Linux による USB デバイスドライバ作成と制御インタフェース開発

Development of a USB device driver working on Linux and Control Interface

福谷武司・小谷章二・高橋智一
Takeshi Fukutani, Shoji Kodani and Tomokazu Takahashi

近年、PC-UNIX（主として Linux）は、携帯電話をはじめとする組み込み分野用途に使われることが増えつつある。この組み込み用途の目的として、最近のネットワークインタフェースやＵＳＢ等のシリアルインタフェースという複雑なプロトコルを持つ通信手段に対して、Ｌｉｎｕｘの資産を流用しようというねらいが垣間見られる。サーバ技術が脚光を浴びることが多いＬｉｎｕｘではあるが、ここで改めてハードウェア制御ができるという側面に注目してみたい。ひいては、ＬｉｎｕｘＯＳにおけるＵＳＢドライバとはどういうものかを確認し、外部ハードウェアとの連携を如何にとるべきなのか、基礎事項を確認し、実際に作成した基本的なＵＳＢ対応のキャラクタ型デバイスドライバの作成方法と、その動作例について述べる。

Recently, it’s becoming more popular to utilize Linux for controlling hardware like a cellular phone. Integrated protocol is not easy to develop, so people tend to make use of a ready-to-use environment. Most of the time, Linux serves the role as a server such as a filesever, webserver and so on, but we focus on the aspect of hardware controlling instruments. We present the mechanism of a USB device driver in a Linux system, and pick up some points that should be taken into consideration.

1. はじめに

近年、多機能携帯電話やデジタル家電など、便利さを追求して、色々な製品が登場している。またさらに、それにも増して人間が使いやすいようなボタン操作や画面上の指示が必要であったり、パソコンとのデータ授受などで USB や IEEE1394 など多様なデータのやりとりに対応した製品開発へのニーズが高まってきている。こういった状況の下、効率良く製品の操作系インタフェースを実現するには、何らかのサブルーチン群を導入することが必要となってくる。

このサブルーチン群は、現在、有料および無料のものがあるが、有料のツールに関して言えば、その効果を開発前の段階で、判別することができないため、採用に踏み切るのはコスト的なリスクが伴う。
このサブルーチン群の動向としては、制御ＣＰＵそれぞれに合わせたアセンブラプログラムが従来使われてきたが、開発効率向上のニーズから、サブルーチンをどのＣＰＵに対してもＣ言語のプログラムを使い、共通化する考え方が定着してきた。このように基本的なハードウェアとのやりとりの仕組みを体系化することにより、使いやすくパッケージングしたＯＳのひとつである、Ｌｉｎｕｘにつてい具体的なプログラム例を取り上げ周辺装置とのデータ授受を行うＵＳＢデバイスのインターフェース開発を行ったので報告する。

2. 開発環境

今回の開発環境構成は、以下の通りである。

【開発マシン】

- PC
  (Celeron300MHz,ﾒﾓﾘ 256MB,HDD:12GB)
- OS: Vine Linux 3.2（カーネル 2.4.31）
- C コンパイラ：GCC 3.3.2

