**新人情報共有資料**

**2022/04/27**

**Rev3.0**

|  |  |  |
| --- | --- | --- |
| 日付 | 修正者 | 修正内容 |
| 2013 | 羽野/藤井/松木/清水 | 新規作成 |
| 2015/05/17 | 高倉 | init.rcの章を追加 |
| 2022/mm/dd | 庭 | 全面的に見直し、不足な情報を追記 |

# 目次

[1 目次 1](#_Toc101872390)

[1 デバイス・ドライバ 5](#_Toc101872391)

[1.1 概要 5](#_Toc101872392)

[1.2 詳細 5](#_Toc101872393)

[1.2.1 プラットフォームデバイスの登録方法(デバイスツリー) 5](#_Toc101872394)

[1.2.2 プラットフォームデバイスの登録方法(レガシータイプ) 6](#_Toc101872395)

[2 ドライバの登録方法 9](#_Toc101872396)

[2.1 構造体 9](#_Toc101872397)

[2.2 関数 9](#_Toc101872398)

[2.3 実装例 10](#_Toc101872399)

[2.4 デバイスとドライバを登録するタイミング 11](#_Toc101872400)

[2.4.1 登録順の変更 11](#_Toc101872401)

[3 デバイスツリー 12](#_Toc101872402)

[3.1 プラットフォームデバイスの登録方法(デバイスツリー) 12](#_Toc101872403)

[3.2 ドライバでのプロパティ取得方法 18](#_Toc101872404)

[3.2.1 ヘッダファイル 18](#_Toc101872405)

[3.2.2 I/F 18](#_Toc101872406)

[4 モジュールパラメータ 21](#_Toc101872407)

[4.1 概要 21](#_Toc101872408)

[4.2 詳細 22](#_Toc101872409)

[4.2.1 使用方法 22](#_Toc101872410)

[4.2.2 使用例 23](#_Toc101872411)

[4.3 応用的な使用方法 23](#_Toc101872412)

[4.3.1 使用方法 24](#_Toc101872413)

[4.3.2 使用例 25](#_Toc101872414)

[5 ログ 27](#_Toc101872415)

[5.1 概要 27](#_Toc101872416)

[5.2 printk 27](#_Toc101872417)

[5.2.1 書式 27](#_Toc101872418)

[5.2.2 LOG\_LEVELの一覧 27](#_Toc101872419)

[5.2.3 書式指定子 28](#_Toc101872420)

[5.3 dump\_stack 28](#_Toc101872421)

[5.3.1 書式 28](#_Toc101872422)

[5.4 dmesg 29](#_Toc101872423)

[5.4.1 書式 29](#_Toc101872424)

[6 マルチプロセッサ手法について 30](#_Toc101872425)

[6.1 概要 30](#_Toc101872426)

[6.2 SMP(Symmetric Multi Processing) 30](#_Toc101872427)

[6.2.1 概要 30](#_Toc101872428)

[6.2.2 書式 30](#_Toc101872429)

[6.2.3 SMPの注意点 32](#_Toc101872430)

[6.2.4 関数の使用例 33](#_Toc101872431)

[6.2.5 シーケンス図（調査中） 35](#_Toc101872432)

[6.3 プリエンティブ 35](#_Toc101872433)

[7 パーミッション 36](#_Toc101872434)

[7.1 概要 36](#_Toc101872435)

[7.2 パーミッションの種類 36](#_Toc101872436)

[8 サスペンド機能 37](#_Toc101872437)

[8.1 サスペンド・レジューム 37](#_Toc101872438)

[8.1.1 Linuxにおけるサスペンド機能 37](#_Toc101872439)

[8.1.2 実装例 37](#_Toc101872440)

[8.2 サスペンド抑止 39](#_Toc101872441)

[8.2.1 wakelock 39](#_Toc101872442)

[8.2.1 pm\_wakeup 41](#_Toc101872443)

[8.3 wakelockの確認 42](#_Toc101872444)

[9 Runtime PM 43](#_Toc101872445)

[10 ファイルシステム 44](#_Toc101872446)

[10.1 概要 44](#_Toc101872447)

[10.1.1 ファイルシステム 44](#_Toc101872448)

[10.1.2 疑似ファイルシステム 44](#_Toc101872449)

[10.2 debugfs 45](#_Toc101872450)

[10.2.1 概要 45](#_Toc101872451)

[10.2.2 実装方法 45](#_Toc101872452)

[10.2.3 実装例 48](#_Toc101872453)

[10.3 procfs 49](#_Toc101872454)

[10.3.1 概要 49](#_Toc101872455)

[10.3.2 詳細 49](#_Toc101872456)

[10.3.3 実装 51](#_Toc101872457)

[10.3.4 実装例 54](#_Toc101872458)

[10.4 sysfs 56](#_Toc101872459)

[10.4.1 概要 56](#_Toc101872460)

[10.4.2 詳細 56](#_Toc101872461)

[11 GPIO 57](#_Toc101872462)

[11.1 概要 57](#_Toc101872463)

[11.2 詳細 58](#_Toc101872464)

[11.3 GPIOライブラリ 59](#_Toc101872465)

[11.3.1 使用方法 59](#_Toc101872466)

[11.4 pinctrl 61](#_Toc101872467)

[12 タイマ 62](#_Toc101872468)

[12.1 タイマ処理の役割 62](#_Toc101872469)

[12.1.1 時刻処理 62](#_Toc101872470)

[12.1.2 時限処理 62](#_Toc101872471)

[12.2 タイマーリスト 62](#_Toc101872472)

[12.2.1 timer\_list構造体 63](#_Toc101872473)

[12.2.2 タイマ処理に関係する関数 64](#_Toc101872474)

[13 遅延処理 69](#_Toc101872475)

[13.1 概要 69](#_Toc101872476)

[13.2 タスクレット 69](#_Toc101872477)

[13.2.1 tasklet\_struct構造体 69](#_Toc101872478)

[13.2.2 タスクレット実装の流れ 70](#_Toc101872479)

[13.3 ワークキュー 71](#_Toc101872480)

[13.3.1 概要 71](#_Toc101872481)

[13.3.2 ワークキュー実装の流れ 71](#_Toc101872482)

[13.3.3 ワークキューに関係する関数 72](#_Toc101872483)

[14 排他制御 75](#_Toc101872484)

[14.1 Spinlock 76](#_Toc101872485)

[14.2 Semaphore 77](#_Toc101872486)

[14.3 Mutex 78](#_Toc101872487)

[15 メモリ管理 79](#_Toc101872488)

[15.1 ページング 79](#_Toc101872489)

[15.2 buddyシステム 79](#_Toc101872490)

[15.2.1 概要 79](#_Toc101872491)

[**15.2.2** I/F 79](#_Toc101872492)

[15.2.3 procfs 80](#_Toc101872493)

[15.3 slabアロケータ 81](#_Toc101872494)

[15.3.1 概要 81](#_Toc101872495)

[15.3.2 I/F 81](#_Toc101872496)

[15.3.3 procfs 82](#_Toc101872497)

[15.4 vmalloc 83](#_Toc101872498)

[15.4.1 概要 83](#_Toc101872499)

[15.4.2 I/F 83](#_Toc101872500)

[15.4.3 procfs 83](#_Toc101872501)

[15.5 空きメモリ回収 84](#_Toc101872502)

[15.5.1 kswapd 84](#_Toc101872503)

[15.5.2 direct reclaim 84](#_Toc101872504)

[15.5.3 OOM 84](#_Toc101872505)

[15.5.4 Low Memory Killer 84](#_Toc101872506)

[16 init.rc 85](#_Toc101872507)

[16.1 概要 85](#_Toc101872508)

[16.2 アクションについて 85](#_Toc101872509)

[16.3 サービスについて 85](#_Toc101872510)

[16.4 詳細 86](#_Toc101872511)

[16.4.1 アクションの記述方法 86](#_Toc101872512)

[**16.4.2** サービスの記述方法 87](#_Toc101872513)

# デバイス・ドライバ

## 概要

デバイスとは、CPU・メモリなどのコンピュータ内部の装置や、マウス・キーボードなどの周辺機器のこと。

ドライバとは、デバイスを制御・操作するためのソフトウェア。

様々な種類のデバイス・ドライバが存在するが、本書では現場で一番使用されているプラットフォームデバイス・ドライバを使用した方法を記載する。

## 詳細

### プラットフォームデバイスの登録方法(デバイスツリー)

現在、デバイスの登録はデバイスツリーを使用して行われる。

記述方法は[デバイスツリー](#_デバイスツリー)の章を参照。

### プラットフォームデバイスの登録方法(レガシータイプ)

**本章の方法は古いため、特別な理由がない限り使用しないこと。**

* 説明

platform\_device構造体を定義し、platform\_device\_registerを呼び出す。

複数のデバイスを登録したい場合は、platform\_deviceの配列を作成する。

そして、定義した構造体を引数にして、関数platform\_add\_devicesを呼び出す。

* 構造体

1. platform\_device

|  |  |  |
| --- | --- | --- |
| **構造体名** | platform\_device | |
| **メンバ** | 定義型 | 概要 |
| const char \*name | デバイスの名前 |
| int id | デバイスを識別するID |
| struct device dev | デバイス情報 |
| u32 num\_resource | resource構造体のサイズ |
| struct resource \*resource | リソース情報 |
| struct platform\_device\_id \*id\_entry | デバイスとドライバを一致させるID |
| struct pdev\_archdata archdata | アーキテクチャ追加情報 |
| **ファイルパス** | kernel\include\linux\platform\_device.h | |

1. resource

|  |  |  |
| --- | --- | --- |
| **構造体名** | resource | |
| **メンバ** | 定義型 | 概要 |
| resource\_size\_t start | リソース開始アドレス |
| resource\_size\_t end | リソース終了アドレス |
| const char \*name | リソース名 |
| unsigned long flags | 各種フラグ（詳細は③を参照） |
| struct resource \*parent | 親ノード |
| struct resource \*sibling | （調査中） |
| struct resource \*child | 子ノード |
| **ファイルパス** | kernel\include\linux\ioport.h | |
| **説明** | Linuxにおけるデバイス・ドライバはツリー状に管理する。原則として子ノードのアドレスは親ノードのアドレスの範囲に収まらなければならない。 | |

1. 各種フラグ（調査中）

リソースのタイプを指定できる。

|  |  |  |  |
| --- | --- | --- | --- |
| **ファイルパス** | kernel\include\linux\ioport.h | | |
| **マクロ名** | | **定義** | **説明** |
| IORESOURCE\_IO | | 0x00000100 | 入出力ポート |
| IORESOURCE\_MEM | | 0x00000200 | メモリ領域 |
| IORESOURCE\_IRQ | | 0x00000400 |  |
| IORESOURCE\_DMA | | 0x00000800 |  |
| IORESOURCE\_BUS | | 0x00001000 |  |

* 関数

|  |  |  |
| --- | --- | --- |
| **関数名** | int platform\_device\_register | |
| **引数** | struct platform\_device \*pdev | 登録するデバイス |
| **戻り値** | 0 | 登録成功 |
| 1 | 登録失敗 |
| **ファイルパス** | kernel\drivers\base\platform.c | |
| **説明** | デバイスをカーネルに登録する。  device\_initializeを呼び出し、platform\_device構造体の一部を初期化する。platform\_device\_addを呼び出し、デバイスを追加する。 | |

※platform\_deviceとplatform\_driveはnameに同じ引数を渡すことで関連付けられる。

|  |  |  |
| --- | --- | --- |
| **関数名** | int platform\_add\_devices | |
| **引数** | struct platform\_device \*\*devs | 登録する構造体配列の先頭アドレス |
| int num | 登録するデバイスの数 |
| **戻り値** | 0 | 登録成功 |
| 1 | 登録失敗 |
| **ファイルパス** | kernel\drivers\base\platform.c | |
| **説明** | 複数のデバイスを同時に登録する。  １つでも登録に失敗するとエラーとなり、 それまでに登録したデバイスは登録解除する、 | |

* 実装例

ベースアドレスやデバイス名は、デバイスの設計時に決めておく。

#define TEST\_DEVICE\_BASE\_ADDR 0x00220000

#define TEST\_RESOURCE\_SIZE 0x00000200

static struct resource test\_device\_resources[] = {

{

.start = TEST\_DEVICE\_BASE\_ADDR,

.end = TEST\_DEVICE\_BASE\_ADDR + TEST\_RESOURCE\_SIZE,

.flags = IORESOURCE\_MEM,

},

};

struct platform\_device test\_device = {

.name = "test\_device",

.id = -1,

.num\_resources = ARRAY\_SIZE(test\_device\_resources),

.resource = test\_device\_resources,

};

static void \_\_init test\_init\_machine(void)

{

platform\_device\_register(&test\_device);

}

# ドライバの登録方法

platform\_driver構造体を定義し、それをplatform\_driver\_registerに渡して登録する。

platform\_driver構造体には、デバイスとドライバを一致させる情報と、デバイス登録、デバイス削除、サスペンド、シャットダウンの際に呼び出す関数を登録する。

## 構造体

|  |  |  |  |
| --- | --- | --- | --- |
| **構造体名** | platform\_driver | | |
| **メンバ** | int (\*probe)(struct platform\_device \*) | | probe時に呼び出す関数 |
| int (\*remove)(struct platform\_device \*) | | remove時に呼び出す関数 |
| void (\*shutdown)(platform\_device \*) | | shutdown時に呼び出す関数 |
| int (\*suspend)(struct platform\_device \*,  pm\_message\_t state) | | suspend時に呼び出す関数 |
| int (\*resume) (struct platform\_device \*) | | resume時に呼び出す関数 |
| struct device\_driver driver | .name | デバイスとドライバの情報を格納 |
|  | .owner | THIS\_MODULEを設定する |
|  | .of\_match\_table | デバイスツリーに記述したデバイスと対応されるための情報 |
|  | |  |
| **ファイルパス** | kernel\include\linux\platform\_device.h | | |

## 関数

|  |  |  |
| --- | --- | --- |
| **関数名** | int platform\_driver\_register | |
| **戻り値** | 0 : 成功  1 : 失敗 | |
| **IN/OUT** | struct platform\_driver \*drv | 登録するドライバ |
| **ファイルパス** | kernel\drivers\base\platform.c | |
| **説明** | 複数のデバイスを同時に登録する。  １つでも登録に失敗するとエラーとなり、 それまでに登録したデバイスは登録解除する、 | |

## 実装例

module\_initマクロで登録した関数内でドライバの登録を行う。

デバイスツリーに登録したデバイス情報と紐づけるための情報を「.driver.of\_match\_table」に追加する。

|  |
| --- |
| static const struct of\_device\_id test\_dt\_match[] = {  デバイスツリーのcompatibleプロパティと一致する文字列とする  {.compatible = "hisho,Testdrv"},  NULL終端を必ず入れること  **{},**  };  MODULE\_DEVICE\_TABLE(of, test\_dt\_match); |

static struct platform\_driver test\_platform\_driver = {

.driver = {

.name = "Testdrv",

.owner = THIS\_MODULE,

.of\_match\_table = test\_dt\_match,

},

.probe = test\_probe,

.remove = test\_remove,

.suspend = test\_suspend,

.resume = test\_resume,

};

static int \_\_init test\_driver\_init (void)

