードをアップグレードするには、新しいライセンス文字列をパラメータとして `WDC_DriverOpen()` (PCI の場合)、`WDU_Init ()` (USB の場合) 、または `WD_License()` (デフォルトで使用される `WDC_xxx` API の代わりに低水準の `WD_xxx` API を使用している場合) に渡します。

他のオペレーティング システムでインストールをアップグレードするには、上記と同じ手順で行います。インストールの詳細については、各インストール セクションを参照してください。

4.4 インストールの確認

4.4.1 Windows、Linux および Solaris コンピュータの場合

1. Windows の場合 **[スタート]** メニューから **[プログラム] - [WinDriver] - [DriverWizard]** を選択して DriverWizard を実行するか、またはデスクトップに作成されたショートカットを使用します。コマンド プロンプトから wdizard.exe を実行して DriverWizard を開始することもできます。

   Linux および Solaris では、wizard のサブディレクトリからファイル マネージャを使用して、ウィザード アプリケーションへアクセスすることができます。
   または、shell から ウィザード アプリケーションへアクセスすることもできます。

2. WinDriver のライセンスを確認します(セクション 4.2 の「WinDriver のインストール」を参照してください)。評価版を使用している場合、ライセンスをインストールする必要はありません。

3. PCI カードの場合 – PCI バスにカードを挿入します。Driver Wizard が検出するのを確認します。

4. ISA カードの場合 – ISA バスにカードを挿入します。DriverWizard をカードのリソースに合わせて設定し、DriverWizard からカードを読み書きできるかどうかを確認してください。

4.4.2 Windows CE コンピュータの場合

1. Windows ホスト マシンの [スタート] メニューから [プログラム] - [WinDriver] - [DriverWizard] を選択して、DriverWizard を実行してください。

2. 登録ユーザーの場合、WinDriver ライセンスがインストールされているか確認してください。評価版を使用している場合はライセンスをインストールする必要はありません。

3. Plug-and-Play デバイス (PCI、PCMCIA、または CardBus) の場合、デバイスを適切なバス スロットへ挿入し、DriverWizard を検出するかを確認してください。

4. ISA カードの場合、カードを ISA バスに挿入し、DriverWizard をカードのリソースに合わせて設定し、DriverWizard からカードを読み書きできるかどうかを確認してください。

5. Visual C++ for CE をアクティブにします。

6. WinDriver サンプル (例: **`WinDriver\samples\speaker\speaker.dsw`**) をロードします。

7. Visual C++ WCE 設定ツールバーで、ターゲット プラットフォームを x86em に選択します。

8. スピーカー サンプルのコンパイルし、実行をします。Windows ホスト マシンのスピーカーが CE エミュレーション環境でアクティブになるはずです。

4.5 WinDriver をアンインストールするには

評価版または登録版の WinDriver をアンインストールする必要がある場合は、このセクションを参照してください。

4.5.1 Windows WinDriver をアンインストールするには

注意:

- **Windows 98 / Me** では、**wdreg** への参照を **wdreg16** に変更します。
- **Windows 2000 / XP / Server 2003 / Vista** では、**wdreg.exe** の代わりに **wdreg_gui.exe** を使用することができます。
- **wdreg.exe**、**wdreg_gui.exe**、および **wdreg16.exe** は **WinDriver\util** ディレクトリにあります (これらのユーティリティの詳細は、第 13 章 を参照してください)。

1. 開いている WinDriver アプリケーション (DriverWizard、Debug Monitor (**wddebug_gui.exe**) およびその他の WinDriver アプリケーション) を閉じます。
2. Kernel PlugIn ドライバを作成した場合には、以下を実行します。
   - 作成した Kernel PlugIn ドライバをインストールしてる場合、**wdreg** ユーティリティを使用してアンインストールします。
     wdreg -name <Kernel PlugIn の名前> uninstall

     注意: Kernel PlugIn の名前は、*.sys 拡張子無しで指定してください。
   - Kernel PlugIn ドライバを **%windir%\system32\drivers** ディレクトリから削除します。
3. Plug-and-Play をサポートしている Windows システム (Windows 98 / Me / 2000 / XP / Server 2003 / Vista) の場合:
   - INF ファイルを通じて WinDriver と動作するように登録された Plug-and-Play デバイス (USB / PCI / PCMCIA) をアンインストールします。
     - Windows 2000 / XP / Server 2003 / Vista では、**wdreg** ユーティリティを使用してアンインストールします。
       wdreg -inf <*inf ファイルへのフルパス> uninstall
     - Windows 98 / Me では、手動でデバイス マネージャから対象のデバイスをアンインストール (削除) します。
   - **%windir%\inf** ディレクトリまたは **%windir%\inf\other** ディレクトリ (Windows 98 / Me) に、WinDriver のカーネル モジュール (**windrvr6.sys**) と動作するように登録した デバイスの *.inf ファイルが存在しないことをご確認ください。
4. WinDriver をアンインストールします。
   - WinDriver ツールキットがインストールされている開発用 PC の場合:
     [スタート] メニューから [プログラム] - [WinDriver] - [Uninstall] を選択するか、または **WinDriver** ディレクトリにある **uninstall.exe** を実行します。

     WinDriver カーネル モジュール (**windrvr6.sys**) を停止し、アンロードし、**%windir%inf** ディレクトリ (Windows 2000 / XP / Server 2003 / Vista) または **%windir%inf\other** ディレクトリ (Windows 98 / Me) から **windrvr6.inf** ファイルのコピーを削除し、Windows の スタート メニューから WinDriver を削除し、デスクトップからは DriverWizard と Debug Monitor のショートカット アイコンを削除します。また、WinDriver インストール ディレクトリ (お客様が追加したファイルは除きます) も削除します。
   - すべての WinDriver ツールキットではなく、WinDriver カーネル モジュール (**windrvr6.sys**) がインストールされているターゲット PC の場合:

wdreg ユーティリティを使用して、ドライバを停止し、アンロードします。

– **windrvr6.sys** をアンインストールするには、以下のように実行します。
wdreg -inf <windrvr6.inf へのパス> uninstall

注意: このコマンドを実行する際には、**windrvr6.sys** と **windrvr6.inf** ファイルが同じディレクトリにある必要があります。

(開発用 PC では、アンインストール ユーティリティによって、適切な wdreg アンインストール コマンドが実行されます。)

注意:

– (**uninstall** ユーティリティを使用するか、または直接 **wdreg** アンインストール コマンドを実行するかに関わらず) アンインストール時に、WinDriver への開いているハンドル (例: WinDriver アプリケーションや INF ファイルを通じて WinDriver と動作するように登録された Plug-and-Play デバイス (Windows 98 / Me / 2000 / XP / Server 2003 / Vista の場合) など) があると、警告メッセージが表示されます。メッセージでは、開いているアプリケーションを閉じてから再試行するか、開いているアプリケーションをすべてアンインストールするか、開いているアプリケーションに関連するデバイスを切断してから再試行するか、アンインストールをキャンセルするかを選択できます。キャンセルを選択すると、**windrvr6.sys** カーネル ドライバはアンインストールされません。このため、WinDriver カーネル モジュール (**windrvr6.sys**) は使用されている限り、アンインストールされません。

– Debug Monitor (**WinDriver\util\wddebug_gui.exe**) を実行して、WinDriver カーネル モジュールがロードされているかチェックすることが可能です。ドライバがロードされている場合は、Debug Monitor のログにドライバと OS の情報が表示され、そうでない場合はエラー メッセージが表示されます。開発用 PC では、このユーティリティはアンインストール コマンドによって削除されます。アンインストール後に使用する場合は、アンインストールを実行する前に、**wddebug_gui.exe** のコピーを作成しておきます。

5. **windrvr6.sys** をアンロード後、以下のファイルが存在する場合は、削除します。
   - **%windir%\system32\drivers\windrvr6.sys**
   - **%windir%\inf\windrvr6.inf** (Windows 2000 / XP / Server 2003 / Vista)
   - **%windir%\inf\Jungowindrvr6.inf** (Windows 98 / Me)
   - **%windir%\system32\wdapi900.dll**
   - **%windir%\sysWOW64\wdapi900.dll** (Windows x64)
6. コンピュータを再起動します。

4.5.2 Linux から WinDriver をアンインストールするには

注意: アンインストール手順を実行するには、root でログインする必要があります。

1. WinDriver のモジュールが他のプログラムに使用されていないか確認します。

- モジュールを使用しているプログラムとモジュールのリストを表示します。
  /# /sbin/lsmod
- WinDriver のモジュールを使用しているアプリケーションをすべて閉じます。
- WinDriver のモジュールを使用しているモジュールをすべてアンロードします。
  /sbin# rmmod

2. WinDriver のモジュールをアンロードします。
   /sbin# rmmod windrvr6
3. **udev** ファイル システムをサポートする Linux カーネル 2.6.x を使用していない場合は、**/dev** ディレクトリ以下の古いデバイス ノードを削除します。
   /# rm –rf /dev/windrvr6
4. Kernel PlugIn ドライバを作成している場合は、同様に削除します。
5. /etc ディレクトリにある .windriver.rc ファイルを削除します。
   /# rm -rf /etc/.windriver.rc
6. **$HOME** にある **.windriver.rc** ファイルを削除します。
   /# rm -rf $HOME/.windriver.rc
7. DriverWizard へのシンボリック リンクを作成した場合、リンクを削除します。
   /# rm -f /usr/bin/wdwizard
8. Windriver インストレーション ディレクトリを削除します。
   /# rm -rf ~/WinDriver
9. 次の共有オブジェクトファイルが存在する場合は、削除します。
   /usr/lib/libwdapi900.so (32 ビット PowerPC または 32 ビット x86)
   /usr/lib/64/libwdapi900.so (64 ビット x86)

4.5.3 Solaris から WinDriver をアインストールするには

注意: アンインストール手順を実行するには、root でログインする必要があります。

1. 他のプログラムで WinDriver モジュールが使用されていないことを確認してください:
2. Kernel PlugIn を作成した場合、Kernel PlugIn も削除します。
   ① **# /usr/sbin/rem_drv kpname**
   ② 64 ビット プラットフォーム (64 ビット SPARC) の場合:
   # rm /kernel/drv/sparcv9/kpname
   32 ビット プラットフォーム (32 ビット x86/SPARC) の場合:
   # rm /kernel/drv/kpname
   ③ **# rm /kernel/drv/kpname.conf**
3. アンインストール スクリプトを実行します。
   ~/WinDriver# **./remove_windrvr**

4. 次のコマンドを実行します。

   - 64 ビット プラットフォーム (64 ビット SPARC) の場合:
     # rm -rf /kernel/drv/sparcv9/windrvr6 /kernel/drv/windrvr6.conf

   - 32 ビット プラットフォーム (32 ビット x86/SPARC) の場合:
     # rm -rf /kernel/drv/windrvr6 /kernel/drv/windrvr6.conf

5. **/etc** ディレクトリから **.windriver.rc** ファイルを削除します。
   # rm -rf /etc/.windriver.rc

6. **$HOME** から **.windriver.rc** ファイルを削除します。
   # rm -rf $HOME/.windriver.rc

7. DriverWizard へのシンボリック リンクを作成した場合、リンクを削除します。
   # rm -f /usr/bin/wdwizard

8. Windriver インストレーション ディレクトリを削除します。
   # rm -rf ~/WinDriver

9. 以下の共有オブジェクト ファイルが存在する場合は、削除します。
   /lib/32/libwdapi900.so (32 ビット SPARC または 32 ビット x86)
   /lib/64/libwdapi900.so (64 ビット SPARC)

第 5 章
DriverWizard

この章では WinDriver DriverWizard のハードウェア診断およびドライバ コード生成の機能について説明します。デバイス ファームウェア開発用の WinDriver USB Device DriverWizard に関する詳細は第 16 章 を参照してください。

注意: CardBus デバイスは WinDriver の PCI API を通して扱われます。そのため、この章の PCI への言及には CardBus が含まれます。

5.1 DriverWizard の概要

(WinDriver ツールキットに含まれる) DriverWizard は、デバイスドライバのコードを生成する前にそのハードウェアに実際にアクセスする GUI ベースの診断およびドライバを生成するツールです。メモリ範囲の読み込み、レジスタのトグル、割り込みの確認、デバイスの設定およびパイプ情報の表示、パイプのデータ転送、パイプのリセットなどの診断をグラフィック ユーザー インターフェイスを通して行います。デバイスが正しく動作していることを確認すると、DriverWizard は、ハードウェア リソースにアクセス可能な関数を持つドライバ ソース コードの雛形を生成します。

WinDriver がサポートする USB または PCI チップ セット (PLX 9030、9050、9052、9054、9056、9080、9656、Altera、Xilinx VirtexII、AMCC S5933、Cypress EZ-USB ファミリ、Microchip PIC18F4550、Philips PDIUSBD12、Texas Instruments TUSB3410、TUSB3210、TUSB2136、TUSB5052、Agere USS2828、Silicon Laboratories C8051F320) がベースのカードのドライバを開発する前に、特定の チップセットのサポートについて説明している第 8 章「特定の PCI および USB チップ セット サポート」を参照することを推奨します。

DriverWizard は、ハードウェアを診断および Windows 98 / Me / 2000 / XP / Server 2003 / Vista オペレーティング システムで動作するハードウェアの INF ファイルを生成するのに使用することができます (Windows NT では、ハードウェアの INF ファイルを生成しません)。上記の特定の PCI および USB チップセット [第 8 章] のをベースとしたデバイスのコードを生成する際には、DriverWizard を使用しないでください。DriverWizard は汎用的なコードを生成するので、デバイスの特定の機能に応じて DriverWizard が生成したコードを修正する必要があります。多くの PCI チップセット用に作成された、(パッケージに添付されている) ソース コード ライブラリおよびサンプル アプリケーションを使用することを推奨します。

DriverWizard を利用して開発を行うと、次の利点があります。

ハードウェアの診断: ハードウェア開発の終了後、ハードウェアを適切なスロット (PCI / CardBus / ISA / ISAPnP / EISA / CompactPCI) に挿入するか、USB デバイスの場合は、USB ポートに挿入します。DriverWizard を使ってハードウェアが正しく動作しているかどうか確認します。

コードの生成: ドライバ コードを開発する際に、DriverWizard がドライバ コードの雛形を生成します。

DriverWizard が生成するコードには、次のものが含まれます。

- デバイスのリソースの各要素にアクセスするためのライブラリ関数 (メモリ範囲、I/O 範囲、レジスタ、割り込み)。
- デバイスの診断を行う 32 ビット コンソール アプリケーション。このアプリケーションは DriverWizard が生成したライブラリ関数を利用します。この診断プログラムをデバイスドライバの雛形として使用してください。
- 開発環境にプロジェクト情報やファイルのすべてを自動的にロードするプロジェクト ワークスペース / ソリューション。WinDriver Linux、WinDriver Solaris および DriverWizard は、それぞれオペレーティング システムにあった makefile を生成します。

5.2 DriverWizard の使い方

次に DriverWizard の使い方を説明します。

1. ハードウェアをコンピュータに接続します。

   PCI カードの場合、コンピュータの適切なスロットに接続します。USB デバイスの場合、コンピュータの USB ポートに接続します。

   または

   DriverWizard を使用して、実際のデバイスをインストールすることなく、PCI デバイスのコードを生成するオプションがあります。このオプションを選択すると、DriverWizard は 仮想 PCI デバイスのコードを生成します。

   注意: 仮想 PCI デバイス オプションを選択した場合、デバイスのリソースの定義を行います。IO/メモリ範囲を指定すると、ランタイム レジスタ (オフセットは、BARs に関連しています) をさらに細かく定義することも可能です。また、ランタイム レジスタを通じて割り込みを認識するコードを生成したい場合、IRQ を指定する必要があります。IRQ ナンバーと IO/メモリ範囲のサイズは無関係です。これらは、物理デバイスをインストールしたときに、DriverWizard により自動的に認識されます。

2. ウィザードを実行してデバイスを選択します。

   ① [スタート] メニューから [プログラム] - [WinDriver] - [DriverWizard] を選択するか、デスクトップの [DriverWizard] アイコンをダブル クリックします。または **`WinDriver/wizard/`** ディレクトリから **wdwizard** ユーティリティを実行します。

   ② [New host driver project] をクリックして新しいプロジェクトを開始します。または、[Open an existing project] をクリックして保存したセッションを開きます。

図 5.1: WinDriver のプロジェクトを開く、または新規作成

③ DriverWizard が検出したデバイスの一覧から**デバイス**を選択します。PCI の場合、**Plug-and-Play カード**を選択します。Plug-and-Play カード以外の場合、**ISA** を選択します。接続していないデバイスのコードを生成する場合、**PCI: VIRTUAL DEVICE** を選択します。

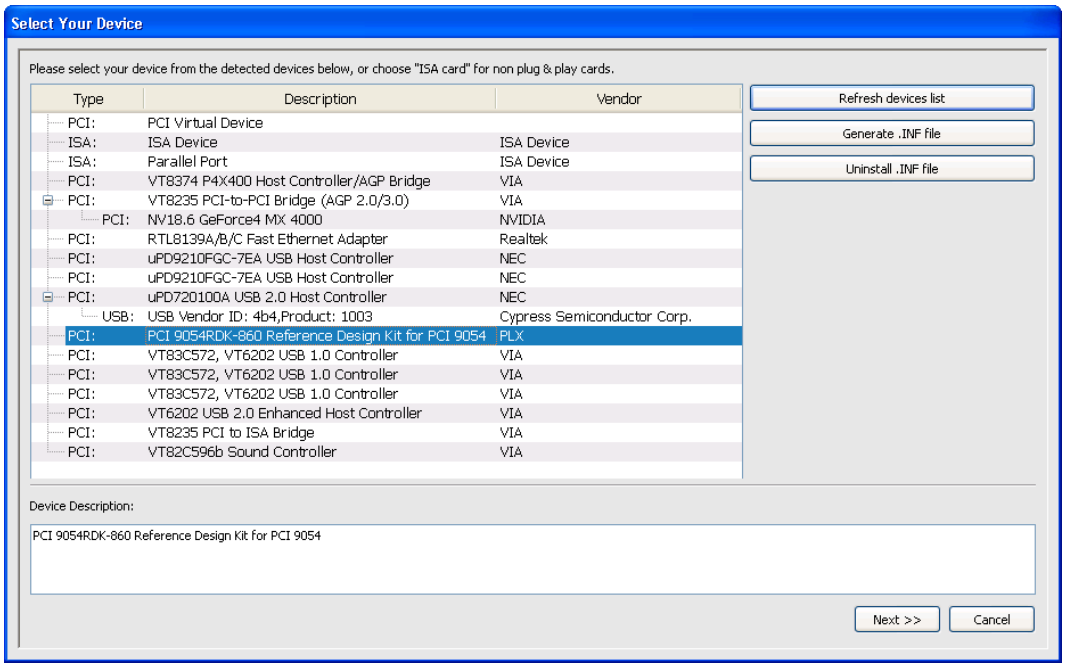


図 5.2: デバイスの選択

注意: Windows 98 の場合、一覧に対象の USB デバイスがない場合、USB デバイスを再接続して [新しいハードウェアの検出] ウィザードを表示します。次の手順でデバイスの INF が生成されるまでダイアログ ボックスを開いたままにします。

3. DriverWizard で INF ファイルを作成します。

Plug-and-Play Windows オペレーティング システム (Windows 98 / ME / 2000 / XP / Server 2003 / Vista) 用のドライバを開発する場合は、対象のデバイスの INF ファイルをインストールする必要があります。このファイルは、**windrvr6.sys** ドライバと動作するように Plug-and-Play デバイスを登録します。このステップで DriverWizard が生成したファイルは、Windows 98 / Me / 2000 / XP / Server 2003 / Vista を使用しているユーザーに配布する際に、その PC にインストールする必要があります。

また、生成した INF ファイルは、DriverWizard がデバイスの診断を行う際に使用します (たとえば、PCI / USB デバイス用のドライバがインストールされていない場合で使用します)。上記で説明したとおり、これは、WinDriver を使用して、Plug-and-Play システム (Windows 98 / Me / 2000 / XP / Server 2003 / Vista) で Plug-and-Play デバイス (PCI / PCMCIA / USB) をサポートする場合のみ必要です。INF ファイルの必要性は、セクション 15.1.1 で説明します。

INF ファイルを生成する必要がない場合 (DriverWizard を Linux で使用している場合など) は、以下のステップをスキップしてください。

以下のステップで、DriverWizard で INF ファイルを生成します。

① [Select Your Device] 画面で、[Generate .INF file] ボタンまたは [Next] ボタンを押します。

② DriverWizard は、Vendor ID、Device ID、Device Class、メーカー名およびデバイス名を含むデバイスに関する情報を問い合わせます。メーカーおよびデバイス名、およびデバイスのクラス情報を変更することができます。

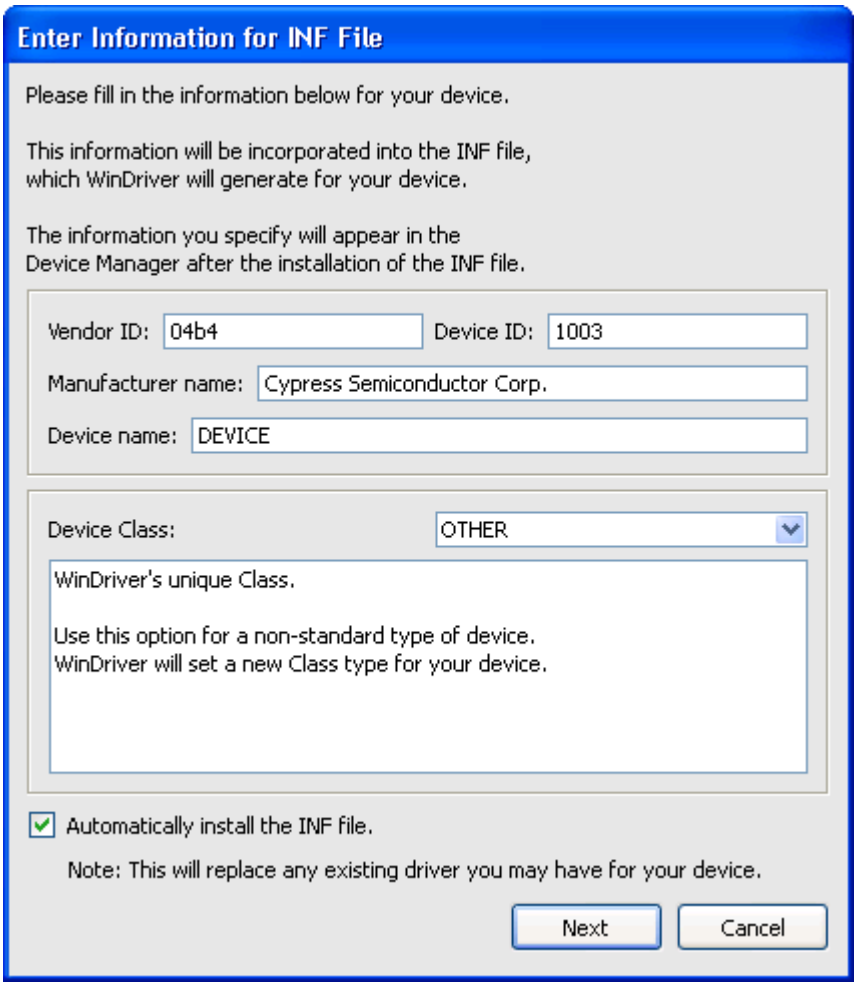


図 5.3: DriverWizard INF ファイル情報

③ マルチ インターフェイスの USB デバイスの場合、各インターフェイスに対して別々に INF ファイルを作成するか、すべてまたはマルチ インターフェイスに対して 1 つの INF ファイルを作成するかを選択することができます。

– 各インターフェイスの USB デバイスに対して別々に INF ファイルを作成する場合、[Enter Information for INF File] ダイアログで各インターフェイスに対する INF ファイルを設定します。

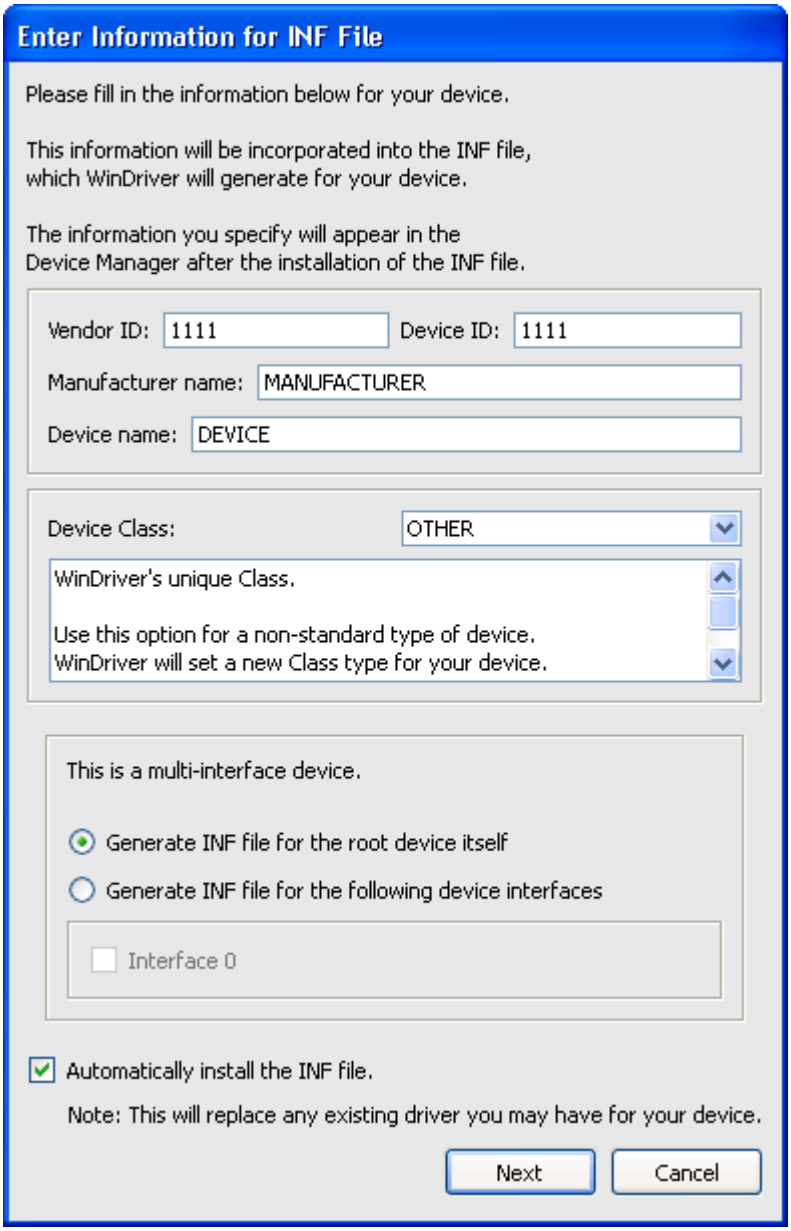


図 5.4: DriverWizard のマルチ インターフェイスの INF ファイル情報 (特定のインターフェイスをそれぞれ設定する場合)

– マルチ インターフェイスに対して 1 つの INF ファイルを作成する場合、[Enter Information for INF File] ダイアログでルート デバイス用の INF ファイルの生成、または特定のインターフェイス用の INF ファイルの生成を選択することができます。ルート デバ

イス用の INF ファイルの生成を選択すると、複数のアクティブなインターフェイスを処理できるようになります。

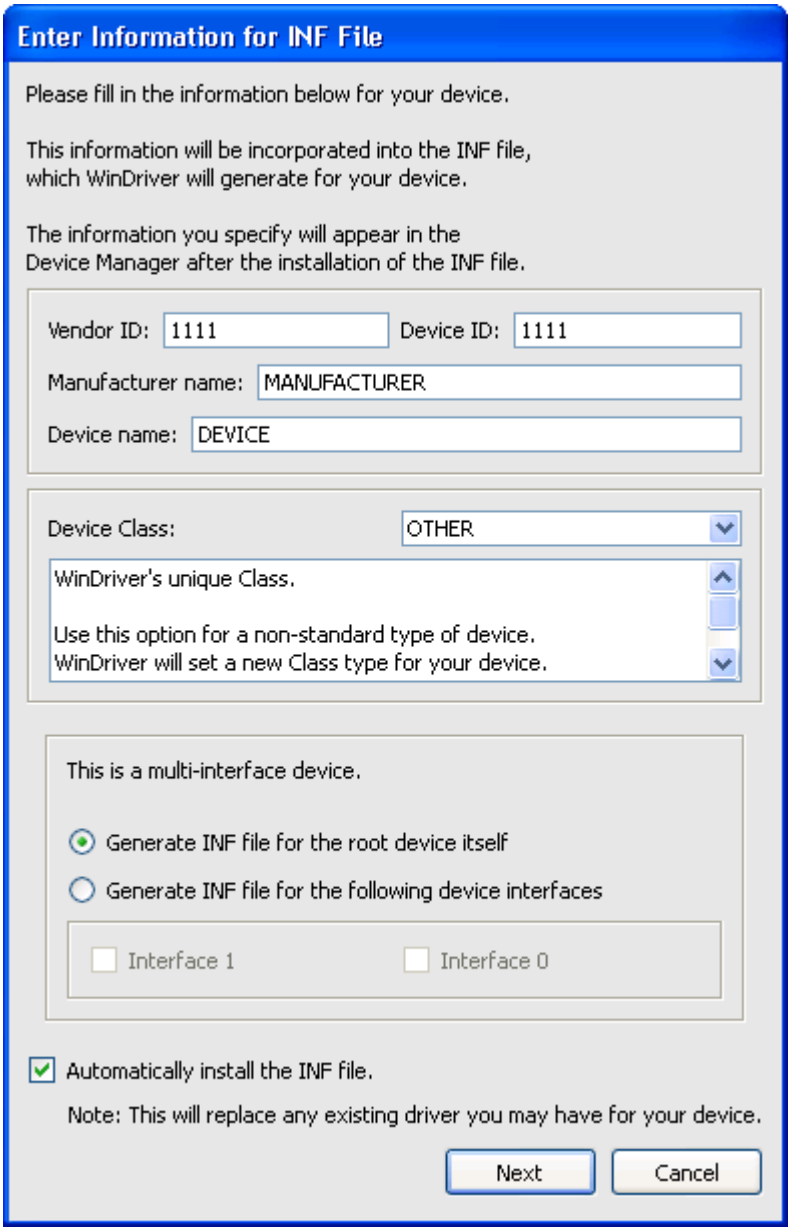


図 5.5: DriverWizard のマルチ インターフェイスの INF ファイル情報 (1 つのインターフェイスを設定する場合)

④ [Next] を押して、生成される INF ファイルを保存するディレクトリを選択します。DriverWizard は、自動的に INF ファイルを生成します。

Windows 2000 / XP / Server 2003 / Vista 上では、DriverWizard で [Automatically Install INF file] オプションをオン (USB デバイスでは、このオプションはデフォルトでオンです) にすることによって INF ファイルを自動的に DriverWizard からインストールできます。Windows 98 / Me 上では、Windows の [新規ハードウェアの追加ウィザード] または [デバイスドライバの更新ウィザード] を使用して INF ファイルを手動でインストールする必要があります。セクション 15.1 で説明します。Windows 2000 / XP / Server 2003 / Vista 上で INF ファイルの自動インス

トールに失敗した場合、DriverWizard は手動での INF ファイルのインストール方法を表示します。

⑤ INF ファイルのインストールが終了すると、[Select Your Device] 画面の一覧からデバイスを選択して開きます。

4. デバイスの INF ファイルのアンインストールします。

アンインストール オプションを使用して、対象の Plug-and-Play デバイス (PCI / PCMCIA / USB) の INF ファイルをアンインストールします。INF ファイルをアンインストールすると、そのデバイスは **windrvr6.sys** と動作するように登録されず、Windows のルート ディレクトリから INF ファイルを削除します。

INF ファイルをアンインストールする必要がない場合、このステップをスキップしてください。

① [Select Your Device] 画面で、[Uninstall .INF file] ボタンをクリックします。

② INF ファイルを選択し、削除します。

5. デバイスの診断

デバイスドライバのコードを記述する前に、ハードウェアが正常に動作することを確認します。DriverWizard を使用してハードウェアを診断します。すべてのアクティビティは DriverWizard のログに残るので、テスト結果を分析できます。

PCI デバイスの場合

① デバイスを診断します。

② PCI デバイスの I/O、メモリ範囲、レジスタ、割り込みを定義および検証します。

- DirverWizard は自動的に Plug-and-Play ハードウェア リソース (I/O 範囲、メモリ範囲、割り込み) を検出します。
  非 Plug-and-Play ハードウェアの場合、ハードウェアのリソースを手動で定義します。

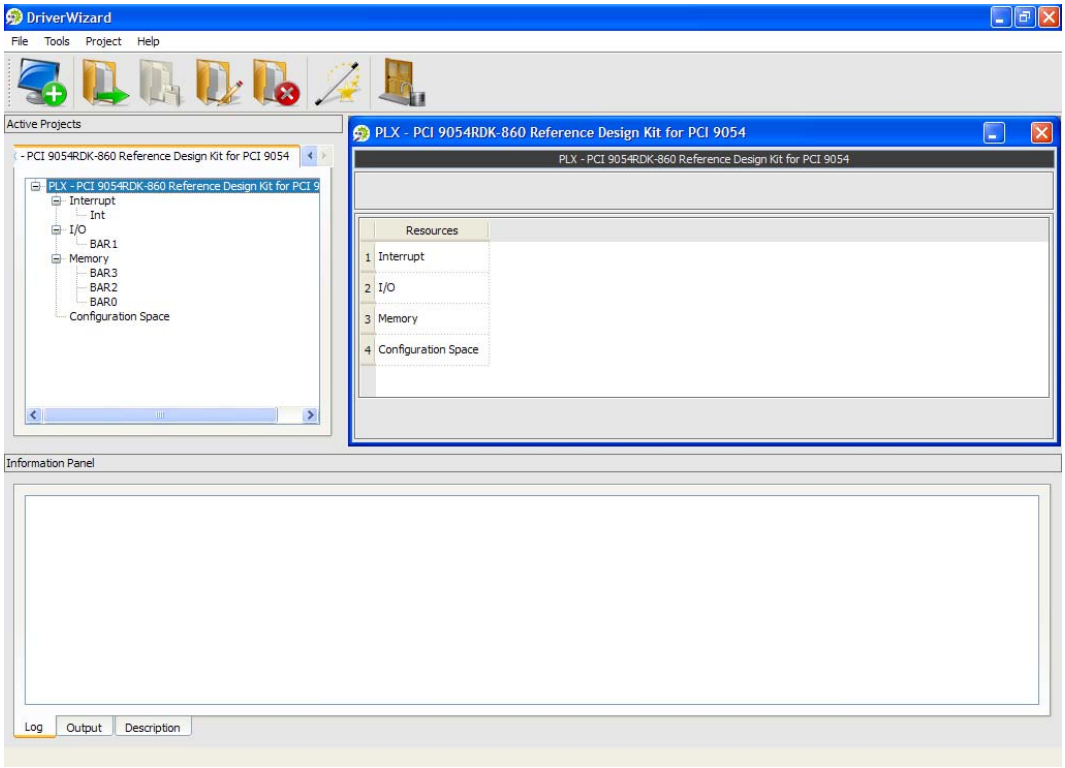


図 5.7: PCI のリソース画面

レジスタを手動で定義します。

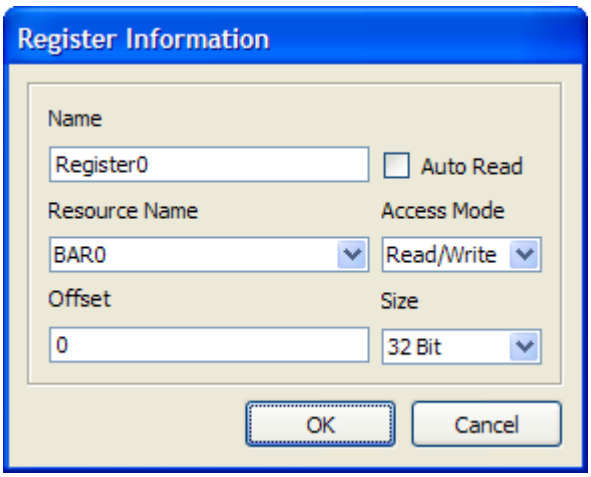


図 5.8: レジスタの定義

注意: [Register Information] ウィンドウの [Auto Read] チェック ボックスがあります。[Auto Read] チェック ボックスを ON にしたレジスタを Wizard で実行したレジスタの read (読み込み) / write (書き込み) で実行自動的に読み込みます (Wizard の [Log] ウィンドウに読み込み結果を表示します)。

– I/O ポート、メモリ スペース、定義したレジスタへの読み込みと書き込みをします。

図 5.9: メモリおよび I/O の Read / Write

注意: メモリ マップされた領域にアクセスする際には、Linux PowerPC は、PCI バス (リトル エンディアンを使用) とは対照的に、メモリ ストレージを処理するのにビッグ エンディアンを使用します。リトル / ビッグ エンディアンに関する詳細情報はセクション [9.7] を参照してください。

– ハードウェアの割り込みを 'Listen' (確認) します。

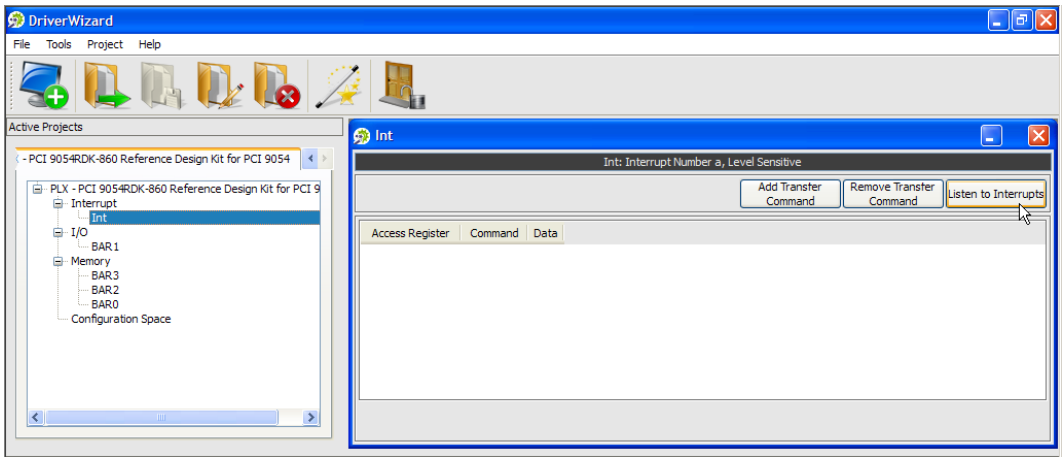


図 5.10: 割り込みの Listen (確認)

注意: PCI カードの割り込みなど、レベル センシティブな割り込みの場合、DriverWizard で割り込みの確認をする前に、DriverWizard を使用して、割り込みステータス レジスタを定義し、割り込みを認識 (解除) するための read (読み込み) / write (書き込み) コマンドを割り当てる必要があります。正確に定義しない場合には、OS がハングする可能性があります。特定の割り込みの認識情報はハードウェアの仕様となります。

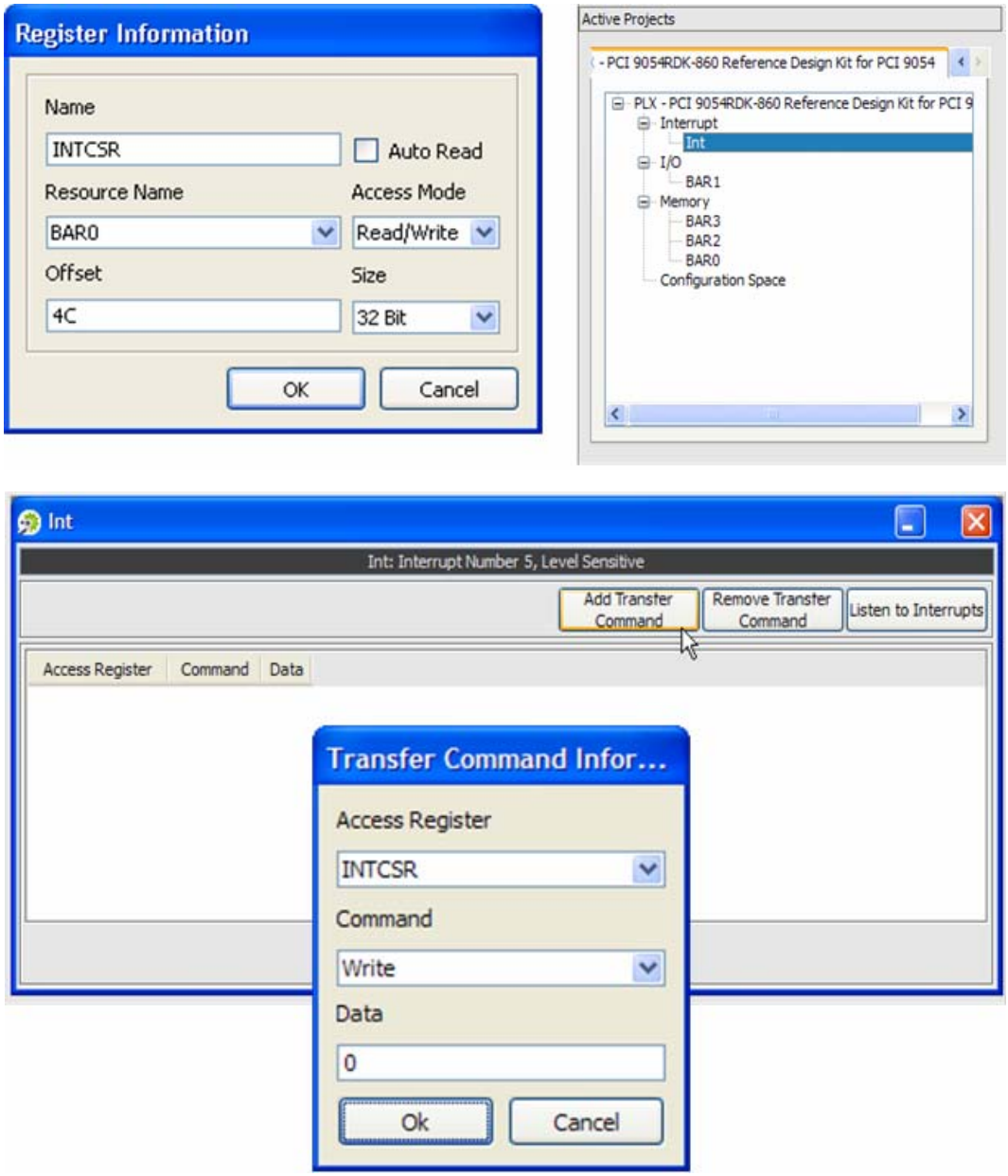


図 5.11: レベル センシティブな割り込みの転送コマンドの定義

USB デバイスの場合

① USB デバイスにおける代替設定を選択します。

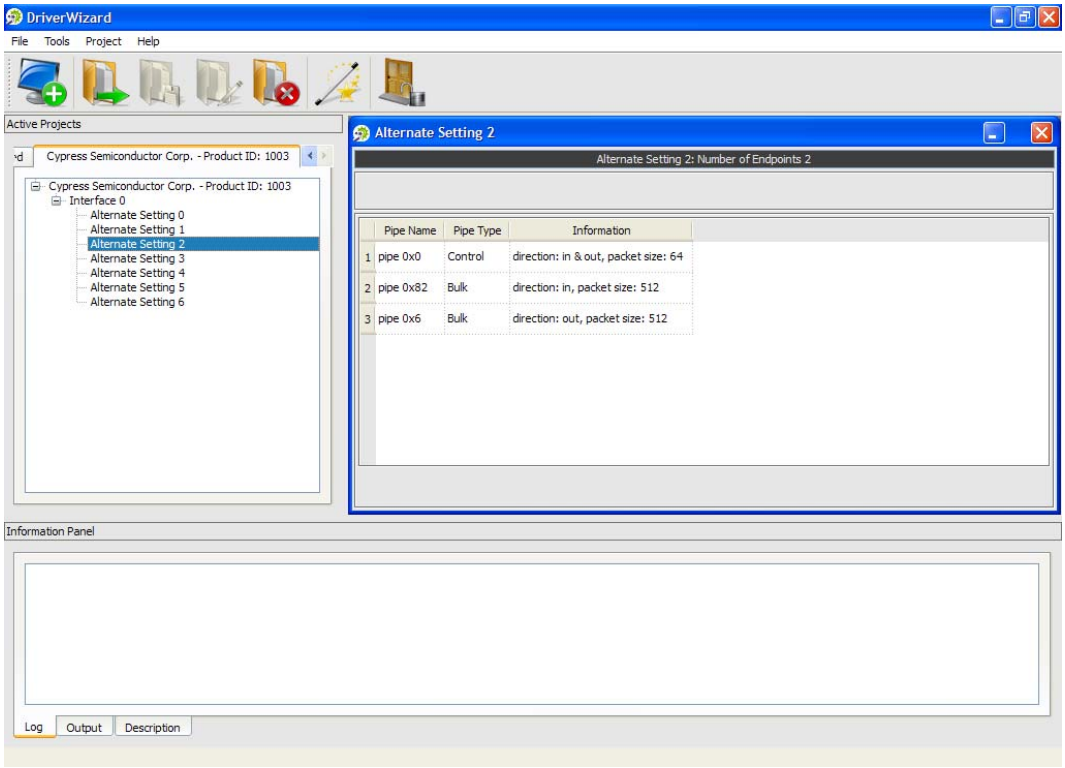


図 5.12: USB デバイスのインターフェースの選択

DriverWizard はサポートするすべてのデバイスの代替設定を読み込み表示します。表示されたリストから設定する代替設定を選択します。

注意: 設定されている代替設定が一つしかない USB デバイスの場合、DriverWizard は自動的に検出された代替設定を選択するので、[Select Device Interface] ダイアログは表示されません)。

② USB デバイス のパイプを検証します。

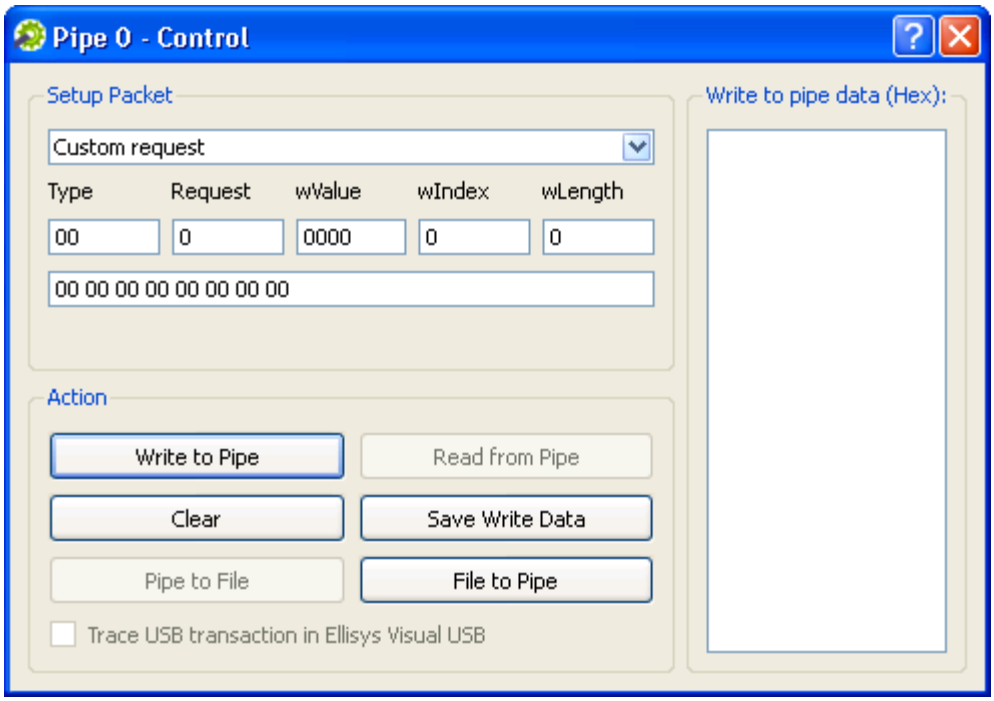


図 5.13: USB コントロール転送

DriverWizard は、選択した代替設定により検出したパイプを表示します。USB データ転送を行う場合は、次の手順に従ってください。

i. 使用するパイプを選択します。

ii. コントロール パイプ (双方向パイプ) の場合、[Read/Write to Pipe] を選択します。新しいダイアログ ボックスが表示され、標準 USB 要求 (図 5.8 を参照) を選択またはカスタム要求を入力できます。
利用可能な標準 USB 要求を選択すると、選択した要求のセットアップ パケット情報を自動的に入力し、[Request Description] ボックスに要求の詳細を表示します。
カスタム要求の場合、セットアップ パケット情報を入力し、データ (ある場合) を書き込む必要があります。セットアップ パケットのサイズは 8 バイト長にし、リトル エンディアン バイト オーダーを使用して定義します。セットアップ パケット情報は、USB 設定パラメータ (bmRequestType、bRequest、wValue、wIndex、wLength) を設定します。

