JUNGO CONNECTIVITY LTD.

WinDriver

ユーザーズ ガイド

エクセルソフト株式会社

COPYRIGHT

Copyright (c) 2016 Jungo Connectivity Ltd. All Rigths Reserved.

Jungo Connectivity Ltd.

1st. Hamachshev street, POB 8190, Netanya 4250711, Israel
Phone (USA) +1 408 351 6854 (WorldWide) +972 9 970 8607
Fax (USA) +1 408 404 0202 (WorldWide) +972 9 777 0299

ご注意

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

WinDriver はイスラエル国 Jungo Connectivity Ltd. 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、Windows Vista、Windows 7、Windows 8、Windows Server 2012、Windows 8.1 および Window 10 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

〒108-0073 東京都港区三田3-9-9 森伝ビル6F
TEL 03-5440-7875　FAX 03-5440-7876
E-MAIL: xlsoftkk@xlsoft.com
Home Page: http://www.xlsoft.com/

Rev. 12.21 – 09/2016

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 12
1.4 まとめ 13
1.5 WinDriver の利点 13
1.6 WinDriver のアーキテクチャ 14
1.7 WinDriver がサポートするプラットフォーム 16
1.8 評価版 (Evaluation Version) の制限 16
1.9 WinDriver を使用してドライバを開発するには 16
1.9.1 Windows および Linux 16
1.9.2 Windows CE 17
1.10 WinDriver ツールキットの内容 17
1.10.1 WinDriver のモジュール 17
1.10.2 ユーティリティ 18
1.10.3 拡張サポートを提供するチップセット 18
1.10.4 サンプル 18
1.11 WinDriver で作成したドライバを配布できますか 19

第 2 章 デバイスドライバの理解 20

2.1 デバイスドライバの概要 20
2.2 機能によるドライバの分類 20
2.2.1 モノリシックドライバ 20
2.2.2 レイヤードドライバ 21
2.2.3 ミニポートドライバ 22
2.3 OS によるドライバの分類 22
2.3.1 WDM ドライバ 22
2.3.2 Unix デバイスドライバ 23
2.3.3 Linux デバイスドライバ 23
2.4 ドライバのエントリー ポイント 23
2.5 ハードウェアとドライバとの関連付け 23

2.6 ドライバとの通信 24

第 3 章 WinDriver USB の概要 25

3.1 USB の概要 25
3.2 WinDriver USB の利点 25
3.3 USB のコンポーネント 26
3.4 USB デバイスのデータ フロー 26
3.5 USB データ交換 27
3.6 USB データ転送タイプ 28
3.6.1 コントロール転送 (Control Transfer) 28
3.6.2 アイソクロナス転送 (Isochronous Transfer) 28
3.6.3 インタラプト転送 (Interrupt Transfer) 28
3.6.4 バルク転送 (Bulk Transfer) 29
3.7 USB 設定 29
3.8 WinDriver USB 30
3.9 WinDriver USB のアーキテクチャ 32

第 4 章 WinDriver のインストール 34

4.1 動作環境 34
4.1.1 Windows 34
4.1.2 Windows CE 34
4.1.3 Linux 34
4.2 WinDriver のインストール 35
4.2.1 Windows にインストールするには 35
4.2.2 WinDriver CE のインストール 38
4.2.3 Linux に WinDriver をインストールするには 40
4.3 アップグレード版のインストール 43
4.4 インストールの確認 43
4.4.1 Windows および Linux コンピュータの場合 43
4.4.2 Windows CE コンピュータの場合 43
4.5 WinDriver をアンインストールするには 44
4.5.1 Windows WinDriver をアンインストールするには 44
4.5.2 Linux から WinDriver をアンインストールするには 45

第 5 章 DriverWizard 47

5.1 DriverWizard の概要 47
5.2 DriverWizard の使い方 48
5.2.1 自動コード生成 59
5.2.2 生成されたコードをコンパイルする 60

第 6 章 ドライバの作成 61

6.1 WinDriver でデバイスドライバをビルドするには 61

6.2 DriverWizard を使わずにドライバを記述するには 61
6.2.1 必要な WinDriver ファイルのインクルード 61
6.2.2 コードの作成: PCI / CardBus / PCMCIA ドライバの場合 63
6.2.3 コードの作成: USB ドライバの場合 63
6.2.4 設定とコードのビルド 64
6.3 Windows CE で開発を行うには 64

第 7 章 デバッグ 66

7.1 ユーザーモード デバッグ 66
7.2 Debug Monitor 66
7.2.1 wddebug_gui ユーティリティ 66
7.2.2 wddebug ユーティリティ 69

第 8 章 特定のチップ セットの拡張サポート 73

8.1 概要 73
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 73

第 9 章 実行に当たっての問題 74

9.1 DMA の実行 74
9.1.1 スキャッタ / ギャザー (Scatter / Gather) DMA の実装 75
9.1.2 Contiguous Buffer (連続バッファ) DMA の実装 77
9.2 割り込み処理 80
9.2.1 割り込み処理の概要 81
9.2.2 WinDriver の割り込み処理手順 82
9.2.3 Window 7 以降で非 Plug-and-Play ハードウェアの IRQ の登録 83
9.2.4 ハードウェアがサポートする割り込みタイプの決定 83
9.2.5 PCI カードの割り込みタイプの決定 84
9.2.6 カーネルモードの割り込み転送コマンドの設定方法 84
9.2.7 WinDriver の MSI / MSI-X 割り込み処理 86
9.2.8 ユーザーモードの WinDriver 割り込み処理のコード例 87
9.2.9 Windows CE の割り込み 89
9.3 バイト オーダー 90
9.3.1 エンディアンネスとは 90
9.3.2 WinDriver のバイト オーダー マクロ 91
9.3.3 PCI ターゲット アクセスのマクロ 91
9.3.4 PCI マスター アクセスのマクロ 92
9.4 複数プロセス間のバッファ共有 92
9.5 64 ビット OS のサポート 93
9.5.1 64 ビット アーキテクチャのサポート 93
9.5.2 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 93
9.5.3 64 ビットおよび 32 ビットのデータ型 94

第 10 章 USB 転送 96

10.1 概要 96
10.2 USB コントロール転送 96
10.2.1 USB コントロール転送の概要 96
10.2.2 WinDriver でコントロール転送を行う 99
10.3 機能 USB データ転送 101
10.3.1 機能 USB データ転送の概要 101
10.3.2 シングル ブロッキング転送 102
10.3.3 ストリーミング データ転送 102

第 11 章 パフォーマンスの向上 104

11.1 概要 104
11.1.1 パフォーマンスを向上するためのチェックリスト 105
11.2 ユーザーモード ドライバのパフォーマンスの向上 105
11.2.1 メモリ マップの領域への直接アクセス 106
11.2.2 ブロック転送および複数の転送のグループ化 106
11.2.3 64 ビット データ転送を行う 106

第 12 章 Kernel PlugIn について 108

12.1 Kernel PlugIn の概要 108
12.2 Kernel PlugIn を作成する前に 108
12.3 期待される効果 108
12.4 開発プロセスの概要 108
12.5 Kernel PlugIn の構造 109
12.5.1 構造の概要 109
12.5.2 WinDriver のカーネルと Kernel Plugin の相互作用 110
12.5.3 Kernel Plugin コンポーネント 110
12.5.4 Kernel PlugIn イベント シーケンス 110
12.6 Kernel PlugIn の仕組み 113
12.6.1 Kernel PlugIn ドライバの作成に必要な条件 113
12.6.2 Kernel PlugIn の実装 114
12.6.3 Kernel PlugIn ドライバの生成されたコードとサンプル コード 119
12.6.4 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 120
12.6.5 Kernel PlugIn での割り込み処理 122
12.6.6 メッセージの受け渡し 125

第 13 章 Kernel PlugIn の作成 126

13.1 Kernel PlugIn が必要かどうかを確認する 126
13.2 ユーザーモードのソース コードを用意する 126
13.3 Kernel PlugIn プロジェクトの新規作成 127
13.4 Kernel PlugIn へのハンドルをオープン 127

13.5 Kernel PlugIn での割り込み処理の設定 128
13.6 Kernel PlugIn での I/O 処理の設定 129
13.7 Kernel PlugIn ドライバのコンパイル 129
13.7.1 Windows で Kernel PlugIn ドライバをコンパイル 129
13.7.2 Linux でKernel PlugIn ドライバをコンパイル 132
13.8 Kernel PlugIn ドライバのインストール 134
13.8.1 Windows の場合 134
13.8.2 Linux の場合 134

第 14 章 ドライバの動的ロード 135

14.1 なぜ動的にロード可能なドライバが必要なのか 135
14.2 Windows の動的ドライバ ロード 135
14.2.1 WDREG ユーティリティ 135
14.2.2 windrvr1221.sys INF ファイルの動的ロード / アンロード 138
14.2.3 Kernel PlugIn ドライバを動的にロード / アンロード 138
14.3 Linux の動的ドライバ ロード 139
14.3.1 Kernel PlugIn ドライバを動的にロード / アンロード 139
14.4 Windows CE の動的ドライバ ロード 140

第 15 章 ドライバの配布 141

15.1 WinDriver の有効なライセンスを取得するには 141
15.2 Windowsの場合 141
15.2.1 配布パッケージの用意 141
15.2.2 ターゲット コンピュータにドライバをインストール 142
15.2.3 ターゲット コンピュータに Kernel PlugIn をインストール 144
15.3 Windows CE の場合 145
15.3.1 新規の Windows CE プラットフォームへの配布 145
15.3.2 Windows CE コンピュータへの配布 146
15.4 Linux の場合 146
15.4.1 配布パッケージの用意 146
15.4.2 ターゲットで WinDriver のドライバ モジュールをビルドおよびインストール 149
15.4.3 ターゲットで Kernel PlugIn ドライバ モジュールをビルドおよびインストール 150
15.4.4 ユーザーモードのハードウェア コントロール アプリケーションまたは共有オブジェクトをインストール 151

第 16 章 ドライバのインストール - 高度な問題 152

16.1 Windows INF ファイル 152
16.1.1 なぜ INF ファイルを作成する必要があるのか 152
16.1.2 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 152
16.1.3 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 153
16.2 WinDriver カーネル ドライバの名前変更 154
16.2.1 Windows ドライバの名前変更 154

16.2.2 Linuxドライバの名前変更 156
16.3 Windows のデジタル ドライバの署名と認定 157
16.3.1 概要 157
16.3.2 WinDriver ベースのドライバのドライバ署名と認定 158
16.4 Windows XP Embedded の WinDriver のコンポーネント 159

第 17 章 PCI Express 162

17.1 PCI Express の概要 162
17.2 WinDriver PCI Express 163

図表

図 1.1: WinDriver アーキテクチャ 15
図 2.1: モノリシックドライバ 21
図 2.2: レイヤードドライバ 21
図 2.3: ミニポートドライバ 22
図 3.1: USB エンドポイント 27
図 3.2: USB パイプ 28
図 3.3: デバイス ディスクリプタ 29
図 3.4: WinDriver USB アーキテクチャ 32
図 4.1: ライセンスの登録 37
図 5.1: WinDriver のプロジェクトを開く、または新規作成 48
図 5.2: デバイスの選択 49
図 5.3: DriverWizard INF ファイル情報 50
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) 51
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) 52
図 5.7: PCI のリソース画面 53
図 5.8: レジスタの定義 54
図 5.9: メモリおよび I/O の Read / Write 54
図 5.10: 割り込みの Listen (確認) 54
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 55
図 5.12: USB デバイスのインターフェースの選択 55
図 5.13: USB コントロール転送 56
図 5.14: パイプの確認 57
図 5.15: パイプへの書き込み 57
図 5.16: コード生成のオプション 58
図 5.17: ドライバ オプションの選択 58
図 7.1: Debug Monitor の起動 67
図 7.2: Debug Options の設定 67
図 7.3: wddebug Windows CE ログ開始メッセージ 72
図 7.4: wddebug Windows CE ログ停止メッセージ 72
図 10.1: USB データ交換 96
図 10.2: USB のリードとライト 97
図 10.3: カスタム要求 100
図 10.4: 要求一覧 100
図 10.5: USB 要求ログ 101
図 11.1: KernelPlugIn の構造 109
図 11.2: Kernel PlugIn なしでの割り込みの処理 123

図 11.3: Kernel PlugIn ありでの割り込み処理.....124

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 10 / 8.1 / Server 2012 R2 / 8 / Server 2012 / 7 /Server 2008 R2 / Vista / Server 2008 / Server 2003 / XP 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 および USB です。PCMCIA は Windows でのみサポートされています。CardBus / ISA および EISA は、Windows、Windows CE (Windows Mobile を含む) および Linux でサポートされています。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、すべての PCI / PCMCIA / CardBus / ISA / EISA / CompactPCI / PCI Express および USB チップセット向けのドライバ開発をサポートします。また、特定のチップセットに対しては拡張サポートを提供しています。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 12 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 12 章 および第 13 章 を参照してください。

WinDriver およびその他の開発ツールに関する最新情報を入手するには、エクセルソフト(株) のホームページ (http://www.xlsoft.com/jp/products/windriver/products.html) および開発元の Jungo Connectivity 社のホームページ (http://www.jungo.com/st/products/windriver/) 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 MS Visual Studio、MS eMbedded Visual C++、MS Platform Builder C++、GCC、Windows GCC などの最適なコンパイラまたは開発環境を使って「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 10 / 8.1 / Server 2012 R2 / 8 / Server 2012 / 7 / Server 2008 R2 / Vista / Server 2008 / Server 2003 / XP, Embedded Windows 10 / 8.1 / 8 / 7 / XP、Windows CE (別名 Windows Embedded Compact) 4.x – 8.x (Windows Mobile を含む) 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、グラフィカルな診断ツールで、ドライバ コードを記述する前に、わずか数クリックで、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します。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ドライバのソースコード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 11 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 を使用しなくても最適なパフォーマンスを達成できます。セクション [9.2.98] 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。

1.4 まとめ

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。

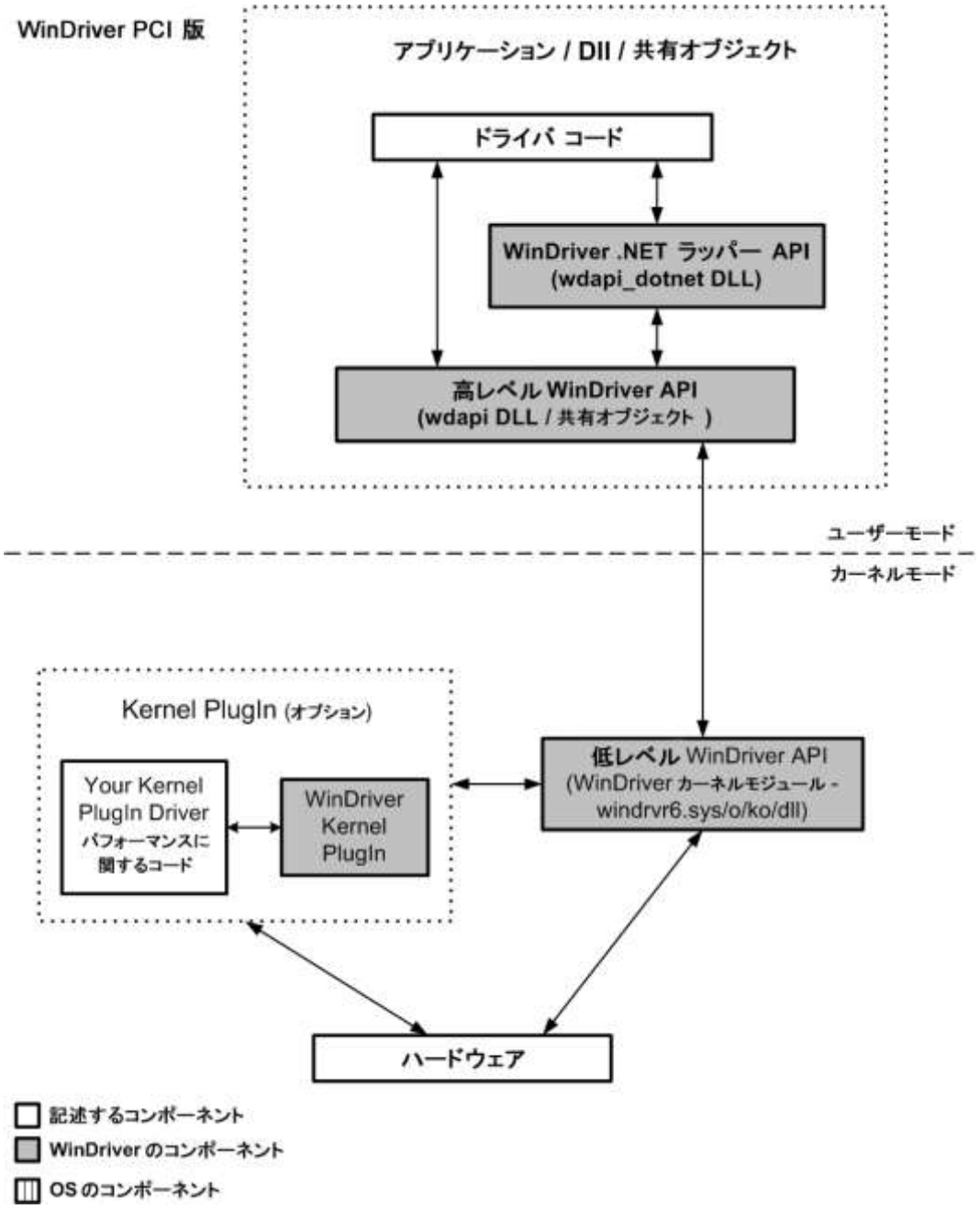
これで、すべての対応するプラットフォームから新しいハードウェアにアクセスするアプリケーションを作成できます。(コードは Windows 10 / 8.1 / Server 2012 R2 / 8 / Server 2012/ 7 / Server 2008 R2 / Vista / Server 2008 / Server 2003 / XP プラットフォームでバイナリ互換性があります。そのため、これらの OS 間でドライバを移植する場合は再ビルドする必要はありません。)

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# 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。
- 汎用的な PCI チップセットのサポートに加え、特定の PCI チップセットを拡張サポートします。そのため、開発者は PCI チップセットの詳細を特に知る必要はありません。
- 汎用的な USB チップセットのサポートに加え、特定の USB チップセットを拡張サポートします。そのため、開発者は USB チップセットの詳細を特に知る必要はありません。
- 作成されるアプリケーションは Windows 10 / 8.1 / Server 2012 R2 / 8 / Server 2012 / 7 / Server 2008 R2 / Vista / Server 2008 / Server 2003 / XP でバイナリ互換です。
- 作成されるアプリケーションはWindows 10 / 8.1 / Server 2012 R2 / 8 / Server 2012 / 7 / Server 2008 R2 / Vista / Server 2008 / Server 2003 / XP 、Embedded Windows 10 / 8.1 / 8 / 7 / XP、Windows CE (別名 Windows Embedded Compact) 4.x – 8.x (Windows Mobile を含む) および Linux でソースコード互換です。
- MS Visual Studio、MS eMbedded Visual C++、MS Platform Builder C++、GCC、Windows GCC などの最適なコンパイラまたは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# の詳細なサンプルが用意されています。
- WinDriver Windows のドライバーは Microsoft 社のWindows 認定プログラムと互換性があります。
- 2 ヶ月間の無料テクニカルサポート (インストール、ライセンス、配布に関する質問)。
- 作成したドライバを無料で使用、配布できます。

1.6 WinDriver のアーキテクチャ



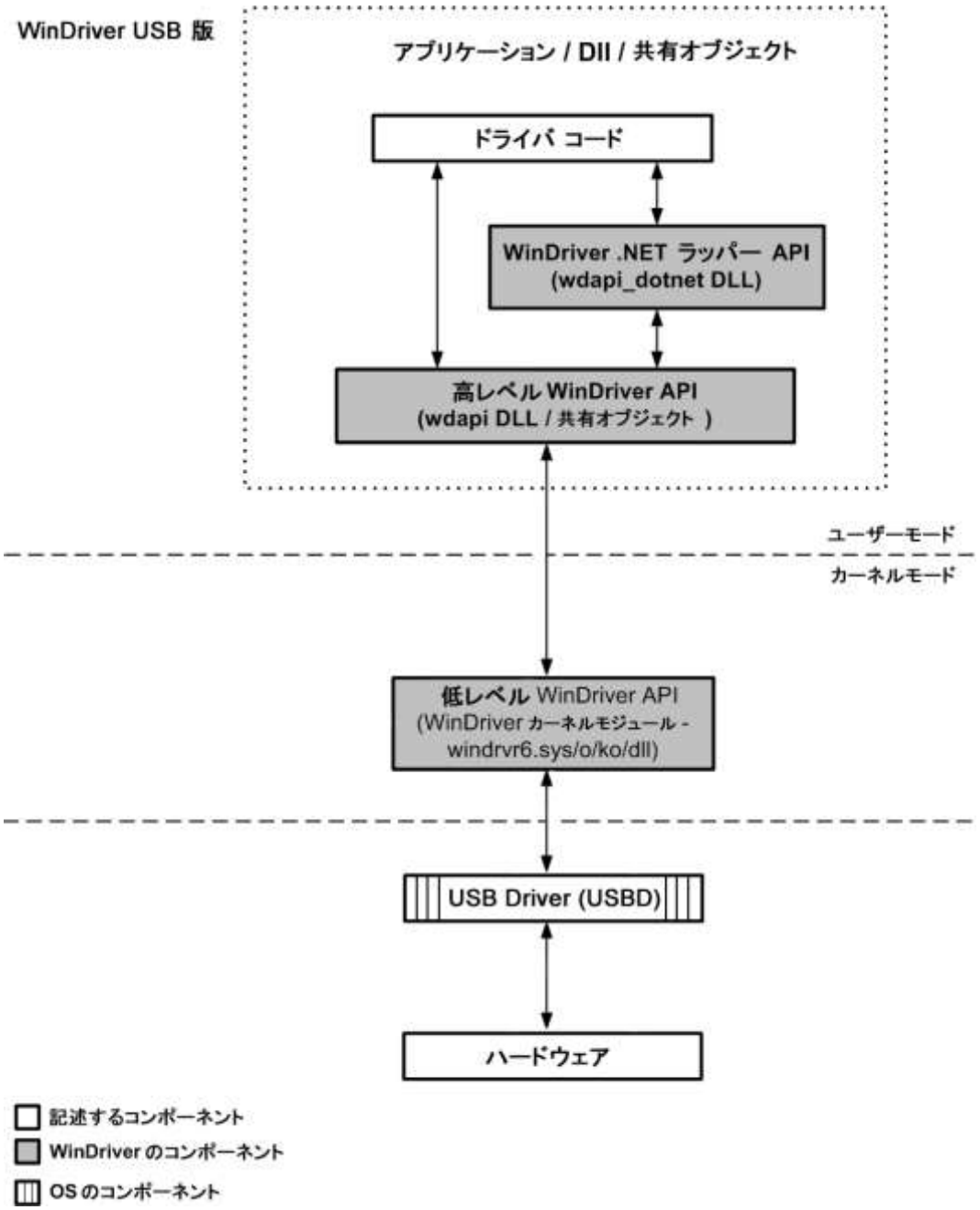


図 1.1: WinDriver アーキテクチャ

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

WinDriver は、ユーザーモードで実行されてもパフォーマンスにあまり影響しないように設計されています。しかし、ハードウェアのドライバとは、ユーザーモードでは達成し得ないパフォーマンスを要求される場合があります。このような場合、ユーザーモードで開発したコードからパフォーマンスが必要なモジュール (割り込みハンドラ等) のコードを変更せずに WinDriver の Kernel PlugIn に移動します。これにより、WinDriver カーネルがカーネル モードでこのモジュールを呼び出し、パフォーマンスを向上させます。そのため、ユーザーモードで容易にドライバの開発、デバッグを行い、必要な部分のパフォーマンスを向上できます。Kernel

PlugIn に関する詳細は第 12 章を参照してください。Windows CE の場合、ユーザーモードとカーネル モードの境界がないため、Kernel PlugIn を使用しなくても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 10 / 8.1 / Server 2012 R2 / 8 / Server 2012 / 7 / Server 2008 R2 / Vista / Server 2008 / Server 2003 / XP 、Embedded Windows 10 / 8.1 / 8 / 7 / XP – これ以降、"**Windows**" と呼びます。
- Windows CE (別名 Windows Embedded Compact) 4.x - 8.x (Windows Mobile を含む) – これ以降、"**Windows CE**" と呼びます。
- Linux

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 – ターゲットのプラットフォームでコードをリコンパイルするだけです。またソースコードは Windows 10 / 8.1 / Server 2012 R2 / 8 / Server 2012 / 7 / Server 2008 R2 / Vista / Server 2008 / Server 2003 / XP 間ではバイナリ互換があります。WinDriver の実行形式をリコンパイルせずにバイナリ互換があるプラットフォーム間で移植が可能です。
これらのサポートする OS の 1 つに対してのみ作成したコードであっても、WinDriver を使用することにより、コードの変更を行わずに他の OS に柔軟に移植ができます。

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、再ロードする必要があります。
- Windows の評価版はインストール後、30 日間使用できます。
- 詳細は、別冊 PDFの「評価版 (Evaluation Version) の制限」の章を参照してください。

1.9 WinDriver を使用してドライバを開発するには

1.9.1 Windows 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. 雛型となるコードを生成するか、または WinDriver のサンプルをドライバ アプリケーションの雛型とします。各チップセット特有の拡張サポートに関する詳細は第 8 章 を参照してください。
3. DriverWizard が生成するコードを修正してアプリケーションに必要な機能を作成してください。
4. ユーザーモードでドライバのテストやデバッグを行います。
5. コードにパフォーマンス的にクリティカルな部分が含まれている場合、第 11 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. ハードウェアの仕様にあわせて、MS eMbedded Visual C++ でこのコードを修正します。MS Platform Builder を使用している場合、対象のプロジェクトのソリューションへ 生成された *.pbp を挿入します。
5. Windows CE プラットフォームを組み込んだターゲットで対象のドライバをテストします。

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

- 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ベーシック テクニカル サポート (問い合わせフォーム: https://www.xlsoft.com/jp/qa/)
- WinDriver のモジュール
- ユーティリティ
- 拡張サポートを提供するチップセット用のサンプルとラッパー API

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:
  - **`windrvr.h`:** WinDriver API の宣言および定義。
  - **`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。
  - **`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) ライブラリの宣言および定義。
  - **`windrvr_int_thread.h`:** 割り込み処理を簡略化するラッパー関数の定義。
  - **`windrvr_events.h`**: イベント処理および PnP 通知を実装する関数の定義。
  - **`utils.h`:** 一般的なユーティリティ関数の宣言。
  - **`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 の宣言。
- DriverWizard (**`WinDriver¥wizard¥wdwizard`**): ハードウェアを診断し、対象のドライバのコードを簡単に生成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)。

- Debug Monitor:ドライバの実行中にデバッグ情報を収集するデバッグ ツール。グラフィカルなアプリケーション (**WinDriver¥util¥wddebug_gui**) とコンソールモードのアプリケーション (**WinDriver¥util¥wddebug**) の両方を利用可能です。コンソールモードのバージョンでは、コマンドライン プロンプトを持たない Windows CE プラットフォームで GUI 実行もサポートします。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 12 章 を参照してください。
- 本書: WinDriver マニュアル。いくつかの形式で **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。

1.10.3 拡張サポートを提供するチップセット

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。
- Altera Qsys デザイン - **WinDriver¥altera¥qsys_design** に保存されています。
- Xilinx Bus Master DMA (BMD) デザイン - **WinDriver¥xilinx¥bmd_design** に保存されています。

Xilinx BMD と Altera Qsys デザイン用に、関連する拡張サポートの API を使用するカスタマイズしたドライバ コードを生成するオプションもあります。

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)

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。詳細についてはライセンス同意書 (**WinDriver¥docs¥wd_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 ドライバの手引きを紹介し、デバイス ドライバの構造的な要素を説明します。

注意: WinDriver を使用すれば、ドライバ開発の内部構造を意識する必要はありません。WinDriver のシンプルな 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ます。ハードウェアにパラメータを設定したり、カーネルからデバイスへデータ転送したり、デバイスから戻ってきたデータをカーネルへ渡したり、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。

