飛翔ソフトウェア

新人研修

Linuxドライバ開発編

|  |  |  |
| --- | --- | --- |
| 日付 | 変更内容 | 変更者 |
| 2021/04/07 | 新人研修向けに新規作成 | 庭 |
|  |  |  |

# 研修の目的

モバイル開発現場の業務に近い形の研修を行うことで、配属先でスムーズに作業を行えるようにする。

# 使用する機器・ソフト

* DragonBoard 410c
* UbuntuがインストールされたPC

-git/repo

-docker

-Androdi OS ソースコード

* WindowsがインストールされたPC

-Tera Term

-VNC Viewer

-Android platform-tools

-Vysor

-秀丸エディタやさくらエディタなどのテキストエディタ

* USBケーブル(Type-A to micro-B)

# DragonBoard 410cとは

Qualcomm社のSoC「Snapdragon」を搭載したシングルボードコンピュータ。

Snapdragonは多くの携帯電話・スマートフォンに使用されている。

※Soc・・・System-on-a-chip。集積回路の1個のチップ上に、プロセッサコアをはじめ一般的なマイクロコントローラが持つような機能の他、応用目的の機能なども集積し、連携してシステムとして機能するよう設計されている、集積回路製品

主要スペック

|  |  |
| --- | --- |
| プロセッサ | Snapdragon 410(Cortex-A53/4コア、クロック最大1.2GHz、32-bit and 64-bit support) |
| GPU | Adreno 306(400MHz) |
| メモリ | 1GB(LPDDR3 533MHz) |
| ストレージ | 8GB eMMC 4.5 |
| 通信機能 | 2.4GHzのIEEE 802.11b/g/nに準拠した無線LAN(Wi-Fi)、  Bluetooth 4.1、GPS |
| I/O | HDMI(Type A)、USB 2.0(A端子×2、micro B端子×1)、40ピン拡張コネクタ、60ピンハイスピードコネクタ、microSDスロット(SD 3.0、UHS-I) |
| サポートOS | Android 5.1 (Lollipop) on Linux Kernel 3.10  Android 6.0 (Marshmallow) on Linux Kernel 3.10  Linux based on Debian 8.0  Open Embedded  Ubuntu Core  Windows 10 IoT Core |

# Ubuntuとは

Debian GNU/Linuxを母体としたオペレーティングシステム（OS）である。Linuxディストリビューションの1つであり、フリーソフトウェアとして提供されている。

※LinuxディストリビューションとはLinuxカーネルとその他ソフトウェア群を1つにまとめ、利用者が容易にインストール・利用できるようにしたものである。

# Linuxとは

Linuxとは、狭義にはUnix系オペレーティングシステムカーネルであるLinuxカーネルを指し、広義にはそれをカーネルとして周辺を整備したシステム全体のことをいう。

Linuxは、スーパーコンピュータ・メインフレーム・サーバ・パーソナルコンピュータ・組み込みシステム（携帯電話やテレビなど）など、幅広い種類のハードウェアで使用されている。

Linuxカーネルのソースコードは無償で入手でき、GNU一般公衆利用許諾書のもとにおいて、非営利・営利に関わらず誰でも自由に使用・修正・頒布できる。

スマートフォンやタブレット端末用プラットフォームAndroidはLinuxカーネルの上に構築されている。

# 機材構成図

今回の研修の大まかな構成図を記載する。

WindowsPC

UbuntuPC

DragonBoard410c

adb/ Vysor

fastboot

ソースコード

repo/git

ソースコード

docker

ビルドツール



テキストエディタ

ダウンロード

編集

書き込み

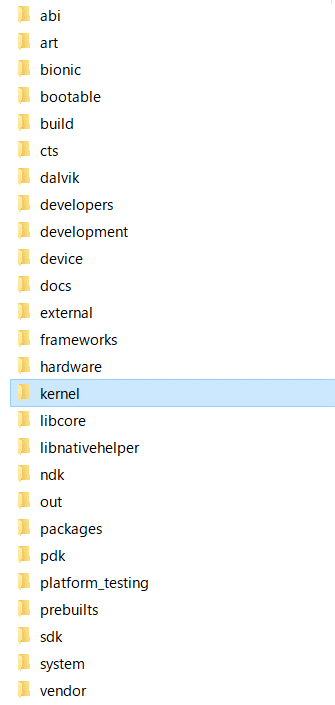
デバッグ



1. WindowsPCからUbuntuPCへアクセスし、ソースコードの編集を行う
2. UbuntuPCでdockerを使用してビルドに必要な環境を構築し、ソースコードのビルドを行う（ビルドのコマンドはWindowsPCからでも可）
3. WindowsPCからUbuntuPCへアクセスし、ビルドの成果物を入手
4. WindowsPCからfastbootツールを使用してビルドの成果物をDragonBoard410cへ書き込む
5. WindowsPCからadbやVysorを使用してデバッグを行う

# デバイスドライバの追加方法

## フォルダ構成(Android配下)



/bootloader…abootのソースコード

/recovery…リカバリのソースコード

…initプロセスのソースコードやinit.rc、各種デーモンのソースコードが格納されている

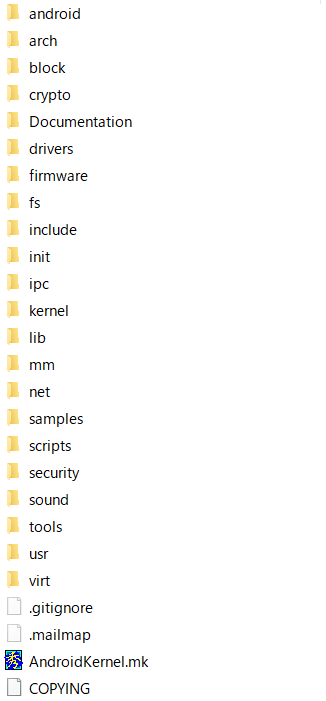
…Linux kernelのソースコード

主にこの配下を編集する

…Android frameworkのソースコードが格納されている

…デバイス固有の情報を記載するmakefileが格納されている

## フォルダ構成(kernel配下)



…ドライバのソースコードが格納されている

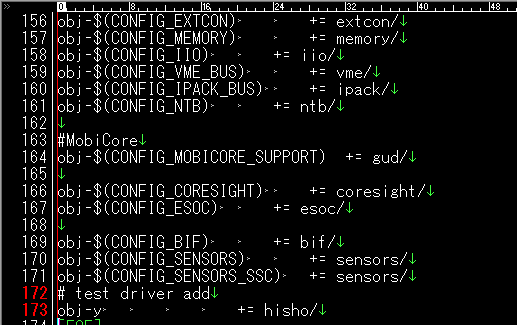
## デバイスドライバの追加

既存のフォルダにファイルを追加してもよいが、分かりやすいように新規フォルダを作成し、その配下にデバイスドライバのソースコードを格納する。

1. kernel/drivers配下にフォルダを作成する。フォルダ名に日本語は使用しないこと
2. kernel/drivers/Makefileを編集し、1で追加したフォルダ名を追記する

