デバイスドライバとデバイスの一体設計手法へのSpecCの適用性評価

本田晋也[†] 高田広章[†]

我々は，組込みシステムのデバイスドライバ開発を効率することを目的に，デバイスドライバとデバイスの一体設計手法について研究を進めている．本論文では，システムレベル記述のための言語として提案されているSpecCが，本設計手法におけるデバイスドライバとデバイスの一体記述のための言語として用いることが可能であるか，その適用性を評価した．具体的には，SIOシステムを例として，SpecCによる記述を行い，デバイスドライバとデバイスの一体記述が可能であることを確認した．また，SIOシステムのSpecC記述からデバイスドライバ，デバイス，およびその間のインタフェースへの変換を手作業で行い，提示した記述指針に従えばSpecC記述から実装記述への機械的生成が可能であることを確認した．これらより，SpecCは本手法の一体記述言語として適用可能であることを示した．

Evaluation of Applying SpecC to the Integrated Design Method of a Device Driver and a Device

Shinya Honda[†] and Hiroaki Takada[†]

We are investigating an integrated design method of a device driver and a device for the efficient development of device drivers for embedded systems. This paper evaluates if SpecC, which is proposed as a system-level description language, is applicable to integrated description of a device driver and a device. We use as example a SIO system to confirm that integrated description of a device driver and a device by SpecC is possible. Also,we manually convert the SpecC description to the device driver, the device, and the interface between them, and confirm that the conversion can be automated. As a result, we show that SpecC could apply as an integrated design language of the integrated design method.

1. はじめに

近年，組込みシステムの大規模化・複雑化にともない，開発期間や開発コストの増大に加え，設計品質や信頼性の低下が大きな問題になっている．組込みシステムは，組み込む対象の機器に専用化して設計されるため，システムごとにハードウェア構成や周辺デバイスが異なる．そのため，デバイスを直接制御するソフトウェアであるデバイスドライバはシステムごとに開発する必要があり，組込みソフトウェア開発においてはデバイスドライバ開発の比率が大きい[1)]．

ところが現状では，デバイスドライバ開発に大きな開発工数がかかっているという問題がある．その主な原因の1つとして，デバイス設計者とデバイスドライバ設計者の間の意志疎通の問題があげられる．デバイスのマニュアルは，多くの場合，デバイスレジスタの説明を中心に記述されており，デバイスドライバ設計に必要なソフトウェアからの制御方法の観点が欠けている．そのため，マニュアルのみからデバイスドライバ設計に必要な情報を得るのは容易ではない．さらに，デバイスドライバ側の事情を考慮せずにデバイスが設計されているケースが多いことも，問題を難しくしている[2)]．

また別の原因として，デバイスドライバはデバイスの定めるタイミングに従って動作しなければならないために，検証やデバッグが難しいことがあげられる．一般に，タイミングに依存するソフトウェアの検証は，あらゆるタイミングでの動作を確かめなければならず，検証工数が大きくなる．また，問題があった場合にもその再現性が低く，バグの発見も困難である．

この問題を解決するために我々は，組込みシステムにおけるデバイスドライバ開発の効率化を目的とした，デバイスドライバ（ソフトウェア）とデバイス（ハー

† 豊橋技術科学大学情報工学系
Department of Information and Computer Sciences, Toyohashi University of Technology

ドウェア）の一体設計手法について研究を行っている．具体的には，システムLSI設計の分野で研究の進んでいるシステムレベル設計やコデザイン技術を適用して，デバイスドライバとデバイスを1つの言語で一体に記述し，そこから両者を生成するアプローチをとる．これにより，デバイス設計者とデバイスドライバ設計者の間の意志疎通の問題が解決され，デバイスドライバ開発の効率化が期待される．また，コデザイン技術を適用することで，デバイスドライバとデバイスの切り分けを柔軟に変更できるという利点も期待できる．

本研究は，デバイスドライバとデバイスを一体に記述するための言語としてSpecC[3]をとりあげ，その適用性の評価を目的とする．ここでの適用性とは，デバイスドライバとデバイスの一体記述が可能であることに加えて，その記述から実装記述を機械的に生成できることをいう．SpecCはC言語をベースとして，ハードウェア記述のための概念と構文が追加されおり，ソフトウェアとハードウェアが明確に区別されていないシステムレベルの記述から，それらが区別された実装設計の手前までをカバーする言語として提案されている．

しかしながら，SpecCは言語仕様が作成されてから日が浅く，追加された概念と構文のセマンティクスに関しては，ハードウェア記述ないしハードウェア生成の観点からの議論が中心で，ソフトウェア生成，特にリアルタイムカーネルを利用するソフトウェアの生成方法は考えられていない．大規模化するシステムLSI上のソフトウェア構築には，リアルタイムカーネルの利用が必須であると考えられ，提案する設計手法で生成するデバイスドライバはリアルタイムカーネルの使用を前提とする．そのため，提案する設計手法にSpecCが適用可能であるか評価する必要がある．

SpecCの適用性を評価するために，本論文では，シリアルライン上にTCP/IPパケットを送るためのプロトコルであるPPP（Point-to-Point Protocol）の処理機能を含むシリアルI/Oシステム（以下，SIOシステム）を例に用いる．SIOシステムを例としたのは，システムLSIの重要な適用分野である通信システムとして，簡単ではあるが典型的な構造を持っており，最初の評価対象として適当と考えたためである．

具体的な評価方法として，まず，SIOシステムを構成するデバイスドライバとデバイスをSpecCによって一体に記述する．これによりSpecCによりSIOシステムが記述可能か評価する．あわせて，SpecC記述の指針についても提示する．次にその記述を，デバイスドライバ，デバイス，およびその間のインタフェースへ，手作業によって変換する．これにより生成の機械化の可能性について評価する．特に，SpecCの提供する並行性や同期・通信を記述するための構文から，リアルタイムカーネルの機能を効率的に用いたデバイスドライバやインタフェースに機械的に生成可能であるかに着目する．

