デバイスドライバの記述

Part No: E38872
2013 年 1 月

130221@25097

目次

図目次

表目次

例目次

はじめに

『デバイスドライバの記述』では、Oracle Solaris オペレーティングシステム (Oracle Solaris OS) のための文字指向のデバイス、ブロック指向のデバイス、ネットワークデバイス、SCSI ターゲットと HBA デバイス、および USB デバイスのドライバの開発に関する情報を提供します。このドキュメントでは、Solaris OS DDI/DKI (デバイスドライバインタフェース、ドライバカーネルインタフェース) に準拠するすべてのアーキテクチャーのためのマルチスレッド化された再入可能なデバイスドライバを開発する方法について説明します。エンディアンの種類やデータの順序付けなどのプラットフォーム固有の問題を気にすることなくドライバを記述できる、共通ドライバプログラミングアプローチについて説明します。

その他のトピックとして、Oracle Solaris ドライバの強化、電源管理、ドライバの自動構成、プログラム式入出力、ダイレクトメモリーアクセス (DMA)、デバイスコンテキスト管理、ドライバのコンパイル、インストール、およびテスト、ドライバのデバッグ、Oracle Solaris ドライバの 64 ビット環境への移植などが含まれています。

注 – この Oracle Solaris のリリースでは、SPARC および x86 系列のプロセッサアーキテクチャーを使用するシステムをサポートしています。サポートされるシステムは、Oracle Solaris OS: Hardware Compatibility Lists に記載されています。本書では、プラットフォームにより実装が異なる場合は、それを特記します。

本書の x86 に関連する用語については、次を参照してください。

- x86 は、64 ビットおよび 32 ビットの x86 互換製品系列を指します。
- x64 は特に 64 ビット x86 互換 CPU を指します。
- 「32 ビット x86」は、x86 をベースとするシステムに関する 32 ビット特有の情報を指します。

サポートされるシステムについては、Oracle Solaris OS: Hardware Compatibility Lists を参照してください。

対象読者

このドキュメントは UNIX デバイスドライバに精通している UNIX プログラマ向けに書かれています。概要情報も記載されていますが、本書はデバイスドライバに関する一般的な学習教材ではありません。

注 – Oracle Solaris オペレーティングシステム (Oracle Solaris OS) は SPARC アーキテクチャーと x86 アーキテクチャーの両方で動作します。また、Oracle Solaris OS は 64 ビットおよび 32 ビットのアドレス空間で動作します。このドキュメントの情報は、特に明記されていないかぎり、すべてのプラットフォームとアドレス空間にあてはまります。

内容の紹介

このドキュメントは次の章で構成されています。

- 第 1 章「Oracle Solaris デバイスドライバの概要」では、Oracle Solaris プラットフォームのデバイスドライバおよび関連するエントリポイントの概要を説明します。デバイスドライバタイプごとのエントリポイントを表形式で示します。
- 第 2 章「Oracle Solaris カーネルとデバイスツリー」では、Solaris カーネルの概要を説明するほか、デバイスツリーでデバイスがどのようにノードとして表現されるかについても説明します。
- 第 3 章「マルチスレッド」では、Solaris のマルチスレッドカーネルについて、デバイスドライバ開発者に関係の深い側面について説明します。
- 第 4 章「プロパティー」では、デバイスのプロパティーを使用するためのインタフェースセットについて説明します。
- 第 5 章「イベントの管理とタスクのキュー」では、デバイスドライバでイベントをログに記録する方法と、タスクキューを使用してタスクをあとで実行する方法について説明します。
- 第 6 章「ドライバの自動構成」では、ドライバが自動構成のために提供する必要のあるサポートについて説明します。
- 第 7 章「デバイスアクセス: プログラム式入出力」では、ドライバがデバイスメモリーに対して読み書きを行うためのインタフェースや手法について説明します。
- 第 8 章「割り込みハンドラ」では、割り込みを処理するメカニズムについて説明します。これらのメカニズムには、割り込みの割り当て、登録、処理、および削除が含まれます。
- 第 9 章「ダイレクトメモリーアクセス (DMA)」では、ダイレクトメモリーアクセス (DMA) と DMA のインタフェースについて説明します。
- 第 10 章「デバイスメモリーおよびカーネルメモリーのマッピング」では、デバイスメモリーとカーネルメモリーを管理するインタフェースについて説明します。

- 第 11 章「デバイスコンテキスト管理」では、デバイスドライバがデバイスへのユーザーアクセスを管理できるようにするインタフェースセットについて説明します。
- 第 12 章「電源管理」ではPower Management のインタフェースと、電力消費を管理するためのフレームワークについて説明します。
- 第 13 章「Oracle Solaris ドライバの強化」では、入出力デバイスドライバに障害管理機能を組み込む方法、防御力を高めるプログラミング手法を組み込む方法、およびドライバ強化テストハーネスを使用する方法について説明します。
- 第 14 章「階層化ドライバインタフェース (LDI)」では、カーネルモジュールがシステム内のほかのデバイスにアクセスできるようにする LDI について説明します。
- 第 15 章「文字デバイスのドライバ」では、文字指向デバイスのドライバについて説明します。
- 第 16 章「ブロックデバイスのドライバ」では、ブロック指向デバイスのドライバについて説明します。
- 第 17 章「SCSI ターゲットドライバ」では、Sun Common SCSI Architecture (SCSA) と、SCSI ターゲットドライバの要件について概説します。
- 第 18 章「SCSI ホストバスアダプタドライバ」では、SCSA を SCSI ホストバスアダプタ (HBA) ドライバに適用する方法について説明します。
- 第 19 章「ネットワークデバイスのドライバ」では、汎用 LAN ドライバ (GLD) について説明します。GLDv3 フレームワークは、MAC プラグインと、MAC ドライバサービスのルーチンおよび構造体に対する、関数呼び出しベースのインタフェースです。
- 第 20 章「USB ドライバ」では、USBA 2.0 フレームワーク を使用してクライアント USB デバイスドライバを記述する方法について説明します。
- 第 22 章「ドライバのコンパイル、ロード、パッケージ化、およびテスト」では、ドライバのコンパイル、リンク、およびインストールに関する情報を提供します。
- 第 23 章「デバイスドライバのデバッグ、テスト、およびチューニング」では、ドライバのデバッグ、テスト、およびチューニングを行うための手法について説明します。
- 第 24 章「推奨されるコーティング方法」では、ドライバを記述するための推奨のコーディング手法について説明します。
- 付録 A「ハードウェアの概要」では、デバイスドライバのマルチプラットフォームハードウェアの問題について説明します。
- 付録 B「Solaris DDI/DKI サービスのサマリー」では、デバイスドライバ用のカーネル関数の表を提供します。非推奨となった関数も明記します。
- 付録 C「64 ビットデバイスドライバの準備」では、64 ビット環境で動作するようにデバイスドライバを更新するためのガイドラインを提供します。

- 付録 D「コンソールフレームバッファードライバ」では、フレームバッファードライバに必要なインタフェースを追加することで、そのドライバを Oracle Solaris カーネル端末エミュレータと対話できるようにする方法について説明します。

関連ドキュメントと論文

デバイスドライバインタフェースの詳細な参照情報については、Section 9 のマニュアルページを参照してください。Section 9E の `Intro(9E)` では、DDI/DKI (デバイスドライバインタフェース/ドライバカーネルインタフェース) のドライバエントリポイントについて説明します。Section 9F の `Intro(9F)` では、DDI/DKI のカーネル関数について説明します。Sections 9P および 9S の `Intro(9S)` では、DDI/DKI のプロパティとデータ構造について説明します。

ハードウェアとその他のドライバ関連の問題については、次のドキュメントを参照してください。

- 『Device Driver Tutorial 』
- 『Solaris モジューラデバッガ』
- 『Oracle Solaris 11.1 Dynamic Tracing Guide 』
- 『アプリケーションパッケージ開発者ガイド』
- 『マルチスレッドのプログラミング』
- 『Solaris 64 ビット 開発ガイド』
- 『STREAMS Programming Guide 』
- 『Open Boot PROM Toolkit User's Guide』

また、次のドキュメントが役立つ場合があります。

- SPARC インターナショナル、『The SPARC Architecture Manual』 バージョン 9、Prentice Hall、1993; ISBN 978-0130992277

Oracle サポートへのアクセス

Oracle のお客様は、My Oracle Support を通じて電子的なサポートを利用することができます。詳細は、`http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` を参照してください。聴覚に障害をお持ちの場合は、`http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` を参照してください。

表記上の規則

次の表では、このドキュメントで使用される表記上の規則について説明します。

表 P–1 表記上の規則

字体	説明	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 machine_name% you have mail.
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	machine_name% **su** Password:
aabbcc123	プレースホルダ: 実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、 rm *filename* と入力します。
AaBbCc123	書名、新しい単語、および強調する単語を示します。	『ユーザーズガイド』の第 6 章を参照してください。 キャッシュは、ローカルに格納されるコピーです。 ファイルを保存しないでください。 注: いくつかの強調された項目は、オンラインでは太字で表示されます。

コマンド例のシェルプロンプト

Oracle Solaris OS に含まれるシェルで使用する、UNIX のデフォルトのシステムプロンプトとスーパーユーザープロンプトを次に示します。コマンド例に示されるデフォルトのシステムプロンプトは、Oracle Solaris のリリースによって異なります。

表 P–2 シェルプロンプト

シェル	プロンプト
Bash シェル、Korn シェル、および Bourne シェル	$
Bash シェル、Korn シェル、および Bourne シェルのスーパーユーザー	#
C シェル	machine_name%

表 P–2 シェルプロンプト (続き)

シェル	プロンプト
C シェルのスーパーユーザー	machine_name#

パ ー ト I

Oracle Solaris プラットフォーム用デバイスドライバの設計

このマニュアルの第 1 部では、Oracle Solaris プラットフォームでデバイスドライバを開発するための一般的な情報を提供します。ここに含まれる章は次のとおりです。

- 第 1 章「Oracle Solaris デバイスドライバの概要」では、Oracle Solaris プラットフォームのデバイスドライバおよび関連するエントリポイントの概要を説明します。デバイスドライバタイプごとのエントリポイントを表形式で示します。
- 第 2 章「Oracle Solaris カーネルとデバイスツリー」では、Oracle Solaris カーネルの概要を説明するほか、デバイスツリーでデバイスがどのようにノードとして表現されるかについても説明します。
- 第 3 章「マルチスレッド」では、Oracle Solaris のマルチスレッドカーネルについて、デバイスドライバ開発者に関係の深い側面について説明します。
- 第 4 章「プロパティー」では、デバイスのプロパティーを使用するためのインタフェースセットについて説明します。
- 第 5 章「イベントの管理とタスクのキュー」では、デバイスドライバでイベントをログに記録する方法と、タスクキューを使用してタスクをあとで実行する方法について説明します。

- 第 6 章「ドライバの自動構成」では、ドライバが自動構成のために提供する必要のあるサポートについて説明します。
- 第 7 章「デバイスアクセス: プログラム式入出力」では、ドライバがデバイスメモリーに対して読み書きを行うためのインタフェースや手法について説明します。
- 第 8 章「割り込みハンドラ」では、割り込みを処理するメカニズムについて説明します。これらのメカニズムには、割り込みの割り当て、登録、処理、および削除が含まれます。
- 第 9 章「ダイレクトメモリーアクセス (DMA)」では、ダイレクトメモリーアクセス (DMA) と DMA のインタフェースについて説明します。
- 第 10 章「デバイスメモリーおよびカーネルメモリーのマッピング」では、デバイスメモリーとカーネルメモリーを管理するインタフェースについて説明します。
- 第 11 章「デバイスコンテキスト管理」では、デバイスドライバがデバイスへのユーザーアクセスを管理できるようにするインタフェースセットについて説明します。
- 第 12 章「電源管理」では、消費電力を管理するためのフレームワークである Power Management 機能のインタフェースについて説明します。
- 第 13 章「Oracle Solaris ドライバの強化」では、入出力デバイスドライバに障害管理機能を組み込む方法、防御力を高めるプログラミング手法を組み込む方法、およびドライバ強化テストハーネスを使用する方法について説明します。
- 第 14 章「階層化ドライバインタフェース (LDI)」では、カーネルモジュールがシステム内のほかのデバイスにアクセスできるようにする LDI について説明します。

◆ ◆ ◆ 第 1 章

1 Oracle Solaris デバイスドライバの概要

この章では、Oracle Solaris デバイスドライバの概要を説明します。この章では、次の内容について説明します。

デバイスドライバの基本

このセクションでは、Oracle Solaris プラットフォームでのデバイスドライバとそのエントリポイントについて説明します。

デバイスドライバとは

デバイスドライバとは、ハードウェアデバイスの低レベルの入出力操作を管理するカーネルモジュールのことです。デバイスドライバは、カーネルがデバイスとのインタフェースを取るために呼び出すことのできる標準的なインタフェースで書かれています。また、デバイスドライバをソフトウェア専用にして、RAM ディスク、バス、擬似端末など、ソフトウェアでのみ存在するデバイスをエミュレートすることもできます。

デバイスドライバには、デバイスとの通信に必要なデバイス固有のすべてのコードが含まれています。このコードには、システムの残りの部分へのインタフェースの標準セットが含まれています。このインタフェースは、システムコールインタフェースがアプリケーションプログラムをプラットフォーム固有の詳細から保護するように、カーネルをデバイス固有の詳細から遮蔽します。アプリケーションプログラムやカーネルの残りの部分では、デバイスに対応するためのデバイス固有のコードがたとえあったとしても、ほとんど必要ありません。このような点で、デバイスドライバはシステムの移植性を高め、システムの保守を容易にします。

Oracle Solaris オペレーティングシステム (Oracle Solaris OS) が初期化されると、デバイスは自らを識別し、デバイスの階層であるデバイスツリーに編成されます。実際には、デバイスツリーはカーネルのハードウェアモデルです。ツリー内では、個々のデバイスドライバは子を持たないノードで表されます。このようなノードは、リーフドライバと呼ばれます。ほかのドライバにサービスを提供するドライバはバスネクサスドライバと呼ばれ、子を持つノードで表されます。ブートプロセスの一環として、物理デバイスがツリー内のドライバにマップされるため、必要なときにそれらのドライバを特定できるようになります。Oracle Solaris OS によるデバイスへの対応の仕方についての詳細は、第 2 章「Oracle Solaris カーネルとデバイスツリー」を参照してください。

デバイスドライバは入出力の処理方法によって分類されます。デバイスドライバは 3 つの大まかなカテゴリに分類されます。

- ブロックデバイスドライバ – 入出力データを非同期のチャンクとして処理することが適切な場合。ブロックドライバは通常、ディスクなどの物理的にアドレス可能な記憶メディアでデバイスを管理するために使用されます。
- 文字デバイスドライバ – 入出力を連続的なバイトの流れで実行するデバイス向け。

  注 – ファイルシステムに対して 2 つの異なるインタフェースを設定した場合は、1 つのドライバを同時にブロックと文字の両方のドライバとして使用できます。60 ページの「特殊ファイルとしてのデバイス」を参照してください。

  STREAMS モデル (下記を参照) を使用するドライバ、プログラム式入出力、ダイレクトメモリーアクセス、SCSI バス、USB、その他のネットワーク入出力は文字カテゴリに含まれます。
- **STREAMS** デバイスドライバ – カーネル内の文字入出力に対して `streamio(7I)` セットのルーチンを使用する文字ドライバのサブセット。

デバイスドライバのエントリポイントとは

エントリポイントとは、なんらかのドライバ機能を利用したり、デバイスを操作したりするために外部エンティティーから呼び出すことができるデバイスドライバ内の関数のことです。各デバイスドライバには、エントリポイントとして関数の標準セットが用意されています。すべてのドライバタイプ用のエントリポイントの一覧については、`Intro(9E)` のマニュアルページを参照してください。Oracle Solaris カーネルでは、次の一般的なタスク領域でエントリポイントを使用します。

- ドライバのロードとアンロード
- デバイスの自動構成 – 自動構成とは、デバイスドライバのコードと静的データをメモリーにロードして、ドライバがシステムに登録されるようにするプロセスです。

- ドライバへの入出力サービスの提供

各種デバイスのドライバには、デバイスが実行する処理の種類に応じてさまざまなエントリポイントのセットがあります。たとえば、メモリーマッピングされた文字処理型のデバイスは devmap(9E) エントリポイントをサポートしますが、ブロックドライバはこのエントリをサポートしません。

ドライバ関数に一意の名前を付けるには、ドライバの名前に基づく接頭辞を使用します。通常、この接頭辞はドライバの名前になります。たとえば、xx_open() はドライバ xx の open(9E) ルーチンを表しています。詳細は、589 ページの「一意の接頭辞を使用してカーネルシンボルの衝突を回避する」を参照してください。このドキュメントの以降の例では、xx をドライバ接頭辞として使用します。

デバイスドライバのエントリポイント

このセクションでは、次のカテゴリのエントリポイントの一覧を示します。

すべてのドライバに共通のエントリポイント

モジュールのロードに必要な関数や必須の自動構成エントリポイントなど、一部の処理はどのタイプのドライバでも実行できます。このセクションでは、すべてのドライバに共通したエントリポイントの種類について説明します。共通のエントリポイントの一覧は、43 ページの「共通のエントリポイントのサマリー」に、マニュアルページへのリンクとその他の関連情報とともに記載してあります。

デバイスアクセスのエントリポイント

文字デバイスとブロックデバイス用のドライバは、cb_ops(9S) 構造体をエクスポートします。この構造体では、ブロックデバイスアクセスと文字デバイスアクセス用のドライバエントリポイントを定義します。どちらのタイプのドライバも、open(9E) および close(9E) エントリポイントをサポートする必要があります。ブロックドライバは strategy(9E) をサポートする必要がありますが、文字ドライバはデバイスのタイプに適していれば、read(9E)、write(9E)、ioctl(9E)、mmap(9E)、または devmap(9E) エントリポイントのどのような組み合わせの実装も選択できます。さらに、文字ド

ライバは chpoll(9E) を通じてポーリングインタフェースをサポートすることもできます。非同期入出力は、ブロックドライバと、ブロックファイルシステムおよび文字ファイルシステムの両方を使用できるドライバで、aread(9E) および awrite(9E) を通じてサポートされます。

ロード可能なモジュールのエントリポイント

すべてのドライバは、ドライバモジュールのロード、アンロード、および関連情報の報告を行うためにロード可能なモジュールのエントリポイント _init(9E)、_fini(9E)、および _info(9E) を実装する必要があります。

ドライバは、_init(9E) でグローバルリソースの割り当てと初期化を行います。ドライバは、_fini(9E) でそのリソースを解放します。

注 – Oracle Solaris OS では、ロード可能なモジュールルーチンだけを、ドライバオブジェクトモジュールの外から表示できる必要があります。その他のルーチンには、ストレージクラス `static` を指定できます。

自動構成のエントリポイント

ドライバは、デバイスの自動構成のために attach(9E)、detach(9E)、および getinfo(9E) エントリポイントを実装する必要があります。また、SCSI ターゲットデバイスなど、デバイスがブート中に自らを識別できない場合は、オプション指定のエントリポイント probe(9E) を実装できます。これらのルーチンの詳細は、第 6 章「ドライバの自動構成」を参照してください。

カーネル統計情報のエントリポイント

Oracle Solaris プラットフォームには、カーネルレベルの統計情報 (*kstats* とも呼ばれる) を保持およびエクスポートするための豊富なインタフェースセットが用意されています。ドライバは、これらのインタフェースを自由に使用して、ドライバの内部状態を監視するために、ユーザーアプリケーションで使用できるドライバとデバイスの統計情報をエクスポートします。カーネル統計情報を処理するために 2 つのエントリポイントが提供されています。

- ks_snapshot(9E) は、特定の時間に kstats を取り込みます。
- ks_update(9E) は、kstats データを自由に更新するために使用できます。`ks_update()` は、デバイスがカーネルデータを追跡するように設定されているが、そのデータの抽出に時間がかかる場合に役立ちます。

詳細は、kstat_create(9F) と kstat(9S) のマニュアルページを参照してください。581 ページの「カーネル統計」も参照してください。

電源管理のエントリポイント

電源管理機能を備えたハードウェアデバイスのドライバは、任意指定のエントリポイント power(9E) をサポートできます。このエントリポイントの詳細は、第 12 章「電源管理」を参照してください。

共通のエントリポイントのサマリー

次の表に、すべてのタイプのドライバで使用できるエントリポイントの一覧を示します。

表 1–1 すべてのドライバタイプ用のエントリポイント

カテゴリ/エントリポイント	用途	説明
cb_ops エントリポイント		
open(9E)	必須	デバイスにアクセスできます。追加情報: ■ 309 ページの「open() エントリポイント (文字ドライバ)」 ■ 337 ページの「open() エントリポイント (ブロックドライバ)」
close(9E)	必須	デバイスへのアクセスを中止します。STREAMS ドライバ用の close() のバージョンには文字ドライバやブロックドライバとは異なるシグニチャーがあります。追加情報: ■ 310 ページの「close() エントリポイント (文字ドライバ)」 ■ 338 ページの「close() エントリポイント (ブロックドライバ)」
ロード可能なモジュールのエントリポイント		
_init(9E)	必須	ロード可能なモジュールを初期化します。追加情報: 101 ページの「ロード可能なドライバインタフェース」
_fini(9E)	必須	ロード可能なモジュールのアンロードの準備をします。すべてのドライバタイプに必須です。追加情報: 101 ページの「ロード可能なドライバインタフェース」
_info(9E)	必須	ロード可能なモジュールに関する情報を返します。追加情報: 101 ページの「ロード可能なドライバインタフェース」
自動構成のエントリポイント		
attach(9E)	必須	初期化中にデバイスをシステムに追加します。中断されていたシステムの再開にも使用されます。追加情報: 109 ページの「attach() エントリポイント」
detach(9E)	必須	デバイスをシステムから切り離します。一時的にデバイスを中断する場合にも使用されます。追加情報: 115 ページの「detach() エントリポイント」

表 1–1 すべてのドライバタイプ用のエントリポイント (続き)

カテゴリ/エントリポイント	用途	説明
getinfo(9E)	必須	デバイス番号とそれに対応するインスタンス間のマッピングなど、ドライバに固有のデバイス情報を取得します。追加情報: ■ 116 ページの「getinfo() エントリポイント」 ■ 367 ページの「getinfo() エントリポイント (SCSI ターゲットドライバ)」
probe(9E)	「説明」を参照	自己識別しないデバイスが存在するかどうかを判別します。自らを識別できないデバイスには必須です。追加情報: ■ 106 ページの「probe() エントリポイント」 ■ 362 ページの「probe() エントリポイント (SCSI ターゲットドライバ)」
カーネル統計情報のエントリポイント		
ks_snapshot(9E)	任意	kstat(9S) データのスナップショットを取得します。追加情報: 581 ページの「カーネル統計」
ks_update(9E)	任意	kstat(9S) データを動的に更新します。追加情報: 581 ページの「カーネル統計」
電源管理のエントリポイント		
power(9E)	必須	デバイスの電源レベルを設定します。使用しない場合は、NULL に設定します。追加情報: 223 ページの「power() エントリポイント」
その他のエントリポイント		
prop_op(9E)	「説明」を参照	ドライバのプロパティー情報を報告します。ddi_prop_op(9F) が代用されないかぎり必須です。追加情報: ■ 80 ページの「プロパティーの作成と更新」 ■ 82 ページの「prop_op() エントリポイント」
dump(9E)	「説明」を参照	システム障害の発生中にメモリーの内容をデバイスにダンプします。パニックの発生時にダンプデバイスとして使用するデバイスには必須です。追加情報: ■ 350 ページの「dump() エントリポイント (ブロックドライバ)」 ■ 376 ページの「ダンプの処理」
identify(9E)	廃止	このエントリポイントは使用しないでください。dev_ops 構造体で、このエントリポイントに nulldev(9F) を割り当ててください。

ブロックデバイスドライバ用のエントリポイント

ファイルシステムをサポートするデバイスは、ブロックデバイスと呼ばれます。このデバイス用に作成されたドライバは、ブロックデバイスドライバと呼ばれます。ブロックデバイスドライバは、buf(9S) 構造体の形でファイルシステム要求を受

け取り、指定されたブロックを転送するために入出力操作をディスクに発行します。ファイルシステムへのメインインタフェースは、strategy(9E) ルーチンです。詳細は、第 16 章「ブロックデバイスのドライバ」を参照してください。

ブロックデバイスドライバは、文字ドライバインタフェースを提供することもできます。これにより、ユーティリティープログラムはファイルシステムをバイパスして、デバイスに直接アクセスできます。このデバイスアクセスは一般に、ブロックデバイスへの *raw* インタフェースと呼ばれます。

次の表に、ブロックデバイスドライバで使用できる追加のエントリポイントの一覧を示します。41 ページの「すべてのドライバに共通のエントリポイント」も参照してください。

表 1–2 ブロックドライバ用の追加のエントリポイント

エントリポイント	用途	説明
aread(9E)	任意	非同期読み取りを実行します。aread() エントリポイントをサポートしないドライバは、nodev(9F) エラー戻り関数を使用します。追加情報: ■ 313 ページの「同期入出力と非同期入出力の違い」 ■ 317 ページの「DMA 転送 (非同期)」
awrite(9E)	任意	非同期書き込みを実行します。awrite () エントリポイントをサポートしないドライバは、nodev(9F) エラー戻り関数を使用します。追加情報: ■ 313 ページの「同期入出力と非同期入出力の違い」 ■ 317 ページの「DMA 転送 (非同期)」
print(9E)	必須	システムコンソールにドライバメッセージを表示します。追加情報: 350 ページの「print() エントリポイント (ブロックドライバ)」
strategy(9E)	必須	ブロック入出力を実行します。追加情報: ■ 179 ページの「DMA コールバックの取り消し」 ■ 316 ページの「DMA 転送 (同期)」 ■ 319 ページの「strategy() エントリポイント」 ■ 317 ページの「DMA 転送 (非同期)」 ■ 355 ページの「一般的な制御フロー」 ■ 431 ページの「x86 ターゲットドライバの構成プロパティー」

文字デバイスドライバ用のエントリポイント

文字デバイスドライバは通常、バイトストリームで入出力を実行します。文字ドライバを使用するデバイスの例には、テープドライブやシリアルポートがあります。文字デバイスドライバは、入出力制御 (ioctl) コマンド、メモリーマッピング、デバイスポーリングなど、ブロックドライバには存在しない追加のインタフェースも提供できます。詳細は、第 15 章「文字デバイスのドライバ」を参照してください。

デバイスドライバの主なタスクは入出力を実行することですが、多くの文字デバイスドライバは、バイトストリームまたは文字 I/O と呼ばれる処理を行います。ドライバは、デバイスとの間のデータ転送を、特定のデバイスアドレスを使用しないで行います。このような転送は、ファイルシステム要求の一部がデバイス上の特定の位置を識別するブロックデバイスドライバとは大きく異なります。

read(9E) および write(9E) エントリポイントは、標準的な文字ドライバのバイトストリーム入出力を処理します。詳細は、310 ページの「入出力要求の処理」を参照してください。

次の表に、文字デバイスドライバで使用できる追加のエントリポイントの一覧を示します。その他のエントリポイントについては、41 ページの「すべてのドライバに共通のエントリポイント」を参照してください。

表 1–3 文字ドライバ用の追加のエントリポイント

エントリポイント	用途	説明
chpoll(9E)	任意	STREAMS 以外の文字ドライバのイベントをポーリングします。追加情報: 321 ページの「ファイル記述子に対する入出力の多重化」
ioctl(9E)	任意	文字ドライバの一連の入出力コマンドを実行します。ioctl() ルーチンでは、適宜 copyin(9F)、copyout(9F)、ddi_copyin(9F)、および ddi_copyout(9F) を明示的に使用して、ユーザーデータが確実にカーネルのアドレス空間に、またはカーネルのアドレス空間からコピーされるようにする必要があります。追加情報: ■ 324 ページの「ioctl() エントリポイント (文字ドライバ)」 ■ 658 ページの「よく知られている ioctl インタフェース」
read(9E)	必須	デバイスからデータを読み取ります。追加情報: ■ 311 ページの「ベクトル化された入出力」 ■ 313 ページの「同期入出力と非同期入出力の違い」 ■ 314 ページの「プログラム式入出力転送」 ■ 316 ページの「DMA 転送 (同期)」 ■ 355 ページの「一般的な制御フロー」
segmap(9E)	任意	デバイスメモリーをユーザー空間にマップします。追加情報: ■ 188 ページの「マッピングのエクスポート」 ■ 193 ページの「ユーザーアクセス用カーネルメモリーの割り当て」 ■ 210 ページの「ユーザーマッピングとドライバ通知の関連付け」
write(9E)	必須	データをデバイスに書き込みます。追加情報: ■ 124 ページの「デバイスアクセス関数」 ■ 311 ページの「ベクトル化された入出力」 ■ 313 ページの「同期入出力と非同期入出力の違い」 ■ 314 ページの「プログラム式入出力転送」 ■ 316 ページの「DMA 転送 (同期)」 ■ 355 ページの「一般的な制御フロー」

STREAMS デバイスドライバ用のエントリポイント

STREAMS は、文字ドライバを作成するための別のプログラミングモデルです。端末やネットワークデバイスなど、データを非同期に受信するデバイスは、STREAMS の実装に適しています。STREAMS デバイスドライバは、ロードと自動構成 (第 6 章「ドライバの自動構成」を参照) をサポートする必要があります。STREAMS ドライバの作成方法については、『STREAMS Programming Guide 』を参照してください。

次の表に、STREAMS デバイスドライバで使用できる追加のエントリポイントの一覧を示します。その他のエントリポイントについては、41 ページの「すべてのドライバに共通のエントリポイント」および45 ページの「文字デバイスドライバ用のエントリポイント」を参照してください。

表 1–4　STREAMS ドライバ用のエントリポイント

エントリポイント	用途	説明
put(9E)	「説明」を参照	ストリーム内のあるキューから次のキューへのメッセージの受け渡しを調整します。データを読み取るドライバの側を除き、必須です。追加情報:『STREAMS Programming Guide 』
srv(9E)	必須	キューに入っているメッセージを操作します。追加情報:『STREAMS Programming Guide 』

メモリーマッピングされたデバイス用のエントリポイント

フレームバッファーなどの特定のデバイスでは、バイトストリーム入出力を行うよりも、デバイスメモリーへの直接アクセスをアプリケーションプログラムに許可したほうが効率的です。アプリケーションのアドレス空間にデバイスメモリーをマッピングするには、mmap(2) システムコールを使用します。メモリーマッピングをサポートするために、デバイスドライバは segmap(9E) と devmap(9E) のエントリポイントを実装します。devmap(9E) については、第 10 章「デバイスメモリーおよびカーネルメモリーのマッピング」を参照してください。segmap(9E) については、第 15 章「文字デバイスのドライバ」を参照してください。

devmap(9E) エントリポイントを定義するドライバは通常、read(9E) と write(9E) のエントリポイントを定義しません。これは、アプリケーションプログラムが、mmap(2) の呼び出し後にデバイスに対して直接入出力を実行するからです。

次の表に、devmap フレームワークを使ってメモリーマッピングを実行する文字デバイスドライバで使用できる追加のエントリポイントを示します。その他のエントリポイントについては、41 ページの「すべてのドライバに共通のエントリポイント」および45 ページの「文字デバイスドライバ用のエントリポイント」を参照してください。

表 1–5 メモリーマッピングに devmap を使用する文字ドライバ用のエントリポイント

エントリポイント	用途	説明
devmap(9E)	必須	メモリーマッピングされたデバイスの仮想マッピングを検証し、変換します。追加情報: 188 ページの「マッピングのエクスポート」
devmap_access(9E)	任意	検証や保護に関する問題のあるマッピングに対してアクセスが行われたときにドライバに通知します。追加情報: 204 ページの「devmap_access() エントリポイント」
devmap_contextmgt(9E)	必須	マッピングに対してデバイスコンテキストの切り替えを行います。追加情報: 205 ページの「devmap_contextmgt() エントリポイント」
devmap_dup(9E)	任意	デバイスマッピングを複製します。追加情報: 207 ページの「devmap_dup() エントリポイント」
devmap_map(9E)	任意	デバイスマッピングを作成します。追加情報: 203 ページの「devmap_map() エントリポイント」
devmap_unmap(9E)	任意	デバイスマッピングを取り消します。追加情報: 208 ページの「devmap_unmap() エントリポイント」

ネットワークデバイスドライバ用のエントリポイント

GLDv3 (Generic LAN Driver version 3) フレームワークを使用するネットワークデバイスドライバ用のエントリポイントの一覧については、表 19–1 を参照してください。詳細は、第 19 章「ネットワークデバイスのドライバ」の 433 ページの「GLDv3 ネットワークデバイスドライバフレームワーク」および 434 ページの「GLDv3 の MAC 登録関数」を参照してください。

SCSI HBA ドライバ用のエントリポイント

次の表に、SCSI HBA デバイスドライバで使用できる追加のエントリポイントを示します。SCSI HBA トランスポート構造体については、scsi_hba_tran(9S) を参照してください。その他のエントリポイントについては、41 ページの「すべてのドライバに共通のエントリポイント」および45 ページの「文字デバイスドライバ用のエントリポイント」を参照してください。

表 1–6 SCSI HBA ドライバ用の追加のエントリポイント

エントリポイント	用途	説明
tran_abort(9E)	必須	SCSI HBA (Host Bus Adapter) ドライバに転送されている、指定の SCSI コマンドを中止します。追加情報: 426 ページの「tran_abort() エントリポイント」

表 1–6 SCSI HBA ドライバ用の追加のエントリポイント (続き)

エントリポイント	用途	説明
tran_bus_reset(9E)	任意	SCSI バスをリセットします。追加情報: 427 ページの「tran_bus_reset() エントリポイント」
tran_destroy_pkt(9E)	必須	SCSI パケットに割り当てられているリソースを解放します。追加情報: 412 ページの「tran_destroy_pkt() エントリポイント」
tran_dmafree(9E)	必須	SCSI パケットに割り当てられている DMA リソースを解放します。追加情報: 414 ページの「tran_dmafree() エントリポイント」
tran_getcap(9E)	必須	HBA ドライバによって提供される特定の機能の現在の値を取得します。追加情報: 421 ページの「tran_getcap() エントリポイント」
tran_init_pkt(9E)	必須	SCSI パケットにリソースを割り当てて初期化します。追加情報: 405 ページの「リソース割り当て」
tran_quiesce(9E)	任意	通常は動的再構成のために、SCSI バス上のすべての動作を停止します。追加情報: 428 ページの「動的再構成 (DR)」
tran_reset(9E)	必須	SCSI バスまたはターゲットデバイスをリセットします。追加情報: 426 ページの「tran_reset() エントリポイント」
tran_reset_notify(9E)	任意	バスのリセットに関して SCSI ターゲットデバイスの通知を要求します。追加情報: 427 ページの「tran_reset_notify() エントリポイント」
tran_setcap(9E)	必須	SCSI HBA ドライバによって提供される特定の機能の値を設定します。追加情報: 423 ページの「tran_setcap() エントリポイント」
tran_start(9E)	必須	SCSI コマンドのトランスポートを要求します。追加情報: 415 ページの「tran_start() エントリポイント」
tran_sync_pkt(9E)	必須	HBA ドライバまたはデバイスによるデータのビューを同期させます。追加情報: 413 ページの「tran_sync_pkt() エントリポイント」
tran_tgt_free(9E)	任意	ターゲットデバイスに代わって、割り当てられた SCSI HBA リソースが解放されるように要求します。追加情報: ■ 405 ページの「tran_tgt_free() エントリポイント」 ■ 393 ページの「トランスポート構造体の複製」
tran_tgt_init(9E)	任意	ターゲットデバイスに代わって、SCSI HBA リソースが初期化されるように要求します。追加情報: ■ 403 ページの「tran_tgt_init() エントリポイント」 ■ 389 ページの「scsi_device 構造体」
tran_tgt_probe(9E)	任意	SCSI バス上の指定されたターゲットを調べます。追加情報: 404 ページの「tran_tgt_probe() エントリポイント」
tran_unquiesce(9E)	任意	通常は動的再構成のために、tran_quiesce(9E) が呼び出されたあとで SCSI バス上の入出力動作を再開します。追加情報: 428 ページの「動的再構成 (DR)」

PC カードドライバ用のエントリポイント

次の表に、PC カードドライバで使用できる追加のエントリポイントの一覧を示します。その他のエントリポイントについては、41 ページの「すべてのドライバに共通のエントリポイント」および45 ページの「文字デバイスドライバ用のエントリポイント」を参照してください。

表 1–7　PC カードドライバ専用のエントリポイント

エントリポイント	用途	説明
csx_event_handler(9E)	必須	PC カードドライバのイベントを処理します。ドライバは、cb_ops のような構造体フィールドを使用するのではなく、csx_RegisterClient(9F) 関数を明示的に呼び出してエントリポイントを設定する必要があります。

デバイスドライバの設計上の考慮事項

デバイスドライバは、サービスのコンシューマとしてもプロバイダとしても、Oracle Solaris OS と互換性がある必要があります。このセクションでは、デバイスドライバを設計する上で考慮する点として、次の項目について説明します。

DDI/DKI の機能

Oracle Solaris DDI/DKI インタフェースは、ドライバの移植性のために提供されています。DDI/DKI を使用すると、開発者はハードウェアやプラットフォームの違いについて心配することなく標準的な方法でドライバコードを作成できます。このセクションでは、DDI/DKI インタフェースのさまざまな局面について説明します。

デバイス ID

DDI インタフェースを使用すると、ドライバは持続的な一意の ID をデバイスに割り当てることができます。デバイス ID は、デバイスを識別したり、検出したりするために使用できます。この ID は、デバイスの名前や番号 (dev_t) には依存していません。アプリケーションでは、libdevid(3LIB) で定義された関数を使用すると、ドライバが登録したデバイス ID を読み取り、操作できます。

デバイスプロパティー

デバイスまたはデバイスドライバの属性は、プロパティーによって指定されます。プロパティーは名前-値ペアです。名前は、対応付けられた値を使ってプロパティーを識別する文字列です。プロパティーは、自己識別デバイスの FCode またはハードウェア構成ファイル (`driver.conf(4)` のマニュアルページを参照) で定義することも、ドライバ自身が `ddi_prop_update(9F)` ファミリのルーチンを使用して定義することもできます。

割り込み処理

DDI/DKI では、デバイス割り込み処理の次の局面に対応します。

- システムへのデバイス割り込みの登録
- デバイス割り込みの削除
- 割り込みハンドラへの割り込みのディスパッチ

デバイス割り込みのソースは、*interrupt* と呼ばれるプロパティーに含まれています。このプロパティーは、自己識別デバイスの PROM、ハードウェア構成ファイル、または x86 プラットフォームでのブートシステムによって提供されます。

コールバック関数

一部の DDI メカニズムには、コールバックメカニズムが備わっています。DDI 関数には、条件が満たされたときにコールバックをスケジュールするためのメカニズムが備わっています。コールバック関数は、次の一般的な条件に対して使用できます。

- 転送が完了した
- リソースが利用できるようになった
- タイムアウト期間が経過した

コールバック関数は、割り込みハンドラなどのエントリポイントに多少似ています。コールバックを許可する DDI 関数は、コールバック関数が特定のタスクを実行するものとみなします。DMA ルーチンの場合、コールバック関数は、障害が発生した場合にコールバック関数を再スケジュールする必要があるかどうかを示す値を返します。

コールバック関数は、単独の割り込みスレッドとして実行されます。コールバックは、通常のマルチスレッド問題をすべて処理する必要があります。

注 – ドライバは、デバイスを切り離す前に、スケジュールされたすべてのコールバック関数を取り消す必要があります。

ソフトウェアの状態管理

状態構造体を割り当てる際にデバイスドライバの作成者を支援するために、DDI/DKI ではソフトウェア状態管理ルーチン (*soft-state* ルーチン) と呼ばれる一連のメモリー管理ルーチンを提供します。これらのルーチンは、指定されたサイズのメモリー項目の動的な割り当て、取得、および破棄を行い、リスト管理の詳細を非表示にします。インスタンス番号は、目的のメモリー項目を特定するために使われます。この番号は通常、システムによって割り当てられるインスタンス番号です。

ルーチンは、次のタスクに対して提供されます。

- ドライバのソフト状態リストを初期化する
- ドライバのソフト状態のインスタンスに領域を割り当てる
- ドライバのソフト状態のインスタンスを指すポインタを取得する
- ドライバのソフト状態のインスタンスのメモリーを解放する
- ドライバのソフト状態リストの使用を終了する

これらのルーチンの使用方法の例については、101 ページの「ロード可能なドライバインタフェース」を参照してください。

プログラム式入出力デバイスアクセス

プログラム式入出力デバイスアクセスとは、ホストの CPU からデバイスレジスタやデバイスメモリーの読み書きを行う動作のことです。Oracle Solaris DDI には、カーネルによってデバイスのレジスタやメモリーをマッピングするためのインタフェースのほかに、ドライバからデバイスメモリーの読み書きを行うためのインタフェースも備わっています。これらのインタフェースを使用すると、デバイスとホストのエンディアンネスのあらゆる違いを自動的に管理したり、デバイスで設定されたメモリーとストアのシーケンス要件を適用したりすることによって、プラットフォームやバスに依存しないドライバを開発できます。

ダイレクトメモリーアクセス (DMA)

Oracle Solaris プラットフォームでは、DMA 対応デバイスをサポートするための、アーキテクチャーに依存しないハイレベルなモデルを定義しています。Oracle Solaris DDI は、プラットフォーム固有の詳細からドライバを遮蔽します。この概念を使えば、複数のプラットフォームやアーキテクチャー上で 1 つの共通のドライバを動作させることができます。

階層化ドライバインタフェース

DDI/DKI には、階層化デバイスインタフェース (LDI) と呼ばれるインタフェースグループがあります。このインタフェースを使用すると、Oracle Solaris カーネルの内部からデバイスにアクセスできます。この機能を使用すると、開発者はカーネルデバイスの使用を監視するアプリケーションを作成できます。たとえば、`prtconf(1M)` と

fuser(1M) の両方のコマンドで LDI を使用すると、システム管理者はデバイス使用のさまざまな局面を追跡できます。LDI については、第 14 章「階層化ドライバインタフェース (LDI)」で詳しく説明しています。

ドライバコンテキスト

ドライバコンテキストとは、ドライバが現在動作している状況のことを意味します。コンテキストは、ドライバが実行できる操作を制限します。ドライバコンテキストは、呼び出される実行コードによって異なります。ドライバコードは、次の 4 つのコンテキストで実行されます。

- ユーザーコンテキスト。同期方式でユーザースレッドによって呼び出された場合、ドライバのエントリポイントにユーザーコンテキストがあります。つまり、ユーザースレッドは、呼び出されたエントリポイントからシステムが復帰するのを待ちます。たとえば、ドライバの read(9E) エントリポイントが read(2) システムコールから呼び出された場合、そのエントリポイントにユーザーコンテキストがあります。この場合、ドライバはデータをユーザースレッドにコピーしたりユーザースレッドからコピーしたりするためにユーザー領域にアクセスできます。
- カーネルコンテキスト。カーネルの一部から呼び出された場合、ドライバ関数はカーネルコンテキストで動作します。ブロックデバイスドライバでは、デバイスにページを書き込むために、strategy(9E) エントリポイントが pageout デーモンによって呼び出されることがあります。このページデーモンは現在のユーザースレッドとは関係がないため、この場合、strategy(9E) はカーネルコンテキストで動作します。
- 割り込みコンテキスト。割り込みコンテキストはカーネルコンテキストのより制限された形式です。割り込みコンテキストは、割り込みが処理された結果、呼び出されます。ドライバ割り込みルーチンは、関連付けられた割り込みレベルを使って割り込みコンテキストで動作します。コールバックルーチンも同様に割り込みコンテキストで動作します。詳細は、第 8 章「割り込みハンドラ」を参照してください。
- 高レベルの割り込みコンテキスト。高レベルの割り込みコンテキストは、割り込みコンテキストの、より制限の厳しい形式です。ddi_intr_hilevel(9F) で割り込みが高レベルであることが示された場合、ドライバ割り込みハンドラは高レベルの割り込みコンテキストで動作します。詳細は、第 8 章「割り込みハンドラ」を参照してください。

セクション 9F のマニュアルページには、各関数に使用できるコンテキストが記載されています。たとえば、カーネルコンテキストでは、ドライバは copyin(9F) を呼び出すことはできません。

エラーの出力

デバイスドライバは通常、メッセージを出力しませんが、データの破損などの予期しないエラーが発生した場合は例外です。ドライバのエントリポイントは代わりにエラーコードを返して、アプリケーションがエラーの処理方法を決定できるようにします。cmn_err(9F) 関数を使用してメッセージをシステムログに書き出せば、そのあとでコンソールに表示できます。

cmn_err(9F) によって解釈される書式文字列指定子は printf(3C) 書式文字列指定子に似ていますが、ビットフィールドを出力する %b 書式が追加されています。この書式文字列の最初の文字には特別な意味を持たせることができます。cmn_err(9F) の呼び出しでは、出力される重要度レベルを示すメッセージレベル (*level*) も指定します。詳細は、cmn_err(9F) のマニュアルページを参照してください。

CE_PANIC レベルには、システムをクラッシュさせるという副作用があります。このレベルは、システムが、続行することによってより多くの問題が発生するような不安定な状態である場合にだけ使用されます。このレベルは、デバッグ時にシステムコアダンプを取るためにも使用できます。CE_PANIC は本稼働デバイスドライバでは使用しません。

動的メモリー割り当て

デバイスドライバは、ドライバが作動させると宣言したすべての接続デバイスを同時に処理する準備ができている必要があります。ドライバが処理するデバイスの数には制限がありません。デバイスごとの情報をすべて動的に割り当てる必要があります。

```
void *kmem_alloc(size_t size, int flag);
```

標準的なカーネルメモリー割り当てルーチンは、kmem_alloc(9F) です。kmem_alloc() は C ライブラリルーチン malloc(3C) に似ていますが、flag 引数が追加されています。flag 引数には、要求されたサイズが使用できない場合に呼び出し側がブロックするかどうかを示す KM_SLEEP または KM_NOSLEEP を指定できます。KM_NOSLEEP が設定されていて、メモリーが使用できない場合、kmem_alloc(9F) は NULL を返します。

kmem_zalloc(9F) は kmem_alloc(9F) に似ていますが、割り当てられたメモリーの内容もクリアします。

注 – カーネルメモリーは限られたリソースであり、ページング不可能なため、物理メモリーをめぐってユーザーアプリケーションやカーネルの残りの部分と競合します。ドライバが大量のカーネルメモリーを割り当てると、システム性能が低下する可能性があります。

```
void kmem_free(void *cp, size_t size);
```

kmem_alloc(9F) または kmem_zalloc(9F) によって割り当てられたメモリーは、kmem_free(9F) を使ってシステムに返されます。kmem_free() は C ライブラリルーチン free(3C) に似ていますが、size 引数が追加されています。ドライバは、あとで kmem_free(9F) を呼び出すために、割り当てられた各オブジェクトのサイズを追跡し記録する必要があります。

ホットプラグによる取り付け

このマニュアルには、ホットプラグによる取り付けに関する情報は載っていません。このドキュメントに記載されたデバイスドライバ作成のための規則や提案に従えば、ドライバでホットプラグによる取り付けを扱えるようになります。特に、自動構成 (第 6 章「ドライバの自動構成」を参照) と detach(9E) の両方がドライバで正しく機能することを確認してください。また、電源管理を使用するドライバを設計している場合は、第 12 章「電源管理」に記載された情報に従うようにしてください。SCSI HBA ドライバでは、ホットプラグによる取り付け機能を利用するために、cb_ops 構造体をその dev_ops 構造体 (第 18 章「SCSI ホストバスアダプタドライバ」を参照) に追加することが必要な場合があります。

以前の Oracle Solaris OS バージョンでは、DT_HOTPLUG プロパティーを組み込むためにホットプラグ対応ドライバが必要でしたが、このプロパティーは必要なくなりました。ただし、ドライバの作成者が適切と思えば、DT_HOTPLUG プロパティーを組み込んで使用しても構いません。

◆ ◆ ◆ 第 2 章

Oracle Solaris カーネルとデバイスツリー

デバイスドライバはオペレーティングシステムの構成部分として透過的に動作する必要があります。カーネルの動作方法を理解することは、デバイスドライバについて学習するための前提条件になります。この章では、Oracle Solaris カーネルとデバイスツリーの概要を説明します。デバイスドライバの動作方法の概要については、第 1 章「Oracle Solaris デバイスドライバの概要」を参照してください。

この章では、次の内容について説明します。

カーネルとは

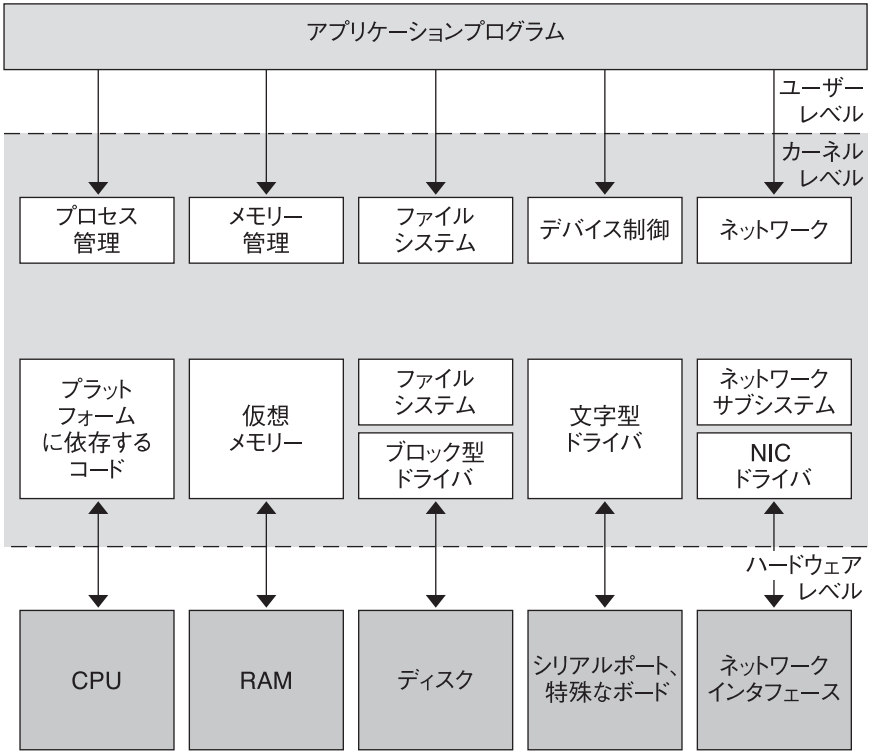
Oracle Solaris カーネルは、システムリソースを管理するプログラムです。カーネルはシステムハードウェアからアプリケーションを隔離し、入出力 (I/O) 管理、仮想メモリー、スケジューリングなどの重要なシステムサービスをアプリケーションに提供します。カーネルは、必要なときにメモリーに動的にロードされるオブジェクトモジュールから構成されます。

Oracle Solaris カーネルは論理的に 2 つの部分に分割できます。カーネルと呼ばれる最初の部分は、ファイルシステム、スケジューリング、および仮想メモリーを管理します。*I/O* サブシステムと呼ばれる 2 番目の部分は物理コンポーネントを管理します。

カーネルはアプリケーションが使用するインタフェースのセットを提供します。これらのインタフェースにはシステムコールを通してアクセスできます。システムコールについては、リファレンスマニュアルコレクション(Intro(2)を参照) の第 2 セクションに説明があります。一部のシステムコールは、デバイスドライバを呼び出して入出力を実行するために使用されます。デバイスドライバはロード可能なカーネルモジュールであり、カーネルの残りの部分をデバイスハードウェアから隔離しながら、データ転送を管理します。オペレーティングシステムとの互換性を確保するため、デバイスドライバはマルチスレッド化、仮想メモリーアドレス指定、32 ビットと 64 ビット両方の操作などの機能に対応できる必要があります。

次の図はカーネルを示しています。カーネルモジュールはアプリケーションプログラムからのシステムコールを処理します。I/O モジュールはハードウェアと通信します。

図 2–1 Oracle Solaris カーネル



カーネルは次の機能を通してデバイスドライバへのアクセスを提供します。

- デバイスからドライバへのマッピング。カーネルはデバイスツリーを維持管理します。ツリー内の各ノードは仮想デバイスまたは物理デバイスを表します。カーネルはデバイスノード名を、システムにインストールされているドライバのセットと照合し、各ノードをドライバにバインドします。アプリケーションからデバイスにアクセスできるようになるのは、ドライバのバインドが存在する場合のみです。
- **DDI/DKI** インタフェース。DDI/DKI (デバイスドライバインタフェース/ドライバカーネルインタフェース) インタフェースは、ドライバと、カーネル、デバイスハードウェア、およびブート/構成ソフトウェアとの間の対話処理を標準化します。これらのインタフェースは、カーネルからのドライバの独立を維持し、特定マシンでの継続的なオペレーティングシステムのリリース全体にわたってドライバの移植性を向上させます。
- **LDI**。LDI (階層化ドライバインタフェース) は DDI/DKI の拡張機能です。カーネルモジュールは LDI によってシステム内のその他のドライバにアクセスできます。LDI を使用すると、カーネルによって現在使用されているデバイスを特定することもできます。第 14 章「階層化ドライバインタフェース (LDI)」を参照してください。

マルチスレッドの実行環境

Oracle Solaris カーネルはマルチスレッドです。マルチプロセッサマシンでは、複数のカーネルスレッドでカーネルコードを実行可能であり、カーネルコードを平行して実行できます。カーネルスレッドも、いつでもその他のカーネルスレッドに横取りできます。

カーネルのマルチスレッド化では、デバイスドライバにいくつかの追加の制限が発生します。マルチスレッドの考慮事項の詳細情報については、第 3 章「マルチスレッド」を参照してください。デバイスドライバのコードは、多くの異なるスレッドの要求時に必要に応じて実行されるように記述する必要があります。ドライバはスレッドごとに、I/O 要求が重なることで発生する競合の問題を処理する必要があります。

仮想メモリー

Oracle Solaris 仮想メモリーシステムの全体的な概要については、このドキュメントが対象とする範囲に含まれていませんが、デバイスドライバについて説明するときに特に重要な仮想メモリー用語として、仮想アドレスとアドレス空間という 2 つの用語を使用しています。

- 仮想アドレス。仮想アドレスは、メモリー管理ユニット (MMU) によってハードウェアの物理アドレスにマップされたアドレスです。ドライバによって直接アクセスできるアドレスはすべて、カーネル仮想アドレスです。カーネル仮想アドレスは、カーネルアドレス空間を参照します。

- アドレス空間。アドレス空間は、仮想アドレスセグメントのセットです。各セグメントは、連続した範囲の仮想アドレスです。各ユーザープロセスは、ユーザーアドレス空間と呼ばれるアドレス空間を持ちます。カーネルには、カーネルアドレス空間と呼ばれる独自のアドレス空間があります。

特殊ファイルとしてのデバイス

デバイスは、ファイルシステム内では特殊ファイルによって表されます。Oracle Solaris OS では、これらのファイルは `/devices` ディレクトリ階層にあります。

特殊ファイルは、ブロック型または文字型のいずれかです。この型は、どの種類のデバイスドライバがデバイスを操作するかを示します。ドライバは両方の型で動作するように実装できます。たとえば、ディスクドライバは `fsck(1)` および `mkfs(1)` ユーティリティーで使用するために文字インタフェースをエクスポートし、ファイルシステムで使用するためにブロックインタフェースをエクスポートします。

各特殊ファイルにはデバイス番号 (`dev_t`) が関連付けられます。デバイス番号はメジャー番号とマイナー番号から構成されています。メジャー番号は、その特殊ファイルに関連付けられているデバイスドライバを識別します。マイナー番号は、特殊ファイルをさらに区別するために、デバイスドライバによって作成され、使用されます。通常、マイナー番号は、ドライバがアクセスする必要があるデバイスインスタンスと、実行する必要のあるアクセスの種類を識別するために使用されるエンコードです。たとえば、マイナー番号によって、バックアップのために使用されるテープデバイスを識別し、バックアップ操作が完了したときにはテープを巻き戻す必要があることを指定できます。

DDI/DKI インタフェース

System V Release 4 (SVR4) では、デバイスドライバと残りの UNIX カーネルとの間のインタフェースが DDI/DKI として標準化されました。DDI/DKI については、リファレンスマニュアルコレクションの第 9 セクションに説明があります。第 9E セクションにはドライバのエントリポイントについての説明、第 9F セクションにはドライバから呼び出すことができる関数についての説明、第 9S セクションにはデバイスドライバによって使用されるカーネルのデータ構造についての説明があります。`Intro(9E)`、`Intro(9F)`、および`Intro(9S)`を参照してください。

DDI/DKI はデバイスドライバと残りのカーネルの間のインタフェースをすべて標準化し、文書化することを目的としています。さらに、DDI/DKI によって、Oracle Solaris OS を実行する任意のマシンでは、プロセッサのアーキテクチャーが SPARC であるか x86 であるかにかかわらず、ドライバのソースとバイナリの互換性が実現されます。DDI/DKI の一部になっているカーネル機能のみを使用するドライバは、*DDI/DKI* 準拠デバイスドライバと呼ばれます。

DDI/DKI を使用すると、Oracle Solaris OS を実行する任意のマシンのためにプラットフォームに依存しないデバイスドライバを記述できます。これらのバイナリ互換性があるドライバによって、サードパーティーのハードウェアとソフトウェアを、より簡単に Oracle Solaris OS を実行する任意のマシンに統合できます。DDI/DKI はアーキテクチャーに依存しないため、同じドライバが一連の多様なマシンアーキテクチャーにわたって動作可能です。

プラットフォームへの非依存は、次の領域における DDI のデザインによって実現されています。

- モジュールの動的なロードおよびアンロード
- 電源管理
- 割り込み処理
- カーネルプロセスまたはユーザープロセスからデバイス領域へのアクセス。つまり、レジスタマッピングとメモリーマッピング
- DMA サービスを使用した、デバイスからのカーネルプロセス空間またはユーザープロセス空間へのアクセス
- デバイスのプロパティーの管理

デバイスツリーの概要

Oracle Solaris OS でのデバイスは、相互に接続されたデバイス情報ノードのツリーで表されます。デバイスツリーは、特定のマシンのためにロードされたデバイスの構成を記述します。

デバイスツリーコンポーネント

システムでは、ブート時にマシンに接続されていたデバイスに関する情報を含むツリー構造が構築されます。デバイスツリーは、システムの通常運用中に、動的な再構成操作によって変更することもできます。ツリーは、プラットフォームを表すルートデバイスノードで始まります。

ルートノードの下に、デバイスツリーのブランチがあります。ブランチは、1 つ以上のバスネクサスデバイスと、終端リーフデバイスから構成されます。

バスネクサスデバイスは、デバイスツリー内の従属デバイスにバスマッピングと変換サービスを提供します。PCI - PCI ブリッジ、PCMCIA アダプタ、および SCSI HBA はすべてネクサスデバイスの例です。ネクサスデバイス用ドライバの記述についての説明は、SCSI HBA ドライバの開発に限定されています (第 18 章「SCSI ホストバスアダプタドライバ」を参照)。

リーフデバイスは一般に、ディスク、テープ、ネットワークアダプタ、フレームバッファーなどの周辺デバイスです。リーフデバイスのドライバは従来の文字型ド

ライバインタフェースとブロック型ドライバインタフェースをエクスポートします。これらのインタフェースによって、ユーザープロセスはストレージや通信デバイスに対してデータの読み書きを行うことができます。

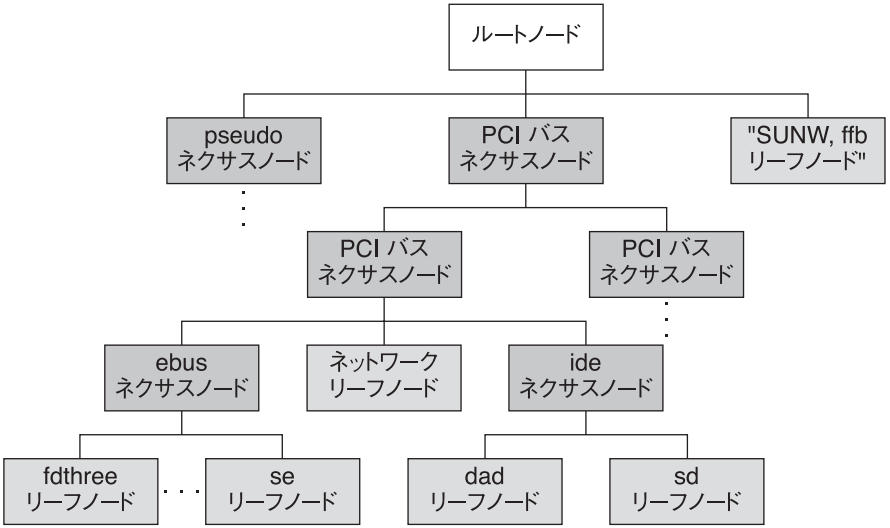
システムは次の手順を通してツリーを構築します。

1. CPU が初期化され、ファームウェアが検索されます。
2. 主要なファームウェア (OpenBoot、BIOS (Basic Input/Output System)、または `Bootconf`) が初期化され、既知または自己識別型のハードウェアを使用してデバイスツリーが作成されます。
3. 主要なファームウェアは、デバイスで互換性のあるファームウェアを見つけると、そのデバイスを初期化し、デバイスのプロパティーを取得します。
4. ファームウェアはオペレーティングシステムを見つけてブートします。
5. ツリーのルートノードでカーネルが起動され、一致するデバイスドライバを検索して、そのドライバをデバイスにバインドします。
6. ネクサスデバイスの場合、カーネルはファームウェアによって検出されていない子デバイスを探します。カーネルは、ネクサスノードの下のツリーにすべての子デバイスを追加します。
7. カーネルは、デバイスノードを作成する必要がなくなるまで手順 5 からのプロセスを繰り返します。

各ドライバはデバイス操作構造体 `dev_ops(9S)` をエクスポートして、デバイスドライバが実行できる操作を定義します。デバイス操作構造体には、`attach(9E)`、`detach(9E)`、`getinfo(9E)` などの一般的な操作への関数ポインタが格納されています。構造体には、バスネクサスドライバに固有の操作セットへのポインタと、リーフドライバに固有の操作セットへのポインタも格納されます。

ツリー構造では、ノード間に親子関係が作成されます。この親子関係がアーキテクチャー面での独立の鍵となっています。リーフドライバまたはバスネクサスドライバが、アーキテクチャーの点で性質上依存関係があるサービスを要求すると、そのドライバは自身の親に、サービスを提供するよう要求します。このアプローチによって、マシンやプロセッサのアーキテクチャーにかかわらずドライバが機能することができます。次の図に一般的なデバイスツリーを示します。

図 2–2 デバイスツリーの例



ネクサスノードは 1 つ以上の子を持つことができます。リーフノードは個々のデバイスを表します。

デバイスツリーの表示

デバイスツリーは次の 3 つの方法で表示できます。

- `libdevinfo` ライブラリは、プログラムからデバイスツリーの内容にアクセスするためのインタフェースを提供します。
- `prtconf(1M)` コマンドはデバイスツリーのすべての内容を表示します。
- `/devices` 階層は、デバイスツリーを表したものです。`ls(1)` コマンドを使用して階層を表示します。

注 – `/devices` に表示されるデバイスは、システム内でドライバが構成済みのもののみです。`prtconf(1M)` コマンドでは、デバイスのドライバがシステムに存在するかどうかにかかわらず、すべてのデバイスノードが表示されます。

`libdevinfo` ライブラリ

`libdevinfo` ライブラリは、すべての公開デバイスの構成データにアクセスするためのインタフェースを提供します。インタフェースの一覧については、`libdevinfo(3LIB)` のマニュアルページを参照してください。

prtconf コマンド

次に抜粋した prtconf(1M) コマンドの例では、システム内のすべてのデバイスを表示しています。

```
System Configuration:  Sun Microsystems  sun4u
Memory size: 128 Megabytes
System Peripherals (Software Nodes):

SUNW,Ultra-5_10
    packages (driver not attached)
        terminal-emulator (driver not attached)
        deblocker (driver not attached)
        obp-tftp (driver not attached)
        disk-label (driver not attached)
        SUNW,builtin-drivers (driver not attached)
        sun-keyboard (driver not attached)
        ufs-file-system (driver not attached)
    chosen (driver not attached)
    openprom (driver not attached)
        client-services (driver not attached)
    options, instance #0
    aliases (driver not attached)
    memory (driver not attached)
    virtual-memory (driver not attached)
    pci, instance #0
        pci, instance #0
            ebus, instance #0
                auxio (driver not attached)
                power, instance #0
                SUNW,pll (driver not attached)
                se, instance #0
                su, instance #0
                su, instance #1
                ecpp (driver not attached)
                fdthree, instance #0
                eeprom (driver not attached)
                flashprom (driver not attached)
                SUNW,CS4231 (driver not attached)
            network, instance #0
            SUNW,m64B (driver not attached)
            ide, instance #0
                disk (driver not attached)
                cdrom (driver not attached)
                dad, instance #0
                sd, instance #15
        pci, instance #1
            pci, instance #0
                pci108e,1000 (driver not attached)
                SUNW,hme, instance #1
                SUNW,isptwo, instance #0
                    sd (driver not attached)
                    st (driver not attached)
                    sd, instance #0 (driver not attached)
                    sd, instance #1 (driver not attached)
                    sd, instance #2 (driver not attached)
                    ...
    SUNW,UltraSPARC-IIi (driver not attached)
```

```
    SUNW,ffb, instance #0
    pseudo, instance #0
```

/devices ディレクトリ

/devices 階層では、デバイスツリーを表す名前空間が提供されます。次に示すのは、/devices 名前空間の短縮形の一覧です。このサンプル出力は、前に示したデバイスツリーの例と prtconf(1M) の出力に対応しています。

```
/devices
/devices/pseudo
/devices/pci@1f,0:devctl
/devices/SUNW,ffb@1e,0:ffb0
/devices/pci@1f,0
/devices/pci@1f,0/pci@1,1
/devices/pci@1f,0/pci@1,1/SUNW,m64B@2:m640
/devices/pci@1f,0/pci@1,1/ide@3:devctl
/devices/pci@1f,0/pci@1,1/ide@3:scsi
/devices/pci@1f,0/pci@1,1/ebus@1
/devices/pci@1f,0/pci@1,1/ebus@1/power@14,724000:power_button
/devices/pci@1f,0/pci@1,1/ebus@1/se@14,400000:a
/devices/pci@1f,0/pci@1,1/ebus@1/se@14,400000:b
/devices/pci@1f,0/pci@1,1/ebus@1/se@14,400000:0,hdlc
/devices/pci@1f,0/pci@1,1/ebus@1/se@14,400000:1,hdlc
/devices/pci@1f,0/pci@1,1/ebus@1/se@14,400000:a,cu
/devices/pci@1f,0/pci@1,1/ebus@1/se@14,400000:b,cu
/devices/pci@1f,0/pci@1,1/ebus@1/ecpp@14,3043bc:ecpp0
/devices/pci@1f,0/pci@1,1/ebus@1/fdthree@14,3023f0:a
/devices/pci@1f,0/pci@1,1/ebus@1/fdthree@14,3023f0:a,raw
/devices/pci@1f,0/pci@1,1/ebus@1/SUNW,CS4231@14,200000:sound,audio
/devices/pci@1f,0/pci@1,1/ebus@1/SUNW,CS4231@14,200000:sound,audioctl
/devices/pci@1f,0/pci@1,1/ide@3
/devices/pci@1f,0/pci@1,1/ide@3/sd@2,0:a
/devices/pci@1f,0/pci@1,1/ide@3/sd@2,0:a,raw
/devices/pci@1f,0/pci@1,1/ide@3/dad@0,0:a
/devices/pci@1f,0/pci@1,1/ide@3/dad@0,0:a,raw
/devices/pci@1f,0/pci@1
/devices/pci@1f,0/pci@1/pci@2
/devices/pci@1f,0/pci@1/pci@2/SUNW,isptwo@4:devctl
/devices/pci@1f,0/pci@1/pci@2/SUNW,isptwo@4:scsi
```

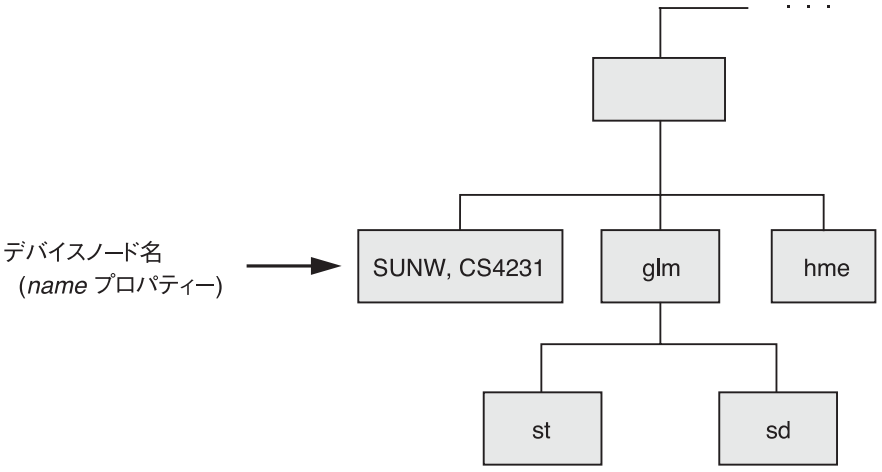
ドライバのデバイスへのバインド

カーネルは、デバイスツリーを構築することに加えて、デバイスの管理に使用されるドライバを判別します。

ドライバのデバイスへのバインドとは、システムが特定のデバイスを管理するためにドライバを選択するプロセスを指します。バインド名とは、ドライバを、デバイス情報ツリー内の一意のデバイスノードにリンクする名前のことです。システムはデバイスツリー内のデバイスごとに、インストール済みのドライバの一覧からドライバを選択しようとします。

各デバイスノードには関連付けられた *name* プロパティーがあります。このプロパティーは、PROM などの外部エージェントから割り当てる、システムブート時に割り当てる、または `driver.conf` 構成ファイルから割り当てることができます。どの場合にも、`name` プロパティーはデバイスツリー内のデバイスに割り当てられるノード名を表します。ノード名は、`/devices` に表示され、`prtconf(1M)` の出力に一覧で示される名前です。

図 2–3 デバイスノード名



デバイスノードには *compatible* プロパティーを関連付けることもできます。*compatible* プロパティーには、1 つ以上の使用可能なドライバ名またはデバイスのドライバ別名の一覧が、並び替えて格納されています。

システムは *compatible* プロパティーと *name* プロパティーの両方を使用してデバイスのドライバを選択します。システムは *compatible* プロパティーが存在する場合、最初に *compatible* プロパティーの内容を、システム上のドライバと照合しようとします。システムは *compatible* プロパティーの一覧に記載された最初のドライバ名から処理を開始し、ドライバ名をシステム上の既知のドライバと照合しようとします。一覧の各エントリは、システムが一致するものを見つけるか、一覧の最後に達するまで処理されます。

name プロパティーまたは *compatible* プロパティーのいずれかの内容がシステム上のドライバと一致する場合、そのドライバがデバイスノードにバインドされます。一致するものが見つからない場合、どのドライバもデバイスノードにはバインドされません。

汎用デバイス名

一部のデバイスでは、*name* プロパティーの値として汎用デバイス名が指定されています。汎用デバイス名は、そのデバイス用に特定のドライバを実際に結び付けるのではなく、デバイスの機能を説明するものです。たとえば、ある SCSI ホストバスアダプタが `scsi` という汎用デバイス名を持っている場合があります。また、イーサネットデバイスが `ethernet` という汎用デバイス名を持っていることがあります。

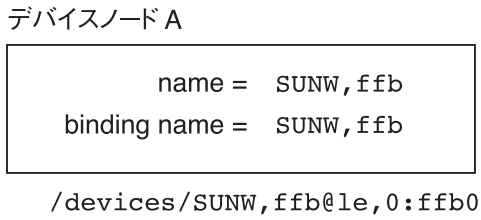
compatible プロパティーを利用すると、システムは汎用デバイス名を持つデバイス用の代替ドライバを判定できます。たとえば、`scsi` HBA デバイスドライバには `glm` を、`ethernet` デバイスドライバには `hme` を見つけることができます。

汎用デバイス名を持つデバイスには、*compatible* プロパティーを指定する必要があります。

注 – 汎用デバイス名の詳細な説明については、『IEEE 1275 Open Firmware Boot Standard』を参照してください。

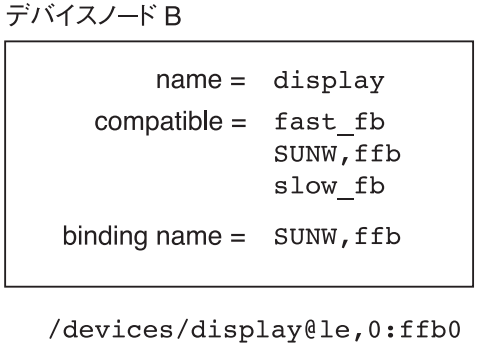
次の図は、固有のデバイス名を持つデバイスノードを示しています。ドライバのバインド名 `SUNW,ffb` はデバイスノード名と同じ名前です。

図 2–4 ドライバノードの固有名のバインド



次の図は、汎用デバイス名 `display` を持つデバイスノードを示しています。ドライバのバインド名 `SUNW,ffb` は、システムのドライバ一覧に記載されているドライバと一致している、*compatible* プロパティーのドライバ一覧で最初の名前です。この場合、`display` がフレームバッファーの汎用デバイス名です。

図2–5 汎用ドライバノードのバインド



◆ ◆ ◆ 第 3 章

3 マルチスレッド

この章では、Oracle Solaris マルチスレッドカーネルのロックプリミティブおよびスレッド同期メカニズムについて説明します。マルチスレッドを利用するようにデバイスドライバを設計することをお勧めします。この章では、次の内容について説明します。

ロックプリミティブ

従来の UNIX システムでは、カーネルコードのどのセクションも、`sleep(1)` を明示的に呼び出してプロセッサを放棄するか、ハードウェア割り込みを使用することによって終了します。Oracle Solaris OS の動作は異なります。別のスレッドを実行するためにカーネルスレッドをいつでも横取りできます。すべてのカーネルスレッドはカーネルアドレス空間を共有し、同じデータの読み取りおよび変更を行う必要がしばしば生じるため、スレッドが共有データを破棄することを防ぐために、カーネルは多くのロックプリミティブを提供します。これらのメカニズムには、相互排他ロック (*mutex* とも呼ばれる)、読み取り/書き込みロック、およびセマフォーが含まれます。

ドライバデータのストレージクラス

データへのアクセスを制御するための明示的手順をドライバで実行する必要があるかどうかは、データのストレージクラスによって決まります。データストレージクラスには次の 3 種類があります。

- 自動 (スタック) データ。すべてのスレッドはプライベートスタックを備えているため、ドライバで自動変数のロックが必要になることはありません。

- グローバル静的データ。グローバル静的データは、ドライバ内の任意の数のスレッドによって共有できます。ドライバでこの種類のデータをロックすることが必要になる場合があります。
- カーネルヒープデータ。kmem_alloc(9F) によって割り当てられるデータなど、カーネルヒープデータは、ドライバ内の任意の数のスレッドによって共有できます。ドライバはこのような共有データを常に保護する必要があります。

相互排他ロック

相互排他ロック (*mutex*) は通常、データセットと関連付けられ、そのデータへのアクセスを制御します。mutex は、ある時点で 1 つのスレッドのみがそのデータにアクセスできるようにする手段を提供します。mutex 関数には次のものがあります。

mutex_destroy(9F)	関連付けられたストレージを解放します。
mutex_enter(9F)	mutex を取得します。
mutex_exit(9F)	mutex を解放します。
mutex_init(9F)	mutex を初期化します。
mutex_owned(9F)	mutex が現在のスレッドによって保持されているかどうかを評価します。ASSERT(9F) でのみ使用します。
mutex_tryenter(9F)	利用可能な場合に mutex を取得しますが、ブロックはしません。

mutex の設定

デバイスドライバは通常、ドライバデータ構造ごとに mutex を割り当てます。一般に mutex は、kmutex_t 型の構造体のフィールドです。使用する mutex を準備するために mutex_init(9F) が呼び出されます。この呼び出しは通常、デバイスごとの mutex については attach(9E) の時点で、グローバルドライバ mutex については _init(9E) の時点で行われます。

例:

```
struct xxstate *xsp;
/* ... */
mutex_init(&xsp->mu, NULL, MUTEX_DRIVER, NULL);
/* ... */
```

mutex 初期化のより完全な例については、第 6 章「ドライバの自動構成」を参照してください。

ドライバはアンロードされる前に mutex_destroy(9F) を使用して mutex を破棄する必要があります。mutex の破棄は通常、デバイスごとの mutex については detach(9E) の時点で、グローバルドライバ mutex については _fini(9E) の時点で行われます。

mutex の使用

共有データ構造を読み書きする必要があるドライバコードのすべてのセクションで、次のタスクを実行する必要があります。

- mutex の取得
- データへのアクセス
- mutex の解放

mutex のスコープ、つまり mutex が保護するデータはすべてプログラマが管理します。mutex によってデータ構造が保護されるのは、データ構造にアクセスするすべてのコードパスが、mutex の保持中にデータにアクセスする場合に限られます。

読み取り/書き込みロック

読み取り/書き込みロックはデータセットへのアクセスを制御します。読み取り/書き込みロックに関しては、多数のスレッドが読み取りのためにロックを同時に保持できる一方で、書き込みのためにロックを保持できるのは 1 つのスレッドに限られます。

ほとんどのデバイスドライバは読み取り/書き込みロックを使用しません。これらのロックは mutex よりも低速です。これらのロックがパフォーマンス面で優位になるのは、読み取りの割合が多く、書き込みの頻度の低いデータを保護するときのみです。このようなケースでは、mutex の競合がボトルネックになる可能性があるため、読み取り/書き込みロックを使用したほうが効率的な場合があります。読み取り/書き込みロック関数を次の表にまとめています。詳細は、rwlock(9F) のマニュアルページを参照してください。読み取り/書き込みロック関数には次のものがあります。

rw_destroy(9F)	読み取り/書き込みロックを破棄します。
rw_downgrade(9F)	読み取り/書き込みロックを保持するスレッドを書き込みから読み取りに降格します。
rw_enter(9F)	読み取り/書き込みロックを取得します。
rw_exit(9F)	読み取り/書き込みロックを解放します。
rw_init(9F)	読み取り/書き込みロックを初期化します
rw_read_locked(9F)	読み取り/書き込みロックが、読み取りと書き込みのどちらの目的で保持されているかを調べます。
rw_tryenter(9F)	待機することなく、読み取り/書き込みロックの取得を試みます。
rw_tryupgrade(9F)	読み取り/書き込みロックを読み取りから書き込みに昇格しようと試みます。

セマフォー

カウントセマフォーは、デバイスドライバ内部でスレッドを管理するための代替のプリミティブとして使用できます。詳細については、semaphore(9F) のマニュアルページを参照してください。セマフォー関数には次のものがあります。

sema_destroy(9F)	セマフォーを破棄します。
sema_init(9F)	セマフォーを初期化します。
sema_p(9F)	セマフォーを 1 減らします。ブロックする可能性があります。
sema_p_sig(9F)	セマフォーを 1 減らしますが、シグナルが保留中の場合にはブロックしません。77 ページの「スレッドがシグナルを受信できない」を参照してください。
sema_tryp(9F)	セマフォーを 1 減らそうと試みますが、ブロックしません。
sema_v(9F)	セマフォーを 1 増やします。待機中スレッドをブロック解除する可能性があります。

スレッド同期

共有データの保護に加えて、ドライバでは複数スレッドの実行を同期することがしばしば必要になります。

スレッド同期における条件変数

条件変数はスレッド同期の標準的な形式です。条件変数は mutex と組み合わせて使用するように設計されています。mutex を関連付けて使用することにより、条件を原子的にチェックできることが保証されます。また、条件の変更や、条件が変更されたことのシグナルを取りこぼさずに、関連付けられた条件変数でスレッドをブロックできることも保証されます。

condvar(9F) 関数には次のものがあります。

cv_broadcast(9F)	条件変数で待機しているすべてのスレッドにシグナルを送信します。
cv_destroy(9F)	条件変数を破棄します。
cv_init(9F)	条件変数を初期化します。
cv_signal(9F)	条件変数で待機している 1 つのスレッドにシグナルを送信します。

cv_timedwait(9F)	条件、タイムアウト、またはシグナルを待機します。77 ページの「スレッドがシグナルを受信できない」を参照してください。
cv_timedwait_sig(9F)	条件またはタイムアウトを待機します。
cv_wait(9F)	条件を待機します。
cv_wait_sig(9F)	条件を待機します。シグナルを受信した場合は 0 を返します。77 ページの「スレッドがシグナルを受信できない」を参照してください。

条件変数の初期化

条件ごとに kcondvar_t 型の条件変数を宣言します。条件変数は通常、ドライバのソフト状態構造体で宣言されます。それぞれの条件変数を初期化するには、cv_init(9F) を使用します。条件変数は通常、mutex と同様に attach(9E) の時点で初期化されます。条件変数の初期化の一般的な例を次に示します。

```
cv_init(&xsp->cv, NULL, CV_DRIVER, NULL);
```

条件変数の初期化のより完全な例については、第 6 章「ドライバの自動構成」を参照してください。

条件の待機

条件変数を使用するには、条件を待機するコードパスで次の手順に従います。

1. 条件をガードしている mutex を取得します。
2. 条件を評価します。
3. 評価結果によりスレッドの続行が許可されない場合、cv_wait(9F) を使用して、その条件で現在のスレッドをブロックします。cv_wait(9F) 関数はスレッドをブロックする前に mutex を解放し、戻る前に mutex を再取得します。cv_wait(9F) から戻ったら、評価を繰り返します。
4. 評価によりスレッドの再開が許可されたら、条件を新しい値に設定します。たとえば、デバイスフラグをビジーに設定します。
5. mutex を解放します。

条件のシグナル送信

条件のシグナルを送信するには、コードパスで次の手順に従います。

1. 条件をガードしている mutex を取得します。
2. 条件を設定します。
3. cv_broadcast(9F) を使用して、ブロックされているスレッドにシグナルを送信します。

4. mutex を解放します。

次の例では、mutex と条件変数に加えてビジーフラグを使用し、転送を開始する前に、デバイスがビジー状態でなくなるまで read(9E) ルーチンを強制的に待機させます。

例 3–1 mutex と条件変数の使用

```
static int
xxread(dev_t dev, struct uio *uiop, cred_t *credp)
{
        struct xxstate *xsp;
        /* ... */
        mutex_enter(&xsp->mu);
        while (xsp->busy)
                cv_wait(&xsp->cv, &xsp->mu);
        xsp->busy = 1;
        mutex_exit(&xsp->mu);
        /* perform the data access */
}

static uint_t
xxintr(caddr_t arg)
{
        struct xxstate *xsp = (struct xxstate *)arg;
        mutex_enter(&xsp->mu);
        xsp->busy = 0;
        cv_broadcast(&xsp->cv);
        mutex_exit(&xsp->mu);
}
```

cv_wait() および cv_timedwait() 関数

cv_wait(9F) によってスレッドが条件でブロックされ、その条件が発生しない場合、スレッドは待機し続けることになります。その状況を避けるには、別のスレッドに依存して復帰処理を実行する cv_timedwait(9F) を使用します。cv_timedwait() は絶対待ち時間を引数に取ります。指定された時間に達してもイベントが発生しなかった場合、cv_timedwait() は -1 を返します。条件が満たされた場合、cv_timedwait() は正の値を返します。

cv_timedwait(9F) には、システムが最後にリブートしてからのクロックティックで表した絶対待ち時間を指定する必要があります。待ち時間は、ddi_get_lbolt(9F) で現在の値を取得することによって調べることができます。ドライバは通常、最大待ち時間の値を秒またはマイクロ秒単位で保持しているため、drv_usectohz(9F) によってこの値がクロックティックに変換され、ddi_get_lbolt(9F) で得られた値に加算されます。

次の例は、呼び出し元に EIO を返す前に、cv_timedwait(9F) を使用してデバイスアクセスを最大 5 秒待機する方法を示しています。

例 3–2 cv_timedwait() の使用

```
clock_t            cur_ticks, to;
mutex_enter(&xsp->mu);
while (xsp->busy) {
        cur_ticks = ddi_get_lbolt();
        to = cur_ticks + drv_usectohz(5000000); /* 5 seconds from now */
        if (cv_timedwait(&xsp->cv, &xsp->mu, to) == -1) {
                /*
                 * The timeout time 'to' was reached without the
                 * condition being signaled.
                 */
                /* tidy up and exit */
                mutex_exit(&xsp->mu);
                return (EIO);
        }
}
xsp->busy = 1;
mutex_exit(&xsp->mu);
```

デバイスドライバの書き込み処理では通常、cv_wait(9F) よりも cv_timedwait(9F) を優先的に使用しますが、cv_wait(9F) を使用するほうが望ましい場合があります。たとえば、ドライバが次のような条件で待機している場合は、cv_wait(9F) のほうが適しています。

- 内部ドライバ状態の変更。このような状態変更が発生するには、何らかのコマンドの実行または一定時間の経過が必要な場合があります。
- ドライバでシングルスレッド処理する必要がある何らかの要素
- タイムアウトの可能性をすでに管理している何らかの状況。「A」が「B」に依存しており、「B」が cv_timedwait(9F) を使用している場合などです。

cv_wait_sig() 関数

発生する可能性がない条件や、長期間にわたって発生しない条件をドライバが待機している場合があります。そのような場合、ユーザーがシグナルを送信してスレッドを終了させることができます。ドライバの設計によっては、シグナルを送信してもドライバが復帰しない可能性があります。

cv_wait_sig(9F) を使用すると、シグナルによってスレッドのブロックを解除できます。ユーザーは kill(1) を使用してシグナルをスレッドに送信するか、または割り込み文字を入力することによって、長期に及ぶ可能性がある待機を解除できます。cv_wait_sig(9F) は、シグナルを送信してから戻る場合は 0 を返し、条件が発生した場合は 0 以外を返します。ただし、シグナルを受信できないケースも存在します。77 ページの「スレッドがシグナルを受信できない」を参照してください。

次の例は、cv_wait_sig(9F) を使用してシグナルを送信し、スレッドのブロックを解除する方法を示しています。

例 3–3 cv_wait_sig() の使用

```
mutex_enter(&xsp->mu);
while (xsp->busy) {
        if (cv_wait_sig(&xsp->cv, &xsp->mu) == 0) {
        /* Signaled while waiting for the condition */
                /* tidy up and exit */
                mutex_exit(&xsp->mu);
                return (EINTR);
        }
}
xsp->busy = 1;
mutex_exit(&xsp->mu);
```

cv_timedwait_sig() 関数

cv_timedwait_sig(9F) は cv_timedwait(9F) および cv_wait_sig(9F) に似ています。異なる点として、cv_timedwait_sig() はタイムアウト到達後、条件のシグナルを送信せずに -1 を返します。また、kill(2) などのシグナルがスレッドに送信された場合は 0 を返します。

cv_timedwait(9F) と cv_timedwait_sig(9F) の両方で、時間は最後にシステムがリブートしてからの絶対クロックティックで計測されます。

ロック構成の選択

ほとんどのデバイスドライバで、ロックスキームを簡潔に保つことをお勧めします。追加のロックを使用すると並行性が向上しますが、オーバーヘッドが増加します。ロックの使用数を減らすと消費時間が減少しますが、並行性は低下します。一般的な目安として、データ構造ごとに 1 つの mutex を、ドライバが待機する必要があるイベントまたは条件ごとに 1 つの条件変数を、ドライバ単位でグローバルな主要データセットごとに 1 つの mutex を使用します。長期間にわたって mutex を保持しないでください。ロック構成を選択するときは次のガイドラインを使用します。

- エントリポイントのマルチスレッド動作のうち、十分な利点が得られるものを使用します。
- すべてのエントリポイントを再入可能にします。静的変数を自動変数に変更することにより、共有データの量を減らすことができます。
- ドライバで複数の mutex を取得する場合は、すべてのコードパスで、同じ順序で mutex を取得および解放します。
- 同じ関数空間の内部でロックを保持および解放します。
- KM_SLEEP を指定した kmem_alloc(9F) のように、ブロックの可能性がある DDI インタフェースを呼び出すときはドライバ mutex を保持しないようにします。

ロックの使用状態を確認するには、lockstat(1M) を使用します。lockstat(1M) はすべてのカーネルロックイベントを監視し、イベントの頻度およびタイミングデータを収集し、データを表示します。

マルチスレッド操作の詳細は、『マルチスレッドのプログラミング』を参照してください。

ロックの潜在的な危険性

mutex は同じスレッドによって再入可能ではありません。mutex をすでに保有している場合、この mutex をもう一度取得しようとすると次のパニックが発生します。

```
panic: recursive mutex_enter. mutex %x caller %x
```

現在のスレッドが保持していない mutex を解放すると、次のパニックが発生します。

```
panic: mutex_adaptive_exit: mutex not held by thread
```

次のパニックは単一プロセッサ環境でのみ発生します。

```
panic: lock_set: lock held and only one CPU
```

lock_set パニックは、ほかの CPU がスピン mutex を解放できないため、この mutex が保持されてスピンし続けることを示します。ドライバが特定のコードパスで mutex を解放しなかった場合や、mutex の保持中にドライバがブロックされた場合に、この状況に陥る可能性があります。

lock_set パニックのよくある原因は、割り込みレベルの高いデバイスが、cv_wait(9F) のようにブロック処理を行うルーチンを呼び出したときに発生します。別の一般的な原因は、mutex_enter(9F) を呼び出すことによって適応型 mutex を獲得する高レベルハンドラです。

スレッドがシグナルを受信できない

スレッドがシグナルを受信したときに、sema_p_sig()、cv_wait_sig()、および cv_timedwait_sig() の各関数を復帰させることができます。一部のスレッドがシグナルを受信できないことが原因で問題が起きる可能性があります。たとえば、アプリケーションが close(2) を呼び出した結果として close(9E) が呼び出されたときは、シグナルの受信が可能です。一方、開いているすべてのファイル記述子を閉じる exit(2) 処理の内部から close(9E) が呼び出されたとき、スレッドはシグナルを受

信できません。スレッドがシグナルを受信できないとき、sema_p_sig() は sema_p() として、cv_wait_sig() は cv_wait() として、cv_timedwait_sig() は cv_timedwait() としてそれぞれ動作します。

発生しない可能性があるイベントで、スリープし続けないように注意する必要があります。決して発生しないイベントは、強制終了できない (機能停止した) スレッドを作成し、システムをリブートするまでデバイスを使用不能にします。機能停止したプロセスはシグナルを受信できません。

現在のスレッドがシグナルを受信できるかどうかを検出するには、ddi_can_receive_sig(9F) 関数を使用します。ddi_can_receive_sig() 関数が B_TRUE を返す場合、受信したシグナルによって前述の関数を復帰させることができます。ddi_can_receive_sig() 関数が B_FALSE を返す場合、受信したシグナルによって前出の関数群を復帰させることはできません。ddi_can_receive_sig() 関数が B_FALSE を返す場合、デバイスドライバでは timeout(9F) 関数などの代替手段を使用して、再度復帰処理を行うべきです。

この問題が発生する重要なケースの 1 つはシリアルポートです。リモートシステムがフロー制御を表明し、出力データの排出を試みる間 close(9E) 関数がブロックする場合、フロー制御条件が解決されるか、システムがリブートされるまでポートがスタックする可能性があります。そのようなドライバでは、このケースを検出し、フロー制御条件の持続時間が長すぎるときに排出操作を中止するためのタイマーを設定するべきです。

この問題は、『STREAMS Programming Guide』の第 7 章「STREAMS Framework – Kernel Level」で説明する qwait_sig(9F) 関数にも影響します。

◆ ◆ ◆ 第 4 章

4 プロパティー

プロパティーは、DDI/DKI インタフェースを使って管理される、ユーザー定義の名前-値ペアの構造体です。この章では、次の内容について説明します。

デバイスプロパティー

デバイス属性情報は、プロパティーと呼ばれる名前-値ペアの表記で表すことができます。

たとえば、デバイスレジスタとオンボードメモリーは reg プロパティーで表すことができます。reg プロパティーは、デバイスハードウェアレジスタを記述するソフトウェアの抽象化です。reg プロパティーの値により、デバイスレジスタのアドレス位置とサイズがエンコードされます。ドライバは、reg プロパティーを使用してデバイスレジスタにアクセスします。

別の例として interrupt プロパティーがあります。interrupt プロパティーは、デバイス割り込みを表します。interrupt プロパティーの値により、デバイス割り込み PIN がエンコードされます。

プロパティーには次の 5 つの型の値を割り当てることができます。

- バイト配列 – 任意の長さの一連のバイト
- 整数プロパティー – 整数値
- 整数配列プロパティー – 整数の配列
- 文字列プロパティー – null で終わる文字列
- 文字列配列プロパティー – null で終わる文字列のリスト

値を持たないプロパティーは、boolean プロパティーとみなされます。boolean プロパティーが存在する場合は真 (true) です。boolean プロパティーが存在しない場合は偽 (false) です。

デバイスプロパティー名

厳密に言えば、DDI/DKI ソフトウェアのプロパティー名には制限がありません。ただし、推奨される使い方がいくつかあります。IEEE 1275-1994「Standard for Boot Firmware」では、プロパティーは次のように定義されています。

> プロパティーとは、1 から 31 文字までのプリント可能文字で構成される、人間が読めるテキスト文字列のことです。プロパティー名には、大文字や「/」、「\」、「:」、「[」、「]」、「@」の文字を含めることはできません。「+」文字で始まるプロパティー名は、将来のバージョンの IEEE 1275-1994 用に予約されています。

通常、下線はプロパティー名に使用しません。代わりにハイフン (-) を使用します。通常、疑問符 (?) で終わるプロパティー名には、文字列の値 (一般には TRUE または FALSE) を指定します (例: `auto-boot?`)。

ドライバ構成ファイルへのプロパティーの追加については、`driver.conf(4)` のマニュアルページを参照してください。`pm(9P)` と `pm-components(9P)` のマニュアルページには、電源管理におけるプロパティーの使用方法が記載されています。デバイスドライバのマニュアルページにプロパティーがどのように文書化されているかの例として、`sd(7D)` のマニュアルページをお読みください。

プロパティーの作成と更新

ドライバのプロパティーを作成したり、既存のプロパティーを更新したりするには、適切なプロパティー型を持つ DDI ドライバ更新インタフェース (`ddi_prop_update_int(9F)` や `ddi_prop_update_string(9F)` など) の 1 つを使用します。使用できるプロパティーインタフェースの一覧については、表 4–1 を参照してください。これらのインタフェースは通常、ドライバの `attach(9E)` エントリポイントから呼び出されます。次の例では、`ddi_prop_update_string()` は、`needs-suspend-resume` という値を使って `pm-hardware-state` という文字列プロパティーを作成します。

```
/* The following code is to tell cpr that this device
 * needs to be suspended and resumed.
 */
(void) ddi_prop_update_string(device, dip,
     "pm-hardware-state", "needs-suspend-resume");
```

ほとんどの場合、プロパティーの更新には `ddi_prop_update()` ルーチンを使用すれば十分です。ただし、頻繁に値が変わるプロパティーの場合は更新による

オーバーヘッドが原因で、パフォーマンスに関する問題が発生することがあります。ddi_prop_update() の使用を避けるためにプロパティー値のローカルインスタンスを使用する方法については、82 ページの「prop_op() エントリポイント」を参照してください。

プロパティーの検索

ドライバは、その親にプロパティーを要求できます。そしてその親は、さらにその親に要求できます。ドライバは、要求をその親よりも上位に対して行えるかどうかを制御できます。

たとえば、次の例の esp ドライバは、ターゲットごとに targetx-sync-speed という整数プロパティーを保持します。targetx-sync-speed 内の x はターゲット番号を表します。prtconf(1M) コマンドは、ドライバのプロパティーを冗長モードで表示します。次の例は、esp ドライバの部分的なリストを示しています。

```
% prtconf -v
...
       esp, instance #0
            Driver software properties:
                name <target2-sync-speed> length <4>
                    value <0x00000fa0>.
...
```

次の表に、プロパティーインタフェースのサマリーを示します。

表 4–1 プロパティーインタフェースの使い方

ファミリ	プロパティーインタフェース	説明
ddi_prop_lookup	ddi_prop_exists(9F)	プロパティーを検索し、そのプロパティーが存在する場合は正常に復帰します。プロパティーが存在しない場合は失敗します。
	ddi_prop_get_int(9F)	整数プロパティーを検索し、それを返します。
	ddi_prop_get_int64(9F)	64 ビットの整数プロパティーを検索し、それを返します。
	ddi_prop_lookup_int_array(9F)	整数配列プロパティーを検索し、それを返します。
	ddi_prop_lookup_int64_array(9F)	64 ビットの整数配列プロパティーを検索し、それを返します。
	ddi_prop_lookup_string(9F)	文字列プロパティーを検索し、それを返します。

表 4–1 プロパティーインタフェースの使い方 (続き)

ファミリ	プロパティーインタフェース	説明
	ddi_prop_lookup_string_array(9F)	文字列配列プロパティーを検索し、それを返します。
	ddi_prop_lookup_byte_array(9F)	バイト配列プロパティーを検索し、それを返します。
ddi_prop_update	ddi_prop_update_int(9F)	整数プロパティーを更新または作成します。
	ddi_prop_update_int64(9F)	単一の 64 ビットの整数プロパティーを更新または作成します。
	ddi_prop_update_int_array(9F)	整数配列プロパティーを更新または作成します。
	ddi_prop_update_string(9F)	文字列プロパティーを更新または作成します。
	ddi_prop_update_string_array(9F)	文字列配列プロパティーを更新または作成します。
	ddi_prop_update_int64_array(9F)	64 ビットの整数配列プロパティーを更新または作成します。
	ddi_prop_update_byte_array(9F)	バイト配列プロパティーを更新または作成します。
ddi_prop_remove	ddi_prop_remove(9F)	プロパティーを削除します。
	ddi_prop_remove_all(9F)	デバイスに関連付けられているすべてのプロパティーを削除します。

int プロパティーインタフェースを使用するときは、できるだけ ddi_prop_update_int64(9F) などの 32 ビットバージョンではなく、ddi_prop_update_int(9F) などの 64 ビットバージョンを使用してください。

prop_op() エントリポイント

prop_op(9E) エントリポイントは通常、デバイスプロパティーまたはドライバプロパティーの値をシステムに報告するために必要です。ドライバが専用のプロパティーを作成または管理する必要がない場合は、ddi_prop_op(9F) 関数をこのエントリポイントに使用できます。

ddi_prop_op(9F) がデバイスドライバの cb_ops(9S) 構造体で定義されている場合は、ddi_prop_op() をそのドライバの prop_op(9E) エントリポイントとして使用できます。ddi_prop_op() を使用すると、リーフデバイスでプロパティー値を検索して、それをデバイスのプロパティーリストから取得できます。

頻繁に値が変わるプロパティーをドライバで保持する必要がある場合は、ddi_prop_op() を呼び出すのではなく、cb_ops 構造体の中にドライバ固有の prop_op() ルーチンを定義します。この手法により、ddi_prop_update() を繰り返し使用するという非効率性を解消できます。ドライバは、そのソフト状態構造体の中またはドライバ変数でプロパティー値のコピーを保持します。

prop_op(9E) エントリポイントは、特定のドライバプロパティーやデバイスプロパティーの値をシステムに報告します。多くの場合、ddi_prop_op(9F) ルーチンは、cb_ops(9S) 構造体でドライバの prop_op() エントリポイントとして使用できます。ddi_prop_op() は、必要なすべての処理を実行します。デバイスプロパティーリクエストの処理時に特別な処理を必要としないドライバには、ddi_prop_op() で十分です。

ただし、ドライバが prop_op() エントリポイントを提供することが必要な場合があります。たとえば、頻繁に値が変わるプロパティーをドライバで保持する場合、変更のたびに ddi_prop_update(9F) を使用してプロパティーを更新することは効率的ではありません。代わりに、ドライバはインスタンスのソフト状態によってプロパティーのシャドウコピーを保持します。ドライバは、プロパティーの値が変わると、ddi_prop_update() ルーチンを一切使わずにシャドウコピーを更新します。prop_op() エントリポイントは、ddi_prop_op() にリクエストを渡してプロパティーリクエストを処理する前に、このプロパティーのリクエストを遮断し、いずれかの ddi_prop_update() ルーチンを使用してプロパティーの値を更新する必要があります。

次の例では、prop_op() は temperature プロパティーの要求を遮断しています。ドライバは、プロパティーが変わるたびに状態構造体で変数を更新します。ただし、プロパティーが更新されるのは、要求が出されたときだけです。その際、ドライバは ddi_prop_op() を使用してプロパティーリクエストを処理します。プロパティー要求がデバイスに固有のものでない場合、ドライバはその要求を遮断しません。このような状況は、dev パラメータの値が DDI_DEV_T_ANY (ワイルドカードのデバイス番号) である場合に示されます。

例 4–1 prop_op() ルーチン

```
static int
xx_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
    int flags, char *name, caddr_t valuep, int *lengthp)
{
        minor_t instance;
        struct xxstate *xsp;
        if (dev != DDI_DEV_T_ANY) {
                return (ddi_prop_op(dev, dip, prop_op, flags, name,
                    valuep, lengthp));
        }

        instance = getminor(dev);
        xsp = ddi_get_soft_state(statep, instance);
        if (xsp == NULL)
                return (DDI_PROP_NOTFOUND);
```

例 4–1 prop_op() ルーチン (続き)

```
        if (strcmp(name, "temperature") == 0) {
                ddi_prop_update_int(dev, dip, name, temperature);
        }

        /* other cases */
}
```

◆ ◆ ◆ 第 5 章

イベントの管理とタスクのキュー

ドライバは、イベントを使用して状態の変化に応答します。この章では、イベントに関する次の内容について説明します。

ドライバは、タスクキューを使用してタスク間のリソース依存性を管理します。この章では、タスクキューに関する次の内容について説明します。

イベントの管理

システムはしばしば、ユーザーアクションやシステム要求など、状態の変化に応答する必要があります。たとえばデバイスは、あるコンポーネントが過熱し始めたときに警告を発したり、DVDがドライブに挿入されたときにムービープレーヤーを起動したりする可能性があります。デバイスドライバは、イベントと呼ばれる特殊なメッセージを使用することで、状態の変化が発生したことをシステムに通知します。

イベントの概要

イベントとは、状態の変化が発生したことを示すためにデバイスドライバから対象のエンティティーに送信されるメッセージのことです。Oracle Solaris OS ではイベントはユーザー定義の名前-値ペア構造体として実装されており、この構造体は nvlist* 関数を使用して管理されます。nvlist_alloc(9F) のマニュアルページを参照してください。イベントはベンダー、クラス、およびサブクラスに基づいて編成さ

れます。たとえば、環境の状態を監視するためのクラスを定義します。環境用のクラスには、温度、ファンのステータス、および電力の変化を示すためのサブクラスが存在する場合があります。

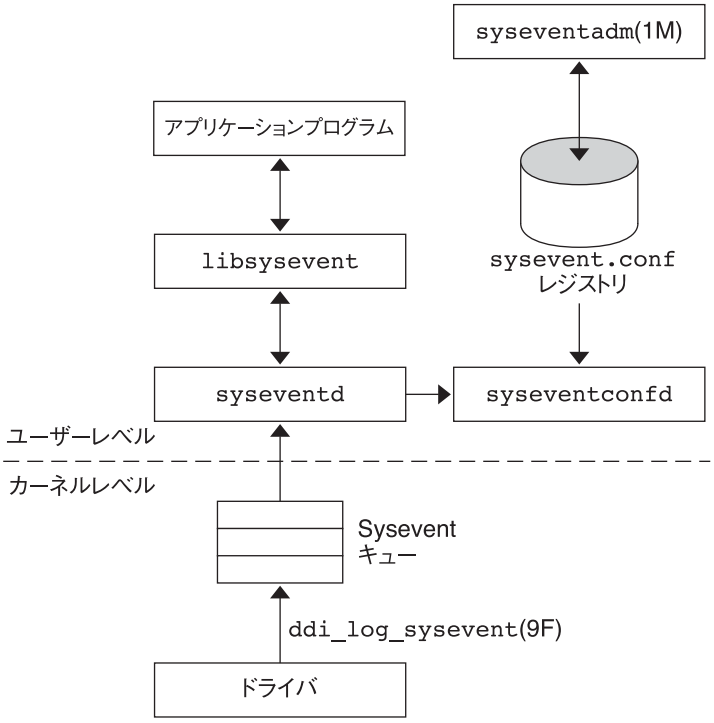
状態の変化が発生すると、デバイスはドライバに通知します。次に、ドライバは ddi_log_sysevent(9F) 関数を使用して、sysevent という名前のキュー内にこのイベントをロギングします。sysevent キューは、syseventd デーモンまたは syseventconfd のいずれかのデーモン経由でユーザーレベルにイベントを処理対象として渡します。これらのデーモンは、指定されたイベントの通知を受け取るように登録しているすべてのアプリケーションに、通知を送信します。

ユーザーレベルアプリケーションの設計者にとって、イベントを処理するには次の 2 つの方法があります。

- アプリケーションは、libsysevent(3LIB) 内のルーチンを使用することで、特定のイベントの発生時に通知を受け取ることができるように syseventd デーモンに登録できます。
- 開発者は、イベントに応答するための独立したユーザーレベルアプリケーションを記述できます。このタイプのアプリケーションは、syseventadm(1M) に登録する必要があります。syseventconfd は、指定されたイベントが発生すると、そのアプリケーションを実行し、適切にそのイベントに対処します。

このプロセスを示したのが次の図です。

図 5–1 イベントの plumb



ddi_log_sysevent() を使用したイベントのロギング

デバイスドライバは ddi_log_sysevent(9F) インタフェースを使用することで、イベントの生成とそのイベントのシステムへのロギングを行います。

ddi_log_sysevent() の構文

ddi_log_sysevent() で使用される構文は、次のとおりです。

```
int ddi_log_sysevent(dev_info_t *dip, char *vendor, char *class,
    char *subclass, nvlist_t *attr-list, sysevent_id_t *eidp, int sleep-flag);
```

各表記の意味は次のとおりです。

dip このドライバの *dev_info* ノードへのポインタ。

vendor ドライバのベンダーを定義する文字列へのポインタ。他社製ドライバでは、その会社の銘柄記号や、銘柄記号と同様の永続性のある識別子を使用します。Sun が提供するドライバでは DDI_VENDOR_SUNW が使用されます。

class イベントのクラスを定義する文字列へのポインタ。*class* はドライバに固有の値です。クラスの例には、あるデバイスに影響を与える一連の環境状態を表す文字列があります。この値は、イベントのコンシューマが認識できるものである必要があります。

subclass *class* 引数のサブセットを表すドライバ固有の文字列。たとえば、環境の状態を表すクラス内に、デバイスの温度を表すイベントサブクラスを追加できます。この値は、イベントのコンシューマが理解できるものである必要があります。

attr-list このイベントに関連する名前-値属性のリストを含む `nvlist_t` 構造体へのポインタ。名前-値属性はドライバごとに定義される値であり、デバイスの特定の属性や状態を表すことができます。

たとえば、CD-ROM と DVD の両方を読み取るデバイスを考えます。このデバイスには、名前が `disc_type` で値が `cd_rom`、`dvd` のいずれかに等しいような属性を割り当てることができます。

イベントのコンシューマは *class* や *subclass* の場合と同様に、名前-値ペアを解釈できる必要があります。

名前-値ペアや `nvlist_t` 構造体の詳細については、89 ページの「イベント属性の定義」を参照するほか、`nvlist_alloc(9F)` のマニュアルページも参照してください。

属性を持たないイベントでは、この引数を `NULL` に設定するべきです。

eidp `sysevent_id_t` 構造体のアドレス。`sysevent_id_t` 構造体を使用すると、イベントを一意に識別できます。`ddi_log_sysevent`(9F) から返されるこの構造体には、システムによって提供されるイベントシーケンス番号とタイムスタンプが含まれます。`sysevent_id_t` 構造体の詳細については、`ddi_log_sysevent(9F)` のマニュアルページを参照してください。

sleep-flag リソースが使用可能でない場合に呼び出し元がどのように対処するのかを示すフラグ。*sleep-flag* が `DDI_SLEEP` に設定されると、リソースが使用可能になるまでドライバがブロックされます。`DDI_NOSLEEP` の状態では割り当てがスリープしないため、処理の成功が保証されません。`DDI_ENOMEM` が返された場合、ドライバは処理をあとで再試行する必要があります。

`DDI_SLEEP` の状態でも、システムビジーや `sysevent`d デーモンの不応答、割り込みコンテキストでのイベントのロギング試行など、その他のエラーがこのインタフェースから返される可能性があります。

イベントロギング用のサンプルコード

デバイスドライバは、イベントのロギングを行うために次のタスクを実行します。

- nvlist_alloc(9F) を使用して属性リスト用のメモリーを割り当てる
- 名前-値ペアを属性リストに追加する
- ddi_log_sysevent(9F) 関数を使用して sysevent キューにイベントをロギングする
- 属性リストが不要になった時点で nvlist_free(9F) を呼び出す

次の例は、ddi_log_sysevent() の使用方法を示しています。

例 5–1 ddi_log_sysevent() の呼び出し

```
char *vendor_name = "DDI_VENDOR_JGJG"
char *my_class = "JGJG_event";
char *my_subclass = "JGJG_alert";
nvlist_t *nvl;
/* ... */
nvlist_alloc(&nvl, nvflag, kmflag);
/* ... */
(void) nvlist_add_byte_array(nvl, propname, (uchar_t *)propval, proplen + 1);
/* ... */
if (ddi_log_sysevent(dip, vendor_name, my_class,
    my_subclass, nvl, NULL, DDI_SLEEP)!= DDI_SUCCESS)
    cmn_err(CE_WARN, "error logging system event");
nvlist_free(nvl);
```

イベント属性の定義

イベント属性は名前-値ペアのリストとして定義します。Oracle Solaris DDI には、情報を名前-値ペアとして格納するためのルーチンや構造体が用意されています。名前-値ペアは、ドライバからは不透明な nvlist_t 構造体に保持されます。名前-値ペアの値は、ブール、int、バイト、文字列、nvlist のいずれか、またはこれらのデータ型の配列になります。int は、16 ビット、32 ビット、64 ビットのいずれかとして定義できるほか、符号付き、符号なしのいずれかにすることができます。

名前-値ペアのリストを作成する手順は、次のとおりです。

1. nvlist_alloc(9F) を使用して nvlist_t 構造体を作成します。

   nvlist_alloc () インタフェースが取る引数は次の 3 つです。

   - *nvlp* – nvlist_t 構造体へのポインタへのポインタ
   - *nvflag* – ペアの名前の一意性を示すフラグ。このフラグを NV_UNIQUE_NAME_TYPE に設定すると、新しいペアの名前と型に一致する既存のペアがすべて、リストから削除されます。このフラグを NV_UNIQUE_NAME に設定すると、型が何であっても、同じ名前を持つ既存のペアがすべて削除されます。NV_UNIQUE_NAME_TYPE を指定すると、型が異なるかぎり、名前が同じペアを複数個リストに含めることが可能となります。一方、NV_UNIQUE_NAME の場

合、リストに含めることのできるペア名のインスタンスは 1 つのみになります。このフラグを設定しなかった場合、一意性のチェックは行われず、リストのコンシューマが重複への対処を担当することになります。

- *kmflag* – カーネルメモリーの割り当てポリシーを示すフラグ。この引数を KM_SLEEP に設定すると、要求したメモリーが割り当て可能になるまでドライバがブロックされます。KM_SLEEP 割り当ての場合、スリープが発生する可能性がありますが、処理が成功することは保証されます。KM_NOSLEEP 割り当ての場合、スリープが発生しないことが保証されますが、使用可能なメモリーが現在存在しない場合には NULL が返される可能性があります。

2. nvlist に名前-値ペアを設定します。たとえば、文字列を追加するには、nvlist_add_string(9F) を使用します。32 ビット整数の配列を追加するには、nvlist_add_int32_array(9F) を使用します。nvlist_add_boolean(9F) のマニュアルページに、ペア追加用インタフェースの完全な一覧が含まれています。

リストの割り当てを解除するには、nvlist_free(9F) を使用します。

次のサンプルコードは、名前-値リストの作成方法を示したものです。

例 5–2 名前-値ペアリストの作成およびデータ設定

```
nvlist_t*
create_nvlist()
    {
    int err;
    char *str = "child";
    int32_t ints[] = {0, 1, 2};
    nvlist_t *nvl;

    err = nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0);    /* allocate list */
    if (err)
        return (NULL);
    if ((nvlist_add_string(nvl, "name", str) != 0) ||
        (nvlist_add_int32_array(nvl, "prop", ints, 3) != 0)) {
        nvlist_free(nvl);
        return (NULL);
    }
    return (nvl);
}
```

ドライバから nvlist の要素を取得するには、nvlist_lookup_int32_array(9F) など、その型の検索用関数を使用します。検索用関数は、検索対象となるペアの名前を引数として取ります。

注 – これらのインタフェースが動作するのは、nvlist_alloc(9F) の呼び出し時に NV_UNIQUE_NAME、NV_UNIQUE_NAME_TYPE のいずれかが指定された場合のみです。それ以外の場合は ENOTSUP が返されます。同じ名前の複数のペアをリストに含めることができないためです。

名前-値リストペアのリストは、連続するメモリー内に配置できます。このアプローチは、通知を受け取るように登録したエンティティーにリストを渡す場合に便利です。最初のステップは、リストに必要なメモリーブロックのサイズを、nvlist_size(9F) を使用して取得することです。次のステップは、nvlist_pack(9F) でリストをバッファーにパックすることです。バッファーの内容を受け取るコンシューマは、nvlist_unpack(9F) でバッファーを展開できます。

名前-値ペアを操作するための関数は、ユーザーレベル、カーネルレベルのどちらの開発者も使用できます。『SunOS リファレンスマニュアル 3: ライブラリインタフェースおよびヘッダー』と『SunOS リファレンスマニュアル 9: DDI/DKI カーネル関数』の両方に、これらの関数に対する同一のマニュアルページが含まれています。名前-値ペアを処理対象とする関数の一覧については、次の表を参照してください。

表 5–1 名前-値ペアを使用するための関数

マニュアルページ	目的/関数
nvlist_add_boolean(9F)	名前-値ペアをリストに追加します。関数は次のとおりです。 nvlist_add_boolean()、nvlist_add_boolean_value()、nvlist_add_byte()、nvlist_add_int8()、nvlist_add_uint8()、nvlist_add_int16()、nvlist_add_uint16()、nvlist_add_int32()、nvlist_add_uint32()、nvlist_add_int64()、nvlist_add_uint64()、nvlist_add_string()、nvlist_add_nvlist()、nvlist_add_nvpair()、nvlist_add_boolean_array()、nvlist_add_int8_array, nvlist_add_uint8_array()、nvlist_add_nvlist_array()、 nvlist_add_byte_array()、nvlist_add_int16_array()、nvlist_add_uint16_array()、 nvlist_add_int32_array()、nvlist_add_uint32_array ()、nvlist_add_int64_array()、 nvlist_add_uint64_array()、nvlist_add_string_array()
nvlist_alloc(9F)	名前-値リストのバッファーを操作します。関数は次のとおりです。 nvlist_alloc()、nvlist_free()、nvlist_size()、nvlist_pack()、nvlist_unpack()、nvlist_dup()、nvlist_merge()

表 5–1 名前-値ペアを使用するための関数 (続き)

マニュアルページ	目的/関数
nvlist_lookup_boolean(9F)	名前-値ペアを検索します。関数は次のとおりです。 nvlist_lookup_boolean()、nvlist_lookup_boolean_value()、nvlist_lookup_byte()、nvlist_lookup_int8()、nvlist_lookup_int16()、nvlist_lookup_int32()、nvlist_lookup_int64()、nvlist_lookup_uint8()、nvlist_lookup_uint16()、nvlist_lookup_uint32()、nvlist_lookup_uint64()、nvlist_lookup_string()、nvlist_lookup_nvlist()、nvlist_lookup_boolean_array、nvlist_lookup_byte_array()、nvlist_lookup_int8_array()、nvlist_lookup_int16_array()、nvlist_lookup_int32_array()、nvlist_lookup_int64_array()、nvlist_lookup_uint8_array()、nvlist_lookup_uint16_array()、nvlist_lookup_uint32_array()、 nvlist_lookup_uint64_array()、nvlist_lookup_string_array()、 nvlist_lookup_nvlist_array()、nvlist_lookup_pairs()
nvlist_next_nvpair(9F)	名前-値ペアのデータを取得します。関数は次のとおりです。 nvlist_next_nvpair()、nvpair_name()、nvpair_type()
nvlist_remove(9F)	名前-値ペアを削除します。関数は次のとおりです。 nv_remove()、nv_remove_all()

タスクのキュー

このセクションでは、タスクキューを使用して一部のタスクの処理を延期し、その実行を別のカーネルスレッドに委任する方法について説明します。

タスクキューの概要

カーネルプログラミングの一般的な操作は、タスクがあとで別のスレッドによって実行されるようにスケジュールすることです。次の例は、タスクを別のスレッドにあとで実行する理由をいくつか列挙したものです。

- 現在のコードパスはタイムクリティカルである。実行する追加タスクはタイムクリティカルでない。
- 追加タスクで、別のスレッドが現在保持しているロックを獲得しなければならない可能性がある。
- 現在のコンテキストではブロックできない。追加タスクは、メモリー待機などの理由でブロックしなければならない可能性がある。
- ある条件のためにコードパスが完了できない状態になっているが、現在のコードパスをスリープさせたり失敗させたりできない。条件の解消後に実行できるよう、現在のタスクをキューに入れる必要がある。

- 複数のタスクを並列して起動する必要がある。

これらの各ケースでは、別のコンテキスト内でタスクが実行されます。別のコンテキストとは通常、異なる一連のロックを保持し、優先順位もおそらく異なる別のカーネルスレッドのことです。タスクキューは、非同期タスクをスケジュールするための汎用カーネル API を提供します。

タスクキューとは、タスクリストを処理するためのスレッドを 1 つ以上備えたタスクリストのことです。タスクキューのサービススレッドが 1 つの場合、リストに追加された順番ですべてのタスクが実行されることが保証されます。タスクキューのサービススレッドが複数存在する場合、タスクの実行順番は不明になります。

注 – タスクキューのサービススレッドが複数存在する場合には、あるタスクの実行がほかのどのタスクの実行にも依存しないことを確認してください。タスク間に依存関係があると、デッドロックが発生する可能性があります。

タスクキューのインタフェース

次の DDI インタフェースはタスクキューを管理します。これらのインタフェースは `sys/sunddi.h` ヘッダーファイル内で定義されています。これらのインタフェースの詳細については、`taskq(9F)` のマニュアルページを参照してください。

`ddi_taskq_t`	不透明なハンドル
`TASKQ_DEFAULTPRI`	システムのデフォルト優先順位
`DDI_SLEEP`	メモリーのためにブロックできる
`DDI_NOSLEEP`	メモリーのためにブロックできない
`ddi_taskq_create()`	タスクキューを作成する
`ddi_taskq_destroy()`	タスクキューを破棄する
`ddi_taskq_dispatch()`	タスクキューにタスクを追加する
`ddi_taskq_wait()`	保留中のタスクが完了するまで待つ
`ddi_taskq_suspend()`	タスクキューを一時停止する
`ddi_taskq_suspended()`	タスクキューが一時停止されているかどうかをチェックする
`ddi_taskq_resume()`	一時停止されたタスクキューを再開する

タスクキューの監視

ドライバでの典型的な使用方法は、attach(9E) でタスクキューを作成することです。taskq_dispatch() 呼び出しの大部分は、割り込みコンテキストから行われます。

このセクションでは、タスクキューで消費されるシステムリソースを監視するために使用可能な 2 つの手法について説明します。タスクキューは、タスクキュースレッドによるシステム時間の使用量に関する統計情報をエクスポートします。さらにタスクキューは、DTrace SDT プローブを使用してタスクキューがタスクの実行を開始した時刻と終了した時刻を判定します。

タスクキューのカーネル統計カウンタ

すべてのタスクキューに一連の kstat カウンタが関連付けられます。次の kstat(1M) コマンドの出力を確認してください。

```
$ kstat -c taskq
module: unix                            instance: 0
name:   ata_nexus_enum_tq               class:    taskq
        crtime                          53.877907833
        executed                        0
        maxtasks                        0
        nactive                         1
        nalloc                          0
        priority                        60
        snaptime                        258059.249256749
        tasks                           0
        threads                         1
        totaltime                       0

module: unix                            instance: 0
name:   callout_taskq                   class:    taskq
        crtime                          0
        executed                        13956358
        maxtasks                        4
        nactive                         4
        nalloc                          0
        priority                        99
        snaptime                        258059.24981709
        tasks                           13956358
        threads                         2
        totaltime                       120247890619
```

上で示した kstat 出力には、次の情報が含まれています。

- タスクキューの名前とそのインスタンス番号
- スケジュールされたタスクの数 (tasks) と実行されたタスクの数 (executed)
- タスクキューを処理するカーネルスレッドの数 (threads) とその優先順位 (priority)
- すべてのタスクの処理に費やされた合計時間 (ナノ秒) (totaltime)

次の例は、kstat コマンドを使用してあるカウンタ (スケジュールされたタスクの数) が時間の経過とともに増加する様子を監視する方法について示したものです。

```
$ kstat -p unix:0:callout_taskq:tasks 1 5
unix:0:callout_taskq:tasks      13994642

unix:0:callout_taskq:tasks      13994711

unix:0:callout_taskq:tasks      13994784

unix:0:callout_taskq:tasks      13994855

unix:0:callout_taskq:tasks      13994926
```

タスクキューの DTrace SDT プローブ

タスクキューは便利な SDT プローブをいくつか提供します。このセクションで説明するプローブはすべて、次の 2 つの引数を持ちます。

- ddi_taskq_create() から返されるタスクキューポインタ
- taskq_ent_t 構造体へのポインタ。このポインタを D スクリプト内で使用することで関数や引数を抽出します。

これらのプローブを使用すると、個々のタスクキューやそれらを通じて実行される個々のタスクに関する高精度のタイミング情報を収集できます。たとえば、次のスクリプトは、タスクキュー経由でスケジュールされた関数を 10 秒に 1 度出力します。

```
# !/usr/sbin/dtrace -qs

sdt:genunix::taskq-enqueue
{
  this->tq  = (taskq_t *)arg0;
  this->tqe = (taskq_ent_t *) arg1;
  @[this->tq->tq_name,
    this->tq->tq_instance,
    this->tqe->tqent_func] = count();
}

tick-10s
{
  printa ("%s(%d): %a called %@d times\n", @);
  trunc(@);
}
```

特定のマシン上で上の D スクリプトを実行すると、次のような出力が生成されます。

```
callout_taskq(1): genunix‘callout_execute called 51 times
callout_taskq(0): genunix‘callout_execute called 701 times
kmem_taskq(0): genunix‘kmem_update_timeout called 1 times
kmem_taskq(0): genunix‘kmem_hash_rescale called 4 times
callout_taskq(1): genunix‘callout_execute called 40 times
```

```
USB_hid_81_pipehndl_tq_1(14): usba‘hcdi_cb_thread called 256 times
callout_taskq(0): genunix‘callout_execute called 702 times
kmem_taskq(0): genunix‘kmem_update_timeout called 1 times
kmem_taskq(0): genunix‘kmem_hash_rescale called 4 times
callout_taskq(1): genunix‘callout_execute called 28 times
USB_hid_81_pipehndl_tq_1(14): usba‘hcdi_cb_thread called 228 times
callout_taskq(0): genunix‘callout_execute called 706 times
callout_taskq(1): genunix‘callout_execute called 24 times
USB_hid_81_pipehndl_tq_1(14): usba‘hcdi_cb_thread called 141 times
callout_taskq(0): genunix‘callout_execute called 708 times
```

◆ ◆ ◆ 第 6 章

ドライバの自動構成

自動構成とは、ドライバがコードと静的データをメモリー内にロードすることを示します。その後、この情報がシステムに登録されます。また、自動構成では、そのドライバによって制御される個々のデバイスインスタンスの接続も行われます。

この章では、次の内容について説明します。

ドライバのロードとアンロード

システムは、ドライバのバイナリモジュールを、自動構成のためのカーネルモジュールディレクトリの `drv` サブディレクトリからロードします。545 ページの「モジュールディレクトリへのドライバのコピー」を参照してください。

モジュールが、解決されたすべてのシンボルとともにメモリーに読み取られたあと、システムはそのモジュールの `_init(9E)` エントリポイントを呼び出します。`_init()` 関数は、そのモジュールを実際にロードする `mod_install(9F)` を呼び出します。

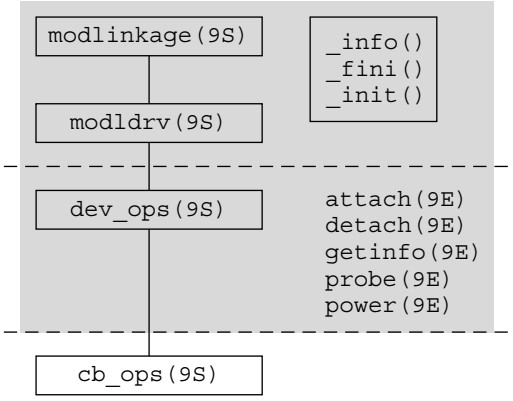
注 – `mod_install()` の呼び出し中、ほかのスレッドは、`mod_install()` が呼び出されるとすぐに `attach(9E)` を呼び出すことができます。プログラミングの観点からは、`mod_install()` が呼び出される前に、`_init()` によるすべての初期化が実行される必要があります。`mod_install()` が失敗した (つまり、ゼロ以外の値が返された) 場合は、その初期化をバックアウトする必要があります。

_init() が正常に完了すると、そのドライバはシステムに正常に登録されています。この時点では、そのドライバはどのデバイスもアクティブに管理していません。デバイス管理は、デバイス構成の一部として実行されます。

システムは、システムメモリーを節約するため、またはユーザーからの明示的な要求によってドライバのバイナリモジュールをアンロードします。ドライバのコードとデータをメモリーから削除する前に、そのドライバの _fini(9E) エントリポイントが呼び出されます。ドライバがアンロードされるのは、_fini() が成功を返した場合だけです。

次の図は、デバイスドライバの構造的な概要を示しています。陰付きの領域は、ドライバのデータ構造体とエントリポイントを強調しています。陰付きの領域の上半分には、ドライバのロードとアンロードをサポートするデータ構造体とエントリポイントが含まれています。下半分は、ドライバの構成に関連しています。

図6–1 モジュールのロードと自動構成のエントリポイント



ドライバに必要なデータ構造体

自動構成をサポートするには、ドライバで次のデータ構造体を静的に初期化する必要があります。

- modlinkage(9S)
- modldrv(9S)
- dev_ops(9S)
- cb_ops(9S)

ドライバは、図 5-1 のデータ構造体に依存しています。これらの構造体を正しく提供し、初期化する必要があります。これらのデータ構造体がない場合、ドライバが正しくロードされない可能性があります。その結果、必要なルーチンがロードされない可能性があります。ある操作がドライバによってサポートされていない場合

は、nodev(9F) ルーチンのアドレスをプレースホルダとして使用できます。場合によっては、ドライバがエントリポイントをサポートするため、成功または失敗を返すだけで済むことがあります。その場合は、nulldev(9F) ルーチンのアドレスを使用できます。

注 – これらの構造体は、コンパイル時に初期化してください。ドライバは、それ以外のときにこれらの構造体へのアクセスや変更を行うべきではありません。

---

modlinkage 構造体

```
static struct modlinkage xxmodlinkage = {
    MODREV_1,       /* ml_rev */
    &xxmodldrv,     /* ml_linkage[] */
    NULL            /* NULL termination */
};
```

最初のフィールドは、サブシステムをロードするモジュールのバージョン番号です。このフィールドは MODREV_1 にします。2 番目のフィールドは、次に定義されているドライバの modldrv 構造体を指しています。この構造体の最後の要素は、常に NULL にします。

modldrv 構造体

```
static struct modldrv xxmodldrv = {
    &mod_driverops,           /* drv_modops */
    "generic driver v1.1",    /* drv_linkinfo */
    &xx_dev_ops               /* drv_dev_ops */
};
```

この構造体では、モジュールについてさらに詳細に記述します。最初のフィールドでは、モジュールのインストールに関連した情報を提供します。このフィールドは、ドライバモジュールの &mod_driverops に設定します。2 番目のフィールドは、modinfo(1M) で表示される文字列です。2 番目のフィールドには、ドライバのバイナリを生成したソースコードのバージョンを識別するために十分な情報を含めます。最後のフィールドは、次のセクションで定義されるドライバの dev_ops 構造体を指しています。

dev_ops 構造体

```
static struct dev_ops xx_dev_ops = {
    DEVO_REV,       /* devo_rev */
    0,              /* devo_refcnt  */
    xxgetinfo,      /* devo_getinfo: getinfo(9E) */
    nulldev,        /* devo_identify: identify(9E) */
```

```
    xxprobe,        /* devo_probe: probe(9E) */
    xxattach,       /* devo_attach: attach(9E) */
    xxdetach,       /* devo_detach: detach(9E) */
    nodev,          /* devo_reset */
    &xx_cb_ops,     /* devo_cb_ops */
    NULL,           /* devo_bus_ops */
    &xxpower        /* devo_power: power(9E) */
};
```

dev_ops(9S) 構造体は、カーネルがデバイスドライバの自動構成エントリポイントを見つけることができるようにします。devo_rev フィールドは、構造体のリビジョン番号を識別します。このフィールドは DEVO_REV に設定する必要があります。devo_refcnt フィールドは 0 に初期化する必要があります。次の場合を除き、関数のアドレスフィールドには該当するドライバエントリポイントのアドレスを入力します。

- devo_identify フィールドを nulldev(9F) に設定します。identify() エントリポイントは廃止されています。
- probe(9E) ルーチンが必要ない場合は、devo_probe フィールドを nulldev(9F) に設定します。
- devo_reset フィールドを nodev(9F) に設定します。nodev() 関数は ENXIO を返します。
- power() ルーチンが必要ない場合は、devo_power フィールドを NULL に設定します。電源管理機能を提供するデバイスのドライバには、power(9E) エントリポイントが必要です。第 12 章「電源管理」を参照してください。

devo_cb_ops メンバーには、cb_ops(9S) 構造体のアドレスを含めます。devo_bus_ops フィールドは NULL に設定する必要があります。

cb_ops 構造体

```
static struct cb_ops xx_cb_ops = {
    xxopen,         /* open(9E) */
    xxclose,        /* close(9E) */
    xxstrategy,     /* strategy(9E) */
    xxprint,        /* print(9E) */
    xxdump,         /* dump(9E) */
    xxread,         /* read(9E) */
    xxwrite,        /* write(9E) */
    xxioctl,        /* ioctl(9E) */
    xxdevmap,       /* devmap(9E) */
    nodev,          /* mmap(9E) */
    xxsegmap,       /* segmap(9E) */
    xxchpoll,       /* chpoll(9E) */
    xxprop_op,      /* prop_op(9E) */
    NULL,           /* streamtab(9S) */
    D_MP | D_64BIT, /* cb_flag */
    CB_REV,         /* cb_rev */
    xxaread,        /* aread(9E) */
    xxawrite        /* awrite(9E) */
};
```

cb_ops(9S) 構造体には、デバイスドライバの文字操作とブロック操作のためのエントリポイントが含まれています。ドライバでサポートされていないエントリポイントはすべて、nodev(9F) に初期化します。たとえば、文字デバイスドライバは、ブロックのみのフィールド (cb_stategy など) をすべて nodev(9F)に設定します。mmap(9E) エントリポイントは、以前のリリースとの互換性のために保持されていることに注意してください。ドライバは、devmap(9E) エントリポイントをデバイスメモリーのマッピングのために使用します。devmap(9E) がサポートされている場合は、mmap(9E) を nodev(9F) に設定します。

streamtab フィールドは、ドライバが STREAMS ベースかどうかを示します。STREAMS ベースであるのは、第 19 章「ネットワークデバイスのドライバ」で説明されているネットワークデバイスドライバのみです。STREAMS ベースでないドライバではすべて、streamtab フィールドを NULL に設定する必要があります。

cb_flag メンバーには、次のフラグが含まれています。

- D_MP フラグは、このドライバがマルチスレッド化に対して安全であることを示します。Oracle Solaris OS はスレッドセーフなドライバのみをサポートしているため、D_MP を設定する必要があります。
- D_64BIT フラグを指定すると、ドライバは uio(9S) 構造体の uio_loffset フィールドを使用します。64 ビットオフセットを正しく処理するため、ドライバは cb_flag フィールドに D_64BIT フラグを設定します。
- D_DEVMAP フラグは、devmap(9E) エントリポイントをサポートしています。devmap(9E) については、第 10 章「デバイスメモリーおよびカーネルメモリーのマッピング」を参照してください。

cb_rev は cb_ops 構造体のリビジョン番号です。このフィールドは CB_REV に設定する必要があります。

ロード可能なドライバインタフェース

デバイスドライバは、動的にロード可能である必要があります。ドライバは、メモリーリソースの節約に役立つように、アンロード可能にもなっています。また、アンロード可能なドライバは、テスト、デバッグ、パッチ適用がより容易でもあります。

ドライバのロードとアンロードをサポートするには、各デバイスドライバが _init(9E)、_fini(9E)、および _info(9E) エントリポイントを実装する必要があります。次の例は、ロード可能なドライバインタフェースの標準的な実装を示しています。

例 6–1 ロード可能なインタフェースのセクション

```
static void *statep;                /* for soft state routines */
static struct cb_ops xx_cb_ops;     /* forward reference */
static struct dev_ops xx_ops = {
```

例 6–1 ロード可能なインタフェースのセクション (続き)

```
    DEVO_REV,
    0,
    xxgetinfo,
    nulldev,
    xxprobe,
    xxattach,
    xxdetach,
    xxreset,
    nodev,
    &xx_cb_ops,
    NULL,
    xxpower
};

static struct modldrv modldrv = {
    &mod_driverops,
    "xx driver v1.0",
    &xx_ops
};

static struct modlinkage modlinkage = {
    MODREV_1,
    &modldrv,
    NULL
};

int
_init(void)
{
    int error;
    ddi_soft_state_init(&statep, sizeof (struct xxstate),
        estimated_number_of_instances);
    /* further per-module initialization if necessary */
    error = mod_install(&modlinkage);
    if (error != 0) {
        /* undo any per-module initialization done earlier */
        ddi_soft_state_fini(&statep);
    }
    return (error);
}

int
_fini(void)
{
    int error;
    error = mod_remove(&modlinkage);
    if (error == 0) {
        /* release per-module resources if any were allocated */
        ddi_soft_state_fini(&statep);
    }
    return (error);
}

int
_info(struct modinfo *modinfop)
{
```

例 6–1 ロード可能なインタフェースのセクション (続き)

```
    return (mod_info(&modlinkage, modinfop));
}
```

_init() の例

次の例は、標準的な _init(9E) インタフェースを示しています。

例 6–2 _init() 関数

```
static void *xxstatep;
int
_init(void)
{
    int error;
    const int max_instance = 20;    /* estimated max device instances */

    ddi_soft_state_init(&xxstatep, sizeof (struct xxstate), max_instance);
    error = mod_install(&xxmodlinkage);
    if (error != 0) {
        /*
         * Cleanup after a failure
         */
        ddi_soft_state_fini(&xxstatep);
    }
    return (error);
}
```

ドライバは、1 回限りのリソース割り当てまたはデータ初期化を _init() でのドライバのロード中に実行します。たとえば、ドライバは、そのドライバに対してグローバルな mutex をすべてこのルーチン内で初期化します。ただし、ドライバは _init(9E) を使用して、デバイスの特定のインスタンスと関係があるものを割り当てたり、初期化したりしません。インスタンスごとの初期化は、attach(9E) で実行する必要があります。たとえば、プリンタのドライバが一度に複数のプリンタを処理できる場合、そのドライバは、各プリンタインスタンスに固有のリソースを attach() で割り当てます。

注 – _init(9E) が mod_install(9F) を呼び出したあとは、システムがそのデータ構造体をコピーまたは変更する可能性があるため、ドライバは modlinkage(9S) 構造体に接続されたデータ構造体を変更しません。

_fini() の例

次の例は、_fini() ルーチンを示しています。

```
int
_fini(void)
{
        int error;
    error = mod_remove(&modlinkage);
    if (error != 0) {
        return (error);
    }
    /*
     * Cleanup resources allocated in _init()
     */
    ddi_soft_state_fini(&xxstatep);
    return (0);
}
```

同様に、ドライバは `_fini()` で、`_init()` で割り当てられたリソースをすべて解放します。ドライバは、システムモジュールリストから自身を削除する必要があります。

注 – ドライバがハードウェアインスタンスに接続されているときに `_fini()` が呼び出される可能性があります。この場合は、`mod_remove`(9F) が失敗を返します。そのため、`mod_remove()` が成功を返すまで、ドライバリソースを解放しません。

`_info()` の例

次の例は、`_info`(9E) ルーチンを示しています。

```
int
_info(struct modinfo *modinfop)
{
    return (mod_info(&xxmodlinkage, modinfop));
}
```

ドライバは、モジュール情報を返すために呼び出されます。このエントリポイントは、上に示すように実装します。

デバイス構成の概念

カーネルデバイスツリー内のノードごとに、システムは、ノード名と `compatible` プロパティーに基づいてそのノードのドライバを選択します (65 ページの「ドライバのデバイスへのバインド」を参照)。複数のデバイスノードに同じドライバがバインドされる可能性があります。ドライバは、システムによって割り当てられたインスタンス番号により、異なるノードを区別できます。

デバイスノードに対してドライバが選択されたあと、そのデバイスがシステム上に存在するかどうかを判定するために、そのドライバの `probe`(9E) エントリポイントが呼び出されます。`probe()` が成功した場合は、デバイスを設定および管理するため

に、そのドライバの attach(9E) エントリポイントが呼び出されます。デバイスを開くことができるのは、attach() が成功を返した場合のみです (109 ページの「attach() エントリポイント」を参照)。

デバイスは、システムメモリーリソースを節約するため、またはシステムの実行中にそのデバイスを削除できるようにするために、構成解除される可能性があります。デバイスを構成解除できるようにするために、システムはまず、そのデバイスインスタンスが参照されているかどうかをチェックします。このチェックでは、そのドライバのみが認識している情報を取得するために、ドライバの getinfo(9E) エントリポイントを呼び出します (116 ページの「getinfo() エントリポイント」を参照)。そのデバイスインスタンスが参照されていない場合は、そのデバイスを構成解除するために、ドライバの detach(9E) ルーチンが呼び出されます (115 ページの「detach() エントリポイント」を参照)。

まとめると、各ドライバは、デバイス構成のためにカーネルによって使用される次のエントリポイントを定義する必要があります。

- probe(9E)
- attach(9E)
- detach(9E)
- getinfo(9E)

attach()、detach()、および getinfo() は必須であることに注意してください。probe() は、自身を識別できないデバイスに対してのみ必要です。自身を識別できるデバイスの場合は、明示的な probe() ルーチンを提供するか、または probe() エントリポイントの dev_ops 構造体で nulldev(9F) を指定できます。

デバイスインスタンスとインスタンス番号

システムは、各デバイスにインスタンス番号を割り当てます。ドライバは、特定のデバイスに割り当てられたインスタンス番号の値を確実には予測できない可能性があります。ドライバは ddi_get_instance(9F) を呼び出すことによって、割り当てられた特定のインスタンス番号を取得します。

インスタンス番号は、デバイスに対するシステムの概念を表します。特定のドライバに対する各 dev_info (つまり、デバイスツリー内の各ノード) には、カーネルによってインスタンス番号が割り当てられます。さらに、インスタンス番号は、特定の物理デバイスに固有のデータのインデックスを作成するために便利なメカニズムを提供します。インスタンス番号のもっとも一般的な使用法は、インスタンス番号を使用して特定の物理デバイスのソフト状態データを取得する、ddi_get_soft_state(9F) です。

注意 – 疑似デバイス (つまり、疑似ネクサスの子) の場合、インスタンス番号は、instance プロパティーを使用して driver.conf(4) ファイルで定義されます。driver.conf ファイルに instance プロパティーが含まれていない場合の動作は未定義です。ハードウェアデバイスノードの場合、システムは、そのデバイスが最初に OS によって認識されたときにインスタンス番号を割り当てます。インスタンス番号は、システムのリブートや OS のアップグレードのあとも持続されます。

マイナーノードとマイナー番号

ドライバは、自身のマイナー番号名前空間の管理を担当します。たとえば、sd ドライバはディスクごとに、8 文字のマイナーノードと 8 ブロックのマイナーノードをファイルシステムにエクスポートする必要があります。各マイナーノードは、ディスクのある部分に対するブロックインタフェースまたは文字インタフェースのどちらかを表します。getinfo(9E) エントリポイントは、マイナー番号からデバイスインスタンスへのマッピングについてシステムに通知します (116 ページの「getinfo() エントリポイント」を参照)。

probe() エントリポイント

自身を識別できないデバイスの場合は、probe(9E) エントリポイントで、そのハードウェアデバイスがシステム上に存在するかどうかを判定します。

probe() でデバイスのインスタンスが存在するかどうかを判定するには、probe() は、一般に attach(9E) によっても実行される多くのタスクを実行する必要があります。特に、probe() によるデバイスレジスタのマッピングが必要になることがあります。

デバイスレジスタのプローブはデバイス固有の操作ですドライバは多くの場合、ハードウェアが実際に存在することを確認するために、そのハードウェアの一連のテストを実行する必要があります。このテストの条件は、デバイスの誤った識別を回避できるほど十分に厳格である必要があります。たとえば、別のデバイスが予期したデバイスと同様に動作しているように見えるため、そのデバイスが実際には使用できないのに、存在しているように見える可能性があります。

このテストは次のフラグを返します。

- プローブが成功した場合は、DDI_PROBE_SUCCESS
- プローブが失敗した場合は、DDI_PROBE_FAILURE
- プローブは失敗したが、attach(9E) を引き続き呼び出す必要がある場合は、DDI_PROBE_DONTCARE
- 現在はインスタンスが存在しないが、将来存在する可能性がある場合は、DDI_PROBE_PARTIAL

特定のデバイスインスタンスの場合は、そのデバイスに対して probe(9E) が少なくとも 1 回成功するまで attach(9E) は呼び出されません。

probe() は複数回呼び出される可能性があるため、probe(9E) は、probe() が割り当てたすべてのリソースを解放する必要があります。ただし、probe(9E) が成功した場合でも、必ずしも attach(9E) が呼び出されるわけではありません。

ドライバの probe(9E) ルーチンで ddi_dev_is_sid(9F) を使用すると、デバイスが自身を識別できるかどうかを判定できます。ddi_dev_is_sid() は、同じデバイスの自身を識別できるバージョンおよび自身を識別できないバージョンのために記述されたドライバで有効です。

次の例は、サンプルの probe() ルーチンです。

例 6–3 probe(9E) ルーチン

```
static int
xxprobe(dev_info_t *dip)
{
    ddi_acc_handle_t dev_hdl;
    ddi_device_acc_attr_t dev_attr;
    Pio_csr *csrp;
    uint8_t csrval;

    /*
     * if the device is self identifying, no need to probe
     */
    if (ddi_dev_is_sid(dip) == DDI_SUCCESS)
    return (DDI_PROBE_DONTCARE);

    /*
     * Initalize the device access attributes and map in
     * the devices CSR register (register 0)
     */
    dev_attr.devacc_attr_version = DDI_DEVICE_ATTR_V0;
    dev_attr.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;
    dev_attr.devacc_attr_dataorder = DDI_STRICTORDER_ACC;

    if (ddi_regs_map_setup(dip, 0, (caddr_t *)&csrp, 0, sizeof (Pio_csr),
    &dev_attr, &dev_hdl) != DDI_SUCCESS)
    return (DDI_PROBE_FAILURE);

    /*
     * Reset the device
     * Once the reset completes the CSR should read back
     * (PIO_DEV_READY | PIO_IDLE_INTR)
     */
    ddi_put8(dev_hdl, csrp, PIO_RESET);
    csrval = ddi_get8(dev_hdl, csrp);

    /*
     * tear down the mappings and return probe success/failure
     */
    ddi_regs_map_free(&dev_hdl);
    if ((csrval & 0xff) == (PIO_DEV_READY | PIO_IDLE_INTR))
```

例 6–3 probe(9E) ルーチン (続き)

```
    return (DDI_PROBE_SUCCESS);
    else
    return (DDI_PROBE_FAILURE);
}
```

ドライバの probe(9E) ルーチンが呼び出されたとき、そのドライバは、プローブされているデバイスがバス上に存在するかどうかを認識していません。そのため、ドライバは、存在しないデバイスのデバイスレジスタへのアクセスを試みる可能性があります。その結果、一部のバスではバス障害が発生することがあります。

次の例は、ddi_poke8(9F) を使用してデバイスの存在をチェックする probe(9E) ルーチンを示しています。ddi_poke8() は、必要な場合はこのプロセスを支援するために親ネクサスドライバを使用して、指定された仮想アドレスへの値の書き込みを慎重に試みます。アドレスが有効でないか、またはエラーを発生させずに値を書き込むことができない場合は、エラーコードが返されます。ddi_peek(9F)も参照してください。

この例では、デバイスレジスタをマッピングするために ddi_regs_map_setup(9F) が使用されています。

例 6–4 ddi_poke8(9F) を使用した probe(9E) ルーチン

```
static int
xxprobe(dev_info_t *dip)
{
    ddi_acc_handle_t dev_hdl;
    ddi_device_acc_attr_t dev_attr;
    Pio_csr *csrp;
    uint8_t csrval;

    /*
     * if the device is self-identifying, no need to probe
     */
    if (ddi_dev_is_sid(dip) == DDI_SUCCESS)
    return (DDI_PROBE_DONTCARE);

    /*
     * Initialize the device access attrributes and map in
     * the device's CSR register (register 0)
     */
    dev_attr.devacc_attr_version - DDI_DEVICE_ATTR_V0;
    dev_attr.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;
    dev_attr.devacc_attr_dataorder = DDI_STRICTORDER_ACC;

    if (ddi_regs_map_setup(dip, 0, (caddr_t *)&csrp, 0, sizeof (Pio_csr),
    &dev_attr, &dev_hdl) != DDI_SUCCESS)
    return (DDI_PROBE_FAILURE);

    /*
     * The bus can generate a fault when probing for devices that
     * do not exist.  Use ddi_poke8(9f) to handle any faults that
```

例 6–4 ddi_poke8(9F) を使用した probe(9E) ルーチン (続き)

```
     * might occur.
     *
     * Reset the device.  Once the reset completes the CSR should read
     * back (PIO_DEV_READY | PIO_IDLE_INTR)
     */
    if (ddi_poke8(dip, csrp, PIO_RESET) != DDI_SUCCESS) {
    ddi_regs_map_free(&dev_hdl);
    return (DDI_FAILURE);

    csrval = ddi_get8(dev_hdl, csrp);
    /*
     * tear down the mappings and return probe success/failure
     */
    ddi_regs_map_free(&dev_hdl);
    if ((csrval & 0xff) == (PIO_DEV_READY | PIO_IDLE_INTR))
    return (DDI_PROBE_SUCCESS);
    else
    return (DDI_PROBE_FAILURE);
}
```

attach() エントリポイント

カーネルは、デバイスのインスタンスを接続するため、または電源管理フレームワークによって中断または停止されたデバイスのインスタンスの操作を再開するために、ドライバの attach(9E) エントリポイントを呼び出します。このセクションでは、デバイスインスタンスを接続する操作についてのみ説明します。電源管理については、第 12 章「電源管理」で説明します。

ドライバの attach(9E) エントリポイントは、そのドライバにバインドされたデバイスの各インスタンスを接続するために呼び出されます。このエントリポイントは、接続するデバイスノードのインスタンスと、attach(9E) に対する cmd 引数として指定された DDI_ATTACH を使用して呼び出されます。attach エントリポイントには通常、次のタイプの処理が含まれます。

- デバイスインスタンスへのソフト状態構造体の割り当て
- インスタンスごとの mutex の初期化
- 条件変数の初期化
- デバイスの割り込みの登録
- デバイスインスタンスのレジスタとメモリーのマッピング
- デバイスインスタンスのマイナーデバイスノードの作成
- デバイスインスタンスが接続されたことの報告

ドライバのソフト状態管理

デバイスドライバ作成者が状態構造体を割り当てるのを支援するために、Oracle Solaris DDI/DKI には、ソフトウェア状態管理ルーチンと呼ばれる一連のメモリー管理ルーチン (ソフト状態ルーチンとも呼ばれる) が用意されています。これらの

ルーチンは、指定されたサイズのメモリー項目の動的な割り当て、取得、および破棄を行い、リスト管理の詳細を非表示にします。目的のメモリー項目は、インスタンス番号で識別されます。この番号は通常、システムによって割り当てられるインスタンス番号です。

ドライバは通常、ddi_soft_state_zalloc(9F) を呼び出し、デバイスのインスタンス番号を渡すことによって、そのドライバに接続されたデバイスインスタンスごとにソフト状態構造体を割り当てます。2 つのデバイスノードが同じインスタンス番号を持つことはできないため、指定されたインスタンス番号に対する割り当てがすでに存在する場合は、ddi_soft_state_zalloc(9F) が失敗します。

ドライバの文字またはブロックエントリポイント (cb_ops(9S)) は、まずエントリポイント関数に渡された dev_t 引数からデバイスのインスタンス番号をデコードすることによって、特定のソフト状態構造体を参照します。次に、ドライバが ddi_get_soft_state(9F) を呼び出して、ドライバごとのソフト状態リストと、派生したインスタンス番号を渡します。NULL の戻り値は、そのデバイスが事実上存在しないため、ドライバが該当するコードを返すことを示します。

インスタンス番号とデバイス番号 (dev_t) がどのように関連するかの詳細については、110 ページの「マイナーデバイスノードの作成」を参照してください。

ロック変数と条件変数の初期化

ドライバは、接続中にインスタンスごとのロック変数と条件変数をすべて初期化します。ドライバの割り込みハンドラによって取得されたロックはすべて、いずれかの割り込みハンドラを追加する前に初期化する必要があります。ロックの初期化と使用法の説明については、第 3 章「マルチスレッド」を参照してください。割り込みハンドラとロックの問題の説明については、第 8 章「割り込みハンドラ」を参照してください。

マイナーデバイスノードの作成

接続プロセスの重要な部分として、デバイスインスタンスに対するマイナーノードの作成があります。マイナーノードには、デバイスと DDI フレームワークによってエクスポートされた情報が含まれています。システムはこの情報を使用して、/devices の下にマイナーノードのための特殊ファイルを作成します。

マイナーノードは、ドライバが ddi_create_minor_node(9F) を呼び出したときに作成されます。ドライバは、マイナー番号、マイナー名、マイナーノードタイプ、およびそのマイナーノードがブロックデバイスまたは文字デバイスのどちらを表すかを指定します。

ドライバは、1 つのデバイスに対して任意の数のマイナーノードを作成できます。Oracle Solaris DDI/DKI は、特定のクラスのデバイスには特定の形式で作成されたマイナーノードがあるものと想定します。たとえば、ディスクドライバでは、接続された物理ディスクインスタンスごとに 16 のマイナーノードが作成されると予期し

ています。a - h のブロックデバイスインタフェースを表す 8 つのマイナーノードに加え、a,raw - h,raw の文字デバイスインタフェースのための 8 つのマイナーノードが追加で作成されます。

ddi_create_minor_node(9F) に渡されるマイナー番号は、完全にドライバによって定義されます。マイナー番号は通常、マイナーノード識別子を含む、デバイスのインスタンス番号のエンコーディングです。前の例でドライバは、デバイスのインスタンス番号を 3 ビット左にシフトし、マイナーノードインデックスにその結果の OR を使用することによって各マイナーノードのマイナー番号を作成します。マイナーノードインデックスの値は 0 - 7 です。マイナーノード a と a,raw が同じマイナー番号を共有することに注意してください。これらのマイナーノードは、ddi_create_minor_node() に渡される *spec_type* 引数によって区別されます。

ddi_create_minor_node(9F) に渡されるマイナーノードタイプによって、ディスク、テープ、ネットワークインタフェース、フレームバッファーなどの、デバイスのタイプが分類されます。

次の表に、作成される可能性のあるノードのタイプを示します。

表 6–1 可能性のあるノードタイプ

定数	説明
DDI_NT_SERIAL	シリアルポート
DDI_NT_SERIAL_DO	ダイヤルアウトポート
DDI_NT_BLOCK	ハードディスク
DDI_NT_BLOCK_CHAN	チャネルまたはターゲット番号を持つハードディスク
DDI_NT_CD	ROM ドライブ (CD-ROM)
DDI_NT_CD_CHAN	チャネルまたはターゲット番号を持つ ROM ドライブ
DDI_NT_FD	フロッピーディスク
DDI_NT_TAPE	テープドライブ
DDI_NT_NET	ネットワークデバイス
DDI_NT_DISPLAY	ディスプレイデバイス
DDI_NT_MOUSE	マウス
DDI_NT_KEYBOARD	キーボード
DDI_NT_AUDIO	オーディオデバイス
DDI_PSEUDO	一般的な疑似デバイス

ノードタイプ DDI_NT_BLOCK、DDI_NT_BLOCK_CHAN、DDI_NT_CD、および DDI_NT_CD_CHAN を指定すると、devfsadm(1M) はデバイスインスタンスをディスクとして識別し、/dev/dsk または /dev/rdsk ディレクトリ内に名前を作成します。

ノードタイプ DDI_NT_TAPE を指定すると、devfsadm(1M) はデバイスインスタンスをテープとして識別し、/dev/rmt ディレクトリ内に名前を作成します。

ノードタイプ DDI_NT_SERIAL および DDI_NT_SERIAL_DO を指定すると、devfsadm(1M) は次のアクションを実行します。

- デバイスインスタンスをシリアルポートとして識別する
- /dev/term ディレクトリ内に名前を作成する
- /etc/inittab ファイルにエントリを追加する

ベンダーから提供された文字列には、その文字列を一意のものにするために、名前や銘柄記号などの識別値が含まれているはずです。この文字列を devfsadm(1M) や devlinks.tab ファイル (devlinks(1M) のマニュアルページを参照) と組み合わせて使用することにより、/dev 内に論理名を作成できます。

遅延接続

対応するインスタンスに対して attach(9E) が成功する前に、マイナーデバイスに対して open(9E) が呼び出される可能性があります。その場合、open() は ENXIO を返す必要があります。これにより、システムはそのデバイスの接続を試みます。attach() が成功した場合は、open() が自動的に再試行されます。

例 6–5　標準的な attach() エントリポイント

```
/*
 * Attach an instance of the driver.  We take all the knowledge we
 * have about our board and check it against what has been filled in
 * for us from our FCode or from our driver.conf(4) file.
 */
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int instance;
    Pio *pio_p;
    ddi_device_acc_attr_t   da_attr;
    static int pio_validate_device(dev_info_t *);

    switch (cmd) {
    case DDI_ATTACH:

    /*
     * first validate the device conforms to a configuration this driver
     * supports
     */
    if (pio_validate_device(dip) == 0)
        return (DDI_FAILURE);

    /*
```

例 6–5 標準的な attach() エントリポイント (続き)

```
     * Allocate a soft state structure for this device instance
     * Store a pointer to the device node in our soft state structure
     * and a reference to the soft state structure in the device
     * node.
     */
    instance = ddi_get_instance(dip);
    if (ddi_soft_state_zalloc(pio_softstate, instance) != 0)
        return (DDI_FAILURE);
    pio_p = ddi_get_soft_state(pio_softstate, instance);
    ddi_set_driver_private(dip, (caddr_t)pio_p);
    pio_p->dip = dip;

    /*
     * Before adding the interrupt, get the interrupt block
     * cookie associated with the interrupt specification to
     * initialize the mutex used by the interrupt handler.
     */
    if (ddi_get_iblock_cookie(dip, 0, &pio_p->iblock_cookie) !=
      DDI_SUCCESS) {
        ddi_soft_state_free(pio_softstate, instance);
        return (DDI_FAILURE);
    }

    mutex_init(&pio_p->mutex, NULL, MUTEX_DRIVER, pio_p->iblock_cookie);

    /*
     * Now that the mutex is initialized, add the interrupt itself.
     */
    if (ddi_add_intr(dip, 0, NULL, NULL, pio_intr, (caddr_t)instance) !=
      DDI_SUCCESS) {
        mutex_destroy(&pio_p>mutex);
        ddi_soft_state_free(pio_softstate, instance);
        return (DDI_FAILURE);
    }

    /*
     * Initialize the device access attributes for the register mapping
     */
    dev_acc_attr.devacc_attr_version = DDI_DEVICE_ATTR_V0;
    dev_acc_attr.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;
    dev_acc_attr.devacc_attr_dataorder = DDI_STRICTORDER_ACC;

    /*
     * Map in the csr register (register 0)
     */
    if (ddi_regs_map_setup(dip, 0, (caddr_t *)&(pio_p->csr), 0,
        sizeof (Pio_csr), &dev_acc_attr, &pio_p->csr_handle) !=
        DDI_SUCCESS) {
        ddi_remove_intr(pio_p->dip, 0, pio_p->iblock_cookie);
        mutex_destroy(&pio_p->mutex);
        ddi_soft_state_free(pio_softstate, instance);
        return (DDI_FAILURE);
    }

    /*
     * Map in the data register (register 1)
```

