システムソフトウェアとオペレーティング・システム 86－6
(2001. 3. 5)

ITRONデバイスドライバ設計ガイドラインの移植性と評価

村中 健二　高田 広章
豊橋技術科学大学
〒441-8580 豊橋市天伯町雲雀ヶ丘 1-1
{muranaka,hiro}@ertl.ics.tut.ac.jp

児玉 剛
アルパイン情報システム株式会社
t-kodama@aisi.alpine.co.jp

宿口 雅弘
三菱電機マイコン機器ソフトウエア株式会社
ms89019@mms.co.jp

館 義宏
セイコーエプソン株式会社
Tachi.Yoshihiro@exc.epson.co.jp

綱川 寧孝
ソニー株式会社
セミコンダクタネットワークカンパニー
tunakawa@sldc.sony.co.jp

横澤 彰
株式会社東芝
セミコンダクター社
akira.yokosawa@toshiba.co.jp

あらまし
近年の組み込みシステムにおいて, プロセッサにつながる周辺デバイスは多種多様になりつつあり, また開発サイクルも短く, デバイスドライバの再利用性向上が強く望まれている。この様な中, 「ITRONデバイスドライバ設計ガイドラインワーキンググループ」では, デバイスドライバの設計指針を示すガイドラインの策定を行っている。ITRONデバイスドライバ設計ガイドラインではデバイスドライバ自体の移植性を確保するための階層化構造を導入している。この階層構造では, デバイスドライバをターゲットシステムのカーネル, プロセッサ, 割り込みコントローラ (IRC) , デバイスの結線方法により階層分けを行い, デバイスドライバの移植性を向上させている。本稿ではデバイスドライバ設計ガイドラインの概説を行い, デバイスドライバ開発時に良く起るエンディアン問題を通してデバイスドライバの移植性をどのように向上させるかを示す。また実装例を用いて移植性の評価を行う。

キーワード　デバイスドライバ, 移植性, 設計, ガイドライン

Portability and Evaluation of ITRON Device Driver Design Guideline

Kenji Muranaka　Hiroaki Takada
Toyohashi Univ. of Technology
1–1 Hibarigaoka, Tempaku-cho,
Toyohashi 441-8580, Japan
{muranaka,hiro}@ertl.ics.tut.ac.jp

Tsuyoshi Kodama
ALPINE Information System Inc.
t-kodama@aisi.alpine.co.jp

Masahiro Shukuguchi
Mitsubishi Electric Micro-Computer
Application Software Co.,Ltd.
ms89019@mms.co.jp

Yoshihiro Tachi
SEIKO EPSON CORPORATION
Tachi.Yoshihiro@exc.epson.co.jp

Yasutaka Tsunakawa
Semiconductor Network Company,
Sony Corporation
tunakawa@sldc.sony.co.jp

Akira Yokosawa
Toshiba Corporation
Semiconductor Company
akira.yokosawa@toshiba.co.jp

Abstract
In recently embedded systems, becase devices that are connected to a processor are increasingly varied and because of short development cycle, reusability of device driver is strongly required. In this background, "ITRON Device Driver Design Guideline Working Group" is investigating on a guideline which gives design rules of device drivers. This guideline adopts a layered structure to improve portability of device drivers. This layered structure separates dependencies on kernel, processor, interrupt request controler, connection method of devices to different layers, then improves the portability. In this paper, we give an outline of the ITRON Device Driver Design Guideline, and show how the portability is improved through an example solution of an endian problem which frequently occurs on development of device drivers.We also evaluate portability with a sample implementation.

Key words　Device Driver, Portability, Design, Guideline

1 はじめに

近年, 組み込みシステムにおいてシステムの多様化・複雑化に伴って使用される周辺デバイスも同様に多様化・複雑化している。デバイスドライバの開発はタイミングの制約, ハードウェアとの密接な絡み, ハードウェアのマニュアルの理解等が必要であり, 他のソフトウェアと違った独特の労力を要する。また製品・システム開発の短工期化といった現状もあり, デバイスドライバ自体の流通を促し, 再利用することで工期を削減することが望まれている。しかし, デバイスドライバはターゲットのハードウェアに密接に絡むため, 移植性を上げることは容易ではない。

このような背景で,「ITRON デバイスドライバ設計ガイドラインワーキンググループ」では, デバイスドライバの移植性を高めることを目的としたガイドラインの策定を行っている。このガイドラインではデバイスドライバのハードウェア・ソフトウェアの依存部を階層構造を用いて分割することで移植性の向上を目指している。

一般にデバイスを扱うためにはプロセッサを経由してアクセスし, またデバイスからの割り込み要求は割り込みコントローラを経由してプロセッサに伝えられる。つまり, デバイスドライバはデバイス以外にプロセッサと割り込みコントローラに依存する。また, デバイスがレジスタを持つ場合それがシステム上で何番地にマッピングされるかやアクセス方法が異なったり, デバイス特有の入出力方法等デバイスのシステム上での実装 (結線) 方法が異なる。このガイドラインではこのようなデバイスドライバの依存要因を分割し, 各要因毎に階層分けを行うことでデバイスドライバの移植性向上を目指している。

