システムソフトウェアと 91－6
オペレーティング・システム
（2002．8．21）

USB デバイスのソフトウェアを対象とした SpecC による協調設計

片山徹郎 * 福元善之 **

宮崎大学 工学部 情報システム工学科
〒 889-2192 宮崎市学園木花台西 1-1
*E-mail : kat@cs.miyazaki-u.ac.jp
** 現在、富士ソフト ABC 株式会社

本研究は、システム・レベル言語 SpecC を用いて、 USB(Universal Serial Bus) デバイスを使用したシステムの協調設計を行ない、設計生産性を向上させることを目標としている。本稿では、その予備研究として、 USB のソフトウェア側であるデバイスドライバとファームウェアの協調設計を試みた。まず、 Linux の USB デバイスドライバとファームウェアの仕様を SpecC で記述し、その仕様をシミュレーションで検証した。次に、 SpecC で記述した仕様から、コンパイル可能なデバイスドライバとファームウェアのソースコードに変換するデコーダを、言語 Perl で製作した。デコーダが生成した 2 つのソースコードを実行し、仕様どおりに動作することを確認した。

Co-design of Device Driver and Firmware for USB Devices in SpecC

Tetsuro Katayama Yoshiyuki Fukumoto

Department of Computer Science and Systems Engineering,
Faculty of Engineering, Miyazaki University
1-1 Gakuen-kibanadai nishi, Miyazaki 889-2192, Japan.

This research aims at increasing productivity of design for systems used USB(Universal Serial Bus) by co-design with SpecC, which is one of system level languages. As a preparatory research, this paper attempts co-design of device driver and firmware which are softwares of USB. First, specification of USB device driver and firmware of Linux are described with SpecC, and the specification was verified with simulation. Next, a decoder which transforms the specification into the source codes of device driver and firmware which can be compiled is implemented in Perl language. Executing the both source codes which are generated by the decoder confirmed that they satisfied the specification.

1 はじめに

LSI の構造は年々複雑度が増加している。しかしながら、その複雑度に設計生産性の向上度が追いついていないという現状がある [1]。そこで、そのギャップを解消する一つの解決策として、設計の抽象度を上げることが提案されている。抽象度を上げることで生産性の問題を克服し、効率や生産性を向上させることが可能となる。抽象度を上げた場合に考えられるシステム LSI の設計手法としては、以下が考えられる [2]。

- プラットホームベースによる設計。
- IP (Intellectual Property: 知的財産) を組み合わせる設計。
- システムの自動合成。

これらの設計手法を実現するための具体的な方法論の一つに、システム・レベル言語がある。システム・

レベル言語は、仕様を検討したり、実際のシステムの作成を担当するソフトウェア開発者や回路設計者に仕様を伝えるために利用される。

システム・レベル言語の利点は、以下がある。

- 仕様書が自然言語で記述されていることに起因するトラブルを解消可能
- 実行可能 (シミュレーションが可能)
- ハードウェア・ソフトウェアの協調検証が可能
- 仕様から実現までの記述言語の統一が可能

本研究では、日本を中心に普及推進活動が進められているシステム・レベル言語 SpecC[3] を用いて、現在 PC の様々な周辺機器の接続形式として利用され、今後さらに利用が増加すると予想される USB (Universal Serial Bus) デバイス [4] を使用したシステムの協調設計を行ない、設計生産性を向上させることを目標としている。本稿では、その予備研究として、 USB のソフトウェア側であるデバイスドライバとファームウェアを協調設計することを試みる。具体的には、まず、 Linux USB デバイスドライバとファームウェアの仕様を、システム・レベル言語 SpecC を用いて記述する。次に、記述した仕様をファームウェアとデバイスドライバとに分割し、かつ、それぞれコンパイルできる形式に変換するデコーダを製作する。例として、デバイスドライバから送られてきたデータを解析して、そのデータに応じて 7 セグメントと単体 LED を点灯させるというシステムの構築を取り上げる。

デコーダの製作には、文字列処理に適したプログラミング言語 Perl[5] を使用する。対象の USB デバイスとしてサイプレス社製の EZ-USB (AN2131S)[6] を使用し、それを搭載しているプライムシステムズ社製の FLEX10KE 評価キット [7] を用いた。

