

JUNGO
WinDriver

ユーザーズ ガイド



エクセルソフト株式会社

COPYRIGHT

Copyright (c) 1997 – 2009 Jungo Ltd. All Rights Reserved.

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できます。
- 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。

WinDriver はイスラエル国 Jungo 社の商標です。

Windows、Win32、Windows 98、Windows Me、Windows CE、Windows NT、Windows 2000、Windows XP、Windows Server 2003、Windows Server 2008 および Windows Vista は米国マイクロソフト社の登録商標です。

その他の製品名、機種名は、各社の商標または登録商標です。

エクセルソフト株式会社

〒108-0014 東京都港区芝 5-1-9 プゼンヤビル 4F

TEL 03-5440-7875 FAX 03-5440-7876

E-MAIL: xlsoftkk@xlsoft.com

Home Page: <http://www.xlsoft.com/>

Rev. 10.0 – 1/2009

目次

目次.....	3
図表.....	9
第 1 章 WinDriver の概要	11
1.1 はじめに.....	11
1.2 背景.....	11
1.2.1 チャレンジ	11
1.2.2 WinDriver の特長.....	12
1.3 WinDriver の処理速度	13
1.4 最後に	13
1.5 WinDriver の利点	13
1.6 WinDriver のアーキテクチャ	15
1.7 WinDriver がサポートするプラットフォーム	17
1.8 評価版 (Evaluation Version) の制限	17
1.9 WinDriver を使用してドライバを開発するには	17
1.9.1 Windows および Linux	17
1.9.2 Windows CE	18
1.10 WinDriver ツールキットの内容.....	18
1.10.1 WinDriver のモジュール	18
1.10.2 ユーティリティ	19
1.10.3 特定チップセットのサポート	20
1.10.4 サンプル.....	20
1.11 WinDriver で作成したドライバを配布できますか	21
第 2 章 デバイス ドライバの理解.....	22
2.1 デバイス ドライバの概要.....	22
2.2 機能によるドライバの分類	22
2.2.1 モリシック ドライバ	22
2.2.2 レイヤード ドライバ	23
2.2.3 ミニポート ドライバ	24
2.3 OS によるドライバの分類	24
2.3.1 WDM ドライバ	24

2.3.2	VxD ドライバ.....	25
2.3.3	Unix デバイス ドライバ.....	25
2.3.4	Linux デバイス ドライバ.....	25
2.4	ドライバのエントリー ポイント.....	25
2.5	ハードウェアとドライバの連結.....	26
2.6	ドライバとの通信.....	26
第 3 章 WinDriver USB の概要.....		27
3.1	USB の概要.....	27
3.2	WinDriver USB の利点.....	27
3.3	USB のコンポーネント.....	28
3.4	USB デバイスのデータ フロー	28
3.5	USB データ交換.....	29
3.6	USB データ転送タイプ.....	30
3.6.1	コントロール転送 (Control Transfer).....	30
3.6.2	等時性転送 (Isochronous Transfer)	31
3.6.3	割り込み転送 (Interrupt Transfer)	31
3.6.4	バルク転送 (Bulk Transfer)	31
3.7	USB 設定.....	31
3.8	WinDriver USB	33
3.9	WinDriver USB のアーキテクチャ	34
3.10	WinDriver USB を使って作成できるドライバ.....	35
第 4 章 WinDriver のインストール.....		36
4.1	動作環境	36
4.1.1	Windows	36
4.1.2	Windows CE	36
4.1.3	Linux	36
4.2	WinDriver のインストール	37
4.2.1	Windows にインストールするには	37
4.2.2	WinDriver CE のインストール	40
4.2.3	Linux に WinDriver をインストールするには	43
4.3	アップグレード版のインストール	46
4.4	インストールの確認	46
4.4.1	Windows および Linux コンピュータの場合	46
4.4.2	Windows CE コンピュータの場合	47
4.5	WinDriver をアンインストールするには	47
4.5.1	Windows WinDriver をアンインストールするには	47
4.5.2	Linux から WinDriver をアンインストールするには	49

第 5 章 DriverWizard.....	50
5.1 DriverWizard の概要.....	50
5.2 DriverWizard の使い方.....	51
5.2.1 WinDriver API呼び出しのログ.....	64
5.2.2 DriverWizard のログ.....	64
5.2.3 自動コード生成.....	64
5.2.4 生成されたコードをコンパイルする.....	65
5.2.5 Bus Analyzer の統合 - Ellisys Visual USB.....	66
第 6 章 ドライバの作成.....	68
6.1 WinDriver でデバイス ドライバを開発するには.....	68
6.2 DriverWizard を使わずにドライバを記述するには.....	69
6.2.1 必要な WinDriver ファイルのインクルード.....	69
6.2.2 コードの作成: PCI / ISA ドライバの場合.....	70
6.2.3 コードの作成: USB ドライバの場合.....	71
6.3 Windows CE で開発を行うには.....	71
6.4 Visual Basic および Delphi で開発を行うには.....	72
6.4.1 DriverWizard を使用する.....	72
6.4.2 サンプル.....	72
6.4.3 Kernel PlugIn.....	72
6.4.4 ドライバを生成するには.....	72
第 7 章 デバッグ.....	73
7.1 ユーザー モード デバッグ.....	73
7.2 Debug Monitor.....	73
7.2.1 グラフィック モードで Debug Monitor を使用するには - wddebug_gui	73
7.2.2 コンソール モードで Debug Monitor を使用するには - wddebug	76
第 8 章 特定のチップ セットの拡張サポート.....	79
8.1 概要	79
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	79
第 9 章 実行に当たっての問題	81
9.1 DMA の実行	81
9.1.1 Scatter/Gather DMA	82
9.1.2 Contiguous Buffer (連続バッファ) DMA	84
9.1.3 SPARC での DMA の実行	86
9.2 割り込み処理	86
9.2.1 割り込み処理の概要	86

9.2.2	WinDriver の割り込み処理手順	87
9.2.3	ハードウェアがサポートする割り込みタイプの決定	88
9.2.4	PCI カードの割り込みタイプの決定	89
9.2.5	カーネルモードの割り込み転送コマンドの設定方法	89
9.2.6	WinDriver の MSI / MSI-X 割り込み処理	91
9.2.7	ユーザー モードの WinDriver 割り込み処理のコード例	92
9.2.8	Windows CE の割り込み	94
9.3	USB コントロール転送	95
9.3.1	USB データ交換	95
9.3.2	コントロール転送の詳細	96
9.3.3	セットアップ パケット	97
9.3.4	USB セットアップ パケットのフォーマット	97
9.3.5	標準デバイスが要求するコード	98
9.3.6	セットアップ パケットの例	98
9.4	WinDriver でコントロール転送を行う	100
9.4.1	DriverWizard でのコントロール転送	100
9.4.2	WinDriver API でのコントロール転送	101
9.5	機能 USB データ転送	102
9.5.1	機能 USB データ転送の概要	102
9.5.2	シングル ブロッキング転送	102
9.5.3	ストリーミング データ転送	102
9.6	64 ビット OS のサポート	104
9.6.1	64 ビット アーキテクチャのサポート	104
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	104
9.6.3	64 ビットおよび 32 ビットのデータ型	104
9.7	バイト オーダー	104
9.7.1	エンディアンネスとは	104
9.7.2	WinDriver のバイト オーダー マクロ	105
9.7.3	PCI ターゲット アクセスのマクロ	105
9.7.4	PCI マスター アクセスのマクロ	106
第 10 章 パフォーマンスの向上		107
10.1	概要	107
10.1.1	パフォーマンスを向上するためのチェックリスト	107
10.2	ユーザー モード ドライバのパフォーマンスの向上	108
10.2.1	メモリ マップの領域への直接アクセス	108
10.2.2	ロック転送および複数の転送のグループ化	109
10.2.3	64 ビット データ転送を行う	109
第 11 章 Kernel PlugIn について		111

11.1	Kernel PlugIn の概要	111
11.2	Kernel PlugIn を作成する前に.....	111
11.3	期待される効果	111
11.4	開発プロセスの概要	112
11.5	Kernel PlugIn の構造	112
11.5.1	構造の概要	112
11.5.2	WinDriver のカーネルと Kernel Plugin の相互作用	113
11.5.3	Kernel Plugin コンポーネント	113
11.5.4	Kernel PlugIn イベントシーケンス	113
11.6	Kernel PlugIn の仕組み	116
11.6.1	Kernel PlugIn ドライバの作成に必要な条件	117
11.6.2	Kernel PlugIn の実装	117
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	122
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	123
11.6.5	Kernel PlugIn での割り込み処理	126
11.6.6	メッセージの受け渡し	128
第 12 章 Kernel PlugIn の作成		130
12.1	Kernel PlugIn が必要かどうかを確認する	130
12.2	ユーザー モードのソース コードを用意する	130
12.3	Kernel PlugIn プロジェクトの新規作成	131
12.4	Kernel PlugIn へのハンドルの作成	132
12.5	Kernel PlugIn での割り込み処理の設定	132
12.6	Kernel PlugIn での I/O 処理の設定	133
12.7	Kernel PlugIn ドライバのコンパイル	133
12.7.1	Windows でのコンパイル	133
12.7.2	Linux でのコンパイル	135
12.8	Kernel PlugIn ドライバのインストール	136
12.8.1	Windows の場合	136
12.8.2	Linux の場合	136
第 13 章 ドライバの動的ロード		137
13.1	なぜ動的にロード可能なドライバが必要なのか	137
13.2	Windows の動的ドライバ ロード	137
13.2.1	Windows ドライバの種類	137
13.2.2	WDREG ユーティリティ	137
13.2.3	windrvr6.sys INF ファイルの動的ロード / アンロード	141
13.2.4	Kernel PlugIn ドライバを動的にロード / アンロード	141
13.3	Linux の動的ドライバ ロード	142

13.4	Windows Mobile の動的ドライバロード	142
第 14 章 ドライバの配布		143
14.1	WinDriver の有効なライセンスを取得するには.....	143
14.2	Windowsの場合	143
14.2.1	配布パッケージの用意	144
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.....	144
14.2.3	ターゲットコンピュータに Kernel PlugIn をインストール	146
14.3	Windows CE の場合	147
14.3.1	新規の Windows CE プラットフォームへの配布	147
14.3.2	Windows CE コンピュータへの配布.....	149
14.4	Linux の場合	149
14.4.1	カーネル モジュール	150
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	151
14.4.3	Kernel Plugin モジュール	151
14.4.4	インストール スクリプト.....	152
第 15 章 ドライバのインストール - 高度な問題		153
15.1	Windows INF ファイル	153
15.1.1	なぜ INF ファイルを作成する必要があるのか	153
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.....	153
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.....	154
15.2	WinDriver カーネル ドライバの名前変更	155
15.2.1	Windows ドライバの名前変更	155
15.2.2	Linux ドライバの名前変更	157
15.3	デジタル ドライバの署名と認証 - Windows 2000 / XP / Server 2003 / Server 2008 / Vista	158
15.3.1	概要	158
15.3.2	WinDriver ベースのドライバのドライバ署名と認証	159
15.4	Windows XP Embedded の WinDriver のコンポーネント	161
第 16 章 PCI Express		163
16.1	PCI Express の概要	163
16.2	WinDriver PCI Express	164

図表

図 1.1: WinDriver アーキテクチャ	16
図 2.1: モノリシック ドライバ	23
図 2.2: レイヤード ドライバ	23
図 2.3: ミニポート ドライバ	24
図 3.1: USB エンドポイント	29
図 3.2: USB パイプ	30
図 3.3: デバイス ディスクリプタ	32
図 3.4: WinDriver USB アーキテクチャ	34
図 4.1: ライセンスの登録	39
図 5.1: WinDriver のプロジェクトを開く、または新規作成	52
図 5.2: デバイスの選択	52
図 5.3: DriverWizard INF ファイル情報	53
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)	54
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)	55
図 5.7: PCI のリソース画面	57
図 5.8: レジスタの定義	57
図 5.9: メモリおよび I/O の Read / Write	58
図 5.10: 割り込みの Listen (確認)	58
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	59
図 5.12: USB デバイスのインターフェースの選択	60
図 5.13: USB コントロール転送	61
図 5.14: パイプの確認	62
図 5.15: パイプへの書き込み	62
図 5.16: コード生成のオプション	63
図 5.17: ドライバオプションの選択	63
図 5.18: Ellisys Visual USB の統合	67
図 7.1: Debug Monitor の起動	74
図 7.2: Debug Options の設定	75
図 9.1: USB データ交換	96
図 9.2: USB のリードとライト	97
図 9.3: カスタム要求	100
図 9.4: 要求一覧	101

図 9.5: USB 要求ログ	101
図 11.1: KernelPlugIn の構造	112
図 11.2: Kernel PlugIn なしでの割り込みの処理	127
図 11.3: Kernel PlugIn ありでの割り込み処理	127

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 2000 / XP / Server 2003 / Server 2008 / 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 / Server 2008 / 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.xlssoft.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 および Linux) 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、プログラマは次の内容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 (WDK、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 (WDK、ETK、DDI / DKI など) 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 2000 / XP / Server 2003 / Server 2008 / Vista、Windows CE.NET、Windows Embedded CE v6.00、Windows Mobile 5.0 / 6.0 および Linux** 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 [9.2.8] 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。

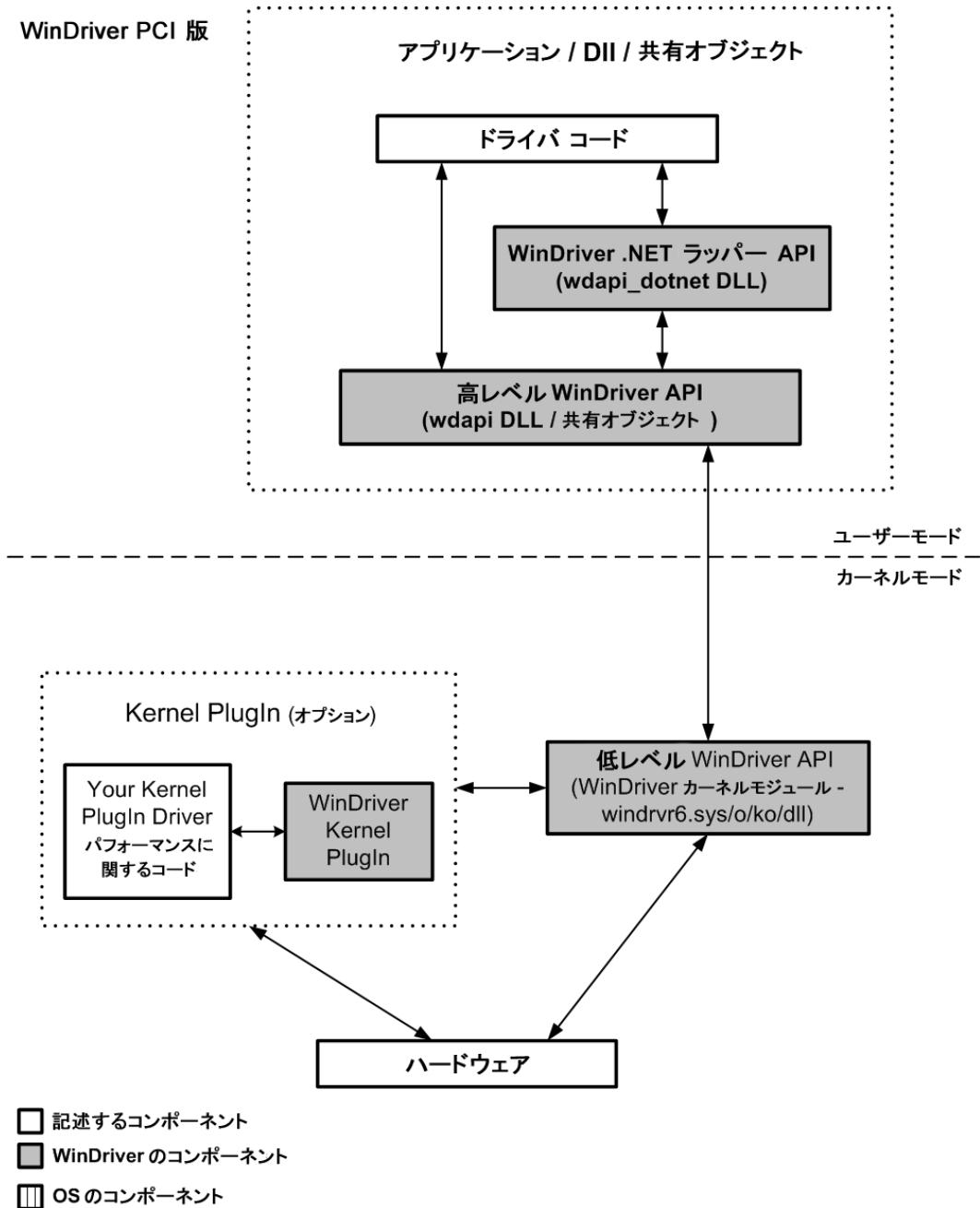
これで、すべての対応するプラットフォームから新しいハードウェアにアクセスするアプリケーションを作成できます。(コードは Windows 2000 / XP / Server 2003 / Server 2008 / Vista プラットフォームでバイナリ互換性があります。そのため、これらの OS 間でドライバを移植する場合は再ビルドする必要はありません。)

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 2000 / XP / Server 2003 / Server 2008 / 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 2000 / XP / Server 2003 / Server 2008 / Vista、Windows CE.NET、Windows Embedded CE v6.00、Windows Mobile 5.0 / 6.0 および Linux 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。
- WDK、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



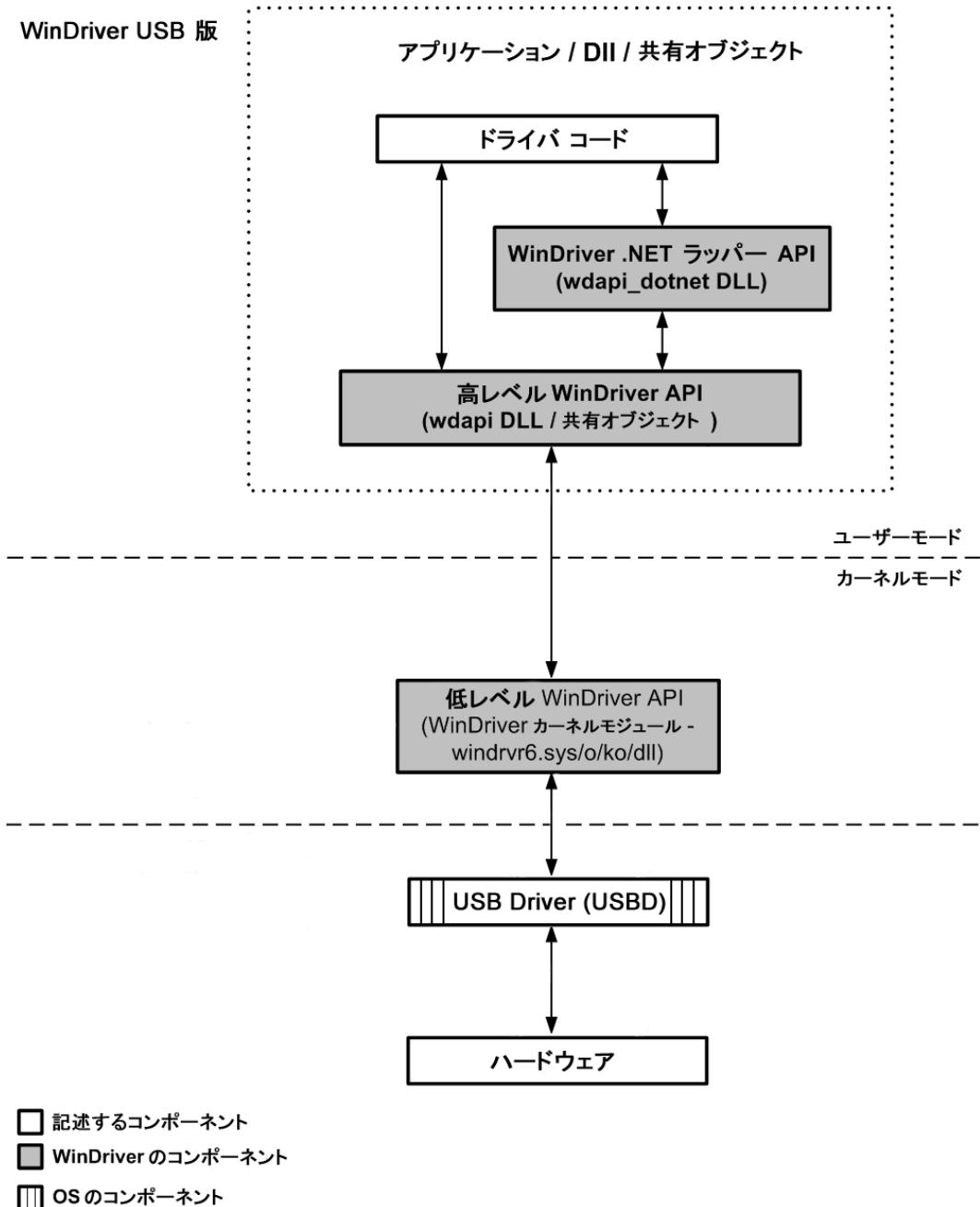


図 1.1: WinDriver アーキテクチャ

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、ユーザーで実行されてもパフォーマンスにあまり影響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で実行されてもパフォーマンスにあまり影響ないように設計されています。このように場合、ユーザーで実行されてもパフォーマンスが重要なモジュール (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、ユーザーで実行されてもパフォーマンスが重要なモジュールをカーネル モードで実行できます。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、ユーザーとカーネル モードで実行されるモジュールをカーネル モードで実行できます。

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 2000 / XP / Server 2003 / Server 2008 / 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

Windows NT 4.0 と VxWorks は以前のバージョンでサポートしています。

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 2000 / XP / Server 2003 / Server 2008 / 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 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、再ロードが必要があります。
- 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 および Linux

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. 離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離型とします。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を有効にし、割り込みを確認し、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 マシンにターゲット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。主に以下のファイルが含まれます:
 - `windrivr.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** と **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) ライブリの宣言および定義します。
- **windrivr_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。
- **windrivr_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): ユーザーに配布するファイル。
- WinDriver Kernel PlugIn: Kernel PlugIn ドライバ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 6466、9030、9050、9052、9054、9056、9080 および 9656 – これらは `WinDriver/plx` ディレクトリに保存されています。
- AMCC S5933 - `WinDriver/amcc` に保存されています。
- Altera pci_dev_kit - `WinDriver/altera/pci_dev_kit/` に保存されています。
- Xilinx VirtexII および Virtex 5 - `WinDriver/xilinx/` に保存されています。

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/` に保存されています。
- Microchip PIC18F4550 - `WinDriver/microchip/pic18f4550/` に保存されています。
- Philips PDIUSBD12 - `WinDriver/pdiusbd12/` に保存されています。
- Texas Instruments TUSB3410、TUSB3210、TUSB2136、TUSB5052 - `WinDriver/ti/` に保存されています。
- Agere USS2828 - `WinDriver/agere/` に保存されています。
- Silicon Laboratories C8051F320 USB - `WinDriver/silabs/` に保存されています。

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` と `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、WDK、ETK、DDI / DKI 関数を使用してハードウェアを制御します。