{

return platform\_driver\_register(&test\_platform\_driver);

}

static void \_\_exit test\_driver\_exit(void)

{

platform\_driver\_unregister(&test\_platform\_driver);

}

module\_init(test\_driver\_init);

module\_exit(test\_driver\_exit);

## デバイスとドライバを登録するタイミング

静的リンクの場合、カーネルイメージ(vmlinux)の一部に組み込まれてカーネル起動時にロードされる。

動的リンクの場合、ローダブル・カーネル・モジュール(LKM)と呼び、拡張子 \*.ko のファイルとして実装される。実行する際は、ファイルシステムの何処かに置き、必要なときにロードする形態を取る。

ロードはinsmodやmodprobeコマンドで行う。

ロードされているモジュールを確認するには、lsmodコマンドを使う。

### 登録順の変更

デバイスには、特定のデバイスドライバよりも後に登録しないと動作しない物がある。

init関数を登録する為のマクロは、module\_init以外にも複数ある。これらにはレベルがあり、レベルの小さい順から先に呼ばれる。マクロを使い分ける事でinit関数が呼ばれる順序を変更する事が出来る。

同じレベル内でのデバイスドライバの登録順番は、makeのリンクされる順番で決められる。リンク順を変えるにはMakefileを編集する。

* + - * 1. init関数を登録するマクロの一覧

|  |  |
| --- | --- |
| マクロ名 | レベル |
| early\_initcall | early |
| pure\_initcall | 0 |
| core\_initcall | 1 |
| core\_initcall\_sync | 1s |
| postcore\_initcall | 2 |
| postcore\_initcall\_sync | 2s |
| arch\_initcall | 3 |
| arch\_initcall\_sync | 3s |
| subsys\_initcall | 4 |
| subsys\_initcall\_sync | 4s |
| fs\_initcall | 5 |
| fs\_initcall\_sync | 5s |
| rootfs\_initcall | rootfs |
| device\_initcall / module\_init | 6 |
| device\_initcall\_sync | 6s |
| late\_initcall | 7 |
| late\_initcall\_sync | 7s |

# デバイスツリー

## プラットフォームデバイスの登録方法(デバイスツリー)

現在、デバイスの登録はデバイスツリーを使用して行われる。

デバイスツリーは、カーネルの共有化を図るために導入された仕組みである。

ターゲットボード毎に異なるデバイスやパラメータの差異を吸収して、複数のターゲットボードで

同一のカーネルを使用できるようにすることを目的として導入された。

ツリー構造でデバイス情報を定義し、ノードとプロパティからなるフォーマットに従いソースコード(DTS/DTSI)で記述する。

DTS：Device Tree Source

ソースコードはDevice Tree Compiler(DTC)でビルドされてDTB(Device Tree Blob)ファイルが生成される。

公式文書

<https://elinux.org/Device_Tree_Usage>

* レイアウト

DTSファイルのレイアウトを下記に示す。

|  |
| --- |
| /dts-v1/;  /include/ "ファイル名"  /{  [プロパティ定義]  [ラベル:] ノード名[@装置アドレス] {  [プロパティ定義]  [子ノード]  };  }; |

|  |
| --- |
| DTSの記述例  / {  node1 {  a-string-property = "A string";  a-string-list-property = "first string", "second string";  a-byte-data-property = [0x01 0x23 0x34 0x56];  child-node1 {  first-child-property;  second-child-property = <1>;  a-string-property = "Hello, world";  };  child-node2 {  };  };  node2 {  an-empty-property;  a-cell-property = <1 2 3 4>; /\* each number (cell) is a uint32 \*/  child-node1 {  };  };  }; |

デバイスの定義を追加する際は、そのデバイスのノード定義を追加し、ノード内に

compatible = "デバイス名";

のプロパティ定義を追加する。

各デバイス情報のノードは、statusプロパティで無効にできる。以下のように定義するとこのデバイスノードは無効となるので、対応するドライバのprobeは呼び出されなくなる。

尚、statusプロパティが存在しない場合はデフォルトで有効(status = “ok”と同じ扱い)となる。

例）

|  |
| --- |
| / {  fj-walkmotion {  compatible = "fj-walkmotion";  “okay”または”ok”だと有効になる。  interrupt-parent = <&tlmm>;  interrupts = <66 0>;  status = "disabled";  };  }; |

* include

C言語のように、外部ファイルを読みだす事が可能。

/include/ "ファイル名.dtsi"

であらわす。

一般的にはDTSからincludeするファイルは拡張子を.dtsiとする。

* DTSバージョン

DTSフォーマットがVersion 1に準拠したものであれば、dtsファイルの先頭に

/dts-v1/;

を記述する。（ない場合はVersion 0として扱われる）

なお、記述するのはDTSのみで、DTSIには記述不要。

* ノード定義

{ };

がノード定義の開始と終わりを示しており、デバイスツリーノードはノード名と装置アドレスで定義される。

/ { ... };

はデバイスツリーのルートノードを表す。

ノードの前にラベルをつけることが可能。（ノード参照する場合などに使用する）

ノードの後ろに装置アドレスをつけることが可能。※ラベル、装置アドレスは必須ではない。

ノードの中にプロパティ定義と子ノード定義が含まれている場合、プロパティ定義を子ノードの前に記述すること。

ルートノード

例）

|  |
| --- |
| / {  slim\_msmノード  slim\_msm: slim@fe12f000 {  compatible = "qcom,slim-ngd";  slim\_msmのプロパティ  taiko\_codec {  compatible = "qcom,taiko-slim-pgd";  taiko\_codecノード  (slim\_msmノードの子ノード)  elemental-addr = [00 01 A0 00 17 02];  };  };  }; |

taiko\_codecのプロパティ

* コメント

C言語と同様のコメントをサポートしている。

/\*コメント\*/

// コメント

* エスケープシーケンス

以下の一般的なエスケープシーケンスをサポートしている。

"\n"

"\t"

"\r"

"\(octal value)"

"\x(hex value)"

* プロパティ定義

プロパティに値がある場合は

[ラベル:] プロパティ名 = 値;

であらわす。

プロパティに値がない場合は

[ラベル:] プロパティ名;

であらわす。

値の定義方法は次項以降を参照。

ラベルは必須ではない。

* 配列

配列は< >で囲むこと。

カッコ内にスペースで区切った数値を列挙する。

数値は10進数か16進数で記述し、32ビットがデフォルトとなる。

例）

interrupts = <17 0xc>;

2つの32ビット要素で64ビット値を表すことが可能。

例）

clock-frequency = <0x00000001 0x00000000>;

また、/bits/を使用する事で値のサイズを変更することが可能。

例1）8bitに指定

interrupts = /bits/ 8 <17 0xc>;

例2）64bitに指定

clock-frequency = /bits/ 64 <0x0000000100000000>;

d#で小数、b#でバイナリ、o#で8進数を表すことも可能（使用例なし）。

* 文字列

文字列は、” ”で囲むこと (取得時は終端NULL文字を含む)。

例）

compatible = "simple-bus";

* byte配列

byte配列は[　]で囲むこと。数値は16進数となる。

各byte間のスペースはオプションのため、省いても問題ない。

例）

local-mac-address = [00 00 12 34 56 78];

local-mac-address = [000012345678];

* 組み合わせ

複数の表現要素を **,** で区切って列挙する事が可能。

列挙した値は連結される。

例）

compatible = "ns16550", "ns8250";

example = <0xf00f0000 19>, "a strange property format";

* 別ノード参照

配列内に&ラベル名を記述して別ノードを参照することが可能。

（配列の外で別ノードを参照すると、そのノードのフルパスに変換される）

例）

mpic: interrupt-controller@40000 {

...

};

interrupt-parent = < &mpic >;

&の後にフルパスを記述する事も可能。

例）

interrupt-parent = < &{/soc/interrupt-controller@40000} >;

* 別ノード参照後の編集

別のDTS（DTSI）ファイル等から

/delete-property/ プロパティ名

にてプロパティの削除

/delete-node/ ノード名

にてノードの削除

を行う事が可能。

また、参照したノード内にプロパティを記載する事で、上書き・追記をする事も可能。

この場合、上書き・追記・削除の記述が後になるよう、include順には注意すること。

例)プロパティの追加・変更

|  |
| --- |
| DTSIファイルA  gdsc\_venus: qcom,gdsc@184c018 {  compatible = "qcom,gdsc";  regulator-name = "gdsc\_venus";  reg = <0x184c018 0x4>;  status = "disabled";  }; |

|  |
| --- |
| DTSIファイルB  &gdsc\_venus {  clock-names = "bus\_clk", "core\_clk";  clocks = <&clock\_gcc clk\_gcc\_venus0\_axi\_clk>,  <&clock\_gcc clk\_gcc\_venus0\_vcodec0\_clk>;  status = "ok";  }; |

例)プロパティの削除

|  |
| --- |
| DTSIファイルA  usb3: ssusb@7000000 {  USB3\_GDSC-supply = <&gdsc\_usb30>;  } |

|  |
| --- |
| DTSIファイルB  &usb3 {  /delete-property/ USB3\_GDSC-supply;  }; |

例)ノードの削除

|  |
| --- |
| DTSIファイルA  &soc {  hbtp {  compatible = "qcom,hbtp-input";  vcc\_ana-supply = <&pm8953\_l10>;  vcc\_dig-supply = <&pm8953\_l5>;  qcom,afe-load = <50000>;  qcom,afe-vtg-min = <2850000>;  qcom,afe-vtg-max = <2850000>;  qcom,dig-load = <15000>;  qcom,dig-vtg-min = <1800000>;  qcom,dig-vtg-max = <1800000>;  };  }; |

|  |
| --- |
| DTSIファイルB  &soc {  /delete-node/ hbtp;  }; |

## ドライバでのプロパティ取得方法

ドライバでデバイスツリーの情報を使用する場合、Open Firmwareドライバを使用する。Open FirmwareドライバはLinuxカーネルからDTBの情報にアクセスするためのI/Fであり、本章ではプログラムからDTBの情報にアクセスするための手段などを記載する。

### ヘッダファイル

（1）プロパティ取得API

#include <linux/of.h>

（2）GPIOプロパティ取得API

#include <linux/of\_gpio.h>

### I/F

デバイスドライバ内でDevice Treeのプロパティ参照を行う場合、以降に記述する関数で実現できる。

デバイスノードのポインタにはstruct device構造体内メンバのof\_nodeを渡す。

* 要素数取得

|  |  |  |
| --- | --- | --- |
| **関数名** | **int of\_property\_count\_strings(struct device\_node \*np,const char \*propname)** | |
| **内容** | プロパティにいくつの文字列があるか個数を取得する。 | |
| **入力** | np | デバイスノードのポインタ |
| propname | プロパティ名 |
| **出力** | なし | |
| **戻り値** | 正値 | 正常終了、個数を返す |
| 負値 | 異常終了 |

* 文字列取得（単一定義用）

|  |  |  |
| --- | --- | --- |
| **関数名** | **int of\_property\_read\_string(struct device\_node \*np, const char \*propname,const char \*\*out\_string);** | |
| **内容** | プロパティから文字列を取得する。 | |
| **入力** | np | デバイスノードのポインタ |
| propname | プロパティ名 |
| **出力** | out\_string | 取得文字列ポインタ |
| **戻り値** | 0 | 正常終了 |
| 負値 | 異常終了 |

* 文字列取得（配列定義用）

|  |  |  |
| --- | --- | --- |
| **関数名** | **int of\_property\_read\_string\_index(struct device\_node \*np, const char \*propname,int index, const char \*\*output)** | |
| **内容** | プロパティから指定したindexの文字列を取得する。 | |
| **入力** | np | デバイスノードのポインタ |
| propname | プロパティ名 |
| index | Index |
| **出力** | out\_string | 取得文字列ポインタ |
| **戻り値** | 0 | 正常終了 |
| 負値 | 異常終了 |

* 正数値取得（単一定義版）

|  |  |  |
| --- | --- | --- |
| **関数名** | **int of\_property\_read\_u32(const struct device\_node \*np, const char \*propname,u32 \*out\_value)** | |
| **内容** | プロパティから32ビットの符号なし正数値を取得する。 | |
| **入力** | np | デバイスノードのポインタ |
| propname | プロパティ名 |
| **出力** | out\_value | 取得正数値 |
| **戻り値** | 0 | 正常終了 |
| 負値 | 異常終了 |

* 正数値取得（配列定義版）

|  |  |  |
| --- | --- | --- |
| **関数名** | **int of\_property\_read\_u32\_array(const struct device\_node \*np,const char \*propname, u32 \*out\_values,size\_t sz)** | |
| **内容** | プロパティから32ビットの符号なし正数値の配列データを取得する。 | |
| **入力** | np | デバイスノードのポインタ |
| propname | プロパティ名 |
| sz | 取得サイズ |
| **出力** | out\_value | 取得正数値 |
| **戻り値** | 0 | 正常終了 |
| 負値 | 異常終了 |

* 存在チェック

|  |  |  |
| --- | --- | --- |
| **関数名** | **bool of\_property\_read\_bool(const struct device\_node \*np,const char \*propname)** | |
| **内容** | プロパティが存在するかどうかのbool値を取得する。 | |
| **入力** | np | デバイスノードのポインタ |
| propname | プロパティ名 |
| **出力** | なし | |
| **戻り値** | true | プロパティあり |
| false | プロパティなし |

* Byte配列取得

|  |  |  |
| --- | --- | --- |
| **関数名** | **void \*of\_get\_property(const struct device\_node \*np,const char \*propname,int \*lenp);** | |
| **内容** | プロパティからバイナリデータへのポインタを取得する。 | |
| **入力** | np | デバイスノードのポインタ |
| propname | プロパティ名 |
| **出力** | lenp | サイズ |
| **戻り値** | NULL以外 | 正常終了、バイナリデータのポインタ |
| NULL | 異常終了 |

* GPIOプロパティ取得API

|  |  |  |
| --- | --- | --- |
| **関数名** | int of\_get\_gpio\_flags(struct device\_node \*np, int index enum of\_gpio\_flags \*flags) | |
| **内容** | gpiosプロパティからGPIO番号を取得する。 | |
| **入力** | np | デバイスノードのポインタ |
| **出力** | flags | フラグの値 |
| **戻り値** | 正値 | 正常終了、GPIO番号を返す |
| 負値 | 異常終了 |

# モジュールパラメータ

## 概要

* モジュールパラメータとは

プログラム内部の指定した変数をパラメータとしてパラメータファイルに出力する機能のこと。

モジュールパラメータを使用することで、カーネルモジュールとの情報のやり取りが出来る。

プログラム内部の変数をモジュールパラメータとして設定することで、パラメータファイルが作成される。

* パラメータファイルとは

現在の変数の値が書き込まれているファイルのこと。パラメータファイルを変更することで、プログラム内部の変数の値も書き換わる。

パラメータファイルは

/sys/module/モジュール名/parameters/パラメータ名

に作成される。

## 詳細

### 使用方法

* + - * モジュールパラメータとなる変数を設定
        + 変数を宣言

static int パラメータ名となる変数;