例 6–5 標準的な attach() エントリポイント (続き)

```
     */
    if (ddi_regs_map_setup(dip, 1, (caddr_t *)&(pio_p->data), 0,
        sizeof (uchar_t), &dev_acc_attr, &pio_p->data_handle) !=
        DDI_SUCCESS) {
        ddi_remove_intr(pio_p->dip, 0, pio_p->iblock_cookie);
        ddi_regs_map_free(&pio_p->csr_handle);
        mutex_destroy(&pio_p->mutex);
        ddi_soft_state_free(pio_softstate, instance);
        return (DDI_FAILURE);
    }

    /*
     * Create an entry in /devices for user processes to open(2)
     * This driver will create a minor node entry in /devices
     * of the form:  /devices/..../pio@X,Y:pio
     */
    if (ddi_create_minor_node(dip, ddi_get_name(dip), S_IFCHR,
        instance, DDI_PSEUDO, 0) == DDI_FAILURE) {
        ddi_remove_intr(pio_p->dip, 0, pio_p->iblock_cookie);
        ddi_regs_map_free(&pio_p->csr_handle);
        ddi_regs_map_free(&pio_p->data_handle);
        mutex_destroy(&pio_p->mutex);
        ddi_soft_state_free(pio_softstate, instance);
        return (DDI_FAILURE);
    }

    /*
     * reset device (including disabling interrupts)
     */
    ddi_put8(pio_p->csr_handle, pio_p->csr, PIO_RESET);

    /*
     * report the name of the device instance which has attached
     */
    ddi_report_dev(dip);
    return (DDI_SUCCESS);

    case DDI_RESUME:
    return (DDI_SUCCESS);

    default:
    return (DDI_FAILURE);
    }
}
```

注 – attach() ルーチンでは、異なるデバイスインスタンスに対する呼び出しの順序に関して、どのような想定もしてはいけません。システムは、異なるデバイスインスタンスに対して attach() を同時に呼び出す可能性があります。システムはまた、異なるデバイスインスタンスに対して attach() と detach() を同時に呼び出す可能性もあります。

detach() エントリポイント

カーネルは、デバイスのインスタンスを切り離すため、または電源管理によるデバイスのインスタンスに対する操作を中断するために、ドライバの detach(9E) エントリポイントを呼び出します。このセクションでは、デバイスインスタンスを切り離す操作について説明します。電源管理の問題の説明については、第 12 章「電源管理」を参照してください。

ドライバの detach() エントリポイントは、そのドライバにバインドされたデバイスのインスタンスを切り離すために呼び出されます。このエントリポイントは、切り離されるデバイスノードのインスタンスと、このエントリポイントに対する cmd 引数として指定された DDI_DETACH を使用して呼び出されます。

ドライバは、すべてのタイムアウトまたはコールバックを取り消すか、または待機して完了したあと、戻る前にデバイスインスタンスに割り当てられたリソースをすべて解放する必要があります。ドライバが何らかの理由で、使用されていないリソースの未処理のコールバックを取り消すことができない場合、ドライバはデバイスを元の状態に戻し、エントリポイントから DDI_FAILURE を返す必要があります。これにより、デバイスインスタンスは接続された状態のままになります。

コールバックルーチンには、取り消すことができるコールバックと、取り消すことができないコールバックの 2 つのタイプがあります。timeout(9F) と bufcall(9F) のコールバックは、detach(9E) 中にドライバが原子的に取り消すことができます。scsi_init_pkt(9F) や ddi_dma_buf_bind_handle(9F) などのほかのタイプのコールバックは、取り消すことができません。ドライバは、コールバックが完了するまで detach() 内でブロックするか、または切り離しの要求を失敗させるかのどちらかを行う必要があります。

例 6–6　標準的な detach() エントリポイント

```
/*
 * detach(9e)
 * free the resources that were allocated in attach(9e)
 */
static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    Pio     *pio_p;
    int     instance;

    switch (cmd) {
    case DDI_DETACH:

    instance = ddi_get_instance(dip);
    pio_p = ddi_get_soft_state(pio_softstate, instance);

    /*
     * turn off the device
     * free any resources allocated in attach
     */
    ddi_put8(pio_p->csr_handle, pio_p->csr, PIO_RESET);
```

例 6–6 標準的な detach() エントリポイント (続き)

```
    ddi_remove_minor_node(dip, NULL);
    ddi_regs_map_free(&pio_p->csr_handle);
    ddi_regs_map_free(&pio_p->data_handle);
    ddi_remove_intr(pio_p->dip, 0, pio_p->iblock_cookie);
    mutex_destroy(&pio_p->mutex);
    ddi_soft_state_free(pio_softstate, instance);
    return (DDI_SUCCESS);

    case DDI_SUSPEND:
    default:
    return (DDI_FAILURE);
    }
}
```

getinfo() エントリポイント

システムは、ドライバのみが認識している構成情報を取得するために getinfo(9E) を呼び出します。デバイスインスタンスへのマイナー番号のマッピングは、ドライバによって完全に制御されます。システムは、特定の dev_t がどのデバイスを表すかをドライバに問い合わせることが必要になる場合があります。

getinfo() 関数は、*infocmd* 引数として DDI_INFO_DEVT2INSTANCE または DDI_INFO_DEVT2DEVINFO のどちらかを取ることができます。DDI_INFO_DEVT2INSTANCE コマンドは、デバイスのインスタンス番号を要求します。DDI_INFO_DEVT2DEVINFO コマンドは、デバイスの dev_info 構造体へのポインタを要求します。

DDI_INFO_DEVT2INSTANCE の場合は、*arg* が dev_t であり、getinfo() は dev_t 内のマイナー番号をインスタンス番号に変換する必要があります。次の例では、マイナー番号がインスタンス番号と同じであるため、getinfo() はマイナー番号を戻すのみで済みます。この場合は、getinfo() が attach() の前に呼び出される可能性があるため、ドライバでは状態構造体が使用できることを前提にしてはいけません。ドライバによって定義されるマイナーデバイス番号とインスタンス番号の間のマッピングは、必ずしも例に示したマッピングに従う必要はありません。ただし、いずれの場合も、マッピングは静的である必要があります。

DDI_INFO_DEVT2DEVINFO の場合も、*arg* が dev_t であるため、getinfo() は最初にデバイスのインスタンス番号をデコードします。getinfo() は次に、次の例に示すように、該当するデバイスに対するドライバのソフト状態構造体に保存された dev_info ポインタを戻します。

例 6–7 標準的な getinfo() エントリポイント

```
/*
 * getinfo(9e)
 * Return the instance number or device node given a dev_t
 */
static int
```

例 6–7 標準的な getinfo() エントリポイント (続き)

```
xxgetinfo(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
{
    int error;
    Pio *pio_p;
    int instance = getminor((dev_t)arg);

    switch (infocmd) {

    /*
     * return the device node if the driver has attached the
     * device instance identified by the dev_t value which was passed
     */
    case DDI_INFO_DEVT2DEVINFO:
    pio_p = ddi_get_soft_state(pio_softstate, instance);
    if (pio_p == NULL) {
        *result = NULL;
        error = DDI_FAILURE;
    } else {
        mutex_enter(&pio_p->mutex);
        *result = pio_p->dip;
        mutex_exit(&pio_p->mutex);
        error = DDI_SUCCESS;
    }
    break;

    /*
     * the driver can always return the instance number given a dev_t
     * value, even if the instance is not attached.
     */
    case DDI_INFO_DEVT2INSTANCE:
    *result = (void *)instance;
    error = DDI_SUCCESS;
    break;
    default:
    *result = NULL;
    error = DDI_FAILURE;
    }
    return (error);
}
```

注 – getinfo() ルーチンは、ドライバが作成するマイナーノードとの同期が維持される必要があります。マイナーノードが同期から外れた場合は、ホットプラグ操作がすべて失敗し、システムパニックが発生することがあります。

デバイス ID の使用

Oracle Solaris DDI インタフェースでは、ドライバが、デバイスの永続的な一意の識別子であるデバイス *ID* を提供できます。デバイス *ID* を使用すると、デバイスを識別または検索できます。デバイス *ID* は、`/devices` の名前やデバイス番号 (`dev_t`) とは独立しています。`libdevid(3LIB)` で定義された関数を使用すると、アプリケーションは、ドライバによって登録されたデバイス ID を読み取ったり操作したりすることができます。

ドライバがデバイス *ID* をエクスポートできるようにするには、ドライバは、そのデバイスが一意の ID を提供できるか、またはホストで生成された一意の ID を通常はアクセス不可能な領域に格納できることを確認する必要があります。WWN (World-Wide Number) は、デバイスによって提供される一意の ID の例です。デバイスの NVRAM や予約されたセクターは、ホストで生成された一意の ID を安全に格納できる、アクセス不可能な領域の例です。

デバイス ID の登録

ドライバは通常、そのドライバの `attach(9E)` ハンドラでデバイス ID を初期化して登録します。先に説明したように、ドライバは持続的なデバイス *ID* の登録を担当します。そのため、ドライバでは、一意の ID (WWN) を直接提供できるデバイスと、作成された ID が安定した記憶領域に書き込まれたり、その記憶領域から読み取られたりするデバイスの両方の処理が必要になることがあります。

デバイスによって提供された ID の登録

デバイスがドライバに一意の識別子を提供できる場合、ドライバは単にこの識別子を使用してデバイス *ID* を初期化し、その ID を Oracle Solaris DDI に登録できます。

```
/*
 * The device provides a guaranteed unique identifier,
 * in this case a SCSI3-WWN.  The WWN for the device has been
 * stored in the device's soft state.
 */
if (ddi_devid_init(dip, DEVID_SCSI3_WWN, un->un_wwn_len, un->un_wwn,
    &un->un_devid) != DDI_SUCCESS)
    return (DDI_FAILURE);

(void) ddi_devid_register(dip, un->un_devid);
```

組み立てられた ID の登録

ドライバはまた、一意の ID を直接提供しないデバイスのデバイス ID を登録する可能性もあります。これらの ID を登録するには、そのデバイスが少量のデータを予約領域に格納したり、取得したりできることが必要です。それにより、ドライバは組み立てられたデバイス ID を作成し、それを予約領域に書き込むことができます。

```
/*
 * the device doesn't supply a unique ID, attempt to read
 * a fabricated ID from the device's reserved data.
 */

if (xxx_read_deviceid(un, &devid_buf) == XXX_OK) {
    if (ddi_devid_valid(devid_buf) == DDI_SUCCESS) {
        devid_sz = ddi_devi_sizeof(devid_buf);
        un->un_devid = kmem_alloc(devid_sz, KM_SLEEP);
        bcopy(devid_buf, un->un_devid, devid_sz);
        ddi_devid_register(dip, un->un_devid);
        return (XXX_OK);
    }
}

/*
 * we failed to read a valid device ID from the device
 * fabricate an ID, store it on the device, and register
 * it with the DDI
 */

if (ddi_devid_init(dip, DEVID_FAB, 0, NULL, &un->un_devid)
    == DDI_FAILURE) {
    return (XXX_FAILURE);
}

if (xxx_write_deviceid(un) != XXX_OK) {
    ddi_devid_free(un->un_devid);
    un->un_devid = NULL;
    return (XXX_FAILURE);
}

ddi_devid_register(dip, un->un_devid);
return (XXX_OK);
```

デバイス ID の登録解除

ドライバは通常、detach(9E) 処理の一部として、割り当てられたすべてのデバイス *ID* の登録を解除して解放します。ドライバはまず、ddi_devid_unregister(9F) を呼び出して、デバイスインスタンスのデバイス *ID* の登録を解除します。ドライバは次に、ddi_devid_free(9F) を呼び出し、ddi_devid_init(9F) によって返されたハンドルを渡すことによって、デバイス *ID* のハンドル自体を解放する必要があります。ドライバは、WWN またはシリアル番号データに割り当てられたすべての領域の管理を担当します。

◆ ◆ ◆ 第 7 章

7 デバイスアクセス: プログラム式入出力

Oracle Solaris OS ではドライバ開発者向けに、デバイスメモリーにアクセスするための包括的なインタフェースセットが用意されています。これらのインタフェースはドライバがプラットフォーム固有の依存関係を意識しないですむような設計になっており、プロセッサとデバイスとの間のエンディアンの不一致を処理するほか、デバイスで発生する可能性のあるデータ順序依存関係を適用します。これらのインタフェースを使用すれば、SPARC、x86 の両方のプロセッサアーキテクチャーで動作することはもちろん、それぞれのプロセッサファミリに属する各種プラットフォーム上でも動作するような単一ソースのドライバを開発できます。

この章では、次の内容について説明します。

デバイスメモリー

プログラム式入出力をサポートするデバイスには、デバイスのアドレス指定可能領域にマップされるバスアドレス空間の 1 つ以上の領域が割り当てられます。これらのマッピングは、デバイスに関連付けられた reg プロパティー内の値ペアとして記述されます。各値ペアはバスアドレスの 1 つのセグメントを記述します。

ドライバは、特定のバスアドレスマッピングを特定するために、レジスタ番号または regspec を指定します。この番号は、デバイスの reg プロパティー内でのインデックスです。reg プロパティーは、デバイスの busaddr と size を特定します。ドライバは、ddi_regs_map_setup(9F) などの DDI 関数を呼び出すときに、このレジスタ番号を渡します。ドライバ内で、デバイスに割り当てられたマップ可能領域の個数を判定するには、ddi_dev_nregs(9F) を呼び出します。

デバイスとホストのエンディアンの違いの管理

ホストのデータ形式は、デバイスのデータ形式とは異なるエンディアン特性を備えている可能性があります。そのような場合、ホストとデバイスとの間で転送されるデータをバイトスワップすることで、転送先のデータ形式の要件に合わせる必要があります。ホストと同じエンディアン特性を備えたデバイスでは、データのバイトスワップを行う必要はありません。

ドライバでデバイスのエンディアン特性を指定するには、ddi_device_acc_attr(9S) に渡される ddi_regs_map_setup(9F) 構造体の対応するフラグを設定します。そうすると、ドライバが ddi_get8(9F) などの ddi_get*X* ルーチンや ddi_put16(9F) などの ddi_put*X* ルーチンを呼び出してデバイスメモリーに対する読み取りまたは書き込みを行うときに、DDI フレームワークによって必要なバイトスワップがすべて実行されます。

データ順序付け要件の管理

各プラットフォームでは、データのロードや格納の順序を変更することでパフォーマンスを最適化できます。デバイスによっては順序変更が許されない場合もあるため、ドライバでデバイスへのマッピングを設定する際にデバイスの順序付け要件を指定する必要があります。

ddi_device_acc_attr 構造体

この構造体は、デバイスのエンディアン要件とデータ順序要件を記述します。ドライバは、この構造体を初期化し、ddi_regs_map_setup(9F) の引数として渡す必要があります。

```
typedef struct ddi_device_acc_attr {
    ushort_t       devacc_attr_version;
    uchar_t    devacc_attr_endian_flags;
    uchar_t    devacc_attr_dataorder;
} ddi_device_acc_attr_t;
```

devacc_attr_version
: DDI_DEVICE_ATTR_V0 を指定します。

devacc_attr_endian_flags
: デバイスのエンディアン特性を記述します。ビット値として指定され、使用可能な値は次のとおりです。

- DDI_NEVERSWAP_ACC – データのスワップを一切行わない
- DDI_STRUCTURE_BE_ACC – デバイスのデータ形式がビッグエンディアンである
- DDI_STRUCTURE_LE_ACC – デバイスのデータ形式がリトルエンディアンである

`devacc_attr_dataorder`

CPU がデバイスから要求されたデータを参照する際に従うべき順序を記述します。列挙値として指定されます。ただし、データアクセス制限は、厳格度の高いものから順に並べられます。

- `DDI_STRICTORDER_ACC` – ホストは、プログラマによって指定された順番で参照を発行する必要があります。このフラグがデフォルトの動作です。
- `DDI_UNORDERED_OK_ACC` – ホストは、デバイスメモリーに対するロードや格納の順序を変更できます。
- `DDI_MERGING_OK_ACC` – ホストは、個々の格納をマージして連続する場所にまとめることができます。この設定では暗黙的に順序変更も許可されます。
- `DDI_LOADCACHING_OK_ACC` – 格納が発生するまで、ホストはデバイスからデータを読み取ることができます。
- `DDI_STORECACHING_OK_ACC` – ホストは、デバイスに書き込むデータをキャッシュに格納しておくことができます。このときホストは、デバイスへのデータ書き込みを、将来必要になるまで遅らせることができます。

注 – システムは、ドライバで `devacc_attr_dataorder` に指定されているよりも厳格に、データへのアクセスを行うことができます。厳格なデータ順序付けからキャッシュ格納へと移る間、ドライバによるデータアクセスに関してホストの制限が小さくなります。

デバイスメモリーのマッピング

ドライバは通常、`attach(9E)` の実行中にデバイスのすべての領域をマップします。ドライバ内でデバイスメモリーの 1 つの領域をマップするには、そのマップする領域のレジスタ番号、領域のデバイスアクセス属性、オフセット、およびサイズを指定して `ddi_regs_map_setup(9F)` を呼び出します。DDI フレームワークは、そのデバイス領域のマッピングを設定し、不透明なハンドルをドライバに返します。このデータアクセスハンドルは、デバイスのその領域からデータを読み取ったりその領域にデータを書き込んだりする際に、`ddi_get8(9F)` または `ddi_put8(9F)` ルーチンファミリの引数として渡されます。

ドライバは、デバイスマッピングの形式がドライバで求められる形式と一致しているかどうかを確認するために、デバイスからエクスポートされたマッピングの数を

チェックします。ドライバは、ddi_dev_nregs(9F) を呼び出したあと、各マッピングのサイズを確認するために ddi_dev_regsize(9F) を呼び出します。

マッピングの設定例

次に、DDI データアクセスインタフェースの簡単な例を示します。このドライバは、一度に 1 つの文字を受け付け、次の文字の受け付け準備が整ったら割り込みを生成するような、架空のリトルエンディアンデバイスに対するものです。このデバイスには 2 つのレジスタセットが実装されています。1 つは 8 ビット CSR レジスタ、もう 1 つは 8 ビットデータレジスタです。

例 7–1 マッピングの設定

```
#define CSR_REG 0
#define DATA_REG 1
/*
 * Initialize the device access attributes for the register
 * mapping
 */
dev_acc_attr.devacc_attr_version = DDI_DEVICE_ATTR_V0;
dev_acc_attr.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;
dev_acc_attr.devacc_attr_dataorder = DDI_STRICTORDER_ACC;
/*
 * Map in the csr register (register 0)
 */
if (ddi_regs_map_setup(dip, CSR_REG, (caddr_t *)&(pio_p->csr), 0,
  sizeof (Pio_csr), &dev_acc_attr, &pio_p->csr_handle) != DDI_SUCCESS) {
mutex_destroy(&pio_p->mutex);
ddi_soft_state_free(pio_softstate, instance);
return (DDI_FAILURE);
}
/*
 * Map in the data register (register 1)
 */
if (ddi_regs_map_setup(dip, DATA_REG, (caddr_t *)&(pio_p->data), 0,
  sizeof (uchar_t), &dev_acc_attr, &pio_p->data_handle) \
        != DDI_SUCCESS) {
mutex_destroy(&pio_p->mutex);
ddi_regs_map_free(&pio_p->csr_handle);
ddi_soft_state_free(pio_softstate, instance);
return (DDI_FAILURE);
}
```

デバイスアクセス関数

ドライバは、ddi_get8(9F) および ddi_put8(9F) ルーチンファミリと、ddi_regs_map_setup(9F) から返されたハンドルとを組み合わせて使用することで、デバイス間でデータを転送します。DDI フレームワークは、ホストまたはデバ

イスのエンディアン形式に合わせるために必要なすべてのバイトスワップを自動的に処理するとともに、デバイスで発生する可能性のあるすべての格納順序付け制約を適用します。

DDI には、8 ビット、16 ビット、32 ビット、64 ビットの各サイズでデータを転送するためのインタフェースが用意されているほか、複数の値を繰り返し転送するためのインタフェースも用意されています。ddi_get8(9F)、ddi_put8(9F)、ddi_rep_get8(9F)、および ddi_rep_put8(9F) ルーチンファミリのマニュアルページで、これらのインタフェースの一覧表や説明を参照してください。

次の例は、ドライバ内でデバイスの CSR レジスタとデータレジスタのマッピングを行なった例 7–1 に基づいたものです。このドライバの write(9E) エントリポイントは、呼び出し時にデータバッファーを一度に 1 バイトずつデバイスに書き込みます。

例 7–2 マッピングの設定: バッファー

```
static  int
pio_write(dev_t dev, struct uio *uiop, cred_t *credp)
{
    int  retval;
    int  error = OK;
    Pio *pio_p = ddi_get_soft_state(pio_softstate, getminor(dev));
    if (pio_p == NULL)
    return (ENXIO);
    mutex_enter(&pio_p->mutex);
    /*
     * enable interrupts from the device by setting the Interrupt
     * Enable bit in the devices CSR register
     */
    ddi_put8(pio_p->csr_handle, pio_p->csr,
      (ddi_get8(pio_p->csr_handle, pio_p->csr) | PIO_INTR_ENABLE));
    while (uiop->uio_resid > 0) {
    /*
     * This device issues an IDLE interrupt when it is ready
     * to accept a character; the interrupt can be cleared
     * by setting PIO_INTR_CLEAR.  The interrupt is reasserted
     * after the next character is written or the next time
     * PIO_INTR_ENABLE is toggled on.
     *
     * wait for interrupt (see pio_intr)
     */
     cv_wait(&pio_p->cv, &pio_p->mutex);
     /*
      * get a character from the user's write request
      * fail the write request if any errors are encountered
      */
     if ((retval = uwritec(uiop)) == -1) {
         error = retval;
         break;
     }
     /*
```

例 7–2 マッピングの設定: バッファー (続き)

```
        * pass the character to the device by writing it to
        * the device's data register
        */
       ddi_put8(pio_p->data_handle, pio_p->data, (uchar_t)retval);
      }
      /*
       * disable interrupts by clearing the Interrupt Enable bit
       * in the CSR
       */
      ddi_put8(pio_p->csr_handle, pio_p->csr,
        (ddi_get8(pio_p->csr_handle, pio_p->csr) & ~PIO_INTR_ENABLE));
      mutex_exit(&pio_p->mutex);
      return (error);
}
```

代替のデバイスアクセスインタフェース

Oracle Solaris OS には、ddi_get8(9F) and ddi_put8(9F) インタフェースファミリ経由ですべてのデバイスアクセスを実装する方法だけでなく、特定のバス実装に固有のインタフェースも用意されています。これらの関数は一部のプラットフォームで効率が高くなる可能性もありますが、これらのルーチンを使用すると、デバイスのさまざまなバスバージョンでのドライバの移植性が低下する可能性があります。

メモリー空間アクセス

メモリーマップアクセスでは、デバイスレジスタがメモリーアドレス空間に現れます。ddi_get*X* および ddi_put*X* ルーチンファミリが、標準のデバイスアクセスインタフェースの代替手段としてドライバから使用可能となっています。

入出力空間アクセス

入出力空間アクセスでは、デバイスレジスタが入出力空間に現れます。このとき、アドレス指定可能な各要素は入出力ポートと呼ばれます。ddi_io_get8(9F) および ddi_io_put8(9F) ルーチンが、標準のデバイスアクセスインタフェースの代替手段としてドライバから使用可能となっています。

PCI 構成空間アクセス

通常のデバイスアクセスインタフェースを使用しないで PCI 構成空間にアクセスするには、ドライバ内で、PCI 構成空間をマップするために、pci_config_setup(9F) の代わりに ddi_regs_map_setup(9F) を呼び出す必要があります。その後ドライバ内で、pci_config_get8(9F) および pci_config_put8(9F) インタフェースファミリを呼び出して PCI 構成空間にアクセスできます。

◆ ◆ ◆ 第 8 章

8 割り込みハンドラ

この章では、割り込みの割り当て、登録、保守、削除など、割り込みを処理するためのメカニズムについて説明します。この章では、次の内容について説明します。

割り込みハンドラの概要

割り込みとは、デバイスから CPU へのハードウェアシグナルのことです。割り込みによって、デバイスが注意を必要としていること、および CPU は現在の処理を停止してデバイスに応答するべきことが CPU に通知されます。CPU が割り込みの優先順位より高い優先順位を持つタスクを実行していない場合、CPU は現在のスレッドを中断します。次に、CPU は、割り込みシグナルを送信したデバイスの割り込みハンドラを呼び出します。割り込みハンドラのジョブは、デバイスを保守すること、およびデバイスによる割り込みを停止することです。割り込みハンドラが復帰すると、CPU は割り込みが発生する前に行っていた処理を再開します。

Oracle Solaris DDI/DKI には、次のタスクを実行するためのインタフェースが用意されています。

- 割り込みタイプと登録要件の判定
- 割り込みの登録
- 割り込みの保守
- 割り込みのマスク
- 割り込みの中断情報の取得
- 優先順位情報の取得と設定

デバイス割り込み

入出力バスは、2 つの一般的な方法、つまりベクター方式とポーリング方式で割り込みを実装します。一般に、どちらの方法でも、バス割り込みの優先順位レベルが指定されます。ベクター方式のデバイスでは、割り込みベクターも指定されます。ポーリング方式のデバイスでは、割り込みベクターは指定されません。

変化を続けるバス技術を取り入れるため、Oracle Solaris OS は拡張され、新しいタイプの割り込みと、長年使用されてきた従来の割り込みの両方に対応するようになりました。特に、Solaris OS は次の 3 つのタイプの割り込みを認識するようになりました。

- レガシー割り込み – レガシー割り込みまたは固定割り込みとは、古いバス技術を使用する割り込みのことです。この技術では、割り込みのシグナルが「帯域外」で接続された、つまりバスのメインラインとは別個に接続された 1 つ以上の外部ピンを使って送信されます。PCI Express などの新しいバス技術では、帯域内メカニズムによってレガシー割り込みをエミュレートすることで、ソフトウェアの互換性が維持されます。このエミュレートされた割り込みは、ホスト OS によってレガシー割り込みとして扱われます。
- メッセージシグナル割り込み – メッセージシグナル割り込み (MSI) は、ピンを使用しない帯域内メッセージであり、ホストブリッジ内のアドレスをターゲットにすることができます。ホストブリッジの詳細については、605 ページの「PCI ローカルバス」を参照してください。MSI では、割り込みメッセージとともにデータを送信できます。各 MSI は共有されないため、デバイスに割り当てられる MSI はシステム内で一意であることが保証されます。PCI 関数は、最大で 32 個の MSI メッセージを要求できます。
- 拡張メッセージシグナル割り込み – 拡張メッセージシグナル割り込み (MSI-X) は、MSI の拡張バージョンです。MSI-X 割り込みでは、次の利点が追加されています。
  - 32 個ではなく 2048 個のメッセージをサポートしている
  - メッセージごとに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ている
  - メッセージごとのマスキングをサポートしている
  - ソフトウェアによるベクターの割り当て数がハードウェアの要求より少ない場合に柔軟性に優れている。ソフトウェアは、MSI-X の同じアドレスとデータを複数の MSI-X スロットで再利用できます。

注 – PCI Express などの一部の新しいバス技術には MSI が必要ですが、INTx エミュレーションを使用することでレガシー割り込みに対応できます。INTx エミュレーションは互換性を確保するために使用されますが、優れた方法であるとは考えられていません。

高レベルの割り込み

バスは、バス割り込みレベルのデバイス割り込みを優先します。次に、バス割り込みレベルは、プロセッサ割り込みレベルにマッピングされます。スケジューラ優先順位レベルよりも上位にある CPU 割り込み優先順位にマッピングされるバス割り込みレベルは、高レベルの割り込みと呼ばれます。高レベルの割り込みハンドラは、次の DDI インタフェースの呼び出しに限定されています。

- 高レベルの割り込みに関連付けられた割り込み優先順位によって初期化される mutex の `mutex_enter(9F)` および `mutex_exit(9F)`
- `ddi_intr_trigger_softint(9F)`
- 次の DDI `get` および `put` ルーチン: `ddi_get8(9F)`、`ddi_put8(9F)`、`ddi_get16(9F)`、`ddi_put16(9F)`、`ddi_get32(9F)`、`ddi_put32(9F)`、`ddi_get64(9F)`、および `ddi_put64(9F)`

バス割り込みレベルは、それだけでデバイス割り込みが高レベルかどうかを判定するわけではありません。特定のバス割り込みレベルを、あるプラットフォームでは高レベルの割り込みにマッピングし、別のプラットフォームでは通常の割り込みにマッピングできます。

高レベルの割り込みを持つデバイスをサポートするためのドライバは必要ありません。ただし、割り込みレベルをチェックするドライバは必要です。割り込み優先順位がシステムの最上位の優先順位以上である場合、割り込みハンドラは高レベルの割り込みコンテキストで実行されます。この場合、ドライバは接続に失敗するか、2 つのレベルのスキームを使って割り込みを処理する可能性があります。詳細については、155 ページの「高レベルの割り込みの処理」を参照してください。

レガシー割り込み

システムがデバイス割り込みに関して持っている唯一の情報は、バス割り込みの優先順位レベルと割り込み要求番号です。バス割り込みの優先順位レベルの例としては、SPARC マシンの SBus の IPL があります。割り込み要求番号の例としては、x86 マシンの ISA バスの IRQ があります。

割り込みハンドラが登録されると、そのハンドラはシステムによって各 IPL または IRQ の潜在的な割り込みハンドラのリストに追加されます。割り込みが発生すると、システムは、指定された IPL または IRQ に関連付けられたすべてのデバイスの中から、実際に割り込みを発生させたデバイスを判定する必要があります。システムは、ハンドラが割り込みを取り込むまで、指定された IPL または IRQ のすべての割り込みハンドラを呼び出します。

次のバスは、ポーリング方式の割り込みをサポートできます。

- SBus
- ISA

- PCI

標準メッセージシグナル割り込みと拡張メッセージシグナル割り込み

標準メッセージシグナル割り込み (MSI) と拡張メッセージシグナル割り込み (MSI-X) の両方が、帯域内メッセージとして実装されます。メッセージシグナル割り込みは、ソフトウェアによって指定されたアドレスと値を持つ書き込みとして送信されます。

MSI 割り込み

従来の PCI 仕様には、任意のサポートとしてメッセージシグナル割り込み (MSI) が含まれています。MSI は、送信される書き込みとして実装される帯域内メッセージです。MSI のアドレスとデータはソフトウェアによって指定され、ホストブリッジに固有です。メッセージは帯域内にあるため、メッセージの受信は、割り込みに関連付けられたデータの「プッシュ」に使用できます。定義上、MSI 割り込みは共有されません。デバイスに割り当てられた各 MSI メッセージは、システム内で一意のメッセージであることが保証されています。PCI 関数は、1、2、4、8、16、または 32 個の MSI メッセージを要求できます。システムソフトウェアは、関数の要求より少ない数の MSI メッセージを関数に割り当てることができます。ホストブリッジは、デバイスに割り当てられた一意の MSI メッセージの数で制限できます。

MSI-X 割り込み

MSI-X 割り込みは MSI 割り込みの拡張バージョンであり、MSI 割り込みと同じ機能を持っていますが、次のような主な違いがあります。

- デバイスごとに最大 2048 個の MSI-X 割り込みベクターがサポートされている。
- アドレスとデータのエントリが割り込みベクターごとに一意である。
- MSI-X は、関数ごとのマスキングとベクターごとのマスキングをサポートしている。

MSI-X 割り込みでは、デバイスの未割り当ての割り込みベクターは、前に追加または初期化された MSI-X 割り込みベクターを使用して、同じベクターアドレス、ベクターデータ、割り込みハンドラ、およびハンドラ引数を共有できます。`ddi_intr_dup_handler(9F)` 関数を使用すると、関連付けられたデバイスの未割り当ての割り込みベクターに対して、Oracle Solaris OS から提供されるリソースに別名を付けることができます。たとえば、2 つの MSI-X 割り込みがドライバに割り当てられ、32 個の割り込みがデバイスでサポートされている場合、そのドライバは `ddi_intr_dup_handler()` を使用して、デバイスの 30 個の追加割り込みに対して、受信した 2 つの割り込みに別名を付けることができます。

`ddi_intr_dup_handler()` 関数は、`ddi_intr_add_handler(9F)` を使って追加された、または`ddi_intr_enable(9F)` を使って初期化された割り込みを複製できます。

複製された割り込みは、最初は無効になっています。複製された割り込みを有効にするには、ddi_intr_enable () を使用します。元の MSI-X 割り込みハンドラは、この元の割り込みハンドラに関連付けられているすべての複製された割り込みハンドラが削除されるまで削除できません。複製された割り込みハンドラを削除するには、まず ddi_intr_disable(9F) を呼び出し、次に ddi_intr_free(9F) を呼び出します。元の割り込みハンドラに関連付けられているすべての複製された割り込みハンドラが削除されたら、ddi_intr_remove_handler(9F) を使用して元の MSI-X 割り込みハンドラを削除できます。例については、ddi_intr_dup_handler (9F) のマニュアルページを参照してください。

ソフトウェア割り込み

Oracle Solaris DDI/DKI は、ソフトウェア割り込み (ソフト割り込み とも呼ばれる) をサポートしています。ソフト割り込みは、ハードウェアデバイスではなくソフトウェアによって開始されます。この割り込みのハンドラは、システムとの間でも追加および削除する必要があります。ソフト割り込みハンドラは割り込みコンテキストで実行されるため、割り込みハンドラに属するタスクの多くを実行するために使用できます。

ハードウェア割り込みハンドラは、これらのタスクの実行中にほかのシステムアクティビティーを中断しなければならないことがあるため、タスクをすばやく実行する必要があります。この要件は、システムスケジューラの優先順位レベルより高い優先順位レベルで動作する高レベルの割り込みハンドラに特に当てはまります。高レベルの割り込みハンドラは、システムクロックの割り込み処理を含め、優先順位の低いすべての割り込み処理をマスクします。したがって、割り込みハンドラは、mutex の獲得など、割り込みハンドラのスリープの原因になることがあるアクティビティーにかかわることを避ける必要があります。

ハンドラがスリープ状態になると、クロックがマスクされてスリープスレッドのスケジューリングができなくなるため、システムがハングアップすることがあります。この理由で、高レベルの割り込みハンドラは通常、最小限の処理を優先順位の高いレベルで実行し、ほかのタスクをソフトウェア割り込みに委任します。ソフトウェア割り込みは、高レベルの割り込みハンドラの優先順位レベルの下で動作します。ソフトウェア割り込みハンドラはシステムスケジューラの優先順位レベルの下で動作するため、ソフトウェア割り込みハンドラは、高レベルの割り込みハンドラができなかった処理を行うことができます。

DDI 割り込み関数

Oracle Solaris OS では、割り込みの登録と登録解除を行うフレームワークが用意されており、メッセージシグナル割り込み (MSI) がサポートされています。割り込み管理インタフェースを使用して、割り込み優先順位、割り込み許可フラグ、および割り込みマスクを操作したり、中断情報を取得したりすることができます。

割り込み許可フラグ関数

次の関数を使用すると、割り込み情報を取得できます。

ddi_intr_get_navail(9F)	指定したハードウェアデバイスと割り込みタイプで使用できる割り込みの数を返します。
ddi_intr_get_nintrs(9F)	指定した割り込みタイプでデバイスがサポートしている割り込みの数を返します。
ddi_intr_get_supported_types(9F)	デバイスとホストの両方でサポートされているハードウェア割り込みのタイプを返します。
ddi_intr_get_cap(9F)	指定した割り込みの割り込み許可フラグを返します。

割り込み初期化関数と割り込み破棄関数

次の関数を使用すると、割り込みを作成および削除することができます。

ddi_intr_alloc(9F)	指定したタイプの割り込みのシステムリソースと割り込みベクターを割り当てます。
ddi_intr_free(9F)	指定した割り込みハンドルのシステムリソースと割り込みベクターを解放します。
ddi_intr_set_cap(9F)	DDI_INTR_FLAG_LEVEL および DDI_INTR_FLAG_EDGE フラグを使用して、指定した割り込みの許可フラグを設定します。
ddi_intr_add_handler(9F)	割り込みハンドラを追加します。
ddi_intr_dup_handler(9F)	MSI-X とともにのみ使用します。割り当てられた割り込みベクターのアドレスとデータのペアを、同じデバイスの未使用の割り込みベクターにコピーします。
ddi_intr_remove_handler(9F)	指定した割り込みハンドラを削除します。

ddi_intr_enable(9F)	指定した割り込みを有効にします。
ddi_intr_disable(9F)	指定した割り込みを無効にします。
ddi_intr_block_enable(9F)	MSI とともにのみ使用します。指定した範囲の割り込みを有効にします。
ddi_intr_block_disable(9F)	MSI とともにのみ使用します。指定した範囲の割り込みを無効にします。
ddi_intr_set_mask(9F)	指定した割り込みが有効になっている場合に、割り込みマスクを設定します。
ddi_intr_clr_mask(9F)	指定した割り込みが有効になっている場合に、割り込みマスクをクリアします。
ddi_intr_get_pending(9F)	割り込み中断ビットがホストブリッジまたはデバイスでサポートされている場合に、そのようなビットを読み取ります。

優先順位管理関数

次の関数を使用すると、優先順位情報を取得および設定することができます。

ddi_intr_get_pri(9F)	指定した割り込みの現在のソフトウェア優先順位設定を返します。
ddi_intr_set_pri(9F)	指定した割り込みの割り込み優先順位レベルを設定します。
ddi_intr_get_hilevel_pri(9F)	高レベルの割り込みの最小優先順位レベルを返します。

ソフト割り込み関数

次の関数を使用すると、ソフト割り込みおよびソフト割り込みハンドラを操作できます。

ddi_intr_add_softint(9F)	ソフト割り込みハンドラを追加します。
ddi_intr_trigger_softint(9F)	指定したソフト割り込みをトリガーします。
ddi_intr_remove_softint(9F)	指定したソフト割り込みハンドラを削除します。
ddi_intr_get_softint_pri(9F)	指定した割り込みのソフト割り込み優先順位を返します。

ddi_intr_set_softint_pri(9F)	指定したソフト割り込みの相対ソフト割り込み優先順位を変更します。

割り込み関数の例

このセクションでは、次のタスクの実行例を示します。

- ソフト割り込み優先順位の変更
- 割り込みの中断の確認
- 割り込みマスクの設定
- 割り込みマスクのクリア

例 8–1 ソフト割り込み優先順位の変更

ソフト割り込み優先順位を 9 に変更するときは、ddi_intr_set_softint_pri(9F) 関数を使用します。

```
if (ddi_intr_set_softint_pri(mydev->mydev_softint_hdl, 9) != DDI_SUCCESS)
    cmn_err (CE_WARN, "ddi_intr_set_softint_pri failed");
```

例 8–2 割り込みの中断の確認

割り込みが中断されているかどうか確認するときは、ddi_intr_get_pending(9F) 関数を使用します。

```
if (ddi_intr_get_pending(mydevp->htable[0], &pending) != DDI_SUCCESS)
    cmn_err(CE_WARN, "ddi_intr_get_pending() failed");
else if (pending)
    cmn_err(CE_NOTE, "ddi_intr_get_pending(): Interrupt pending");
```

例 8–3 割り込みマスクの設定

デバイスが割り込みを受信しないように割り込みマスクを設定するときは、ddi_intr_set_mask(9F) 関数を使用します。

```
if ((ddi_intr_set_mask(mydevp->htable[0]) != DDI_SUCCESS))
    cmn_err(CE_WARN, "ddi_intr_set_mask() failed");
```

例 8–4 割り込みマスクのクリア

割り込みマスクをクリアするときは、ddi_intr_clr_mask(9F) 関数を使用します。指定した割り込みが有効になっていない場合、ddi_intr_clr_mask(9F) 関数は失敗します。ddi_intr_clr_mask(9F) 関数が成功した場合、デバイスは割り込みの生成を開始します。

```
if (ddi_intr_clr_mask(mydevp->htable[0]) != DDI_SUCCESS)
    cmn_err(CE_WARN, "ddi_intr_clr_mask() failed");
```

割り込みの登録

デバイスドライバが割り込みを受信および保守するには、ドライバが ddi_intr_add_handler(9F) を呼び出して割り込みハンドラをシステムに登録する必要があります。割り込みハンドラを登録すると、システムは割り込みハンドラを割り込み仕様に関連付けることができます。割り込みハンドラは、デバイスがその割り込みを担当している可能性のあるときに呼び出されます。ハンドラは、割り込みを処理するかどうかの判定、および処理する場合にはその割り込みの取り込みを担当します。

ヒント – サポートされている SPARC または x86 システムでデバイスの登録済み割り込み情報を取得するときは、mdb または kmdb デバッガの ::interrupts コマンドを使用します。

レガシー割り込みの登録

ドライバの割り込みハンドラを登録する場合、ドライバは通常、その attach(9E) エントリポイントで次の手順を実行します。

1. ddi_intr_get_supported_types(9F) を使用して、サポートされている割り込みのタイプを判定します。
2. ddi_intr_get_nintrs(9F) を使用して、サポートされている割り込みタイプの数を判定します。
3. kmem_zalloc(9F) を使用して、DDI 割り込みハンドルにメモリーを割り当てます。
4. 割り当てる割り込みタイプごとに、次の手順を実行します。
   a. ddi_intr_get_pri(9F) を使用して、割り込みの優先順位を取得します。
   b. 割り込みに新しい優先順位を設定する必要がある場合は、ddi_intr_set_pri(9F) を使用します。
   c. mutex_init(9F) を使用して、ロックを初期化します。
   d. ddi_intr_add_handler(9F) を使用して、割り込みのハンドラを登録します。
   e. ddi_intr_enable(9F) を使用して、割り込みを有効にします。
5. 次の手順を実行して、各割り込みを解放します。
   a. ddi_intr_disable(9F) を使用して、各割り込みを無効にします。
   b. ddi_intr_remove_handler(9F) を使用して、割り込みハンドラを削除します。
   c. mutex_destroy(9F) を使用して、ロックを削除します。
   d. ddi_intr_free(9F) および kmem_free(9F) を使用して DDI 割り込みハンドルに割り当てられたメモリーを解放し、割り込みを解放します。

例 8–5 レガシー割り込みの登録

次の例は、mydev というデバイスの割り込みハンドラをインストールする方法を示しています。この例では、mydev が 1 つの割り込みだけをサポートしていることを前提としています。

```
/* Determine which types of interrupts supported */
ret = ddi_intr_get_supported_types(mydevp->mydev_dip, &type);

if ((ret != DDI_SUCCESS) || (!(type & DDI_INTR_TYPE_FIXED))) {
    cmn_err(CE_WARN, "Fixed type interrupt is not supported");
    return (DDI_FAILURE);
}

/* Determine number of supported interrupts */
ret = ddi_intr_get_nintrs(mydevp->mydev_dip, DDI_INTR_TYPE_FIXED,
    &count);

/*
 * Fixed interrupts can only have one interrupt. Check to make
 * sure that number of supported interrupts and number of
 * available interrupts are both equal to 1.
 */
if ((ret != DDI_SUCCESS) || (count != 1)) {
    cmn_err(CE_WARN, "No fixed interrupts");
    return (DDI_FAILURE);
}

/* Allocate memory for DDI interrupt handles */
mydevp->mydev_htable = kmem_zalloc(sizeof (ddi_intr_handle_t),
    KM_SLEEP);
ret = ddi_intr_alloc(mydevp->mydev_dip, mydevp->mydev_htable,
    DDI_INTR_TYPE_FIXED, 0, count, &actual, 0);

if ((ret != DDI_SUCCESS) || (actual != 1)) {
    cmn_err(CE_WARN, "ddi_intr_alloc() failed 0x%x", ret);
    kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));

    return (DDI_FAILURE);
}

/* Sanity check that count and available are the same. */
ASSERT(count == actual);

/* Get the priority of the interrupt */
if (ddi_intr_get_pri(mydevp->mydev_htable[0], &mydevp->mydev_intr_pri)) {
    cmn_err(CE_WARN, "ddi_intr_alloc() failed 0x%x", ret);

    (void) ddi_intr_free(mydevp->mydev_htable[0]);
    kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));

    return (DDI_FAILURE);
}

cmn_err(CE_NOTE, "Supported Interrupt pri = 0x%x", mydevp->mydev_intr_pri);

/* Test for high level mutex */
```

例 8–5 レガシー割り込みの登録 *(続き)*

```
if (mydevp->mydev_intr_pri >= ddi_intr_get_hilevel_pri()) {
    cmn_err(CE_WARN, "Hi level interrupt not supported");

    (void) ddi_intr_free(mydevp->mydev_htable[0]);
    kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));

    return (DDI_FAILURE);
}

/* Initialize the mutex */
mutex_init(&mydevp->mydev_int_mutex, NULL, MUTEX_DRIVER,
    DDI_INTR_PRI(mydevp->mydev_intr_pri));

/* Register the interrupt handler */
if (ddi_intr_add_handler(mydevp->mydev_htable[0], mydev_intr,
   (caddr_t)mydevp, NULL) !=DDI_SUCCESS) {
    cmn_err(CE_WARN, "ddi_intr_add_handler() failed");

    mutex_destroy(&mydevp->mydev_int_mutex);
    (void) ddi_intr_free(mydevp->mydev_htable[0]);
    kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));

    return (DDI_FAILURE);
}

/* Enable the interrupt */
if (ddi_intr_enable(mydevp->mydev_htable[0]) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "ddi_intr_enable() failed");

    (void) ddi_intr_remove_handler(mydevp->mydev_htable[0]);
    mutex_destroy(&mydevp->mydev_int_mutex);
    (void) ddi_intr_free(mydevp->mydev_htable[0]);
    kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));

    return (DDI_FAILURE);
}
return (DDI_SUCCESS);
}
```

例 8–6 レガシー割り込みの削除

次の例は、レガシー割り込みを削除する方法を示しています。

```
/* disable interrupt */
(void) ddi_intr_disable(mydevp->mydev_htable[0]);

/* Remove interrupt handler */
(void) ddi_intr_remove_handler(mydevp->mydev_htable[0]);

/* free interrupt handle */
(void) ddi_intr_free(mydevp->mydev_htable[0]);

/* free memory */
kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));
```

MSI 割り込みの登録

ドライバの割り込みハンドラを登録する場合、ドライバは通常、その attach(9E) エントリポイントで次の手順を実行します。

1. ddi_intr_get_supported_types(9F) を使用して、サポートされている割り込みのタイプを判定します。
2. ddi_intr_get_nintrs(9F) を使用して、サポートされている MSI 割り込みタイプの数を判定します。
3. ddi_intr_alloc(9F) を使用して、MSI 割り込みにメモリーを割り当てます。
4. 割り当てる割り込みタイプごとに、次の手順を実行します。
   a. ddi_intr_get_pri(9F) を使用して、割り込みの優先順位を取得します。
   b. 割り込みに新しい優先順位を設定する必要がある場合は、ddi_intr_set_pri(9F) を使用します。
   c. mutex_init(9F) を使用して、ロックを初期化します。
   d. ddi_intr_add_handler(9F) を使用して、割り込みのハンドラを登録します。
5. 次の関数のいずれかを使用して、すべての割り込みを有効にします。
   - ブロック内のすべての割り込みを有効にするときは、ddi_intr_block_enable(9F) を使用します。
   - 各割り込みを個別に有効にするときは、ループで ddi_intr_enable(9F) を使用します。

例 8–7　一連の MSI 割り込みの登録

次の例は、mydev というデバイスの MSI 割り込みを登録する方法を示しています。

```
/* Get supported interrupt types */
if (ddi_intr_get_supported_types(devinfo, &intr_types) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "ddi_intr_get_supported_types failed");
    goto attach_fail;
}

if (intr_types & DDI_INTR_TYPE_MSI)
    mydev_add_msi_intrs(mydevp);

/* Check count, available and actual interrupts */
static int
mydev_add_msi_intrs(mydev_t *mydevp)
{
    dev_info_t    *devinfo = mydevp->devinfo;
    int           count, avail, actual;
    int           x, y, rc, inum = 0;

    /* Get number of interrupts */
    rc = ddi_intr_get_nintrs(devinfo, DDI_INTR_TYPE_MSI, &count);
    if ((rc != DDI_SUCCESS) || (count == 0)) {
```

例 8–7 一連のMSI割り込みの登録 *(続き)*

```
        cmn_err(CE_WARN, "ddi_intr_get_nintrs() failure, rc: %d, "
            "count: %d", rc, count);

        return (DDI_FAILURE);
    }
    /* Get number of available interrupts */
    rc = ddi_intr_get_navail(devinfo, DDI_INTR_TYPE_MSI, &avail);
    if ((rc != DDI_SUCCESS) || (avail == 0)) {
        cmn_err(CE_WARN, "ddi_intr_get_navail() failure, "
            "rc: %d, avail: %d\n", rc, avail);
        return (DDI_FAILURE);
    }
    if (avail < count) {
        cmn_err(CE_NOTE, "nitrs() returned %d, navail returned %d",
            count, avail);
    }
    /* Allocate memory for MSI interrupts */
    mydevp->intr_size = count * sizeof (ddi_intr_handle_t);
    mydevp->htable = kmem_alloc(mydevp->intr_size, KM_SLEEP);

    rc = ddi_intr_alloc(devinfo, mydevp->htable, DDI_INTR_TYPE_MSI, inum,
        count, &actual, DDI_INTR_ALLOC_NORMAL);

    if ((rc != DDI_SUCCESS) || (actual == 0)) {
        cmn_err(CE_WARN, "ddi_intr_alloc() failed: %d", rc);

        kmem_free(mydevp->htable, mydevp->intr_size);
        return (DDI_FAILURE);
    }

    if (actual < count) {
        cmn_err(CE_NOTE, "Requested: %d, Received: %d", count, actual);
    }

    mydevp->intr_cnt = actual;
    /*
     * Get priority for first msi, assume remaining are all the same
     */
    if (ddi_intr_get_pri(mydevp->htable[0], &mydev->intr_pri) !=
        DDI_SUCCESS) {
        cmn_err(CE_WARN, "ddi_intr_get_pri() failed");

        /* Free already allocated intr */
        for (y = 0; y < actual; y++) {
            (void) ddi_intr_free(mydevp->htable[y]);
        }

        kmem_free(mydevp->htable, mydevp->intr_size);
        return (DDI_FAILURE);
    }
    /* Call ddi_intr_add_handler() */
    for (x = 0; x < actual; x++) {
        if (ddi_intr_add_handler(mydevp->htable[x], mydev_intr,
           (caddr_t)mydevp, NULL) != DDI_SUCCESS) {
            cmn_err(CE_WARN, "ddi_intr_add_handler() failed");
```

例 8–7　一連の MSI 割り込みの登録　　(続き)

```
            /* Free already allocated intr */
            for (y = 0; y < actual; y++) {
                (void) ddi_intr_free(mydevp->htable[y]);
            }

            kmem_free(mydevp->htable, mydevp->intr_size);
            return (DDI_FAILURE);
        }
    }

    (void) ddi_intr_get_cap(mydevp->htable[0], &mydevp->intr_cap);
    if (mydev->m_intr_cap & DDI_INTR_FLAG_BLOCK) {
        /* Call ddi_intr_block_enable() for MSI */
        (void) ddi_intr_block_enable(mydev->m_htable, mydev->m_intr_cnt);
    } else {
        /* Call ddi_intr_enable() for MSI non block enable */
        for (x = 0; x < mydev->m_intr_cnt; x++) {
            (void) ddi_intr_enable(mydev->m_htable[x]);
        }
    }
    return (DDI_SUCCESS);
}
```

例 8–8　MSI 割り込みの削除

次の例は、MSI 割り込みを削除する方法を示しています。

```
static void
mydev_rem_intrs(mydev_t *mydev)
{
    int        x;

    /* Disable all interrupts */
    if (mydev->m_intr_cap & DDI_INTR_FLAG_BLOCK) {
        /* Call ddi_intr_block_disable() */
        (void) ddi_intr_block_disable(mydev->m_htable, mydev->m_intr_cnt);
    } else {
        for (x = 0; x < mydev->m_intr_cnt; x++) {
            (void) ddi_intr_disable(mydev->m_htable[x]);
        }
    }

    /* Call ddi_intr_remove_handler() */
    for (x = 0; x < mydev->m_intr_cnt; x++) {
        (void) ddi_intr_remove_handler(mydev->m_htable[x]);
        (void) ddi_intr_free(mydev->m_htable[x]);
    }

    kmem_free(mydev->m_htable, mydev->m_intr_size);
}
```

割り込みリソース管理

このセクションでは、多数の異なる割り込み条件を生成する可能性のあるデバイスのドライバで、割り込みリソース管理機能を利用して割り込みベクターの割り当てを最適化する方法について説明します。

割り込みリソース管理機能

割り込みリソース管理機能を使用すると、ドライバの割り込み構成を動的に管理することで、デバイスドライバで使用する割り込みリソースを増やすことができます。割り込みリソース管理機能を使用しないと、割り込み処理の構成は通常、ドライバの `attach` (9E) ルーチンでのみ行われます。割り込みリソース管理機能では、システムの変更を監視し、その変更に応答して各デバイスに許可する割り込みベクターの数を再計算して、ドライバの新しい割り込みベクター割り当ての影響を受ける、関連するドライバに通知します。関連するドライバとは、コールバックハンドラを登録したドライバのことです (142 ページの「コールバックのインタフェース」を参照)。割り込みベクターの再割り当ての原因になる可能性のある変更には、デバイスの追加や削除、明示的な要求が含まれます (146 ページの「要求する割り込みベクターの数の変更」を参照)。

割り込みリソース管理機能は、すべての Oracle Solaris プラットフォームで利用できるわけではありません。この機能は、MSI-X 割り込みを利用する PCIe デバイスでのみ利用できます。

注 – 割り込みリソース管理機能を利用するドライバは、この機能が利用できないときに適切に順応できる必要があります。

割り込みリソース管理機能が利用できる場合、ドライバは、この機能が利用できない場合に割り当てられるよりも多くの割り込みベクターにアクセスできます。ドライバで利用できる割り込みベクターの数が増えると、ドライバが割り込み条件を処理する効率が向上することがあります。

割り込みリソース管理機能では、次の制約に応じて、関連するドライバに許可される割り込みベクターの数を動的に調整します。

- 利用可能な合計数。システムでは割り込みベクターの数に制限があります。
- 要求される合計数。ドライバにはこれより少ない数を許可できますが、ドライバが要求した割り込みベクターよりも多くの割り込みベクターを許可することはできません。

- ほかのドライバとの公平性。利用可能な割り込みベクターの合計数は、各ドライバが要求する合計数と関連して公平な方法で、多数のドライバによって共有されます。

指定された任意の時点でデバイスで利用できるようになる割り込みベクターの数は、次の条件により異なることがあります。

- ほかのデバイスが動的にシステムに追加されたり削除されたりする
- ロードに応答してドライバが要求する割り込みベクターの数はドライバにより動的に変更される

ドライバは、割り込みリソース管理機能を利用するために次のサポートを提供する必要があります。

- コールバックサポート。ドライバは、利用可能な割り込みの数がシステムよって変更されたときに通知されるように、コールバックハンドラを登録する必要があります。ドライバは、割り込みの使用率を増やしたり減らしたりできる必要があります。
- 割り込みの要求。ドライバは、使用する割り込みの数を指定する必要があります。
- 割り込みの使用率。ドライバは、次の点に基づいて、任意の時点で正確な数の割り込みを要求する必要があります。
  - ドライバのハードウェアが生成する可能性のある割り込み条件
  - それらの条件を並列で処理するために使用できるプロセッサの数
- 割り込みの柔軟性。ドライバには、利用可能な割り込みの現在の数に最適な方法で、1つ以上の割り込み条件を各割り込みベクターに割り当てるために十分な柔軟性が必要です。ドライバでは、利用可能な割り込みの数が任意の時点で増減したときに、この割り当てを再構成する必要が生じる場合があります。

コールバックのインタフェース

ドライバは、次のインタフェースを使用してコールバックサポートを登録する必要があります。

表8–1 コールバックサポートのインタフェース

インタフェース	データ構造体	説明
`ddi_cb_register()`	`ddi_cb_flags_t`、`ddi_cb_handle_t`	コールバックハンドラ関数を登録して、特定のタイプのアクションを受信します。

表 8–1 コールバックサポートのインタフェース (続き)

インタフェース	データ構造体	説明
ddi_cb_unregister()	ddi_cb_handle_t	コールバックハンドラ関数の登録を解除します。
(*ddi_cb_func_t)()	ddi_cb_action_t	処理する各アクションに関連のあるコールバックアクションと特定の引数を受信します。

コールバックハンドラ関数の登録

ドライバのコールバックハンドラ関数を登録するときは、ddi_cb_register(9F) 関数を使用します。

```
int
ddi_cb_register (dev_info_t *dip, ddi_cb_flags_t cbflags,
                 ddi_cb_func_t cbfunc, void *arg1, void *arg2,
                 ddi_cb_handle_t *ret_hdlp);
```

ドライバは、1 つのコールバック関数のみを登録できます。この 1 つのコールバック関数が、個々のすべてのコールバックアクションを処理するために使用されます。cbflags パラメータは、アクションが発生したときにドライバが受信するアクションのタイプを判定します。cbfunc() ルーチンは、関連するアクションをドライバが処理するときに常に呼び出されます。ドライバは、cbfunc() ルーチンを実行するたびに自分に送信する 2 つのプライベート引数 (arg1 と arg2) を指定します。

cbflags() パラメータは、ドライバがサポートするアクションを指定する列挙型です。

```
typedef enum {
        DDI_CB_FLAG_INTR
} ddi_cb_flags_t;
```

割り込みリソース管理アクションのサポートを登録するには、ドライバでハンドラを登録して DDI_CB_FLAG_INTR フラグを含める必要があります。コールバックハンドラが正常に登録されると、ret_hdlp パラメータを介して不透明なハンドルが返されます。ドライバでコールバックハンドラの処理が終了すると、ドライバは ret_hdlp パラメータを使用してコールバックハンドラの登録を解除できます。

コールバックハンドラをドライバの attach(9F) エントリポイントで登録します。不透明なハンドルをドライバのソフト状態に保存します。ドライバの detach(9F) エントリポイントでコールバックハンドラの登録を解除します。

コールバックハンドラ関数の登録解除

ドライバのコールバックハンドラ関数の登録を解除するときは、ddi_cb_unregister(9F) 関数を使用します。

```
int
ddi_cb_unregister (ddi_cb_handle_t hdl);
```

この呼び出しは、ドライバの detach(9F) エントリポイントで行います。この呼び出しのあと、ドライバはコールバックアクションを受信しなくなります。

ドライバは、登録されたコールバック処理関数があることによって取得した、システムによる追加サポートも失います。たとえば、ドライバで前に利用できるようになった一部の割り込みベクターは、ドライバがコールバック処理関数の登録を解除するとすぐに元に戻されます。正常に復帰する前に、ddi_cb_unregister() 関数は、システムのサポートを失うことによって発生する、最終的なアクションをドライバに通知します。

コールバックハンドラ関数

コールバックアクションを受信し、処理する各アクションに固有の引数を受信するときは、登録されたコールバック処理関数を使用します。

```
typedef int (*ddi_cb_func_t)(dev_info_t *dip, ddi_cb_action_t cbaction,
                             void *cbarg, void *arg1, void *arg2);
```

cbaction パラメータは、ドライバがコールバックを受信するときの処理対象となるアクションを指定します。

```
typedef enum {
        DDI_CB_INTR_ADD,
        DDI_CB_INTR_REMOVE
} ddi_cb_action_t;
```

DDI_CB_INTR_ADD アクションは、ドライバで使用可能な割り込みが増えることを意味します。DDI_CB_INTR_REMOVE アクションは、ドライバで使用可能な割り込みが減ることを意味します。追加または削除された割り込みの数を判定するときは、cbarg パラメータを int にキャストします。cbarg 値は、利用可能な割り込みの数の変化を表します。

たとえば、利用可能な割り込みの数の変化を取得する場合は、次のようになります。

```
count = (int)(uintptr_t)cbarg;
```

cbaction が DDI_CB_INTR_ADD である場合は、cbarg 個の割り込みベクターを追加します。cbaction が DDI_CB_INTR_REMOVE である場合は、cbarg 個の割り込みベクターを解放します。

arg1 および arg2 の説明については、ddi_cb_register(9F) を参照してください。

コールバック処理関数は、関数が登録される期間全体を通じて、正しく実行できる必要があります。コールバック関数は、コールバック関数の登録が正常に解除される前に破棄される可能性のあるデータ構造体に依存することはできません。

コールバック処理関数は、次のいずれかの値を返す必要があります。

- アクションを正しく処理した場合は DDI_SUCCESS
- 内部エラーが発生した場合は DDI_FAILURE
- 認識できないアクションを受信した場合は DDI_ENOTSUP

割り込み要求のインタフェース

ドライバは、次のインタフェースを使用して、システムからの割り込みベクターを要求する必要があります。

表 8–2 割り込みベクター要求のインタフェース

インタフェース	データ構造体	説明
ddi_intr_alloc()	ddi_intr_handle_t	割り込みを割り当てます。
ddi_intr_set_nreq()		要求する割り込みベクターの数を変更します。

割り込みの割り当て

最初に割り込みを割り当てるときは、ddi_intr_alloc(9F) 関数を使用します。

```
int
ddi_intr_alloc (dev_info_t *dip, ddi_intr_handle_t *h_array, int type,
                int inum, int count, int *actualp, int behavior);
```

ドライバは、この関数を呼び出す前に、要求する数の割り込みを格納するのに十分な大きさの空のハンドル配列を割り当てる必要があります。ddi_intr_alloc() 関数は、count 個の割り込みハンドルを割り当てて、inum パラメータで指定されたオフセットから始まる割り当て済みの割り込みベクターを使って配列を初期化しようとします。actualp パラメータは、割り当てられた割り込みベクターの実際の数を返します。

ドライバは、次の 2 つの方法で ddi_intr_alloc() 関数を使用できます。

- ドライバは、ddi_intr_alloc() 関数を複数回呼び出して、割り込みハンドル配列の個々のメンバーに別々の手順で割り込みベクターを割り当てることができます。
- ドライバは、ddi_intr_alloc() 関数を 1 回呼び出して、デバイスのすべての割り込みベクターを一度に割り当てることができます。

割り込みリソース管理機能を使用している場合は、 ddi_intr_alloc() を 1 回呼び出して、すべての割り込みベクターを一度に割り当てます。count パラメータは、ドライバが要求する割り込みベクターの合計数です。actualp の値が count の値より小さい場合、システムは要求を完全に満たすことはできません。割り込みリソース管理

機能では、この要求 (count が nreq になる - 下記を参照) を保存するため、あとでさらに多くの割り込みベクターをこのドライバに割り当てることができるようになる場合があります。

注 – 割り込みリソース管理機能を使用すると、ddi_intr_alloc() を追加で呼び出しても、要求される割り込みベクターの合計数は変化しません。要求する割り込みベクターの数を変更するときは、ddi_intr_set_nreq(9F) 関数を使用します。

要求する割り込みベクターの数の変更

要求する割り込みベクターの数を変更するときは、ddi_intr_set_nreq(9F) 関数を使用します。

```
int
ddi_intr_set_nreq (dev_info_t *dip, int nreq);
```

割り込みリソース管理機能が利用できるとき、ドライバは ddi_intr_set_nreq() 関数を使用して、要求する割り込みベクターの合計数を動的に調整できます。ドライバは、ドライバが接続されると存在するようになる実際のロードに応答してこれを行うことができます。

ドライバは、最初に ddi_intr_alloc(9F) を呼び出して、割り込みベクターの最初の数を要求する必要があります。ddi_intr_alloc() 呼び出しのあとで、ドライバは ddi_intr_set_nreq() を呼び出してその要求サイズをいつでも変更できます。指定した nreq 値は、ドライバの、要求する割り込みベクターの新しい合計数です。割り込みリソース管理機能では、この新しい要求に応答して、システムの各ドライバに割り当てられた割り込みの数のバランスをとり直すことがあります。割り込みリソース管理機能によって、ドライバに割り当てられた割り込みの数のバランスがとり直されると、影響を受ける各ドライバは常に、ドライバが使用可能な割り込みベクターが増えたまたは減ったというコールバック通知を受信します。

たとえば、処理中の特定のトランザクションに関連して割り込みを使用する場合、ドライバは要求する割り込みベクターの合計数を動的に調整することがあります。ストレージドライバは、DMA エンジンを進行中の各トランザクションに関連付け、その理由で割り込みベクターを要求することがあります。ドライバは、open(9F) および close(9F) ルーチンで ddi_intr_set_nreq() を呼び出して、ドライバの実際の使用に応じて割り込みの使用率を調整することがあります。

割り込みの使用率と柔軟性

多数の異なる割り込み条件をサポートするデバイスのドライバは、それらの条件を任意の数の割り込みベクターにマッピングできる必要があります。そのようなドライバは、割り当てられる割り込みベクターが利用可能なままであると想定することはできません。現在利用可能な一部の割り込みは、システム内のほかのドライバの必要に対応するため、あとでシステムによって元に戻されることがあります。

ドライバは、次のことができる必要があります。

- ハードウェアがサポートする割り込みの数を判定する。
- 使用に適した割り込みの数を判定する。たとえば、システムのプロセッサの合計数がこの評価に影響することがあります。
- 必要な割り込みの数と、指定された任意の時点で利用可能な割り込みの数とを比較する。

まとめると、ドライバは、一連の割り込み処理関数を選択し、ハードウェアをプログラムして必要性と割り込みの可用性に応じて割り込みを生成できる必要があります。場合によっては、複数の割り込みが同じベクターのターゲットになることがあります。また、その割り込みベクターの割り込みハンドラは、発生した割り込みを判定する必要があります。デバイスのパフォーマンスは、ドライバが割り込みベクターに割り込みをマッピングする能力の影響を受けることがあります。

割り込みリソース管理の実装例

割り込みリソース管理の好例となるデバイスドライバのタイプの 1 つは、ネットワークデバイスのドライバです。ネットワークデバイスのハードウェアは、複数の送受信チャネルをサポートしています。

ネットワークデバイスは、デバイスがいずれかの受信チャネルでパケットを受信するか、いずれかの送信チャネルでパケットを送信すると常に、一意の割り込み条件を生成します。ハードウェアは、発生する可能性のあるイベントごとに特定の MSI-X 割り込みを送信できます。ハードウェア内のテーブルによって、イベントごとに生成する MSI-X 割り込みが判定されます。

パフォーマンスを最適化するために、ドライバはシステムからの十分な割り込みを要求して、別個の各割り込みに独自の割り込みベクターを提供します。ドライバは、`attach`(9F) ルーチンで最初に `ddi_intr_alloc`(9F) を呼び出すときにこの要求を行います。

次に、ドライバは、`actualp` の `ddi_intr_alloc()` から受信した割り込みの実際の数を評価します。ドライバは、要求したすべての割り込みを受信することもあり、要求より少ない割り込みを受信することもあります。

ドライバ内の別個の関数は、利用可能な割り込みの合計数を使用して、イベントごとに生成する MSI-X 割り込みを計算します。この関数は、その結果に従ってハードウェア内のテーブルをプログラムします。

- 要求したすべての割り込みベクターをドライバが受信した場合、ハードウェアのテーブル内の各エントリに、独自の一意の MSI-X 割り込みがあることになります。割り込み条件と割り込みベクターの間には、1 対 1 のマッピングが存在します。ハードウェアは、イベントのタイプごとに一意の MSI-X 割り込みを生成します。
- ドライバが持つ利用可能な割り込みベクターが少ない場合は、一部の MSI-X 割り込み番号をハードウェアのテーブルで複数回使用する必要があります。ハードウェアは、複数のタイプのイベントで同じ MSI-X 割り込みを生成します。

ドライバには、2 つの異なる割り込みハンドラ関数があるはずです。

- 1 つの割り込みハンドラ関数は、割り込みに応答して特定のタスクを実行します。この単純な関数が、可能性のあるハードウェアイベントのうちただ 1 つのイベントによって生成される割り込みを処理します。
- もう 1 つの割り込みハンドラ関数はさらに複雑です。この関数は、複数の割り込みが同じ MSI-X 割り込みベクターにマッピングされる場合を扱います。

このセクションのドライバ例では、関数 `xx_setup_interrupts()` は、利用可能な割り込みベクターの数を使用してハードウェアをプログラムし、それらの割り込みベクターごとに適切な割り込みハンドラを呼び出します。`xx_setup_interrupts()` 関数は、2 か所で呼び出されます。`ddi_intr_alloc()` が `xx_attach ()` で呼び出されたあとと、割り込みベクター割り当てが `xx_cbfunc()` コールバックハンドラ関数で調整されたあとです。

```
int
xx_setup_interrupts(xx_state_t *statep, int navail, xx_intrs_t *xx_intrs_p);
```

`xx_setup_interrupts()` 関数は、`xx_intrs_t` データ構造体の配列を使って呼び出されます。

```
typedef struct {
        ddi_intr_handler_t      inthandler;
        void                    *arg1;
        void                    *arg2;
} xx_intrs_t;
```

この `xx_setup_interrupts()` 機能は、割り込みリソース管理機能が利用できるかどうかに関係なくドライバに存在している必要があります。ドライバは、接続時に要求する数より少ない割り込みベクターで機能できる必要があります。割り込みリソース管理機能が利用できる場合は、利用可能な割り込みベクターの新しい数に動的に調整されるようにドライバを変更できます。

割り込みリソース管理機能が利用できるかどうかに関係なくドライバが提供する必要のある別の機能には、ハードウェアを休止および再開する機能が含まれます。休

止と再開は、電源管理とホットプラグによる取り付けに関係する特定のイベントに必要です。休止と再開は、割り込みコールバックアクションの処理にも必要です。

休止関数は、xx_detach() で呼び出されます。

```
int
xx_quiesce(xx_state_t *statep);
```

再開関数は、xx_attach() で呼び出されます。

```
int
xx_resume(xx_state_t *statep);
```

このデバイスドライバを拡張して割り込みリソース管理機能を使用するときは、次の変更を行います。

- コールバックハンドラの登録。ドライバは、利用可能な割り込みが多い時と少ない時を示すアクションのために登録を行う必要があります。
- コールバックの処理。ドライバは、コールバックアクションに応答してハードウェアを休止し、割り込み処理をプログラムし、ハードウェアを再開する必要があります。

```
/*
 * attach(9F) routine.
 *
 * Creates soft state, registers callback handler, initializes
 * hardware, and sets up interrupt handling for the driver.
 */
xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        xx_state_t              *statep = NULL;
        xx_intr_t               *intrs = NULL;
        ddi_intr_handle_t       *hdls;
        ddi_cb_handle_t         cb_hdl;
        int                     instance;
        int                     type;
        int                     types;
        int                     nintrs;
        int                     nactual;
        int                     inum;

        /* Get device instance */
        instance = ddi_get_instance(dip);

        switch (cmd) {
        case DDI_ATTACH:

                /* Get soft state */
                if (ddi_soft_state_zalloc(state_list, instance) != 0)
                        return (DDI_FAILURE);
                statep = ddi_get_soft_state(state_list, instance);
                ddi_set_driver_private(dip, (caddr_t)statep);
                statep->dip = dip;
```

```
                /* Initialize hardware */
                xx_initialize(statep);

                /* Register callback handler */
                if (ddi_cb_register(dip, DDI_CB_FLAG_INTR, xx_cbfunc,
                    statep, NULL, &cb_hdl) != 0) {
                        ddi_soft_state_free(state_list, instance);
                        return (DDI_FAILURE);
                }
                statep->cb_hdl = cb_hdl;

                /* Select interrupt type */
                ddi_intr_get_supported_types(dip, &types);
                if (types & DDI_INTR_TYPE_MSIX) {
                        type = DDI_INTR_TYPE_MSIX;
                } else if (types & DDI_INTR_TYPE_MSI) {
                        type = DDI_INTR_TYPE_MSI;
                } else {
                        type = DDI_INTR_TYPE_FIXED;
                }
                statep->type = type;

                /* Get number of supported interrupts */
                ddi_intr_get_nintrs(dip, type, &nintrs);

                /* Allocate interrupt handle array */
                statep->hdls_size = nintrs * sizeof (ddi_intr_handle_t);
                statep->hdls = kmem_zalloc(statep->hdls_size, KMEM_SLEEP);

                /* Allocate interrupt setup array */
                statep->intrs_size = nintrs * sizeof (xx_intr_t);
                statep->intrs = kmem_zalloc(statep->intrs_size, KMEM_SLEEP);

                /* Allocate interrupt vectors */
                ddi_intr_alloc(dip, hdls, type, 0, nintrs, &nactual, 0);
                statep->nactual = nactual;

                /* Configure interrupt handling */
                xx_setup_interrupts(statep, statep->nactual, statep->intrs);

                /* Install and enable interrupt handlers */
                for (inum = 0; inum < nactual; inum++) {
                        ddi_intr_add_handler(&hdls[inum],
                            intrs[inum].inthandler,
                            intrs[inum].arg1, intrs[inum].arg2);
                        ddi_intr_enable(hdls[inum]);
                }

                break;

        case DDI_RESUME:

                /* Get soft state */
                statep = ddi_get_soft_state(state_list, instance);
                if (statep == NULL)
                        return (DDI_FAILURE);

                /* Resume hardware */
                xx_resume(statep);
```

```
                break;
        }

        return (DDI_SUCESS);
}

/*
 * detach(9F) routine.
 *
 * Stops the hardware, disables interrupt handling, unregisters
 * a callback handler, and destroys the soft state for the driver.
 */
xx_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
        xx_state_t      *statep = NULL;
        int             instance;
        int             inum;

        /* Get device instance */
        instance = ddi_get_instance(dip);

        switch (cmd) {
        case DDI_DETACH:

                /* Get soft state */
                statep = ddi_get_soft_state(state_list, instance);
                if (statep == NULL)
                        return (DDI_FAILURE);

                /* Stop device */
                xx_uninitialize(statep);

                /* Disable and free interrupts */
                for (inum = 0; inum < statep->nactual; inum++) {
                        ddi_intr_disable(statep->hdls[inum]);
                        ddi_intr_remove_handler(statep->hdls[inum]);
                        ddi_intr_free(statep->hdls[inum]);
                }

                /* Unregister callback handler */
                ddi_cb_unregister(statep->cb_hdl);

                /* Free interrupt handle array */
                kmem_free(statep->hdls, statep->hdls_size);

                /* Free interrupt setup array */
                kmem_free(statep->intrs, statep->intrs_size);

                /* Free soft state */
                ddi_soft_state_free(state_list, instance);

                break;

        case DDI_SUSPEND:

                /* Get soft state */
                statep = ddi_get_soft_state(state_list, instance);
                if (statep == NULL)
```

```
                        return (DDI_FAILURE);

                /* Suspend hardware */
                xx_quiesce(statep);

                break;
        }

        return (DDI_SUCCESS);
}

/*
 * (*ddi_cbfunc)() routine.
 *
 * Adapt interrupt usage when availability changes.
 */
int
xx_cbfunc(dev_info_t *dip, ddi_cb_action_t cbaction, void *cbarg,
    void *arg1, void *arg2)
{
        xx_state_t      *statep = (xx_state_t *)arg1;
        int             count;
        int             inum;
        int             nactual;

        switch (cbaction) {
        case DDI_CB_INTR_ADD:
        case DDI_CB_INTR_REMOVE:

                /* Get change in availability */
                count = (int)(uintptr_t)cbarg;

                /* Suspend hardware */
                xx_quiesce(statep);

                /* Tear down previous interrupt handling */
                for (inum = 0; inum < statep->nactual; inum++) {
                        ddi_intr_disable(statep->hdls[inum]);
                        ddi_intr_remove_handler(statep->hdls[inum]);
                }

                /* Adjust interrupt vector allocations */
                if (cbaction == DDI_CB_INTR_ADD) {

                        /* Allocate additional interrupt vectors */
                        ddi_intr_alloc(dip, statep->hdls, statep->type,
                            statep->nactual, count, &nactual, 0);

                        /* Update actual count of available interrupts */
                        statep->nactual += nactual;

                } else {

                        /* Free removed interrupt vectors */
                        for (inum = statep->nactual - count;
                            inum < statep->nactual; inum++) {
                                ddi_intr_free(statep->hdls[inum]);
                        }
```

```
                        /* Update actual count of available interrupts */
                        statep->nactual -= count;
                }

                /* Configure interrupt handling */
                xx_setup_interrupts(statep, statep->nactual, statep->intrs);

                /* Install and enable interrupt handlers */
                for (inum = 0; inum < statep->nactual; inum++) {
                        ddi_intr_add_handler(&statep->hdls[inum],
                            statep->intrs[inum].inthandler,
                            statep->intrs[inum].arg1,
                            statep->intrs[inum].arg2);
                        ddi_intr_enable(statep->hdls[inum]);
                }

                /* Resume hardware */
                xx_resume(statep);

                break;

        default:
                return (DDI_ENOTSUP);
        }

        return (DDI_SUCCESS);
}
```

割り込みハンドラの機能

ドライバのフレームワークとデバイスは、それぞれが割り込みハンドラに要求を提示します。すべての割り込みハンドラは、次のタスクを実行する必要があります。

- デバイスが割り込みを行っているかどうかを判定する。また、割り込みを拒否する場合もある。

  割り込みハンドラは、まずデバイスを調べて、そのデバイスが割り込みを発行したかどうかを判定します。そのデバイスが割り込みを発行しなかった場合、ハンドラは`DDI_INTR_UNCLAIMED`を返す必要があります。この手順により、デバイスポーリングの実装が可能になります。指定された割り込み優先順位レベルのデバイスは、割り込みを発行した可能性があります。デバイスポーリングにより、デバイスが割り込みを発行したかどうかがシステムに通知されます。

- デバイスが保守されていることをデバイスに通知する。

  保守についてデバイスに通知することは、大部分のデバイスに必要となる、デバイス固有の処理です。たとえば、SBus デバイスは、ドライバが SBus デバイスに停止を指示するまで割り込みを行う必要があります。この方法により、同じ優先順位レベルで割り込みを行うすべての SBus デバイスが保守されることが保証されます。

- 任意の入出力リクエスト関連の処理を実行する。

デバイスは、転送完了や転送エラーなど、さまざまな理由で割り込みを行います。この手順は、必要に応じてデバイスのデータバッファーを読み取り、デバイスのエラーレジスタを調べ、データ構造体のステータスフィールドを設定するために、データアクセス関数の使用を伴うことがあります。割り込みのディスパッチや処理には比較的時間がかかります。

- 別の割り込みを妨げる可能性のある追加の処理を行う。

  たとえば、デバイスから次のデータ項目を読み取ります。

- `DDI_INTR_CLAIMED` を返す。
- **MSI** 割り込みを常に取り込む必要がある。

  MSI-X 割り込みの場合、割り込みの取り込みは任意です。どちらの場合でも、割り込みの所有権を確認する必要はありません。これは、MSI 割り込みと MSI-X 割り込みはほかのデバイスと共有されないためです。

- ホットプラグによる取り付けと複数の **MSI** または **MSI-X** 割り込みをサポートするドライバが、ホットプラグイベント用の別個の割り込みを保持し、その割り込み用の別個の **ISR (**割り込みサービスルーチン**)** を登録するようにする。

次の例は、`mydev` というデバイスの割り込みルーチンを示しています。

例 **8–9** 割り込みの例

```
static uint_t
mydev_intr(caddr_t arg1, caddr_t arg2)
{
    struct mydevstate *xsp = (struct mydevstate *)arg1;
    uint8_t     status;
    volatile  uint8_t  temp;

    /*
     * Claim or reject the interrupt.This example assumes
     * that the device's CSR includes this information.
     */
    mutex_enter(&xsp->high_mu);
    /* use data access routines to read status */
    status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
    if (!(status & INTERRUPTING)) {
        mutex_exit(&xsp->high_mu);
        return (DDI_INTR_UNCLAIMED); /* dev not interrupting */
    }
    /*
     * Inform the device that it is being serviced, and re-enable
     * interrupts. The example assumes that writing to the
     * CSR accomplishes this. The driver must ensure that this data
     * access operation makes it to the device before the interrupt
     * service routine returns. For example, using the data access
     * functions to read the CSR, if it does not result in unwanted
     * effects, can ensure this.
     */
    ddi_put8(xsp->data_access_handle, &xsp->regp->csr,
        CLEAR_INTERRUPT | ENABLE_INTERRUPTS);
    /* flush store buffers */
    temp = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
```

例 8–9 割り込みの例 (続き)

```
    mutex_exit(&xsp->mu);
    return (DDI_INTR_CLAIMED);
}
```

割り込みルーチンによって実行される手順のほとんどは、デバイス自体の詳細によって異なります。割り込みの原因の判定、エラー条件の検出、およびデバイスのデータレジスタへのアクセスについては、そのデバイスのハードウェアマニュアルを参照してください。

高レベルの割り込みの処理

高レベルの割り込みとは、スケジューラおよびそれより上のレベルで行われる割り込みのことです。このレベルでは、スケジューラは実行できません。したがって、スケジューラであらかじめ高レベルの割り込みハンドラを無効にすることはできません。スケジューラが原因となって、高レベルの割り込みをブロックできません。高レベルの割り込みで可能なのは、相互排他ロックを使ってロックすることのみです。

ドライバは、デバイスが高レベルの割り込みを使用しているかどうかを判定する必要があります。このテストは、割り込みの登録時にドライバの `attach`(9E) エントリポイントで行います。156 ページの「高レベルの割り込み処理の例」を参照してください。

- `ddi_intr_get_pri`(9F) から返される割り込み優先順位が `ddi_intr_get_hilevel_pri`(9F) から返される優先順位以上である場合、ドライバは接続に失敗することがあります。または、ドライバは高レベルの割り込みハンドラを実装する可能性があります。高レベルの割り込みハンドラは、優先順位の低いソフトウェア割り込みを使用してデバイスを扱います。許可する並行処理の程度を上げるには、別個の mutex を使ってデータを高レベルのハンドラから保護します。
- `ddi_intr_get_pri`(9F) から返される割り込み優先順位が `ddi_intr_get_hilevel_pri`(9F) から返される優先順位より低い場合、`attach`(9E) エントリポイントは通常の割り込み登録になります。この場合、ソフト割り込みは必要ありません。

高レベルの mutex

高レベルの割り込みを表す割り込み優先順位で初期化された mutex は、高レベルの *mutex* と呼ばれます。ドライバは、高レベルの mutex を保持している間、高レベルの割り込みハンドラと同じ制限に従います。

高レベルの割り込み処理の例

次の例では、高レベルの mutex (xsp->high_mu) は、高レベルの割り込みハンドラとソフト割り込みハンドラの間で共有されるデータを保護するためにのみ使用されます。保護されるデータには、高レベルの割り込みハンドラと低レベルのハンドラの両方で使用されるキューと、低レベルのハンドラが実行されていることを示すフラグが含まれています。別個の低レベルの mutex (xsp->low_mu) は、ドライバの残りの部分をソフト割り込みハンドラから保護します。

例 8–10 attach() を使用した高レベルの割り込みの処理

```
static int
mydevattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    struct mydevstate *xsp;
    /* ... */

    ret = ddi_intr_get_supported_types(dip, &type);
    if ((ret != DDI_SUCCESS) || (!(type & DDI_INTR_TYPE_FIXED))) {
        cmn_err(CE_WARN, "ddi_intr_get_supported_types() failed");
        return (DDI_FAILURE);
    }

    ret = ddi_intr_get_nintrs(dip, DDI_INTR_TYPE_FIXED, &count);

    /*
     * Fixed interrupts can only have one interrupt. Check to make
     * sure that number of supported interrupts and number of
     * available interrupts are both equal to 1.
     */
    if ((ret != DDI_SUCCESS) || (count != 1)) {
    cmn_err(CE_WARN, "No fixed interrupts found");
            return (DDI_FAILURE);
    }

    xsp->xs_htable = kmem_zalloc(count * sizeof (ddi_intr_handle_t),
        KM_SLEEP);

    ret = ddi_intr_alloc(dip, xsp->xs_htable, DDI_INTR_TYPE_FIXED, 0,
        count, &actual, 0);

    if ((ret != DDI_SUCCESS) || (actual != 1)) {
    cmn_err(CE_WARN, "ddi_intr_alloc failed 0x%x", ret");
        kmem_free(xsp->xs_htable, sizeof (ddi_intr_handle_t));
        return (DDI_FAILURE);
    }

    ret = ddi_intr_get_pri(xsp->xs_htable[0], &intr_pri);
    if (ret != DDI_SUCCESS) {
        cmn_err(CE_WARN, "ddi_intr_get_pri failed 0x%x", ret");
        (void) ddi_intr_free(xsp->xs_htable[0]);
        kmem_free(xsp->xs_htable, sizeof (ddi_intr_handle_t));
        return (DDI_FAILURE);
    }

    if (intr_pri >= ddi_intr_get_hilevel_pri()) {
```

例 8–10 attach() を使用した高レベルの割り込みの処理 (続き)

```
    mutex_init(&xsp->high_mu, NULL, MUTEX_DRIVER,
        DDI_INTR_PRI(intr_pri));

    ret = ddi_intr_add_handler(xsp->xs_htable[0],
        mydevhigh_intr, (caddr_t)xsp, NULL);
    if (ret != DDI_SUCCESS) {
        cmn_err(CE_WARN, "ddi_intr_add_handler failed 0x%x", ret");
        mutex_destroy(&xsp>xs_int_mutex);
            (void) ddi_intr_free(xsp->xs_htable[0]);
            kmem_free(xsp->xs_htable, sizeof (ddi_intr_handle_t));
        return (DDI_FAILURE);
    }

    /* add soft interrupt */
    if (ddi_intr_add_softint(xsp->xs_dip, &xsp->xs_softint_hdl,
        DDI_INTR_SOFTPRI_MAX, xs_soft_intr, (caddr_t)xsp) !=
        DDI_SUCCESS) {
        cmn_err(CE_WARN, "add soft interrupt failed");
        mutex_destroy(&xsp->high_mu);
        (void) ddi_intr_remove_handler(xsp->xs_htable[0]);
            (void) ddi_intr_free(xsp->xs_htable[0]);
            kmem_free(xsp->xs_htable, sizeof (ddi_intr_handle_t));
        return (DDI_FAILURE);
    }


    xsp->low_soft_pri = DDI_INTR_SOFTPRI_MAX;

    mutex_init(&xsp->low_mu, NULL, MUTEX_DRIVER,
        DDI_INTR_PRI(xsp->low_soft_pri));

    } else {
    /*
     * regular interrupt registration continues from here
     * do not use a soft interrupt
     */
    }

    return (DDI_SUCCESS);
}
```

高レベルの割り込みルーチンは、デバイスを保守し、データをキューに入れます。低レベルのルーチンが実行されていない場合、高レベルのルーチンはソフトウェア割り込みをトリガーします。次に例を示します。

例 8–11 高レベルの割り込みルーチン

```
static uint_t
mydevhigh_intr(caddr_t arg1, caddr_t arg2)
{
   struct mydevstate    *xsp = (struct mydevstate *)arg1;
   uint8_t    status;
   volatile  uint8_t  temp;
```

例 8–11 高レベルの割り込みルーチン (続き)

```
   int    need_softint;

    mutex_enter(&xsp->high_mu);
    /* read status */
    status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
    if (!(status & INTERRUPTING)) {
        mutex_exit(&xsp->high_mu);
        return (DDI_INTR_UNCLAIMED); /* dev not interrupting */
    }

    ddi_put8(xsp->data_access_handle,&xsp->regp->csr,
        CLEAR_INTERRUPT | ENABLE_INTERRUPTS);
    /* flush store buffers */
    temp = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);

    /* read data from device, queue data for low-level interrupt handler */

    if (xsp->softint_running)
        need_softint = 0;
    else {
        xsp->softint_count++;
        need_softint = 1;
    }
    mutex_exit(&xsp->high_mu);

    /* read-only access to xsp->id, no mutex needed */
    if (need_softint) {
        ret = ddi_intr_trigger_softint(xsp->xs_softint_hdl, NULL);
        if (ret == DDI_EPENDING) {
            cmn_err(CE_WARN, "ddi_intr_trigger_softint() soft interrupt "
                "already pending for this handler");
        } else if (ret != DDI_SUCCESS) {
            cmn_err(CE_WARN, "ddi_intr_trigger_softint() failed");
        }
    }

    return (DDI_INTR_CLAIMED);
}
```

低レベルの割り込みルーチンは、ソフトウェア割り込みをトリガーする高レベルの割り込みルーチンによって開始されます。低レベルの割り込みルーチンは、処理するものがなくなるまで実行されます。次に例を示します。

例 8–12 低レベルのソフト割り込みルーチン

```
static uint_t
mydev_soft_intr(caddr_t arg1, caddr_t arg2)
{
    struct mydevstate *mydevp = (struct mydevstate *)arg1;
    /* ... */
    mutex_enter(&mydevp->low_mu);
    mutex_enter(&mydevp->high_mu);
    if (mydevp->softint_count > 1) {
        mydevp->softint_count--;
```

例 8–12 低レベルのソフト割り込みルーチン (続き)

```
        mutex_exit(&mydevp->high_mu);
        mutex_exit(&mydevp->low_mu);
        return (DDI_INTR_CLAIMED);
    }

    if ( /* queue empty */ ) {
        mutex_exit(&mydevp->high_mu);
        mutex_exit(&mydevp->low_mu);
        return (DDI_INTR_UNCLAIMED);
    }

    mydevp->softint_running = 1;
    while (EMBEDDED COMMENT:data on queue) {
        ASSERT(mutex_owned(&mydevp->high_mu);
        /* Dequeue data from high-level queue. */
        mutex_exit(&mydevp->high_mu);
        /* normal interrupt processing */
        mutex_enter(&mydevp->high_mu);
    }

    mydevp->softint_running = 0;
    mydevp->softint_count = 0;
    mutex_exit(&mydevp->high_mu);
    mutex_exit(&mydevp->low_mu);
    return (DDI_INTR_CLAIMED);
}
```

◆ ◆ ◆ 第 9 章

9 ダイレクトメモリーアクセス (DMA)

多くのデバイスは、バスを一時的に制御できます。これらのデバイスは、メインメモリーなどのデバイスを必要とするデータ転送を行うことができます。デバイスは CPU の助けを借りずにその処理を行なっているため、このようなデータ転送をダイレクトメモリーアクセス (Direct Memory Access、DMA) と呼びます。実行できる DMA 転送の種類は次のとおりです。

- 2 つのデバイス間
- デバイスとメモリー間
- メモリーとメモリー間

この章では、デバイスとメモリー間の転送についてのみ説明します。この章では、次の内容について説明します。

DMA モデル

DDI/DKI (Oracle Solaris Device Driver Interface/Driver-Kernel Interface) には、アーキテクチャーに依存しないハイレベルな DMA モデルが用意されています。このモデルを使用すると、フレームワーク (つまり DMA ルーチン) で次のようなアーキテクチャー固有の詳細を隠すことができます。

- DMA マッピングの設定
- scatter/gather リストの作成

- 入出力と CPU 間のキャッシュの一貫性の確保

DDI/DKI では、DMA トランザクションのさまざまな局面を記述するためにいくつかの抽象化が使用されます。

- **DMA** オブジェクト – DMA 転送の転送元または転送先になるメモリー。
- **DMA** ハンドル – 正常に実行された `ddi_dma_alloc_handle(9F)` 呼び出しから返される不透明なオブジェクト。DMA ハンドルをそのあとの DMA サブルーチン呼び出しで使用すると、そのような DMA オブジェクトを参照できます。
- **DMA cookie** – `ddi_dma_cookie(9S)` 構造体 (`ddi_dma_cookie_t`) では、デバイスから完全にアドレス可能な DMA オブジェクトの連続した部分を記述します。cookie には、DMA エンジンのプログラミングに必要な DMA アドレッシング情報が含まれています。

デバイスドライバは、オブジェクトを直接メモリー内にマップするのではなく、DMA リソースをメモリーオブジェクトに割り当てます。次に、DMA ルーチンは DMA アクセス用のオブジェクトの設定に必要なプラットフォーム固有のすべての操作を実行します。ドライバは DMA ハンドルを受け取って、オブジェクトに割り当てられている DMA リソースを特定します。このハンドルはデバイスドライバに対して不透明です。ドライバはこのハンドルを保存し、そのあとの呼び出しで DMA ルーチンに渡す必要があります。ドライバがこのハンドルを解釈することはありません。

次のサービスを提供する処理は DMA ハンドルで定義されます。

- DMA リソースの操作
- DMA オブジェクトの同期
- 割り当てられたリソースの属性の取り出し

デバイス DMA の種類

デバイスは、次の 3 種類の DMA のいずれかを実行します。

- バスマスター DMA
- サードパーティー DMA
- ファーストパーティー DMA

バスマスター DMA

ドライバは、デバイスが本物のバスマスターのように動作する場合に、そのデバイスの DMA レジスタを直接プログラミングします。たとえば、DMA エンジンがデバイスボード上にある場合、デバイスはバスマスターのように動作します。転送のアドレスとカウントが DMA cookie から取得されてデバイスに渡されます。

サードパーティー DMA

サードパーティー DMA はメインシステムボード上にあるシステムの DMA エンジンを使用します。メインシステムボードには、デバイスで使用できる DMA チャネルがいくつか備わっています。デバイスとメモリー間のデータ転送には、システムの DMA エンジンが使用されています。ドライバは、DMA エンジンのルーチン (ddi_dmae(9F) 関数を参照) を使用して DMA エンジンの初期化とプログラミングを行います。DMA データ転送ごとに、ドライバは DMA エンジンをプログラミングし、そのエンジンと連携して転送を開始するためのコマンドをデバイスに提供します。

ファーストパーティー DMA

ファーストパーティー DMA では、デバイスはシステムの DMA エンジンのチャネルを使用して、そのデバイスの DMA バスサイクルを作動させます。ddi_dmae_1stparty(9F) 関数を使用して、DMA エンジンが転送の妨げにならないように、カスケードモードでこのチャネルを構成します。

ホストプラットフォームの DMA の種類

デバイスが稼働するプラットフォームでは、ダイレクトメモリーアクセス (DMA) または DVMA (Direct Virtual Memory Access) を提供します。

DMA をサポートするプラットフォームでは、転送を行うために物理アドレスがデバイスに提供されます。この場合、DMA オブジェクトの転送は実際には物理的に不連続ないくつかの転送で構成されていることがあります。たとえば、アプリケーションで、連続したいくつかの仮想ページにまたがるバッファーを転送する際、それらの仮想ページが物理的に不連続なページにマップされるような場合です。不連続なメモリーを処理するために、通常これらのプラットフォームのデバイスにはある種の scatter/gather DMA 機能が備わっています。通常、x86 システムではダイレクトメモリー転送に物理アドレスを提供します。

DVMA をサポートするプラットフォームでは、転送を行うために仮想アドレスがデバイスに提供されます。この場合、ベースとなるプラットフォームが提供するメモリー管理ユニット (Memory Management Unit、MMU) は、これらの仮想アドレスへのデバイスアクセスを適切な物理アドレスに変換します。デバイスは、不連続な物理ページにマップされる可能性のある連続した仮想イメージに対して転送を行います。これらのプラットフォームで稼働するデバイスには、scatter/gather DMA 機能は必要ありません。通常、SPARC プラットフォームではダイレクトメモリー転送に仮想アドレスを提供します。

DMA ソフトウェアコンポーネント:ハンドル、ウィンドウ、cookie

DMA ハンドルは、オブジェクト (通常はメモリーバッファーやメモリーアドレス) を表す隠されたポインタです。DMA ハンドルを使用すると、デバイスは DMA 転送を行うことができます。さまざまな DMA ルーチンの呼び出しで、このハンドルを使用して、オブジェクトに割り当てられている DMA リソースを特定しています。

DMA ハンドルで表されるオブジェクトは、1 つ以上の *DMA cookie* に完全に収まります。DMA cookie は、DMA エンジンがデータ転送で使用するメモリーの連続した部分を表します。システムは、次の情報に基づいてオブジェクトを複数の cookie に分けます。

- ドライバが提供する `ddi_dma_attr(9S)` 属性構造体
- ターゲットオブジェクトのメモリー位置
- ターゲットオブジェクトの配置

オブジェクトが DMA エンジンの制限内に収まらない大きさの場合は、そのオブジェクトを複数の *DMA* ウィンドウに分ける必要があります。リソースをアクティブにして割り当てることができるのは、1 度に 1 つのウィンドウだけです。`ddi_dma_getwin(9F)` 関数は、オブジェクト内のウィンドウ間の位置を決めるために使用します。各 DMA ウィンドウは、1 つ以上の DMA cookie で構成されます。詳細は、182 ページの「DMA ウィンドウ」を参照してください。

DMA エンジンの中には、複数の cookie を受け入れられるものがあります。そのようなエンジンは、システムの助けを借りずに scatter/gather 入出力を実行します。1 つのバインドから複数の cookie が返される場合、ドライバは `ddi_dma_nextcookie(9F)` を繰り返し呼び出して、各 cookie を取得します。これらの cookie は、エンジン内にプログラミングする必要があります。その後、これらの DMA cookie の集合体の総バイト数を転送するようにデバイスをプログラミングできます。

DMA 操作

DMA 転送の手順は、どの種類の DMA でも似ています。以降のセクションでは、DMA 転送を実行する方法について説明します。

注 – ファイルシステムから生成されるバッファー用のブロックドライバのメモリー内に DMA オブジェクトをロックしておく必要はありません。メモリー内のデータは、ファイルシステムによってすでにロックされています。

バスマスター DMA 転送の実行

ドライバは、バスマスター DMA に関する次の手順を実行します。

1. DMA 属性を記述します。この手順により、DMA ルーチンで、デバイスがバッファーにアクセスできることを保証できるようになります。
2. DMA ハンドルを割り当てます。
3. DMA オブジェクトがメモリー内でロックされていることを確認します。physio(9F) または ddi_umem_lock(9F) のマニュアルページを参照してください。
4. DMA リソースをオブジェクトに割り当てます。
5. デバイス上で DMA エンジンをプログラミングします。
6. エンジンを起動します。
7. 転送が完了したら、バスマスター操作を続行します。
8. 必要なオブジェクト同期があれば、それを行います。
9. DMA リソースを解放します。
10. DMA ハンドルを解放します。

ファーストパーティー DMA 転送の実行

ドライバは、ファーストパーティー DMA に関する次の手順を実行します。

1. DMA チャネルを割り当てます。
2. ddi_dmae_1stparty(9F) を使用してチャネルを構成します。
3. DMA オブジェクトがメモリー内でロックされていることを確認します。physio(9F) または ddi_umem_lock(9F) のマニュアルページを参照してください。
4. DMA リソースをオブジェクトに割り当てます。
5. デバイス上で DMA エンジンをプログラミングします。
6. エンジンを起動します。
7. 転送が完了したら、バスマスター操作を続行します。
8. 必要なオブジェクト同期があれば、それを行います。
9. DMA リソースを解放します。
10. DMA チャネルを解放します。

サードパーティー DMA 転送の実行

ドライバは、サードパーティー DMA に関する次の手順を実行します。

1. DMA チャネルを割り当てます。
2. ddi_dmae_getattr(9F) を使用してシステムの DMA エンジンの属性を取得します。

3. DMA オブジェクトをメモリー内にロックします。physio(9F) または ddi_umem_lock(9F) のマニュアルページを参照してください。
4. DMA リソースをオブジェクトに割り当てます。
5. ddi_dmae_prog(9F) を使用して、転送を行うようにシステムの DMA エンジンをプログラミングします。
6. 必要なオブジェクト同期があれば、それを行います。
7. ddi_dmae_stop(9F) を使用して DMA エンジンを停止します。
8. DMA リソースを解放します。
9. DMA チャネルを解放します。

ハードウェアプラットフォームの中には、バス固有の方法で DMA 機能を制限しているものもあります。ドライバは ddi_slaveonly(9F) を使用して、DMA を使用できるスロット内にデバイスがあるかどうかを確認します。

DMA 属性

DMA 属性には、次のような DMA エンジンの属性と制限が記述されています。

- デバイスがアクセスできるアドレスの制限
- 最大転送数
- アドレスの配置の制限

デバイスドライバは、ddi_dma_attr(9S) 構造体を通じて DMA エンジンのすべての制限をシステムに通知する必要があります。この処理により、システムが割り当てた DMA リソースにデバイスの DMA エンジンからアクセスできるようになります。システムでは、デバイス属性にさらなる制限を加えることはありますが、ドライバが設定した制限を削除することはありません。

ddi_dma_attr 構造体

DMA 属性構造体は、次のメンバーで構成されます。

```
typedef struct ddi_dma_attr {
    uint_t      dma_attr_version;       /* version number */
    uint64_t    dma_attr_addr_lo;       /* low DMA address range */
    uint64_t    dma_attr_addr_hi;       /* high DMA address range */
    uint64_t    dma_attr_count_max;     /* DMA counter register */
    uint64_t    dma_attr_align;         /* DMA address alignment */
    uint_t      dma_attr_burstsizes;    /* DMA burstsizes */
    uint32_t    dma_attr_minxfer;       /* min effective DMA size */
    uint64_t    dma_attr_maxxfer;       /* max DMA xfer size */
    uint64_t    dma_attr_seg;           /* segment boundary */
    int         dma_attr_sgllen;        /* s/g length */
    uint32_t    dma_attr_granular;      /* granularity of device */
    uint_t      dma_attr_flags;         /* Bus specific DMA flags */
} ddi_dma_attr_t;
```

各表記の意味は次のとおりです。

`dma_attr_version`	属性構造体のバージョン番号です。`dma_attr_version` は DMA_ATTR_V0 に設定されます。
`dma_attr_addr_lo`	DMA エンジンがアクセスできる最下位のバスアドレスです。
`dma_attr_addr_hi`	DMA エンジンがアクセスできる最上位のバスアドレスです。
`dma_attr_count_max`	DMA エンジンが 1 つの cookie で処理できる最大転送数を指定します。上限は、最大数から 1 を引いた数で表されます。この数はビットマスクとして使用されるため、2 のべき乗よりも小さくする必要もあります。
`dma_attr_align`	`ddi_dma_mem_alloc(9F)` からメモリーを割り当てるときの配置要件を指定します。配置要件の例として、ページ境界での配置があります。`dma_attr_align` フィールドはメモリーの割り当て時にしか使用されません。バインド操作時には、このフィールドは無視されます。バインド操作では、ドライバはバッファーが適切に配置されるようにする必要があります。
`dma_attr_burstsizes`	デバイスがサポートするバーストサイズを指定します。バーストサイズとは、デバイスがバスを放棄する前に転送できるデータ量のことです。バーストこのメンバーはバイナリ符号化方式のバーストサイズです。このサイズは、2 のべき乗になるものとみなされます。たとえば、デバイスが 1 バイト、2 バイト、4 バイト、および 16 バイトのバーストを処理できる場合、このフィールドは 0x17 に設定されます。システムでは、このフィールドを使用して配置制限も決めます。
`dma_attr_minxfer`	デバイスが実行できる効果的な最低転送サイズです。このサイズは、配置やパディングの制限にも影響を及ぼします。
`dma_attr_maxxfer`	DMA エンジンが 1 回の入出力コマンドで格納できる最大バイト数を記述します。この制限が意味を持つのは、`dma_attr_maxxfer` が `(dma_attr_count_max + 1) * dma_attr_sgllen` よりも小さい場合のみです。
`dma_attr_seg`	DMA エンジンのアドレスレジスタの上限です。`dma_attr_seg` は、アドレスレジスタの上位 8 ビットがセグメント番号を含むラッチである場合に頻繁に使用されます。下位 24 ビットはセグメントのアドレス指定に使用されます。この場合、`dma_attr_seg` は 0xFFFFFF に設定されま

	す。これにより、オブジェクトへのリソースの割り当て時にシステムは 24 ビットのセグメント境界を越えることはできません。
dma_attr_sgllen	scatter/gather リストに含まれる最大エントリ数を指定します。dma_attr_sgllen は、DMA エンジンがデバイスへの 1 回の入出力要求で使用できる cookie の数です。DMA エンジンに scatter/gather リストがない場合、このフィールドは 1 に設定されます。
dma_attr_granular	このフィールドは、デバイスの DMA 転送機能の粒度をバイト単位で指定します。この値の使用方法の例として、外部ストレージデバイスのセクターサイズの指定が挙げられます。バインド操作で部分的なマッピングが必要な場合、このフィールドを使用して、DMA ウィンドウ内の cookie のサイズ合計が粒度の整数倍になるようにします。ただし、デバイスに scatter/gather 機能がない場合は、DDI で粒度を保証することはできません。この場合、 dma_attr_granular フィールドの値は 1 になります。
dma_attr_flags	このフィールドには DDI_DMA_FORCE_PHYSICAL を設定できます。これは、システムが物理入出力アドレスと仮想入出力アドレスの両方をサポートしている場合に、仮想ではなく物理アドレスが返されることを示します。システムが物理 DMA をサポートしていない場合、ddi_dma_alloc_handle(9F) の戻り値は DDI_DMA_BADATTR になります。この場合、ドライバは DDI_DMA_FORCE_PHYSICAL をクリアし、処理を再試行する必要があります。

SBus の例

SPARC マシンの SBus 上の DMA エンジンには、次の属性があります。

- 0xFF000000 から 0xFFFFFFFF までのアドレスにのみアクセス
- 32 ビットの DMA カウンタレジスタ
- バイト整列された転送を処理できる
- 1 バイト、2 バイト、および 4 バイトのバーストサイズをサポート
- 効果的な最低転送サイズが 1 バイト
- 32 ビットのアドレスレジスタ
- scatter/gather リストなし
- セクターのみ (ディスクなど) での操作

SPARC マシンの SBus 上の DMA エンジンには、次の属性構造体があります。

```
static ddi_dma_attr_t attributes = {
    DMA_ATTR_V0,   /* Version number */
```

```
    0xFF000000,    /* low address */
    0xFFFFFFFF,    /* high address */
    0xFFFFFFFF,    /* counter register max */
    1,             /* byte alignment */
    0x7,           /* burst sizes: 0x1 | 0x2 | 0x4 */
    0x1,           /* minimum transfer size */
    0xFFFFFFFF,    /* max transfer size */
    0xFFFFFFFF,    /* address register max */
    1,             /* no scatter-gather */
    512,           /* device operates on sectors */
    0,             /* attr flag: set to 0 */
};
```

ISA バスの例

x86 マシンの ISA バス上の DMA エンジンには、次の属性があります。

- 最初の 16M バイトのメモリーにのみアクセス
- 1 回の DMA 転送で 1M バイト境界を越えることはできない
- 16 ビットのカウンタレジスタ
- バイト整列された転送を処理できる
- 1 バイト、2 バイト、および 4 バイトのバーストサイズをサポート
- 効果的な最低転送サイズが 1 バイト
- 17 以下の scatter/gather 転送を保持できる
- セクターのみ (ディスクなど) での操作

x86 マシンの ISA バス上の DMA エンジンには、次の属性構造体があります。

```
static ddi_dma_attr_t attributes = {
    DMA_ATTR_V0,   /* Version number */
    0x00000000,    /* low address */
    0x00FFFFFF,    /* high address */
    0xFFFF,        /* counter register max */
    1,             /* byte alignment */
    0x7,           /* burst sizes */
    0x1,           /* minimum transfer size */
    0xFFFFFFFF,    /* max transfer size */
    0x000FFFFF,    /* address register max */
    17,            /* scatter-gather */
    512,           /* device operates on sectors */
    0,             /* attr flag: set to 0 */
};
```

DMA リソースの管理

このセクションでは、DMA リソースを管理する方法について説明します。

オブジェクトのロック

DMA リソースをメモリーオブジェクトに割り当てる前に、そのオブジェクトを移動できないようにする必要があります。そうしないと、デバイスがオブジェクトへの書き込みを試している間に、システムがそのオブジェクトをメモリーから削除する可能性があります。オブジェクトが見つからないと、データ転送に失敗し、システムが破壊される恐れがあります。DMA 転送中にメモリーオブジェクトが移動できないようにするプロセスは、オブジェクトのロックダウンと呼ばれます。

次のオブジェクトタイプは、明示的なロックが必要ありません。

- `strategy(9E)` によってファイルシステムから生成されるバッファー。これらのバッファーはファイルシステムによってすでにロックされています。
- デバイスドライバ内で割り当てられたカーネルメモリー (`ddi_dma_mem_alloc(9F)` で割り当てられたものなど)。

ユーザー空間から生成されるバッファーなど、その他のオブジェクトについては、`physio(9F)` または `ddi_umem_lock(9F)` を使用してロックダウンする必要があります。これらの関数を使ったオブジェクトのロックダウンは通常、文字デバイスドライバの `read(9E)` または `write(9E)` ルーチン内で実行されます。314 ページの「データ転送方法」を参照してください。

DMA ハンドルの割り当て

DMA ハンドルは、そのあとに割り当てられる DMA リソースへの参照として使用される不透明なオブジェクトです。DMA ハンドルは通常、`ddi_dma_alloc_handle(9F)` を使用する、ドライバの `attach()` エントリポイントで割り当てられます。`ddi_dma_alloc_handle()` 関数は、*dip* によって参照されるデバイス情報と、`ddi_dma_attr(9S)` 構造体によってパラメータとして記述されたデバイスの DMA 属性を取得します。`ddi_dma_alloc_handle()` 関数の構文は次のとおりです。

```
int ddi_dma_alloc_handle(dev_info_t *dip,
    ddi_dma_attr_t *attr, int (*callback)(caddr_t),
    caddr_t arg, ddi_dma_handle_t *handlep);
```

各表記の意味は次のとおりです。

dip デバイスの `dev_info` 構造体へのポインタ。

attr `ddi_dma_attr(9S)` 構造体へのポインタです (166 ページの「DMA 属性」を参照)。

callback リソース割り当てエラーに対処するためのコールバック関数のアドレスです。

arg コールバック関数に渡される引数です。

handlep 返されたハンドルを格納するための DMA ハンドルへのポインタです。

DMA リソースの割り当て

DMA リソースは次の 2 つのインタフェースで割り当てられます。

- ddi_dma_buf_bind_handle(9F) – buf(9S) 構造体とともに使用されます
- ddi_dma_addr_bind_handle(9F) – 仮想アドレスとともに使用されます

DMA リソースは通常、ドライバの xxstart() ルーチンで割り当てられます (xxstart() ルーチンが存在する場合)。xxstart() については、345 ページの「非同期データ転送 (ブロックドライバ)」を参照してください。これらの 2 つのインタフェースの構文は次のとおりです。

```
int ddi_dma_addr_bind_handle(ddi_dma_handle_t handle,
    struct as *as, caddr_t addr,
    size_t len, uint_t flags, int (*callback)(caddr_t),
    caddr_t arg, ddi_dma_cookie_t *cookiep, uint_t *ccountp);

int ddi_dma_buf_bind_handle(ddi_dma_handle_t handle,
    struct buf *bp, uint_t flags,
    int (*callback)(caddr_t), caddr_t arg,
    ddi_dma_cookie_t *cookiep, uint_t *ccountp);
```

次の引数は、ddi_dma_addr_bind_handle(9F) と ddi_dma_buf_bind_handle(9F) の両方に共通です。

handle DMA ハンドルと、リソースを割り当てるためのオブジェクトです。

flags 転送方向などの属性を指定するフラグセットです。DDI_DMA_READ はデバイスからメモリーへのデータ転送を指定します。DDI_DMA_WRITE はメモリーからデバイスへのデータ転送を指定します。利用できるフラグの詳細は、ddi_dma_addr_bind_handle(9F) または ddi_dma_buf_bind_handle(9F) のマニュアルページを参照してください。

callback リソース割り当てエラーに対処するためのコールバック関数のアドレスです。ddi_dma_alloc_handle(9F) のマニュアルページを参照してください。

arg コールバック関数に渡される引数です。

cookiep このオブジェクトの最初の DMA cookie へのポインタです。

ccountp このオブジェクトの DMA cookie の数へのポインタです。

ddi_dma_addr_bind_handle(9F) の場合、オブジェクトは、次のパラメータでアドレス範囲を指定することによって記述します。

as アドレス空間構造体へのポインタです。*as* の値は NULL になります。

addr　オブジェクトのベースカーネルアドレスです。

len　オブジェクトの長さをバイト単位で指定します。

ddi_dma_buf_bind_handle(9F) の場合、オブジェクトは、bp が示す buf(9S) 構造体で記述します。

デバイスレジスタ構造体

DMA 対応デバイスには、前述の例で使用したものよりも多くのレジスタが必要になります。

scatter/gather 機能を使わずに DMA 対応デバイスをサポートするには、デバイスレジスタ構造体で次のフィールドを使用します。

```
uint32_t      dma_addr;      /* starting address for DMA */
uint32_t      dma_size;      /* amount of data to transfer */
```

scatter/gather 機能を使って DMA 対応デバイスをサポートするには、デバイスレジスタ構造体で次のフィールドを使用します。

```
struct sglentry {
    uint32_t    dma_addr;
    uint32_t    dma_size;
} sglist[SGLLEN];

caddr_t       iopb_addr;     /* When written, informs the device of the next */
                             /* command's parameter block address. */
                             /* When read after an interrupt, contains */
                             /* the address of the completed command. */
```

DMA コールバックの例

例 9–1 では、コールバック関数として xxstart() を使用します。また、xxstart() の引数としてデバイスごとの状態構造体を使用します。xxstart() 関数は、コマンドを開始しようとします。リソースが利用できないためにコマンドを開始できない場合、xxstart() はあとでリソースが利用可能になったときに呼び出されるようにスケジュールされます。

xxstart() は DMA コールバックとして使用されるため、xxstart() は DMA コールバックに課せられている次の規則に従う必要があります。

- リソースは利用可能であると想定できない。コールバックでリソースの割り当てを再度試みる必要がある。

- コールバックで、割り当てが成功したかどうかをシステムに知らせる必要がある。コールバックがリソースの割り当てに失敗すると、DDI_DMA_CALLBACK_RUNOUT が返されます。その場合は、あとで xxstart() を再度呼び出す必要があります。DDI_DMA_CALLBACK_DONE は成功を示しているため、これ以上コールバックは必要ありません。

例 9–1 DMA コールバックの例

```
static int
xxstart(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    struct device_reg *regp;
    int flags;
    mutex_enter(&xsp->mu);
    if (xsp->busy) {
        /* transfer in progress */
        mutex_exit(&xsp->mu);
        return (DDI_DMA_CALLBACK_RUNOUT);
    }
    xsp->busy = 1;
    regp = xsp->regp;
    if ( /* transfer is a read */ ) {
        flags = DDI_DMA_READ;
    } else {
        flags = DDI_DMA_WRITE;
    }
    mutex_exit(&xsp->mu);
    if (ddi_dma_buf_bind_handle(xsp->handle,xsp->bp,flags, xxstart,
        (caddr_t)xsp, &cookie, &ccount) != DDI_DMA_MAPPED) {
        /* really should check all return values in a switch */
        mutex_enter(&xsp->mu);
        xsp->busy=0;
        mutex_exit(&xsp->mu);
        return (DDI_DMA_CALLBACK_RUNOUT);
    }
    /* Program the DMA engine. */
    return (DDI_DMA_CALLBACK_DONE);
}
```

最大バーストサイズの決定

ドライバは、そのデバイスがサポートする DMA バーストサイズを、ddi_dma_attr(9S) 構造体の dma_attr_burstsizes フィールドで指定します。このフィールドは、サポートされるバーストサイズのビットマップです。ただし、システムでは、DMA リソースが割り当てられたときに、デバイスが実際に使用する可能性のあるバーストサイズに対してさらなる制限を加えることがあります。ddi_dma_burstsizes(9F) ルーチンを使用すると、許容されるバーストサイズを取得できます。このルーチンは、デバイスの適切なバーストサイズビットマップを返します。ドライバは、DMA リソースが割り当てられるときに、DMA エンジンに使用する適切なバーストサイズをシステムに問い合わせることができます。

例 9–2 バーストサイズの決定

```
#define BEST_BURST_SIZE 0x20 /* 32 bytes */

    if (ddi_dma_buf_bind_handle(xsp->handle,xsp->bp, flags, xxstart,
        (caddr_t)xsp, &cookie, &ccount) != DDI_DMA_MAPPED) {
        /* error handling */
    }
    burst = ddi_dma_burstsizes(xsp->handle);
    /* check which bit is set and choose one burstsize to */
    /* program the DMA engine */
    if (burst & BEST_BURST_SIZE) {
        /* program DMA engine to use this burst size */
    } else {
        /* other cases */
    }
```

プライベート DMA バッファーの割り当て

一部のデバイスドライバでは、ユーザースレッドやカーネルから要求された転送を実行するのに加え、DMA 転送用のメモリーを割り当てることが必要になる場合があります。プライベート DMA バッファーを割り当てる例として、デバイスとの通信用の共有メモリーを設定したり、中間転送バッファーを割り当てたりすることが挙げられます。DMA 転送用のメモリーを割り当てるには、ddi_dma_mem_alloc(9F) を使用します。

```
int ddi_dma_mem_alloc(ddi_dma_handle_t handle, size_t length,
    ddi_device_acc_attr_t *accattrp, uint_t flags,
    int (*waitfp)(caddr_t), caddr_t arg, caddr_t *kaddrp,
    size_t *real_length, ddi_acc_handle_t *handlep);
```

各表記の意味は次のとおりです。

handle DMA ハンドルです

length 目的とする割り当ての長さ (バイト単位) です

accattrp デバイスアクセス属性構造体へのポインタです

flags データ転送モードフラグです。使用できる値は、DDI_DMA_CONSISTENT と DDI_DMA_STREAMING です。

waitfp リソース割り当てエラーに対処するためのコールバック関数のアドレスです。ddi_dma_alloc_handle(9F) のマニュアルページを参照してください。

arg コールバック関数に渡される引数です

kaddrp 割り当てられたストレージのアドレスを含む正常な復帰へのポインタです

real_length 割り当てられた長さ (バイト単位) です

handlep　　データアクセスハンドルへのポインタです

デバイスが非順次方式でアクセスする場合、*flags* パラメータは DDI_DMA_CONSISTENT に設定されます。ddi_dma_sync(9F) を使用する同期手順は、小さなオブジェクトに頻繁に適用されるため、できるだけ軽量にします。このようなアクセスは一般に、一貫性アクセスと呼ばれています。一貫性アクセスは、デバイスとドライバ間の通信に使用される入出力パラメータブロックに特に役立ちます。

x86 プラットフォームでは、物理的に連続したメモリーである DMA メモリーを割り当てる際に次の要件があります。

- ddi_dma_attr(9S) 構造体に含まれる scatter/gather リスト dma_attr_sgllen の長さを 1 に設定する必要がある。
- DDI_DMA_PARTIAL を指定しない。DDI_DMA_PARTIAL を指定すると、部分的なリソース割り当てが可能になります。

次の例は、IOPB メモリーと、そのメモリーへのアクセスに必要な DMA リソースの割り当て方法を示しています。DMA リソースを引き続き割り当て、DDI_DMA_CONSISTENT フラグを割り当て関数に渡す必要があります。

例 9–3　ddi_dma_mem_alloc(9F) の使用

```
if (ddi_dma_mem_alloc(xsp->iopb_handle, size, &accattr,
    DDI_DMA_CONSISTENT, DDI_DMA_SLEEP, NULL, &xsp->iopb_array,
    &real_length, &xsp->acchandle) != DDI_SUCCESS) {
    /* error handling */
    goto failure;
}
if (ddi_dma_addr_bind_handle(xsp->iopb_handle, NULL,
    xsp->iopb_array, real_length,
    DDI_DMA_READ | DDI_DMA_CONSISTENT, DDI_DMA_SLEEP,
    NULL, &cookie, &count) != DDI_DMA_MAPPED) {
    /* error handling */
    ddi_dma_mem_free(&xsp->acchandle);
    goto failure;
}
```

メモリー転送が順次、単方向、ブロックサイズ、およびブロック整列の方式で行われる場合、*flags* パラメータは DDI_DMA_STREAMING に設定されます。このようなアクセスは一般に、ストリーミングアクセスと呼ばれています。

場合によっては、入出力キャッシュを使用すると入出力転送の速度を上げることができます。入出力キャッシュは、最低でも 1 つのキャッシュラインを転送します。ddi_dma_mem_alloc(9F) ルーチンでは、データの破壊を避けるために size を四捨五入してキャッシュラインの倍数にします。

ddi_dma_mem_alloc(9F) 関数は、割り当てられたメモリーオブジェクトの実際のサイズを返します。パディングと配置の要件のために、実際のサイズは要求されたサイズよりも大きくなることがあります。ddi_dma_addr_bind_handle(9F) 関数には実際の長さを指定する必要があります。

ddi_dma_mem_alloc(9F) で割り当てられたメモリーを解放するには、ddi_dma_mem_free(9F) 関数を使用します。

注 – ドライバは、バッファーが適切に配置されるようにする必要があります。下方バインドされた DMA バッファーの配置要件が設定されているデバイスのドライバは、それらの要件を満たすドライバの中間バッファーにデータをコピーしてから、その中間バッファーを DMA 用の DMA ハンドルにバインドすることが必要な場合があります。ドライバの中間バッファーを割り当てるには、ddi_dma_mem_alloc(9F) を使用します。アクセスするデバイスにメモリーを割り当てるには、kmem_alloc(9F) ではなく、必ず ddi_dma_mem_alloc(9F) を使用します。

リソース割り当てエラーの処理

リソース割り当てルーチンは、割り当てエラーの処理時にいくつかのオプションをドライバに提供します。*waitfp* 引数は、次の表に示すように、割り当てルーチンがブロックする、すぐに復帰する、コールバックをスケジュールするのいずれになるかを示します。

表 9–1　リソース割り当て処理

waitfp 値	指示された処理
DDI_DMA_DONTWAIT	ドライバはリソースが利用できるようになるのを待たない
DDI_DMA_SLEEP	ドライバはリソースが利用できるようになるまで待ち続ける
その他の値	リソースが利用できるようになった可能性があるときに呼び出される関数のアドレス

DMA エンジンのプログラミング

リソースの割り当てが正常に完了したら、デバイスをプログラミングする必要があります。DMA エンジンのプログラミングはデバイスごとに異なりますが、開始アドレスと転送カウントはすべての DMA エンジンに必要です。デバイスドライバはこれらの 2 つの値を、ddi_dma_addr_bind_handle(9F)、ddi_dma_buf_bind_handle(9F)、または ddi_dma_getwin(9F) の呼び出しが正常に行われた場合に返される *DMA cookie* から取得します。これらの関数はすべて、最初の DMA cookie と、DMA オブジェクト

が複数の cookie で構成されているかどうかを示す cookie カウントを返します。cookie カウント *N* が 1 よりも大きければ、ddi_dma_nextcookie(9F) を *N*-1 回だけ呼び出して残りの cookie をすべて取得します。

DMA cookie の型は ddi_dma_cookie(9S) です。この型の cookie には次のフィールドがあります。

```
uint64_t    _dmac_ll;       /* 64-bit DMA address */
uint32_t    _dmac_la[2];    /* 2 x 32-bit address */
size_t      dmac_size;      /* DMA cookie size */
uint_t      dmac_type;      /* bus specific type bits */
```

dmac_laddress フィールドには、デバイスの DMA エンジンのプログラミングに適した 64 ビットの入出力アドレスを指定します。デバイスに 64 ビットの DMA アドレスレジスタがある場合、ドライバはこのフィールドを使用して DMA エンジンをプログラミングします。dmac_address フィールドには、32 ビットの DMA アドレスレジスタを持つデバイスに使用される 32 ビットの入出力アドレスを指定します。dmac_size フィールドには、転送カウントが入ります。cookie の dmac_type フィールドがドライバで必要になるかどうかは、バスアーキテクチャーによって決まります。ドライバは、cookie に対して論理操作や算術操作などの操作は一切行いません。

例 **9–4** ddi_dma_cookie(9S) の例

```
ddi_dma_cookie_t            cookie;

     if (ddi_dma_buf_bind_handle(xsp->handle,xsp->bp, flags, xxstart,
     (caddr_t)xsp, &cookie, &xsp->ccount) != DDI_DMA_MAPPED) {
         /* error handling */
      }
     sglp = regp->sglist;
     for (cnt = 1; cnt <= SGLLEN; cnt++, sglp++) {
     /* store the cookie parms into the S/G list */
     ddi_put32(xsp->access_hdl, &sglp->dma_size,
         (uint32_t)cookie.dmac_size);
     ddi_put32(xsp->access_hdl, &sglp->dma_addr,
         cookie.dmac_address);
     /* Check for end of cookie list */
     if (cnt == xsp->ccount)
         break;
     /* Get next DMA cookie */
     (void) ddi_dma_nextcookie(xsp->handle, &cookie);
     }
    /* start DMA transfer */
     ddi_put8(xsp->access_hdl, &regp->csr,
     ENABLE_INTERRUPTS | START_TRANSFER);
```

DMA リソースの解放

DMA 転送の完了後、通常は割り込みルーチンで、ドライバは ddi_dma_unbind_handle(9F) を呼び出して、DMA リソースを解放できます。

180 ページの「メモリーオブジェクトの同期」で説明しているように、ddi_dma_unbind_handle(9F) が ddi_dma_sync(9F) を呼び出すことで、明示的に同期を行う必要がなくなります。ddi_dma_unbind_handle(9F) の呼び出し後、DMA リソースは無効になり、それ以上そのリソースを参照しても結果は保証されません。次の例は、ddi_dma_unbind_handle(9F) の使用方法を示しています。

例 9–5　DMA リソースの解放

```
static uint_t
xxintr(caddr_t arg)
{
     struct xxstate *xsp = (struct xxstate *)arg;
     uint8_t    status;
    volatile   uint8_t   temp;
     mutex_enter(&xsp->mu);
     /* read status */
     status = ddi_get8(xsp->access_hdl, &xsp->regp->csr);
     if (!(status & INTERRUPTING)) {
        mutex_exit(&xsp->mu);
        return (DDI_INTR_UNCLAIMED);
     }
     ddi_put8(xsp->access_hdl, &xsp->regp->csr, CLEAR_INTERRUPT);
      /* for store buffers */
     temp = ddi_get8(xsp->access_hdl, &xsp->regp->csr);
     ddi_dma_unbind_handle(xsp->handle);
     /* Check for errors. */
     xsp->busy = 0;
     mutex_exit(&xsp->mu);
     if ( /* pending transfers */ ) {
        (void) xxstart((caddr_t)xsp);
     }
     return (DDI_INTR_CLAIMED);
}
```

DMA リソースが解放されます。次の転送で別のオブジェクトが使用される場合、DMA リソースを再割り当てする必要があります。ただし、同じオブジェクトが常に使用される場合、リソースの割り当ては一度で構いません。途中の ddi_dma_sync(9F) の呼び出しが残っているかぎり、リソースの再利用が可能です。

DMA ハンドルの解放

ドライバが切り離される場合は、DMA ハンドルを解放する必要があります。ddi_dma_free_handle(9F) 関数は、DMA ハンドルと、システムがそのハンドル上でキャッシュしている残りのリソースをすべて破棄します。DMA ハンドルをそれ以上参照しても結果は保証されません。

DMA コールバックの取り消し

DMA コールバックを取り消すことはできません。DMA コールバックを取り消すには、ドライバの detach(9E) エントリポイントにコードを追加する必要があります。未処理のコールバックが存在する場合は、detach() ルーチンが DDI_SUCCESS を返すことはありません。例 9–6 を参照してください。DMA コールバックが発生すると、detach() ルーチンはコールバックが実行されるのを待つ必要があります。コールバックが完了したら、detach() はコールバックが再スケジュールを行わないようにする必要があります。次の例に示すように、状態構造体で追加のフィールドを指定することによって、コールバックが再スケジュールを行わないようにすることができます。

例 9–6 DMA コールバックの取り消し

```
static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
     /* ... */
     mutex_enter(&xsp->callback_mutex);
     xsp->cancel_callbacks = 1;
     while (xsp->callback_count > 0) {
        cv_wait(&xsp->callback_cv, &xsp->callback_mutex);
     }
     mutex_exit(&xsp->callback_mutex);
     /* ... */
 }

static int
xxstrategy(struct buf *bp)
{
     /* ... */
     mutex_enter(&xsp->callback_mutex);
       xsp->bp = bp;
     error = ddi_dma_buf_bind_handle(xsp->handle, xsp->bp, flags,
         xxdmacallback, (caddr_t)xsp, &cookie, &ccount);
     if (error == DDI_DMA_NORESOURCES)
       xsp->callback_count++;
     mutex_exit(&xsp->callback_mutex);
     /* ... */
}

static int
xxdmacallback(caddr_t callbackarg)
{
     struct xxstate *xsp = (struct xxstate *)callbackarg;
     /* ... */
     mutex_enter(&xsp->callback_mutex);
     if (xsp->cancel_callbacks) {
        /* do not reschedule, in process of detaching */
        xsp->callback_count--;
        if (xsp->callback_count == 0)
           cv_signal(&xsp->callback_cv);
        mutex_exit(&xsp->callback_mutex);
        return (DDI_DMA_CALLBACK_DONE);    /* don't reschedule it */
     }
```

例 9–6 DMA コールバックの取り消し (続き)

```
     /*
      * Presumably at this point the device is still active
      * and will not be detached until the DMA has completed.
      * A return of 0 means try again later
      */
     error = ddi_dma_buf_bind_handle(xsp->handle, xsp->bp, flags,
         DDI_DMA_DONTWAIT, NULL, &cookie, &ccount);
     if (error == DDI_DMA_MAPPED) {
        /* Program the DMA engine. */
        xsp->callback_count--;
        mutex_exit(&xsp->callback_mutex);
        return (DDI_DMA_CALLBACK_DONE);
     }
     if (error != DDI_DMA_NORESOURCES) {
        xsp->callback_count--;
        mutex_exit(&xsp->callback_mutex);
        return (DDI_DMA_CALLBACK_DONE);
     }
     mutex_exit(&xsp->callback_mutex);
     return (DDI_DMA_CALLBACK_RUNOUT);
}
```

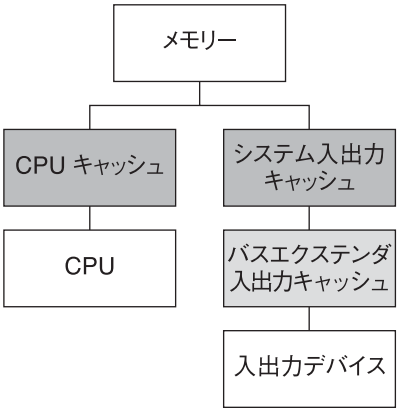
メモリーオブジェクトの同期

メモリーオブジェクトにアクセスする過程で、ドライバはさまざまなキャッシュに対してメモリーオブジェクトを同期させることが必要な場合があります。このセクションでは、メモリーオブジェクトの同期を行うタイミングとその方法に関するガイドラインを示します。

キャッシュ

CPU キャッシュは、CPU とシステムのメインメモリーの間に位置する非常に高速なメモリーです。入出力キャッシュは、デバイスとシステムのメインメモリーの間に位置します (次の図を参照)。

図 9–1 CPU キャッシュとシステムの入出力キャッシュ



メインメモリーからデータを読み取ろうとすると、関連付けられたキャッシュは要求されたデータがあるかどうかを確認します。データが利用可能である場合、キャッシュはすぐにデータを提供します。データがキャッシュに含まれていない場合、キャッシュはメインメモリーからそのデータを取り出します。次に、そのデータを要求元に渡し、次の要求に備えてデータを保存します。

同様に、書き込みサイクルでは、データはすぐにキャッシュに格納されます。CPU やデバイスは、実行 (つまりデータの転送) を続けることができます。データをキャッシュに格納するのにかかる時間は、データがメモリーに書き込まれるのを待つ時間よりもはるかに短くてすみます。

このモデルでは、デバイス転送が完了したあとも、引き続きデータを入出力キャッシュに格納でき、メインメモリーにはデータが入っていません。CPU がメモリーにアクセスした場合、CPU は CPU キャッシュから間違ったデータを読み取る可能性があります。ドライバは、同期ルーチンを呼び出して、入出力キャッシュからデータをフラッシュし、新しいデータで CPU キャッシュを更新する必要があります。この処理により、CPU のメモリーのビューの一貫性が確保されます。同様に、CPU によって変更されたデータにデバイスからアクセスする場合も、同期段階が必要です。

デバイスとメモリーの間に、バスエクステンダやブリッジなどの追加のキャッシュやバッファーを作成できます。`ddi_dma_sync(9F)` を使用して、該当するすべてのキャッシュを同期させます。

`ddi_dma_sync()` 関数

メモリーオブジェクトには、DMA ハンドルによって、CPU やデバイス用などの複数のマッピングが定義されている場合があります。複数のマッピングを持つドライバは、メモリーオブジェクトの変更にいずれかのマッピングが使用される場合、`ddi_dma_sync(9F)` を呼び出す必要があります。`ddi_dma_sync()` を呼び出すと、メ

モリーオブジェクトが別のマッピングを通じてアクセスされる前に、そのオブジェクトの変更が必ず完了しているようになります。ddi_dma_sync() 関数は、オブジェクトへのキャッシュされた参照が古くなっている場合に、オブジェクトのほかのマッピングに知らせることもできます。また、ddi_dma_sync() は必要に応じて古いキャッシュ参照をフラッシュしたり、無効にしたりします。

通常、ドライバは DMA 転送が完了したときに ddi_dma_sync() を呼び出す必要があります。ただし、ddi_dma_unbind_handle(9F) を使用した DMA リソースの解放により、ドライバに代わって ddi_dma_sync() が暗黙的に実行される場合は、この規則に当てはまりません。ddi_dma_sync() の構文は次のとおりです。

```
int ddi_dma_sync(ddi_dma_handle_t handle, off_t off,
size_t length, uint_t type);
```

オブジェクトがデバイスの DMA エンジンによって読み取られようとしている場合は、*type* を DDI_DMA_SYNC_FORDEV に設定して、デバイスのオブジェクトビューを同期させる必要があります。デバイスの DMA エンジンがメモリーオブジェクトに書き込みを行なったあとで、そのオブジェクトが CPU によって読み取られようとしている場合は、*type* を DDI_DMA_SYNC_FORCPU に設定して、CPU のオブジェクトビューを同期させる必要があります。

次の例は、CPU のための DMA オブジェクトの同期を示しています。

```
if (ddi_dma_sync(xsp->handle, 0, length, DDI_DMA_SYNC_FORCPU)
    == DDI_SUCCESS) {
    /* the CPU can now access the transferred data */
    /* ... */
} else {
    /* error handling */
}
```

メモリーが ddi_dma_mem_alloc(9F) によって割り当てられる場合のように、カーネルのためのマッピングしかない場合は、DDI_DMA_SYNC_FORKERNEL フラグを使用します。システムは、CPU のビューよりもかなり迅速にカーネルのビューを同期させようとします。システムがカーネルビューを高速で同期できない場合、システムは DDI_DMA_SYNC_FORCPU フラグが設定されているかのように動作します。

DMA ウィンドウ

オブジェクトが DMA エンジンの制限内に収まらない大きさの場合は、その転送をより小さい一連の転送に分ける必要があります。ドライバは転送そのものを分割できます。別の方法として、ドライバはシステムがオブジェクトの一部にだけリソースを割り当てられるようにすることができます。その結果、一連の DMA ウィンドウが作成されます。システムがリソースを割り当てられるようにすることが推奨される解決方法です。ドライバがリソースを管理するよりも効果的にリソースを管理できるからです。

DMA ウィンドウには 2 つの属性があります。*offset* 属性はオブジェクトの先頭から測定されます。*length* 属性は、割り当てられるメモリーのバイト数です。部分的な割り当てが行われたあと、*offset* で始まる *length* バイトの範囲だけにリソースが割り当てられています。

DMA ウィンドウを要求するには、ddi_dma_buf_bind_handle(9F) または ddi_dma_addr_bind_handle(9F) のパラメータとして DDI_DMA_PARTIAL フラグを指定します。ウィンドウを作成できる場合は、どちらの関数も DDI_DMA_PARTIAL_MAP を返します。ただし、システムがオブジェクト全体にリソースを割り当てることがあります。この場合は DDI_DMA_MAPPED が返されます。ドライバは、戻り値を調べて、DMA ウィンドウが使用されているかどうかを判断します。次の例を参照してください。

例 9–7　DMA ウィンドウの設定

```
static int
xxstart (caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    struct device_reg *regp = xsp->reg;
    ddi_dma_cookie_t cookie;
    int status;
    mutex_enter(&xsp->mu);
    if (xsp->busy) {
        /* transfer in progress */
        mutex_exit(&xsp->mu);
        return (DDI_DMA_CALLBACK_RUNOUT);
    }
    xsp->busy = 1;
    mutex_exit(&xsp->mu);
    if ( /* transfer is a read */) {
        flags = DDI_DMA_READ;
    } else {
        flags = DDI_DMA_WRITE;
    }
    flags |= DDI_DMA_PARTIAL;
    status = ddi_dma_buf_bind_handle(xsp->handle, xsp->bp,
        flags, xxstart, (caddr_t)xsp, &cookie, &ccount);
    if (status != DDI_DMA_MAPPED &&
        status != DDI_DMA_PARTIAL_MAP)
        return (DDI_DMA_CALLBACK_RUNOUT);
    if (status == DDI_DMA_PARTIAL_MAP) {
        ddi_dma_numwin(xsp->handle, &xsp->nwin);
        xsp->partial = 1;
        xsp->windex = 0;
    } else {
        xsp->partial = 0;
    }
    /* Program the DMA engine. */
    return (DDI_DMA_CALLBACK_DONE);
}
```

2 つの関数が DMA ウィンドウで動作します。最初の関数 ddi_dma_numwin(9F) は、特定の DMA オブジェクト用の DMA ウィンドウの数を返します。もう 1 つの関数 ddi_dma_getwin(9F) は、オブジェクト内での再配置、つまりシステムリソースの再割

り当てを許可します。ddi_dma_getwin() 関数は、現在のウィンドウをオブジェクト内の新しいウィンドウにシフトします。ddi_dma_getwin() はシステムリソースを新しいウィンドウに再割り当てするので、以前のウィンドウは無効になります。

注意 – 現在のウィンドウへの転送が完了する前に、ddi_dma_getwin() の呼び出しを使って DMA ウィンドウを移動しないでください。現在のウィンドウへの転送が完了するまで待ってください。このタイミングで割り込みが発生します。次に、データの破壊を避けるために ddi_dma_getwin() を呼び出してください。

Example 9–8() に示すように、例 9–8 関数は通常、割り込みルーチンから呼び出されます。最初の DMA 転送は、ドライバの呼び出しによって開始されます。それ以降の転送は、割り込みルーチンから開始されます。

割り込みルーチンは、デバイスのステータスを調べて、デバイスが転送を正常に完了しているかどうかを判断します。完了していない場合は、エラー回復処理が行われます。転送が正常に行われた場合、割り込みルーチンは論理転送が完了しているかどうかを調べる必要があります。完全な転送には、buf(9S) 構造体で指定されたオブジェクト全体が含まれています。部分的な転送では、1 つの DMA ウィンドウだけが移動されます。部分的な転送では、割り込みルーチンは ddi_dma_getwin(9F) を使ってウィンドウを移動し、新しい cookie を取得して、別の DMA 転送を開始します。

論理要求が完了している場合、割り込みルーチンは保留中の要求がないかどうか確認します。必要があれば、割り込みルーチンは転送を開始します。必要がなければ、割り込みルーチンは別の DMA 転送を呼び出さずに復帰します。次の例は、通常のフロー制御を示しています。

例 9–8 DMA ウィンドウを使用した割り込みハンドラ

```
static uint_t
xxintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    uint8_t    status;
    volatile   uint8_t   temp;
    mutex_enter(&xsp->mu);
    /* read status */
    status = ddi_get8(xsp->access_hdl, &xsp->regp->csr);
    if (!(status & INTERRUPTING)) {
        mutex_exit(&xsp->mu);
        return (DDI_INTR_UNCLAIMED);
    }
    ddi_put8(xsp->access_hdl,&xsp->regp->csr, CLEAR_INTERRUPT);
    /* for store buffers */
    temp = ddi_get8(xsp->access_hdl, &xsp->regp->csr);
    if ( /* an error occurred during transfer */ ) {
        bioerror(xsp->bp, EIO);
        xsp->partial = 0;
    } else {
```

例 9–8 DMA ウィンドウを使用した割り込みハンドラ (続き)

```
        xsp->bp->b_resid -= /* amount transferred */ ;
    }

    if (xsp->partial && (++xsp->windex < xsp->nwin)) {
        /* device still marked busy to protect state */
        mutex_exit(&xsp->mu);
        (void) ddi_dma_getwin(xsp->handle, xsp->windex,
            &offset, &len, &cookie, &ccount);
        /* Program the DMA engine with the new cookie(s). */
        return (DDI_INTR_CLAIMED);
    }
    ddi_dma_unbind_handle(xsp->handle);
    biodone(xsp->bp);
    xsp->busy = 0;
    xsp->partial = 0;
    mutex_exit(&xsp->mu);
    if ( /* pending transfers */ ) {
        (void) xxstart((caddr_t)xsp);
    }
    return (DDI_INTR_CLAIMED);
}
```

◆ ◆ ◆ 第 10 章

10 デバイスメモリーおよびカーネルメモリーのマッピング

一部のデバイスドライバでは、アプリケーションから mmap(2) 経由でデバイスメモリーまたはカーネルメモリーにアクセスできるようになっています。たとえば、フレームバッファードライバでは、フレームバッファーをユーザースレッド内にマップできます。別の例として、共有カーネルメモリープールを使用してアプリケーションとの通信を行う擬似ドライバなどが挙げられます。この章では、次の内容について説明します。

メモリーマッピングの概要

デバイスメモリーまたはカーネルメモリーをエクスポートする場合にドライバ内で行う必要のある手順を次に示します。

1. cb_ops(9S) 構造体の cb_flag フラグで D_DEVMAP フラグを設定します。
2. devmap(9E) ドライバエントリポイントと省略可能な segmap(9E) エントリポイントを定義してマッピングをエクスポートします。
3. devmap_devmem_setup(9F) を使用してデバイスへのユーザーマッピングを設定します。カーネルメモリーへのユーザーマッピングを設定するには、devmap_umem_setup(9F) を使用します。

マッピングのエクスポート

このセクションでは、segmap(9E) および devmap(9E) エントリポイントの使用方法について説明します。

segmap(9E) エントリポイント

segmap(9E) エントリポイントは、mmap(2) システムコールから要求されたメモリーマッピングを設定する役割を担います。多くのメモリーマッピングデバイスに対応するドライバは、独自の segmap(9E) ルーチンを定義する代わりに、ddi_devmap_segmap(9F) をエントリポイントとして使用します。ドライバは、segmap() エントリポイントを提供することにより、マッピング作成の前後で一般的なタスクを処理できます。たとえば、ドライバ内でマッピング許可を確認し、非公開のマッピングリソースを割り当てることができます。また、ドライバ内でマッピングを調整し、ページ境界割り当てされていないデバイスバッファーに対応させることもできます。segmap() エントリポイントは、戻り値を返す前に ddi_devmap_segmap(9F) 関数を呼び出す必要があります。ddi_devmap_segmap() 関数は、ドライバの devmap(9E) エントリポイントを呼び出すことで実際のマッピングを実行します。

segmap() 関数の構文は次のとおりです。

```
int segmap(dev_t dev, off_t off, struct as *asp, caddr_t *addrp,
     off_t len, unsigned int prot, unsigned int maxprot,
     unsigned int flags, cred_t *credp);
```

各表記の意味は次のとおりです。

dev マップするメモリーを含むデバイス。

off マッピングの開始位置となる、デバイスメモリー内のオフセット。

asp デバイスメモリーのマッピング先となるアドレス空間へのポインタ。

この引数には、例 10–1 に示した struct as *、例 10–2 に示した ddi_as_handle_t のどちらを指定してもかまいません。これは、ddidevmap.h に次の宣言が含まれているからです。

```
typedef struct as *ddi_as_handle_t
```

addrp デバイスメモリーのマッピング先となる、アドレス空間内のアドレスへのポインタ。

len マップするメモリーの長さ (バイト)。

prot 保護を指定するビットフィールド。指定可能な設定は、PROT_READ、PROT_WRITE、PROT_EXEC、PROT_USER、および PROT_ALL です。詳細はマニュアルページを参照してください。

maxprot 試みられたマッピングで使用可能な最大の保護フラグ。ユーザーが特殊ファイルを読み取り専用で開いた場合には、PROT_WRITE ビットをマスキングして除外できます。

flags マッピングのタイプを示すフラグ。指定可能な値は MAP_SHARED と MAP_PRIVATE です。

credp ユーザー資格構造体へのポインタ。

次の例のドライバは、書き込み専用マッピングを許可するフレームバッファーを制御しています。このドライバは、アプリケーションが読み取りアクセス権の取得を試みた場合に EINVAL を返し、続いて ddi_devmap_segmap(9F) を呼び出してユーザーマッピングを設定しています。

例 10–1 segmap(9E) ルーチン

```
static int
xxsegmap(dev_t dev, off_t off, struct as *asp, caddr_t *addrp,
    off_t len, unsigned int prot, unsigned int maxprot,
    unsigned int flags, cred_t *credp)
{
    if (prot & PROT_READ)
        return (EINVAL);
    return (ddi_devmap_segmap(dev, off, as, addrp,
        len, prot, maxprot, flags, cred));
}
```

次の例は、レジスタ空間内でページ境界割り当てされていないバッファーを持つデバイスの処理方法を示したものです。この例では、バッファーの先頭に対応するアドレスが mmap(2) から返されるように、オフセット 0x800 から始まるバッファーをマップしています。devmap_devmem_setup(9F) 関数は、ページの全体をマップし、マッピングがページ境界割り当てされるように要求したあと、ページの先頭のアドレスを返します。このアドレスが segmap(9E) をパススルーしたり、segmap() エントリポイントが未定義の場合には、バッファーの先頭に対応するアドレスではなくページの先頭に対応するアドレスが、mmap() から返されます。この例では、結果として返されるアドレスが目的とするバッファーの先頭位置になるように、devmap_devmem_setup からの戻り値であるページ境界割り当てされたアドレスに、バッファーのオフセットが加算されています。

例 10–2 mmap() 呼び出しから返されるアドレスを segmap() 関数を使用して変更する

```
#define    BUFFER_OFFSET 0x800

int
xx_segmap(dev_t dev, off_t off, ddi_as_handle_t as, caddr_t *addrp, off_t len,
    uint_t prot, uint_t maxprot, uint_t flags, cred_t *credp)
{
        int rval;
        unsigned long pagemask = ptob(1L) - 1L;
```

例 10–2 mmap() 呼び出しから返されるアドレスを segmap() 関数を使用して変更する (続き)

```
        if ((rval = ddi_devmap_segmap(dev, off, as, addrp, len, prot, maxprot,
            flags, credp)) == DDI_SUCCESS) {
                /*
                  * The address returned by ddi_devmap_segmap is the start of the page
                  * that contains the buffer.  Add the offset of the buffer to get the
                  * final address.
                  */
                *addrp += BUFFER_OFFSET & pagemask);
        }
        return (rval);
}
```

devmap(9E) エントリポイント

devmap(9E) エントリポイントは、segmap(9E) エントリポイントの内側にある ddi_devmap_segmap(9F) 関数から呼び出されます。

devmap(9E) エントリポイントは、mmap(2) システムコールの結果として呼び出されます。devmap(9E) 関数は、デバイスメモリーまたはカーネルメモリーをユーザーアプリケーションにエクスポートするために呼び出されます。devmap() 関数は次の操作のために使用されます。

- デバイスメモリーまたはカーネルメモリーへのユーザーマッピングが妥当かどうか検査する
- アプリケーションマッピング内の論理オフセットをデバイスメモリーまたはカーネルメモリー内の対応するオフセットに変換する
- マッピング情報をシステムに渡し、マッピングを設定できるようにする

devmap() 関数の構文は次のとおりです。

```
int devmap(dev_t dev, devmap_cookie_t handle, offset_t off,
     size_t len, size_t *maplen, uint_t model);
```

各表記の意味は次のとおりです。

dev マップするメモリーを含むデバイス。

handle システムがデバイス内またはカーネル内の連続するメモリーへのマッピングを記述するために作成して使用するデバイスマッピングハンドル。

off アプリケーションマッピング内の論理オフセット。ドライバはこのオフセットを、デバイスメモリーまたはカーネルメモリー内の対応するオフセットに変換する必要があります。

len マップするメモリーの長さ (バイト)。

maplen ドライバが、異なるカーネルメモリー領域または物理的に不連続な複数のメモリー領域を、連続する 1 つのユーザーアプリケーションマッピングに関連付けることを可能にします。

model 現在のスレッドのデータモデルタイプ。

システムは、1 つの mmap(2) システムコールで複数のマッピングハンドルを作成します。たとえば、マッピングには、物理的に不連続なメモリー領域が複数含まれる可能性があります。

最初、パラメータ *off* と *len* を指定して devmap(9E) が呼び出されます。これらのパラメータは、アプリケーションから mmap(2) に渡されます。devmap(9E) は、**maplen* を、*off* から連続するメモリー領域の末尾までの長さに設定します。 **maplen* の値は、ページサイズの倍数に切り上げる必要があります。 **maplen* の値は、元のマッピング長 `len` よりも小さい値に設定してもかまいません。そうした場合、システムは、新しいマッピングハンドルと調整後の *off* および *len* パラメータを使用しながら、最初のマッピング長に達するまで devmap(9E) を繰り返し呼び出します。

ドライバが複数のアプリケーションデータモデルをサポートする場合は、ddi_model_convert_from(9F) に *model* を渡す必要があります。`ddi_model_convert_from()` 関数は、現在のスレッドとデバイスドライバとの間にデータモデルの不一致が存在しているかどうかを判定します。ユーザースレッドが異なるデータモデルをサポートしている場合、デバイスドライバは、データ構造体の構造を調整したあとでその構造体をユーザースレッドにエクスポートしなければならない可能性があります。詳細については、付録 C 「64 ビットデバイスドライバの準備」のページを参照してください。

devmap(9E) エントリポイントは、論理オフセット `off` がドライバからエクスポートされるメモリーの範囲外である場合には *-1* を返す必要があります。

ユーザーマッピングへのデバイスメモリーの関連付け

デバイスメモリーをユーザーアプリケーションにエクスポートするには、ドライバの devmap_devmem_setup(9F) エントリポイントから devmap(9E) を呼び出します。

`devmap_devmem_setup(9F)` 関数の構文は次のとおりです。

```
int devmap_devmem_setup(devmap_cookie_t handle, dev_info_t *dip,
    struct devmap_callback_ctl *callbackops, uint_t rnumber,
    offset_t roff, size_t len, uint_t maxprot, uint_t flags,
    ddi_device_acc_attr_t *accattrp);
```

各表記の意味は次のとおりです。

handle システムがマッピングの識別子として使用する不透明なデバイスマッピングハンドル。

dip デバイスの dev_info 構造体へのポインタ。

callbackops マッピングに関するユーザーイベントの通知をドライバが受け取れるようにするための devmap_callback_ctl(9S) 構造体へのポインタ。

rnumber レジスタアドレス空間セットへのインデックス番号。

roff デバイスメモリー内へのオフセット。

len エクスポートされる長さ (バイト)。

maxprot ドライバが、エクスポートされるデバイスメモリー内の個々の領域ごとに保護を指定できるようにします。

flags DEVMAP_DEFAULTS に設定する必要があります。

accattrp ddi_device_acc_attr(9S) 構造体へのポインタ。

roff および *len* 引数は、レジスタセット *rnumber* で指定されたデバイスメモリー内の範囲を記述します。*rnumber* が参照するレジスタ指定は、reg プロパティーによって記述されます。レジスタセットが 1 つしかないデバイスでは、rnumber にゼロを渡します。範囲は *roff* と *len* で定義されます。この範囲は、devmap(9E) エントリポイントから渡された *offset* の位置で、ユーザーのアプリケーションマッピングからアクセス可能となります。通常、ドライバは devmap(9E) のオフセットを devmap_devmem_setup(9F) に直接渡します。このとき、mmap(2) から返されるアドレスは、レジスタセットの先頭アドレスにマップします。

ドライバは *maxprot* 引数を使用することで、エクスポートされるデバイスメモリー内の個々の領域ごとに保護を指定できます。たとえば、ある領域の書き込みアクセス権を拒否するには、その領域で PROT_READ と PROT_USER のみを設定します。

次の例は、デバイスメモリーをアプリケーションにエクスポートする方法を示したものです。ドライバはまず、要求されたマッピングがデバイスメモリーの領域内に収まっているかどうかを判定します。デバイスメモリーのサイズは、ddi_dev_regsize(9F) を使用して判定します。マッピングの長さは、ptob(9F) および btopr(9F) を使用してページサイズの倍数に切り上げられます。その後、devmap_devmem_setup(9F) が呼び出され、デバイスメモリーがアプリケーションにエクスポートされます。

例 10–3 devmap_devmem_setup() ルーチンの使用

```
static int
xxdevmap(dev_t dev, devmap_cookie_t handle, offset_t off, size_t len,
    size_t *maplen, uint_t model)
{
    struct xxstate *xsp;
    int    error, rnumber;
    off_t regsize;

    /* Set up data access attribute structure */
    struct ddi_device_acc_attr xx_acc_attr = {
```

例 10–3 devmap_devmem_setup() ルーチンの使用 (続き)

```
        DDI_DEVICE_ATTR_V0,
        DDI_NEVERSWAP_ACC,
        DDI_STRICTORDER_ACC
    };
    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL)
        return (-1);
    /* use register set 0 */
    rnumber = 0;
    /* get size of register set */
    if (ddi_dev_regsize(xsp->dip, rnumber, &regsize) != DDI_SUCCESS)
        return (-1);
    /* round up len to a multiple of a page size */
       len = ptob(btopr(len));
    if (off + len > regsize)
        return (-1);
    /* Set up the device mapping */
    error = devmap_devmem_setup(handle, xsp->dip, NULL, rnumber,
    off, len, PROT_ALL, DEVMAP_DEFAULTS, &xx_acc_attr);
    /* acknowledge the entire range */
    *maplen = len;
    return (error);
}
```

ユーザーマッピングへのカーネルメモリーの関連付け

一部のデバイスドライバでは、カーネルメモリーを割り当て、そのメモリーにユーザープログラムから mmap(2) 経由でアクセスできるようにしなければならない場合があります。たとえば、2 つのアプリケーション間で通信を行うための共有メモリーを設定する場合が挙げられます。また、ドライバとアプリケーションとの間でメモリーを共有する場合なども挙げられます。

カーネルメモリーをユーザーアプリケーションにエクスポートする場合は、次の手順に従います。

1. ddi_umem_alloc(9F) を使用してカーネルメモリーを割り当てます。
2. devmap_umem_setup(9F) を使用してメモリーをエクスポートします。
3. メモリーが不要になったら、ddi_umem_free(9F) を使用してメモリーを解放します。

ユーザーアクセス用カーネルメモリーの割り当て

アプリケーションにエクスポートするカーネルメモリーを割り当てるには、ddi_umem_alloc(9F) を使用します。ddi_umem_alloc() で使用される構文は、次のとおりです。

```
void *ddi_umem_alloc(size_t size, int flag, ddi_umem_cookie_t
*cookiep);
```

