デバドラ開発不要!
アプリケーションからのペリフェラル制御

Yoichi Yuasa

OSAKA NDS Embedded Linux Cross Forum #6

自己紹介

- 湯浅陽一
- 1999年よりLinux kernel開発に参加
- MIPSアーキテクチャのいくつかのCPUへLinux kernelを移植

Linuxにおける
一般的なペリフェラル操作

- デバイスドライバを作成
- データの送受信やデバイス制御はデバイスドライバ
- アプリケーションはデバイスドライバが取り出したデータ部分のみ扱う
- アプリケーションからのペリフェラル制御は抽象化
- アプリケーションからの直接的操作はほぼできない

デバイスドライバのメリット

- 共通機能を利用できる(非常に大きなメリット)
  - デバイスに依存した処理のみ記述すれば後は共通機能が対応
- ソースコードが多く公開されているので再利用可能
- 既存デバイスドライバに似たデバイスがあれば実装は非常に簡単

デバイスドライバ開発時の問題点

- 情報が少ない
- Linux kernelにドキュメントも含まれているが
  - 内容が古いことがある
  - 内容が十分ではない
- 結局ソースコードを見るしかない
  - 似たデバイスドライバのソースコード
  - デバイスドライバが利用している関数のソースコード
- マイナーなデバイスドライバは書ける人が非常に少ない

デバイスドライバ開発時の問題点

- デバイスドライバを開発するほどのものではないけど必要になる場合がある
  - 一括で設定を行うだけ
  - 単にレジスタを定期的に読み出したいだけ

アプリケーションから直接ペリフェラル制御

- 処理を中継するデバイスドライバは存在する
  - Linux kernelに含まれている
- デバイスドライバは必要最小限の処理でkernel内部処理を呼び出すよう作られている
- デバイスドライバは抽象的ではない直接的なアクセスを提供
- インターフェースが固定されている(kernel内部のようにすぐに変更されない)
- 利用にはデバイスドライバを組み込む必要がある

GPIO /sys/class/gpio/…
(sysfs interface)

- sysfsからファイルアクセスとしてGPIOを操作
- 1ピン毎に制御
  - 制御可能設定
  - 入出力設定
  - 出力値設定
  - 入力値確認

GPIOデバイスドライバの有効化

- /sys/class/gpio/... (sysfs interface)有効化
- ボードの対応するGPIOドライバを有効化
- /sysをmount(通常はmountされている)
  - sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)

GPIO設定と操作

```
# cd /sys/class/gpio
# ls
export      gpiochip360  gpiochip418  gpiochip452  gpiochip496
gpiochip356  gpiochip392  gpiochip436  gpiochip467  unexport
# cd gpiochip360
# ls
base  device  label  ngpio  power  subsystem  uevent
# cat label
e6055400.gpio
# cat base
360
# cat ngpio
32
```

GPIOコントローラーのベースアドレスが区別のためにラベル設定されている

先頭のGPIOピン番号

このGPIOコントローラーが持つGPIOのピン数

GPIO設定と操作

```
# echo 384 > export   384ピンを有効化
# ls
export   gpiochip356  gpiochip392  gpiochip436  gpiochip467  unexport
gpio384  gpiochip360  gpiochip418  gpiochip452  gpiochip496
# cd gpio384
# ls
active_low  device  direction  edge  power  subsystem  uevent  value
# cat direction 入出力設定
in
# echo out > direction 出力に設定
# cat direction
out
# cat value 設定値
0
# echo 1 > value
# cat value
1
```

GPIOまとめ

- ファイルへの操作がそのままGPIOレジスタのkernel内部設定に繋がっている
  - echo 1 > valueはgpiod_set_value_cansleep()
  - cat directionはgpiod_get_direction()
- アプリケーションから直接的にGPIOが制御できる

I2C device interface

- I2C Busコントローラードライバに直接データを渡すioctlインターフェイスがある
- I2Cの場合2種類の方法がある
  - ioctl(I2C_SLAVE)とread()/write()の組み合わせ
  - ioctl(I2C_RDWR)
- ioctl(I2C_RDWR)はI2Cバス上に流すデータを直接的に設定できる

I2Cバス設定

- I2C device interfaceを有効化
- ボードの対応するI2Cホストコントローラードライバを有効化
- ボード上のデバイスファイル確認

  # ls /dev/i2c*

  /dev/i2c-2 /dev/i2c-4 /dev/i2c-7 /dev/i2c-8
- デバイスファイルのopen

  int fd;

  fd = open(“/dev/i2c-4”, O_RDWR);

I2C書き込み

- 7bitアドレス、I2Cデバイスのレジスタアドレスは16bit、データ8bit

```
struct i2c_msg msg[1];                  /* /usr/include/linux/i2c.h */
struct i2c_rdwr_ioctl_data packets;     /* /usr/include/linux/i2c-dev.h */
unsigned char data[3];
int ret;

msg[0].addr = 7bitアドレス; /* addrは16bit幅 */
msg[0].flags = 0;               /* read、writeやアドレス長の指定に利用 */
msg[0].len = 3;                 /* bufに指定するdataのサイズ */
msg[0].buf = data;
```

I2C書き込み

data[0] = レジスタアドレス上位8bit;

data[1] = レジスタアドレス下位8bit;

data[2] = 書き込み値;

```
packets.msgs = msg;
packets.nmsgs = 1; /* msgのサイズ指定 */
ret = ioctl(fd, I2C_RDWR, &packets);
```

I2C読み出し

- 7bitアドレス、I2Cデバイスのレジスタアドレスは16bit、データ8bit

```
struct i2c_msg msg[2];
struct i2c_rdwr_ioctl_data packets;
unsigned char data1[2], data2;
int ret;

msg[0].addr = 7bitアドレス;
msg[0].flags = 0;                /* 最初はアドレスを書き込み */
msg[0].len = 2;
msg[0].buf = data1;
```

I2C読み出し

```
data1[0] = レジスタアドレス上位8bit;
data1[1] = レジスタアドレス下位8bit;

msg[1].addr = 7bitアドレス;
msg[1].flags = I2C_M_RD; /* リード時に設定 */
msg[1].len = 1;                  /* データを1byte読み出し */
msg[1].buf = &data2;          /* data2に読み出しデータが入る */

packets.msgs = msg;
packets.nmsgs = 2;           /* アドレス書き込みとデータ読み出しでmsgは2 */
ret = ioctl(fd, I2C_RDWR, &packets);
```

I2Cまとめ

- ioctl(I2C_RDWR)がそのままkernel内部でi2c_transfer()を呼び出している
- i2c_transfer()はデバイスドライバが利用する関数そのまま
- アプリケーションからも直接的I2Cバスにアクセスできる
- SPI(Serial Peripheral Interface)にも似た仕組みがある

Userspace I/O drivers

- デバイスへのアクセスや割り込み利用のための共通の仕組み
- デバイス特有部分は個別のデバイスドライバを組み合わせる
  - PCI 2.3 & PCI Express
  - Memory mapped I/Oなど
- /dev/memとは違いそのデバイスのみにアクセス可能になる
- アクセスはsysfsと/dev/uio*の組み合わせ

Userspace I/O drivers

- デバイスツリーで指定したアドレスエリアにアプリケーションからアクセスできる
- 割り込みをアプリケーションで検出できる
- ほぼすべてをアプリケーションから制御するためこれまでと比べてアプリケーション規模が大きくなる

参考

- Documentation/gpio/sysfs.txt
- Documentation/i2c/dev-interface
- Documentation/spi/spidev
- https://www.kernel.org/doc/html/v4.15/driver-api/uio-howto.html