2 研究の準備

この章では、研究に用いたデバイスなどについて簡単に説明する。

2.1 SpecC

SpecC は、カリフォルニア大学アーバイン校が中心となって多くの大学や有力企業、業界団体の支援を得て、多年に渡る研究開発により誕生した、システムを記述・設計するためのシステム・レベル言語である。仕様記述のために、 ANCI C の構文にハードウェアやリアルタイム処理を伴うソフトウェアの動作を表現するための文法を追加している (詳細は [1])。

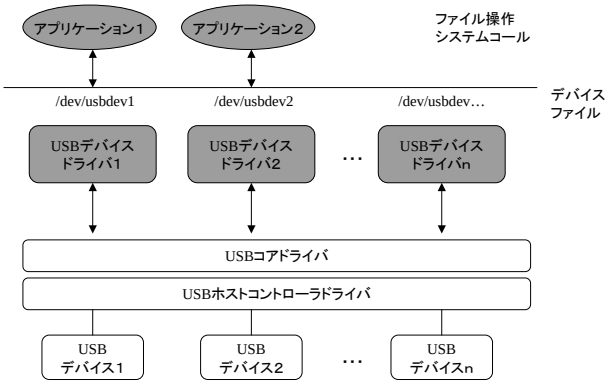

図 1: Linux USB ドライバの構造

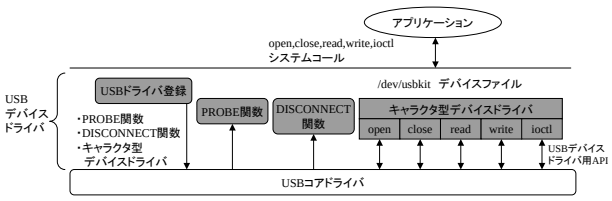

図 2: USB デバイスドライバ

2.2 Linux USB ドライバ

Linux USB ドライバの構造を図 1に示す。ドライバに割り付けられたデバイスファイルに対して、 open, close, ioctl などのシステムコールを発行して USB デバイスを操作する。 USB デバイスドライバは各デバイスごとに存在し、アプリケーションからの要求を処理する。

USB コアドライバは、 USB デバイスドライバから呼び出す関数を提供する部分である。すなわち、 USB デバイスドライバは、 USB コア関数群を利用して USB デバイスとのデータ転送を行なう。

USB デバイスドライバ部分の構造を図 2に示す。 USB デバイスドライバを構成する各部分の役割を以下で説明する (詳細は [4])。

- USB デバイスドライバ登録 — USB デバイスドライバの Linux カーネル組み込みの初期化時に登録処理を実行する。具体的には、 USB デバイスドライバを構成する「PROBE 関数」、「DISCONNECT 関数」、「キャラクタ型デバイスドライバ」を USB コアドライバへ登録する。 USB

コアドライバが USB デバイスを検出すると、登録済みの PROBE 関数を呼び出し、キャラクタ型デバイスを機能させることができる。

- PROBE 関数 — USB デバイスの接続時のドライバ検出関数である。USB コアドライバが USB デバイスを検出すると、PROBE 関数を呼び出す。呼び出された PROBE 関数は、VenderID、ProductID、エンドポイントなどのデバイス情報が引数に与えられ、デバイス情報から自分の担当する USB デバイスかどうかを判断する。PROBE 関数が対象デバイスであると判断すると、ドライバが使用するメモリなどのリソース確保を行ない、USB デバイスドライバが使用可能となる。
- DISCONNECT 関数 — USB デバイスの取り外し時に呼び出される終了処理であり、PROBE 関数で確保したリソースの解放を行なう。PROBE 関数、DISCONNECT 関数を使用することによって、デバイスの接続中だけリソースを消費する仕組みである。
- キャラクタ型デバイスドライバ —アプリケーションが USB デバイスを操作するためのデバイスドライバである。ファイルシステム上に名前が与えられたデバイスファイルがあり、open, close, ioctl などのファイル操作システムコールで USB デバイスを操作できる。この部分の構造は、他の Linux キャラクタ型デバイスドライバとまったく同じである。