* + - * + パラメータファイルを作成

ファイル名を指定しない場合は①、ファイル名を指定する場合は②を使用する。

パーミッションについては[第7章](#_パーミッション)を参照。

|  |  |  |
| --- | --- | --- |
| **マクロ名** | module\_param | |
| **引数** | name | 変数名(パラメータ名), |
| type | 変数の型(intなど) , |
| perm | パーミッション |
| **ファイルパス** | kernel\include\linux\moduleparam.h | |
| **説明** | 変数をモジュールパラメータとして設定する。  ファイル名は変数の名前になる。 | |

|  |  |  |
| --- | --- | --- |
| **マクロ名** | module\_param\_named | |
| **引数** | name | 保存するファイル名, |
| value | 変数名(パラメータ名), |
| type | 変数の型(intなど), |
| perm | パーミッション |
| **ファイルパス** | kernel\include\linux\moduleparam.h | |
| **説明** | 変数をモジュールパラメータとして設定する。  保存するファイル名を指定する場合に使用する。 | |

これらのマクロは関数の外に記述する。

* + - * 変数値の参照・変更

作成したパラメータファイルの値を参照・変更することでプログラム内の変数を書き換えることが出来る。参照にはcatコマンド、変更にはechoコマンドを使用する。

* + - * + 変数値の参照

cat /sys/module/モジュール名/parameters/パラメータ名

* + - * + 変数値の変更

echo 書き込む値 > /sys/module/モジュール名/parameters/パラメータ名

### 使用例

ここでは、kernel/arch/arm/mach-msm/smd.cを例とする。

* + - * モジュールパラメータとなる変数を設定
        + 変数を宣言

static int msm\_smd\_debug\_mask;

* + - * + パラメータファイルを作成

module\_param\_named(debug\_mask,

msm\_smd\_debug\_mask,

int,

S\_IRUGO | S\_IWUSR | S\_IWGRP)

* + - * 変数値の参照・変更
        + adbに接続

　adb shell

* + - * + 変数値の参照

cat /sys/module/smd/parameters/debug\_mask

* + - * + 変数値の変更

echo 1 > /sys/module/smd/parameters/debug\_mask

※今回はcat/echoを使用しているが、

　モジュールパラメータのファイルに対して読み書きするものならば何でも良い。

　（例：debug\_maskに対してread()/write()を行うテストモジュールを作成する。）

## 応用的な使用方法

パラメータファイルの変更、参照時に呼び出す関数を変更することができる。

これにより、独自の方法でモジュールパラメータの実装を行うことができる。

### 使用方法

* + - * モジュールパラメータ書き換え時に呼び出す関数の変更

|  |  |  |
| --- | --- | --- |
| **マクロ名** | module\_param\_call | |
| **引数** | name | モジュールパラメータ, |
| set | パラメータ変更時に呼ぶ関数のアドレス, |
| get | パラメータ参照時に呼ぶ関数のアドレス, |
| arg | パラメータ変数のアドレス |
| perm | パーミッション |
| **ファイルパス** | kernel\include\linux\moduleparam.h | |
| **説明** | モジュールパラメータ変更・参照時に呼び出す関数を指定し、 \_\_module\_param\_call()を呼び出すマクロ。set, get に自作の関数を指定することで、任意の関数を呼び出せる。 | |

※module\_param()では標準のset(),get()関数が使われている。

* + - * 呼び出す関数の例

関数名は自由に定義できる。

ここではset\_param()、get\_param()とする。

|  |  |  |
| --- | --- | --- |
| **関数名** | int set\_param | |
| **引数** | const char \*val | 書き込む値 |
| struct kernel\_param \*kp | カーネルパラメータ |
| **戻り値** | 0 | 正常な場合 |
| 0以外 | エラーの場合 |
| **説明** | モジュールパラメータの値を変更する。  valに格納されている文字列を適切な型に変換してモジュールパラメータに入れる。  エラーの場合は0以外を返す。 | |

|  |  |  |
| --- | --- | --- |
| **関数名** | int get\_param | |
| **引数** | const char \*buf | モジュールパラメータの値 |
| struct kernel\_param \*kp | カーネルパラメータ |
| **戻り値** | 格納した文字数 | 正常な場合 |
| 負の値 | エラーの場合 |
| **説明** | モジュールパラメータの値をbufに格納する。 | |

### 使用例

set、get関数を作成し、module\_param\_callの引数に設定する。

今回はアクセス権限を８進数で表現している。（全員の読み込み＋所有者の書き込み許可）

|  |
| --- |
| static int test\_val;  static int test\_set\_param(const char \*str, struct kernel\_param \*kp)  {  int ret;  ret = param\_set\_int(str, kp);  if (ret) {  return ret;  }  param\_set\_intが成功していればtest\_valに値が入っている  return ret;  }  static int test\_get\_param(char \*str, struct kernel\_param \*kp)  {  int ret;  ret = param\_get\_int(str, kp);  if (ret) {  return ret;  }  param\_get\_intが成功していればtest\_valの値を文字列にしてstrへ格納している  return ret;  }  module\_param\_call(test\_val, test\_set\_param, test\_get\_param, &test\_val, 0644); |

ここで設定した変数がparam\_set\_int/param\_get\_intで使用される

param\_set\_int/param\_get\_intは変数の型に合わせて利用する

|  |
| --- |
| int param\_set\_byte(const char \*val, const struct kernel\_param \*kp);  int param\_get\_byte(char \*buffer, const struct kernel\_param \*kp);  int param\_set\_short(const char \*val, const struct kernel\_param \*kp);  int param\_get\_short(char \*buffer, const struct kernel\_param \*kp);  int param\_set\_ushort(const char \*val, const struct kernel\_param \*kp);  int param\_get\_ushort(char \*buffer, const struct kernel\_param \*kp);  int param\_set\_int(const char \*val, const struct kernel\_param \*kp);  int param\_get\_int(char \*buffer, const struct kernel\_param \*kp);  int param\_set\_uint(const char \*val, const struct kernel\_param \*kp);  int param\_get\_uint(char \*buffer, const struct kernel\_param \*kp);  int param\_set\_long(const char \*val, const struct kernel\_param \*kp);  int param\_get\_long(char \*buffer, const struct kernel\_param \*kp);  int param\_set\_ulong(const char \*val, const struct kernel\_param \*kp);  int param\_get\_ulong(char \*buffer, const struct kernel\_param \*kp);  int param\_set\_charp(const char \*val, const struct kernel\_param \*kp);  int param\_get\_charp(char \*buffer, const struct kernel\_param \*kp);  int param\_set\_bool(const char \*val, const struct kernel\_param \*kp);  int param\_get\_bool(char \*buffer, const struct kernel\_param \*kp); |

# ログ

## 概要

ドライバにはカーネルの標準出力のようなものが存在しないため、たとえばprintfのような標準関数などは使用できない。そのため、ログを出力するには別の手段を用いる必要がある。ここではカーネルログを用いてログを出力する「printk」について説明する。

## printk

カーネルバッファに文字列を出力するための関数。主にドライバのデバッグに使用する。

そのため、ドライバではprintkを使用し、カーネルを経由させてログを出力する。

カーネルバッファはリングバッファで実装されているため、古いログから順に上書きしていく。

### 書式

int printk(<LOG\_LEVEL> char \*format, ... )

基本的にprintfと同じ使い方で問題ない。書式指定子もprintfとほぼ同じものが使用できる。また、printk独自の仕様としてformatの先頭にログレベルを指定することもできる。

### LOG\_LEVELの一覧

|  |  |  |  |
| --- | --- | --- | --- |
| 定義名 | 定義 | 説明 | ログレベル |
| KERN\_EMERG | <0> | システム停止 | 高  ↑  ｜  ｜  ｜  ｜  ↓  低 |
| KERN\_ALERT | <1> | 深刻なエラー |
| KERN\_CRIT | <2> | 致命的なエラー |
| KERN\_ERR | <3> | エラー |
| KERN\_WARNING | <4> | 警告 |
| KERN\_NOTICE | <5> | 重要な情報 |
| KERN\_INFO | <6> | 情報 |
| KERN\_DEBUG | <7> | デバッグメッセージ |
| KERN\_DEFAULT | <d> | カーネルの設定値 |
| KERN\_CONT | <c> | 手前のログに続けて出力 |

LOG\_LEVELにKERN\_CONTを指定すると、タイムスタンプの付与を行わずにログを続けて出力する。  
一番最初に出力するログにKERN\_CONTを指定した場合の動作については記述が無いため、  
動作保障されないものとする。

ログレベルの定義は<>を使って特殊な記述をしているように見える。

しかし実際はただの文字列であるためprintk(“<%d>”log\n”, level)のように記述すると、レベル指定に変数を使用することもできる。

### 書式指定子

|  |  |
| --- | --- |
| 指定子 | 説明 |
| %c | １文字 |
| %s | 文字列 |
| %d | 符号あり10進整数 |
| %u | 符号なし10進整数 |
| %x | 16進整数 |
| %p | アドレス値 |
| %pf | 関数ポインタのシンボル名 |
| %pF | 関数ポインタのシンボル名＋スタートアドレス＋オフセットアドレス |
| %pS | データポインタのシンボル名 |
| %pB | 関数のスタックトレース |
| %pK | カーネルポインタ |
| %pR | 構造体 |

補足：

スタックトレース …関数の実行過程を表示する。

カーネルポインタ …内容が深く、本筋から外れてしまうため省略。必要であれば外部資料を参照。

パス：\kernel\Documentation\sysctl\kernel.txt

項：kptr\_restrict

## pr\_xxxx

printkには利便性を高めるためのマクロが用意されており、こちらを使用することが多い。

|  |  |
| --- | --- |
| マクロ | 対応するレベル |
| pr\_emerg | KERN\_EMERG |
| pr\_alert | KERN\_ALERT |
| pr\_crit | KERN\_CRIT |
| pr\_err | KERN\_ERR |
| pr\_warn | KERN\_WARNING |
| pr\_notice | KERN\_NOTICE |
| pr\_info | KERN\_INFO |
| pr\_debug | KERN\_DEBUG |
| pr\_cont | KERN\_CONT |

デバイスドライバが通常使用するのはpr\_err、pr\_info、pr\_debug

pr\_debugは通常は出力されないが、dynamic debugという仕組みが備わっており、動的に出力を切り替えることが出来る。

|  |
| --- |
| echo -n 'file drivers/hisho/test\_driver.c +flmpt' > /sys/kernel/debug/dynamic\_debug/control |

を実行するとtest\_driver.cのpr\_debugのログが有効になる。

|  |
| --- |
| echo -n 'file drivers/hisho/test\_driver.c func test\_func +flmpt' > /sys/kernel/debug/dynamic\_debug/control |

を実行するとtest\_driver.cのtest\_func関数のpr\_debugのログだけが有効になる。

## dump\_stack

スタックトレースをログに出力する関数。現在実行している関数が、どのような経緯で実行されたかを知る事が出来る。

エラーが出た際に出力する事で、バグの解析に役立つ。

### 書式

void dump\_stack()

show\_stack(NULL,NULL)という関数を呼び出しても、同じ内容を出力できる。

dump\_stackを使用するには<linux/printk.h>を、show\_stackを使用するには<linux/sched.h>をincludeする必要がある。

## dmesg

カーネルバッファの内容を表示するコマンド。に格納しているログを表示する。

表示するログの種類やレベルを指定できるため、ログファイルを直接参照するよりも効率よくログを見ることができる。

### 書式

dmesg [ -c ] [ -n level ] [ -s bufsize ]

-c ：リングバッファの内容を表示した後、カーネルバッファの内容を消去する。

-n level ：ログレベルをlevelに設定する。指定したログレベル未満のログは出力されなくなる。

例えば-n 1とすれば、KERN\_EMERGのみ出力する設定に変更する。

ただし、/proc/kmsgには全レベルのログが書き込まれる。

-n オプションを指定すると、カーネルバッファの表示や消去はできない。

複数のオプションを指定すると、最後に指定したオプションだけが有効になる。

-s bufsize ：カーネルバッファへの問い合わせにbufsizeの大きさのバッファを用いる。

デフォルトでは16392バイトになっている。

カーネルバッファのサイズをデフォルトより大きくした場合に、

バッファ全体を見るにはこのオプションが必要になる。

# マルチプロセッサ手法について

## 概要

マルチプロセッサとは、一つのコンピュータに複数のCPUを搭載することである。マルチプロセッサの手法として、どのCPUも等しく扱うSMP(Symmetric Multi Processing)と処理装置ごとに役割が決まっているASMP(Asymmetric Multi Processing)がある。

## SMP(Symmetric Multi Processing)

### 概要

SMPとは、複数のCPUが同じ立場で処理を分担するマルチプロセッサ手法のことである。

複数のCPUで一つのメモリ空間を使用し、データを共有する。一つのCPUが故障しても他のCPUで処理を続行できるので、耐障害性の向上に繋がる。

SMPに対応したOSであれば、マルチスレッドの機能を活用した処理の高速化を図る事が出来る。

### I/F

通常のデバイスドライバが本I/Fを使用することは少ない

|  |  |  |
| --- | --- | --- |
| **関数名** | int smp\_call\_function | |
| **引数** | smp\_call\_func\_t func | 実行させる関数 |
| void \*info | funcに渡す引数 |
| int wait | 他のCPUとの同期フラグ  真で同期、偽で非同期 |
| **戻り値** | 0 | 戻り値は0のみ |
| **ファイルパス** | kernel\kernel\smp.c | |
| **説明** | 引数で渡された関数を自分以外の全CPUで実行する。 | |

|  |  |  |
| --- | --- | --- |
| **関数名** | int smp\_call\_function\_single | |
| **引数** | int cpu | 実行するCPU番号 |
| smp\_call\_func\_t func | 実行させる関数 |
| void \*info | funcに渡す引数 |
| int wait | 他のCPUとの同期フラグ  真で同期、偽で非同期 |
| **戻り値** | 0 | 正常な場合 |
| -ENXIO | エラーの場合 |
| **ファイルパス** | kernel\kernel\smp.c | |
| **説明** | 引数で渡された関数を指定したCPUで実行する。  自分のCPUを指定する事も可能。 | |

|  |  |  |
| --- | --- | --- |
| **関数名** | void smp\_call\_function\_many | |
| **引数** | const struct cpumask \*mask | ビットマスク  ->name：マスクの識別名  ->bits：マスクデータ |
| smp\_call\_func\_t func | 実行させる関数 |
| void \*info | funcに渡す引数 |
| bool wait | 他のCPUとの同期フラグ  真同期、偽で非同期 |
| **戻り値** | 無し |  |
| **ファイルパス** | kernel\kernel\smp.c | |
| **説明** | 引数で指定した複数のCPUに処理を実行させる。  ただし、自分のCPUは指定しても実行されない。 | |

