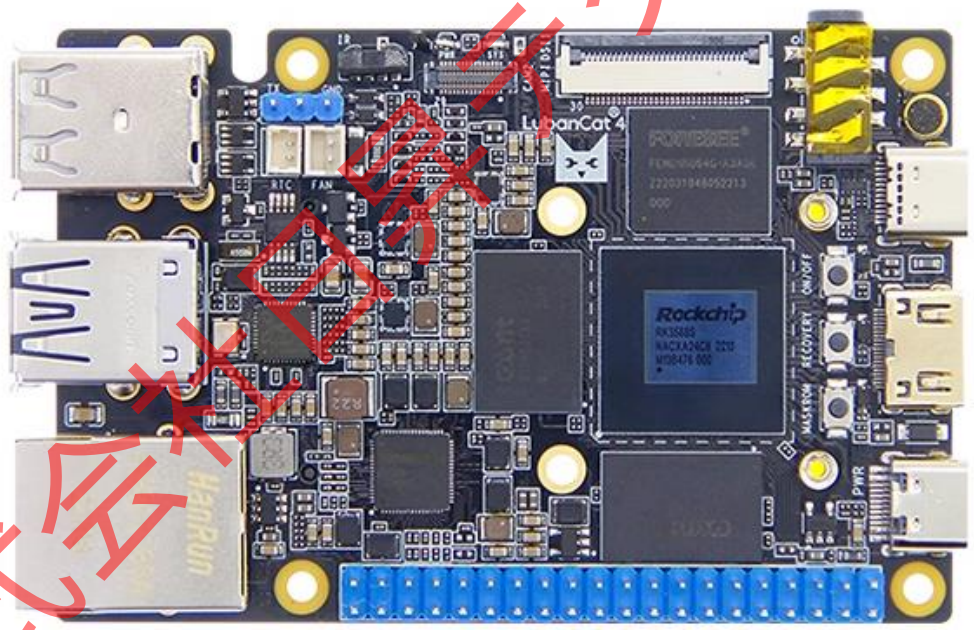
組み込み Linux ドライバ開発実践ガイド

株式会社日昇テクノロジー

https://www.csun.co.jp

info@csun.co.jp

作成日 2024/06/04



株式会社日昇テクノロジー

copyright@2024

- **修正履歴**

NO	バージョン	修正内容	修正日
1	Ver1.0	新規作成	2024/06/04

※ この文書の情報は、文書を改善するため、事前の通知なく変更されることがあります。最新版は弊社ホームページからご参照ください。「https://www.csun.co.jp」

※ (株)日昇テクノロジーの書面による許可のない複製は、いかなる形態においても厳重に禁じられています。

株式会社日昇テクノロジー

目　次

株式会社日昇テクノロジー

株式会社日昇テクノロジー

株式会社日昇テクノロジー

株式会社日昇テクノロジー

株式会社日昇テクノロジー

株式会社日昇テクノロジー

株式会社日昇テクノロジー

株式会社日昇テクノロジー

株式会社日昇テクノロジー

株式会社日昇テクノロジー

株式会社日昇テクノロジー

株式会社日昇テクノロジー

株式会社日昇テクノロジー

株式会社日昇テクノロジー

株式会社日昇テクノロジー

文書の説明

この文書は、ある程度の Linux の基礎を持ち、Lubancat_RK ボードを基に Linux ドライバ開発を学びたいと考えている組み込みソフトウェアエンジニアや学生などを主な対象としています。

本文書に付随するハードウェアプラットフォームは RK プロセッサを採用し、技術の原理をよりよく理解することで技術をより良く活用する原則に基づいて記述されています。内容は 4 部分に分かれており、内容は順を追って進み、Linux ドライバーでよく使用されるフレームワークに関わるものです。このチュートリアルを通じて、学習者が Linux ドライバ開発のいくつかの概念を確立し、学習者が迷わずに進めるようにすることを願っています。

本文書は以下の部分で構成され、解説されます：

・第一部分では、Linux ドライバの基礎知識です。Linux ドライバ開発の初心者として、ドライバ開発には多くの概念が必要であり、Linux ドライバの基礎知識は出発点となり、初心者が Linux ドライバ開発の基本的な面貌を構築できるようになります。

・第二部分は、Linux ドライバフレームワークです。Linux ドライバの基礎知識を身につけたことで、Linux ドライバ開発の基本概念を理解したことでしょう。これらの基本概念から出発して、正式に Linux ドライバの中でよく見られる各種フレームワーク、例えば GPIO サブシステム、I2C サブシステム、割り込みサブシステムなどを体験できます。これらのドライバフレームワークを通じて、Linux ドライバが各種デバイスの抽象概念、包装形式、そしてフレンドリーな開発インターフェイスを垣間見ることができます。

・第三部分は、Linux ドライバの上級知識です。Linux ドライバの基礎知識に基づき、デバイスの性能向上と機能拡張に伴い、Linux ドライバ開発でさらに注意を払うべきいくつかの詳細について学ぶ必要があります。

・第四部分は、追加されたドライバモジュールです。ボード上の多くのドライバモジュール、およびその他のドライバ関連の知識です。

株式会社日昇テクノロジー

本チュートリアルで使用する開発環境の説明は以下の通りです：

・PC システム Linux：Ubuntu20.04。Windows 上で VirtualBox 仮想マシンを使用したインストールの説明をしていますが、VMware などの他の仮想マシンを使用しても構いません。

• 開発ボードシステム：Lubancat 汎用イメージ（Debian10、Ubuntu20.04 など）を使用します。

第 1 章 ドライバセクションの実験環境構築

本章の主な目的は、ドライバセクションの実験環境を構築することで、後続の章では実験環境の構築に多くのページを費やすことなく、デバイスドライバの原理について主に説明します。このセクションの内容は多岐にわたる知識点が関連しており、ある程度の基礎がないと理解できない内容もあります。一部の知識点が理解できない場合は飛ばしても構いません。関連知識に触れる機会があれば、後で学習することができます。

まず、プログラムは最終的にボード上で実行されることを理解する必要があります。ボード上で直接コンパイルするか、または PC 上でクロスコンパイラを使用してコンパイルすることができます。ソースコード、カーネルソースコードまたは該当するカーネルヘッダーファイル（Kernel Headers）のダウンロード、ソースコードのコンパイル、ドライバモジュールやデバイスツリーなどのコンパイルが必要です。最終的に、ドライバモジュールとデバイスツリーを開発ボード上で実行するためにコピーします。

また、ドライバモジュールは独立した機能を持つプログラムであり、単独でコンパイルは可能ですが、単独で実行することはできません。実行時には、カーネルの一部としてカーネル空間で実行されるため、カーネルにリンクされます。そのため、特定のカーネルバージョン上で書いたカーネルモジュールを実行したい場合は、そのカーネルバージョン上でコンパイルする必要があります。

ヒント：この章では、コンパイルしたカーネルを焼き付ける必要はありません。カーネルのコンパイルは開発ボード上でも個人の PC 上でも行うことができ、ドライバプログラムの補助的なコンパイル

株式会社日昇テクノロジー

のためです。また、カーネルヘッダーファイルの deb パッケージをコンパイル出力し、インストールすることで、同様にドライバモジュールをコンパイルすることができます。カーネルを更新する必要がある場合は、イメージの構築とデプロイメントの章を参照してください。

1.1 コンパイル環境の構築

カーネルのコンパイルは、PC の仮想マシン上でクロスコンパイルを行うか、ボード上でローカルコンパイルを行うことができます。PC の仮想マシンの性能はボードよりはるかに優れているため、一般的には仮想マシン上でのコンパイルを推奨します。しかし、仮想マシンを使用したくない場合や、コンパイル時間が長くても構わないと考える場合は、ボード上で直接開発しても構いません。

PC 上では、VirtualBox や VMWare を使用して ubuntu の仮想マシンを構築できます。

Ubuntu18.04 または ubuntu20.04 のバージョンを使用することをお勧めします。詳細な構築方法については、LubanCat-RK ボードのアプリケーション開発マニュアルの章を参照してください。

関連するライブラリやツールをインストールしてコンパイル環境を構築するには、以下のコマンドを実行します：

```
sudo apt update
sudo apt install gcc make git bc libssl-dev liblz4-tool device-tree-compiler bison flex u-boot-tools gcc-aarch64-linux-gnu
```

1.2 カーネルソースコードの取得

ボードで使用しているカーネルバージョンは、コマンド uname -a で確認できます。カーネルソースコードの取得には、公式が提供するカーネルソースコードを直接 git クローンするか、Lubancat-SDK のソースコードをダウンロードすることをお勧めします。SDK ソースコードにはカーネルソースコードが含まれています。

株式会社日昇テクノロジー

1.2.1 カーネルソースコードを直接クローン

1.2.2 SDK を通じてカーネルソースコードを取得

オンラインストレージリソース紹介ページにアクセスし、SDK ソースコードの圧縮パッケージを取得します：

SDK **ダウンロード** URL: https://www.dragonwake.com/download/LubanCat4/8-SDK/Linux/LubanCat_Linux_rk3588_SDK_20240510.7z

重要：ソースコードの圧縮パッケージは容量が大きいため、大幅な修正が行われた安定バージョンのみ更新されます。その発行日はイメージの公開日と一致しない可能性があります。ローカルで圧縮パッケージを解凍した後、Github を利用して少量の更新を行うことで、最新バージョンに同期することができます。

以下のプロセスは LubanCat_RK358x_Linux_SDK を例にしており、実際のファイル名はダウンロードした SDK によります。

```
# 7z 圧縮ツールのインストール
sudo apt install p7zip-full
# ユーザーのホームディレクトリに LubanCat_SDK ディレクトリを作成
mkdir ~/LubanCat_SDK
# ダウンロードした SDK ソースコードを LubanCat_SDK ディレクトリに移動、xxx は日付
mv LubanCat_RK358x_Linux_SDK_xxx.7z ~/LubanCat_SDK
# LubanCat_SDK ディレクトリに移動
cd ~/LubanCat_SDK
# SDK 圧縮ファイルの解凍
7z x LubanCat_RK358x_Linux_SDK_xxx.7z
```

株式会社日昇テクノロジー

```
# .repo ディレクトリ下の git リポジトリのチェックアウト

.repo/repo/repo sync -l

# カーネルディレクトリに移動

cd ~/LubanCat_SDK/kernel

# カーネルの更新

git pull

# SDK 全体を更新することもできます

# すべてのソースコードリポジトリを最新バージョンに同期

cd ~/LubanCat_SDK/ && .repo/repo/repo sync -c
```

git pull または repo sync -c を実行時にネットワーク接続のタイムアウトが表示された場合は、github へのアクセスを確認してください。github に正常にアクセスできる場合は、git pull または repo sync -c コマンドを複数回実行して同期を試みてください。github にアクセスできない場合は、ソースコードリポジトリを最新バージョンに同期するステップをスキップしても構いません。

1.3 カーネルのコンパイル

1.3.1 PC 上でのクロスコンパイル（推奨）

前のセクションでコンパイル環境を構築し、ソースコードをダウンロードした後、カーネルソースコードのルートディレクトリに移動し、具体的なボードに合わせて設定ファイルを設定します。

RK3588 ボードユーザーは以下のコマンドでカーネルソースコードをコンパイルします：

```
# 以前に生成されたすべてのファイルと設定をクリア
make mrproper
# lubancat_linux_rk3588_defconfig 設定ファイルをロード、rk3588 はこの設定ファイルを使用
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- lubancat_linux_rk3588_defconfig
# カーネルをコンパイル、クロスコンパイルツールを指定、8 スレッドでコンパイル、スレッド数は PC の性能に応じて自由に設定可能
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- -j8
```

クロスコンパイルツールがない場合や、コンパイルツールのバージョンが一致しない場合は、

株式会社日昇テクノロジー

Lubancat-SDK ソースコードの gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu バージョンのコンパイルツールチェーンを使用することもできます。

以下のコマンドを実行して、コンパイルツールの環境変数を設定してください：

```
# コンパイルツールチェーンの取得
git clone https://github.com/LubanCat/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu.git
# 環境変数をエクスポートする、実際にコンパイルツールチェーンの絶対パスを指定する必要があります
export PATH=/root/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu/bin:$PATH
# コンパイルツールチェーンを確認する、COLLECT_LTO_WRAPPER 変数が指定されたパスであれば、設定が成功しています
aarch64-linux-gnu-gcc -v
```

以上の設定は環境変数を一時的にエクスポートするもので、他のターミナルを開いたり、再起動したりすると、再度環境変数をエクスポートする必要があります。永久に保存するには、エクスポートするコマンドを~/.bashrc ファイルの末尾に記述し、source ~/.bashrc を実行して設定を再読み込みします。

1.3.2 ボード上でカーネルをローカルにコンパイルする

前のセクションでコンパイル環境を構築し、ソースコードをダウンロードした後、カーネルソースコードのルートディレクトリに入り、具体的なボードの設定ファイルに従って設定します。

RK3588 シリーズボードのユーザーは、以下のコマンドを実行してカーネルソースコードをコンパイルしてください。

株式会社日昇テクノロジー

```
# 以前に生成されたすべてのファイルと設定をクリア
make mrproper
# lubancat_linux_rk3588_defconfig 設定ファイルを読み込む、rk3588 はこの設定ファイルを使用
make lubancat_linux_rk3588_defconfig
# カーネルをコンパイル、クロスコンパイルツールを指定し、4 スレッドでコンパイル
make -j4
```

ボード上でのローカルコンパイルは時間がかかります。rk3588 では 1 時間以上、RK358x では半時間程度かかる場合があります。

カーネルが成功裏にコンパイルされた後、次の内容の学習を続けることができます。

1.4 カーネルドライバーモジュールのコンパイルとロード

1.4.1 カーネルドライバーモジュールのコンパイル

カーネルモジュールをカーネルにロードすることができ、カーネルモジュールを個別のモジュールとしてコンパイルし、カーネル起動後にユーザーが手で動的にロードすることも、モジュールを直接カーネルに組み込んでカーネル起動時に自動的にロードすることもできます。テストでは通常、個別のカーネルモジュールとしてコンパイルし、手でロードすることで、デバッグを容易にし、時間を節約します。

公式はドライバーチュートリアルのソースコードを提供しており、以下のコマンドで取得できます：

```
# github から取得
git clone https://github.com/LubanCat/lubancat_rk_code_storage
```

ソースコードを取得した後、ソースコードディレクトリ下の linux_driver フォルダーはドライバーチュートリアルの例示ファイルを格納しています。対応するドライバープログラムコードをカーネルコードと同じディレクトリレベルに配置する必要があります。これは、カーネルモジュールをコンパ

株式会社日昇テクノロジー

イルする際に、ドライバープログラムがコンパイルされた Linux カーネルに依存するためです。ドライバーモジュールの Makefile 内にカーネルのパスが指定されています。例示の利用を容易にするために、同一のディレクトリ構造に配置してください。

```
cat@lubancat:~/lubancat_rk_code_storage$ ls
base_linux    lubancat_ai_manual_code    python_lubancat_RK_tutorial_code  README.md
linux_driver  lubancat_qt_tutorial_code  quick_start                       test
cat@lubancat:~/lubancat_rk_code_storage$ mv linux_driver/ ..
cat@lubancat:~/lubancat_rk_code_storage$ ls
base_linux              lubancat_qt_tutorial_code         quick_start  test
lubancat_ai_manual_code  python_lubancat_RK_tutorial_code  README.md
cat@lubancat:~/lubancat_rk_code_storage$ cd ..
cat@lubancat:~$ ls
linux_driver  lubancat_rk_code_storage
cat@lubancat:~$
```

カーネルドライバーモジュールの構築とコンパイルは複雑であり、Linux カーネルのビルドシステムの強力な機能を利用しています。現時点では、この部分の知識を深く理解する必要はありません。簡単な Make ツールを使用して、必要なカーネルドライバーモジュールをコンパイルできます。ここでは、hellomodule カーネルモジュールのコンパイルを例にとり、hellomodule ディレクトリに移動して make コマンドを使用します：

```
cd linux_driver/module/hellomodule/

make
```

重要：Makefile 内で指定されているディレクトリ「KERNEL_DIR=../../../kernel/」は、実際にカーネルをコンパイルした際に指定した出力ディレクトリと一致させる必要があります。カーネルをコンパイルする際に特定の出力ディレクトリを指定していない場合は、この変数をカーネルソースのルートディレクトリに指定してください。これは PC 上で、クロスコンパイルツールを使用してカーネルモジュールをコンパイルする環境であり、ボード上でカーネルモジュールをコンパイルする場合も似ています。ボードシステムの gcc ツールを使用し、Makefile で CROSS_COMPILE や ARCH を指定しなくてもよいです。

module/hellomodule ディレクトリに切り替え、make コマンドを直接実行すると、プログラムがコンパイルされます。

株式会社日昇テクノロジー

リスト 1: Makefile(module/hellomodule/Makefile)

```
1 KERNEL_DIR=../../../kernel/
2 ARCH=arm64
3 CROSS_COMPILE=aarch64-linux-gnu-
4 export ARCH CROSS_COMPILE
5
6 obj-m := hellomodule.o
7
8 all:
9 $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) modules
10
11 .PHONE:clean
12
13 clean:
14 $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) clean
```

株式会社日昇テクノロジー

- 第 1 行：カーネルディレクトリを指定します。自身がカーネルをコンパイルした際に指定した出力ディレクトリに合わせて、相対パスまたは絶対パスを使用できます。カーネルをコンパイルする際に特定の出力ディレクトリを指定していない場合は、この変数をカーネルソースのルートディレクトリに指定します。

- 第 2 行：arm64 アーキテクチャを指定します。

- 第 3 行：クロスコンパイルツールチェーンを指定します。Lubancat-SDK のクロスコンパイルツール gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu を使用できます。

- 第 4 行：環境変数をエクスポートします。

- 第 6 行：モジュールとしてコンパイルします。

- 第 8 行：all は単なるラベルで、make のデフォルト実行ターゲットとして自由に定義できます。

- 第 9 行：$(MAKE)の MAKE は Makefile 内のマクロ変数で、マクロ変数を参照するには$を使用します。ここでは、make プログラムを指します。-C オプションはディレクトリ変更のためのもので、-C dir は dir ディレクトリに移動します。M=$(CURDIR)は現在のディレクトリを返します。これは、make がデフォルトのターゲット all を実行するときに、-C(KVDIR)でカーネルソースディレクトリに移動してそこの Makefile を実行し、-C $(KERNEL_DIR)でカーネルソースディレクトリに移動してそこの Makefile を実行し、M=$(CURDIR)で再び現在のディレクトリに戻って現在の Makefile を実行することを意味します。

- 第 11 行：clean は、make によって生成されたファイルを削除します。

```
csun@ubuntu:~/LubanCat_SDK/lubancat_rk_code_storage/linux_driver/module/hellomodule$ make
make -C ../../../../kernel/ M=/home/csun/LubanCat_SDK/lubancat_rk_code_storage/linux_driver/module/hellomodule modules
make[1]: Entering directory '/home/csun/LubanCat_SDK/kernel'
  CC [M]  /home/csun/LubanCat_SDK/lubancat_rk_code_storage/linux_driver/module/hellomodule/hellomodule.o
WARNING: Symbol version dump "Module.symvers" is missing.
         Modules may not have dependencies or modversions.
  MODPOST /home/csun/LubanCat_SDK/lubancat_rk_code_storage/linux_driver/module/hellomodule/Module.symvers
WARNING: modpost: Symbol info of vmlinux is missing. Unresolved symbol check will be entirely skipped.
  CC [M]  /home/csun/LubanCat_SDK/lubancat_rk_code_storage/linux_driver/module/hellomodule/hellomodule.mod.o
  LD [M]  /home/csun/LubanCat_SDK/lubancat_rk_code_storage/linux_driver/module/hellomodule/hellomodule.ko
make[1]: Leaving directory '/home/csun/LubanCat_SDK/kernel'
```

module/hellomodule/に、新しく hellomodule.ko ファイルが追加され、これがコンパイルされたカーネルドライバーモジュールです。

1.4.2 カーネルドライバーモジュールのロード

コンパイルされたカーネルドライバーモジュールは、NFS ネットワークファイルシステム、scp コマンド、sftp コマンドなど、さまざまな方法で開発ボードにコピーできます。NFS 環境の構築については、Linux の章「NFS ネットワークファイルシステムのマウント」を参照してください。

scp コマンドは Linux 間でファイルやディレクトリをコピーするために使用され、ssh に基づいています。ssh 環境を構築する必要があります。scp コマンドの形式は以下の通りです：

株式会社日昇テクノロジー

```
scp local_file remote_username@remote_ip:remote_folder
```

例えば：

```
scp hellomodule.ko cat@192.168.103.129:/home/cat/
```

これにより、hellomodule.ko が 192.168.103.129 の/home/cat/ディレクトリに送信されます。192.168.103.129 は開発ボードの IP で、実際には状況に応じて変わります。開発ボードのユーザー名は cat です。yes と入力し、パスワードを入力して認証し、転送が完了するのを待ちます。この時点で、開発ボードには hellomodule.ko ファイルがあります。ローカルでコンパイルした場合は、転送する必要はありません。

```
csun@ubuntu:~/LubanCat_SDK/lubancat_rk_code_storage/linux_driver/module/hellomodule$ scp hellomodule.ko cat@192.168.11.151:/home/cat/
The authenticity of host '192.168.11.151 (192.168.11.151)' can't be established.
ED25519 key fingerprint is SHA256:b9zz8dJ0F4qf3I9vAMcaILk2G9+VsgnVU0Na5rgIZMU.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '192.168.11.151' (ED25519) to the list of known hosts.
cat@192.168.11.151's password:
hellomodule.ko
```

カーネルドライバーモジュールのインストールとアンインストールには insmod と rmmod コマンドを使用します：

```
# ホームディレクトリに移動

cd /home/cat/

# カーネルモジュールをロード

sudo insmod hellomodule.ko
```

```
# 現在ロードされているカーネルモジュールを確認

lsmod

# カーネルモジュールのアンロード

sudo rmmod hellomodule.ko
```

株式会社日昇テクノロジー

```
root@lubancat:~# cd /home/cat/
root@lubancat:/home/cat# insmod hellomodule.ko
[  133.929919] hellomodule: loading out-of-tree module taints kernel.
[  133.930054] hellomodule: module license 'GPL2' taints kernel.
[  133.930075] Disabling lock debugging due to kernel taint
[  133.931581] [ KERN_EMERG ]  Hello  Module Init
[  133.931598] [ default ]  Hello  Module Init
root@lubancat:/home/cat#
Message from syslogd@lubancat at Aug 26 21:36:48 ...
 kernel:[  133.931581] [ KERN_EMERG ]  Hello  Module Init

root@lubancat:/home/cat# lsmod
Module                  Size  Used by
hellomodule            16384  0
```

ロードされたカーネルモジュールを確認するには、lsmod コマンドを使用します。その他の情報は、/sys/module ディレクトリ下で確認できます。例えば、hellomodule.ko モジュールが成功してロードされた後、/sys/module/hellomodule 下で確認できます。

1.5 デバイストリーのコンパイルとロード方法

1.5.1 デバイストリーのコンパイル

Linux 3.x 以降のバージョンでは、デバイストリー（Device Tree）の概念とメカニズムが導入されました。デバイストリーは、ハードウェアプラットフォームを記述するための静的なデータ構造であり、ハードウェアデバイス、バス、割り込みコントローラーなどの詳細な説明が含まれています。後に記述するドライバーはデバイストリーに依存するため、ここではデバイストリーのコンパイル方法とロード方法を紹介します。ここではコードの説明を行わず、具体的な原理については後続の Linux デバイストリーチャプターを参照してください。

1.5.1.1 カーネルツールを使用してデバイストリーをコンパイル

Linux カーネルをコンパイルすると、dtc（Device Tree Compiler）というツールが生成されます。このツールは、デバイストリーソースファイル（.dts または.dtsi ファイル）をバイナリのデバイストリーファイル（.dtb ファイル）に自動コンパイルするために使用されます。また、以下のコマンドで dtc コンパイルツールをダウンロードすることもできます。

株式会社日昇テクノロジー

```
# dts から dtb へコンパイル

カーネルディレクトリ/scripts/dtc/dtc -I dts -O dtb -o xxx.dtb xxx.dts

# dtc ツールをダウンロードしてコンパイル

sudo apt install device-tree-compiler

dtc -I dts -O dtb -o xxx.dtb xxx.dts
```

カーネルが使用する dtc ツールのコマンドは大まかに上記のようなものですが、実際にはデバイストリーには多くの依存関係があるため、通常は dtc コマンドだけでコンパイルすることはありません。以上は擬似コードであり、参考のためのもので、理解しておくことが重要です。

1.5.1.2 カーネルのビルドスクリプトを使用してデバイストリーをコンパイル

カーネルのビルドスクリプトを使用してデバイストリーをコンパイルすることもできます。使用するデバイストリーファイルは、カーネルソースコードの/arch/arm64/boot/dts/rockchip 内に格納されています。

カーネルをコンパイルする際に自動的にデバイストリーもコンパイルされますが、カーネルのコンパイルは時間がかかるため、以下のコマンドを使用してデバイストリーのみをコンパイルすることをお勧めします。

rk3588 ボードは、以下のコマンドを実行します：

```
# 設定ファイルをロード

make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- lubancat_linux_rk3588_defconfig

# dtbs パラメータを使用してデバイストリーのみをコンパイル

make ARCH=arm64 -j4 CROSS_COMPILE=aarch64-linux-gnu- dtbs
```

株式会社日昇テクノロジー

前述したように、カーネルをコンパイルする際にデバイストリーもコンパイルされますが、変更や追加がない場合、ビルドスクリプトを使用してデバイストリーを単独でコンパイルしても出力はありません。手でデバイストリーファイルを変更してからコンパイルテストを行うことができます。

例えば、カーネルソースコード/arch/arm64/boot/dts/rockchip 内の xxx.dts ファイルを変更してからコンパイルします。対応する dtb ファイルに関連する dts ソースファイルを見つけ、簡単に dts ファイル内にスペースを追加して、誤って変更したことでコンパイルエラーが発生するのを防ぎます。

ここでは、RK3588-lubancat4.dts の変更を例に挙げます。

```
csun@ubuntu:~/LubanCat_SDK/kernel$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- lubancat_linux_rk3588_defconfig
#
# configuration written to .config
#
```

```
csun@ubuntu:~/LubanCat_SDK/kernel$ sudo make ARCH=arm64 -j4 CROSS_COMPILE=aarch64-linux-gnu- dtbs
[sudo] password for csun:
  SYNC    include/config/auto.conf.cmd
csun@ubuntu:~/LubanCat_SDK/kernel$
```

コマンドを実行すると、変更されたデバイストリーがコンパイルされ、生成されたデバイストリーファイル(.dtb)はソースディレクトリ内のカーネルソースコード/arch/arm64/boot/dts/rockchip に位置します。

1.5.2 デバイストリーのロード

コンパイルされた新しいデバイストリーファイルを、対応するボードのデバイストリーに置き換え、/boot/dtb/ディレクトリ内のデバイストリーファイルを置き換えます。

1.5.2.1 ボードが使用するデバイストリーファイルの確認

ボードを起動してログインした後、/boot/ディレクトリ内のシンボリックリンクを確認することで、現在のボードが使用しているデバイストリーファイルを特定できます。

ボードで以下のコマンドを実行します：

```
ls -l /boot/
```

株式会社日昇テクノロジー

```
cat@lubancat:~$ ls -l /boot/
total 52376
-rw-rw-r-- 1 root root     1537 Mar 12 13:37 boot.cmd
-rw-r--r-- 1 root root        0 Jun 18  2023 boot_init
-rw-rw-r-- 1 root root     1609 Mar 12 13:37 boot.scr
-rw-rw-r-- 1 root root   204447 Mar 12 13:37 config-5.10.160
drwxrwxr-x 3 root root     4096 Mar 12 13:37 dtb
drwxrwxr-x 2 root root     4096 Mar 12 13:37 extlinux
-rw-rw-r-- 1 root root 38918656 Mar 12 13:37 Image-5.10.160
-rw-rw-r-- 1 root root  7069963 Mar 12 13:37 initrd-5.10.160
drwxrwxr-x 2 root root     4096 Jun 18  2023 kerneldeb
-rw-rw-r-- 1 root root  1215954 Jun 18  2023 logo.bmp
-rw-rw-r-- 1 root root  1215954 Mar 12 13:37 logo_kernel.bmp
drwx------ 2 root root    16384 Mar 12 13:37 lost+found
lrwxrwxrwx 1 root root       26 Jun 18  2023 rk-kernel.dtb -> dtb/rk3588s-lubancat-4.dtb
-rw-rw-r-- 1 root root  7586157 Mar 12 13:37 System.map-5.10.160
drwxrwxr-x 2 root root     4096 Jun 18  2023 uEnv
cat@lubancat:~$
```

1.5.2.2 デバイストリーの置換

デバイストリーファイルは RK3588-lubancat-1n.dtb です。カーネルソースコード /arch/arm64/boot/dts/rockchip/ディレクトリからコンパイルされたデバイストリーファイルを SCP または NFS を使用して開発ボードにコピーし、/boot/dtb/ディレクトリの RK3588-lubancat-1n.dtb を置換します。

システム上でデバイストリーロードの状況を確認するために、例えばデバイストリールートディレクトリに leds というデバイストリーノードがある場合、新しいデバイストリーが正しくロードされているかを以下の方法で確認できます。デバイストリーノードはファイルシステム内で"/proc/device-tree"ディレクトリに対応するファイルとして存在します。"/proc/device-tree"ディレクトリの内容を以下のように確認できます。

```
cat@lubancat:~$ ls /proc/device-tree
'#address-cells'            dmac@fe530000                i2s@fe420000                 npu@fde40000
'#size-cells'               dmac@fe550000                i2s@fe430000                 opp-table2
 __symbols__                dmc                          iep@fdef0000                 otp@fe38c000
 adc-keys                   dmc-fsp                      interrupt-controller@fd400000 pcie@fe260000
 aliases                    dmc-opp-table                interrupt-parent             pcie@fe270000
 arm-pmu                    dmcdbg                       iommu@fde4b000               pcie@fe280000
 audpwm@fe470000            dsi@fe060000                 iommu@fdea0800               pdm@fe440000
 backlight                  dsi@fe070000                 iommu@fded0480               phy@fe820000
 bus-npu                    dwmmc@fe000000               iommu@fdee0800               phy@fe830000
 bus-npu-opp-table          dwmmc@fe2b0000               iommu@fdef0800               phy@fe840000
 cam-avdd                   dwmmc@fe2c0000               iommu@fdf40f00               phy@fe8c0000
 cam-dovdd                  ebc@fdec0000                 iommu@fdf80800               pinctrl
 cam-dvdd                   edp@fe0c0000                 iommu@fdfe0800               power-management@fdd9
 can@fe570000               eink@fdf00000                iommu@fdff1a00               psci
 can@fe580000               ethernet@fe010000            iommu@fe043e00               pvtm@fde00000
 can@fe590000               ethernet@fe2a0000            jpegd@fded0000               pvtm@fde80000
 chosen                     external-camera-clock        leds                         pvtm@fde90000
 clock-controller@fdd00000  external-gmac0-clock        lpddr3-params                pwm@fdd70000
 clock-controller@fdd20000  external-gmac1-clock        lpddr4-params                pwm@fdd70010
 codec-digital@fe478000     fiq-debugger                 lpddr4x-params               pwm@fdd70020
```

株式会社日昇テクノロジー

led フォルダに進み、led ノードに定義された属性やその子ノードを確認できます。

```
cat@lubancat:/proc/device-tree/leds$ ls
compatible  name  phandle  status  sys-status-led
cat@lubancat:/proc/device-tree/leds$
```

"name"属性を含むノードのプロパティはファイルであり、子ノードはフォルダです。"sys-status-led"フォルダに進み、"compatible"、"name"、"status"などのプロパティファイルを"cat"コマンドで確認できます。

```
cat@lubancat:/proc/device-tree/leds/sys-status-led$ ls
default-state  gpios  label  linux,default-trigger  name  phandle  pinctrl-0  pinctrl-names
cat@lubancat:/proc/device-tree/leds/sys-status-led$ cat name
sys-status-ledcat@lubancat:/proc/device-tree/leds/sys-status-led$
```

以上で、デバイストリーの置換とロードが成功したことになります。

1.6 デバイストリープラグインのコンパイルとロード

1.6.1 デバイストリープラグインのコンパイル

Linux 4.4 以降、ダイナミックデバイストリー(Dynamic Device Tree)が導入されました。デバイストリープラグインは、システムに動的にロードされ、カーネルによって認識されることができます。

注意：デバイストリープラグインとデバイストリーは互いに代替する関係ではなく、補完的な関係にあります。デバイストリープラグインは、メインデバイストリーが定型化された後に、メインデバイストリーが記述していない機能に対して動的に拡張を行うことができます。例えば、A ボードのデバイストリーにはシリアルポート 1 の機能が有効になっていないが、B ボードではシリアルポート 1 の機能を有効にする必要がある場合、A ボードのデバイストリーをそのまま使用し、デバイストリープラグインを使ってシリアルポート 1 を拡張することで、B ボードの要求を満たすことができます。

1.6.1.1 カーネルツールを使用してデバイストリープラグインをコンパイル

デバイストリープラグインのコンパイルはデバイストリーのコンパイルと似ています。ここでは、カーネルの dtc ツールを使用してデバイストリープラグインをコンパイルする過程を紹介します。カーネルで xxx.dts を xxx.dtbo にコンパイルする過程の例を示します。参考のためのものであり、実

株式会社日昇テクノロジー

際には以下のように行います：

```
カーネルビルドディレクトリ/scripts/dtc/dtc -I dts -O dtb -o xxx.dtbo xxx.dts
```

例えば、カーネルソースの arch/arm64/boot/dts/rockchip/overlay/ディレクトリ下の RK358x-lubancat-uart3-m0-overlay.dts を RK358x-lubancat-uart3-m0-overlay.dtbo にコンパイルします。

```
scripts/dtc/dtc -I dts -O dtb -o arch/arm64/boot/dts/rockchip/overlay/RK358x-lubancat-uart3-m0-overlay.dtbo arch/arm64/boot/dts/rockchip/overlay/RK358x-lubancat-uart3-m0-overlay.dts
```

コンパイルコマンドを実行した後、カーネルソースの arch/arm64/boot/dts/rockchip/overlay/ディレクトリ内で対応する dtbo ファイルを見つけることができます。

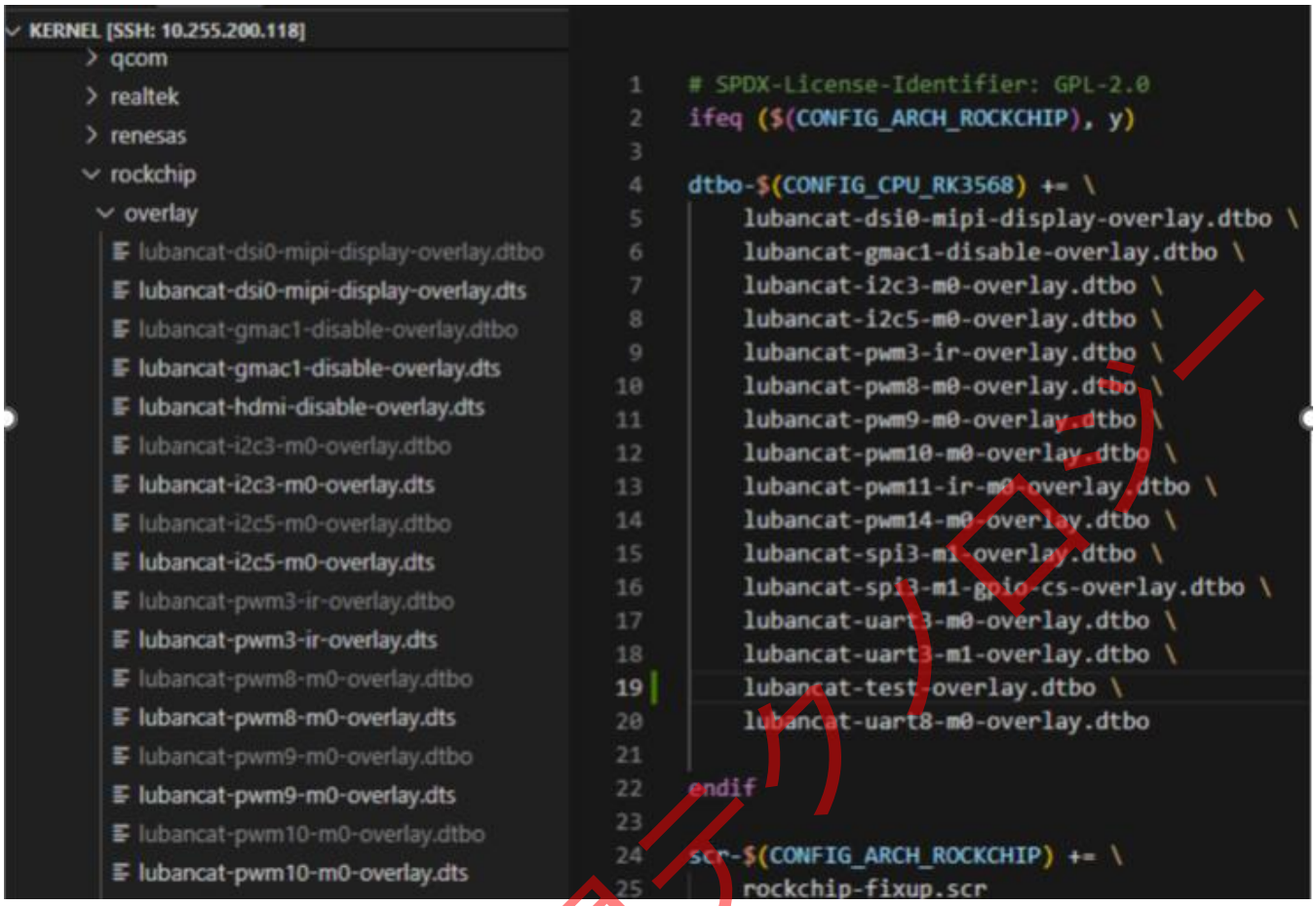
デバイストリープラグインのコンパイルもデバイストリーと同様に依存関係が関係しており、コンパイルプロセスも複雑です。一般的に、カーネルディレクトリ下で make コマンドを実行し、dtb オプションを指定することで、追加したデバイストリープラグインの自動コンパイルが可能です。

1.6.1.2 カーネルのビルドスクリプトを使用してデバイストリープラグインをコンパイル

デバイストリープラグインもデバイストリーと同様に DTC ツールを使用してコンパイルされますが、デバイストリーは.dtb に、デバイストリープラグインは.dtbo にコンパイルされます。DTC コンパイルコマンドを使用して.dtbo を生成することもできますが、これは煩雑でエラーが発生しやすいです。

ルバンキャットの開発ボードでは、多くの外部デバイスの記述が dtbo プラグインの形式で提供されており、デバイストリープラグインを使用してハードウェア外部デバイスを非常に柔軟に設定できます。

株式会社日昇テクノロジー



上記はデバイストリープラグインファイルのリストであり、カーネルをコンパイルする際にコンパイルするデバイストリープラグインファイルを指定しています。CONFIG_CPU_RK3588 が有効化された条件下で、ルバンキャットのボードでは、CONFIG_CPU_RK3588 に含まれるデバイストリープラグインがコンパイルされます。

rk3588 のボードのデバイストリープラグインは、CONFIG_CPU_RK3588 の条件文内に追加されます。

自分のデバイストリープラグインを試しに書いてみるときは、そのデバイストリープラグインソースファイルをカーネルソースの arch/arm64/boot/dts/rockchip/overlays ディレクトリに追加し、arch/arm64/boot/dts/rockchip/overlays/Makefile ファイルを変更してコンパイルオプションを追加します。例として、lubancat-test-overlay.dtbo を追加する場合の dts ソースコードは以下の通りです。

株式会社日昇テクノロジー

```
/dts-v1/;

/plugin/;

/ {

compatible = "rockchip,rk3588";

fragment@0 {

target = <&uart3>;

__overlay__ {

status = "okay";

};

};

};
```

上記は実際に rk3588-lubancat-uart3-m0-overlay.dts の内容であり、UART3 を"okay"状態に設定することを目的としています。これは、UART3 の周辺機器インターフェースを有効にすることを意味します。実際には、他のプラグインの内容をコピーしてテストすることも可能です。ボードの設定ファイルを確認して、衝突しないプラグインを選択することができます。ボードで cat /boot/uEnv/uEnv.txt を使用して確認することができます。

デバイストリープラグインソースファイルを overlays ディレクトリに追加した後、デバイストリーのコンパイルコマンドを実行すると、デバイストリープラグインのコンパイルも同時に完了します。

以下のコマンドを実行します：

```
# 設定ファイルをロード

make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- lubancat_linux_rk3588_defconfig

# デバイスツリーを個別にコンパイルするために dtbs パラメータを使用

make ARCH=arm64 -j4 CROSS_COMPILE=aarch64-linux-gnu- dtbs
```

株式会社日昇テクノロジー

新しいデバイスツリープラグインを修正または追加した場合、コマンドを実行した後、ターミナルに適切なコンパイル出力が表示されます。修正または追加された dts は dtbo ファイルにコンパイルされます。

1.6.2 デバイストリープラグインのロード

公式 Lubancat_RK ボードは、uboot を通じてデバイストリープラグインをロードし、カーネルによって認識されるようにすることをサポートしています。SCP または NFS を通じて.dtbo デバイストリープラグインを開発ボードの/boot/dtb/overlays/ディレクトリにコピーします。以下の操作はすべて開発ボード上で行います。

以下に示します：

```
cat@lubancat:/boot/dtb/overlay$ mv /home/cat/lubancat-test-overlay.dtbo ./
cat@lubancat:/boot/dtb/overlay$ ls lubancat-t*
lubancat-test-overlay.dtbo
cat@lubancat:/boot/dtb/overlay$
```

デバイスツリープラグインを該当デバイスにコピーします。

デバイストリーと同様に、デバイストリープラグインの設定ファイルも異なるボードの設定ファイルをロードするためにシンボリックリンクが使用されています。システムが実際にロードする設定ファイルを確認するには、以下のコマンドを実行します。

```
ls -l /boot/uEnv/
```

```
root@lubancat:/boot/uEnv# ls -l /boot/uEnv/
total 31
lrwxrwxrwx 1 root root   17 Aug 27 00:17 uEnv.txt -> uEnvLubanCat2.txt
-rw-r--r-- 1 root root  302 Aug 23 16:53 uEnvLubanCat-series.txt
-rw-r--r-- 1 root root 2561 Aug 23 16:53 uEnvLubanCat1.txt
-rw-r--r-- 1 root root 1380 Aug 23 16:53 uEnvLubanCat1H.txt
-rw-r--r-- 1 root root 1620 Aug 23 16:53 uEnvLubanCat1IO.txt
-rw-r--r-- 1 root root 2133 Aug 23 16:53 uEnvLubanCat1N.txt
-rw-r--r-- 1 root root 2656 Aug 23 16:53 uEnvLubanCat2-V1.txt
-rw-r--r-- 1 root root 3230 Aug 23 16:53 uEnvLubanCat2-V2.txt
-rw-r--r-- 1 root root 2656 Aug 23 16:53 uEnvLubanCat2.txt
-rw-r--r-- 1 root root 2153 Aug 23 16:53 uEnvLubanCat2IO.txt
-rw-r--r-- 1 root root 2595 Aug 23 16:53 uEnvLubanCat2N.txt
-rw-r--r-- 1 root root 1820 Aug 23 16:53 uEnvLubanCatZN.txt
-rw-r--r-- 1 root root 1896 Aug 23 16:53 uEnvLubanCatZW.txt
root@lubancat:/boot/uEnv#
```

上記の情報から、現在のシステムが実際にロードしている設定ファイルは /boot/uEnv/uEnvLubancat4.txt であることがわかります。

株式会社日昇テクノロジー

/boot/uEnv/uEnvLubancat4.txt 設定ファイルにロードしたいデバイストリープラグインを記述します。システムの起動中に uEnv.txt ファイルが自動的に読み込まれ、指定されたデバイストリープラグインがロードされます。"/boot/uEnv/"ディレクトリの uEnvLubancat4.txt ファイルを開き、以下に示すようにします：

```
uname_r=5.10.160
size=0x1000000
cmdline="earlyprintk console=ttyFIQ0 console=tty1 consoleblank=0 logleve
l=7 rootwait rw rootfstype=ext4"

enable_uboot_overlays=1
#overlay_start

#dtoverlay=/dtb/overlay/rk3588-lubancat-i2c2-m4-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-i2c3-m1-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-i2c5-m3-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-i2c6-m3-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-i2c8-m2-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-pwm3-ir-m3-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-pwm10-m2-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-pwm11-ir-m3-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-pwm14-m2-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-pwm15-ir-m3-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-spi0-m2-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-uart0-m2-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-uart4-m2-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-uart6-m1-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-uart7-m1-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-uart7-m2-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-uart9-m2-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-can0-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-can2-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam0-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam1-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588s-lubancat-4-cam2-overlay.dtbo
dtoverlay=/dtb/overlay/lubancat-test-overlay.dtbo
```

デバイストリープラグインを uEnvLubancat4.txt に記入するには、vim または nano エディタでファイルを開き、赤いボックスの内容を参考にして「dtoverlay=<デバイストリープラグインパス>」の形式で記述します。

変更を加えた後、保存して終了し、システムを再起動するために reboot コマンドを実行します。正常に再起動した後、"/proc/device-tree"または"/sys/firmware/devicetree/base/"ディレクトリ下にデバイスノードと同名のフォルダが見つかるはずです。これはデバイストリープラグインが正常にロー

株式会社日昇テクノロジー

ドされたことを証明しています。もし対応するノードが見つからない場合は、他のノードとの衝突が考えられるため、lubancat-test-overlay.dts の内容を設定ファイルの他のプラグインの内容に変更して再度テストしてください。

第 2 章 Linux カーネルモジュール

この章から、Linux デバイスドライバの世界に真剣に踏み込んでいきます。Linux システムでは、デバイスドライバがカーネルモジュールの形で存在します。Linux カーネルモジュールプログラミングを学ぶことは、ドライバ開発の前提条件です。Linux カーネルモジュールに初めて触れる際には、「Linux カーネルモジュールとは何か」、「Linux カーネルモジュールの動作原理」、「Linux カーネルモジュールの使用方法」について、この視点で Linux カーネルの世界に入っていきます。

Linux カーネルモジュールの紹介は大きく 4 つの部分に分けられます：

1. カーネルモジュールの概念：カーネルモジュールとは何ですか？カーネルモジュールメカニズムを導入した理由は何ですか？
2. カーネルモジュールの原理：カーネル内でのカーネルモジュールのロード、アンロードプロセス、カーネルモジュールがどのようにシンボルをエクスポートするかを深く分析します。
3. hellomodule 実験：カーネルモジュールのコードフレームワークと原理を理解し、自分のモジュールを書き、モジュールの使用方法などを学びます。
4. カーネルモジュールのパラメータ渡しとシンボル共有実験：カーネルモジュールのパラメータモード、シンボル共有を理解し、カーネルモジュールの動作メカニズムを検証します。

この章では最初の 2 部分を紹介し、残りの 2 部分は次章のカーネルモジュール実験で主に取り扱います。この章の内容は非常に関連性が高いため、順序よく注意深く読むことをお勧めします。

株式会社日昇テクノロジー

2.1 カーネルモジュールの概念

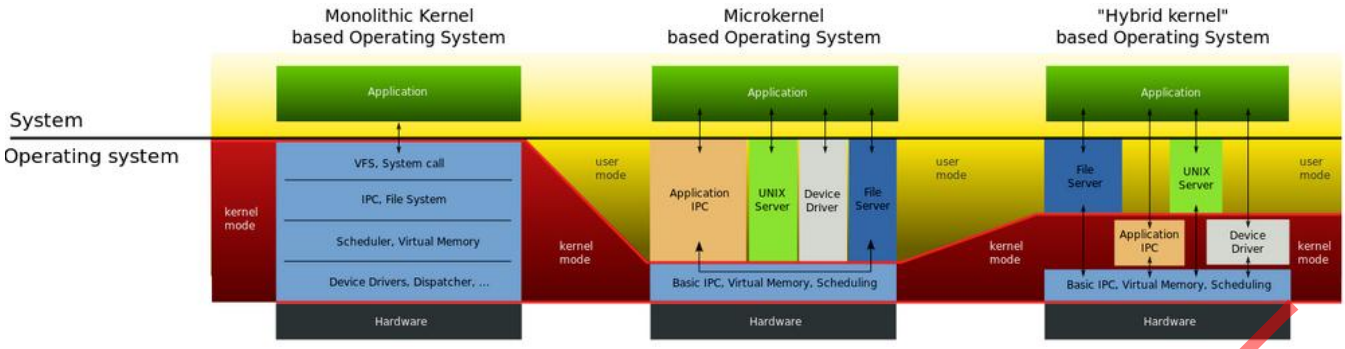
2.1.1 カーネル

カーネルは、オペレーティングシステムの核心部分です。ハードウェア上の最初のソフトウェア拡張であり、オペレーティングシステムの最も基本的な機能を提供し、オペレーティングシステムの動作の基盤となり、システムのパフォーマンスと安定性を決定します。

カーネルはアーキテクチャによって、マイクロカーネル（Micro Kernel）、モノリシックカーネル（Monolithic Kernel）、ハイブリッドカーネル（Hybrid Kernel）などに分かれます。マイクロカーネルアーキテクチャでは、カーネルはオペレーティングシステムの核心機能のみを提供し、プロセス管理、メモリ管理、プロセス間通信、I/O デバイス管理などを実装しますが、その他のアプリケーションレイヤの IPC、ファイルシステム機能、デバイスドライバモジュールはカーネル機能に含まれず、マイクロカーネル外のモジュールとなります。これらのモジュールの変更がマイクロカーネルの核心機能に影響を与えないため、動的な拡張性が強いという利点があります。Windows オペレーティングシステムや Huawei の HarmonyOS は、このタイプのマイクロカーネルアーキテクチャに属します。

一方、モノリシックカーネルアーキテクチャでは、上記のマイクロカーネルとその外部のアプリケーションレイヤの IPC、ファイルシステム機能、デバイスドライバモジュールをすべて一つにコンパイルします。その利点は実行効率が非常に高いことですが、カーネルの一部機能（例えばデバイスドライバプログラム）を変更または追加する場合には、カーネル全体を再コンパイルする必要があります。Linux オペレーティングシステムは、このモノリシックカーネル構造を採用しています。この欠点を解決するために、Linux にはカーネルモジュールメカニズムが導入されました。具体的な違いについては、以下の図を参照してください：

株式会社日昇テクノロジー



2.1.2 カーネルモジュールメカニズムの導入

2.1.2.1 カーネルモジュール導入の理由

Linux はクロスプラットフォームのオペレーティングシステムであり、多数のデバイスをサポートしています。Linux カーネルソースコードの 50%以上がデバイスドライバに関連しています。Linux はモノリシックカーネルアーキテクチャであり、すべての機能を有効にするとカーネルが非常に大きくなります。カーネルモジュールは、特定の機能を実装するカーネルコードの一部であり、カーネルの実行中にこのコード部分をカーネルにロードすることで、カーネルの機能を動的に拡張できます。この特性に基づいて、デバイスドライバを開発する際にカーネルモジュールの形式でコードを記述し、関連するドライバコードのみをコンパイルすることで、全体のカーネルを再コンパイルする必要がなくなります。

2.1.2.2 カーネルモジュールの導入

カーネルモジュールの導入は、システムの柔軟性を高めるだけでなく、開発者にとっても大きな利便性を提供します。デバイスドライバの開発プロセスでは、テスト中のドライバプログラムをカーネルに追加したり、カーネルから削除したりすることが自由にできます。カーネルモジュールのコードを変更するたびにカーネルを再起動する必要はありません。開発ボード上にカーネルモジュールプログラム、つまりデバイスドライバプログラムの ELF ファイルを保存する必要もなく、不必要なストレージスペースを占有することもありません。カーネルモジュールが必要なときには、NFS サーバーを

株式会社日昇テクノロジー

マウントして、他のデバイスに保存されているカーネルモジュールを開発ボードにロードできます。特定の状況でシステムのカーネルモジュールを必要に応じてロード/アンロードすることで、現在の環境により良いサービスを提供できます。

2.1.3 カーネルモジュールの定義と特徴

カーネルモジュールの導入とそれがもたらす多くの利点を理解した上で、カーネルモジュールについての初期の理解を頭の中で形成できます。次に、カーネルモジュールの具体的な定義を示しましょう：カーネルモジュール、全名 Loadable Kernel Module（LKM）、はカーネルが実行中に特定の機能を実現するために一連の目的コードをロードするメカニズムです。

モジュールは独立した機能を持つプログラムで、単独でコンパイル可能ですが、単独で実行することはできません。実行時にはカーネルの一部としてカーネル空間でリンクされて実行されます。これは、ユーザースペースで実行されるプロセスとは異なります。モジュールは、一連の関数とデータ構造からなり、ファイルシステム、ドライバプログラム、その他のカーネル上層機能を実装するために使用されます。したがって、カーネルモジュールは以下の特徴を持っています：

- モジュール自体はカーネルイメージにコンパイルされず、これによりカーネルのサイズを制御します。

- モジュールがロードされると、それはカーネルの他の部分と全く同じようになります。

カーネルモジュールの概念を理解したので、次にカーネルモジュールの動作メカニズムを詳しく見ていきましょう。

2.2 カーネルモジュールの動作メカニズム

作成したカーネルモジュールは、コンパイルされた後、.ko という拡張子を持つ ELF ファイルとして形成されます。file コマンドを使用してそれを確認できます。

```
csun@ubuntu:~/LubanCat_SDK/lubancat_rk_code_storage/linux_driver/module/hellomodule$ file hellomodule.o
hellomodule.o: ELF 64-bit LSB relocatable, ARM aarch64, version 1 (SYSV), with debug_info, not stripped
csun@ubuntu:~/LubanCat_SDK/lubancat_rk_code_storage/linux_driver/module/hellomodule$
```

株式会社日昇テクノロジー

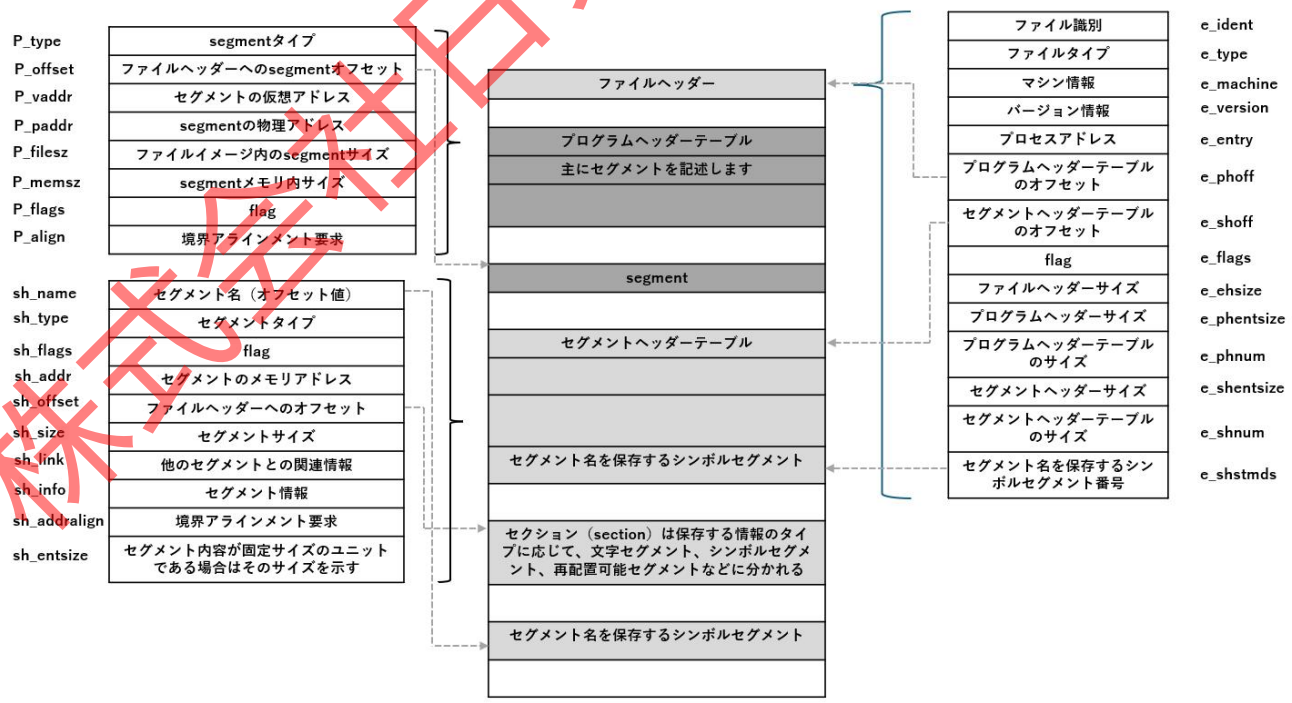
では、このようなファイルがどのようにしてカーネルによって一歩一歩取り込まれ、うまく機能するのかを見てみましょう。カーネルモジュールのロード/アンロードプロセスをよりよく理解するために、まず ELF ファイル形式について学び、.ko がどのようなものかを理解しましょう。次に、カーネルソースコードを一緒に見て、カーネルモジュールのロード/アンロードやシンボルのエクスポートがどのように行われるかを探求します。