注意: 標準 USB 要求の詳細は、セクション 9.3「USB コントロール転送」およびセクション 9.4「WinDriver でコントロール転送を行う」を参照してください。

iii. 入力パイプ (データをデバイスからホストに転送) の場合、[Listen to Pipe] を選択します。HID 以外のデバイスでこの操作を正しく行うには、まずデバイスがデータをホストに送るかどうかを確認する必要があります。データが送信されない場合、しばらく listening をしたあとに「Transfer Failed」と表示されます。

iv. 読み込みを中止する場合は、[Stop Listen to Pipe] をクリックします。

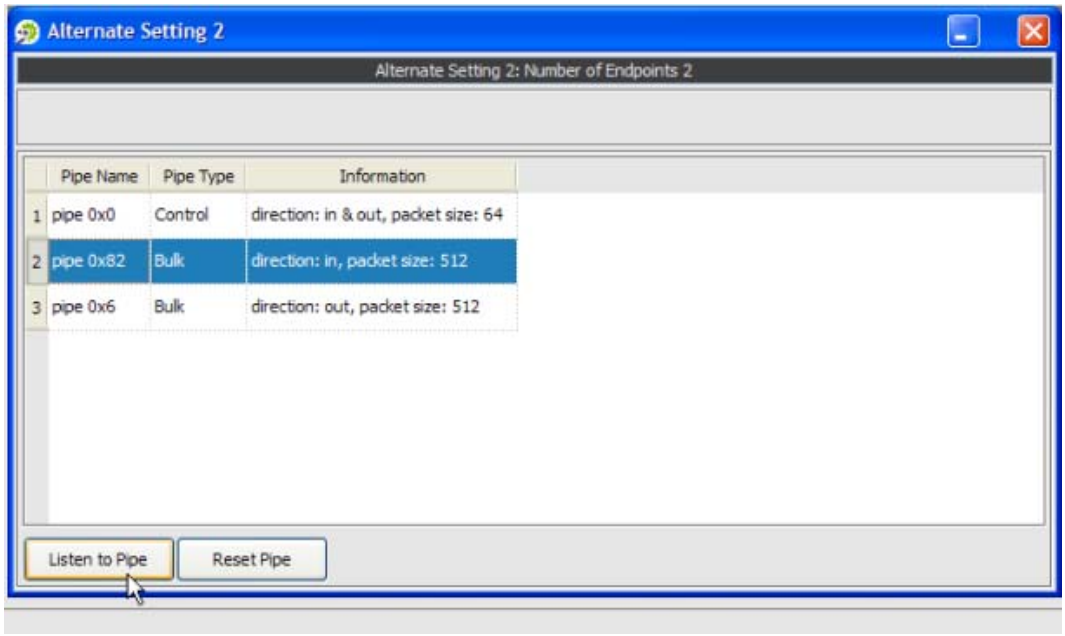


図 5.14: パイプの確認

v. 出力パイプ (データをホストからデバイスに転送) の場合、[Write to Pipe] を選択します。新しいダイアログ ボックス (図 5.9 を参照) が表示され、書き込みデータを入力します。DriverWizard はこの操作の結果を記録します。

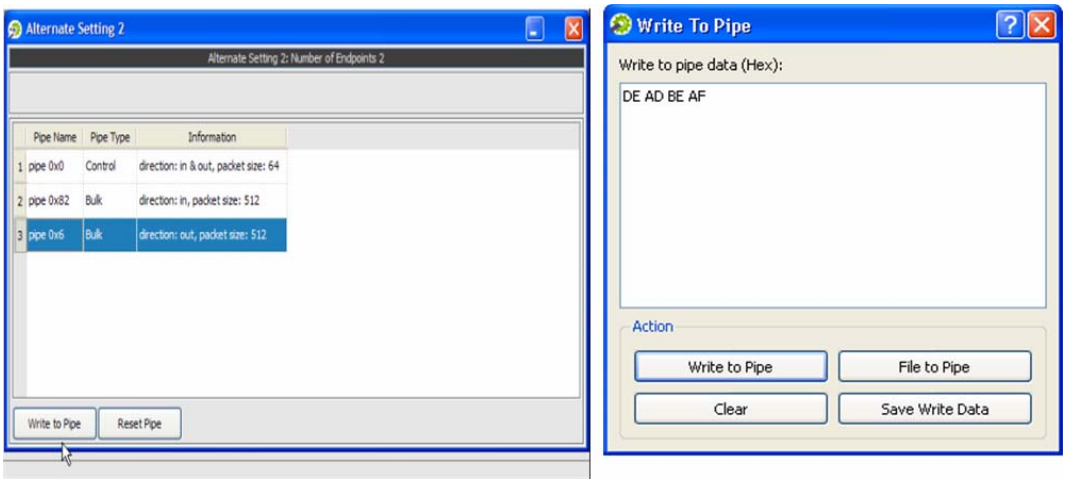


図 5.15: パイプへの書き込み

vi. 選択したパイプで [Reset Pipe] をクリックして、入力パイプと出力パイプをリセットできます。

6. 雛型となるドライバ コードを生成します。

① [Project] メニューから [Generate Code] を選択、または [Generate Code] ツールバー アイコンを選択してコードを生成します。

② [Select Code Generation Options] ダイアログボックスが表示されます。生成されるコードの言語と開発環境を選択し、[Next] を選択してコードを生成します。

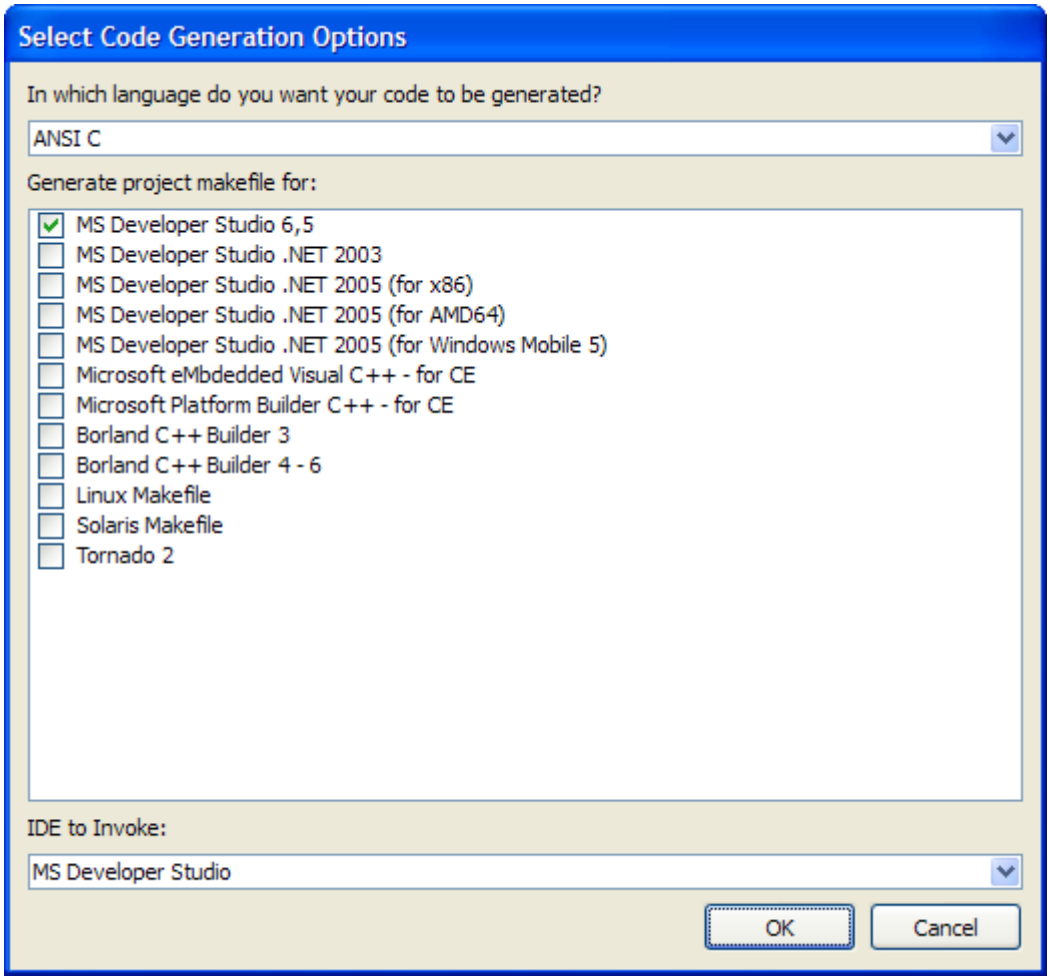


図 5.16: コード生成のオプション

③ PCI カードの場合、[Next] を選択して、Plug-and-Play イベントおよびパワーマネージメント イベントを処理するか選択し、また、KernelPlugIn コードを生成するか選択します。

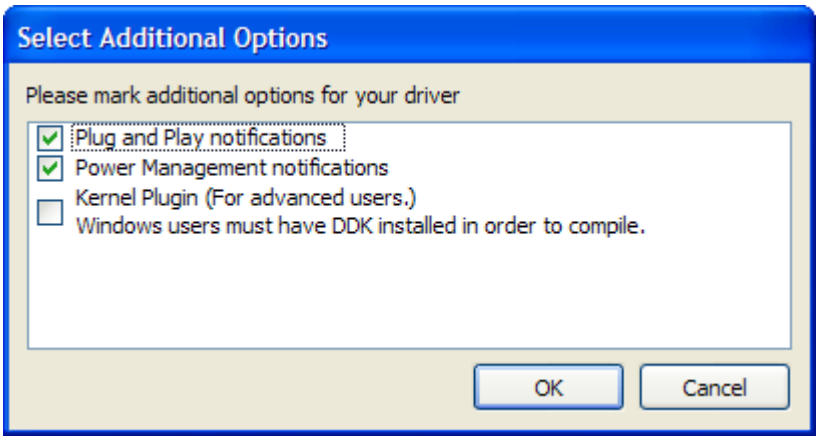


図 5.17: ドライバ オプションの選択

注意: Kernel PlugIn を使用する場合、Kernel PlugIn コードを生成する前に適切な Microsoft DDK をインストールする必要があります。

④ プロジェクトを保存します。[OK] を押して生成したドライバの開発環境を開きます。

⑤ DriverWizard を終了します。

7. 生成されたコードをコンパイルし、実行します。

   ○ このコードをデバイスドライバの雛形として使用します。ドライバの特有の機能を実行する場合には、必要に応じて修正します。

   ○ DriverWizard が生成したソース コードは 32 ビット コンパイラでコンパイル可能で、コードの修正をせずに対応するすべてのプラットフォームで動作します。

5.3 DriverWizard ノート

5.3.1 リソースの共有

複数のドライバが同じリソースを共有する場合は、そのリソースを「shared」として定義する必要があります。リソースを shared として定義するには:

1. リソースを選択します。
2. リソースを右クリックします。
3. メニューから [Share] を選択します。

注意: 新しく定義した割り込みのデフォルトは Shared です。共有しない割り込みとして定義する場合は、次に説明する手順にしたがってください。

5.3.2 リソースの無効化

カードの診断中にリソースを無効に設定することによって、そのリソースに対するコードを自動生成しないように設定できます。

リソースを無効化するには:

1. リソースを選択します。
2. リソースを右クリックします。
3. メニューから [Disable] を選択します。

5.3.3 WinDriver API 呼び出しのログ

DriverWizard には、API 呼び出しの入力および出力パラメータを含む、すべての WinDriver API 呼び出しのログを採取するオプションがあります。[Tools] メニューの [Log API calls] オプションを選択するか、DriverWizard のツールバーの [Log API calls] アイコンをクリックしてこのオプションを選択します。

5.3.4 DriverWizard のログ

新しいプロジェクトを開く際に、[Device Resources] ダイアログと一緒に表示されるブランク ウィンドウに DriverWizard のログが記録されます。ログは、診断中に行ったすべての入出力を記録するため、あとからデバイスのパフォーマンスを解析できます。あとで参照できるようにログを保存できます。プロジェクトを保存すると、このログも保存されます。ログはプロジェクトごとに作成されます。

5.3.5 自動コード生成

デバイスの診断が終了し、デバイスが仕様どおりに動作することを確認したら、ドライバのコードを生成します。

5.3.5.1 コードを生成する

DriverWizard の [Generate Code] ツールバー アイコンまたは [Project] メニューから [Generate Code] のいずれかを選択してコードを生成します。DriverWizard はドライバのソース コードを生成し、プロジェクト ファイル (**xxx.wdp**、xxx はプロジェクト名) と同じディレクトリに作成します。DriverWizard が生成するディレクトリに [Generate Code] ダイアログ ボックスで選択した開発環境とオペレーティング システム用にファイルを保存します。

5.3.5.2 PCI / PCMCIA / ISA 用の C コードを生成する

DriverWizard により作成された API 用の型の定義および関数の宣言が含まれた xxxlib.h ファイル、および生成されたデバイスを特定した API が適応された **xxx_lib.c** ソース ファイルがソース コード ディレクトリに新規に作成されます。

さらに、`main()` 関数を含む **xxx_diag.c** ソース ファイルも作成されます。この関数はデバイスと通信するために DriverWizard で生成された API を利用するサンプル診断アプリケーションを実行します。

DriverWizard が生成するコードには、次のものが含まれます (“**xxx**” は DriverWizard のプロジェクト名を表します)。

- カードのリソースにアクセスするためのライブラリ関数 (メモリ範囲、I/O 範囲、レジスタ、割り込み)。

  xxx_lib.c - WinDriver Card (WDC) API を利用して、**xxx_lib.h** の中にあるハードウェア特有の API の実行します。

  xxx_lib.h - **xxx_lib.c** ソース ファイルで実装される API 用の型の定義および関数の宣言を含んでいます。DriverWizard によって生成される API を使用するために、このファイルをソース ファイルに含める必要があります。

- **xxx_lib.h** で宣言される DriverWizard で生成された API がデバイスと通信するため使用される診断プログラム

  xxx_diag.c - 生成された診断コンソール アプリケーションのソース コード。この診断プログラムをデバイス ドライバの雛形として使用してください。

- 作成されたすべてのファイルのリストは **xxx_files.txt** に作成されます。

コードの生成が終了したら Win32 コンパイラを使ってコンパイルしてください。

`main()` 関数を変更してドライバに必要な機能を追加できます。

5.3.5.3 USB 用の C コードを生成する

ソース コード ディレクトリに **xxx_diag.c** ソース ファイルが新規に作成されます (**xxx** は DriverWizard プロジェクトで選択した名前です)。このファイルは、USB デバイスの場所を見つけて通信を行う WinDriver の USB API の使用方法を示す USB アプリケーション診断を実行します。この診断には、Plug-and-Play イベント (デバイスの取り付け/取り外しなど) の検出、パイプの読み書き転送の実行、パイプのリセット、デバイスの動的な代替設定の変更が含まれています。

生成されたアプリケーションは複数の同一 USB デバイスの処理をサポートします。

5.3.5.4 Visual Basic または Delphi コードの作成

DriverWizard が生成する Visual Basic および Delphi コードは、セクション 5.3.5 で説明した C コードに似た機能を提供します。

生成される Delphi コードは (C コードのように) コンソール アプリケーションを実装し、Visual Basic コードは GUI アプリケーションを実装します。

5.3.5.5 C# または Visual Basic コードの作成

DriverWizard が生成する C# および Visual Basic .NET コードは、セクション 5.3.5.2 で説明した C コードに似た機能を、GUI .NET プログラムから提供します。

5.3.6 生成されたコードをコンパイルする

5.3.6.1 Windows と Windows CE のコンパイル

上記で説明したとおり、Windows では、サポートされている IDE (統合開発環境) のプロジェクト、ワークスペース/ソリューション ファイルを生成します。サポートされている IDE は、MSDEV / Visual C++ 5 / 6 /、MSDEV .NET 2003 / 2005、Borland C++ Builder、Visual Basic 6.0、Borland Delphi、MS eMbedded Visual C++、MS Platform Builder です。選択した IDE がウィザードから自動的に起動し、すぐにコードをコンパイルおよび実行できます。

また、他の IDE で生成されたコードを、生成されたコード言語でビルドすることもできます。選択した IDE 用の新しいプロジェクトファイルを作成し、生成されたソース ファイルをプロジェクトに追加して、コードをコンパイルおよび実行します。

注意:

- **`Windows 2000 / XP / Server 2003 / Vista`** では、生成された IDE ファイルは、**`x86`** ディレクトリ (32 ビット プロジェクトの場合) または **`amd64`** ディレクトリ (64 ビット プロジェクトの場合) に保存されます。
- Windows CE では、生成された **`Windows Mobile`** のコードは、Windows Mobile 5.0 / 6.0 ARMV4I SDK をターゲットとします。

5.3.6.2 Linux と Solaris の場合

DriverWizard が作成した makfile を使用して、任意のコンパイラ (GCC を推奨) で生成されたコードをビルドします。

5.3.7 Bus Analyzer の統合 - Ellisys Visual USB

DriverWizard は、Windows XP 以降 (32 ビットのみ) で Ellisys Explorer 200 USB Analyzer をネイティブにサポートしています。これにより、次のことが実現可能です。

- DriverWizard から直接 USB トラフィックの収集を開始
- 離散コントロール転送の収集

USB トラフィックの収集:

1. [Tools] - [Start USB Analyzer Capture] を選択して、USB データの収集を開始します。

2. データ収集を終了するには、[Tools] - [Stop USB Analyzer Capture] を選択します。DriverWizard により収集結果が保存された場所を示すダイアログ ボックスが表示されます。[Yes] をクリックして、収集したデータで Ellisys Visual Analyzer を実行します。

離散コントロール転送を収集するには、コントロール転送のダイアログ ボックスで [Trace USB transaction in Ellisys Visual USB] チェック ボックスをオンにします。

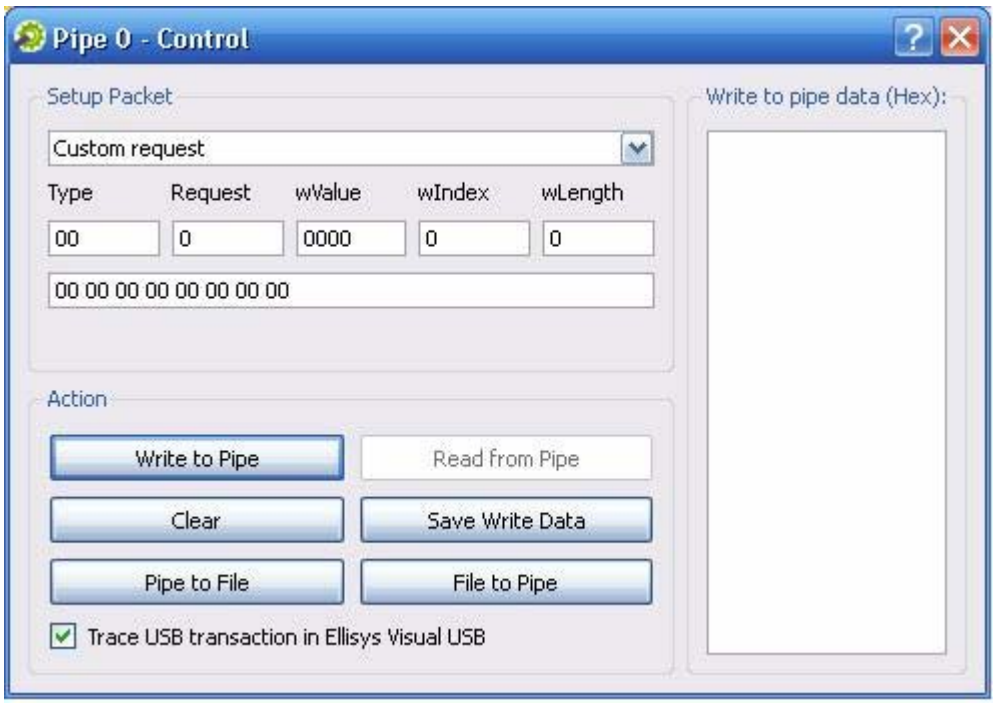


図 5.18: Ellisys Visual USB の統合

第 6 章
ドライバの作成

この章では、WinDriver を使用した開発サイクルを紹介します。

注意: デバイスが WinDriver が拡張サポートする次のチップセット (PLX 9030、9050、9052、9054、9056、9080、9656、Altera、Xilinx VirtexII、AMCC S5933、Cypress EZ-USB ファミリ、Microchip PIC18F4550、Philips PDIUSBD12、Texas Instruments TUSB3410、TUSB3210、TUSB2136、TUSB5052、Agere USS2828、Silicon Laboratories C8051F320) を使用している場合、まず次の概要を参照してください。次に第 8 章 をお読みください。

6.1 WinDriver でデバイスドライバを開発するには

- DriverWizard を使ってカードの診断を行います。カードがサポートする IO、メモリ範囲、レジスタおよび USB デバイスのパイプを読み書き、PCI 設定レジスタ情報の表示、カードのレジスタのおよびレジスタの読み書きの定義、割り込みを聞きます。デバイスが期待通りの動作をするかどうかを確認します。

- DriverWizard を使ってデバイスドライバの雛形となるコードを C、C#、Visual Basic .NET、Delphi または Visual Basic で作成します 。DriverWizard についての詳細は、第 5 章 の「DriverWizard」を参照してください。

- サポートしているチップセット (PLX 9030、9050、9052、9054、9056、9080、9656、Altera、Xilinx VirtexII、AMCC S5933、Cypress EZ-USB ファミリ、Microchip PIC18F4550、Philips PDIUSBD12、Texas Instruments TUSB3410、TUSB3210、TUSB2136、TUSB5052、Agere USS2828、Silicon Laboratories C8051F320) を USB チップセットまたは PCI チップセットに使用する場合は、使用するチップの特定のサンプル コードをドライバ コードの雛形として使用することを推奨します。WinDriver がサポートする特定の PCI および USB チップセットに関する詳細は、第 8 章 の「特定 PCI チップ セット サポート」を参照してください。

- C / .NET / Delphi / Visual Basic コンパイラ (MSDEV、Visual C/C++、MSDEV .NET、Borland C++ Builder、Borland Delphi、Visual Basic 6.0、MS eMbedded Visual C++、MS Platform Builder C++、GCC など) で必要な雛型ドライバをコンパイルします。

- Linux および Solaris の場合、GCC を使用してコードをビルドします。

- これでユーザー モードドライバの作成は完了です。作成したドライバのパフォーマンスを向上させるには、第 10 章 の「パフォーマンスの向上」を参照してください。

WinDriver の PCI/ISA/CardBus API および USB API に関する詳細は付録、DriverWizard を自動に処理しない方法、転送のコントロールの実装方法については第 9 章 を参照してください。

6.2 DriverWizard を使わずにドライバを記述するには

DriverWizard を使用せずに直接ドライバを記述する場合、以下のステップに従って新しいドライバ プロジェクトを作成するか、または、記述するドライバに最も近いサンプルに修正を加えてください。

6.2.1 必要な WinDriver ファイルのインクルード

PCI / ISA の場合

1. 関連した WinDriver ヘッダー ファイルをプロジェクトにインクルードします (すべてのヘッダー ファイルは **WinDriver/include/** ディレクトリに保存されています)。すべての WinDriver プロジェクトには **windrvr.h** ヘッダー ファイルが必要です。

   PCI / ISA の場合

   WDC_xxx API を使用する場合、**wdc_lib.h** および **wdc_defs.** ヘッダー ファイル (これらのファイルは既に **windrvr.h** をインクルードしています) をインクルードします。

   USB の場合

   WDU_xxx WinDriver USB API を使用する場合、**wdu_lib.h** ヘッダー ファイル (このファイルは既に **windrvr.h** をインクルードしています) をインクルードします。

   コードから使用する API を提供するその他のヘッダー ファイルをインクルードします (たとえば、**WinDriver/samples/shared/** ディレクトリからのファイルは便利な診断関数があります)。

2. ソース コードから関連したヘッダー ファイルをインクルードします。

   PCI / ISA の場合

   たとえば、**windrvr.h** ヘッダー ファイルからAPI を使用するには、コードに次の行を追加します。

   ```
   #include "windrvr.h"
   ```

   USB の場合

   たとえば、**wdu_lib.h** ヘッダー ファイルから USB API を使用するには、コードに次の行を追加します。

   ```
   #include "wdu_lib.h"
   ```

3. コードを **wdapi900** ライブラリまたは共有オブジェクトにリンクします。

   - Windows 98 / Me / 2000 /XP / Server 2003 / Vista の場合: **WinDriver\lib\<CPU>\wdapi900.lib** または **wdapi900_borland.lib** (Borland C++ Builder の場合) にリンクします。**CPU** ディレクトリは、**x86** (32 ビット プラットフォーム対応 32 ビット バイナリ)、**am64** (64 ビット プラットフォーム対応 64 ビット バイナリ)、または **am64\x86** (64 ビット プラットフォーム対応 32 ビット バイナリ)。
   - Windows CE の場合: **WinDriver\lib WINCE\<CPU>\wdapi900.lib**
   - Linux または Solaris の場合: **WinDriver/lib/libwdapi900.so**

   ライブラリにリンクする代わりに、**WinDriver/src/wdapi/** ディレクトリから、ライブラリのソース ファイルをインクルードすることもできます。

注意: **wdapi900** ライブラリまたは共有オブジェクトをリンクする際、ドライバと共に **wdapi900** の DLL または共有オブジェクトを配布する必要があります。Windows では、**WinDriver\redist** ディレクトリにある **wdapi900.dll** または **wdapi900_32.dll** (64 ビット プラットフォームをターゲットとする 32 ビット アプリケーションの場合) を配布します。Linux および Solaris では、**WinDriver/lib/libwdapi900.so** を配布します。詳細は第 14 章 を参照してください)。

4. コードで使用する API を実装する その他の WinDriver ソース ファイルを追加します (たとえば、**WinDriver/samples/shared/** ディレクトリからのファイル)。

6.2.2 コードの作成: PCI / ISA ドライバの場合

このセクションでは、WDC_xxx API を使用した際の呼び出し順序を説明します。

1. WDC_DriverOpen() を呼び出し、WinDriver および WDC ライブラリのハンドルを開きます。ロードしたドライバとドライバ ソース ファイルのバージョンを比較し、(登録ユーザー用の) WinDriver ライセンスに登録します。

2. PCI/CardBus/PCMCIA デバイスでは、`WDC_PciScanDevices()` / `WDC_PcmciaScanDevices()` を呼び出して、PCI/PCMCIA バスをスキャンしデバイスの場所を検出します。

3. PCI/CardBus/PCMCIA デバイスでは、`WDC_PciGetDeviceInfo()` / `WDC_PcmciaGetDeviceInfo()` を呼び出して、選択したデバイスのリソース情報を取得します。
   ISA デバイスでは、`WD_CARD` 内でリソース自身を定義します。

4. デバイスに適切な関数 (`WDC_PciDeviceOpen()` / `WDC_PcmciaDeviceOpen()` / `WDC_IsaDeviceOpen()`) を呼び出し、デバイスのリソース情報の関数を渡します。これらの関数はハンドルからWDC_xxx API を使用するデバイスと通信するのに使用するデバイスへ返ります。

5. WDC_xxx API (詳細は付録を参照してください) を使用するデバイスと通信します。
   割り込みを有効にするには、`WDC_IntEnable()` を呼び出します。
   Plug-and-Play および パワー マネージメント イベント用の通知受け取りを登録するには、`WDC_EventRegister()` を呼び出します。

6. 終了する場合、`WDC_IntDisable()` を呼び出し、割り込み処理を無効にします (有効だった場合)。`WDC_EventRegister()` を呼び出し、Plug-and-Play および パワー マネージメント イベント処理の登録を取り消します (登録されていた場合)。最後にデバイスに適切な関数 (`WDC_PciDeviceClose()` / `WDC_PcmciaDeviceClose()` / `WDC_IsaDeviceClose()`) を呼び出し、デバイスのハンドルと閉じます。

7. `WDC_DriverClose ()` を呼び出し、WinDriver および WDC ライブラリのハンドルを閉じます。

6.2.3 コードの作成: USB ドライバの場合

1. 対象の USB デバイスに対し WinDriver を初期化するプログラムの初めに WDU_Init() を呼び、device-attach callback を待機します。各デバイス情報を attach callback で取得します。

2. attach callback を受信すると、WDU_Transfer() 関数の一つを使用して、データの送受信ができます。

3. 終了する場合は、WDU_Uninit() を呼んで、デバイスから登録解除を行います。

6.3 Windows CE で開発を行うには

Windows CE でドライバの開発を行うには、初めにデバイスを WinDriver で動くように登録する必要があります。これはWindows (Windows 98 / Me / 2000 / XP / Server 2003 / Vista) の Plug-and-Play 用のドライバを開発する際にデバイス用の INF ファイルをインストールするのに似ています。INF ファイルに関する詳細はセクション [15.1] を参照してください。

PCI の場合

次のレジストリの例は、PCI バスドライバへデバイスを登録する方法を示しています (**platform.reg**ファイルへ追加することもできます)。

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\PCI\Template\MyCard]
"Class"=dword:04
"SubClass"=dword:01
"ProgIF"=dword:00
"VendorID"=multi_sz:"1234","1234"
"DeviceID"=multi_sz:"1111","2222"
```

詳細は MSDN ライブラリの PCI バスドライバ レジストリの設定セクションを参照してください。

USB の場合

WinDriver で動作するように USB デバイスを登録するには:

- Windows CE システムにデバイスを差し込む前に **WDU_Init()** を呼び出します。

  または

- レジストリに次のように追加します (**platform.reg**ファイルへ追加することもできます)。

  ```
  [HKEY_LOCAL_MACHINE\DRIVERS\USB\LoadClients\<ID>\Default\Default\WDR]: "DLL"="windrvr6.dll"
  ```

  <ID> は アンダースコア (_) で区切られた vendor ID および product ID で構成されています (例: <MY VENDOR ID>_<MY PRODUCT ID>)。

  このキーへデバイス特有の情報を入力します。このキーはデバイスを Windows CE Plug-and-Play (USB ドライバ) として登録し、起動時にデバイスを認識します。WDU_Inti() を呼び出した後、レジストリを参照することができます。その後このキーは存続します。これによりデバイスは Windows CE で認識されます。デバイスが永続的なレジストリの場合、この追加情報は削除するまで残ります。

  詳細は MSDN ライブラリの USB ドライバ レジストリの設定セクションを参照してください。

6.4 Visual Basic および Delphi で開発を行うには

Visual Basic および Delphi でドライバを開発するには、WinDriver API を使用します。

6.4.1 DriverWizard を使用する

DriverWizard を使用して、ハードウェアを診断したり、コーディングを始める前にハードウェアが正常に動作しているか確認します。次に、ウィザードを使用して、Delphi や Visual Basic を含むさまざまな言語でソースコードを自動的に生成します。詳細は、第 5 章 およびセクション 6.4.4 を参照してください。

6.4.2 サンプル

Delphi または Visual Basic で WinDriver API を使用して記述したサンプルが以下にあります。

1. **`WinDriver\delphi\samples`**
2. **`WinDriver\vb\samples`**

ドライバ開発の第一歩として、これらのサンプルを使用します。

6.4.3 Kernel PlugIn

Kernel PlugIn を生成するのに Delphi および Visual Basic は使用できません。ユーザー モードで Delphi または VB で WinDriver を使用している開発者は、Kernel PlugIn を記述するときは、C を使用する必要があります。

6.4.4 ドライバを生成するには

Visual Basic での開発方法は、DriverWizard の自動コード生成機能を使用する C での開発方法と同じです。

以下の手順に従ってください。

- DriverWizard を使用して、ハードウェアの診断を行います。
- ハードウェアが正常に動作しているかを確認します。
- ドライバ コードを生成します。
- ドライバをアプリケーションに統合します。
- WinDriver のサンプルを WinDriver API を取得およびドライバ コードの雛型として使用できます。

第 7 章
デバッグ

この章では、ハードウェアにアクセスするアプリケーションをデバッグ方法について説明します。

7.1 ユーザー モード デバッグ

- WinDriver はユーザー モードからアクセスされるので、デバッグには標準のデバッグ ソフトウェアを使用してください。
- Debug Monitor [7.2] は、WinDriver のカーネル モジュール および ユーザー モード API からのデバッグ メッセージを記録します。WinDriver API を使用して、デバッグ メッセージを Debug Monitor に送信することもできます。
- デバッグ モニタ [7.2] が作動している場合、WinDriver のカーネル モジュールは、WinDriver の API (WD_Transfer など) を使用している場合のメモリ範囲の有効性を確認します。すなわち、メモリからの読み出し、またはメモリへの書き込みがカードへ定義される範囲内にあるかどうかを確認します。
- デバッグ処理でメモリとレジスタの値をチェックするには、DriverWizard を使用します。

7.2 Debug Monitor

Debug Monitor は、WinDriver カーネル (**`windrvr6.sys/.dll/.o/.ko`**) が処理するすべてのアクティビティを監視する、強力なツールです。このツールを使用して、各コマンドがどのようにカーネルに送られて処理されているのかを監視できます。また、`WD_DebugAdd()` や高水準の `PrintDbgMessage()` を使用して、デバッグ メッセージを Debug Monitor に出力することができます。

Debug Monitor には、グラフィック モードとコンソール モードの 2 つがあります。次に各モードでの Debug Monitor の操作方法を説明します。

7.2.1 グラフィック モードで Debug Monitor を使用するには

Windows 98 / Me / 2000 / XP / Server 2003 / Vista、Linux および Solaris で Debug Monitor のグラフィック モード (GUI) を使用できます。Windows 2000 / XP / Server 2003 / Vista で Windows CE エミュレータ上で動作する Windows CE ドライバ コードも Debug Monitor でデバッグ可能です。Windows CE をターゲットにしている場合、または VxWorks では、コンソール モードの Debug Monitor を使用してください [7.2.2]。

1. Debug Monitor を実行する
   - DebugMinitor ("**`wddebug_gui.exe`**") は、**`WinDriver/util/`** ディレクトリにあります。
   - Debug Monitor は、DriverWizard の [Tool] メニューから起動することができます。

○ [スタート] メニューから [プログラム] - [WinDriver] - [Monitor Debug Messages] を選択して、Debug Monitor を起動できます。

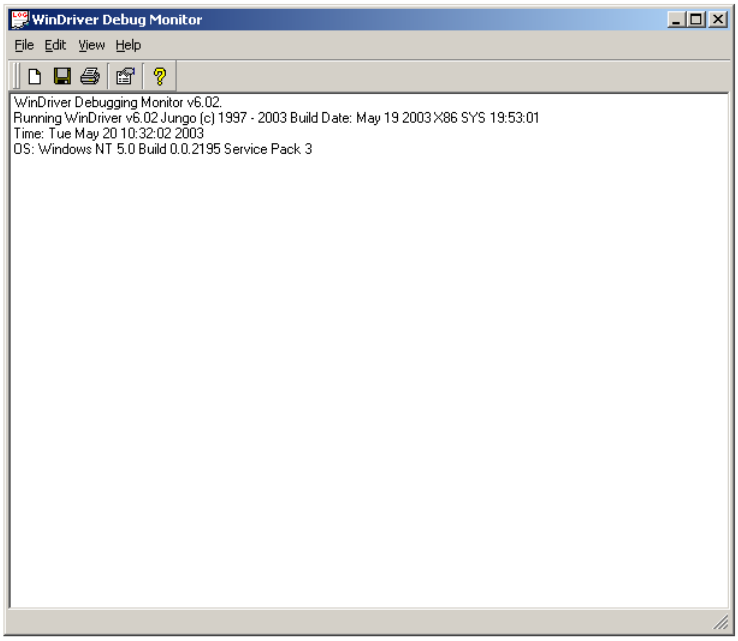


図 7.1: Debug Monitor の起動

2. [View] - [Debug Options] メニューを選択するか、ツールバーにある [Debug Options] ボタンをクリックして、[Debug Options] ダイアログ ボックスを表示し、Debug Monitor のステータス、トレース レベル、およびデバッグするセクションを設定します。

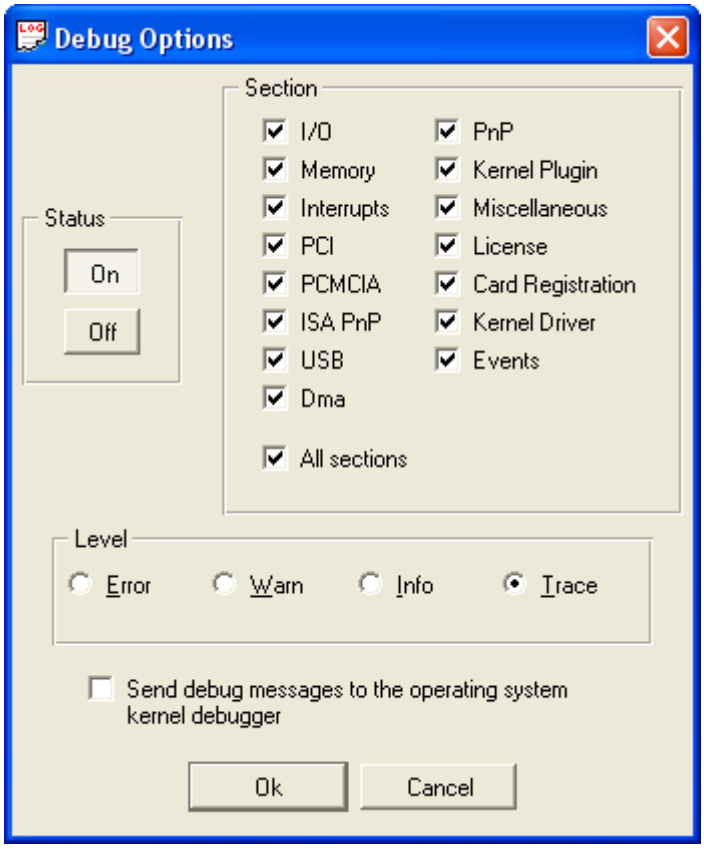


図 7.2: Debug Options の設定

- **Status** - トレースを [ON] または [OFF] にセットします。
- **Section** - 監視する WinDriver API の一部を選択します。PCI カードの割り込み処理に問題がある場合は、[Interrupts] と [PCI] チェック ボックスを選択してください。
  USB デバイスのドライバをデバッグする場合は、[USB] ボックスを選択してください。

ヒント: 監視するオプションを選択するときは慎重に行ってください。必要以上にオプションを選択すると、情報が多すぎて、問題を見つけるのが困難になります。

- **Level** - 定義されたリソースから調査するメッセージ レベルを選択します。

  Error を選択すると、トレースは最小限に表示されます。

  Trace を選択すると、WinDriver カーネルのすべての操作が表示されます。

- OS カーネル デバッガにデバッグ メッセージを送る場合、[Send debug messages to the operating system kernel debugger] チェックボックスを選択します。

- このオプションは、WinDrinver のカーネル モジュールから受け取った全てのデバッグ情報を外部のカーネル デバッガへ送れるようにします。アプリケーションを実行して問題を再現し、外部カーネル デバッガのログでデバッグ情報を参照します。Windows ユーザーの場合は、Microsoft の WinDbg ツールを使用することができます。WinDbg ツールは、Microsoft の Web サイトより Microsoft のドライバ開発キット (DDK) と共に提供されています。(Microsoft デバッグ ツール ページを参照してください)。

3. トレースする部分とレベルを決定したら [OK] をクリックして [Debug Options] ダイアログボックスを閉じます。

4. デバッグするプログラムを実行 (ステップ実行など) します。

5. モニタに表示されるエラーや予期しないメッセージを監視してください。

7.2.2 コンソール モード Debug Monitor を使用するには

コンソール モード Debug Monitor (**wddebug**) は、サポートしているすべてのオペレーティング システムで利用可能です。

7.2.2.1 Windows、Windows CE、Linux または Solaris で wddebug を使用するには

Windows 98 / Me / 2000 / XP / Server 2003 / Vista、CE、Linux または Solaris で **wddebug** (コンソール モード Debug Monitor) を使用するには、**WinDriver/util/** ディレクトリから次のコマンドを実行します。
wddebug

注意: Windows CE では、ターゲット上で Windows CE コマンド ウィンドウ (**CMD.EXE**) を実行し、このシェル内でプログラム **WDDEBUG.EXE** を実行します。

- **stat:** Debug Monitor のステータスです。
  - **on:** Debug Monitor をオンにします。
  - **off:** Debug Monitor をオフにします。
  - **dbg_on:** Debug Monitor からカーネル デバッガにデバッグ メッセージを送り、Debug Monitor をオンにします。
  - **dbg_off:** Debug Monitor からカーネル デバッガへのデバッグ メッセージの転送を停止します。
  - **dump:** ユーザーが ENTER キーを押すまで、デバッグ情報の表示を続行します。
- **level**: デバッグするトレース レベルです。トレース レベルには ERROR、WARN、INFO、TRACE があり、ERROR はトレースを最小限に表示し、TRACE はすべてのメッセージを表示します。
- **section:** デバッグするセクションです。WinDriver API のどの部分を監視するかを指定します。パラメータを指定せずに **wddebug** を実行すると、プログラムの操作方法が表示され、サポートされているすべてのセクションが表示されます。

wddebug status コマンドを実行すると、Debug Monitor のバージョン、実行中の WinDriver ドライバ モジュールのバージョン、現在の Debug Monitor の情報 (トレース レベル、セクションを含む) が表示されます。

トレース レベルやセクションを設定し Debug Monitor をオンにしてから、**wddebug dump** を実行してデバッグ メッセージを表示します。終了後、Enter キーを押してダンプを停止し、**wddebug off** を実行して Debug Monitor をオフにします。

例:

wddebug の一般的な操作手順を次に示します。

- すべてのセクションとすべてのメッセージを表示するレース レベルを指定し Debug Monitor をオンにします。
  wddebug on TRACE ALL

- Enter キーを押すまで、デバッグ メッセージのダンプを続行します。
  wddebug dump

- Debug Monitor をオフにします。
  wddebug off

第 8 章
特定のチップ セットの拡張サポート

8.1 概要

前述の章で説明した標準 WinDriver API および PCI / ISA / PCMCIA / CardBus および USB 用のドライバ開発をサポートする DriverWizard のコード生成機能に加えて、WinDriver は 特定のチップ セットに対する拡張サポートを提供しています。拡張サポートには、それらのチップセット用に特別に用意されたカスタム API およびサンプル診断コードが含まれます。

現在 WinDriver の拡張サポートは次のチップセット (PLX 9030、9050、9052、9054、9056、9080、9656、Altera、Xilinx VirtexII、AMCC S5933、Cypress EZ-USB ファミリ、Microchip PIC18F4550、Philips PDIUSBD12、Texas Instruments TUSB3410、TUSB3210、TUSB2136、TUSB5052、Agere USS2828、Silicon Laboratories C8051F3) で利用できます。

注意: Cypress EZ-USB FX2LP CY7C68013A、Microchip PIC18F4550、Philips PDIUSBD12、および Silicon Laboratories C8051F320 チップセット用 USB デバイス ファームウェアの開発向けの WinDriver USB Device ツールキットの拡張サポートに関する詳細は第 16 章 を参照してください。

8.2 特定のチップ セット サポートを利用したドライバ開発

拡張サポートを利用可能なチップセット [8.1] を使用したデバイス用ドライバを開発する場合は、次の手順に従って WinDriver のチップセット特有のサポートを使用します。

1. **`WinDriver/chip_vendor/chip_name/`** ディレクトリにあるデバイス用のサンプル診断プログラムの場所を見つけます。ほとんどのサンプル診断プログラムの名前はサンプルの目的から来ています (たとえば、ファームウェアをダウンロードするサンプルは download_sample です)。ソースコードは特定のチップセットの名前 chip_name/ ディレクトリに保存されています。

   プログラムの実行ファイルはターゲットになるオペレーティング システムのサブディレクトリに保存されてます (たとえば、Windows の場合 **`WIN32\`** ディレクトリです)。

2. カスタム診断プログラムを実行してデバイスを診断し、サンプル プログラムによって提供されるオプションを把握してください。

3. この診断プログラムのソース コードをデバイスドライバの雛形として使用します。開発用途に合わせてコードを修正します。コードを修正する場合、特定のチップ用のカスタム WinDriver API を利用することができます。このカスタム API は **`WinDriver/chip_vendor/lib/`** ディレクトリに保存されています。

4. 以上の手順で作成したユーザー モード ドライバのパフォーマンスを向上させる必要がある場合は (割り込み処理等)、WinDriver Kernel PlugIn を説明している第 11 章 の「Kernel PlugIn について」

を参照してください。ソースコードの一部を WinDriver の Kernel PlugIn に移動して、関数の呼び出しにかかるオーバーヘッドを解消し、最大のパフォーマンスを得ることができます。

第 9 章
実行に当たっての問題

この章ではドライバ開発においての問題を説明します。また DriverWizard が自動的に処理できない操作を WinDriver を使用して実行する手順を説明します。

WinDriver の特定チップセット [第 8 章] 向けの拡張サポートは、DMA 割り込み処理などのハードウェア特有のタスクを実行すうるカスタム API を含んでいます。そのため、これらのチップセット用ドライバの開発者は、これらのタスクを実行するコードを実装する必要はありません。

9.1 DMA の実行

このセクションでは、バスマスタとして実行されるデバイスのためのバスマスタ ダイレクト メモリ アクセス (**DMA**) を実装する WinDriver の使用方法を説明します。

DMA とは、接続されたデバイスからホストのメモリへ直接データを転送可能な PCI、PCMCIA、および CardBus を含んだコンピュータのバス構造によって提供される機能です。CPU はデータ転送に関与しないため、ホスト側のパフォーマンスの向上につながります。

DMA バッファを次の 2 つの方法で割り当てることができます。

- **Contiguous Buffer (連続バッファ):** 連続メモリ ブロックを割り当てます。
- **Scatter/Gather:** 割り当てられたバッファは物理メモリ内では断片的で、連続して割り当てる必要はありません。割り当てられた物理メモリブロックは呼び出し処理の仮想アドレス空間で連続バッファへマップされています。そのため割り当てられた物理メモリ ブロックへ容易にアクセスすることができます。

デバイスの DMA コントローラのプログラミングはハードウェアにより異なります。通常、**ローカル アドレス** (デバイス上)、**ホスト アドレス** (PC の物理メモリ アドレス)、および**転送カウント** (転送するメモリ ブロック サイズ) を使用してデバイスをプログラムし、次に転送を開始するレジスタを設定します。

WinDriver は Contiguous Buffer DMA および Scatter/Gather DMA (ハードウェアがサポートしている場合) を実装する API を提供します (`WDC_DMAContigBufLock()`、`WDC_DMASGBufLock()`、および `WDC_DMABufUnlock()` の詳細を参照してください)。低水準 WD_DMAxxx API は WinDriver PCI 低水準 API リファレンスで説明されていますが、代わりにラッパー WDC_xxx API を使用することを推奨します。

このセクションでは Scatter/Gather および Contiguous Buffer DMA を実装する WinDriver の使用方法を実演するサンプル コードを紹介します。

注意:

- このサンプル ルーチンは、割り込みまたはポーリングを使用して DMA の完了を測定するデモです。

- このサンプル ルーチンは DMA バッファを割り当て、DMA 割り込みを有効にします (ポーリングが使用されていない場合)。次にバッファを解放し、各 DMA 転送への割り込みを無効にします (有効の場合)。しかし、実際の DMA コードを実行する場合、アプリケーションの初めに一度 DMA バッファを割り当てることができ、DMA の割り込みを有効にすることができます (ポーリングが使用されたいない場合)。次に、同じバッファを使用して DMA 転送を繰り返し実行し、割り込みを無効にします (有効の場合) 。アプリケーションが DMA を実行する必要がなくなった場合のみバッファを解放します。

9.1.1 Scatter/Gather DMA

DMA 実装のサンプル

次のサンプル ルーチンは WinDriver の WDC API を使用して Scatter/Gather DMA バッファを割り当て、バスマスタ DMA 転送を実行します。

PLX チップセット [第 8 章] 用の拡張サポートの詳細な例は **WinDriver/plx/lib/plx_lib.c** ライブラリ ファイルおよび **WinDriver/plx/diag_lib/plx_diag_lib.c** 診断ライブラリ ファイル (**plx_lib.c** DMA API を使用) に保存されています。

Altera PCI 開発キットボード用 Scatter/Gather DMA を実装する WD_DMAxxx API を使用したサンプルは **WinDriver/altera/pci_dev_kit/lib/altera_lib.c** ライブラリ ファイルに保存されています。

9.1.1.1 Scatter/Gather DMA 実装のサンプル

