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2

Yoichi Yuasa

OSAKA NDS Embedded Linux Cross Forum #7

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制御

- 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デバイス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
  - 割り込みのタイミングでレジスタを読み出したいだけ
  - メモリエリアをコピーするだけ

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にすでに含まれているものが利用できる
- 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

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

Userspace I/O drivers

- アプリケーションからデバイスへのアクセスや割り込みを利用する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
- 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
- デバイス操作で問題が起きてもkernelがクラッシュしない
- ほぼすべてをアプリケーションから制御することになるためデバイスドライバを利用した場合と比べてアプリケーション規模が大きくなる

Userspace I/Oはどのような場合に利用すればよいのか?

- メモリアクセスができればデバイス制御できる場合
- 割り込みをアプリケーションで利用したい場合
- デバイスがLinux kernelが持っているサブシステムに当てはまらない場合

Userspace I/O driversの有効化

- Linux kernelコンフィグレーションを有効化
  - Device Drivers
    - Userspace I/O drivers
      - Userspace I/O platform driver with generic IRQ handling
      - Userspace platform driver with generic irq and dynamic memory
      - Generic driver for PCI 2.3 and PCI Express cards
- /sysをmount(通常はmountされている)

  $ mount

  sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)

Userspace I/O platform driver with generic IRQ handling device tree

- compatibleはmodule paramで自由に設定

uio_pdrv_genirq.cuio_pdrv_genirq.c内の記述

```
static struct of_device_id uio_of_genirq_match[] = {
        { /* This is filled with module_parm */ },
        { /* Sentinel */ },
};
MODULE_DEVICE_TABLE(of, uio_of_genirq_match);
module_param_string(of_id, uio_of_genirq_match[0].compatible,
128, 0);
MODULE_PARM_DESC(of_id, "Openfirmware id of the device to be
handled by uio");
```

Userspace platform driver with generic irq and dynamic memory device tree

- module paramの仕組みがない

```
static const struct of_device_id uio_of_genirq_match[] = {
        { /* empty for now */ },
};
MODULE_DEVICE_TABLE(of, uio_of_genirq_match);
```

- ソースコードに追記してからビルドする必要がある

Device tree設定例

- アドレス先頭が0xfe860000サイズが0x2000
- 割り込みはGIC共有ペリフェラル割り込みの192番でHIGHレベルトリガー

```
#address-cells = <2>;
#size-cells = <2>;
uio@fe860000 {
    compatible = "generic,uio-pdrv";
    reg = <0 0xfe860000 0 0x2000>;
    interrupts = <GIC_SPI 192 IRQ_TYPE_LEVEL_HIGH>;
};
```

uio_pdrv_genirqモジュールのロード

- modprobeの引数of_id=の値とdevice treeのcompatible=の値を一致させる

  # modprobe uio_pdrv_genirq of_id="generic,uio-pdrv"

- デバイスドライバをstaticにリンクしている場合はKernel command line設定

  uio_pdrv_genirq.of_id="generic,uio-pdrv"

sysfsインタフェース

```
$ ls -R /sys/class/uio/uio0/
/sys/class/uio/uio0/:
dev        event      name       subsystem  version
device     maps       power      uevent

/sys/class/uio/uio0/maps:
map0

/sys/class/uio/uio0/maps/map0:
addr    name    offset  size

/sys/class/uio/uio0/power:
autosuspend_delay_ms    runtime_active_time     runtime_suspended_time
control                 runtime_status
```

sysfsインタフェース

```
$ cat /sys/class/uio/uio0/maps/map0/addr
0x00000000fe860000 (先頭アドレス)
$ cat /sys/class/uio/uio0/maps/map0/name
/soc/uio@fe860000 (デバイス名)
$ cat /sys/class/uio/uio0/maps/map0/offset
0x0 (オフセット)
$ cat /sys/class/uio/uio0/maps/map0/size
0x0000000000002000 (サイズ)
$ /sys/class/uio/uio0/dev
246:0 (デバイス番号)
$ cat /sys/class/uio/uio0/event
0 (割り込み数)
$ /sys/class/uio/uio0/version
devicetree (Device treeで設定した場合)
```

Userspace I/Oのdevインタフェース

- /dev/uio0, /dev/uio1, /dev/uio2, …

```
$ ls -l /dev/uio*
crw-------    1 root     root      246,   0 Jun 19 12:20 /dev/uio0
```

- キャラクタデバイス
- open()、close()、read()、write()、mmap()、select()などが利用できる

Userspace I/Oのread()

- 割り込み待ちとして機能
- 読み出しサイズは32bit intのみ
- ブロッキングI/O処理でread()を呼び出すと割り込み発生待ちとなる
- 割り込みが発生するとすぐにread()処理から戻る
- select()で複数Userspace I/Oの割り込みを待つことも可能

Userspace I/Oのwrite()

- 割り込み制御機能
- 書き込めるサイズは32bit intのみ
- 0を書き込むと割り込み禁止
- 0以外を書き込むと割り込み許可

Userspace I/Oのmmap()

- device treeで設定した領域をアプリケーションからアクセスできるように設定
- 設定例にあったdevice treeの場合は以下のようにmmap()する

```
void *map;
map = mmap(NULL, 0x2000UL, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
if (map == MAP_FAILED) {
    <エラー処理>
}
<mapにread/writeアクセス>
...
munmap(map, 0x2000UL);
```

Userspace I/Oで2つの領域をアクセス可能にする場合

- device tree

```
uio@fe860000 {
    compatible = "generic,uio-pdrv";
    reg = <0 0xfe860000 0 0x2000>, <0 0xfe870000 0 0x2000>;
}
```

- mmap()は最後の引数offsetで領域を選択

```
map0 = mmap(NULL, 0x2000UL, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
map1 = mmap(NULL, 0x2000UL, PROT_READ|PROT_WRITE, MAP_SHARED, fd,
                                                  1 * getpagesize());
```

2つの領域をmmap()した結果

```
# ./a.out
[71.786219] uio uio0: map0 addr 0x00000000fe860000 mapping
map0: 0x0xffff92a99000
[73.792011] uio uio0: map1 addr 0x00000000fe870000 mapping
map1: 0x0xffff92a97000
```

Userspace I/O driversまとめ

- 特殊なデバイス(Linux kernelのサポートが無い)向けにデバイスドライバを開発するときはUserspace I/O driverとアプリケーションによるペリフェラル制御も検討の余地がある
- kernelをバイパスして直接ペリフェラル制御したい場合にも有効

参考

- Documentation/gpio/sysfs.txt
- Documentation/i2c/dev-interface
- Documentation/driver-api/uio-howto.rst
- https://www.kernel.org/doc/html/v4.17/driver-api/uio-howto.html