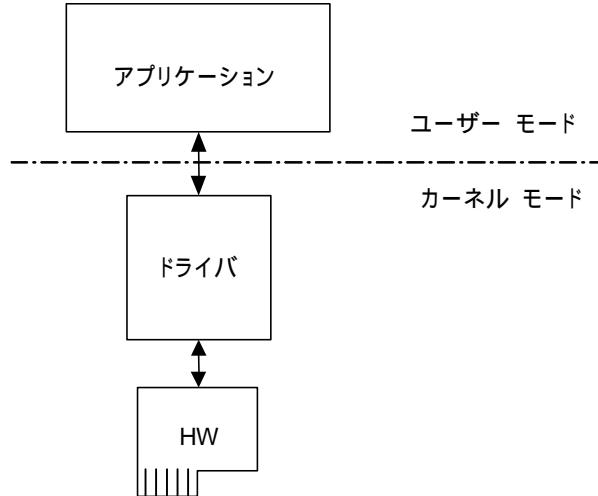


図 2.1: モノリシック ドライバ

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および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、暗号化および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。

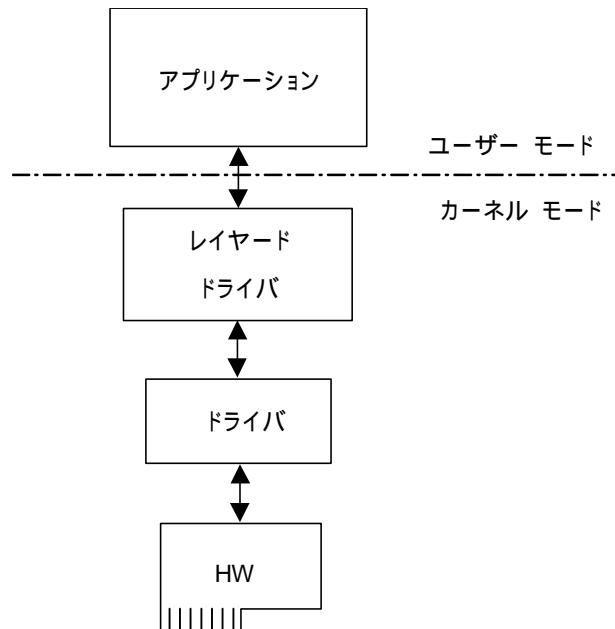


図 2.2: レイヤード ドライバ

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 4.0 / 2000 / XP / Server 2003 / Server 2008 / Vista でサポートされています。

Windows NT 4.0 / 2000 / XP / Server 2003 / Server 2008 / 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。

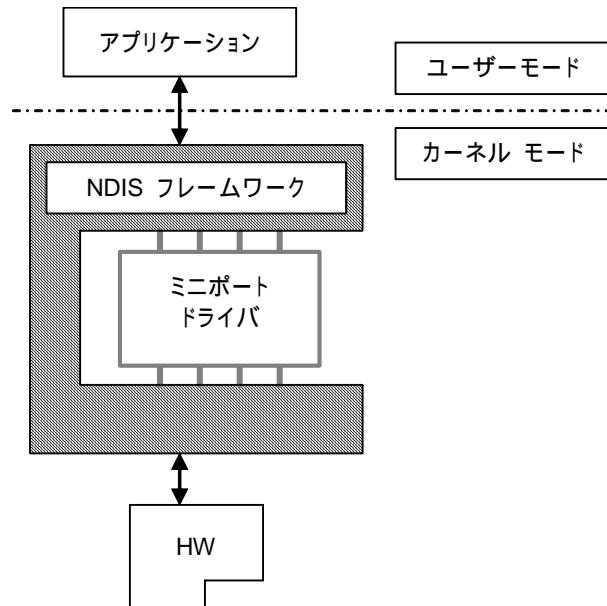


図 2.3: ミニポート ドライバ

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 4.0 / 2000 / XP / Server 2003 / Server 2008 / 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ーとすることがですが、ドライバの構造自体は、レイヤ化しません。

2.3.3 Unix 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 レイヤーのレイヤード 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所以です。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。

2.4 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()` ルーチンの役割をします。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()` ルーチンで定義します。`init_module()` ルーチンは、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きます。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。

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どがあなた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できます (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 480Mbps (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できます。

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 バブリッシング、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。
- デバイス ドライバの自動マッピングと自動設定。

- コンピュータの起動中にデバイスを接続しても周辺機器を再設定可能。
- データ転送率が数 Kb/s から数百 Mb/s まで幅広いデバイスの帯域幅に最適。
- 同じケーブルで等時性転送と非同期転送をサポート。
- 複数のデバイスの同時処理をサポート (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。
- 帯域幅と短い待ち時間を保証。電話やオーディオに最適 (等時性転送は、バス帯域幅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。
- 堅牢性: エラー処理機能を備え、動的なデバイスの着脱をサポート。
- PC 業界の標準。
- 周辺機器とホスト ハードウェアの統合を最適化。
- 実装のコストを抑え、周辺機器の開発コストを削減。
- 低価格なケーブルとコネクタ。
- 電源管理や電源供給機能を提供。

3.3 USB のコンポーネント

以下、USB を構成する主なコンポーネントです:

USB ホスト: USB ホスト プラットフォームは、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 ドライバが起動します。USB ホスト コントローラは、ホストと USB 周辺機器間のインターフェイスです。ホストは、USB デバイスの脱着の検出、ホストとデバイス間のコントロール、データ フローの管理、USB デバイスへの電力供給などの機能があります。

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上の情報をコントロールし、機能を提供する USB デバイスです。通常、USB 機能は、ケーブルを使用し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 ケーブルを使用して複数の機能と組み込みハブを実装する物理パッケージとして、複合デバイスを作成することも可能です。複合デバイスは、ホストへは、取り外し不可能な複数の USB デバイスを持つハブのように見え、外部デバイスとの接続をサポートするポートを持つ場合があります。

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。

ホストと1つのデバイス間でのみピアツーピア通信でデータを転送することができます。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バス上のデータは、ホスト上で動作するソフトウェアのメモリバッファとデバイス上のエンドポイント間を動く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デバイス上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間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、パイプで使用するデータ転送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。ストリームパイプは、割り込み、バルクおよび等時性転送を処理し、これに対し、メッセージパイプは、コントロール転送タイプをサポートします。これらのUSB転送タイプは、次に説明します。

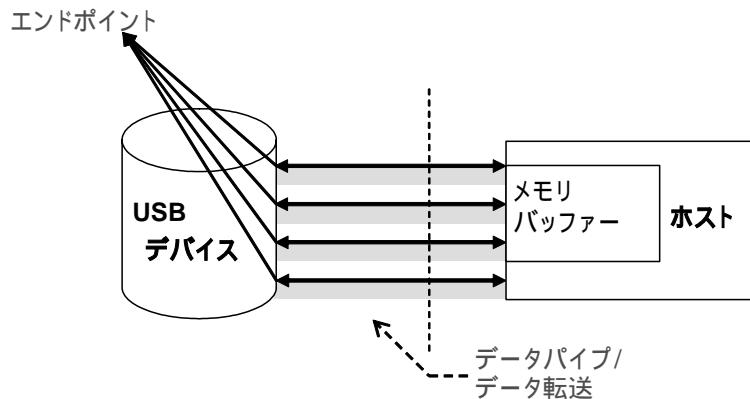


図3.1: USBエンドポイント

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、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。

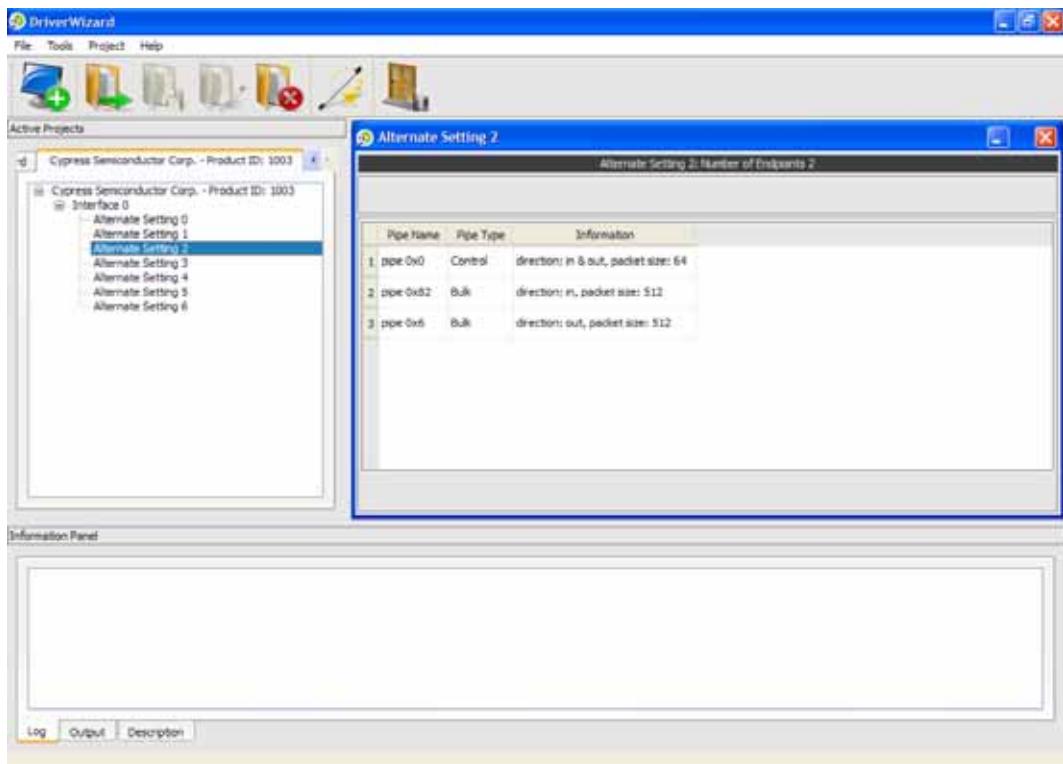


図 3.2: USB パイプ

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 ポイントの間のパイプを使用してデータを転送してホストと通信を行います。USB は 4 つの転送タイプをサポートします。デバイスとソフトウェアの要件に応じて、特定のエンドポイントに対して転送タイプを選択します。特定のエンドポイントの転送タイプは、エンドタイプのディスクリプタで決まります。

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)

コントロール転送を使用して、ホストのソフトウェアとデバイス間で主に設定操作、コマンド操作、ステータス操作をサポートします。低速、フル、および高速デバイスでこの転送タイプを使用します。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、バーストで非定期な通信です。コントロール パイプは双方のパイプで、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電話機など、時間に依存する情報を扱う転送タイプです。フルおよび高速デバイスでこの転送タイプを使用し、低速デバイスでは使用しません。等時性転送は定期的で連続的です。等時性パイプは単方向で、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の場合、各方向に 1 つずつ、2 つの等時性パイプが必要です。USB は、決まった待ち時間の範囲内で USB の帯域幅 (USB フレームの必要なバイト数を予約) へのアクセスを保証し、転送データが十分でない場合を除き、パイプを使用したデータ転送率を保証します。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、データ転送中にエラーが発生してもリトライは行われません。ただし、データの受信側は、バスでエラーが発生したことを判断できます。

3.6.3 割り込み転送 (Interrupt Transfer)

頻度の低い少量のデータを送受信や非同期のタイム フレームで情報をやり取りするデバイスで割り込み転送を使用します。この転送タイプは、低速、フル、高速デバイスで使用できます。割り込み転送タイプは、最大サービス ピリオドを保証し、バス上でエラーが発生した場合、次のピリオドで転送を再試行することを保証します。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)

一般的に時間に依存しないセンシティブな大量のデータを転送するデバイスやプリンタやスキャナなど利用可能な帯域幅をすべて使用するデバイス用にバルク転送を使用します。この転送タイプは、フルおよび高速デバイスで使用できますが、低速デバイスでは使用できません。バルク転送は、非定期で大きいパケットのバースト通信です。バルク転送は、利用可能なバスへのアクセスを許可し、データ転送を保証するが待ち時間は保証しません。エラー チェック メカニズムを持ち、エラーが発生した場合には、再試行します。他のストリーム パイプ (等時性と割り込み) と同様にバルク 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 つ必要です。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。ホストは USB デバイスから設定情報を取得して設定を行います。USB デバイスはディスクリプタの属性を提供します。ディスクリプタはデータを転送する定義済みの構造体とフォーマットです。USB ディスクリプタの詳細は 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 ディスクリプタは 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ディスクリプタは 1 つしかありません。各デバイスには 1 つ以上の設定があり、各設定には 1 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）。

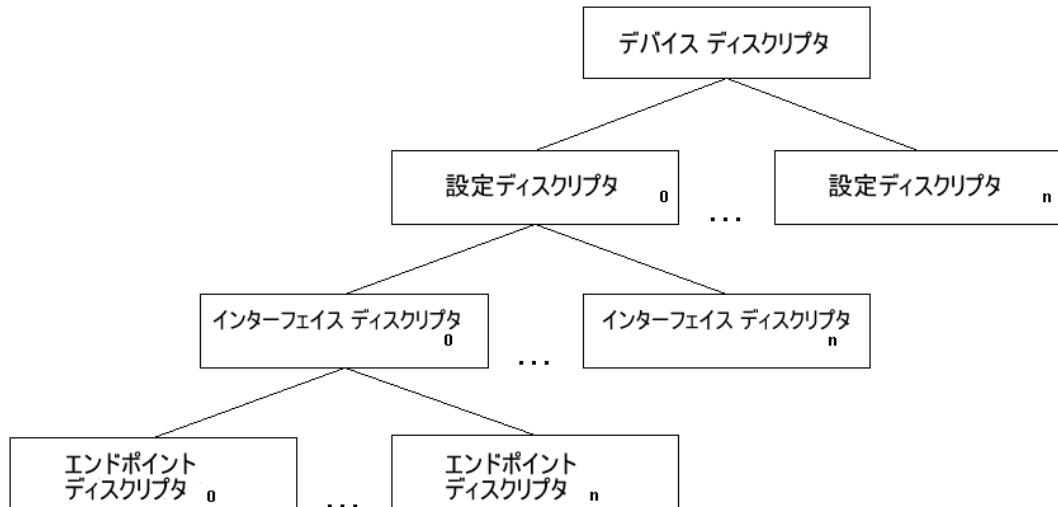


図 3.3: デバイスディスクリプタ

デバイス レベル: デバイスディスクリプタには、USB デバイスに関する一般的な情報（すべてのデバイス設定に関するグローバルな情報）が含まれます。デバイスディスクリプタには、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、必ずデバイスディスクリプタが存在します。

設定レベル: USB デバイスには 1 つ以上の設定ディスクリプタが存在します。各ディスクリプタは、設定の分類分けしたインターフェイスの数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）。一度に 1 つの設定しかロードしません。たとえば、ISDN アダプタは 2 つの異なる設定を持ち、1 つは 128Kbps のインターフェイスを 1 つ持ち、もう 1 つは各 64Kbps インターフェイスを 2 つ持ちます。

インターフェイス レベル: インターフェイスは、デバイスの特定の機能を提供するエンドポイントの関連するセットです。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。インターフェイスディスクリプタは、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する場合のプロトコルの値を表します。さらに、インターフェイスは代替設定を持つこともあります。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更できます。

エンドポイント レベル: 一番低レベルがエンドポイントディスクリプタで、ホストにエンドポイントのデータ転送タイプと最大パケットサイズに関する情報を提供します。等時性エンドポイントの場合、最大パケットサイズを使用してデータ転送に必要なバス時間を予約します（帯域幅）。他のエンドポイントの属性は、バスアクセスの周波数、エンドポイントの番号、エラー処理メカニズムや転送の方向を表します。同じエンドポイントは、異なる代替設定に異なるプロパティを持つことができます（その結果、異なる用途を持ちます）。

ここまで USB の特長を簡単に説明してきましたが、USB の設定処理は複雑に見えますでしょうか。

WinDriver は USB の設定処理を自動化します。WinDriver の GUI アプリケーション DriverWizard ユーティリティと WinDriver に含まれる USB 診断アプリケーションは、USB バスをスキャンし、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始める前に必要な設定を選択できます。

WinDriver は、エンドポイントディスクリプタが持つエンドポイント転送タイプを取得します。WinDriver を使用して作成したドライバは、瞬時にすべての設定情報を取得します。

3.8 WinDriver USB

WinDriver を使用することによって、開発者は USB の仕様や OS の内部構造を学習しなくても、また、OS の開発キットを使用しなくても、高性能な USB ベースのデバイスドライバを簡単に開発できます。たとえば、Windows OS の場合、WDK (Windows Driver Kit) を使用しなくても、また、WDM (Windows Driver Model) を学習しなくても、USB ドライバを開発することができます。

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 2000 / XP / Server 2003 / Server 2008 / 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 2000 / XP / Server 2003 / Server 2008 / 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 デバイスとあらゆるタイプの設定を持つ USB デバイスをサポートする汎用的なツールキットです。

WinDriver USB は、USB の仕様やアーキテクチャをカプセル化するので、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。ドライバのコードを記述する前に、WinDriver USB の GUI アプリケーション DriverWizard ユーティリティを使用して、対象のハードウェアの検出、設定情報の確認、動作確認を簡単に行えます。使いやすい GUI を持つ DriverWizard で、最初に必要な設定、インターフェイス、代替設定を選択します。対象の USB デバイスの検出と設定後、ハードウェアが期待通りに動作するのを確認するために、パイプ上のデータ転送、コントロールリクエストの送信、パイプのリセットなど、デバイスとの通信テストを行います。

ハードウェアの診断を終了したら、DriverWizard を使用して、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 は、ユーザー モードの API を用意しているので、アプリケーションから API を呼び出して、対象のデバイスとの通信を実装できます。WinDriver USB の API には、パイプやデバイスのリセットなど、USB 特有の操作が含まれます。DriverWizard で生成したコードは診断アプリケーションを実装し、WinDriver USB の API を使用して対象のデバイスを制御する方法が確認できます。アプリケーションを使用するには、単純にコードをコンパイルして起動するだけです。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、必要に応じてコードを修正し、対象のデバイスに必要な機能を実装します。

DriverWizard は、対象のデバイスが 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 USB デバイスを正確に認識し処理するために必要です。INF ファイルを作成する理由に関しては、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

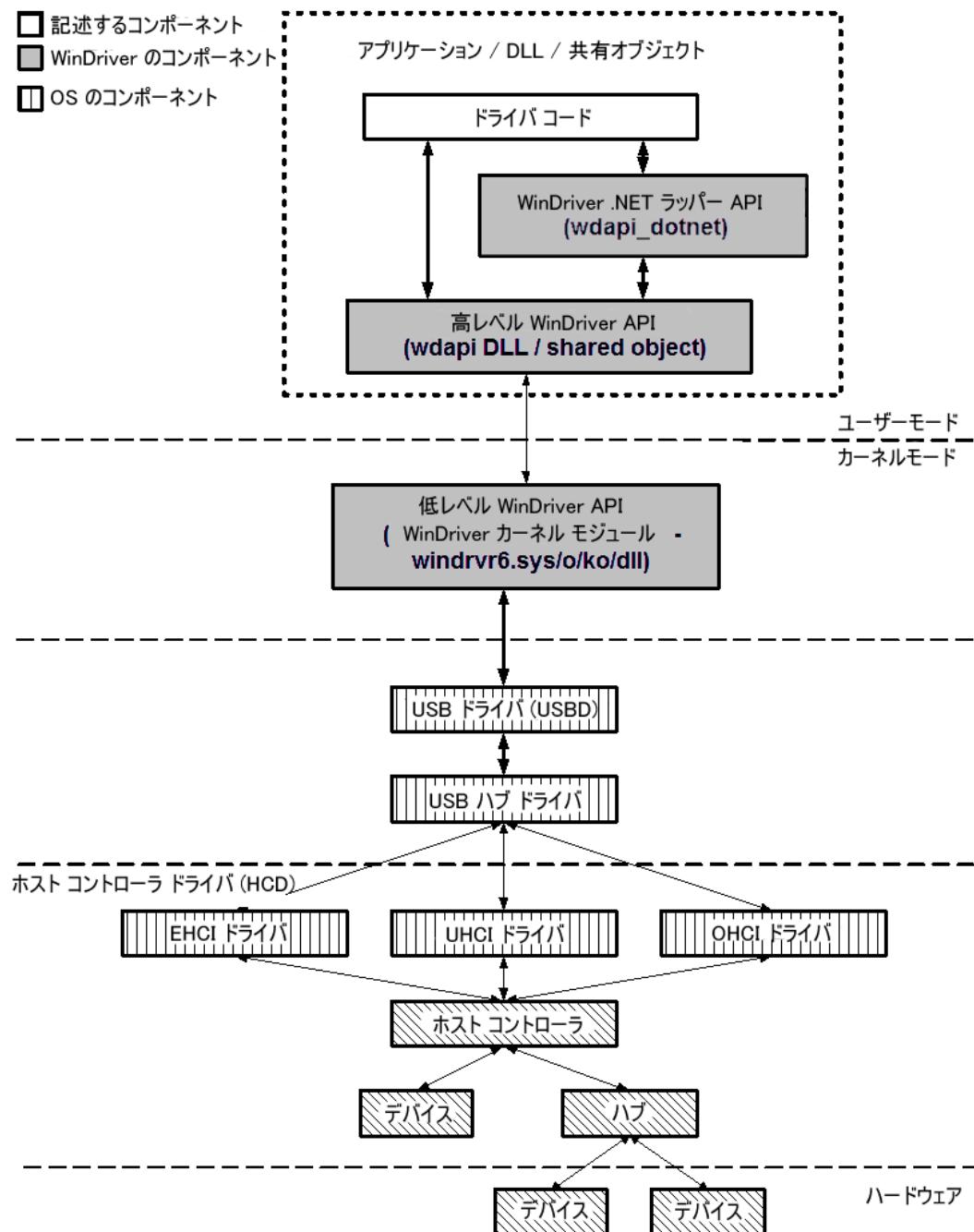


図 3.4: WinDriver USB アーキテクチャ

対象のハードウェアにアクセスするには、アプリケーションは、WinDriver USB の API を使用して、WinDriver のカーネル モジュールを呼び出します。高レベル関数は低レベル関数を利用し、IOCTL を使用して、WinDriver のカーネル 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します。WinDriver のカーネル モジュールは、ネイティブな OS コールを通して対象の USB デバイスのリソースにアクセスします。

USB デバイスと USB デバイス ドライバを抽象化する 2 つのレイヤーがあります。上位のレイヤーが USB ドライバ (USBD) レイヤーで、USB ハブ ドライバや USB コア ドライバを含みます。下位のレイヤーがホストコント

ロールドライバ(HCD)レイヤーです。HCD レイヤーと USBD レイヤーの役割は定義されておらず、OS に依存します。HCD と USBD の両者はソフトウェアインターフェイスで、OS のコンポーネントであり、HCD レイヤーが下位の抽象化レイヤーとなります。

HCD は、ホストコントローラ ハードウェアの抽象化を提供するソフトウェア レイヤーで、一方、USBD は、USB デバイスの抽象化を提供し、ホストソフトウェアと USB デバイスの機能間のデータ転送を提供します。

USBD は USB ドライバインターフェイス (USBDI) を使用して、クライアントと通信を行います。より低レベルでは、コア ドライバと USB ハブ ドライバは、ホストコントローラ ドライバインターフェイス (HCDI) を使用して HCD と通信することによって、ハードウェア アクセスとデータ転送を実装します。