これによって追加したフォルダがビルド時にMakefileを走査する対象に加えられる

下記例ではhishoというフォルダを作成している



1. 1で作成したフォルダへソースコードとMakefileを追加する

下記例ではtest\_driver.cというソースコードを追加する

ファイルの先頭には必ずGPLライセンスの表記とCOPYRIGHTを記載する

**文字コードはUFF-8、改行コードはLFで保存すること**

kernel/drivers/hisho/test\_driver.c

|  |
| --- |
| // SPDX-License-Identifier: GPL-2.0  /\*----------------------------------------------------------------------------\*/  // COPYRIGHT HISHO Software 2021  /\*----------------------------------------------------------------------------\*/  #include <linux/module.h>  カーネル内ではprintk関数を使うことでログ出力する。  ログはdmesgなどのコマンドでみることができる。  #include <linux/init.h>  static int test\_driver\_init(void)  {  printk(KERN\_INFO "test\_driver loaded\n");  return 0;  }  static void test\_driver\_exit(void)  module\_init()/module\_exit()  マクロに関数を記載することで、ドライバがロード時、アンロード時に関数が実行されるようになる  {  printk(KERN\_INFO "test\_driver unloaded\n");  }  module\_init(test\_driver\_init);  module\_exit(test\_driver\_exit);  MODULE\_LICENSE("GPL v2"); |

kernel/drivers/hisho/Makefile

|  |
| --- |
| obj-**y** += test\_driver.o |

# ビルド手順

## UbuntuPC側

1. Android環境のビルド

|  |
| --- |
| docker run -it --rm -v [ANDROID\_BUILD\_TOP]:/src android-build-trusty  ※[ANDROID\_BUILD\_TOP]はビルドするソースコードが格納されているパス  cd /src  source build/envsetup.sh  lunch msm8916\_64-userdebug  make -j2 |& tee make\_$(date +%m%d%H%M).log  ※-j2はUbuntuマシンのCPU数を超えないようにすること |

上記をスクリプト化しているため、以下を実行する

|  |
| --- |
| ./start\_docker.sh  cd src  ./mk\_kernel\_userdebug.sh |

# DragonBoardへのソフト書き込み手順

## WindowsPC側の手順

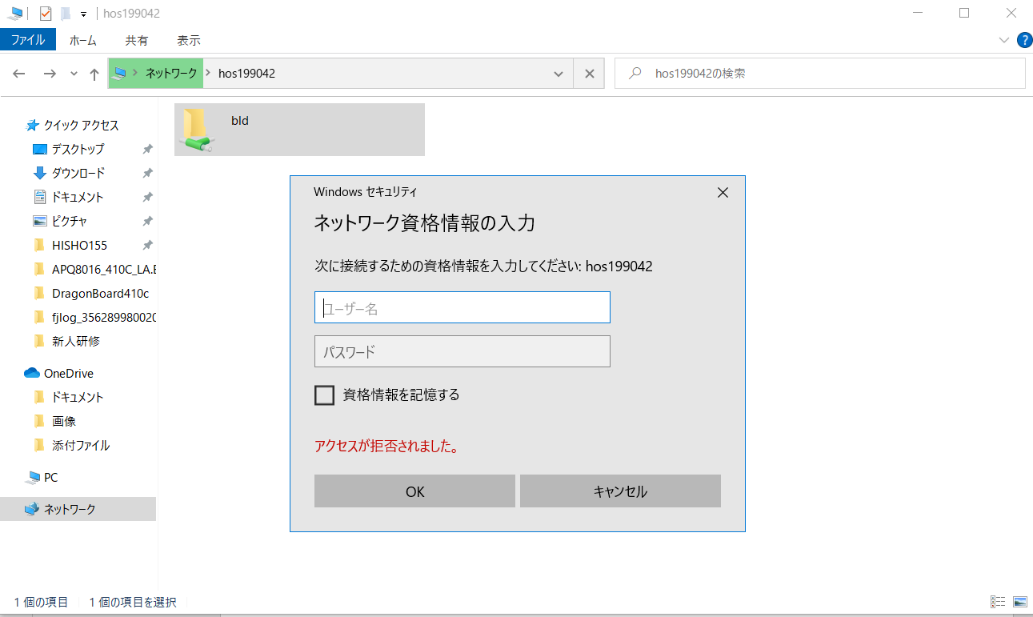
1. WindowsPCをUbuntuPCと同じネットワークに接続する
2. エクスプローラーを開き、アドレス欄に