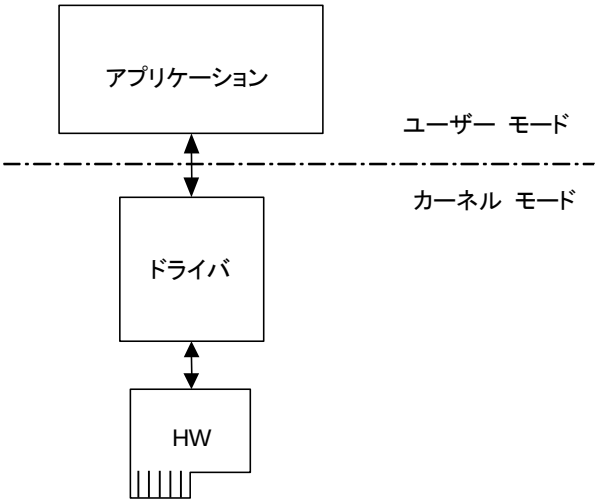


図 2.1: モノリシック ドライバ

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 OS 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化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。

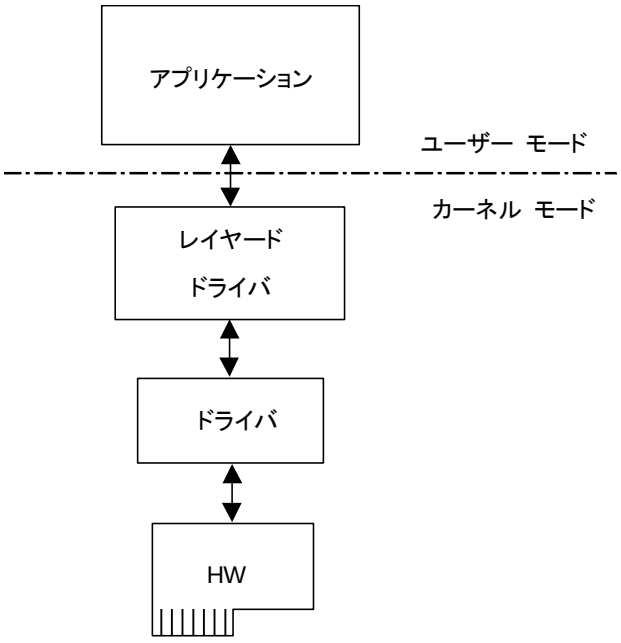


図 2.2: レイヤード ドライバ

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 XP および以降の OS でサポートされています。

Windows XP および以降の OS 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、Windows 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 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。

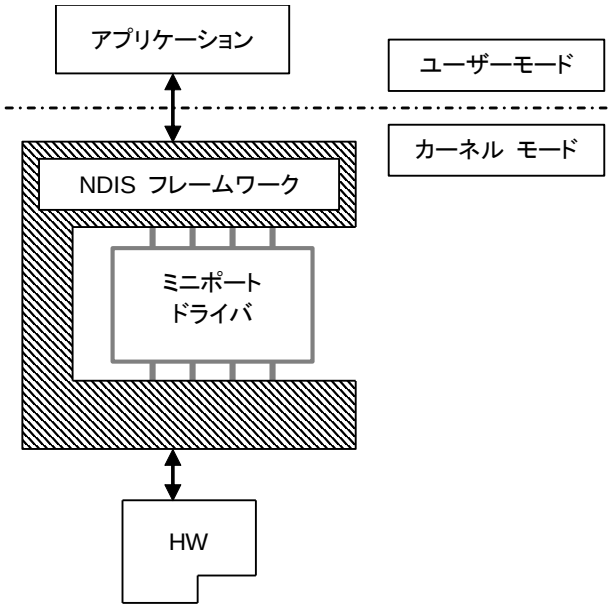


図 2.3: ミニポートドライバ

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 OS のカーネル モード ドライバです。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 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 レイヤのレイヤード ドライバとして知られています。

2.3.3 Linux デバイス ドライバ

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

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

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

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

2.4 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 のコンソール 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()` ルーチンの役割です。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との関連付け

OS がデバイスをそのドライバにどのように関連付けるかは、OS によって異なります。Windows の場合、INF ファイルによって、ハードウェア ドライバとの関連付けを行います。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。この関連付けを `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確認し、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、ハードウェア ドライバとの関連付けを `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

ハードウェアを操作するドライバとユーザーモード アプリケーション間の通信は、カスタムの OS API (Application Program Interface) を使用して、各オペレーティング システムで実装が異なります。

Windows、Windows CE、および Linux では、OS のファイル アクセス API を使用してドライバへのハンドルをオープンし (例えば、Windows では `CreateFile()` 関数、または Linux では `open()` 関数を使用して)、関連する OS のファイル アクセス関数へハンドルを渡すことでデバイスからの read およびデバイスへの write を行います (例えば、Windows では `ReadFile()` および `WriteFile()` 関数、または Linux では `read()` および `write()` 関数)。

アプリケーションは、その目的に応じてカスタムの OS API を使用して、I/O コントロール (IOCTL) を呼び出してドライバへのリクエストを送信します (例えば、Windows では `DeviceIoControl()`、Linux では `ioctl()`)。
IOCTL を呼び出してドライバとアプリケーション間で渡されたデータをカスタムの OS メカニズムを使用してカプセル化します。例えば、Windows では、IRP (I/O Request Packet) 構造体によってデータを渡し、I/O Manager でデータをカプセル化します。

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ことなどがその目標として掲げられました。

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

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

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

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

3.2 WinDriver USB の利点

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

- 最大限に簡単に使用できる外部接続。
- 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。
- 同じケーブルでアイソクロナス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 (High-Speed) を公式にサポートしている OS では最大 480 Mb/s 、USB 1.1 (Full-Speed) 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 (アイソクロナス転送は、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。
- カスタム USB HID デバイスに対する特定のライブラリのサポート

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 つのホストが直接通信することはできませんし、または 2 つの USB デバイスと通信することもできません (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経由して転送されます。

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 (Low、Full、High-Speed) はすべて、1 つの双方向コントロール エンドポイント (エンドポイント 0) と 15 個の一方向エンドポイントをサポートします。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、帯域要求、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、インタラプト、バルクおよびアイソクロナス転送を処理し、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。

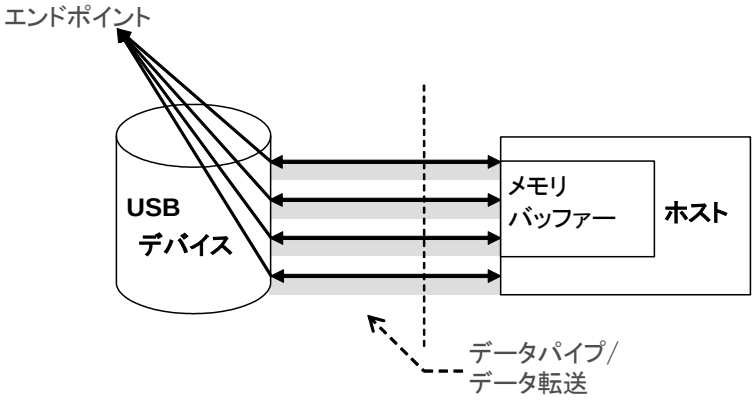


図 3.1: USB エンドポイント

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、インタラプト転送、アイソクロナス転送の 3 種類のデータ転送があります。
- コントロール交換は、デバイスを識別し、設定条件を決定して、デバイスを設定するのに使用されます。デバイス上の他のパイプのコントロールを含む、その他のデバイス特有の目的にも使用することができます。
  コントロール交換はコントロール パイプ (一般的にはデフォルトで、Pipe 0 です) を経由して転送されます。コントロール交換は、セットアップ ステージ (セットアップ パケットはホストからデバイスに送られます)、オプショナル データ ステージ、およびステータス ステージから構成されます。

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

図 3.2: USB パイプ

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

3.6 USB データ転送タイプ

USB デバイス (機能) は、ホストのメモリ バッファとデバイスのエンド ポイントの間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。Low-Speed、Full-Speed、および High-Speed 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 パイプは双方向のパイプで、データは両方向に流れることができます。コントロール転送にはまた、頑強なエラー検出、エラー リカバリ、再発信する機能が実装されており、これはドライバと独立してリトライを行います。コントロール エンドポイントの最大パケット サイズは、Low-Speed デバイスでは 8 バイトのみ、Full-Speed デバイスでは 8、16、32、または 64 バイト、High-Speed デバイスでは 64 バイトのみです。

3.6.2 アイソクロナス転送 (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。Full-Speed および High-Speed デバイスでこの転送タイプを使用し、Low-Speed デバイスでは使用しません。アイソクロナス転送は定期的で連続的です。アイソクロナスパイプは単方向で、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。双方向のアイソクロナス通信の場合、各方向に 1 つずつ、2 つのアイソクロナスパイプが必要です。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 インタラプト転送 (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インタラプト転送を使用します。この転送タイプは、Low-Speed、Full-Speed、High-Speed デバイスで使用できます。インタラプト転送タイプは、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、アイソクロナス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、Low-Speed デバイスでは 8 バイト以下、Full-Speed デバイスでは 64 バイト以下、High-Speed 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、Full-Speed および High-Speed デバイスで使用できますが、Low-Speed 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。他の転送に USB の帯域幅を使用してない場合、システムはその帯域幅をバルク転送に使用します。他のストリーム パイプ (アイソクロナスとインタラプト) 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、Full-Speed デバイスでは 8、16、32、または 64 バイト、High-Speed 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で属性を提供します。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)。

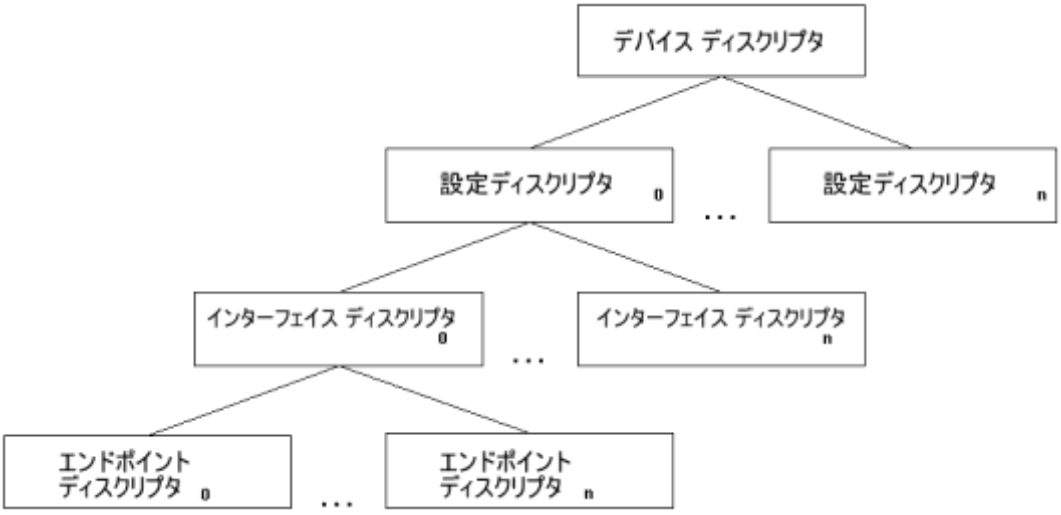


図 3.3: デバイス ディスクリプタ

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、設定でグループ化したインターフェイスの数と設定の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。アイソクロナス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 10 / 8.1 / Server 2012 R2 / 8 / Server 2012 / 7 / Server 2008 R2 / Vista / Server 2008 / Server 2003 / XP - 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 10 / 8.1 / Server 2012 R2 / 8 / Server 2012 / 7 / Server 2008 R2 / Vista / Server 2008 / Server 2003 / XP、Windows CE (別名 Windows Embedded Compact) 4.x - 8.x (Windows Mobile を含む) 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/jp/products/windriver/products.html)。

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、または .NET 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 16.1.1 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 (MS Visual Studio、MS eMbedded Visual C++、MS Platform Builder C++、GCC、Windows GCC など) を使用して行えます。
WinDriver による USB 転送の実装に関する詳細は、第 9 章を参照してください。

3.9 WinDriver USB のアーキテクチャ

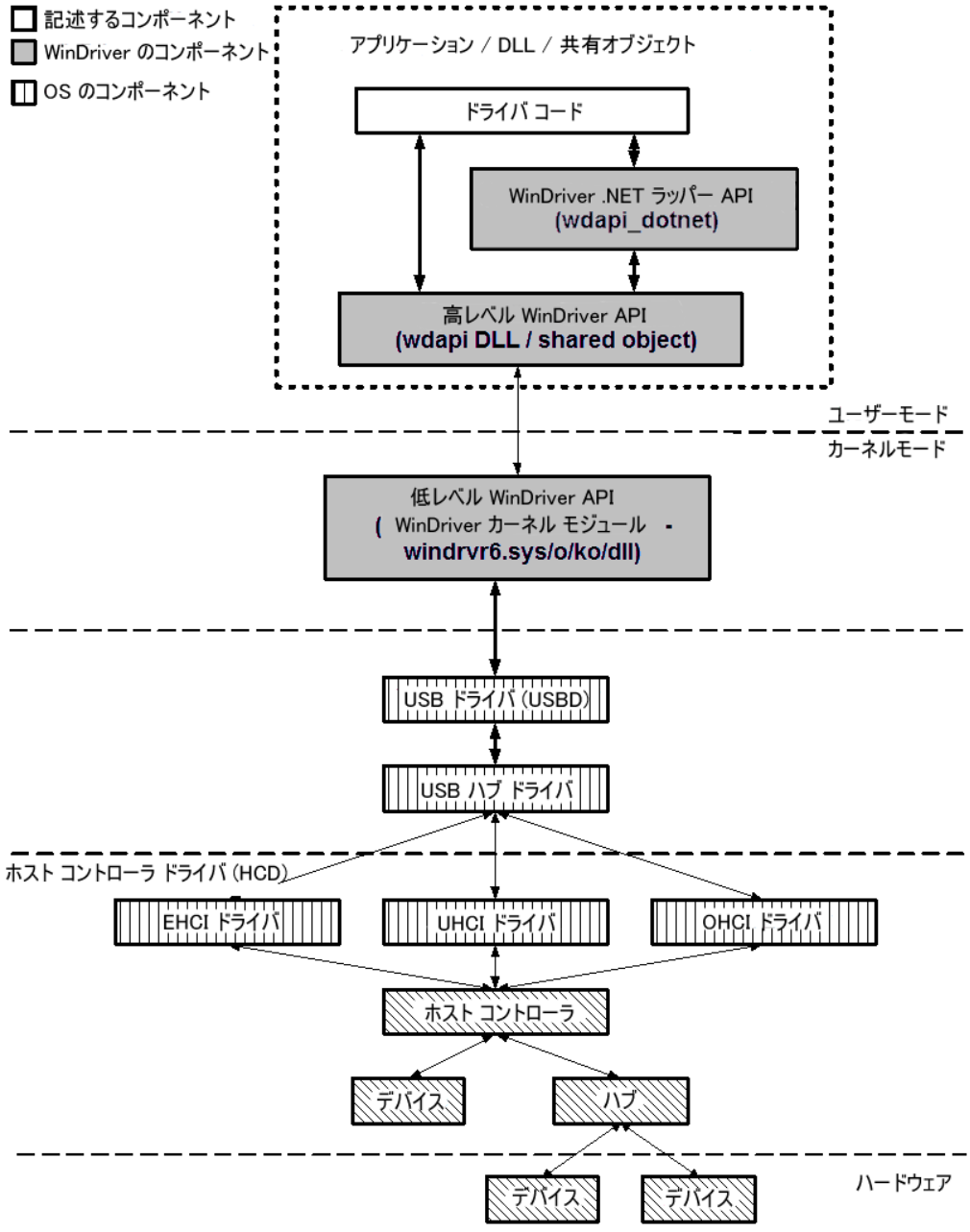


図 3.4: WinDriver USB アーキテクチャ

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。

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紹介します。

4.1 動作環境

4.1.1 Windows

- x86 32 ビットまたは 64 ビット (x64: Intel EM64T または AMD64) プロセッサ
- Cまたは .NET をサポートする開発環境
- Windows XP の場合 Service Pack 2 以上

4.1.2 Windows CE

- Windows CE (別名 Windows Embedded Compact) 4.x – 8.x (Windows Mobile を含む) が起動する x86 / ARM ターゲット プラットフォーム
  Windows CE 4.x – 7.x (Windows Mobile を含む) が起動する MIPS ターゲット プラットフォーム
- Windows 10 / 8.1 / Server 2012 R2 / 8 / Server 2012 / 7 / Server 2008 R2 / Vista / Server 2008 / Server 2003 / XP ホスト開発プラットフォーム
- 開発環境:
  Windows CE 4.x – 5.x (Windows Mobile を含む) の場合:
  対応するターゲット SDK と Microsoft eMbedded Visual C++ または Microsoft Visual Studio 2005 / 2008
  または
  ターゲット プラットフォームに対応する BSP (Board Support Package) と Microsoft Platform Builder
  Windows CE 6.x の場合:
  対応するターゲット SDK または Windows CE 6.0 plugin と Microsoft Visual Studio 2005 / 2008
  Windows CE 7.x の場合:
  Windows CE 7 plugin と Microsoft Visual Studio 2008
  Windows CE 8.x の場合:
  Application Builder for Windows Embedded Compact 2013 plugin と Microsoft Visual Studio 2012 またはそれ以降

4.1.3 Linux

- Linux カーネル 2.6.x またはそれ以降と以下のプロセッサ アーキテクチャ:
  x86 32 ビット プロセッサ
  または
  x86 64 ビット AMD64 または Intel EM64T (**x86_64**) プロセッサ
  または
  PowerPC 32 ビットまたは 64 ビット (**ppc64**)
- GCC コンパイラ

> **注意:** GCC コンパイラは起動中の Linux カーネルと同じバージョンをご使用ください。

- C をサポートする 32 ビットまたは 64 ビット開発環境
- 開発環境 PC: **glibc2.14.x** (またはそれ以降)
- GUI WinDriver アプリケーション (DriverWizard、Debug Monitor など) の起動用に以下のライブラリが必要:
  libstdc++.so.6
  libpng12.so.0
  libQtGui.so.4
  libQtCore.so.4
  libQtNetwork.so.4

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 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 にインストールするには、システムの管理者権限のあるユーザーで行う必要があります。

1. WinDriver CD を CD-ROM ドライブに挿入します (WinDriver CD からインストールせずに、ダウンロードした WinDriver をインストールする場合は、ダウンロードしたインストール ファイル (**WDxxx.EXE**、**xxx** はバージョン番号。例: WD1221.EXE) をダブルクリックして、手順 3 に進んでください)。
2. インストール プログラムが自動的に起動します。自動的に起動しない場合は、**WDxxx.EXE** ファイルをダブルクリックしてください。[Install WinDriver] ボタンをクリックします。
3. 画面に表示されるライセンス同意書をお読みください。[Yes] を選択してライセンスに同意してください。
4. WinDriver をインストールする場所を選択します。
5. [**Setup Type**] ダイアログボックスで、次のいずれかを選択します。
   - **Typical** – すべての WinDriver モジュール (WinDriver ツールキットと特定チップセット用の API) をインストールします。
   - **Compact** – WinDriver ツールキットだけをインストールします。
   - **Custom** – インストールする WinDriver のモジュールを選択します。
6. インストールに必要なファイルのコピーが完了すると、クイック スタート ガイドを表示するか選択します。
7. セットアップを完了したら、コンピュータを再起動してください。

注意: WinDriver のインストールでは、**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 [第 5 章] でコードを生成する際には、この変数を使用します - 生成したコードを保存するデフォルトのディレクトリで、生成された project / make ファイルの include パスに使用します。サンプルの Kernel PlugIn プロジェクトおよび makefile でも、この変数を使用します。従って、WinDriver のインストール後、WinDriver のディレクトリの名前 / 場所を変更する場

合、WD_BASEDIR 環境変数の値を変更し、新しい WinDriver のディレクトリの場所を指すように設定する必要があります。以下の手順で、WD_BASEDIR の値を変更できます:

1. システムのプロパティダイアログを開きます:
   [スタート] - [設定] - [コントロール パネル] - [システム]
2. [詳細設定] タブで、[環境変数] ボタンをクリックします。
3. [システム環境変数] ボックスで、WD_BASEDIR 変数を選択し、[編集] ボタンをクリックするか、変数をダブル クリックします。
4. [システム変数の編集] ダイアログで、[変数値] を新しい WinDriver のディレクトリのフルパスに置き換えて、[OK] をクリックし、[環境変数] ダイアログで [OK] ボタンをクリックし、 更に、[システムのプロパティ] ダイアログで、[OK] ボタンをクリックします。

ERROR_FILE_NOT_FOUND エラーでインストールに失敗する場合、Windows のレジストリの **HKEY_LOCAL_MACHINE¥SOFTWARE¥Microsoft¥Windows¥CurrentVersion** に **RunOnce** キーが存在することをご確認ください。Windows Plug-and-Play で INF ファイルを使用してドライバを正しくインストールする際にこのレジストリ キーが必要です。**RunOnce** キーがない場合、作成し、再度 INF ファイルのインストールをお試しください。

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. [Help] メニューから [Licensing Information] を選択して、[Licensing 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、[Deactivate license] 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. 評価期間中に WinDriver を使用して作成したソース コードを登録するには、PCI 版の場合、`WDC_DriverOpen()` 関数のセクションを参照してください。デフォルトで使用する `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、`WD_License()` 関数のセクションを参照してください。USB 版の場合、`WDU_Init()` 関数のセクションを参照してください。

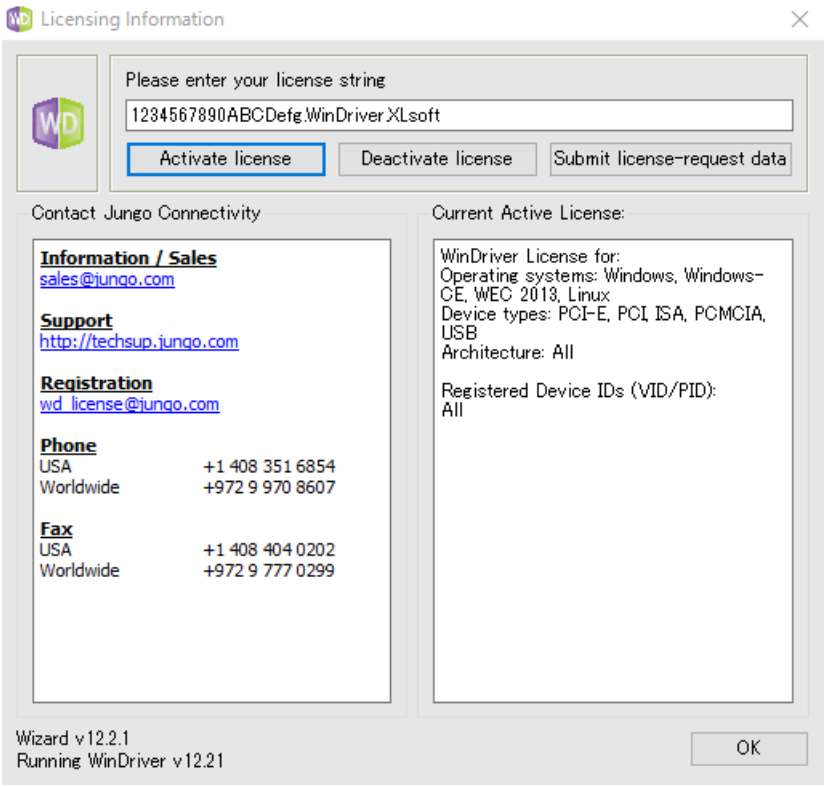


図 4.1: ライセンスの登録

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」の用紙の説明に従い、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 Connectivity 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 Registration code、Vendor ID、Product ID) 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 – 0073
東京都港区三田3-9-9 森伝ビル6F
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

注意:
以下の手順は、Windows CE Platform Builder を使用して、または対応する Windows CE plugin と MS Visual Studio 2005 / 2008 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。この手順では、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 ファイルを編集します。
   WinDriver¥samples¥wince_install¥project_wd.reg
2. 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)。
3. プラットフォームにドライバを統合します。
   a. 対象のプラットフォームにドライバを統合します。
      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 Release Directory] を選択します。
   c. WinDriver CE カーネル ファイル – **WinDriver¥redist¥TARGET_CPU¥windrvr1221.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¥project_wd.reg** ファイルの内容を **%_FLATRELEASEDIR%¥project.reg** レジストリ ファイルに追加します。
   e. **WinDriver¥samples¥wince_install¥project_wd.bib** ファイルの内容をバイナリ イメージ ビルダ ファイルの FILES セクション - **%_FLATRELEASEDIR%¥project.bib** ファイルに追加し、ターゲット プラットフォーム (コピーしたテキストの “TODO” コメントを参照) に一致する行のコメントを外します。

      WinDriver CE カーネル ファイル (**windrvr1221.dll**) を永続的に Windows CE イメージ (NK.BIN) の一部とする場合にのみこのステップが必要です。例えば、フロッピーディスクを使用してターゲット プラットフォームにカーネル ファイルを移す場合などがこれに該当します。オン デマンドで **CESH/PPSH** サービスを通して **windrvr1221.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。
4. [Build] メニューより [Make Run-Time Image] を選択し、新しいイメージ NK.BIN の名前をつけます。
5. 新規カーネルをターゲット プラットフォームにダウンロードして初期化します ([Target] メニューより [Attach Device] を選択します。あるいはブートディスクを使用します)。Windows CE 4.x の場合、[Target] メニューより [Attach Device] ではなく [Download/Initialize] を選択します。
6. 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。
7. 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 コンピュータ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 MS eMbedded Visual C++ または MS Visual Studio 2005 / 2008 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 **windrvr1221.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++ / MS Visual Studio 2005 / 2008 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 CE の多くのバージョンでは、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 CE 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 10 / 8.1 / Server 2012 R2 / 8 / Server 2012 / 7 / Server 2008 R2 / Vista / Server 2008 / Server 2003 / XP 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。

注意: WinDriver Windows ツールキットを同じホスト PC にインストールする場合、インストール時に、Windows CE のインストールで設定した WD_BASEDIR 変数の値を上書きするのでご注意ください。

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ため、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 をインストールしていない場合、対象の Linux のディストリビューションのインストール手順に従って、カーネルのソースコードと一緒に Linux をインストールしてください。
- Linux が既に対象のマシン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' を確認してください。ソース コードがインストールされていない場合、対象の Linux のディストリビューションのインストール手順に従って、ソースコードをインストールするか、Linux をソース コードつきで再インストールします。

version.h のインストール

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

  ① スーパー スーザーになります:
  $ **su**

  ② Linux のソース ディレクトリに移動します:
  # `cd /usr/src/linux`

  ③ 以下を入力します:
  # **make xconfig**

  ④ **Save and Exit** を選択して設定情報を保存します。

  ⑤ 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 6.0 (**libstdc++.so.6**) および libpng ライブラリのバージョン 12.0 (**libpng12-0**) が必要です。これらのファイルがインストールされていない場合は、対象の Linux ディストリビューションの適切なパッケージ (例: **libstdc++6** と **libpng12-0**) 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 3.0 カーネルの場合、次を入力します。

ln -s linux-3.0/ 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/tmp**)。
   $ cd /home/username/tmp

> **注意:** インストール ディレクトリへのパスにはスペースを含めないでください。

3. **WD1221LN.tgz** または **WD1221LNx86_64.tgz** ファイルを解凍します。
   $ tar -xvzf /<file location>/WD1221LN[x86_64].tgz

   例:

   - CD の場合:
     $ tar -xvzf /mnt/cdrom/LINUX/WD1221LN.tgz
   - ダウンロードファイルの場合:
     $ tar -xvzf /home/username/WD1221LN.tgz

4. **WinDriver/redist** ディレクトリに移動します (この **WinDriver** 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です。デフォルトのカーネル ソース ディレクトリは **/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。

> **注意: configure** スクリプト オプションのフル リストを確認するには、以下のように **--with-kernel** オプションを使用してください:
> **./configure --help**

   - <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、他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/windrvr1221** 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 システムの 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、以下の行を追加して、read (読み取り) および write (書き込み) 権限を与えます。

```
windrvr1221:root:root:0666
```