本論文では，まず2章で，デバイスドライバとデバイスの一体設計手法の概要を述べる．3章では，SpecCの概要について，SIOシステムの記述に関連する構文を中心に解説する．次の4章では，適用性評価の評価項目と評価手順について述べる．次の5章では，SpecCによるSIOシステムの記述の概要と，SpecC記述の指針について述べる．続く6章では，デバイスドライバ，デバイス，およびその間のインタフェースへの変換方法について述べ，機械的な変換が可能であるかを検討する．最後に7章では，デバイスドライバとデバイスの切り分けを変更できるか，いい換えると，同じ記述からソフトウェアとハードウェアのいずれにも変換できるかについて，評価を試みる．

2. デバイスドライバとデバイスの一体設計手法

組込みシステムのデバイスドライバ開発を容易化するための試みとして，デバイスドライバの設計ガイドラインの策定[4]や，デバイスドライバを自動生成する方向の研究[5]がなされている．しかしながら，設計ガイドラインの策定は，問題を部分的にしか解決できず，自動生成には，デバイスドライバとデバイスの仕様の記述方法に問題がある．また，これらの試みはすでに完成しているデバイスのためのデバイスドライバを対象としているが，組込みシステムにおいては，デバイスもシステムごとに設計される場合が多く，そのような場合でも効果が限定される．

我々は，デバイスドライバ開発を効率化するためのより効果的なアプローチとして，デバイスドライバとデバイスを一体に記述し，その記述からデバイスドライバとデバイスを生成する手法について研究を行っている．デバイスドライバとデバイスを一体に記述することにより，デバイス設計者とデバイスドライバ設計者の間の意志疎通の問題が解決され，また，デバイスドライバ側の事情を考慮したデバイス設計を促すという効果もある．さらに，一体記述が実行可能であれば，検証を早期に行うという目的にも有用である．

本手法におけるデバイスドライバとデバイスの設計

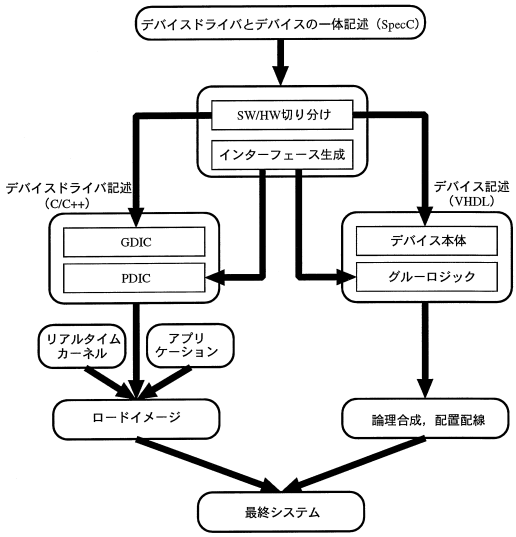

図 1 デバイスドライバとデバイスの一体設計の流れ
Fig. 1 Integrated design flow of device driver and device.

の流れを**図 1** に示す☆．まず SpecC により，デバイスドライバとデバイスを一体に記述する．次にこの一体記述を，ソフトウェアで実装する部分と，ハードウェアで実装する部分に切り分けるとともに，その間のインタフェースを生成する．現時点では，ソフトウェアとハードウェアの切り分け方は設計者が指定するものとしているが，コデザイン技術の発展によりその自動化も可能になると考えられる．

ソフトウェアで実装する部分は C/C++記述に変換し，生成されたインタフェースのソフトウェア側（PDIC）とともに，デバイスドライバ記述となる．C/C++ 記述への変換の際に，SpecC の並行性や同期・通信の記述は，リアルタイムカーネルのタスクやシステムコールへと変換される．また，ハードウェアで実装する部分は，論理合成が可能なレベルの VHDL 記述に変換し，インタフェースのハードウェア側（グルーロジック）とともに，デバイス記述となる．SpecC 記述から VHDL 記述への変換手法については，他の研究成果[6]を利用することとし，本研究では主たる研究対象とはしない．

本章の初めに，デバイスドライバの自動生成には仕様の記述方法に問題があると述べたが，デバイスドライバとデバイスの仕様を SpecC で記述するものと考えれば，本手法はデバイスドライバの自動生成手法の一種ととらえることもできる．

3. SpecC

SpecC は，ANSI C をベースとして，並行性，同期・通信，タイミングなどの概念とそのための構文が追加された言語である．本章では，以降の章と関連する SpecC の構文について説明する．その他の構文については，文献 7) を参照されたい．

SpecC では，システムの機能要素をビヘイビアと呼ばれるオブジェクトとして記述する．ビヘイビアは複数のメソッドを持ち，ビヘイビアが実行されると main メソッドが実行される．システムはビヘイビアのインスタンス（以下，単にビヘイビアと呼ぶ）の階層構造としてモデル化される．ビヘイビア間の通信は，ポートもしくはチャネルを介して行う．

並行性の記述のためには，**par** 文が追加されている．**par** 文中に記述されたビヘイビアは，並行に実行される．

ビヘイビア間の同期は，**event** 型の変数を介して **wait** 文と **notify** 文により行う．**wait** 文は引数で指定されたイベントのいずれか 1 つを受け取るまでビヘイビアの動作を中断する．**notify** 文は引数で指定されたイベントを送り，それを待っているすべてのビヘイビアの動作を再開させる．

ビヘイビア間の複雑な同期・通信は，チャネルと呼ばれるオブジェクトとして記述する．チャネルは，内部変数とメソッドを持ち，ビヘイビア間の同期・通信機構をカプセル化する．ビヘイビアはチャネルのメソッドを呼び出すことにより同期・通信を行う．同期・通信のための排他制御を実現するために，チャネルのメソッドは，**wait** 文を除いてアトミックに実行される．

例外処理は，**try-trap-interrupt** 構文を用いて記述する．**try-trap-interrupt** 構文の例を以下に示す．

```
try{A.main();}
  trap(e1){B.main();}
  interrupt(e2){C.main();}
```