【対象 USB デバイス】

- USB コントローラ：USBN9604
  （ナショナル・セミコンダクター社）
- コントローラ操作マイコン：H8/3048

回路構成と装置外観を図１と図２に示す。

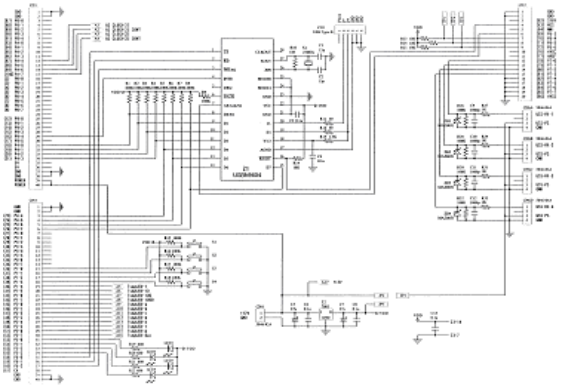

図１　回路構成

図２　装置外観

3.　Ｌｉｎｕｘにおける Ｉ／Ｏ制御

3.1　カーネル空間とユーザ空間

Linux においては、カーネル空間とユーザ空間が存在する。カーネル空間とは、ハードウェア管理に使われるアドレスを直接取り扱うことのできるモードで、外部機器とつながるＩ／Ｏは、このカーネル空間内でのやりとりによってのみ操作が可能である。カーネル空間は、ユーザプログラムが直接操作することはできないため、カーネル操作用の低レベル関数（read(),write(),ioctl()など）をデバイスドライバプログラムに記述して、カーネル空間に常駐させる。ユーザ空間からカーネル空間への操作は、派生した定義関数をユーザ空間のプログラムから呼び出すことで実行される。そのデバイスドライバ内の低レベル関数から派生した定義関数をユーザ空間のプログラムから呼び出すことが唯一のユーザ空間からカーネル空間への操作方法である。この際、ハードウェアは、デバイスファイル（別名：スペシャルファイル）という、I/O をファイルに見立てたものをあらかじめ作成及び登録（その方法は後述）することが必要である。これに対して、定義関数を通して読み書きを行うことで、I/O データの操作が可能となっている。この様子を図３に示す。