```
BOOL DMARoutine(WDC_DEVICE_HANDLE hDev, DWORD dwBufSize,
    UINT32 u32LocalAddr, DWORD dwOptions, BOOL fPolling, BOOL fToDev)
{
    PVOID pBuf;
    WD_DMA *pDma = NULL;
    BOOL fRet = FALSE;

    /* Allocate a user-mode buffer for Scatter/Gather DMA */
    pBuf = malloc(dwBufSize);
    if (!pBuf)
        return FALSE;

    /* Lock the DMA buffer and program the DMA controller */
    if (!DMAOpen(hDev, pBuf, u32LocalAddr, dwBufSize, fToDev, &pDma))
        goto Exit;

    /* Enable DMA interrupts (if not polling) */
    if (!fPolling)
    {
        if (!MyDMAInterruptEnable(hDev, MyDmaIntHandler, pDma))
            goto Exit; /* Failed enabling DMA interrupts */
    }

    /* Flush the CPU caches (see documentation of WDC_DMASyncCpu()) */
    WDC_DMASyncCpu(pDma);

    /* Start DMA - write to the device to initiate the DMA transfer */
    MyDMAStart(hDev, pDma);

    /* Wait for the DMA transfer to complete */
    MyDMAWaitForCompletion(hDev, pDma, fPolling);

    /* Flush the I/O caches (see documentation of WDC_DMASyncIo()) */
```

```
    WDC_DMASyncIo(pDma);

    fRet = TRUE;
Exit:
    DMAClose(pDma, fPolling);
    free(pBuf);
    return fRet;
}


/* DMAOpen: Locks a Scatter/Gather DMA buffer */
BOOL DMAOpen(WDC_DEVICE_HANDLE hDev, PVOID pBuf, UINT32 u32LocalAddr,
    DWORD dwDMABufSize, BOOL fToDev, WD_DMA **ppDma)
{
    DWORD dwStatus, i;
    DWORD dwOptions = fToDev ? DMA_TO_DEVICE : DMA_FROM_DEVICE;

    /* Lock a Scatter/Gather DMA buffer */
    dwStatus = WDC_DMASGBufLock(hDev, pBuf, dwOptions, dwDMABufSize, ppDma);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed locking a Scatter/Gather DMA buffer. Error 0x%lx
     - %s\n",
            dwStatus, Stat2Str(dwStatus));
        return FALSE;
    }

    /* Program the device's DMA registers for each physical page */
    MyDMAProgram((*ppDma)->Page, (*ppDma)->dwPages, fToDev);

    return TRUE;
}

/* DMAClose: Unlocks a previously locked Scatter/Gather DMA buffer */
void DMAClose(WD_DMA *pDma, BOOL fPolling)
{
    /* Disable DMA interrupts (if not polling) */
    if (!fPolling)
        MyDMAInterruptDisable(hDev);

    /* Unlock and free the DMA buffer */
    WDC_DMABufUnlock(pDma);
}
```

9.1.1.2 実行しなければならないものは

上記のサンプル コードで、MyDMAxxx() ルーチン以下の実装はデバイスの使用により異なります。

MyDMAProgram(): デバイスの DMA レジスタをプログラムします。
詳細は、デバイスのデータシートを参照してください。

- MyDMAStart(): DMA 転送を開始するデバイスへ書き込みます。
- MyDMAInterruptEnable() および MyDMAInterruptDisable(): WDC_IntEnable() および WDC_IntDisable() を使用して、ソフトウェアの割り込みを有効/無効にし、物理的にハードウェア DMA 割り込みを有効/無効にするためにデバイスの関連したレジスタを書き込み/読み取ります。(WinDriver での割り込み処理に関する詳細はセクション [9.2] を参照してください)。

- `MyDMAWaitForComplete ()`: 転送の完了をデバイスにポーリングするか、"DMA DONE" (DMA の完了) 割り込みを待機します。

注意: WD_xxx API (WinDriver PCI 低水準 API リファレンスを参照) を使用して、1MB より大きい Scatter/Gather DMA バッファを割り当てる場合、FAQ (http://www.xlsoft.com/jp/products/windriver/support/faq.html#dma1) で説明されている通り、WD_DMALock() で DMA_LARGE_BUFFER フラグを設定し、追加のメモリ ページ用のメモリを割り当てる必要があります。しかし、WDC_DMASGBufLock() を使用して DMA バッファを割り当てる場合、関数が処理するため大きいバッファを割り当てる特別な実装は必要ありません。

9.1.2 Contiguous Buffer (連続バッファ) DMA

次のサンプル ルーチンは WinDriver の WDC API を使用して Contiguous DMA バッファを割り当て、バスマスタ DMA 転送を実行します。

PLX チップセット [第 8 章] 用の拡張サポートの詳細な例は **`WinDriver/plx/lib/plx_lib.c`** ライブラリ ファイルおよび **`WinDriver/plx/diag_lib/plx_diag_lib.c`** 診断ライブラリ ファイル (**`plx_lib.c`** DMA API を使用) に保存されています。

AMCC 5933 用 Contiguous Buffer DMA を実装する WD_DMAxxx API を使用したサンプルは **`WinDriver/amcc/lib/amcclib.c`** ライブラリ ファイルに保存されています (WD_DMAxxx API については、WinDriver PCI 低水準 API リファレンスを参照してください)。

9.1.2.1 Contiguous Buffer DMA 実装のサンプル

```
BOOL DMARoutine(WDC_DEVICE_HANDLE hDev, DWORD dwDMABufSize,
    UINT32 u32LocalAddr, DWORD dwOptions, BOOL fPolling, BOOL fToDev)
{
    PVOID pBuf = NULL;
    WD_DMA *pDma = NULL;
    BOOL fRet = FALSE;

    /* Allocate a DMA buffer and open DMA for the selected channel */
    if (!DMAOpen(hDev, &pBuf, u32LocalAddr, dwDMABufSize, fToDev, &pDma))
        goto Exit;

    /* Enable DMA interrupts (if not polling) */
    if (!fPolling)
    {
        if (!MyDMAInterruptEnable(hDev, MyDmaIntHandler, pDma))
            goto Exit; /* Failed enabling DMA interrupts */
    }

    /* Flush the CPU caches (see documentation of WDC_DMASyncCpu()) */
    WDC_DMASyncCpu(pDma);

    /* Start DMA - write to the device to initiate the DMA transfer */
    MyDMAStart(hDev, pDma);

    /* Wait for the DMA transfer to complete */
    MyDMAWaitForCompletion(hDev, pDma, fPolling);

    /* Flush the I/O caches (see documentation of WDC_DMASyncIo()) */
    WDC_DMASyncIo(pDma);

    fRet = TRUE;
```

```
Exit:
    DMAClose(pDma, fPolling);
    return fRet;
}


/* DMAOpen: Allocates and locks a Contiguous DMA buffer */
BOOL DMAOpen(WDC_DEVICE_HANDLE hDev, PVOID *ppBuf, UINT32 u32LocalAddr,
    DWORD dwDMABufSize, BOOL fToDev, WD_DMA **ppDma)
{
    DWORD dwStatus;
    DWORD dwOptions = fToDev ? DMA_TO_DEVICE : DMA_FROM_DEVICE;

    /* Allocate and lock a Contiguous DMA buffer */
    dwStatus = WDC_DMAContigBufLock(hDev, ppBuf, dwOptions, dwDMABufSize,
ppDma);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed locking a Contiguous DMA buffer. Error 0x%lx - %s\n",
            dwStatus, Stat2Str(dwStatus));
        return FALSE;
    }

    /* Program the device's DMA registers for the physical DMA page */
    MyDMAProgram((*ppDma)->Page, (*ppDma)->dwPages, fToDev);

    return TRUE;
}

/* DMAClose: Frees a previously allocated Contiguous DMA buffer */
void DMAClose(WD_DMA *pDma, BOOL fPolling)
{
    /* Disable DMA interrupts (if not polling) */
    if (!fPolling)
        MyDMAInterruptDisable(hDev);

    /* Unlock and free the DMA buffer */
    WDC_DMABufUnlock(pDma);
}
```

9.1.2.2 実行しなければならないものは

上記のサンプル コードで、MyDMAxxx() ルーチン以下の実装はデバイスの使用により異なります。

- MyDMAProgram(): デバイスの DMA レジスタをプログラムします。
  詳細は、デバイスのデータシートを参照してください。

- MyDMAStart(): DMA 転送を開始するデバイスへ書き込みます。

- MyDMAInterruptEnable() および MyDMAInterruptDisable(): WDC_IntEnable() および WDC_IntDisable() を使用して、ソフトウェアの割り込みを有効/無効にし、物理的にハードウェア DMA 割り込みを有効/無効にするためにデバイスの関連したレジスタを書き込み/読み取ります。(WinDriver での割り込み処理に関する詳細はセクション [9.2] を参照してください) 。

- MyDMAWaitForComplete (): 転送の完了をデバイスにポーリングするか、”DMA DONE” (DMA の完了) 割り込みを待機します。

9.1.3 SPARC での DMA の実行

Solaris の SPARC では、**DVMA** (Direct Virtual Memory Access) をサポートします。DVMA をサポートするプラットフォームでは、物理アドレスではなく仮想アドレスを持つデバイスを提供することによって、転送を実行します。このメモリ アクセスの方法で、提供された仮想アドレスへのデバイス アクセスを MMU (Memory Management Unit) を使用する適切な物理アドレスへ移します。デバイスは dis-contiguous 物理ページへマップされる連続仮想イメージへ / から転送します。これらのプラットフォームで操作するデバイスは、Scatter/Gather DAM 機能を必要としません。

9.2 割り込み処理

WinDriver はデバイスからの割り込みを処理するタスクを簡素化するために、API、DriverWizard コード生成、およびサンプルを提供しています。

WinDriver で拡張サポートされるチップ セット [第 8 章] を使用したデバイス用のドライバを開発している場合、割り込み処理を実行する手段として、特定チップ用のカスタム WinDriver 割り込み API を使用することを推奨します。これらのルーチンはターゲットのハードウェアで実装されます。

その他のチップの場合、DriverWizard を使用してデバイスの割り込みに関する情報 (割り込み要求 (IRQ) 番号、タイプ、共有状態など) を検出/定義し、割り込みが発生した時にカーネルで実行するコマンドを定義します。次に、ウィザードで定義した情報を基にデバイスの割り込みを処理する WinDrivir API の使用方法を実例する割り込みルーチンを含む診断コードの雛形を生成します。

以下のセクションでは PCI、PCMCIA、および ISA 割り込みを処理する WinDriver API の使用方法を説明します。サンプルおよび DriverWizard で生成された割り込みコードを理解し、オリジナルの割り込み処理を作成するために、次のセクションをお読みください。

注意: このセクションでは、ユーザー モード アプリケーションから割り込みを処理する WinDriver の使用方法を説明します。割り込み処理はパフォーマンス上重大なタスクであるため、カーネルで割り込みを直接処理することもできます。WinDriver の Kernel PlugIn [第 11 章] は、カーネルの割り込みルーチンの実装を許可しています。Kernel PlugIn から割り込みを処理する方法についてはセクション 11.6.5 を参照してください。

9.2.1 一般的な割り込み処理

WinDriver を使用した割り込み処理は以下の手順で行います。

1. デバイス上で割り込みを有効した場合、発生した割り込みを処理するスレッドを作成します
2. スレッドは割り込みが発生するのを待つ無限ループを実行します。
3. 割り込みが発生すると、WinDriver はカーネルでユーザーにより設定された転送コマンド (セクション [9.2.2.1] を参照) を実行し、コントロールがユーザー モードにリターンしたとき、ドライバの割り込み処理ルーチンが呼び出されます。
4. 割り込み処理コードがリターンすると、待ちループが継続します。

デバイスからの割り込みを待つために使用される低水準の `WinDriver WD_IntWait()` 関数 (割り込みを有効にする関数 `WDC_IntEnable()` / `InterruptEnable()` によって呼び出される) は、割り込みが発生するまでスレッドをスリープ状態にします。割り込みを待つ間は CPU を消費しません。割り込みが発生

すると、割り込みは最初に WinDriver カーネルで処理され、次に `WD_IntWait()` が割り込み処理スレッドを呼び出し、リターンします。

割り込みスレッドはユーザー モードで実行するので、ファイル操作および GDI 関数を含む任意の Windows API 関数を呼び出せます。

9.2.1.1 WDC ライブラリを使用した割り込み処理

WinDriver Card (WDC) ライブラリは、簡素化された割り込み処理関数 (`WDC_IntEnable()`、`WDC_IntDisable()`、および `WDC_IntIsEnabled()`) を含む便利なラッパー関数を WinDriver PCI/PCMCIA/ISA API へ提供します。

以下のサンプル コードは、簡単な割り込み処理を実装する WDC 割り込み API を使用方法を示しています。これらの関数を使用する割り込み処理のソース コードは WinDriver **`pci_diag`** (**`WinDriver/samples/pci_diag/`**)、**`pcmcia_diag`** (**`WinDriver/samples/pcmcia_diag/`**)、PLX (**`WinDriver/plx/`**) のサンプル、および DriverWizard で生成された PCI/PCMCIA/ISA コードを参照してください。

```
VOID DLLCALLCONV interrupt_handler (PVOID pData)
{
    PWDC_DEVICE pDev = (PWDC_DEVICE)pData;

    /* Implement your interrupt handler routine here */
    printf("Got interrupt %d\n", pDev->Int.dwCounter);
}

...

int main()
{
    DWORD dwStatus;
    WDC_DEVICE_HANDLE hDev;
    ...
    WDC_DriverOpen(WDC_DRV_OPEN_DEFAULT, NULL);
    ...
    hDev = WDC_IsaDeviceOpen(...);
    ...
    /* Enable interrupts. This sample passes the WDC device handle as the data
        for the interrupt handler routine */
    dwStatus = WDC_IntEnable(hDev, NULL, 0, 0,
        interrupt_handler, (PVOID)hDev, FALSE);
    /* WDC_IntEnable() allocates and initializes the required WD_INTERRUPT
        structure, stores it in the WDC_DEVICE structure, then calls
        InterruptEnable(), which calls WD_IntEnable() and creates an
         Interrupt handler thread */
    if (WD_STATUS_SUCCESS != dwStatus)
    {
       printf ("Failed enabling interrupt. Error: 0x%x - %s\n",
           dwStatus, Stat2Str(dwStatus));
    }
    else
    {
        printf("Press Enter to uninstall interrupt\n");
        fgets(line, sizeof(line), stdin);

        WDC_IntDisable(hDev);
        /* WDC_IntDisable() calls InterruptDisable(), which calls
           WD_IntDisable() */
```

```
    }
    ...
    WDC_IsaDeviceClose(hDev);
    ...
    WDC_DriverClose();
}
```

9.2.2 ISA / EISA および PCI 割り込み

一般に、ISA/EISA 割り込みは、PCI 割り込みがレベル依存であるのに反し、エッジ トリガーされます。この違いは割り込み処理ルーチンを書く上で、多くの意味があります。

エッジ トリガーされる割り込みは、物理割り込み信号が Low から High になるときに、1 回だけ生成されます。したがって正確に 1 個の割り込みが生成されます。これによって Windows OS が WinDriver カーネル割り込みハンドラを呼び出し、`WD_IntWait()` 関数で待っているスレッドを解放します。この割り込みを認識するのに特別の作業は必要ありません。

レベル センシティブ割り込みは、物理割り込み信号が High である限り生成されます。割り込み信号がカーネルによる割り込み処理の最後に Low にならないときは、Windows OS が WinDriver カーネル割り込みハンドラを再び呼び出します。このため PC はハングします。この状態が起きるのを防ぐには、割り込みが WinDriver カーネル割り込みハンドラによって認識される必要があります。

9.2.2.1 カーネル レベルの転送コマンド (割り込みの認識)

通常、PCI カードの割り込み処理 (レベル センシティブ割り込みハンドラ) は、割り込みレベル (割り込みの認識) を Low にするためにカーネルで転送コマンドを実行する必要があります。転送コマンドは、PCI カードのランタイム レジスタへの書き込みを行うので、ハードウェアの割り込みを除去します。しかし、割り込みを認識するために実行される転送コマンドはハードウェアにより異なります。そのため、WinDriver を使用して、レベル センシティブ割り込み処理を行う場合、割り込みが発生した場合にカーネルで実行する転送コマンドを事前に設定する必要があります。

(`WD_IntWait()` が返る前に) WinDriver カーネル割り込み処理で実行される転送コマンドを渡すには、(`WD_TRANSFER` 構造体を使用して定義された) コマンド配列 を用意し、`WDC_IntEnable()`、または低水準な `InterruptEnable()` 関数、`WD_IntEnable()` 関数を使用して割り込みを有効にした時、コマンド配列を WinDriver へ渡します。

割り込みを有効にする関数は、割り込みマスクを定義して、割り込みのソースを確認することもできます。割り込みマスク コマンドは、転送コマンド配列で read (読み込み) 転送コマンドの直後に設定する必要があります。割り込みマスク コマンドを設定する場合 (`trans[i].cmdTrans = CMD_MASK`)、カーネルで割り込みを受信すると、WinDriver は、前の読み込みコマンドでカードから読み込まれた値と、割り込みマスクに設定されているマスクを比較します。値が一致する場合、WinDriver は割り込みを要求し、配列内の残りの転送コマンドを実行して、コントロールがユーザー モードへ返った時点で割り込みハンドラ ルーチンを呼び出します。値が一致しない場合、WinDriver は割り込みを拒否し、配列内の残りの転送コマンドを実行せずに、割り込みハンドラ ルーチンも呼び出しません。

割り込みが発生した際、IO ポート dwAddr へマップされる割り込みコマンド状態レジスタ (INTCSR) の値が intrMask であり、INTCSR へ「0」を記述することによって割り込みをクリアする場合のコード例を以下に示します。このコードは、INTCSR レジスタを最初に読み取る転送コマンドの配列を定義し、値を保存します。次に、値をマスクして割り込みのソースを確認し、割り込みを認識する「0」をこのレジスタへ記述します。すべてのコマンドは DWORD モードで実行されます。

```
WD_TRANSFER trans[3]; /* Array of WinDriver transfer command
                         structures */
BZERO(trans);

/* 1st command: Read a DWORD from the INTCSR I/O port */
trans[0].cmdTrans = RP_DWORD;
/* Set address of IO port to read from: */
trans[0].dwPort = dwAddr; /* Assume dwAddr holds the address of INTCSR */

/* 2nd command: Mask the interrupt to verify its source */
trans[1].cmdTrans = CMD_MASK;
trans[1].Data.Dword = intrMask; /* Assume intrMask holds your interrupt
                                   mask */

/* 3rd command: Write DWORD to the INTCSR I/O port
   This command will only be executed if the value read from INTCSR in the
   1st command matches the interrupt mask set in the 2nd command. */
trans[2].cmdTrans = WP_DWORD;
/* Set the address of IO port to write to: */
trans[2].dwPort = dwAddr; /* Assume dwAddr holds the address of INTCSR */
/* Set the data to write to the INTCSR IO port: */
trans[2].Data.Dword = 0;
```

転送コマンドを定義した後、割り込みを有効にすることができます。

以下のコードは、上記で用意した転送コマンドを使用して、割り込みを有効にする WDC ライブラリの使用方法を示します。

```
/* Enable the interrupts:
    hDev: WDC_DEVICE_HANDLE received from a previous call to
WDC_PciDeviceOpen()
    INTERRUPT_CMD_COPY: Used to save the read data - see explanation below
    interrupt_handler: Your user-mode interrupt handler routine
    pData: The data to pass to the interrupt handler routine */
WDC_IntEnable(hDev, &trans, 2, INTERRUPT_CMD_COPY, interrupt_handler,
    pData, FALSE);
```

9.2.3 Windows CE の割り込み

Windows CE は、物理割り込み番号ではなく論理割り込みスキームを使用します。論理割り込みスキームは、論理 IRQ 番号に物理 IRQ 番号をマップする内部カーネル テーブルを管理します。Windows CE からの割り込みを要求する場合、デバイスドライバは論理割り込み番号の使用を期待します。この場合、割り込みのマップには以下の 3 つのアプローチがあります。

1. **割り込みマッピング用の Windows CE Plug-and-Play を使用する** (PCI バスドライバ)

   Windows CE で割り込みマッピングを行うにはこのアプローチを推奨します。PCI バスドライバを使用してデバイスを登録します。この方法の後に PCI バスドライバが IRQ マッピングを実行し、WinDriver にこれを使用するように指示します。

   PCI バスドライバを使用してデバイスを登録する例はセクション 6.3 を参照してください。

2. **プラットフォーム割り込みマッピングを使用する** (X86 または ARM)

多くの x86 または MIPS プラットフォームでは、予約済みの割り込みを除き、すべての物理割り込みを以下の簡単なマップを使用して静的にマップします。
logical interrupt = SYSINTR_FIRMWARE + physical interrupt

Windows CE Plug-and-Play にデバイスを登録していない場合、WinDriver は次のマッピングを行います。

3. **マップされた割り込み値を指定する**

注意: このオプションは、Platform Builder によってのみ実行できます。

デバイスのマップされた論理割り込み値を提供します。入手できない場合、物理 IRQ を論理割り込みへ性的にマップします。次に 論理割り込みと **INTERRUPT_CE_INT_ID** フラグを設定して `WD_CardRegister()` を呼びます。静的割り込みマップは **CFWPC.C** ファイル (**%_TARGETPLATROOT%\KERNEL\HAL** ディレクトリ) にあります。

静的マップはまた予約済み割り込みマップを使用する場合にも役立ちます。対象のプラットフォームの静的マップは以下のようになります:

- **IRQ0**: タイマー 割り込み
- **IRQ2**: 第 2 PIC 用のカスケード 割り込み
- **IRQ6**: フロッピー コントローラ
- **IRQ7**: LPT1 (PPSH が割り込みを使用しないため)
- **IRQ9**
- **IRQ13**: 数値コプロセッサ

これらの割り込みを初期化、または使用を試みても失敗します。ただし、PPSH を使用せずに、他の目的でパラレル ポートを再要求する場合には、これらの割り込みの一つを使用できる場合があります。

この問題を解決するには、単純に以下のようなコードを含む **CFWPC.C** (**%_TARGETPLATROOT%\ KERNEL\HAL** ディレクトリ以下にあります) を編集します。割り込みマップ テーブルの割り込みの値を 7 に設定します。

```
SETUP_INTERRUPT_MAP(SYSINTR_FIRMWARE+7,7);
```

IRQ9 を割り当てられた PCI カードの場合、WinCE はデフォルトでは、この割り込みをマップしないので、カードからの割り込みを受信できません。この場合、IRQ9 に同様のエントリを挿入する必要があります。

```
SETUP_INTERRUPT_MAP(SYSINTR_FIRMWARE+9,9);
```

9.2.3.1 Windows CE で割り込み待ち時間を向上させる

レジストリおよびコードを若干変更して PCI デバイス用の Windows CE での割り込み待ち時間を短縮することができます。

1. Windows CE プラットフォームでドライバを開発する場合、セクション 6.3 で説明したとおり、初めに WinDriver でデバイスが動作するように登録する必要があります。

レジストリの最後の値を `"WdIntEnh"=dword:0` から `"WdIntEnh"=dword:1` へ変更します。この行を除外したり、値を 0 のままにした場合は、割り込み待ち時間は短縮されません。

2. プロジェクトの "Preprocessor Definitions" に **`WD_CE_ENHANCED_INTR`**を追加し、プロジェクト全体を再コンパルします。Microsoft eMbedded Visual C++ を使用している場合、"Preprocessor Definitions" は "Project Settings" の下にあります。

3. 低水準 `WD_xxx API` (WinDriver PCI 低水準 API リファレンスを参照) を使用する場合は、`InterruptEnable()` を呼び出した直後に `CEInterruptEnhance()` を呼び出します。

   注意: WinDriver WDC API を使用して割り込みを処理する場合、`CEInterruptEnhance()` は `WDC_IntEnable()` によって自動的に呼び出されるため、呼び出す必要はありません。

   `CEInterruptEnhance()` は、パラメータを 2 つ受け取ります。

   ```
   void CEInterruptEnhance(HANDLE hThread, DWORD dwSysintr);
   ```

   - **`hThread`**: `InterruptEnable()` から割り込みスレッドを受け取る処理
   - **`dwSysintr`**: `WD_CardRegister()` から `cardReg.Card.Item[i].I.Val.dw4` (`'i'` は割り込みアイテムの配列のインデックス) として返されるマップされた割り込み値

9.3 USB コントロール転送

9.3.1 USB データ交換

USB 標準はホストとデバイス間で 2 種類のデータ交換をサポートします。

- **機能データ交換**は、デバイスからまたはデバイスへのデータの転送に使用されます。バルク転送、割り込み転送、等時性 (Isochronous) 転送の 3 種類のデータの転送があります。
- **コントロール交換**は、デバイスを最初に接続したときにデバイスの設定に使用され、デバイスの他のパイプの制御を含むその他のデバイス特有の目的のためにも使用されます。コントロール交換は、常駐である主にデフォルトで Pipe 0 のコントロール パイプから生じます。

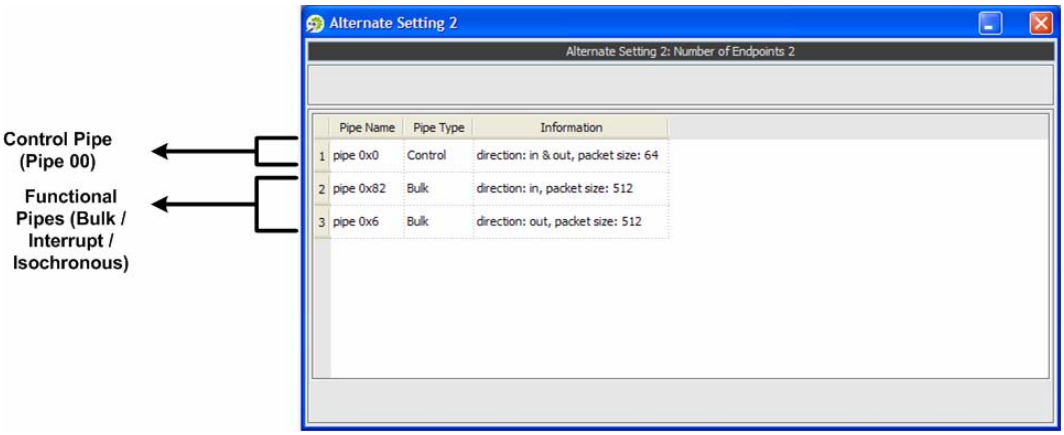


図 9.1: USB データ交換

9.3.2 コントロール転送の詳細

コントロール トランザクションは常にセットアップ ステージから始まります。次に、ゼロまたは要求された操作の特別な情報を送信するコントロール データ トランザクション (データ ステージ) がそれに続きます。最後に、ステータス トランザクションがホストへステータスを返すことによりコントロール転送が完了します。

セットアップ ステージでは、8 バイト セットアップ パケットがデバイスのコントロール エンドポイントへ情報を伝達するために使用されます。セットアップ パケットのフォーマットは USB の仕様で指定されています。

コントロール転送はリード トランザクションまたはライト トランザクションです。リード トランザクションではセットアップ パケットはデバイスからリードされる特性と大量のデータを含んでいます。ライト トランザクションでは、セットアップ パケットはライト トランザクションに関連するデバイスへ送られた (書かれた) コマンドとコントロール データを含みます。これらはデータ ステージでデバイスに送られます。

図 9.2 (USB の仕様から引用) は、read (読み取り) および write (書き込み) トランザクションのシーケンスを示しています。'in' はデバイスからホストへデータが流れることを意味し、'out' はホストからデバイスへデータが流れることを意味します。

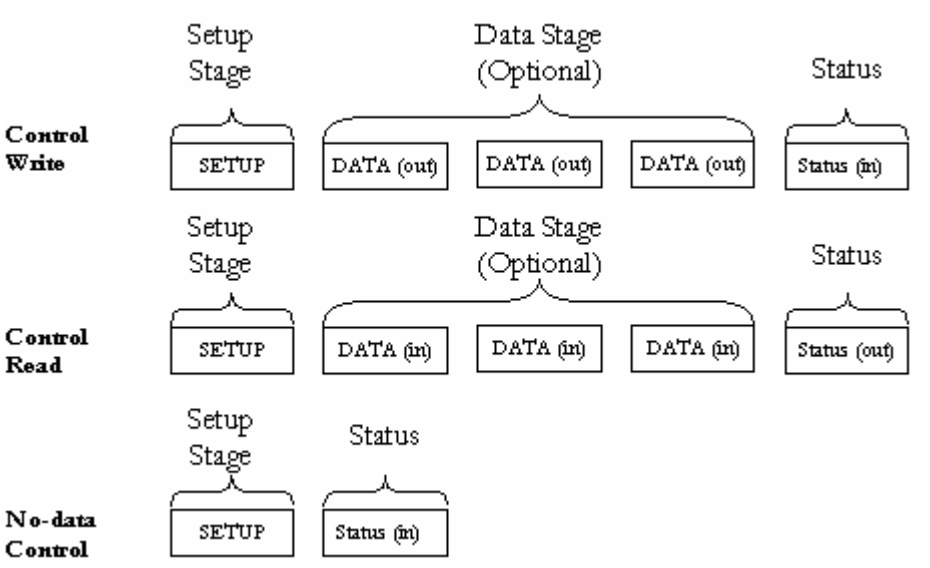


図 9.2: USB のリードとライト

9.3.3 セットアップ パケット

セットアップ パケット (コントロール データ ステージおよびステータス ステージを組み合わせたもの) は設定に使用され、デバイスへコマンドを送信します。USB の仕様の第 9 章 では、標準デバイス 要求を定義します。これらの USB 要求は、セットアップ パケットを使用してホストからデバイスへ送信されます。USB デバイスはこれらの要求に正確に応答する必要があります。また、それぞれのベンダーは、デバイス特有のセットアップ パケットを定義し、デバイス特有の操作を実行することもできます。標準セットアップ パケット(標準 USB デバイス要求 の詳細を次節で説明します。ベンダーのデバイス特有セットアップ パケットは、各 USB デバイスに対して、ベンダーのデータ ブックに記されています。

9.3.4 USB セットアップ パケットのフォーマット

次の表は USB セットアップ パケットのフォーマットを示しています。詳細については、http://www.usb.org の USB の仕様を参照してください。

バイト	フィールド	説明
0	bmRequest Type	Bit 7: リクエスト方向 (0=ホストからデバイス - out, 1=デバイスからホスト - in) Bits 5..6: リクエスト タイプ (0=標準, 1=クラス, 2=ベンダー, 3=reserved) Bits 0..4: 受信側 (0=デバイス, 1=インターフェイス, 2=エンドポイント,3=その他)
1	bRequest	実際のリクエスト (次の 9.3.5「標準デバイスが要求するコード」を参照してください)
2	wValueL	リクエストにより異なるワード サイズ値 (たとえば、CLEAR_FEATURE リクエストでは値は機能の選択に使用され、GET_DESCRIPTOR リクエストでは、値は記述子のタイプを示し、SET_ADDRESS リクエストでは値はデバイス アドレスを含みます。)
3	wValueH	Value ワードの上位バイト
4	wIndexL	リクエストにより異なるワード サイズ値。索引は一般的にエンドポイントまたはインターフェイスを指定するために使用されます。
5	wIndexH	Index ワードの上位バイト
6	wLengthL	データ ステージがある場合は、転送されるバイト数を示した ワード サイズ値
7	wLengthH	Length ワードの上位バイト

9.3.5 標準デバイスが要求するコード

以下の表は、標準デバイスが要求するコードを表しています。

BRequest	値
GET_STATUS	0
CLEAR_FEATURE	1
Reserved for future use	2
SET_FEATURE	3
Reserved for future use	4
SET_ADDRESS	5
GET_DESCRIPTOR	6

SET_DESCRIPTOR	7
GET_CONFIGURATION	8
SET_CONFIGURATION	9
GET_INTERFACE	10
SET_INTERFACE	11
SYNCH_FRAME	12

9.3.6 セットアップ パケットの例

セットアップ パケットの構成と、その中でのフィールドを図式化した、標準 USB デバイスの一例を挙げます。セットアップ パケットは Hex 形式です。

次のセットアップ パケットは、USB デバイスから 'Device descriptor' を取り込む 'Control Read' トランザクションです。'Device descriptor' は USB 標準リビジョン、ベンダー ID、およびプロダクト ID などの情報を含みます。

GET_DESCRIPTOR (デバイス) セットアップ パケット

80	06	00	01	00	00	12	00

セットアップ パケットの意味:

バイト	フィールド	値	説明
0	BmRequest Type	80	8h=1000b bit 7=1 -> データ方向 (デバイスからホスト) 0h=0000b bits 0..1=00 -> 受信側は "デバイス"
1	bRequest	06	リクエストは 'GET_DESCRIPTOR'
2	wValueL	00	
3	wValueH	01	記述子のタイプはデバイスです。(値は USB spec で定義されます。)
4	wIndexL	00	(デバイス記述子が 1 つなのでこのセットアップ パケットでは Index は関係ありません。
5	wIndexH	00	
6	wLengthL	12	取り込まれるデータの長さ: 18(12h) バイト ('device descriptor' の長さ)
7	wLengthH	00	

これに応えて、デバイスは 'Device Descriptor' データを送信します。たとえば、これは 'Cypress EZ-USB Integrated Circuit' の 'Device Descriptor' です:

バイト番号	0	1	2	3	4	5	6	7	8	9	10
データ	12	01	00	01	Ff	ff	ff	40	47	05	80

バイト番号	11	12	13	14	15	16	17
データ	00	01	00	00	00	00	01

USB の仕様で定義づけられているように、バイト 0 はデスクリプタの長さを示し、バイト 2-3 は USB の仕様リリース ナンバーを含みます。バイト 7 はエンドポイント 00 に対して最も大きいパケット サイズです。バイト 8-9 は ベンダー ID で、バイト 10-11 はプロダクト ID を示します。

9.4 WinDriver でコントロール転送を行う

DriverWizard を使用して、対象のデバイスを診断を行う際に、WinDriver で Pipe00 でコントロール転送を簡単に送受信することができます。対象のハードウェアに対して DriverWizard [第 5 章] で生成された API を使用するか、もしくはアプリケーションから直接 WinDriver の `WDU_Transfer()` 関数 を呼ぶことができます。

9.4.1 DriverWizard でのコントロール転送

1. **Pipe00** を選択し、[Read/Write to Pipe] をクリックします。
2. カスタム セットアップ パケットを入力するか、もしくは標準 USB 要求を使用します。
   - カスタム要求の場合: 必要なセットアップ パケット フィールドを入力します。データ ステージを含む書き込みトランザクションの場合、[Write to pipe data (Hex)] フィールドにデータを入力します。必要なトランザクションに応じて、[Read From Pipe] または [Write To Pipe] を選択します (図 9.3 を参照してください)。

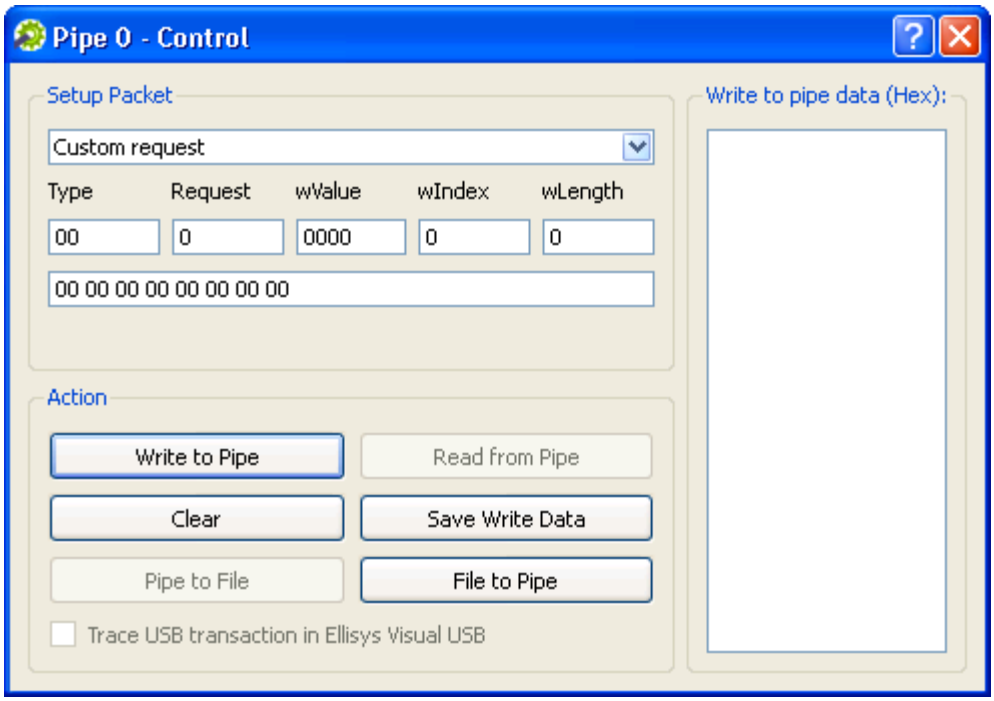


図 9.3: カスタム要求

- 標準 USB 要求の場合: **GET_DESCRIPTOR CONFIGURATION**、**GET_DESCRIPTOR DEVICE** 、**GET_STATUS DEVICE** などの要求一覧から USB 要求を選択します (図 9.4 を参照してください)。選択するとダイアログ ボックスの右側に各要求の説明が表示されます。

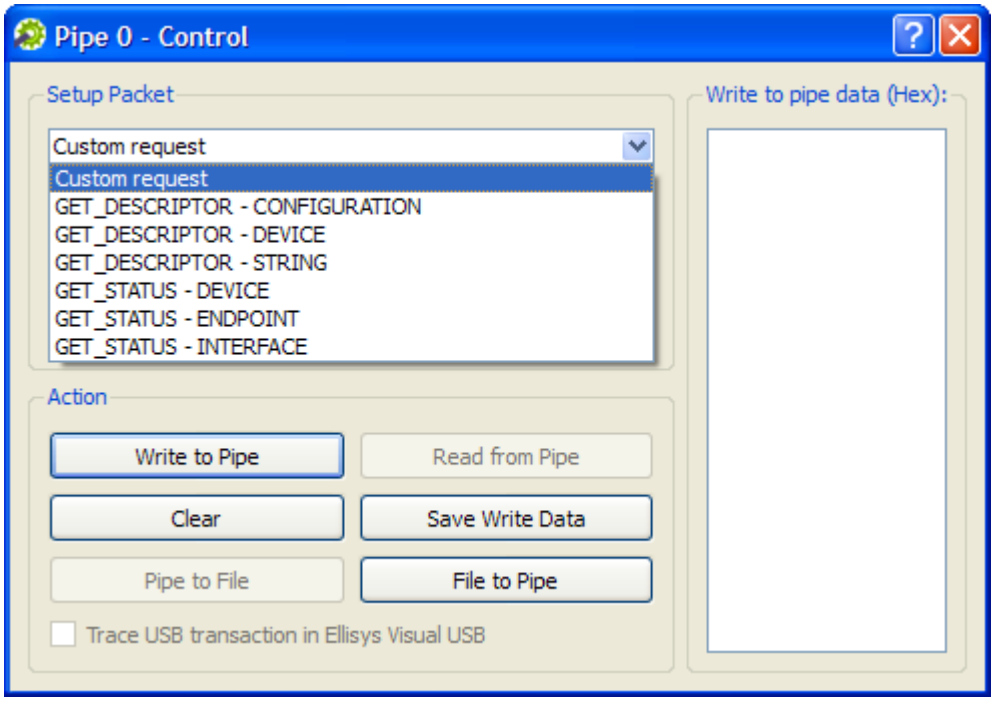


図 9.4: 要求一覧

3. 読み込まれたデータやエラーなど、転送結果を DriverWizard の [Log] 画面から参照できます。**GET_DESCRIPTOR DEVICE** 要求が処理された後の [Log] 画面を、次の図 9.5 に示します。

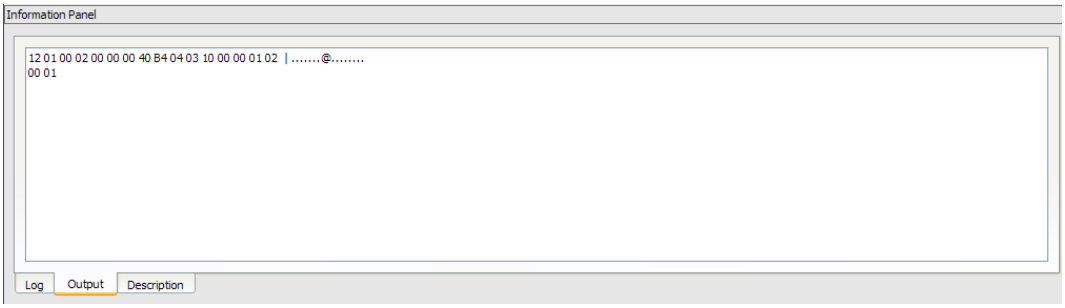


図 9.5: USB 要求ログ

9.4.2 WinDriver API でのコントロール転送

リードまたはライトトランザクションをコントロール パイプ上で実行する場合、ハードウェアに対して DriverWizard で生成された API を使用するか、またはアプリケーションで WinDriver `WDU_Transfer()` 関数を直接呼ぶことができます。

セットアップ パケットを `BYTE` の配列 `setupPacket[8]` に格納します。そして、これらの関数を呼び、Pipe00 にセットアップ パケットを送信したり、デバイスからコントロール データやステータス データを受信します。

- 次のサンプルは、`setupPacket[8]` 変数に `GET_DESCRIPTOR` セットアップ パケットを格納する方法を示します。

```
setupPacket[0] = 0x80;    /* BmRequstType */
setupPacket[1] = 0x6;     /* bRequest  [0x6 == GET_DESCRIPTOR] */
setupPacket[2] = 0;       /* wValue */
setupPacket[3] = 0x1;    /* wValue [Descriptor Type: 0x1 == DEVICE] */
setupPacket[4] = 0;       /* wIndex */
setupPacket[5] = 0;       /* wIndex */
setupPacket[6] = 0x12;   /* wLength  [Size for the returned buffer] */
setupPacket[7] = 0;       /* wLength */
```

- 次のサンプルでセットアップ パケットをコントロール パイプに送る方法を示します (`GET` 命令。デバイスは、`pBuffer` 値で要求された情報を返します)。

```
WDU_TransferDefaultPipe(hDev, TRUE, 0, pBuffer, dwSize,
bytes_transferred, &setupPacket[0], 10000);
```

- 次のサンプルでセットアップ パケットをコントロール パイプに送る方法を示します (`SET` 命令)。