この記述が実行されると，まず **try** 節に記述されているビヘイビア A が実行される．ビヘイビア A の実行中にイベント e1 を受け取ると，ビヘイビア A の実行は中断され，ビヘイビア B が実行される．ビヘイビア B の実行が終了すると **try-trap-interrupt** ブロックを脱出する．**interrupt** 節は，**trap** 節と異なり，ビヘイビア C の実行終了後，ビヘイビア A の実行を再

☆ 図中の PDIC（Primitive Device Interface Component）とは，デバイスを扱う最低限のインタフェースソフトウェアのことをいい，GDIC（General Device Interface Component）とは，リアルタイムカーネルに依存しデバイスを扱う上位のソフトウェアをいう[4]．

開する．

4. 適用性の評価項目と評価手順

4.1 評価項目

SpecCが，デバイスドライバとデバイスの一体設計手法のための一体記述言語に適用できるかを評価するための項目を以下に示す．

(1) SpecCによりデバイスドライバおよびデバイスを一体に記述できるか．

(2) SpecCの提供する並行性や同期・通信を記述するための構文から，リアルタイムカーネルの機能を効率的に用いたデバイスドライバに機械的に変換可能であるか．

(3) SpecC記述からソフトウェアとハードウェア間のインタフェースへの機械的な変換が可能であるか．

なお，本研究では 2 章で述べたように SpecC 記述からの変換に関しては，デバイスドライバとインタフェースのへの変換手法のみについて評価し，デバイスへの変換については扱わない．

4.2 評価手順

評価項目 (1) を評価するため，まず SIO システムを構成するデバイスドライバとデバイスを SpecC よって一体に記述する．記述する SIO システムの概要を**図 2** に示す．本システムは，基本的なシリアル送受信機能に加えて，TCP/IP パケットを送るためのプロトコルである PPP パケットの生成/解析機能を持つ．シリアル通信の通信方式は調歩同期方式とし，通信フォーマットはボーレート 19600 bps，データ 8 bit，ストップビット 1 bit，パリティなしとした．

次に評価項目 (2)，(3) を評価するため，SIO システムの SpecC 記述をデバイスドライバ，デバイスおよびその間のインタフェースへ手作業により変換する．なお，SpecC 記述からの変換後のソフトウェアが利用するリアルタイムカーネルは，μITRON4.0 仕様[8]準拠のリアルタイムカーネルとする．

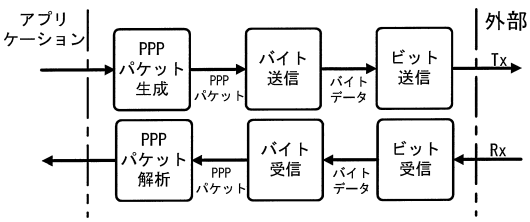

図 2 SIO システムの概要図
Fig. 2 Overview of SIO system.

5. SpecC による SIO システムの記述

本章では，4.1 節の評価項目 (1) を評価するため，SpecC による SIO システムの記述について述べる．

5.1 SpecC 記述

SpecC 記述による SIO システムの送信部のモデルを**図 3** に，受信部のモデルを**図 4** に示す．角の丸い四角形はビヘイビア，両端が半円の四角形はチャネルを表す．受信部・送信部のビヘイビアとそこから呼び出される関数，およびチャネルの記述量は，589 行（17.2 kbyte）である．以下，送信部，受信部に分けて詳細を説明する．

5.1.1 送信部

送信部は，PPP パケット生成ビヘイビア（tx_ppp），バイト送信ビヘイビア（tx_byte），ビット送信ビヘイビア（tx_bit），調歩同期用送信クロック生成ビヘイビア（tx_clk），ボーレートの 16 倍クロック生成ビヘイビア（gen_clk16）で構成され，各ビヘイビアは並行に動作する．アプリケーションと tx_ppp の間，tx_ppp と tx_byte の間，tx_byte と tx_bit の間は，それぞれチャネルにより接続されている．

tx_ppp，tx_byte，tx_bit は，実行が開始されると，それぞれ上位層からのデータを待つ．tx_ppp はアプリケーションから IP パケットを受け取ると，PPP パケットを生成して tx_byte に送る．tx_byte は PPP パケットを受け取ると，それを 1 バイトずつ tx_bit に送

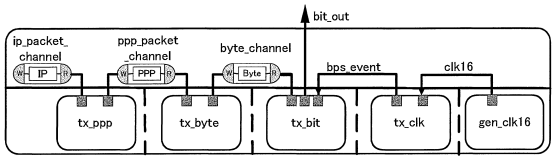

図 3 SpecC 記述の SIO システムの送信部
Fig. 3 Transmitter part of SIO system in SpecC.

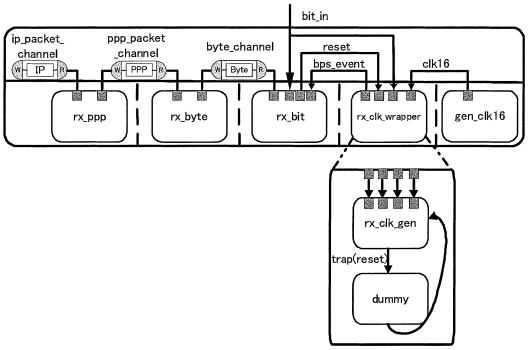

図 4 SpecC 記述の SIO システムの受信部
Fig. 4 Receiver part of SIO system in SpecC.