または、以下のように chmod コマンドを使用することもできます。

```
chmod 666 /dev/windrvr1221
```

9. **WD_BASEDIR** 環境変数を定義し、WinDriver ディレクトリの場所を示すように設定します (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)。この変数を WinDriver のサンプルおよび DriverWizard で生成されたコードの make およびソース ファイルで使用し、DriverWizard [第 5 章] で生成されたプロジェクトを保存するデフォルトのディレクトリにも使用します。この変数を定義しない場合、WinDriver の makefile を使用して、サンプルおよびDriverWizard で生成されたコードをビルドする際に、定義するように要求されます。

10. スーパーユーザーモードを終了します。

```
# exit
```

11. 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 windrvr1221
```

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. [Help] メニューから [Licensing Information] オプションを選択して、[Licensing 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、[Deactivate license] 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. 評価期間中に 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) のセクションを参照してください。デフォルトで使用する `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()` 関数のセクション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/windrvr1221 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を与え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/windrvr1221** 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/windrvr1221** 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 で WinDriver の新しいバージョンにアップグレードするには、セクション 4.2.1 の「Windows 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) 新しいライセンス コードを入力します。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 コード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()` (デフォルトで使用する `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。
- ドライバ コード内で `WD_DriverName()` への呼び出しが新しいドライバのモジュール名 (**windrvr1221** または名前を変更したモジュール名) を使用しているか確認してください。
  DriverWizard を使用して生成されたコード、または新しい WinDriver のバージョンのサンプル コードを使用している場合には、そのコードでは新しいバージョンのデフォルトのドライバ名を既に使用しています。また、以前のバージョンの WinDriver の DriverWizard を使用して生成されたコード、またはそのバージョンのサンプル コードを使用している場合、新しいバージョンの **windrvr.h** でコードをリビルドする際には、新しいデフォルトのドライバのモジュール名を使用するようにコードをアップデートしてください (`WD_DEFAULT_DRIVER_NAME_BASE` の定義を使用するため)。
  WinDriver のドライバ モジュールの名前を変更した場合、コード内で変更したドライバ名で `WD_DriverName()` を呼んでいることを確認してください。新しいバージョンのドライバ モジュールの名前を以前のバージョンで使用していた名前に変更している場合には、コードを編集する必要はありません。

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. DriverWizard を起動します - **<path to WinDriver>/wizard/wdwizard**。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。
2. 登録ユーザーの場合、WinDriver のライセンスが登録されているのを確認します (セクション 4.2 の WinDriver のインストールとライセンスの登録に関する説明を参照してください)。評価ユーザーの場合、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。
4. ISA カードの場合 – ISA バスにカードを挿入します。DriverWizard をカードのリソースに合わせて設定し、DriverWizard からカードを読み書きできるかどうかを確認してください。

4.4.2 Windows CE コンピュータの場合

1. 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。
2. 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 GUI 版の **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 14 章 を参照してください)。

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

     注意: Kernel PlugIn の名前は、*.sys 拡張子無しで指定してください。
   - Kernel PlugIn ドライバを **%windir%¥system32¥drivers** ディレクトリから削除します。
3. 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 (**windrvr1221.sys**) と動作するように登録した デバイスの INF ファイルが存在しないことをご確認ください。
4. WinDriver をアンインストールします。
   - WinDriver ツールキットがインストールされている開発用 PC の場合:
     [スタート] メニューから [プログラム] - [WinDriver] - [Uninstall] を選択するか、または **¥WinDriver** ディレクトリにある **uninstall.exe** を実行します。

     WinDriver カーネル モジュール (**windrvr1221.sys**) を停止し、アンロードし、**¥%windir%¥inf** ディレクトリから **windrvr1221.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 (追加したファイルは除きます) も削除します。
   - WinDriver ツールキットではなく、WinDriver のカーネル モジュール (**windrvr1221.sys**) をインストールしたターゲット 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 <windrvr1221.inf へのパス> uninstall

注意: このコマンドを実行する際には、**windrvr1221.sys** と **windrvr1221.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:

- WinDriver をアンインストール時に、WinDriver のサービスに対して開いているハンドルがある場合 (**windrvr1221.sys** か、または名前を変更したドライバ)、または WinDriver のサービスと動作するように登録された Plug-and-Play デバイスが接続されてるか、または有効な場合、wdreg はドライバのアンインストールに失敗します。これは、ドライバの使用中は、アンインストールできないようになっています。
- 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確認することが可能です。ドライバがロードされている場合は、Debug Monitor のログにドライバと OS の情報が表示され、そうでない場合はエラー メッセージが表示されます。開発用 PC では、このユーティリティはアンインストール コマンドによって削除されます。アンインストール後に使用する場合は、アンインストールを実行する前に、**wddebug_gui.exe** のコピーを作成しておきます。

5. **windrvr1221.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¥windrvr1221.sys**
   - **%windir%¥inf¥windrvr1221.inf**
   - **%windir%¥system32¥wdapi1221.dll**
   - **%windir%¥sysWOW64¥wdapi1221.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 **windrvr1221** 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。
   - Kernel PlugIn ドライバを作成した場合、Kernel PlugIn のドライバ モジュールをアンロードします。
     # **/sbin/rmmod kp_xxx_module**
2. WinDriver をマニュアルでインストールした場合、以下のコマンドで WinDriver のドライバ モジュールをアンロードします:
   # /sbin/modprobe -r windrvr1221
3. Kernel PlugIn ドライバを作成した場合、同様に Kernel PlugIn のドライバ モジュールを削除します。

4. 以下のコマンドで **/etc** ディレクトリから **.windriver.rc** ファイルを削除します:
   # **rm -f /etc/.windriver.rc**

5. 以下のコマンドで **$HOME** から **.windriver.rc** ファイルを削除します:
   # **rm -f $HOME.windriver.rc**

6. DriverWizard へのシンボリック リンクを作成した場合、以下のコマンドでリンクを削除します:
   # **rm -f /usr/bin/wdwizard**

7. 以下のコマンドで WinDriver のインストール ディレクトリを削除します:
   # **rm -rf <path to WinDriver ディレクトリ>**
   (例えば、# **rm -rf /usr/local/WinDriver**)

8. WinDriver の共有オブジェクト ファイルが存在する場合、以下のコマンドで削除します:
   /usr/lib/libwdapi1221.so (32 ビット x86 または 32 ビット PowerPC)
   /usr/lib/64/libwdapi1221.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: WinDriver の PCI API を使用して CardBus デバイスをハンドルします。そのため、この章の PCI の説明は、CardBus をサポートするオペレーティング システム上で、CardBus にもあてはまります。

5.1 DriverWizard の概要

(WinDriver ツールキットに含まれる) DriverWizard は、以下のハードウェアとドライバ開発の 2 つの段階をターゲットとする GUI (グラフィカルユーザー インタフェース) のツールです。

ハードウェアの診断:ドライバ コードを 1 行書く前に、DriverWizard を使用して、ハードウェアのリソースに対して読み書きができます。ハードウェア開発の終了後、PCI の場合、ハードウェアを対象のマシンの適切なバス スロットに挿入します。DriverWizard を使って、ハードウェアのリソース (メモリ、I/O 範囲、PCI コンフィグレーション レジスタや割り込み) を確認し、メモリや I/O アドレスへの読み書き、カスタム レジスタの定義とアクセス、割り込みの確認などを行って、ハードウェアが正しく動作しているか確認します。USB の場合、対象のデバイスを対象のマシンの USB ポートに挿入します。DriverWizard を使って、ハードウェアのコンフィグレーションやパイプの情報を確認し、パイプのデータを転送したり、コントロール パイプにスタンダード リクエストを送信したり、パイプをリセットして、ハードウェアが正しく動作しているか確認します。

コードの生成: 対象のデバイスが期待通りに動作するのを確認したら、DriverWizard を使用して、対象のハードウェアのリソースを確認およびアクセスする機能を持つ雛形となるドライバのソースコードを生成します。

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 Qsys デザイン、Xilinx BMD デザイン) 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「特定のチップ セット サポート」を参照することを推奨します。

Windows では、DriverWizard を使用して対象のハードウェアの INF ファイルを生成することもできます。

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 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バス 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。

   ① DriverWizard を起動します - **`<path to WinDriver>¥wizard¥wdwizard`**。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。
   注意: Windows Vista およびそれ以降の OS では、管理者権限で DriverWizard を起動する必要があります。

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

図 5.1: WinDriver のプロジェクトを開く、または新規作成

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

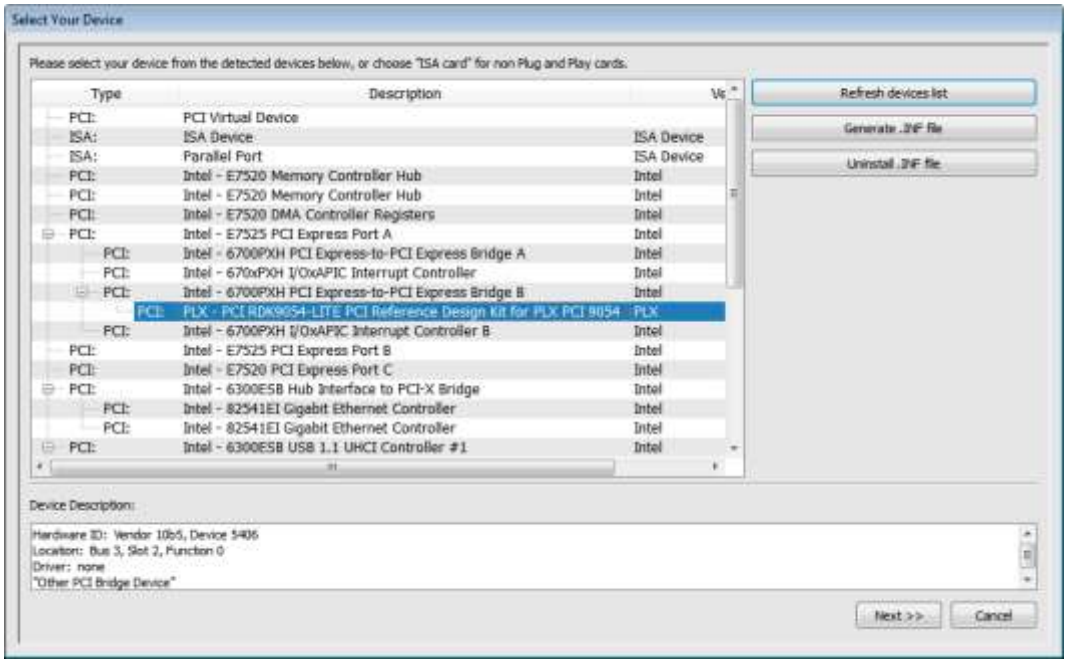


図 5.2: デバイスの選択

3. DriverWizard で INF ファイルを作成し、インストールします。

サポートする Windows OS では、Plug-and-Play デバイス (PCI / PCMCIA / USB) 用のドライバをインストールするには、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。DriverWizard を使用して、対象のデバイスを WinDriver を動作するように登録する INF ファイルを生成します (**windrvr1221.sys** ドライバと動作するように)。DriverWizard で生成した INF ファイルは、Windows ユーザーに配布し、その PC にインストールする必要があります。

また、このステップで生成した INF ファイルは、DriverWizard を使用して Windows で Plug-and-Play デバイスを診断する場合にも必要です。INF ファイルの必要性は、セクション 16.1.1 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、インストールします。

① [Select Your Device] 画面 (ステップ2を参照) で、[Generate .INF file] ボタンまたは [Next] ボタンを押します。

② DriverWizard は、Vendor ID、Device ID、Device Class、メーカー名およびデバイス名を含むデバイスに関する情報を表示します。この情報を変更することができます。

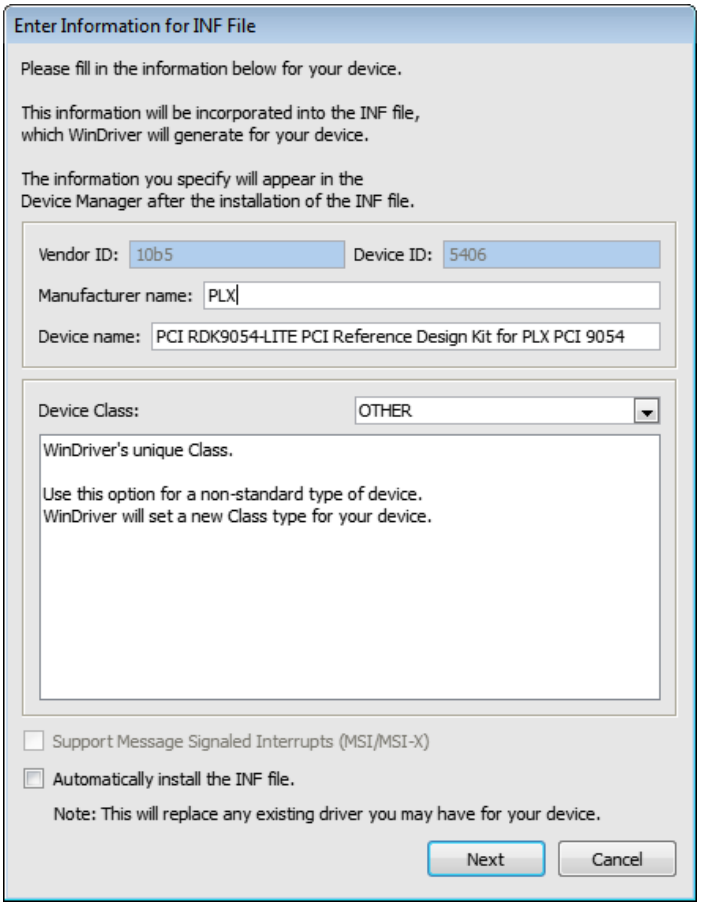


図 5.3: DriverWizard INF ファイル情報

③ マルチ インターフェイスの USB デバイスの場合、各インターフェイスに対して別々に INF ファイルを作成するか、すべてまたはマルチ インターフェイスの複合デバイスに対して 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。

- マルチ インターフェイスの USB デバイスの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。

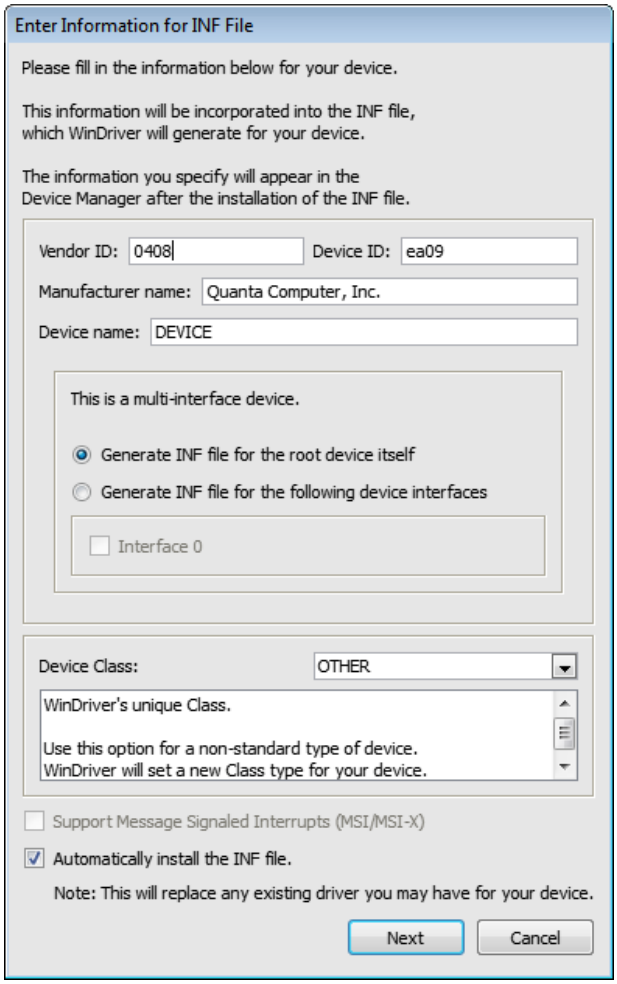


図 5.4: DriverWizard のマルチ インターフェイスの INF ファイル情報
(特定のインターフェイスをそれぞれ設定する場合)

- マルチ インターフェイスの USB デバイスの複合デバイスに対して 1 つの INF ファイルを作成する場合、[Enter Information for INF File] ダイアログでルート デバイス用の INF ファイルの生成、または特定のインターフェイス用の INF ファイルの生成を選択することができます。ルート デバイス用の INF ファイルの生成を選択すると、複数のアクティブなインターフェイスを処理できるようになります。

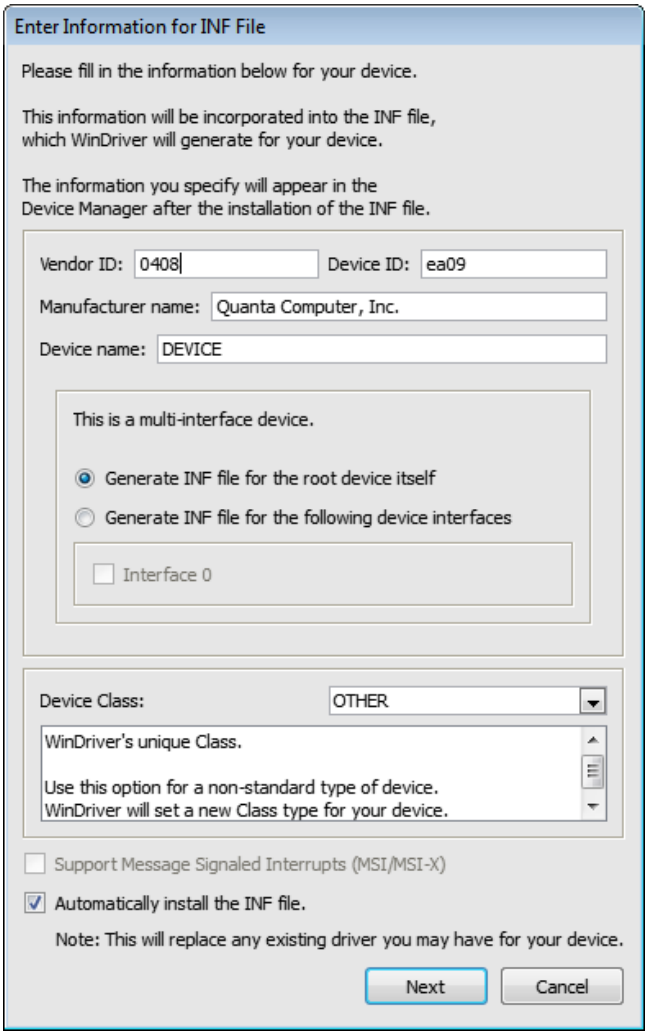


図 5.5: DriverWizard のマルチ インターフェイスの INF ファイル情報
(1 つのインターフェイスを設定する場合)

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

DriverWizard で [Automatically Install INF file] オプションをオン (USB デバイスでは、このオプションはデフォルトでオンです) にすることによって INF ファイルを自動的に DriverWizard からインストールできます。
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 16.1 で説明します。

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

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 およびそれ以降で、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。

   Windows で DriverWizard を使用して、以前にインストールした対象の 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できます。これで現行のドライバからデバイスの登録を解除し、Windows の INF ディレクトリにある INF ファイルのコピーを削除します。

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

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

   ② INF ファイルを選択し、削除します。

5. デバイスを診断します。

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

   PCI デバイスの場合

   ① デバイスを診断します。

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

➢ DirverWizard は自動的に Plug-and-Play ハードウェア リソース (I/O 範囲、メモリ範囲、割り込み) を検出します。

図 5.7: PCI のリソース画面

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。

注意: Windows 7 およびそれ以降では、非 Plug-and-Play のハードウェアにリソースを割り当てる前に WinDriver で IRQ の登録が必要な場合があります。

以下の図のように手動でハードウェアのレジストも定義します。

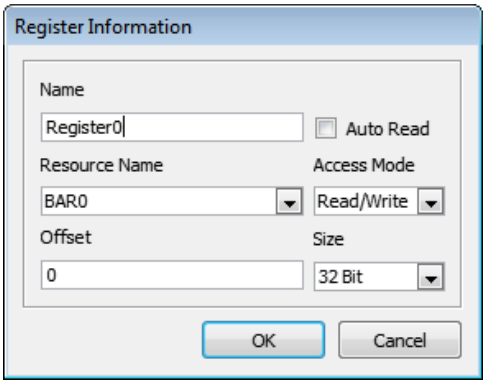


図 5.8: レジスタの定義

> **注意:** レジスタを定義する際に、[Register Information] ウィンドウの [Auto Read] チェック ボックスをチェックします。[Auto Read] としてマークされたレジスタは、DriverWizard から実行したレジスタの read (読み込み) / write (書き込み) の操作に対して、自動的に読み込まれます。DriverWizard の [Log] ウィンドウに読み込み結果を表示します。

➢ 以下の図のように、I/O ポート、メモリ スペース、定義したレジスタへの読み込みと書き込みをします。

> **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。

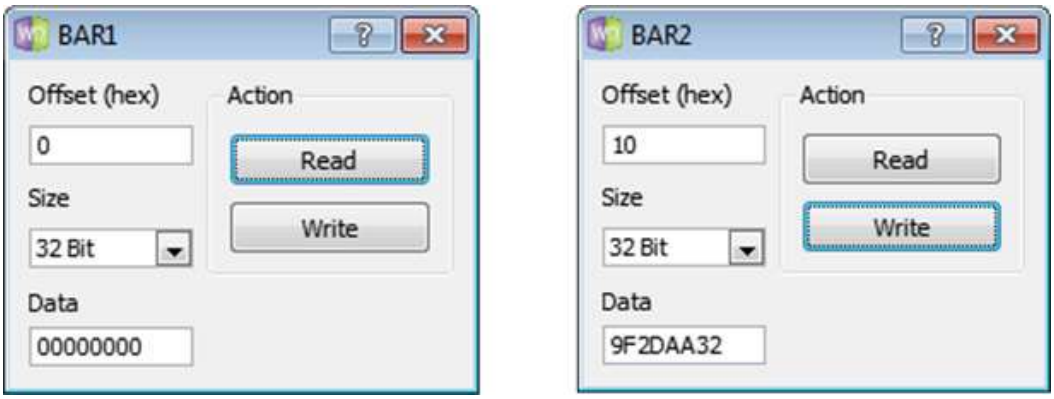


図 5.9: メモリおよび I/O の Read / Write

➢ ハードウェアの割り込みを 'Listen' (確認) します。

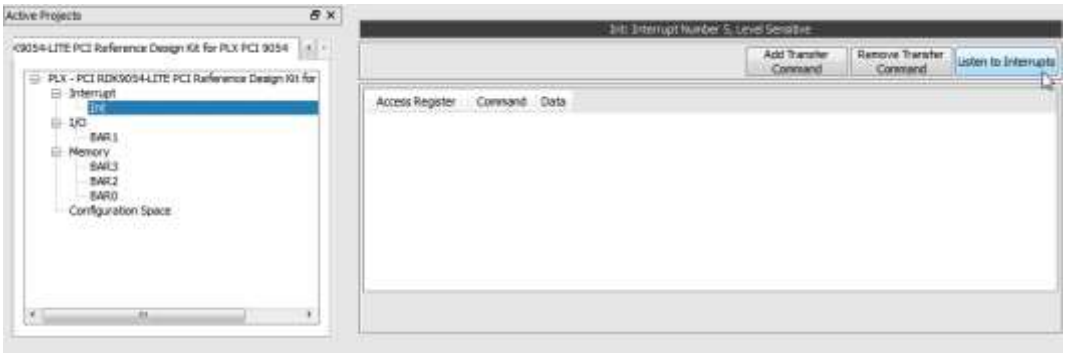


図 5.10: 割り込みの Listen (確認)

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。

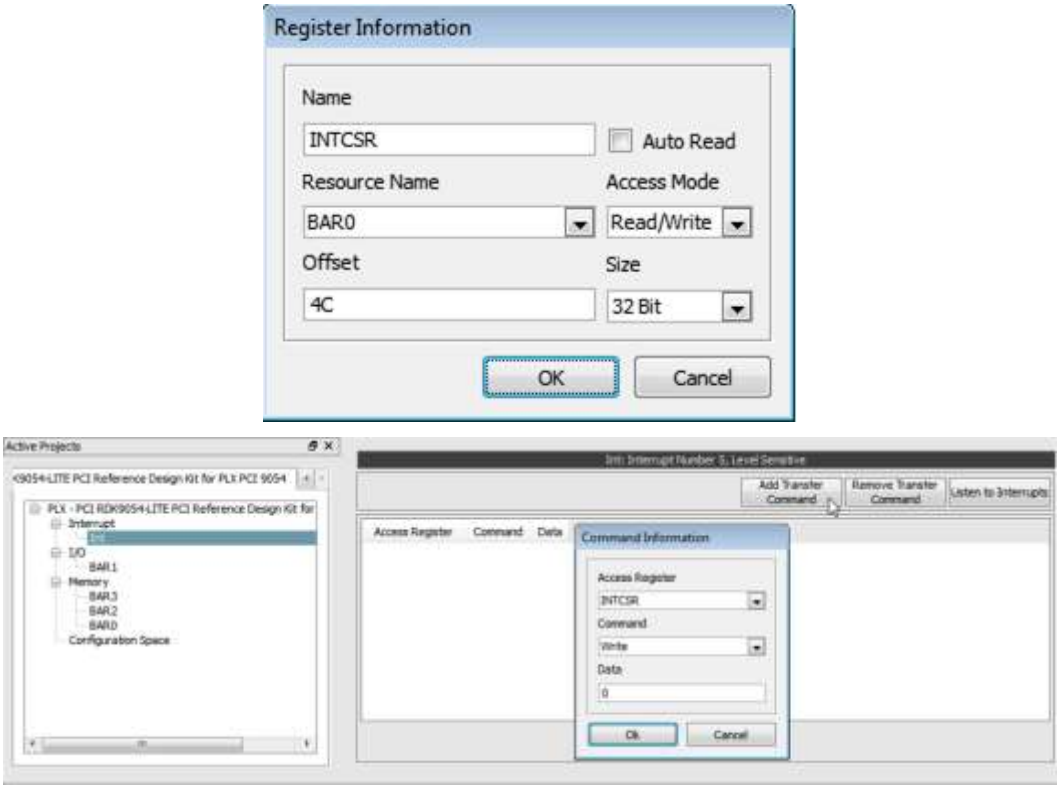


図 5.11: レベル センシティブな割り込みの転送コマンドの定義

USB デバイスの場合

① USB デバイスの対象の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検出し、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。DriverWizard は選択した代替設定のパイプ情報を表示します。

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 は自動的に検出した代替設定を選択するので、[Select Device Interface] ダイアログは表示されません)。

② USB デバイスを診断します。
デバイス ドライバのコードを記述する前に、対象のハードウェアが期待通りに動作するか確認することは重要です。DriverWizard を使用して対象のハードウェアを診断します。すべての動作は DriverWizard のログに記録されるので、テスト結果を後で分析できます。

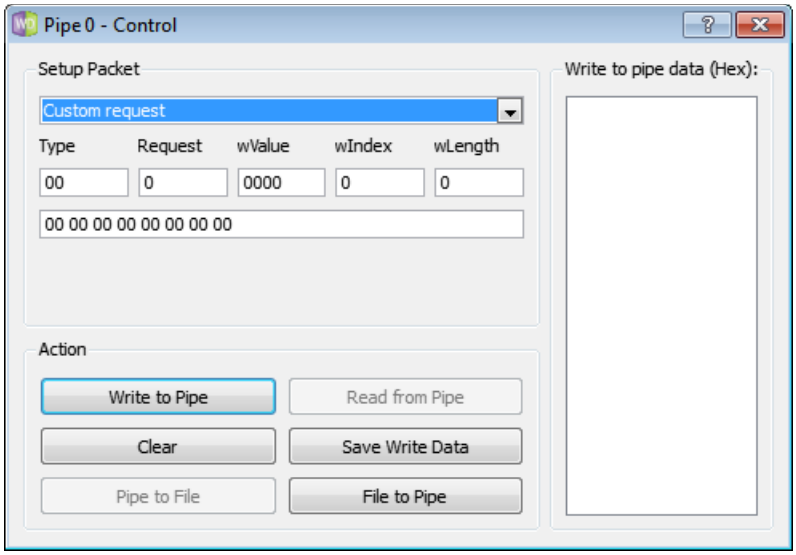


図 5.13: USB コントロール転送

対象の USB デバイスのパイプのテスト:
DriverWizard は、選択した代替設定に対して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 **エラー! 参照元が見つかりません。**「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。
読み込みを中止する場合は、[Stop Listen to Pipe] をクリックします。

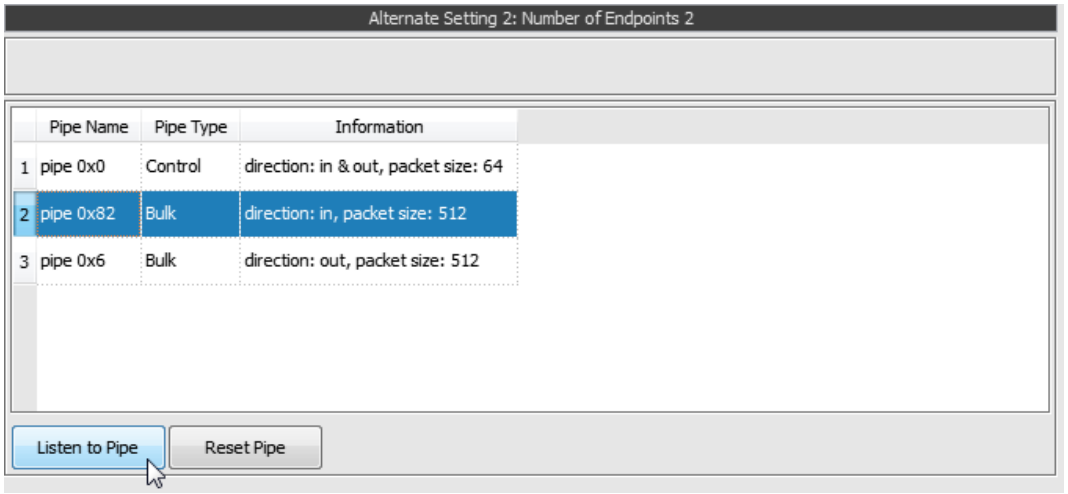


図 5.14: パイプの確認

iv. 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。

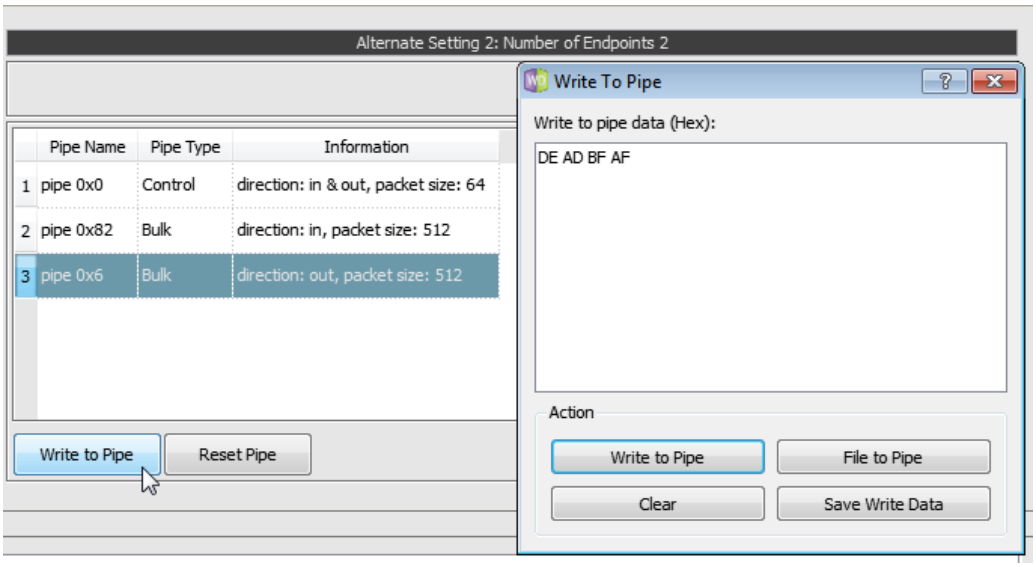


図 5.15: パイプへの書き込み

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

6. 雛型となるドライバ コードを生成します。

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

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

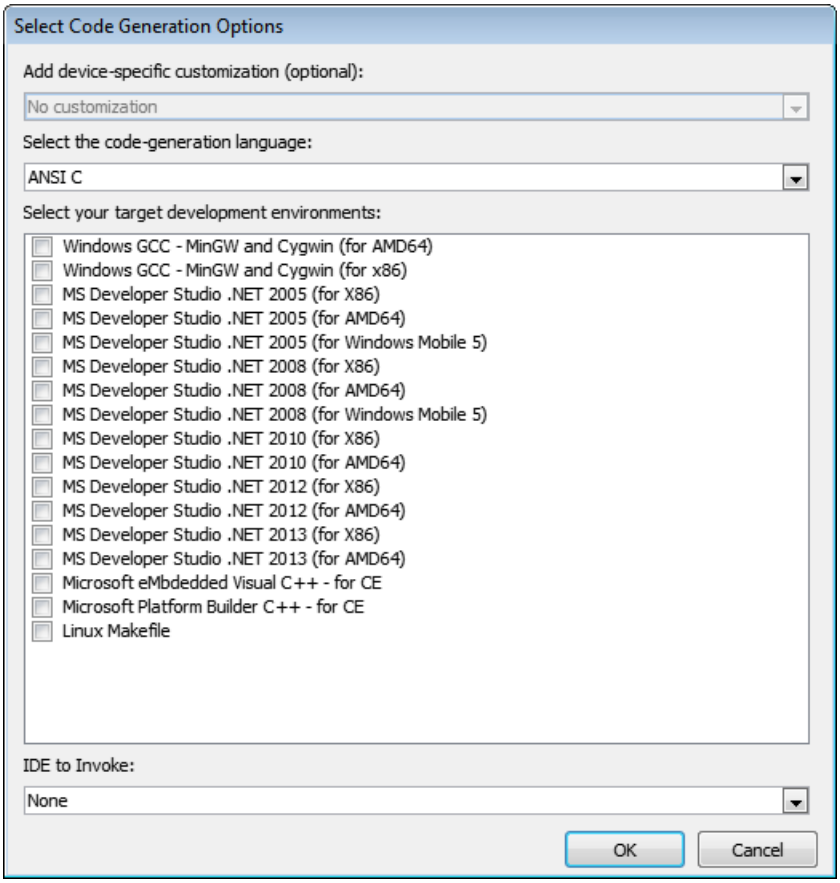


図 5.16: コード生成のオプション

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

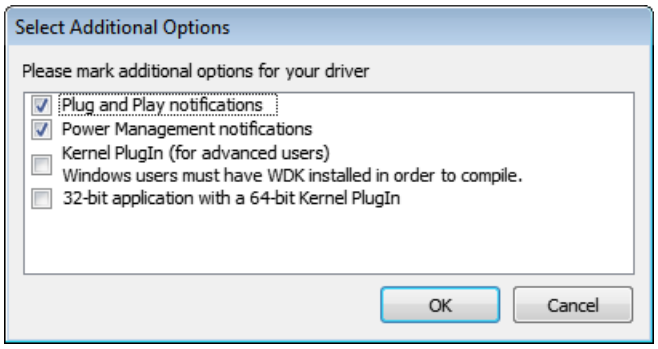


図 5.17: ドライバ オプションの選択

Kernel PlugIn の Windows プロジェクトの注意:

- 生成した Kernel PlugIn のコードをコンパイルする際には、適切な WDK (Windows Driver Kit) をインストールする必要があります。
- MS Visual Studio を使用して Kernel PlugIn のプロジェクトをビルドする際には、プロジェクトのディレクトリへのパスにスペースを含めないでください。

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

⑤ DriverWizard を終了します。

7. 生成されたコードをコンパイルし、実行します。

   - このコードをデバイス ドライバの雛形として使用します。ドライバの特定の機能を実行する場合には、必要に応じて修正します。
   - DriverWizard が生成したソース コードは 32 ビット コンパイラでコンパイル可能で、コードの修正をせずに対応するすべてのプラットフォームで動作します。

5.2.1 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。

5.2.1.1 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。

5.2.1.2 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。

5.2.1.3 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。

5.2.2 生成されたコードをコンパイルする

5.2.2.1 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、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、MS Visual Studio、Windows GCC (MinGW/Cygwin)、MS eMbedded Visual C++、または 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 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。

Kernel PlugIn プロジェクトをビルドするには (Windows で)、セクション 12.7.1 の手順を参照してください。

5.2.2.2 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。

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 が拡張サポートを提供する次のチップセット (PLX 6466、9030、9050、9052、9054、9056、9080 および 9656、Altera Qsys デザイン、Xilinx BMD デザイン) を使用している場合、まず次の概要を参照して、この章をスキップして、第 8 章 に進んでください。

6.1 WinDriver でデバイス ドライバをビルドするには

- 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、割り込みを確認します。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 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。
- WinDriver が拡張サポートを提供しているチップセット (PLX 6466、9030、9050、9052、9054、9056、9080 および 9656、Altera Qsys デザイン、Xilinx BMD デザイン) を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 が拡張サポートを提供する特定のチップセットに関する詳細は、第 8 章の「特定のチップ セットの拡張サポート」を参照してください。
- C / .NET コンパイラまたは開発環境 (作成したコードによります) を使用して必要な雛形となるドライバをビルドします。
  WinDriver では次の開発環境およびコンパイラに対して固有のサポートを提供します: MS Visual Studio、MS eMbedded Visual C++、MS Platform Builder C++、GCC、Windows GCC。
- 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 11 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 API リファレンスを参照してください。DriverWizard が自動的に処理できない操作の実行に関しては、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 を提供するその他のヘッダー ファイルをインクルードします (たとえば、**WinDriver¥samples¥shared** ディレクトリからのファイルは便利な診断関数があります)。

2. ソース コードから関連したヘッダー ファイルをインクルードします。

   PCI / ISA の場合

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

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

   USB の場合

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

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

3. コードを WDAPI (Windows の場合) / 共有オブジェクト (Linux の場合) にリンクします。

   - Windows の場合: **WinDriver¥lib¥<CPU>¥wdapi1221.lib** 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>¥wdapi1221.lib**
   - Linux の場合: **WinDriver/lib/** ディレクトリ - **libwdapi1221.so** または **libwdapi1221_32.so** (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)。

   注意: **libwdapi1221_32.so** を使用する場合、まず初めに、異なるディレクトリにこのファイルをコピーし、ファイル名を **libwdapi1110.so** に変更し、対象のコードにリンクします。

   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: wdapi1221** ライブラリまたは共有オブジェクトをリンクする際、ドライバと共に **wdapi1221** 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 **wdapi1221.dll** または **wdapi1221_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** ディレクトリにある **libwdapi1221.so** または

libwdapi1221_32.so (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。

注意: Windows および Linux の 64 ビット プラットフォームで 32 ビット アプリケーション用の DLL / 共有ライブラリを使用する場合 (**wdapi1221_32.dll** / **libwdapi1221_32.so**)、_32 の部分を削除して、配布パッケージに含めるファイル名を変更します。配布手順に関する詳細は、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 / CardBus / PCMCIA ドライバの場合

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

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

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

3. PCI / CardBus / PCMCIA デバイスでは、WDC_PciGetDeviceInfo() / WDC_PcmciaGetDeviceInfo() を呼び出して、選択したデバイスのリソース情報を取得します。ISA デバイスでは、WD_CARD 構造体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を関数へ渡します。これらの関数は、デバイスのハンドルを返します。その後、WDC_xxx API を使用してデバイスと通信ができます。

5. WDC_xxx API を使用して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を閉じます。

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

6.2.3 コードの作成: USB ドライバの場合

1. 対象の USB デバイスに対し WinDriver を初期化するプログラムの初めに WDU_Init() を呼び、デバイス アタッチのコースバックを待機します。関連するデバイスの情報は、このアタッチのコールバックで取得することになります。

2. アタッチのコールバックを受信するば、WDU_Transfer() 系の関数のいずれかを使用して、データの送受信が始められます。

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

6.2.4 設定とコードのビルド

必要なファイルと記述したコードを含めた後、必要なビルド フラグを設定し、開発変数を設定してあることを確認して、コードをビルドしてください。

注意: コードをビルドする前に、WD_BASEDIR 環境変数が WinDriver のインストール ディレクトリに設定されていることをご確認ください。
Windows、Windows CE および Linux では、WD_BASEDIR 環境変数をグローバルに定義可能です – 第 3 章の説明のとおり。

6.3 Windows CE で開発を行うには

WinDriver を使用して Plug-and-Play デバイスをハンドルするには、まず最初に、WinDriver のカーネル モジュール (**windrvr1221.dll**) と動作するようにデバイスを登録する必要があります。

PCI の場合

WinDriver と動作するようにデバイスを登録するには、class (<*CLASS*>)、subclass (<*SUBCLASS*>)、vendor ID (<*VENDOR_ID*>)、および device ID (<*DEVICE_ID*>) (16 進数) でデバイスを認識できるようにレジストリを編集し、**windrvr1221.dll** へデバイスをリンクします。以下のように、関連する情報を **project.reg** ファイルへ追加することでレジストリを編集可能です:

```
[HKEY_LOCAL_MACHINE¥Drivers¥BuiltIn¥PCI¥Template¥WDCard]
"Prefix"="WDR"
"Dll"="windrvr1221.dll"
"Class"=dword:<CLASS>
"SubClass"=dword:<SUBCLASS>
"Order"=dword:ff
"VendorID"=dword:<VENDOR_ID>
"DeviceID"=dword:<DEVICE_ID>
"IsrDll"="giisr.dll"
"IsrHandler"="ISRHandler"
"WdIntEnh"=dword:0
```

WdIntEnh レジストリの設定と割り込みのレイテンシーに関しては、第 9 章を参照してください。

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

USB の場合

WinDriver で動作するように USB デバイスを登録するには、以下のいずれかの方法を実行します:

- Windows CE システムにデバイスを差し込む前に、WDU_Init() を呼んで、デバイスの vendor ID と product ID でデバイスを認識し、WinDriver と動作するように登録します。

  または

- レジストリを編集して、デバイスを認識して、**windrvr1221.dll** にリンクします。関連する情報を **project.reg** ファイルへ追加することでレジストリを編集可能です。
  - vendor ID (<*VID*>) および product ID (<*PID*>) (10 進数) でデバイスを認識できるように以下を追加します:
    [HKEY_LOCAL_MACHINE¥DRIVERS¥USB¥LoadClients¥<ID>¥Default¥Default¥WDR]: "DLL"="windrvr1221.dll"

- デバイスの USB class (<*CLASS*>)、subclass (<*SUBCLASS*>)、および protocol (<*PROT*>) (10 進数) でデバイスを認識できるように以下を追加します:
  `[HKEY_LOCAL_MACHINE¥Drivers¥USB¥LoadClients¥Default¥Default¥<CLASS>_<SUBCLASS>_<PROT>¥WDR]: "DLL"="windrvr1221.dll"`

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

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、以下の 2 つのバージョンがあります:

- **`wddebug_gui`** - Windows および Linux 用の GUI バージョン。
- **`wddebug`** – Windows、Windows CE および Linux 用のコンソールモード バージョン。**`wddebug`** は GUI 実行もサポートします。

両方の Debug Monitor のバージョンは、**`WinDriver/util`** ディレクトリ以下にあります。

7.2.1 wddebug_gui ユーティリティ

`wddebug_gui` は Windows および Linux 用の Debug Monitor ユーティリティの完全なグラフィック (GUI) バージョンです。

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。
   - Windows では、[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。

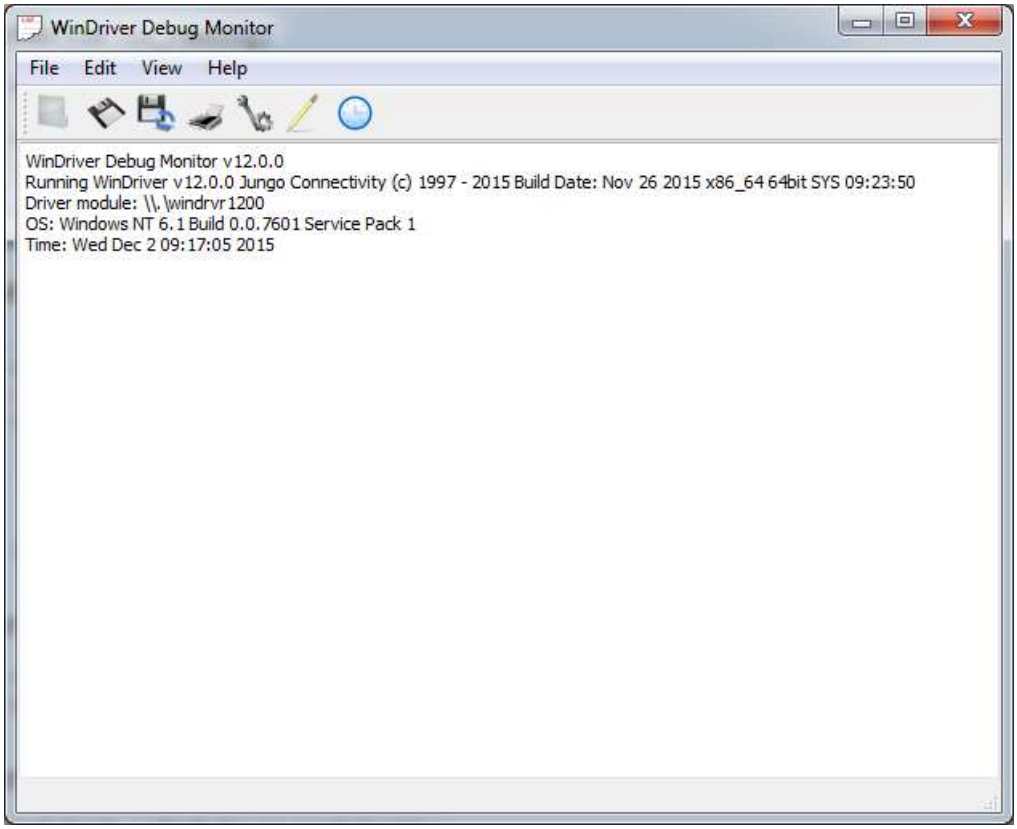


図 7.1: Debug Monitor の起動

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。

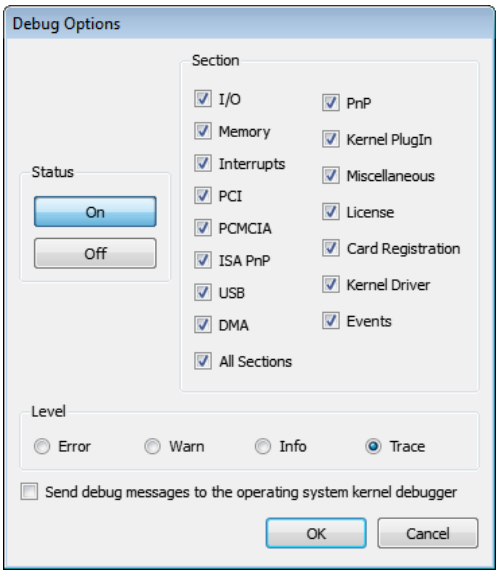


図 7.2: Debug Options の設定

- **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 およびそれ以降の OS 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. Debug Monitor のメニューとツールバーからオプションで追加の設定を作成することができます。
   注意: OS のクラッシュやハングをデバッグする場合、File -> Toggle Auto-Save メニュー オプション（またツールバー アイコン）を設定して Debug Monitor ログを自動的に保存できます。さらに、OS のカーネル デバッガーへデバッグ メッセージを送付することもできます。

5. 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。
   Edit -> Add Custom Message... メニュー オプション (またはツールバー アイコン) を使用して、カスタム メッセージをログに追加することができます。これはログで異なる実行セクションを区別するのに役立ちます。

6. Debug Monitor (または設定した場合にはカーネル デバッガー ログ) 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 wddebug_gui を起動するには

デフォルトでは、**wddebug_gui** は、WinDriver のカーネル モジュール (**windrvr1221.sys/.o/.ko**) 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 (windrvr1221 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、**windrvr1221.sys** (Windows の場合) や **windrvr1221.o** (Linux の場合) ではなく **windrvr1221** を指定します。

たとえば、Windows でデフォルトの **windrvr1221.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 wddebug ユーティリティ

7.2.2.1 コンソールモードの wddebug の起動

Debug Monitor ユーティリティの wddebug バージョンは、サポートするすべての OS (Windows、Windows CE および Linux) でコンソールモードのアプリケーションとして利用可能です。コンソール モードの Debug Monitor バージョンを使用するには、**WinDriver¥util¥wddebug** を起動します。詳細は下記のとおりです。

注意: Windows CE でコンソール モードで実行する場合、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。
Windows CE GUI から **wddebug** を実行することもできます。詳細はセクション 7.2.2.2 を参照してください。

WDDEBUG のコンソール モードの使用方法

```
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 **windrvr1221** (デフォルト)、または **windrvr1221** 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、**windrvr1221.sys** (Windows の場合) や **windrvr1221.o** (Linux の場合) ではなく **windrvr1221** 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 およびそれ以降の OS 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。
- **clock_on**: タイムスタンプを各デバッグ メッセージへ追加します。タイムスタンプは、ドライバの起動時間、または最後の clock_reset コマンドの時間に関連します。
- **clock_off**: タイムスタンプをデバッグ メッセージに追加しません。
- **clock_reset**: デバッグ メッセージのタイムスタンプの時間をリセットします。
- **sect_info_on**: セクション情報を各デバッグ メッセージに追加します。
- **sect_info_off**: セクション情報を各デバッグ メッセージに追加しません。
- **help**: 使用方法を表示します。
- **オプションなし**: 引数なしで (コマンドなしを含む)、**wddebug** を起動できます。Windows CE 以外のプラットフォームでは、**wddebug help** と同等です。Windows CE では、引数なしで wddebug を起動すると、Windows CE GUI バージョンを有効にします。詳細はセクション 7.2.2.2 を参照してください。

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 (デフォルト) 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。

注意: <sections> フラグを設定する場合、<level> フラグも設定する必要があります (デフォルト設定ではない)。

<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 (**sections** フラグを設定する場合、**level** フラグも設定する必要があります)。
  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 **windrvr1221** です。
- OS のカーネル デバッガー (**dbg_on** コマンドを使用) にデバッグ メッセージをリダイレクトしない場合、**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 プロンプトで以下の説明のとおり、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で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。
- stop を選択するまで、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
- 使用方法を表示します:
  wddebug help

  上記の説明のとおり、Windows CE 以外のプラットフォームでは、引数なしで **wddebug** を実行するのと同じです。

7.2.2.2 Windows CE GUI wddebug を使用するには

Windows CE では、引数なしで **wddebug** を実行することでもデバッグ メッセージを出力できます。この方法は、コマンドライン プロンプトを持たない Windows CE プラットフォームでデバッグ メッセージを出力できるように設計されています。このようなプラットフォームでは、**wddebug** 実行ファイルをダブルクリックすることで、デバッグ メッセージの出力を有効にできます。これは、コマンドライン プロンプトから引数なしで起動するアプリケーションの実行に相当します。

引数なしで **wddebug** を実行した場合、GUI メッセージ ボックスが表示され、ログ メッセージを事前定義したログ ファイル (Windows CE のルート ディレクトリの **wdlog.txt**) に格納し、キャンセルか継続か選択します。

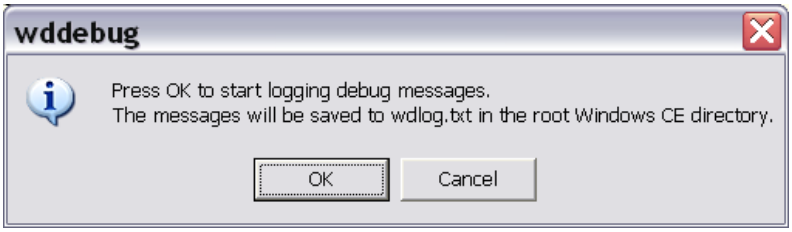


図 7.3: wddebug Windows CE ログ開始メッセージ

継続を選択した場合、トレース レベルを **TRACE** およびデバッグ セクション を **ALL** の設定でデバッグ メッセージの出力をオンにして、Debug Monitor は **wdlog.txt** ログ ファイルにデバッグ メッセージのダンプを開始します。

いつでも Debug Monitor の GUI メッセージ ボックスからログの停止およびデバッグ メッセージの出力をオフにできます。

図 7.4: wddebug Windows CE ログ停止メッセージ

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 PCI チップセット (PLX 6466、9030、9050、9052、9054、9056、9080 と 9656、Altera Qsys デザイン、Xilinx BMD デザイン) で利用できます。

Altera Qsys デザインと Xilinx BMD デザインのチップセットでは、カスタマイズされたコード生成を利用可能です。

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。Altera Qsys design または Xilinx BMD デザインをベースにしたデバイスの場合、DriverWizard のコード生成オプションのダイアログで、このオプションを選択して、これらのデバイス用にカスタマイズしたコードを生成できます。または、他の拡張サポートを提供しているデバイスを使用している場合、以下の手順のように、開発のスタートで、拡張サポートを提供している WinDriver のサンプルを使用できます:

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 **`xxx_diag`** で、ソースコードは通常 **`xxx_diag`** サブディレクトリ以下にあります。プログラムの実行形式はターゲットの OS 用のサブディレクトリ (たとえば、Windows 用の場合には WIN32) 以下にあります。

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. 上記の手順で作成したユーザーモードのドライバのパフォーマンスを向上させる場合は (割り込み処理等)、ソース コードの一部を WinDriver の Kernel PlugIn ドライバに移動して、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。詳細は、第 12 章 の「Kernel PlugIn について」を参照してください。

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

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

DMA バッファを次の 2 つの方法で割り当てることができます。

- **Contiguous Buffer (連続バッファ):** 連続メモリ ブロックを割り当てます。
- **スキャッタ / ギャザー (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 は連続バッファ DMA およびスキャッタ / ギャザー 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 WinDriver を使用してスキャッタ / ギャザーおよび連続バッファ DMA を実装する方法をサンプル コードで紹介し、Windows で連続 DMA バッファの事前割り当ての方法を紹介します。

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、(ポーリングが使用されていない場合) DMA 割り込みを有効にし、次にバッファを解放し、(割り込みが有効の場合は) 各 DMA 転送に対して割り込みを無効にします。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、DMA の割り込みを有効にし (ポーリングが使用されていない場合)、その後、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 (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 スキャッタ / ギャザー (Scatter / Gather) DMA の実装

DMA 実装のサンプル

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

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

スキャッタ / ギャザー 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);
}
```

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 より大きいスキャッタ / ギャザー 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 の実装

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

ハードウェア独自の連続 DMA の詳細な例は、以下の拡張サポートを提供しているチップセット [第 8 章] のサンプル ライブラリ ファイルを参照してください:

- PLX – **`WinDriver¥plx¥lib¥plx_lib.c`** および **`WinDriver¥plx¥diag_lib¥plx_diag_lib.c`** (**`plx_lib.c`** DMA API を使用)
- Xilinx Bus Master DMA (BMD) デザイン - **`WinDriver¥xilinx¥bmd_design¥bmd_lib.c`**

Contiguous Buffer (連続バッファ) 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);
}
```

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。

9.1.2.1 Windows で連続 DMA バッファの事前割り当て

WinDriver では、DMA API を使用して割り当て可能な DMA バッファのサイズに制限はありません。ただし、DMA の割り当て処理は、割り当て時の利用可能なシステム リソースの量に依存し

ます。そのため、バッファをより速く割り当てれば、より良い割り当てができます。
WinDriver for Windows では、対象のデバイスの INF ファイルを編集して、起動時に連続 DMA バッファを再割り当てすることができます。これによって、割り当てに成功する確度を上げます。

注意: host-to-device のバッファおよび / または device-to-host へのバッファの 2 方向のバッファのいずれれの最大値、または 2 方向のバッファに事前割り当てができます。

Windows で連続 DMA バッファを事前に割り当てるには、以下の手順で行います:

1. 以下のように、対象のデバイス INF ファイルの [UpdateRegistryDevice] レジストリキー以下に必要な設定を追加します。

   注意:

   - 以下、二つのバッファの事前割り当ての設定の例です (一方向まはた二方向のいずれか一方、ただし、もちろん一つのバッファのみ (またはまったくなし) を事前割り当てすることもできます。
   - DriverWizard で生成したデバイスの INF ファイルおよび WinDriver のサンプルに含まれるデバイスの INF ファイルには、既に一方向のバッファの設定行が含まれているので、各行の最初のコメント (';') を削除し、二方向のバッファの場合には、DMA 方向のフラグを変更し、タイトルのコメントを編集する必要があります。