ここで, 他のアーキテクチャとの違いを上げると,

- Linux ではデバイスドライバとカーネルとのインターフェース, デバイスドライバとターゲットハードウェアとのインターフェースは設けられているが, ターゲットシステムにおいてデバイスがどのように実装 (結線) されているかの違いを吸収する階層は設けられておらず, デバイスドライバの移植性は決して良くないない。[3]
- UDI[1]では, 汎用的な環境を想定してデバイスドライバと OS とのインターフェースを規定しているが, デバイスドライバのインターフェースを統一し, デバイスの機能を抽象化してしまうことは組み込みシステムにおいては必ずしも好ましいとは言えない。[4]
- UDI,Linux において, デバイスのターゲットシステム上での実装方法の違いを吸収するためのガイドライン等は無く多様な組み込みシステムにおける開発時にはこういったデバイスの実装方法に応じたデバイスドライバの設計指針を与えることも重要であると考えられる。このガイドラインでは割り込みコントローラやデバイス自体のターゲットシステムでの実装方法を明確にしソフトウェアの開発をスムーズに行うことも目的としている。

といったことが挙げられる。

以下本稿では,2 節で ITRON デバイスドライバ設計ガイドラインの概要を説明し, 3 節でエンディアン問題を取り上げ, この問題をこのアーキテクチャでどのように解決しデバイスドライバの移植性を上げるかを示す。4 節ではこのガイドラインに基づいて実際に実装した LAN コントローラのデバイスドライバの移植性を検証する。

[1]Uniform Driver Interface : 汎用的な OS のためのデバイスドライバ規格

2 デバイスドライバ設計ガイドライン

この節ではデバイスドライバ設計ガイドラインの概要を示す。

2.1 ガイドラインの策定動機・目的

ガイドラインの策定動機を以下に挙げる。

- ソフトウェア開発者からの動機 :
  デバイスドライバはソフトウェアのサイズが小さい割に設計に高度なノウハウが必要でありデバッグも難しいと言われている。ソフトウェア開発者の立場からはこのようなソフトウェアは再利用することで開発工数を削減したい。そのため, デバイスドライバの再利用性, 流通性を高めるガイドラインが求められる。またガイドラインに従って開発することで品質の高いデバイスドライバが開発できればメリットは大きいと言える。
- デバイス提供者からの動機 :
  デバイスを提供する半導体メーカからは, デバイスにデバイスドライバを付加することで付加価値を高め, さらに容易にデバイスを利用できるようにすることでデバイスの普及を高めたい。そのためにはデバイスドライバの再利用性, 流通性に加えて広い適用性が必要になる。つまり, プロセッサ, 割り込みコントローラなどのターゲットハードウェアやカーネルに依存しないデバイスドライバの形態が要求される。このようなガイドラインがあればメリットは大きいと言える。
- 技術者教育面からの動機 :
  技術者の教育面からは, ガイドラインがデバイスドライバ作成の教科書となると利用価値が大きい。さらに, 用語の意味を統一することで, 技術者間での意思の疎通が的確に行える効果もある。
- ITRON プロジェクトからの動機 :
  μITRON 仕様カーネルの問題点の一つとして, デバイスドライバがそろっていないことが挙げられる。μITRON 仕様にはデバイスドライバに関する API がないため, デバイスドライバの形態が標準化されていない。このためにデバイスドライバの流通が促せないことが原因として考えられる。

以上で述べた動機より, このガイドライン策定の目的として, 次の 4 つを挙げることができる。

1. μITRON 仕様カーネル用のデバイスを扱うソフトウェアモジュールの標準的なモデルを提示し, その流通を促すこと。このガイドラインに従って構築されたデバイスを扱うソフトウェアモジュールを,**Device Interface Component** (**DIC** と略す) と呼ぶ。
2. デバイスを扱う最低限のインターフェースソフトウェア (これを **Primitive DIC** と呼び,**PDIC** と略す) を, プロセッサやカーネルに依存せずに記述する枠組みを与えること。PDIC の適用性をさらに広げるために,PDIC はカーネルを用いない場合にも適用できることを想定する。
3. ガイドラインにしたがって DIC を設計することで, 熟練していない技術者にも安定した品質の DIC が開発できるようにすること。

4. デバイスを扱うソフトウェアに関連する用語の意味を標準化すること。

2.2 DIC の基本アーキテクチャ

2.2.1 DIC のコンセプト

DIC とはソフトウェアからデバイスのアクセスを容易にするためのインターフェースソフトウェアで, このガイドラインにしたがって構築されたものを DIC(Device Interface Component) と呼ぶ。

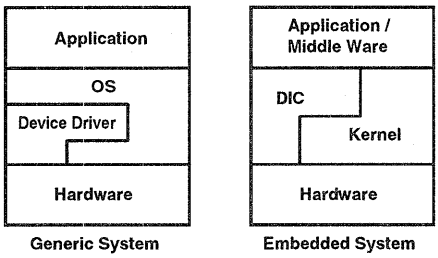

図 1: DIC の位置づけ

組み込みシステムでは扱うデバイスがアプリケーション毎に多種多様であり, 限られたリソースでシステムを構築しなければならないため, デバイスドライバとカーネルのインターフェースを規定しデバイスの機能をカーネル内に隠蔽してしまうことは必ずしも適しているとはいえない。デバイスの機能を最大限活用するには DIC をカーネル内に隠蔽し機能を制限してしまうのではなく, 上位アプリケーションまたはミドルウェアに最適なインターフェースを用意し実装する方が好ましい。従って,DIC アーキテクチャでは図 1に示すようにデバイスドライバとカーネルとのインターフェースを規定することを一義的な目的とはしていない。DIC 自体の移植性を向上させる設計ガイドラインを提示し, 再利用性を高めることを目的の一つとしている。