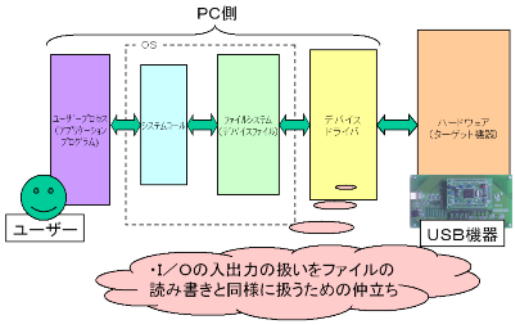

図３　デバイスドライバの役割

デバイスドライバは、キャラクタ型、ブロック型、ネットワーク型の３つの型があるが、ここではキャラクタ型のデバイスドライバを扱う。

3.2　Ｌｉｎｕｘデバイスドライバ

3.2.1　Linux デバイスドライバの様式

Ｌｉｎｕｘのデバイスドライバの基本形を図４に示す。

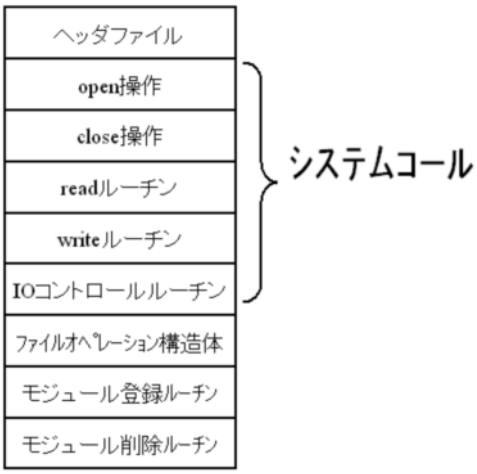

図４　デバイスドライバの基本形

基本的なＣ言語プログラムソースを次に示す。

```
#include <linux/errno.h>
#include <linux/fs.h>
#include <linux/module.h>
#include <linux/mm.h>
#include <asm/uaccess.h>
#define CASE1 1
#define CASE2 2

static unsigned int counter = 0;
static char string [132];
static int data;

static int skeleton_open (struct inode *inode,
struct file *file)
{
    MOD_INC_USE_COUNT;
    return 0;
}

static int skeleton_release (struct inode *inode,
struct file *file)
{
    MOD_DEC_USE_COUNT;
    return 0;
}


static ssize_t skeleton_read (struct file *file,
char *buf, size_t count, loff_t *ppos)
{
    int len, err;

    if( counter <= 0 )
            return 0;

    err = copy_to_user(buf,string,counter);
    if (err != 0)
            return -EFAULT;

    len = counter;
    counter = 0;

    return len;
}

static ssize_t skeleton_write (struct file *file,
const char *buf, size_t count, loff_t *ppos)
{
    int err;

    err = copy_from_user(string,buf,count);
    if (err != 0)
            return -EFAULT;

    counter += count;

    return count;
}

static int skeleton_ioctl(struct inode *inode,
struct file *file,
unsigned int cmd, unsigned long arg)
{
    int retval = 0;

    switch ( cmd ) {
      case CASE1:/* for writing data to arg */
         if (copy_from_user(&data, (int *) arg,
sizeof(int)))
            return -EFAULT;
          break;
       case CASE2:/* for reading data from arg */
       if (copy_to_user((int *) arg, &data,
sizeof(int)))
            return -EFAULT;
       break;
       default:
            retval = -EINVAL;
    }
return retval;
}

static struct file_operations skeleton_fops =
{
#if LINUX_VERSION_CODE < KERNEL_VERSION(2,3,0)
    NULL, /* skeleton_llseek */
    skeleton_read, /* skeleton_read */
    skeleton_write, /* skeleton_write */
    NULL, /* skeleton_readdir */
    NULL, /* skeleton_poll */
    skeleton_ioctl, /* skeleton_ioctl */
    NULL, /* skeleton_mmap */
    skeleton_open, /* skeleton_open */
    NULL, /* skeleton_flush */
    skeleton_release, /* skeleton_release */
    NULL, /* skeleton_fsync */
    NULL, /* skeleton_fasync */
    NULL, /* skeleton_check_media_change */
    NULL, /* skeleton_revalidate */
    NULL /* skeleton_lock */
    #else /* for LINUX_VERSION_CODE 2.4.0 and later
    */
    THIS_MODULE, /* struct module *owner;*/
    NULL, /* skeleton_llseek */
    skeleton_read, /* skeleton_read */
    skeleton_write, /* skeleton_write */
    NULL, /* skeleton_readdir */
    NULL, /* skeleton_poll */
    skeleton_ioctl, /* skeleton_ioctl */
    NULL, /* skeleton_mmap */
    skeleton_open, /* skeleton_open */
    NULL, /* skeleton_flush */
    skeleton_release, /* skeleton_release */
    NULL, /* skeleton_fsync */
    NULL, /* skeleton_fasync */
    NULL, /* skeleton_lock */
    NULL, /* skeleton_readv */
    NULL /* skeleton_writev */
    #endif
    };

    int init_module (void)
    {
        int i;

        i = register_chrdev (63, "test", & test_fops);
        if (i != 0) return - EIO;

        return 0;
    }

    void cleanup_module (void)
    {
        unregister_chrdev (63, "test");
    }
```

図５　デバイスドライバの基本プログラム

3.2.2　Linux デバイスドライバのコンパイルと組み込み

Ｌｉｎｕｘのデバイスドライバをコンパイルし、組み込むには次のように行う。まず、プログラムをＧＣＣ（GNUコンパイラコレクション。フリーのＣ言語コンパイラを含む様々な言語のコンパイラをセットにしたもの）にてコンパイルする。その際にオブジェクトファイル作成にとどめる。具体的には、シェル（＝コマンドライン）から次のようなコマンドを実行する。

```
$ gcc –Wall –O2 -c –D__KERNEL__ -DMODULE test.c
```

ここで、＄はユーザー権限でコマンドラインを入力する際の入力プロンプトを表している。gcc に続く文字列は、コンパイルオプションであるが、**-Wall** は、ワ