```
WDU_TransferDefaultPipe(hDev, FALSE, 0, NULL, 0, bytes_transferred,
   &setupPacket[0], 10000);
```

`WDU_TransferDefaultPipe()` および `WDU_Transfer()` についての詳細は、付録を参照してください。

9.5 機能 USB データ転送

9.5.1 機能 USB データ転送の概要

機能 USB データ交換は、デバイスからまたはデバイスへのデータの転送に使用されます。バルク転送、割り込み転送、等時性 (Isochronous) 転送の 3 種類の USB データ転送があります。詳細は、セクション 3.6.2 から 3.6.4 までを参照してください。

機能 USB データ転送は、次のセクションで説明するように、シングル ブロッキング転送とストリーミング転送の 2 つの方法で実装できます。WinDriver は、どちらの方法もサポートしています。DriverWizard で生成される USB コード [5.3.5] と WinDriver/util/usb_diag.exe ユーティリティ [1.10.2] (WinDriver/samples/usb_diag ディレクトリにソース コードがあります) により、ユーザーは実行する転送タイプを選択できます。

9.5.2 シングル ブロッキング転送

シングル ブロッキング USB データ転送では、ホストからの要求ごとに (シングル転送)、ホストとデバイス間でデータのブロックが同期転送 (ブロッキング) されます。

9.5.2.1 WinDriver でのシングル ブロッキング転送の実行

WinDriver の WDU_Transfer() 関数、WDU_TransferBulk() 簡易関数、WDU_TransferIsoch() 簡易関数、および WDU_TransferInterrupt() 簡易関数 (各関数の詳細は、セクション A.3.7 を参照) により、簡単にシングル ブロッキング USB データ転送を実装することができます。また、セクション 5.2 に示すように、(WDU_Transfer() 関数を使用する) DriverWizard ユーティリティを使用して、シングル ブロッキング転送を実行することもできます。

9.5.3 ストリーミング データ転送

ストリーミング USB データ転送では、ホスト ドライバによって割り当てられた内部バッファ (ストリーム) を使用して、ホストとデバイス間で継続的にデータをストリーミングします。

ストリーム転送では、ホストとデバイス間のシーケンシャル データ フローが可能で、複数の関数呼び出しやユーザーモードとカーネルモード間のコンテキスト スイッチを起因とするシングル ブロッキング転送のオーバーヘッドを削減します。ホストとデバイス間のデータ フロー ギャップにより、ホストが読み取る前にデータを上書きする可能性がある小さなデータ バッファを使用するデバイスでは特に役立ちます。

9.5.3.1 WinDriver でのストリーミングの実行

セクション A.3.8 に示すように、WinDriver の **`WDU_StreamXXX()`** 関数により、USB ストリーミング データ転送を実装できます。

ストリーム転送を実行するには、**`WDU_StreamOpen()`** 関数を呼び出します。この関数が呼び出されると、WinDriver は、指定されたデータ パイプ用の新しいストリーム オブジェクトを作成します。コントロール パイプ (Pipe 0) を除く、すべてのパイプのストリームを開くことができます。ストリームのデータ転送方向 (読み取り/書き込み) は、パイプの方向により決定されます。

WinDriver は、ブロッキング ストリーム転送とノンブロッキング ストリーム転送の両方をサポートしています。この関数の `fBlocking` パラメータは、実行する転送タイプを指定します (下記の説明を参照)。これ以降、ブロッキング転送を実行するストリームはブロッキング ストリームとし、ノンブロッキング転送を実行するストリームはノンブロッキング ストリームとします。

関数の `dwRxTxTimeout` パラメータは、ストリームとデバイス間のタイムアウトを指定します。

ストリームを開いたら、**`WDU_StreamStart()`**を呼び出して、ストリーム データ バッファとデバイス間のデータ転送を開始します。

読み取りストリームの場合、ドライバはあらかじめ定義されたブロック サイズ (**`WDU_StreamOpen()`** 関数の `dwRxSize` パラメータで指定) で、デバイスからストリーム バッファへ定期的にデータを読み取ります。書き

込みストリームの場合、ドライバは定期的にストリーム データ バッファを確認して、検出したデータをデバイスに書き込みます。

読み取りストリームからユーザーモードのホスト アプリケーションへデータを読み取るには、**WDU_StreamRead()**を呼び出します。

ブロッキング ストリームの場合、この関数はアプリケーションにより要求されたすべてのデータがストリームからアプリケーションに転送されるか、ストリームによるデバイスからのデータの読み取りがタイムアウトになるまでブロックします。

ノンブロッキング ストリームの場合、この関数は要求されたデータをできるだけ多く (ストリーム データ バッファにある利用可能なデータ量により異なる) アプリケーションに転送して、直ちにリターンします。

ユーザーモードのホスト アプリケーションから書き込みストリームへデータを書き込むには、**WDU_StreamWrite()**を呼び出します。

ブロッキング ストリームの場合、この関数はストリームにすべてのデータが書き込まれるか、またはストリームによるデバイスへのデータの書き込みがタイムアウトになるまでブロックします。

ノンブロッキング ストリームの場合、この関数はできるだけ多くのデータをストリームに書き込み、直ちにリターンします。

ブロッキング転送およびノンブロッキング転送では、read/write 関数はストリームと呼び出しアプリケーション間で実際に転送されたバイト数を、出力パラメータ `*pdwBytesRead` / `*pdwBytesWritten` に格納して返します。

ストリーム データ バッファのすべてのコンテンツをデバイスに書き込み (書き込みストリームの場合)、ストリームのすべての待機中 I/O が処理されるまでブロックする **WDU_StreamFlush()** 関数を呼び出して、いつでもアクティブなストリームをフラッシュできます。

ブロッキング ストリームとノンブロッキング ストリームの両方をフラッシュできます。

アクティブなストリームとデバイス間のデータ ストリーミングを停止するには、**WDU_StreamStop()**を呼び出します。書き込みストリームの場合、この関数はストリームを停止する前にフラッシュ (ストリームのコンテンツをデバイスに書き込むなど) します。

開いているストリームは、閉じられるまでいつでも停止/再開できます。

開いているストリームを閉じるには、**WDU_StreamClose()**を呼び出します。この関数はストリームを閉じる前に、ストリームを停止し、データをデバイスにフラッシュします (書き込みストリームの場合)。

注意: 必要なクリーンアップ処理を行うために、**WDU_StreamOpen()** への各呼び出しに対して、それ以降のコードで対応する **WDU_StreamClose()** を呼び出す必要があります。

9.6 64 ビット OS のサポート

9.6.1 64 ビット アーキテクチャのサポート

WinDriver は 次の 64 ビット プラットフォームをサポートしています。

- Solaris SPARC (64 ビット) (PCI の場合のみ)。WinDriver がサポートする Solaris プラットフォームの一覧は、セクション 4.1.5 を参照してください。

- Linux AMD64 またはインテル 64 (**x86_64**)。WinDriver がサポートする Linux プラットフォームの一覧は、セクション 4.1.4 を参照してください。

- Windows AMD64 またはインテル 64 (**x64**)。WinDriver がサポートする Windows プラットフォームの一覧は、セクション 4.1.1 および 4.1.2 を参照してください。

WinDriver での 64 ビット データ転送 (32 ビット プラットフォームでのデータ転送も含む) については、セクション 10.2.3 を参照してください。

9.6.2 64 ビット アーキテクチャでの 32 ビット アプリケーションのサポート

WinDriver Solaris SPARC (64 ビット) (PCI の場合のみ)、Linux AMD64、Windows AMD64 は、32 ビット アプリケーションと 64 ビット アプリケーションの両方をサポートしています。これらのプラットフォーム向けに 32 ビット アプリケーションをビルドする場合は、適切な 32 ビット コンパイラで `-DKERNEL_64BIT` コンパイル フラグを使用します。ただし、64 ビット アプリケーションの方がより効果的です。

9.6.3 64 ビットおよび 32 ビットのデータ型

一般的に、DWORD は unsigned long です。DWORD は、32 ビット コンパイラでは 32 ビットとして処理されますが、64 ビット コンパイラでは処理が異なります。Windows ベースの 64 ビット コンパイラでは、32 ビットとして処理され、UNIX ベースの 64 ビット コンパイラ (例: GCC、SUN Forte) では 64 ビットとして処理されます。32 ビット アドレスまたは 64 ビット アドレスを参照する場合は、コンパイラ依存問題を回避するために、クロス プラットフォームな UINT32 および UINT64 を使用してください。

9.7 バイト オーダー

9.7.1 エンディアンネスとは

メモリ ストレージを処理するには主に 2 つのアーキテクチャがあります。ビッグ エンディアンとリトル エンディアンと呼ばれ、メモリに格納されるバイトの順番を表します。

- ビッグ エンディアンとは、最下位メモリ アドレスにマルチ バイト データ フィールドの最上位バイトから順番に格納します。

  これは、0x1234 のような 16 進文字列を (0x12 0x34) としてメモリに格納します。ビッグ エンドまたは上位エンドを初めに格納します。4 バイトの値について次のことが同じように当てはまります。たとえば、0x12345678 は、(0x12 0x34 0x56 0x78) と順番に格納されます。

- リトル エンディアンとは、最下位メモリ アドレスにマルチ バイト データ フィールドの最下位バイトから順番に格納します。

  これは、0x1234 のような 16 進文字列を (0x34 0x12) としてメモリに格納します。リトル エンドまたは下位エンドを初めに保存します。4 バイトの値について次のことが同じように当てはまります。たとえば、0x12345678 は、(0x78 0x56 0x34 0x12) と順番に格納されます。

すべてのプロセッサはビッグ エンディアンまたはリトル エンディアンのいずれかでデザインされています。Intel x86 系プロセッサはリトル エンディアンを採用しています。Sun SPARC、Motorola 68K および PowerPC ファミリはすべてビッグ エンディアンを採用しています。

エンディアンネスの違いによって、コンピュータが異なるフォーマットで記述されたバイナリ データを共有メモリまたはファイルから読み込む場合に、問題が発生する場合があります。

ビッグ エンディアンとリトル エンディアンの名前の由来は、小説「ガリバー旅行記」(Jonathan Swift 1726) の小人国の話から来ており、ゆで卵を小さい方から割るか、大きい方から割るかの対立を描いています。

9.7.2 WinDriver のバイト オーダー マクロ

x86 アーキテクチャに対応するようにリトル エンディアンとして PCI バスをデザインしています。PCI バスと SPARC、PowerPC のアーキテクチャ間のバイト オーダーの違いから問題が生じないように、WinDriver には、リトル エンディアンとビッグ エンディアン間でデータを変換するマクロ定義が含まれています。

WinDriver を使用してドライバを開発した場合、これらのマクロ定義によって、クロス プラットフォーム間で互換性が有効になります。これらのマクロを使用することによって、安全に x86 アーキテクチャにドライバを配布できます。

以下のセクションでは、そのマクロの説明と使用方法を紹介します。

9.7.3 PCI ターゲット アクセスのマクロ

PCI デバイスのメモリ マップされた領域を使用する PCI カードから読み込みまたは PCI カードへ書き込みを行う際に、エンディアンネスを変換するために PCI ターゲット アクセスの WinDriver のマクロを使用します。

注意: これらのマクロ定義は Linux PowerPC アーキテクチャに当てはまります。

- **`dtoh16`** - WORD (device to host) 変換用のマクロ定義
- **`dtoh32`** - DWORD (device to host) 変換用のマクロ定義
- **`dtoh64`** - QWORD (device to host) 変換用のマクロ定義

以下の場合に WinDriver のマクロ定義を使用します。

1. メモリ マップされた領域を使用してカードへ直接書き込みアクセスをする場合、デバイスへ書き込むデータにマクロを適応する場合。

   たとえば:
   ```
   DWORD data = VALUE; *mapped_address = dtoh32(data);
   ```

2. メモリ マップされた領域を使用してカードから直接読み込みアクセスをする場合、デバイスから読み込むデータにマクロを適応する場合。

   たとえば:
   ```
   WORD data = dtoh16(*mapped_address);
   ```

   注意: WinDriver API (`WDC_Read/WriteXXX()`関数、`WDC_MultiTransfer()`関数、低水準 `WD_Transfer()`関数および低水準 `WD_MultiTransfer()` 関数は必要なバイト オーダー変換を実行するため、これらの API を使用してメモリ アドレスの読み取り/書き込みを行う場合は、`dtoh16/32/64()` マクロを使用してデータを変換する必要はありません (I/O アドレスについても同様)。

9.7.4 PCI マスター アクセスのマクロ

PCI マスタ デバイスがアクセスするホスト メモリ内のデータのエンディアンネスを変換するために PCI マスタ アクセスの WinDriver のマクロを使用します。つまり、ホストではなくデバイスからアクセスする場合。

注意: これらのマクロの定義は Linux PowerPC と SPARC アーキテクチャの両方に当てはまります。

- **htod16** - WORD (host to device) 変換用のマクロの定義
- **htod32** - DWORD (host to device) 変換用のマクロ定義
- **htod64** - QWORD (host to device) 変換用のマクロ定義

以下の場合に WinDriver のマクロ定義を使用します。

カードで読み込み/書き込みを行うホスト メモリ上で準備したデータにマクロを適応する場合。そのような場合の例は、scatter/gather DMA 用の一連の記述子です。

以下の例は、WinDriver ライブラリ (**WinDriver/plx/lib/plx_lib.c** を参照) の **PLX_DMAOpen()** 関数から抜粋したサンプルです:

```
/* setting chain of DMA pages in the memory */
for (dwPageNumber = 0, u32MemoryCopied = 0;
   dwPageNumber < pPLXDma->pDma->dwPages;
   dwPageNumber++)
{
  pList[dwPageNumber].u32PADR =
      htod32((UINT32)pPLXDma->pDma->
      Page[dwPageNumber].pPhysicalAddr);
  pList[dwPageNumber].u32LADR =
      htod32((u32LocalAddr + (fAutoinc ? u32MemoryCopied : 0)));
  pList[dwPageNumber].u32SIZ =
      htod32((UINT32)pPLXDma->pDma->Page[dwPageNumber].dwBytes);
  pList[dwPageNumber].u32DPR =
      htod32((u32StartOfChain + sizeof(DMA_LIST) * (dwPageNumber + 1))
      | BIT0 | (fIsRead ? BIT3 : 0));
      u32MemoryCopied += pPLXDma->pDma->Page[dwPageNumber].dwBytes;
}

pList[dwPageNumber - 1].u32DPR |= htod32(BIT1); /* Mark end of chain */
```

第 10 章
パフォーマンスの向上

10.1 概要

ユーザー モードドライバの開発を終了した時点で、コード内のモジュールが必要なパフォーマンスを発揮していないことに気づいたとします (たとえば、割り込み処理や I/O マップされた領域へのアクセスなど)。この場合、以下のいずれかの方法を使ってパフォーマンスを向上させることが可能です。

- ユーザー モードドライバのパフォーマンスを向上させる。[10.2]
- Kernel PlugIn ドライバ [第 11 章] を作成し、パフォーマンスを向上させる必要があるコードを Kernel PlugIn に移動させる。

注意: Windows CEにはカーネル モードとユーザー モードの区別がないため Kernel PlugIn を実行できませんが、Kernel PlugIn を使用しなくても最高のパフォーマンスに向上できます。Windows CE で割り込み処理率を向上するには、セクション [9.2.3] を参照してください。

次の チェックリストを利用して、どのようにドライバのパフォーマンスを向上させるかを検討してください。

10.1.1 パフォーマンスを向上させるためのチェックリスト

次のチェックリストを使用してドライバのパフォーマンス向上に役立ててください。

	問題	解決方法
#1	ISA カード - カード上の I/O にマップした領域へのアクセス	• 複数の WD_Transfer を WD_MultiTransfer に変更する (セクション 10.2.2 の「I/O にマップした領域へのアクセス」を参照してください)。大量のデータを転送する場合は、ブロック (文字列) 転送を使用したり、いくつかのデータ転送関数の呼び出しを 1 つのマルチ転送関数の呼び出しにまとめます (10.2.2 を参照)。 • 問題が解決しない場合、Kernel PlugIn ドライバを用意して I/O をカーネル モードでハンドルする (詳しくは、第 11 章 および第 12 章 の Kernel Plugin の説明を参照してください)。
#2	PCI カード - カード上の I/O にマップした領域へのアクセス	• まず、PCI 設定レジスタのアドレス スペースのビット 0 を 0 に変更してカードを I/O マップからメモリ マップに変更し、問題 #3 の解決方法を試す。BAR0、1、2、3、4、5 レジスタを異なる値で初期化するように EERPOM を再プログラムする必要があります。 • 上の方法で問題が解決しない場合、問題 #1 の解決方法を試す。
#3	カード上のメモリにマップした領域へのアクセス	• 関数を使用せずにメモリ直接アクセスする (セクション 10.2.1 を参照)。大量のデータを転送する場合には、上記の #1 も参考にしてください。 • これで問題が解決しない場合、ハードウェアのデザインに問題があると考えられます。ソフトウェア デザインの変更や Kernel PlugIn を利用してもパフォーマンスを向上できません。
#4	割り込み (割り込みの受け取りもれ、割り込みの受け取りが遅すぎる場合)	Kernel PlugIn ドライバを利用して割り込みをカーネル モードで処理する必要があります (詳しくは、第 11 章 および第 12 章 の Kernel Plugin の説明を参照してください)。

10.2 ユーザー モード ドライバのパフォーマンスの向上

一般的にメモリにマップされた領域への転送は、I/O にマップされた領域への転送よりも高速です。これは、WinDriver では関数を呼び出さずに、ユーザー モードからメモリにマップされた領域に直接アクセスできるためです (10.2.1 を参照)。

また、WinDriver API ではブロック (文字列) 転送を使用したり、いくつかのデータ転送関数の呼び出しを 1 つのマルチ転送関数の呼び出しにまとめて (10.2.2 を参照)、I/O およびメモリ データ転送のパフォーマンスを向上できます。

10.2.1 メモリにマップされた領域への直接アクセス

`WDC_xxxDeviceOpen()` 関数 (PCI / PCMCIA / ISA) または低水準 `WD_CardRegister()` 関数で PCI/PCMCIA/ISA を登録すると、ユーザー モードおよびカーネル モードにおけるカードの物理メモリ領域

へのマップが返されます。これらのアドレスを使用して、どちらのモードからでもカード上のメモリ領域へ直接アクセスできます。これにより、ユーザー モードとカーネル モード間のコンテキスト スイッチとメモリへアクセスするための関数呼び出しオーバーヘッドが削除されます。

WDC_MEM_DIRECT_ADDR マクロは、カード上の指定されたメモリ アドレス領域に直接アクセスするためのベース アドレス (ユーザー モードから呼び出された場合はユーザー モード マップ、Kernel PlugIn ドライバ [第 11 章] から呼び出された場合はカーネル モード マップ) を返します。マップされたベース アドレスを指定されたメモリ領域内のオフセットと共に WDC_ReadMem8/16/32/64 マクロや WDC_WriteMem8/16/32/64 マクロへ渡し、ユーザー モードまたはカーネル モードから、カード上のメモリ アドレスに直接アクセスすることができます。

さらに、WDC_ReadAddrBlock() と WDC_WriteAddrBlock() を除く WDC_ReadAddrXXX() 関数および WDC_WriteAddrXXX() 関数はすべて、呼び出しコンテキスト (ユーザー モード/カーネル モード) のマップを使用して、直接メモリ アドレスにアクセスします。

低水準 WD_xxx() APIを使用すると、WD_CardRegister() により、カード リソース アイテムの構造体である pCardReg->Card.Item[i] の dwTransAddr フィールドおよび dwUserDirectAddr フィールドに保存されている、カードの物理メモリ領域のユーザー モード マップおよびカーネル モード マップが返されます。dwTransAddr は、WD_Transfer() や WD_MultiTransfer() の呼び出しにおいて、または Kernel PlugIn ドライバ [第 11 章] から直接メモリにアクセスする際に、ベース アドレスとして使用します。ユーザー モードから直接メモリにアクセスするには、dwUserDirectAddr を通常のポインタとして使用します。

どの方法でカード上のメモリにアクセスする場合でも、特に文字列転送コマンドを使用する際には、データ型のサイズに応じてベース アドレスのアラインが重要です。または、転送を小さく分割します。データをアラインする最も簡単な方法は、バッファを定義する際に、基本の型を使用することです。

例:

```
BYTE buf[len];   /* BYTE 転送の場合 ― アラインなし */
WORD buf[len];   /* WORD 転送の場合 ― 2 バイト境界でアライン */
UINT32 buf[len]; /*DWORD 転送の場合 ― 4 バイト境界でアライン */
UINT64 buf[len]; /* QWORD 転送の場合 ― 8 バイト境界でアライン */
```

10.2.2 ブロック転送および複数の転送のグループ化

メモリ アドレスや (メモリ アドレスとは異なり、直接アクセスすることができない) I/O アドレス から、またはメモリ アドレスや I/O アドレスへ大量のデータを転送するには、次の方法を使用して関数呼び出しオーバーヘッドおよびユーザー モードとカーネル モード間のコンテキスト スイッチを削減し、パフォーマンスを向上させます。

- WDC_ReadAddrBlock() / WDC_WriteAddrBlock() または低水準 WD_Transfer() 関数を使用してブロック (文字列) 転送を行う。
- WDC_MultiTransfer() または低水準 WD_MultiTransfer() 関数を使用して複数の転送を 1 つの関数呼び出しにまとめる。

10.2.3 64-ビット データ転送を行う

注意: 実際に 64 ビット転送を実行の能力は、ハードウェア、CPU、ブリッジなどによる転送のサポートに依存し、これらの仕様の組合せになどのよっても影響を受けます。

WinDriver は、サポートしている 64 ビット Windows、Linux、Solaris プラットフォーム (一覧はセクション 9.6 を参照) および 32 ビット Windows、Linux、Solaris x86 プラットフォーム上での 、64 ビット PCI データ転送をサポートしています。PCI ハードウェア (カードおよびバス) が 64 ビットの場合、対象のホスト オペレーティング システムが 32 ビットであっても、より高い処理能力をハードウェアに与えることができます。

この新しい技術は、32 ビットプラットフォームで以前では実現できなかったデータ転送の能力を飛躍的に高めます。DDK または他のドライバ開発ツールで記述したドライバよりも高いパフォーマンスを WinDrinver で開発したドライバが発揮します。Jungo 社によるベンチマーク テストでは、64 ビット データ転送で 32 ビット データ転送に比べ、データ転送速度が飛躍的に向上する結果が得られました。WinDriver で開発されたドライバは、通常の 32 ビット データ転送で得られる性能よりも高い数字を得られることが実証されました。

次の方法で 64 ビット データ転送を実行できます。

- `WDC_ReadAddr64()`または `WDC_WriteAddr64()` を呼び出す。
- アクセス モード `WDC_SIZE_64` と共に `WDC_ReadAddrBlock()`または `WDC_WriteAddrBlock()` を呼び出す。
- `QWORD` 読み取り/書き込み転送コマンドと共に `WDC_MultiTransfer()`、低水準 `WD_Transfer()`、または `WD_MultiTransfer()` を呼び出す (詳細は、各関数の説明を参照してください)。

`WDC_PciReadCfg64()` / `WDC_PciWriteCfg64()` または `WDC_PciReadCfgBySlot64()` / `WDC_PciWriteCfgBySlot64()` を使用して、PCI 設定空間からまたは PCI 設定空間への 64 ビット転送を実行することもできます。

第 11 章
Kernel PlugIn について

この章では、WinDriver の Kernel PlugIn の機能について説明します。

注意: Windows CEにはカーネル モードとユーザー モードの区別がないため Kernel PlugIn を実行できませんが、Kernel PlugIn を使用しなくても最高のパフォーマンスに向上できます。
Windows CE で割り込み処理率を向上するには、セクション 9.2.3 を参照してください。

11.1 Kernel PlugIn の概要

ユーザー モードで作成されたドライバは、カーネル からユーザー モードへの関数呼び出しにかなりの量のオーバーヘッドがあることも事実であり、必要なパフォーマンスが得られない場合もあります。そのような場合、コードには手を加えず Kernel PlugIn 機能を使用し、ドライバ コードの問題となるセクションをカーネルへ移すことが可能です。WinDriver の Kernel PlugIn 機能を使用すると、性能を低下させることなくドライバが動作します。

Kernel PlugIn を利用した場合の利点を次に示します。

- すべてのドライバ コードをユーザー モードで開発、デバッグが可能です。
- カーネル モードに移されたコード セグメントは本質的に変更されていないため、カーネル デバッグの必要がありません。
- Kernel PlugIn を使ってカーネルで動作するコードは、プラットフォームに依存しないため、WinDriver および Kernel PlugIn がサポートするすべてのプラットフォームで動作します。一般的なカーネル モードのドライバは、特定のプラットフォームでしか動作しません。

WinDriver の Kernel PlugIn 機能を使用することにより、パフォーマンスを低下させずにドライバを作成できます。

11.2 Kernel PlugIn を作成する前に

すべてのパフォーマンスに関する問題を Kernel PlugIn で解決する必要はありません。問題によっては、WinDriver に用意されている関数をうまく組み合わせることによって、ユーザー モードでパフォーマンスの向上が可能です。詳細は、第 10 章 の「パフォーマンスの向上」を参照してください。

11.3 期待される効果

WinDriver Kernel PlugIn を使用して割り込み処理を作成できるため、1 秒間に約 100,000 の割り込みを逃すことがなく、すべて処理可能です。

11.4 開発プロセスの概要

WinDriver の Kernel PlugIn を使用する際に、まず標準の WinDriver ツールを使用してユーザー モードでドライバを開発およびデバッグします。パフォーマンスに影響する部分のコード (割り込み処理、I/O マップされたメモリ範囲へのアクセスなど) を検出し、カーネル モードで実行する Kernel PlugIn を作成します。パフォーマンスに影響する部分のコードを Kernel PlugIn ドライバへ移します。これにより呼び出しのオーバーヘッドおよびユーザー モードで同じタスクを実装する時に発生するコンテキスト スイッチを削除します。

このユニークなアーキテクチャで、開発期間を短縮し、パフォーマンスの低下を防げます。

11.5 WinDriver Kernel PlugIn の構造

11.5.1 構造の概要

ユーザー モードで記述したドライバは、デバイスにアクセスする際に WinDriver の関数 (`WDC_xxx` および/または `WD_xxx`) を使用します。ユーザー モードで実装され、カーネル レベルのパフォーマンスの達成が必要な関数 (割り込み処理など) の場合、WinDriver Kernel PlugIn に移します。通常、ユーザー モードと Kernel PlugIn の両方で同じ WinDriver API をサポートしているので、コードの修正をせずに、ユーザー モードからカーネルへ `WDC_xxx` / `WD_xxx` 関数呼び出しを使用するようにコードを移行できます。

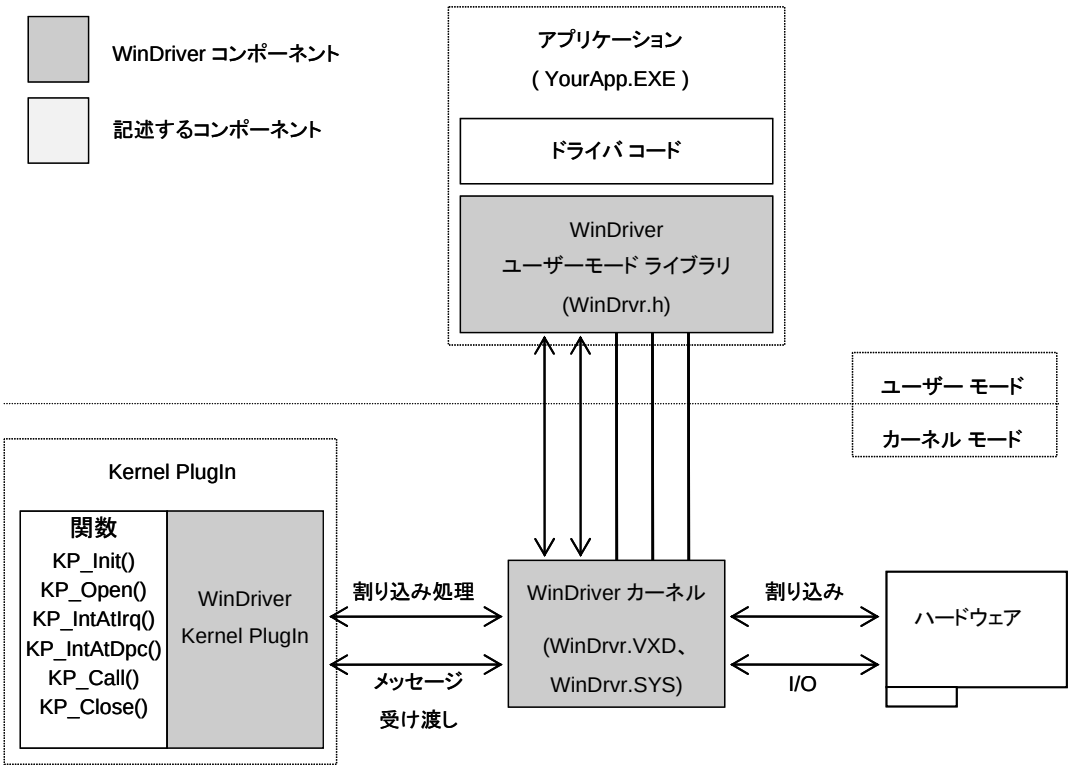


図 11.1: KernelPlugIn の構造

11.5.2 WinDriver Kernel と Kernel Plugin のインターフェイス

WinDriver カーネルと WinDriver Kernel PlugIn は次の 2 種類のインターフェイスがあります。

1. **割り込み処理:** WinDriver が割り込みを受け取ると、ユーザー モードの割り込み処理をデフォルトで有効にします。しかし、割り込みを Kernel PlugIn ドライバが処理するように設定し、WinDriver が割り込みを受信すると、Kernel PlugIn ドライバのカーネル モードで割り込み処理が始動します。Kernel PlugIn 割り込み処理は、基本的に Kernel PlugIn へ移動する前にユーザー モードで割り込み処理を記述およびデバッグしていたコードと同様ですが、ユーザー モードのコードの一部を編集する必要があります。KernelPlugIn で割り込みの検知および処理を行うコードを再記述し、KernelPlugIn の柔軟性を有効にします (セクション 11.6.5 を参照してください)。

2. **メッセージ受け渡し:** カーネル モードで関数を実行する場合 (I/O 処理関数など)、ユーザー モードのドライバは WinDriver Kernel PlugIn に「メッセージ」を渡します。このメッセージは特定の関数にマップされ、カーネル内で実行されます。この関数はユーザー モードで開発されたものと同じコードが含まれます。
   ユーザー モード アプリケーションから Kernel PlugIn ドライバへメッセージを使用してデータを渡すこともできます。

11.5.3 Kernel Plugin コンポーネント

Kernel PlugIn の開発サイクルを終了すると、作成したドライバは以下のエレメントを持つことになります。

- `WDC_xxx` / `WD_xxx API` 関数で記述されたユーザー モード ドライバ アプリケーション(**<アプリケーション名>/.exe**)。
- WinDriver カーネル モジュール (**windrvr6/.sys/.o**)。
- カーネル レベルへ移動したドライバ機能を含む `WDC_xxx` / `WD_xxx` API 関数で記述された Kernel PlugIn ドライバ (**<Kernel PlugIn ドライバ名>/.sys/.o**)

11.5.4 Kernel PlugIn イベント シーケンス

次に Kernel PlugIn で実装できるすべての関数の一般的なイベント シーケンスを示します。

11.5.4.1 ユーザー モードから Kernel PlugIn へのハンドルを開く

イベント / コールバック	備考
イベント: Windows は Kernel PlugIn ドライバをロードします。	このイベントはブート時にダイナミック ロードにより行われるか、またはレジストリからの指示として行われます。

コールバック: KP_Init() Kernel PlugIn ルーチンを呼び出します。	KP_Init() が WinDriver に KP_Open() ルーチンの名前を知らせます。アプリケーションがドライバを開く場合 (Kernel PlugIn ドライバを開く名前で WDC_xxxDeviceOpen() を呼び出す場合か、または (ラッパー WDC_xxxDeviceOpen() 関数に呼び出される) 低水準の WD_KernelPlugInOpen() 関数を呼び出す場合)、WinDriver はこのルーチンを呼び出します。
イベント: ユーザー モード ドライバ アプリケーションは Kernel PlugIn ドライバを開く名前で WDC_xxxDeviceOpen() を呼び出すか、または (ラッパー WDC_xxxDeviceOpen() 関数に呼び出される) 低水準の WD_KernelPlugInOpen() 関数を呼び出します。	
コールバック: KP_Open() Kernel PlugIn ルーチンを呼び出します。	KP_Open() 関数は WinDriver への Kernel PlugIn ドライバで実装した全コールバック関数名の通知に使用されます。また、必要に応じて Kernel Plugin ドライバの開始に使用されます。

11.5.4.2 Kernel PlugIn からのユーザー モード要求処理

イベント / コールバック	備考
イベント: アプリケーションは WDC_callKerPlug()、または 低水準の WD_KernelPlugInCall() 関数を呼び出します。	アプリケーションは WDC_CallKerPlug() / WD_KernelPlugInCall() を呼び出し、(Kernel PlugIn ドライバの) カーネル モードでコードを実行します。アプリケーションは Kernel PlugIn ドライバへメッセージを渡します。Kernel PlugIn ドライバは送られたメッセージに従って実行するコードを選択します。
コールバック: KP_Call() Kernel PlugIn ルーチンを呼び出します。	KP_Call() はユーザー モードより渡されたメッセージに従ってコードを実行します。

11.5.4.3 割り込み処理の有効化/無効化および高い割り込み要求処理

イベント / コールバック	備考
イベント: アプリケーションは fUseKP 引数に TRUE を設定して WDC_IntEnable() を呼ぶか (Kernel PlugIn でデバイスを開いた後)、または、	

KernelPlugIn ドライバへのハンドルでより低レベルな `InterruptEnable()` または `WD_IntEnable()`関数を呼びます (関数へ渡された `WD_INTERRUPT` 構造体の `hKernelPlugIn` フィールドに設定)。	
コールバック: `KP_IntEnable()` Kernel PlugIn ルーチンを呼び出します。	この関数には Kernel PlugIn の割り込み処理に必要な初期化設定を含めてください。
イベント: ハードウェアが割り込みを発生します。	
コールバック: `KP_IntAtIrql()` Kernel PlugIn ルーチンを呼び出します。	`KP_IntAtIrql()` は高い優先度で実行されるため、基本的な割り込み処理 (割り込みを識別するために、レベル センシティブ割り込みの HW 割り込みシグナルの低くするなど) だけを実行します。 より多くの割り込み処理が必要な場合、`KP_IntAtDpc()` 関数で追加処理を引き継ぐために `KP_IntAtIrql()` は TRUE を返します。
イベント: 割り込みが Kernel PlugIn で有効になっている場合 (割り込みを有効にするイベントの詳細を参照) 、アプリケーションは `WDC_IntDisable()` を呼び出すか、または、低水準の `InterruptDisable()` または `WD_IntDisable()` 関数を呼び出します。	
コールバック: `KP_IntDisable()` Kernel PlugIn ルーチンが呼び出されます。	この関数は `KP_IntEnable()` コールバックにより割り当てられたメモリを解放します。

11.5.4.4 割り込み処理 – 異なる処理の呼び出し

イベント / コールバック	備考
イベント: Kernel PlugIn `KP_IntAtIrql()` 関数が `TRUE` を戻します。	カーネルで引き継いだ手順として追加の割り込み処理を WinDriver へ伝えます。
コールバック: `KP_IntAtDpc()` Kernel PlugIn ルーチンを呼び出します。	残りの割り込みコードを処理しますが `KP_IntAtIrql()` よりは優先度が低いです。
イベント: `KP_IntAtDpc()` は 0 よりも大きい値を戻します	ユーザー モードで処理するための割り込みコードが必要です。

コールバック: WD_IntWait() を戻します。	ユーザー モード割り込みハンドラから実行が再開されます。

11.5.4.5 Plug-and-Play およびパワー マネージメント

イベント / コールバック	備考
イベント: アプリケーションは fUseKP 引数に TRUE を設定して WDC_EventRegister() を呼んで、Kernel PlugIn ドライバを使用して、Plug-and-Play およびパワー マネージメントの通知を受け取るように登録します (Kernel PlugIn でデバイスを開いた後)。または、Kernel PlugIn ドライバへのハンドルでより低レベルな EventRegister() または WD_EventRegister() 関数を呼び出します (関数に渡された WD_EVENT 構造体の hKernelPlugIn フィールドに設定)。	
イベント: Plug-and-Play またはパワー マネージメント イベントが発生します。	
コールバック: KP_Event() が呼び出されます。	KP_Event() は、発生したイベントについての情報を受け取ります。
イベント: KP_Event() は TRUE を返します。	イベントは、ユーザー モード アプリケーションで処理される必要があります。
コールバック: WD_Intwait() を返します。	ユーザー モード割り込みハンドラ アプリケーション イベント ハンドラで処理を再開します。

11.6 Kernel PlugIn の仕組み

このセクションでは Kernel PlugIn の開発サイクルを説明します。

まず初めにドライバ コードを作成し、ユーザー モードでドライバ コード全体をデバッグすることを推奨します。次にパフォーマンス上の問題に直面したり、柔軟性が必要な場合、コードの一部を Kernel PlugIn ドライバへポーティングします。

11.6.1 Kernel PlugIn ドライバの作成に必要な条件

Kernel PlugIn ドライバをビルドするには以下のツールが必要です。

- Windows 98 / Me / 2000 / XP / Server 2003 / Vista の場合
  - Visual C (VC) コンパイラ (**cl.exe**、**rc.exe**、**link.exe** および **nmake.exe**)

- ターゲットとなるオペレーティング システム用の Windows デバイスドライバ開発キット (DDK) がインストールされたホスト マシン

  注意

  – Windows DDK は MSDN サブスクリプションの一部として利用できます。また、http://www.microsoft.com/whdc/ddk/winddk.mspx から購入できます。

  – ターゲット PC が **Windows 98 / Me** 用の Kernel PlugIn ドライバを開発している場合、Windows 2000 またはそれ以降をホスト プラットフォームとして使用してください。

  – **Linux** および **Solaris** の場合、**GCC**、**gmake** または **make** が必要です。

注意:必要条件ではありませんが、Kernel PlugIn ドライバを開発する場合は、2 台のコンピュータ (ホスト プラットフォーム用に 1 台、ターゲット プラットフォーム用に 1 台) を使用することを推奨します。ホスト コンピュータでドライバを開発し、ターゲット コンピュータで開発したドライバを実行しテストします。

11.6.2 Kernel PlugIn の実装

11.6.2.1 始める前に

このセクションでは、Kernel PlugIn ドライバで実装されるコールバック関数 (呼び出しイベントが発生した際に呼び出されます) について説明します。たとえば、`KP_Init()` はドライバがロードされたときに呼び出されるコールバック関数です。ロード時に実行するコードはこの関数に記述しておく必要があります。

ドライバ名は `KP_Init()` で渡されます。ここで渡された名前で実装される必要があります。その他のコールバック関数の場合、このリファレンス ガイドでは `KP_xxx()` 関数 (`KP_Open()` など) のように関数名を付けます。ただし、Kernel PlugIn ドライバを開発する際には、コールバック関数に他の名前を付けることもできます。DriverWizard で Kernel PlugIn コードを生成する際には、コールバック関数名 (`KP_Init()` 関数以外) は "**KP_<ドライバ名>_<コールバック関数>**" の形式で名前を付けます。たとえば、プロジェクト名が **MyDevice** の場合、Kernel PlugIn `KP_Open()` 関数の名前は `KP_MyDevice_Open()` となります。

11.6.2.2 KP_Init() 関数の記述

`KP_Init()` 関数は以下のようになります。

```
BOOL __cdecl KP_Init(KP_INIT *kpInit);
```

`KP_INIT` は次のような構造体になります:

```
typedef struct {
    DWORD     dwVerWD; // WinDriver Kernel PlugIn ライブラリのバージョン
    CHAR cDriverName[12]; //  デバイス ドライバ名を返します、最大 12 文字
    KP_FUNC_OPEN funcOpen; // KP_Open 関数を返します
} KP_INIT;
```

この関数はドライバがロードされるたびに呼び出されます。`KP_INIT` 構造体には `KP_Open()` 関数 (**WinDriver/samples/pci_diag/kp_pci/kp_pci.c** のサンプルを参照してください) のアドレスと Kernel PlugIn 名が格納されます。

注意:

1. Kernel PlugIn ドライバの選択する名前は、作成するドライバの名前にしてください (KP_Init() で KP_INIT 構造体の cDriverName で設定されます)。たとえば、**XXX.SYS** という名前のドライバを生成する場合、KP_INIT 構造体の cDriverName 項目に名前 "XXX" を設定します。

2. ユーザー モードで設定されたドライバ名、WDC_xxxDeviceOpen() の呼び出しで設定されたドライバ名、低水準 WD_KernelPlugInOpen()関数へ渡された WD_KERNEL_PLUGIN 構造体の pcDriverName フィールド (WDC ライブラリを使用していない場合) で設定されたドライバ名、および KP_Init() へ渡された KP_INTI 構造体の cDriverName フィールドで設定されたドライバ名が同一であるかを確認してください。

   ユーザー モード アプリケーションと Kernel PlugIn ドライバで共有するヘッダー ファイルでドライバ名を定義し、関連したすべての場所で定義した値を使用すると実装することができます。

KP_PCI サンプルから抜粋 (WinDriver/samples/pci_diag/kp_pci/kp_pci.c):

```
/* KP_Init is called when the Kernel PlugIn driver is loaded.
   This function sets the name of the Kernel PlugIn driver and the driver's
   open callback function. */
BOOL __cdecl KP_Init(KP_INIT *kpInit)
{
    /* Verify that the version of the WinDriver Kernel PlugIn library
        is identical to that of the windrvr.h and wd_kp.h files */
    if (WD_VER != kpInit->dwVerWD)
    {
       /* Re-build your Kernel PlugIn driver project with the compatible
           version of the WinDriver Kernel PlugIn library
(kp_nt<version>.lib)
           and windrvr.h and wd_kp.h files */
       return FALSE;
    }

    kpInit->funcOpen = KP_PCI_Open;
    strcpy (kpInit->cDriverName, KP_PCI_DRIVER_NAME);

    return TRUE;
}
```

ドライバ名はプリプロセッサ名を使用して設定されます。この定義は、**pci_diag** のユーザー モード アプリケーションおよび **KP_PCI** Kernel PlugIn ドライバで共有される **WinDriver/samples/pci_diag/pci_lib.h** ヘッダー ファイルに保存されています。

```
/* Kernel PlugIn driver name (should be no more than 8 characters) */
#define KP_PCI_DRIVER_NAME "KP_PCI"
```

11.6.2.3 KP_Open() 関数の記述

KP_Open() 関数は以下のようになります。

```
BOOL __cdecl KP_Open(KP_OPEN_CALL *kpOpenCall, HANDLE hWD,
    PVOID pOpenData, PVOID *ppDrvContext);
```

ユーザー モード アプリケーションが Kernel PlugIn ドライバの名前で WDC_xxxDeviceOpen() を呼び出す際か、または 低水準の WD_KernelPlugInOpen() 関数 (ラッパー WDC_xxxDeviceOpen() 関数で呼び出された場合) を呼び出す際に、このコールバックを呼び出します。

KP_Open() 関数には、Kernel PlugIn に組み込むコールバックを定義してください。

次に組み込み可能なコールバックを示します。

コールバック	機能
KP_Close()	ユーザー モード アプリケーションが Kernel PlugIn ドライバで開くデバイス用の WDC_xxxDeviceClose() を呼び出す場合か、または、(ラッパー WDC_xxxDeviceClose() 関数で呼び出される) 低水準の WD_KernelPlugInClose() 関数を呼び出す場合に呼び出されます。
KP_Call()	ユーザー モード アプリケーションが WDC_CallKerPlug() 関数または低水準の (ラッパー WDC_CallKerPlug() 関数で呼び出される) WD_KernelPlugInCall() 関数を呼び出した場合に呼び出されます。 この関数は Kernel PlugIn メッセージ ハンドラを実装します。
KP_Event()	Plug-and-Play および パワーマネージメント イベントが発生した場合に呼び出されるか、または fUseKP 引数に TRUE を設定して WDC_EventRegister() を呼ぶか (Kernel PlugIn でデバイスを開いた後)、Kernel PlugIn ドライバへのハンドルで低水準の EventRegister() または WD_EventRegister() を呼んで (関数へ渡される WD_EVENT 構造体の hKernelPlugIn フィールドで設定)、Kernel PlugIn のこのイベントの通知を受け取るように予め登録したユーザーモード アプリケーションを指定します。
KP_IntEnable()	ユーザー モード アプリケーションが Kernel PlugIn の割り込みを有効にした場合、(Kernel PlugIn でデバイスが開かれた後) fUseKP パラメータが TRUE の WDC_IntEnable() を呼び出した場合、または、(関数に渡された WD_INTERRUPT 構造体の hKernelPlugIn フィールドで設定された) Kernel PlugIn ドライバで処理する低水準の InterruptEnable() または WD_IntEnable() 関数を呼び出した場合に呼び出されます。 この関数には Kernel PlugIn の割り込み処理に必要な初期化設定を含めてください。
KP_IntDisable()	ユーザー モード アプリケーションが WDC_IntDisable() を呼び出した場合か、または Kernel PlugIn ドライバで割り込みを有効にしていた場合に低水準の InterruptDisable() か WD_IntDisable() 関数を呼び出した場合に呼び出します。 この関数は KP_IntEnable() コールバックにより割り当てられたメモリを解放します。
KP_IntAtIrql()	WinDriver が割り込みを受け取った場合に呼び出されます (Kernel PlugIn へのハンドルを持つことによって可能となる割り込みを提供)。この関数はカーネル モードで割り込みを処理する関数です。この関数は高い割り込み要求レベルで実行されます。追加の引継ぎ処理は KP_IntAtDpc() またはユーザー モードで実行されます。
KP_IntAtDpc()	KP_IntAtIrql() コールバックが TRUE に返る割り込み処理を要求された場合に呼び出されます。

	この関数に優先度の低いカーネル モードの割り込み処理コードを含めます。 アプリケーションのユーザー モードの割り込み処理ルーチンが引き起こされる回数を決定します。

上記で説明したとおり、これらのハンドラはユーザー モード アプリケーションが Kernel PlugIn ドライバを (`WDC_xxxDeviceOpen()` / `WD_KernelPlugInOpen()`, `WDC_xxxDeviceClose()` / `WD_KernelPlugInClose()` を使用して) 開くまたは閉じる場合、(`WDC_CallKerPlug()` / `WD_KernelPlugInCall()` を呼び出して) Kernel PlugIn ドライバへメッセージを送信する場合、(Kernel PlugIn でデバイスを開いた後、`fUseKP` パラメータを `TRUE` に設定して `WDC_IntEnable()` を呼び出す、または関数へ渡された `WD_INTERRUPT` 構造体の `hKernelPlugIn` フィールドに設定した Kernel PlugIn ハンドルで `InterruptEnable()` または `WD_InterruptEnable()` を呼び出して) Kernel PlugIn ドライバで割り込みを有効にする場合、または Kernel PlugIn ドライバを使用して有効にした `WDC_IntDisable()` / `InterruptDisable()` / `WD_IntDisable()` 割り込みを無効にする場合に呼び出されます。

Kernel PlugIn の割り込み処理は、Kernel PlugIn ドライバを使用して割り込みが有効の場合に割り込みが発生した際に呼び出されます。

Kernel PlugIn のイベント ハンドラは、(Kernel PlugIn でデバイスを開いた後、`fUseKP` 引数に `TRUE` を設定して `WDC_EventRegister()` を呼び出す、または関数へ渡された `WD_EVENT` 構造体の `hKernelPlugIn` フィールドに設定した Kernel PlugIn へのハンドルで `EventRegister()` または `WD_EventRegister()` を呼び出して) Kernel PlugIn ドライバを使用して発生したイベントの通知を受け取るようにアプリケーションを登録した場合、Plug-and-Play またはパワーマネージメント イベントが発生した際に呼び出されます。

Kernel PlugIn コールバック関数の定義に加え、`KP_Open()` で Kenerl Plugin に必要な初期化設定を実行するコードを実装することもできます。KP_PCI ドライバのサンプルおよび DriverWizard で生成された Kernel PlugIn ドライバでは、たとえば、`KP_Open()` は、共有ライブラリの初期化関数を呼び出し、ユーザー モードから関数へ渡されるデバイス情報を保存するために使用される Kernel PlugIn ドライバ コンテキスト用のメモリを割り当てます。

KP_PCI サンプルから抜粋 (WinDriver/samples/pci_diag/kp_pci/kp_pci.c):

```
/* KP_PCI_Open is called when WD_KernelPlugInOpen() is called from the user
mode.
  pDrvContext will be passed to the rest of the Kernel PlugIn callback
functions. */
BOOL __cdecl KP_PCI_Open(KP_OPEN_CALL *kpOpenCall, HANDLE hWD,
    PVOID pOpenData, PVOID *ppDrvContext)
{
    PWDC_DEVICE pDev;
    WDC_ADDR_DESC *pAddrDesc;
    DWORD dwSize, dwStatus;
    void *temp;

    KP_PCI_Trace("KP_PCI_Open entered\n");

    kpOpenCall->funcClose = KP_PCI_Close;
    kpOpenCall->funcCall = KP_PCI_Call;
    kpOpenCall->funcIntEnable = KP_PCI_IntEnable;
    kpOpenCall->funcIntDisable = KP_PCI_IntDisable;
```

```
    kpOpenCall->funcIntAtIrql = KP_PCI_IntAtIrql;
    kpOpenCall->funcIntAtDpc = KP_PCI_IntAtDpc;
    kpOpenCall->funcEvent = KP_PCI_Event;

    /* Initialize the PCI library */
    dwStatus = PCI_LibInit();
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        KP_PCI_Err("KP_PCI_Open: Failed to initialize the PCI library: %s",
            PCI_GetLastErr());
        return FALSE;
    }

    /* Create a copy of device information in the driver context */
    dwSize = sizeof(WDC_DEVICE);
    pDev = malloc(dwSize);
    if (!pDev)
        goto malloc_error;

    COPY_FROM_USER(&temp, pOpenData, sizeof(void *));
    COPY_FROM_USER(pDev, temp, dwSize);

    dwSize = sizeof(WDC_ADDR_DESC) * pDev->dwNumAddrSpaces;
    pAddrDesc = malloc(dwSize);

    if (!pAddrDesc)
        goto malloc_error;

    COPY_FROM_USER(pAddrDesc, pDev->pAddrDesc, dwSize);
    pDev->pAddrDesc = pAddrDesc;

    *ppDrvContext = pDev;