2.2.2 DIC の階層構造

DIC アーキテクチャでは依存する対象ハードウェアまたはソフトウェアによってソフトウェアモジュールの階層化を行っている。

デバイスドライバの移植性を阻害する要因には以下に示すような要因が考えられる。

(1) プロセッサ・割り込みコントローラ (IRC)

(2) デバイスのボード上での実装 (結線) 方法

(3) カーネル

具体的には, 各要因で以下のような違いがあげられる。(1) ではプロセッサの違いにより入出力命令や DIC の実装に必要なタイミングの生成方法が異なる。またプロセッサ・割り込みコントローラにより割り込み管理機構が異なる。(2) ではデバイスのボード上での実装 (結線) 方法によりデバイスレジスタのマッピングや, 入出力方法, デバイス特有の端子制御方法などが異なる。(3) はカーネルの API が異なる。

これらの移植性阻害要因毎に階層分離し実装することでソフトウェアモジュールの移植性・再利用性を向上させることができる。

DIC アーキテクチャでは移植性を向上させるために図 2に示す階層構造を用いている。この階層構造ではターゲットハードウェアを直接アクセスしターゲットシステムのプロセッサ・割り込み機構を抽象化する SIL (System Interface Layer), ターゲットデバイスの結線 (実装) 方法を抽象化したアクセスインターフェース, ターゲットのデバイス自身の機能を抽象化した PDIC (Primitive DIC), さらにカーネルの機構を用いてデバイスの機能を抽象化した GDIC (General DIC) の 4 つのソフトウェア階層に分けられる。

PDIC はデバイスの持つ機能をそのまま提供し, ソフトウェアによる機能の追加を行わない。PDIC をカーネルがない場合も用いることができるように, カーネルの持つタスク管理機能を利用することはできない。したがって,PDIC 内で待ち状態に入ることはない。

SIL はある特定の PDIC に特化したものではなくあくまでターゲットシステム上でのプロセッサおよび割り込みシステムの機能のみを隠蔽することを目的とする。

これに対し PDIC に属するアクセスインターフェースは PDIC をデバイスがターゲットシステム上でどのように実装 (結線) されているかを意識することなく実装するためのインターフェースである。これらによりターゲットプロセッサや割り込み機構, ボード上でのデバイスの実装方法を隠蔽し,PDIC の移植性を向上させることができる。

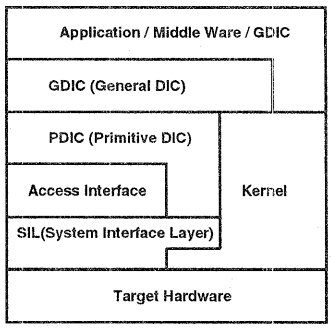

図 2: DIC の階層構造

2.3 システムインターフェースレイヤー (SIL)

システムインターフェースレイヤーは DIC の開発時に必要なプロセッサと割り込みコントローラの機能を抽象化したインターフェース層を提供する。具体的には,

- デバイスとのデータ入出力
- 微小時間待ち
- 割り込みロック状態の制御
- ISR の管理

の 4 つの機能を提供する。図 3に SIL の構成を示す。SIL はターゲットシステムに依存し, ターゲットデバイスの DIC には依存しない。SIL は DIC の実装者ではなくターゲットのシステム開発者が別途用意する。このためには PDIC から呼び出せる標準的な API を定める必要がある。

デバイスとのデータ入出力 API

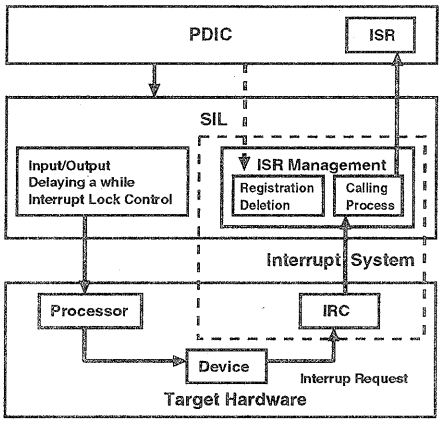

図 3: システムインターフェースレイヤ

ターゲットデバイスとのデータ入出力にはターゲットハードウェアに応じて以下のような違いがある。

- アドレス空間（メモリ空間または IO 空間）
- データサイズ
- エンディアン

よって, これらの違い毎に API を定めることでデバイスとの入出力方法におけるプロセッサの機能を抽象化できる。

表 1から表 3に入出力 API の一覧を示す。[2]