\\hisho021

と入力

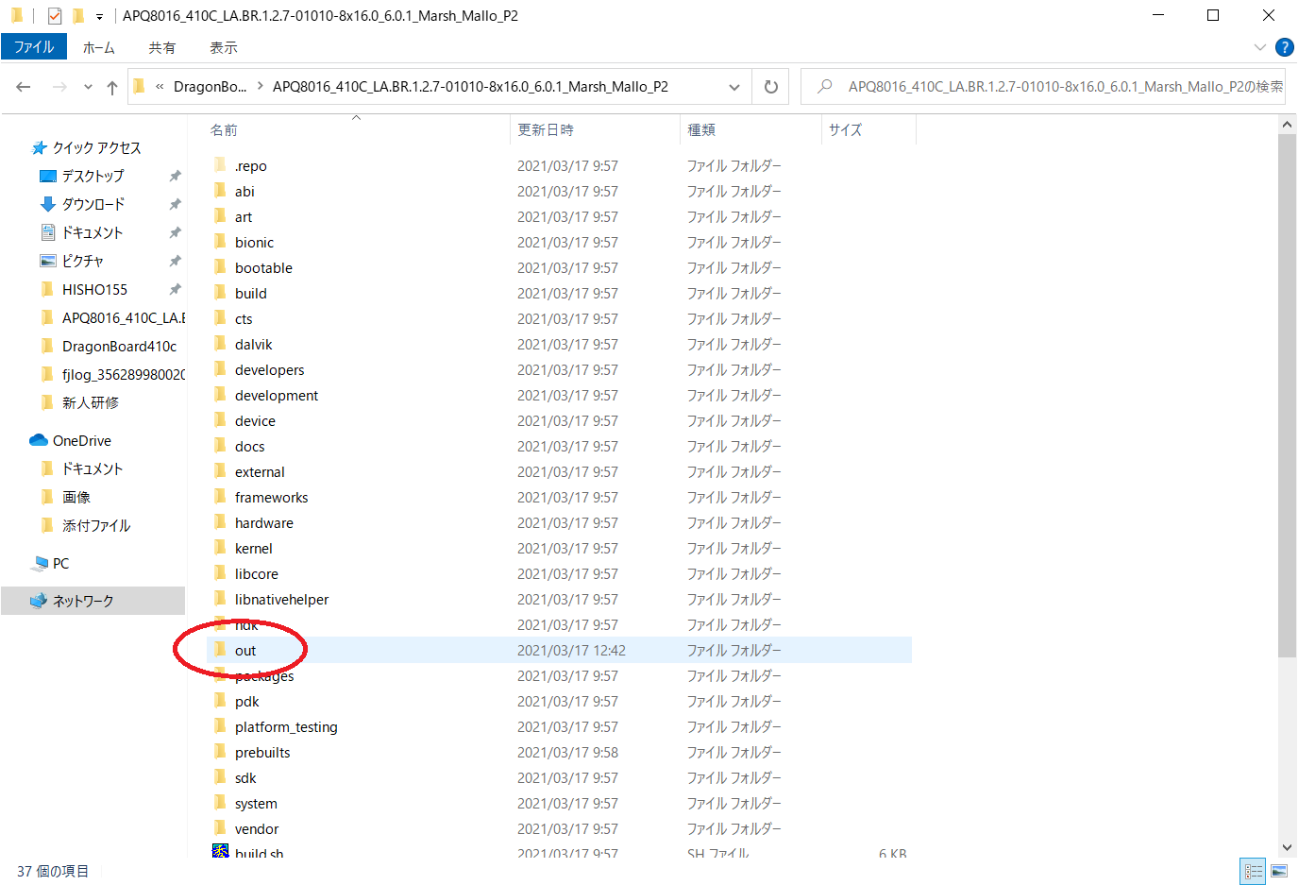
1. あらかじめ作成した共有フォルダがあるので開く

開いた際にユーザー名とパスワードを求められる場合は先生役に訊くこと

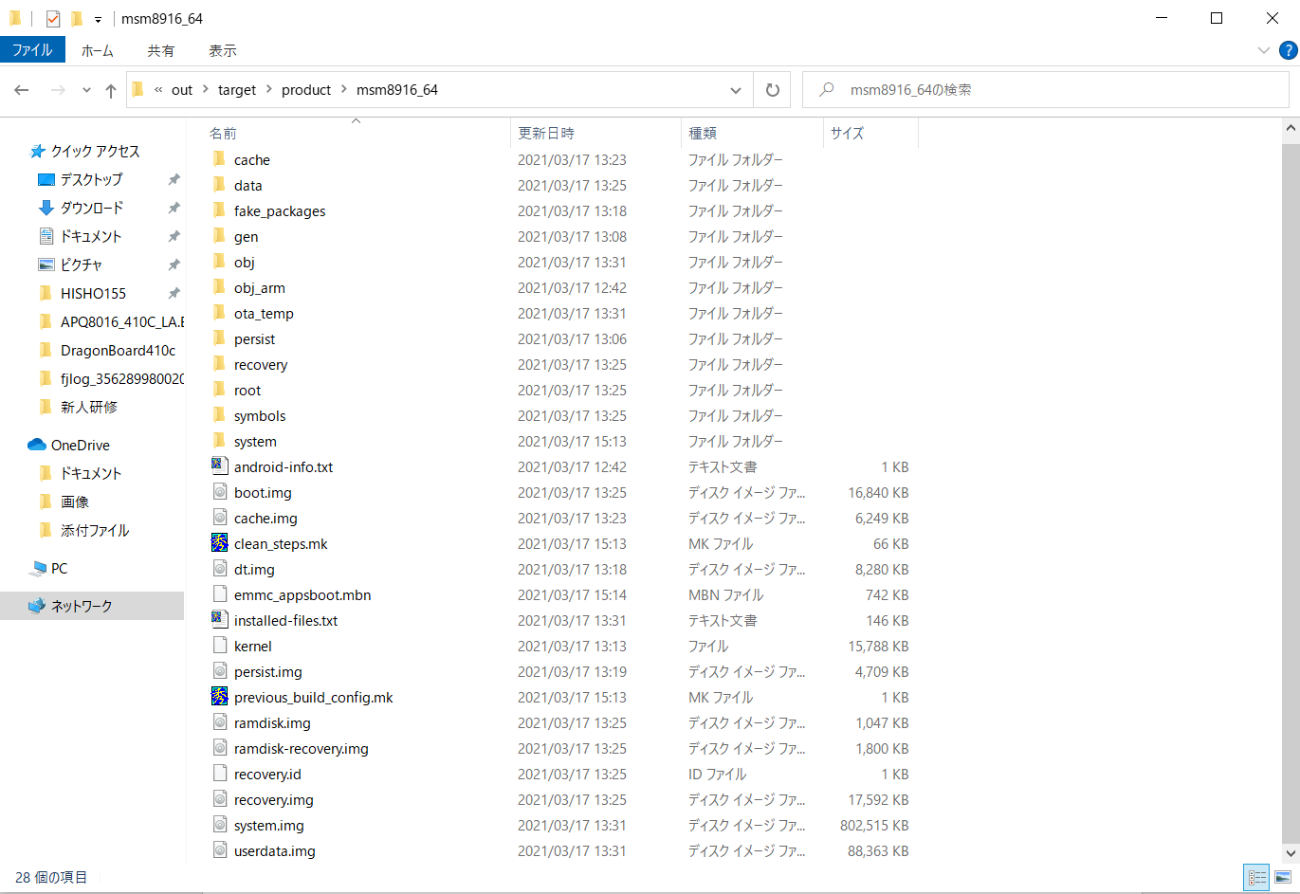


1. Androidのソースコードの環境を参照する

outフォルダ内にビルドした生成物が格納されている



1. 成果物を確認する



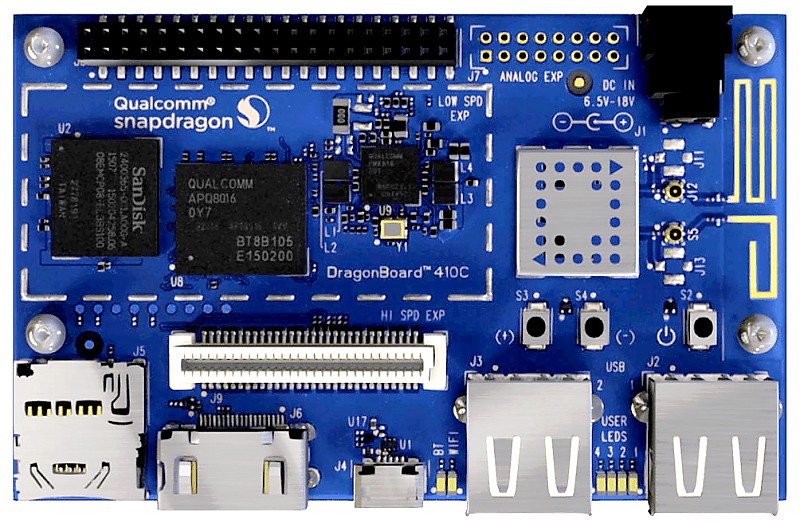
out\target\product\msm8916\_64に

以下のファイルが存在していることを確認する

|  |  |
| --- | --- |
| ファイル名 | 中身 |
| boot.img | Linux Kernelのビルド生成物 |
| recovery.img | Recoveryのビルド生成物 |
| emmc\_appsboot.mbn | bootloader(aboot)のビルド生成物 |
| system.img | Androidのビルド生成物 |
| persist.img | ベンダー固有の設定領域の空データ |
| cache.img | アプリのキャッシュデータ保存領域の空データ |
| userdata.img | ユーザーデータ保存領域の空データ |