各表記の意味は次のとおりです。

size 割り当てるバイト数。

flag スリープ条件やメモリータイプを決定するために使用されます。

cookiep カーネルメモリー cookie へのポインタ。

ddi_umem_alloc(9F) は、ページ境界割り当てされたカーネルメモリーを割り当てます。ddi_umem_alloc() は、割り当てられたメモリーへのポインタを返します。最初、メモリーにはゼロが設定されます。割り当てられるバイト数はシステムページサイズの倍数であり、*size* パラメータから切り上げられます。割り当てられたメモリーは、カーネル内で使用してもかまいません。このメモリーは、アプリケーションにエクスポートすることもできます。*cookiep* は、割り当てられるカーネルメモリーを記述するカーネルメモリー cookie へのポインタです。*cookiep* は、ドライバがカーネルメモリーをユーザーアプリケーションにエクスポートするときに、devmap_umem_setup(9F) で使用されます。

flag 引数は、ddi_umem_alloc(9F) がブロックするかそれともすぐにリターンするのかと、割り当てられるカーネルメモリーがページング可能かどうかを示します。flag 引数の値は次のとおりです。

DDI_UMEM_NOSLEEP ドライバは、メモリーが使用可能になるまで待機する必要はありません。メモリーが使用可能でない場合は NULL を返します。

DDI_UMEM_SLEEP ドライバは、メモリーが使用可能になるまで無期限に待機できます。

DDI_UMEM_PAGEABLE ドライバは、メモリーのページアウトを許可します。これを設定しないと、メモリーがロックダウンされます。

ddi_umem_lock() 関数は、デバイスロックメモリーのチェックを実行できます。この関数は、project.max-locked-memory に指定された制限値に基づいてチェックします。現在のプロジェクトのロックメモリー使用量が制限値を下回っている場合、そのプロジェクトのロックメモリーバイト数が増やされます。制限チェックのあとでメモリーがロックされます。ddi_umem_unlock() 関数を呼び出すとメモリーのロックが解除され、プロジェクトのロックメモリーバイト数が減らされます。

使用されるアカウンティング方式は、精度の低いフルプライスモデルです。たとえば、オーバーラップしたメモリー領域を含む同一プロジェクト内で umem_lockmemory() を呼び出した 2 つの呼び出し元は、2 回課金されます。

ゾーンがインストールされた Oracle Solaris システムの project.max-locked-memory および zone.max-locked_memory リソース制御については、『Solaris Containers: Resource

Management and Solaris Zones Developer's Guide』を参照し、さらに resource_controls(5) も参照してください。

次の例は、アプリケーションアクセス用のカーネルメモリーを割り当てる方法を示したものです。ドライバは、複数のアプリケーションによって共有メモリー領域として使用される 1 枚のカーネルメモリーページをエクスポートします。メモリーの割り当ては、アプリケーションが共有ページをはじめてマップしたときに segmap(9E) 内で行われます。ドライバが複数のアプリケーションデータモデルをサポートする必要がある場合には、追加のページが割り当てられます。たとえば、64 ビットドライバが、64 ビットアプリケーションと 32 ビットアプリケーションの両方にメモリーをエクスポートするとします。このとき、64 ビットアプリケーションが 1 枚目のページを共有し、32 ビットアプリケーションが 2 枚目のページを共有します。

例 **10–4** ddi_umem_alloc() ルーチンの使用

```
static int
xxsegmap(dev_t dev, off_t off, struct as *asp, caddr_t *addrp, off_t len,
    unsigned int prot, unsigned int maxprot, unsigned int flags,
    cred_t *credp)
{
    int error;
    minor_t instance = getminor(dev);
    struct xxstate *xsp = ddi_get_soft_state(statep, instance);

    size_t mem_size;
        /* 64-bit driver supports 64-bit and 32-bit applications */
    switch (ddi_mmap_get_model()) {
        case DDI_MODEL_LP64:
             mem_size = ptob(2);
             break;
        case DDI_MODEL_ILP32:
             mem_size = ptob(1);
             break;
    }

    mutex_enter(&xsp->mu);
    if (xsp->umem == NULL) {

        /* allocate the shared area as kernel pageable memory */
        xsp->umem = ddi_umem_alloc(mem_size,
            DDI_UMEM_SLEEP | DDI_UMEM_PAGEABLE, &xsp->ucookie);
    }
    mutex_exit(&xsp->mu);
    /* Set up the user mapping */
    error = devmap_setup(dev, (offset_t)off, asp, addrp, len,
        prot, maxprot, flags, credp);

    return (error);
}
```

アプリケーションへのカーネルメモリーのエクスポート

カーネルメモリーをユーザーアプリケーションにエクスポートするには、devmap_umem_setup(9F) を使用します。devmap_umem_setup () は、ドライバの devmap(9E) エントリポイントから呼び出す必要があります。 devmap_umem_setup() の構文は次のとおりです。

```
int devmap_umem_setup(devmap_cookie_t handle, dev_info_t *dip,
    struct devmap_callback_ctl *callbackops, ddi_umem_cookie_t cookie,
    offset_t koff, size_t len, uint_t maxprot, uint_t flags,
    ddi_device_acc_attr_t *accattrp);
```

各表記の意味は次のとおりです。

handle マッピングを記述するために使用される不透明な構造体。

dip デバイスの dev_info 構造体へのポインタ。

callbackops devmap_callback_ctl(9S) 構造体へのポインタ。

cookie ddi_umem_alloc(9F) から返されたカーネルメモリー cookie。

koff cookie で指定されたカーネルメモリー内へのオフセット。

len エクスポートされる長さ (バイト)。

maxprot エクスポートされるマッピングで使用可能な最大の保護を指定します。

flags DEVMAP_DEFAULTS に設定する必要があります。

accattrp ddi_device_acc_attr(9S) 構造体へのポインタ。

handle は、システムがマッピングの識別子として使用するデバイスマッピングハンドルです。*handle* は、devmap(9E) エントリポイントから渡されます。 *dip* は、デバイスの dev_info 構造体へのポインタです。*callbackops* は、マッピングに関するユーザーイベントの通知をドライバが受け取れるようにします。カーネルメモリーのエクスポート時には、ほとんどのドライバで *callbackops* が NULL に設定されます。

koff および *len* は、ddi_umem_alloc(9F) によって割り当てられたカーネルメモリー内の範囲を指定します。この範囲は、devmap(9E) エントリポイントから渡されたオフセットの位置で、ユーザーのアプリケーションマッピングからアクセス可能となります。 通常、ドライバは devmap(9E) のオフセットを devmap_umem_setup(9F) に直接渡します。このとき、mmap(2) から返されるアドレスは、ddi_umem_alloc(9F) から返されたカーネルアドレスにマップします。*koff* と *len* は、ページ境界割り当てされる必要があります。

ドライバは *maxprot* を使用することで、エクスポートされるカーネルメモリー内の個々の領域ごとに保護を指定できます。たとえば、ある領域の書き込みアクセス権を許可しないようにするには、PROT_READ と PROT_USER のみを設定します。

次の例は、カーネルメモリーをアプリケーションにエクスポートする方法を示したものです。ドライバはまず、要求されたマッピングが、割り当てられたカーネルメモリー領域内に収まっているかチェックします。64 ビットドライバは、32 ビットアプリケーションからマッピング要求を受け取ると、その要求をカーネルメモリー領域の 2 枚目のページにリダイレクトします。このリダイレクトにより、同じデータモデルを使用してコンパイルされたアプリケーションのみが同一ページを共有するようになります。

例 10–5 devmap_umem_setup(9F) ルーチン

```
static int
xxdevmap(dev_t dev, devmap_cookie_t handle, offset_t off, size_t len,
    size_t *maplen, uint_t model)
{
    struct xxstate *xsp;
    int    error;

    /* round up len to a multiple of a page size */
    len = ptob(btopr(len));
    /* check if the requested range is ok */
    if (off + len > ptob(1))
        return (ENXIO);
    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL)
        return (ENXIO);

    if (ddi_model_convert_from(model) == DDI_MODEL_ILP32)
        /* request from 32-bit application. Skip first page */
        off += ptob(1);

    /* export the memory to the application */
    error = devmap_umem_setup(handle, xsp->dip, NULL, xsp->ucookie,
        off, len, PROT_ALL, DEVMAP_DEFAULTS, NULL);
    *maplen = len;
    return (error);
}
```

ユーザーアクセス用にエクスポートされたカーネルメモリーの解放

ドライバがアンロードされたら、ddi_umem_alloc(9F) によって割り当てられたメモリーを、ddi_umem_free(9F) を呼び出して解放する必要があります。

```
void ddi_umem_free(ddi_umem_cookie_t cookie);
```

cookie は、ddi_umem_alloc(9F) から返されたカーネルメモリー cookie です。

◆ ◆ ◆ 第 11 章

11 デバイスコンテキスト管理

グラフィックスハードウェア用ドライバなどの一部のデバイスドライバは、デバイスへの直接アクセス機能をユーザープロセスに提供します。これらのデバイスではしばしば、デバイスに同時にアクセスするプロセスを 1 つだけにすることが求められます。

この章では、そのようなデバイスへのアクセスをデバイスドライバが管理できるようにするための一連のインタフェースについて説明します。この章では、次の内容について説明します。

デバイスコンテキストの概要

このセクションでは、デバイスコンテキストとコンテキスト管理モデルの概要を説明します。

デバイスコンテキストとは

デバイスのコンテキストとは、デバイスハードウェアの現在の状態のことです。あるプロセスのデバイスコンテキストは、そのプロセスに代わってデバイスドライバによって管理されます。ドライバは、デバイスにアクセスするプロセスごとに異なるデバイスコンテキストを維持管理する必要があります。デバイスドライバは、あるプロセスがデバイスにアクセスするときに正しいデバイスコンテキストの復元を担当します。

コンテキスト管理モデル

フレームバッファーは、デバイスコンテキスト管理の良い例になります。高速化フレームバッファーでは、ユーザープロセスがメモリーマッピングされたアクセス経由でデバイスの制御レジスタを直接操作できます。これらのプロセスは従来のシステムコールを使用しないため、デバイスにアクセスするプロセスからデバイスドライバを呼び出す必要はありません。一方、あるプロセスがデバイスにアクセスしようとしたときに、デバイスドライバにそれが通知される必要があります。ドライバは、正しいデバイスコンテキストを復元する必要があるほか、必要とされるすべての同期機能を提供する必要もあります。

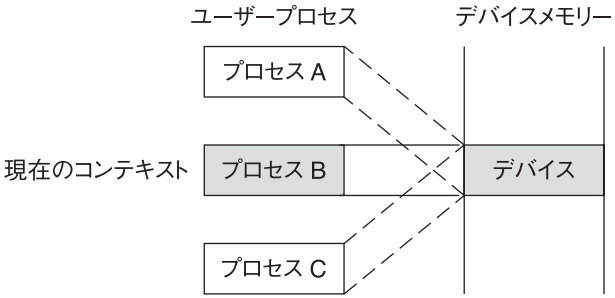
この問題を解決するため、デバイスコンテキスト管理インタフェースでは、デバイスドライバが、あるユーザープロセスがデバイスのメモリーマッピングされた領域にアクセスしたときに通知を受け取り、デバイスのハードウェアへのアクセスを制御できるようになっています。さまざまなデバイスコンテキストの同期や管理は、デバイスドライバが担当します。あるユーザープロセスがマッピングにアクセスしたときに、デバイスドライバはそのプロセスの正しいデバイスコンテキストを復元する必要があります。

ユーザープロセスが次のいずれかのアクションを実行するたびに、デバイスドライバにそれが通知されます。

- マッピングにアクセスする
- マッピングを複製する
- マッピングを解放する
- マッピングを作成する

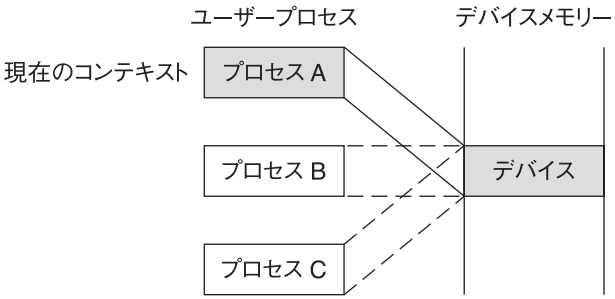
次の図は、1 つのデバイスに対して複数のユーザープロセスからメモリーマッピングが行われている様子を示したものです。ドライバがデバイスへのアクセスをプロセス B に許可したため、プロセス B がアクセスしても、その通知はドライバには送信されません。一方、プロセス A またはプロセス C のいずれかがデバイスにアクセスした場合は、引き続き、ドライバにそれが通知されます。

図 11–1　デバイスコンテキスト管理



将来のある時点で、プロセス A がデバイスにアクセスします。デバイスドライバは通知を受け、プロセス B からデバイスへの以降のアクセスをブロックします。次にドライバは、プロセス B のデバイスコンテキストを保存します。ドライバは、プロセス A のデバイスコンテキストを復元します。続いてドライバは、次の図に示すように、プロセス A にアクセスを許可します。この時点で、プロセス B またはプロセス C のいずれかがデバイスにアクセスすると、デバイスドライバにそれが通知されるようになります。

図 11–2　デバイスコンテキストがユーザープロセス A に切り替わる様子



マルチプロセッサマシンでは、複数のプロセスが同時にデバイスへのアクセスを試みる可能性があります。この状況ではスラッシングが生じる可能性があります。一部のデバイスでは、デバイスコンテキストの復元に、より長い時間が必要になります。デバイスコンテキストを実際に利用するときに使用される CPU 時間よりも、デバイスコンテキストを復元するときに使用される CPU 時間のほうが長くならないように、プロセスがデバイスにアクセスするときに最低限必要になる時間を、`devmap_set_ctx_timeout(9F)` を使用して設定できるようになっています。

カーネルは、デバイスドライバがあるプロセスにアクセスをいったん許可すると、`devmap_set_ctx_timeout(9F)` で指定された時間間隔の間、同じデバイスへのアクセスをほかのプロセスから一切要求できなくなることを保証します。

コンテキスト管理の処理

デバイスコンテキスト管理を実行する場合の基本手順は、次のとおりです。

1. devmap_callback_ctl(9S) 構造体を定義します。
2. 必要であれば、デバイスコンテキストを保存するための領域を割り当てます。
3. devmap_devmem_setup(9F) を使用して、デバイスへのユーザーマッピングとドライバ通知を設定します。
4. devmap_load(9F) と devmap_unload(9F) を使用してデバイスへのユーザーアクセスを管理します。
5. 必要であれば、デバイスコンテキスト構造体を解放します。

devmap_callback_ctl 構造体

デバイスドライバは、デバイスコンテキスト管理用のエントリポイントルーチンに関する情報をシステムに知らせるために、devmap_callback_ctl(9S) 構造体を割り当てて初期化する必要があります。

この構造体で使用される構文は、次のとおりです。

```
struct devmap_callback_ctl {
    int devmap_rev;
    int (*devmap_map)(devmap_cookie_t dhp, dev_t dev,
    uint_t flags, offset_t off, size_t len, void **pvtp);
    int (*devmap_access)(devmap_cookie_t dhp, void *pvtp,
    offset_t off, size_t len, uint_t type, uint_t rw);
    int (*devmap_dup)(devmap_cookie_t dhp, void *pvtp,
    devmap_cookie_t new_dhp, void **new_pvtp);
    void (*devmap_unmap)(devmap_cookie_t dhp, void *pvtp,
    offset_t off, size_t len, devmap_cookie_t new_dhp1,
    void **new_pvtp1, devmap_cookie_t new_dhp2,
    void **new_pvtp2);
};
```

`devmap_rev` devmap_callback_ctl 構造体のバージョン番号。このバージョン番号は DEVMAP_OPS_REV に設定する必要があります。

`devmap_map` ドライバの devmap_map(9E) エントリポイントのアドレスに設定する必要があります。

`devmap_access` ドライバの devmap_access(9E) エントリポイントのアドレスに設定する必要があります。

`devmap_dup` ドライバの devmap_dup(9E) エントリポイントのアドレスに設定する必要があります。

`devmap_unmap` ドライバの devmap_unmap(9E) エントリポイントのアドレスに設定する必要があります。

デバイスコンテキスト管理用のエントリポイント

デバイスコンテキストの管理に使用されるエントリポイントは、次のとおりです。

- devmap(9E)
- devmap_access(9E)
- devmap_contextmgt(9E)
- devmap_dup(9E)
- devmap_unmap(9E)

devmap_map() エントリポイント

devmap(9E) の構文は次のとおりです。

```
int xxdevmap_map(devmap_cookie_t handle, dev_t dev, uint_t flags,
    offset_t offset, size_t len, void **new-devprivate);
```

devmap_map() エントリポイントは、ドライバがその devmap() エントリポイントから返り、システムがデバイスメモリーへのユーザーマッピングを確立したあとで呼び出されます。devmap() エントリポイントを使用すると、ドライバは追加の処理を実行したり、マッピング固有の非公開データを割り当てたりできます。たとえば、コンテキストの切り替えをサポートするには、ドライバはコンテキスト構造体を割り当てる必要があります。続いてドライバは、そのコンテキスト構造体をマッピングに関連付ける必要があります。

システムは、ドライバが割り当て済みの非公開データへのポインタを **new-devprivate* に返すことを予期します。ドライバは、マッピングの範囲を定義する *offset* と *len* をその非公開データ内に格納する必要があります。あとでシステムが devmap_unmap(9E) を呼び出したときに、ドライバはこの情報に基づいて、マッピングの対応づけを解除する量がどれだけになるのかを判断します。

flags は、ドライバがマッピングの非公開コンテキストを割り当てるかどうかを示します。たとえば、*flags* が MAP_PRIVATE に設定されている場合、ドライバはデバイスコンテキストを格納するためのメモリー領域を割り当てることができます。MAP_SHARED が設定されている場合、ドライバは共有領域へのポインタを返します。

次の例は、devmap() エントリポイントを示したものです。ドライバは新しいコンテキスト構造体を割り当てています。続いてドライバは、エントリポイントから渡された関連パラメータを保存しています。次に、マッピングに新しいコンテキストを割り当てるために、コンテキストの割り当てを行うか、すでに存在している共有コンテキストにマッピングを関連付けています。マッピングがデバイスにアクセスする最小時間間隔は、1 ミリ秒に設定されています。

例 11–1 devmap() ルーチンの使用

```
static int
int xxdevmap_map(devmap_cookie_t handle, dev_t dev, uint_t flags,
    offset_t offset, size_t len, void **new_devprivate)
{
    struct xxstate *xsp = ddi_get_soft_state(statep,
                getminor(dev));
    struct xxctx *newctx;

    /* create a new context structure */
    newctx = kmem_alloc(sizeof (struct xxctx), KM_SLEEP);
    newctx->xsp = xsp;
    newctx->handle = handle;
    newctx->offset = offset;
    newctx->flags = flags;
    newctx->len = len;
    mutex_enter(&xsp->ctx_lock);
    if (flags & MAP_PRIVATE) {
        /* allocate a private context and initialize it */
        newctx->context = kmem_alloc(XXCTX_SIZE, KM_SLEEP);
        xxctxinit(newctx);
    } else {
        /* set a pointer to the shared context */
        newctx->context = xsp->ctx_shared;
    }
    mutex_exit(&xsp->ctx_lock);
    /* give at least 1 ms access before context switching */
    devmap_set_ctx_timeout(handle, drv_usectohz(1000));
    /* return the context structure */
    *new_devprivate = newctx;
    return(0);
}
```

devmap_access() エントリポイント

devmap_access(9E) エントリポイントは、変換が無効になっているマッピングへのアクセスが発生した場合に呼び出されます。マッピングの変換が無効化されるのは、mmap(2) への応答として devmap_devmem_setup(9F) でマッピングが作成された場合、fork(2) によってマッピングが複製された場合、または devmap_unload(9F) 呼び出しによってマッピングが明示的に無効化された場合です。

devmap_access() の構文は次のとおりです。

```
int xxdevmap_access(devmap_cookie_t handle, void *devprivate,
    offset_t offset, size_t len, uint_t type, uint_t rw);
```

各表記の意味は次のとおりです。

handle ユーザープロセスからアクセスされたマッピングのマッピングハンドル。

devprivate マッピングに関連付けられたドライバ非公開データへのポインタ。

offset アクセスされたマッピング内のオフセット。

len アクセス対象となるメモリーのバイト単位の長さ。

type アクセス操作のタイプ。

rw アクセスの向きを指定します。

システムは、devmap_access(9E) が devmap_do_ctxmgt(9F)、または devmap_default_access(9F) のいずれかを呼び出してメモリーアドレス変換をロードしたあとで、devmap_access() が復帰することを予期します。コンテキスト切り替えをサポートするマッピングの場合、デバイスドライバは devmap_do_ctxmgt() を呼び出します。このルーチンには、devmap_access(9E) からのすべてのパラメータと、コンテキスト切り替えを処理するドライバエントリポイント devmap_contextmgt(9E) へのポインタが渡されます。コンテキスト切り替えをサポートしないマッピングの場合、ドライバは devmap_default_access(9F) を呼び出します。devmap_default_access() の目的は、devmap_load(9F) を呼び出してユーザー変換をロードすることです。

次の例は、devmap_access(9E) エントリポイントを示したものです。マッピングは 2 つの領域に分割されています。オフセット OFF_CTXMG から CTXMGT_SIZE バイトの長さの領域は、コンテキスト管理をサポートします。 マッピングの残りの部分はデフォルトアクセスをサポートします。

例 11–2 devmap_access() ルーチンの使用

```
#define OFF_CTXMG      0
#define CTXMGT_SIZE    0x20000
static int
xxdevmap_access(devmap_cookie_t handle, void *devprivate,
    offset_t off, size_t len, uint_t type, uint_t rw)
{
    offset_t diff;
    int    error;

    if ((diff = off - OFF_CTXMG) >= 0 && diff < CTXMGT_SIZE) {
        error = devmap_do_ctxmgt(handle, devprivate, off,
            len, type, rw, xxdevmap_contextmgt);
    } else {
        error = devmap_default_access(handle, devprivate,
            off, len, type, rw);
    }
    return (error);
}
```

devmap_contextmgt() エントリポイント

devmap_contextmgt(9E) の構文は次のとおりです。

```
int xxdevmap_contextmgt(devmap_cookie_t handle, void *devprivate,
    offset_t offset, size_t len, uint_t type, uint_t rw);
```

devmap_contextmgt() は、デバイスに現在アクセスしているマッピングのハンドルを指定して devmap_unload(9F) を呼び出します。このアプローチにより、そのマッピングの変換が無効化されます。このアプローチは、現在のマッピングへのアクセスが次回発生したときに、そのマッピングで devmap_access(9E) 呼び出しが発生することを保証します。アクセスイベントが発生する原因となったマッピングのマッピング変換を有効化する必要があります。したがって、ドライバは、アクセスを要求しているプロセスのデバイスコンテキストを復元する必要があります。さらにドライバは、このエントリポイントへの呼び出しを生成したマッピングの *handle* を指定して devmap_load(9F) を呼び出す必要もあります。

devmap_load() 呼び出しによってマッピング変換が有効化されたマッピング部分へのアクセスが発生しても、devmap_access() への呼び出しは生成されません。その後、devmap_unload() を呼び出すと、マッピング変換が無効化されます。この呼び出しによって、devmap_access () がふたたび呼び出されるようになります。

devmap_load() または devmap_unl ad() のいずれかからエラーが返された場合、devmap_contextmgt() はただちにそのエラーを返します。デバイスドライバがデバイスコンテキストの復元中にハードウェア障害を検出した場合、-1 が返されます。それ以外の場合、アクセス要求の処理が成功したら、devmap_contextmgt() はゼロを返します。devmap_contextmgt() からゼロ以外の値が返された場合、SIGBUS または SIGSEGV がプロセスに送信されます。

次の例は、ページ数 1 のデバイスコンテキストを管理する方法を示したものです。

注 – xxctxsave() と xxctxrestore() は、コンテキストを保存および復元するためのデバイス依存関数です。xxctxsave() は、レジスタからデータを読み取り、そのデータをソフト状態構造体に保存します。 xxctxrestore() は、ソフト状態構造体に保存されたデータを取得し、そのデータをデバイスのレジスタに書き込みます。読み取り、書き込み、保存はすべて、DDI/DKI データアクセスルーチンを使用して実行します。

例 11–3 devmap_contextmgt() ルーチンの使用

```
static int
xxdevmap_contextmgt(devmap_cookie_t handle, void *devprivate,
    offset_t off, size_t len, uint_t type, uint_t rw)
{
    int    error;
    struct xxctx *ctxp = devprivate;
    struct xxstate *xsp = ctxp->xsp;
    mutex_enter(&xsp->ctx_lock);
    /* unload mapping for current context */
    if (xsp->current_ctx != NULL) {
        if ((error = devmap_unload(xsp->current_ctx->handle,
            off, len)) != 0) {
            xsp->current_ctx = NULL;
            mutex_exit(&xsp->ctx_lock);
            return (error);
```

例 11–3 devmap_contextmgt() ルーチンの使用 (続き)

```
        }
    }
    /* Switch device context - device dependent */
    if (xxctxsave(xsp->current_ctx, off, len) < 0) {
        xsp->current_ctx = NULL;
        mutex_exit(&xsp->ctx_lock);
        return (-1);
    }
    if (xxctxrestore(ctxp, off, len) < 0){
        xsp->current_ctx = NULL;
        mutex_exit(&xsp->ctx_lock);
        return (-1);
    }
    xsp->current_ctx = ctxp;
    /* establish mapping for new context and return */
    error = devmap_load(handle, off, len, type, rw);
    if (error)
        xsp->current_ctx = NULL;
    mutex_exit(&xsp->ctx_lock);
    return (error);
}
```

devmap_dup() エントリポイント

devmap_dup(9E) エントリポイントは、fork(2) を呼び出すユーザープロセスなどによってデバイスマッピングが複製されるときに呼び出されます。ドライバからは、新しいマッピングのための新しいドライバ非公開データが生成されることが予期されます。

devmap_dup() の構文は次のとおりです。

```
int xxdevmap_dup(devmap_cookie_t handle, void *devprivate,
    devmap_cookie_t new-handle, void **new-devprivate);
```

各表記の意味は次のとおりです。

handle 複製されるマッピングのマッピングハンドル。

new-handle 複製されたマッピングのマッピングハンドル。

devprivate 複製されるマッピングに関連付けられたドライバ非公開データへのポインタ。

**new-devprivate* 新しいマッピングの新しいドライバ非公開データを指すように設定します。

devmap_dup() で作成されたマッピングではデフォルトで、マッピング変換が無効化されています。マッピング変換が無効になっていると、マッピングへのアクセスがはじめて発生したときに devmap_access(9E) エントリポイントが強制的に呼び出されます。

次の例は、典型的な devmap_dup() ルーチンを示したものです。

例 11–4 devmap_dup() ルーチンの使用

```
static int
xxdevmap_dup(devmap_cookie_t handle, void *devprivate,
    devmap_cookie_t new_handle, void **new_devprivate)
{
    struct xxctx *ctxp = devprivate;
    struct xxstate *xsp = ctxp->xsp;
    struct xxctx *newctx;
    /* Create a new context for the duplicated mapping */
    newctx = kmem_alloc(sizeof (struct xxctx), KM_SLEEP);
    newctx->xsp = xsp;
    newctx->handle = new_handle;
    newctx->offset = ctxp->offset;
    newctx->flags = ctxp->flags;
    newctx->len = ctxp->len;
    mutex_enter(&xsp->ctx_lock);
    if (ctxp->flags & MAP_PRIVATE) {
        newctx->context = kmem_alloc(XXCTX_SIZE, KM_SLEEP);
        bcopy(ctxp->context, newctx->context, XXCTX_SIZE);
    } else {
        newctx->context = xsp->ctx_shared;
    }
    mutex_exit(&xsp->ctx_lock);
    *new_devprivate = newctx;
    return(0);
}
```

devmap_unmap() エントリポイント

devmap_unmap(9E) エントリポイントは、マッピングの対応づけを解除するときに呼び出されます。対応づけの解除が発生する可能性があるのは、ユーザープロセスが終了するときと、munmap(2) システムコールが呼び出されたときです。

devmap_unmap() の構文は次のとおりです。

```
void xxdevmap_unmap(devmap_cookie_t handle, void *devprivate,
    offset_t off, size_t len, devmap_cookie_t new-handle1,
    void **new-devprivate1, devmap_cookie_t new-handle2,
    void **new-devprivate2);
```

各表記の意味は次のとおりです。

handle 解放されるマッピングのマッピングハンドル。

devprivate マッピングに関連付けられたドライバ非公開データへのポインタ。

off 対応づけ解除の開始位置となる、論理デバイスメモリー内のオフセット。

len 対応づけ解除の対象となるメモリーのバイト単位の長さ。

new-handle1 *off* - 1 で終わる新しい領域を記述するためにシステムが使用するハンドル。*new-handle1* の値は NULL でもかまいません。

new-devprivate1 ドライバによる、*off* -1 で終わる新しい領域の非公開ドライバマッピングデータの設定先となるポインタ。*new-handle1* が NULL の場合、*new-devprivate1* は無視されます。

new-handle2 *off* + *len* から始まる新しい領域を記述するためにシステムが使用するハンドル。*new-handle2* の値は NULL でもかまいません。

new-devprivate2 ドライバによる、*off* + *len* から始まる新しい領域のドライバ非公開マッピングデータの設定先となるポインタ。*new-handle2* が *NULL* の場合、new-devprivate2 は無視されます。

マッピングの作成時には、devmap_unmap() ルーチンが devmap_map(9E) または devmap_dup(9E) のいずれかによって割り当てられたすべてのドライバ非公開リソースを解放することが予期されます。対応づけ解除の対象がマッピングの一部のみである場合、ドライバは、残りのマッピングに対する新しい非公開データを割り当てたあとで、古い非公開データを解放する必要があります。解放されるマッピングのハンドルが、有効な変換を含むマッピングを指している場合でも、そのハンドルを指定して devmap_unload(9F) を呼び出す必要はありません。ただし、あとで devmap_access(9E) で問題が発生しないように、現在のマッピング表現は「現在のマッピングがない」状態に設定されていることを、デバイスドライバ内で確認するようにしてください。

次の例は、典型的な devmap_unmap() ルーチンを示したものです。

例 11–5 devmap_unmap() ルーチンの使用

```
static void
xxdevmap_unmap(devmap_cookie_t handle, void *devprivate,
    offset_t off, size_t len, devmap_cookie_t new_handle1,
    void **new_devprivate1, devmap_cookie_t new_handle2,
    void **new_devprivate2)
{
    struct xxctx *ctxp = devprivate;
    struct xxstate *xsp = ctxp->xsp;
    mutex_enter(&xsp->ctx_lock);

    /*
     * If new_handle1 is not NULL, we are unmapping
     * at the end of the mapping.
     */
    if (new_handle1 != NULL) {
        /* Create a new context structure for the mapping */
        newctx = kmem_alloc(sizeof (struct xxctx), KM_SLEEP);
        newctx->xsp = xsp;
           if (ctxp->flags & MAP_PRIVATE) {
               /* allocate memory for the private context and copy it */
            newctx->context = kmem_alloc(XXCTX_SIZE, KM_SLEEP);
            bcopy(ctxp->context, newctx->context, XXCTX_SIZE);
        } else {
```

例 11–5 devmap_unmap() ルーチンの使用 (続き)

```
            /* point to the shared context */
            newctx->context = xsp->ctx_shared;
        }
        newctx->handle = new_handle1;
        newctx->offset = ctxp->offset;
        newctx->len = off - ctxp->offset;
        *new_devprivate1 = newctx;
    }
    /*
     * If new_handle2 is not NULL, we are unmapping
     * at the beginning of the mapping.
     */
    if (new_handle2 != NULL) {
        /* Create a new context for the mapping */
        newctx = kmem_alloc(sizeof (struct xxctx), KM_SLEEP);
        newctx->xsp = xsp;
        if (ctxp->flags & MAP_PRIVATE) {
            newctx->context = kmem_alloc(XXCTX_SIZE, KM_SLEEP);
            bcopy(ctxp->context, newctx->context, XXCTX_SIZE);
        } else {
            newctx->context = xsp->ctx_shared;
        }
        newctx->handle = new_handle2;
        newctx->offset = off + len;
        newctx->flags = ctxp->flags;
        newctx->len = ctxp->len - (off + len - ctxp->off);
        *new_devprivate2 = newctx;
    }
    if (xsp->current_ctx == ctxp)
        xsp->current_ctx = NULL;
    mutex_exit(&xsp->ctx_lock);
    if (ctxp->flags & MAP_PRIVATE)
        kmem_free(ctxp->context, XXCTX_SIZE);
    kmem_free(ctxp, sizeof (struct xxctx));
}
```

ユーザーマッピングとドライバ通知の関連付け

ユーザープロセスが mmap(2) を使用してデバイスへのマッピングを要求すると、ドライバの segmap(9E) エントリポイントが呼び出されます。ドライバでデバイスコンテキストを管理する必要がある場合、ドライバはメモリーマッピングの設定時に ddi_devmap_segmap(9F) または devmap_setup(9F) を使用する必要があります。どちらの関数もドライバの devmap(9E) エントリポイントを呼び出しますが、このエントリポイントでは、devmap_devmem_setup(9F) を使用してデバイスメモリーとユーザーマッピングが関連付けられます。デバイスメモリーのマッピング方法の詳細については、第 10 章「デバイスメモリーおよびカーネルメモリーのマッピング」を参照してください。

ドライバは、ユーザーマッピングへのアクセスの通知を受け取ることができるように、devmap_callback_ctl(9S) のエントリポイントの情報をシステムに知らせる必要があります。ドライバは、システムに情報を知らせるために、devmap_callback_ctl(9S) 構造体へのポインタを devmap_devmem_setup(9F) に指定します。devmap_callback_ctl(9S) 構造体は、一連のコンテキスト管理用エントリポイントを記述します。これらのエントリポイントがシステムによって呼び出され、デバイスマッピングに関するイベントを管理するようにデバイスドライバに通知されます。

システムは、各マッピングにマッピングハンドルを関連付けます。このハンドルは、コンテキスト管理用の各エントリポイントに渡されます。マッピングハンドルを使用すると、マッピング変換を無効化および有効化できます。ドライバがマッピング変換を無効化すると、その後マッピングへのアクセスが発生するたびに、それがドライバに通知されます。ドライバがマッピング変換を有効化すると、マッピングへのアクセスが発生してもドライバにはそれが通知されなくなります。マッピングの作成時には常にマッピング変換が無効化されますが、これは、マッピングへのアクセスがはじめて発生したときにドライバに通知が届くようにするためです。

次の例は、デバイスコンテキスト管理インタフェースを使用してマッピングを設定する方法を示したものです。

例 11–6　コンテキスト管理サポート付きの devmap (9E) エントリポイント

```
static struct devmap_callback_ctl xx_callback_ctl = {
    DEVMAP_OPS_REV, xxdevmap_map, xxdevmap_access,
    xxdevmap_dup, xxdevmap_unmap
};

static int
xxdevmap(dev_t dev, devmap_cookie_t handle, offset_t off,
    size_t len, size_t *maplen, uint_t model)
{
    struct xxstate *xsp;
    uint_t rnumber;
    int    error;

    /* Setup data access attribute structure */
    struct ddi_device_acc_attr xx_acc_attr = {
        DDI_DEVICE_ATTR_V0,
        DDI_NEVERSWAP_ACC,
        DDI_STRICTORDER_ACC
    };
    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL)
        return (ENXIO);
    len = ptob(btopr(len));
    rnumber = 0;
    /* Set up the device mapping */
    error = devmap_devmem_setup(handle, xsp->dip, &xx_callback_ctl,
        rnumber, off, len, PROT_ALL, 0, &xx_acc_attr);
    *maplen = len;
    return (error);
```

例 11–6 コンテキスト管理サポート付きの devmap (9E) エントリポイント (続き)

```
}
```

マッピングへのアクセスの管理

有効なマッピング変換を持たないメモリーマッピングされた領域内のアドレスにユーザープロセスがアクセスすると、デバイスドライバに通知されます。アクセスイベントが発生した場合、デバイスに現在アクセスしているプロセスのマッピング変換を無効化する必要があります。デバイスへのアクセスを要求したプロセスのデバイスコンテキストを復元する必要があります。さらに、アクセスを要求しているプロセスのマッピング変換を有効化する必要があります。

マッピング変換を有効化および無効化するには、関数 devmap_load(9F) および devmap_unload(9F) を使用します。

devmap_load() エントリポイント

devmap_load(9F) の構文は次のとおりです。

```
int devmap_load(devmap_cookie_t handle, offset_t offset,
    size_t len, uint_t type, uint_t rw);
```

devmap_load() は、handle、offset、および len で指定されるマッピングページのマッピング変換を有効化します。ドライバは、これらのページのマッピング変換を有効化することで、これらのマッピングのページへのアクセスを横取りしないようシステムに指示しています。さらにシステムは、デバイスドライバへの通知を行わないでアクセスの処理続行を許可してはいけません。

アクセスが完了するためには、アクセスイベントを生成したマッピングのオフセットとハンドルを指定して devmap_load() を呼び出す必要があります。このハンドルで devmap_load(9F) が呼び出されなかった場合、マッピング変換が有効化されないため、プロセスは SIGBUS を受信します。

devmap_unload() エントリポイント

devmap_unload(9F) の構文は次のとおりです。

```
    int devmap_unload(devmap_cookie_t handle, offset_t offset,
     size_t len);
```

devmap_unload() は、*handle*、*offset*、および *len* で指定されるマッピングページのマッピング変換を無効化します。デバイスドライバは、マッピングのこれらのページのマッピング変換を無効化することで、これらのページへのアクセスを横取りするようシステムに指示しています。さらにシステムは次回、devmap_access(9E)

エントリポイントを呼び出すことで、これらのマッピングページへのアクセスが発生したことをデバイスドライバに通知する必要があります。

どちらの関数の場合も、要求が影響を与えるのは、*offset* を含むページの全体、*offset* + *len* で示される最後のバイトを含むページの全体、およびその 2 つのページの間にあるすべてのページです。デバイスドライバは、マッピングされるデバイスメモリーの各ページについて、ある任意の時点で有効な変換を持つプロセスは 1 つのみであることを確認する必要があります。

成功時には、どちらの関数からもゼロが返されます。一方、マッピング変換を有効化または無効化するときにエラーが発生した場合、そのエラーがデバイスドライバに返されます。デバイスドライバはこのエラーをシステムに返す必要があります。

◆ ◆ ◆ 第 12 章

12 電源管理

電源管理により、コンピュータシステムまたはデバイスの電力使用を制御および管理できます。電源管理を使用すると、コンピュータがアイドル状態のときは電力消費を抑え、コンピュータが使用されていないときは完全にシャットダウンすることによってシステムのエネルギー消費を節約できます。たとえば、電力消費の大きいデスクトップコンピュータシステムが、特に夜間にアイドル状態のままになっていることがよくあります。電源管理ソフトウェアはシステムが使用されていないことを検出できます。電源管理では、検出した状態に基づいて、システムまたは一部のシステム部品の電源を切ることができます。

この章では、次の内容について説明します。

電源管理システムのフレームワーク

Oracle Solaris の電源管理フレームワークでは、デバイスドライバを利用して、デバイス単位の電源管理機能を実装します。フレームワークの実装には次の 2 種類があります。

- デバイス電源管理 – 使用されていないデバイスの電源を自動的に切り、電力消費を削減します。
- システム電源管理 – システム全体がアイドル状態のときに自動的にコンピュータの電源を切ります。

デバイス電源管理

このフレームワークでは、デバイスのアイドル状態が一定時間続いたときに、そのデバイスのエネルギー消費を削減できます。電源管理の一環として、システムソフトウェアがアイドル状態のデバイスを確認します。電源管理フレームワークは、システムソフトウェアとデバイスドライバの間で通信を行うためのインタフェースをエクスポートします。

Oracle Solaris の電源管理フレームワークは、デバイスの電源管理のために次の機能を提供します。

- 電源管理に対応したデバイス向けのデバイス非依存モデル。
- ワークステーションの電源管理を構成するための `dtpower`(1M) ツール。`power.conf`(4) および `/etc/default/power` ファイルを使用すると電源管理を実装することもできます。
- 電源管理の互換性やアイドル状態についてフレームワークに通知するための DDI インタフェース群。

システム電源管理

システム電源管理では、システムの電源を切る前にシステムの状態を保存します。これにより、ふたたび電源を入れたときに、システムを以前と同じ状態に戻すことができます。

シャットダウン前の状態に戻すことを前提にシステム全体をシャットダウンするには、次の手順を実行します。

- カーネルスレッドおよびユーザープロセスを停止します。あとからこれらのスレッドおよびプロセスを再起動します。
- システム上のすべてのデバイスのハードウェア状態をディスクに保存します。あとからこれらの状態を復元します。

SPARC のみ – システム電源管理は現在、Oracle Solaris OS がサポートする一部の SPARC システムのみに実装されています。詳細は、`power.conf`(4) のマニュアルページを参照してください。

Oracle Solaris OS のシステム電源管理フレームワークは、システム電源管理のための次の機能を提供します。

- システムのアイドル状態のプラットフォーム非依存モデル。
- ワークステーションの電源管理を構成するための `pmconfig`(1M) ツール。`power.conf`(4) および `/etc/default/power` ファイルを使用すると電源管理を実装することもできます。

- ハードウェア状態を持つドライバを調べるためのメソッドをオーバーライドする、デバイスドライバのインタフェース群。
- フレームワークからドライバを呼び出してデバイス状態を保存および復元するためのインタフェース群。
- レジューム操作が発生したことをプロセスに通知するためのメカニズム。

デバイス電源管理モデル

次のセクションでは、デバイス電源管理モデルの詳細を説明します。このモデルには次の要素が含まれます。

- 部品
- アイドル状態
- 電源レベル
- 依存関係
- ポリシー
- デバイス電源管理インタフェース
- 電源管理エントリポイント

電源管理の部品

デバイスがアイドル状態のときにデバイスの電力消費を削減できる場合、そのデバイスは電源管理に対応しています。概念的には、電源管理に対応したデバイスは、電源管理に対応した多数のハードウェアユニットで構成され、このユニットを部品と呼びます。

デバイスドライバは、デバイス部品と、各部品の電源レベルをシステムに通知します。そのため、ドライバの初期化中に、ドライバの attach(9E) エントリポイントで pm-components(9P) プロパティーが作成されます。

電源管理に対応したほとんどのデバイスは、単一部品のみを実装します。電源管理に対応した単一部品デバイスの例がディスクです。ディスクは、アイドル状態のときにスピンドルモーターを停止することによって消費電力を節減できます。

デバイスに電源管理に対応したユニットが複数あり、それらが個別に制御可能である場合、そのデバイスは複数の部品を実装します。

電源管理に対応した 2 部品デバイスの例は、モニターを備えるフレームバッファーカードです。最初の部品 [部品 0] はフレームバッファー電子回路です。使用されていないときにフレームバッファーの電力消費を削減できます。 2 番目の部品 [部品 1] はモニターです。モニターが使用されていないとき、モニターを低電力モードに切り替えることもできます。フレームバッファー電子回路とモニターは、システムからは 2 つの部品で構成される 1 つのデバイスとして認識されます。

複数の電源管理部品

電源管理フレームワークから見た場合、すべての部品が等価であり、互いに完全に独立しています。部品の状態間に完全な互換性がない場合は、デバイスドライバによって、望ましくない状態の組み合わせが発生しないようにする必要があります。たとえば、フレームバッファー/モニターカードがとりうる状態は D0、D1、D2、および D3 です。カードに接続されたモニターがとりうる状態は On、Standby、Suspend、および Off です。これらの状態が相互互換であるとは限りません。たとえば、モニターが On 状態の場合、フレームバッファーは D0 状態 (完全にオン) である必要があります。フレームバッファーが D3 状態のときに、モニターの電源を入れて On 状態にする要求をフレームバッファードライバが受け取った場合、ドライバはモニターを On に設定する前に pm_raise_power(9F) を呼び出してフレームバッファーを起動する必要があります。モニターが On 状態の間は、フレームバッファーの電源レベルを下げるシステム要求がドライバによって拒否される必要があります。

電源管理状態

デバイスの各部品の状態はビジーまたはアイドルのいずれかです。デバイスドライバは pm_busy_component(9F) および pm_idle_component(9F) を呼び出すことによって、デバイス状態の変化をフレームワークに通知します。部品が最初に作成された時点では、その部品はアイドル状態とみなされます。

電源レベル

デバイス電源管理フレームワークは、デバイスがエクスポートする pm-components プロパティーに基づいて、そのデバイスがサポートする電源レベルを認識します。電源レベルの値は正の整数である必要があります。電源レベルの解釈はデバイスドライバの開発者が決定します。pm-components プロパティーに、増分が一定の昇順で電源レベルを列挙する必要があります。電源レベル 0 は「オフ」としてフレームワークに解釈されます。依存関係に従ってフレームワークがデバイスの電源を入れる必要があるとき、フレームワークは各部品を最高の電源レベルに設定します。

次の例は、ドライバの .conf ファイルの pm-components エントリを示します。このエントリは、1 個のディスクスピンドルモーターで構成される、電源管理に対応した単一部品を実装します。ディスクスピンドルモーターは部品 0 です。スピンドルモーターは 2 段階の電源レベルをサポートします。各レベルは「停止」と「全速回転」を表します。

例 12–1　pm-component エントリの例

```
pm-components="NAME=Spindle Motor", "0=Stopped", "1=Full Speed";
```

次の例は、ドライバの attach() ルーチンで例 12–1 を実装する方法を示します。

例 12–2　pm-components プロパティーを使用した attach(9E) ルーチン

```
static char *pmcomps[] = {
    "NAME=Spindle Motor",
    "0=Stopped",
    "1=Full Speed"
};
/* ... */
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    /* ... */
    if (ddi_prop_update_string_array(DDI_DEV_T_NONE, dip,
        "pm-components", &pmcomp[0],
        sizeof (pmcomps) / sizeof (char *)) != DDI_PROP_SUCCESS)
        goto failed;
    /* ... */
```

次の例は、2 つの部品を実装するフレームバッファーを示します。部品 0 は、4 段階の電源レベルをサポートするフレームバッファー電子回路です。部品 1 は、接続されたモニターの電源管理の状態を表します。

例 12–3　複数部品の pm-components エントリ

```
pm-components="NAME=Frame Buffer", "0=Off", "1=Suspend", \
    "2=Standby", "3=On",
    "NAME=Monitor", "0=Off", "1=Suspend", "2=Standby", "3=On";
```

デバイスドライバが最初に接続された時点で、フレームワークはデバイスの電源レベルを認識していません。次のタイミングで電源遷移が発生する可能性があります。

- ドライバが pm_raise_power(9F) または pm_lower_power(9F) を呼び出す。
- 時間しきい値を超過したため、フレームワークが部品の電源レベルを下げた。
- このデバイスとの間に依存関係が存在する別のデバイスの電源レベルが変化した。220 ページの「電源管理の依存関係」を参照してください。

電源遷移のあとで、フレームワークはデバイスの各部品の電源レベルの追跡を開始します。ドライバが電源レベルをフレームワークに通知した場合にも追跡が行われます。ドライバは pm_power_has_changed(9F) を呼び出すことによって、電源レベルの変化をフレームワークに通知します。

システムは、起きる可能性のある電源遷移ごとにデフォルトのしきい値を計算します。これらのしきい値はシステムのアイドル状態しきい値に基づいています。pmconfig または power.conf(4) を使用するとデフォルトのしきい値をオーバーライドできます。部品の電源レベルが不明なときは、システムのアイドル状態しきい値に基づく別のデフォルトしきい値が使用されます。

電源管理の依存関係

デバイスの中には、別のデバイスの電源も切れているときにのみ電源を切ることが望ましいものがあります。たとえば、CD-ROM ドライブの電源を切ることが許可されていると、CD の取り出しなどの必要な機能が利用できなくなる可能性があります。

単独でデバイスの電源を切れないようにするために、あるデバイスと、電源が入っている可能性が高い別のデバイスの間に依存関係を設定できます。ユーザーがシステムを利用しているときは常にモニターの電源が入っているのが一般的なため、通常はデバイスとフレームバッファーの間に依存関係を設定します。

デバイス間の依存関係は `power.conf(4)` ファイルで指定します (デバイスツリーにおける親ノードは、その子ノードに黙示的に依存します。この依存関係は電源管理フレームワークによって自動的に処理されます)。次の形式の `power.conf(4)` エントリを使用すると、特定の依存関係を指定できます。

```
device-dependency dependent-phys-path phys-path
```

dependent-phys-path は、CD-ROM ドライブなど、電源が入った状態を維持するデバイスです。*phys-path* は、フレームバッファーなど、その電源状態が依存の対象になるデバイスを表します。

新しいデバイスをシステムに接続するたびに、エントリを `power.conf` に追加することは面倒な作業です。次の構文では、より一般的な形式で依存関係を指示できます。

```
device-dependency-property property phys-path
```

property プロパティーをエクスポートするデバイスが、*phys-path* で指定されたデバイスに依存するようにエントリを記述する必要があります。この依存関係は特にリムーバブルメディアデバイスに適用されるため、`/etc/power.conf` にはデフォルトで次の行が含まれます。

```
device_dependent-property  removable-media  /dev/fb
```

この構文は、コンソールフレームバッファーの電源も切らないかぎり、`removable-media` プロパティーをエクスポートするデバイスの電源を切ることができないことを意味します。

詳細は、`power.conf(4)` および `removable-media(9P)` のマニュアルページを参照してください。

デバイスの自動電源管理

pmconfig または power.conf(4) によって自動電源管理が有効化されている場合、pm-components(9P) プロパティーを持つすべてのデバイスは自動的に電源管理を使用します。部品がアイドル状態になってからデフォルトの許容時間を経過すると、部品の電源レベルが自動的に 1 段階下がります。デフォルトの許容時間は電源管理フレームワークによって自動的に計算され、デバイス全体が、システムのアイドル状態しきい値の範囲内でもっとも低いレベルの電源状態に設定されます。

注 – 初回出荷が 1999 年 7 月 1 日以降のすべての SPARC デスクトップシステムで、自動電源管理はデフォルトで有効です。それ以外のすべてのシステムで、この機能はデフォルトで無効です。使用中のマシンで自動電源管理が有効かどうかを調べる手順については、power.conf(4) のマニュアルページを参照してください。

フレームワークによって計算されたデフォルト設定は、power.conf(4) を使用するとオーバーライドできます。

デバイス電源管理インタフェース

電源管理に対応した部品を含むデバイスをサポートするデバイスドライバは、pm-components(9P) プロパティーを作成する必要があります。このプロパティーは、電源管理に対応した部品がデバイスに含まれることをシステムに指示します。pm-components は、利用可能な電源レベルもシステムに指示します。ドライバは通常、ドライバの attach(9E) エントリポイントから ddi_prop_update_string_array(9F) を呼び出すことによってシステムに通知します。システムへの通知に driver.conf(4) ファイルを使用する方法もあります。詳細は、pm-components(9P) のマニュアルページを参照してください。

ビジーとアイドルの状態遷移

デバイスの状態がビジーとアイドルの間で切り替わったとき、ドライバはその変化をフレームワークに継続的に通知する必要があります。これらの遷移が発生する箇所はシステムごとにまったく異なります。ビジーとアイドルの状態遷移は、デバイスの性質と、個別の部品によって抽象化される機能に依存します。たとえば、SCSI ディスクターゲットドライバは通常、SCSI ターゲットディスクドライブが回転しているかどうかを表す単一部品をエクスポートします。ドライブに対する要求が多いときは常に、部品はビジー状態とマークされます。キューの最後の要求が完了した時点で、部品はアイドル状態とマークされます。作成される部品の中には、決してビジー状態とマークされないものもあります。たとえば、pm-components(9P) によって作成される部品はアイドル状態で作成されます。

pm_busy_component(9F) および pm_idle_component(9F) インタフェースは、ビジーとアイドルの状態遷移を電源管理フレームワークに通知します。pm_busy_component(9F) の呼び出しの構文は次のとおりです。

```
int pm_busy_component(dev_info_t *dip, int component);
```

pm_busy_component(9F) は *component* をビジー状態とマークします。部品がビジー状態の間、その部品の電源を切ることは望ましくありません。部品の電源がすでに切れている場合、その部品をビジー状態とマークしても電源レベルは変化しません。このことに対処するため、ドライバは pm_raise_power(9F) を呼び出す必要があります。pm_busy_component(9F) の呼び出しは累積的であり、部品をアイドル状態にするには pm_idle_component(9F) が同じ回数だけ呼び出されている必要があります。

pm_idle_component(9F) ルーチンの構文は次のとおりです。

```
int pm_idle_component(dev_info_t *dip, int component);
```

pm_idle_component(9F) は *component* をアイドル状態とマークします。アイドル状態の部品は電源を切っても安全です。部品をアイドル状態にするには、pm_busy_component(9F) の呼び出しごとに pm_idle_component(9F) を 1 回呼び出す必要があります。

デバイスの電源状態の遷移

デバイスドライバで pm_raise_power(9F) を呼び出すと、部品を少なくとも所定の電源レベルに設定することを要求できます。電源が切れている部品を使用する前に、この方法で電源レベルを設定する必要があります。たとえば、ディスクの電源が切れている場合に、SCSI ディスクターゲットドライバの read(9E) ルーチンでディスクを回転させることが必要になったとします。pm_raise_power(9F) 関数は、デバイスの電源状態の遷移を開始し、電源レベルを上げることを電源管理フレームワークに要求します。通常、部品の電源レベルの低下はフレームワークによって開始されます。ただし、使用されていないデバイスの電源消費をできるかぎり削減するためには、デバイスドライバを切り離すときに pm_lower_power(9F) を呼び出すことが推奨されます。

一部のデバイスでは、電源レベルを下げることにリスクが伴います。たとえば、テープドライブの電源を切るとテープが損傷することがあります。同様に、パワーサイクルのたびにヘッドが接触するため、一部のディスクドライブではパワーサイクルの許容範囲が制限されています。デバイスドライバがデバイスのすべてのパワーサイクルを制御することをシステムに通知するには、no-involuntary-power-cycles(9P) プロパティーを使用します。この方法により、デバイスドライバがその detach(9E) エントリポイントから pm_lower_power(9F) を呼び出してデバイスの電源を切った場合を除いて、デバイスドライバを切り離している最中にデバイスの電源が切れるのを防ぐことができます。

操作に必要な部品の電源レベルが不足していることをドライバが検出すると、pm_raise_power(9F) 関数が呼び出されます。ドライバはこのインタフェースを使用して、部品の現在の電源レベルを必要なレベルにまで上げます。このデバイスに依存するすべてのデバイスも、この呼び出しによってフルパワーに復帰します。

デバイスへのアクセスが不要になったら、デバイスを切り離すときに pm_lower_power(9F) 関数を呼び出します。pm_lower_power (9F) を呼び出すと、各部品の電源レベルが最低に設定されるため、使用されていない間はデバイスが最小限の電力しか消費しなくなります。pm_lower_power() 関数は detach() エントリポイントから呼び出す必要があります。ドライバのほかの箇所から pm_lower_power() 関数を呼び出しても効果がありません。

pm_power_has_changed(9F) 関数は、電源遷移についてフレームワークに通知するために呼び出されます。デバイスの電源レベルの変更自体によって遷移が起きる場合もあれば、サスペンド-レジュームのような操作によって遷移が起きる場合もあります。pm_power_has_changed(9F) の構文は pm_raise_power(9F) と同じです。

power() エントリポイント

電源管理フレームワークは power(9E) エントリポイントを使用します。

power() の構文は次のとおりです。

```
int power(dev_info_t *dip, int component, int level);
```

部品の電源レベルを変更する必要があるとき、システムは power(9E) エントリポイントを呼び出します。このエントリポイントで実行される処理はデバイスドライバによって異なります。前出の SCSI ターゲットディスクドライバの例では、電源レベルを 0 に設定すると、ディスクの回転を停止する SCSI コマンドが送信されます。一方、電源レベルをフルパワーに設定すると、ディスクを回転させる SCSI コマンドが送信されます。

電源遷移が原因でデバイスの状態が失われる可能性がある場合、ドライバは必要な状態をすべてメモリーに保存し、あとから復元できるようにする必要があります。電源遷移の際に、以前に保存された状態を復元しないとデバイスがふたたび使用可能にならない場合は、ドライバがその状態を復元する必要があります。電源自動管理デバイスの状態が失われる原因となる電源遷移や、状態の復元が必要になる電源遷移の種類について、フレームワークの側では想定されていません。次の例はサンプルの power() ルーチンを示しています。

例 12–4　単一部品デバイスに対する power() ルーチンの使用

```
int
xxpower(dev_info_t *dip, int component, int level)
{
    struct xxstate *xsp;
    int instance;

    instance = ddi_get_instance(dip);
    xsp = ddi_get_soft_state(statep, instance);
   /*
    * Make sure the request is valid
    */
```

例 12–4 単一部品デバイスに対する power() ルーチンの使用 (続き)

```
    if (!xx_valid_power_level(component, level))
        return (DDI_FAILURE);
    mutex_enter(&xsp->mu);
   /*
    * If the device is busy, don't lower its power level
    */
    if (xsp->xx_busy[component] &&
        xsp->xx_power_level[component] > level) {
        mutex_exit(&xsp->mu);
        return (DDI_FAILURE);
    }

    if (xsp->xx_power_level[component] != level) {
       /*
        * device- and component-specific setting of power level
        * goes here
        */
        xsp->xx_power_level[component] = level;
    }
    mutex_exit(&xsp->mu);
    return (DDI_SUCCESS);
}
```

次の例は、2 つの部品で構成されるデバイスに対する power() ルーチンです。部品 1 がオンのときは部品 0 がオンである必要があります。

例 12–5 複数部品デバイスに対する power(9E) ルーチン

```
int
xxpower(dev_info_t *dip, int component, int level)
{
    struct xxstate *xsp;
    int instance;

    instance = ddi_get_instance(dip);
    xsp = ddi_get_soft_state(statep, instance);
   /*
    * Make sure the request is valid
    */
    if (!xx_valid_power_level(component, level))
        return (DDI_FAILURE);
    mutex_enter(&xsp->mu);
   /*
    * If the device is busy, don't lower its power level
    */
    if (xsp->xx_busy[component] &&
        xsp->xx_power_level[component] > level) {
        mutex_exit(&xsp->mu);
        return (DDI_FAILURE);
    }
   /*
    * This code implements inter-component dependencies:
    * If we are bringing up component 1 and component 0
    * is off, we must bring component 0 up first, and if
```

例 12–5 複数部品デバイスに対する power(9E) ルーチン (続き)

```
    * we are asked to shut down component 0 while component
    * 1 is up we must refuse
    */
    if (component == 1 && level > 0 && xsp->xx_power_level[0] == 0) {
        xsp->xx_busy[0]++;
        if (pm_busy_component(dip, 0) != DDI_SUCCESS) {
           /*
            * This can only happen if the args to
            * pm_busy_component()
            * are wrong, or pm-components property was not
            * exported by the driver.
            */
            xsp->xx_busy[0]--;
            mutex_exit(&xsp->mu);
            cmn_err(CE_WARN, "xxpower pm_busy_component()
                failed");
            return (DDI_FAILURE);
        }
        mutex_exit(&xsp->mu);
        if (pm_raise_power(dip, 0, XX_FULL_POWER_0) != DDI_SUCCESS)
            return (DDI_FAILURE);
        mutex_enter(&xsp->mu);
    }
    if (component == 0 && level == 0 && xsp->xx_power_level[1] != 0) {
        mutex_exit(&xsp->mu);
        return (DDI_FAILURE);
    }
    if (xsp->xx_power_level[component] != level) {
       /*
        * device- and component-specific setting of power level
        * goes here
        */
        xsp->xx_power_level[component] = level;
    }
    mutex_exit(&xsp->mu);
    return (DDI_SUCCESS);
}
```

システム電源管理モデル

このセクションでは、システム電源管理モデルについて詳しく説明します。このモデルには次のコンポーネントが含まれます。

- 自動停止しきい値
- ビジー状態
- ハードウェア状態
- ポリシー
- 電源管理エントリポイント

自動停止しきい値

構成された時間にわたってアイドル状態が続いたあとに、システムを自動的にシャットダウンする、つまりシステムの電源を切ることができます。この時間のことを自動停止しきい値と呼びます。1995 年 10 月 1 日から 1999 年 6 月 30 日までの間に初回出荷された SPARC デスクトップシステムでは、この動作がデフォルトで有効です。詳細は、power.conf(4) のマニュアルページを参照してください。dtpower(1M) または power.conf(4) を使用すると自動停止機能をオーバーライドできます。

ビジー状態

システムのビジー状態は複数の方法で測定できます。現在サポートされている組み込みの基準値項目は、キーボード入力、マウス操作、tty 入力、平均負荷率、ディスク読み取り、および NFS 要求です。これらの項目のうち 1 つでもシステムをビジー状態にする可能性があります。組み込みの基準値に加えて、システムがビジー状態であることを示すことができるユーザー指定のプロセスを実行するためのインタフェースが定義されています。

ハードウェア状態

reg プロパティーをエクスポートするデバイスには、システムをシャットダウンする前に保存しなければならないハードウェア状態があるものとみなされます。reg プロパティーのないデバイスは状態なしとみなされます。ただし、デバイスドライバによってこの想定をオーバーライドできます。

値が needs-suspend-resume の pm-hardware-state プロパティーをドライバがエクスポートする場合、状態を保存および復元するには、ハードウェア状態を持つが reg プロパティーは持たないデバイス (SCSI ドライバなど) を呼び出す必要があります。それ以外の場合、reg プロパティーがないことは、デバイスにハードウェア状態がないことを意味するものと解釈されます。デバイスのプロパティーの詳細は、第 4 章「プロパティー」を参照してください。

reg プロパティーを持ちハードウェア状態を持たないデバイスは、値が no-suspend-resume の pm-hardware-state プロパティーをエクスポートできます。pm-hardware-state プロパティーの値として no-suspend-resume を使用すると、フレームワークがその状態を保存および復元するためにドライバを呼び出すことはありません。電源管理プロパティーの詳細は、pm-components(9P) のマニュアルページを参照してください。

システムの自動電源管理

次の条件に当てはまる場合、システムはシャットダウンされます。

- dtpower(1M) または power.conf(4) によって自動停止機能が有効化された。
- 自動停止しきい値の時間 (分単位) にわたってシステムのアイドル状態が続いた。
- power.conf で指定するすべての基準値が満たされた。

システム電源管理で使用されるエントリポイント

デバイスのハードウェア状態の保存をドライバに要求するとき、システム電源管理はドライバの detach(9E) エントリポイントに DDI_SUSPEND コマンドを渡します。デバイスのハードウェア状態の復元をドライバに要求するとき、システム電源管理はドライバの attach(9E) エントリポイントに DDI_RESUME コマンドを渡します。

detach() エントリポイント

detach(9E) の構文は次のとおりです。

```
int detach(dev_info_t *dip, ddi_detach_cmd_t cmd);
```

reg プロパティーを持つデバイスと、pm-hardware-state プロパティーが needs-suspend-resume に設定されたデバイスは、デバイスのハードウェア状態を保存できる必要があります。フレームワークはドライバの detach(9E) エントリポイントを呼び出して、ドライバで状態を保存し、システムの電源が復旧したときに状態を復元できるようにします。DDI_SUSPEND コマンドを処理するために、detach(9E) は次のタスクを実行する必要があります。

- デバイスがレジュームされるまで、dump(9E) 要求を除き、後続の操作が開始されないようにします。
- 未処理の操作が完了するまで待ちます。未処理の操作を再開できる場合は、ユーザーがその操作を中止できます。
- 保留中のタイムアウトおよびコールバックをすべてキャンセルします。
- 揮発性のハードウェア状態をすべてメモリーに保存します。状態にはデバイスレジスタの内容が含まれ、ダウンロードされたファームウェアが含まれることもあります。

デバイスをサスペンドしてその状態をメモリーに保存する処理をドライバで実行できない場合、ドライバは DDI_FAILURE を返す必要があります。フレームワークはそれ以降のシステム電源管理操作を中止します。

デバイスの電源を切ることに何らかのリスクを伴う場合があります。たとえば、テープが入った状態でテープドライブの電源を切ると、テープが損傷する可能性があります。そのような場合、attach(9E) で次の処理を実行する必要があります。

- ddi_removing_power(9F) を呼び出して、DDI_SUSPEND コマンドによりデバイスの電源が切れる可能性があるかどうかを調べます。
- 電源を切った場合に問題が起きる可能性があるかどうかを調べます。

両方の可能性がある場合、DDI_SUSPEND 要求を拒否する必要があります。例 12–6 は、attach(9E) ルーチンで ddi_removing_power(9F) を使用して、DDI_SUSPEND コマンドが問題を引き起こすかどうかを調べる方法を示しています。

ダンプ要求は尊重される必要があります。フレームワークでは dump(9E) エントリポイントを使用して、メモリーの内容を格納した状態ファイルを書き出します。このエントリポイントの使用時にデバイスドライバに課せられる制限については、dump(9E) のマニュアルページを参照してください。

電源管理に対応した部品の detach(9E) エントリポイントを呼び出すとき、DDI_SUSPEND コマンドを使用すると、デバイスの電源が切れるときに状態が保存されます。ドライバは保留中のタイムアウトをキャンセルします。またドライバは、dump(9E) 要求を除いて、pm_raise_power(9F) の一切の呼び出しを抑制します。DDI_RESUME コマンドを使用した attach(9E) の呼び出しによってデバイスがレジュームされると、タイムアウトおよび pm_raise_power () の呼び出しもレジューム可能になります。このような可能性に適切に対処できるよう、ドライバはその状態の十分な追跡を継続する必要があります。次の例は、DDI_SUSPEND コマンドを実装した detach(9E) ルーチンを示しています。

例 12–6 DDI_SUSPEND を実装する detach(9E) ルーチン

```
int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    struct xxstate *xsp;
    int instance;

    instance = ddi_get_instance(dip);
    xsp = ddi_get_soft_state(statep, instance);

    switch (cmd) {
    case DDI_DETACH:
       /* ... */
    case DDI_SUSPEND:
       /*
        * We do not allow DDI_SUSPEND if power will be removed and
        * we have a device that damages tape when power is removed
        * We do support DDI_SUSPEND for Device Reconfiguration.
        */
        if (ddi_removing_power(dip) && xxdamages_tape(dip))
            return (DDI_FAILURE);
        mutex_enter(&xsp->mu);
        xsp->xx_suspended = 1;  /* stop new operations */
       /*
        * Sleep waiting for all the commands to be completed
        *
        * If a callback is outstanding which cannot be cancelled
        * then either wait for the callback to complete or fail the
```

例 12–6 DDI_SUSPEND を実装する detach(9E) ルーチン (続き)

```
        * suspend request
        *
        * This section is only needed if the driver maintains a
        * running timeout
        */
        if (xsp->xx_timeout_id) {
            timeout_id_t temp_timeout_id = xsp->xx_timeout_id;

            xsp->xx_timeout_id = 0;
            mutex_exit(&xsp->mu);
            untimeout(temp_timeout_id);
            mutex_enter(&xsp->mu);
        }
        if (!xsp->xx_state_saved) {
           /*
            * Save device register contents into
            * xsp->xx_device_state
            */
        }
        mutex_exit(&xsp->mu);
        return (DDI_SUCCESS);
    default:
        return (DDI_FAILURE);
}
```

attach() エントリポイント

attach(9E) の構文は次のとおりです。

```
int attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
```

システムの電源が復旧すると、reg プロパティーを持つか、またはpm-hardware-state プロパティーの値が needs-suspend-resume である各デバイスの attach(9E) エントリポイントが、コマンド値 DDI_RESUME を使用して呼び出されます。システムのシャットダウンが中止された場合、電源が切れなかったにもかかわらず、サスペンドされた各ドライバがレジュームのために呼び出されます。したがって、attach(9E) のレジュームコードは、システムの電源が実際に切れたかどうかを想定しないものにする必要があります。

DDI_RESUME の時点で、電源管理フレームワークは部品の電源レベルを不明とみなします。デバイスの性質に応じて、ドライバの開発者には 2 つの選択肢があります。

- 部品の電源を入れなくても、レジスタを読み取るなどしてデバイス部品の実際の電源レベルをドライバで特定できる場合、ドライバで pm_power_has_changed(9F) を呼び出して、各部品の電源レベルをフレームワークに通知します。
- ドライバで部品の電源レベルを特定できない場合、ドライバでは各部品の電源レベルを内部的に不明とマークし、各部品に最初にアクセスする前に pm_raise_power(9F) を呼び出します。

次の例は、DDI_RESUME コマンドを使用した attach(9E) ルーチンを示します。

例 12–7 DDI_RESUME を実装する attach(9E) ルーチン

```
int
xxattach(devinfo_t *dip, ddi_attach_cmd_t cmd)
{
    struct xxstate *xsp;
    int    instance;

    instance = ddi_get_instance(dip);
    xsp = ddi_get_soft_state(statep, instance);

    switch (cmd) {
    case DDI_ATTACH:
    /* ... */
    case DDI_RESUME:
        mutex_enter(&xsp->mu);
        if (xsp->xx_pm_state_saved) {
            /*
             * Restore device register contents from
             * xsp->xx_device_state
             */
        }
       /*
        * This section is optional and only needed if the
        * driver maintains a running timeout
        */
        xsp->xx_timeout_id = timeout( /* ... */ );

        xsp->xx_suspended = 0;        /* allow new operations */
        cv_broadcast(&xsp->xx_suspend_cv);
       /* If it is possible to determine in a device-specific
        * way what the power levels of components are without
        * powering the components up,
        * then the following code is recommended
        */
        for (i = 0; i < num_components; i++) {
            xsp->xx_power_level[i] = xx_get_power_level(dip, i);
            if (xsp->xx_power_level[i] != XX_LEVEL_UNKNOWN)
                (void) pm_power_has_changed(dip, i,
                    xsp->xx_power_level[i]);
        }
        mutex_exit(&xsp->mu);
        return(DDI_SUCCESS);
    default:
        return(DDI_FAILURE);
    }
}
```

注 – detach(9E) および attach(9E) インタフェースを使用すると、休止されているシステムをレジュームすることもできます。

電源管理のデバイスアクセスの例

電源管理がサポートされており、例 12–6 および例 12–7 のように detach(9E) および attach(9E) が使用されている場合、read(2)、write(2)、ioctl(2) などのユーザーコンテキストからデバイスへのアクセスを実行できます。

次の例はこのアプローチを示しています。この例の前提として、操作を実行するためには、部品 component が電源レベル level で動作している必要があります。

例 12–8 デバイスアクセス

```
mutex_enter(&xsp->mu);
/*
 * Block command while device is suspended by DDI_SUSPEND
 */
while (xsp->xx_suspended)
    cv_wait(&xsp->xx_suspend_cv, &xsp->mu);
/*
 * Mark component busy so xx_power() will reject attempt to lower power
 */
xsp->xx_busy[component]++;
if (pm_busy_component(dip, component) != DDI_SUCCESS) {
    xsp->xx_busy[component]--;
   /*
    * Log error and abort
    */
}
if (xsp->xx_power_level[component] < level) {
    mutex_exit(&xsp->mu);
    if (pm_raise_power(dip, component, level) != DDI_SUCCESS) {
       /*
        * Log error and abort
        */
    }
    mutex_enter(&xsp->mu);
}
```

デバイス操作が完了したら、デバイスの割り込みハンドラなどで次の例のコードフラグメントを使用できます。

例 12–9 デバイス操作完了

```
/*
 * For each command completion, decrement the busy count and unstack
 * the pm_busy_component() call by calling pm_idle_component(). This
 * will allow device power to be lowered when all commands complete
 * (all pm_busy_component() counts are unstacked)
 */
xsp->xx_busy[component]--;
if (pm_idle_component(dip, component) != DDI_SUCCESS) {
    xsp->xx_busy[component]++;
   /*
    * Log error and abort
    */
```

例 12–9 デバイス操作完了 (続き)

```
}
/*
 * If no more outstanding commands, wake up anyone (like DDI_SUSPEND)
 * waiting for all commands to  be completed
 */
```

電源管理の制御フロー

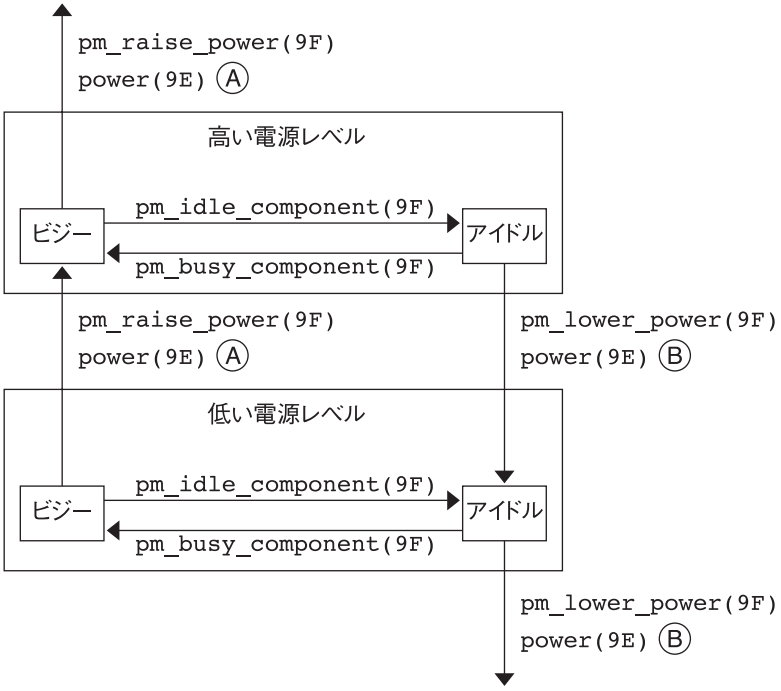
図 12–1 に、電源管理フレームワークの制御フローを示します。

部品の動作が完了したら、ドライバは pm_idle_component(9F) を呼び出して部品をアイドル状態とマークできます。しきい値として設定された時間だけ部品のアイドル状態が続いたら、フレームワークは部品の電源レベルを 1 段階下げることができます。フレームワークは power(9E) 関数を呼び出して、サポートされている 1 段階下の電源レベルに部品を設定します (そのレベルが存在する場合)。部品がビジー状態のとき、ドライバの power(9E) 関数では、その部品の電源レベルを下げようとする試みを拒否する必要があります。power(9E) 関数では、下のレベルに状態を遷移する前に、その遷移によって失われる可能性があるすべての状態を保存する必要があります。

部品の電源レベルを上げる必要があるとき、ドライバは pm_busy_component(9F) を呼び出します。この呼び出しは、フレームワークが電源レベルをそれ以上下げないようにしてから、部品に対して pm_raise_power(9F) を呼び出します。フレームワークは次に、pm_raise_power(9F) の呼び出しが戻る前に、power(9E) を呼び出して部品の電源レベルを上げます。ドライバの power(9E) のコードでは、低い電源レベルへの遷移時に失われるが、高い電源レベルでは必要であるすべての状態を復元する必要があります。

ドライバを切り離すとき、ドライバでは各部品に対して pm_lower_power(9F) を呼び出し、電源レベルをそれぞれ最低段階まで下げます。フレームワークは次に、pm_lower_power(9F) の呼び出しが戻る前に、ドライバの power(9E) ルーチンを呼び出して部品の電力レベルを下げます。

図 12–1 電源管理の概念を示す状態図



Ⓐ power(9E) は依存関係に従って部品の電源レベルを上げるためにフレームワークによって呼び出されることも、ドライバの pm_raise_power(9F) の呼び出しの結果としてフレームワークによって呼び出されることもあります。

Ⓑ power(9E) はデバイスのアイドル状態の結果として部品の電源レベルを下げるためにフレームワークによって呼び出されることも、デバイスを切り離すときにドライバの pm_lower_power(9F) の呼び出しの結果としてフレームワークによって呼び出されることもあります。

注:
9E ルーチンは必ずフレームワークによって呼び出されます。
9F ルーチンは必ずドライバによって呼び出されます。

電源管理インタフェースの変更点

Solaris 8 よりも前のリリースでは、デバイスの電源管理は自動ではありませんでした。電源を管理するデバイスごとに、開発者が /etc/power.conf にエントリを追加する必要がありました。フレームワークでは、すべてのデバイスが 2 つの電源レベル (0 および標準電力) のみをサポートすることを想定していました。

電源では、他のすべての部品が暗示的に部品 0 に依存することを想定していました。部品 0 がレベル 0 に変化すると、DDI_PM_SUSPEND コマンドを指定してドライバの detach(9E) が呼び出され、ハードウェアの状態を保存していました。部品 0 がレベル

0 から変化すると、DDI_PM_RESUME コマンドを指定して attach(9E) ルーチンが呼び出され、ハードウェアの状態を復元していました。

次のインタフェースおよびコマンドは廃止済みですが、バイナリ目的のために引き続きサポートされています。

- ddi_dev_is_needed(9F)
- pm_create_components(9F)
- pm_destroy_components(9F)
- pm_get_normal_power(9F)
- pm_set_normal_power(9F)
- DDI_PM_SUSPEND
- DDI_PM_RESUME

Solaris 8 リリース以降、autopm が有効な場合、pm-components プロパティーをエクスポートするデバイスは自動的に電源管理を使用します。

現在のフレームワークは、pm-components プロパティーに基づいて、各デバイスがサポートする電源レベルを認識します。

フレームワークでは、デバイスのさまざまな部品間の依存関係についての想定を行いません。電源レベルが変化したときに、必要に応じてハードウェアの状態を保存および復元する処理はデバイスドライバが担います。

これらの変更により、電源管理フレームワークで最新のデバイス技術に対応できるようになっています。現在の電源管理は、電力の節減効果が大幅に向上しています。電力を節減可能なデバイスをフレームワークで自動検出できます。フレームワークはデバイスの中間的な電源状態を使用できます。システム全体の電源を切らなくても、また何らかの機能を使用しなくても、システムでエネルギー消費目標を達成できます。

表 12–1 電源管理インタフェース

削除されたインタフェース	代替インタフェース
pm_create_components(9F)	pm-components(9P)
pm_set_normal_power(9F)	pm-components(9P)
pm_destroy_components(9F)	なし
pm_get_normal_power(9F)	なし
ddi_dev_is_needed(9F)	pm_raise_power(9F)
なし	pm_lower_power(9F)
なし	pm_power_has_changed(9F)
DDI_PM_SUSPEND	なし

表 12–1 電源管理インタフェース (続き)

削除されたインタフェース	代替インタフェース
DDI_PM_RESUME	なし

◆ ◆ ◆ 第 13 章

13 Oracle Solaris ドライバの強化

障害管理アーキテクチャー (FMA) の入出力障害サービスを利用することで、ドライバ開発者は障害管理機能を入出力デバイスドライバに統合できます。Oracle Solaris の入出力障害サービスフレームワークは、すべてのドライバがエラー処理のための基本的なタスクおよびアクティビティーを調整し、実行できるようにする一連のインタフェースを定義します。Oracle Solaris FMA 全体で、応答と回復に加えて、エラー処理と障害診断のための機能を提供します。FMA は、予測的自己修復戦略のコンポーネントです。

エラー処理と診断のための入出力障害サービスフレームワークに加えて、このドキュメントで説明する防御的プログラミング手法をドライバで使用すると、そのドライバは強化されていると見なされます。ドライバ強化テストハーネスは、入出力障害サービスおよび防御的プログラミングの要件が正しく満たされていることをテストします。

このドキュメントの内容は次のセクションのとおりです。

- 238 ページの「Oracle 障害管理アーキテクチャーの入出力障害サービス」では、障害管理機能を入出力デバイスドライバに統合しようとしているドライバ開発者向けのリファレンスを提供します。
- 260 ページの「Oracle Solaris デバイスドライバの防御的プログラミング手法」では、Oracle Solaris デバイスドライバを防御的に記述する方法に関する一般的な情報を提供します。
- 266 ページの「ドライバ強化テストハーネス」では、開発中のドライバがハードウェアにアクセスするときに擬似的なハードウェア障害を投入するドライバ開発ツールについて説明します。

Oracle 障害管理アーキテクチャーの入出力障害サービス

このセクションでは、障害管理のためのエラー報告、エラー処理、および診断の各機能を入出力デバイスドライバに統合する方法について説明します。このセクションでは、入出力障害サービスフレームワークの詳細な説明と、デバイスドライバの内部で入出力障害サービスの API を利用する方法を示します。

このセクションの内容は次のとおりです。

- 238 ページの「予測的自己修復とは」では、Sun 障害管理アーキテクチャーの技術的背景および概要を示します。
- 239 ページの「Oracle Solaris Fault Manager」では、Oracle Solaris Fault Manager (`fmd`(1M)) の概要を中心に、追加の背景を解説します。
- 242 ページの「エラー処理」はドライバ開発者にとってもっとも重要なセクションです。このセクションでは、高可用性のための最適なコーディング手法と、ドライバコードで入出力障害サービスを使用して FMA と対話する方法を重点的に解説します。
- 257 ページの「障害の診断」では、ドライバによって検出されるエラーをもとに障害が診断されるしくみについて説明します。
- 259 ページの「イベントレジストリ」では、Sun のイベントレジストリについての情報を提供します。

予測的自己修復とは

従来、システムはハードウェアおよびソフトウェアのエラー情報を、syslog メッセージの形式で人間の管理者および管理ソフトウェアに直接エクスポートしていました。多くの場合、エラーの検出、診断、報告、および処理のロジックは各ドライバのコードに組み込まれていました。

Oracle Solaris OS の予測的自己修復システムは、最初に登場した最先端の自己診断機能です。自己診断とは、監視された症状から問題を自動的に診断するための技術をシステムが提供し、その診断結果を利用して自動的な応答および回復を開始できることを意味します。ハードウェアの障害またはソフトウェアの不具合は、エラーと呼ばれる、監視対象の一連の症状と関連付けることができます。エラーの監視結果としてシステムによって生成されるデータは、エラーレポートまたは *ereport* と呼ばれます。

自己修復機能を備えるシステムでは、ereport がシステムによって取り込まれ、名前と値のペアの集合としてエンコードされます。これらのペアは、*ereport* イベントを形成するために拡張イベントプロトコルによって記述されます。ereport イベントおよびその他のデータは自己修復を促進するために収集され、診断エンジンと呼ばれるソフトウェアコンポーネントにディスパッチされます。このエンジンは、システムによって監視されるエラーの症状に対応する根本的な問題を診断するように設計

されています。診断エンジンはバックグラウンドで動作し、診断を生成または障害を予測できるまで、エラーリモート測定を暗黙のうちに使用します。

リモート測定の処理が完了して結論が得られたら、診断エンジンはフォルトイベントと呼ばれる別のイベントを生成します。生成されたフォルトイベントは、その特定のフォルトイベントの配信対象であるすべてのエージェントにブロードキャストされます。エージェントは、回復処理を開始し、特定のフォルトイベントに応答するソフトウェアコンポーネントです。Oracle Solaris Fault Manager (fmd(1M)) と呼ばれるソフトウェアコンポーネントは、ereport ジェネレータ、診断エンジン、およびエージェントソフトウェア間でのイベントの多重化を管理します。

Oracle Solaris Fault Manager

Oracle Solaris Fault Manager (fmd(1M)) は、インバウンドのエラーリモート測定イベントを適切な診断エンジンにディスパッチします。診断エンジンは、エラー症状の発生原因となっているハードウェアの障害またはソフトウェアの不具合を識別します。fmd(1M) デーモンは、Oracle Solaris OS による Falut Manager の実装です。このデーモンはブート時にブートし、システム上で利用可能なすべての診断エンジンおよびエージェントをロードします。Oracle Solaris Fault Manager は、システム管理者およびサービス担当者が障害管理アクティビティーを監視するためのインタフェースも提供します。

診断、疑いリスト、フォルトイベント

診断が完了すると、*list.suspect* イベントの形式で診断結果が出力されます。list.suspect イベントは、可能性のある 1 つ以上の障害または不具合イベントで構成されるイベントです。診断では、エラーの原因を 1 つの障害または不具合に絞り込めない場合があります。たとえば、コントローラをメインシステムバスに接続する配線の故障が根本的な問題である可能性があります。バス上の部品に問題がある可能性もあれば、バス自体に問題がある可能性もあります。この特定のケースでは、list.suspect イベントには複数のフォルトイベントが含まれています。バスに接続されたコントローラーごとのイベントと、バス自体に対応する 1 つのイベントです。

フォルトイベントには、診断された障害の記述に加えて、診断の適用対象となる 4 種類のペイロードメンバーも含まれています。

- リソースは障害があると診断されたコンポーネントです。fmdump(1M) コマンドでは、このペイロードメンバーは「Problem in」(問題の箇所) と表示されます。
- 自動システム回復ユニット (ASRU) は、エラー症状の発生を止めるために無効化する必要があるハードウェアまたはソフトウェアコンポーネントです。fmdump(1M) コマンドでは、このペイロードメンバーは「Affects」(影響先) と表示されます。
- 現場交換可能ユニット (FRU) は、エラーの原因となっている問題を解決するために交換または修理する必要があるコンポーネントです。

- ラベルペイロードは、シャーシまたはマザーボードに (たとえば、DIMM スロットや PCI カードスロットの隣に) 印刷されているのと同じ形式で FRU の位置を示す文字列です。fmdump コマンドでは、このペイロードメンバーは「Location」(位置) と表示されます。

たとえば、特定のメモリー位置に関して一定時間内に一定数の訂正可能 ECC エラーを受信したあとに、CPU およびメモリーの診断エンジンは DIMM 障害の診断 (list.suspect イベント) を発行します。

```
# fmdump -v -u 38bd6f1b-a4de-4c21-db4e-ccd26fa8573c
TIME                 UUID                                 SUNW-MSG-ID
Oct 31 13:40:18.1864 38bd6f1b-a4de-4c21-db4e-ccd26fa8573c AMD-8000-8L
100%  fault.cpu.amd.icachetag

Problem in: hc:///motherboard=0/chip=0/cpu=0
Affects: cpu:///cpuid=0
FRU: hc:///motherboard=0/chip=0
Location: SLOT 2
```

この例では、fmd(1M) によってリソースの問題、具体的には CPU (hc:///motherboard=0/chip=0/cpu=0) の問題が識別されました。以後のエラー症状を抑制し、訂正不能エラーの発生を防ぐために、ASRU (cpu:///cpuid=0) がリタイアメントの対象として識別されます。交換が必要なコンポーネントは FRU (hc:///motherboard=0/chip=0) です。

応答エージェント

エージェントは、診断または修復に応答してアクションを実行するソフトウェアコンポーネントです。たとえば、CPU およびメモリーのリタイアエージェントは、fault.cpu.* イベントを含む list.suspect イベントに応答するように設計されています。cpumem-retire エージェントは、CPU のオフライン化、または物理メモリーページのサービスからのリタイアメントを試みます。エージェントが成功すると、リタイアメントに成功したページまたは CPU のエントリが Fault Manager の ASRU キャッシュに追加されます。次の例で示すように、fmadm(1M) ユーティリティーは、障害があると診断されたメモリーランクのエントリを表示します。システムによってオフライン化、リタイアメント、または無効化できない ASRU のエントリも ASRU キャッシュに存在しますが、これらの ASRU は縮退状態と認識されます。縮退とは、ASRU に関連付けられたリソースに障害があるが、その ASRU をサービスから除去できない状態のことです。Oracle Solaris のエージェントソフトウェアは現時点で、入出力 ASRU (デバイスインスタンス) を扱うことができません。キャッシュに存在する、障害のある入出力リソースのエントリはすべて縮退状態です。

```
# fmadm faulty
   STATE RESOURCE / UUID
-------- ------------------------------------------------------------------------
degraded mem:///motherboard=0/chip=1/memory-controller=0/dimm=3/rank=0
         ccae89df-2217-4f5c-add4-d920f78b4faf
-------- ------------------------------------------------------------------------
```

リタイアエージェントの第一の目的は、障害があると診断されたハードウェアまたはソフトウェアを切り離す (サービスから安全に除去する) ことです。

エージェントはほかにも、次のような重要なアクションを実行します。

- SNMP トラップを介してアラートを送信します。これにより、診断を SNMP 用のアラートに変換し、既存のソフトウェアメカニズムに組み込むことができます。
- syslog メッセージを送信します。メッセージ固有の診断 (たとえば、syslog メッセージエージェント) では、診断結果を受け取ってそれを syslog メッセージに変換することができ、管理者はこのメッセージに基づいて特定のアクションを実行できます。
- FRUID の更新などのその他のエージェントアクション。プラットフォームごとに固有の応答エージェントを用意できます。

メッセージ ID と辞書ファイル

syslog メッセージエージェントは、診断の出力 (list.suspect イベント) を取得し、個別のメッセージをコンソールまたは `/var/adm/messages` に出力します。コンソールのメッセージは理解しにくいことがよくあります。FMA にはこの問題に対処するために、list.suspect イベントが syslog メッセージに配信されるたびに生成される、定義済みのフォルトメッセージ構造体が用意されています。

syslog エージェントはメッセージ識別子 (MSG ID) を生成します。イベントレジストリが生成する辞書ファイル (`.dict` ファイル) は、list.suspect イベントと構造化メッセージ識別子を対応づけます。この識別子は、関連付けられたナレッジ記事を識別して表示するために使用されます。メッセージファイル (`.po` ファイル) は、診断エンジンが生成する可能性がある、障害疑いの全リストを対象に、メッセージ ID とローカライズされたメッセージを対応づけます。次に示すのは、テストシステム上で出力されるフォルトメッセージの例です。

```
SUNW-MSG-ID: AMD-8000-7U, TYPE: Fault, VER: 1, SEVERITY: Major
EVENT-TIME: Fri Jul 28 04:26:51 PDT 2006
PLATFORM: Sun Fire V40z, CSN: XG051535088, HOSTNAME: parity
SOURCE: eft, REV: 1.16
EVENT-ID: add96f65-5473-69e6-dbe1-8b3d00d5c47b
DESC: The number of errors associated with this CPU has exceeded
acceptable levels. Refer to http://sun.com/msg/AMD-8000-7U for
more information.
AUTO-RESPONSE: An attempt will be made to remove this CPU from service.
IMPACT: Performance of this system may be affected.
REC-ACTION: Schedule a repair procedure to replace the affected CPU.
Use fmdump -v -u <EVENT_ID> to identify the module.
```

システムトポロジ

障害が発生した可能性のある箇所を識別するために、診断エンジンは、特定のソフトウェアまたはハードウェアシステムを表現したトポロジを備える必要があります。`fmd(1M)` デーモンは、診断中に使用できるトポロジのスナップショットへのハン

ドルを診断エンジンに提供します。トポロジ情報は、個々のフォルトイベントで見つかったリソース、ASRU、および FRU を表現するために使用されます。トポロジはプラットフォームラベル、FRUID、およびシリアル番号 ID を格納する目的にも使用できます。

フォルトイベントのリソースペイロードメンバーは常に、外側のプラットフォームシャーシからの物理的なパス位置によって表されます。たとえば、メインシステムバスから PCI ローカルバスにブリッジされる PCI コントローラ機能は、次に示すように、その hc スキームパス名によって表されます。

```
hc:///motherboard=0/hostbridge=1/pcibus=0/pcidev=13/pcifn=0
```

フォルトイベントの ASRU ペイロードメンバーは通常、ハードウェアコントローラ、デバイス、または機能と結びついた Oracle Solaris デバイスツリーインスタンス名によって表されます。FMA では dev スキームを使用して、ASRU をそのネイティブ形式で表現します。この形式は、入出力デバイス専用に設計されたリタイアエージェントの将来の実装によって実行される可能性があるアクションに対応しています。

```
dev:////pci@1e,600000/ide@d
```

フォルトイベントの FRU ペイロードの表現は、障害と診断された入出力リソースにもっとも近い交換可能部品によって異なります。たとえば、組み込み PCI コントローラの故障のフォルトイベントでは、次のように、交換が必要な FRU としてシステムのマザーボードが指定される場合があります。

```
hc:///motherboard=0
```

ラベルペイロードは、シャーシまたはマザーボードに (たとえば、DIMM スロットや PCI カードスロットの隣に) 印刷されているのと同じ形式で FRU の位置を示す文字列です。

```
Label: SLOT 2
```

エラー処理

このセクションでは、ドライバの内部で入出力障害サービスの API を使用してエラーを処理する方法について説明します。このセクションでは、ドライバで障害管理機能を指定および初期化する方法、エラーレポートを生成する方法、および、ドライバのエラーハンドラルーチンを登録する方法を説明します。

FMA エラーレポートリモート測定を提供するように設計されたドライバは、エラーを検出し、それらのエラーがドライバによって提供されるサービスに及ぼす影響を判断します。ドライバが、エラーの検出後にそのサービスが影響を受けたタイミングと影響の程度を特定するようにしてください。

入出力ドライバは、検出されたエラーにただちに応答する必要があります。適切な応答には次のものがあります。

- 回復を試みる
- 入出力トランザクションを再試行する
- フェイルオーバー手法を試みる
- 呼び出し側のアプリケーション/スタックにエラーを報告する
- ほかのどの方法によってもエラーを抑制できない場合、パニック状態に移行する

ドライバによって検出されたエラーは *ereport* として障害管理デーモンに伝達されます。ereport は、FMA イベントプロトコルによって定義された構造化イベントです。イベントプロトコルは各種の共通データフィールドの仕様であり、障害疑いのリストだけでなく、すべてのエラーイベントおよびフォルトイベントはこのプロトコルを使用して記述する必要があります。ereport は収集されてエラーリモート測定のフローにまとめられ、診断エンジンにディスパッチされます。

障害管理機能の宣言

強化されたデバイスドライバでは、その障害管理機能を入出力障害管理フレームワークに宣言する必要があります。ドライバの障害管理機能を宣言するには、`ddi_fm_init(9F)` 関数を使用します。

```
void ddi_fm_init(dev_info_t *dip, int *fmcap, ddi_iblock_cookie_t *ibcp)
```

`ddi_fm_init()` 関数は、カーネルコンテキストからドライバの `attach(9E)` または `detach(9E)` エントリポイントで呼び出すことができます。`ddi_fm_init()` 関数は通常、`attach()` エントリポイントから呼び出されます。`ddi_fm_init()` 関数は、*fmcap* に従ってリソースを割り当てて初期化します。*fmcap* パラメータには、次に示す障害管理機能のビット単位の論理和を設定する必要があります。

- `DDI_FM_EREPORT_CAPABLE` - ドライバはエラー条件を検出したときに FMA プロトコルのエラーイベント (ereport) を生成する必要があり、そのための能力を備えています。
- `DDI_FM_ACCCHK_CAPABLE` - ドライバは 1 つ以上のアクセス入出力トランザクションが完了したときにエラーをチェックする必要があり、そのための能力を備えています。
- `DDI_FM_DMACHK_CAPABLE` - ドライバは 1 つ以上の DMA 入出力トランザクションが完了したときにエラーをチェックする必要があり、そのための能力を備えています。
- `DDI_FM_ERRCB_CAPABLE` - ドライバはエラーコールバック関数を備えています。

強化されたリーフドライバでは通常、これらすべての機能を設定します。ただし、そのドライバの親ネクサスが要求された機能をどれもサポートしない場合、機能に関連付けられたビットがクリアされてからドライバに返されます。`ddi_fm_init(9F)` から戻る前に、一連の障害管理機能プロパティー (`fm-ereport-capable`、`fm-accchk-capable`、`fm-dmachk-capable`、および

fm-errcb-capable) を作成します。現在サポートされている障害管理機能のレベルは、prtconf(1M) コマンドを使用すると確認できます。

選択した障害管理機能をドライバでサポートするには、driver.conf(4) ファイルで障害管理機能レベルのプロパティーをエクスポートし、先に説明した値を設定します。fm-capable プロパティーの設定および読み取りは、目的の機能リストを指定して ddi_fm_init() を呼び出す前に行われる必要があります。

bge ドライバからの次の例では bge_fm_init() 関数が示されていますが、この関数から ddi_fm_init(9F) 関数が呼び出されています。bge_fm_init() 関数は bge_attach() 関数内で呼び出されます。

```
static void
bge_fm_init(bge_t *bgep)
{
        ddi_iblock_cookie_t iblk;

        /* Only register with IO Fault Services if we have some capability */
        if (bgep->fm_capabilities) {
                bge_reg_accattr.devacc_attr_access = DDI_FLAGERR_ACC;
                dma_attr.dma_attr_flags = DDI_DMA_FLAGERR;
                /*
                 * Register capabilities with IO Fault Services
                 */
                ddi_fm_init(bgep->devinfo, &bgep->fm_capabilities, &iblk);
                /*
                 * Initialize pci ereport capabilities if ereport capable
                 */
                if (DDI_FM_EREPORT_CAP(bgep->fm_capabilities) ||
                    DDI_FM_ERRCB_CAP(bgep->fm_capabilities))
                        pci_ereport_setup(bgep->devinfo);
                /*
                 * Register error callback if error callback capable
                 */
                if (DDI_FM_ERRCB_CAP(bgep->fm_capabilities))
                        ddi_fm_handler_register(bgep->devinfo,
                        bge_fm_error_cb, (void*) bgep);
        } else {
                /*
                 * These fields have to be cleared of FMA if there are no
                 * FMA capabilities at runtime.
                 */
                bge_reg_accattr.devacc_attr_access = DDI_DEFAULT_ACC;
                dma_attr.dma_attr_flags = 0;
        }
}
```

障害管理リソースのクリーンアップ

ddi_fm_fini(9F) 関数は、*dip* の障害管理をサポートするために割り当てられたリソースをクリーンアップします。

```
void ddi_fm_fini(dev_info_t *dip)
```

ddi_fm_fini() 関数は、カーネルコンテキストからドライバの attach(9E) または detach(9E) エントリポイントで呼び出すことができます。

bge ドライバからの次の例では bge_fm_fini() 関数が示されていますが、この関数から ddi_fm_fini(9F) 関数が呼び出されています。bge_fm_fini() 関数は bge_unattach() 関数内で呼び出され、この関数は bge_attach() および bge_detach() の両関数内で呼び出されます。

```
static void
bge_fm_fini(bge_t *bgep)
{
        /* Only unregister FMA capabilities if we registered some */
        if (bgep->fm_capabilities) {
                /*
                 * Release any resources allocated by pci_ereport_setup()
                 */
                if (DDI_FM_EREPORT_CAP(bgep->fm_capabilities) ||
                    DDI_FM_ERRCB_CAP(bgep->fm_capabilities))
                        pci_ereport_teardown(bgep->devinfo);
                /*
                 * Un-register error callback if error callback capable
                 */
                if (DDI_FM_ERRCB_CAP(bgep->fm_capabilities))
                        ddi_fm_handler_unregister(bgep->devinfo);
                /*
                 * Unregister from IO Fault Services
                 */
                ddi_fm_fini(bgep->devinfo);
        }
}
```

障害管理機能ビットマスクの取得

ddi_fm_capable(9F) 関数は、*dip* に現在設定されている機能ビットマスクを返します。

```
void ddi_fm_capable(dev_info_t *dip)
```

エラーの報告

このセクションでは、次のトピックについての情報を提供します。

- 246 ページの「キューへのエラーイベントの送信」では、エラーイベントをキューに入れる方法について説明します。
- 246 ページの「PCI 関連エラーの検出と報告」では、PCI 関連のエラーを報告する方法について説明します。
- 247 ページの「標準入出力コントローラのエラーの報告」では、標準入出力コントローラのエラーを報告する方法について説明します。
- 249 ページの「サービス影響関数」では、デバイスによって提供されるサービスにエラーが影響を及ぼしたかどうかを報告する方法について説明します。

キューへのエラーイベントの送信

ddi_fm_ereport_post(9F) 関数は、障害管理デーモン fmd(1M) に配信する ereport イベントをキューに入れます。

```
void ddi_fm_ereport_post(dev_info_t *dip,
                         const char *error_class,
                         uint64_t ena,
                         int sflag, ...)
```

sflag パラメータは、システムメモリーリソースおよびイベントチャネルリソースが利用可能になるまで呼び出し側が待機するかどうかを示します。

ENA はこのエラーレポートに対するエラー数値関連付け (Error Numeric Association) を意味します。ENA は、バスネクサスドライバのような別のエラー検出ソフトウェアモジュールから初期化および取得されている可能性があります。ENA を 0 に設定した場合、ENA は ddi_fm_ereport_post() によって初期化されます。

名前-値ペア (*nvpair*) 変数の引数リストの内容は、配列以外の data_type_t 型の場合は 1 つ以上の名前、型、値ポインタの (*nvpair* 組であり、data_type_t 配列型の場合は 1 つ以上の名前、型、要素数、値ポインタの組です。*nvpair* 組は、診断のために必要な ereport イベントペイロードを構成します。引数リストの終了は NULL によって指定されます。

error_class には、入出力コントローラに関して247 ページの「標準入出力コントローラのエラーの報告」で説明されている ereport クラス名およびペイロードが適宜使用されます。説明にない ereport クラス名およびペイロードを定義できますが、そのような名前およびペイロードは、ドライバ固有の診断エンジンソフトウェアまたは Eversholt フォルトツリー (eft) ルールとともに Sun イベントレジストリに登録する必要があります。

```
void
bge_fm_ereport(bge_t *bgep, char *detail)
{
        uint64_t ena;
        char buf[FM_MAX_CLASS];
        (void) snprintf(buf, FM_MAX_CLASS, "%s.%s", DDI_FM_DEVICE, detail);
        ena = fm_ena_generate(0, FM_ENA_FMT1);
        if (DDI_FM_EREPORT_CAP(bgep->fm_capabilities)) {
                ddi_fm_ereport_post(bgep->devinfo, buf, ena, DDI_NOSLEEP,
                    FM_VERSION, DATA_TYPE_UINT8, FM_EREPORT_VERS0, NULL);
        }
}
```

PCI 関連エラーの検出と報告

pci_ereport_post(9F) を使用する場合、PCI、PCI-X、PCI-E を含む PCI 関連のエラーが自動的に検出および報告されます。

```
void pci_ereport_post(dev_info_t *dip, ddi_fm_error_t *derr, uint16_t *xx_status)
```

PCI ローカルバス構成ステータスレジスタで発生するエラーについては、ドライバ固有の ereport をドライバで生成する必要はありません。pci_ereport_post() 関数は、データパリティーエラー、マスターアボート、ターゲットアボート、シグナル付きシステムエラーなどを報告できます。

ドライバ内で pci_ereport_post() を使用する場合は、ドライバの attach(9E) ルーチンで pci_ereport_setup(9F) が事前に呼び出されている必要があるほか、あとでドライバの detach(9E) ルーチンで pci_ereport_teardown(9F) が呼び出される必要があります。

次に示す bge ドライバのコード例では、ドライバのエラーハンドラから pci_ereport_post() 関数を呼び出しています。254 ページの「エラーハンドラの登録」も参照してください。

```
/*
 * The I/O fault service error handling callback function
 */
/*ARGSUSED*/
static int
bge_fm_error_cb(dev_info_t *dip, ddi_fm_error_t *err, const void *impl_data)
{
     /*
      * as the driver can always deal with an error
      * in any dma or access handle, we can just return
      * the fme_status value.
      */
     pci_ereport_post(dip, err, NULL);
     return (err->fme_status);
}
```

標準入出力コントローラのエラーの報告

入出力コントローラでよく発生するエラーに対応した、デバイス ereport の標準セットが定義されています。これらの ereport は、このセクションで説明するいずれかのエラー症状が検出されるたびに生成されます。

このセクションで説明する ereport は、標準ルールの共通セットを使用して ereport を診断する eft 診断エンジンにディスパッチされます。その他のエラーをデバイスドライバで検出する場合、そのエラーは ereport イベントの形式で、デバイス固有の診断ソフトウェアまたは eft ルールとともに Sun イベントレジストリで定義されている必要があります。

DDI_FM_DEVICE_INVAL_STATE
: ドライバはデバイスが無効な状態であることを検出しました。

  ドライバでは、送信または受信するデータが無効と思われることを検出した時点でエラーを送信するようにしてください。たとえば、bge のコード内の bge_chip_reset() および bge_receive_ring() ルーチンは、無効なデータを検出した時点で ereport.io.device.inval_state エラーを生成します。

```
/*
 * The SEND INDEX registers should be reset to zero by the
 * global chip reset; if they're not, there'll be trouble
 * later on.
 */
sx0 = bge_reg_get32(bgep, NIC_DIAG_SEND_INDEX_REG(0));
if (sx0 != 0) {
    BGE_REPORT((bgep, "SEND INDEX - device didn't RESET"));
    bge_fm_ereport(bgep, DDI_FM_DEVICE_INVAL_STATE);
    return (DDI_FAILURE);
}
/* ... */
/*
 * Sync (all) the receive ring descriptors
 * before accepting the packets they describe
 */
DMA_SYNC(rrp->desc, DDI_DMA_SYNC_FORKERNEL);
if (*rrp->prod_index_p >= rrp->desc.nslots) {
    bgep->bge_chip_state = BGE_CHIP_ERROR;
    bge_fm_ereport(bgep, DDI_FM_DEVICE_INVAL_STATE);
    return (NULL);
}
```

DDI_FM_DEVICE_INTERN_CORR
: デバイスは自己訂正内部エラーを報告しました。たとえば、デバイスの内部バッファーで、訂正可能な ECC エラーがハードウェアによって検出されました。

  bge ドライバではこのエラーフラグは使用されていません。

DDI_FM_DEVICE_INTERN_UNCORR
: デバイスは訂正不能な内部エラーを報告しました。たとえば、デバイスの内部バッファーで、訂正不能な ECC エラーがハードウェアによって検出されました。

  bge ドライバではこのエラーフラグは使用されていません。

DDI_FM_DEVICE_STALL
: ドライバは、データ転送が予期せずストールしたことを検出しました。

  bge_factotum_stall_check() ルーチンは、ストール検出の例を示しています。

```
dogval = bge_atomic_shl32(&bgep->watchdog, 1);
if (dogval < bge_watchdog_count)
    return (B_FALSE);

BGE_REPORT((bgep, "Tx stall detected,
watchdog code 0x%x", dogval));
bge_fm_ereport(bgep, DDI_FM_DEVICE_STALL);
return (B_TRUE);
```

DDI_FM_DEVICE_NO_RESPONSE
: デバイスはドライバのコマンドに応答していません。

```
bge_chip_poll_engine(bge_t *bgep, bge_regno_t regno,
        uint32_t mask, uint32_t val)
{
        uint32_t regval;
```

```
        uint32_t n;

        for (n = 200; n; --n) {
                regval = bge_reg_get32(bgep, regno);
                if ((regval & mask) == val)
                        return (B_TRUE);
                drv_usecwait(100);
        }
        bge_fm_ereport(bgep, DDI_FM_DEVICE_NO_RESPONSE);
        return (B_FALSE);
}
```

DDI_FM_DEVICE_BADINT_LIMIT
デバイスが連続で発生させた無効な割り込みの数が多すぎます。

bge ドライバの bge_intr() ルーチンは、問題のある割り込みを検出する例を示しています。bge_fm_ereport() 関数は ddi_fm_ereport_post(9F) 関数のラッパーです。246 ページの「キューへのエラーイベントの送信」の bge_fm_ereport() の例を参照してください。

```
if (bgep->missed_dmas >= bge_dma_miss_limit) {
    /*
     * If this happens multiple times in a row,
     * it means DMA is just not working.  Maybe
     * the chip has failed, or maybe there's a
     * problem on the PCI bus or in the host-PCI
     * bridge (Tomatillo).
     *
     * At all events, we want to stop further
     * interrupts and let the recovery code take
     * over to see whether anything can be done
     * about it ...
     */
    bge_fm_ereport(bgep,
        DDI_FM_DEVICE_BADINT_LIMIT);
    goto chip_stop;
}
```

サービス影響関数

障害管理機能を備えたドライバは、デバイスによって提供されるサービスにエラーが影響を及ぼしたかどうかを示す必要があります。エラーの検出 (および、必要に応じたサービスのシャットダウン) 後に、ドライバが ddi_fm_service_impact(9F) ルーチンを呼び出して、デバイスインスタンスの現在のサービス状態を反映するようにしてください。診断および回復ソフトウェアでは、サービス状態を利用することにより、問題の識別または問題への対応が容易になります。

ドライバ自体がエラーを検出したときも、フレームワークがエラーを検出してアクセスハンドルまたは DMA ハンドルに障害のマークを付けたときも、ddi_fm_service_impact() ルーチンが呼び出されるようにしてください。

```
void ddi_fm_service_impact(dev_info_t *dip, int svc_impact)
```

ddi_fm_service_impact() に渡すことのできるサービス影響値 (*svc_impact*) は次のとおりです。

DDI_SERVICE_LOST	デバイスによって提供されるサービスが、デバイスの障害またはソフトウェアの不具合のために利用できません。
DDI_SERVICE_DEGRADED	ドライバは通常のサービスを提供できませんが、部分的または縮退レベルのサービスを提供できます。たとえば、ドライバは目的の操作が成功するまで操作を繰り返し試行する必要があるか、または構成された速度よりも低速に動作しています。
DDI_SERVICE_UNAFFECTED	ドライバはエラーを検出しましたが、デバイスインスタンスによって提供されるサービスは影響を受けません。
DDI_SERVICE_RESTORED	デバイスのすべてのサービスは復元されました。

ddi_fm_service_impact() を呼び出すと、サービス影響ルーチンに渡されたサービス影響引数に基づいて、ドライバの代わりに次の ereport が生成されます。

- ereport.io.service.lost
- ereport.io.service.degraded
- ereport.io.service.unaffected
- ereport.io.service.restored

次に示す bge のコードでは、ドライバはエラーの結果としてパケットの送受信を正常に再開できないと判断します。デバイスのサービス状態は DDI_SERVICE_LOST に遷移します。

```
/*
 * All OK, reinitialize hardware and kick off GLD scheduling
 */
mutex_enter(bgep->genlock);
if (bge_restart(bgep, B_TRUE) != DDI_SUCCESS) {
    (void) bge_check_acc_handle(bgep, bgep->cfg_handle);
    (void) bge_check_acc_handle(bgep, bgep->io_handle);
    ddi_fm_service_impact(bgep->devinfo, DDI_SERVICE_LOST);
    mutex_exit(bgep->genlock);
    return (DDI_FAILURE);
}
```

注 – 登録済みのコールバックルーチンから ddi_fm_service_impact() 関数を呼び出さないでください。

アクセス属性構造体

DDI_FM_ACCCHK_CAPABLE デバイスドライバは、レジスタの読み取りまたは書き込み中に発生するプログラム式入出力 (PIO) アクセスエラーを処理できることを示すように、そのアクセス属性を設定する必要があります。ddi_device_acc_attr(9S) 構造体の devacc_attr_access フィールドは、ドライバがデータパスエラーのチェックと処理を実行できることをシステムに知らせるために設定します。ddi_device_acc_attr 構造体には次のメンバーがあります。

```
ushort_t devacc_attr_version;
uchar_t devacc_attr_endian_flags;
uchar_t devacc_attr_dataorder;
uchar_t devacc_attr_access;             /* access error protection */
```

デバイスとの間のデータパスで検出されるエラーは、デバイスドライバの親ネクサスのうちの 1 つ以上によって処理できます。

devacc_attr_version フィールドは DDI_DEVICE_ATTR_V1 以上に設定する必要があります。devacc_attr_version フィールドが DDI_DEVICE_ATTR_V1 以上に設定されていない場合、devacc_attr_access フィールドは無視されます。

devacc_attr_access フィールドには次の値を設定できます。

DDI_DEFAULT_ACC
: このフラグは、エラーが発生したときにシステムがデフォルトのアクション (必要であればパニック状態に移行) を実行することを示します。この属性は DDI_FM_ACCCHK_CAPABLE ドライバでは使用できません。

DDI_FLAGERR_ACC
: このフラグは、アクセスハンドルと関連付けられたエラーの処理およびエラーからの回復をシステムが試みることを示します。ドライバでは、260 ページの「Oracle Solaris デバイスドライバの防御的プログラミング手法」で説明されている手法を使用し、呼び出し側アプリケーションにデータを返却することをドライバで許可する前に ddi_fm_acc_err_get(9F) を使用して定期的にエラーをチェックするようにしてください。

  DDI_FLAGERR_ACC フラグを指定すると、次のことが可能になります。

  - ドライバコールバックを介したエラー通知
  - ddi_fm_acc_err_get(9F) によって監視可能なエラー条件

DDI_CAUTIOUS_ACC
: DDI_CAUTIOUS_ACC フラグを指定すると、ドライバによって行われる毎回のプログラム式入出力アクセスの保護レベルが高まります。

注 – このフラグを使用すると、ドライバのパフォーマンスが多大な影響を受けます。

DDI_CAUTIOUS_ACC フラグは、アクセスする側のドライバがエラーを予期していることを示します。システムはこのハンドルと関連付けられたエラーを、できるかぎり正常に処理してエラーから回復することを試みます。結果としてエラーレポートは生成されませんが、ハンドルの `fme_status` フラグは DDI_FM_NONFATAL に設定されます。このフラグは機能的には `ddi_peek(9F)` および `ddi_poke(9F)` と同等です。

DDI_CAUTIOUS_ACC を使用すると、次のことが可能になります。

- バスへの排他的アクセス
- オントラップ保護 - (`ddi_peek()` および `ddi_poke()`)
- `ddi_fm_handler_register(9F)` によって登録されるドライバコールバックを介したエラー通知
- `ddi_fm_acc_err_get(9F)` によって監視可能なエラー条件

ドライバでは通常、データの一貫性を保証するため、また入出力ソフトウェアスタックで適正なエラーステータスが提示されることを保証するために、コードパス内の適切な分岐点でデータパスエラーをチェックするようにしてください。

DDI_FM_ACCCHK_CAPABLE デバイスドライバでは、その `devacc_attr_access` フィールドを DDI_FLAGERR_ACC または DDI_CAUTIOUS_ACC に設定する必要があります。

DMA 属性構造体

アクセスハンドル設定と同様に、DDI_FM_DMACHK_CAPABLE デバイスドライバはその `ddi_dma_attr(9S)` 構造体の `dma_attr_flag` フィールドを DDI_DMA_FLAGERR フラグに設定する必要があります。システムは、DDI_DMA_FLAGERR が設定されたハンドルと関連付けられているエラーからの回復を試みます。`ddi_dma_attr` 構造体には次のメンバーが含まれています。

```
uint_t          dma_attr_version;       /* version number */
uint64_t        dma_attr_addr_lo;       /* low DMA address range */
uint64_t        dma_attr_addr_hi;       /* high DMA address range */
uint64_t        dma_attr_count_max;     /* DMA counter register */
uint64_t        dma_attr_align;         /* DMA address alignment */
uint_t          dma_attr_burstsizes;    /* DMA burstsizes */
uint32_t        dma_attr_minxfer;       /* min effective DMA size */
```

```
uint64_t        dma_attr_maxxfer;       /* max DMA xfer size */
uint64_t        dma_attr_seg;           /* segment boundary */
int             dma_attr_sgllen;        /* s/g length */
uint32_t        dma_attr_granular;      /* granularity of device */
uint_t          dma_attr_flags;         /* Bus specific DMA flags */
```

DDI_DMA_FLAGERR フラグを設定するドライバでは、260 ページの「Oracle Solaris デバイスドライバの防御的プログラミング手法」で説明されている手法を使用し、DMA トランザクションが完了するたびに、またはコードパス内の重要なポイントで ddi_fm_dma_err_get(9F) を使用してデータパスエラーをチェックするようにしてください。これにより、一貫したデータと適正なエラーステータスが入出力ソフトウェアスタックに提示されるようになります。

DDI_DMA_FLAGERR を使用すると、次のことが可能になります。

- ddi_fm_handler_register() によって登録されるドライバコールバックを介したエラー通知
- ddi_fm_dma_err_get() の呼び出しによるエラー条件の監視

エラーステータスの取得

障害が発生し、ハンドルによって対応づけられたリソースにその影響が及ぶ場合、バスによる、または入出力データパス上のその他のデバイスドライバによるエラー処理中に取り込まれたエラー情報を反映して、エラーステータス構造体が

```
void ddi_fm_dma_err_get(ddi_dma_handle_t handle, ddi_fm_error_t *de, int version)
```

```
void ddi_fm_acc_err_get(ddi_acc_handle_t handle, ddi_fm_error_t *de, int version)
```

ddi_fm_dma_err_get(9F) 関数は DMA のエラーステータスを返し、ddi_fm_acc_err_get(9F) 関数はアクセスハンドルのエラーステータスを返します。version フィールドは DDI_FME_VERSION に設定してください。

アクセスハンドルのエラーは、そのアクセスハンドルを使用して行われる、デバイスとの間の PIO トランザクションに影響するエラーが検出されたことを意味します。最近の ddi_get8(9F) の呼び出しなどを介してドライバが受信したデータが部分的に破壊されているとみなすようにしてください。また、最近の ddi_put32(9F) の呼び出しなどを介してデバイスに送信されたデータも、破壊されているか、またはまったく受信されていない可能性があります。ただし、原因となっている障害は一時的である可能性があるため、ドライバでは ddi_fm_acc_err_clear(9F) を呼び出し、デバイスをリセットして既知の状態に戻し、失敗した可能性があるすべてのトランザクションを再試行することによって回復を試みることができます。

DMA ハンドルのエラーが示された場合、そのハンドルに現在バインドされている (または、その時点でバインドされていない場合は最近バインドされた) メモリーとデバイスの間の DMA トランザクションに影響するエラーが検出されたことを意味します。考えられる原因には、DMA データパス上のコンポーネントの障害や、デバイ

スが無効な DMA アクセスを試みたことなどがあります。ドライバでは、再試行してメモリーを再割り当てすることによって処理を継続できる可能性があります。ハンドルに現在バインドされている (または以前にバインドされていた) メモリーの内容を不定とみなし、そのメモリーを解放してシステムに返却するようにしてください。現在のトランザクションと関連付けられている障害の指示は、ハンドルがバインドまたは再バインドされた時点で失われますが、障害は持続する可能性があり、以後の DMA 操作も成功しない可能性があります。

エラーのクリア

ハンドルによってエラーが検出されたあとに、ドライバで `ddi_fm_acc_err_clear()` および `ddi_fm_dma_err_clear(9F)` ルーチンを呼び出すと、ハンドルを解放して再割り当てしなくても要求を再試行できます。

```
void ddi_fm_acc_err_clear(ddi_acc_handle_t handle, int version)

void ddi_fm_dma_err_clear(ddi_dma_handle_t handle, int version)
```

エラーハンドラの登録

エラー処理アクティビティーは、トラップまたはエラー割り込みを介してオペレーティングシステムがエラーを検出した時点で開始される可能性があります。エラー処理を担うソフトウェア (エラーハンドラ) が、失敗した入出力操作に関与していたデバイスをただちに切り離すことができない場合、そのソフトウェアは、エラーの分離を実行できるソフトウェアモジュールをデバイスツリーから探すことを試みる必要があります。Oracle Solaris のデバイスツリーは、ネクサスドライバのエラー処理アクティビティーを下位ノードに伝播するための構造的手段を提供します。この下位ノードがエラーをより詳しく把握し、エラー状態を取り込んで問題のデバイスを切り離すことができる可能性があります。

ドライバでは、エラーハンドラコールバックを入出力障害サービスフレームワークに登録できます。エラーハンドラは、エラーの種類ごとに、またエラー検出が発生したサブシステムごとに固有のものとしてください。ドライバのエラーハンドラルーチンが呼び出されたとき、ドライバではデバイストランザクションと関連付けられた未処理のエラーをすべてチェックして、ereport イベントを生成する必要があります。ドライバはその `ddi_fm_error(9S)` 構造体でエラーハンドラのステータスを返す必要もあります。たとえば、システムの整合性が損なわれていると判断された場合、エラーハンドラによってシステムをパニック状態にすることがもっとも適切なアクションである可能性があります。

コールバックは、エラーが特定のデバイスインスタンスと関連付けられている場合に親ネクサスドライバによって呼び出されます。エラーハンドラを登録するデバイスドライバは DDI_FM_ERRCB_CAPABLE である必要があります。

```
void ddi_fm_handler_register(dev_info_t *dip, ddi_err_func_t handler, void *impl_data)
```

ddi_fm_handler_register(9F) ルーチンは、エラーハンドラコールバックを入出力障害サービスフレームワークに登録します。コールバック登録のための ddi_fm_handler_register() 関数は、ドライバの障害管理機能の初期化 (ddi_fm_init()) よりもあとに、ドライバの attach(9E) エントリポイントで呼び出されるようにしてください。

エラーハンドラのコールバック関数は次の処理を実行する必要があります。

- デバイストランザクションと関連付けられている未処理のハードウェアエラーをすべてチェックし、診断のための ereport イベントを生成します。PCI、PCI-x、または PCI Express デバイスの場合、これは通常、246 ページの「PCI 関連エラーの検出と報告」で説明されているように pci_ereport_post() を使用すると実行できます。
- 次に示すエラーハンドラのステータスを、その ddi_fm_error 構造体で返します。
  - DDI_FM_OK
  - DDI_FM_FATAL
  - DDI_FM_NONFATAL
  - DDI_FM_UNKNOWN

ドライバのエラーハンドラは次の情報を受け取ります。

- ドライバの制御下にあるデバイスインスタンスへのポインタ (*dip*)
- エラー処理のための共通の障害管理データおよびステータスを格納したデータ構造体 (ddi_fm_error)
- ハンドラの登録時に指定された実装固有データへのポインタ (*impl_data*)

ddi_fm_handler_register() および ddi_fm_handler_unregister(9F) ルーチンは、カーネルコンテキストから、ドライバの attach(9E) または detach(9E) エントリポイントで呼び出す必要があります。登録済みのエラーハンドルコールバックは、カーネル、割り込み、または高レベル割り込みの各コンテキストから呼び出すことができます。そのため、エラーハンドラには次の条件があります。

- ロックを保持してはならない
- 休眠状態でリソースを待機してはならない

デバイスドライバは次の処理を担います。

- エラーを引き起こした可能性があるデバイスインスタンスの切り離し
- エラーと関連付けられたトランザクションの回復
- エラーがサービスに及ぼす影響の報告
- エラーが致命的とみなされた場合の、デバイスのシャットダウンのスケジューリング

これらのアクションはエラーハンドラ関数の内部で実行できます。ただし、ロックに関する制限のため、さらには障害が発生した時点でドライバが実行していた処理

のコンテキストをエラーハンドラ関数が必ずしも認識しているとはかぎらないため、前に説明したドライバの通常のパス内で ddi_fm_acc_err_get(9F) と ddi_fm_dma_err_get(9F) のインライン呼び出しのあとにこれらのアクションを実行する方がより一般的です。

```
/*
 * The I/O fault service error handling callback function
 */
/*ARGSUSED*/
static int
bge_fm_error_cb(dev_info_t *dip, ddi_fm_error_t *err, const void *impl_data)
{
     /*
      * as the driver can always deal with an error
      * in any dma or access handle, we can just return
      * the fme_status value.
      */
     pci_ereport_post(dip, err, NULL);
     return (err->fme_status);
}
```

障害管理のデータおよびステータスの構造体

ドライバのエラー処理コールバックには、エラー処理のための共通の障害管理データおよびステータスが格納されたデータ構造へのポインタが渡されます。

データ構造 ddi_fm_error に格納されるのは、FMA プロトコルによる現在のエラーの ENA、エラーハンドラコールバックのステータス、エラー予測フラグ、および、親ネクサスによって検出されたエラーと関連付けられているすべてのアクセスハンドルまたは DMA ハンドルです。

fme_ena	このフィールドは呼び出し側の親ネクサスによって初期化され、ドライバの登録済みコールバックルーチンに到達する前に、エラー処理伝播チェーンの途中で増分されている可能性があります。ドライバが独自に関連エラーを検出した場合、ddi_fm_ereport_post() を呼び出す前にドライバでこの ENA を増分してください。
fme_acc_handle、fme_dma_handle	親のレベルで検出されたエラーを、デバイスドライバによって対応づけまたはバインドされたハンドルに親が関連付けることができた場合、これらのフィールドには有効なアクセスハンドルまたは DMA ハンドルが格納されています。
fme_flag	DDI_CAUTIOUS_ACC 保護操作の結果としてエラーが発生したと呼び出し側の親が判断した場合、fme_flag フラグは

DDI_FM_ERR_EXPECTED に設定されます。この場合、`fme_acc_handle` は有効であり、ドライバでは DDI_CAUTIOUS_ACC 保護操作と関連付けられていないエラーのみをチェックして報告するようにしてください。それ以外の場合、`fme_flag` は DDI_FM_ERR_UNEXPECTED に設定され、ドライバはすべての範囲のエラー処理タスクを実行する必要があります。

`fme_status`

そのエラーハンドルコールバックから戻るときに、ドライバは `fme_status` を次のいずれかの値に設定する必要があります。

- DDI_FM_OK – エラーは検出されませんでした。このデバイスインスタンスの動作状態は同じままです。
- DDI_FM_FATAL – エラーが発生しました。ドライバはそのエラーがシステムにとって致命的であると認識しています。たとえば、`pci_ereport_post`(9F) の呼び出しによって、システムの致命的エラーが検出された可能性があります。この場合、ドライバがドライバのコンテキストで保持している補足的なエラー情報をすべて報告するようにしてください。
- DDI_FM_NONFATAL – ドライバによってエラーが検出されましたが、エラーはシステムにとって致命的であるとは認識されていません。ドライバはエラーを識別し、エラーを分離したか、または今後エラーを分離することを確定しています。
- DDI_FM_UNKNOWN – エラーが検出されましたが、ドライバはデバイスを切り離すことができないか、またはエラーがシステムの稼働状態に及ぼす影響を特定できません。

障害の診断

障害管理デーモン `fmd`(1M) は、診断エンジン (DE) プラグインモジュールの開発のためのプログラミングインタフェースを提供します。任意または特定のエラーリ

モート測定を使用して診断するように DE を記述できます。eft DE は、Eversholt 言語で指定された診断ルールに基づいて、多数の ereport クラスを診断するように設計されました。

標準リーフデバイス診断

ほとんどの入出力サブシステムは eft DE およびルールセットを使用して、デバイスおよびデバイスドライバ関連の問題を診断します。PCI リーフデバイス向けとしては、247 ページの「標準入出力コントローラのエラーの報告」の一覧に示した標準 ereport のセットが定義されています。これらの ereport に付随する eft 診断ルールは、リモート測定を使用し、関連付けられたデバイス障害を識別します。これらの ereport を生成するドライバでは、追加の診断ソフトウェアまたは eft ルールを配布する必要は一切ありません。

これらの ereport の検出および生成により、次のフォルトイベントが生成されます。

`fault.io.pci.bus-linkerr`	PCI バス上のハードウェア障害
`fault.io.pci.device-interr`	デバイス内部のハードウェア障害
`fault.io.pci.device-invreq`	デバイスが無効な要求を送信する原因である、デバイスのハードウェア障害またはドライバの不具合
`fault.io.pci.device-noresp`	ドライバが有効な要求に応答しない原因である、デバイスのハードウェア障害
`fault.io.pciex.bus-linkerr`	リンク上のハードウェア障害
`fault.io.pciex.bus-noresp`	リンクがダウンしているためデバイスが有効な要求に応答できない
`fault.io.pciex.device-interr`	デバイス内部のハードウェア障害
`fault.io.pciex.device-invreq`	デバイスが無効な要求を送信する原因である、デバイスのハードウェア障害またはドライバの不具合
`fault.io.pciex.device-noresp`	有効な要求に応答しない原因である、デバイスのハードウェア障害

特殊なデバイス診断

追加の ereport を生成したり、より特殊な診断ソフトウェアまたは eft ルールを提供したりする必要があるドライバ開発者は、C ベースの DE または eft 診断ルールセットを記述することによってこれを行えます。

イベントレジストリ

Sun イベントレジストリは、すべてのクラス名、ereport、障害、不具合、アップセット、および疑いリスト (list.suspect) イベントの集中リポジトリです。イベントレジストリには、すべてのイベントメンバーペイロードの現在の定義に加えて、内部ドキュメント、疑いリスト、辞書、ナレッジ記事など、ペイロード関連以外の重要情報も格納されます。たとえば、`ereport.io` と `fault.io` は、入出力ドライバの開発者にとって特に重要な意味を持つ基底クラス名の中の 2 つです。

FMA イベントプロトコルは、登録される個々のイベントに付随するペイロードメンバーの基本セットを定義します。開発者は、診断エンジン (または eft ルール) が疑いリストを特定の障害に絞り込むために役立つ追加のイベントを定義することもできます。

用語

このセクションでは次の用語を使用します。

エージェント	fault.* または list.* イベントを購読する障害管理モジュールを指して使用される一般的な用語です。エージェントは、障害を起こしたリソースをリタイアさせたり、診断結果を管理者に通知したり、より上位の管理フレームワークに処理を引き継いだりするために使用されます。
ASRU (自動システム再構成ユニット)	ASRU は、問題をシステムから切り離して、以降のエラーレポートを抑制するために、ソフトウェアまたはハードウェアによって無効化できるリソースです。
DE (診断エンジン)	障害管理モジュールの 1 つで、その目的は、1 つ以上の着信エラーイベントのクラスを登録することによって問題を診断し、これらのイベントを使って、システム上の各問題と関連付けられたケースを解決することです。
ENA (エラー数値関連付け)	エラー数値関連付け (ENA) は、特定の障害領域および期間の範囲内でエラーレポートを一意に識別するエンコード整数です。また ENA は、以前のエラーに対する、副次的影響としてのエラーの関係性も示します。
エラー	予期しない状況、結果、シグナル、またはデータ。エラーはシステム上の問題の兆候です。1 つ 1 つの問題から、多くの異なる種類のエラーが発生するのが一般的です。
ereport (エラーレポート)	特定のエラーに関して取り込まれるデータ。エラーレポートの形式は事前に定義されます。これは、エラーレポートに名前を付けるクラスを作成し、Sun イベントレジストリを使用してスキーマを定義することによって行います。
ereport イベント (エラーイベント)	エラーレポートのインスタンスを表現するデータ構造。エラーイベントは名前-値ペアのリストとして表されます。
障害	ハードウェアコンポーネントの異常な動作。

障害境界	特定の障害が列挙される、ハードウェアまたはソフトウェア要素の論理区分。
フォルトイベント	プロトコルでエンコードされた障害診断のインスタンス。
Fault Manager	1 つ以上の診断エンジンと状態管理による障害診断の役割を担うソフトウェアコンポーネント。
FMRI (障害管理対象リソース識別子)	FMRI は URL のような識別子であり、障害管理システム内の特定リソースの正規名として機能します。個々の FMRI には、リソースの種類を識別するスキームと、そのスキームに固有の 1 つ以上の値が含まれています。FMRI は URL のような文字列として、または名前-値ペアのリストからなるデータ構造として表現できます。
FRU (現場交換可能ユニット)	FRU は、顧客またはサービスプロバイダが現場で交換できるリソースです。FRU はハードウェア (例: システムボード) またはソフトウェア (例: ソフトウェアパッケージ、パッチ) に対して定義できます。

Oracle Solaris デバイスドライバの防御的プログラミング手法

このセクションでは、デバイスドライバにおいて、システムのパニックやハングアップ、システムリソースの浪費、データ破壊の拡散を回避するための手法について説明します。エラー処理と診断のための入出力障害サービスフレームワークに加えて、ここで説明する防御的プログラミング手法をドライバで使用すると、そのドライバは強化されていると認識されます。

すべての Oracle Solaris ドライバで、次のコーディング手法を実践するようにしてください。

- ハードウェアの各部品が、デバイスドライバの別個のインスタンスによって制御されるようにします。104 ページの「デバイス構成の概念」を参照してください。
- プログラム式入出力 (PIO) は、DDI アクセス関数を介し、適切なデータアクセスハンドルを使用する方法でのみ実行される必要があります。第 7 章「デバイスアクセス: プログラム式入出力」を参照してください。
- デバイスドライバは、デバイスから受信するデータが破壊されている可能性を想定する必要があります。データを使用する前に、ドライバでデータの整合性をチェックする必要があります。
- ドライバは不正なデータがシステムのほかの部分に流されないようにする必要があります。
- ドライバでは、ドキュメント化された DDI 関数およびインタフェースのみを使用します。

- ドライバによって全面的に制御される DMA バッファー (DDI_DMA_READ) 内のメモリーページのみにデバイスが書き込みを行うことをドライバで保証する必要があります。これには、DMA の障害によってシステムのメインメモリーの不特定箇所が破壊されることを防ぐ意味があります。
- デバイスが動作停止した場合に、デバイスドライバがシステムリソースを際限なく浪費してはなりません。デバイスから継続的にビジー状態の応答がある場合は、ドライバをタイムアウトします。またドライバでは、正常でない (問題のある) 割り込み要求を検出して適切なアクションを実行します。
- デバイスドライバは、Oracle Solaris OS のホットプラグをサポートする必要があります。
- デバイスドライバは、リソースを待機する代わりにコールバックを使用する必要があります。
- ドライバは障害のあとにリソースを解放する必要があります。たとえば、ハードウェアで障害が発生したあとでも、システムがすべてのマイナーデバイスを閉じてドライバインスタンスを切り離せるようにする必要があります。

別個のデバイスドライバインスタンスの使用

Oracle Solaris カーネルでは、ドライバの複数のインスタンスが許可されます。各インスタンスは個別のデータ領域を持ちますが、テキストや一部のグローバルデータをほかのインスタンスと共有します。デバイスはインスタンス単位で管理されます。ドライバでは、フェイルオーバーを内部的に処理するように設計されている場合を除いて、ハードウェアの部品ごとに別個のインスタンスを使用するようにしてください。たとえば、複数の機能を備えるカードの使用時に、1 つのスロットに付き 1 つのドライバの複数のインスタンスが発生する可能性があります。

DDI アクセスハンドルの排他的使用

ドライバによるすべての PIO アクセスでは、次のルーチンファミリの Oracle Solaris DDI アクセス関数を使用する必要があります。

- ddi_get*X*
- ddi_put*X*
- ddi_rep_get*X*
- ddi_rep_put*X*

ドライバでは、マップされたレジスタに、ddi_regs_map_setup(9F) から返されたアドレスで直接アクセスしてはいけません。ddi_peek(9F) および ddi_poke(9F) ルーチンはアクセスハンドルを使用しないため、これらのルーチンを使わないようにします。

DDI アクセスメカニズムが重要な理由は、DDI アクセスの利用により、カーネルへのデータ読み込みの形式を制御できるようになるためです。

破壊されたデータの検出

次のセクションでは、データ破壊が発生する可能性がある場所と、破壊を検出する方法について説明します。

デバイス管理データおよび制御データの破壊

ドライバでは、PIO によるか DMA によるかを問わず、デバイスから取得するすべてのデータがすでに破壊されている可能性があると想定するようにしてください。特に、デバイスからのデータに基づくポインタ、メモリーオフセット、および配列インデックスについては細心の注意を払う必要があります。そのような値は悪質である、つまり、間接参照された場合にカーネルパニックを引き起こす可能性があります。そのようなすべての値について、使用する前に範囲および配列 (必要な場合) をチェックしてください。

悪質でないポインタであっても、誤動作の原因となる可能性があります。たとえば、有効だが正しくないオブジェクトのインスタンスをポインタが指し示す可能性があります。ドライバでは可能なかぎり、ポインタとその指示先のオブジェクトをクロスチェックするか、それが難しい場合はそのポインタを介して取得したデータを検証するようにしてください。

パケット長、ステータス語、チャネル ID など、その他の種類のデータも誤動作の原因となる可能性があります。これらの種類のデータを可能な範囲内でチェックするようにしてください。パケット長については、範囲チェックを実行することにより、長さが負ではないこと、格納先バッファーの長さを超えてもいないことを保証できます。ステータス語については「不可能」ビットのチェックを実行できます。チャネル ID については、有効な ID のリストとの照合を実行できます。

値を使用してストリームを識別する箇所で、ドライバはストリームがまだ存在していることを保証する必要があります。STREAMS 処理の非同期的な性質は、ストリームが分解可能な一方で、デバイス割り込みが未処理であることを意味します。

ドライバでデバイスからデータを再読み取りしないでください。データは 1 回だけ読み取られ、検証され、ドライバのローカル状態に保存されるようにしてください。これにより、データを最初に読み取ったときは正確だが、あとで再読み取りしたときにデータが誤っているという危険性を回避できます。

ドライバでは、すべてのループの境界が確定していることも確認してください。たとえば、継続的な BUSY ステータスを返すデバイスによって、システム全体が動作停止されないようにする必要があります。

受信データの破壊

デバイスエラーの結果、破壊されたデータが受信バッファーに配置される可能性があります。そのような破壊は、デバイスの領域を超えて (たとえば、ネットワークの内部で) 発生する破壊と区別することができません。既存のソフトウェアは通常、そ

のような破壊を処理するしくみをすでに備えています。1 つの例は、プロトコルスタックのトランスポート層における整合性チェックです。別の例は、デバイスを使用するアプリケーション内部での整合性チェックです。

上位層で受信データの整合性がチェックされない場合、ドライバ自体の内部でデータの整合性をチェックできます。受信データの破壊を検出する方法は通常、デバイスごとに異なります。実行できるチェックの種類の例としては、チェックサムや CRC があります。

DMA 遮断

障害のあるデバイスは、バス上で不適切な DMA 転送を開始する可能性があります。このデータ転送によって、以前に配信された正常なデータが破壊されてしまう可能性があります。障害のあるデバイスは、そのデバイスのドライバに属さないメモリーにまで悪影響を及ぼすような、破壊されたアドレスを生成する可能性があります。

IOMMU を備えるシステムでは、デバイスは DMA 用に書き込み可能としてマップされたページに限って書き込むことができます。したがって、そのようなページは 1 つのドライバインスタンスが単独で所有するようにしてください。これらのページは、ほかのどのカーネル構造とも共有しないでください。該当するページが DMA 用に書き込み可能としてマップされている場合でも、ドライバではそのページ内のデータを疑うようにしてください。ページをドライバの外部に渡す前に、またはデータを検証する前に、ページと IOMMU のマッピングを解除する必要があります。

`ddi_umem_alloc(9F)` を使用すると、アライメントされたページ全体が割り当てられることを保証したり、複数のページを割り当てて、最初のページ境界よりも下のメモリーを無視したりできます。`ddi_ptob(9F)` を使用すると、IOMMU ページのサイズを調べることができます。

別の方法として、ドライバでメモリーの安全な部分にデータをコピーしてから、そのデータを処理することもできます。この場合、最初に `ddi_dma_sync(9F)` を使用してデータを同期させる必要があります。

`ddi_dma_sync()` を呼び出すときは、DMA を使用してデータをデバイスに転送する前に `SYNC_FOR_DEV` を指定し、デバイスからメモリーに DMA を使用してデータを転送したあとに `SYNC_FOR_CPU` を指定するようにしてください。

IOMMU を備えた一部の PCI ベースのシステムでは、デバイスは PCI デュアルアドレスサイクル (64 ビットアドレス) を使用すると IOMMU をバイパスできます。この機能により、デバイスでメインメモリーのいずれかの領域が破壊される可能性が生じます。デバイスドライバでは、そのようなモードの使用を試みてはならず、モードを無効にしておくべきです。

問題のある割り込みの処理

ドライバでは問題のある割り込みを識別する必要があります。これは、割り込みが際限なく発生し続けるとシステムのパフォーマンスが著しく低下し、シングルプロセッサーのマシンではほぼ確実にストールしてしまうためです。

ドライバで特定の割り込みを無効と識別することが困難な場合もあります。ネットワークドライバの場合は、受信した割り込みが指示されても、新しいバッファーが利用できなければ作業は不要です。この状況が単独で発生した場合は問題ありません。実際の作業は (読み取りサービスなどの) 別のルーチンによってすでに完了している可能性があるためです。

一方、ドライバが処理する作業を伴わない割り込みが連続した場合は、問題のある割り込みの列を示している可能性があります。そのため、防御手段を講じる前に、プラットフォームが明らかに無効な割り込みを多数発生させてしまうことになります。

処理する作業がありそうなのにハングアップしてしまったデバイスは、対応するバッファー記述子を更新できなかった可能性があります。ドライバでは、このような繰り返しの要求を防御するようにしてください。

場合によっては、プラットフォーム固有のバスドライバの側で、要求に基づかない持続的な割り込みを識別し、障害のあるデバイスを無効化できることがあります。ただしこれは、有効な割り込みを識別して適切な値を返すことができるという、ドライバの能力に依存します。ドライバでは、デバイスが正当な割り込みをかけたことを検出した場合を除き、`DDI_INTR_UNCLAIMED` の結果を返すようにしてください。割り込みが正当であるのは、デバイスが実際に、何らかの有用な処理を行うことをドライバに要求している場合に限られます。

偶発性の高いその他の割り込みの正当性を証明することは、さらに困難です。割り込み想定フラグは、割り込みが有効かどうかを評価するために役立つ手段です。デバイスの記述子すべてがすでに割り当てられている場合に生成できる、記述子なしのような割り込みを例として考えます。ドライバがカードの最後の記述子を使用したことを検出した場合、割り込み想定フラグを設定できます。関連付けられた割り込みが配信されたときにこのフラグが設定されていない場合、その割り込みは疑わしいと判断できます。

メディアが切断されたことやフレーム同期が失われたことを知らせるものなど、情報通知のための割り込みの中には予測できないものがあります。そのような割り込みに問題があるかどうかを検出するもっとも簡単な方法は、最初の発生時にこの特定の送信元を次のポーリングサイクルまでマスクすることです。

無効化されている間にふたたび割り込みが発生した場合、その割り込みを偽とみなすようにします。デバイスによっては、関連付けられた送信元をマスクレジスタが無効にし、割り込みを発生させない場合でも読み取ることのできる、割り込みステータスビットがあります。ドライバの開発者は、デバイスに合わせてより適切なアルゴリズムを工夫できます。

割り込みステータスビットが無限ループに陥らないようにしてください。パスの開始時に設定されたステータスビットがいずれも実際の作業を必要としない場合は、このようなループを切断してください。

プログラミングのその他の考慮事項

これまでのセクションで述べた要件に加えて、次の問題を考慮してください。

- スレッドの対話
- トップダウン要求の脅威
- 適応型戦略

スレッドの対話

デバイスドライバにおけるカーネルパニックの多くは、デバイス障害の発生後の、カーネルスレッドの予期しない対話によって引き起こされます。デバイスで障害が発生すると、開発者が予想しなかった形でスレッドの対話が起きることがあります。

処理ルーチンが早期終了した場合、予期しているシグナルが与えられないことにより、条件変数の待機側がブロックされます。ほかのモジュールに障害を通知しようとしたり、予想外のコールバックを処理しようとしたりすると、望ましくない形でスレッドの対話が発生する可能性があります。デバイス障害の際に発生する可能性がある、mutex の取得と放棄の順序について検討してください。

アップストリームの STREAMS モジュールを起点とするスレッドは、予想に反してそのモジュールをコールバックするために使用された場合、望ましくない矛盾した状況に陥る可能性があります。代替スレッドを使用して例外メッセージを処理することを検討してください。たとえば、プロシージャーでは、読み取り側の putnext(9F) でエラーを直接処理するのではなく、読み取り側のサービスルーチンを使用すると M_ERROR を伝達できます。

障害の発生した STREAMS デバイスが、クローズ時に障害が原因で静止できなかった場合、ストリームが分解されたあとに割り込みが発生する可能性があります。割り込みハンドラは、古いストリームポインタを使用してメッセージを処理しようとしてはなりません。

トップダウン要求の脅威

ドライバの開発者は、ハードウェアの故障からシステムを保護する一方で、ドライバの誤用を防ぐ必要もあります。ドライバは、カーネル基盤は常に正しい (信頼できるコア) ということを前提にできますが、ドライバに渡されるユーザー要求が有害な場合があります。

たとえば、ユーザーが提供したデータブロック (M_IOCTL) に対してアクションを実行することをユーザーが要求し、そのデータブロックがメッセージの制御部で指示されたサイズより小さいという場合があります。ドライバはユーザーアプリケーションを信頼してはなりません。

ドライバが受信できる各タイプの ioctl の構造と、ioctl が引き起こす可能性がある潜在的な損害について検討してください。ドライバでは、不正な形式の ioctl を処理しないようにチェックを実行するようにしてください。

適応型戦略

ドライバは、障害の起きたハードウェアを使用することでサービスの提供を継続できます。デバイスにアクセスするための代替的な戦略を用いることによって、特定された問題への対処を試みることができます。ハードウェアの故障が予測不能であることと、設計の複雑さが増すことのリスクを考慮すれば、適応型戦略が常に賢明とは限りません。この戦略は、定期的な割り込みポーリングや再試行といった範囲に限定するようにしてください。デバイスを定期的に再試行することにより、ドライバはデバイスがいつ回復したかを把握できます。定期的なポーリングを使用すると、割り込みの無効化をドライバが強制されたあとでも、割り込みメカニズムを制御できます。

不可欠のシステムサービスを提供するための代替デバイスをシステムが常に備えていることが理想的です。 カーネルまたはユーザー空間でのサービス多重化は、デバイスで障害が起きたときにシステムサービスを維持するための最良の手段です。ただし、このセクションではそのような方式について扱いません。

ドライバ強化テストハーネス

ドライバ強化テストハーネスは、入出力障害サービスおよび防御的プログラミングの要件が正しく満たされていることをテストします。強化されたデバイスドライバは、潜在的なハードウェア障害に対する耐性を備えています。開発者は、デバイスドライバ開発プロセスの一環として、ドライバの回復性能をテストする必要があります。この種類のテストでは、広範囲の一般的なハードウェア障害を、制御された反復可能な方法でドライバが処理できることを確認します。ドライバ強化テストハーネスを使用すると、開発者はそのようなハードウェア障害をソフトウェアでシミュレートできます。

ドライバ強化テストハーネスは、Oracle Solaris デバイスのドライバ開発ツールです。このテストハーネスは、開発中のドライバがそのハードウェアにアクセスするときに、広範囲の擬似的なハードウェア障害を投入します。このセクションでは、テストハーネスを構成し、エラー投入仕様 (*errdef*) を作成して、デバイスドライバのテストを実行する方法について説明します。

テストハーネスはドライバからの各種 DDI ルーチンの呼び出しを横取りし、あたかもハードウェアが原因であるかのように、呼び出しの結果を破壊します。ハーネスでは、特定のレジスタへのアクセスに対する破壊に加えて、よりランダムな種類の破壊も定義できます。

テストハーネスでは、指定されたワークロードの実行中に、すべてのレジスタアクセスのほか、ダイレクトメモリーアクセス (DMA) と割り込みの使用状況を追跡することにより、テストスクリプトを自動生成できます。生成されたスクリプトでは、そのワークロードが再実行されるのと並行して、アクセスのたびに一連の障害が投入されます。

ドライバのテスト担当者は、生成されたスクリプトから、重複しているテストケースを削除するようにしてください。

テストハーネスは、`bofi` という名前 (bus_ops 障害投入を意味する) のデバイスドライバと、2 つのユーザーレベルユーティリティー `th_define(1M)` および `th_manage(1M)` として実装されています。

テストハーネスは次のタスクを行います。

- Oracle Solaris DDI サービスが仕様に準拠して使用されていることを検証する
- プログラム式入出力 (PIO) および DMA 要求の破壊と、割り込みに対する干渉を管理のもとで行い、ドライバが管理するハードウェアで発生する障害をシミュレートする
- 親ネクサスドライバから報告される、CPU とデバイス間のデータパスにおいて、障害のシミュレーションを行う
- 指定されたワークロードの間、ドライバのアクセスを監視し、障害投入スクリプトを生成する

障害投入

ドライバ強化テストハーネスは、ドライバがハードウェアにアクセスするたびにそれを横取りし、必要に応じてアクセスを破壊します。このセクションでは、ドライバの回復性能をテストする障害を作成するにあたり、理解しておくことが望ましい情報を提供します。

Oracle Solaris のデバイスは、デバイスツリー (devinfo ツリー) と呼ばれるツリー状構造の内部で管理されます。devinfo ツリーの各ノードには、システムに存在するデバイスの特定のインスタンスに関する情報が格納されます。各リーフノードはデバイスドライバに対応し、ほかのすべてのノードはネクサスノードと呼ばれます。通常は、1 つのネクサスが 1 つのバスを表します。バスノードはリーフドライバをバスへの依存性から切り離し、アーキテクチャー的に独立したドライバを生成できるようにします。

多くの DDI 関数、特にデータアクセス関数は、結果としてバスネクサスドライバへの上位呼び出しになります。リーフドライバはハードウェアにアクセスするとき、アクセスルーチンにハンドルを渡します。バスネクサスは、ハンドルを操作して要求に応じる方法を理解しています。DDI 準拠のドライバはこれらの DDI アクセスルーチンを使用してのみ、ハードウェアにアクセスします。テストハーネスは、指定されたバスネクサスにこれらの上位呼び出しが到達する前に横取りします。ドライバのテスト担当者が指定した基準にデータアクセスが一致すれば、アクセスは破壊されます。基準に一致しなかったデータアクセスは、バスネクサスに渡されて通常どおり処理されます。

ドライバは ddi_regs_map_setup(9F) 関数を使用してアクセスハンドルを取得します。

```
ddi_regs_map_setup(dip, rset, ma, offset, size, handle)
```

引数では、対応づけ対象の「オフボード」メモリーを指定します。ハンドルはドライバをバス階層構造の詳細から分離するためのものなので、対応づけされた入出力アドレスをドライバが参照するときは、返されたハンドルを使用する必要があります。したがって、返される対応づけされたアドレスである *ma* を直接使用しないでください。対応づけされたアドレスを直接使用すると、その時点以降、データアクセス関数のメカニズムを使用しないことになります。

プログラム式入出力では、データアクセス関数の組み合わせは次のようになります。

- 入出力からホストへ:

  ```
  ddi_getX(handle, ma)
  ddi_rep_getX(handle, buf, ma, repcnt, flag)
  ```

- ホストから入出力へ:

  ```
  ddi_putX(handle, ma, value)
  ddi_rep_putX()
  ```

X および *repcnt* は転送されるバイト数です。*X* はバス転送サイズ (8、16、32、または 64 バイト) です。

DMA にはこれに類似した、より充実したデータアクセス関数の集合が用意されています。

テストハーネスの設定

ドライバ強化テストハーネスは、Oracle Solaris Developer Cluster に含まれています。この Oracle Solaris クラスタがインストールされていない場合、プラットフォームに適合したテストハーネスパッケージを手作業でインストールする必要があります。

テストハーネスのインストール

テストハーネスパッケージ (SUNWftduu および SUNWftdur) をインストールするには、pkgadd(1M) コマンドを使用します。

スーパーユーザーとして、パッケージが存在するディレクトリに移動し、次のように入力します。

```
# pkgadd -d . SUNWftduu SUNWftdur
```

テストハーネスの構成

テストハーネスをインストールしたら、/kernel/drv/bofi.conf ファイルのプロパティーを構成して、テストするドライバと対話するようにハーネスを構成します。ハーネスの構成が完了したら、システムをリブートしてハーネスドライバをロードします。

テストハーネスの動作は、/kernel/drv/bofi.conf 設定ファイルで設定されるブート時プロパティーによって制御されます。

ハーネスを最初にインストールするときに、次のプロパティーを設定して、テストするドライバへの DDI アクセスをハーネスが横取りするようにします。

bofi-nexus　バスのネクサスタイプ (例: PCI バス)

bofi-to-test　テストするドライバの名前

たとえば、xyznetdrv という名前の PCI バスネットワークドライバをテストするには、プロパティー値を次のように設定します。

```
bofi-nexus="pci"
bofi-to-test="xyznetdrv"
```

上記以外のプロパティーは、PIO を使用する周辺機器との間の読み取りおよび書き込みと、DMA を使用する周辺機器との間のデータ転送のための Oracle Solaris DDI データアクセスメカニズムの使用状況やハーネスチェックに関連するプロパティーです。

bofi-range-check　このプロパティーを設定すると、テストハーネスは PIO データアクセス関数に渡される引数の整合性をチェックします。

bofi-ddi-check　このプロパティーを設定すると、テストハーネスは、ddi_map_regs_setup(9F) によって返されるマップされたアドレスがデータアクセス関数のコンテキストの外部で使用されていないことを検証します。

bofi-sync-check　このプロパティーを設定すると、テストハーネスは DMA 関数の使用方法が正しいことを検証し、ドライバが仕様に準拠して ddi_dma_sync(9F) を使用していることを確認します。

ドライバのテスト

このセクションでは、th_define(1M) および th_manage(1M) コマンドを使用して、障害を作成および投入する方法について説明します。

障害の作成

th_define ユーティリティーは、errdef を定義するための、bofi デバイスドライバへのインタフェースを提供します。1 つの *errdef* が、デバイスドライバによるハードウェアへのアクセスを破壊する方法に関する 1 つの仕様に対応します。th_define のコマンド行引数は、投入する障害の正確な種類を指定します。指定した引数によって矛盾のない errdef が定義される場合、th_define プロセスは bofi ドライバに errdef を格納します。errdef で定められている基準に達するまで、プロセスは一時停止状態になります。実際には、アクセスカウントが 0 になると一時停止状態が終了します。

障害の投入

テストハーネスはデータアクセスのレベルで動作します。データアクセスには次のような特性があります。

- アクセスしているハードウェアの種類 (ドライバ名)
- アクセスしているハードウェアのインスタンス (ドライバインスタンス)
- テストされているレジスタセット
- ターゲットとなるレジスタセットのサブセット
- 転送の方向 (読み取りまたは書き込み)
- アクセスの種類 (PIO または DMA)

テストハーネスはデータアクセスを横取りし、適切な障害をドライバに投入します。th_define(1M) コマンドで指定された errdef によって、次の情報がエンコードされます。

- テストされているドライバインスタンスおよびレジスタセット (-n *name*、-i *instance*、および -r *reg_number*)。
- 破壊の対象となるレジスタセットのサブセット。このサブセットは、レジスタセット内のオフセットと、そのオフセットからの長さを指定することによって指示されます (-l *offset* [*len*])。
- 横取りするアクセスの種類: log、pio、dma、pio_r、pio_w、dma_r、dma_w、intr (-a *acc_types*)。
- 障害が発生するアクセスの数 (-c *count* [*failcount*])。
- 該当するアクセスに適用する破壊の種類 (-o *operator* [*operand*])。
  - データを固定値に置き換える (EQUAL)
  - データに対するビット単位の演算を実行する (AND、OR、XOR)

- 転送を無視する (ホストから入出力へのアクセス、NO_TRANSFER)
- 割り込みを失う、遅らせる、または偽の割り込みを投入する (LOSE、DELAY、EXTRA)

errdef でフレームワークの障害をシミュレートするには、-a *acc_chk* オプションを使用します。

障害投入プロセス

障害投入のプロセスには 2 つの段階があります。

1. th_define(1M) コマンドを使用して errdef を作成します。

   errdef を作成するときは、定義を格納する bofi ドライバにテスト定義を渡し、th_manage(1M) コマンドによって定義にアクセスできるようにします。

2. ワークロードを作成し、th_manage コマンドを使用して errdef を起動および管理します。

   th_manage コマンドは、bofi ハーネスドライバが認識する各種の ioctl へのユーザーインタフェースです。th_manage コマンドはドライバの名前およびインスタンスのレベルで動作し、アクセスハンドルを列挙する get_handles、errdef を起動する start、errdef を停止する stop などのコマンドを含んでいます。

   errdef を起動すると、対象となるデータアクセスで障害が発生します。th_manage ユーティリティーは、errdef の現在の状態を表示する broadcast と、errdef をクリアする clear_errors の各コマンドをサポートします。

   詳細は、th_define(1M) および th_manage(1M) のマニュアルページを参照してください。

テストハーネスの警告

次の方法で警告メッセージを処理するようにテストハーネスを構成できます。

- 警告メッセージをコンソールに書き込む
- 警告メッセージをコンソールに書き込んでからシステムをパニック状態にする

2 番目の方法を使用すると、問題の根本原因を特定しやすくなります。

bofi-range-check プロパティーの値を warn に設定した場合、テスト対象のドライバによる DDI 関数の範囲違反をハーネスが検出した時点で、ハーネスは次のメッセージを出力します (パニック動作が設定されている場合はパニック状態に移行します)。

```
ddi_getX() out of range addr %x not in %x
ddi_putX() out of range addr %x not in %x
ddi_rep_getX() out of range addr %x not in %x
ddi_rep_putX() out of range addr %x not in %x
```

X は 8、16、32、または 64 です。

ハーネスが 1000 を超える割り込みを挿入するように要求されており、ドライバが割り込みジャバーを検出しない場合、次のメッセージが出力されます。

```
undetected interrupt jabber - %s %d
```

スクリプトによるテストプロセスの自動化

th_define(1M) ユーティリティーのロギングアクセスタイプを使用すると、障害投入テストスクリプトを作成できます。

```
# th_define -n name -i instance -a log [-e fixup_script]
```

th_define コマンドはインスタンスをオフラインにし、またオンラインに戻します。th_define は次に、*fixup_script* によって記述されたワークロードを実行し、ドライバインスタンスが行う入出力アクセスをログに記録します。

fixup_script は、オプションのいくつかの引数を指定して 2 回呼び出されます。このスクリプトはインスタンスがオフラインにされる直前に 1 回呼び出され、オンラインに戻されたあとにふたたび呼び出されます。