2.2.1 カーネルモジュールの詳細なロード/アンロードプロセス

2.2.1.1 ko ファイルのファイル形式

ko ファイルはデータの組織形式が ELF（Executable And Linking Format）形式で、通常の再配置可能なオブジェクトファイルです。この種類のファイルにはコードとデータが含まれており、実行可能ファイルや共有オブジェクトファイルにリンクされたり、静的リンクライブラリに分類されたりすることができます。

ELF ファイル形式の可能なレイアウトは以下の通りです。



ファイルの先頭には ELF ヘッダ（ELF Header）があり、ファイル全体の構成を記述しています。こ

株式会社日昇テクノロジー

れらの情報はプロセッサに依存せず、ファイルの残りの内容からも独立しています。

readelf ツールを使用して、ELF ファイルのヘッダ情報を詳しく見ることができます。

```
csun@ubuntu:~/LubanCat_SDK/lubancat_rk_code_storage/linux_driver/module/hellomodule$ readelf -h hellomodule.ko
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           AArch64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          137280 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         37
  Section header string table index: 36
csun@ubuntu:~/LubanCat_SDK/lubancat_rk_code_storage/linux_driver/module/hellomodule$
```

プログラムヘッダーテーブル（Program Header Table）は、配列構造をしており、その各要素は以下を示します：

- ある「セグメント」：一つ以上の「セクション」を含み、プログラムヘッダーは実行可能ファイルや共有オブジェクトファイルにのみ意味があります。

- その他の情報：プログラム実行の準備に必要なその他の情報です。

セクションヘッダーテーブル／セグメント表（Section Header Table）は、ELF ファイル内に多種多様なセグメントが存在し、このセグメント表はこれらのセグメントの基本属性を保存する構造です。ELF ファイルのセグメント構造はセグメント表によって決定され、コンパイラ、リンカ、ローダーはこのセグメント表を利用して各セグメントの属性を位置づけてアクセスします。これにはファイルセクションの情報が含まれます。

ELF ヘッダーには：

- e_shoff：ファイルヘッダーからセクションヘッダーテーブルまでのオフセット（バイト数）を示します。

- e_shnum：テーブル内のエントリ数を示します。

株式会社日昇テクノロジー

- e_shentsize：各エントリのサイズ（バイト数）を示します。

これらの情報からセクションの具体的な位置、長さを正確に特定でき、プログラムヘッダーテーブルと同様に、セクションヘッダーテーブル内の各エントリには対応するセクションが存在します。これにより、セクションヘッダーテーブルが連続した空間であり、各エントリが構造体であることがわかります。

readelf ツールに-S オプションを付けることで、ELF ファイルのセクションヘッダーテーブルの詳細情報を読み取ることができます。

```
csun@ubuntu:~/LubanCat_SDK/lubancat_rk_code_storage/linux_driver/module/hellomodule$ readelf -s hellomodule.ko

Symbol table '.symtab' contains 47 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .note.gnu.build-id
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    2 .note.Linux
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3 .text
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4 .init.text
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    6 .exit.text
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    8 .plt
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    9 .init.plt
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT   10 .text.ftrace_tra[...]
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT   11 .rodata.str1.8
    10: 0000000000000000     0 SECTION LOCAL  DEFAULT   12 .modinfo
    11: 0000000000000000     0 SECTION LOCAL  DEFAULT   13 __mcount_loc
    12: 0000000000000000     0 SECTION LOCAL  DEFAULT   15 .data
    13: 0000000000000000     0 SECTION LOCAL  DEFAULT   16 .gnu.linkonce.th[...]
    14: 0000000000000000     0 SECTION LOCAL  DEFAULT   18 .bss
    15: 0000000000000000     0 SECTION LOCAL  DEFAULT   19 .note.GNU-stack
    16: 0000000000000000     0 SECTION LOCAL  DEFAULT   20 .comment
    17: 0000000000000000     0 SECTION LOCAL  DEFAULT   21 .debug_info
    18: 0000000000000000     0 SECTION LOCAL  DEFAULT   23 .debug_abbrev
    19: 0000000000000000     0 SECTION LOCAL  DEFAULT   24 .debug_aranges
    20: 0000000000000000     0 SECTION LOCAL  DEFAULT   26 .debug_rnglists
    21: 0000000000000000     0 SECTION LOCAL  DEFAULT   28 .debug_line
    22: 0000000000000000     0 SECTION LOCAL  DEFAULT   30 .debug_str
    23: 0000000000000000     0 SECTION LOCAL  DEFAULT   31 .debug_line_str
    24: 0000000000000000     0 SECTION LOCAL  DEFAULT   32 .debug_frame
    25: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hellomodule.mod.c
```

株式会社日昇テクノロジー

```
26: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT   16 $d
27: 0000000000000050     9 OBJECT  LOCAL  DEFAULT   12 __UNIQUE_ID_depe[...]
28: 0000000000000059    17 OBJECT  LOCAL  DEFAULT   12 __UNIQUE_ID_name264
29: 000000000000006a    41 OBJECT  LOCAL  DEFAULT   12 __UNIQUE_ID_verm[...]
30: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT    2 $d
31: 0000000000000000    24 OBJECT  LOCAL  DEFAULT    2 _note_7
32: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hellomodule.c
33: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT   11 $d
34: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT    4 $x
35: 0000000000000000    56 FUNC    LOCAL  DEFAULT    4 hello_init
36: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT    6 $x
37: 0000000000000000    28 FUNC    LOCAL  DEFAULT    6 hello_exit
38: 0000000000000000    18 OBJECT  LOCAL  DEFAULT   12 __UNIQUE_ID_alias266
39: 0000000000000012    31 OBJECT  LOCAL  DEFAULT   12 __UNIQUE_ID_desc[...]
40: 0000000000000031    18 OBJECT  LOCAL  DEFAULT   12 __UNIQUE_ID_author264
41: 0000000000000043    13 OBJECT  LOCAL  DEFAULT   12 __UNIQUE_ID_lice[...]
42: 0000000000000000   896 OBJECT  GLOBAL DEFAULT   16 __this_module
43: 0000000000000000    28 FUNC    GLOBAL DEFAULT    6 cleanup_module
44: 0000000000000000    56 FUNC    GLOBAL DEFAULT    4 init_module
45: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printk
46: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _mcount
```

セクションヘッダーテーブルには多くのサブテーブル情報が含まれていますが、ここでは二つを簡単に見てみましょう。

- リロケーションテーブル

リロケーションテーブル（".rel.text"）はセグメント表の後に位置し、そのタイプ（sh_type）は"SHT_REL"、つまりリロケーションテーブルです。リンカーはターゲットファイルを処理する際に、ターゲットファイル内の特定の位置をリロケート（再配置）する必要があり、これらのリロケーション情報は ELF ファイルのリロケーションテーブルに記録されています。リロケーションが必要な各コードセグメントやデータセグメントには、対応するリロケーションテーブルが存在します。リロケーションテーブルも ELF のセグメントの一つで、そのタイプは"SHT_REL"です。

リロケーションテーブルを読み取る。

株式会社日昇テクノロジー

```
csun@ubuntu:~/LubanCat_SDK/lubancat_rk_code_storage/linux_driver/module/hellomodule$ readelf -r hellomodule.ko

Relocation section '.rela.init.text' at offset 0x12298 contains 7 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000010  002e0000011b R_AARCH64_CALL26  0000000000000000 _mcount + 0
000000000014  000900000113 R_AARCH64_ADR_PRE 0000000000000000 .rodata.str1.8 + 0
000000000018  000900000115 R_AARCH64_ADD_ABS 0000000000000000 .rodata.str1.8 + 0
00000000001c  002d0000011b R_AARCH64_CALL26  0000000000000000 printk + 0
000000000020  000900000113 R_AARCH64_ADR_PRE 0000000000000000 .rodata.str1.8 + 28
000000000024  000900000115 R_AARCH64_ADD_ABS 0000000000000000 .rodata.str1.8 + 28
000000000028  002d0000011b R_AARCH64_CALL26  0000000000000000 printk + 0

Relocation section '.rela.exit.text' at offset 0x12340 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000004  000900000113 R_AARCH64_ADR_PRE 0000000000000000 .rodata.str1.8 + 50
000000000008  000900000115 R_AARCH64_ADD_ABS 0000000000000000 .rodata.str1.8 + 50
000000000010  002d0000011b R_AARCH64_CALL26  0000000000000000 printk + 0

Relocation section '.rela__mcount_loc' at offset 0x12388 contains 1 entry:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000000  000400000101 R_AARCH64_ABS64   0000000000000000 .init.text + 10

Relocation section '.rela.gnu.linkonce.this_module' at offset 0x123a0 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000158  002c00000101 R_AARCH64_ABS64   0000000000000000 init_module + 0
000000000360  002b00000101 R_AARCH64_ABS64   0000000000000000 cleanup_module + 0

Relocation section '.rela.debug_info' at offset 0x123d0 contains 2438 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000008  001200000102 R_AARCH64_ABS32   0000000000000000 .debug_abbrev + 0
00000000000d  001600000102 R_AARCH64_ABS32   0000000000000000 .debug_str + 2f2
000000000012  001700000102 R_AARCH64_ABS32   0000000000000000 .debug_line_str + 0
000000000016  001700000102 R_AARCH64_ABS32   0000000000000000 .debug_line_str + 5f
00000000001a  001400000102 R_AARCH64_ABS32   0000000000000000 .debug_rnglists + c
000000000026  001500000102 R_AARCH64_ABS32   0000000000000000 .debug_line + 0
00000000002d  001600000102 R_AARCH64_ABS32   0000000000000000 .debug_str + 570
00000000003a  001600000102 R_AARCH64_ABS32   0000000000000000 .debug_str + a58
000000000046  001600000102 R_AARCH64_ABS32   0000000000000000 .debug_str + 516
00000000004d  001600000102 R_AARCH64_ABS32   0000000000000000 .debug_str + 149
000000000054  001600000102 R_AARCH64_ABS32   0000000000000000 .debug_str + 89b
00000000005b  001600000102 R_AARCH64_ABS32   0000000000000000 .debug_str + 902
000000000062  001600000102 R_AARCH64_ABS32   0000000000000000 .debug_str + 5f8
000000000070  001600000102 R_AARCH64_ABS32   0000000000000000 .debug_str + 13b
000000000077  001600000102 R_AARCH64_ABS32   0000000000000000 .debug_str + bd
00000000007e  001600000102 R_AARCH64_ABS32   0000000000000000 .debug_str + 1f8
000000000085  001600000102 R_AARCH64_ABS32   0000000000000000 .debug_str + 88c
00000000008a  001600000102 R_AARCH64_ABS32   0000000000000000 .debug_str + 8ec
00000000009a  001600000102 R_AARCH64_ABS32   0000000000000000 .debug_str + d6e
```

- 文字列テーブル

ELF ファイルでは多くの文字列が使用されます。例えば、セグメント名、変数名などです。文字列の長さは通常不定であるため、固定の構造で表現するのが難しいです。一般的な方法は、文字列を一つのテーブルに集めて、表中のオフセットを使用して文字列を参照することです。一般的に文字列テーブルは ELF ファイル内でもセグメントの形で保存され、一般的なセグメント名は".strtab"（文字列テーブル）または".shstrtab"（セクションヘッダー文字列テーブル）です。

セクション文字列テーブルを読み取る。

株式会社日昇テクノロジー

```
csun@ubuntu:~/LubanCat_SDK/lubancat_rk_code_storage/linux_driver/module/hellomodule$ readelf -p 28 hellomodule.ko

String dump of section '.debug_line':
  Note: This section has relocations against it, but these have NOT been applied to this dump.
  [     a0]  .^E^E^F==^E^B=^E^A^F^S^B^C
  [     cb]  !^E^A-^E^E!^E^A!^B^B
  [     da]  r^B
  [     e2]  j^B
  [    129]  n
```

ELF ファイル形式に関する知識はやや複雑ですが、大まかな理解があれば十分です。主にカーネルモジュールのロードとアンロード、シンボルのエクスポートを理解するために、後に関連する用語について触れた時に馴染みがないわけではないようにするためです。

2.2.1.2 カーネルモジュールのロードプロセス

ko カーネルモジュールファイルのフォーマットについての理解を深めた後、カーネルモジュールは特別に加工されたコードであることがわかります。加工されたコードであるため、カーネルはカーネルモジュール内に残された情報を利用して、カーネルモジュールを活用できます。

次に、カーネルモジュールのロードプロセスについて見ていきましょう。

まず、insmod はファイルシステムを通じて.ko モジュールをユーザースペースのメモリ領域に読み込み、その後、sys_init_module()システムコールを実行してモジュールを解析します。この時、カーネルは vmalloc 領域に ko ファイルと同じサイズのメモリを一時的に割り当てて ko ファイルを保持します。保持した後、ko ファイルを解析し、ファイル内の各セクションを init セグメントと core セグメントに割り当て、modules 領域に init セグメントと core セグメントのためのメモリを割り当て、対応するセクションを modules 領域の最終実行アドレスにコピーします。relocate 関数アドレスなどの操作を経て、ko の init 操作を実行できます。これで ko のロードプロセスが完了します。同時に、init セグメントは解放され、core セグメントのみが実行のために残されます。

リスト 1: sys_init_module() (カーネルソースコード/kernel/module.c)

```
1 SYSCALL_DEFINE3(init_module, void __user *, umod,
2 unsigned long, len, const char __user *, uargs)
```

株式会社日昇テクノロジー

```
3 {
4 int err;
5 struct load_info info = { };
6
7 err = may_init_module();
8 if (err)
9 return err;
10
11 pr_debug("init_module: umod=%p, len=%lu, uargs=%p¥n",
12 umod, len, uargs);
13
14 err = copy_module_from_user(umod, len, &info);
15 if (err)
16 return err;
17
18 return load_module(&info, uargs, 0);
19 }
```

- 第 14 行: vmalloc を使用して vmalloc 領域にメモリ空間を割り当て、カーネルモジュールをこの空間にコピーし、info->hdr はこの空間の先頭アドレス、つまり ko の elf ヘッダに直接指します。

- 第 18 行: 次に load_module() を通じてモジュールのロードのコア処理を行います。ここではモジュールの移動、リダイレクトなどの困難なプロセスが完了します。

以下は、load_module()の詳細なプロセスで、コードは私によって簡略化されました。主に setup_load_info()と layout_and_allocate()を含んでいます。

株式会社日昇テクノロジー

リスト 2: load_module()(カーネルソースコード/kernel/module.c)

```
1 /* モジュールを割り当ててロードする*/
2 static int load_module(struct load_info *info, const char __user *uargs,
3 int flags)
4 {
5 struct module *mod;
6 long err = 0;
7 char *after_dashes;
8 ...
9 err = setup_load_info(info, flags);
10 ...
11 mod = layout_and_allocate(info, flags);
12 ...
13 }
```

- 第 9 行:setup_load_info() では struct load_info と struct module、rewrite_section_headers をロードし、各セクションの sh_addr を現在のイメージがあるメモリアドレスに変更します。セクション名文字列テーブルのアドレスは ELF ヘッダの e_shstrndx からセクションヘッダ文字列テーブルのインデックスを取得し、対応するセクションの ELF ファイル内のオフセットを見つけ、ELF ファイルの開始アドレスに加えることでメモリ内のアドレスを得ます。

- 第 11 行: layout_and_allocate() では、layout_sections() がセクションを core と init の二つの大きなカテゴリーに分類し、ko の第二の移動を準備します。move_module() は ko を最終的な実行アドレスに移動します。これにより、カーネルモジュールのロードコードの移動プロセスが完了します。

株式会社日昇テクノロジー

しかし、この時点でカーネルモジュールが動作するためには、シンボルのエクスポートが必要です。カーネルモジュールのシンボルエクスポートについては、後のセクションで詳しく説明します。

2.2.1.3 カーネルモジュールのアンロードプロセス

アンロードプロセスはロードよりも比較的シンプルで、rmmod コマンドを入力すると、システムカーネル内で sys_delete_module を呼び出して実行されます。

具体的なプロセスは、最初にユーザースペースからアンロードするモジュール名を渡し、その名前に基づいてアンロードするモジュールのポインタを見つけ、アンロードしたいモジュールが他のモジュールに依存していないことを確認し、モジュール自身の exit 関数を見つけてアンロードを実行します。

リスト 3：カーネルモジュールのアンロード（カーネルソースコード/kernel/module.c）

```
1 SYSCALL_DEFINE2(delete_module, const char __user *, name_user,
2 unsigned int, flags)
3 {
4 struct module *mod;
5 char name[MODULE_NAME_LEN];
6 int ret, forced = 0;
7
8 if (!capable(CAP_SYS_MODULE) || modules_disabled)
9 return -EPERM;
10
11 if (strncpy_from_user(name, name_user, MODULE_NAME_LEN-1) < 0)
12 return -EFAULT;
```

株式会社日昇テクノロジー

```
13 name[MODULE_NAME_LEN-1] = '¥0';

14

15 audit_log_kern_module(name);

16

17 if (mutex_lock_interruptible(&module_mutex) != 0)

18 return -EINTR;

19

20 mod = find_module(name);

21 if (!mod) {

22 ret = -ENOENT;

23 goto out;

24 }

25

26 if (!list_empty(&mod->source_list)) {

27 ret = -EWOULDBLOCK;

28 goto out;

29 }

30

31 /* Doing init or already dying? */

32 if (mod->state != MODULE_STATE_LIVE) {

33 /* FIXME: if (force), slam module count damn the torpedoes */

34 pr_debug("%s already dying¥n", mod->name);
```

株式会社日昇テクノロジー

```
35 ret = -EBUSY;
36 goto out;
37 }
38
39 if (mod->init && !mod->exit) {
40 forced = try_force_unload(flags);
41 if (!forced) {
42 /* This module can't be removed */
43 ret = -EBUSY;
44 goto out;
45 }
46 }
47
48 ret = try_stop_module(mod, flags, &forced);
49 if (ret != 0)
50 goto out;
51
52 mutex_unlock(&module_mutex);
53 /* Final destruction now no one is using it. */
54 if (mod->exit != NULL)
55 mod->exit();
56 blocking_notifier_call_chain(&module_notify_list,MODULE_STATE_GOING,    mod);
```

株式会社日昇テクノロジー

```
57 klp_module_going(mod);
58 ftrace_release_mod(mod);
59
60 async_synchronize_full();
61
62 /* Store the name of the last unloaded module for diagnostic purposes */
63 strlcpy(last_unloaded_module, mod->name, sizeof(last_unloaded_module));
64
65 free_module(mod);
66 return 0;
67 out:
68 mutex_unlock(&module_mutex);
69 return ret;
70 }
```

- 第 8 行：挿入および削除モジュールの権利が制限されず、モジュールが挿入または削除されることが禁止されていないことを確認
- 第 11 行：モジュール名を取得
- 第 20 行：アンロードするモジュールのポインタを見つける
- 第 26 行：依存するモジュールがある場合、それらを先にアンロードする必要がある
- 第 39 行：モジュールの終了関数をチェック
- 第 48 行：マシンを停止し、参照カウントが動かないようにし、モジュールを無効にする
- 第 56 行：モジュールの状態が MODULE_STATE_GOING に変更されたことを、通知チェーン module_notify_list 上のリスナーに通知

株式会社日昇テクノロジー

- 第 60 行：すべての非同期関数コールが完了するのを待つ

2.2.2 カーネルがシンボルをどのようにエクスポートするか

シンボルとは何か？なぜシンボルをエクスポートする必要があるのか？カーネルモジュールはどのようにシンボルをエクスポートするのか？他のモジュールはこれらのシンボルをどのように見つけるのか？

実際には、シンボルはカーネルモジュール内で EXPORT_SYMBOL 宣言された関数や変数を指します。モジュールがカーネルにロードされた後、エクスポートされたシンボルはすべて公共のカーネルシンボルテーブルに記録されます。insmod コマンドを使用してモジュールをロードした後、モジュールはカーネルにリンクされ、カーネルの共用シンボルにアクセスできるようになります。

通常、他のモジュールから便利な機能を提供したい場合を除き、シンボルをエクスポートする必要はありません。これはモジュールカスケーディング技術と呼ばれます。たとえば、msdos ファイルシステムは fat モジュールからエクスポートされたシンボルに依存しています。USB 入力デバイスモジュールは、usbcore および input モジュールの上にカスケードされています。つまり、モジュールを複数の層に分け、各層を簡素化することで、複雑なプロジェクトを実現することができます。

modprobe はカスケードモジュールを扱うツールであり、指定されたモジュールをロードするだけでなく、指定されたモジュールが依存する他のモジュールも同時にロードする点で、insmod を複数回使用するのと同等の機能を持っています。

モジュールをエクスポートする場合は、以下のマクロを使用します。

```
EXPORT_SYMBOL(name)

EXPORT_SYMBOL_GPL(name) //name はエクスポートしたいシンボルです
```

シンボルはモジュールファイルのグローバルセクションでエクスポートする必要があり、関数内で使用することはできません。_GPL は、エクスポートされたモジュールが GPL ライセンスのモジュー

株式会社日昇テクノロジー

ルにのみ使用されるようにします。モジュールをコンパイルするとき、これらのマクロは特別な変数の宣言に展開され、ELF ファイルのシンボルテーブルに保存されます。具体的には、ELF ファイルのシンボルテーブルには以下の情報が含まれます。

- st_name：シンボル名がシンボル名文字列テーブル内のインデックス値です。
- st_value：シンボルが存在するメモリアドレスです。
- st_size：シンボルのサイズです。
- st_info：シンボルのタイプとバインド情報です。
- st_shndx：シンボルが存在するセクションを表します。

ELF のシンボルテーブルがカーネルにロードされた後、simplify_symbols 関数が ELF ファイルのシンボルテーブル全体を走査します。st_shndx に基づいてシンボルが存在するセクションと、st_value でのセクション内のオフセットから、実際のメモリアドレスを得ます。そして、最終的にシンボルのメモリアドレスとシンボル名のポインターをカーネルのシンボルテーブルに保存します。

simplify_symbols 関数のプロトタイプは以下のとおりです。

リスト４： static int simplify_symbols(struct module *mod, const struct load_info *info)

```
1 static int simplify_symbols(struct module *mod, const struct load_info    *info)
```

パラメータと戻り値は以下のとおりです。

パラメータ

- mod：struct module 型の構造体ポインター
- info：const struct load_info 型の構造体ポインター

戻り値

- ret：エラーコード

カーネルがエクスポートするシンボルテーブルの構造には 2 つのフィールドがあります。一つはメモリ内のシンボルのアドレス、もう一つはシンボル名のポインターです。シンボル名は

株式会社日昇テクノロジー

__ksymtab_strings セクションに格納されています。EXPORT_SYMBOL を例にすると、シンボルは ___ksymtab というセクションに格納されます。この構造体は、エクスポートされたシンボルテーブルを構成するものであり、通常の意味でのシンボルテーブルではありません。

リスト 5: kernel_symbol 構造体（カーネルソースコード/include/linux/export.h）

```
1 struct kernel_symbol {
2 unsigned long value;
3 const char *name;
4 };
```

value：メモリ内のシンボルのアドレス

name：シンボル名

他のカーネルモジュールがシンボルを検索する際には、resolve_symbol_wait を呼び出してカーネルおよび他のモジュール内のシンボル名によって目的のシンボルを寻址します。resolve_symbol_wait は resolve_symbol を呼び出し、それが find_symbol を呼び出します。シンボルが見つかった後、その実際のアドレスをシンボルテーブル sym[i].st_value = ksym->value に割り当てます。

リスト 6: find_symbol 関数（カーネルソースコード/kernel/module.c）

```
1 /* シンボルを見つけ、それに関連する（オプションの）crc と（オプションの）モジュールを返します。プリエンプションまたはモジュールの相互排他を無効にする必要があります。*/
2 const struct kernel_symbol *find_symbol(const char *name,
3 struct module **owner,
4 const s32 **crc,
5 bool gplok,
6 bool warn)
```

株式会社日昇テクノロジー

```
7 {
8 struct find_symbol_arg fsa;
9
10 fsa.name = name;
11 fsa.gplok = gplok;
12 fsa.warn = warn;
13
14 if (each_symbol_section(find_symbol_in_section, &fsa)) {
15 if (owner)
16 *owner = fsa.owner;
17 if (crc)
18 *crc = fsa.crc;
19 return fsa.sym;
20 }
21
22 pr_debug("Failed to find symbol %s¥n", name);
23 return NULL;
24 }
25 EXPORT_SYMBOL_GPL(find_symbol);
```

株式会社日昇テクノロジー

第 14 行：each_symbol_section 内で、2 箇所を検索しました。一つはカーネルのエクスポートシンボルテーブル、つまりカーネルシンボルがどのようにエクスポートされるかを定義したグローバル変数、もう一つは既にロードされたカーネルモジュールを走査します。検索アクションは each_symbol_in_section 内で完了します。

第 25 行：エクスポートシンボルフラグ

これによりシンボルの検索が完了し、最後にすべてのセクションを ELF ファイルのリダイレクションテーブルを使ってリダイレクトすることで、そのシンボルを使用できるようになります。

これでカーネルはカーネルモジュールのロード/アンロードおよびシンボルのエクスポートを完了しました。興味のある読者はカーネルソースコードディレクトリの/kernel/module.c を参照してください。

第 3 章 Linux カーネルモジュール実験

3.1 hellomodule 実験

3.1.1 実験説明

3.1.1.1 ハードウェア紹介

本節の実験では、Lubancat_RK のボードを使用します。

3.1.2 実験コード解説

本章のサンプルコードディレクトリは以下の通りです：linux_driver/module/hellomodule

これまでにカーネルモジュールの動作原理を理解したので、このセクションではコーディングを開始します。以下に、最もシンプルな hellomodule フレームワークを示します。

リスト 1: hellomodule フレームワーク

```
1 #include <linux/module.h>
2 #include <linux/init.h>
3 #include <linux/kernel.h>
4
5 static int __init hello_init(void)
```

株式会社日昇テクノロジー

```
6 {
7 printk(KERN_EMERG "[ KERN_EMERG ] Hello Module Init¥n");
8 printk( "[ default ] Hello Module Init¥n");
9 return 0;
10 }
11
12 static void __exit hello_exit(void)
13 {
14 printk("[ default ] Hello Module Exit¥n");
15 }
16
17 module_init(hello_init);
18 module_exit(hello_exit);
19
20 MODULE_LICENSE("GPL2");
21 MODULE_AUTHOR("embedfire ");
22 MODULE_DESCRIPTION("hello module");
23 MODULE_ALIAS("test_module");
```

以下では、コードの各行の意味を理解し、Linux 上でこのモジュールを実行して、これまでの理論を検証し、次章のドライバ開発の基礎を築きます。

3.1.2.1 コードフレームワーク分析

Linux カーネルモジュールのコードフレームワークは通常、以下の部分から構成されます：

株式会社日昇テクノロジー

- モジュールロード関数（必須）：insmod または modprobe コマンドでカーネルモジュールをロードすると、モジュールのロード関数が自動的にカーネルによって実行され、このモジュールに関連する初期化作業を完了します。

- モジュールアンロード関数（必須）：rmmod コマンドでモジュールをアンロードすると、モジュールアンロード関数が自動的にカーネルによって実行され、関連するクリーンアップ作業を行います。

- モジュールライセンス宣言（必須）：ライセンス宣言はカーネルモジュールの使用許可を記述し、モジュールが宣言されていない場合、カーネルが汚染された警告が表示されます。

- モジュールパラメータ：モジュールがロードされるとき、モジュール内のパラメータに値を渡すことができます。

- モジュールエクスポートシンボル：モジュールは、他のカーネルモジュールが呼び出すことができる変数や関数をシンボルとしてエクスポートすることができます。

- モジュールのその他の関連情報：モジュールの作者などの情報を宣言できます。

上記の例の hello module プログラムは、上記の三つの必須部分およびモジュールのその他の情報宣言を含んでいます（モジュールパラメータとエクスポートシンボルは次章の実験で登場します）。

ヘッダファイルには<linux/init.h>と<linux/module.h>が含まれており、これら二つのヘッダファイルはカーネルモジュールを記述する際に必須です。モジュール初期化関数 hello_init は、printk 関数を呼び出しており、カーネルモジュールが実行される過程で、C ライブラリの関数に依存できないため、printf 関数は使用できません。代わりに、printf 関数と類似した使用方法の printk 関数を使用する必要があります。モジュール初期化関数を完成させた後、hello_init 関数を初期化に使用することをカーネルに伝えるために、マクロ module_init を呼び出す必要があります。モジュールアンロード関数も printk 関数を使用して文字列をプリントし、マクロ module_exit を使用してカーネルにこのモジュールのアンロード関数を登録します。最後に、このモジュールが GPL2 プロトコルに従うことを

株式会社日昇テクノロジー

宣言する必要があります。

3.1.2.2 ヘッダファイル

前述のように、Linux のアプリケーションプログラミングに触れ、Linux のヘッダファイルが /usr/include に保存されていることを学びました。カーネルモジュールの記述に必要なヘッダファイルは、上記に述べたディレクトリにはなく、Linux カーネルソースコードの include フォルダ内にあります。

- #include <linux/module.h>：カーネルモジュール情報宣言に関連する関数が含まれています。
- #include <linux/init.h>：module_init()および module_exit()関数の宣言が含まれています。
- #include <linux/kernel.h>：printk のようなカーネルが提供する各種関数が含まれています。

カーネルモジュールの記述に頻繁に使用されるヘッダファイルには、<linux/init.h>と <linux/module.h>の二つがあります。ヘッダファイルの前にあるフォルダ名 linux は、include フォルダの下の linux フォルダに対応しています。このフォルダに行って、これら二つのヘッダファイルにどのような内容が含まれているかを確認します。

リスト 2: init.h ヘッダファイル（カーネルソースコード/include/linux/init.h に位置する）

```
1 #define pure_initcall(fn) __define_initcall(fn, 0)
2
3 #define core_initcall(fn) __define_initcall(fn, 1)
4 #define core_initcall_sync(fn) __define_initcall(fn, 1s)
5 #define postcore_initcall(fn) __define_initcall(fn, 2)
6 #define postcore_initcall_sync(fn) __define_initcall(fn, 2s)
7 #define arch_initcall(fn) __define_initcall(fn, 3)
8 #define arch_initcall_sync(fn) __define_initcall(fn, 3s)
```

株式会社日昇テクノロジー

```
9 #define subsys_initcall(fn) __define_initcall(fn, 4)

10 #define subsys_initcall_sync(fn) __define_initcall(fn, 4s)

11 #define fs_initcall(fn) __define_initcall(fn, 5)

12 #define fs_initcall_sync(fn) __define_initcall(fn, 5s)

13 #define rootfs_initcall(fn) __define_initcall(fn, rootfs)

14 #define device_initcall(fn) __define_initcall(fn, 6)

15 #define device_initcall_sync(fn) __define_initcall(fn, 6s)

16 #define late_initcall(fn) __define_initcall(fn, 7)

17 #define late_initcall_sync(fn) __define_initcall(fn, 7s)

18

19 #define __initcall(fn) device_initcall(fn)

20

21 #define __exitcall(fn) ¥

22 static exitcall_t __exitcall_##fn __exit_call = fn
```

init.h ヘッダファイルには、いくつかのマクロ定義とカーネルの initcall メカニズムが含まれています。

リスト 3: module.h ヘッダファイル（カーネルソースコード/include/linux/module.h に位置）

```
1 /* Generic info of form tag = "info" */

2 #define MODULE_INFO(tag, info) __MODULE_INFO(tag, tag, info)

3

4 #define MODULE_ALIAS(_alias) MODULE_INFO(alias, _alias)

5 #define MODULE_LICENSE(_license) MODULE_INFO(license, _license)
```

株式会社日昇テクノロジー

```
6

7 #define MODULE_AUTHOR(_author) MODULE_INFO(author, _author)

8

9 #define module_init(x) __initcall(x);

10

11 #define module_exit(x) __exitcall(x);
```

このコードには、カーネルモジュールのロード、アンロード関数の宣言といくつかのマクロ定義が記載されています。これらのマクロの中にはオプショナルなものもありますが、MODULE_LICENSE はそのカーネルモジュールのライセンスを指定するもので、必須のものです。

3.1.2.3 モジュールロード/アンロード関数

module_init

マイコンを使用する際、例えばシリアルポートなどの外部デバイスを使用するために初期化関数を呼び出す必要がありました。この関数では、シリアルポートの GPIO を初期化し、ボーレート、データビット、ストップビットなどの関連パラメータを設定します。func_init 関数もカーネルモジュール内で初期化に関連する作業を行います。

リスト 4: カーネルモジュールロード関数（カーネルソースコード/linux/init.h に位置）

```
static int __init func_init(void)

{

}

module_init(func_init);
```

戻り値：

- 0：モジュールの初期化に成功し、/sys/module の下にモジュール名でディレクトリが新規作成

株式会社日昇テクノロジー

されます（下図の赤い枠の部分のように）。

- 非 0：モジュールの初期化に失敗。

```
cat@lubancat:~$ ls /sys/module/
8021q           btintel      dns_resolver    i2c_algo_bit       mac80211
8250            btmrvl       drm             i2c_hid            mali
8250_core       btmrvl_sdio  drm_kms_helper  input_sub_system   midgard_kbase
ahci            btrtl        dynamic_debug   ip_vs              mmcblk
asix            btusb        ehci_hcd        ipv6               nf_conntrack
auth_rpcgss     cdc_ncm      fb              kernel             nfs
b43             cec          fiq_debugger    keyboard           nfs_layout_flexfiles
bfusb           cfg80211     firmware_class  libahci            nfs_layout_nfsv41_files
bifrost_kbase   cifs         ftdi_sio        libata             nfsd
block           configfs     fuse            libertas_tf        nfsv4
bluetooth       cpufreq      g_mass_storage  lkdtm              ntfs
brcmsmac        cpuidle      hci_vhci        lockd              nvme
brd             cryptomgr    hellomodule     loop               nvme_core
bridge          devres       hid             lt6911uxc          nvmem_rockchip_otp
btbcm           dm_mod       hidp            lt8619c            optee
```

C 言語における static キーワードの役割は以下の通りです：

1. static で修飾された静的ローカル変数は、プログラムが終了するまで解放されず、ローカル変数の生存期間を延長します。

2. static で修飾されたグローバル変数は、そのファイル内でのみアクセス可能で、他のファイルからはアクセスできません。

3. static で修飾された関数は、そのファイル内でのみ呼び出すことができ、他のファイルからは呼び出せません。

カーネルモジュールのコードは、実際にはカーネルコードの一部です。もしカーネルモジュールが定義する関数がカーネルソースコード内のある関数と重複していた場合、コンパイラはエラーを報告し、コンパイルが失敗するため、カーネルモジュールのコードに static 修飾子を追加することで、このようなエラーを回避できます。

リスト 5: __init、__initdata マクロ定義（カーネルソースコード/linux/init.h に位置）

```
1 #define __init __section(.init.text) __cold notrace

2 #define __initdata __section(.init.data)
```

上記のコードで、__init および__initdata マクロ定義（カーネルソースコード/linux/init.h に位置）では、__init は関数を修飾し、__initdata は変数を修飾するために使用されます。__init 修飾子を持

株式会社日昇テクノロジー

つ関数は、実行ファイルの__init セクションに配置され、このセクションの内容はモジュールの初期化段階でのみ使用され、初期化が完了するとその内容は解放されます。

リスト 6: module_init マクロ定義

```
1 #define module_init(x) __initcall(x);
```

マクロ定義 module_init は、カーネルがモジュールを初期化する際に、どの関数を使用して初期化するかをカーネルに通知するために使用されます。この関数のアドレスを対応するセクションに追加することで、システム起動時にモジュールが自動的にロードされます。

module_exit

モジュールロードの内容を理解した後、モジュールアンロード関数の学習は比較的簡単です。カーネルロード関数とは逆に、カーネルモジュールアンロード関数 func_exit は、初期化段階で割り当てられたメモリや割り当てられたデバイス番号などを解放するために主に使用され、初期化プロセスの逆のプロセスです。

リスト 7: カーネルモジュールアンロード関数

```
1 static void __exit func_exit(void)
2 {
3 }
4 module_exit(func_exit);
```

func_init 関数との違いは、この関数の戻り値が void 型であり、修飾子も異なることです。ここでは__exit を使用し、この関数を実行ファイルの__exit セクションに配置します。モジュールのアンロード段階が完了すると、このエリアのスペースが自動的に解放されます。

リスト 8: __exit、__exitdata マクロ定義

```
1 #define __exit __section(.exit.text) __exitused __cold notrace
2 #define __exitdata __section(.exit.data)
```

株式会社日昇テクノロジー

モジュールロード関数と同様に、__exit は関数を修飾し、__exitdata は変数を修飾するために使用されます。マクロ定義 module_exit は、モジュールをアンロードする際に、どの関数を呼び出す必要があるかをカーネルに通知するために使用されます。

printk 関数

- printf：glibc によって実装されたプリント関数で、ユーザースペースで動作します。
- printk：カーネルモジュールは glibc ライブラリ関数を使用できないため、カーネル自身が実装した printf 関数に似た関数ですが、プリントレベルを指定する必要があります。
- #define KERN_EMERG "<0>" 通常はシステムクラッシュ前の情報
- #define KERN_ALERT "<1>" 即時対応が必要なメッセージ
- #define KERN_CRIT "<2>" 重大な状況
- #define KERN_ERR "<3>" エラー状態
- #define KERN_WARNING "<4>" 問題がある状況
- #define KERN_NOTICE "<5>" 注意情報
- #define KERN_INFO "<6>" 一般メッセージ
- #define KERN_DEBUG "<7>" デバッグ情報

現在のシステムの printk プリントレベルを見るには：cat /proc/sys/kernel/printk、左から順に現在のコンソールログレベル、デフォルトのメッセージログレベル、最小のコンソールレベル、デフォルトのコンソールログレベルに対応します。

```
cat@lubancat:~$ cat /proc/sys/kernel/printk
4       4       1       7
cat@lubancat:~$
```

カーネルのすべてのプリント情報を表示するには：dmesg、ただしカーネルログバッファのサイズには制限があり、バッファ内のデータが上書きされる可能性があります。

株式会社日昇テクノロジー

3.1.2.4 ライセンス

Linux は無料のオペレーティングシステムで、GPL ライセンスを採用しており、ユーザーはそのソースコードを自由に変更することができます。GPL ライセンスの主な内容は、ある GPL ライセンス製品が提供するライブラリを使用して新しい製品が派生した場合でも、そのソフトウェア製品は GPL ライセンスを採用しなければならず、つまりオープンソースで無料で使用可能である必要があります。したがって、GPL ライセンスは伝染性があると言えます。そのため、Linux ではさまざまな無料ソフトウェアを使用することができます。Linux を学習する過程で、30 日間の試用期間やアクティベーションコードの入力を求められることは決してありません。

リスト 9: ライセンス

```
1 #define MODULE_LICENSE(_license) MODULE_INFO(license, _license)
```

カーネルモジュールのライセンスには「GPL」、「GPL v2」、「GPL and additional rights」、「Dual BSD/GPL」、「Dual MPL/GPL」、「Proprietary」があります。

3.1.2.5 関連情報宣言

ここで、カーネルモジュールプログラム構造の最後の部分について説明します。この部分は、そのモジュールを使用する読者に「取扱説明書」を提供するためのものであり、あってもなくても大きな影響はありませんが、あれば役立つ情報です。

カーネルモジュール情報宣言関

関数	機能
MODULE_LICENSE()	モジュールコードが受け入れるソフトウェアライセンス契約を示します。Linux カーネルは GPL V2 オープンソースライセンスに従っているので、カーネルモジュールも Linux カーネルと一致することが望ましいです。
MODULE_AUTHOR()	モジュールの作者情報を記述します。
MODULE_DESCRIPTION()	モジュールの簡単な紹介です。
MODULE_ALIAS()	モジュールに別名を設定します。

株式会社日昇テクノロジー

作者情報

リスト 10: カーネルモジュール作者マクロ定義（カーネルソースコード/linux/module.h に位置）

```
1 #define MODULE_AUTHOR(_author) MODULE_INFO(author, _author)
```

前述の modinfo で表示されるモジュール情報の中の「author」情報は、マクロ定義 MODULE_AUTHOR から来ています。このマクロ定義は、そのモジュールの作者を宣言するために使用されます。

モジュール情報

リスト 11: モジュール記述情報(カーネルソースコード/linux/module.h に位置)

```
1 #define MODULE_DESCRIPTION(_description) MODULE_INFO(description, description)
```

モジュール情報内の「description」情報は、マクロ MODULE_DESCRIPTION に由来しています。このマクロは、そのモジュールの機能や目的を記述するために使用されます。

モジュール別名

リスト 12: カーネルモジュール別名マクロ定義(カーネルソースコード/linux/module.h に位置)

```
1 #define MODULE_ALIAS(_alias) MODULE_INFO(alias, _alias)
```

モジュール情報内の「alias」情報は、マクロ定義 MODULE_ALIAS によって提供されます。このマクロ定義は、カーネルモジュールに別名を付けるために使用されます。このモジュールの別名を使用する場合、そのモジュールを/lib/modules/カーネルソースコード/下にコピーし、コマンド depmod を使用してモジュールの依存関係を更新する必要があります。そうでなければ、Linux カーネルはこのモジュールが別の名前を持っていることをどのように知ることができるでしょうか。

3.1.3 実験準備

カーネルモジュールのソースコードを取得し、付属のコード linux_driver をカーネルコードと同じ

株式会社日昇テクノロジー

ディレクトリレベルに解凍した後、linux_driver/module/hellomodule フォルダに入ります。

3.1.3.1 Makefile 説明

カーネルモジュールにとって、それはカーネルの一部のコードであり、ただしカーネルソースコード内には存在しません。そのため、コンパイルする際にはカーネルソースコードディレクトリ下でコンパイルする必要があります。カーネルモジュールを使用して Makefile ファイルをコンパイルする際に使用される Makefile ファイルは、以前 C コードをコンパイルする際に使用された Makefile ファイルと大体同じです。これは、Linux カーネルのコンパイルに使用される Kbuild システムのおかげです。そのため、カーネルモジュールをコンパイルする際には、環境変数 ARCH と CROSS_COMPILE の値を指定する必要があります。

リスト 13: Makefile(linux_driver/module/hellomodule/Makefile)

```
1 KERNEL_DIR=../../../kernel/

2

3 ARCH=arm64

4 CROSS_COMPILE=aarch64-linux-gnu-

5 export ARCH CROSS_COMPILE

6

7 obj-m := hellomodule.o

8 all:

9 $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) modules

10
```

株式会社日昇テクノロジー

```
11 .PHONE:clean
12
13 clean:
14 $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) clean
```

上記のコードは、カーネルモジュールをコンパイルするための Makefile について提供されています。

- 第 1 行：この Makefile は変数 KERNEL_DIR を定義しており、カーネルソースコードのディレクトリを保存するために使用されます。カーネルコンパイル出力ディレクトリに指定する必要があります。
- 第 3-5 行：ツールチェーンを指定し、環境変数をエクスポートしています
- 第 6 行：変数 obj-m は、モジュールとしてコンパイルする必要があるターゲットファイル名を保存します。
- 第 8 行：'$(MAKE)modules'は実際には Linux トップレベル Makefile の疑似ターゲット modules を実行しています。オプション'-C'を使用すると、make ツールがソースコードディレクトリにジャンプしてトップレベル Makefile を読み込むことができます。「M=$(CURDIR)」は現在のディレクトリに戻り、現在のディレクトリの Makefile を読み込んで実行し、カーネルモジュールのコンパイルを開始します。CURDIR は make の組み込み変数で、自動的に現在のディレクトリに設定されます。

3.1.3.2 コンパイルコマンド説明

実験ディレクトリ linux_driver/module/hellomodule で以下のコマンドを入力してドライバーモジュールをコンパイルします：

```
make
```

コンパイルが成功すると、実験ディレクトリに「hellomodule.ko」という名前のドライバーモジュー

株式会社日昇テクノロジー

ルファイルが生成されます。

3.1.4 プログラム実行結果

3.1.4.1 カーネルモジュールのロード方法

望み通りに自身のカーネルモジュールをコンパイルしたので、次にこのカーネルモジュールをどのように使用するかを理解する必要があります。hellomodule.ko を scp や NFS を通じて開発ボードにコピーします。これらのツールについて一つずつ説明します。

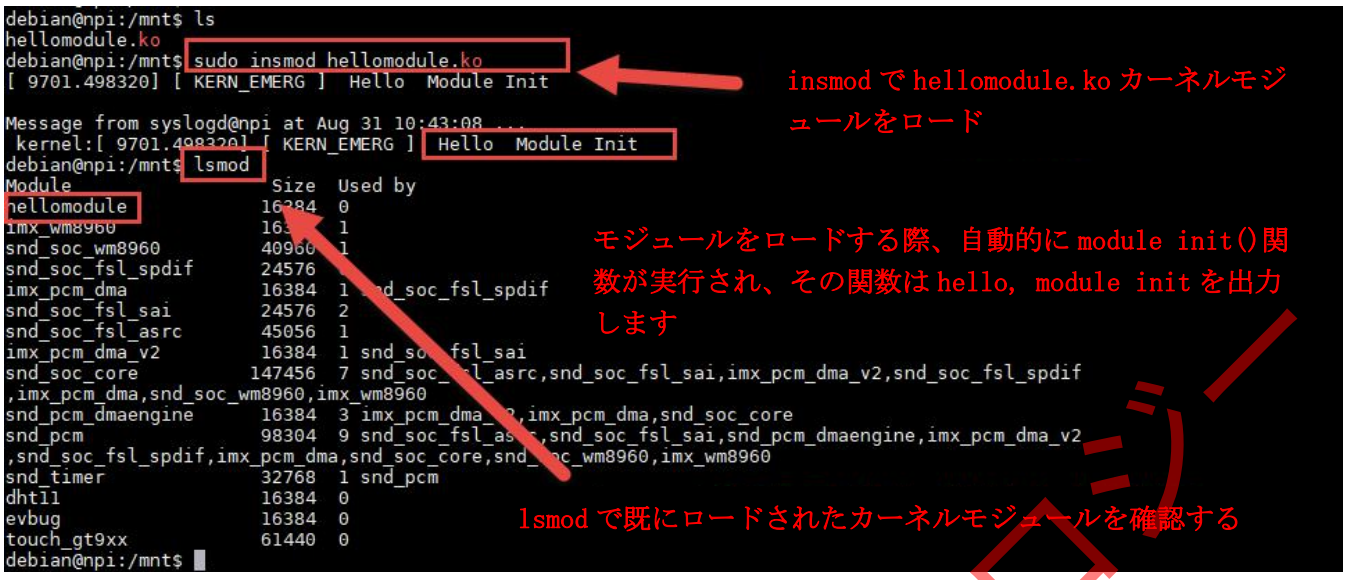
lsmod

lsmod は、現在のカーネルにロードされているモジュールをリストし、端末にフォーマットされた表示を行います。その原理は、/proc/modules 内の情報をフォーマットに調整して出力することです。lsmod の出力リストには「Used by」という列があり、これはこのモジュールが他のモジュールによって使用されていることを示し、モジュール間の依存関係を表示します。

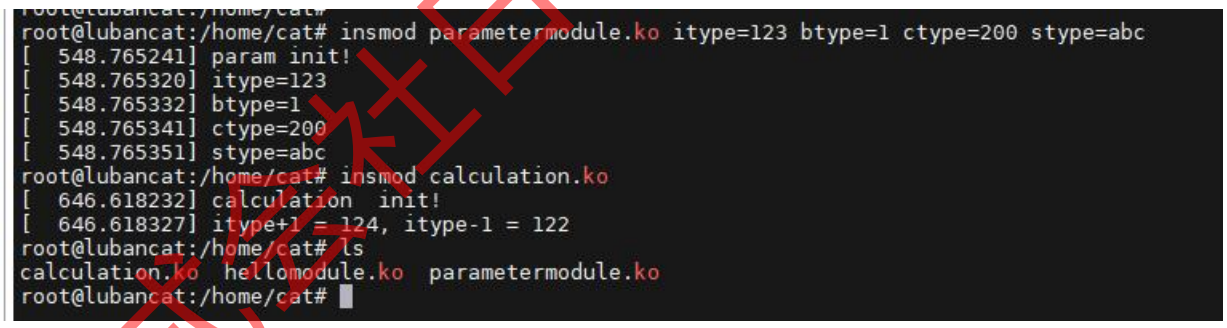
insmod

モジュールをカーネルにロードする最も簡単な方法は insmod です。insmod + モジュールの完全パスで目的を達成できます。前提として、モジュールが他のモジュールに依存していないこと、また sudo 権限が必要であることに注意してください。他のモジュールのシンボルを使用しているかどうか不確かな場合は、modprobe を試すこともできます。後ほどその詳細な使用法について説明します。insmod コマンドで hellomodule.ko メモリモジュールをロードすると、このメモリモジュールは自動的に module_init()関数を実行し、初期化操作を行います。この関数は「hello module init」と表示します。再びシステムにロードされたカーネルモジュールを確認すると、リストに hellomodule.ko の姿が見えます。

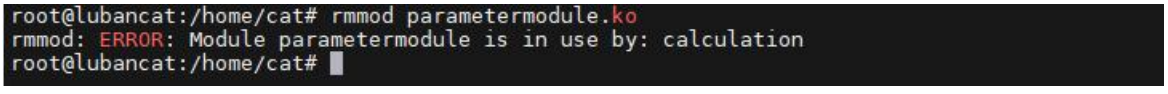
株式会社日昇テクノロジー



しかし、一部のカーネルモジュールには依存関係があり、直接 insmod でロードすることはできません。依存しているモジュールを「前置」してロードした後に、そのモジュールをロードする必要があります。後続の実験である「カーネルモジュールのパラメータ渡しとシンボル共有実験」のセクションでは、calculation.ko と parametermodule.ko には依存関係があります。calculation.ko は parametermodule.ko のパラメータと関数に依存しているため、先に手で parametermodule.ko をロードし、その後で calculation.ko をロードする必要があります。



同様に、アンロードする際には、parametermodule.ko のパラメータと関数が calculation.ko によって呼び出されているため、先に calculation.ko をアンロードし、その後で parametermodule.ko をアンロードする必要があります。そうしないと、「ERROR: Module parametermodule is in use by: calculation」というエラーが発生します。



modprobe

modprobe は insmod と同じ機能を持ち、モジュールをカーネルにロードすることができます。それに加えて、modprobe はモジュール間の依存関係をチェックし、それらを順番にロードすることができます。これは、insmod を順番に複数回実行することと同じです。

後続の実験である「カーネルモジュールのパラメータ渡しとシンボル共有実験」では、calculation.ko と parametermodule.ko を順番にロードする必要があります。modprobe ツールを使用すると、parametermodule.ko を直接ロードできます。もちろん、modprobe を使用する前に、depmod -a を使用してモジュール間の依存関係を構築する必要があります。しかし、depmod -a を使用するには、ドライバーモジュールをシステムのドライバーモジュール格納および設定フォルダに配置する必要があります。そうしないと、依存関係を管理することはできません。

depmod

modprobe が特定のモジュールが依存している他のモジュールをどのように知っているのでしょうか？このプロセスでは、depend が決定的な役割を果たしています。modprobe を実行すると、モジュールのインストールディレクトリ下で module.dep ファイルを検索します。これは depmod が作成したモジュール依存関係のファイルです。

Linux システムでは、/lib/modules ディレクトリは通常、カーネル関連のモジュールと設定ファイルを含んでいます。このフォルダには、カーネルバージョン番号に関連するフォルダが含まれており、モジュールと設定情報を格納するために使用されます。

```
root@lubancat:/home/cat# ls /lib/modules/4.19.232/ -l
total 116
lrwxrwxrwx 1 root root    31 Jul 10 16:21 build -> /usr/src/linux-headers-4.19.232
drwxr-xr-x 4 root root  4096 Aug 26 23:20 kernel
-rw-r--r-- 1 root root  4794 Aug 27 16:40 modules.alias
-rw-r--r-- 1 root root  5820 Aug 27 16:40 modules.alias.bin
-rw-r--r-- 1 root root 28136 Aug 23 16:53 modules.builtin
-rw-r--r-- 1 root root 30808 Aug 27 16:40 modules.builtin.bin
-rw-r--r-- 1 root root  1510 Aug 27 16:40 modules.dep
-rw-r--r-- 1 root root  2584 Aug 27 16:40 modules.dep.bin
-rw-r--r-- 1 root root    74 Aug 27 16:40 modules.devname
-rw-r--r-- 1 root root  1023 Aug 23 16:53 modules.order
-rw-r--r-- 1 root root    55 Aug 27 16:40 modules.softdep
-rw-r--r-- 1 root root  5168 Aug 27 16:40 modules.symbols
-rw-r--r-- 1 root root  6361 Aug 27 16:40 modules.symbols.bin
root@lubancat:/home/cat#
```

株式会社日昇テクノロジー

以上の設定ファイルまたはディレクトリの説明は以下の通りです

設定ファイルまたはフォルダ	役割
build	現在実行中のカーネルソースコードへのシンボリックリンク
kernel	ンパイル後のカーネルモジュールファイル（.ko）が含まれています
modules.alias	モジュールエイリアスを定義するファイル
modules.alias.bin	モジュールエイリアスファイルのバイナリキャッシュバージョン
modules.builtin	カーネルによってビルドされたモジュール（カーネル内に静的にリンクされている）をリストアップ
modules.builtin.bin	カーネルによってビルドされたモジュールリストのバイナリキャッシュバージョン
modules.dep	モジュール間の依存関係をリストアップ
modules.dep.bin	モジュール依存関係ファイルのバイナリキャッシュバージョン
modules.devname	各モジュールデバイスの名前が含まれています
modules.order	モジュールのロード順序を定義するファイル
modules.symbols	エクスポートされたシンボル情報を保存
modules.symbols.bin	エクスポートされたシンボル情報のバイナリキャッシュバージョン
modules.softdep	モジュールのソフト依存関係が含まれるファイル

最も関心がある設定ファイルは modules.dep で、このファイルにはモジュール間の依存関係がリストされています。depmod -a を実行してモジュール間の依存関係を構築すると、その依存関係が modules.dep に書き込まれます。

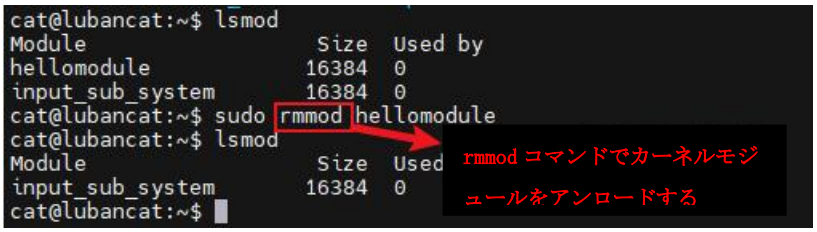
rmmod

rmmod ツールは、単にカーネルで実行中のモジュールを削除するもので、パスを渡すだけで実現できます。

rmmod コマンドで特定のメモリモジュールをアンロードすると、メモリモジュールは自動的に *_exit()関数を実行し、クリーンアップ操作を行います。hellomodule の*_exit()関数は 1 行の内容を出力しましたが、コンソールには表示されませんでした。これは printk のプリントレベルに関連しているため、dmesg を使用して確認することができます。rmmod は、モジュールが依存しているモジュールをアンロードしません。依存するモジュールを順番にアンロードする必要があります。もちろ

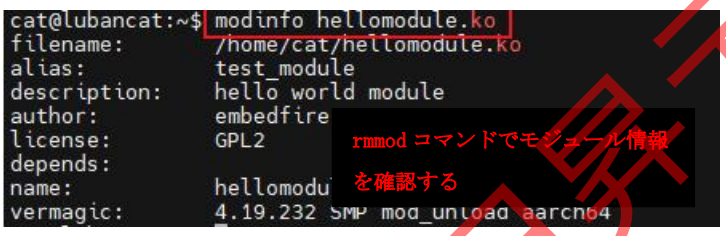
株式会社日昇テクノロジー

ん、modprobe -r を使用すると、ワンクリックでアンロードできます。



Modinfo

modinfo は、カーネルモジュール内で定義されたいくつかのマクロを表示するために使用されます。modinfo を使用して hellomodule を見ると、このモジュールが GPL プロトコルに従っていること、モジュールの作成者が embedfire であること、モジュールの vermagic などの情報を出力から知ることができます。これらの情報は、モジュールコード内の関連するカーネルモジュール情報宣言関数によって宣言されています。