   - 一方向のバッファを事前割り当てするには、以下の行を追加します:

```
; Host-to-device DMA buffer:
;HKR,,"DmaToDeviceBytes",0x00010001,0x100000 ; Buffer size, in bytes
;HKR,,"DmaToDeviceOptions",0x00010001,0x41 ; DMA flags (0x40=DMA_TO_DEVICE
                                          ;                                 +
0x1=DMA_KERNEL_BUFFER_ALLOC)
; Device-to-host DMA buffer:
;HKR,,"DmaFromDeviceBytes",0x00010001,0x100000 ; Buffer size, in bytes
;HKR,,"DmaFromDeviceOptions",0x00010001,0x21 ; DMA flags
                                            ; (0x20=DMA_FROM_DEVICE +
                                           ;
0x1=DMA_KERNEL_BUFFER_ALLOC)
```

   - 二方向のバッファを事前割り当てするには、以下の行を追加します:

```
; Bidirectional DMA buffer:
;HKR,,"DmaToDeviceBytes",0x00010001,0x100000 ; Buffer size, in bytes
;HKR,,"DmaToDeviceOptions",0x00010001,0x61 ; DMA flags
                                           ; (0x60=DMA_TO_FROM_DEVICE
                                          ;                                  +
0x1=DMA_KERNEL_BUFFER_ALLOC)
; Second bidirectional DMA buffer:
;HKR,,"DmaFromDeviceBytes",0x00010001,0x100000 ; Buffer size, in bytes
;HKR,,"DmaFromDeviceOptions",0x00010001,0x61 ; DMA flags
                                            ; (0x60=DMA_TO_FROM_DEVICE
                                           ;                                  +
                                     0x1=DMA_KERNEL_BUFFER_ALLOC)
```

   注意: 二方向のバッファの INF ファイルの設定では、一方向のバッファと同じレジストリ キーを使用します (DmaToDeviceXXX と DmaFromDeviceXXX)、ただし、DMA オプション キーの値(DmaToDeviceOptions / DmaFromDeviceOptions) は **DMA_TO_FROM_DEVICE** フラグ(0x06) を設定してください (DMA_TO_DEVICE (0x04) または DMA_FROM_DEVICE (0x20) フラグではなく)。

2. 必要に応じて、INF ファイルのバッファ サイズを編集し、オプション マスクへフラグを追加します。ただし、上記の手順 1 の説明の通り、方向のフラグと DMA_KERNEL_BUFFER_ALLOC フラグを設定する必要があるので、注意してください。

> **注意:** WinDriver でサポートする DMA フラグに関しては、別紙の WD_DMA 構造体の dwOptions フィールドの説明を参照してください。INF ファイルに設定する関連するフラグの値を探すには、**WinDriver¥include¥windrvr.h** ファイルのフラグの定義を参照してください; (たとえば、DMA_KERNEL_BUFFER_ALLOC フラグを含む enum を探すなど)。

4. コードでは、連続 DMA ロック関数 (WDC_DMAContigBufLock()) への最初の呼び出しまたは最初の二つの呼び出しでは、INF ファイルのバッファ設定に一致するパラメータの値を設定してください。

   - **device-to-host のバッファまたは最初の二方向のバッファの割り当ての場合**、DMA オプション マスクのパラメータ (dwOptions) には、DmaFromDeviceOptions レジストリ キーの値に同じフラグ設定を設定する必要があり、バッファ サイズのパラメータ (dwDMABufSize) を DmaFromDeviceBytes レジストリ キーの値に設定する必要があります。
   - **host-to-device のバッファまたは二番目の二方向のバッファの割り当ての場合**、DMA オプション マスクのパラメータ (dwOptions) には、DmaToDeviceOptions レジストリ キーの値に同じフラグ設定を設定する必要があり、バッファ サイズのパラメータ (dwDMABufSize) を DmaToDeviceBytes レジストリ キーの値に設定する必要があります。

> **注意:**
>
> - WDC_DMAContigBufLock() を使用する場合、DMA_KERNEL_BUFFER_ALLOC フラグ (INF ファイルの設定では設定する必要がある) を明示的に設定する必要はありません (関数が自動的にこのフラグを設定するので)。
> - 低レベルな WinDriver の WD_DMALock() 関数を使用する場合、関数の pDma->dwOptions パラメータに DMA オプションを設定し (DMA_KERNEL_BUFFER_ALLOC フラグも設定する必要があります)、pDma->dwBytes パラメータにバッファ サイズを設定する必要があります。
> - 十分なリソースがないために事前割り当てに失敗する場合、非ページ プールのサイズを増やす必要があります (メモリの割り当てから)。

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、雛形となる診断コードを生成します。このコードには、ウィザードで定義した情報を基に、WinDriver の API を使用して、デバイスの割り込みを処理する方法を記述した割り込みルーチンが含まれます。

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 Interrups) は PCI バスの v3.0 以降で利用可能です。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、Windows Vista およびそれ以降では、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ておく必要があります。

   `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 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 (デバイスからの割り込みを待機させる高レベルな割り込みを有効にする関数で使用される) 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。割り込みを待機している間、CPU の使用はありません。割り込みが発生すると、上記で説明したとおり、最初に WinDriver のカーネルで処理して、`WD_IntWait()` で割り込みハンドラのスレッドを起動して戻り値を戻します。
- ユーザーモードで割り込みハンドラを起動するので、この関数からファイル処理や GDI 関数を含む OS API を呼び出すことができます。

9.2.3 Window 7 以降で非 Plug-and-Play ハードウェアの IRQ の登録

Windows 7 およびそれ以降では、非 Plug-and-Play デバイスに割り込み要求 (IRQ) を割り当てる前に WinDriver を使用して IRQ を登録する必要があます (たとえば、ISA カードなど)。

Windows で WinDriver で IRQ を登録するには、以下のステップを実行します:

1. デバイス マネージャーを開き、[表示] --> [リソース (種類別)] を選択します。

2. 割り込み要求 (IRQ) セクションにあるリストの中から使っていない IRQ を選択します。

3. WinDriver で選択した IRQ を登録します:

   a. WinDriver\redist ディレクトリーにファイルのバックアップを取ります。

   b. windrvr1221.inf を編集します:

      i. [DriverInstall.NT] セクションに以下の行を追加します:

      ```
      LogConfig=config_irq
      ```

      ii. config_irq セクションの追加 ("<IRQ>" は、選択した IRQ 番号を表します – たとえば 10):

      ```
      [config_irq]

      IRQConfig=<IRQ>
      ```

   c. コマンドライン プロンプトから以下を実行して、WinDriver を再インストールします ("<**windrvr1221.inf のパス**>" は、修正した WinDriver INF ファイルのパス):

   wdreg -inf <windrvr1221.inf のパス> install

   d. WinDriver で IRQ が正常に登録された確認します:

デバイス マネージャーを開き、WinDriver のデバイスを選択します。デバイスのプロパティーに登録された IRQ の [リソース] タブが登録されているはずです。

注意: この手順で WinDriver の Virtual Device の IRQ を登録します。同じマシンで起動している他の WinDriver のインスタンスとの競合を避けるには、**windrvr1221** ドライバ モジュールの名前を変更することを推奨します。
ドライバの名前を変更する場合、上記の IRQ の登録手順で **windrvr1221.inf** への参照を名前を変更したWinDriver の INF ファイルに置き換えます。

9.2.4 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。

9.2.5 PCI カードの割り込みタイプの決定

Linux または Windows Vista およびそれ以降で 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。

9.2.6 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 の割り込みを有効にする関数は、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 の割り込みを有効にする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は、ユーザーモードへコントロールを返す前に、割り

込みを有効にする関数が処理する各割り込みに対して割り込みを有効にする関数内で割り込みハンドラ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ておく必要があります。

9.2.6.1 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、割り込みの要因を確認するのに使用します。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) に CMD_MASK 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 を実装できないのでご注意ください。

9.2.6.2 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) ライブラリ 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。以下のシナリオ用のサンプル 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 (intMask) に対して、読み込み INTCSR の値をマスクして、割り込みの要因を確認します。

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ておく必要があります。
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);
```

9.2.7 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 、Windows Vista およびそれ以降では、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。
MSI / MSIx をサポートする OS で PCI デバイスの割り込みを有効にする場合、WinDriver ではまず MSI-X か MSI を有効にしようとします (デバイスでサポートされていて、失敗する場合、レガシーのレベル センシティブな割り込みを有効にしようとします)。

Windows では、MSI または MSIx の割り込みを有効にするには、対象のデバイスの関連する INF ファイルをまず最初にインストールする必要があります。

Linux では、WDC_IntEnable() 関数の dwOptions パラメータを使用するか、低レベルな InterruptEnable() 関数を使用して、デバイスに対して有効にする PCI 割り込みのタイプを指定できます (この場合、WinDriver では、指定したタイプの割り込みのみを有効にしようとします (デバイスでサポートされている場合))。

WinDriver のカーネルモードの割り込みハンドラは、カーネル割り込み処理は、割り込みを有効にする関数または割り込みを待機する関数へ渡される 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してるように、ユーザーモードの割り込みハンドラ ルーチンへ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。

9.2.7.1 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 の割り込みを処理するには、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 およびそれ以降で 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 およびそれ以降で 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 BMD (Bus Master DMA) のサンプルでは、MSI の処理を紹介し、この目的のために関連する INF ファイルが含まれます – **WinDriver¥xilinx¥bmd_design¥xilinx_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 をサポートしてる場合でも、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処理方法を使用して、対象のカードの割り込みを処理します。

9.2.8 ユーザーモードの WinDriver 割り込み処理のコード例

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処理の完全なソース 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 割り込み処理のサンプルに関しては、同じ API を使用する、**Xilinx BMD**（**Bus Master DMA**）のサンプル（**WinDriver¥xilinx¥bmd_design**）または MSI / MSI-X をサポートする OS (Linux または Windows Vista およびそれ以降) 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 およびそれ以降で、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();
}
```

9.2.9 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);
```

9.2.9.1 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 バイト オーダー

9.3.1 エンディアンネスとは

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

- ビッグ エンディアンとは、最下位メモリ アドレスにマルチ バイト データ フィールドの最上位バイトから順番に格納します。
  これは、0x1234 のような 16 進文字列を (0x12 0x34) としてメモリに格納します。ビッグ エンドまた

は上位エンドを初めに格納します。4 バイトの値について次のことが同じように当てはまります。たとえば、0x12345678 は、(0x12 0x34 0x56 0x78) と順番に格納されます。

- リトル エンディアンとは、最下位メモリ アドレスにマルチ バイト データ フィールドの最下位バイトから順番に格納します。
  これは、0x1234 のような 16 進文字列を (0x34 0x12) としてメモリに格納します。リトル エンドまたは下位エンドを初めに保存します。4 バイトの値について次のことが同じように当てはまります。たとえば、0x12345678 は、(0x78 0x56 0x34 0x12) と順番に格納されます。

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

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

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

9.3.2 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 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。

9.3.3 PCI ターゲット アクセスのマクロ

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

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

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

以下の場合に WinDriver のマクロ定義を使用します。

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

   たとえば:

   ```
   DWORD data = VALUE; *mapped_address = dtoh32(data);
   ```

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

   たとえば:

   ```
   WORD data = dtoh16(*mapped_address);
   ```

注意: WinDriver API (WDC_Read/WriteXXX()関数、WDC_MultiTransfer()関数、低レベル 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)。

9.3.4 PCI マスター アクセスのマクロ

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

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

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

以下の場合に WinDriver のマクロ定義を使用します。

カードで読み込み/書き込みを行うホスト メモリ上で準備したデータにマクロを適応する場合。そのような場合の例は、スキャッタ / ギャザー DMA 用の一連のディスクリプタです。

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

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

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

9.4 複数プロセス間のバッファ共有

このセクションでは複数プロセス間での連続 DNA バッファとカーネル バッファを供する方法を紹介します。バッファ共有のメカニズムを使用することで、開発者は不要なバッファのコピーをせずに、複数プロセスの動作の改良を図ることができます。
WinDriver IPC のメカニズムとあわせて、二つのプロセスが大きなメッセージ / データを転送するための方法としても使用できます。

現在、WinDriver では共有 DMA 連続バッファ (WDC_DMAContigBufLock() を参照) とカーネル バッファ (WDS_SharedBufferAlloc() を参照) をサポートしています。

バッファを共有するには、開発者はまず最小にそのグローバル ハンドルを他のプロセスへ渡す必要があります。バッファのグローバル ハンドルの取得に関しては、マクロ WDS_SharedBufferGetGlobalHandle() と WDC_DMAGetGlobalHandle() を参照してください。そして、WinDriver IPC の呼び出し、WDS_IpcUidUnicast()、WDS_IpcSubGroupMulticast() または WDS_IpcMulticast() で、ハンドルを他のプロセスに渡します。

プロセスがグローバル バッファ ハンドルを取得すると、WDC_DMABufGet() または WDS_SharedBufferGet() を使用して、バッファを取得します。WinDriver はバッファを新しいプロセスの仮想メモリ (例、user-space) へ再マップし、バッファを呼び出したプロセスから割り当てたように、すべての必要な情報を書き込みます。

プロセスが共有バッファを必要としなくなると、通常の unlock / free 呼び出し (WDC_DMABufUnlock()、WDS_SharedBufferFree()) を使用します。

注意: WinDriver はすべてのシステム リソースを管理するので、WDC_DMABufUnlock()、WDS_SharedBufferFree() への呼び出しは、すべてのプロセスがバッファを使用しなくなるまで、システム リソースを削除しません。

9.5 64 ビット OS のサポート

9.5.1 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 11.2.3 を参照してください。

9.5.2 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

デフォルトでは、WinDriver の 64 ビット バージョンを使用して作成したアプリケーションは、64 ビット アプリケーションになります。このようなアプリケーションは 32 ビット アプリケーションよりも効果的なアプリケーションです。しかし、WinDriver の 64 ビット バージョンを使用して、サポートする 64 ビット プラットフォーム (Linux AMD64 および Windows AMD64) で起動する 32 ビット アプリケーションを作成することも可能です。

注意: 以下のドキュメントでは、**<WD64>** は対象の OS 向けに 64 ビット WinDriver のインストール ディレクトリへのパスを示します。また、**<WD32>** は同じ OS 向けに 32 ビット WinDriver のインストール ディレクトリへのパスを示します。

64 ビット プラットフォーム向けの 32 ビット アプリケーションを作成するには、WinDriver の 64 ビット バージョンを使用して、以下の方法を行ってください。

1. このマニュアルに説明されているように、WinDriver アプリケーションを作成します (例: DriverWizard を使用してコードを作成、または WinDriver サンプルを使用)。
2. 以下の設定を使用して、対象の OS 用に適切な 32 ビット コンパイラでアプリケーションをビルドします。
   - プロジェクトまたは makefile に **KERNEL_64BIT** プリプロセッサ定義を追加します。
     注意: makefile では、-D フラグ (**-DKERNEL_64BIT**) を使用して定義を追加します。

サンプルおよびウィザードで生成された Linux の makefile や Windows MSDEV のプロジェクト (64 ビット WinDriver ツールキット内) は、この定義を追加済みです。

- アプリケーションを、64 ビット プラットフォームで実行される 32 ビット アプリケーション用の WinDriver API ライブラリまたは共有オブジェクトの特定のバージョンでリンクします (Windows の場合: **<WD64>¥lib¥amd64¥x86¥wdapi1221.lib**、Linux の場合: **<WD64>/lib/libwdapi1221_32.so**)。

  Linuxでは、開発マシン上の 64 ビット WinDriver ツールキットのインストールで、**/usr/lib** ディレクトリに **libwdapi1221.so** のシンボリック リンク (**<WD64>/lib/libwdapi1221_32.so** へリンク) と、**/usr/lib64** ディレクトリに **libwdapi1221.so** のシンボリック リンク (この共有オブジェクトの 64 ビット バージョン **<WD64>/lib/libwdapi1221_32.so** へリンク) を作成します。サンプルおよびウィザードで生成された WinDriver の makefile は、適切な共有オブジェクトでリンクするそのシンボリック リンクに依存します (そのコードが 32 ビット コンパイラを使用してコンパイルしたか、あるいは 64 ビット コンパイラを使用してコンパイルしたかに依存します)。

  Windows では、サンプルおよびウィザードで生成された MSDEV プロジェクトでは、**wdapi1221.lib** でリンクを定義します (AdditionalDependencies を参照)。しかし、リンカー ライブラリのパスは、**<WD64>¥lib¥amd64** ディレクトリの 64 ビット ライブラリ ファイルを参照します (AdditionalLibraryDirectories を参照)。このようなプロジェクトを使用して、64 ビット プラットフォーム向けに 32 ビット アプリケーションをコンパイルする場合、コードを **<WD64>¥lib¥amd64¥x86¥wdapi1221.lib** でリンクするために、x86 をライブラリ パスに追加します。

注意:

- アプリケーションを対象の 64 ビット プラットフォームに配布する場合、64 ビット プラットフォームで実行する 32 ビット アプリケーション用の WinDriver API DLL または共有オブジェクトを用意する必要があります (Windows の場合: **<WD64>¥redist¥wdapi1221_32.dll**、Linux の場合: **<WD64>/lib/libwdapi1221_32.so**)。このファイルを配布する前に、"_32" を削除して配布パッケージのファイルのコピーの名前を変更してください。対象の OS にインストールするには、名前を変更した DLL または共有オブジェクトを関連する OS のディレクトリ (Windows の場合: **%windir%¥sysWOW64**、Linux の場合: **/usr/lib**) にコピーする必要があります。ほかのすべての配布ファイルは、その他の 64 ビット WinDriver 配布向けのファイルと同様です。詳細は、第 14 章を参照してください。
- このセクションで説明した方法を使用して作成したアプリケーションは、32 ビット プラットフォームでは動作しません。32 ビット プラットフォーム向けの WinDriver アプリケーションは、KERNEL_64BIT 定義なしでコンパイルする必要があります。さらに、32 ビット WinDriver インストールで展開した 標準 32 ビット バージョンの WinDriver API ライブラリまたは共有オブジェクトでリンクする必要があります (Windows の場合: **<WD32>¥lib¥x86¥wdapi1221.lib**、Linux の場合: **<WD32>/lib/libwdapi1221.so**)。また、32 ビット プラットフォーム向けの WinDriver アプリケーションは、第 14 章で説明されているように、標準 32 ビット WinDriver API DLL または共有オブジェクト (Windows の場合: **<WD32>¥redist¥wdapi1221.dll**、Linux の場合: **<WD32>/lib/libwdapi1221.so**) とその他の必要な 32 ビット配布ファイルをすべて配布する必要があります。

9.5.3 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。

第 10 章
USB 転送

10.1 概要

このセクションでは WinDriver を使用して USB 転送を実装する方法を詳しく紹介します。

第 3 章で説明したとおり、USB 標準ではホストとデバイス間で二種類のデータ交換 (コントロール交換と機能データ交換) をサポートしています。
WinDriver の API を使用して、コントロールと機能データ転送の両方を実装できます。

以下の図のように、DriverWizard ユーティリティでデバイスのパイプを表示し、GUI 環境からの転送を実行します。

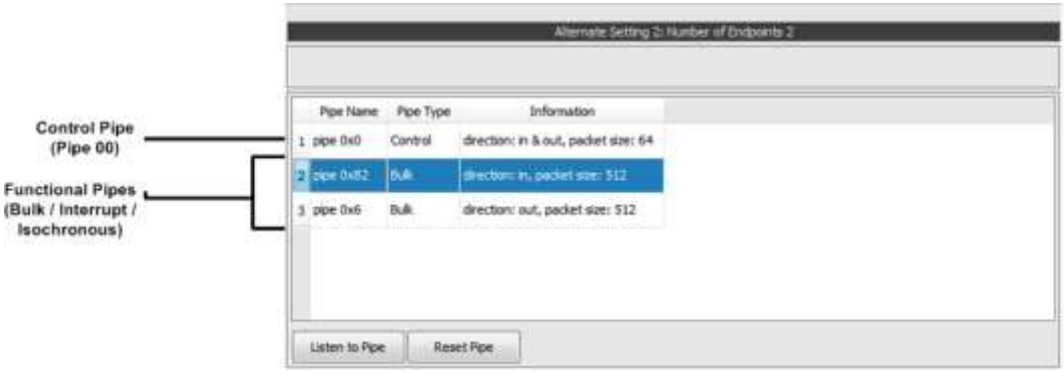


図 10.1: USB データ交換

以下、セクション 10.2 では、USB コントロール転送と WinDriver を使用した実装方法を詳しく紹介します。

以下、セクション 10.3 では、WinDriver で供する機能データ転送の実装オプションに関して紹介します。

10.2 USB コントロール転送

10.2.1 USB コントロール転送の概要

10.2.1.1 コントロール データ交換

USB コントロール交換を使用して、デバイス識別子 (ID)、構成要件を決め、デバイスを構成します; デバイスのその他のパイプのコントロールを含む、その他のデバイス独自の目的のためにも使用します。
コントロール交換はコントロール パイプを使用して実行します (必ず存在するデフォルトの *pipe 0*)。コントロール転送は、セットアップ ステージ (ホストからデバイスへセットアップ パケットを送信)、オプションのデータ ステージとステータス ステージから構成されます。

10.2.1.2 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。

以下の図 (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。

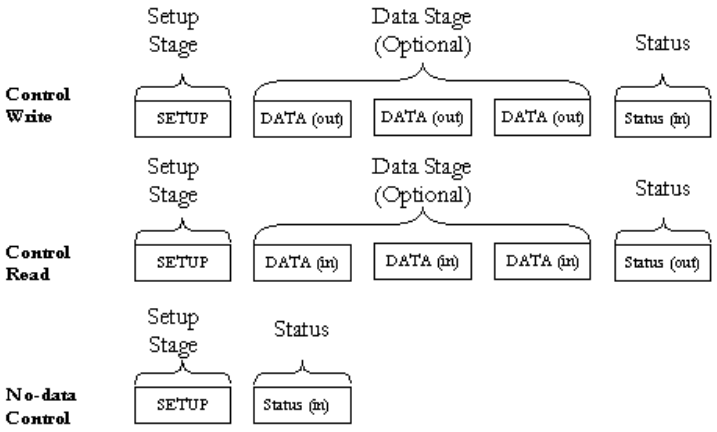


図 10.2: USB のリードとライト

10.2.1.3 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。

10.2.1.4 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 **エラー! 参照元が見つかりません。**「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

10.2.1.5 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

10.2.1.6 セットアップ パケットの例

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

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

GET_DESCRIPTOR (デバイス) セットアップ パケット

80	06	00	01	00	00	12	00

セットアップ パケットの意味:

バイト	フィールド	値	説明
0	BmRequest Type	80	8h=1000b bit 7=1 -> データ方向 (デバイスからホスト) 0h=0000b bits 0..1=00 -> 受信側は "デバイス"
1	bRequest	06	リクエストは 'GET_DESCRIPTOR'
2	wValueL	00	
3	wValueH	01	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 はデスクリプタの長さを示し、バイト 2-3 は USB の仕様リリース ナンバーを含みます。バイト 7 はエンドポイント 00 に対して最も大きいパケット サイズです。バイト 8-9 はベンダー ID で、バイト 10-11 はプロダクト ID を示します。

10.2.2 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 でコントロール パイプ (pipe 0) 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。

10.2.2.1 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 10.3 を参照してください)。

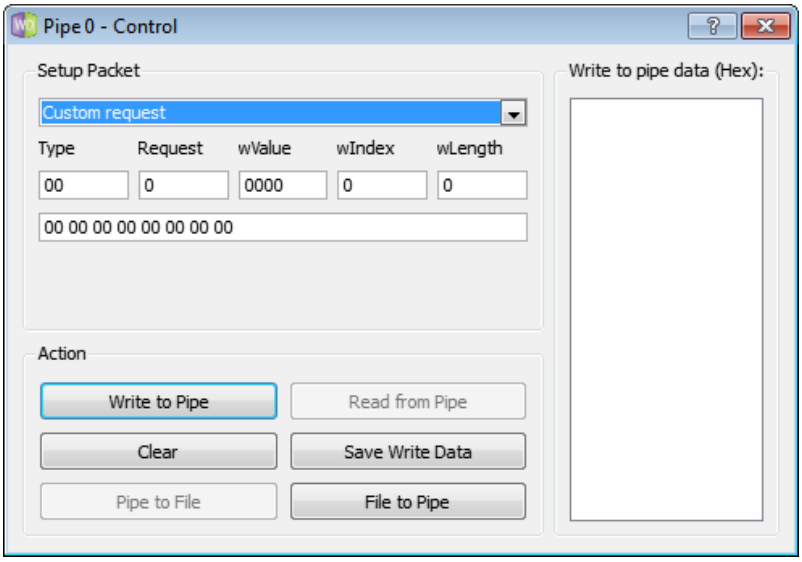


図 10.3: カスタム要求

- 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 10.4 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。

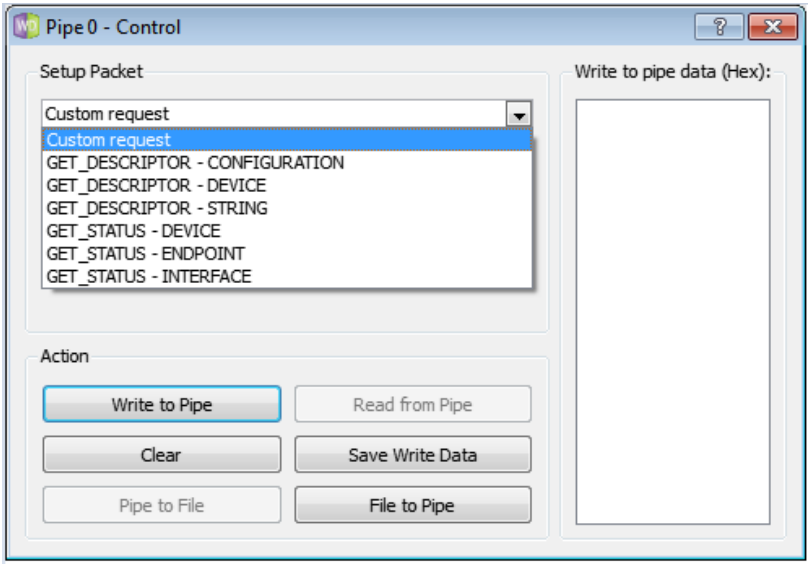


図 10.4: 要求一覧

3. 読み込まれたデータやエラーなど、転送結果を DriverWizard の **Log** 画面から参照できます。**GET_DESCRIPTOR DEVICE** 要求が処理された後の **Log** 画面を、次の図 10.5 に示します。

図 10.5: USB 要求ログ

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

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

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

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

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

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

10.3 機能 USB データ転送

10.3.1 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、インタラプト転送、アイソクロナス転送の 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 [5.2.1] 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。

10.3.2 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。

10.3.2.1 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。

10.3.3 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。

10.3.3.1 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。

注意: 現在 Windows と Windows CE プラットフォームのみでこの関数をサポートしています。

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。

第 11 章
パフォーマンスの向上

11.1 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。[11.2]
- Kernel PlugIn ドライバ [第 12 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 9.2.9 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。

11.1.1 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	• 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 (11.2.2 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 12 章 および第 13 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	• 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。代わりに、メモリ マップ領域を使用するとアクセスの速度が大幅に向上します。
#3	カード上のメモリ マップ領域へのアクセス	• 関数を使用せずにメモリ直接アクセスする (セクション 11.2.1 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 12 章 および第 13 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。

11.2 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 (11.2.1 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 (11.2.2 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。

11.2.1 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のオーバーヘッドを取り除きます。

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 12 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 pTransAddr フィールドおよび pUserDirectAddr 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。pTransAddr 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 12 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、pUserDirectAddr 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 */
```

11.2.2 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。

11.2.3 64 ビット データ転送を行う

注意: 実際の 64 ビット転送の実行能力は、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 9.5 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 WinDrinver で開発したドライバが発揮します。Jungo Connectivity 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 12 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 9.2.9 を参照してください。

12.1 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。

12.2 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 11 章 の「パフォーマンスの向上」を参照してください。

12.3 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。

12.4 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。

12.5 Kernel PlugIn の構造

12.5.1 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 (割り込みハンドラなど) 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。