USB ハブ ドライバは、特定のハブに対してデバイスの着脱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の着脱のシグナルを受信すると、ホストソフトウェアと USB コア ドライバを使用してデバイスの認識と設定を行います。設定を実装するソフトウェアには、ハブ ドライバ、デバイス ドライバおよびその他のソフトウェアを含めることができます。

WinDriver USB は、上記で説明したとおり、開発者に対して設定手順とハードウェア アクセスを抽象化します。WinDriver USB の API を使用して、開発者は、これらの操作をサポートする低レベルの実装をマスターすることなく、すべてのハードウェア関連の操作を行うことができます。

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、仮想 Com ポート USB 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、カーネル レベルの実装が必要となるので、お問い合わせください。

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

- 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

4.1.2 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 / Server 2008 / 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 / 2008

4.1.3 Linux

- 32 ビットの x86 プロセッサと Linux カーネル 2.2.x (PCI 版のみ)、2.4.x または 2.6.x
または
64 ビットの x86 プロセッサ AMD64 または インテル EM64T (**x86_64**) と Linux カーネル 2.4.x または 2.6.x
または
PowerPC 32 ビット アーキテクチャと Linux カーネル 2.4.x または 2.6.x

または

PowerPC 64 ビットアーキテクチャと Linux カーネル 2.6.x

- 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`

4.2 WinDriver のインストール

WinDriver インストール CD-ROM 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 2000 / XP / Server 2003 / Server 2008 / Vista x86 32 ビットと x64 64 ビット用の WinDriver が収められています。他のオペレーティングシステム用の WinDriver 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 2000 / XP / Server 2003 / Server 2008 / 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 (`WDxxxx.EXE`、`xxxx` はバージョン番号。例: `WD1000.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、`WDxxxx.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。

7. **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。

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 [Registration Options] 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 の代わりに低レベルの `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。

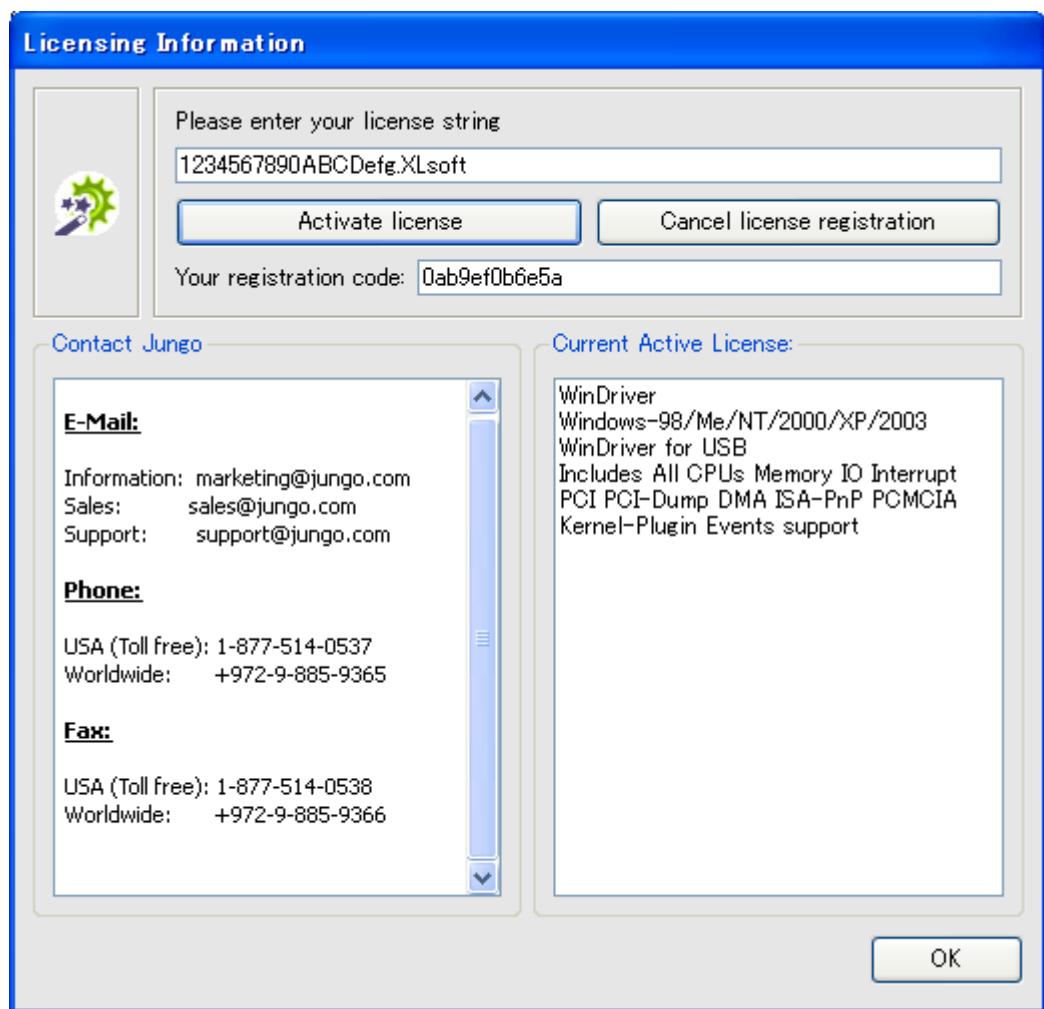


図 4.1: ライセンスの登録

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、[DriverWizard] を実行して、[File] メニューから [Registration Options] 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 のライセンスをドライバコードに登録するには

WinDriver の DriverWizard を使用しないで、ドライバコードから直接 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コード内で登録します。

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

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

    hWD = WD_Open();
```

```

if (hWD != INVALID_HANDLE_VALUE)
{
    // 以下の文字列にライセンスコードを入力します
    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 / 2008 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 CE 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、エントリを追加します。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`(たとえば、`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つて 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 / 2008 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 (`windrivr6.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 (`windrivr6.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。たとえば、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 `windrivr6.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 CE 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 (たとえば、`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 CE 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。たとえば、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、Windows CE 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 Windows CE 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 / 2008 (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。

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. Windows ホスト開発 PC の `WinDriver\redist\WINCE\<TARGET_CPU>` ディレクトリから WinDriver のカーネル モジュール `windrvr6.dll` をターゲットの Windows CE プラットフォームの `Windows\` ディレクトリにコピーします。
4. 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 / 2008 (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。

5. 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)。
6. 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 / Server 2008 / 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。

注意: WinDriver Windows 2000 / XP / Server 2003 / Server 2008 / 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 モジュールをインストールするので、インストール時に 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。

スーパー ユーザーになります:

```
$ su
```

Linux のソース ディレクトリに移動します:

```
# cd /usr/src/linux
```

以下を入力します:

```
# make xconfig
```

Save and Exit を選択して設定情報を保存します。

以下を入力します:

```
# make dep
```

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、**libstdc++** ライブラリのバージョン 5 (**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. WD1000LN.tgz ファイルを解凍します。

```
$ tar xvzf /<file location>/WD1000LN.tgz
```

例:

- CD の場合:

```
$ tar xvzf /mnt/cdrom/LINUX/WD1000LN.tgz
```

- ダウンロードファイルの場合:

```
$ tar xvzf /home/username/WD1000LN.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 --disable-usb-support

注意: **configure** スクリプトで、起動中のカーネルベースの **makefile** を作成します。

インストールした他のカーネルソースベースでも **configure** スクリプトにフラグ

--with-kernel-source=<path> を付けて、**configure** スクリプトを起動できます。

<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。

Linux カーネルバージョンが 2.6.26 またはそれ以降の場合、**configure** は、**kbuild** を使用してカーネルモジュールをコンパイルする **makefile** を生成します。以前のバージョンの Linux で、**kbuild** を強制的に使用するには、**configure** に **--enable-kbuild** フラグを渡します。

- <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 <full path to WinDriver>/WinDriver/wizard/wdwizard/
usr/bin/wdwizard
7. **wdwizard** ファイルに read (読み取り) および execute (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) および 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 666 /dev/windrvr6
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。
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: **WinDriver/util/wdreg** スクリプトを使用して、WinDriver のカーネル モジュールをロードします。システムの起動時に、自動的に WinDriver をロードするには、ターゲットの Linux のブート ファイル (**/etc/rc.d/rc.local**) に以下の行を追加して wdreg を起動します:
<path to wdreg>/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 [Registration Options] 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 の代わりに低レベルの 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。

4.3 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 / Server 2008 / 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 2000 / XP / Server 2003 / Server 2008 / 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 の代わりに低レベルの 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 および Linux 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 wdwizard.exe を実行して DriverWizard を開始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。DriverWizard が検出するのを確認します。

- 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

- コンソール モードの Debug Monitor ユーティリティ – `WinDriver\util\wddebug\<TARGET_CPU\wddebug.exe` – を Windows ホストマシンからターゲットの Windows CE デバイスのディレクトリにコピーします。
- ターゲット デバイス上で以下のように status コマンドで Debug Monitor を起動します:
`wddebug.exe status`
 WinDriver のインストールが成功している場合、アプリケーションは、Debug Monitor のバージョンと現在のステータス情報、起動している WinDriver のカーネル モジュールに関する情報、および一般的なシステム情報を表示します。

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:

- `wdreg.exe` の代わりに `wdreg_gui.exe` を使用することができます。
 - `wdreg.exe` および `wdreg_gui.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)。
- 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。
 - 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いる場合、`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。
 - INF ファイルを使用して WinDriver と動作するように登録されたすべての Plug-and-Play デバイス (USB / PCI / PCMCIA) をアンインストールします:
 - `wdreg` ユーティリティを使用してデバイスを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`
 - `%windir%\inf` ディレクトリに、WinDriver のカーネル モジュール (`windrivr6.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。
 - 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 (**windrivr6.sys**) を停止し、アンロードし、
%windir%\inf\ ディレクトリから **windrivr6.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 (**windrivr6.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:

```
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、**windrivr6.sys** と **windrivr6.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、**windrivr6.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 (**windrivr6.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 ユーティリティ (**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できます。
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. **windrivr6.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
- %windir%\system32\wdapi1000.dll
- %windir%\sysWOW64\wdapi1000.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: root 権限で以下のコマンドを実行する必要があります。

1. WinDriver のドライバ 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 のドライバ モジュールを使用しているアプリケーションとモジュールを確認します（デフォルトでは、WinDriver のモジュール名は `windrvr6` で始まります）。
 - WinDriver のドライバ 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/modprobe -r <module_name>`
2. WinDriver のモジュールをアンロードします。
`/# /sbin/modprobe -r 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 インストール 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/libwdapi1000.so` (32 ビット x86 または 32 ビット PowerPC)
`/usr/lib/64/libwdapi10000.so` (64 ビット x86)

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。

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 が拡張サポートを提供する USB または PCI チップ セット (PLX 6466、9030、9050、9052、9054、9056、9080 および 9656、Altera pci_dev_kit、Xilinx VirtexII および Virtex 5、AMCC S5933、Cypress EZ-USB ファミリ、Microchip PIC18F4550、Philips PDIUSBD12、Texas Instruments TUSB3410、TUSB3210、TUSB2136、TUSB5052、Agere USS2828、Silicon Laboratories C8051F320) をベースとしたカードのドライバを開発する場合、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 を使用して、ハードウェアの診断および Windows で対象のハードウェア用の INF ファイルを生成することができます。

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。

ハードウェアおよびドライバの開発フェーズで DriverWizard を使用すると、主な利点が 2 つあります。

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。Linux の場合、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、仮想 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 DriverWizard の **PCI Virtual Device** 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。
2. DirverWizard を起動して対象のデバイスを選択します。

[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。

注意: Windows Vista では、管理者権限で DriverWizard を起動する必要があります。

[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

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。

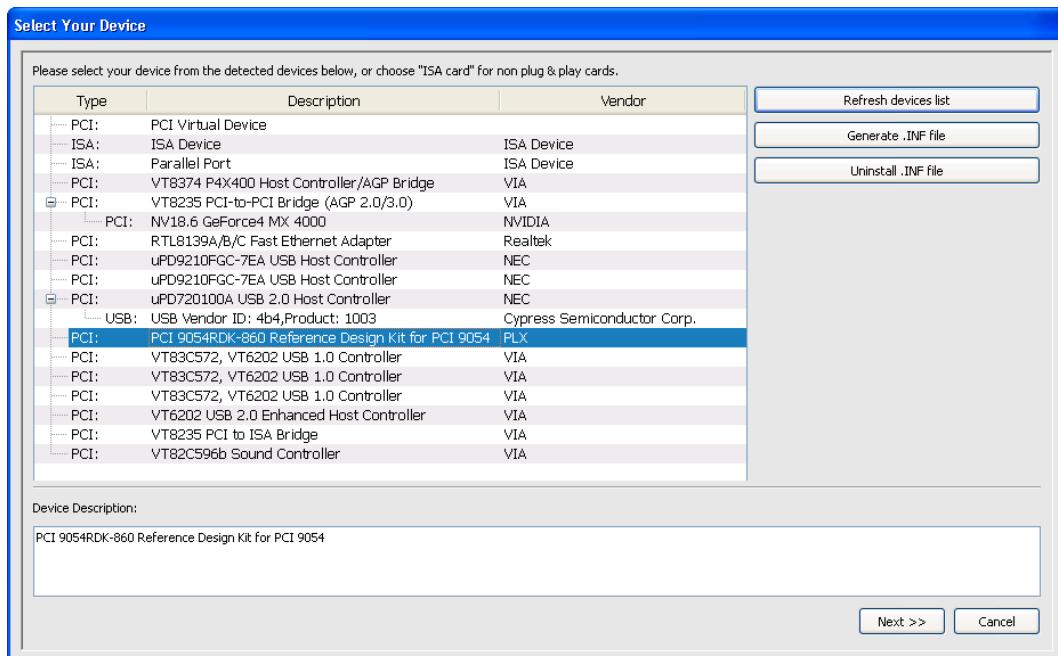


図 5.2: デバイスの選択

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 2000 / XP / Server 2003 / Server 2008 / 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、`windrivr6.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 2000 / XP / Server

2003 / Server 2008 / 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 2000 / XP / Server 2003 / Server 2008 / 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。

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

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。

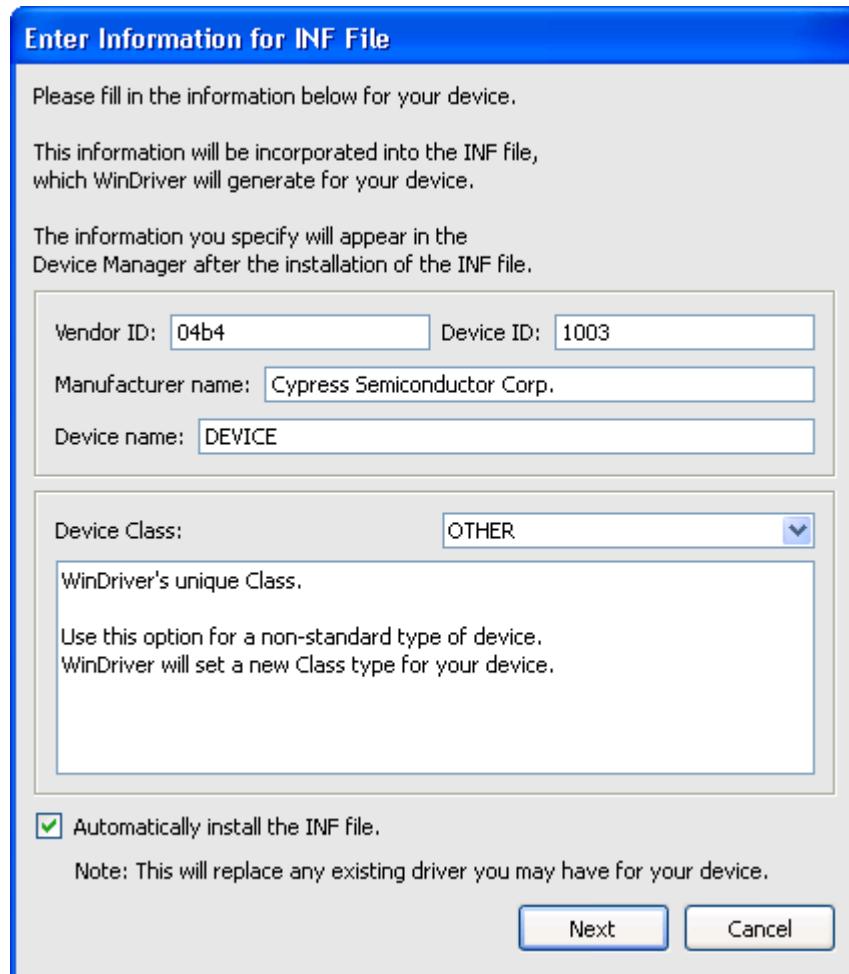


図 5.3: DriverWizard INF ファイル情報

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。

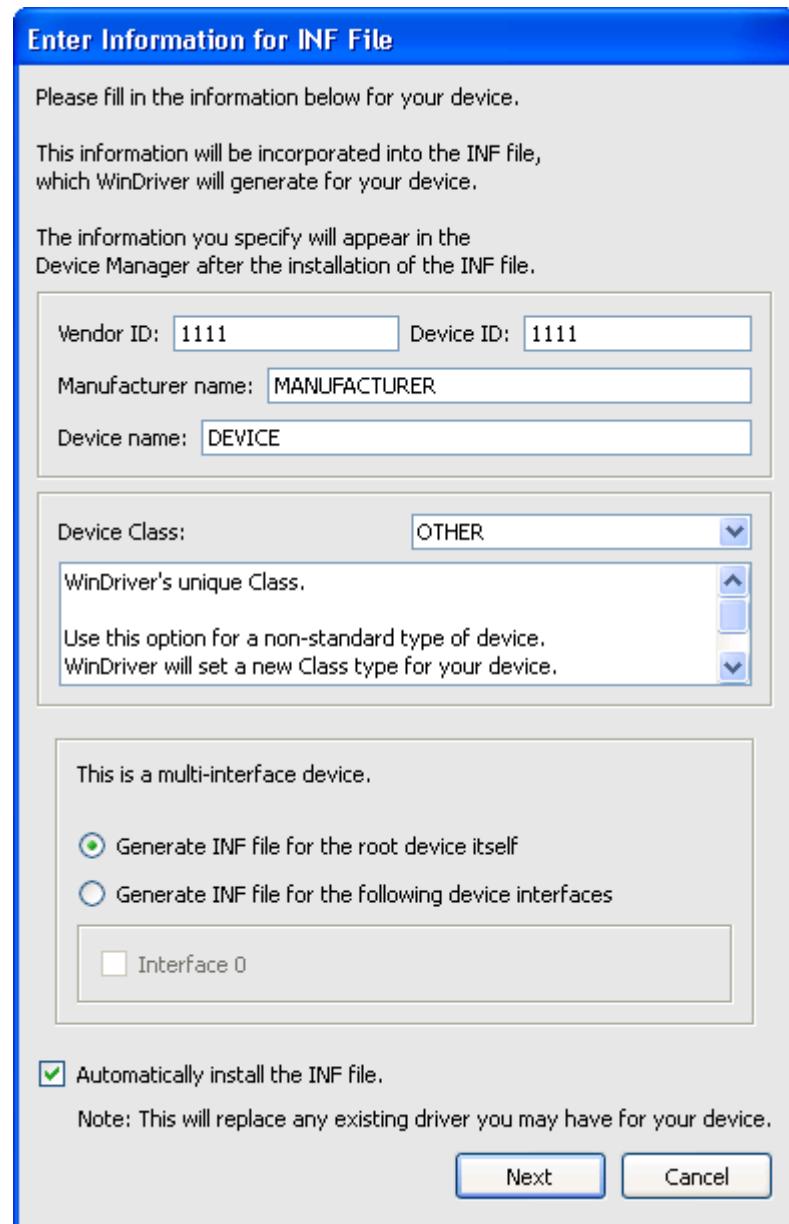
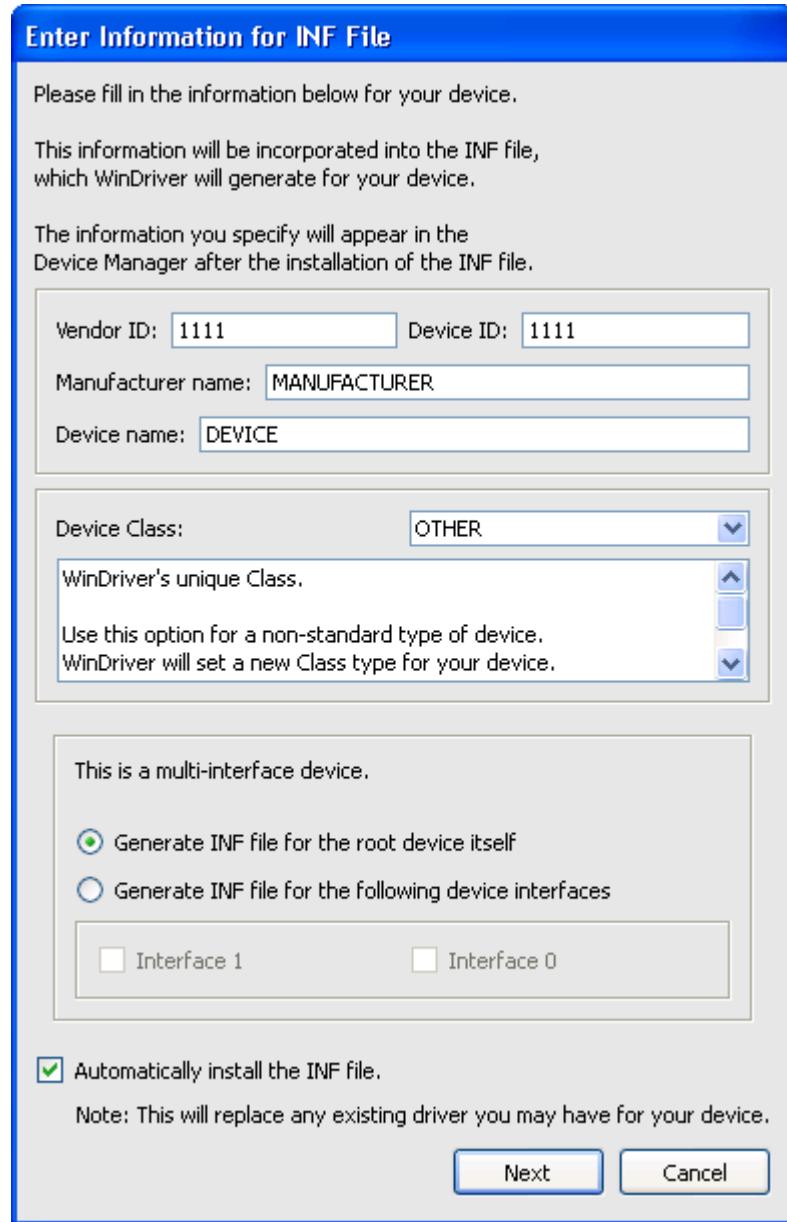


図 5.4: DriverWizard のマルチインターフェイスの INF ファイル情報
(特定のインターフェイスをそれぞれ設定する場合)

- 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)**

[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。

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フです) 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。