※ビットマスクはビット列とcpuを対応付けたもの。

cpu0の場合、32ビットの下位4ビットが0001になり、cpu0と1の場合は0011になる。

|  |  |  |
| --- | --- | --- |
| **関数名** | int on\_each\_cpu | |
| **引数** | smp\_call\_func\_t func | 実行させる関数 |
| void \*info | funcに渡す引数 |
| int wait | 他のCPUと同期フラグ  真で同期、偽で非同期 |
| **戻り値** | 0 | 戻り値は0のみ |
| **ファイルパス** | kernel\kernel\smp.c | |
| **説明** | 引数で渡された関数を全CPUで実行する。 | |

### SMPの注意点

* + - * CPU同士の干渉

共有メモリ(複数のCPUがアクセス可能なRAMの領域のこと)でデータを共有しているため、複数のCPUが同じデータ構造を使う際に互いに干渉してしまう。また、プログラマがポインタや配列の確保を間違った場合に、共有メモリにある他のプロセスに影響を与える可能性がある。

これらは、共有する必要があるものだけを共有メモリに置き、それ以外を共有メモリとして確保していない領域に置くことで、ほとんど起こらなくなる。

共有されないようにするには、DEFINE\_PER\_CPUマクロを使用する。詳細は省略する。

* + - * プリエンティブ

smp\_call\_functionなどの関数で実行させた関数は、すべてプリエンティブ禁止なので、中断して待ち状態に入る事が出来ない。

### シーケンス図

CPU3

CPU1

smp\_call\_function

CPU0

実行

呼び出し

return

CPU2

実行

実行

呼び出し

呼び出し

## プリエンティブ

プリエンティブとは、現在のスレッドを中断してスケジューラを呼び出し、他のスレッドを実行すること。

preempt\_disable()を呼び出すことでプリエンティブを不可にし、割り込みを防ぐことが出来る。そのため、処理を中断してはいけない部分で使用する。

その後、preempt\_enable()を呼び出して、プリエンティブを許可する必要がある。

preempt\_enable()を呼び出さなかった場合は、スレッドが終了した後もスケジューラが呼ばれず、他のスレッドに移れなくなる。

# パーミッション

## 概要

UNIX系のOSで、ディレクトリやファイルに対するアカウントごとのアクセス権のこと。Linuxのように数字で指定することもできる。その場合は先頭に0を付け、8進数表記とする。

## パーミッションの種類

|  |  |  |
| --- | --- | --- |
| パーミッション | 8進表記 | 説明 |
| S\_IFMT | 0170000 | ファイル種別を示すビット領域を表すビットマスク |
| S\_IFSOCK | 0140000 | ソケット |
| S\_IFLNK | 0120000 | シンボリックリンク |
| S\_IFREG | 0100000 | 通常のファイル |
| S\_IFBLK | 0060000 | ブロック・デバイス |
| S\_IFDIR | 0040000 | ディレクトリ |
| S\_IFCHR | 0020000 | キャラクター・デバイス |
| S\_IFIFO | 0010000 | FIFO |
| S\_ISUID | 0004000 | set-user-ID bit |
| S\_ISGID | 0002000 | set-group-ID bit |
| S\_ISVTX | 0001000 | スティッキー・ビット |
| S\_IRWXU | 0000700 | ファイル所有者のアクセス許可用ビットマスク |
| S\_IRUSR | 0000400 | 所有者の読み込み許可 |
| S\_IWUSR | 0000200 | 所有者の書き込み許可 |
| S\_IXUSR | 0000100 | 所有者の実行許可 |
| S\_IRWXG | 0000070 | グループのアクセス許可用ビットマスク |
| S\_IRGRP | 0000040 | グループの読み込み許可 |
| S\_IWGRP | 0000020 | グループの書き込み許可 |
| S\_IXGRP | 0000010 | グループの実行許可 |
| S\_IRWXO | 0000007 | 他人(others)のアクセス許可用ビットマスク |
| S\_IROTH | 0000004 | 他人の読み込み許可 |
| S\_IWOTH | 0000002 | 他人の書き込み許可 |
| S\_IXOTH | 0000001 | 他人の実行許可 |
| S\_IRWXUGO | 0000777 | 全ユーザのアクセス許可用ビットマスク |
| S\_IRUGO | 0000444 | 全ユーザの読み込み権限付与 |
| S\_IWUGO | 0000222 | 全ユーザの書き込み権限付与 |
| S\_IXUGO | 0000111 | 全ユーザの実行権限付与で使われるものと同じ |

# サスペンド機能

## サスペンド・レジューム

端末の状態を保持したままデバイスの停止、再開を行う機能。

Androidでは、デバイスを停止することをサスペンド、デバイスを再開することをレジュームと呼ぶ。

広義の意味では、サスペンド・レジュームの明確な区別は無い。

この機能を使用することで、アプリケーションやOSの実行状態を保持したまま電力消費量を抑えることができる。

### Linuxにおけるサスペンド機能

/sys/power/stateファイルをechoコマンドなどによって書き換えると、カーネルの実行状態が変更される。

実行状態には以下の３種類がある。  
上記ファイルを参照することでどの状態に対応しているかを調べられる。

standby ：スタンバイ (ACPI,S1)

mem ：サスペンド (ACPI,S3)

disk ：ハイバネート (ACPI,S4)

Androidでは基本的にmemのみ対応。

### 実装例

サスペンド・レジューム時のみ登録する基本的な方法と、サスペンド前・サスペンド時・サスペンド後のようにタイミングを細かく指定する方法の２つがある。

* + - * + 基本的な方法

platform\_driverにサスペンドとレジュームを行う関数を登録する。

#include <linux/platform\_device.h>

int test\_driver\_suspend(struct platform\_device \*pdev, pm\_message\_t st) { /\* サスペンド処理 \*/ }

int test\_driver\_resume(struct platform\_device \*pdev) { /\* レジューム処理 \*/ }

static struct platform\_driver test\_driver = { /\* サスペンド・レジュームを行う関数を登録\*/

.suspend = test\_driver\_suspend,

.resume = test\_driver\_resume,

}

static \_\_init int test\_driver\_init(void){

return platform\_driver\_register(&test\_driver); /\*デバイスドライバをカーネルに登録\*/

}

module\_init(test\_driver\_init);

* + - * + dev\_pm\_opsを使用する方法

基本的な方法ではサスペンド時・レジューム時に呼び出す関数しか指定できないが

dev\_pm\_opsを使用することで、より細かいタイミングを指定できる。

#include <linux/platform\_device.h>

int test\_driver\_ prepare (struct device \*dev) {}

int test\_driver\_suspend(struct device \*dev) {}

int test\_driver\_resume(struct device \*dev) {}

void test\_driver\_ complete (struct device \*dev) {}

static struct dev\_pm\_ops test\_pm\_ops = {

.prepare = test\_driver\_prepare,

.complete = test\_driver\_complet,

}

static struct platform\_driver test\_driver = {

.driver = {

.name = "test-driver",

.owner = THIS\_MODULE,

.pm = &test\_pm\_ops,

},

~~.suspend = test\_driver\_suspend,~~ /\* 処理されない \*/

~~.resume = test\_driver\_resume,~~ /\* 処理されない \*/

}

static \_\_init int test\_driver\_init(void){

return platform\_driver\_register(&test\_driver);

}

module\_init(test\_driver\_init);

dev\_pm\_ops構造体に関数を登録し、それをplatform\_driverのメンバpmに格納する。

pmに要素がある場合、platform\_driverより優先して実行されるため注意が必要。

dev\_pm\_opsのメンバの代表的なものを以下に示す。

メンバに関数を登録することで、サスペンド・レジューム時にその関数が呼ばれる。

特に断りが無い場合、メンバに登録する関数の書式は全てint \*関数名(struct device \*dev)である。

|  |  |  |
| --- | --- | --- |
| メンバ名 | 呼び出すタイミング | 備考 |
| prepare | サスペンド開始時 |  |
| suspend | サスペンド処理時 |  |
| suspend\_late | サスペンド処理後 |  |
| suspend\_noirq | サスペンド完了時 | 割り込み禁止 |
| resume\_noirq | レジューム開始時 | 割り込み禁止 |
| resume\_early | レジューム処理前 |  |
| resume | レジューム処理時 |  |
| complete | レジューム完了時 | 関数の戻り値無し |

## サスペンド抑止

サスペンドを抑止する機能がwakelockである。

wakelockを行うI/F　wakelock/pm\_wakeupについて記載する。

### wakelock

* + - * 使用準備

以下の構造体を定義し、初期化関数を使用して初期化する。

構造体の詳しいメンバについては、すべてwakelock用の関数群が扱うため省略。

* + - * + 構造体

|  |  |  |
| --- | --- | --- |
| **構造体名** | struct wake\_lock | |
| **メンバ** | （省略） | （省略） |
| **ファイルパス** | kernel\include\linux\wakelock.h | |
| **説明** | wakelockを掛けるのに必要な構造体。 | |

* + - * + 初期化関数

|  |  |  |
| --- | --- | --- |
| **関数名** | wake\_lock\_init | |
| **引数** | struct wake\_lock \* lock | 定義したwake\_lock構造体 |
| int type | wakelockの種類（後述） |
| const char \* name | ログに出力するためのwakelockの名前 |
| **戻り値** | 無し |  |
| **ファイルパス** | kernel\kernel\power\wakelock.c | |
| **説明** | wake\_lock構造体の初期化を行う。  構造体の定義をした後に、必ず一度呼び出す必要がある。  ここでメンバの設定などを行っている。 | |

※wakelockの種類

WAKE\_LOCK\_SUSPEND ：全システムのサスペンドを抑止する。

~~WAKE\_LOCK\_IDLE ：Android4.0以前は未調査、4.0以降では削除されている。~~

* + - * wakelock

タイムアウトを指定するか否かで、２つの関数を使い分ける。

* + - * + タイムアウト指定なし

|  |  |  |
| --- | --- | --- |
| **関数名** | void wake\_lock | |
| **引数** | struct wake\_lock \*lock | 定義したwake\_lock構造体 |
| **戻り値** | 無し |  |
| **ファイルパス** | kernel\kernel\power\wakelock.c | |
| **説明** | wakelockを掛ける。  既にwakelockを掛けている構造体の場合には、効果が無い。 | |

* + - * + タイムアウト指定あり

|  |  |  |
| --- | --- | --- |
| **関数名** | void wake\_lock\_timeout | |
| **引数** | struct wake\_lock \*lock | 定義したwake\_lock構造体 |
| long timeout | タイムアウトするまでの時間(10ミリ秒単位) |
| **戻り値** | 無し |  |
| **ファイルパス** | kernel\kernel\power\wakelock.c | |
| **説明** | タイムアウト付きでwakelockを掛ける。  既にwakelockを掛けている構造体の場合には、timeoutを再設定する。 | |

* + - * wakelock解除

以下の関数を使用してwakelockを解除する。

|  |  |  |
| --- | --- | --- |
| **関数名** | void wake\_unlock | |
| **引数** | struct wake\_lock \*lock | 定義したwake\_lock構造体 |
| **戻り値** | 無し |  |
| **ファイルパス** | kernel\kernel\power\wakelock.c | |
| **説明** | wakelockを解除する。 | |

※wake\_lock()は何回かけても、wake\_unlock()1回で解除される

### pm\_wakeup

* + - * 初期化

|  |  |  |
| --- | --- | --- |
| **関数名** | int device\_init\_wakeup(struct device \*dev, bool val) | |
| **引数** | struct device \*dev | wakelock制御を行いたいドライバの  struct device構造体ポインタ |
| bool val | true：wakelock制御を行う  false： wakelock制御を行わない |
| **戻り値** | int | 0：成功  0以外：失敗 |
| **ファイルパス** | kernel/include/linux/pm\_wakeup.h | |
| **説明** | 通常はドライバのprobe処理で行う。  struct device構造体はstruct platform\_deviceのメンバdevを使用する。 | |

* + - * wakelock

|  |  |  |
| --- | --- | --- |
| **関数名** | void pm\_stay\_awake(struct device \*dev) | |
| **引数** | struct device \*dev | ドライバのstruct device構造体ポインタ |
| **戻り値** | 無し |  |
| **ファイルパス** | kernel/drivers/base/power/wakeup.c | |
| **説明** | wakelockを掛ける。  既にwakelockを掛けている構造体の場合には、効果が無い。 | |

* + - * wakelock解除

|  |  |  |
| --- | --- | --- |
| **関数名** | void pm\_relax(struct device \*dev) | |
| **引数** | struct device \*dev | ドライバのstruct device構造体ポインタ |
| **戻り値** | 無し |  |
| **ファイルパス** | kernel/drivers/base/power/wakeup.c | |
| **説明** | wakelockを解除する。 | |

※wake\_lock()は何回かけても、wake\_unlock()1回で解除される

## wakelockの確認

定義されているwakelockの詳細な情報を確認することが出来る。

確認方法には「/proc/wakelocks」を開いて確認する方法がある。

* + - * wakelocks

wakelockを使用するモジュールの一覧と、wakelockの情報を確認できる。

* + - * + ログ

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Name | count | expire\_count | wake\_count | active\_since | total\_time | sleep\_time | max\_time | last\_change |
| mmc0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| snfc\_intu\_lock | 1 | 1 | 0 | 0 | 1992168426 | 276026248 | 1992168426 | 25626016290 |
| fj-walkmotion | 4 | 3 | 0 | 0 | 110348576737 | 4481127728 | 57048422055 | 79879762661 |

※可読性重視のため、表形式に変更しています。実際のログはテキスト形式です。

* + - * + 各項目について（調査中）

|  |  |
| --- | --- |
| Name | モジュール名 |
| count | wakelock回数 |
| expire\_count | ファイルの最大数 |
| Wake\_count |  |
| active\_since | ロックを掛けてから経過した時間(10ミリ秒単位) |
| total\_time | 合計wakelock時間 |
| sleep\_time | スリープ時間 |
| max\_time | 1回のwakelock時間の最大 |
| Last\_change | 最後にwakelockを実行・解除した時間（恐らくカーネルの起動時間を使用） |

# Runtime PM

Runtime PMとは、システムのサスペンド・レジュームだけではなく使用状況からデバイス毎のサスペンド・レジューム処理を実装する仕組み。

使用しないデバイスの電源を落とすことで省電力を実現できる。

呼ばれるタイミング

.suspend:システムのサスペンド時

.resume：システムのレジューム時

.runtime\_suspend：デバイスのサスペンド時

.runtime\_resume：デバイスのレジューム時

.runtime\_idle：デバイスのサスペンド直前

static const struct dev\_pm\_ops test\_dev\_pm\_ops = {

.suspend = test\_suspend,

.resume = test\_resume,

.runtime\_suspend = test\_runtime\_suspend,

.runtime\_resume = test\_runtime\_resume,

.runtime\_idle = test\_runtime\_idle,

};

static struct platform\_driver test\_platform\_driver = {

.driver = {

.name = "Testdrv",

.owner = THIS\_MODULE,

.pm = &test\_dev\_pm\_ops,

.of\_match\_table = test\_dt\_match,

},

.probe = test\_probe,

.remove = test\_remove,

~~.suspend = test\_suspend,~~

~~.resume = test\_resume,~~

};

static int \_\_init test\_driver\_init (void)