2.3 EZ-USB と FLEX10KE 評価キット

本研究では、対象の USB デバイスとしてサイプレス社製の EZ-USB (AN2131S)[6] を使用した。このデバイスは、8051 を CPU コアにした USB コントローラを持っており、サイプレス社が無償で配布しているツールを使って、USB からプログラムをダウンロードして実行することができる。このように、簡単に USB のファームウェアのプログラムを実行できる特徴を持つため、本研究ではこのデバイスを使用した。

また、この EZ-USB を搭載しているプライムシステムズ社製の FLEX10KE 評価キット [7] を使用した。これは、アルテラ社製 FLEX10KE シリーズを搭載した PLD 評価キットである。FLEX10KE 評価キットの外観を図 3に示す。

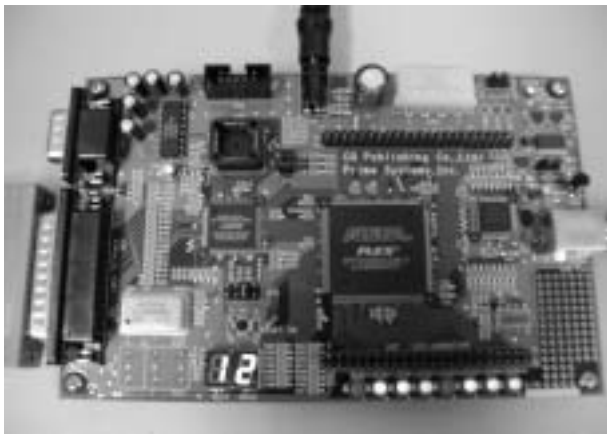

図 3: FLEX10KE 評価キット

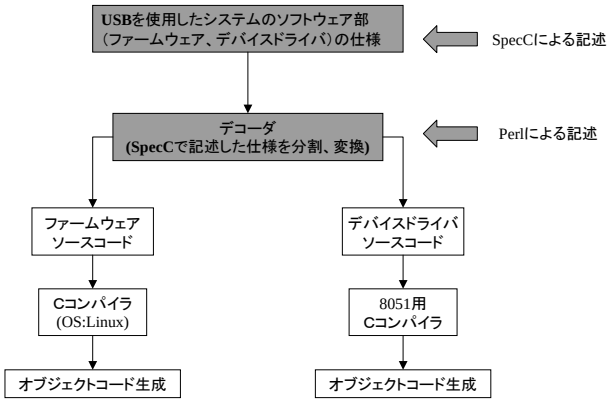

図 4: 本研究の流れ

3 SpecC 言語での記述

本稿における、USB デバイスを使用したシステムのソフトウェア部の流れを、図 4に示す。ソフトウェア部の仕様を SpecC で記述し、SpecC で記述した仕様を分割、変換するデコーダを Perl で記述することにより実現する。この章では、SpecC でシステムの設計方針を含めた記述方法を述べる。例として、デバイスドライバから送られてきたデータを解析して、そのデータに応じて 7 セグメントと単体 LED を点灯させるというシステムの構築を取り上げる。なお、デバイスドライバは、デバイスを同時に使用できる数を一つのみとし、多重オープンを禁止する。