次の変数が、呼び出される実行可能ファイルの環境に渡されます。

DRIVER_PATH	インスタンスのデバイスパス
DRIVER_INSTANCE	ドライバのインスタンス番号
DRIVER_UNCONFIGURE	インスタンスが間もなくオフラインにされるときは 1 に設定
DRIVER_CONFIGURE	インスタンスがオンラインに戻されたときは 1 に設定

fixup_script は通常、テスト中のデバイスがオフラインになってもよい状態 (未構成) であること、またはエラー投入に適した状態 (構成済み、エラーなし、ワークロードを処理中など) であることを確認します。次に示すのは、ネットワークドライバ用の最小限のスクリプトの例です。

```
#!/bin/ksh
driver=xyznetdrv
ifnum=$driver$DRIVER_INSTANCE

if [[ $DRIVER_CONFIGURE = 1 ]]; then
   ifconfig $ifnum plumb
   ifconfig $ifnum ...
   ifworkload start $ifnum
elif [[ $DRIVER_UNCONFIGURE = 1 ]]; then
   ifworkload stop $ifnum
   ifconfig $ifnum down
```

```
   ifconfig $ifnum unplumb
fi
exit $?
```

注 – `ifworkload` コマンドは、バックグラウンドタスクとしてワークロードを開始する必要があります。障害の投入は、*fixup_script* がテスト中のドライバを構成し、オンラインに戻したあとに行われます (DRIVER_CONFIGURE は 1 に設定されます)。

`-e` *fixup_script* オプションを指定する場合、コマンド行の最後のオプションとして指定する必要があります。`-e` オプションを指定しない場合、デフォルトのスクリプトが使用されます。デフォルトのスクリプトは、テスト中のデバイスのオフライン化とオンライン化を繰り返し試行します。そのため、ワークロードはドライバの `attach()` および `detach()` パスで構成されます。

結果のログは、自動での障害投入テストの実行に適した、いくつかの実行可能スクリプトに変換されます。これらのスクリプトは、`driver.test.id` という名前で、カレントディレクトリのサブディレクトリに作成されます。スクリプトは、*fixup_script* によって記述されたワークロードの実行中、一度に 1 つずつ障害を投入します。

ドライバのテスト担当者は、テスト自動化プロセスによって生成された errdef の大部分を制御できます。`th_define(1M)` のマニュアルページを参照してください。

テスト担当者がテストスクリプトに適した範囲のワークロードを選択すると、ハーネスはドライバ強化の多くの側面をテストできます。ただし、すべてを網羅するためには、テスト担当者が追加のテストケースを手作業で作成しなければならない場合があります。これらのケースをテストスクリプトに追加します。テストを時間内に完了させるために、重複したテストケースを手作業で削除することが必要になる場合があります。

自動テストプロセス

自動テストのプロセスは、次のようになります。

1. ドライバのどの部分をテストするかを識別します。

   次に示すような、ドライバがハードウェアと対話する部分はすべてテストします。

   - 接続と切り離し
   - スタック下の plumb と unplumb
   - 通常のデータ転送
   - ドキュメント化されたデバッグモード

   使用モードごとに別個のワークロードスクリプト (*fixup_script*) を生成する必要があります。

2. 使用モードごとに、デバイスを構成および構成解除し、ワークロードを作成および終了する実行可能プログラム (*fixup_script*) を準備します。
3. errdef とアクセスの種類 (`-a` *log*) を指定して、`th_define`(1M) コマンドを実行します。
4. ログがいっぱいになるまで待ちます。

   ログの内容は `bofi` ドライバの内部バッファーのダンプです。このデータはスクリプトの先頭部分に含まれています。

   ログの作成には数秒から数分かかるため、`th_manage broadcast` コマンドを使用して進捗状況を確認します。
5. 作成されたテストディレクトリに移動し、マスターテストスクリプトを実行します。

   マススクリプトは、生成された個々のテストスクリプトを順に実行します。レジスタセットごとに個別のテストスクリプトが生成されます。
6. 分析のために結果を格納します。

   `success (corruption reported)` や `success (corruption undetected)` のような成功したテスト結果は、テスト中のドライバが正常に動作していることを示します。ドライバが障害を報告したあとにサービスへの影響を報告できなかったことをハーネスが検出した場合や、アクセスハンドルまたは DMA ハンドルが障害とマーク付けされたことをドライバが検出できない場合、`failure (no service impact reported)` という結果が報告されます。

   `test not triggered` エラーが出力にいくつか含まれていても問題はありません。ただし、そのようなエラーが多数にのぼる場合は、テストが適切に機能していないことを示しています。これらのエラーは、テストスクリプトが生成されたときと同じレジスタにドライバがアクセスしていない場合に発生することがあります。
7. ドライバの複数のインスタンスに対して同時にテストを実行し、エラーパスのマルチスレッド化をテストします。

   たとえば、`th_define` コマンドは毎回、テストスクリプトとマススクリプトを収めたディレクトリを新しく作成します。

   ```
   # th_define -n xyznetdrv -i 0 -a log -e script
   # th_define -n xyznetdrv -i 1 -a log -e script
   ```

   マススクリプトが作成されたら、それらを同時に実行します。

注 – 生成されたスクリプトは、ロギング対象の errdef がアクティブであった期間中に記録されたログ内容に基づく障害投入のシミュレーションのみを実行します。開発者がワークロードを定義するときは、必要な結果が確実にログに記録されるようにします。また、結果のログと障害投入仕様を分析します。生成されたテストスクリプトによるハードウェアアクセスの範囲が、必要な基準を満たしていることを確認してください。

◆ ◆ ◆ 第 14 章

階層化ドライバインタフェース (LDI)

LDI は、カーネルモジュールがシステム内のほかのデバイスにアクセスできるようにする DDI/DKI のセットです。LDI を使用すると、カーネルモジュールによって現在使用されているデバイスを特定することもできます。

この章で扱う内容は、次のとおりです。

LDI の概要

LDI には次の 2 つのカテゴリのインタフェースが含まれます。

- カーネルインタフェース。ユーザーアプリケーションはシステムコールを使用して、デバイスドライバによって管理されているデバイスをカーネル内から開く、読み込む、および書き込む操作を行います。カーネルモジュールは LDI カーネルインタフェースを使用することで、別のデバイスドライバによって管理されているデバイスをカーネル内から開く、読み込む、および書き込む操作を行うことができます。たとえば、同じデバイスを読み取るために、ユーザーアプリケーションは `read(2)` を、カーネルモジュールは `ldi_read(9F)` を使用できます。278 ページの「カーネルインタフェース」を参照してください。
- ユーザーインタフェース。LDI ユーザーインタフェースは、現在カーネル内でどのデバイスがほかのデバイスによって使用されているかに関する情報をユーザープロセスに提供できます。295 ページの「ユーザーインタフェース」を参照してください。

LDI についての説明では、次の用語がよく使用されます。

- ターゲットデバイス。ターゲットデバイスとは、デバイスドライバによって管理され、デバイスコンシューマによってアクセスされているカーネル内のデバイスです。

- デバイスコンシューマ。デバイスコンシューマは、ターゲットデバイスを開いてアクセスするユーザープロセスまたはカーネルモジュールです。デバイスコンシューマは通常、ターゲットデバイスに対して `open`、`read`、`write`、`ioctl` などの操作を実行します。
- カーネルデバイスコンシューマ。カーネルデバイスコンシューマは、特定の種類のデバイスコンシューマです。カーネルデバイスコンシューマは、ターゲットデバイスにアクセスするカーネルモジュールです。カーネルデバイスコンシューマは通常、アクセスされているターゲットデバイスを管理しているデバイスドライバではありません。その代わりカーネルデバイスコンシューマは、ターゲットデバイスを管理しているデバイスドライバを通して間接的にターゲットデバイスにアクセスします。
- 階層化ドライバ。階層化ドライバは、特定の種類のカーネルデバイスコンシューマです。階層化ドライバは、どのハードウェアも直接には管理しないカーネルドライバです。その代わり階層化ドライバは、ターゲットデバイスを管理しているデバイスドライバを通して間接的に、1 つまたは複数のターゲットデバイスにアクセスします。ボリュームマネージャーと STREAMS マルチプレクサは階層化ドライバの代表的な例です。

カーネルインタフェース

いくつかの LDI カーネルインタフェースを使用すると、LDI でカーネルデバイスの使用状態情報を追跡し、報告できます。278 ページの「階層化識別子 – カーネルデバイスコンシューマ」を参照してください。

その他に、カーネルモジュールから `open`、`read`、`write` などのアクセス操作をターゲットデバイスで実行できる LDI カーネルインタフェースがあります。これらの LDI カーネルインタフェースを使用すると、カーネルデバイスコンシューマから、ターゲットデバイスに関するプロパティーやイベント情報を照会することもできます。279 ページの「階層化ドライバのハンドル – ターゲットデバイス」を参照してください。

284 ページの「LDI カーネルインタフェースの例」では、これらの LDI インタフェースの多くを使用するドライバの例を示しています。

階層化識別子 – カーネルデバイスコンシューマ

LDI は階層化識別子によって、カーネルデバイスの使用状態情報を追跡し、報告できます。階層化識別子 (`ldi_ident_t`) は、カーネルデバイスコンシューマを識別します。カーネルデバイスコンシューマは LDI を使用してターゲットデバイスを開く前に階層化識別子を取得する必要があります。

階層化ドライバは、唯一サポートされている種類のカーネルデバイスコンシューマです。そのため、階層化ドライバはデバイス番号に関連付けられている階層化識別

子、デバイス情報ノード、または階層化ドライバのストリームを取得する必要があります。階層化識別子は階層化ドライバに関連付けられていますが、ターゲットデバイスとは関連付けられていません。

LDI によって収集されたカーネルデバイスの使用状態情報は、libdevinfo(3LIB) インタフェース、fuser(1M) コマンド、または prtconf(1M) コマンドを使用すると取得できます。たとえば、prtconf(1M) コマンドでは、階層化ドライバがアクセスしているターゲットデバイスを表示したり、特定のターゲットデバイスにアクセスしているの階層化ドライバを表示したりできます。デバイス使用状態情報を取得する方法の詳細については、295 ページの「ユーザーインタフェース」を参照してください。

次に、LDI 階層化識別子インタフェースについて説明します。

ldi_ident_t	階層化識別子。不透明な型です。
ldi_ident_from_dev(9F)	dev_t デバイス番号に関連付けられる階層化識別子の割り当てと取得を行います。
ldi_ident_from_dip(9F)	dev_info_t デバイス情報ノードに関連付けられる階層化識別子の割り当てと取得を行います。
ldi_ident_from_stream(9F)	ストリームに関連付けられる階層化識別子の割り当てと取得を行います。
ldi_ident_release(9F)	ldi_ident_from_dev(9F)、ldi_ident_from_dip(9F)、または ldi_ident_from_stream(9F) に関連付けられていた階層化識別子を解放します。

階層化ドライバのハンドル – ターゲットデバイス

カーネルデバイスコンシューマは階層化ドライバのハンドル (ldi_handle_t) を使用して、LDI インタフェースを通してターゲットデバイスにアクセスする必要があります。ldi_handle_t 型は LDI インタフェースでのみ有効です。LDI は、デバイスが正常に開かれたときにこのハンドルを割り当てて返します。次にカーネルデバイスコンシューマはこのハンドルを使用することで、LDI インタフェースを通してターゲットデバイスにアクセスできます。LDI はデバイスを閉じるときにハンドルの割り当てを解除します。例については、284 ページの「LDI カーネルインタフェースの例」を参照してください。

このセクションでは、カーネルデバイスコンシューマからターゲットデバイスにアクセスし、さまざまな種類の情報を取得する方法について説明します。カーネルデバイスコンシューマでターゲットデバイスを開いたり閉じたりする方法については、280 ページの「ターゲットデバイスを開く操作と閉じる操作」を参照してください。カーネルデバイスコンシューマから、ターゲットデバイスで read、write、strategy、ioctl などの操作を実行する方法については、280 ページの「ターゲットデバイスへのアクセス」を参照してください。282 ページ

の「ターゲットデバイス情報の取得」では、デバイス公開型やデバイスマイナー名などのターゲットデバイスの情報を取得するインタフェースについて説明します。282 ページの「ターゲットデバイスのプロパティー値の取得」では、ターゲットデバイスのプロパティーの値とアドレスを取得するインタフェースについて説明します。カーネルデバイスコンシューマでターゲットデバイスからイベント通知を受信する方法については、283 ページの「非同期デバイスイベントの通知の受信」を参照してください。

ターゲットデバイスを開く操作と閉じる操作

このセクションでは、ターゲットデバイスを開き、閉じるための LDI カーネルインタフェースについて説明します。open インタフェースは、階層化ドライバのハンドルへのポインタを取得します。open インタフェースは、デバイス番号、デバイス ID、またはパス名によって指定されたターゲットデバイスを開こうとします。開く操作が成功すると、open インタフェースはターゲットデバイスにアクセスするために使用できる階層化ドライバのハンドルを割り当てて返します。close インタフェースは、指定された階層化ドライバのハンドルと関連付けられているターゲットデバイスを閉じ、階層化ドライバのハンドルを解放します。

`ldi_handle_t`	ターゲットデバイスアクセス用の階層化ドライバのハンドルです。デバイスが正常に開かれたときに返される不透明なデータ構造体です。
`ldi_open_by_dev(9F)`	`dev_t` デバイス番号パラメータによって指定されたデバイスを開きます。
`ldi_open_by_devid(9F)`	`ddi_devid_t` デバイス ID パラメータによって指定されたデバイスを開きます。開くマイナーノード名も指定する必要があります。
`ldi_open_by_name(9F)`	パス名によってデバイスを開きます。パス名は、カーネルアドレス空間内の NULL で終わっている文字列です。パス名はスラッシュ (/) で始まる絶対パスにする必要があります。
`ldi_close(9F)`	`ldi_open_by_dev`(9F)、`ldi_open_by_devid`(9F)、または `ldi_open_by_name` (9F) によって開かれたデバイスを閉じます。`ldi_close`(9F) が返ったあと、閉じられたデバイスの階層化ドライバのハンドルは有効ではなくなります。

ターゲットデバイスへのアクセス

このセクションでは、ターゲットデバイスにアクセスするための LDI カーネルインタフェースについて説明します。これらのインタフェースによって、カーネルデバイスコンシューマは、階層化ドライバのハンドルによって指定されたターゲットデ

バイスで操作を実行できます。カーネルデバイスコンシューマは、ターゲットデバイスで read、write、strategy、ioctl などの操作を実行できます。

ldi_handle_t	ターゲットデバイスアクセス用の階層化ドライバのハンドルです。不透明なデータ構造体です。
ldi_read(9F)	ターゲットデバイスのデバイスエントリポイントに read 要求を渡します。この操作はブロックデバイス、文字デバイス、および STREAMS デバイスでサポートされます。
ldi_aread(9F)	ターゲットデバイスのデバイスエントリポイントに非同期の read 要求を渡します。この操作はブロックデバイスと文字デバイスでサポートされます。
ldi_write(9F)	ターゲットデバイスのデバイスエントリポイントに write 要求を渡します。この操作はブロックデバイス、文字デバイス、および STREAMS デバイスでサポートされます。
ldi_awrite(9F)	ターゲットデバイスのデバイスエントリポイントに非同期の write 要求を渡します。この操作はブロックデバイスと文字デバイスでサポートされます。
ldi_strategy(9F)	ターゲットデバイスのデバイスエントリポイントに strategy 要求を渡します。この操作はブロックデバイスと文字デバイスでサポートされます。
ldi_dump(9F)	ターゲットデバイスのデバイスエントリポイントに dump 要求を渡します。この操作はブロックデバイスと文字デバイスでサポートされます。
ldi_poll(9F)	ターゲットデバイスのデバイスエントリポイントに poll 要求を渡します。この操作はブロックデバイス、文字デバイス、および STREAMS デバイスでサポートされます。
ldi_ioctl(9F)	ターゲットデバイスのデバイスエントリポイントに ioctl 要求を渡します。この操作はブロックデバイス、文字デバイス、および STREAMS デバイスでサポートされます。LDI は STREAMS リンクと STREAMS ioctl コマンドをサポートしています。ldi_ioctl(9F) のマニュアルページの「STREAM IOCTLS」セクションを参照してください。streamio(7I) のマニュアルページの ioctl コマンドも参照してください。
ldi_devmap(9F)	ターゲットデバイスのデバイスエントリポイントに devmap 要求を渡します。この操作はブロックデバイスと文字デバイスでサポートされます。
ldi_getmsg(9F)	ストリームからメッセージブロックを取得します。
ldi_putmsg(9F)	メッセージブロックをストリーム上に配置します。

ターゲットデバイス情報の取得

このセクションでは、指定したターゲットデバイスに関するデバイス情報を取得するためにカーネルデバイスコンシューマが使用できる LDI インタフェースについて説明します。ターゲットデバイスは、階層化ドライバのハンドルによって指定します。カーネルデバイスコンシューマは、デバイス番号、デバイス公開型、デバイス ID、デバイスマイナー名、デバイスサイズなどの情報を受け取ることができます。

ldi_get_dev(9F)	階層化ドライバのハンドルによって指定されたターゲットデバイスの dev_t デバイス番号を取得します。
ldi_get_otyp(9F)	階層化ドライバのハンドルによって指定されたターゲットデバイスを開くために使用された open フラグを取得します。このフラグは、ターゲットデバイスが文字デバイスであるかブロックデバイスであるかを通知します。
ldi_get_devid(9F)	階層化ドライバのハンドルによって指定されたターゲットデバイスの ddi_devid_t デバイス ID を取得します。デバイス ID の使用を終えて ddi_devid_t を解放するには、ddi_devid_free(9F) を使用します。
ldi_get_minor_name(9F)	ターゲットデバイスのために開かれたマイナーノードの名前を格納しているバッファーを取得します。マイナーノード名の使用を終えてバッファーを解放するには、kmem_free(9F) を使用します。
ldi_get_size(9F)	階層化ドライバのハンドルによって指定されたターゲットデバイスのパーティションサイズを取得します。

ターゲットデバイスのプロパティー値の取得

このセクションでは、指定したターゲットデバイスに関するプロパティー情報を取得するためにカーネルデバイスコンシューマが使用できる LDI インタフェースについて説明します。ターゲットデバイスは、階層化ドライバのハンドルによって指定します。カーネルデバイスコンシューマはプロパティーの値とアドレスを受け取って、プロパティーが存在するかどうかを判定できます。

ldi_prop_exists(9F)	階層化ドライバのハンドルによって指定されたターゲットデバイスのプロパティーが存在する場合は 1 を返します。指定されたターゲットデバイスのプロパティーが存在しない場合は 0 を返します。
ldi_prop_get_int(9F)	階層化ドライバのハンドルによって指定されたターゲットデバイスに関連付けられている

	int 整数プロパティーを検索します。整数プロパティーが見つかった場合はプロパティー値を返します。
ldi_prop_get_int64(9F)	階層化ドライバのハンドルによって指定されたターゲットデバイスに関連付けられている int64_t 整数プロパティーを検索します。整数プロパティーが見つかった場合はプロパティー値を返します。
ldi_prop_lookup_int_array(9F)	階層化ドライバのハンドルによって指定されたターゲットデバイスの int 整数配列プロパティー値のアドレスを取得します。
ldi_prop_lookup_int64_array(9F)	階層化ドライバのハンドルによって指定されたターゲットデバイスの int64_t 整数配列プロパティー値のアドレスを取得します。
ldi_prop_lookup_string(9F)	階層化ドライバのハンドルによって指定されたターゲットデバイスの NULL で終わっている文字列プロパティー値のアドレスを取得します。
ldi_prop_lookup_string_array(9F)	文字列の配列のアドレスを取得します。文字列の配列は、階層化ドライバのハンドルによって指定されたターゲットデバイスの NULL で終わっている文字列プロパティー値へのポインタの配列です。
ldi_prop_lookup_byte_array(9F)	バイトの配列のアドレスを取得します。バイトの配列は、階層化ドライバのハンドルによって指定されたターゲットデバイスのプロパティー値です。

非同期デバイスイベントの通知の受信

カーネルデバイスコンシューマは LDI によって、イベント通知を登録し、ターゲットデバイスからイベント通知を受信できます。カーネルデバイスコンシューマは、イベントが発生したときに呼び出されるイベントハンドラを登録できます。カーネルデバイスコンシューマで LDI イベント通知インタフェースを使用してイベント通知を登録するには、カーネルデバイスコンシューマでデバイスを開き、階層化ドライバのハンドルを受信する必要があります。

LDI イベント通知インタフェースによって、カーネルデバイスコンシューマはイベント名を指定し、関連付けられているカーネルイベントの cookie を取得できます。カーネルデバイスコンシューマは次に、階層化ドライバのハンドル (ldi_handle_t)、cookie (ddi_eventcookie_t)、および ldi_add_event_handler(9F) への

イベントハンドラを渡して、イベント通知を登録できます。登録が正常に完了すると、カーネルデバイスコンシューマは一意の LDI イベントハンドラ識別子 (ldi_callback_id_t) を受信します。LDI イベントハンドラ識別子は、LDI イベント通知インタフェースでのみ使用できる不透明な型です。

LDI はほかのデバイスによって生成されたイベントを登録するための枠組みを提供します。LDI 自体は、イベントの種類を定義したり、イベントを生成するためのインタフェースを提供したりすることはありません。

次に、LDI 非同期イベント通知インタフェースについて説明します。

ldi_callback_id_t	イベントハンドラ識別子。不透明な型です。
ldi_get_eventcookie(9F)	階層化ドライバのハンドルによって指定されたターゲットデバイスのイベントサービス cookie を取得します。
ldi_add_event_handler(9F)	ldi_callback_id_t 登録識別子によって指定されたコールバックハンドラを追加します。コールバックハンドラは、ddi_eventcookie_t cookie によって指定されたイベントが発生したときに呼び出されます。
ldi_remove_event_handler(9F)	ldi_callback_id_t 登録識別子によって指定されたコールバックハンドラを削除します。

LDI カーネルインタフェースの例

このセクションでは、この章の前のセクションで説明した LDI 呼び出しを使用するカーネルデバイスコンシューマの例を示します。このセクションでは、このモジュール例の次の面について説明します。

- 285 ページの「デバイス構成ファイル」
- 285 ページの「ドライバソースファイル」
- 294 ページの「階層化ドライバのテスト」

このカーネルデバイスコンシューマの例は lyr という名前です。lyr モジュールは、LDI 呼び出しを使用してターゲットデバイスにデータを送信する階層化ドライバです。lyr ドライバは open(9E) エントリポイントで、lyr.conf 構成ファイル内の lyr_targ プロパティーによって指定されたデバイスを開きます。lyr ドライバは write(9E) エントリポイントで、lyr_targ プロパティーによって指定されたデバイスに、すべての受信データを書き込みます。

デバイス構成ファイル

次に示す構成ファイルでは、lyr ドライバの書き込み先ターゲットデバイスはコンソールです。

例 14–1　構成ファイル

```
#
# Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
#pragma ident    "%Z%%M%    %I%    %E% SMI"

name="lyr" parent="pseudo" instance=1;
lyr_targ="/dev/console";
```

ドライバソースファイル

次に示したドライバソースファイルで、lyr_state_t 構造体は lyr ドライバのソフト状態を保持しています。ソフト状態には、lyr_targ デバイスの階層化ドライバのハンドル (lh) と、lyr デバイスの階層化識別子 (li) が格納されています。ソフト状態の詳細については、580 ページの「ドライバのソフト状態情報の取得」を参照してください。

lyr_open() エントリポイントで、ddi_prop_lookup_string(9F) は lyr_targ プロパティーから、開く lyr デバイスのターゲットデバイスの名前を取得しています。ldi_ident_from_dev(9F) 関数は lyr デバイスの LDI 階層化識別子を取得します。ldi_open_by_name(9F) 関数は lyr_targ デバイスを開き、lyr_targ デバイスの階層化ドライバのハンドルを取得します。

lyr_open() で何らかのエラーが発生した場合、ldi_close(9F)、ldi_ident_release(9F)、および ddi_prop_free(9F) の呼び出しによって、実行されたすべてのことが取り消されます。ldi_close(9F) 関数は lyr_targ デバイスを閉じます。ldi_ident_release(9F) 関数は lyr 階層化識別子を解放します。ddi_prop_free(9F) 関数は、lyr_targ デバイス名が取得されたときに割り当てられたリソースを解放します。エラーが発生しなければ、lyr_close() エントリポイントで ldi_close(9F) 関数と ldi_ident_release(9F) 関数が呼び出されます。

ドライバモジュールの最終行では、ldi_write(9F) 関数が呼び出されています。ldi_write(9F) 関数は lyr_write() エントリポイントで lyr デバイスに書き込まれたデータを取得し、そのデータを lyr_targ デバイスに書き込みます。ldi_write(9F) 関数は lyr_targ デバイスの階層化ドライバのハンドルを使用して、データを lyr_targ デバイスに書き込みます。

例 14–2　ドライバソースファイル

```
#include <sys/types.h>
#include <sys/file.h>
#include <sys/errno.h>
```

例 14–2 ドライバソースファイル (続き)

```
#include <sys/open.h>
#include <sys/cred.h>
#include <sys/cmn_err.h>
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/stat.h>

#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/sunldi.h>

typedef struct lyr_state {
    ldi_handle_t    lh;
    ldi_ident_t     li;
    dev_info_t      *dip;
    minor_t         minor;
    int             flags;
    kmutex_t        lock;
} lyr_state_t;

#define LYR_OPENED      0x1     /* lh is valid */
#define LYR_IDENTED     0x2     /* li is valid */


static int lyr_info(dev_info_t *, ddi_info_cmd_t, void *, void **);
static int lyr_attach(dev_info_t *, ddi_attach_cmd_t);
static int lyr_detach(dev_info_t *, ddi_detach_cmd_t);

static int lyr_open(dev_t *, int, int, cred_t *);
static int lyr_close(dev_t, int, int, cred_t *);
static int lyr_write(dev_t, struct uio *, cred_t *);

static void *lyr_statep;

static struct cb_ops lyr_cb_ops = {
    lyr_open,        /* open */
    lyr_close,       /* close */
    nodev,           /* strategy */
    nodev,           /* print */
    nodev,           /* dump */
    nodev,           /* read */
    lyr_write,       /* write */
    nodev,           /* ioctl */
    nodev,           /* devmap */
    nodev,           /* mmap */
    nodev,           /* segmap */
    nochpoll,        /* poll */
    ddi_prop_op,     /* prop_op */
    NULL,            /* streamtab  */
    D_NEW | D_MP,    /* cb_flag */
    CB_REV,          /* cb_rev */
    nodev,           /* aread */
    nodev            /* awrite */
};
```

例 14–2 ドライバソースファイル　(続き)

```
static struct dev_ops lyr_dev_ops = {
    DEVO_REV,        /* devo_rev, */
    0,               /* refcnt  */
    lyr_info,        /* getinfo */
    nulldev,         /* identify */
    nulldev,         /* probe */
    lyr_attach,      /* attach */
    lyr_detach,      /* detach */
    nodev,           /* reset */
    &lyr_cb_ops,     /* cb_ops */
    NULL,            /* bus_ops */
    NULL             /* power */
};

static struct modldrv modldrv = {
    &mod_driverops,
    "LDI example driver",
    &lyr_dev_ops
};

static struct modlinkage modlinkage = {
    MODREV_1,
    &modldrv,
    NULL
};


int
_init(void)
{
    int rv;

    if ((rv = ddi_soft_state_init(&lyr_statep, sizeof (lyr_state_t),
        0)) != 0) {
        cmn_err(CE_WARN, "lyr _init: soft state init failed\n");
        return (rv);
    }

    if ((rv = mod_install(&modlinkage)) != 0) {
        cmn_err(CE_WARN, "lyr _init: mod_install failed\n");
        goto FAIL;
    }

    return (rv);
    /*NOTEREACHED*/
FAIL:
    ddi_soft_state_fini(&lyr_statep);
    return (rv);
}


int
_info(struct modinfo *modinfop)
{
    return (mod_info(&modlinkage, modinfop));
```

例 14–2 ドライバソースファイル (続き)

```
}


int
_fini(void)
{
    int rv;

    if ((rv = mod_remove(&modlinkage)) != 0) {
        return(rv);
    }

    ddi_soft_state_fini(&lyr_statep);

    return (rv);
}

/*
 * 1:1 mapping between minor number and instance
 */
static int
lyr_info(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
{
    int inst;
    minor_t minor;
    lyr_state_t *statep;
    char *myname = "lyr_info";

    minor = getminor((dev_t)arg);
    inst = minor;
    switch (infocmd) {
    case DDI_INFO_DEVT2DEVINFO:
        statep = ddi_get_soft_state(lyr_statep, inst);
        if (statep == NULL) {
            cmn_err(CE_WARN, "%s: get soft state "
                "failed on inst %d\n", myname, inst);
            return (DDI_FAILURE);
        }
        *result = (void *)statep->dip;
        break;
    case DDI_INFO_DEVT2INSTANCE:
        *result = (void *)inst;
        break;
    default:
        break;
    }

    return (DDI_SUCCESS);
}


static int
lyr_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int inst;
```

例 14–2 ドライバソースファイル　(続き)

```
    lyr_state_t *statep;
    char *myname = "lyr_attach";

    switch (cmd) {
    case DDI_ATTACH:
        inst = ddi_get_instance(dip);

        if (ddi_soft_state_zalloc(lyr_statep, inst) != DDI_SUCCESS) {
            cmn_err(CE_WARN, "%s: ddi_soft_state_zallac failed "
                "on inst %d\n", myname, inst);
            goto FAIL;
        }

        statep = (lyr_state_t *)ddi_get_soft_state(lyr_statep, inst);
        if (statep == NULL) {
            cmn_err(CE_WARN, "%s: ddi_get_soft_state failed on "
                "inst %d\n", myname, inst);
            goto FAIL;
        }
        statep->dip = dip;
        statep->minor = inst;

        if (ddi_create_minor_node(dip, "node", S_IFCHR, statep->minor,
            DDI_PSEUDO, 0) != DDI_SUCCESS) {
            cmn_err(CE_WARN, "%s: ddi_create_minor_node failed on "
                "inst %d\n", myname, inst);
            goto FAIL;
        }
        mutex_init(&statep->lock, NULL, MUTEX_DRIVER, NULL);
        return (DDI_SUCCESS);

    case DDI_RESUME:
    case DDI_PM_RESUME:
    default:
        break;
    }
    return (DDI_FAILURE);
    /*NOTREACHED*/
FAIL:
    ddi_soft_state_free(lyr_statep, inst);
    ddi_remove_minor_node(dip, NULL);
    return (DDI_FAILURE);
}


static int
lyr_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    int inst;
    lyr_state_t *statep;
    char *myname = "lyr_detach";

    inst = ddi_get_instance(dip);
    statep = ddi_get_soft_state(lyr_statep, inst);
    if (statep == NULL) {
```

例 14–2 ドライバソースファイル (続き)

```
        cmn_err(CE_WARN, "%s: get soft state failed on "
            "inst %d\n", myname, inst);
        return (DDI_FAILURE);
    }
    if (statep->dip != dip) {
        cmn_err(CE_WARN, "%s: soft state does not match devinfo "
            "on inst %d\n", myname, inst);
        return (DDI_FAILURE);
    }

    switch (cmd) {
    case DDI_DETACH:
        mutex_destroy(&statep->lock);
        ddi_soft_state_free(lyr_statep, inst);
        ddi_remove_minor_node(dip, NULL);
        return (DDI_SUCCESS);
    case DDI_SUSPEND:
    case DDI_PM_SUSPEND:
    default:
        break;
    }
    return (DDI_FAILURE);
}

/*
 * on this driver's open, we open the target specified by a property and store
 * the layered handle and ident in our soft state.  a good target would be
 * "/dev/console" or more interestingly, a pseudo terminal as specified by the
 * tty command
 */
/*ARGSUSED*/
static int
lyr_open(dev_t *devtp, int oflag, int otyp, cred_t *credp)
{
    int rv, inst = getminor(*devtp);
    lyr_state_t *statep;
    char *myname = "lyr_open";
    dev_info_t *dip;
    char *lyr_targ = NULL;

    statep = (lyr_state_t *)ddi_get_soft_state(lyr_statep, inst);
    if (statep == NULL) {
        cmn_err(CE_WARN, "%s: ddi_get_soft_state failed on "
            "inst %d\n", myname, inst);
        return (EIO);
    }
    dip = statep->dip;

    /*
     * our target device to open should be specified by the "lyr_targ"
     * string property, which should be set in this driver's .conf file
     */
    if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, DDI_PROP_NOTPROM,
        "lyr_targ", &lyr_targ) != DDI_PROP_SUCCESS) {
        cmn_err(CE_WARN, "%s: ddi_prop_lookup_string failed on "
```

例 14–2 ドライバソースファイル　(続き)

```
            "inst %d\n", myname, inst);
        return (EIO);
    }

    /*
     * since we only have one pair of lh's and li's available, we don't
     * allow multiple on the same instance
     */
    mutex_enter(&statep->lock);
    if (statep->flags & (LYR_OPENED | LYR_IDENTED)) {
        cmn_err(CE_WARN, "%s: multiple layered opens or idents "
            "from inst %d not allowed\n", myname, inst);
        mutex_exit(&statep->lock);
        ddi_prop_free(lyr_targ);
        return (EIO);
    }

    rv = ldi_ident_from_dev(*devtp, &statep->li);
    if (rv != 0) {
        cmn_err(CE_WARN, "%s: ldi_ident_from_dev failed on inst %d\n",
            myname, inst);
        goto FAIL;
    }
    statep->flags |= LYR_IDENTED;

    rv = ldi_open_by_name(lyr_targ, FREAD | FWRITE, credp, &statep->lh,
        statep->li);
    if (rv != 0) {
        cmn_err(CE_WARN, "%s: ldi_open_by_name failed on inst %d\n",
            myname, inst);
        goto FAIL;
    }
    statep->flags |= LYR_OPENED;

    cmn_err(CE_CONT, "\n%s: opened target '%s' successfully on inst %d\n",
        myname, lyr_targ, inst);
    rv = 0;

FAIL:
    /* cleanup on error */
    if (rv != 0) {
        if (statep->flags & LYR_OPENED)
            (void)ldi_close(statep->lh, FREAD | FWRITE, credp);
        if (statep->flags & LYR_IDENTED)
            ldi_ident_release(statep->li);
        statep->flags &= ~(LYR_OPENED | LYR_IDENTED);
    }
    mutex_exit(&statep->lock);

    if (lyr_targ != NULL)
        ddi_prop_free(lyr_targ);
    return (rv);
}

/*
```

例 14–2 ドライバソースファイル　(続き)

```
 * on this driver's close, we close the target indicated by the lh member
 * in our soft state and release the ident, li as well.  in fact, we MUST do
 * both of these at all times even if close yields an error because the
 * device framework effectively closes the device, releasing all data
 * associated with it and simply returning whatever value the target's
 * close(9E) returned.  therefore, we must as well.
 */
/*ARGSUSED*/
static int
lyr_close(dev_t devt, int oflag, int otyp, cred_t *credp)
{
    int rv, inst = getminor(devt);
    lyr_state_t *statep;
    char *myname = "lyr_close";

    statep = (lyr_state_t *)ddi_get_soft_state(lyr_statep, inst);
    if (statep == NULL) {
        cmn_err(CE_WARN, "%s: ddi_get_soft_state failed on "
            "inst %d\n", myname, inst);
        return (EIO);
    }

    mutex_enter(&statep->lock);

    rv = ldi_close(statep->lh, FREAD | FWRITE, credp);
    if (rv != 0) {
        cmn_err(CE_WARN, "%s: ldi_close failed on inst %d, but will ",
            "continue to release ident\n", myname, inst);
    }
    ldi_ident_release(statep->li);
    if (rv == 0) {
        cmn_err(CE_CONT, "\n%s: closed target successfully on "
            "inst %d\n", myname, inst);
    }
    statep->flags &= ~(LYR_OPENED | LYR_IDENTED);

    mutex_exit(&statep->lock);
    return (rv);
}

/*
 * echo the data we receive to the target
 */
/*ARGSUSED*/
static int
lyr_write(dev_t devt, struct uio *uiop, cred_t *credp)
{
    int rv, inst = getminor(devt);
    lyr_state_t *statep;
    char *myname = "lyr_write";

    statep = (lyr_state_t *)ddi_get_soft_state(lyr_statep, inst);
    if (statep == NULL) {
        cmn_err(CE_WARN, "%s: ddi_get_soft_state failed on "
            "inst %d\n", myname, inst);
```

例 14–2 ドライバソースファイル　(続き)

```
        return (EIO);
    }

    return (ldi_write(statep->lh, uiop, credp));
}
```

▼ 階層化ドライバを構築してロードする方法

1. ドライバをコンパイルします。

   -D_KERNEL オプションを使用して、これがカーネルモジュールであることを示します。

   - SPARC アーキテクチャー向けにコンパイルする場合は、-xarch=v9 オプションを使用します。

     ```
     % cc -c -D_KERNEL -xarch=v9 lyr.c
     ```

   - 32 ビット x86 アーキテクチャー向けにコンパイルする場合は、次のコマンドを使用します。

     ```
     % cc -c -D_KERNEL lyr.c
     ```

2. ドライバをリンクします。

   ```
   % ld -r -o lyr lyr.o
   ```

3. 構成ファイルをインストールします。

   root ユーザーとして、マシンのカーネルドライバ領域に構成ファイルをコピーします。

   ```
   # cp lyr.conf /usr/kernel/drv
   ```

4. ドライババイナリをインストールします。

   - root ユーザーとして、SPARC アーキテクチャー上の sparcv9 ドライバ領域にドライババイナリをコピーします。

     ```
     # cp lyr /usr/kernel/drv/sparcv9
     ```

   - root ユーザーとして、32 ビット x86 アーキテクチャー上の drv ドライバ領域にドライババイナリをコピーします。

     ```
     # cp lyr /usr/kernel/drv
     ```

5. ドライバをロードします。

   root ユーザーとして、add_drv(1M) コマンドを使用してドライバをロードします。

   ```
   # add_drv lyr
   ```

疑似デバイスを一覧表示して、lyr デバイスが存在するようになったことを確認します。

```
# ls /devices/pseudo | grep lyr
lyr@1
lyr@1:node
```

階層化ドライバのテスト

lyr ドライバをテストするには、lyr デバイスにメッセージを書き込み、メッセージが lyr_targ デバイスに表示されることを確認します。

例 14–3 階層化デバイスへの短いメッセージの書き込み

この例では、lyr_targ デバイスが、lyr デバイスがインストールされているシステムのコンソールです。

参照している表示が、lyr デバイスがインストールされているシステムのコンソールデバイスの表示でもある場合は、コンソールに書き込むと表示が壊れることに注意してください。コンソールメッセージはウィンドウシステムの外側に表示されます。lyr ドライバのテスト後、表示を再描画またはリフレッシュする必要があります。

参照している表示が、lyr デバイスがインストールされているシステムのコンソールデバイスの表示ではない場合、ログインするなどして、ターゲットコンソールデバイスの表示内容を取得します。

次のコマンドでは、非常に短いメッセージが lyr デバイスに書き込まれます。

```
# echo "\n\n\t===> Hello World!! <===\n" > /devices/pseudo/lyr@1:node
```

ターゲットコンソールに次のメッセージが表示されます。

```
console login:

    ===> Hello World!! <===

lyr:
lyr_open: opened target ’/dev/console’ successfully on inst 1
lyr:
lyr_close: closed target successfully on inst 1
```

lyr_open() および lyr_close() からのメッセージは、lyr_open() および lyr_close() のエントリポイントの cmn_err(9F) 呼び出しからのものです。

例 14–4 階層化デバイスへの長いメッセージの書き込み

次のコマンドでは、長いメッセージが lyr デバイスに書き込まれます。

```
# cat lyr.conf > /devices/pseudo/lyr@1:node
```

ターゲットコンソールに次のメッセージが表示されます。

```
lyr:
lyr_open: opened target ’/dev/console’ successfully on inst 1
#
# Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
#pragma ident   "%Z%%M% %I%     %E% SMI"

name="lyr" parent="pseudo" instance=1;
lyr_targ="/dev/console";
lyr:
lyr_close: closed target successfully on inst 1
```

例 14–5 ターゲットデバイスの変更

ターゲットデバイスを変更するには、/usr/kernel/drv/lyr.conf を編集し、lyr_targ プロパティーの値を、異なるターゲットデバイスへのパスに変更します。たとえば、ローカル端末での tty コマンドの出力をターゲットデバイスにできます。/dev/pts/4 はそうしたデバイスパスの例です。

新しいターゲットデバイスを使用するようにドライバを更新する前に、lyr デバイスを使用中でないことを確認してください。

```
# modinfo -c | grep lyr
174          3 lyr                                 UNLOADED/UNINSTALLED
```

lyr.conf 構成ファイルを再ロードするには、update_drv(1M) コマンドを使用します。

```
# update_drv lyr
```

もう一度 lyr デバイスにメッセージを書き込み、メッセージが新しい lyr_targ デバイスに表示されることを確認します。

ユーザーインタフェース

LDI には、デバイスの階層化と使用状態の情報を報告する、ユーザーレベルのライブラリとコマンドインタフェースが含まれています。296 ページの「デバイス情報ライブラリインタフェース」では、デバイス階層化の情報を報告するための libdevinfo(3LIB) インタフェースについて説明します。298 ページの「システム構成

の出力コマンドインタフェース」 では、カーネルデバイスの使用状態情報を報告する、prtconf(1M) インタフェースについて説明します。301 ページの「デバイスユーザーコマンドインタフェース」 では、デバイスコンシューマの情報を報告する、fuser(1M) インタフェースについて説明します。

デバイス情報ライブラリインタフェース

LDI には、デバイス階層化情報のスナップショットを報告する libdevinfo(3LIB) インタフェースが用意されています。デバイス階層化は、システム内の 1 つのデバイスが、システム内の別のデバイスのコンシューマであるときに発生します。デバイス階層化情報が報告されるのは、コンシューマとターゲットの両方が、スナップショット内に含まれる 1 つのデバイスノードにバインドされている場合のみです。

デバイス階層化情報は、libdevinfo(3LIB) インタフェースによって有向グラフとして報告されます。*i* ノードは、グラフの頂点を表し、デバイスノードにバインドされた抽象概念です。libdevinfo(3LIB) インタフェースを使用すると、ノードの名前やデバイス番号などの i ノードのプロパティーにアクセスできます。

グラフのエッジはリンクによって表されます。リンクには、デバイスコンシューマを表すソース i ノードがあります。また、リンクにはターゲットデバイスを表すターゲット i ノードがあります。

次に、libdevinfo(3LIB) デバイス階層化情報のインタフェースについて説明します。

DINFOLYR	デバイス階層化情報を得られるようにするスナップショットフラグです。
di_link_t	2 つのエンドポイント間の有向リンクです。各エンドポイントは 1 つの di_lnode_t です。不透明な構造体です。
di_lnode_t	リンクのエンドポイントです。不透明な構造体です。di_lnode_t は di_node_t にバインドされます。
di_node_t	1 つのデバイスノードを表します。不透明な構造体です。di_node_t は di_lnode_t にバインドされるとは限りません。
di_walk_link(3DEVINFO)	スナップショット内のすべてのリンクを調べます。
di_walk_lnode(3DEVINFO)	スナップショット内のすべての i ノードを調べます。

di_link_next_by_node(3DEVINFO)	指定された di_node_t ノードがソースとターゲットのいずれかである次のリンクへのハンドルを取得します。
di_link_next_by_lnode(3DEVINFO)	指定された di_lnode_t i ノードがソースとターゲットのいずれかである次のリンクへのハンドルを取得します。
di_link_to_lnode(3DEVINFO)	di_link_t リンクの指定されたエンドポイントに対応する i ノードを取得します。
di_link_spectype(3DEVINFO)	リンクの spectype を取得します。spectype は、ターゲットデバイスがアクセスされる方法を示します。ターゲットデバイスはターゲット i ノードによって表されます。
di_lnode_next(3DEVINFO)	指定の di_node_t デバイスノードに関連付けられている、指定された di_lnode_t i ノードの次のオカレンスへのハンドルを取得します。
di_lnode_name(3DEVINFO)	指定された i ノードに関連付けられている名前を取得します。
di_lnode_devinfo(3DEVINFO)	指定された i ノードに関連付けられているデバイスノードへのハンドルを取得します。
di_lnode_devt(3DEVINFO)	指定された i ノードに関連付けられているデバイスノードのデバイス番号を取得します。

LDI によって返されるデバイス階層化情報は非常に複雑な場合があります。そのため LDI には、デバイスツリーやデバイスの使用状態のグラフをたどる助けになるインタフェースが用意されています。これらのインタフェースによって、デバイスツリースナップショットのコンシューマはカスタムデータポインタを、スナップショット内の異なる構造に関連付けることができます。たとえば、アプリケーションは、i ノードをたどるときに、各 i ノードに関連付けられているカスタムポインタを更新し、参照済みの i ノードにマークを付けることができます。

次に、libdevinfo(3LIB) ノードおよびリンクのマーク付けインタフェースについて説明します。

di_lnode_private_set(3DEVINFO)	指定されたデータを指定された i ノードと関連付けます。この関連付けによって、スナップショット内の i ノードをたどることができます。

di_lnode_private_get(3DEVINFO)	di_lnode_private_set(3DEVINFO) への呼び出しを通して、i ノードに関連付けられていたデータへのポインタを取得します。
di_link_private_set(3DEVINFO)	指定されたデータを指定されたリンクと関連付けます。この関連付けによって、スナップショット内のリンクをたどることができます。
di_link_private_get(3DEVINFO)	di_link_private_set(3DEVINFO) への呼び出しを通して、リンクに関連付けられていたデータへのポインタを取得します。

システム構成の出力コマンドインタフェース

prtconf(1M) コマンドは、カーネルデバイス使用状態の情報を表示するように拡張されています。デフォルトの prtconf(1M) 出力は変更されていません。デバイス使用状態情報は、prtconf(1M) コマンドとともに詳細オプション (-v) を指定したときに表示されます。特定のデバイスに関する使用状態情報は、prtconf(1M) コマンド行で、そのデバイスへのパスを指定したときに表示されます。

prtconf -v	デバイスマイナーノードとデバイス使用状態情報を表示します。カーネルコンシューマと、各カーネルコンシューマが現在開いているマイナーノードを表示します。
prtconf *path*	*path* によって指定されたデバイスのデバイス使用状態情報を表示します。
prtconf -a *path*	*path* によって指定されたデバイスのデバイス使用状態情報と、*path* の上位ノードであるすべてのデバイスノードを表示します。
prtconf -c *path*	*path* によって指定されたデバイスのデバイス使用状態情報と、*path* の子であるすべてのデバイスノードを表示します。

例 14–6 デバイス使用状態情報

特定のデバイスに関する使用状態情報が必要なときには、有効な任意のデバイスパスを *path* パラメータの値として指定できます。

```
% prtconf /dev/cfg/c0
SUNW,isptwo, instance #0
```

例 14–7 上位ノードの使用状態情報

特定のデバイスと、その特定デバイスの上位ノードになっているすべてのデバイスノードに関する使用状態情報を表示するには、prtconf(1M) コマンドに -a フラグを

例 14–7 上位ノードの使用状態情報 (続き)

指定します。上位ノードには、デバイスツリーのルートまでのすべてのノードが含まれます。prtconf(1M) コマンドに -a フラグを指定した場合は、デバイスの *path* 名も指定する必要があります。

```
% prtconf -a /dev/cfg/c0
SUNW,Sun-Fire
    ssm, instance #0
        pci, instance #0
            pci, instance #0
                SUNW,isptwo, instance #0
```

例 14–8 子ノードの使用状態情報

特定のデバイスと、その特定デバイスの子になっているすべてのデバイスノードに関する使用状態情報を表示するには、prtconf(1M) コマンドに -c フラグを指定します。prtconf(1M) コマンドに - c フラグを指定した場合は、デバイスの *path* 名も指定する必要があります。

```
% prtconf -c /dev/cfg/c0
SUNW,isptwo, instance #0
    sd (driver not attached)
    st (driver not attached)
    sd, instance #1
    sd, instance #0
    sd, instance #6
    st, instance #1 (driver not attached)
    st, instance #0 (driver not attached)
    st, instance #2 (driver not attached)
    st, instance #3 (driver not attached)
    st, instance #4 (driver not attached)
    st, instance #5 (driver not attached)
    st, instance #6 (driver not attached)
    ses, instance #0 (driver not attached)
    ...
```

例 14–9 階層化およびデバイスマイナーノードの情報 – キーボード

特定デバイスに関するデバイス階層化とデバイスマイナーノードの情報を表示するには、prtconf(1M) コマンドに -v フラグを指定します。

```
% prtconf -v /dev/kbd
conskbd, instance #0
    System properties:
        ...
    Device Layered Over:
        mod=kb8042 dev=(101,0)
            dev_path=/isa/i8042@1,60/keyboard@0
    Device Minor Nodes:
        dev=(103,0)
            dev_path=/pseudo/conskbd@0:kbd
```

例 14–9 階層化およびデバイスマイナーノードの情報 – キーボード (続き)

```
            spectype=chr type=minor
            dev_link=/dev/kbd
        dev=(103,1)
            dev_path=/pseudo/conskbd@0:conskbd
            spectype=chr type=internal
            Device Minor Layered Under:
            mod=wc accesstype=chr
                dev_path=/pseudo/wc@0
```

この例では、/dev/kbd デバイスがハードウェアキーボードデバイス (/isa/i8042@1,60/keyboard@0) の上に階層化されていることが示されています。また、/dev/kbd デバイスにはデバイスマイナーノードが 2 つあることが示されています。最初のマイナーノードには、ノードにアクセスするために使用できる /dev リンクがあります。2 つ目のマイナーノードは、ファイルシステムからはアクセスできない内部ノードです。2 つ目のマイナーノードは、ワークステーションコンソールである wc ドライバによって開かれています。この例の出力を、例 14–12 の出力と比較してください。

例 14–10 階層化およびデバイスマイナーノードの情報 – ネットワークデバイス

この例は、現在接続されているネットワークデバイスをどのデバイスが使用しているかを示しています。

```
% prtconf -v /dev/iprb0
pci1028,145, instance #0
    Hardware properties:
        ...
    Interrupt Specifications:
        ...
    Device Minor Nodes:
        dev=(27,1)
            dev_path=/pci@0,0/pci8086,244e@1e/pci1028,145@c:iprb0
                spectype=chr type=minor
                alias=/dev/iprb0
        dev=(27,4098)
            dev_path=<clone>
            Device Minor Layered Under:
                mod=udp6 accesstype=chr
                    dev_path=/pseudo/udp6@0
        dev=(27,4097)
            dev_path=<clone>
            Device Minor Layered Under:
                mod=udp accesstype=chr
                    dev_path=/pseudo/udp@0
        dev=(27,4096)
            dev_path=<clone>
            Device Minor Layered Under:
                mod=udp accesstype=chr
                    dev_path=/pseudo/udp@0
```

例 14–10 階層化およびデバイスマイナーノードの情報 – ネットワークデバイス (続き)

この例では、iprb0 デバイスが udp および udp6 の下でリンクされたことが示されています。udp と udp6 が使用しているマイナーノードへのパスは表示されていないことに注目してください。ここでパスが表示されていないのは、iprb ドライバの clone による開く操作でマイナーノードが作成され、その結果これらのノードのアクセスに使用できるファイルシステムのパスがないことが理由です。この例の出力を、例 14–11 の出力と比較してください。

デバイスユーザーコマンドインタフェース

fuser(1M) コマンドは、デバイス使用状態の情報を表示するように拡張されています。fuser(1M) コマンドは、*path* がデバイスマイナーノードを表している場合のみ、デバイス使用状態情報を表示します。-d フラグが fuser(1M) コマンドで有効なのは、デバイスマイナーノードを表す *path* を指定した場合だけです。

`fuser` *path* — *path* がデバイスマイナーノードを表している場合に、アプリケーションデバイスコンシューマとカーネルデバイスコンシューマに関する情報を表示します。

`fuser -d` *path* — *path* で表されたデバイスマイナーノードと関連付けられている、配下のデバイスのユーザーをすべて表示します。

カーネルデバイスコンシューマは次の 4 つの形式のいずれかで報告されます。カーネルデバイスコンシューマは常に角括弧で囲まれます ([])。

```
[kernel_module_name]
[kernel_module_name,dev_path=path]
[kernel_module_name,dev=(major,minor)]
[kernel_module_name,dev=(major,minor),dev_path=path]
```

fuser(1M) コマンドでファイルまたはデバイスユーザーが表示されるときの出力は、stdout でのプロセス ID と、それに続く stderr での文字から構成されます。stderr での文字は、ファイルまたはデバイスがどのように使用されているかを説明するものです。カーネルコンシューマ情報はすべて stderr に表示されます。stdout にはカーネルコンシューマ情報は表示されません。

-d フラグを使用しない場合、fuser(1M) コマンドは *path* によって指定されたデバイスマイナーノードのコンシューマについてのみ報告します。-d フラグを使用する場合、fuser(1M) コマンドは、*path* によって指定されたマイナーノードの下にあるデバイスノードのコンシューマについて報告します。次の例は、これら 2 つの場合のレポート出力における違いを示しています。

例 14–11 配下のデバイスノードのコンシューマ

ほとんどのネットワークデバイスは、デバイスが開かれたときにマイナーノードを複製します。クローンのマイナーノードのデバイス使用状態情報を要求すると、使用状態情報に、そのデバイスを使用してるプロセスはないと表示される可能性があります。代わりに、配下のデバイスノードのデバイス使用状態情報を要求すると、使用状態情報に、1 つのプロセスがそのデバイスを使用してると表示される可能性があります。この例では、デバイスの *path* のみが `fuser`(1M) コマンドに渡された場合、報告されるデバイスコンシューマはありません。`-d` フラグが使用されると、出力には、`udp` および `udp6` によってデバイスが使用されていることが表示されます。

```
% fuser /dev/iprb0
/dev/iprb0:
% fuser -d /dev/iprb0
/dev/iprb0:  [udp,dev_path=/pseudo/udp@0] [udp6,dev_path=/pseudo/udp6@0]
```

この例の出力を、例 14–10 の出力と比較してください。

例 14–12 キーボードデバイスのコンシューマ

この例では、カーネルコンシューマは `/dev/kbd` にアクセスしています。`/dev/kbd` デバイスにアクセスしているカーネルコンシューマは、ワークステーションコンソールドライバです。

```
% fuser -d /dev/kbd
/dev/kbd:  [genunix] [wc,dev_path=/pseudo/wc@0]
```

この例の出力を、例 14–9 の出力と比較してください。

パート II

特定の種類のデバイスドライバの設計

このドキュメントの第 2 部では、各ドライバタイプに固有の設計情報を提供します。

- 第 15 章「文字デバイスのドライバ」では、文字指向デバイスのドライバについて説明します。
- 第 16 章「ブロックデバイスのドライバ」では、ブロック指向デバイスのドライバについて説明します。
- 第 17 章「SCSI ターゲットドライバ」では、Sun Common SCSI Architecture (SCSA) と、SCSI ターゲットドライバの要件について概説します。
- 第 18 章「SCSI ホストバスアダプタドライバ」では、SCSA を SCSI ホストバスアダプタ (HBA) ドライバに適用する方法について説明します。
- 第 19 章「ネットワークデバイスのドライバ」では、汎用 LAN ドライバ (GLD) について説明します。GLDv3 フレームワークは、MAC プラグインと、MAC ドライバサービスのルーチンおよび構造体に対する、関数呼び出しベースのインタフェースです。
- 第 20 章「USB ドライバ」では、USBA 2.0 フレームワーク を使用してクライアント USB デバイスドライバを記述する方法について説明します。

◆ ◆ ◆ 第 15 章

15 文字デバイスのドライバ

文字デバイスには、入出力が通常はバイトストリームで実行される、テープドライブやシリアルポートなどの物理的にアドレス可能なストレージメディアはありません。この章では、文字デバイスドライバの構造について、特に文字ドライバのエントリポイントに重点を置いて説明します。さらに、この章では、同期および非同期入出力転送のコンテキストでの physio(9F) と aphysio(9F) の使用について説明します。

この章では、次の内容について説明します。

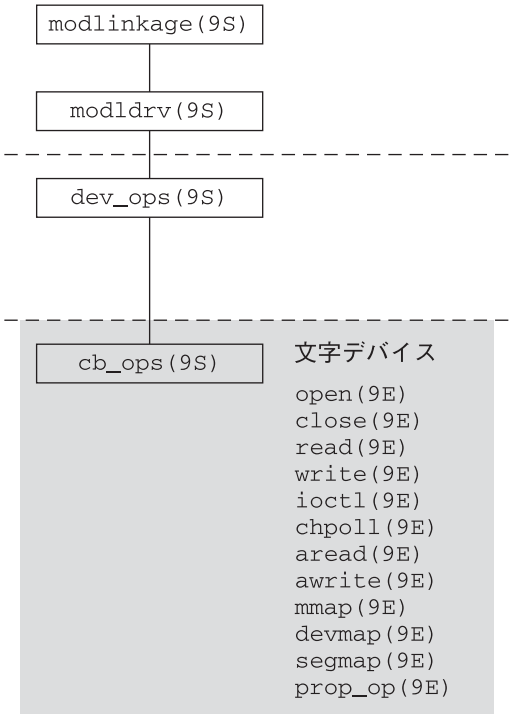
文字ドライバの構造の概要

図 15–1 は、文字デバイスドライバの構造を定義するデータ構造体とルーチンを示しています。デバイスドライバには通常、次の要素が含まれています。

- デバイスでロード可能なドライバセクション
- デバイス構成セクション
- 文字ドライバのエントリポイント

次の図の陰付きのデバイスアクセスセクションは、文字ドライバのエントリポイントを示しています。

図15–1 文字ドライバのロードマップ



各デバイスドライバに関連して dev_ops(9S) 構造体が存在し、これがさらに cb_ops(9S) 構造体を参照しています。これらの構造体には、次のドライバエントリポイントへのポインタが含まれています。

- open(9E)
- close(9E)
- read(9E)
- write(9E)
- ioctl(9E)
- chpoll(9E)
- aread(9E)
- awrite(9E)
- mmap(9E)
- devmap(9E)
- segmap(9E)
- prop_op(9E)

注 – これらのエントリポイントの一部は、必要に応じて nodev(9F) または nulldev(9F) で置き換えることができます。

文字デバイスの自動構成

attach(9E) ルーチンは、次に示すような、すべてのデバイスに必要な一般的な初期化タスクを実行します。

- インスタンスごとの状態構造体の割り当て
- デバイス割り込みの登録
- デバイスのレジスタのマッピング
- mutex 変数と条件変数の初期化
- 電源管理可能なコンポーネントの作成
- マイナーノードの作成

これらのタスクのコード例については、109 ページの「attach() エントリポイント」を参照してください。

文字デバイスドライバは、タイプ S_IFCHR のマイナーノードを作成します。S_IFCHR のマイナーノードを指定すると、このノードを表す文字型特殊ファイルが最終的に /devices 階層に表示されます。

次の例は、文字ドライバの標準的な attach(9E) ルーチンを示しています。デバイスに関連付けられているプロパティーは一般に、attach() ルーチンで宣言されます。この例では、定義済みの Size プロパティーを使用しています。Size は、ブロックデバイスでのパーティションのサイズを取得するための Nblocks プロパティーに相当します。たとえば、ディスク装置で文字入出力を実行している場合は、Size を使用してパーティションのサイズを取得できます。Size は 64 ビットのプロパティーであるため、64 ビットのプロパティーインタフェースを使用する必要があります。この場合は、ddi_prop_update_int64(9F)を使用します。プロパティーの詳細については、79 ページの「デバイスプロパティー」を参照してください。

例 15–1 文字ドライバの attach() ルーチン

```
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
  int instance = ddi_get_instance(dip);
  switch (cmd) {
  case DDI_ATTACH:
      /*
       * Allocate a state structure and initialize it.
       * Map the device's registers.
       * Add the device driver's interrupt handler(s).
       * Initialize any mutexes and condition variables.
       * Create power manageable components.
```

例 15–1 文字ドライバの attach() ルーチン (続き)

```
        *
        * Create the device's minor node. Note that the node_type
        * argument is set to DDI_NT_TAPE.
        */
        if (ddi_create_minor_node(dip, minor_name, S_IFCHR,
            instance, DDI_NT_TAPE, 0) == DDI_FAILURE) {
            /* Free resources allocated so far. */
            /* Remove any previously allocated minor nodes. */
            ddi_remove_minor_node(dip, NULL);
            return (DDI_FAILURE);
        }
       /*
        * Create driver properties like "Size." Use "Size"
        * instead of "size" to ensure the property works
        * for large bytecounts.
        */
        xsp->Size = size_of_device_in_bytes;
        maj_number = ddi_driver_major(dip);
        if (ddi_prop_update_int64(makedevice(maj_number, instance),
            dip, "Size", xsp->Size) != DDI_PROP_SUCCESS) {
            cmn_err(CE_CONT, "%s: cannot create Size property\n",
                ddi_get_name(dip));
                /* Free resources allocated so far. */
            return (DDI_FAILURE);
        }
       /* ... */
       return (DDI_SUCCESS);
case DDI_RESUME:
       /* See the "Power Management" chapter in this book. */
default:
       return (DDI_FAILURE);
  }
}
```

デバイスアクセス (文字ドライバ)

1 つ以上のアプリケーションプログラムによるデバイスへのアクセスは、open(9E) エントリポイントと close(9E) エントリポイントを通して制御されます。文字デバイスを表す特殊ファイルへの open(2) システムコールを呼び出すと、常にドライバの open(9E) ルーチンが呼び出されます。特定のマイナーデバイスの場合は、open(9E) を複数回呼び出すことができます。close(9E) ルーチンは、そのデバイスへの最後の参照が削除された場合にのみ呼び出されます。デバイスがファイル記述子を通してアクセスされている場合、close(9E) の最後の呼び出しは、close(2) または exit(2) システムコールの結果として実行されることがあります。デバイスがメモリーマッピングを通してアクセスされている場合、close(9E) の最後の呼び出しは、munmap(2) システムコールの結果として実行されることがあります。

open() エントリポイント (文字ドライバ)

open() の主な機能は、オープンの要求が許可されていることの確認です。open(9E) の構文は次のとおりです。

```
int xxopen(dev_t *devp, int flag, int otyp, cred_t *credp);
```

各表記の意味は次のとおりです。

devp デバイス番号へのポインタ。ドライバがマイナー番号を変更できるように、open() ルーチンにはポインタが渡されます。ドライバは、このポインタを使用して、デバイスのマイナーインスタンスを動的に作成できます。1 つの例として、ドライバが開かれた場合は常に新しい仮想端末を作成する疑似端末ドライバがあります。マイナー番号を動的に選択するドライバは通常、attach(9E) で ddi_create_minor_node(9F) を指定してマイナーデバイスノードを 1 つだけ作成したあと、makedevice(9F) と getmajor(9F) を使用して *devp のマイナー番号コンポーネントを変更します。

```
*devp = makedevice(getmajor(*devp), new_minor);
```

新しいマイナー番号のために ddi_create_minor_node(9F) を呼び出す必要はありません。ドライバが *devp のメジャー番号を変更してはいけません。ドライバは、使用可能なマイナー番号を内部的に常時追跡する必要があります。

flag デバイスが読み取り (FREAD)、書き込み (FWRITE)、またはその両方のために開かれているかどうかを示すビットを含むフラグ。open(2) システムコールを発行するユーザースレッドは、デバイスへの排他的アクセスを要求するか (FEXCL)、またはオープンがどのような理由でもブロックされないように指定する (FNDELAY) こともできますが、ドライバはその両方のケースを適用します。プリンタなどの書き込み専用デバイスのドライバによって、読み取りのための open(9E) が無効と見なされることがあります。

otyp open() が呼び出された方法を示す整数。ドライバは、*otyp* の値がデバイスに適していることを確認する必要があります。文字ドライバの場合、*otyp* は OTYP_CHR です (open(9E) のマニュアルページを参照)。

credp ユーザー ID やグループ ID などの、呼び出し元に関する情報を含む資格構造体へのポインタ。ドライバは、この構造体を直接検査するべきではなく、代わりに drv_priv(9F) を使用して root 権限の一般的なケースを確認します。この例では、root または PRIV_SYS_DEVICES 特権を持つユーザーのみが、書き込みのためのデバイスのオープンを許可されています。

次の例は、文字ドライバの open(9E) ルーチンを示しています。

例 15–2 文字ドライバの open(9E) ルーチン

```
static int
xxopen(dev_t *devp, int flag, int otyp, cred_t *credp)
{
    minor_t        instance;

    if (getminor(*devp)         /* if device pointer is invalid */
        return (EINVAL);
    instance = getminor(*devp); /* one-to-one example mapping */
    /* Is the instance attached? */
    if (ddi_get_soft_state(statep, instance) == NULL)
        return (ENXIO);
    /* verify that otyp is appropriate */
    if (otyp != OTYP_CHR)
        return (EINVAL);
    if ((flag & FWRITE) && drv_priv(credp) == EPERM)
        return (EPERM);
    return (0);
}
```

close() エントリポイント (文字ドライバ)

close(9E) の構文は次のとおりです。

```
int xxclose(dev_t dev, int flag, int otyp, cred_t *credp);
```

close() は、マイナーデバイスの使用を完了するために必要なクリーンアップをすべて実行し、デバイス (およびドライバ) を次回のオープンのために準備します。たとえば、open ルーチンが、排他的アクセス (FEXCL) フラグを使用して呼び出されているとします。close(9E) を呼び出すと、追加の open ルーチンを続行できるようになります。close(9E) が実行する可能性のあるほかの機能は次のとおりです。

- 戻る前の、出力バッファーからの入出力の排出の待機
- テープの巻き戻し (テープデバイス)
- 電話回線の切断 (モデムデバイス)

入出力の排出を待機するドライバは、フロー制御などの外部条件のために排出が滞ると、永久に待機する可能性があります。この問題を回避する方法については、77 ページの「スレッドがシグナルを受信できない」を参照してください。

入出力要求の処理

このセクションでは、入出力要求の処理について詳細に説明します。

ユーザーアドレス

ユーザースレッドが write(2) システムコールを発行する場合、そのスレッドはユーザー空間内のバッファーのアドレスを渡します。

```
char buffer[] = "python";
count = write(fd, buffer, strlen(buffer) + 1);
```

システムは、iovec(9S) 構造体を割り当て、iov_base フィールドを write(2) に渡されたアドレス (この場合は、buffer) に設定することによって、この転送を記述するための uio(9S) 構造体を構築します。この uio(9S) 構造体は、ドライバの write(9E) ルーチンに渡されます。uio(9S) 構造体の詳細については、311 ページの「ベクトル化された入出力」を参照してください。

iovec(9S) 内のアドレスはカーネル空間ではなく、ユーザー空間内にあります。そのため、このアドレスは現在メモリー内に存在することも、有効なアドレスであることも保証されません。いずれの場合も、デバイスドライバまたはカーネルから直接ユーザーアドレスにアクセスすると、システムがクラッシュする可能性があります。そのため、デバイスドライバは、直接ユーザーアドレスにアクセスしてはいけません。代わりに、Oracle Solaris DDI/DKI にあるデータ転送ルーチンを使用してカーネルとの間のデータ転送を行うべきです。これらのルーチンは、ページフォルトを処理できます。DDI/DKI のルーチンは、コピーを透過的に続行するための適切なユーザーページを取得できます。あるいは、無効なアクセスに対してエラーを返すこともできます。

copyout(9F) を使用すると、カーネル空間からユーザー空間にデータをコピーできます。copyin(9F) は、ユーザー空間からカーネル空間にデータをコピーできます。ddi_copyout(9F) と ddi_copyin(9F) も同様に動作しますが、ioctl(9E) ルーチンで使用されることを想定しています。copyin(9F) と copyout(9F) は、各 iovec(9S) 構造体で記述されたバッファーに対して使用できます。または、uiomove(9F) は、隣接するドライバの領域またはデバイスメモリーとの間の転送全体を実行できます。

ベクトル化された入出力

文字ドライバでは、転送は uio(9S) 構造体で記述されます。uio(9S) 構造体には、転送の方向とサイズに関する情報に加え、転送の一方の端のためのバッファーの配列が含まれています。もう一方の端はデバイスです。

uio(9S) 構造体には、次のメンバーが含まれています。

```
iovec_t       *uio_iov;       /* base address of the iovec */
                              /* buffer description array */
int           uio_iovcnt;     /* the number of iovec structures */
off_t         uio_offset;     /* 32-bit offset into file where */
                              /* data is transferred from or to */
```

```
offset_t      uio_loffset;    /* 64-bit offset into file where */
                              /* data is transferred from or to */
uio_seg_t     uio_segflg;     /* identifies the type of I/O transfer */
                              /* UIO_SYSSPACE:  kernel <-> kernel */
                              /* UIO_USERSPACE: kernel <-> user */
short         uio_fmode;      /* file mode flags (not driver setTable) */
daddr_t       uio_limit;      /* 32-bit ulimit for file (maximum */
                              /* block offset). not driver settable. */
diskaddr_t    uio_llimit;     /* 64-bit ulimit for file (maximum block */
                              /* block offset). not driver settable. */
int           uio_resid;      /* amount (in bytes) not */
                              /* transferred on completion */
```

uio(9S) 構造体は、ドライバの read(9E) エントリポイントと write(9E) エントリポイントに渡されます。この構造体は、*gather* 書き込みおよび*scatter* 読み取りと呼ばれるものをサポートするために一般化されています。デバイスに書き込む場合、書き込まれるデータバッファーがアプリケーションメモリー内で隣接している必要はありません。同様に、デバイスからメモリーに転送されるデータは隣接ストリームで受信されますが、アプリケーションメモリーの不連続領域に格納できます。scatter/gather 入出力の詳細については、readv(2)、writev(2)、pread(2)、および pwrite(2) のマニュアルページを参照してください。

各バッファーは、iovec(9S) 構造体で記述されます。この構造体には、データ領域へのポインタと転送されるバイト数が含まれています。

```
caddr_t    iov_base;    /* address of buffer */
int        iov_len;     /* amount to transfer */
```

uio 構造体には、iovec(9S) 構造体の配列へのポインタが含まれています。この配列の基底アドレスは uio_iov 内に保持されており、要素の数は uio_iovcnt 内に格納されています。

uio_offset フィールドには、アプリケーションが転送を開始する必要がある、デバイスへの 32 ビットオフセットが含まれています。 uio_loffset は、64 ビットのファイルオフセットのために使用されます。デバイスがオフセットの概念をサポートしていない場合は、これらのフィールドを安全に無視できます。ドライバは、uio_offset と uio_loffset の両方ではなく、このどちらかを解釈します。ドライバが cb_ops(9S) 構造体内の D_64BIT フラグを設定した場合、そのドライバは uio_loffset を使用します。

uio_resid フィールドは、開始時は転送されるバイト数、つまり、uio_iov 内のすべての iov_len フィールドの合計に設定されます。このフィールドは、戻る前にドライバによって、転送されなかったバイト数に設定される必要があります。read(2) システムコールと write(2) システムコールは、read(9E) エントリポイントと write(9E) エントリポイントからの戻り値を使用して、失敗した転送を判定します。失敗が発生した場合、これらのルーチンは -1 を返します。戻り値が成功を示している場合、システムコールは、要求されたバイト数から uio_resid を引いた値を返しま

す。uio_resid がドライバによって変更されない場合、read(2) と write(2) の呼び出しは 0 を返します。すべてのデータが転送されたにもかかわらず、0 の戻り値によってファイルの終わりが示されます。

サポートルーチン uiomove(9F)、physio(9F)、および aphysio(9F) は、uio(9S) 構造体を直接更新します。これらのサポートルーチンは、デバイスオフセットをデータ転送を考慮して更新します。ドライバが、位置の概念を使用するシーク可能なデバイスで使用されている場合は、uio_offset フィールドと uio_loffset フィールドのどちらも調整する必要はありません。この方法でデバイスに対して実行される入出力は、uio_offset または uio_loffset の取り得る最大値によって制約されます。このような使用法の例として、ディスク上の raw 入出力があります。

デバイスに位置の概念がない場合、ドライバは次の手順を実行できます。

1. uio_offset または uio_loffset を保存します。
2. 入出力操作を実行します。
3. uio_offset または uio_loffset をフィールドの初期値に復元します。

この方法でデバイスに対して実行される入出力は、uio_offset または uio_loffset の取り得る最大値によって制約されません。このような使用法の例として、シリアル回線上の入出力があります。

次の例は、read(9E) 関数で uio_loffset を保持するための 1 つの方法を示しています。

```
static int
xxread(dev_t dev, struct uio *uio_p, cred_t *cred_p)
{
    offset_t off;
    /* ... */
    off = uio_p->uio_loffset;  /* save the offset */
    /* do the transfer */
    uio_p->uio_loffset = off;  /* restore it */
}
```

同期入出力と非同期入出力の違い

データ転送は、同期または非同期のどちらかで行われます。この決定要因は、転送をスケジュールするエントリポイントがただちに戻るか、または入出力が完了するまで待つかのどちらであるかです。

read(9E) エントリポイントと write(9E) エントリポイントは、同期エントリポイントです。転送は、入出力が完了するまで復帰してはいけません。ルーチンから復帰すると、プロセスは、転送が成功したかどうかを認識できます。

aread(9E) エントリポイントと awrite(9E) エントリポイントは、非同期エントリポイントです。非同期エントリポイントは入出力をスケジュールし、ただちに戻りま

す。復帰すると、要求を発行したプロセスは、入出力がスケジュールされたこと、およびあとで入出力のステータスを判定する必要があることを認識できます。その間に、プロセスはほかの操作を実行できます。

カーネルへの非同期入出力要求では、その入出力が進行している間、プロセスは待機する必要がありません。プロセスは複数の入出力要求を実行することができ、カーネルでデータ転送の詳細を処理できます。非同期入出力要求を使用すると、トランザクション処理などのアプリケーションは、並行プログラミングの方法を使用してパフォーマンスや応答時間を向上させることができます。ただし、非同期入出力を使用するアプリケーションのパフォーマンス向上はすべて、プログラミングの複雑さの増加を犠牲にして得られます。

データ転送方法

データは、プログラム式入出力または DMA のどちらかを使用して転送できます。これらのデータ転送方法は、デバイスの機能に応じて、同期エントリポイントまたは非同期エントリポイントのどちらかで使用できます。

プログラム式入出力転送

プログラム式入出力デバイスは、データ転送を実行する CPU に依存します。プログラム式入出力データ転送は、デバイスレジスタに対するほかの読み取りおよび書き込み操作と同一です。デバイスメモリーの値の読み取りまたは格納のために、さまざまなデータアクセスルーチンが使用されます。

uiomove(9F) を使用すると、一部のプログラム式入出力デバイスにデータを転送できます。uiomove(9F) は、uio(9S) 構造体で定義されたユーザー空間とカーネルの間でデータを転送します。uiomove() はページフォルトを処理できるため、データの転送先のメモリーをロックダウンする必要はありません。uiomove() はまた、uio(9S) 構造体内の uio_resid フィールドも更新します。次の例は、RAM ディスクの read(9E) ルーチンを記述するための 1 つの方法を示しています。このルーチンは同期入出力を使用し、RAM ディスクの状態構造体内に次のフィールドが存在することを前提にしています。

```
caddr_t    ram;        /* base address of ramdisk */
int        ramsize;    /* size of the ramdisk */
```

例 15–3 uiomove(9F) を使用した RAM ディスクの read(9E) ルーチン

```
static int
rd_read(dev_t dev, struct uio *uiop, cred_t *credp)
{
     rd_devstate_t     *rsp;

     rsp = ddi_get_soft_state(rd_statep, getminor(dev));
     if (rsp == NULL)
```

例 15–3 uiomove(9F) を使用した RAM ディスクの read(9E) ルーチン (続き)

```
        return (ENXIO);
      if (uiop->uio_offset >= rsp->ramsize)
        return (EINVAL);
      /*
       * uiomove takes the offset into the kernel buffer,
       * the data transfer count (minimum of the requested and
       * the remaining data), the UIO_READ flag, and a pointer
       * to the uio structure.
       */
      return (uiomove(rsp->ram + uiop->uio_offset,
          min(uiop->uio_resid, rsp->ramsize - uiop->uio_offset),
          UIO_READ, uiop));
}
```

プログラム式入出力の別の例として、デバイスのメモリーに直接データを一度に 1 バイト書き込むドライバがあります。各バイトは、uio(9S) を使用して uwritec(9F) 構造体から取得されます。次に、そのバイトがデバイスに送信されます。read(9E) は、ureadc(9F) を使用して、デバイスから uio(9S) 構造体で記述された領域に 1 バイトを転送できます。

例 15–4 uwritec(9F) を使用したプログラム式入出力の write(9E) ルーチン

```
static int
xxwrite(dev_t dev, struct uio *uiop, cred_t *credp)
{
    int    value;
    struct xxstate     *xsp;

    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL)
        return (ENXIO);
    /* if the device implements a power manageable component, do this: */
    pm_busy_component(xsp->dip, 0);
    if (xsp->pm_suspended)
        pm_raise_power(xsp->dip, normal power);

    while (uiop->uio_resid > 0) {
        /*
         * do the programmed I/O access
         */
        value = uwritec(uiop);
        if (value == -1)
               return (EFAULT);
        ddi_put8(xsp->data_access_handle, &xsp->regp->data,
            (uint8_t)value);
        ddi_put8(xsp->data_access_handle, &xsp->regp->csr,
            START_TRANSFER);
        /*
         * this device requires a ten microsecond delay
         * between writes
         */
        drv_usecwait(10);
```

例 15–4 uwritec(9F) を使用したプログラム式入出力の write(9E) ルーチン (続き)

```
    }
    pm_idle_component(xsp->dip, 0);
    return (0);
}
```

DMA 転送 (同期)

文字ドライバは一般に、例 15–5 に示すように、physio(9F) を使用して read(9E) と write(9E) で DMA 転送のための設定作業を行います。

```
int physio(int (*strat)(struct buf *), struct buf *bp,
     dev_t dev, int rw, void (*mincnt)(struct buf *),
     struct uio *uio);
```

physio(9F) では、ドライバは strategy(9E) ルーチンのアドレスを指定する必要があります。physio(9F) では、メモリー空間が確実にロックダウンされます。つまり、データ転送の期間中、メモリーをページアウトすることはできません。DMA 転送ではページフォルトを処理できないため、DMA 転送にはこのロックダウンが必要です。physio(9F) ではまた、大きな転送を、より管理しやすい一連の小さな転送に分割するための自動化された方法も提供されます。詳細については、318 ページの「minphys() エントリポイント」を参照してください。

例 15–5 physio(9F) を使用した read(9E) ルーチンと write(9E) ルーチン

```
static int
xxread(dev_t dev, struct uio *uiop, cred_t *credp)
{
     struct xxstate *xsp;
     int ret;

     xsp = ddi_get_soft_state(statep, getminor(dev));
     if (xsp == NULL)
        return (ENXIO);
     ret = physio(xxstrategy, NULL, dev, B_READ, xxminphys, uiop);
     return (ret);
}

static int
xxwrite(dev_t dev, struct uio *uiop, cred_t *credp)
{
     struct xxstate *xsp;
     int ret;

     xsp = ddi_get_soft_state(statep, getminor(dev));
     if (xsp == NULL)
        return (ENXIO);
     ret = physio(xxstrategy, NULL, dev, B_WRITE, xxminphys, uiop);
     return (ret);
}
```

physio(9F) の呼び出しで、xxstrategy はドライバの strategy() ルーチンへのポインタです。buf(9S) 構造体ポインタとして NULL を渡した場合は、physio(9F) に buf(9S) 構造体を割り当てるよう指示します。ドライバが physio(9F) に buf(9S) 構造体を提供する必要がある場合は、getrbuf(9F) を使用してこの構造体を割り当てるべきです。転送が正常に完了した場合、physio(9F) は 0 を返します。失敗した場合は、エラー番号を返します。strategy(9E) を呼び出したあと、physio(9F) は、転送の完了または失敗までブロックするために biowait(9F) を呼び出します。physio(9F) の戻り値は、bioerror(9F) によって設定される buf(9S) 構造体内のエラーフィールドによって決定されます。

DMA 転送 (非同期)

aread(9E) と awrite(9E) をサポートする文字ドライバは、physio(9F) の代わりに aphysio(9F) を使用します。

```
int aphysio(int (*strat)(struct buf *), int (*cancel)(struct buf *),
     dev_t dev, int rw, void (*mincnt)(struct buf *),
     struct aio_req *aio_reqp);
```

注 – anocancel(9F) のアドレスは、現在 aphysio(9F) への 2 番目の引数として渡すことのできる唯一の値です。

---

aphysio(9F) では、ドライバは strategy(9E) ルーチンのアドレスを渡す必要があります。aphysio(9F) では、メモリー空間が確実にロックダウンされます。つまり、データ転送の期間中、メモリー空間をページアウトすることはできません。DMA 転送ではページフォルトを処理できないため、DMA 転送にはこのロックダウンが必要です。aphysio(9F) ではまた、大きな転送を、より管理しやすい一連の小さな転送に分割するための自動化された方法も提供されます。詳細については、318 ページの「minphys() エントリポイント」を参照してください。

例 15–5 と例 15–6 は、aread(9E) エントリポイントと awrite(9E) エントリポイントが、read(9E) エントリポイントと write(9E) エントリポイントと比べてほんのわずかしか違わないことを示しています。その違いは主に、physio(9F) の代わりに aphysio(9F) を使用している点にあります。

例 15–6　aphysio(9F) を使用した aread(9E) ルーチンと awrite(9E) ルーチン

```
static int
xxaread(dev_t dev, struct aio_req *aiop, cred_t *cred_p)
{
     struct xxstate *xsp;

     xsp = ddi_get_soft_state(statep, getminor(dev));
     if (xsp == NULL)
         return (ENXIO);
     return (aphysio(xxstrategy, anocancel, dev, B_READ,
```

例 15–6 aphysio(9F) を使用した aread(9E) ルーチンと awrite(9E) ルーチン (続き)

```
     xxminphys, aiop));
}

static int
xxawrite(dev_t dev, struct aio_req *aiop, cred_t *cred_p)
{
     struct xxstate *xsp;

     xsp = ddi_get_soft_state(statep, getminor(dev));
     if (xsp == NULL)
        return (ENXIO);
     return (aphysio(xxstrategy, anocancel, dev, B_WRITE,
     xxminphys,aiop));
}
```

aphysio(9F) の呼び出しで、xxstrategy() はドライバの strategy ルーチンへのポインタです。*aiop* は、aio_req(9S) 構造体へのポインタです。*aiop* は、aread(9E) と awrite(9E) に渡されます。aio_req(9S) には、データをユーザー空間内のどこに格納するかが記述されます。入出力要求が正常にスケジュールされた場合、aphysio(9F) は 0 を返します。失敗した場合は、エラー番号を返します。strategy(9E) を呼び出したあと、aphysio(9F) は、入出力の完了または失敗を待機することなく復帰します。

minphys() エントリポイント

minphys() エントリポイントは、physio(9F) または aphysio(9F) から呼び出される関数へのポインタです。*xxminphys* の目的は、要求された転送のサイズがドライバで決められた制限を超えていないことを確認することです。ユーザーがより大きな転送を要求した場合は、strategy(9E) が繰り返し呼び出され、一度に転送するサイズを決められた制限以下にするよう要求されます。DMA リソースが制限されているため、このアプローチは重要です。プリンタなどの低速なデバイスのドライバは、リソースを長時間占有しないように注意するべきです。

通常、ドライバはカーネル関数 minphys(9F) のアドレスを渡しますが、ドライバは代わりに独自の xxminphys() ルーチンを定義できます。xxminphys() の役割は、buf(9S) 構造体の b_bcount フィールドをドライバの制限以下に維持することです。ドライバは、ほかのシステム制限にも従うべきです。たとえば、ドライバの xxminphys() ルーチンは b_bcount フィールドを設定したあと戻る前に、システムの minphys(9F) ルーチンを呼び出します。

例 15–7 minphys(9F) ルーチン

```
#define XXMINVAL (512 << 10)    /* 512 KB */
static void
xxminphys(struct buf *bp)
{
       if (bp->b_bcount > XXMINVAL)
        bp->b_bcount = XXMINVAL
```

例 15–7 minphys(9F) ルーチン (続き)

```
      minphys(bp);
}
```

strategy() エントリポイント

strategy(9E) ルーチンは、ブロックドライバが元になっています。strategy 関数の名前は、ブロックデバイスへの入出力要求を効率的にキューに入れるための方針を実装することから来ています。また、文字指向のデバイスのドライバも strategy(9E) ルーチンを使用できます。ここで提供されている文字入出力モデルの場合、strategy(9E) は要求のキューを保持するのではなく、一度に 1 つの要求を処理します。

次の例では、文字指向の DMA デバイスの strategy(9E) ルーチンは、同期データ転送のための DMA リソースを割り当てます。strategy() は、デバイスレジスタをプログラミングすることによってコマンドを開始します。詳細については、第 9 章「ダイレクトメモリーアクセス (DMA)」を参照してください。

注 – strategy(9E) は、パラメータとしてデバイス番号 (dev_t) を受け取りません。代わりに、デバイス番号は、strategy(9E) に渡された buf(9S) 構造体の b_edev フィールドから取得されます。

例 15–8 strategy(9E) ルーチン

```
static int
xxstrategy(struct buf *bp)
{
     minor_t            instance;
     struct xxstate     *xsp;
     ddi_dma_cookie_t   cookie;

     instance = getminor(bp->b_edev);
     xsp = ddi_get_soft_state(statep, instance);
     /* ... */
      * If the device has power manageable components,
      * mark the device busy with pm_busy_components(9F),
      * and then ensure that the device is
      * powered up by calling pm_raise_power(9F).
      */
     /* Set up DMA resources with ddi_dma_alloc_handle(9F) and
      * ddi_dma_buf_bind_handle(9F).
      */
     xsp->bp = bp; /* remember bp */
     /* Program DMA engine and start command */
     return (0);
}
```

注 – strategy() は int を返すように宣言されますが、strategy() は常に 0 を返す必要があります。

DMA 転送が完了すると、デバイスは割り込みを生成して、割り込みルーチンが呼び出されるようにします。次の例では、xxintr() は、この割り込みを生成した可能性のあるデバイスの状態構造体へのポインタを受け取ります。

例 15–9 割り込みルーチン

```
static u_int
xxintr(caddr_t arg)
{
     struct xxstate *xsp = (struct xxstate *)arg;
     if ( /* device did not interrupt */ ) {
        return (DDI_INTR_UNCLAIMED);
     }
     if ( /* error */ ) {
        /* error handling */
     }
     /* Release any resources used in the transfer, such as DMA resources.
      * ddi_dma_unbind_handle(9F) and ddi_dma_free_handle(9F)
      * Notify threads that the transfer is complete.
      */
     biodone(xsp->bp);
     return (DDI_INTR_CLAIMED);
}
```

このドライバは、bioerror(9F) を呼び出すことによってエラーを示します。ドライバは、転送が完了したときか、または bioerror(9F) でエラーを示したあとに biodone(9F) を呼び出す必要があります。

デバイスメモリーのマッピング

フレームバッファーなどの一部のデバイスには、メモリーマッピングを介してユーザースレッドから直接アクセス可能なメモリーがあります。これらのデバイスのドライバは通常、read(9E) インタフェースや write(9E) インタフェースをサポートしていません。代わりに、これらのドライバは、devmap(9E) エントリポイントによるメモリーマッピングをサポートしています。たとえば、フレームバッファードライバは、ユーザースレッドでフレームバッファーをマップできるようにするために devmap(9E) エントリポイントを実装する可能性があります。

devmap(9E) エントリポイントは、デバイスメモリーまたはカーネルメモリーをユーザーアプリケーションにエクスポートするために呼び出されます。devmap() 関数は、segmap(9E) の内部にある devmap_setup(9F) から、または ddi_devmap_segmap(9F) に代わって呼び出されます。

segmap(9E) エントリポイントは、mmap(2) システムコールから要求されたメモリーマッピングを設定する役割を担います。多くのメモリーマッピングデバイスに対応するドライバは、独自の segmap(9E) ルーチンを定義する代わりに、ddi_devmap_segmap(9F) をエントリポイントとして使用します。

詳細については、第 10 章「デバイスメモリーおよびカーネルメモリーのマッピング」および第 11 章「デバイスコンテキスト管理」を参照してください。

ファイル記述子に対する入出力の多重化

スレッドは、時には複数のファイル記述子に対する入出力を処理することが必要になります。1 つの例として、温度検知デバイスから温度を読み取ったあと、その温度を対話型ディスプレイに報告する必要があるアプリケーションプログラムがあります。使用可能なデータがない状態で読み取り要求を行うプログラムは、ユーザーとふたたび対話する前の、温度の待機中にブロックされるべきではありません。

poll(2) システムコールはユーザーに、開かれたファイルを参照する一連のファイル記述子に対する入出力を多重化するためのメカニズムを提供します。poll(2) は、プログラムがブロックを使用せずにデータを送受信できるファイル記述子、または特定のイベントが発生したファイル記述子を識別します。

プログラムが文字ドライバをポーリングできるようにするには、そのドライバが chpoll(9E) エントリポイントを実装する必要があります。システムは、ユーザープロセスが、デバイスに関連付けられたファイル記述子に対する poll(2) システムコールを発行したときに chpoll(9E) を呼び出します。chpoll(9E) エントリポイントのルーチンは、ポーリングをサポートする必要のある STREAMS 以外の文字デバイスドライバによって使用されます。

chpoll(9E) 関数は、次の構文を使用します。

```
int xxchpoll(dev_t dev, short events, int anyyet, short *reventsp,
     struct pollhead **phpp);
```

chpoll(9E) エントリポイントでは、ドライバは次の規則に従う必要があります。

- chpoll(9E) エントリポイントが呼び出されるときは、次のアルゴリズムを実装します。

```
if ( /* events are satisfied now */ ) {
    *reventsp = mask_of_satisfied_events
} else {
    *reventsp = 0;
    if (!anyyet)
        *phpp = &local_pollhead_structure;
}
return (0);
```

チェックするイベントの説明については、chpoll(9E) のマニュアルページを参照してください。chpoll (9E) エントリポイントはこのとき、**reventsp* に戻りイベントを設定することによって、満たされたイベントのマスクを返します。

イベントが発生していない場合、そのイベントの戻りフィールドはクリアされます。*anyyet* フィールドが設定されていない場合、ドライバは pollhead 構造体のインスタンスを返す必要があります。pollhead 構造体は通常、状態構造体内に割り当てられます。この pollhead 構造体は、ドライバによって不透明として扱われます。pollhead のどのフィールドも参照するべきではありません。

- 例 15–10 に示されているタイプ events のデバイス条件が発生した場合は常に、pollwakeup(9F) を呼び出します。この関数は、一度に 1 つのイベントでのみ呼び出されます。条件が発生した場合は、割り込みルーチンで pollwakeup(9F) を呼び出すことができます。

例 15–10 と例 15–11 は、ポーリング手法の実装方法および pollwakeup(9F) の使用方法を示しています。

次の例は、POLLIN イベントと POLLERR イベントを処理する方法を示しています。ドライバはまず、デバイスの現在のステータスを判定するために、ステータスレジスタを読み取ります。パラメータ events は、ドライバがチェックする条件を指定します。該当する条件が発生した場合、ドライバは **reventsp* 内のそのビットを設定します。どの条件も発生せず、かつ *anyyet* が設定されていない場合は、pollhead 構造体のアドレスが **phpp* で返されます。

例 **15–10** chpoll(9E) ルーチン

```
static int
xxchpoll(dev_t dev, short events, int anyyet,
    short *reventsp, struct pollhead **phpp)
{
     uint8_t status;
     short revent;
     struct xxstate *xsp;

     xsp = ddi_get_soft_state(statep, getminor(dev));
     if (xsp == NULL)
         return (ENXIO);
     revent = 0;
     /*
    * Valid events are:
    * POLLIN | POLLOUT | POLLPRI | POLLHUP | POLLERR
    * This example checks only for POLLIN and POLLERR.
    */
     status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
     if ((events & POLLIN) && data available to read) {
        revent |= POLLIN;
     }
     if (status & DEVICE_ERROR) {
        revent |= POLLERR;
     }
     /* if nothing has occurred */
```

例 15–10 chpoll(9E) ルーチン (続き)

```
     if (revent == 0) {
        if (!anyyet) {
        *phpp = &xsp->pollhead;
        }
     }
       *reventsp = revent;
     return (0);
}
```

次の例は、pollwakeup(9F) 関数の使用方法を示しています。pollwakeup(9F) 関数は通常、サポートされている条件が発生したときに割り込みルーチンで呼び出されます。割り込みルーチンはステータスレジスタからステータスを読み取り、これらの条件をチェックします。次に、ポーリングスレッドにもう一度チェックすることを通知するために、イベントごとに pollwakeup(9F) を呼び出します。何らかのロックが保持された状態で pollwakeup(9F) を呼び出すと、別のルーチンが chpoll(9E) に入り、同じロックをつかもうとしてデッドロックが発生する可能性があるため、これを行うべきではないことに注意してください。

例 15–11 chpoll(9E) をサポートしている割り込みルーチン

```
static u_int
xxintr(caddr_t arg)
{
     struct xxstate *xsp = (struct xxstate *)arg;
     uint8_t    status;
     /* normal interrupt processing */
     /* ... */
     status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
     if (status & DEVICE_ERROR) {
        pollwakeup(&xsp->pollhead, POLLERR);
     }
     if ( /* just completed a read */ ) {
        pollwakeup(&xsp->pollhead, POLLIN);
     }
     /* ... */
     return (DDI_INTR_CLAIMED);
}
```

その他の入出力制御

ioctl(9E) ルーチンは、ユーザースレッドが、デバイスに関連付けられたファイル記述子に対する ioctl(2) システムコールを発行したときに呼び出されます。入出力制御メカニズムとは、デバイス固有のパラメータを取得したり、設定したりするためのものです。このメカニズムは、内部のドライバのソフトウェアフラグを設定するか、またはデバイスにコマンドを書き込むことによって、デバイス固有のモードを設定するために頻繁に使用されます。また、この制御メカニズムは、現在のデバイ

スの状態に関する情報をユーザーに返すためにも使用されます。つまり、この制御メカニズムは、アプリケーションやドライバで実行する必要のあるすべてのことを実行できます。

ioctl() エントリポイント (文字ドライバ)

```
int xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
     cred_t *credp, int *rvalp);
```

cmd パラメータは、ioctl(9E) がどのコマンドを実行するかを示します。慣例により、入出力制御コマンドが関連付けられているドライバは、そのコマンドのビット 8 から 15 で示されます。通常は、文字の ASCII コードがドライバを表します。ドライバ固有のコマンドはビット 0 から 7 にあります。次の例には、いくつかの入出力コマンドの作成が示されています。

```
#define XXIOC    ('x' << 8)     /* 'x' is a character that represents device xx */
#define XX_GET_STATUS    (XXIOC | 1)  /* get status register */
#define XX_SET_CMD       (XXIOC | 2)  /* send command */
```

arg の解釈は、コマンドによって異なります。入出力制御コマンドは、ドライバのドキュメントまたはマニュアルページで説明されています。コマンドはまた、アプリケーションがコマンドの名前、コマンドが何を実行するか、およびコマンドが *arg* として何を受け入れるか、または返すかを判定できるように、公開ヘッダーファイルでも定義されます。ドライバとの間の *arg* のデータ転送はすべて、ドライバによって実行される必要があります。

フレームバッファーやディスクなどの特定のクラスのデバイスは、標準的な一連の入出力制御要求をサポートする必要があります。これらの標準の入出力制御インタフェースは、Solaris 8 のリファレンスマニュアルコレクションで説明されています。たとえば、fbio(7I) ではフレームバッファーがサポートする必要のある入出力制御について、また dkio(7I) では標準のディスク入出力制御について説明しています。入出力制御の詳細については、323 ページの「その他の入出力制御」を参照してください。

ドライバは、*arg* のデータをユーザーレベルのアプリケーションからカーネルレベルに転送するために ddi_copyin(9F) を使用する必要があります。ドライバは、カーネルレベルからユーザーレベルにデータを転送するために ddi_copyout(9F) を使用する必要があります。ddi_copyin(9F) または ddi_copyout(9F) を使用しないと、2 つの条件でパニックが発生することがあります。アーキテクチャーによってカーネルとユーザーのアドレス空間が分離されている場合、またはユーザーアドレスがスワップアウトされている場合はパニックが発生します。

ioctl(9E) は通常、サポートされている各 ioctl(9E) 要求に対するケースを含む switch 文です。

例 15–12 ioctl(9E) ルーチン

```
static int
xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
    cred_t *credp, int *rvalp)
{
     uint8_t        csr;
     struct xxstate     *xsp;

     xsp = ddi_get_soft_state(statep, getminor(dev));
     if (xsp == NULL) {
        return (ENXIO);
     }
     switch (cmd) {
     case XX_GET_STATUS:
       csr = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
       if (ddi_copyout(&csr, (void *)arg,
           sizeof (uint8_t), mode) != 0) {
           return (EFAULT);
       }
       break;
     case XX_SET_CMD:
       if (ddi_copyin((void *)arg, &csr,
         sizeof (uint8_t), mode) != 0) {
         return (EFAULT);
       }
       ddi_put8(xsp->data_access_handle, &xsp->regp->csr, csr);
       break;
     default:
       /* generic "ioctl unknown" error */
       return (ENOTTY);
     }
     return (0);
}
```

cmd 変数は、特定のデバイス制御操作を識別します。*arg* にユーザー仮想アドレスが含まれていると、問題が発生することがあります。ioctl(9E) は、*arg* によって示されたアプリケーションプログラム内のデータ構造体とドライバの間でデータを転送するために ddi_copyin(9F) または ddi_copyout(9F) を呼び出す必要があります。例 15–12 では、XX_GET_STATUS 要求のケースに対して、*xsp->regp->csr* の内容が *arg* 内のアドレスにコピーされます。ioctl(9E) は、成功した要求を行った ioctl(2) システムコールへの戻り値として任意の整数値を **rvalp* 内に格納できます。負の戻り値 (-1 など) は避けるべきです。多くのアプリケーションプログラムは、負の値が失敗を示すことを前提にしています。

次の例は、前の段落で説明した入出力制御を使用するアプリケーションを示しています。

例 15–13 ioctl(9E) の使用

```
#include <sys/types.h>
#include "xxio.h"     /* contains device's ioctl cmds and args */
int
```

例 15–13 ioctl(9E) の使用 (続き)

```
main(void)
{
     uint8_t    status;
     /* ... */
     /*
      * read the device status
      */
     if (ioctl(fd, XX_GET_STATUS, &status) == -1) {
         /* error handling */
     }
     printf("device status %x\n", status);
     exit(0);
}
```

64 ビットに対応したデバイスドライバに対する入出力制御のサポート

Oracle Solaris カーネルは、32 ビットアプリケーションと 64 ビットアプリケーションの両方をサポートしている適切なハードウェア上では 64 ビットモードで動作します。両方のサイズのプログラムからの入出力制御コマンドをサポートするには、64 ビットのデバイスドライバが必要です。32 ビットプログラムと 64 ビットプログラムの違いは、C 言語の型モデルです。32 ビットプログラムは ILP32 であり、64 ビットプログラムは LP64 です。C データ型モデルについては、付録 C 「64 ビットデバイスドライバの準備」を参照してください。

プログラムとカーネルの間を流れるデータの形式が同じでない場合は、ドライバがモデルの不一致に対処できる必要があります。モデルの不一致に対処するには、データに対して適切な調整を行う必要があります。

モデルの不一致が存在するかどうかを判定するために、ioctl(9E) のモードパラメータは、ドライバにデータモデルのビットを渡します。例 15–14 に示すように、このモードパラメータは次に、何らかのモデル変換が必要かどうかを判定するために ddi_model_convert_from(9F) に渡されます。

ioctl(9E) ルーチンにデータモデルを渡すために、モード引数のフラグサブフィールドが使用されます。このフラグは、次のいずれかの値に設定されます。

- DATAMODEL_ILP32
- DATAMODEL_LP64

FNATIVE は、カーネル実装のデータモデルに一致するように条件付きで定義されます。*mode* 引数からフラグを抽出するには、FMODELS マスクを使用します。これにより、ドライバは、明示的にデータモデルを検査してアプリケーションのデータ構造体をコピーする方法を判定できます。

DDI 関数の ddi_model_convert_from(9F) は、ioctl() 呼び出しに関して一部のドライバを支援できる簡易ルーチンです。この関数は引数としてユーザーアプリケーションのデータ型モデルを受け取り、次のいずれかの値を返します。

- DDI_MODEL_ILP32 – ILP32 アプリケーションから変換する
- DDI_MODEL_NONE – 変換は必要なし

アプリケーションとドライバのデータモデルが同じである場合のように、データ変換が必要ない場合は DDI_MODEL_NONE が返されます。LP64 モデルにコンパイルされていて、32 ビットアプリケーションと通信するドライバには DDI_MODEL_ILP32 が返されます。

次の例では、ドライバは、ユーザーアドレスを含むデータ構造体をコピーします。このデータ構造体は、ILP32 から LP64 にサイズが変更されます。それに応じて、64 ビットのドライバは、32 ビットアプリケーションとの通信には構造体の 32 ビットバージョンを使用します。

例 15–14 32 ビットアプリケーションと 64 ビットアプリケーションをサポートする ioctl(9E) ルーチン

```
struct args32 {
    uint32_t    addr;    /* 32-bit address in LP64 */
    int     len;
}
struct args {
    caddr_t     addr;    /* 64-bit address in LP64 */
    int     len;
}

static int
xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
    cred_t *credp, int *rvalp)
{
    struct  xxstate  *xsp;
    struct  args     a;
    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL) {
        return (ENXIO);
    }
    switch (cmd) {
    case XX_COPYIN_DATA:
        switch(ddi_model_convert_from(mode)) {
        case DDI_MODEL_ILP32:
        {
            struct args32 a32;

            /* copy 32-bit args data shape */
            if (ddi_copyin((void *)arg, &a32,
                sizeof (struct args32), mode) != 0) {
                return (EFAULT);
            }
            /* convert 32-bit to 64-bit args data shape */
            a.addr = a32.addr;
            a.len = a32.len;
```

例 15–14　32 ビットアプリケーションと 64 ビットアプリケーションをサポートする ioctl(9E) ルーチン　*(続き)*

```
            break;
        }
        case DDI_MODEL_NONE:
            /* application and driver have same data model. */
            if (ddi_copyin((void *)arg, &a, sizeof (struct args),
                mode) != 0) {
                return (EFAULT);
            }
        }
        /* continue using data shape in native driver data model. */
        break;

    case XX_COPYOUT_DATA:
        /* copyout handling */
        break;
    default:
        /* generic "ioctl unknown" error */
        return (ENOTTY);
    }
    return (0);
}
```

copyout() のオーバーフローの処理

ドライバでは、32 ビットのサイズの構造体には収まらないネイティブな量のコピー出力が必要になる場合があります。この場合、ドライバは、呼び出し元に EOVERFLOW を返します。次の例に示すように、EOVERFLOW は、返される値を保持するにはインタフェース内のデータ型が小さすぎることを示すものとして機能します。

例 15–15　copyout(9F) のオーバーフローの処理

```
int
    xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
     cred_t *cr, int *rval_p)
    {
        struct resdata res;
        /* body of driver */
        switch (ddi_model_convert_from(mode & FMODELS)) {
        case DDI_MODEL_ILP32: {
            struct resdata32 res32;

            if (res.size > UINT_MAX)
                    return (EOVERFLOW);
            res32.size = (size32_t)res.size;
            res32.flag = res.flag;
            if (ddi_copyout(&res32,
                (void *)arg, sizeof (res32), mode))
                    return (EFAULT);
        }
        break;
```

例 15–15 copyout(9F) のオーバーフローの処理 (続き)

```
        case DDI_MODEL_NONE:
            if (ddi_copyout(&res, (void *)arg, sizeof (res), mode))
                    return (EFAULT);
            break;
        }
        return (0);
    }
```

32 ビットと 64 ビットのデータ構造体マクロ

例 15–15 の方法は、多くのドライバで正常に機能します。この代わりに、<sys/model.h> で提供されているデータ構造体マクロを使用して、アプリケーションとカーネルの間でデータを移動する方法があります。これらのマクロによってコードが整理され、機能の点から見てまったく同様に動作するようになります。

例 15–16 データ構造体マクロを使用したデータの移動

```
int
    xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
        cred_t *cr, int *rval_p)
    {
        STRUCT_DECL(opdata, op);

        if (cmd != OPONE)
            return (ENOTTY);

        STRUCT_INIT(op, mode);

        if (copyin((void *)arg,
            STRUCT_BUF(op), STRUCT_SIZE(op)))
                return (EFAULT);

        if (STRUCT_FGET(op, flag) != XXACTIVE ||
            STRUCT_FGET(op, size) > XXSIZE)
                return (EINVAL);
        xxdowork(device_state, STRUCT_FGET(op, size));
        return (0);
}
```

構造体マクロの動作のしくみ

64 ビットのデバイスドライバでは、構造体マクロを使用すると、両方のサイズのデータ構造体でカーネルメモリーの同じ部分を使用できます。メモリーバッファーには、データ構造体のネイティブ形式 (つまり、LP64 形式と ILP32 形式) の内容が保持されます。構造体への各アクセスは、条件式によって実装されま

す。32 ビットのドライバとしてコンパイルされた場合は、1 つのデータモデル (ネイティブ形式) のみがサポートされます。条件式は使用されません。

64 ビットバージョンのマクロは、データ構造体のシャドウバージョンの定義に依存します。シャドウバージョンには、固定幅の型を含む 32 ビットインタフェースが記述されます。シャドウデータ構造体の名前は、ネイティブなデータ構造体の名前に「32」を追加することによって作成されます。将来の保守コストを軽減するために、便宜上、シャドウ構造体の定義はネイティブな構造体と同じファイル内に置きます。

これらのマクロは、次の引数を取ることができます。

structname `struct` キーワードのあとに入力されたデータ構造体のネイティブ形式の構造体名。

umodel `ioctl(9E)` のモードパラメータから抽出されたユーザーのデータモデル (`FILP32` や `FLP64` など) を含むフラグワード。

handle これらのマクロで操作される構造体の特定のインスタンスを参照するために使用される名前。

fieldname 構造体内のフィールドの名前。

構造体マクロを使用する場合

マクロを使用すると、あるデータ項目のフィールドのみへのインプレース参照を行うことができます。マクロでは、データモデルに基づいた別のコードパスを取るための方法は提供されません。データ構造体内のフィールドの数が多い場合は、マクロを避けるべきです。また、これらのフィールドを参照する頻度が高い場合も、マクロを避けるべきです。

マクロでは、データモデル間の違いの多くがマクロの実装内に覆い隠されます。その結果、このインタフェースを使用して記述されたコードは一般に読みやすくなります。32 ビットのドライバとしてコンパイルされた場合、結果のコードは煩わしい `#ifdefs` が必要ないため簡潔になりますが、データ型のチェックは引き続き保持されます。

構造体ハンドルの宣言と初期化

`STRUCT_DECL(9F)` と `STRUCT_INIT(9F)` を使用すると、スタック上の `ioctl` をデコードするためのハンドルと領域を宣言して初期化できます。`STRUCT_HANDLE(9F)` と `STRUCT_SET_HANDLE(9F)` は、スタック上の領域を割り当てることなくハンドルを宣言して初期化します。構造体が非常に大きいか、またはほかのデータ構造体に含まれている場合は、後者のマクロが役立つことがあります。

注 – STRUCT_DECL(9F) マクロと STRUCT_HANDLE(9F) マクロはデータ構造体の宣言まで拡張されるため、これらのマクロは C コードでこのような宣言を使用してグループ化されます。

構造体を宣言して初期化するためのマクロは次のとおりです。

STRUCT_DECL(*structname* , *handle*)
structname データ構造体のための、handle という名前の構造体ハンドルを宣言します。STRUCT_DECL は、スタック上のネイティブ形式のための領域を割り当てます。ネイティブ形式は、構造体の ILP32 形式より大きいか、または等しいと見なされます。

STRUCT_INIT(*handle* , *umodel*)
handle のデータモデルを umodel に初期化します。このマクロは、STRUCT_DECL(9F) を使用して宣言された構造体ハンドルへの何らかのアクセスが行われる前に呼び出す必要があります。

STRUCT_HANDLE(*structname* , *handle*)
handle という名前の構造体ハンドルを宣言します。STRUCT_DECL(9F) と対比されます。

STRUCT_SET_HANDLE(*handle* , *umodel*, *addr*)
handle のデータモデルを *umodel* に初期化し、*addr* を以降の操作に使用されるバッファーとして設定します。このマクロは、STRUCT_DECL(9F) を使用して宣言された構造体ハンドルへのアクセスの前に呼び出します。

構造体ハンドルに対する操作

構造体に対する操作を実行するためのマクロは次のとおりです。

size_t STRUCT_SIZE(*handle*)
handle によって参照される構造体のサイズを、その組み込みのデータモデルに応じて返します。

typeof *fieldname* STRUCT_FGET(*handle*, *fieldname*)
handle によって参照されるデータ構造体内の示されているフィールドを返します。このフィールドはポインタ以外の型です。

typeof *fieldname* STRUCT_FGETP(*handle*, *fieldname*)
handle によって参照されるデータ構造体内の示されているフィールドを返します。このフィールドはポインタ型です。

STRUCT_FSET(*handle* , *fieldname*, *val*)
handle によって参照されるデータ構造体内の示されているフィールドを値 *val* に設定します。*val* の型は、*fieldname* の型に一致します。このフィールドはポインタ以外の型です。

STRUCT_FSETP(*handle* , *fieldname*, *val*)
handle によって参照されるデータ構造体内の示されているフィールドを値 *val* に設定します。このフィールドはポインタ型です。

typeof *fieldname* *STRUCT_FADDR(*handle*, *fieldname*)
handle によって参照されるデータ構造体内の示されているフィールドのアドレスを返します。

struct structname *STRUCT_BUF(*handle*)
handle で記述されたネイティブな構造体へのポインタを返します。

その他の操作

その他の構造体マクロのいくつかを次に示します。

size_t SIZEOF_STRUCT(*struct_name* , *datamodel*)
指定されたデータモデルに基づいた *struct_name* のサイズを返します。

size_t SIZEOF_PTR(*datamodel*)
指定されたデータモデルに基づいたポインタのサイズを返します。

◆ ◆ ◆ 第 16 章

16 ブロックデバイスのドライバ

この章では、ブロックデバイスドライバの構造について説明します。カーネルはブロックデバイスを、一連のランダムにアクセス可能な論理ブロックとして認識します。ファイルシステムは、buf(9S) 構造体のリストを使用して、ブロックデバイスとユーザー空間の間にあるデータブロックをバッファリングします。ファイルシステムをサポートできるのはブロックデバイスのみです。

この章では、次の内容について説明します。

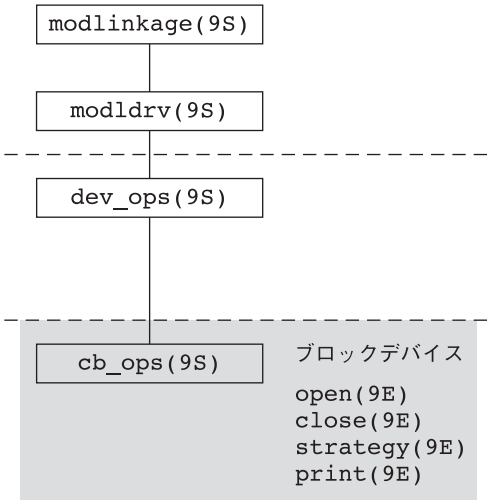
ブロックドライバの構造の概要

図 16–1 は、ブロックデバイスドライバの構造を定義するデータ構造体とルーチンを示しています。デバイスドライバには通常、次の要素が含まれています。

- デバイスでロード可能なドライバセクション
- デバイス構成セクション
- デバイスアクセスセクション

次の図の陰付きのデバイスアクセスセクションは、ブロックドライバのエントリポイントを示しています。

図16–1 ブロックドライバのロードマップ



各デバイスドライバに関連して dev_ops(9S) 構造体が存在し、これがさらに cb_ops(9S) 構造体を参照しています。ドライバのデータ構造体の詳細については、第6章「ドライバの自動構成」を参照してください。

ブロックデバイスドライバは、次のエントリポイントを提供します。

- open(9E)
- close(9E)
- strategy(9E)
- print(9E)

注 – 一部のエントリポイントは、必要に応じて nodev(9F) または nulldev(9F) で置き換えることができます。

ファイル入出力

ファイルシステムは、ディレクトリとファイルがツリー状に階層化されたものです。UNIX ファイルシステム (UFS) などの一部のファイルシステムは、ブロック指向のデバイス上に存在します。ファイルシステムは、format(1M) と newfs(1M) によって作成されます。

アプリケーションが UFS ファイルシステム上の通常ファイルに read(2) または write(2) システムコールを発行したとき、ファイルシステムは、そのファイルシステムが存在するブロックデバイスに対するデバイスドライバの strategy(9E) エントリ

ポイントを呼び出すことができます。ファイルシステムのコードは、1 回の read(2) または write(2) システムコールに対して strategy(9E) を複数回呼び出すことができます。

ファイルシステムのコードは、通常ファイルのブロックごとに論理デバイスアドレス、つまり論理ブロック番号を決定します。その後、ブロック入出力要求が、ブロックデバイスを宛先とする buf(9S) 構造体の形式で作成されます。次に、ドライバの strategy(9E) エントリポイントが buf(9S) 構造体を解釈し、要求を完了します。

ブロックデバイスの自動構成

attach(9E) は、デバイスのインスタンスごとに、次の一般的な初期化タスクを実行します。

- インスタンスごとの状態構造体の割り当て
- デバイスのレジスタのマッピング
- デバイス割り込みの登録
- mutex と条件変数の初期化
- 電源管理可能なコンポーネントの作成
- マイナーノードの作成

ブロックデバイスドライバは、タイプ S_IFBLK のマイナーノードを作成します。その結果、このノードを表すブロック型特殊ファイルが /devices 階層に表示されます。

ブロックデバイスの論理デバイス名は /dev/dsk ディレクトリに表示され、コントローラ番号、バスアドレス番号、ディスク番号、およびスライス番号で構成されています。ノードタイプが DDI_NT_BLOCK または DDI_NT_BLOCK_CHAN に設定されている場合、これらの名前は devfsadm(1M) プログラムによって作成されます。そのデバイスがチャネル、つまり、追加のレベルのアドレス指定能力を備えたバス上で通信する場合は、DDI_NT_BLOCK_CHAN を指定します。SCSI ディスクがその良い例です。DDI_NT_BLOCK_CHAN を指定すると、論理名にバスアドレスフィールド (t*N*) が表示されます。ほかのほとんどのデバイスには DDI_NT_BLOCK を使用するべきです。

マイナーデバイスは、ディスク上のパーティションを参照します。ドライバは、マイナーデバイスごとに nblocks または Nblocks プロパティーを作成する必要があります。この整数プロパティーは、DEV_BSIZE (つまり、512 バイト) の単位で表された、マイナーデバイスでサポートされているブロック数を指定します。ファイルシステムは、nblocks プロパティーと Nblocks プロパティーを使用してデバイスの制限を判定します。Nblocks は、nblocks の 64 ビットバージョンです。Nblocks は、ディスクあたり 1T バイトを超える記憶領域を保持できるストレージデバイスに使用するべきです。詳細については、79 ページの「デバイスプロパティー」を参照してください。

例 16–1 は、デバイスのマイナーノードと Nblocks プロパティーの作成に重点を置いて、標準的な attach(9E) エントリポイントを示したものです。この例では nblocks

ではなく Nblocks を使用しているため、ddi_prop_update_int(9F) の代わりに ddi_prop_update_int64(9F) が呼び出されることに注意してください。

参考までに、この例は makedevice(9F) を使用して ddi_prop_update_int64() のデバイス番号を作成する方法を示しています。makedevice 関数は、dev_info_t 構造体へのポインタからメジャー番号を生成する ddi_driver_major(9F) を使用します。ddi_driver_major() の使用は、dev_t 構造体ポインタを取得する getmajor(9F) の使用に似ています。

例 16–1 ブロックドライバの attach() ルーチン

```
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
     int instance = ddi_get_instance(dip);
     switch (cmd) {
       case DDI_ATTACH:
       /*
        * allocate a state structure and initialize it
        * map the devices registers
        * add the device driver's interrupt handler(s)
        * initialize any mutexes and condition variables
        * read label information if the device is a disk
        * create power manageable components
        *
        * Create the device minor node. Note that the node_type
        * argument is set to DDI_NT_BLOCK.
        */
       if (ddi_create_minor_node(dip, "minor_name", S_IFBLK,
          instance, DDI_NT_BLOCK, 0) == DDI_FAILURE) {
          /* free resources allocated so far */
          /* Remove any previously allocated minor nodes */
          ddi_remove_minor_node(dip, NULL);
          return (DDI_FAILURE);
        }
       /*
        * Create driver properties like "Nblocks". If the device
        * is a disk, the Nblocks property is usually calculated from
        * information in the disk label.  Use "Nblocks" instead of
        * "nblocks" to ensure the property works for large disks.
        */
       xsp->Nblocks = size;
       /* size is the size of the device in 512 byte blocks */
       maj_number = ddi_driver_major(dip);
       if (ddi_prop_update_int64(makedevice(maj_number, instance), dip,
          "Nblocks", xsp->Nblocks) != DDI_PROP_SUCCESS) {
          cmn_err(CE_CONT, "%s: cannot create Nblocks property\n",
               ddi_get_name(dip));
         /* free resources allocated so far */
         return (DDI_FAILURE);
       }
       xsp->open = 0;
       xsp->nlayered = 0;
       /* ... */
       return (DDI_SUCCESS);
```

例 16–1 ブロックドライバの attach() ルーチン (続き)

```
    case DDI_RESUME:
       /* For information, see Chapter 12, "Power Management," in this book. */
       default:
          return (DDI_FAILURE);
     }
}
```

デバイスアクセスの制御

このセクションでは、ブロックデバイスドライバ内の open() 関数と close() 関数のエントリポイントについて説明します。open(9E) と close(9E) の詳細については、第 15 章「文字デバイスのドライバ」を参照してください。

open() エントリポイント (ブロックドライバ)

open(9E) エントリポイントは、指定されたデバイスへのアクセスを取得するために使用されます。ブロックドライバの open(9E) ルーチンは、ユーザースレッドがマイナーデバイスに関連付けられたブロック型特殊ファイルに対して open(2) または mount(2) システムコールを発行したとき、または階層化ドライバが open(9E) を呼び出したときに呼び出されます。詳細については、334 ページの「ファイル入出力」を参照してください。

open() エントリポイントは、次の条件を確認します。

- デバイスを開くことができる。つまり、デバイスがオンラインで、準備ができている。
- 要求どおりにデバイスを開くことができる。デバイスがこの操作をサポートしている。デバイスの現在の状態が要求と競合していない。
- 呼び出し元がデバイスを開くためのアクセス権を持っている。

次の例は、ブロックドライバの open(9E) エントリポイントを示しています。

例 16–2 ブロックドライバの open(9E) ルーチン

```
static int
xxopen(dev_t *devp, int flags, int otyp, cred_t *credp)
{
       minor_t         instance;
       struct xxstate        *xsp;

     instance = getminor(*devp);
     xsp = ddi_get_soft_state(statep, instance);
     if (xsp == NULL)
           return (ENXIO);
     mutex_enter(&xsp->mu);
```

例 16–2 ブロックドライバの open(9E) ルーチン (続き)

```
     /*
    * only honor FEXCL. If a regular open or a layered open
    * is still outstanding on the device, the exclusive open
    * must fail.
    */
     if ((flags & FEXCL) && (xsp->open || xsp->nlayered)) {
       mutex_exit(&xsp->mu);
       return (EAGAIN);
     }
     switch (otyp) {
       case OTYP_LYR:
         xsp->nlayered++;
         break;
      case OTYP_BLK:
         xsp->open = 1;
         break;
     default:
         mutex_exit(&xsp->mu);
         return (EINVAL);
     }
   mutex_exit(&xsp->mu);
      return (0);
}
```

otyp 引数は、デバイスに対するオープンのタイプを指定するために使用されます。OTYP_BLK は、ブロックデバイスの標準的なオープンのタイプです。otyp が OTYP_BLK に設定されている場合は、デバイスを複数回開くことができます。close(9E) は、デバイスに対するタイプ OTYP_BLK の最後のクローズが実行されるときに 1 回だけ呼び出されます。デバイスが階層化デバイスとして使用されている場合、otyp は OTYP_LYR に設定されます。階層化ドライバは、タイプ OTYP_LYR のすべてのオープンに対してタイプ OTYP_LYR の対応するクローズを発行します。この例ではオープンの各タイプを常時監視しているため、ドライバはデバイスがいつ close(9E) で使用されていないかを判定できます。

close() エントリポイント (ブロックドライバ)

close(9E) エントリポイントは、1 つの例外を除いて open(9E) と同じ引数を使用します。dev はデバイス番号であり、デバイス番号へのポインタではありません。

close() ルーチンは、open(9E) エントリポイントについての説明と同じ方法で otyp を確認します。次の例では、close() は、実際にデバイスをいつ閉じることができるかを判定する必要があります。閉じる操作は、ブロックのオープンと階層化されたオープンの数によって影響を受けます。

例 16–3 ブロックデバイスの close(9E) ルーチン

```
static int
xxclose(dev_t dev, int flag, int otyp, cred_t *credp)
{
```

例 16–3 ブロックデバイスの close(9E) ルーチン (続き)

```
    minor_t instance;
    struct xxstate *xsp;

    instance = getminor(dev);
    xsp = ddi_get_soft_state(statep, instance);
      if (xsp == NULL)
         return (ENXIO);
    mutex_enter(&xsp->mu);
    switch (otyp) {
      case OTYP_LYR:
      xsp->nlayered--;
      break;
     case OTYP_BLK:
      xsp->open = 0;
      break;
    default:
      mutex_exit(&xsp->mu);
      return (EINVAL);
      }

    if (xsp->open || xsp->nlayered) {
      /* not done yet */
      mutex_exit(&xsp->mu);
      return (0);
    }
      /* cleanup (rewind tape, free memory, etc.) */
  /* wait for I/O to drain */
    mutex_exit(&xsp->mu);

    return (0);
}
```

strategy() エントリポイント

strategy(9E) エントリポイントは、ブロックデバイスとの間でのデータバッファーの読み取りと書き込みのために使用されます。*strategy* という名前は、このエントリポイントによって、デバイスへの要求を順序付けるための何らかの最適な方針が実装される可能性がある点を指しています。

strategy(9E) は、一度に 1 つの要求を処理する、つまり、同期転送を行うように記述できます。strategy() はまた、非同期転送のように、デバイスへの複数の要求をキューに入れるように記述することもできます。これらの方法を選択する場合は、デバイスの機能や制限事項を考慮に入れるべきです。

strategy(9E) ルーチンには、buf(9S) 構造体へのポインタが渡されます。この構造体には転送要求を記述しますが、復帰したときにはステータス情報が含まれています。buf(9S) と strategy(9E) が、ブロックデバイス操作の焦点です。

buf 構造体

ブロックドライバには、次の buf 構造体メンバーが重要です。

```
int             b_flags;        /* Buffer status */
struct buf      *av_forw;       /* Driver work list link */
struct buf      *av_back;       /* Driver work list link */
size_t          b_bcount;       /* # of bytes to transfer */
union {
    caddr_t     b_addr;         /* Buffer's virtual address */
} b_un;
daddr_t         b_blkno;        /* Block number on device */
diskaddr_t      b_lblkno;       /* Expanded block number on device */
size_t          b_resid;        /* # of bytes not transferred after error */
int             b_error;        /* Expanded error field */
void            *b_private;     /* "opaque" driver private area */
dev_t           b_edev;         /* expanded dev field */
```

各表記の意味は次のとおりです。

av_forw と av_back	ドライバでバッファーのリストを管理するためにドライバが使用できるポインタ。av_forw ポインタと av_back ポインタの説明については、345 ページの「非同期データ転送 (ブロックドライバ)」を参照してください。
b_bcount	デバイスによって転送されるバイト数を指定します。
b_un.b_addr	データバッファーのカーネル仮想アドレス。bp_mapin(9F) の呼び出しのあとでのみ有効です。
b_blkno	512 バイトの DEV_BSIZE の単位で表された、データ転送用にデバイス上に存在する、先頭の 32 ビット論理ブロック番号。ドライバは、b_blkno と b_lblkno の両方ではなく、このどちらかを使用します。
b_lblkno	512 バイトの DEV_BSIZE の単位で表された、データ転送用にデバイス上に存在する、先頭の 64 ビット論理ブロック番号。ドライバは、b_blkno と b_lblkno の両方ではなく、このどちらかを使用します。
b_resid	エラーのために転送されなかったバイト数を示すためにドライバによって設定されます。b_resid の設定の例については、例 16–7 を参照してください。b_resid メンバーはオーバーロードされます。b_resid はまた、disksort(9F) によっても使用されます。
b_error	転送エラーが発生した場合、ドライバによってエラー番号が設定されます。b_error は、b_flags の B_ERROR ビットとともに設定されます。エラー値の詳細については、Intro(9E) のマニュアルページを参照してください。ドライバは、b_error を直接設定するのではなく、bioerror(9F) を使用します。

b_flags	buf 構造体のステータスと転送の属性を含むフラグ。B_READ が設定されている場合、buf 構造体は、デバイスからメモリーへの転送を示します。それ以外の場合、この構造体はメモリーからデバイスへの転送を示します。データ転送中にドライバでエラーが発生した場合、ドライバは、b_flags メンバーの B_ERROR フィールドを設定します。さらに、ドライバは、b_error でより具体的なエラー値を提供します。ドライバは、B_ERROR を設定するのではなく、bioerror(9F) を使用します。

注意 – ドライバが b_flags をクリアしないようにしてください。

b_private	ドライバの非公開データを格納するために、ドライバによって排他的に使用されます。
b_edev	転送で使用されたデバイスのデバイス番号が含まれています。

bp_mapin 構造体

デバイスドライバの strategy(9E) ルーチンには、buf 構造体ポインタを渡すことができます。ただし、b_un.b_addr によって参照されるデータバッファーは、必ずしもカーネルのアドレス空間内にマッピングされるわけではありません。そのため、ドライバはデータに直接アクセスできません。ほとんどのブロック指向のデバイスは DMA 機能を備えているため、データバッファーに直接アクセスする必要はありません。代わりに、これらのデバイスは DMA マッピングルーチンを使用して、そのデバイスの DMA エンジンがデータ転送を実行できるようにします。DMA の使用の詳細については、第 9 章「ダイレクトメモリーアクセス (DMA)」を参照してください。

ドライバがデータバッファーに直接アクセスする必要がある場合、そのドライバはまず、bp_mapin(9F) を使用してそのバッファーをカーネルのアドレス空間内にマッピングする必要があります。ドライバがそのデータに直接アクセスする必要がなくなった場合は、bp_mapout(9F) を使用するべきです。

注意 – bp_mapout(9F) は、そのデバイスドライバによって割り当てられ、かつ所有されているバッファーに対してのみ呼び出すべきです。ファイルシステムなどの、strategy(9E) エントリポイントを介してドライバに渡されたバッファーに対しては bp_mapout() を呼び出してはいけません。bp_mapin(9F) は参照カウントを保持しません。bp_mapout(9F) は、デバイスドライバ上のレイヤーが依存している可能性のあるカーネルマッピングをすべて削除します。

同期データ転送 (ブロックドライバ)

このセクションでは、同期入出力転送を実行するための単純な方法を示します。この方法では、ハードウェアが、DMA を使用して一度に 1 つのデータバッファーのみを転送できる単純なディスク装置であることを前提にしています。また、ソフトウェアコマンドでディスクを起動および停止できることも前提にしています。デバイスドライバの strategy(9E) ルーチンは、新しい要求を受け付ける前に現在の要求の完了を待機します。デバイスは、転送が完了したときに割り込みます。デバイスはまた、エラーが発生した場合にも割り込みます。

ブロックドライバの同期データ転送を実行するための手順は次のとおりです。

1. 無効な buf(9S) 要求をチェックします。

   strategy(9E) に渡された buf(9S) 構造体の有効性をチェックします。すべてのドライバは、次の条件を確認します。

   - 要求が有効なブロックで始まっている。ドライバは、 b_blkno フィールドを正しいデバイスオフセットに変換したあと、そのオフセットがデバイスで有効かどうかを判定します。
   - 要求がデバイス上の最終ブロックを超えていない。
   - デバイス固有の要件が満たされている。

   エラーが検出された場合、ドライバは bioerror(9F) を使用して該当するエラーを示します。ドライバはその後、biodone(9F) を呼び出すことによって要求を完了します。biodone () は、strategy(9E) の呼び出し元に転送が完了したことを通知します。この場合は、エラーのために転送が停止されました。

2. デバイスがビジー状態かどうかをチェックします。

   同期データ転送では、デバイスへのシングルスレッドアクセスが許可されます。デバイスドライバがこのアクセスを実施する場合、次の 2 つの方法があります。

   - ドライバが、mutex によって保護されるビジー状態フラグを保持する。
   - デバイスがビジー状態の場合、ドライバは cv_wait(9F) を使用して条件変数に関して待機する。

   デバイスがビジー状態の場合、スレッドは、割り込みハンドラによってデバイスがビジー状態でなくなったことが示されるまで待機します。使用可能なステータスは、cv_broadcast(9F) または cv_signal(9F) 関数のどちらかで示すことができます。条件変数の詳細については、第 3 章「マルチスレッド」を参照してください。

   デバイスがビジー状態でなくなった場合、strategy(9E) ルーチンは、そのデバイスを使用可能としてマークします。strategy() は次に、転送のためにバッファーとデバイスを準備します。

3. DMA のためのバッファーを設定します。

   ddi_dma_alloc_handle(9F) を使用して DMA ハンドルを割り当てることにより、DMA 転送のためのデータバッファーを準備します。データバッファーをハンドルにバインドするには、ddi_dma_buf_bind_handle(9F) を使用します。DMA リソースおよび関連するデータ構造体の設定については、第 9 章「ダイレクトメモリーアクセス (DMA)」を参照してください。

4. 転送を開始します。

   この時点では、buf(9S) 構造体へのポインタがデバイスの状態構造体に保存されています。それにより、割り込みルーチンは、biodone(9F) を呼び出すことによって転送を完了できます。

   デバイスドライバは次に、データ転送を開始するためにデバイスレジスタにアクセスします。ほとんどの場合、ドライバが、mutex を使用してデバイスレジスタをほかのスレッドから保護します。この場合は、strategy(9E) がシングルスレッドであるため、デバイスレジスタを保護する必要はありません。データロックの詳細については、第 3 章「マルチスレッド」を参照してください。

   実行スレッドがデバイスの DMA エンジンを起動した場合、ドライバは、次のように呼び出し元のルーチンに実行制御を戻すことができます。

```
static int
xxstrategy(struct buf *bp)
{
    struct xxstate *xsp;
    struct device_reg *regp;
    minor_t instance;
    ddi_dma_cookie_t cookie;
    instance = getminor(bp->b_edev);
    xsp = ddi_get_soft_state(statep, instance);
    if (xsp == NULL) {
       bioerror(bp, ENXIO);
       biodone(bp);
       return (0);
    }
    /* validate the transfer request */
    if ((bp->b_blkno >= xsp->Nblocks) || (bp->b_blkno < 0)) {
       bioerror(bp, EINVAL);
       biodone(bp);
       return (0);
    }
    /*
     * Hold off all threads until the device is not busy.
     */
    mutex_enter(&xsp->mu);
    while (xsp->busy) {
       cv_wait(&xsp->cv, &xsp->mu);
    }
    xsp->busy = 1;
    mutex_exit(&xsp->mu);
    /*
     * If the device has power manageable components,
     * mark the device busy with pm_busy_components(9F),
```

```
         * and then ensure that the device
         * is powered up by calling pm_raise_power(9F).
         *
         * Set up DMA resources with ddi_dma_alloc_handle(9F) and
         * ddi_dma_buf_bind_handle(9F).
         */
        xsp->bp = bp;
        regp = xsp->regp;
        ddi_put32(xsp->data_access_handle, &regp->dma_addr,
            cookie.dmac_address);
        ddi_put32(xsp->data_access_handle, &regp->dma_size,
             (uint32_t)cookie.dmac_size);
        ddi_put8(xsp->data_access_handle, &regp->csr,
             ENABLE_INTERRUPTS | START_TRANSFER);
        return (0);
    }
```

5. 割り込んでいるデバイスを処理します。

   デバイスがデータ転送を完了すると、そのデバイスは割り込みを生成します。それにより、最終的にドライバの割り込みルーチンが呼び出されます。ほとんどのドライバは、割り込みを登録するときに、割り込みルーチンへの引数としてデバイスの状態構造体を指定します。`ddi_add_intr(9F)` のマニュアルページおよび 135 ページの「割り込みの登録」を参照してください。それにより、割り込みルーチンは、転送されている `buf(9S)` 構造体に加え、状態構造体から取得できるほかのすべての情報にアクセスできます。

   割り込みハンドラは、転送がエラーなしで完了したかどうかを判定するために、デバイスのステータスレジスタをチェックします。エラーが発生した場合、ハンドラは `bioerror(9F)` を使用して該当するエラーを示します。ハンドラはまた、デバイスの保留中の割り込みをクリアしたあと、`biodone(9F)` を呼び出すことによって転送を完了します。

   最後のタスクとして、ハンドラはビジー状態フラグをクリアします。ハンドラは次に、条件変数に関して `cv_signal(9F)` または `cv_broadcast(9F)` を呼び出すことにより、デバイスがビジー状態でなくなったことを通知します。この通知により、`strategy(9E)` でこのデバイスを待機しているほかのスレッドが、次のデータ転送に進むことができるようになります。

   次の例は、同期割り込みルーチンを示しています。

例 16–4 ブロックドライバの同期割り込みルーチン

```
static u_int
xxintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    struct buf *bp;
    uint8_t status;
    mutex_enter(&xsp->mu);
    status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
    if (!(status & INTERRUPTING)) {
       mutex_exit(&xsp->mu);
       return (DDI_INTR_UNCLAIMED);
```

例 16–4 ブロックドライバの同期割り込みルーチン (続き)

```
    }
    /* Get the buf responsible for this interrupt */
    bp = xsp->bp;
    xsp->bp = NULL;
    /*
     * This example is for a simple device which either
     * succeeds or fails the data transfer, indicated in the
     * command/status register.
     */
    if (status & DEVICE_ERROR) {
       /* failure */
       bp->b_resid = bp->b_bcount;
       bioerror(bp, EIO);
    } else {
       /* success */
       bp->b_resid = 0;
    }
    ddi_put8(xsp->data_access_handle, &xsp->regp->csr,
       CLEAR_INTERRUPT);
    /* The transfer has finished, successfully or not */
    biodone(bp);
    /*
     * If the device has power manageable components that were
     * marked busy in strategy(9F), mark them idle now with
     * pm_idle_component(9F)
     * Release any resources used in the transfer, such as DMA
     * resources ddi_dma_unbind_handle(9F) and
     * ddi_dma_free_handle(9F).
     *
     * Let the next I/O thread have access to the device.
     */
    xsp->busy = 0;
    cv_signal(&xsp->cv);
    mutex_exit(&xsp->mu);
    return (DDI_INTR_CLAIMED);
}
```

非同期データ転送 (ブロックドライバ)

このセクションでは、非同期入出力転送を実行するための方法を示します。ドライバは入出力要求をキューに入れたあと、呼び出し元に制御を戻します。ここでも、ハードウェアが、一度に 1 つの転送が可能な単純なディスク装置であることを前提にしています。デバイスは、データ転送が完了したときに割り込みます。また、エラーが発生した場合にも割り込みが実行されます。非同期データ転送を実行するための基本的な手順は次のとおりです。

1. 無効な buf(9S) 要求をチェックします。
2. 要求をキューに入れます。
3. 最初の転送を開始します。

4. 割り込んでいるデバイスを処理します。

無効な buf 要求のチェック

同期の場合と同様に、デバイスドライバは、strategy(9E) に渡された buf(9S) 構造体の有効性をチェックします。詳細については、342 ページの「同期データ転送 (ブロックドライバ)」を参照してください。

要求のキューへの入力

同期データ転送とは異なり、ドライバは非同期要求の完了を待機しません。代わりに、ドライバはその要求をキューに追加します。キューの先頭は、現在の転送である場合があります。キューの先頭はまた、例 16–5 に示すように、アクティブな要求を保持するための状態構造体の別のフィールドである場合もあります。

キューが最初に空である場合は、ハードウェアがビジー状態ではないため、strategy(9E) は復帰する前に転送を開始します。それ以外の場合、空でないキューで転送が完了すると、割り込みルーチンは新しい転送を開始します。例 16–5 では、新しい転送を開始するかどうかの判定を便宜上、別のルーチンで行なっています。

ドライバは、buf(9S) 構造体の av_forw メンバーと av_back メンバーを使用して転送要求のリストを管理できます。1 つのポインタを使用して片方向リンクリストを管理することも、両方のポインタを一緒に使用して両方向リンクリストを構築することもできます。デバイスのハードウェア仕様によって、デバイスのパフォーマンスを最適化するために (ポリシーの挿入などの) どのタイプのリスト管理を使用するかが指定されます。転送リストはデバイスごとのリストであるため、このリストの先頭と最後尾は状態構造体に格納されます。

次の例は、転送リストなどのドライバの共有データにアクセスできる複数のスレッドを示しています。共有データを識別し、mutex を使用してそのデータを保護する必要があります。mutex のロックの詳細については、第 3 章「マルチスレッド」を参照してください。

例 16–5 ブロックドライバに対するデータ転送要求のキューへの入力

```
static int
xxstrategy(struct buf *bp)
{
    struct xxstate *xsp;
    minor_t instance;
    instance = getminor(bp->b_edev);
    xsp = ddi_get_soft_state(statep, instance);
    /* ... */
    /* validate transfer request */
```

例 16–5 ブロックドライバに対するデータ転送要求のキューへの入力 (続き)

```
    /* ... */
    /*
     * Add the request to the end of the queue. Depending on the device, a sorting
     * algorithm, such as disksort(9F) can be used if it improves the
     * performance of the device.
     */
    mutex_enter(&xsp->mu);
    bp->av_forw = NULL;
    if (xsp->list_head) {
       /* Non-empty transfer list */
       xsp->list_tail->av_forw = bp;
       xsp->list_tail = bp;
    } else {
       /* Empty Transfer list */
       xsp->list_head = bp;
       xsp->list_tail = bp;
    }
    mutex_exit(&xsp->mu);
    /* Start the transfer if possible */
    (void) xxstart((caddr_t)xsp);
    return (0);
}
```

最初の転送の開始

キューへの入力を実装しているデバイスドライバは通常、start() ルーチンを備えています。start() は次の要求をキューから取り出し、デバイスとの間のデータ転送を開始します。この例では、start() はデバイスの状態 (ビジー状態か空いているか) には関係なく、すべての要求を処理します。

注 – start() は、任意のコンテキストから呼び出されるように記述する必要があります。start() は、カーネルコンテキスト内の strategy ルーチンと、割り込みコンテキスト内の割り込みルーチンの両方から呼び出すことができます。

---

start() は、アイドルのデバイスを起動できるように strategy(9E) が要求をキューに入れるたびに strategy() から呼び出されます。デバイスがビジー状態の場合、start() はただちに復帰します。

start() はまた、取り込まれた割り込みから割り込みハンドラが復帰する前に、その割り込みハンドラからも呼び出されるため、空でないキューを処理できます。キューが空の場合、start() はただちに復帰します。

start() は非公開のドライバルーチンであるため、start() は任意の引数を取り、任意の型を返すことができます。次のコーディング例は、DMA コールバックとして使用するために記述されています (ただし、その部分は示されていません)。した

がって、この例では caddr_t を引数として取り、int を返す必要があります。DMA コールバックルーチンの詳細については、176 ページの「リソース割り当てエラーの処理」を参照してください。

例 16–6 ブロックドライバに対する最初のデータ要求の開始

```
static int
xxstart(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    struct buf *bp;

    mutex_enter(&xsp->mu);
    /*
     * If there is nothing more to do, or the device is
     * busy, return.
     */
    if (xsp->list_head == NULL || xsp->busy) {
       mutex_exit(&xsp->mu);
       return (0);
    }
    xsp->busy = 1;
    /* Get the first buffer off the transfer list */
    bp = xsp->list_head;
    /* Update the head and tail pointer */
    xsp->list_head = xsp->list_head->av_forw;
    if (xsp->list_head == NULL)
       xsp->list_tail = NULL;
    bp->av_forw = NULL;
    mutex_exit(&xsp->mu);
    /*
     * If the device has power manageable components,
     * mark the device busy with pm_busy_components(9F),
     * and then ensure that the device
     * is powered up by calling pm_raise_power(9F).
     *
     * Set up DMA resources with ddi_dma_alloc_handle(9F) and
     * ddi_dma_buf_bind_handle(9F).
     */
    xsp->bp = bp;
    ddi_put32(xsp->data_access_handle, &xsp->regp->dma_addr,
        cookie.dmac_address);
    ddi_put32(xsp->data_access_handle, &xsp->regp->dma_size,
         (uint32_t)cookie.dmac_size);
    ddi_put8(xsp->data_access_handle, &xsp->regp->csr,
         ENABLE_INTERRUPTS | START_TRANSFER);
    return (0);
}
```

割り込んでいるデバイスの処理

割り込みルーチンは非同期バージョンに似ていますが、start() の呼び出しが追加され、cv_signal(9F) の呼び出しが削除されています。

例 16–7　非同期割り込みのためのブロックドライバルーチン

```
static u_int
xxintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    struct buf *bp;
    uint8_t status;
    mutex_enter(&xsp->mu);
    status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
    if (!(status & INTERRUPTING)) {
        mutex_exit(&xsp->mu);
        return (DDI_INTR_UNCLAIMED);
    }
    /* Get the buf responsible for this interrupt */
    bp = xsp->bp;
    xsp->bp = NULL;
    /*
     * This example is for a simple device which either
     * succeeds or fails the data transfer, indicated in the
     * command/status register.
     */
    if (status & DEVICE_ERROR) {
        /* failure */
        bp->b_resid = bp->b_bcount;
        bioerror(bp, EIO);
    } else {
        /* success */
        bp->b_resid = 0;
    }
    ddi_put8(xsp->data_access_handle, &xsp->regp->csr,
        CLEAR_INTERRUPT);
    /* The transfer has finished, successfully or not */
    biodone(bp);
    /*
     * If the device has power manageable components that were
     * marked busy in strategy(9F), mark them idle now with
     * pm_idle_component(9F)
     * Release any resources used in the transfer, such as DMA
     * resources (ddi_dma_unbind_handle(9F) and
     * ddi_dma_free_handle(9F)).
     *
     * Let the next I/O thread have access to the device.
     */
    xsp->busy = 0;
    mutex_exit(&xsp->mu);
    (void) xxstart((caddr_t)xsp);
    return (DDI_INTR_CLAIMED);
}
```

dump() エントリポイントと print() エントリポイント

このセクションでは、dump(9E) エントリポイントと print(9E) エントリポイントについて説明します。

dump() エントリポイント (ブロックドライバ)

dump(9E) エントリポイントは、システム障害が発生した場合に、仮想アドレス空間の一部を指定されたデバイスに直接コピーするために使用されます。dump() はまた、チェックポイント操作中にカーネルの状態をディスクにコピーするためにも使用されます。詳細については、cpr(7) と dump(9E) のマニュアルページを参照してください。チェックポイント操作中は割り込みが無効になっているため、このエントリポイントは、割り込みを使用せずにこの操作を実行できる必要があります。

```
int dump(dev_t dev, caddr_t addr, daddr_t blkno, int nblk)
```

各表記の意味は次のとおりです。

dev ダンプを受信するデバイスのデバイス番号。

addr ダンプを開始する基準となるカーネル仮想アドレス。

blkno ダンプを開始するブロック。

nblk ダンプするブロック数。

このダンプは、既存のドライバの正常な動作に依存します。

print() エントリポイント (ブロックドライバ)

```
int print(dev_t dev, char *str)
```

print(9E) エントリポイントは、検出された例外に関するメッセージを表示するためにシステムから呼び出されます。print(9E) は、システムに代わってメッセージをコンソールに送信するために cmn_err(9F) を呼び出します。次の例は、標準的な print() エントリポイントを示しています。

```
static int
 xxprint(dev_t dev, char *str)
 {
     cmn_err(CE_CONT, “xx: %s\n”, str);
     return (0);
 }
```

ディスク装置ドライバ

ディスク装置は、ブロックデバイスドライバの重要なクラスを表します。

ディスクの ioctl

Oracle Solaris ディスクドライバは、Oracle Solaris ディスクドライバに固有の `ioctl` コマンドの最小限のセットをサポートする必要があります。これらの入出力制御は、`dkio(7I)` のマニュアルページで指定されています。ディスク入出力制御は、デバイスドライバとの間でディスク情報を転送します。Oracle Solaris ディスク装置は、`format(1M)` や `newfs(1M)` などのディスクユーティリティーコマンドでサポートされています。必須のディスク入出力制御は次のとおりです。

`DKIOCINFO` ディスク制御装置を記述した情報を返します。

`DKIOCGAPART` ディスクのパーティションマップを返します。

`DKIOCSAPART` ディスクのパーティションマップを設定します。

`DKIOCGGEOM` ディスクのジオメトリを返します。

`DKIOCSGEOM` ディスクのジオメトリを設定します。

`DKIOCGVTOC` ディスクのボリューム構成テーブルを返します。

`DKIOCSVTOC` ディスクのボリューム構成テーブルを設定します。

ディスクパフォーマンス

Oracle Solaris DDI/DKI は、ファイルシステムのパフォーマンスを向上させるために入出力転送を最適化する機能を提供します。このメカニズムでは、ファイルシステムのディスクアクセスが最適化されるように入出力要求のリストが管理されます。入出力要求のキューへの入力の説明については、345 ページの「非同期データ転送 (ブロックドライバ)」を参照してください。

`diskhd` 構造体は、入出力要求のリンクリストを管理するために使用されます。

```
struct diskhd {
    long     b_flags;         /* not used, needed for consistency*/
    struct   buf *b_forw,    *b_back;       /* queue of unit queues */
    struct   buf *av_forw,    *av_back;    /* queue of bufs for this unit */
    long     b_bcount;            /* active flag */
};
```

`diskhd` データ構造体には、ドライバが操作できる 2 つの `buf` ポインタがあります。`av_forw` ポインタは、最初のアクティブな入出力要求を指しています。2 番目のポインタである `av_back` は、リスト上の最後のアクティブな要求を指しています。

disksort(9F) には、この構造体へのポインタが、処理されている現在の buf 構造体へのポインタとともに引数として渡されます。disksort() ルーチンは、ディスクのシークを最適化するために buf 要求をソートします。このルーチンは次に、buf ポインタを diskhd リストに挿入します。disksort() プログラムは、buf 構造体の b_resid 内にある値をソートキーとして使用します。ドライバは、この値の設定を担当します。ほとんどの Oracle ディスクドライバは、シリンダグループをソートキーとして使用します。このアプローチによって、ファイルシステムの先読みアクセスが最適化されます。

diskhd リストにデータが追加されたら、デバイスはそのデータを転送する必要があります。デバイスが要求の処理によってビジー状態でない場合、xxstart() ルーチンは最初の buf 構造体を diskhd リストから取得し、転送を開始します。

デバイスがビジー状態の場合、ドライバは xxstrategy() エントリポイントから復帰します。ハードウェアでのデータ転送が完了すると、割り込みが生成されます。次に、デバイスに対する処理を行うために、ドライバの割り込みルーチンが呼び出されます。割り込みに対する処理を行ったあと、ドライバは、diskhd リスト内の次の buf 構造体を処理するために start() ルーチンを呼び出すことができます。

◆ ◆ ◆ 第 17 章

17 SCSI ターゲットドライバ

Oracle Solaris DDI/DKI では、SCSI デバイスに対するソフトウェアインタフェースは、ターゲットドライバとホストバスアダプタ *(HBA)* ドライバという 2 つの大きな部分に分割されています。ターゲットは、ディスクやテープドライブなどの SCSI バス上のデバイスのドライバを指します。ホストバスアダプタは、ホストマシン上の SCSI コントローラのドライバを指します。SCSA は、これら 2 つのコンポーネント間のインタフェースを定義します。この章では、ターゲットドライバについてのみ説明します。ホストバスアダプタのドライバについては、第 18 章「SCSI ホストバスアダプタドライバ」を参照してください。

注 – 「ホストバスアダプタ」および「HBA」という用語は、SCSI の仕様で定義されている「ホストアダプタ」と同等です。

この章では、次の内容について説明します。

ターゲットデバイスの概要

デバイスに応じて、文字またはブロックのデバイスドライバのいずれかがターゲットドライバとなります。テープドライブ用のドライバは、通常、文字デバイスドライバによって、ディスクはブロックデバイスドライバによって処理されます。この章では、SCSI ターゲットドライバを記述する方法について説明します。この章では、SCSA によって生じる、SCSI ターゲットデバイス用のブロックおよび文字ドライバの追加の要件について説明します。

次の参照ドキュメントには、ターゲットドライバおよびホストバスアダプタドライバの設計者に必要な補足情報が記載されています。

『Small Computer System Interface 2 (SCSI-2)』、ANSI/NCITS X3.131-1994、Global Engineering Documents、1998 年。ISBN 1199002488。

『*The Basics of SCSI*』、第 4 版、ANCOT Corporation、1998 年。ISBN 0963743988。

ハードウェアベンダーによって提供されている、ターゲットデバイスの SCSI コマンド仕様も参照してください。

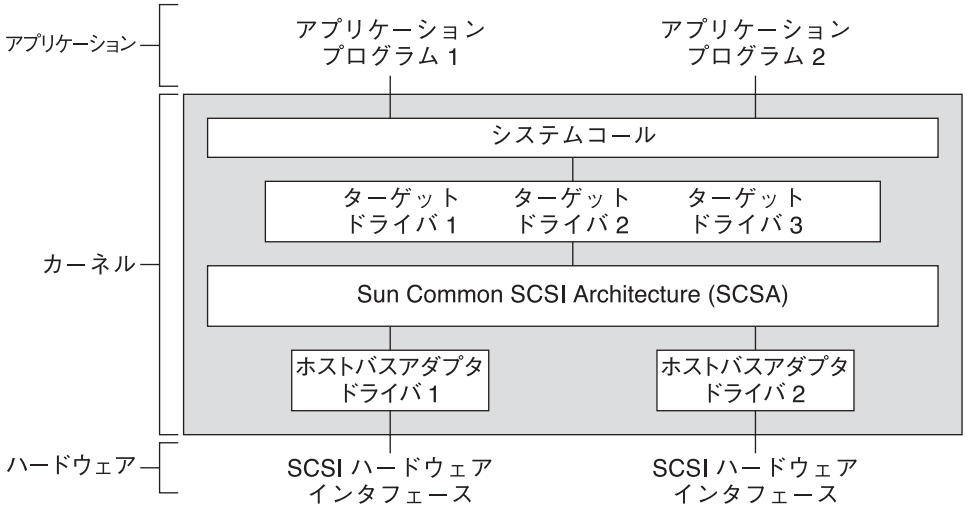
Sun Common SCSI Architecture の概要

Sun Common SCSI Architecture (SCSA) は、ターゲットドライバからホストバスアダプタドライバに SCSI コマンドを転送するための Oracle Solaris DDI/DKI プログラミングインタフェースです。このインタフェースは、ホストバスアダプタのハードウェアの種類、プラットフォーム、プロセッサのアーキテクチャー、およびインタフェース上でトランスポートされる SCSI コマンドには依存しません。

SCSA に準拠することで、ターゲットドライバはホストバスアダプタのハードウェア実装を認識することなく、SCSI コマンドをターゲットデバイスに渡すことができます。

概念上、SCSA は SCSI コマンドを構築することと、SCSI バスをまたいでデータとともにコマンドをトランスポートすることを区別しています。このアーキテクチャーは、高レベルと低レベルのソフトウェアコンポーネントの間のソフトウェアインタフェースを定義しています。高レベルソフトウェアコンポーネントは、1 つ以上の SCSI ターゲットドライバによって構成されます。このドライバは、I/O 要求を周辺デバイスにとって適切な SCSI コマンドに変換します。次の例に、SCSI のアーキテクチャーを示します。

図 17–1　SCSA ブロック図



下位レベルのソフトウェアコンポーネントは、SCSA インタフェース層と、1 つ以上のホストバスアダプタドライバから構成されています。ターゲットドライバは、必要な機能を実行するために必要とされる適切な SCSI コマンドの生成と、結果の処理を行います。

一般的な制御フロー

トランスポートエラーが発生しないと仮定すると、読み込みまたは書き込み要求の一般的な制御フローは、次に説明するステップのようになります。

1. ターゲットドライバの `read(9E)` または `write(9E)` エントリポイントが呼び出されます。メモリをロックダウンするために `physio(9F)` が使用され、`buf` 構造体の準備と strategy ルーチンの呼び出しが行われます。
2. ターゲットドライバの `strategy(9E)` ルーチンが要求をチェックします。`strategy()` は次に、`scsi_init_pkt(9F)` を使用して、`scsi_pkt(9S)` を割り当てます。ターゲットドライバはパケットを初期化し、`scsi_setup_cdb(9F)` 関数を使用して、SCSI コマンド記述子ブロック (CDB) を設定します。ターゲットドライバはタイムアウトも指定します。次に、ドライバはコールバック関数へのポインタを提供します。コールバック関数は、コマンドの完了時にホストバスアダプタドライバによって呼び出されます。`buf(9S)` ポインタは SCSI パケットのターゲット専用領域に保存される必要があります。
3. ターゲットドライバは `scsi_transport(9F)` を使用して、パケットをホストバスアダプタドライバに送信します。これでターゲットドライバは解放され、別の要求を受け入れることができます。ターゲットドライバは、パケットのトランスポート中にパケットにアクセスしてはいけません。ホストバスアダプタドライバ

またはターゲットのいずれかがキューイングをサポートしている場合は、パケットのトランスポート中に新しい要求を送信できます。

4. SCSI が解放され、ターゲットがビジー状態でなくなるとすぐに、ホストバスアダプタドライバはターゲットを選択して CDB を渡します。ターゲットドライバはコマンドを実行します。ターゲットは次に、要求されたデータ転送を実行します。
5. ターゲットが完了ステータスを送信してコマンドが完了すると、ホストバスアダプタドライバはそれをターゲットドライバに通知します。通知を実行するため、ホストは SCSI パケットで指定されていた完了関数を呼び出します。この時点で、ホストバスアダプタドライバはパケットの処理を担当しておらず、ターゲットドライバがパケットの所有権を再取得しています。
6. SCSI パケットの完了ルーチンが、返された情報を分析します。完了ルーチンは次に、SCSI 操作が成功したかどうかを判定します。エラーが発生した場合、ターゲットドライバはもう一度 `scsi_transport(9F)` を呼び出して、コマンドを再試行します。ホストバスアダプタドライバが要求の自動検知をサポートしていない場合、ターゲットドライバは要求検知パケットを送信して、チェック条件に該当する場合に検知データを取得する必要があります。
7. 正常に完了したあと、またはコマンドを再試行できない場合、ターゲットドライバは `scsi_destroy_pkt(9F)` を呼び出します。`scsi_destroy_pkt()` は、データを同期します。次に `scsi_destroy_pkt()` はパケットを解放します。パケットを解放する前にターゲットドライバがデータにアクセスする必要がある場合は、`scsi_sync_pkt(9F)` が呼び出されます。
8. 最後に、ターゲットドライバは要求元アプリケーションに、読み取りまたは書き込みトランザクションが完了したことを通知します。この通知は、文字デバイスの場合、ドライバ内の `read(9E)` エントリポイントから通知が返されることによって行われます。それ以外の場合、通知は `biodone(9F)` を通して間接的に行われます。

SCSA によって、これらの操作の多くをプロセスのさまざまな時点で、オーバーラップとキューイングの両方の方法で実行できます。このモデルでは、ホストバスアダプタドライバでシステムリソースの管理が行われます。ソフトウェアインタフェースによって、さまざまな機能レベルの SCSI バスアダプタを使用して、ホストバスアダプタドライバでターゲットドライバ関数を実行できます。

SCSA 関数

SCSA では、リソースの割り当てと解放、制御状態の検知と設定、SCSI コマンドのトランスポートを管理する関数が定義されています。次の表にこれらの関数を示します。