{

return platform\_driver\_register(&test\_platform\_driver);

}

static void \_\_exit test\_driver\_exit(void)

{

platform\_driver\_unregister(&test\_platform\_driver);

}

module\_init(test\_driver\_init);

module\_exit(test\_driver\_exit);

void pm\_runtime\_enable(struct device \*dev)・・・RuntimePMの制御を有効にする

void pm\_runtime\_disable(struct device \*dev)・・・RuntimePMの制御を無効にする

int pm\_runtime\_get\_sync(struct device \*dev)・・・デバイスを使用する際に実行する

int pm\_runtime\_put\_sync(struct device \*dev)・・・デバイスを使用しなくなった際に実行する

上記以外にも関数は存在するため適宜使い分ける

# ファイルシステム

## 概要

### ファイルシステム

記憶装置に格納されたデータを管理する機能。

データの管理を行うソフトウェアや記憶媒体の管理領域や管理情報のことをファイルシステムと呼ぶこともある。

記憶装置にファイルやディレクトリを作成したり、移動や削除を行ったりする方法、データを記録する方式、管理領域の場所や利用方法などが定められている。方式によって、ファイル名の文字長や、アクセス権の管理方法などが異なる。通常、ファイルシステムはOSの持つ機能の一つとして提供される。

### 疑似ファイルシステム

カーネルの情報にアクセスするための仮想的なファイルシステム。

通常のファイルにアクセスするのと同様な方法で、システムの状態などにアクセスすることが出来る。

## debugfs

### 概要

カーネルのデバッグ情報を参照するための疑似ファイルシステム。

debugfsは、各モジュールで入出力する情報を任意に定義できる。

ファイルを操作した際に、カーネルが表示用関数を呼ぶ。

同じ名前で、ファイルシステム自体のデバッグを行うLinuxのコマンドが存在するが、これとは関係ない。

### 実装方法

* + - * 実装

ヘッダファイル<linux/seq\_file.h>と<linux/debugfs.h>が必要である。

* + - * + debugfsの設定

ドライバのprobe関数などでdebugfs\_create\_fileを呼び出す。

|  |  |  |
| --- | --- | --- |
| **関数名** | struct dentry \* debugfs\_create\_file | |
| **引数** | const char \*name | 作成するファイル名 |
| umode\_t mode | パーミッション |
| struct dentry \*parent | ファイルを作成するフォルダ。  NULLを指定すると  /sys/kernel/debug/  配下へ作成される。  debugfs\_create\_dirを使用して独自のフォルダ配下にファイルを作成する際に設定する。 |
| void \*data | 固有情報 |
| const struct file\_operations \*fops | ファイルを取り扱うための構造体 |
| **戻り値** | dentry構造体へのポインタ |  |
| **ファイルパス** | kernel\fs\debugfs\inode.c | |
| **説明** | debugfsのファイルを作成。  dataに固有情報のアドレスを設定することでファイル操作時の関数からinode->i\_privateを参照することでポインタを取得することが出来る。 | |

パーミッションについては[パーミッションの章](#_パーミッション)を参照。

* + - * + 値の設定

構造体file\_operationsにそれぞれ適切な値を設定する。

Linuxのバージョンによって中身が異なるため注意が必要である。

|  |  |  |
| --- | --- | --- |
| **構造体名** | file\_operations | |
| **メンバ** | struct module \*owner | ローカルファイルシステムのモジュール |
| loff\_t \*llseek | シークに使用する関数を指定 |
| ssize\_t \*read | 読み込みに使用する関数を指定 |
| ssize\_t \*write | 書き込みに使用する関数を指定 |
| ssize\_t \*aio\_read | 非同期の読み込みに使用する関数を指定 |
| ssize\_t \*aio\_write | 非同期の書き込みに使用する関数を指定 |
| int \*readdir | ディレクトリの読み込みに使用する関数を指定 |
| unsigned int \*poll | ファイルの変更が行われたか確認する関数を指定 |
| long \*unlocked\_ioctl | 対応するデバイスに命令する関数を指定 |
| long \*compat\_ioctl | 対応するデバイスに命令する関数を指定？ |
| int \*mmap | ファイルのメモリマッピングを行う関数を指定 |
| int \*open | ファイルを扱うための関数を指定 |
| int \*flush | ファイルへの参照が減ったときに呼び出される？ |
| int \*release | ファイル解放を行う関数を指定 |
| int \*fsync | キャッシュされたデータをディスクへ書き込む関数を指定 |
| int \*aio\_fsync | 非同期でキャッシュをディスクへ書き込む関数を指定 |
| int \*fasync | シグナルによる非同期I/Oの通知の許可または禁止する関数を指定 |
| int \*lock | ファイルをロックする関数を指定 |
| ssize\_t \*sendpage | あるファイルから別のファイルにデータを転送する関数を指定 |
| unsigned long \*get\_unmapped\_area | 未使用の仮想アドレスを取得する関数を指定 |
| int \*check\_flags | フラグをチェックする関数を指定 |
| int \*flock | ファイルをロックする関数を指定？ |
| ssize\_t \*splice\_write | データを継ぎ合わせて書き込む関数を指定？ |
| ssize\_t \*splice\_read | データを継ぎ合わせて読み込む関数を指定？ |
| int \*setlease |  |
| long \*fallocate | ファイルに割り当てられたディスク空間を操作する関数を指定 |
| **ファイルパス** | kernel\include\linux\fs.h | |
| **説明** | ファイルを操作する関数群。 | |

* + - * + ファイルを扱う関数の作成

構造体file\_operations->openに指定する関数を作成する。

* + - * + 表示用関数

以下の関数を使用する。

|  |  |  |
| --- | --- | --- |
| **関数名** | int seq\_printf | |
| **引数** | struct seq\_file \*s | ファイルを扱うためのseq\_file |
| const char \*str | フォーマットを指定する文字列 |
| ・・・ | 対応したフォーマットに入れる値 |
| **戻り値** | 0 | 成功 |
| -1 | 表示する文字列がページサイズを超過 |
| **ファイルパス** | kernel\fs\seq\_file.c | |
| **説明** | デバッグ情報の表示をする関数。  使用方法はprintfと殆ど同じである。  表示する内容がページサイズを超えるとエラーになる。  ページサイズは一般的に4KBである。 | |

他にも次のような関数がある

int seq\_putc(struct seq\_file \*m, char c);

int seq\_puts(struct seq\_file \*m, const char \*s);

int seq\_write(struct seq\_file \*seq, const void \*data, size\_t len);

int seq\_escape(struct seq\_file \*, const char \*, const char \*);

int seq\_path(struct seq\_file \*, struct path \*, char \*);

int seq\_dentry(struct seq\_file \*, struct dentry \*, char \*);

int seq\_path\_root(struct seq\_file \*m, struct path \*path, struct path \*root,char \*esc);

* + - * 表示
        + ファイルのマウント  
          adb上で"mount -t debugfs none ファイルパス"を実行すると、指定したファイルパスの場所にdebugfsをマウントする。基本的には "/sys/kernel/debug" を指定する。Androidでは起動時に自動でマウントされる。
        + 値の表示  
          adb上から"cat /sys/kernel/debug/ファイル名"というコマンドを実行する。

### 実装例

ここではkernel\drivers\gpio\gpiolib.cでの実装例を示す。

* + - * 実装
        + debugfsの設定

|  |
| --- |
| static int \_\_init gpiolib\_debugfs\_init(void)  {  (void) debugfs\_create\_file("gpio", S\_IFREG | S\_IRUGO,NULL, NULL, &gpiolib\_operations);  return 0;  }  subsys\_initcall(gpiolib\_debugfs\_init); |

gpioというファイルを全ユーザが読み込み可能な通常のファイルとして、

/sys/kernel/debug直下に作成される。

gpioは構造体gpiolib\_operationsに扱われる。

* + - * + 値の設定

|  |
| --- |
| static const struct file\_operations gpiolib\_operations = {  .open = gpiolib\_open,  .read = seq\_read,  .llseek = seq\_lseek,  .release = single\_release,  }; |

openに、ファイルを開いた時に呼び出す関数を指定する。

他のメンバには、seq\_read, seq\_lseek, single\_release というseq\_fileを扱う為に用意されている関数を入れている。

* + - * + ファイルを扱う関数の作成

|  |
| --- |
| static int gpiolib\_open(struct inode \*inode, struct file \*file)  {  return single\_open(file, gpiolib\_show, NULL);  } |

single\_openの2番目の引数に表示用の関数を指定する。

* + - * + 表示用関数の作成

|  |
| --- |
| static int gpiolib\_show(struct seq\_file \*s, void \*unused)  {  if (chip->label)  seq\_printf(s, ", %s", chip->label);  if (chip->can\_sleep)  seq\_printf(s, ", can sleep");  } |

* + - * 表示
        + ファイルのマウント  
          adb shell

mount -t debugfs none /sys/kernel/debug

* + - * + 値の表示  
          cat /sys/kernel/debug/gpio

## procfs

### 概要

カーネルが管理するプロセスの情報を参照するための疑似ファイルシステム。

カーネル内の情報を表示する、カーネル内のパラメータ値の表示や変更行うなど、様々な目的で

使用される。

Linuxではコマンドの実装に必要となるため、標準でマウントされる。

### 詳細

* + - * プロセスに関する情報

プロセスに関する情報は、/proc/プロセスID(PID)ディレクトリで提供されている。

提供されている情報には、以下のようなものがある。

|  |  |
| --- | --- |
| /proc/PID/cmdline | プロセス起動時に表示されるコマンドライン文字列 |
| /proc/PID/cwd | カレントディレクトリへのシンボリックリンク |
| /proc/PID/environ | プロセスの設定している環境変数 |
| /proc/PID/exe | 実行ファイルへのシンボリックリンク |
| /proc/PID/fd | ファイル記述子に対応するシンボリックリンク |
| /proc/PID/fdinfo | ファイル記述子に対応したファイル位置やフラグ情報があるディレクトリ |
| /proc/PID/maps | プロセスのメモリマップとパーミッション |
| /proc/PID/mountinfo | マウントポイントについての情報 |
| /proc/PID/root | ルートディレクトリへのシンボリックリンク  (通常は/となる) |
| /proc/PID/stack | スタックの情報 |
| /proc/PID/stat | プロセスの状態についての情報  psコマンド(実行中のプロセスの表示)に使用される |
| /proc/PID/status | プロセスの動作状態やメモリ使用状況などの情報 |
| /proc/PID/statm | ページ単位でのメモリ使用量 |
| /proc/PID/task | そのプロセスを親としている全プロセスへのハードリンクを格納したディレクトリ |

ハードリンク：ファイルシステム上のファイルやディレクトリ等の資源と、その資源につけられた名前を結びつけること。また、その結びつきのこと。

* + - * プロセスと無関係な情報

プロセスと直接無関係なものは以下のようなものがある。

|  |  |
| --- | --- |
| /proc/cmdline | 起動時にLinuxカーネルに渡された引数 |
| /proc/cpuinfo | CPUの情報 |
| /proc/devices | メジャーデバイス番号の一覧 |
| /proc/diskstats | ディスクの統計情報 |
| /proc/filesystems | カーネルが対応しているファイルシステムの一覧 |
| /proc/interrupts | CPUごとの割り込み回数 |
| /proc/iomem | I/Oメモリマップ |
| /proc/ioports | I/Oポートマップ |
| /proc/kallsyms | カーネルの外部シンボル定義 |
| /proc/kcore | システムの物理メモリ |
| /proc/loadavg | ロードアベレージの情報 |
| /proc/locks | ファイルロックの情報 |
| /proc/meminfo | メモリの使用量に関する統計情報 |
| /proc/modules | カーネルモジュールの情報 |
| /proc/net | ネットワークに関する情報 |
| /proc/partitions | パーティションブロックの割り当て情報 |
| /proc/self | 動作中のプロセス自身の/proc/PIDへのシンボリックリンク |
| /proc/slabinfo | カーネルキャッシュの情報 |
| /proc/stat | システムに関する統計情報 |
| /proc/swaps | 使用中のスワップ領域の情報 |
| /proc/sys | 動的に変更可能なカーネルオプションへのアクセス |
| /proc/tty | 疑似端末に関する情報 |
| /proc/uptime | 起動してからの時間とアイドル時間 |
| /proc/version | カーネルのバージョン番号 |
| /proc/vmstat | 仮想メモリの情報 |
| /proc/zoneinfo | 物理メモリの領域に関する情報 |

メジャーデバイス番号：デバイスドライバ等の識別に使用される。

割り当て方はOSやプラットフォームによって異なる。

ロードアベレージ：実行待ちプロセス数の平均値

### 実装

ヘッダファイル<linux/proc\_fs.h>が必要である。

* + - * procfsの登録

create\_proc\_entryを使って/procに登録する。

①create\_proc\_entry関数

|  |  |  |
| --- | --- | --- |
| **関数名** | struct proc\_dir\_entry \*create\_proc\_entry | |
| **引数** | const char \*name | 作成するファイル名 |
| mode\_t mode | パーミッション |
| struct proc\_dir\_entry \*parent | 作成するフォルダ  NULLの場合/proc直下に作られる |
| **戻り値** | proc\_dir\_entry構造体へのポインタ | 関数の登録に使用 |
| NULL | 失敗した場合 |
| **ファイルパス** | kernel\fs\proc\generic.c | |
| **説明** | procfsのファイルを作成。 | |

取得したproc\_dir\_entry構造体へのポインタを使用して、read\_proc、write\_procに

関数を登録する。

②proc\_dir\_entry構造体

|  |  |  |
| --- | --- | --- |
| **構造体名** | proc\_dir\_entry | |
| **メンバ** | unsigned int low\_ino | inode番号 |
| mode\_t mode | パーミッション |
| nlink\_t nlink | リンク数 |
| uid\_t uid | ユーザID |
| gid\_t gid | グループID |
| loff size | ファイルサイズ |
| const struct inode\_operations \*proc\_iops | inodeオペレーション |
| const struct file\_operations \*proc\_fops | ファイルオペレーション |
| struct proc\_dir\_entry \*next, | 親ディレクトリ |
| \*parent, | 親ディレクトリが同一の子リンク |
| \*subdir | 子リンク |
| void \*data | 個別データ |
| read\_proc\_t \*read\_proc | 読み込み用関数 ※１ |
| write\_proc\_t \*write\_proc | 書き込み用関数 ※１ |
| atomic\_t count | 参照カウント |
| int pde\_users | モジュールの使用者数（調査中） |
| struct completion \*pde\_unload\_completion | （調査中） |
| struct list\_head pde\_openers | （調査中） |
| spinlock\_t pde\_unload\_lock | （調査中） |
| int deleted | 削除されたかどうかのフラグ |
| u8 namelen; | ファイル名の長さ |
| const char \*name | /proc/以下に作成するファイル名 |
| **ファイルパス** | kernel\include\linux\proc\_fs.h | |
| **説明** | procファイルシステムのファイル情報。 | |