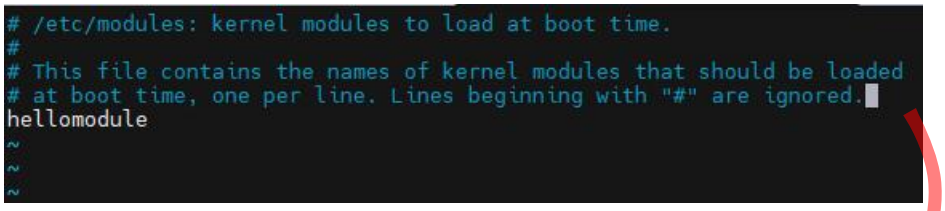
3.1.4.2 システム自動ロードモジュール

自分でモジュールを書いた場合、あるいは開発ボードの起動時に自動的にロードさせるにはどうすればいいのでしょうか？ここでは、上述の depmod と modprobe ツールを使用する必要があります。まず、自動ロードしたいモジュールを"/lib/modules/カーネルバージョン"ディレクトリに統一して配置する必要があります。カーネルバージョンは uname -r で照会し、テストでは 4.19.232 を使用しています。次に、depmod -a コマンドを使用してモジュール間の依存関係を構築します。この時点で、modules.dep にモジュールの依存関係を見ることができます。以下のコマンドを使用して確認できます。

```
cat /lib/modules/4.19.232/modules.dep | grep hellomodule
```

株式会社日昇テクノロジー

最後に、/etc/modules または/etc/modules-load.d/<filename>.conf に自分のモジュールを追加します。この設定ファイル内で、モジュールを.ko 形式で記述しない場合、そのモジュールはカーネルと密接に結合されることを意味します。例えば、mm サブシステムなど、一部はシステムがカーネルと密接に結合する必要がありますが、一般的には.ko 形式で記述する方が良いでしょう。エラーが発生してもカーネルパニックにならないようにするためです。カーネルに統合されると、エラーが発生するとパニックが発生します。



開発ボードを再起動すると、lsmod でモジュールが起動時にカーネルにロードされているのを確認できます。

3.2 カーネルモジュールのパラメータ渡しとシンボル共有実験

3.2.1 実験説明

3.2.1.1 ハードウェア紹介

この実験セクションでは Lubancat_RK のボードを使用します。

3.2.2 実験コード解説

この章のサンプルコードディレクトリは：linux_driver/module/parametermodule

この実験セクションでは、カーネルモジュールのパラメータ渡しとシンボル共有を検証します。

3.2.2.1 カーネルモジュールのパラメータ渡しコード解説

カーネルモジュールは、Linux カーネルに柔軟性を提供する拡張可能な動的モジュールとして機能しますが、時には異なるアプリケーションシナリオに応じてカーネルに異なるパラメータを渡す必要があります。例えば、プログラム内でデバッグモードを有効にする、詳細出力モードを設定する、また

株式会社日昇テクノロジー

は特定のモジュールに関連するオプションを指定するなど、これらはすべてパラメータの形でモジュールの挙動を変更することができます。

Linux カーネルは、モジュールのパラメータ渡しを実現するためにマクロを提供しています。

リスト 14: module_param 関数(カーネルソースコード/include/linux/moduleparam.h)

```
1 #define module_param(name, type, perm) ¥¥

2 module_param_named(name, name, type, perm)

3 #define module_param_array(name, type, nump, perm) ¥¥

4 module_param_array_named(name, name, type, nump, perm)
```

- name：定義した変数名；
- type：パラメータのタイプ。現在カーネルがサポートしているパラメータタイプには byte、short、ushort、int、uint、long、ulong、charp、bool、invbool があります。ここで、charp は文字ポインタを意味し、bool はブール型で、その値は 0 または 1 のみです。invbool は逆ブール型で、その値も 0 または 1 のみですが、true の値は 0 を、false の値は 1 を意味します。char 型の変数の場合、パラメータは byte のみ、char*の場合は charp のみです。
- perm：ファイルの権限を表します。具体的なパラメータ値は以下の表を参照してください。

表ファイル権限

リスト 1: ファイル権限:ヘッダー: "ユーザーグループ", "フラグ", "説明"

:widths: 10, 10,30

現在のユーザー	S_IRUSR	ユーザーは読み取り権限を持つ
	S_IWUSR	ユーザーは書き込み権限を持つ
現在のユーザーグループ	S_IRGRP	現在のユーザーグループの他のユーザーは読み取り権限を持つ
	S_IWUSR	現在のユーザーグループの他のユーザーは書き込み権限を持つ
他のユーザー	S_IROTH	他のユーザーは読み取り権限を持つ

株式会社日昇テクノロジー

	S_IWOTH	

上述のファイル権限には、実行権限の設定が含まれていません。このファイルが実行権限を持つことは許可されていないことに注意してください。もし S_IXUGO という実行権限を示すパラメータ値を強制的にそのパラメータに割り当てた場合、最終的に生成されたカーネルモジュールをロードする際にエラーが発生します。

```
[76535.102167] ------------[ cut here ]------------
[76535.102211] WARNING: CPU: 0 PID: 1911 at /home/jinsheng/imx6ull/4.19/ebf-buster-linux/fs/sysfs/group.c:60 internal_create_group+0x
378/0x3c0
[76535.102268] Attribute btype: Invalid permissions 0700
[76535.102274] Modules linked in: module_param(PO+) helloworld(PO) imx_wm8960(O) snd_soc_wm8960 snd_soc_fsl_asrc snd_soc_fsl_sai snd_
soc_fsl_spdif imx_pcm_dma imx_pcm_dma_v2 snd_soc_core snd_pcm_dmaengine snd_pcm snd_timer evbug dht11(O) touch_gt9xx(O) [last unloade
d: module_param]
[76535.102358] CPU: 0 PID: 1911 Comm: insmod Tainted: P        W  O      4.19.71-imx-r1 #1stable
```

以下は実験コードの一例です：

リスト 15: サンプルプログラム

```
1 static int itype=0;
2 module_param(itype,int,0);
3
4 static bool btype=0;
5 module_param(btype,bool,0644);
6
7 static char ctype=0;
8 module_param(ctype,byte,0);
9
10 static char *stype=0;
11 module_param(stype,charp,0644);
12
```

株式会社日昇テクノロジー

```
13 static int __init param_init(void)
14 {
15 printk(KERN_ALERT "param init!¥n");
16 printk(KERN_ALERT "itype=%d¥n",itype);
17 printk(KERN_ALERT "btype=%d¥n",btype);
18 printk(KERN_ALERT "ctype=%d¥n",ctype);
19 printk(KERN_ALERT "stype=%s¥n",stype);
20 return 0;
21 }
```

• 第 1-11 行：4 つの一般的な変数を定義し、module_param マクロを使用してこれら 4 つのパラメータを宣言しています

• 第 13-21 行：param_init 内で上述の 4 つのパラメータを出力します。

定義した 4 つのモジュールパラメータは、「/sys/module/モジュール名/parameters」下にモジュールパラメータ名のファイルとして存在します。itype と ctype の権限が 0 なので、これらのパラメータを閲覧する権限はありません。

```
root@lubancat:~# ls -al /sys/module/parametermodule/parameters/
total 0
drwxr-xr-x 2 root root    0 Aug 27 17:41 .
drwxr-xr-x 6 root root    0 Aug 27 17:41 ..
-rw-r--r-- 1 root root 4096 Aug 27 17:42 btype
-rw-r--r-- 1 root root 4096 Aug 27 17:42 stype
root@lubancat:~#
```

3.2.2.2 シンボル共有コード解説

前で、エクスポートされたシンボルについて詳しく分析しました。シンボルとは、カーネルモジュール内でエクスポートされた関数や変数のことで、モジュールがロードされる時に公共カーネルシンボルテーブルに記録され、他のモジュールから呼び出されます。このメカニズムにより、階層的な考え方を用いて、複雑なモジュール設計の問題を解決することができます。ドライバを書く際には、ド

株式会社日昇テクノロジー

ライバを機能に応じていくつかのカーネルモジュールに分け、シンボル共有を利用してモジュール間のインターフェース呼び出しや変数共有を実現できます。

リスト 16: シンボルのエクスポート

```
1 #define EXPORT_SYMBOL(sym) ¥¥
2 __EXPORT_SYMBOL(sym, "")
```

EXPORT_SYMBOL マクロは、シンボルをカーネルにエクスポートするために使用されます。これにより、他のモジュールもエクスポートしたシンボルを使用できるようになります。

以下のコードを通して、モジュールがシンボルをエクスポートする方法を紹介します。

リスト 17: parametermodule.c

```
1 //... コード省略...
2 static int itype=0;
3 module_param(itype,int,0);
4
5 EXPORT_SYMBOL(itype);
6
7 int my_add(int a, int b)
8 {
9 return a+b;
10 }
11
12 EXPORT_SYMBOL(my_add);
13
14 int my_sub(int a, int b)
```

株式会社日昇テクノロジー

```
15 {
16 return a-b;
17 }
18
19 EXPORT_SYMBOL(my_sub);
20 //... コード省略...
```

- 第 2-3 行：パラメータ itype を定義し、EXPORT_SYMBOL マクロでエクスポート
- 第 7-12 行：関数 my_add を定義し、EXPORT_SYMBOL マクロでエクスポート
- 第 14-21 行：関数 my_sub を定義し、EXPORT_SYMBOL マクロでエクスポート

上記のコードでは、カーネルモジュールプログラムの他の内容（ヘッダーファイル、ロード/アンロード関数など）は省略されています。

リスト 18: calculation.h

```
1 #ifndef __CALCULATION_H__
2 #define __CALCULATION_H__
3
4 extern int itype;
5
6 int my_add(int a, int b);
7 int my_sub(int a, int b);
8
9 #endif
```

追加の変数 itype、関数 my_add、my_sub を宣言します。

株式会社日昇テクノロジー

リスト 19: calculation.c

```
1 //... コード省略...
2 #include "calculation.h"
3
4 //... コード省略...
5 static int __init calculation_init(void)
6 {
7 printk(KERN_ALERT "calculation init!¥n");
8 printk(KERN_ALERT "itype+1 = %d, itype-1 = %d¥n", my_add(itype,1), my_sub(itype,1));
9 return 0;
10 }
11 //... コード省略...
```

calculation.c では、extern キーワードを使用してパラメータ itype を宣言し、my_add()、my_sub()関数を呼び出して計算を行います。

3.2.3 実験準備

カーネルモジュールのソースコードを取得し、付属のコード linux_driver/をカーネルコードと同じディレクトリレベルに配置し、その後 linux_driver/module/parametermodule ディレクトリに入ります。

株式会社日昇テクノロジー

3.2.3.1 makefile 説明

リスト 20: Makefile (linux_driver/module/hellomodule/Makefile に位置)

```
1 ... コード省略...
2 obj-m := parametermodule.o calculation.o
3 ... コード省略...
```

上記の Makefile は、前の実験とはターゲットファイルが異なる以外は同じです。

3.2.3.2 コンパイルコマンド説明

実験ディレクトリで以下のコマンドを入力してドライバモジュールをコンパイルします：

```
make
```

```
csun@ubuntu:~/LubanCat_SDK/linux_driver/module/parametermodule$ make
make -C ../../../kernel/ M=/home/csun/LubanCat_SDK/linux_driver/module/parametermodule modules
make[1]: Entering directory '/home/csun/LubanCat_SDK/kernel'
  CC [M]  /home/csun/LubanCat_SDK/linux_driver/module/parametermodule/parametermodule.o
  CC [M]  /home/csun/LubanCat_SDK/linux_driver/module/parametermodule/calculation.o
WARNING: Symbol version dump "Module.symvers" is missing.
         Modules may not have dependencies or modversions.
  MODPOST /home/csun/LubanCat_SDK/linux_driver/module/parametermodule/Module.symvers
WARNING: modpost: Symbol info of vmlinux is missing. Unresolved symbol check will be entirely skipped.
  CC [M]  /home/csun/LubanCat_SDK/linux_driver/module/parametermodule/calculation.mod.o
  LD [M]  /home/csun/LubanCat_SDK/linux_driver/module/parametermodule/calculation.ko
  CC [M]  /home/csun/LubanCat_SDK/linux_driver/module/parametermodule/parametermodule.mod.o
  LD [M]  /home/csun/LubanCat_SDK/linux_driver/module/parametermodule/parametermodule.ko
make[1]: Leaving directory '/home/csun/LubanCat_SDK/kernel'
csun@ubuntu:~/LubanCat_SDK/linux_driver/module/parametermodule$ ls
calculation.c  calculation.ko   calculation.mod.c  calculation.o  modules.order   parametermodule.c   parametermodule.mod    parametermodule.mod.o
calculation.h  calculation.mod  calculation.mod.o  Makefile       Module.symvers  parametermodule.ko  parametermodule.mod.c  parametermodule.o
```

コンパイルが成功すると、実験ディレクトリに「parametermodule.ko」と「calculation.ko」という名前のドライバモジュールファイルが生成されます。

3.2.4 プログラム実行結果

NFS を通してコンパイル済みの module_param.ko を開発ボードにコピーし、module_param.ko をロードしてパラメータを渡します。すると、宣言した 4 つの変数の値が、割り当てた値になります。

```
sudo insmod parametermodule.ko itype=123 btype=1 ctype=200 stype=abc
```

カーネルにエクスポートされたシンボルテーブルを見るには「cat /proc/kallsyms」を実行します。

株式会社日昇テクノロジー

カーネルモジュールを/lib/modules/4.19.232/kernel ディレクトリに配置し、その後 depmod -a を実行すると、modules.dep 設定ファイルで calculation.ko が parametermodule.ko に依存していること、およびモジュールの配置場所が記録されていることが分かります。以下の図のように表示されます：

```
root@lubancat:/lib/modules/4.19.232# ls kernel/
calculation.ko  drivers  hellomodule.ko  net  parametermodule.ko
root@lubancat:/lib/modules/4.19.232# depmod -a
root@lubancat:/lib/modules/4.19.232# cat modules.dep | grep calculation
kernel/calculation.ko: kernel/parametermodule.ko
root@lubancat:/lib/modules/4.19.232#
```

modprobe calculation コマンドを実行すると、自動的に parametermodule.ko モジュールがロードされ、その後に calculation.ko モジュールがロードされます。

```
root@lubancat:/lib/modules/4.19.232# modprobe calculation
[ 2843.628141] param init!
[ 2843.628291] itype=0
[ 2843.628304] btype=0
[ 2843.628316] ctype=0
[ 2843.628328] stype=(null)
[ 2843.632803] calculation  init!
[ 2843.632928] itype+1 = 1, itype-1 = -1
root@lubancat:/lib/modules/4.19.232#
root@lubancat:/lib/modules/4.19.232# lsmod
Module                  Size  Used by
calculation            16384  0
parametermodule        16384  1 calculation
8821cu               2482176  0
rtk_btusb              57344  0
btusb                  40960  0
btrtl                  16384  1 btusb
btbcm                  16384  1 btusb
btintel                20480  1 btusb
bluetooth             450560  23 btrtl,btintel,btbcm,btusb
root@lubancat:/lib/modules/4.19.232#
```

第 4 章文字デバイスドライバ

前章では、カーネルモジュールとは何か、モジュールのロードおよびアンロードの詳細なプロセス、カーネルモジュールの使用について学びました。この章では、文字デバイスの使用、文字デバイスドライバに関連する概念について学び、文字デバイスドライバプログラムの基本フレームワークを理解し、文字デバイスドライバの実装と管理についてソースコードから分析します。主に以下の五つの部分に分かれています：

1. Linux デバイスの分類
2. 文字デバイスの抽象化、文字デバイスの設計思想

株式会社日昇テクノロジー

3. 文字デバイスに関連する概念およびデータ構造、デバイス番号などの基本概念や file_operations、file、inode に関連するデータ構造の理解

4. 文字デバイスドライバプログラムのフレームワーク、例えばカーネルがデバイス番号をどのように管理するか、システムが file_operation インターフェースを関連付け、呼び出すプロセス、open 関数に関連する知識など

5. デバイスドライバプログラムの実験

4.1 Linux デバイスの分類

Linux はファイルシステムベースであり、すべてのハードウェアは対応するディレクトリ(/dev)下に該当するファイルで表されます。Windows システムでは、デバイスと言えばハードディスクやディスクドライブなどの実際のハードウェアを指しますが、ファイルシステムの Linux では、これらのデバイスに関連付けられたファイルを通じて実際のハードウェアにアクセスできます。ファイルにアクセスするようにハードウェアデバイスを操作することで、すべてがはるかに簡単になります。データの読み書き方法に基づいて、デバイスを次の数種類に分類できます：文字デバイス、ブロックデバイス、ネットワークデバイス。

文字デバイス: アプリケーションがデータを文字/バイト単位で読み書きするデバイスを指します。これらのデバイスノードは通常、ファックス、仮想端末、シリアルモデム、キーボードなどのデバイスにストリーム通信サービスを提供します。これらは一般的にランダムアクセスデータをサポートしません。文字デバイスは実装時にほとんどキャッシュを使用しません。システムはデバイスから直接各文字を読み取り/書き込みます。

ブロックデバイス: 通常、ランダムアクセスとアドレッシングをサポートし、キャッシュを使用します。オペレーティングシステムは入出力のためにデータブロックをキャッシュに割り当てます。プログラムがデバイスにデータの読み取りや書き込みを要求すると、システムはデータの各文字を適切なキャッシュに格納します。

株式会社日昇テクノロジー

ネットワークデバイス: 特殊なデバイスで、/dev 下には存在しません。主にネットワークデータの送受信に使用されます。

Linux カーネルは、オブジェクト指向の設計思想を随所に反映しています。デバイスを統一的に扱うために、Linux システムはデバイスを struct cdev、struct block_device、struct net_device の三つのオブジェクトに抽象化しています。具体的なデバイスはこれら三つのオブジェクトを含むことで、それぞれの属性と操作を継承し、各自のドライバモデルに追加されて一元的に管理され、操作されます。

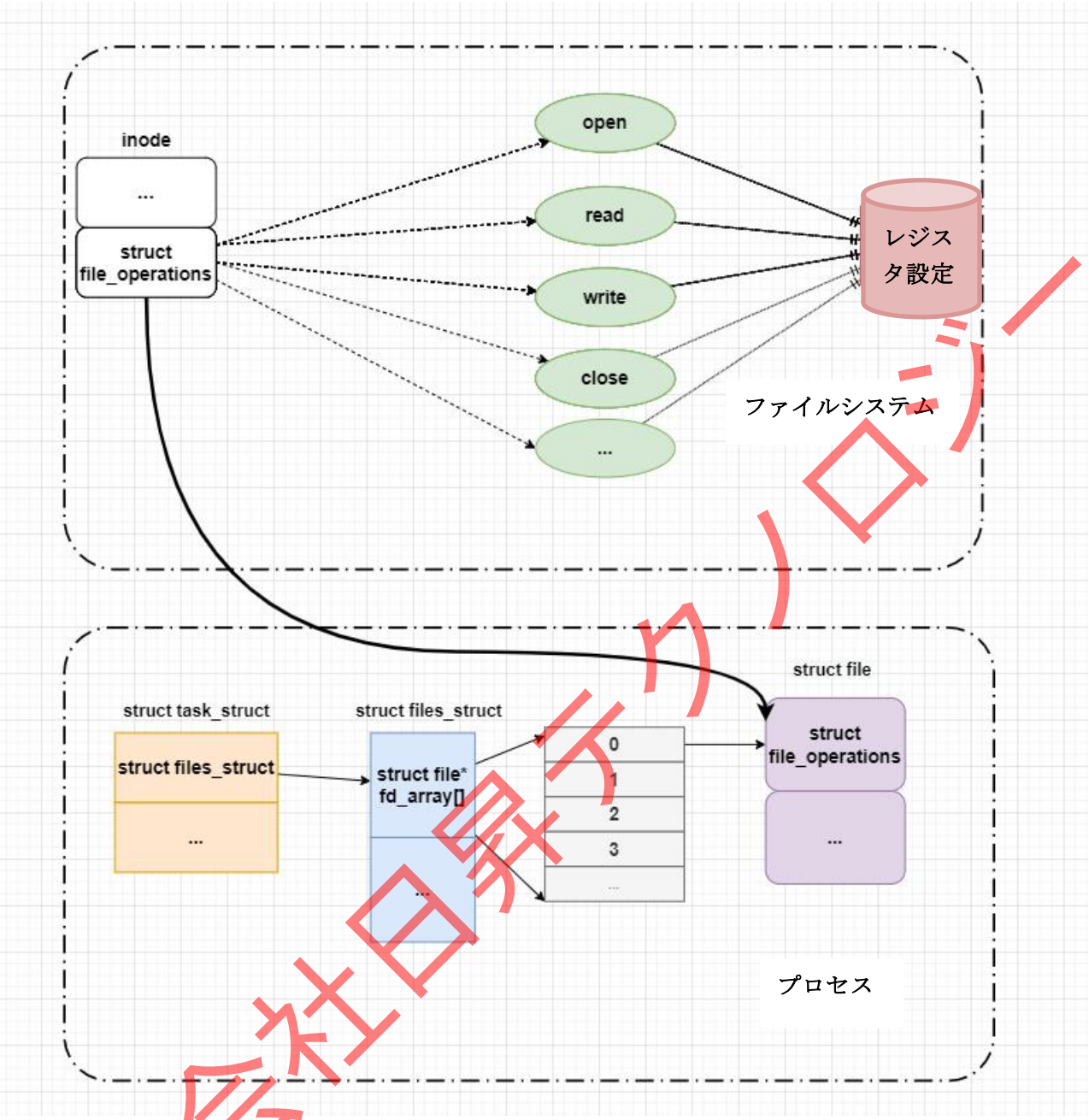
文字デバイスドライバは、多くの単純なハードウェアデバイスに適しており、ブロックデバイスやネットワークドライバよりも理解しやすいため、文字デバイスから学び始めることにしました。

4.2 文字デバイスの抽象化

Linux カーネルは文字デバイスを具体的なデータ構造(struct cdev)として抽象化しています。これを文字デバイスオブジェクトと理解できます。cdev は文字デバイスの関連情報（デバイス番号、カーネルオブジェクト）、開く、読み書き、閉じるなどの操作インターフェース（file_operations）を記録します。文字デバイスを追加したい場合は、

このオブジェクトをカーネルに登録し、デバイスノード（ファイル）を作成して cdev オブジェクトにバインドします。ファイルに対する読み書き操作を行うと、仮想ファイルシステムを介してカーネル内のこのオブジェクトとその操作インターフェースを見つけ出し、デバイスを制御できます。

株式会社日昇テクノロジー



ハードウェア層では、ハードウェアの回路図やチップのデータシートを確認することで、下層で設定する必要があるレジスタを特定できます。これは、ベアメタル開発に似ており、下層のレジスタの設定や読み書き操作をファイル操作インターフェース内で行います。つまり、file_operations 構造体を実装することです。ドライバ層では、ファイル操作インターフェースをカーネルに登録し、カーネルは内部のハッシュテーブルを使用して主要・副デバイス番号を記録します。ファイルシステム層では、そのファイル操作インターフェースにバインドされた新しいファイルを作成し、アプリケーションは指定されたファイルのファイル操作インターフェースを操作して下層のレジスタを設定します。

株式会社日昇テクノロジー

実際には、Linux でドライバプログラムを書くことは、「穴埋め」のようなものです。Linux は基本的なフレームワークを提供してくれているため、このフレームワークに従ってドライバを書けば、カーネルはうまく受け入れて、要求通りに動作します。「工欲善其事、必先利其器」という言葉がありますが、このフレームワークを理解する前に、文字デバイスドライバ関連の概念やデータ構造を学ぶ時間を少し取る必要があります。

4.3 関連概念とデータ構造

Linux では、デバイスはデバイス番号で表されます。メインデバイス番号はデバイスのカテゴリを区別し、副デバイス番号は具体的なデバイスを識別します。cdev 構造体は、カーネルがデバイス番号を記録するために使用されます。デバイスを使用する際、通常はデバイスノードを開き、デバイスノードの inode 構造体、file 構造体を介して最終的に file_operations 構造体に到達し、file_operations 構造体からデバイスを操作する具体的な方法を得ます。

4.3.1 デバイス番号

文字デバイスのアクセスは、ファイルシステムの名前を介して行われます。これらの名前は特殊ファイル、デバイスファイル、あるいは単にファイルシステムツリーのノードと呼ばれます。Linux のルートディレクトリ下には/dev というフォルダがあり、これはデバイスのドライバプログラムを格納するために使用されます。ls -l コマンドを使用すると、システム内のすべてのデバイスをリスト形式で表示できます。各行が 1 つのデバイスを表し、各行の最初の文字はデバイスのタイプを示します。

以下の図のように、'c'は文字デバイスを、'b'はブロックデバイスを示します。例えば、autofs は文字デバイス c で、メインデバイス番号は 10、副デバイス番号は 235 です；loop0 はブロックデバイスで、メインデバイス番号は 7、副デバイス番号は 0 です。また、loop0-loop3 は同じメインデバイス番号を共有し、副デバイス番号は 0 から始まります。

株式会社日昇テクノロジー

```
debian@npi:~$ ls -l /dev/
total 0
crw-r--r-- 1 root    root     10, 235 Feb 14  2019 autofs
drwxr-xr-x 2 root    root          680 Jan  1  1970 block
drwxr-xr-x 3 root    root           60 Jan  1  1970 bus
drwxr-xr-x 2 root    root         2860 Sep  3 13:13 char
crw------- 1 root    root      5,   1 Sep  3 13:13 console
crw------- 1 root    root     10,  63 Feb 14  2019 cpu_dma_latency
drwxr-xr-x 7 root    root          140 Jan  1  1970 disk
drwxr-xr-x 3 root    root           80 Feb 14  2019 dri
crw-rw---- 1 root    video    29,   0 Feb 14  2019 fb0
crw-rw---- 1 root    video    29,   1 Feb 14  2019 fb1
lrwxrwxrwx 1 root    root           13 Jan  1  1970 fd -> /proc/self/fd
crw-rw-rw- 1 root    root      1,   7 Feb 14  2019 full
crw-rw-rw- 1 root    root     10, 229 Feb 14  2019 fuse
crw-rw---- 1 root    gpio    254,   0 Feb 14  2019 gpiochip0
crw-rw---- 1 root    gpio    254,   1 Feb 14  2019 gpiochip1
crw-rw---- 1 root    gpio    254,   2 Feb 14  2019 gpiochip2
crw-rw---- 1 root    gpio    254,   3 Feb 14  2019 gpiochip3
crw-rw---- 1 root    gpio    254,   4 Feb 14  2019 gpiochip4
crw-rw---- 1 root    gpio    254,   5 Feb 14  2019 gpiochip5
crw------- 1 root    root     10, 183 Feb 14  2019 hwrng
crw------- 1 root    root     89,   0 Feb 14  2019 i2c-0
crw------- 1 root    root     89,   1 Feb 14  2019 i2c-1
crw------- 1 root    root    250,   0 Feb 14  2019 iio:device0
crw------- 1 root    root    250,   1 Feb 14  2019 iio:device1
lrwxrwxrwx 1 root    root           12 Feb 14  2019 initctl -> /run/initctl
drwxr-xr-x 3 root    root          120 Feb 14  2019 input
crw-r--r-- 1 root    root      1,  11 Feb 14  2019 kmsg
lrwxrwxrwx 1 root    root           28 Feb 14  2019 log -> /run/systemd/journal/dev-log
crw-rw---- 1 root    disk     10, 237 Feb 14  2019 loop-control
brw-rw---- 1 root    disk      7,   0 Feb 14  2019 loop0
brw-rw---- 1 root    disk      7,   1 Feb 14  2019 loop1
brw-rw---- 1 root    disk      7,   2 Feb 14  2019 loop2
brw-rw---- 1 root    disk      7,   3 Feb 14  2019 loop3
```

一般的に、メインデバイス番号はデバイスのドライバプログラムを指し、副デバイス番号は具体的なデバイスを指します。上の例では、I2C-0、I2C-1 は異なるデバイスですが、同じドライバプログラムを共有しています。

4.3.1.1 カーネル内のデバイス番号の意味

カーネル内では、dev_t はデバイス番号を表すために使用されます。dev_t は 32 ビットの数値で、上位 12 ビットがメインデバイス番号、下位 20 ビットが副デバイス番号を表します。つまり、理論上のメインデバイス番号の範囲は 0-2^12、副デバイス番号の範囲は 0-2^20 です。実際には、カーネルソースコード内の__register_chrdev_region(...)関数で、major は 0-CHRDEV_MAJOR_MAX に制限されています。CHRDEV_MAJOR_MAX はマクロで、値は 512 です。kdev_t 内では、デバイス番号はシフト操作によって最終的に主/副デバイス番号を得ることができます。また、主/副デバイス番号もビット演算を通じて dev_t 型のデバイス番号に変換できます。具体的な実装は、以下のコード MAJOR(dev)、MINOR(dev)、MKDEV(ma,mi)を参照してください。

株式会社日昇テクノロジー

リスト 1: dev_t の定義（カーネルソースコード/include/types.h）

```
1 typedef u32 __kernel_dev_t;

2

3 typedef __kernel_dev_t dev_t;
```

リスト 2: デバイス番号関連マクロ（カーネルソースコード/include/linux/kdev_t.h）

```
1 #define MINORBITS 20

2 #define MINORMASK ((1U << MINORBITS) - 1)

3

4 #define MAJOR(dev) ((unsigned int) ((dev) >> MINORBITS))

5 #define MINOR(dev) ((unsigned int) ((dev) & MINORMASK))

6 #define MKDEV(ma,mi) (((ma) << MINORBITS) | (mi))
```

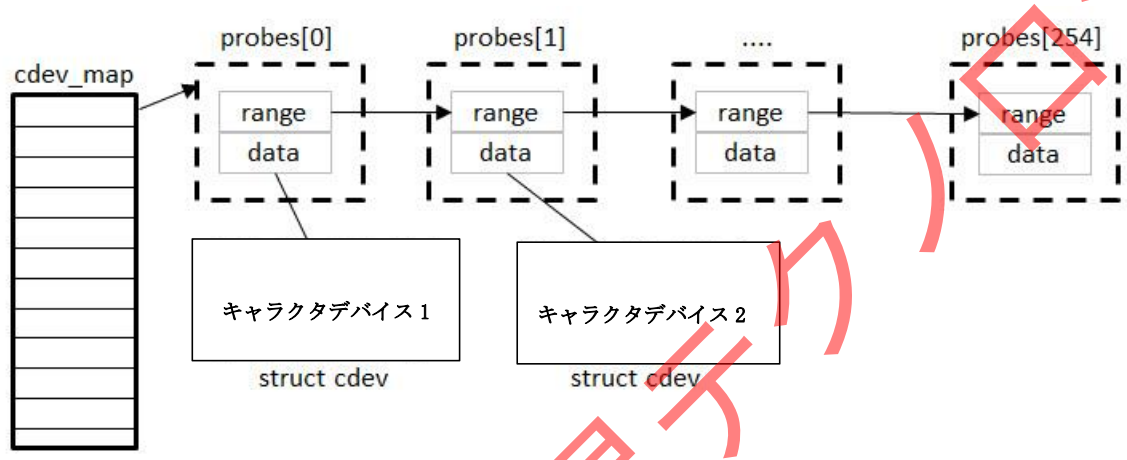
- 第 4-5 行：カーネルは、デバイスのデバイス番号からメジャーデバイス番号とマイナーデバイス番号を取得するための別の 2 つのマクロ定義 MAJOR と MINOR を提供しています。
- 第 6 行：マクロ定義 MKDEV は、メジャーデバイス番号とマイナーデバイス番号を 1 つのデバイス番号に合成するために使用されます。メジャーデバイスは、カーネルソースコードの Documentation/devices.txt ファイルを参照することで見つけることができ、マイナーデバイス番号は通常 0 から始まります。

4.3.1.2 cdev 構造体

カーネルは散布リスト（ハッシュテーブル）を使用してデバイス番号を記録します。ハッシュテーブルは配列とリンクリストの組み合わせで、配列の検索速度とリンクリストの追加・削除の効率の高さ、拡張性の良さなどの利点を活用します。

株式会社日昇テクノロジー

メインデバイス番号を cdev_map 番号として使用し、ハッシュ関数 f(major)=major%255 を使用して配列のインデックスを計算します（ハッシュ関数を使用するのは、リンクリストノードが各配列要素にできるだけ均等に分散され、検索効率を高めるためです）。メインデバイス番号が競合する場合は、副デバイス番号を比較値としてリンクリストノードをソートします。以下の図のように、カーネルは struct cdev 構造体を使用して文字デバイスを記述し、struct kobj_map 型のハッシュテーブル cdev_map を使用して、現在のシステム内のすべての文字デバイスを管理します。



リスト 3: cdev 構造体（カーネルソースコード/include/linux/cdev.h 内）

```
1 struct cdev {

2 struct kobject kobj;

3 struct module *owner;

4 const struct file_operations *ops;

5 struct list_head list;

6 dev_t dev;

7 unsigned int count;

8 } __randomize_layout;
```

- struct kobject kobj：組み込みのカーネルオブジェクトで、これによりデバイスが「Linux デバイスドライバモデル」に統一して管理されます（例えば、オブジェクトの参照カウント、電源管理、

株式会社日昇テクノロジー

ホットプラグ、ライフサイクル、ユーザーとの通信など）。

- struct module *owner：文字デバイスドライバプログラムが存在するカーネルモジュールオブジェクトへのポインター。

- const struct file_operations *ops：ファイル操作で、文字デバイスドライバ内で非常に重要なデータ構造です。アプリケーションがファイルシステム（VFS）を介してデバイスドライバプログラムに呼び出されるファイル操作クラス関数を実装する際、ops は橋渡しの役割を果たします。VFS とファイルシステムおよびデバイスファイル間のインターフェースは file_operations 構造体のメンバー関数で、この構造体にはファイルを開く、閉じる、読み書き、制御などの一連のメンバー関数が含まれます。

- struct list_head list：システム内の文字デバイスをリスト化するために使用されます（これはカーネルリストのリンク要素で、多くのカーネル構造体でこのタイプの構造を見ることができます）。

- dev_t dev：文字デバイスのデバイス番号で、メインデバイス番号と副デバイス番号から構成されます。

- unsigned int count：同一のメインデバイス番号に属する副デバイス番号の数で、デバイスドライバプログラムが制御する実際の同種デバイスの数を表します。

4.3.2 デバイスノード

デバイスノード（デバイスファイル）：Linux でデバイスノードは「mknod」コマンドを使用して作成されます。デバイスノードは実際にはファイルであり、Linux ではデバイスファイルと呼ばれます。Linux では、すべてのデバイスアクセスがファイルを通じて行われるため、通常のデータファイルをプログラムファイルとし、デバイスノードをデバイスファイルと呼びます。デバイスノードは /dev 下に作成され、カーネルとユーザーレイヤーをつなぐ中心的な役割を果たし、どのようなインタ

株式会社日昇テクノロジー

ーフェースのどの ID にデバイスが接続されているかを示します。これはハードディスクの inode と似ており、 Linux でのハードウェアデバイスの位置と情報を記録します。

Linux では、すべてのデバイスが/dev ディレクトリ下のファイルの形で存在し、ファイルを通じてアクセスされます。デバイスノードは Linux カーネルにおけるデバイスの抽象化であり、1 つのデバイスノードは 1 つのファイルです。アプリケーションは一連の標準化された呼び出しを使用してデバイスにアクセスし、これらの呼び出しは特定のドライバに依存しません。ドライバはこれらの標準呼び出しを実際のハードウェアの特定の操作にマッピングする責任があります。

4.3.3 データ構造

ドライバ開発プロセスでは、避けて通れない 3 つの重要なカーネルデータ構造があります。それらはファイル操作方法（file_operations）、ファイル記述構造体（struct file）、および inode 構造体です。ドライバプログラムのコードを読み始める前に、これら 3 つの構造体について理解することが重要です。

4.3.3.1 file_operations 構造体

file_operations は、システムコールとドライバプログラムを関連付ける鍵となるデータ構造です。この構造体の各メンバーは、あるシステムコールに対応しています。file_operation 内の対応する関数ポインタを読み取り、制御をその関数ポインタが指す関数に渡すことで、Linux デバイスドライバプログラムの仕事が完成します。

株式会社日昇テクノロジー

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    __poll_t (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    unsigned long mmap_supported_flags;
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned lo
    int (*check_flags)(int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **, void **);
    long (*fallocate)(struct file *file, int mode, loff_t offset,
              loff_t len);
    void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
    unsigned (*mmap_capabilities)(struct file *);
```

システム内部では、I/O デバイスへのアクセス操作は特定のエントリポイントを通じて行われ、これらのエントリポイントはデバイスドライバプログラムによって提供されます。通常、このグループのデバイスドライバプログラムインターフェースは、file_operations 構造体によってシステムに説明されます。これは、ebf_buster_linux/include/linux/fs.h 内で定義されています。伝統的に、file_operation 構造またはそのポインタは fops（またはその変種）と呼ばれます。構造内の各メンバーは、特定の操作を実装するドライバ内の関数を指さなければならず、サポートされていない操作には NULL を設定します。指定された関数ポインタが NULL の場合、カーネルの正確な挙動は関数ごとに異なります。

以下のコードでは、本章で使用される関数の一部のみをリストアップしています。

株式会社日昇テクノロジー

リスト 4: file_operations 構造体（カーネルソースコード/include/linux/fs.h 内）

```
1 struct file_operations {
2 struct module *owner;
3 loff_t (*llseek) (struct file *, loff_t, int);
4 ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
5 ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
6 long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
7 int (*open) (struct inode *, struct file *)
8 int (*release) (struct inode *, struct file *);
9 };
```

- llseek: ファイルの現在の読み書き位置を変更し、変更後の位置を返します。file パラメータには対応するファイルポインタが渡されます。この上で説明されているすべての関数にはこの形式のパラメータがあります。通常、ファイルの情報（ファイルタイプ、読み書き権限など）を読み取るために使用されます。loff_t パラメータはオフセットのサイズを指定します。int パラメータは新しい位置をファイルの特定の位置からオフセットするために使用されます。SEEK_SET はファイルの開始からオフセットすることを意味し SEEK_CUR は現在の位置から、SEEK_END はファイルの終わりからオフセットします。

- read: デバイスからデータを読み取り、成功した読み取りバイト数を返します。この関数ポインタが NULL に設定されている場合、システムコール read 関数を呼び出すと、「不正なパラメータ」のエラーが報告されます。この関数には 3 つのパラメータがあります。`file`型ポインタ変数、char __user*型のデータバッファ、__user は変数がユーザースペースのアドレス空間にあることを示すために使用されます。カーネルモジュールはこのデータを直接使用できず、copy_to_user 関数を使用して操作を行います。size_t 型変数は読み取るデータのサイズを指定します。

株式会社日昇テクノロジー

- write: デバイスにデータを書き込み、成功した書き込みバイト数を返します。write 関数のパラメータ使用法は read 関数と似ていますが、__user 修飾されたデータバッファにアクセスする場合は copy_from_user 関数を使用する必要があります。

- unlocked_ioctl: デバイスが関連する制御コマンドを実行する方法を提供します。これはアプリケーションの fcntl 関数および`ioctl`関数に対応します。kernel 3.0 では、struct file_operations の ioctl 関数ポインタが完全に削除されました。

- open: デバイスドライバーで最初に実行される関数で、通常はハードウェアの初期化に使用されます。このメンバが NULL に設定されている場合、このデバイスのオープン操作は常に成功します。

- release: file 構造体が解放されるときに呼び出されます。open 関数とは逆に、この関数は使用されなくなったリソースの解放に使用できます。

上記で read および write 関数に言及されたとき、copy_to_user および copy_from_user 関数を使用してデータアクセスを行う必要があります。これらの関数は成功すると 0 を返し、失敗するとコピーされなかったバイト数を返します。

リスト 5: copy_to_user および copy_from_user 関数（カーネルソースコード/include/asm-generic/uaccess.h 内）

```
1 static inline long copy_from_user(void *to, const void __user * from, unsigned long n)
2 static inline long copy_to_user(void __user *to, const void *from, unsigned long n)
```

関数パラメータと戻り値は以下の通りです：

- パラメータ

- to: 目的地アドレス、つまりデータが配置されるアドレスを指定します。

- from: ソースアドレス、つまりデータの出所を指定します。

- n: 書き込み/読み取りデータのバイト数を指定します。

株式会社日昇テクノロジー

- 戻り値

- 書き込み/読み取りデータのバイト数です。

4.3.3.2 file 構造体

カーネルは file 構造体を使用して、オープンされた各ファイルを表します。ファイルをオープンするたびに、カーネルはこの構造体を作成し、そのファイル上の操作関数をこの構造体のメンバ変数 f_op に渡します。ファイルのすべてのインスタンスが閉じられると、カーネルはこの構造体を解放します。

リスト 6: file 構造体（カーネルソースコード/include/fs.h 内）

```
1 struct file {
2 {.....}
3 const struct file_operations *f_op;
4 /* needed for tty driver, and maybe others */
5 void *private_data;
6 {.....}
7 };
```

- f_op: ファイル操作に関連する一連の関数ポインタを格納します。open、read、write などの関数が含まれます。

- private_data: このポインタ変数はデバイスドライバプログラムでのみ使用され、カーネルはこのメンバに対して操作を行いません。通常、ドライバプログラムではデバイスを記述する構造体を指すために使用されます。

4.3.3.3 inode 構造体

VFS inode には、ファイルのアクセス権限、所有者、グループ、サイズ、作成時間、アクセス時間、

株式会社日昇テクノロジー

最終変更時間などの情報が含まれます。これは Linux がファイルシステムを管理する基本的な単位であり、任意のサブディレクトリやファイルへのブリッジです。カーネルは inode 構造体を使用して、内部でファイルを表します。したがって、これは複数の file 構造体が同一のファイルの複数のファイル記述子を表すことができるが、これらすべての`file`構造体は唯一の inode 構造体を指す必要があることを意味します。inode 構造体にはファイルに関連する多くの情報が含まれていますが、ドライバコードに関しては、以下の 2 つのフィールドのみが関心の対象です：

リスト 7: inode 構造体（カーネルソースコード/include/linux/fs.h 内）

```
1 struct inode {
2
3 dev_t i_rdev;
4 {.....}
5 union {
6 struct pipe_inode_info *i_pipe; /* Linux カーネルパイプ */
7 struct block_device *i_bdev; /* これがブロックデバイスの場合、設定して使用 */
8 struct cdev *i_cdev; /* これがキャラクタデバイスの場合、設定して使用 */
9 char *i_link;
10 unsigned i_dir_seq;
11 };
12 {.....}
13 };
```

- dev_t i_rdev: デバイスファイルのノードを表します。このフィールドにはデバイス番号が含まれています。

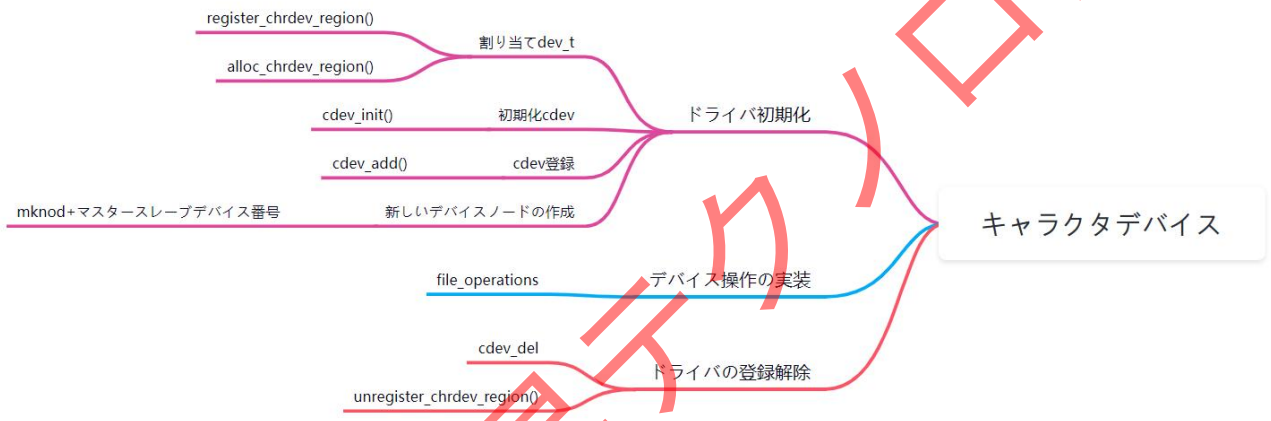
- struct cdev *i_cdev: struct cdev はカーネルの内部構造で、文字デバイスを表すために使用されま

株式会社日昇テクノロジー

す。inode ノードが文字デバイスファイルを指す場合、このフィールドは inode 構造へのポインターです。

4.4 字符デバイスドライバフレームワーク

文字デバイスドライバフレームワークについて話してきましたが、一体全体文字ファイルプログラムフレームワークとは何でしょうか？以下の思考マップからカーネルソースコードを解釈することができます。



文字デバイスを作成する際、まず必要とされるのはデバイス番号を得ることです。デバイス番号の割り当て方法には静的割り当てと動的割り当てがあります；デバイスのユニーク ID を得たら、file_operation を実装し、cdev に保存する必要があり、cdev の初期化を行います；次に、行った作業をカーネルに伝え、cdev_add()を使用して cdev を登録する必要があります；最後に、後で file_operation インターフェースを呼び出せるようにデバイスノードを作成する必要があります。

デバイスを登録解除する際は、カーネル内の cdev を解放し、申請したデバイス番号を返却し、作成したデバイスノードを削除する必要があります。

デバイス操作の実装では、open 関数が実際に何を行っているかを見ることができます。

株式会社日昇テクノロジー

4.4.1 ドライバの初期化と登録解除

4.4.1.1 デバイス番号の申請と返却

Linux カーネルは、以下のようにして文字デバイスを定義する 2 つの方法を提供しています。

リスト 8: 文字デバイスの定義

```
1 //第一の方法
2 static struct cdev chrdev;
3 //第二の方法
4 struct cdev *cdev_alloc(void);
```

第一の方法は、我々がよく見る変数の定義です；第二の方法は、カーネルが提供する動的割り当て方法であり、この関数を呼び出すと、文字デバイスを記述する struct cdev 型のポインタが返されます。

カーネルからある文字デバイスを削除する場合は、cdev_del 関数を呼び出す必要があります。以下に示します。

リスト 9: cdev_del 関数

```
1 void cdev_del(struct cdev *p)
```

関数のパラメータと戻り値は以下の通りです：

パラメータ：

- p：この関数は文字デバイス構造体のアドレスをパラメータとして渡す必要があり、それによりカーネルからその文字デバイスを削除することができます。

戻り値：なし

register_chrdev_region 関数

register_chrdev_region 関数は、1 つまたは複数のデバイス番号を文字デバイスに静的に申請するために使用されます。関数のプロトタイプは以下の通りです。

株式会社日昇テクノロジー

リスト 10: register_chrdev_region 関数

```
1 int register_chrdev_region(dev_t from, unsigned count, const char *name)
```

関数のパラメータと戻り値は以下の通りです：

パラメータ：

• from：dev_t 型の変数で、文字デバイスの開始デバイス番号を指定します。もし登録しようとするデバイス番号が他のデバイスによって既に登録されていた場合、登録に失敗します。

• count：申請するデバイス番号の個数を指定します。count の値はあまり大きくしてはいけません。そうでないと、次のメインデバイス番号と重なってしまいます。

• name：そのデバイスの名称を指定します。/proc/devices でこのデバイスを見ることができます。

戻り値：0 を返すと申請成功、失敗するとエラーコードを返します

alloc_chrdev_region 関数

register_chrdev_region 関数を使用する際は、カーネルソースコードの Documentation/devices.txt ファイルを調べる必要があり、これは非常に不便です。したがって、カーネルは動的にデバイス番号を割り当てる別の方法を提供しています：alloc_chrdev_region。alloc_chrdev_region 関数を呼び出すと、カーネルは自動的に未使用のメインデバイス番号を割り当ててくれます。カーネルが割り当てたメインデバイス番号は、“cat /proc/devices”コマンドで確認することができます。

リスト 11: alloc_chrdev_region 関数のプロトタイプ

```
1 int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count,const char *name)
```

関数のパラメータと戻り値は以下の通りです：

パラメータ：

• dev：割り当てられたデバイス番号の開始値を格納する dev_t 型データへのポインタ変数；

• baseminor：サブデバイス番号の開始値、通常は 0 に設定されます；

株式会社日昇テクノロジー

• count、name：register_chrdev_region 関数と同様に、割り当てるデバイス番号の数とデバイスの名称を指定します。

戻り値：0 が返された場合は申請成功を意味し、失敗した場合はエラーコードが返されます。

unregister_chrdev_region 関数

文字デバイスを削除する際には、割り当てられたデバイス番号をカーネルに返却する必要があります。register_chrdev_region 関数および alloc_chrdev_region 関数で割り当てられたデバイス番号については、unregister_chrdev_region 関数を使用してこれを実現することができます。

リスト 12: unregister_chrdev_region 関数（カーネルソースコード/fs/char_dev.c）

```
1 void unregister_chrdev_region(dev_t from, unsigned count)
```

関数のパラメータと戻り値は以下の通りです：

パラメータ：

• from：注销する文字デバイスのデバイス番号の開始値を指定します。通常、定義された dev_t 変数を実パラメータとして使用します。

• count：注销する文字デバイスのデバイス番号の数を指定します。この値は申請関数の count 値と同じであるべきで、通常はマクロで管理されます。

戻り値：なし

register_chrdev 関数

上述の 2 つの方法に加え、カーネルは register_chrdev 関数も提供しており、デバイス番号の割り当てに使用されます。この関数はインライン関数であり、静的にも動的にもデバイス番号を申請でき、メインデバイス番号を返します。関数のプロトタイプは以下の通りです。

リスト 13: register_chrdev 関数プロトタイプ（カーネルソースコード/include/linux/fs.h ファイル）

株式会社日昇テクノロジー

```
1 static inline int register_chrdev(unsigned int major, const char *name,
2 const struct file_operations *fops)
3 {
4 return __register_chrdev(major, 0, 256, name, fops);
5 }
```

関数のパラメータと戻り値は以下の通りです：

パラメータ：

- major：申請する文字デバイスのメインデバイス番号を指定します。0 に設定すると、カーネルは使用されていないメインデバイス番号を自動的に割り当てます。
- name：文字デバイスの名称を指定します。
- fops：そのデバイスを操作するための関数インターフェースポインタ。

戻り値：メインデバイス番号

上記のコードからわかるように、register_chrdev 関数を使用してカーネルにデバイス番号を申請すると、同一の文字デバイス（つまり、メインデバイス番号が同じ）に対して 256 個が申請されることになり、通常、これほど多くのデバイスを必要とすることはありません。これは大きなリソースの浪費を引き起こします。

unregister_chrdev 関数：register 関数を使用して申請したデバイス番号は、unregister_chrdev 関数を使用して登録解除する必要があります。

リスト 14: unregister_chrdev 関数（カーネルソースコード/include/linux/fs.h）

```
1 static inline void unregister_chrdev(unsigned int major, const char *name)
2 {
3 __unregister_chrdev(major, 0, 256, name);
4 }
```

株式会社日昇テクノロジー

関数のパラメータと戻り値は以下の通りです：

パラメータ：

- major：解放する文字デバイスのメインデバイス番号を指定します。通常は register_chrdev 関数の戻り値を実パラメータとして使用します。
- name：解放する文字デバイスの名称を指定します。

戻り値：なし

4.4.1.2 cdev の初期化

前述の通り、文字デバイスを作成する上で最も重要なことは、file_operations 構造体内の関数を実装することです。実装後、この構造体をどのようにして文字デバイス構造体と関連付けるか？カーネルは cdev_init 関数を提供し、このプロセスを実現します。

リスト 15: cdev_init 関数（カーネルソースコード/fs/char_dev.c）

```
1 void cdev_init(struct cdev *cdev, const struct file_operations *fops)
```

関数のパラメータと戻り値は以下の通りです：

パラメータ：

- cdev：struct cdev 型のポインタ変数で、関連付ける必要のある文字デバイス構造体を指します；
- fops：file_operations 型の構造体ポインタ変数で、通常、このデバイスを操作する構造体 file_operations を実パラメータとして渡します。

戻り値：なし

株式会社日昇テクノロジー


```
static struct file_operations chr_dev_fops =
{
    .owner = THIS_MODULE,
    .open = chr_dev_open,
    .release = chr_dev_release,
    .write = chr_dev_write,
    .read = chr_dev_read,
};

static int chr_dev_open(struct inode *inode, struct file *filp) ···

static int chr_dev_release(struct inode *inode, struct file *filp) ···

static ssize_t chr_dev_write(struct file *filp, const char __user * buf, size_t count, loff_t *ppos) ···

static ssize_t chr_dev_read(struct file *filp, char __user * buf, size_t count, loff_t *ppos) ···

static int __init chrdev_init(void)
{
    int ret = 0;
    printk("chrdev init\n");
    ret = alloc_chrdev_region(&devno, 0, DEV_CNT, DEV_NAME);
    if(ret < 0){
        printk("fail to alloc devno\n");
        goto alloc_err;
    }
    cdev_init(&chr_dev, &chr_dev_fops);
    ret = cdev_add(&chr_dev, devno, DEV_CNT);
    if(ret < 0)
    {
        printk("fail to add cdev\n");
        goto add_err;
    }
    return 0;
add_err:
    unregister_chrdev_region(devno, DEV_CNT);
alloc_err:
    return ret;
}
```

4.4.2 デバイスの登録と登録解除

cdev_add 関数は、新しい文字デバイスをカーネルの cdev_map ハッシュテーブルに追加するために使用されます。以下に示します。

リスト 16: cdev_add 関数（カーネルソースコード/fs/char_dev.c）

```
1 int cdev_add(struct cdev *p, dev_t dev, unsigned count)
```

関数のパラメータと戻り値は以下の通りです：

パラメータ：

- p：struct cdev 型のポインタで、追加する必要のある文字デバイスを指します；
- dev：dev_t 型の変数で、デバイスの開始番号を指します；
- count：登録するデバイスの数を指します。

戻り値：エラーコード

株式会社日昇テクノロジー

cdev をシステムから削除すると、cdev デバイスはもはや開けられなくなりますが、すでに開いている cdev は変わらずに残り、cdev_del が返された後でもその FOP は呼び出し可能です。

リスト 17: cdev_del 関数（カーネルソースコード/fs/char_dev.c）

```
1 void cdev_del(struct cdev *p)
```

関数のパラメータと戻り値は以下の通りです：

パラメータ：

- p：struct cdev 型のポインタで、削除する必要のある文字デバイスを指します；

戻り値：なし

4.4.3 デバイスノードの作成と破棄

ファイルシステムにデバイスを登録し、それを作成する

リスト 18: device_create 関数（カーネルソースコード/drivers/base/core.c）

```
1 struct device *device_create(struct class *class, struct device *parent,
2                 dev_t devt, void *drvdata, const char *fmt, ...)
```

関数のパラメータと戻り値は以下の通りです：

パラメータ：

- class：このデバイスが登録されるべき struct クラスへのポインタ；
- parent：この新しいデバイスの親構造体デバイス（存在する場合）へのポインタ；
- devt：追加される文字デバイスの開発番号；
- drvdata：デバイスに追加され、コールバックのためのデータ；
- fmt：デバイス名の入力。

戻り値：成功時は struct device 構造体のポインタ、エラー時は ERR_PTR()を返します。

device_create 関数で作成したデバイスを削除する

株式会社日昇テクノロジー

リスト 19: device_destroy 関数（カーネルソースコード/drivers/base/core.c）