る．tx_bit はバイトデータを受け取ると，tx_clk からのイベント（bps_event）に同期して，ビットごとにシリアルライン（bit_out）に送信する．各ビヘイビアは上記の処理を無限に繰り返す．

5.1.2 受 信 部

SpecC による受信部の記述モデルを図 4 に示す．受信部も送信部と同様に，並行動作するビヘイビアに分けて記述した．送信部とは反対に，PPP 解析ビヘイビア（rx_ppp）とバイト受信ビヘイビア（rx_byte）は下位層からのデータを待つ．

スタートビット検出のためのシリアルラインの立ち下がりエッジの検出は，受信クロック生成ビヘイビア（rx_clk_gen）が行う．rx_clk_gen のコードを**図 5** に示す．rx_clk_gen は立ち下がりエッジを検出すると，ボーレートの 16 倍クロックで 8 クロック経過後（ボーレートの半周期），ビット受信ビヘイビア（rx_bit）にイベント（bps_event）を送る．その後，ボーレートに応じた周期で，rx_bit にイベントを送り続ける．

rx_bit は，rx_clk_gen から最初のイベントを受け取ると，シリアルライン上のデータがスタートビットであるか判定する．スタートビットであればそれ以降，rx_clk_gen からのイベントに同期して受信データを取り込む．スタートビットでなければ，スタートビットの再検出のために rx_clk_gen をリセットする．また rx_bit は，1 バイト分のデータを受信した後も rx_clk_gen をリセットする．

rx_clk_gen のリセットは，**try-trap** 構文，ラッパビヘイビアとダミービヘイビアを用いて実現した．ラッパビヘイビア（rx_clk_wrapper）は，**図 6** に示すように，**try-trap** 構文の **try** 節で rx_clk_gen を実行する．rx_clk_gen の実行中にイベント reset を受け取ると rx_clk_gen の実行を中断し，rx_clk_gen を再び先頭から実行する．**trap** 節には，何らかのビヘイビアを記述する必要があるため，何も行わないダミービヘイビアを用いている．

5.1.3 チャネル

送信部と受信部の記述には，それぞれ 3 種類のチャネルを用いている．tx_byte と tx_bit の間および rx_byte と rx_bit の間では，1 バイト単位でデータを受け渡すため，データをコピーする値渡しを行っている．それに対して，IP パケットや PPP パケットを受け渡すアプリケーションと tx_ppp の間や tx_ppp と tx_byte の間などは，データのコピーを避けるために，ポインタ渡しによってデータを受け渡している．

チャネル記述の例として，バイトデータを受け渡すバイトチャネルのコードを**図 7** に示す．バイトチャネル以

```
behavior rx_clk_gen(in bit[0:0] bit_in,
                    in event clk16,
                    out event bit_event){
void main(void){
    int i;
    bit[0:0] pre_bit;
    while(1){
        pre_bit = 0B
        while(!(bit_in == 0B & pre_bit == 1B)){
            pre_bit = bit_in;
            wait(clk16);
        }
        for(i = 0; i < 8; i++)
            wait(clk16);
        notify(bit_event);
        while(1){
            for(i = 0; i < 16; i++)
                wait(clk16);
            notify(bit_event);
        }
    }
}};
```

図 5 受信クロック生成ビヘイビア（rx_clk_gen）
Fig. 5 Receive clock generation behavior (rx_clk_gen).

```
behavior rx_clk_wrapper(in bit[0:0] x,
                        in event clk16,
                        out event bit_event,
                        in event reset){
rx_clk_gen be_rx_clk_gen(x,clk16,bit_event);
dummy trap_dummy();
void main(void){
    while(1){
        try{ be_rx_clk_gen.main();}
            trap (reset){trap_dummy.main();}
    }
}};
```

図 6 ラッパビヘイビア（rx_clk_gen_wrapper）
Fig. 6 Wrapper behavior (rx_clk_gen_wrapper).

```
channel byte_channel(void)
    implements byte_interface{
    unsigned char data_buffer; /* バッファ */
    bool data_valid;           /* バッファ・ステータス */
    event sync_write;          /* 書き込み用イベント */
    event sync_read;           /* 読み込み用イベント */
    void write(unsigned char data){
        while(data_valid)
            wait(sync_read);
        data_buffer = data;
        data_valid = true;
        notify(sync_write);
        }
    unsigned char read(void){
        while(!data_valid)
            wait(sync_write);
        data_valid = false;
        notify(sync_read);
        return(data_buffer);
    }
};
```

図 7 バイトチャネル（byte_channel）
Fig. 7 Byte channel (byte_channel).

外の PPP パケット用チャネル（ppp_packet_channel）と IP パケット用チャネル（ip_packet_channel）も，受け渡しするデータの型が異なることを除いては，チャネルの記述はまったく同じである．

5.2 シミュレーション

記述した SIO システムの動作確認のため，送信部と受信部のシリアルラインどうしを接続し，tx_ppp に送信した IP パケットを rx_ppp から受信するテストベンチを用いてシミュレーションを行った．シミュレーション環境としては，CATS/Soliton 社の VisualSpec と，SpecC リファレンスコンパイラ（SCRC 1.1）を用いた．

シミュレーションの結果，記述した SIO システムが正しく動作することを確認した．

5.3 SpecC 記述の指針

5.3.1 ビヘイビアの切り分け

前述の SIO システムの SpecC 記述では，実行すべきタイミングが同じ処理が同じビヘイビアに含まれるように，ビヘイビアの切り分けを行った．いい換えると，実行タイミングごとにビヘイビアを切り分けており，図 3 と図 4 で右側にあるビヘイビアほど実行頻度が高い．これは，以下に述べる理由により，実行頻度の変化する点が，ソフトウェアとハードウェアの分割点として適切であると考えられるためである．

一般に，システムをソフトウェアとハードウェアに分割して実現する場合，実行性能が一定であれば，ソフトウェア側で実現する機能が多いほどソフトウェアのコストが高くなり，逆にハードウェア側で実現する機能が多いほどハードウェアのコストが高くなる．ある機能を実現するためのソフトウェアのコストは，必要なプロセッサの処理能力やメモリ容量で決まる．ソフトウェア構築にリアルタイムカーネルを用いる場合，必要なプロセッサの処理能力として，実現すべき機能自身の処理時間に加えて，リアルタイムカーネルのオーバヘッド，とりわけタスク切換えのオーバヘッドを考慮に入れる必要がある．タスク切換えのオーバヘッドは，タスクの切換え頻度に比例するため，必要なプロセッサの処理能力は，処理の実行頻度が高くなるほど大きくなる．

SIO システムなどのプロトコルスタックを用いた通信システムでは，上位レイヤほど処理は複雑であるが実行頻度は低く，逆に下位のレイヤになるほど処理は単純になるが実行頻度が高くなる．たとえば SIO システムでは，上位層では PPP パケットの生成/解析という比較的複雑な処理を行うが，起動頻度は PPP パケット 1 パケットごとである．それに対して，下位層ではシリアルライン上でビットデータを送受信する単純な処理を行うが，ボーレートごとの起動が必要となる．そのため，上位レイヤほどソフトウェア，下位レイヤほどはハードウェアでの実現が適しているといえる．

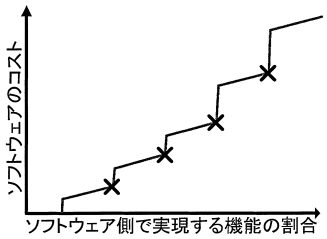

図 8 ソフトウェアコスト
Fig. 8 Software cost.