ーニング表示のオプション、**-O2** は最適化レベルをクラス２までにとどめておくためのオプション、**-c** は実行ファイルを作らずオブジェクトファイルまでを作成するオプション、**-D** はC言語の#define にあたる変数定義のオプションである。ここでは__KERNEL__という変数と MODULE という変数を定義したことを示す。作成したオブジェクトファイルはモジュールと呼ばれ、このモジュールをカーネル空間とユーザ空間の仲立ちとして組み込む作業を行う。Ｌｉｎｕｘのデバイスドライバはデバイスファイルで管理されており、既存のデバイスファイルを使用することもできるが、全く新規のデバイスとして登録することもできる。その際には次のようなコマンドを管理者権限にて入力し、デバイスファイルを作成する。

```
# mknod –m 444 /dev/test u 63 0
```

ここで＃は管理者権限で入力する際の入力プロンプトを表している。最後の２つの数字はＯＳで用意されているデバイス番号で１番目がメジャー番号、２番目がマイナー番号と呼ばれている。メジャー番号は既存のデバイスとの競合を避けて割り当てなければならないが、６０番から６３番は、実験用の番号として使うことが決められている。続いて、ドライバの組み込みは次のようなコマンドを入力する。

```
# insmod test.o
```

これにより、ドライバテーブルへの登録が行われる。これでデバイスドライバの組み込みが完了し、使用できる状態となった。逆に登録を解除する場合は、

```
# rmmod test.o
```

としてドライバテーブルへの登録を解除する。

3.2.3 Linux デバイスドライバの使用

図３－３のプログラムは次のようなシェルコマンドでその動作を確認することができる。

```
$ cat /dev/test
```

cat コマンドにより、標準出力への書き込みに対応するデバイスドライバの関数（図５内 skeleton_write 関数）が呼び出される。その結果、変数 counter がカウントアップされ標準出力（コンソール）に表示される。このプログラムだけではハードウェアにアクセスしているわけではないが、カーネル空間に属するデバイスファイルをソフトウェアで呼ぶ動作の実証になっている。これはつまり、デバイスファイルの先にハードウェアの処理を配置できるところを、あえて標準出力をターゲットとして、その基本動作を実行し、ＯＳ内部の不具合が無いことを確認するプログラムとして活用できる。

4. USB 用 Linux デバイスドライバ

4.1 USB デバイスドライバに必要な事前知識

ＵＳＢ用Ｌｉｎｕｘのデバイスドライバを作成するにあたり、ＵＳＢの内部的な動作やコントローラデバイスＵＳＢＮ９６０４について説明する。

4.1.1 エンドポイント・パイプ・インタフェース

エンドポイントとは、シリアル通信であるＵＳＢにおいて、パラレル通信のポートに似せた、一種の仮想ポートである。そのエンドポイントへのデータ出入口としての配線的役割をするものとしてパイプが存在する。エンドポイントはデバイス上では通し番号をあらかじめ割り振られている。この数はコントローラに設定を送ることで決まる。参考までに、今回使用したＵＳＢＮ９６０４のエンドポイントの数は最大で６である。パイプはエンドポイントよりも通常少ない本数を持っており、エンドポイントへのつなぎ換えのようなイメージで、データ入出力の切替を行う。最初から数が決まっているエンドポイントに対して、パイプは機能ベースで数が決まる。実際のデータ要求／転送は、パイプに対して行われる。このエンドポイントとパイプにおける機能ベースの数本のまとまりはインタフェースと呼ばれる。データのやりとりは、ホスト側から御用聞きのようにターゲット側になされる。これをポーリングという。デバイス側からこれを要求することはない。

4.1.2　ディスクリプタ

ディスクリプタとはターゲット機器の設定情報であり、ターゲット機器内のＵＳＢをコントロールするチップにファームウェアとして書き込まれている。（今回の場合はＨ８／３０４８）その構造は図６のような親子関係がある。エンドポイント、インタフェースに関しては、前節に説明した通りだが、さらに大きな括りとして、デバイスの構成としてのコンフィグレーションディスクリプタとデバイス全般の一般的なＩＤ情報を含むデバイスディスクリプタが存在する。これらの設定情報は、ＵＳＢ機器が端子に接続されたときに、デバイスドライバを呼び出す機能として必要であり、デバイス側のファームウェアに書き込まれている。ホストマシンはターゲットデバイスが接続したときに、これらの情報を受け取り、その後の通信を支障なく行うようにセットアップされる。