表 17–1 SCSA 標準関数

関数名	カテゴリ
scsi_abort(9F)	エラー処理
scsi_alloc_consistent_buf(9F)	
scsi_destroy_pkt(9F)	
scsi_dmafree(9F)	
scsi_free_consistent_buf(9F)	
scsi_ifgetcap(9F)	トランスポートの情報と制御
scsi_ifsetcap(9F)	
scsi_init_pkt(9F)	リソース管理
scsi_poll(9F)	ポーリングが行われる入出力
scsi_probe(9F)	プローブ関数
scsi_reset(9F)	
scsi_setup_cdb(9F)	CDB 初期化関数
scsi_sync_pkt(9F)	
scsi_transport(9F)	コマンドのトランスポート
scsi_unprobe(9F)	

注 – ドライバで SCSI-1 デバイスを操作する必要がある場合は、makecom(9F) を使用します。

ハードウェア構成ファイル

SCSI デバイスに自己識別機能はないため、ターゲットドライバにはハードウェア構成ファイルが必要です。詳細については、driver.conf(4) と scsi_free_consistent_buf(9F) のマニュアルページを参照してください。次に、典型的な構成ファイルを示します。

```
name="xx" class="scsi" target=2 lun=0;
```

システムは自動構成中にファイルを読み込みます。システムは *class* プロパティーを使用して、そのドライバの親になる可能性のあるドライバを識別します。システムは次に、そのドライバを、*scsi* クラスである任意の親ドライバに接続しようとします。ホストバスアダプタドライバはすべてこのクラスです。*parent* プロパティーでは

なく、*class* プロパティーを使用することが推奨されています。この方法では、指定された *target* および *lun* の ID で予期されたデバイスを見つけたすべてのホストバスアダプタドライバがターゲットに接続できます。probe(9E) ルーチンでクラスの確認を行うのは、ターゲットドライバの側です。

宣言とデータ構造体

ターゲットドライバでは、ヘッダーファイル <sys/scsi/scsi.h> をインクルードする必要があります。

SCSI ターゲットドライバは次のコマンドを使用して、バイナリモジュールを生成する必要があります。

```
ld -r xx xx.o -N"misc/scsi"
```

scsi_device 構造体

ホストバスアダプタドライバは、probe(9E) または attach(9E) ルーチンのいずれかが呼び出される前に、ターゲットドライバの scsi_device(9S) 構造体を割り当てて初期化します。この構造体には、一般的な情報とデバイス固有の情報を含む情報領域を指すポインタなど、各 SCSI 論理ユニットに関する情報が格納されます。システムに接続されている論理ユニットごとに 1 つの scsi_device(9S) 構造体が存在します。ターゲットドライバは、ddi_get_driver_private(9F) を呼び出すことで、この構造体へのポインタを取得できます。

注意 – ホストバスアダプタドライバは、ターゲットデバイスの dev_info 構造体内にある非公開フィールドを使用するため、ターゲットドライバが ddi_set_driver_private(9F) を使用してはいけません。

scsi_device(9S) 構造体には、次のフィールドがあります。

```
struct scsi_device {
    struct scsi_address          sd_address;    /* opaque address */
    dev_info_t                   *sd_dev;       /* device node */
    kmutex_t                     sd_mutex;
    void                         *sd_reserved;
    struct scsi_inquiry          *sd_inq;
    struct scsi_extended_sense   *sd_sense;
    caddr_t                      sd_private;
};
```

各表記の意味は次のとおりです。

sd_address　SCSI リソースの割り当てのためのルーチンに渡されるデータ構造体です。

`sd_dev` ターゲットの `dev_info` 構造体を指すポインタです。

`sd_mutex` ターゲットドライバが使用する mutex です。この mutex はホストバスアダプタドライバによって初期化され、ターゲットドライバによって、デバイスごとの mutex として使用できます。この mutex は、`scsi_transport(9F)` または `scsi_poll(9F)` への呼び出しにわたって保持しないでください。mutex の詳細については、第 3 章「マルチスレッド」を参照してください。

`sd_inq` ターゲットデバイスの SCSI 照会データへのポインタです。`scsi_probe(9F)` ルーチンは、バッファーを割り当て、照会データでそのバッファーを満たして、このフィールドに追加します。

`sd_sense` デバイスから送られた SCSI 要求検知データを収めるバッファーを指すポインタです。ターゲットドライバは、このバッファーを割り当てて管理する必要があります。364 ページの「attach() エントリポイント (SCSI ターゲットドライバ)」を参照してください。

`sd_private` ターゲットドライバが使用するポインタフィールドです。このフィールドは、ターゲットドライバの非公開の状態構造体を指すポインタの格納によく使われます。

scsi_pkt 構造体 (ターゲットドライバ)

`scsi_pkt` 構造体には次のフィールドが含まれます。

```
struct scsi_pkt {
    opaque_t  pkt_ha_private;         /* private data for host adapter */
    struct scsi_address pkt_address;  /* destination packet is for */
    opaque_t  pkt_private;            /* private data for target driver */
    void      (*pkt_comp)(struct scsi_pkt *);  /* completion routine */
    uint_t    pkt_flags;              /* flags */
    int       pkt_time;               /* time allotted to complete command */
    uchar_t  *pkt_scbp;               /* pointer to status block */
    uchar_t  *pkt_cdbp;               /* pointer to command block */
    ssize_t   pkt_resid;              /* data bytes not transferred */
    uint_t    pkt_state;              /* state of command */
    uint_t    pkt_statistics;         /* statistics */
    uchar_t   pkt_reason;             /* reason completion called */
};
```

各表記の意味は次のとおりです。

`pkt_address` `scsi_init_pkt(9F)` によって設定されたターゲットドライバのアドレスです。

`pkt_private` ターゲットドライバのプライベートデータを格納する場所です。*pkt_private* は通常、コマンドの `buf(9S)` ポインタを保存するために使用されます。

pkt_comp　完了ルーチンのアドレスです。ホストバスアダプタドライバはドライバがコマンドをトランスポートしたときにこのルーチンを呼び出します。コマンドのトランスポートは、コマンドが成功したことを意味するわけではありません。ターゲットがビジー状態になっていた可能性があります。タイムアウト期間が経過する前にターゲットが応答しなかった可能性もあります。pkt_time フィールドの説明を参照してください。ターゲットドライバは、このフィールドで有効な値を指定する必要があります。ドライバへの通知が不要な場合は、この値を NULL にできます。

注 – 2 つの異なる SCSI コールバックルーチンが用意されています。pkt_comp フィールドは、ホストバスアダプタが処理を完了したときに呼び出される *completion callback* ルーチンを指します。また、現在使用できないリソースが利用可能になりそうなときに呼び出される *resource callback* ルーチンが用意されています。scsi_init_pkt(9F) のマニュアルページを参照してください。

---

pkt_flags　たとえば、切断するための特権なしにコマンドをトランスポートしたり (FLAG_NODISCON)、コールバックを無効にしたり (FLAG_NOINTR) するために、追加の制御情報を提供します。詳細については、scsi_pkt(9S) のマニュアルページを参照してください。

pkt_time　タイムアウト値 (秒単位) です。コマンドがこの時間内に完了しない場合、ホストバスアダプタは pkt_reason を CMD_TIMEOUT に設定して、完了ルーチンを呼び出します。ターゲットドライバはこのフィールドを、コマンドの実行にかかる可能性のある最大時間より大きい値に設定する必要があります。タイムアウトを 0 に設定すると、タイムアウトは要求されません。タイムアウトの起点は、コマンドが SCSI バス上で送信された時点です。

pkt_scbp　SCSI ステータスの完了ブロックへのポインタです。このフィールドの値はホストバスアダプタドライバによって格納されます。

pkt_cdbp　ターゲットデバイスに送信される実際のコマンドが格納されている、SCSI コマンド記述子ブロックへのポインタです。ホストバスアダプタドライバが、このフィールドの解釈を行うことはありません。ターゲットドライバはこのフィールドに、ターゲットデバイスが処理できるコマンドを入れる必要があります。

pkt_resid　未処理の操作内容です。pkt_resid が使用される方法に応じて、pkt_resid フィールドには 2 つの異なる用途があります。scsi_init_pkt(9F) コマンドでの DMA リソースの割り当てのために pkt_resid が使用される場合、pkt_resid は割り当てできないバイト数を示します。DMA リソースは、DMA ハードウェアの

分散と集中や、その他のデバイスの制限のために、割り当てができない場合もあります。コマンドのトランスポート後、pkt_resid は転送できないデータバイト数を示します。このフィールドの値は、完了ルーチンが呼び出される前に、ホストバスアダプタドライバによって格納されます。

pkt_state　コマンドの状態を示します。コマンドの進行状況につれて、ホストバスアダプタドライバがこのフィールドに値を格納します。次に示す 5 つのコマンドの状態ごとに 1 ビットがこのフィールドに設定されます。

- STATE_GOT_BUS – バスの取得
- STATE_GOT_TARGET – ターゲットの選択
- STATE_SENT_CMD – コマンドの送信
- STATE_XFERRED_DATA – データの転送 (適切な場合)
- STATE_GOT_STATUS – デバイスからのステータスの受信

pkt_statistics　ホストバスアダプタドライバによって設定されたトランスポート関連の統計情報が格納されます。

pkt_reason　完了ルーチンが呼び出された理由を示します。完了ルーチンがこのフィールドをデコードします。ルーチンはその後、適切な処理を実行します。トランスポートエラーが発生せず、コマンドが完了すると、このフィールドは CMD_CMPLT に設定されます。このフィールドにほかの値がある場合はエラーを示します。コマンドが完了したら、ターゲットドライバは pkt_scbp フィールドでチェック条件のステータスを調べる必要があります。詳細については、scsi_pkt(9S) のマニュアルページを参照してください。

SCSI ターゲットドライバの自動構成

SCSI ターゲットドライバには、標準の自動構成ルーチン _init(9E)、_fini(9E)、および _info(9E) を実装する必要があります。詳細については、101 ページの「ロード可能なドライバインタフェース」を参照してください。

次のルーチンも必要ですが、これらのルーチンは特定の SCSI および SCSA の処理を実行する必要があります。

- probe(9E)
- attach(9E)
- detach(9E)
- getinfo(9E)

probe() エントリポイント (SCSI ターゲットドライバ)

SCSI ターゲットデバイスは自己識別しないため、ターゲットドライバが probe(9E) ルーチンを備えている必要があります。このルーチンは、予期される種類のデバイスが存在していて、応答しているかどうかを判定する必要があります。

probe(9E) ルーチンの一般的な構造とリターンコードは、ほかのデバイスドライバの構造およびリターンコードと同じです。SCSI ターゲットドライバは、probe(9E) エントリポイントで scsi_probe(9F) ルーチンを使用する必要があります。scsi_probe(9F) はデバイスに SCSI 照会コマンドを送信し、結果を示すコードを返します。SCSI 照会コマンドが成功すると、scsi_probe(9F) は scsi_inquiry(9S) 構造体を割り当てて、構造体にデバイスの照会データを格納します。scsi_probe(9F) から復帰すると、scsi_device(9S) 構造体の sd_inq フィールドが、この scsi_inquiry(9S) 構造体を指します。

probe(9E) はステートレスである必要があるため、ターゲットドライバは、scsi_probe(9F) が失敗した場合でも、 probe(9E) が復帰する前に scsi_unprobe(9F) を呼び出す必要があります。

例 17–1 に、一般的な probe(9E) ルーチンを示します。この例のルーチンは、dev_info 構造体の非公開フィールドから scsi_device(9S) 構造体を取得しています。また、メッセージ内に出力するためのデバイスの SCSI ターゲットと論理ユニット番号を取得しています。probe(9E) ルーチンは次に scsi_probe(9F) を呼び出して、予期されるデバイス (この場合はプリンタ) が存在することを確認しています。

成功すると、scsi_probe(9F) は scsi_inquiry(9S) 構造体内にあるデバイスの SCSI 照会データを、scsi_device(9S) 構造体の sd_inq フィールドに追加します。ドライバはその後、デバイスの種類がプリンタであるかどうかを判定できます。この情報は inq_dtype フィールドで報告されます。デバイスがプリンタである場合、scsi_dname(9F) を使用してデバイスの種類を文字列に変換し、scsi_log(9F) でその種類が報告されます。

例 17–1　SCSI ターゲットドライバの probe (9E) ルーチン

```
static int
xxprobe(dev_info_t *dip)
{
    struct scsi_device *sdp;
    int rval, target, lun;
    /*
     * Get a pointer to the scsi_device(9S) structure
     */
    sdp = (struct scsi_device *)ddi_get_driver_private(dip);

    target = sdp->sd_address.a_target;
    lun = sdp->sd_address.a_lun;
    /*
```

例 17–1 SCSI ターゲットドライバの probe (9E) ルーチン (続き)

```
     * Call scsi_probe(9F) to send the Inquiry command. It will
     * fill in the sd_inq field of the scsi_device structure.
     */
    switch (scsi_probe(sdp, NULL_FUNC)) {
    case SCSIPROBE_FAILURE:
    case SCSIPROBE_NORESP:
    case SCSIPROBE_NOMEM:
       /*
        * In these cases, device might be powered off,
        * in which case we might be able to successfully
        * probe it at some future time - referred to
        * as 'deferred attach'.
        */
        rval = DDI_PROBE_PARTIAL;
        break;
    case SCSIPROBE_NONCCS:
    default:
        /*
         * Device isn't of the type we can deal with,
         * and/or it will never be usable.
         */
        rval = DDI_PROBE_FAILURE;
        break;
    case SCSIPROBE_EXISTS:
        /*
         * There is a device at the target/lun address. Check
         * inq_dtype to make sure that it is the right device
         * type. See scsi_inquiry(9S)for possible device types.
         */
        switch (sdp->sd_inq->inq_dtype) {
        case DTYPE_PRINTER:
        scsi_log(sdp, "xx", SCSI_DEBUG,
           "found %s device at target%d, lun%d\n",
            scsi_dname((int)sdp->sd_inq->inq_dtype),
            target, lun);
        rval = DDI_PROBE_SUCCESS;
        break;
        case DTYPE_NOTPRESENT:
        default:
        rval = DDI_PROBE_FAILURE;
        break;
        }
    }
    scsi_unprobe(sdp);
    return (rval);
}
```

probe(9E) ルーチンをより詳細に記述すると、scsi_inquiry(9S) をチェックして、そのデバイスが特定のドライバで予期されている種類であることを確認できます。

attach() エントリポイント (SCSI ターゲットドライバ)

probe(9E) ルーチンで、予期されるデバイスが存在することを確認したら、attach(9E) が呼び出されます。attach() は次のタスクを実行します。

- インタンスごとのデータを割り当てて初期化する。
- マイナーデバイスノード情報を作成する。
- デバイスまたはシステムの一時停止後にハードウェアの状態を復元する。詳細については、109 ページの「attach() エントリポイント」を参照してください。

SCSI ターゲットドライバは scsi_probe(9F) をもう一度呼び出して、デバイスの照会データを取得する必要があります。ドライバは、SCSI 要求検知パケットを作成する必要もあります。アタッチが成功した場合、attach() 関数は scsi_unprobe(9F) を呼び出してはいけません。

要求検知パケットを作成するには、scsi_alloc_consistent_buf(9F)、scsi_init_pkt(9F)、および scsi_setup_cdb(9F) の 3 つのルーチンが使用されます。scsi_alloc_consistent_buf(9F) は、変化しない DMA に適したバッファーを割り当てます。次に scsi_alloc_consistent_buf () が buf(9S) 構造体へのポインタを返します。変化しないバッファーの利点は、データを明示的に同期する必要がない点です。つまり、ターゲットドライバはコールバック後にデータにアクセスできます。検知バッファーのアドレスを使用して、デバイスの scsi_device(9S) 構造体の sd_sense 要素を初期化する必要があります。scsi_init_pkt(9F) は scsi_pkt(9S) 構造体を作成し、部分的に初期化します。scsi_setup_cdb(9F) は SCSI コマンド記述子ブロックを作成します。この場合は、SCSI 要求検知コマンドを作成することで、それを行っています。

SCSI デバイスは自己識別しないため、reg プロパティーを持っていないことに注意してください。結果として、ドライバで pm-hardware-state プロパティーを設定する必要が生じます。pm-hardware-state を設定することで、このデバイスは一時停止してから再開する必要があることをフレームワークに通知します。

次の例では、SCSI ターゲットドライバの attach() ルーチンを示します。

例 17–2　SCSI ターゲットドライバの attach (9E) ルーチン

```
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    struct xxstate         *xsp;
    struct scsi_pkt        *rqpkt = NULL;
    struct scsi_device     *sdp;
    struct buf         *bp = NULL;
    int            instance;
    instance = ddi_get_instance(dip);
    switch (cmd) {
```

例 17–2 SCSI ターゲットドライバの attach (9E) ルーチン (続き)

```
        case DDI_ATTACH:
        break;
        case DDI_RESUME:
        /* For information, see the "Directory Memory Access (DMA)" */
        /* chapter in this book. */
        default:
        return (DDI_FAILURE);
    }
    /*
     * Allocate a state structure and initialize it.
     */
    xsp = ddi_get_soft_state(statep, instance);
    sdp = (struct scsi_device *)ddi_get_driver_private(dip);
    /*
     * Cross-link the state and scsi_device(9S) structures.
     */
    sdp->sd_private = (caddr_t)xsp;
    xsp->sdp = sdp;
    /*
     * Call scsi_probe(9F) again to get and validate inquiry data.
     * Allocate a request sense buffer. The buf(9S) structure
     * is set to NULL to tell the routine to allocate a new one.
     * The callback function is set to NULL_FUNC to tell the
     * routine to return failure immediately if no
     * resources are available.
     */
    bp = scsi_alloc_consistent_buf(&sdp->sd_address, NULL,
    SENSE_LENGTH, B_READ, NULL_FUNC, NULL);
    if (bp == NULL)
        goto failed;
    /*
     * Create a Request Sense scsi_pkt(9S) structure.
     */
    rqpkt = scsi_init_pkt(&sdp->sd_address, NULL, bp,
    CDB_GROUP0, 1, 0, PKT_CONSISTENT, NULL_FUNC, NULL);
    if (rqpkt == NULL)
        goto failed;
    /*
     * scsi_alloc_consistent_buf(9F) returned a buf(9S) structure.
     * The actual buffer address is in b_un.b_addr.
     */
    sdp->sd_sense = (struct scsi_extended_sense *)bp->b_un.b_addr;
    /*
     * Create a Group0 CDB for the Request Sense command
     */
    if (scsi_setup_cdb((union scsi_cdb *)rqpkt->pkt_cdbp,
        SCMD_REQUEST_SENSE, 0, SENSE__LENGTH, 0) == 0)
         goto failed;;
    /*
     * Fill in the rest of the scsi_pkt structure.
     * xxcallback() is the private command completion routine.
     */
    rqpkt->pkt_comp = xxcallback;
    rqpkt->pkt_time = 30; /* 30 second command timeout */
    rqpkt->pkt_flags |= FLAG_SENSING;
```

例 17–2 SCSI ターゲットドライバの attach (9E) ルーチン (続き)

```
    xsp->rqs = rqpkt;
    xsp->rqsbuf = bp;
    /*
     * Create minor nodes, report device, and do any other initialization. */
     * Since the device does not have the 'reg' property,
     * cpr will not call its DDI_SUSPEND/DDI_RESUME entries.
     * The following code is to tell cpr that this device
     * needs to be suspended and resumed.
     */
    (void) ddi_prop_update_string(device, dip,
     "pm-hardware-state", "needs-suspend-resume");
    xsp->open = 0;
    return (DDI_SUCCESS);
failed:
    if (bp)
        scsi_free_consistent_buf(bp);
    if (rqpkt)
        scsi_destroy_pkt(rqpkt);
    sdp->sd_private = (caddr_t)NULL;
    sdp->sd_sense = NULL;
    scsi_unprobe(sdp);
    /* Free any other resources, such as the state structure. */
    return (DDI_FAILURE);
}
```

detach() エントリポイント (SCSI ターゲットドライバ)

detach(9E) エントリポイントは、attach(9E) の逆の操作を行うものです。detach() では、attach () で割り当てられたすべてのリソースを解放する必要があります。成功した場合、detach は scsi_unprobe(9F) を呼び出す必要があります。次の例では、ターゲットドライバの detach() ルーチンを示します。

例 17–3 SCSI ターゲットドライバの detach (9E) ルーチン

```
static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    struct xxstate *xsp;
    switch (cmd) {
    case DDI_DETACH:
      /*
       * Normal detach(9E) operations, such as getting a
       * pointer to the state structure
       */
      scsi_free_consistent_buf(xsp->rqsbuf);
      scsi_destroy_pkt(xsp->rqs);
      xsp->sdp->sd_private = (caddr_t)NULL;
      xsp->sdp->sd_sense = NULL;
      scsi_unprobe(xsp->sdp);
```

例 17–3 SCSI ターゲットドライバの detach (9E) ルーチン (続き)

```
      /*
       * Remove minor nodes.
       * Free resources, such as the state structure and properties.
       */
          return (DDI_SUCCESS);
    case DDI_SUSPEND:
      /* For information, see the "Directory Memory Access (DMA)" */
      /* chapter in this book. */
    default:
      return (DDI_FAILURE);
    }
}
```

getinfo() エントリポイント (SCSI ターゲットドライバ)

SCSI ターゲットドライバの getinfo(9E) ルーチンは、ほかのドライバとほぼ同じです (DDI_INFO_DEVT2INSTANCE の場合の詳細については 116 ページの「getinfo() エントリポイント」を参照してください)。ただし、getinfo() ルーチンの DDI_INFO_DEVT2DEVINFO の場合、ターゲットドライバは dev_info ノードへのポインタを返す必要があります。このポインタは、ドライバの状態構造体に保存することも、scsi_device(9S) 構造体の sd_dev フィールドから取得することもできます。次の例は、代替の SCSI ターゲットドライバの getinfo() コードフラグメントを示しています。

例 17–4 代替 SCSI ターゲットドライバの getinfo() コードフラグメント

```
case DDI_INFO_DEVT2DEVINFO:
    dev = (dev_t)arg;
    instance = getminor(dev);
    xsp = ddi_get_soft_state(statep, instance);
    if (xsp == NULL)
        return (DDI_FAILURE);
    *result = (void *)xsp->sdp->sd_dev;
    return (DDI_SUCCESS);
```

リソース割り当て

SCSI コマンドをデバイスに送信するには、ターゲットドライバで scsi_pkt(9S) 構造体を作成して初期化する必要があります。次に、この構造体をホストバスアダプタドライバに渡す必要があります。

scsi_init_pkt() 関数

scsi_init_pkt(9F) ルーチンは scsi_pkt(9S) 構造体を割り当てて 0 に設定します。scsi_init_pkt() は、pkt_private、 *pkt_scbp、および *pkt_cdbp へのポインタも設定します。さらに、scsi_init_pkt () はリソースを使用できない場合を処理するコールバックメカニズムを提供します。この関数の構文は次のとおりです。

```
struct scsi_pkt *scsi_init_pkt(struct scsi_address *ap,
     struct scsi_pkt *pktp, struct buf *bp, int cmdlen,
     int statuslen, int privatelen, int flags,
     int (*callback)(caddr_t), caddr_t arg)
```

各表記の意味は次のとおりです。

ap scsi_address 構造体へのポインタです。ap は、デバイスの scsi_device(9S) 構造体の sd_address フィールドです。

pktp 初期化する scsi_pkt(9S) 構造体へのポインタです。このポインタが NULL に設定されている場合、新しいパケットが割り当てられます。

bp buf(9S) 構造対へのポインタです。このポインタが NULL ではなく、有効なバイトカウントがある場合、DMA リソースが割り当てられます。

cmdlen SCSI コマンド記述子ブロックの長さ (バイト単位) です。

statuslen SCSI ステータス完了ブロックの必要な長さ (バイト単位) です。

privatelen pkt_private フィールドのために割り当てるバイト数です。

flags フラグのセットです。

- PKT_CONSISTENT – scsi_alloc_consistent_buf(9F) を使用して DMA バッファーが割り当てられた場合は、このビットを設定する必要があります。この場合、ホストバスアダプタドライバによって、ターゲットドライバのコマンド完了コールバックを実行する前にデータ転送が適切に同期されることが保証されます。
- PKT_DMA_PARTIAL – ドライバが部分的な DMA マッピングを受け入れる場合、このビットを設定できます。設定されている場合、scsi_init_pkt(9F) は DDI_DMA_PARTIAL フラグを設定して、DMA リソースを割り当てます。scsi_pkt(9S) 構造体の pkt_resid フィールドは、ゼロ以外の未処理の内容を格納して復帰できます。ゼロ以外の値は、scsi_init_pkt(9F) が DMA リソースを割り当てることができなかったバイト数を示します。

callback リソースを使用できない場合に実行する処理を指定します。NULL_FUNC に設定されている場合、scsi_init_pkt(9F) はすぐに値 NULL を返します。SLEEP_FUNC に設定されている場合、リソースが使用可能になるまで

scsi_init_pkt() は復帰しません。その他すべての有効なカーネルアドレスは、リソースが使用可能になる可能性が高いときに呼び出される関数のアドレスとして解釈されます。

arg コールバック関数に渡されるパラメータです。

scsi_init_pkt() ルーチンはトランスポート前にデータを同期します。トランスポート後にドライバがデータにアクセスする必要がある場合は、ドライバで scsi_sync_pkt(9F) を呼び出して、中間キャッシュをすべてフラッシュする必要があります。scsi_sync_pkt () ルーチンを使用すると、キャッシュされているすべてのデータを同期できます。

scsi_sync_pkt() 関数

ターゲットドライバが、データの変更後にパケットを再送信する必要がある場合は、scsi_transport(9F) を呼び出す前に scsi_sync_pkt(9F) を呼び出す必要があります。ただし、ターゲットドライバがデータにアクセスする必要がない場合は、トランスポート後に scsi_sync_pkt() を呼び出す必要はありません。

scsi_destroy_pkt() 関数

scsi_destroy_pkt(9F) ルーチンは、必要な場合、パケットに関連付けられている、残りすべてのキャッシュデータを同期します。このルーチンは次に、パケットと、関連付けられているコマンド、ステータス、およびターゲットのドライバの非公開データ領域を解放します。このルーチンは、コマンド完了ルーチンで呼び出す必要があります。

scsi_alloc_consistent_buf() 関数

ほとんどの入出力要求で、ドライバのエントリポイントに渡されるデータバッファーは、ドライバから直接アクセスされることはありません。バッファーは scsi_init_pkt(9F) に渡されるのみです。ドライバがSCSI コマンドを送信し、その操作対象がドライバ自体が調べているバッファーである場合、バッファーの DMA に整合性が必要です。SCSI 要求検知コマンドがその良い例です。scsi_alloc_consistent_buf(9F) ルーチンは buf(9S) 構造体と、DMA が一定である操作に適したデータバッファーを割り当てます。HBA は、コマンド完了コールバックを実行する前に、必要なバッファーの同期をすべて実行します。

注 – scsi_alloc_consistent_buf(9F) は不足しているシステムリソースを使用します。そのため、 scsi_alloc_consistent_buf() は慎重に使用してください。

---

scsi_free_consistent_buf() 関数

scsi_free_consistent_buf(9F) は、buf(9S) 構造体と、scsi_alloc_consistent_buf(9F)によって割り当てられた、関連付けられているデータバッファーを解放します。例については、364 ページの「attach() エントリポイント (SCSI ターゲットドライバ)」と366 ページの「detach() エントリポイント (SCSI ターゲットドライバ)」を参照してください。

コマンドの構築とトランスポート

ホストバスアダプタドライバは、デバイスへのコマンドの転送を担当します。さらに、ドライバは低レベル SCSI プロトコルの処理も担当します。scsi_transport(9F) ルーチンは、転送のためにパケットをホストバスアダプタドライバに渡します。ターゲットドライバは、有効な scsi_pkt(9S) 構造体の作成を担当します。

コマンドの構築

ルーチン scsi_init_pkt(9F) は、この例で示すように、SCSI CDB に領域を割り当て、必要な場合には DMA リソースを割り当てて、pkt_flags フィールドを設定します。

```
pkt = scsi_init_pkt(&sdp->sd_address, NULL, bp,
CDB_GROUP0, 1, 0, 0, SLEEP_FUNC, NULL);
```

この例では、渡された buf(9S) 構造体ポインタで指定されているように DMA リソースを割り当てるとともに、新しいパケットを作成しています。SCSI CDB はグループ 0 (6 バイト) のコマンドに割り当てられています。pkt_flags フィールドは 0 に設定されていますが、pkt_private フィールドに割り当てられた領域はありません。scsi_init_pkt(9F) に対するこの呼び出しは、SLEEP_FUNC パラメータのため、使用できるリソースが現在ない場合はいつまでもリソースを待機します。

次のステップは、scsi_setup_cdb(9F) 関数を使用して SCSI CDB を初期化することです。

```
    if (scsi_setup_cdb((union scsi_cdb *)pkt->pkt_cdbp,
     SCMD_READ, bp->b_blkno, bp->b_bcount >> DEV_BSHIFT, 0) == 0)
     goto failed;
```

この例では、グループ 0 のコマンド記述子ブロックを構築しています。この例の pkt_cdbp フィールドには次のように値が格納されます。

- コマンド自体はバイト 0 に格納されます。コマンドはパラメータ SCMD_READ から設定されます。

- アドレスフィールドはバイト 1、バイト 2、およびバイト 3 の 0-4 ビット内です。アドレスは bp->b_blkno から設定されます。
- カウントフィールドはバイト 4 内です。カウントは最後のパラメータから設定されます。この場合、count は bp->b_bcount >> DEV_BSHIFT に設定されます。ここで DEV_BSHIFT はブロックの数に変換される転送のバイトカウントです。

注 – scsi_setup_cdb(9F) は、ターゲットデバイスの論理ユニット番号 (LUN) を、SCSI コマンドブロックのバイト 1 の 5-7 ビットで設定することをサポートしていません。この要件は SCSI-1 で定義されています。コマンドブロック内に LUN ビットが設定されている必要がある SCSI-1 デバイスの場合は、scsi_setup_cdb(9F) ではなく、makecom_g0(9F) または同等の関数を使用します。

SCSI CDB を初期化したら、パケット内にあるほかの 3 つのフィールドを初期化し、パケットへのポインタとして状態構造体内に格納します。

```
pkt->pkt_private = (opaque_t)bp;
pkt->pkt_comp = xxcallback;
pkt->pkt_time = 30;
xsp->pkt = pkt;
```

buf(9S) ポインタは、後から完了ルーチンで使用するため、pkt_private フィールドに保存されます。

ターゲット機能の設定

ターゲットドライバは scsi_ifsetcap(9F) を使用して、ホストバスアダプタドライバの機能を設定します。*cap* は名前と値の組で、NULL で終わる文字列と整数値から構成されます。機能の現在の値は、scsi_ifgetcap(9F) を使用して取得できます。scsi_ifsetcap(9F) では、バス上のすべてのターゲットに対して機能を設定できます。

ただし、一般に、ターゲットドライバによって所有されていないターゲットの機能を設定することは推奨されません。この運用方法は、HBA ドライバで汎用的にサポートされているわけではありません。切断と同期などの一部の機能は、デフォルトでは HBA ドライバによって設定できます。その他の機能は、ターゲットドライバによって明示的に設定する必要が生じる可能性があります。たとえば、wide-xfer とタグ付きキューイングはターゲットドライバで設定する必要があります。

コマンドのトランスポート

scsi_pkt(9S) 構造体に値が格納されたら、scsi_transport(9F) を使用して、構造体をバスアダプタドライバに渡します。

```
if (scsi_transport(pkt) != TRAN_ACCEPT) {
 bp->b_resid = bp->b_bcount;
 bioerror(bp, EIO);
 biodone(bp);
 }
```

scsi_transport(9F) からのほかの戻り値は次のとおりです。

- TRAN_BUSY – 指定されたターゲットのコマンドはすでに処理中です。
- TRAN_BADPKT – パケット内の DMA カウントが大きすぎるか、ホストバスアダプタドライバが何らかの理由でこのパケットを拒否しました。
- TRAN_FATAL_ERROR – ホストアダプタドライバはこのパケットを受け取ることができません。

注 – scsi_device(9S) 構造体内の sd_mutex という mutex は、scsi_transport(9F) の呼び出し全体にわたって保持してはいけません。

---

scsi_transport(9F) が TRAN_ACCEPT を返す場合、パケットはホストバスアダプタドライバで管理されるものになっています。パケットは、コマンド完了ルーチンが呼び出されるまで、ターゲットドライバからアクセスしてはいけません。

同期 scsi_transport() 関数

パケット内に FLAG_NOINTR が設定されている場合、コマンドが完了するまで scsi_transport(9F) は復帰しません。コールバックは実行されません。

注 – 割り込みのコンテキストで FLAG_NOINTR を使わないでください。

コマンドの完了

ホストバスアダプタドライバがコマンドを完了すると、ドライバはパケットの完了コールバックルーチンを呼び出します。ドライバは次に、scsi_pkt(9S) 構造体へのポインタをパラメータとして渡します。パケットのデコード後、完了ルーチンが適切な操作を実行します。

例 17–5は、簡単な完了コールバックルーチンを表しています。このコードは、トランスポートの失敗をチェックします。失敗の場合、ルーチンはコマンドを再試行す

るのではなく、実行を断念します。ターゲットがビジー状態の場合、あとでコマンドを再送信するために、追加のコードが必要です。

コマンドがチェック条件になった場合、要求の自動検知が有効になっている場合を除き、ターゲットドライバが要求検知コマンドを送信する必要があります。

その他の場合、コマンドは成功したことになります。コマンドの処理の最後に、コマンドはパケットを破棄し、biodone(9F) を呼び出します。

バスのリセットやパリティーの問題など、トランスポートエラーが発生した場合、ターゲットドライバは scsi_transport(9F) を使用してパケットを再送信できます。再送信の前に、パケット内の値を変更する必要はありません。

次の例では、完了しなかったコマンドの再試行は試みていません。

注 – 割り込みのコンテキストでは、通常、ターゲットドライバのコールバック関数が呼び出されます。結果としてコールバック関数は、スリープすることがあってはなりません。

例 17–5　SCSI ドライバの完了ルーチン

```
static void
xxcallback(struct scsi_pkt *pkt)
{
    struct buf        *bp;
    struct xxstate    *xsp;
    minor_t           instance;
    struct scsi_status *ssp;
    /*
     * Get a pointer to the buf(9S) structure for the command
     * and to the per-instance data structure.
     */
    bp = (struct buf *)pkt->pkt_private;
    instance = getminor(bp->b_edev);
    xsp = ddi_get_soft_state(statep, instance);
    /*
     * Figure out why this callback routine was called
     */
    if (pkt->pkt_reason != CMP_CMPLT) {
       bp->b_resid = bp->b_bcount;
       bioerror(bp, EIO);
       scsi_destroy_pkt(pkt);          /* Release resources */
       biodone(bp);                    /* Notify waiting threads */ ;
    } else {
       /*
        * Command completed, check status.
        * See scsi_status(9S)
        */
       ssp = (struct scsi_status *)pkt->pkt_scbp;
       if (ssp->sts_busy) {
           /* error, target busy or reserved */
       } else if (ssp->sts_chk) {
```

例 17–5 SCSI ドライバの完了ルーチン (続き)

```
            /* Send a request sense command. */
        } else {
         bp->b_resid = pkt->pkt_resid;  /* Packet completed OK */
         scsi_destroy_pkt(pkt);
         biodone(bp);
        }
    }
}
```

パケットの再利用

ターゲットドライバは次の方法でパケットを再利用できます。

- 変更されていないパケットを再送信します。
- scsi_sync_pkt(9F) を使用してデータを同期します。次に、ドライバでデータを処理します。最後に、パケットを再送信します。
- scsi_dmafree(9F) を使用して DMA リソースを解放し、pkt ポインタを scsi_init_pkt(9F) に渡して新しい bp にバインドします。ターゲットドライバがパケットの再初期化を行う必要があります。CDB の長さは前の CDB と同じになっている必要があります。
- scsi_init_pkt(9F) への最初の呼び出しで、不完全な DMA のみが割り当てられた場合、以後の scsi_init_pkt(9F) の呼び出しは同じパケットに対して行うことができます。bp に対しても呼び出しを行い、転送の次の部分に対する DMA リソースを調整できます。

自動要求検知モード

キューイングが使用される場合、タグ付きのキューキングであるか、タグなしのキューイングであるかにかかわらず、自動要求検知モードを使用することが推奨されます。CAC (Contingent Allegiance Condition) は後続のコマンドによってクリアされ、結果として検知データは失われます。ほとんどの HBA ドライバは、ターゲットドライバのコールバックを実行する前に次のコマンドを開始します。その他の HBA ドライバは、別の優先順位が低いスレッドを使用してコールバックを実行できます。この方法では、パケットがチェック条件で完了したことをターゲットドライバに通知するために必要な時間が増加する可能性があります。この場合、ターゲットドライバは、検知データの取得に間に合うように要求検知コマンドを送信できないことがあります。

この検知データの損失を回避するには、チェック条件が検出された場合に HBA ドライバまたはコントローラが要求検知コマンドを発行する必要があります。このモードは、自動要求検知モードと呼ばれます。すべての HBA ドライバが自動要求検

知モードに対応しているわけではなく、一部のドライバは自動要求検知モードが有効な場合にのみ動作可能であることに注意してください。

ターゲットドライバは、scsi_ifsetcap(9F) を使用して自動要求検知モードを有効にします。次に、自動要求検知を有効にする例を示します。

例 17–6 自動要求検知モードの有効化

```
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    struct xxstate *xsp;
    struct scsi_device *sdp = (struct scsi_device *)
    ddi_get_driver_private(dip);
    /*
     * Enable auto-request-sense. An auto-request-sense command might
     * fail due to a BUSY condition or transport error. Therefore,
     * it is recommended to allocate a separate request sense
     * packet as well.
     * Note that scsi_ifsetcap(9F) can return -1, 0, or 1
     */
    xsp->sdp_arq_enabled =
    ((scsi_ifsetcap(ROUTE, "auto-rqsense", 1, 1) == 1) ? 1 : 0);
    /*
     * If the HBA driver supports auto request sense then the
     * status blocks should be sizeof (struct scsi_arq_status).
     * Else, one byte is sufficient.
     */
    xsp->sdp_cmd_stat_size =  (xsp->sdp_arq_enabled ?
    sizeof (struct scsi_arq_status) : 1);
    /* ... */
}
```

scsi_init_pkt(9F) を使用してパケットが割り当てられていて、このパケットで自動要求検知が必要な場合、追加の領域が必要になります。ターゲットドライバは、自動要求検知構造体を保持するため、ステータスブロック用のこの領域を要求する必要があります。要求検知コマンドで使用される検知の長さは、struct scsi_extended_sense の sizeof です。ステータスブロックに対して struct scsi_status から sizeof を割り当てることで、個々のパケットごとに自動要求検知を無効にできます。

パケットは通常どおり、scsi_transport(9F) を使用して送信されます。このパケットでチェック条件が発生すると、ホストバスアダプタドライバは次のステップを実行します。

- コントローラが自動要求検知機能を備えていない場合は、要求検知コマンドを発行します。
- 検知データを取得します。
- パケットのステータスブロックに scsi_arq_status の情報を格納します。

- パケットの pkt_state フィールドに STATE_ARQ_DONE を設定します。
- パケットのコールバックハンドラ (pkt_comp()) を呼び出します。

ターゲットドライバのコールバックルーチンでは、pkt_state 内の STATE_ARQ_DONE ビットをチェックして、検知データを使用可能であることを検証する必要があります。STATE_ARQ_DONE は、チェック条件が発生したこと、および要求検知が実行されたことを意味します。パケットで自動要求検知が一時的に無効にされている場合、以降の検知データの取得を保証することはできません。

その後、ターゲットドライバで自動要求検知コマンドが正常に完了し、検知データがデコードされたかどうかを検証する必要があります。

ダンプの処理

dump(9E) エントリポイントは、システム障害やチェックポイント操作が発生した場合に、仮想アドレス空間の一部を、指定されたデバイスに直接コピーします。cpr(7) と dump(9E) のマニュアルページを参照してください。dump(9E) エントリポイントは、割り込みを使わずにこの操作を実行できる必要があります。

dump() の引数は次のとおりです。

dev ダンプデバイスのデバイス番号

addr ダンプを開始するカーネルの仮想アドレス

blkno デバイス上の最初の出力先ブロック

nblk ダンプするブロックの数

例 17–7 dump(9E) ルーチン

```
static int
xxdump(dev_t dev, caddr_t addr, daddr_t blkno, int nblk)
{
    struct xxstate     *xsp;
    struct buf         *bp;
    struct scsi_pkt    *pkt;
    int    rval;
    int    instance;

    instance = getminor(dev);
    xsp = ddi_get_soft_state(statep, instance);

    if (tgt->suspended) {
    (void) pm_raise_power(DEVINFO(tgt), 0, 1);
    }

    bp = getrbuf(KM_NOSLEEP);
    if (bp == NULL) {
    return (EIO);
```

例 17–7 dump(9E) ルーチン (続き)

```
    }

/* Calculate block number relative to partition. */

bp->b_un.b_addr = addr;
    bp->b_edev = dev;
    bp->b_bcount = nblk * DEV_BSIZE;
    bp->b_flags = B_WRITE | B_BUSY;
    bp->b_blkno = blkno;

    pkt = scsi_init_pkt(ROUTE(tgt), NULL, bp, CDB_GROUP1,
    sizeof (struct scsi_arq_status),
    sizeof (struct bst_pkt_private), 0, NULL_FUNC, NULL);
    if (pkt == NULL) {
    freerbuf(bp);
    return (EIO);
    }
    (void) scsi_setup_cdb((union scsi_cdb *)pkt->pkt_cdbp,
        SCMD_WRITE_G1, blkno, nblk, 0);
    /*
     * While dumping in polled mode, other cmds might complete
     * and these should not be resubmitted. we set the
     * dumping flag here which prevents requeueing cmds.
     */
    tgt->dumping = 1;
    rval = scsi_poll(pkt);
    tgt->dumping = 0;

    scsi_destroy_pkt(pkt);
    freerbuf(bp);

    if (rval != DDI_SUCCESS) {
    rval = EIO;
    }

    return (rval);
}
```

SCSI オプション

SCSA では、制御とデバッグのためにグローバル変数の *scsi_options* が定義されています。*scsi_options* で定義されているビットは `<sys/scsi/conf/autoconf.h>` ファイルに記載されています。*scsi_options* では次のようにビットを使用します。

`SCSI_OPTIONS_DR` グローバルな切断または再接続を有効にします。

`SCSI_OPTIONS_FAST` グローバルな FAST SCSI サポートを有効にします。転送速度は 10MB/s です。HBA は、SCSI_OPTIONS_FAST (0x100) ビットが設定されていない場合は FAST SCSI モードで動作してはいけません。

SCSI_OPTIONS_FAST20	グローバルな FAST20 SCSI サポートを有効にします。転送速度は 20MB/s です。HBA は、SCSI_OPTIONS_FAST20 (0x400) ビットが設定されていない場合は FAST20 SCSI モードで動作してはいけません。
SCSI_OPTIONS_FAST40	グローバルな FAST40 SCSI サポートを有効にします。転送速度は 40MB/s です。HBA は、SCSI_OPTIONS_FAST40 (0x800) ビットが設定されていない場合は FAST40 SCSI モードで動作してはいけません。
SCSI_OPTIONS_FAST80	グローバルな FAST80 SCSI サポートを有効にします。転送速度は 80MB/s です。HBA は、SCSI_OPTIONS_FAST80 (0x1000) ビットが設定されていない場合は FAST80 SCSI モードで動作してはいけません。
SCSI_OPTIONS_FAST160	グローバルな FAST160 SCSI サポートを有効にします。転送速度は 160MB/s です。HBA は、SCSI_OPTIONS_FAST160 (0x2000) ビットが設定されていない場合は FAST160 SCSI モードで動作してはいけません。
SCSI_OPTIONS_FAST320	グローバルな FAST320 SCSI サポートを有効にします。転送速度は 320MB/s です。HBA は、SCSI_OPTIONS_FAST320 (0x4000) ビットが設定されていない場合は FAST320 SCSI モードで動作してはいけません。
SCSI_OPTIONS_LINK	グローバルなリンクサポートを有効にします。
SCSI_OPTIONS_PARITY	グローバルなパリティーサポートを有効にします。
SCSI_OPTIONS_QAS	クイックアービトレーション選択 (Quick Arbitration Select) 機能を有効にします。QAS は、デバイスがバスのアービトレーションを行ってバスにアクセスするときに、プロトコルのオーバーヘッドを軽減するために使用されます。QAS は Ultra4 (FAST160) SCSI デバイスでのみサポートされます。ただし、すべての Ultra4 SCSI デバイスが QAS をサポートしているわけではありません。HBA は、SCSI_OPTIONS_QAS (0x100000) ビットが設定されていない場合は QAS SCSI モードで動作してはいけません。使用しているマシンが QAS をサポートするかどうかは、適切な Oracle のハードウェアドキュメントで確認してください。
SCSI_OPTIONS_SYNC	グローバルな同期転送機能を有効にします。
SCSI_OPTIONS_TAG	グローバルなタグ付きキューイングのサポートを有効にします。
SCSI_OPTIONS_WIDE	グローバルな WIDE SCSI を有効にします。

注 – *scsi_options* の設定は、システム上に存在するすべてのホストバスアダプタドライバと、すべてのターゲットドライバに影響します。特定のホストアダプタに対するこれらのオプションの制御については、`scsi_hba_attach(9F)` のマニュアルページを参照してください。

◆ ◆ ◆ 第 18 章

18 SCSI ホストバスアダプタドライバ

この章では、SCSI ホストバスアダプタ (Host Bus Adapter、HBA) ドライバの作成について説明します。この章には、一般的な HBA ドライバの構造体を示すサンプルコードが記載されています。このサンプルコードは、SCSA (Sun Common SCSI Architecture) が提供する HBA ドライバインタフェースの使い方を示しています。この章では、次の内容について説明します。

ホストバスアダプタドライバの概要

第 17 章「SCSI ターゲットドライバ」で説明しているように、DDI/DKI では、SCSI デバイスへのソフトウェアインタフェースが次の大きな 2 つの部分に分割されています。

- ターゲットデバイスおよびドライバ
- ホストバスアダプタデバイスおよびドライバ

ターゲットデバイスは、ディスクやテープドライブなどの SCSI バス上のデバイスを指します。ターゲットドライバは、デバイスドライバとしてインストールされるソフトウェアコンポーネントを指します。SCSI バス上の各ターゲットデバイスは、ターゲットドライバの 1 つのインスタンスで制御されます。

ホストバスアダプタデバイスは、SBus や PCI SCSI アダプタカードなどの HBA ハードウェアのことです。ホストバスアダプタドライバは、デバイスドライバとしてインストールされるソフトウェアコンポーネントのことです。たとえば、SPARC マシン

上の esp ドライバ、x86 マシン上の ncrs ドライバ、どちらのアーキテクチャーでも動作する isp ドライバがこれに該当します。HBA ドライバのインスタンスは、システムに構成されているそれぞれのホストバスアダプタデバイスを制御します。

SCSA (Sun Common SCSI Architecture) は、ターゲットコンポーネントと HBA コンポーネント間のインタフェースを定義します。

注 – SCSI ターゲットドライバを理解しておくと、効果的な SCSI HBA ドライバを作成できます。SCSI ターゲットドライバについては、第 17 章「SCSI ターゲットドライバ」を参照してください。ターゲットドライバの開発者にとっても、この章を読むことで利点を得ることができます。

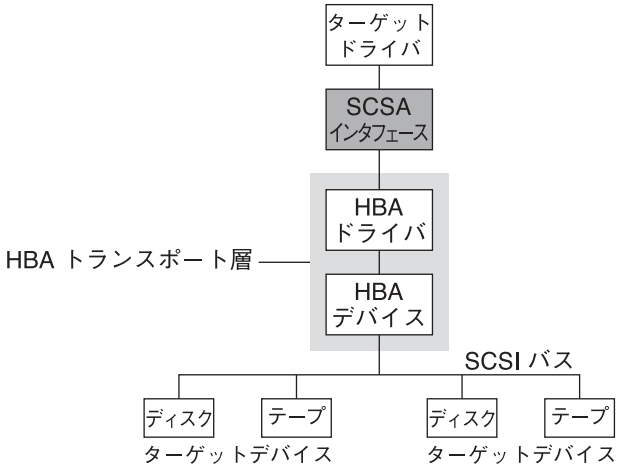
ホストバスアダプタドライバは、次のタスクを実行します。

- ホストバスアダプタハードウェアの管理
- SCSI ターゲットドライバからの SCSI コマンドの受け入れ
- 指定された SCSI ターゲットデバイスへのそれらのコマンドのトランスポート
- コマンドによって要求されるすべてのデータ転送の実行
- ステータスの収集
- 自動要求検知の処理 (オプション)
- ターゲットドライバへのコマンドの完了または失敗の通知

SCSI インタフェース

SCSA は、ターゲットドライバからホストアダプタドライバに SCSI コマンドを転送するための DDI/DKI プログラミングインタフェースです。SCSA に従うことで、ターゲットドライバは SCSI コマンドとシーケンスのどのような組み合わせでも簡単にターゲットデバイスに渡すことができます。ホストアダプタのハードウェア実装の知識は必要ありません。概念上、SCSA は SCSI コマンドを構築することと、データとともにコマンドを SCSI バスにトランスポートすることを区別しています。SCSA は、次の図に示すように、HBA トランスポート層を通じて、ターゲットドライバと HBA ドライバ間の接続を管理します。

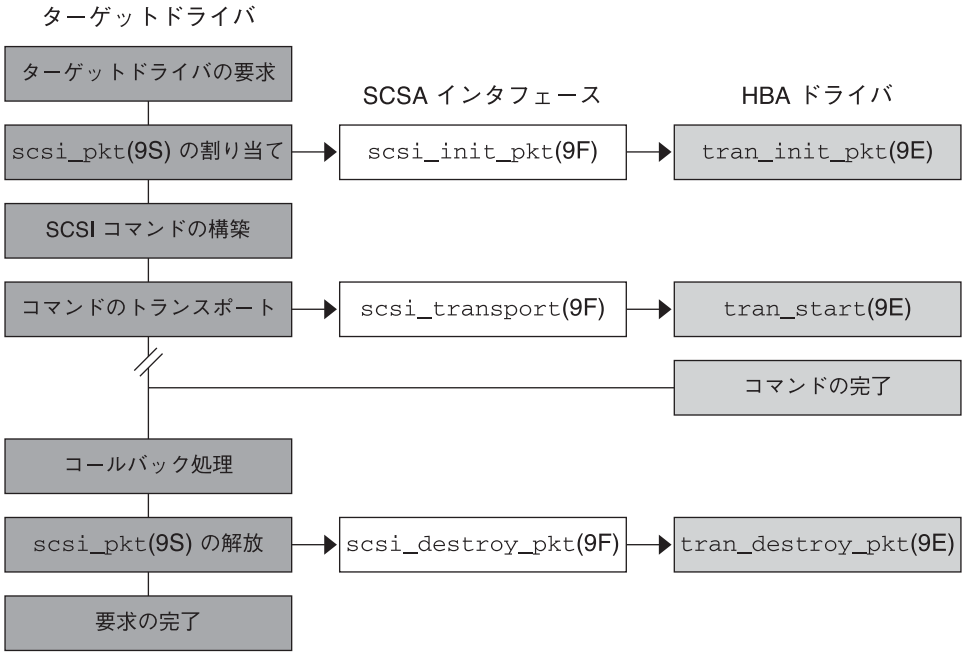
図 18–1 SCSA インタフェース



HBA トランスポート層は、SCSI コマンドの SCSI ターゲットデバイスへのトランスポートを担当するソフトウェアとハードウェアの層です。HBA ドライバは、SCSA 経由で SCSI ターゲットドライバが発行する要求に応えて、リソースの割り当て、DMA 管理、およびトランスポートの各サービスを提供します。また、コマンドの実行に必要なホストアダプタハードウェアと SCSI プロトコルの管理も行います。コマンドが完了すると、HBA ドライバはターゲットドライバの SCSI `pkt` コマンド完了ルーチンを呼び出します。

次の例は、ターゲットドライバから SCSA、さらに HBA ドライバへの転送に重点を置いて、この流れを示しています。この図には、一般的なトランスポートのエントリポイントと関数呼び出しも示されています。

図 18–2 トランスポート層の流れ



SCSA HBA インタフェース

SCSA HBA インタフェースには、HBA エントリポイント、HBA データ構造体、および HBA フレームワークが含まれています。

SCSA HBA エントリポイントのサマリー

SCSA では、HBA ドライバのエントリポイントをいくつか定義しています。次の表に、これらのエントリポイントを示します。このエントリポイントは、HBA ドライバに接続されたターゲットドライバインスタンスが構成されるときにシステムから呼び出されます。また、ターゲットドライバが SCSA 要求を出したときにも呼び出されます。詳細については、402 ページの「SCSA HBA ドライバのエントリポイント」を参照してください。

表 18–1 SCSA HBA エントリポイントのサマリー

関数名	呼び出される原因となる操作
tran_abort(9E)	ターゲットドライバによる scsi_abort(9F) の呼び出し
tran_bus_reset(9E)	システムによるバスのリセット

表 18–1　SCSA HBA エントリポイントのサマリー　(続き)

関数名	呼び出される原因となる操作
tran_destroy_pkt(9E)	ターゲットドライバによる scsi_destroy_pkt(9F) の呼び出し
tran_dmafree(9E)	ターゲットドライバによる scsi_dmafree(9F) の呼び出し
tran_getcap(9E)	ターゲットドライバによる scsi_ifgetcap(9F) の呼び出し
tran_init_pkt(9E)	ターゲットドライバによる scsi_init_pkt(9F) の呼び出し
tran_quiesce(9E)	システムによるバスの休止
tran_reset(9E)	ターゲットドライバによる scsi_reset(9F) の呼び出し
tran_reset_notify(9E)	ターゲットドライバによる scsi_reset_notify(9F) の呼び出し
tran_setcap(9E)	ターゲットドライバによる scsi_ifsetcap(9F) の呼び出し
tran_start(9E)	ターゲットドライバによる scsi_transport(9F) の呼び出し
tran_sync_pkt(9E)	ターゲットドライバによる scsi_sync_pkt(9F) の呼び出し
tran_tgt_free(9E)	システムによるターゲットデバイスインスタンスの切り離し
tran_tgt_init(9E)	システムによるターゲットデバイスインスタンスの接続
tran_tgt_probe(9E)	ターゲットドライバによる scsi_probe(9F) の呼び出し
tran_unquiesce(9E)	システムによるバス上の動作の再開

SCSA HBA データ構造体

SCSA では、データ構造体を定義して、ターゲットドライバと HBA ドライバ間で情報交換ができるようにしています。含まれているデータ構造体は次のとおりです。

- scsi_hba_tran(9S)
- scsi_address(9S)
- scsi_device(9S)
- scsi_pkt(9S)

scsi_hba_tran() 構造体

HBA ドライバの各インスタンスでは、scsi_hba_tran(9S) エントリポイントで scsi_hba_tran_alloc(9F) 関数を使用して attach(9E) 構造体を割り当てる必要があります。scsi_hba_tran_alloc() 関数は、scsi_hba_tran 構造体を初期化します。HBA ドライバは、HBA ドライバ内のエントリポイントを指すように、トランスポート構造体にある特定のベクトルを初期化する必要があります。scsi_hba_tran 構造体が初期化されたあとで、HBA ドライバは scsi_hba_attach_setup(9F) 関数を呼び出して、このトランスポート構造体を SCSA にエクスポートします。

注意 – SCSA ではトランスポート構造体を指すポインタを devinfo ノード上のドライバの非公開フィールドに保持するため、HBA ドライバは ddi_set_driver_private(9F) を使用してはいけません。ただし、ddi_get_driver_private(9F) を使用して、トランスポート構造体を指すポインタを検出することはできます。

SCSA インタフェースでは、HBA ドライバは scsi_hba_tran 構造体を介して呼び出し可能ないくつかのエントリポイントを提供する必要があります。詳細については、402 ページの「SCSA HBA ドライバのエントリポイント」を参照してください。

scsi_hba_tran 構造体には、次のフィールドがあります。

```
struct scsi_hba_tran {
    dev_info_t          *tran_hba_dip;          /* HBAs dev_info pointer */
    void                *tran_hba_private;      /* HBA softstate */
    void                *tran_tgt_private;      /* HBA target private pointer */
    struct scsi_device  *tran_sd;               /* scsi_device */
    int                 (*tran_tgt_init)();     /* Transport target */
                                                /* Initialization */
    int                 (*tran_tgt_probe)();    /* Transport target probe */
    void                (*tran_tgt_free)();     /* Transport target free */
    int                 (*tran_start)();        /* Transport start */
    int                 (*tran_reset)();        /* Transport reset */
    int                 (*tran_abort)();        /* Transport abort */
    int                 (*tran_getcap)();       /* Capability retrieval */
    int                 (*tran_setcap)();       /* Capability establishment */
    struct scsi_pkt     *(*tran_init_pkt)();    /* Packet and DMA allocation */
    void                (*tran_destroy_pkt)();  /* Packet and DMA */
                                                /* Deallocation */
    void                (*tran_dmafree)();      /* DMA deallocation */
    void                (*tran_sync_pkt)();     /* Sync DMA */
    void                (*tran_reset_notify)(); /* Bus reset notification */
    int                 (*tran_bus_reset)();    /* Reset bus only */
    int                 (*tran_quiesce)();      /* Quiesce a bus */
    int                 (*tran_unquiesce)();    /* Unquiesce a bus */
    int                 tran_interconnect_type; /* transport interconnect */
};
```

次に、scsi_hba_tran 構造体の各フィールドについて詳しく説明します。

tran_hba_dip	HBA デバイスインスタンスの dev_info 構造体を指すポインタです。このフィールドは、scsi_hba_attach_setup(9F) 関数によって設定されます。
tran_hba_private	HBA ドライバによって保持される非公開データを指すポインタです。通常、tran_hba_private には、HBA ドライバの状態構造体を指すポインタが入ります。
tran_tgt_private	複製の使用時に HBA ドライバによって保持される非公開データを指すポインタです。scsi_hba_attach_setup(9F) の呼び出し時に SCSI_HBA_TRAN_CLONE を指定することにより、scsi_hba_tran(9S) 構造体が 1 つのターゲットにつき 1 度だけ複製されます。この方法により、HBA は tran_tgt_init(9E) エントリポイントでターゲットインスタンスごとのデータ構造体を指すようにこのフィールドを初期化できます。SCSI_HBA_TRAN_CLONE を指定しない場合、tran_tgt_private は NULL となり、tran_tgt_private が参照されることはありません。詳細については、393 ページの「トランスポート構造体の複製」を参照してください。
tran_sd	複製時に使用されるターゲットインスタンスごとの scsi_device(9S) 構造体を指すポインタです。SCSI_HBA_TRAN_CLONE が scsi_hba_attach_setup(9F) に渡された場合、tran_sd はターゲットごとの scsi_device 構造体を指すように初期化されます。この初期化は、そのターゲットに代わって HBA 関数が呼び出される前に行われます。SCSI_HBA_TRAN_CLONE を指定しない場合、tran_sd は NULL となり、tran_sd が参照されることはありません。詳細については、393 ページの「トランスポート構造体の複製」を参照してください。
tran_tgt_init	ターゲットデバイスインスタンスの初期化時に呼び出される HBA ドライバのエントリポイントを指すポインタです。ターゲットごとの初期化が必要ない場合、HBA は tran_tgt_init を NULL に設定しておくことができます。
tran_tgt_probe	ターゲットドライバインスタンスが scsi_probe(9F) を呼び出したときに呼び出される HBA ドライバのエントリポイントを指すポインタです。このルーチンは、ターゲットデバイスの存在を調べるために呼び出されます。この HBA にターゲットのプローブカスタマイズが必要ない場合、HBA は tran_tgt_probe を scsi_hba_probe(9F) に設定します。

tran_tgt_free	ターゲットデバイスのインスタンスが破棄されるときに呼び出される HBA ドライバのエントリポイントを指すポインタです。ターゲットごとの解放が必要ない場合、HBA は tran_tgt_free を NULL に設定しておくことができます。
tran_start	ターゲットドライバが scsi_transport(9F) を呼び出したときに呼び出される HBA ドライバのエントリポイントを指すポインタです。
tran_reset	ターゲットドライバが scsi_reset(9F) を呼び出したときに呼び出される HBA ドライバのエントリポイントを指すポインタです。
tran_abort	ターゲットドライバが scsi_abort(9F) を呼び出したときに呼び出される HBA ドライバのエントリポイントを指すポインタです。
tran_getcap	ターゲットドライバが scsi_ifgetcap(9F) を呼び出したときに呼び出される HBA ドライバのエントリポイントを指すポインタです。
tran_setcap	ターゲットドライバが scsi_ifsetcap(9F) を呼び出したときに呼び出される HBA ドライバのエントリポイントを指すポインタです。
tran_init_pkt	ターゲットドライバが scsi_init_pkt(9F) を呼び出したときに呼び出される HBA ドライバのエントリポイントを指すポインタです。
tran_destroy_pkt	ターゲットドライバが scsi_destroy_pkt(9F) を呼び出したときに呼び出される HBA ドライバのエントリポイントを指すポインタです。
tran_dmafree	ターゲットドライバが scsi_dmafree(9F) を呼び出したときに呼び出される HBA ドライバのエントリポイントを指すポインタです。
tran_sync_pkt	ターゲットドライバが scsi_sync_pkt(9F) を呼び出したときに呼び出される HBA ドライバのエントリポイントを指すポインタです。
tran_reset_notify	ターゲットドライバが tran_reset_notify(9E) を呼び出したときに呼び出される HBA ドライバのエントリポイントを指すポインタです。
tran_bus_reset	ターゲットをリセットしないで SCSI バスをリセットする関数エントリです。

tran_quiesce	未処理のコマンドが完了するのを待ち、発行されたすべての入出力要求をブロックする (またはキューに入れる) 関数エントリです。
tran_unquiesce	入出力動作が SCSI バス上で再開できるようにする関数エントリです。
tran_interconnect_type	services.h ヘッダーファイルに定義されたとおりにトランスポートの相互接続タイプを示す整数値です。

scsi_address 構造体

scsi_address(9S) 構造体は、ターゲットドライバのインスタンスによって割り当てられ、トランスポートされる各 SCSI コマンドのトランスポートとアドレス指定の情報を提供します。

scsi_address 構造体には、次のフィールドがあります。

```
struct scsi_address {
    struct scsi_hba_tran    *a_hba_tran;    /* Transport vectors */
    ushort_t                a_target;       /* Target identifier */
    uchar_t                 a_lun;          /* LUN on that target */
    uchar_t                 a_sublun;       /* Sub LUN on that LUN */
                                            /* Not used */
};
```

a_hba_tran	HBA ドライバによって割り当てられ、初期化されたときの scsi_hba_tran(9S) 構造体を指すポインタです。scsi_hba_attach_setup(9F) のフラグとして SCSI_HBA_TRAN_CLONE を指定した場合、a_hba_tran はその構造体のコピーを指します。
a_target	SCSI バス上の SCSI ターゲットを識別します。
a_lun	SCSI ターゲット上の SCSI 論理ユニットを識別します。

scsi_device 構造体

HBA フレームワークでは、ターゲットデバイスのインスタンスごとに scsi_device(9S) 構造体を割り当て、初期化します。この割り当てと初期化は、フレームワークで HBA ドライバの tran_tgt_init(9E) エントリポイントを呼び出す前に行われます。この構造体には、一般的な情報とデバイス固有の情報を含む情報領域を指すポインタなど、各 SCSI 論理ユニットに関する情報が格納されます。システムに接続されているターゲットデバイスインスタンスごとに 1 つの scsi_device(9S) 構造体が存在します。

ターゲットごとの初期化が正常に行われると、HBA フレームワークは ddi_set_driver_private(9F) を使用して、ターゲットドライバのインスタンスごとの

非公開データが scsi_device(9S) 構造体を指すように設定します。初期化が正常に行われるのは、tran_tgt_init() が成功を返した場合またはベクトルが null の場合です。

scsi_device(9S) 構造体には、次のフィールドがあります。

```
struct scsi_device {
    struct scsi_address           sd_address;    /* routing information */
    dev_info_t                    *sd_dev;       /* device dev_info node */
    kmutex_t                      sd_mutex;      /* mutex used by device */
    void                          *sd_reserved;
    struct scsi_inquiry           *sd_inq;
    struct scsi_extended_sense    *sd_sense;
    caddr_t                       sd_private;    /* for driver's use */
};
```

各表記の意味は次のとおりです。

sd_address　SCSI リソースの割り当てのためのルーチンに渡されるデータ構造体です。

sd_dev　ターゲットの dev_info 構造体を指すポインタです。

sd_mutex　ターゲットドライバが使用する mutex です。この mutex は、HBA フレームワークによって初期化されます。ターゲットドライバは、この mutex をデバイスごとの mutex として使用できます。この mutex は、scsi_transport(9F) または scsi_poll(9F) が呼び出されると保持されません。mutex の詳細については、第 3 章「マルチスレッド」を参照してください。

sd_inq　ターゲットデバイスの SCSI 照会データへのポインタです。scsi_probe(9F) ルーチンは、バッファーを割り当て、そのバッファーを満たして、このフィールドに追加します。

sd_sense　デバイスから送られた要求検知データを収めるバッファーを指すポインタです。ターゲットドライバは、このバッファーそのものを割り当てて管理する必要があります。詳細については、109 ページの「attach() エントリポイント」 に記載されたターゲットドライバの attach(9E) ルーチンを参照してください。

sd_private　ターゲットドライバが使用するポインタフィールドです。このフィールドは、ターゲットドライバの非公開の状態構造体を指すポインタの格納によく使われます。

scsi_pkt 構造体 (HBA)

SCSI コマンドを実行するには、ターゲットドライバはまずそのコマンドに scsi_pkt(9S) 構造体を割り当てる必要があります。次に、そのターゲットドライバ専用の非公開データ領域の大きさ、コマンドのステータス、およびコマンドの長さを

指定する必要があります。HBA ドライバは、tran_init_pkt(9E) エントリポイントでパケット割り当てを実装します。また、その tran_destroy_pkt(9E) エントリポイントでパケットの解放を行います。詳細については、359 ページの「scsi_pkt 構造体 (ターゲットドライバ)」を参照してください。

scsi_pkt(9S) 構造体には、次のフィールドがあります。

```
struct scsi_pkt {
    opaque_t pkt_ha_private;             /* private data for host adapter */
    struct scsi_address pkt_address;     /* destination address */
    opaque_t pkt_private;                /* private data for target driver */
    void (*pkt_comp)(struct scsi_pkt *); /* completion routine */
    uint_t  pkt_flags;                   /* flags */
    int     pkt_time;                    /* time allotted to complete command */
    uchar_t *pkt_scbp;                   /* pointer to status block */
    uchar_t *pkt_cdbp;                   /* pointer to command block */
    ssize_t pkt_resid;                   /* data bytes not transferred */
    uint_t  pkt_state;                   /* state of command */
    uint_t  pkt_statistics;              /* statistics */
    uchar_t pkt_reason;                  /* reason completion called */
};
```

各表記の意味は次のとおりです。

pkt_ha_private　コマンドごとの HBA ドライバの非公開データを指すポインタです。

pkt_address　このコマンドのアドレス情報を提供する scsi_address(9S) 構造体を指すポインタです。

pkt_private　パケットごとのターゲットドライバの非公開データを指すポインタです。

pkt_comp　トランスポート層でこのコマンドが実行されたときに HBA ドライバによって呼び出されるターゲットドライバ完了ルーチンを指すポインタです。

pkt_flags　コマンドのフラグです。

pkt_time　コマンドの完了タイムアウトを秒単位で指定します。

pkt_scbp　コマンドのステータス完了ブロックを指すポインタです。

pkt_cdbp　コマンドのコマンド記述子ブロック (CDB) を指すポインタです。

pkt_resid　コマンドが完了したときに転送されなかったデータバイト数です。このフィールドを使用すると、リソースが割り当てられなかったデータの量も指定できます。HBA は、トランスポート時にこのフィールドを変更する必要があります。

pkt_state　コマンドの状態です。HBA は、トランスポート時にこのフィールドを変更する必要があります。

pkt_statistics	トランスポート層に存在している間に、コマンドが検出したイベントの履歴を提供します。HBA は、トランスポート時にこのフィールドを変更する必要があります。
pkt_reason	コマンド完了の理由です。HBA は、トランスポート時にこのフィールドを変更する必要があります。

ターゲットインスタンスごとのデータ

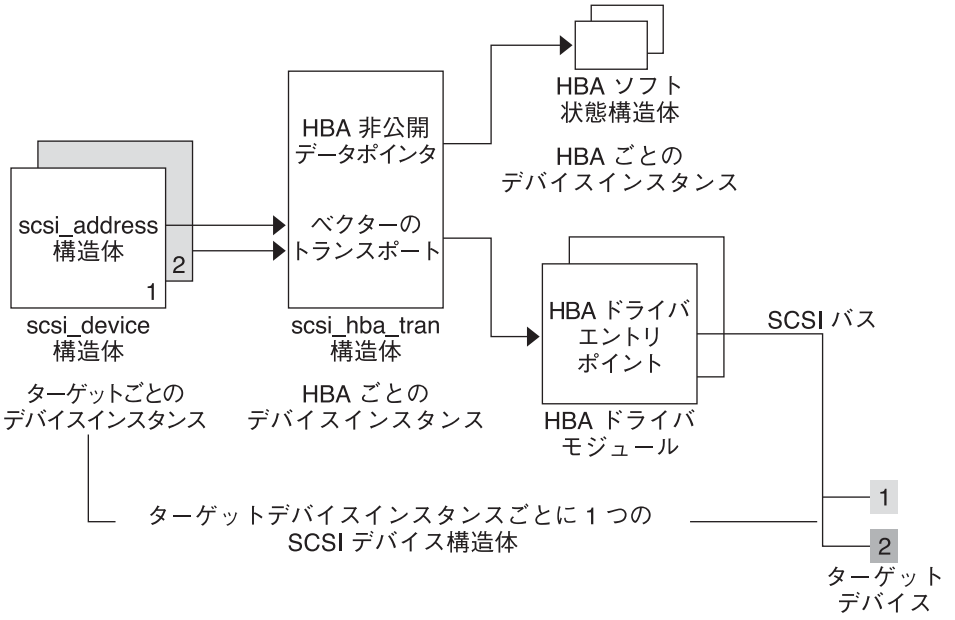
HBA ドライバは、attach(9E) の実行中に scsi_hba_tran(9S) 構造体を割り当てる必要があります。次に、HBA ドライバは、HBA ドライバの必須のエントリポイントを指すように、このトランスポート構造体にあるベクトルを初期化する必要があります。その後、この scsi_hba_tran 構造体は scsi_hba_attach_setup(9F) に渡されます。

scsi_hba_tran 構造体には tran_hba_private フィールドがあり、HBA ドライバのインスタンスごとの状態を参照するために使用できます。

各 scsi_address(9S) 構造体には、scsi_hba_tran 構造体を指すポインタが含まれています。また、scsi_address 構造体には、特定のターゲットデバイスのターゲット (a_target) と論理ユニット (a_lun) のアドレスもあります。HBA ドライバの各エントリポイントには、scsi_address 構造体を指すポインタが直接渡されるか、scsi_device(9S) 構造体を経由して間接的に渡されます。その結果、HBA ドライバは自身の状態を参照できます。HBA ドライバは、アドレス指定されているターゲットデバイスを識別することもできます。

次の図は、トランスポート操作のための HBA データ構造体を示しています。

図 18–3 HBA トランスポート構造体



トランスポート構造体の複製

HBA ドライバが scsi_hba_tran(9S) 構造体でターゲットごとの非公開データを保持する必要がある場合は、複製が役立つことがあります。複製は、scsi_address(9S) 構造体で提供されるアドレスよりも複雑なアドレスを保持するためにも使用できます。

複製プロセスでは、HBA ドライバは引き続き attach(9E) の実行時に scsi_hba_tran 構造体を割り当てる必要があります。また、HBA ドライバの tran_hba_private ソフト状態ポインタとエントリポイントベクトルを初期化する必要もあります。違いが生じるのは、フレームワークがターゲットドライバのインスタンスを HBA ドライバに接続開始するときです。HBA ドライバの tran_tgt_init(9E) エントリポイントを呼び出す前に、フレームワークでは HBA のそのインスタンスに関連付けられている scsi_hba_tran 構造体を複製します。その結果、特定のターゲットデバイスインスタンスに割り当てられ、初期化された各 scsi_address 構造体は、scsi_hba_tran 構造体のターゲットインスタンスごとのコピーを指します。scsi_address 構造体は、attach() の実行時に HBA ドライバによって割り当てられた scsi_hba_tran 構造体を指しません。

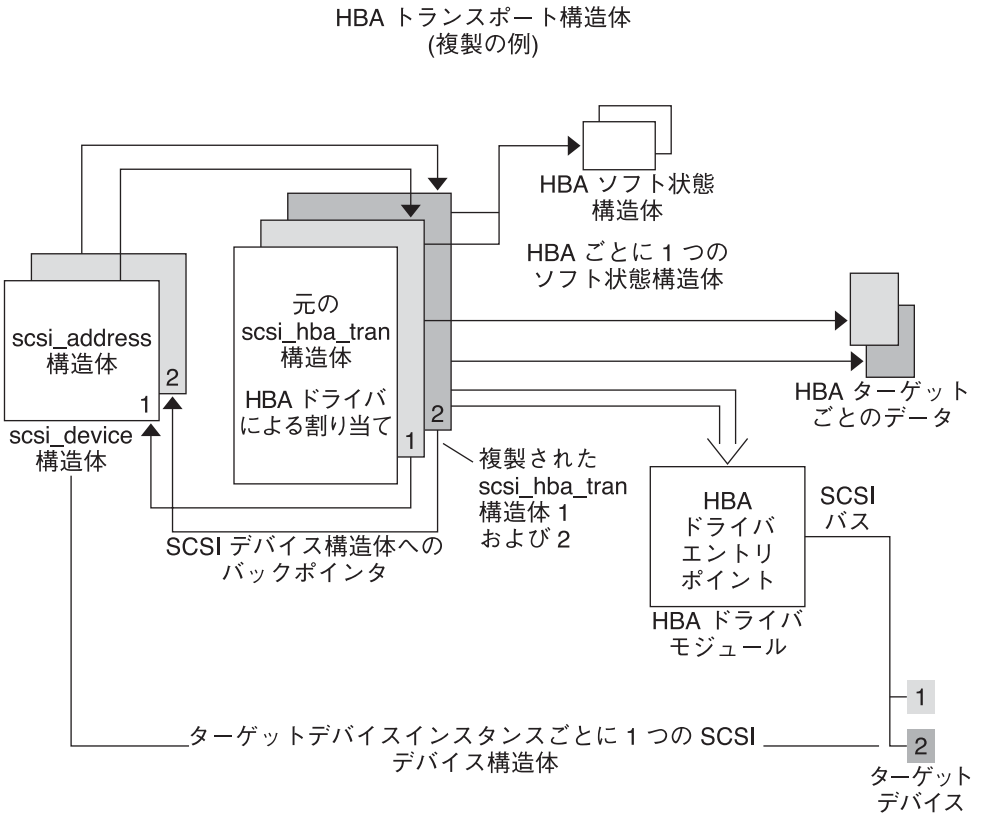
複製の指定時には、HBA ドライバは 2 つの重要なポインタを使用できます。これらのポインタは、scsi_hba_tran 構造体に含まれています。最初のポインタは tran_tgt_private フィールドであり、ドライバがターゲットごとの HBA 非公開データを指すために使用できます。 tran_tgt_private ポインタは、HBA ドライバが a_target や a_lun で提供されるアドレスよりも複雑なアドレスを保持する必要がある

場合などに役立ちします。2 番目のポインタは tran_sd フィールドであり、これは特定のターゲットデバイスを参照する scsi_device(9S) 構造体を指すポインタです。

複製の指定時は、HBA ドライバはターゲットごとのデータを割り当て、初期化する必要があります。次に、HBA ドライバは、その tran_tgt_init(9E) エントリポイントの実行中にこのデータを指すように tran_tgt_private フィールドを初期化する必要があります。HBA ドライバは、その tran_tgt_free(9E) エントリポイントの実行中に、このターゲットごとのデータを解放する必要があります。

複製時にフレームワークでは、HBA ドライバの tran_tgt_init() エントリポイントが呼び出される前に、scsi_device 構造体を指すように tran_sd フィールドを初期化します。ドライバが複製を要求するときは、SCSI_HBA_TRAN_CLONE フラグを scsi_hba_attach_setup(9F) に渡します。次の図は、トランスポート操作を複製するための HBA データ構造体を示しています。

図 18–4 トランスポート操作の複製



SCSA HBA 関数

SCSA では、いくつかの関数も提供しています。これらの関数は次の表に一覧表示されており、HBA ドライバで使用されます。

表 18–2 SCSA HBA 関数

関数名	呼び出し元のドライバエントリポイント
scsi_hba_init(9F)	_init(9E)
scsi_hba_fini(9F)	_fini(9E)
scsi_hba_attach_setup(9F)	attach(9E)
scsi_hba_detach(9F)	detach(9E)
scsi_hba_tran_alloc(9F)	attach(9E)
scsi_hba_tran_free(9F)	detach(9E)
scsi_hba_probe(9F)	tran_tgt_probe(9E)
scsi_hba_pkt_alloc(9F)	tran_init_pkt(9E)
scsi_hba_pkt_free(9F)	tran_destroy_pkt(9E)
scsi_hba_lookup_capstr(9F)	tran_getcap(9E) および tran_setcap(9E)

HBA ドライバの依存性と構成に関する問題

開発者は、SCSA HBA のエントリポイント、構造体、および関数をドライバに組み込むだけでなく、ドライバの依存性と構成に関する問題にも対処する必要があります。これらの問題は、構成プロパティー、依存性宣言、状態構造体とコマンド別構造体、モジュール初期化用のエントリポイント、および自動構成エントリポイントに関係しています。

宣言と構造体

HBA ドライバには、次のヘッダーファイルを含める必要があります。

```
#include <sys/scsi/scsi.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

モジュールが SCSA ルーチンに依存していることをシステムに通知するには、次のコマンドを使用してドライバのバイナリを生成する必要があります。SCSA ルーチンの詳細については、384 ページの「SCSA HBA インタフェース」を参照してください。

```
% ld -r xx.o -o xx -N "misc/scsi"
```

サンプルコードは、QLogic Intelligent SCSI Peripheral デバイス用の簡略化された isp ドライバから入手できます。isp ドライバは、最大 15 台のターゲットデバイスを接続可能で、1 つのターゲットに 8 台までの論理ユニット (LUN) を接続できるワイド SCSI をサポートしています。

コマンド別構造体

HBA ドライバは通常、ターゲットドライバが実行するコマンドごとに状態を保持する構造体を定義する必要があります。このコマンド別構造体のレイアウトは、デバイスドライバの作成者がすべて管理します。レイアウトに必要なのは、ドライバで使用されるハードウェアの機能とソフトウェアのアルゴリズムを反映させることです。

コマンド別構造体の例を次に示します。この章の残りのコードフラグメントは、この構造体を使って HBA インタフェースを示しています。

```
struct isp_cmd {
     struct isp_request     cmd_isp_request;
     struct isp_response    cmd_isp_response;
     struct scsi_pkt        *cmd_pkt;
     struct isp_cmd         *cmd_forw;
     uint32_t               cmd_dmacount;
     ddi_dma_handle_t       cmd_dmahandle;
     uint_t                 cmd_cookie;
     uint_t                 cmd_ncookies;
     uint_t                 cmd_cookiecnt;
     uint_t                 cmd_nwin;
     uint_t                 cmd_curwin;
     off_t                  cmd_dma_offset;
     uint_t                 cmd_dma_len;
     ddi_dma_cookie_t       cmd_dmacookies[ISP_NDATASEGS];
     u_int                  cmd_flags;
     u_short                cmd_slot;
     u_int                  cmd_cdblen;
     u_int                  cmd_scblen;
 };
```

モジュール初期化用のエントリポイント

このセクションでは、SCSI HBA ドライバによって実行される操作のエントリポイントについて説明します。

次の SCSI HBA ドライバ用のコードは、代表的な dev_ops(9S) 構造体を示しています。ドライバは、この構造体の devo_bus_ops フィールドを NULL に初期化する必要があります。SCSI HBA ドライバは、特別な目的でリーフドライバインタフェースを提供することがあります。その場合、devo_cb_ops フィールドは cb_ops(9S) 構造体を指すことがあります。この例では、リーフドライバインタフェースはエクスポートされないため、devo_cb_ops フィールドは NULL に初期化されます。

_init() エントリポイント (SCSI HBA ドライバ)

_init(9E) 関数は、ロード可能なモジュールを初期化します。_init() は、ロード可能なモジュール内のほかのすべてのルーチンの前に呼び出されます。

SCSI HBA では、_init() 関数は、mod_install(9F) を呼び出す前に、scsi_hba_init(9F) を呼び出して、HBA ドライバの存在をフレームワークに通知する必要があります。scsi_hba__init() ルーチンがゼロ以外の値を返す場合、_init() はこの値を返します。それ以外の場合、_init() は mod_install(9F) によって返された値を返す必要があります。

ドライバは、mod_install(9F) を呼び出す前に、必要なグローバル状態を初期化します。

mod_install() が失敗した場合、_init() 関数は割り当てられているグローバルリソースをすべて解放する必要があります。_init() は、復帰する前に scsi_hba_fini(9F) を呼び出す必要があります。

次の例では、グローバルな mutex を使用して、ドライバのすべてのインスタンスにグローバルなデータの割り当て方法を示しています。このコードでは、グローバルな mutex とソフト状態構造体の情報を宣言しています。グローバルな mutex とソフト状態は、_init() の実行中に初期化されます。

_fini() エントリポイント (SCSI HBA ドライバ)

_fini(9E) 関数は、システムが SCSI HBA ドライバをアンロードしようとするときに呼び出されます。_fini() 関数は、mod_remove(9F) を呼び出して、ドライバがアンロード可能かどうかを判定する必要があります。mod_remove() が 0 を返した場合、このモジュールはアンロード可能です。HBA ドライバは、_init(9E) で割り当てられたグローバルリソースをすべて解放する必要があります。また、scsi_hba_fini(9F) も呼び出す必要があります。

_fini() は、mod_remove() によって返された値を返す必要があります。

注 – HBA ドライバは、mod_remove(9F) が 0 を返さないかぎり、リソースを解放したり、scsi_hba_fini(9F) を呼び出したりすることはできません。

例 18–1 は、SCSI HBA 用のモジュールの初期化を示しています。

例 18–1　SCSI HBA 用のモジュールの初期化

```
static struct dev_ops isp_dev_ops = {
    DEVO_REV,       /* devo_rev */
    0,              /* refcnt  */
    isp_getinfo,    /* getinfo */
    nulldev,        /* probe */
    isp_attach,     /* attach */
```

例 18–1　SCSI HBA 用のモジュールの初期化　　(続き)

```
    isp_detach,     /* detach */
    nodev,          /* reset */
    NULL,           /* driver operations */
    NULL,           /* bus operations */
    isp_power,      /* power management */
};

/*
 * Local static data
 */
static kmutex_t      isp_global_mutex;
static void          *isp_state;

int
_init(void)
{
    int     err;

    if ((err = ddi_soft_state_init(&isp_state,
        sizeof (struct isp), 0)) != 0) {
        return (err);
    }
    if ((err = scsi_hba_init(&modlinkage)) == 0) {
        mutex_init(&isp_global_mutex, "isp global mutex",
        MUTEX_DRIVER, NULL);
        if ((err = mod_install(&modlinkage)) != 0) {
            mutex_destroy(&isp_global_mutex);
            scsi_hba_fini(&modlinkage);
            ddi_soft_state_fini(&isp_state);
        }
    }
    return (err);
}

int
_fini(void)
{
    int     err;

    if ((err = mod_remove(&modlinkage)) == 0) {
        mutex_destroy(&isp_global_mutex);
        scsi_hba_fini(&modlinkage);
        ddi_soft_state_fini(&isp_state);
    }
    return (err);
}
```

自動構成のエントリポイント

各デバイスドライバには dev_ops(9S) 構造体が関連付けられています。この構造体により、カーネルはドライバの自動構成エントリポイントを見つけることができます。これらの自動構成ルーチンについては、第 6 章「ドライバの自動構成」に詳しく説明されています。このセクションでは、そのようなエントリポイントのう

ち、SCSI HBA ドライバで実行される操作に関連付けられたものについてのみ説明します。このようなエントリポイントには、attach(9E) と detach(9E) があります。

attach() エントリポイント (SCSI HBA ドライバ)

SCSI HBA ドライバの attach(9E) エントリポイントは、デバイスに対してこのドライバのインスタンスを構成および接続するときにいくつかのタスクを実行します。実際のデバイスの一般的なドライバでは、次のオペレーティングシステムとハードウェアに関する問題に対処する必要があります。

- ソフト状態構造体
- DMA
- トランスポート構造体
- HBA ドライバの接続
- レジスタマッピング
- 割り込み仕様
- 割り込み処理
- 電源管理可能なコンポーネントの作成
- 接続ステータスのレポート

ソフト状態構造体

デバイスインスタンスごとのソフト状態構造体を割り当てる際にエラーが発生した場合、ドライバは慎重にクリーンアップを行う必要があります。

DMA

HBA ドライバは、ddi_dma_attr_t 構造体を正しく初期化することで、その DMA エンジンの属性を記述する必要があります。

```
static ddi_dma_attr_t isp_dma_attr = {
     DMA_ATTR_V0,        /* ddi_dma_attr version */
     0,                  /* low address */
     0xffffffff,         /* high address */
     0x00ffffff,         /* counter upper bound */
     1,                  /* alignment requirements */
     0x3f,               /* burst sizes */
     1,                  /* minimum DMA access */
     0xffffffff,         /* maximum DMA access */
     (1<<24)-1,          /* segment boundary restrictions */
     1,                  /* scatter-gather list length */
     512,                /* device granularity */
     0                   /* DMA flags */
};
```

また、DMA を提供する場合は、そのハードウェアが DMA 対応スロットに取り付けられていることも確認するべきです。

```
if (ddi_slaveonly(dip) == DDI_SUCCESS) {
    return (DDI_FAILURE);
}
```

トランスポート構造体

HBA ドライバは、このインスタンスにさらにトランスポート構造体を割り当て、初期化します。tran_hba_private フィールドは、このインスタンスのソフト状態構造体を指すように設定します。特別なプローブカスタマイズが必要ない場合は、tran_tgt_probe フィールドを NULL に設定して、デフォルトの動作を実行できます。

```
tran = scsi_hba_tran_alloc(dip, SCSI_HBA_CANSLEEP);

isp->isp_tran                   = tran;
isp->isp_dip                    = dip;

tran->tran_hba_private          = isp;
tran->tran_tgt_private          = NULL;
tran->tran_tgt_init             = isp_tran_tgt_init;
tran->tran_tgt_probe            = scsi_hba_probe;
tran->tran_tgt_free             = (void (*)())NULL;

tran->tran_start                = isp_scsi_start;
tran->tran_abort                = isp_scsi_abort;
tran->tran_reset                = isp_scsi_reset;
tran->tran_getcap               = isp_scsi_getcap;
tran->tran_setcap               = isp_scsi_setcap;
tran->tran_init_pkt             = isp_scsi_init_pkt;
tran->tran_destroy_pkt          = isp_scsi_destroy_pkt;
tran->tran_dmafree              = isp_scsi_dmafree;
tran->tran_sync_pkt             = isp_scsi_sync_pkt;
tran->tran_reset_notify         = isp_scsi_reset_notify;
tran->tran_bus_quiesce          = isp_tran_bus_quiesce
tran->tran_bus_unquiesce        = isp_tran_bus_unquiesce
tran->tran_bus_reset            = isp_tran_bus_reset
tran->tran_interconnect_type    = isp_tran_interconnect_type
```

HBA ドライバの接続

HBA ドライバは、デバイスのこのインスタンスを接続し、必要があれば、エラーのクリーンアップを実行します。

```
i = scsi_hba_attach_setup(dip, &isp_dma_attr, tran, 0);
if (i != DDI_SUCCESS) {
    /* do error recovery */
    return (DDI_FAILURE);
}
```

レジスタマッピング

HBA ドライバは、そのデバイスのレジスタをマップします。ドライバでは次の項目を指定する必要があります。

- レジスタセットインデックス
- デバイスのデータアクセス特性
- マップされるレジスタのサイズ

```
ddi_device_acc_attr_t    dev_attributes;

     dev_attributes.devacc_attr_version = DDI_DEVICE_ATTR_V0;
     dev_attributes.devacc_attr_dataorder = DDI_STRICTORDER_ACC;
     dev_attributes.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;

     if (ddi_regs_map_setup(dip, 0, (caddr_t *)&isp->isp_reg,
     0, sizeof (struct ispregs), &dev_attributes,
     &isp->isp_acc_handle) != DDI_SUCCESS) {
        /* do error recovery */
        return (DDI_FAILURE);
     }
```

割り込みハンドラの追加

ドライバはまず、*iblock cookie* を取得して、ドライバハンドラで使用される mutex をすべて初期化する必要があります。それらの mutex の初期化が完了している場合にのみ、割り込みハンドラを追加できます。

```
i = ddi_get_iblock_cookie(dip, 0, &isp->iblock_cookie};
if (i != DDI_SUCCESS) {
    /* do error recovery */
    return (DDI_FAILURE);
}

mutex_init(&isp->mutex, "isp_mutex", MUTEX_DRIVER,
(void *)isp->iblock_cookie);
i = ddi_add_intr(dip, 0, &isp->iblock_cookie,
0, isp_intr, (caddr_t)isp);
if (i != DDI_SUCCESS) {
    /* do error recovery */
    return (DDI_FAILURE);
}
```

高レベルのハンドラが必要な場合、そのようなハンドラを提供するようにドライバをコーディングします。それ以外の場合、ドライバはその接続に失敗できる必要があります。高レベルの割り込み処理については、155 ページの「高レベルの割り込みの処理」を参照してください。

電源管理可能なコンポーネントの作成

電源管理を使用すると、すべてのターゲットアダプタの電源レベルが 0 のときにホストバスアダプタの電源のみを切る必要がある場合、HBA ドライバは `power(9E)` エントリポイントを提供するだけで済みます。第 12 章「電源管理」を参照してください。HBA ドライバは、デバイスが実装するコンポーネントについて記述するための `pm-components(9P)` プロパティーも作成する必要があります。

これ以上は何も必要ありません。コンポーネントがデフォルトでアイドル状態になり、電源管理フレームワークのデフォルトの依存性処理によって、ターゲットアダプタの電源が入ると、ホストバスアダプタの電源も確実に入るようになるためです。自動電源管理が自動的に使用可能になる場合、この処理では、すべてのターゲットアダプタの電源が切れると、ホストバスアダプタの電源も切れます。

接続ステータスのレポート

最後に、HBA ドライバは、デバイスのこのインスタンスが接続され、成功を返すことをレポートします。

```
ddi_report_dev(dip);
    return (DDI_SUCCESS);
```

detach() エントリポイント (SCSI HBA ドライバ)

HBA ドライバは、scsi_hba_detach(9F) の呼び出しなど、標準的な切り離し操作を実行します。

SCSA HBA ドライバのエントリポイント

HBA ドライバは、SCSA インタフェースを介してターゲットドライバとともに動作できます。SCSA インタフェースでは、HBA ドライバは、scsi_hba_tran(9S) 構造体を介して呼び出し可能なエントリポイントをいくつか提供する必要があります。

これらのエントリポイントは、次の 5 つの機能グループに分けられます。

- ターゲットドライバインスタンスの初期化
- リソースの割り当てと解放
- コマンドのトランスポート
- 機能管理
- 中止およびリセット処理
- 動的再構成 (DR)

次の表に、SCSA HBA のエントリポイントを機能グループべつに示します。

表 18–3 SCSA エントリポイント

機能グループ	グループ内のエントリポイント	説明
ターゲットドライバインスタンスの初期化	tran_tgt_init(9E)	ターゲットごとの初期化を実行します (オプション)
	tran_tgt_probe(9E)	ターゲットが存在するかどうか SCSI バスを調べます (オプション)
	tran_tgt_free(9E)	ターゲットごとの解放を実行します (オプション)

表 18–3 SCSA エントリポイント (続き)

機能グループ	グループ内のエントリポイント	説明
リソース割り当て	tran_init_pkt(9E)	SCSI パケットと DMA リソースを割り当てます
	tran_destroy_pkt(9E)	SCSI パケットと DMA リソースを解放します
	tran_sync_pkt(9E)	DMA の前後にメモリーを同期させます
	tran_dmafree(9E)	DMA リソースを解放します
コマンドのトランスポート	tran_start(9E)	SCSI コマンドをトランスポートします
機能管理	tran_getcap(9E)	機能の値について問い合わせます
	tran_setcap(9E)	機能の値を設定します
中止とリセット	tran_abort(9E)	未処理の SCSI コマンドを中止します
	tran_reset(9E)	ターゲットデバイスまたは SCSI バスをリセットします
	tran_bus_reset(9E)	SCSI バスをリセットします
	tran_reset_notify(9E)	バスのリセットをターゲットに知らせるように要求します (オプション)
動的再構成 (DR)	tran_quiesce(9E)	バス上の動作を停止します
	tran_unquiesce(9E)	バス上の動作を再開します

ターゲットドライバインスタンスの初期化

以降のセクションでは、ターゲットエントリポイントについて説明します。

tran_tgt_init() エントリポイント

tran_tgt_init(9E) エントリポイントを使用すると、HBA はターゲットごとのリソースを割り当てて初期化できます。また、tran_tgt_init() を使用すると、HBA はデバイスのアドレスをその特定の HBA で有効かつサポート可能であるとみなすことができます。DDI_FAILURE を返すことにより、そのデバイスのターゲットドライバのインスタンスにはプローブも接続も行われません。

tran_tgt_init() は必須ではありません。tran_tgt_init() を指定しない場合、フレームワークでは該当するターゲットドライバの可能なインスタンスをすべてプローブおよび接続しようとします。

```
static int
isp_tran_tgt_init(
    dev_info_t            *hba_dip,
    dev_info_t            *tgt_dip,
    scsi_hba_tran_t       *tran,
    struct scsi_device    *sd)
{
    return ((sd->sd_address.a_target < N_ISP_TARGETS_WIDE &&
        sd->sd_address.a_lun < 8) ? DDI_SUCCESS : DDI_FAILURE);
}
```

tran_tgt_probe() エントリポイント

tran_tgt_probe(9E) エントリポイントを使用すると、HBA は必要に応じて scsi_probe(9F) の操作をカスタマイズできます。このエントリポイントは、ターゲットドライバが scsi_probe() を呼び出した場合にのみ呼び出されます。

HBA ドライバは、scsi_hba_probe(9F) を呼び出し、その戻り値を返すことで、scsi_probe() の通常の操作を保持できます。

このエントリポイントは必須ではありません。必要がない場合、HBA ドライバは、scsi_hba_tran(9S) 構造体の tran_tgt_probe ベクトルが scsi_hba_probe() を指すように設定します。

scsi_probe() は、scsi_inquiry(9S) 構造体を割り当て、scsi_device(9S) 構造体の sd_inq フィールドが scsi_inquiry のデータを指すように設定します。scsi_hba_probe() はこのタスクを自動的に処理します。次に、scsi_unprobe(9F) は scsi_inquiry データを解放します。

scsi_inquiry データの割り当てを除き、同じ SCSI デバイスが tran_tgt_probe() を何度も呼び出す可能性があるため、tran_tgt_probe() をステートレスにする必要があります。通常、scsi_inquiry データの割り当ては scsi_hba_probe () によって処理されます。

注 – scsi_inquiry(9S) 構造体の割り当ては scsi_hba_probe() によって自動的に処理されます。この情報は、カスタムの scsi_probe() 処理が必要な場合にだけ関係があります。

```
static int
isp_tran_tgt_probe(
    struct scsi_device    *sd,
    int                   (*callback)())
{
    /*
     * Perform any special probe customization needed.
     * Normal probe handling.
     */
    return (scsi_hba_probe(sd, callback));
}
```

tran_tgt_free() エントリポイント

tran_tgt_free(9E) エントリポイントを使用すると、HBA はターゲットのインスタンスに対して解放またはクリーンアップの手順を実行できます。このエントリポイントはオプションです。

```
static void
isp_tran_tgt_free(
    dev_info_t            *hba_dip,
    dev_info_t            *tgt_dip,
    scsi_hba_tran_t       *hba_tran,
    struct scsi_device    *sd)
{
    /*
     * Undo any special per-target initialization done
     * earlier in tran_tgt_init(9F) and tran_tgt_probe(9F)
     */
}
```

リソース割り当て

以降のセクションでは、リソースの割り当てについて説明します。

tran_init_pkt() エントリポイント

tran_init_pkt(9E) エントリポイントは、ターゲットドライバの要求に応じて scsi_pkt(9S) 構造体と DMA リソースの割り当てと初期化を行います。

tran_init_pkt(9E) エントリポイントは、ターゲットドライバが SCSA 関数 scsi_init_pkt(9F) を呼び出したときに呼び出されます。

tran_init_pkt(9E) エントリポイントの各呼び出しは、次の 3 つの可能なサービスの 1 つ以上を実行するよう要求するものです。

- scsi_pkt(9S) 構造体の割り当てと初期化
- データ転送のための DMA リソースの割り当て
- データ転送の次回分のための DMA リソースの再割り当て

scsi_pkt(9S) 構造体の割り当てと初期化

tran_init_pkt(9E) エントリポイントは、pkt が NULL の場合、scsi_hba_pkt_alloc(9F) を介して scsi_pkt(9S) 構造体を割り当てる必要があります。

scsi_hba_pkt_alloc(9F) では、次の項目に領域を割り当てます。

- scsi_pkt(9S)
- SCSI CDB (長さは cmdlen)
- SCSI ステータスの完了領域 (長さは statuslen)
- パケットごとのターゲットドライバの非公開データ領域 (長さは tgtlen)

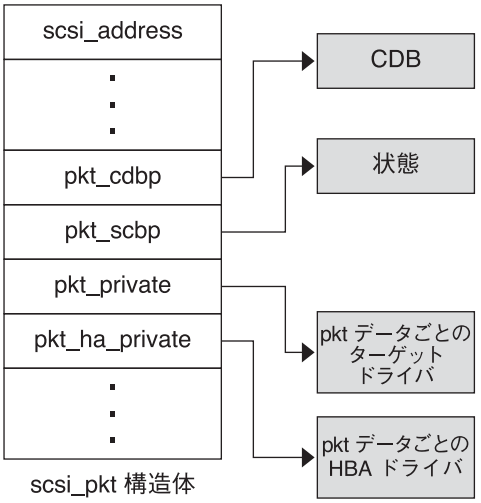
- パケットごとの HBA ドライバの非公開データ領域 (長さは `hbalen`)

`scsi_pkt(9S)` 構造体のメンバー (`pkt` など) は、次のメンバーを除いてゼロに初期化する必要があります。

- `pkt_scbp` – ステータスの完了
- `pkt_cdbp` – CDB
- `pkt_ha_private` – HBA ドライバの非公開データ
- `pkt_private` – ターゲットドライバの非公開データ

次の図に示すように、これらのメンバーはフィールドの値が格納されるメモリー空間を指すポインタです。詳細については、390 ページの「`scsi_pkt` 構造体 (HBA)」を参照してください。

図 18–5 `scsi_pkt(9S)` 構造体のポインタ



次の例は、`scsi_pkt` 構造体の割り当てと初期化を示しています。

例 18–2 HBA ドライバでの SCSI パケット構造体の初期化

```
static struct scsi_pkt                  *
isp_scsi_init_pkt(
    struct scsi_address    *ap,
    struct scsi_pkt        *pkt,
    struct buf             *bp,
    int                    cmdlen,
    int                    statuslen,
    int                    tgtlen,
    int                    flags,
    int                    (*callback)(),
    caddr_t                arg)
```

例 18–2 HBA ドライバでの SCSI パケット構造体の初期化 *(続き)*

```
{
    struct isp_cmd         *sp;
    struct isp             *isp;
    struct scsi_pkt        *new_pkt;

    ASSERT(callback == NULL_FUNC || callback == SLEEP_FUNC);

    isp = (struct isp *)ap->a_hba_tran->tran_hba_private;
   /*
    * First step of isp_scsi_init_pkt:  pkt allocation
    */
    if (pkt == NULL) {
        pkt = scsi_hba_pkt_alloc(isp->isp_dip, ap, cmdlen,
            statuslen, tgtlen, sizeof (struct isp_cmd),
            callback, arg);
        if (pkt == NULL) {
            return (NULL);
       }

       sp = (struct isp_cmd *)pkt->pkt_ha_private;
      /*
       * Initialize the new pkt
       */
       sp->cmd_pkt         = pkt;
       sp->cmd_flags       = 0;
       sp->cmd_scblen      = statuslen;
       sp->cmd_cdblen      = cmdlen;
       sp->cmd_dmahandle   = NULL;
       sp->cmd_ncookies    = 0;
       sp->cmd_cookie      = 0;
       sp->cmd_cookiecnt   = 0;
       sp->cmd_nwin        = 0;
       pkt->pkt_address    = *ap;
       pkt->pkt_comp       = (void (*)())NULL;
       pkt->pkt_flags      = 0;
       pkt->pkt_time       = 0;
       pkt->pkt_resid      = 0;
       pkt->pkt_statistics = 0;
       pkt->pkt_reason     = 0;

       new_pkt = pkt;
    } else {
       sp = (struct isp_cmd *)pkt->pkt_ha_private;
       new_pkt = NULL;
    }
   /*
    * Second step of isp_scsi_init_pkt:  dma allocation/move
    */
    if (bp && bp->b_bcount != 0) {
        if (sp->cmd_dmahandle == NULL) {
            if (isp_i_dma_alloc(isp, pkt, bp,
            flags, callback) == 0) {
            if (new_pkt) {
                scsi_hba_pkt_free(ap, new_pkt);
            }
            return ((struct scsi_pkt *)NULL);
```

例 18–2 HBA ドライバでの SCSI パケット構造体の初期化 (続き)

```
        }
        } else {
            ASSERT(new_pkt == NULL);
            if (isp_i_dma_move(isp, pkt, bp) == 0) {
                return ((struct scsi_pkt *)NULL);
            }
        }
    }
    return (pkt);
}
```

DMA リソースの割り当て

tran_init_pkt(9E) エントリポイントは、次の条件が真の場合、データ転送用の DMA リソースを割り当てる必要があります。

- bp が null 以外である。
- bp->b_bcount がゼロ以外である。
- DMA リソースがこの scsi_pkt(9S) にまだ割り当てられていない。

HBA ドライバは、DMA リソースが特定のコマンドに対してどのように割り当てられるかを追跡する必要があります。この割り当ては、パケットごとの HBA ドライバの非公開データに含まれるフラグビットまたは DMA ハンドルを使って行われます。

pkt に含まれる PKT_DMA_PARTIAL フラグを使用すると、ターゲットドライバはデータ転送を複数の SCSI コマンドに分けることで完全な要求に対応できます。この方法は、HBA ハードウェアの分散および集中機能または DMA リソースが 1 つの SCSI コマンドで要求を完了できない場合に役立ちます。

PKT_DMA_PARTIAL フラグを使用すると、HBA ドライバは DDI_DMA_PARTIAL フラグを設定できます。DDI_DMA_PARTIAL フラグは、この SCSI コマンドの DMA リソースが割り当てられるときに役立ちます。たとえば、ddi_dma_buf_bind_handle(9F)) コマンドを使用して DMA リソースを割り当てることができます。DMA リソースの割り当て時に使われる DMA 属性には、DMA を実行する HBA ハードウェアの機能に対する制約が正確に記述されています。システムが要求の一部にしか DMA リソースを割り当てられない場合、ddi_dma_buf_bind_handle(9F) は DDI_DMA_PARTIAL_MAP を返します。

tran_init_pkt(9E) エントリポイントは、この転送に割り当てられていない DMA リソースの量を pkt_resid フィールドに返す必要があります。

ターゲットドライバは、tran_init_pkt(9E) への 1 回の要求で、scsi_pkt(9S) 構造体とその pkt の DMA リソースを同時に割り当てることがあります。このときに HBA ドライバが DMA リソースを割り当てられない場合、HBA ドライバは割り当てられている scsi_pkt(9S) を解放してから復帰する必要があります。scsi_pkt(9S) を解放するには、scsi_hba_pkt_free(9F) を呼び出します。

ターゲットドライバは、最初に scsi_pkt(9S) を割り当て、あとでこの pkt に DMA リソースを割り当てることがあります。この場合、HBA ドライバは DMA リソースを割り当てられなくても、pkt を解放してはいけません。この場合は、ターゲットドライバが pkt の解放を担当します。

例 18–3 HBA ドライバでの DMA リソースの割り当て

```
static int
isp_i_dma_alloc(
    struct isp          *isp,
    struct scsi_pkt     *pkt,
    struct buf          *bp,
    int                 flags,
    int                 (*callback)())
{
    struct isp_cmd      *sp  = (struct isp_cmd *)pkt->pkt_ha_private;
    int                 dma_flags;
    ddi_dma_attr_t      tmp_dma_attr;
    int                 (*cb)(caddr_t);
    int                 i;

    ASSERT(callback == NULL_FUNC || callback == SLEEP_FUNC);

    if (bp->b_flags & B_READ) {
        sp->cmd_flags &= ~CFLAG_DMASEND;
        dma_flags = DDI_DMA_READ;
    } else {
        sp->cmd_flags |= CFLAG_DMASEND;
        dma_flags = DDI_DMA_WRITE;
    }
    if (flags & PKT_CONSISTENT) {
        sp->cmd_flags |= CFLAG_CMDIOPB;
        dma_flags |= DDI_DMA_CONSISTENT;
    }
    if (flags & PKT_DMA_PARTIAL) {
        dma_flags |= DDI_DMA_PARTIAL;
    }

    tmp_dma_attr = isp_dma_attr;
    tmp_dma_attr.dma_attr_burstsizes = isp->isp_burst_size;

    cb = (callback == NULL_FUNC) ? DDI_DMA_DONTWAIT :
DDI_DMA_SLEEP;

    if ((i = ddi_dma_alloc_handle(isp->isp_dip, &tmp_dma_attr,
      cb, 0, &sp->cmd_dmahandle)) != DDI_SUCCESS) {
        switch (i) {
        case DDI_DMA_BADATTR:
            bioerror(bp, EFAULT);
            return (0);
        case DDI_DMA_NORESOURCES:
            bioerror(bp, 0);
            return (0);
        }
    }

    i = ddi_dma_buf_bind_handle(sp->cmd_dmahandle, bp, dma_flags,
```

例 18–3 HBA ドライバでの DMA リソースの割り当て (続き)

```
    cb, 0, &sp->cmd_dmacookies[0], &sp->cmd_ncookies);

    switch (i) {
    case DDI_DMA_PARTIAL_MAP:
    if (ddi_dma_numwin(sp->cmd_dmahandle, &sp->cmd_nwin) ==
            DDI_FAILURE) {
        cmn_err(CE_PANIC, "ddi_dma_numwin() failed\n");
    }

    if (ddi_dma_getwin(sp->cmd_dmahandle, sp->cmd_curwin,
        &sp->cmd_dma_offset, &sp->cmd_dma_len,
        &sp->cmd_dmacookies[0], &sp->cmd_ncookies) ==
           DDI_FAILURE) {
        cmn_err(CE_PANIC, "ddi_dma_getwin() failed\n");
    }
    goto get_dma_cookies;

    case DDI_DMA_MAPPED:
    sp->cmd_nwin = 1;
    sp->cmd_dma_len = 0;
    sp->cmd_dma_offset = 0;

get_dma_cookies:
    i = 0;
    sp->cmd_dmacount = 0;
    for (;;) {
        sp->cmd_dmacount += sp->cmd_dmacookies[i++].dmac_size;

        if (i == ISP_NDATASEGS || i == sp->cmd_ncookies)
        break;
        ddi_dma_nextcookie(sp->cmd_dmahandle,
        &sp->cmd_dmacookies[i]);
    }
    sp->cmd_cookie = i;
    sp->cmd_cookiecnt = i;

    sp->cmd_flags |= CFLAG_DMAVALID;
    pkt->pkt_resid = bp->b_bcount - sp->cmd_dmacount;
    return (1);

    case DDI_DMA_NORESOURCES:
    bioerror(bp, 0);
    break;

    case DDI_DMA_NOMAPPING:
    bioerror(bp, EFAULT);
    break;

    case DDI_DMA_TOOBIG:
    bioerror(bp, EINVAL);
    break;

    case DDI_DMA_INUSE:
    cmn_err(CE_PANIC, "ddi_dma_buf_bind_handle:"
        " DDI_DMA_INUSE impossible\n");
```

例 18–3 HBA ドライバでの DMA リソースの割り当て (続き)

```
    default:
    cmn_err(CE_PANIC, "ddi_dma_buf_bind_handle:"
        " 0x%x impossible\n", i);
    }

    ddi_dma_free_handle(&sp->cmd_dmahandle);
    sp->cmd_dmahandle = NULL;
    sp->cmd_flags &= ~CFLAG_DMAVALID;
    return (0);
}
```

データ転送のための DMA リソースの再割り当て

前回割り当てたパケットにまだ転送されていないデータが残っている場合は、次の条件に適合したときに tran_init_pkt(9E) エントリポイントで DMA リソースを再割り当てする必要があります。

- 部分的な DMA リソースがすでに割り当てられている。
- 前回の tran_init_pkt(9E) の呼び出しで、ゼロ以外の pkt_resid が返された。
- bp が null 以外である。
- bp->b_bcount がゼロ以外である。

転送の次回分に DMA リソースを再割り当てするとき、tran_init_pkt(9E) は、この転送に割り当てられていない DMA リソースの量を pkt_resid フィールドに返す必要があります。

DMA リソースを移動しようとしているときにエラーが発生した場合、tran_init_pkt(9E) は scsi_pkt(9S) を解放してはいけません。この場合は、ターゲットドライバがパケットの解放を担当します。

コールバックパラメータが NULL_FUNC である場合、tran_init_pkt(9E) エントリポイントは関数をスリープしたり、スリープの可能性のある関数を呼び出したりしてはいけません。コールバックパラメータが SLEEP_FUNC であり、リソースがすぐに利用できない場合、tran_init_pkt(9E) エントリポイントはスリープになります。要求を満たすことが不可能でないかぎり、tran_init_pkt() はリソースが利用できるようになるまでスリープになります。

例 18–4 HBA ドライバでの DMA リソースの再割り当て

```
static int
isp_i_dma_move(
    struct isp          *isp,
    struct scsi_pkt     *pkt,
    struct buf          *bp)
{
    struct isp_cmd      *sp  = (struct isp_cmd *)pkt->pkt_ha_private;
    int                 i;

    ASSERT(sp->cmd_flags & CFLAG_COMPLETED);
```

例 18–4 HBA ドライバでの DMA リソースの再割り当て (続き)

```
    sp->cmd_flags &= ~CFLAG_COMPLETED;
   /*
    * If there are no more cookies remaining in this window,
    * must move to the next window first.
    */
    if (sp->cmd_cookie == sp->cmd_ncookies) {
   /*
    * For small pkts, leave things where they are
    */
    if (sp->cmd_curwin == sp->cmd_nwin && sp->cmd_nwin == 1)
        return (1);
   /*
    * At last window, cannot move
    */
    if (++sp->cmd_curwin >= sp->cmd_nwin)
        return (0);
    if (ddi_dma_getwin(sp->cmd_dmahandle, sp->cmd_curwin,
        &sp->cmd_dma_offset, &sp->cmd_dma_len,
        &sp->cmd_dmacookies[0], &sp->cmd_ncookies) ==
        DDI_FAILURE)
        return (0);
        sp->cmd_cookie = 0;
    } else {
   /*
    * Still more cookies in this window - get the next one
    */
    ddi_dma_nextcookie(sp->cmd_dmahandle,
        &sp->cmd_dmacookies[0]);
    }
   /*
    * Get remaining cookies in this window, up to our maximum
    */
    i = 0;
    for (;;) {
    sp->cmd_dmacount += sp->cmd_dmacookies[i++].dmac_size;
    sp->cmd_cookie++;
    if (i == ISP_NDATASEGS ||
        sp->cmd_cookie == sp->cmd_ncookies)
            break;
    ddi_dma_nextcookie(sp->cmd_dmahandle,
        &sp->cmd_dmacookies[i]);
    }
    sp->cmd_cookiecnt = i;
    pkt->pkt_resid = bp->b_bcount - sp->cmd_dmacount;
    return (1);
}
```

tran_destroy_pkt() エントリポイント

tran_destroy_pkt(9E) エントリポイントは、scsi_pkt(9S) 構造体を解放する HBA ドライバの関数です。tran_destroy_pkt() エントリポイントは、ターゲットドライバが scsi_destroy_pkt(9F) を呼び出したときに呼び出されます。

tran_destroy_pkt() エントリポイントは、パケットに割り当てられているすべての DMA リソースを解放する必要があります。転送が完了したあと、DMA リソースが

解放され、キャッシュされたデータが残っている場合は、暗黙的に DMA の同期が行われます。tran_destroy_pkt() エントリポイントは、scsi_hba_pkt_free(9F) を呼び出して、SCSI パケットを解放します。

例 18–5　HBA ドライバの tran_destroy_pkt(9E) エントリポイント

```
static void
isp_scsi_destroy_pkt(
    struct scsi_address    *ap,
    struct scsi_pkt    *pkt)
{
    struct isp_cmd *sp = (struct isp_cmd *)pkt->pkt_ha_private;
   /*
    * Free the DMA, if any
    */
    if (sp->cmd_flags & CFLAG_DMAVALID) {
        sp->cmd_flags &= ~CFLAG_DMAVALID;
        (void) ddi_dma_unbind_handle(sp->cmd_dmahandle);
        ddi_dma_free_handle(&sp->cmd_dmahandle);
        sp->cmd_dmahandle = NULL;
    }
   /*
    * Free the pkt
    */
    scsi_hba_pkt_free(ap, pkt);
}
```

tran_sync_pkt() エントリポイント

tran_sync_pkt(9E) エントリポイントは、DMA 転送の前またはあとで、scsi_pkt(9S) 構造体に割り当てられた DMA オブジェクトを同期させます。tran_sync_pkt() エントリポイントは、ターゲットドライバが scsi_sync_pkt(9F) を呼び出したときに呼び出されます。

データ転送の方向がデバイスからメモリーへの DMA 読み取りの場合、tran_sync_pkt() は CPU のデータビューを同期させる必要があります。データ転送の方向がメモリーからデバイスへの DMA 書き込みの場合、tran_sync_pkt() はデバイスのデータビューを同期させる必要があります。

例 18–6　HBA ドライバの tran_sync_pkt(9E) エントリポイント

```
static void
isp_scsi_sync_pkt(
    struct scsi_address    *ap,
    struct scsi_pkt        *pkt)
{
    struct isp_cmd *sp = (struct isp_cmd *)pkt->pkt_ha_private;

    if (sp->cmd_flags & CFLAG_DMAVALID) {
        (void)ddi_dma_sync(sp->cmd_dmahandle, sp->cmd_dma_offset,
        sp->cmd_dma_len,
        (sp->cmd_flags & CFLAG_DMASEND) ?
        DDI_DMA_SYNC_FORDEV : DDI_DMA_SYNC_FORCPU);
    }
```

例 18–6 HBA ドライバの tran_sync_pkt(9E) エントリポイント (続き)

```
}
```

tran_dmafree() エントリポイント

tran_dmafree(9E) エントリポイントは、scsi_pkt(9S) 構造体に割り当てられている DMA リソースを解放します。tran_dmafree() エントリポイントは、ターゲットドライバが scsi_dmafree(9F) を呼び出したときに呼び出されます。

tran_dmafree() は、scsi_pkt(9S) 構造体に割り当てられた DMA リソースのみを解放し、scsi_pkt(9S) そのものは解放しません。DMA リソースが解放されると、暗黙的に DMA の同期が行われます。

注 – scsi_pkt(9S) の解放は、tran_destroy_pkt(9E) への別の要求で行われます。tran_destroy_pkt() では DMA リソースの解放も行う必要があるため、HBA ドライバは scsi_pkt() 構造体に DMA リソースが割り当てられているかどうかの正確な記録を保持する必要があります。

例 18–7 HBA ドライバの tran_dmafree (9E) エントリポイント

```
static void
isp_scsi_dmafree(
    struct scsi_address    *ap,
    struct scsi_pkt        *pkt)
{
    struct isp_cmd    *sp = (struct isp_cmd *)pkt->pkt_ha_private;

    if (sp->cmd_flags & CFLAG_DMAVALID) {
        sp->cmd_flags &= ~CFLAG_DMAVALID;
        (void)ddi_dma_unbind_handle(sp->cmd_dmahandle);
        ddi_dma_free_handle(&sp->cmd_dmahandle);
        sp->cmd_dmahandle = NULL;
    }
}
```

コマンドのトランスポート

HBA ドライバは、コマンドのトランスポートの一環として、次の手順を実行します。

1. ターゲットドライバからコマンドを受け入れます。
2. そのコマンドをデバイスハードウェアに発行します。
3. 発生する割り込みをすべて処理します。
4. タイムアウトを管理します。

tran_start() エントリポイント

SCSI HBA ドライバの tran_start(9E) エントリポイントは、アドレス指定されたターゲットに SCSI コマンドをトランスポートするために呼び出されます。SCSI コマンドは、ターゲットドライバが tran_init_pkt(9E) エントリポイントを介して割り当てた scsi_pkt(9S) 構造体の中に完全に記述されます。コマンドがデータ転送を伴う場合は、scsi_pkt(9S) 構造体に対して DMA リソースも割り当てられている必要があります。

tran_start() エントリポイントは、ターゲットドライバが scsi_transport(9F) を呼び出したときに呼び出されます。

tran_start() は、基本的なエラーチェックと、コマンドが必要とする初期化を行います。tran_start() の動作は、scsi_pkt(9S) 構造体の pkt_flags フィールドに設定される FLAG_NOINTR フラグの影響を受けることがあります。FLAG_NOINTR が設定されていない場合、tran_start() はハードウェア上の実行用コマンドをキューに入れて、すぐに復帰する必要があります。コマンドが完了すると同時に、HBA ドライバは pkt 完了ルーチンを呼び出します。

FLAG_NOINTR が設定されている場合、HBA ドライバは pkt 完了ルーチンを呼び出してはいけません。

次の例は、tran_start(9E) エントリポイントの処理方法を示しています。ISP ハードウェアは、ターゲットデバイスごとにキューを提供しています。アクティブな未処理のコマンドを 1 つしか管理できないデバイスの場合、ドライバは通常、ターゲットごとのキューを管理する必要があります。次に、ラウンドロビン方式で現在のコマンドが完了すると同時にドライバは新しいコマンドを起動します。

例 18–8　HBA ドライバの tran_start (9E) エントリポイント

```
static int
isp_scsi_start(
    struct scsi_address    *ap,
    struct scsi_pkt        *pkt)
{
    struct isp_cmd         *sp;
    struct isp             *isp;
    struct isp_request     *req;
    u_long                 cur_lbolt;
    int                    xfercount;
    int                    rval = TRAN_ACCEPT;
    int                    i;

    sp = (struct isp_cmd *)pkt->pkt_ha_private;
    isp = (struct isp *)ap->a_hba_tran->tran_hba_private;

    sp->cmd_flags = (sp->cmd_flags & ~CFLAG_TRANFLAG) |
                CFLAG_IN_TRANSPORT;
    pkt->pkt_reason = CMD_CMPLT;
   /*
    * set up request in cmd_isp_request area so it is ready to
    * go once we have the request mutex
```

例 18–8 HBA ドライバの tran_start (9E) エントリポイント (続き)

```
    */
    req = &sp->cmd_isp_request;

    req->req_header.cq_entry_type = CQ_TYPE_REQUEST;
    req->req_header.cq_entry_count = 1;
    req->req_header.cq_flags        = 0;
    req->req_header.cq_seqno = 0;
    req->req_reserved = 0;
    req->req_token = (opaque_t)sp;
    req->req_target = TGT(sp);
    req->req_lun_trn = LUN(sp);
    req->req_time = pkt->pkt_time;
    ISP_SET_PKT_FLAGS(pkt->pkt_flags, req->req_flags);
   /*
    * Set up data segments for dma transfers.
    */
    if (sp->cmd_flags & CFLAG_DMAVALID) {

        if (sp->cmd_flags & CFLAG_CMDIOPB) {
            (void) ddi_dma_sync(sp->cmd_dmahandle,
            sp->cmd_dma_offset, sp->cmd_dma_len,
            DDI_DMA_SYNC_FORDEV);
        }

        ASSERT(sp->cmd_cookiecnt > 0 &&
            sp->cmd_cookiecnt <= ISP_NDATASEGS);

        xfercount = 0;
        req->req_seg_count = sp->cmd_cookiecnt;
        for (i = 0; i < sp->cmd_cookiecnt; i++) {
            req->req_dataseg[i].d_count =
            sp->cmd_dmacookies[i].dmac_size;
            req->req_dataseg[i].d_base =
            sp->cmd_dmacookies[i].dmac_address;
            xfercount +=
            sp->cmd_dmacookies[i].dmac_size;
        }

        for (; i < ISP_NDATASEGS; i++) {
            req->req_dataseg[i].d_count = 0;
            req->req_dataseg[i].d_base = 0;
        }

        pkt->pkt_resid = xfercount;

        if (sp->cmd_flags & CFLAG_DMASEND) {
            req->req_flags |= ISP_REQ_FLAG_DATA_WRITE;
        } else {
            req->req_flags |= ISP_REQ_FLAG_DATA_READ;
        }
    } else {
        req->req_seg_count = 0;
        req->req_dataseg[0].d_count = 0;
    }
   /*
    * Set up cdb in the request
```

例 18–8　HBA ドライバの tran_start (9E) エントリポイント　　(続き)

```
    */
    req->req_cdblen = sp->cmd_cdblen;
    bcopy((caddr_t)pkt->pkt_cdbp, (caddr_t)req->req_cdb,
    sp->cmd_cdblen);
   /*
    * Start the cmd.  If NO_INTR, must poll for cmd completion.
    */
    if ((pkt->pkt_flags & FLAG_NOINTR) == 0) {
        mutex_enter(ISP_REQ_MUTEX(isp));
        rval = isp_i_start_cmd(isp, sp);
        mutex_exit(ISP_REQ_MUTEX(isp));
    } else {
        rval = isp_i_polled_cmd_start(isp, sp);
    }
    return (rval);
}
```

割り込みハンドラとコマンドの完了

割り込みハンドラは、デバイスのステータスをチェックして、問題になっている割り込みがデバイスによって生成されていることを確認する必要があります。また、割り込みハンドラはエラーが発生していないかどうかをチェックし、デバイスによって生成された割り込みを処理する必要もあります。

データが転送された場合は、実際に転送されたデータ量を調べるためにハードウェアがチェックされます。scsi_pkt(9S) 構造体の pkt_resid フィールドにはこの転送の残りが設定されます。

tran_init_pkt(9E) を介して DMA リソースが割り当てられるときに PKT_CONSISTENT フラグでマーク付けされたコマンドは、特別な処理を行います。HBA ドライバは、ターゲットドライバのコマンド完了コールバックが実行される前に、必ずそのコマンドのデータ転送が正しく同期されるよう保証する必要があります。

コマンドが完了したら、2 つの要件に基づいて操作する必要があります。

- 新しいコマンドがキューに入っている場合は、できるだけ速くハードウェア上でそのコマンドを起動します。
- コマンド完了コールバックを呼び出します。コールバックは、コマンドが完了したときにターゲットドライバに通知するために、ターゲットドライバによって scsi_pkt(9S) 構造体で設定されています。

可能な場合は、PKT_COMP コマンド完了コールバックを呼び出す前に、新しいコマンドをハードウェア上で起動するようにしてください。コマンド完了処理には、かなりの時間を要することがあります。通常、ターゲットドライバは biodone(9F) や場合によっては scsi_transport(9F) などの関数を呼び出して新しいコマンドを開始します。

この割り込みがこのドライバから要求された場合、割り込みハンドラは DDI_INTR_CLAIMED を返す必要があります。それ以外の場合は、 DDI_INTR_UNCLAIMED を返します。

次の例は、SCSI HBA ドライバ isp の割り込みハンドラを示しています。割り込みハンドラが attach(9E) で追加されるときに、caddr_t パラメータが設定されます。このパラメータは通常、状態構造体を指すポインタであり、インスタンスごとに割り当てられます。

例 18–9 HBA ドライバの割り込みハンドラ

```
static u_int
isp_intr(caddr_t arg)
{
    struct isp_cmd         *sp;
    struct isp_cmd         *head, *tail;
    u_short                response_in;
    struct isp_response    *resp;
    struct isp             *isp = (struct isp *)arg;
    struct isp_slot        *isp_slot;
    int                    n;

    if (ISP_INT_PENDING(isp) == 0) {
        return (DDI_INTR_UNCLAIMED);
    }

    do {
again:
       /*
        * head list collects completed packets for callback later
        */
        head = tail = NULL;
       /*
        * Assume no mailbox events (e.g., mailbox cmds, asynch
        * events, and isp dma errors) as common case.
        */
        if (ISP_CHECK_SEMAPHORE_LOCK(isp) == 0) {
            mutex_enter(ISP_RESP_MUTEX(isp));
           /*
            * Loop through completion response queue and post
            * completed pkts.  Check response queue again
            * afterwards in case there are more.
            */
            isp->isp_response_in =
            response_in = ISP_GET_RESPONSE_IN(isp);
           /*
            * Calculate the number of requests in the queue
            */
            n = response_in - isp->isp_response_out;
            if (n < 0) {
                n = ISP_MAX_REQUESTS -
                isp->isp_response_out + response_in;
            }
            while (n-- > 0) {
                ISP_GET_NEXT_RESPONSE_OUT(isp, resp);
                sp = (struct isp_cmd *)resp->resp_token;
```

例 18–9 HBA ドライバの割り込みハンドラ (続き)

```
                /*
                 * Copy over response packet in sp
                 */
                 isp_i_get_response(isp, resp, sp);
            }
            if (head) {
                tail->cmd_forw = sp;
                tail = sp;
                tail->cmd_forw = NULL;
            } else {
                tail = head = sp;
                sp->cmd_forw = NULL;
            }
            ISP_SET_RESPONSE_OUT(isp);
            ISP_CLEAR_RISC_INT(isp);
            mutex_exit(ISP_RESP_MUTEX(isp));

            if (head) {
                isp_i_call_pkt_comp(isp, head);
            }
        } else {
            if (isp_i_handle_mbox_cmd(isp) != ISP_AEN_SUCCESS) {
                return (DDI_INTR_CLAIMED);
            }
           /*
            * if there was a reset then check the response
            * queue again
            */
            goto again;
        }

    } while (ISP_INT_PENDING(isp));

    return (DDI_INTR_CLAIMED);
}

static void
isp_i_call_pkt_comp(
    struct isp              *isp,
    struct isp_cmd          *head)
{
    struct isp              *isp;
    struct isp_cmd          *sp;
    struct scsi_pkt         *pkt;
    struct isp_response     *resp;
    u_char                  status;

    while (head) {
        sp = head;
        pkt = sp->cmd_pkt;
        head = sp->cmd_forw;

        ASSERT(sp->cmd_flags & CFLAG_FINISHED);

        resp = &sp->cmd_isp_response;
```

例 18–9 HBA ドライバの割り込みハンドラ (続き)

```
        pkt->pkt_scbp[0] = (u_char)resp->resp_scb;
        pkt->pkt_state = ISP_GET_PKT_STATE(resp->resp_state);
        pkt->pkt_statistics = (u_long)
            ISP_GET_PKT_STATS(resp->resp_status_flags);
        pkt->pkt_resid = (long)resp->resp_resid;
       /*
        * If data was xferred and this is a consistent pkt,
        * do a dma sync
        */
        if ((sp->cmd_flags & CFLAG_CMDIOPB) &&
            (pkt->pkt_state & STATE_XFERRED_DATA)) {
                (void) ddi_dma_sync(sp->cmd_dmahandle,
                sp->cmd_dma_offset, sp->cmd_dma_len,
                DDI_DMA_SYNC_FORCPU);
        }

        sp->cmd_flags = (sp->cmd_flags & ~CFLAG_IN_TRANSPORT) |
            CFLAG_COMPLETED;
       /*
        * Call packet completion routine if FLAG_NOINTR is not set.
        */
        if (((pkt->pkt_flags & FLAG_NOINTR) == 0) &&
            pkt->pkt_comp) {
                (*pkt->pkt_comp)(pkt);
        }
    }
}
```

タイムアウトハンドラ

HBA ドライバは、タイムアウトの適用も担当します。scsi_pkt(9S) 構造体でタイムアウトがゼロに指定されていないかぎり、コマンドは指定時間内に完了する必要があります。

コマンドがタイムアウトになると、HBA ドライバは、CMD_TIMEOUT に設定されたpkt_reasonと、STAT_TIMEOUT との論理和を取得した pkt_statistics で scsi_pkt(9S) をマーク付けします。また、HBA ドライバはターゲットとバスの回復も試みないといけません。この回復処理を正常に実行できる場合、ドライバは、STAT_BUS_RESET または STAT_DEV_RESET との論理和を取得した pkt_statistics を使用して、scsi_pkt(9S) をマーク付けします。

回復の試行が完了したあとで、HBA ドライバはコマンド完了コールバックを呼び出します。

注 – 回復処理が正常に行われなかったり、試されなかったりした場合、ターゲットドライバは scsi_reset(9F) を呼び出してタイムアウトから回復しようとすることがあります。

ISP ハードウェアでは、コマンドのタイムアウトを直接管理し、タイムアウトしたコマンドを必要なステータスとともに返します。isp サンプルドライバのタイムアウトハンドラでは、60 秒ごとに 1 回だけアクティブなコマンドのタイムアウト状態をチェックします。

isp サンプルドライバは、timeout(9F) 機能を使用して、カーネルが 60 秒ごとにタイムアウトハンドラを呼び出すように設定します。caddr_t 引数は、attach(9E) の実行時にタイムアウトが初期化されるときに設定されるパラメータです。この場合、caddr_t 引数は、ドライバのインスタンスごとに割り当てられた状態構造体を指すポインタです。

タイムアウトが発生したときに、タイムアウトしたコマンドが ISP ハードウェアから返されなかった場合は、問題が発生しています。ハードウェアが正しく機能していないため、リセットする必要があります。

機能管理

以降のセクションでは、機能管理について説明します。

tran_getcap() エントリポイント

SCSI HBA ドライバの tran_getcap(9E) エントリポイントは、scsi_ifgetcap(9F) によって呼び出されます。ターゲットドライバは、SCSA 定義機能セットのいずれかの現在の値を調べるために scsi_ifgetcap() を呼び出します。

ターゲットドライバは、whom パラメータをゼロ以外の値に設定することで、特定のターゲットの機能の現在の設定を要求できます。whom 値をゼロに設定することは、SCSI バスまたはアダプタハードウェアの一般的な機能の現在の設定を要求することを意味します。

tran_getcap() エントリポイントは、-1 (機能が未定義の場合) または要求された機能の現在の値を返します。

HBA ドライバは、scsi_hba_lookup_capstr(9F) 関数を使用して、機能文字列を定義済みの標準的な機能セットと比較できます。

例 18–10　HBA ドライバの tran_getcap (9E) エントリポイント

```
static int
isp_scsi_getcap(
    struct scsi_address    *ap,
    char                   *cap,
    int                    whom)
{
    struct isp             *isp;
    int                    rval = 0;
    u_char                 tgt = ap->a_target;
```

例 18–10 HBA ドライバの tran_getcap (9E) エントリポイント (続き)

```
/*
 * We don't allow getting capabilities for other targets
 */
 if (cap == NULL || whom  == 0) {
     return (-1);
 }
 isp = (struct isp *)ap->a_hba_tran->tran_hba_private;
 ISP_MUTEX_ENTER(isp);

 switch (scsi_hba_lookup_capstr(cap)) {
 case SCSI_CAP_DMA_MAX:
     rval = 1 << 24; /* Limit to 16MB max transfer */
     break;
 case SCSI_CAP_MSG_OUT:
     rval = 1;
     break;
 case SCSI_CAP_DISCONNECT:
     if ((isp->isp_target_scsi_options[tgt] &
         SCSI_OPTIONS_DR) == 0) {
         break;
     } else if (
         (isp->isp_cap[tgt] & ISP_CAP_DISCONNECT) == 0) {
         break;
     }
     rval = 1;
     break;
 case SCSI_CAP_SYNCHRONOUS:
     if ((isp->isp_target_scsi_options[tgt] &
         SCSI_OPTIONS_SYNC) == 0) {
         break;
     } else if (
         (isp->isp_cap[tgt] & ISP_CAP_SYNC) == 0) {
         break;
     }
     rval = 1;
     break;
 case SCSI_CAP_WIDE_XFER:
     if ((isp->isp_target_scsi_options[tgt] &
         SCSI_OPTIONS_WIDE) == 0) {
         break;
     } else if (
         (isp->isp_cap[tgt] & ISP_CAP_WIDE) == 0) {
         break;
     }
     rval = 1;
     break;
 case SCSI_CAP_TAGGED_QING:
     if ((isp->isp_target_scsi_options[tgt] &
         SCSI_OPTIONS_DR) == 0 ||
         (isp->isp_target_scsi_options[tgt] &
         SCSI_OPTIONS_TAG) == 0) {
         break;
     } else if (
         (isp->isp_cap[tgt] & ISP_CAP_TAG) == 0) {
         break;
     }
```

例 18–10 HBA ドライバの tran_getcap (9E) エントリポイント (続き)

```
        rval = 1;
        break;
    case SCSI_CAP_UNTAGGED_QING:
        rval = 1;
        break;
    case SCSI_CAP_PARITY:
        if (isp->isp_target_scsi_options[tgt] &
            SCSI_OPTIONS_PARITY) {
            rval = 1;
        }
        break;
    case SCSI_CAP_INITIATOR_ID:
        rval = isp->isp_initiator_id;
        break;
    case SCSI_CAP_ARQ:
        if (isp->isp_cap[tgt] & ISP_CAP_AUTOSENSE) {
            rval = 1;
        }
        break;
    case SCSI_CAP_LINKED_CMDS:
        break;
    case SCSI_CAP_RESET_NOTIFICATION:
        rval = 1;
        break;
    case SCSI_CAP_GEOMETRY:
        rval = (64 << 16) | 32;
        break;
    default:
        rval = -1;
        break;
    }

    ISP_MUTEX_EXIT(isp);
    return (rval);
}
```

tran_setcap() エントリポイント

SCSI HBA ドライバの tran_setcap(9E) エントリポイントは、scsi_ifsetcap(9F) によって呼び出されます。ターゲットドライバは、SCSA 定義機能セットのいずれかの現在の値を変更するために scsi_ifsetcap() を呼び出します。

ターゲットドライバは、whom パラメータをゼロ以外の値に設定することで、特定のターゲットに新しい値が設定されるように要求できます。whom 値をゼロに設定することは、一般に SCSI バスまたはアダプタハードウェアの新しい値を設定するよう要求することを意味します。

tran_setcap() は必要に応じて次の値を返します。

- -1 (機能が未定義の場合)
- 0 (HBA ドライバが要求された値に機能を設定できない場合)
- 1 (HBA ドライバが要求された値に機能を設定できる場合)

HBA ドライバは、scsi_hba_lookup_capstr(9F) 関数を使用して、機能文字列を定義済みの標準的な機能セットと比較できます。

例 18–11　HBA ドライバの tran_setcap (9E) エントリポイント

```
static int
isp_scsi_setcap(
    struct scsi_address    *ap,
    char                   *cap,
    int                    value,
    int                    whom)
{
    struct isp             *isp;
    int                    rval = 0;
    u_char                 tgt = ap->a_target;
    int                    update_isp = 0;
   /*
    * We don't allow setting capabilities for other targets
    */
    if (cap == NULL || whom == 0) {
        return (-1);
    }

    isp = (struct isp *)ap->a_hba_tran->tran_hba_private;
    ISP_MUTEX_ENTER(isp);

    switch (scsi_hba_lookup_capstr(cap)) {
    case SCSI_CAP_DMA_MAX:
    case SCSI_CAP_MSG_OUT:
    case SCSI_CAP_PARITY:
    case SCSI_CAP_UNTAGGED_QING:
    case SCSI_CAP_LINKED_CMDS:
    case SCSI_CAP_RESET_NOTIFICATION:
   /*
    * None of these are settable through
    * the capability interface.
    */
        break;
    case SCSI_CAP_DISCONNECT:
        if ((isp->isp_target_scsi_options[tgt] &
            SCSI_OPTIONS_DR) == 0) {
                break;
        } else {
            if (value) {
                isp->isp_cap[tgt] |= ISP_CAP_DISCONNECT;
            } else {
                isp->isp_cap[tgt] &= ~ISP_CAP_DISCONNECT;
            }
        }
        rval = 1;
        break;
    case SCSI_CAP_SYNCHRONOUS:
        if ((isp->isp_target_scsi_options[tgt] &
            SCSI_OPTIONS_SYNC) == 0) {
                break;
        } else {
            if (value) {
                isp->isp_cap[tgt] |= ISP_CAP_SYNC;
```

例 18–11 HBA ドライバの tran_setcap (9E) エントリポイント (続き)

```
            } else {
                isp->isp_cap[tgt] &= ~ISP_CAP_SYNC;
            }
        }
        rval = 1;
        break;
    case SCSI_CAP_TAGGED_QING:
        if ((isp->isp_target_scsi_options[tgt] &
            SCSI_OPTIONS_DR) == 0 ||
            (isp->isp_target_scsi_options[tgt] &
            SCSI_OPTIONS_TAG) == 0) {
                break;
        } else {
            if (value) {
                isp->isp_cap[tgt] |= ISP_CAP_TAG;
            } else {
            isp->isp_cap[tgt] &= ~ISP_CAP_TAG;
            }
        }
        rval = 1;
        break;
    case SCSI_CAP_WIDE_XFER:
        if ((isp->isp_target_scsi_options[tgt] &
            SCSI_OPTIONS_WIDE) == 0) {
                break;
        } else {
            if (value) {
                isp->isp_cap[tgt] |= ISP_CAP_WIDE;
            } else {
                isp->isp_cap[tgt] &= ~ISP_CAP_WIDE;
            }
        }
        rval = 1;
        break;
    case SCSI_CAP_INITIATOR_ID:
        if (value < N_ISP_TARGETS_WIDE) {
            struct isp_mbox_cmd mbox_cmd;
            isp->isp_initiator_id = (u_short) value;
           /*
            * set Initiator SCSI ID
            */
            isp_i_mbox_cmd_init(isp, &mbox_cmd, 2, 2,
            ISP_MBOX_CMD_SET_SCSI_ID,
            isp->isp_initiator_id,
            0, 0, 0, 0);
            if (isp_i_mbox_cmd_start(isp, &mbox_cmd) == 0) {
                rval = 1;
            }
        }
        break;
    case SCSI_CAP_ARQ:
        if (value) {
            isp->isp_cap[tgt] |= ISP_CAP_AUTOSENSE;
        } else {
            isp->isp_cap[tgt] &= ~ISP_CAP_AUTOSENSE;
        }
```

例 18–11　HBA ドライバの tran_setcap (9E) エントリポイント　(続き)

```
        rval = 1;
        break;
    default:
        rval = -1;
        break;
    }
    ISP_MUTEX_EXIT(isp);
    return (rval);
}
```

中止およびリセット管理

以降のセクションでは、SCSI HBA の中止とリセットのエントリポイントについて説明します。

tran_abort() エントリポイント

SCSI HBA の tran_abort(9E) エントリポイントは、特定のターゲットに対して現在トランスポート中のコマンドを中止するために呼び出されます。このエントリポイントは、ターゲットドライバが scsi_abort(9F) を呼び出したときに呼び出されます。

tran_abort() エントリポイントは、pkt パラメータで示されたコマンドの中止を試みます。pkt パラメータが NULL の場合、tran_abort() は特定のターゲットまたは論理ユニットのトランスポート層にある未処理のコマンドをすべて中止しようとします。

正常に中止された各コマンドは、pkt_reason CMD_ABORTED と、STAT_ABORTED との論理和を取得した pkt_statistics でマーク付けされます。

tran_reset() エントリポイント

SCSI HBA ドライバの tran_reset(9E) エントリポイントは、SCSI バスまたは特定の SCSI ターゲットデバイスをリセットするために呼び出されます。このエントリポイントは、ターゲットドライバが scsi_reset(9F) を呼び出したときに呼び出されます。

tran_reset() エントリポイントは、レベルが RESET_ALL の場合に SCSI バスをリセットする必要があります。レベルが RESET_TARGET の場合は、特定のターゲットまたは論理ユニットのみをリセットする必要があります。

リセットの影響を受けるアクティブなコマンドは、pkt_reason CMD_RESET でマーク付けされます。リセットの種類によって、pkt_statistics の論理和を取得するために STAT_BUS_RESET または STAT_DEV_RESET のどちらが使われるかが決まります。

トランスポート層にあり、まだターゲット上でアクティブになっていないコマンドは、pkt_reason CMD_RESET と、STAT_ABORTED との論理和が取得された pkt_statistics でマーク付けされる必要があります。

tran_bus_reset() エントリポイント

tran_bus_reset(9E) は、ターゲットをリセットしないで SCSI バスをリセットする必要があります。

```
#include <sys/scsi/scsi.h>

int tran_bus_reset(dev_info_t *hba-dip, int level);
```

各表記の意味は次のとおりです。

**hba-dip* SCSI HBA に関連付けられたポインタです

level ターゲットはリセットされず、SCSI バスのみがリセットされるように、RESET_BUS に設定します

scsi_hba_tran(9S) 構造体の tran_bus_reset() ベクトルは、HBA ドライバの attach(9E) の実行中に初期化されます。このベクトルは、ユーザーがバスのリセットを開始したときに呼び出される予定の HBA エントリポイントを指します。

実装方法はハードウェアによって異なります。HBA ドライバがターゲットに影響を与えずに SCSI バスをリセットできない場合、ドライバは RESET_BUS に失敗するか、このベクトルの初期化を行いません。

tran_reset_notify() エントリポイント

tran_reset_notify(9E) エントリポイントは、SCSI バスのリセットが行われたときに使用します。この関数は、コールバックによってターゲットドライバに通知するように SCSI HBA ドライバに要求します。

例 18–12 HBA ドライバの tran_reset_notify (9E) エントリポイント

```
isp_scsi_reset_notify(
    struct scsi_address    *ap,
    int                    flag,
    void                   (*callback)(caddr_t),
    caddr_t                arg)
{
    struct isp                          *isp;
    struct isp_reset_notify_entry       *p, *beforep;
    int                                 rval = DDI_FAILURE;

    isp = (struct isp *)ap->a_hba_tran->tran_hba_private;
    mutex_enter(ISP_REQ_MUTEX(isp));
   /*
    * Try to find an existing entry for this target
    */
    p = isp->isp_reset_notify_listf;
    beforep = NULL;

    while (p) {
        if (p->ap == ap)
            break;
```

例 18–12 HBA ドライバの tran_reset_notify (9E) エントリポイント (続き)

```
        beforep = p;
        p = p->next;
    }

    if ((flag & SCSI_RESET_CANCEL) && (p != NULL)) {
        if (beforep == NULL) {
            isp->isp_reset_notify_listf = p->next;
        } else {
            beforep->next = p->next;
        }
        kmem_free((caddr_t)p, sizeof (struct
            isp_reset_notify_entry));
        rval = DDI_SUCCESS;
    } else if ((flag & SCSI_RESET_NOTIFY) && (p == NULL)) {
        p = kmem_zalloc(sizeof (struct isp_reset_notify_entry),
            KM_SLEEP);
        p->ap = ap;
        p->callback = callback;
        p->arg = arg;
        p->next = isp->isp_reset_notify_listf;
        isp->isp_reset_notify_listf = p;
        rval = DDI_SUCCESS;
    }

    mutex_exit(ISP_REQ_MUTEX(isp));
    return (rval);
}
```

動的再構成 (DR)

最小限のホットプラグ操作をサポートするために、ドライバはバスの休止、バスの休止解除、およびバスのリセットのサポートを実装することが必要な場合があります。scsi_hba_tran(9S) 構造体は、これらの操作をサポートします。ハードウェアが休止、休止解除、またはリセットを必要としない場合、ドライバの変更は必要ありません。

scsi_hba_tran 構造体には、次のフィールドがあります。

```
int (*tran_quiesce)(dev_info_t *hba-dip);
int (*tran_unquiesce)(dev_info_t *hba-dip);
int (*tran_bus_reset)(dev_info_t *hba-dip, int level);
```

これらのインタフェースは、SCSI バスを休止および休止解除します。

```
#include <sys/scsi/scsi.h>

int prefixtran_quiesce(dev_info_t *hba-dip);
int prefixtran_unquiesce(dev_info_t *hba-dip);
```

tran_quiesce(9E) および tran_unquiesce(9E) は、ホットプラグ用に設計されていない SCSI デバイスに使用します。HBA ドライバで動的再構成 (DR) をサポートするためには、これらの関数を実装する必要があります。

scsi_hba_tran(9S) 構造体の tran_quiesce() および tran_unquiesce() ベクトルは、attach(9E) の実行中に HBA エントリポイントを指すように初期化されます。これらの関数は、ユーザーが休止および休止解除操作を開始したときに呼び出されます。

tran_quiesce() エントリポイントは、SCSI バスに接続されているデバイスの再構成前または再構成中に SCSI バス上のすべての動作を停止します。tran_unquiesce() エントリポイントは、SCSA フレームワークによって呼び出され、再構成操作が完了したあとで、SCSI バス上の動作を再開します。

HBA ドライバは、すべての未処理コマンドが完了するのを待ってから成功を返すことで、tran_quiesce() を処理する必要があります。ドライバがバスを休止したあとは、対応する tran_unquiesce() エントリポイントを SCSA フレームワークが呼び出すまで、新しい入出力要求をすべてキューに入れる必要があります。

HBA ドライバは、キューに入っているターゲットドライバの入出力要求を開始することで、tran_unquiesce() の呼び出しを処理します。

SCSI HBA ドライバに固有の問題

このセクションでは、SCSI HBA ドライバに固有の問題について説明します。

HBA ドライバのインストール

SCSI HBA ドライバはリーフドライバと同様の方法でインストールされます。第 22 章「ドライバのコンパイル、ロード、パッケージ化、およびテスト」を参照してください。両者の相違点は、add_drv(1M) コマンドではドライバクラスを SCSI として指定する必要があることです。次に例を示します。

```
# add_drv -m" * 0666 root root" -i'"pci1077,1020"' -c scsi isp
```

HBA の構成プロパティー

HBA デバイスインスタンスへの接続時に、scsi_hba_attach_setup(9F) はその HBA インスタンス用の SCSI 設定プロパティーをいくつか作成します。特定のプロパティーが作成されるのは、HBA インスタンスにすでに接続されている同じ名前の既

存のプロパティーがない場合のみです。この制限により、HBA 構成ファイル内のデフォルトのプロパティー値が上書きされなくなります。

HBA ドライバは、ddi_prop_get_int(9F) を使用して各プロパティーを取得します。次に、HBA ドライバはプロパティーのデフォルト値の変更または受け入れを行って、HBA ドライバに固有の操作を構成します。

scsi-reset-delay プロパティー

scsi-reset-delay プロパティーは、SCSI バスまたは SCSI デバイスによるリセット遅延の回復時間をミリ秒単位で指定する整数です。

scsi-options プロパティー

scsi-options プロパティーは、個別に定義されたビットを通じていくつかのオプションを指定する整数です。

- SCSI_OPTIONS_DR (0x008) – 設定しない場合、HBA はターゲットデバイスに切り離しの特権を与えません。
- SCSI_OPTIONS_LINK (0x010) – 設定しない場合、HBA はリンクされたコマンドを有効にしません。
- SCSI_OPTIONS_SYNC (0x020) – 設定しない場合、HBA ドライバは同期データ転送のネゴシエーションを行いません。ドライバは、ターゲットによって開始された同期データ転送のネゴシエーションの試みを拒否します。
- SCSI_OPTIONS_PARITY (0x040) – 設定しない場合、HBA はパリティーなしで SCSI バスを実行します。
- SCSI_OPTIONS_TAG (0x080) – 設定しない場合、HBA はタグ付きコマンドキューイングモードで動作しません。
- SCSI_OPTIONS_FAST (0x100) – 設定しない場合、HBA はバスを FAST SCSI モードで操作してはいけません。
- SCSI_OPTIONS_WIDE (0x200) – 設定しない場合、HBA はバスを WIDE SCSI モードで操作してはいけません。

ターゲットごとの scsi-options

HBA ドライバは、次の形式でターゲットごとの scsi-options 機能をサポートできます。

```
target<n>-scsi-options=<hex value>
```

この例では、*<n>* はターゲット ID です。ターゲットごとの scsi-options プロパティーが定義されている場合、HBA ドライバは HBA ドライバインスタンスごとの scsi-options プロパティーではなく、この値を使用します。この方法では、特定の

ターゲットデバイス 1 つだけに対して同期データ転送を無効にする必要がある場合など、より厳密な制御を行うことができます。ターゲットごとの scsi-options プロパティーは、driver.conf(4) ファイルで定義できます。

次の例は、ターゲットデバイス 3 の同期データ転送を無効にするためのターゲットごとの scsi-options プロパティー定義を示しています。

```
target3-scsi-options=0x2d8
```

x86 ターゲットドライバの構成プロパティー

cmdk ディスク用のドライバなど、一部の x86 SCSI ターゲットドライバは、次の構成プロパティーを使用します。

- disk
- queue
- flow_control

cmdk サンプルドライバを使用して x86 プラットフォーム用の HBA ドライバを作成する場合は、適切なプロパティーをすべて driver.conf(4) ファイルで定義する必要があります。

注 – これらのプロパティー定義は、HBA ドライバの driver.conf(4) ファイルにしか表示されません。HBA ドライバ自身は、これらのプロパティーを調べたり、解釈しようとしたりすることは決してありません。これらのプロパティーは助言にすぎず、cmdk ドライバの付属物として機能します。これらのプロパティーは決して信頼できるものではありません。これらのプロパティー定義は、将来のリリースで使用できない可能性があります。

---

disk プロパティーは、cmdk によってサポートされるディスクの種類を定義するために使用できます。SCSI HBA の場合、disk プロパティーに指定できるのは次の値のみです。

- disk="scdk" – ディスクの種類は SCSI ディスクです

queue プロパティーは、strategy(9E) の実行中にディスクドライバが受信要求のキューをソートする方法を定義します。指定できる値は、次の 2 つがあります。

- queue="qsort" – disksort(9F) が提供する、一方向のエレベータキューイングモデルです
- queue="qfifo" – FIFO、つまり先入れ先出しのキューイングモデルです

flow_control プロパティーは、コマンドが HBA ドライバにトランスポートされる方法を定義します。指定できる値は、次の 3 つがあります。

- flow_control="dsngl" – 1 つの HBA ドライバにつき 1 つのコマンド
- flow_control="dmult" – 1 つの HBA ドライバにつき複数のコマンド。HBA キューが満杯になると、ドライバは TRAN_BUSY を返します
- flow_control="duplx" – HBA はキューごとに複数のコマンドを収容できる個別の読み取りキューと書き込みキューをサポートできます。書き込みキューには FIFO の順序付けが使用されます。読み取りキューに使用されるキューイングモデルは、*queue* プロパティーによって記述されます。HBA キューが満杯になると、ドライバは TRAN_BUSY を返します

次の例は、cmdk サンプルドライバ用に設計された x86 HBA PCI デバイスで使用する driver.conf(4) ファイルを示しています。

```
#
# config file for ISP 1020 SCSI HBA driver
#
       flow_control="dsngl" queue="qsort" disk="scdk"
       scsi-initiator-id=7;
```

キューイングのサポート

タグ付きのキューイングの定義については、SCSI-2 の仕様を参照してください。タグ付きのキューイングをサポートするためには、最初に *scsi_options* の SCSI_OPTIONS_TAG フラグをチェックして、タグ付きのキューイングがグローバルに有効かどうかを確認します。次に、ターゲットが SCSI-2 デバイスであるかどうか、およびターゲットでタグ付きのキューイングが有効になっているかどうかを確認します。これらの条件がすべて真である場合は、scsi_ifsetcap(9F) を使用してタグ付きのキューイングを有効にしてみてください。

タグ付きのキューイングに失敗した場合は、タグなしのキューイングを設定してみることができます。このモードでは、ホストアダプタドライバに必要または最適と思われるだけの数のコマンドを送信します。次にホストアダプタは、タグ付きのキューイングとは対照的に、それらのコマンドをターゲットのキューに 1 度に 1 つずつ入れます。タグ付きのキューイングでは、ホストアダプタは、ターゲットからキューが満杯であることが示されるまで、できるだけ多くのコマンドを送信します。

◆ ◆ ◆ 第 19 章

19 ネットワークデバイスのドライバ

Oracle Solaris OS 用のネットワークドライバを記述するには、Oracle Solaris Generic LAN Driver (GLD) フレームワークを使用します。

- 新しい Ethernet ドライバに関しては、GLDv3 フレームワークを使用します。433 ページの「GLDv3 ネットワークデバイスドライバフレームワーク」を参照してください。GLDv3 フレームワークは、関数呼び出しベースのインタフェースです。
- 古い Ethernet、トークンリング、または FDDI ドライバを管理するには、GLDv2 フレームワークを使用します。448 ページの「GLDv2 ネットワークデバイスドライバフレームワーク」を参照してください。GLDv2 は、ドライバ間で共有する共通コードを提供するカーネルモジュールです。
- NIC ドライバを記述する場合は、第 21 章「SR-IOV ドライバ」も参照してください。

GLDv3 ネットワークデバイスドライバフレームワーク

GLDv3 フレームワークは、MAC プラグインと、MAC ドライバサービスのルーチンおよび構造体に対する、関数呼び出しベースのインタフェースです。GLDv3 フレームワークは、必要な STREAMS エントリポイントを GLDv3 準拠ドライバに代わって実装し、DLPI との互換性を実現します。

このセクションの内容は次のとおりです。

GLDv3 の MAC 登録

GLDv3 は、MAC_PLUGIN_IDENT_ETHER タイプのプラグインに登録するドライバ用のドライバ API を定義します。

GLDv3 の MAC 登録プロセス

GLDv3 デバイスドライバは、次の手順を実行して MAC 層に登録する必要があります。

- sys/mac.h、sys/mac_ether.h、および sys/mac_provider.h の 3 つの MAC ヘッダーファイルをインクルードします。その他の MAC 関連ヘッダーファイルはドライバにインクルードしないでください。
- mac_callbacks 構造体を生成します。
- mac_init_ops() 関数をその _init() エントリポイントで呼び出します。
- mac_alloc() 関数をその attach() エントリポイントで呼び出し、mac_register 構造体を割り当てます。
- mac_register 構造体を生成し、mac_register() 関数をその attach() エントリポイントで呼び出します。
- mac_unregister() 関数をその detach() エントリポイントで呼び出します。
- mac_fini_ops() 関数をその _fini() エントリポイントで呼び出します。
- misc/mac への依存性とリンクします。

```
# ld -N"misc/mac" xx.o -o xx
```

GLDv3 の MAC 登録関数

GLDv3 インタフェースには、MAC 層への登録中に通知されるドライバエントリポイントと、ドライバによって呼び出される MAC エントリポイントが含まれています。

mac_init_ops() および mac_fini_ops() 関数

```
void mac_init_ops(struct dev_ops *ops, const char *name);
```

GLDv3 デバイスドライバは mod_install(9F) を呼び出す前に、mac_init_ops(9F) 関数をその _init(9E) エントリポイントで呼び出す必要があります。

```
void mac_fini_ops(struct dev_ops *ops);
```

GLDv3 デバイスドライバは mod_remove(9F) を呼び出したあとに、mac_fini_ops(9F) 関数をその _fini(9E) エントリポイントで呼び出す必要があります。

例 19–1 mac_init_ops() および mac_fini_ops() 関数

```
int
_init(void)
{
```

例 19–1 mac_init_ops() および mac_fini_ops() 関数 (続き)

```
        int     rv;
        mac_init_ops(&xx_devops, "xx");
        if ((rv = mod_install(&xx_modlinkage)) != DDI_SUCCESS) {
                mac_fini_ops(&xx_devops);
        }
        return (rv);
}

int
_fini(void)
{
        int     rv;
        if ((rv = mod_remove(&xx_modlinkage)) == DDI_SUCCESS) {
                mac_fini_ops(&xx_devops);
        }
        return (rv);
}
```

mac_alloc() および mac_free() 関数