このようなシステムにおいて，ソフトウェア側で実現する機能の割合を高くするということは，ソフトウェアで実現されるレイヤを下位に広げていくことである．そのため，ソフトウェアの割合を高くするほど，ソフトウェアで実現される処理の実行頻度が高くなる．実行頻度の上昇は機能の増加と共に連続的に上昇するのではなく，レイヤの境界などで不連続に上昇する．前述のようにソフトウェアのコストには，タスク切換えのオーバヘッドが含まれるため，ソフトウェアのコストは，実行頻度が不連続に高くなる点で，不連続に増加することになる．この様子を**図 8** に示す．このことから，実行頻度が不連続に増加する直前の点（図 8 に×印で示した点）で，システムのトータルコストが極小になる可能性が高く，ソフトウェアとハードウェアの分割点として有力であると考えられる．

5.3.2 同期・通信機構のチャネルへの集約

実行タイミングの異なるビヘイビア間の同期・通信機構は，チャネルに集約して記述した．同期・通信機構をチャネルに集約することで，機能の記述と同期・通信の記述を独立化することは，SpecC にチャネルが導入された目的である．実際，SIO システムの SpecC 記述においても，ビヘイビア内にはそれ自身の機能のみを記述すればよく，ビヘイビアの記述性や可読性が向上することが確認できた．特に，tx_ppp や tx_byte のように，上位層と下位層の 2 つのビヘイビアと通信しなければならない場合には，チャネルなしでの記述は困難であった．このことから，同期・通信機構は可能な限りチャネルに集約して記述すべきであるといえる．

さらに，同期・通信機構をチャネルに集約することで，SpecC 記述からより効率的なリアルタイムカーネ

ル上のソフトウェアへの変換が可能になるという利点もある．これについては，6.1.2 項で詳しく述べる．

5.4 まとめ

以上のように，リセット動作の記述が冗長になるという問題点はあるが，並列性や同期・通信のための構文を用いて SIO システムが記述できたことより，SpecC は 4.1 節であげた評価項目 (1) を満たしているといえる．

6. SpecC 記述の変換

本章では，SpecC 記述から，デバイスドライバ，デバイス，およびその間のインタフェースへの変換手法について述べ，デバイスドライバ，インタフェースの変換については，それぞれ 4.1 節の評価項目 (2) と (3) を評価する．具体的には，ソフトウェアとハードウェアの分割点をバイトチャネルとし，それより上位層の記述をデバイスドライバの C 記述へ，バイトチャネルより下位層の記述をデバイスの VHDL 記述へ，バイトチャネルをその間のインタフェースへ，それぞれ可能な限り機械的に変換した．

変換によって得られた C 記述のソフトウェア部は，μITRON4.0 仕様のリアルタイムカーネルとともにコンパイル・リンクしてロードモジュールとし，VHDL 記述のハードウェア部は FPGA を対象に論理合成・配置配線を行い，プロセッサと FPGA が実装された評価ボードを用いて動作確認を行った．動作確認をシミュレーションによる動作確認と同様の方法で行った結果，シミュレーションと同等の動作をしていることを確認した．

なお，C 記述のソフトウェア部のコンパイルには GNU C Compiler 2.95.2 を，VHDL 記述のハードウェア部の論理合成と配置配線には，それぞれ Exemplar Logic 社の Leonardo Spectrum と ALTERA 社の MAX+plus II を用いた．また，変換後のソフトウェア部が使用する μITRON4.0 仕様のリアルタイムカーネルには，TOPPERS/JSP カーネル[9]を用いた．また，評価ボードとしては，プロセッサに SH-3（日立製 SH7709）を用いた MU200-RSH3（三菱電機マイコン機器ソフトウエア製）と，FPGA に FLEX10K（ALTERA 製 EPF10K10QC208-4）を用いた MU200-EA10（三菱電機マイコン機器ソフトウエア製）を用いた．これらのボードはお互いに接続されており，CPU のバスが FPGA に接続されている．

6.1 デバイスドライバ

SIO システムにおけるデバイスドライバ化の対象は，PPP 生成/解析ビヘイビアからバイト送/受信ビヘイビアまでの記述である．

表 1 デバイスドライバ部の SpecC 記述と C 記述の記述量
Table 1 The amount of description of a device driver part for SpecC and C.