```
1 void device_destroy(struct class *class, dev_t devt)
```

関数のパラメータと戻り値は以下の通りです：

パラメータ：

- class：このデバイスが登録された struct クラスへのポインタ；
- devt：以前に登録されたデバイスの開発番号；

戻り値：なし

コードを使用してデバイスノードを作成する以外に、mknod コマンドを使用してデバイスノードを作成することもできます。

使用方法：mknod デバイス名 デバイスタイプ メインデバイス番号 副デバイス番号

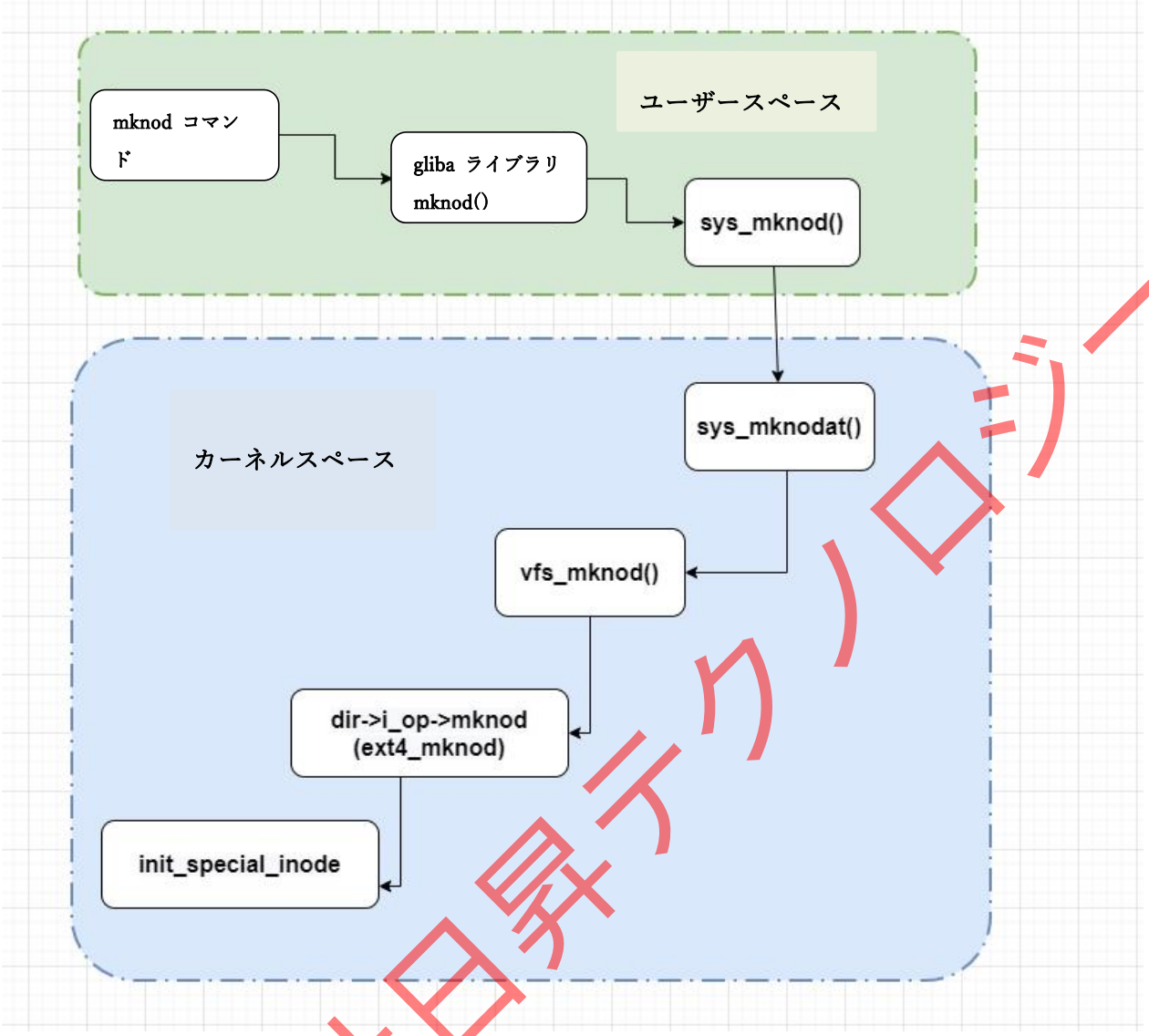
タイプが”p”の場合はメインデバイス番号と副デバイス番号を指定する必要はありませんが、それ以外の場合は必須です。メインデバイス番号と副デバイス番号が”0x”または”0X”で始まる場合は 16 進数として解釈され、”0”で始まる場合は 8 進数、その他の場合は 10 進数として解釈されます。使用可能なタイプには以下が含まれます：

- b (バッファー付き)ブロック特殊ファイルの作成
- c, u (バッファーなし)文字特殊ファイルの作成
- p FIFO（先入れ先出し）特殊ファイルの作成

例：mknod /dev/test c 2 0

メインデバイス番号が 2 で副デバイス番号が 0 の文字デバイス/dev/test を作成します。

株式会社日昇テクノロジー



上記のコマンドを使用して文字デバイスファイルを作成した場合、実際にはデバイスノード inode 構造体が作成され、そのデバイスのデバイス番号がメンバー i_rdev に記録され、メンバー f_op ポインタが def_chr_fops 構造体を指します。

これが mknod が担当する作業内容であり、具体的なコードは以下に示されます。

リスト 20: mknod の呼び出し関係（カーネルソースコード/mm/shmem.c）

```
1 static struct inode *shmem_get_inode(struct super_block *sb, const struct inode *dir,
2 umode_t mode, dev_t dev, unsigned long flags)
3 {
```

株式会社日昇テクノロジー

```
4 inode = new_inode(sb);
5 if (inode) {
6 ......
7 switch (mode & S_IFMT) {
8 default:
9 inode->i_op = &shmem_special_inode_operations;
10 init_special_inode(inode, mode, dev);
11 break;
12 ......
13 }
14 } else
15 shmem_free_inode(sb);
16 return inode;
17 }
```

• 10 行目：mknod コマンドは最終的に init_special_inode 関数を実行します

リスト 21: init_special_inode 関数（カーネルソースコード/fs/inode.c）

```
1 void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
2 {
3 inode->i_mode = mode;
4 if (S_ISCHR(mode)) {
5 inode->i_fop = &def_chr_fops;
6 inode->i_rdev = rdev;
```

株式会社日昇テクノロジー

```
7 } else if (S_ISBLK(mode)) {
8 inode->i_fop = &def_blk_fops;
9 inode->i_rdev = rdev;
10 } else if (S_ISFIFO(mode))
11 inode->i_fop = &pipefifo_fops;
12 else if (S_ISSOCK(mode))
13 ; /* leave it no_open_fops */
14 else
15 printk(KERN_DEBUG "init_special_inode: bogus i_mode (%o) for"
16 " inode %s:%lu¥n", mode, inode->i_sb->s_id,
17 inode->i_ino);
18 }
```

- 4-17 行目：ファイルの inode タイプを判断し、文字デバイスタイプであれば、そのファイルの操作インターフェイスとして def_chr_fops を設定し、デバイス番号を inode->i_rdev に記録します。

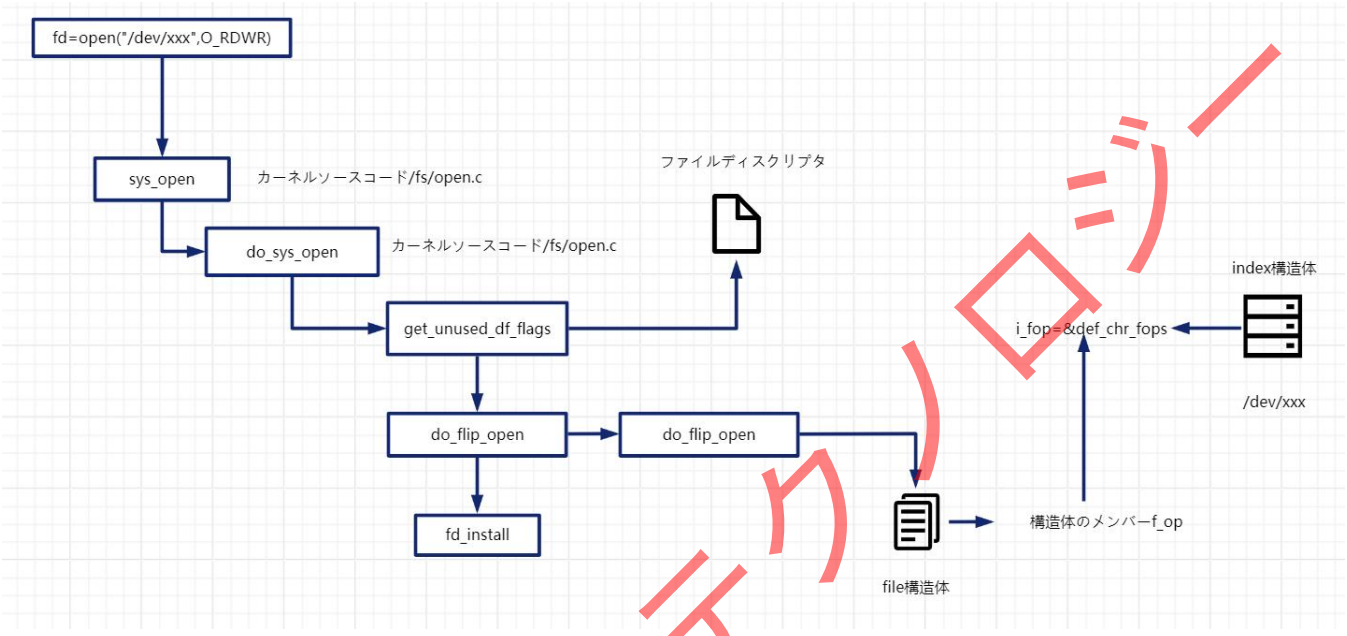
inode 上の file_operation は自作の file_operation ではなく、文字デバイス共通の def_chr_fops です。自作の file_operation などは、アプリケーションが open 関数を呼び出した後に、ファイルにバインドされます。次に、open 関数が実際に何を行ったのかを見てみましょう。

4.5 open 関数の役割

デバイスを使用する前に、通常は open 関数を呼び出します。この関数は、デバイス専用データの初期化、関連リソースの申請、デバイスの初期化などの作業に使用されます。単純なデバイスの場合、open 関数では具体的な作業を行わなくてもよいかもしれません。アプリケーションレイヤーからシステムコール open を使ってデバイスを開くと、正常に開ければそのデバイスのファイルディスクリプ

株式会社日昇テクノロジー

タを得ることができます。その後、このディスクリプタを通じてデバイスに対する read や write などの操作を行うことができます。open 関数は具体的にどのような作業を行うのでしょうか？以下に open 関数の実行プロセスを示します。



ユーザースペースから open()システムコール関数を使って文字デバイスを開く場合（int fd = open("dev/xxx", O_RDWR)）の大まかなプロセスは以下の通りです：

- 仮想ファイルシステム VFS で、文字デバイスに対応する struct inode ノードを探します
- 散布表 cdev_map を走査し、inode ノード内の cdev_t デバイス番号に基づいて cdev オブジェクトを見つけます
- struct file オブジェクトを作成します（システムは開かれた複数のデバイスを管理するために配列を使用し、各ファイルディスクリプタは配列のインデックスとしてデバイスオブジェクトを識別します）
- struct file オブジェクトを初期化し、struct file オブジェクトの file_operations メンバを struct cdev オブジェクトの file_operations メンバに指向させます（file->fops = cdev->fops）
- file->fops->open 関数をコールバックします

カーネル内での open 関数に対応するのは sys_open 関数であり、sys_open 関数は do_sys_open 関

株式会社日昇テクノロジー

数を呼び出します。do_sys_open 関数では、最初に get_unused_fd_flags 関数を呼び出して未使用のファイルディスクリプタ fd を取得します。このファイルディスクリプタは、最終的に open 関数を通じて得られる値です。直後に、do_filp_open 関数が呼び出され、この関数は get_empty_filp 関数を通じて新しい file 構造体を取得します。その後のコードでは、ファイルパスの解析、ファイルの inode ノードの検索など、多くの複雑な作業を行います。最終的に do_dentry_open 関数に到達します。以下に示します。

リスト 22: do_dentry_open 関数（ebf-busrerlinux/fs/open.c に位置）

```
1 static int do_dentry_open(struct file *f,struct inode *inode,int (*open)(struct inode *, struct file
*),const struct cred *cred)
2 {
3 //……
4 f->f_op = fops_get(inode->i_fop);
5 //……
6 if (!open)
7 open = f->f_op->open;
8 if (open) {
9 error = open(inode, f);
10 if (error)
11 goto cleanup_all;
12 }
13 //……
14 }
```

• 4 行目：fops_get 関数を使用して、ファイルノード inode のメンバ変数 i_fop を取得します。

株式会社日昇テクノロジー

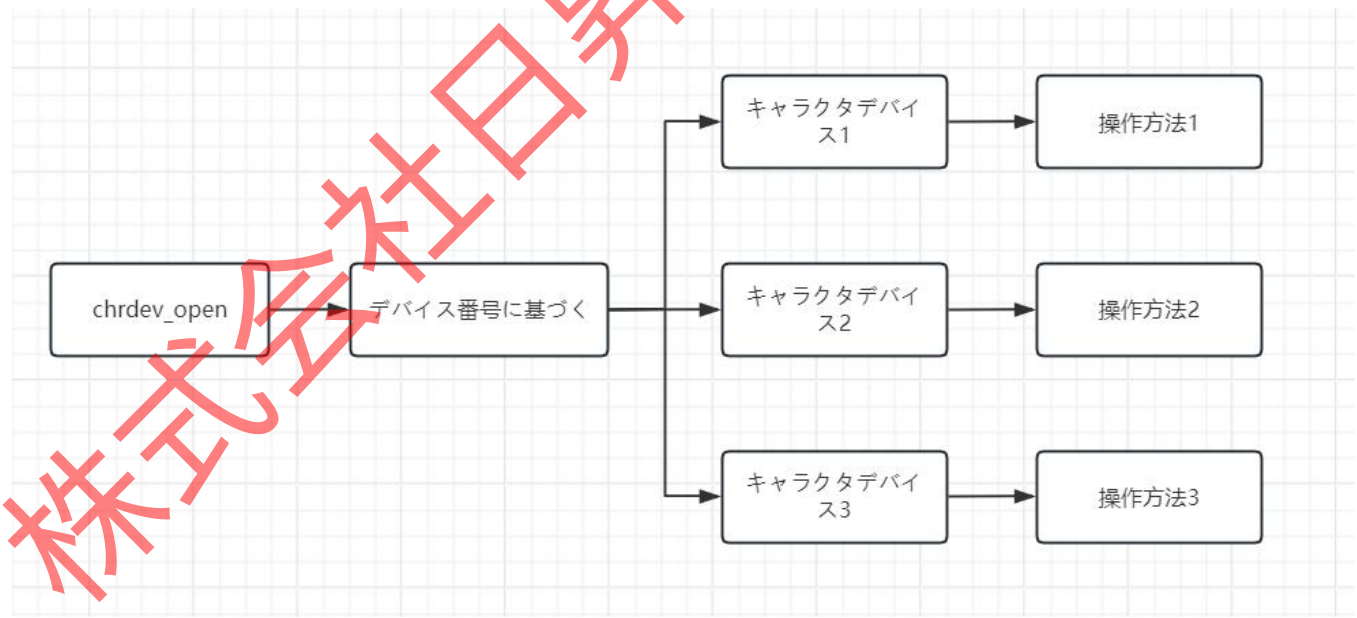
上記の例で mknod を使って文字デバイスファイルを作成した際、def_chr_fops 構造体をそのデバイスファイル inode の i_fop メンバに割り当てました。

• 7 行目：ここに到達すると、新しく作成された file 構造体のメンバ f_op は def_chr_fops を指しています。

リスト 23: def_chr_fops 構造体（カーネルソースコード/fs/char_dev.c）

```
1 const struct file_operations def_chr_fops = {
2 .open = chrdev_open,
3 .llseek = noop_llseek,
4 };
```

最終的に、def_chr_fops 内の open 関数、つまり chrdev_open 関数が実行されます。これは文字デバイスの汎用初期化関数と理解でき、文字デバイスのデバイス番号に基づいて該当する文字デバイスを見つけ、そのデバイスを操作する方法を得ることができます。コードの実装は以下の通りです。



株式会社日昇テクノロジー

リスト 24: chrdev_open 関数（カーネルソースコード/fs/char_dev.c）

```
1 static int chrdev_open(struct inode *inode, struct file *filp)
2 {
3 const struct file_operations *fops;
4 struct cdev *p;
5 struct cdev *new = NULL;
6 int ret = 0;
7 spin_lock(&cdev_lock);
8 p = inode->i_cdev;
9 if (!p) {
10 struct kobject *kobj;
11 int idx;
12 spin_unlock(&cdev_lock);
13 kobj = kobj_lookup(cdev_map, inode->i_rdev, &idx);
14 if (!kobj)
15 return -ENXIO;
16 new = container_of(kobj, struct cdev, kobj);
17 spin_lock(&cdev_lock);
18 /* Check i_cdev again in case somebody beat us to it whilewe dropped the lock.*/
19 p = inode->i_cdev;
20 if (!p) {
21 inode->i_cdev = p = new;
```

株式会社日昇テクノロジー

```
22 list_add(&inode->i_devices, &p->list);

23 new = NULL;

24 } else if (!cdev_get(p))

25 ret = -ENXIO;

26 } else if (!cdev_get(p))

27 ret = -ENXIO;

28 spin_unlock(&cdev_lock);

29 cdev_put(new);

30 if (ret)

31 return ret;

32

33 ret = -ENXIO;

34 fops = fops_get(p->ops);

35 if (!fops)

36 goto out_cdev_put;

37

38 replace_fops(filp, fops);

39 if (filp->f_op->open) {

40 ret = filp->f_op->open(inode, filp);

41 if (ret)

42 goto out_cdev_put;

43 }
```

株式会社日昇テクノロジー

```
44
45 return 0;
46
47 out_cdev_put:
48 cdev_put(p);
49 return ret;
50 }
```

Linux カーネルでは、文字デバイスを記述するために cdev 構造体が使用されます。

- 8 行目：inode->i_rdev には文字デバイスのデバイス番号が保存されています、
- 13 行目：kobj_lookup 関数を使って、そのデバイスファイル cdev 構造体の kobj メンバを見つけることができます、
- 16 行目：そして、container_of 関数を使用することで、その文字デバイスに対応する cdev 構造体を得ることができます。container_of の役割は、構造体変数のあるメンバのアドレスからその構造体変数の先頭アドレスを見つけ出すことです。同時に、cdev 構造体をファイルノード inode の i_cdev に記録し、次回そのファイルを開く際に利用できるようにします。
- 38-43 行目：chrdev_open 関数は最終的に、そのファイル構造体 file のメンバ f_op を cdev に対応する ops メンバに置き換え、ops 構造体内の open 関数を実行します。

最後に、上述の fd_install 関数を呼び出し、ファイルディスクリプタとファイル構造体 file の関連付けを完了します。その後、このファイルディスクリプタ fd を使って read や write 関数を呼び出すと、実際には file 構造体に対応する関数が呼び出され、それは実際には cdev 構造体内の ops 構造体にある関連関数を呼び出すことになります。

要約すると、open 関数を使用してデバイスファイルを開くとき、そのデバイスのデバイス番号に基づいて対応するデバイス構造体を見つけ、そのデバイスを操作する方法を得ることができます。つま

株式会社日昇テクノロジー

り、新しいデバイスを追加する場合、デバイス番号、デバイス構造体、およびそのデバイスを操作する方法（file_operations 構造体）を提供する必要があります。

4.6 文字デバイスドライバプログラムの実験

4.6.1 ハードウェアの紹介

この実験では、Lubancat_RK のボードを使用します。

4.6.2 実験コードの説明

この章のサンプルコードのディレクトリは以下の通りです：linux_driver/EmbedCharDev/CharDev/

これまでの全ての知識点を組み合わせて、まず、文字デバイスドライバプログラムはカーネルモジュールの形で存在します。システムに新しい文字デバイスを登録するには、以下のものが必要です：文字デバイス構造体 cdev、デバイス番号 devno、そして最も重要な操作方法構造体 file_operations です。また、この実験のデバイスファイルは手で mknod を使用して作成する必要があります。

以下、自身の文字デバイスドライバプログラムを作成していきます。

4.6.2.1 カーネルモジュールのフレームワーク

デバイスプログラムがカーネルモジュールとして存在するのであれば、まず基本的なカーネルフレームワークを書く必要があります。以下に示します。

リスト 25: カーネルモジュールのロード関数

（../linux_driver/EmbedCharDev/CharDev/chrdev.c に位置）

```
1 #define DEV_NAME "EmbedCharDev"
2 #define DEV_CNT (1)
3 #define BUFF_SIZE 128
4 //文字デバイスのデバイス番号を定義
```

株式会社日昇テクノロジー

```
5 static dev_t devno;
6 //文字デバイス構造体 chr_dev を定義
7 static struct cdev chr_dev;
8 static int __init chrdev_init(void)
9 {
10 int ret = 0;
11 printk("chrdev init¥n");
12 //第一ステップ
13 //動的割り当ての方法を使用し、デバイス番号を取得、副デバイス番号は 0、
14 //デバイス名は EmbedCharDev とし、cat /proc/devices で確認可能
15 //DEV_CNT は 1、現在は 1 つのデバイス番号のみを申請
16 ret = alloc_chrdev_region(&devno, 0, DEV_CNT, DEV_NAME);
17 if (ret < 0) {
18 printk("デバイス番号の割り当てに失敗¥n");
19 goto alloc_err;
20 }
21 //第二ステップ
22 //文字デバイス構造体 cdev とファイル操作構造体 file_operations を関連付け
23 cdev_init(&chr_dev, &chr_dev_fops);
24 //第三ステップ
25 //デバイスを cdev_map 散布表に追加
26 ret = cdev_add(&chr_dev, devno, DEV_CNT);
```

株式会社日昇テクノロジー

```
27 if (ret < 0) {
28 printk("cdev の追加に失敗¥n");
29 goto add_err;
30 }
31 return 0;
32
33 add_err:
34 //デバイスの追加に失敗した場合、デバイス番号を解除する必要があります
35 unregister_chrdev_region(devno, DEV_CNT);
36 alloc_err:
37 return ret;
38 }
39 module_init(chrdev_init);
```

• 16 行目：動的割り当て(alloc_chrdev_region)を使用してデバイス番号を取得し、デバイスの名前を"EmbedCharDev"と指定し、副デバイス番号は 0 として、1 つのデバイス番号のみを申請します。

• 19 行目：C 言語の goto 文を使用して、取得に失敗した場合は対応するエラーコードを直接返します。デバイス番号が正常に取得できた後、文字デバイス構造体とファイルの操作方法が必要になります。

• 23 行目：上記のコードでは、文字デバイス構造体 chr_dev を変数定義の方法で定義し、cdev_init 関数を呼び出して chr_dev 構造体とファイル操作構造体を関連付けています。この構造体の具体的な実装は次のセクションで説明します。

• 26 行目：最後に、cdev_add 関数を呼び出して文字デバイスを文字デバイス管理リスト

株式会社日昇テクノロジー

cdev_map に追加するだけです。

• 29 行目：ここでも goto 文を使用し、デバイスの追加に失敗した場合は申請したデバイス番号を解除する必要があります。良い習慣を身につけ、「使用しないリソースを占有しない」ことが重要です。

モジュールのアンロード関数は比較的シンプルで、デバイス番号の解除と文字デバイスの削除のみを行います。以下に示します。

リスト 26: カーネルモジュールのアンロード関数

(../linux_driver/EmbedCharDev/CharDev/chrdev.c に位置)

```
1 static void __exit chrdev_exit(void)
2 {
3 printk("chrdev exit¥n");
4 unregister_chrdev_region(devno, DEV_CNT);
5 cdev_del(&chr_dev);
6 }
7 module_exit(chrdev_exit);
```

4.6.2.2 ファイル操作方法の実装

以下、文字デバイスで最も重要な部分であるファイル操作方法構造体 file_operations の実装を始めます。以下に示します。

株式会社日昇テクノロジー

リスト 27: file_operations 構造体（../linux_driver/EmbedCharDev/CharDev/chrdev.c に位置）

```
1 #define BUFF_SIZE 128
2 //データバッファ
3 static char vbuf[BUFF_SIZE];
4 static struct file_operations chr_dev_fops = {
5 .owner = THIS_MODULE,
6 .open = chr_dev_open,
7 .release = chr_dev_release,
8 .write = chr_dev_write,
9 .read = chr_dev_read,
10 };
```

この文字デバイスは仮想のデバイスであり、ハードウェアとは関連がないため、open 関数と release 関数は 0 を直接返すだけでよいです。write 関数と read 関数の実装に重点を置きます。

リスト 28: chr_dev_open 関数と chr_dev_release 関数
（../linux_driver/EmbedCharDev/CharDev/chrdev.c に位置）

```
1 static int chr_dev_open(struct inode *inode, struct file *filp)
2 {
3 printk("¥nopen¥n");
4 return 0;
5 }
6 static int chr_dev_release(struct inode *inode, struct file *filp)
7 {
```

株式会社日昇テクノロジー

```
8 printk("¥nrelease¥n");

9 return 0;

10 }
```

open 関数と release 関数で関連するデバッグ情報を出力します、上記のコードのように。

リスト 29: chr_dev_write 関数（../linux_driver/EmbedCharDev/CharDev/chrdev.c に位置）

```
1 static ssize_t chr_dev_write(struct file *filp, const char __user * buf, size_t count, loff_t *ppos)

2 {

3 unsigned long p = *ppos;

4 int ret;

5 int tmp = count ;

6 if (p > BUFF_SIZE)

7 return 0;

8 if (tmp > BUFF_SIZE - p)

9 tmp = BUFF_SIZE - p;

10 ret = copy_from_user(vbuf, buf, tmp);

11 *ppos += tmp;

12 return tmp;

13 }
```

アプリケーションが write 関数を呼び出すと、最終的には chr_dev_write 関数が呼び出されます。

・3 行目：変数 p は現在のファイルの読み書き位置を記録します。

・6-9 行目：データバッファのサイズ（128 バイト）を超える場合は、直接 0 を返します。また、読み書きするデータの個数がデータバッファの残りの内容を超える場合は、残りの内容のみを読

株式会社日昇テクノロジー

み取ります。

・10-11 行目：copy_from_user を使用して、ユーザースペースからデータバッファに tmp バイトのデータをコピーし、ファイルの読み書き位置を同じバイト数だけオフセットします。

リスト 30: chr_dev_read 関数（../linux_driver/EmbedCharDev/CharDev/chrdev.c に位置）

```
1 static ssize_t chr_dev_read(struct file *filp, char __user * buf, size_t    count, loff_t *ppos)
2 {
3 unsigned long p = *ppos;
4 int ret;
5 int tmp = count ;
6 if (p >= BUFF_SIZE)
7 return 0;
8 if (tmp > BUFF_SIZE - p)
9 tmp = BUFF_SIZE - p;
10 ret = copy_to_user(buf, vbuf+p, tmp);
11 *ppos +=tmp;
12 return tmp;
13 }
```

同様に、アプリケーションが read 関数を呼び出すと、chr_dev_read 関数が実行されます。この関数の実装は chr_dev_write 関数と似ていますが、データバッファから tmp バイトのデータをユーザースペースに copy_to_user を使用してコピーします。

4.6.2.3 簡単なテストプログラム

以下、文字デバイスを読み書きするためのアプリケーションを作成します。以下に示します。

株式会社日昇テクノロジー

リスト 31: main.c 関数（../linux_driver/EmbedCharDev/CharDev/main.c に位置）

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <fcntl.h>
4 #include <string.h>
5 char *wbuf = "Hello World¥n";
6 char rbuf[128];
7 int main(void)
8 {
9 printf("EmbedCharDev test¥n");
10 //ファイルを開く
11 int fd = open("/dev/chrdev", O_RDWR);
12 //データを書き込む
13 write(fd, wbuf, strlen(wbuf));
14 //書き込み完了後、ファイルを閉じる
15 close(fd);
16 //ファイルを開く
17 fd = open("/dev/chrdev", O_RDWR);
18 //ファイル内容を読み取る
19 read(fd, rbuf, 128);
20 //読み取った内容を出力
21 printf("The content : %s", rbuf);
```

株式会社日昇テクノロジー

```
22 //読み取り完了後、ファイルを閉じる
23 close(fd);
24 return 0;
25 }
```

・11 行目：読み書き可能な方法で作成した文字デバイスドライバを開きます

・12-15 行目：データを書き込んでから閉じます

・17-21 行目：デバイスを再度開き、データを読み取ります

4.6.3 実験の準備

カーネルモジュールのソースコードを取得し、付属のコード/linux_driver/embedCharDev/charDev をカーネルコードと同じレベルのディレクトリに解凍します。

4.6.3.1 makefile の変更説明

リスト 32: makefile（../linux_driver/EmbedCharDev/CharDev/Makefile に位置）

```
1 KERNEL_DIR=../../kernel/
2
3 ARCH=arm64
4 CROSS_COMPILE=aarch64-linux-gnu-
5 export ARCH CROSS_COMPILE
6
7 obj-m := chrdev.o
8 out = chrdev_test
9
```

株式会社日昇テクノロジー

```
10 all:
11 $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) modules
12 $(CROSS_COMPILE)gcc -o $(out) main.c
13
14 .PHONY:clean
15 clean:
16 $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) clean
17 rm $(out)
```

Makefile は以前と比べて、テストプログラムのコンパイル部分が追加されました。

・1 行目：Makefile は変数 KERNEL_DIR を定義して、カーネルソースコードのディレクトリを保存します。

・3-5 行目：ツールチェーンを指定し、環境変数をエクスポートします。

・7 行目：変数 obj-m は、モジュールとしてコンパイルする必要があるターゲットファイル名を保存します。

・8 行目：変数 out は、テストプログラムとしてコンパイルする必要があるターゲットファイル名を保存します。

・11 行目：'$(MAKE) modules'は、Linux トップレベル Makefile の擬似ターゲット modules を実際に実行します。オプション'-C'を使って make ツールをソースコードディレクトリに移動させ、トップレベル Makefile を読み込ませます。'M=$(CURDIR)'は現在のディレクトリに戻り、現在のディレクトリの Makefile を読み込んでカーネルモジュールのコンパイルを開始します。CURDIR は make の組み込み変数で、自動的に現在のディレクトリに設定されます。

・12 行目：交差コンパイルツールチェーンを使用してテストプログラムをコンパイルします。

株式会社日昇テクノロジー

4.6.3.2 コンパイルコマンドの説明

```
make
```

コンパイルに成功すると、実験ディレクトリに"chrdev.ko"ドライバモジュールファイルと"chrdev_test"テストプログラムが生成されます。

4.6.4 プログラム実行結果

用意した Makefile で make を実行すると、chrdev.ko ファイルとドライバテストプログラム chrdev_test が生成されます。nfs ネットワークファイルシステムや scp を使ってファイルを開発ボードにコピーします。以下のコマンドを実行します：

```
sudo insmod chrdev.ko

cat /proc/devices
```

```
debian@npi:~/dirver$ sudo insmod chrdev.ko
debian@npi:~/dirver$ cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 29 fb
 81 video4linux
 89 i2c
 90 mtd
108 ppp
116 alsa
128 ptm
136 pts
153 spi
180 usb
189 usb_device
207 ttymxc
216 rfcomm
226 drm
244 EmbedCharDev
245 rpmb
246 ttyGS
247 ttyLP
248 watchdog
249 tee
250 iio
```

/proc/devices ファイルから、登録した文字デバイス EmbedCharDev のメインデバイス番号が 244 であることがわかります。このメインデバイス番号は後で使用するので、実際のボードに応じて調整

株式会社日昇テクノロジー

が必要です。

```
mknod /dev/chrdev c 244 0
```

root 権限で mknod コマンドを使って、新しいデバイス chrdev を作成します、下図を参照。

```
debian@npi:~/dirver$ ls -l /dev/chrdev
crw-r--r-- 1 root root 244, 0 Sep 11 03:00 /dev/chrdev
```

root 権限で chrdev_test テストプログラムを実行します、実行結果は下図の通りです。

```
crw-r--r-- 1 root root 244, 0 Sep 11 03:00 /dev/chrdev
debian@npi:~/dirver$ sudo ./chrdev_test
EmbedCharDev test
The content : Hello World
debian@npi:~/dirver$
```

実際には、echo や cat コマンドを使用して、デバイスドライバプログラムをテストすることもできます。

```
echo "EmbedCharDev test" > /dev/chrdev
```

su 権限を持っていない場合は、次のように使用することもできます

```
sudo sh -c "echo 'EmbedCharDev test' > /dev/chrdev"
```

```
cat /dev/chrdev
```

```
debian@npi:~/dirver$ cat /dev/chrdev
Hello World
debian@npi:~/dirver$ sudo sh -c "echo 'EmbedCharDev test' > /dev/chrdev"
debian@npi:~/dirver$ cat /dev/chrdev
EmbedCharDev test
debian@npi:~/dirver$
```

このカーネルモジュールが不要になった場合は、以下のコマンドを実行します：

```
rmmod chrdev.ko

rm /dev/chrdev
```

rmmod コマンドでカーネルモジュールをアンロードし、関連するデバイスファイルを削除します。

4.7 一つのドライバが複数のデバイスをサポート

Linux カーネルでは、メインデバイス番号はデバイスに対応するドライバプログラムを識別し、Linux カーネルにどのドライバプログラムがそのデバイスをサービスするかを伝えます。しかし、副デバイス番号は同種の複数のデバイスを表します。各デバイスの機能は異なります。どのように一つ

株式会社日昇テクノロジー

のドライバプログラムで様々なデバイスを制御するのでしょうか？明らかに、まず副デバイス番号に基づいて様々なデバイスを区別することができます；次に、以前に触れた file 構造体の private_data メンバを使用します。このメンバを利用することで、なぜ open 関数と close 関数のパラメータだけが file 構造体を持っているのかが理解できます。ドライバプログラムが最初に実行する操作は open であり、open 関数を通じて制御したい下層ハードウェアを制御できます。

4.7.1 ハードウェアの紹介

このセクションの実験では、Lubancat_RK ボードを使用します。

4.7.2 実験コードの説明

4.7.2.1 複数のデータバッファを管理する実装方法 1

以下は、前のセクションのプログラムを改善して、2 つのデバイスを生成し、それぞれが独自のデータバッファを管理する第一の実装方法を紹介します。

サンプルコードのディレクトリはこちらです：linux_driver/EmbedCharDev/1_SupportMoreDev/

リスト 33: chrdev.c の変更部分

（../linux_driver/EmbedCharDev/1_SupportMoreDev/chrdev.c に位置）

```
1 #define DEV_NAME "EmbedCharDev"
2 #define DEV_CNT (2) (1)
3 #define BUFF_SIZE 128
4 //文字デバイスのデバイス番号を定義
5 static dev_t devno;
6 //文字デバイス構造体 chr_dev を定義
7 static struct cdev chr_dev;
```

株式会社日昇テクノロジー

```
8 //データバッファ
9 static char vbuf1[BUFF_SIZE];
10 static char vbuf2[BUFF_SIZE];
```

・2 行目：マクロ DEV_CNT を修正し、元々の 1 から 2 に変更しました。これにより、ドライバプログラムは 2 つのデバイスを管理できるようになります。

・9-10 行目：2 つのデータバッファを定義しました。

リスト 34: chr_dev_open 関数の変更

(../linux_driver/EmbedCharDev/1_SupportMoreDev/chrdev.c に位置)

```
1 static int chr_dev_open(struct inode *inode, struct file *filp)
2 {
3 printk("¥nopen¥n ");
4 switch (MINOR(inode->i_rdev)) {
5 case 0 : {
6 filp->private_data = vbuf1;
7 break;
8 }
9 case 1 : {
10 filp->private_data = vbuf2;
11 break;
12 }
13 }
14 return 0;
15 }
```

株式会社日昇テクノロジー

inode 構造体には、デバイスファイルのデバイス番号が i_rdev メンバに保存されています。

・4 行目：chr_dev_open 関数では、マクロ MINOR を使用してデバイスファイルの副デバイス番号を取得し、private_data を各自のデータバッファに指向させます。

・5-12 行目：副デバイス番号が 0 のデバイスは vbuf1 のデータを管理し、副デバイス番号が 1 のデバイスは vbuf2 のデータを管理することで、1 つのデバイスドライバが複数のデバイスを管理できるようになります。

これにより、ドライバは private_data を読み書きするだけでよくなります。

リスト 35: chr_dev_write 関数の変更

(../linux_driver/EmbedCharDev/1_SupportMoreDev/1_SupportMoreDev.c に位置)

```
1 static ssize_t chr_dev_write(struct file *filp, const char __user * buf, size_t count, loff_t *ppos)
2 {
3 unsigned long p = *ppos;
4 int ret;
5 char *vbuf = filp->private_data;
6 int tmp = count ;
7 if (p > BUFF_SIZE)
8 return 0;
9 if (tmp > BUFF_SIZE - p)
10 tmp = BUFF_SIZE - p;
11 ret = copy_from_user(vbuf, buf, tmp);
12 *ppos += tmp;
13 return tmp;
14 }
```

株式会社日昇テクノロジー

chr_dev_write 関数はほとんど変更がなく、5 行目にコードを追加して、元々の vbuf データを private_data に向けるようにしました。これにより、副デバイス番号が 0 のデバイスへの書き込み時は vbuf1 にデータが書き込まれ、副デバイス番号が 1 のデバイスへの書き込み時も同様です。

リスト 36: chr_dev_read 関数の変更

(../linux_driver/EmbedCharDev/1_SupportMoreDev/chrdev.c に位置)

```
1 static ssize_t chr_dev_read(struct file *filp, char __user * buf, size_t count, loff_t *ppos)
2 {
3 unsigned long p = *ppos;
4 int ret;
5 int tmp = count ;
6 char *vbuf = filp->private_data;
7 if (p >= BUFF_SIZE)
8 return 0;
9 if (tmp > BUFF_SIZE - p)
10 tmp = BUFF_SIZE - p;
11 ret = copy_to_user(buf, vbuf+p, tmp);
12 *ppos +=tmp;
13 return tmp;
14 }
```

chr_dev_read 関数も同様に、6 行目にコードを追加して、元々の vbuf を private_data メンバに向けました。

株式会社日昇テクノロジー

4.7.2.2 実装方式二 i_cdev 変数

以前に inode の i_cdev メンバについて説明しました。デバイスファイルを開く過程で、対応する文字デバイス構造体 cdev をこの変数に保存することで、デバイスファイルへのアクセスを容易にします。その変数を利用することもできます。

サンプルコードのディレクトリ：linux_driver/EmbedCharDev/2_SupportMoreDev/

リスト 37: デバイスの定義

(../linux_driver/EmbedCharDev/2_SupportMoreDev/2_SupportMoreDev.c に位置)

```
1 /* 仮想文字デバイス */
2 struct chr_dev {
3     struct cdev dev;
4     char vbuf[BUFF_SIZE];
5 };
6 //文字デバイス 1
7 static struct chr_dev vcdev1;
8 //文字デバイス 2
9 static struct chr_dev vcdev2;
```

新しい構造体 struct chr_dev が定義されており、2 つのメンバがあります：文字デバイス構造体 dev とデバイスに対応するデータバッファです。struct chr_dev 型を使用して、2 つの仮想デバイス vcdev1 と vcdev2 を定義しています。

株式会社日昇テクノロジー

リスト 38: chrdev_init 関数

(../linux_driver/EmbedCharDev/2_SupportMoreDev/chrdev.c に位置)

```
1 static int __init chrdev_init(void)
2 {
3 int ret;
4 printk("4 chrdev init¥n");
5 ret = alloc_chrdev_region(&devno, 0, DEV_CNT, DEV_NAME);
6 if (ret < 0)
7 goto alloc_err;
8 //最初のデバイスと関連付ける：vdev1
9 cdev_init(&vcdev1.dev, &chr_dev_fops);
10 ret = cdev_add(&vcdev1.dev, devno+0, 1);
11 if (ret < 0) {
12 printk("vcdev1 の追加に失敗 ");
13 goto add_err1;
14 }
15 //2 番目のデバイスと関連付ける：vdev2
16 cdev_init(&vcdev2.dev, &chr_dev_fops);
17 ret = cdev_add(&vcdev2.dev, devno+1, 1);
18 if (ret < 0) {
19 printk("vcdev2 の追加に失敗 ");
20 goto add_err2;
```

株式会社日昇テクノロジー

```
21 }
22 return 0;
23 add_err2:
24 cdev_del(&(vcdev1.dev));
25 add_err1:
26 unregister_chrdev_region(devno, DEV_CNT);
27 alloc_err:
28 return ret;
29 }
```

・10、17 行目：cdev_add を使用して文字デバイスを追加する際、順に追加します。

・23-24 行目：仮想デバイス 1 の追加に失敗した場合、直接デバイス番号の登録解除だけを行います。

・25-26 行目：仮想デバイス 2 の追加に失敗した場合は、仮想デバイス 1 を削除し、その後でデバイス番号の登録解除を行います。

リスト 39: chrdev_exit 関数

（../linux_driver/EmbedCharDev/2_SupportMoreDev/chrdev.c に位置）

```
1 static void __exit chrdev_exit(void)
2 {
3 printk("chrdev exit¥n");
4 unregister_chrdev_region(devno, DEV_CNT);
5 cdev_del(&(vcdev1.dev));
6 cdev_del(&(vcdev2.dev));
7 }
```

株式会社日昇テクノロジー

chrdev_exit 関数は、申請したデバイス番号を登録解除し、cdev_del を使用して 2 つの仮想デバイスを削除します。

リスト 40: chr_dev_open 関数および chr_dev_release 関数の変更

(../linux_driver/EmbedCharDev/2_SupportMoreDev/chrdev.c に位置)

```
1 static int chr_dev_open(struct inode *inode, struct file *filp)
2 {
3 printk("open¥n");
4 filp->private_data = container_of(inode->i_cdev, struct chr_dev, dev);
5 return 0;
6 }
7 static int chr_dev_release(struct inode *inode, struct file *filp)
8 {
9 printk("release¥n");
10 return 0;
11 }
```

inode の i_cdev メンバには、対応する文字デバイス構造体のアドレスが保存されますが、仮想デバイスは cdev をカプセル化した構造体です。どうやって仮想デバイスのデータバッファにアクセスできるでしょうか？そのために、Linux は container_of マクロを提供しています。このマクロは、構造体のあるメンバのアドレスからその構造体のアドレスを取得するために使用されます。このマクロは、構造体メンバの実際のアドレス、構造体の型、および構造体メンバの名前の 3 つのパラメータを必要とします。chr_dev_open 関数では、inode の i_cdev メンバを通じて対応する仮想デバイス構造体にアクセスし、それをファイルポインタ filp の private_data メンバに保存します。例えば、仮想デバイス 1 を開いた場合、inode->i_cdev は vcdev1 の dev メンバを指します。container_of マクロを

株式会社日昇テクノロジー

使用して、vcdev1 構造体のアドレスを取得し、対応するデータバッファにアクセスできます。

リスト 41: chr_dev_write 関数の変更

(../linux_driver/EmbedCharDev/2_SupportMoreDev/chrdev.c に位置)

```
1 static ssize_t chr_dev_write(struct file *filp, const char __user * buf, size_t count, loff_t *ppos)
2 {
3 unsigned long p = *ppos;
4 int ret;
5 //ファイルのプライベートデータを取得する
6 struct chr_dev *dev = filp->private_data;
7 char *vbuf = dev->vbuf;
8 int tmp = count ;
9 if (p > BUFF_SIZE)
10 return 0;
11 if (tmp > BUFF_SIZE - p)
12 tmp = BUFF_SIZE - p;
13 ret = copy_from_user(vbuf, buf, tmp);
14 *ppos += tmp;
15 return tmp;
16 }
```

実際には、第 6 行のコードの追加により、ファイルポインタ filp の private_data メンバから対応する仮想デバイスにアクセスします。第 7 行のコードを変更し、対応するデバイスのデータバッファを指す char 型のポインタ変数を定義します。

株式会社日昇テクノロジー

リスト 42: chr_dev_read 関数の変更

(../linux_driver/EmbedCharDev/2_SupportMoreDev/chrdev.c に位置)

```
1 static ssize_t chr_dev_read(struct file *filp, char __user * buf, size_t count, loff_t *ppos)
2 {
3 unsigned long p = *ppos;
4 int ret;
5 int tmp = count ;
6 //ファイルのプライベートデータを取得する
7 struct chr_dev *dev = filp->private_data;
8 char *vbuf = dev->vbuf;
9 if (p >= BUFF_SIZE)
10 return 0;
11 if (tmp > BUFF_SIZE - p)
12 tmp = BUFF_SIZE - p;
13 ret = copy_to_user(buf, vbuf+p, tmp);
14 *ppos +=tmp;
15 return tmp;
16 }
```

読み取り関数も、書き込み関数の変更と基本的に同じで、ここではコードのみを掲載し、詳細な説明は省略します。

4.7.3 実験の準備

2 つの異なる方法でのカーネルモジュールのソースコードをそれぞれ取得し、linux_driver ディレク

株式会社日昇テクノロジー

トリ内の対応するコードをカーネルと同じレベルのディレクトリに解凍します。EmbedCharDev ディレクトリに移動し、1_SupportMoreDev と 2_SupportMoreDev のディレクトリを見つけます。

4.7.3.1 makefile の説明

Makefile については、前セクションと同様であるため、ここでは省略します。

4.7.3.2 コンパイルコマンドの説明

実験ディレクトリで以下のコマンドを入力してドライバモジュールをコンパイルします：

```
make
```

コンパイルに成功すると、実験ディレクトリにドライバモジュールファイルがそれぞれ生成されます。

4.7.4 プログラム実行結果

NFS や SCP を使用して、コンパイルされたドライバモジュールを開発ボードにコピーします。

```
insmod 1_SupportMoreDev.ko

sudo mknod /dev/chrdev1 c 244 0

sudo mknod /dev/chrdev2 c 244 1
```

上のコマンドを使用して、新しいカーネルモジュールをロードし、2 つの新しい文字デバイスを手で作成します。メインデバイス番号は/proc/devices に記載されている通りに設定し、/dev/chrdev1 と/dev/chrdev2 として設定します。読み書きテストを開始します：

株式会社日昇テクノロジー

```
echo "hello world" > /dev/chrdev1

# または

sudo sh -c "echo 'hello world' > /dev/chrdev1"

echo "123456" > /dev/chrdev2

# または

sudo sh -c "echo '123456' > /dev/chrdev2"
```

```
cat /dev/chrdev1

cat /dev/chrdev2
```

```
debian@npi:~/dirver$ sudo sh -c "echo 'hello world' > /dev/chrdev1"
debian@npi:~/dirver$ sudo sh -c "echo '123456' > /dev/chrdev2"
debian@npi:~/dirver$ cat /dev/chrdev1
hello world
debian@npi:~/dirver$ cat /dev/chrdev2
123456
debian@npi:~/dirver$
```

chrdev1 デバイスには文字列「hello world」が、chrdev2 デバイスには文字列「123456」が保存されていることがわかります。わずか数行のコードで、複数のデバイスを制御できるドライバプログラムを実装できました。

要点として、一つのドライバが複数のデバイスをサポートする具体的な実装方法は、file の private_data メンバの利用方法に焦点を当てています。第一の方法では、各デバイスのデータバッファをこのメンバに格納し、読み書き関数で直接該当するデータバッファに対して操作を行います。第二の方法では、データバッファと文字デバイス構造体を一緒にカプセル化し、file 構造体の inode メンバの i_cdev が対応する文字デバイス構造体を指しているため、container_of マクロを使用してカプセル化された構造体のアドレスを取得し、対応するデータバッファにアクセスします。

これで、文字デバイスドライバの説明は終わりです。理解できない部分が多い場合は、本章を再度整理して、文字デバイスドライバフレームワーク全体の理解を深めることをお勧めします。

株式会社日昇テクノロジー

第 5 章 文字デバイスドライバ - LED ライト点灯実験

文字デバイスの章で学んだ知識をもとに、基本的なフレームワークを把握しました。具体的には、デバイス番号の申請と解放、デバイスの追加と削除、cdev 構造体の初期化、追加、削除、そして cdev_init 関数による cdev と file_operations の関連付けを理解しました。cdev 構造体と file_operations 構造体は非常に重要であるため、重点的に理解することをお勧めします。

このセクションでは、興奮させる小さな実験、LED の点灯を行います。以前はレジスタ操作によって LED を点灯させましたが、このセクションでは Linux 環境下で LED を点灯させる方法を学びます。

まず、レジスタを直接操作して LED を点灯させる方法と、ドライバプログラムを介して LED を点灯させる方法の違いを理解する必要があります。

5.1 デバイスドライバの役割と本質

レジスタ操作による LED の点灯と、ドライバプログラムによる LED の点灯の最も本質的な違いは、操作システムの使用の有無です。操作システムの存在により、アプリケーションソフトウェアとハードウェアプラットフォームの結合度が大幅に低下し、ハードウェアとアプリケーションソフトウェアの間の橋渡し役を果たします。これにより、アプリケーションソフトウェアはドライバプログラムの API インターフェースを呼び出すだけで、ハードウェアに要求されたタスクを実行させることができます。アプリケーションソフトウェアは、ハードウェアの具体的な動作方法を知る必要がありません。これにより、アプリケーションプログラムの移植性と開発効率が大幅に向上します。

5.1.1 ドライバの役割

デバイスドライバは、底層のハードウェアと直接対話し、ハードウェアデバイスの具体的な動作方

株式会社日昇テクノロジー

式に従ってデバイスレジスタを読み書きし、デバイスのポーリング、割り込み処理、DMA 通信を行い、物理メモリから仮想メモリへのマッピングを行います。これにより、通信デバイスがデータの送受信を行い、表示デバイスがテキストや画像を表示し、記憶デバイスがファイルやデータを記録できるようになります。

オペレーティングシステムが存在しない場合、エンジニアはハードウェアデバイスの特性に基づいてインターフェースを自由に定義することができます。例えば、LED に対して LightOn()、LightOff()などを定義することができます。しかし、オペレーティングシステムが存在する場合、デバイスドライバのアーキテクチャは該当するオペレーティングシステムによって定義され、ドライバエンジニアはこれに従ってデバイスドライバを設計する必要があります。これにより、デバイスドライバはオペレーティングシステムのカーネルにうまく統合されることができます。

5.1.2 オペレーティングシステムの有無の違い

1) オペレーティングシステムがない場合、デバイスドライバは直接レジスタを操作してハードウェアを制御します。このようなシステムでは、デバイスドライバは存在しなければなりません。通常、各種デバイスドライバはソフトウェアモジュールとして定義され、.h ファイルと.c ファイルで構成されます。前者はデバイスドライバのデータ構造を定義し、外部関数を宣言し、後者はデバイスドライバの具体的な実装を行います。他のモジュールがこのデバイスを使用する場合は、デバイスドライバのヘッダーファイルを含めて外部インターフェース関数を呼び出すだけでよいです。これは STM32 の開発でよく見られ、比較的簡単です。

2) オペレーティングシステムがある場合、ドライバがハードウェアを動作させる部分は依然として不可欠です。加えて、デバイスドライバをカーネルに統合する必要があります。この統合を実現するためには、すべてのデバイスドライバに対して、オペレーティングシステムカーネル向けのインターフェースを設計する必要があります。このインターフェースは、オペレーティングシステムによって

株式会社日昇テクノロジー

規定され、特定のデバイスに依存しない一貫した構造を持っています。これにより、デバイスドライバはオペレーティングシステムのカーネル内でハードウェアとの対話モジュールとして機能します。オペレーティングシステムの存在により、デバイスドライバにはより多くのコードと機能が追加され、単一のドライバからオペレーティングシステム内でハードウェアと対話するモジュールへと変貌します。これにより、API としてオペレーティングシステムに対して提示されます。

オペレーティングシステムの存在がもたらす利点とは何でしょうか？

まず、オペレーティングシステムによって多タスクの並行実行が可能になります。次に、オペレーティングシステムによってメモリ管理機構が提供され、32 ビット Linux オペレーティングシステムでは、各プロセスが独立して 4GB のメモリ空間にアクセスできるようになります。アプリケーションプログラムにとって、統一されたシステムコールインターフェースを使用してさまざまなデバイスにアクセスできるため、write()、read()などの関数を使用してファイルを読み書きするだけで、特定のデバイスのタイプや動作方式を気にすることなく、さまざまな文字デバイスやブロックデバイスにアクセスできます。

5.2 メモリ管理ユニット MMU

Linux 環境で直接物理メモリにアクセスすることは非常に危険です。ユーザーがうっかりメモリ内のデータを変更してしまうと、エラーの原因となったり、システムがクラッシュする可能性があります。これらの問題を解決するために、カーネルは MMU を導入しています。

5.2.1 MMU の機能

MMU は、プログラミングに便利な統一されたメモリ空間の抽象を提供します。実際には、プログラムに書かれた変数のアドレスは仮想メモリの中のアドレスです。プロセッサがこのアドレスにアクセスしようとすると、MMU はこの仮想アドレス(Virtual Address)を実際の物理アドレス(Physical Address)に変換し、その後プロセッサは実際の物理アドレスに対して操作を行います。MMU は実際

株式会社日昇テクノロジー

のハードウェアであり、ソフトウェアプログラムではありません。その主な役割は、仮想アドレスを実際の物理アドレスに変換することと、メモリの管理と保護です。異なるプロセスはそれぞれ独自の仮想アドレス空間を持ち、あるプロセスのプログラムが別のプロセスが使用する物理アドレスを変更することはできません。これにより、プロセス間の干渉を防ぎ、相互に隔離します。また、仮想アドレス空間の連続したアドレスを使用して、物理メモリ内の断片化された大きなメモリバッファにアクセスすることができます。多くのリアルタイムオペレーティングシステムは、MMU のない CPU 上で動作することができます。例えば、uCOS、FreeRTOS、uCLinux などです。以前は、Linux システムを実行するためには、その CPU が MMU を持っている必要がありましたが、現在では Linux も MMU のない CPU 上で動作することができます。全体として、MMU には以下のような機能があります：

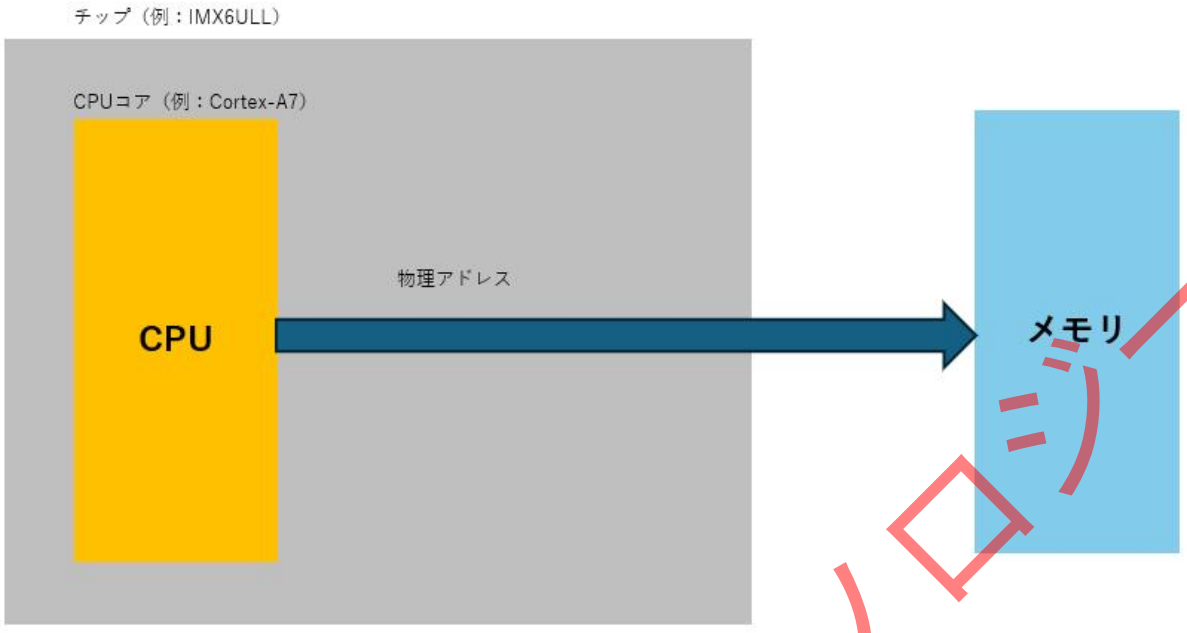
- メモリの保護：MMU は、指定されたメモリブロックに対して読み取り、書き込み、実行の権限を設定します。これらの権限はページテーブルに保存され、MMU は CPU が特権モードにあるかユーザーモードにあるかをチェックし、オペレーティングシステムが設定した権限と一致する場合にのみアクセスを許可します。CPU が仮想アドレスにアクセスしようとすると、MMU はその仮想アドレスを物理アドレスに変換します。一致しない場合は例外が発生し、メモリが悪意のある変更から保護されます。

- 便利で統一されたメモリ空間の抽象を提供し、仮想アドレスから物理アドレスへの変換を実現します：CPU は仮想メモリ上で動作し、仮想メモリは実際の物理メモリよりもはるかに大きいことが多いため、CPU は比較的大きなアプリケーションプログラムを実行できます。

仮想アドレスと物理アドレスとは何か？

MMU が有効になっていない場合、CPU が命令を読み取るかメモリにアクセスするときに出力されるアドレスは直接チップのピンに出力され、このアドレスは直接メモリに受け取られます。このアドレスは物理アドレスと呼ばれます。