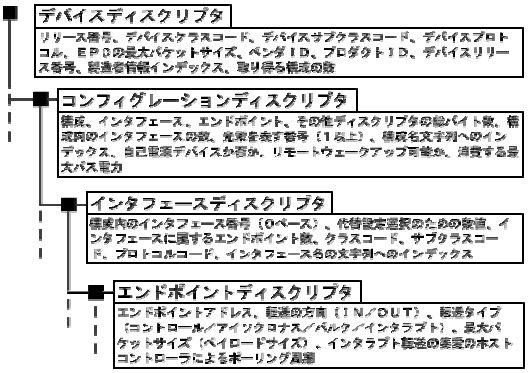

図６　ディスクリプタの体系

デバイスドライバにもこれらの情報をＰＣに取り込むためのルーチンが用意されている。

4.2　USB 用 Linux デバイスドライバの様式

ＵＳＢ用Ｌｉｎｕｘのデバイスドライバにおいても、通常のＬｉｎｕｘデバイスドライバと同様に、その様式を見ることにする。

図７にその基本形を示した。ＵＳＢに関するルーチンが追加されていることが見て取れる。

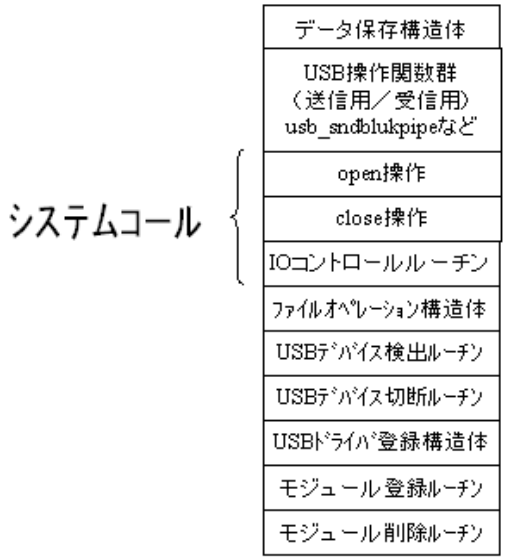

図６　ＵＳＢ用Ｌｉｎｕｘデバイスドライバの基本形

実際のソースは長くなるので抜粋とその役目を記載する。

```
    :
#define USBTEST_MINOR  32
    :
```

最初の方で、見られる define 文でマイナー番号の記載ができる。ＵＳＢデバイスドライバのメジャー番号は１８０であるが、これは決まっているのでデバイスドライバには書かれない。また、マイナー番号は１６の倍数にする決まりがある。

```
static int usbtest_set_led(usbtest_data_t* pdata,
int led)
{
    int r, rlen,i;
    unsigned long int pipe;
    unsigned char buf[2];

    if (pdata->dev == NULL) {
        return -1;
    }
    buf[0] = (unsigned char)(0x03);
    buf[1] = (unsigned char)(led & 0xff);

    pipe     =     usb_sndbulkpipe(pdata->dev,
pdata->ep_bulk_out);

    r = usb_bulk_msg(pdata->dev, pipe, buf, 2,
&rlen, HZ*10);

    if (r < 0) {
        return -1;
    }
    return 0;
}
```

このコードは、ＬＥＤを点灯するルーチンである。usb_sndbulkpipe()関数でパイプを作り、送信用として設定した上で、usb_bulk_msg()関数で指定したデバイ

スにデータを送る様子を表している。

```
static usbtest_data_t* g_usbtest_data;

static struct usb_device_id usbtest_table[] = {
    {USB_DEVICE(VID_USBTEST,PID_USBTEST)},
    {}
};

MODULE_DEVICE_TABLE (usb, usbtest_table);
```