	行数	コードサイズ
SpecC 記述	415 行	13.3 kbyte
C 記述	332 行	9.5 kbyte

この箇所の SpecC 記述と変換後の C 記述の記述量を**表 1** に示す．

以下，ビヘイビアとチャネルの変換方法，優先度の割付け方法について述べ，適用性の評価として 4.1 節の評価項目 (2) を評価する．

6.1.1 ビヘイビアの変換

SpecC のビヘイビアは，並行実行されるものと関数的に呼び出されるものに分類できる．そこで，並行動作するビヘイビアはリアルタイムカーネルの並行動作単位であるタスクへ，その他のビヘイビアは関数へと変換することを基本とする．具体的には，**par** 文により並行実行されるビヘイビアは，それぞれタスクに対応させる．**par** 文を実行したビヘイビアは，**par** 文中に記述したすべてのビヘイビアの実行が終了するまで実行を中断するため，**par** 文を実行する親タスクは，すべての子タスクが実行を終了するまで待ち状態とする．

try-trap-interrupt 構文は，μITRON のタスク例外処理機能[10]を用いた記述に変換する．具体的には，**try** 節に記述したビヘイビアに対応するタスクを用意し，イベントをタスクへの例外要求で実現する．**trap** 節および **interrupt** 節に記述したビヘイビアは関数化しておき，タスクの例外処理ルーチンの中から呼び出す．

その他のビヘイビア，具体的には，逐次実行されるビヘイビアや FSM を記述する **fsm** 構文内のビヘイビアは，関数に変換する．

SIO システムでは，変換対象である PPP 生成/解析ビヘイビアとバイト送/受信ビヘイビアは，並行に動作し内部で他のビヘイビアを呼び出さないため，タスクと 1 対 1 に対応させた．

6.1.2 チャネルの変換

ソフトウェア化されるビヘイビア間を接続するチャネルは，メソッドのアトミックな実行やイベントの通知（**notify** 文）・待ち合わせ（**wait** 文）を，それぞれ μITRON の持つ同期・通信機構であるセマフォやイベントフラグなどで実現することで，リアルタイムカーネル上で動作するソフトウェアに機械的に変換することができる．しかしながら μITRON は，データ

キューなど高レベルの同期・通信機構も持っており，SIO システムに用いたチャネルは，データキューに変換した方が実行効率が高い．

SIO システムでは，ここで変換対象となるチャネルは PPP パケット用チャネルと IP パケット用チャネルである．これらのチャネルは，μITRON のデータキューにより実現することができた．そのため，C 記述では，SpecC 記述でのチャネルの記述に相当する記述が必要でないため（データキューのコードはカーネルに含まれている），表 1 に示すようにコードサイズが減少した．

6.1.3 優先度の割付け

μITRON の各タスクには，時間制約を満たすように優先度を割り付ける必要があるが，SpecC のビヘイビアには優先度の概念がないために，何らかの方法でタスクの優先度を決定することが必要である．

SIO システムでは，代表的な静的優先度割付け手法である Deadline Monotonic Scheduling[11]を用いて，タスクの優先度を決定した．この手法は，デッドラインの短いタスクほど高い優先度を割り付けるもので，SIO システムでは，実行頻度が高いビヘイビアほどデッドラインが短いために，バイト送/受信タスクに PPP 生成/解析タスクより高い優先度を設定した．

6.1.4 ま と め

以上の結果より，4.1 節の評価項目 (2) を評価する．

- ビヘイビアの変換

ビヘイビアを機械的に変換するためには，ビヘイビアをタスクとするか関数とするかの判定方法がポイントとなる．6.1.1 項で述べた規則は，SpecC 記述の比較的単純な解析により判定可能であるため，機械的な変換が可能と考えられる．

- チャネルの変換

6.1.2 項で述べたように，μITRON の低レベルの同期/ 通信機構を用いて，チャネルを機械的に変換することは可能である．それに対して，図 7 に示したようなコードから，その機能が μITRON のデータキューで実現できることを機械的に判定するのは容易ではない．結果として，機械的な変換で得られるソフトウェアは，効率的なものであるとは限らないことになる．

類似のチャネルが頻繁に使用されることに着目すると，効率的なソフトウェアへと変換するために，チャネルのライブラリを用意する方法が有力と考えられる．ライブラリに含まれるチャネルに対しては，μITRON の同期・通信機構を用いた変換方法をあらかじめ用意しておくことで，効率的なソフトウェアへ機械的に変換することが可能になる．ただし，チャネルへ受け渡すデータの型は，チャネルが利用される状況によって異なるため，C++の STL（Standard Template Library）のように，ライブラリに含まれるチャネルを任意のデータ型に適用できる機構が望まれる．

以上より，ソフトウェア化されるチャネルを機械的に変換するためには，できる限りライブラリに含まれるチャネルを用いることとし，それ以外のチャネルについては低レベルの同期・通信機構を用いて変換するのが妥当である．

- 優先度の割付け

最適な優先度割付けを機械的に行うためには，各タスクのデッドラインなど SpecC 記述に含まれない情報が必要であり，これらの情報を別途与える必要がある．ビヘイビアの実行頻度については，SpecC 記述のシミュレーションにより求めることも可能で，それにより近似的な優先度割付けは機械化できる．最適な優先度割付けに必要なデッドラインなどの情報を SpecC 記述に追加するかは，今後の課題である．

以上より，SpecC の提供する並行性や同期・通信を記述するための構文から，リアルタイムカーネルの機能を用いたデバイスドライバに機械的に変換可能であるといえる．また，効率的なソフトウェアへの変換に関しては，チャネルのライブラリを用意することで，実現可能であるといえる．

表 2 デバイス部の SpecC 記述と VHDL 記述の記述
Table 2 The amount of description of a device part for SpecC and VHDL.

	行数	コードサイズ
SpecC 記述	129 行	3.0 kbyte
VHDL 記述	319 行	8.9 kbyte

6.2 デバイス

SIO システムにおけるデバイス化の対象は，ビット送/受信ビヘイビア以下の階層の記述である．これらの記述を，FSM 動作を行う VHDL の **process** 文に変換した．ビット送/受信以下の階層のビヘイビアは，クロックに同期した動作をするものであり，SpecC 記述の時点でクロックを考慮した記述となっている．そのため，効率的なデバイス記述にほぼ機械的に変換することができた．一般には，この変換は容易ではない．

表 2 にデバイス化した箇所の SpecC 記述と変換後の VHDL 記述の記述量を示す．

また，後述のインタフェースのハードウェア部と合わせて FPGA 上に実装した場合のセル数は 231 セル，

最大動作周波数は 37MHz であった．

6.3 インタフェース

本研究では，5.3.1 項で述べた指針により，ソフトウェアとハードウェアの有力な分割点でビヘイビアを切り分けており，ビヘイビア間のチャネルを，ソフトウェアとハードウェアの境界とする．すなわち，ソフトウェア化されるビヘイビアとハードウェア化されるビヘイビアを接続するチャネルを，ソフトウェアとハードウェアの間のインタフェースに変換すればよい．