       KP_PCI_Trace("KP_PCI_Open: Kernel PlugIn driver opened
successfully\n");

    return TRUE;

malloc_error:
    KP_PCI_Err("KP_PCI_Open: Failed allocating %ld bytes\n", dwSize);
    PCI_LibUninit();
    return FALSE;
}
```

11.6.2.4 残りの Kernel PlugIn コールバックの記述

使用したい残りの Kernel PlugIn ルーチン(割り込みを処理する `KP_Intxxx()` 関数、Plug-and-Play およびパワーマネージメント イベントを処理する `KP_Event()` など) を実装します。

11.6.3 Kernel PlugIn ドライバの生成されたコードとサンプル コード

DriverWizard を使用して、デバイスの Kernel PlugIn ドライバの雛型が生成します。その雛型を Kernel PlugIn ドライバの開発のベースとして使用できます (推奨)。または、**`WinDriver/samples/pci_diag/kp_pci/ Kernel PlugIn`** ディレクトリ以下のサンプル (**`KP_PCI`**) を使用できます。

Kernel PlugIn ドライバはスタンド アロン モジュールではありません。ドライブと通信を開始するユーザー モード アプリケーションが必要です。関連したアプリケーションは、DriverWizard を使用して生成した Kernel

PlugIn コードを使用した場合に生成されます。pci_diag アプリケーション (**WinDriver/samples/pci_diag/** ディレクトリに保存されています) は、サンプル **KP_PCI** ドライバと通信します。

KP_PCI サンプルおよび DriverWizard で生成されたコードはユーザー モード アプリケーション (**pci_diag / xxx_diag** - 'xxx' は生成されるドライバ名です) と Kernel PlugIn ドライバ (**kp_pci .sys/.o/.ko / kp_xxx.sys/.o/.ko**) 間の通信を行います。

サンプルおよび生成されたコードは Kernel PlugIn の KP_Open() 関数へのデータの渡し方法と Kernel PlugIn で他の関数により使用されるグローバル Kernel PlugIn ドライバ コンテキストを割り当て、保管する関数を使用方法を示します。

サンプルおよび生成された Kernel PlugIn コードは、ユーザー モードから Kernel PlugIn で特定の関数を開始する方法、およびメッセージを通じて Kernel PlugIn ドライバとユーザー モード WinDriver アプリケーションの間でデータを渡す方法を示すためにドライバのバージョン番号を入手するメッセージを実装します。

サンプルおよび生成されたコードには Kernel PlugIn での割り込み処理方法も含まれています。

Kernel PlugIn は割り込みカウンタを実装しています。Kernel PlugIn 割り込みハンドラは引継ぎ処理を実行し、割り込みの着信を 5 回おきにユーザー モード アプリケーションへ通知します。

KP_PCI サンプルは PCI の割り込み処理を実演していますが、サンプル KP_IntAtIrql() 関数のコメントで説明したとおり、割り込みの認識はハードウェアごとに異なるため、デバイス独自に割り込みを認識するコードを実装するには、この関数を修正する必要があります。

DriverWizard で生成されたコードは選択されたデバイス (PCI/PCMCIA/ISA) 用の割り込みコードのサンプルを含んでいます。生成された Kp_IntAtIrql() 関数は、ウィザード ([Interrput] タブで、レジスタにカードの割り込みへの読み取り/書き込みコマンドを指定) で定義した割り込み転送コマンドを実装するコードを含みます。割り込みを受け取った際にカーネルで認識される必要がある PCI および PCMCIA の割り込みの場合 (セクション 9.2.2)、生成されたコードが、定義したコマンドを実行するために必要なコードを既に含んでいるように、Kernel PlugIn コード生成の前にウィザードを使用して割り込みを認識 (消去) するコマンドを定義することを推奨します。

さらに、サンプルおよび生成されたコードには Kernel PlugIn で Plug-and-Play およびパワーマネージメント イベントの通知の受け取り方法も含まれています。

ヒント: Kernel PlugIn ドライバを記述する前に、Kernel PlugIn の生成されたプロジェクトおよびサンプル プログラムをビルドして実行することを推奨します。(デバイス用の必要な割り込みの認識を実装するために、コードを修正しないで対象の PCI デバイス独自の割り込みを処理する際に、汎用的な KP_PCI ドライバを使用することはできません)。

11.6.4 Kernel PlugIn のサンプル コードと生成されたコードのディレクトリ構造

11.6.4.1 pci_diag および kp_pci のサンプル ディレクトリ

kp_pci.c ファイルで、Kernel PlugIn のサンプル コード (KP_PCI) を実装してます。このサンプル ドライバは、WinDriver PCI 診断プログラムのサンプル (pci_diag) の一部で、KP_PCI ドライバに加え、ドライバ (**pci_diag**) と通信を行うユーザーモード アプリケーション、およびそのユーザーモード アプリケーションと Kernel PlugIn ドライバの両方で使用できる API を含む共有ライブラリが含まれます。C 言語でこのサンプルのソースを実装してます。

以下、**WinDriver/pci_diag/** ディレクトリ以下のファイルの概要です。

- **kp_pci/** - 以下の **KP_PCI** Kernel PlugIn ドライバ ファイルを含みます。
  - **kp_pci.c**: **KP_PCI**ドライバのソースコード
  - Kernel PlugIn のビルド用の Project および/または make ファイルと関連ファイル。Windows プロジェクト ファイルは、**x86** (32 ビット) および **amd64** (64 ビット) ディレクトリ以下のターゲット IDE のサブディレクトリ(**msdev2005** / **msdev2003** / **msdev_6**) にあります。Solaris の makefile は **SOLARIS/** にあります。
  - ターゲット OS 用の KP_PCI Kernel PlugIn ドライバのプリコンパイル済みバージョン:
    - **WINNT.i386\kp_pci.sys**: WinNT DDK でビルドした Windows x86 32 ビット用ドライバ
    - **WINNT.x86_64\kp_pci.sys**: Windows Server 2003 DDK でビルドした Windows x64 (64 ビット) 用ドライバ
    - **SOLARIS/kp_pci**: Solaris 用
    - Linux: Linux カーネル モジュールは、ターゲットにインストールされているカーネル バージョンのヘッダー ファイルでコンパイルする必要があるため、プリコンパイル済みバージョンはありません (セクション 14.4 を参照してください)。
- **pci_lib.c**: WinDriver の WDC API を使用して PCI デバイスにアクセスするライブラリの実装。ライブラリの API をユーザーモード アプリケーション (**pci_diag.c**) と Kernel PlugIn ドライバ (**kp_pci.c**) の両方で使用します。
- **pci_lib.h**: **pci_lib** ライブラリのインターフェイスを提供するヘッダーファイル
- **pci_diag.c**: サンプルの診断ユーザーモード コンソール (CUI) アプリケーションの実装で、**pci_lib** と WDC ライブラリを使用して PCI デバイスとの通信を行います。

  このサンプルでは、ユーザーモードの WinDriver アプリケーションから Kernel PlugIn ドライバへのアクセスを行います。デフォルトでは、**KP_PCI** Kernel PlugIn ドライバへのハンドルで、選択した PCI デバイスを開きます。成功した場合、セクション [11.6.3] の説明のとおり、Kernel PlugIn ドライバと通信を行います。Kernel PlugIn へのハンドルを開くのに失敗した場合、デバイスとのすべての通信をユーザーモードから実行します。
- **pci.inf** (Windows): Windows 98 / Me / 2000 / XP / Server 2003 / Vista 用のサンプル WinDriver PCI INF ファイル。注意: このファイルを使用するには、ファイル内の Vendor および Device ID を対象のデバイスの Vendor および Device ID に変更してください。
- **pci_diag**: **pci_diag** ユーザー モード アプリケーションのビルド用の Project および/または make ファイル。

  Windows プロジェクト ファイルは、**x86** (32 ビット) および **amd64** (64 ビット) ディレクトリ以下のターゲット IDE のサブディレクトリ(**msdev2005** / **msdev2003** / **msdev_6** / **cbuilder4** / **cbuilder3**) にあります。

  MSDEV ディレクトリには、Kernel PlugIn ドライバおよびユーザー モード アプリケーションのプロジェクト用のワークスペース/ソリューション ファイルも含まれています。

Linux および Solaris の makefile は、それぞれ **LINUX/** および **SOLARIS/** にあります。

- 対象の OS 用のユーザーモード アプリケーション (**pci_diag**) のプリコンパイル済みバージョン
  - Windows: **WIN32\pci_diag.exe**
  - Linux: **LINUX/pci_diag**
  - Solaris: **SOLARIS/pci_diag**
- **files.txt**: サンプル **pci_diag** ファイルの一覧
- **readme.txt**: サンプル Kernel PlugIn ドライバ、ユーザーモード アプリケーション、ビルド手順およびコードのテスト手順の概要

11.6.4.2 DriverWizard で生成された Kernel PlugIn ディレクトリ

対象のデバイス用に **DriverWizard で生成された Kernel PlugIn のコード**には、カーネルモードの Kernel PlugIn のプロジェクトと通信を行うユーザーモード アプリケーションが含まれます。汎用的な **KP_PCI** と **pci_diag** サンプルとは対照的に、Wizard で生成されたコードは、対象のデバイス用に検出または定義したリソース情報を使用します。同様に、コードを生成する前に Wizard で定義したデバイス独自の情報も使用します。

セクション [11.6.3] の説明のとおり、PCI または PCMCIA の割り込みを処理するドライバを使用する際には、Wizard で生成された割り込み処理のコードで、定義したハードウェア独自の情報を使用できるようにするためには、コードを生成する前に、DriverWizard で割り込みを検知するのに読み書きするレジスタを定義し、これらのレジスタから読み込む、またはレジスタへ書き込むコマンドを設定することを強く推奨します。

以下、DriverWizard で Kernel PlugIn のコードを生成した場合の、生成されたファイルの概要です (**xxx** は、コードを生成する際に指定したドライバの名前を表します。また、**kp_xxx** は、コードを保存先として指定したディレクトリを表します)。注意: 以下の概要は、生成される C コードについて示しています。Windows では、C Kernel PlugIn ドライバ (C# ではカーネル モード ドライバを実装できないため)、.NET C# ライブラリ、Kernel PlugIn ドライバと通信する C# ユーザー モード アプリケーションを含む、類似の C# コードを生成することもできます。

- **kermode/** - 以下の **KP_XXX** Kernel PlugIn ドライバ ファイルを含みます。
  - **kp_xxx.c**: **KP_XXX** ドライバのソースコード
  - Kernel PlugIn ドライバのビルド用の Project および/または make ファイルと関連ファイル 。Windows プロジェクト ファイルは、**x86** (32 ビット) および **amd64** (64 ビット) ディレクトリ以下のターゲット IDE のサブディレクトリ(**msdev2005** / **msdev2003** / **msdev_6**) にあります。Linux および Solaris の makefile は、それぞれ **LINUX/** および **SOLARIS/** にあります。
- **xxx_lib.c**: WinDriver の WDC API を使用して、対象のデバイスへアクセスするライブラリの実装。このライブラリの API をユーザーモード アプリケーション (**xxx_diag**) と Kernel PlugIn ドライバ (**KP_XXX**) の両方で使用します。
- **xxx_lib.h**: **xxx_lib** ライブラリのインターフェイスを提供するヘッダー ファイル
- **xxx_diag.c**: サンプルの診断ユーザーモード コンソール (CUI) アプリケーションの実装で、**xxx_lib** と WDC ライブラリを使用して PCI デバイスとの通信を行います。

このサンプルでは、ユーザーモードの WinDriver アプリケーションから Kernel PlugIn ドライバへのアクセスを行います。デフォルトでは、**KP_XXX** Kernel PlugIn ドライバへのハンドルで、選択した PCI デバイスを開きます。成功した場合、セクション [11.6.3] の説明のとおり、Kernel PlugIn ドライバと通信を行います。Kernel PlugIn へのハンドルを開くのに失敗した場合、デバイスとのすべての通信をユーザーモードから実行します。

- **xxx_diag**: **xxx_diag** ユーザー モード アプリケーションのビルド用の Project および/または make ファイル。

  Windows プロジェクト ファイルは、**x86** (32 ビット) および **amd64** (64 ビット) ディレクトリ以下のターゲット IDE のサブディレクトリ(**msdev2005** / **msdev2003** / **msdev_6** / **cbuilder4** / **cbuilder3**) にあります。

  MSDEV ディレクトリには、Kernel PlugIn ドライバおよびユーザー モード アプリケーションのプロジェクト用のワークスペース/ソリューション ファイルも含まれています。

  Linux および Solaris の makefile は、それぞれ **linux/** および **solaris/** にあります。

- **xxx_files.**txt: 生成されたファイルの一覧と生成されたコードのビルド手順

- **xxx.inf**: 対象のデバイスの WinDriver INF ファイル (Windows 98 / Me / 2000 / XP / Server 2003 / Vista で、PCI または PCMCIA などの Plug and Play デバイスの場合のみ)

11.6.5 Kernel PlugIn での割り込み処理

セクション [11.6.5.2] の説明のとおり、Kernel PlugIn ドライバの使用を有効にした場合、Kernel PlugIn ドライバで割り込みを処理します。

Kernel PlugIn の割り込みを有効にした場合、WinDriver がハードウェアの割り込みを受信した際には、Kernel PlugIn ドライバの `KP_IntAtIrql()` 関数を呼びます。`KP_IntAtIrql()` が `TRUE` を返す場合、`KP_IntAtIrql()` が処理を終え、`TRUE` を返した後に、遅延した `KP_IntAtDpc()` Kernel PlugIn 関数を呼びます。

`KP_IntAtDpc()` の戻り値は、ユーザーモードの割り込み処理ルーティンを実行する回数です。たとえば、KP_PCI のサンプルでは、Kernel PlugIn で実行中の割り込みハンドラは割り込みを 5 回カウントし、5 回毎にユーザー モードに通知します。従って、`WD_IntWait()` は、ユーザー モードでは受け取った割り込みの 5 回に 1 回しか通知しません (`KP_IntAtIrql()` は、`KP_IntAtDpc()` をアクティブにするために、5 回に 1 回 `TRUE` を返します。`KP_IntAtIrql()` は、`KP_IntAtIrql()` からの遅延された DPC呼出しの累積数を返します。つまりユーザー モードの割り込み処理は、5 回に 1 回しか実行されません)。

11.6.5.1 ユーザー モードの割り込み処理 (Kernel PlugIn なし)

Kernel PlugIn 割り込み処理が無効の場合、割り込みを受信する度に `WD_IntWait()` を返し、WinDriver がカーネルで割り込み処理を終了すると、ユーザーモードの割り込み処理ルーティンを起動します (主に、`WDC_IntEnable()` またはより低水準の `InterruptEnable()` または `WD_IntEnable()` への呼び出しで渡される割り込み転送コマンドの実行) – 図 [11.2] を参照。

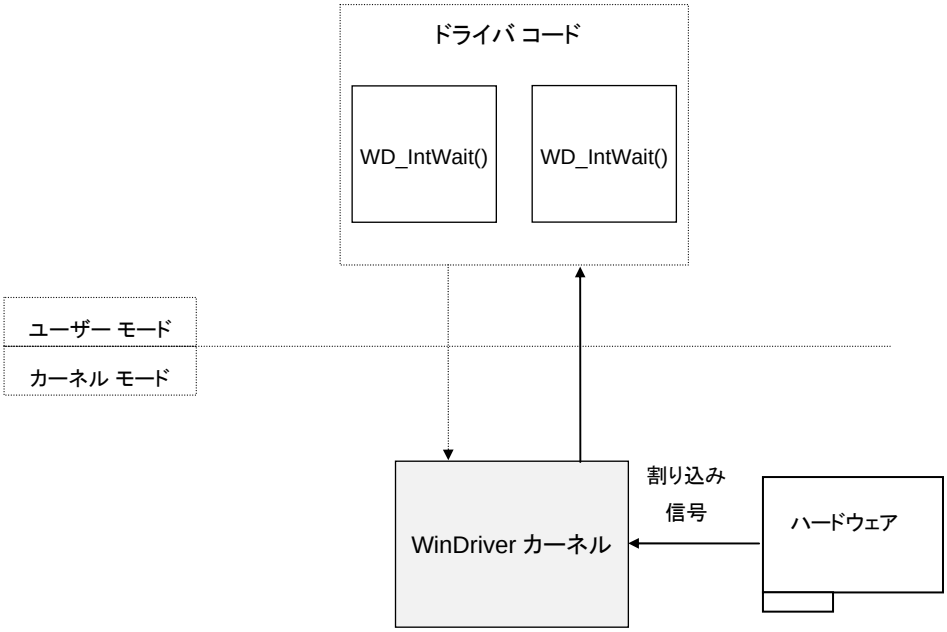


図 11.2: Kernel PlugIn なしでの割り込みの処理

11.6.5.2 カーネルでの割り込み処理 (Kernel PlugIn あり)

Kernel PlugIn で割り込みを処理するには、Kernel PlugInドライバの名前を`WDC_xxxDeviceOpen()`関数へ渡すことによって、ユーザーモード アプリケーションが Kernel PlugInドライバでデバイスへのハンドルを開き (PCI、PCMCIA、ISA)、そして `fUseKP` パラメータに `TRUE` を設定して、`WDC_IntEnable()` を呼びます。低水準 `WD_xxx()` API に関する詳細は、WinDriver PCI 低水準 API リファレンスを参照してください。

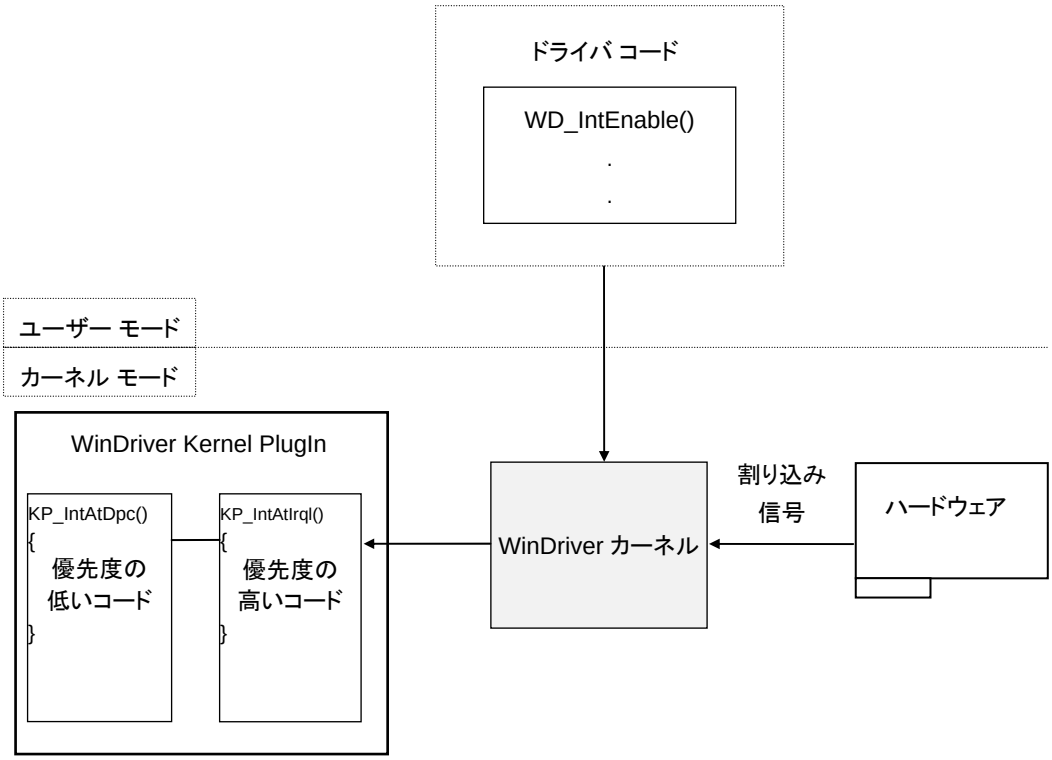


図 11.3: Kernel PlugIn ありでの割り込み処理

WDC_xxx API を使用しない場合、アプリケーションは、Kernel PlugIn ドライバへのハンドルを WD_IntEnable() 関数またはラッパー InterruptEnable() 関数へ渡します (WD_IntEnable() と WD_IntWait() を呼びます)。Kernel PlugIn 割り込み処理を有効にします。(関数へ渡される WD_INTERRUPT 構造体の hKernelPlugIn フィールド内に Kernel PlugIn ハンドルを渡します。)

Kernel PlugIn で割り込みを有効にするために、WDC_IntEnable() / InterruptEnable() / WD_IntEnable() を呼ぶときに、Kernel PlugIn の KP_IntEnable() コールバック関数を有効にします。この関数で、Kernel PlugIn 割り込み処理へ渡される割り込みコンテキストを設定できます。また同様に、ハードウェアで実際に割り込みを有効にするためにデバイスへの書き込みや、デバイスの割り込みを正確に有効にするために必要なコードを実装できます。

Kernel PlugIn 割り込みハンドラが有効な場合、KP_IntAtIrql() が割り込みのたびに呼び出されます。KP_IntAtIrql() 関数のコードは HIGH IRQL で実行されます。このコードの実行中はシステムが停止します (そのため、コンテキスト スイッチや、優先度の低い割り込みが処理されません)。

KP_IntAtIrql() 関数内のコードは、次の制約があります。

- ページしないメモリに対してのみアクセス可能です。
- 次の関数だけを呼び出し可能です (または、これらの関数を呼び出したラッパー関数)。
    - WDC_MultiTransfer()、WD_Transfer()、WD_MultiTransfer() または WD_DebugAdd()。
    - 高い 割り込み要求レベルから呼び出される OS 固有のカーネル関数 (WinDDK 関数など)。(これらの関数を使用すると、その他の OS とのコード互換性が損なわれる場合があるのでご注意ください。)

malloc()、free() または上記の関数以外の WDC_xxx または WD_xxx API 関数は呼びません。

前述の制限のため、KP_IntAtIrql() に記述するコードはできだけ小さくします (レベル センシティブ割り込みの検知／消去など)。割り込み処理で実行するその他をコードを KP_IntAtDpc() で実装します。KP_IntAtDpc() は、遅延した割り込みレベルで実行し、KP_IntAtIrql() と同じ制限を持っていません。KP_IntAtIrql() が戻り値を返した後に (TRUE を返した場合)、KP_IntAtDpc() を呼びます。

ユーザーモードで割り込み処理を行うこともできます。カーネルモードでの割り込み処理が終了した後に、ユーザーモードの割り込み処理ルーティンを呼ぶ回数が KP_IntAtDpc() の戻り値となります。

11.6.6 メッセージの受け渡し

WinDriver アーキテクチャでは、WDC_CallKerPlug() または低水準の WD_KernelPlugInCall() 関数を使用して、ユーザーモードから Kernel PlugIn ドライバへメッセージを渡すことによって、ユーザーモードからカーネルモードの関数を有効にすることができます。

ドライバのユーザーモードとカーネルモードの plugin 部分の両方に共通なヘッダーファイルにそのメッセージを定義します。**pci_diag KP_PCI** サンプル コードと DriverWizard で生成されたコードで、サンプル コードの場合には、共有ライブラリのヘッダーファイル **pci_lib.h** で、生成されたコードの場合には、**xxx_lib.h** で、メッセージを定義します。

ユーザー モードからメッセージを受け取ると、WinDriver は KP_Call() Kernel PlugIn コールバック関数を実行します。その関数は、受信したメッセージを確認し、このメッセージに対応したコードを実行します (Kernel PlugIn で実装されるように)。

生成された / サンプル Kernel PlugIn コードは、Kernel PlugIn へデータを渡すためにドライバのバージョンを取得するためのメッセージを実装します。KP_Call() でバージョン番号を設定するコードは、Kernel PlugIn がユーザー モード アプリケーションからメッセージを受信するときはいつでも Kernel PlugIn の中で実行されます。ヘッダー ファイル **pci_lib.h**/**xxx_lib.h** の中でメッセージの定義を参照できます。ユーザー モード アプリケーション (**pci_diag.exe**/**xxx_diag.exe**) は、WD_KernelPlugInCall() 関数から Kernel PlugIn ドライバへメッセージを送信します。

第 12 章
Kernel PlugIn の作成

Kernel PlugIn ドライバを記述する最も簡単な方法は、DriverWizard を使用して、ハードウェアの Kernel PlugIn コードを作成することです (セクション 11.6.3 および 11.6.4.2 を参照してください)。また、PCI Kernel PlugIn ドライバの開発の雛型として、`WinDriver/samples/pci_diag/kp_pci` ディレクトリにある `KP_PCI` Kernel PlugIn ドライバのサンプルを使用することもできます(セクション 11.6.3 および 11.6.4.1 を参照してください)。あるいは、コードをゼロから開発することもできます。

Kernel PlugIn ドライバの作成には、次のステップに従ってください。

12.1 Kernel PlugIn が必要かどうかを確認する

Kernel PlugIn はユーザー モードでドライバの開発、デバッグが終了してから使用します。開発やデバッグが容易なユーザー モードでドライバを作成してから移行してください。

ドライバのパフォーマンスを向上する方法を説明している第 10 章 の「パフォーマンスの向上」を参照して Kernel PlugIn が必要かどうかを確認してください。さらに、Kernel PlugIn では、ユーザーモードでドライバを記述する際に、ユーザーモードでは利用できないような柔軟性を提供します (特に、割り込み処理に関して)。

12.2 ユーザー モードのソース コードを用意する

1. 必要な関数を Kernel PlugIn へ移動して隔離します。
2. その関数からプラットフォーム固有のコードをすべて削除します。Kernel が使用する関数のみを使用します。
3. ユーザー モードでドライバをリコンパイルします。
4. ユーザー モードでドライバをデバッグして、変更後、コードが動作するか確認します。

注意: カーネル スタックはサイズに制限があります。このめた、Kernel PlugIn へ移動するコードには、静的なメモリ割り当てを持たないようにしてください。代わりに `malloc()` 関数を使用して、動的にメモリを割り当てます。これは特に大きいデータ構造に重要です。

注意: カーネルへ移植しているユーザー モード コードが、直接メモリ アドレスへアクセスする場合、物理アドレスのユーザー モード マッピングを使用する場合、低水準 `WD_CardRegister()` 関数から返されます。カーネル内で、代わりに物理アドレスのカーネル マッピングを使用する必要があります (カーネル マッピングは、`WD_CardRegister()` から返されます)。詳細は、本マニュアルの `WD_CardRegister()` についての説明を参照してください。**`WDC`** ライブラリの API を使用して、メモリにアクセスする場合、このことを考慮す

る必要はありません。関連する API をユーザーモードまたはカーネルモードでの使用に応じて、この API がメモリのマップを正しく行っているかを確認します。

12.3 Kernel PlugIn プロジェクトの新規作成

前述のように DriverWizard を使用して、デバイスの Kernel PlugIn のプロジェクト (ユーザー モード プロジェクトに対応) を新規作成できます (推奨)。また、開発の雛型として **KP_PCI** サンプルを使用して作成することもできます。

開発の雛型として **KP_PCI** サンプルを使用するように選択した場合、以下の手順に従ってください。

1. **WinDriver/samples/pci_diag/kp_pci** ディレクトリのコピーを作成します。たとえば、**KP_MyDrv** という Kernel PlugIn プロジェクトを新規作成する場合、**WinDriver/samples/pci_diag/kp_pci** を **WinDriver/samples/mydrv** へコピーします。

2. 新規に作成したディレクトリのすべての Kernel PlugIn ファイルの "KP_PCI" と "kp_pci" のすべてのインスタンスをそれぞれ "KP_MyDrv" と "kp_mydrv'' に変更します (注意: コードを正しく機能するには、**kp_pci.c** ファイルの KP_PCI_xxx() 関数名を変更する必要はありませんが、関数名にドライバ名を使用した方が、コードがより分かりやすくなります。)

3. ファイル名の “KP_PCI” という文字列を “kp_mydrv”へ変更します。

4. Kernel PlugIn ドライバとユーザーモード アプリケーションから共有 pci_lib ライブラリ API を使用するには、**pci_lib.h** と **pci_lib.c** ファイルを **WinDriver/samples/pci_diag/** ディレクトリから新規に作成した **mydrv/** ディレクトリにコピーします。ライブラリの関数名を変更して、"PCI'' ではなく、ドライバ名 (**MyDrv**) を使用できますが、この場合には、作成したKernel PlugIn プロジェクトとユーザーモード アプリケーションからこれらの関数へのすべての呼び出しで、名前を変更する必要があるので、注意してください。
   新規のプロジェクトに共有ライブラリをコピーしない場合、サンプルの Kernel PlugIn コードを編集し、PCI_xxx ライブラリ API へのすべての参照を他のコードに置き換える必要があります。

5. 必要に応じて、プロジェクト ファイル と make ファイルのファイルとディレクトリ パス、およびソース ファイルの #include パスを変更します (新規作成したプロジェクト ディレクトリの保存場所に依存します)。

6. **pci_diag** ユーザーモード アプリケーションを使用するには、**WinDriver/samples/pci_diag/pci_diag.c**、関連する pci_diag プロジェクト、ワークスペース/ソリューションまたは make ファイルを **mydrv/** ディレクトリへコピーし、ファイル名を変更し (希望に応じて)、ファイル内のすべての "pci_diag" の参照を変更したユーザーモード アプリケーションの名前に変更します。ワークスペース/ソリューション ファイルを使用するには、ファイル内の "KP_PCI" への参照を新規の Kernel PlugIn ドライバに変更します。たとえば、"KP_MyDrv"。そして、実装したいドライバの機能用にサンプル コードを変更します。

生成されたおよびサンプルコードの説明は、それぞれセクション 11.6.3 およびセクション 11.6.4 を参照してください。

12.4 Kernel PlugIn へのハンドルの作成

ユーザーモード アプリケーションまたはライブラリ ソース コードでは、Kernel PlugIn を使用してデバイスへのハンドルを開くには、Kernel PlugIn ドライバの名前で、WDC_PciDeviceOpen() /

WDC_PcmciaDeviceOpen() / WDC_IsaDeviceOpen() を呼びます (対象のデバイスの種類に依存します)。
DriverWizard で生成されたコードおよびサンプルの **pci_diag** 共有ライブラリで、このことを実装してます – 生成されたコードまたはサンプルの XXX_DeviceOpen()/PCI_DeviceOpen() ライブラリ関数 (生成されたコードまたはサンプルの **xxx_diag**/**pci_diag** ユーザーモード アプリケーションから呼ばれます) を参照してください。

コードで WDC ライブラリを使用しない場合、Kernel PlugIn ドライバへのハンドルを開くには、コードの初めで WD_KernelPlugInOpen() を呼ぶ、アプリケーションの終了前または Kernel PlugIn ドライバの使用を終了する前で、WD_KernelPlugInClose() を呼ぶ必要があります。WD_KernelPlugInOpen() は、関数へ渡された WD_KERNEL_PLUGIN 構造体の hKernelPlugIn フィールド内の Kernel PlugIn ドライバへのハンドルを返します。

12.5 Kernel PlugIn での割り込み処理の設定

1. WDC_IntEnable() を呼ぶ場合 (セクション 12.4 で説明したとおり、Kernel PlugIn ドライバ名で WDC_xxxDeviceOpen() を呼んで、Kernel PlugIn ドライバを使用してデバイスへのハンドルを開いた後)、fUseKP 関数の引数を TRUE に設定して、開いたデバイスに対して、Kernel PlugIn ドライバで割り込みを有効にすることを表します。

   DriverWizard で生成されたコードおよびサンプルの **pci_diag** 共有ライブラリ (**xxx_lib.c**/**pci_lib.c**) で実装してます – 生成されたコードまたはサンプルの XXX_IntEnable() / PCI_IntEnable() ライブラリ関数 (生成されたコードまたはサンプルの **xxx_diag** / **pci_diag** ユーザーモード アプリケーションから呼ばれます) を参照してください。

   WDC_xxx API を使用しない場合、Kernel PlugIn で割り込みを有効にするには、WD_IntEnable() または(WD_IntEnable() を呼び出す) InterruptEnable() を呼び出して、WD_KernelPlugInOpen() から受信した Kernel PlugIn へのハンドル (関数へ渡された WD_KERNEL_PLUGIN 構造体の hKernelPlugIn フィールド内) を渡します。

2. WDC_IntEnable() / InterruptEnable() / WD_IntEnable() を呼んで、Kernel PlugIn で割り込みを有効にする場合、WinDriver は Kernel PlugIn の KP_IntEnable() コールバック関数 を有効にします。この関数を実装して、Kernel PlugIn 割り込み処理 (KP_IntAtIrql() / KP_IntAtDpc()) へ渡される割り込みコンテキストを設定します。同様に、デバイスへ書き込むことによって、実際にハードウェアで割り込みを有効にします。たとえば、対象のデバイスの割り込みを正しく有効にするために、その他の必要なコードを実行します。

3. ユーザーモードの割り込み処理の実装または、この実装の関連部分を Kernel PlugIn の割り込み処理関数へ移動します。レベル センシティブな割り込みの検知 (クリア) 用のコードなど、優先度の高いコードを、高い割り込み要求のレベルで動作する KP_IntAtIrql()関数 へ移動する必要があります。遅延した割り込み処理を KP_IntAtDpc() へ移動することができます。KP_IntAtIrql() が割り込み処理を終了し、TRUE を返すと KP_IntAtDpc() を実行します。直接カーネルで高度な割り込み処理を行うために、コードを編集して、より効果的に割り込みを処理することもでき、より高い柔軟性を提供します (たとえば、特定のレジスタから値を読み込んだり、読み込んだ値を書き戻したり、特定のレジスタ ビットを換えたりします)。Kernel PlugIn ドライバを使用したカーネルでの割り込み処理の方法に関しては、セクション 11.6.5 を参照してください。

12.6 Kernel PlugIn での I/O 処理の設定

1. ユーザー モードから I/O 処理のコードを Kernel PlugIn メッセージ ハンドラ `KP_Call()` へ移動します。

2. ユーザーモードから I/O 処理を実行するカーネルのコードを有効にするには、`WDC_CallKerPlug()` または、Kernel PlugIn で実行したい各異なる機能の関連するメッセージで、低水準の `WD_KernelPlugInCall()` 関数を呼びます。

3. ユーザーモード アプリケーション(メッセージを送信する) と Kernel PlugIn ドライバ (メッセージを実装する) で共有するヘッダー ファイルでこれらのメッセージを定義します。

   サンプルまたは DriverWizard で生成された Kernel PlugIn プロジェクトでは、ユーザーモード アプリケーションと Kernel PlugIn ドライバで共有するメッセージ ID とその他の情報を **`pci_lib.h`** / **`xxx_lib.h`** 共有ライブラリ ヘッダー ファイルで定義します。

12.7 Kernel PlugIn ドライバのコンパイル

12.7.1 Windows でのコンパイル

サンプルの **`WinDriver\samples\pci_diag\kp_pci\`** Kernel PlugIn ディレクトリと Driver Wizard で生成されたKernel PlugIn の **`<project_dir>\kermode\`** ディレクトリ (**`<project_dir>`** は、生成されたドライバ プロジェクトを保存したディレクトリ) には、以下の Kernel PlugIn プロジェクト ファイルが含まれます (**`xxx`** はドライバ名。サンプルの場合は **`pci`**、またはウィザードでコードを生成する際に指定した名前。):

- **`x86\`**: 32 ビット プロジェクト ファイル
  - **`msdev_2005\kp_xxx.vcproj`**: 32 ビット MSDEV 2005 プロジェクト
  - **`msdev_2003\kp_xxx.vcproj`**: 32 ビット MSDEV 2003 プロジェクト
  - **`msdev_6\kp_xxx.dsp`**: 32 ビット MSDEV 6.0 プロジェクト
- **`amd64\`**: 64 ビット プロジェクト ファイル
  - **`msdev_2005\kp_xxx.vcproj`**: 64 ビット MSDEV 2005 プロジェクト

サンプルの **`WinDriver\samples\pci_diag\`** ディレクトリと生成された **`<project_dir>\`** ディレクトリには、それぞれ Kernel PlugIn ドライバを実行するユーザーモード アプリケーション用のプロジェクト ファイルが含まれます(**`xxx`** はドライバ名。サンプルの場合は **`pci`**、またはウィザードでコードを生成する際に指定した名前。):

- **`x86\`**: 32 ビット プロジェクト ファイル
  - **`msdev_2005\xxx_diag.vcproj`**: 32 ビット MSDEV 2005 プロジェクト
  - **`msdev_2003\xxx_diag.vcproj`**: 32 ビット MSDEV 2003 プロジェクト
  - **`msdev_6\xxx_diag.dsp`**: 32 ビット MSDEV 6.0 プロジェクト

- **cbuilder4\xxx.bpr** および **xxx.cpp**: Borland C++ Builder 4.0 プロジェクト ファイルと関連 CPP ファイル。これらのファイルは、Borland C++ Builder 5.0 および 6.0 でも使用できます。
- **cbuilder3\xxx.bpr** および **xxx.cpp**: Borland C++ Builder 3.0 プロジェクトファイルと関連 CPP ファイル

- **amd64**: 64 ビット プロジェクト ファイル
  - **msdev_2005\xxx_diag.vcproj**: 64 ビット MSDEV 2005 プロジェクト

上記の MSDEV ディレクトリには、Kernel PlugIn とユーザーモード アプリケーション両方のプロジェクト ファイルを含む **xxx_diag.dsw/sln** ワークスペース / ソリューション ファイルも含まれています。

Kernel PlugIn ドライバと各ユーザーモード アプリケーションをビルドするには、以下のステップを実行します。

1. ご使用の PC に対象の OS 用の Windows DDK (Driver Development Kit) がインストールされていることを確認します (セクション 11.6.1 を参照してください)。

   対象の OS とは、作成するドライバが動作するオペレーティング システムのことです。たとえば、Windows XP 用のドライバを作成する場合には、Windows XP DDK をインストールします。

   それぞれの OS 用に複数の DDK をインストールでき、ドライバが対応する対象の OS に応じて、Kernel PlugIn ドライバをリビルドします。

2. 対象のプラットフォーム用の Windows DDK の場所を示すように **BASEDIR** 環境変数を設定します。たとえば、Windows XP 用の Kernel PlugIn ドライバをビルドするには、Windows XP DDK をインストールしたディレクトリを示すように **BASEDIR** 環境変数を設定します。

3. Microsoft Developer Studio (MSDEV) を開始し、下記のステップを実行します。

   ① ドライバのプロジェクト ディレクトリから、生成されたワークスペース／ソリューション ファイル (**<project_dir>\<MSDEV_dir>\xxx_diag.dsw/.sln**) を開きます。
   <project_dir> は、ドライバのプロジェクト ディレクトリです (サンプル コードの場合は **pci_diag**、または Driver Wizard で生成されたコードの保存先のディレクトリ)。
   <MSDEV_dir> は、ターゲット MSDEV ディレクトリ (**msdev2005** / **msdev2003** / **msdev_6**) です。**xxx** はドライバ名です (サンプルの場合は **pci**、または Driver Wizard でコードを生成する際に指定した名前)。

   DriverWizard で MSDEV IDE 用にコードを生成するように選択した場合、コード ファイルを生成した後、Wizard が自動的に MSDEV を起動し、生成されたワークスペース ファイルまたはソリューション ファイルを開くので注意してください。ただし、コード生成ダイアログの "IDE to Invoke" オプション を"None" に設定することによって、この動作を回避できます。

   ② Kernel PlugIn SYS ドライバ (サンプルの場合は **kp_pci.sys**、ウィザードで生成されたコードの場合は **kp_xxx.sys**) をビルドするには以下のステップを実行します。

      i. Kernel PlugIn プロジェクト (**kp_pci.dsp/vcproj** または **kp_xxx.dsp/vcproj**) をアクティブなプロジェクトとして設定します。

ii. 対象のプラットフォーム用のアクティブな構成を選択します。[ビルド] メニューから [構成マネージャ] (MSDEV 2003 / 2005 の場合) または [アクティブな構成の設定 …] (MSDEV 6.0 の場合) を選択し、使用する構成を選択します。

注意: アクティブな設定は、ビルドするドライバのターゲットの OS と対応している必要があります。たとえば、Windows 2000 の場合、**Win32 win2k free** (リリース モード) または **Win32 win2k checked** (デバッグ モード) のどちらかを選択します。

iii. ドライバをビルドします。ショートカット キー (MSDEV 6.0 では F7 キー) を押すか、[Build] メニューから実行してください。

③ Kernel PlugIn ドライバを実行するユーザーモード アプリケーションをビルドするには、以下のステップを実行します (サンプルの場合は **pci_diag.exe**、ウィザードで生成されたコードの場合は **xxx_diag.exe**):

i. ユーザーモード プロジェクト (サンプルの場合は **pci_diag.dsp/vcproj**、ウィザードで生成されたコードの場合は **xxx_diag.dsp/vcproj**) をアクティブなプロジェクトとして設定します。

ii. アプリケーションをビルドします。ショートカット キー (MSDEV 6.0 では F7 キー) を押すか、[ビルド] メニューから実行します。

注意: Windows 98 / Me では、生成されたコードを上記で説明した方法では、SYS ドライバをビルドできません。ただし、Windows 2000 / XP / Server 2003 / Vista で Windows 98 / Me プラットフォームをターゲットとして SYS ドライバをビルドできます – つまり、Windows 2000 / XP / Server 2003 / Vista が起動する PC でコードをビルドします。しかし、Windows 98 / Me DDK の場所を示すように **BASEDIR** 環境変数を設定し、Windows 98 / Me 用に Kernel PlugIn プロジェクトでビルドターゲットを設定し、Windows 98 / Me で生成されたドライバを使用します。

12.7.2 Linux でのコンパイル

1. Shell ターミナルを開きます。

2. Kernel PlugIn ディレクトリに移動します。たとえば、サンプル KP_PCI ドライバをコンパイルする場合は、次のコマンドを実行します。
   cd WinDriver/samples/pci_diag/kp_pci

   DriverWizard で生成された Kernel PlugIn コード用の Kernel PlugIn ドライバをコンパイルする場合は、次のコマンドを実行します (**<path>** は生成された DriverWizard プロジェクトのディレクトリ。例: **/home/user/WinDriver/wizard/my_projects/my_kp/**):
   cd <path>/kermode/linux/

3. **configure** スクリプトを使用して、**makefile** を生成します。
   ./configure

   注意: configure スクリプトは、起動してるカーネルをベースとしたカーネル独自の **makefile** を作成します。**configure** スクリプトに **--with-kernel-source=<path>** フラグを追加することによって、インストールした他のカーネル ソースをベースとした configure スクリプトを起動できます。<path> には、カーネル ソースのディレクトリへのフル パスを指定します。

4. Kernel PlugIn モジュールをビルドします。“**make**” コマンドを使用します。

このコマンドは、作成された **kp_xxx.o/.ko** ドライバを含む、新しい **LINUX.xxx/** ディレクトリを作成します (**xxx** は Linux カーネルにより異なります)。

5. サンプル ユーザーモード診断アプリケーションの makefile のあるディレクトリに移動します。
   KP_PCI サンプル ドライバの場合:
   cd ../LINUX/

   DriverWizard で生成された Kernel PlugIn ドライバの場合:
   cd ../../linux/

6. サンプル診断プログラムをコンパイルします。"**make**" コマンドを使用します。

12.7.3 Solaris でのコンパイル

注意: WinDriver は、GNU **make** ユーティリティ用にのみ **makefiles** を生成します。

GNU **make** ではなく、標準的な **make** ユーティリティを使用する場合、WinDriver が生成する **makefile** を修正する必要があります。http://www.sunfreeware.com から GNU **make** パッケージを入手できます。

1. Shell ターミナルを開きます。

2. ドライバのプロジェクト ディレクトリに移動します。

   たとえば、サンプル KP_PCI ドライバをコンパイルする場合は、次のコマンドを実行します。
   cd WinDriver/samples/pci_diag/

3. "**make**" コマンドを使用して、Kernel PlugIn モジュールをビルドします。たとえば、サンプル KP_PCI ドライバをビルドする場合は、次のコマンドを実行します。
   make -C kp_pci /SOLARIS

   DriverWizard で生成された Kernel PlugIn ドライバをビルドする場合は、次のコマンドを実行します。
   make -C kermode/solaris

4. サンプル診断プログラムをコンパイルします。
   make -C solaris

注意: 64 ビット カーネルの場合、Kernel PlugIn モジュールおよびユーザーモード アプリケーションの両方共に 64 ビット モードでコンパイルする必要があります。WinDriver で提供される makefile は、特別な宣言をせずに CC および LD 環境変数を使用します。したがって、ご使用のコンパイラおよびリンカに対応する独自のフラグに合うように、これらの変数を設定する必要がある場合があります。

たとえば、GCC でコンパイルする場合には、CC および LD 変数を以下のように設定する必要があります。

Kernel PlugIn モジュールのコンパイルの場合:

```
$ export LD="gcc -m64 -melf64_sparc -nostdlib"
$ export CC="gcc -m64 -isystem /usr/include/"
```

ユーザーモード アプリケーションのコンパイルの場合:

```
$ export LD="gcc -m64"
$ export CC="gcc -m64"
```

12.8 Kernel PlugIn ドライバのインストール

12.8.1 Windows の場合

1. ドライバ ファイル (**xxx.sys**) をターゲット プラットフォームのドライバ ディレクトリ **%windir%\system32\drivers** にコピーします。
   (例: Windows 2000 の場合は **C:\WINNT\system32\drivers**、Windows XP / Server2003 / Vista の場合は **C:\Windows\system32\drivers**。)

   注意: Kernel PlugIn ドライバを Windows 98 / Me 用に開発する場合、Windows 2000 / XP / Server 2003 / Vista ホスト PC でドライバを開発してください (セクション 12.7 の注意を参照してください)。

2. Windows 2000 / XP / Server 2003 / Vista の場合、**wdreg.exe** (または **wdreg_gui.exe**) ユーティリティを使用して、以下のようにドライバを登録またはロードします。Windows 98 / Me の場合、**wdreg16.exe** ユーティリティを使用します。

   注意:

   - 下記の手順では、.sys 拡張子のない、"KP_NAME" は Kernel PlugIn ドライバの名前を表しています。
   - Windows 98 / Me の場合、"wdreg" の文字列を "wdreg16" に置き換えます。WDREG ユーティリティに関する詳細は、セクション 13.2.2 を参照してください。

   SYS ドライバのインストール:
   ```
   WinDriver\util> wdreg -name KP_NAME install
   ```

注意: Windows 98 / Me 上の SYS ドライバの例外では、Kernel PlugIn ドライバは動的にロードできます。そのため、ロード時にリブートする必要はありません。

12.8.2 Linux の場合

1. Kernel PlugIn ドライバのディレクトリに移動します。

   たとえば、サンプル KP_PCI ドライバをインストールする場合は、次のコマンドを実行します。
   ```
   cd WinDriver/samples/pci_diag/kp_pci
   ```

   DriverWizard で生成された Kernel PlugIn ファイルを使用して作成したドライバをインストールする場合は、次のコマンドを実行します (**<path>** は生成された DriverWizard プロジェクトのディレクトリ。例: **/home/user/WinDriver/wizard/my_projects/my_kp/**):
   ```
   cd <path>/kermode/
   ```

2. 次のコマンドを実行して、Kernel PlugIn ドライバをインストールします。
   ```
   make install
   ```

12.8.3 Solaris の場合

Kernel PlugIn ドライバのインストールは、root でログインするか、root 権限を持つユーザー (super user) になってシステム管理者となって実行してください。

1. Kernel PlugIn ディレクトリに移動します。

サンプル KP_PCI ドライバをインストールする場合は、次のコマンドを実行します。
 # cd WinDriver/samples/pci_diag/kp_pci

DriverWizard で生成された Kernel PlugIn ファイルを使用して作成したドライバをインストールする場合は、次のコマンドを実行します (**<path>** は生成された DriverWizard プロジェクトのディレクトリ。例: **/home/user/WinDriver/wizard/my_projects/my_kp/**):
 # cd <path>/kermode

2. 設定ファイル (**kp_pci.conf**) をターゲットの **kernel/drv/** ディレクトリにコピーします。

   サンプル **KP_PCI** Kernel PlugIn 設定ファイルは、**kp_pci /** サンプル ディレクトリの直下にあります。サンプル ドライバをインストールする場合は、次のコマンドを実行します。
   # cp kp_pci.conf /kernel/drv

   DriverWizard で生成された Kernel PlugIn 設定ファイルは、プロジェクトの **kermode/solaris** サブディレクトリ以下にあります。生成された Kernel PlugIn ドライバをインストールする場合は、次のコマンドを実行します。
   # cp solaris/kp_pci.conf /kernel/drv

3. Kernel PlugIn の Solaris サブディレクトリに移動します。

   サンプル KP_PCI ドライバの場合は、次のコマンドを実行します。
   # cd SOLARIS

   DriverWizard で生成された Kernel PlugIn ファイルを使用して作成したドライバをインストールする場合は、次のコマンドを実行します。
   # cd solaris

4. Kernel PlugIn ドライバをターゲットのドライバ ディレクトリにコピーします。

   サンプル KP_PCI ドライバをコピーする場合:

   64 ビット プラットフォームでは、次のコマンドを実行します。
   # cp kp_pci /kernel/drv/sparcv9

   32 ビット プラットフォームでは、次のコマンドを実行します。
   # cp kp_pci /kernel/drv

注意: その他、Solaris 上でドライバをインストールする際に便利なコマンドを紹介します。

- **modinfo** – ロードしてるカーネル モジュールの一覧を表示
- **rem_drv** – カーネル モジュールを削除

第 13 章
ドライバの動的ロード

13.1 なぜ動的にロード可能なドライバが必要なのか

新しいドライバを追加した際に、システムにドライバをロードするには、システムを再起動する必要がある場合があります。WinDriver は動的にロード可能なドライバなので、ドライバをインストール後、システムの再起動をせずに直ぐにアプリケーションを使用できます。ユーザーモード ドライバまたはカーネルモード (Kernel PlugIn について [第 11 章] を参照) ドライバのどちらを作成しても、動的にドライバをロードできます。

注意: ドライバのアンロードを行うには、WinDriver アプリケーション、Kernel PlugIn、INF ファイルを使用して WinDriver へ登録された Plug-and-Play デバイスのハンドルが開いていないことを確認してください。

13.2 Windows の動的ドライバ ロード

13.2.1 Windows ドライバの種類

Windows ドライバを以下のいずれの種類としても実装することができます。

- WDM (Windows Driver Model) ドライバ: Windows 98 / Me / 2000 / XP / Server 2003 / Vista 上で **`.sys`** 拡張子のファイル。たとえば、**`windrvr6.sys`**。WDM ドライバは、INF ファイルをインストールすることによってインストールされます。

- 非 WDM / レガシー ドライバ: 非 Plug-and-Play Windows OS (Windows NT4.0) 用のドライバ、Windows 98 / Me の **`.vxd`** 拡張子のファイルおよびすべての Kernel PlugIn ドライバ ファイル (つまり、**`MyKPDriver.sys`**) が含まれます。

  注意: WinDriver のバージョン 6.21 以降 より .vxd ドライバはサポートされていません。

WinDriver Windows カーネル モジュール (**`windrvr6.sys`**) は、フル WDM ドライバです。下記のセクションで説明しますが、**`wdreg`** ユーティリティを使用してインストールできます。

13.2.2 WDREG ユーティリティ

WinDriver には、動的にドライバをロードおよびアンロードするユーティリティがあるので、Windows のデバイス マネージャによる手動の作業を行いません (デバイスの INF には使用します)。Windows 2000 / XP / Server 2003 / Vista の場合、このユーティリティを **wdreg** と **wdreg_gui** という 2 つの形態で提供しています。これらは **`WinDriver\util`** ディレクトリにあり、コマンドラインから実行でき、同じ機能を持っています。これらファイルの違いとしては、**wdreg_gui** は、インストールメッセージをグラフィカルに表示し、**wdreg** はコンソール モードのメッセージを表示します。

Windows 98 / Me の場合、wdreg16 ユーティリティを提供します。

ここでは、Windows オペレーティング システムの **wdreg** / **wdreg_gui** / **wdreg16** の使用方法を説明します。

注意:

1. Windows 2000 / XP / Server 2003 / Vista の **wdreg** は、**-compat** オプションで起動しない場合を除き、Driver Install Framework API (**DIFxAPI**) - **difxapi.dll** に依存します。**difxapi.dll** は WinDriver\util ディレクトリ以下にあります。

2. **wdreg** に関しては、以下のサンプルと説明を参照してください。Windows 2000 / XP / Server 2003 / Vista の場合、**wdreg** の文字列を **wdreg_gui** に置き換えられます。Windows 98 / Me の場合、**wdreg** を **wdreg16** に置き換えてください。

3. Windows 98 / Me では、**wdreg16** のみを使用して、**windrvr6.inf** をインストールして、WDM サービス **windrv6.sys** および Kernel PlugIn ドライバをインストールします。しかし、他のいかなる INF ファイルをインストールするのに **wdreg16** を使用することはできません。

13.2.2.1 WDM ドライバ

このセクションでは、**wdreg** ユーティリティを使用して、Windows 98 / Me / 2000 / XP / Server 2003 / Vista で WDM **windrvr6.sys** ドライバをインストールする方法、または Windows 2000 / XP / Server 2003 / Vista で Plug-and-Play のデバイス (PCI または PCMCIA など) および USB デバイスを **windrvr6.sys** ドライバと動作するように登録する INF ファイルのインストール方法を解説します。

注意:

- 上記の指定のように、Windows 98 / Me では、**wdreg16** を使用して、**windrvr6.inf** をインストールして、**windrvr6.sys** WDM ドライバをインストールします。しかし、**wdreg16** を使用して、他のいかなる INF ファイルもインストールできません。

- Kernel PlugIn ドライバは、WDM ドライバではなく、かつ INF からインストールされていないので、この節の説明には当てはまりません。Windows 98 / Me / 2000 / XP / Server 2003 / Vista で、**wdreg** を使用して Kernel PlugIn ドライバをインストールする方法に関しては、セクション [13.2.2.2] を参照してください。

使用法: 以下で紹介するように、**wdreg** ユーティリティを二通りの方法で使用できます。

1. **wdreg -inf <ファイル名> [-silent] [-log <ログファイル>] [install | uninstall | enable | disable]**

2. **wdreg -rescan <enumerator> [-silent] [-log <ログファイル>]**

- オプション

  wdreg は次の基本オプションをサポートします。

  - **[-inf]** – 動的にインストールされる INF ファイルへのパス。

  - **[-rescan <enumerator>]** (Windows 2000 / XP / Server 2003 / Vista) –ハードウェアが変更した場合に、enumerator (ROOT、ACPI、PCI、USB など) を再スキャンします。一つの enumerator のみ指定できます。

  - **[-silent]** – いかなるメッセージも表示しません。(オプション)

- **[-log <ログファイル>]** – 指定したファイルに全てのメッセージを記録します。(オプション)
- **[-compat]** (Windows 2000 / XP / Server 2003 / Vista) – 新しい Driver Install Framework API (**DIFxAPI**) ではなくトラディショナルな **SetupDi** API を使用します。

- アクション

  wdreg は、次の基本アクションをサポートします。

  - **[install]** – INF ファイルをインストールし、対象の場所へファイルをコピーし、古いバージョンと置き換えることによって INF ファイル名で指定したドライバを動的にロードします (必要な場合)。
  - **[-preinstall]** (Windows 2000 / XP / Server 2003 / Vista) – マシン上にないデバイスの INF ファイルをプリインストールします。
  - **[uninstall]** – 次に起動する際にロードしないようにレジストリからドライバを削除します。
  - **[enable]** – ドライバを有効にします。
  - **[disable]** – ドライバを無効にします。動的にドライバをアンロードするが、システムが起動後、ドライバはリロードします。

注意: WinDriver を正常に無効 / アンインストールするには、初めに、**windrvr6.sys** サービスへのハンドルが開いている場合、そのハンドルをすべて閉じてください (開いている WinDriver アプリケーションをすべて閉じます。さらに、デバイス マネージャまたは **wdreg** を使用して、**windrvr6.sys** サービスと一緒に動作するように登録されている PCI / PCMCIA / USB デバイスをすべてアンインストールします (または、デバイスを削除します))。**windrvr6.sys** サービスへのハンドルが開いている状態でサービスを停止しようとした場合、**wdreg** は関連エラーメッセージを表示します。開いているハンドルをすべて閉じて再試行するか、キャンセルし PC を再起動してコマンドを終了するかを選択できます。

13.2.2.2 非 WDM ドライバ

このセクションでは、**wdreg** ユーティリティを使用して 非 WDM ドライバ (Windows 98 / Me / 2000 / XP / Server 2003 / Vista 上での Kernel PlugIng ドライバ) をインストールする方法を説明します。

使用法:

wdreg [-file <ファイル名>] [-name <ドライバ名>] [-startup <level>] [-silent] [-log <ログファイル>] Action [Action ...]

- オプション

  wdreg は次の基本オプションをサポートします。

  - **[-startup]** – ドライバを開始するときに指定します。以下の引数の 1 つが必要となります。
    - **boot:** オペレーティング システムのローダーで開始されるドライバを表示します。そして、OS (たとえば、Atdisk) をロードする必要のあるドライバにのみ使用されます。
    - **system:** OS の初期化中に開始されたドライバを表示します。

- **automatic:** システムが起動中に Service Control Manager によって開始されたドライバを表示します。
- **demand:** 要求に応じて Service Control Manager によって開始されたドライバを表示します。(ドライバをプラグインした場合など)
- **disabled:** 開始されないドライバを表示します。

注意: デフォルトでは、**-**statup オプションは **automatic** に設定されています。

- **[-name]** – Kernel PlugIn を使用している場合のみ関連があります (デフォルトでは、**wdreg** コマンドは windrvr6 サービスに関連しています)。ドライバのシンボリック名を設定します。この名前はユーザーモード アプリケーションがドライバを処理するのに使用します。このオプションの引数として、ドライバのシンボリック名 (*.sys 拡張子なし) を引数に設定します。引数は KernelPlugIn プロジェクトにある KP_Init() 関数内で設定するドライバ名と同じ必要があります。