3.1 仕様の設計方針

USB デバイスを使用したシステムの仕様を図 5のように設計する。その際の記述の規則を以下のように定

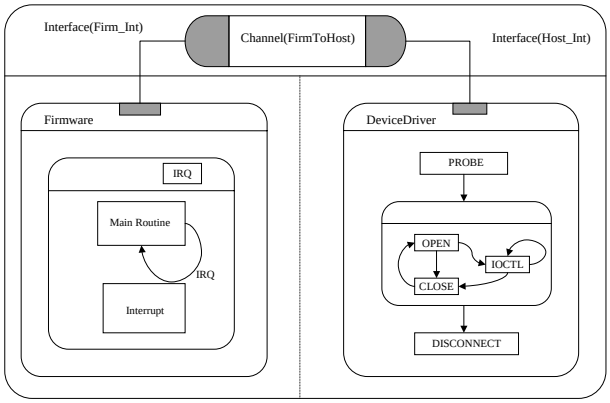

図 5: システムの仕様

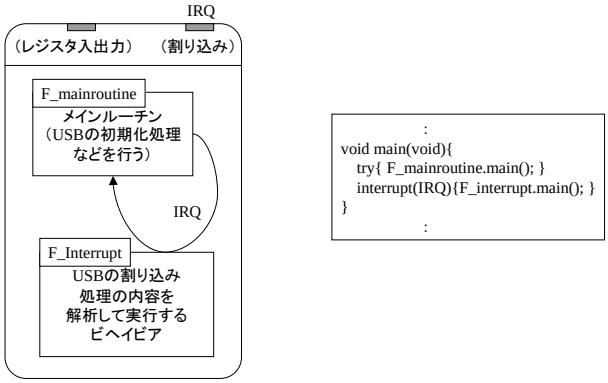

図 6: ファームウェアの仕様と SpecC による記述

めた。この中には、デコーダでの分割処理を容易に行なうためのものも含む。

- ファームウェアで実装するビヘイビアは、ビヘイビア名の先頭に ”F_”を付け、デバイスドライバで実装するビヘイビアは、先頭に ”D_”を付ける。
- レジスタは、ファームウェアとデバイスドライバを接続しているチャネル内に記述する。
- チャネルにより、コアドライバから以下 USB デバイスまでの動作を隠蔽し、ソフトウェア同士の接続を実現する。
- デバイスドライバ側の処理である、PROBE 関数、DISCONNECT 関数や、OPEN()、CLOSE()、IOCTL() などのシステムコール処理を行なうビヘイビア名の語尾はそれぞれ、_probe, _disconnect, _open, _close, _ioctl と記述する。
- ファームウェアやデバイスドライバの固有の関数を使う場合、チャネルに SpecC 上で同じように動作するように関数を記述する。その際の関数名は、後のデコーダの処理で固有の関数名に戻せる名前にする。例えば、デバイスドライバの終了待ちキュー起動関数 `wakeup` は、`I.wakeup` のように記述する (ここで `I` はインターフェース名)。

3.2 ファームウェアの記述

USB のファームウェアの仕様と SpecC での記述を、図 6に示す。ファームウェアは、メインルーチンでレジスタの初期化を実行したあと、ホストからの割り込み命令を待ち、割り込み命令が発生した際には、

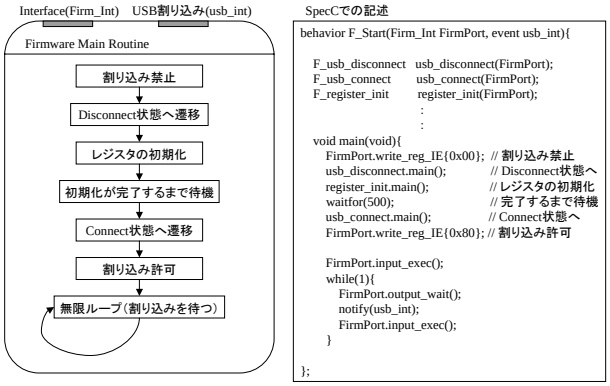

図 7: ファームウェアのメインルーチンの仕様記述

割り込み処理関数に動作を移し、命令を解析して実行する。実行が終了したあとメインルーチンに戻り、また割り込みを待つという処理の流れになる。

メインルーチンの仕様と SpecC の記述の一部を、図 7に示す。メインルーチンは、USB デバイスが起動した際に最初に実行される関数である。ここで、レジスタの初期化を行ない、割り込みを待つという処理をする。

割り込み処理関数の仕様と SpecC の記述の一部を、図 8に示す。割り込み処理関数では、割り込みが入った際に呼ばれ、その内容を処理し、その命令に従った処理を渡すビヘイビアである。

3.3 デバイスドライバの記述

デバイスドライバの記述の際には、ビヘイビア 1 つあたり 1 関数として記述する。

PROBE 関数の仕様を図 9に示す。本研究で記述した PROBE 関数は、まず、デバイス情報の一つである VenderID と ProductID をチェックして担当する