1. DragonBoardにACアダプタを接続し、WindowsPCとUSBケーブルで接続する

ACアダプタ接続先



電源キー

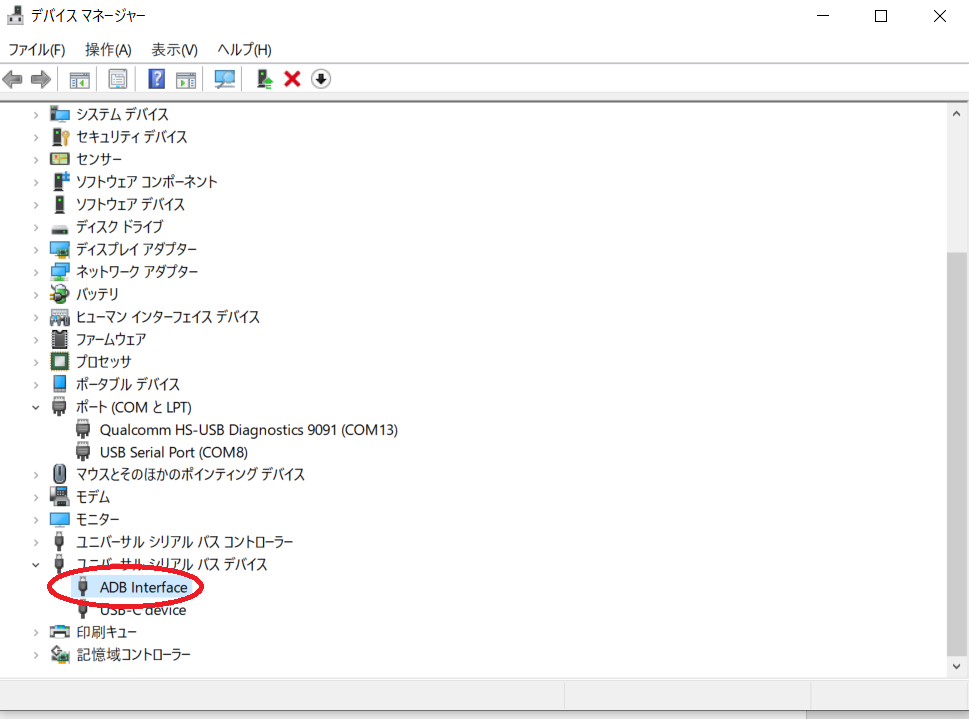
USBケーブル接続先

1. デバイスマネージャーからadbデバイスが見えたら、

コマンドプロンプトから

adb reboot bootloader

を実行してDragonBoardをfastboot（書き込み用のモード）に移行させる



1. デバイスマネージャーからAndroid Bootloader Interfaceが見えたら、

fastbootコマンドを使用してDragonBoardへビルドの成果物を書き込む

fastboot flash boot boot.img

※今回の研修では以下は不要

fastboot flash recovery recovery.img

fastboot flash aboot emmc\_appsboot.mbn

fastboot flash system system.img

fastboot flash cache cache.img

fastboot flash userdata userdata.img

XXX.imgは成果物のフルパスを入れること

書き込み成功後、

fastboot reboot

にて再起動させる

# DragonBoardのデバッグ方法

## ADBを使用する方法

DragonBoardとWindowsPCをUSBケーブルで接続し、

コマンドプロンプトから

adb shell

を実行する

logcat

と実行するとAndroidのログが確認できる

dmesg

または

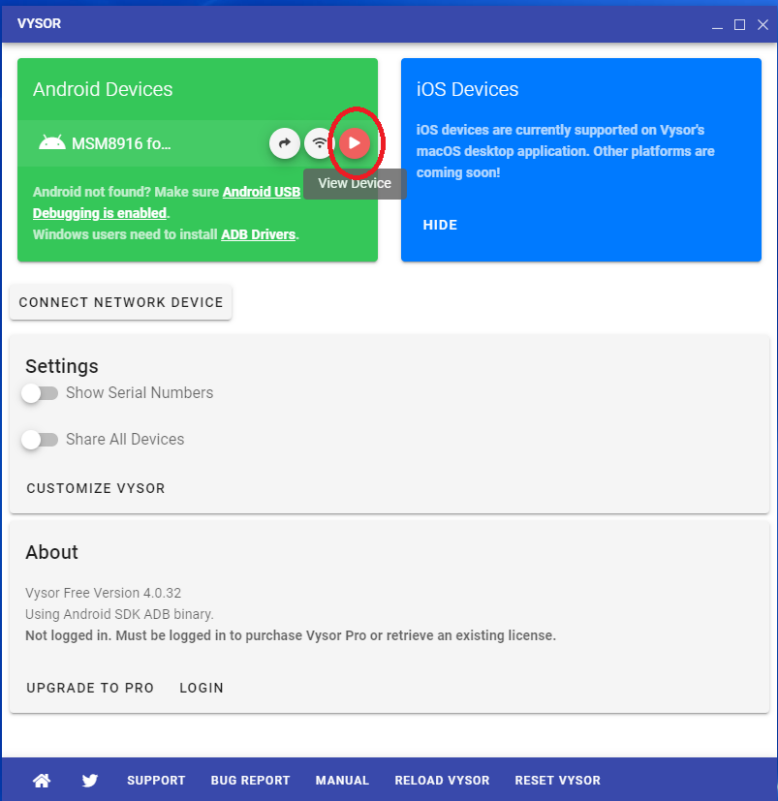
logcat -b kernel

と実行するとkernel側のログが確認できる

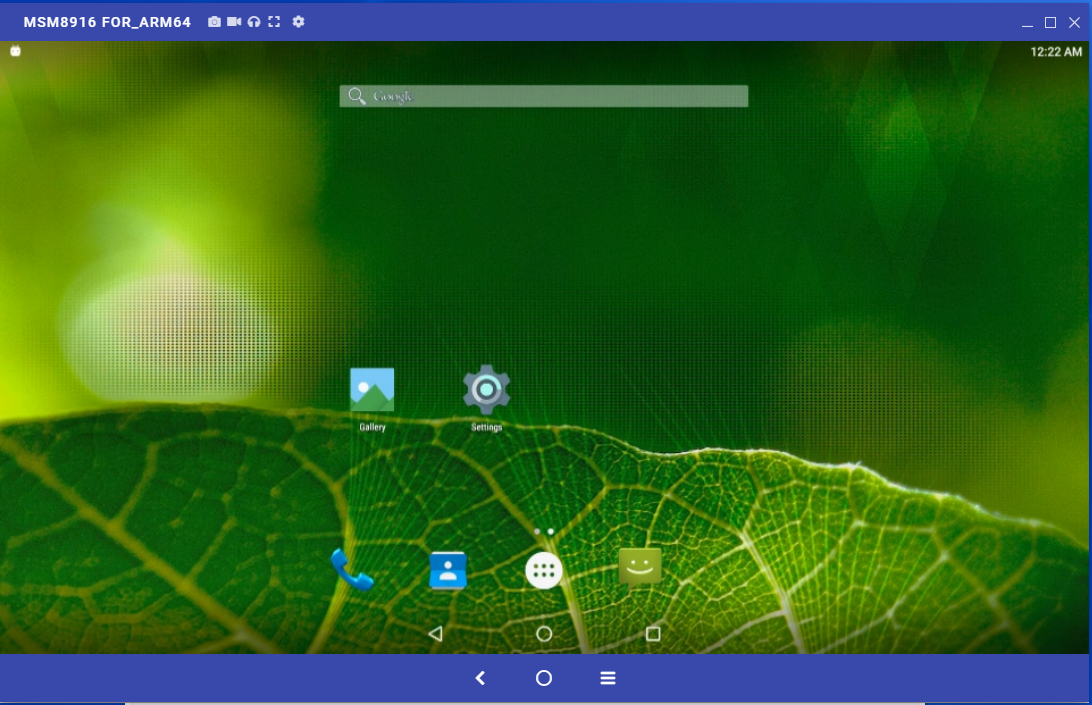
lsやcdなどの基本的なLinuxのコマンドが使用できるため、ファイルシステムを介して様々なデバッグを行うことができる

## Vysorを使用する方法

DragonBoardとWindowsPCをUSBケーブルで接続し、Vysorを起動してAndroid Devicesへ接続する