  ```
  strcpy (kpInit->cDriverName , XX_DRIVER_NAME)
  ```

- **[-file]** – Kernel PlugIn を使用している場合のみ関連があります。**wdreg** を使用して物理ファイル名と異なる名前でレジストリにドライバをインストールできます。このオプションにはドライバのファイル名が引数として必要です (*.sys 拡張子なし)。

  wdreg は Windows のインストール ディレクトリ (**<WINDIR>\system32\drivers**) を検索します。したがって、ドライバをインストールする前に関連ディレクトリにドライバ ファイルが検出できることを確認する必要があります。

  使用法:
  WDREG -name <新しいドライバ名> -file <オリジナルのドライバ名> install

- **[-silent]** – いかなるメッセージも表示しません。
- **[-log <ログファイル>]** – 指定したファイルに全てのメッセージを記録します。

- アクション

  wdreg は、次の基本アクションをサポートします。

  - **[create]** – ドライバをレジストリに追加することにより、次に Windows を起動する際にロードするようにします。
  - **[delete]** – 次に起動する際にロードしないように レジストリからドライバを削除します。
  - **[start]** – ドライバを動的にロードします。ドライバを **start** する前に **create** する必要があります。
  - **[stop]** – メモリからドライバを動的にアンロードします。

    注意: windrvr6.sys サービスを正常に終了させるには、初めに、このサービスへのハンドルが開いている場合、そのハンドルをすべて閉じてください (開いている WinDriver アプリケーションを閉じます)。ハンドルが開いている状態でサービスを停止しようとした場合、**wdreg** が関連エラーメッセージを表示します。

- ショートカット

wdreg には次の便利なショートカットが用意されています。

- **[install]** – ドライバを作成し、開始します。

  これは、初めに wdreg **stop** アクション (ドライバのバージョンが現在ロードされている場合) または wdreg **start** アクション (ドライバのバージョンが現在ロードされていない場合) を使用し、wdreg **start** アクションを使用するのと同様です。

- **[preinstall]** (Windows 2000 / XP / Server 2003 / Vista) – 接続していないデバイスのドライバを作成し、開始します。

- **[uninstall]** – 次の起動時にロードしないように、メモリからドライバをアンロードし、レジストリからドライバを削除します。

  これは、初めに wdreg **stop** アクションを使用し、wdreg **delete** アクションを使用するのと同様です。

  注意: WinDriver のサービスを正常に終了するには、**windrvr6.sys** ドライバ (WinDriver アプリケーションなど) へのハンドルが開いていないことを確認してください。このことは、**install** および **uninstall** のショートカットにも当てはまります。WinDriver のサービスを停止するコマンドが含まれています。**windrvr6.sys** へのハンドルが開いている状態でサービスを停止しようとした場合、**wdreg** が関連エラーメッセージを表示します。

13.2.3 windrvr6.sys INF ファイルの動的ロード / アンロード

WinDriver を使用する場合、汎用ドライバである **windrvr6.sys** (WinDriver のカーネル モード) を使用してハードウェアにアクセスしてコントロールするユーザーモード アプリケーションを開発します。したがって、ドライバ **windrvr6.sys** を動的にロード / アンロードするのに、**wdreg** を使用できます。また、WDM 互換の OS 上では Plug-and-Play デバイス用の INFファイルを動的にロードする必要があります。Windows 2000 / XP / Server 2003 / Vista 上では、**wdreg** で自動的に動的ロードします。ここでは、前述のセクションの説明に基づき、**wdreg** の使用例について説明します。

例:

- Windows 98 / Me / 2000 / XP / Server 2003 / Vista 上で **windrvr6.sys** を開始するには、次のコマンドを実行します。

  ```
  wdreg –inf [windrvr6.inf へのパス] install
  ```

  このコマンドは、**windrvr6.inf** ファイルをロードし、**windrvr6.sys** サービスを開始します。

- Windows 2000 / XP / Server 2003 / Vista 上で **C:\temp** ディレクトリにある **device.inf** と言う名の INF ファイルをロードするには、次のコマンドを実行します。