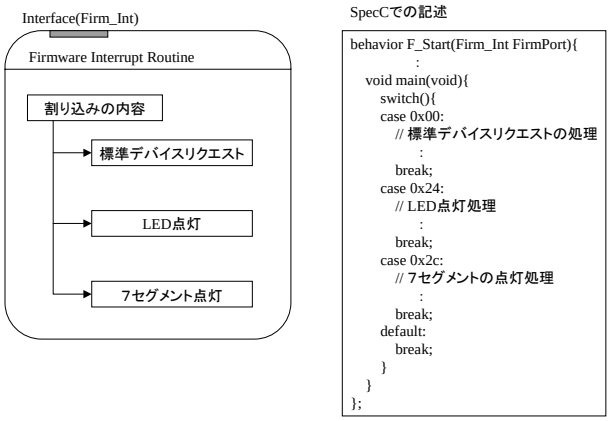

図 8: ファームウェアの割り込み処理関数の仕様記述

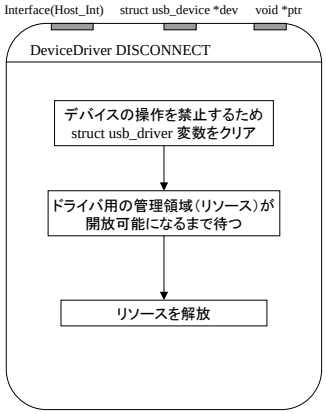

図 10: DISCONNECT 関数の仕様記述

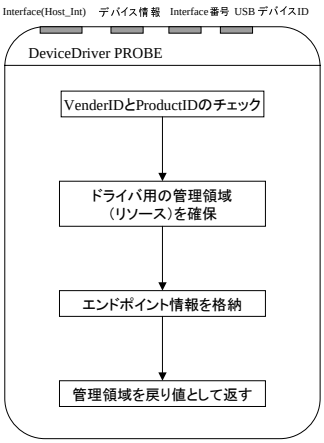

図 9: PROBE 関数の仕様記述

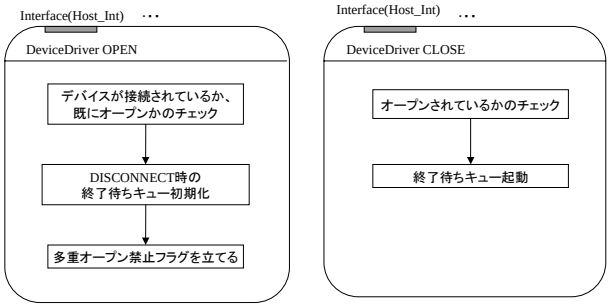

図 11: OPEN(), CLOSE() システムコールの仕様記述

USB デバイスかどうかを調べる。担当するデバイスであった場合、デバイスドライバの管理領域を確保し、エンドポイント情報を確保した管理領域に格納して、管理領域のポインタを返り値として返すという初期処理を行なうこととした。

DISCONNECT 関数の仕様を、図 10に示す。本研究で記述した DISCONNECT 関数は、まず、デバイスの操作を禁止するためにデバイスの情報が入っている usb_driver 構造体へのポインタを NULL ポインタに置き換える。次に、その時点で実行されているタスクの終了を待ち、リソースが開放可能な状態に移行すると、リソースを開放して動作を終了する。

同様に記述した、 OPEN(), CLOSE() システムコール処理関数の仕様を図 11に、 IOCTL() システムコール処理関数の仕様を図 12に、それぞれ示す。

返り値のある関数の動作の記述例を図 13に示す。SpecC では、メインビヘイビア以外のビヘイビア内のメインは必ず `void` 型で、引数を持たない。つまり、ビヘイビアの外からは、ビヘイビア内のメイン関数しか呼べないので、返り値を持つ関数の動作は記述できない。この解決策として、 2 つの方法を考えた。 2 つの方法とも、呼び出す関数に返り値発生用のイベントを増やし、 `try-trap` 構文を使用してそのビヘイビアを呼び出し、エラー発生時には、 `trap` 側に記述されているエラー処理関数に処理がわたるように記述する。異なる点は、返り値の処理やビヘイビア内の記述方法である。図の左側の方法は、実際に動作をする部分をビヘイビア内の別関数として記述し、ビヘイビア内のメイン内でその関数を呼び出し、メイン内で返り値を判定することによりエラー判定を行ない、イベントを起こすという方法である。右側の方法は、ビヘイビアのポートに返り値用のポートを増やし、`return()` 関数を記述する場所で返り値用のポートにエラー値を代入し、イベントを起こす方法である。