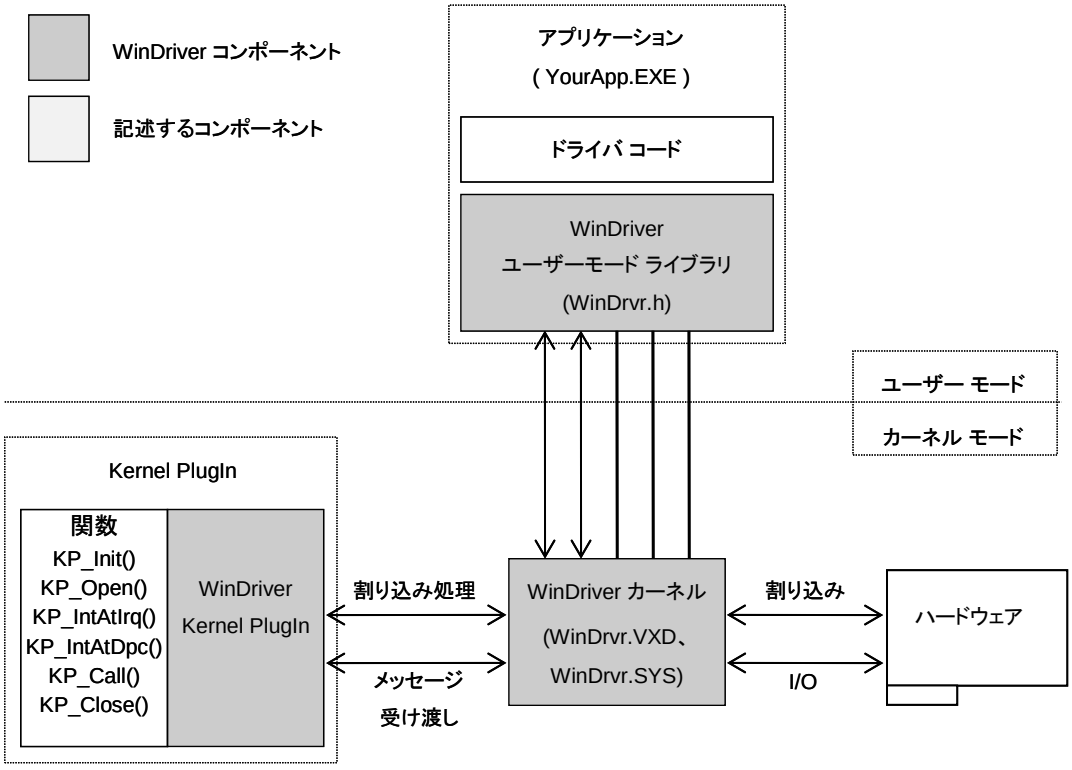


図 11.1: KernelPlugIn の構造

12.5.2 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を有効にします。Kernel PlugIn の割り込みハンドラは、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 12.6.5 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。

12.5.3 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 - **windrvr1221.sys/.o/.ko** (OS に依存)。
- 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/.ko/.kext**)

12.5.4 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:

12.5.4.1 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。
コールバック: `KP_Init()` Kernel PlugIn ルーチンを呼び出します。	`KP_Init()` が WinDriver に `KP_Open()` ルーチンの名前を知らせます。Kernel PlugIn ドライバへのハンドルを開くユーザーモードのリクエストがある場合、WinDriver は関連するオープン ルーチンを呼びます。

イベント: ユーザーモード ドライバ アプリケーションは次の関数のいずれか一つを呼んで、Kernel PlugIn ドライバへハンドルをリクエストします – WDC_KernelPlugInOpen() 関数、WDC_xxxDeviceOpen() 関数 (PCI / PCMCIA / ISA – Kernel PlugIn ドライバの名前と一緒に)、WD_KernelPlugInOpen() 関数 (低レベルの WinDriver API を使用する場合)。	
コールバック: 関連する KP_Open() Kernel PlugIn コールバック ルーチンを呼び出します。	KP_Open() コールバックを使用して、Kernel PlugInドライバで実装したすべてのコールバック関数の名前の WinDriver を通知し、必要に応じて、Kernel PlugIn ドライバを起動します。

12.5.4.2 Kernel PlugIn からのユーザーモード要求処理

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

12.5.4.3 割り込み処理の有効化/無効化および高い割り込み要求レベルの処理

イベント / コールバック	備考
イベント: アプリケーションは fUseKP 引数に TRUE を設定して WDC_IntEnable() を呼ぶか (Kernel PlugIn でデバイスを開いた後)、または、KernelPlugIn ドライバへのハンドルでより低レベルな InterruptEnable() または WD_IntEnable() 関数を呼びます (関数へ渡された WD_INTERRUPT 構造体の hKernelPlugIn フィールドに設定)。	

コールバック: KP_IntEnable() Kernel PlugIn ルーチンを呼び出します。	この関数には Kernel PlugIn の割り込み処理に必要な初期化設定を含めてください。
イベント: ハードウェアが割り込みを発生します。	
コールバック: 高い IRQL の Kernel PlugIn の割り込みハンドラ 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 (割り込みを確認するために、レベル センシティブ割り込みの HW 割り込みシグナルを下げるなど) 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_IntAtIrql() (レガシー割り込み) または KP_IntAtIrqlMSI() (MSI / MSI-X) では TRUE を返し、関連する割り込み遅延処理ハンドラ (KP_IntAtDpc() または KP_IntAtDpcMSI()) で追加の処理を遅延処理することができます。
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 ルーチンを呼び出します。	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。

12.5.4.4 割り込み処理 – 遅延処理の呼び出し

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 を返します。	カーネルで DPC (Deferred Procedure Call) として追加の割り込み処理が必要であることを WinDriver へ通知します。
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 KP_IntAtDpcMSI() (MSI / MSI-X) を呼び出します。	残りの割り込みコードを処理しますが 、高い IRQL 割り込みハンドラよりは優先度が低いです。

イベント: DPC 割り込みハンドラ – KP_IntAtDpc() または KP_IntAtDpcMSI() は 0 よりも大きい値を返します。	WinDriver に追加のユーザーモードの割り込み処理が必要であることを通知します。
コールバック: WD_IntWait() を戻します。	ユーザーモードの割り込みハンドラ ルーチンを実行します。

12.5.4.5 Plug-and-Play およびパワー マネージメント

イベント / コールバック	備考
イベント: アプリケーションは、fUseKP 引数に TRUE を設定して WDC_EventRegister() を呼んで、Kernel PlugIn ドライバを使用して、Plug-and-Play およびパワー マネージメントの通知を受け取るように登録します (Kernel PlugIn でデバイスを開いた後)。または、Kernel PlugIn ドライバへのハンドルでより低レベルな EventRegister() または WD_EventRegister() 関数を呼び出します (関数に渡された WD_EVENT 構造体の hKernelPlugIn フィールドに設定)。	
イベント: Plug-and-Play またはパワー マネージメント イベントが発生します。	
コールバック: KP_Event() Kernel PlugIn ルーチンを呼び出します。	KP_Event() は、発生したイベントに関する情報を受け取り、必要に応じて、イベントの処理を開始します。
イベント: KP_Event() は TRUE を返します。	イベントもユーザーモード アプリケーションで処理する必要があることを WinDriver に通知します。
コールバック: WD_Intwait() を返します。	ユーザーモードのイベント ハンドラ ルーチンを実行します。

12.6 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。

12.6.1 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 場合: WDK (Windows Driver Kit) とその C のビルド ツール

> **注意:** WDK は MSDN (Microsoft Development Network) サブスクリプションの一部として、または Microsoft Connect から利用できます。詳細は、以下のサイトを参照してください: http://msdn.microsoft.com/en-us/library/windows/hardware/gg487428.aspx

- Linux の場合: GCC、gmake または make

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。

12.6.2 Kernel PlugIn の実装

12.6.2.1 始める前に

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

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

12.6.2.2 `KP_Init()` 関数の記述

`KP_Init()` 関数は以下のプロトタイプのようになります:

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

ドライバをロードする際に、この関数を一度呼びます。この関数は、Kernel PlugIn ドライバの名前、WinDriver の Kernel PlugIn ドライバ ライブラリの名前、およびドライバの `KP_Open` コールバックで受信した `KP_INIT` 構造体を格納します (**`WinDriver¥samples¥pci_diag¥kp_pci¥kp_pci.c`** のサンプルを参照してください)。

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。
- ユーザーモードの Kernel PlugIn ドライバへハンドルを開く際に使用するドライバ名は (`WDC_KernelPlugInOpen()` 関数または `WDC_xxxDeviceOpen()` 関数の `pKPOpenData` パラメータ、または低レベルの WD_KernelPlugInOpen() 関数へ渡されるpKernelPlugIn パラメータの `pcDriverName` フィールド)、`KP_Init` へ渡す `KP_INIT` 構造体の `cDriverName` フィールドに設定されたドライバ名に一致していることを確認してください。
  これを実装する最も良い方法は、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ヘッダー ファイル内にドライバ名を定義し、関連するすべての場所で定義した場所を使用することです。

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;
    kpInit->funcOpen_32_64 = KP_PCI_VIRT_Open_32_64;
    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"
```

12.6.2.3 KP_Open() 関数の記述

ターゲットの設定次第で、一つまたは二つの KP_Open() 関数のいずれかを実装できます。KP_Open() 関数は以下のプロトタイプのようになります:

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

ユーザーモードから Kernel PlugIn ドライバへハンドルをオープンする場合 (つまり、WD_KernelPlugInOpen() 関数を呼び出す際に、直接、または WDC_KernelPlugInOpen() 関数または WDC_xxxDeviceOpen() 関数を使用)、このコールバックを呼び出します。

KP_Open() 関数には、Kernel PlugIn で実装する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()	ユーザーモードから WD_KernelPlugInClose() 関数を呼び出す場合 (Kernel PlugIn のオープン ハンドルを含むデバイスに対して呼び出す際に、直接、または高レベルの WDC_xxxDeviceClose() 関数のいずれか一つを使用)、呼び出します。

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 WD_KernelPlugInCall() 関数を呼び出した場合に呼び出されます (ラッパー WDC_CallKerPlug() 関数で呼び出される)。 この関数は Kernel PlugIn メッセージ ハンドラを実装します。
KP_IntEnable()	fUseKP パラメータを TRUE に設定して WDC_IntEnable() を呼び出すか (Kernel PlugIn のハンドルをオープンした後)、Kernel PlugIn ドライバへのハンドルで低レベルの InterruptEnable() または WD_IntEnable() 関数を呼び出すことによって (関数に渡した WD_INTERRUPT 構造体の hKernelPlugIn フィールドを設定)、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、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()	Kernel PlugIn ドライバで割り込みを有効にした場合に、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、または低レベルの InterruptDisable() か WD_IntDisable() 関数を呼び出した場合、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。
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 へのハンドルで有効にして受信した割り込みを指定)。この関数はカーネルモードでレガシー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。追加の割り込みの遅延処理は KP_IntAtDpc() および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 を返すことによってレガシー割り込みの遅延処理を要求した場合に呼び出されます。 この関数はより優先度の低いカーネルモードの割り込みハンドラ コードを含みます。 この関数の戻り値が、アプリケーションのユーザーモードの割り込みハンドラ ルーチンを実行する回数を決定します (可能な限り)。
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 へのハンドルで受信した割り込みに対して有効にした MSI / MSI-X を提供)。この関数はカーネルモードで MSI / MSI-X を処理します。この関数は高い割り込み要求レベルで実行されます。追加の割り込みの遅延処理は KP_IntAtDpcMSI() およびユーザーモードで実行されます。 **注意:** Linux、Windows Vista およびそれ以降で 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 を返すことによって MSI / MSI-X 割り込みの遅延処理を要求した場合に呼び出されます。 この関数はより優先度の低いカーネルモードの MSI / MSI-X ハンドラ コードを含みます。 この関数の戻り値がアプリケーションのユーザーモードの割り込みハンドラ ルーチンを実行する回数を決定します。

	注意: Linux、Windows Vista およびそれ以降で 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ます (fUseKP 引数に TRUE を設定して WDC_EventRegister() を呼ぶか (Kernel PlugIn ハンドルを開いた後)、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 ドライバへのハンドルを開く / 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 へのハンドルを開いた後 / 関数へ渡す WD_INTERRUPT 構造体の hKernelPlugIn フィールドに設定した Kernel PlugIn へのハンドルで InterruptEnable() または WD_InterruptEnable() を呼び出した後、fUseKP パラメータを TRUE に設定して WDC_IntEnable() 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、Kernel PlugIn ドライバを使用して有効にした割り込みを無効にする場合 (WDC_IntDisable() / InterruptDisable() / WD_IntDisable()) に、それぞれ呼び出されます。

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 の割り込みハンドラは、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 / EventRegister() を読んだ後、fUseKP 引数に TRUE を設定して WDC_EventRegister() を呼び出すか、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が登録した場合、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() コールバックで 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、Kernel PlugIn オープン コールバックは、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;

    /* 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;
    }

    KP_PCI_Trace("KP_PCI_Open entered. PCI library initialized.\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;

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

    COPY_FROM_USER(pAddrDesc, pDevAddrDesc->pAddrDesc, dwSize);
    pDevAddrDesc->pAddrDesc = pAddrDesc;
    *ppDrvContext = pDevAddrDesc;

    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);
    if (pDevAddrDesc)
        free(pDevAddrDesc);
    PCI_LibUninit();
    return FALSE;
}
```

注意: KP_PCI サンプルには、32-bit アプリケーションから 64-bit Kernel PlugIn へのハンドルを開く際に使用するために、同様の `KP_PCI_Open_32_64` コールバックも定義しています。

12.6.2.4 残りの Kernel PlugIn コールバックの記述

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

12.6.3 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** ディレクトリ以下にあります) を中心に説明しています。
Xilinx BMD (Bus Mastering DMA) デザインの PCI-Express カードを使用している場合、開発のベースとしてこのカード用の特定の **KP_BMD** Kernel PlugIn サンプルを使用することができます。
WinDriver¥xilinx¥bmd_design ディレクトリに関連するすべてのサンプル 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。関連するアプリケーションは、DriverWizard を使用して Kernel PlugIn コードを生成した際に、対象のドライバ用に生成されます。**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/.kext** / **kp_xxx.sys/.o/.ko/.kext** – OS によります) 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と割り込み処理を実装しています (割り込み処理の遅延処理、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などを含む)。
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()` 関数のコメントの説明のとおり、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 (**Interrput** タブで、レジスタにカードの割り込みへの読み取り / 書き込みコマンドを指定) で定義した割り込み転送コマンドを実装するコードが含まれます。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に、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ておく必要があります。

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 ドライバを記述またはコードを修正する前に、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。

12.6.4 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

12.6.4.1 pci_diag および kp_pci のサンプル ディレクトリ

kp_pci.c ファイルで、Kernel PlugIn のサンプル コード (**KP_PCI**) を実装してます。このサンプル ドライバは、WinDriver PCI 診断プログラムのサンプル (**pci_diag**) の一部で、`KP_PCI` ドライバに加え、ドライバ (**pci_diag**) と通信を行うユーザーモード アプリケーション、およびそのユーザーモード アプリケーションと Kernel PlugIn ドライバの両方で使用できる API を含む共有ライブラリが含まれます。C 言語でこのサンプルのソースを実装してます。

以下、**WinDriver¥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の開発環境 (**msdev <version>** / **win_gcc**) にあります。
    kp_pci ディレクトリ以下にある **configure** スクリプトを使用して Linux の makefile を生成します。
  - 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 XP およびそれ以降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 およびそれ以降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、ターゲットのマシン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、Linux 用のドライバの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 15.4 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 [12.6.3] 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 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 およびそれ以降で使用する場合 (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の開発環境 (**msdev <version>** / **win_gcc**) にあります。**msdev <version>** MS Visual Studio 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 の `makefile` は、**LINUX** サブ ディレクトリにあります。

- 対象の OS 用のユーザーモード アプリケーション (**pci_diag**) のプリコンパイル済みバージョン
  - Windows: **WIN32¥pci_diag.exe**
  - Linux: **LINUX/pci_diag**

- **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: Xilinx BMD (Bus Mastering DMA) デザインの PCI Express カード用のサンプル ディレクトリ (**WinDriver¥xilinx¥bmd_design**) の構造は、次の項目を除いて、汎用的な PCI のサンプル **pci_diag** ディレクトリと同様です。サンプルの **bmd_diag** ユーザーモード アプリケーションのファイルは、**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_BMD**) のソース ファイルと Windows 用にのみ `makefile` が含まれます。

12.6.4.2 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 [12.6.3] 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とく必要があります。

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# ライブラリ、Kernel PlugIn ドライバと通信する C# ユーザーモード アプリケーションを含む、類似の C# コードを生成することもできます。

- **kermode** - 以下の **KP_XXX** Kernel PlugIn ドライバ ファイルを含みます。
  - **kp_xxx.c**: **KP_XXX** ドライバのソースコード
  - Kernel PlugIn ドライバのビルド用の Project および / または make ファイルと関連ファイル 。Windows プロジェクト ファイルは、**x86** (32 ビット) および **amd64** (64 ビット) ディレクトリ以下のターゲット開発環境 (**msdev_*** / **win_gcc**) にあります。
    LINUX サブ ディレクトリにある **configure** スクリプトを使用して Linux の `makefile` を生成します。
- **xxx_lib.c**: WinDriver の WDC API を使用して、対象のデバイスへアクセスするライブラリの実装。このライブラリの API をユーザーモード アプリケーション (**xxx_diag**) と Kernel PlugIn ドライバ (**KP_XXX**) の両方で使用します。
- **xxx_lib.h**: **xxx_lib** ライブラリのインターフェイスを提供するヘッダー ファイル
- **xxx_diag.c**: サンプルの診断ユーザーモード コンソール (CUI) アプリケーションの実装で、**xxx_lib** と WDC ライブラリを使用して PCI デバイスとの通信を行います。
  このサンプルでは、ユーザーモードの WinDriver アプリケーションから Kernel PlugIn ドライバへのアクセスを行います。デフォルトでは、**KP_XXX** Kernel PlugIn ドライバへのハンドルで、選択した PCI デバイスを開きます。成功した場合、セクション [12.6.3] 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開発環境 (**msdev <version>** / **win_gcc**) にあります。**msdev <version>** MS Visual Studio 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ソリューション ファイルも含まれています。
  Linux の `makefile` は、**LINUX** サブ ディレクトリにあります。
- **xxx_files.**txt: 生成されたファイルの一覧と生成されたコードのビルド手順
- **xxx.inf**: 対象のデバイスの WinDriver INF ファイル (Windows で、PCI または PCMCIA などの Plug and Play デバイスの場合のみ)

12.6.5 Kernel PlugIn での割り込み処理

セクション [12.6.5.2] 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 の割り込み遅延ハンドラ (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 (KP_IntAtDpc() または KP_IntAtDpcMSI()) を有効にし、DPC 関数は高い IRQL ハンドラからの実際の 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。

12.6.5.1 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。

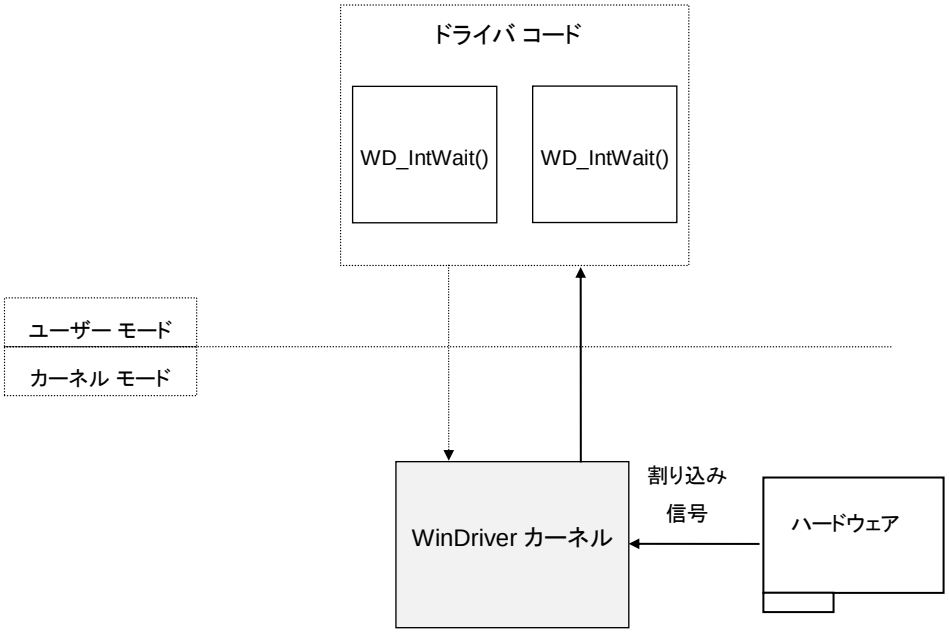


図 11.2: Kernel PlugIn なしでの割り込みの処理

12.6.5.2 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、ユーザーモード アプリケーションが Kernel PlugIn ドライバへのハンドルをオープンし、そして fUseKP パラメータに TRUE を設定して、WDC_IntEnable() を呼びます。

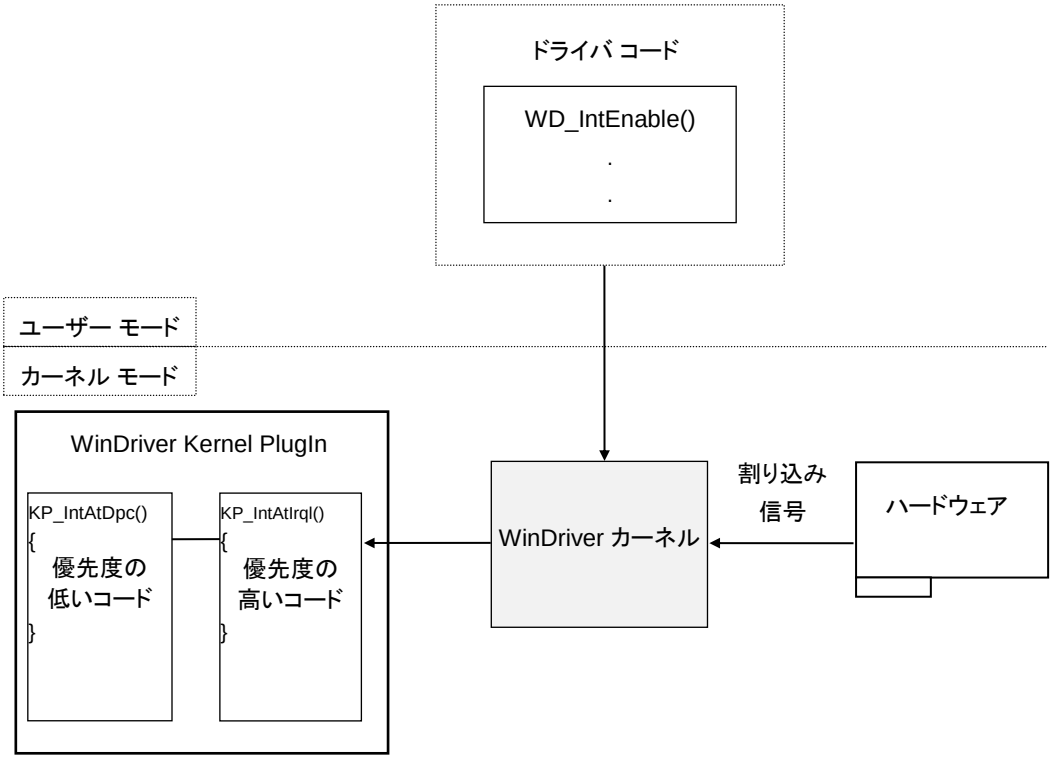


図 11.3: Kernel PlugIn ありでの割り込み処理

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 (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 (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のコードを DPC 関数 (KP_IntAtDpc() または KP_IntAtDpcMSI()) で実装します。DPC 関数は、遅延割り込みレベルで実行され、高い IRQL ハンドラと同じような制限はありません。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。

12.6.6 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 の中で実行されます。ヘッダー ファイル **pci_lib.h** / **xxx_lib.h** の中でメッセージの定義を参照できます。ユーザーモード アプリケーション (**pci_diag.exe** / **xxx_diag.exe**) は、WD_KernelPlugInCall() 関数から Kernel PlugIn ドライバへメッセージを送信します。

第 13 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 12.6.3 および 12.6.4.2 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。
Xilinx BMD (Bus Mastering DMA) デザインの PCI Express カードを使用してる場合、開発のベースとしてこのチップ用の特定の **KP_BMD** Kernel PlugIn サンプルを使用することができます。
WinDriver¥xilinx¥bmd_design ディレクトリに関連するサンプルの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。

13.1 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 11 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)。

13.2 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。このめた、Kernel PlugIn へ移動するコードには、静的なメモリ割り当てを持たないようにしてください。代わりに `malloc()` 関数を使用して、動的にメモリを割り当てます。これは特に大きいデータ構造に重要です。
- 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。

13.3 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 WinDriver の Kernel PlugIn のサンプルを使用して作成することもできます。

注意: MS Visual Studio を使用して Kernel PlugIn プロジェクトをビルドする際には、プロジェクトのディレクトリへのパスにスペースを含めないでください。

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、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。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 12.6.3 およびセクション 12.6.4 を参照してください。

13.4 Kernel PlugIn へのハンドルをオープン

Kernel PlugIn ドライバへのハンドルをオープンするには、ユーザーモードから WD_KernelPlugIn() 関数を呼ぶ必要があります。この低レベルの関数を WDC_PcmciaDeviceOpen() 関数および

WDC_IsaDeviceOpen() 関数の両方から呼びます (これらの関数を Kernel PlugIn ドライバの名前で呼ぶ場合)。

高レベルの WDC API を使用する場合、以下のいずれかの方法を使用して Kernel PlugIn のハンドルをオープンできます:

- 最初に通常のデバイスをハンドルをオープンします (Kernel PlugIn ドライバの名前なしで関連する WDC_xxxDeviceOpen() 関数を呼び出して)。そして WDC_KernelPlugInOpen() 関数を呼んで、オープンしたデバイスへのハンドルを渡します。WDC_KernelPlugInOpen() 関数は Kernel PlugIn ドライバへのハンドルをオープンし、指定したデバイスの構造体の kerPlug フィールド内にハンドルを格納します。
- 関連する WDC_xxxDeviceOpen() 関数を使用して、関数の pcKPDriveName パラメータ内の Kernel PlugIn ドライバの名前を渡して、デバイスへのハンドルをオープンします。関数が返したデバイスのハンドルには、関数がオープンした Kernel PlugIn のハンドルも含まれます (kerPlug フィールド内)。
  注意: この方法を使用して、32-bit アプリケーションから 64-bit Kernel Plugin ドライバへのハンドルをオープンしたり、.NET アプリケーションから Kernel PlugIn ハンドルをオープンすることはできません。

ヒント: すべてのサポートする設定で正しく動作するか確認するには、上記の最初の方法を使用してください。

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、Kernel PlugIn へのハンドルをオープンする方法を実装しています – 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。

ユーザーモードから WD_KernelPlugInClose() を呼んだ場合、Kernel PlugIn ドライバへのハンドルをクローズします。低レベルの WinDriver API を使用する場合、ユーザーモード アプリケーションから直接この関数を呼びます。高レベルの WDC API を使用する場合、オープンした Kernel PlugIn ハンドルを含むデバイスのハンドルで WDC_xxxDeviceClose() 関数を呼ぶ際に、自動的に関数を呼びます。この関数は、どの特定のオープンした Kernel PlugIn のハンドルに対しても、アプリケーションのクリーンナップの一部として WinDriver によっても呼び出されます。

13.5 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 13.4 で説明したとおり、Kernel PlugIn ドライバへのハンドルをオープンした後)、fUseKP 関数の引数を TRUE に設定して、オープンしたデバイスに対して Kernel PlugIn ドライバで割り込みを有効にします。
   DriverWizard で生成されたコードおよびサンプルの **pci_diag** 共有ライブラリ (**xxx_lib.c** / **pci_lib.c**) で実装してます – 生成されたコードまたはサンプルの XXX_IntEnable() / PCI_IntEnable() ライブラリ関数 (生成されたコードまたはサンプルの **xxx_diag** / **pci_diag** ユーザーモード アプリケーションから呼ばれます) を参照してください。
   WDC_xxx API を使用しない場合、Kernel PlugIn で割り込みを有効にするには、WD_IntEnable() または (WD_IntEnable() を呼び出す) InterruptEnable() を呼び出して、WD_KernelPlugInOpen() から受信した Kernel PlugIn へのハンドル (関数へ渡された WD_KERNEL_PLUGIN 構造体の hKernelPlugIn フィールド内) を渡します。
2. WDC_IntEnable() / InterruptEnable() / WD_IntEnable() を呼んで、Kernel PlugIn で割り込みを有効にする場合、WinDriver は Kernel PlugIn の KP_IntEnable() コールバック関数 を有効にします。この関数を実装して、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 (`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 (`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 12.6.5 を参照してください。