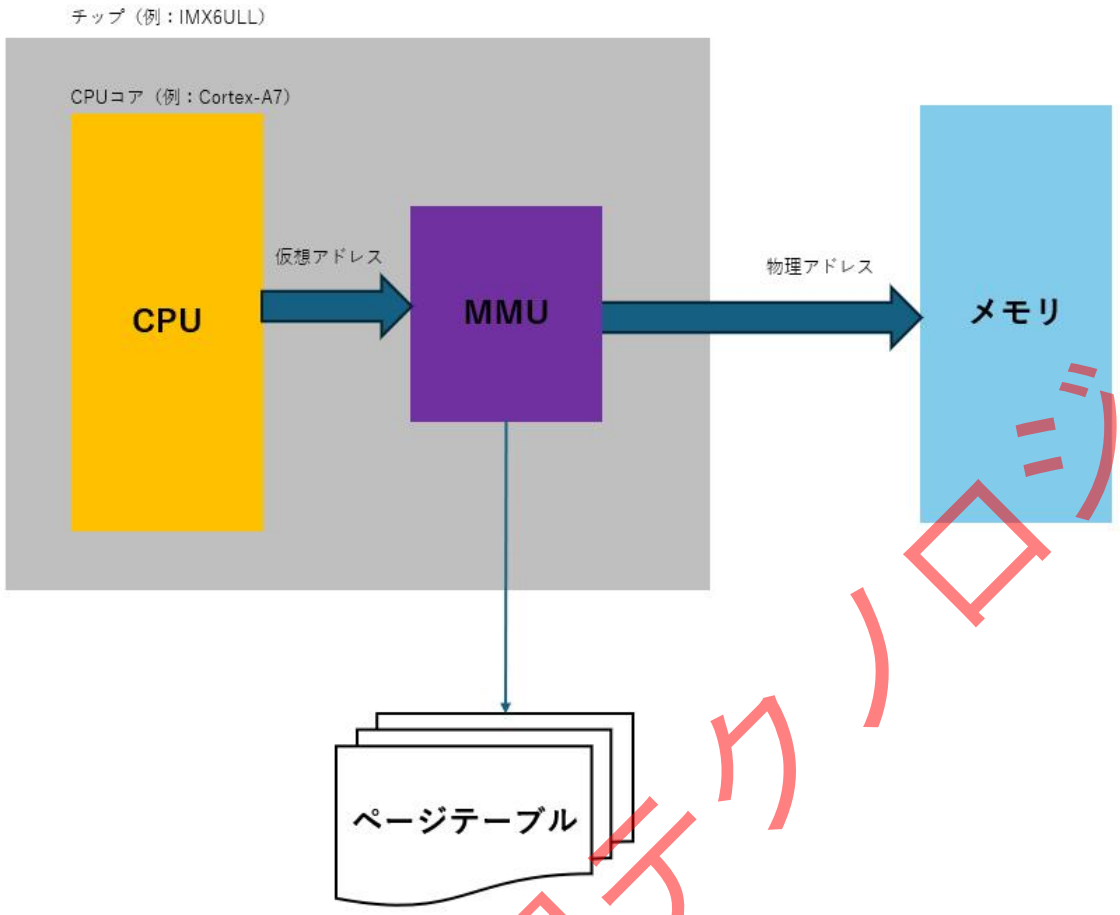
株式会社日昇テクノロジー



簡単に言うと、物理アドレスはメモリセルの絶対アドレスです。たとえば、コンピュータに 8GB のメモリカードが装着されている場合、最初のストレージユニットは物理アドレス 0x0000、メモリカードの 6 番目のストレージユニットは 0x0005 です。プロセッサがどのように処理しても、物理アドレスは最終的にアクセスされる対象です。

CPU が MMU を有効にすると、CPU が出力するアドレスは MMU に送られ、MMU に送られたこのアドレスを仮想アドレスと呼びます。その後、MMU はページテーブルのアドレスレジスタを参照してメモリ内のページテーブル（1 レベルページテーブルと仮定）のエントリを見つけ、実際の物理アドレスを翻訳します。下の図のように

株式会社日昇テクノロジー



32 ビットプロセッサにおいて、その仮想アドレス空間は 4GB(2^32)あります。CPU が MMU（Memory Management Unit：メモリ管理ユニット）を有効にすると、CPU が発行するアドレスは常に仮想アドレスです。仮想アドレスから物理アドレスへのマッピングを実現するために、MMU 内部にはページテーブルの具体的な位置を格納するページテーブルアドレスレジスタがあります。ioremap を使用してある範囲のアドレスをマッピングすることは、ユーザ空間のある範囲のアドレスをデバイスメモリに関連付けることを意味します。これにより、割り当てられた仮想アドレス範囲内での読み書き操作は、実質的にデバイス（レジスタ）へのアクセスとなります。

5.2.2 TLB（Translation Lookaside Buffer）の役割

MMU に関して話すと、TLB の役割についても触れずにはいられません。上記のアドレス変換プロセスからわかるように、一段階のページテーブルでアドレス変換を行う場合、CPU はデータを読み書きするたびに 2 回メモリにアクセスする必要があります。一回目はメモリ内のページテーブルにアク

株式会社日昇テクノロジー

セスし、二回目はそのページテーブルを基に実際にデータを読み書きするメモリアドレスを見つけます。2 段階のページテーブルを使用する場合、CPU はデータを読み書きするたびに 3 回メモリにアクセスする必要があります。これは非常に面倒で CPU の性能を大幅に消費しますが、この問題を解決するために TLB が開発されました。CPU が仮想アドレスを出力する際、MMU は最初に TLB にアクセスします。TLB にその仮想アドレスを直接変換するアドレス記述子が含まれている場合、その記述子を使用して権限のチェックとアドレス変換を直接行います。TLB にその記述子がない場合、MMU はページテーブルにアクセスして記述子を見つけ、権限のチェックとアドレス変換を行い、その後で TLB にその記述子を追加して次回の使用に備えます。TLB は大きくはないため、TLB が満杯になった場合は、round-robin アルゴリズムを使用してエントリの一つを選んで上書きします。

MMU は非常に複雑なため、ここでは深入りせず、その役割を大まかに理解してもらうことにします。興味のある方は、インターネットで関連資料を調べることをお勧めします。初心者には、まず全体像を掴み、その後で重要な詳細に深く掘り下げることをお勧めします。この節では、MMU のアドレス変換機能を主に使用します。Linux 環境で MMU を有効にした後、具体的なレジスタ（物理アドレス）を読み書きするには、物理アドレスから仮想アドレスへの変換関数が必要です。

5.3 アドレス変換関数

物理アドレスから仮想アドレスへの変換関数には、ioremap（アドレスマッピング）関数と iounmap（マッピング解除）関数が含まれます。

5.3.1 ioremap 関数

関数の定義は以下の通りです（カーネルソースコード/arch/arc/mm/ioremap.c から抜粋）。

```
1 void __iomem *ioremap(phys_addr_t paddr, unsigned long size)

2 #define ioremap ioremap
```

パラメータ：

株式会社日昇テクノロジー

- paddr：マッピングされる IO の開始アドレス（物理アドレス）。

- size：マッピングされる空間のサイズ（バイト単位）。

戻り値：__iomem 型のポインタ。マッピングが成功すると、仮想アドレス空間の開始アドレスが返されます。これにより、この仮想アドレスにアクセスすることで、実際の物理アドレスの読み書きが可能になります。

ioremap 関数は、__ioremap 関数によって実現されていますが、__ioremap の最後にマッピングされる I/O 空間と関連する権限フラグ flag は 0 です。ioremap 関数を使用して物理アドレスを仮想アドレスに変換した後、理論的には I/O メモリを直接読み書きすることができますが、ドライバのプラットフォーム間互換性と移植性を考慮して、Linux で指定された関数（iowrite8()、iowrite16()、iowrite32()、ioread8()、ioread16()、ioread32()など）を使用して I/O メモリを読み書きするべきです。

リスト 2: 読み書き I/O 関数

```
1 unsigned int ioread8(void __iomem *addr)
2 unsigned int ioread16(void __iomem *addr)
3 unsigned int ioread32(void __iomem *addr)
4
5 void iowrite8(u8 b, void __iomem *addr)
6 void iowrite16(u16 b, void __iomem *addr)
7 void iowrite32(u32 b, void __iomem *addr)
```

- 第 1 行：1 バイト(8bit)を読み取る
- 第 2 行：1 ワード(16bit)を読み取る
- 第 3 行：1 ダブルワード(32bit)を読み取る
- 第 5 行：1 バイト(8bit)を書き込む

- 第 6 行：1 ワード(16bit)を書き込む
- 第 7 行：1 ダブルワード(32bit)を書き込む

読み込み I/O については、それぞれ__iomem 型ポインタのパラメータがあり、マッピングされたアドレスを指しており、読み取ったデータを返す；書き込み I/O については、2 つのパラメータがあり、1 つ目は書き込むデータ、2 つ目は書き込み先のアドレスで、戻り値はない。これらの関数と似たものに writeb、writew、writel、readb、readw、readl などがあり、ARM アーキテクチャ下では、writex(readx)関数と iowritex(ioreadx)の間にいくつかの違いがあり、writex(readx)はエンディアンのチェックを行わず、iowritex(ioreadx)はエンディアンのチェックを行う。

例えば、RGB ライトの青色 LED のデータレジスタを操作する必要がある場合、51 や STM32 ではマニュアルを参照して対応するレジスタを探し、レジスタの該当するビットにデータ 0 または 1 を書き込むことで LED の点灯/消灯を実現できます（出力モードやプルアップ/プルダウン等が既に設定されていると仮定）。前述の Linux 環境ではない場合も似たような方法を使用しますが、Linux 環境で MMU が有効になっている場合は、LED ライトのピンに対応するデータレジスタ（物理アドレス）をプログラムの仮想アドレス空間にマッピングし、レジスタを操作するように仮想アドレスを操作できます。具体的なコードは以下の通りです。

リスト 3: アドレスマッピング（GPIO1_A7 を例として）

```
1 #define GPIO1_BASE (0xFD5F8000)
2 #define GPIO1_DR (GPIO1_BASE+0x0000)
3
4 va_dr = ioremap(GPIO1_DR, 4); // 物理アドレスを仮想アドレスにマッピング：GPIO1 のデータレジスタ
5 val = ioread32(va_dr);
6 val |= (0x00400000); // GPIO1_A7 ピンを低電平に設定
7 writel(val, va_dr);
```

株式会社日昇テクノロジー

• 第 1-2 行：レジスタの物理アドレスを定義

• 第 4 行：物理アドレス GPIO1_DR を仮想アドレスポインタにマッピングし、このアドレスのサイズは 4 バイト

• 第 5-6 行：そのアドレスの値を読み取り、一時変数に保存し、再割り当て

• 第 7 行：値をマッピング後の仮想アドレスに再度書き込むことで、実際にレジスタにデータを書き込んだ

5.3.2 iounmap 関数

iounmap 関数の定義は以下の通りです（カーネルソースコード/arch/arc/mm/ioremap.c から抜粋）。

リスト 4: アドレスマッピング解除関数（カーネルソースコード/arch/arc/mm/ioremap.c）

```
1 void iounmap(void *addr)
2 #define iounmap iounmap
```

- **パラメータ**：

- addr：ioremap によってマッピングされた後の開始アドレス（仮想アドレス）。

- **戻り値**：なし

ioremap でマッピングされたアドレスを解除するには、以下のように書きます。

リスト 5: ioremap マッピングアドレスの解除:linenos:

```
iounmap(va_dr); // ioremap でマッピングした開始アドレス（仮想アドレス）を解放する
```

5.4 LED ライトの点灯実験

これまでの章でカーネルモジュールからキャラクタデバイスドライバーに至るまで、理論から実験まで、すべてが準備完了しました。さあ、LED のドライバーコードを書き始めましょう。まず、操作するレジスタアドレスを含む LED キャラクタデバイス構造体が必要です。次に、デバイスの登録が必要なモジュールのロード関数と、要求されたリソースを解放するアンロード関数が必要です。そして、

株式会社日昇テクノロジー

file_operations 構造体および open、write、read 関連のインターフェースの実装が続きます。

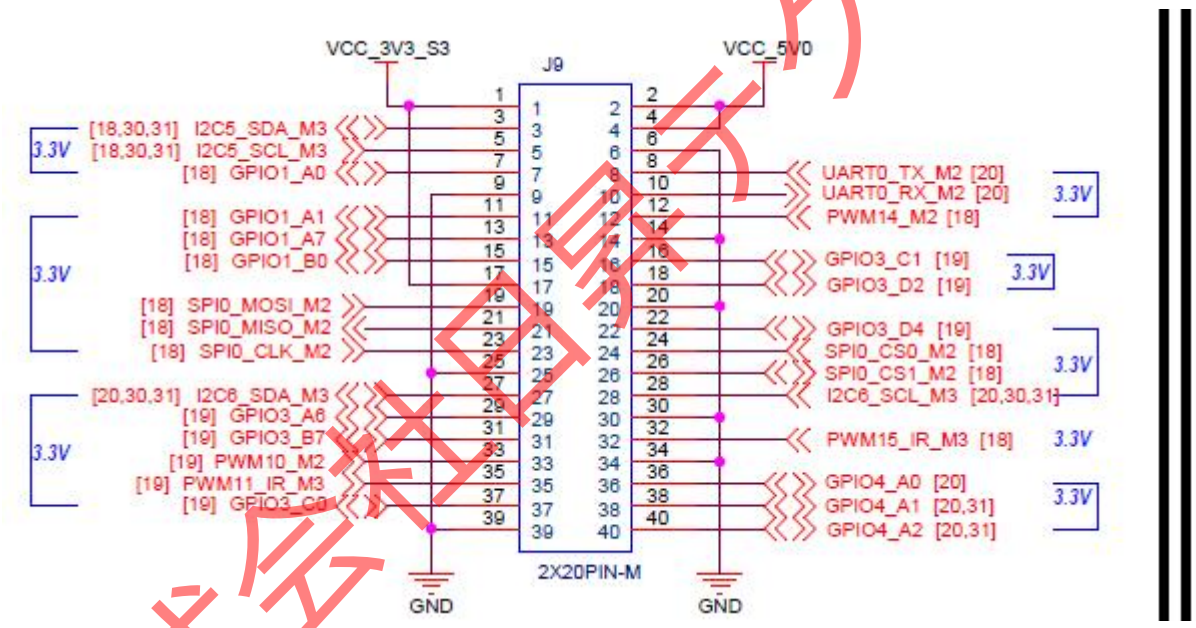
5.4.1 実験説明

5.4.1.1 ハードウェア紹介

このセクションの実験では、lubancat_RK のボード上のシステム LED ライト（または 40pin に自身で LED を外部接続）を使用します。

5.4.1.2 ハードウェア回路図分析

LubanCat4 のボードは、使用できる GPIO はたくさんありますので一つ選択して LED を接続してテストしてください。



ヒント: RK3588 では、ピンは GPIO 番号で識別され、5 つのグループ(GPIO1～4)に分けられ、各グループには 32 の多機能ピンがあります。さらに、4 つのサブグループ(A、B、C、D)に分けられ、各サブグループには 8 つのピン(0～7)があります。例えば、 GPIO1_C7 は、 GPIO1 の大グループ、第 3 のサブグループ、第 8 のピンです。

LED ライトを制御するためには、上記の GPIO のレジスタを読み書きすることになります。大まかに以下のステップに分けられます：

株式会社日昇テクノロジー

- GPIO のクロックを有効にする（デフォルトでオンなので設定不要）

- ピンの多機能を GPIO に設定する（リセットデフォルトが GPIO なので設定不要）

- ピンの属性（プルアップ/プルダウン、速度、ドライブ能力、デフォルト）を設定する

- GPIO ピンを出力として制御し、高/低電圧を出力する

GPIO のクロックはデフォルトでオンになっており、ピンはデフォルトで GPIO として多機能になっているため、GPIO のピンの入出力モードと電圧レベルの設定のみを行えばよいです。

5.4.1.3 LED ライトのレジスタ設定

RK3588 チップのレジスタについて、使用するレジスタを簡単に整理しますが、各レジスタの設定方法については深くは掘り下げません。この部分の内容は、伝統的な STM32 マイクロコントローラの設定に似ており、公式 STM32 マイクロコントローラの関連チュートリアルを学習することができます。

ヒント: rk3588 関連のデータシートとリファレンスマニュアルは、資料のクラウドストレージでダウンロードできます：lubancat/5-RockChip 公式文書。

rk3588 に関連する文書は以下の通りです：

- Rockchip_RK3588_TRM_Part1_xxx.pdf

- Rockchip_RK3588_Datasheet_xxx.pdf

- Rockchip_RK3588S_Datasheet_xxx.pdf

5.4.1.3.1 ピンの多機能設定

rockchip 系列のチップについては、リファレンスマニュアルとデータシートを参照してピンの多機能を確認する必要があります。ピンの多機能に関する情報は、データシートで検索できます。

GPIO1_C7 の例では、Rockchip_RK3588_Datasheet を参照すると、このピンは GPIO 機能や PWM などの機能に多機能設定できることがわかります。

株式会社日昇テクノロジー

PDM1_SDI0_M1/PCIE30X1_1_PERSTN_M2/PWM3_IR_M3/SPI2_CS0_M0/GPIO1_A7_u	C25

Rockchip_RK3588_TRM_Part1 を参照すると、GPIO1 グループの多機能設定は合計 8 つのレジスタがあることがわかります。

Name	Offset	Size	Reset Value	Description
BUS_IOC_GPIO0B_IOMUX_SEL_H	0x000C	W	0x00000000	GPIO0B IOMUX Select High bits
BUS_IOC_GPIO0C_IOMUX_SEL_L	0x0010	W	0x00000000	GPIO0C IOMUX Select Low bits
BUS_IOC_GPIO0C_IOMUX_SEL_H	0x0014	W	0x00000000	GPIO0C IOMUX Select High bits
BUS_IOC_GPIO0D_IOMUX_SEL_L	0x0018	W	0x00000000	GPIO0D IOMUX Select Low bits
BUS_IOC_GPIO0D_IOMUX_SEL_H	0x001C	W	0x00000000	GPIO0D IOMUX Select High bits
BUS_IOC_GPIO1A_IOMUX_SEL_L	0x0020	W	0x00000000	GPIO1A IOMUX Select Low bits
BUS_IOC_GPIO1A_IOMUX_SEL_H	0x0024	W	0x00000000	GPIO1A IOMUX Select High bits
BUS_IOC_GPIO1B_IOMUX_SEL_L	0x0028	W	0x00000000	GPIO1B IOMUX Select Low bits
BUS_IOC_GPIO1B_IOMUX_SEL_H	0x002C	W	0x00000000	GPIO1B IOMUX Select High bits
BUS_IOC_GPIO1C_IOMUX_SEL_L	0x0030	W	0x00000000	GPIO1C IOMUX Select Low bits

BUS_IOC_GPIO1A_IOMUX_SEL_H レジスタについて、Rockchip_RK3588_TRM_Part1 を参照すると、

株式会社日昇テクノロジー

BUS_IOC_GPIO1A_IOMUX_SEL_H

Address: Operational Base + offset (0x0024)

Bit	Attr	Reset Value	Description
31:16	WO	0x0000	write_enable Write enable for lower 16bits, each bit is individual 1'b0: Write access disable 1'b1: Write access enable
15:12	RW	0x0	gpio1a7_sel 4'h0: GPIO 4'h2: PDM1_SDI0_M1 4'h4: PCIE30X1_1_PERSTN_M2 4'h7: HDMI_DEBUG0 4'h8: SPI2_CS0_M0 4'hb: PWM3_IR_M3
11:8	RW	0x0	gpio1a6_sel 4'h0: GPIO 4'h5: HDMI_TX1_HPD_M0 4'h8: SPI2_CLK_M0
7:4	RW	0x0	gpio1a5_sel 4'h0: GPIO 4'h5: HDMI_TX0_HPD_M0 4'h8: SPI2_MOSI_M0
3:0	RW	0x0	gpio1a4_sel 4'h0: GPIO 4'h5: HDMI_TX1_SCL_M2 4'h8: SPI2_MISO_M0

Operational Base　Address

MIPICDPHY0_GRF	0xFD5E8000
MIPICDPHY1_GRF	0xFD5EC000
PMU1_IOC	0xFD5F0000
PMU2_IOC	0xFD5F4000
BUS_IOC	0xFD5F8000
VCCIO1_4_IOC	0xFD5F9000
VCCIO3_5_IOC	0xFD5FA000
VCCIO2_IOC	0xFD5FB000

このレジスタは合計 32 ビットで、12~15 の 4 ビットは gpio1a7 の設定です。このレジスターの Address は Operational Base + offset (0x0024)です。この Address の 12~15 の 4 ビットを 4'h0 に設定すれば gpio1a7 を gpio に設定できます。

5.4.1.3.2 ピンの電圧レベル

GPIO レジスタを設定して、入出力、高低電圧、割り込み、デバウンスなど、ピンのドライブ能力や電気的属性などを設定します。主に General Register Files (GRF)を設定します（GPIO1 グループを

株式会社日昇テクノロジー
Confidencial

例として、詳細は Rockchip_RK3588_TRM_Part1 マニュアルを参照してください）。

20.4.1 Registers Summary

Name	Offset	Size	Reset Value	Description
GPIO_SWPORT_DR_L	0x0000	W	0x00000000	Port Data Register (Low)
GPIO_SWPORT_DR_H	0x0004	W	0x00000000	Port Data Register (High)
GPIO_SWPORT_DDR_L	0x0008	W	0x00000000	Port Data Direction Register (Low)
				Port Data Direction Register

- GPIO_SWPORT_DR_L：低位ピンデータレジスタで、高低電圧を設定します。

- GPIO_SWPORT_DR_H：高位ピンデータレジスタで、高低電圧を設定します。

GPIO_SWPORT_DR_L および GPIO_SWPORT_DR_H レジスタは以下の通りです。

20.4.2 Detail Registers Description

GPIO_SWPORT_DR_L

Address: Operational Base + offset (0x0000)

Bit	Attr	Reset Value	Description
31:16	WO	0x0000	write_mask Write enable for lower 16 bits, each bit is individual. 1'b0: Write access disable 1'b1: Write access enable

Copyright 2022 © Rockchip Electronics Co., Ltd.

Bit	Attr	Reset Value	Description
15:0	RW	0x0000	swport_dr_low Output data for the lower 16 bits of I/O Port, each bit is individual. 1'b0: Low 1'b1: High Values written to this register are output on the I/O signals for the lower 16 bits of I/O Port if the corresponding data direction bits for I/O Port are set to Output mode. The value read back is equal to the last value written to this register.

株式会社日昇テクノロジー

GPIO_SWPORT_DR_H

Address: Operational Base + offset (0x0004)

Bit	Attr	Reset Value	Description
31:16	WO	0x0000	write_mask Write enable for lower 16 bits, each bit is individual. 1'b0: Write access disable 1'b1: Write access enable
15:0	RW	0x0000	swport_dr_high Output data for the upper 16 bits of I/O Port, each bit is individual. 1'b0: Low 1'b1: High Values written to this register are output on the I/O signals for the upper 16 bits of I/O Port if the corresponding data direction bits for I/O Port are set to Output mode. The value read back is equal to the last value written to this register.

GPIO_SWPORT_DR_L レジスタについて説明します。このレジスタは上位 16 ビットと下位 16 ビットから構成され、上位 16 ビットが下位 16 ビットの書き込み有効を制御し、下位 16 ビットが GPIO の高低電圧を制御します。GPIO_SWPORT_DR_H も同様です。 GPIO1_C7 の高低電圧を制御する場合は、GPIO_SWPORT_DR_H レジスタを書き込む必要があります。C7 は GPIO1 の A-D グループ、合計 64 ピンのうちの上位 32 ピンの範囲に属するため、GPIO_SWPORT_DR_H レジスタの第 7 ビットと 7+16 ビットを 1 に設定します。

5.4.1.3.3 ピンの入出力モード

20.4.1 Registers Summary

Name	Offset	Size	Reset Value	Description
GPIO_SWPORT_DR_L	0x0000	W	0x00000000	Port Data Register (Low)
GPIO_SWPORT_DR_H	0x0004	W	0x00000000	Port Data Register (High)
GPIO_SWPORT_DDR_L	0x0008	W	0x00000000	Port Data Direction Register (Low)
				Port Data Direction Register

- GPIO_SWPORT_DDR_L：低位ピンデータ方向レジスタで、入力または出力を制御します。
- GPIO_SWPORT_DDR_H：高位ピンデータ方向レジスタで、入力または出力を制御します。

PIO_SWPORT_DDR_L と GPIO_SWPORT_DDR_H レジスタの具体的な説明は以下の通りです

株式会社日昇テクノロジー Confidencial

GPIO_SWPORT_DDR_L

Address: Operational Base + offset (0x0008)

Bit	Attr	Reset Value	Description
31:16	WO	0x0000	write_mask Write enable for lower 16 bits, each bit is individual. 1'b0: Write access disable 1'b1: Write access enable
15:0	RW	0x0000	swport_ddr_low Data direction for the lower 16 bits of I/O Port, each bit is individual. 1'b0: Input 1'b1: Output Values written to this register independently control the direction of the corresponding data bit in the lower 16 bits of I/O Port.

GPIO_SWPORT_DDR_H

Address: Operational Base + offset (0x000C)

Bit	Attr	Reset Value	Description
31:16	WO	0x0000	write_mask Write enable for lower 16 bits, each bit is individual. 1'b0: Write access disable 1'b1: Write access enable

Bit	Attr	Reset Value	Description
15:0	RW	0x0000	swport_ddr_high Data direction for the upper 16 bits of I/O Port, each bit is individual. 1'b0: Input 1'b1: Output Values written to this register independently control the direction of the corresponding data bit in the upper 16 bits of I/O Port.

GPIO_SWPORT_DDR_L レジスタについて説明します。このレジスタは上位 16 ビットと下位 16 ビットから構成され、上位 16 ビットが下位 16 ビットの書き込み有効を制御し、下位 16 ビットが GPIO の出力方向を制御します。GPIO_SWPORT_DDR_H も同様です。GPIO1_A7 を出力モードに設定する場合は、GPIO_SWPORT_DDR_H レジスタを書き込む必要があります。A7 は GPIO1 の A-D グループ、合計 64 ピンのうちの上位 32 ピンの範囲に属するため、GPIO_SWPORT_DDR_H レジスタの第 7 ビットと 7+16 ビットを 1 に設定します。

株式会社日昇テクノロジー

5.4.1.3.4 ピンのプルアップ/プルダウン

PMU_GRF_GPIO0A_P	0x0020	W	0x00006962	GPIO0A PU/PD control
PMU_GRF_GPIO0B_P	0x0024	W	0x00009555	GPIO0B PU/PD control
PMU_GRF_GPIO0C_P	0x0028	W	0x0000AAAA	GPIO0C PU/PD control
PMU_GRF_GPIO0D_P	0x002C	W	0x00002A85	GPIO0D PU/PD control

PMU_GRF_GPIO1X_P は、対応する GPIO のプルアップまたはプルダウンを制御するレジスタです。GPIO1_A7 について説明します。

VCCIO1_4_IOC_GPIO1A_P

Address: Operational Base + offset (0x0110)

Bit	Attr	Reset Value	Description
31:16	WO	0x0000	write_enable Write enable for lower 16bits, each bit is individual 1'b0: Write access disable 1'b1: Write access enable

Bit	Attr	Reset Value	Description
15	RW	0x1	gpio1a7_ps GPIO1A7 PS control Weak PU/PD Resistor Selection 1'b0: PD Selection 1'b1: PU Selection
14	RW	0x1	gpio1a7_pe GPIO1A7 PE control Active High Weak PU/PD Resistor Enable 1'b0: PU/PD Disable 1'b1: PU/PD Enable

PMU_GRF_GPIO1A_P レジスタは上位 16 ビットと下位 16 ビットから構成され、上位 16 ビットが下位 16 ビットの書き込み有効を制御し、下位 16 ビットが GPIO のプルアップ/プルダウン設定を制御します。GPIO1_A7 をプルアップモードに設定する場合は、IOC_GPIO1A_P レジスタの[15:14]ビットを 01 に設定し、つまり 14 ビット目と 14+16 ビットを 1 に設定します。

これにより、ハードウェアの原理とレジスタの設定説明は終わりです。より多くのハードウェア情報とレジスタの設定については、回路図とチップマニュアルを参照してください。

5.4.2 コード解説

この章のサンプルコードディレクトリは：linux_driver/led_cdev/

株式会社日昇テクノロジー

5.4.2.1 GPIO レジスタの物理アドレスを定義

GPIO1_C7 を例に、まず GPIO1 のベースアドレスを確定する必要があります。他のレジスタはベースアドレスにオフセットを加えて定義されます。Rockchip_RK3588_TRM_Part1 マニュアルを参照すると、 GPIO1 のベースアドレスは 0XFDB40000 であることがわかります。

Module	Start Address	Size	Module	Start Address	Size
TIMER_NPU(2CH)	FDB00000	32KB	PWM3	FEBF0000	64KB
Reserved	FDB08000	160KB	TSADC	FEC00000	64KB
PVTM_GPU	FDB30000	64KB	SARADC	FEC10000	64KB
Reserved	FDB40000	64KB	GPIO1	FEC20000	64KB
VDPU	FDB50000	64KB	GPIO2	FEC30000	64KB
RGA3_0	FDB60000	64KB	GPIO3	FEC40000	64KB

ベースアドレスを確定した後、残りのレジスタのベースアドレスに対するオフセットを決定する必要があります。

20.4.1 Registers Summary

Name	Offset	Size	Reset Value	Description
GPIO_SWPORT_DR_L	0x0000	W	0x00000000	Port Data Register (Low)
GPIO_SWPORT_DR_H	0x0004	W	0x00000000	Port Data Register (High)
GPIO_SWPORT_DDR_L	0x0008	W	0x00000000	Port Data Direction Register (Low)
GPIO_SWPORT_DDR_H	0x000C	W	0x00000000	Port Data Direction Register (High)
GPIO_INT_EN_L	0x0010	W	0x00000000	Interrupt Enable Register (Low)
GPIO_INT_EN_H	0x0014	W	0x00000000	Interrupt Enable Register (High)

上記の図から、設定する必要のあるレジスタのアドレスは base+offset であることがわかります。コードは以下のようになります：

株式会社日昇テクノロジー

リスト 6: LED ライトに使用する GPIO リソース

```
1 #define GPIO1_BASE (0xFD5F8000)
2
3 // 32 ビットのレジスタで、上位 16 ビットは書き込み有効ビットで、下位 16 ビットの書き込みを制御します；下位 16 ビットは 16 個のピンに対応し、ピンの出力電圧を制御します
4 #define GPIO1_DR_L (GPIO1_BASE + 0x0000) // GPIO1 の下位 16 ピンのデータレジスタアドレス
5 #define GPIO1_DR_H (GPIO1_BASE + 0x0004) // GPIO1 の上位 16 ピンのデータレジスタアドレス
6
7 // 32 ビットのレジスタで、上位 16 ビットは書き込み有効ビットで、下位 16 ビットの書き込みを制御します；下位 16 ビットは 16 個のピンに対応し、ピンの入出力モードを制御します
8 #define GPIO1_DDR_L (GPIO1_BASE + 0x0008) // GPIO1 の下位 16 ピンのデータ方向レジスタアドレス
9 #define GPIO1_DDR_H (GPIO1_BASE + 0x000C) // GPIO1 の上位 16 ピンのデータ方向レジスタアドレス
```

コードではマクロ定義を使用して、LED ライトで使う GPIO リソースの物理アドレスを定義しています。後でこれらの物理アドレスのレジスタを仮想アドレスにマッピングし、設定に使用します。その他の多機能、プルアップ/プルダウンなどのレジスタはデフォルトのままにしておけばよいです。

株式会社日昇テクノロジー

5.4.2.2 LED キャラクタデバイス構造体の作成と初期化

リスト 7: led キャラクタデバイス構造体

```
1 struct led_chrdev {
2     struct cdev dev;
3     unsigned int __iomem *va_dr; // データレジスタの仮想アドレス保持変数
4     unsigned int __iomem *va_ddr; // データ方向レジスタの仮想アドレス保持変数
5
6     unsigned int led_pin; // ピン番号
7 };
8
9 static struct led_chrdev led_cdev[DEV_CNT] = {
10     {.led_pin = 7},
11 };
```

上記のコードでは、LED ライトのための構造体を定義し、RGB ライトの構造体配列を定義して初期化しています。初期化では、.演算子と変数名を使用して構造体メンバを直接初期化しています。メンバ変数を初期化する際には`,`で区切ります。この方法はシンプルで、データ構造のメンバを管理するのに便利です。

- 第 3 行：データレジスタの仮想アドレスを保存する変数を定義
- 第 4 行：データ方向レジスタの仮想アドレスを保存する変数を定義
- 第 6 行：LED のピン
- 第 9 行：LED ランプ構造体メンバ変数を初期化、ボードに複数の LED がある場合は、自分で追加可能

株式会社日昇テクノロジー

5.4.2.3 カーネル RGB モジュールのロードとアンロード関数

第一部分は、カーネル RGB モジュールのロード関数で、主に以下のタスクを完成します：

• LED 構造体内の仮想アドレスをマッピングし、GPIO の物理レジスタアドレスと対応させる

• alloc_chrdev_region()関数を呼び出して、システムから使用されていないデバイス番号を動的に要求する。alloc_chrdev_region()を使用する利点は、使用されていないデバイス番号を自分で調べる手間を省け、デバイス番号の重複問題を避けられることです；

• class_create()関数を呼び出して、RGB ランプのデバイスクラスを作成する；

• 3 つの LED に対してそれぞれ、対応するキャラクタデバイス構造体 cdev と led_chrdev_fops との関連付けを行い、キャラクタデバイス構造体を初期化し、最後にデバイスを登録して作成する。

第二部分は、カーネル RGB モジュールのアンロード関数で、主に以下のタスクを完成します：

• LED 構造体内のマッピングされた仮想アドレスを解放する

• device_destroy()関数を呼び出して、Linux カーネルシステムのデバイスドライバーモデルからデバイスを削除し、/sys/devices/virtual ディレクトリと/dev/ディレクトリ下の対応するデバイスファイルを削除する；

• cdev_del()関数を呼び出して、ハッシュテーブル内のオブジェクトと cdev 構造体自体を解放する；

• 使用されたデバイス番号を解放し、デバイスクラスを削除する。

下記のコードから、これら 3 つの LED が同一のメジャーデバイス番号を使用しており、彼らのマイナーデバイス番号だけが異なることが分かります。

株式会社日昇テクノロジー

リスト 8: カーネル RGB モジュールのロードとアンロード関数

```
1 static __init int led_chrdev_init(void)
2 {
3            int i = 0;
4           dev_t cur_dev;
5           unsigned int val = 0;
6
7           printk("led_chrdev init (lubancat4 GPIO1_A7)¥n");
8
9    led_cdev[0].va_dr = ioremap(GPIO1_DR_H, 4); // GPIO1_C7 のデータレジスタ物理アドレスを仮想アドレスにマッピング
10    led_cdev[0].va_ddr = ioremap(GPIO1_DDR_H, 4); // GPIO1_C7 のデータ方向レジスタ物理アドレスを仮想アドレスにマッピング
11
12    alloc_chrdev_region(&devno, 0, DEV_CNT, DEV_NAME);
13
14    led_chrdev_class = class_create(THIS_MODULE, "led_chrdev");
15
16    for (; i < DEV_CNT; i++) {
17        cdev_init(&led_cdev[i].dev, &led_chrdev_fops);
18        led_cdev[i].dev.owner = THIS_MODULE;
19
```

株式会社日昇テクノロジー

```
20        cur_dev = MKDEV(MAJOR(devno), MINOR(devno) + i);
21
22        cdev_add(&led_cdev[i].dev, cur_dev, 1);
23
24        device_create(led_chrdev_class, NULL, cur_dev, NULL, DEV_NAME "%d", i);
25    }
26
27    return 0;
28 }
29 module_init(led_chrdev_init);
30
31 static __exit void led_chrdev_exit(void)
32 {
33
34    int i;
35    dev_t cur_dev;
36    printk("led chrdev exit (lubancat4  GPIO1_C7)¥n");
37
38    for (i = 0; i < DEV_CNT; i++) {
39        iounmap(led_cdev[i].va_dr); // データレジスタの仮想アドレスを解放
40        iounmap(led_cdev[i].va_ddr); // データ方向レジスタの仮想アドレスを解放
41    }
```

株式会社日昇テクノロジー

```
42
43    for (i = 0; i < DEV_CNT; i++) {
44        cur_dev = MKDEV(MAJOR(devno), MINOR(devno) + i);
45
46        device_destroy(led_chrdev_class, cur_dev);
47
48        cdev_del(&led_cdev[i].dev);
49    }
50
51    unregister_chrdev_region(devno, DEV_CNT);
52    class_destroy(led_chrdev_class);
53 }
54 module_exit(led_chrdev_exit);
```

- 9-10 行：LED ライト構造体メンバの初期化で、物理レジスタアドレスを仮想アドレス空間にマッピングします。

- 12 行：動的にデバイス番号を割り当てます。

- 14 行：デバイスクラスを作成します。

- 17 行：led_cdev と led_chrdev_fops を関連付けます。

- 22 行：デバイスを登録します。

- 24 行：デバイスを作成します。

- 29 行：モジュールのロード処理です。

- 37-38 行：`init`関数で割り当てた仮想アドレス空間を解放します。

株式会社日昇テクノロジー

- 42 行：デバイス番号を計算します。

- 44 行：デバイスを削除します。

5.4.2.4 file_operations 構造体のメンバ関数の実装

リスト 9: file_operations の open 関数の実装

```
1 static int led_chrdev_open(struct inode *inode, struct file *filp)
2 {
3    unsigned int val = 0;
4    struct led_chrdev *led_cdev = (struct led_chrdev *)container_of(inode->i_cdev, struct led_chrdev,
dev);
5    filp->private_data = container_of(inode->i_cdev, struct led_chrdev, dev);
6
7    printk("open¥n");
8
9    // 出力モードを設定
10    val = ioread32(led_cdev->va_ddr);
11    val |= ((unsigned int)0x1 << (led_cdev->led_pin+16));
12    val |= ((unsigned int)0x1 << (led_cdev->led_pin));
13    iowrite32(val, led_cdev->va_ddr);
14
15    // 高電位を出力
16    val = ioread32(led_cdev->va_dr);
17    val |= ((unsigned int)0x1 << (led_cdev->led_pin+16));
```

株式会社日昇テクノロジー

```
18    val |= ((unsigned int)0x1 << (led_cdev->led_pin));
19    iowrite32(val, led_cdev->va_dr);
20
21    return 0;
22 }
```

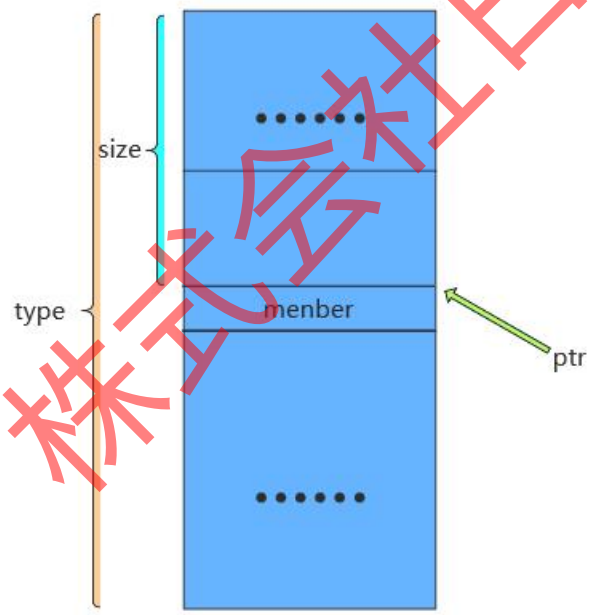
- 4 行目：led_chrdev 構造体変数の dev メンバのアドレスを使用して、この構造体変数の先頭アドレスを見つけます。

- 5 行目：ファイルのプライベートデータ private_data をデバイス構造体 led_cdev にポイントさせます。

- 10-19 行：レジスタの設定を行います。

file_operations の open 関数の実装は非常に重要です。以下にこの関数が具体的に何を行っているかを詳しく分析します。

1. container_of()関数:



Linux ドライバプログラミングにおいて、container_of()関数は頻繁に利用されますので、特に皆さんと共有したいと思います。実際にこの関数の機能は多くありませんが、もし一人でカーネルソース

株式会社日昇テクノロジー

コードを読み解くとなると、理解するのが非常に難しいかもしれません。カーネルソースコードを分析するには、良い知識の蓄積が必要です。以下に container_of()関数の大まかな動作内容を簡単に説明します。そのマクロ定義は以下のようになっています：

リスト 10: container_of()関数（カーネルソースコード /driver/gpu/drm/mkregtable.c に位置している）

```
1 #define container_of(ptr, type, member) ({                ¥
2    const typeof( ((type *)0)->member ) *__mptr = (ptr);    ¥
3    (type *)( (char *)__mptr - offsetof(type,member) );})
```

- **パラメータ**：
  - ptr：構造体変数のあるメンバのアドレス
  - type：構造体のタイプ
  - member：その構造体変数の具体的な名前
- **戻り値**：構造体 type の先頭アドレス

原理は実際には非常にシンプルで、既知の型 type のメンバ member のアドレス ptr を通じて、構造体 type の先頭アドレスを計算します。type の先頭アドレス = ptr - size で、サイズはバイト単位で計算されることに注意が必要です。container_of()関数は以下のようになります：

- ptr と member が同一の型であるかを判断する
- size の大きさを計算し、構造体の開始アドレス = (type *)((char *)ptr - size) （注：その構造体のポインタに強制変換する）

この関数を通じて、私たちは簡単に led_chrdev 構造体の先頭アドレスを取得できます。

2. ファイルプライベートデータ:

多くの Linux ドライバでは、ファイルのプライベートデータ private_data をデバイス構造体にポイントさせ、ユーザー定義のデバイス構造体のアドレスを保存します。private_data に保存されたア

株式会社日昇テクノロジー

ドレスを使用して、読み書きなどの操作でデバイス構造体のメンバにアクセスできます。これにより、Linux のオブジェクト指向のプログラミング思想が反映されています。

3. ioremap()関数によるアドレスのマッピング:

以前にも ioremap()関数について分析しましたが、led_chrdev_open()関数での使用法は同様です。LED ライトに使用される時計制御レジスタのアドレスをマッピングすることにより、プログラム内の仮想アドレスを操作して物理レジスタを間接的に制御することができます。ドライバプログラムでレジスタを記述するのは柔軟性に欠けるため、後のセクションでデバイスツリー（デバイスツリープラグイン）を使用してレジスタとその関連属性を記述する方法を紹介します。

4. ioread32()と iowrite32()関数によるレジスタ操作:

STM32 と同様に、I/O ピンの時計を有効にし、そのポートの多機能設定（この場合は GPIO として）、電気的属性、入出力方向、および出力電圧の高低などを設定します。レジスタにアクセスする際には、まずそのアドレスのデータを変数に読み込み、変数を変更した後、その変数を元のアドレスに書き戻します。このような操作を行う際には、Linux が提供する I/O アクセス関数（iowrite8()、iowrite16()、iowrite32()、ioread8()、ioread16()、ioread32()など）を使用します。ここで、直接仮想アドレスを操作できる理論上の可能性にもかかわらず、Linux はそのような操作を推奨しません。。

次に、file_operations 構造体内の write 関数の実装を分析します。

リスト 11: file_operations 内の write 関数の実装

```
1 static ssize_t led_chrdev_write(struct file *filp, const char __user *buf,
2 size_t count, loff_t *ppos)
3 {
4     unsigned long val = 0;
5     char ret = 0;
6     struct led_chrdev *led_cdev = (struct led_chrdev *)filp->private_data;
```

株式会社日昇テクノロジー

```
7
8    get_user(ret, buf);
9    val = ioread32(led_cdev->va_dr);
10    if (ret == '0') {
11        val |= ((unsigned int)0x01 << (led_cdev->led_pin+16));
12        val &= ~((unsigned int)0x01 << (led_cdev->led_pin)); // GPIO ピンを低電圧出力に設定
13    } else {
14        val |= ((unsigned int)0x01 << (led_cdev->led_pin+16));
15        val |= ((unsigned int)0x01 << (led_cdev->led_pin)); // GPIO ピンを高電圧出力に設定
16    }
17    iowrite32(val, led_cdev->va_dr);
18
19    return count;
20 }
```

- 6 行目：ファイルのプライベートデータアドレスを led_cdev 構造体ポインタに割り当てます。

- 8 行目：ユーザースペースのバッファからカーネルスペースにデータをコピーします。

- 9 行目：間接的にデータレジスタからデータを読み取ります。

- 18 行目：データをレジスタに再度書き込み、LED の点灯/消灯を制御します。

最後に、file_operations 構造体内の release 関数の実装を分析します。ユーザープロセスがデバイスを閉じる close() システムコールを実行するとき、カーネルはドライバの release()関数を呼び出します。release 関数の主な任務は、未完了の入出力操作をクリーンアップし、リソースを解放し、ユーザー定義の排他フラグをリセットすることです。前述のように、ioremap()を使用して物理アドレス空

株式会社日昇テクノロジー

間を仮想アドレス空間にマッピングしましたが、その仮想アドレス空間を使用し終わったら、iounmap()関数を使用して解放する必要があります。ただし、この解放処理はドライバモジュールの終了時に行われるため、ここでは操作は行いません。

リスト 12: file_operations 内の release 関数の実装

```
1 static int led_chrdev_release(struct inode *inode, struct file *filp)
2 {
3     return 0;
4 }
```

5.4.2.5 LED ドライバの完全なコード

ここまででコードの分析は完了しました。以下はこのドライバの完全なコードです。

リスト 13: led_cdev.c の完全なコード

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3 #include <linux/cdev.h>
4 #include <linux/fs.h>
5 #include <linux/uaccess.h>
6 #include <linux/io.h>
7
8 #define DEV_NAME "led_chrdev"
9 #define DEV_CNT (1)
10
11 #define GPIO1_BASE (0xFD5F8000) //GPIO1 のベースアドレス
```

株式会社日昇テクノロジー

```
12
13 //32 ビットのレジスタで、上位 16 ビットはライトイネーブルビットで、下位 16 ビットのライトイネーブルを制御します；下位 16 ビットは 16 ピンに対応し、ピンの出力レベルを制御します
14 #define GPIO1_DR_L (GPIO1_BASE + 0x0000) // GPIO1 の下位 16 ピンのデータレジスタアドレス
15 #define GPIO1_DR_H (GPIO1_BASE + 0x0004) // GPIO1 の上位 16 ピンのデータレジスタアドレス
16
17 //32 ビットのレジスタで、上位 16 ビットはライトイネーブルビットで、下位 16 ビットのライトイネーブルを制御します；下位 16 ビットは 16 ピンに対応し、ピンの入出力モードを制御します
18 #define GPIO1_DDR_L (GPIO1_BASE + 0x0008) // GPIO1 の下位 16 ピンのデータ方向レジスタアドレス
19 #define GPIO1_DDR_H (GPIO1_BASE + 0x000C) // GPIO1 の上位 16 ピンのデータ方向レジスタアドレス
20
21 static dev_t devno;
22 struct class *led_chrdev_class;
23
24 struct led_chrdev {
25 struct cdev dev;
26 unsigned int __iomem *va_dr;
27 unsigned int __iomem *va_ddr;
28
```

株式会社日昇テクノロジー

```
29 unsigned int led_pin; // ピン
30 };
31
32 static int led_chrdev_open(struct inode *inode, struct file *filp)
33 {
34 unsigned int val = 0;
35 struct led_chrdev *led_cdev = (struct led_chrdev *)container_of(inode->i_cdev, struct led_chrdev,
dev);
36 filp->private_data = container_of(inode->i_cdev, struct led_chrdev, dev);
37
38 printk("open¥n");
39
40 // 出力モード設定
41 val = ioread32(led_cdev->va_ddr);
42 val |= ((unsigned int)0x1 << (led_cdev->led_pin+16));
43 val |= ((unsigned int)0X1 << (led_cdev->led_pin));
44 iowrite32(val,led_cdev->va_ddr);
45
46 //高電圧出力
47 val = ioread32(led_cdev->va_dr);
48 val |= ((unsigned int)0x1 << (led_cdev->led_pin+16));
49 val |= ((unsigned int)0x1 << (led_cdev->led_pin));
```

株式会社日昇テクノロジー

```
50 iowrite32(val, led_cdev->va_dr);

51

52 return 0;

53 }

54

55 static int led_chrdev_release(struct inode *inode, struct file *filp)

56 {

57 return 0;

58 }

59

60 static ssize_t led_chrdev_write(struct file *filp, const char __user * buf, size_t count, loff_t * ppos)

61 {

62 unsigned long val = 0;

63 char ret = 0;

64 struct led_chrdev *led_cdev = (struct led_chrdev *)filp->private_data;

65

66 get_user(ret, buf);

67 val = ioread32(led_cdev->va_dr);

68 if (ret == '0'){

69 val |= ((unsigned int)0x01 << (led_cdev->led_pin+16));

70 val &= ~((unsigned int)0x01 << (led_cdev->led_pin)); /* GPIO ピンを低電圧出力に設定*/

71 }
```

株式会社日昇テクノロジー

```
72 else{
73 val |= ((unsigned int)0x01 << (led_cdev->led_pin+16));
74 val |= ((unsigned int)0x01 << (led_cdev->led_pin)); /* GPIO ピンを高電圧出力に設定*/
75 }
76 iowrite32(val, led_cdev->va_dr);
77
78 return count;
79 }
80
81 static struct file_operations led_chrdev_fops = {
82 .owner = THIS_MODULE,
83 .open = led_chrdev_open,
84 .release = led_chrdev_release,
85 .write = led_chrdev_write,
86 };
87
88 static struct led_chrdev led_cdev[DEV_CNT] = {
89 {.led_pin = 7}, // オフセット、GPIO1_A7 オフセット 7 ビット
90 };
91
92 static __init int led_chrdev_init(void)
93 {
```

株式会社日昇テクノロジー

```
94 int i = 0;
95 dev_t cur_dev;
96 unsigned int val = 0;
97
98 printk("led_chrdev init (lubancat4  GPIO1_C7)¥n");
99
100 led_cdev[0].va_dr = ioremap(GPIO1_DR_H, 4); // データレジスタの物理アドレスを仮想アドレスにマッピング、GPIO1_A7 は GPIO1_DR_H を設定する必要があります
101 led_cdev[0].va_ddr = ioremap(GPIO1_DDR_H, 4); // データ方向レジスタの物理アドレスを仮想アドレスにマッピング、GPIO1_A7 は GPIO1_DDR_H を設定する必要があります
102
103 alloc_chrdev_region(&devno, 0, DEV_CNT, DEV_NAME);
104
105 led_chrdev_class = class_create(THIS_MODULE, "led_chrdev");
106
107 for (; i < DEV_CNT; i++) {
108 cdev_init(&led_cdev[i].dev, &led_chrdev_fops);
109 led_cdev[i].dev.owner = THIS_MODULE;
110
111 cur_dev = MKDEV(MAJOR(devno), MINOR(devno) + i);
112
113 cdev_add(&led_cdev[i].dev, cur_dev, 1);
```

株式会社日昇テクノロジー

```
114
115 device_create(led_chrdev_class, NULL, cur_dev, NULL, DEV_NAME "%d", i);
116 }
117
118 return 0;
119 }
120
121 module_init(led_chrdev_init);
122
123 static __exit void led_chrdev_exit(void)
124 {
125 int i;
126 dev_t cur_dev;
127 printk("led chrdev exit (lubancat4  GPIO1_C7)¥n");
128
129 for (i = 0; i < DEV_CNT; i++) {
130 iounmap(led_cdev[i].va_dr); // データレジスタの仮想アドレスを解放
131 iounmap(led_cdev[i].va_ddr); // データ方向レジスタの仮想アドレスを解放
132 }
133
134 for (i = 0; i < DEV_CNT; i++) {
135 cur_dev = MKDEV(MAJOR(devno), MINOR(devno) + i);
```

株式会社日昇テクノロジー

```
136

137 device_destroy(led_chrdev_class, cur_dev);

138

139 cdev_del(&led_cdev[i].dev);

140

141 }

142 unregister_chrdev_region(devno, DEV_CNT);

143 class_destroy(led_chrdev_class);

144

145 }

146

147 module_exit(led_chrdev_exit);

148

149 MODULE_AUTHOR("Embedfire");

150 MODULE_LICENSE("GPL");
```

他のピンに変更する場合は、上記のハイライトされた部分のみを変更する必要があります。RK3588 ハートビート LED は GPIO4_A2 になります。Rockchip_RK3588_TRM_Part1 マニュアルで GPIO4 のベースアドレスとオフセットを確認し、コードを以下のように変更します。

リスト 14: GPIO4_A2 への変更

```
1 ......