以下では，チャネルのインタフェースへ変換方法について述べ，適用性の評価として，4.1 節の評価項目 (3) を評価する．

6.3.1 チャネル内変数の記憶場所の決定

チャネルの内部変数（イベントを除く）の記憶領域は，メインメモリに確保する方法，プロセッサとデバイスとの共有メモリ上に確保する方法，デバイス側にデバイスレジスタとして用意する方法がある．この中でメインメモリに確保する方法は，システム構成によって実現困難な場合があるため，小さい記憶領域はデバイスレジスタとして，大きい記憶領域は共有メモリ上に確保する方法が妥当と考えられる．

6.3.2 イベントの変換

ハードウェアとソフトウェアを接続するチャネルにおいては，内部で用いられるイベントを，ソフトウェアからハードウェアへのイベントと，ハードウェアからソフトウェアへのイベントに分類することができる．

ソフトウェアからハードウェアへのイベントは，デバイス側にデバイスレジスタを用意し，その中の 1 ビットを割り当てる．デバイスドライバ側からの **notify** 文は，デバイスレジスタの該当ビットをセットすることで実現する．デバイス側の **wait** 文は，このビットの立ち上がりエッジを監視し，エッジの検出をイベントの通知と見なす．

ハードウェアからソフトウェアへのイベントには，デバイスからプロセッサへの割込みを利用する．デバイス側からの **notify** 文は，割込み発生回路と，μITRON のシステムコールによりタスクにイベントを通知する割込みハンドラに変換される．イベントを待つデバイスドライバ側の **wait** 文は，このイベント通知を待つシステムコールに変換される．

しかしながらこの方法では，デバイスを使用しない場合にも割込みを発生させてしまう．そのため，割込みの許可/禁止のためのビットをデバイスレジスタに用意し，デバイスドライバ側がイベント待ちに入る前に割込みを許可し，割込みが受け付けられた後にそれ以上の割込みを禁止する．

6.3.3 グルーロジック

グルーロジックとしては，バスインタフェースとアドレスデコーダが必要となる．バスインタフェースは，デバイスを接続するバスの仕様にあわせてあらかじめ用意しておく．

6.3.4 送信部のバイトチャネルの変換

前述の変換方法を，送信部のバイトチャネル（図 7）をインタフェースに変換する例を用いて，詳しく説明する．このチャネルは，デバイスドライバ側が write メソッドによりバイトデータを書き込み，デバイス側は read メソッドによりデータを読み取る．

チャネル内の要素のうち，デバイスレジスタへの変換対象となるのは，data_buffer と data_valid の 2 つの変数と，データ書き込み終了を通知するイベント sync_write，割込みの禁止/許可を示すビットである．これらを，デバイスレジスタ 2 個に変換した．それぞれのデバイスレジスタの詳細を以下に示す．

- **REG0** 送信データレジスタ．バッファ変数（data_buffer）に対応．
- **REG1** 送信コントロールレジスタ．
  ビット 0：バッファ・ステータス変数（data_valid）に対応．‘1’ が true，‘0’ が false．
  ビット 1：データ書き込み終了通知（sync_write）に対応．‘1’ の書き込みによりイベントを通知．
  ビット 4：割込み許可/禁止ビット．‘1’ で割込み許可，‘0’ で割込み禁止．

データの読み込みを通知するイベント sync_read は，割込み発生回路と，イベントフラグによりイベント通知を行う割込みハンドラへと変換した．

以上の変換より作成したデバイスアクセス関数（driver_send）および割込みハンドラ（send_handler）と，バイトチャネルの read メソッドおよび write メソッドとの対応を**図 9** に示す．デバイス側には，イベント待ちに対応して REG0 のビット 1 のエッジ待ちをする回路，data_valid への true 書き込みに対応してデータ送信終了時に REG0 ビット 0 をセットする回路，sync_read に対する **notify** 文に対応して，割込みを発生する回路を記述した．

同様に受信部部のバイトチャネルについても同様に変換した際の送信部・受信部両方のバイトチャネルの SpecC 記述と変換後の記述量を**表 3** に示す．

6.3.5 ま と め

6.3.4 項に示したように，インタフェースのソフトウェア側については，機械的な変換が可能である．ハードウェア側については，バスインタフェース回路をバスの種類ごとに用意しておくことで，自動変換が可能

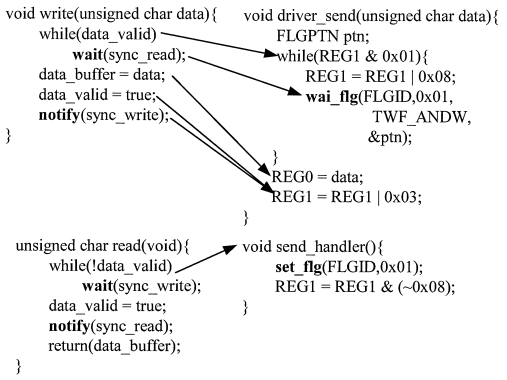

図 9 バイトチャネルのインタフェース変換（ソフトウェア部）
Fig. 9 Conversion of byte channel to interface (software part).

表 3 インタフェース部の SpecC 記述と変換後の記述量
Table 3 The amount of description of a interface part for SpecC and after conversion.