表 1: ナチュラルエンディアン用入出力 API

データサイズ	入出力方向	アドレス空間	
		メモリ空間	IO 空間
ワード	書き込み	sil_wrw_mem()	sil_wrw_iop()
	読み出し	sil_rew_mem()	sil_rew_iop()
ハーフワード	書き込み	sil_wrh_mem()	sil_wrh_iop()
	読み出し	sil_reh_mem()	sil_reh_iop()
バイト	書き込み	sil_wrb_mem()	sil_wrb_iop()
	読み出し	sil_reb_mem()	sil_reb_iop()

表 2: ビッグエンディアン用入出力 API

データサイズ	入出力方向	アドレス空間	
		メモリ空間	IO 空間
ワード	書き込み	sil_wrw_bem()	sil_wrw_bep()
	読み出し	sil_rew_bem()	sil_rew_bep()
ハーフワード	書き込み	sil_wrh_bem()	sil_wrh_bep()
	読み出し	sil_reh_bem()	sil_reh_bep()

表 3: リトルエンディアン用入出力 API

データサイズ	入出力方向	アドレス空間	
		メモリ空間	IO 空間
ワード	書き込み	sil_wrw_lem()	sil_wrw_lep()
	読み出し	sil_rew_lem()	sil_rew_lep()
ハーフワード	書き込み	sil_wrh_lem()	sil_wrh_lep()
	読み出し	sil_reh_lem()	sil_reh_lep()

微小時間待ち API

デバイスまたはターゲットハードウェアによっては微小のアクセスインターバルが必要であったり, 微小のタイムアウト用のタイミング生成等が必要になる場合がある。タイミング生成はターゲットハードウェアに依存するので SIL で実装することで機能を抽象化できる。

割り込みロック状態の制御 API

PDIC を実装する場合に必要となる排他制御を実現するための機能である。

ISR の管理機能

割り込みに依存した IRC・プロセッサの処理を SIL で行うことで ISR をポータブルに記述することができる。このために SIL で以下の 2 つの機能を実装し抽象化する。

(1) ISR の登録・削除

(2) ISR の呼び出し処理

2.4 PDIC(Primitive DIC)

PDIC はデバイスを扱う最低限のインターフェースモジュールを提供する。PDIC 自体はデバイス以外のターゲットハードウェアやカーネルには依存しない。

PDIC を構成する要素には以下の 4 つがある。

- アクセスインターフェース
- デバイスサービスルーチン (DSR)
- 割り込みサービスルーチン (ISR)
- 上位層へのコールバックルーチン (CBR) インターフェース

ここでは,PDIC の移植性に関わるアクセスインターフェースについて触れる。

2.4.1 アクセスインターフェース

同じデバイスであってもターゲットハードウェアによって結線（実装）方法は異なる。アクセスインターフェースはデバイスのボード上の結線（実装）方法を隠蔽し,PDIC の移植性を向上させる役割がある。

具体的にはアクセスインターフェースでは以下の事柄を抽象化する。

- デバイスレジスタまたは共有メモリのマッピング空間 (IO 空間またはメモリ空間)
- デバイスレジスタまたは共有メモリのマッピングアドレス
- デバイスとプロセッサのエンディアンの違い
- プロセッサからのデバイスまたは共有メモリへのアクセスタイミング制御
- デバイス特有の制御端子へのアクセス方法

デバイスレジスタまたは共有メモリへのアクセスおよびアクセスタイミング制御には下位層の SIL の API を用いる。このことにより DIC の移植者はプロセッサの詳細を意識することなくデバイスの結線方法に基づいてアクセスインターフェースを実装できる。

3 エンディアン問題

PDIC の移植性を阻害する要因の一つとしてデバイス－プロセッサ間のエンディアンの違いが挙げられる。デバイス－プロセッサ間のエンディアン問題は, 解決策は単純であるが, 実際に問題が起こった場合どのように解決したらよいかの検討が複雑になり, よくアドホックに解決されてしまうことがある。この章ではエンディアン問題を分類し, さらにデバイスドライバレベルで起こるエンディアン問題の DIC アーキテクチャでの解法を通して PDIC の移植性を示す。

[2] 引数は省略している

3.1 エンディアン問題とは

元来エンディアンとはマルチバイトデータの MSB (Most Significant Byte),LSB (Least Significant Byte) をバイトアドレスの下位側, 上位側のどちら側に置くかである。MSB を下位側におくものをビッグエンディアン, 上位側におくものをリトルエンディアンという。エンディアン問題とは通常このマルチバイトデータのメモリ上での配置が異なるため起こる問題である。