マウス・キーボードを使用してAndroidの画面を操作できる



# 基本設計

## 調査・検討

制御仕様書からドライバに必要な処理を挙げだし、実現方法を検討する。

・ボタン

[接続されているGPIOの設定]…pinctrlを使用して設定。

|  |
| --- |
| pinctrlとは…  SoC等のI/O pinをLinux から設定するための機能。Device Treeに設定を記述し、ドライバから設定を行う  Device Treeとは…  デバイスのベースアドレスや、クロック、割り込み番号といった、ハードウェア固有のプロパティをカーネルから分離しデバイスドライバの再利用性を高めることが目的の機能。Device Treeに記載したプロパティをデバイスドライバから読み込んで使用する。 |

[GPIOの割り込みを受ける]…irqに登録。割り込みハンドラは専用スレッド上で実行させる。sleep中でも割り込みを検出できるよう、wakeup設定を行う。

|  |
| --- |
| IRQとは…  Interrupt ReQuest。割り込み要求。割り込み要求は、キーボードやマウスなどの外部デバイスからCPUを呼び出すときに発生する。呼び出されたCPUは要求を受け付けるかどうかを判断して、受け付ける場合は、現在実行中の処理を中断して要求された処理（割り込みハンドラ）を優先して行う。割り当てられたIRQの一覧は以下を読み出すことで確認できる  /proc/interrupts  wakeup設定とは…  CPUがsleep中に割り込みが発生した場合にCPUを起床させて割り込み処理を行えるようにする設定 |

|  |
| --- |
| スレッドとは…  プロセス内で命令を逐次実行する部分であり、CPUコアを利用する単位。  プロセスとは…  実行中のプログラムのことで、1つのプロセスには、1つのメモリ空間が割り当てられ、実行するプログラム、変数や演算結果を格納するスタック領域などが含まれる。これをコンテキストと呼び、プロセス・スレッドが切り替わる事をコンテキストスイッチと呼ぶ。  なお、割り込み処理は実行中のプロセス・スレッドのコンテキストを間借りして実行されるため、割り込み処理中にコンテキストスイッチおよびそれが行われる可能性がある処理を行うことは禁止されている。  プロセス・スレッドの一覧は以下のコマンドで確認できる。  ps -A |