2

3 #define GPIO4_BASE (0xFD5F8000) //GPIO4 の基本アドレス
```

株式会社日昇テクノロジー

```
4
5 //32 ビットのレジスタで、上位 16 ビットは書き込み有効ビットで、下位 16 ビットの書き込みを
制御します；下位 16 ビットは 16 個の
ピンに対応し、ピンの出力レベルを制御します
6 #define GPIO4_DR_L (GPIO4_BASE + 0x0000) // GPIO4 の下位 16 ビットピンのデータレジス
タアドレス
7 #define GPIO4_DR_H (GPIO4_BASE + 0x0004) // GPIO4 の上位 16 ビットピンのデータレジス
タアドレス
8
9 //32 ビットのレジスタで、上位 16 ビットは書き込み有効ビットで、下位 16 ビットの書き込みを
制御します；下位 16 ビットは 16 個のピンに対応し、ピンの入出力モードを制御します
10 #define GPIO4_DDR_L (GPIO4_BASE + 0x0008) // GPIO4 の下位 16 ビットピンのデータ方向
レジスタアドレス
11 #define GPIO4_DDR_H (GPIO4_BASE + 0x000C) // GPIO4 の上位 16 ビットピンのデータ方向
レジスタアドレス
12
13 .....
14
15 static struct led_chrdev led_cdev[DEV_CNT] = {
16 {.led_pin = 13}, // オフセット、GPIO4_A2 はオフセット 13 ビット目、すなわち 8+5 ビット
17 };
18
```

株式会社日昇テクノロジー

```
19 ......
20
21 led_cdev[0].va_dr = ioremap(GPIO4_DR_L, 4); // データレジスタの物理アドレスを仮想アドレスにマッピング、GPIO4_A2 は下位 32 ピンにあるため、GPIO4_DR_L を設定する必要があります
22 led_cdev[0].va_ddr = ioremap(GPIO4_DDR_L, 4); // データ方向レジスタの物理アドレスを仮想アドレスにマッピング、GPIO4_A2 は下位 32 ピンにあるため、GPIO4_DDR_L を設定する必要があります
23
24 ......
```

5.4.3 実験準備

Permission denied や類似のメッセージが表示された場合は、ユーザー権限に注意してください。ほとんどのハードウェア外部デバイス操作機能は root ユーザー権限が必要です。簡単な解決策は、コマンド実行前に sudo を追加するか、root ユーザーでプログラムを実行します。

5.4.3.1 LED ドライバ Makefile

リスト 15: Makefile

```
1 KERNEL_DIR=../../kernel/
2 ARCH=arm64
3 CROSS_COMPILE=aarch64-linux-gnu-
4 export ARCH CROSS_COMPILE
5
6 obj-m := led_cdev.o
```

株式会社日昇テクノロジー

```
7
8 all:
9 $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) modules
10
11 .PHONE:clean copy
12
13 clean:
14 $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) clean
```

Makefile は前のものと大差ありません。obj-m オブジェクトを led_cdev.o に変更するだけです。ここでは詳しく説明しません。

5.4.3.2 コンパイルコマンドの説明

実験ディレクトリで以下のコマンドを入力してドライバーモジュールをコンパイルします：

```
make
```

コンパイルが成功すると、実験ディレクトリに"led_cdev.ko"のドライバーモジュールファイルと"led_cdev_test"のアプリケーションが生成されます。

```
dev@ubuntu_118:~/rk/linux_driver/led_cdev$ make
make -C ../../kernel/build M=/home/dev/rk/linux_driver/led_cdev modules
make[1]: Entering directory '/home/dev/rk/kernel/build'
  CC [M]  /home/dev/rk/linux_driver/led_cdev/led_cdev.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/dev/rk/linux_driver/led_cdev/led_cdev.mod.o
  LD [M]  /home/dev/rk/linux_driver/led_cdev/led_cdev.ko
make[1]: Leaving directory '/home/dev/rk/kernel/build'
dev@ubuntu_118:~/rk/linux_driver/led_cdev$ ls
led_cdev.c  led_cdev.ko  led_cdev.mod.c  led_cdev.mod.o  led_cdev.o  led_test.c  Makefile  modules.order  Module.symvers
```

5.4.4 プログラム実行結果

この節の実験では、lubancat のボードには、デフォルトで LED のデバイス機能が有効になってい

ます。デバイスツリーの leds ノードを無効にするには、leds ノードの status = "okay"; を status = "disabled";に変更し、デバイスツリーをコンパイルして置き換えるか、ボードで以下のコマンドを直接使用して、システム leds ドライバーによる LED の制御を停止します：

```
sudo sh -c 'echo 0 > /sys/class/leds/sys_status_led/brightness'
```

LED の明るさを 0 にし、同時に LED のトリガー条件を自動的に none に変更し、leds ドライバーによる LED の制御をキャンセルします。

scp または nfs を使用して上記の 2 つのファイルを開発ボードにコピーし、次にコマンドを実行してロードします。ボード上でカーネルモジュールをコンパイルする場合は、直接コマンドを実行してドライバーをロードします：

LED ドライバーのロード：

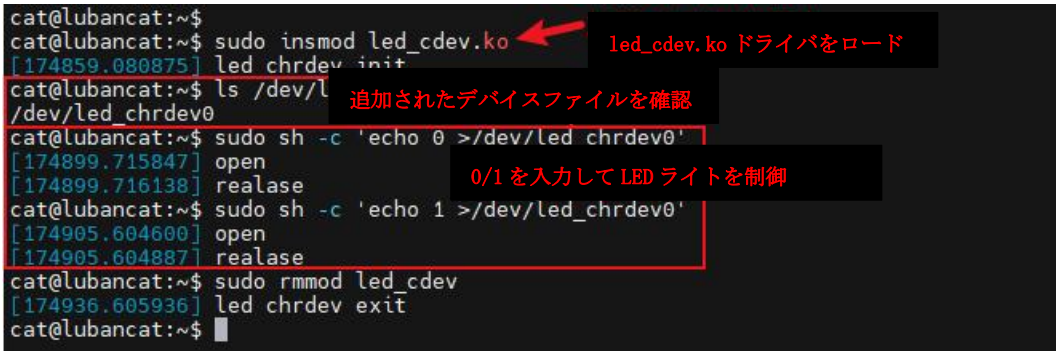
```
sudo insmod led_cdev.ko
```

その後、/dev/ディレクトリで led_chrdev0 というデバイスを見つけることができます。デバイスに直接 1/0 を書き込むことで LED の点灯/消灯を制御することも、テストプログラムを使用して LED を制御することもできます。

```
# 緑のライトを点灯
sudo sh -c 'echo 0 >/dev/led_chrdev0'
# 緑のライトを消灯
sudo sh -c 'echo 1 >/dev/led_chrdev0'
```

ボード上でライトの点灯と消灯が見られます：

株式会社日昇テクノロジー



第 6 章 Linux のデバイスモデル

前に書いたドライバーで、コードを入れるだけの固定パターンがあることがわかりました。その間の大まかな流れは以下のようにまとめることができます：

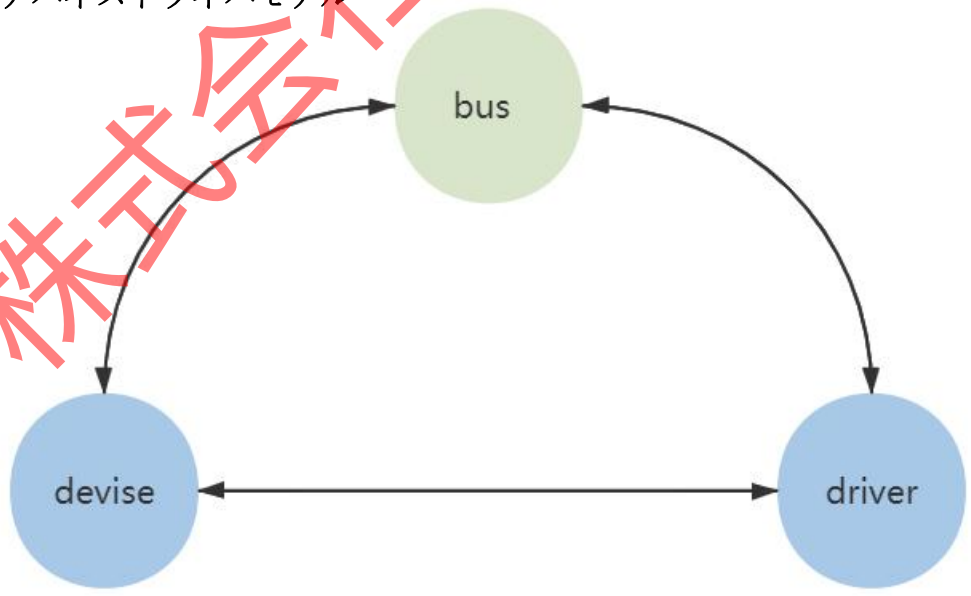
- エントリ関数 xxx_init()とアンロード関数 xxx_exit()を実装します
- デバイス番号を申請します register_chrdev_region()
- 文字デバイスを初期化します、cdev_init 関数、cdev_add 関数
- ハードウェアを初期化します、例えばクロックレジスタの設定有効化、GPIO を入出力モードに設定するなど
- file_operation 構造体の内容を構築し、ハードウェアの各関連操作を実装します
- 端末で mknod を使用してデバイス番号に基づいてデバイスファイル(ノード)を作成するか、自動的に作成します(ドライバーが class_create を使用してデバイスクラスを作成し、そのクラスの下で device_create を使用してデバイスノードを作成します)

したがって、Linux でドライバーを開発する際に、これらの「ルーチン」をマスターすることができれば、ドライバーを開発することは難しくありません。カーネルソースコードの drivers ディレクトリには、多くのデバイスドライバーコードが格納されています。ドライバーを書く前に、この内容を先に確認することをお勧めします。これらのディレクトリで必要なドライバーコードを見つけることができるかもしれません。

株式会社日昇テクノロジー

このようにして、手順に従ってドライバーコードを書くことは簡単ですが、問題があります。ハードウェアの情報をすべてドライバーに書き込んだ場合、特定のハードウェアに対して書かれたドライバーは、ピンインタフェースを少し変更するだけで、このドライバーコードを再度変更しなければならないため、これは明らかに不合理です。では、適切な解決策はあるのでしょうか？答えはもちろんです。Linux はデバイスドライバーモデルの階層概念を導入しました。これにより、書かれたドライバーコードは 2 つの部分に分けられます：デバイスとドライバー。デバイスはハードウェアリソースを提供し、ドライバーコードはこれらのデバイスが提供するハードウェアリソースを使用します。そして、バスがそれらをつなぎ合わせます。これにより、以下の図に示すような関係が形成されます。

デバイスドライバモデル



株式会社日昇テクノロジー

デバイスモデルは、いくつかのデータ構造を通じて、現在のシステム内のバス、デバイス、およびドライバーの動作状況を反映します。以下の重要な概念が提案されています：

- デバイス(device)：特定のバスに取り付けられた物理デバイス；
- ドライバー(driver)：特定のデバイスに関連するソフトウェアで、そのデバイスを初期化し、そのデバイスを操作するいくつかの方法を提供します；
- バス(bus)：特定のバスに取り付けられたデバイスおよびドライバーを管理します；
- クラス(class)：同じ機能を持つデバイスに対して、カテゴリに分類し、管理します；

Linux ではすべてが「ファイル」ということを知っています。ルートファイルシステムには/sys ファイルディレクトリがあり、そこには各デバイス間の関係が記録されています。以下では、/sys 以下のいくつかの重要なディレクトリの役割を紹介します。

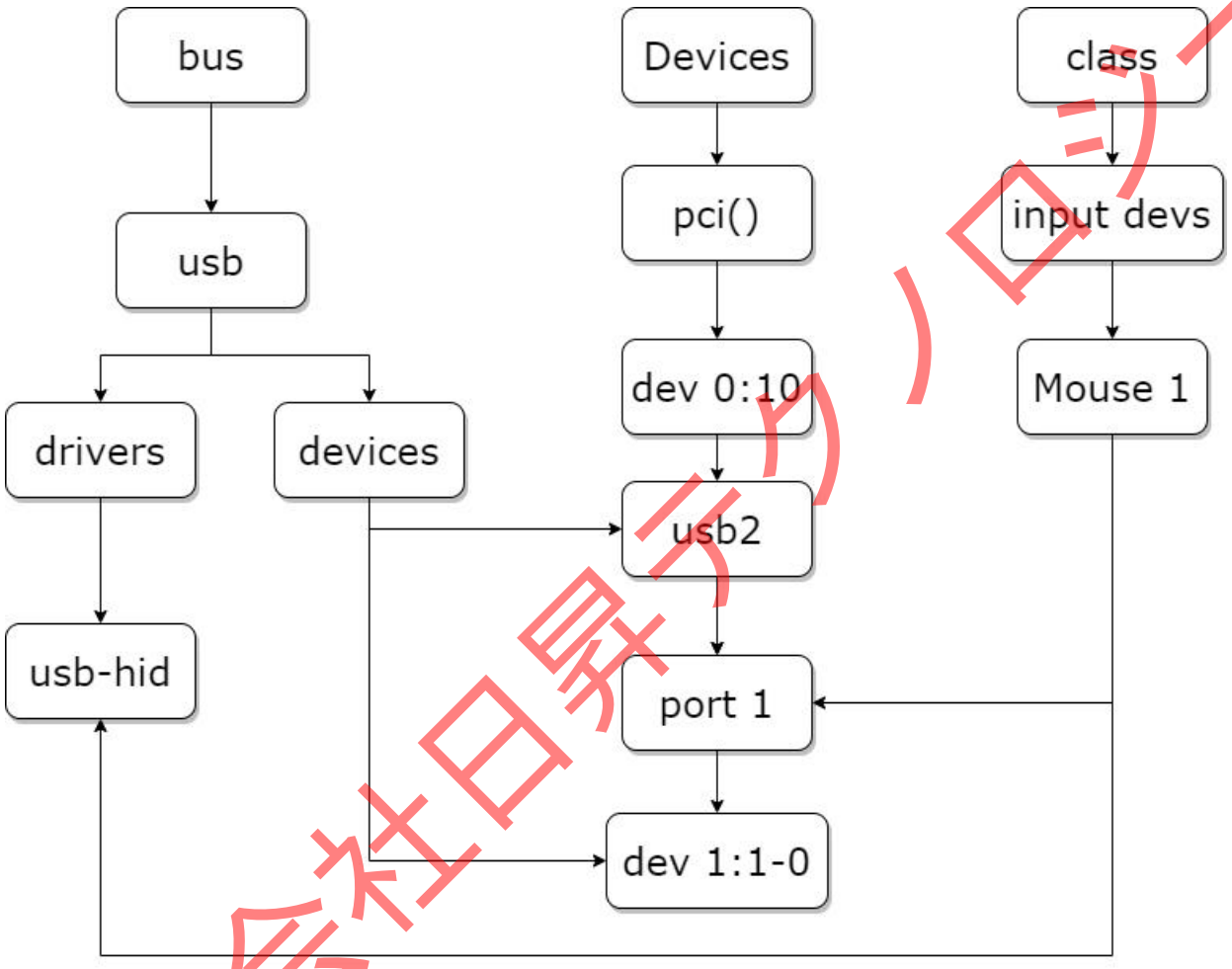
/sys/bus ディレクトリ以下の各サブディレクトリは、登録されたバスタイプです。これは、デバイスがバスタイプに従って階層的に配置されたディレクトリ構造です。各サブディレクトリ（バスタイプ）には、devices と drivers の 2 つのサブディレクトリが含まれています。devices 以下には、そのバスタイプにあるすべてのデバイスがありますが、これらのデバイスはすべてシンボリックリンクで、それぞれが実際のデバイス（/sys/devices/以下）を指しています。bus 以下の usb バスの device は、Devices ディレクトリの/pci()/dev 0:10/usb2 へのシンボリックリンクです。drivers 以下には、このバス上に登録されたすべてのドライバーがあり、各 driver サブディレクトリには、観察および変更可能ないくつかの driver パラメータがあります。

/sys/devices ディレクトリ以下には、各種バス上に登録されたすべての発見された物理デバイスの全体的なデバイス構造が含まれています。一般的に、すべての物理デバイスは、バス上のトポロジ構造に従って表示されます。/sys/devices は、システム内のすべてのデバイスの階層的表現モデルであり、/sys ファイルシステムでデバイスを管理する最も重要なディレクトリ構造です。

/sys/class ディレクトリ以下には、kernel 内に登録されているすべてのデバイスタイプが含まれてい

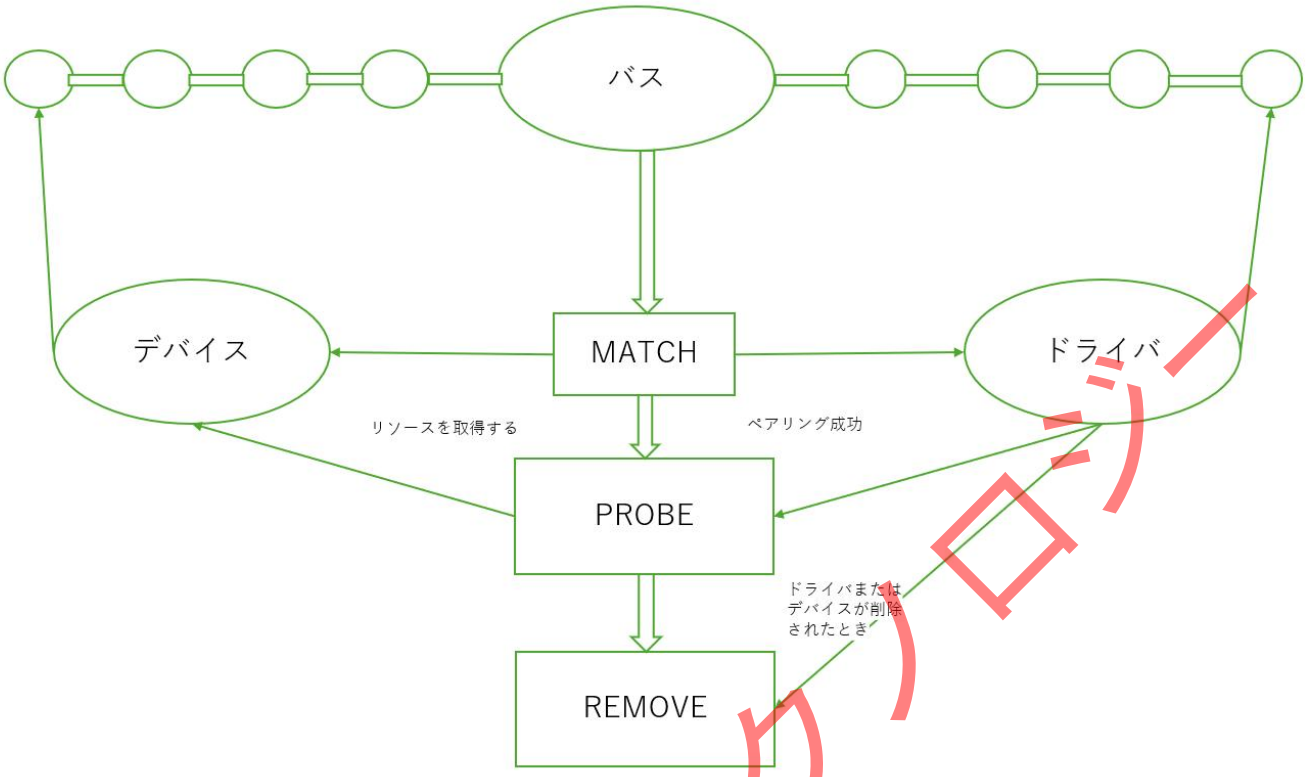
株式会社日昇テクノロジー

ます。これは、デバイス機能によるデバイスの分類モデルで、各種デバイスがそれぞれ特定の機能を持っていることを知っています。例えば、マウスの機能は人間とコンピュータの間の入力として機能します。デバイス機能による分類では、それがどのバスに取り付けられていても、すべてのデバイスは/sys/class/input などのカテゴリに分類されます。



これらを一体化すると、上記のトポロジ図が形成され、デバイス間の関係が記録されます。この章では、bus フォルダディレクトリに焦点を当て、自分のバスタイプや devices、drivers を作成します。上記のデバイス間のトポロジ図を理解した後、"バス-デバイス-ドライバー"モデルに戻りましょう。"バス-デバイス-ドライバー"はどのように協調して動作するのでしょうか？

株式会社日昇テクノロジー



バスは 2 つのリストを管理しており、それぞれがデバイスとドライバーを管理しています。システムにドライバーを登録するときは、新しいドライバーをドライバー管理リストに挿入し、同様にシステムにデバイスを登録するときは、新しいデバイスをデバイス管理リストに挿入します。挿入と同時に、バスは bus_type 構造体内の match メソッドを実行して、新たに挿入されたデバイス/ドライバーをマッチングします。（最も単純なマッチング方法は名前を比較することで、名前が同じデバイス/ドライバーが見つかればマッチングに成功します）。マッチングに成功すると、ドライバーの device_driver 構造体内の probe メソッド（通常、probe 内でデバイスリソースを取得し、具体的な機能はドライバーの開発者によってカスタマイズされます）が呼び出され、デバイスやドライバーを削除する際には、device_driver 構造体内の remove メソッドが呼び出されます。

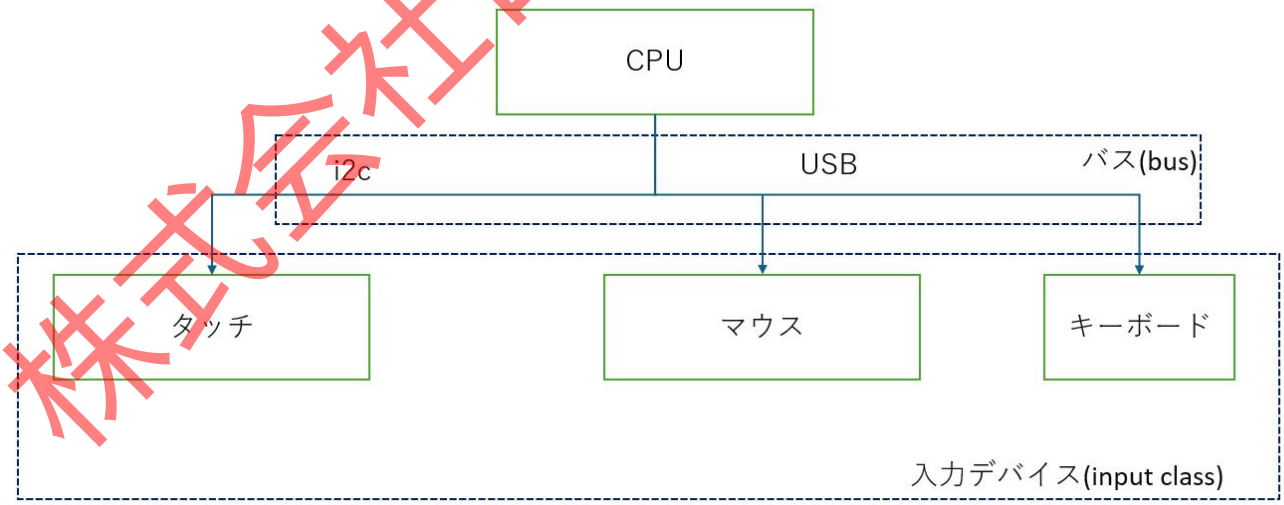
これらはデバイスドライバーモデルのメカニズムに過ぎず、上記の match、probe、remove などのメソッドは、必要な機能を実装するために行う必要があります。ここまで読んで、デバイスドライバーモデルの大まかな全体像を理解できたはずです。今後、プラットフォームデバイスドライバー、ブロックデバイスドライバー、またはその他のバスデバイスを学ぶ際も、すべて Linux デバイスモデル

株式会社日昇テクノロジー

と密接に関連しています。sysfs ファイルシステムは、カーネルのデバイスドライバーをユーザースペースにエクスポートし、ユーザーが sys ディレクトリ及びその下のファイルにアクセスして、カーネルの一部のドライバーデバイスを閲覧または制御できるようにします。次に、バス、ドライバー、デバイスについてさらに理解を深め、コードを使用して自分のバスを作成し、そのバス上にデバイスとドライバーを作成する方法を具体的に学びます。また、ドライバーの特定の制御変数をユーザースペースにエクスポートすることもできます。

6.1 バス

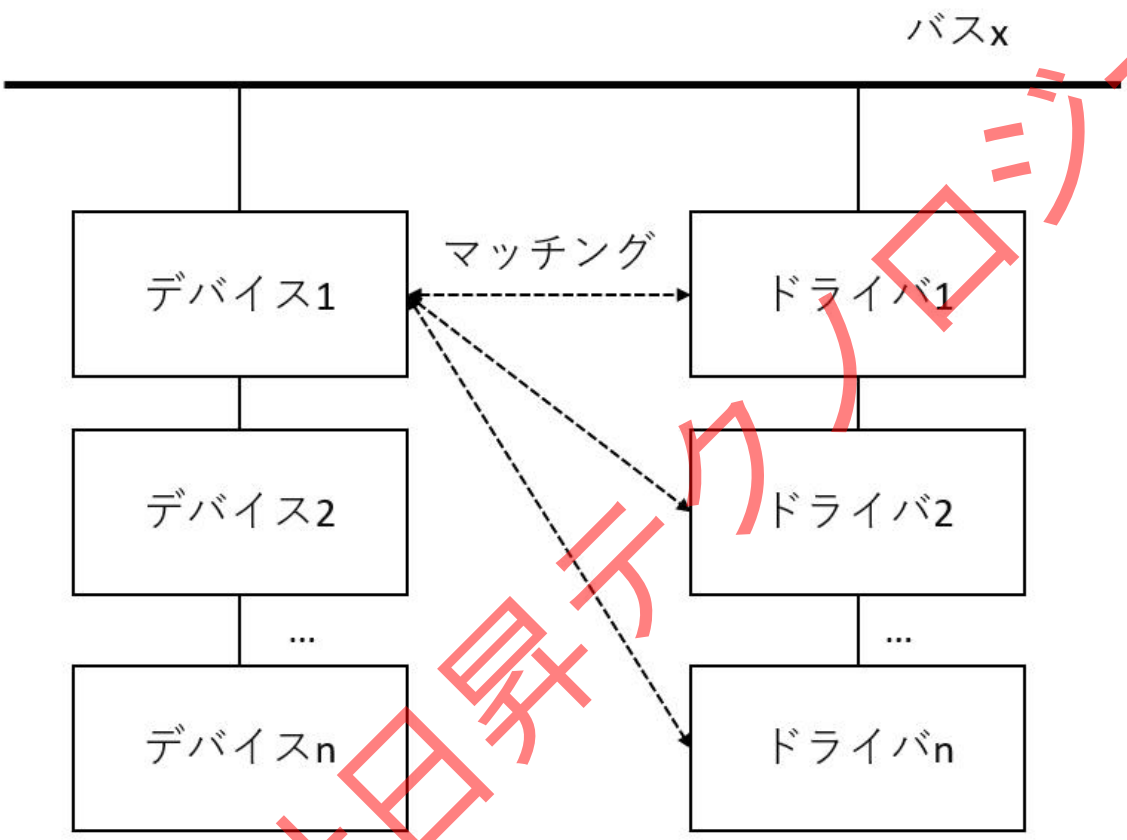
バスは、プロセッサとデバイスの間の橋渡しであり、同じ種類のデバイスが共通の動作タイミングを守る必要があることを意味します。接触するデバイスの大部分はバスを通じて通信しており、物理的な接続は図に示されています。たとえば、公式開発ボードでは、タッチチップは I2C に依存しており、マウスやキーボードなどの HID デバイスは USB に依存しています。機能的には、これらのデバイスは文字、キャラクター、制御コマンド、または収集したデータなどの情報をコンピューターに入力します。



バスドライバーは、バスの各種動作を実現する責任があり、そのバスに追加されたデバイスリストとそのバスに登録されたドライバーリストの 2 つのリストを管理します。バスにデバイス（ドライバ

株式会社日昇テクノロジー

ー）を追加（削除）するとき、対応するリストに新しいノードが追加され、そのバスにマウントされたドライバーおよびデバイスがマッチングされます。マッチングプロセスでは、既にドライバーがマッチングされているデバイスは無視されます。



株式会社日昇テクノロジー

カーネルでは、バスを表すために bus_type 構造体が使用されます。以下に示します：

リスト 1: bus_type 構造体（カーネルソースコード/include/linux/device.h）

```
1 struct bus_type {
2 const char *name;
3 const struct attribute_group **bus_groups;
4 const struct attribute_group **dev_groups;
```

```
5 const struct attribute_group **drv_groups;

6

7 int (*match)(struct device *dev, struct device_driver *drv);

8 int (*uevent)(struct device *dev, struct kobj_uevent_env *env);

9 int (*probe)(struct device *dev);

10 int (*remove)(struct device *dev);

11

12 int (*suspend)(struct device *dev, pm_message_t state);

13 int (*resume)(struct device *dev);

14

15 const struct dev_pm_ops *pm;

16

17 struct subsys_private *p;

18

19 };
```

• name: 新しいバスタイプを登録するとき、/sys/bus ディレクトリに新しいディレクトリが作成され、そのディレクトリ名がこのパラメータの値になります。

• drv_groups、dev_groups、bus_groups: それぞれドライバー、デバイス、バスの属性を表します。これらの属性には内部変数、文字列などがあります。通常、対応する/sys ディレクトリにファイルとして存在し、ドライバーの場合は、ディレクトリ/sys/bus/<bus-name>/driver/<driver-name>にデバイスのデフォルト属性が置かれます。デバイスの場合は、ディレクトリ/sys/bus/<bus-name>/devices/<driver-name>に置かれます。これらのファイルは通常読み書き可能で、ユーザーはこれらの attribute の値を読み書きすることで取得および設定できます。

株式会社日昇テクノロジー

• match: 新しいデバイスや新しいドライバーをバスに登録するときに、このコールバック関数が呼び出されます。この関数は、登録されたドライバーが新しいデバイスに適しているか、または新しいドライバーがバス上でまだマッチしていない登録済みのデバイスをドライブできるかどうかを判断する責任があります。

• uevent: バス上のデバイスに追加、削除、その他のアクションが発生すると、この関数が呼び出され、ドライバーに対応する措置を通知します。

• probe: バスがデバイスとドライバーをマッチングした後、このコールバック関数が実行され、最終的にドライバーが提供する probe 関数が呼び出されます。

• remove: デバイスがバスから削除されるときに、このコールバック関数が呼び出されます。

• suspend、resume: 電源管理の関連関数です。バスがスリープモードに入ると、suspend コールバック関数が呼び出されます。一方、バスがウェイクアップ状態になると、resume コールバック関数が実行されます。

• pm: 電源管理の構造体で、バスの電源管理に関連する一連の関数が含まれています。これは、device_driver 構造体内の pm_ops と関連しています。

• p: この構造体は、特定のプライベートデータを格納するために使用され、そのメンバー klist_devices と klist_drivers は、そのバスにマウントされたデバイスとドライバーを記録しています。

実際に Linux ドライバーモジュールを書く際には、Linux カーネルはすでにほとんどのバスドライバーを提供しているため、通常は新しいバスを登録する必要はありません。カーネルは、バスを登録するための bus_register 関数と、バスを登録解除するための bus_unregister 関数を提供しています。これらの関数のプロトタイプは以下のとおりです：

リスト 2: バスの登録/登録解除 API（カーネルソースコード/drivers/base/bus.c）

```
1 int bus_register(struct bus_type *bus);
```

株式会社日昇テクノロジー

パラメータ： bus: bus_type 型の構造体ポインタ

戻り値：

- 成功： 0
- 失敗：負の数

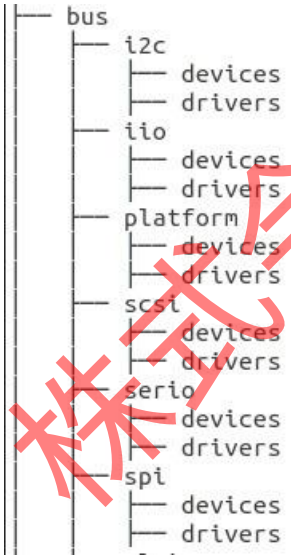
リスト 3: バスの登録/登録解除 API（カーネルソースコード/drivers/base/bus.c）

```
1 void bus_unregister(struct bus_type *bus);
```

パラメータ： bus :bus_type 型の構造体ポインタ

戻り値：なし

バスを成功裏に登録すると、/sys/bus/ディレクトリに新しいディレクトリが作成され、新しく登録されたバス名がディレクトリ名となります。bus ディレクトリには、i2c、spi、platform など、現在システムに登録されているすべてのバスが含まれています。各バスディレクトリには、そのバスにマウントされているすべてのデバイスとドライバーを記録する devices と drivers の 2 つのサブディレクトリがあります。



6.2 デバイス

ドライバー開発のプロセスでは、デバイスと対応するドライバーが最も関心のあるものです。ドライバーを書く目的は、最終的にデバイスが正常に動作するようにすることです。Linux では、すべてが

株式会社日昇テクノロジー

ファイルとして存在しており、デバイスも例外ではありません。/sys/devices ディレクトリには、システム内のすべてのデバイスが記録されており、実際には sys ディレクトリ下のすべてのデバイスファイルが最終的にこのディレクトリの対応するデバイスファイルを指しています。さらに、/sys/dev ディレクトリにはすべてのデバイスノードが記録されていますが、実際にはリンクファイルであり、同様に devices ディレクトリ下のファイルを指しています。

```
/sys/dev
├── block
│   ├── 1:9 -> ../../devices/virtual/block/ram9
│   ├── 7:0 -> ../../devices/virtual/block/loop0
│   ├── 7:1 -> ../../devices/virtual/block/loop1
│   ├── 7:2 -> ../../devices/virtual/block/loop2
│   ├── 7:3 -> ../../devices/virtual/block/loop3
│   ├── 7:4 -> ../../devices/virtual/block/loop4
│   ├── 7:5 -> ../../devices/virtual/block/loop5
│   ├── 7:6 -> ../../devices/virtual/block/loop6
│   └── 7:7 -> ../../devices/virtual/block/loop7
└── char
    ├── 10:130 -> ../../devices/soc0/soc/2000000.aips-bus/20bc000.wdog/misc/watchdog
    ├── 10:183 -> ../../devices/virtual/misc/hw_random
    ├── 10:229 -> ../../devices/virtual/misc/fuse
    ├── 10:235 -> ../../devices/virtual/misc/autofs
    ├── 189:0 -> ../../devices/soc0/soc/2100000.aips-bus/2184200.usb/ci_hdrc.1/usb1
```

カーネルは device 構造体を使用して物理デバイスを記述します。以下に示します（省略あり）：

リスト 4: device 構造体（カーネルソースコード/include/linux/device.h）

```
1 struct device {
2 const char *init_name;
3 struct device *parent;
4 struct bus_type *bus;
5 struct device_driver *driver;
6 void *platform_data;
7 void *driver_data;
8 struct device_node *of_node;
9 dev_t devt;
```

株式会社日昇テクノロジー

```
10 struct class *class;

11 void (*release)(struct device *dev);

12 const struct attribute_group **groups; /* optional groups */

13 struct device_private *p;

14 .....

15 };
```

• init_name: そのデバイスの名前を指定します。バスマッチ時には、名前を比較してペアリングを行うことが一般的です。

• parent: そのデバイスの親オブジェクトを表します。以前は、デバイス間には関連性がありませんでしたが、Linux デバイスモデルの導入により、デバイス間が木構造を形成し、さまざまなデバイスの管理が容易になりました。

• bus: そのデバイスが依存するバスを表します。デバイスを登録する際に、カーネルはそのデバイスを対応するバスに登録します。

• of_node: デバイスツリー内のマッチしたデバイスノードを格納します。カーネルがデバイスツリーを有効にすると、バスはドライバーの of_match_table とデバイスツリーの compatible 属性を比較した後、マッチしたノードをこの変数に保存します。

• platform_data: プラットフォームに関連するデータを保存するポインタです。具体的なドライバーモジュールは、一部のプライベートデータをここに一時的に保存し、必要なときに取り出すことができます。したがって、デバイスモデルはこのポインタの実際の意味を気にしません。

• driver_data: 上記と同様、ドライバ層は dev_set/get_drvdata 関数を通じてこのメンバーを取得できます。

• class: そのデバイスが属するクラスを指します。冒頭で触れたタッチ、マウス、キーボードなどのデバイスは、コンピューターにとって同じ機能を持ち、入力デバイスに分類されます。

株式会社日昇テクノロジー

/sys/class ディレクトリの対応するクラスでそのデバイスを見つけることができます（例: input、leds、pwm など）。

• dev: dev_t 型の変数で、デバイスを識別するデバイス番号です。この変数は主に、/sys ディレクトリに対応するデバイスをエクスポートするために使用されます。

• release: デバイスが登録解除されるときに呼び出されるコールバック関数です。この関数が定義されていない場合、デバイスを削除すると「Device'xxxx'does not have a release() function, it is broken and must be fixed」というエラーが表示されます。

• group: struct attribute_group 型のポインタで、そのデバイスの属性を指定します。

• **p**: プライベートデータ構造のポインタで、このポインタにはサブデバイスリスト、bus/driver/parent などのデバイスに追加されるリストヘッドなどが保存されます。

カーネルはデバイスの登録および登録解除のための API も提供しています。以下に示します：

リスト 5: カーネルによるデバイスの登録/登録解除（カーネルソースコード/driver/base/core.c）

```
1 int device_register(struct device *dev);
```

パラメータ: dev :struct device 構造体のポインタ

戻り値：

- 成功: 0

- 失敗: 負の数

リスト 6: カーネルによるデバイスの登録/登録解除（カーネルソースコード/driver/base/core.c）

```
1 void device_unregister(struct device *dev);
```

パラメータ: dev :struct device 構造体のポインタ

戻り値: なし

バスについて話したとき、成功したバスの登録は/sys/bus ディレクトリに該当するバスのディレクトリを作成し、その下に drivers と devices の 2 つのサブディレクトリが作成されると説明しました。

株式会社日昇テクノロジー

device_register で登録されたデバイスが特定のバスに属している場合、そのバスの devices ディレクトリにそのデバイスファイルが存在します。

6.3 ドライバー

前の 2 節で、バスとデバイスについて大まかに説明しました。デバイスが正常に動作するかどうかは、ドライバーに依存します。ドライバーはカーネルに対して、どのデバイスを動かせるか、デバイスをどのように初期化するかを伝える必要があります。カーネルでは、device_driver 構造体を使用してドライバーを記述します。以下に示します：

リスト 7: device_driver 構造体（カーネルソースコード/include/linux/device.h）

```
1 struct device_driver {
2 const char *name;
3 struct bus_type *bus;
4
5 struct module *owner;
6 const char *mod_name; /* used for built-in modules */
7
8 bool suppress_bind_attrs; /* disables bind/unbind via sysfs */
9
10 const struct of_device_id *of_match_table;
11 const struct acpi_device_id *acpi_match_table;
12
13 int (*probe) (struct device *dev);
14 int (*remove) (struct device *dev);
```

株式会社日昇テクノロジー

```
15
16 const struct attribute_group **groups;
17 struct driver_private *p;
18
19 };
```

- name: ドライバー名を指定し、バスがマッチングする際に、このメンバをデバイス名と比較します。

- bus: そのドライバーが依存するバスを示します。カーネルはドライバーが実行される前に、対応するバスが正常に動作していることを保証する必要があります。

- suppress_bind_attrs: Boolean 値で、sysfs を通じて bind および unbind ファイルをエクスポートするかどうかを指定します。bind および unbind ファイルは、ドライバーが関連するデバイスをバインド/アンバインドするために使用されます。

- owner: そのドライバーの所有者で、通常は THIS_MODULE に設定されます。

- of_match_table: そのドライバーがサポートするデバイスタイプを指定します。カーネルがデバイスツリーを有効にする場合、このメンバを使用してデバイスツリー内の compatible 属性と比較します。

- remove: デバイスがオペレーティングシステムから取り外された場合やシステムが再起動された場合に、このコールバック関数が呼び出されます。

- probe: ドライバーとデバイスがマッチした後、このコールバック関数が実行され、デバイスの初期化が行われます。通常のコードでは main 関数から実行が開始されますが、カーネルのドライバーコードでは probe 関数から開始されます。

- group: struct attribute_group 型のポインタを指し、そのドライバーの属性を指定します。

カーネルは driver_register 関数および driver_unregister 関数を提供してドライバーの登録/登録解

株式会社日昇テクノロジー

除を行います。成功した登録は/sys/bus/<bus>/drivers ディレクトリに記録されます。関数のプロトタイプは以下のとおりです：

リスト 8: device_driver 構造体（カーネルソースコード/include/linux/device.h）

```
1 int driver_register(struct device_driver *drv);
```

パラメータ: drv :struct device_driver 構造体のポインタ

戻り値：

- 成功: 0
- 失敗: 負の数

リスト 9: device_driver 構造体（カーネルソースコード/include/linux/device.h）

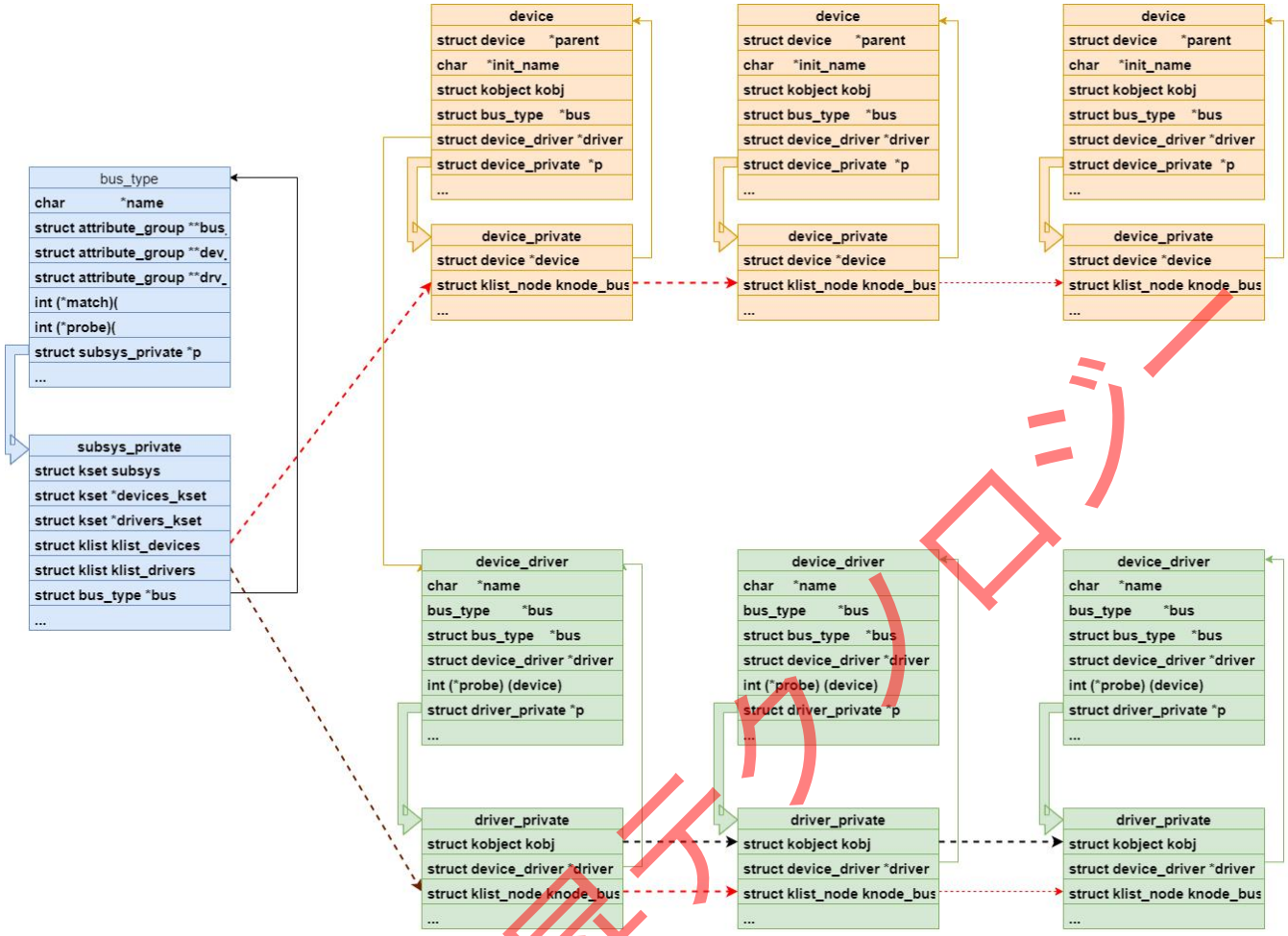
```
1 void driver_unregister(struct device_driver *drv);
```

パラメータ: drv :struct device_driver 構造体のポインタ

戻り値: なし

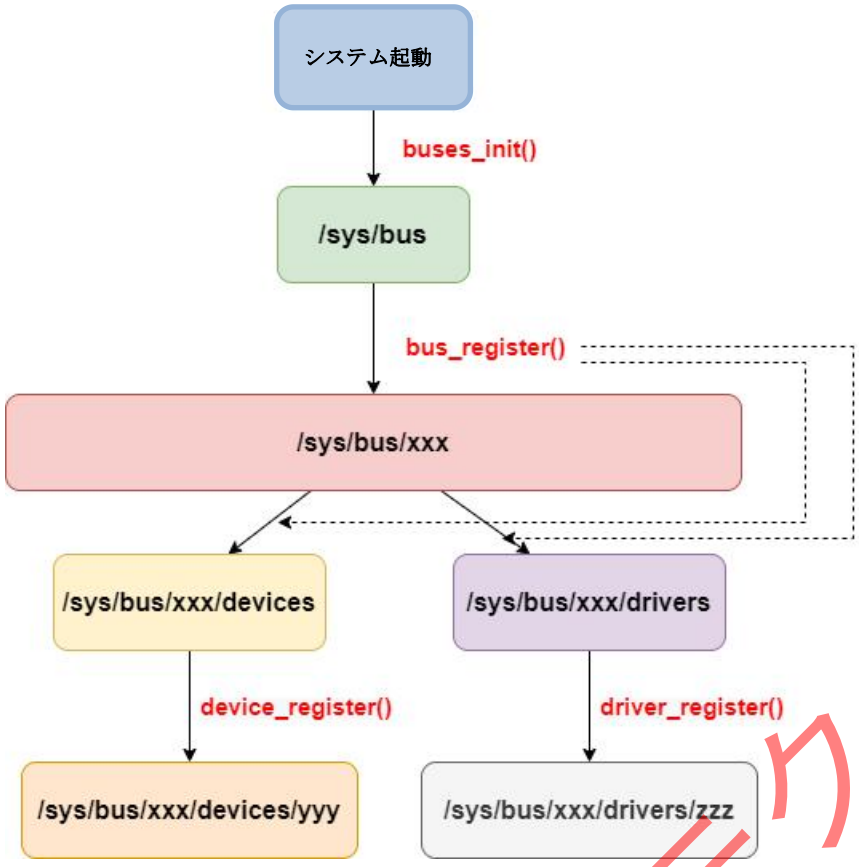
これまでにバス、デバイス、ドライバーのデータ構造および登録/登録解除インターフェース関数を簡単に紹介しました。次の図は、バスにデバイスとドライバーが関連付けられた後のデータ構造の関係図です。

株式会社日昇テクノロジー



大まかな登録プロセスは以下の通りです。

株式会社日昇テクノロジー



システム起動後に buses_init 関数が呼び出され、/sys/bus ファイルディレクトリが作成されます。この部分はシステムが起動時に準備してくれます。次に、bus_register 関数を使用してバスを登録し、バスのディレクトリ下に devices フォルダと drivers フォルダが生成された後、device_register および driver_register 関数を使用して、それぞれのデバイスとドライバーを登録します。

6.4 attribute 属性ファイル

/sys ディレクトリには、さまざまなサブディレクトリやファイルがあります。新しいバス、デバイス、またはドライバを登録すると、カーネルは対応する場所に新しいディレクトリを作成し、そのディレクトリ名はそれぞれの構造体の name メンバになります。各サブディレクトリ内のファイルは、デバイスを制御するためにカーネルがユーザースペースにエクスポートしたものです。カーネルは attribute 構造体を使用して、/sys ディレクトリ下のファイルを記述します。以下に示します：

株式会社日昇テクノロジー

リスト 10: struct attribute 構造体（カーネルソースコード/include/linux/sysfs.h）

```
1 struct attribute {
2     const char *name;
3     umode_t mode;
4 };
```

- name: ファイル名を指定します。

- mode: ファイルの権限を指定します。

bus_type、device、device_driver 構造体には、struct attribute_group というデータ型が含まれています。これは複数の attribute ファイルの集合で、一つ一つ attribute を登録することなく、それらを初期化するために使用されます。

リスト 11: struct attribute_group 構造体（カーネルソースコード/include/linux/sysfs.h）

```
1 struct attribute_group {
2 const char *name;
3 umode_t (*is_visible)(struct kobject *,
4 struct attribute *, int);
5 struct attribute **attrs;
6 struct bin_attribute **bin_attrs;
7 };
```

6.4.1 デバイス属性ファイル

マイコンを開発する際、特定のレジスタの値を読み取りたい場合、新しいコードを追加して再コンパイルする必要があります。しかし、Linux カーネルでは、毎回ソースコードをコンパイルするのは時間と労力の無駄です。そこで、Linux は以下のインターフェースを提供して、デバイス属性ファイル

株式会社日昇テクノロジー

を登録および登録解除します。これらのインターフェースを使用すると、ユーザーレイヤーで直接クエリや変更を行い、カーネルの再コンパイルを避けることができます。

リスト 12: デバイス属性ファイルインターフェイス（カーネルソースコード /include/linux/device.h）

```
1 struct device_attribute {
2 struct attribute attr;
3 ssize_t (*show)(struct device *dev, struct device_attribute *attr,
4 char *buf);
5 ssize_t (*store)(struct device *dev, struct device_attribute *attr,
6 const char *buf, size_t count);
7 };
8
9 #define DEVICE_ATTR(_name, _mode, _show, _store) ¥
10 struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _store)
11 extern int device_create_file(struct device *device,
12 const struct device_attribute *entry);
13 extern void device_remove_file(struct device *dev,
14 const struct device_attribute *attr);
```

• DEVICE_ATTR マクロ定義は、device_attribute 型の変数を定義するために使用されます。## は##の左右にあるタグを連結することを意味します。したがって、変数の名前は dev_attr_プレフィックスを持つべきです。このマクロ定義には_name、_mode、_show、_store の 4 つのパラメータを渡す必要があり、それぞれファイル名、ファイルの権限、show コールバック関数、store コールバック関数を表します。show コールバック関数および store コールバック関数は、それ

株式会社日昇テクノロジー

ぞれユーザーレイヤの cat および echo コマンドに対応しています。/sys ディレクトリ下のあるファイルを cat コマンドで取得するとき、最終的に show コールバック関数が実行されます。echo コマンドを使用すると、store コールバック関数が実行されます。_mode パラメータの値には、S_IRUSR、S_IWUSR、S_IXUSR などのマクロ定義を使用できます。更なるオプションは、ファイル権限に関する読み書きファイルの章を参照してください。

• device_create_file 関数はファイルを作成するために使用されます。これには 2 つのパラメータがあり、第 1 のパラメータはデバイスを示しています。以前に device 構造体について説明したとき、そのメンバーには bus_type 変数があり、デバイスを特定のバスにマウントし、バスの devices サブディレクトリにそのデバイスのディレクトリを作成するために使用されると説明しました。device パラメータは、どのデバイスディレクトリ下にデバイスファイルを作成するかを理解するために使用できます。第二のパラメータは、自分で定義した device_attribute 型の変数です。

• device_remove_file 関数はファイルを削除するために使用されます。ドライバが登録解除されるとき、対応するディレクトリおよびファイルも削除される必要があります。そのパラメータは device_create_file 関数のパラメータと同じです。

6.4.2 ドライバ属性ファイル

ドライバ属性ファイルは、デバイス属性ファイルと同様の役割を持ちますが、関数パラメータが異なります。関数インターフェースは以下の通りです：

リスト 13: ドライバ属性ファイルインターフェース（カーネルソースコード /include/linux/device.h）

```
1 struct driver_attribute {

2 struct attribute attr;
```

株式会社日昇テクノロジー

```
3 ssize_t (*show)(struct device_driver *driver, char *buf);

4 ssize_t (*store)(struct device_driver *driver, const char *buf,

5 size_t count);

6 };

7

8 #define DRIVER_ATTR_RW(_name) ¥

9 struct driver_attribute driver_attr_##_name = __ATTR_RW(_name)

10 #define DRIVER_ATTR_RO(_name) ¥

11 struct driver_attribute driver_attr_##_name = __ATTR_RO(_name)

12 #define DRIVER_ATTR_WO(_name) ¥

13 struct driver_attribute driver_attr_##_name = __ATTR_WO(_name)

14

15 extern int __must_check driver_create_file(struct device_driver *driver,

16 const struct driver_attribute *attr);

17 extern void driver_remove_file(struct device_driver *driver,

18 const struct driver_attribute *attr);
```

• DRIVER_ATTR_RW、DRIVER_ATTR_RO、DRIVER_ATTR_WO マクロ定義は、driver_attribute 型の変数を定義するために使用され、driver_attr_ のプレフィックスが付きます。ファイル権限が異なる点が特徴で、RW サフィックスはファイルが読み書き可能であることを意味し、RO サフィックスはファイルが読み取り専用であることを、WO サフィックスはファイルが書き込み専用であることを意味します。また、DRIVER_ATTR 型のマクロ定義には show および store コールバック関数を設定するパラメータがないことに気づくでしょう。では、これら 2 つのパラメータをどのように設定するのでしょうか？ドライバーコードを書く際には、

株式会社日昇テクノロジー

xxx_store および xxx_show の 2 つの関数を提供し、それらの関数の xxx が DRIVER_ATTR 型のマクロ定義の名前と一致していることを確認するだけです。

• driver_create_file および driver_remove_file 関数は、ファイルを作成および削除するために使用されます。driver_create_file 関数を使用すると、/sys/bus/<bus-name>/drivers/<driver-name>/ディレクトリ下にファイルが作成されます。

6.4.3 バス属性ファイル

同様に、Linux はバス用にも対応する関数インターフェースを提供しています：

リスト 14: バス属性ファイルインターフェース（カーネルソースコード/include/linux/device.h）

```
1 struct bus_attribute {
2 struct attribute attr;
3 ssize_t (*show)(struct bus_type *bus, char *buf);
4 ssize_t (*store)(struct bus_type *bus, const char *buf, size_t count);
5 };
6 #define BUS_ATTR(_name, _mode, _show, _store) ¥
7 struct bus_attribute bus_attr_##_name = __ATTR(_name, _mode, _show, _store)
8 extern int __must_check bus_create_file(struct bus_type *,
9 struct bus_attribute *);
10 extern void bus_remove_file(struct bus_type *, struct bus_attribute *);
```

• BUS_ATTR マクロ定義は、bus_attribute 変数を定義するために使用されます。

• bus_create_file 関数を使用すると、/sys/bus/<bus-name>下に対応するファイルが作成されます。

• bus_remove_file は、そのファイルを削除するために使用されます。

株式会社日昇テクノロジー

6.5 ドライバデバイスモデルコードの作成と説明

デバイスモデルフレームワークの下で、デバイスドライバの開発は非常に簡単です。まず、struct device 型の変数を割り当て、必要な情報を記入して対応するバスに登録します。次に、struct device_driver 型を作成し、必要な情報を記入して登録します。適切なタイミング（ドライバとデバイスがマッチした時）に、ドライバの probe、release などのコールバック関数が呼び出されます。また、実際のプログラミングでは、直接 device や device_driver を使用することは少なく、platform device など、それらに一層のラッパーを加えたものを使用することが多いです。

以下の実験では、学んだ理論知識を使用して、仮想のバス xbus を作成し、それにドライバ xdrv とデバイス xdev をマウントします。この章のサンプルコードディレクトリは、linux_driver/linux_device_model です。

6.5.1 プログラミングアプローチ

1. Makefile の作成

2. バス構造体を宣言し、xbus バスを作成し、デバイスとドライバのマッチングを行う match メソッドを実装

3. デバイス構造体を宣言し、xbus バスにマウント

4. ドライバ構造体を宣言し、xbus バスにマウントし、probe および remove メソッドを実装

5. バス、デバイス、ドライバの属性ファイルをユーザースペースにエクスポート

6.5.2 Makefile

良い仕事をするための最初のステップは、適切なツールを準備することです。プログラムを書き始める前に、Makefile を準備する必要があります。

株式会社日昇テクノロジー

リスト 15: Makefile（../linux_driver/linux_device_model/Makefile に位置）

```
1 KERNEL_DIR=../../kernel/

2

3 ARCH=arm64

4 CROSS_COMPILE=aarch64-linux-gnu-

5 export ARCH CROSS_COMPILE

6

7 obj-m := xdev.o xbus.o xdrv.o

8

9 all:

10 $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) modules

11 modules clean:

12 $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) clean
```

PC でクロスコンパイルを行い、開発ボードにアップロードして使用するか、またはボード上で直接コンパイルして使用します。

6.5.3 バス

6.5.3.1 新しいバスの定義

バス構造体の定義に関する重要な点は、デバイスとドライバをマッチングさせる match コールバック関数です。この関数がなければ、デバイスとドライバはマッチングできません。

株式会社日昇テクノロジー

リスト 16: bus_type 構造体の定義（../linux_driver/linux_device_model/xbus.c に位置）

```
1 int xbus_match(struct device *dev, struct device_driver *drv)
2 {
3 printk("%s-%s¥n",__FILE__, __func__);
4 if(!strncmp(dev_name(dev), drv->name, strlen(drv->name))){
5 printk("dev & drv match¥n");
6 return 1;
7 }
8 return 0;
9 }
10
11 static struct bus_type xbus = {
12 .name = "xbus",
13 .match = xbus_match,
14 };
15 EXPORT_SYMBOL(xbus);
```

- 11-15 行目: xbus という名前のバス構造体を定義しました。バス構造体内で最も重要なのは、デバイスとドライバのマッチングを担う match コールバック関数です。この関数がないと、デバイスとドライバはマッチングできません。

- 1-9 行目: ドライバとデバイスの名前を比較することでマッチングを行い、一致する場合は 1 を、そうでない場合は 0 を返します。

株式会社日昇テクノロジー

6.5.3.2 バス属性ファイルのエクスポート

BUS_ATTR マクロを使用して、独自の変数を/sys ディレクトリにエクスポートし、ユーザーがクエリを行うことができます。

リスト 17: bus_type 構造体の定義（../linux_driver/linux_device_model/xbus.c に位置）

```
1 static char *bus_name = "xbus";
2
3 ssize_t xbus_test_show(struct bus_type *bus, char *buf)
4 {
5 return sprintf(buf, "%s¥n", bus_name);
6 }
7
8 BUS_ATTR(xbus_test, S_IRUSR, xbus_test_show, NULL);
```

- 1 行目: バスの名前を保持する bus_name 変数を定義しました。

- 3-8 行目: show コールバック関数を提供し、ユーザーが cat コマンドを使用してバスの名前をクエリできるようにし、ファイルの権限を所有者のみ読み取り可能に設定します。

6.5.3.3 バスの登録

カーネルドライバコードはカーネルモジュールベースであり、モジュールの初期化関数でバスを登録し、モジュールの終了関数でバスを登録解除します。

株式会社日昇テクノロジー

リスト 18: モジュールの初期化および終了関数（../linux_driver/linux_device_model/xbus.c に位置）

```
1 static __init int xbus_init(void)
2 {
3 printk("xbus init¥n");
4
5 bus_register(&xbus);
6 bus_create_file(&xbus, &bus_attr_xbus_test);
7 return 0;
8 }
9 module_init(xbus_init);
10
11
12 static __exit void xbus_exit(void)
13 {
14 printk("xbus exit¥n");
15 bus_remove_file(&xbus, &bus_attr_xbus_test);
16 bus_unregister(&xbus);
17 }
18 module_exit(xbus_exit);
19
20 MODULE_AUTHOR("embedfire");
21 MODULE_LICENSE("GPL");
```

- 1-9 行目: バスのロード関数を実装し、バスを登録してバスの属性ファイルをエクスポートしま

株式会社日昇テクノロジー

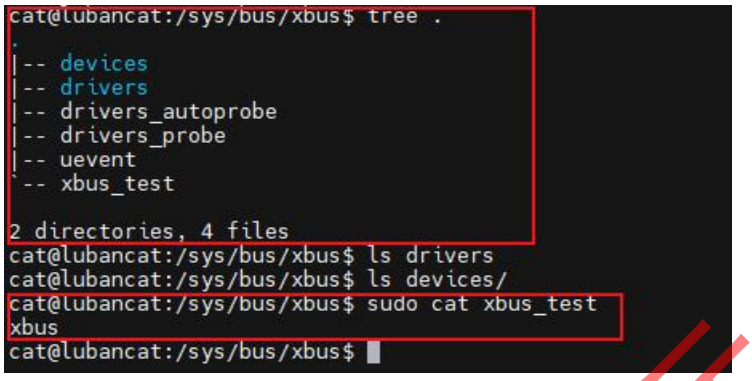
す。

- 11-17 行目: バスのアンロード関数を実装し、バスを登録解除してバスの属性ファイルを削除します。

insmod xbus.ko コマンドを使用して xbus カーネルモジュールをロードする

```
sudo insmod xbus.ko
```

そうすると、カーネルに新しいバス xbus が登場します。



デバイスとドライバのディレクトリにアクセスすると、それらは空であり、このバスには何もマウントされていません。赤枠で囲まれた部分が、カスタマイズしたバス属性ファイルです。「cat xbus_test」コマンドを実行すると、端末に「xbus」という文字列が表示されます。

6.5.4 デバイス

Linux デバイスモデルでは、バスは既に登録されていますが、デバイスとドライバーがまだ不足しています。新しいデバイスを登録する際に主に行う作業は二つあります。一つは名前で、これがマッチングの根拠となります。もう一つは、そのデバイスがどのバスに接続されているかで、適切ではないものを接続してはいけません。

ここでは、デバイス xdev を登録し、変数 id を定義して、この変数をユーザー空間にエクスポートし、ユーザーが sysfs ファイルシステムを通じてこの変数の値を変更できるようにします。

株式会社日昇テクノロジー

6.5.4.1 新しいデバイスの定義

リスト 19: device 構造体の定義(../linux_driver/linux_device_model/xdev.c に位置)