  ```
  wdreg –inf c:\tmp\device.inf install
  ```

  Windows 2000 / XP / Server 2003 / Vista では、上記の **install** オプションを **preinstall** オプションに置き換えて、PC に接続していないデバイスの INF ファイルをプリインストールできます。

ドライバ / INF ファイルをアンロードするには、同じコマンドを使用しますが、上記の例で、 **install** オプションを **uninstall** オプションに置き換えます。

13.2.4 Kernel PlugIn ドライバを動的にロード / アンロード

WinDriver を使用して Kernel PlugIn ドライバを作成した場合は、WinDriver の汎用ドライバ **windrvr6.sys**をロードした後に、Kernel PlugIn をロードする必要があります。

ドライバをアンインストールする際には、**windrvr6.sys** をアンロードする前に Kernel PlugIn ドライバをアンロードする必要があります。

注意: Windows 98 / Me では、Kernel PlugIn を動的にロードしないので、初期ロード後に再起動する必要があります。その他の Windows プラットフォームでは、Kernel PlugIn を動的にロードするので、再起動する必要はありません。

Kernel PlugIn ドライバ (**<ドライバ名>.sys**) をロード / アンロードするには、上記の windrvr6.sys の説明のように、"name" フラグ (Kernel PlugIn ドライバの名前を追加した後) をつけて、**wdreg** コマンドを使用します。

注意: ドライバ名に拡張子 ***.sys** を追加しないでください。

例:

- **KPDriver.sys** と呼ばれる KrenelPlugin ドライバをロードするには、次のコマンドを実行します。
  wdreg -name KPDriver install
- MPEG_Encoder と呼ばれる Kernel PlugIn ドライバを **MPEGENC.sys** とロードするには、次のコマンドを実行します。
  wdreg -name MPEG_Encoder -file MPEGENC install
- **KPDriver.sys** と呼ばれる KernelPlugIn ドライバをアインストールするには、次のコマンドを実行します。
  wdreg -name KPDriver uninstall
- MPEG_Encoder と呼ばれる KernelPlugIn ドライバと MPEGENC.sys ファイルをアインストールするには、次のコマンドを実行します。
  wdreg -name MPEG_Encoder -file MPEGENC uninstall

13.3 Linux の動的ドライバ ロード

- Linux で WinDriver を動的にロードするには、次のコマンドを実行します。
  /sbin/modprobe windrvr6
- WinDriver を動的にアンロードするには、次のコマンドを実行します。
  /sbin/rmmod windrvr6
- また、**WinDriver/Util/** ディレクトリ以下にある **wdreg** スクリプトを使用して、**windrvr6.o/.ko** をインストール (ロード) できます。
  wdreg <モジュール名>

  たとえば、**windrvr6** をインストールするには、次のコマンドを実行します。
  wdreg windrvr6

  ヒント: Linux の **/etc/rc.d/rc.local** ファイルに次の行を追加して **wdreg** スクリプトを起動すると、システムを再起動するごとにドライバ モジュール (**windrvr6.o/.ko**) を自動的にロード

します。
wdreg windrvr6

13.4 Solaris の動的ドライバ ロード

- 初期インストール後、Solaris で WinDriver を動的にロードするには、次のコマンドを実行します。
  /usr/sbin/add_drv windrvr6
- WinDriver を動的にアンロードするには、次のコマンドを実行します。
  /usr/sbin/rem_drv windrvr6

13.5 Windows Mobile の動的ドライバ ロード

WinDriver\redist\Windows_Mobile_5_ARMV4I\wdreg.exeユーティリティは、WinDriver カーネル モジュール (**windrvr6.dll**) を Windows Mobile プラットフォームにロードします。

ヒント: Windows Mobile では、オペレーティング システムのセキュリティ上、ブート時に未署名のドライバはロードされません。このため、ブート後に WinDriver カーネル モジュールをリロードする必要があります。Windows Mobile プラットフォームで OS 起動時に WinDriver をロードするように設定するには、**wdreg.exe** ユーティリティを **Windows\StartUp** ディレクトリにコピーします。

Windows Mobile の **wdreg.exe** ユーティリティのソースコードは、開発 PC の **WinDriver\samples\wince_install\wdreg** ディレクトリにあります。

第 14 章
ドライバの配布

この章は、ドライバ開発の最終段階です。ドライバの配布方法を紹介します。

14.1 WinDriver の有効なライセンスを取得するには

WinDriver ライセンスを取得するには、添付の申込用紙または **WinDriver\docs** ディレクトリ にある申込用紙 (**order.txt**) を使用します。必要事項をご記入の上、FAX および電子メールでエクセルソフト株式会社までご返送ください。Registered 版 (登録版) の WinDriver を開発に使用するマシンにインストールするには、および評価版で開発したドライバコードを有効にするには、セクション 4.2 で記述されているインストール手順に従ってください。

14.2 Windowsの場合

注意:

- Windows 2000 / XP / Server 2003 / Vista の場合、この章の説明の "**wdreg**" と記述している個所を "**wdreg_gui**" に置き換えることができます。同じ機能ですが、コンソール モード メッセージの代わりに GUI メッセージが表示されます。
- Windows 98 / Me の場合、この章の説明の "**wdreg**" と記述している個所を "**wdreg16**" に置き換えてください。詳細は、wdreg ユーティリティの説明、第 13 章 を参照してください。
- WinDriver のインストール ディレクトリには、**redist** と **redist_win98_compat** の二つの配布ディレクトリが含まれます。
  - **WinDriver\redist** ディレクトリには、Windows 2000 / XP / Server 2003 / Vista 用の署名入りの WHQL 互換の **windrvr6.sys** ドライバと関連 INF およびカタログ ファイルが含まれます。
  - **WinDriver\redist_win98_compat** ディレクトリには、**Windows 98 / Me / 2000 / XP / Server 2003 / Vista** 用の署名なしの **windrvr6.sys** ドライバと関連 INF が含まれます。

  Windows 98 / Me にドライバを配布する際には、このドキュメントの **WinDriver\redist** の記述を **WinDriver\redist_win98_compat** に置き換えてください。**Windows 2000 / XP / Server 2003 / Vista** 用にもこのディレクトリのファイルを使用することもできますが、これらのプラットフォームには、**WinDriver\redist** ディレクトリ以下のファイルを使用することを推奨いたします。
- **Windows 2000 / XP / Server 2003 / Vista** で WinDriver のカーネル モジュール (**windrvr6.sys**) の名前を変更する場合、**windrvr6** に関連する参照を対象のドライバ名に置き換え、**WinDriver\redist** ディレクトリへの参照を変更したインストール ファイルを含むディレクトリ

のパスに置き換えてください。たとえば、DriverWizard で生成したドライバ プロジェクトに名前を変更したドライバ ファイルを使用する場合、**WinDriver\redist** への参照を生成された **xxx_installation\redist** ディレクトリに置き換えてください (**xxx** は生成されたドライバ プロジェクトの名前です) 。DriverWizard を使用して変更したドライバ ファイルを使用する場合、**windrvr6.inf** ファイルに相当するファイルの名前を **xxx_driver.inf** に変更します。さらに、**wd900.cat** ファイルへの参照を対象のドライバ用の新しいカタログ ファイルに置き換えるか、もしくは、カタログ ファイルの配布を行わないようにします。

作成したドライバを配布するには、いくつかのステップを行う必要があります。まずドライバをターゲット システムにインストールする配布パッケージを作成します。次に、ターゲット マシンにドライバをインストールします。このプロセスは **windrvr6.sys** と **windrvr6.inf** 、デバイス用 (Plug-and-Play ハードウェア – PCI / USB 用) の INF ファイル、Kernel PlugIn ドライバ (作成した場合) をインストールします。最後に WinDriver で開発したハードウェア コントロールアプリケーション をインストールして実行します。これら全ての手順は、**wdreg** ユーティリティで行えます。

注意: このセクションでは ***.sys** ファイルの配布について説明しています。WinDriver のバージョン 6.21 以降より ***.vxd** ドライバはサポートされていません。

14.2.1 配布パッケージの用意

配布するパッケージには、次のファイルを含めます。

- ハードウェア コントロール アプリケーション / DLL
- **windrvr6.sys** (**WinDriver\redist** ディレクトリにあります)
- **windrvr6.inf** (**WinDriver\redist** ディレクトリにあります)
- **wd900.cad** (Windows 2000 / XP / Server 2003 / Vista。**WinDriver\redist** ディレクトリにあります)
- **wd_api900.dll** (**WinDriver\redist** ディレクトリにあります。32 ビット バイナリを 32 ビットのターゲット プラットフォーム、64 ビット バイナリを 64 ビットのプラットフォームに配布します。)

  wdapi900_32.dll (**WinDriver\redist** ディレクトリにあります。32 ビット バイナリを 64 ビットのターゲット プラットフォームに配布します。)
- **difxapi.dll** (**wdreg.exe** ユーティリティに必要。**WinDriver\util** ディレクトリにあります。)
- デバイス用の INF ファイル (PCI / PCMCIS / USB などの Plug-and-Play デバイスには必要です)。DriverWizard でこのファイルを生成します。詳細は、セクション 5.2 を参照してください。
- Kernel PlugIn ドライバ (**<KD ドライバ名>.sys**) (作成した場合)

14.2.2 ターゲット コンピュータにドライバをインストール

注意: ドライバをターゲット コンピュータにインストールするにはターゲット コンピュータの**管理者権限**が必要です。

以下の手順に従い、ターゲットコンピュータにドライバをインストールします。

- インストールの前に

- システムの再起動を防ぐには、**windrvr6.sys** サービスへのハンドルを開いていないことをドライバのインストール前に確認します。この手順で、このサービスを使用しているアプリケーションがない、および **windrvr6.sys** と動作するように登録されている PCI / USB デバイスへの接続がないことを確認します。つまり、この時点で、PC に接続してる PCI / USB デバイスで、このドライバと動作するようにインストールされた INF ファイル はなく、またはファイルはインストールされているが、デバイスは無効です。たとえば、古い WinDriver のバージョンで開発したドライバをアップグレードする際に、この処理が必要になります (バージョン 6.0 以降では、以前のバージョンで使用していたモジュール名が異なります)。

  このため、デバイス マネージャから WinDriver と動作するように登録されたすべての PCI / USB ドライバを無効またはアンインストールするか (Win 98 / Me では、[プロパティ] – [アンインストール] または [削除])、もしくは PC からデバイスを切断します。この処理をしない場合、**wdreg** を使用している新しいドライバのインストールを試みると、WinDriver で動作するように登録されているすべてのデバイスをアンインストールするか、あるいはインストール コマンドを正常に実行するために PC を再起動するようにと言うメッセージが表示されます。

- Windows 2000 の場合、デバイス用に作成した新しい INF ファイルをインストールする前に、古いバージョンの WinDriver で開発され、Plug-and-Play 用にインストールされている INF ファイルをすべて、**%windir%inf** から削除します。これにより、Windows が自動的に古いファイルを検出し、インストールするのを回避します。INF ディレクトリで、デバイスのベンダー ID とデバイス / プロダクト ID でデバイスの関連ファイルを検索することもできます。

- WinDriver のカーネルモ ジュールのインストール:

  ① **windrvr6.sys**、**windrvr6.inf** と wd900.cat ファイルを同じディレクトリにコピーします。
  注意: wd900.cat には、Windows 2000 / XP / Server 2003 / Vista 用のドライバ認証のデジタル署名が含まれます。インストール中に他のディレクトリにこのファイルを配置する場合には、**windrvr6.inf** ファイルの以下の行を編集して、選択したカタログ ファイルの場所を指定します:
  CatalogFile = wd900.cat
  INF ファイルのこの行をコメント アウトまたは削除して、カタログ ファイルを配布しないことも可能ですが、この場合、インストールではドライバのデジタル署名を使用しないので、推奨いたしません。

  ② **wdreg** / **wdreg16** ユーティリティを使用して、ターゲット コンピュータに WinDriver のカーネル モジュールをインストールします。
  注意: wdreg は、**difxapi.dll DLL** に依存します。

  Windows 2000 / XP / Server 2003 / Vista では、コマンド ラインから以下のように入力します。
  wdreg -inf <windrvr6.inf のパス> install

  Windows 98 / Me では、コマンド ラインから以下のように入力します。
  wdreg16 -inf <windrvr6.inf のパス> install

  たとえば、**windrvr6.inf** および **windrvr6.inf** をターゲット コンピュータの **d:\MyDevice** ディレクトリにある場合、以下のようになります。
  wdreg -inf d:\MyDevice\windrvr6.inf install

  WinDriver ツールキットの **WinDriver\util** ディレクトリ以下にがあります。このユーティリティの一般的な説明および使用方法に関しては、第 13 章 を参照してください。

注意: **wdreg** は対話型のユーティリティです。問題があるとメッセージを表示して問題を解決する方法を示します。場合によってはコンピュータの再起動を指示します。**wdreg** は対話型のユーティリティです。

注意: ドライバの配布時に、新しいバージョンの **windrvr6.sys** を Windows ドライバ ディレクトリ (**%windir%\system32\driver**) の古いバージョンのファイルで上書きしないようにご注意ください。インストール プログラムまたは INF ファイルでインストーラが自動的にタイムスタンプを比較して新しいバージョンを古いバージョンで上書きしないように設定することを推奨いたします。

- 対象のデバイスの INF ファイルのインストール (**windrvr6.sys** と動作するように登録した Plug and Play デバイス):

  ① Windows 2000 / XP / Server 2003 / Vista: **wdreg** ユーティリティを使用して、自動的に INF ファイルをロードします。

  Windows 2000 / XP / Server 2003 / Vista で対象の INF ファイルを自動的にインストールし Windows デバイス マネージャを更新するには、以下のように **wdreg** を起動して、**install** コマンドを実行します。
  wdreg -inf <対象の INF ファイルのパス> install

  preinstall コマンドを実行して、PC に接続されていないデバイスの INF ファイルを pre-install することができます。
  wdreg -inf <対象の INF ファイルのパス> preinstall

  注意: Windows 2000 では、対象のデバイスの INF ファイルを以前にインストールした場合 (WinDriver の以前のバージョンで使用した Plug and Play で動作するようにデバイスを登録)、作成した新しい INF ファイルをインストールする前に **%windir%\inf** ディレクトリからデバイスの INF ファイルをすべて削除します。このプロセスで、Windows が自動的に使用していないファイルを検出したりインストールしたりするのを防ぎます。INF ディレクトリで、デバイスのベンダー ID とデバイス / プロダクト ID でデバイスの関連ファイルを検索することもできます。

  ② Windows 98 / Me: 下記のセクション 15.1 を参考に、Windows の [新しいハードウェアの追加ウィザード] または [デバイスドライバの更新ウィザード] を使用して、手動で INF ファイルをインストールします。

- Kernel PlugIn ドライバのインストール:

  Kernel PlugIn ドライバを作成した場合は、セクション 14.2.3 の手順に従って、ドライバをインストールします。

- **Wdapi900.dll** のインストール:

  (サンプルおよび DriverWizard で生成された WinDriver のプロジェクトのように) ハードウェア コントロール アプリケーション / DLL が **wdapi900.dll** を使用する場合、この DLL をターゲットの **%windir%\system32** ディレクトリにコピーします。

  32 ビット アプリケーション / DLL を 64 ビットのターゲット プラットフォームに配布する場合は、**wdapi900_32.dll** から **wdapi900.dll** に名前を変更し、ターゲットの **%windir%\sysWOW64** ディレクトリにコピーします。

注意: 64 ビットのプログラムをインストールする 32 ビットのインストール プログラムを作成し、64 ビットの **wdapi900.dll** を **%windir%\system32** ディレクトリにコピーすると、ファイルは実際には 32 ビットの **%windir%\sysWOW64** ディレクトリにコピーされます。これは、Windows x64 プラットフォームでは 64 ビットのディレクトリを参照する 32 ビットのコマンドを、32 ビットのディレクトリを参照するように変換するためです。これを回避するには、**WinDriver\redist** ディレクトリにある **system64.exe** プログラムを使用し、64 ビットのコマンドを使用してインストールを実行します。

- ハードウェア コントロール アプリケーション / DLL のインストール:

  ハードウェア コントロール アプリケーション / DLL をターゲットにコピーして、実行します。

14.2.3 ターゲット コンピュータに Kernel PlugIn をインストール

注意: ドライバをターゲット コンピュータにインストールするにはターゲット コンピュータの管理者権限が必要です。

Kernel PlugIn ドライバを作成した場合、以下の手順に従ってください。

1. Kernel PlugIn ドライバ (**<KP ドライバ名>.sys**) をターゲット コンピュータの Windows ドライバ ディレクトリにコピーします (**%windir%\system32\drivers**)。

2. **wdreg** ユーティリティを使用して、Windows の起動時にデバイス ドライバのリストに Kernel PlugIn ドライバを追加します。次のコマンドを使用します。

   SYS Kernel PlugIn ドライバをインストールする場合:
   wdreg -name <ドライバ名 (sys 拡張子は付けません)> install

   wdreg の実行ファイルは、**WinDriver\util** ディレクトリにあります。このユーティリティの説明と使用方法は、第 13 章 を参照してください (特に、セクション 13.2.4 の「Kernel PlugIn のインストール」を参照してください)。

14.3 Windows CE の場合

14.3.1 新規の Windows CE プラットフォームへの配布

注意: 以下の手順は、Windows CE Platform Builder、または MSDEV 2005 と Windows CE 6.0 plugin を使用して Windows CE カーネル イメージをビルドするプラットフォーム開発者向けです。本手順では、これらのプラットフォームの参照を "**Windows CD IDE**" の表記を使用します。

WinDriver で開発したドライバをターゲット Windows CE プラットフォームに配布するには、次の手順に従います。

1. ターゲット ハードウェアに一致したプロジェクト レジストリ ファイルを編集します。ステップ 2 で、WinDriver コンポーネントを使用するように選択した場合、編集するレジストリ ファイルは、**WinDriver\samples\wince_install\<TARGET_CPU>\WinDriver.reg** (例えば、**WinDriver\samples\wince_install\ARMV4I\WinDriver.reg**) となります。もしくは、**WinDriver\samples\wince_install\project_wd.reg** ファイルを編集します。

2. Sysgen プラットフォームのコンパイル ステージの前に、このステップで記述されている手順に従って Windows CE プラットフォームにドライバを簡単に統合できます。

注意:

- このステップに記載されている手順は、Windows CE 4.x - 5.x with Platform Builder を使用する開発者のみに関連します。Windows CE 6.x with MSDEV 2005 を使用する開発者は次のステップ 3 に進んでください。
- この手順では、対象の Windows CE プラットフォームに WinDriver を統合する便利な方法を紹介します。この方法を使用しない場合、Sysgen ステージの後で、ステップ 4 で記述されている手動の統合ステップを実行する必要があります。
- このステップで記述されている手順で、WinDriver のカーネル モジュール (**windrvr6.dll**) を対象の OS イメージに追加します。WinDriver CE カーネル ファイル (windrvr6.dll) を永続的に Windows CE イメージ (NK.BIN) の一部とする場合にのみこのステップが必要です。例えば、フロッピーディスクを使用してターゲット プラットフォームにカーネル ファイルを移す場合などがこれに該当します。オン デマンドで CESH/PPSH サービスを通して **windrvr6.dll** をロードする場合、このステップで記述されている手順を実行しないで、ステップ 4 で記述されている手動による統合の方法を実行する必要があります。

a. Windows CD IDE を実行してプラットフォームを開きます。

b. **File** メニューから **Manage Catalog Items**.... を選択し、**Import**… ボタンをクリックし、関連する **WinDriver\samples\wince_install\<TARGET_CPU>**\ ディレクトリ (例えば、**WinDriver\samples\wince_install\ARMV4I**) から **WinDriver.cec** を選択します。
これで WinDriver のコンポーネントを Platform Builder Catalog へ追加します。

c. **Catalog** ビューで、**Third Party** ツリーの **WinDriver Component** ノードをマウスの右クリックし、**Add to OS design** を選択します。

3. 対象の Windows CE プラットフォームをコンパイルします (Sysgen ステージ)。

4. 上記のステップ 2 で記述された手順を実行しなかった場合、対象のプラットフォームに手動でドライバを統合するために、Sysgen ステージの後で、以下のステップを実行してください。
注意: 上記のステップ 2 で記述された手順を実行した場合には、このステップをスキップし、直接ステップ 5 へ進んでください。

a. Windows CD IDE を実行してプラットフォームを開きます。

b. **Build** メニューから **Open Build Release Directory** を選択します。

c. WinDriver CE カーネル ファイル - **WinDriver\redist\<TARGET_CPU>\windrvr6.dll** - を開発プラットフォーム上の **%_FLATRELEASEDIR%** サブディレクトリにコピーします。

d. **WinDriver\samples\wince_install**\ ディレクトリの **project_wd.reg** ファイルの内容を **%_FLATRELEASEDIR%** サブディレクトリの **project.reg** ファイルに追加します。

e. **WinDriver\samples\wince_install**\ ディレクトリの **project_wd.did** ファイルの内容を **%_FLATRELEASEDIR%** サブディレクトリの **project.did** ファイルに追加します。

WinDriver CE カーネル ファイル (**windrvr6.dll**) を永続的に Windows CE イメージ (**NK.BIN**) の一部とする場合にのみこのステップが必要です。例えば、フロッピーディスクを

使用してターゲット プラットフォームにカーネル ファイルを移す場合などがこれに該当します。オン デマンドで CESH/PPSH サービスを通して **windrvr6.dll** をロードする場合、永続カーネルをビルドするまでこのステップを実行する必要はありません。

5. **Build** メニューより **Make Run-Time Image** を選択し、新しいイメージ **NK.BIN** の名前をつけます。

6. ターゲット プラットフォームに新しいカーネルをダウンロードし、**Target** メニューより **Download** / **Initialize** を選択するか、またはフロッピー ディスクを使用して初期化します。

7. ターゲット CE プラットフォームを再起動します。WinDriver CE カーネルは自動的にロードします。

8. ハードウェア コントロール アプリケーション / DLL をインストールします。
ハードウェア コントロール アプリケーション / DLL が **wdapi900.dll** (WinDriver のサンプルまたは DriverWizard を使用して生成されたプロジェクトをそのまま使用する場合)、Windows ホスト開発 PC の **WinDriver\redist\WINCE\<TARGET_CPU>** ディレクトリからこの DLL をターゲットの **Windows** ディレクトリにコピーします。

14.3.2 Windows CE コンピュータへの配布

注意: 指定がない限り、このセクションの "Windows CE" の記述は、Windows Mobile を含む、対応するすべての Windows CE プラットフォームを表します。

1. WinDriver CD を Windows ホスト マシンの CD ドライブにセットします。

2. 自動インストールを終了します。

3. CD の **WINCE** ディレクトリにある **WDxxxCE.EXE** をダブル クリックします。このプログラムは必要な WinDriver のファイルをホスト開発プラットフォームにコピーします。

4. WinDriver CE カーネル モジュール - **windrvr6.dll** – Windows ホスト開発 PC の **WinDriver\redist\TARGET_CPU** ディレクトリからターゲットの CE コンピュータの **WINDOWS** ディレクトリにコピーします。

5. 起動時に Windows CE がロードするデバイス ドライバのリストに WinDriver を追加します:

   - **\WinDriver\samples\wince_install\PROJECT_WD.REG** ファイルに記載されたエントリに従って、レジストリを編集します。ハンドヘルド CE コンピュータの Windows CE Pocket Registry を使用するか、または MS eMbedded Visual C++ (Windows CE 4.x - 5.x) / MSDEV .NET 2005 (Windows Mobile または Windows CE 6.x) で提供される Remote CE Registry Editor Tool を使用して実行します。Remote CE Registry Editor ツールを使用するには、対象の Windows ホスト プラットフォームに Windows CE Services がインストールされている必要があります。

   - Windows Mobile では、起動時に OS のセキュリティ スキーマが署名されていないドライバのロードを防ぎます。従って、起動後に、WinDriver のカーネル モジュールを再ロードする必要があります。ターゲットの Windows Mobile プラットフォームで、OS の起動時に毎回、WinDriver をロードするには、
     WinDriver\redist\Windows_Mobile_5_ARMV4I\wdreg.exe ユーティリティをターゲットの Windows\StartUp\ ディレクトリにコピーします。

6. ターゲット CE コンピュータを再起動します。WinDriver CE カーネルは自動的にロードします。suspend/resume ではなく、システムの再起動を行ってください (ターゲット CE コンピュータのリセットまたは電源ボタンを使用します) 。

7. ハードウェア コントロール アプリケーション / DLL をインストールします。
ハードウェア コントロール アプリケーション / DLL が **wdapi900.dll** (WinDriver のサンプルまたは DriverWizard を使用して生成されたプロジェクトをそのまま使用する場合)、Windows ホスト開発 PC の **WinDriver\redist\WINCE\<TARGET_CPU>** ディレクトリからこの DLL をターゲットの **Windows** ディレクトリにコピーします。

14.4 Linux の場合

注意:

- Linux のカーネルは開発途中で、カーネル データ構造体はたびたび変更されます。このような流動的な開発環境をサポートし、安定したカーネルを作成するために、Linux のカーネル開発者は、カーネル 自身をコンパイルしたヘッダー ファイルと同一のヘッダーファイルを使用して、カーネル モジュールをコンパイルすることを決定しました。バージョン番号がカーネル ヘッダー ファイルに挿入され、カーネルにエンコードされているバージョンと照合されます。Linux のドライバ開発者は、ターゲット システムのカーネル バージョンでドライバを再コンパイルする必要があります。

- WinDriver のカーネル モジュール (**windrvr6.o / .ko**) の名前を変更した場合、セクション 15.2 の説明のとおり、**windrvr6** の参照を変更したドライバ名に置き換え、WinDriver の **redist/**、**lib/** および **include/** ディレクトリへの参照を関連するディレクトリのコピーへのパスに置き換えてください。
例えば、対象のドライバ プロジェクト用に DriverWizard を使用して生成されたドライバ ファイルの名前を変更して使用する場合、セクション 15.2.2.1 の説明のとおり、**WinDriver/redist** への参照を生成された **xxx_installation/redist** ディレクトリへの参照に置き換えてください (**xxx** は生成された対象のドライバ プロジェクトの名前)。

14.4.1 WinDriver Kernel モジュール

windrvr6.o/.ko はカーネル モジュールであるため、**windrvr6.o** をロードするすべてのカーネル バージョンで、再コンパイルする必要があります。この作業を簡単に行えるように、Linux カーネルから WinDriver カーネル モジュールを分離するために次のコンポーネントを提供しています。特定の指定がない限り、すべてのコンポーネントは、**WinDriver/redist** ディレクトリ以下にあります。

- **windrvr_gcc_v2.a**、**windrvr_gcc_v3.a** および **windrvr_gcc_v3_regparm.a**: WinDriver のカーネル モジュール用にコンパイルしたオブジェクト コード。**windrvr_gcc_v2.a** を GCC v2.x.x でコンパイルしたカーネル用に使用し、**windrvr_gcc_v3.a** を GCC v3.x.x でコンパイルしたカーネル用に使用します。**windrvr_gcc_v3_regparm** を **regparm** フラグで GCC v3.x.x でコンパイルしたカーネル用に使用します。

- **linux_wrappers.c/h**: WinDriver カーネル モジュールを Linux カーネルに結合するラッパー ライブラリのソース コード。

- **linux_common.h**、**windrvr.h**、**wd_ver.h** および **wdusb_interface.h**: ターゲット マシンで、WinDriver カーネル モジュールをビルドするのに必要なヘッダー ファイル。
(**wdusb_interface.h** は、USB だけでなく、PCI/PCMCIA/ISA ドライバでも必要です。)

- **wdusb_linux.c**: USB スタックを利用するために使用します。このファイルは、USB ファイルですが、PCI/PCMCIA/ISA ドライバにも必要となるファイルです。

- **configure**: **windrvr6.o/.ko** モジュールをコンパイルしカーネルへ挿入する **makefile** を作成する構成スクリプト。

- **makefile.in**、**wdreg** および **setup_inst_dir:** この構成スクリプトは makefile を作成する **makefile.in** を使用します。この makefile は **wdreg** ユーティリティ シェル スクリプトおよび **WinDriver/util** 以下にある **setup_inst_dir** を呼び出します。この 3 つともターゲットへコピーする必要があります。

ヒント: Linux の **/etc/rc.d/rc.local** ファイルに次の行を追加して **wdreg** スクリプトを起動すると、システムを再起動するごとにドライバ モジュール (**windrvr6.o/.ko**) を自動的にロードします。
wdreg windrvr6

これらのコンポーネントをドライバ のソース コードまたはオブジェクト コードと一緒に配布する必要があります。

14.4.2 ユーザーモード ハードウェア コントロール アプリケーション / 共有オブジェクト

WinDriver で作成したハードウェア コントロール アプリケーション / 共有オブジェクトをターゲットにコピーします。

(サンプルおよび DriverWizard で生成された WinDriver のプロジェクトのように) ハードウェア コントロール アプリケーション / 共有オブジェクトが **libwdapi900.so** を使用する場合、この共有オブジェクトを開発用 PC の **WinDriver/lib** ディレクトリから、ターゲットのライブラリ ディレクトリ (32 ビット PowerPC、32 ビット x86 、IA64 の場合は **/usr/lib/**、64 ビット x86 の場合は **/usr/lib64**) にコピーします。

ユーザー モード ハードウェア コントロール アプリケーション / 共有オブジェクト はカーネル バージョン番号と一致する必要はないので、バイナリ コード (ソース コードの無許可のコピーを防ぎたい場合) または、ソース コードとして配布してもかまいません。ただし、Jungo 社とのソフトウェア ライセンス契約により、libwdapi900.so 共有オブジェクトのソースコードを配布することは禁じられている点に注意してください。

注意: ソース コードとして配布する場合、コード中に使用してる WinDriver のライセンス コードを配布しないように注意してください。

14.4.3 Kernel Plugin モジュール

Kernel Plugin モジュール (作成した場合) はカーネル モジュールのため、有効なカーネルのバージョン番号と一致する必要があります。したがって、ターゲット システム用に再コンパイルが必要になります。ユーザーが再コンパイルできるように Kernel Plugin モジュールのソース コードを配布することをお勧めします。Kernel PlugIn のコード生成で Driver Wizard が生成した **configure** スクリプトを使用して、Kernel PlugIn モジュールをビルドし、配布する Kernel PlugIn モジュールに挿入できます。

注意: configure スクリプトは、ファイルの場所 (パス) など、調整を必要とすることがあります。

さまざまな Linux ターゲットで Kernel PlugIn ドライバを再コンパイルできるように、次のファイルを配布することができます:
kp_linux_gcc_v2.o、**kp_linux_gcc_v3.o**、**kp_linux_gcc_v3_regparm.o**、

kp_wdapi900_gcc_v2.a、**kp_wdapi900_gcc_v3.a** および **kp_wdapi900_gcc_v3_regparm.a**

xxx_gcc_v2.o/a、**xxx_gcc_v3.o/a**、**xxx_gcc_v3_regparm.o/a** ファイルはそれぞれ、GCC v2.x.x、GCC v3.x.x、GCC v3.x.x (**regparm** フラグを使用) で、カーネルをコンパイルする場合に使用されます。

14.4.4 インストール スクリプト

作成したドライバの実行ファイル / DLL を適切な場所 (**/usr/local/bin** など) にコピーするインストール シェル スクリプトを提供し、**make** または **gmake** を起動して、WinDriver のカーネル モジュールおよび Kernel PlugIn モジュールをビルドしインストールすることをお勧めします。

14.5 Solaris の場合

注意: WinDriver のカーネル モジュール (**windrvr6.o / .ko**) の名前を変更した場合、セクション 15.2 の説明のとおり、**windrvr6** の参照を変更したドライバ名に置き換え、WinDriver の **redist/**、**lib/** および **include/** ディレクトリへの参照を関連するディレクトリのコピーへのパスに置き換えてください。
例えば、対象のドライバ プロジェクト用に DriverWizard を使用して生成されたドライバ ファイルの名前を変更して使用する場合、セクション 15.2.2.1 の説明のとおり、**WinDriver/redist** への参照を生成された **xxx_installation/redist** ディレクトリへの参照に置き換えてください (**xxx** は生成された対象のドライバ プロジェクトの名前)。

Solaris の場合、作成したドライバをユーザーがターゲットにインストールをできるように、次のものを提供する必要があります。

- **WinDriver カーネル モジュール:** WinDriver カーネル モジュールを実装する **windrvr6** と **windrvr6.conf** ファイル。
- **ユーザーモード ハードウェア コントロール アプリケーション / 共有オブジェクト:** 作成したユーザーモード ハードウェア コントロール アプリケーション / 共有オブジェクト バイナリ。
- (サンプルおよび DriverWizard で生成された WinDriver のプロジェクトのように) ハードウェア コントロール アプリケーション / 共有オブジェクトが **libwdapi900.so** を使用する場合、この共有オブジェクトを開発用 PC の **WinDriver/lib** ディレクトリから、ターゲットのライブラリ ディレクトリ (32 ビット SPARC、32 ビット x86 の場合は **/lib/32**、64 ビット SPARC の場合は **/lib/64**) にコピーします。
- **Kernel Plugin モジュール:** Kernel Plugin モジュールを使用した場合、**mykp** や **mykp.cnf** などの関連ファイルを提供する必要があります。
- **インストール スクリプト:** ドライバの実行ファイルを適切な場所 (**/usr/local/bin** など) にコピーするインストール シェル スクリプトを提供し、WinDriver カーネルをインストールすることをお勧めします。必要に応じて、(開発環境マシンの WinDriver ディレクトリにある) **install_windrvr6** ユーティリティ スクリプトを採用してください。

第 15 章
ドライバのインストール - 高度な問題

15.1 INF ファイル - Windows 98 / Me / 2000 / XP / Server 2003 / Vista

デバイス情報ファイル (INF) は、Windows 98 / Me / 2000 / XP / Server 2003 / Vista の「Plug-and-Play」機能で使用される情報を提供するテキスト ファイルです。このファイルを使用して、ハードウェア デバイスをサポートするためのソフトウェアをインストールします。INF ファイルは、USB や PCI などのハードウェアを識別するのに必要です。INF ファイルには、デバイスとインストールするファイルに関する必要な情報がすべて含まれています。ハードウェア メーカーは新製品を提供する際に、デバイス クラスごとに必要なリソースとファイルを明確に定義する INF ファイルを作成する必要があります。

デバイスによっては、オペレーティング システムで INF ファイルが提供されています。そうでない場合は、デバイス用の INF ファイルを作成する必要があります。DriverWizard は、デバイス用の INF ファイルを生成します。INF ファイルは、対象デバイスの処理を WinDriver が行うことを OS に通知するのに使用されます。

USB デバイスの場合、最初に windrvr6.sys と動作するようにデバイスを登録しないと、WinDriver で (DriverWizard またはコードから) デバイスにアクセスすることはできません。デバイスの INF ファイルをインストールすると、デバイスが登録されます。DriverWizard は、デバイスの INF ファイルを自動的に生成することができます。DriverWizard を使用して、セクション 5.2 で説明するように開発マシンで INF ファイルを生成し、次のセクションで説明するようにドライバを配布するマシンにインストールすることができます。

15.1.1 なぜ INF ファイルを作成する必要があるのか

- 起動時に Windows オペレーティング システムの [新しいハードウェアの検出ウィザード] が表示されないようにするため。
- Windows 98 / Me / 2000 / XP / Server 2003 / Vista 上で OS により PCI 設定レジスタが初期化されるのを確認するため (PCI の場合のみ)。
- Plug-and-Play システムにインストールする Plug-and-Play ハードウェアの新しいドライバ を開発する際に、INF ファイルを作成する必要があるため。
- 既存のドライバを新規のものに置き換えるため。
- WinDriver および DriverWizard の USB デバイスへのアクセスを有効にするため (USB の場合のみ)。
- OS により USB デバイスに物理アドレスが確実に割り当てられるようにするため (USB の場合のみ)。
- デバイス用に作成された新しいドライバをロードするため。

15.1.2 ドライバがない場合に INF ファイルをインストールするには

注意: Windows 98 / Me / 2000 / XP / Server 2003 / Vista で INF ファイルをインストールするには、管理者権限が必要です。

- Windows 2000 / XP / Server 2003 / Vista の場合

  Windows 2000 / XP / Server 2003 / Vista 上では、**wdreg** ユーティリティと **install** コマンドを使用して、INF ファイルを自動的にインストールできます。
  wdreg -inf <INF ファイルのパス> install

  詳細は、セクション 13.2.2 を参照してください。

  開発 PC では、DriverWizard で INF ファイルを生成する際に、[INF generation] ウィンドウの [Automatically Install the INF file] チェックボックスをオンにすることによって、自動的に INF ファイルをインストールできます (セクション 5.2 を参照)。

  Windows 2000 / XP / Server 2003 / Vista では、次のいずれかの方法で INF ファイルを手動でインストールすることもできます。

  - Windows の [新しいハードウェアの検出ウィザード]: このウィザードは、デバイスを接続したとき、またはデバイスが既に接続済みで、デバイス マネージャがハードウェアの変更をスキャンしたときに有効になります。
  - Windows の [ハードウェアの追加ウィザード]: [スタート] メニューから [コントロール パネル] - [ハードウェアの追加] を選択します。
  - Windows の [ハードウェアの更新ウィザード]: [マイコンピュータ] を右クリックして [プロパティ] を選択し、[ハードウェア] タブで [デバイス マネージャ] をクリックします。デバイスを右クリックして [プロパティ] を選択し、[ドライバ] タブで [ドライバの更新...] をクリックします。Windows 2000 / XP / Server 2003 / Vista では、[デバイス マネージャ] でデバイスを右クリックして、[ドライバの更新...] を直接選択できます。

  手動でのインストール方法では、インストール中に INF ファイルの場所を指定する必要があります。手動でインストールするのではなく、wdreg ユーティリティを使用して、自動的に INF ファイルをインストールすることを推奨します。

- Windows 98 / Me の場合

  Windows 98 / Me では、下記で説明するように、Windows の [ハードウェアの追加ウィザード] または [ハードウェアの更新ウィザード] を使用して、手動で INF ファイルをインストールする必要があります。

  - Windows の [ハードウェアの追加ウィザード]:

    注意: この方法は、対象デバイスのドライバがインストールされていない場合、またはインストール済みのドライバをアンインストール (削除) した後に使用することができます。それ以外の場合、[ハードウェアの追加ウィザード] を有効にする Windows の [新しいハードウェアの検出ウィザード] は表示されません。

    i. Windows の [ハードウェアの追加ウィザード] を有効にするには、コンピュータにハードウェア デバイスを装着します。デバイスが既に装着されている場合、ハードウェアの変更をスキャン (更新) します。

ii. Windows の [ハードウェアの追加ウィザード] が表示されたら、画面のインストール手順に従ってください。必要に応じて、配布パッケージの INF ファイルの場所を指定します。

- Windows の [ハードウェアの更新ウィザード]:

  i. Windows の [デバイス マネージャ] を開く: [マイコンピュータ] を右クリックして [プロパティ] を選択し、[ハードウェア] タブで [デバイス マネージャ] ボタンをクリックします。

  ii. デバイスを右クリックして [プロパティ] を選択し、[ドライバ] タブで [ドライバの更新] ボタンをクリックします。

  PCI の場合、[デバイス マネージャ] でデバイスの場所を特定するには、[表示] - [デバイス (接続別)] を選択します。PCI デバイスの場合、[標準 PC] - [PCI バス] - [対象デバイス] を参照します。

  USB の場合、[デバイス マネージャ] でデバイスの場所を特定するには、[表示] - [デバイス (接続別)] を選択して、[標準 PC] - [PCI バス] - [USB ユニバーサル ホスト コントローラ (または、OHCI/EHCI など使用しているその他のコントローラ) への PCI] - [USB ルート ハブ] - [対象デバイス] を参照します。

  iii. 表示される [ハードウェアの更新ウィザード] の指示に従ってください。必要に応じて、配布パッケージの INF ファイルの場所を指定します。

15.1.3 INF ファイルを使用して既存のドライバを置き換えるには

注意: Windows 98 / Me / 2000 / XP / Server 2003 / Vista でドライバを置き換えるには、管理者権限が必要です。

1. Windows 2000 で、WinDriver の以前のバージョンで動作するように登録された PCI/PCMCIA デバイスまたは USB デバイスを更新する場合、新しく作成した INF ファイルの代わりに古い INF ファイルがインストールされるのを防ぐために、Windows INF ディレクトリ (%windir%inf) からデバイスの古い INF ファイルをすべて削除することを推奨します。対象デバイスのベンダー ID とデバイス ID を含むファイルを検索して、削除してください。

1. INF ファイルをインストールします。

   - Windows 2000 / XP / Server 2003 / Vista では、自動的に INF ファイルをインストールします。

     `wdreg` ユーティリティと **`install`** コマンドを使用して、Windows 2000 / XP / Server 2003 / Vista に自動的に INF ファイルをインストールできます。
     `wdreg -inf <INF ファイルのパス> install`

     詳細は、セクション 13.2.2 を参照してください。

     開発 PC では、DriverWizard で INF ファイルを生成する際に、[INF generation] ウィンドウの [Automatically Install the INF file] チェックボックスをオンにすることによって、自動的に INF ファイルをインストールできます (セクション 5.2 を参照)。

     Windows 2000 / XP / Server 2003 / Vista では、次のいずれかの方法で INF ファイルを手動でインストールすることもできます。

– Windows の [新しいハードウェアの検出ウィザード]: このウィザードは、デバイスを接続したとき、またはデバイスが既に接続済みで、デバイス マネージャがハードウェアの変更をスキャンしたときに有効になります。

– Windows の [ハードウェアの追加ウィザード]: [スタート] メニューから [コントロール パネル] - [ハードウェアの追加] を選択します。

– Windows の [ハードウェアの更新ウィザード]: [マイコンピュータ] を右クリックして [プロパティ] を選択し、[ハードウェア] タブで [デバイス マネージャ] をクリックします。デバイスを右クリックして [プロパティ] を選択し、[ドライバ] タブで [ドライバの更新...] をクリックします。Windows 2000 / XP / Server 2003 / Vista では、[デバイス マネージャ] でデバイスを右クリックして、[ドライバの更新...] を直接選択できます。

手動でのインストール方法では、インストール中に INF ファイルの場所を指定する必要があります。インストール ウィザードのデフォルトの INF ファイルとは別の INF ファイルをインストールする場合、[他のドライバをインストールする] を選択して一覧から INF ファイルを選択します。手動でインストールするのではなく、wdreg ユーティリティを使用して、自動的に INF ファイルをインストールすることを推奨します。

○ Windows 98 / Me では、下記で説明するように、Windows の [ハードウェアの追加ウィザード] または [ハードウェアの更新ウィザード] を使用して、手動で INF ファイルをインストールする必要があります。

– Windows の [ハードウェアの追加ウィザード]:

注意: この方法は、対象デバイスのドライバがインストールされていない場合、またはインストール済みのドライバをアンインストール (削除) した後に使用することができます。それ以外の場合、[ハードウェアの追加ウィザード] を有効にする Windows の [新しいハードウェアの検出ウィザード] は表示されません。

i. Windows の [ハードウェアの追加ウィザード] を有効にするには、コンピュータにハードウェア デバイスを装着します。デバイスが既に装着されている場合、ハードウェアの変更をスキャン (更新) します。

ii. Windows の [ハードウェアの追加ウィザード] が表示されたら、画面のインストール手順に従ってください。必要に応じて、配布パッケージの INF ファイルの場所を指定します。

– Windows の [ハードウェアの追加ウィザード]:

i. Windows の [デバイス マネージャ] を開く: [マイコンピュータ] を右クリックして [プロパティ] を選択し、[ハードウェア] タブで [デバイス マネージャ] ボタンをクリックします。

ii. デバイスを右クリックして [プロパティ] を選択し、[ドライバ] タブで [ドライバの更新] ボタンをクリックします。

PCI の場合、[デバイス マネージャ] でデバイスの場所を特定するには、[表示] - [デバイス (接続別)] を選択します。PCI デバイスの場合、[標準 PC] - [PCI バス] - [対象デバイス] を参照します。

USB の場合、[デバイス マネージャ] でデバイスの場所を特定するには、[表示] - [デバイス (接続別)] を選択して、[標準 PC] - [PCI バス] - [USB ユニバーサル ホスト コントローラ (または、OHCI/EHCI など使用しているその他のコントローラ) への PCI] - [USB ルート ハブ] - [対象デバイス] を参照します。

iii. 表示される [ハードウェアの更新ウィザード] の指示に従ってください。必要に応じて、配布パッケージの INF ファイルの場所を指定します。

15.2 WinDriver カーネル ドライバの名前変更

WinDriver API は、主要なドライバ機能を提供し、ユーザーモードから特定のドライバ ロジックをコード化できます [1.6]。**windrvr6.sys/.dll/.o/.ko** (OS により異なる) カーネル ドライバ モジュール内に実装されています。

Windows、Linux、および Solaris では、WinDriver カーネル モジュールの名前を任意のドライバ名に変更し、**windrvr6.sys/.o/.ko** の代わりに配布することができます。次のセクションでは、サポートしている OS ごとにドライバ名の変更方法を説明しています。

名前変更した WinDriver カーネル ドライバは、**windrvr6.sys/.o/.ko** カーネル モジュールと同じ PC にインストールできます。また、名前変更した複数の WinDriver ドライバを同じ PC にインストールすることもできます。

ヒント: インストール PC の他のドライバとの競合を回避するために、ドライバには一意な名前を付けてください。

15.2.1 Windows ドライバの名前変更

次のセクションで説明する 2 つの方法のいずれかを使用して、WinDriver Windows カーネル ドライバ (**windrvr6.sys**) の名前を変更します。

ヒント: 作業のほとんどが自動化されているため、セクション 15.2.1.1 で説明する方法を推奨します。

注意: このセクションの **WinDriver\redist** ディレクトリへの参照は、**WinDriver\redist_win98_compat** に置き換えることができます。**redist** ディレクトリには、署名済みの WHQL 認定ドライバと関連ファイルが含まれています。ドライバの名前を変更する場合、必要に応じて、セクション 15.3 で説明するようにドライバの署名を再度取得する必要があります。**redist_win98_compat** ディレクトリには、WHQL によって認定されていない Windows 98 / Me 以降用のドライバが含まれています。

15.2.1.1 DriverWizard を使用する Windows ドライバの名前変更

このセクションの手順に従い DriverWizard を使用して WinDriver Windows カーネル ドライバの名前を変更できます (推奨)。

このセクションの **xxx** への参照は、生成される DriverWizard ドライバ プロジェクトの名前に置き換える必要があります。

1. DriverWizard ユーティリティを使用して、Windows ハードウェア用のドライバ コードを生成します [5.2 (①)]。生成されるドライバ プロジェクトの名前には、ドライバ名 (**xxx**) が使用されます。

   生成されるプロジェクト ディレクトリ (**xxx**) には、**xxx_installation** ディレクトリと以下のファイルおよびディレクトリが含まれます。

- **redist** ディレクトリ

  - **xxx.sys** - 新しいドライバ。**windrvr6.sys** ドライバのコピーを名前変更したもの。

    注意: 生成されるドライバ ファイルのプロパティ (ファイルのバージョン、会社名、など) は、**windrvr6.sys** ドライバのプロパティと同じです。以下に説明するように、生成される **xxx_installation\sys** ディレクトリのファイルを使用して、新しいプロパティでドライバをリビルドすることができます。

  - **xxx_driver.inf** - 変更済み **windrvr6.inf** ファイル。新しい **xxx.sys** ドライバのインストールに使用します。必要に応じて、ファイル内の文字列やコメントの定義を変更するなど、追加の変更を加えることができます。

  - **xxx_device.inf** - DriverWizard で生成される標準的なデバイスの INF ファイルを修正したもの。デバイスとドライバ (**xxx.sys**) を登録します。必要に応じて、メーカー名やドライバの提供元を変更するなど、追加の変更を加えることができます。

  - **wdapi900.dll** - WinDriver API DLL のコピー。DLL は、ドライバの配布を簡単にするために、ここにコピーされます。ドライバのメイン インストール ディレクトリとして、**WinDriver\redist** ディレクトリの代わりに、生成される **xxx\redist** ディレクトリを使用できるようになります。

- **sys** ディレクトリ: このディレクトリには、ドライバ ファイルのプロパティを変更するための、上級者ユーザー向けのファイルが含まれています。

  注意: ファイルのプロパティを変更する場合、Windows Driver Development Kit (DDK) を使用してドライバ モジュールをリビルドする必要があります。

  xxx.sys ドライバ ファイルのプロパティを変更するには:

  i. 開発 PC またはネットワーク上に Windows DDK がインストールされていることを確認して、BASEDIR 環境変数を DDK インストール ディレクトリに設定します。

  ii. 別のドライバ ファイル プロパティを設定するために、生成される sys\ ディレクトリの **xxx.rc** リソース ファイルを変更します。

  iii. 次のコマンドを実行してドライバをリビルドします。
  ddk_make <OS> <ビルド モード (free/checked)>

  たとえば、Windows XP 用ドライバのリリース版をビルドするには、次のコマンドを実行します。
  ddk_make winxp free

  注意: **ddk_make.bat** ユーティリティは **WinDriver\util** ディレクトリにあります。インストール コマンドを実行すると、Windows によって自動的に識別されます。

xxx.sys ドライバをリビルドした後に、生成される **xxx\redist** ディレクトリに新しいドライバ ファイルをコピーします。

2. WinDriver 関数を呼び出す前に、新しいドライバ名を使用して、アプリケーションで `WD_DriverName()` 関数を呼び出せるか確認してください。

サンプルおよび DriverWizard で生成される WinDriver アプリケーションには、既にこの関数の呼び出しが含まれています。ただし、デフォルトのドライバ名 (**windrvr6**) を使用しているため、関数に渡すドライバ名を新しいドライバ名に変更する必要があります。

3. ユーザーモード ドライバ プロジェクトが、`WD_DRIVER_NAME_CHANGE` プリプロセッサ フラグ (例: え) を使用してビルドされていることを確認してください。

   注意: サンプルおよび DriverWizard で生成される WinDriver プロジェクトおよび makefile では、デフォルトでこのプリプロセッサ フラグが設定されています。

4. セクション 14.2 の手順に従い WinDriver インストール ファイルの代わりに、生成される **xxx_installation** ディレクトリの変更済みファイルを使用して、新しいドライバをインストールします。

15.2.1.2 手動による Windows ドライバの名前変更

次の手順に従って、手動で WinDriver Windows カーネル ドライバの名前を変更できます。

1. **WinDriver\redist** ディレクトリのコピーを作成して、次のようにファイルを変更します。

   - **windrvr6.sys** および **windrvr6.inf** の名前を変更します。ファイル名の **windrvr6** という部分を選択したドライバ名 (例: **my_driver**) に置き換えます。

     ファイルの拡張子 (***.sys / *.inf**) はそのまま残してください。

   - 名前変更した **windrvr6.inf** ファイルを変更します。

     i. ファイル内の **windrvr6** という部分をすべて新しいドライバ名 (例: **my_driver**) に置き換えます。

     ii. INF ファイルの [`DriverInstall.NT.Services`] 以下ににある `AddService` キーのドライバ サービスの名前を選択したドライバ名 (例: **MyDriver**) に変更します。

2. デバイスと選択したドライバを登録するデバイスの INF ファイル内にある **windrvr6** という部分をすべて新しいドライバ名 (例: **my_driver**) に置き換えます。必要に応じて、メーカー名やドライバの提供元を変更するなど、追加の変更を加えることもできます。

   注意: DriverWizard は、生成するドライバ プロジェクト ディレクトリに、自動的にデバイスの INF を生成します。ファイルの名前には、ドライバ プロジェクトの名前 (例: **<my_driver>.inf**) が使用されます。ホストドライバ プロジェクトの作成を選択すると表示される DriverWizard の [デバイスの選択] 画面で [INF ファイルの生成] オプションを選択して、デバイスの INF ファイルを生成することもできます。この方法では、セクション 5.2 (3) で説明するように、ウィザードから直接独自の INF 文字列 (メーカー名、デバイス名、デバイス クラスなど) を指定して、DriverWizard でデバイスの INF を生成できます。デバイスの INF ファイルの生成方法に関わらず、新しいドライバ モジュールを使用するには、上記の説明のように、**windrvr6** という部分を新しいドライバ名に変更する必要があります。

3. WinDriver 関数を呼び出す前に、新しいドライバ名を使用して、アプリケーションで `WD_DriverName()` 関数を呼び出せるか確認してください。

   サンプルおよび DriverWizard で生成される WinDriver アプリケーションには、既にこの関数の呼び出しが含まれています。ただし、デフォルトのドライバ名 (**windrvr6**) を使用しているため、関数に渡すドライバ名を新しいドライバ名に変更する必要があります。

4. ユーザーモード ドライバ プロジェクトが、WD_DRIVER_NAME_CHANGE プリプロセッサ フラグ (例: -DWD_DRIVER_NAME_CHANGE) を使用してビルドされていることを確認してください。

   注意: サンプルおよび DriverWizard で生成される WinDriver プロジェクトおよび makefile では、デフォルトでこのプリプロセッサ フラグが設定されています。

5. セクション 14.2 の手順に従い WinDriver インストール ファイルの代わりに、新しいインストール ディレクトリの変更済みファイルを使用して、新しいドライバをインストールします。

15.2.2 Linux ドライバの名前変更

次のセクションで説明する 2 つの方法のいずれかを使用して、WinDriver Linux カーネル ドライバ (**windrvr6.o/.ko**) の名前を変更します。

ヒント: 作業のほとんどが自動化されているため、セクション 15.2.2.1 で説明する方法を推奨します。

15.2.2.1 DriverWizard を使用する Linux ドライバの名前変更

このセクションの手順に従い DriverWizard を使用して WinDriver Linux カーネル ドライバの名前を変更できます (推奨)。

このセクションの **xxx** への参照は、生成される DriverWizard ドライバ プロジェクトの名前に置き換える必要があります。

1. DriverWizard ユーティリティを使用して、Linux ハードウェア用のドライバ コードを生成します [5.2 (①)]。生成されるドライバ プロジェクトの名前には、ドライバ名 (**xxx**) が使用されます。

   生成されるプロジェクト ディレクトリ (**xxx/**) には、**xxx_installation/** ディレクトリと以下のファイルおよびディレクトリが含まれます。

   - **redist/** ディレクトリ: このディレクトリには、**WinDriver/redist** インストール ディレクトリ内のファイルのコピーが含まれています。ただし、**windrvr6.o/.ko** の代わりに、**xxx.o/.ko** ドライバをビルドするように変更されています。
   - **lib/** ディレクトリおよび **include/** ディレクトリこれらのディレクトリは、WinDriver の library および include ディレクトリのコピーです。サポートしている WinDriver Linux カーネル ドライバのビルド方法では、**redist/** ディレクトリと同じ親ディレクトリの直下にこれらのディレクトリを必要とするため、ここにコピーが作成されます。

2. WinDriver 関数を呼び出す前に、新しいドライバ名を使用して、アプリケーションで WD_DriverName() 関数を呼び出せるか確認してください。

   サンプルおよび DriverWizard で生成される WinDriver アプリケーションには、既にこの関数の呼び出しが含まれています。ただし、デフォルトのドライバ名 (**windrvr6**) を使用しているため、関数に渡すドライバ名を新しいドライバ名に変更する必要があります。

3. ユーザーモード ドライバ プロジェクトが、WD_DRIVER_NAME_CHANGE プリプロセッサ フラグ (例: -DWD_DRIVER_NAME_CHANGE) を使用してビルドされていることを確認してください。

   注意: サンプルおよび DriverWizard で生成される WinDriver プロジェクトおよび makefile では、デフォルトでこのプリプロセッサ フラグが設定されています。

4. セクション 14.4 の手順に従い WinDriver インストール ファイルの代わりに、生成される **xxx_installation/** ディレクトリの変更済みファイルを使用して、新しいドライバをインストールします。インストールの一部として、セクション 14.4.1 の手順に従い新しいインストール ディレクトリのファイルを使用して、新しいカーネル ドライバ モジュールをビルドします。

15.2.2.2 手動による Linux ドライバの名前変更

次の手順に従って、手動で WinDriver Linux カーネル ドライバの名前を変更できます。

1. 新しいインストール ディレクトリを作成して、WinDriver の **redist/**、**lib/**、および **include/** ディレクトリをコピーします。

2. コピーした **redist/** ディレクトリで、次のようにファイルを変更します。

   ① **linux_wrappers.c** ファイル内の **%DRIVER_NAME%** という文字列を新しいドライバ名 (例: **my_driver**) に置き換えます。

   ② 新しいディレクトリにある構成スクリプトを **WinDriver/wizard/.windrvr6_configure.src** ファイルのコピーに置き換えて、ファイル名を **configure** とします。そして、このファイル内の **%DRIVER_NAME_CHANGE_FLAG%** という文字列を **-DWD_DRIVER_NAME_CHANGE** 変更します。

   注意: **lib/** ディレクトリと **include/** ディレクトリのコピーは変更しないでください。サポートしている WinDriver Linux カーネル ドライバのビルド方法では、**redist/** ディレクトリと同じ親ディレクトリの直下にこれらのディレクトリを必要とするため、これらのファイルがコピーされます。

3. WinDriver 関数を呼び出す前に、新しいドライバ名を使用して、アプリケーションで `WD_DriverName()` 関数を呼び出せるか確認してください。

   サンプルおよび DriverWizard で生成される WinDriver アプリケーションには、既にこの関数の呼び出しが含まれています。ただし、デフォルトのドライバ名 (**windrvr6**) を使用しているため、関数に渡すドライバ名を新しいドライバ名に変更する必要があります。

4. ユーザーモード ドライバ プロジェクトが、`WD_DRIVER_NAME_CHANGE` プリプロセッサ フラグ (例: `-DWD_DRIVER_NAME_CHANGE`) を使用してビルドされていることを確認してください。

   注意: サンプルおよび DriverWizard で生成される WinDriver プロジェクトおよび makefile では、デフォルトでこのプリプロセッサ フラグが設定されています。

5. セクション 14.4 の手順に従い WinDriver インストール ファイルの代わりに、新しいインストール ディレクトリの変更済みファイルを使用して、新しいドライバをインストールします。インストールの一部として、セクション 14.4.1 の手順に従い新しいインストール ディレクトリのファイルを使用して、新しいカーネル ドライバ モジュールをビルドします。

15.2.3 Solaris ドライバの名前変更

次のセクションで説明する 2 つの方法のいずれかを使用して、WinDriver Solaris カーネル ドライバ (**windrvr6**) の名前を変更します。

ヒント: 作業のほとんどが自動化されているため、セクション 15.2.3.1 で説明する方法を推奨します。

15.2.3.1 DriverWizard を使用する Solaris ドライバの名前変更

このセクションの手順に従い DriverWizard を使用して WinDriver Solaris カーネル ドライバの名前を変更できます (推奨)。

このセクションの **xxx** への参照は、生成される DriverWizard ドライバ プロジェクトの名前に置き換える必要があります。

1. DriverWizard ユーティリティを使用して、Solaris ハードウェア用のドライバ コードを生成します [5.2 (①)]。生成されるドライバ プロジェクトの名前には、ドライバ名 (**xxx**) が使用されます。

   生成されるプロジェクト ディレクトリ (**xxx/**) には、**xxx_installation/** ディレクトリと以下のファイルおよびディレクトリが含まれます。

   - **redist/** ディレクトリこのディレクトリには、**WinDriver/redist** ディレクトリのファイル (例: **windrvr6** カーネル モジュール、**windrvr6.conf** インストール ファイル) を新しいドライバ名 (**xxx**) を使用するように変更したものが含まれています。
   - **lib/** ディレクトリこのディレクトリは WinDriver の **lib/** ディレクトリのコピーです。ドライバのビルド方法では、**redist/** ディレクトリと同じ親ディレクトリの直下にこのディレクトリを必要とするため、ここにコピーが作成されます。
   - **install_xxx** スクリプトおよび **remove_xxx** スクリプト **WinDriver/** ディレクトリにある **install_windrvr** ファイルおよび **rename_windrvr** ファイルのコピーです。新しいドライバ名を参照するように変更されています。

2. WinDriver 関数を呼び出す前に、新しいドライバ名を使用して、アプリケーションで `WD_DriverName()` 関数を呼び出せるか確認してください。

   サンプルおよび DriverWizard で生成される WinDriver アプリケーションには、既にこの関数の呼び出しが含まれています。ただし、デフォルトのドライバ名 (**windrvr6**) を使用しているため、関数に渡すドライバ名を新しいドライバ名に変更する必要があります。

3. ユーザーモード ドライバ プロジェクトが、`WD_DRIVER_NAME_CHANGE` プリプロセッサ フラグ (例: `-DWD_DRIVER_NAME_CHANGE`) を使用してビルドされていることを確認してください。

   注意: サンプルおよび DriverWizard で生成される WinDriver プロジェクトおよび makefile では、デフォルトでこのプリプロセッサ フラグが設定されています。

4. セクション 14.5 の手順に従い WinDriver インストール ファイルの代わりに、生成される **xxx_installation/** ディレクトリの変更済みファイルを使用して、新しいドライバをインストールします。