```
mac_register_t *mac_alloc(uint_t version);
```

mac_alloc(9F) 関数は新しい mac_register 構造体を割り当て、この構造体へのポインタを返します。新しい構造体を mac_register() に渡す前に構造体のメンバーを初期化してください。mac_alloc() が戻る前に、MAC 専用要素が MAC 層によって初期化されます。*version* の値は MAC_VERSION_V1 である必要があります。

```
void mac_free(mac_register_t *mregp);
```

mac_free(9F) 関数は、以前に mac_alloc() によって割り当てられた mac_register 構造体を解放します。

mac_register() および mac_unregister() 関数

```
int mac_register(mac_register_t *mregp, mac_handle_t *mhp);
```

新しいインスタンスを MAC 層に登録するために、GLDv3 ドライバは mac_register(9F) 関数をその attach(9E) エントリポイントで呼び出す必要があります。*mregp* 引数は、mac_register 登録情報構造体へのポインタです。成功した場合、*mhp* 引数は新しい MAC インスタンスの MAC ハンドルへのポインタです。このハンドルは、mac_tx_update()、mac_link_update()、mac_rx() などのほかのルーチンで必要になります。

例 19–2 mac_alloc()、mac_register()、および mac_free() 関数と mac_register 構造体

```
int
xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        mac_register_t        *macp;
```

例 19–2 mac_alloc()、mac_register()、および mac_free() 関数と mac_register 構造体 (続き)

```
/* ... */

        if ((macp = mac_alloc(MAC_VERSION)) == NULL) {
                xx_error(dip, "mac_alloc failed");
                goto failed;
        }

        macp->m_type_ident = MAC_PLUGIN_IDENT_ETHER;
        macp->m_driver = xxp;
        macp->m_dip = dip;
        macp->m_src_addr = xxp->xx_curraddr;
        macp->m_callbacks = &xx_m_callbacks;
        macp->m_min_sdu = 0;
        macp->m_max_sdu = ETHERMTU;
        macp->m_margin = VLAN_TAGSZ;

        if (mac_register(macp, &xxp->xx_mh) == DDI_SUCCESS) {
                mac_free(macp);
                return (DDI_SUCCESS);
        }

/* failed to register with MAC */
        mac_free(macp);
failed:
        /* ... */
}
```

```
int mac_unregister(mac_handle_t mh);
```

mac_unregister(9F) 関数は、以前に mac_register() によって登録された MAC インスタンスを登録解除します。*mh* 引数は、mac_register() によって割り当てられた MAC ハンドルです。mac_unregister() は detach(9E) エントリポイントから呼び出してください。

例 19–3 mac_unregister() 関数

```
int
xx_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
        xx_t        *xxp; /* driver soft state */

        /* ... */

        switch (cmd) {
        case DDI_DETACH:

                if (mac_unregister(xxp->xx_mh) != 0) {
                        return (DDI_FAILURE);
                }
        /* ... */
}
```

GLDv3 の MAC 登録データ構造体

このセクションで説明する構造体は、sys/mac_provider.h ヘッダーファイルで定義されます。sys/mac.h、sys/mac_ether.h、および sys/mac_provider.h の 3 つの MAC ヘッダーファイルを GLDv3 ドライバにインクルードします。その他の MAC 関連ヘッダーファイルはインクルードしないでください。

mac_register(9S) データ構造体は、mac_alloc() によって割り当てられ、mac_register() に渡される MAC 登録情報構造体です。新しい構造体を mac_register() に渡す前に構造体のメンバーを初期化してください。mac_alloc() が戻る前に、MAC 専用要素が MAC 層によって初期化されます。構造体の m_version メンバーは MAC バージョンです。MAC バージョンを変更しないでください。構造体の m_type_ident メンバーは MAC タイプ識別子です。MAC タイプ識別子は MAC_PLUGIN_IDENT_ETHER に設定してください。mac_register 構造体の m_callbacks メンバーは、mac_callbacks 構造体のインスタンスへのポインタです。

mac_callbacks(9S) データ構造体は、デバイスドライバがそのエントリポイントを MAC 層に公開するために使用する構造体です。これらのエントリポイントは、ドライバを制御するために MAC 層によって使用されます。これらのエントリポイントは、アダプタの起動と停止、マルチキャストアドレスの管理、プロミスキュアス (promiscuous) モードの設定、アダプタの機能の照会、プロパティーの取得と設定などのタスクを実行するために使用されます。すべての必須およびオプション GLDv3 エントリポイントの一覧については、表 19–1 を参照してください。mac_register 構造体の m_callbacks フィールドには、mac_callbacks 構造体へのポインタを指定します。

mac_callbacks 構造体の mc_callbacks メンバーは、次のフラグを組み合わせたビットマスクであり、ドライバで実装されるオプションエントリポイントを指定します。mac_callbacks 構造体のほかのメンバーは、ドライバの各エントリポイントへのポインタです。

MC_IOCTL	mc_ioctl() エントリポイントが存在します。
MC_GETCAPAB	mc_getcapab() エントリポイントが存在します。
MC_SETPROP	mc_setprop() エントリポイントが存在します。
MC_GETPROP	mc_getprop() エントリポイントが存在します。
MC_PROPINFO	mc_propinfo() エントリポイントが存在します。
MC_PROPERTIES	すべてのプロパティーエントリポイントが存在します。MC_PROPERTIES を設定すると、MC_SETPROP、MC_GETPROP、および MC_PROPINFO の 3 つのフラグすべてを設定したことになります。

例 19–4 mac_callbacks 構造体

```
#define XX_M_CALLBACK_FLAGS \
    (MC_IOCTL | MC_GETCAPAB | MC_PROPERTIES)
```

例 19–4 mac_callbacks 構造体 *(続き)*

```
static mac_callbacks_t xx_m_callbacks = {
        XX_M_CALLBACK_FLAGS,
        xx_m_getstat,     /* mc_getstat() */
        xx_m_start,       /* mc_start() */
        xx_m_stop,        /* mc_stop() */
        xx_m_promisc,     /* mc_setpromisc() */
        xx_m_multicst,    /* mc_multicst() */
        xx_m_unicst,      /* mc_unicst() */
        xx_m_tx,          /* mc_tx() */
        NULL,             /* Reserved, do not use */
        xx_m_ioctl,       /* mc_ioctl() */
        xx_m_getcapab,    /* mc_getcapab() */
        NULL,             /* Reserved, do not use */
        NULL,             /* Reserved, do not use */
        xx_m_setprop,     /* mc_setprop() */
        xx_m_getprop,     /* mc_getprop() */
        xx_m_propinfo     /* mc_propinfo() */
};
```

GLDv3 の機能

GLDv3 が実装する機能メカニズムでは、GLDv3 ドライバでサポートされている機能をフレームワークで照会して有効化できます。機能を報告するには、mc_getcapab(9E) エントリポイントを使用します。機能がドライバによってサポートされている場合は、機能固有のエントリポイントやフラグなど、その機能についての情報を mc_getcapab() を通して渡します。mc_getcapab() エントリポイントへのポインタを mac_callback 構造体で渡します。mac_callbacks 構造体の詳細は、437 ページの「GLDv3 の MAC 登録データ構造体」を参照してください。

```
boolean_t mc_getcapab(void *driver_handle, mac_capab_t cap, void *cap_data);
```

cap 引数は、照会する機能のタイプを指定します。*cap* に指定できる値は MAC_CAPAB_HCKSUM (ハードウェアチェックサムオフロード) または MAC_CAPAB_LSO (ラージセグメントオフロード) です。機能データをフレームワークに返すには、*cap_data* 引数を使用します。

ドライバが *cap* の機能をサポートする場合、mc_getcapab() エントリポイントは B_TRUE を返す必要があります。ドライバが *cap* の機能をサポートしない場合、mc_getcapab() は B_FALSE を返す必要があります。

例 19–5 mc_getcapab() エントリポイント

```
static boolean_t
xx_m_getcapab(void *arg, mac_capab_t cap, void *cap_data)
{
        switch (cap) {
        case MAC_CAPAB_HCKSUM: {
```

例 19–5 mc_getcapab() エントリポイント (続き)

```
                uint32_t *txflags = cap_data;
                *txflags = HCKSUM_INET_FULL_V4 | HCKSUM_IPHDRCKSUM;
                break;
        }
        case MAC_CAPAB_LSO: {
                /* ... */
                break;
        }
        default:
                return (B_FALSE);
        }
        return (B_TRUE);
}
```

次のセクションからは、サポートされている機能と、対応する機能ごとに返されるデータについて説明します。

ハードウェアチェックサムオフロード

ハードウェアチェックサムオフロードのサポートについてのデータを取得するために、フレームワークは *cap* 引数で MAC_CAPAB_HCKSUM を送信します。439 ページの「ハードウェアチェックサムオフロード機能情報」を参照してください。

ハードウェアによるチェックサム計算が有効なときに、チェックサムオフロードのメタデータを照会したり、パケット単位でのハードウェアによるチェックサム計算のメタデータを取得したりするには、mac_hcksum_get(9F) を使用します。440 ページの「mac_hcksum_get() 関数のフラグ」を参照してください。

チェックサムオフロードのメタデータを設定するには、mac_hcksum_set(9F) を使用します。440 ページの「mac_hcksum_set() 関数のフラグ」を参照してください。

詳細は、442 ページの「ハードウェアによるチェックサム計算: ハードウェア」および443 ページの「ハードウェアによるチェックサム計算: MAC 層」を参照してください。

ハードウェアチェックサムオフロード機能情報

MAC_CAPAB_HCKSUM 機能についての情報をフレームワークに渡すために、ドライバは uint32_t を指し示す *cap_data* で次のフラグの組み合わせを設定する必要があります。これらのフラグは、アウトバウンドパケットに対してドライバが実行できるハードウェアチェックサムオフロードのレベルを示します。

HCKSUM_INET_PARTIAL　1 の補数による部分チェックサム機能

HCKSUM_INET_FULL_V4　IPv4 パケット対象の 1 の補数による完全チェックサム機能

HCKSUM_INET_FULL_V6	IPv6 パケット対象の 1 の補数による完全チェックサム機能
HCKSUM_IPHDRCKSUM	IPv4 ヘッダーのチェックサムオフロード機能

mac_hcksum_get() 関数のフラグ

mac_hcksum_get() の *flags* 引数は次の値の組み合わせです。

HCK_FULLCKSUM	このパケットの完全チェックサムを計算します。
HCK_FULLCKSUM_OK	完全チェックサムがハードウェアで検証済みであり、正確です。
HCK_PARTIALCKSUM	mac_hcksum_get() に渡されるほかのパラメータに基づいて、1 の補数による部分チェックサムを計算します。HCK_PARTIALCKSUM は HCK_FULLCKSUM と相互排他です。
HCK_IPV4_HDRCKSUM	IP ヘッダーのチェックサムを計算します。
HCK_IPV4_HDRCKSUM_OK	IP ヘッダーのチェックサムはハードウェアで検証済みであり、正確です。

mac_hcksum_set() 関数のフラグ

mac_hcksum_set() の *flags* 引数は次の値の組み合わせです。

HCK_FULLCKSUM	完全チェックサムが計算され、*value* 引数を介して渡されました。
HCK_FULLCKSUM_OK	完全チェックサムがハードウェアで検証済みであり、正確です。
HCK_PARTIALCKSUM	部分チェックサムが計算され、*value* 引数を介して渡されました。HCK_PARTIALCKSUM は HCK_FULLCKSUM と相互排他です。
HCK_IPV4_HDRCKSUM	IP ヘッダーのチェックサムが計算され、*value* 引数を介して渡されました。
HCK_IPV4_HDRCKSUM_OK	IP ヘッダーのチェックサムはハードウェアで検証済みであり、正確です。

ラージセグメント (送信) オフロード

ラージセグメント (送信) オフロードのサポートを照会するために、フレームワークは *cap* 引数で MAC_CAPAB_LSO を送信し、mac_capab_lso(9S) 構造体を指し示す *cap_data* に情報が返されることを想定します。フレームワークは mac_capab_lso 構造体を割り当て、この構造体へのポインタを *cap_data* に渡します。mac_capab_lso 構造体は lso_basic_tcp_ipv4(9S) 構造体と lso_flags メンバーで構成されます。ドライバイン

スタンスが TCP/IPv4 での LSO をサポートする場合、LSO_TX_BASIC_TCP_IPV4 フラグを lso_flags に設定し、デバイスインスタンスがサポートする最大ペイロードサイズを lso_basic_tcp_ipv4 構造体の lso_max メンバーに設定します。

パケット単位での LSO のメタデータを取得するには、mac_lso_get(9F) を使用します。このパケットに対して LSO が有効な場合、mac_lso_get() の *flags* 引数に HW_LSO フラグが設定されます。ラージセグメントのセグメンテーション中に使用される最大セグメントサイズ (MSS) は、*mss* 引数によって指示される場所を介して返されます。詳細は、442 ページの「ラージセグメントオフロード」を参照してください。

GLDv3 のデータパス

データパスエントリポイントは、次のコンポーネントで構成されます。

- ドライバによってエクスポートされ、パケット送信のために GLDv3 フレームワークによって呼び出されるコールバック。
- 転送フロー制御およびパケット受信のためにドライバによって呼び出される GLDv3 フレームワークのエントリポイント。

送信データパス

GLDv3 フレームワークは、転送エントリポイント mc_tx(9E) を使用して、メッセージブロックのチェーンをドライバに渡します。mc_tx() エントリポイントへのポインタは mac_callbacks 構造体で指定します。mac_callbacks 構造体の詳細は、437 ページの「GLDv3 の MAC 登録データ構造体」を参照してください。

例 19–6 mc_tx() エントリポイント

```
mblk_t *
xx_m_tx(void *arg, mblk_t *mp)
{
        xx_t    *xxp = arg;
        mblk_t   *nmp;

        mutex_enter(&xxp->xx_xmtlock);

        if (xxp->xx_flags & XX_SUSPENDED) {
                while ((nmp = mp) != NULL) {
                        xxp->xx_carrier_errors++;
                        mp = mp->b_next;
                        freemsg(nmp);
                }
                mutex_exit(&xxp->xx_xmtlock);
                return (NULL);
        }

        while (mp != NULL) {
                nmp = mp->b_next;
                mp->b_next = NULL;
```

例 19–6 mc_tx() エントリポイント (続き)

```
                if (!xx_send(xxp, mp)) {
                        mp->b_next = nmp;
                        break;
                }
                mp = nmp;
        }
        mutex_exit(&xxp->xx_xmtlock);

        return (mp);
}
```

次のセクションでは、ハードウェアへのデータ転送に関連する事項について説明します。

フロー制御

ハードウェアリソース不足のためドライバがパケットを送信できない場合、ドライバは送信できなかったパケットのサブチェーンを返します。利用可能な記述子があとから増えたとき、ドライバは mac_tx_update(9F) を呼び出してフレームワークに通知する必要があります。

ハードウェアによるチェックサム計算: ハードウェア

ドライバでハードウェアチェックサムのサポート (439 ページの「ハードウェアチェックサムオフロード」を参照) を指定した場合、ドライバは次のタスクを実行する必要があります。

- mac_hcksum_get(9F) を使用して、ハードウェアチェックサムのメタデータの全パケットを確認します。
- 必要なチェックサム計算を実行するようにハードウェアをプログラミングします。

ラージセグメントオフロード

ドライバで LSO 機能 (440 ページの「ラージセグメント (送信) オフロード」を参照) を指定した場合、ドライバは mac_lso_get(9F) を使用して、パケットに対して LSO を実行する必要があるかどうかを照会する必要があります。

仮想 LAN: ハードウェア

管理者が VLAN を構成した場合、アウトバウンドパケットが mc_tx() エントリポイントを介してドライバに渡される前に、MAC 層が必要な VLAN ヘッダーをアウトバウンドパケットに挿入します。

受信データパス

ドライバの割り込みハンドラで mac_rx(9F) 関数を呼び出して、1 つ以上のパケットのチェーンをスタック上位の MAC 層に渡します。mac_rx() の呼び出し中に mutex ロックまたはその他のロックを保持しないでください。特に、mac_rx() の呼び出し中に転送スレッドによって取得される可能性があるロックを保持しないでください。MAC 層に送信される必要があるパケットについては、mc_unicst(9E)を参照してください。

次のセクションでは、MAC 層へのデータ送信に関連する事項について説明します。

ハードウェアによるチェックサム計算: MAC 層

ドライバでハードウェアチェックサムのサポート (439 ページの「ハードウェアチェックサムオフロード」を参照) を指定した場合、ドライバは mac_hcksum_set(9F) 関数を使用して、ハードウェアによるチェックサム計算のメタデータをパケットと関連付ける必要があります。

仮想 LAN: MAC 層

VLAN パケットは、そのタグとともに MAC 層に渡される必要があります。パケットから VLAN ヘッダーを取り除かないでください。

GLDv3 の状態変更通知

ドライバは次の関数を呼び出して、ドライバの状態が変化したことをネットワークスタックに通知できます。

```
void mac_tx_update(mac_handle_t mh);
```

mac_tx_update(9F) 関数は、利用可能な TX 記述子が増えたことをフレームワークに通知します。空でないパケットチェーンを mc_tx() が返した場合、ドライバはリソースが利用可能になったらただちに mac_tx_update() を呼び出して、未送信として返送されたパケットの再送を試みるよう MAC 層に通知する必要があります。mc_tx() エントリポイントの詳細は、441 ページの「送信データパス」を参照してください。

```
void mac_link_update(mac_handle_t mh, link_state_t new_state);
```

mac_link_update(9F) 関数は、メディアリンクの状態が変化したことを MAC 層に通知します。*new_state* 引数は次のいずれかの値である必要があります。

LINK_STATE_UP	メディアリンクは稼働しています。
LINK_STATE_DOWN	メディアリンクはダウンしています。
LINK_STATE_UNKNOWN	メディアリンクの状態は不明です。

GLDv3 のネットワーク統計情報

デバイスドライバは、管理対象のデバイスインスタンスに関する一連の統計情報を管理します。MAC 層は、ドライバの mc_getstat(9E) エントリポイントを介してこれらの統計情報を照会します。

```
int mc_getstat(void *driver_handle, uint_t stat, uint64_t *stat_value);
```

GLDv3 フレームワークは *stat* を使用して、照会する統計情報を指定します。ドライバは *stat_value* を使用して、*stat* で指定された統計情報の値を返します。統計情報の値が返された場合、mc_getstat() は 0 を返す必要があります。*stat* の統計情報がドライバでサポートされていない場合、mc_getstat() は ENOTSUP を返す必要があります。

GLDv3 でサポートされる統計情報は、汎用の MAC 統計情報と Ethernet 固有の統計情報を合わせたものです。サポートされる統計情報の完全な一覧については、mc_getstat(9E) のマニュアルページを参照してください。

例 19–7 mc_getstat() エントリポイント

```
int
xx_m_getstat(void *arg, uint_t stat, uint64_t *val)
{
        xx_t    *xxp = arg;

        mutex_enter(&xxp->xx_xmtlock);
        if ((xxp->xx_flags & (XX_RUNNING|XX_SUSPENDED)) == XX_RUNNING)
                xx_reclaim(xxp);
        mutex_exit(&xxp->xx_xmtlock);

        switch (stat) {
        case MAC_STAT_MULTIRCV:
                *val = xxp->xx_multircv;
                break;
        /* ... */
        case ETHER_STAT_MACRCV_ERRORS:
                *val = xxp->xx_macrcv_errors;
                break;
        /* ... */
        default:
                return (ENOTSUP);
        }
        return (0);
}
```

GLDv3 のプロパティー

プロパティーの不変属性を返すには、mc_propinfo(9E) エントリポイントを使用します。この情報にはアクセス権、デフォルト値、および値の許容範囲が含まれていま

す。この特定のドライバインスタンスのプロパティー値を設定するには、mc_setprop(9E) を使用します。プロパティーの現在の値を返すには、mc_getprop(9E)を使用します。

プロパティーとその型の完全な一覧については、mc_propinfo(9E) のマニュアルページを参照してください。

mc_propinfo() エントリポイントは、mac_prop_info_set_perm()、mac_prop_info_set_default()、および mac_prop_info_set_range() 関数を呼び出して、デフォルト値、アクセス権、値の許容範囲など、照会するプロパティーの特定の属性を関連付けます。

mac_prop_info_set_default_uint8(9F)、mac_prop_info_set_default_str(9F)、および mac_prop_info_set_default_link_flowctrl(9F) 関数は、デフォルト値を特定のプロパティーと関連付けます。mac_prop_info_set_range_uint32(9F) 関数は、特定のプロパティーについて値の許容範囲を関連付けます。

mac_prop_info_set_perm(9F) 関数はプロパティーのアクセス権を指定します。アクセス権は次のいずれかの値になります。

MAC_PROP_PERM_READ　　プロパティーは読み取り専用

MAC_PROP_PERM_WRITE　　プロパティーは書き込み専用

MAC_PROP_PERM_RW　　プロパティーは読み書き可能

mc_propinfo() エントリポイントで特定のプロパティーについて mac_prop_info_set_perm() を呼び出さない場合、GLDv3 フレームワークはそのプロパティーのアクセス権を読み書き可能 (MAC_PROP_PERM_RW) と想定します。

mc_propinfo(9E) のマニュアルページに列挙されたプロパティーに加えて、ドライバではそのドライバ専用のプロパティーも公開できます。ドライバでサポートするドライバ専用プロパティーを指定するには、mac_register 構造体の m_priv_props フィールドを使用します。フレームワークは MAC_PROP_PRIVATE プロパティー ID を mc_setprop()、mc_getprop()、または mc_propinfo() に渡します。詳細は、mc_propinfo (9E) のマニュアルページを参照してください。

GLDv3 のインタフェースのサマリー

次の表は、GLDv3 ネットワークデバイスドライバのフレームワークを構成するエントリポイント、その他の DDI 関数、およびデータ構造体の一覧です。

表 19–1 GLDv3 のインタフェース

インタフェース名	説明
必須のエントリポイント	
mc_getstat(9E)	ドライバからネットワーク統計情報を取得します。444 ページの「GLDv3 のネットワーク統計情報」を参照してください。
mc_start(9E)	ドライバインスタンスを起動します。GLDv3 フレームワークは、何らかの操作が試みられる前に起動エントリポイントを呼び出します。
mc_stop(9E)	ドライバインスタンスを停止します。MAC 層は、デバイスが切り離される前に停止エントリポイントを呼び出します。
mc_setpromisc(9E)	デバイスドライバインスタンスのプロミスキュアス (promiscuous) モードを変更します。
mc_multicst(9E)	マルチキャストアドレスを追加または削除します。
mc_unicst(9E)	プライマリユニキャストアドレスを設定します。デバイスは mac_rx() を使用して、宛先 MAC アドレスが新しいユニキャストアドレスと一致するパケットの送信を開始する必要があります。mac_rx() の詳細は、443 ページの「受信データパス」を参照してください。
mc_tx(9E)	1 つ以上のパケットを送信します。441 ページの「送信データパス」を参照してください。
省略可能なエントリポイント	
mc_ioctl(9E)	省略可能な ioctl ドライバインタフェース。この機能はデバッグ目的の使用のみが想定されています。
mc_getcapab(9E)	機能を取得します。438 ページの「GLDv3 の機能」を参照してください。
mc_setprop(9E)	プロパティー値を設定します。444 ページの「GLDv3 のプロパティー」を参照してください。
mc_getprop(9E)	プロパティー値を取得します。444 ページの「GLDv3 のプロパティー」を参照してください。

表 19–1 GLDv3 のインタフェース (続き)

インタフェース名	説明
mc_propinfo(9E)	プロパティーについての情報を取得します。444 ページの「GLDv3 のプロパティー」を参照してください。
データ構造体	
mac_register(9S)	登録情報。437 ページの「GLDv3 の MAC 登録データ構造体」を参照してください。
mac_callbacks(9S)	ドライバコールバック。437 ページの「GLDv3 の MAC 登録データ構造体」を参照してください。
mac_capab_lso(9S)	LSO メタデータ。440 ページの「ラージセグメント (送信) オフロード」を参照してください。
lso_basic_tcp_ipv4(9S)	TCP/IPv4 用の LSO メタデータ。440 ページの「ラージセグメント (送信) オフロード」を参照してください。
MAC 登録関数	
mac_alloc(9F)	新しい mac_register 構造体を割り当てます。434 ページの「GLDv3 の MAC 登録」を参照してください。
mac_free(9F)	mac_register 構造体を解放します。
mac_register(9F)	MAC 層に登録します。
mac_unregister(9F)	MAC 層から登録解除します。
mac_init_ops(9F)	ドライバの dev_ops(9S) 構造体を初期化します。
mac_fini_ops(9F)	ドライバの dev_ops 構造体を解放します。
データ転送関数	
mac_rx(9F)	受信したパケットを上位層に渡します。443 ページの「受信データパス」を参照してください。
mac_tx_update(9F)	TX リソースが利用可能です。443 ページの「GLDv3 の状態変更通知」を参照してください。
mac_link_update(9F)	リンク状態が変化しました。
mac_hcksum_get(9F)	ハードウェアチェックサム情報を取得します。439 ページの「ハードウェアチェックサムオフロード」および441 ページの「送信データパス」を参照してください。

表 19–1 GLDv3 のインタフェース (続き)

インタフェース名	説明
mac_hcksum_set(9F)	ハードウェアチェックサム情報を添付します。439 ページの「ハードウェアチェックサムオフロード」および443 ページの「受信データパス」を参照してください。
mac_lso_get(9F)	LSO 情報を取得します。440 ページの「ラージセグメント (送信) オフロード」を参照してください。
プロパティー関数	
mac_prop_info_set_perm(9F)	プロパティーのアクセス権を設定します。444 ページの「GLDv3 のプロパティー」を参照してください。
mac_prop_info_set_default_uint8(9F)、mac_prop_info_set_default_str(9F)、mac_prop_info_set_default_link_flowctrl(9F)	プロパティー値を設定します。
mac_prop_info_set_range_uint32(9F)	プロパティー値の範囲を設定します。

GLDv2 ネットワークデバイスドライバフレームワーク

GLDv2 は、ローカルエリアネットワーク用のデバイスドライバをサポートする、マルチスレッド化された、ロード可能かつ複製可能なカーネルモジュールです。Oracle Solaris OS のローカルエリアネットワーク (LAN) デバイスドライバは、Data Link Provider Interface (DLPI) を使用してネットワークプロトコルスタックと通信する STREAMS ベースのドライバです。これらのプロトコルスタックは、ネットワークドライバを使用して LAN 上でパケットを送受信します。GLDv2 では、STREAMS および DLPI の機能のうち、Oracle Solaris LAN ドライバのためのほとんどの機能を実装しています。GLDv2 は、多くのネットワークドライバで共有可能な共通コードを提供します。GLDv2 を使用することで、コードの重複が減少し、ネットワークドライバが簡素化されます。

GLDv2 の詳細は、gld(7D) のマニュアルページを参照してください。

STREAMS ドライバについては、『STREAMS Programming Guide』のパート II「Kernel Interface」に記載されています。特に、STREAMS ガイドの第 9 章「STREAMS Drivers」を参照してください。STREAMS フレームワークはメッセージベースのフレームワークです。STREAMS ドライバ固有のインタフェースには、STREAMS メッセージキュー処理エントリポイントが含まれています。

DLPI は、OSI 参照モデルのデータリンク層のデータリンクサービス (DLS) に対するインタフェースの仕様を定義します。DLPI により、DLS ユーザーは DLS プロバイダのプロトコルについて特別な知識がなくても、仕様に準拠した各種 DLS プロバイダ

にアクセスして利用できます。DLPI では、M_PROTO および M_PCPROTO タイプの STREAMS メッセージを利用して DLS プロバイダにアクセスする仕様が定められています。DLPI モジュールは、STREAMS ioctl 呼び出しを使用して MAC 下位層にリンクします。Oracle Solaris 固有の DLPI 拡張機能を含む DLPI プロトコルの詳細は、dlpi(7P) のマニュアルページを参照してください。DLPI 全般に関する情報は、DLPI 標準 (http://www.opengroup.org/pubs/catalog/c811.htm) を参照してください。

GLDv2 を使用して実装される Oracle Solaris ネットワークドライバは、次の 2 つの部分に分けることができます。

- 汎用コンポーネント。STREAMS および DLPI インタフェースを扱います。
- デバイス固有コンポーネント。特定のハードウェアデバイスを扱います。
  - misc/gld への依存性とリンクすることにより、GLDv2 モジュールへのドライバの依存性を示します。GLDv2 モジュールの場所は、SPARC システムでは /kernel/misc/sparcv9/gld、64 ビット x86 システムでは /kernel/misc/amd64/gld、32 ビット x86 システムでは /kernel/misc/gld です。
  - GLDv2 への登録: ドライバはその attach(9E) エントリポイントで、ドライバのほかのエントリポイントへのポインタを GLDv2 に知らせます。GLDv2 はこれらのポインタを使用して、gld(9E) エントリポイントを呼び出します。
  - gld(9F) 関数を呼び出して、データを処理したり、ほかの GLDv2 サービスを利用したりします。gld_mac_info(9S) 構造体は、GLDv2 とデバイス固有ドライバ間のプライマリデータインタフェースです。

GLDv2 ドライバは、完全形式の MAC 層パケットを処理する必要があり、論理リンク制御 (LLC) 処理を実行してはなりません。

このセクションの内容は次のとおりです。

GLDv2 のデバイスサポート

GLDv2 フレームワークは次の種類のデバイスをサポートします。

- DL_ETHER: ISO 8802-3、IEEE 802.3 プロトコル
- DL_TPR: IEEE 802.5、Token Passing Ring
- DL_FDDI: ISO 9314-2、Fibre Distributed Data Interface

Ethernet V2 および ISO 8802-3 (IEEE 802.3)

DL_ETHER タイプとして宣言されたデバイスについて、GLDv2 は Ethernet V2 および ISO 8802-3 (IEEE 802.3) の両方のパケット処理をサポートします。Ethernet V2 では、ユーザーはプロバイダのプロトコルについて特別な知識がなくても、仕様に準拠したデータリンクサービスプロバイダにアクセスできます。サービスアクセスポイント (SAP) は、ユーザーがサービスプロバイダと通信するときに通過するポイントです。

0 - 255 の範囲の SAP 値にバインドされたストリームは同等として扱われ、ユーザーが 8802-3 モードの使用を求めていることを示します。DL_BIND_REQ の SAP 値がこの範囲内の場合、GLDv2 は、そのストリームの後続の各 DL_UNITDATA_REQ メッセージの長さを計算します。この長さには 14 バイトのメディアアクセス制御 (MAC) ヘッダーは含まれません。GLDv2 はその後、計算した長さの 8802-3 フレームを MAC フレームヘッダーの type フィールドに転送します。この長さは 1500 を超えることはありません。

0 - 1500 の範囲の type フィールドを持つフレームは 8802-3 フレームとみなされます。これらのフレームは、8802-3 モードで開いているすべてのストリームに経路指定されます。SAP 値が 0 - 255 の範囲であるストリームは 8802-3 モードとみなされます。複数のストリームが 8802-3 モードである場合、着信フレームは複製され、これらのストリームに経路指定されます。

1500 より大きい SAP 値にバインドされたストリームは Ethernet V2 モードとみなされます。これらのストリームは、Ethernet MAC ヘッダーの type の値が、ストリームのバインド先である SAP の値と完全一致する着信パケットを受信します。

TPR と FDDI: SNAP 処理

DL_TPR および DL_FDDI のメディアタイプ向けに、GLDv2 は最小限の SNAP (Sub-Net Access Protocol) 処理を実装します。この処理は、255 より大きい SAP 値にバインドされたあらゆるストリームが対象となります。0 - 255 の範囲の SAP 値は LLC SAP 値です。そのような値は、メディアパケットフォーマットによって通常どおりに伝送されます。SAP 値が 255 より大きい場合、16 ビット Ethernet V2 スタイルの SAP 値を伝送するには、LLC ヘッダーに従属する SNAP ヘッダーが必要です。

SNAP ヘッダーは、宛先 SAP 0xAA が指定された LLC ヘッダーとして伝送されます。SAP 値が 255 より大きい発信パケットには、次の形式の LLC+SNAP ヘッダーが必要です。

```
AA AA 03 00 00 00 XX XX
```

XX XX は、Ethernet V2 スタイルの type に対応する 16 ビット SAP を表します。このヘッダーは、0 以外の組織固有識別子フィールドのサポートに固有のものです。03 以外の LLC 制御フィールドは、SAP 0xAA を持つ LLC パケットとみなされます。これ以外の SNAP フォーマットを使用するクライアントは、LLC を使用し、SAP 0xAA にバインドする必要があります。

着信パケットが前述したフォーマットに適合しているかどうかのチェックが行われます。適合するパケットは、パケットの 16 ビット SNAP タイプにバインドされたすべてのストリームと照合されます。また、これらのパケットは LLC SNAP SAP 0xAA に一致するものとみなされます。

LLC SAP として受信されたすべてのパケットは、メディアタイプ DL_ETHER の項目で説明したように、LLC SAP にバインドされているすべての上位層ストリームに渡されます。

TPR: 発信元ルーティング

DL_TPR タイプのデバイス向けに、GLDv2 は発信元ルーティングの最小限のサポートを実装します。

発信元ルーティングのサポートには次のタスクが含まれています。

- ブリッジメディアを介して送信されるパケットのルーティング情報を指定します。ルーティング情報は MAC ヘッダーに格納されます。この情報は経路の特定に使用されます。
- 経路を学習します。
- 可能性のある複数の経路についての情報を要求し、そのような情報の要求に応答します。
- 利用可能な経路の中から選択します。

発信元ルーティングにより、発信パケットの MAC ヘッダーにルーティング情報フィールドが付加されます。また、このサポートにより着信パケットの経路指定情報フィールドが認識されます。

GLDv2 の発信元ルーティングのサポートでは、ISO 8802-2 (IEEE 802.2) のセクション 9 で定義されている、完全な経路特定エンティティー (route determination entity、RDE) は実装されません。ただしこのサポートでは、同一ネットワークまたはブリッジ接続されたネットワークに存在する可能性がある、あらゆる RDE 実装との相互運用が可能です。

GLDv2 の DLPI プロバイダ

GLDv2 は、Style 1 および Style 2 両方の DLPI プロバイダを実装します。物理接続点 (PPA) は、システムが物理通信メディアと接続するポイントです。その物理メディアを介したすべての通信が PPA を通過します。Style 1 プロバイダは、開かれているメジャーデバイスまたはマイナーデバイスに基づいて、ストリームを特定の PPA に接続します。Style 2 プロバイダの場合、DLS ユーザーは DL_ATTACH_REQ を使用して、接続先 PPA を明示的に指定する必要があります。この場合、open(9E) がユーザーと GLDv2 の間にストリームを作成したあとに、DL_ATTACH_REQ が特定の PPA をそのストリームと関連付けます。Style 2 は 0 のマイナー番号によって示されます。マイナー番号が 0 でないデバイスノードが開かれている場合、Style 1 が示され、マイナー番号から 1 を引いた PPA が関連付けられます。Style 1 と Style 2 の両方が開いている場合、デバイスは複製されます。

GLDv2 の DLPI プリミティブ

GLDv2 はいくつかの DLPI プリミティブを実装します。DL_INFO_REQ プリミティブは、DLPI ストリームについての情報を要求します。メッセージは 1 つの M_PROTO メッセージブロックで構成されます。GLDv2 はこの要求に対して、デバイス依存の値を DL_INFO_ACK 応答で返します。これらの値は、gld_register(9F) 関数に渡す gld_mac_info(9S) 構造体で GLDv2 ベースのドライバが指定した情報に基づきます。

すべての GLDv2 ベースのドライバに代わって、GLDv2 が次の値を返します。

- バージョンは DL_VERSION_2 です。
- サービスモードは DL_CLDLS です。GLDv2 はコネクションレスモードのサービスを実装します。
- プロバイダのスタイルは、ストリームが開かれた方法に応じて DL_STYLE1 または DL_STYLE2 です。
- 省略可能なサービス品質 (QOS) サポートはありません。QOS フィールドは 0 です。

注 – DLPI 仕様とは異なり、ストリームが PPA に接続される前であっても、GLDv2 は DL_INFO_ACK でデバイスの正確なアドレス長およびブロードキャストアドレスを返します。

DL_ATTACH_REQ プリミティブは、PPA をストリームと関連付けるために使用されます。この要求は、Style 2 の DLS プロバイダが通信を行う物理メディアを特定するために必要です。完了すると、状態が DL_UNATTACHED から DL_UNBOUND に変化します。メッセージは 1 つの M_PROTO メッセージブロックで構成されます。Style 1 モードの使用時はこの要求は許可されません。Style 1 を使用して開かれたストリームは、オープンの完了時点ですでに PPA に接続されているためです。

DL_DETACH_REQ プリミティブは、PPA をストリームから切り離すことを要求します。この切り離しは、ストリームが Style 2 を使用して開かれた場合にのみ許可されます。

DL_BIND_REQ および DL_UNBIND_REQ プリミティブは、DLSAP (データリンクサービスアクセスポイント) をストリームにバインドまたはバインド解除します。ストリームに関連付けられた PPA は、そのストリームの DL_BIND_REQ の処理が完了する前に初期化を完了します。複数のストリームを同じ SAP にバインドできます。この場合、それぞれのストリームが、その SAP で受信されたすべてのパケットのコピーを受信します。

DL_ENABMULTI_REQ および DL_DISABMULTI_REQ プリミティブは、個々のマルチキャストグループアドレスの受信を有効または無効にします。アプリケーションまたはその他の DLS ユーザーは、これらのプリミティブを繰り返し使用するとマルチキャストアドレスの集合を作成または変更できます。これらのプリミティブが受け入れられるには、ストリームが PPA に接続されている必要があります。

DL_PROMISCON_REQ および DL_PROMISCOFF_REQ プリミティブは、ストリーム単位でプロミスキュアス (promiscuous) モードを有効または無効にします。この制御は物理レベルまたは SAP レベルで機能します。DL プロバイダは、メディアで受信したすべてのメッセージを DLS ユーザーに経路指定します。このルーティングは、DL_DETACH_REQ または DL_PROMISCOFF_REQ が受信されるか、ストリームが閉じられるまで有効です。物理レベルのプロミスキュアスの受信は、メディア上のすべてのパケットに対して、またはマルチキャストパケットに限定して指定できます。

注 – これらのプロミスキュアス (promiscuous) モードプリミティブが受け入れられるには、ストリームが PPA に接続されている必要があります。

DL_UNITDATA_REQ プリミティブは、コネクションレス型転送でデータを送信するために使用されます。このサービスは肯定応答を伴わないため、配信は保証されません。メッセージは 1 つの M_PROTO メッセージブロックと、それに続く 1 つ以上の M_DATA ブロック (少なくとも 1 バイトのデータを含む) で構成されます。

DL_UNITDATA_IND タイプは、パケットをアップストリームに渡すときに使用します。パケットは、プリミティブを DL_UNITDATA_IND に設定した M_PROTO メッセージに格納されます。

DL_PHYS_ADDR_REQ プリミティブは、ストリームに接続された PPA とその時点で関連付けられている MAC アドレスを要求します。アドレスは DL_PHYS_ADDR_ACK プリミティブによって返されます。Style 2 を使用している場合、このプリミティブは DL_ATTACH_REQ が成功したあとに限り有効です。

DL_SET_PHYS_ADDR_REQ プリミティブは、ストリームに接続された PPA とその時点で関連付けられている MAC アドレスを変更します。このプリミティブは、現在および将来にわたってこのデバイスに接続されるすべてのストリームに影響を及ぼしま

す。アドレスの変更後は、現在または将来に開かれてこのデバイスに接続されるすべてのストリームが、この新しい物理アドレスを取得します。新しい物理アドレスは、このプリミティブを使用して物理アドレスをふたたび変更するか、またはドライバが再ロードされるまで有効です。

注 – スーパーユーザーは、ほかのストリームが同じ PPA にバインドされている間も PPA の物理アドレスを変更できます。

DL_GET_STATISTICS_REQ プリミティブは、ストリームに接続された PPA に関連する統計情報を含む DL_GET_STATISTICS_ACK 応答を要求します。DL_ATTACH_REQ を使用して Style 2 のストリームを特定の PPA に接続しておかないと、このプリミティブは成功しません。

GLDv2 の入出力制御関数

GLDv2 は、ここで説明する ioctl *ioc_cmd* 関数を実装します。認識できない ioctl コマンドを受信した場合、gld(9E) で説明しているように、GLDv2 はそのコマンドをデバイス固有ドライバの gldm_ioctl() ルーチンに渡します。

DLIOCRAW ioctl 関数は、snoop(1M) コマンドをはじめとした一部の DLPI アプリケーションで使用されます。DLIOCRAW コマンドはストリームを raw モードにします。raw モードでは、ドライバはパケットを DL_UNITDATA_IND 形式に変換せずに、MAC レベルの着信パケットをそのまま M_DATA メッセージでアップストリームに渡します。DL_UNITDATA_IND 形式は、通常時に着信パケットの報告に使用されます。パケットの SAP フィルタリングは、raw モードのストリームに対しても実行されます。ストリームのユーザーがすべての着信パケットの受信を求める場合、ユーザーは適切なプロミスキュアス (promiscuous) モードを選択する必要もあります。raw モードの選択に成功したあとも、アプリケーションは完全形式のパケットを、転送用の M_DATA メッセージとしてドライバに送信できます。DLIOCRAW には引数がありません。いったん raw モードが有効になると、ストリームを閉じるまでモードは変化しません。

GLDv2 ドライバの要件

GLDv2 ベースのドライバでは、ヘッダーファイル <sys/gld.h> をインクルードする必要があります。

また、GLDv2 ベースのドライバは、次のように -N*"misc/gld"* オプションを使用してリンクする必要があります。

```
%ld -r -N"misc/gld" xx.o -o xx
```

GLDv2 では、デバイス固有ドライバのために次の関数が実装されています。

- open(9E)
- close(9E)
- put(9E) (STREAMS に必要)
- srv(9E) (STREAMS に必要)
- getinfo(9E)

module_info(9S) 構造体の mi_idname 要素は、ドライバの名前を指定する文字列です。この文字列は、ファイルシステムで定義されたドライバモジュールの名前と完全に一致する必要があります。

読み取り側の qinit(9S) 構造体では、次の要素を指定します。

```
qi_putp      NULL

qi_srvp      gld_rsrv

qi_qopen     gld_open

qi_qclose    gld_close
```

書き込み側の qinit(9S) 構造体では、次の要素を指定します。

```
qi_putp      gld_wput

qi_srvp      gld_wsrv

qi_qopen     NULL

qi_qclose    NULL
```

dev_ops(9S) 構造体の devo_getinfo 要素では、getinfo(9E) ルーチンとして gld_getinfo を指定します。

ドライバの attach(9E) 関数は、ハードウェア固有デバイスドライバを GLDv2 の機能と関連付けます。attach() はその後、デバイスおよびドライバを使用できるように準備します。

attach(9E) 関数は、gld_mac_alloc() を使用して gld_mac_info(9S) 構造体を割り当てます。ドライバは通常、macinfo 構造体で定義されているよりも多くの情報をデバイスごとに保存する必要があります。ドライバで、必要な追加のデータ構造体を割り当て、その構造体へのポインタを gld_mac_info(9S) 構造体の gldm_private メンバーに保存します。

attach(9E) ルーチンは、gld_mac_info(9S) のマニュアルページの説明に従って macinfo 構造体を初期化する必要があります。その後、attach() ルーチンは gld_register() を呼び出して、ドライバを GLDv2 モジュールとリンクします。ドライバでは必要に応じてレジスタをマップし、完全に初期化して、gld_register() を呼び出す前に割り込みを受け付けるように準備してください。attach(9E) 関数は割

り込みを追加しますが、デバイスによってこれらの割り込みを生成させてはなりません。ドライバでは、ハードウェアが静止状態であることを保証するために、gld_register() を呼び出す前にハードウェアをリセットしてください。gld_register() が呼び出される前にデバイスが割り込みを生成する可能性がある状態にデバイスが陥らないようにする必要があります。デバイスはあとから、GLDv2 がドライバの gldm_start() エントリポイントを呼び出したときに起動されます。このエントリポイントについては gld(9E) のマニュアルページで説明しています。gld_register() が成功したあとであれば、GLDv2 によって gld(9E) エントリポイントをいつでも呼び出すことができます。

gld_register() が成功した場合、attach(9E) ルーチンは DDI_SUCCESS を返します。gld_register() が失敗した場合、DDI_FAILURE が返されます。エラーが発生した場合、attach(9E) ルーチンは、gld_register() が呼び出される前に割り当てられたすべてのリソースを解放します。さらにその後、接続ルーチンは DDI_FAILURE を返します。エラーの起きた macinfo 構造体は決して再利用しないでください。そのような構造体は、gld_mac_free() を使用して解放してください。

detach(9E) 関数は、gld_unregister() を呼び出すことによって、ドライバを GLDv2 から登録解除することを試みます。gld_unregister() の詳細は、gld(9F) のマニュアルページを参照してください。detach(9E) ルーチンは ddi_get_driver_private(9F) を使用することで、必要な gld_mac_info(9S) 構造体へのポインタをデバイスの非公開データから取得できます。gld_unregister() は、ドライバを切り離すことのできない特定の条件をチェックします。チェックに失敗した場合、gld_unregister() は DDI_FAILURE を返します。この場合、ドライバの detach(9E) ルーチンはデバイスを動作状態にしたまま DDI_FAILURE を返す必要があります。

チェックに成功した場合、gld_unregister() はデバイスの割り込み中止を確認します。必要に応じて、ドライバの gldm_stop() ルーチンが呼び出されます。ドライバは GLDv2 フレームワークからリンク解除されます。その後、gld_unregister() が DDI_SUCCESS を返します。この場合、detach(9E) ルーチンは割り込みを削除し、attach(9E) ルーチンで割り当てられたすべての macinfo データ構造体を gld_mac_free() を使用して解放します。その後、detach() ルーチンは DDI_SUCCESS を返します。ルーチンでは gld_mac_free() を呼び出す前に割り込みを削除する必要があります。

GLDv2 のネットワーク統計情報

Oracle Solaris ネットワークドライバは統計情報変数を実装する必要があります。GLDv2 はいくつかのネットワーク統計情報を集計しますが、その他の統計情報は GLDv2 ベースのドライバによって計測される必要があります。GLDv2 は、GLDv2 ベースのドライバによる、標準的なネットワークドライバ統計情報の報告をサポートします。GLDv2 は kstat(7D) および kstat(9S) メカニズムを使用して統計情報を報告します。DLPI コマンド DL_GET_STATISTICS_REQ を使用して現在の統計情報カ

ウンタを取得することもできます。すべての統計情報は符号なしで管理されます。特に指定のないかぎり、統計情報は 32 ビットです。

GLDv2 は次の統計情報を管理および報告します。

`rbytes64` インタフェース上で正常に受信された合計バイト数。64 ビット統計情報を格納します。

`rbytes` インタフェース上で正常に受信された合計バイト数。

`obytes64` インタフェース上で転送を要求した合計バイト数。64 ビット統計情報を格納します。

`obytes` インタフェース上で転送を要求した合計バイト数。

`ipackets64` インタフェース上で正常に受信された合計パケット数。64 ビット統計情報を格納します。

`ipackets` インタフェース上で正常に受信された合計パケット数。

`opackets64` インタフェース上で転送を要求した合計パケット数。64 ビット統計情報を格納します。

`opackets` インタフェース上で転送を要求した合計パケット数。

`multircv` グループアドレスや機能アドレスを含む、正常に受信されたマルチキャストパケット数 (`long`)。

`multixmt` グループアドレスや機能アドレスを含む、転送が要求されたマルチキャストパケット数 (`long`)。

`brdcstrcv` 正常に受信されたブロードキャストパケット数 (`long`)。

`brdcstxmt` 転送を要求したブロードキャストパケット数 (`long`)。

`unknowns` どのストリームにも受け入れられなかった有効受信パケット数 (`long`)。

`noxmtbuf` 転送バッファーがビジーだったか、転送用にバッファーを割り当てることができなかったため出力時に破棄されたパケット数 (`long`)。

`blocked` キューがフロー制御されていたため受信パケットをストリームに入れることができなかった回数 (`long`)。

`xmtretry` リソース不足のため遅延されたあとに転送が再試行された回数 (`long`)。

`promisc` インタフェースの現在の「プロミスキュアス」状態 (文字列)。

デバイス依存ドライバは、次の統計情報を追跡し、インスタンスごとに専用の構造体に格納します。統計情報を報告するために、GLDv2 はドライバの `gldm_get_stats()` エントリポイントを呼び出します。`gldm_get_stats()` はその

後、gld_stats(9S) 構造体内のデバイス固有の統計情報を更新します。詳細は、gldm_get_stats(9E) のマニュアルページを参照してください。GLDv2 はその後、次に示す名前付き統計情報変数を使用して、更新された統計情報を報告します。

ifspeed	インタフェースの現在の推定帯域幅 (ビット/秒)。64 ビット統計情報を格納します。
media	デバイスで使用している現在のメディアタイプ (文字列)。
intr	割り込みハンドラが呼び出されて割り込みが発生した回数 (long)。
norcvbuf	受信用バッファーを割り当てることができなかったために、有効な着信パケットが破棄されたことが判明している回数 (long)。
ierrors	受信されたがエラーにより処理できなかった合計パケット数 (long)。
oerrors	エラーが原因で正常に転送されなかった合計パケット数 (long)。
missed	受信時にハードウェアによってドロップされたことが判明しているパケット数 (long)。
uflo	転送時に FIFO がアンダーフローした回数 (long)。
oflo	受信中に受信側がオーバーフローした回数 (long)。

次の統計情報グループは、DL_ETHER 型のネットワークに適用されます。これらの統計情報は、前述のように、そのタイプのデバイス固有ドライバによって管理されます。

align_errors	受信時にフレームエラーが発生した (整数個のオクテットが含まれていなかった) パケット数 (long)。
fcs_errors	受信時に CRC エラーが記録されたパケット数 (long)。
duplex	インタフェースの現在の二重モード (文字列)。
carrier_errors	転送試行時にキャリアが失われたか、または検出されなかった回数 (long)。
collisions	転送中の Ethernet 衝突数 (long)。
ex_collisions	転送時に発生した衝突が多すぎて、転送エラーとなったフレーム数 (long)。
tx_late_collisions	遅れて (512 ビットタイム) 発生した転送衝突の回数 (long)。
defer_xmts	衝突は発生しなかったが、メディアがビジー状態だったために初回の転送試行が遅延されたパケット数 (long)。
first_collisions	衝突が 1 回だけ発生したが正常に転送されたパケット数。
multi_collisions	複数の衝突が発生したが正常に転送されたパケット数。

sqe_errors	SQE テストエラーが報告された回数。
macxmt_errors	キャリアエラーと衝突エラー以外の転送 MAC エラーが発生したパケット数。
macrcv_errors	align_errors、fcs_errors、および toolong_errors 以外の MAC エラーが発生した受信パケット数。
toolong_errors	最大許容長を超えていた受信パケット数。
runt_errors	最小許容長に満たなかった受信パケット数 (long)。

次の統計情報グループは、DL_TPR タイプのネットワークに適用されます。これらの統計情報は、前述のように、そのタイプのデバイス固有ドライバによって管理されます。

line_errors	非データビットまたは FCS エラーのあった受信パケット数。
burst_errors	ハーフビットタイマー 5 回の間に変位が検出されなかったことが検出された回数。
signal_losses	リングでシグナル損失状態が検出された回数。
ace_errors	A と C がともに 0 の AMP または SMP フレームのあとに、AMP フレームが介在することなく別の SMP フレームが続いた回数。
internal_errors	ステーションが内部エラーを認識した回数。
lost_frame_errors	転送中に TRR タイマーが期限切れになった回数。
frame_copied_errors	このステーション宛てのフレームが、FS フィールドの「A」ビットを 1 に設定して受信された回数。
token_errors	アクティブモニターとして動作しているステーションが、トークンの転送を必要とするエラー状態を認識した回数。
freq_errors	着信シグナルの周波数が予期された周波数と異なっていた回数。

次の統計情報グループは、DL_FDDI タイプのネットワークに適用されます。これらの統計情報は、前述のように、そのタイプのデバイス固有ドライバによって管理されます。

mac_errors	別の MAC ではエラーが検出されず、この MAC ではエラーが検出されたフレーム数。
mac_lost_errors	フレームが取り除かれるなどのフォーマットエラーが発生した受信フレーム数。

mac_tokens	制限なしトークンと制限付きトークンを合計した受信トークン数。
mac_tvx_expired	TVX が期限切れになった回数。
mac_late	この MAC のリセット後またはトークン受信後の TRT 期限切れの回数。
mac_ring_ops	リングが「Ring Not Operational」状態から「Ring Operational」状態になった回数。

GLDv2 の宣言とデータ構造体

このセクションでは、gld_mac_info(9S) および gld_stats 構造体について説明します。

gld_mac_info 構造体

GLDv2 MAC 情報 (gld_mac_info) 構造体は、デバイス固有ドライバを GLDv2 とリンクするメインデータインタフェースです。この構造体には、GLDv2 に必要なデータと、追加の省略可能なドライバ固有情報構造体へのポインタが含まれています。

gld_mac_info 構造体は gld_mac_alloc() を使用して割り当てます。構造体を解放するには gld_mac_free() を使用します。この構造体の長さは Oracle Solaris OS、GLDv2、またはその両方のリリースによって異なる可能性があるため、ドライバ側でこの構造体の長さを想定してはいけません。GLDv2 専用であり、このドキュメントに記載されていない構造体のメンバーを、デバイス固有ドライバによって設定したり読み取ったりしないでください。

gld_mac_info(9S) 構造体には次のフィールドが含まれています。

```
caddr_t             gldm_private;             /* Driver private data */
int                 (*gldm_reset)();          /* Reset device */
int                 (*gldm_start)();          /* Start device */
int                 (*gldm_stop)();           /* Stop device */
int                 (*gldm_set_mac_addr)();   /* Set device phys addr */
int                 (*gldm_set_multicast)();  /* Set/delete multicast addr */
int                 (*gldm_set_promiscuous)(); /* Set/reset promiscuous mode */
int                 (*gldm_send)();           /* Transmit routine */
uint_t              (*gldm_intr)();           /* Interrupt handler */
int                 (*gldm_get_stats)();      /* Get device statistics */
int                 (*gldm_ioctl)();          /* Driver-specific ioctls */
char                *gldm_ident;              /* Driver identity string */
uint32_t            gldm_type;                /* Device type */
uint32_t            gldm_minpkt;              /* Minimum packet size */
                                              /* accepted by driver */
uint32_t            gldm_maxpkt;              /* Maximum packet size */
                                              /* accepted by driver */
uint32_t            gldm_addrlen;             /* Physical address length */
```

```
int32_t             gldm_saplen;            /* SAP length for DL_INFO_ACK */
unsigned char       *gldm_broadcast_addr;   /* Physical broadcast addr */
unsigned char       *gldm_vendor_addr;      /* Factory MAC address */
t_uscalar_t         gldm_ppa;               /* Physical Point of */
                                            /* Attachment (PPA) number */
dev_info_t          *gldm_devinfo;          /* Pointer to device's */
                                            /* dev_info node */
ddi_iblock_cookie_t gldm_cookie;            /* Device's interrupt */
                                            /* block cookie */
```

構造体の gldm_private メンバーはデバイスドライバから認識できます。gldm_private はデバイス固有のドライバ専用でもあります。gldm_private が GLDv2 によって使用または変更されることはありません。gldm_private は従来、非公開データへのポインタとして使用され、ドライバによって定義と割り当ての両方が行われるインスタンスごとのデータ構造体を指し示します。

次のグループの構造体メンバーは、gld_register() を呼び出す前にドライバによって設定する必要があり、その後はドライバによって変更しないでください。gld_register() は構造体メンバーの値を使用またはキャッシュする可能性があるため、gld_register() の呼び出し後にドライバによって値が変更されると予期しない結果を招くことがあります。これらの構造体の詳細は、gld(9E) のマニュアルページを参照してください。

gldm_reset — ドライバのエントリポイントへのポインタ。

gldm_start — ドライバのエントリポイントへのポインタ。

gldm_stop — ドライバのエントリポイントへのポインタ。

gldm_set_mac_addr — ドライバのエントリポイントへのポインタ。

gldm_set_multicast — ドライバのエントリポイントへのポインタ。

gldm_set_promiscuous — ドライバのエントリポイントへのポインタ。

gldm_send — ドライバのエントリポイントへのポインタ。

gldm_intr — ドライバのエントリポイントへのポインタ。

gldm_get_stats — ドライバのエントリポイントへのポインタ。

gldm_ioctl — ドライバのエントリポイントへのポインタ。このポインタは null にすることができます。

gldm_ident — デバイスの簡単な説明を含む文字列へのポインタ。このポインタは、システムメッセージでデバイスを識別するために使用されます。

gldm_type — ドライバが扱うデバイスのタイプ。GLDv2 では現在、次の値がサポートされています。

- DL_ETHER (ISO 8802-3 (IEEE 802.3) および Ethernet Bus)
- DL_TPR (IEEE 802.5 Token Passing Ring)

- `DL_FDDI` (ISO 9314-2 Fibre Distributed Data Interface)

GLDv2 を正しく動作させるには、この構造体メンバーを適切に設定する必要があります。

`gldm_minpkt` — 最小の *Service Data Unit* サイズ: デバイスが転送できる最小のパケットサイズであり、MAC ヘッダーを含みません。必要なパディングをデバイス固有ドライバが処理する場合、このサイズは 0 に設定できます。

`gldm_maxpkt` — 最大の *Service Data Unit* サイズ: デバイスが転送できる最大のパケットサイズであり、MAC ヘッダーを含みません。Ethernet の場合、この数値は 1500 です。

`gldm_addrlen` — デバイスが処理する物理アドレスの長さ (バイト単位)。Ethernet、トークンリング、および FDDI の場合、この構造体メンバーの値は 6 です。

`gldm_saplen` — ドライバが使用する SAP アドレスの長さ (バイト単位)。GLDv2 ベースのドライバの場合、この長さは常に `-2` に設定されます。`-2` の長さは、2 バイトの SAP 値がサポートされること、および DLSAP アドレスで物理アドレスの後ろに SAP が入ることを示します。詳細は、DLPI 仕様の Appendix A.2「Message DL_INFO_ACK」を参照してください。

`gldm_broadcast_addr` — 転送に使用するブロードキャストアドレスが格納されたバイト配列の長さ `gldm_addrlen` へのポインタ。ドライバはブロードキャストアドレスを保持する領域を提供し、その領域に適切な値を入れ、そのアドレスを指し示すように `gldm_broadcast_addr` を設定する必要があります。Ethernet、トークンリング、および FDDI の場合、ブロードキャストアドレスは通常、`0xFF-FF-FF-FF-FF-FF` です。

`gldm_vendor_addr` — ベンダーが提供するデバイスのネットワーク物理アドレスが格納されたバイト配列の長さ `gldm_addrlen` へのポインタ。ドライバはこのアドレスを保持する領域を提供し、デバイスから読み取った情報をその領域に入れ、そのアドレスを指し示すように `gldm_vendor_addr` を設定する必要があります。

`gldm_ppa` — デバイスのこのインスタンスに対応する PPA 番号。PPA 番号は常に、`ddi_get_instance(9F)` から返されるインスタンス番号に設定されます。

`gldm_devinfo` — このデバイスに対応する `dev_info` ノードへのポインタ。

`gldm_cookie`

次のいずれかのルーチンによって返される割り込みブロック cookie。

- `ddi_get_iblock_cookie(9F)`
- `ddi_add_intr(9F)`
- `ddi_get_soft_iblock_cookie(9F)`
- `ddi_add_softintr(9F)`

この cookie は、`gld_recv()` の呼び出し元になる、デバイスの受信割り込みに対応する必要があります。

`gld_stats` 構造体

`gldm_get_stats()` を呼び出したあとに、GLDv2 ベースのドライバは `gld_stats` 構造体を使用して、統計情報および状態情報を GLDv2 に伝達します。`gld`(9E) および `gld`(7D) のマニュアルページを参照してください。この構造体のメンバーは、GLDv2 ベースのドライバによって設定され、GLDv2 が統計情報を報告するときに使用されます。次の表では、GLDv2 によって報告される統計情報変数の名前をコメントで示しています。各統計情報の意味の詳しい説明は、`gld(7D)` のマニュアルページを参照してください。

ドライバではこの構造体の長さについて何らかの想定を行ってはなりません。構造体の長さは Oracle Solaris OS、GLDv2、またはその両方のリリースによって異なる場合があります。GLDv2 専用であり、このドキュメントに記載されていない構造体のメンバーを、デバイス固有ドライバによって設定したり読み取ったりしないでください。

構造体の次のメンバーは、すべてのメディアタイプに対して定義されます。

```
uint64_t    glds_speed;                 /* ifspeed */
uint32_t    glds_media;                 /* media */
uint32_t    glds_intr;                  /* intr */
uint32_t    glds_norcvbuf;              /* norcvbuf */
uint32_t    glds_errrcv;                /* ierrors */
uint32_t    glds_errxmt;                /* oerrors */
uint32_t    glds_missed;                /* missed */
uint32_t    glds_underflow;             /* uflo */
uint32_t    glds_overflow;              /* oflo */
```

構造体の次のメンバーは、メディアタイプ `DL_ETHER` に対して定義されます。

```
uint32_t    glds_frame;                 /* align_errors */
uint32_t    glds_crc;                   /* fcs_errors */
uint32_t    glds_duplex;                /* duplex */
uint32_t    glds_nocarrier;             /* carrier_errors */
uint32_t    glds_collisions;            /* collisions */
uint32_t    glds_excoll;                /* ex_collisions */
uint32_t    glds_xmtlatecoll;           /* tx_late_collisions */
uint32_t    glds_defer;                 /* defer_xmts */
uint32_t    glds_dot3_first_coll;       /* first_collisions */
```

```
uint32_t    glds_dot3_multi_coll;         /* multi_collisions */
uint32_t    glds_dot3_sqe_error;          /* sqe_errors */
uint32_t    glds_dot3_mac_xmt_error;      /* macxmt_errors */
uint32_t    glds_dot3_mac_rcv_error;      /* macrcv_errors */
uint32_t    glds_dot3_frame_too_long;     /* toolong_errors */
uint32_t    glds_short;                   /* runt_errors */
```

構造体の次のメンバーは、メディアタイプ DL_TPR に対して定義されます。

```
uint32_t    glds_dot5_line_error          /* line_errors */
uint32_t    glds_dot5_burst_error         /* burst_errors */
uint32_t    glds_dot5_signal_loss         /* signal_losses */
uint32_t    glds_dot5_ace_error           /* ace_errors */
uint32_t    glds_dot5_internal_error      /* internal_errors */
uint32_t    glds_dot5_lost_frame_error    /* lost_frame_errors */
uint32_t    glds_dot5_frame_copied_error  /* frame_copied_errors */
uint32_t    glds_dot5_token_error         /* token_errors */
uint32_t    glds_dot5_freq_error          /* freq_errors */
```

構造体の次のメンバーは、メディアタイプ DL_FDDI に対して定義されます。

```
uint32_t    glds_fddi_mac_error;          /* mac_errors */
uint32_t    glds_fddi_mac_lost;           /* mac_lost_errors */
uint32_t    glds_fddi_mac_token;          /* mac_tokens */
uint32_t    glds_fddi_mac_tvx_expired;    /* mac_tvx_expired */
uint32_t    glds_fddi_mac_late;           /* mac_late */
uint32_t    glds_fddi_mac_ring_op;        /* mac_ring_ops */
```

前出の統計情報変数のほとんどは、特定のイベントが検出された回数を示すカウンタです。次の統計情報は回数を表しません。

glds_speed　インタフェースの現在の推定帯域幅 (ビット/秒)。帯域幅の変動のないインタフェース、または正確な推定ができないインタフェースの場合、このオブジェクトには公称の帯域幅が入ります。

glds_media　ハードウェアで使用されているメディア (配線) またはコネクタのタイプ。次のメディア名がサポートされています。

- GLDM_AUI
- GLDM_BNC
- GLDM_TP
- GLDM_10BT
- GLDM_100BT
- GLDM_100BTX
- GLDM_100BT4
- GLDM_RING4
- GLDM_RING16
- GLDM_FIBER
- GLDM_PHYMII
- GLDM_UNKNOWN

`glds_duplex` インタフェースの現在の二重状態。サポートされる値は `GLD_DUPLEX_HALF` および `GLD_DUPLEX_FULL` です。`GLD_DUPLEX_UNKNOWN` を指定することもできます。

GLDv2 関数の引数

GLDv2 ルーチンでは次の引数が使用されます。

macinfo `gld_mac_info`(9S) 構造体へのポインタ。

macaddr 有効な MAC アドレスが格納された文字配列の先頭へのポインタ。配列の長さはドライバによって、`gld_mac_info` (9S) 構造体の `gldm_addrlen` 要素で指定されます。

multicastaddr マルチキャストアドレス、グループアドレス、または機能アドレスが格納された文字配列の先頭へのポインタ。配列の長さはドライバによって、`gld_mac_info`(9S) 構造体の `gldm_addrlen` 要素で指定されます。

multiflag マルチキャストアドレスの受信を有効にするか無効にするかを示すフラグ。この引数に指定する値は `GLD_MULTI_ENABLE` または `GLD_MULTI_DISABLE` です。

promiscflag 有効にするプロミスキュアス (promiscuous) モードが存在する場合に、その種類を示すフラグ。この引数に指定する値は `GLD_MAC_PROMISC_PHYS`、`GLD_MAC_PROMISC_MULTI`、または `GLD_MAC_PROMISC_NONE` です。

mp `gld_ioctl()` は、実行する `ioctl` が格納されている STREAMS メッセージブロックへのポインタとして *mp* を使用します。`gldm_send()` は、転送するパケットが格納されている STREAMS メッセージブロックへのポインタとして *mp* を使用します。`gld_recv()` は、受信したパケットが格納されているメッセージブロックへのポインタとして *mp* を使用します。

stats 統計情報カウンタの現在値が入る `gld_stats`(9S) 構造体へのポインタ。

q `ioctl` への応答で使用される queue(9S) 構造体へのポインタ。

dip デバイスの `dev_info` 構造体へのポインタ。

name デバイスのインタフェース名。

GLDv2 のエントリポイント

エントリポイントは、GLDv2 とのインタフェースとして設計されたデバイス固有ネットワークドライバによって実装される必要があります。

gld_mac_info(9S) 構造体は、デバイス固有ドライバと GLDv2 モジュール間の通信のためのメイン構造体です。gld(7D) のマニュアルページを参照してください。この構造体の一部の要素は、ここで説明するエントリポイントへの関数ポインタです。デバイス固有ドライバは、gld_register() を呼び出す前に、その attach(9E) ルーチンでこれらの関数ポインタを初期化する必要があります。

gldm_reset() エントリポイント

int *prefix*_reset(gld_mac_info_t **macinfo*);

gldm_reset() は、ハードウェアを初期状態にリセットします。

gldm_start() エントリポイント

int *prefix*_start(gld_mac_info_t **macinfo*);

gldm_start() は、デバイスによる割り込みの生成を有効にします。また gldm_start() は、gld_recv() を呼び出して受信データパケットを GLDv2 に配信できるようにドライバを準備します。

gldm_stop() エントリポイント

int *prefix*_stop(gld_mac_info_t **macinfo*);

gldm_stop() は、デバイスによる割り込みの生成を無効にし、デバイスが gld_recv() を呼び出してデータパケットを GLDv2 に配信しないようにします。GLDv2 は、デバイスがこれ以上割り込みをかけないようにするために、gldm_stop() ルーチンを利用します。gldm_stop() による処理は失敗してはなりません。この関数は常に GLD_SUCCESS を返します。

gldm_set_mac_addr() エントリポイント

int *prefix*_set_mac_addr(gld_mac_info_t **macinfo*, unsigned char **macaddr*);

gldm_set_mac_addr() は、ハードウェアがデータの受信に使用する物理アドレスを設定します。この関数は、渡された MAC アドレス *macaddr* を介してデバイスをプログラミングできるようにします。要求にこたえるために十分なリソースが現在利用できない場合、gldm_set_mac_add() は GLD_NORESOURCES を返します。要求された関数がサポートされていない場合、gldm_set_mac_add() は GLD_NOTSUPPORTED を返します。

gldm_set_multicast() エントリポイント

```
int prefix_set_multicast(gld_mac_info_t *macinfo,
     unsigned char *multicastaddr, int multiflag);
```

gldm_set_multicast() は、特定のマルチキャストアドレスのデバイスレベルでの受信を有効または無効にします。3 番目の引数 *multiflag* を GLD_MULTI_ENABLE に設定した場合、gldm_set_multicast() はマルチキャストアドレス宛てのパケットを受信するようにインタフェースを設定します。gldm_set_multicast() は、2 番目の引数によって指示されたマルチキャストアドレスを使用します。*multiflag* を GLD_MULTI_DISABLE に設定した場合、ドライバは指定されたマルチキャストアドレスの受信を無効にできます。

この関数は、GLDv2 でマルチキャストアドレス、グループアドレス、または機能アドレスの受信を有効または無効にしようとするたびに呼び出されます。GLDv2 は、デバイスがどのような方法でマルチキャストをサポートするのか、またどのような方法でこの関数を呼び出して特定のマルチキャストアドレスを有効または無効にするのかについて、何の想定も行いません。デバイスによっては、ハッシュアルゴリズムとビットマスクを使用して、マルチキャストアドレスの集合を有効にする場合があります。この手順は認められており、GLDv2 が余分なパケットをフィルタリングして除外します。1 つのアドレスを無効にするとデバイスレベルで複数のアドレスが無効になる可能性がある場合、デバイスドライバで必要な情報を保持するようにしてください。これによって、GLDv2 が有効にしたが無効にはしていないアドレスを無効にすることを防ぎます。

gldm_set_multicast() は、すでに有効になっている特定のマルチキャストアドレスを有効にするために呼び出されることはありません。同様に、現在有効になっていないアドレスを無効にするために gldm_set_multicast () が呼び出されることもありません。GLDv2 は同じマルチキャストアドレスに対する複数の要求を追跡します。GLDv2 がドライバのエントリポイントを呼び出すのは、特定のマルチキャストアドレスの有効化を求める最初の要求、または無効化を求める最後の要求が行われたときだけです。その時点でリソースが不足していて要求を満たすことができない場合、この関数は GLD_NORESOURCES を返します。要求された関数がサポートされていない場合、この関数は GLD_NOTSUPPORTED を返します。

gldm_set_promiscuous() エントリポイント

```
int prefix_set_promiscuous(gld_mac_info_t *macinfo, int promiscflag);
```

gldm_set_promiscuous() は、プロミスキュアス (promiscuous) モードを有効または無効にします。この関数は、GLDv2 がメディア上のすべてのパケットの受信を有効または無効にしようとするたびに呼び出されます。関数の対象範囲を、メディア上のマルチキャストパケットに限定することもできます。2 番目の引数 *promiscflag* の値を GLD_MAC_PROMISC_PHYS に設定すると、この関数は物理レベルのプロミスキュアス (promiscuous) モードを有効にします。物理レベルのプロミスキュアス (promiscuous) モードが有効になると、メディア上のすべてのパケットが受信されます。*promiscflag*

を GLD_MAC_PROMISC_MULTI に設定すると、すべてのマルチキャストパケットの受信が有効になります。*promiscflag* を GLD_MAC_PROMISC_NONE に設定すると、プロミスキュアス (promiscuous) モードが無効になります。

プロミスキュアス (promiscuous) マルチキャストモードの場合、マルチキャスト限定のプロミスキュアス (promiscuous) モードを備えていないデバイス用のドライバでは、デバイスを物理プロミスキュアス (promiscuous) モードに設定する必要があります。これにより、すべてのマルチキャストパケットの受信が保証されます。この場合、ルーチンは GLD_SUCCESS を返します。GLDv2 ソフトウェアが余分なパケットをフィルタリングして除外します。その時点でリソースが不足していて要求を満たすことができない場合、この関数は GLD_NORESOURCES を返します。要求された関数がサポートされていない場合、gld_set_promiscuous() 関数は GLD_NOTSUPPORTED を返します。

上位互換性のために、gldm_set_promiscuous() ルーチンは、*promiscflag* の認識できない値をすべて GLD_MAC_PROMISC_PHYS として扱います。

gldm_send() エントリポイント

```
int prefix_send(gld_mac_info_t *macinfo, mblk_t *mp);
```

gldm_send() は、転送するパケットをデバイスのキューに入れます。このルーチンには、送信するパケットが格納された STREAMS メッセージを渡します。メッセージには複数のメッセージブロックが含まれる場合があります。send() ルーチンは、メッセージのすべてのメッセージブロックをたどって、送信パケット全体にアクセスする必要があります。長さが 0 のメッセージ継続ブロックがチェーンに含まれる場合に、そのブロックを認識してスキップするようにドライバを準備してください。またドライバでは、パケットが最大許容パケットサイズを超えないことをチェックしてください。ドライバは必要に応じて、最小許容パケットサイズにパケットをパディングする必要があります。送信ルーチンがパケットを正常に転送するか、またはキューに入れた場合、GLD_SUCCESS が返されます。

パケットの転送がただちに受け入れられなかった場合、送信ルーチンは GLD_NORESOURCES を返します。この場合、GLDv2 はあとで再試行します。gldm_send() が GLD_NORESOURCES を返した場合、あとでリソースが利用可能になった時点でドライバは gld_sched() を呼び出す必要があります。この gld_sched() の呼び出しは、ドライバが以前に転送キューに入れることができなかったパケットを再試行するよう GLDv2 に通知します (ドライバの gldm_stop() ルーチンが呼び出されても、ドライバが gldm_send() ルーチンから GLD_NORESOURCES を返すまで、ドライバはこの義務を免除されます。ただし、gld_sched() を余分に呼び出しても、誤った動作になることはありません)。

ドライバの送信ルーチンが GLD_SUCCESS を返した場合、メッセージが不要になった時点でドライバはそのメッセージを解放する必要があります。ハードウェアが DMA を使用してデータを直接読み取る場合、ドライバはハードウェアが完全にデータを読

み取るまでメッセージを解放してはなりません。この場合、ドライバは割り込みルーチンでメッセージを解放できます。ドライバは別の方法として、将来の送信操作の開始時にバッファーを再要求することもできます。送信ルーチンが GLD_SUCCESS 以外に何も返さない場合、ドライバはメッセージを解放してはなりません。ネットワークまたはリンクパートナーとの物理接続がないときに gldm_send() が呼び出された場合は、GLD_NOLINK を返します。

gldm_intr() エントリポイント

```
int prefix_intr(gld_mac_info_t *macinfo);
```

gldm_intr() は、デバイスが割り込みをかけた可能性があるときに呼び出されます。割り込みをほかのデバイスと共有している可能性があるので、ドライバはデバイスのステータスを調べ、実際に割り込みが発生したかどうかを判別する必要があります。ドライバが制御しているデバイスで割り込みが起きなかった場合、このルーチンは DDI_INTR_UNCLAIMED を返す必要があります。それ以外の場合は、割り込みを処理して DDI_INTR_CLAIMED を返す必要があります。パケットの正常な受信によって割り込みが発生した場合、このルーチンは受信パケットを M_DATA 型の STREAMS メッセージに格納し、メッセージを gld_recv() に渡します。

gld_recv() は、着信パケットをアップストリーム方向に、ネットワークプロトコルスタックの該当する次の層に渡します。gld_recv() を呼び出す前に、ルーチンは STREAMS メッセージの b_rptr および b_wptr メンバーを正しく設定する必要があります。

ドライバでは、gld_recv() の呼び出し中に mutex ロックまたはまたはその他のロックを保持しないようにしてください。特に、転送スレッドが使用する可能性のあるロックが、gld_recv() の呼び出し中に保持されていてはなりません。場合によっては、gld_recv() を呼び出す割り込みスレッドが発信パケットを送信し、結果的にドライバの gldm_send() ルーチンが呼び出されてしまうためです。gld_recv() の呼び出し時に gldm_intr() によって保持されている mutex を gldm_send() が取得しようとすると、mutex エントリの再帰性が原因でパニックが発生します。gld_recv() の呼び出しでドライバが保持する mutex をほかのドライバのエントリポイントが取得しようとすると、デッドロックに陥る可能性があります。

割り込みコードは、どのようなエラーの発生時にも統計情報カウンタを 1 増やします。このエラーには、受信データに必要なバッファーの割り当て失敗に加えて、CRC エラーやフレームエラーなどのハードウェア固有エラーが含まれています。

gldm_get_stats() エントリポイント

```
int prefix_get_stats(gld_mac_info_t *macinfo, struct gld_stats *stats);
```

gldm_get_stats() は、ハードウェア、ドライバ専用カウンタ、またはその両方から統計情報を収集し、*stats* で指し示された gld_stats (9S) 構造体を更新します。この

ルーチンは、統計情報要求を受けたときに GLDv2 によって呼び出されます。GLDv2 は統計情報要求に対する応答を作成する前に、gldm_get_stats() メカニズムを使用してデバイス依存の統計情報をドライバから取得します。定義された統計情報カウンタの詳細は、gld_stats(9S)、gld(7D)、および qreply(9F) のマニュアルページを参照してください。

gldm_ioctl() エントリポイント

```
int prefix_ioctl(gld_mac_info_t *macinfo, queue_t *q, mblk_t *mp);
```

gldm_ioctl() は、デバイス固有の ioctl コマンドを実装します。ドライバで ioctl 関数を実装しない場合、この要素は null として指定できます。ドライバはメッセージブロックを ioctl 応答メッセージに変換し、GLD_SUCCESS を返す前に qreply(9F) 関数を呼び出す必要があります。この関数は常に GLD_SUCCESS を返します。ドライバでは必要に応じて、qreply(9F) に渡すメッセージでエラーを報告してください。gldm_ioctl 要素が NULL として指定されている場合、GLDv2 は M_IOCNAK 型のメッセージを EINVAL エラーとともに返します。

GLDv2 の戻り値

GLDv2 の一部のエントリポイント関数は、これまでに説明した制限事項に従い、次の値を返す可能性があります。

GLD_BADARG	誤ったマルチキャストアドレス、誤った MAC アドレス、誤ったパケットなどの不適切な引数を関数が検出した場合
GLD_FAILURE	ハードウェア障害の場合
GLD_SUCCESS	成功した場合

GLDv2 のサービスルーチン

このセクションでは、GLDv2 のサービスルーチンの構文および説明を示します。

gld_mac_alloc() 関数

```
gld_mac_info_t *gld_mac_alloc(dev_info_t *dip);
```

gld_mac_alloc() は新しい gld_mac_info(9S) 構造体を割り当て、その構造体へのポインタを返します。構造体の GLDv2 専用要素の一部は、gld_mac_alloc() が戻る前に初期化される可能性があります。ほかのすべての要素は 0 に初期化されます。デバイスドライバは、gld_mac_info へのポインタを gld_register() に渡す前に、gld_mac_info(9S) のマニュアルページの説明に従って一部の構造体メンバーを初期化する必要があります。

gld_mac_free() 関数

```
void gld_mac_free(gld_mac_info_t *macinfo);
```

gld_mac_free() は、以前に gld_mac_alloc() によって割り当てられた gld_mac_info (9S) 構造体を解放します。

gld_register() 関数

```
int gld_register(dev_info_t *dip, char *name, gld_mac_info_t *macinfo);
```

gld_register() は、デバイスドライバの attach(9E) ルーチンから呼び出されます。gld_register() は、GLDv2 ベースのデバイスドライバを GLDv2 フレームワークとリンクします。gld_register() を呼び出す前に、デバイスドライバの attach(9E) ルーチンは gld_mac_alloc() を使用して gld_mac_info(9S) 構造体を割り当ててから、いくつかの構造体要素を初期化します。詳細は、gld_mac_info(9S) を参照してください。gld_register() の呼び出しが成功すると、次の処理が実行されます。

- デバイス固有ドライバを GLDv2 システムとリンクする
- ddi_set_driver_private(9F) を使用して、デバイス固有ドライバの非公開データポインタが macinfo 構造体を指し示すように設定する
- マイナーデバイスノードを作成する
- DDI_SUCCESS を返す

gld_register() に渡されるデバイスインタフェース名は、ファイルシステムに存在するドライバモジュールの名前と完全一致する必要があります。

gld_register() が成功した場合、ドライバの attach(9E) ルーチンは DDI_SUCCESS を返します。gld_register() が DDI_SUCCESS を返さない場合、attach(9E) ルーチンは gld_register() を呼び出す前に、割り当て済みのリソースをすべて解放してから DDI_FAILURE を返します。

gld_unregister() 関数

```
int gld_unregister(gld_mac_info_t *macinfo);
```

gld_unregister() はデバイスドライバの detach(9E) 関数によって呼び出され、成功した場合に次のタスクを実行します。

- 必要であればドライバの gldm_stop() ルーチンを呼び出して、デバイスの割り込みを確実に中止させる
- マイナーデバイスノードを削除する
- デバイス固有ドライバを GLDv2 システムからリンク解除する
- DDI_SUCCESS を返す

gld_unregister() が DDI_SUCCESS を返す場合、detach(9E) ルーチンは、attach(9E) ルーチンで割り当てられたすべてのデータ構造体を解放し (gld_mac_free() を使用して macinfo 構造体を解放し)、DDI_SUCCESS を返します。gld_unregister() が DDI_SUCCESS を返さない場合、ドライバの detach(9E) ルーチンは、デバイスを動作状態にしたまま DDI_FAILURE を返す必要があります。

gld_recv() 関数

```
void gld_recv(gld_mac_info_t *macinfo, mblk_t *mp);
```

gld_recv() はドライバの割り込みハンドラによって呼び出され、受信したパケットをアップストリームに渡します。ドライバは raw パケットが格納された M_DATA 型の STREAMS メッセージを作成して渡す必要があります。gld_recv () は、どの STREAMS キューがパケットのコピーを受信するかを調べ、必要に応じてパケットを複製します。gld_recv() はその後、必要に応じて DL_UNITDATA_IND メッセージをフォーマットし、データをすべての該当するストリームに渡します。

ドライバでは、gld_recv() の呼び出し中に mutex ロックまたはまたはその他のロックを保持しないようにしてください。特に、転送スレッドが使用する可能性のあるロックが、gld_recv() の呼び出し中に保持されていてはなりません。場合によっては、gld_recv() を呼び出す割り込みスレッドが、発信パケットの送信を含む処理を実行します。パケット転送の結果として、ドライバの gldm_send() ルーチンが呼び出されてしまいます。gld_recv() の呼び出し時に gldm_intr() によって保持されている mutex を gldm_send() が取得しようとすると、mutex エントリの再帰性が原因でパニックが発生します。gld_recv() の呼び出しでドライバが保持する mutex をほかのドライバのエントリポイントが取得しようとすると、デッドロックに陥る可能性があります。

gld_sched() 関数

```
void gld_sched(gld_mac_info_t *macinfo);
```

gld_sched() はデバイスドライバによって呼び出され、停止されていた発信パケットを再スケジューリングします。ドライバの gldm_send() ルーチンが GLD_NORESOURCES を返すたびに、ドライバは gld_sched() を呼び出して、以前に送信できなかったパケットを再試行するよう GLDv2 フレームワークに通知する必要があります。 リソースが利用可能になると、GLDv2 がドライバの gldm_send() ルーチンに発信パケットを渡す処理を再開するよう、ただちに gld_sched() が呼び出されます (ドライバの gldm_stop() ルーチンが呼び出される場合、gldm_send() から GLD_NORESOURCES が返されるまでの間、ドライバは再試行する必要はありません。ただし、gld_sched() を余分に呼び出しても、誤った動作になることはありません)。

gld_intr() 関数

```
uint_t gld_intr(caddr_t);
```

gld_intr() は GLDv2 のメイン割り込みハンドラです。gld_intr() は通常、デバイスドライバの ddi_add_intr(9F) 呼び出しで割り込みルーチンとして指定されます。割り込みハンドラへの引数は、ddi_add_intr(9F) の呼び出しで *int_handler_arg* として指定されます。この引数は gld_mac_info(9S) 構造体へのポインタである必要があります。gld_intr() は、該当する場合、デバイスドライバの gldm_intr() 関数を呼び出し、gld_mac_info(9S) 構造体を指し示すそのポインタを渡します。ただし、高レベル割り込みを使用する場合、ドライバは独自の上位割り込みハンドラを提供し、その中からソフト割り込みをトリガーする必要があります。この場合、gld_intr() は通常、ddi_add_softintr() の呼び出しでソフト割り込みハンドラとして指定されます。gld_intr() は、割り込みハンドラに適した値を返します。

◆ ◆ ◆ 第 20 章

20 USB ドライバ

この章では、Oracle Solaris 環境用の USBA 2.0 フレームワーク を使用してクライアント USB デバイスドライバを記述する方法について説明します。この章では、次の内容について説明します。

Oracle Solaris 環境での USB

Oracle Solaris USB アーキテクチャーには USBA 2.0 フレームワーク と USB クライアントドライバが含まれています。

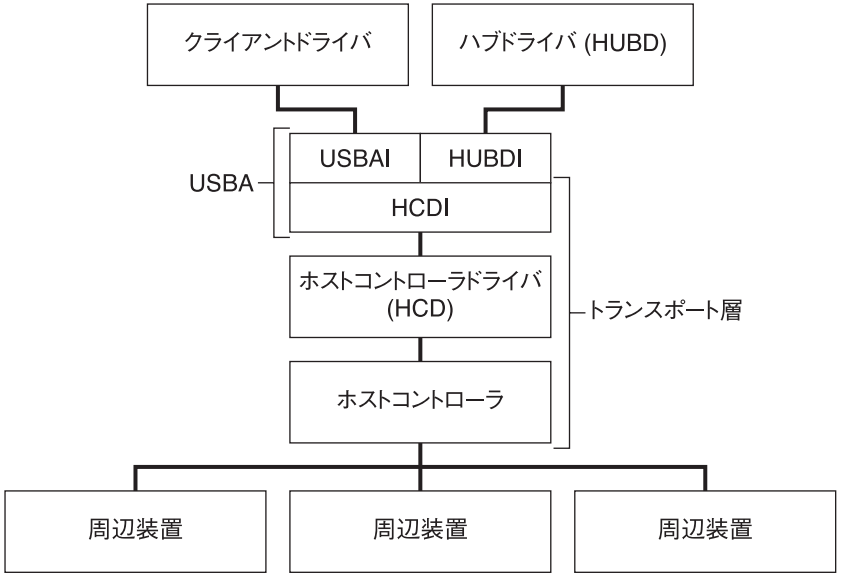
USBA 2.0 フレームワーク

USBA 2.0 フレームワーク は、USBA 準拠のクライアントドライバに対して USB デバイスの抽象ビューを提供するサービス層です。このフレームワークは、USBA 準拠のクライアントドライバが自身の USB デバイスを管理できるようにします。USBA 2.0 フレームワーク は、高速アイソクロナスパイプ以外の USB 2.0 仕様をサポートします。USB 2.0 仕様については、`http://www.usb.org/home` を参照してください。

USBA 2.0 フレームワーク はプラットフォームに依存しません。次の図に Oracle Solaris USB アーキテクチャーを示します。図の USBA 層が USBA 2.0 フレームワークです。この層は、ハードウェア固有のホストコントローラドライバとのやり取り

を、ハードウェアに依存しないホストコントローラドライバインタフェース経由で行います。ホストコントローラドライバは、管理対象のホストコントローラ経由で USB 物理デバイスにアクセスします。

図 20–1　Oracle Solaris USB アーキテクチャー



USBAI:　Solaris USB アーキテクチャーインタフェース、
USBA とクライアントドライバの間のインタフェース

HUBDI:　ハブドライバインタフェース

HCDI:　ホストコントローラドライバインタフェース

USB クライアントドライバ

USBA 2.0 フレームワークはデバイスドライバそのものではありません。この章では、図 20–1 と図 20–2 に示されているクライアントドライバについて説明します。クライアントドライバは、外部ストレージデバイスやプリンタ、ヒューマンインタフェースデバイスなど、さまざまな種類の USB デバイスと対話します。ハブドライバとは、ネクサスドライバでもあるクライアントドライバのことです。ハブドライバは、自身のポート上のデバイスを列挙し、それらのデバイスの `devinfo` ノードを作成したあと、クライアントドライバを接続します。この章では、ハブドライバの記述方法については説明しません。

USB ドライバは、その他のすべての Oracle Solaris ドライバと同じ構造を持っています。USB ドライバは、ブロックドライバ、文字ドライバ、STREAMS ドライバのいず

れかになります。USB ドライバは、呼び出し規則に従うほか、Oracle Solaris OS セクション 9 のマニュアルページで説明されているデータ構造体とルーチンを使用します。Intro(9E)、Intro(9F)、および Intro(9S) を参照してください。

USB ドライバとほかの Oracle Solaris ドライバとの違いは、USB ドライバでは、デバイスに直接アクセスする代わりに USBA 2.0 フレームワーク 関数を呼び出してデバイスにアクセスする点にあります。USBA 2.0 フレームワーク は標準の Oracle Solaris DDI ルーチンを補足します。次の図を参照してください。

図 20–2 ドライバとコントローラのインタフェース

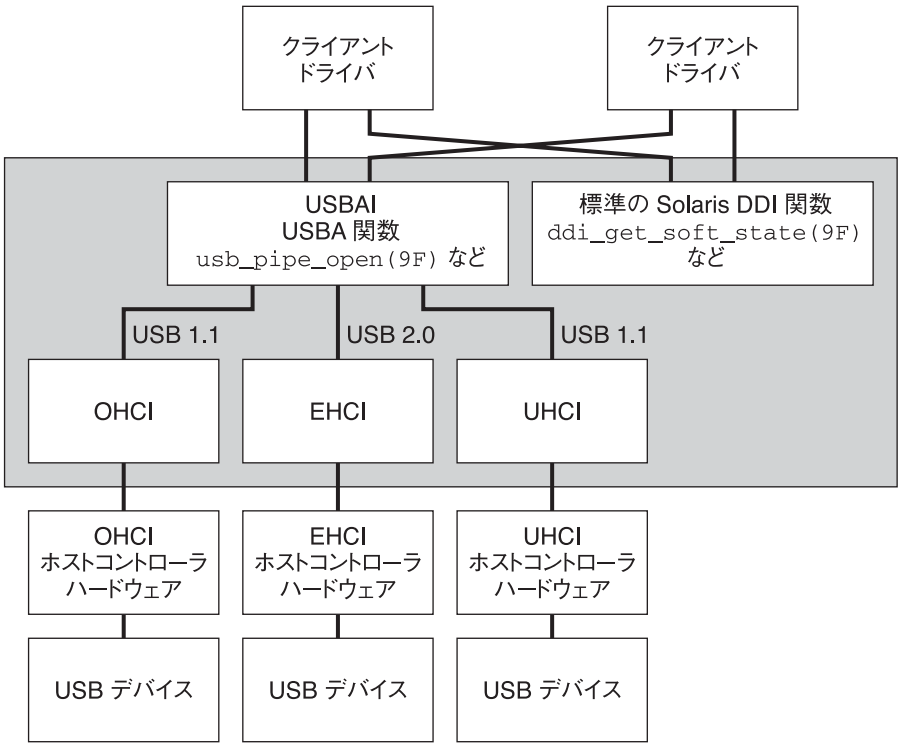


図 20–2 は、インタフェースを図 20–1 よりも詳しく示したものです。図 20–2 は、クライアントドライバから DDI 関数を呼び出せるのとまったく同様に、USBA がクライアントドライバから呼び出し可能なカーネルサブシステムであることを示しています。

必ずしもすべてのシステムが、図 20–2 に示されたすべてのホストコントローラインタフェースを持つわけではありません。OHCI (オープンホストコントローラインタフェース) ハードウェアがもっとも普及しているのは、SPARC システムと他社製 USB PCI カードです。UHCI (ユニバーサルホストコントローラインタフェース) ハードウェアがもっとも普及しているのは、x86 システムです。ただし、OHCI、UHCI のどちらのハードウェアも任意のシステム上で使用できます。EHCI (拡張ホストコントローラインタフェース) ハードウェアが存在する場合、その EHCI ハードウェアは OHCI または UHCI と同じカード上に存在し、同じポートを共有します。

ホストコントローラ、ホストコントローラドライバ、および HCDI がトランスポート層を形成し、そのトランスポート層が USBA によって制御されます。OHCI、EHCI、または UHCI への呼び出しを直接行うことはできません。これらのインタフェースへの呼び出しは、プラットフォームに依存しない USBA インタフェース経由で間接的に行います。

クライアントドライバのバインド

このセクションでは、デバイスへのドライバのバインドについて説明します。単一のインタフェースを備えたデバイスと複数のインタフェースを備えたデバイスの互換デバイス名についても説明します。

USB デバイスがシステムからどのように見えるか

USB デバイスは複数の構成をサポートできます。ある瞬間にアクティブになっている構成は、1 つだけです。そのアクティブな構成は現在の構成と呼ばれます。

構成には、複数のインタフェースを含めることができますが、1 つの機能に対して 2 つ以上のインタフェースをグループ化するインタフェース関連付けに干渉する可能性があります。構成のすべてのインタフェースが同時にアクティブになります。各インタフェースがそれぞれ異なるデバイスドライバによって操作される可能性があります。

各インタフェースは代替設定を使用することで、ホストシステムに対してさまざまな方法で自身を表現できます。ある特定のインタフェースでアクティブになる代替設定は、1 つだけです。

各代替設定は、エンドポイント経由でデバイスアクセスを提供します。各エンドポイントには特定の目的があります。ホストシステムは、エンドポイントとの通信チャネルを確立することで、デバイスと通信します。この通信チャネルはパイプと呼ばれます。

USB デバイスと Oracle Solaris デバイスツリー

構成とインタフェースをそれぞれ 1 つずつ備え、デバイスクラスがゼロであるような USB デバイスは、単一のデバイスノードとして表現されます。複数のインタフェースを備えた USB デバイスは、階層デバイス構造として表現されます。階層デバイス構造では、各インタフェースのデバイスノードは最上位デバイスノードの子になります。複数のインタフェースを備えたデバイスの一例として、オーディオ制御インタフェースとオーディオストリーミングインタフェースの両方を同時にホストコンピュータに対して提供するオーディオデバイスが挙げられます。オーディオ制御インタフェースとオーディオストリーミングインタフェースはそれぞれ独自のドライバを使って制御できます。

互換デバイス名

Oracle Solaris ソフトウェアは、各デバイス内に保持された識別情報に基づいて、USB バインド用に互換デバイス名の順序付きリストを構築します。この情報にはデバイスクラス、サブクラス、ベンダー ID、製品 ID、リビジョン、プロトコルが含まれています。USB のクラスやサブクラスの一覧については、`http://www.usb.org/home` を参照してください。

この名前の階層では、デバイス固有のドライバが使用可能でない場合には汎用ドライバへのバインドが可能です。汎用ドライバの例としては、クラス固有のドライバが挙げられます。`usbif` で始まるデバイス名は、単一インタフェースのデバイスを表します。例については、例 20–1 を参照してください。USBA 2.0 フレームワークは、あるデバイスのすべての互換名を定義します。これらのデバイス名を表示するには、例 20–2 に示すように `prtconf` コマンドを使用します。

次の例は、USB マウスデバイスの互換デバイス名の例を示したものです。このマウスデバイスは、全体が単一のドライバによって制御される複合ノードを表します。USBA 2.0 フレームワーク はこのデバイスノードに、例に示す名前を表示された順番で与えます。

例 **20–1** USB マウスの互換デバイス名

```
1. 'usb430,100.102'      Vendor 430, product 100, revision 102
2. 'usb430,100'          Vendor 430, product 100
3. 'usbif430,class3.1.2' Vendor 430, class 3, subclass 1, protocol 2
4. 'usbif430,class3.1'   Vendor 430, class 3, subclass 1
5. 'usbif430,class3'     Vendor 430, class 3
6. 'usbif,class3.1.2'    Class 3, subclass 1, protocol 2
7. 'usbif,class3.1'      Class 3, subclass 1
8. 'usbif,class3'        Class 3
```

上の例の名前は、具体的な名前から一般的な名前の順に並んでいます。エントリ 1 は、特定ベンダーの特定製品の特定リビジョンにしかバインドしません。エントリ 3、4、5 は、ベンダー 430 によって製造されたクラス 3 デバイス用です。エントリ

6、7、8 は、任意のベンダーのクラス 3 デバイス用です。バインド処理では、最上位の名前から下向きに、名前の一致が検索されます。バインドするためには、ドライバをシステムに追加する際に、これらの名前のいずれかに一致する別名を使用する必要があります。ドライバを追加するときに、バインド先となる互換デバイス名のリストを取得するには、`prtconf -vp` コマンドの出力に含まれるデバイスの `compatible` プロパティーをチェックします。

次の例は、キーボードとマウスの互換プロパティーリストを示したものです。バインドされたドライバを表示するには、`prtconf -D` コマンドを使用します。

例 20–2 構成出力コマンドによって表示された互換デバイス名

```
# prtconf -vD | grep compatible
            compatible: 'usb430,5.200' + 'usb430,5' + 'usbif430,class3.1.1'
+ 'usbif430,class3.1' + 'usbif430,class3' + 'usbif,class3.1.1' +
'usbif,class3.1' + 'usbif,class3'
            compatible: 'usb2222,2071.200' + 'usb2222,2071' +
'usbif2222,class3.1.2' + 'usbif2222,class3.1' + 'usbif2222,class3' +
'usbif,class3.1.2' + 'usbif,class3.1' + 'usbif,class3'
```

デバイスやデバイスグループのドライバをより正確に特定するには、できるだけ具体的な名前を使用してください。特定製品の特定リビジョン向けに記述されたドライバをバインドするには、一致する名前の中でもっとも具体的なものを使用します。たとえば、ベンダー 430 によって製品 100 のリビジョン 102 向けに記述された USB マウスドライバがある場合は、次のコマンドを使用してそのドライバをシステムに追加します。

```
add_drv -n -i '"usb430,100.102"' specific_mouse_driver
```

ベンダー 430 の任意の USB マウス (クラス 3、サブクラス 1、プロトコル 2) 向けに記述されたドライバを追加するには、次のコマンドを使用します。

```
add_drv -n -i '"usbif430,class3.1.2"' more_generic_mouse_driver
```

これらのドライバを両方ともインストールしたあとで互換デバイスを接続すると、接続されたデバイスにシステムによって正しいドライバがバインドされます。たとえば、これらのドライバを両方ともインストールしたあとでベンダー 430、モデル 100、リビジョン 102 のデバイスを接続すると、そのデバイスは `specific_mouse_driver` にバインドされます。ベンダー 430、モデル 98 のデバイスを接続すると、そのデバイスは `more_generic_mouse_driver` にバインドされます。別のベンダーのマウスを接続すると、そのデバイスも `more_generic_mouse_driver` にバインドされます。特定のデバイスで複数のドライバが使用可能である場合、ドライババインディングフレームワークは、互換名リスト内で先に最初に一致した互換名を持つドライバを選択します。

複数のインタフェースを備えたデバイス

複合デバイスとは、複数のインタフェースをサポートするデバイスのことです。複合デバイスには、インタフェースごとの互換名リストがあります。この互換名リストにより、使用可能なドライバの中で最適なものがインタフェースにバインドされます。もっとも汎用的なマルチインタフェースエントリは、usb,device です。

USB オーディオ複合デバイスの場合、互換名は次のようになります。

```
1. 'usb471,101.100'     Vendor 471, product 101, revision 100
2. 'usb471,101'         Vendor 471, product 101
3. 'usb,device'         Generic USB device
```

名前 usb,device は、任意の USB デバイス全体を表す互換名です。USB デバイス全体を要求したドライバがほかに存在しなければ、usb_mid(7D) ドライバ (USB マルチインタフェースドライバ) が usb,device デバイスノードにバインドします。usb_mid ドライバは、物理デバイスのインタフェースごとに子デバイスノードを 1 つずつ作成します。さらに、usb_mid ドライバは各インタフェースの一連の互換名も生成します。生成されたそれらの互換名はどれも usbif で始まります。システムはその後、生成されたそれらの互換名を使用して各インタフェースの最適なドライバを検索します。このようにして、1 つの物理デバイスの各インタフェースにそれぞれ異なるドライバをバインドすることが可能となります。

たとえば、usb_mid ドライバは、マルチインタフェースのオーディオデバイスにその usb,device ノード名を介してバインドします。続いて usb_mid ドライバは、インタフェース固有のデバイスノードを作成します。これらのインタフェース固有のデバイスノードはそれぞれ、独自の互換名リストを持ちます。オーディオ制御インタフェースノードの互換名リストは、次の例に示すようなリストになります。

例 20–3　USB オーディオ互換デバイス名

```
1. 'usbif471,101.100.config1.0' Vend 471, prod 101, rev 100, cnfg 1, iface 0
2. 'usbif471,101.config1.0'     Vend 471, product 101, config 1, interface 0
3. 'usbif471,class1.1.0'        Vend 471, class 1, subclass 1, protocol 0
4. 'usbif471,class1.1'          Vend 471, class 1, subclass 1
5. 'usbif471,class1'            Vend 471, class 1
6. 'usbif,class1.1.0'           Class 1, subclass 1, protocol 0
7. 'usbif,class1.1'             Class 1, subclass 1
8. 'usbif,class1'               Class 1
```

vendor_model_audio_usb という名前のベンダー固有、デバイス固有のクライアントドライバを、例 20–3 に示したベンダー固有、デバイス固有で構成 1、インタフェース 0 のインタフェース互換名にバインドするには、次のコマンドを使用します。

```
add_drv -n -i '"usbif471,101.config1.0"' vendor_model_audio_usb
```

audio_class_usb_if_driver という名前のクライアントドライバを、例 20–3 に示したより汎用的なクラス 1、サブクラス 1 のインタフェース互換名にバインドするには、次のコマンドを使用します。

```
add_drv -n -i '"usbif,class1.1"' audio_class_usb_if_driver
```

デバイスとそのドライバの一覧を表示するには、prtconf -D コマンドを使用します。次の prtconf -D コマンドの例から、usb_mid ドライバが audio デバイスを管理していることがわかります。usb_mid ドライバが audio デバイスをいくつかのインタフェースに分割しています。audio デバイス名の下で各インタフェースがインデントされています。prtconf -D コマンドのインデントされたリスト内のインタフェースごとに、そのインタフェースを管理するドライバが表示されています。

```
audio, instance #0 (driver name: usb_mid)
    sound-control, instance #2 (driver name: usb_ac)
    sound, instance #2 (driver name: usb_as)
    input, instance #8 (driver name: hid)
```

デバイスドライバのバインディングのチェック

ファイル /etc/driver_aliases には、すでにシステム上に存在しているバインディングのエントリが含まれています。/etc/driver_aliases ファイルの各行には、ドライバ名、空白、デバイス名がこの順番で表示されます。デバイスドライバの既存バインディングをチェックするには、このファイルを使用します。

注 – /etc/driver_aliases ファイルを手動で編集しないでください。バインディングを確立するには、add_drv(1M) コマンドを使用します。バインディングを変更するには、update_drv(1M) コマンドを使用します。

基本的なデバイスアクセス

このセクションでは、USB デバイスへのアクセス方法やクライアントドライバの登録方法について説明します。このセクションでは、記述子ツリーについても説明します。

クライアントドライバが接続される前

クライアントドライバが接続される前に、次のイベントが発生します。

1. 最初のクライアントドライバが接続される前に、PROM (OBP/BIOS) と USBA フレームワークがデバイスにアクセスします。
2. ハブドライバが、ハブの各ポート上のデバイスでアイデンティティーと構成を調べます。
3. 各デバイスへのデフォルト制御パイプが開かれ、各デバイスのデバイス記述子が調べられます。

4. デバイスごとに、そのデバイス記述子とインタフェース記述子に基づいて互換名プロパティーが構築されます。

互換名プロパティーは、クライアントドライバに個別にバインド可能なデバイスのさまざまな部分を定義します。クライアントドライバは、デバイス全体にバインドすることも、1 つのインタフェースだけにバインドすることもできます。478 ページの「クライアントドライバのバインド」を参照してください。

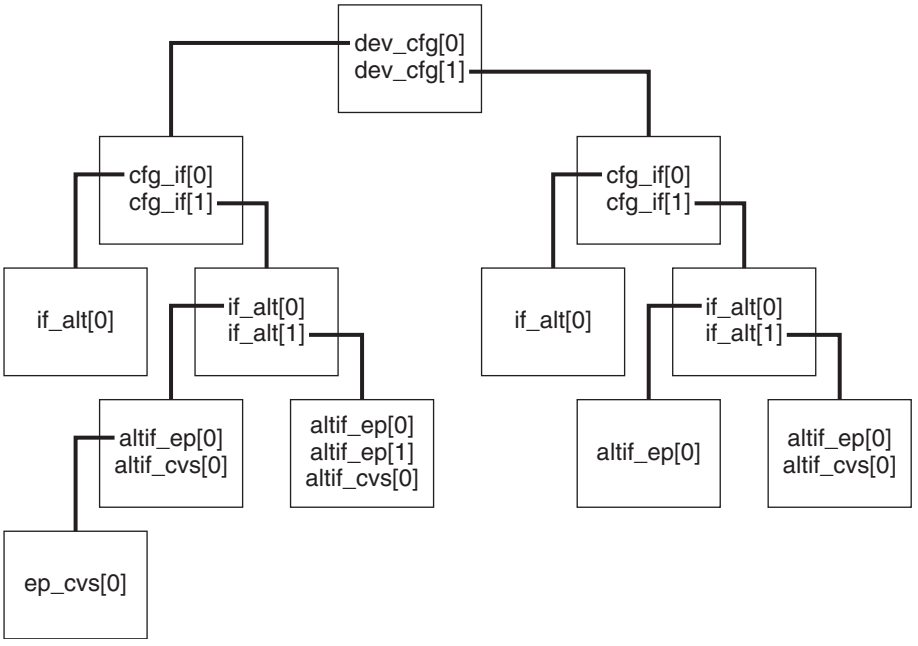
記述子ツリー

記述子の解析時には、構造体のメンバーが自然な境界に位置合わせされたあと、ホストの CPU のエンディアンに変換されます。解析後の標準 USB 構成記述子、インタフェース記述子、およびエンドポイント記述子は、各構成の階層ツリーの形でクライアントドライバから使用可能となります。クラス固有またはベンダー固有の raw 記述子情報もすべて、同じ階層ツリー内でクライアントドライバから使用可能となります。

階層記述子ツリーを取得するには、`usb_get_dev_data(9F)` 関数を呼び出します。`usb_get_dev_data` (9F) のマニュアルページの「関連項目」セクションには、各標準 USB 記述子のマニュアルページの一覧が含まれています。raw 記述子情報を解析するには、`usb_parse_data(9F)` 関数を使用します。

2 つの構成を含むデバイスの記述子ツリーは、次の図に示すようなツリーになります。

図 20–3 階層 USB 記述子ツリー



上図に示した dev_cfg 配列には、構成に対応するノードが含まれています。各ノードに含まれる情報は次のとおりです。

- 解析後の構成記述子
- その構成のインタフェースに対応する記述子の配列へのポインタ
- クラス固有またはベンダー固有の raw データが存在する場合はその配列へのポインタ

2 番目のインデックスで指定された構成の 2 番目のインタフェースを表すノードは、図で dev_cfg[1].cfg_if[1] の位置にあります。そのノードには、そのインタフェースの代替設定を表すノードの配列が含まれています。USB 記述子の階層はツリーの全体にわたって伝播されます。文字列記述子データに含まれる ASCII 文字列は、USB 仕様でこれらの文字列が存在するとされている場所に接続されます。

構成の配列は間が空いておらず、構成インデックスでインデックス指定されます。最初の有効な構成 (構成 1) は、dev_cfg[0] です。インタフェースと代替設定のインデックスは、それらの個数に対応した値になります。各代替設定のエンドポイントは連続的にインデックス指定されます。各代替設定の最初のエンドポイントはインデックス 0 です。

この採番方式によりツリーのトラバースが容易に行えます。たとえば、エンドポイントインデックス 0、代替 0、インタフェース 1、構成インデックス 1 の raw 記述子データは、次のパスで定義されるノードの位置にあります。

```
dev_cfg[1].cfg_if[1].if_alt[0].altif_ep[0].ep_descr
```

記述子ツリーを直接使用する代わりに usb_lookup_ep_data(9F) 関数を使用することもできます。usb_lookup_ep_data(9F) 関数は、引数としてインタフェース、代替設定、エンドポイント、エンドポイントタイプ、および向きを取ります。usb_lookup_ep_data(9F) 関数を使用すると、記述子ツリーをたどって特定のエンドポイントを取得できます。詳細については、usb_get_dev_data(9F) のマニュアルページを参照してください。

デバイスアクセスを取得するためのドライバの登録

クライアントドライバからの USBA 2.0 フレームワーク への最初の 2 つの呼び出しは、usb_client_attach(9F) 関数と usb_get_dev_data(9F) 関数の呼び出しです。これら 2 つの呼び出しは、クライアントドライバの attach(9E) エントリポイントから行われます。usb_get_dev_data(9F) 関数を呼び出す前に usb_client_attach(9F) 関数を呼び出す必要があります。

usb_client_attach(9F) 関数は、USBA 2.0 フレームワーク にクライアントドライバを登録します。usb_client_attach(9F) 関数ではバージョン管理が実行されます。クライアントドライバのソースファイルは必ず、次の行で始まっている必要があります。

```
#define USBDRV_MAJOR_VER        2
#define USBDRV_MINOR_VER        minor-version
#include <sys/usb/usba.h>
```

minor-version の値は USBA_MINOR_VER 以下である必要があります。シンボル USBA_MINOR_VER は <sys/usb/usbai.h> ヘッダーファイル内で定義されています。<sys/usb/usbai.h> ヘッダーファイルは <sys/usb/usba.h> ヘッダーファイルによってインクルードされます。

USBDRV_VERSION は、USBDRV_MAJOR_VERSION と USBDRV_MINOR_VERSION からバージョン番号を生成するマクロです。usb_client_attach() の第 2 引数は USBDRV_VERSION である必要があります。この第 2 引数が USBDRV_VERSION でない場合や USBDRV_VERSION が無効なバージョンを反映している場合には、usb_client_attach() 関数が失敗します。この制限により、プログラミングインタフェースの互換性が確保されます。

usb_get_dev_data() 関数は、適切な USB デバイス管理に必要な情報を返します。たとえば、usb_get_dev_data() 関数から次の情報が返されます。

- デフォルト制御パイプ
- mutex の初期化に使用する *iblock_cookie* (mutex_init(9F) を参照)
- 解析後のデバイス記述子
- ID 文字列
- 483 ページの「記述子ツリー」で説明したツリー階層

usb_get_dev_data() 関数の呼び出しは必須です。usb_get_dev_data() を呼び出すことが、デフォルト制御パイプを取得したり mutex の初期化に必要な *iblock_cookie* を取得したりするための唯一の方法です。

クライアントドライバの attach(9E) ルーチンは通常、usb_get_dev_data() を呼び出したあとで、必要な記述子やデータを記述子ツリーからドライバのソフト状態にコピーします。ソフト状態にコピーされたエンドポイント記述子は、あとでそれらのエンドポイントへのパイプを開く際に使用されます。attach(9E) ルーチンは通常、記述子をコピーしたあとで、usb_free_descr_tree(9F) を呼び出して記述子ツリーを解放します。あるいは記述子ツリーを保持し、記述子をコピーしないようにしてもかまいません。

次の 3 つの解析レベルのいずれかを usb_get_dev_data(9F) 関数に指定することで、返される記述子ツリーの範囲を決定します。ドライバがデバイスのより広い範囲にバインドする必要がある場合、より広い範囲のツリーが必要となります。

- USB_PARSE_LVL_IF: クライアントドライバが特定のインタフェースにバインドする場合、ドライバが必要とするのは、そのインタフェースの記述子だけです。それらの記述子のみを取得するには、usb_get_dev_data() の呼び出し時に解析レベルとして USB_PARSE_LVL_IF を指定します。
- USB_PARSE_LVL_CFG: クライアントドライバがデバイス全体にバインドする場合は、USB_PARSE_LVL_CFG を指定して現在の構成のすべての記述子を取得します。
- USB_PARSE_LVL_ALL: すべての構成のすべての記述子を取得するには、USB_PARSE_LVL_ALL を指定します。たとえば、usb_print_descr_tree(9F) を使用してあるデバイスのすべての構成の記述子ダンプを出力するには、ツリーの範囲をこの最高レベルにする必要があります。

クライアントドライバの detach(9E) ルーチンは、usb_free_dev_data(9F) 関数を呼び出すことで、usb_get_dev_data() 関数によって割り当てられたリソースをすべて解放する必要があります。usb_free_dev_data() 関数はハンドルを受け取りますが、そのハンドルでは、記述子ツリーがすでに usb_free_descr_tree() 関数で解放されています。さらにクライアントドライバの detach() ルーチンは、usb_client_detach(9F) 関数も呼び出すことで、usb_client_attach(9F) 関数によって割り当てられたリソースをすべて解放する必要があります。

デバイス通信

USB デバイスは、パイプと呼ばれる通信チャネル経由で要求を渡すことにより動作します。要求を送信するには、まずパイプをオープンする必要があります。パイプのフラッシュ、クエリー、およびクローズも行えます。このセクションでは、パイプ、データ転送とコールバック、およびデータ要求について説明します。

USB エンドポイント

4 種類の USB エンドポイントと通信を行う 4 種類のパイプを次に示します。

- 制御: 制御パイプは主に、コマンドを送信してステータスを取得するために使用されます。制御パイプは、小さいサイズの構造化データのホスト起動要求/応答通信を非定期的に行うためのものです。制御パイプは双方向です。デフォルトパイプは制御パイプです。487 ページの「デフォルトパイプ」を参照してください。
- 一括: 一括パイプは主にデータ転送に使用されます。一括パイプにより、大量のデータを確実に転送できます。一括パイプによるデータ配信は、適時に行われるとはかぎりません。一括パイプは単方向です。
- 割り込み: 割り込みパイプにより、少量の非構造化データの通信を適時かつ確実に行うことができます。定期的なポーリングは通常、割り込み入力パイプ上で開始されます。割り込み入力パイプは、デバイス上にデータが用意された時点でそのデータをホストに返します。一部のデバイスでは割り込み出力パイプも使用できます。割り込み出力パイプは割り込み入力パイプと同じく、適時性があり確実という「割り込みパイプ」の特性に基づいて、デバイスへのデータ転送を行います。割り込みパイプは単方向です。
- アイソクロナス: アイソクロナスパイプは、オーディオデバイスの場合のように、一定速度の時間重視のデータを転送するためのチャネルを提供します。エラーが発生してもデータの再試行は行われません。アイソクロナスパイプは単方向です。

これらのエンドポイントに対応する転送タイプの詳細については、USB 2.0 仕様書の第 5 章を参照するか、490 ページの「要求」を参照してください。

デフォルトパイプ

各 USB デバイスは、デフォルトエンドポイントと呼ばれる特殊な制御エンドポイントを持ちます。その通信チャネルはデフォルトパイプと呼ばれます。すべてではないにしろ、ほとんどのデバイス設定がこのパイプ経由で行われます。多くの USB デバイスは、自身の唯一の制御パイプとしてこのパイプを持ちます。

`usb_get_dev_data(9F)` 関数は、デフォルト制御パイプをクライアントドライバに提供します。このパイプは、ほかのパイプを開く前に必要とされるどのような特殊な設定にも対応できるよう、事前に開かれています。このデフォルト制御パイプは次の点で特殊です。

- このパイプは共有されます。同じデバイスのほかのインタフェースを操作しているドライバも、同じデフォルト制御パイプを使用します。USBA 2.0 フレームワークはこのパイプを複数のドライバ間で調停します。
- このパイプのオープン、クローズ、またはリセットをクライアントドライバから行うことはできません。この制限が設けられているのは、このパイプが共有されるからです。

- このパイプは例外発生時に自動的にクリアされます。

その他の制御パイプを含むほかのパイプは、明示的に開く必要があり、排他的にしか開くことができません。

パイプの状態

パイプの状態は次のいずれかになります。

- USB_PIPE_STATE_IDLE
  - すべての制御パイプと一括パイプ、割り込み出力パイプ、およびアイソクロナス出力パイプ: 進行中の要求がありません。
  - 割り込み入力パイプとアイソクロナス入力パイプ: 進行中のポーリングがありません。
- USB_PIPE_STATE_ACTIVE
  - すべての制御パイプと一括パイプ、割り込み出力パイプ、およびアイソクロナス出力パイプ: パイプがデータを転送しているか、入出力要求がアクティブになっています。
  - 割り込み入力パイプとアイソクロナス入力パイプ: ポーリングがアクティブになっています。
- USB_PIPE_STATE_ERROR: エラーが発生しました。このパイプがデフォルトパイプではなく、かつ自動クリアが有効になっていない場合、クライアントドライバは usb_pipe_reset(9F) 関数を呼び出す必要があります。
- USB_PIPE_STATE_CLOSING: パイプが閉じられようとしています。
- USB_PIPE_STATE_CLOSED: パイプが閉じられました。

パイプの状態を取得するには、usb_pipe_get_state(9F) 関数を呼び出します。

パイプのオープン

パイプを開くには、その開くパイプに対応するエンドポイント記述子を usb_pipe_open(9F) 関数に渡します。記述子ツリーからエンドポイント記述子を取得するには、usb_get_dev_data(9F) および usb_lookup_ep_data(9F) 関数を使用します。usb_pipe_open(9F) 関数はパイプへのハンドルを返します。

パイプを開くときにパイプポリシーを指定する必要があります。パイプポリシーには、このパイプで必要になると予想される、独立したスレッドを必要とする同時非同期処理の数が含まれています。予想されるスレッド数は、コールバック中に発生する可能性のある並列処理の数です。この予想値は 2 以上である必要があります。パイプポリシーの詳細については、usb_pipe_open(9F) のマニュアルページを参照してください。

パイプのクローズ

ドライバは、usb_pipe_close(9F) 関数を使用してデフォルトパイプ以外のパイプを閉じる必要があります。usb_pipe_close(9F) 関数は、パイプ内の残りの要求がすべて完了するまで待ちます。次にこの関数は、それらの要求のすべてのコールバックが完了するまで 1 秒間待ちます。

データ転送

どのパイプタイプの場合も、プログラミングモデルは次のようになります。

1. 要求を割り当てます。
2. いずれかのパイプ転送関数を使用して要求を送信します。usb_pipe_bulk_xfer(9F)、usb_pipe_ctrl_xfer(9F)、usb_pipe_intr_xfer(9F)、および usb_pipe_isoc_xfer(9F) のマニュアルページを参照してください。
3. 完了通知が届くまで待ちます。
4. 要求を解放します。

要求の詳細については、490 ページの「要求」を参照してください。次の各セクションでは、さまざまな要求タイプの機能について説明します。

同期および非同期転送とコールバック

転送は同期型と非同期型のいずれかになります。同期転送は、転送が完了するまでブロックします。非同期転送では、転送完了時にクライアントドライバへのコールバックが発生します。*flags* 引数に USB_FLAGS_SLEEP フラグを設定して転送関数を呼び出した場合、その大部分は同期型になります。

ポーリングやアイソクロナス転送といった継続的な転送を同期型にすることはできません。継続的な転送向けの転送関数を呼び出すときに USB_FLAGS_SLEEP フラグを設定した場合、転送が始まる前に、リソースに対して待機するためだけに呼び出しがブロックされます。

同期転送は、設定がもっとも単純な転送です。これは、同期転送ではコールバック関数が一切必要ないからです。同期転送関数は、転送が完了するまでブロックするにもかかわらず、転送開始ステータスを返します。転送が完了すると、要求の完了理由フィールドとコールバックフラグフィールドに、転送ステータスに関する追加情報が表示されます。完了理由フィールドとコールバックフラグフィールドについては後述します。

flags 引数に USB_FLAGS_SLEEP フラグを指定しなかった場合、その転送処理は非同期型になります。この規則の例外は、アイソクロナス転送です。非同期転送処理は、転送を設定して開始したあと、転送が完了する前に戻ります。非同期転送処理は転送開始ステータスを返します。クライアントドライバは、コールバックハンドラ経由で転送完了ステータスを受け取ります。

コールバックハンドラとは、非同期転送の完了時に呼び出される関数のことです。非同期転送を設定する際には必ずコールバックを指定してください。コールバックハンドラには、正常終了ハンドラと例外ハンドラの 2 種類があります。このどちらの場合にも同じハンドラが呼び出されるように指定することもできます。

- 正常終了: 正常終了コールバックハンドラは、転送が正常終了したことを通知するために呼び出されます。
- 例外: 例外コールバックハンドラは、転送が異常終了したことを通知し、エラーを処理するために呼び出されます。

完了ハンドラと例外ハンドラはどちらも、転送の要求を引数として受け取ります。例外ハンドラは、要求内の完了理由とコールバックステータスに基づいて何が起こったのかを調べます。完了理由 (`usb_cr_t`) は、元のトランザクションがどのように完了したのかを示します。たとえば、完了理由 `USB_CR_TIMEOUT` は、転送がタイムアウトしたことを示します。別の例として、USB デバイスが使用中に削除された場合、クライアントドライバは自身の未処理の要求で、`USB_CR_DEV_NOT_RESP` を完了理由として受け取ります。コールバックステータス (`usb_cb_flags_t`) は、状況に対処するために USBA フレームワークが何を行ったのかを示します。たとえば、コールバックステータス `USB_CB_STALL_CLEARED` は、USBA フレームワークが機能ストールステータスをクリアしたことを示します。完了理由の詳細については、`usb_completion_reason(9S)` のマニュアルページを参照してください。コールバックステータスフラグの詳細については、`usb_callback_flags(9S)` のマニュアルページを参照してください。

コールバックのコンテキストと要求が実行されるパイプのポリシーによって、コールバック内で行える処理が制限されます。

- コールバックコンテキスト: 大部分のコールバックはカーネルコンテキストで実行され、通常はブロック可能です。一部のコールバックは割り込みコンテキストで実行され、ブロックできません。割り込みコンテキストであることを示すには、コールバックフラグに `USB_CB_INTR_CONTEXT` フラグを設定します。コールバックコンテキストの詳細やブロックの詳細については、`usb_callback_flags(9S)` のマニュアルページを参照してください。
- パイプポリシー: 同時非同期処理に関するパイプポリシーのヒントにより、並列実行可能な処理 (コールバックハンドラから実行される処理を含む) の数が制限されます。同期処理でのブロックは、1 つの処理としてカウントされます。パイプポリシーの詳細については、`usb_pipe_open(9F)` のマニュアルページを参照してください。

要求

このセクションでは要求構造体、および各種の要求の割り当てと割り当て解除について説明します。

要求の割り当てと割り当て解除

要求は、初期化された要求構造体として実装されます。各エンドポイントタイプはそれぞれ異なるタイプの要求を受け取ります。要求タイプごとに異なる要求構造体型が用意されています。次の表に、各要求タイプの構造体型を示します。この表では、各構造体型の割り当てや解放に使用する関数の一覧も示しています。

表 20–1 要求の初期化

パイプまたはエンドポイントのタイプ	要求構造体	要求構造体割り当て関数	要求構造体解放関数
制御	usb_ctrl_req_t (usb_ctrl_request(9S) のマニュアルページを参照)	usb_alloc_ctrl_req(9F)	usb_free_ctrl_req(9F)
一括	usb_bulk_req_t (usb_bulk_request(9S) のマニュアルページを参照)	usb_alloc_bulk_req(9F)	usb_free_bulk_req(9F)
割り込み	usb_intr_req_t (usb_intr_request(9S) のマニュアルページを参照)	usb_alloc_intr_req(9F)	usb_free_intr_req(9F)
アイソクロナス	usb_isoc_req_t (usb_isoc_request(9S) のマニュアルページを参照)	usb_alloc_isoc_req(9F)	usb_free_isoc_req(9F)

次の表は、各要求タイプで使用可能な転送関数の一覧です。

表 20–2 要求転送の設定

パイプまたはエンドポイントのタイプ	転送関数
制御	usb_pipe_ctrl_xfer(9F)、usb_pipe_ctrl_xfer_wait(9F)
一括	usb_pipe_bulk_xfer(9F)
割り込み	usb_pipe_intr_xfer(9F)、usb_pipe_stop_intr_polling(9F)
アイソクロナス	usb_pipe_isoc_xfer(9F)、usb_pipe_stop_isoc_polling(9F)

要求の割り当てと割り当て解除を行う際には、次の手順に従います。

1. 適切な割り当て関数を使用して、必要な要求タイプの要求構造体を割り当てます。要求構造体割り当て関数のマニュアルページについては、表 20–1 を参照してください。

2. 構造体内で必要なフィールドをすべて初期化します。詳細については、492 ページの「要求の機能とフィールド」または対応する要求構造体のマニュアルページを参照してください。要求構造体のマニュアルページについては、表 20–1 を参照してください。
3. データ転送が完了したら、適切な解放関数を使用して要求構造体を解放します。要求構造体解放関数のマニュアルページについては、表 20–1 を参照してください。

要求の機能とフィールド

ドライバが STREAMS 型、文字型、ブロック型のいずれであっても、データが画一的に処理されるように、すべての要求のデータがメッセージブロックとして渡されます。メッセージブロック型 `mblk_t` については、`mblk(9S)` のマニュアルページを参照してください。DDI には、メッセージブロックを操作するためのルーチンがいくつか用意されています。たとえば、`allocb(9F)` や `freemsg(9F)` などがあります。メッセージブロックを操作するためのその他のルーチンについて学ぶには、`allocb(9F)` および `freemsg(9F)` のマニュアルページの「関連項目」セクションを参照してください。また、『STREAMS Programming Guide』も参照してください。

すべての転送タイプに含まれる要求フィールドを次に示します。各フィールド名の *xxxx* は、*ctrl*、*bulk*、*intr*、*isoc* のいずれかの値になります。

xxxx_client_private	このフィールドの値は、クライアントドライバで要求と一緒に渡される内部データ用のポインタになります。デバイスにデータを転送するためにこのポインタが使用されることはありません。
xxxx_attributes	このフィールドの値は、一連の転送属性になります。このフィールドはすべての要求構造体に共通しますが、その初期化方法は転送タイプごとに若干異なります。詳細については、対応する要求構造体のマニュアルページを参照してください。これらのマニュアルページについては、表 20–1 を参照してください。また、`usb_request_attributes(9S)` のマニュアルページも参照してください。
xxxx_cb	このフィールドの値は、転送が正常終了した場合のコールバック関数になります。この関数は、非同期転送がエラーなしで完了した場合に呼び出されます。
xxxx_exc_cb	このフィールドの値は、エラー処理用のコールバック関数になります。この関数が呼び出されるのは、非同期転送でエラーが発生した場合だけです。

xxxx_completion_reason	このフィールドには転送自体の完了ステータスが保持されます。エラーが発生した場合にこのフィールドを見れば、どのような問題が発生したのかがわかります。詳細については、usb_completion_reason(9S) のマニュアルページを参照してください。このフィールドは USBA 2.0 フレームワーク によって更新されます。
xxxx_cb_flags	このフィールドには、USBA 2.0 フレームワーク がコールバックハンドラを呼び出す前に行った復旧処理のリストが含まれています。USB_CB_INTR_CONTEXT フラグは、コールバックが割り込みコンテキストで実行されているかどうかを示します。詳細については、usb_callback_flags(9S) のマニュアルページを参照してください。このフィールドは USBA 2.0 フレームワーク によって更新されます。

次の各セクションでは、4 つの転送タイプでそれぞれ異なる要求フィールドについて説明します。これらのセクションでは、それらの構造体フィールドを初期化する方法について説明します。また、属性やパラメータのさまざまな組み合わせに関する制限についてもこれらのセクションで説明します。

制御要求

制御パイプ経由でメッセージ転送を開始するには、制御要求を使用します。後述するように、転送は手動で設定できます。また、usb_pipe_ctrl_xfer_wait(9F) ラッパー関数を使用して同期転送を設定および送信することもできます。

クライアントドライバは、USB 2.0 仕様書に記載されているように、*ctrl_bmRequestType*、*ctrl_bRequest*、*ctrl_wValue*、*ctrl_wIndex*、*ctrl_wLength* の各フィールドを初期化する必要があります。

要求の *ctrl_data* フィールドは、データバッファーを指すように初期化する必要があります。usb_alloc_ctrl_req(9F) 関数は、バッファーの *len* に正の値が渡された場合、このフィールドを初期化します。もちろん、アウトバウンド転送ではこのバッファーを必ず初期化する必要があります。いずれにしても、クライアントドライバは、転送完了時に要求を解放する必要があります。

複数の制御要求をキューに登録できます。キューに登録する要求として、同期要求と非同期要求を組み合わせてもかまいません。

ctrl_timeout フィールドは、要求が処理されまでの最大待機時間を定義します。ただし、キューでの待機時間は含めません。このフィールドは同期要求と非同期要求の両方に適用されます。 *ctrl_timeout* フィールドは秒単位で指定します。

ctrl_exc_cb フィールドには、例外の発生時に呼び出す関数のアドレスを指定します。この例外ハンドラの引数については、usb_ctrl_request(9S) のマニュアルページを参照してください。例外ハンドラの第 2 引数は、usb_ctrl_req_t 構造体です。要求

構造体を引数として渡せば、例外ハンドラから要求の *ctrl_completion_reason* および *ctrl_cb_flags* フィールドをチェックし、最適な復旧処理を決定することが可能となります。

USB_ATTRS_ONE_XFER および USB_ATTRS_ISOC_* フラグは、すべての制御要求で無効な属性です。USB_ATTRS_SHORT_XFER_OK フラグは、ホスト宛ての要求でのみ有効です。

一括要求

処理時間の要件が高くないデータを送信するには、一括要求を使用します。バス全体の負荷によっては、一括要求が完了するまでに数 USB フレームかかる可能性があります。

すべての要求は、初期化済みのメッセージブロックを受け取る必要があります。mblk_t メッセージブロック型の説明については、mblk(9S) のマニュアルページを参照してください。このメッセージブロックは、転送の向きに応じてデータ提供とデータ格納のいずれかを行います。詳細については、usb_bulk_request(9S) のマニュアルページを参照してください。

USB_ATTRS_ONE_XFER および USB_ATTRS_ISOC_* フラグは、すべての一括要求で無効な属性です。USB_ATTRS_SHORT_XFER_OK フラグは、ホスト宛ての要求でのみ有効です。

usb_pipe_get_max_bulk_transfer_size(9F) 関数は、要求当たりの最大バイト数を指定します。取得した値は、クライアントドライバの minphys(9F) ルーチンで使用される最大値として指定できます。

複数の一括要求をキューに登録できます。

割り込み要求

割り込み要求は通常、定期的なインバウンドデータ用です。割り込み要求は、定期的にデバイスにポーリングしてデータの有無を確認します。ただし、USBA 2.0 フレームワーク では、1 回かぎりのインバウンド割り込みデータ要求やアウトバウンド割り込みデータ要求がサポートされています。USB 割り込み転送機能の適時性と再試行は、どの割り込み要求でも利用できます。

USB_ATTRS_ISOC_* フラグは、すべての割り込み要求で無効な属性です。USB_ATTRS_SHORT_XFER_OK および USB_ATTRS_ONE_XFER フラグは、ホスト宛ての要求でのみ有効です。

1 回かぎりのポーリングは必ず同期割り込み転送として実行されます。要求で USB_ATTRS_ONE_XFER 属性を指定すると、1 回かぎりのポーリングになります。

定期的なポーリングは、非同期割り込み転送として開始されます。元の割り込み要求を usb_pipe_intr_xfer(9F) に渡します。ポーリング中に返す新しいデータが見つかると、新しい usb_intr_req_t 構造体が元の要求から複製され、初期化済みのデータ

ブロックが設定されます。要求を割り当てる際には、usb_alloc_intr_req(9F) 関数の *len* 引数にゼロを指定します。*len* 引数をゼロにするのは、USBA 2.0 フレームワークがコールバックごとに新しい要求の割り当てやデータ設定を行うからです。要求構造体の割り当て後、その *intr_len* フィールドを設定することで、ポーリングごとにフレームワークが割り当てるバイト数を指定します。*intr_len* バイトを超えるデータは返されません。

クライアントドライバは、受け取った各要求を解放する必要があります。メッセージブロックをアップストリームに送信する場合、送信を行う前にメッセージブロックを要求から切り離します。メッセージブロックを要求から切り離すには、要求のデータポインタを NULL に設定します。要求のデータポインタを NULL に設定すると、要求が割り当て解除されるときに、メッセージブロックが解放されるのを防ぐことができます。

定期的なポーリングを取り消すには、usb_pipe_stop_intr_polling(9F) 関数を呼び出します。ポーリングを停止したりパイプを閉じたりすると、例外コールバック経由で元の要求構造体が返されます。この返される要求構造体の完了理由には、USB_CR_STOPPED_POLLING が設定されます。

ポーリングがすでに進行中である場合にポーリングを開始しないでください。usb_pipe_stop_intr_polling(9F) の呼び出しの進行中にポーリングを開始しないでください。

アイソクロナス要求

アイソクロナス要求は、一定速度の時間重視のストリーミングデータ用です。エラー時の再試行は行われません。アイソクロナス要求には次の要求固有フィールドが含まれています。

isoc_frame_no

転送全体を特定のフレーム番号から始める必要がある場合に、このフィールドを指定します。このフィールドの値は、現在のフレーム番号よりも大きい必要があります。現在のフレーム番号を取得するには、usb_get_current_frame_number(9F) を使用します。現在のフレーム番号は刻々変化します。低速バスやフルスピードバスでは、現在のフレームは 1 ミリ秒ごとに更新されます。高速バスでは、現在のフレームは 0.125 ミリ秒ごとに更新されます。*isoc_frame_no* フィールドが認識されるように、USB_ATTR_ISOC_START_FRAME 属性を設定します。

このフレーム番号フィールドを無視してできるだけすぐに開始するには、USB_ATTR_ISOC_XFER_ASAP フラグを設定します。

isoc_pkts_count

このフィールドは、要求内のパケット数です。この値は、usb_get_max_pkts_per_isoc_request(9F) 関数から返される値と *isoc_pkt_descr* 配列 (後述の説明を参照) のサイズによって制限されます。この要求で転送可能なバイト数は、この *isoc_pkts_count* 値とエンドポイントの *wMaxPacketSize* 値の積に等しくなります。

isoc_pkts_length	このフィールドは、要求のすべてのパケットの長さを合計したものになります。この値はイニシエータによって設定されます。この値はゼロに設定して、*isoc_pkt_descr* リストの *isoc_pkts_length* の合計が自動的に使用され、この要素にチェックが適用されないようにする必要があります。
isoc_error_count	このフィールドは、エラーが発生して完了したパケットの数です。この値は USBA 2.0 フレームワーク によって設定されます。
isoc_pkt_descr	このフィールドは、パケットごとに転送するデータ量を定義したパケット記述子の配列を指します。送信要求の場合、この値は処理対象のサブ要求の非公開キューを定義します。受信要求の場合、この値は、データがどのように分割されて到着するかを記述します。送信要求の場合、クライアントドライバがこれらの記述子を割り当てます。受信要求の場合、フレームワークがこれらの記述子の割り当てと初期化を行います。この配列の各記述子にはフレームワークによって初期化されたフィールドが含まれており、それらのフィールドには、実際に転送されたバイト数と転送ステータスが保持されています。詳細については、`usb_isoc_request(9S)` のマニュアルページを参照してください。

すべての要求は、初期化済みのメッセージブロックを受け取る必要があります。このメッセージブロックはデータ提供とデータ格納のいずれかを行います。`mblk_t` メッセージブロック型の説明については、`mblk(9S)` のマニュアルページを参照してください。

`USB_ATTR_ONE_XFER` フラグは不正な属性です。システムが使用可能なパケット経由でデータ量をどのように変化させるかを決定するからです。`USB_ATTR_SHORT_XFER_OK` フラグは、ホスト宛てのデータでのみ有効です。

`usb_pipe_isoc_xfer(9F)` 関数は、`USB_FLAGS_SLEEP` フラグの設定の有無にかかわらず、すべてのアイソクロナス転送を非同期にします。アイソクロナス入力要求では必ずポーリングが開始されます。

定期的なポーリングを取り消すには、`usb_pipe_stop_isoc_polling(9F)` 関数を呼び出します。ポーリングを停止したりパイプを閉じたりすると、例外コールバック経由で元の要求構造体が返されます。この返される要求構造体の完了理由には、`USB_CR_STOPPED_POLLING` が設定されます。

ポーリングは、次のいずれかのイベントが発生するまで継続されます。

- `usb_pipe_stop_isoc_polling(9F)` 呼び出しが受信された。
- 例外コールバック経由でデバイスの切り離しが報告された。
- `usb_pipe_close(9F)` 呼び出しが受信された。

パイプのフラッシュ

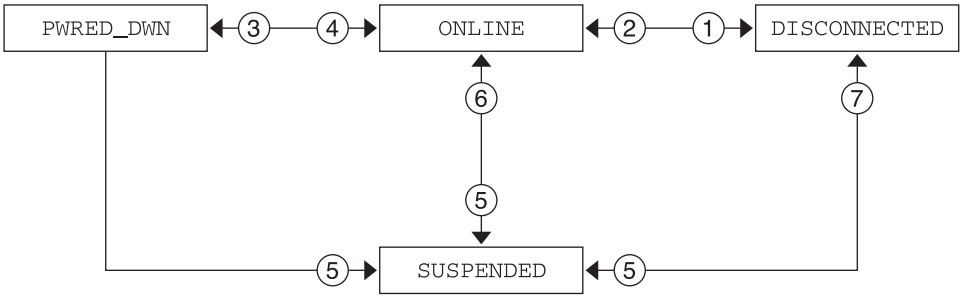
エラー発生後にパイプをクリーンアップする必要に迫られたり、パイプが空になるまで待機したりする可能性があります。パイプのフラッシュやクリアを行うには、次のいずれかの方法を使用します。

- usb_pipe_reset(9F) 関数はパイプをリセットし、すべての要求をフラッシュします。これは、自動クリアが有効になっていないパイプがエラー状態になった場合に行ってください。パイプの状態を確認するには、usb_pipe_get_state(9F) を使用します。
- usb_pipe_drain_reqs(9F) 関数は、保留中のすべての要求が完了するまで待ちながらブロックしたあと、処理を続行します。この関数は、無期限に待機することも、指定された期間のあとでタイムアウトすることもできます。usb_pipe_drain_reqs(9F) 関数は、パイプを閉じることもフラッシュすることもしません。

デバイス状態管理

USB デバイスの管理では、ホットプラグ、システム電源管理 (チェックポイントと復元再開)、およびデバイス電源管理を考慮する必要があります。クライアントドライバは必ず、次の図に示す基本的な状態マシンを実装するべきです。詳細については、/usr/include/sys/usb/usbai.h を参照してください。

図 20–4 USB デバイスの状態マシン



① デバイスが抜かれる
② 元のデバイスが再接続
③ デバイスが時間 T の間アイドル状態で低電力状態に遷移
④ デバイスまたはデバイスに入出力を送信するアプリケーションによるリモートウェイクアップ
⑤ DDI_SUSPEND によって状態を保存する通知
⑥ DDI_RESUME によって正しいデバイスで状態を復元する通知
⑦ DDI_RESUME によって切断されたデバイスまたは間違ったデバイスで状態を復元する通知

この状態マシンと 4 つの状態は、ドライバ固有の状態で拡張できます。`0x80` から `0xff` のデバイス状態を定義して使用できるのは、クライアントドライバだけです。

USB デバイスのホットプラグ

USB デバイスはホットプラグをサポートします。USB デバイスの挿入や取り外しはいつでも行えます。クライアントドライバは、オープン状態のデバイスの取り外しと再挿入を処理する必要があります。オープン状態のデバイスを処理するには、ホットプラグコールバックを使用します。クローズ状態のデバイスの挿入と取り外しは、`attach(9E)` および `detach(9E)` エントリポイントによって処理されます。

ホットプラグコールバック

USBA 2.0 フレームワーク では次のイベント通知がサポートされています。

- デバイスが電源を入れたまま取り外された場合、クライアントドライバはコールバックを受け取ります。
- デバイスが電源を入れたまま取り外されたあとで元に戻された場合、クライアントドライバはコールバックを受け取ります。このイベントコールバックが発生する可能性があるのは、デバイスのドライバインスタンスがオフラインになってい

ない状態でユーザーがデバイスを元のポートに戻した場合です。ドライバインスタンスがオープン状態に保たれていれば、ドライバインスタンスがオフラインになることはありません。

クライアントドライバは、usb_register_hotplug_cbs(9F) ルーチン内で attach(9E) を呼び出してイベントコールバック用の登録を行う必要があります。ドライバは、detach(9E) ルーチン内で usb_unregister_hotplug_cbs(9F) を呼び出したあとで、設定解除を行う必要があります。

電源を入れたまま挿入

電源を入れたまま USB デバイスを挿入した場合のイベントシーケンスは、次のとおりです。

1. ハブドライバ hubd(7D) が、ポート接続ステータスが変化するまで待ちます。
2. hubd ドライバがポート接続を検出します。
3. hubd ドライバがデバイス内の情報を列挙し、子デバイスノードを作成し、クライアントドライバを接続します。互換名の定義については、478 ページの「クライアントドライバのバインド」を参照してください。
4. クライアントドライバがデバイスを管理します。ドライバの状態は ONLINE です。

電源を入れたまま取り外し

電源を入れたまま USB デバイスを取り外した場合のイベントシーケンスは、次のとおりです。

1. ハブドライバ hubd(7D) が、ポート接続ステータスが変化するまで待ちます。
2. hubd ドライバがポート切り離しを検出します。
3. hubd ドライバが、切り離しイベントを子クライアントドライバに送信します。子クライアントドライバが hubd ドライバまたは usb_mid(7D) マルチインタフェースドライバである場合、子クライアントドライバはそのイベントを子に伝播します。
4. クライアントドライバが、カーネルスレッドコンテキストで切り離しイベント通知を受け取ります。カーネルスレッドコンテキストでは、ドライバの切り離しハンドラのブロックが可能となります。
5. クライアントドライバの状態が DISCONNECTED に遷移します。device not responding という完了理由で未処理の入出力転送が失敗します。新しい入出力転送やデバイスノードのオープン試行もすべて失敗します。クライアントドライバがパイプを閉じる必要はありません。ドライバは、デバイスがふたたび接続された場合に復元する必要のあるデバイスやドライバのコンテキストを保存しておく必要があります。
6. hubd ドライバは、OS デバイスノードとその子を下から順にオフラインにしようとします。

hubd ドライバがデバイスノードをオフラインにしようとした時点でデバイスノードが開いていなかった場合、次のイベントが発生します。

1. クライアントドライバの detach(9E) エントリポイントが呼び出されます。
2. デバイスノードが破棄されます。
3. そのポートが新しいデバイスから使用可能になります。
4. ホットプラグのイベントシーケンスが最初から始まります。hubd ドライバが、ポート接続ステータスが変化するまで待ちます。

hubd ドライバがデバイスノードをオフラインにしようとした時点でデバイスノードが開いていた場合、次のイベントが発生します。

1. hubd ドライバが、定期的オフライン再試行キューにオフライン要求を置きます。
2. そのポートは、新しいデバイスから使用不可能なままになります。

hubd ドライバがデバイスノードをオフラインにしようとした時点ではデバイスノードが開いていたが、その後ユーザーがそのデバイスノードを閉じた場合、hubd ドライバによるデバイスノードの定期的なオフライン化が成功し、次のイベントが発生します。

1. クライアントドライバの detach(9E) エントリポイントが呼び出されます。
2. デバイスノードが破棄されます。
3. そのポートが新しいデバイスから使用可能になります。
4. ホットプラグのイベントシーケンスが最初から始まります。hubd ドライバが、ポート接続ステータスが変化するまで待ちます。

ユーザーがそのデバイスを使用するアプリケーションをすべて閉じると、そのポートがまた使用可能になります。アプリケーションが終了しない場合やデバイスがクローズされない場合、そのポートは使用不可能なままになります。

電源を入れたまま再挿入

デバイスのデバイスノードがまだ開いている間に以前に取り外されたデバイスが同じポートに再挿入された場合、次のイベントが発生します。

1. ハブドライバ hubd(7D) がポート接続を検出します。
2. hubd ドライバがバスアドレスとデバイス構成を復元します。
3. hubd ドライバがオフライン再試行要求を取り消します。
4. hubd ドライバが、接続イベントをクライアントドライバに送信します。
5. クライアントドライバが接続イベントを受け取ります。
6. クライアントドライバが、新しいデバイスが以前接続されていたデバイスと同じものかどうかを判定します。クライアントドライバはこの判定を行うために、まずデバイス記述子を比較します。クライアントドライバは、シリアル番号や構成記述子クラウドも比較する可能性があります。

クライアントドライバが、現在のデバイスが以前接続されていたデバイスと同じものでないと判定した場合、次のイベントが発生する可能性があります。

1. クライアントドライバがコンソールに警告メッセージを発行する可能性があります。
2. ユーザーがデバイスを再度取り外す可能性があります。ユーザーがデバイスを再度取り外した場合、電源を入れたまま取り外した場合のイベントシーケンスが最初から始まります。hubd ドライバがポート切り離しを検出します。ユーザーがデバイスを再度取り外さなかった場合、次のイベントが発生します。
   a. クライアントドライバの状態が DISCONNECTED のままになり、要求やオープンがすべて失敗します。
   b. そのポートは使用不可能なままになります。ユーザーはポートを解放するために、デバイスを閉じて取り外す必要があります。
   c. ポートが解放されると、ホットプラグのイベントシーケンスが最初から始まります。hubd ドライバが、ポート接続ステータスが変化するまで待ちます。

クライアントドライバが、現在のデバイスが以前接続されていたデバイスと同じものである判定した場合、次のイベントが発生する可能性があります。

1. クライアントドライバが状態を復元し、通常の動作を継続する可能性があります。このポリシーに従うかどうかは、クライアントドライバしだいです。クライアントドライバが処理を継続するべきである場合の例として、オーディオスピーカーが挙げられます。
2. 再接続されたデバイスを使用し続けても安全である場合には、ホットプラグのイベントシーケンスが最初から始まります。hubd ドライバが、ポート接続ステータスが変化するまで待ちます。デバイスがふたたびサービスを提供できる状態になります。

電源管理

このセクションでは、デバイス電源管理とシステム電源管理について説明します。

デバイス電源管理は、個々の USB デバイスをその入出力動作やアイドル状態かどうかに応じて管理します。

システム電源管理は、チェックポイントと復元再開を使用することで、システムの状態をファイル内にチェックポイントし、システムを完全に停止します (チェックポイントは「システム保存停止」と呼ばれることもあります)。システムの電源を再度入れると、システムは保存停止される前の状態に復元再開されます。

デバイス電源管理

次のサマリーは、USB デバイスの電源管理のためにドライバ内で行う必要のある処理を箇条書きにしたものです。電源管理の詳細については、このサマリーのあとで説明します。

1. attach(9E) 内で電源管理部品を作成します。usb_create_pm_components(9F) のマニュアルページを参照してください。
2. power(9E) エントリポイントを実装します。
3. デバイスにアクセスする前に、pm_busy_component(9F) と pm_raise_power(9F) を呼び出します。
4. デバイスへのアクセスが終了したら pm_idle_component(9F) を呼び出します。

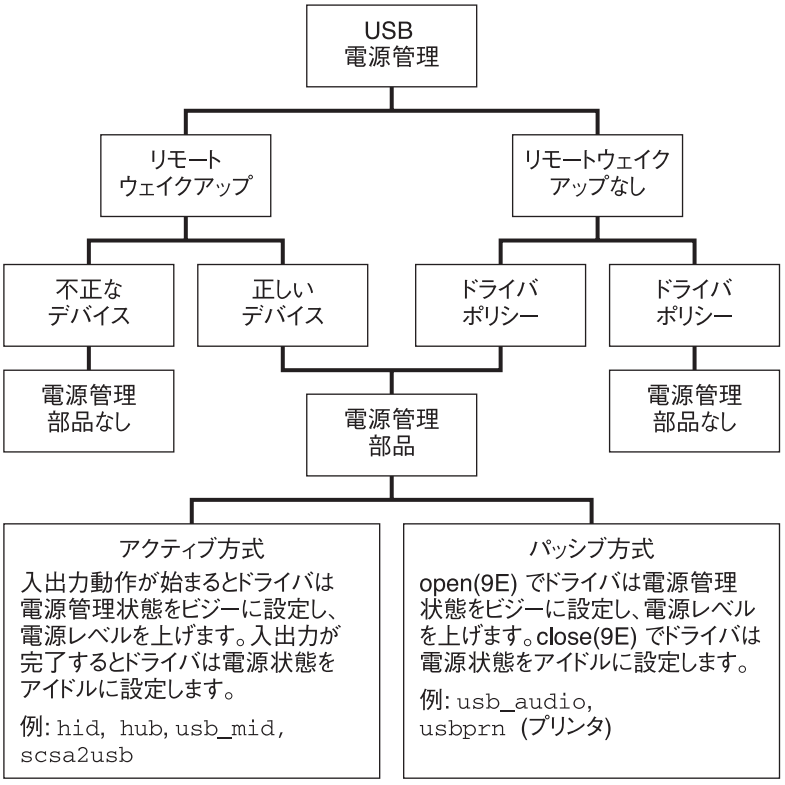
USBA 2.0 フレームワークは、USB インタフェースの電源管理仕様で規定されている 4 つの電源レベルをサポートしています。USB 電源レベルとオペレーティングシステム電源レベルとの対応関係については、/usr/include/sys/usb/usbai.h を参照してください。

デバイスの状態が USB_DEV_OS_PWR_OFF に遷移すると、hubd ドライバはポートを保存停止します。デバイスの状態が USB_DEV_OS_PWR_1 以上に遷移すると、hubd ドライバはポートを復元再開します。ポート保存停止とシステム保存停止は異なります。ポート保存停止の場合、停止されるのは USB ポートだけです。システム保存停止については、505 ページの「システム電源管理」 で定義します。

クライアントドライバでは、デバイスでのリモートウェイクアップを有効化できます。usb_handle_remote_wakeup(9F) のマニュアルページを参照してください。hubd ドライバは、あるポート上でリモートウェイクアップを検出すると、ウェイクアップ処理を実行したあと、pm_raise_power(9F) を呼び出して子に通知します。

次の図は、電源管理のさまざまな部分の間の関係を示したものです。

図 20–5　USB 電源管理



ドライバは、図 20–5 の下部で説明した 2 つの電源管理方式のいずれかを実装できます。パッシブ方式は、デバイス転送中に電源管理を行わないので、アクティブ方式よりも単純です。

アクティブ電源管理

このセクションでは、アクティブ電源管理スキームを実装するために使用する必要のある関数について説明します。

ドライバの attach(9E) エントリポイントで次の処理を行います。

1. usb_create_pm_components(9F) を呼び出します。
2. 必要に応じて、第 2 引数に USB_REMOTE_WAKEUP_ENABLE を指定して usb_handle_remote_wakeup(9F) を呼び出すことで、デバイスでのリモートウェイクアップを有効にします。
3. pm_busy_component(9F) を呼び出します。
4. pm_raise_power(9F) を呼び出して電源レベルを USB_DEV_OS_FULL_PWR にします。

5. デバイスとの通信を行ってデバイスを初期化します。
6. pm_idle_component(9F) を呼び出します。

ドライバの detach(9E) エントリポイントで次の処理を行います。

1. pm_busy_component(9F) を呼び出します。
2. pm_raise_power(9F) を呼び出して電源レベルを USB_DEV_OS_FULL_PWR にします。
3. attach(9E) エントリポイントで usb_handle_remote_wakeup (9F) 関数を呼び出した場合、ここで第 2 引数に USB_REMOTE_WAKEUP_DISABLE を指定して usb_handle_remote_wakeup(9F) を呼び出します。
4. デバイスとの通信を行ってデバイスを完全に停止させます。
5. pm_lower_power(9F) を呼び出して電源レベルを USB_DEV_OS_PWR_OFF にします。

   これが、クライアントドライバ内で pm_lower_power(9F) を呼び出す唯一の場合です。
6. pm_idle_component(9F) を呼び出します。

あるドライバスレッド内でデバイスへの入出力を開始する必要がある場合、そのスレッドで次のタスクを実行します。

1. pm_busy_component(9F) を呼び出します。
2. pm_raise_power(9F) を呼び出して電源レベルを USB_DEV_OS_FULL_PWR にします。
3. 入出力転送を開始します。

ドライバ内で入出力転送が完了したという通知を受け取ったら、pm_idle_component(9F) を呼び出します。

ドライバの power(9E) エントリポイントでは、遷移先の電源レベルが有効かどうかをチェックします。また、複数のスレッドが同時に power(9E) への呼び出しを行うことを考慮しなければならない可能性もあります。

デバイスのアイドル状態が一定時間続いたか、またはシステムが停止しようとしている場合、デバイスの状態を USB_DEV_OS_PWR_OFF にするために power(9E) ルーチンが呼び出される可能性があります。この状態は、図 20–4 で示した PWRED_DWN 状態に対応しています。デバイスの状態が USB_DEV_OS_PWR_OFF に遷移する場合、power(9E) ルーチンで次の処理を行います。

1. 開かれたパイプをすべてアイドル状態にします。たとえば、割り込みパイプのポーリングを停止します。
2. 保存する必要のあるデバイスやドライバのコンテキストをすべて保存します。

   power(9E) の呼び出しが完了したあと、デバイスが接続されているポートが保存停止されます。

デバイス起動のリモートウェイクアップまたはシステム起動のウェイクアップが受信されると、デバイスの電源を入れるために power(9E) ルーチンが呼び出される可能

性があります。ウェイクアップ通知が発生するのは、アイドル時間が長く続いたかシステムが保存停止したために、デバイスの電源レベルが低下したあとです。デバイスの状態が USB_DEV_OS_PWR_1 に遷移する場合、power(9E) ルーチンで次の処理を行います。

1. デバイスやドライバの必要なコンテキストをすべて復元します。
2. 指定された電源レベルに適したアクティビティーをパイプ上で再開します。たとえば、割り込みパイプでポーリングを開始します。

デバイスが接続されているポートが以前保存停止されていた場合、power(9E) が呼び出される前にそのポートが復元再開されます。

パッシブ電源管理

パッシブ電源管理方式は、前述のアクティブ電源管理方式より単純です。このパッシブ方式では、転送中の電源管理は行われません。このパッシブ方式を実装するには、デバイスを開くときに pm_busy_component(9F) と pm_raise_power(9F) を呼び出します。その後、デバイスを閉じるときに pm_idle_component(9F) を呼び出します。

システム電源管理

システム電源管理は、状態保存後のシステム全体の電源切断とシステム電源再投入後の状態復元とで構成されます。この処理は *CPR* (チェックポイントおよび復元再開) と呼ばれます。CPR に関する USB クライアントドライバの動作は、ほかのクライアントドライバの動作と同じです。デバイスを保存停止する場合は、ドライバの detach(9E) エントリポイントが *cmd* 引数 DDI_SUSPEND で呼び出されます。デバイスを復元再開する場合は、ドライバの attach(9E) エントリポイントが *cmd* 引数 DDI_RESUME で呼び出されます。detach(9E) ルーチンで DDI_SUSPEND コマンドを処理する場合、あとで完全な復元再開を行うのに必要な範囲で、デバイス状態やドライバ状態をクリーンアップします (この状態は図 20–4 の SUSPENDED に対応しています)。attach(9E) ルーチンで DDI_RESUME コマンドを処理する場合、システムとデバイスの同期が取れるように、必ずデバイスの電源レベルをフルにします。

USB デバイスの場合、保存停止/復元再開はホットプラグの切り離し/再接続 (498 ページの「USB デバイスのホットプラグ」を参照) と同様に処理されます。CPR とホットプラグの重要な違いは、CPR では、デバイスが保存停止可能な状態にない場合にドライバでのチェックポイント処理が失敗する可能性があるという点です。たとえば、デバイスのエラー回復が進行中である場合、デバイスの保存停止は行えません。デバイスがビジー状態で安全に停止できない場合も、デバイスの保存停止は行えません。

直列化

ドライバでは一般に、相互排他を保持した状態で USBA 関数を呼び出すべきではありません。したがって、クライアントドライバ内で競合状態が発生することが避けられない可能性があります。

切り離しや CPR といった非同期イベントの処理コードと通常動作コードの同時実行を許可しないでください。これらのタイプの非同期イベントでは通常、パイプのクリーンアップと設定解除が行われるため、通常動作コードに悪影響が及ぶ可能性があります。

競合状態を管理して通常動作コードを保護する方法の 1 つは、排他アクセス権を持つ同期オブジェクトの取得と解放を行える直列化機能を記述することです。USBA 関数の呼び出し中に同期オブジェクトを安全に保持できるように、直列化機能を記述できます。`usbskel` サンプルドライバでこの手法が使用されています。`usbskel` ドライバについては、509 ページの「サンプル USB デバイスドライバ」を参照してください。

ユーティリティー関数

このセクションでは、一般的な用途の関数をいくつか説明します。

デバイス構成機能

このセクションでは、デバイス構成に関係する関数について説明します。

インタフェース番号の取得

マルチインタフェースデバイスを使用していて、`usb_mid(7D)` ドライバによって呼び出し元ドライバから使用可能なインタフェースが 1 つだけに限定されている場合、呼び出し元ドライバがバインドされているインタフェースの番号を知らなければならない可能性があります。`usb_get_if_number(9F)` 関数を使用して次のいずれかのタスクを行います。

- 呼び出し元ドライバがバインドされているインタフェースの番号を返します。この場合、`usb_get_if_number(9F)` 関数はゼロより大きいインタフェース番号を返します。
- 呼び出し元ドライバがマルチインタフェースデバイス全体を管理していることを検出します。このドライバはデバイスレベルでバインドされているため、`usb_mid` によるデバイスの分割は行われませんでした。この場合、`usb_get_if_number(9F)` 関数は `USB_DEVICE_NODE` を返します。
- 呼び出し元ドライバが、現在の構成でデバイスが提供する唯一のインタフェースを管理することでデバイス全体を管理していることを検出します。この場合、`usb_get_if_number(9F)` 関数は `USB_COMBINED_NODE` を返します。

デバイス全体の管理

ある複合デバイス全体を管理するドライバは、ベンダー ID、製品 ID、およびリビジョン ID を含む互換名を使用することで、そのデバイス全体にバインドできます。複合デバイス全体にバインドされたドライバは、ネクサスドライバと同様に、そのデバイスのすべてのインタフェースを管理する必要があります。一般に、複合デバイス全体にドライバをバインドするべきではありません。代わりに、汎用マルチインタフェースドライバ usb_mid(7D) を使用してください。

ドライバがあるデバイス全体を所有しているかどうかを判定するには、usb_owns_device(9F) 関数を使用します。デバイスは複合デバイスであってもかまいません。ドライバがデバイス全体を所有している場合、usb_owns_device(9F) 関数から TRUE が返されます。

複数の構成を持つデバイス

USB デバイスは、ある時点でホストから使用可能な構成を 1 つだけに限定します。ほとんどのデバイスで、単一の構成しかサポートされていません。ただし、いくつかの USB デバイスでは複数の構成がサポートされています。

複数の構成を持つデバイスは必ず、使用可能なドライバが存在する最初の構成に配置されます。一致を検索するとき、デバイス構成が番号順に検討されます。一致するドライバが見つからなかった場合、デバイスは最初の設定に設定されます。その場合、usb_mid ドライバがデバイスを管理し、デバイスをいくつかのインタフェースノードに分割します。デバイスの現在の構成を返すには、usb_get_cfg(9F) 関数を使用します。

次の 2 つの方法のいずれかを使用すると、別の構成を要求できます。これら 2 つの方法のいずれかを使用してデバイス構成を変更すると、USBA モジュールとデバイスの同期が確実に維持されます。

- cfgadm_usb(1M) コマンドを使用します。
- ドライバから usb_set_cfg(9F) 関数を呼び出します。

  デバイス設定を変更するとデバイス全体に影響が及ぶため、クライアントドライバが次の条件をすべて満たしていないと、usb_set_cfg(9F) 関数の呼び出しが成功しません。

  - クライアントドライバがデバイス全体を所有している必要があります。
  - デバイスの子ノードが 1 つも存在していない必要があります。ほかのドライバが子ノードを通じてデバイスを駆動する可能性があるからです。
  - デフォルトパイプを除くすべてのパイプが閉じられている必要があります。
  - デバイスが複数の構成を持っている必要があります。

注意 – SET_CONFIGURATION USB 要求を手動で行うことでデバイス設定を変更しないでください。SET_CONFIGURATION 要求による設定の変更はサポートされていません。

代替設定の変更または取得

クライアントドライバから usb_set_alt_if(9F) 関数を呼び出すと、現在選択されているインタフェースの選択済みの代替設定を変更できます。明示的に開かれたパイプを必ずすべて閉じてください。usb_set_alt_if(9F) 関数は、代替設定を切り替える際に、デフォルトパイプのみが開かれていることを確認します。usb_set_alt_if(9F) を呼び出す前に、デバイスの準備が整っていることを確認してください。

代替設定を変更すると、ドライバから使用可能なエンドポイント、クラス固有記述子、およびベンダー固有記述子に影響が及ぶ可能性があります。エンドポイントや記述子の詳細については、483 ページの「記述子ツリー」を参照してください。

現在の代替設定の番号を取得するには、usb_get_alt_if(9F) 関数を呼び出します。

注 – 新しい代替設定、新しい構成、または新しいインタフェースを要求する際には、デバイスへのデフォルトパイプ以外のすべてのパイプを閉じる必要があります。これは、代替設定、構成、またはインタフェースを変更すると、デバイスの動作モードが変わるからです。また、代替設定、構成、またはインタフェースを変更すると、システムでのデバイスの表現も変わります。

その他のユーティリティー関数

このセクションでは、USB デバイスドライバで役立つその他の関数について説明します。

文字列記述子の取得

インデックスを指定して文字列記述子を取得するには、usb_get_string_descr(9F) 関数を呼び出します。一部の構成、インタフェース、またはデバイス記述子には、文字列の ID が関連付けられています。そのような記述子には、ゼロ以外の値を含む文字列インデックスフィールドが含まれています。文字列インデックスフィールドの値を usb_get_string_descr(9F) に渡すと、対応する文字列を取得できます。

パイプの非公開データ機能

各パイプは、クライアントドライバ専用に確保された領域のポインタを 1 つずつ備えています。値をインストールするには、usb_pipe_set_private(9F) 関数を使用します。値を取得するには、usb_pipe_get_private(9F) 関数を使用します。この機能がコールバックで役に立つのは、パイプが独自のクライアント定義状態を特定処理の対象としてコールバックに渡す必要がある場合です。

USB 状態のクリア

次のタスクを行うには、usb_clr_feature(9F) 関数を使用します。

- あるエンドポイントの停止状態をクリアするために、USB CLEAR_FEATURE 要求を発行します。
- デバイス上のリモートウェイクアップ状態をクリアします。
- デバイスレベル、インタフェースレベル、またはエンドポイントレベルのデバイス固有状態をクリアします。

デバイス、インタフェース、エンドポイントの各ステータスの取得

USB GET_STATUS 要求を発行してデバイス、インタフェース、またはエンドポイントのステータスを取得するには、usb_get_status(9F) 関数を使用します。

- デバイスのステータス: 自己電源とリモートウェイクアップが有効。
- インタフェースのステータス: USB 2.0 仕様に従ってゼロを返す。
- エンドポイントのステータス: エンドポイントが停止。このステータスは機能ストールを示します。停止をクリアしないと、デバイスをふたたび動作させることができません。

  プロトコルストールは、サポートされない制御パイプ要求が行われたことを示します。プロトコルストールは、次回の制御転送の開始時に自動的にクリアされます。

デバイスのバスアドレスの取得

デバイスの USB バスアドレスをデバッグ目的で取得するには、usb_get_addr(9F) 関数を使用します。このアドレスは特定の USB ポートに対応します。

サンプル USB デバイスドライバ

このセクションでは、Oracle Solaris 環境用の USBA 2.0 フレームワーク を使用するテンプレート USB デバイスドライバについて説明します。このドライバには、この章で説明した機能の多くの使用例が含まれています。このテンプレートドライバまたはスケルトンドライバの名前は、usbskel です。

usbskel ドライバは、ユーザーが独自の USB デバイスドライバを開始する際に使用可能なテンプレートです。usbskel ドライバには次の機能の使用例が含まれています。

- デバイスの raw 構成データの読み取り。USB デバイスは必ず、自身の raw 構成データを報告できる必要があります。
- パイプの管理。usbskel ドライバは、パイプの管理方法を示すために割り込みパイプを開きます。

- ポーリング。`usbskel` ドライバのコメントで、ポーリングの実行方法を説明しています。
- USB バージョン管理と登録。
- USB ロギング。
- USB ホットプラグへの対応。
- Oracle Solaris の保存停止/復元再開への対応。
- 電源管理への対応。
- USB 直列化。
- USB コールバックの使用。

◆ ◆ ◆ 第 21 章

21 SR-IOV ドライバ

この章では、シングルルート IO 仮想化 (SR-IOV) デバイスドライバについて説明し、次のトピックに関する情報を提供します。

SR-IOV の概要

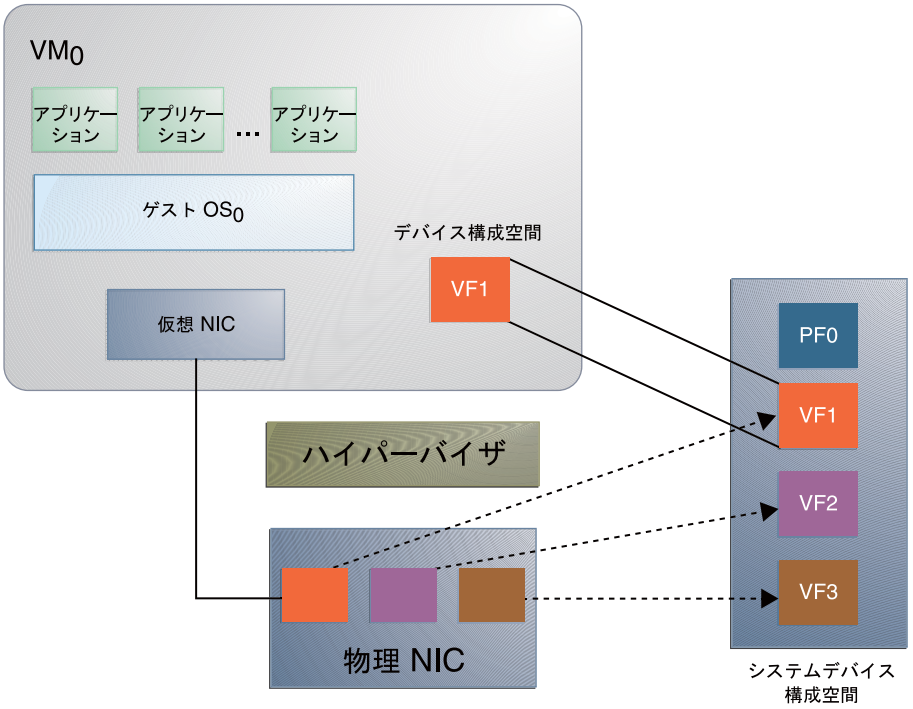
SR-IOV テクノロジは、パフォーマンスとスケーラビリティーの両方を改善する、ハードウェアベースの仮想化ソリューションです。SR-IOV 標準は仮想マシン間での PCIe (Peripheral Component Interconnect Express) デバイスの効率的な共有を可能にし、この標準は、ネイティブのパフォーマンスに匹敵する I/O パフォーマンスを実現できるようハードウェアに実装されています。SR-IOV 仕様では、作成された新しいデバイスで仮想マシンを I/O デバイスに直接接続できるようにするための新たな標準が規定されています。

SR-IOV 仕様は、PCI-SIG (`http://www.pcisig.com`) によって策定および維持されています。

多くの仮想マシンが単一の入出力リソースを共有できます。共有されたデバイスでは、専用のリソースが提供されるほか、共有された共通リソースも利用されます。このような方法で、各仮想マシンは一意のリソースにアクセスできます。したがって、適切なハードウェアおよび OS サポート付きの SR-IOV 対応 Ethernet ポートなどの PCIe デバイスを、独自の構成空間を備えた異なる複数の物理デバイスとして提供できます。

次の図は、PCIe ハードウェアの SR-IOV テクノロジを示したものです。

図 21–1　SR-IOV テクノロジ



SR-IOV の 2 つの新しい機能タイプ:

物理機能 (PF)　SR-IOV 仕様に規定された SR-IOV 機能をサポートする PCI 機能。PF は SR-IOV 機能構造を含んでおり、SR-IOV の機能を管理するために使用されます。PF は完全な機能を備えた PCIe 機能であり、他の PCIe デバイスと同様に発見、管理、および操作を行えます。PF は完全な構成リソースを備えているため、PCIe デバイスの構成や制御に使用できます。

仮想機能 (VF)　物理機能に関連付けられた機能。VF は軽量な PCIe 機能であり、物理機能やその PF に関連付けられたほかの VF と、1 つ以上の物理リソースを共有します。VF が持つことを許可されている構成リソースは、自身の動作に対するものだけです。

各 SR-IOV デバイスは 1 つの物理機能 (PF) を持つことができ、各 PF には最大 64,000 個の仮想機能 (VF) を関連付けることができます。VF の作成は、この目的のための特性に配慮して設計されたレジスタ経由で PF によって行われます。

PF で SR-IOV が有効化されると、PF のバス、デバイス、および機能番号 (ルーティング ID) を使って各 VF の PCI 構成空間にアクセスできます。各 VF には、レジスタセットのマッピングに使用される PCI メモリー空間もあります。VF のデバイスドライバは、レジスタセットを操作してその機能を有効にし、実際に存在する PCI デバイスとして表示されます。VF の作成が完了したら、IO ゲストドメインや個々のアプリケーション (基本的なプラットフォーム上の Oracle Solaris ゾーンなど) に VF を直接割り当てることができます。この機能により、それらは物理デバイスを共有したり、CPU やハイパーバイザソフトウェアのオーバーヘッドなしに入出力を実行したりできるようになります。

SR-IOV の利点

SR-IOV 標準では、IO ゲストドメイン間で PCIe デバイスを効率的に共有できます。SR-IOV デバイスでは、1 つの物理機能 (PF) に数百個の仮想機能 (VF) を関連付けることができます。VF の作成は、SR-IOV 機能を有効にするために設計されたレジスタ経由で、PF から動的に制御できます。デフォルトでは、SR-IOV 機能は無効になっており、PF は従来の PCIe デバイスとして動作します。

SR-IOV 機能を備えたデバイスは次の利点を享受できます。

- パフォーマンス – 仮想マシン環境からハードウェアへの直接アクセス。
- コスト削減 – 設備費や運用費の節約:
  - 節電
  - アダプタ数の減少
  - より少ないケーブル
  - スイッチポート数の減少

サポートされるプラットフォーム

- SPARC: すべての T3 および T4 シリーズのシステムで Oracle Solaris SR-IOV 対応デバイスがサポートされます。
- x86: Intel Nehalem-EX ベースのシステム (x4470 や x4800 (G+) など) で Oracle Solaris SR-IOV 対応デバイスがサポートされます。

次のカードは SR-IOV 対応デバイスをサポートします。

- Kawela (82576、1G) など、Intel ベースの NIC
- Niantic (82599、10G)

注 – SR-IOV 対応デバイスの実行を可能にするには、システムファームウェアが SR-IOV 機能をサポートすべきです。

用語

制御ドメイン	仮想化のポリシーを管理するドメイン。
ルートドメイン	PCIe ファブリックを管理するドメイン。
IO ドメイン	割り当てられた IO デバイスに排他的にアクセスするドメイン。
ファブリック	ルートコンプレックス、ルートポート、スイッチ、ブリッジなどの PCIe ファブリックコンポーネント。エンドポイントの構成空間レジスタ
VM	仮想マシン
PCI デバイス **(PCI** コンポーネント)	■ 単一または複数の PCI 機能 ■ 通常は 1 片のシリコン
PCI 機能	■ PCI ファブリック内の、アドレス指定可能な独立した最小のユニット ■ 一連の PCI 構成空間レジスタ
PCI 仮想機能	■ ハードウェア内の軽量 PCI 機能 ■ ソフトウェアからは、PCI 機能とほとんど同じに見えます

SR-IOV デバイスドライバの概要

SR-IOV 機能は、物理機能 (PF) ドライバと仮想機能 (VF) ドライバから構成されます。次の各セクションでは、PF ドライバと VF ドライバ、および必要なデバイス構成の詳細について説明します。

物理機能 (PF) ドライバ

SR-IOV デバイスの PF ドライバは、SR-IOV 対応デバイスの物理機能 (PF) を管理するために使用されます。SR-IOV 機能をサポートする PCI 機能については、SR-IOV 仕様で規定されています。PF は SR-IOV 機能構造を含んでおり、SR-IOV の機能を管理するために使用されます。PF は完全な機能を備えた PCIe 機能であり、ほかの PCIe デバイスと同様に発見、管理、および操作を行えます。PF は完全な構成リソースを備えているため、PCIe デバイスの構成や制御に使用できます。PF ドライバが示す特性は次のとおりです。

- ルートドメインでのみ表示されます

- データ移動機能を備えている場合も備えていない場合もあります。PF ドライバは SR-IOV モードでも機能すべきです。
- Oracle Solaris IOV フレームワークで提供されている API 経由で SR-IOV 機能の有効化と無効化を制御します。
- ある特定の PF で構成される VF の数は、システム管理者によって決定されます。この数は、Sparc OVM プラットフォームではマシン記述子 (MD) 内に、基本的な環境では構成ファイル内に、それぞれ定義されます。
- PF ドライバは、接続段階で VF を有効化するために、DDI インタフェース経由で Oracle Solaris IOV フレームワークを呼び出します。接続時に PF ドライバが VF を有効化しなかった場合、そのドライバが IOV 機能を備えていることがドライバコールバックフラグに示されているかぎり、Oracle Solaris IOV フレームワークはその接続後に VF の構成を試みます。
- PF はデバイス固有のメカニズムを介して、関連付けられた各 VF を個別に有効化および無効化できます。

仮想機能 (VF) ドライバ

物理機能に関連付けられた機能。VF は軽量な PCIe 機能であり、物理機能やその PF に関連付けられたほかの VF と、1 つ以上の物理リソースを共有します。VF ドライバが示す特性は次のとおりです。

- ルートドメインと I/O ドメインの両方に表示されます
- HW メールボックスと OS 提供インタフェースのいずれかを使って PF との通信を開始できます
- 次の条件を満たすまでルートドメインに表示されません。
  - ルートドメインがブート済みである
  - PF ドライバが接続し、VF の構成を呼び出す
  - ルートドメインの Oracle Solaris IOV フレームワークによって VF が有効化される
  - システムファームウェアによってリソースが VF に割り当てられる
- 次の条件を満たすまで I/O ドメインに表示されません。
  - VF がすでに有効化され、ルートドメインに表示されている
  - I/O ドメインに VF が割り当てられている
  - Oracle Solaris ファームウェア (OBP) によって I/O ドメイン内で VF がプローブされる

注 – SR-IOV 対応の PF および VF ドライバは、割り込みリソース管理 (IRM) コールバックを登録し、この機能のサポートを提供する必要があります。IRM インタフェースの詳細や使用方法については、第 8 章「割り込みハンドラ」を参照してください。

注 – VF がネットワーク VF の場合、numVFs の有効化が完了すると、次のパラメータを構成できます。VF を有効にする前に構成を完了しておくべきです。

- mac-addr
- vlan (Virtual LAN) ID
- port-vlan-id
- alt-mac-addrs
- mtu

デバイス構成パラメータ

PF ドライバは、次の表に記載された構成パラメータをサポートする必要があります。これらのパラメータは Sparc OVM Manager にエクスポートできます。構成が完了するのは、すべてのパラメータを構成した場合だけです。

表 21–1 構成パラメータの定義

構成パラメータ	定義	例
標準に関する構成パラメータ	サポート可能な VF の数 注 – VF の数を変更した場合、PF デバイスを切り離してから再度接続する必要があります。	max-config-vfs – 実際に構成可能な VF の最大数。PF ドライバがサポートする最大 VF 数が、SR-IOV 機能から示された性能と異なる場合、PF ドライバはこのパラメータをエクスポートできます。

表 21–1 構成パラメータの定義 (続き)

構成パラメータ	定義	例
リソース固有およびデバイス固有のパラメータ	帯域幅、プール、および Q-ペア。これらのパラメータを変更すると、その影響は PF ドライバと VF ドライバの両方に及びます。 デバイス固有のパラメータは、フレームワークから認識されていない可能性があり、PF ドライバにのみ認識されている可能性があります。VF を有効化する前にこれらのパラメータが認識され、PF ドライバがそのハードウェアを正しく初期化できるようにするべきです。 IOV フレームワークにエクスポートされたデバイス固有のパラメータを取得する方法を学ぶには、`igb(7D)` および `ixgbe(7D)` を参照してください。	■ `pvid-exclusive` – `port-vlan-id` と `vlan-ids` を同時にサポートすることはできないことを示します。 ■ `max-vf-mtu` – 1 つの VF で許可される最大の MTU。 ■ `max-vlans` – ネットワーククラスの PF ドライバでサポートされる vlan スロットの最大数。
クラス固有のパラメータ	デバイスのクラスに基づく一般的なプロパティー。たとえば Ethernet デバイスは、MAC アドレス、VLAN-ID、Port-VLAN-ID、帯域幅などのプロパティーを持つことができます。 クラス固有の構成を使用することが期待されており、構成では各パラメータの動作を定義できます。	なし

注 – デバイス構成パラメータを変更したら、デバイスを接続し直すようにしてください。

注 – VF を有効化する前に次の順序でパラメータを構成します。クラス固有のパラメータは、クラス固有のコンフィギュレータに基づいています。

1. 標準に関するパラメータ
2. リソース固有およびデバイス固有のパラメータ
3. クラス固有のパラメータ

pci.conf ファイル

PCI 構成情報ファイル /etc/pci.conf は、システムが PCI 構成 (特定 PF の VF 数など) を保存できるようにします。pci.conf ファイルは次を提供します。

- システムのブート時に VF を自動的に作成できるように PCI 構成を持続させること。
- この構成ファイルは boot_archive の一部になっているため、システムのブート中に VF を使用できます。

注 – /etc/pci.conf ファイルを Oracle Solaris ブートプロセスに含めるには、/etc/pci.conf を /boot/solaris/filelist.ramdisk ファイルに追加します。

詳細は、付録 E 「pci.conf ファイル」を参照してください。

デバイス構成パラメータの設定

- SPARC: ldm コマンド経由でパラメータを設定できます。詳細は、ldm(1M) のマニュアルページを参照してください。
- x86: pci.conf ファイルにクラス固有のパラメータを指定できます。次の例は、pci.conf ファイルに設定されたパラメータを示したものです。

例 21–1 デバイス構成パラメータの設定