```
1 extern struct bus_type xbus;
2
3 void xdev_release(struct device *dev)
4 {
5 printk("%s-%s¥n", __FILE__, __func__);
6 }
7
8
9 static struct device xdev = {
10 .init_name = "xdev",
11 .bus = &xbus,
12 .release = xdev_release,
13 };
```

- 第 1 行：外部のバス変数 xbus を宣言しています。

- 第 3-6 行：モジュールをアンロードする際にエラーが発生しないように release 関数を記述しています。

- 第 8-12 行：xdev という名前のデバイスを定義し、それを xbus に接続しています。

バスを登録するのに比べて、比較的シンプルです。

株式会社日昇テクノロジー

6.5.4.2 デバイス属性ファイルのエクスポート

リスト 20: デバイス属性ファイルの定義(../linux_driver/linux_device_model/xdev.c に位置)

```
1 unsigned long id = 0;
2 ssize_t xdev_id_show(struct device *dev, struct device_attribute *attr,
3 char *buf)
4 {
5 return sprintf(buf, "%d¥n", id);
6 }
7
8 ssize_t xdev_id_store(struct device *dev, struct device_attribute *attr,
9 const char *buf, size_t count)
10 {
11 kstrtoul(buf, 10, &id);
12 return count;
13 }
14
15
16 DEVICE_ATTR(xdev_id, S_IRUSR|S_IWUSR, xdev_id_show, xdev_id_store);
```

- 第 1-13 行：show コールバック関数では、id の値を sprintf 関数を使用して buf にコピーします。store コールバック関数では、kstrtoul 関数を使用します。この関数には 3 つのパラメータがあり、2 番目のパラメータは何進法を使用するかを示し、ここでは 10 を渡しています。これは、buf の内容が 10 進数に変換され、id に渡されることを意味し、sysfs を通じてドライバ

株式会社日昇テクノロジー

ーを変更する目的を実現しています。

- 第 15 行：DEVICE_ATTR マクロを使用して xdev_id を定義し、このファイルのファイル権限をファイルの所有者が読み書き可能、グループ内のメンバーや他のメンバーは操作できないように設定しています。

- kstrtoul()関数について:

リスト 21: kstrtoul()関数の解析(カーネルソースコード/include/linux/kernel.h 内)

```
1 static inline int __must_check kstrtoul(const char *s, unsigned int base,
2 unsigned long *res)
3 {
4 /*
5 * 関数コールをショートカットしたいが
6 * __builtin_types_compatible_p(unsigned long, unsigned long long) = 0。
7 */
8 if (sizeof(unsigned long) == sizeof(unsigned long long) &&
9 __alignof__(unsigned long) == __alignof__(unsigned long long))
10 return kstrtoull(s, base, (unsigned long long *)res);
11 else
12 return _kstrtoul(s, base, res);
13 }
```

この関数は、文字列を符号なし長整数データに変換する機能を持っています。関数のパラメータと戻り値は以下の通りです：

パラメーター：

- s：文字列の開始アドレスで、この文字列はヌル文字で終わる必要があります。

株式会社日昇テクノロジー

- base：変換の基数で、base が 0 の場合、関数は文字列の型を自動的に判断し、十進数で出力します。たとえば、「0xa」は十進数として扱われ（大文字小文字は同じ）、出力は 10 になります。0 で始まる場合は八進数として解析され、それ以外の場合は小数として解析されます。

- res：変換に成功した結果のアドレスを指すポインターです。

返り値：関数が成功した場合は 0 を返し、オーバーフローした場合は-ERANGE を返し、解析エラーが発生した場合は-EINVAL を返します。

6.5.4.3 デバイスの登録

最後に、device_register 関数と device_create_file 関数を呼び出して、上記のデバイス構造体と属性ファイル構造体をカーネルに登録するだけです。

リスト 22: デバイスの登録/登録解除(../linux_driver/linux_device_model/xdev.c に位置)

```
1 static __init int xdev_init(void)
2 {
3 printk("xdev init¥n");
4 device_register(&xdev);
5 device_create_file(&xdev, &dev_attr_xdev_id);
6 return 0;
7 }
8 module_init(xdev_init);
9
10
11 static __exit void xdev_exit(void)
12 {
```

株式会社日昇テクノロジー

```
13 printk("xdev exit¥n");
14 device_remove_file(&xdev, &dev_attr_xdev_id);
15 device_unregister(&xdev);
16 }
17 module_exit(xdev_exit);
18
19 MODULE_AUTHOR("embedfire");
20 MODULE_LICENSE("GPL");
```

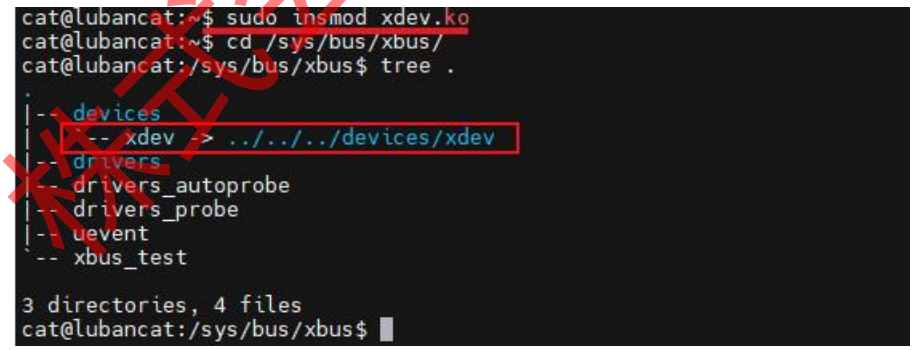
- 第 1-8 行：モジュールの読み込み関数を実装し、デバイスを登録し、デバイス属性ファイルをエクスポートします。

- 第 10-16 行：モジュールのアンロード関数を実装し、デバイスを登録解除し、デバイス属性ファイルを削除します。

insmod コマンドを使用して xdev カーネルモジュールをロードします：

```
sudo insmod xdev.ko
```

カーネルモジュールをロードした後、/sys/bus/xbus/devices/に新しいデバイス xdev が追加されていることがわかります。これはリンクファイルで、最終的に/sys/devices のデバイスを指しています。



直接 xdev のディレクトリに移動すると、カスタマイズした属性ファイル xdev_id が見えます。

echo および cat コマンドを使用して、以下に示すように、変更およびクエリを実行できます：

株式会社日昇テクノロジー

```
cat@lubancat:/sys/bus/xbus$ cd devices/xdev
cat@lubancat:/sys/bus/xbus/devices/xdev$ ls
power  subsystem  uevent  xdev_id
cat@lubancat:/sys/bus/xbus/devices/xdev$ sudo cat xdev_id
0
cat@lubancat:/sys/bus/xbus/devices/xdev$ sudo sh -c "echo 2 > xdev_id"
cat@lubancat:/sys/bus/xbus/devices/xdev$ sudo cat xdev_id
2
cat@lubancat:/sys/bus/xbus/devices/xdev$
```

6.5.5 ドライバー

ドライバーに関しては、この章の実験に具体的な物理デバイスがないため、デバイスの初期化、デバイスの関数インターフェースなどの内容は含まれていません。

6.5.5.1 新しいドライバーの定義

リスト 23: device_driver 構造体の定義(../linux_driver/linux_device_model/xdrv.c に位置)

```
1 extern struct bus_type xbus;
2
3 int xdrv_probe(struct device *dev)
4 {
5 printk("%s-%s¥n", __FILE__, __func__);
6 return 0;
7 }
8
9 int xdrv_remove(struct device *dev)
10 {
11 printk("%s-%s¥n", __FILE__, __func__);
12 return 0;
13 }
```

株式会社日昇テクノロジー

```
14
15 static struct device_driver xdrv = {
16 .name = "xdev",
17 .bus = &xbus,
18 .probe = xdrv_probe,
19 .remove = xdrv_remove,
20 };
```

- 第 1 行：外部のバス変数 xbus を宣言しています。

- 第 3-7 行：ドライバーとデバイスが成功裏にマッチした後、ドライバーの probe 関数が実行され、ここでは現在のファイル名と関数名を端末にプリントしています。

- 第 9-13 行：xdrv_remove 関数は、ドライバーを登録解除する際に、物理デバイスの一部の機能を閉じる必要がある場合がありますが、ここでは現在のファイル名と関数名をプリントしています。

- 第 15-20 行：xdrv という名前のドライバー構造体を定義しています。.name メンバーはデバイスの.name と同じでなければならず、そうでなければマッチングに成功しません。このドライバーは、既に登録されている xbus バスにマウントされます。

株式会社日昇テクノロジー

6.5.5.2 ドライバー属性ファイルのエクスポート

リスト 24: device_driver 構造体の定義(../linux_driver/linux_device_model/xdrv.c に位置)

```
1 char *name = "xdrv";
2 ssize_t drvname_show(struct device_driver *drv, char *buf)
3 {
4 return sprintf(buf, "%s¥n", name);
5 }
6
7 DRIVER_ATTR_RO(drvname);
```

- ドライバー属性ファイルを定義する際に、DRIVER_ATTR_RO を使用してドライバー属性ファイルを定義し、show および store コールバック関数にパラメーターを設定できない場合、store と show 関数のプレフィックスがドライバー属性ファイルと一致している必要があります。コードに示されているように、drvname 属性ファイルを定義し、show コールバック関数の関数名を drvname_show としています。これにより、両者の関連付けが完了します。

6.5.5.3 ドライバーの登録

最後に、driver_register 関数と driver_create_file 関数を呼び出して、ドライバーとドライバー属性ファイルを登録します。

リスト 25: モジュールの登録/登録解除関数(../linux_driver/linux_device_model/xdrv.c に位置)

```
1 static __init int xdrv_init(void)
2 {
3 printk("xdrv init¥n");
4 driver_register(&xdrv);
```

株式会社日昇テクノロジー

```
5 driver_create_file(&xdrv, &driver_attr_drvname);
6 return 0;
7 }
8 module_init(xdrv_init);
9
10 static __exit void xdrv_exit(void)
11 {
12 printk("xdrv exit¥n");
13 driver_remove_file(&xdrv, &driver_attr_drvname);
14 driver_unregister(&xdrv);
15 }
16 module_exit(xdrv_exit);
17
18 MODULE_AUTHOR("embedfire");
19 MODULE_LICENSE("GPL");
```

- 第 1-8 行：モジュールの読み込み関数を実装し、ドライバーを登録し、ドライバー属性ファイルをエクスポートします。

- 第 10-16 行：モジュールのアンロード関数を実装し、ドライバーを登録解除し、ドライバー属性ファイルを削除します。

insmod コマンドを使用して xdrv カーネルモジュールをロードします：

```
sudo insmod xdrv.ko
```

ドライバーを成功裏にロードした後、/sys/bus/xbus/drivers に新しいドライバーディレクトリ xdev が追加されていることがわかります。そのディレクトリ内には、カスタマイズした属性ファイルが存

株式会社日昇テクノロジー

在し、cat コマンドでそのファイルの内容を読むと、ターミナルには「xdrv」という文字列がプリントされます。

```
cat@lubancat:~$ sudo insmod xdrv.ko
cat@lubancat:~$ cd /sys/bus/xbus/
cat@lubancat:/sys/bus/xbus$ tree .
.
|-- devices
|   `-- xdev -> ../../../devices/xdev
|-- drivers
|   `-- xdev
|       |-- bind
|       |-- drvname
|       |-- uevent
|       |-- unbind
|       `-- xdev -> ../../../../devices/xdev
|-- drivers_autoprobe
|-- drivers_probe
|-- uevent
`-- xbus_test

5 directories, 8 files
```

dmesg | tail コマンドを使用して、モジュールのロードプロセスのプリント情報を確認することができます。デバイスとドライバーをロードした後、バスはマッチングを開始し、match 関数を実行します。これら二つのデバイスの名前が一致していることが確認できると、デバイスとドライバーが関連付けられ、最終的にドライバーの probe 関数が実行されます。

```
[  175.306102] xbus init
[  180.618821] xdev init
[  184.539727] xdrv init
[  184.539792] /home/embedfire/workdir/5/linux_device_model/xbus.c-xbus_match
[  184.539798] dev & drv match
[  184.539866] /home/embedfire/workdir/5/linux_device_model/xdrv.c-xdrv_probe
```

第 7 章プラットフォームデバイスドライバ

以前のキャラクターデバイスプログラムのドライバでは、open()関数を呼び出して対応するデバイスファイルを開けば、read()/write()関数を使って file_operations というファイル操作インタフェースを通じてハードウェアを制御できました。このドライバ開発方式はシンプルで直感的ですが、ソフトウェア設計の観点から見ると非常に悪い方法です。

それは、デバイス情報とドライバコードが混在しており、ドライバプログラム内に様々なハードウェアレジスタのアドレスが見られるという深刻な問題があります。本質的に、このドライバ開発方式はマイコンのドライバ開発と大差なく、ハードウェア情報が変更されたりデバイスがなくなったりする

株式会社日昇テクノロジー

と、ドライバソースコードを修正する必要があります。これまでの作業は、単にファイル操作インタフェースの外壳をかぶせただけです。

このようなドライバコードとデバイス情報の結合問題を解決するために、Linux はこれらのデバイスをさらに抽象化し、デバイスドライバモデルを導入しました。前章ではデバイスドライバモデルについて深く分析しましたが、このモデルでは、ドライバコードとデバイス情報を分離するためにバスの概念を導入しました。しかし、ドライバ内のバスの概念はソフトウェアレベルの抽象であり、我々の SOC 内の物理バスの概念とは厳密には一致しません：

- 物理バス：チップと各機能外部デバイス間で情報を転送する公共通信幹線で、データバス、アドレスバス、コントロールバスなどを含み、各種通信タイミングを伝送します。

- ドライババス：デバイスとドライバの管理を担当します。デバイスとドライバのマッチングルールを定め、新しいデバイスやドライバがバスに登録されると、それらのペアリングを試みます。

一般的に、I2C、SPI、USB などの一般的な物理バスについては、Linux カーネルが自動的に対応するドライババスを作成するため、I2C デバイス、SPI デバイス、USB デバイスは自然と対応するバスに登録されます。しかし、実際のプロジェクト開発では、多くのシンプルな構造のデバイスがあり、これらを制御するために特別なタイミングは必要ありません。それらには対応する物理バスがないため、例えば LED、RTC 時計、ブザー、ボタンなどは、Linux カーネルが対応するドライババスを作成しません。このようなデバイスのドライバ開発もデバイスドライバモデルに従うために、Linux カーネルは仮想のバスであるプラットフォームバス(platform bus)を導入しました。

プラットフォームバスは、対応する物理バスがないデバイスの管理、登録を担当し、これらのデバイスはプラットフォームデバイスと呼ばれ、対応するデバイスドライバはプラットフォームドライバと呼ばれます。プラットフォームデバイスドライバの核心は依然として Linux デバイスドライバモデルで、プラットフォームデバイスは platform_device 構造体を使って表現され、この構造体はデバイスドライバモデルの device 構造体を継承しています。また、プラットフォームドライバは

株式会社日昇テクノロジー

platform_driver 構造体を使って表現され、これはデバイスドライバモデルの device_driver 構造体を継承しています。

重点学習：バスのマッチングメカニズム、デバイスドライバとデバイス情報の充填方法、プラットフォームデバイスドライバとキャラクターデバイスの関係理解。

7.1 プラットフォームデバイス

7.1.1 platform_device 構造体

カーネルは platform_device 構造体を使用してプラットフォームデバイスを記述します。構造体の原型は以下の通りです：

リスト 1: platform_device 構造体(カーネルソースコード/include/linux/platform_device.h 内)

```
1 struct platform_device {
2 const char *name;
3 int id;
4 struct device dev;
5 u32 num_resources;
6 struct resource *resource;
7 const struct platform_device_id *id_entry;
8 /* その他のメンバ省略 */
9 };
```

- name：デバイスの名前。バスがデバイスとドライバの名前が一致するか比較する際に使用します；

- id：デバイスの番号を指定します。Linux は同名のデバイスをサポートしており、同名デバイス間ではこの番号で区別されます；

株式会社日昇テクノロジー

- dev：Linux デバイスモデルの device 構造体。Linux カーネルはオブジェクト指向の考え方を大量に使用しており、platform_device はこの構造体を継承することで関連するコードを再利用し、カーネルがプラットフォームデバイスを管理しやすくなります；

- num_resources：リソースの数を記録します。構造体メンバ resource が配列として格納されている場合、resource 配列の数を記録する必要があります。カーネルは ARRAY_SIZE マクロを提供して配列の数を計算します；

- resource：プラットフォームデバイスがドライバに提供するリソース（irq、dma、メモリなど）。この構造体は後ほど説明されます；

- id_entry：プラットフォームバスが提供する別のマッチング方法。原理は文字列を比較することによりますが、この部分の内容はプラットフォームバスの節で説明されます。ここでの id_entry はマッチングの結果を保存するために使用されます；

7.1.2 デバイス情報とは？

プラットフォームデバイスの仕事は、ドライバプログラムにデバイス情報を提供することです。デバイス情報にはハードウェア情報とソフトウェア情報の二つの部分が含まれます。

- ハードウェア情報：どのレジスタを使用するか、どの割り込み番号、メモリリソース、IO ポートなどを占有するかなど

- ソフトウェア情報：イーサネットカードの MAC アドレス、I2C デバイスのデバイスアドレス、SPI デバイスのチップセレクト信号線など

ハードウェア情報については、struct resource 構造体を使用してデバイスが提供するリソースを保存します。例えば、デバイスが使用する割り込み番号やレジスタの物理アドレスなどです。構造体の原型は以下の通りです：

株式会社日昇テクノロジー

リスト 2: resource 構造体(カーネルソースコード/include/linux/ioport.h 内)

```
1 /*
2 * リソースはツリー構造で、
3 * ネストなどが可能です。
4 */
5
6 struct resource {
7 resource_size_t start;
8 resource_size_t end;
9 const char *name;
10 unsigned long flags;
11 /* その他のメンバ省略 */
12 };
```

- name：リソースの名前を指定します。NULL に設定することもできます；

- start、end：リソースの開始アドレスおよび終了アドレスを指定します

- flags：リソースのタイプを指定するために使用されます。Linux では、I/O、Memory、Register、IRQ、DMA、Bus など多くのタイプのリソースがありますが、最も一般的なものはいくつかあります：

リソースマクロ定義	説明
IORESOURCE_IO	IO アドレス空間用で、IO ポートマッピング方式に対応しています。
IORESOURCE_MEM	直接アドレス指定可能な周辺機器のアドレス空間用です。
IORESOURCE_IRQ	該当デバイスが使用する特定の割り込みを指定するために使用されます。
IORESOURCE_DMA	使用する DMA チャンネルを指定するために使用されます。

株式会社日昇テクノロジー

デバイスドライバの主要な目的は、デバイスのレジスタを操作することです。異なるアーキテクチャのコンピュータは異なる操作インターフェイスを提供しており、主に IO ポートマッピングと IO メモリマッピングの二つの方法があります。IO ポートマッピング方式に対応して、専用のインターフェイス関数（例えば inb、outb）を通じてのみアクセスできます。IO メモリマッピング方式を採用すると、メモリにアクセスするようにレジスタを読み書きすることができます。組み込みシステムでは、基本的に IO アドレス空間がないため、通常は IORESOURCE_MEM を使用します。

リソースの開始アドレスと終了アドレスにおいて、IORESOURCE_IO または IORESOURCE_MEM については、使用するメモリの開始位置及び終了位置を示します。割り込みピンやチャネルなどを使用する場合、start と end メンバの値は等しくなければなりません。

ソフトウェア情報については、この特別な情報をプライベートデータとして封装して保存する必要があります。platform_device 構造体には、device 構造体タイプのメンバ dev があります。前の章で触れたように、Linux デバイスモデルは物理デバイスを抽象化するために device 構造体を使用しており、この構造体のメンバ platform_data はデバイスのプライベートデータを保存するために使用できます。platform_data は void *タイプの万能ポインタで、提供したい内容が何であれ、データのアドレスを platform_data に割り当てるだけでよいのです。GPIO ピン番号を例にすると、以下のコードのようになります：

リスト 3：サンプルコード

```
1 unsigned int pin = 10;
2
3 struct platform_device pdev = {
4 .dev = {
5 .platform_data = &pin;
```

株式会社日昇テクノロジー

```
6 }

7 }
```

GPIO ピン番号を保存した変数 pin のアドレスを platform_data ポインタに割り当てることで、ドライバプログラムはプラットフォームデバイスバスのコア関数を呼び出すことにより、必要なピン番号を取得できます。

重要: ハードウェア情報の記述には、新しい方法としてデバイスツリーが使用されます。詳細は後のデバイスツリー（DT）とデバイスツリーオーバーレイ（DTO）の章を参照してください。

7.1.3 プラットフォームデバイスの登録/登録解除

platform_device 構造体を定義し初期化した後、それをプラットフォームデバイスバスに登録し、マウントする必要があります。プラットフォームデバイスを登録するには platform_device_register()関数を使用します。この関数のプロトタイプは以下の通りです：

リスト 4: platform_device_register 関数(カーネルソースコード/drivers/base/platform.c 内)

```
1 int platform_device_register(struct platform_device *pdev)
```

関数のパラメータと戻り値は以下の通りです：

パラメータ： pdev: platform_device 型の構造体ポインタ

戻り値：

- 成功： 0

- 失敗： 負の数

同様に、特定のプラットフォームデバイスを登録解除し、削除する必要がある場合は、platform_device_unregister 関数を使用して、プラットフォームデバイスバスにそのデバイスの削除を通知する必要があります。

株式会社日昇テクノロジー

リスト 5: platform_device_unregister 関数(カーネルソースコード/drivers/base/platform.c 内)

```
1 void platform_device_unregister(struct platform_device *pdev)
```

関数のパラメータと戻り値は以下の通りです：

パラメータ： pdev: platform_device 型の構造体ポインタ

戻り値： なし

これで、プラットフォームデバイスに関する知識は説明が完了しました。プラットフォームデバイスの主要な内容は、ハードウェア部分のコードとドライバ部分のコードを分け、プラットフォームデバイスバスに登録することです。プラットフォームデバイスバスは、デバイスとドライバの間に橋を架ける役割を果たし、統一されたデータ構造と関数インターフェイスを通じて、デバイスとドライバのデータ交換を直接行います。

7.2 プラットフォームデバイスドライバ

プラットフォームデバイスドライバは、プラットフォームバス専用のデバイスに特化したものであり、伝統的なバスデバイス、例えば I2C や SPI デバイスはプラットフォームデバイスであり、I2C や SPI バスに依存しています。

7.2.1 platform_driver 構造体

カーネルは platform_driver 構造体を使用してプラットフォームドライバを記述します。構造体の原型は以下の通りです：

リスト 6: platform_driver 構造体(カーネルソースコード/include/platform_device.h 内)

```
1 struct platform_driver {
2
3 int (*probe)(struct platform_device *);
4 int (*remove)(struct platform_device *);
```

株式会社日昇テクノロジー

```
5 struct device_driver driver;
6 const struct platform_device_id *id_table;
7 .....
8 };
```

- probe：関数ポインタ。ドライバ開発者はこの関数ポインタをドライバプログラム内で初期化する必要があり、バスがデバイスとドライバをマッチングさせた後に、この関数がコールバックとして実行されます。この関数を通じて、デバイスの一連の初期化を行うことが一般的です。

- remove：関数ポインタ。ドライバ開発者はこの関数ポインタをドライバプログラム内で初期化する必要があり、特定のプラットフォームデバイスを削除する際に、この関数がコールバックとして実行されます。この関数で実装される操作は、通常、probe 関数で実装された操作の逆プロセスです。

- driver：Linux デバイスモデルでドライバを抽象化するための device_driver 構造体。platform_driver はこの構造体を継承しているため、デバイスモデルドライバオブジェクトの特性を得ることができます。

- id_table：そのドライバが対応可能なデバイスのタイプを示します。

platform_device_id 構造体の原型は以下の通りです:

リスト 7: id_table 構造体(カーネルソースコード/include/linux/mod_devicetable.h 内)

```
1 struct platform_device_id {
2 char name[PLATFORM_NAME_SIZE];
3 kernel_ulong_t driver_data;
4
5 };
```

platform_device_id 構造体には 2 つのメンバがあります。1 つ目はドライバの名前を指定するため

株式会社日昇テクノロジー

の配列で、バスがマッチングを行う際には、この構造体の name メンバと platform_device の name 変数を比較してマッチングします。もう 1 つのメンバ変数 driver_data は、デバイスの設定を保存するために使用されます。同系列のデバイスでは、一部のレジスタの設定だけが異なることがあります。コードの冗長性を減らし、一つのドライバが複数のデバイスに対応できるようにするために、この構造体が使用されます。次に、imx チップのシリアルポートを例に、この構造体の役割を具体的に見ていきます：

リスト 8: サンプルコード(カーネルソースコード/drivers/tty/serial/imx.c 内)

```
1 static struct imx_uart_data imx_uart_devdata[] = {
2
3 [IMX1_UART] = {
4 .uts_reg = IMX1_UTS,
5 .devtype = IMX1_UART,
6 },
7
8 [IMX21_UART] = {
9 .uts_reg = IMX21_UTS,
10 .devtype = IMX21_UART,
11 },
12
13 [IMX6Q_UART] = {
14 .uts_reg = IMX21_UTS,
15 .devtype = IMX6Q_UART,
```

株式会社日昇テクノロジー

```
16 },
17
18 };
19
20 static struct platform_device_id imx_uart_devtype[] = {
21
22 {
23 .name = "imx1-uart",
24 .driver_data = (kernel_ulong_t) &imx_uart_devdata[IMX1_UART],
25 },
26
27 {
28 .name = "imx21-uart",
29 .driver_data = (kernel_ulong_t) &imx_uart_devdata[IMX21_UART],
30 },
31
32 {
33 .name = "imx6q-uart",
34 .driver_data = (kernel_ulong_t) &imx_uart_devdata[IMX6Q_UART],
35
36 },
37
```

株式会社日昇テクノロジー

```
38 {
39 /* sentinel */
40
41 }
42
43 };
```

- 第 1-18 行：異なるプラットフォームのシリアルタイプを表す構造体配列を宣言しています。

- 第 20-42 行：platform_device_id 構造体の driver_data メンバを使用して、上記のシリアル情報を保存します。

上記のコードでは、imx1、imx21、imx6q の 3 つの異なるのチップに対応するシリアルデバイスがサポートされています。これらの違いは、シリアルの test レジスタのアドレスが異なる点にあります。バスがプラットフォームドライバとプラットフォームデバイスを正常にペアリングすると、対応する id_table エントリがプラットフォームデバイスの id_entry メンバに割り当てられ、プラットフォームドライバの probe 関数はプラットフォームデバイスをパラメータとして受け取るため、現在のデバイスのシリアル test レジスタのアドレスを取得できます。

7.2.2 プラットフォームドライバの登録/登録解除

platform_driver を初期化した後、platform_driver_register()関数を使用してプラットフォームドライバを登録します。この関数のプロトタイプは以下の通りです：

リスト 9: platform_driver_register 関数

```
1 int platform_driver_register(struct platform_driver *drv);
```

関数のパラメータと戻り値：

- **パラメータ**： drv: platform_driver 型の構造体ポインタ

株式会社日昇テクノロジー

- **戻り値**：

  - 成功：0

  - 失敗：負の数

platform_driver が driver 構造体を継承しているため、Linux デバイスモデルの知識と組み合わせると、プラットフォームドライバが成功裏に登録されると、/sys/bus/platform/drivers ディレクトリ下に新しいディレクトリが生成されます。

ドライバモジュールをアンロードする際には、登録されたプラットフォームドライバを登録解除する必要があります。platform_driver_unregister()関数は、登録されたプラットフォームドライバを登録解除するために使用されます。この関数のプロトタイプは以下の通りです：

リスト 10: platform_driver_unregister 関数(カーネルソースコード/drivers/base/platform.c)

```
1 void platform_driver_unregister(struct platform_driver *drv);
```

- **パラメータ**： drv: platform_driver 型の構造体ポインタ

- **戻り値**：なし

上述の内容は最も基本的なプラットフォームドライバーフレームワークに関するもので、probe 関数と remove 関数を実装し、platform_driver 構造体を初期化して、platform_driver_register を呼び出して登録するだけで済みます。

7.2.3 プラットフォームドライバでのデバイス情報の取得

プラットフォームデバイスでは、構造体 resource を使用してハードウェア情報を抽象的に表現し、ソフトウェア情報は device 構造体の platform_data メンバで保存します。まず、プラットフォームデバイスの resource 構造体が提供するリソースを取得する方法を見てみましょう。

platform_get_resource()関数は通常、ドライバの probe 関数内で実行され、プラットフォームデバイスが提供するリソース構造体を取得するために使用されます。この関数のプロトタイプは以下の通

株式会社日昇テクノロジー

りです：

リスト 11: platform_get_resource 関数

```
1 struct resource *platform_get_resource(struct platform_device *dev, unsigned int type, unsigned int
num);
```

- **パラメータ**：

  - dev：どのプラットフォームデバイスのリソースを取得するか指定します；

  - type：取得するリソースのタイプを指定します。例：IORESOURCE_MEM、IORESOURCE_IO など；

  - num：取得するリソースの番号を指定します。デバイスによっては複数のリソースが必要な場合があり、カーネルはこれらのリソースに番号を付けています。異なるリソースタイプ間で番号は独立しています。

- **戻り値**：

  - 成功：struct resource 型のポインタ

  - 失敗：NULL

IORESOURCE_IRQ タイプのリソースを取得する場合、プラットフォームデバイスドライバは以下の関数インターフェースを提供します：

リスト 12: platform_get_irq 関数

```
1 int platform_get_irq(struct platform_device *pdev, unsigned int num)
```

- **パラメータ**：

  - pdev：どのプラットフォームデバイスのリソースを取得するか指定します；

  - num：取得するリソースの番号を指定します。

- **戻り値**：

  - 成功：使用可能な割り込み番号

株式会社日昇テクノロジー

- 失敗：負の数

device 構造体の platform_data メンバに保存されたソフトウェア情報にアクセスするには、dev_get_platdata 関数を使用します。この関数のプロトタイプは以下の通りです：

リスト 13: dev_get_platdata 関数

```
1 static inline void *dev_get_platdata(const struct device *dev)
2 {
3 return dev->platform_data;
4 }
```

- **パラメータ**：

  - dev：struct device 型のポインタ

- **戻り値**：device 構造体の platform_data メンバのポインタ

以上が、プラットフォームデバイスからリソースを取得するための一般的な関数インターフェースです。これでプラットフォームドライバの部分はほぼ終了です。まとめると、プラットフォームドライバは probe 関数を実装し、プラットフォームバスがドライバとデバイスを正常にマッチングさせると、ドライバの probe 関数が呼び出されます。この関数内で上述の関数インターフェイスを使用してリソースを取得し、デバイスを初期化します。最後に、platform_driver 構造体を満たし、platform_driver_register を呼び出して登録します。

7.3 プラットフォームバス

7.3.1 プラットフォームバスの登録とマッチング方法

Linux のデバイスドライバモデルにおいて、バスは最も重要な要素の一つです。前章で、バスはデバイスとドライバのマッチングを担当し、登録されたプラットフォームデバイスとプラットフォームドライバのリストを管理していると述べました。新しいデバイスやドライバがバスに加わると、バスは

株式会社日昇テクノロジー

platform_match 関数を呼び出して、新たなデバイスやドライバのペアリングを行います。バスは、システム内のバスを抽象的に記述するために bus_type 構造体を使用します。プラットフォームバスの構造体原型は以下の通りです：

リスト 14: platform_bus_type 構造体(カーネルソースコード/driver/base/platform.c)

```
1 struct bus_type platform_bus_type = {
2
3 .name = "platform",
4 .dev_groups = platform_dev_groups,
5 .match = platform_match,
6 .uevent = platform_uevent,
7 .pm = &platform_dev_pm_ops,
8
9 };
10
11 EXPORT_SYMBOL_GPL(platform_bus_type);
```

カーネルは platform_bus_type を使用してプラットフォームバスを記述し、Linux カーネルの起動時に自動で登録されます。

リスト 15: platform_bus_init 関数(カーネルソースコード/driver/base/platform.c)

```
1 int __init platform_bus_init(void)
2 {
3 int error;
4 ...
```

株式会社日昇テクノロジー

```
5 error = bus_register(&platform_bus_type);

6 ...

7 return error;

8 }
```

第 5 行：Linux カーネルに platform バスを登録します

ここでの重点は、platform バスの match 関数ポインタです。この関数ポインタは、プラットフォームバスとプラットフォームデバイスのマッチングプロセスを実行する関数を指します。各ドライババスは、この関数ポインタをインスタンス化する必要があります。platform_match の関数原型は以下の通りです：

リスト 16: platform_match 関数(カーネルソースコード/driver/base/platform.c)

```
1 static int platform_match(struct device *dev, struct device_driver *drv)

2 {

3

4 struct platform_device *pdev = to_platform_device(dev);

5 struct platform_driver *pdrv = to_platform_driver(drv);

6

7 /* ドライバのオーバーライドが設定されている場合、一致するドライバにのみバインドする */

8 if (pdev->driver_override)

9 return !strcmp(pdev->driver_override, drv->name);

10

11 /* 最初に OF スタイルのマッチを試みる */

12 if (of_driver_match_device(dev, drv))
```

株式会社日昇テクノロジー

```
13 return 1;
14
15 /* 次に ACPI スタイルのマッチを試みる */
16 if (acpi_driver_match_device(dev, drv))
17 return 1;
18
19 /* id テーブルによるマッチを試みる */
20 if (pdrv->id_table)
21 return platform_match_id(pdrv->id_table, pdev) != NULL;
22
23 /* フォールバックとしてドライバ名のマッチを試みる */
24 return (strcmp(pdev->name, drv->name) == 0);
25
26 }
```

- 第 4-5 行：to_platform_device()と to_platform_driver()マクロを呼び出しています。これらのマクロは以下のように定義されています：

リスト 17: to_platform_xxx マクロ定義(カーネルソースコード/include/linux/platform_device.h)

```
1 #define to_platform_device(x) (container_of((x), struct platform_device, dev))
2 #define to_platform_driver(drv) (container_of((drv), struct platform_driver, driver))
```

to_platform_device と to_platform_driver は、container_of のラッパーであり、dev、driver をそれぞれ platform_device、platform_driver のメンバ変数として、マッチング中の platform_driver と platform_device を取得できます。

- 第 8-21 行：platform バスは 4 つのマッチング方式を提供しており、優先順位は次の通りで

株式会社日昇テクノロジー

す：デバイスツリー方式>ACPI マッチング方式>id_table 方式>文字列比較方式。マッチング方法は多岐にわたりますが、いずれも特定のメンバの文字列が一致するかどうかを比較するだけのシンプルなものです。デバイスツリーはハードウェア情報を記述するデータ構造で、C 言語以外のスクリプトでこれらのデバイス情報を記述します。ドライバとデバイスのマッチングは、compatible の値を比較することによって行われます。ACPI は主に電源管理に使用されるため、ここでは詳しく説明しません。デバイスツリーのマッチングメカニズムについては、デバイスツリーの章で詳しく説明されます。

7.3.2 id_table によるマッチング方式

この章では、プラットフォームバスの id_table マッチング方式に焦点を当てます。platform_driver 構造体を定義する際には、対応可能なデバイスを示す id_table 配列を提供する必要があります。ドライバがロードされると、バスの match 関数が id_table が非 null であることを発見し、id_table 内の name メンバとプラットフォームデバイスの name メンバを比較し、一致する場合はマッチングしたエントリを返します。具体的な実装は以下の通りです：

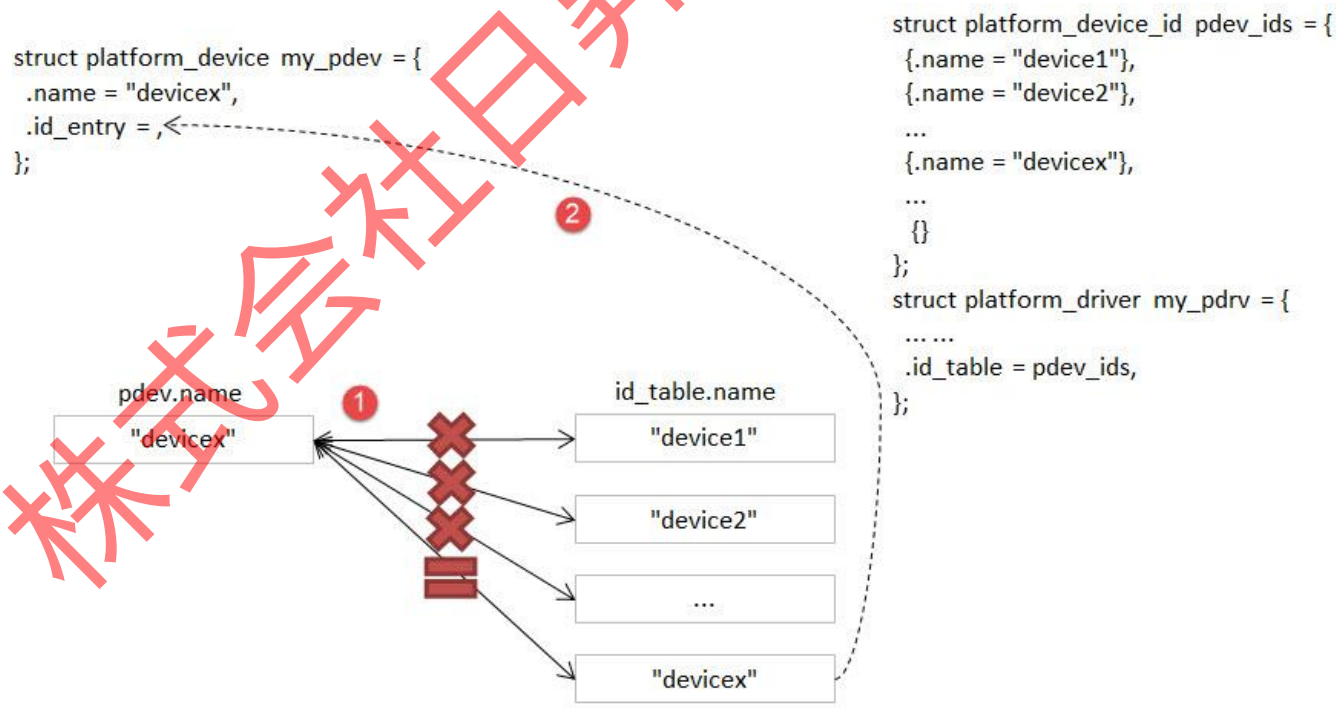
リスト 18: platform_match_id 関数(カーネルソースコード/drivers/base/platform.c)

```
1 static const struct platform_device_id *platform_match_id(
2 const struct platform_device_id *id,
3 struct platform_device *pdev)
4
5 {
6 while (id->name[0]) {
7 if (strcmp(pdev->name, id->name) == 0) {
8 pdev->id_entry = id;
```

株式会社日昇テクノロジー

```
9 return id;
10 }
11 id++;
12 }
13 return NULL;
14 }
```

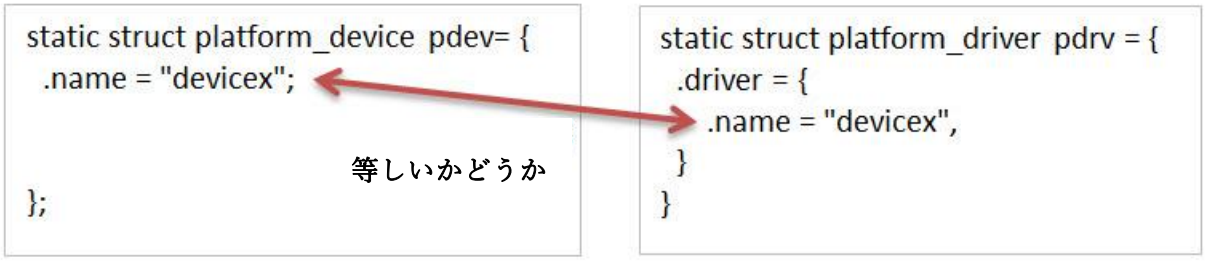
このコードはシンプルで、文字列でのペアリングのみを行います。新しいドライバやデバイスがバスに追加されると、バスは match 関数を呼び出して新しいデバイスやドライバをペアリングします。platform_match_id 関数の第一パラメータはドライバが提供する id_table で、第二パラメータはペアリング対象のプラットフォームデバイスです。プラットフォームデバイスの name フィールドの値がドライバの id_table の値と一致すると、現在のマッチング項目を platform_device の id_entry に割り当て、非 null ポインタを返します。マッチングに失敗すると、null ポインタを返します。



もしドライバが前述の 3 つのマッチング方式のいずれも提供していない場合、バスは platform_device の name フィールドと platform_driver 内の device_driver の name フィールドを

株式会社日昇テクノロジー

比較してマッチングを行います。



マッチングに成功すると、マッチングされたドライバの probe 関数が呼び出され、struct platform_device パラメータが渡されます。

7.4 プラットフォームデバイス実験コード解説

このセクションの実験では、Lubancat_RK ボードを使用し、サンプルコードのディレクトリは linux_driver/platform_driver です。このセクションでは、LED キャラクタデバイスドライバのコードにプラットフォームデバイスドライバを適用し、ハードウェアとソフトウェアのコードを分離することで、プラットフォームデバイスドライバの学習を強化します。LED に関連するドライバは、「キャラクタデバイスドライバ - LED ライトを点灯」セクションを参照してください。

7.4.1 プログラミングの考え方

1. 最初のカーネルモジュール led_pdev.c を書く
2. カーネルモジュール内にプラットフォームデバイスを定義し、LED に関連するデバイス情報を充填する
3. そのモジュールのエントリ関数で、このプラットフォームデバイスを登録/マウントする
4. 2 つ目のカーネルモジュール led_pdrv.c を書く
5. カーネルモジュール内にプラットフォームドライバを定義し、probe 関数でキャラクタデバイスドライバの作成を完了する
6. そのモジュールのエントリ関数で、このプラットフォームドライバを登録/マウントする

プラットフォームデバイスバス上で、プラットフォームデバイスとプラットフォームドライバを登録

株式会社日昇テクノロジー

/マウントすると、自動的にペアリングされます。ペアリングに成功すると、プラットフォームドライバの probe 関数がコールバックされ、キャラクタデバイスドライバの作成が完了します。

7.4.2 コード分析

7.4.2.1 プラットフォームデバイスの定義

キャラクタデバイスのハードウェア情報を抽出し、それを独立したコードとしてプラットフォームデバイスに登録する必要があります。LED ライトを点灯するには、LED に関連するレジスタ、GPIO クロックレジスタ、IO 設定レジスタ、IO データレジスタなどを制御する必要があります。ここでのリソースは、実際にはレジスタのアドレスであり、IORESOURCE_MEM を使用して処理できます。さらに、レジスタのオフセットなどの追加情報も提供する必要がありますが、これはプラットフォームデバイスのプライベートデータを利用して管理できます。

リスト 19: レジスタマクロ定義(../linux_driver/platform_driver/led_pdev.c 内)

```
1 //キャラクタデバイス LED ドライバを参照、 GPIO1_C7 ピン
2 #define GPIO1_BASE (0xFD5F8000) // GPIO1 のベースアドレス定義
3 #define GPIO1_DR ( GPIO1_BASE + 0x0004) // GPIO1_DR_H 定義
4 #define GPIO1_DDR ( GPIO1_BASE + 0x000C) // GPIO1_DDR_H 定義
```

- GPIO ピンのレジスタをマクロで包装します。

以下に示すように、上述のレジスタアドレスを格納するための resource 構造体を定義し、ドライバーが使用するために提供します：

株式会社日昇テクノロジー

リスト 20: リソース配列の定義(../linux_driver/platform_driver/led_pdev.c 内)

```
1 static struct resource rled_resource[] = {
2 [0] = DEFINE_RES_MEM( GPIO1_DR, 4),
3 [1] = DEFINE_RES_MEM( GPIO1_DDR, 4)
4 };
```

- カーネルソースコード/include/linux/ioport.h では、DEFINE_RES_MEM、DEFINE_RES_IO、DEFINE_RES_IRQ、DEFINE_RES_DMA などのマクロを提供しており、必要なリソースタイプを定義できます。DEFINE_RES_MEM は IORESOURCE_MEM タイプのリソースを定義するために使用されます。

リスト 21: プラットフォームデバイスのプライベートデータ定義
(../linux_driver/platform_driver/led_pdev.c 内)

```
1 unsigned int led_hwinfo[1] = { 7 };
```

- led_hwinfo 配列を使用してレジスタのオフセットを記録し、プラットフォームデータにこの配列の先頭アドレスを割り当てます。

リスト 22: プラットフォームデバイスの定義(../linux_driver/platform_driver/led_pdev.c 内)

```
1 static int led_cdev_release(struct inode *inode, struct file *filp)
2 {
3 return 0;
4 }
5
6 /* red led device */
7 static struct platform_device rled_pdev = {
```

株式会社日昇テクノロジー

```
8 .name = "led_pdev",
9 .id = 0,
10 .num_resources = ARRAY_SIZE(led_resource),
11 .resource = led_resource,
12 .dev = {
13 .release = led_release,
14 .platform_data = led_hwinfo,
15 },
16
17 };
```

- 第 1-4 行：led_cdev_release 関数を宣言しました。これは、モジュールのアンロード時にカーネルからエラーメッセージが表示されないようにするためです。

- 第 7-9 行："led_pdev"という名前のデバイスを定義しました。この名前はドライバの名前と一致している必要があります。そうでなければマッチングに失敗します。id 番号は 0 に設定され、この番号を使用してデバイスを登録します。

- 第 10-11 行：先に実装した rled_resource 配列を resource メンバに割り当てました。また、リソースの数を指定する必要があり、カーネルは ARRAY_SIZE マクロを提供しています。これは配列の長さを計算するためのもので、そのため num_resources は ARRAY_SIZE(rled_resource) に直接設定されます。

- 第 12-15 行：dev メンバの値を設定し、rled_hwinfo を platform_data に格納しました。

最後に、モジュールのロード関数内で platform_device_register 関数を呼び出すことで、このカーネルモジュールをロードする際に新しいプラットフォームデバイスがカーネルに登録されます。具体的な実装方法は以下のとおりです：

株式会社日昇テクノロジー

リスト 23: モジュールの初期化(../linux_driver/platform_driver/led_pdev.c 内)

```
1 static __init int led_pdev_init(void)
2 {
3 printk("pdev init¥n");
4 platform_device_register(&rled_pdev);
5 return 0;
6
7 }
8 module_init(led_pdev_init);
9
10
11 static __exit void led_pdev_exit(void)
12 {
13 printk("pdev exit¥n");
14 platform_device_unregister(&rled_pdev);
15
16 }
17 module_exit(led_pdev_exit);
18
19
20 MODULE_AUTHOR("Embedfire");
21 MODULE_LICENSE("GPL");
22 MODULE_DESCRIPTION("the example for platform driver");
```

株式会社日昇テクノロジー

- 第 1-8 行：モジュールのエントリ関数を実装し、情報を出力してプラットフォームデバイスを登録します。

- 第 10-16 行：モジュールのエグジット関数を実装し、情報を出力してデバイスを登録解除します。

- 第 18-20 行：モジュールが遵守するライセンスとその他のモジュール情報について記述します。

これにより、新しいデバイスを実装しました。開発ボード上でこのモジュールをロードすると、プラットフォームバスに LED ライトのプラットフォームデバイスがマウントされます。

7.4.2.2 プラットフォームドライバの定義

新しいプラットフォームデバイスを登録しました。ドライバは、そのデバイスが提供するリソースを取得し、対応する操作方法を提供するだけでよいです。ここでは、LED ライトを制御するためのキャラクターデバイスを引き続き使用します。LED ライトのキャラクターデバイスコードについては、既に皆さんがよく知っていると思いますので、この部分のコードについては詳細な説明は省略します。重点をプラットフォームドライバーに置きましょう。

ドライバは、id_table を提供することでデバイスをマッチングします。platform_device_id 型の変数 led_pdev_ids を定義し、ドライバがサポートするデバイスを指定します。ここでは、led_pdev という名前のデバイス 1 つだけをサポートします。これはプラットフォームデバイスが提供する名前と一致させる必要があります。

株式会社日昇テクノロジー

リスト 24: id_table (../linux_driver/platform_driver/led_pdrv.c に位置)

```
1 static struct platform_device_id led_pdev_ids[] = {
2 {.name = "led_pdev"},
3 {}
4 };
5 MODULE_DEVICE_TABLE(platform, led_pdev_ids);
```

- このコードは、ドライバがどのデバイスをサポートするかを提供します
- MODULE_DEVICE_TABLE()。このマクロは、ドライバがサポートできるデバイスを記述する ID テーブルを公開し、デバイスのマッチングに使用されます。

これは、バスがマッチングを行うために必要な最初のコンテンツのみを完了しています。マッチング成功後、ドライバはデバイスのリソースを抽出する必要があり、この作業はすべて probe 関数内で行われます。キャラクターデバイスのフレームワークを使用しているため、probe のプロセスでは、キャラクターデバイスの登録などの作業も完成させる必要があります。具体的な実装コードは以下の通りです：

リスト 25: led_pdrv_probe 関数 (../linux_driver/platform_driver/led_pdrv.c に位置)

```
1 struct led_data {
2 unsigned int led_pin;
3 unsigned int __iomem *va_MODER;
4 unsigned int __iomem *va_OTYPER;
5
6 struct cdev led_cdev;
7 };
```

株式会社日昇テクノロジー

```
8 /* コードの一部省略 */
9 static int led_pdrv_probe(struct platform_device *pdev)
10 {
11 struct led_data *cur_led;
12 unsigned int *led_hwinfo;
13
14 struct resource *mem_DR;
15 struct resource *mem_DDR;
16
17 dev_t cur_dev;
18
19 int ret = 0;
20
21 printk("led platform driver probe¥n");
22
23 // 第一歩: プラットフォームデバイスが提供するリソースの抽出
24 // devm_kzalloc 関数で cur_led と led_hwinfo 構造体のメモリサイズを確保
25 cur_led = devm_kzalloc(&pdev->dev, sizeof(struct led_data), GFP_KERNEL);
26 if(!cur_led)
27 return -ENOMEM;
28 led_hwinfo = devm_kzalloc(&pdev->dev, sizeof(unsigned int), GFP_KERNEL);
29 if(!led_hwinfo)
```

株式会社日昇テクノロジー

```
30 return -ENOMEM;
31
32 /* LED のピンとレジスタのシフトを取得 */
33 // dev_get_platdata 関数でプライベートデータを取得し、LED ライトのレジスタオフセットを
cur_led->led_pin に割り当て
34 led_hwinfo = dev_get_platdata(&pdev->dev);
35
36 cur_led->led_pin = led_hwinfo[0];
37 /* プラットフォームリソースの取得 */
38 // platform_get_resource 関数を使用して、各レジスタのアドレスを取得できる
39 mem_DR = platform_get_resource(pdev, IORESOURCE_MEM, 0);
40 mem_DDR = platform_get_resource(pdev, IORESOURCE_MEM, 1);
41
42 // devm_ioremap で取得したレジスタアドレスを仮想アドレスに変換
43 cur_led->va_DR = devm_ioremap(&pdev->dev, mem_DR->start, resource_size(mem_DR));
44 cur_led->va_DDR = devm_ioremap(&pdev->dev, mem_DDR->start, resource_size(mem_DDR));
45
46 // 第二步: キャラクターデバイスの登録
47 cur_dev = MKDEV(DEV_MAJOR, pdev->id);
48
49 register_chrdev_region(cur_dev, 1, "led_cdev");
50
```

株式会社日昇テクノロジー

```
51 cdev_init(&cur_led->led_cdev, &led_cdev_fops);

52

53 ret = cdev_add(&cur_led->led_cdev, cur_dev, 1);

54 if(ret < 0)

55 {

56 printk("cdev の追加に失敗¥n");

57 goto add_err;

58 }

59

60 device_create(led_test_class, NULL, cur_dev, NULL, DEV_NAME "%d", pdev->id);

61

62 /* drvdata として保存 */

63 // platform_set_drvdata 関数で、LED データ情報をプラットフォームドライバ構造体の pdev-

>dev->driver_data に保存

64 platform_set_drvdata(pdev, cur_led);

65

66 return 0;

67

68 add_err:

69 unregister_chrdev_region(cur_dev, 1);

70 return ret;

71 }
```

株式会社日昇テクノロジー

• 第 1-7 行: LED ライトのハードウェア情報を管理するために、再び led_data 構造体を使用し、クロックレジスタの仮想アドレス変数を定義。

• 第 25-30 行: devm_kzalloc 関数で cur_led と led_hwinfo 構造体のメモリサイズを確保。

• 第 34 行: dev_get_platdata 関数でプライベートデータを取得。

• 第 39-40 行: platform_get_resource 関数を利用して各レジスタのアドレスを取得。

• 第 43-44 行: devm_ioremap で取得したレジスタアドレスを仮想アドレスに変換し、リソースの抽出作業を完了。

• 第 47-60 行: LED キャラクターデバイスの登録が必要。MKDEV マクロを使用してデバイス番号を作成し、register_chrdev_region、cdev_init、cdev_add などの関数を呼び出してキャラクターデバイスを登録。

• 第 64 行: platform_set_drvdata 関数を使用して、LED データ情報をプラットフォームドライバ構造体の pdev->dev->driver_data に保存。

ドライバのカーネルモジュールがアンロードされる時、登録したドライバを解除し、対応するキャラクターデバイスも同様に解除する必要があります。具体的な実装コードは以下の通りです：

リスト 26: led_pdrv_remove 関数 (../linux_driver/platform_driver/led_pdrv.c に位置)

```
1 static int led_pdrv_remove(struct platform_device *pdev)
2 {
3 dev_t cur_dev;
4 // platform_get_drvdata で、現在の LED ライトに対応する構造体を取得
5 struct led_data *cur_data = platform_get_drvdata(pdev);
6
7 printk("led platform driver remove¥n");
8
```

株式会社日昇テクノロジー

```
9 cur_dev = MKDEV(DEV_MAJOR, pdev->id);

10

11 // cdev_del で対応するキャラクターデバイスを削除

12 cdev_del(&cur_data->led_cdev);

13

14 // /dev ディレクトリ下のデバイスを削除

15 device_destroy(led_test_class, cur_dev);

16

17 // unregister_chrdev_region で、現在のキャラクターデバイス番号を解除

18 unregister_chrdev_region(cur_dev, 1);

19

20 return 0;

21

22 }
```

- 第 5 行: probe 関数で platform_set_drvdata を呼び出し、現在の LED ライトデータ構造体を pdev の driver_data メンバーに保存。この関数で platform_get_drvdata を呼び出し、現在の LED ライトに対応する構造体を取得。この構造体にはキャラクターデバイスが含まれている。
- 第 12-20 行: cdev_del を呼び出して対応するキャラクターデバイスを削除し、/dev ディレクトリ下のデバイスを削除するために device_destroy を呼び出し、最後に unregister_chrdev_region を使用して、現在のキャラクターデバイス番号を解除。

LED ライトキャラクターデバイスの操作方法については、以下の通り実装されます。ここでは簡単な紹介のみ行い、具体的な紹介は LED ライトキャラクターデバイスのセクションで参照してくださ

株式会社日昇テクノロジー

い。

リスト 27: LED ライトのキャラクターデバイスフレームワーク

(../linux_driver/platform_driver/led_pdrv.c に位置)

```
1 static int led_cdev_open(struct inode *inode, struct file *filp)
2 {
3 unsigned int val = 0;
4 struct led_data *cur_led = container_of(inode->i_cdev, struct led_data, led_cdev);
5
6 printk("led_cdev_open() ¥n");
7
8 // ピンを出力に設定
9 val = readl(cur_led->va_DDR);
10 val |= ((unsigned int)0X1 << (cur_led->led_pin+16));
11 val |= ((unsigned int)0X1 << (cur_led->led_pin));
12 writel(val,cur_led->va_DDR);
13
14 // デフォルトで高電圧を出力設定
15 val = readl(cur_led->va_DR);
16 val |= ((unsigned int)0X1 << (cur_led->led_pin+16));
17 val |= ((unsigned int)0x1 << (cur_led->led_pin));
18 writel(val, cur_led->va_DR);
19
20 filp->private_data = cur_led;
```

株式会社日昇テクノロジー

```
21
22 return 0;
23 }
24
25 static int led_cdev_release(struct inode *inode, struct file *filp)
26 {
27 return 0;
28 }
29
30 static ssize_t led_cdev_write(struct file *filp, const char __user * buf, size_t count, loff_t * ppos)
31 {
32 unsigned long val = 0;
33 unsigned long ret = 0;
34
35 int tmp = count;
36
37 struct led_data *cur_led = (struct led_data *)filp->private_data;
38
39 val = kstrtoul_from_user(buf, tmp, 10, &ret);
40
41 val = readl(cur_led->va_DR);
42 if (ret == 0)
```

株式会社日昇テクノロジー