15.2.3.2 手動による Solaris ドライバの名前変更

次の手順に従って、手動で WinDriver Solaris カーネル ドライバの名前を変更できます。

1. 新しいインストール ディレクトリを作成して、WinDriver の **redist/** ディレクトリ、**lib/** ディレクトリ、**install_windrvr** ファイル、および **remove_windrvr** ファイルをコピーします。

2. コピーした **redist/** ディレクトリ内のファイル、**install_windrvr** ファイル、**remove_windrvr** ファイルの **windrvr6** または **windrvr** という部分すべて (ファイル名とファイル内の文字列の両方) を選択したドライバ名 (例: **my_driver**) に置き換えます。

注意: **lib/** ディレクトリのコピーは変更しないでください。ドライバのビルド方法では、**redist/** ディレクトリと同じ親ディレクトリの直下にこのディレクトリを必要とするため、ここにコピーが作成されます。

3. WinDriver 関数を呼び出す前に、新しいドライバ名を使用して、アプリケーションで `WD_DriverName()` 関数を呼び出せるか確認してください。

   サンプルおよび DriverWizard で生成される WinDriver アプリケーションには、既にこの関数の呼び出しが含まれています。ただし、デフォルトのドライバ名 (**windrvr6**) を使用しているため、関数に渡すドライバ名を新しいドライバ名に変更する必要があります。

4. ユーザーモード ドライバ プロジェクトが、`WD_DRIVER_NAME_CHANGE` プリプロセッサ フラグ (例: `-DWD_DRIVER_NAME_CHANGE`) を使用してビルドされていることを確認してください。

   注意: サンプルおよび DriverWizard で生成される WinDriver プロジェクトおよび makefile では、デフォルトでこのプリプロセッサ フラグが設定されています。

5. セクション 14.5 の手順に従い WinDriver インストール ファイルの代わりに、新しいインストール ディレクトリの変更済みファイルを使用して、新しいドライバをインストールします。

15.3 WHQL 認証とドライバーの署名 - Windows 2000 / XP / Server 2003 / Vista

15.3.1 概要

Microsoft Windows ロゴ プログラム (http://www.microsoft.com/whdc/winlogo/default.mspx) では、ハードウェア モジュールおよびソフトウェア モジュール (ドライバを含む) に対し Microsoft の資格認定を取得するためのの手順について説明しています。テストに Pass したハードウェアまたはソフトウェアは、Microsoft 資格認定を取得することができます。

既存の Windows オペレーティング システムのほとんどでは、Microsoft により認定、デジタル署名されていなくても、ドライバを正常にインストールすることができます。ただし、署名を取得することには次のようなメリットがあります。

- 未署名ドライバのインストールがブロックされるターゲットへのドライバのインストール。
- ドライバ インストール時の Windows の未署名ドライバに関する警告の回避。
- Windows XP 以降での INF ファイル [15.1] のプレインストール。

次の Web サイトで説明されているように、Windows Vista オペレーティング システムでは、デジタル コード署名の重要性がさらに拡張されています。
http://www.microsoft.com/whdc/winlogo/drvsign/drvsign.mspx

デバイスドライバの認定を取得する際は、対象ハードウェアと共に申し込む必要があります。デジタル署名および認定を取得するには、Microsoft の Windows Hardware Quality Labs (WHQL) テストにドライバとハードウェアを提出してください。

WHQL 認定プロセスに関する詳細は、次の Microsoft の Web サイトを参照してください。

- WHQL ホームページ:
  http://www.microsoft.com/whdc/whql/default.mspx
- WHQL ポリシー ページ:
  http://www.microsoft.com/whdc/whql/policies/default.mspx
- Windows Quality Online Services (Winqual) ホームページ:
  https://winqual.microsoft.com/
- Winqual ヘルプ:
  https://winqual.microsoft.com/Help/
- WHQL テスト、手順書、およびフォームのダウンロード ページ:
  http://www.microsoft.com/whdc/whql/WHQLdwn.mspx
- Windows Driver Kit (WDK):
  http://www.microsoft.com/whdc/devtools/wdk/default.mspx
- Driver Test Manager (DTM):
  http://www.microsoft.com/whdc/DevTools/WDK/DTM.mspx

注意: 一部のサイトでは Windows Internet Explorer が必要です。

15.3.2 WinDriver ベースのドライバの WHQL 認定

WinDriver カーネル モジュール (**windrvr6.sys**) は、さまざまなハードウェア デバイスのドライバとして使用可能な一般的なドライバであるため、スタンドアロン ドライバとして署名を取得することはできません。ただし、WinDriver を使用して対象ハードウェア用の Windows 2000 / XP / Server 2003 / Vista ドライバを開発した場合、Microsoft WHQL 認定にハードウェアとドライバの両方を提出することができます。既に多くの WinDriver ユーザーがこの手続きを行い、WinDriver ベースのドライバのデジタル署名を取得しています。

15.3.3 DTM テストに関する注意点

WHQL ドキュメントで説明されているように、ドライバのテスト申請を行う前に Microsoft の Driver Test Manager (DTM) (http://www.microsoft.com/whdc/DevTools/WDK/DTM.mspx) をダウンロードして、ハードウェア / ソフトウェアに対し関連テストを実行する必要があります。DTM テストに Pass できることを確認したら、必要なログ パッケージを作成して、Microsoft のドキュメントの手順に従ってください。

DTM テストを実行する際には、次の点に注意してください。

- WinDriver ベースのドライバの DTM テスト クラスは、**Unclassified - Universal Device** です。
- DTM テストを実行する PC では、ドライバがインストールされていないデバイスなど、さまざまなエラー状態のデバイスがあってはなりません。この問題を回避するには、対象ハードウェアとドライバだけがインストールされているクリーンな Windows ロゴ マシンでテストを実行することを推奨します。別の方法として、テストを実行する前に、テストに関係のないハードウェアおよびドライバを取り外したり、無効にすることもできます。
- Driver Verifier テストは、テスト マシン上で検出されるすべての未署名ドライバに適用されます。このため、テスト PC にインストールされている未署名ドライバ (テスト対象の **windrvr6.sys** を除く) の数を最小限に抑えることが重要です。

- USB Selective Suspend テストでは、USB デバイス ツリーにおけるテスト対象 USB の階層は、少なくとも 1 つの外部ハブおよび 2 つ以下の外部ハブでなければなりません。
- ACPI Stress テストでは、BIOS の ACPI 設定で S3 状態のサポートを必須としています。
- PC の **boot.ini** ファイルのブート フラグに /PAE スイッチが追加されていることを確認してください。
- テスト申請には、***.pdb** デバッグ シンボル ファイルと **PREfast** ユーティリティの出力ファイル (**defects.xml**) が必要です。**windrvr6.sys** ドライバ用のこれらのファイルのコピーは、**WinDriver\redist** ディレクトリにあります。

第 16 章
WinDriver USB Device

この章では、Cypress EZ-USB FX2LP CY7C68013A、Microchip PIC18F4550、Philips PDIUSBD12 および Silicon Laboratories C8051F320 ハードウェアをベースとした USB デバイス ファームウェアを開発する WinDriver USB Device ツールキットについて説明します。

注意: WinDriver USB Device ツールキットは、Windows のみサポートしています (サポートするオペレーティング システムに関する詳細は、セクション 16.2 を参照してください)。

16.1 WinDriver USB Device の概要

WinDriver USB Device ツールキットは、Cypress EZ-USB FX2LP CY7C68013A、Microchip PIC18F4550、Philips PDIUSBD12 および Silicon Laboratories C8051F320 ハードウェアをベースとした USB デバイスのファームウェアの開発を簡素化、促進します。これらのハードウェアは、この章では "ターゲット ハードウェア" と呼ばれます。

このツールキットは、WinDriver USB ツールキットを補足するものです。これらのツールキットで、デバイスのファームウェアおよびホストドライバ開発の両方の段階において、USB デバイス開発ソフトウェアとして完全なソリューションを提供します。

USB デバイス ベンダーは、USB (Universal Serial Bus) の仕様 (第 3 章 を参照) をサポートする必要があります。USB インターフェイスを 2 つのレベルで実装します。USB プロトコルの低レイヤを SIE (Serial Interface Engine) で実装し、プロトコルの高レイヤをデバイスのファームウェアで実装します。

ファームウェアはソフトウェアとデータからなります。データは、デバイスの設定を定義し、PROM、EPROM、EEPROM およびフラッシュ チップなどのさまざまなプログラム可能な ROM チップを使用してメモリに半永久的にインストールされます。

WinDriver USB Device で、"ターゲット ハードウェア" の開発者は、直感的な GUI を使用して、対象のデバイス用に必要な USB インターフェイスを定義するファームウェアを簡単に作成できます。

WinDriver USB Device には、"ターゲット ハードウェア" 用にファームウェアのライブラリ[16.3.5]が含まれます。これらのライブラリには、一般的な USB ファームウェアの機能を実行する関数が含まれます。そのため、デバイス メーカーは、ファームウェアのコードを記述するのに多くの時間を費やすことなくデバイスをリリースできます。また、Microchip PIC18F4550 開発ボード用には、USB Mass Storage Class 規格 (http://www.usb.org/developers/devclass_docs/usb_msc_overview_1.2.pdf) および USB Mass Storage Bulk-Only Transport 規格 (http://www.usb.org/developers/devclass_docs/usbmassbulk_10.pdf) に準拠した Mass Storage API を提供する、特別な Mass Storage ファームウェア ライブラリも含まれます。

WinDriver USB Device は、WinDriver USB ドライバ開発キットのグラフィカルな DriverWizard ユーティリティの機能を持っていますが、異なる機能を持っており、対象のデバイスの USB のインターフェイス[16.4.1]を定義できます。たとえば、デバイス ID とデバイス クラス、インターフェイスの数、代替設定 (alternate settings) の数、エンドポイント (endpoints) の数および属性など。使いやすい GUI ダイアログを使用して、ウィザードのダ

イアログ[16.4.2]で定義した情報を基にデバイスのファームウェアのコードを生成します。DriverWizard で生成されたファームウェアのコードには、便利な API が含まれます。API は、WinDriver USB Device ファームウェア ライブラリ API を利用して、デバイスのファームウェアのフル機能を実装します。

Microchip PIC18F4550 開発ボード用に、WinDriver では Mass Storage USB デバイスを定義するための特別な GUI ダイアログを用意しています (セクション 16.4.1 のステップ 4 を参照してください)。このオプションを使用すると、生成されるファームウェア コードには、WinDriver USB Device Microchip PIC18F4550 Mass Storage ファームウェア ライブラリを利用して、Mass Storage デバイス ファームウェアを実装する便利な API が含まれます。

リファレンスで、WinDriver USB Device ファームウェア ライブラリおよび DriverWizard で生成された API の詳細を説明します。

さらに、ツールキットには、異なる目的のためにライブラリ API を利用する方法を示す、ターゲット ハードウェア用のファームウェア サンプルが含まれます [16.3]。

注意: ターゲット ハードウェア用に提供される API およびウィザード オプションは、第 9 章の USB 2.0 規格およびターゲット ハードウェアの規格に準拠しているため、これらの規格についての知識を必要としません。また、Microchip PIC18F4550 Mass Storage API およびウィザード オプションを利用することにより、USB Mass Storage Class 規格についての知識も必要としません。

ファームウェアのコードを生成した後は、開発プロセス を簡素化[16.4.3]する WinDriver USB Device の API を使用して、ファームウェアを実装するために必要な修正を行うことができます。ファームウェアの実装が終了したら、ファームウェアをビルド[16.4.3.2]してデバイスにダウンロード[16.4.3.3]します。

第 5 章 で説明した WinDriver USB ドライバ開発キット DriverWizard のハードウェア診断機能は、WinDriver USB Device の DriverWizard でも利用できます。つまり、一旦ファームウェアを開発してデバイスにダウンロードしたら、DriverWizard を使用して、ウィザードのグラフィカル インターフェイスから、デバイスの設定の確認とデバイスとの接続テストを行い、ハードウェアをデバッグできます[16.4.4]。

WinDriver USB ドライバ開発キットの登録版ユーザーの場合、デバイスのファームウェア開発とハードウェアのデバッグが終了したら、WinDriver USB ツールキットを使用して、対象のデバイスのドライバを開発できます[16.4.5]。

16.2 必要なシステムおよびハードウェア

WinDriver USB Device ツールキットを使用するのに必要なホスト システムおよびハードウェアは次のとおりです。

- オペレーティング システム: Windows 98 / Me / 2000 / XP / Server 2003。

  Cypress EZ-USB FX2LP CY7C68013A、Microchip PIC18F4550 および Silicon Laboratories C8051F320 ファームウェア コードをコンパイルしビルドするには、Windows 2000 / XP / Server 2003 が必要です。

  Philips PDIUSBD12 ファームウェア コードをコンパイルしビルドするには、Windows 98 / Me / 2000 / XP / Server 2003 または DOS が必要です。

  PDIUSBD12 評価版ファームウェアをデバイスにダウンロードするには、DOS が必要です。

- CPU アーキテクチャ: 32 ビット x86 または 64 ビット (x64: AMD64 またはインテル EM64T) プロセッサ
- サンプルおよび生成されたファームウェア コードをビルドするには、次の開発ツールを開発用 PC にインストールする必要があります。
  - Cypress EZ-USB FX2LP CY7C68013A ハードウェアの場合:
    Cypress EZ-USB FX2LP Development Kit
  - Microchip PIC18F4550 ハードウェアの場合:
    Microchip MCC18 Toolchain
  - Cypress EZ-USB FX2LP CY7C68013A および Silicon Laboratories C8051F320 ハードウェアの場合:
    Keil Cx51 Development Tools for 8x51 (バージョン 6.0 以降)
  - Philips PDIUSBD12 ハードウェアの場合:
    32 ビット DOS C コンパイラ
- サンプルおよび生成されたファームウェア コードは、次の開発環境 (オプション) をサポートしています。
  - Cypress EZ-USB FX2LP CY7C68013A および Silicon Laboratories C8051F320 ハードウェアの場合:
    Keil *u*Vision IDE (バージョン 2.0 以降)
  - Microchip PIC18F4550 ハードウェアの場合:
    Microchip MPLAB IDE (バージョン 7.20)
  - Silicon Laboratories C8051F320 ハードウェアの場合:
    Silicon Laboratories IDE (バージョン 1.9)
  - Philips PDIUSBD12 ハードウェアの場合:
    Borland C++ (バージョン 3.1)、32 ビット DOS コンパイラ ("Turbo C")

16.3 WinDriver デバイス ファームウェア (WDF) ディレクトリの概要

このセクションでは、**`WinDriver\wdf`** ディレクトリのディレクトリ構造とファイルについて説明します。

`wdf\` ディレクトリには、次のディレクトリが含まれます。

- **`cypress\`** ディレクトリ: Cypress EZ-USB FX2LP CY7C68013A 開発ボードをベースとしたデバイス用のファイルが含まれます。
- **`microchip\`** ディレクトリ: PIC18F4550 開発ボードをベースとしたデバイス用のファイルが含まれます。
- **`philips\`** ディレクトリ: Philips PDIUSBD12 ハードウェアをベースとしたデバイス用のファイルが含まれます。

- **silabs** ディレクトリ: Silicon Laboratories C8051F320 開発ボードをベースとしたデバイス用のファイルが含まれます。

16.3.1 cypress ディレクトリ

WinDriver\wdf\cypress ディレクトリには、次のディレクトリが含まれます。

- **FX2LP** ディレクトリ: FX2LP CY7C68013A 開発ボードをベースとしたデバイス用のファイルが含まれます (以後、このセクションでは "FX2LP ボード" と呼ぶ)。

FX2LP ディレクトリには、次のサブディレクトリおよびファイルが含まれます。

- **include** ディレクトリ:
  - **wdf_cypress_lib.h**: FX2LP開発ボードをベースとしたデバイス用の、ファームウェア ライブラリの種類、一般的な定義および関数プロトタイプを含むヘッダー ファイル。このファイルは、ボードのファームウェア ライブラリ (評価版ユーザーの場合、**wdf_cypress_fx2lp_eval.lib**。登録版ユーザーの場合、ライブラリのソース コードは DriverWizard デバイスのファームウェア コード生成の一部として作成されます。セクション [16.3.3] の WinDriver USB Device ファームウェア ライブラリに関する説明を参照) のインターフェイスを提供します。
  - **wdf_cypress.h:** Cypress FX2LP API を利用するのに必要なファームウェア ライブラリ定義および `#include` 文を含むヘッダー ファイル。
  - **periph.h**: FX2LP 開発ボードをベースとしたデバイス用の、USB 周辺機器機能をサポートする関数プロトタイプを含むヘッダー ファイル。

    関数の実装は、デバイス用に定義した特定の設定に従います。デバイス用の実装を含む **periph.c** ソース ファイルは、ウィザードで定義した USB デバイスの設定を基にデバイスのファームウェア コードを生成するときに、DriverWizard によって作成されます。詳細は、DriverWizard で生成されるファイル [16.4.3.1] の説明を参照してください。
- **lib** ディレクトリ:
  - **wdf_cypress_fx2lp_eval.lib**: FX2LP 開発ボード用の評価用ファームウェア ライブラリ ([16.3.5] の説明を参照)
- **samples** ディレクトリ: FX2LP 開発ボード用のデバイス ファームウェアのサンプル
  - **loopback** ディレクトリ: ループバック サンプル。サンプルは、IN エンドポイントの FIFO バッファから読み込んだデータを OUT エンドポイントの FIFO バッファに格納する、ループバックを実装します。
    - **periph.c**: **periph.h** ヘッダー ファイル (上記を参照) で宣言した関数のサンプル実装を含むソース ファイル
    - **wdf_dscr.a51**: FX2LP 開発ボード用のサンプル記述子データ テーブル定義を含むアセンブリ ファイル
    - **build.bat**: サンプル ファームウェア コードのビルド用のユーティリティ。

注意: ビルド ユーティリティは、評価版のファームウェア ライブラリ (**wdf_cypress_fx2lp_eval.lib**) を使用します。

– **loopback_eval.hex**: サンプル コードと **build.bat** ユーティリティで作成した FX2LP 開発ボード用のサンプル ループバック ファームウェア。

注意: ファームウェアは、評価版のファームウェア ライブラリ (**wdf_cypress_fx2lp_eval.lib**) を使用します。

16.3.2 microchip ディレクトリ

WinDriver\wdf\microchip ディレクトリには、次のディレクトリが含まれます。

- **18F4550** ディレクトリ: PIC18F4550 開発ボードをベースとしたデバイス用のファイルが含まれます。

18F4550 ディレクトリには、次のサブディレクトリおよびファイルが含まれます。

- **include** ディレクトリ:
  - **class\msd** ディレクトリ: PIC18F4550 ボード用の USB Mass Storage Device Class ファームウェア API の宣言および型の定義
    – **wdf_msd.h**: PIC18F4550 ボード用の Mass Storage ファームウェア ライブラリの種類、一般的な定義、および関数プロトタイプを含むヘッダー ファイル
    – **wdf_disk.h**: PIC18F4550 ボードをベースとした Mass Storage Device のストレージ メディアにアクセスするための種類、一般的な定義、および関数プロトタイプを含むヘッダー ファイル

      注意: このヘッダーで宣言される関数の実装は、ハードウェアに依存します。生成される DriverWizard **wdf_xxx_hw.c** ファイルには、関数の実装 Stub が含まれています。**WinDriver\wdf\microchip18F4550\samples\msd\sdcard.c** ファイルには、SD Card のストレージ メディアにアクセスする関数の実装サンプルが含まれています。
    – **wdf_microchip_lib.h**: PIC18F4550 ボードをベースとしたデバイスの一般的なファームウェア ライブラリの種類、定義、関数プロトタイプを含むヘッダー ファイル。このファイルは、**wdf_usb9.h**、**wdf_msd.h** (Mass Storage ファームウェアを開発する場合) と共に、ボードのファームウェア ライブラリ (評価版ユーザーの場合は、**wdf_microchip_18f4550_eval.lib** / **wdf_microchip_msd_18f4550_eval.lib** (Mass Storage)。登録版ユーザーの場合、ライブラリのソース コードは DriverWizard デバイスのファームウェア コード生成の一部として作成されます。WinDriver USB Device ファームウェア ライブラリについては、セクション 16.3.5 を参照) のインターフェイスを提供します。
    – **wdf_usb9.h**: 第 9 章 の USB 2.0 規格に準拠した、ファームウェア ライブラリ USB 記述子の型の定義および関数の宣言を含むヘッダー ファイル
    – **wdf_microchip.h**: PIC18F4550 ボード用の一般的なファームウェア ライブラリの定義を含むヘッダー ファイル。このヘッダー ファイルには、PIC18F4550 ボードで必要な他

のすべてのヘッダー ファイルが含まれています。このため、このボードのファームウェア開発を行う場合は、ソース ファイルからこのヘッダーのみ含める必要があります。

- **types.h**: PIC18F4550 ボード用のデータ型を定義するヘッダー ファイル
- **periph.h**: PIC18F4550 ボードをベースとしたデバイス用の、USB 周辺機器機能をサポートする関数プロトタイプを含むヘッダー ファイル。関数の実装は、デバイス用に定義した特定の設定に従います。デバイス用の実装を含む **periph.c** ソース ファイルは、ウィザードで定義した USB デバイスの設定を基にデバイスのファームウェア コードを生成するときに、DriverWizard によって作成されます。詳細は、DriverWizard で生成されるファイル [16.4.3.1] の説明を参照してください。

- **lib** ディレクトリ:
  - **wdf_microchip_18f4550_eval.lib**: PIC18F4550 ボード用の評価版ファームウェア ライブラリ
  - **wdf_microchip_msd_18f4550_eval.lib**: PIC18F4550 ボード用の評価版 Mass Storage ファームウェア ライブラリ

    ファームウェア ライブラリに関する詳細は、セクション 16.3.5 を参照してください。

- **samples** ディレクトリ: PIC18F4550 ボード用のデバイス ファームウェア サンプル
  - **loopback** ディレクトリ: ループバック サンプル。サンプルは、IN エンドポイントの FIFO バッファから読み込んだデータを OUT エンドポイントの FIFO バッファに格納する、ループバックを実装します。
    - **periph.c**: **periph.h** ヘッダー ファイル (上記を参照) で宣言した関数のサンプル実装を含む C ソース ファイル
    - **wdf_dscr.h**: PIC18F4550 ボード用のサンプル デバイス記述子情報を含むヘッダー ファイル
    - **wdf_dscr.c**: PIC18F4550 ボード用のデバイス記述子データ構造体の定義を含むソース ファイル
    - **build.bat**: サンプル ファームウェア コードのビルド用のユーティリティ

      注意: ビルド ユーティリティは、評価版のファームウェア ライブラリ (**wdf_microchip_18f4550_eval.lib**) を使用します。
    - **loopback_eval.hex**: サンプル コードと **build.bat** ユーティリティで作成した PIC18F4550 ボード用のサンプル ループバック ファームウェア

      注意: ファームウェアは、評価版のファームウェア ライブラリ (**wdf_microchip_18f4550_eval.lib**) を使用します。
    - **loopback_eval.lkr**: ループバック サンプル用のリンカ ファイル
  - **msd** ディレクトリ: Mass Storage デバイス サンプル。サンプルは、PIC18F4550 Mass Storage 評価版ライブラリ(wdf_microchip_msd_18f4550_eval.lib) を使用して、Secure Digital Card (SD カード) 付き Microchip PICTail ドーター ボード用の Mass Storage デバイスを実装します。

サンプルでは、次の SD カードをサポートしています: EP Memory 512MB。Lexar 256MB、512MB および 1GB。SunDisk 128MB、512MB および 2GB。SimpleTech 256MB および 1GB。Viking 512MB および 256MB。ATP 1GB。

- **periph.c**: **periph.h** ヘッダー ファイル (上記を参照) で宣言した関数のサンプル実装を含む C ソース ファイル
- **wdf_dscr.h**: PIC18F4550 ボード用のサンプル デバイス記述子情報を含むヘッダー ファイル
- **wdf_dscr.c**: PIC18F4550 ボード用のデバイス記述子データ構造体の定義を含むソース ファイル
- **sdcard.h**: サポートする SD カード (上記を参照) 用の種類および一般的な定義を含むヘッダー ファイル
- **sdcard.c**: サポートする SD カード用に、wdf_disk.h ヘッダー ファイル [16.3.2] で宣言された Microchip PIC18F4550 Mass Storage ライブラリのストレージ メディア アクセス関数を実装するソース ファイル
- **build.bat**: サンプル ファームウェア コードのビルド用のユーティリティ

  注意: ビルド ユーティリティは、Mass Storage 評価版のファームウェア ライブラリ (**wdf_microchip_msd_18f4550_eval.lib**) を使用します。
- **msd_eval.hex**: サンプル コードと **build.bat** ユーティリティで作成した PIC18F4550 ボード用のサンプル Mass Storage ファームウェア

  注意: ファームウェアは、Mass Storage 評価版のファームウェア ライブラリ (**wdf_microchip_18f4550_eval.lib**) を使用します。
- **msd_eval.lkr**: Mass Storage サンプル用のリンカ ファイル

16.3.3 philips ディレクトリ

WinDriver\wdf\philips ディレクトリには、次のディレクトリが含まれます。

- **d12** ディレクトリ: PDIUSBD12 をベースとしたデバイス用のファイルが含まれます

d12 ディレクトリには、次のサブディレクトリおよびファイルが含まれます。

- **include** ディレクトリ:
  - **d12_lib .h**: PDIUSBD12 をベースとしたデバイス用のファームウェア ライブラリの種類、一般的な定義および関数プロトタイプを含むヘッダー ファイル。このファイルは、PDIUSBD12 のファームウェア ライブラリ (評価版ユーザーの場合は、**d12_eval.lib**。登録版ユーザーの場合、ライブラリのソース コードは DriverWizard デバイスのファームウェア コード生成の一部として作成されます。WinDriver USB Device ファームウェア ライブラリについては、セクション 16.3.5 を参照) のインターフェイスを提供します。
  - **types.h**: PDIUSBD12 用のデータ型を定義するヘッダー ファイル

- **d12_io.h**: ハードウェアに依存する、一般的なファームウェア ライブラリの定義および関数の宣言を含むヘッダー ファイル。このヘッダーは、ライブラリのハードウェア抽象レイヤー のインターフェイスを提供します。

  このファイルのデフォルトの実装は、ISA カードを使用して PDIUSBD12 ベースのボードを x86 PC へ接続する D12-ISA (PC) Eval Kit (バージョン 1.4) をターゲットにしています。WinDriver USB Device の登録版のユーザーは、他のマイクロコントローラ (セクション 16.4.3.2 の注意を参照) をサポートするために、このファイルと **d12_io.c** [16.4.3.1] にあるライブラリのハードウェア固有 API 実装を編集することができます。

- **periph.h**: PDIUSBD12 をベースとしたデバイス用の、USB 周辺機器機能をサポートする関数プロトタイプを含むヘッダー ファイル。関数の実装は、デバイス用に定義した特定の設定に従います。デバイス用の実装を含む **periph.c** ソース ファイルは、ウィザードで定義した USB デバイスの設定を基にデバイスのファームウェア コードを生成するときに、DriverWizard によって作成されます。詳細は、DriverWizard で生成されるファイル [16.4.3.1] の説明を参照してください。

- **lib** ディレクトリ:
  - **d12_eval.lib**: PDIUSBD12 用の評価版ファームウェア ライブラリ [16.3.5]

- **Samples** ディレクトリ: PDIUSBD12 用のデバイス ファームウェア サンプル:
  - **loopback** ディレクトリ: ループバック サンプル。サンプルは、IN エンドポイントの FIFO バッファから読み込んだデータを OUT エンドポイントの FIFO バッファに格納する、ループバックを実装します。
  - **periph.c**: **periph.h** ヘッダー ファイル (上記を参照) で宣言した関数のサンプル実装を含む C ソース ファイル
  - **wdf_dscr.c**: PDIUSBD12 用のデバイス記述子データ構造体の定義を含むソース ファイル
  - **build.bat**: サンプル ファームウェア コードのビルド用のユーティリティ

    注意: ビルド ユーティリティは、評価版のファームウェア ライブラリ (**d12_eval.lib**) を使用します。
  - **LOOPBACK.EXE**: サンプル コードと **build.bat** ユーティリティで作成した PDIUSBD12 用のサンプル ループバック ファームウェア

    注意: ファームウェアは、評価版のファームウェア ライブラリ (**d12_eval.lib**) を使用します。
  - **lb_eval.mak**: Turbo C コンパイラで、ループバック サンプルをビルドするための makefile
  - **dma** ディレクトリ: DMA サンプル。サンプルは、ベンダー要求を受信したら DMA 転送を開始し、PDIUSBD12 の DMA 能力を示します。DMA は、x86 オンボード DMA コントローラで実行されます。別のコントローラを使用するには、コードのハードウェア依存部分を、使用するコントローラ用のハードウェア固有コードに変更する必要があります。ファームウェア ライブラリのハードウェア固有部分を変更するには、WinDriver USB Device ツールキットの登録版ユーザーでなければならない点に注意してください (詳細は、セクション 16.3.5 を参照)。

- **periph.c**: **periph.h** ヘッダー ファイル (上記を参照) で宣言した関数のサンプル実装を含む C ソース ファイル
- **wdf_dscr.c**: PDIUSBD12 用のデバイス記述子データ構造体の定義を含むソース ファイル
- **build.bat**: サンプル ファームウェア コードのビルド用のユーティリティ

  注意: ビルド ユーティリティは、評価版のファームウェア ライブラリ (**d12_eval.lib**) を使用します。
- **D12DMA.EXE**: サンプル コードと **build.bat** ユーティリティで作成した PDIUSBD12 用のサンプル DMA ファームウェア ダウンロード プログラム

  注意: ファームウェアは、評価版のファームウェア ライブラリ (**d12_eval.lib**) を使用します。
- **dma_eval.mak**: Turbo C コンパイラで、DMA サンプルをビルドするための makefile

16.3.4 silabs ディレクトリ

WinDriver\wdf\silabs ディレクトリには、次のディレクトリが含まれます。

- **F320** ディレクトリ: C8051F320 開発ボードをベースとしたデバイス用のファイルが含まれます。

F320 ディレクトリには、次のサブディレクトリおよびファイルが含まれます。

- **include** ディレクトリ:
  - **wdf_silabs_lib.h**: C8051F320 ボードをベースとしたデバイス用の、ファームウェア ライブラリの種類、関数プロトタイプを含むヘッダー ファイル。このファイルは、ボードのファームウェア ライブラリ (評価版ユーザーの場合、**wdf_silabs_f320_eval.lib**。登録版ユーザーの場合、ライブラリのソース コードは DriverWizard のデバイス ファームウェア コード生成の一部として作成されます。セクション [16.3.5] の WinDriver USB Device ファームウェア ライブラリに関する説明を参照) のインターフェイスを提供します。
  - **c8051f320.h**: C8051F320 ボードの一般的なファームウェア ライブラリ定義を含むヘッダー ファイル
  - **c8051f320regs.h**: C8051F320 ボードのレジスタ/ビット定義を含むヘッダー ファイル
  - **periph.h**: C8051F320 ボードをベースとしたデバイス用の、USB 周辺機器機能をサポートする関数プロトタイプを含むヘッダー ファイル。

    関数の実装は、デバイス用に定義した特定の設定に従います。デバイス用の実装を含む **periph.c** ソース ファイルは、ウィザードで定義した USB デバイスの設定を基にデバイスのファームウェア コードを生成するときに、DriverWizard によって作成されます。詳細は、DriverWizard で生成されるファイル [16.4.3.1] の説明を参照してください。

- **lib** ディレクトリ:
  - **wdf_silabs_f320_eval.lib**: C8051F320 ボード用の評価用ファームウェア ライブラリ ([16.3.5] の説明を参照)。
- **samples** ディレクトリ: C8051F320 ボード用のデバイス ファームウェアのサンプル
  - **loopback** ディレクトリ: ループバック サンプル: サンプルは、IN エンドポイントの FIFO バッファから読み込んだデータを OUT エンドポイントの FIFO バッファに格納する、ループバックを実装します。
    - **periph.c**: **periph.h** ヘッダー ファイル (上記を参照) で宣言した関数のサンプル実装を含むソース ファイル
    - **wdf_dscr.h**: C8051F320 ボード用のサンプル デバイス記述子情報を含むヘッダー ファイル
    - **wdf_dscr.c**: C8051F320 ボード用のデバイス記述子データ構造体の定義を含むソース ファイル
    - **build.bat**: サンプル ファームウェア コードのビルド用のユーティリティ

      注意: ビルド ユーティリティは、評価版のファームウェア ライブラリ (**wdf_silabs_f320_eval.lib**) を使用します。
    - **loopback_eval.hex**: サンプル コードと **build.bat** ユーティリティで作成した C8051F320 開発ボード用のサンプル ループバック ファームウェア。

      注意: ファームウェアは、評価版のファームウェア ライブラリ (**wdf_silabs_f320_eval.lib**) を使用します。

16.3.5 WinDriver USB Device ファームウェア ライブラリ

WinDriver USB Device ツールキットの登録版を使用して DriverWizard でファームウェアのコードを生成すると、生成したコードに、一般的な USB ファームウェアの機能を実行する API を含む、WinDriver USB Device ファームウェア ライブラリのソース ファイルが含まれます。

登録版のツールキットを使用して、PIC18F4550 ボード用の Mass Storage ファームウェア コードを生成する場合、生成されるコードには、USB Mass Storage Class 規格に準拠した Mass Storage USB デバイス開発用 API を含む **wdf_microchip_msd_18f4550_eval.lib** Mass Storage ファームウェア ライブラリのソース コードも含まれます。

DriverWizard で生成されるファイルに関する詳細は、セクション 16.4.3.1 を参照してください。

ライブラリ ソース ファイルは、ツールキットの 評価版には含まれません。ツールキットの評価版には、WinDriver USB Device の評価ができるように、コンパイル済みの評価版ライブラリが含まれています。ライブラリは、デバイス ファームウェアのサンプルおよび DriverWizard で生成された評価用ファームウェア コードで利用されます。

評価版ライブラリは、次の制限を除いて、登録版ライブラリと機能は同じです: あらかじめ設定されている数 (標準の評価版ライブラリでは 25,000、PIC18F4550 Mass Storage 評価版ライブラリでは 1,000,000) の転送を実行できます。この数を超えると、ライブラリは動作しなくなります。

WinDriver USB Device ライセンスを取得後に、評価期間中に作成したファームウェアを登録する方法については、セクション 16.4.3.2 の注意を参照してください。

16.3.6 サンプル コードのビルド

WinDriver\wdf\<vendor>\<hardware>\samples\xxx 以下のサンプル (xxx はサンプル名。例: Cypress ループバック サンプルの場合、**WinDriver\wdf\cypress\FX2LP\samples\loopback**) をビルドするには、対象サンプルの **build.bat** ユーティリティ (例: **FX2LP\samples\loopback\build.bat**) を使用します。

ユーティリティを実行する前に、**build.bat** ファイルの情報がシステム構成と一致していることを確認してください。

- Cypress EZ-USB FX2LP CY7C68013A ボードおよび Silicon Laboratories C8051F320 ボードの場合: **build.bat** ファイル内の KEIL 変数に Keil Development Tools のディレクトリが設定されていることを確認します。**build.bat** ファイルで使用されるデフォルトの Keil ディレクトリは、**C:\Keil** です。Keil を別の場所にインストールした場合は、**build.bat** ファイル内の次の行を変更し、正しい場所を指定します。
  set KEIL=C:\Keil

  たとえば、Keil を **D:\MyTools\Keil** にインストールした場合は、次のように変更します。
  set KEIL=D:\MyTools\Keil

- ファイル内の CYPRESS 変数に Cypress EZ-USB Development Kit のディレクトリが設定されていることを確認します。**build.bat** ファイルで使用されるデフォルトのディレクトリは、**C:\Cypress** です。Cypress EZ-USB Development Kit を別の場所にインストールした場合は、**build.bat** ファイル内の次の行を変更し、正しい場所を指定します。
  set CYPRESS=C:\Cypress

  たとえば、Cypress EZ-USB Development Kit を **D:\Cypress** にインストールした場合は、次のように変更します。
  set CYPRESS=D:\Cypress

- Microchip PIC18F4550 サンプルの場合: ファイル内の MCC 変数に MCC18 のディレクトリが設定されていることを確認します。**build.bat** ファイルで使用されるデフォルトのディレクトリは、**C:\mcc18** です。MCC18 Toolchain を別の場所にインストールした場合は、**build.bat** ファイル内の次の行を変更し、正しい場所を指定します。
  set MCC=C:\mcc18

  たとえば、MCC18 Toolchain を **D:\microchip\mcc18** にインストールした場合は、次のように変更します。
  set MCC=D:\microchip\mcc18

- Philips PDIUSBD12 サンプルの場合: ファイル内の MCC 変数に Turbo C のディレクトリが設定されていることを確認します。**build.bat** ファイルで使用されるデフォルトのディレクトリは、**C:\borlandc** です。Turbo C コンパイラを別の場所にインストールした場合は、**build.bat** ファイル内の次の行を変更し、正しい場所を指定します。
  set TURBOC=c:\borlandc

  たとえば、Turbo C コンパイラを **D:\TurboC** にインストールした場合は、次のように変更します。
  set TURBOC=d:\TurboC

- **build.bat** ユーティリティを実行して、サンプル ファームウェアをビルドします。

16.4 WinDriver USB Device の開発プロセス

以下の手順で、WinDriver USB Device を使用して (ターゲット ハードウェアをベースとした) USB デバイス用のファームウェアを開発します。

16.4.1 Device USB インターフェイスの定義

WinDriver USB Device の DriverWizard ユーティリティを使用して、デバイスの USB インターフェイスを定義します。

1. 次のいずれかの方法を使用して、DriverWizard を実行します。
   - [スタート] - [プログラム] - [WinDriver] - [DriverWizard] の順に選択します。
   - デスクトップの DriverWizard アイコンをダブルクリックします。
   - **WinDriver\wizard\wdwizard.exe** をダブルクリックするか、もしくはコマンドライン プロンプトで入力して実行します。
2. ウィザードの [Choose Your Project] ダイアログで [New device firmware project] オプションを選択して、[Next >>] をクリックします。

   DriverWizard の [File] メニューから、もしくはウィザードのツールバーでファームウェア プロジェクト アイコンをクリックして、新しいデバイス ファームウェア プロジェクトを作成することもできます。

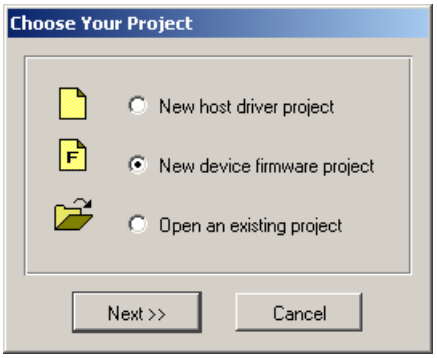


図 16.1: デバイス ファームウェア プロジェクトの作成

3. [Choose Your Development Board] ダイアログで対象のハードウェアを選択して、[OK] をクリックします。

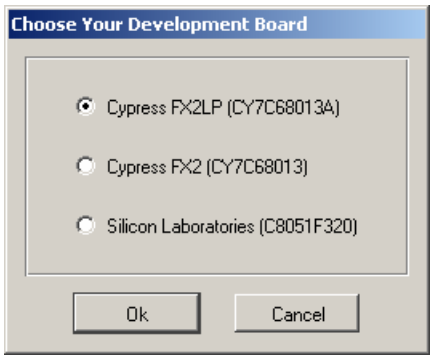


図 16.2: 開発ボードの選択

4. Microchip PIC18F4550 ボードを選択した場合は、[Choose Your Device Function] ダイアログボックスでデバイス関数を指定します。

図 16.3: Microchip – デバイス関数の選択

5. [Edit Device Descriptor] ダイアログで、ベンダーとデバイス ID、メーカーとデバイスの説明、デバイス クラスとサブクラスなどの、対象デバイスの基本的なデバイス記述子情報を定義します。

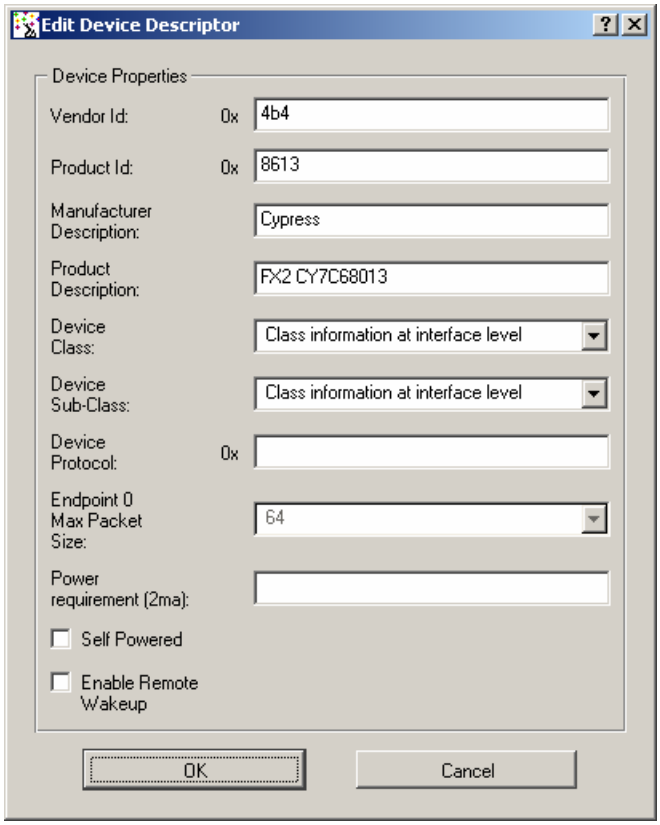


図 16.4: デバイス記述子の編集

6. [Configure Your Device] ダイアログで、デバイスの USB 設定を定義します。

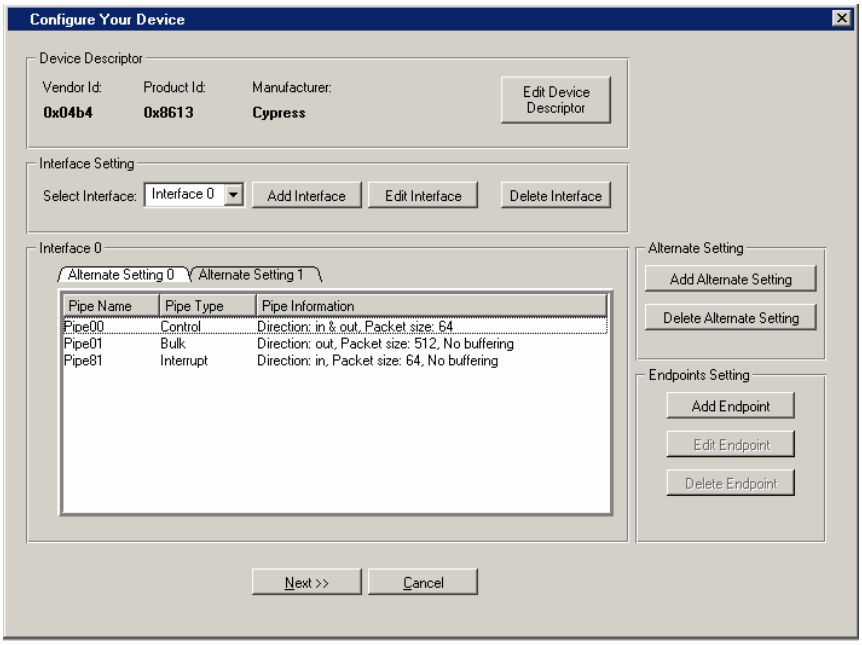


図 16.5: デバイスの設定

このダイアログでは、デバイス インターフェイスの追加、各インターフェイスの代替設定の追加、および各代替設定で必要なエンドポイントの追加を行うことができます。

注意:

① 複数のインターフェイスを定義する場合、DriverWizard はコンポジット デバイス用のファームウェア コードを生成します。ウィザードは、2 つ目のインターフェイスを追加する際に警告を表示します。

② Silicon Laboratories C8051F320 ハードウェアおよび Philips PDIUSBD12 ハードウェアでは、マルチインターフェイスはサポートされていません。

③ Microchip PIC18F4550 ボード用の Mass Storage ファームウェアを生成する場合、ファームウェアは特定のインターフェイスおよび代替設定用に定義されているため、インターフェイスまたは代替設定の追加はできません。

コンポーネントを追加する際に、インターフェイスのクラスとサブクラスやエンドポイントのアドレス、転送タイプ、最大パケット サイズなどの、各コンポーネントに関連する属性を定義できます。ウィザードは、設定が簡単に行えるように、デバイス用の適切な設定オプションのみを表示します。また、設定に潜在的な問題がある場合は警告を表示します。

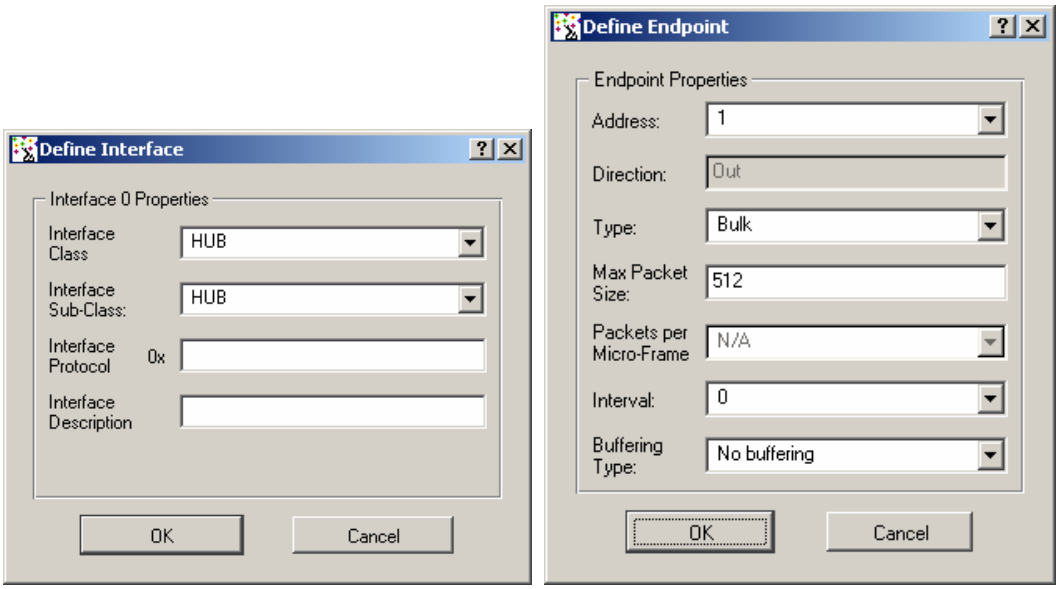


図 16.6: インターフェイスおよびエンドポイントの定義

Philips PDIUSBD12 ファームウェアを生成する場合、(該当する場合) メイン等時性エンドポイントを選択すると、ウィザードよってメイン エンドポイントが定義されます。

両方のメイン エンドポイントを等時性として定義する `ISO_IN_OUT` モードを選択した場合に、ウィザードによって定義されるメイン エンドポイントを図 16.7 に示します。

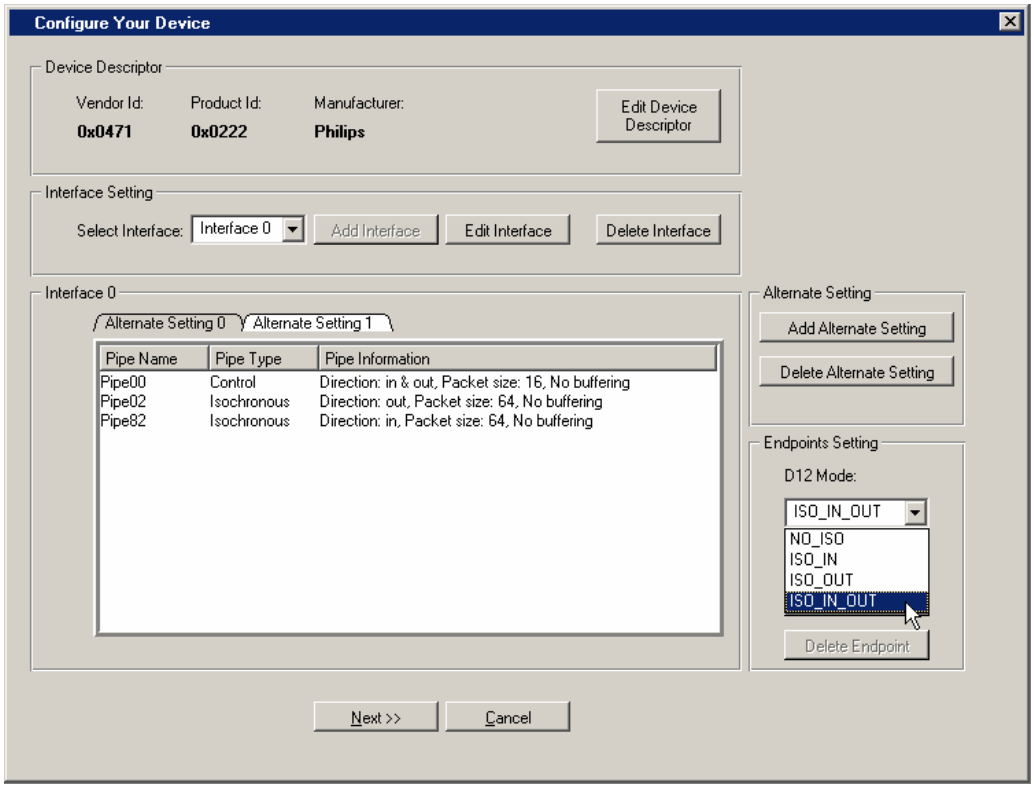


図 16.7: Philips PDIUSBD12 – メイン エンド ポイント パイプの定義

Cypress EZ-USB FX2LP CY7C68013A 開発ボードでエンドポイントを設定する方法の詳細は、この後にあります。

Microchip PIC18F4550 ボードをベースとした Mass Storage デバイスの設定を定義する場合は、[Edit Inquiry Descriptor] オプションを選択して、ホストからの SCSI 照会要求に対するデバイスの応答方法を定義することができます。

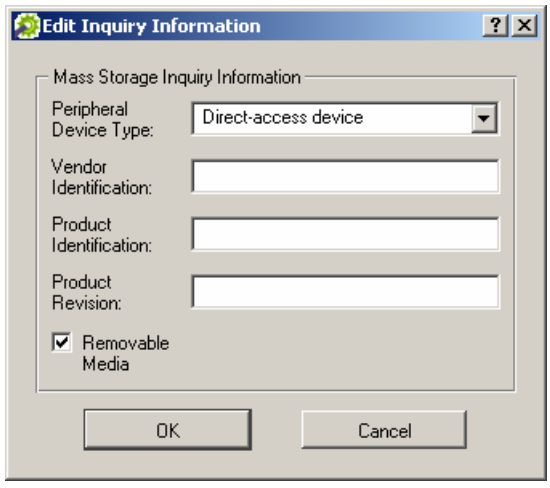


図 16.8: Microchip PIC18F4550 Mass Storage – 照会情報の編集

いつでも、デバイスの設定ダイアログ ボックスにから追加した設定情報を削除したり、設定情報を編集できます。

7. [File] メニューから、もしくはウィザードのツールバー アイコンを使用して、DriverWizard デバイス ファームウェア プロジェクトをいつでも保存できます。一旦プロジェクトを**xxx.wdp** 形式で保存したら、後から DriverWizard で開いて、終了した時点から作業を再開できます。

デバイスの USB インターフェイスの定義が終了したら、DriverWizard の定義 (次のセクション [16.4.2] を参照) に基づいて、デバイスのファームウェア コードを生成します。

16.4.1.1 EZ-USB エンドポイント バッファの設定

このセクションには、EZ-USB エンドポイント バッファの設定に関する EZ-USB テクニカル リファレンス マニュアル (**EZ-USB_TRM.pdf**) のセクション 1.18 の情報が含まれています。この情報は、DriverWizard を使用して Cypress EZ-USB FX2LP CY7C68013A 開発ボードをベースとしたデバイスのエンドポイント設定を定義するときに役立ちます。

詳細は、Cypress\USB\Doc\FX2LP\ ディレクトリにある EZ-USB テクニカル リファレンス マニュアルを参照してください。次の URL からオンラインでも入手できます。
http://www.keil.com/dd/docs/datashts/cypress/fx2_trm.pdf

USB 2.0 仕様では、エンドポイントをデータのソースまたはシンクとして定義しています。USB はシリアル バスなので、実際のデバイスのエンドポイントは、連続した空のまたは USB データ バイトが格納された FIFO です。ホストは 4 ビットのアドレスおよび方向ビットを送信してデバイスのエンドポイントを選択します。したがって、USB は、IN0 から IN15 および OUT0 から OUT15 の 32 のエンドポイントをユニークにアドレスできます。

EZ-USB から見ると、エンドポイントは受信したバイト、またはバス上を送信するバイトが含まれているバッファです。OUT エンドポイント バッファからデータを読み込み、IN エンドポイント バッファにホストに送信するデータを書き込みます。

EZ-USB には、図 16.9 のように 3 つの 64 バイト エンドポイント バッファと 4KB のバッファ空間が含まれており、12 通りに設定できます。3 つの 64 バイト バッファは、すべての設定で共通です。

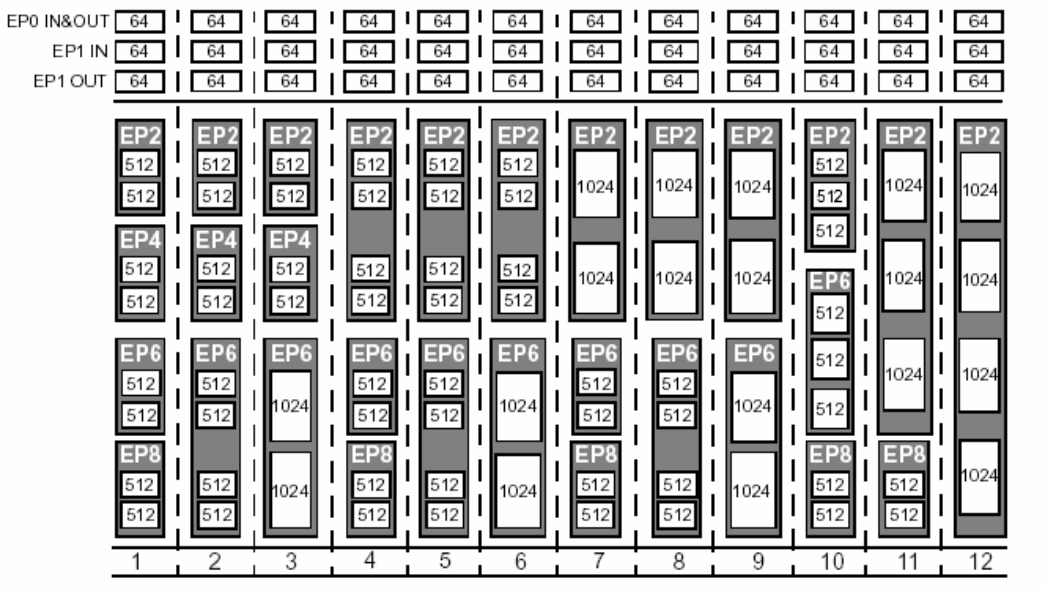


図 16.9: EZ-USB エンドポイント バッファ

16.4.2 デバイス ファームウェア コードの生成

[Configure Your Device] ダイアログ ボックスから、次のいずれかの方法でデバイス ファームウェア コードを生成します。

- [Next >>] ボタンをクリックするか、Alt+N ショートカット キーを使用する
- [Generate Code] ツールバー アイコンを選択する
- [Build] メニューから [Generate Code] オプションを選択する

ウィザードの [Select Code Generation Options] ダイアログが表示されます。

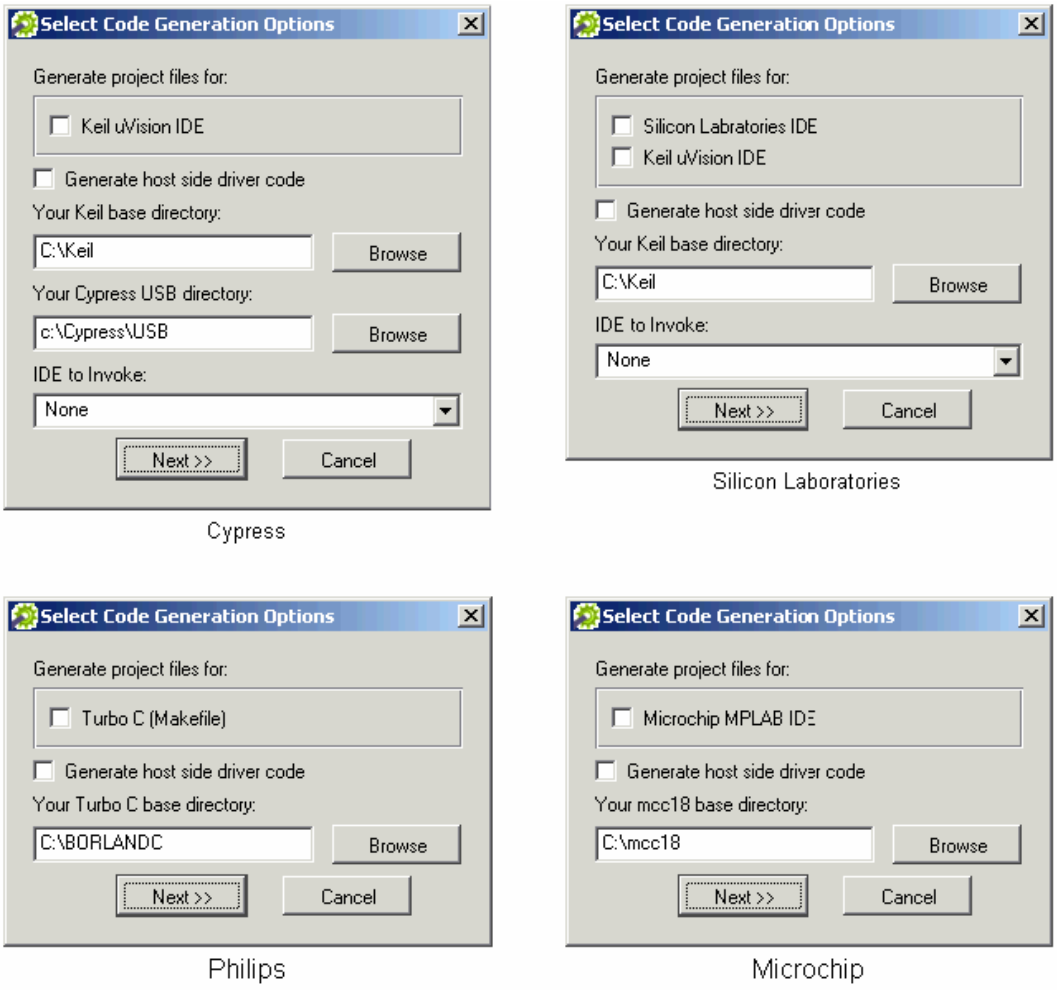


図 16.10: ファームウェア コードの生成

ダイアログのすべてのディレクトリ パスが PC 上の正しい場所を指していることを確認します。

- Cypress EZ-USB FX2LP CY7C68013A ハードウェアおよび Silicon Laboratories C8051F320 ハードウェアの場合: [Your Keil base directory] に Keil Cx51 Development Tools for 8x51 のインストール ディレクトリを指定します。
- Cypress EZ-USB FX2LP CY7C68013A ハードウェアの場合: [Your Cypress USB directory] に Cypress EZ-USB FX2LP Development Kit の Cypress\USB ディレクトリを指定します。

- Microchip PIC18F4550 ハードウェアの場合: [Your mcc18 base directory] に MCC18 Toolchain のインストール ディレクトリを指定します。
- Philips PDIUSBD12 ハードウェアの場合: [Your Turbo C base directory] に Turbo C コンパイラのインストール ディレクトリを指定します。

[Select Code Generation Options] ダイアログ ボックスの対応するチェック ボックスをオンにして、ターゲット ハードウェア用にサポートされている開発環境用の特定の project/make ファイルを生成することができます。

Keil *u*Vision IDE または Silicon Laboratories IDE 用のプロジェクト ファイルを生成する場合、ウィザードは選択された IDE が起動されるように自動的に [IDE to Invoke] を変更します。[IDE to Invoke] を変更しない場合、ウィザードはコード生成後にこの IDE を起動します。

上のダイアログで表示されている USB ホストドライバ コードの生成オプションは、WinDriver USB ツールキットの製品版を利用している場合、または WinDriver の評価期間内にのみサポートされます。このオプションが選択されていると、ウィザードは、デバイスのファームウェア コード以外に (ウィザードで定義した) USB デバイス用の WinDriver USB ホストドライバ アプリケーションのスケルトンも生成します。DriverWizard デバイスドライバ コードの生成に関する詳細は、第 5 章 およびセクション 16.4.5 を参照してください。

16.4.3 デバイス ファームウェアの開発

ウィザードでファームウェア コードを生成した後、必要に応じて、コードを修正できます。開発が効率的に行えるように、ライブラリおよび生成した WinDriver USB Device ファームウェア API を使用して、必要なファームウェアの機能を実装します。

USB ファームウェアの API および生成されるコードは、リファレンスで説明します。

注意: WinDriver ライブラリおよび生成したデバイスのファームウェア コードを修正する場合、コードがターゲット ハードウェアの仕様を満たしていることを確認してください。

- FX2LP CY7C68013A ボードの場合: **`EZ-USB_TRM.pdf`** - 特に、セクション 15.6「Endpoint Configuration」を参照してください。このドキュメントは、**`Cypress\USB\Doc\FX2LP\`** ディレクトリにあります。次の URL からオンラインでも入手できます。
  http://www.keil.com/dd/docs/datashts/cypress/fx2_trm.pdf
- PIC18F4550 ボードの場合: **`39632b.pdf`** (特に、セクション 17.3「USB RAM」および 17.4「Buffer Descriptors and the Buffer Descriptors Table」を参照してください。このドキュメントは、次の Web サイトから入手できます。
  http://ww1.microchip.com/downloads/en/DeviceDoc/39632b.pdf
- PDIUSBD12 の場合: **`PDIUSBD12-08.pdf`**。次の Web サイトから入手できます。
  http://www.semiconductors.philips.com/pip/PDIUSBD12D.html#datasheet
- Silicon Laboratories C8051F320 ボードの場合: **`C8051F32xRev1_1.pdf`** (特に、セクション 15.5「FIFO Management」および 15.11「Configuring Endpoints 1-3」を参照してください。このドキュメントは、**`Silabs\MCU\Documentation\Datasheets\`** ディレクトリにあります (Silicon Laboratories IDE をインストールした場合)。また、次の Web サイトからも入手できます。
  http://www.keil.com/dd/docs/datashts/silabs/c8051f32x.pdf

16.4.3.1 DriverWizard で生成される USB デバイス ファームウェア ファイル

デバイスのファームウェア コードを生成するとき、DriverWizard は **xxx_FW** ディレクトリに次のファイルを作成します:

- **periph.c**: **WinDriver\wdf\<vendor>\<hardware>\include\ periph.h** ヘッダー ファイル (例: **WinDriver\wdf\cypress\FX2LP\include\ periph.h**) [16.3.1] で宣言される、デバイス用の USB 周辺機器機能をサポートする関数の実装を含む C のソース ファイル。関数の実装は、DriverWizard で定義した特定のデバイス設定により行われます。
- DriverWizard で定義した情報を利用する、デバイス記述子情報:
  - Cypress EZ-USB FX2LP CY7C68013A ハードウェアの場合:
    wdf_dscr.a51: アセンブリ ファイル
  - Microchip PIC18F4550、Philips PDIUSBD12 および Silicon Laboratories C8051F320 ハードウェアの場合:
    wdf_dscr.c および **wdf_dscr.h** (Microchip および Silicon Laboratories): C ファイル
- **build.bat**: ファームウェア コードのビルド用のコマンドライン ユーティリティ
- **xxx.Uv2/mcp/wsp/mak**: ([Select Code Generation Options] ダイアログ ボックスでコンパイラ / IDE を選択した場合) 指定されたコンパイラ / IDE (Keil *u*Vision / Microchip MPLAB / Silicon Laboratories IDE / Turbo C) でコードをビルドするための project / make ファイル
- Microchip PIC18F4550 ボードの場合:
  - **xxx.lkr**: リンカ ファイル
  - ボード用の Mass Storage ファームウェアを生成する場合:
    - **wdf_xxx_hw.c**: ハードウェア固有のストレージ メディア アクセス関数 (**18F4550\include\classmsd\wdf_disk.h** ヘッダー ファイルで宣言される [16.3.2]) の実装用 stub を含む C ソース ファイル

次のファイルは、WinDriver USB Device ファームウェア ライブラリのソース コードを含みます。これらのファイルは、WinDriver USB Device ツールキットの登録版 (評価版と登録版の違いについては、セクション 16.4.3.2 を参照) を使用している場合にのみ生成されます。

- **main.c**: ファームウェアのメインのエントリ ポイントの実装を含む C ソース ファイル。Silicon Laboratories C8051F320 ハードウェアおよび Philips PDIUSBD12 ハードウェアをベースとしたデバイスの場合、必要な USB 割り込みサービス ルーチン (`USB_ISR()` (Silicon Laboratories) / `UsbISR()` (Philips)) の実装も含まれます。

  注意: Philips コードの ISR の実装は、プラットフォームに依存します。デフォルトの実装は D12-ISA (PC) Eval Kit で、ターゲットは x86 です。その他のマイクロコントローラをサポートするように、実装を変更することができます (セクション 16.4.3.2 を参照)。
- **wdf_<vendor>_lib.c** (Cypress、Microchip および Silicon Laboratories の場合。例: **wdf_cypress_lib.c**) / **<hardware>_lib.c** (Philips の場合。例: **d12_lib.c**): 選択されたターゲット ハードウェア用の WinDriver USB Device ファームウェア ライブラリ関数の実装を含む C ソース ファイル

- Microchip PIC18F4550 ボードの場合:
  - **wdf_usb9.c**: **wdusb9_.h** ヘッダー ファイル [16.3.2] で宣言される、ファームウェア ライブラリ USB ディクスリプタ関数の実装を含む C ソース ファイル
  - ボード用の Mass Storage デバイス ファームウェアを生成する場合:
    - **wdf_msd.c**: **WinDriver\wdf\microchip\18F4550\include\class\msd\wdf_msd.h** ヘッダー ファイル [16.3.2] で宣言される、Mass Storage Class ファームウェア関数の実装を含む C ソース ファイル
- Philips PDIUSBD12 の場合:
  - **d12_ci.h** および **d12_ci.c**: PDIUSBD12 コマンド インターフェイスの一般的な定義と関数の宣言を含むヘッダー ファイル (**d12_ci.h**)、および コマンド関数の実装を含む C ソース ファイル (**d12_ci.c**)
  - **d12_io.c**: **d12_io.h** ヘッダー ファイル [16.3.2] で宣言される、ハードウェア固有のファームウェア ライブラリ関数の実装を含む C ソース ファイル

16.4.3.2 DriverWizard で生成されたファームウェアのビルド

生成されたファームウェア コードをビルドするには、次のいずれかの方法を使用します。

- コマンドラインから生成された **build.bat** ユーティリティを実行する
- サポートされている IDE (Cypress および Silicon Laboratories の場合は Keil *u*Vision。Microchip の場合は Microchip MPLAB。Silicon Laboratories の場合は Silicon Laboratories IDE) 用のプロジェクト ファイルの生成を選択した場合、この IDE を使用して生成されたプロジェクトをビルドする

ビルド出力は、**xxx.hex** ファームウェア ファイル (Cypress、Microchip および Silicon Laboratories の場合) / **XXX.EXE** ファームウェア ファイル (Philips PDIUSBD12 の場合) です。**xxx** はファームウェア プロジェクト用に選択した名前です。

注意: 生成される **build.bat** およびコンパイラ固有のプロジェクト ファイルまたは make ファイルは、WinDriver USB Device の登録版と評価版で異なります。また、出力も異なります。

評価版の場合、評価用ファームウェア ライブラリが使用され、転送に制限 (標準のライブラリの場合は最大 25,000、Microchip PIC18F4550 Mass Storage ライブラリの場合は 1,000,000) があります ([16.3.5] を参照)。

登録版の場合、生成されたライブラリ ソース ファイルが使用され、転送に制限はありません。

Philips PDIUSBD12 ファームウェア ライブラリおよびこのライブラリを利用するサンプルは、ISA カードを使用して PDIUSBD12 ベースのボードを x86 に接続する D12-ISA (PC) Eval Kit をターゲットとしています。このため、作成されるファームウェアは、DOS 実行ファイルです。登録版の WinDriver USB Device ユーザーは、他のマイクロコントローラをサポートするために、ライブラリの x86 ハードウェア固有の部分 (特に、**d12_io.h**、**d12_io.c** [16.4.3.1] および **main.c** のISR) とサンプル ソース コードを変更し、サポートされている方法でそのマイクロコントローラ用のファームウェアをビルドおよびダウンロードすることができます。

Windriver USB Device ツールキットの評価期間中に作成したファームウェア プロジェクト ファイル (**xxx.wdp**) がある場合は、WinDriver USB Device ツールキットを登録後に、これらのファイルを開いてウィザードでファームウェア コードを再生成し、新しい登録版の **build.bat** とプロジェクト ファイルを作成して

ください。それから、これらのファイルを使用して登録版のフル機能のファームウェアをビルドし、ファームウェアをデバイスにダウンロードしてください。

16.4.3.3 デバイスへのファームウェアのダウンロード

ファームウェアをビルドした後、ハードウェアのベンダーのファームウェア ダウンロード ツールを使用してファームウェアをハードウェアにダウンロードします。

注意: WinDriver USB ドライバ開発キットの登録版または評価版を使用している場合、サンプル ファームウェア ダウンロード アプリケーション (Cypress: `WinDriver\ cypress\ firmware_sample\WIN32\ download_sample.exe`、Microchip: `WinDriver\ microchip\ pic18f4550\ bootloader_sample\WIN32\bootloader_demo.exe`) を使用して、Cypress EZ-USB FX2LP CY7C68013A ファームウェアおよび標準のMicrochip PIC18F4550 ファームウェアをダウンロードできます。

16.4.4 ハードウェアの診断およびデバッグ

ファームウェアをデバイスにダウンロードしたら、セクション [5.2] のように、DriverWizard ユーティリティを使用してファームウェアをデバッグできます (この章の USB の説明を参照)。

注意: セクション [5.2] で説明されているデバイス ドライバ コード生成オプションは、WinDriver USB Device ライセンスの一部ではありません。

16.4.5 USB デバイス ドライバの開発

デバイスの開発が完了した後、WinDriver USB ドライバ開発キットの登録版ユーザーの場合、または WinDriver の評価版を使用している場合、第 6 章 で説明したように、WinDriver を使用して対象のデバイス用のドライバを開発できます。

セクション [16.4.2] で説明したように、両方のライセンスがある場合、DriverWizard のファームウェア生成ダイアログから WinDriver USB デバイス ドライバ アプリケーションのスケルトンを生成するオプションが追加されます。

第 17 章
PCI Express

17.1 PCI Express の概要

PCI Express (PCIe) バス アーキテクチャ (以前の 3GIO または 3rd Generation I/O) は、将来に向けた PC I/O 規格を確立するために、インテルとその他のリーディング カンパニー (IBM、Dell、Compaq、HP および Microsoft) の協力のもとに開発されました。

PCI-Express は、PCI 2.2 バスよりもスケーラビリティが高い広帯域を提供します。

標準 PCI 2.2 バスは、すべてのデータが単一の並列データ バスを介して設定速度で送信されるように設計されています。標準 PCI 2.2 バスでは、接続されているすべてのデバイス間で帯域幅が共有され、デバイス間に優先度はありません。最大帯域幅は 132MB/s で、接続されているすべてのデバイス間で共有されます。

PCI Express は、ポイント・ツー・ポイントのシリアル伝送を行う個別のクロック信号を持つレーンから構成されています。各レーンは、データの双方向伝送を同時に行うことができる 2 本のデータ レーンから構成されています。バス スロットは、バス上のデータ フローを制御するスイッチに接続されていて、PCI Express デバイスと PCI Express スイッチ間の接続をリンクと呼びます。各リンクは、1 本または複数のレーンから構成されていて、1 本のレーンで構成されるリンクは x1 リンク、2 本のレーンで構成されるリンクは x2 リンクと呼びます。PCI Express は x1、x2、x4、x8、x12、x16、および x32 のリンク幅 (レーン) をサポートしています。PCI Express アーキテクチャでは、レーンあたり約500MB/s の最大帯域幅が可能です。このため、潜在的な最大帯域幅は、x1 では 500MB/s 、x2 では 1,000MB/s、x4 では 2,000MB/s 、x8 では 4,000MB/s 、x12 では 6,000MB/s、x16 では 8,000MB/s となります。これは、標準の 32 ビット PCI バスの最大帯域幅である 132MB/s と比べて大きな改善であるといえます。

広帯域幅をサポートする PCI Express は、増加を続けるハードドライブ コントローラ、ビデオ ストリーミング デバイス、ネットワーク カードなど、広帯域を必要とするデバイスにとって理想的です。

PCI Express バスのデータ フローを制御するスイッチを使用することによって、複数のデバイス間でバスを共有する代わりに、各デバイスがバスに直接アクセスできるようになり、共有 PCI バスを改善できます。これにより、各デバイスは、単一の共有バスの最大帯域幅を奪い合うことなく、帯域幅をフルに使用することができます。

これらに加え、各デバイスが PCI Express バスにアクセスするためのレーンを考慮すると、PCI Express では、以前の PCI と比べより広域な帯域幅の制御が可能です。また、PCI Express では、デバイス同士が直接通信できます (ピアツーピア通信)。

さらに、PCI Express バストポロジーでは、共有バストポロジーとは異なり、転送およびリソース管理が中央化されています。このため、PCI Express は QoS (品質保証) をサポートしています。PCI Express スイッチはパケットに優先度を与え、リアルタイム ストリーミング パケット (例: ビデオ ストリームやオーディオ ストリームなど) は、タイム クリティカルでないパケットに対し優先権を得ることができます。

PCI スロット、AGP スロット、または PCI-X などその他の新しい I/O バス ソリューションと比べ、低価格で製造できることも PCI Express の利点です。

PCI Express は、既存の PCI バス、PCI デバイス (これらのバスのアーキテクチャが異なる場合も含め) とハードウェアおよびソフトウェアの完全な互換性を維持するよう設計されています。

PCI 2.2 バスとの下位互換性の一部として、以前の PCI 2.2 デバイスを PCI Express-to-PCI ブリッジを介して PCI Express システムに挿入することができます。PCI Express-to-PCI ブリッジは、マザーボードまたは外部カードのいずれかにあり、PCI Express パケットを標準 PCI 2.2 バス信号に変換します。

17.2 WinDriver PCI Express

WinDriver は、PCI Express ボードにおいて標準 PCI 機能との下位互換性を完全にサポートしています。サポートには、豊富な API セット、ハードウェアのデバッグおよびドライバ コード生成用のコードサンプルとグラフィカルな DriverWizard が含まれます。これらのサポートは、PCI Express デバイス (以前の PCI デバイスとの下位互換性を持つよう設計されている) でも利用できます。

また、WinDriver の PCI API を使用し、PCI Express-to-PCI ブリッジおよび スイッチ (例: PLX 8111/8114 ブリッジ、PLX 8532 スイッチ) によって PC に接続された PCI デバイスと簡単に通信することもできます。

さらに、WinDriver (Windows および Linux) では、PCI Express の拡張設定空間へのアクセスを簡単にする API セットを提供してます (詳細は、`WDC_PciReadCfgXXX()` 関数、`WDC_PciWriteCfgXXX()` 関数、低水準 `WD_PciConfigDump()` 関数の説明を参照)。

PCI Express バスの拡張サポート (特定の PCI Express WinDriver ライブラリおよび特別な機能) については、WinDriver の将来のリリースで追加される予定です。

WinDriver

ユーザーズ ガイド

2007 年 6 月 5 日

発行　エクセルソフト株式会社
〒108-0014 東京都港区芝5-1-9 ブゼンヤビル4F
TEL 03-5440-7875　FAX 03-5440-7876
E-MAIL: xlsoftkk@xlsoft.com
ホームページ: http://www.xlsoft.com/

Translated by

米国 XLsoft Corporation
12K Mauchly
Irvine, CA 92618 USA
URL: http://www.xlsoft.com/
E-Mail: sales@xlsoft.com