[チャタリング対策]…専用スレッド上で実行される割り込みハンドラで行う。gpiolibを使用してGPIO値の読み出しとボタン状態の判定を行う。

|  |
| --- |
| gpiolibとは…  GPIOを操作するための機能 |

・LED

[接続されているGPIOの設定]…pinctrlを使用して設定

[LEDの点滅]…タイマ、ワークキューなどの実現方法も考えられるが、専用スレッドを生成して実行する方法とする。

[点滅間隔をユーザースペースから変更可能とする]…モジュールパラメータを使用する

|  |
| --- |
| ユーザースペースとカーネルスペース…  アプリケーション（OS上で動作するソフトウェア）が使用するメモリ領域のことであり、カーネルスペースとは、カーネルが使用するメモリ領域のことを指す。  モジュールパラメータとは…  カーネルプログラム内部の指定した変数をパラメータとして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 |

[sleep防止]…wakelockを使用してLED点滅中はsleepしないよう制御する

|  |
| --- |
| wake lockとは…  CPUがサスペンド状態になるのを抑止する機能 |

waitキューを用いて割り込みハンドラとの待ち合わせを行う

|  |
| --- |
| プロセス・スレッドの状態は、実行可能状態と待機状態（休止状態）があり、実行可能状態のプロセスはスケジューリングの対象となり、順番に実行権が与えられる。待機状態のプロセス・スレッドは、何らかの事象を待ち合わせている状態であり、条件が整うまでそれ以上の処理を継続できずスケジューリングの対象とならない。待機の対象となる可能性のあるカーネルオブジェクト（ファイルや、実ページ、端末など）は、それぞれwaitキューを用意し、待機状態に遷移したプロセスは、ある事象の待ち合わせ用に用意されているwaitキューに登録される。  待機状態のプロセスにはシグナルを受け付ける待機状態とシグナルを受け付けない待機状態の2種類がある。 |

## モジュール構成図

ドライバの中にはどのような処理、機能があり、どんな機能が関連しているか。

大まかな構成図を描く。

test driver

IRQスレッド

LED点滅スレッド

irq

pinctrl

gpiolib

点滅間隔

モジュールパラメータ

ユーザー空間

カーネル空間

wakelock

waitキュー

# 機能設計

## 外部Ｉ/F設計

test driver外へ提供するI/Ｆを決める。

### モジュールパラメータ

|  |  |
| --- | --- |
| パス | /sys/module/test\_driver/parameters/blink\_interval |
| 権限 | 所有者：read/write  所有グループ：read  その他：read |
| 所有者・グループ | root ・ root  ※今回の研修ではアプリ側の実装がないため、具体的な所有者は決めず、rootとしておく。 |

|  |
| --- |
| モジュールパラメータは/sys/module/[ドライバ名]/parameters/[パラメータ名]に生成される  Linuxではファイルやディレクトリ一個一個に許可属性を与え、管理出来るようになっている。これを「パーミッション」や「権限」という。  それぞれ「所有者」「グループ」「その他のユーザー」にパラメータ（「読み込み」「書き込み」「実行」）を割り振りが可能となっている。 |

読み出し時

点滅間隔のms単位で返す。

書き込み時

ms単位の点滅間隔を書き込むことで点滅間隔を変更できる。今回の研修では複雑な条件を排除するため、LED点滅中であっても変更可能とする。

## 内部I/F設計

test driver内の機能同士でI/Fを決めることがあれば、検討する。

今回の研修では不要。

# 詳細設計

## フローチャート

test driver内の処理フローを描いてく

### 起動時

START

wakeup設定

waitキュー初期化

END

必要な管理情報の確保

pinctrl設定

LED点滅スレッド

生成

irq設定

### IRQスレッド

START

wakelock

10回繰り返し

ボタンGPIO値取得

①

5ms wait

②

GPIO値を0とする

③

点滅の設定値を更新

LED点滅スレッド

起床

END

1. 3回連続で同じ値が読めたか？
2. 値を確定できたか？
3. 押し検出か？

Yes

No

Yes

No

Yes

No

### LED点滅スレッド

START

スレッド停止要求がない間繰り返し

①

wait event

LED点灯

END

1. LED点滅の設定値がONか？
2. スレッド停止要求がない、且つ点滅の設定値がONの間繰り返し

Yes

No

wake unlock

点滅時間分wait

LED消灯

点滅時間分wait

②

## シーケンス図

複数のスレッドが関連する場合など、シーケンス図を描いた方が分かり易い場合は記載する

irq

IRQスレッド

LED点滅スレッド

ボタン状態チェック

LED点滅開始

ボタン押しによる

割り込み発生

wakeup

ボタン押しによる

割り込み発生

ボタン状態チェック

LED点滅停止

wakeup

wakeup

wakeup

# コーディング

## プラットフォームドライバの登録

### デバイスツリーの編集

kernel\arch\arm64\boot\dts\qcom\apq8016-sbc.dtsi

の一番下に以下を追記する

|  |
| --- |
| &tlmm\_pinmux {  test\_pin {  label = "test\_pin";  ○LED制御用GPIO36の設定  qcom,pin-func:0で通常のGPIOとして使用する  pin\_high：点灯時の設定値  drive-strength：2mA  bias-disable:no pull設定  output-high：high出力設定  pin\_low：消灯時の設定値  drive-strength：2mA  bias-disable:no pull設定  output-high：low出力設定  qcom,pins = <&gp 36>;  qcom,pin-func = <0>;  qcom,num-grp-pins = <1>;  test\_pin\_high: pin\_high {  drive-strength = <2>;  bias-disable;  output-high;  };  test\_pin\_low: pin\_low {  drive-strength = <2>;  bias-disable;  output-low;  };  };  }; |

|  |
| --- |
| &tlmm\_pinmux {  ○ボタン制御用GPIO13の設定  qcom,pin-func:0で通常のGPIOとして使用する  button\_active：使用時の設定値  drive-strength：2mA  bias-disable:no pull設定  input-enable：入力設定  test\_button {  label = "test\_button";  qcom,pins = <&gp 13>;  qcom,pin-func = <0>;  qcom,num-grp-pins = <1>;  test\_button\_active: button\_active {  drive-strength = <2>;  bias-disable;  input-enable;  };  };  }; |

|  |
| --- |
| &soc {  ○ドライバの設定  pinctrlに以下の設定情報を登録  init…LED消灯設定、ボタン使用時設定  led-low…LED消灯設定、ボタン使用時設定  led-high…LED点灯設定、ボタン使用時設定  ボタン制御用GPIO番号  button-gpios  ボタンのチャタリング対策で使用する固定値  check-interval  check-match-count  check-max-count  test\_driver {  compatible = "hisho,Testdrv";  pinctrl-names = "init", "led-low", "led-high";  pinctrl-0 = <&test\_pin\_low &test\_button\_active>;  pinctrl-1 = <&test\_pin\_low &test\_button\_active>;  pinctrl-2 = <&test\_pin\_high &test\_button\_active>;  button-gpios = <&msm\_gpio 13 0x0>;  check-interval = <5>;  check-match-count = <5>;  check-max-count = <20>;  status = "okay";  };  }; |