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。セクション 15.1 で説明します。

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。

注意: PCI MSI (Message-Signaled Interrupts) と MSI-X (Extended Message-Signaled Interrupts) の処理を行うには、セクション 9.2.6.1 の説明のとおり、デバイスの INF ファイルに特定の設定が必要です。

Windows Vista で、対象のハードウェアが MSI か MSI-X をサポートする場合、DriverWizard の INF 生成ダイアログの **Support Message Signaled Interrupts** オプションがデフォルトで有効になります。このオプションをチェックすると、対象のデバイス用に DriverWizard で生成した INF ファイルに MSI / MSI-X 処理のサポートが含まれます。ただし、このオプションのチェックを外すと(無効にすると)、対象のハードウェアと OS が MSI / MSI-X をサポートしている場合でも、PCI 割り込みをレガシーなレベル センシティブ割り込みの方法を使用して処理します。

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 **windrivr6.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。

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

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

デバイスを診断します。

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。

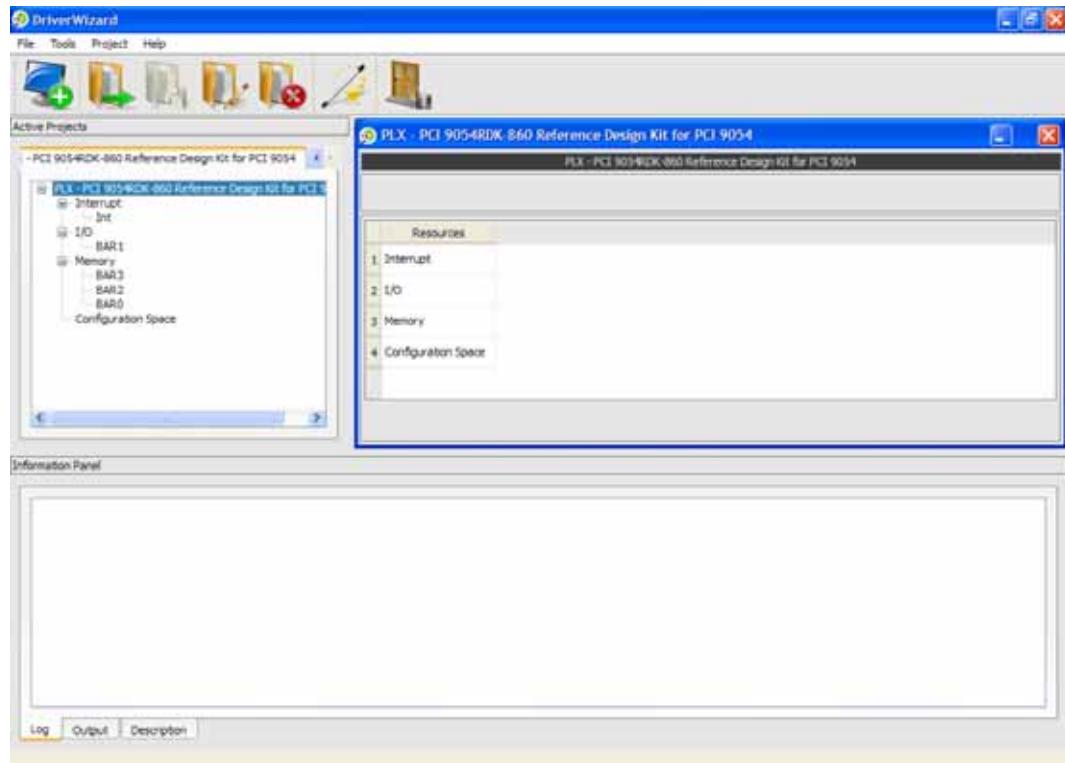


図 5.7: PCI のリソース画面

レジスタを手動で定義します。

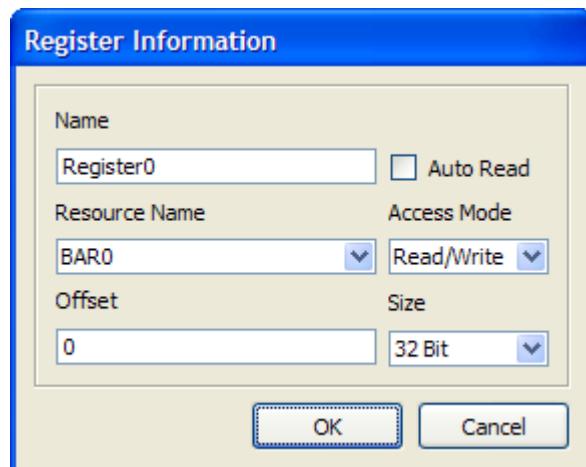


図 5.8: レジスタの定義

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、メモリースペース、定義したレジスタへの読み込みと書き込みをします。

注意: メモリマップ範囲にアクセスする際に、Linux Power PC でメモリストレージをビッグエンディアンを使用して処理する場合、リトルエンディアンを使用する PCI バスとは反対になるので注意してください。詳細は、セクション 9.7「バイトオーダー」を参照してください。



図 5.9: メモリおよび I/O の Read / Write

- ハードウェアの割り込みを ‘Listen’ (確認) します。

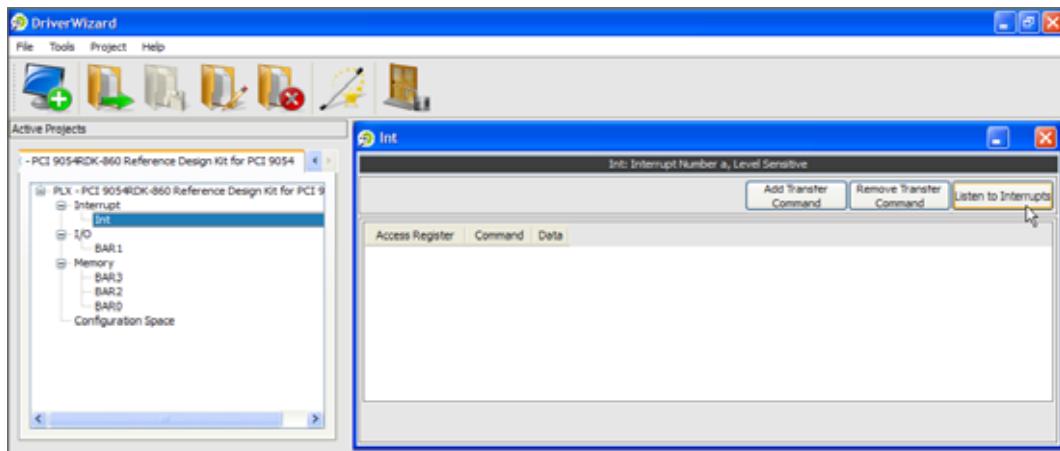


図 5.10: 割り込みの Listen (確認)

注意: レガシーな 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。以下、図 5.11 で、定義済みの INTCSR ハードウェア レジスタ用の割り込み確認コマンドを定義する方法を紹介します。ただし、割り込みの確認情報はハードウェア独自となります。

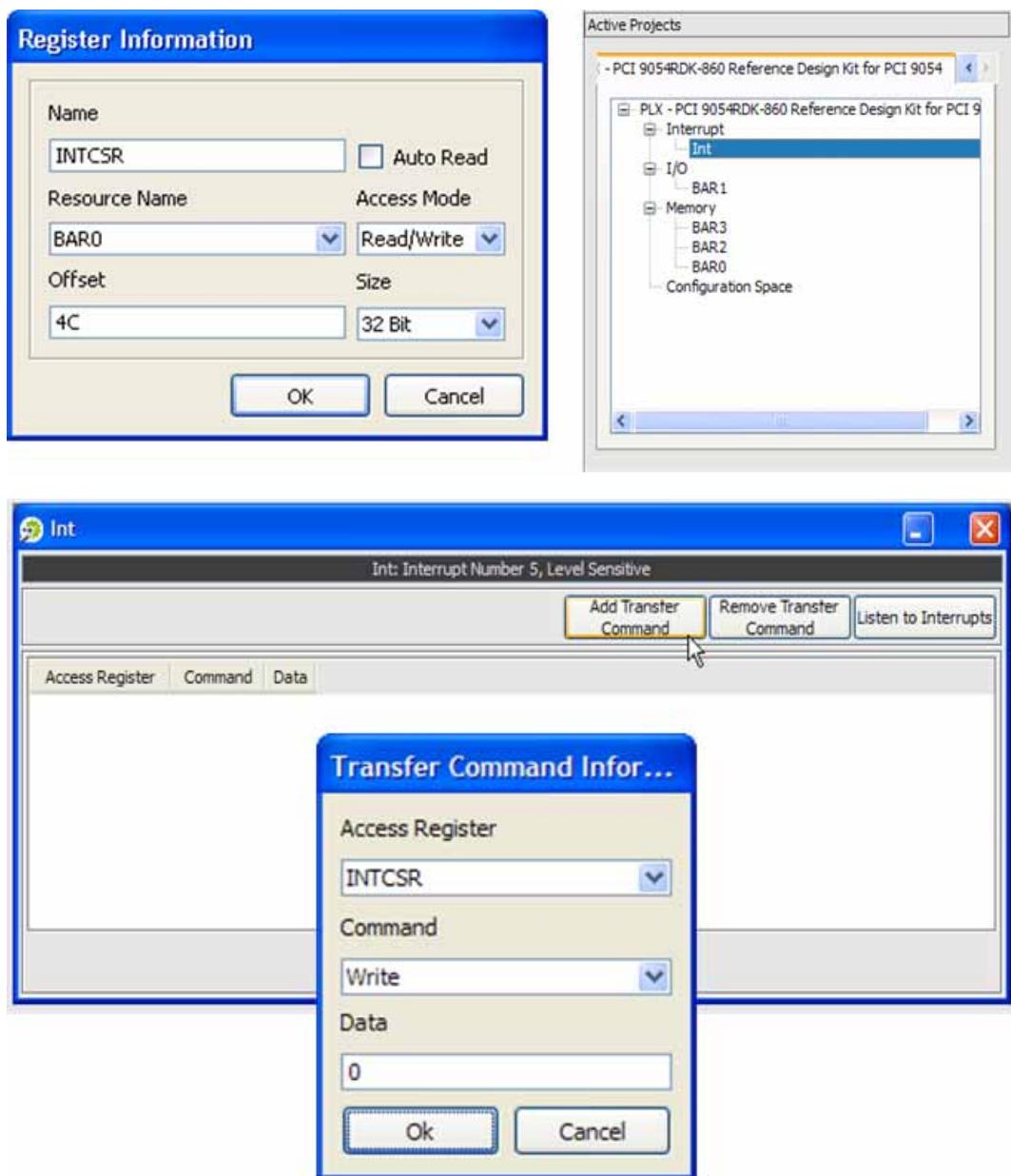


図 5.11: レベル センシティブな割り込みの転送コマンドの定義

USB デバイスの場合

USB デバイスにおける代替設定を選択します。

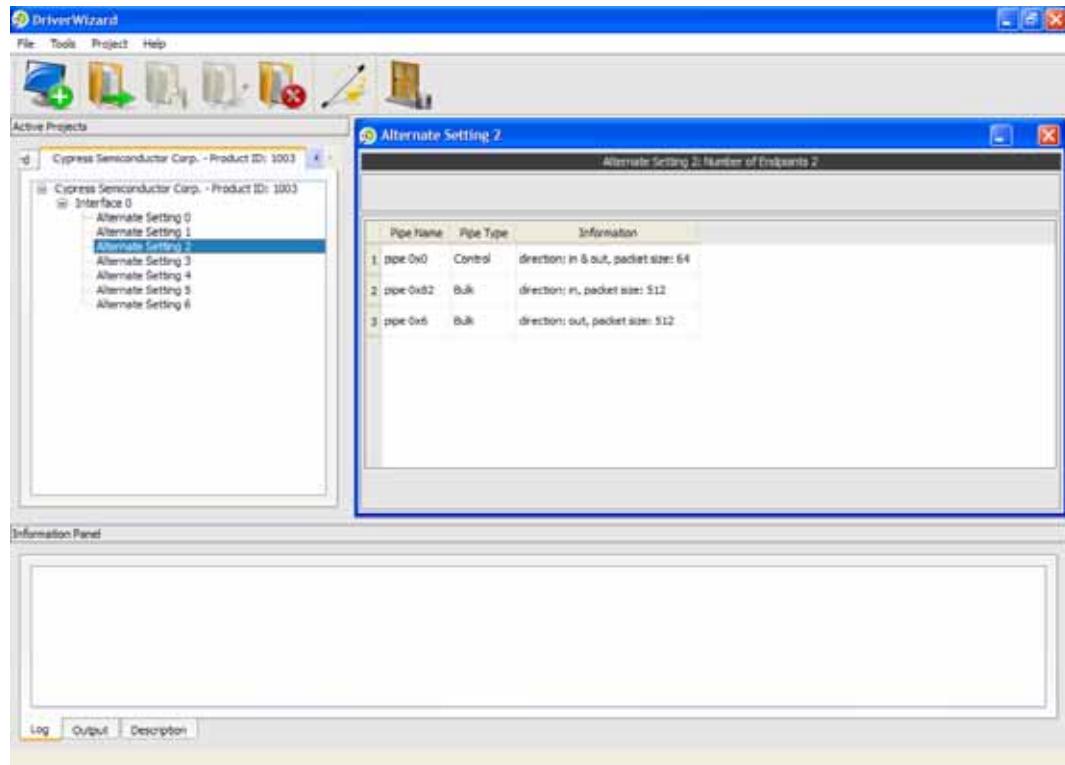


図 5.12: USB デバイスのインターフェースの選択

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 1 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)。

USB デバイス のパイプを検証します。

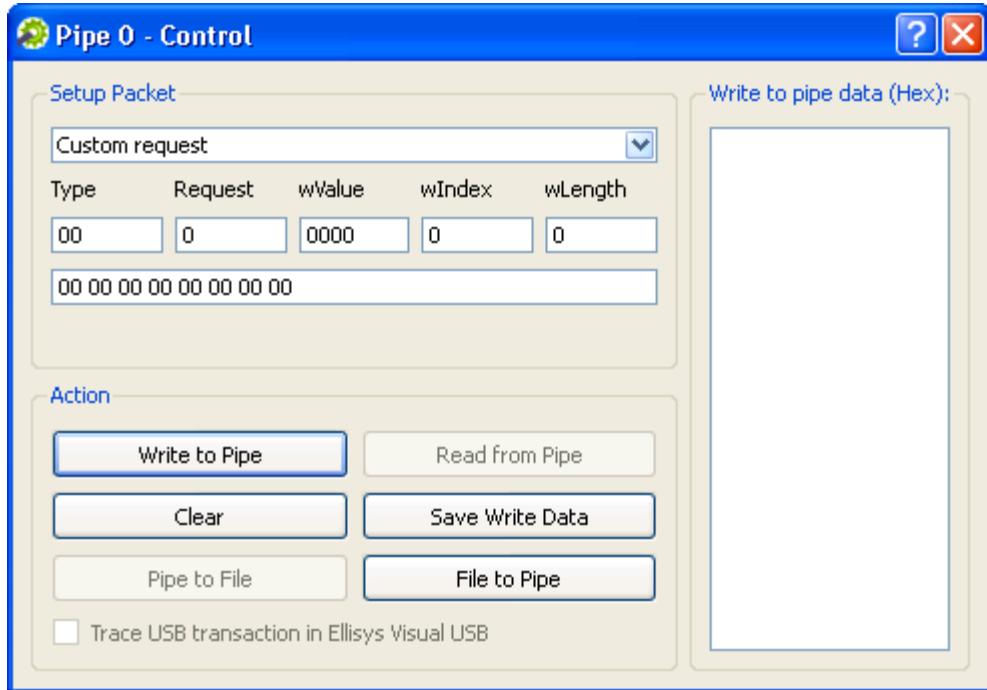


図 5.13: USB コントロール転送

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。

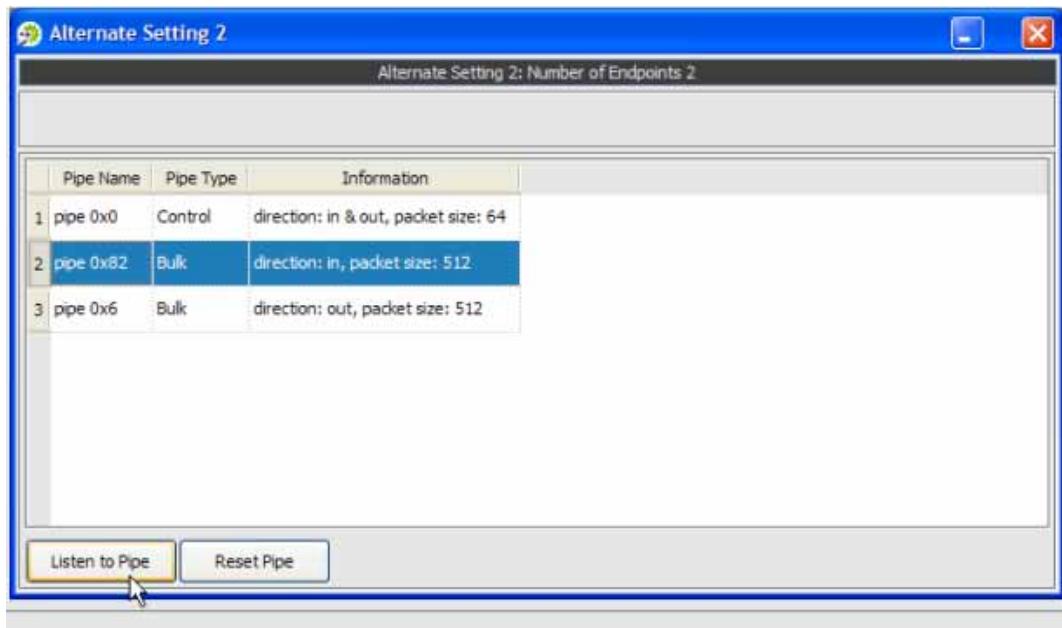


図 5.14: パイプの確認

- 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。

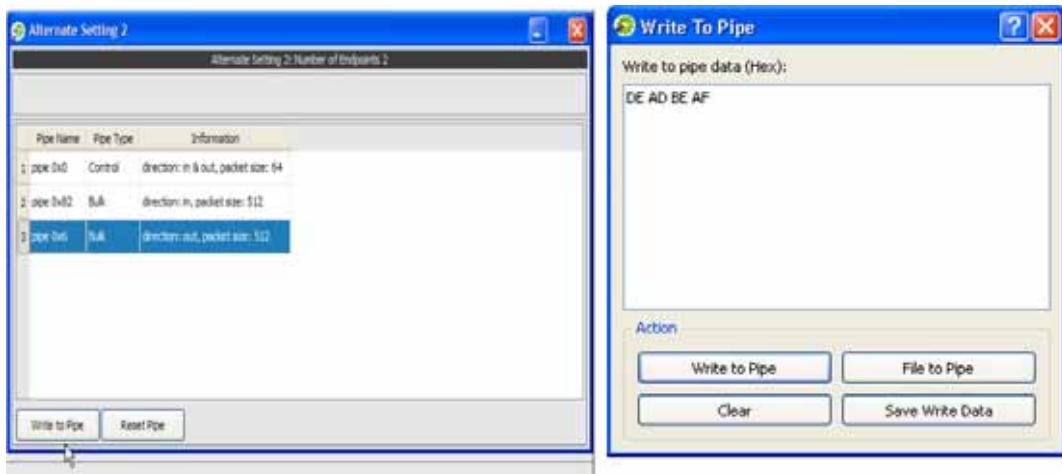


図 5.15: パイプへの書き込み

- 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. 離型となるドライバ コードを生成します。

[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。

[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。

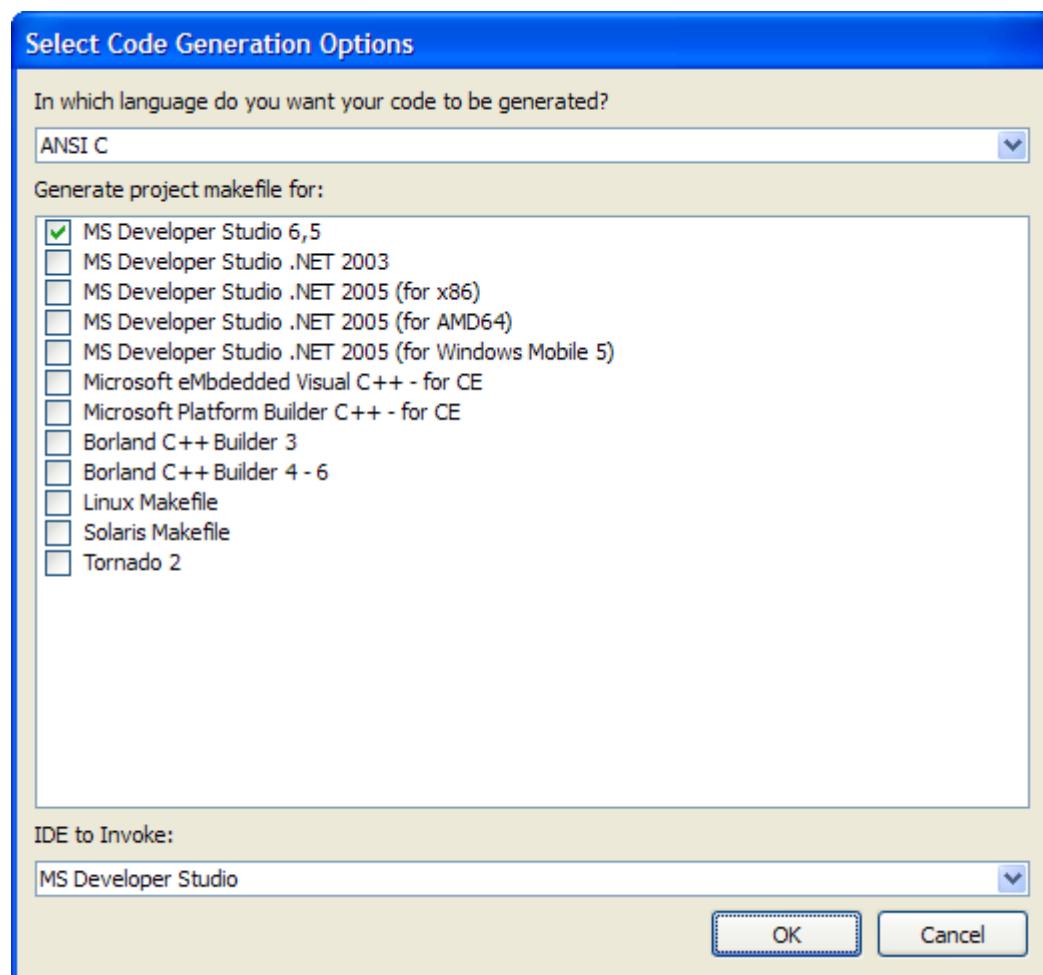


図 5.16: コード生成のオプション

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。

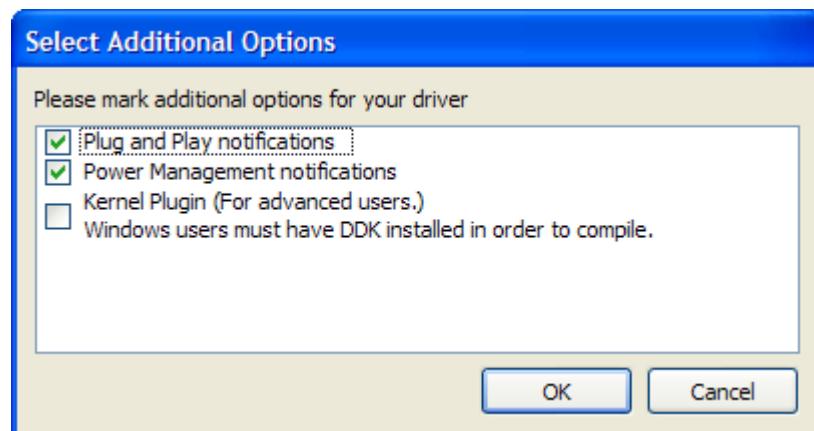


図 5.17: ドライバ オプションの選択

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 WDK (Windows Driver Kit) をインストールする必要があります。