右側の方法では、デコーダで処理する際に、ビヘイビアの返り値の型を判定し、メイン関数の型を返り値

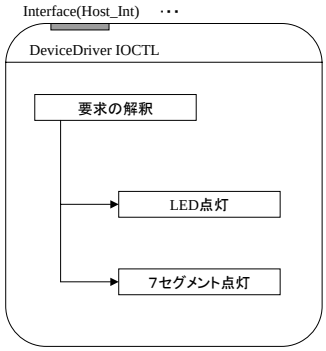

図 12: ioctl() システムコール処理の仕様記述

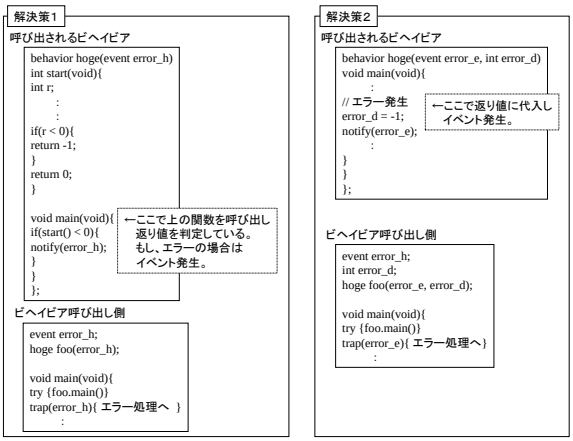

図 13: 返り値のある関数に対する 2 つの解決策

の型に変換する。そして、返り値を代入してイベントを発生している部分を return 関数に変換する必要がある。左側の方法であれば、ビヘイビア内の実際に動作する部分をそのまま抽出すればよいだけである。今回は、デコーダのコーディング量が少なくなる左側の方法を選んだ。また、デコーダでの処理を考慮し、返り値のあるビヘイビアのメイン関数名を start と定義した。

3.4 シミュレーション実行結果

以上のように、 USB デバイスを使用したシステムの仕様を記述した。図 14にその一部を示す。記述した仕様をシミュレーションした結果の一部を図 15に示す。図 15は、 7 セグメントを選択して、値 16 を入力している。出力は、一桁目は ”0”、二桁目は ”1” を表示している。この表示は、 16 進表示であるため出力が正しいことが分かる。シミュレーションの実行結果から、記述した仕様が正しいことを確認した。

```
behavior F_seg_disp
      (unsigned char dat, Firm_Int FirmPort){
  unsigned char adrs;
  unsigned char data1;
  F_portb_out portb_out(adrs, data1, FirmPort);
  void main(void){
    adrs = 0x02;
    data1 = led_pat[dat & 0xf];
    portb_out.main();
    adrs = 0x04;
    data1 = led_pat[(dat >> 4)&0xf];
    portb_out.main();
   }
};

behavior D_ezusb_ioctl
      (struct inode *inode, struct file *file,
       unsigned int cmd, unsigned long arg,
       Host_Int HostPort, event function_end){
  int led, seg;
  ezusb_data_t *pdata;
  event error_end_bulk;
  int flag;
  D_ezusb_set_bulk set_led
      (pdata, led, LED, HostPort, error_end_bulk);
  D_ezusb_set_bulk set_seg
      (pdata, seg, SEG, HostPort, error_end_bulk);

  int start(void){
    pdata = file->private_data;
    flag= 0;
    switch(cmd){
        case EZUSB_SET_LED:       // LED 設定
            led = (int)arg;
            set_led.main();
            break;
        case EZUSB_SET_SEG:       // 7SEG 設定
            seg = (int)arg;
            set_seg.main();
            break;
        default:
            return(-ENOIOCTLCMD);
        }
        return(0);
    }
    void main(void){
        if(start() < 0){
            notify(function_end);
        }
    }
};
```

図 14: SpecC で記述したシステムの仕様の一部

4 デコーダ

USB デバイスを使用したシステムの SpecC 言語で記述したソフトウェア部を、ファームウェアとデバイスドライバとに分割し、かつ、それぞれをコンパイルし実行できる形式に変換するデコーダを製作する。デコーダを実装する言語として、テキスト処理に優れているプログラミング言語 Perl[5] を使用する。