```
[[path=/pci@0,0/pci8086,3a40@1c/pci108e,4848@0,1]]
num-vf=2

[Device_Configuration]
[[path=/pci@0,0/pci8086,3a40@1c/pci108e,4848@0,1]]
VF[0] = {
        primary-mac-addr = 0xaabbccddeeff
        alt-mac-addrs = 0x102233445556, 0x102233445557
        vlan-id = 20, 30
}

VF[1] = {
        primary-mac-addr = 0xaabbccddeef1
        alt-mac-addrs = 0x102233445568
        vlan-id = 20, 30, 40, 50
```

例 21–1　デバイス構成パラメータの設定　　(続き)

```
}
```

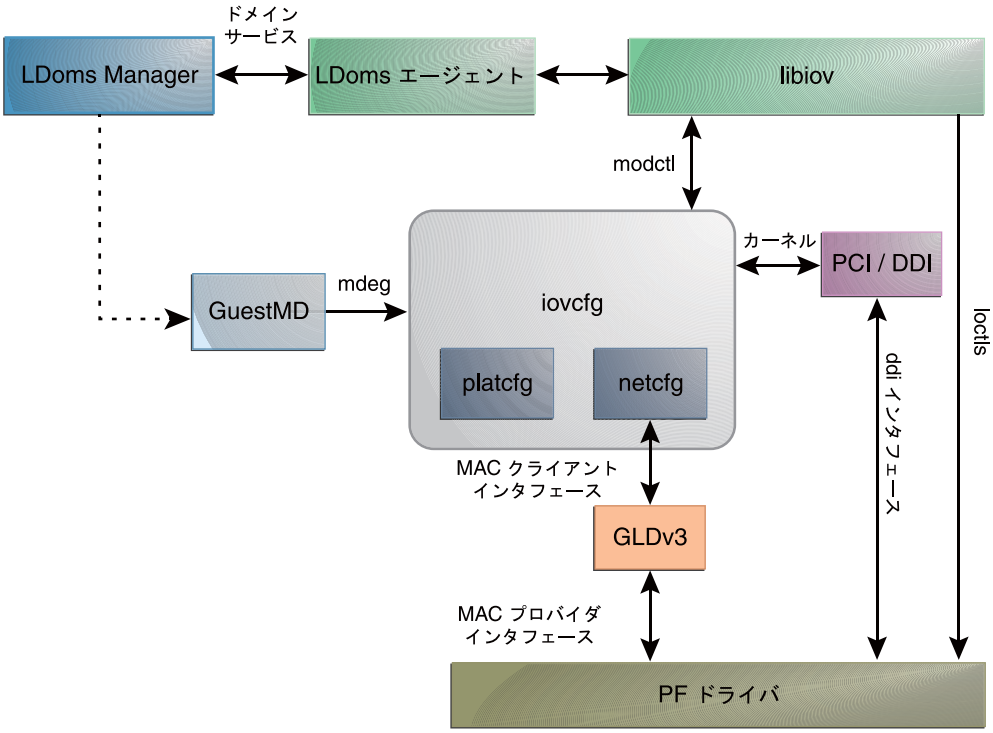
Sparc OVM プラットフォームでの SR-IOV 構成

すべての Sparc OVM プラットフォームで、SPARC OVM Manager が SR-IOV 構成を担当します。Sparc OVM Manager は次の処理を担当します。

- SR-IOV 対応ドライバを持つ PF のリストを取得します
- ドライバでサポートされるデバイス固有のパラメータを取得します
- 特定のデバイス構成を検証します
- マシン記述子 (MD) ファイルをすべての有効な構成詳細で更新するほか、VF の割り当てや削除も行います

次の図に、Sparc OVM 構成の概要図を示します。

図 21–2　Sparc OVM 構成の概要図



基本的なプラットフォームでの SR-IOV 構成

Oracle Solaris 10 Update 11 リリースの時点では、x86 のような基本的なプラットフォーム上で SR-IOV を構成する際に使用できる構成ツールはありません。

ブート構成シーケンス

SR-IOV 対応 PF ドライバは接続時に次のアクションを実行します。

1. `pciv_vf_config()` 関数を呼び出して VF の数を取得します。
2. PF と VF の両方のデバイス固有パラメータを取得し、それらを検証します。
3. それに応じてハードウェアを初期化します
4. `pciv_vf_config()` インタフェースを呼び出して VF を有効化します
5. PF ドライバがネットワークドライバの場合、ドライバは接続時に `mac_register()` インタフェースを使って GLDv3 フレームワークに登録します。さらに、PF ドライバはクラス固有の初期化も実行します。その結果、次の一連のアクションが発生します。
   - GLDv3 インタフェースが PF デバイスの存在を認識します。
   - 新しい MAC プロバイダインタフェースのセットが PF ドライバによってエクスポートされます。このプロセスにより、ドライバが PF ドライバであることを MAC 層が認識できるようになります。また MAC 層は、VF ドライバの詳細情報の取得も行います。

   ネットワークドライバおよびネットワークインタフェースの詳細については、第 19 章「ネットワークデバイスのドライバ」 および 『Oracle Solaris 管理: ネットワークインタフェースとネットワーク仮想化』の章を参照してください。

これで VF インスタンスが初期化されました。VF ドライバが接続されるのは、その VF がルートドメインに割り当てられている場合だけです。

SR-IOV インタフェースのサマリー

次の表に、SR-IOV ドライバで使用可能な Oracle Solaris インタフェースの一覧を示します。PF や VF のパラメータを取得したり、VF を構成したり、構成パラメータを任意の呼び出し元アプリケーションにエクスポートしたりする際に、これらのインタフェースを呼び出します。

注 – `pciv_send()` を除いて、この表に記載されたインタフェースはすべて PF ドライバにのみ適用可能です。

表 21–2 SR-IOV ドライバのインタフェース

インタフェース名	説明
pci_param_get(9F)	現在構成されているパラメータの名前-値ペアのリストを取得するために、PCI デバイスドライバによって使用されます
pci_param_free(9F)	デバイスドライバが *param* ハンドルを使ってデバイスパラメータを取得したあと、デバイスドライバから呼び出す必要があります。この呼び出しにより、pci_param_get() および pci_param_get_ioctl() インタフェースによって割り当てられたリソースが解放されます。
pci_plist_get(9F)	PF デバイスの名前-値ペアリストを取得するために、SR-IOV デバイスドライバによって使用されます。
pci_plist_getvf(9F)	VF デバイスの名前-値ペアリストを取得するために、SR-IOV デバイスドライバによって使用されます。
pci_param_get_ioctl(9F)	SR-IOV デバイスドライバに対するヘルパー関数で、このドライバが ioctl IOV_VALIDATE_PARAM を実装する場合、PF および VF デバイスのパラメータを arg 引数から抽出します。
pciv_vf_config(9F)	VF 構成パラメータを取得したり VF を構成したりするために、PF ドライバによって使用されます。
pci_plist_lookup(9F)	pci_plist_get() および pci_plist_getvf() インタフェースから取得されたリストから、PF および VF のパラメータを取得するために使用されます。
pciv_send(9F)	情報を送信するために、PF および VF ドライバによって使用されます。
ddi_cb_register(9F)	コールバック登録メカニズムのインタフェース。

ドライバの ioctl

SR-IOV ドライバの ioctl は、管理者が構成できるデバイス固有のパラメータを特定したり、特定の構成を適用する前にその構成を検証したりする場合に使用されます。次の ioctl とそのデータ構造体が定義されています。

- 530 ページの「iov_param_ver_info 構造体」
- 531 ページの「iov_param_validate 構造体」
- 531 ページの「iov_param_desc 構造体」
- 532 ページの「IOV_GET_VER_INFO ioctl 」
- 532 ページの「IOV_GET_PARAM_INFO Ioctl」
- 533 ページの「IOV_VALIDATE_PARAM ioctl」

SR-IOV ドライバのインタフェース

次の各セクションでは、SR-IOV ドライバのインタフェースについて説明します。

pci_param_get() インタフェース

SR-IOV ドライバで現在構成されているパラメータのリストを取得するには、pci_param_get(9F) インタフェースを使用する必要があります。このインタフェースの呼び出しは、ドライバの接続時やほかの適切なタイミングで行われます。返されるデータはパラメータリストへのポインタになりますが、これには、PF デバイスと対応する VF デバイスの両方の名前-値情報が含まれています。

```
int pci_param_get(dev_info_t *dip, pci_param_t *php)
```

各表記の意味は次のとおりです。

`dip` `dev_info` 構造体へのポインタ

`php` `param` ハンドル `pci_param_t` へのポインタ

デバイスドライバは、`pci_param_get` インタフェースを呼び出したあと、次の手順を実行して PF と VF のパラメータのリストを取得すべきです。

1. pci_plist_get(9F) インタフェースを呼び出して PF デバイスのパラメータのリストを取得し、pci_plist_getvf(9F) インタフェースを呼び出して構成対象 VF のパラメータのリストを取得します。
2. pci_plist_lookup(9F) インタフェースを呼び出してデバイスパラメータを取得します。
3. すべての PF および VF パラメータを検証します
4. パラメータが現在の構成と一致しない場合、ドライバはデバイスの接続を失敗させるべきです。
5. pci_param_free(9F) インタフェースを呼び出すことで、PF デバイスと構成対象 VF デバイスのパラメータを取得するパラメータハンドルへのポインタを解放します。例 21–2 を参照してください

注 – VF を構成する前に、パラメータの検証を完了するようにしてください。

名前-値ペアはデバイスごとに定義します。名前-値ペアは、PF 用に 1 セット、構成対象 VF ごとに 1 セットずつ存在します。名前-値ペアはオプションであり、デバイスの一部またはすべてで存在していない可能性があります。

例 21–2 SR-IOV pci_param_get(9F) ルーチン

```
    pci_param_t my_params;
pci_plist_t pf_plist;
pci_plist_t  vf_plist[8];
labelp = NULL;
rval = pci_param_get(dip,&my_params);
if (rval || (my_params == NULL)) {
   cmn_err(CE_NOTE, "No params available\n");
goto continue_with_attach;
}
rval = pci_plist_get(my_params, &pf_list);
if (rval || (pf_plist == NULL)) {
        cmn_err(CE_NOTE, "No params for PF \n");
goto continue_with_attach;
}
for (i = 0; i < 8; i++) {
    rval = pci_plist_getvf(my_params, i, &vf_plist[i]);
    if (rval || (vf_plist[i] == NULL)) {
     cmn_err(CE_WARN, "No params for VF %d\n", i);
     continue;
 }
}
pci_param_free(my_params);
/*
* Validate the PF and VF params lists.
* Fail the attach if the params are incompatible or exceed the
* resources available.
*/
continue_with_attach:
```

pci_param_get_ioctl() インタフェース

SR-IOV デバイスドライバは、IOV_VALIDATE_PARAM ioctl を実装している場合、pci_param_get_ioctl(9F) インタフェースを使用して、*arg* 引数から PF および VF デバイスのパラメータを抽出できます。

int pci_param_get_ioctl(dev_info_t *dip, intptr_t *arg*, int *mode*,pci_param_t **php*)

各表記の意味は次のとおりです。

dip　dev_info 構造体へのポインタ

arg　ドライバの ioctl コールから取得された引数

mode　ドライバの ioctl コールから取得された引数

php　pci_param_get() または pci_param_get_ioctl() インタフェースを呼び出すことで取得されたハンドルである *param* ハンドル pci_param_t へのポインタ

ドライバは、パラメータを取得したあと、pci_param_free() インタフェースを呼び出すことで、この呼び出しで返された param ハンドルを解放すべきです。

pci_plist_get() インタフェース

pci_plist_get(9F) インタフェースは、pci_param_get(9F) 呼び出しとpci_param_get_ioctl(9F) 呼び出しのいずれかから取得された param ハンドルからパラメータリストを取得するために使用します。

```
int pci_plist_get(pci_param_t param, pci_plist_t *plist_p)
```

各表記の意味は次のとおりです。

param　　pci_param_get() または pci_param_get_ioctl() インタフェースから取得されたハンドル。

plist_p　　pci_plist_t へのポインタ (成功時には NULL 以外の plist が返される)。

pci_plist_get() 呼び出しから返される plist は、PF 機能専用です。構造体 pci_plist_t では、次のデータ型の配列がサポートされます。

- int8_t
- uint8
- int16_t
- uint16_t
- int32_t
- uint32_t
- int64_t
- uint64_t
- char *

pci_plist_getvf() インタフェース

pci_plist_getvf(9F) インタフェースは、VF デバイスの名前-値ペアリストを取得するために使用します。

```
int pciv_plist_getvf (pci_param_t param, uint16_t vf_index, pci_plist_t *vfplist_p
)
```

各表記の意味は次のとおりです。

param　　pci_param_get() または pci_param_get_ioctl() インタフェースから取得されたハンドル。

vf_index　　0 から #VFS - 1 までの値。

*vfplist_p　　pci_plist_t 構造体へのポインタ。

pciv_vf_config() インタフェース

pciv_vf_config(9F) インタフェースは SR-IOV ドライバで、VF に関する構成情報を取得するために使用されるほか、ドライバの接続時に VF を構成するためにも使用されます。

```
#include <sys/sunddi.h>
int pciv_vf_config(dev_info_t *dip, pciv_config_vf_t *vfcfg_p)
```

各表記の意味は次のとおりです。

`dip` `dev_info` 構造体へのポインタ。

`vfcfg_p` `pciv_config_vf` 構造体へのポインタ。

```
typedef enum {
                PCIV_VFCFG_PARAM,
                PCIV_VF_ENABLE,
                        PCIV_VF_DISABLE
                PCIV_EVT_VFENABLE_PRE,
                PCIV_EVT_VFENABLE_POST,
                PCIV_EVT_VFDISABLE_PRE,
                PCIV_EVT_VFDISABLE_POST
            } pciv_vf_config_cmd_t;
```

`pciv_config_vf` 構造体には次のフィールドが含まれます。

```
typedef struct pciv_config_vf {
              int version;
               pciv_vf_config_cmd_t cmd;
               uint16_t num_vf;
               uint16_t first_vf_offset;
               uint16_t vf_stride;
               boolean_t ari_cap;
               uint32_t page_size;
    } pciv_config_vf_t;
```

各表記の意味は次のとおりです。

`version` バージョン番号。

`cmd` このインタフェースを構成情報の取得のために呼び出すのか、VF の接続のために呼び出すのかを示すために使用されます。

- `PCIV_VFCFG_PARAM` – 構成情報を取得します
- `PCIV_VF_ENABLE` – VF を有効にします
  - `PCIV_EVT_VFENABLE_PRE`
  - `PCIV_EVT_VFDISABLE_PRE`
  - `PCIV_EVT_VFENABLE_POST`
  - `PCIV_EVT_VFDISABLE_POST`

`num_vf` バックエンドで定義される VF の数。

vf_stride	各 VF 間の距離。
first_vf_offset	最初の VF と PF とのオフセット。
ari_cap	ARI 可能な階層。
page_size	システムのページサイズを指定します。

ドライバはまず、構成情報を取得するために、cmd フィールドを PCIV_VFCFG_PARAM に設定して pciv_vfconfig() インタフェースを呼び出すべきです。ドライバは次に、VF を構成するために、cmd フィールドを PCIV_VF_ENABLE に設定してこのインタフェースを再度呼び出すべきです。

ドライバは次のいずれかのエラーコードを返すことができます。

DDI_SUCCESS

DDI_FAILURE

PCIV_REQRESET

PCIV_REQREATTACH

ドライバは、pciv_vf_config() インタフェースを呼び出して VF を有効化する前に、DDI_CB_FLAG_SRIOV ビットを設定してコールバックを登録しておく必要はありません。しかし、SR-IOV フレームワークによって VF が構成解除されたときに通知を受け取るためには、ドライバは、VF の有効化後に DDI_CB_FLAG_SRIOV ビットを設定してコールバックを登録する必要があります。追加情報については、533 ページの「ドライバのコールバック」を参照してください。

VF をサポートできる PF ドライバはすべて、その機能を PCIe フレームワークに知らせるために、flags 引数に DDI_CB_FLAG_SRIOV フラグを設定して ddi_cb_register(9F) を呼び出すべきです。ddi_cb_register() 関数は、ドライバの接続ルーチン内で呼び出す必要があります。PF デバイスドライバがその接続ルーチン内で pciv_vf_config() 関数を呼び出して VF を有効化する場合、PF ドライバは VF の有効化後に ddi_cb_register() 関数を呼び出すべきです。

フレームワークは、次のアクションを実行する際に DDI_CB_FLAG_SRIOV フラグを必要とします。

- VF をサポートできるデバイスドライバが存在することを Sparc OVM エージェントに示します。すると、Sparc OVM エージェントは VF デバイスの作成を許可します。この機能がないと、ユーザーは SPARC プラットフォーム上で VF を作成できません。
- フレームワークは、VF を無効にする前とあとで、PF ドライバに対してコールバックを発行します。これにより、PF ドライバは VF サポートのための内部管理作業を容易に行えるようになります。

pci_plist_lookup() インタフェース

ドライバで pci_plist_lookup(9F) インタフェースを使用すると、サポートされているさまざまなデータ型の名前-値ペアを検索できます。この関数は、名前とインタフェース名で示された型に一致する nvpair (名前-値ペア) を検索します。見つかった場合、*nelem* と *val* が変更され、値の要素数とデータの開始アドレスがそれぞれ格納されます。

pci_plist_lookup() インタフェースでは次のデータ型がサポートされています。

- int pci_plist_lookup_int8(pci_plist_t *plist* , const char **name*,int8_t * *val*)
- int pci_plist_lookup_uint8(pci_plist_t *plist* , const char **name*,uint8_t * *val*)
- int pci_plist_lookup_int16(pci_plist_t *plist* , const char **name*,int16_t * *val*)
- int pci_plist_lookup_uint16(pci_plist_t *plist* , const char **name*, uint16_t * *val*)
- int pci_plist_lookup_int32(pci_plist_t *plist* , const char **name*,int32_t * *val*)
- int pci_plist_lookup_uint32(pci_plist_t *plist* , const char **name*, uint32_t * *val*)
- int pci_plist_lookup_int64(pci_plist_t *plist* , const char **name*,int64_t * *val*)
- int pci_plist_lookup_uint64(pci_plist_t *plist* , const char **name*, uint64_t * *val*)
- int pci_plist_lookup_string(pci_plist_t *plist* , const char **name*, char ** *val*)
- int pci_plist_lookup_plist(pci_plist_t *plist* , const char **name*, pci_plist_t * * *val*)
- int pci_plist_lookup_int8_array(pci_plist_t *plist*, const char **name*,int8_t * *val*, uint_t **nelem*)
- int pci_plist_lookup_uint8_array(pci_plist_t *plist*, const char **name*, int8_t * *val*, uint_t **nelem*)
- int pci_plist_lookup_int16_array(pci_plist_t *plist*, const char **name*,int16_t * *val*, uint_t **nelem*)
- int pci_plist_lookup_uint16_array(pci_plist_t *plist*, const char **name*, uint16_t **val*, uint_t **nelem*)
- int pci_plist_lookup_int32_array(pci_plist_t *plist*, const char **name*,int32_t * *val*, uint_t **nelem*)
- int pci_plist_lookup_uint32_array(pci_plist_t *plist*, const char **name*,uint32_t **val*, uint_t **nelem*)
- int pci_plist_lookup_int64_array(pci_plist_t *plist*, const char **name*, int64_t **val*, uint_t **nelem*)

- `int pci_plist_lookup_uint64_array(pci_plist_t *plist*,const char *name*,uint64_t * *val*, uint_t *nelem*)`
- `int pci_plist_lookup_string_array(pci_plist_t *plist*, const char *name*,char ** *val*, uint_t *nelem*)`

各表記の意味は次のとおりです。

`plist` 処理対象となる `pci_plist_t` 構造体へのポインタ。

`name` 検索する名前-値ペアの名前。

`nelem` 値の要素数を格納するアドレス。

`val` データの開始アドレス。

`pci_plist_lookup()` 関数は、成功時は 0 を返し、失敗時はエラー値を返します。次のエラー値がサポートされています。

`DDI_EINVAL` 無効な引数

`ENOENT` 一致する名前-値ペアが見つかりません

`ENOTSUP` エンコードまたはデコードの方式がサポートされていません

`pci_param_free()` インタフェース

`param` ハンドルを使ってデバイスパラメータを取得したあと、ドライバによって `pci_param_free(9F)` インタフェースが呼び出される必要があります。この呼び出しにより、`pci_param_get()` および `pci_param_get_ioctl()` インタフェースによって割り当てられたリソースが解放されます。

```
int pci_param_free (pci_param_t param)
```

ここで *param* は、`pci_param_get()` または `pci_param_get_ioctl()` インタフェースから取得されたハンドルです。

`pciv_send()` インタフェース

`pciv_send(9F)` インタフェースは、SR-IOV に対応した PF および VF ドライバが互いに通信を行うために使用されます。PF ドライバはいずれの VF ドライバとも通信できますが、VF ドライバはそれ自身の PF ドライバとしか通信できません。

```
int pciv_send(dev_info_t *dip, pciv_pvp_req_t *req
```

各表記の意味は次のとおりです。

dip　dev_info 構造体へのポインタ。

req　pciv_pvp_req_t 構造体へのポインタ。

pciv_pvp_req_t の構造:

```
typedef struct pciv_pvp_req {
int pvp_dstfunc;
caddr_t pvp_buf;
size_t pvp_nbyte;
buf_cb_t pvp_cb;
caddr_t pvp_cb_arg;
uint_t pvp_flag;
} pciv_pvp_req_t;
```

各表記の意味は次のとおりです。

pvp_dstfunc　PF ドライバから呼び出された場合、VF インデックスの範囲は 1 から *num_vf* になります。呼び出し元が VF ドライバの場合、これは常に PCIV_PF にすべきです。

pvp_buf　送信対象となる呼び出し元のバッファーのバッファーアドレス。

pvp_nbyte　送信されるバイトの数 (8K 未満でなければならない)。

pvp_cb　*pvp_flag* が PCIV_NOWAIT として設定された場合のコールバック関数のポインタ。

pvp_flag を PCIV_NOWAIT に設定した場合、呼び出しがすぐに戻り、*pvp_buf* のデータが宛先に送信される前に *pvp_cb* のコールバックルーチンが呼び出されます。呼び出し元はその後、コールバックルーチン内でバッファーを解放することを許可されます。

```
typedef void (*buf_cb_t)(int rc, caddr_t buf, size_t size, caddr_t cb_arg);
```

各表記の意味は次のとおりです。

rc　送信用の DDI リターンコード。

buf　送信対象となる呼び出し元のバッファーのバッファーアドレス。

size　送信されるバイトの数。

cb_arg　呼び出し元がルーチンの呼び出し時に設定した入力引数。

pvp_cb_arg　*pvp_flag* が PCIV_NOWAIT として設定された場合の、*pvp_cb* のコールバック入力引数。

pvp_flag

- PCIV_NOWAIT – 受信側の応答を待ちません。

- PCIV_WAIT – これがデフォルトの状態です。受信側から送信の確認を受け取るまで待ちます。

pciv_send() インタフェースは次のいずれかの戻り値を返します。

DDI_SUCCESS	バッファーの送信が成功しました。
DDI_ENOTSUP	デバイスドライバがこの処理をサポートしていません。呼び出し元はハードウェアメールボックスなど、その他のメカニズムを使用できます。
DDI_EINVAL	*pvp_nbyte* または *pvp_dstfunc* が無効です。
DDI_ENOMEM	リソース不足により処理が失敗しました。
DDI_ETRANSPORT	着信した送信を処理するためのコールバックが、リモート側で登録されませんでした。
DDI_FAILURE	不明な理由により失敗しました。

SR-IOV ドライバの ioctl

SR-IOV ドライバの ioctl は、管理者が構成できるデバイス固有のパラメータを特定したり、特定の構成を適用する前にその構成を検証したりする場合に使用されます。次の各セクションでは、ioctl のデータ構造体やインタフェースについて説明します。

データ構造体

次の各セクションでは、ioctl を実装する前に PF ドライバで定義および初期化しておくべきデータ構造体を示します。

iov_param_ver_info 構造体

iov_param_ver_info 構造体は次のように定義されています。

```
#define IOV_IOCTL (('I' << 24) | ('O' << 16) | ('V' << 8))
#define IOV_GET_VER_INFO (IOV_IOCTL | 0)
#define IOV_GET_PARAM_INFO (IOV_IOCTL | 1)
#define IOV_VALIDATE_PARAM (IOV_IOCTL | 2)
#define IOV_PARAM_DESC_VERSION  1
```

iov_param_ver_info 構造体には次のフィールドが含まれています。

```
typedef struct iov_param_ver_info {
uint32_t version;
```

```
uint32_t num_params;
} iov_param_ver_info_t;
```

各表記の意味は次のとおりです。

`version` バージョン情報

`num_params` パラメータの数

`iov_param_validate` 構造体

`iov_param_validate` 構造体は次のように定義されています。

```
  #define IOV_IOCTL (('I' << 24) | ('O' << 16) | ('V' << 8))
  #define IOV_GET_VER_INFO (IOV_IOCTL | 0)
  #define IOV_GET_PARAM_INFO (IOV_IOCTL | 1)
  #define IOV_VALIDATE_PARAM  (IOV_IOCTL | 2)
  #define IOV_PARAM_DESC_VERSION  1
```

`iov_param_validate` には次のフィールドが含まれています。

```
typedef struct iov_param_validate {
     char pv_reason[MAX_REASON_LEN + 1];
     int32_t pv_buflen;
     /* encoded buffer containing params */
    char pv_buf[1];  /* size of buf is pv_buflen */
} iov_param_validate_t;
```

各表記の意味は次のとおりです。

`pv_reason` ioctl コールが失敗した場合の失敗理由を説明する ASCII 文字列。

`pv_buflen` バッファー *pv_buf* の長さ

`pv_buf` パラメータを含むバッファー

`iov_param_desc` 構造体

`iov_param_desc` 構造体は次のように定義されています。

```
     #define IOV_IOCTL  (('I' << 24) | ('O' << 16) | ('V' << 8))
  #define IOV_GET_VER_INFO (IOV_IOCTL | 0)
  #define IOV_GET_PARAM_INFO (IOV_IOCTL | 1)
  #define IOV_VALIDATE_PARAM (IOV_IOCTL | 2)
  #define IOV_PARAM_DESC_VERSION  1
```

構造体 `iov_param_desc` には次のフィールドが含まれています。

```
typedef struct iov_param_desc {
char pd_name[MAX_PARAM_NAME_SIZE];
char pd_desc[MAX_PARAM_DESC_SIZE];
int32_t pd_flag;   /* applicable for PF or VF or both */
int32_t pd_data_type; /* integer, string, plist */
```

```
/* Following 3 are applicable for integer data types */
uint64_t pd_default_value;
uint64_t pd_min64;
uint64_t pd_max64;
char pd_default_string [MAX_PARAM_DEFAULT_STRING_SIZE];
} iov_param_desc_t;
```

各表記の意味は次のとおりです。

`pd_name`	`ldm(1M)` コマンド内または `pci.conf` ファイル内で、パラメータに値を割り当てるために使用されます。
`pd_desc`	パラメータの簡易説明。
`pd_flag`	パラメータの適用対象が、PF のみ、VF のみ、PF と VF の両方のいずれであるかを示します。
`pd_default_value`	`ldm()` コマンドや `pci.conf` ファイルでパラメータが指定されていない場合にドライバによって割り当てられる値。
`pd_min64`	整数パラメータの値の最小範囲を指定します。
`pd_max64`	整数パラメータの値の最大範囲を指定します。
`pd_default_string`	パラメータが文字列の場合に使用されるデフォルト文字列を指定します。

IOV_GET_VER_INFO ioctl

`IOV_GET_VER_INFO IOCTL()` ioctl を実装する SR-IOV デバイスドライバは、`iov_param_ver_info` 構造体の *version* および *num_params* フィールドを設定し、その値を呼び出し元の関数に返すべきです。続いて呼び出し元の関数は、`IOV_GET_PARAM_INFO()` ioctl コールを使ってパラメータの記述を取得する際に必要となるバッファーのサイズを、*version* および *num_params* パラメータを使用して決定します。

IOV_GET_PARAM_INFO Ioctl

`IOV_GET_PARAM_INFO()` ioctl を呼び出すドライバの一般的な制御フローは次のようになります。

1. `iov_param_desc_t` 構造体でサポートされる構成可能パラメータのそれぞれの記述を含む、この構造体の配列を保持します。この構造体の説明については、iov_param_desc 構造体を参照してください。
2. `iov_param_desc_t` 構造体の配列をコピーして `arg` パラメータに出力します。`iov_param_desc_t` 構造体のフィールドは静的であり、コンパイル時に定義できます。

配列の要素数は、IOV_GET_VER_INFO() ioctl コールから返された num_params 値になります。バッファーのサイズは sizeof (iov_param_desc_t) * num_params になります

IOV_VALIDATE_PARAM ioctl

IOV_VALIDATE_PARAM() を呼び出すドライバの一般的な制御フローは次のようになります。

1. *arg* パラメータを pci_param_get_ioctl() インタフェースに送信し、*pci_param_t* 構造体へのポインタを取得します。
2. *param* 検証が失敗した場合は、*pv_reason* 配列に説明文字列を書き込みます。
3. pci_get_plist() インタフェースに続いて pci_plist_lookup() インタフェースを呼び出すことで、デバイスパラメータを取得します。
4. この構成の検証用として構成すべき VF の数を取得するために、PF plist 内で *vfs* 名前-値ペアを検索します。ドライバは、16 ビット以上の整数データ型を使って *vfs* 名前-値ペアの検索を行うべきです。pciv_plist_getvf() インタフェースを使って VF デバイスの *plist* パラメータを取得します。
5. デバイスにパラメータを実際に適用せずにパラメータを検証します。
6. 有効な構成が見つかった場合は 0 を返します。

注意 – 上の手順で検証されるパラメータは、デバイスの現在の構成とは一切関係ありません。それらは、将来の構成になる可能性があると仮定して、個別に検証する必要があります。そうでない場合、ドライバは不適切な構成を示す DDI_EINVAL を返すべきです。さらに、無効な構成が見つかった場合は、ドライバは iov_param_validate 構造体の *pv_reason* フィールドに説明文字列を提供すべきです。この文字列は、構成が失敗した理由を管理者に伝えます。

ドライバのコールバック

コールバックを登録するには、DDI インタフェース ddi_cb_register() と ddi_cb_unregister() を使用します。コールバックはイベント通知および着信データトラフィックに使用されます。コールバックは、送信時の各イベントのイベントハンドラとしての役割を果たします。

SR-IOV ドライバは、VF の構成または構成解除の前後に PF ドライバに通知するための追加コールバックを実装すべきです。

ドライバでコールバックを実装するには、次の DDI コールバック登録メカニズムインタフェースを使用します。

-

  ```
  int ddi_cb_register(dev_info_t *dip, ddi_cb_flags_t flags, ddi_cb_func_t cbfunc,
  void *arg1, void *arg2,ddi_cb_handle_t *ret_hdlp)
  ```

  詳細は、ddi_cb_register(9F) のマニュアルページを参照してください。

-

  ```
  typedef int(*ddi_cb_func_t)
  (dev_info_t *dip, ddi_cb_action_t action,
  void *cbarg, void *arg1, void *arg2)
  ```

  各表記の意味は次のとおりです。

  dip　　dev_info 構造体へのポインタです

  cbarg　　構造体 pciv_config_vf_t へのポインタです。

  action　　VF 構成への変更に関する通知を受け取るには、DDI_CB_PCIV_CONFIG_VF として設定します。

  arg1　　cbfunc() ルーチンが実行されるたびに自身に送信される非公開引数。

  arg2　　cbfunc() ルーチンが実行されるたびに自身に送信される非公開引数。

  追加情報については、143 ページの「コールバックハンドラ関数の登録」を参照してください。

注 – SR-IOV に対応している PF ドライバは必ず、ddi_cb_flags_t DDI_CB_FLAG_SRIOV を使用することで、PF ドライバが SR-IOV に対応していることを Oracle Solaris IOV フレームワークに知らせる必要があります。

ドライバ ioctl のサンプルコード

```
enum ioc_reply
igb_ioctl(igb_t *igb, struct iocblk *iocp, mblk_t *mp)
{
        int rval = 0;
        iov_param_ver_info_t *iov_param_ver;
        iov_param_validate_t pvalidate;
        pci_param_t my_params;
        char reason[81];

if (mp->b_cont == NULL)
        return (IOC_INVAL);
if ((int)iocp->ioc_count < 0)
        return (IOC_INVAL);
switch (iocp->ioc_cmd) {
        case IOV_GET_PARAM_VER_INFO:
            if (iocp->ioc_count < sizeof (iov_param_ver_info_t))
                return (IOC_INVAL);
```

```
            iov_param_ver = (iov_param_ver_info_t *)(mp->b_cont->b_rptr);
            iov_param_ver->version = IOV_PARAM_DESC_VERSION;
            iov_param_ver->num_params = NUM_OF_PARAMS;
        return (IOC_REPLY);
        case IOV_GET_PARAM_INFO:
            if (iocp->ioc_count < sizeof (pci_list))
                return (IOC_INVAL);
            memcpy((caddr_t)(mp->b_cont->b_rptr), &pci_list,sizeof (pci_list));
            return (IOC_REPLY);
        case IOV_VALIDATE_PARAM:
            if (iocp->ioc_count <= 0)
                return (IOC_INVAL);
            strcpy(reason, "Failed to read params sent\n");
            rval = pci_param_get_ioctl(igb->dip,(uintptr_t)(mp->b_cont->b_rptr),
                    iocp->ioc_flag | FKIOCTL,&my_params );
            if (rval == 0) {
                rval = validate_params(igb->dip, my_params, reason);
            pci_param_free(my_params);
    }
if (rval) {
        memcpy(mp->b_cont->b_rptr, reason, sizeof (reason));
        iocp->ioc_count = sizeof (reason);
        return (IOC_REPLY);
}
iocp->ioc_count = 0;
return (IOC_REPLY);
iov_param_ver_info_t iov_param_ver;
iov_param_validate_t pvalidate;
pci_param_t    my_params;

switch (cmd) {
case IOV_GET_PARAM_VER_INFO:
        iov_param_ver.version = IOV_PARAM_DESC_VERSION;
        iov_param_ver.num_params = NUM_OF_PARAMS;
        if (ddi_copyout(&iov_param_ver, (caddr_t)arg,
            sizeof (iov_param_ver_info_t), mode) != DDI_SUCCESS)
        return (DEFAULT);
        return (0);
case IOV_GET_PARAM_INFO:
        if (ddi_copyout(&pci_list, (caddr_t)arg,param_list_size, mode) != DDI_SUCCESS)
        return (DEFAULT);
        return (0);
case IOV_VALIDATE_PARAM:
        strcpy(reason, "Failed to read params sent\n");
        rv = pci_param_get_ioctl(state->dip, arg, mode, &my_params);
        if (rv == 0)
            rv = validate_params(state->dip, my_params, reason);
        else
            return (rv);
        pci_param_free(my_params);
    if (rv) {
        if (ddi_copyout(reason,iov_param_validate_t *)arg)->pv_reason,
             sizeof (reason), mode) != DDI_SUCCESS)
        return (DEFAULT);
 return (rv);
    }
return (0);
```

パート III

デバイスドライバの構築

このドキュメントの第 3 部では、Oracle Solaris オペレーティングシステム用デバイスドライバの構築に関するアドバイスを提供します。

- 第 22 章「ドライバのコンパイル、ロード、パッケージ化、およびテスト」では、ドライバのコンパイル、リンク、およびインストールに関する情報を提供します。
- 第 23 章「デバイスドライバのデバッグ、テスト、およびチューニング」では、ドライバのデバッグ、テスト、およびチューニングを行うための手法について説明します。
- 第 24 章「推奨されるコーティング方法」では、ドライバを記述するための推奨のコーディング手法について説明します。

◆ ◆ ◆ 第 22 章

22 ドライバのコンパイル、ロード、パッケージ化、およびテスト

この章では、コードの配置、コンパイル、パッケージ化、テストを含むドライバ開発の手順について説明します。

この章では、次の内容について説明します。

ドライバ開発のサマリー

この章と、第 23 章「デバイスドライバのデバッグ、テスト、およびチューニング」および第 24 章「推奨されるコーティング方法」の 2 つの章では、デバイスドライバの開発について詳しく説明します。

デバイスドライバを構築するには、次の手順に従います。

1. 新しいコードを記述、コンパイル、リンクします。

   ファイルの命名規則については、540 ページの「ドライバコードの配置」を参照してください。C コンパイラを使用してドライバをコンパイルします。`ld(1)` を使用してドライバをリンクします。543 ページの「ドライバのコンパイルとリンク」と544 ページの「モジュールの依存関係」を参照してください。

2. 必要なハードウェア構成ファイルを作成します。

   デバイスに固有の、*xx* `.conf` という名前のハードウェア構成ファイルを作成します。*xx* にはデバイスの接頭辞が入ります。このファイルは `driver.conf(4)` ファイルを更新するために使用されます。545 ページの「ハードウェア構成ファイルの記述」を参照してください。擬似デバイスドライバの場合は、`pseudo(4)` ファイルを作成します。

3. 適切なモジュールディレクトリにドライバをコピーします。

   545 ページの「モジュールディレクトリへのドライバのコピー」を参照してください。

4. add_drv(1M) を使用してデバイスドライバをインストールします。

   add_drv を使用したドライバのインストールは、通常はインストール後スクリプトの一部として実行されます。547 ページの「add_drv を使用したドライバのインストール」を参照してください。update_drv(1M) コマンドを使用して、必要な変更をドライバに加えます。547 ページの「ドライバ情報の更新」を参照してください。

5. ドライバをロードします。

   デバイスにアクセスすることでドライバを自動的にロードできます。548 ページの「ドライバのロードとアンロード」と548 ページの「パッケージのインストール後処理」を参照してください。ドライバは modload(1M) コマンドを使用してもロードできます。modload コマンドはモジュール内のどのルーチンも実行しないので、テストの際に役立ちます。562 ページの「テストモジュールのロードとアンロード」を参照してください。

6. ドライバをテストします。

   ドライバは次の領域で厳しくテストする必要があります。

   - 550 ページの「構成のテスト」
   - 551 ページの「機能テスト」
   - 551 ページの「エラー処理」
   - 552 ページの「ロードとアンロードのテスト」
   - 552 ページの「ストレス、パフォーマンス、および相互運用性のテスト」
   - 553 ページの「DDI/DKI コンプライアンスのテスト」
   - 553 ページの「インストールとパッケージ化のテスト」

   ドライバ固有のその他のテストについては、554 ページの「特定の種類のドライバのテスト」を参照してください。

7. 必要に応じてドライバを削除します。

   rem_drv(1M) コマンドを使用してデバイスドライバを削除します。547 ページの「ドライバの削除」と549 ページの「パッケージの削除前処理」を参照してください。

ドライバコードの配置

デバイスドライバのコードは通常、次のファイルに分割されます。

- ヘッダーファイル (.h ファイル)
- ソースファイル (.c ファイル)
- オプション構成ファイル (driver.conf ファイル)

ヘッダーファイル

ヘッダーファイルには次の定義が記述されています。

- デバイス固有のデータ構造体 (デバイスレジスタを表す構造など)
- ドライバによって定義された、状態情報を維持管理するためのデータ構造体
- 定義済みの定数 (デバイスレジスタのビットを表す定数など)
- マクロ (マイナーデバイス番号とインスタンス番号の間の静的マッピングを定義するマクロなど)

状態構造体など、ヘッダーファイルの定義の一部は、デバイスドライバでのみ必要になることがあります。この情報はデバイスドライバ自体によってのみ含められる *private* ヘッダーファイルに入れる必要があります。

入出力制御コマンドなど、アプリケーションが必要とすることがある情報はすべて、*public* ヘッダーファイルに含める必要があります。これらのファイルはドライバと、デバイスに関する情報を必要とするすべてのアプリケーションによって含められます。

private ファイルと public ファイルの命名に関する標準はありませんが、private ヘッダーファイルの場合は *xx*`impl.h` が、public ヘッダーファイルの場合は *xx*`io.h` が命名規則の 1 つです。

ソースファイル

デバイスドライバの C ソースファイル (`.c` ファイル) は次のことを行います。

- データ宣言と、ドライバのエントリポイントのコードを格納します。
- ドライバに必要な `#include` 文を格納します。
- `extern` 参照を宣言します。
- ローカルデータを宣言します。
- `cb_ops` 構造体と `dev_ops` 構造体を設定します。
- モジュール構成の選択セクション、つまり `modlinkage(9S)` 構造体と `modldrv(9S)` 構造体を宣言して初期化します。
- その他に必要なすべての宣言を行います。
- ドライバのエントリポイントを定義します。

構成ファイル

一般に、ドライバの構成ファイルでは、ドライバが必要とするすべてのプロパティーが定義されます。ドライバ構成ファイルのエントリは、存在するかどうかをドライバが調べる可能性があるデバイスインスタンスを指定します。ドライバのグローバルプロパティーはドライバの設定ファイル内で設定できます。詳細については、driver.conf(4) のマニュアルページを参照してください。

ドライバ構成ファイルは、自己識別型ではないデバイスのために必要です。

自己識別デバイス (SID) については、ドライバ構成ファイルをオプションです。自己識別デバイスの場合、SID ノードにプロパティーを追加するために構成ファイルを使用できます。

次のプロパティーは、ドライバ設定ファイルで設定されないプロパティーの例です。

- 周辺機器用の SBus バスを使用するドライバは通常、SBus カードからプロパティー情報を取得します。追加のプロパティーが必要な場合、sbus(4) で定義されているプロパティーをドライバ構成ファイルに含めることができます。
- PCI バスのプロパティーは通常、PCI 構成スペースから派生できます。private ドライバのプロパティーが必要な場合は、pci(4) で定義されているプロパティーをドライバ構成ファイルに含めることができます。
- ISA バス上のドライバは、isa(4) で定義されている追加のプロパティーを使用できます。

ドライバインストールの準備

ドライバのインストールの前に次の手順があります。

1. ドライバをコンパイルします。
2. 必要に応じて構成ファイルを作成します。
3. 次のいずれかの方法で、システムでドライバモジュールが識別されるようにします。
   - ドライバの名前を、デバイスノードの名前と一致させます。
   - add_drv(1M) または update_drv(1M) を使用して、システムにモジュール名を通知します。

システムは、ドライバモジュールの名前と dev_info ノードの名前の間に 1 対 1 の関連付けを維持します。たとえば、mydevice という名前のデバイスの dev_info ノードについて考えます。デバイス mydevice は、同様に mydevice という名前のドライバモジュールによって処理されます。mydevice モジュールは、モジュールのパス内にある drv という名前のサブディレクトリに置かれます。32 ビットカーネルを使用して

いる場合、モジュールは drv/mydevice 内にあります。64 ビット SPARC カーネルを使用している場合、モジュールは drv/sparcv9/mydevice 内にあります。64 ビット x86 カーネルを使用している場合、モジュールは drv/amd64/mydevice 内にあります。

STREAMS ネットワークドライバの場合、ドライバ名が次の制約を満たしている必要があります。

- 英数字 (a-z、A-Z、0-9) と下線 ('_') のみを使用できます。
- 名前の最初と最後の文字はどちらも数字にすることができません。
- 名前の長さは 16 文字を超えることはできません。推奨される名前の長さは 3 文字から 8 文字の範囲です。

ドライバで、異なる名前を持つ dev_info ノードを管理する必要がある場合は、add_drv(1M) ユーティリティーで別名を作成できます。-i フラグでは、ドライバで処理するほかの dev_info ノードの名前を指定します。update_drv コマンドでも、インストールされているデバイスドライバの別名を変更できます。

ドライバのコンパイルとリンク

ドライバのソースファイルをそれぞれコンパイルし、得られたオブジェクトファイルをドライバモジュール内にリンクする必要があります。Oracle Solaris OS は Oracle Solaris Studio C コンパイラ、および Free Software Foundation, Inc. の GNU C コンパイラの両方と互換性があります。このセクションの例では、特に記載している場合を除いて、Oracle Solaris Studio C コンパイラを使用しています。Oracle Solaris Studio C コンパイラについては、『Oracle Solaris Studio 12.3: C ユーザーガイド』および Oracle Technology Network Web サイトにある Oracle Solaris Studio のドキュメントを参照してください。コンパイルとリンクのオプションの詳細については、Oracle Solaris Studio のマニュアルページ (http://docs.oracle.com/cd/E24457_01/html/E22003/index.html) を参照してください。GNU C コンパイラは /usr/sfw ディレクトリにあります。GNU C コンパイラについては、http://gcc.gnu.org/ を参照してください。

下の例は、2 つの C ソースファイルを持つ *xx* という名前のドライバを示しています。*xx* という名前のドライバモジュールが生成されます。この例で作成されるドライバは 32 ビットカーネル用です。ドライバが持つオブジェクトモジュールが 1 つだけであっても ld -r を使用する必要があります。

```
% cc -D_KERNEL -c xx1.c
% cc -D_KERNEL -c xx2.c
% ld -r -o xx xx1.o xx2.o
```

このコードがカーネルモジュールを定義していることを示すため、_KERNEL シンボルが定義されている必要があります。ドライバの private シンボルを除き、ほかのシンボルが定義されていてはいけません。DEBUG シンボルを定義すると、ASSERT(9F) に対するすべての呼び出しを有効にすることができます。

Sun Studio 9、Sun Studio 10、または Sun Studio 11 を使用して 64 ビット SPARC アーキテクチャー向けにコンパイルする場合は、`-xarch=v9` オプションを使用します。

```
% cc -D_KERNEL -xarch=v9 -c xx.c
```

Sun Studio 12 を使用して 64 ビット SPARC アーキテクチャー向けにコンパイルする場合は、`-m64` オプションを使用します。

```
% cc -D_KERNEL -m64 -c xx.c
```

Sun Studio 10 または Sun Studio 11 を使用して 64 ビット x86 アーキテクチャー向けにコンパイルする場合は、`-xarch=amd64` オプションと `-xmodel=kernel` オプションの両方を使用します。

```
% cc -D_KERNEL -xarch=amd64 -xmodel=kernel -c xx.c
```

Oracle Solaris Studio 12 を使用して 64 ビット x86 アーキテクチャー向けにコンパイルする場合は、`-m64` オプション、`-xarch=sse2a` オプション、および `-xmodel=kernel` オプションを使用します。

```
% cc -D_KERNEL -m64 -xarch=sse2a -xmodel=kernel -c xx.c
```

注 – Sun Studio 9 は 64 ビット x86 アーキテクチャーをサポートしていません。64 ビット x86 アーキテクチャー用ドライバのコンパイルとデバッグを行うには、Sun Studio 10、Sun Studio 11、または Oracle Solaris Studio 12 を使用します。

ドライバが安定したら、最適化フラグを追加して本稼働品質のドライバをビルドできます。Oracle Solaris Studio C コンパイラでの最適化に関する具体的な情報については、Oracle Solaris Studio 12.3 のコマンド行リファレンスマニュアルにある cc(1) のマニュアルページを参照してください。

グローバル変数は、デバイスドライバ内では `volatile` として処理する必要があります。`volatile` タグについては、592 ページの「変数の volatile 宣言」で詳細に説明します。このフラグの用途はプラットフォームに応じて異なります。マニュアルページを参照してください。

モジュールの依存関係

ドライバモジュールが別のカーネルモジュールによってエクスポートされたシンボルに依存している場合、ローダー `ld(1)` の `-dy` および `-N` オプションによって依存関係を指定できます。ドライバが `misc/mySymbol` によってエクスポートされたシンボルに依存している場合、下の例を使用してドライババイナリを作成する必要があります。

```
% ld -dy -r -o xx xx1.o xx2.o -N misc/mySymbol
```

ハードウェア構成ファイルの記述

デバイスが自己識別型でない場合、カーネルのために、そのデバイスのハードウェア構成ファイルが必要です。ドライバの名前が *xx* の場合、ドライバのハードウェア構成ファイルの名前は *xx* `.conf` にする必要があります。ハードウェア構成ファイルの詳細については、`driver.conf`(4)、`pseudo`(4)、`sbus`(4)、`scsi_free_consistent_buf`(9F)、および `update_drv`(1M) のマニュアルページを参照してください。

ハードウェア構成ファイルでは任意のプロパティーを定義できます。構成ファイル内のエントリは、*property*= *value* という形式です。*property* はプロパティー名で、*value* はプロパティーの初期値です。構成ファイルを使うアプローチでは、プロパティー値を変更することでデバイスの構成が可能です。

ドライバのインストール、更新、および削除

ドライバを使用する前に、ドライバが存在することをシステムに通知する必要があります。`add_drv`(1M) ユーティリティーを使用して、デバイスドライバを正しくインストールする必要があります。ドライバがインストールされたら、`add_drv` コマンドを使用せずに、メモリーからそのドライバをロードおよびアンロードできます。

モジュールディレクトリへのドライバのコピー

デバイスドライバモジュールのパスは次の 3 つの条件によって決まります。

- ドライバが実行されるプラットフォーム
- ドライバがコンパイルされる対象のアーキテクチャー
- ブート時にパスが必要かどうか

デバイスドライバは次の場所に置かれます。

`/platform/‘uname -i‘/kernel/drv`
: 特定のプラットフォームでのみ実行される 32 ビットドライバが格納されます。

`/platform/‘uname -i‘/kernel/drv/sparcv9`
: 特定の SPARC ベースのプラットフォームでのみ実行される 64 ビットドライバが格納されます。

`/platform/‘uname -i‘/kernel/drv/amd64`
: 特定の x86 ベースのプラットフォームでのみ実行される 64 ビットドライバが格納されます。

`/platform/‘uname -m‘/kernel/drv`
: 特定ファミリのプラットフォームでのみ実行される 32 ビットドライバが格納されます。

`/platform/‘uname -m‘/kernel/drv/sparcv9`
特定ファミリの SPARC ベースのプラットフォームでのみ実行される 64 ビットドライバが格納されます。

`/platform/‘uname -m‘/kernel/drv/amd64`
特定ファミリの x86 ベースのプラットフォームでのみ実行される 64 ビットドライバが格納されます。

`/usr/kernel/drv`
プラットフォームに依存しない 32 ビットドライバが格納されます。

`/usr/kernel/drv/sparcv9`
プラットフォームに依存しない、SPARC ベースのシステムの 64 ビットドライバが格納されます。

`/usr/kernel/drv/amd64`
プラットフォームに依存しない、x86 ベースのシステムの 64 ビットドライバが格納されます。

32 ビットドライバをインストールするには、モジュールパスの `drv` ディレクトリにドライバと構成ファイルをコピーする必要があります。たとえば、ドライバを `/usr/kernel/drv` にコピーするには、次のように入力します。

```
$ su
# cp xx /usr/kernel/drv
# cp xx.conf /usr/kernel/drv
```

SPARC ドライバをインストールするには、モジュールパスの `drv/sparcv9` ディレクトリにドライバをコピーします。ドライバ構成ファイルは、モジュールパスの `drv` ディレクトリにコピーします。たとえば、ドライバを `/usr/kernel/drv` にコピーするには、次のように入力します。

```
$ su
# cp xx /usr/kernel/drv/sparcv9
# cp xx.conf /usr/kernel/drv
```

64 ビット x86 ドライバをインストールするには、モジュールパスの `drv/amd64` ディレクトリにドライバをコピーします。ドライバ構成ファイルは、モジュールパスの `drv` ディレクトリにコピーします。たとえば、ドライバを `/usr/kernel/drv` にコピーするには、次のように入力します。

```
$ su
# cp xx /usr/kernel/drv/amd64
# cp xx.conf /usr/kernel/drv
```

注 – すべてのドライバ構成ファイル (.conf ファイル) はモジュールパスの drv ディレクトリに入れる必要があります。.conf ファイルを drv ディレクトリのサブディレクトリ内に入れることはできません。

add_drv を使用したドライバのインストール

ドライバをシステムにインストールするには、add_drv(1M) コマンドを使用します。ドライバが正しくインストールされると、 add_drv が devfsadm(1M) を実行し、/dev ディレクトリ内に論理名が作成されます。

```
# add_drv xx
```

この場合、デバイスが自身を *xx* だと識別します。デバイスの特殊ファイルにはデフォルトの所有権とアクセス権があります (0600 root sys)。add_drv コマンドでは、デバイスの追加の名前 (別名) を指定することもできます。別名を追加すること、およびファイルのアクセス許可を明示的に設定することについては、add_drv(1M) のマニュアルページを参照してください。

注 – add_drv コマンドを使用して STREAMS モジュールをインストールしないでください。詳細については、『STREAMS Programming Guide』を参照してください。

ドライバが、ディスク、テープ、ポートなどの端末デバイスを表さないマイナーノードを作成する場合、devfsadm が /dev 内に論理デバイス名を作成するように /etc/devlink.tab を変更できます。別の方法として、ドライバのインストール時に実行されるプログラムで論理名を作成できます。

ドライバ情報の更新

インストールされているデバイスドライバに対する変更をシステムに通知するには、update_drv(1M) コマンドを使用します。デフォルトでは、システムはドライバ構成ファイルを再度読み込み、ドライバのバイナリモジュールを再ロードます。

ドライバの削除

ドライバをシステムから削除するには、rem_drv(1M) コマンドを使用し、次にモジュールパスから、ドライバモジュールと構成ファイルを削除します。add_drv(1M) を使用してドライバを再インストールするまで、ドライバを再使用することはできません。SCSI HBA ドライバの削除を有効にするにはリブートする必要があります。

ドライバのロードとアンロード

デバイスドライバに関連付けられている特殊ファイルを開く (デバイスにアクセスする) と、そのドライバがロードされます。modload(1M) コマンドを使用するとドライバをメモリーにロードできますが、モジュール内のルーチンは modload では呼び出されません。推奨されるのはデバイスを開く方法です。

通常、デバイスが使用されなくなると、システムがデバイスドライバを自動的にアンロードします。開発時には、modunload(1M) を使用してドライバを明示的にアンロードする場合があります。modunload が正しく機能するには、デバイスドライバがアクティブでない必要があります。open(2) や mmap(2) などを通して、デバイスに対する未処理の参照が存在していてはいけません。

modunload コマンドは引数として、ランタイム依存の module_id を受け取ります。module_id を見つけるには、grep を使用して、modinfo(1M) の出力で目的のドライバ名を検索します。最初の列を確認します。

```
# modunload -i module-id
```

現在ロードできないモジュールをすべてアンロードするには、モジュール ID として 0 を指定します。

```
# modunload -i 0
```

modunload(1M) が成功するには、ドライバがアクティブでないことに加えて、ドライバで detach(9E) ルーチンと _fini(9E) ルーチンが動作している必要があります。

ドライバのパッケージ化

ソフトウェアの通常の提供手段は、すべてのソフトウェアコンポーネントを含むパッケージを作成することです。パッケージは、ソフトウェア製品のすべてのコンポーネントをインストールおよび削除するための制御されたメカニズムを提供します。パッケージには、製品を使用するためのファイルに加えて、アプリケーションのインストールとアンインストールを行うための制御ファイルが含まれています。インストール後と削除前の 2 つのインストールスクリプトは、そうした制御ファイルです。

パッケージのインストール後処理

ドライババイナリとともにパッケージがシステムにインストールされたあとは、add_drv(1M) コマンドを実行する必要があります。add_drv コマンドはドライバのインストールを完了します。通常、add_drv は次の例のように、インストール後スクリプトで実行されます。

```
#!/bin/sh
#
#       @(#)postinstall 1.1

PATH="/usr/bin:/usr/sbin:${PATH}"
export PATH

#
# Driver info
#
DRV=<driver-name>
DRVALIAS="<company-name>,<driver-name>"
DRVPERM='* 0666 root sys'

ADD_DRV=/usr/sbin/add_drv

#
# Select the correct add_drv options to execute.
# add_drv touches /reconfigure to cause the
# next boot to be a reconfigure boot.
#
if [ "${BASEDIR}" = "/" ]; then
    #
    # On a running system, modify the
    # system files and attach the driver
    #
    ADD_DRV_FLAGS=""
else
    #
    # On a client, modify the system files
    # relative to BASEDIR
    #
    ADD_DRV_FLAGS="-b ${BASEDIR}"
fi

#
# Make sure add_drv has not been previously executed
# before attempting to add the driver.
#
grep "^${DRV} " $BASEDIR/etc/name_to_major > /dev/null 2>&1
if [ $? -ne 0 ]; then
    ${ADD_DRV} ${ADD_DRV_FLAGS} -m "${DRVPERM}" -i "${DRVALIAS}" ${DRV}
    if [ $? -ne 0 ]; then
        echo "postinstall: add_drv $DRV failed\n" >&2
        exit 1
    fi
fi
exit 0
```

パッケージの削除前処理

ドライバが含まれるパッケージを削除するときには、ドライバのバイナリとその他のコンポーネントを削除する前に、rem_drv(1M) コマンドを実行する必要があります。次の例では、ドライバの削除のために rem_drv コマンドを使用する削除前スクリプトを示します。

```
#!/bin/sh
#
#       @(#)preremove  1.1

PATH="/usr/bin:/usr/sbin:${PATH}"
export PATH

#
# Driver info
#
DRV=<driver-name>
REM_DRV=/usr/sbin/rem_drv

#
# Select the correct rem_drv options to execute.
# rem_drv touches /reconfigure to cause the
# next boot to be a reconfigure boot.
#
if [ "${BASEDIR}" = "/" ]; then
    #
    # On a running system, modify the
    # system files and remove the driver
    #
    REM_DRV_FLAGS=""
else
    #
    # On a client, modify the system files
    # relative to BASEDIR
    #
    REM_DRV_FLAGS="-b ${BASEDIR}"
fi

${REM_DRV} ${REM_DRV_FLAGS} ${DRV}

exit 0
```

デバイステストの条件

デバイスドライバが機能するようになったら、配布前にそのドライバを完全にテストする必要があります。従来の UNIX デバイスドライバに備わる機能をテストするほかに、Oracle Solaris ドライバでは、ドライバの動的なロードとアンロードなどの電源管理機能もテストする必要があります。

構成のテスト

複数のデバイス構成を扱うドライバの機能は、テストプロセスの重要な部分です。ドライバが単純な構成やデフォルトの構成で機能するようになったら、追加の構成をテストする必要があります。デバイスによっては、ジャンパーまたは DIP スイッチを変更することで構成のテストを完了できます。可能な構成の数が少ない場合は、すべての構成を試してください。数が多い場合は、可能な構成から成るさまざまなクラスを定義し、各クラスからサンプリングした構成をテストする必要があ

ります。これらのクラスの定義は、異なる構成パラメータ間でどれだけの対話が可能かに左右されます。これらの対話はデバイスの種類と、ドライバが記述された方法との相関関係です。

デバイス構成ごとに、基本の機能をテストする必要があります。それには、デバイスをロードする、開く、読み取る、書き込む、閉じる、アンロードする機能が含まれます。構成に依存するすべての機能には特に注意を払う必要があります。たとえば、デバイスレジスタのベースメモリーアドレスを変更することは、おそらくほとんどのドライバ機能の動作には影響を与えません。ドライバが 1 つのアドレスで正常に機能する場合、そのドライバはおそらく、異なるアドレスでも機能します。一方、特別な入出力制御呼び出しでは、特定のデバイス構成に応じて効果が異なる可能性があります。

さまざまな構成でドライバをロードすることで、probe(9E) および attach(9E) のエントリポイントが異なるアドレスで確実にデバイスを見つけることができます。基本機能のテストの場合、文字デバイスについては、cat(1) や dd(1M) などの通常の UNIX コマンドを使えば通常は十分です。ブロックデバイスについては、マウントやブートが必要な場合があります。

機能テスト

ドライバの構成を完全にテストしたら、ドライバのすべての機能を完全にテストする必要があります。これらのテストでは、ドライバのすべてのエントリポイントの操作を実行する必要があります。

多くのドライバには、機能をテストするためのカスタムアプリケーションが必要です。ただし、ディスク、テープ、非同期ボードなどのデバイスの基本ドライバは、標準のシステムユーティリティーを使用するとテストできます。このプロセスでは、該当する場合は devmap(9E)、chpoll(9E)、ioctl(9E) を含むすべてのエントリポイントをテストする必要があります。ioctl() テストは、ドライバごとに大きく異なる可能性があります。標準でないデバイスでは、通常はカスタムテストアプリケーションが必要です。

エラー処理

ドライバは理想的な環境では正常に実行されても、間違った操作や正しくないデータなどのエラーがある場合は失敗することがあります。そのため、ドライバのエラー処理のテストが、ドライバのテストの重要な一部となっています。

実際のハードウェアの誤動作に関するエラーの条件を含めて、ドライバの考えられるすべてのエラー条件を与える必要があります。一部のハードウェアエラー条件は発生させるのが困難な場合がありますが、可能であれば、そうしたエラーを強制したりシミュレートしたりするように努める必要があります。これらの条件は、実地

ではすべて発生する可能性があります。ケーブルを抜いたりゆるめたり、ボードを取り外したり、エラーのあるユーザーアプリケーションコードを記述したりして、そうしたエラーパスをテストする必要があります。第 13 章「Oracle Solaris ドライバの強化」も参照してください。

注意 – テスト時には電気的に適切な対策をとるようにしてください。

ロードとアンロードのテスト

ロードまたはアンロードされないドライバは予定外の停止時間の原因となるため、ロートとアンロードは完全にテストしておく必要があります。

次の例のようなスクリプトで十分です。

```
#!/bin/sh
cd <location_of_driver>
while [ 1 ]
do
    modunload -i 'modinfo | grep " <driver_name> " | cut -cl-3' &
    modload <driver_name> &
done
```

ストレス、パフォーマンス、および相互運用性のテスト

ドライバのパフォーマンスを確保する助けになるように、ドライバには厳しいストレステストを適用する必要があります。For example, running single threads through a driver does not test locking logic or conditional variables that have to wait. いくつかのスレッドで同じコードを同時に実行させるには、複数のプロセスで同時にデバイスの操作を実行する必要があります。

同時テストを実行するための手法はドライバによってさまざまです。一部のドライバでは特殊なテストアプリケーションが必要ですが、その他のドライバでは、バックグラウンドでいくつかの UNIX コマンドを開始すれば十分です。どのようなテストが適切であるかは、各ドライバでロックと条件変数が使われている場所によって異なります。マルチプロセッサマシンでドライバをテストすると、シングルプロセッサマシンでテストする場合より問題が表面化する可能性が高くなります。

ドライバ間の相互運用性もテストする必要があります。これは特に、さまざまなデバイスが割り込みレベルを共有できるためです。可能であれば、1 つのドライバをテストしているときに、別のデバイスを同じ割り込みレベルで構成します。ストレス

テストを行うと、ドライバが自身の割り込みを正しく要求し、予測どおりに動作するかどうかを判断できます。ストレステストは両方のデバイスで一度に実行する必要があります。デバイスが割り込みレベルを共有していない場合でも、このテストは有用です。たとえば、ネットワークドライバのテスト時に、シリアル通信デバイスでエラーが発生している場合を考えてください。同じ問題が原因で、システムの残りの部分でも割り込みの待ち時間の問題が発生することもあります。

これらのストレステストを適用したドライバのパフォーマンスは、UNIX のパフォーマンス測定ツールを使用して測定してください。この種類のテストは簡単で、`time(1)` コマンドを、ストレステストで使用されるコマンドとともに使用するだけで実行できます。

DDI/DKI コンプライアンスのテスト

後のリリースとの互換性と、最新リリースに対する信頼できるサポートを保証するため、すべてのドライバを DDI/DKI 準拠にする必要があります。『SunOS リファレンスマニュアル 9 : DDI/DKI カーネル関数』および『man pages section 9: DDI and DKI Driver Entry Points 』のカーネルルーチンと、『man pages section 9: DDI and DKI Properties and Data Structures 』のデータ構造体のみが使用されていることを確認します。

インストールとパッケージ化のテスト

ドライバはお客様に、パッケージで提供されます。パッケージは標準のメカニズムを使用して、システムに対する追加または削除が可能です (『アプリケーションパッケージ開発者ガイド』を参照)。

ユーザーがシステムに対してパッケージの追加や削除を行う機能をテストする必要があります。テストでは、リリース時に使用されるすべての種類のメディアに対して、パッケージのインストールと削除の両方を行ってください。このテストには、いくつかのシステム構成を含める必要があります。パッケージが、インストール先システムのディレクトリ環境について、間違った想定をすることがあってはなりません。ただし、標準のカーネルファイルが保管されている場所については、有効な場所を想定することが可能です。また、開発環境用に変更されていない新しくインストールされたマシンで、パッケージの追加と削除をテストします。パッケージ化でよくあるエラーは、パッケージが開発でのみ使用されているツールやファイルに依存しているというものです。たとえば、ソース互換性パッケージ `SUNWscpu` のツールがドライバインストールプログラムで使用されていてはいけません。

ドライバのインストールは、オプションパッケージを一切追加していない最小限の Oracle Solaris システムでテストする必要があります。

特定の種類のドライバのテスト

このセクションでは、特定の種類の標準デバイスをテストする方法について、いくつかの提案を示します。

テープドライバ

テープドライバは、保管と復元のいくつかの操作を実行してテストする必要があります。この目的のために、cpio(1) コマンドと tar(1) コマンドを使用できます。ディスクパーティション全体をテープに書き込むには、dd(1M) コマンドを使用します。次に、データを読み戻して、同じサイズの別のパーティションにデータを書き込みます。その後、2 つのコピーを比較します。mt(1) コマンドは、テープドライバ固有の入出力制御の大半を実行できます。mtio(7I) のマニュアルページを参照してください。すべてのオプションを使用してみます。次の 3 つの手法で、テープドライバのエラー処理機能をテストできます。

- テープを取り出し、さまざまな操作を試みる
- テープを書き込み保護にして書き込みを試みる
- さまざまな操作の途中で電源をオフにする

テープドライバは、通常、排他的アクセスの open(9E) 呼び出しを実装しています。これらの open() 呼び出しは、デバイスを開き、2 番目のプロセスで同じデバイスを開こうとすることでテストできます。

ディスクドライバ

ディスクドライバは raw モードとブロックデバイスモードの両方でテストする必要があります。ブロックデバイステストの場合、デバイス上に新しいファイルシステムを作成します。次に、新しいファイルシステムのマウントを試みます。次に、複数ファイルの操作を実行してみます。

注 – ファイルシステムはページキャッシュを使用するので、同じファイルを繰り返し読み取ってもドライバを働かせることになりません。ページキャッシュは、mmap(2) を使用してファイルをメモリーマッピングすることで、デバイスからデータを取得するように設定できます。次に msync(3C) を使用して、メモリー内のコピーを無効にします。

同じサイズの別の (マウントされていない) パーティションを raw デバイスにコピーします。次に、fsck(1m) などのコマンドを使用して、コピーが正しいことを検証します。新しいパーティションもマウントし、あとで古いパーティションとファイル単位で比較できます。

非同期通信ドライバ

非同期ドライバは、シリアルポートへの login 用の回線を設けることで、基本レベルでテストできます。ユーザーがこの回線でログインできるかどうかを確認するのが適切なテストです。ただし非同期ドライバを十分にテストするには、割り込みを多数発生させて、すべての入出力制御関数を高速でテストする必要があります。ループバックシリアルケーブルを使用し、高速なデータ転送速度でテストを行うと、ドライバの信頼性を判断する助けになります。回線上で uucp(1C) を実行すると、一部のテストを行えます。ただし、uucp は独自のエラー処理を実行するため、ドライバが uucp プロセスに報告するエラーの数が多すぎないか検証してください。

これらの種類のデバイスは、通常、STREAMS ベースのデバイスです。詳細については、『STREAMS Programming Guide』を参照してください。

ネットワークドライバ

ネットワークドライバは標準のネットワークユーティリティーを使用するとテストできます。ネットワークのそれぞれの側でファイルを比較できるため、ftp(1) コマンドと rcp(1) コマンドが役立ちます。ドライバは高いネットワーク負荷をかけてテストする必要があるため、複数のプロセスで多様なコマンドを実行できます。

高いネットワーク負荷に該当するのは、次のような条件です。

- テストマシンへのトラフィックが非常に多い。
- ネットワーク上のすべてのマシン間のトラフィックが非常に多い。

テストの実行中にネットワークケーブルを抜き、発生したエラー状態から正常に回復することを確認する必要があります。別の重要なテストは、複数のパケットをドライバにたて続けに受信させることです。つまり、連続パケットを使用します。この場合、比較的高速なホストを負荷の少ないネットワーク上に置き、複数のパケットを間断なくテストマシンに送信します。受信側のドライバで、2 つ目以降のパケットが欠落していないことを確認します。

これらの種類のデバイスは、通常、STREAMS ベースのデバイスです。詳細については、『STREAMS Programming Guide』を参照してください。

SR-IOV ドライバのテスト

SR-IOV をサポートするドライバでは追加のテストを行う必要があります。標準の基本的なテストも必要となりますが、ネットワークデバイス用の ftp や rcp など、基本的なテストで使用されるユーティリティーを使用できます。

SR-IOV ドライバについては、第 21 章「SR-IOV ドライバ」を参照してください。

仮想機能 (VF) のステータスをテストするには、次のコマンドを使用します。

- 有効化された VF – `hotplug install`

- 無効化された VF – `hotplug uninstall`
- 割り当て済みの VF – `hotplug list`

SPARC システム上で VF のステータスをテストする場合、`hotplug` コマンドのほかに `ldm(1M)` コマンドも使用します。

また、さまざまな仮想化構成で SR-IOV デバイスをテストすることも重要です。SPARC と x86 の両プラットフォームで SR-IOV ドライバのテストを行う際は、次のオプションを試してください。

- 仮想機能 (VF) を一切構成しません
- VF を 1 つだけ構成します
- VF の最大数に達するまで、構成された VF を 2 の累乗ずつ増やします

SPARC プラットフォームでは、さまざまな数の IO ドメインを使用し、それらのドメイン内で VF をさまざまに分散させて、機能をテストします。次の構成を試してください。

- 単一の IO ドメインに単一の VF を割り当てます
- 単一の IO ドメインに 2 の累乗個の VF (最大数まで) を割り当てます
- 2、4、または 8 個の IO ドメインを作成し、それぞれのドメインにさまざまな数の VF を割り当てます
- ルートドメインにいくつかの VF を割り当て、IO ドメインにもいくつかの VF を割り当てます

デバイスまたはプラットフォームが次の機能をサポートしている場合は、それらのテストを行います。

- VF から IO ドメインをブートします
- SR-IOV カードを物理的にホットプラグしたり取り外したりします
- SR-IOV カードで動的再構成処理を実行します

デバイスが Oracle Solaris の複数のバージョンでサポートされている場合、最終的なテストを実行するには、いくつかのテストでルートおよび IO ドメイン全体で OS バージョンを混在させます。

◆ ◆ ◆ 第 23 章

23 デバイスドライバのデバッグ、テスト、およびチューニング

この章では、デバイスドライバのテスト、デバッグ、およびチューニングを支援するために提供されている各種ツールの概要を説明します。この章では、次の内容について説明します。

- 557 ページの「ドライバのテスト」 – ドライバをテストすると、システムの機能が損なわれる可能性があります。シリアル接続と代替カーネルの両方を使用すると、クラッシュからの復旧が容易になります。
- 568 ページの「デバッグツール」 – 組み込みのデバッグ機能を使用すると、独立したデバッガを実行しなくても、ドライバ機能の動作テストの実行や監視を容易に行えます。
- 581 ページの「ドライバのチューニング」 – Oracle Solaris OS には、デバイスドライバのパフォーマンスを測定する機能が用意されています。デバイスのカーネル統計構造体を記述すると、デバイスの動作中に連続的な統計情報がエクスポートされます。パフォーマンス改善の対象領域を判定した場合、DTrace 動的計測ツールを使用すると、どのような問題もより厳密に特定しやすくなります。

ドライバのテスト

データ損失などの問題を避けるため、新しいデバイスドライバをテストするときには、特に注意を払うようにしてください。このセクションでは、さまざまなテスト方針について説明します。たとえば、シリアル接続経由で制御可能な別個のシステムを設定するのが、新しいドライバをテストする場合にもっとも安全な方法です。さまざまなカーネル変数設定を含むテストモジュールをロードし、さまざまなカーネル条件の下でパフォーマンスをテストできます。システムがクラッシュした場合に備え、バックアップデータの復元、任意のクラッシュダンプの解析、およびデバイスディレクトリの再構築を行う準備を整えておくべきです。

ハードハングを避けるためのデッドマン機能の有効化

システムがハードハング状態に陥ると、デバッガを起動できなくなります。デッドマン機能を有効にすると、システムはいつまでもハングする代わりにパニック状態になります。その後、kmdb(1) カーネルデバッガを使用して問題の解析を行えます。

デッドマン機能は1秒に一度、システムクロックが更新されているかどうかをチェックします。システムクロックが更新されていない場合、システムが無期限のハングに陥っていることになります。システムクロックが 50 秒間更新されないと、デッドマン機能によってパニックが引き起こされ、デバッガが起動されます。

デッドマン機能を有効化するには、次の手順に従います。

1. クラッシュイメージの取得中であることを dumpadm(1M) で確認します。
2. /etc/system ファイル内で snooping 変数を設定します。/etc/system ファイルについては、system(4) のマニュアルページを参照してください。

   ```
   set snooping=1
   ```
3. /etc/system ファイルが再度読み取られて snooping 設定が有効になるように、システムをリブートします。

システム上のすべてのゾーンもデッドマン設定を継承します。

デッドマン機能が有効な状態でシステムがハングすると、次の例のような出力がコンソール上に表示されるはずです。

```
panic[cpu1]/thread=30018dd6cc0: deadman: timed out after 9 seconds of
clock inactivity

panic: entering debugger (continue to save dump)
```

デバッガの内部から ::cpuinfo コマンドを使用することで、クロック割り込みの発行に失敗してシステム時間を進めることができなかった原因を調査します。

シリアル接続を使用したテスト

ドライバをテストするときには、シリアル接続を使用することをお勧めします。ホストシステムとテストシステムの間でシリアル接続を確立するには、tip(1) コマンドを使用します。このアプローチでは、ホストコンソールの *tip* ウィンドウがテストマシンのコンソールとして使用されます。詳細については、tip(1) のマニュアルページを参照してください。

tip ウィンドウには次のような利点があります。

- テストシステムやカーネルデバッグとの対話内容を監視できます。たとえば、ドライバが原因でテストシステムがクラッシュした場合、このウィンドウは使用中のセッションのログを記録できます。
- *tip* ホストマシンにログインし、tip(1) を使用してテストマシンに接続することにより、テストマシンにリモートアクセスできます。

注 – Oracle Solaris デバイスドライバのデバッグに tip 接続や 2 台目のマシンの使用が必須というわけではありませんが、それでもこの手法をお勧めします。

▼ tip 接続用にホストシステムを設定するには

1. ホストシステムとテストマシンとを、両マシン上のシリアルポート A を使用して接続します。

   この接続を行う際にはヌルモデムケーブルを使用する必要があります。

2. ホストシステム上で、**/etc/remote** 内に接続用のエントリが存在していることを確認します。詳細は、**remote(4)** のマニュアルページを参照してください。

   この端末エントリは、使用するシリアルポートに一致している必要があります。シリアルポート B については、Oracle Solaris オペレーティングシステムに適切なエントリが含まれていますが、シリアルポート A については端末エントリを追加する必要があります。

   ```
   debug:\
           :dv=/dev/term/a:br#9600:el=^C^S^Q^U^D:ie=%$:oe=^D:
   ```

   注 – ボーレートは 9600 に設定する必要があります。

   ---

3. ホストのシェルウィンドウで **tip(1)** を実行し、エントリの名前を指定します。

   ```
   % tip debug
   connected
   ```

   これによって、そのシェルウィンドウがテストマシンのコンソールへの接続を含む tip ウィンドウになります。

注意 – テストマシンを停止するとき、ホストマシンが SPARC マシンの場合は STOP-A キーを、x86 アーキテクチャーマシンの場合は F1-A キーを、それぞれホストマシン上で使用しないでください。この操作を行うと、実際にはホストマシンが停止されます。テストマシンに BREAK 信号を送信するには、tip ウィンドウで **~#** と入力します。**~#** などのコマンドが認識されるのは、それらの文字が行の先頭にある場合のみです。このコマンドで効果がない場合は、Return キーまたは Control-U キーを押してください。

SPARC プラットフォームのターゲットシステムの設定

SPARC プラットフォームのテストマシンを設定する簡単な方法は、マシンの電源を入れる前にキーボードを取り外すことです。これによって、マシンのシリアルポート A が自動的にコンソールとして使用されます。

テストマシンを設定するもう 1 つの方法は、ブート PROM コマンドを使用してシリアルポート A をコンソールにすることです。テストマシンのブート PROM の ok プロンプトで、コンソール入出力をシリアル回線に転送します。テストマシンの起動時に必ずシリアルポート A がコンソールとして使用されるようにするには、環境変数 input-device と output-device を設定します。

例 **23–1** ブート PROM コマンドによる input-device と output-device の設定

```
ok setenv input-device ttya
ok setenv output-device ttya
```

eeprom コマンドを使用すると、シリアルポート A をコンソールにすることもできます。スーパーユーザーとして次のコマンドを実行し、input-device および output-device パラメータがシリアルポート A を指すようにします。次の例はその eeprom コマンドを示したものです。

例 **23–2** eeprom コマンドによる input-device と output-device の設定

```
# eeprom input-device=ttya
# eeprom output-device=ttya
```

この eeprom コマンドにより、その後システムがブートされるたびにコンソールがシリアルポート A にリダイレクトされます。

x86 プラットフォーム上のターゲットシステムの設定

x86 プラットフォームでは、eeprom コマンドを使用してシリアルポート A をコンソールにします。その手順は、SPARC プラットフォームの手順と同じです。560 ページの「SPARC プラットフォームのターゲットシステムの設定」を参照してください。この eeprom コマンドにより、リブート中にコンソールがシリアルポート A (COM1) に切り替わります。

注 – x86 マシンでは、BIOS がシリアルポートへのコンソールリダイレクションをサポートしていないかぎり、ブート処理のある初期段階に達するまでコンソールの制御が tip 接続に移りません。SPARC マシンではブート処理の全体を通じて、tip 接続がコンソールの制御を維持します。

テストモジュールの設定

/etc ディレクトリ内の system(4) ファイルを使用すると、ブート時にカーネル変数の値を設定できます。カーネル変数を使用すると、ドライバの複数の動作を切り替えたり、カーネルが提供するデバッグ機能を活用したりできます。デバッグ時に非常に役立つ可能性のあるカーネル変数 moddebug と kmem_flags については、このセクションで後述します。558 ページの「ハードハングを避けるためのデッドマン機能の有効化」も参照してください。

ブート後のカーネル変数の変更は、信頼できません。/etc/system はカーネルのブート時に一度だけ読み取られるからです。このファイルを変更したあと、その変更を有効にするには、システムをリブートする必要があります。ファイルの変更後にシステムが機能しなくなった場合は、ask (-a) オプションを指定してブートします。その後、/dev/null をシステムファイルとして指定します。

注 – 将来のリリースでもカーネル変数が存在すると仮定することはできません。

カーネル変数の設定

set コマンドは、モジュール変数またはカーネル変数の値を変更します。モジュール変数を設定するには、次のようにモジュール名と変数を指定します。

```
set module_name:variable=value
```

たとえば、myTest という名前のドライバの変数 test_debug を設定するには、set を次のように使用します。

```
% set myTest:test_debug=1
```

カーネル自体によってエクスポートされた変数を設定する場合は、モジュール名を省略します。

値の設定には、ビット単位の論理和演算を使用することもできます。次に例を示します。

```
% set moddebug | 0x80000000
```

テストモジュールのロードとアンロード

コマンド modload(1M)、modunload(1M)、および modinfo(1M) を使用するとテストモジュールを追加できます。これは、ドライバのデバッグや負荷テストを行うための便利な手法です。一般に、これらのコマンドは通常運用時には必要ありません。必要なモジュールのロードや未使用モジュールのアンロードは、カーネルが自動的に行うからです。情報の提供と制御の設定を行うため、moddebug カーネル変数はこれらのコマンドと連携して動作します。

modload() 関数の使用

あるモジュールを強制的にメモリー内にロードするには、modload(1M) を使用します。modload コマンドは、ドライバのロード時にそのドライバに未解決の参照が含まれていないことを確認します。ドライバのロードは必ずしも、そのドライバが接続可能であることを意味するわけではありません。ドライバのロードが成功すると、ドライバの _info(9E) エントリポイントが呼び出されます。attach() エントリポイントは、必ずしも呼び出されるわけではありません。

modinfo() 関数の使用

ドライバがロードされたことを確認するには、modinfo(1M) を使用します。

例 23–3 modinfo によるロード済みドライバの確認

```
$ modinfo
 Id Loadaddr   Size Info Rev Module Name
  6 101b6000    732   -   1  obpsym (OBP symbol callbacks)
  7 101b65bd  1acd0 226   1  rpcmod (RPC syscall)
  7 101b65bd  1acd0 226   1  rpcmod (32-bit RPC syscall)
  7 101b65bd  1acd0   1   1  rpcmod (rpc interface str mod)
  8 101ce8dd  74600   0   1  ip (IP STREAMS module)
  8 101ce8dd  74600   3   1  ip (IP STREAMS device)
...
$ modinfo | grep mydriver
169 781a8d78   13fb   0   1  mydriver (Test Driver 1.5)
```

info フィールドの番号は、そのドライバ用に選択されたメジャー番号です。モジュール ID を利用できる場合は、modunload(1M) コマンドを使用してモジュールをアンロードできます。モジュール ID は、modinfo 出力の左側の列に表示されます。

modunload の発行後に予期したとおりにドライバがアンロードされない場合がありますが、これはドライバがビジー状態であると判定されたためです。この状況が発生するのは、ドライバが実際にビジー状態であるか、または detach エントリポイントの実装が間違っているために、ドライバの detach(9E) が失敗した場合です。

modunload() の使用

現時点で使用されていないすべてのモジュールをメモリーから削除するには、モジュール ID に 0 を指定して modunload(1M) を実行します。

```
# modunload -i 0
```

moddebug カーネル変数の設定

moddebug カーネル変数は、モジュールのロード処理を制御します。moddebug の使用可能な値は次のとおりです。

0x80000000	モジュールのロードまたはアンロード時にコンソールにメッセージを出力します。
0x40000000	より詳しいエラーメッセージを提供します。
0x20000000	ロードまたはアンロード時に、アドレスやサイズを含めるなど、より詳しい情報を出力します。
0x00001000	ドライバの自動アンロードを行いません。システムリソースが少なくなっても、システムがデバイスドライバのアンロードを試みません。
0x00000080	ストリームの自動アンロードを行いません。システムリソースが少なくなっても、システムが STREAMS モジュールのアンロードを試みません。
0x00000010	すべてのタイプのカーネルモジュールの自動アンロードを行いません。
0x00000001	kmdb を使用して実行する場合、各モジュールの _init() ルーチンが呼び出される前に、moddebug によってブレークポイントが実行され、kmdb への復帰がただちに発生します。また、この設定ではモジュールの _info() および _fini() ルーチンの実行時に追加のデバッグメッセージが生成されます。

kmem_flags デバッグフラグの設定

kmem_flags カーネル変数は、カーネルのメモリーアロケータのデバッグ機能を有効化します。アロケータのデバッグ機能を有効化するには、kmem_flags を 0xf に設定します。これらの機能には、次のコード条件を検出するための実行時チェックが含まれます。

- バッファー解放後のバッファーへの書き込み
- メモリー初期化前のメモリーの使用
- バッファーの限度を超える書き込み

カーネルメモリーアロケータを使用してそのような問題を解析する方法については、『Solaris モジューラデバッガ』を参照してください。

注 – kmem_flags を 0xf に設定した状態でテストや開発を行うと、潜在的なメモリー破壊バグの検出が容易になる可能性があります。kmem_flags を 0xf に設定するとカーネルメモリーアロケータの内部動作が変化するため、kmem_flags を使用しない状態でも完全なテストを実施するべきです。

テストシステムでのデータ損失の回避

ドライバのバグにより、システムがブートできなくなる場合があります。このセクションで説明するように、予防策を実施しておくことで、このようなイベントが発生した場合でもシステムの再インストールを回避できます。

重要なシステムファイルをバックアップする

多数のドライバ関連システムファイルを再構築することは、不可能でないにしても非常に困難です。ドライバのインストール中にドライバが原因でシステムがクラッシュすると、/etc/name_to_major、/etc/driver_aliases、/etc/driver_classes、/etc/minor_perm などのファイルが破壊される可能性があります。add_drv(1M) のマニュアルページを参照してください。

安全のため、テストマシンの構成が正しく完了したあとで、ルートファイルシステムのバックアップコピーを作成します。/etc/system ファイルの変更を予定している場合には、変更を実施する前にこのファイルのバックアップコピーを作成します。

▼ 代替カーネルでブートするには

システムが動作不能になるのを防ぐには、デフォルトカーネルからブートする代わりに、カーネルと関連バイナリのコピーからブートするべきです。

1. **/platform/* 内のドライバのコピーを作成します。**

   ```
   # cp -r /platform/‘uname -i‘/kernel /platform/‘uname -i‘/kernel.test
   ```

2. **/platform/‘uname -i‘/kernel.test/drv 内にドライバモジュールを配置します。**

3. **デフォルトカーネルの代わりに代替カーネルをブートします。**

   代替カーネルの作成および格納が完了したら、このカーネルをさまざまな方法でブートできます。

   - 次のようにリブートすることで、代替カーネルをブートできます。

     ```
     # reboot -- kernel.test/unix
     ```

   - SPARC システムの場合、次のように PROM からブートすることもできます。

```
ok boot kernel.test/sparcv9/unix
```

注 – kmdb デバッガを使用してブートするには、572 ページの「モジュラーデバッガの使用開始」で説明するように -k オプションを使用します。

- x86 システムの場合、ブート処理で Select (b)oot or (i)nterpreter: メッセージが表示されたときに、次のように入力します。

```
boot kernel.test/unix
```

例 23–4 代替カーネルのブート

次の例は、代替カーネルでのブート方法を示したものです。

```
ok boot kernel.test/sparcv9/unix
Rebooting with command: boot kernel.test/sparcv9/unix
Boot device: /sbus@1f,0/espdma@e,8400000/esp@e,8800000/sd@0,0:a File and \
    args:
kernel.test/sparcv9/unix
```

例 23–5 -a オプションを使用した代替カーネルのブート

ask (-a) オプションを指定してブートすることで、モジュールパスを変更することもできます。このオプションを指定すると、ブート方法を構成するための一連のプロンプトが表示されます。

```
ok boot -a
Rebooting with command: boot -a
Boot device: /sbus@1f,0/espdma@e,8400000/esp@e,8800000/sd@0,0:a File and \
args: -a
Enter filename [kernel/sparcv9/unix]: kernel.test/sparcv9/unix
Enter default directory for modules
[/platform/sun4u/kernel.test /kernel /usr/kernel]: <CR>
Name of system file [etc/system]: <CR>
SunOS Release 5.10 Version Generic 64-bit
Copyright 1983-2002 Sun Microsystems, Inc. All rights reserved.
root filesystem type [ufs]: <CR>
Enter physical name of root device
[/sbus@1f,0/espdma@e,8400000/esp@e,8800000/sd@0,0:a]: <CR>
```

代替バックアップ計画の検討

システムがネットワークに接続されている場合、テストマシンをサーバーのクライアントとして追加できます。問題が発生した場合には、ネットワークからシステムをブートできます。その後、ローカルディスクをマウントして任意の修正を行えます。Oracle Solaris システムの CD-ROM から直接システムをブートすることもできます。

障害から復旧するためのもう 1 つの方法は、別のブート可能なルートファイルシステムを用意することです。format(1M) を使用して、元のパーティションとまったく同じサイズのパーティションを作成します。次に、dd(1M) を使用して、そのブート可能なルートファイルシステムをコピーします。コピーの完了後、新しいファイルシステムに対して fsck(1m) を実行し、その整合性を確認します。

その後、元のルートパーティションからシステムをブートできなくなったら、バックアップパーティションをブートします。dd(1M) を使用してバックアップパーティションを元のパーティションにコピーします。ルートファイルシステムが破損していないにもかかわらず、システムをブートできないような状況が発生する可能性もあります。たとえば、破損箇所がブートブロックやブートプログラムに限定されていることがあります。そのような場合は、ask (-a) オプションを使用してバックアップパーティションからブートできます。その後、元のファイルシステムをルートファイルシステムとして指定できます。

システムクラッシュダンプの取得

システムがパニック状態になると、システムはカーネルメモリーのイメージをダンプデバイスに書き込みます。デフォルトでは、もっとも適したスワップデバイスがダンプデバイスになります。このダンプはシステムのクラッシュダンプであり、アプリケーションによって生成されるコアダンプに似ています。パニックが発生したあとのリブート時に、savecore(1M) はダンプデバイス内でクラッシュダンプの存在の有無をチェックします。ダンプが見つかった場合、savecore は、unix.*n* という名前のカーネルシンボルテーブルのコピーを作成します。次に savecore ユーティリティーは、vmcore.*n* という名前のコアファイルをコアイメージディレクトリ内にダンプします。デフォルトでは、コアイメージディレクトリは /var/crash/*machine_name* になります。/var/crash の容量不足によりコアダンプを格納できない場合には、システムによって必要な容量が表示されるものの、実際のダンプの保存が行われません。その後、コアダンプや保存済みカーネルに対して mdb(1) デバッガを使用できます。

Oracle Solaris オペレーティングシステムでは、クラッシュダンプはデフォルトで有効になっています。dumpadm(1M) コマンドは、システムクラッシュダンプを構成するために使用されます。dumpadm コマンドを使用すると、クラッシュダンプが有効になっていることを確認したり、保存されたコアファイルの場所を特定したりできます。

注 – savecore ユーティリティーがファイルシステムをいっぱいにしてしまうのを防ぐことができます。minfree という名前のファイルを、ダンプの保存先となるディレクトリに追加します。savecore の実行後に空き領域として残すキロバイト数を、このファイル内に指定します。十分な容量を確保できない場合、コアファイルは保存されません。

デバイスディレクトリの復旧

attach(9E) の実行中にドライバがクラッシュすると、/devices および /dev ディレクトリが破損する可能性があります。いずれかのディレクトリが破損した場合にそのディレクトリを再構築するには、システムをブートし、fsck(1m) を実行して破損したルートファイルシステムを修復します。これによって、ルートファイルシステムをマウントできるようになります。/devices および /dev ディレクトリを作成し直すには、devfsadm(1M) を実行し、マウントされたディスク上で /devices ディレクトリを指定します。

次の例は、SPARC システム上の破損したルートファイルシステムを修復する方法を示しています。この例で、破損したディスクは /dev/dsk/c0t3d0s0、代替ブートディスクは /dev/dsk/c0t1d0s0 になっています。

例 23–6 破損したデバイスディレクトリの復旧

```
ok boot disk1
...
Rebooting with command: boot kernel.test/sparcv9/unix
Boot device: /sbus@1f,0/espdma@e,8400000/esp@e,8800000/sd@31,0:a File and \
    args:
kernel.test/sparcv9/unix
...
# fsck /dev/dsk/c0t3d0s0** /dev/dsk/c0t3d0s0
** Last Mounted on /
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Cyl groups
1478 files, 9922 used, 29261 free
     (141 frags, 3640 blocks, 0.4% fragmentation)
# mount /dev/dsk/c0t3d0s0 /mnt
# devfsadm -r /mnt
```

注 – /devices および /dev ディレクトリを修正すると、システムのほかの部分が壊れたままでも、システムをブートできる可能性があります。そのような修復は、システムを再インストールする前にシステムクラッシュダンプなどの情報を保存するための一時的な修正にすぎません。

デバッグツール

このセクションでは、デバイスドライバに適用可能な 2 つのデバッガについて説明します。両デバッガの詳細については、『Solaris モジューラデバッガ』を参照してください。

- kmdb(1) カーネルデバッガは、ブレークポイント、ウォッチポイント、シングルステップ実行などの典型的な実行時デバッガ機能を提供します。 kmdb デバッガは、以前のリリースで使用可能だった kadb を置き換えるものです。kmdb では新しい機能のほかに、kadb で以前に使用可能だったコマンドも使用されています。kadb はブート時にしかロードできませんでしたが、kmdb はいつでもロードできます。kmdb デバッガは、実行制御機能を備えているため、ライブでの対話式デバッグに適しています。
- mdb(1) モジュラーデバッガは、リアルタイムデバッガとしては kmdb よりも制限がありますが、mdb には事後デバッグ用の豊富な機能が備わっています。

デバッガ kmdb と mdb は基本的に同じユーザーインタフェースを共有しています。したがって、両ツールの同じコマンドでは、多くのデバッグ手法が適用可能となっています。どちらのデバッガもマクロ、dcmd、および dmod をサポートしています。*dcmd* (「ディーコマンド」と発音) は、現在のターゲットプログラムのすべてのプロパティーにアクセス可能なデバッガ内のルーチンです。dcmd は実行時に動的にロードできます。デバッガモジュールの短縮形である *dmod* は dcmd をパッケージ化したものであり、標準以外の動作を提供するためにロードできます。

mdb と kmdb はどちらも、adb や kadb といった旧バージョンのデバッガと下位互換性を保っています。mdb デバッガは、kmdb で使用可能なすべてのマクロを実行できるだけでなく、adb 向けの旧バージョンの任意のユーザー定義マクロも実行できます。標準マクロセットの検索場所については、『Oracle Solaris モジューラデバッガ』を参照してください。

事後デバッグ

事後解析は、ドライバ開発者にさまざまなメリットを提供します。1 つの問題を複数の開発者が並行して検査できます。単一のクラッシュダンプに対してデバッガの複数のインスタンスを同時に使用できます。解析をオフラインで実行できるため、可能な場合にはクラッシュしたシステムを稼働状態に戻すことができます。事後解析では、ユーザーが開発したデバッガ機能を dmod の形式で使用できます。kmdb などのリアルタイムデバッガではメモリー消費量が多すぎて使用できない機能でも、dmod ではバンドルできます。

kmdb がロードされた状態でシステムがパニック状態になると、その調査をただちに行えるようにデバッガに制御が渡されます。kmdb で現在の問題を解析するのが適切でないと考えられる場合、推奨される方針は、:c を使用して実行を継続し、ク

ラッシュダンプを保存することです。システムがリブートしたら、保存したクラッシュダンプに対して mdb を使用して事後解析を実行できます。この処理は、アプリケーションのクラッシュをプロセスコアファイルに基づいてデバッグするのに似ています。

注 – 以前のバージョンの Solaris オペレーティングシステムでは、adb(1) が事後解析の推奨ツールでした。現在の Oracle Solaris オペレーティングシステムでは、mdb(1) が事後解析の推奨ツールです。mdb() の機能セットは、旧バージョンの crash(1M) ユーティリティーのコマンドセットよりも優れています。Oracle Solaris オペレーティングシステムでは crash ユーティリティーは使用できなくなりました。

---

kmdb カーネルデバッガの使用

kmdb デバッガは対話型のカーネルデバッガであり、次の機能を提供します。

- カーネル実行の制御
- カーネル状態の検査
- コードのライブ変更

このセクションでは、ユーザーがすでに kmdb デバッガについて精通していることを前提にしています。このセクションの重点は、デバイスドライバの設計時に役立つ kmdb 機能にあります。kmdb の使用方法について詳しく学ぶには、kmdb(1) のマニュアルページと『Solaris モジューラデバッガ』を参照してください。kadb に精通している場合は、kadb(1M) のマニュアルページを参照し、kadb と kmdb の主な違いを確認してください。

kmdb デバッガのロードやアンロードは、任意で実行できます。kmdb のロードやアンロードの手順については、『Oracle Solaris モジューラデバッガ』を参照してください。安全性と利便性の観点から、代替カーネルでのブートを強くお勧めします。このセクションで説明するように、ブート処理は SPARC プラットフォームと x86 プラットフォームとでわずかに異なります。

注 – kmdb はデフォルトで、kmdb の実行中のプロンプトとして CPU ID を使用します。この章の例では、特に明記されていないかぎり、[0] をプロンプトとして使用しています。

SPARC プラットフォームでの代替カーネルを使用した kmdb のブート

kmdb と代替カーネルの両方を使用して SPARC システムをブートするには、次のいずれかのコマンドを使用します。

```
boot kmdb -D kernel.test/sparcv9/unix
boot kernel.test/sparcv9/unix -k
```

x86 プラットフォームでの代替カーネルを使用した kmdb のブート

kmdb と代替カーネルの両方を使用して x86 システムをブートするには、次のいずれかのコマンドを使用します。

```
b kmdb -D kernel.test/unix
b kernel.test/unix -k
```

kmdb でのブレークポイントの設定

ブレークポイントを設定するには、次の例に示すように bp コマンドを使用します。

例 23–7 kmdb での標準ブレークポイントの設定

```
[0]> myModule‘myBreakpointLocation::bp
```

ターゲットモジュールがロードされていない場合、この状態を示すエラーメッセージが表示され、ブレークポイントは作成されません。この場合には遅延ブレークポイントを使用できます。遅延ブレークポイントは、指定されたモジュールがロードされた時点で自動的に有効になります。遅延ブレークポイントを設定するには、bp コマンドのあとにターゲット位置を指定します。次に遅延ブレークポイントの例を示します。

例 23–8 kmdb での遅延ブレークポイントの設定

```
[0]>::bp myModule‘myBreakpointLocation
```

ブレークポイントの使用方法の詳細については、『Oracle Solaris モジューラデバッガ』を参照してください。次の 2 つの行のいずれかを入力してもヘルプが得られます。

```
> ::help bp
> ::bp dcmd
```

ドライバ開発者向けの kmdb マクロ

kmdb(1M) デバッガでサポートされているマクロを使用すると、カーネルデータ構造体を表示できます。kmdb のマクロを表示するには $M を使用します。マクロは次の形式で使用します。

```
[ address ] $<macroname
```

注 – これらのマクロから表示される情報もその表示形式も、インタフェースの一部ではありません。したがって、その情報と形式は常に変更される可能性があります。

次の表に含まれる kmdb マクロは、デバイスドライバの開発者に特に役立ちます。便宜上、該当する場合は旧バージョンのマクロ名も示します。

表 23–1 kmdb マクロ

dcmd	旧バージョンのマクロ	説明
::devinfo	devinfo devinfo_brief devinfo.prop	あるデバイスノードのサマリーを出力します
::walk devinfo_parents	devinfo.parent	デバイスノードの上位ノードを調査します
::walk devinfo_sibling	devinfo.sibling	あるデバイスノードの兄弟ノードを調査します
::minornodes	devinfo.minor	指定されたデバイスノードに対応するマイナーノードを出力します
::major2name		指定されたデバイスノードにバインドされたデバイスの名前を出力します。
::devbindings		指定されたデバイスノードまたはメジャー番号にバインドされたデバイスノードを出力します。

::devinfo dcmd で表示されるノード状態は、次のいずれかの値を取ります。

DS_ATTACHED　ドライバの attach(9E) ルーチンが正常に復帰しました。

DS_BOUND　このノードはドライバにバインドされていますが、ドライバの probe(9E) ルーチンがまだ呼び出されていません。

DS_INITIALIZED　親のネクサスがドライバ用のバスアドレスを割り当てました。実装に固有の初期化が完了しています。ドライバの probe(9E) ルーチンは、現時点でまだ呼び出されていません。

DS_LINKED　このデバイスノードがカーネルのデバイスツリー内にリンクされましたが、システムはまだこのノードのドライバを検出していません。

DS_PROBED	ドライバの probe(9E) ルーチンが正常に復帰しました。
DS_READY	このデバイスは完全に構成されています。

mdb モジュラーデバッガの使用

mdb(1) モジュラーデバッガは、次のタイプのファイルに適用できます。

- ライブのオペレーティングシステムコンポーネント
- オペレーティングシステムのクラッシュダンプ
- ユーザープロセス
- ユーザープロセスのコアダンプ
- オブジェクトファイル

mdb デバッガは、カーネルの問題を解析するための高度なデバッグサポートを提供します。このセクションでは mdb 機能の概要を説明します。mdb の完全な説明については、『Solaris モジューラデバッガ』を参照してください。

mdb を使用してライブカーネル状態を変更することも可能ですが、mdb には、kmdb で提供されているカーネル実行制御の機能がありません。このため、実行時デバッグには kmdb をお勧めします。mdb デバッガは通常、静的な状況で使用されます。

注 – mdb のプロンプトは > です。

モジュラーデバッガの使用開始

mdb デバッガにはデバッガモジュールを実装するための拡張プログラミング API が用意されているため、ドライバ開発者はカスタムのデバッグサポートを実装できます。mdb デバッガには、コマンド行編集、コマンド履歴、出力ページャー、オンラインヘルプなど、多数のユーザビリティー機能も用意されています。

注 – adb マクロは使用するべきではありません。その機能の大部分が、mdb の dcmd に置き換えられました。

---

mdb デバッガには一連の豊富なモジュールや dcmd が用意されています。これらのツールを使用すると、Oracle Solaris カーネル、任意の関連モジュール、およびデバイスドライバをデバッグできます。これらの機能を使用すると、次のようなタスクを実行できます。

- 複雑なデバッグクエリーを構築する
- 特定のスレッドによって割り当てられたすべてのメモリーを検出する
- カーネル STREAM のビジュアル画像を出力する

- 特定のアドレスが参照している構造タイプを判定する
- カーネルの中でリークしているメモリーブロックを検出する
- スタックトレースを検出するためのメモリーを分析する
- カスタマイズされた操作を作成するために、複数の dcmd を組み合わせて *dmod* と呼ばれるモジュールを構築する

使用を開始するには、次の例に示すように、クラッシュディレクトリに移動して `mdb` と入力し、システムクラッシュダンプを指定します。

例 **23–9** クラッシュダンプに対する `mdb` の呼び出し

```
% cd /var/crash/testsystem
% ls
bounds     unix.0    vmcore.0
% mdb unix.0 vmcore.0
Loading modules: [ unix krtld genunix ufs_log ip usba s1394 cpc nfs ]
> ::status
debugging crash dump vmcore.0 (64-bit) from testsystem
operating system: 5.10 Generic (sun4u)
panic message: zero
dump content: kernel pages only
```

`mdb` からの応答として `>` プロンプトが返されたら、コマンドを実行できます。

ライブシステム上の実行中のカーネルを検査するには、次のようにシステムプロンプトから `mdb` を実行します。

例 **23–10** 実行中のカーネルに対する `mdb` の呼び出し

```
# mdb -k
Loading modules: [ unix krtld genunix ufs_log ip usba s1394 ptm cpc ipc nfs ]
> ::status
debugging live kernel (64-bit) on testsystem
operating system: 5.10 Generic (sun4u)
```

kmdb と mdb を使用した便利なデバッグタスク

このセクションでは便利なデバッグタスクの例について説明します。このセクションに含まれるタスクは、特に明記されていないかぎり、`mdb`、`kmdb` のいずれかを使用して実行できます。このセクションでは、ユーザーに `kmdb` と `mdb` の使用に関する基礎知識があることを前提にしています。ここで示す情報は、使用するシステムの種類に依存します。これらの例を作成する際には、64 ビットカーネルが稼働する Sun Blade 100 ワークステーションを使用しました。

注意 – カーネル構造体のデータを変更すると不可逆なデータ破壊が起こる可能性があるため、細心の注意を払うべきです。Oracle Solaris DDI の一部でない構造体に含まれるデータを変更したり、そのようなデータに依存したりしないようにしてください。Oracle Solaris DDI の一部である構造体については、Intro(9S) のマニュアルページを参照してください。

kmdb によるシステムレジスタの調査

kmdb デバッガを使用すると、マシンのレジスタを 1 つのグループとして表示することも、個々のレジスタ単位で表示することもできます。すべてのレジスタを 1 つのグループとして表示するには、次の例に示すように $r を使用します。

例 23–11 kmdb による SPARC プロセッサ上のすべてのレジスタの読み取り

```
[0]: $r

g0    0                                   l0      0
g1    100130a4      debug_enter           l1      edd00028
g2    10411c00      tsbmiss_area+0xe00    l2      10449c90
g3    10442000      ti_statetbl+0x1ba     l3      1b
g4    3000061a004                         l4      10474400      ecc_syndrome_tab+0x80
g5    0                                   l5      3b9aca00
g6    0                                   l6      0
g7    2a10001fd40                         l7      0
o0    0                                   i0      0
o1    c                                   i1      10449e50
o2    20                                  i2      0
o3    300006b2d08                         i3      10
o4    0                                   i4      0
o5    0                                   i5      b0
sp    2a10001b451                         fp      2a10001b521
o7    1001311c      debug_enter+0x78      i7      1034bb24      zsa_xsint+0x2c4
y     0
tstate: 1604  (ccr=0x0, asi=0x0, pstate=0x16, cwp=0x4)
pstate: ag:0 ie:1 priv:1 am:0 pef:1 mm:0 tle:0 cle:0 mg:0 ig:0
winreg: cur:4 other:0 clean:7 cansave:1 canrest:5 wstate:14
tba   0x10000000
pc    edd000d8 edd000d8:        ta      %icc,%g0 + 125
npc   edd000dc edd000dc:        nop
```

デバッガは各レジスタの値を、そのレジスタと同じ名前を持つ変数にエクスポートします。変数を読み取ると、レジスタの現在の値が返されます。変数に書き込みを行うと、関連するマシンレジスタの値が変更されます。次の例では、x86 マシン上の %o0 レジスタの値を、0 から 1 に変更しています。

例 23–12 kmdb による x86 マシン上のレジスタの読み取りと書き込み

```
[0]> &lt;eax=K
        c1e6e0f0
[0]> 0>eax
```

例 23–12 kmdb による x86 マシン上のレジスタの読み取りと書き込み (続き)

```
[0]> &lt;eax=K
        0
[0]>  c1e6e0f0>eax
```

別のプロセッサのレジスタを検査する必要がある場合は、::cpuregs dcmd を使用できます。次の例に示すように、検査対象となるプロセッサの ID は、この dcmd へのアドレスとして指定することも、-c オプションの値として指定することもできます。

例 23–13 別のプロセッサのレジスタの検査

```
[0]> 0::cpuregs
   %cs = 0x0158            %eax = 0xc1e6e0f0 kmdbmod`kaif_dvec
   %ds = 0x0160            %ebx = 0x00000000
```

次の例では、SPARC マシン上でプロセッサを 0 から 3 に切り替えています。%g3 レジスタが検査されたあと、クリアされています。新しい値を確認するため、%g3 が再度読み取られています。

例 23–14 指定されたプロセッサからの、特定のレジスタ値の取得

```
[0]> 3::switch
[3]> <g3=K
        24
[3]> 0>g3
[3]> <g3
        0
```

カーネルメモリーリークの検出

::findleaks dcmd は、カーネルクラッシュダンプ内のメモリーリークを検出する強力で効率的な機能を提供します。カーネルメモリーデバッグ機能のフルセットを有効化しないと、::findleaks が有効になりません。詳細については、563 ページの「kmem_flags デバッグフラグの設定」を参照してください。ドライバの開発中やテスト中に ::findleaks を実行すると、メモリーリークが発生したためにカーネルリソースが浪費されているコードが検出されます。『Solaris モジューラデバッガ』の第 9 章「カーネルメモリーアロケータを使用するデバッギング」Oracle Solaris Modular Debugger Guide を参照してください。

注 – カーネルメモリーリークが発生したコードがあると、サービス拒否攻撃に対するシステムの脆弱性が高まる可能性があります。

---

mdb を使用したデバッガコマンドの記述

mdb デバッガには、ドライバのデバッグ用にカスタマイズ可能なデバッガ機能を実装するための強力な API が用意されています。このプログラミング API の詳細については、『Oracle Solaris モジューラデバッガ』で説明されています。

SUNWmdbdm パッケージは、ディレクトリ /usr/demo/mdb に mdb のサンプルソースコードをインストールします。mdb を使用すると、ドライバが正しく動作していることを確認するための時間のかかるデバッグ作業やデバッグ支援を自動化できます。また、mdb デバッグモジュールをドライバ製品と一緒にパッケージ化することもできます。パッケージ化すると、サービス担当者がこれらの機能を顧客サイトで使用できるようになります。

カーネルデータ構造体情報の取得

Oracle Solaris カーネルはデータ型の情報を構造体として提供しており、これらの構造体は、kmdb と mdb のいずれかを使用して検査できます。

注 – kmdb と mdb の dcmd は、mdb 用として設計された圧縮シンボリックデバッギング情報を含むオブジェクトでしか使用できません。現在、この情報を利用できるのは、特定の Oracle Solaris カーネルモジュールのみです。このシンボリックデバッギング情報を処理するには、SUNWzlib パッケージをインストールする必要があります。

次の例は、scsi_pkt 構造体のデータを表示する方法を示しています。

例 23–15 デバッガによるカーネルデータ構造体の表示

```
> 7079ceb0::print -t 'struct scsi_pkt'
{
    opaque_t pkt_ha_private = 0x7079ce20
    struct scsi_address pkt_address = {
        struct scsi_hba_tran *a_hba_tran = 0x70175e68
        ushort_t a_target = 0x6
        uchar_t a_lun = 0
        uchar_t a_sublun = 0
    }
    opaque_t pkt_private = 0x708db4d0
    int (*)() *pkt_comp = sd_intr
    uint_t pkt_flags = 0
    int pkt_time = 0x78
    uchar_t *pkt_scbp = 0x7079ce74
    uchar_t *pkt_cdbp = 0x7079ce64
    ssize_t pkt_resid = 0
    uint_t pkt_state = 0x37
    uint_t pkt_statistics = 0
    uchar_t pkt_reason = 0
}
```

データ構造体のサイズは、デバッグ時に役立つ可能性があります。ある構造体のサイズを取得するには、次の例に示すように ::sizeof dcmd を使用します。

例 23–16 カーネルデータ構造体のサイズの表示

```
> ::sizeof struct scsi_pkt
sizeof (struct scsi_pkt) = 0x58
```

デバッグ時には、構造体内の特定のメンバーのアドレスも役立ちます。あるメンバーのアドレスを調査する場合に使用可能な方法は、いくつか存在します。

構造体の特定のメンバーのオフセットを取得するには、次の例のように ::offsetof dcmd を使用します。

例 23–17 カーネルデータ構造体へのオフセットの表示

```
> ::offsetof struct scsi_pkt pkt_state
offsetof (struct pkt_state) = 0x48
```

構造体のすべてのメンバーのアドレスを表示するには、次の例のように ::print dcmd で -a オプションを指定します。

例 23–18 カーネルデータ構造体の相対アドレスの表示

```
> ::print -a struct scsi_pkt
{
    0 pkt_ha_private
    8 pkt_address {
    ...
    }
    18 pkt_private
    ...
}
```

::print でアドレスと -a オプションを組み合わせて指定すると、各メンバーの絶対アドレスが表示されます。

例 23–19 カーネルデータ構造体の絶対アドレスの表示

```
> 10000000::print -a struct scsi_pkt
{
    10000000 pkt_ha_private
    10000008 pkt_address {
    ...
    }
    10000018 pkt_private
    ...
}
```

::print、::sizeof、::offsetof の各 dcmd を使用すると、ドライバが Oracle Solaris カーネルと対話する際に発生した問題をデバッグできます。

注意 – この機能を使用すると、生のカーネルデータ構造体にアクセスできます。ユーザーは、構造体が DDI の一部として表示されるかどうかにかかわらず、任意の構造体を検査できます。したがって、明示的に DDI の一部になっていないデータ構造体には依存しないようにしてください。

注 – 前述の dcmd は、mdb 用として設計された圧縮シンボリックデバッギング情報を含むオブジェクトでしか使用しないようにしてください。現在、シンボリックデバッギング情報を使用できるのは、特定の Oracle Solaris カーネルモジュールのみです。シンボリックデバッギング情報を処理するには、SUNWzlib (32 ビット) または SUNWzlibx (64 ビット) 圧縮ソフトウェアをインストールする必要があります。kmdb デバッガは、SUNWzlib または SUNWzlibx パッケージの有無にかかわらず、シンボリックタイプのデータを処理できます。

デバイスツリー情報の取得

mdb デバッガには、カーネルデバイスツリーを表示するための ::prtconf dcmd が用意されています。::prtconf dcmd の出力は、prtconf(1M) コマンドの出力に似ています。

例 23–20 ::prtconf dcmd の使用

```
> ::prtconf
300015d3e08      SUNW,Sun-Blade-100
    300015d3c28      packages (driver not attached)
        300015d3868      SUNW,builtin-drivers (driver not attached)
        300015d3688      deblocker (driver not attached)
        300015d34a8      disk-label (driver not attached)
        300015d32c8      terminal-emulator (driver not attached)
        300015d30e8      obp-tftp (driver not attached)
        300015d2f08      dropins (driver not attached)
        300015d2d28      kbd-translator (driver not attached)
        300015d2b48      ufs-file-system (driver not attached)
    300015d3a48      chosen (driver not attached)
    300015d2968      openprom (driver not attached)
```

ノードを表示するには、次の例に示すように ::devinfo dcmd などのマクロを使用します。

例 23–21 特定のノードのデバイス情報の表示

```
> 300015d3e08::devinfo
300015d3e08      SUNW,Sun-Blade-100
```

例 23–21 特定のノードのデバイス情報の表示 (続き)

```
        System properties at 0x300015abdc0:
            name='relative-addressing' type=int items=1
                value=00000001
            name='MMU_PAGEOFFSET' type=int items=1
                value=00001fff
            name='MMU_PAGESIZE' type=int items=1
                value=00002000
            name='PAGESIZE' type=int items=1
                value=00002000
        Driver properties at 0x300015abe00:
            name='pm-hardware-state' type=string items=1
                value='no-suspend-resume'
```

::prtconf を使用すると、デバイスツリー内でドライバが接続されている場所を確認したり、デバイスのプロパティーを表示したりできます。また、次のように ::prtconf に詳細 (-v) フラグを指定することで、各デバイスノードのプロパティーを表示することもできます。

例 23–22 詳細モードの ::prtconf dcmd の使用

```
> ::prtconf -v
DEVINFO          NAME
300015d3e08      SUNW,Sun-Blade-100
        System properties at 0x300015abdc0:
            name='relative-addressing' type=int items=1
                value=00000001
            name='MMU_PAGEOFFSET' type=int items=1
                value=00001fff
            name='MMU_PAGESIZE' type=int items=1
                value=00002000
            name='PAGESIZE' type=int items=1
                value=00002000
        Driver properties at 0x300015abe00:
            name='pm-hardware-state' type=string items=1
                value='no-suspend-resume'
        ...
        300015ce798      pci10b9,5229, instance #0
                Driver properties at 0x300015ab980:
                    name='target2-dcd-options' type=any items=4
                        value=00.00.00.a4
                    name='target1-dcd-options' type=any items=4
                        value=00.00.00.a2
                    name='target0-dcd-options' type=any items=4
                        value=00.00.00.a4
```

ドライバのインスタンスを特定する別の方法として、::devbindings dcmd が挙げられます。次の例に示すように、ドライバ名が指定されると、このコマンドは指定されたドライバのすべてのインスタンスの一覧を表示します。

例 23–23 ::devbindings dcmd を使用したドライバインスタンスの特定

```
> ::devbindings dad
300015ce3d8      ide-disk (driver not attached)
300015c9a60      dad, instance #0
        System properties at 0x300015ab400:
            name='lun' type=int items=1
                value=00000000
            name='target' type=int items=1
                value=00000000
            name='class_prop' type=string items=1
                value='ata'
            name='type' type=string items=1
                value='ata'
            name='class' type=string items=1
                value='dada'
...
300015c9880      dad, instance #1
        System properties at 0x300015ab080:
            name='lun' type=int items=1
                value=00000000
            name='target' type=int items=1
                value=00000002
            name='class_prop' type=string items=1
                value='ata'
            name='type' type=string items=1
                value='ata'
            name='class' type=string items=1
                value='dada'
```

ドライバのソフト状態情報の取得

ドライバをデバッグするときの一般的な問題は、ある特定のドライバインスタンスの ソフト状態を取得することです。ソフト状態は、ddi_soft_state_zalloc(9F) ルーチンで割り当てられます。ドライバからソフト状態を取得するには、ddi_get_soft_state(9F) を使用します。ソフト状態ポインタの名前が、ddi_soft_state_init(9F) の第一の引数になります。名前を使用すると、mdb で ::softstate dcmd を使用して特定のドライバインスタンスのソフト状態を取得できます。

```
> *bst_state::softstate 0x3
702b7578
```

この場合、::softstate を使用して bst サンプルドライバのインスタンス 3 のソフト状態を取得しています。このポインタは、ドライバがこのインスタンスの状態を追跡するために使用する bst_soft 構造体を参照しています。

カーネル変数の変更

カーネル変数などのカーネル状態を変更する場合、kmdb と mdb のどちらを使用してもかまいません。mdb は変更前にカーネルを停止しないため、mdb によるカーネル状態の変更は注意して行うべきです。kmdb を使用すると一連の変更を原子的に行えま

すが、これは、kmdb がユーザーにアクセスを許可する前にカーネルを停止するためです。mdb デバッガでは、原子的な変更は 1 つしか行えません。

変更の実行時には、必ず適切な書式指定子を使用してください。書式は次のとおりです。

- w – 各式の値の下位 2 バイトを、ドットで指定された位置から始まるターゲットに書き込む
- W – 各式の値の下位 4 バイトを、ドットで指定された位置から始まるターゲットに書き込む
- Z – 各式の値の 8 バイトすべてを、ドットで指定された位置から始まるターゲットに書き込む

変更する変数のサイズを確認するには、::sizeof dcmd を使用します。

次の例では、moddebug の値を値 0x80000000 で上書きしています。

例 23–24 デバッガによるカーネル変数の変更

```
> moddebug/W 0x80000000
    moddebug:       0 = 0x80000000
```

ドライバのチューニング

Oracle Solaris OS には、ユーザーがドライバのカウンタを実装できるように、カーネル統計構造体が用意されています。DTrace 機能を使用すると、パフォーマンスの解析をリアルタイムで行えます。このセクションで説明するデバイスのパフォーマンスに関するトピックは、次のとおりです。

- 581 ページの「カーネル統計」 – Oracle Solaris OS には、カーネル内のパフォーマンス統計を取得するための一連のデータ構造体と関数が用意されています。カーネル統計 (*kstat* と呼ばれる) を使用すると、ドライバは、システムの実行中に連続的な統計情報をエクスポートできます。kstat データは、kstat 関数を使用してプログラム的に処理されます。
- 588 ページの「動的計測を行うための DTrace」 – DTrace を使用すると、ドライバに計測機能を動的に追加できるため、システム解析やパフォーマンス測定などのタスクを実行できます。DTrace では事前定義された kstat 構造体が利用されます。

カーネル統計

パフォーマンスチューニングを支援するため、Oracle Solaris カーネルには kstat(3KSTAT) 機能が用意されています。kstat 機能が提供する一連の関数やデータ

構造体を使用すると、デバイスドライバなどのカーネルモジュールからモジュール固有のカーネル統計をエクスポートできます。

kstat は、デバイスの使用に関する定量化可能な側面を記録するためのデータ構造体です。kstat は、NULL で終わるリンクリストとして格納されます。各 kstat には共通のヘッダーセクションとタイプ固有のデータセクションとが含まれます。ヘッダーセクションは kstat_t 構造体によって定義されます。

カーネル統計構造体のメンバー

kstat 構造体のメンバーは次のとおりです。

ks_class[KSTAT_STRLEN]	kstat のタイプを bus、controller、device_error、disk、hat、kmem_cache、kstat、misc、net、nfs、pages、partition、rps、ufs、vm、vmem のいずれかに分類します。
ks_crtime	kstat が作成された時間。ks_crtime は一般に、各種カウンタのレートを計算するときに使用されます。
ks_data	kstat のデータセクションを指します。
ks_data_size	データセクションの合計サイズ (バイト)。
ks_instance	この kstat を作成したカーネルモジュールのインスタンス。ks_instance を ks_module、ks_name と組み合わせることで、意味のある一意の名前が kstat に付けられます。
ks_kid	kstat の一意の ID。
ks_module[KSTAT_STRLEN]	この kstat を作成したカーネルモジュールを識別します。ks_module を ks_instance、ks_name と組み合わせることで、意味のある一意の名前が kstat に付けられます。KSTAT_STRLEN は ks_module の最大長を設定します。
ks_name[KSTAT_STRLEN]	ks_module および ks_instance とともに kstat に割り当てられた名前。KSTAT_STRLEN は ks_module の最大長を設定します。
ks_ndata	複数レコードをサポートする kstat タイプ KSTAT_TYPE_RAW、KSTAT_TYPE_NAMED、および KSTAT_TYPE_TIMER のデータレコード数を示します。
ks_next	チェーン内での次の kstat を指します。
ks_resv	予約済みのフィールド。

ks_snaptime	前回のデータスナップショットのタイムスタンプ。レートの計算時に役立ちます。
ks_type	データタイプ。バイナリデータの場合は KSTAT_TYPE_RAW、名前/値ペアの場合は KSTAT_TYPE_NAMED、割り込み統計の場合は KSTAT_TYPE_INTR、入出力統計の場合は KSTAT_TYPE_IO、イベントタイマーの場合は KSTAT_TYPE_TIMER になります。

カーネル統計構造体

さまざまな種類の kstat 用の構造体を次に示します。

kstat(9S)	デバイスドライバからエクスポートされる各カーネル統計 (kstat) は、ヘッダーセクションとデータセクションから構成されます。kstat(9S) 構造体は統計のヘッダー部分です。
kstat_intr(9S)	割り込み kstat 用の構造体。割り込みのタイプは次のとおりです。 ■ ハード割り込み – ハードウェアデバイス自体がソースとなる ■ ソフト割り込み – システムが何らかのシステム割り込みソースを使用することで引き起こされる ■ ウォッチドッグ割り込み – 定期的なタイマー呼び出しによって引き起こされる ■ 見せかけの割り込み – 割り込みエントリポイントに入ったが、処理するべき割り込みが存在しなかった ■ 複数の処理 – ほかの任意のタイプから制御が戻る直前に、割り込みが検出されて処理された ドライバは一般に、自身のハンドラから要求されたハード割り込みとソフト割り込みのみを報告しますが、見せかけのクラスの割り込みの測定は、自動ベクトル化デバイスが特定のシステム構成内で任意の割り込み待ち時間の問題を特定する際に役立ちます。同じタイプの割り込みを複数持つデバイスでは、複数の構造体を使用するべきです。
kstat_io(9S)	入出力 kstat 用の構造体。
kstat_named(9S)	名前付き kstat 用の構造体。名前付き kstat は名前-値ペアの配列です。これらのペアは kstat_named 構造体に保持されます。

カーネル統計関数

kstat を使用するための関数を次に示します。

kstat_create(9F)
kstat(9S) 構造体を割り当てて初期化します。

kstat_delete(9F)
システムから kstat を削除します。

kstat_install(9F)
完全に初期化された kstat をシステムに追加します。

kstat_named_init(9F)、kstat_named_setstr(9F)
名前付き kstat を初期化します。kstat_named_setstr() は、名前付き kstat のポインタに文字列 str を関連付けます。

kstat_queue(9F)
多数の入出力サブシステムは、管理対象となる基本的なトランザクションキューを少なくとも 2 つ持ちます。1 つは、処理対象として受け付けられたものの、まだ処理が開始されていないトランザクション用のキューです。もう 1 つは、アクティブに処理されているものの、まだ完了していないトランザクション用のキューです。このため、待機時間と実行時間 という 2 つの累計時間統計が保持されます。待機時間は処理の開始前です。実行時間は処理中です。次の kstat_queue() ファミリ関数は、ドライバの待機キューと実行キュー間の遷移に基づいてこれらの時間を管理します。

- kstat_runq_back_to_waitq(9F)
- kstat_runq_enter(9F)
- kstat_runq_exit(9F)
- kstat_waitq_enter(9F)
- kstat_waitq_exit(9F)
- kstat_waitq_to_runq(9F)

Oracle Solaris Ethernet ドライバのカーネル統計

次の表で説明する kstat インタフェースは、Ethernet 物理層の統計をドライバから取得するための効果的な方法です。ユーザーが Ethernet 物理層の問題の診断や修復をより適切に行えるように、Ethernet ドライバからこれらの統計をエクスポートするようにしてください。link_up 以外のすべての統計では、存在しない場合のデフォルト値は 0 になります。link_up 統計の値は 1 を想定します。

次の例では、すべての共有リンク設定を提供しています。この場合、mii を使用して統計をフィルタリングしています。

```
kstat ce:0:mii:link_*
```

表 23–2 Ethernet MII/GMII 物理層インタフェースのカーネル統計

kstat 変数	タイプ	説明
xcvr_addr	KSTAT_DATA_UINT32	現在使用中のトランシーバの MII アドレスを提供します。 ■ (0) - (31) は、ある特定の Ethernet デバイスで使用中の物理層デバイスの MII アドレス用です。 ■ (-1) は、外部的にアクセス可能な MII インタフェースが存在しないため、MII アドレスが未定義または無関係である場合に使用されます。
xcvr_id	KSTAT_DATA_UINT32	現在使用中のトランシーバの特定のベンダー ID またはデバイス ID を提供します。
xcvr_inuse	KSTAT_DATA_UINT32	現在使用中のトランシーバのタイプを示します。IEEE aPhytType は次のセットを列挙します。 ■ (0) その他、未定義 ■ (1) MII インタフェースが存在しないが、トランシーバが接続されていない ■ (2) 10Mbps Clause 7 10Mbps Manchester ■ (3) 100BASE-T4 Clause 23 100Mbps 8B/6T ■ (4) 100BASE-X Clause 24 100Mbps 4B/5B ■ (5) 100BASE-T2 Clause 32 100Mbps PAM5X5 ■ (6) 1000BASE-X Clause 36 1000Mbps 8B/10B ■ (7) 1000BASE-T Clause 40 1000Mbps 4D-PAM5 このセットは、ifMauType で指定されるセットより小さくなります。ifMauType は、前出のすべてとその半二重/全二重オプションを含むように定義されています。この情報は cap_* 統計から提供可能であるため、xcvr_inuse と cap_* を組み合わせて不足する定義を派生させることで、ifMayType のすべての組み合わせを提供できます。
cap_1000fdx	KSTAT_DATA_CHAR	デバイスが 1Gbps 全二重をサポートすることを示します。
cap_1000hdx	KSTAT_DATA_CHAR	デバイスが 1Gbps 半二重をサポートすることを示します。
cap_100fdx	KSTAT_DATA_CHAR	デバイスが 100Mbps 全二重をサポートすることを示します。
cap_100hdx	KSTAT_DATA_CHAR	デバイスが 100Mbps 半二重をサポートすることを示します。
cap_10fdx	KSTAT_DATA_CHAR	デバイスが 10Mbps 全二重をサポートすることを示します。
cap_10hdx	KSTAT_DATA_CHAR	デバイスが 10Mbps 半二重をサポートすることを示します。
cap_asmpause	KSTAT_DATA_CHAR	デバイスが非対称ポーズ Ethernet フロー制御をサポートすることを示します。

表 23–2 Ethernet MII/GMII 物理層インタフェースのカーネル統計 (続き)

kstat 変数	タイプ	説明
cap_pause	KSTAT_DATA_CHAR	cap_pause が 1 に設定され、cap_asmpause が 0 に設定されている場合、デバイスが対称ポーズ Ethernet フロー制御をサポートすることを示します。cap_asmpause が 1 に設定されている場合、cap_pause の意味は次のようになります。 ■ cap_pause = 0 受信輻輳に基づいてポーズを送信します。 ■ cap_pause = 1 ポーズを受信し、輻輳を避けるために送信速度を落とします。
cap_rem_fault	KSTAT_DATA_CHAR	デバイスがリモート障害インジケーションをサポートすることを示します。
cap_autoneg	KSTAT_DATA_CHAR	デバイスが自動ネゴシエーションをサポートすることを示します。
adv_cap_1000fdx	KSTAT_DATA_CHAR	デバイスが 1Gbps 全二重のサポートを通知していることを示します。
adv_cap_1000hdx	KSTAT_DATA_CHAR	デバイスが 1Gbps 半二重のサポートを通知していることを示します。
adv_cap_100fdx	KSTAT_DATA_CHAR	デバイスが 100Mbps 全二重のサポートを通知していることを示します。
adv_cap_100hdx	KSTAT_DATA_CHAR	デバイスが 100Mbps 半二重のサポートを通知していることを示します。
adv_cap_10fdx	KSTAT_DATA_CHAR	デバイスが 10Mbps 全二重のサポートを通知していることを示します。
adv_cap_10hdx	KSTAT_DATA_CHAR	デバイスが 10Mbps 半二重のサポートを通知していることを示します。
adv_cap_asmpause	KSTAT_DATA_CHAR	デバイスが非対称ポーズ Ethernet フロー制御のサポートを通知していることを示します。
adv_cap_pause	KSTAT_DATA_CHAR	adv_cap_pause が 1 に設定され、adv_cap_asmpause が 0 に設定されている場合、デバイスが対称ポーズ Ethernet フロー制御のサポートを通知していることを示します。adv_cap_asmpause が 1 に設定されている場合、adv_cap_pause の意味は次のようになります。 ■ adv_cap_pause = 0 受信輻輳に基づいてポーズを送信します。 ■ adv_cap_pause = 1 ポーズを受信し、輻輳を避けるために送信速度を落とします。
adv_rem_fault	KSTAT_DATA_CHAR	リンク先に転送する予定の障害がデバイスで発生していることを示します。

表 23–2 Ethernet MII/GMII 物理層インタフェースのカーネル統計 *(続き)*

kstat 変数	タイプ	説明
adv_cap_autoneg	KSTAT_DATA_CHAR	デバイスが自動ネゴシエーションのサポートを通知していることを示します。
lp_cap_1000fdx	KSTAT_DATA_CHAR	リンク先のデバイスが 1Gbps 全二重をサポートすることを示します。
lp_cap_1000hdx	KSTAT_DATA_CHAR	リンク先のデバイスが 1Gbps 半二重をサポートすることを示します。
lp_cap_100fdx	KSTAT_DATA_CHAR	リンク先のデバイスが 100Mbps 全二重をサポートすることを示します。
lp_cap_100hdx	KSTAT_DATA_CHAR	リンク先のデバイスが 100Mbps 半二重をサポートすることを示します。
lp_cap_10fdx	KSTAT_DATA_CHAR	リンク先のデバイスが 10Mbps 全二重をサポートすることを示します。
lp_cap_10hdx	KSTAT_DATA_CHAR	リンク先のデバイスが 10Mbps 半二重をサポートすることを示します。
lp_cap_asmpause	KSTAT_DATA_CHAR	リンク先のデバイスが非対称ポーズ Ethernet フロー制御をサポートすることを示します。
lp_cap_pause	KSTAT_DATA_CHAR	lp_cap_pause が 1 に設定され、lp_cap_asmpause が 0 に設定されている場合、リンク先のデバイスが対称ポーズ Ethernet フロー制御をサポートすることを示します。lp_cap_asmpause が 1 に設定されている場合、lp_cap_pause の意味は次のようになります。 ■ lp_cap_pause = 0 リンク先が受信輻輳に基づいてポーズを送信します。 ■ lp_cap_pause = 1 リンク先がポーズを受信し、輻輳を避けるために送信速度を落とします。
lp_rem_fault	KSTAT_DATA_CHAR	リンクの障害がリンク先で発生していることを示します。
lp_cap_autoneg	KSTAT_DATA_CHAR	リンク先のデバイスが自動ネゴシエーションをサポートすることを示します。
link_asmpause	KSTAT_DATA_CHAR	リンクが非対称ポーズ Ethernet フロー制御で動作していることを示します。

表 23–2 Ethernet MII/GMII 物理層インタフェースのカーネル統計 (続き)

kstat 変数	タイプ	説明
link_pause	KSTAT_DATA_CHAR	ポーズ機能の解決を示します。link_pause が 1 に設定され、link_asmpause が 0 に設定されている場合、リンクが対称ポーズ Ethernet フロー制御で動作していることを示します。link_asmpause が 1 に設定されていて、それがリンクのローカルビューに対して相対的である場合、link_pause の意味は次のようになります。 ■ link_pause = 0 このステーションが受信輻輳に基づいてポーズを送信します。 ■ link_pause = 1 このステーションがポーズを受信し、輻輳を避けるために送信速度を落とします。
link_duplex	KSTAT_DATA_CHAR	リンクのデュプレックスを示します。 ■ link_duplex = 0 リンクが停止しており、デュプレックスは不明です。 ■ link_duplex = 1 リンクが動作しており、モードは半二重です。 ■ link_duplex = 2 リンクが動作しており、モードは全二重です。
link_up	KSTAT_DATA_CHAR	リンクが動作しているか、それとも停止しているかを示します。 ■ link_up = 0 リンクが停止しています。 ■ link_up = 1 リンクが動作しています。

動的計測を行うための DTrace

DTrace は、ユーザープログラムとオペレーティングシステム自体の両方の動作を検査するための包括的な動的トレース機能です。DTrace を使用すると、プローブと呼ばれる環境内の戦略的な場所で、データを収集できます。DTrace では、スタックトレースやタイムスタンプ、関数の引数などのデータを記録できるほか、単純にプローブの起動回数を記録することもできます。DTrace ではプローブを動的に挿入できるため、コードをコンパイルし直す必要はありません。DTrace の詳細については、『Oracle Solaris 11.1 Dynamic Tracing Guide』を参照してください。

◆ ◆ ◆ 第 24 章

24 推奨されるコーティング方法

この章では、堅牢なドライバを記述する方法について説明します。この章で説明するガイダンスに従ってドライバを記述すると、デバッグが容易になります。この推奨される方法に従えば、システムもハードウェアやソフトウェアの障害から保護されます。

この章では、次の内容について説明します。

デバッグ準備手法

ドライバコードのデバッグは、次の理由から、ユーザープログラムの場合より困難です。

- ドライバはハードウェアと直接対話する
- ドライバは、ユーザープロセスに提供されるオペレーティングシステムの保護なしに動作する

ドライバにはデバッグのサポートを組み込むようにします。このサポートは、保守作業と将来の開発を促進します。

一意の接頭辞を使用してカーネルシンボルの衝突を回避する

それぞれの関数、データ要素、およびドライバプリプロセッサの定義は、ドライバごとに一意であることが必要です。

ドライバモジュールはカーネルにリンクされます。特定のドライバに対して一意である各シンボルの名前は、決してほかのカーネルシンボルと衝突しないようにしてください。そのような衝突を回避するには、特定のドライバの関数とデータ要素のそれぞれに、そのドライバに共通の接頭辞を使用して名前を付ける必要があります。接頭辞を使用すると、各ドライバシンボルに一意の名前を付けるのに十分なはずです。通常、この接頭辞はドライバの名前またはドライバ名の略語になります。たとえば `xx_open()` は、ドライバ `xx` の `open`(9E) ルーチンの名前です。

ドライバを作成するときには、ドライバに必ずいくつかのシステムヘッダーファイルを含める必要があります。これらのヘッダーファイル内に記載された、グローバルに認識される名前を予測することはできません。これらの名前との衝突を避けるには、識別用の接頭辞を使用して、各ドライバプリプロセッサの定義に一意の名前を付ける必要があります。

ドライバシンボルの接頭辞を識別できれば、トラブルシューティング時にシステムログやパニックを解読する助けにもなります。あいまいな `attach()` 関数に関連するエラーを確認する代わりに、`xx_attach()` に関するエラーメッセージを確認します。

`cmn_err()` を使用してドライバの活動を記録する

`cmn_err`(9F) 関数を使用して、デバイスドライバ内からシステムログにメッセージを出力します。カーネルモジュール用の `cmn_err` (9F) 関数はアプリケーション用の `printf`(3C) 関数と似ています。`cmn_err`(9F) 関数には、デバイスレジスタのビットを出力する `%b` 書式などの追加の書式文字が用意されています。`cmn_err`(9F) 関数はシステムログにメッセージを書き込みます。`/var/adm/messages` にあるこれらのメッセージは、`tail`(1) コマンドを使用して監視します。