※１ read\_proc\_t、write\_proc\_tは以下のように型定義されている。

* + - * + read\_proc\_t

typedef int (read\_proc\_t)(char \*page, char \*\*start, off\_t off,

int count, int \*eof, void \*data);

|  |  |
| --- | --- |
| char \*page | カーネルで確保されるメモリ領域 |
| char \*\*start | ドライバが書き込んだデータのポインタ |
| off\_t off | ドライバが返すデータのオフセット |
| int count | ユーザプロセスが一度に読み込むサイズ |
| int \*eof | データの終端 |
| void \*data | create\_proc\_entryで設定したプライベートエリア |

* + - * + write\_proc\_t（調査中）

typedef int (write\_proc\_t)(struct file \*file, const char \_\_user \*buffer,

unsigned long count, void \*data);

|  |  |
| --- | --- |
| struct file \*file, |  |
| const char \_\_user \*buffer |  |
| unsigned long count |  |
| void \*data |  |

write\_proc\_tではユーザプロセスからデータが渡されるので、カーネル領域にデータを登録するにはcopy\_from\_user関数を使用する。

copy\_from\_user関数

|  |  |  |
| --- | --- | --- |
| **関数名** | unsigned long copy\_from\_user | |
| **引数** | void \*to | コピー先のカーネルアドレス |
| const void \_\_user \*from | コピー元のユーザアドレス |
| unsigned long n | コピーするサイズ  fromが不正である場合は、toに0がsize分、設定される |
| **戻り値** | 0 | 成功 |
| n | コピーできなかったサイズ |
| **ファイルパス** | kernel\include\linux\proc\_fs.h | |
| **説明** | ユーザ空間からカーネル空間へデータをコピーする。 | |

### 実装例

読み込み用、書き込み用の関数を作成し、procfsに登録する。

procfsへ登録

|  |
| --- |
| static void procfs\_init(struct seq\_file \*s)  {  struct proc\_dir\_entry \*entry;    /\*  \* procfsの登録内容  \* name：ファイル名はproc\_sample  \* mode：全ユーザに読み書きの権限を与える  \* parent：指定なし（/proc直下に作成）  \*/  entry = create\_proc\_entry("proc\_sample ", 0666, NULL);    /\* 関数の登録 \*/  entry->read\_proc = procfs\_read;  entry->write\_proc = procfs\_write;  } |

読み込み用、書き込み用関数

|  |
| --- |
| static char procfs\_buf[256];  /\* read\_procに登録する関数 \*/  int procfs\_read(char \*page, char \*\*start, off\_t off, int count, int \*eof, void \*data)  {  int len;    /\* メモリにprocfs\_bufの内容を読み込む \*/  len = sprintf(page, “%s\n”, procfs\_buf);    return len;  }  /\* write\_procに登録する関数 \*/  int procfs\_write(struct file \*file, const char \_\_user \*buffer, unsigned long count, void \*data)  {  if (count > 256)  return -EINVAL;  if (!buffer)  return -EINVAL;    /\* procfs\_bufに0を書き込む \*/  memset(procfs\_buf, 0, sizeof(procfs\_buf));  /\* データをコピーできたかの判定 \*/  if (copy\_from\_user(procfs\_buf, buffer, count))  return -EFAULT;    return count;  } |

## sysfs

### 概要

デバイスやドライバに関する情報をユーザ空間へエクスポートする疑似ファイルシステム。

主にデバイスなどの情報をユーザとやり取りする目的で使用される。

procfsとは機能的な違いは無い。

sysfsでは、実際のデバイスへのシンボリックリンクが置かれ、それをディレクトリの階層構造と

して表現する。

### 詳細

標準的なLinux 2.6では、次のような情報を提供する。

|  |  |
| --- | --- |
| /sys/block | ブロックデバイスへのシンボリックリンク |
| /sys/bus | バスの所属 |
| /sys/class | クラスで分類されたデバイスの親子関係  この下にデバイスファイルを用意することで、ハードウェアが変更されても同じインターフェースで済む |
| /sys/dev | ブロックデバイスとキャラクターデバイス |
| /sys/devices | デバイスの親子関係 |
| /sys/firmware | ファームウェア情報 |
| /sys/fs | ファイルシステム情報 |
| /sys/hypervisor | ハイパーバイザ情報 |
| /sys/kernel | カーネル情報 |
| /sys/module | モジュール情報 |
| /sys/power | 電源情報 |

ブロックデバイス：データの入出力をブロック単位で行うデバイス

キャラクタデバイス：データの入出力をバイト単位で行うデバイス

ハイパーバイザ：コンピュータを仮想化し、複数の異なるOSを並列に実行できるようにするソフトウェア。

# GPIO

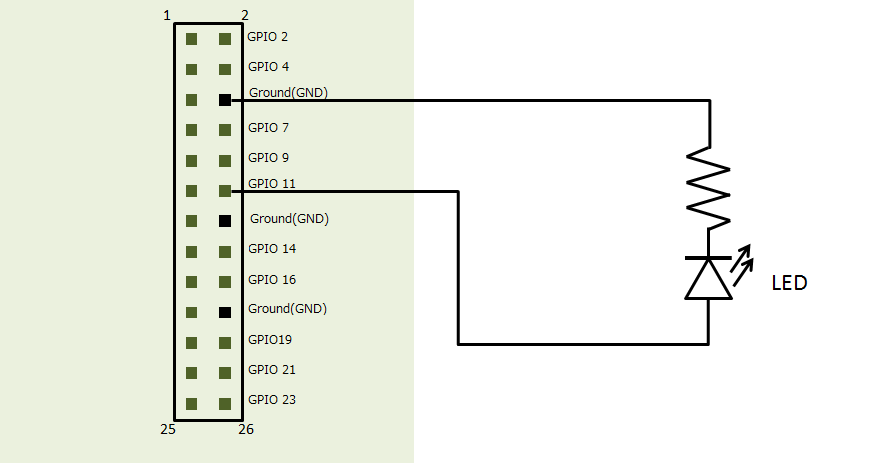
## 概要

この章では、主にソフトウェア側のGPIO制御について説明する。

GPIOとは、組み込みボードに実装される入出力端子の一種である。GPIOは特定の用途や信号が決まっておらず、ソフトウェアの指示によって任意の入力あるいは出力に利用することができる。複数の端子がGPIOに割り当てられている場合には、これを一つのグループとして一括して制御することができ、「GPIOポート」などと呼ばれる。

GPIOの状態には"High"と"Low"がある。読んで字のごとく"高い電圧がかかる状態"と"低い電圧がかかる状態"である。

たとえば下図のような回路がありLEDを点灯させたい場合、GPIO11を「入出力＝出力設定」「出力値＝High」と設定することで点灯させることができる。



## 詳細

GPIOはチップセットによって様々な設定ができる。

Qualcomm社のMSMシリーズでは、以下の設定ができる。

|  |  |
| --- | --- |
| In／Out | In ：入力設定  Out ：出力設定  ※モード設定がFunctionの場合は無効。 |
| モード設定 | 0 ：GPIO設定  1～5 ：Function設定  ※MSMシリーズではモード設定を1～5にすることで、各GPIOにある独自機能を使用できる。本資料ではGPIO設定のみ説明し、Functionについての説明は行わない。 |
| 出力時の設定値 | Low ：Low  High ：High |
| Pull\_Up/Down設定値 | 内部Pull Up/Pull Downの設定  In設定時、入力電圧が無い場合の読み値が不定とならないように設定する  NoPull ：指定なし（外部回路側に実装がある場合に使用）  PullUp ：High  PullDown：Low |
| Drive STRENGTH（Outのみ） | 出力強度 ：2mA～16mAまで2mA刻みで設定可能（デフォルトは2mA） |

## GPIOライブラリ

Linux 2.6.3以降のバージョンでは、GPIOを扱いやすくするためのライブラリgpiolibが追加された。

本項では主にこのライブラリについて説明する。

参考：Linux 2.6ドキュメント - GPIO Interfaces

※Linux 2.6のものとAndroid4.1のものでは微妙に異なるため注意が必要。

### 使用方法

kernel\drivers\gpio\gpiolib.cに用意されている関数を使用してGPIOの制御を行う。

このライブラリから各ドライバの関数を呼び出すことでハードウェアごとの差異をなくしている。

使用の際はlinux/gpio.hをincludeする。

数が多いため、代表的なもののみの紹介に留める。

* + - * + gpio\_request

|  |  |  |
| --- | --- | --- |
| **関数名** | int gpio\_request | |
| **引数** | unsigned gpio | 使用するGPIO番号 |
| const char \* label | ラベル　主にログ出力時に使用する |
| **戻り値** | 0以上 | 成功：各ドライバのrequest関数の戻り値 |
| 0未満 | 失敗：エラーコード |
| **ファイルパス** | kernel\drivers\gpio\gpiolib.c | |
| **説明** | GPIOの使用を開始する。  使用を終える場合はgpio\_freeを呼び出す。 | |

* + - * + gpio\_free

|  |  |  |
| --- | --- | --- |
| **関数名** | void gpio\_free | |
| **引数** | unsigned gpio | 開放するGPIO番号。 |
| **戻り値** | なし |  |
| **ファイルパス** | kernel\drivers\gpio\gpiolib.c | |
| **説明** | 使用していたGPIOを開放する。  使用していないGPIOに対してこの関数を実行するとエラーメッセージを出力する。  DEBUGマクロが定義されていれば、スタックトレースも出力する。 | |

* + - * + gpio\_direction\_input

|  |  |  |
| --- | --- | --- |
| **関数名** | int gpio\_direction\_input | |
| **引数** | unsigned gpio | 入力に設定するGPIO番号 |
| **戻り値** | 0以上 | 成功：各ドライバのdirection\_input関数の戻り値 |
| 0未満 | 失敗：エラーコード |
| **ファイルパス** | kernel\drivers\gpio\gpiolib.c | |
| **説明** | GPIOピンを入力用に切り替える。 | |

* + - * + gpio\_direction\_output

|  |  |  |
| --- | --- | --- |
| **関数名** | int gpio\_direction\_output | |
| **引数** | unsigned gpio | 出力に設定するGPIO番号 |
| int value | 出力の初期値(Pull設定)  0:Low  1:High |
| **戻り値** | 0以上 | 成功：各ドライバのdirection\_output関数の戻り値 |
| 0未満 | 失敗：エラーコード |
| **ファイルパス** | kernel\drivers\gpio\gpiolib.c | |
| **説明** | GPIOピンを出力用に切り替える。  Open drain pinが1、かつOpen source pinが0になるまで出力には切り替えられず、  代わりにgpio\_direction\_inputが呼び出される。 | |

* + - * + gpio\_set\_value

|  |  |  |
| --- | --- | --- |
| **関数名** | void gpio\_set\_value | |
| **引数** | unsigned gpio | 出力するGPIO番号 |
| int value | 0:Low  1:High |
| **戻り値** | なし |  |
| **ファイルパス** | kernel\drivers\gpio\gpiolib.c | |
| **説明** | GPIOの値を設定する。(0または1のみ) | |

* + - * + gpio\_get\_value

|  |  |  |
| --- | --- | --- |
| **関数名** | int gpio\_get\_value | |
| **引数** | unsigned gpio | 取得するGPIO番号 |
| **戻り値** | int | 0:Low  1:High |
| **ファイルパス** | kernel\drivers\gpio\gpiolib.c | |
| **説明** | GPIOの値を取得する。 | |

※IN(入力)

電気回路のほかの部分からのデジタル信号を読み取る

※OUT(出力)

他デバイスの制御(操作)、や信号の通知を行う

## pinctrl

近年、GPIOの設定はpinctrl機能を使用している。

デバイスツリーへGPIOの設定値を記述し、pinctrl機能のI/Fを使用して設定を切り替えることが出来る。

編集中

# タイマ

## タイマ処理の役割

LINUXにおけるタイマ処理の役割は、大きく下記の２つである。

・時刻を正確に刻む処理(時刻処理)

・一定時間後に指定した処理を行う処理(時限処理)

LIINUXでは、一定間隔で周期的に発生する割り込み(タイマ割り込み)を利用して、

これらの機能を実現している。

### 時刻処理

カーネル起動時にシステムBIOSから日時を取得し、それをもとにカーネルが時刻を刻んでいく。

システムBIOS自体はリアルタイムクロックと呼ばれる、ハードに搭載されている時計から

時刻情報を取得する。リアルタイムクロックは通常電池で駆動し、コンピュータの電源が落ちても時刻情報を保持している

### 時限処理

Linuxの時限処理は、タイマー・リストと呼ばれるメカニズムで実現されている。

## タイマーリスト

タイマーリストは、登録した一定時間後にコールバックされるハンドラを登録することができる、

汎用のメカニズムである。その仕組みの核となっているのがtimer\_list構造体である。

タイマーリストに、任意の関数と起動時刻を登録しておき、起動時刻になると、

登録した関数が呼び出される。ローカルタイマーソフト割り込みハンドラ(run\_timer\_softirq関数)が

この起動時刻を監視し、起動時刻がきたら登録されたハンドラを実行する。

### timer\_list構造体

タイマーリストはtimer\_list構造体によって実現されている。

|  |  |  |
| --- | --- | --- |
| **構造体名** | timer\_list | |
| **メンバ** | struct list\_head entrｙ | ダイナミックタイマーをつなぐ  双方向循環リストのリンク用 |
| unsigned long expires | タイマの期限がいつであるかを指定する  この値がjiffies以下になると、  このタイマは期限が来ていると見なされる |
| spinlock\_t lock | 競合状態を回避するためのスピンロック |
| unsigned long magic | マジックナンバー |
| void (\*function)(unsigned long) | タイマの期限が来たときに実行される関数のアドレス |
| unsigned long data | functionが指す関数に渡す引数を指定する  (functionに渡すのみなので、数値は自由に決めてよい) |
| struct tvec\_t\_base\_s \*base | タイマーリスト副表。tv1～tv5までの副表を持ち、  ハンドラの起動残り時刻に合わせて分類して管理している。 |
| **ファイルパス** | \include\linux\timer.h | |
| **説明** | 時限処理に使われる構造体 | |

### タイマ処理に関係する関数

* + 初期化

kernelバージョンによって初期化マクロが異なるため注意。

**kernel 4.19より前**

動的な初期化（関数内で実行）

|  |  |  |
| --- | --- | --- |
| **マクロ名** | setup\_timer(timer, fn, data) | |
| **引数** | timer | 初期化するタイマーリスト |
| fn | ハンドラ |
| data | ハンドラの引数に渡す値  unsigned long型にキャストすることでポインタも使用可 |
| **戻り値** | void | なし |
| **ファイルパス** | \include\linux\timer.h | |
| **説明** | 登録済みのタイマリストに対して実行しないこと（不正メモリアクセスの可能性がある） | |

|  |  |  |
| --- | --- | --- |
| **マクロ名** | init\_timer(timer) | |
| **引数** | timer | 初期化するタイマーリスト |
| **戻り値** | void | なし |
| **ファイルパス** | \include\linux\timer.h | |
| **説明** | 登録済みのタイマリストに対して実行しないこと（不正メモリアクセスの可能性がある） | |