4.1 共通の処理内容

クラスの判別については、 SpecC でのクラスは、大きく分けてビヘイビアクラス、チャネルクラス、

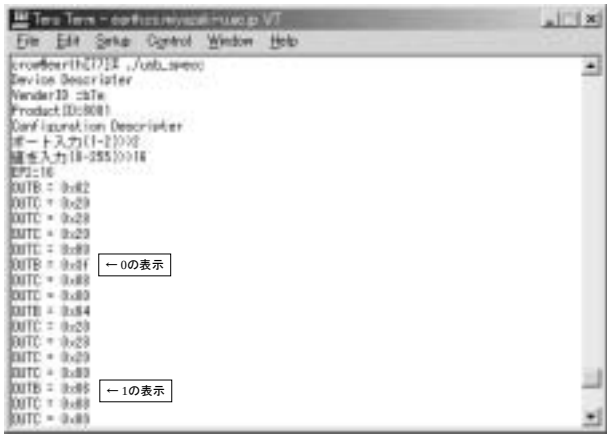

図 15: シミュレーション実行結果の一部

インターフェースクラスに分かれる。これらのクラスは、それぞれ必ず behavior, channel, interface で始まり、最後は };で終わる。これを利用することにより、全部のクラスを別々に処理できる。

ビヘイビアの分割については、仕様記述の段階で、ビヘイビアの名前の先頭に、ファームウェア側の処理ならば"F_"、ドライバ側の処理ならば"D_"を付けた。そのタグを目印にファームウェア側とデバイスドライバ側とに以降の処理を分けることができる。以下にビヘイビア内のデコード処理について説明する。

- ビヘイビア内のメイン関数の関数名と引数を、ビヘイビア名とその引数に変換
- 外部ビヘイビアのビヘイビア名を置換 — 外部のビヘイビアを呼び出す場合には、ビヘイビアを宣言する際に渡す引数を設定し、ビヘイビアを実際に呼び出す際は、ビヘイビアのメイン関数を呼び出す。そこで、ビヘイビア名の内部的な関数宣言を記憶し、その関数を呼び出している箇所を「ビヘイビア名+引数」という形に置換する。
- interface が引数に含まれている場合、そのインターフェースを削除 — 引数に含まれるインターフェースは、デコード後には必要ないので、その宣言箇所も含めて削除する。
- ビヘイビア内の関数の外に変数が定義されている場合、ビヘイビア内の関数の分割の際に変数定義をビヘイビア内の全関数内に記述
- ファームウェアやデバイスドライバの固有の関数を使っている場合、チャネルに SpecC 上で同じように動作するように記述した関数があるので、それを固有の関数名に変換
- 不変的な情報の記述 — ファームウェアのレジストリの宣言は不変的なものなので、この宣言をしてある別ファイルをデコーダ実行時に読み込み、ファームウェアのソースコードに書き出す。

4.2 ファームウェア側の処理

レジスタについては、本研究では、チャネル内に記述している。このため、ファームウェアからは interface の関数を使用してレジスタのデータを Read/Write する。そこで、レジスタ 1つにつき Read 用と Write 用の 2 つの関数を用意する。

割り込み処理関数については、8051 のコンパイラで割り込み処理をするためには、interrupt(8) が関数の先頭に必要である。このため、デコーダで関数の先頭に付加する。

4.3 デバイスドライバ側の処理

デバイスドライバの変数や関数の宣言については、すべて static を先頭に付加する。この理由は、カーネルは膨大なオブジェクト・モジュールをリンクして作られるため、関数をグローバルにしてしまうと、名前の衝突が起こり、リンクで失敗することが考えられるためである。

ビヘイビアの返り値に伴う処理については、返り値のあるビヘイビアのメイン関数名を start と定義したので、この start 関数をビヘイビアの宣言名に置き換える。それに伴い、メイン関数は削除する。

fileoperations 構造体と USB ドライバ登録用データの登録については、デバイスドライバのモジュール登録時に必要となる fileopetarions 構造体に、USB のシステムコール処理の関数を登録する。本研究では、OPEN, CLOSE, IOCTL の処理関数が該当する。それぞれ、ビヘイビア名の最後に、_open, _close, _ioctl と付けたものを登録する。USB ドライバ登録用データでも同様である。