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。

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 ドライバの雰囲として使用します。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に対応するすべてのプラットフォームで動作します。

5.2.1 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。

5.2.2 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空の 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。

5.2.3 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。

5.2.3.1 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。

5.2.3.2 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選択した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。

5.2.3.3 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 (デバイスの挿入や取り外しなど) 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。

5.2.3.4 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 5.2.3 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。

5.2.3.5 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。

5.2.4 生成されたコードをコンパイルする

5.2.4.1 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 / 2008、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 / Server 2008 / 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。

5.2.4.2 Linux 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。

5.2.5 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。

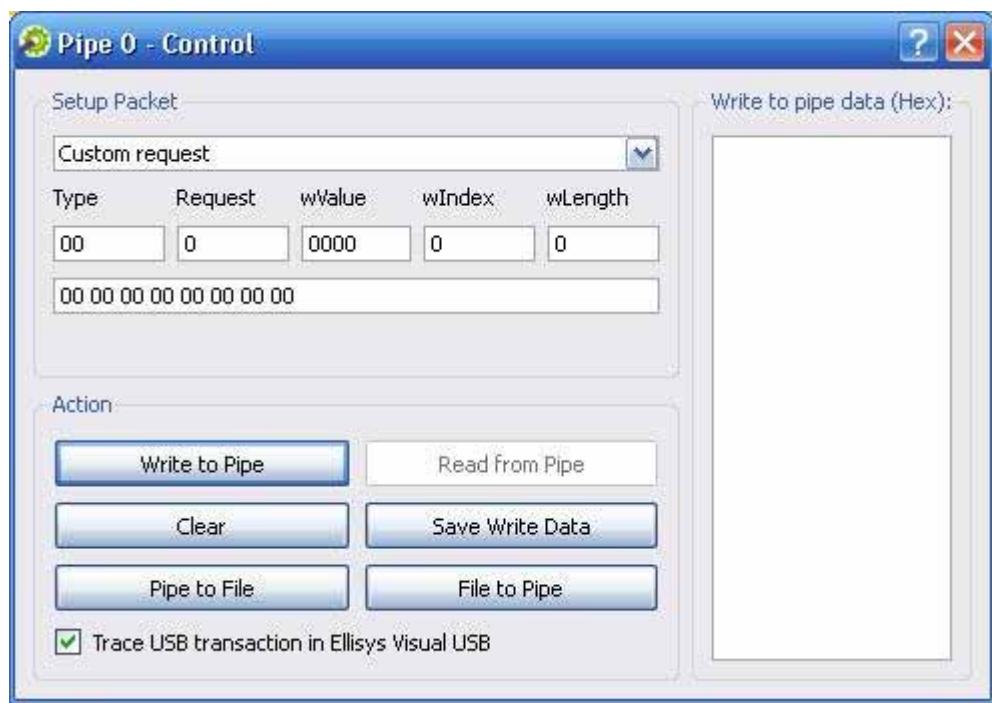


図 5.18: Ellisys Visual USB の統合

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 6466、9030、9050、9052、9054、9056、9080 および 9656、Altera pci_dev_kit、Xilinx VirtexII および Virtex 5、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 6466、9030、9050、9052、9054、9056、9080 および 9656、Altera pci_dev_kit、Xilinx VirtexII および Virtex 5、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 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.h` 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 `wdapi1000` ライブラリまたは共有オブジェクトにリンクします。

- Windows 2000 / XP / Server 2003 / Server 2008 / Vista の場合:

`WinDriver\lib\<CPU>\wdapi1000.lib` または `wdapi1000_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>\wdapi1000.lib`

- Linux の場合: `WinDriver/lib/libwdapi1000.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: `wdapi1000` ライブライ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 `wdapi1000` 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 `wdapi1000.dll` または `wdapi1000_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 では、`WinDriver/lib/libwdapi1000.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 ライブライ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 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 ライブライ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

- 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。
- attach callback を受信すると、WDU_Transfer() 関数の 1 つを使用して、データの送受信ができます。
- 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 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_Init() 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 (`windrivr6.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 – `wddebug_gui`

Windows 2000 / XP / Server 2003 / Server 2008 / Vista および Linux 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 / Server 2008 / Vista プラットフォームで 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、コンソール モードで Debug Monitor を使用してください [7.2.2]。

1. 以下のいずれかの方法で Debug Monitor を起動します
 - `WinDriver/util/wddebug_gui` を起動します。
 - DriverWizard の [Tool] メニューから Debug Monitor を起動します。

- [スタート] メニューから [プログラム] - [WinDriver] - [Debug Monitor] を選択して、Debug Monitor を起動します。



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。

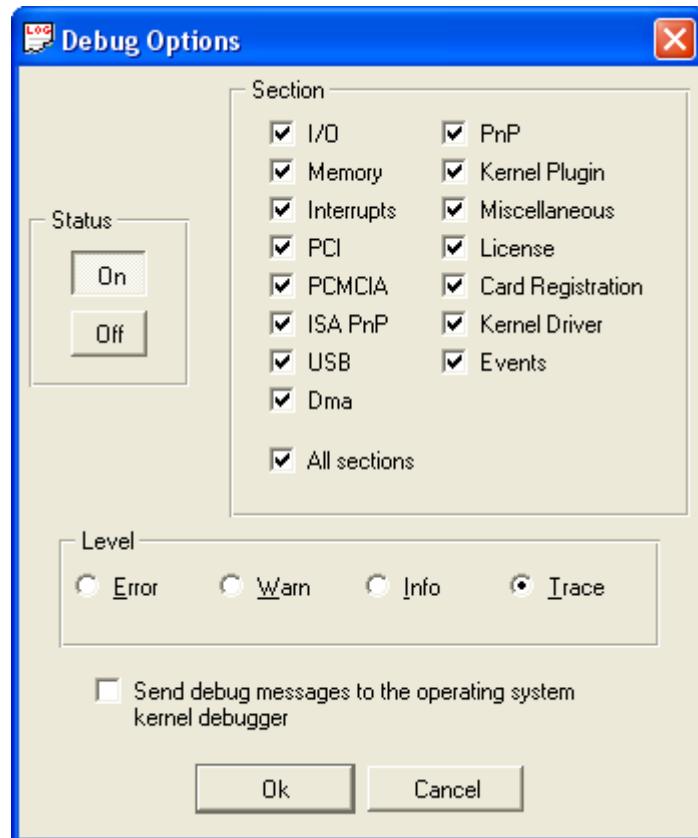


図 7.2: Debug Options の設定

- **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。

- WinDriver のカーネル モジュールから受信したデバッグ メッセージを外部のカーネル デバッガに送る場合、[Send debug messages to the operating system kernel debugger] チェック ボックスを選択します。

注意: Windows Vista では、最初にこのオプションを有効にする際に、PC を再起動する必要があります。

ヒント: 無償の Windows のカーネル デバッガーとして、WinDbg があります。WinDbg は WDK (Windows Driver Kit) および Debugging Tools for Windows package の一部として提供され、Microsoft の Web サイトから入手できます。

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。

7.2.1.1 名前変更したドライバ用にグラフィック モードで Debug Monitor を起動するには

デフォルトでは、グラフィック モードの Debug Monitor プログラム – `wddebug_gui` – は、`windrivr6.sys/.o/.ko` ドライバからのメッセージを出力しますが、`wddebug_gui` を使用して、以下のようにコマンド ラインで `driver_name` オプションを付けて、`wddebug_gui` を起動して、名前変更したドライバからのデバッグ メッセージも出力することができます (windrvr6 ドライバ モジュールの名前変更に関しては、セクション 15.2 を参照してください):

```
Wddebug_gui <driver_name>
```

注意: `driver_name` には、ファイルの拡張子なしでドライバ ファイル名を指定します。たとえば、`windrivr6.sys` (Windows の場合) や `windrivr6.o` (Linux の場合) ではなく `windrivr6` を指定します。

たとえば、Windows でデフォルトの `windrivr6.sys` ドライバを `my_driver.sys` に名前を変更した場合、以下のコマンドを使用して、Debug Monitor を起動して、ドライバからのログ メッセージを出力できます:

```
Wddebug_gui my_driver
```

7.2.2 コンソール モードで Debug Monitor を使用するには - wddebug

コンソール モードの Debug Monitor (`WinDriver/util/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。

Windows 2000 / XP / Server 2003 / Server 2008 / Vista、Windows CE および Linux で、`wddebug` (コンソール モード Debug Monitor) を使用するには、`WinDriver/util/wddebug` ユーティリティを起動します。詳細は下記のとおりです。

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`) を起動し、このシェル内でプログラム `WDDEBUG.EXE` を実行します。

WDDEBUG の使用方法

```
wddebug [<driver_name>] <command> [<level>] [<sections>]
```

注意: `wddebug` コマンド オプション は上記の手順どおりに実行してください。

<driver_name>:

コマンドに使用する対象のドライバ名。

ドライバ名は `windrivr6` (デフォルト)、または `windrivr6` ドライバ モジュールの名前を変更したドライバ名を設定します (詳細は、セクション 12.2 を参照してください)。

注意: ドライバ名には、ファイルの拡張子なしでドライバ ファイル名を指定します。たとえば、`windrivr6.sys` (Windows の場合) や `windrivr6.o` (Linux の場合) ではなく `windrivr6` を指定します。

<command>:

実行する Debug Monitor のコマンド:

- コマンドの実行:

- **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 (オンになっていない場合)。
- 注意:** Windows Vista では、最初にこのオプションを有効にする際に、PC を再起動する必要があります。
- **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。

注意: 下記の説明のとおり、`on` と `dbg_on` コマンドを `level` および (または) `sections` オプションと一緒に実行できます。

- **dump:** ユーザーが Esc キーを押下するまで、デバッグ情報の表示 (“dump”) し続行します。
- **status:** 起動中の `<driver_name>` で指定したカーネル モジュールに関する情報 (アクティブなデバッグ `level` と `sections` (Debug Monitor がオンの場合) を含む) とデバッグ メッセージのバッファ サイズを表示します。

注意: 上記で説明したとおり、Debug Monitor が `on` と `dbg_on` の場合にのみ、以下のオプションを使用できます。

<level>:

デバッグするトレース レベルです。次のフラグのいずれかを `level` に設定可能です: `ERROR`、`WARN`、`INFO` または `TRACE`。`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。デフォルトのトレース レベルは `ERROR` です。

<sections>:

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。すべてのサポートするデバッグ セクションのリストに関しては、引数なしで `wddebug` を起動し、使用方法を参照します。デフォルトのデバッグ セクションのフラグは `ALL` です (サポートするすべてのデバッグ セクションを設定)。

使用手順

`wddebug` を使用して、メッセージを出力する場合、以下の手順を実行します:

- `on` コマンドまたは `dbg_on` コマンドのいずれかで `wddebug` を起動して、Debug Monitor をオンにします – Debug Monitor をオンにする前にデバッグ メッセージをカーネル デバッガへ転送します。
`level` および (または) `sections` フラグを使用して、出力用にデバッグ レベルおよび (または) セクションを設定できます。これらのオプションを指定しない場合、デフォルトの値を使用します。
 ドライバ名を指定してコマンドを実行して、名前を変更した WinDriver のドライバのメッセージを出力できます (上記の `<driver_name>` オプションを参照してください)。デフォルトで監視するドライバは `windrvr6` です。
- `dump` コマンドで `wddebug` を起動し、コマンド プロンプトへデバッグ メッセージのダンプを開始します。
 コマンド プロンプトで `Esc` を押下して、デバッグ メッセージの出力をオフにできます。
- ドライバを使用するアプリケーションを起動して、コマンド プロンプトまたはカーネル デバッガへデバッグ メッセージを出力する際に表示します。
- Debug Monitor がオンの状態で、`status` コマンドで `wddebug` を起動して、現在のデバッグ レベルとセクション参照でき、また同様に起動中の `<driver_name>` のカーネル モジュールに関する情報も参照できます。
- Debug Monitor がオンの状態で、`dbg_on` と `dbg_off` を使用して、カーネル デバッガへのデバッグ メッセージの転送を切り替えできます。
- `off` コマンドで `wddebug` を起動して、Debug Monitor をオフにします。

注意: Debug Monitor がオフの状態で、`status` コマンドで `wddebug` を起動して、起動中の `<driver_name>` のカーネル モジュールに関する情報を参照することもできます。

例

以下は一般的な `wddebug` の使用手順の例です。`<driver_name>` を指定していないので、以下のコマンドでは、デフォルトのドライバ (`windriver6`) を使用します。

- Debug Monitor をオンにし、すべてのセクションすべてのメッセージを表示するトレース レベルを指定します:
`wddebug on TRACE ALL`

注意: `ALL` はデフォルトのデバッグ セクションのオプションなので、”`wddebug on TRACE`” を実行するのと同じです。

- Esc を押下するまで、デバッグ メッセージをダンプします:
`wddebug dump`
- ドライバを使用して、コマンド プロンプトでデバッグ メッセージを表示します。
- 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 6466、9030、9050、9052、9054、9056、9080 と 9656、Altera pci_dev_kit、Xilinx VirtexII と Virtex 5、AMCC S5933、Cypress EZ-USB ファミリ、Microchip PIC18F4550、Philips PDIUSBD12、Texas Instruments TUSB3410、TUSB3210、TUSB2136 と 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. 対象のデバイスのサンプル診断プログラムは `WinDriver/chip_vendor/chip_name/` ディレクトリにあります。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、バスマスターとして実行されるデバイスのためのバスマスター 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)。低レベル WD_DMAxxx API は WinDriver PCI 低レベル 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ていない場合)。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 (有効な場合)。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 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_DMAMxxx` API を使用したサンプルは `WinDriver/altera/pci_dev_kit/lib/altera_lib.c` ライブ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 必要な実装

上記のサンプルコードで、対象のデバイスの仕様に応じて、以下の MyDMAxxx() ルーチンを実装します。

- 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(): デバイスのレジスタへ書き込みを行って、DMA 転送を開始します。
- MyDMAInterruptEnable() と MyDMAInterruptDisable(): WDC_IntEnable() と
 WDC_IntDisable() を使用して、ソフトウェアの割り込みを有効または無効にし、デバイスの関
 連するレジスタを書き込みまたは読み込みをして、物理的にハードウェアの DMA 割り込みを有効
 または無効にします。(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 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>) で説明している通り、`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、バスマスター 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_DMAMxxx` API を使用したサンプルは `WinDriver/amcc/lib/amccplib.c` ライブ ラリ ファイルに保存されています。

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 必要な実装

上記のサンプルコードで、対象のデバイスの仕様に応じて、以下の MyDMAxxx() ルーチンを実装します。

- 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(): デバイスへ書き込みをして、DMA 転送を開始します。
- MyDMAInterruptEnable() と MyDMAInterruptDisable(): WDC_IntEnable() と
WDC_IntDisable() を使用して、ソフトウェアの割り込みを有効または無効にし、デバイスの関
連するレジスタを書き込みまたは読み込みをして、物理的にハードウェアの DMA 割り込みを有効
または無効にします。(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 (およびイメージから) データを転送します。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 API、DriverWizard のコード生成およびサンプルを提供して、対象のデバイスからの割り込みを処理するタスクを簡素化します。

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) を検出および定義し、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 WinDriver の API の使用方法を実例と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。

9.2.1 割り込み処理の概要

PCI、PCMCIA および ISA ハードウェアは割り込みを使用してホストに信号を送ります。PCI の割り込み処理には代表的な 2 つの方法があります。

レガシー割り込み:

ラインベース メカニズムを使用した従来の割り込み処理です。この方法では、割り込みは "out-of-band" つまり、メインのバス ラインから別々に接続される複数の外部ピンを使用して、信号を受けます。

レガシー割り込みには 2 つのグループがあります:

- **レベル センシティブな割り込み:** 物理的な割り込み信号が High である限りこの割り込みを生成します。割り込み信号がカーネルで割り込み処理の終了までに Low にならない場合、OS が繰り返しカーネルの割込みハンドラを呼ぶので、ホスト プラットフォームがハングします。この状態が起きるのを防ぐには、割り込みを WinDriver カーネル割り込みハンドラによって認識させる必要があります。
- レガシー PCI 割り込みはレベル センシティブです。
- **エッジトリガ割り込み:** 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。
- ISA/EISA 割り込みはエッジトリガです。

MSI / MSI-X:

PCI バス v2.2 以降と PCI Express で利用可能な、新しい PCI バス技術は MSI (Message-Signaled Interrupts) をサポートしています。この方法はピンの代わりに "in-band" メッセージを使い、ホストブリッジのアドレスをターゲットにすることができます。PCI 機能は 32 MSI メッセージまで要求することができます。

注意: MSI と MSI-X はエッジトリガで、カーネルでの確認は必要ありません。

MSI には以下の利点があります:

- MSI は割り込みメッセージと一緒にデータを送信することができます。
- レガシー PCI 割り込みとは対照的に、MSI は共有されません。つまり、デバイスに割り当てられる MSI はシステム内でユニークになるように保証されます。

MSI-X (Extended Message-Signaled Interrupts) は PCI バスの v3.0 以降で利用可能で、この方法は MSI メカニズムの拡張バージョンを提供します。次の利点があります:

- 標準の MSI がサポートする 32 メッセージではなく、2,048 メッセージをサポートします。
- 各メッセージにおいて、独立したメッセージアドレスとメッセージデータをサポートします。
- メッセージごとにマスクをサポートします。
- ソフトウェアがハードウェアより少ない要求をする時、より柔軟に対応します。ソフトウェアは複数の MSI-X スロットで、同じ MSI-X アドレスとデータを再利用することができます。

MSI / MSI-X をサポートする新しい PCI バスは、帯域内メカニズムでレガシー割り込みをエミュレートすることによって、ソフトウェアとレガシー ラインベースの割り込みメカニズムの互換性を維持します。これらのエミュレートした割り込みをホスト OS はレガシー割り込みとして扱います。

WinDriver は、サポートするすべての OS 上で (Windows、Windows CE および Linux)、レガシー ラインベースの割り込み (エッジトリガ割り込みとレベルセンシティブ割り込みの両方) をサポートします (Windows CE に関しては、詳細はセクション 9.2.8 を参照してください)。

WinDriver は、セクション 9.2.6 の説明のとおり、Linux および Windows Vista 上で PCI MSI / MSI-X 割り込み (ハードウェアによりサポートされている場合) のサポートも行っています (以前の Windows バージョンでは PCI MSI / MSI-X をサポートしてません)。

WinDriver はレガシー割り込みと MSI / MSI-X 割り込みの両方を処理する API のセットを提供します。

9.2.2 WinDriver の割り込み処理手順

注意: このセクションでは、WinDriver を使用したユーザー 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、重大なタスクのため、カーネルで割り込みを直接処理することが求められます。WinDriver の Kernel PlugIn [第 11 章] を使用して、カーネルモードの割り込みルーチンを実装します。Kernel PlugIn から割り込み処理を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。

WinDriver を使用した割り込み処理は以下の手順で行います:

1. ユーザー アプリケーションで WinDriver の割り込みを有効にする関数の 1 つを呼び出して (WDC_IntEnable()、低レベル InterruptEnable() または WD_IntEnable() 関数)、デバイス上で割り込みを有効にします。

割り込みが発生すると、これらの関数は、カーネルで実行される読み込みまたは書き込み転送コマンドのオプションの配列を受信します。

注意:

- WinDriver を使用してレベルセンシティブな割り込みを処理する場合、セクション 9.2.5 の説明のとおり、割り込みを認識するために転送コマンドを設定する必要があります。
- 割り込みが無効になるまで、転送コマンド用に割り当てたメモリを利用可能にしつく必要があります。

WDC_IntEnable() または低レベル InterruptEnable() 関数を呼んだ場合、WinDriver はスレッドを生成して、受信割り込みを処理します。低レベル InterruptEnable() 関数を使用した場合、自分自身でスレッドを生成する必要があります。

注意: 割り込みを有効にする前に、WinDriver をデバイスのドライバとして OS に登録する必要があります。Windows プラットフォームの Plug-and-Play ハードウェア (PCI / PCI Express / PCMCIA) の場合、デバイスの INF ファイルをインストールすることでこのアクションを実行します。INF ファイルがインストールされていない場合、割り込みを有効にする関数は、WD_NO_DEVICE_OBJECT エラーで失敗します。

2. そのスレッドは無限ループを起動して、割り込みが発生するのを待機します。
 3. 割り込みが発生した場合、WinDriver は、カーネルで、ユーザーが事前に準備し、WinDriver の割り込みを有効にする関数へ渡した転送コマンドを実行します。詳細はセクション 9.2.5 を参照してください。
- コントロールがユーザー モードへ戻ると、ドライバのユーザー モードの割り込みハンドラー ルーチン (WDC_IntEnable() または InterruptEnable() で割り込みを有効にする際に WinDriver へ渡されます) が呼ばれます。
4. ユーザー モードの割り込みハンドラーがリターンを返すと、待機ループが継続します。
 5. 割り込みを処理する必要がない場合、またはユーザー モード アプリケーションが終了する場合、関連する WinDriver の割り込みを無効にする関数を呼んでください – WDC_IntDisable()、低レベル interruptDisable() または WD_IntDisable() 関数。

注意:

- 低レベル WD_IntWait() WinDriver 関数を高レベルな割り込みを有効にする関数で使用してデバイスからの割り込みを待機し、低レベル WD_IntWait() WinDriver 関数は、割り込みが発生するまでスレッドをスリープにします。
- ユーザー モードで割り込みハンドラーを起動するので、この関数からファイル処理や GDI 関数を含む OS API を呼びだすことができます。

9.2.3 ハードウェアがサポートする割り込みタイプの決定

WDC_PciGetDeviceInfo() (PCI)、WDC_PcmciaGetDeviceInfo() (PCMCIA)、低レベルの WD_PciGetCardInfo() または WD_PcmciaGetCardInfo() 関数を使用して、Plug-and-Play デバイスのリソース情報を取得する場合、関数はハードウェアがサポートする割り込みタイプの情報を返します。返ってきた割り込みリソース (WDC 関数の pDeviceInfo->Card.Item[i].I.Int.dwOptions または、低レベル関数の pPciCard->Card.Item[i].I.Int.dwOptions) の dwOptions フィールド内にこの情報を返します。割り込みオプションのビットマスクには、以下の割り込みタイプのフラグのいずれかの組み合わせが含まれます:

- **INTERRUPT_MESSAGE_X**: MSI-X (Extended Message-Signaled Interrupt)
- **INTERRUPT_MESSAGE_X**: MSI (Message-Signaled Interrupt)
- **INTERRUPT_LEVEL_SENSITIVE**: レガシー レベル センシティブ割り込み
- **INTERRUPT_LATCHED**: レガシー エッジトリガ割り込み
このフラグの値は 0 で、他の割り込みフラグが設定されない場合のみ有効です。