静的な初期化（初期化された変数として宣言）

|  |  |  |
| --- | --- | --- |
| **マクロ名** | DEFINE\_TIMER(\_name, \_function, \_expires, \_data) | |
| **引数** | \_name | 宣言するタイマーリスト変数の名前 |
| \_function | ハンドラ |
| \_expires | ハンドラを起床する時刻（初期化点では0でよい） |
| \_data | ハンドラの引数に渡す値 |
| **戻り値** | void | なし |
| **ファイルパス** | \include\linux\timer.h | |
| **説明** |  | |

**kernel 4.19以降**

動的な初期化（関数内で実行）

|  |  |  |
| --- | --- | --- |
| **マクロ名** | timer\_setup(timer, callback, flags) | |
| **引数** | timer | 初期化するタイマーリスト |
| callback | ハンドラ |
| flags | フラグ |
| **戻り値** | void | なし |
| **ファイルパス** | \include\linux\timer.h | |
| **説明** | 登録済みのタイマリストに対して実行しないこと（不正メモリアクセスの可能性がある） | |

静的な初期化（初期化された変数として宣言）

|  |  |  |
| --- | --- | --- |
| **マクロ名** | DEFINE\_TIMER(\_name, \_function) | |
| **引数** | \_name | 宣言するタイマーリスト変数の名前 |
| \_function | ハンドラ |
| **戻り値** | void | なし |
| **ファイルパス** | \include\linux\timer.h | |
| **説明** |  | |

* + タイマの登録

|  |  |  |
| --- | --- | --- |
| **関数名** | void add\_timer | |
| **引数** | timer\_list \*timer | 登録するタイマーリスト |
| **戻り値** | void | なし |
| **ファイルパス** | \include\linux\timer.h | |
| **説明** | 本関数を呼び出したcpu用のタイマーリストにハンドラを登録する。  既に登録されているタイマリストに対して実行するとkernel panicするため、mod\_timerの方が使いやすい。 | |

* + タイマの登録（変更）

|  |  |  |
| --- | --- | --- |
| **関数名** | int mod\_timer | |
| **引数** | timer\_list \*timer  unsigned long expires | 起動時刻を変更するタイマーリスト  変更する起動時刻 |
| **戻り値** | int | タイマーが登録されているなら　　１  タイマーが登録されていないなら　 0 |
| **ファイルパス** | \include\linux\timer.h | |
| **説明** | タイマーリストに登録されているハンドラの起動時刻を変更する。  expiresに設定する時刻はjiffiesの単位。msなどの間隔をmsecs\_to\_jiffiesを使用して変換して設定すること  例)  100ms後であれば  jiffies + msecs\_to\_jiffies(100) | |

* + タイマの削除

|  |  |  |
| --- | --- | --- |
| **関数名** | void del\_timer | |
| **引数** | timer\_list \*timer | 削除するタイマーリスト |
| **戻り値** | int | 削除に成功したら1、失敗なら0 |
| **ファイルパス** | \include\linux\timer.h | |
| **説明** | タイマリストからハンドラを削除 | |

* + タイマが登録済みかをチェック

|  |  |  |
| --- | --- | --- |
| **関数名** | int timer\_pending | |
| **引数** | timer\_list \*timer | 参照するタイマーリスト |
| **戻り値** | int | 引数で渡されたタイマが、タイマリストに  登録されているなら　　1  登録されていなければ　0 |
| **ファイルパス** | \include\linux\timer.h | |
| **説明** | 指定されたハンドラが、タイマリストに登録されているか否かを確認する | |

* + タイマハンドラの引数について

**kernel4.19より前**

kernel4.19より前のタイマハンドラは引数がunsigned long型である。ドライバの管理領域などをタイマハンドラで参照したい場合はtimer\_list構造体のメンバdataへ参照したいアドレスをキャストして設定する。

タイマハンドラ側でも引数を参照したいアドレス型へキャストすることで使用することができる。

|  |
| --- |
| struct test\_driver\_info {  struct timer\_list test\_timer;  };  static void test\_timer\_expired(unsigned long arg)  {  struct test\_driver\_info \*info = (struct test\_driver\_info \*)arg;  }  static int test\_probe(struct platform\_device \*pdev)  {  int ret = 0;  struct device \*dev = &pdev->dev;  struct test\_driver\_info \*info;  info = devm\_kzalloc(dev, sizeof(struct test\_driver\_info), GFP\_KERNEL);  if (info == NULL) {  ret = -ENOMEM;  goto out;  }  setup\_timer(&info->test\_timer, test\_timer\_expired, (unsigned long)info);  ～～～～～～ |

**kernel4.19以降**

kernel4.19以降はタイマハンドラでfrom\_timerマクロを使用することでtimer\_list変数のアドレスから構造体の先頭位置を算出することが出来る。

|  |
| --- |
| struct test\_driver\_info {  struct timer\_list test\_timer;  };  static void test\_timer\_expired(struct timer\_list \*arg)  {  struct test\_driver\_info \*info = from\_timer(info, arg, test\_timer);  }  from\_timer(ポインタ変数, ハンドラの引数, 構造体内のtimer\_list構造体のメンバ名)  static int test\_probe(struct platform\_device \*pdev)  {  int ret = 0;  struct device \*dev = &pdev->dev;  struct test\_driver\_info \*info;  info = devm\_kzalloc(dev, sizeof(struct test\_driver\_info), GFP\_KERNEL);  if (info == NULL) {  ret = -ENOMEM;  goto out;  }  timer\_setup(&info->test\_timer, test\_timer\_expired, 0);  ～～～～～～ |

# 遅延処理

## 概要

遅延処理とはボトムハーフの一種で、処理を後で実行するための機能である。

Linuxカーネルではタスクレットとワークキューをと呼ばれる仕組みを提供している。

本章では、タスクレットとワークキューについてそれぞれ説明していく。

## タスクレット

タスクレットを使う場合DECLARE\_TASKLETを使ってtasklet\_struct構造体を初期化する。

### tasklet\_struct構造体

tasklet\_struct構造体

|  |  |  |
| --- | --- | --- |
| **構造体名** | tasklet\_struct | |
| **メンバ** | struct tasklet\_struct \*next | 次のタスクレット構造体 |
| unsigned long state | タスクレットの状態  TASKLET\_STATE\_SCHED(スケジューリングされている)  TASKLET\_STATE\_RUN(実行中) |
| atomic\_t count | 0　　　：スケジューリングされる  0以外　：無効 |
| void (\*func)(unsigned long) | タスクレットで処理したい関数 |
| unsigned long data | タスクレットで処理する関数の引数 |
| **ファイルパス** | include/linux/sched.h | |
| **説明** | タスクレットの核となる構造体  この構造体にタスクレットハンドラを登録し、tasklet\_schedule関数が呼び出されると、割り込みコンテキストとして実行される | |

タスクレット構造体は以下図のように、tasklet\_vec構造体によって

双方向リストとして管理されている。

|  |
| --- |
|  |

### タスクレット実装の流れ

実装の流れ

-------------------------------------------------------------------------------------

①タスクレット構造体の作成

　　　　　↓

②タスクレット構造体に処理したい関数を入れる

　　　　　↓

③タスクレットをスケジューリングの対象にする

-------------------------------------------------------------------------------------

①と②は下記のマクロで実装できる

： DECLARE\_TASKLET( 構造体名, 遅延する関数, 関数に渡す引数)

③は下記の関数で実装できる

tasklet\_schedule( &作成したタスクレット名)を使う。

内部でraise\_softirq\_irqoff(TASKLET\_SOFTIRQ)を呼び出すことにより、

ソフト割り込みコンテキストとして、ハンドラが実行されることとなる

taklet\_struct構造体はsoft\_irq[]配列の要素(ハンドラ)として登録されている

tasklet\_actioln()から呼び出され、登録sれているハンドラが起動する

ハード割り込みハンドラ起動

↓

ソフト割り込みハンドラ起動要求(soft\_irq[]をスケジューリングの対象にする)

↓

soft\_irq[]に登録されている、action\_taskret()を呼び出す

↓

tasklet\_structに登録されている関数が実行される。

## ワークキュー

### 概要

ワークキューはタスクレットでは実装不可能な遅延処理を行いたい場合に使う。

例）

・sleepする必要がある。

・メモリを確保したい(ページアウトのためにsleepする可能性がある)

・セマフォ等の同期処理を行う

主な理由としては、タスクレットは割り込みコンテキストで実装されていて、

ワークキューはプロセスコンテキストで実装されていることにある。

### ワークキュー実装の流れ

ワークキュー実装の流れは、

独自のワークキュー(スレッド)を作成する or カーネル共有のワークキューを作成する

指定した時間だけ遅延させる or 遅延する時間を指定しない

で処理を変える必要がある。

ワークキューの生成

|  |  |
| --- | --- |
| 独自のワークキュー | カーネル共通のワークキュー |
| create\_workqueue()  create\_singlethread\_workqueue()  等 | 不要 |

ワーク情報の準備

|  |  |  |
| --- | --- | --- |
|  | 遅延時間指定なし | 遅延時間指定あり |
| ワーク情報型 | struct work\_struct | struct delayed\_work |
| 初期化（静的） | DECLARE\_WORK | DECLARE\_DELAYED\_WORK |
| 初期化（動的） | INIT\_WORK | INIT\_DELAYED\_WORK |

ワーク処理登録

|  |  |  |
| --- | --- | --- |
|  | 遅延時間指定なし | 遅延時間指定あり |
| カーネル共通のワークキュー使用 | schedule\_work | schedule\_delayed\_work |
| 独自のワークキュー使用 | queue\_work | queue\_delayed\_work |

### ワークキューに関係する関数

( a ) create\_workqueue

|  |  |  |
| --- | --- | --- |
| **マクロ名** | create\_workqueue | |
| **引数** | char \* | 作成するワークキューの名前 |
| **戻り値** | struct workqueue\_struct \* | ワークキュー情報のポインタ。失敗時はNULLを返す |
| **ファイルパス** | include/linux/workqueue.h | |
| **説明** | workqueueをcpuの数だけ生成する | |

( b ) create\_single\_workqueue

|  |  |  |
| --- | --- | --- |
| **関数名** | create\_singlethread\_workqueue | |
| **引数** | char \* | 作成するワークキューの名前 |
| **戻り値** | struct workqueue\_struct \* | ワークキュー情報のポインタ。失敗時はNULLを返す |
| **ファイルパス** | include/linux/workqueue.h | |
| **説明** | workqueueを一つだけ生成する。 | |

( c )queue\_work

|  |  |  |
| --- | --- | --- |
| **関数名** | queue\_work | |
| **引数** | workqueue\_struct \*  work\_queue \* | 登録するワークキューのポインタ  登録するワーク情報のポインタ |
| **戻り値** | void | なし |
| **ファイルパス** | include/linux/workqueue.h | |
| **説明** | 指定したワークキューへワーク情報を登録する | |

( d ) queue\_delayed\_work

|  |  |  |
| --- | --- | --- |
| **関数名** | queue\_delayed\_work | |
| **引数** | workqueue\_struct \*  delayed\_work \*  unsigned long | 登録するワークキューのポインタ  登録するワーク情報のポインタ  遅延時間 |
| **戻り値** | void | なし |
| **ファイルパス** | include/linux/workqueue.h | |
| **説明** | 指定したワークキューへワーク情報を登録する（遅延あり）  ※遅延時間はjiffiesの単位。msなどの間隔をmsecs\_to\_jiffiesを使用して変換して設定すること | |

( e ) schedule\_work

|  |  |  |
| --- | --- | --- |
| **関数名** | schedule\_work | |
| **引数** | work\_struct \* | 登録するワーク情報のポインタ |
| **戻り値** | void | なし |
| **ファイルパス** | include/linux/workqueue.h | |
| **説明** | カーネル共通のワークキューへワーク情報を登録 | |

( f ) schedule\_delayed\_work

|  |  |  |
| --- | --- | --- |
| **関数名** | schedule\_delayed\_work | |
| **引数** | struct delayed\_work \* | 登録するワーク情報のポインタ |
| unsigned long | 遅延時間 |
| **戻り値** | void | なし |
| **ファイルパス** | include/linux/workqueue.h | |
| **説明** | カーネル共通のワークキューへワーク情報を登録（遅延あり） | |

( g ) flush\_work

|  |  |  |
| --- | --- | --- |
| **関数名** | flush\_work | |
| **引数** | struct work\_struct \*work | ワーク情報のポインタ |
| **戻り値** | bool | true:処理終了  false：既に終了済 |
| **ファイルパス** | include/linux/workqueue.h | |
| **説明** | ワーク情報が終了するまで待つ | |

( h ) flush\_delayed\_work

|  |  |  |
| --- | --- | --- |
| **関数名** | flush\_delayed\_work | |
| **引数** | struct delayed\_work \*dwork | ワーク情報のポインタ |
| **戻り値** | bool | true：処理終了  false：既に終了済 |
| **ファイルパス** | include/linux/workqueue.h | |
| **説明** | ワーク情報（遅延あり）が終了するまで待つ | |

* + - * ワーク処理の引数について

ドライバの管理領域などをワークキューの関数内で参照したい場合はcontainer\_ofマクロを使用することでワーク情報変数のアドレスから構造体の先頭位置を算出することが出来る。

container\_of(ハンドラの引数, 構造体名, 構造体内のワーク情報のメンバ名.work)

|  |
| --- |
| struct test\_driver\_info {  struct delayed\_work test\_dwork;  struct work\_struct test\_work;  };  static void test\_dwork\_func(struct work\_struct \*work)  {  struct test\_driver\_info \*info = container\_of(work,  struct test\_driver\_info, test\_dwork.work);  }  container\_of(ハンドラの引数, 構造体名, 構造体内のワーク情報のメンバ名.work)  static void test\_work\_func(struct work\_struct \*work)  {  struct test\_driver\_info \*info = container\_of(work,  struct test\_driver\_info, test\_work);  } |

# 排他制御

複数のプロセスから一つの資源への同時アクセスにより競合が発生する場合は、１つのプロセスにのみ資源を独占的に利用させて、他のプロセスが利用できないようにする事で整合性を保つ必要がある。

この処理を排他制御と呼ぶ。排他制御を実現するための主な実装技術には以下がある

* + Spinlock（スピンロック）
  + Semaphore（セマフォ）
  + Mutex（ミューテックス）

## Spinlock

spinlockはロックを獲得できるまで単純にループ（スピン）して定期的にロックをチェックしながら待つ方式。

資源獲得時にlock関数、解放時にunlock関数を実行する。

待っている最中はCPUを占有するためlock中の処理は必要最低限とすること。

lock中はプリエンプションが禁止されるため、sleepが発生する処理は禁止。

主に変数やキュー・リストの操作時に使用される。

* includeファイル

<linux/spinlock.h>

* 必要なロック変数

spinlock\_t型

* ロック変数の初期化

動的な初期化

spin\_lock\_init(&ロック変数)

静的な初期化

DEFINE\_SPINLOCK(ロック変数の名前)