	行数	コードサイズ
SpecC 記述	33 行	0.7 kbyte
ソフトウェア部（C 記述）	66 行	1.2 kbyte
ハードウェア部（VHDL 記述）	280 行	7.2 kbyte
グルーロジック（VHDL 記述）	295 行	10.5 kbyte

になると考えられる．

また，6.1.2 項で述べたのと同様に，チャネルをライブラリ化しチャネルごとのインタフェース記述を用意しておくことで，より効率的な実装への機械的な変換が可能になる．

以上より，SpecC 記述からソフトウェア・ハードウェア間のインタフェースが機械的に変換可能であり，4.1 節の評価項目 (3) を満たしているといえる．しかしながら，この変換は，5.3.1 項で述べた指針に従って記述し，インタフェースへの変換対象をチャネルとした場合にのみいえることである．5.3.1 項で述べた指針に従わずに記述した場合は，ビヘイビア内部でデバイスドライバとデバイスを分割する可能性があるため，ビヘイビア記述の変更が避けられず，機械的な変換は困難である．

7. 切り分けの変更

ソフトウェアとハードウェアの最適な分割点は，システムに対する要求や制約によって異なるため，一体記述からの変換において，デバイスドライバとデバイスの切り分けを柔軟に変更できるとメリットが大きい．両者の切り分けを変更可能とするためには，1 つのビヘイビア記述から，ソフトウェアとハードウェアのいずれにも変換できることが必要である．

1 つの例として，SIO システムにおいて，ソフトウェアとハードウェアの分割点を PPP パケット用チャネルに変更した場合を検討する．PPP パケット用チャネルからインタフェースへの変換については，バイトチャネルと同様の方法で機械的に行うことができる．分割点の変更により，バイト送/受信ビヘイビアがハードウェアで実装されることになる．PPP パケット用チャネルがポインタ渡しを用いているため，バイト送/受信ビヘイビアには，ポインタによるメモリアクセスが含まれる．

ポインタによるメモリアクセスをそのままハードウェア化するには，DMA によるメモリアクセスを用いる必要がある．そのためには，バスインタフェース回路と同様に，バスの種類ごとに DMA 回路も用意しておけばよい．SIO システムのバイト送/受信ビヘイビアは，メモリアクセス以外には複雑な処理を行っておらず，メモリアクセスも含めて機械的な変換が可能と考えられる．ただし，我々が用いている評価ボードでは，デバイス側をバスマスタとして動作させることが不可能であったため，その実証は今後の課題とする．

さらに上位の PPP 生成/解析ビヘイビアは，クロックをまったく考慮していない SpecC 記述となっており，処理自身もやや複雑であるため，ハードウェアへの機械的な変換は容易ではないと考えられる．クロックを考慮しないビヘイビアレベルの記述から，クロックを考慮した RTL レベルへの変換に関しては，他の研究成果を利用することを考えている．

8. ま と め

我々は，デバイスドライバ開発を効率化する観点から，デバイスドライバとデバイスの一体設計手法について研究を行っている．本論文では，その第 1 段階として，SIO システムを例として SpecC による記述と手作業での実装を行い，デバイスドライバとデバイスの一体設計への SpecC の適用性の評価を行った．

評価の結果，ビヘイビアを実行頻度により切り分ける，同期・通信機構をチャネルに集約するといった記述指針に従って記述し，チャネルのライブラリを用意することで，記述した SIO システムを機械的に変換することが可能であることが分かった．あわせて，並行性や同期・通信のための構文を用いて SIO システムが記述できたため，SpecC がデバイスドライバとデバイスの一体設計手法の一体記述言語として適用できることを確認した．

今後の研究の方向性としては，デバイスドライバとデバイスの切り分けの変更が可能であるかについてさ

らに評価するとともに，SIO システムとは別の例についても記述・実装を行う．さらに，それらの成果をふまえて，デバイスドライバとデバイスの自動生成システムの実現へと研究を進めていく計画である．

謝辞 SpecC 記述に関してご意見を下さった SpecC Technology Open Consortium（STOC）の事例 WG のメンバ各位に感謝します．

参 考 文 献

1) 高田広章：組込みシステム開発技術の現状と展望，情報処理学会論文誌，Vol.42, No.4, pp.930–938 (2001).
2) 高田広章：ハード設計者が苦労して得た性能をソフト開発者が浪費する!!，*Design Wave Magazine*, Vol.2, pp.71–72 (2001).
3) Gajski, D.D., Zhu, J., Dömer, R., Gerstlauer, A. and Zhao, S.: *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers (2000).
4) 村中健二，高田広章ほか：ITRON デバイスドライバ設計ガイドラインの移植性と評価，信学技法（2001 年実時間処理に関するワークショップ），Vol.2001, No.21, pp.23–30 (2001).
5) 福田 晃，最所圭三，片山徹朗，中西恒夫：組込みシステム向け実行環境の自動生成，信学技法（2000 年実時間処理に関するワークショップ），Vol.99, No.726, pp.17–22 (2000).
6) Gajski, D., Dutt, N., Wu, C.H. and Lin, Y.L.: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1994).
7) Dömer, R., Gerstlauer, A. and Gajski, D.: SpecC Language Reference Manual Version 1.0. Available from http://www.specc.gr.jp/tech/
8) 坂村 健（監修），高田広章（編）：μITRON4.0 仕様 Ver. 4.00.00，トロン協会 (1999).
9) TOPPERS/JSP Project ホームページ．http://www.ertl.ics.tut.ac.jp/TOPPERS/
10) 本田晋也，高田広章：μITRON4.0 仕様の例外処理のための機能とその評価，情報処理学会論文誌，Vol.42, No.6, pp.1514–1524 (2001).
11) Klein, M.H., Ralya, T., Pollak, B., Obenza, R. and Harbour, M.G.: *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*, Kluwer Academic Publishers (1993).

(平成 13 年 9 月 25 日受付)
(平成 14 年 3 月 14 日採録)

本田 晋也

2002 年豊橋技術科学大学大学院情報工学専攻修士課程修了．現在，同大学院電子・情報工学専攻に在学．リアルタイム OS，ソフトウェア・ハードウェアコデザインの研究に従事．修士（工学）．

高田 広章（正会員）

豊橋技術科学大学情報工学系助教授．1988 年東京大学大学院理学系研究科情報科学専攻修士課程修了．同学科の助手などを経て，1997 年 12 月より現職．リアルタイム OS，リアルタイムスケジューリング理論，組込みシステム開発技術などの研究に従事．ITRON 仕様の標準化活動に，中心的メンバとして参加．博士（理学）．IEEE, ACM，電子情報通信学会，日本ソフトウェア科学会各会員．