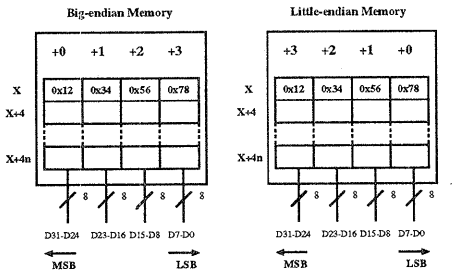

図 4: データバスにおけるエンディアン

しかし, これはソフトウェア側から見たエンディアンの一面であり, 実際には図 4に示すようにハードウェアの特性に依存しているエンディアンがある。図 4では 4 バイトデータ 0x12345678 を X 番地に格納したメモリ及びデータバスのイメージである。データバスにおけるエンディアンは通常バスの MSB 側を下位番地とするか (ビッグエンディアン), 上位番地とするか (リトルエンディアン) が異なるものである。つまりバス上でのバイトの並びはエンディアンに関係なく同じであり, 異なるのはバス上の各 8 ビットに対する番地付けである。またデータの転送サイズによってバスの使用方法が異なる。例えばハーフワード (16 ビット) のデータをバイトアドレスの X 番地に出力する場合, ビッグエンディアンでは D16-D31 (バスの上位 16 ビット) が使われ,X 番地に MSB が配置され,X+1 番地に LSB が配置される。それに対し, リトルエンディアンでは D0-D15(バスの下位 16 ビット) が使われ,X+1 番地に MSB が,X 番地に LSB が配置される。よって, 物理的に使用バスが異なり且つ論理的なアドレスも上下反転する。さらにバイトデータの場合,X 番地に出力するとビッグエンディアンでは D24-D31 が使用されるのに対し, リトルエンディアンでは D0-D7 が使用される。つまりバイトデータにもエンディアン問題が存在してしまう。

3.2 エンディアン問題の階層性

一口にエンディアン問題と言っても複数の要因を持つ場合がある。例えば通信システムの場合はアプリケーションレベルのプロトコルのエンディアン問題, ミドルウェアレベルのプロトコルのエンディアン問題, ハードウェアのプロセッサ－デバイス間, プロセッサ－共有メモリ－デバイス間の問題とあり, これらの問題を明確に分類しておく必要がある。

このようなシステムを開発する上では各レイヤー毎に問題を解決することがソフトウェアのモジュール性を高めることにつながる。つまり, アプリケーションで発生するエンディアン問題はアプリケーションで解決し, ミドルウェア間の問題はミドルウェアで, ハードウェア間の問題はハードウェアもしくはハードウェアでできない場合は DIC で解決することが望ましい。アプリケーションやミドルウェア間のエンディアン問題は良く知られているように, マルチバイトデータの MSB をアドレスの下位番地に配置する (ビッグエンディアン) か上位番地に配置する (リトルエンディアン) かだけの問題である。しかし, ハードウェアにおけるエンディアン問題ではプロセッサのエンディアンやデバイスのエンディアンさらにプロセッサ－デバイス－共有メモリ間の接続方法によって解決の仕方が異なる。

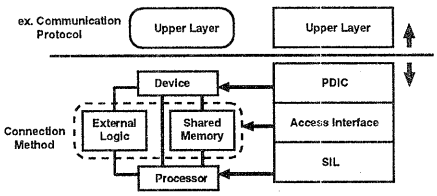

図 5: エンディアン問題の解決層

図 5に各エンディアン問題と DIC アーキテクチャとの対応関係を示す。PDIC より上位層の問題は上位層で解決し, デバイス自身のエンディアンは PDIC で対応し, プロセッサ自身のエンディアンは SIL で対応する。デバイスとプロセッサの結線によるエンディアンの違いはアクセスインターフェースで吸収する。

つまり,PDIC の実装者は対象デバイスのエンディアンのみを考慮して実装し, ターゲットハードウェアのプロセッサや結線方法を考慮しなくてもよい。ターゲットハードウェアの開発者が SIL を実装し, デバイスの移植者がアクセスインターフェースを実装すればよい。

3.3 デバイスドライバレベルのエンディアン問題

デバイスドライバレベルのエンディアン問題はデバイスとプロセッサのエンディアンの違いにより発生するエンディアン問題である。具体的にはデータバスの結線方法に応じて表 4,5に示すような処理が必要になる。ここでストレート結線というのは通常通り MSB,LSB 同士を接続する結線であり, バイトスワップ結線というのは 8bit 毎に結線を反転させる結線である。

つまり, デバイスとプロセッサの結線方法によってエンディアン変換のための処理が異なる。通常デバイスレジスタのみを使用するデバイスに対してはストレート結線し, 共有メモリを使用するデバイスに対してはバイトスワップ結線するべきであると考えられる。これは, デバイスレジスタはアクセスサイズが限られており, バス幅より大きなレジスタは分割して扱うためである。これに対して共有メモリではアクセスサイズはデバイスやデバイスドライバ, 上位ソフトウェアによって異なり, ストレート結線ではエンディアン変換処理が一意に決まらないためである。よってエンディアン変換の処理は, デバイスレジスタに関してはアドレス変換のみでよく[3], 共有メモリを扱うデバイスに関してのみバイトスワップ処理が必要になると考えられる。

[3]アドレスに対して予め変換処理を施せば動的な処理は不要

表 4: ストレート結線時のエンディアン変換処理

	データサイズ		
バスサイズ	ワード	ハーフワード	バイト
32	*	M2	M3
16	HS	*	M1
8	BS	BS	*

M1　A0 マンギング (ADDRESS XOR 0x01)
M2　A1 マンギング (ADDRESS XOR 0x02)
M3　A0,A1 マンギング (ADDRESS XOR 0x03)
HS　ハーフワードスワップ
BS　バイトスワップ
*　エンディアン変換処理不要