13.6 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。

13.7 Kernel PlugIn ドライバのコンパイル

注意: Kernel PlugIn は後方互換性はありません。従って、WinDriver のバージョンを変更する場合には、新しいバージョンを使用して、Kernel PlugIn ドライバをリビルドする必要があります。

13.7.1 Windows で Kernel PlugIn ドライバをコンパイル

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>¥kermode`** ディレクトリ (**`<project_dir>`** は、生成されたドライバ プロジェクトを保存したディレクトリ) には、以下の Kernel PlugIn プロジェクト ファイルが含まれます (**`xxx`** はドライバ名。サンプルの場合は **`pci`**、またはウィザードでコードを生成する際に指定した名前):

- **`x86`**: 32 ビット プロジェクト ファイル
  - **`msdev <version>¥kp_xxx.vcproj`**: 32 ビット MS Visual Studio プロジェクト ファイル (**`<version>`** は、IDE のバージョンを表します – 例えば “2012”)
  - **`win_gcc¥makefike:`** 32 ビット Windows GCC (MinGW / Cygwin) makefik**e**
- **`amd64`**: 64 ビット プロジェクト ファイル
  - **`msdev <version>¥kp_xxx.vcproj`**: 64 ビット MS Visual Studio プロジェクト ファイル (**`<version>`** は、IDE バージョンを表します – 例えば “2012”)

- **win_gcc¥makefike:** 64 ビット Windows GCC (MinGW / Cygwin) makefik**e**

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 <version>¥xxx_diag.vcproj**: 32 ビット MS Visual Studio プロジェクト ファイル (**<version>** は、IDE バージョンを表します – 例えば “2012”)
  - **win_gcc¥makefike:** 32 ビット **W**indows GCC (MinGW / Cygwin) makefike
- **amd64: 64** ビット プロジェクト ファイル
  - **msdev <version>¥xxx_diag.vcproj:** 64 ビット MS Visual Studio プロジェクト ファイル (<version> は、IDE バージョンを示します – 例えば “2012”)
  - **win_gcc¥makefike:** 64 ビット Windows GCC (MinGW / Cygwin) makefike

上記の **msdev <version>** MS Visual Studio 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.sln** ソリューション ファイルも含まれています。

DriverWizard を使用してコードを生成し、DLL (ダイナミックリンク ライブラリ) の生成を選択した場合、生成した **<project_dir>** ディレクトリには **libproj** DLL プロジェクトのディレクトリも含まれます。このディレクトリには、**x86** (32-bit) と / または **amd64** (64-bit) ディレクトリがあり、選択した IDE 用に **msdn_<version>** ディレクトリが含まれ、各 IDE のディレクトリには、DLL のビルド用に **xxx_libapi.vcproj** プロジェクト ファイルがあります。DriverWizard で生成したユーザーモードの診断プロジェクト (**xxx_diag.vcproj**) から DLL を使用します。Windows で 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. WDK (Windows Driver Kit) がインストールされていることを確認します。
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. Kernel PlugIn SYS ドライバ (サンプルの **kp_pci.sys**、または DriverWizard で生成された **kp_xxx.sys**) をビルドします。
   - MS Visual Studio を使用する場合 - Microsoft Visual Studio を起動し、下記を実行します。
     - ドライバのプロジェクト ディレクトリから、Visual Studio Kernel PlugIn ソリューション ファイル (**<project_dir>¥msdev <version>¥xxx_diag.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 <version>** は、ターゲットの Visual Studio ディレクトリ (例 **msdev_2012**) 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)。

       注意: DriverWizard で MS Visual Studio 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、[IDE to Invoke] オプションを使用して、選択した IDE で Wizard が自動的に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 ファイルを開きます。
       MS Visual Studio を使用して Kernel PlugIn プロジェクトをビルドする際には、プロジェクトのディレクトリへのパスにスペースを含めないでください。

- Kernel PlugIn プロジェクト (**kp_pci.vcproj** または **kp_xxx.vcproj**) をアクティブなプロジェクトとして設定します。
- 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] を選択し、使用する構成を選択します。
- 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。

  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 XP およびそれ以降をサポートする場合、**Win32 winxp free** (リリース モード) または **Win32 winxp checked** (デバッグ モード) のどちらかを選択します。

- ドライバをビルドします。ショートカット キー (Visual Studio 2008 では F7 キー) を押すか、[Build] メニューから実行してください。

- Windows GCC を使用する場合 – 選択した Windows GCC 開発環境 (MinGW / Cygwin) から下記のステップを実行します。
  - ターゲットの Windows GCC Kernel PlguIn プロジェクトのディレクトリへ移動します - **<project_dir>¥<kernel_dir>¥<CPU>¥win_gcc**、**<project_dir>** は対象のドライバのプロジェクト ディレクトリ (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)、**<kernel_dir>** は対象のプロジェクトの Kernel PkugIn ディレクトリ (サンプル コードの場合は **kp_pci**、または DriverWizard で生成されたコードの場合は **kermode**)、**<CPU>** は対象の CPU アーキテクチャ (x86 プラットフォームの場合は **x86**、x64 プラットフォームの場合は **amd64**) です。
    たとえば、サンプルの KP_PCI ドライバの 32-bit バージョンをビルドする場合:
    $ cd WinDriver¥samples¥pci_diag¥kp_pci¥x86¥win_gcc

    DriverWizard で生成された Kernel PlugIn ドライバの 64-bit バージョンをビルドする場合:
    $ cd <project_dir>¥kermode¥amd64¥win_gcc
    <project_dir> は DriverWizard で生成されたプロジェクト ディレクトリへのパスです (たとえば、~**¥WinDriver¥wizard¥my_projects¥my_kp**)。
  - Kernel PlugIn の makefile の ddk_make.bat コマンドを編集して、対象のビルド構成を設定します – つまり、ターゲットの OS とビルド モード (release – `free`、または debug - `checked`)。デフォルトでは、WinDriver のサンプルと DriverWizard で生成された makefile には、ターゲットの OS のパラメータに Windows XP (32-bit の場合 `winxp`、64-bit の場合 `x64`)、およびビルド モードに release (`free`) を設定します。

注意:

- ddk_make.bat ユーティリティは WinDriver\util ディレクトリ以下にあり、ビルド コマンドを起動する際に、Windows が自動的に認識します。パラメータなしで ddk_make.bat を起動すると、このユーティリティで利用可能かオプションを表示します。
- 選択したビルド OS は、WinDriver の CPU アーキテクチャのバージョンと一致させてください。たとえば、WinDriver 32-bit 版を使用している場合には、64-bit `win7_x64` OS フラグを選択できません。
- 複数の OS 用にドライバをビルドする場合には、ドライバがサポートする最も下位の OS バージョンを選択してください。たとえば、Windows XP およびそれ以降をサポートする場合、OSパラメータには、`winxp` (32-bit の場合) または `x64` (64-bit の場合) を設定してください。

- `make` コマンドを使用して Kernel PlugIn ドライバをビルドします。

4. Kernel PlugIn ドライバと動作するユーザーモード アプリケーションをビルドします (サンプルの **`pci_diag.ese`**、または DriverWizard で生成された **`xxx_diag.exe`**)。
   - MS Visual Studio を使用する場合:
     - ユーザーモードのプロジェクトをアクティブなプロジェクトとして設定します (サンプルの場合は **`pci_diag.vcproj`**、または DriverWizard で生成されたコードの場合は **`xxx_diag.vcproj`**)。
     - アプリケーションをビルドします: **Buid** メニューからプロジェクトをビルドするか、関連するショートカット キー (たとえば、Visual Studio 2008 の場合 **F7**) を使用してビルドします。
   - Windows GCC を使用する場合 – 選択した Windows GCC 開発環境 (MinGW / Cygwin) から下記のステップを実行します:
     - ターゲットの Windows GCC アプリケーションのディレクトリへ移動します - **`<project_dir>¥<CPU>¥win_gcc`**、**`<project_dir>`** は対象のドライバのプロジェクト ディレクトリ (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)、**`<CPU>`** は対象の CPU アーキテクチャ (x86 プラットフォームの場合は **`x86`**、および x64 プラットフォームの場合は **`amd64`**) です。
       たとえば:
       サンプルの pci_diag アプリケーションの 32-bit バージョンをビルドする場合:
       `$ cd WinDriver¥samples¥pci_diag¥x86¥win_gcc`

       DriverWizard で生成されたユーザーモードのアプリケーションの 64-bit バージョンをビルドする場合:
       `$ cd <project_dir>¥amd64¥win_gcc`
       `<project_dir>` は DriverWizard で生成されたプロジェクト ディレクトリへのパスです (たとえば、~**`¥WinDriver¥wizard¥my_projects¥my_kp`**)。
     - `make` コマンドを使用してアプリケーションをビルドします。

13.7.2 Linux でKernel PlugIn ドライバをコンパイル

Linux で Kernel PlugIn ドライバと関連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てください:

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。
   $ **cd <project_dir>/kermode/linux**
   (**<project_dir>** は DriverWizard で生成されたプロジェクトのディレクトリへのパスです (たとえば、~**/WinDriver/wizard/my_projects/my_kp**)。

3. **configure** スクリプトを使用して、**makefile** を生成します。
   $ **./configure**

   ヒント: WinDriver のカーネル モジュールの名前を変更した場合、スクリプトを実行する前に、**-DWD_DRIVER_NAME_CHANGE** フラグでドライバをビルドするので、Kernel PlugIn の **configure** スクリプトの以下の行のコメントを外してください ("#" を削除)。
   # **ADDITIONAL_FLAGS="-DWD_DRIVER_NAME_CHANGE"**

   注意:

   - **configure** スクリプトは、起動してるカーネルをベースとしたカーネル独自の **makefile** を作成します。**configure** スクリプトに **--with-kernel-source=<path>** フラグを追加することによって、インストールした他のカーネル ソースをベースとした **configure** スクリプトを起動できます。<path> には、カーネル ソースのディレクトリへのフル パスを指定します。
   - 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** スクリプトは、**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** スクリプトに **--enable-kbuild** フラグを渡します。

   ヒント: **configure** スクリプトのオプションの一覧を表示するには、以下のように --help オプションを使用します:
   ./configure --help

4. **make** コマンドを使用して Kernel PlugIn モジュールをビルドします。

   このコマンドは、作成された **kp_xxx_module.o/.ko** ドライバを含む、新しい **LINUX.<kernel version>.<CPU>** ディレクトリを作成します。

5. サンプル ユーザーモード診断アプリケーションの makefile のあるディレクトリに移動します。
   KP_PCI サンプル ドライバの場合:
   cd ../LINUX

   DriverWizard で生成された Kernel PlugIn ドライバの場合:
   cd ../../linux

6. **make** コマンドを使用してサンプル診断プログラムをコンパイルします。

13.8 Kernel PlugIn ドライバのインストール

13.8.1 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。

1. ドライバ ファイル (**xxx.sys**) をターゲット プラットフォームのドライバ ディレクトリ **%windir%¥system32¥drivers** にコピーします (例: **C: ¥Windows¥system32¥drivers**)。

2. **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します。そのため、ロードおよびアンロード時にリブートする必要はありません。

13.8.2 Linux の場合

1. Kernel PlugIn ドライバのディレクトリに移動します。

   たとえば、サンプル **KP_PCI** ドライバをインストールする場合は、次のコマンドを実行します。

   ```
   $ cd WinDriver/samples/pci_diag/kp_pci
   ```

   DriverWizard で生成された Kernel PlugIn ファイルを使用して作成したドライバをインストールする場合は、次のコマンドを実行します (**<path>** は生成された DriverWizard プロジェクトのディレクトリ。例: **~/WinDriver/wizard/my_projects/my_kp**):

   ```
   $ cd <path>/kermode
   ```

2. 次のコマンドを実行して、Kernel PlugIn ドライバをインストールします:

   注意: root 権限で下記のコマンドを実行してください。

   ```
   # make install
   ```

   注意: Kernel PlugIn ドライバは動的にロードします。そのため、ロードおよびアンロード時にリブートする必要はありません。

第 14 章
ドライバの動的ロード

14.1 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 12 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 のサービス (**windrvr1221.sys** または名前変更したドライバ) へのハンドルが開いていないことをご確認ください。また、WinDriver のサービスと動作するように登録したPlug-and-Play デバイスが接続されていない、および有効になっていないことをご確認ください。

14.2 Windows の動的ドライバ ロード

Windows XP およびそれ以降で WDM (Windows Driver Model)ドライバを使用する場合: ***.sys** 拡張子のファイル (たとえば、**windrvr1221.sys)** です。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。

WinDriver Windows のカーネル モジュール (**windrvr1221.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。

14.2.1 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**) DLL - **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。

14.2.1.1 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 **windrvr1221.sys** ドライバをインストールする方法、または Plug-and-Play のデバイス (PCI または PCMCIA など) および USB デバイスを **windrvr1221.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、このセクションの説明には当てはまりません。Windows 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 14.2.1.2 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: ドライバの無効 / アンロードを行うには、WinDriver のサービス (**windrvr1221.sys** または名前変更したドライバ) へのハンドルが開いていないことをご確認ください。また、WinDriver のサービスと動作するように登録したPlug-and-Play デバイスが接続されていない、および有効になっていないことをご確認ください。

14.2.1.2 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 上での Kernel PlugIng ドライバ) をインストールする方法を説明します。

使用法:

wdreg [-file <ファイル名>] [-name <ドライバ名>] [-startup <level>] [-silent] [-log <ログファイル>] Action [Action ...]

- オプション

wdreg は次の基本オプションをサポートします。

- **[-startup]** – ドライバを開始するときに指定します。以下の引数の 1 つが必要となります。
  - **boot:** オペレーティング システムのローダーで開始されるドライバを表示します。そして、OS (たとえば、Atdisk) をロードする必要のあるドライバにのみ使用されます。
  - **system:** OS の初期化中に開始されたドライバを表示します。
  - **automatic:** システムが起動中に Service Control Manager によって開始されたドライバを表示します。
  - **demand:** 要求に応じて Service Control Manager によって開始されたドライバを表示します (ドライバをプラグインした場合など)。
  - **disabled:** 開始されないドライバを表示します。

注意: デフォルトでは、**-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 windrvr1221 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。
- 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。

14.2.2 windrvr1221.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 **windrvr1221.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 **windrvr1221.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 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:

- **windrver6.inf** をロードして **windrvr1221.sys** のサービスを開始するには、次のコマンドを実行します:
  wdreg -inf [windrvr1221.inf へのパス] install

- 例えば **C:¥temp** ディレクトリにある **device.inf** と言う名前の 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。

注 意 : **ERROR_FILE_NOT_FOUND** エラーでインストールに失敗する場合、**HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion** に **RunOnce** キーが存在するが Windows のレジストリを確認してください。INF ファイルを使用して正しくドライバをインストールするには、Windows Plug-and-Play にはこのレジストリキーが必要です。**RunOnce** キーがない場合、作成して、再度 INF ファイルをインストールしてください。

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。

14.2.3 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 **windrvr1221.sys**をロードした後に、このドライバをロードする必要があります。
ドライバをアンロードする際には、**windrvr1221.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。

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 **windrvr1221.sys** の説明のように、**"name"** フラグ (Kernel PlugIn ドライバの名前を追加した後) をつけて、**wdreg** コマンドを使用します。

注意: ドライバ名に拡張子 ***.sys** を追加しないでください。

例:

- **KPDriver.sys** と呼ばれる KrenelPlugin ドライバをロードするには、次のコマンドを実行します:
  wdreg -name KPDriver install
- **MPEG_Encoder** と呼ばれる Kernel PlugIn ドライバを **MPEGENC.sys** とロードするには、次のコマンドを実行します:
  wdreg -name MPEG_Encoder -file MPEGENC install
- **KPDriver.sys** と呼ばれる KernelPlugIn ドライバをアインストールするには、次のコマンドを実行します:
  wdreg -name KPDriver uninstall
- **MPEG_Encoder** と呼ばれる KernelPlugIn ドライバと **MPEGENC.sys** ファイルをアインストールするには、次のコマンドを実行します:
  wdreg -name MPEG_Encoder -file MPEGENC uninstall

14.3 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 windrvr1221
- 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 windrvr1221

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 (例えば、**/etc/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> windrvr1221

14.3.1 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 **windrvr1221.o/.ko** をロードした後に、このドライバをロードする必要があります。
ドライバをアンロードする際には、**windrvr1221.o/.ko** 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。

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。以下のコマンドを使用して、Kernel PlugIn ドライバを動的にロード / アンロードします。

注意: root 権限で次のコマンドを実行してください。

注意: コマンド内の **xxx** は選択した Kernel PlugIn ドライバのプロジェクト名を表します。

- 対象の KrenelPlugin ドライバをロードするには、次のコマンドを実行します:
  # /sbin/insmod <kp_xxx_module.o/.ko へのパス>

注意: 開発環境のマシンで Kernel PlugIn ドライバをビルドする場合、対象の Kernel PlugIn プロジェクトの **kermode/linux/LINUX.<kernel version>.<CPU>** ディレクトリで Kernel PlugIn モジュールを作成します。ターゲットの配布先のマシンでドライバをビルドする場合、通常、**xxx_installation/redist/LINUX.<kernel version>.<CPU>.KP** ディレクトリでドライバ モジュールを作成します。

- Kernel PlugIn を動的にアンロードするには、次のコマンドを実行します:
  # /sbin/rmmod kp_xxx_module

注意: 対象の Linux のブート ファイル (例えば、**/etc/rc.local**) に次の行を追加して、WinDriver のドライバ モジュール (windrvr1221) のロード コマンドの後 (対象の Kernel PlugIn モジュールへのパスを **<kp_xxx_module.o/.ko へのパス>** に置換)、システムを起動するごとに Kernel PlugIn ドライバを自動的にロードします:
 # /sbin/insmod <kp_xxx_module.o/.ko へのパス>

14.4 Windows CE 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 (**windrvr1221.dll**) を Windows Mobile プラットフォームにロードします。

ヒント: Windows CE の多くのバージョンでは、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 のカーネル モジュールをリロードする必要があります。Windows CE の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** ユーティリティをターゲットの PC の **Windows¥StartUp** ディレクトリにコピーします。

Windows CE 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 15 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。

15.1 WinDriver の有効なライセンスを取得するには

ドライバを配布する前に、WinDriver の有効なライセンスを取得する必要があります。ライセンス コードの申請方法に関しては、別紙の「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」を参照してください。Registered 版 (登録版) の WinDriver を開発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。

15.2 Windowsの場合

注意:

- この章の説明の "**wdreg**" と記述している個所を "**wdreg_gui**" に置き換えることができます。同じ機能ですが、コンソール モード メッセージの代わりに GUI メッセージが表示されます。
- WinDriver のカーネル モジュール (**windrvr1221.sys**) の名前を変更する場合、**windrvr1221** 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) **windrvr1221.cat** ファイルへの参照を新た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)。

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 **windrvr1221.sys** と **windrvr1221.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。

15.2.1 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。

注意: 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。

- ハードウェア コントロール アプリケーション / DLL
- **windrvr1221.sys** (**WinDriver¥redist** ディレクトリにあります)。
- **windrvr1221.inf** (**WinDriver¥redist** ディレクトリにあります)。
- **wd1221.cad** (**WinDriver¥redist** ディレクトリにあります) 。
- **wd_api1221.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)。
  wdapi1221_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)

15.2.2 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

  ドライバをインストールする際には、WinDriver のサービス (**windrvr1221.sys** または名前変更したドライバ) へのハンドルが開いていないことをご確認ください。また、WinDriver のサービスと動作するように登録したPlug-and-Play デバイスが接続されていない、および有効になっていないことをご確認ください。たとえば、これはドライバのバージョンをアップグレードする際にも関連します。サービスが使用されている場合には、**wdreg** を使用して新しいドライバをインストールすると失敗します。デバイス マネージャから接続したデバイスを無効またはアンインストールするか ([プロパティ] – [無効] または [削除]) 、もしくは PC からデバイスを物理的に切断します。
  注意: WinDriver v11.90 以降、デフォルトのドライバのモジュール名には、WinDriver のバージョン番号が含まれます。そのため、ドライバのモジュール名を以前に使用していたモジュール名に変更しなくて、以前のバージョンのドライバとの競合の問題は起こりません。

- WinDriver のカーネルモ ジュールのインストール:

  ① **windrvr1221.sys**、**windrvr1221.inf** と **windrvr1221.cat** ファイルを同じディレクトリにコピーします。
  注意: **windrvr1221.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、**windrvr1221.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 (**windrvr1221.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 。
  - **windrvr1221.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 = windrvr1221.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定および対象の 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。

② **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 <windrvr1221.inf のパス> install
```

たとえば、**windrvr1221.inf** および **windrvr1221.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¥windrvr1221.inf install
```

WinDriver ツールキットの **WinDriver¥util** ディレクトリ以下にがあります。このユーティリティの一般的な説明および使用方法に関しては、第 14 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。

注意: ドライバの配布時に、新しいバージョンの **windrvr1221.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。
windrvr1221.sys ファイルは **COPYFLG_NO_VERSION_DIALOG** INF ディレクティブを使用します。これは、コピー先の既存のファイルがコピー元のファイルよりも新しい場合、コピー元のファイルでコピー先のファイルを上書きしないようにデザインされています。同様に **COPYFLG_OVERWRITE_OLDER_ONLY** INF ディレクティブがあります。これは、コピー先のファイルがより新しいバージョンに取って代えられる場合のみ、コピー元のファイルをコピー先のディレクトリへコピーするようにデザインされています。ただし、これらの INF ディレクティブの両方は、デジタル署名されたドライバには適用されませんので、ご注意ください。Microsoft の INF CopyFiles Directive ドキュメントの説明のとおり (http://msdn.microsoft.com/en-us/library/ff546346%28v=vs.85%29.aspx)、ドライバ パッケージがデジタル署名されている場合、Windows はパッケージ全体をインストールし、対象のコンピュータに既に他のバージョンが存在する場合でもパッケージのファイルを選択して除外しません。Jungo Connectivity から提供する **windrvr1221.sys** ドライバはデジタル署名されています。

- 対象のデバイスの INF ファイルのインストール (**windrvr1221.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： **ERROR_FILE_NOT_FOUND** エラーでインストールに失敗する場合、**HKEY_LOCAL_MACHINE¥SOFTWARE¥Microsoft¥Windows¥CurrentVersion** に **RunOnce** キーが存在するが Windows のレジストリを確認してください。INF ファイルを使用して正しくドライバをインストールするには、Windows Plug-and-Play にはこのレジストリ キーが必要です。**RunOnce** キーがない場合、作成して、再度 INF ファイルをインストールしてください。

- 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 15.2.3 の手順に従って、ドライバをインストールします。

- **Wdapi1221.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 **wdapi1221.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、**wdapi1221_32.dll** から **wdapi1221.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 **wdapi1221.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。

15.2.3 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 14 章 を参照してください (特に、セクション 14.2.3 の「Kernel PlugIn のインストール」を参照してください)。

15.3 Windows CE の場合

15.3.1 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 MS Visual Studio 2005 / 2008 と Windows CE 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新規のターゲット 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. プロジェクト レジストリ ファイル (**WinDriver¥samples¥wince_install¥project_wd.reg**) を編集して、対象のターゲット デバイスのエントリを追加します (まだ実行していない場合)。
2. Windows CE プラットフォーム (Sysgen stage) をコンパイルします。
3. 対象のプラットフォームにドライバを統合します。
    a. Windows CE IDE を実行してプラットフォームを開きます。
    b. **Build** メニューから **Open Release Directory** を選択します。
    c. WinDriver CE カーネル ファイル - **WinDriver¥redist¥<TARGET_CPU>¥windrvr1221.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 (新しいコマンド ウィンドウのカレント ディレクトリになるはずです)。
    d. **WinDriver¥samples¥wince_install¥project_wd.reg** ファイルの内容を **%_FLATRELEASEDIR%¥project.reg** レジストリ ファイルに追加します。
    e. **WinDriver¥samples¥wince_install¥project_wd.bib** ファイルの内容を **%_FLATRELEASEDIR%¥project.bib** バイナリ イメージ ビルダー ファイルの FILES セクションにコピーします。ターゲット プラットフォームに一致する行のコメントを外します (コピーしたテキストの “TODO” コメントを参照してください)。

    注意: WinDriver CE カーネル ファイル (**windrvr1221.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 **windrvr1221.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。
4. **Build** メニューより **Make Run-Time Image** を選択し、新しいイメージ **NK.BIN** の名前をつけます。
5. 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 **Attache Device** を選択するか、またはブート ディスクを使用して初期化します。Windows CE 4.x の場合、メニューの名前は、**Attache Device** ではなく **Download** / **Initialize** です。
6. 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。
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 **wdapi1221.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。

15.3.2 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 CE カーネル モジュール - **windrvr1221.dll** – を 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。

2. 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 Editor を使用するか、MS eMbedded Visual C++ または MS Visual Studio 2005 / 2008 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。

   - 多くの Windows CE のバージョンでは、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 CE 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。

3. 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) 。

4. ターゲットにハードウェア 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 **wdapi1221.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。

15.4 Linux の場合

ドライバを配布するには、セクション 14.4.1 の説明のとおり、必要なファイルを含む配布パッケージを用意します。そして、セクション 14.4.2 – 14.4.4 の説明のとおり、ターゲットに必要はドライバ コンポーネントをビルドしてインストールします。

注意:

- WinDriver のドライバ モジュールの名前を変更した場合、以下の手順の **windrvr1221** への参照を名前を変更したドライバ モジュールの名前に置き換えて参照してください。
- ターゲットでビルドとインストール手順を自動化するインストール シェル スクリプトを用意することを推奨します。

15.4.1 配布パッケージの用意

このセクションの説明のとおり、必要なファイルを含む配布パッケージをを用意します。

注意:

- 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配布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。
- 以下の手順で、<**source_dir**> は配布ファイルをコピーするコピー元のソース ディレクトリを表します。デフォルトのソース ディレクトリは WinDriver のインストール ディレクトリです。
  ただし、WinDriver のドライバ モジュールの名前を変更した場合、ソース ディレクトリは、名前を変更したドライバをコンパイルおよびインストールするために変更したファイルを含むディレクトリです。DriverWizard を使用してドライバ コードを生成した場合、名前を変更したドライバのソース ディレクトリは、生成された **xxx_installation** ディレクトリです (xxx は生成されたドライバ プロジェクトの名前です)。

15.4.1.1 カーネル モジュール コンポーネント

WinDriver ベースのドライバは、WinDriver API を実装する **windrvr1221.o/.ko** はカーネル モジュールに依存します。さらに、Kernel PlugIn ドライバを作成した場合、**kp_xxx_module.o/.ko** カーネル ドライバ モジュールで、このドライバの機能を実装します (xxx は選択した対象のドライバのプロジェクト名)。

注意: カーネル ドライバ モジュールはそのままでは配布できません。ターゲットのカーネル バージョンと一致させるためにターゲット マシンごとに再コンパイルする必要があります。これは次の理由のためです: Linux のカーネルは開発途中で、カーネル データ構造体はたびたび変更されます。このような流動的な開発環境をサポートし、安定したカーネルを作成するために、Linux のカーネル開発者は、カーネル自身をコンパイルしたヘッダー ファイルと同一のヘッダーファイルを使用して、カーネル モジュールをコンパイルすることを決定しました。バージョン番号がカーネル ヘッダー ファイルに挿入され、カーネルにエンコードされているバージョンと照合されます。Linux のドライバ開発者は、ターゲット システムのカーネル バージョンでドライバを再コンパイルする必要があります。

以下、ターゲットのマシンで対象のカーネル ドライバ モジュールをコンパイルするのに配布が必要なコンポーネントのリストです。

注意: コピー元のサブディレクトリと一致する配布先のディレクトリのサブディレクトリへファイルをコピーすることを推奨します (指定した場合を除き、redist および include など)。そのように選択しない場合、**configure** スクリプトと関連する makefile のテンプレートのファイル パスを変更し、配布先のディレクトリのファイルの場所と一致させる必要があります。