### ドライバの編集

kernel\drivers\hisho\test\_driver.c

に赤字部分を追加

|  |
| --- |
| #include <linux/module.h>  #include <linux/init.h>  #include <linux/platform\_device.h>  static int test\_probe(struct platform\_device \*pdev)  {  int ret = 0;  printk(KERN\_INFO "test\_probe was called %d\n", ret);  test\_probe…ドライバ登録時に実行される関数  test\_remove…ドライバ削除時に実行される関数  return ret;  }  static int test\_remove(struct platform\_device \*pd)  {  printk(KERN\_INFO "test\_remove was called.\n");  return 0;  } |

|  |
| --- |
| static const struct of\_device\_id test\_dt\_match[] = {  .compatibleに設定する文字列はデバイスツリーに記載したcompatibleの値と一致させること  {.compatible = "hisho,Testdrv"},  {},  };  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,  }; |

platform\_driver\_registerでプラットフォームドライバの登録を行う

|  |
| --- |
| static int test\_driver\_init(void)  {  printk(KERN\_INFO "test\_driver loaded\n");  platform\_driver\_register(&test\_platform\_driver);  return 0;  }  static void test\_driver\_exit(void)  {  printk(KERN\_INFO "test\_driver unloaded\n");  platform\_driver\_unregister(&test\_platform\_driver);  } |

### 動作確認

端末起動後、

adb shell dmesg > log.txt

を実行し、PCに保存されたlog.txtを開く。

test\_driver loaded

および

test\_probe was called

のログが出力されていること

## 起動時の初期化処理の実装

### ドライバの編集

kernel\drivers\hisho\test\_driver.c

に赤字部分を追加

|  |
| --- |
| #include <linux/module.h>  #include <linux/init.h>  #include <linux/platform\_device.h>  #include <linux/interrupt.h>  #include <linux/gpio.h>  #include <linux/of.h>  #include <linux/of\_gpio.h>  #include <linux/pm.h>  #include <linux/kthread.h>  #include <linux/delay.h>  struct test\_driver\_info {  struct platform\_device \*pdev;  struct pinctrl \*pctrl;  struct pinctrl\_state \*pins\_init;  ドライバに必要な情報を構造体にまとめる  struct pinctrl\_state \*pins\_low;  struct pinctrl\_state \*pins\_high;  struct task\_struct \*thread;  wait\_queue\_head\_t queue\_wait;  bool led\_switch;  int button\_gpio;  u32 check\_interval;  u32 check\_match\_count;  u32 check\_max\_count;  }; |

|  |
| --- |
| static int \_test\_pinctrl\_init(struct device \*dev, struct test\_driver\_info \*info)  {  int ret = 0;  info->pctrl = devm\_pinctrl\_get(dev);  デバイスツリーからpinctrlの情報を取得  if (IS\_ERR(info->pctrl)) {  ret = PTR\_ERR(info->pctrl);  goto out;  }  info->pins\_init = pinctrl\_lookup\_state(info->pctrl, "init");  if (IS\_ERR(info->pins\_init)) {  デバイスツリーからinitのpinctrl設定を取得  ret = PTR\_ERR(info->pins\_init);  goto out;  }  info->pins\_low = pinctrl\_lookup\_state(info->pctrl, "led-low");  if (IS\_ERR(info->pins\_low)) {  ret = PTR\_ERR(info->pins\_low);  デバイスツリーからled-lowのpinctrl設定を取得  goto out;  }  info->pins\_high = pinctrl\_lookup\_state(info->pctrl, "led-high");  if (IS\_ERR(info->pins\_high)) {  デバイスツリーからled-highのpinctrl設定を取得  ret = PTR\_ERR(info->pins\_high);  goto out;  }  ret = pinctrl\_select\_state(info->pctrl, info->pins\_init);  if (ret) {  initのpinctrl設定を実行  goto out;  }  out:  return ret;  } |

|  |
| --- |
| static int test\_probe(struct platform\_device \*pdev)  {  int ret = 0;  struct device \*dev = &pdev->dev;  ドライバに必要な情報を格納する領域を確保  struct test\_driver\_info \*info;  info = devm\_kzalloc(dev, sizeof(struct test\_driver\_info), GFP\_KERNEL);  if (info == NULL) {  ret = -ENOMEM;  goto out;  }  info->pdev = pdev;  wakelockが使用できるように初期化  device\_init\_wakeup(dev, true);  init\_waitqueue\_head(&info->queue\_wait);  waitキューの初期化  ret = \_test\_pinctrl\_init(dev, info);  if (ret)  goto out;  out:  return ret;  } |

### 動作確認

端末起動後、

adb shell

cat /d/pinctrl/1000000.pinctrl/pinmux-pins

を実行し、出力結果に以下が含まれていること

pin 13 (gp-13): test\_driver.69 (GPIO UNCLAIMED) function test\_button-func group test\_button

pin 36 (gp-36): test\_driver.69 (GPIO UNCLAIMED) function test\_pin-func group test\_pin

cat /d/wakeup\_sources

を実行し、出力結果に

test\_driver.69

が含まれていること

## LED点滅スレッドの登録

### ドライバの編集

kernel\drivers\hisho\test\_driver.c

に赤字部分を追加

|  |
| --- |
| static int test\_led\_blink(void \*arg)  {  struct test\_driver\_info \*info = (struct test\_driver\_info \*)arg;  struct device \*dev = &info->pdev->dev;  スレッドが停止されるまでループ  while (!kthread\_should\_stop()) {  wake lock解除  pm\_relax(dev);  wait\_event\_interruptible(info->queue\_wait, info->led\_switch);  }  waitキューに登録して待つ  return 0;  } |

|  |
| --- |
| static int \_test\_init\_led\_thread(struct test\_driver\_info \*info)  {  int ret = 0;  スレッドの生成  info->thread = kthread\_run(test\_led\_blink, info, "test\_led\_blink");  if (IS\_ERR(info->thread)) {  ret = PTR\_ERR(info->thread);  goto out;  }  out:  return ret;  } |

|  |
| --- |
| 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;  }  info->pdev = pdev;  device\_init\_wakeup(dev, true);  init\_waitqueue\_head(&info->queue\_wait);  ret = \_test\_pinctrl\_init(dev, info);  if (ret)  goto out;  ret = \_test\_init\_led\_thread(info);  if (ret)  goto out;  out:  return ret;  } |