表 5: バイトスワップ結線時のエンディアン変換処理

	データサイズ		
バスサイズ	ワード	ハーフワード	バイト
32	BS	BS	*
16	BS	BS	*
8	BS	BS	*

BS　バイトスワップ
*　エンディアン変換処理不要

3.4 DIC アーキテクチャにおける解法

DIC アーキテクチャでは, デバイスドライバレベルのエンディアン問題を解決できるように階層構造を形成している。前述のデバイス—プロセッサ間で生じるエンディアン問題はアクセスインターフェースで解決する。

PDIC では共有メモリやデバイスレジスタに対するアクセスをアクセスインターフェースとして分離して実装する。またデバイスレジスタの場合マッピングアドレスもここで抽象化する。

アクセスインターフェースではターゲットハードウェアでのデバイスの結線に適合するように実装する。以下アクセスインターフェースにおけるデバイスの結線方法の違いによるエンディアン問題の解決方法を示す。

ストレート結線時

バス幅と同じか小さいデータのアクセスに関してはデバイス—プロセッサ間におけるマルチバイトデータのバイトオーダは変わらない。異なるのはアドレスだけであり, アドレス変換を行うだけでよい。データアクセスには SIL のナチュラルエンディアン用 API を用いる。

バイトスワップ結線時

SIL のビッグ／リトルエンディアン用入出力 API を用いることで各データサイズ毎に動的なバイトスワップ処理を施しデータ入出力を行う。

SIL ではプロセッサ自身を抽象化する。例えば異なるアドレス空間へのアクセス命令や, バイトスワップ処理用インストラクション等のプロセッサ特有のインストラクションを抽象化する。

3.5 DIC アーキテクチャにおける解法例

ここで,DIC アーキテクチャにおけるエンディアン問題の解法例を示す。デバイスには LAN コントローラの Intel i82596, プロセッサにはビッグエンディアンの Motorola 68040 を例にとり説明する。i82596 に関してはリトルエンディアンモードで動作させる場合を考える[4]。また, これらは共に 32 ビットデータバスを持っているので 32 ビットバスで接続される。

[4]i82596 はバイエンディアンのデバイスであるがここでは具体例を示すためにリトルエンディアンで考える。

i82596 は共有メモリを介してプロセッサとのデータのやり取りを行う。例えば送受信するパケットデータや送受信バッファのアドレス, デバイスのコンフィグレーションワード, デバイスコマンド, ステータスワードなどバイト, ハーフワード, ワードサイズのデータを共有メモリ上に置く。

この場合 PDIC の API の一つとして以下のようなパケット送信用 API が考えられる。これは引数 buf 番地から先頭の len バイトのデータを送信するという API である。

```
i82596_snd_pkt(void *buf, int len)
```

上記の PDIC API を実装するためにアクセスインターフェースとして, 以下のバッファ先頭番地を書き込むインターフェース, パケットデータをブロックコピーするインターフェースを用意する必要がある。[5]

```
i82596_write_ptr(void **mem, void *ptr)
i82596_copy_block(char *dst, char *src, int len)
```

3.5.1 ストレート結線時

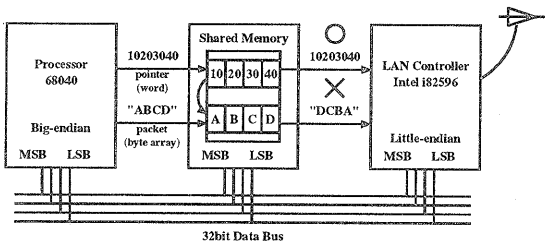

図 6: ストレート結線時のエンディアン問題

ストレート結線時の結線図を図 6に示す。ストレート結線時はデバイスとプロセッサは 32 ビットバスで接続されているため, バッファアドレス等のワードデータに関してはエンディアン変換の処理無しで入出力が可能である。またパケットデータ等のバイトデータに関してはアドレスマンギング処理[6]を行う必要がある。つまりデータのサイズに関わらずデータ入出力にはナチュラルエンディアン用入出力 API を使用し, ワードデータ以外のデータに関してはアドレスマンギングの処理を行えばよい。ストレート結線時のアクセスインターフェースを図 7に示す。[7]

3.5.2 バイトスワップ結線時

バイトスワップ結線時の結線図を図 8に示す。バイトスワップ結線時はバイトデータとアドレスの対応はとれるため問題なく入出力できる。しかしマルチバイトデータは結線で反転しているためバイトスワップの処理が必要になる。つまりこの場合 SIL のリトルエンディアン用入出力 API を用いればよい。バイトスワップ結線時のアクセスインターフェースを図 9に示す。

この他にもブロックコピーの最適化を行うためにバスサイズの入出力 API を使ったアクセスインターフェースの実装も考えられる。

[5]この他にも共有メモリに対するアクセスをアクセスインターフェースとして分離する必要がある

[6]この場合 ADDRESS XOR 0x03

[7]sil_wrx_mem() の第一引数は書き込み先番地, 第二引数は書き込みデータ。

```
void i82596_write_prt(void **mem, void *ptr)
{
    sil_wrw_mem(mem, ptr)
}

#define MANGING_CONST 0x03
void i82596_copy_block(char *dst, char *src, int len)
{
    while (len--){
        sil_wrb_mem(dst++ ^ MANGING_CONST, *src++);
    }
}
```