- **<source_dir>/include** ディレクトリから、**windrvr.h**、**wd_ver.h**、および **windrvr_usb.h** をコピーします (ターゲットでカーネル モジュールをビルドするのに必要)。**windrvr_usb.h** は USB 以外のドライバにも必要なのでご注意ください。
- **<WinDriver installation directory>/util** ディレクトリから (または DriverWizard で生成された **xxx_installation/redist** ディレクトリから)、wdreg をコピーします (WinDriver のカーネル モジュールをロードするスクリプト)。
- **<source_dir>/include** ディレクトリから、特に指定がない限り、以下のファイルをコピーします:
  - **setup_inst_dir**: **wdreg** を使用して、WinDriver のドライバ モジュールをインストールするスクリプト。
  - **linux_wrappers.c/h**: カーネル モジュールを Linux カーネルに結合するラッパー ライブラリのソース コード ファイル。

- **linux_common.h** および **wdusb_interface.h**: ターゲット マシンで、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 以外のドライバにも必要なので、ご注意ください。

- 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_v3.a**: GCC v3.x.x コンパイル用
  - **windrvr_gcc_v3_regparm.a**: **regparm** フラグで GCC v3.x.x コンパイル用
  - **windrvr_gcc_v2.a**: GCC v2.x.x コンパイル用 (このファイルは 64-bit 版の WinDriver のインストールにはありません。64-bit Linux アーキテクチャでは GCC v2 を使用しません。)
- WinDriver のカーネル ドライバ モジュールをビルドおよびインストールするための **configure** スクリプトと makefile のテンプレート。
  注意: 名前に .kbuild を含むファイルは、ドライバのコンパイルに kbuild を使用してください。
- **configure**: **makefile.in** テンプレートを使用して WinDriver のドライバ モジュールをビルドおよびインストールするための makefile を作成し、**configure.wd** スクリプトを実行する **configure** スクリプト。
- **configure.wd**: **makefile.wd[.kbuild].in** テンプレートを使用して **windrvr1221.o/.ko** ドライバ モジュールをビルドするための **makefile.wd[.kbuild]** makefile を作成するスクリプト。
- **makefile.in**: **makefile.wd[.kbuild]** を使用して WinDriver のカーネル ドライバ モジュールをビルドおよびインストールするためのメインの makefile 用のテンプレート。
- **makefile.wd.in** と **makefile.wd.kbuild.in**: **windrvr1221.o/.ko** ドライバ モジュールをビルドおよびインストールするために makefile.wd[.kbuild] makefile を作成するためのテンプレート。

Kernel PlugIn ドライバを作成した場合、同様に以下のファイルをコピーしてください:

- DriverWizard で生成された **xxx_installation/redist** ディレクトリから (**xxx** は対象のドライバ プロジェクトの名前)、以下の **configure** スクリプトと makefile テンプレートをコピーします (Kernel PlugIn ドライバをビルドおよびインストールするための makefile を作成用)。

  注意: DriverWizard を使用して Kernel PlugIn ドライバを生成しない場合、対象の Kernel PlugIn プロジェクトからファイルをコピーしてください
  (たとえば、**WinDriver/samples/pci_diag/kp_pci** ディレクトリいかにある **KP_PCI** サンプルのファイル)。
  ファイルをコピーする前に、WinDriver のドライバ モジュール ファイルと区別するために、ファイル名に ".kp" を追加して名前を変更してください (**xxx_installation/redist** ファイル名)。またファイル名とファイルのパスを編集して、配布先のディレクトリの構造に一致させる必要があります。

- **configure.kp**: **makefile.kp[.kbuild].in** テンプレートを使用して、Kernel PlugIn ドライバ モジュールをビルドおよびインストールすための **makefile.kp** makefile を作成する **cofigure** スクリプト。

  注意: WinDriver のカーネル モジュールの名前を変更した場合、スクリプトを実行する前に、**-DWD_DRIVER_NAME_CHANGE** フラグでドライバをビルドするので、Kernel PlugIn の **configure**

スクリプトの以下の行のコメントを外してください (“#” を削除)。
ADDITIONAL_FLAGS=”-DWD_DRIVER_NAME_CHANGE”

- **makefile.kp.in** と **makefile.kp.kbuild.in**: Kernel PlugIn ドライバ モジュールをビルドおよびインストールするために **makefile.kp** makefile を作成するためのテンプレート。
- **<source_dir>/lib** ディレクトリから、コンパイル済みの WinDriver の API オブジェクト コードをコピー:
  - **kp_wdapi1221_gcc_v3.a**: GCC v3.x.x コンパイル用
  - **kp_wdapi1221_gcc_v3_regparm.a**: **regparm** フラグで GCC v3.x.x コンパイル用
  - **kp_wdapi1221_gcc_v2.a**: GCC v2.x.x コンパイル用 (このファイルは 64-bit 版の WinDriver のインストールにはありません。64-bit Linux アーキテクチャでは GCC v2 を使用しません。)
- 開発マシンで Kernel PlugIn をビルドする際に生成される **kermode/linux/LINUX.<kernel version>.<CPU>** ディレクトリから、Kernel PlugIn ドライバ モジュールをビルドするためのコンパイル済みのオブジェクト コードを lib 配布サブディレクトリへコピー (xxx は Kernel PlugIn ドライバ プロジェクトの名前):
  - **kp_xxx_gcc_v3.a**: GCC v3.x.x コンパイル用
  - **kp_xxx_gcc_v3_regparm.a**: **regparm** フラグで GCC v3.x.x コンパイル用
  - **kp_xxx_gcc_v2.a**: GCC v2.x.x コンパイル用 (このファイルは 64-bit 版の WinDriver のインストールにはありません。64-bit Linux アーキテクチャでは GCC v2 を使用しません。)

15.4.1.2 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配布パッケージにコピーします。

(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 **libwdapi1221.so** を使用する場合、このファイルを **<source_dir>/lib** ディレクトリから、対象の配布パッケージにコピーします。ターゲットの 64-bit プラットフォームに 32-bit アプリケーション / 共有オブジェクトを配布する場合、**WinDriver/lib** ディレクトリの **libwdapi1221_32.so** を対象の配布パッケージへコピーし、そのコピーを **libwdapi1221.so** に名前を変更します。

対象のハードウェア 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 オブジェクト (ソース コードの無許可のコピーを防ぎたい場合) として配布してもかまいません。ただし、対象のドライバのソース コードを配布する場合、Jungo Connectivity 社とのソフトウェア ライセンス契約により、**libwdapi1221.so** 共有オブジェクトのソースコード、またはコード内の WinDriver のライセンス コードを配布することは禁じられているのでご注意ください。

15.4.2 ターゲットで WinDriver のドライバ モジュールをビルドおよびインストール

configure スクリプトと関連するビルドおよびインストール ファイルを含む配布パッケージのサブディレクトリから (通常 **redist** サブディレクトリ)、以下のステップを実行してターゲットでドライバ モジュールのビルドとインストールを実行します:

1. 必要な 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に makefile を作成します。**--with-kernel-source=<path>** でスクリプトを実行することで (**<path>** はカーネルのソース ディレクトリへのフルパスです。たとえば、**/usr/src/linux**)、他のインストール済みのカーネルのソースを選択することができます。
   Linux のカーネル バージョンが 2.6.26 またはそれ以降の場合、**configure** スクリプトは **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。それ以前の Linux のバージョンでは、**--enable-kbuild** フラグで **configure** スクリプトを実行することで **kbuild** を強制的に使用することができます。

   ヒント:
   --help オプションを使用して、**configure** スクリプトのオプションのすべてのリストを表示します:
   ./configure --help

2. WinDriver のドライバ モジュールをビルドします:
   $ make
   これで、新規にコンパイルしたドライバ モジュール **windrvr1221.o/.ko** を含む、**LINUX.<kernel version>.<CPU>** ディレクトリを作成します。

3. **windrvr1221.o/.ko** ドライバ モジュールをインストール:

   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。

   # make install
   このインストールは、**setup_inst_dir** スクリプトを使用し (ターゲットのロード可能なカーネル モジュールのディレクトリへドライバ モジュールをコピー)、**wdred** スクリプトを使用してドライバ モジュールをロードします。

4. User および Group ID を変更してデバイス ファイル **/dev/windrvr1221** への **read / write** 権限を設定します (この作業は、ユーザーに対象のハードウェアへのアクセス権限の設定に依存します)。セキュリティ上の理由から、デフォルトでは、root ユーザーのみの権限でデバイス ファイルを作成します。**/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、以下の行を追加して **read / write** 権限を指定します:
   windrvr1221:root:root:0666

ヒント: **wdreg** スクリプトを使用して、ターゲットでシステムを起動するごとに動的に WinDriver のドライバ モジュールをロードします。この作業を自動化するには、**wdreg** をターゲットのマシンにコピーして、ターゲットの Linux のブート ファイル (例えば、**/etc/rc.local**) に次の行を追加します:
<wdreg へのパス> windrvr1221

15.4.3 ターゲットで Kernel PlugIn ドライバ モジュールをビルドおよびインストール

Kernel PlugIn ドライバを作成した場合、**configure.kp** スクリプトと関連するビルドおよびインストール ファイルを含む配布パッケージのサブディレクトリから (通常 **redist** サブディレクトリ)、以下のステップを実行してターゲットでこのドライバ (**kp_xxx_module.o/.ko**) のビルドとインストールを実行します:

1. Kernel Plugin の makefile – **makefile.kp** を生成します:
   $ ./configure.kp

注意:
configure スクリプトは起動しているカーネルをベースに makefile を作成します。**--with-kernel-source=<path>** でスクリプトを実行することで (**<path>** はカーネルのソース ディレクトリへのフルパスです。たとえば、**/usr/src/linux**)、他のインストール済みのカーネルのソースを選択することができます。
Linux のカーネル バージョンが 2.6.26 またはそれ以降の場合、**configure** スクリプトは **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。それ以前の Linux のバージョンでは、**--enable-kbuild** フラグで **configure** スクリプトを実行することで **kbuild** を強制的に使用することができます。

ヒント:
--help オプションを使用して、**configure** スクリプトのオプションのすべてのリストを表示します:
./configure.kp --help

2. Kernrl PlugIn のドライバ モジュールをビルドします:
   $ make -f makefile.kp
   これで、新規にコンパイルしたドライバ モジュール **kp_xxx_module.o/.ko** を含む、**LINUX.<kernel version>.<CPU>.KP** ディレクトリを作成します。

3. Kernel PlugIn モジュールをインストール:

   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。

   # make install -f makefile.kp

ヒント: ターゲットでシステムを起動するごとに自動的に Kernel PlugIn ドライバをロードするには、ターゲットの Linux のブート ファイル (例えば、**/etc/rc.local**) に、WinDriver のドライバ モジュールをロードするコマンドの後に、次の行を追加します (**<kp_xxx_module.o/.ko へのパス**> を対象の Kernel PlugIn のドライバ モジュールへのパス (対象の **LINUX.<kernel version>.<CPU>.KP** 配布ディレクトリ以下にあります) に置き換えてください):
/sbin/insmod <kp_xxx_moduel.o/.kp へのパス>

15.4.4 ユーザーモードのハードウェア コントロール アプリケーションまたは共有オブジェクトをインストール

ユーザーモードのハードウェア コントロール アプリケーションまたは共有オブジェクトを **libwdapi1221.so** を使用する場合、配布パッケージから **libwdapi1221.so** をターゲットのライブラリディレクトリへコピーします:

- **/usr/lib** - 32-bit アプリケーション / 共有オブジェクトを 32bit または 64-bit ターゲットへ配布する場合
- **/usr/lib64** - 64-bit アプリケーション / 共有オブジェクトを 64-bit ターゲットへ配布する場合

アプリケーション / 共有オブジェクトのソースコードを配布する場合、同様にソースコードをターゲットへコピーします。

注意: **libwdapi1221.so** 共有オブジェクトのソースコード、またはコード内の WinDriver のライセンスコードを配布することは禁じられているのでご注意ください。

第 16 章
ドライバのインストール - 高度な問題

16.1 Windows INF ファイル

デバイス情報ファイル (INF) は、Windows 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 windrvr1221.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。

16.1.1 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。

16.1.2 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 14.2.1 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。

注 意 : **ERROR_FILE_NOT_FOUND** エラーでインストールに失敗する場合、**HKEY_LOCAL_MACHINE¥SOFTWARE¥Microsoft¥Windows¥CurrentVersion** に **RunOnce** キーが存在するが Windows のレジストリを確認してください。INF ファイルを使用して正しくドライバをインストールするには、Windows Plug-and-Play にはこのレジストリ キーが必要です。**RunOnce** キーがない場合、作成して、再度 INF ファイルをインストールしてください。

16.1.3 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. 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 14.2.1 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。

注意： **ERROR_FILE_NOT_FOUND** エラーでインストールに失敗する場合、**HKEY_LOCAL_MACHINE¥SOFTWARE¥Microsoft¥Windows¥CurrentVersion** に **RunOnce** キーが存在するが Windows のレジストリを確認してください。INF ファイルを使用して正しくドライバをインストールするには、Windows Plug-and-Play にはこのレジストリキーが必要です。**RunOnce** キーがない場合、作成して、再度 INF ファイルをインストールしてください。

16.2 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。WinDriver のカーネル ドライバ モジュール (**windrvr1221.sys/.dll/.o/.ko** - OS により異なる) 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、デフォルトのカーネル モジュール (**windrvr1221.sys/.o/.ko**) ではなく名前を変更したドライバを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ます。

注意: Debug Monitor を使用して名前を変更したドライバのデバッグ メッセージをログする方法に関しては、セクション を参照してください。

名前を変更した WinDriver カーネル ドライバをオリジナルのカーネル モジュール (**windrvr1221.sys/.o/.ko**) 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 に同時にインストールすることもできます。

ヒント: ドライバをインストールするターゲットのマシンで他のドライバとの競合を回避するために、ドライバには固有の名前を付けてください。

16.2.1 Windows ドライバの名前変更

DriverWizard で WinDriver Windows カーネル ドライバ (**windrvr1221.sys**) の名前変更の作業の多くを自動化できます。

注意:

- ドライバの名前を変更する場合、開発プラットフォームの CPU アーキテクチャ (32-bit / 64-bit) と WinDriver のインストーラをターゲット プラットフォームと一致させてください。
- 署名済みの **windrvr1221.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定に関する情報は、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 16.3.2 を参照してください。

ヒント: このセクションの **xxx** への参照は、DriverWizard で生成したドライバ プロジェクトの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:

1. DriverWizard ユーティリティを使用して、Windows ハードウェア用のドライバ コードを生成します。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。**windrvr1221.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、オリジナル

の **windrvr1221.sys** ドライバのプロパティと同じです。以下に説明のとおり、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 **windrvr1221.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。

- **wdapi1221.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。

- **xxx_install.bat** - **xxx_driver.inf** と **xxx_device.inf** をインストールするために **wdreg** コマンドを実行するインストール スクリプト。このスクリプトは、名前を変更した **xxx_driver.sys** ドライバのインストールと、このドライバと対象のデバイスの登録を簡素化するためにデザインされています。

- **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:

  - 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。
  - 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。
  - 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。パラメータなしで **ddk_make.bat** ユーティリティを実行すると、このユーティリティで利用可能なオプションを表示します。
    > - 選択したビルド OS と WinDriver のインストーラの CPU アーキテクチャを一致させる必要があります。たとえば、WinDriver Windows 32-bit 版のインストーラを使用する場合、64-bit **win7_64** OS フラグを選択することはできません。

  xxx.sys ドライバをリビルドした後に、生成される **xxx_installation¥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、ユーザーモード アプリケーションが新しいドライバ名で 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が、デフォルトのドライバ名 (**windrvr1221**) を使用しているため、コード内で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るので、ご注意ください。

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 (たとえば、**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 15.2 の手順のとおり、オリジナルの WinDriver のインストール 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。

16.2.2 Linux ドライバの名前変更

DriverWizard で WinDriver Linux カーネル ドライバ (**windrvr1221.o/.ko**) の名前変更の作業の多くを自動化できます。

ヒント: 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、**windrvr1221.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: オリジナルの WinDriver の library および include ディレクトリのコピー。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以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、ユーバーモードのアプリケーションが新しいドライバ名で 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が、デフォルトのドライバ名 (**windrvr1221**) を使用しているため、コード内で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るので、ご注意ください。

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 (たとえば、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。Kernel PlugIn ドライバを作成した場合、Kernel PlugIn ドライバの **configure** スクリプトの以下の行のコメントを外して、このフラグを追加する必要があります:
   # ADDITIONAL_FLAGS="-DWD_DRIVER_NAME_CHANGE"

4. セクション 15.4 の手順のとおり、オリジナルの 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 15.4.1.1 の手順のとおり、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。

16.3 Windows のデジタル ドライバの署名と認定

16.3.1 概要

対象のドライバを配布する前に、Microsoft 社の Authenticode メカニズムを使用してドライバをデジタル署名するか、もしくは Microsoft 社の Windows 認定プログラム (HLK / HCK / WLP) へドライバを申請して認定を取得できます。

Windows 7 などの一部のWindows OS では、ドライバがデジタル署名または認定を取得されていない場合でも、警告のポップアップが表示されることはありますが、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、ドライバのデジタル署名や認定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 7 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る。

Windows 8 およびそれ以降の 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。

> **注意:** ドライバの開発中、Windows OS の設定で、署名していないドライバーを一時的にインストールすることができます。

デジタル ドライバの署名と認定に関する詳細情報は、MSDN (Microsoft Development Network) ライブラリの以下のドキュメントを参照してください:

- Driver Signing Requirements for Windows
- Introduction to Code Signing
- Digital Signatures for Kernel Modules on Windows
  このホワイト ペーパーには、カーネルモードのコード署名、テスト署名、および開発期間中の署名の無効に関する情報が含まれます。
- https://msdn.microsoft.com/en-us/windows-drivers/develop/signing_a_driver

これらのドキュメントには、古い用語を使用している場合もあります。たとえば、**WLP** (Windows Logo Program)、**WHQL** (Windows Hardware Quality Labs)、**Windows Certification Program** や **HCK** (Windows Hardware Certification Kit) への参照は、現在では **HLK** (Windows Hardware Lab Kit) へ変更されています。また **Winqual** (Windows Quality Online Services) への参照も **Hardware Dashboard** (Windows Dev Center Hardware Dashboard Service) へ変更されています。

16.3.1.1 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、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¥windrvr1221.sys ドライバは Authenticode デジタル署名を持っています。

16.3.1.2 Windows 認定プログラム

Microsoft 社の Windows 認定プログラム (WLP 旧 Windows ロゴ プログラム) では、ハードウェア モジュールおよびソフトウェア モジュール (ドライバを含む) に対し Microsoft の資格認定を取得するためのの手順について説明しています。テストに Pass したハードウェアまたはソフトウェアは、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 ドライバのデジタル署名と認定を取得するには、**Hardware Dashboard** (Windows Dev Center Hardware Dashboard) を使用して、ドライバと関連するハードウェアを含む **HLK** (Windows Hardware Lab Kit) パッケージをテストのために Windows 認定プログラムに申請します。

Jungo Connectivity 社のプロフェッショナル サービス チームが Jungo Connectivity ベースのドライバに対して Windows ドライバの pre-certification サービスを提供しています。Jungo Connectivity 社のエンジニアが Jungo Connectivity 社の専用の Windows 認定プログラムで必要なすべてのテストを行うので、テストに掛かる手間と工数を大幅に軽減できます。Microsoft へ申請できるように、テスト結果を HLK / HCK の申請に必要なフォーマットにパッケージ化して提供いたします。
詳細は、http://www.xlsoft.com/jp/products/windriver/whql_service.html を参照してください。

Microfoft 社の Windows 認定プログラムと認定プロセスに関する詳細は、次の Microsoft 社の Web サイトを参照してください:

- MSDN Windows 認定プログラム ページ:
  http://msdn.microsoft.com/library/windows/hardware/gg463010.aspx
- MSDN Windows Dev Center を含むドキュメント、Hardware Dashboard Service ページ:
  http://msdn.microsoft.com/library/windows/hardware/gg463091

16.3.2 WinDriver ベースのドライバのドライバ署名と認定

上記の説明のとおり、**WinDriver¥redist¥windrvr1221.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 (**windrvr1221.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、スタンドアロン ドライバとして Microsoft 社の Windows 認定プログラムに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 ドライバを開発した場合、以下で説明するとおり、Microsoft 社の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。

ドライバの認定と署名の手順は (Authenticode または Windows 認定プログラムのいずれかで)、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 **windrvr1221.sys** ドライバは一致するカタログ ファイル (**WinDriver¥redist¥windrvr1221.cat**) と提供されます。このファイルは **windrvr1221.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、**windrvr1221.sys** ドライバ、および (または) 関連する **windrvr1221.inf** ファイルの名前変更する場合、**windrvr1221.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、**windrvr1221.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、**windrvr1221.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 Connectivity のアプリオリでこの INF ファイルを署名できません。

windrvr1221.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、**windrvr1221.inf** ファイル (または名前を変更したファイル) から **windrvr1221.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 Windows 認定プログラムに申請するか、Authenticode でドライバを署名する。**WinDriver¥redist¥windrvr1221.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 Microsoft 社の Windows 認定プログラムの認定要件と互換性があります。

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

- Microsoft 社の Windows 認定プログラムのドキュメントで説明してるとおり、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 **windrvr1221.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。(**windrvr1221.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 **windrvr1221.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。)
- Microsoft 社の Windows 認定プログラムに申請するか、Authenticode でドライバを署名する。Windows 認定プログラムに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。

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定の取得に成功しています。

16.3.2.1 HCK テストに関する注意点

Microsoft 社の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と認定を行う前に Windows の **HCK** (Hardware Certification Kit) 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。HCK 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。詳細については、MSDN Windows HCK (Hardware Certification Kit) のページを参照ください - http://msdn.microsoft.com/library/windows/hardware/hh833788

16.4 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 (**windrvr1221.inf** ファイルと WinDriver カーネル ドライバ (**windrvr1221.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 (**wdapi1221.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、以下のようになります:
     ```
     "my_dev_pci"=Install, PCIVEN_1111&DEV_2222
     ```

   または

   - DriverWizard 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 (行をコメントアウトするには、行の先頭にコロン (::) を二つ追加します):
   ```
   wdreg -inf dev.inf install
   ```

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

   ① DBMgr (Windows Embedded Component Database Manager) を開きます。

   ② **Import** をクリックします。

   ③ 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:**

① Target Designer で対象のプロジェクトを開きます。

② 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。

③ 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、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 プラットフォームでドライバをインストールする必要があります。

第 17 章
PCI Express

17.1 PCI Express の概要

PCI Express (PCIe) バス アーキテクチャ (以前の 3GIO または 3rd Generation I/O) は、将来に向けた PC I/O 規格を確立するために、インテルとその他のリーディング カンパニー (IBM、Dell、Compaq、HP および Microsoft) の協力のもとに開発されました。

PCI-Express は、PCI 2.2 バスよりもスケーラビリティが高い広帯域を提供します。

標準 PCI 2.2 バスは、すべてのデータが単一の並列データ バスを介して設定速度で送信されるように設計されています。標準 PCI 2.2 バスでは、接続されているすべてのデバイス間で帯域幅が共有され、デバイス間に優先度はありません。最大帯域幅は 132MB/s で、接続されているすべてのデバイス間で共有されます。

PCI Express は、ポイント・ツー・ポイントのシリアル伝送を行う個別のクロック信号を持つレーンから構成されています。各レーンは、データの双方向伝送を同時に行うことができる 2 本のデータ レーンから構成されています。バス スロットは、バス上のデータ フローを制御するスイッチに接続されていて、PCI Express デバイスと PCI Express スイッチ間の接続をリンクと呼びます。各リンクは、1 本または複数のレーンから構成されていて、1 本のレーンで構成されるリンクは x1 リンク、2 本のレーンで構成されるリンクは x2 リンクと呼びます。PCI Express は x1、x2、x4、x8、x12、x16、および x32 のリンク幅 (レーン) をサポートしています。PCI Express アーキテクチャでは、レーンあたり約500MB/s の最大帯域幅が可能です。このため、潜在的な最大帯域幅は、x1 では 500MB/s 、x2 では 1,000MB/s、x4 では 2,000MB/s 、x8 では 4,000MB/s 、x12 では 6,000MB/s、x16 では 8,000MB/s となります。これは、標準の 32 ビット PCI バスの最大帯域幅である 132MB/s と比べて大きな改善であるといえます。

広帯域幅をサポートする PCI Express は、増加を続けるハード ドライブ コントローラ、ビデオ ストリーミング デバイス、ネットワーク カードなど、広帯域を必要とするデバイスにとって理想的です。

PCI Express バスのデータ フローを制御するスイッチを使用することによって、複数のデバイス間でバスを共有する代わりに、各デバイスがバスに直接アクセスできるようになり、共有 PCI バスを改善できます。これにより、各デバイスは、単一の共有バスの最大帯域幅を奪い合うことなく、帯域幅をフルに使用することができます。

これらに加え、各デバイスが PCI Express バスにアクセスするためのレーンを考慮すると、PCI Express では、以前の PCI と比べより広域な帯域幅の制御が可能です。また、PCI Express では、デバイス同士が直接通信できます (ピアツーピア通信)。

さらに、PCI Express バストポロジーでは、共有バストポロジーとは異なり、転送およびリソース管理が中央化されています。このため、PCI Express は QoS (品質保証) をサポートしています。PCI Express スイッチはパケットに優先度を与え、リアルタイム ストリーミング パケット (例: ビデオ ストリームやオーディオ ストリームなど) は、タイム クリティカルでないパケットに対し優先権を得ることができます。

PCI スロット、AGP スロット、または PCI-X などその他の新しい I/O バス ソリューションと比べ、低価格で製造できることも PCI Express の利点です。

PCI Express は、既存の PCI バス、PCI デバイス (これらのバスのアーキテクチャが異なる場合も含め) とハードウェアおよびソフトウェアの完全な互換性を維持するよう設計されています。

PCI 2.2 バスとの下位互換性の一部として、以前の PCI 2.2 デバイスを PCI Express-to-PCI ブリッジを介して PCI Express システムに挿入することができます。PCI Express-to-PCI ブリッジは、マザーボードまたは外部カードのいずれかにあり、PCI Express パケットを標準 PCI 2.2 バス信号に変換します。

17.2 WinDriver PCI Express

WinDriver は、PCI Express ボードにおいて標準 PCI 機能との下位互換性を完全にサポートしています。サポートには、豊富な API セット、ハードウェアのデバッグおよびドライバ コード生成用のコードサンプルとグラフィカルな DriverWizard が含まれます。これらのサポートは、PCI Express デバイス (以前の PCI デバイスとの下位互換性を持つよう設計されている) でも利用できます。

また、WinDriver の PCI API を使用し、PCI Express-to-PCI ブリッジおよび スイッチ (例: PLX 8111/8114 ブリッジ、PLX 8532 スイッチ) によって PC に接続された PCI デバイスと簡単に通信することもできます。

さらに、WinDriver (Windows および Linux) では、PCI Express の拡張設定空間へのアクセスを簡単にする API セットを提供してます (詳細は、`WDC_PciReadCfgXXX()` 関数、`WDC_PciWriteCfgXXX()` 関数、低レベル `WD_PciConfigDump()` 関数の説明を参照)。

Linux、Windows Vista およびそれ以降では、WinDirver の割込み処理 API は MSI (Message-Signaled Interrups) と MSI-X (Extended Message-Signaled Interrups) 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、Xilinx の BMD (Bus Mastering DMA Validation Design) デザイン、または Altera Qsys デザインを搭載した PCI Express カードに対して、拡張サポートを提供します。**`WinDriver¥xilinx¥bmd_design`** ディレクトリと **`WinDriver¥altera¥qsys_design`** ディレクトリには、それぞれ DMA と MSI 処理を含む各デバイス タイプと通信を行うライブラリ API とユーザーモードの診断アプリケーションのサンプルが含まれています。Xilinx BMD ディレクトリには、Kernel PlugIn ドライバ コードのサンプルも含まれています。さらに、DriverWizard を使用して、各カードに対してカスタマイズしたコードを生成することができます。

WinDriver

ユーザーズ ガイド

2016年 9月 15日

発行　**エクセルソフト株式会社**
〒108-0073 東京都港区三田3-9-9 森伝ビル6F
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 Connectivity 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