デバイスディスクリプタと関連するルーチンである。ＶＩＤはベンダーＩＤ、ＰＩＤはプロダクトＩＤであり、これらの設定値をＰＣに取り込む様子を表している。

```
static int usbtest_ioctl(struct inode *inode,
struct file *file,
            unsigned int cmd, unsigned long arg)
{
    int led, dip;
    usbtest_data_t* pdata;


    int err = 0, tmp;
    int retval = 0;


    pdata = file->private_data;
    printk("file->private_data=%x¥n",file->pri
vate_data);
    printk("pdata->ep_bulk_out=%d¥n",pdata->ep
_bulk_out);

    switch(cmd) {
    case USBTEST_SET_LED:
        led = (int)arg;
        if (usbtest_set_led(pdata, led) < 0)
            return -EIO;
        break;

    case USBTEST_GET_DIPSW:
        if ((dip = usbtest_get_dipsw(pdata)) < 0)
            return -EIO;
        if    (copy_to_user((int*)arg,    &dip,
sizeof(led)))
            return -EFAULT;
        break;

    case USBTEST_WAIT_INTERRUPT:
        return usbtest_intr_wait(pdata);

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

実際のハードウェア動作は ioctl（アイオーコントロール＝入出力制御）ルーチンで行う。中程の switch 文で入力時の動作、出力時の動作を両方決めている。関数名を usbtest_ioctl()としているが、この名前は同じプログラム内に登場する、ファイルオペレーション構造体に登録することで役割をもたせることができる。

```
static struct file_operations usbtest_fileopes =
{
    owner:          THIS_MODULE,
    ioctl:          usbtest_ioctl,
    open:           usbtest_open,
    release:        usbtest_close,
};
```

コロン（：）を挟んで左にラベル、右に関数名の構成であり、ioctl のラベルと対応する関数に ioctl の役目をさせるという意味である。この書き方は、ＧＣＣの機能拡張によるものである。これ以外に、ドット（.）で始めるＣ９９形式と、単に決められた順番で並べる形式のものがある。

少々長くなるが、次のコードはＵＳＢデバイス検出ルーチンである。

```
static void* usbtest_probe(struct usb_device *dev, unsigned int ifnum,const struct
usb_device_id *id)
{
    struct usb_interface_descriptor *inf;
    struct usb_config_descriptor    *config;
    unsigned int pipe;
    int n, intr_interval;

    if ((dev->descriptor.idVendor  != VID_USBTEST) ||
        (dev->descriptor.idProduct != PID_USBTEST))
    {
        return NULL;
    }

    if (g_usbtest_data) {
        printk(KERN_INFO "USBTEST: no more probe.¥n");
        return NULL;
    }

    config = dev->actconfig;
    inf    = &config->interface[ifnum].altsetting[0];
    g_usbtest_data = kmalloc(sizeof(usbtest_data_t), GFP_KERNEL);
    if (g_usbtest_data == NULL) {
        printk(KERN_INFO "no memory¥n");
        return NULL;
    }

    intr_interval = 0;
    g_usbtest_data->ep_intr_in  = -1;
    g_usbtest_data->ep_bulk_out = -1;
    g_usbtest_data->ep_bulk_in  = -1;
    for(n = 0; n < inf->bNumEndpoints; n++) {
        switch(inf->endpoint[n].bEndpointAddress&USB_ENDPOINT_NUMBER_MASK) {
        case 1:
            if (! (inf->endpoint[n].bEndpointAddress&USB_ENDPOINT_DIR_MASK))
                break;
            if  ((inf->endpoint[n].bmAttributes&USB_ENDPOINT_XFERTYPE_MASK)  ==
USB_ENDPOINT_XFER_INT) {
                g_usbtest_data->ep_intr_in = 1;
                intr_interval = inf->endpoint[n].bInterval;
            }
            break;