図 7: ストレート結線時のアクセスインターフェース

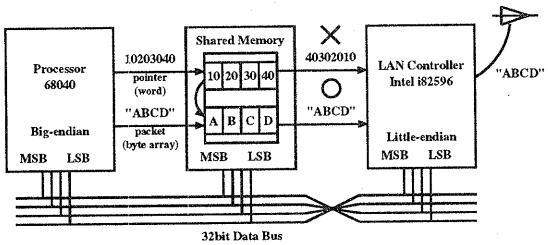

図 8: バイトスワップ結線時のエンディアン問題

以上のように結線の違いによるエンディアン変換処理をアクセスインターフェースで吸収する。あるデバイスがボード上でどう実装されているかは多様化している組み込みシステムではシステム開発者によって異なる可能性がある。このようにデバイスドライバの結線依存部を局所化し, 分離することでモジュール性を高め, さらに移植性向上につながると考えられる。

4 移植性評価

この節では PDIC の移植性について, 実際に LAN コントローラでの実装を例にとり検証する。評価環境を表 6 に示す。対象としたデバイスは前節の例で示した Intel 製の i82596 である。ターゲット環境は電産製の CPU ボード DVE-68K/40 と DVE-SH7700 である。これらは両方とも LAN コントローラとして i82596 を持ちプロセッサがそれぞれ Motorola MC68LC040, 日立 SH3 と異なる。プロセッサが異なるため割り込み関連の処理も異なる。またそれぞれの環境でカーネルがある場合と無い場合の検証も行う。残念ながらプロセッサ及び LAN コントローラのエンディアンはすべてビッグエンディアンであったためエンディアン問題解法の実装検証はここではできなかった。

```
void i82596_write_prt(void **mem, void *ptr)
{
    sil_wrw_lem(mem, ptr)
}

void i82596_copy_block(char *dst, char *src, int len)
{
    while (len--){
        sil_wrb_mem(dst++, *src++);
    }
}
```

図 9: バイトスワップ結線時のアクセスインターフェース

表 6: 評価環境

	環境 1	環境 2
ボード名称	電産製 DVE-68K/40	電産製 DVE-SH7700
プロセッサ	Motorola 製 MC68040	日立製 SH7708(SH3)
IRC	電産製ゲートアレイ DGA-001	電産製ゲートアレイ DGA-001
対象デバイス	LAN コントローラ Intel 製 i82596	
使用カーネル	μITRON4.0 仕様カーネル TOPPERS/JSP(1.0 β) 及び無しの環境	
開発環境	コンパイラ : gcc version 2.95.2 OS : Linux 2.2.18	

この環境で PDIC, アクセスインターフェース,SIL を実装し PDIC を書き換えること無く別の環境へ移植が可能であることを確認する。アクセスインターフェースは移植時に書き換えが必要なモジュールであるので, この部分のコード量を測り, 移植に必要な労力を検証する。

SIL はターゲット環境のカーネル・プロセッサおよび IRC を抽象化するものであるので, ターゲットのシステム構築時に必要となる。つまり PDIC の移植時に書き換えるものではないため移植性に直接影響することは無い。

4.1 PDIC の実装

PDIC には LAN コントローラとして一般的に必要な機能を実装する。以下にデバイスサービスルーチンを示す。

- デバイスの初期化
  `ER i82596_ini_dev(VB macadr[6])`
- デバイスの終了処理
  `ER i82596_ter_dev()`
- パケットの送信
  `ER i82596_sup_rbf(VP buf, int size)`
- 受信パケット用のバッファ供給
  `ER i82596_snd_pkt(VP buf, int tcnt)`

コールバックルーチンとして以下のルーチンを用意する。

- 送信完了通知
  `void i82596_csnd_pkt(UH stat, VP buf, UH count)`
- 受信完了通知
  `void i82596_ercv_pkt(VP buf, UH count)`
- 受信エラー通知
  `void i82596_ercv_err(UH stat, VP buf, UH count)`
- 受信パケット用リソース無し通知
  `void i82596_enrs_rbf()`

4.2 アクセスインターフェースの実装

i82596 は基本的に共有メモリを介してプロセッサとの通信を行い, デバイスレジスタは存在しない。またこのデバイスではエンディアンはビッグエンディアン・リトルエンディアン両タイプに対応しているが, ビッグエンディアンに関しては完全には対応していない特殊なデバイスである。[8]つまり, プロセッサがビックエンディアンの場合通常のビッグエンディアンデバイスとは異なる処理が

[8]デバイスのステップ番号によって対応度が異なる

必要になる。具体的にはSTEP-Aデバイスでは, バイト, ハーフワードデータに関しては完全にビッグエンディアンで動作するが, ワードデータに関してはハーフワード毎に反転させなければならない。このためリンクポインタ, カウンタ等のワードデータを共有メモリと入出力する場合はハーフワードスワップの処理が必要になる。