```
% tail -f /var/adm/messages
```

`ASSERT()` を使用して無効な前提条件を見つける

アサーションは非常に役立つ形式のアクティブドキュメントです。`ASSERT`(9F) の構文は次のとおりです。

```
void ASSERT(EXPRESSION)
```

`ASSERT()` マクロは、真であることが予期されている条件が実際には偽である場合、カーネルの実行を停止します。`ASSERT()` はプログラマに、特定のコードによって作成された前提条件を検証する方法を提供します。

`ASSERT()` マクロが定義されるのは、`DEBUG` コンパイルシンボルが定義されている場合のみです。`DEBUG` が定義されていない場合、`ASSERT()` マクロは有効になりません。

次のアサーションの例では、特定のポインタの値は `NULL` ではないという前提条件をテストしています。

```
ASSERT(ptr != NULL);
```

ドライバが DEBUG を使用してコンパイルされていて、実行のこの時点で ptr の値が NULL である場合、次のパニックメッセージがコンソールに出力されます。

```
panic: assertion failed: ptr != NULL, file: driver.c, line: 56
```

注 – ASSERT(9F) は DEBUG コンパイルシンボルを使用するため、どの条件付きデバッグコードでも DEBUG を使用します。

---

mutex_owned() を使用してロック要件の検証とドキュメント化を行う

mutex_owned(9F) の構文は次のとおりです。

```
int mutex_owned(kmutex_t *mp);
```

ドライバ開発のかなりの部分で、複数のスレッドを正しく処理する必要があります。mutex が取得されるときには常にコメントを使用する必要があります。明らかに必要な mutex が取得されていない場合は、コメントがさらに役立つことがあります。mutex がスレッドによって保持されているかどうかを判定するには、ASSERT(9F) 内で mutex_owned() を使用します。

```
void helper(void)
{
    /* this routine should always be called with xsp's mutex held */
    ASSERT(mutex_owned(&xsp->mu));
    /* ... */
}
```

注 – mutex_owned() は ASSERT () マクロ内でのみ有効です。ドライバの動作を制御するためには mutex_owned() を使用してください。

条件付きコンパイルを使用してコストの高いデバッグ機能を切り替える

DEBUG などのプリプロセッサシンボルを使用するか、グローバル変数を使用すると、条件付きコンパイルによってドライバ内にデバッグ用のコードを挿入できます。条件付きコンパイルを使用すると、本番ドライバ内で不要なコードを削除できます。実行時のデバッグの出力量を設定するには変数を使用します。出力は、ioctl またはデバッガを使用して実行時のデバッグレベルを設定することで指定できます。通常は、これらの 2 つの方法が組み合わせられます。

次の例はコンパイラを使って到達不能コード (この場合は常に偽となるゼロのテストに続くコード) を削除しています。この例では、/etc/system で設定したり、デバッガによってパッチを適用したりできるローカル変数も提供しています。

```
#ifdef DEBUG
/* comments on values of xxdebug and what they do */
static int xxdebug;
#define dcmn_err if (xxdebug) cmn_err
#else
#define dcmn_err if (0) cmn_err
#endif
/* ... */
    dcmn_err(CE_NOTE, "Error!\n");
```

このメソッドは、cmn_err(9F) が可変数の引数を持つ状況を処理します。もう 1 つのメソッドは、マクロには引数が 1 つ (cmn_err(9F) 用の括弧付きの引数の一覧) あることを利用します。マクロはこの引数を削除します。このマクロは、DEBUG が定義されていない場合はマクロをゼロに展開することによって、オプティマイザへの依存も削除します。

```
#ifdef DEBUG
/* comments on values of xxdebug and what they do */
static int xxdebug;
#define dcmn_err(X) if (xxdebug) cmn_err X
#else
#define dcmn_err(X) /* nothing */
#endif
/* ... */
/* Note:double parentheses are required when using dcmn_err. */
    dcmn_err((CE_NOTE, "Error!"));
```

この手法は多くの方法で拡張できます。1 つの方法として、xxdebug の値に応じて、cmn_err(9F) にあるさまざまなメッセージを指定します。ただし、そのような場合は、デバッグ情報が多すぎてコードが複雑でわかりにくくならないように気を付ける必要があります。

もう 1 つ一般的なのは、xxlog() 関数を記述する方法です。この関数は vsprintf(9F) または vcmn_err(9F) を使用して可変変数の一覧を処理します。

変数の volatile 宣言

volatile は、デバイスレジスタを参照する変数を宣言するときに適用する必要があるキーワードです。volatile を使用しないと、コンパイル時のオプティマイザが偶然、重要なアクセスを削除することがあります。volatile を使用しないでいると、追跡するのが困難なバグが発生する可能性があります。

見つけにくいバグを防ぐには、volatile を正しく使用する必要があります。volatile キーワードはコンパイラに、宣言されているオブジェクトの正確なセマンティクスを使用するように指示します。これは特に、オブジェクトへのアクセ

スの削除や並び替えを行わないようにするためです。デバイスドライバで volatile 修飾子を使う必要があるのは次の 2 つの場合です。

- データが外部ハードウェアデバイスのレジスタ、つまりストレージのみでない副次的作用を持ったメモリーを参照している場合。ただし、デバイスレジスタにアクセスするために DDI データアクセス関数が使用されている場合は、volatile を使用する必要がないことに注意してください。
- 複数のスレッドからアクセス可能で、ロックによって保護されておらず、順序付けのメモリーアクセスに依存しているグローバルメモリーをデータが参照している場合。volatile 使用時のリソースの消費はロックを使用する場合より少なくなります。

次の例では volatile を使用しています。デバイスがビジーのときにスレッドが継続しないようにビジーフラグが使用され、フラグはロックによって保護されていません。

```
while (busy) {
  /* do something else */
  }
```

テストスレッドは、別のスレッドが busy フラグをオフにした場合は続行されます。

```
busy = 0;
```

busy はテストスレッドで頻繁にアクセスされるため、コンパイラは毎回のテストの前にメモリー内の busy の値を読み込まずに、busy の値をレジスタに置き、レジスタの内容をテストすることでテストを最適化できる可能性があります。テストスレッドから見ると busy が変化することはなく、ほかのスレッドのみがメモリー内の busy の値を変更することがあるため、デッドロックが発生する結果になります。busy フラグを volatile と宣言することで、各テストの前にフラグの値を強制的に読み取ります。

注 – busy フラグの代わりになる方法は、条件変数を使用することです。72 ページの「スレッド同期における条件変数」を参照してください。

---

volatile 修飾子を使用するときには、不注意による抜けが生じないようにします。たとえば、次のコードの場合を考えます。

```
struct device_reg {
 volatile uint8_t csr;
 volatile uint8_t data;
 };
 struct device_reg *regp;
```

これは、もう 1 つの例よりも推奨される記述方法です。

```
struct device_reg {
 uint8_t csr;
 uint8_t data;
 };
 volatile struct device_reg *regp;
```

2 つの例は機能上は同等ですが、2 つ目の例では、コードの記述者は `struct` 型の `device_reg` のすべての宣言で `volatile` を使用するようにする必要があります。最初の例では、すべての宣言でデータが volatile として処理されることになるため、これが推奨される方法です。前述したように、デバイスレジスタにアクセスするために DDI データアクセス関数を使用すると、`volatile` としての修飾変数が不要になります。

保守性

保守性を確保するには、ドライバが次の動作を実行できるようにする必要があります。

- 障害の発生したデバイスを検出して障害を報告する
- Oracle Solaris ホットプラグモデルによってサポートされているとおりにデバイスを取り外す
- Oracle Solaris ホットプラグモデルによってサポートされているとおりに新しいデバイスを追加する
- 定期的な健全性検査を実行して潜在的な障害の検出を可能にする

定期的な健全性検査

潜在的な障害とは、何らかのほかの動作が発生するするまではそれ自身が表面化しない障害のことです。たとえば、コールドスタンバイになっているデバイスで発生しているハードウェアの障害は、マスターデバイスで障害が発生するまで検出されないままになる場合があります。この時点で、システムには障害のあるデバイスが 2 つ含まれることになり、処理を継続できない可能性があります。

検出されないままの潜在的な障害は、通常、最終的にはシステム障害の原因となります。潜在的な障害の検査を行わないと、冗長システムの全体での可用性が損なわれます。このような状況を回避するには、デバイスドライバで潜在的な障害を検出し、ほかの障害と同じ方法で報告する必要があります。

ドライバには、デバイスで定期的な健全性検査を行うためのメカニズムを備える必要があります。デバイスがセカンダリデバイスまたはフェイルオーバーデバイスになる場合がある、耐障害を備えた状況では、プライマリデバイスで障害が発生する前に、障害が発生したセカンダリデバイスを早期に検出し、セカンダリデバイスを修復または交換できるようにすることが非常に重要です。

定期的な健全性検査を使用すると、次の活動を実行できます。

- 最後のポーリング以降に値が変更された可能性のあるデバイスで、レジスタまたはメモリーの場所を検査します。

  通常、決定にかかわる動作を示すデバイスの機能には、ハートビートセマフォー、デバイスタイマー(たとえば、ダウンロードによって使用されるローカルの `lbolt`)、およびイベントカウンタが含まれます。更新された予測可能な値をデバイスから読み取ることで、処理が正常に行われているという確信を、ある程度持つことができます。
- 転送ブロックやドライバによって発行されたコマンドなどの送信要求にタイムスタンプを付けます。

  定期的な健全性検査によって、完了していないと疑われる要求を探すことができます。
- 次に予定されている検査の前に完了する必要のある動作を、デバイスで開始します。

  この動作が割り込みである場合、この検査は、デバイスの回路から割り込みを出力できることを確認する最適な方法です。

パート IV

付録

付録では次の背景情報を提供します。

- 付録 A「ハードウェアの概要」では、デバイスドライバのマルチプラットフォームハードウェアの問題について説明します。
- 付録 B「Solaris DDI/DKI サービスのサマリー」では、デバイスドライバ用のカーネル関数の表を提供します。非推奨となった関数も明記します。
- 付録 C「64 ビットデバイスドライバの準備」では、64 ビット環境で動作するようにデバイスドライバを更新するためのガイドラインを提供します。
- 付録 D「コンソールフレームバッファードライバ」では、フレームバッファードライバに必要なインタフェースを追加することで、そのドライバを Oracle Solaris カーネル端末エミュレータと対話できるようにする方法について説明します。
- 付録 E「`pci.conf` ファイル」では、`pci.conf(4)` 構成ファイルについて説明します。

◆ ◆ ◆ 付 録 A

ハードウェアの概要

この付録では、Oracle Solaris OS をサポートできるハードウェアに関する一般的な問題について説明します。この説明には、Oracle Solaris OS がサポートするプロセッサ、バスアーキテクチャー、およびメモリーモデルが含まれます。また、デバイスのさまざまな問題や、使用される PROM についても説明します。

注 – この付録に記載されている資料は、情報の提供のみを目的としています。この情報は、ドライバのデバッグ時に役立つことがあります。ただし、実装に関するこれらの詳細の多くは、Oracle Solaris DDI/DKI インタフェースによってデバイスドライバから隠されています。

この付録では、次の内容について説明します。

SPARC プロセッサの問題

このセクションでは、データ割り当て、バイト順序、レジスタウィンドウ、浮動小数点命令の利用可能性など、SPARC プロセッサ固有のいくつかのトピックについて説明します。x86 プロセッサ固有のトピックについては、601 ページの「x86 プロセッサの問題」を参照してください。

注 – 浮動小数点演算はカーネルでサポートされていないため、ドライバでこの演算を実行しないでください。

SPARC のデータ割り当て

すべての数量は、標準の C データ型を使用して、自然な境界に割り当てる必要があります。

- `short` 整数は、16 ビットの境界に割り当てます。
- `int` 整数は、32 ビットの境界に割り当てます。
- `long` 整数は、SPARC システムでは 64 ビットの境界に割り当てます。データモデルについては、付録 C 「64 ビットデバイスドライバの準備」を参照してください。
- `long long` 整数は、64 ビットの境界に割り当てます。

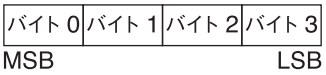
通常、割り当ての問題はコンパイラで処理されます。ただし、ドライバの作成者は、デバイスへのアクセスに適切なデータ型を使用する必要があるため、割り当てに注意を払う傾向があります。一般にデバイスレジスタにはポインタ参照によってアクセスするため、ドライバはデバイスへのアクセス時にポインタが適正に割り当てるようにする必要があります。

SPARC 構造体のメンバー割り当て

SPARC プロセッサによって生じるデータ割り当ての制限のため、C 構造体にも割り当ての要件があります。構造体の割り当て要件は、もっとも厳密に割り当てられた構造体コンポーネントによって生じます。たとえば、文字だけを含む構造体には割り当ての制限はありませんが、`long long` メンバーを含む構造体は、このメンバーが 64 ビットの境界で割り当てられることを保証するように構築する必要があります。

SPARC のバイト順序

SPARC プロセッサでは、ビッグエンディアンのバイト順序を使用します。整数の最上位バイト (MSB) は、その整数の一番低いアドレスに格納されます。最下位バイトは、このプロセッサのワードの一番高いアドレスに格納されます。たとえば、バイト 63 は 64 ビットプロセッサの最下位バイトです。



SPARC のレジスタウィンドウ

SPARC プロセッサでは、レジスタウィンドウを使用します。各レジスタウィンドウは、8 個の インレジスタ、8 個の ローカルレジスタ、8 個の アウトレジスタ、および 8 個の グローバルレジスタで構成されています。アウトレジスタは、次のウィンドウのインレジスタになります。レジスタウィンドウの数は、プロセッサの実装によって 2 - 32 の範囲で変わります。

ドライバは通常 C で記述されるため、レジスタウィンドウが使用されていることは、通常、コンパイラによって隠されています。ただし、ドライバをデバッグするときには、レジスタウィンドウを使用する必要が生じる場合があります。

SPARC の乗算命令と除算命令

バージョン 7 の SPARC プロセッサには、乗算命令または除算命令はありません。乗算命令および除算命令は、ソフトウェアでエミュレートされます。ドライバはバージョン 7、バージョン 8、またはバージョン 9 のプロセッサで実行されることがあるため、負荷がかかる整数の乗算や除算は避けてください。代わりに、ビット単位の左シフトや右シフトを使用して、2 のべき乗で乗算や除算を行ってください。

『SPARC Architecture Manual, Version 9』には、SPARC CPU の詳細が記載されています。『SPARC Compliance Definition, Version 2.4』には、SPARC V9 のアプリケーションバイナリインタフェース (ABI) の詳細が記載されています。このマニュアルでは、32 ビット SPARC V8 ABI と 64 ビット SPARC V9 ABI. について説明しています。このドキュメントは SPARC International (`http://www.sparc.com`) から取得できます。

x86 プロセッサの問題

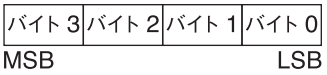
データ型に割り当ての制限はありません。ただし、割り当てが正しくないデータ転送を x86 プロセッサで適正に処理するには、追加のメモリーサイクルが必要になることがあります。

注 – カーネルでは浮動小数点演算はサポートされていないため、ドライバでこの演算を実行しないでください。

---

x86 のバイト順序

x86 プロセッサでは、リトルエンディアンのバイト順序を使用します。整数の最下位バイト (LSB) は、その整数の一番低いアドレスに格納されます。最上位バイトは、このプロセッサのデータ項目の一番高いアドレスに格納されます。たとえば、バイト 7 は 64 ビットプロセッサの最上位バイトです。



x86 アーキテクチャーのマニュアル

Intel Corporation も AMD も、x86 ファミリのプロセッサに関する書籍を多数出版しています。http://www.intel.com および http://www.amd.com を参照してください。

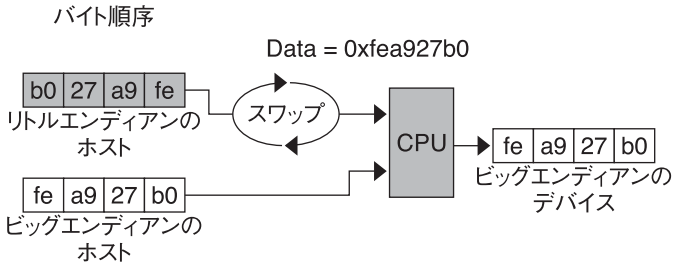
エンディアン

マルチプラットフォーム、複数命令セットアーキテクチャーの互換性の目標を達成するために、ホストバス依存性がドライバから削除されました。依存性に関して最初に取り組むべき問題は、プロセッサのエンディアン、つまりバイト順序でした。たとえば、x86 プロセッサファミリはリトルエンディアンですが、SPARC アーキテクチャーはビッグエンディアンです。

バスアーキテクチャーにも、プロセッサと同じエンディアンのタイプがあります。たとえば、PCI ローカルバスはリトルエンディアン、SBus はビッグエンディアン、ISA バスはリトルエンディアン、などです。

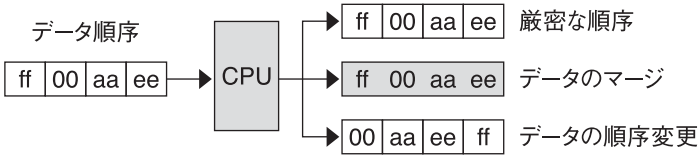
プロセッサとバスの間の互換性を維持するために、DDI 準拠のドライバはエンディアンに関して中立である必要があります。ドライバは、実行時検査や、ソースコードの `#ifdef _LITTLE_ENDIAN` などのプリプロセッサ指令によって、エンディアンを管理できますが、長期的な保守がわずらわしくなる可能性があります。場合によっては、DDI フレームワークでソフトウェアを利用してバイトスワッピングを実行します。また、メモリー管理ユニット (MMU) の場合のようにハードウェアのページレベルスワッピングによって、または特別なマシン命令によって、バイトスワッピングを実行することもできます。DDI フレームワークでは、ハードウェアの機能を活用してパフォーマンスを向上させることができます。

図 A–1 ホストバス依存性に必要なバイト順序



互換性のあるドライバは、エンディアンに関して中立であるとともに、プロセッサのデータ順序から独立している必要もあります。ほとんどの環境では、ドライバが命令する順序でデータを転送する必要があります。ただし、次の図に示すように、データをマージしたり、バッチ処理したり、順序を変更したりして、データ転送を合理化できることもあります。たとえば、データマージを適用して、フレームバッファーでグラフィックス表示を高速化できます。ドライバは、転送時にほかの最適なデータ転送メカニズムを使用するように DDI フレームワークに指示することもできます。

図 A–2 データ順序のホストバス依存性



ストアバッファー

パフォーマンスを向上させるため、CPU は内部ストアバッファーを使用して一時的にデータを格納します。内部バッファーを使用すると、デバイスの入出力処理の同期に影響が及ぶことがあります。したがって、ドライバは、明示的な手順を実行して、レジスタへの書き込みが適切なときに完了するようにする必要があります。

たとえば、レジスタやフレームバッファーなどのデバイス空間へのアクセスがロックによって同期される場合を考えてみます。ドライバは、デバイス空間への格納がロックの解放前に実際に完了したことを確認する必要があります。ロックの解放は、I/O バッファーのフラッシュを保証するものではありません。

別の例として、ドライバは通常、割り込みに対して確認応答するときに、デバイス制御レジスタのビットを設定またはクリアします。ドライバは、制御レジスタへの書き込みが割り込みハンドラの復帰前にデバイスに到達するよう保証する必要があります。同様に、デバイスに遅延が必要になることがあります。つまり、制御レジ

スタにコマンドを書き込んだあと、ドライバはビジー状態で待機します。そのような場合、ドライバは、遅延の前に書き込みがデバイスに到達するよう保証する必要があります。

好ましくない結果がなく、デバイスレジスタを読み取ることができる場合、書き込みの検証は、単純に書き込み直後のレジスタの読み取りで構成できます。好ましくない結果がなく、その特定のレジスタを読み取ることができない場合は、同じレジスタセット内の別のデバイスレジスタを使用できます。

システムのメモリーモデル

システムのメモリーモデルは、ロードやストアなどのメモリー処理のセマンティクスを定義し、それらの処理をプロセッサが実行する順序を、それらの処理がメモリーに到達する順序に関連付ける方法を指定します。メモリーモデルは、単一プロセッサとメモリー共有型マルチプロセッサの両方に適用されます。トータルストアオーダリング (TSO) とパーシャルストアオーダリング (PSO) の 2 つのメモリーモデルがサポートされています。

トータルストアオーダリング (TSO)

TSO では、ストア、フラッシュ、および原子的ロード/ストアの各命令が指定されたプロセッサのメモリーに出現する順序が、それらの命令をプロセッサが実行する順序と同じであることが保証されます。

x86 プロセッサと SPARC プロセッサの両方が TSO をサポートしています。

パーシャルストアオーダリング (PSO)

PSO では、ストア、フラッシュ、および原子的ロード/ストアの各命令が指定されたプロセッサのメモリーに出現する順序が、それらの命令をプロセッサが実行する順序と同じであることが保証されません。プロセッサはストアの順序を変更することができ、その場合、メモリーのストアの順序は CPU が実行するストアの順序と同じでなくなります。

SPARC プロセッサは PSO をサポートしていますが、x86 プロセッサはサポートしていません。

SPARC プロセッサでは、実行順序とメモリー順序の合致は、STBAR 命令を使用するシステムフレームワークによって可能になります。上の命令のうち 2 つがプロセッサの実行順序内で STBAR 命令によって分離される場合、または命令が同じ位置を参照する場合、その 2 つの命令のメモリー順序は実行順序と同じになります。DDI 準拠のドライバでの強いデータ順序の実施は、`ddi_regs_map_setup(9F)` インタフェースによって可能になります。準拠ドライバは、STBAR 命令を直接使用することはできません。

SPARC のメモリーモデルの詳細については、『SPARC Architecture Manual, Version 9』を参照してください。

バスアーキテクチャー

このセクションでは、デバイスの識別、デバイスのアドレス指定、および割り込みについて説明します。

デバイスの識別

デバイスの識別とは、システムに存在しているデバイスを判定するプロセスのことです。一部のデバイスは、自己識別を行います。つまり、デバイス自体がシステムに情報を提供して、使用する必要のあるデバイスドライバをシステムが識別できるようにします。SBus と PCI ローカルバスのデバイスは、自己識別を行うデバイスの例です。SBus では、情報は通常、デバイスの FCode PROM に格納されている小さい Forth プログラムから派生します。ほとんどの PCI デバイスには、デバイスの構成情報を含む構成スペースが用意されています。詳細については、`sbus(4)` および `pci(4)` のマニュアルページを参照してください。

新しいバスアーキテクチャーではすべて、デバイスは自己識別を行う必要があります。

サポートされている割り込みタイプ

Solaris プラットフォームは、ポーリング方式とベクター方式の両方の割り込みをサポートしています。Solaris DDI/DKI 割り込みモデルは、両方のタイプの割り込みで同じです。割り込み処理の詳細については、第 8 章「割り込みハンドラ」を参照してください。

バスの仕様

このセクションでは、Solaris プラットフォームがサポートしているバスに固有の、アドレス指定とデバイス構成の問題について説明します。

PCI ローカルバス

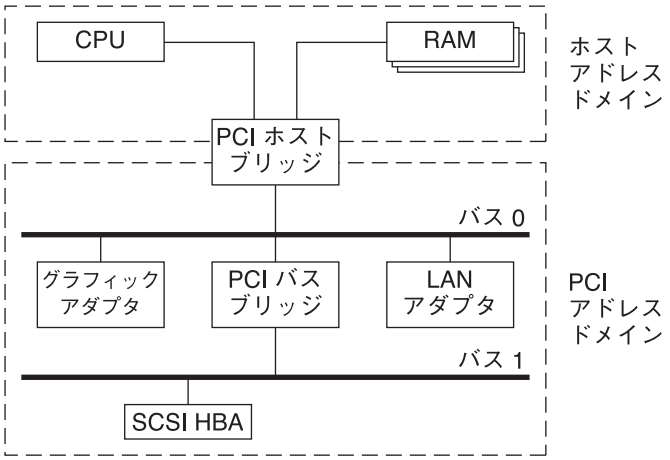
PCI ローカルバスは、高速なデータ転送のために設計された高性能なバスです。PCI バスは、システムボード上にあります。このバスは通常、高度に統合された周辺コンポーネント、周辺アドオンボード、およびホストプロセッサまたはメモリーシス

テムの間の相互接続メカニズムとして使用されます。ホストプロセッサ、メインメモリー、およびPCIバス自体は、図A–3に示すように、PCIホストブリッジを介して接続されます。

相互に接続されたI/Oバスのツリー構造は、一連のPCIバスブリッジを介してサポートされます。下位のPCIバスブリッジをPCIホストブリッジの下で拡張し、単一のバスシステムを拡張して、複数のセカンダリバスを持つ複雑なシステムにできます。PCIデバイスは、これらの1つ以上のセカンダリバスに接続できます。また、SCSIやUSBなどのほかのバスブリッジも接続できます。

すべてのPCIデバイスには、一意なベンダーIDとデバイスIDがあります。同じ種類の複数のデバイスは、デバイスが存在するバスの一意のデバイス番号によってさらに識別されます。

図A–3　マシンのブロック図



PCIホストブリッジによって、プロセッサと周辺コンポーネントが相互に接続されます。プロセッサは、PCIホストブリッジを介して、ほかのPCIバスマスターから独立して直接メインメモリーにアクセスできます。たとえば、CPUがホストブリッジのキャッシュコントローラからデータを取得している間に、ほかのPCIデバイスもホストブリッジを介してシステムメモリーにアクセスできます。このアーキテクチャーの利点は、このアーキテクチャーによってI/Oバスとプロセッサのホストバスが分離されることです。

PCIホストブリッジによって、CPUと周辺入出力デバイスの間のデータアクセスマッピングも提供されます。ブリッジは、すべての周辺デバイスをホストアドレスドメインにマッピングして、プロセッサがプログラム式入出力経由でデバイスにアクセスできるようにします。ローカルバス側では、PCIホストブリッジはシステムメ

モリーを PCI アドレスドメインにマッピングして、PCI デバイスがバスマスターとしてホストメモリーにアクセスできるようにします。図 A–3 は、2 つのアドレスドメインを示しています。

PCI アドレスドメイン

PCI アドレスドメインは、構成、メモリー、および I/O 空間という 3 つの個別のアドレス空間で構成されています。

PCI 構成アドレス空間

構成空間は地理的に定義されます。周辺デバイスの位置は、PCI バスブリッジの相互に接続されたツリー内のその物理的な位置によって決定されます。デバイスは、そのバス番号とデバイス (スロット) 番号によって検出されます。各周辺デバイスには、その PCI 構成空間に一連の十分に定義された構成レジスタが含まれています。レジスタは、デバイスを識別するためだけでなく、構成フレームワークにデバイス構成情報を提供するためにも使用されます。たとえば、デバイスがデータアクセスに応答するためには、デバイス構成空間の基底アドレスレジスタをマッピングする必要があります。

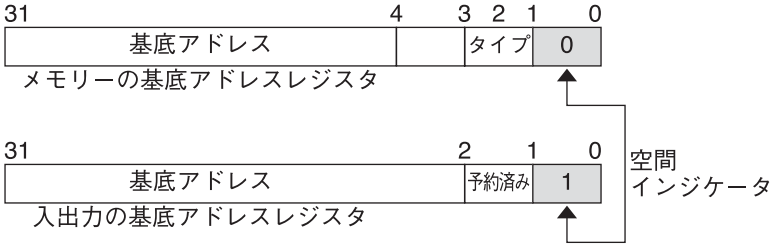
構成サイクルを生成するための方法はホストに依存しています。x86 マシンでは、特殊な I/O ポートが使用されます。ほかのプラットフォームでは、ホストアドレスドメイン内の PCI ホストブリッジに応じて、PCI 構成空間を特定のアドレス位置にメモリーマッピングできます。デバイス構成レジスタにプロセッサがアクセスすると、要求が PCI ホストブリッジにルーティングされます。次に、ブリッジは、そのアクセスをバスの適切な構成サイクルに変換します。

PCI 構成基底アドレスレジスタ

PCI 構成空間は、デバイスごとに最大 6 つの 32 ビット基底アドレスレジスタで構成されています。これらのレジスタは、サイズとデータ型の両方の情報を提供します。システムファームウェアは、PCI アドレスドメインの基底アドレスをこれらのレジスタに割り当てます。

各アドレス指定可能領域には、メモリーまたは I/O 空間を使用できます。基底アドレスレジスタのビット 0 に含まれる値によってタイプが識別されます。ビット 0 にある 0 の値はメモリー空間を示し、1 の値は I/O 空間を示します。次の図は、2 つの基底アドレスレジスタを示しています。1 つはメモリータイプ用で、もう 1 つは入出力タイプ用です。

図A–4 メモリーおよび入出力の基底アドレスレジスタ



PCIメモリーアドレス空間

PCIは、メモリー空間について32ビットと64ビットの両方のアドレスをサポートしています。システムファームウェアは、PCIアドレスドメインのメモリー空間の領域をPCI周辺デバイスに割り当てます。領域の基底アドレスは、デバイスのPCI構成空間の基底アドレスレジスタに格納されます。各領域のサイズは2のべき乗にする必要があり、割り当てられる基底アドレスは領域のサイズと等しい境界で割り当てる必要があります。メモリー空間のデバイスアドレスはホストアドレスドメインにメモリーマッピングされるため、デバイスへのデータアクセスはプロセッサのネイティブのロード命令またはストア命令によって実行できます。

PCI I/Oアドレス空間

PCIは、32ビットI/O空間をサポートしています。I/O空間にアクセスする方法はプラットフォームによって異なります。Intelプロセッサファミリなどの特殊な入出力命令を持つプロセッサは、`in`および`out`命令を使ってI/O空間にアクセスします。特殊な入出力命令を持たないマシンは、ホストアドレスドメインのPCIホストブリッジに応じてアドレス位置にマッピングします。プロセッサがメモリーマッピングされたアドレスにアクセスすると、入出力要求がPCIホストブリッジに送られ、次にアドレスが入出力サイクルに変換されてPCIバスに配置されます。メモリーマッピングされた入出力は、プロセッサのネイティブのロード/ストア命令によって実行されます。

PCIハードウェア構成ファイル

ハードウェア構成ファイルは、PCIローカルバスデバイスには必要ないはずです。ただし、場合によっては、PCIデバイスのドライバはハードウェア構成ファイルを使用してドライバの固有情報を増やす必要があります。詳細は、`driver.conf(4)`および`pci(4)`のマニュアルページを参照してください。

PCI Express

標準の PCI バスは、PCI Express に発展しています。PCI Express は、デスクトップ、モバイル、ワークステーション、サーバー、埋め込み型コンピューティング/通信プラットフォームなどのアプリケーションで周辺デバイスを接続するための、次世代の高性能 I/O バスです。

PCI Express ではバスパフォーマンスが向上し、全体的なシステムコストが減少し、コンピュータ設計の新しい発展を活用しています。PCI Express では、2 つのデバイス間の通信にシリアルのポイントツーポイント型相互接続を使用します。スイッチの使用により、ユーザーはシステム内の多数のデバイスを一緒に接続できます。シリアルの相互接続では、デバイスパッケージあたりのピンが少なくなるので、コストが減少し、パフォーマンスが高度にスケーラブルになります。

PCI Express バスには、次の技術に対応する機能が組み込まれています。

- QoS (Quality of Service)
- ホットプラグによる取り付けとホットスワップ
- 詳細な電源管理
- RAS (信頼性、可用性、保守性)
- 向上したエラー処理
- MSI 割り込み

2 つのデバイスを一緒に接続する PCI Express の相互接続は、リンクと呼ばれます。リンクでは、x1、x2、x4、x8、x12、x16、または x32 の双方向のシグナルペアを使用できます。これらのシグナルは、レーンと呼ばれます。各レーンの帯域幅 (x1) は、全二重モードで 500M バイト/秒です。PCI-X と PCI Express のハードウェア接続は異なりますが、2 つのバスはドライバの作成者の観点からは同じです。PCI-X は共有バスです。たとえば、バス上のすべてのデバイスが 1 つのセットのデータラインとシグナルラインを共有します。PCI-Express はスイッチバスであり、デバイスとシステムバスの間で帯域幅を使用するときの効率を向上させることができます。

PCI Express の詳細については、Web サイト `http://www.pcisig.com/home` を参照してください。

SBus

通常の SBus システムは、マザーボード (CPU と SBus インタフェースロジックを含む)、マザーボード自体にあるいくつかの SBus デバイス、およびいくつかの SBus 拡張スロットで構成されています。SBus は、適切なバスブリッジを介して、ほかのタイプのバスに接続することもできます。

SBus は地理的にアドレス指定されます。各 SBus スロットは、システム内の固定した物理アドレスに存在します。SBus カードは、差し込むスロットによってアドレスが異なります。SBus デバイスを新しいスロットに移動すると、システムはこのデバイスを新しいデバイスと見なします。

SBus では、ポーリング方式の割り込みを使用します。SBus デバイスが割り込みを行っても、システムは、いくつかあるデバイスのどれかが割り込みを実行した可能性があることしか認識できません。システムの割り込みハンドラは、各デバイスのドライバに、そのデバイスが割り込みを行ったかどうか問い合わせる必要があります。

SBus の物理アドレス空間

次の表は、UltraSPARC 2 コンピュータの物理アドレス空間のレイアウトを示しています。UltraSPARC 2 モデルの物理アドレスは、41 ビットで構成されています。41 ビットの物理アドレス空間はさらに、`PA(40:33)` によって識別される複数の 33 ビットアドレス空間に細分化されます。

表 A–1　Ultra 2 のデバイス物理空間

PA(40:33)	33 ビット空間	用途
0x0	`0x000000000 - 0x07FFFFFF`	2G バイトのメインメモリー
0x80 – 0xDF	`Reserved on Ultra 2`	Ultra 2 で予約済み
0xE0	`Processor 0`	プロセッサ 0
0xE1	`Processor 1`	プロセッサ 1
0xE2 – 0xFD	`Reserved on Ultra 2`	Ultra 2 で予約済み
0xFE	`0x000000000 - 0x1FFFFFFF`	UPA スレーブ (FFB)
0xFF	`0x000000000 - 0x0FFFFFFF`	システム I/O 空間
	`0x100000000 - 0x10FFFFFF`	SBus スロット 0
	`0x110000000 - 0x11FFFFFF`	SBus スロット 1
	`0x120000000 - 0x12FFFFFF`	SBus スロット 2
	`0x130000000 - 0x13FFFFFF`	SBus スロット 3
	`0x1D0000000 - 0x1DFFFFFF`	SBus スロット D
	`0x1E0000000 - 0x1EFFFFFF`	SBus スロット E
	`0x1F0000000 - 0x1FFFFFFF`	SBus スロット F

物理 SBus アドレス

SBus には、『SBus Specification』で説明されているように、32 のアドレスビットがあります。次の表で、Ultra 2 でアドレスビットを使用する方法について説明します。

表 A–2　Ultra 2 の SBus のアドレスビット

ビット	説明
0 - 27	これらのビットは、SBus カードがその内容をアドレス指定するために使用する SBus アドレスラインです。
28 - 31	SBus スロットの 1 つを選択するために CPU で使用します。これらのビットは、SlaveSelect ラインを生成します。

このアドレス指定方式により、表 A–1 に示す Ultra 2 アドレスが生成されます。ほかの実装では、異なる数のアドレスビットを使用することがあります。

Ultra 2 には 7 つの SBus スロットがあり、そのうちの 4 つは物理的です。スロット 0 - 3 は、SBus カードで使用できます。スロット 4 - 12 は、予約されています。これらのスロットは、次のように使用します。

- スロット 0 - 3 は、DMA マスター機能のある物理スロットです。
- スロット D、E、および F は、実際の物理スロットではありませんが、オンボードのダイレクトメモリーアクセス (DMA)、SCSI、Ethernet、およびオーディオコントローラを参照します。便宜上、これらのクラスのデバイスは、スロット D、E、および F に差し込まれているものと見なされます。

注 – 一部の SBus スロットは、スレーブのみのスロットです。DMA 機能が必要なドライバでは、`ddi_slaveonly(9F)` を使用して、デバイスが DMA 対応のスロットに入っているかどうかを判定するべきです。この関数の例については、109 ページの「attach() エントリポイント」を参照してください。

SBus のハードウェア構成ファイル

ハードウェア構成ファイルは通常、SBus デバイスには必要ありません。ただし、場合によっては、SBus デバイスのドライバは、ハードウェア構成ファイルを使用して、SBus カードが提供する情報を増やす必要があります。詳細は、`driver.conf(4)` および `sbus(4)` のマニュアルページを参照してください。

デバイスの問題

このセクションでは、特殊なデバイスの問題について説明します。

タイミングクリティカルセクション

ほとんどのドライバの処理は、ロックプリミティブによって提供されるものを超える同期や保護のメカニズムなしで実行できますが、一部のデバイスでは一連のイベントが割り込みなしで順番に発生する必要があります。ロックプリミティブに関連

して、関数 ddi_enter_critical(9F) は、現在のスレッドが横取りされたり割り込まれたりしないことを最大限に保証するようにシステムに求めます。この保証は、閉じるための呼び出しを ddi_exit_critical(9F) に対して行うまで有効です。詳細は、ddi_enter_critical(9F)のマニュアルページを参照してください。

遅延

多くのチップは、それらのチップが指定された間隔でのみアクセスできるように指定します。たとえば、Zilog Z8530 SCC には、1.6 マイクロ秒の「書き込み回復時間」があります。この仕様は、8530 で文字を書き込むときに drv_usecwait(9F) で遅延を実施する必要があることを意味します。場合によっては、この仕様では必要な遅延が明示されないため、経験に基づいて遅延を判定する必要があります。

たとえば、幾千もの SCSI ディスクドライブなど、数多く存在することがあるデバイスのパーツの遅延を大きくしないように注意してください。

内部順序付けロジック

内部順序付けロジックのあるデバイスは、複数の内部レジスタを同じ外部アドレスにマッピングします。さまざまな種類の内部順序付けロジックには、次のタイプが含まれています。

- Intel 8251A と Signetics 2651 は、*2* つの内部モードレジスタの間で同じ外部レジスタを交互に使用します。最初の内部レジスタへの書き込みは、外部レジスタへの書き込みによって完成します。ただし、この書き込みには、チップの順序付けロジックが設定され、次の読み取り/書き込み処理で 2 番目の内部レジスタを参照するという想定外の結果が伴います。
- NEC PD7201 PCC には、複数の内部データレジスタがあります。特定のレジスタにバイトを書き込むには、2 つの手順を実行する必要があります。最初の手順は、後続のデータバイトが入るレジスタの番号をレジスタ 0 に書き込むことです。次に、データは、指定されたデータレジスタに書き込まれます。順序付けロジックによって、次に送信されるバイトがデータレジスタ 0 に入るようにチップが自動的に設定されます。
- AMD 9513 タイマーには、データバイトが入るデータレジスタをポイントするデータポインタレジスタがあります。データレジスタにバイトを送信すると、ポインタは増分されます。ポインタレジスタの現在の値を読み取ることはできません。

割り込みの問題

次の一般的な割り込みの問題に注意してください。

- コントローラによる割り込みは、必ずしもコントローラとそのスレーブデバイスの 1 つの両方の準備ができたことを示すものではありません。一部のコントローラでは、割り込みは、両方の準備ができたことではなく、コントローラの準備ができたか、またはそのデバイスの 1 つの準備ができたかのいずれかを示すことがあります。
- 割り込みを無効にしてもすべてのデバイスのパフォーマンスが向上するわけではありません。また、すべてのデバイスが任意の時点で割り込みを開始できるわけでもありません。
- 一部のデバイスでは、ボードが割り込みを生成したことを判定する方法が用意されていません。
- 割り込みを停止するように指示されたときに、またはバスがリセットされたあとで、すべての割り込みボードが割り込みを停止するわけではありません。

SPARC マシンの PROM

一部のプラットフォームには、オペレーティングシステムなしでデバイスのデバッグを行うためのサポートを提供する PROM モニターがあります。このセクションでは、SPARC マシンで PROM を使してデバイスレジスタをマッピングし、それらのデバイスレジスタにアクセスできるようにする方法について説明します。通常、デバイスは、PROM コマンドを使って十分に動作テストを実行し、正しく動作するかどうかを判定できます。

x86 のブートサブシステムについては、boot(1M)のマニュアルページを参照してください。

PROM には、次のようないくつかの目的があります。

- 電源を入れてから、またはハードリセットの PROM reset コマンドからマシンを起動する
- メモリー、デバイスレジスタ、およびメモリーマッピングを調べたり設定したりする対話型のツールを提供する
- Solaris システムをブートする

  単にコンピュータを拡張し、その PROM を使用してデバイスレジスタを調べようとしても、失敗することがあります。デバイスが正しく取り付けられていても、マッピングは Solaris OS 固有のものであり、Solaris カーネルがブートされるまではアクティブになりません。拡張時に、PROM は、キーボードなどの基本的なシステムデバイスだけをマッピングします。
- sync コマンドを使用して、システムクラッシュダンプを実行する

Open Boot PROM 3

Open Boot PROM の完全なドキュメントについては、『Open Boot PROM Toolkit User's Guide』および monitor(1M)のマニュアルページを参照してください。このセクションの例では、Sun4U アーキテクチャーを参照しています。ほかのアーキテクチャーでは、アクションの実行に別のコマンドが必要になることがあります。

注 – Open Boot PROM は現在、SBus または UPA/PCI を搭載した Oracle マシンで使用されています。Open Boot PROM では、「ok」プロンプトを使用します。古いマシンでは、「n」を入力して「ok」プロンプトを表示する必要が生じる場合があります。

PROM がセキュアモード (security-mode パラメータがなしに設定されていない) になっている場合、PROM パスワードを要求されることがあります (security-password パラメータで設定)。

printenv コマンドは、すべてのパラメータとその値を表示します。

ヘルプは、help コマンドで利用できます。

EMACS スタイルのコマンド行履歴を利用できます。履歴リストをたどるときは、Control + N (次) および Control + P (前) を使用します。

Forth コマンド

Open Boot PROM では、Forth というプログラミング言語を使用します。Forth は、スタックベースの言語です。引数は、正しいコマンドを実行する前にスタック (ワードと呼ばれる) にプッシュする必要があります。結果はスタックに残ります。

数値をスタックに配置するには、その値を入力します。

```
ok 57
ok 68
```

上の 2 つの値をスタックに追加するには、+ 演算子を使用します。

```
ok +
```

結果はスタックに残ります。スタックは .s ワードで表示されます。

```
ok .s
bf
```

デフォルトの基数は 16 進数です。hex と decimal の各ワードを使用すると、基数を切り替えることができます。

```
ok decimal
ok .s
191
```

詳細は、Forth のユーザーガイドを参照してください。

PROM デバイスツリーの調査

コマンド pwd、cd、および ls は、PROM デバイスツリーを調べてデバイスに到達します。pwd が機能する前にツリー内の位置を確立するには、cd コマンドを使用する必要があります。次の例は、SBus で cgsix フレームバッファーを使用する Ultra 1 ワークステーションのものです。

```
ok cd /
```

ツリー内の現在のノードに接続されているデバイスを表示するには、ls を使用します。

```
ok ls
f006a064 SUNW,UltraSPARC@0,0
f00598b0 sbus@1f,0
f00592dc counter-timer@1f,3c00
f004eec8 virtual-memory
f004e8e8 memory@0,0
f002ca28 aliases
f002c9b8 options
f002c880 openprom
f002c814 chosen
f002c7a4 packages
```

完全なノード名を使用できます。

```
ok cd sbus@1f,0
ok ls
f006a4e4 cgsix@2,0
f0068194 SUNW,bpp@e,c800000
f0065370 ledma@e,8400010
f006120c espdma@e,8400000
f005a448 SUNW,pll@f,1304000
f005a394 sc@f,1300000
f005a24c zs@f,1000000
f005a174 zs@f,1100000
f005a0c0 eeprom@f,1200000
f0059f8c SUNW,fdtwo@f,1400000
f0059ec4 flashprom@f,0
f0059e34 auxio@f,1900000
f0059d28 SUNW,CS4231@d,c000000
```

前の例で完全なノード名を使用する代わりに、省略名を使用することもできます。省略したコマンド行エントリは、次の例のようになります。

```
ok cd sbus
```

この名前は、実際には device@slot,offset (SBus デバイスの場合) です。cgsix デバイスはスロット 2 にあり、オフセット 0 から始まります。SBus デバイスがこのツリーに表示される場合、そのデバイスは PROM に認識されています。

.properties コマンドは、デバイスの PROM プロパティーを表示します。これらのプロパティーを調べることで、デバイスがエクスポートするプロパティーを判定できます。この情報は、あとでドライバが正しいハードウェアプロパティーを検索していることを確認するときに役立ちます。これらのプロパティーは、ddi_getprop(9F) で取得できるプロパティーと同じです。

```
ok cd cgsix
ok .properties
character-set            ISO8859-1
intr                     00000005 00000000
interrupts               00000005
reg                      00000002 00000000 01000000
dblbuf                   00 00 00 00
vmsize                   00 00 00 01
...
```

reg プロパティーは、次のフィールドを含むレジスタ記述構造体の配列を定義します。

```
uint_t        bustype;       /* cookie for related bus type*/
uint_t        addr;          /* address of reg relative to bus */
uint_t        size;          /* size of this register set */
```

cgsix の例の場合、アドレスは 0 です。

デバイスのマッピング

デバイスは、テストするメモリーにマッピングする必要があります。次に、PROM を使用して、データ転送コマンドを使用し、バイト、ワード、およびロングワードを転送することで、デバイスの適切な動作を確認できます。PROM から限定的にでもデバイスを操作できる場合は、ドライバもデバイスを操作できるはずです。

初期テスト用にデバイスを設定するには、次の手順を実行します。

1. デバイスが存在する SBus スロット番号を判定します。

   この例では、cgsix デバイスはスロット 2 にあります。

2. デバイスが使用する物理アドレス空間内のオフセットを判定します。

   使用されるオフセットは、デバイスに固有です。cgsix の例では、ビデオメモリーは 0x800000 のオフセットから始まっています。

3. Sbus デバイスを選択するには、select-dev ワードを使用し、デバイスを内部にマッピングするには、map-in ワードを使用します。

   select-dev ワードは、デバイスパスの文字列をその引数と見なします。map-in ワードは、オフセット、スロット番号、およびサイズを、マッピングする引数と見なします。オフセットと同様、バイト転送のサイズはデバイスに固有です。cgsix の例では、サイズは 0x100000 バイトに設定されています。

次のコード例では、Sbus のパスが select-dev ワードの引数として表示されており、フレームバッファーのオフセット、スロット番号、およびサイズの値が map-in ワードの引数として表示されています。select-dev 引数内の開始引用符と / の間の空白に注意してください。使用する仮想アドレスは、スタックの上位に残っています。スタックは .s ワードを使って表示されます。スタックには、constant 処理で名前を割り当てることができます。

```
ok " sbus@1f,0" select-dev
ok 800000 2 100000 map-in
ok .s
ffe98000
ok constant fb
```

読み取りと書き込み

PROM には、8 ビット、16 ビット、および 32 ビットのさまざまな処理が用意されています。一般に、c (文字) 接頭辞は 8 ビット (1 バイト) の処理、w (ワード) 接頭辞は 16 ビット (2 バイト) の処理、L (ロングワード) 接頭辞は 32 ビット (4 バイト) の処理を示します。

接頭辞 ! は、書き込み処理を示します。書き込み処理では、スタックの最初の 2 つの項目を使用します。最初の項目はアドレスで、2 番目の項目は値です。

```
ok 55 ffe98000 c!
```

接頭辞 @ は、読み取り処理を示します。読み取り処理では、スタックのアドレスを使用します。

```
ok ffe98000 c@
ok .s
55
```

接頭辞 ? は、スタックに影響を与えないように値を表示するときに使用します。

```
ok ffe98000 c?
55
```

デバイスにクエリーしようとするときは注意してください。マッピングが正しく設定されていない場合は、読み取りまたは書き込みを行おうとすると、エラーが発生することがあります。このようなケースを扱うときは、特殊なワードを指定します。たとえば、cprobe、wprobe、および lprobe は、指定されたアドレスから読み取りますが、その場所から応答がない場合はゼロを返し、応答がある場合はゼロ以外を返します。

```
ok fffa4000 c@
Data Access Error

ok fffa4000 cprobe
```

```
ok .s0

ok ffe98000 cprobe
ok .s
0 ffffffffffffffff
```

メモリーの領域は、dump ワードで表示できます。このワードではアドレスと長さを使用し、メモリー領域の内容をバイト単位で表示します。

次の例では、fill ワードを使用してビデオメモリーをパターンで埋めます。fill では、アドレス、埋めるバイト数、および使用するバイトを使います。ワードおよびロングワードの場合は、wfill および Lfill を使用します。次の fill の例では、cgsix によって、渡されたバイトに基づいて単純なパターンが表示されます。

```
ok " /sbus" select-dev
ok 800000 2 100000 map-in
ok constant fb
ok fb 10000 ff fill
ok fb 20000 0 fill
ok fb 18000 55 fill
ok fb 15000 3 fill
ok fb 10000 5 fillok fb 5000 f9 fill
```

◆ ◆ ◆ 付 録 B

Solaris DDI/DKI サービスのサマリー

この付録では、Solaris DDI/DKI によって提供されるインタフェースについて説明します。これらの説明を、完全または最終的なものとは見なさないでください。また、使用法を網羅したガイドが提供されるわけでもありません。これらの説明は、関数が実行する内容を一般的に記述することを目的としています。詳細は、`physio(9F)` を参照してください。カテゴリは次のとおりです。

- 647 ページの「リソースマップ管理関数」
- 647 ページの「システムのグローバル状態」
- 647 ページの「ユーティリティー関数」

モジュール関数

モジュール関数には次のものがあります。

`mod_info`	ロード可能なモジュールをクエリーする
`mod_install`	ロード可能なモジュールを追加する
`mod_remove`	ロード可能なモジュールを削除する

デバイス情報ツリーノード (dev_info_t) 関数

デバイス情報ツリーノード関数には次のものがあります。

`ddi_binding_name()`	ドライバのバインディング名を返す
`ddi_dev_is_sid()`	デバイスが自身を識別できるかどうかを指示する
`ddi_driver_major()`	ドライバのメジャーデバイス番号を返す
`ddi_driver_name()`	正規化されたドライバ名を返す
`ddi_node_name()`	`devinfo` ノード名を返す
`ddi_get_devstate()`	デバイスの状態をチェックする
`ddi_get_instance()`	デバイスインスタンス番号を取得する
`ddi_get_name()`	ドライバのバインディング名を返す
`ddi_get_parent()`	デバイス情報構造の親を検索する
`ddi_root_node()`	`dev_info` ツリーのルートを取得する

デバイス (dev_t) 関数

デバイス関数には次のものがあります。

`ddi_create_minor_node()`	デバイスのマイナーノードを作成する
`ddi_getiminor()`	外部の `dev_t` からカーネル内部のマイナー番号を取得する
`ddi_remove_minor_node()`	デバイスのマイナーモードを削除する

getmajor()	メジャーデバイス番号を取得する
getminor()	マイナーデバイス番号を取得する
makedevice()	メジャー番号とマイナー番号からデバイス番号を作成する

プロパティー関数

プロパティー関数には次のものがあります。

ddi_prop_exists()	プロパティーの存在をチェックする
ddi_prop_free()	プロパティー検索で消費されたリソースを解放する
ddi_prop_get_int()	整数プロパティーを検索する
ddi_prop_get_int64()	64 ビットの整数プロパティーを検索する
ddi_prop_lookup_byte_array()	バイト配列プロパティーを検索する
ddi_prop_lookup_int_array()	整数配列プロパティーを検索する
ddi_prop_lookup_int64_array()	64 ビットの整数配列プロパティーを検索する
ddi_prop_lookup_string()	文字列プロパティーを検索する
ddi_prop_lookup_string_array()	文字列配列プロパティーを検索する
ddi_prop_remove()	デバイスのプロパティーを削除する
ddi_prop_remove_all()	デバイスのすべてのプロパティーを削除する
ddi_prop_undefine()	デバイスのプロパティーを非表示にする
ddi_prop_update_byte_array()	バイト配列プロパティーを作成または更新する
ddi_prop_update_int()	整数プロパティーを作成または更新する
ddi_prop_update_int64()	64 ビットの整数プロパティーを作成または更新する
ddi_prop_update_int_array()	整数配列プロパティーを作成または更新する
ddi_prop_update_int64_array()	64 ビットの整数配列プロパティーを作成または更新する
ddi_prop_update_string()	文字列プロパティーを作成または更新する
ddi_prop_update_string_array()	文字列配列プロパティーを作成または更新する

表 B–1 非推奨のプロパティー関数

非推奨の関数	代わりの関数
ddi_getlongprop()	ddi_prop_lookup() を参照
ddi_getlongprop_buf()	ddi_prop_lookup()
ddi_getprop()	ddi_prop_get_int()
ddi_getproplen()	ddi_prop_lookup()
ddi_prop_create()	ddi_prop_lookup()
ddi_prop_modify()	ddi_prop_lookup()
ddi_prop_op()	ddi_prop_lookup()

デバイスソフトウェア状態関数

デバイスソフトウェア状態関数には次のものがあります。

ddi_get_driver_private() デバイスの非公開データ領域のアドレスを取得する

ddi_get_soft_state() インスタンスのソフト状態構造体へのポインタを取得する

ddi_set_driver_private() デバイスの非公開データ領域のアドレスを設定する

ddi_soft_state_fini() ドライバのソフト状態構造体を破棄する

ddi_soft_state_free() インスタンスのソフト状態構造体を解放する

ddi_soft_state_init() ドライバのソフト状態構造体を初期化する

ddi_soft_state_zalloc() インスタンスのソフト状態構造体を割り当てる

メモリー割り当ておよび解放関数

メモリー割り当ておよび解放関数には次のものがあります。

kmem_alloc() カーネルメモリーを割り当てる

kmem_free() カーネルメモリーを解放する

kmem_zalloc() ゼロに初期化されたカーネルメモリーを割り当てる

次の関数は、DMA に使用されるメモリーを割り当ておよび解放します。634 ページの「ダイレクトメモリーアクセス (DMA) 関数」を参照してください。

ddi_dma_mem_alloc() DMA 転送のためのメモリーを割り当てる

`ddi_dma_mem_free()` 以前に割り当てられた DMA メモリーを解放する

次の関数は、ユーザー空間にエクスポートされるメモリーを割り当ておよび解放します。636 ページの「ユーザー空間アクセス関数」を参照してください。

`ddi_umem_alloc()` ページ境界割り当てされたカーネルメモリーを割り当てる

`ddi_umem_free()` ページ境界割り当てされたカーネルメモリーを解放する

表 B–2 非推奨のメモリー割り当ておよび解放関数

非推奨の関数	代わりの FEATURE
`ddi_iopb_alloc()`	`ddi_dma_mem_alloc()`
`ddi_iopb_free()`	`ddi_dma_mem_free()`
`ddi_mem_alloc()`	`ddi_dma_mem_alloc()`
`ddi_mem_free()`	`ddi_dma_mem_free()`

カーネルスレッド制御および同期関数

カーネルスレッド制御および同期関数には次のものがあります。

`cv_broadcast()` すべての待機スレッドを呼び起こす

`cv_destroy()` 割り当てられた条件変数を解放する

`cv_init()` 条件変数を割り当てる

`cv_signal()` 1 つの待機スレッドを呼び起こす

`cv_timedwait()` タイムアウトを使用してイベントを待機する

`cv_timedwait_sig()` タイムアウトを使用してイベントまたはシグナルを待機する

`cv_wait()` イベントを待機する

`cv_wait_sig()` イベントまたはシグナルを待機する

`ddi_can_receive_sig()` 現在のスレッドがシグナルを受信できるかどうかを判定する

`ddi_enter_critical()` 制御のクリティカルリージョンに入る

`ddi_exit_critical()` 制御のクリティカルリージョンを出る

`mutex_destroy()` 相互排他ロックを破棄する

`mutex_enter()` 相互排他ロックを取得する

`mutex_exit()` 相互排他ロックを解放する

mutex_init()	相互排他ロックを初期化する
mutex_owned()	現在のスレッドが相互排他ロックを保持しているかどうかを判定する
mutex_tryenter()	待機することなく相互排他ロックの取得を試みる
rw_destroy()	読み取り/書き込みロックを破棄する
rw_downgrade()	読み取り/書き込みロックの保持を書き込みから読み取りに降格する
rw_enter()	読み取り/書き込みロックを取得する
rw_exit()	読み取り/書き込みロックを解放する
rw_init()	読み取り/書き込みロックを初期化する
rw_read_locked()	読み取り/書き込みロックが読み取りと書き込みのどちらの目的で保持されているかを調べる
rw_tryenter()	待機することなく読み取り/書き込みロックの取得を試みる
rw_tryupgrade()	読み取り/書き込みロックを保持の読み取りから書き込みに昇格しようと試みる
sema_destroy()	セマフォーを破棄する
sema_init()	セマフォーの初期化
sema_p()	セマフォーを 1 減らし、ブロックする可能性がある
sema_p_sig()	セマフォーを 1 減らすが、シグナルが保留中の場合はブロックしない
sema_tryp()	セマフォーを 1 減らそうと試みるが、ブロックしない
sema_v()	セマフォーを 1 増やし、待機中スレッドをブロック解除する可能性がある

タスクキュー管理関数

タスクキュー管理関数には次のものがあります。これらのインタフェースの詳細については、taskq(9F) のマニュアルページを参照してください。

ddi_taskq_create()	タスクキューを作成する
ddi_taskq_destroy()	タスクキューを破棄する
ddi_taskq_dispatch()	タスクキューにタスクを追加する

ddi_taskq_wait()	保留中のタスクが完了するまで待つ
ddi_taskq_suspend()	タスクキューを一時停止する
ddi_taskq_suspended()	タスクキューが一時停止されているかどうかをチェックする
ddi_taskq_resume()	一時停止されたタスクキューを再開する

割り込み関数

割り込み関数には次のものがあります。

ddi_intr_add_handler(9F)	割り込みハンドラを追加します。
ddi_intr_add_softint(9F)	ソフト割り込みハンドラを追加します。
ddi_intr_alloc(9F)	指定したタイプの割り込みのシステムリソースと割り込みベクターを割り当てます。
ddi_intr_block_disable(9F)	指定した範囲の割り込みを無効にします。MSI の場合のみ。
ddi_intr_block_enable(9F)	指定した範囲の割り込みを有効にします。MSI の場合のみ。
ddi_intr_clr_mask(9F)	指定した割り込みが有効になっている場合に、割り込みマスクをクリアします。
ddi_intr_disable(9F)	指定した割り込みを無効にします。
ddi_intr_dup_handler(9F)	MSI-X とともにのみ使用します。割り当てられた割り込みベクターのアドレスとデータのペアを、同じデバイスの未使用の割り込みベクターにコピーします。
ddi_intr_enable(9F)	指定した割り込みを有効にします。
ddi_intr_free(9F)	指定した割り込みハンドルのシステムリソースと割り込みベクターを解放します。
ddi_intr_get_cap(9F)	指定した割り込みの割り込み許可フラグを返します。
ddi_intr_get_hilevel_pri(9F)	高レベルの割り込みの最小優先順位レベルを返します。
ddi_intr_get_navail(9F)	特定のハードウェアデバイスと指定された割り込みタイプで使用できる割り込みの数を返します。

ddi_intr_get_nintrs(9F)	指定した割り込みタイプでデバイスがサポートしている割り込みの数を取得します。
ddi_intr_get_pending(9F)	割り込み中断ビットがホストブリッジまたはデバイスでサポートされている場合に、そのビットを読み取ります。
ddi_intr_get_pri(9F)	指定した割り込みの現在のソフトウェア優先順位設定を返します。
ddi_intr_get_softint_pri(9F)	指定した割り込みのソフト割り込み優先順位を返します。
ddi_intr_get_supported_types(9F)	デバイスとホストの両方でサポートされているハードウェア割り込みのタイプを返します。
ddi_intr_remove_handler(9F)	指定した割り込みハンドラを削除します。
ddi_intr_remove_softint(9F)	指定したソフト割り込みハンドラを削除します。
ddi_intr_set_cap(9F)	指定した割り込みの DDI_INTR_FLAG_LEVEL または DDI_INTR_FLAG_EDGE フラグを設定します。
ddi_intr_set_mask(9F)	指定した割り込みが有効になっている場合に、割り込みマスクを設定します。
ddi_intr_set_pri(9F)	指定した割り込みの割り込み優先順位レベルを設定します。
ddi_intr_set_softint_pri(9F)	指定したソフト割り込みの相対ソフト割り込み優先順位を変更します。
ddi_intr_trigger_softint(9F)	指定したソフト割り込みをトリガーします。

新しいフレームワークの機能を利用するには、上のインタフェースを使用してください。次の表に示されている非推奨のインタフェースを使用しないでください。これらの非推奨のインタフェースは、互換性のためにのみ保持されています。

表 B–3 非推奨の割り込み関数

非推奨の割り込み関数	代わりの関数
ddi_add_intr(9F)	3 ステップの処理: 1. ddi_intr_alloc(9F) 2. ddi_intr_add_handler(9F) 3. ddi_intr_enable(9F)

表 B–3 非推奨の割り込み関数 (続き)

非推奨の割り込み関数	代わりの関数
ddi_add_softintr(9F)	ddi_intr_add_softint(9F)
ddi_dev_nintrs(9F)	ddi_intr_get_nintrs(9F)
ddi_get_iblock_cookie(9F)	3 ステップの処理: 1. ddi_intr_alloc(9F) 2. ddi_intr_get_pri(9F) 3. ddi_intr_free(9F)
ddi_get_soft_iblock_cookie(9F)	3 ステップの処理: 1. ddi_intr_add_softint(9F) 2. ddi_intr_get_softint_pri(9F) 3. ddi_intr_remove_softint(9F)
ddi_intr_hilevel(9F)	3 ステップの処理: 1. ddi_intr_alloc(9F) 2. ddi_intr_get_hilevel_pri(9F) 3. ddi_intr_free(9F)
ddi_remove_intr(9F)	3 ステップの処理: 1. ddi_intr_disable(9F) 2. ddi_intr_remove_handler(9F) 3. ddi_intr_free(9F)
ddi_remove_softintr(9F)	ddi_intr_remove_softint(9F)
ddi_trigger_softintr(9F)	ddi_intr_trigger_softint(9F)

プログラム式入出力関数

プログラム式入出力関数には次のものがあります。

ddi_dev_nregs()	デバイスが備えているレジスタセットの数を返す
ddi_dev_regsize()	デバイスのレジスタのサイズを返す
ddi_regs_map_setup()	レジスタアドレス空間のマッピングを設定する
ddi_regs_map_free()	以前にマップされたレジスタアドレス空間を解放する
ddi_device_copy()	あるデバイスレジスタのデータを別のデバイスレジスタにコピーする
ddi_device_zero()	デバイスをゼロに初期化する
ddi_check_acc_handle()	データアクセスハンドルをチェックする

`ddi_get8()`	マップされたメモリー、デバイスレジスタ、または DMA メモリーから 8 ビットデータを読み取る
`ddi_get16()`	マップされたメモリー、デバイスレジスタ、または DMA メモリーから 16 ビットデータを読み取る
`ddi_get32()`	マップされたメモリー、デバイスレジスタ、または DMA メモリーから 32 ビットデータを読み取る
`ddi_get64()`	マップされたメモリー、デバイスレジスタ、または DMA メモリーから 64 ビットデータを読み取る
`ddi_put8()`	マップされたメモリー、デバイスレジスタ、または DMA メモリーに 8 ビットデータを書き込む
`ddi_put16()`	マップされたメモリー、デバイスレジスタ、または DMA メモリーに 16 ビットデータを書き込む
`ddi_put32()`	マップされたメモリー、デバイスレジスタ、または DMA メモリーに 32 ビットデータを書き込む
`ddi_put64()`	マップされたメモリー、デバイスレジスタ、または DMA メモリーに 64 ビットデータを書き込む
`ddi_rep_get8()`	マップされたメモリー、デバイスレジスタ、または DMA メモリーから複数の 8 ビットデータを読み取る
`ddi_rep_get16()`	マップされたメモリー、デバイスレジスタ、または DMA メモリーから複数の 16 ビットデータを読み取る
`ddi_rep_get32()`	マップされたメモリー、デバイスレジスタ、または DMA メモリーから複数の 32 ビットデータを読み取る
`ddi_rep_get64()`	マップされたメモリー、デバイスレジスタ、または DMA メモリーから複数の 64 ビットデータを読み取る
`ddi_rep_put8()`	マップされたメモリー、デバイスレジスタ、または DMA メモリーに複数の 8 ビットデータを書き込む
`ddi_rep_put16()`	マップされたメモリー、デバイスレジスタ、または DMA メモリーに複数の 16 ビットデータを書き込む
`ddi_rep_put32()`	マップされたメモリー、デバイスレジスタ、または DMA メモリーに複数の 32 ビットデータを書き込む
`ddi_rep_put64()`	マップされたメモリー、デバイスレジスタ、または DMA メモリーに複数の 64 ビットデータを書き込む
`ddi_peek8()`	ある場所から 8 ビット値を慎重に読み取る
`ddi_peek16()`	ある場所から 16 ビット値を慎重に読み取る

`ddi_peek32()`	ある場所から 32 ビット値を慎重に読み取る
`ddi_peek64()`	ある場所から 64 ビット値を慎重に読み取る
`ddi_poke8()`	ある場所に 8 ビット値を慎重に書き込む
`ddi_poke16()`	ある場所に 16 ビット値を慎重に書き込む
`ddi_poke32()`	ある場所に 32 ビット値を慎重に書き込む
`ddi_poke64()`	ある場所に 64 ビット値を慎重に書き込む

上に示した一般的なプログラム式入出力関数は常に、次に示す `mem`、`io`、および `pci_config` 関数の代わりに使用できます。ただし、コンパイル時にアクセスのタイプがわかっている場合は、次の関数を代わりに使用できます。

`ddi_io_get8()`	入出力空間内のマップされたデバイスレジスタから 8 ビットデータを読み取る
`ddi_io_get16()`	入出力空間内のマップされたデバイスレジスタから 16 ビットデータを読み取る
`ddi_io_get32()`	入出力空間内のマップされたデバイスレジスタから 32 ビットデータを読み取る
`ddi_io_put8()`	入出力空間内のマップされたデバイスレジスタに 8 ビットデータを書き込む
`ddi_io_put16()`	入出力空間内のマップされたデバイスレジスタに 16 ビットデータを書き込む
`ddi_io_put32()`	入出力空間内のマップされたデバイスレジスタに 32 ビットデータを書き込む
`ddi_io_rep_get8()`	入出力空間内のマップされたデバイスレジスタから複数の 8 ビットデータを読み取る
`ddi_io_rep_get16()`	入出力空間内のマップされたデバイスレジスタから複数の 16 ビットデータを読み取る
`ddi_io_rep_get32()`	入出力空間内のマップされたデバイスレジスタから複数の 32 ビットデータを読み取る
`ddi_io_rep_put8()`	入出力空間内のマップされたデバイスレジスタに複数の 8 ビットデータを書き込む
`ddi_io_rep_put16()`	入出力空間内のマップされたデバイスレジスタに複数の 16 ビットデータを書き込む
`ddi_io_rep_put32()`	入出力空間内のマップされたデバイスレジスタに複数の 32 ビットデータを書き込む

`ddi_mem_get8()`	メモリー空間または DMA メモリー内のマップされたデバイスから 8 ビットデータを読み取る
`ddi_mem_get16()`	メモリー空間または DMA メモリー内のマップされたデバイスから 16 ビットデータを読み取る
`ddi_mem_get32()`	メモリー空間または DMA メモリー内のマップされたデバイスから 32 ビットデータを読み取る
`ddi_mem_get64()`	メモリー空間または DMA メモリー内のマップされたデバイスから 64 ビットデータを読み取る
`ddi_mem_put8()`	メモリー空間または DMA メモリー内のマップされたデバイスに 8 ビットデータを書き込む
`ddi_mem_put16()`	メモリー空間または DMA メモリー内のマップされたデバイスに 16 ビットデータを書き込む
`ddi_mem_put32()`	メモリー空間または DMA メモリー内のマップされたデバイスに 32 ビットデータを書き込む
`ddi_mem_put64()`	メモリー空間または DMA メモリー内のマップされたデバイスに 64 ビットデータを書き込む
`ddi_mem_rep_get8()`	メモリー空間または DMA メモリー内のマップされたデバイスから複数の 8 ビットデータを読み取る
`ddi_mem_rep_get16()`	メモリー空間または DMA メモリー内のマップされたデバイスから複数の 16 ビットデータを読み取る
`ddi_mem_rep_get32()`	メモリー空間または DMA メモリー内のマップされたデバイスから複数の 32 ビットデータを読み取る
`ddi_mem_rep_get64()`	メモリー空間または DMA メモリー内のマップされたデバイスから複数の 64 ビットデータを読み取る
`ddi_mem_rep_put8()`	メモリー空間または DMA メモリー内のマップされたデバイスに複数の 8 ビットデータを書き込む
`ddi_mem_rep_put16()`	メモリー空間または DMA メモリー内のマップされたデバイスに複数の 16 ビットデータを書き込む
`ddi_mem_rep_put32()`	メモリー空間または DMA メモリー内のマップされたデバイスに複数の 32 ビットデータを書き込む
`ddi_mem_rep_put64()`	メモリー空間または DMA メモリー内のマップされたデバイスに複数の 64 ビットデータを書き込む
`pci_config_setup()`	PCI ローカルバス構成スペースへのアクセスを設定する
`pci_config_teardown()`	PCI ローカルバス構成スペースへのアクセスを破棄する

pci_config_get8()	PCI ローカルバス構成スペースから 8 ビットデータを読み取る
pci_config_get16()	PCI ローカルバス構成スペースから 16 ビットデータを読み取る
pci_config_get32()	PCI ローカルバス構成スペースから 32 ビットデータを読み取る
pci_config_get64()	PCI ローカルバス構成スペースから 64 ビットデータを読み取る
pci_config_put8()	PCI ローカルバス構成スペースに 8 ビットデータを書き込む
pci_config_put16()	PCI ローカルバス構成スペースに 16 ビットデータを書き込む
pci_config_put32()	PCI ローカルバス構成スペースに 32 ビットデータを書き込む
pci_config_put64()	PCI ローカルバス構成スペースに 64 ビットデータを書き込む

表 **B–4** 非推奨のプログラム式入出力関数

非推奨の関数	代わりの **FEATURE**
ddi_getb()	ddi_get8()
ddi_getl()	ddi_get32()
ddi_getll()	ddi_get64()
ddi_getw()	ddi_get16()
ddi_io_getb()	ddi_io_get8()
ddi_io_getl()	ddi_io_get32()
ddi_io_getw()	ddi_io_get16()
ddi_io_putb()	ddi_io_put8()
ddi_io_putl()	ddi_io_put32()
ddi_io_putw()	ddi_io_put16()
ddi_io_rep_getb()	ddi_io_rep_get8()
ddi_io_rep_getl()	ddi_io_rep_get32()
ddi_io_rep_getw()	ddi_io_rep_get16()

表 B–4 非推奨のプログラム式入出力関数 (続き)

非推奨の関数	代わりの FEATURE
ddi_io_rep_putb()	ddi_io_rep_put8()
ddi_io_rep_putl()	ddi_io_rep_put32()
ddi_io_rep_putw()	ddi_io_rep_put16()
ddi_map_regs()	ddi_regs_map_setup()
ddi_mem_getb()	ddi_mem_get8()
ddi_mem_getl()	ddi_mem_get32()
ddi_mem_getll()	ddi_mem_get64()
ddi_mem_getw()	ddi_mem_get16()
ddi_mem_putb()	ddi_mem_put8()
ddi_mem_putl()	ddi_mem_put32()
ddi_mem_putll()	ddi_mem_put64()
ddi_mem_putw()	ddi_mem_put16()
ddi_mem_rep_getb()	ddi_mem_rep_get8()
ddi_mem_rep_getl()	ddi_mem_rep_get32()
ddi_mem_rep_getll()	ddi_mem_rep_get64()
ddi_mem_rep_getw()	ddi_mem_rep_get16()
ddi_mem_rep_putb()	ddi_mem_rep_put8()
ddi_mem_rep_putl()	ddi_mem_rep_put32()
ddi_mem_rep_putll()	ddi_mem_rep_put64()
ddi_mem_rep_putw()	ddi_mem_rep_put16()
ddi_peekc()	ddi_peek8()
ddi_peekd()	ddi_peek64()
ddi_peekl()	ddi_peek32()
ddi_peeks()	ddi_peek16()
ddi_pokec()	ddi_poke8()
ddi_poked()	ddi_poke64()
ddi_pokel()	ddi_poke32()
ddi_pokes()	ddi_poke16()

表 B–4 非推奨のプログラム式入出力関数 (続き)

非推奨の関数	代わりの FEATURE
ddi_putb()	ddi_put8()
ddi_putl()	ddi_put32()
ddi_putll()	ddi_put64()
ddi_putw()	ddi_put16()
ddi_rep_getb()	ddi_rep_get8()
ddi_rep_getl()	ddi_rep_get32()
ddi_rep_getll()	ddi_rep_get64()
ddi_rep_getw()	ddi_rep_get16()
ddi_rep_putb()	ddi_rep_put8()
ddi_rep_putl()	ddi_rep_put32()
ddi_rep_putll()	ddi_rep_put64()
ddi_rep_putw()	ddi_rep_put16()
ddi_unmap_regs()	ddi_regs_map_free()
inb()	ddi_io_get8()
inl()	ddi_io_get32()
inw()	ddi_io_get16()
outb()	ddi_io_put8()
outl()	ddi_io_put32()
outw()	ddi_io_put16()
pci_config_getb()	pci_config_get8()
pci_config_getl()	pci_config_get32()
pci_config_getll()	pci_config_get64()
pci_config_getw()	pci_config_get16()
pci_config_putb()	pci_config_put8()
pci_config_putl()	pci_config_put32()
pci_config_putll()	pci_config_put64()
pci_config_putw()	pci_config_put16()
repinsb()	ddi_io_rep_get8()

表 B–4 非推奨のプログラム式入出力関数 (続き)

非推奨の関数	代わりの FEATURE
repinsd()	ddi_io_rep_get32()
repinsw()	ddi_io_rep_get16()
repoutsb()	ddi_io_rep_put8()
repoutsd()	ddi_io_rep_put32()
repoutsw()	ddi_io_rep_put16()

ダイレクトメモリーアクセス (DMA) 関数

DMA 関数には次のものがあります。

ddi_dma_alloc_handle()	DMA ハンドルを割り当てる
ddi_dma_free_handle()	DMA ハンドルを解放する
ddi_dma_mem_alloc()	DMA 転送のためのメモリーを割り当てる
ddi_dma_mem_free()	以前に割り当てられた DMA メモリーを解放する
ddi_dma_addr_bind_handle()	アドレスを DMA ハンドルにバインドする
ddi_dma_buf_bind_handle()	システムバッファーを DMA ハンドルにバインドする
ddi_dma_unbind_handle()	DMA ハンドル内のアドレスをバインド解除する
ddi_dma_nextcookie()	以降の DMA cookie を取得する
ddi_dma_getwin()	新しい DMA ウィンドウをアクティブにする
ddi_dma_numwin()	DMA ウィンドウの数を取得する
ddi_dma_sync()	メモリーの CPU と入出力のビューの同期をとる
ddi_check_dma_handle()	DMA ハンドルをチェックする
ddi_dma_set_sbus64()	SBus 上の 64 ビット転送を許可する
ddi_slaveonly()	デバイスが、スレーブのみがアクセスできる場所にインストールされているかどうかを報告する
ddi_iomin()	DMA の最小の整列および転送サイズを検索する
ddi_dma_burstsizes()	DMA マッピングに対して許可されているバーストサイズを検索する
ddi_dma_devalign()	DMA のマッピング整列および最小の転送サイズを検索する

ddi_dmae_alloc()	DMA チャネルを取得する
ddi_dmae_release()	DMA チャネルを解放する
ddi_dmae_getattr()	DMA エンジンの属性を取得する
ddi_dmae_prog()	DMA チャネルをプログラムする
ddi_dmae_stop()	DMA エンジンの操作を終了する
ddi_dmae_disable()	DMA チャネルを無効にする
ddi_dmae_enable()	DMA チャネルを有効にする
ddi_dmae_getcnt()	残りの DMA エンジン数を取得する
ddi_dmae_1stparty()	DMA チャネルのカスケードモードを構成する
ddi_dma_coff()	DMA cookie を DMA ハンドル内のオフセットに変換する

表 B–5 非推奨のダイレクトメモリーアクセス (DMA) 関数

非推奨の関数	代わりの FEATURE
ddi_dma_addr_setup()	ddi_dma_alloc_handle()、ddi_dma_addr_bind_handle()
ddi_dma_buf_setup()	ddi_dma_alloc_handle()、ddi_dma_buf_bind_handle()
ddi_dma_curwin()	ddi_dma_getwin()
ddi_dma_free()	ddi_dma_free_handle()
ddi_dma_htoc()	ddi_dma_addr_bind_handle()、ddi_dma_buf_bind_handle()
ddi_dma_movwin()	ddi_dma_getwin()
ddi_dma_nextseg()	ddi_dma_nextcookie()
ddi_dma_segtocookie()	ddi_dma_nextcookie()
ddi_dma_setup()	ddi_dma_alloc_handle()、ddi_dma_addr_bind_handle()、ddi_dma_buf_bind_handle()
ddi_dmae_getlim()	ddi_dmae_getattr()
ddi_iopb_alloc()	ddi_dma_mem_alloc()
ddi_iopb_free()	ddi_dma_mem_free()
ddi_mem_alloc()	ddi_dma_mem_alloc()
ddi_mem_free()	ddi_dma_mem_free()
hat_getkpfnum()	ddi_dma_addr_bind_handle()、ddi_dma_buf_bind_handle()、ddi_dma_nextcookie()

ユーザー空間アクセス関数

ユーザー空間アクセス関数には次のものがあります。

ddi_copyin()	データをドライババッファーにコピーする
ddi_copyout()	ドライバからデータをコピーする
uiomove()	uio 構造体を使用してカーネルデータをコピーする
ureadc()	文字を uio 構造体に追加する
uwritec()	uio 構造体から文字を削除する
getminor()	マイナーデバイス番号を取得する
ddi_model_convert_from()	データモデルタイプの不一致を判定する
IOC_CONVERT_FROM()	M_IOCTL の内容を変換する必要があるかどうかを判定する
STRUCT_DECL()	異なる可能性のあるデータモデル内のアプリケーションデータへのハンドルを確立する
STRUCT_HANDLE()	異なる可能性のあるデータモデル内のアプリケーションデータへのハンドルを確立する
STRUCT_INIT()	異なる可能性のあるデータモデル内のアプリケーションデータへのハンドルを確立する
STRUCT_SET_HANDLE()	異なる可能性のあるデータモデル内のアプリケーションデータへのハンドルを確立する
SIZEOF_PTR()	指定されたデータモデル内のポインタのサイズを返す
SIZEOF_STRUCT()	指定されたデータモデル内の構造体のサイズを返す
STRUCT_SIZE()	アプリケーションデータモデル内の構造体のサイズを返す
STRUCT_BUF()	構造体のネイティブモードインスタンスへのポインタを返す
STRUCT_FADDR()	構造体の指定されたフィールドへのポインタを返す
STRUCT_FGET()	アプリケーションデータモデル内の構造体の指定されたフィールドを返す
STRUCT_FGETP()	アプリケーションデータモデル内の構造体の指定されたポインタフィールドを返す
STRUCT_FSET()	アプリケーションデータモデル内の構造体の指定されたフィールドを設定する

`STRUCT_FSETP()` アプリケーションデータモデル内の構造体の指定されたポインタフィールドを設定する

表 B–6 非推奨のユーザー空間アクセス関数

非推奨の関数	代わりの FEATURE
`copyin()`	`ddi_copyin()`
`copyout()`	`ddi_copyout()`
`ddi_getminor()`	`getminor()`

ユーザープロセスイベント関数

ユーザープロセスイベント関数には次のものがあります。

`pollwakeup()` イベントが発生したことをプロセスに通知する

`proc_ref()` シグナルへのプロセス上のハンドルを取得する

`proc_unref()` シグナルへのプロセス上のハンドルを解放する

`proc_signal()` シグナルをプロセスに送信する

ユーザープロセス情報関数

ユーザープロセス情報関数には次のものがあります。

`ddi_get_cred()` 呼び出し元の資格構造体へのポインタを返す

`drv_priv()` プロセスの資格特権を判定する

`ddi_get_pid()` プロセス ID を返す

表 B–7 非推奨のユーザープロセス情報関数

非推奨の関数	代わりの FEATURE
`drv_getparm()`	`ddi_get_pid()`、`ddi_get_cred()`

ユーザーアプリケーションカーネルおよびデバイスアクセス関数

ユーザーアプリケーションカーネルおよびデバイスアクセス関数には次のものがあります。

`ddi_dev_nregs()`	デバイスが備えているレジスタセットの数を返す
`ddi_dev_regsize()`	デバイスのレジスタのサイズを返す
`ddi_devmap_segmap()`、`devmap_setup()`	`devmap` フレームワークを使用して、デバイスメモリーへのユーザーマッピングを設定する
`devmap_devmem_setup()`	デバイスメモリーをユーザー空間にエクスポートする
`devmap_load()`	メモリーアドレス変換を検証する
`devmap_unload()`	メモリーアドレス変換を無効にする
`devmap_do_ctxmgt()`	マッピング上のデバイスコンテキストの切り替えを実行する
`devmap_set_ctx_timeout()`	コンテキスト管理コールバックのタイムアウト値を設定する
`devmap_default_access()`	デフォルトのドライバメモリーアクセス関数
`ddi_umem_alloc()`	ページ境界割り当てされたカーネルメモリーを割り当てる
`ddi_umem_free()`	ページ境界割り当てされたカーネルメモリーを解放する
`ddi_umem_lock()`	メモリーページをロックする
`ddi_umem_unlock()`	メモリーページをロック解除する
`ddi_umem_iosetup()`	アプリケーションメモリーへの入出力要求を設定する
`devmap_umem_setup()`	カーネルメモリーをユーザー空間にエクスポートする
`ddi_model_convert_from()`	データモデルタイプの不一致を判定する

表 B–8 非推奨のユーザーアプリケーションカーネルおよびデバイスアクセス関数

非推奨の関数	代わりの FEATURE
ddi_mapdev()	devmap_setup()
ddi_mapdev_intercept()	devmap_load()
ddi_mapdev_nointercept()	devmap_unload()
ddi_mapdev_set_device_acc_attr()	devmap()
ddi_segmap()	devmap()
ddi_segmap_setup()	devmap_setup()
hat_getkpfnum()	devmap()
ddi_mmap_get_model()	devmap()

時間関連関数

時間関連関数には次のものがあります。

ddi_get_lbolt()	リブート以降のクロック刻みの数を返す
ddi_get_time()	現在の時間 (秒単位) を返す
ddi_periodic_add()	ナノ秒周期のタイムアウト要求を発行する
ddi_periodic_delete()	ナノ秒周期のタイムアウト要求を取り消す
delay()	指定されたクロック刻みの数だけ実行を遅延させる
drv_hztousec()	クロック刻みをマイクロ秒に変換する
drv_usectohz()	マイクロ秒をクロック刻みに変換する
drv_usecwait()	指定された期間、ビジー状態で待機する
gethrtime()	高精度の時間を取得する
gethrvtime()	高精度の LWP 仮想時間を取得する
timeout()	指定された時間の長さのあとに関数を実行する
untimeout()	以前のタイムアウト関数呼び出しを取り消す
drv_getparm()	ddi_get_lbolt()、ddi_get_time()

表 B–9 非推奨の時間関連関数

非推奨の関数	代わりの FEATURE
drv_getparm()	ddi_get_lbolt()、ddi_get_time()

電源管理関数

電源管理関数には次のものがあります。

ddi_removing_power()	DDI_SUSPEND でデバイスの電源が切れているかどうかをチェックする
pci_report_pmcap()	PCI デバイスの電源管理機能を報告する
pm_busy_component()	コンポーネントをビジー状態としてマークする
pm_idle_component()	コンポーネントをアイドル状態としてマークする
pm_raise_power()	コンポーネントの電源レベルを上げる
pm_lower_power()	コンポーネントの電源レベルを下げる
pm_power_has_changed()	電源管理フレームワークに自律的な電源レベルの変更を通知する
pm_trans_check()	デバイスの電源再投入のアドバイザリチェック

表 B–10 非推奨の電源管理関数

関数名	説明
ddi_dev_is_needed()	システムにデバイスのコンポーネントが必要なことを通知する
pm_create_components()	電源管理可能なコンポーネントを作成する
pm_destroy_components()	電源管理可能なコンポーネントを破棄する
pm_get_normal_power()	デバイスコンポーネントの通常の電源レベルを取得する
pm_set_normal_power()	デバイスコンポーネントの通常の電源レベルを設定する

障害管理関数

障害管理関数には次のものがあります。

`ddi_fm_init()`	宣言された障害管理機能に基づいてリソースを割り当て、初期化する
`ddi_fm_fini()`	`ddi_fm_init()` に対して宣言された障害管理機能をサポートするために、このデバイスインスタンスに割り当てられたリソースをクリーンアップする
`ddi_fm_capable()`	現在このデバイスインスタンスに対して設定されている機能ビットマスクを返す
`ddi_fm_handler_register()`	IO 障害管理フレームワークにエラーハンドラコールバックルーチンを登録する
`ddi_fm_handler_unregister()`	`ddi_fm_handler_register()` で登録されたエラーハンドラコールバックルーチンを削除する
`ddi_fm_acc_err_get()`	アクセスハンドルのエラーステータスを返す
`ddi_fm_dma_err_get()`	DMA ハンドルのエラーステータスを返す
`ddi_fm_acc_err_clear()`	アクセスハンドルのエラーステータスをクリアする
`ddi_fm_dma_err_clear()`	DMA ハンドルのエラーステータスをクリアする
`ddi_fm_ereport_post()`	Fault Manager デーモン `fmd`(1M) に提供するためのエンコードされた障害管理エラー報告の名前と値のペアのリストをキューに入れる
`ddi_fm_service_impact()`	エラーの影響を報告する
`pci_ereport_setup()`	エラー報告作成のサポートを初期化し、PCI、PCI/X、または PCI Express 構成スペースへの以降のアクセスのためのリソースを設定する
`pci_ereport_teardown()`	このデバイスインスタンスに対して `pci_ereport_setup()` によって割り当ておよび設定されたリソースをすべて解放する
`pci_ereport_post()`	PCI、PCI/X、または PCI Express のバスエラーをすべてスキャンして送信する

カーネル統計関数

カーネル統計 (kstats) 関数には次のものがあります。

`kstat_create()`	新しい kstat を作成して初期化する
`kstat_delete()`	システムから kstat を削除する
`kstat_install()`	完全に初期化された kstat をシステムに追加する
`kstat_named_init()`	名前付きの kstat を初期化する
`kstat_runq_back_to_waitq()`	実行キューから待機キューへのトランザクション移行を記録する
`kstat_runq_enter()`	実行キューへのトランザクション追加を記録する
`kstat_runq_exit()`	実行キューからのトランザクション削除を記録する
`kstat_waitq_enter()`	待機キューへのトランザクション追加を記録する
`kstat_waitq_exit()`	待機キューからのトランザクション削除を記録する
`kstat_waitq_to_runq()`	待機キューから実行キューへのトランザクション移行を記録する

カーネルロギングおよび印刷関数

カーネルロギングおよび印刷関数には次のものがあります。

`cmn_err()、vcmn_err()`	エラーメッセージを表示する
`ddi_report_dev()`	デバイスを通知する
`strlog()`	ログドライバにメッセージを送信する
`ddi_dev_report_fault()`	ハードウェア障害を報告する
`scsi_errmsg()`	SCSI 要求検知メッセージを表示する
`scsi_log()`	SCSI デバイス関連のメッセージを表示する
`scsi_vu_errmsg()`	SCSI 要求検知メッセージを表示する

バッファリングされた入出力関数

バッファリングされた入出力関数には次のものがあります。

`physio()`	物理入出力を実行する
`aphysio()`	非同期物理入出力を実行する
`anocancel()`	非同期入出力要求の取り消しを回避する
`minphys()`	`physio()` のバッファーサイズを制限する
`biowait()`	ブロック入出力の完了を保留しているプロセスを中断する
`biodone()`	バッファー入出力転送のあとにバッファーを解放し、ブロックされたスレッドに通知する
`bioerror()`	バッファーのヘッダー内のエラーを示す
`geterror()`	入出力エラーを返す
`bp_mapin()`	仮想アドレス空間を割り当てる
`bp_mapout()`	仮想アドレス空間を解放する
`disksort()`	バッファーをソートするために単方向エレベータシークの方針を使用する
`getrbuf()`	raw バッファーのヘッダーを取得する
`freerbuf()`	raw バッファーのヘッダーを解放する
`biosize()`	バッファー構造体のサイズを返す
`bioinit()`	バッファー構造体を初期化する
`biofini()`	バッファー構造体の初期化を解除する
`bioreset()`	入出力が完了したあとに非公開のバッファーのヘッダーを再利用する
`bioclone()`	別のバッファーを複製する
`biomodified()`	バッファーが変更されているかどうかをチェックする
`clrbuf()`	バッファーの内容を消去する

仮想メモリー関数

仮想メモリー関数には次のものがあります。

`ddi_btop()`	デバイスのバイト数をページ数に変換する (切り捨て)
`ddi_btopr()`	デバイスのバイト数をページ数に変換する (切り上げ)
`ddi_ptob()`	デバイスのページ数をバイト数に変換する
`btop()`	バイト単位のサイズをページ単位のサイズに変換する (切り捨て)
`btopr()`	バイト単位のサイズをページ単位のサイズに変換する (切り上げ)
`ptob()`	ページ単位のサイズをバイト単位のサイズに変換する

表 B–11 非推奨の仮想メモリー関数

非推奨の関数	代わりの **FEATURE**
`hat_getkpfnum()`	`devmap()`、`ddi_dma_*_bind_handle()`、`ddi_dma_nextcookie()`

デバイス ID 関数

デバイス ID 関数には次のものがあります。

`ddi_devid_init()`	デバイス ID 構造体を割り当てる
`ddi_devid_free()`	デバイス ID 構造体を解放する
`ddi_devid_register()`	デバイス ID を登録する
`ddi_devid_unregister()`	デバイス ID の登録を解除する
`ddi_devid_compare()`	2 つのデバイス ID を比較する
`ddi_devid_sizeof()`	デバイス ID のサイズを返す
`ddi_devid_valid()`	デバイス ID を検証する
`ddi_devid_str_encode()`	デバイス ID と minor_name を NULL で終わる ASCII 文字列にエンコードし、その文字列へのポインタを返す
`ddi_devid_str_decode()`	以前にエンコードされた文字列からデバイス ID と minor_name をデコードし、抽出された部分へのポインタを割り当てて返す
`ddi_devid_str_free()`	`ddi_devid_*` 関数によって返されたすべての文字列を解放する

SCSI 関数

SCSI 関数には次のものがあります。

`scsi_probe()`	SCSI デバイスをプローブする
`scsi_unprobe()`	最初のプローブ中に割り当てられたリソースを解放する
`scsi_alloc_consistent_buf()`	SCSI DMA のための入出力バッファーを割り当てる
`scsi_free_consistent_buf()`	以前に割り当てられた SCSI DMA 入出力バッファーを解放する
`scsi_init_pkt()`	完全な SCSI パケットを準備する
`scsi_destroy_pkt()`	割り当てられた SCSI パケットとその DMA リソースを解放する
`scsi_setup_cdb()`	SCSI コマンド記述子ブロック (CDB) を設定する
`scsi_transport()`	SCSI コマンドを開始する
`scsi_poll()`	ポーリングされた SCSI コマンドを実行する
`scsi_ifgetcap()`	SCSI トランスポート機能を取得する
`scsi_ifsetcap()`	SCSI トランスポート機能を設定する
`scsi_sync_pkt()`	メモリーの CPU と入出力のビューの同期をとる
`scsi_abort()`	SCSI コマンドを中止する
`scsi_reset()`	SCSI バスまたはターゲットをリセットする
`scsi_reset_notify()`	ターゲットドライバにバスのリセットを通知する
`scsi_cname()`	SCSI コマンドをデコードする
`scsi_dname()`	SCSI 周辺デバイスのタイプをデコードする
`scsi_mname()`	SCSI メッセージをデコードする
`scsi_rname()`	SCSI パケットの完了の理由をデコードする
`scsi_sname()`	SCSI 検知キーをデコードする
`scsi_errmsg()`	SCSI 要求検知メッセージを表示する
`scsi_log()`	SCSI デバイス関連のメッセージを表示する
`scsi_vu_errmsg()`	SCSI 要求検知メッセージを表示する
`scsi_hba_init()`	SCSI HBA システムの初期化ルーチン
`scsi_hba_fini()`	SCSI HBA システムの完了ルーチン

scsi_hba_attach_setup()	SCSI HBA の接続ルーチン
scsi_hba_detach()	SCSI HBA の切り離しルーチン
scsi_hba_probe()	デフォルトの SCSI HBA プローブ関数
scsi_hba_tran_alloc()	トランスポート構造体を割り当てる
scsi_hba_tran_free()	トランスポート構造体を解放する
scsi_hba_pkt_alloc()	scsi_pkt 構造体を割り当てる
scsi_hba_pkt_free()	scsi_pkt 構造体を解放する
scsi_hba_lookup_capstr()	インデックスマッチング機能文字列を返す

表 **B–12** 非推奨の SCSI 関数

非推奨の関数	代わりの **FEATURE**
free_pktiopb()	scsi_free_consistent_buf()
get_pktiopb()	scsi_alloc_consistent_buf()
makecom_g0()	scsi_setup_cdb()
makecom_g0_s()	scsi_setup_cdb()
makecom_g1()	scsi_setup_cdb()
makecom_g5()	scsi_setup_cdb()
scsi_dmafree()	scsi_destroy_pkt()
scsi_dmaget()	scsi_init_pkt()
scsi_hba_attach()	scsi_hba_attach_setup()
scsi_pktalloc()	scsi_init_pkt()
scsi_pktfree()	scsi_destroy_pkt()
scsi_resalloc()	scsi_init_pkt()
scsi_resfree()	scsi_destroy_pkt()
scsi_slave()	scsi_probe()
scsi_unslave()	scsi_unprobe()

リソースマップ管理関数

リソースマップ管理関数には次のものがあります。

`rmallocmap()`	リソースマップを割り当てる
`rmallocmap_wait()`	リソースマップを割り当て、必要に応じて待機する
`rmfreemap()`	リソースマップを解放する
`rmalloc()`	リソースマップから領域を割り当てる
`rmalloc_wait()`	リソースマップから領域を割り当て、必要に応じて待機する
`rmfree()`	領域を元のリソースマップに解放する

システムのグローバル状態

`ddi_in_panic()`	システムがパニック状態にあるかどうかを判定する

ユーティリティー関数

ユーティリティー関数には次のものがあります。

`nulldev()`	0 を返す関数
`nodev()`	エラー戻り関数
`nochpoll()`	ポーリングできないデバイスのためのエラー戻り関数
`ASSERT()`	式の検証
`bcopy()`	カーネル内のアドレスの場所間でデータをコピーする
`bzero()`	指定されたバイト数のメモリーをクリアする
`bcmp()`	2 つのバイト配列を比較する
`ddi_ffs()`	ロング整数内に設定された最初のビットを検索する
`ddi_fls()`	ロング整数内に設定された最後のビットを検索する
`swab()`	バイトを 16 ビットハーフワード単位でスワップする
`strcmp()`	NULL で終わる 2 つの文字列を比較する
`strncmp()`	NULL で終わる 2 つの文字列を比較する (長さ制限あり)
`strlen()`	文字列内の NULL 以外のバイト数を判定する
`strcpy()`	文字列をある場所から別の場所にコピーする

`strncpy()`	文字列をある場所から別の場所にコピーする (長さ制限あり)
`strchr()`	文字列内の文字を検索する
`sprintf()、vsprintf()`	メモリー内の文字を書式設定する
`numtos()`	整数を 10 進数文字列に変換する
`stoi()`	10 進数文字列を整数に変換する
`max()`	2 つの整数のうちの大きい方を返す
`min()`	2 つの整数のうちの小さい方を返す
`va_arg()`	変数引数リスト内の次の値を検索する
`va_copy()`	変数引数リストの状態をコピーする
`va_end()`	変数引数リストへのポインタを削除する
`va_start()`	変数引数リストの先頭へのポインタを検索する

◆ ◆ ◆ 付録 C

64 ビットデバイスドライバの準備

この付録では、64 ビットカーネルをサポートするようにデバイスドライバを変換するデバイスドライバ作成者向けの情報を提供します。32 ビットと 64 ビットのデバイスドライバの違いについて説明し、32 ビットのデバイスドライバを 64 ビットに変換する手順についても説明します。この情報は、通常のキャラクタデバイスおよびブロックデバイスのドライバ専用です。

この付録では、次の内容について説明します。

64 ビットドライバの設計の紹介

32 ビットカーネルだけをサポートすればよいドライバの場合、既存の 32 ビットのデバイスドライバは再コンパイルなしで引き続き動作します。ただし、ほとんどのデバイスドライバでは、64 ビットカーネルで正しく動作するためにはいくらかの変更が必要です。また、すべてのデバイスドライバで、再コンパイルして 64 ビットのドライバモジュールを作成する必要があります。この付録の情報は、32 ビット環境用と 64 ビット環境用のドライバを共通のソースコードから生成して、コードの移植性を高め、保守の手間を減らすのに役立ちます。

64 ビット環境用にデバイスドライバの変更を開始する前に、32 ビット環境と 64 ビット環境の違いを理解するようにしてください。特に、C 言語のデータ型モデルの ILP32 と LP64 に精通する必要があります。次の表を参照してください。

表 C–1 ILP32 と LP64 のデータ型の比較

C データ型	ILP32	LP64
char	8	8

表 C–1 ILP32 と LP64 のデータ型の比較 *(続き)*

C データ型	ILP32	LP64
short	16	16
int	32	32
long	32	64
long long	64	64
float	32	32
double	64	64
long double	96	128
pointer	32	64

この付録のテーマは、ILP32 と LP64 の違いによるドライバ固有の問題です。一般的な内容については、『Solaris 64 ビット 開発ガイド』を参照してください。

ドライバの作成者は、LP64 用のデータモデル変更をサポートするための一般的なコードクリーンアップに加えて、32 ビットと 64 ビットの両方のアプリケーションのサポートを提供する必要があります。

ioctl(9E)、devmap(9E)、および mmap(9E) の各エントリポイントによって、アプリケーションとデバイスドライバの間で直接データ構造体を共有できます。それらのデータ構造体のサイズが 32 ビット環境と 64 ビット環境の間で変化する場合は、アプリケーションのデータモデルがカーネルのデータモデルと同じかどうかをドライバが判定できるように、エントリポイントを変更する必要があります。データモデルが異なる場合は、データ構造体を調整できます。326 ページの「64 ビットに対応したデバイスドライバに対する入出力制御のサポート」、329 ページの「32 ビットと 64 ビットのデータ構造体マクロ」、および193 ページの「ユーザーマッピングへのカーネルメモリーの関連付け」を参照してください。

多くのドライバで、この種の処理が必要な ioctls は少数です。その他の ioctls は、ioctls がサイズの変化しないデータ構造体を渡すかぎり、変更なしで動作します。

一般的な変換手順

以降のセクションでは、64 ビット環境で動作するようにドライバを変換する方法について説明します。ドライバの作成者は、次のタスクの 1 つ以上を実行する必要があることがあります。

1. 固定幅型をハードウェアレジスタに使用します。
2. 固定幅の共通アクセス関数を使用します。

3. 派生型の使用を確認および拡張します。
4. DDI データ構造体内の変更されたフィールドを確認します。
5. DDI 関数の変更された引数を確認します。
6. 必要に応じて、ユーザーデータを処理するドライバエントリポイントを変更します。
7. x86 プラットフォームで 64 ビットの long 型を使用する構造体を確認します。

これらの手順について、以降で詳しく説明します。

各手順の完了後に、コンパイラの警告をすべて修正し、`lint` を使用してほかの問題を検索します。64 ビットの問題を検索するときは、`-Xarch=v9` および `-errchk=longptr64` を指定して、SC5.0 (またはそれ以降の) バージョンの `lint` を使用してください。`lint` の出力の使用と解釈については、『Solaris 64 ビット 開発ガイド』の注を参照してください。

注 – LP64 用の変換時のコンパイル警告を無視しないでください。以前は ILP32 環境で無視しても安全だった警告が、重大な問題を示すようになっていることがあります。

すべての手順を完了したら、32 ビットと 64 ビットの両方のモジュールとしてドライバをコンパイルおよびテストします。

固定幅型をハードウェアレジスタに使用する

ハードウェアデバイスを操作する多くのデバイスドライバは、C データ構造体を使用してハードウェアのレイアウトを記述します。LP64 データモデルでは、`long` が 64 ビット量になったため、`long` または unsigned long を使用してハードウェアレジスタを定義するデータ構造体はほぼ確実に正しくありません。最初に `<sys/inttypes.h>` を含め、このクラスのデータ構造体を更新して、`long` の代わりに `int32_t` または `uint32_t` を 32 ビットのデバイスデータに使用します。この方法により、32 ビットデータ構造体のバイナリレイアウトが保持されます。たとえば、次の構造体を変更するとします。

```
struct device_regs {
    ulong_t        addr;
    uint_t         count;
};      /* Only works for ILP32 compilation */
```

変更後:

```
struct device_regs {
    uint32_t        addr;
    uint32_t        count;
};      /* Works for any data model */
```

固定幅の共通アクセス関数を使用する

Solaris DDI では、アクセス関数でデバイスレジスタにアクセスすることにより、複数のプラットフォームでの移植性を確保できます。以前は、DDI 共通アクセス関数はバイトやワードなどでデータのサイズを指定しました。たとえば、32 ビット量にアクセスするときは、`ddi_getl`(9F) を使用します。この関数は 64 ビットの DDI 環境では使用できず、動作するビット数を指定するバージョンの関数に置き換えられています。

これらのルーチンは、ドライバの作成者が早期に採用できるように、Solaris 2.6 オペレーティング環境の 32 ビットカーネルに追加されました。たとえば、32 ビットと 64 ビットの両方のカーネルに移植できるようにするには、ドライバは `ddi_getl`(9F) ではなく `ddi_get32`(9F) を使用して 32 ビットのデータにアクセスする必要があります。

共通アクセスルーチンはすべて、同等の固定幅のルーチンに置き換えられています。詳細は、`ddi_get8`(9F)、`ddi_put8`(9F)、 `ddi_rep_get8`(9F)、および `ddi_rep_put8`(9F) のマニュアルページを参照してください。

派生型の使用を確認および拡張する

可能な場合は `size_t` などのシステム派生型を使用して、結果として得られる変数が関数間で渡されるときに意味をなすようにしてください。新しい派生型である `uintptr_t` または `intptr_t` は、ポインタの整数型として使用してください。

固定幅の整数型は、バイナリデータ構造体またはハードウェアレジスタのサイズを明示的に表すのに便利です。一方、`int` などの C 言語の基本データ型は、ループカウンタまたはファイル記述子で引き続き使用できます。

一部のシステム派生型は、32 ビットシステムでは 32 ビット量を表しますが、64 ビットシステムでは 64 ビット量を表します。この方法でサイズを変更する派生型には、`clock_t`、`daddr_t`、`dev_t`、`ino_t`、`intptr_t`、`off_t`、`size_t`、`ssize_t`、`time_t`、`uintptr_t`、および `timeout_id_t` があります。

これらの派生型を使用するドライバを設計するときは、特にドライバがこれらの値を固定幅型などの別の派生型の変数に割り当てる場合、これらの型の使用に特別な注意を払ってください。

DDI データ構造体内の変更されたフィールドを確認する

`buf`(9S) など、DDI データ構造体内の一部のフィールドのデータ型は変更されました。これらのデータ構造体を使用するドライバでは、これらのフィールドが適切に使用されていることを確認するようにしてください。大幅に変更されたデータ構造体とフィールドの一覧を下に示します。

buf 構造体の変更内容

下に一覧表示したフィールドは転送サイズに関係しています。転送サイズは 4G バイトを超えることができるようになりました。

```
size_t        b_bcount;          /* was type unsigned int */
size_t        b_resid;           /* was type unsigned int */
size_t        b_bufsize;         /* was type long */
```

ddi_dma_attr

ddi_dma_attr(9S) 構造体は、DMA エンジンとデバイスの属性を定義します。これらの属性はレジスタサイズを指定するため、基本型の代わりに固定幅データ型が使用されるようになりました。

ddi_dma_cookie 構造体の変更内容

```
uint32_t     dmac_address;    /* was type unsigned long */
size_t       dmac_size;       /* was type u_int */
```

ddi_dma_cookie(9S) 構造体には 32 ビットの DMA アドレスが含まれているため、アドレスの定義には固定幅データ型が使用されるようになりました。サイズは、size_t として定義し直されました。

csi_arq_status 構造体の変更内容

```
uint_t    sts_rqpkt_state;         /* was type u_long */
uint_t    sts_rqpkt_statistics;    /* was type u_long */
```

構造体内のこれらのフィールドは拡張する必要がなく、32 ビット量として定義し直されました。

scsi_pkt 構造体の変更内容

```
uint_t      pkt_flags;           /* was type u_long */
int     pkt_time;        /* was type long */
ssize_t     pkt_resid;           /* was type long */
uint_t      pkt_state;           /* was type u_long */
uint_t      pkt_statistics;      /* was type u_long */
```

scsi_pkt(9S) 構造体内の pkt_flags、pkt_state、および pkt_statistics フィールドは拡張する必要がないため、これらのフィールドは 32 ビット整数として定義し直されました。データ転送サイズの pkt_resid フィールドは拡張するため、ssize_t として定義し直されました。

DDI 関数の変更された引数を確認する

このセクションでは、変更された DDI 関数の引数のデータ型について説明します。

getrbuf() 引数の変更内容

```
struct buf *getrbuf(int sleepflag);
```

前のリリースでは、sleepflag は long 型として定義されました。

drv_getparm() 引数の変更内容

```
int drv_getparm(unsigned int parm, void *value_p);
```

前のリリースでは、value_p は unsigned long 型として定義されました。64 ビットカーネルでは、drv_getparm(9F) は 32 ビット量と 64 ビット量の両方を取得できます。インタフェースではこれらの量のデータ型は定義されないので、単純なプログラミングエラーが起こる可能性があります。

次の新しいルーチンのほうが安全です。

```
clock_t       ddi_get_lbolt(void);
time_t        ddi_get_time(void);
cred_t        *ddi_get_cred(void);
pid_t         ddi_get_pid(void);
```

ドライバの作成者は、できるかぎり drv_getparm(9F) の代わりにこれらのルーチンを使用してください。

delay() および timeout() 引数の変更内容

```
void delay(clock_t ticks);
timeout_id_t timeout(void (*func)(void *), void *arg, clock_t ticks);
```

delay(9F) および timeout(9F) ルーチンの *ticks* 引数は、long から clock_t に変更されました。

rmallocmap() および rmallocmap_wait() 引数の変更内容

```
struct map *rmallocmap(size_t mapsize);
struct map *rmallocmap_wait(size_t mapsize);
```

rmallocmap(9F) および rmallocmap_wait(9F) ルーチンの mapsize 引数は、ulong_t から size_t に変更されました。

scsi_alloc_consistent_buf() 引数の変更内容

```
struct buf *scsi_alloc_consistent_buf(struct scsi_address *ap,
    struct buf *bp, size_t datalen, uint_t bflags,
    int (*callback )(caddr_t), caddr_t arg);
```

前のリリースでは、datalen は int として定義され、bflags は ulong として定義されました。

uiomove() 引数の変更内容

```
int uiomove(caddr_t address, size_t nbytes,
    enum uio_rw rwflag, uio_t *uio_p);
```

nbytes 引数は long 型として定義されましたが、nbytes はサイズをバイト単位で表すため、size_t のほうが適切です。

cv_timedwait() および cv_timedwait_sig() 引数の変更内容

```
int cv_timedwait(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
int cv_timedwait_sig(kcondvar_t *cvp, kmutex_t *mp,    clock_t timeout);
```

前のリリースでは、cv_timedwait(9F) および cv_timedwait_sig(9F) ルーチンの *timeout* 引数は、long 型として定義されました。これらのルーチンは時間を ticks で表すため、clock_t のほうが適切です。

ddi_device_copy() 引数の変更内容

```
int ddi_device_copy(ddi_acc_handle_t src_handle,
    caddr_t src_addr, ssize_t src_advcnt,
    ddi_acc_handle_t dest_handle, caddr_t dest_addr,
    ssize_t dest_advcnt, size_t bytecount, uint_t dev_datasz);
```

src_advcnt、*dest_advcnt*、*dev_datasz* の各引数では、型が変更されました。これらの引数は、以前はそれぞれ long、long、および ulong_t として定義されました。

ddi_device_zero() 引数の変更内容

```
int ddi_device_zero(ddi_acc_handle_t handle,
    caddr_t dev_addr, size_t bytecount, ssize_t dev_advcnt,
    uint_t dev_datasz):
```

前のリリースでは、*dev_advcnt* は long 型として、*dev_datasz* は ulong_t 型として定義されました。

ddi_dma_mem_alloc() 引数の変更内容

```
int ddi_dma_mem_alloc(ddi_dma_handle_t handle,
    size_t length, ddi_device_acc_attr_t *accattrp,
    uint_t flags, int (*waitfp)(caddr_t), caddr_t arg,
    caddr_t *kaddrp, size_t *real_length,
    ddi_acc_handle_t *handlep);
```

前のリリースでは、*length*、*flags*、および *real_length* は、uint_t、ulong_t、および uint_t * 型で定義されました。

データ共有を処理するルーチンを変更する

デバイスドライバが long またはポインタを含むデータ構造体を、ioctl(9E)、devmap(9E)、または mmap(9E) を使用する 32 ビットアプリケーションと共有し、そのドライバが 64 ビットカーネル用に再コンパイルされている場

合、データ構造体のバイナリレイアウトは互換性がありません。フィールドが現在 long 型で定義され、64 ビットのデータ項目が使用されていない場合は、32 ビット量のままのデータ型を使用するようにデータ構造体を変更します (int および unsigned int)。それ以外の場合、ドライバは、ILP32 と LP64 の構造体の形状の違いを認識して、アプリケーションとカーネルの間でモデルの不一致が発生しているかどうかを判定する必要があります。

データモデルの潜在的な相違を処理するには、ユーザーアプリケーションと直接対話する ioctl()、devmap()、および mmap() の各ドライバエントリポイントを記述して、引数がカーネルと同じデータモデルを使用するアプリケーションから来たかどうかを判定する必要があります。

ioctl() でのデータ共有

モデルの不一致がアプリケーションとドライバの間に存在するかどうかを判定するために、ドライバは FMODELS マスクを使用して、ioctl () *mode* 引数からモデル型を判定します。次の値で OR を使用してモードを有効にし、アプリケーションのデータモデルを特定します。

- FLP64 – アプリケーションは LP64 データモデルを使用します
- FILP32 – アプリケーションは ILP32 データモデルを使用します

326 ページの「64 ビットに対応したデバイスドライバに対する入出力制御のサポート」のコード例は、ddi_model_convert_from(9F) を使用してこの状況を処理する方法を示しています。

devmap() でのデータ共有

64 ビットドライバと 32 ビットアプリケーションでメモリーを共有できるようにするには、64 ビットドライバで生成されるバイナリレイアウトが 32 ビットアプリケーションで消費されるレイアウトと同じである必要があります。アプリケーションにエクスポートされるマッピング済みのメモリーには、データモデル依存のデータ構造体を含める必要があることがあります。

カーネルのデータモデルが変わってもデバイスレジスタのサイズは変わらないため、この問題に直面するメモリーマッピングデバイスはほとんどありません。ただし、ユーザーアドレス空間にマッピングをエクスポートする一部の擬似デバイスでは、別のデータ構造体を ILP32 または LP64 アプリケーションにエクスポートできます。データモデルの不一致が発生しているかどうかを判定するために、devmap(9E) は *model* パラメータを使用して、アプリケーションが予期するデータモデルを記述します。*model* パラメータは、次のいずれかの値に設定されます。

- DDI_MODEL_ILP32 – アプリケーションは ILP32 データモデルを使用します
- DDI_MODEL_LP64 – アプリケーションは LP64 データモデルを使用します

model パラメータは、変換せずに ddi_model_convert_from(9F) ルーチンまたは STRUCT_INIT() に渡すことができます。329 ページの「32 ビットと 64 ビットのデータ構造体マクロ」を参照してください。

mmap() でのデータ共有

mmap(9E) にはデータモデル情報を渡すために使用できるパラメータがないため、ドライバの mmap(9E) エントリポイントを記述して、新しい DDI 関数 ddi_model_convert_from(9F) を使用できます。この関数は、次のいずれかの値を返して、アプリケーションのデータ型モデルを示します。

- DDI_MODEL_ILP32 – アプリケーションは ILP32 データモデルを予期します
- DDI_MODEL_ILP64 – アプリケーションは LP64 データモデルを予期します
- DDI_FAILURE – 関数は mmap(9E) から呼び出されませんでした

ioctl() および devmap() の場合と同様、モデルのビットを ddi_model_convert_from(9F) に渡して、データ変換が必要かどうかを判定できます。または、モデルを STRUCT_INIT() に渡すことができます。

または、devmap(9E) エントリポイントをサポートするようにデバイスドライバを移行します。

x86 ベースのプラットフォームで 64 ビット Long データ型の構造体を確認する

x86 プラットフォームでは、uint64_t などの 64 ビット long 型を使用する構造体を慎重に確認してください。配置やサイズが、32 ビットモードと 64 ビットモードのコンパイルで異なることがあります。次のような例を考えます。

```
#include &lt;studio>
#include &ltsys>

struct myTestStructure {
        uint32_t        my1stInteger;
        uint64_t        my2ndInteger;
};

main()
{
        struct myTestStructure a;

        printf("sizeof myTestStructure is: %d\n", sizeof(a));
        printf("offset to my2ndInteger is: %d\n", (uintptr_t)&a.bar - (uintptr_t)&a);
}
```

32 ビットシステムでは、この例の場合は次の結果が表示されます。

```
sizeof myTestStructure is: 12
offset to my2ndInteger is: 4
```

逆に、64 ビットシステムでは、この例の場合は次の結果が表示されます。

```
sizeof myTestStructure is: 16
offset to my2ndInteger is: 8
```

したがって、32 ビットアプリケーションと 64 ビットアプリケーションでは構造体が異なります。その結果、同じ構造体を 32 ビット環境と 64 ビット環境の両方で実行しようとすると、問題が発生することがあります。この状況は、特に構造体が ioctl() 呼び出しを介してカーネルとの間で受け渡しされる場合によく起こります。

よく知られている ioctl インタフェース

多くの ioctl(9E) 処理は、デバイスドライバのクラスに共通しています。たとえば、ほとんどのディスクドライバは、ioctls の dkio(7I) ファミリの多くを実装しています。これらのインタフェースの多くは、データ構造体をカーネルにコピーしたりカーネルからコピーしたりします。これらのデータ構造体の一部では、LP64 データモデルでサイズが変わりました。次のセクションでは、ioctls の dkio、fdio(7I)、fbio(7I)、cdio(7I)、および mtio(7I) ファミリについて、64 ビットドライバの ioctl ルーチンで明示的な変換が必要になった ioctls を一覧表示します。

ioctl コマンド	影響を受けるデータ構造体	参照
DKIOCGAPART	dk_map	dkio(7I)
DKIOCSAPART	dk_allmap	
DKIOGVTOC	partition	dkio(7I)
DKIOSVTOC	vtoc	
FBIOPUTCMAP	fbcmap	fbio(7I)
FBIOGETCMAP		
FBIOPUTCMAPI	fbcmap_i	fbio(7I)
FBIOGETCMAPI		
FBIOCCURSOR	fbcursor	fbio(7I)
FBIOSCURSOR		
CDROMREADMODE1	cdrom_read	cdio(7I)
CDROMREADMODE2		
CDROMCDDA	cdrom_cdda	cdio(7I)
CDROMCDXA	cdrom_cdxa	cdio(7I)
CDROMSUBCODE	cdrom_subcode	cdio(7I)
FDIOCMD	fd_cmd	fdio(7I)
FDRAW	fd_raw	fdio(7I)
MTIOCTOP	mtop	mtio(7I)

ioctl コマンド	影響を受けるデータ構造体	参照
MTIOCGET	mtget	mtio(7I)
MTIOCGETDRIVETYPE	mtdrivetype_request	mtio(7I)
USCSICMD	uscsi_cmd	scsi_free_consistent_buf(9F)

デバイスのサイズ

nblocks プロパティーは、ブロックデバイスドライバの各スライスによってエクスポートされます。このプロパティーには、デバイスの各スライスがサポートできる 512 バイトのブロックの数が含まれています。nblocks プロパティーは、符号付きの 32 ビット量として定義され、スライスの最大サイズを 1T バイトに制限します。

ディスクあたり 1T バイトを超える記憶領域を提供するディスクデバイスは、Nblocks プロパティーを定義する必要があります。このプロパティーには、デバイスがサポートできる 512 バイトのブロックの数を引き続き含めてください。ただし、Nblocks は符号付きの 64 ビット量であり、ディスク容量に対する実際的な制限は削除されます。

nblocks プロパティーは推奨されなくなりました。すべてのディスクデバイスで Nblocks プロパティーを指定してください。

◆ ◆ ◆ 付 録 D

コンソールフレームバッファードライバ

システムコンソールに使用されるフレームバッファーのドライバは、システムがコンソール上にテキストを表示できるインタフェースを提供する必要があります。Solaris OS には、カーネル端末エミュレータがコンソールフレームバッファー上に直接テキストを表示できる、拡張された視覚的な入出力インタフェースが用意されています。この付録では、必要なインタフェースをフレームバッファードライバに追加することにより、そのドライバが Solaris カーネル端末エミュレータと対話できるようにする方法について説明します。

Solaris コンソールとカーネル端末エミュレータ

カーネル端末エミュレータの役割は、コンソールフレームバッファーに、そのフレームバッファーの画面の高さ、幅、およびピクセルの深さモードによって決定される正しい位置と表現でテキストを描画することです。端末エミュレータはまた、スクロールの操作、ソフトウェアカーソルの制御、および ANSI 端末のエスケープシーケンスの解釈も行います。端末エミュレータは、グラフィックスカードに応じて VGA テキストモードまたはピクセルモードのどちらかでコンソールフレームバッファーにアクセスします。フレームバッファードライバを Solaris コンソールフレームバッファードライバとして使用するには、そのドライバが Solaris カーネル端末エミュレータとの互換性を備えている必要があります。ターゲットプラットフォームは、フレームバッファードライバを変更して Solaris カーネル端末エミュレータとの互換性を確保する必要があるかどうかを判定するための、もっとも重要な要因です。

- x86 プラットフォーム – x86 コンソールフレームバッファードライバは、すでにコンソールのフレームバッファーインタフェースをサポートしているため、コンソールフレームバッファードライバを変更する必要はありません。
- SPARC プラットフォーム – コンソールフレームバッファードライバが Solaris カーネル端末エミュレータと対話できるようにするには、そのドライバはこの付録で説明されているインタフェースを使用するべきです。

x86 プラットフォームのコンソール通信

x86 プラットフォームでは、Solaris カーネル端末エミュレータモジュール (tem) は VGA テキストモードを排他的に使用して vgatext モジュールと対話します。vgatext モジュールは、業界標準の VGA テキストモードを使用して、x86 と互換性のあるフレームバッファーデバイスと対話します。vgatext モジュールは、すでにコンソールのフレームバッファーインタフェースをサポートしているため、x86 フレームバッファードライバはカーネルの tem モジュールと互換性があります。x86 フレームバッファードライバに特殊なインタフェースを追加する必要はありません。

この付録の残りの部分は、SPARC プラットフォームにのみ適用されます。

SPARC プラットフォームのコンソール通信

SPARC フレームバッファードライバは通常、VGA テキストモードでは動作しません。SPARC フレームバッファードライバは通常、表示されるテキストとイメージを示すピクセルパターンを送信する必要があります。カーネルの tem が、画面へのデータの描画、スクロールの実行、およびテキストカーソルの表示を容易にする特定のインタフェースをサポートするには、SPARC ドライバが必要です。ドライバが tem から送信されたデータを実際に画面に描画する方法は、デバイスによって異なります。ドライバは通常、ハードウェアとビデオモードに従って、データをビデオメモリーに描画します。

Solaris OS には、カーネル端末エミュレータが互換性のあるコンソールフレームバッファーを直接操作できるインタフェースが用意されています。ドライバを変換してカーネル端末エミュレータとの互換性を確保する利点は、次のとおりです。

- パフォーマンス (特に、スクロール) の大幅な向上
- ANSI 文字列色の機能の拡張
- システムコンソールのストリームがシリアルポートから出力されている場合でも、コンソールフレームバッファー上でログインセッションを開始する機能

SPARC コンソールフレームバッファードライバに、カーネル端末エミュレータとの互換性を確保する必要はありません。コンソールフレームバッファードライバにカーネル端末エミュレータとの互換性がない場合、システムは、OpenBoot PROM 内の FCode 端末エミュレータを使用します。

コンソールフレームバッファーは、EEPROM の screen 環境変数によって識別されます。システムは、フレームバッファードライバが tem-support DDI プロパティーをエクスポートするかどうかをチェックすることによって、コンソールフレームバッファーにカーネル端末エミュレータモジュールとの互換性があるかどうかを判定します。tem-support プロパティーがエクスポートされる場合、システムは、システムブート時のコンソールの構成中にフレームバッファードライバに VIS_DEVINIT 入出力制御 (ioctl) コマンドを発行します。tem-support DDI プロパティーがエクス

ポートされ、かつ VIS_DEVINIT ioctl コマンドが成功して tem に互換性がある
バージョン番号を返した場合、システムは、そのフレームバッファードライバを
カーネル端末エミュレータ経由で利用するようにシステムコンソールを構成しま
す。入出力制御のドライバエントリポイントについては、ioctl(9E) のマニュアル
ページを参照してください。

カーネル端末エミュレータをサポートする SPARC ドライバは、tem-support DDI プロ
パティーをエクスポートします。このプロパティーは、そのドライバがカーネル端
末エミュレータをサポートすることを示します。フレームバッファードライバが
tem-support DDI プロパティーをエクスポートする場合、そのドライバは、コン
ソールの構成中のブートプロセス内の早い段階で処理されます。フレーム
バッファードライバが tem-support プロパティーをエクスポートしない場合、そのド
ライバは、ブートプロセス内の十分早い段階では処理されない可能性があります。

tem-support　1 に設定されていると、この DDI プロパティーは、このドライバが
コンソールのカーネルフレームバッファーインタフェースと互換性
があることを示します。

カーネル端末エミュレータモジュールは、次の 2 つの主要なインタフェースを介し
てコンソールフレームバッファードライバと相互に作用します。

- 通常のシステム動作中に ioctl インタフェースを介して
- スタンドアロンモード中に、ポーリングされた入出力インタフェースを介して

詳細については、次のセクションで説明します。

コンソールの視覚的な入出力インタフェース

カーネル端末エミュレータは、次の 2 つのインタフェースを介してコンソールフ
レームバッファードライバと相互に作用します。通常のシステム動作中 (システムの
ブートが成功したあと)、カーネル端末エミュレータとコンソールフレーム
バッファードライバの間の通信は ioctl インタフェースを介して行われます。スタン
ドアロンモード中 (システムブートの前またはデバッグ中)、カーネル端末エ
ミュレータとコンソールフレームバッファードライバの間の通信は、ポーリングさ
れた入出力インタフェースを介して行われます。カーネル端末エミュレータとコン
ソールフレームバッファードライバの間のすべての動作は、カーネル端末エ
ミュレータにビデオモードの変更を通知するためにコンソールフレーム
バッファードライバによって使用されるコールバック関数を除き、カーネル端末エ
ミュレータによって開始されます。

コンソールの視覚的な入出力インタフェースについては、visual_io(7I) のマニュア
ルページに詳細が説明されています。ビデオモード変更コールバック関数の詳細に
ついては、665 ページの「ビデオモード変更コールバックインタフェース」を参照して
てください。

入出力制御インタフェース

通常のシステム動作中、カーネル端末エミュレータは、次の表に示されている ioctl インタフェースを介してコンソールフレームバッファードライバと通信します。

ioctl の名前	対応するデータ構造体	説明
VIS_DEVINIT	vis_devinit	端末エミュレータモジュールとフレームバッファーの間のセッションを初期化します。666 ページの「VIS_DEVINIT」を参照してください。
VIS_DEVFINI	該当なし	端末エミュレータモジュールとフレームバッファーの間のセッションを終了します。668 ページの「VIS_DEFINI」を参照してください。
VIS_CONSDISPLAY	vis_consdisplay	ピクセルを矩形として表示します。669 ページの「VIS_CONSDISPLAY」を参照してください。
VIS_CONSCOPY	vis_conscopy	ピクセルの矩形をコピーします(スクロール)。670 ページの「VIS_CONSCOPY」を参照してください。
VIS_CONSCURSOR	vis_conscursor	テキストカーソルの表示/非表示を切り替えます670 ページの「VIS_CONSCURSOR」を参照してください。
VIS_PUTCMAP	vis_cmap	端末エミュレータモジュールのカラーマップをフレームバッファードライバに送信します。671 ページの「VIS_PUTCMAP」を参照してください。
VIS_GETCMAP	vis_cmap	端末エミュレータモジュールのカラーマップをフレームバッファーから読み取ります。672 ページの「VIS_GETCMAP」を参照してください。

ポーリングされた入出力インタフェース

ポーリングされた入出力インタフェースは、VIS_CONSDISPLAY、VIS_CONSCOPY、およびVIS_CONSCURSOR ioctl インタフェースと同じ機能を提供します。ポーリングされた入出力インタフェースは、オペレーティングシステムが休止状態にあり、かつスタンドアロンモードにある場合にのみ呼び出されます。詳細については、672 ページの「コンソールフレームバッファードライバでのポーリングされた入出力の実装」を参照してください。

スタンドアロンモード中、カーネル端末エミュレータは、次の表に示されているポーリングされた入出力インタフェースを介してコンソールフレームバッファードライバと通信します。

ポーリングされた入出力関数	対応するデータ構造体	説明
(*display)()	vis_consdisplay	ピクセルを矩形として表示します。
(*copy)()	vis_conscopy	ピクセルの矩形をコピーします (スクロール)。
(*cursor)()	vis_conscursor	テキストカーソルの表示/非表示を切り替えます

ビデオモード変更コールバックインタフェース

コンソールフレームバッファードライバとカーネル端末エミュレータは、常にビデオモードに関して一致している必要があります。ビデオモードには、コンソール画面の高さ、幅、およびピクセルの深さが含まれます。ビデオモードにはまた、カーネル端末エミュレータとコンソールフレームバッファーの間の通信がVGAテキストモードまたはピクセルモードのどちらになっているかも含まれます。

コンソールフレームバッファードライバがカーネル端末エミュレータにビデオモードの変更を通知するために、コンソールフレームバッファードライバは、次の表で説明されている (*modechg_cb)() カーネル端末エミュレータコールバック関数のアドレスで初期化されます。

コールバック関数	対応するデータ構造体	説明
(*modechg_cb)()	vis_modechg_arg vis_devinit	端末エミュレータモジュールとドライバのビデオモード (画面の高さ、幅、およびピクセルの深さ) との同期を維持します。

コンソールフレームバッファードライバでの視覚的な入出力インタフェースの実装

ビデオモード変更コールバックを除き、ドライバとカーネル端末エミュレータの間のすべての動作は tem (端末エミュレータモジュール) によって開始されます。つまり、tem は、このドキュメントで説明されているすべての ioctl コマンドを発行します。以降のセクションでは、各 ioctl コマンドの実装の詳細について説明します。詳細については、visual_io(7I) のマニュアルページおよび /usr/include/sys/visual_io.h インクルードファイルを参照してください。ビデオモード変更コールバック関数の詳細については、665 ページの「ビデオモード変更コールバックインタフェース」を参照してください。

注 – 各 ioctl コマンドは、ioctl フラグ引数で FKIOCTL が設定されているかどうかを判定し、そのビットが設定されていない場合は EPERM を返します。

VIS_DEVINIT

VIS_DEVINIT ioctl コマンドは、フレームバッファードライバをシステムコンソールデバイスとして初期化します。この ioctl は vis_devinit 構造体のアドレスを渡します。

tem はまず、そのビデオモード変更コールバック関数のアドレスを vis_devinit 構造体の modechg_cb フィールドにロードし、次にそのソフト状態を modechg_arg フィールドに読み込みます。tem は次に、VIS_DEVINIT ioctl コマンドを発行します。フレームバッファードライバは次に、自身を初期化したあと、vis_devinit 構造体内の version、width、height、linebytes、depth、mode、および polledio フィールドを設定することによって、その設定のサマリーを tem に戻します。vis_devinit 構造体を次のコードに示します。

```
struct vis_devinit {
      /*
       * This set of fields are used as parameters passed from the
       * layered frame buffer driver to the terminal emulator.
       */
      int             version;        /* Console IO interface rev */
      screen_size_t   width;          /* Width of the device */
      screen_size_t   height;         /* Height of the device */
      screen_size_t   linebytes;      /* Bytes per scan line */
      int             depth;          /* Device depth */
      short           mode;           /* Display mode Mode */
      struct vis_polledio *polledio;  /* Polled output routines */
      /*
       * The following fields are used as parameters passed from the
       * terminal emulator to the underlying frame buffer driver.
       */
      vis_modechg_cb_t modechg_cb;   /* Video mode change callback */
```

```
        struct vis_modechg_arg *modechg_arg;  /* Mode change cb arg */
};
```

コンソールフレームバッファードライバで VIS_DEVINIT ioctl コマンドを実装するには、次の一般的な手順に従います。

1. コンソール固有の状態を含む struct を定義します。この構造体は、コンソールフレームバッファードライバには非公開です。この構造体は、この付録では consinfo と呼ばれます。consinfo 構造体には、次のような情報が含まれています。
   - Blit バッファーの現在のサイズ
   - Blit バッファーへのポインタ
   - カラーマップ情報
   - 行ピッチなどのドライバ描画モード情報
   - 背景色
   - ビデオメモリーアドレス
   - 端末エミュレータコールバックアドレス
2. 次のメモリーを割り当てます。
   a. もっとも高いビデオの深さで妥当なデフォルトサイズのピクセル矩形を格納できるほど十分に大きな Blit バッファーを割り当てます。受信要求がこのバッファーのサイズを超える場合は、増設メモリーを割り当てることができます。フレームバッファードライバの最大のフォントは 12 × 22 です。DEFAULT_HEIGHT が 12、DEFAULT_WIDTH が 22、最大のビデオの深さが 32 であると仮定すると、バッファーサイズは 8448 バイト (DEFAULT_HEIGHT × DEFAULT_WIDTH × 32) になります。
   b. vis_polledio 構造体を割り当てます。
   c. カーソルを保持するためのバッファーを割り当てます。このバッファーは最大の文字のサイズにします。このバッファーはサイズが変更されません。
3. tem のビデオ変更コールバックアドレスとコールバックコンテキストを modechg_cb と modechg_ctx から取得し、これらの情報を consinfo 構造体に格納します。
4. vis_polledio 構造体に、ポーリングされた表示、コピー、およびカーソル関数のエントリポイントアドレスを設定します。
5. tem からドライバに渡された vis_devinit 構造体の各フィールドに適切な情報を指定します。
   a. version フィールドを /usr/include/sys/visual_io.h ヘッダーファイルで定義された定数である VIS_CONS_REV に設定します。
   b. mode フィールドを VIS_PIXEL に設定します。
   c. polledio フィールドを vis_polledio 構造体のアドレスに設定します。
   d. height フィールドをビデオモードの高さ (ピクセル単位) に設定します。
   e. width フィールドをビデオモードの幅 (ピクセル単位) に設定します。

f. depth フィールドをフレームバッファーのピクセルの深さ (バイト単位) に設定します (たとえば、32 ビットのピクセルの深さは 4 バイトになります)。

g. linebytes フィールドを height × width × depth の値に設定します。

これらの情報は、vis_devinit 構造体を使用してドライバから tem に送信されます。これらの情報は、情報を描画し、それをグラフィックスドライバに渡す方法を端末エミュレータに指示します。

コンソールフレームバッファードライバは、ビデオモード (具体的には、height、width、または depth) を変更した場合は常に、tem のビデオモード変更コールバック関数を呼び出して vis_devinit 構造体を更新し、この構造体を端末エミュレータに戻す必要があります。端末エミュレータは、自身のモード変更コールバック関数アドレスを vis_devinit 構造体の modechg_cb フィールドで渡します。モード変更コールバック関数には、次の関数シグニチャーがあります。

```
typedef void (*vis_modechg_cb_t)
      (struct vis_modechg_arg *, struct vis_devinit *);
```

この typedef に示すように、モード変更コールバック関数は 2 つの引数を取ります。最初の引数は modechg_arg であり、2 番目の引数は vis_devinit 構造体です。modechg_arg は、VIS_DEVINIT ioctl コマンドの初期化中に tem からドライバに送信されます。ドライバは、modechg_arg を各ビデオモード変更コールバックとともに tem に戻す必要があります。

6. カーネルコンソールのコンテキストを初期化します。具体的な要件は、グラフィックスデバイスの機能によって異なります。この初期化には、描画エンジンの状態の設定、パレットの初期化、画面にデータの Blit を実行できるようにするためのビデオメモリーや描画エンジンの検索とマッピングなどの手順が含まれることがあります。
7. vis_devinit 構造体を呼び出し元に返します。

VIS_DEFINI

VIS_DEFINI ioctl コマンドはドライバのコンソールリソースを解放し、セッションを完了します。

コンソールフレームバッファードライバで VIS_DEVFINI ioctl コマンドを実装するには、次の一般的な手順に従います。

1. コンソールフレームバッファードライバの状態をリセットします。
2. ポーリングされた入出力エントリポイントとカーネル端末エミュレータのビデオ変更関数コールバックアドレスをクリアーします。
3. メモリーを解放します。

VIS_CONSDISPLAY

VIS_CONSDISPLAY ioctl コマンドは、ピクセルの矩形を指定された場所に表示します。この表示は、矩形の *Blit* とも呼ばれます。vis_consdisplay 構造体には、ドライバと tem の両方が使用しているビデオの深さで矩形を描画するために必要な情報が含まれています。vis_consdisplay 構造体を次のコードに示します。

```
struct vis_consdisplay {
      screen_pos_t    row;      /* Row (in pixels) to display data at */
      screen_pos_t    col;      /* Col (in pixels) to display data at */
      screen_size_t   width;    /* Width of data (in pixels) */
      screen_size_t   height;   /* Height of data (in pixels) */
      unsigned char   *data;    /* Address of pixels to display */
      unsigned char   fg_color; /* Foreground color */
      unsigned char   bg_color; /* Background color */
};
```

コンソールフレームバッファードライバで VIS_CONSDISPLAY ioctl コマンドを実装するには、次の一般的な手順に従います。

1. vis_consdisplay 構造体をコピーします。
2. 表示パラメータを検証します。いずれかの表示パラメータが範囲外の場合はエラーを返します。
3. ビデオメモリーに Blit が実行される矩形のサイズを計算します。このサイズを、VIS_DEVINIT 中に作成された Blit バッファーのサイズに対して検証します。必要に応じて、Blit バッファーのための増設メモリーを割り当てます。
4. Blit データを取得します。このデータは、一致したピクセルの深さで、カーネル端末エミュレータによって準備されています。その深さは、VIS_DEVINIT 中に tem から伝達されたピクセルの深さと同じです。ピクセルの深さは、デバイスドライバが tem へのコールバックを通してビデオモードを変更した場合は常に更新されます。標準的なピクセルの深さは、8 ビットのインデックスカラーマップと 32 ビットのトゥルーカラーです。
5. ユーザーアプリケーションがユーザーメモリーマッピングを通してフレームバッファーハードウェアに同時にアクセスすることができないように、すべてのユーザーコンテキストを無効化します。ポーリングされた入出力モードでは、ユーザーアプリケーションが実行されていないため、この手順は許可されず、また必要でもありません。VIS_CONSDISPLAY ioctl が完了するまでユーザーがページフォルトを通してマッピングを復元することができないように、必ずロックを保持するようにしてください。
6. ドライバ固有のコンソール描画コンテキストを確立します。
7. フレームバッファーが 8 ビットのインデックスカラーモードで実行されている場合は、tem が前の VIS_PUTCMAP ioctl を使用して設定したカーネルコンソールのカラーマップを復元します。パフォーマンスを最適化するために、遅延カラーマップをロードするスキームをお勧めします。遅延スキームでは、コンソールフレームバッファーは、VIS_DEVINIT ioctl が発行されたあとに実際に使用した色のみを復元します。

8. tem から渡されたデータを tem から送信されたピクセル座標に表示します。RGB ピクセルデータのバイト順序の変換が必要になることがあります。

VIS_CONSCOPY

VIS_CONSCOPY ioctl コマンドは、ピクセルの矩形領域をある場所から別の場所にコピーします。この ioctl の 1 つの使用法にスクロールがあります。

コンソールフレームバッファードライバで VIS_CONSCOPY ioctl コマンドを実装するには、次の一般的な手順に従います。

1. vis_conscopy 構造体をコピーします。vis_conscopy 構造体では、ソースとターゲットの矩形のサイズと場所を記述します。
2. 表示パラメータを検証します。いずれかの表示パラメータが範囲外の場合はエラーを返します。
3. ユーザーアプリケーションがユーザーメモリーマッピングを通してフレームバッファーハードウェアに同時にアクセスすることができないように、すべてのユーザーコンテキストを無効化します。ポーリングされた入出力モードでは、ユーザーアプリケーションが実行されていないため、この手順は許可されず、また必要でもありません。VIS_CONSDISPLAY ioctl が完了するまでユーザーがページフォルトを通してマッピングを復元することができないように、必ずロックを保持するようにしてください。
4. 矩形をコピーするための関数を呼び出します。

注 – 最適なパフォーマンスを得るには、グラフィックスデバイスの描画エンジンを使用してコピー関数を実装してください。最適なパフォーマンスが得られるように描画エンジンを設定するには、ドライバ内でどのようにコンテキスト管理を実行するかを決定する必要があります。

VIS_CONSCURSOR

VIS_CONSCURSOR ioctl コマンドは、カーソルの表示/非表示を切り替えますvis_conscursor 構造体を次のコードに示します。

```
struct vis_conscursor {
      screen_pos_t    row;      /* Row to display cursor (in pixels) */
      screen_pos_t    col;      /* Col to display cursor (in pixels) */
      screen_size_t   width;    /* Width of cursor (in pixels) */
      screen_size_t   height;   /* Height of cursor (in pixels) */
      color_t         fg_color; /* Foreground color */
      color_t         bg_color; /* Background color */
      short           action;   /* Show or Hide cursor */
};
```

コンソールフレームバッファードライバで VIS_CONSCOPY ioctl コマンドを実装するには、次の一般的な手順に従います。

1. vis_conscursor 構造体をカーネル端末エミュレータからコピーします。
2. 表示パラメータを検証します。いずれかの表示パラメータが範囲外の場合はエラーを返します。
3. ユーザーアプリケーションがユーザーメモリーマッピングを通してフレームバッファーハードウェアに同時にアクセスすることができないように、すべてのユーザーコンテキストを無効化します。ポーリングされた入出力モードでは、ユーザーアプリケーションが実行されていないため、この手順は許可されず、また必要でもありません。VIS_CONSDISPLAY ioctl が完了するまでユーザーがページフォルトを通してマッピングを復元することができないように、必ずロックを保持するようにしてください。
4. 端末エミュレータは、SHOW_CURSOR または HIDE_CURSOR の 2 つのアクションのいずれかを使用して VIS_CONSCOPY ioctl を呼び出すことができます。次の手順では、ビデオメモリーの読み取りと書き込みによってこの機能を実装する方法について説明します。また、この作業を行うために描画エンジンを使用できる可能性もあります。描画エンジンを使用できるかどうかは、フレームバッファーハードウェアに依存します。

   SHOW_CURSOR 機能を実装するには、次の手順を実行します。

   a. カーソルが描画される矩形内のピクセルを保存します。これらの保存されたピクセルは、カーソルを非表示にするために必要になります。
   b. カーソルが描画される矩形で区切られた画面上のすべてのピクセルをスキャンします。この矩形内の、指定されたカーソル前景色 (fg_color) に一致するピクセルを白いピクセルに置き換えます。指定されたカーソル背景色 (bg_color) に一致するピクセルを黒いピクセルに置き換えます。この視覚的な効果として、白いテキストの上に黒いカーソルが表示されます。この方法は、テキストの任意の前景色と背景色で機能します。カラーマップの位置に基づいて色を反転させようとしても実行できません。HSB 配色 (色合い、彩度、明度) を使用して色の反転を試みるなどの、より高度な方式は必要ありません。

   HIDE_CURSOR 機能を実装するには、カーソルの矩形の下にあるピクセルを、前の SHOW_CURSOR アクションで保存されたピクセルに置き換えます。

VIS_PUTCMAP

VIS_PUTCMAP ioctl コマンドは、コンソールのカラーマップを確立します。端末エミュレータは、カーネルのカラーマップを設定するためにこの関数を呼び出します。vis_cmap 構造体を次のコードに示します。この構造体は、8 ビットのインデックスカラーモードにのみ適用されます。

```
struct vis_cmap {
      int             index;  /* Index into colormap to start updating */
```

```
      int             count;  /* Number of entries to update */
      unsigned char   *red;   /* List of red values */
      unsigned char   *green; /* List of green values */
      unsigned char   *blue;  /* List of blue values */
};
```

VIS_PUTCMAP ioctl コマンドは、FBIOPUTCMAP コマンドに似ています。VIS_PUTCMAP コマンドは、フレームバッファーの端末エミュレータと互換性があるコンソールコードに固有です。

VIS_GETCMAP

端末エミュレータは、コンソールのカラーマップを取得するために VIS_GETCMAP ioctl コマンドを呼び出します。

コンソールフレームバッファードライバでのポーリングされた入出力の実装

ポーリングされた入出力インタフェースはドライバ内の関数として実装され、カーネル端末エミュレータから直接呼び出されます。ドライバは、VIS_DEVINIT ioctl コマンドの実行中に、自身のポーリングされた入出力エントリポイントのアドレスを端末エミュレータに渡します。VIS_DEVINIT コマンドは、端末エミュレータによって開始されます。

vis_polledio 構造体を次のコードに示します。

```
typedef void * vis_opaque_arg_t;

struct vis_polledio {
      struct vis_polledio_arg *arg;
      void    (*display)(vis_opaque_arg_t, struct vis_consdisplay *);
      void    (*copy)(vis_opaque_arg_t, struct vis_conscopy *);
      void    (*cursor)(vis_opaque_arg_t, struct vis_conscursor *);
};
```

ポーリングされた入出力インタフェースは、VIS_CONSDISPLAY、VIS_CONSCOPY、および VIS_CONSCURSOR ioctl インタフェースと同じ機能を提供します。ポーリングされた入出力インタフェースは、対応する ioctl コマンドに関して上で説明されたのと同じ手順に従います。ポーリングされた入出力インタフェースは、このセクションの残りの部分で説明されている追加の制限に非常に厳密に従う必要があります。

ポーリングされた入出力インタフェースは、オペレーティングシステムが休止状態にあり、かつスタンドアロンモードにある場合にのみ呼び出されます。システムは、ユーザーが OpenBoot PROM に入った場合、または kmdb デバッガに入った場合には常にスタンドアロンモードに入り、システムがパニック状態になった場合にもこのモードに入ります。1 つの CPU と 1 つのスレッドのみがアクティブです。ほか

の CPU とスレッドはすべて停止されます。タイムシェアリング、DDI 割り込み、およびシステムサービスはオフに設定されます。

スタンドアロンモードではドライバ機能が厳しく制限されますが、ドライバの同期要件は簡素化されます。たとえば、ユーザーアプリケーションが、ポーリングされた入出力ルーチン内からドライバのメモリーマッピングを通してコンソールフレームバッファードライバにアクセスすることはできません。

スタンドアロンモードでは、コンソールフレームバッファードライバは、次のどのアクションも実行してはいけません。

- 割り込みの待機
- mutex の待機
- メモリーの割り当て
- DDI または LDI インタフェースの使用
- システムサービスの使用

ポーリングされた入出力関数は比較的単純な操作であるため、これらの制限に従うことは難しくありません。たとえば、描画エンジンを操作する場合、コンソールフレームバッファードライバは割り込みを待機するのではなく、このデバイス内のビットをポーリングできます。ドライバは、割り当て済みのメモリーを使用して Blit データを描画できます。DDI または LDI インタフェースは必要ないはずです。

フレームバッファー固有の構成モジュール

ドライバ固有の `fbconfig()` モジュールのために解像度または発色数の変更が発生した場合、その `fbconfig()` モジュールはフレームバッファードライバに `ioctl` を送信する必要があります。この `ioctl` により、フレームバッファードライバは、新しい画面サイズと深さを指定して端末エミュレータのモード変更コールバック関数を呼び出します。フレームバッファードライバと端末エミュレータは、常にビデオモードに関して一致している必要があります。フレームバッファードライバと端末エミュレータがビデオモードに関して一致していない場合、画面上の情報は読み取り不可能になり、無意味になります。

X Window System のフレームバッファー固有の DDX モジュール

X Window System が終了してコマンド行に戻るとき、フレームバッファーの DDX モジュールは、フレームバッファードライバに `ioctl` を送信する必要があります。この `ioctl` により、フレームバッファードライバは、端末エミュレータのモード変更コールバック関数を呼び出します。X Window System が起動したあと、終了する前にビデオ解像度を変更した場合は、この通信によってフレームバッファードライバと

端末エミュレータでのビデオモードの一致が維持されます。フレームバッファードライバと端末エミュレータは、常にビデオモードに関して一致している必要があります。フレームバッファードライバと端末エミュレータがビデオモードに関して一致していない場合、画面上の情報は読み取り不可能になり、無意味になります。

コンソールフレームバッファードライバの開発、テスト、およびデバッグ

アクティブなシステム上でのコンソールフレームバッファードライバのデバッグでは、問題が発生する場合があります。

- システムブートの早い段階で検出されたエラーでは、コアダンプが生成されません。
- エラーまたは通知メッセージが、画面上に正しく表示されない可能性があります。
- USB キーボードの入力が失敗する可能性があります。

このセクションでは、コンソールフレームバッファードライバの開発、テスト、およびデバッグに役立ついくつかの提案を示します。

入出力制御インタフェースのテスト

`ioctl` コマンドをテストするには、ユーザーアプリケーションから呼び出し可能な追加の `ioctl` エントリポイントを作成します。引数を適切にコピーする必要があります。ユーザーのアドレス空間との間でデータを転送するには、`ddi_copyin(9F)` ルーチンと `ddi_copyout(9F)` ルーチンを使用します。次に、描画、スクロール、およびカーソル動作を検証するためのアプリケーションを記述します。これによって、これらの `ioctl` コマンドの開発またはテスト中にコンソールが影響を受けることはなくなります。

`ioctl` コマンドが正しく機能することを確認するには、システムをブートしてログインします。`prstat`(1M)、`ls`(1)、`vi`(1)、`man`(1) などのコマンドを実行したときに予期した動作が得られるかどうかをチェックします。

ANSI カラーが正しく機能することを検証するには、次のスクリプトを実行します。

```
#!/bin/bash
printf "\n\n\n\e[37;40m             Color List       \e[m\n\n"
printf "\e[30m Color 30 black\e[m\n"
printf "\e[31m Color 31 red\e[m\n"
printf "\e[32m Color 32 green\e[m\n"
printf "\e[33m Color 33 yellow\e[m\n"
printf "\e[34m Color 34 blue\e[m\n"
printf "\e[35m Color 35 purple\e[m\n"
```

```
printf "\e[36m Color 36 cyan\e[m\n"
printf "\e[37m Color 37 white\e[m\n\n"
printf "\e[40m Backlight 40 black \e[m\n"
printf "\e[41m Backlight 41 red   \e[m\n"
printf "\e[34;42m Backlight 42 green \e[m\n"
printf "\e[43m Backlight 43 yellow\e[m\n"
printf "\e[37;44m Backlight 44 blue  \e[m\n"
printf "\e[45m Backlight 45 purple\e[m\n"
printf "\e[30;46m Backlight 46 cyan  \e[m\n"
printf "\e[30;47m Backlight 47 white \e[m\n\n"
```

ポーリングされた入出力インタフェースのテスト

ポーリングされた入出力インタフェースは、次の状況でのみ使用できます。

- L1+A のキーストロークシーケンスを使用して OpenBoot PROM に入る場合
- kmdb(1) などのスタンドアロンデバッガを使用してシステムをブートする場合
- システムがパニック状態になった場合

ポーリングされた入出力インタフェースは、ブートプロセス内の特定の時点でのみ使用可能になります。システムが稼働する前に OpenBoot PROM から発行され、ポーリングされた入出力要求は描画されません。同様に、コンソールが構成される前に発行された kmdb プロンプトは描画されません。

ポーリングされた入出力インタフェースをテストするには、L1+A のキーストロークシーケンスを使用して OpenBoot PROM に入ります。ポーリングされた入出力インタフェースが使用されていることを検証するには、OpenBoot PROM の ok プロンプトで次のコマンドを入力します。

```
ok 1b emit ." [32m This is a test" 1b emit ." [m"
```

次の文章の内容が当てはまる場合、ポーリングされた入出力インタフェースは正しく機能しています。

- 上のコマンドを実行すると、This is a test というフレーズが緑色で表示される。
- OpenBoot PROM が引き続き正しく機能している。
- スクロールが予期したとおりに実行される。
- カーソルが正しく表示される。
- システムに繰り返し入って続行できる。

ビデオモード変更コールバック関数のテスト

ビデオモード変更コールバック関数が正しく機能するかどうかを判定するには、システムにログインし、fbconfig(1M) を使用して解像度とフレームバッファーの深さを複数回変更します。コンソールにテキストが引き続き正しく表示される場合、ビデオモード変更コールバック関数は正しく機能しています。異なる画面サイズに対

応するためにカーネル端末エミュレータがフォントサイズを調整する可能性がありますが、それはコンソールフレームバッファードライバにとって重大な問題ではありません。

X Window System とコンソールフレームバッファードライバが相互に正しく作用しているかどうかを判定するには、X Window System のビデオ解像度とコマンド行の解像度を異なる方法で変更しているときに、X Window System とコマンド行を複数回切り換えます。X Window System が終了したときにコンソール文字が正しく表示されない場合は、X Window System がドライバのコンソールコードにビデオモードが変更されたことを通知しなかったか、またはドライバがカーネル端末エミュレータのビデオモード変更コールバック関数を呼び出さなかったかのどちらかです。

コンソールフレームバッファードライバをテストするための追加の提案

ブート中に、システムがカーネル端末エミュレータと互換性があるフレームバッファードライバを見つけられないか、または正常にロードできなかった場合、システムは `/var/adm/messages` にメッセージを送信します。これらのメッセージを監視するには、別のウィンドウで次のコマンドを入力します。

```
% tail -f /var/adm/messages
```

ドライバをデバッグしているときの USB に関する問題を回避するには、EEPROM の `input-device` NVRAM 構成パラメータが、キーボードの代わりにシリアルポートを使用するように変更します。このパラメータの詳細については、`eeprom(1M)` のマニュアルページを参照してください。

◆ ◆ ◆ 付 録 E

pci.conf ファイル

このセクションでは、pci.conf ファイル、その使用方法、および構文について説明します。

説明

pci.conf が導入された目的は、システム上の特定の PF (物理機能) の VF (仮想機能) の数など、PCI 構成を保存することです。このファイルには、次の目的があります。

- ブート時に VF を自動的に作成できるように PCI 構成を持続させること。
- この構成ファイルは boot_archive の一部になっているため、ブート中に VF を使用できます。

注 – /etc/pci.conf ファイルを Oracle Solaris ブートプロセスに含めるには、/etc/pci.conf を /boot/solaris/filelist.ramdisk ファイルに追加します。

VF が基本的なシステム上で使用される場合、このファイルは非 IOV システム構成でも使用されます。現在のところ、これに含まれるのは VF に関する構成だけです。将来は、PCI バス固有のその他の構成や、デバイス固有の回避方法さえも、これに含まれるようになる可能性があります。VF の数の構成は、次に示すように「[System_Configuration]」セクションに保存されます。

```
[System Configuration]
[[path=<pf_device_path>]]
num-vf=<num_of_vf>
```

システム構成セクション

このファイルの [System Configuration] セクションは、Oracle Solaris PCIe フレームワークによって解釈されます。認識されないキーワードにはエラーのフラグが付けられます。[System Configuration] セクションは、ファイル全体で 1 つだけ存在し、ファイルの先頭に存在している必要があります。

[System Configuration] セクションは一連のサブセクションから構成されます。各サブセクションには、一意のテキストラベルが 1 つと、そのあとに、対象デバイスに一致する二重角括弧で囲まれたフィルタのリストが含まれている必要があります。各サブセクションの内容は、一致したデバイスのそれぞれに対してフレームワークが実行するアクションのリストです。例:

```
[System Configuration]
     new_e1kg_driver [[id=0x8086,0x1000,,0x108e,]] [[classcode=0x020000]]
     num-vf=4
```

二重括弧内のフィルタは、デバイス ID 0x1000、Sun のサブシステムベンダー ID、およびネットワークコントローラクラスコードを持つ、システム内のすべての Intel デバイスに一致します。Oracle Solaris はデバイスの VF 数を 4 に設定します。

注 – フィルタ内でデバイスパスを使用すれば、フィルタのスコープを単一のデバイスインスタンスに絞り込むことができます。

デバイス構成セクション

[Device Configuration] セクションも [System Configuration] セクションと同じタイプのサブセクションから構成されますが、その内容を解釈するのはフィルタに一致したデバイスのドライバだけであるという違いがあります。

```
Device Configuration]
     # label must be file-globally unique
     igbe-sriov-test [[path=/pci@0,0/network@2]]
     # number of rx/tx ring pairs for each VF
     dma-channel-distribution=2,2,8,4
```

構文

```
/etc/pci.conf = <system section><device section>
<system section> = "[System_Configuration]" {<framework subsection>}*
<device section> = "[Device_Configuration]" {<device subsection>}*
<framework subsection> = <label> [ {filter>}* ]
[ {<framework action>}+ ]
<device subsection> = <label> [ {<filter>}+* ] <devicenvlist>
<filter> = [ "[[" "id" "=" (vendorid>) "," (<deviceid>) "," (<revisionid>) "," (<subsystem-vendorid>) ","
```

```
(<subsystemid>)"]]"]
| ["[[" "classcode" "=" <classcode> (,<mask>) "]]" ] | [ "[[" "path"
"=" ,devpath> "]]"]
<framework action> = [ "num-vf" "=" <val>]
<deice nvlist> = [ {name string> "=" <value string>} +]
```

参照

pci_param_get(9F)も参照してください。

索引

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

と

な

に

ね

は