        case 2:
            if (inf->endpoint[n].bEndpointAddress&USB_ENDPOINT_DIR_MASK)
                break;
            if    ((inf->endpoint[n].bmAttributes&USB_ENDPOINT_XFER_INT)    ==
USB_ENDPOINT_XFER_BULK) {
                g_usbtest_data->ep_bulk_out = 2;
            }
            break;

        case 5:
            if (! (inf->endpoint[n].bEndpointAddress&USB_ENDPOINT_DIR_MASK))
                break;
            if    ((inf->endpoint[n].bmAttributes&USB_ENDPOINT_XFER_INT)    ==
USB_ENDPOINT_XFER_BULK) {
                g_usbtest_data->ep_bulk_in = 5;
            }
            break;
        }
    }

    if (g_usbtest_data->ep_intr_in == -1) {
        printk("USBTEST:1: endpoint error!!¥n");
        return NULL;
    }
    if (g_usbtest_data->ep_bulk_out == -1) {
        printk("USBTEST:2: endpoint error!!¥n");
```

```
            return NULL;
        }
        if (g_usbtest_data->ep_bulk_in == -1) {
            printk("USBTEST:5: endpoint error!!¥n");
            return NULL;
        }
        g_usbtest_data->dev   = dev;
        g_usbtest_data->isopen = 0;

        pipe = usb_rcvintpipe(g_usbtest_data->dev, g_usbtest_data->ep_intr_in);
        FILL_INT_URB(&g_usbtest_data->irq, dev, pipe, g_usbtest_data->data, 2,
              usbtest_irq, g_usbtest_data, intr_interval);
        return g_usbtest_data;
    }
```

中程にあるcase文で各エンドポイントに対して入力用か出力用かなどの設定を行う。

次のコードはＵＳＢドライバ登録構造体である。さきほどのファイルオペレーション構造体に似ているが、ＵＳＢ用のドライバに特有の構造体である。

```
static struct usb_driver usbtest_data = {
    owner:  THIS_MODULE,
    name:   "usbtest",
    id_table:       usbtest_table,
    probe:  usbtest_probe,
    disconnect:     usbtest_disconnect,
    fops:           &usbtest_fileopes,
    minor:  USBTEST_MINOR,
};
```

ＵＳＢとして使えるように、デバイスドライバを登録するためには、このＵＳＢドライバ登録構造体へのポインタをusb_register()関数の引数として代入する。

```
static int __init usbtest_init(void)
{
    if (usb_register(&usbtest_data) < 0) {
        return -1;
    }
    return 0;
}
```

5. まとめ

このように、ＵＳＢの内部動作とデバイスドライバの構造についての理解を通じてＵＳＢを使ったＩ／Ｏ制御をＬｉｎｕｘを利用して実施した。今回はターゲットデバイス制御用のドライバであるが、ホスト側についても組み込みＬｉｎｕｘ基板にＵＳＢホスト機能が備わるなど、対応機器が増えつつあるのでこの動きと呼応して、開発手法の取得および普及に努めたい。そして、シンプルで使い勝手の良いものを提供するために、開発側としての技術的課題を乗り越えていきたい。今後は、ＵＳＢカメラのＬｉｎｕｘ制御を用いた、画像処理と制御の連携や、組み込みＬｉｎｕｘを使った制御系の小型化等につなげていきたい。

文 献

1)Jonathan Corbet 他：ＬＩＮＵＸデバイスドライバ第３版,株式会社オライリージャパン,(2007)

2)平田　豊：Ｌｉｎｕｘデバイスドライバプログラミング,ソフトバンククリエイティブ株式会社,(2008)

3)インターフェース，ＣＱ出版社,(3),(2000)、(1),(2001)

4)インターフェース編集部編：改訂新版ＵＳＢハード&ソフト開発のすべて,ＣＱ出版社,(2007)