また, このデバイスは通常のデバイスとは異なる入出力方法を持っている。これはデバイスレジスタを持たないために必要となる機能であり, 共有メモリ上のリストデータ構造のルートとなるアドレスをデバイスに知らせたり, デバイスのリセット, セルフテスト等の制御を行う機能である。具体的には, デバイスへデータを出力するポートアクセスと, デバイスへの事象通知を行うチャネルアテンションという機能である。この CPU ボード上では独自の拡張回路を用いてこのアクセスを実現している。

これらのプロセッサ—デバイス間のエンディアンの違い及びデバイス独自の機能のボード上での実装方法はアクセスインターフェースで吸収することで,PDIC の移植性を向上させる。

以下にアクセスインターフェースとして用意したインターフェース[9]を示す。

- リンクポインタの書き込み／読み出し (ビッグ／リトルエンディアンモード, 対象データ毎に用意)
  `void i82596_wrp_xxx(VP mem, VW val)`
  `VW i82596_rep_xxx(VP mem)`
- アドレスの定義。(ポートコマンド発行アドレス, チャネルアテンション発行アドレス)
- ポートコマンド発行インターフェース
  `void i82596_wrt_prt(int cmd, VW ptr)`
- チャネルアテンション発行インターフェース
  `void i82596_ast_ca()`

4.3 評価

表 7に PDIC, アクセスインターフェースの実装コード行数, 書き換え行数を示す。コードの行数には空白行やコメント行も含まれるため厳密な比較はできないがおおよその目安にはなると考えられる。

アクセスインターフェースの具体的な変更・追加個所は以下の通りである。

- アドレス定義の変更
- ポート書き込み時のアドレス変換
- 共有メモリ上のポインタ書き込み時のアドレス変換[10]

表 7: PDIC・アクセスインターフェースのコード量

モジュール	コード行数		書き換え行数
	68040	SH3	
PDIC	998		0
アクセスインターフェース	109	112	13

デバイスの結線依存部であるアクセスインターフェースのコード量は PDIC に対して 1 割程度と十分に小さくでることが確認できる。実装コードは長いものでも数行程度で済んだため, プリプロセッサマクロとコンパイラのインライン展開機能を用いて記述することができた。またデバイスドライバの動作自体を記述している個所は無く, 単純なデバイスアクセスのみの記述で済んだ。これらの実装はデバイスの詳細な動作を理解していなくてもできる程度のものである。

これらの結果より, アクセスインターフェースの切り分けを十分に考慮して行うことで PDIC の移植作業はそれほど労を要するものでは無いと考えられる。

[9]デバイスのエンディアンはプロセッサのエンディアンに合わせて使用することを前提として, チューニングのために送信データのブロックコピーは省いた

[10]DVE-SH7700 環境ではプロセッサから見えるアドレスとデバイスから見えるアドレスが異なるためこの変換を行った

5 まとめ

ITRON デバイスドライバ設計ガイドラインの提案および, この DIC アーキテクチャを用いてのデバイスドライバレベルでのエンディアン問題の解法を例にとり移植性を示した。また LAN コントローラ用の DIC の実装例を通して PDIC の移植性の評価を行った。

このガイドラインでは,PDIC をターゲットハードウェアに依存せずに記述し, デバイスドライバの移植性および再利用性を向上させることを目的としている。移植性を向上させるために PDIC より下位に, デバイスのボード上での結線を抽象化するアクセスインターフェース, プロセッサ・IRC を抽象化する SIL の 2 つの階層を設けることでハードウェアへの依存性を分割し,PDIC の移植性向上を促した。実際, 実装例におけるアクセスインターフェースの実装も PDIC のコード量に対して 1 割程度であり, また全てインライン展開できる程度のものであったので, デバイスのボード上での実装依存部を局所化し, 移植性を十分に軽減できているといえる。

またエンディアン問題についても同様にこの階層構造を用いて, デバイスのエンディアン, プロセッサのエンディアン, プロセッサ—デバイス間の結線方法を各階層で吸収することによりエンディアン問題の階層分けを行い, 問題の迅速な解決を促すと考えられる。

今後の課題としては,DIC アーキテクチャにおける階層化構造を用いたことによるオーバヘッドの検証, アクセスインターフェースにおいて実際にエンディアン問題を解決できるかどうかの具体的な実装検証が必要である。

参考文献

[1] μITRON 仕様研究会デバイスドライバ設計ガイドライン WG, “デバイスドライバ設計ガイドライン WG 中間報告書”, 1999.

[2] 坂村 健 監修, 高田 広章編, “μITRON4.0 仕様”, トロン協会, http://www.itron.gr.jp/, 1999.

[3] ALESSANDRO BUBINI 著, 山崎 康宏 山崎 邦子共訳,“Linux デバイスドライバ”, オライリー・ジャパン, 1998

[4] Project UDI, “UDI Core Specification”, http://www.project-UDI.org/specs.html, 1999

[5] “PowerPC Microprocessor Family: The Programming Environments For 32-Bit Microprocessors(MPCFPE32B/AD 1/97 REV. 1)” - Chapter 3 Operand Conventions , Motorola

[6] “MPC860 PowerQUICC ユーザーズマニュアル (MPC860UMJ/AD 05/2000 Rev. 1)”, Motorola

[7] “SH7708 シリーズハードウェアマニュアル (ADJ-602-119G(O))” - 10. バスステートコントローラ (BSC), 日立製作所