注意:

- **INTERRUPT_MESSAGE** および **INTERRUPT_MESSAGE_X** フラグは PCI デバイスにのみ有効です [9.2.6]。
- **Windows API** は MSI と MSI-X を区別しません。そのため、この OS では、WinDriver の関数は MSI と MSI-X の両方に **INTERRUPT_MESSAGE** フラグをセットします。

9.2.4 PCI カードの割り込みタイプの決定

Linux または Windows Vista で PCI カードの割り込みを有効にする場合、カードがサポートしている場合、WinDriver は最初に MSI-X または MSI を使用します。失敗した場合、WinDriver はレガシー レベル センシティブ割り込みを有効にします。

WinDriver の割り込みを有効にする関数は、カードに対して有効にする割り込みタイプについての情報を返します。この情報は、関数に渡された **WD_INTERRUPT** 構造体の **dwEnabledIntType** フィールド内に返されます。高レベルな **WDC_IntEnable()** 関数を使用する場合、この情報は、関数の **hDev** パラメータによって参照される **WDC デバイス** 構造体の **Int** フィールド内に保存され、**WDC_GET_ENABLED_INT_TYPE** 低レベル WDC マクロを使用して取得できます。

9.2.5 カーネルモードの割り込み転送コマンドの設定方法

割り込みを処理する場合、割り込み発生時に即座にカーネルモード レベルで優先度の高いタスクを実行する必要があります。たとえば、レベル センシティブ割り込みを処理する場合、レガシー PCI 割り込みなど、カーネルで割り込みラインを下げる必要があります (つまり、割り込みを確認する必要があります)、そのように処理しない場合、OS は WinDriver のカーネル割り込みハンドラーを繰り返し呼び出し、ホスト プラットフォームがハングします。割り込みの確認は、ハードウェア独自であり、デバイスの特定の run-time レジスターから書き込みまたは読み込みを実行します。

WinDriver の割り込み有効関数は、**WD_TRANSFER** 構造体の配列へのオプションのポインタを受信します。デバイスのメモリまたは I/O アドレスから読み込み転送コマンドを設定、またはデバイスのメモリまたは I/O アドレスへ書き込み転送コマンドを設定するのにこの構造体を使用できます。

WDC_IntEnable() 関数は、直接パラメータとして配列のこのポインタとコマンドの数を取得します (**pTransCmds** と **dwNumCmds**)。

低レベル **InterruptEnable()** および **WDC_IntEnable()** 関数は **WD_INTERRUPT** 構造体の **Cmd** と **dwCmds** フィールド内でこの情報を受信します。

割り込み受信したら、対象のデバイスへ、またはデバイスからパフォーマンスに関する転送を実行する必要がある場合 (たとえば、レベル センシティブ割り込みを処理する場合)、割り込みを受信した際に、カーネルで実行する読み込み操作または書き込み操作に関して必要な情報を含む **WD_TRANSFER** 構造体の配列を用意し、WinDriver の割り込み有効関数へこの配列を渡します。セクション 9.2.2 (3) で説明したとおり、WinDriver のカーネルモードの割り込みハンドラーは、ユーザーモードへコントロールを返す前に、割り込み有効関数が処理する各割り込みに対して割り込み有効関数内で割り込みハンドラーへ渡された転送コマン

ドを実行します。注意: 割り込みが無効になるまで、転送コマンド用に割り当てたメモリを利用可能にしとく必要があります。

9.2.5.1 割り込みマスク コマンド

WinDriver へ渡す割り込み転送コマンド配列には、割り込みマスク構造体も含まれており、割り込みのソースを確認するのに使用します。CMD_MASK に転送構造体の cmdTrans フィールド(転送コマンドのタイプを定義する)を設定および転送構造体の Data フィールドに関連するマスクを設定することで実行できます。転送コマンド配列の転送コマンドを読み込んだ後に、割り込みマスク コマンドを直接設定する必要があるので注意してください。

マスク割り込みコマンドが WinDriver のカーネル割り込みハンドラーで発生した場合、割り込みマスク コマンドで設定したマスクで、配列の上位の読み込み転送コマンドでデバイスから読み込まれた値をマスクします。マスクが成功した場合、コントロールがユーザー モードに戻ると、WinDriver は割り込みのコントロールを要求し、配列の転送コマンドの残りを実行し、ユーザー モードの割り込みハンドラ ルーチンを呼び出します。ただし、マスクが失敗した場合、WinDriver 割り込みのコントロールを拒否し、割り込み転送コマンドの残りは実行されず、ユーザー モードの割り込みハンドラ ルーチンは呼び出されません。(注意: レガシー割り込みを処理する場合のみ、割り込みの受信と拒否は関連します。MSI / MSI-X 割り込みは共有されないため、WinDriver は常に割り込みなどのコントロールを受信します。)

注意:

- 共有 PCI 割り込みを正しく処理するには、常に割り込み転送コマンドの配列にマスク コマンドを含め、割り込みハンドラーが割り込みの所有権を要求するかどうか確認するようにこのマスクに設定する必要があります。
- **Windows CE** では、共有割り込みの場合、マスク コマンドより上位にあるコマンドを含む、配列の他のコマンドを実行する前に、マスク コマンドより上位にある関連する読み込みコマンドと一緒に、WinDriver の割り込みハンドラーは、指定した割り込み転送コマンドの配列で検出した最初のマスクコマンドを実行します。このマスクの結果によって、割り込みの所有権が決まります。マスクが失敗した場合、転送コマンドの配列から他の転送コマンドは実行されません(配列のマスク コマンドの上位にあるコマンドを含む)。マスクが成功した場合、WinDriver は、転送コマンドの配列の最初のマスク コマンド(および関連する読み込みコマンド)の上位にあるすべてのコマンドを実行するように処理し、配列のマスク コマンドの下位にあるすべてのコマンドを実行するように処理します。
- より柔軟に割り込み処理をコントロールするには、WinDriver の Kernel PlugIn を使用することができます。マニュアルのセクション 11.6.5 で説明しているとおり、Kernel PlugIn を使用して、カーネル モードの割り込みハンドラ ルーチンを記述することができます。Windows CE では、Kernel PlugIn を実装できなのでご注意ください。

9.2.5.2 WinDriver の転送コマンドのコード例

このセクションでは、WDC (WinDriver Card) ライブリ API を使用して割り込み転送コマンドを設定するためのサンプルコードを紹介します。以下のシナリオのようなサンプルコードを紹介します: レベル センシティブ割り込みを発生させる PCI カードがあることを想定します。割り込みが発生した場合、対象のカードの割り込みコマンドステータス レジスタ (INTCSR、I/O ポートアドレス (dwAddr) へマップされます) の値が intrMask となることを想定します。割り込みをクリアおよび確認するには、INTCSR へ 0 を書き込む必要があります。以下のコードは、以下の内容を実行する WinDriver のカーネル モードの割り込みハンドラーを処理する転送コマンドの配列を定義する方法を紹介します:

1. 対象のカードの INTCSR レジスタを読み込み、値を保存します。

2. 指定したマスク (intrMask) に対して、読み込み INTCSR の値をマスクして、割り込みのソースを確認します。
3. マスクが成功した場合、INTCSR に 0 を書き込み、割り込みを確認します。

注意: サンプルのすべてのコマンドを DWORD のモードで実行します。

例

```
WD_TRANSFER trans[3]; /* Array of 3 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 the 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。

注意: 割り込みが無効になるまで、転送コマンド用に割り当てたメモリを利用可能にしとく必要があります。

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_IntEnable()` 関数の使用方法を紹介します:

```
/* 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 WDC_IntEnable().
   interrupt_handler: Your user-mode interrupt handler routine.
   pData: The data to pass to the interrupt handler routine. */
WDC_IntEnable(hDev, &trans, 3, INTERRUPT_CMD_COPY, interrupt_handler,
   pData, FALSE);
```

9.2.6 WinDriver の MSI / MSI-X 割り込み処理

セクション 9.2.1 の説明のとおり、WinDriver は Linux と Windows Vista 上で PCI MSI (Message-Signaled Interrupt) と MSI-X (Extended Message-Signaled Interrupt) をサポートしています (以前のバージョンの Windows では MSI / MSI-X をサポートしていません)。

レガシー割り込みと MSI / MSI-X 割り込みの両方の処理に同じ API を使用し、これらの API は、ハードウェアがサポートする割り込みタイプに関する情報と有効にした割り込みタイプに関する情報を返します。

Windows Vista で WinDriver を使用する場合、WinDriver のカーネルモードの割り込みハンドラーは、カーネル割り込み処理は、割り込み有効または待機関数へ渡される `WD_INTERRUPT` 構造体の `dwLastMessage` フィールドに割り込みメッセージデータを設定します。WinDriver のサンプルコードおよ

び DriverWizard で生成された割り込みのコードで紹介するように、ユーザー モードの割り込み ハンドラ ルーチン ヘ データ の一部として同じ 割り込み 構造体 を渡す 場合、 割り込み ハンドラ から この 情報 に アクセス する こ とが でき ます。 Kernel PlugIn ドライバ を 使用する 場合 に は、 最後の メッセージ データ は、 カーネル モード の KP_IntAtIrqlMSI() 関数 と KP_IntAtDpcMSI() 関数 ハンドラ へ 渡さ れ ます。

低 レベル の WDC_GET_ENABLED_INT_LAST_MSG マクロ を 使用して、 指定 し た WDC デバイス の 最後の メッセージ データ を 取得 でき ます。

9.2.6.1 Windows MSI / MSI-X の デバイス INF ファイル

注意: Windows 上 で 作業 する 場合 のみ、 この セクション の 情報 は 関連 し ます。

Windows で WinDriver を 使用して PCI の 割り込み を 处理 する に は、 セクション 15.1 で 説明 し た と おり、 ま ず 初め に、 WinDriver の カーネル ドライバ と 対象 の PCI カード が 動作 する よう に 登録 する INF ファイル を インストール する 必要 が あ ります。

Windows で MSI / MSI-X を 使用 する に は、 以下 の よう に、 カード の INF ファイル に 特定 の [Install.NT.HW] MSI 情報 が 含ま れ て いる 必要 が あ ります：

```
[Install.NT.HW]
AddReg = Install.NT.HW.AddReg

[Install.NT.HW.AddReg]
HKR, "Interrupt Management", 0x00000010
HKR, "Interrupt Management\MessageSignaledInterruptProperties", 0x00000010
HKR, "Interrupt Management\MessageSignaledInterruptProperties", MSISupported, 0x10001, 1
```

従って、 Windows Vista で WinDriver と MSI/MSI-X を 使用 する に は (対象 の ハードウェア が MSI / MSI-X を サポート する 場合)、 適切 な INF ファイル を インストール する 必要 が あ ります。

Windows Vista で WinDriver を 使用して MSI / MSI-X を サポート する PCI デバイス の INF ファイル を 生成 する と、 INF 生成 ダイアログ で、 MSI / MSI-X を サポート する INF ファイル を 生成 する か 選択 でき ます。

さらに、 WinDriver の Xilinx Virtex 5 BMD の サンプル では、 MSI の 处理 を 紹介 し、 この 目的 の ため に 関連 する INF ファイル が 含ま れ ます – **xilinx/virtex5/bmd/**diag/ml555_bmd.inf。

注意: 対象 の カード の INF ファイル に MSI / MSI-X の 情報 が 含ま れ て い ない 場合、 対象 の ハードウェア が MSI / MSI-X を サポート する 場合 で も、 上記 で 説明 し た と おり、 WinDriver は レガシー の レベル センシティ ブ 割り込み 处理 方法 を 使用 し、 対象 の カード の 割り込み を 处理 し ます。

9.2.7 ユーザー モード の WinDriver 割り込み 处理 の コード 例

以下の サンプル コード で は、 WDC ライブ ラリ [B.2] の 割り込み API を 使用 し て [マニュアル の セクション B.3.43 – B.3.45 で 説明]、 シンプル な ユーザー モード の 割り込み ハンドラ の 実装 方法 を 紹介 し ます：

WDC 割り込み 関数 を 使用 する 割り込み 处理 の 完全 な ソース コード は、 たとえ ば、 WinDriver の **pci_diag** (**WinDriver/samples/pci_diag/**)、 **pcmcia_diag** (**WinDriver/samples/pcmcia_diag/**) と **PLX** (**WinDriver/plx/**) サンプル、 および DriverWizard で 生成 された PCI / PCMCIA / ISA の コード など を 参照 し て ください。 MSI 割り込み 处理 の サンプル に 關して は、 同じ API を 使用 する、 **virtex 5** の サンプル (**xilinx/virtex5/bmd/**) または MSI / MSI-X を サポート する OS (Linux または Windows Vista) で PCI ハードウェア 用 に DriverWizard で 生成 された コード を 参照 し て ください。

注意:

- 以下のサンプル コードでは、エッジトリガの ISA カード用の割り込み処理を紹介します。コードでは、エッジトリガ割り込みの場合または MSI / MSI-X 割り込みの場合に処理可能な、カーネルモードの割り込み転送コマンドを設定しません。WinDriver を使用してユーザー モードからレベルセンシティブな割り込みまたは PCMCIA 割り込みを処理するには、上記およびセクション 9.2.5 で説明したとおり、カーネルで割り込みを確認するために転送コマンドを設定する必要があります。
- セクション 9.2.6 で説明したとおり、WinDriver はレガシー割り込みと MSI / MSI-X 割り込みの両方を処理する API のセットを提供します。従って、Linux または Windows Vista で、以下のコードを使用して、サンプルの WDC_IsaDeviceOpen() と WDC_PciDeviceOpen() を置き換えることにより、MSI / MSI-X PCI を処理することができます (対象のハードウェアがサポートしている場合)。

例

```

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();
}

```

9.2.8 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、割り込みのマッピングには以下の 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ピングを使用して静的にマッピングします：

```
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 を論理割り込みへ静的にマップします。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。

そして、Windows CE イメージ NK.BIN をリビルドし、対象のプラットフォームに新しい実行形式ファイルをダウンロードする必要があります。

静的マッピングはまた、予約済み割り込みマッピング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ピング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 1 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。割り込みマッピング 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、Windows CE 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);
```

9.2.8.1 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 と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イルします。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. 低レベル **WD_xxx API** を使用する場合は、**WD_IntEnable()** を呼び出した直後に **WD_InterruptDoneCe()** を呼び出します。

注意: WinDriver の WDC API を使用して割り込みを処理する場合、または低レベル **InterruptEnable()** 関数を使用して割り込みを有効にする場合、**WDC_IntEnable()** または **InterruptEnable()** が自動的に **WD_InterruptDoneCE()** を呼び出すため、**WD_InterruptDoneCe()** を呼び出す必要はありません。

WD_InterruptDoneCe() は、パラメータを 2 つ受け取ります:

```
void WD_InterruptDoneCe(HANDLE hWD, WD_INTERRUPT pInt);
```

- **hWD:** **WD_Open()** から受信した際の WinDriver のカーネルモード ドライバへのハンドル
- **pInt:** **WD_IntEnable()** から戻ってきた **WD_Interrupt** 構造体へのパラメータ

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。

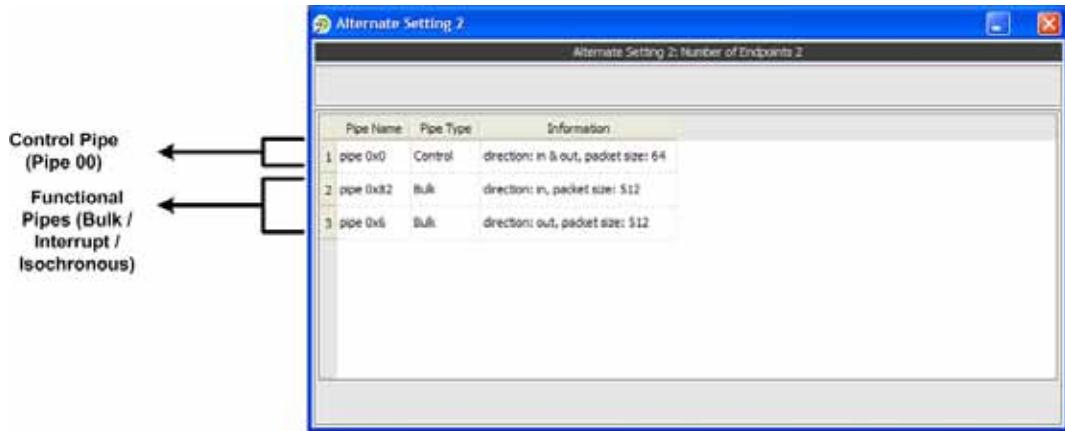


図 9.1: USB データ交換

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。

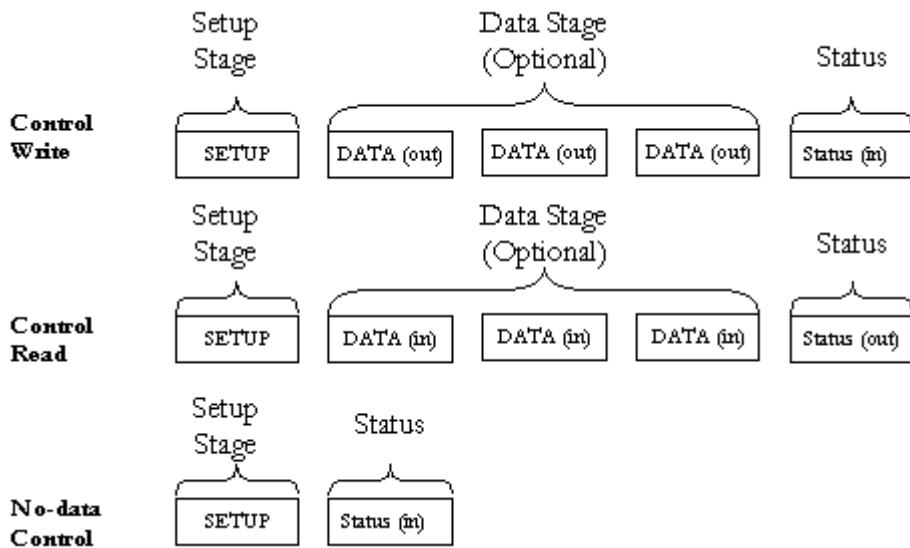


図 9.2: USB のリードとライト

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)を使用して、デバイスへコマンドを設定し、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のセットアップ パケットは、各 USB デバイスに対して、ベンダーのデータ ブックに記載されています。

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となります。詳細は、<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、値はディスクリプタのタイプを示し、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となります。

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	ディスクリプタ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 ディスクリプタが 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 はディスクリプタの長さを示し、バイト 2-3 は USB の仕様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. Pipe 0x0 を選択し、Read / Write ボタンをクリックします。
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)。

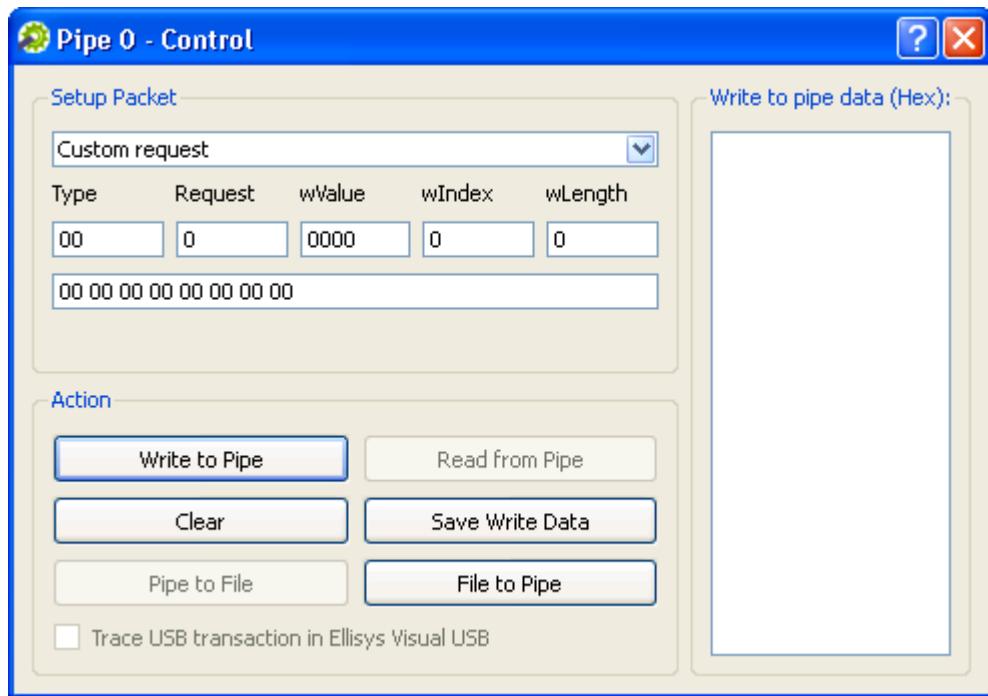


図 9.3: カスタム要求

- 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。

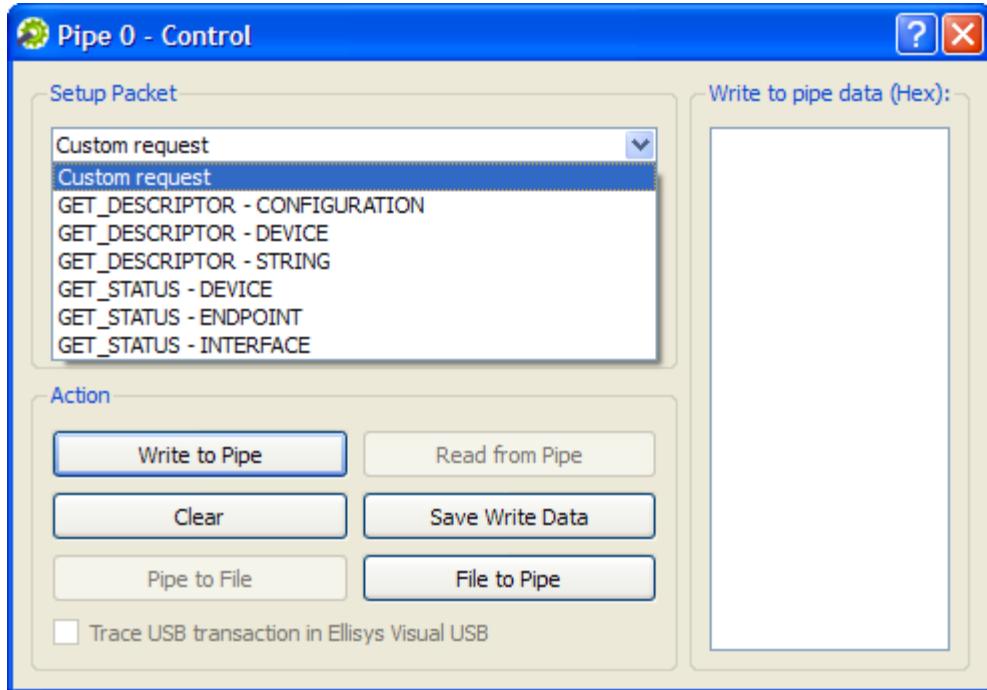


図 9.4: 要求一覧

- 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 の 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; /* BmRequestType */
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 データ交換を使用して、デバイスからまたはデバイスへのデータを転送します。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 [5.2.3] 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、ドライバは事前定義したブロック 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` 関数または `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。

開いているすべてのストリームに対して `WDU_StreamGetStatus()` を呼んで、ストリームの現在のステータス情報を取得できます。

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および再開できます。

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。

- 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 4.1.3 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 for Linux AMD64 および 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) 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) の小人国の人々から来ており、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。

- 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、低レベル WD_Transfer() 関数および低レベル 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 マスター 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 マスター 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 用の一連のディスクリプタです。

以下の例は、WinDriver ライブライ (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することが可能です。

- ユーザーモード ドライバのパフォーマンスを向上する。[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する必要があるコードを Kernel PlugIn に移動する。

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を達成できます。Windows CE で割り込み処理率を向上するには、セクション 9.2.8 を参照してください。

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を向上する最善の方法を検討してください：

10.1.1 パフォーマンスを向上するためのチェックリスト

次のチェックリストは、ドライバのパフォーマンスを向上する方法を選択するのに役立ちます：

問題	解決方法
#1 ISA カード – カード上の I/O マップ領域へのアクセス	<ul style="list-style-type: none"> 複数の <code>WD_Transfer</code> を <code>WD_MultiTransfer</code> 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 マップ領域へのアクセス	<ul style="list-style-type: none"> 対象のハードウェア デザインの I/O 領域の使用を回避します。
#3 カード上のメモリ マップ領域へのアクセス	<ul style="list-style-type: none"> 関数を呼び出さ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 割り込み待ち時間 – 割り込みの受信漏れ、割り込みの受信の遅延	Kernel PlugIn ドライバを記述して、カーネルモードで割り込みを処理します (詳細は、第 11 章 および 第 12 章 の説明を参照してください)。
#5 PCI ターゲット アクセス vs. マスター アクセス	PCI ターゲット アクセスは、通常、PCI マスター アクセス (バス マスター DMA) より速度は遅いです。大きいデータの転送の場合、バス マスター DMA アクセスの方が有効です。マニュアルのセクション 9.1 で WinDriver を使用してバス マスター DMA の実装方法を説明しています。

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 マップの領域への転送は、I/O マップの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 マップの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 マップの領域への直接アクセス

`WDC_xxxDeviceOpen()` 関数 (PCI / PCMCIA / ISA) または低レベル `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 モードまたはカーネル 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。

低レベル 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() または低レベル WD_Transfer() 関数を使用してブロック(文字列)転送を行う。
- WDC_MultiTransfer() または低レベル 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 および Linux 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 および Linux 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。WDK 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 WinDriver 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()`、低レベル `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を達成できます。
Windows CE で割り込み処理率を向上するには、セクション 9.2.8 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、パフォーマンス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 の 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 マップのメモリ領域へのアクセスなど)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 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 の 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。

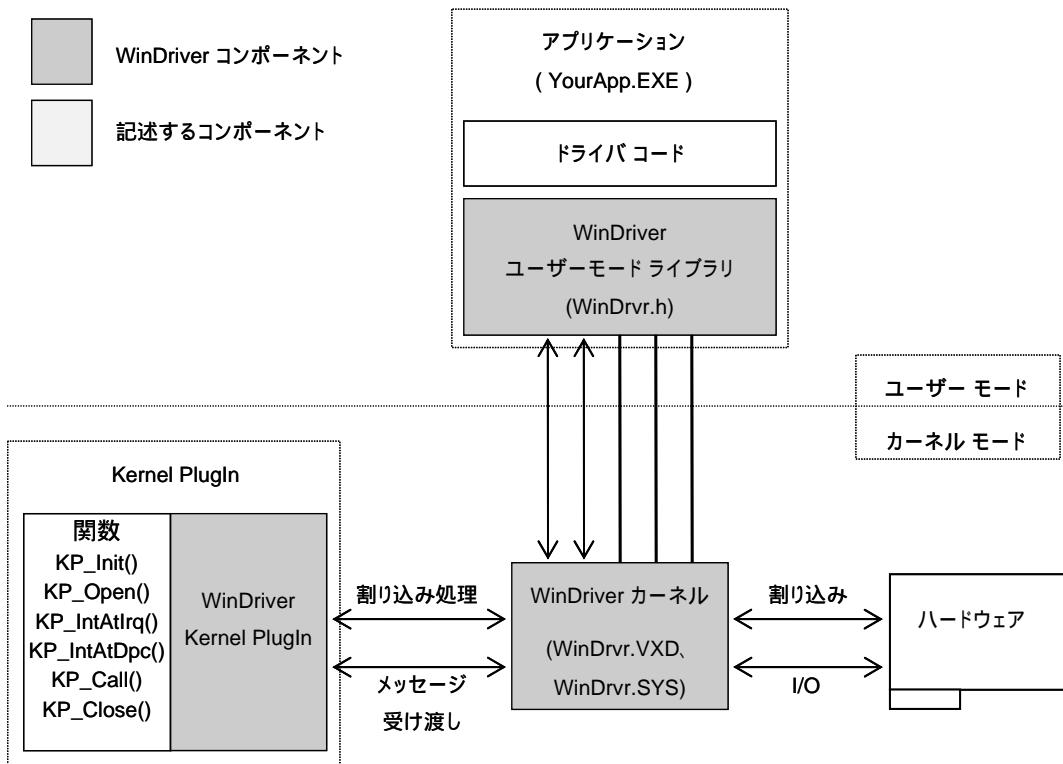


図 11.1: KernelPlugIn の構造

11.5.2 WinDriver のカーネルと Kernel Plugin の相互作用

WinDriver のカーネルと WinDriver の Kernel PlugIn は次の 2 種類の相互作用があります:

1. **割り込み処理:** WinDriver が割り込みを受信すると、ユーザー モードの割り込みハンドラー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 モードの割り込みハンドラーを有効にします。Kernel PlugIn の割り込みハンドラーは、基本的に Kernel PlugIn へ移動する前に、ユーザー モードの割り込みハンドラーで記述およびデバッグし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 の 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コンポーネ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

以下、Kernel PlugIn で実装できるすべての関数の一般的なイベント シーケンスです:

11.5.4.1 ユーザーモードから Kernel PlugIn ドライバ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。

<p>コールバック: KP_Init() Kernel PlugIn ルーチンを呼び出します。</p>	<p>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) 低レベルの 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。</p>
<p>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) 低レベルの WD_KernelPlugInOpen() 関数を呼び出します。</p>	
<p>コールバック: KP_Open() Kernel PlugIn ルーチンを呼び出します。</p>	<p>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。</p>

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
<p>イベント: アプリケーションは WDC_CallKerPlug()、または 低レベルの WD_KernelPlugInCall() 関数を呼び出します。</p>	<p>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。</p>
<p>コールバック: KP_Call() Kernel PlugIn ルーチンを呼び出します。</p>	<p>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。</p>

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
<p>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() を呼びか (Kernel PlugIn でデバイスを開いた後)、または、</p>	

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: 高い IRQL の Kernel PlugIn の割り込み処理ルーチン - KP_IntAtIrql() (レガシー割り込み) または KP_IntAtIrqlMSI() (MSI / MSI-X) を呼び出します。	KP_IntAtIrql() と KP_IntAtIrqlMSI() 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() または KP_IntAtDpcMSI() で追加処理を引き継ぐために KP_IntAtIrql() (レガシー割り込み) または KP_IntAtIrqlMSI() (MSI / MSI-X) 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、低レベルの 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 の高い IRQL の割り込みハンドラ - KP_IntAtIrql() または KP_IntAtIrqlMSI() が TRUE を戻します。	カーネルで引き継いだ手順 (DPC) 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: Kernel PlugIn の DPC 割り込みハンドラ - KP_IntAtDpc() (レガシー割り込み) または	残りの割り込みコードを処理しますが、高い IRQL 割り込み処理よりは優先度が低いです。

KP_IntAtDpcMSI() (MSI / MSI-X) を呼び出します。	
イベント: DPC 割り込みハンドラ - KP_IntAtDpc() は KP_IntAtDpcMSI() 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 ルーチンが実行されます。

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。

まず初めにユーザー モードでドライバ コード全体を記述し、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 ドライバへ移植します。

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 2000 / XP / Server 2003 / Server 2008 / 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 WDK (Windows Driver Kit) がインストールされたホスト マシン

注意: WDK 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/devtools/WDK/WDKpkg.mspx> から購入できます。
- Linux 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 (呼び出しえ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:

- 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。
- 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、低レベル 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_INIT 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、または低レベルの 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) 低レベルの 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() 関数または低レベルの (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_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レベルの 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レベルの 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 がレガシー 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。
KP_IntAtIrqlMSI()	WinDriver が MSI または MSI-X 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 MSI / MSI-X を提供)。この関数は カーネル モードで MSI / MSI-X 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_IntAtDpcMSI() またはユーザー モードで実行されます。 注意: Linux および Windows Vista で MSI / MIS-X をサポートしています。
KP_IntAtDpcMSI	KP_IntAtIrqlMSI() コールバックが TRUE に返る MSI / MSI-X 割り込み処理

()	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 MSI / MSI-X 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。 注意: Linux および Windows Vista で MSI / MIS-X をサポートしています。
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() を呼び出か (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レベルの 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。

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->funcIntAtIrqlMSI = KP_PCI_IntAtIrqlMSI;
    kpOpenCall->funcIntAtDpcMSI = KP_PCI_IntAtDpcMSI;
    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_Intxxxx()` 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 を使用して対象のデバイスの Kernel PlugIn ドライバの雛型を生成します。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、Kernel PlugIn の WinDriver のサンプルを使用することもできます。

注意: このマニュアルの Kernel PlugIn に関して、DriverWizard で生成されたコードと汎用的な PCI の Kernel PlugIn のサンプル `KP_PCI` (`WinDriver/samples/pci_diag/kp_pci/` Kernel PlugIn ディレクトリ以下にあります) を中心に説明しています。

BMD (Bus Mastering DMA Validation Design) ファームウェアを持つ Xilinx Virtex 5 PCI Express チップを使用する場合、開発のベースとしてこのチップ用の特定の `KP_VRTX5` Kernel PlugIn サンプルを使用することができます。

`WinDriver/xilinx/virtex5/bmd/` ディレクトリに Virtex 5 BDM サンプルの関連するすべてのファイルが含まれます。

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。ドライバ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()` 関数へデータを渡す方法と 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、格納する関数の使用方法を紹介します。

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 で特定の機能を開始する方法、およびメッセージを通じて Kernel PlugIn ドライバとユーザー モードの WinDriver アプリケーションの間でデータを渡す方法を紹介するために、ドライバのバージョン番号を取得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` サンプルの `KP_IntAtIrql()` 関数と `KP_IntAtDpc()` 関数ではレガシーなレベル センシティブ PCI の割り込み処理を紹介していますが、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。サンプルの `KP_IntAtIrqlMSI()` 関数と `KP_IntAtDpcMSI()` 関数では MSI (Message-Signaled Interrupts) と MSI-X (Extended Message-Signaled Interrupts) 処理を紹介しています。

DriverWizard で生成されたコードには選択したデバイス (PCI / PCMCIA / ISA) 用のサンプルの割り込み処理のコードが含まれます。生成された `KP_IntAtIrql()` 関数にはウィザード (**Interrupt** タブで、レジス

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が含まれます。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 のレガシー割り込みの場合 (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が含まれるよう、Kernel PlugIn コードを生成する前に、ウィザードを使用して割り込みを認識 (クリア) するコマンドを定義することを推奨します。また、MSI / MSI-X をサポートするハードウェア用に割り込みを処理する場合には、その転送コマンドを用意することを推奨します。用意しない場合には、MSI / MSI-X の有効に失敗したり、デフォルトのレベル センシティブな割り込みを使用して割り込みを処理します (ハードウェアでサポートしている場合)。注意: 割り込みが無効になるまで、転送コマンド用に割り当てたメモリを利用可能にしどく必要があります。

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、生成された Kernel PlugIn のプロジェクトまたはサンプル プロジェクト (および対応するユーザー モード アプリケーション) をそのままビルドして実行することを推奨します。ただし、上記で説明したとおり、サンプルの `Kp_IntAtIrql()` 関数のハードウェア独自の転送コマンドを編集または削除する必要があるのでご注意ください。

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`) を実装しています。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います。

以下、`WinDriver/samples/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 (`msdev2008 / msdev2005 / msdev2003 / msdev_6`) 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:
 - Windows x86 32 ビット: `WINNT.i386\kp_pci.sys` - Windows 2000 およびそれ以前用にビルドしたドライバの 32 ビットバージョン
 - Windows x64: `WINNT.x86_64\kp_pci.sys` - Windows Server 2003 およびそれ以前用にビルドしたドライバの 64 ビットバージョン
 - 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 2000 / XP / Server 2003 / Server 2008 / 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。
- 注意:** MSI (Message-Signaled Interrupt) または MSI-X (Extended Message-Signaled Interrupt) を Windows Vista で使用する場合 (MSI / MIS-X をサポートする PCI カード向け)、サンプルの INF ファイルを修正または置き換えて、特定の MSI 情報を含める必要があります。MSI 情報が含まれない場合、マニュアルのセクション 9.2.6.1 で説明したとおり、WinDriver は対象のカードに対して、レガシーなレベル センシティブ割り込み処理を使用します。
- **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 **makefile**。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 (**msdev2008** / **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 などその他の OS 用の **makefile** は、それぞれ **<os>/** サブ ディレクトリにあります (たとえば、**LINUX/**)。
 - 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**
 - **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

注意: BMD (Bus Mastering DMA Validation Design) ファームウェアを持つ Xilinx Virtex 5 PCI Express チップのサンプル (**WinDriver/xilinx/virtex5/bmd/**) のディレクトリの構造は、次の項目を除いて、汎用的な PCI サンプル **pci_diag** ディレクトリと同様です。サンプルの **virtex5_lib** ライブラリ ファイルは **lib/** サブ ディレクトリ以下にあります。**virtex5_diag** ユーザー モード アプリケーション ファイルは **diag/** サブ ディレクトリおよび **kp/** サブ ディレクトリ以下にあります。**kp/** サブ ディレクトリには、Kernel PlugIn ドライバ (**KP_VRTX5**) のソース ファイルと Windows 用にのみ **makefile** が含まれます。

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、レガシー 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、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関連するコマンドを設定することを強く推奨します。それによって、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なります。また、MSI / MSI-X をサポートするハードウェア用に割り込みを処理する場合には、その転送コマンドを用意することを推奨します。用意しない場合には、MSI / MSI-X の有効に失敗したり、デフォルトのレベル センシティブな割り込みを使用して割り込みを処理します (ハードウェアでサポートしている場合)。注意: 割り込みが無効になるまで、転送コマンド用に割り当てたメモリを利用可能にしつく必要があります。

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。

- `kernmode/` - 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 (`msdev2008` / `msdev2005` / `msdev2003` / `msdev_6`) にあります。Linux などその他の OS 用の `makefile` は、それぞれ `<os>/` サブディレクトリにあります (たとえば、`LINUX/`)
- `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 `makefile`。

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 (`msdev2008` / `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 などその他の OS 用の `makefile` は、それぞれ `<os>/` サブディレクトリにあります (たとえば、`LINUX/`)。

- `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 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 ドライバの高い IRQL ハンドラ (`KP_IntAtIrql()` (レガシー割り込み) または `KP_IntAtIrqlMSI()` (MSI / MSI-X)) を呼びます。高い IRQL ハンドラが `TRUE` を返す場合、高い IRQL ハンドラが処理を終え、`TRUE` を返した後に、Kernel PlugIn の引継ぎ割り込みハンドラー (`KP_IntAtDpc()` (レガシー割り込み) または `KP_IntAtDpcMSI()` (MSI / MSI-X)) を呼びます。DPC 関数の戻り値は、ユーザー モードの割り込み処理ルーチ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。高い IRQL (`KP_IntAtIrql()` または `KP_IntAtIrqlMSI()`) は 5 回の割り込み毎に `TRUE` を返し、DPC ハンドラー (`KP_IntAtDpc()` または `KP_IntAtDpcMSI()`) を有効にし、DPC 関数は高い IRQL ハンドラーからの実際の DPC 呼び出しの回数を返します。つまりユーザー モードの割り込み処理は 5 回の割り込み毎に 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チンを起動します (主に、`WDC_IntEnable()` またはより低レベルの `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。

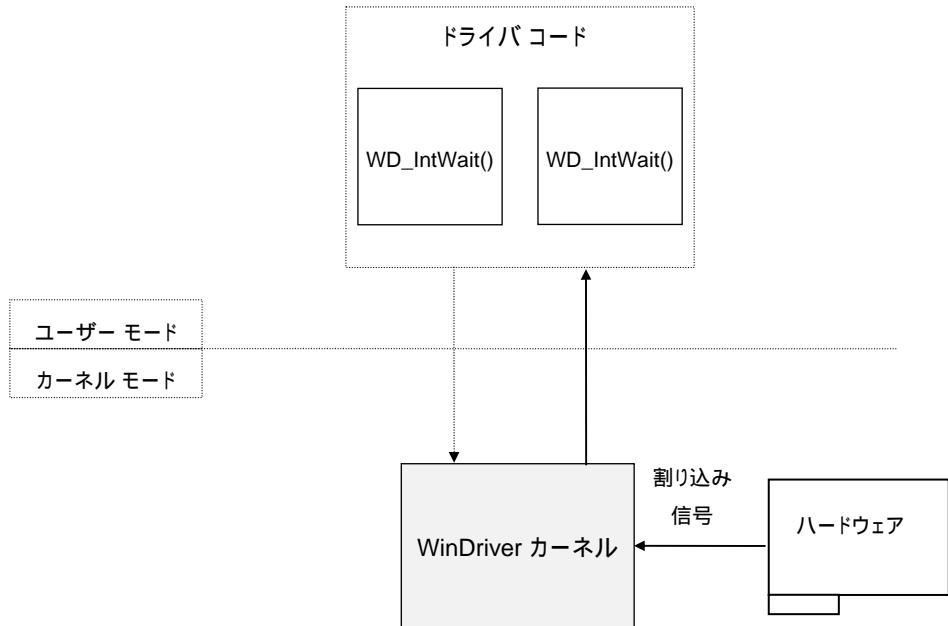


図 11.2: Kernel Plugin なしでの割り込みの処理

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。

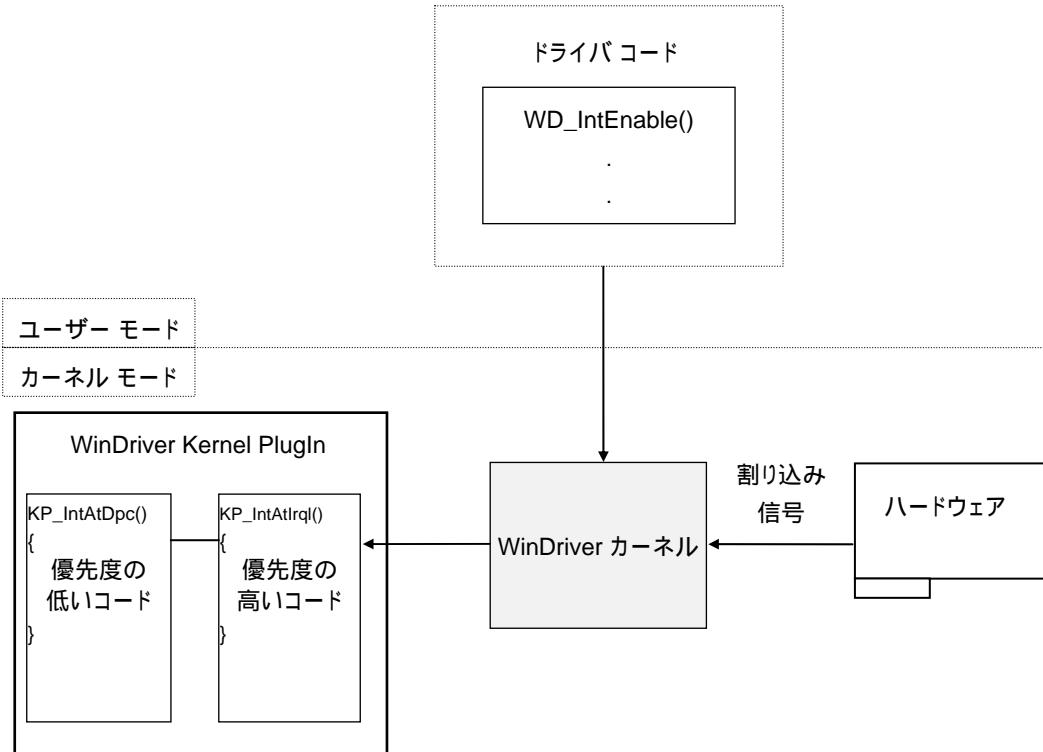


図 11.3: Kernel Plugin ありでの割り込み処理

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)。

WDC_IntEnable() / InterruptEnable() / WD_IntEnable() を呼び出して Kernel PlugIn で割り込みを有効にする際、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、有効になった割り込みの種類を基に、関連する高い IRQL ハンドラー (KP_IntAtIrql() (レガシー割り込み) または KP_IntAtIrqlMSI() (MSI / MSI-X)) が割り込みのたびに呼び出されます。高い IRQL ハンドラーのコードを高い割り込みレベルで実行します。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)。

高い IRQL で実行中のコードは、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_xxx() のアドレスまたは設定空間 read / write 関数
 - WDC_MultiTransfer()、または低レベルの WD_Transfer()、WD_MultiTransfer() または WD_DebugAdd() 関数
 - 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 (WDK 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、高い IRQL ハンドラー (KP_IntAtIrql() または KP_IntAtIrqlMSI()) の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 DPC 関数 (KP_IntAtDpc() または KP_IntAtDpcMSI()) で実装します。DPC 関数は、引継いだ割り込みレベルで実行し、高い IRQL ハンドラーと同じ制限を持ちません。その DPC 関数と一致する高い IRQL 関数が戻り値を返した後に (TRUE を返した場合)、DPC 関数を呼びます。

ユーザー モードで割り込み処理を行うこともできます。DPC 関数 (KP_IntAtDpc() または KP_IntAtDpcMSI()) の戻り値が、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チンを呼ぶ回数となります。

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() または低レベルの 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)。

DriverWizard で生成されたコードおよびサンプルの 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 の中に実行されます。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、Kernel PlugIn ドライバの開発の雛型として、WinDriver の Kernel PlugIn のサンプルを使用することもできます。あるいは、コードをゼロから開発することもできます。

注意: このマニュアルの Kernel PlugIn に関して、DriverWizard で生成されたコードと汎用的な PCI の Kernel PlugIn のサンプル **KP_PCI** (*WinDriver/samples/pci_diag/kp_pci* ディレクトリ以下にあります) を中心に説明しています。

BMD (Bus Mastering DMA Validation Design) ファームウェアを持つ Xilinx Virtex 5 PCI Express チップを使用する場合、開発のベースとしてこのチップ用の特定の **KP_VRTX5** Kernel PlugIn サンプルを使用することができます。

WinDriver/xilinx/virtex5/bmd/ ディレクトリに Virtex 5 BDM サンプルの関連するすべてのファイルが含まれます。

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。カーネル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。このため、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、低レベル `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、高い IRQL および DPC の 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関連する高い IRQL ハンドラー (`KP_IntAtIrql()` 関数 (レガシー割り込み) または `KP_IntAtIrqlMSI()` 関数 (MSI / MSI-X)) へ移動する必要があります。割り込み処理の引継ぎを関連する DPC ハンドラー (`KP_IntAtDpc()` 関数 (レガシー割り込み) または `KP_IntAtDpcMSI()` 関数 (MSI / MSI-X)) へ移動することができます。高い IRQL ハンドラーが割り込み処理を終了し、`TRUE` を返すと関連する DPC ハンドラーを実行します。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、低レベルの `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 DriverWizard で生成された Kernel PlugIn の `<project_dir>\kernmode\` 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_2008\kp_xxx.vcproj`: 32 ビット MSDEV 2008 プロジェクト
 - `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_2008\kp_xxx.vcproj`: 64 ビット MSDEV 2008 プロジェクト
 - `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_2008\xxx_diag.vcproj**: 32 ビット MSDEV 2008 プロジェクト
 - **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_2008\xxx_diag.vcproj**: 64 ビット MSDEV 2008 プロジェクト
 - **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 WDK (Windows Driver 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)。
2. WDK をインストールした場所を示すように **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。

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 DriverWizard 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 (**msdev2008** / **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 DriverWizard 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。

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 / 2008 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: 複数の OS 用にドライバをビルドするには、そのドライバサポートする最も下位の OS を選択します。たとえば、Windows 2000、XP、およびそれ以降をサポートする場合、**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。

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。

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 --disable-usb-support
```