4.4 実行結果

3 章で記述した USB デバイスを使用したシステムの仕様 (図 14に一部を示した) を入力として、デコーダを実行した。デコーダの実行結果として、ファームウェア側の出力結果と、デバイスドライバ側の出力結果とを、それぞれ図 16と図 17に示す。生成した 2 つのソースコードをコンパイルしてオブジェクトコード

```
void seg_disp(unsigned char dat){

unsigned char adrs;
unsigned char data1;
adrs = 0x02;
data1 = led_pat[dat & 0xf];
portb_out(adrs, data1);

adrs = 0x04;
data1 = led_pat[(dat >> 4)&0xf];
portb_out(adrs, data1);
}
```

図 16: ファームウェア側出力結果の一部

```
static int ezusb_ioctl
      (struct inode *inode, struct file *file,
       unsigned int cmd, unsigned long arg){
static int led, seg;
statuc ezusb_data_t *pdata;
static int flag;
pdata = file->private_data;
flag= 0;
switch(cmd){
case EZUSB_SET_LED:      // LED 設定
led = (int)arg;
ezusb_set_bulk(pdata, led, LED);
break;

case EZUSB_SET_SEG:      // 7SEG 設定
seg = (int)arg;
ezusb_set_bulk(pdata, seg, SEG);
break;

default:
return(-ENOIOCTLCMD);
}
return(0);
}
```

図 17: デバイスドライバ側出力結果の一部

を生成することができた。生成したコードを実行し、仕様どおりに動作することを確認した。

5 おわりに

本稿では、USB デバイスの協調設計をするための第一段階として、USB デバイスを使用したシステムのソフトウェア部分であるデバイスドライバとファームウェアの協調設計を試みた。Linux の USB デバイスドライバとファームウェアの仕様をシステム・レベル言語 SpecC で記述し、その実行をシミュレーションで検証できた。さらに、SpecC で記述した仕様から、コンパイル可能なデバイスドライバとファームウェアのソースコードに変換するデコーダを、言語 Perl で製作した。デコーダが生成したデバイスドライバとファームウェアの 2 つのソースコードを実行し、仕様どおりに動作することを確認した。

SpecC を用いてソフトウェアを記述しようとすると、少なからず不便な面があった。例えば、ビヘイビア内のメイン関数は、引数、返り値共に必ず void 型ではなくてはならないなどである。これらについては、SpecC の更なる拡張などが待たれる [8]。

SpecC を用いたデバイスドライバの作成という研究は既に行なわれているが [9]、USB を対象とした研究は、ほとんど行なわれていない。今回、USB デバイスのソフトウェア部分のみではあるが、SpecC で記述できたことにより、今後、その需要が拡大すると予想される USB デバイスの設計生産性向上につながると考えられる。

今後の課題として、以下が挙げられる。

- ハードウェアを含めた協調設計の仕様記述。それに伴うデコーダの開発 — 今回は、ソフトウェアだけの協調設計を行なった。しかしながら、システム・レベル設計とは、本来、ハードウェアを含めたシステム全体の協調設計を行なうことが必要であり、今後研究を進めていく。
- SpecC で記述するための方法論や、IP の充実 — 今回、ソフトウェア的な動作を SpecC で記述しようとすると、少なからず不便な面があった。この問題を解消するために、SpecC の拡張や、記述するための方法論の充実が必要である。また、他のシステムへの適用例を増やしていく事が必要である。

参考文献

[1] D. D. Gajiski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao (木下常雄, 富山宏之訳): “SpecC 仕様記述言語と方法論,” CQ 出版社 (2000).

[2] “組み込みソフトと LSI を C 言語でコデザイン〜システムアーキテクトを目指せ〜,” Design Wave Magazine 2000 年 7 月号, CQ 出版社 (2000).

[3] SpecC Technology Open Consortium (STOC): http://www.specc.gr.jp/

[4] “USB ハード＆ソフト開発のすべて,” TECH I Vol.8, CQ 出版社 (2001).

[5] 武藤健志: “独習 Perl,” 翔泳社 (2000).

[6] Cypress Semiconductor Corporation: http://www.cypress.com

[7] Prime Systems, Inc.: http://www.prime-sys.co.jp

[8] 松本吉弘: “拡張 SpecC によるソフトウェアアーキテクチャ記述,” 第 3 回組込みシステム技術に関するサマーワークショップ (SWEST3) 予稿集, pp.71-82 (2001).

[9] 本田晋也, 高田広章: “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,” 情報処理学会論文誌, Vol.43, No.5, pp.1214-1224 (2002).