### 動作確認

端末起動後、

adb shell

ps

を実行し、出力結果に以下が含まれていること

test\_led\_blink

## 割り込みの登録

### ドライバの編集

kernel\drivers\hisho\test\_driver.c

に赤字部分を追加

|  |
| --- |
| static irqreturn\_t test\_button\_irqt(int irq, void \*arg)  {  struct test\_driver\_info \*info = (struct test\_driver\_info \*)arg;  struct device \*dev = &info->pdev->dev;  printk(KERN\_INFO "test\_button\_irqt\n");  wake lock設定  pm\_stay\_awake(dev);  wake\_up(&info->queue\_wait);  LED点滅スレッドを起こす  return IRQ\_HANDLED;  } |

|  |
| --- |
| static int \_test\_init\_irq(struct device \*dev, struct test\_driver\_info \*info)  {  int ret = 0;  struct device\_node \*np = dev->of\_node;  enum of\_gpio\_flags flags = OF\_GPIO\_ACTIVE\_LOW;  デバイスツリーから各種値を読み出す  ret = of\_property\_read\_u32(np, "check-interval", &info->check\_interval);  if (ret) {  goto out;  }  ret = of\_property\_read\_u32(np, "check-match-count", &info->check\_match\_count);  if (ret) {  goto out;  }  ret = of\_property\_read\_u32(np, "check-max-count", &info->check\_max\_count);  if (ret) {  goto out;  }  info->button\_gpio = of\_get\_named\_gpio\_flags(np, "button-gpios", 0, &flags);  if (!gpio\_is\_valid(info->button\_gpio)) {  ret = -EINVAL;  goto out;  }  割り込みの登録  ret = devm\_request\_threaded\_irq(dev,  gpio\_to\_irq(info->button\_gpio),  NULL, test\_button\_irqt,  IRQF\_TRIGGER\_RISING | IRQF\_TRIGGER\_FALLING | IRQF\_ONESHOT,  "test\_button", info);  IRQF\_TRIGGER\_RISING…立ち上がり時に割り込みが発生する指定  IRQF\_TRIGGER\_FALLING…立ち下がり時に割り込みが発生する指定  IRQF\_ONESHOT…割り込みハンドラ実行中は割り込みを無効化する指定  if (ret) {  goto out;  }  enable\_irq\_wake(gpio\_to\_irq(info->button\_gpio));  out:  sleep中に割り込みが発生した際にCPUを起こす設定  return ret;  } |

|  |
| --- |
| 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;  }  info->pdev = pdev;  device\_init\_wakeup(dev, true);  init\_waitqueue\_head(&info->queue\_wait);  ret = \_test\_pinctrl\_init(dev, info);  if (ret)  goto out;  ret = \_test\_init\_led\_thread(info);  if (ret)  goto out;  ret = \_test\_init\_irq(dev, info);  if (ret)  goto out;  out:  return ret;  } |

### 動作確認

端末起動後、

adb shell

cat /proc/interrupts

を実行し、出力結果に

msm\_tlmm\_irq test\_button

が含まれていること

adb root

adb shell

cat /proc/kmsg

を実行し、ログを確認しながらボタンを押す・放す度に

test\_button\_irqt

のログが出力されること

## モジュールパラメータの登録

### ドライバの編集

kernel\drivers\hisho\test\_driver.c

に赤字部分を追加

|  |
| --- |
| static unsigned int blink\_interval = 1000;  module\_param(blink\_interval, uint, 0644); |

### 動作確認

端末起動後、

adb root

adb shell

ls -l /sys/module/test\_driver/parameters/blink\_interval

を実行し、設定したパーミッションになっていること

cat /sys/module/test\_driver/parameters/blink\_interval

を実行し、

1000

が出力されること

echo 500 > /sys/module/test\_driver/parameters/blink\_interval

cat /sys/module/test\_driver/parameters/blink\_interval

を実行し、

500

が出力されること

## 割り込み処理の実装

### ドライバの編集

kernel\drivers\hisho\test\_driver.c

のtest\_button\_irqt関数の中身を実装する

#### ヒント

* ボタンのGPIO値取得方法

statusの値が0ならLow、1ならHighを表す

|  |
| --- |
| unsigned int status = 0;  status = gpio\_get\_value\_cansleep(info->button\_gpio); |

* 制御に必要な値

info->check\_max\_count：スキャンする最大回数(20)が格納されている

info->check\_match\_count：確定と判断する連続一致回数(3)が格納されている

info->check\_interval：スキャン間隔(5)が格納されている

* 一定時間waitする方法

msleep();の引数に待ちたい時間(単位ms)を設定する

* 点滅・停止状態の管理

info->led\_switch

に点滅要求(1)なのか、停止要求(0)なのかを格納する

## LED点滅スレッドの実装

### ドライバの編集

kernel\drivers\hisho\test\_driver.c

のtest\_led\_blink関数の中身を実装する

#### ヒント

* LED点灯設定

以下の処理を行うことでLEDが点灯する

|  |
| --- |
| pinctrl\_select\_state(info->pctrl, info->pins\_high); |

* LED消灯設定

以下の処理を行うことでLEDが消灯する

|  |
| --- |
| pinctrl\_select\_state(info->pctrl, info->pins\_low); |

* 制御に必要な値

blink\_intervalに点滅間隔(1000)が格納されている

* 一定時間waitする方法

msleep();の引数に待ちたい時間(単位ms)を設定する

* 点滅・停止状態の管理

info->led\_switch

に点滅要求(1)なのか、停止要求(0)なのかが格納されている