注意: **configure** スクリプトは、起動してい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。

Linux カーネル バージョンが 2.6.26 またはそれ以降の場合、**configure** は、**kbuild** を使用してカーネル モジュールをコンパイルする **makefile** を生成します。以前のバージョンの Linux で、**kbuild** を強制的に使用するには、**configure** に **--enable-kbuild** フラグを渡します。

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)。

- 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/
```

- 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。

12.8 Kernel PlugIn ドライバのインストール

12.8.1 Windows の場合

注意: 管理者権限で下記の手順を実行してください。

- 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 / Server 2008 / Vista の場合は C:\Windows\system32\drivers)
- Windows 2000 / XP / Server 2003 / Server 2008 / 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。

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。

SYS ドライバのインストール:

```
WinDriver\util> wdreg -name KP_NAME install
```

注意: 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

- 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/
```

- 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。

注意: root 権限で次のコマンドを実行してください。

```
make install
```

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 / Server 2008 / Vista 上で .sys 拡張子のファイル。たとえば、`windrivr6.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 (`windrivr6.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)。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。

ここでは、Windows オペレーティング システムの `wdreg` / `wdreg_gui` の使用方法を説明します。

注意:

1. `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。**wdreg** の文字列を **wdreg_gui** に置き換えられます。

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 で WDM **windrivr6.sys** ドライバをインストールする方法、または Windows 2000 / XP / Server 2003 / Server 2008 / 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 **windrivr6.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: 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 2000 / XP / Server 2003 / Server 2008 / 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>]** – 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。1 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]** – 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]** – 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 2000 / XP / Server 2003 / Server 2008 / Vista 上での Kernel PlugIn 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、-startup 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]** – 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 / Server 2008 / 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 2000 / XP / Server 2003 / Server 2008 / 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。

- `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
```

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: 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 KernelPlugin 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

注意: root 権限で次のコマンドを実行してください。

- WinDriver を動的にロードするには、次のコマンドを実行します。

```
<wdreg へのパス>/wdreg 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/modprobe windrvr6
```

wdreg は **WinDriver/util/** ディレクトリ以下にあります。

ヒント: Linux のブートファイル (**/etc/rc.d/rc.local**) に次の行を追加して、システムを起動するごとに WinDriver を自動的にロードします。

<wdreg へのパス>/wdreg windrvr6

13.4 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:

- 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。
- 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)。
- 新しい INF ファイルと (または) カタログ ファイルを作成した場合、オリジナルの WinDriver の INF ファイルへの参照と (または) `wd1000.cat` ファイルへの参照を新しいファイル名に置き換えてください (詳細は、セクション 15.2.1 と 15.3.2 を参照してください)。
- 32-bit と 64-bit の両方のターゲット プラットフォームにドライバを配布する場合、各プラットフォーム用にそれぞれ WinDriver のインストール パッケージを別々に用意してください。
各パッケージに必要なファイルは、それぞれのプラットフォーム用の WinDriver のインストール ディレクトリ以下にあります。

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)。
- `wd1000.cad` (WinDriver\redist ディレクトリにあります)。
- `wd_api1000.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)。
`wdapi1000_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 / PCMCIA / 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 に接続してある 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 ファイルではなく、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 ([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:

`windrivr6.sys`、`windrivr6.inf` と `wd1000.cat` ファイルを同じディレクトリにコピーします。

注意: `wd1000.cat` には、ドライバ認証のデジタル署名が含まれます。署名の認証を維持するには、`windrivr6.inf` ファイルと同じインストール ディレクトリにおく必要があります。カタログ ファイルと INF ファイルを異なるディレクトリに配布する場合、またはこれらのファイルを変更またはカタログ ファイルによって参照されるファイル (`windrivr6.sys` など) を変更する場合、以下のいずれかの処理が必要です:

- 新しいカタログ ファイルを作成し、このファイルを使用してドライバを再署名する。

- `windrivr6.inf` ファイルの以下の行をコメントアウトか削除します:

`CatalogFile = wd1000.cat`

そして、ドライバの配布にカタログ ファイルを含めません。ただし、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。

ドライバのデジタル署名と認証および対象の WinDriver ベースのドライバの署名に関しては、マニュアルのセクション 15.3 を参照してください。

`wdreg` 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。

`wdreg -inf <windrvr6.inf のパス> install`

たとえば、`windrivr6.inf` および `windrivr6.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、`difxapi.dll` DLL に依存します。
- `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 `windrivr6.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):

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。

- 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。

- Wdapi1000.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 wdapi1000.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、wdapi1000_32.dll から wdapi1000.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 wdapi1000.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 / 2008 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 CE 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`(たとえば、`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 / 2008 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 (`windrivr6.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 (`windrivr6.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。たとえば、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 `windrivr6.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 CE 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 (たとえば、**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 CE 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。たとえば、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** の名前をつけています。
 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 **wdapi1000.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 **WDxxxxCE.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 / 2008 (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 **wdapi1000.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

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 のドライバ 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る。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。
たとえば、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 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)。
- 32-bit と 64-bit の両方のターゲット プラットフォームにドライバを配布する場合、各 プラットフォーム用にそれぞれ WinDriver のインストール パッケージを別々に用意してください。
各パッケージに必要なファイルは、それぞれの プラットフォーム用の WinDriver のインストール ディレクトリ以下にあります。

14.4.1 カーネル 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_reparm.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_reparm` を `reparm` 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使用します。
- `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。
注意: Linux カーネル バージョンが 2.6.26 またはそれ以降の場合、`configure` は、`kbuild` を使用してカーネル モジュールをコンパイルする `makefile` を生成します。以前のバージョンの Linux で、`kbuild` を強制的に使用するには、`configure` に `--enable-kbuild` フラグを渡します。
注意: PCI 版の場合、コンパイル用に WinDriver の設定を行う場合、USB のサポートを外してください:
`./configure --disable-usb-support`
- `configure.wd`: `makefile.wd` [`.kbuild`] . `in` の `makefile.wd` [`.kbuild`] を作成するスクリプト。

- **makefile.in**: メインの WinDriver makefile 用のテンプレートで、**makefile.wd[.kbuild]** を make することで WinDriver をコンパイルおよびインストールします。
- **makefile.wd.in**: メインの WinDriver カーネル モジュールをコンパイルおよびインストールする makefile 用のテンプレートです。
- **makefile.wd.in.kbuild.in**: **kbuild** を使用してメインの WinDriver カーネル モジュールをコンパイルし、モジュールをインストールする makefile 用のテンプレートです。
- **setup_inst_dir**: 対象のドライバ モジュールをインストールするスクリプトです。
- **wdreg (WinDriver/util ディレクトリ以下にあります)**: 対 WinDriver のカーネル ドライバ モジュールをロードするスクリプトです。
注意: **setup_inst_dir** スクリプトは **wdreg** を使用してドライバ モジュールをロードします。

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 **libwdapi1000.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 x86 または 32 ビット PowerPC ターゲットの場合は **/usr/lib/**、64 ビット x86 ターゲットの場合は **/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、**libwdapi1000.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いる 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推奨します。Kernel PlugIn のコード生成で DriverWizard 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_reparm.o**、
kp_wdapi1000_gcc_v2.a、**kp_wdapi1000_gcc_v3.a** および
kp_wdapi1000_gcc_v3_reparm.a

`xxx_gcc_v2.o/a`、`xxx_gcc_v3.o/a`、`xxx_gcc_v3_reparm.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

インストールシェルスクリプトを作成して、ターゲットで自動的にビルドしインストールすることを推奨します。

第 15 章

ドライバのインストール - 高度な問題

15.1 Windows INF ファイル

デバイス情報ファイル (INF) は、Windows 2000 / XP / Server 2003 / Server 2008 / 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

- 対象の PCI / PCMCIA / USB デバイスが WinDriver のカーネル モジュールと動作するように登録するため。
- 既存のドライバを新規のものに置き換えるため。
- WinDriver ベースのアプリケーションと DriverWizard から PCI / PCMCIA / USB デバイスへのアクセスを有効にするため。
- WinDriver で PCI / PCMCIA デバイスのリソース (I/O 範囲、メモリー範囲および割り込み) に関する Plug-and-Play の情報を取得するため。

注意: MSI (Message-Signaled Interrupts) または MSI-X (Extended Message-Signaled Interrupts) の処理には、INF ファイルに特定の設定が必要です。詳細は、セクション 9.6.2.1 を参照してください。

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: 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。

wdreg ユーティリティで 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）。

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。

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。

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: 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。
2. INF ファイルをインストールします。

wdreg ユーティリティで `install` コマンドを使用して、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）。

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。

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。

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]。`windrivr6.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 および Linux 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、`windrivr6.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、`windrivr6.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

DriverWizard で WinDriver Windows カーネル ドライバ (`windrivr6.sys`) の名前変更の作業の多くを自動化できます。

注意: 署名済みの `windrivr6.sys` ドライバの名前を変更する場合、その署名は無効になります。その場合、新しいドライバを署名するか、または署名なしのドライバを配布するかいずれかを選択することになります。ドライバの署名と認証に関する情報は、セクション 15.3 を参照してください。名前変更したドライバの署名のガイドラインは、セクション 15.3.2 を参照してください。

このセクションの `xxx` への参照は、DriverWizard で生成したドライバ プロジェクトに名前に置き換えてください。

次の手順に従い、WinDriver Windows カーネル ドライバの名前を変更します：

- 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。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。`windrivr6.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。
- **wdapi1000.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、WDK (Windows Driver Kit) 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 WDK 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 WDK 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: `-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 14.2 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。

15.2.2 Linux ドライバの名前変更

DriverWizard で WinDriver Linux カーネル ドライバ (`windrvr6.o/.ko`) の名前変更の作業の多くを自動化できます。

このセクションの `xxx` への参照は、DriverWizard で生成したドライバ プロジェクトに名前に置き換えてください。

次の手順に従い、WinDriver Linux カーネル ドライバの名前を変更します:

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。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。

15.3 デジタル ドライバの署名と認証 - Windows 2000 / XP / Server 2003 / Server 2008 / Vista

15.3.1 概要

対象のドライバを配布する前に、Microsoft 社の Windows ロゴ プログラムの認証と署名へドライバを申請するか、Authenticode でドライバを署名するかいずれかで、ドライバをデジタル署名するか、デジタル認証するかを選択できます。

Windows XP およびそれ以前の Windows OS では、ドライバがデジタル署名または認証されていない場合でも、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、ドライバの署名や認証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ドライバのインストールが行える。
- 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を回避できる。
- Windows XP 以降では、INF ファイルの完全なプレインストールは、署名済みのドライバのみサポートしている。

Windows Vista およびそれ以降 (Vista と Windows Server 2008) の 64 ビットバージョンでは、カーネルモードでロードするソフトウェアの KMCS (Kernel-Mode Code Signing) が必要です。WinDriver ベースのドライバには次の実装があります：

- INF ファイルを使用してインストールするドライバを署名付のカタログ ファイルと一緒に配布する必要があります。
- INF ファイルを使用しないでインストールするパライバ (Kernel PlugIn ドライバ) にドライバの署名を組み込む必要があります。

注意: ドライバの開発中は、一時的に署名なしのドライバをインストールできるように Windows を設定できます。

デジタル ドライバの署名と認証に関する詳細情報は、次の情報を参照してください：

- Windows のドライバ署名の要件:
<http://www.microsoft.com/japan/whdc/winlogo/drvsign/drvsign.mspx>
- MSDN (Microsoft Development Network) のドキュメントの「Introduction to Code Signing」のトピックを参照してください。
- Windows Vista を実行しているシステムでのカーネル モジュールへのデジタル署名:
<http://www.microsoft.com/japan/whdc/winlogo/drvsign/kmsigning.mspx>

15.3.1.1 Authenticode ドライバの署名

Microsoft の Authenticode メカニズムはドライバの発行者の整合性を保証します。Authenticode によって、ドライバ開発者は、デジタル署名を使用して、開発者に関する情報と開発者のプログラムと開発者のコードを含めることができます。ユーザーは、ドライバの発行者が信頼あるエントリのインフラに参加していることを確認できます。ただし、Authenticode の署名は、コードの安全性や機能性を保証するわけではありません。

WinDriver\redist\windrvr6.sys ドライバは Authenticode デジタル署名を持っています。

15.3.1.2 WHQL ドライバ認証

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、ドライバの発行者の整合性とドライバの安全性と機能性を保証する Microsoft 資格認定を取得することができます。

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 WHQL (Windows Hardware Quality Labs) テストにドライバとハードウェアを提出します。この手順により、ドライバの発行者と動作を確認できます。

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ドライバ署名と認証

上記の説明のとおり [15.3.1.1]、**WinDriver\redist\windrvr6.sys** ドライバは Authenticode 署名を持っています。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汎用的なドライバのため、WHQL 認証用にスタンドアロン ドライバとして申請することができません。ただし、WinDriver を使用して対象ハードウェア用の Windows 2000 / XP / Server 2003 / Server 2008 / Vista ドライバを開発した場合、以下で説明するとおり、Microsoft WHQL 認定用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。

ドライバの認証と署名の手順は (Authenticode または WHQL のいずれかで)、ドライバのカタログ ファイルの作成が必要です。このファイルは一種のハッシュで、他のファイルを説明します。署名済みの **windrvr6.sys** ドライバは一致するカタログ ファイル(**WinDriver\redist\wd1000.cat**)と提供されます。このファイルは **windrvr6.inf** ファイル (**redist** ディレクトリでも同様に配布されています) の CatalogFile エントリに割り当てられます。ドライバのインストール中に、このエントリを使用して、ドライバの署名と関連するカタログ ファイルを Windows に通知します。

カタログ ファイルに記載されているファイルの名前、コンテンツ、または日付さえも、変更した場合には、その変更によって、カタログ ファイルとそれに関連するドライバ署名は無効になります。そのため、**windrivr6.sys** ドライバ [15.2]、および (または) 関連する **windrivr6.inf** ファイルの名前変更する場合、**wd1000.cat** カタログファイルと関連するドライバ署名は無効になります。

さらに、WinDriver を使用して Plug-and-Play デバイス向けのドライバを開発する場合、通常は、**windrivr6.sys** ドライバ モジュール (または名前を変更したドライバ) で動作するよう登録するデバイス特有の INF ファイルを作成します。対象のハードウェア独自に、この INF ファイルを作成するため、**wd1000.cat** カタログ ファイルからこの INF ファイルを参照せず、また、Jungo のアブリオリでこの INF ファイルを署名できません。

windrivr6.sys の名前を変更し、対象のデバイスのデバイス独自の INF ファイルを作成する場合、ドライバのデジタル署名に関して、2 つのオプションがあります：

- 対象のドライバにデジタル署名をしない。このオプションを選択する場合、**windrivr6.inf** ファイル (または名前を変更したファイル) から **wd1000.cat** への参照を削除またはコメント アウトします。
- 対象のドライバを WHQL 認証に申請するか、Authenticode でドライバを署名する。
WinDriver\redist\windrvr6.sys の名前を変更すると、そのドライバのデジタル署名は無効になりますが、ドライバは WHQL 互換なので、WHQL テストに申請ができます。

対象のドライバをデジタル署名および認証するには、以下のステップを実行してください：

- Microsoft 社の WHQL のドキュメントで説明しているとおり、対象のドライバのカタログ ファイルを新規に作成します。新しいファイルは **windrivr6.sys** (または名前を変更したドライバ) とドライバのインストール時に使用するすべての INF ファイルの両方を参照します。
- 対象のドライバの INF ファイルの CatalogFile のエントリに新しいカタログ ファイルの名前を割り当てます。**(windrvr6.inf** ファイルの CatalogFile エントリを新しいカタログ ファイルへの参照するように変更し、対象のデバイス独自の INF ファイルに同じエントリを追加するか、または、CatalogFile エントリなどを含むシングル INF ファイルに **windrivr6.inf** ファイルと対象のデバイス独自の INF ファイルの両方を組み込みか、いずれかの方法を行うことができます。)
- WHQL 認証用のドライバを申請する場合、セクション 15.3.2.1 のガイドラインを参照してください。
- WHQL 認証用のドライバまたは Authenticode 署名用のドライバを申請してください。

多くの WinDriver ユーザーが既に WinDriver ベースのドライバでデジタル署名と認証の取得に成功しています。

15.3.2.1 WHQL 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。
- 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。
- 認証用のファイルを申請する前に、カタログ ファイルを新規に作成する必要があります。上記で説明したとおり [15.3.2]、新しいカタログ ファイルは対象のドライバと特定の INF ファイルをリストし、対象の INF ファイルからこのカタログ ファイルを参照します。

15.4 Windows XP Embedded の WinDriver のコンポーネント

Microsoft 社の Windows Embedded Studio の Target Designer ツールを使用して Windows XP Embedded イメージを作成する場合、対象のイメージに追加するコンポーネントを選択できます。

追加したコンポーネントは、イメージをロードする Windows XP Embedded ターゲット上で初期起動時に自動的にインストールされます。

Windows XP Embedded プラットフォームに、必要な WinDriver のファイル (**windrvr6.inf** ファイルと WinDriver カーネル ドライバ (**windrvr6.sys**)、対象の INF ファイル (Plug-and-Play デバイスの場合)、および WinDriver API DLL (**wdapi1000.dll** など) を自動的にインストールするには、関連する WinDriver のコンポーネントを作成し、それを対象の Windows XP Embedded イメージへ追加します。WinDriver はこの作業を簡素化するために既製のコンポーネントを提供しています

(**WinDriver\redist\xp_embedded\wd_component\windriver.sld**)。

このコンポーネントを使用するには、以下のステップを実行してください:

注意: 提供される **windriver.sld** コンポーネントは、同じディレクトリの **wd_files** ディレクトリ以下のファイルに依存します。従って、以下のガイドラインの手順以外、提供される **WinDriver\redist\xp_embedded\wd_component\wd_files** ディレクトリの名前を変更したり、コンテンツを編集しないでください。

1. Plug-and-Play デバイス (PCI / PCMCIA / USB) の場合 – **dev.inf** ファイルを編集:

windriver.sld コンポーネントは、**wd_files** ディレクトリの **dev.inf** ファイルに依存します。

WinDriver のインストールでは、対象の Windows 開発プラットフォームに、汎用的な

WinDriver\redist\xp_embedded\wd_component\wd_files\dev.inf ファイルが配布されます。以下のいずれかの方法で、対象のデバイスに合うようにこのファイルを編集してください:

- 汎用的な **dev.inf** ファイルを編集して対象のデバイスを記述します。少なくとも、テンプレートの [DeviceList] エントリを修正し、対象のデバイスのハードウエア タイプ、ベンダー ID、プロダクト ID を挿入する必要があります。たとえば、ベンダー ID 0x1111 とプロダクト ID

0x2222 を持つ PCI デバイスの場合、以下のようにになります:

```
"my_dev_pci"=Install, PCIEN_1111&DEV_2222
```

または

- DriverWizard [4.2 (3)] を使用して対象のデバイスの INF ファイルを作成し、ファイル名を **dev.inf** とするか、または対象のカードに合う WinDriver が拡張サポートを提供するチップセットの 1 つから INF ファイルを使用して、ファイル名を **dev.inf** に変更します。そして、**dev.inf** デバイス INF ファイルを **WinDriver\redist\xp_embedded\wd_component\wd_files** ディレクトリへコピーします。

非 Plug-and-Play デバイス (ISA) の場合 – WinDriver のコンポーネントから dev.inf のインストールを削除:

インストール ファイル

WinDriver\redist\xp_embedded\wd_component\wd_files\wd_install.bat の以下の行を削除またはコメントアウトします:

```
wdreg -inf dev.inf install
```

2. WinDriver のコンポーネントを Windows Embedded Component Database へ追加:

DBMgr (Windows Embedded Component Database Manager) を開きます。

Import をクリックします。

SLD ファイルとして WinDriver のコンポーネント –

WinDriver\redist\xp_embedded\wd_component\windriver.sld を選択し、**Import** をクリックします。

3. WinDriver のコンポーネントを Windows Embedded イメージへ追加:

Target Designer で対象のプロジェクトを開きます。

WinDriver のコンポーネントをダブルクリックし、対象のプロジェクトに追加します。

注意: 対象のプロジェクトのコンポーネントリストに以前のバージョンの WinDriver のコンポーネントが既に存在する場合、このコンポーネントを右クリックし、**Upgrade** を選択します。

依存性チェックを起動して対象のイメージをビルドします。

これらのステップの後、対象のイメージをロードするターゲットの Windows XP Embedded プラットフォームで初期起動時に WinDriver が自動的にインストールされます。

注意: WinDriver のカーネル モジュールの名前を変更した場合 [15.2]、提供されている **windriver.sld** を使用できません。名前を変更したドライバ用にコンポーネントをビルドするか、**wdreg** ユーティリティを使用してターゲットの Windows XP Embedded プラットフォームのドライバをインストールする必要があります。

第 16 章

PCI Express

16.1 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実行することができる 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。

16.2 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 セットを提供しています (詳細は、WDC_PciReadCfgXXX() 関数、WDC_PciWriteCfgXXX() 関数、低レベル WD_PciConfigDump() 関数の説明を参照)。

Linux および Windows Vista では、WinDirver の割込み処理 API は MSI (Message-Signaled Interrupts) と MSI-X (Extended Message-Signaled Interrupts) もサポートします。詳細は、セクション 9.2 を参照してください。

WinDriver ではまた、BMD (Bus Mastering DMA Validation Design) ファームウェアを搭載した Xilinx Virtex 5 PCI Express チップに対して、拡張サポートを提供します。WinDriver/xilinx/virtex5/bmd/ ディレクトリ以下のサンプルには、DMA と MSI 処理を含む、WinDriver の API を使用してチップと通信を行うためのライブラリ API とサンプル アプリケーションが含まれます。

WinDriver

ユーザーズ ガイド

2009 年 4 月 1 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/>

Copyright © Jungo Ltd. All Rights Reserved.

Translated by

米国 XLsoft Corporation
12K Mauchly
Irvine, CA 92618 USA
URL: <http://www.xlsoft.com/>
E-Mail: sales@xlsoft.com