* lock/unlock

|  |  |
| --- | --- |
| lock | unlock |
| spin\_lock\_irqsave(&ロック変数,flags) | spin\_unlock\_irqrestore(&ロック変数,flags) |
| spin\_lock\_irq(&ロック変数) | spin\_unlock\_irq(&ロック変数) |
| spin\_lock\_bh(&ロック変数) | spin\_unlock\_bh(&ロック変数) |
| spin\_lock(&ロック変数) | spin\_unlock(&ロック変数) |

※flagsは割り込み状態を保存する変数でunsigned long型（&はつけないので注意）

どのコンテキストからも実行できる

spin\_lock\_irqsave()/spin\_unlock\_irqrestore()

が一番使われる。

## Semaphore

セマフォカウンタという資源の個数を管理する値を用いて、プロセス間の排他制御を行っている。

資源獲得時にdown関数、解放時にup関数を実行する。

セマフォカウンタが0の場合、スケジューラを実行して他のプロセスへ実行権を移しながらカウンタが増えるまで待つ。

プリエンプションが発生するため割り込みコンテキストでは使用不可。

* includeファイル

<linux/semaphore.h>

* 必要なロック変数

struct semaphore型

* ロック変数の初期化

動的な初期化

sema\_init(&ロック変数, カウンタの数)

静的な初期化

DEFINE\_SEMAPHORE(ロック変数の名前)

※カウンタには1が設定される

* up/down

void down(&ロック変数)

void up(&ロック変数)

## Mutex

Mutexはロックを獲得できるまでスケジューラを実行して他のプロセスへ実行権を移しながら待つ方式。

資源獲得時にlock関数、解放時にunlock関数を実行する。

Semaphore同様にプロセス間での排他に使用される。

プリエンプションが発生するため割り込みコンテキストでは使用不可。

* includeファイル

<linux/mutex.h>

* 必要なロック変数

struct mutex型

* ロック変数の初期化

動的な初期化

mutex\_init(&ロック変数)

静的な初期化

DEFINE\_MUTEX(ロック変数の名前)

* lock/unlock

mutex\_lock(&ロック変数)

mutex\_unlock(&ロック変数)

# メモリ管理

## ページング

Linux Kernelではページング方式と呼ばれるアルゴリズムを使用した設計となっている。

物理メモリをページ（通常は4KB）と呼ばれる単位に分割し、構造体struct pageで管理する。

CPUがメモリアクセスする際は物理アドレスではなく仮想アドレスを使用し、MMU(Memory Management Unit)と呼ばれる部品を使用して、仮想アドレスから物理アドレスへ変換されて物理メモリへ到達する。

## buddy（バディ）システム

### 概要

断片化を防ぎながら（断片化しないわけではない）連続した実ページを確保する仕組み。

2のべき乗単位でページを管理しており、メモリを割り当てるときは、要求されたサイズを満たす最小のブロックを返す。べき乗部分をorder（オーダー）と呼んでいる。

0=1ページ=4KB

1=2ページ=8KB

2=4ページ=16KB

3=8ページ=32KB

・・・

メモリ解放時は、同じ大きさのブロックの中で連続したメモリがあれば、より大きなブロックにマージする。

### I/F

|  |  |  |
| --- | --- | --- |
| **関数名** | **struct page \*alloc\_pages(gfp\_t gfp\_mask, unsigned int order)** | |
| **引数** | gfp\_t gfp\_mask | オプションフラグ |
| unsigned int order | 確保するオーダー |
| **戻り値** | struct page \* | 確保したページ構造体のポインタ |
| **説明** | buddyシステムへページ獲得要求を行う。  取得したページ構造体からpage\_address()マクロを使用することで仮想アドレスを取得することが出来る。取得した領域は物理的にも連続している。  断片化していると空メモリが十分に残っていても失敗する可能性がある。order=3以上は要注意。 | |

gfp\_maskには確保する際のオプションを指定する

|  |  |
| --- | --- |
| GFP\_KERNEL | カーネル用メモリ通常の方法。スリープ可。 |
| GFP\_ATOMIC | 高優先度。スリープ不可。主に割込み処理で使う。 |
| GFP\_USER | ユーザ空間用のメモリの通常の方法。スリープ可。 |
| その他 | include/linux/gfp.hを参照 |

orderには確保したいオーダーを指定する

|  |  |  |
| --- | --- | --- |
| **関数名** | **void \_\_free\_pages(struct page \*page, unsigned int order)** | |
| **引数** | struct page \*page | 解放するページ構造体のポインタ |
| unsigned int order | 解放するオーダー |
| **戻り値** | なし |  |
| **説明** | buddyシステムで獲得したページの解放を行う。  仮想アドレスを元に解放する関数も用意されている(free\_pages) | |

### procfs

buddyシステムで管理している空ページの情報は以下で確認できる。

/proc/buddyinfo

## スラブアロケータ

### 概要

ページ単位ではなくbyte単位で管理し、あらかじめキャッシュしておくことで効率よく確保・解放が行える。確保時にはキャッシュから取得し、キャッシュが足りなくなれば自動的に確保される。

このキャッシュはスラブキャッシュと呼ばれる。

### I/F

|  |  |  |
| --- | --- | --- |
| **関数名** | **struct kmem\_cache \***  **kmem\_cache\_create(const char \*name, size\_t size, size\_t align,**  **unsigned long flags, void (\*ctor)(void \*))** | |
| **引数** | const char \*name | 名前（/proc/slabinfoで確認する際に使用される） |
| size\_t size | サイズ（キャッシュ全体のサイズではなく、オブジェクト1つ分のサイズ） |
| size\_t align | アライメントサイズ |
| unsigned long flags | 特殊な動作をさせたい場合に指定  include/linux/slab.hに定義されている |
| void (\*ctor)(void \*) | コンストラクタ関数へのポインタ  キャッシュからオブジェクトを取得後、なんらかの初期化処理をしたい場合にのみ関数を実装し、指定する |
| **戻り値** | struct kmem\_cache \* | 作成したキャッシュ情報のポインタ |
| **説明** | スラブキャッシュの作成を行う。 | |

|  |  |  |
| --- | --- | --- |
| **関数名** | **void \*kmem\_cache\_alloc(struct kmem\_cache \*cachep, gfp\_t flags)** | |
| **引数** | struct kmem\_cache \* cachep | キャッシュ情報のポインタ |
| gfp\_t flags | オプションフラグ |
| **戻り値** | void \* | オブジェクトのポインタ |
| **説明** | スラブキャッシュからオブジェクトを取得する。 | |

|  |  |  |
| --- | --- | --- |
| **関数名** | **void kmem\_cache\_free(struct kmem\_cache \*cachep, void \*objp)** | |
| **引数** | struct kmem\_cache \* cachep | キャッシュ情報のポインタ |
| void \*objp | オブジェクトのポインタ |
| **戻り値** | なし |  |
| **説明** | スラブキャッシュから取得したオブジェクトをキャッシュへ解放する。 | |

|  |  |  |
| --- | --- | --- |
| **関数名** | **void kmem\_cache\_destroy(struct kmem\_cache \*s)** | |
| **引数** | struct kmem\_cache \*s | キャッシュ情報のポインタ |
| **戻り値** | なし |  |
| **説明** | スラブキャッシュを破棄する。 | |

* kernel共通のスラブ

kernel共通で使用できるスラブが用意されており、より簡単に使用できる。通常はこの関数がよく使用される。

|  |  |  |
| --- | --- | --- |
| **関数名** | **void \*kmalloc(size\_t size, gfp\_t flags)** | |
| **引数** | size\_t size | 確保したいサイズ |
| gfp\_t flags | オプションフラグ |
| **戻り値** | void \* | 確保したメモリポインタ |
| **説明** | kernelが用意したスラブキャッシュから要求したサイズに近いものを選択し、オブジェクトを取得する | |

|  |  |  |
| --- | --- | --- |
| **関数名** | **void kfree(const void \*objp)** | |
| **引数** | const void \* | 解放したいオブジェクトのポインタ |
| **戻り値** | なし |  |
| **説明** | kmallocで獲得したメモリを解放する | |

### procfs

スラブアロケータで管理している情報は以下で確認できる。

/proc/slabinfo

## vmalloc

### 概要

専用の仮想アドレス空間を管理する。

### I/F

|  |  |  |
| --- | --- | --- |
| **関数名** | **void \*vmalloc(unsigned long size)** | |
| **引数** | unsigned long size | 確保したいサイズ |
| **戻り値** | void \* | 割り当てた仮想アドレス |
| **説明** | vmalloc用の仮想アドレスから確保サイズ分の領域を割り当てる。  この時点では物理アドレスにマッピングされているわけではなく、アクセスしたタイミングでページフォルトが発生し、buddyシステムから1ページ確保して割り当てられる。このため、物理的に連続している保証はない。 | |

|  |  |  |
| --- | --- | --- |
| **関数名** | **void vfree(const void \*addr)** | |
| **引数** | const void \* | 仮想アドレス |
| **戻り値** | なし |  |
| **説明** | vmallocで割り当てた領域を解放する | |

### procfs

vmallocで管理している情報は以下で確認できる。

/proc/vmallocinfo

## 空きメモリ回収

空きメモリ（free page）が少なくなってくると、空きメモリを増やすために様々な機構が動き出す。

空きメモリの回収方法は主に以下の方法がある

* + 1. ページキャッシュ解放

ストレージのデータがキャッシュされたページを解放する。

ストレージとキャッシュの内容が一致していればそのまま解放、一致していなければ（Dirtyページ）、ストレージに書き出してから解放。Dirtyでないページを優先して解放する。

* + 1. Anonymous（匿名）ページのswap

キャッシュ用のページとは異なるページで、主にユーザー層のプログラムで使用しているページ。このページをswap（Linuxでは事前に設定したswap用のストレージ領域、Androidではzram）領域へ書き出して解放する。

* + 1. プロセスの強制停止(kill)

プロセスをkillすることでそのプロセスが使用していたメモリを解放する。

回収者はkswapd、direct reclaim、OOM Killer、Low Memory Killer(Android固有)が存在する

|  |  |
| --- | --- |
| 回収者 | 回収方法 |
| kswapd | A)、B) |
| direct reclaim | A)、B) |
| OOM Killer | C) |
| Low Memory Killer | C) |

### zram

zramはRAM上に生成する圧縮ブロックデバイス。Androidのswapディスクとして使用される。

RAM上に生成することで高速化、ストレージへの負担を減らす効果がある。

データを圧縮することで空きメモリを増やすことが出来る。

### kswapd

空きメモリの回収をバックグラウンドで行うカーネルスレッド。

min、low、highの3種類の閾値（WaterMark）に従って動作している。

WaterMarkの値は/proc/zoneinfoで確認できる。

空きメモリがlowを下回るとkswapdが起床して空きメモリの回収を始める。

空きメモリがhighを超えるとページ回収を終える。

### direct reclaim

direct reclaimはメモリ獲得を行ったプロセスのコンテキストで空きメモリの回収を行う事。

空きメモリがWaterMarkのminを下回るとdirect reclaimが行われる。

### OOM Killer

direct reclaimを行っても空きメモリが回収できず、それ以上処理が続行できない事をOOM(Out of Memory)と呼び、この際にOOM Killerによって適当なプロセスが強制終了(kill)される。

### Low Memory Killer

Android固有の機能。メモリの状況を監視し、空きメモリに応じてプロセスのkillを行う。

killする際はAndroidが管理する優先度を使用する。

# init.rc

## 概要

・initプロセス(system/core/init/init.cpp)と共にAndroidシステムの初期化を行う。

また、init.rc内に記述されているアクション･サービス(後述)によって各種設定が行われる。

以下にAndroid初期化の際の全体の流れを簡単に図示する。

メインRAM、周辺機器の初期化を行う。

**Bootloader**

また、ブートパラメータのメモリ上への配置完了後に

Kernelを起動させる

ブートパラメータを解析し、KernelCoreや各ドライバの初期化を行う。

**Kernel**

最後にAndroidのinitプロセスを起動する。

initプロセスがinit.rcを読み込み初期化を行う。

**Android**

その後仮想マシンが起動し、AndroidFrameworkの

各Serviceが起動され、ユーザが操作可能な状態となる

## アクションについて

・init.rcの要素の一つで、triggerとcommandで構成される。

・triggerに指定したイベント発生時に、commandに記述した処理を実行する。

## サービスについて

・サービスの実行条件を定義

・実行ファイルに対して引数の設定やオプションを付け、起動条件を設定可能

## 詳細

「LINUX/android/system/core/rootdir」内のinit.rcに記述を行う。

また、

<https://android.googlesource.com/platform/system/core/+/master/init/README.md>

に詳細な情報が記述されている。

### アクションの記述方法

以下に書式、triggerとcommandの例を記述する。

1. 書式

|  |
| --- |
| on <trigger>  <command>  <command>  <command> |

（B）trigger例

|  |  |
| --- | --- |
| boot | initスタート時のトリガー |
| <name>=<value> | valueがnameに設定されたときのトリガー |
| "device-added-<path>  device-removed-<path>" | デバイスノードに追加、削除されたときに  トリガーとなる |
| service-exited-<name> | 特定のサービス終了時のトリガー |
| init | 環境変数の設定、ディレクトリの設定時のトリガー |

（C）command例

|  |  |
| --- | --- |
| start <service> | 指定したserviceが未起動なら起動する |
| stop <service> | 指定したserviceが起動していたら終了する |
| chmod<octal-mode> <path> | アクセス権限の設定 |
| ifup <interface> | ネットワークインターフェースを有効にする |
| write　<path>　<string> [ <string> ]\* | <path>にあるファイルを開き、文字列を  書き込む |

使用例

|  |
| --- |
| on boot  export PATH /system/lib /\* 環境変数PATHに/system/libを設定 \*/  mkdir /proc /\* procディレクトリを作成 \*/  mount proc proc /proc /\* procデバイスを/procディレクトリにマウント \*/  ifup lo /\* ネットワークインターフェースloを有効化 \*/  hostname localhost /\* ホスト名をlocalhostに設定 \*/ |

### **サービスの記述方法**

以下に書式、optionの例を記述する。

<書式>

|  |
| --- |
| service <name> <pathname> [ <argument> ]\*  <option>  <option>  ... |

<option例>

|  |  |
| --- | --- |
| setenv <name> <value> | 起動時に環境変数に値を設定 |
| onrestart | サービスの再起動の際にコマンドを実行 |
| user<username> | ユーザー名の変更を行う。初期値はroot |
| critical | 4分間に4回以上終了した場合に、デバイスをリカバリーモードで再起動させる。 |
| class　<name> | サービスにクラス名を付け、グループ化することが出来る |

使用例1

|  |
| --- |
| service serfaceflinger /system/bin/ serfaceflinger  class main ←mainクラスにグループ化  user system ←ユーザをsystemに変更  group graphics drmrpc ←グループをgraphycs、補助グループをdrmrpcに変更  onrestart restart zygote ←サービス再起動時にzyoteサービスを再起動 |

・serfaceflinger：画面描画の実行、管理を行うサービス

・zygote：Android アプリ起動の際にプロセスを生成を行う