```
43 {
44 val |= ((unsigned int)0x1 << ((cur_led->led_pin)+16));
45 val &= ~((unsigned int)0X1 << (cur_led->led_pin));
46 }
47 else
48 {
49 val |= ((unsigned int)0x1 << (cur_led->led_pin+16));
50 val |= ((unsigned int)0X1 << (cur_led->led_pin));
51 }
52 writel(val, cur_led->va_DR);
53
54 *ppos += tmp;
55
56 return tmp;
57 }
58
59 static struct file_operations led_cdev_fops = {
60 .open = led_cdev_open,
61 .release = led_cdev_release,
62 .write = led_cdev_write,
63
64 };
```

株式会社日昇テクノロジー

- 第 1-23 行は、led_cdev_open 関数の内容で、主にハードウェアの初期化を行います。
- 第 25-28 行の led_cdev_release 関数は、警告を防ぐために存在します。
- 第 38-53 行は、GPIO ピンをセットするための制御です。
- 第 60-65 行は、file_operations 構造体を埋めることです。

kstrtoul_from_user()関数については、以下の定義があります：

リスト 28: kstrtoul_from_user()関数 (カーネルソースコード/include/linux/kernel.h)

```
1 int __must_check kstrtoul_from_user(const char __user *s, size_t count, unsigned int base, unsigned
long *res);
```

関数のパラメータと戻り値は以下の通りです：

パラメータ：

- s：文字列の開始アドレスで、この文字列はヌル文字で終わる必要があります；
- count：変換するデータのサイズ；
- base：変換の基数で、base=0 の場合、関数は文字列のタイプを自動的に判断し、10 進数で出力します。たとえば"0xa"は 10 進数として扱われ（大文字小文字問わず）、出力は 10 になります。0 で始まる場合は 8 進数として解析され、それ以外の場合は 10 進数として解析されます；
- res：変換に成功した結果のアドレスへのポインタ。

戻り値：

この関数は kstrtoul と比較して、追加のパラメータ count を持っています。これは、ユーザースペースからカーネルスペースに直接アクセスすることはできないため、カーネルがユーザーバッファからカーネルバッファへのコピーを実現するために kstrtoul_from_user 関数を提供しているからです。同様の関数には、カーネルスペースのバッファからユーザースペースへのコピーを行う copy_to_user があります。使用しているメモリタイプがそれほど複雑でない場合は、put_user や get_user 関数の使用が選択肢となります。

株式会社日昇テクノロジー

LED キャラクターデバイスの操作を記述した後、実装した内容を platform_driver 型の構造体に埋め込み、platform_driver_register 関数を使用して登録するだけです。

リスト 29: プラットフォームドライバの登録 (../linux_driver/platform_driver/led_pdrv.c に位置)

```
1 static struct platform_driver led_pdrv = {
2
3 .probe = led_pdrv_probe,
4 .remove = led_pdrv_remove,
5 .driver.name = "led_pdev",
6 .id_table = led_pdev_ids,
7 };
8
9 static __init int led_pdrv_init(void)
10 {
11 printk("led platform driver init¥n");
12 //class_create で LED クラスを作成
13 led_test_class = class_create(THIS_MODULE, "test_leds");
14 //platform_driver_register 関数を呼び出して、プラットフォームドライバ構造体を登録します。これにより、このカーネルモジュールをロードするときに、新しいプラットフォームドライバがカーネルに追加されます。
15 platform_driver_register(&led_pdrv);
16
17 return 0;
18 }
```

株式会社日昇テクノロジー

```
19 module_init(led_pdrv_init);
20
21 static __exit void led_pdrv_exit(void)
22 {
23 printk("led platform driver exit¥n");
24 platform_driver_unregister(&led_pdrv);
25 class_destroy(led_test_class);
26 }
27 module_exit(led_pdrv_exit);
28
29 MODULE_AUTHOR("Embedfire");
30 MODULE_LICENSE("GPL");
31 MODULE_DESCRIPTION("platform driver example");
```

- 第 1-6 行: led_pdrv では、プラットフォームバスのマッチングプロセス中に、id_table の name 値に基づいてのみマッチングを行い、プラットフォームデバイスの name 値と一致する場合はマッチング成功と見なします。そうでない場合は、マッチングに失敗し、現在のカーネルにはサポートされるデバイスがないことを意味します。
- 第 13 行: class_create 関数を呼び出して LED クラスを作成し、platform_driver_register 関数を使用してプラットフォームドライバ構造体を登録します。これにより、このカーネルモジュールをロードするときに、新しいプラットフォームドライバがカーネルに追加されます。
- 第 21-27 行: 関数 led_pdrv_exit は、初期化関数の逆プロセスです。

株式会社日昇テクノロジー

7.5 実験準備

Permission denied や類似のメッセージが表示された場合は、ユーザー権限に注意してください。ほとんどの場合、ハードウェアデバイスの操作には root ユーザー権限が必要です。簡単な解決策は、コマンドの前に sudo を付けるか、root ユーザーとしてプログラムを実行することです。

7.5.1 ドライバプログラムのコンパイル

7.5.1.1 Makefile の変更説明

このセクションの実験で使用される Makefile は以下の通りです。この Makefile を書く際には、実際の状況に応じて変数 KERNEL_DIR と obj-m を変更するだけで済みます。

リスト 30: Makefile (../linux_driver/platform_driver/Makefile に位置)

```
1 KERNEL_DIR=../../kernel/
2
3 ARCH=arm64
4 CROSS_COMPILE=aarch64-linux-gnu-
5 export ARCH CROSS_COMPILE
6
7 obj-m := led_pdev.o led_pdrv.o
8
9 all:
10 $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) modules
11 modules clean:
12 $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) clean
```

株式会社日昇テクノロジー

7.5.1.2 コンパイルコマンドの説明

本セクションの実験では、lubancat のボードでは、システムのデバイストリーで LED のデバイス機能がデフォルトで有効になっています。デバイストリーの LEDs ノードを無効にするには、LEDs ノードの status = "okay";を status = "disabled";に変更し、デバイストリーをコンパイルして置き換えるか、またはボード上で以下のコマンドを直接使用してシステム LEDs ドライバによる LED の制御を無効にすることができます：

```
sudo sh -c 'echo 0 > /sys/class/leds/sys_status_led/brightness'
```

LED の明るさを 0 に設定することで、同時に LED のトリガー条件が自動的に none に変わり、LEDs ドライバによる LED の制御がキャンセルされます。

実験ディレクトリで以下のコマンドを入力してドライバモジュールをコンパイルします：

```
make
```

コンパイルが成功すると、実験ディレクトリに“led_pdev.ko”、“led_pdrv.ko”という名前の 2 つのドライバモジュールファイルが生成されます。

7.5.2 アプリケーションのコンパイル

本セクションの実験では、Linux システムに付属の“echo”アプリケーションをテストに使用するため、追加でアプリケーションをコンパイルする必要はありません。

7.5.3 ボードへのドライバプログラムのコピー

実際の状況に応じて調整してください。ssh、scp、nfs などの方法を使用するか、ボード上で直接コンパイルした場合は直接ドライバをロードします。

```
# scp コマンドを使用してボードに直接転送する場合、

scp *.ko cat@192.168.103.2:/home/cat/
```

株式会社日昇テクノロジー

7.6 プログラム実行結果

7.6.1 開発ボードに最初のドライバモジュールをロード

コマンド sudo insmod led_pdev.ko を実行した後、/sys/bus/platform/devices 下に登録した LED ライトデバイス led_pdev.0 を確認できます。後ろの数字 0 はプラットフォームデバイス構造体の ID 番号に対応しています。

```
cat@lubancat:~$ sudo insmod led_pdev.ko
cat@lubancat:~$ lsmod
Module                  Size  Used by
led_pdev               16384  0
cat@lubancat:~$ cd /sys/bus/platform/
devices/           drivers/          drivers_autoprobe  drivers_probe      uevent
cat@lubancat:~$ cd /sys/bus/platform/devices/led_pdev.0
cat@lubancat:/sys/bus/platform/devices/led_pdev.0$ ls
driver_override  modalias  power  subsystem  uevent
cat@lubancat:/sys/bus/platform/devices/led_pdev.0$
```

7.6.2 開発ボードに 2 番目のドライバモジュールをロード

コマンド sudo insmod led_pdrv.ko を実行して、LED のプラットフォームドライバをロードします。コマンド“dmesg | tail”を実行してカーネルのプリントアウトを確認すると、ドライバがロードされる際に led platform driver probe が 1 回プリントアウトされ、マッチングが成功したことが分かります。

```
cat@lubancat:~$ dmesg | tail
[  128.933228] headset_interrupt:get pin level again, pin=5,i=0
[  128.933293] (headset in is high level)headset status is in
[  201.237612] led_pdev:
[  201.238119] pdev init
[  375.857621] pdev exit
[  383.984072] pdev init
[  533.406277] led platform driver init
[  533.407066] led platform driver probe
```

プラットフォームデバイス led_pdev.ko をロードする

プラットフォームドライバ led_pdrv.ko をロードする

7.6.3 開発ボードでアプリケーションを実行

ドライバコードを通じて、最終的に/dev 下に LED ライトデバイスが作成され、それは led0 としています。echo コマンドを使用して、LED ドライバが正常に動作しているかをテストできます。

以下のコマンドを使用してライトのオン/オフを制御します：

```
# ライトをオンにする
sudo sh -c "echo 0 > /dev/led0"
# ライトをオフにする
sudo sh -c "echo 1 > /dev/led0"
```

株式会社日昇テクノロジー

第 8 章 Linux デバイスツリー

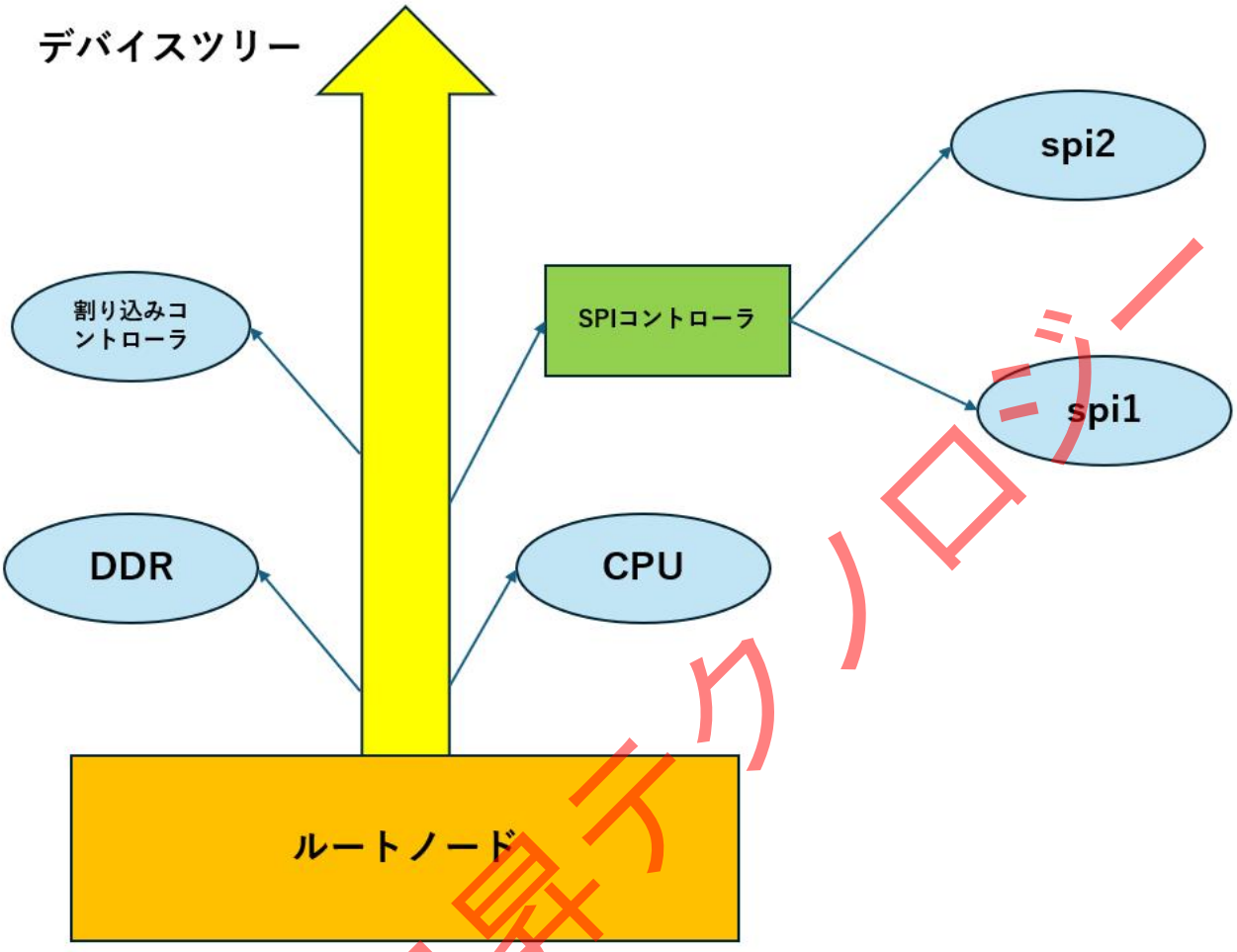
Linux 3.x バージョン以降でデバイスツリーが導入されました。デバイスツリーは、ハードウェアプラットフォームのボードレベルの詳細を記述するために使用されます。以前の Linux カーネルでは、これらの「ハードウェアプラットフォームのボードレベルの詳細」は Linux カーネルディレクトリの「/arch」に保存されていました。例えば ARM の場合、「ハードウェアプラットフォームのボードレベルの詳細」は「/arch/arm/plat-xxx」と「/arch/arm/mach-xxx」のディレクトリに保存されていました。プロセッサの数が増えるにつれて、「ハードウェアプラットフォームのボードレベルの詳細」を記述するファイルが増え続け、Linux カーネルが非常に肥大化していきました。Linux の父はこの問題を発見し、デバイスツリーを使ってこの問題を解決することにしました。デバイスツリーはシンプルで使いやすく、再利用性が高いため、Linux 3.x 以降ではほとんどの場合、ドライバを書く際にデバイスツリーが採用されています。

デバイスツリーについての詳細は、https://www.devicetree.org/ を参照してください。

8.1 デバイスツリーの概要

デバイスツリー（Device Tree）の役割は、ハードウェアプラットフォームのハードウェアリソースを記述することです。一般的に、動的に検出できないデバイスについて記述され、動的に検出できるデバイスは記述の必要がありません。デバイスツリーは、ブートローダー（uboot）からカーネルに渡すことができ、カーネルはデバイスツリーからハードウェア情報を取得することができます。

株式会社日昇テクノロジー



デバイスツリーでハードウェアリソースを記述する際の特徴は 2 つあります。

・第一に、「木構造」でハードウェアリソースを記述します。例えば、ローカルバスを「木」の「幹」として、デバイスツリー内では「ルートノード」と呼ばれ、ローカルバスに接続される IIC バス、SPI バス、UART バスは「枝」として、デバイスツリー内では「ルートノードの子ノード」と呼ばれます。IIC バスの下には複数の IIC デバイスがあり、これらの「枝」はさらに分岐することができます。ルートノード以外のノードはすべて 1 つの親ノードしか持たない。

・第二に、デバイスツリーのソースファイルは、ヘッダファイル（.h ファイル）のように、一つのデバイスツリーファイルが別のデバイスツリーファイルを参照することができます。これにより、「コード」の再利用が実現できます。例えば、複数のハードウェアプラットフォームが rk のプロセッサを主制御チップとして使用している場合、rk チップのハードウェアリソースを一つ

株式会社日昇テクノロジー

の独立したデバイスツリーファイル（通常は「.dtsi」の拡張子を使用）に書き込み、他のボードレベルのデバイスツリーファイルは「#include xxx.dtsi」を使用して直接参照することができます。DTS、DTC、DTB は文書でよく見られる略語です。

・DTS は、.dts フォーマットのファイルを指し、ASII テキストフォーマットで記述されたデバイスツリーです。これは、書く必要があるデバイスツリーのソースコードであり、一般的に 1 つの.dts ファイルが 1 つのハードウェアプラットフォームに対応しています。ARM アーキテクチャにおいては、ソースファイルは Linux ソースコードの「/arch/arm/boot/dts」ディレクトリに位置しています。

・DTC は、デバイスツリーソースコードをコンパイルするためのツールを指します。一般的に、このコンパイルツールは手でインストールする必要があります。

・DTB は、デバイスツリーソースコードからコンパイルされるファイルで、C 言語の「.C」ファイルが「.bin」ファイルにコンパイルされるのに似ています。

8.2 デバイスツリーのフレームワーク

デバイスツリー（Device Tree）は、名前付きのノード（node）とプロパティ（property）で構成されます。例えば、lubancat4 を例に、この章のコード linux_driver/device_tree/RK3588-lubancat4.dts またはカーネルソースコード/arch/arm64/boot/dts/rockchip/RK3588-lubancat4.dts を開いてみましょう。

以下の内容は、デバイスツリーのソースコードを中心に、デバイスツリーのフレームワークと基本構文を解説します。

株式会社日昇テクノロジー

リスト 1: デバイスツリー（カーネルソースコード

/arch/arm64/boot/dts/rockchip/RK3568-lubancat4.dts）

```
1 /dts-v1/;

2

3 #include <dt-bindings/gpio/gpio.h>

4 #include <dt-bindings/pwm/pwm.h>

5 #include <dt-bindings/pinctrl/rockchip.h>

6 #include <dt-bindings/input/rk-input.h>

7 #include <dt-bindings/display/drm_mipi_dsi.h>

8 #include <dt-bindings/sensor-dev.h>

9 #include "rk3568.dtsi"

10

11 / {

12 model = "EmbedFire Lubancat4 HDMI";

13 compatible = "embedfire,lubancat4", "rockchip,rk3568";

14

15 chosen: chosen {

16 bootargs = "earlycon=uart8250,mmio32,0xfe660000 console=ttyFIQ0
root=PARTUUID=614e0000-0000 rw rootwait";

17 };

18

19 fiq-debugger {
```

株式会社日昇テクノロジー

```
20 compatible = "rockchip,fiq-debugger";
21 rockchip,serial-id = <2>;
22 rockchip,wake-irq = <0>;
23 /* If enable uart uses irq instead of fiq */
24 rockchip,irq-mode-enable = <1>;
25 rockchip,baudrate = <1500000>; /* Only 115200 and 1500000 */
26 interrupts = <GIC_SPI 252 IRQ_TYPE_LEVEL_LOW>;
27 pinctrl-names = "default";
28 pinctrl-0 = <&uart2m0_xfer>;
29 status = "okay";
30 };
31 /*-------------略--------------*/
32 &saradc {
33 vref-supply = <&vcca_1v8>;
34 status = "okay";
35 };
36
37 &tsadc {
38 status = "okay";
39 };
40 /*-------------以下略--------------*/
41 }
```

株式会社日昇テクノロジー

リスト 2: RK3588.dtsi ヘッダファイル（カーネルソースコード

/arch/arm64/boot/dts/rockchip/RK3588.dtsi）

```
1 #include <dt-bindings/clock/rk3568-cru.h>

2 #include <dt-bindings/interrupt-controller/arm-gic.h>

3 #include <dt-bindings/interrupt-controller/irq.h>

4 #include <dt-bindings/pinctrl/rockchip.h>

5 #include <dt-bindings/soc/rockchip,boot-mode.h>

6 #include <dt-bindings/phy/phy.h>

7 #include <dt-bindings/power/rk3568-power.h>

8 #include <dt-bindings/soc/rockchip-system-status.h>

9 #include <dt-bindings/suspend/rockchip-rk3568.h>

10 #include <dt-bindings/thermal/thermal.h>

11 #include "rk3568-dram-default-timing.dtsi"

12

13 / {

14 compatible = "rockchip,rk3568";

15

16 interrupt-parent = <&gic>;

17 #address-cells = <2>;

18 #size-cells = <2>;

19

20 aliases {
```

株式会社日昇テクノロジー

```
21 csi2dphy0 = &csi2_dphy0;
22 csi2dphy1 = &csi2_dphy1;
23 csi2dphy2 = &csi2_dphy2;
24 dsi0 = &dsi0;
25 dsi1 = &dsi1;
26 ethernet0 = &gmac0;
27
28 /*-------------以下略 --------------*/
29
30 }
```

デバイスツリーソースコードは 3 部分に分かれており、以下の通りです：

• 第 3-9 行：ヘッダファイル。デバイスツリーは C 言語のように「#include」を使用して「.h」拡張子のヘッダファイル、またはデバイスツリー「.dtsi」拡張子のヘッダファイルを参照できます。RK3588.dtsi は Rockchip 公式に提供される RK3588 プラットフォームの「共用」デバイスツリーファイルです。

• 第 11-30 行：デバイスツリーノード。デバイスツリーがに与える最も直感的な印象は、いくつかのネストされた中括弧「{}」で構成されていることです。各「{}」は一つの「ノード」です。「/ {…};」は「ルートノード」を示し、各デバイスツリーには一つのルートノードのみがあります。もし「RK3588.dtsi」ファイルを開けば、それにもルートノードがあることがわかりますが、「RK3588-lubancat4.dts」が「RK3588.dtsi」ファイルを参照しているからと言って、「RK3588-lubancat4.dts」デバイスツリーに 2 つのルートノードがあるわけではありません。なぜなら異なるファイルのルートノードは最終的に一つに合わされるからです。ルートノード内の「aliases {…}」、「chosen {…}」、「memory {…}」などはすべてルートノードの子ノードです。

株式会社日昇テクノロジー

• 第 32-39 行：デバイスツリーノードへの追加内容。第三部分の子ノードは、ルートノード下の子ノードより「&」が一つ多くなっています。これは、既存の子ノードにデータを追加することを意味します。これら「既に存在するノード」は、「RK3588-lubancat4.dts」ファイルや「RK3588.dtsi」ファイルに含まれるデバイスツリーファイルで定義されている可能性があります。「RK3588-lubancat4.dts」コード中の「&cpu0 {…}」、「&dmc {…}」、「&i2c0 {…}」などの追加対象ノードは、「stm32mp157c.dtsi」中で定義されています。

これまでに、デバイスツリーは一つのルートノードと多くの子ノードで構成されており、子ノードはさらに他のノードを含むことができる、つまり子ノードの子ノードを含むことがわかりました。デバイスツリーの構成は非常にシンプルです。次に、ノードの基本形式とノード属性について見ていきましょう。

8.2.1 ノードの基本形式

ノードの構造は次のように参照されます：

株式会社日昇テクノロジー

```
/dts-v1;  // 必要なDTSファイルバージョンの説明
#include "example.dtsi";  // ヘッダーファイルを含む、.dtsi .dts .hファイルなどが可能
/ {
    node1-name@unit-address{
        compatible = "xxx,xxx";
        a-string-property = "A string";  // ノードの属性と属性値、文字列です
        a-string-list-property = [0x00 0x13 0x24 0x36]

        label: child-node1 {
            first-child-property;
            second-child-property = <1>;
            a-string-property = "Hello,world";

        };      // ノード1の子ノード1 "label"はラベルです
        child-node2 {

        };      // 子ノード2

    };      // ノード1 名前は"node1-name" 単位アドレスとreg属性の最初のアドレスが一致
    node2-name {  // ノード2
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* 各数値（セル）はuint32です */
        child-node1 {
            my-cousin = <&cousin>;

        };    /* ノード2の子ノード */

    };
    ...
};     // ルートノード
```

ノード名 node-name

ノード形式の node-name はノードの名前を指定するために使用されます。長さは 1 から 31 文字で、以下の文字で構成されている必要があります：

ノード名のリスト

文字	説明
0-9	数字
a-z	小文字
A-Z	大文字
,	英語のカンマ
.	英語のピリオド
_	アンダースコア
•	プラス記号
•	マイナス記号

株式会社日昇テクノロジー

また、ノード名は大文字または小文字のアルファベットで始まり、デバイスのカテゴリーを説明できるものであるべきです。

注：ルートノードにはノード名がありません。それは「/」を使用してこれがルートノードであることを直接示します。

@unit-address

@unit-address では、「@」記号は区切り文字として理解でき、「unit-address」は「ユニットアドレス」を指定するために使用され、その値はノードの「reg」属性の最初のアドレスと一致する必要があります。ノードに「reg」属性の値がない場合は、「@unit-address」を省略できますが、この場合、同じレベルのデバイスツリー内（同じレベルの子ノード）でノード名が一意である必要があります。これは、同じレベルの子ノードのノード名が同じであっても、「ユニットアドレス」が異なれば、node-name@unit-address の全体が同じレベルで一意である必要があることを意味します。

8.2.2 ノードラベル

RK3588.dtsi ヘッダファイルでは、「cpu0」の前に「cpu@0」があります。この「cpu0」はノードラベルとして言及されています。通常、ノードラベルはノード名の略称であり、他の場所で参照が必要な場合にノードラベルを使用してそのノードに追加内容を追加することができます。

8.2.3 ノードパス

ルートノードから必要なノードへの完全なパスを指定することにより、デバイスツリー内のノードを一意に識別することができます。異なるレベルのデバイスツリーノードの名前は同じである可能性がありますが、同じレベルのデバイスツリーノードは一意でなければなりません。これは、Windows のファイルシステムと似ています。一つのパスは一つのファイルまたはフォルダを一意に識別します。異なるディレクトリの下のファイル名は同じである可能性があります。たとえば、前のノード構造の参照図では、node1-name の子ノード child-node1 のノードパスは「/node1-name/child-node1」に

株式会社日昇テクノロジー

なります。

8.2.4 ノード属性

ノードの「{}」内に含まれる内容はノード属性であり、通常、一つのノードには複数の属性情報が含まれています。これらの属性情報はカーネルに「ボードレベルのハードウェア記述情報」を伝達するためのもので、ドライバではこれらの情報を取得するためにいくつかの API 関数が使用されます。たとえば、ルートノード「/」には属性 compatible = "rockchip,RK3588"があります。この属性から、ハードウェアデバイスの名前が「RK3588」であり、「RK3588」この SOC を使用していることがわかります。

デバイスツリーを書く際の主要な内容は、ノードの属性を記述することです。通常、一つのノードが一つのデバイスを表し、どのような属性があり、それらの属性をどのように記述し、ドライバ内でこれらの属性をどのように参照するかが、後に詳しく説明される重点です。このセクションでは、デバイスノードがどのような設定可能な属性を持っているかのみを説明します。いくつかのノード属性はすべてのノードで共通ですが、一部は特定のノードにのみ適用されます。ここでは、それらの共通属性を紹介します。その他のノード属性は使用時に詳しく説明されます。

ノード属性には、標準属性とカスタム属性があります。つまり、デバイスツリーで自分の実際のニーズに応じてデバイス属性を定義・追加することができます。標準属性の属性名は固定されており、カスタム属性名は要件に応じて自由に定義できます。

compatible 属性

属性値タイプ：文字列

株式会社日昇テクノロジー

リスト 3: compatible 属性

```
1 model = "EmbedFire Lubancat4 HDMI";

2 compatible = "embedfire,lubancat4", "rockchip,rk3568";

3

4 aliases {

5 csi2dphy0 = &csi2_dphy0;

6 csi2dphy1 = &csi2_dphy1;

7 csi2dphy2 = &csi2_dphy2;

8 dsi0 = &dsi0;

9 dsi1 = &dsi1;

10 ethernet0 = &gmac0;

11 ethernet1 = &gmac1;

12  GPIO1 = & GPIO1;

13 .....

14 };
```

compatible 属性の値は、一つまたは複数の文字列で構成され、複数ある場合は「,」で区切られます。デバイスツリー内の各デバイスを表すノードには compatible 属性が必要です。この属性はシステムがデバイスにバインドするデバイスドライバーを決定するための鍵です。ノードの検索方法として compatible 属性の他に、ノード名やノードパスを通して特定のノードを見つけることもできます。

例えば、システム初期化時に platform バス上のデバイスが初期化される際、デバイスノードの「compatible」属性とドライバ内の of_match_table に対応する値を照合して、一致する場合は対応するドライバがロードされます。

model 属性

株式会社日昇テクノロジー

属性値の型：文字列

例：

リスト 4: model 属性

```
1 model = "EmbedFire Lubancat4 HDMI+MIPI";
```

model 属性はデバイスの製造元とモデルを指定するために使用され、推奨されるフォーマットは「製造元, モデル」ですが、カスタマイズも可能です。

status 属性

属性値の型：文字列

例：

リスト 5: status 属性

```
1 /* 外部サウンドカード */
2 sound: sound {
3     status = "disabled";
4 };
```

status 属性はデバイスの「動作状態」を示し、デバイスの無効化または有効化に使用されます。利用可能な動作状態は以下の表に記載されています。デフォルトでは status 属性が設定されていない場合、デバイスは有効です。

表 1: ノード名

状態値	説明
okay	装置を有効にする
disabled	装置を無効にする
fail	装置が動作しない、現在のドライバーがサポートしていない、修正待ちを意味します。
fail-sss	装置が動作しない、現在のドライバーがサポートしていない、修正待ちを意味します。「sss」の値は特定の装置に依存します。

株式会社日昇テクノロジー

#address-cells と #size-cells

属性値の型：u32

例：

リスト 6: #address-cells と #size-cells

```
1 soc {
2 #address-cells = <1>;
3 #size-cells = <1>;
4 compatible = "simple-bus";
5 interrupt-parent = <&gpc>;
6 ranges;
7 ocrams: sram@900000 {
8 compatible = "fsl,lpm-sram";
9 reg = <0x900000 0x4000>;
10 };
11 };
```

#address-cells と #size-cells の属性は一緒に存在し、デバイスツリーの構造内で子ノードがあるデバイスノードに使用され、子ノードの「reg」属性の「記述フォーマット」を設定するために使用されます。

補足：reg 属性の値は数字の連続で、例えば reg = <0x900000 0x4000>のように、reg 属性の記述フォーマットは reg = <cells cells cells cells cells cells...>となり、実際の長さは状況に応じて異なります。これらのデータはアドレスデータ（アドレスフィールド）と長さデータ（サイズフィールド）に分かれます。

#address-cells は、子ノードの reg 属性の「アドレスフィールド」が占める長さ（セルの数）を指

株式会社日昇テクノロジー

定します。#size-cells は、子ノードの reg 属性の「サイズフィールド」が占める長さ（セルの数）を指定します。

例えば、#address-cells=2, #size-cells=1 の場合、reg 内のデータの意味は reg = <address address size address address size>になります。これは、各セルが 32 ビット幅の数字であるため、64 ビット幅のアドレスを表現するには 2 つの address セルを使用する必要があることを意味します。一方、#address-cells=1, #size-cells=1 の場合、reg 内のデータの意味は reg = <address size address size address size>になります。

要するに、#size-cells と #address-cells は子ノードの reg 属性内のどのデータが「アドレス」で、どのデータが「長さ」の情報であるかを決定します。

reg 属性

属性値の型：アドレス、長さのペア

reg 属性は、デバイスリソースが親バスが定義するアドレス空間内のアドレスを記述します。通常は、一つのレジスタの開始アドレス（オフセットアドレス）と長さを示すために使用されますが、特定の状況では異なる意味を持ちます。例えば、#address-cells = <1>, #size-cells = <1>の場合、reg = <0x9000000 0x4000>では、0x9000000 がアドレスを示し、0x4000 がアドレスの長さを示します。ここでの reg 属性は、開始アドレスが 0x9000000 で、長さが 0x4000 のアドレス空間を指定しています。

ranges 属性

属性値の型：任意数の<子アドレス、親アドレス、アドレス長さ>のコード

例：

株式会社日昇テクノロジー

リスト 7: ranges 属性

```
1 soc {
2    #address-cells = <1>;
3    #size-cells = <1>;
4    compatible = "simple-bus";
5    interrupt-parent = <&gpc>;
6    ranges;
7
8    busfreq {
9       /*-------------以下略--------------*/
10    };
11 }
```

この属性は、子ノードのアドレス空間と親ノードのアドレス空間のマッピング（変換）方法を提供します。一般的なフォーマットは ranges = <子アドレス, 親アドレス, 変換長さ>です。親アドレス空間と子アドレス空間が同じ場合は変換が不要で、例で示されるように、ranges を空にすることができます。この場合、ranges 属性を省略することもできます。

例えば、#address-cells と #size-cells が 1 の場合、ranges=<0x0 0x10 0x20>は、子アドレス空間の 0x0～(0x0 + 0x20)を親アドレス空間の 0x10～(0x10 + 0x20)にマッピングすることを意味します。

name と device_type

属性値の型：文字列

例：

株式会社日昇テクノロジー

リスト 8: name 属性

```
1 example{
2     name = "name"
3 }
```

リスト 9: device_type 属性

```
1 cpus {
2 #address-cells = <2>;
3 #size-cells = <0>;
4
5 cpu0: cpu@0 {
6 device_type = "cpu";
7 compatible = "arm,cortex-a55";
8 reg = <0x0 0x0>;
9 enable-method = "psci";
10 clocks = <&scmi_clk 0>;
11 operating-points-v2 = <&cpu0_opp_table>;
12 cpu-idle-states = <&CPU_SLEEP>;
13 #cooling-cells = <2>;
14 dynamic-power-coefficient = <187>;
15 };
16 ...
17 }
```

株式会社日昇テクノロジー

これら 2 つの属性はあまり使われず（廃止された）、使用されることは推奨されません。name は

ノード名を指定するために古いデバイスツリーで使用されましたが、現在使用されているデバイスツリーでは廃止されています。device_type 属性もほとんど使用されない属性で、CPU とメモリのノードにのみ使用されます。上の例では、CPU ノードに device_type が使用されています。

8.2.5 ノード内容の追加/変更

リスト 10: ノード内容の追加/変更

```
1 &cpu0 {
2     cpu-supply = <&vdd_cpu>;
3 };
```

これらのソースコードはルートノード「/{…}」内に含まれておらず、新しいノードではなく、既存のノードに内容を追加しています。上記のソースコードの例では、「&cpu0」は「cpu0」という「ノードラベル」のノードにデータを追加することを意味しています。このノードは、このファイル内か、またはこのファイルが含むデバイスツリーファイル内に定義されている可能性があります。この例では、「cpu0」は「RK3588.dtsi」ファイル内で定義されています。

8.2.6 特別なノード

aliases 子ノード

aliases 子ノードの役割は、他のノードに別名をつけることです。以下のように示されます。

リスト 11: 別名子ノード

```
1 aliases {
2     csi2dphy0 = &csi2_dphy0;
3     csi2dphy1 = &csi2_dphy1;
4     csi2dphy2 = &csi2_dphy2;
```

株式会社日昇テクノロジー

```
5    /*---------- 省略----------*/
6    mmc0 = &sdhci;
7    mmc1 = &sdmmc0;
8    mmc2 = &sdmmc1;
9    mmc3 = &sdmmc2;
10   serial0 = &uart0;
11   serial1 = &uart1;
12   serial2 = &uart2;
13   /*---------- 以下略----------*/
14 }
```

例えば、「serial0 = &uart0;」では、「serial0」はノード名であり、「serial0」を別名として設定することで「uart0」ノードを指し示すことができます。これはノードラベルに似ています。デバイスツリー内ではノードにラベルを追加することが多く、ノード別名の使用は少ないです。別名の役割は「デバイスツリーノードを素早く見つける」ことです。ドライバ内でノードを探す場合、通常は「ノードパス」を使用して一歩一歩ノードを見つけますが、別名を使用すると「一歩で」ノードを見つけることができます。

chosen 子ノード

chosen 子ノードはルートノードの下にあり、以下のように示されます。

リスト 12: chosen 子ノード

```
1 chosen {
2    bootargs = "earlycon=uart8250,mmio32,0xfe660000 console=ttyFIQ0
root=PARTUUID=614e0000-0000 rw rootwait";
3 };
```

株式会社日昇テクノロジー

chosen 子ノードは実際のハードウェアを表すものではなく、カーネルにパラメータを渡すために使用されます。このノードはまた、uboot から Linux カーネルに設定パラメータを渡す「チャネル」としても機能します。Uboot で設定されたパラメータはこのノードを通じてカーネルに渡されます。これは uboot とカーネルが自動的に行う処理であり、初心者としては深く掘り下げる必要はありません。

割り込みやクロックなどの部分にも独自のノード標準属性があり、学習が進むにつれてこれらのノード標準属性について詳しく説明されます。

8.3 デバイスツリーノード情報の取得

デバイスツリー内の「ノード」は実際のハードウェア内のデバイスに対応しています。デバイスツリーに「led」ノードを追加しましたが、通常、このノードから LED ドライバの作成に必要なすべての情報、例えば LED の関連制御レジスタアドレス、LED のクロック制御レジスタアドレスなどを取得できます。

このセクションでは、デバイスツリーのデバイスノードから必要なデータを取得する方法について学びます。カーネルはデバイスノードからリソース（デバイスノードに定義された属性）を取得するために使用される一連の関数を提供しており、これらの関数は of_で始まり、OF 操作関数と呼ばれます。一般的に使用される OF 関数については次で説明します：

8.3.1 ノード検索関数

8.3.1.1 ノードパスによるノード検索関数

リスト 13: of_find_node_by_path 関数（カーネルソースコード/include/linux/of.h）

```
1 struct device_node *of_find_node_by_path(const char *path)
```

パラメーター：

- path：デバイスツリー内の指定されたノードへのパス。

戻り値：

株式会社日昇テクノロジー

- device_node：構造体ポインター。検索に失敗した場合は NULL を返し、そうでなければ device_node 型の構造体ポインターを返します。これにはデバイスノードの情報が含まれています。

device_node 構造体は以下のようになります。

リスト 14: device_node 構造体

```
1 struct device_node {
2 const char *name;
3 const char *type;
4 phandle phandle;
5 const char *full_name;
6 struct fwnode_handle fwnode;
7
8 struct property *properties;
9 struct property *deadprops; /* removed properties */
10 struct device_node *parent;
11 struct device_node *child;
12 struct device_node *sibling;
13 #if defined(CONFIG_OF_KOBJ)
14 struct kobject kobj;
15 #endif
16 unsigned long _flags;
```

株式会社日昇テクノロジー

```
17 void *data;
18 #if defined(CONFIG_SPARC)
19 const char *path_component_name;
20 unsigned int unique_id;
21 struct of_irq_controller *irq_trans;
22 #endif
23 };
```

- name：name 属性の値
- type：device_type 属性の値
- full_name：ノードの名前。device_node 構造体の後ろに文字列を置き、full_name はそれを指します
- properties：リスト。そのノードのすべての属性を繋ぎます
- parent：親ノードへのポインター
- child：子ノードへのポインター
- sibling：兄弟ノードへのポインター

device_node 構造体を取得した後、他の of 関数を使用してノードの詳細情報を取得することができます。

8.3.1.2 ノード名によるノード検索関数

リスト 15: of_find_node_by_name 関数（カーネルソースコード/include/linux/of.h）

```
1 struct device_node *of_find_node_by_name(struct device_node *from, const char *name);
```

パラメーター：

株式会社日昇テクノロジー

- from：どのノードから検索を開始するかを指定します。それ自体は検索範囲に含まれず、その後のノードのみが検索されます。NULL に設定するとルートノードから検索が始まります。

- name：検索するノードの名前。

戻り値：

- device_node：構造体ポインター。検索に失敗した場合は NULL を返し、そうでなければ device_node 型の構造体ポインターを返します。

8.3.1.3 ノードタイプによるノード検索関数

リスト 16: of_find_node_by_type 関数（カーネルソースコード/include/linux/of.h）

```
1 struct device_node *of_find_node_by_type(struct device_node *from, const char *type)
```

パラメーター：

- from：どのノードから検索を開始するかを指定します。それ自体は検索範囲に含まれず、その後のノードのみが検索されます。NULL に設定するとルートノードから検索が始まります。

- type：検索するノードのタイプ。これは device_node->type となります。

戻り値：

- device_node：構造体ポインター。検索に失敗した場合は NULL を返し、そうでなければ device_node 型の構造体ポインターを返します。

8.3.1.4 ノードタイプと compatible 属性によるノード検索関数

リスト 17: of_find_compatible_node 関数（カーネルソースコード/include/linux/of.h）

```
1 struct device_node *of_find_compatible_node(struct device_node *from, const char *type, const char *compatible)
```

of_find_node_by_name 関数に compatible 属性をフィルターとして追加しました。

パラメーター：

株式会社日昇テクノロジー

- from：どのノードから検索を開始するかを指定します。それ自体は検索範囲に含まれず、その後のノードのみが検索されます。NULL に設定するとルートノードから検索が始まります。

- type：検索するノードのタイプ。

- compatible：検索するノードの compatible 属性。

戻り値：

- device_node：構造体ポインター。検索に失敗した場合は NULL を返し、そうでなければ device_node 型の構造体ポインターを返します。

8.3.1.5 マッチングテーブルによるノード検索関数

リスト 18: of_find_matching_node_and_match 関数（カーネルソースコード/include/linux/of.h）

```
1 static inline struct device_node *of_find_matching_node_and_match(struct device_node *from, const struct of_device_id *matches, const struct of_device_id **match)
```

より多くのマッチングパラメーターを含む構造体を示しており、前述の 3 つのノード検索関数と比べて、この関数はより多くのパラメーターに基づいてノードのフィルタリングを行い、より詳細な選択が可能です。パラメーター match は検索結果を示します。

パラメーター：

• from：どこから検索を開始するかを指定します。自身は検索対象に含まれず、その後のノードのみが検索されます。NULL に設定された場合は、ルートノードから検索が始まります。

• matches：マッチングテーブル。このテーブルに一致するデバイスノードを検索します。

• of_device_id：次のような構造体です。

戻り値：

• device_node：device_node 型の構造体ポインターで、取得したノードを保存します。失敗した場合は NULL を返します。

株式会社日昇テクノロジー

リスト 19: of_device_id 構造体

```
1 /*
2 * デバイスのマッチングに使用される構造体
3 */
4
5 struct of_device_id {
6     char name[32];
7     char type[32];
8     char compatible[128];
9     const void *data;
10 };
```

- name：name 属性の値
- type：device_type 属性の値
- compatible：compatible 属性の値
- data：そのノードの全ての属性をつなぐリスト

8.3.1.6 親ノード検索関数

リスト 20: of_get_parent 関数（カーネルソースコード/include/linux/of.h）

```
1 struct device_node *of_get_parent(const struct device_node *node)
```

パラメーター：

- node：親ノードを検索したいノードを指定します。

戻り値：

株式会社日昇テクノロジー

• device_node：device_node 型の構造体ポインターで、取得したノードを保存します。失敗した場合は NULL を返します。

8.3.1.7 子ノード検索関数

リスト 21: of_get_next_child 関数（カーネルソースコード/include/linux/of.h）

```
1 struct device_node *of_get_next_child(const struct device_node *node, struct device_node *prev)
```

パラメーター：

• node：子ノードを検索したいノードを指定します。

• prev：前の子ノード。これは反復検索のプロセスで、例えば 2 番目の子ノードを検索する場合は、ここに 1 番目の子ノードを指定します。NULL を指定すると、最初の子ノードを検索します。

戻り値：

• device_node：device_node 型の構造体ポインターで、取得したノードを保存します。失敗した場合は NULL を返します。

ここで紹介された 7 つのノード検索関数は、共通の特徴があります—戻り値の型が同じです。ノードが見つかれば、そのノードに対応する device_node 構造体が返されます。ドライバプログラムでは、この device_node を使用してデバイスノードの属性情報を取得し、親ノードや子ノードを調べることができます。最初の関数 of_find_node_by_path は、他の 6 つとは異なり、ノードパスを使用してノードを検索します。これは、デバイスツリーソースファイル(.dts)から得られる「ノードパス」です。中間の 4 つの関数は、特定のノードの属性に基づいて、あるノードの後に条件を満たすデバイスノードを検索します。この「あるノード」とは、既に見つかっている device_node（デバイスノード構造体）を意味します。最後の 2 つの関数は中間の 4 つと似ていますが、ノードの属性を使用せずに、親子関係に基づいて検索します。

株式会社日昇テクノロジー

8.3.2 属性値の取得に関する of 関数

前節では、7 つのノード検索関数について説明しました。これらは共通の特徴を持ち、デバイスノードを見つけると、そのデバイスノードに対応する構造体ポインター(device_node*)を返します。このプロセスは、デバイスツリーのデバイスノードをドライバに「取得」すると理解できます。取得に成功した後、デバイスノード構造体(device_node)から必要なデバイスノード属性情報を取得するために、別の of 関数グループを使用します。

8.3.2.1 ノード属性検索関数

リスト 22: of_find_property 関数（カーネルソースコード/include/linux/of.h）

```
1 struct property *of_find_property(const struct device_node *np, const char *name, int *lenp)
```

パラメーター：

- np：どのデバイスノードの属性情報を取得したいかを指定します。
- name：属性名。
- lenp：取得した属性値のサイズを出力するためのポインター。このパラメーターは出力パラメーターで、取得した属性の実際のサイズが「持ち帰られる」値です。

戻り値：

- property：取得した属性。property 構造体は、ノード属性構造体と呼ばれ、以下のように示されます。失敗した場合は NULL を返します。この構造体から、必要な属性値を取得することができます。

株式会社日昇テクノロジー

リスト 23: property 属性構造体

```
1 struct property {
2     char *name;
3     int length;
4     void *value;
5     struct property *next;
6     #if defined(CONFIG_OF_DYNAMIC) || defined(CONFIG_SPARC)
7     unsigned long _flags;
8     #endif
9     #if defined(CONFIG_OF_PROMTREE)
10     unsigned int unique_id;
11     #endif
12     #if defined(CONFIG_OF_KOBJ)
13     struct bin_attribute attr;
14     #endif
15 };
```

- name：属性名
- length：属性の長さ
- value：属性値
- next：次の属性

株式会社日昇テクノロジー

8.3.2.2 整数属性の読み取り関数

整数属性の読み取りには、8、16、32、64 ビットデータを読み取る一連の関数があります。

リスト 24: of_property_read_uX_array 関数群（カーネルソースコード/include/linux/of.h）

```
1 //8 ビット整数読み取り関数
2 int of_property_read_u8_array(const struct device_node *np, const char *propname, u8 *out_values,
size_t sz)
3
4 //16 ビット整数読み取り関数
5 int of_property_read_u16_array(const struct device_node *np, const char *propname, u16 *out_values,
size_t sz)
6
7 //32 ビット整数読み取り関数
8 int of_property_read_u32_array(const struct device_node *np, const char *propname, u32 *out_values,
size_t sz)
9
10 //64 ビット整数読み取り関数
11 int of_property_read_u64_array(const struct device_node *np, const char *propname, u64
  *out_values, size_t sz)
```

株式会社日昇テクノロジー

パラメーター：

- np：読み取りたいデバイスノード構造体を指定します。
- propname：デバイスノードのどの属性を取得したいかを指定します。
- out_values：これは出力パラメーターで、読み取ったデータを保存する「戻り値」です。

- sz：入力パラメーターで、読み取りたい長さを設定します。

戻り値：

- 成功すると 0 が返され、エラーが発生した場合はエラーステータスコード（非ゼロ値）が返されます。-EINVAL（属性が存在しない）、-ENODATA（読み取るデータがない）、-EOVERFLOW（属性値リストが小さい）など。

8.3.2.3 整数属性読み取り関数の簡略版

こちらの関数は整数属性を読み取る関数の単純なラッパーで、読み取り長さを 1 に設定します。使い方は読み取り関数と完全に同じで、ここでは詳細は割愛します。

リスト 25: of_property_read_uX 関数群（カーネルソースコード/include/linux/of.h）

```
1 //8 ビット整数読み取り関数
2 int of_property_read_u8(const struct device_node *np, const char *propname, u8 *out_values)
3
4 //16 ビット整数読み取り関数
5 int of_property_read_u16(const struct device_node *np, const char *propname, u16 *out_values)
6
7 //32 ビット整数読み取り関数
8 int of_property_read_u32(const struct device_node *np, const char *propname, u32 *out_values)
9
10 //64 ビット整数読み取り関数
11 int of_property_read_u64(const struct device_node *np, const char *propname, u64 *out_values)
```

文字列属性読み取り関数

デバイスノードには compatible、status、type などの多くの文字列属性が存在します。これらの属

株式会社日昇テクノロジー

性はノード属性検索関数 of_find_property を使って取得することができますが、それは比較的面倒です。カーネルは文字列属性を読み取るための関数群を提供しており、次で紹介します：

リスト 26: of_property_read_string 関数（カーネルソースコード/include/linux/of.h）

```
1 int of_property_read_string(const struct device_node *np, const char *propname, const char
**out_string)
```

パラメーター：

- np：特定のデバイスノードの属性情報を取得したい場合に指定します。
- propname：属性名。
- out_string：取得された文字列ポインター。これは「出力」パラメーターで、文字列属性の先頭アドレスを返します。このアドレスはメモリ内の「属性値」の実際の場所を指し、このアドレスを操作することで全ての文字列属性（複数の文字列が連続してメモリに格納され、'0'で区切られている）を取得できます。

戻り値：

- 成功時は 0 を返し、失敗時はエラーステータスコードを返します。

リスト 27: of_property_read_string_index 関数（カーネルソースコード/include/linux/of.h）

```
1 int of_property_read_string_index(const struct device_node *np, const char *propname, int index,
const char **out_string)
```

前述の関数に加えて、パラメーター index が追加されています。これは、属性値内の複数の文字列のうち、何番目の文字列を読み取るかを指定するために使用します。index は 0 から始まります。最初の関数では属性値のアドレス（つまり最初の文字列のアドレス）のみを取得できますが、他の文字列を取得するにはアドレスを手で移動させる必要があり、非常に手間がかかります。そのため、2 番目の関数が推奨されます。

株式会社日昇テクノロジー

ブール型属性を読み取る関数

デバイスノードにはいくつかの属性が BOOL 型であり、当然ながらカーネルは BOOL 型属性を読み取る関数を提供しています。以下に紹介します：

リスト 28: of_property_read_string_index 関数（カーネルソースコード/include/linux/of.h）

```
1 static inline bool of_property_read_bool(const struct device_node *np, const char *propname);
```

パラメーター：

- np：特定のデバイスノードの属性情報を取得したい場合に指定します。

- propname：属性名。

戻り値：

この関数は、通常のパターンに従わず、単に特定のブール型属性が存在するか否かを判断します。属性の値を直接取得することはできません。値を取得したい場合は、以前説明した「万能」関数 of_find_property を使用してノード属性を検索することができます。

8.3.3 メモリマッピング関連の of 関数

デバイスツリーのデバイスノードには、通常、reg 属性などのいくつかのメモリ関連の属性が含まれています。通常、レジスタアドレスを取得した後、ioremap 関数を使用して物理アドレスを仮想アドレスに変換する必要があります。現在、カーネルは of 関数を提供しており、物理アドレスから仮想アドレスへの変換を自動で完了します。以下に紹介します：

リスト 29: of_iomap 関数（カーネルソースコード/drivers/of/address.c）

```
1 void __iomem *of_iomap(struct device_node *np, int index)
```

パラメーター：

- np：特定のデバイスノードの属性情報を取得したい場合に指定します。

- index：通常、reg 属性は複数のセグメントを含むため、どのセグメントをマッピングするかを

株式会社日昇テクノロジー

指定します。インデックスは 0 から始まります。

戻り値：

- 成功時には変換されたアドレスを返し、失敗時には NULL を返します。

カーネルは、通常のアドレス取得用の of 関数も提供しており、これらの関数で得られる値は、デバイスツリーで設定したアドレス値そのものです。以下に紹介します：

リスト 30: of_address_to_resource 関数（カーネルソースコード/drivers/of/address.c）

```
1 int of_address_to_resource(struct device_node *dev, int index, struct resource *r)
```

パラメーター：

- np：特定のデバイスノードの属性情報を取得したい場合に指定します。

- index：通常、reg 属性は複数のセグメントを含むため、どのセグメントをマッピングするかを指定します。インデックスは 0 から始まります。

- r：これは resource 構造体で、取得されたアドレス情報を返す「出力」パラメーターです。

戻り値：

- 成功時は 0 を返し、失敗時はエラーステータスコードを返します。

リスト 31: resource 属性構造体

```
1 struct resource {
2     resource_size_t start;
3     resource_size_t end;
4     const char *name;
5     unsigned long flags;
6     unsigned long desc;
7     struct resource *parent, *sibling, *child;
8 };
```

株式会社日昇テクノロジー

- start：開始アドレス

- end：終了アドレス

- name：属性名

これらは、デバイスツリーからデバイス情報を取得するための基本的な of 関数の紹介です。これらはほとんどのニーズを満たすことができます。他の of 関数については、必要に応じて詳しく説明します。

ここでは、of 関数の 3 つの主要なカテゴリを紹介しました。これらはほとんどのニーズを満たすことができますが、他の of 関数については、必要に応じて後で詳しく説明します。

8.4 デバイスツリーにデバイスノードを追加する実験

8.4.1 実験説明

通常、ゼロからデバイスツリーを書くことはありません。なぜなら、機能的に完全なデバイスツリーは通常かなり大きく、例えばこのチュートリアルで参照されている Rockchip 公式の「RK3588.dtsi」デバイスツリーは数千行に及びます。さらに、公式がすでに主要な部分を書いているため、我々は公式のデバイスツリーを引用し、自分の実際の状況に応じて変更するだけで済むのです。

このセクションの実験では、LubanCat_RK の公式のボードを使用し、例として lubancat4 を使用します。ボードのシステムは ubuntu20.04 で、デバイスツリーファイルは「RK3588-lubancat4.dts」です。

実験中に「Permission denied」や類似のメッセージが表示された場合は、ユーザー権限に注意してください。ほとんどのハードウェアデバイスの操作には root ユーザーの権限が必要であり、コマンドを実行する前に「sudo」を追加するか、root ユーザーとしてプログラムを実行することが簡単な解決策です。

株式会社日昇テクノロジー

8.4.2 コード解説

この章のサンプルコードのディレクトリは「linux_driver/device_tree」です。

実際の応用では、デバイスノードに新しいノードを追加したり、既存のデバイスノードにデータを追加したり、デバイスツリープラグインを書いたりすることが最も一般的な操作です。

以前の説明によると、lubancat4 はデフォルトで「kernel/arch/arm64/boot/dts/rockchip/RK3588-lubancat4.dts」デバイスツリーを使用しています。このデバイスツリーで、デバイスノードを追加してみます。以下に示すように。

リスト 32: RK3588-lubancat4.dts に子ノード led_test を追加

```
1 /*
2 * Copyright (C) 2022 - All Rights Reserved by
3 * EmbedFire LubanCat
4 */
5
6 /dts-v1/;
7
8 #include <dt-bindings/gpio/gpio.h>
9 #include <dt-bindings/pwm/pwm.h>
10 #include <dt-bindings/pinctrl/rockchip.h>
11 #include <dt-bindings/input/rk-input.h>
12 #include <dt-bindings/display/drm_mipi_dsi.h>
13 #include <dt-bindings/sensor-dev.h>
14 #include "RK3588.dtsi"
```

株式会社日昇テクノロジー

```
15 / {
16     model = "EmbedFire Lubancat4 HDMI";
17     compatible = "embedfire,lubancat4", "rockchip,RK3588";
18
19     /*.....................*/
20
21     /* led_test ノードを追加 */
22     get_dts_info_test: get_dts_info_test{
23         compatible = "get_dts_info_test";
24         #address-cells = <1>;
25         #size-cells = <1>;
26
27         led@0x0xfd5f8000{ // GPIO1 のベースアドレスは 0xfd5f8000
28             compatible = "fire,led_test";
29             reg = <0xfd5f8000 0x00000100>;
30             status = "okay";
31         };
32     };
33
34     /*.....................*/
35 };
```

RK3588-lubancat4.dts デバイスツリーファイルに「led_test」という名前の新しいノードを追加しました。このノードにはいくつかの基本属性のみを追加しました。ここでは、デバイスノードの追加方

株式会社日昇テクノロジー

法を学ぶことが目的です。

このコードでは、led_test ノードに対して#address-cells = <1>、#size-cells = <1>が設定されており、これはその子ノードの reg 属性内のデータが「アドレス」と「長さ」の交互の形式であることを意味しています。

ledノードの子ノードの第二部分では、compatible、reg、statusの3つの属性が定義されています。これらの属性は「ノード属性」の章で既に紹介されています。rgb属性に注目する必要があり、親ノードで#address-cells = <1>、#size-cells = <1>が設定されているため、ここでの0xfd5f8000 はアドレス（GPIO1コントロールレジスタのベースアドレスとして記述されています）を表し、0x00000100はその長さを表しています。「led@0xfd5f8000」内のアドレス0xfd5f8000はreg属性の最初のアドレスと一致している必要があります。

ヒント: 他ののボードは、ドライバの章の実験環境構築のセクションでボードが使用するデバイスツリーファイルを確認し、それに従って修正や追加を行ってください。

カーネルのデバイスツリーをコンパイル：

カーネルをコンパイルする際にデバイスツリーも自動的にコンパイルされますが、カーネルのコンパイルは時間がかかるため、以下のコマンドを使用してデバイスツリーのみをコンパイルすることを推奨します：

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- lubancat4_defconfig //RK358x の設定ファイルを例としています
make ARCH=arm64 -j4 CROSS_COMPILE=aarch64-linux-gnu- dtbs
```

コンパイル成功後、生成されたデバイスツリーファイル(.dtb)はソースコードディレクトリのarch/arm64/boot/dts/rockchip/にあり、「RK3588-lubancat4.dtb」というファイル名です。

株式会社日昇テクノロジー

8.4.3 実験結果

8.4.3.1 デバイスツリーのロード

SCP や NFS を使用してコンパイルしたデバイスツリーを開発ボードにコピーし、/boot/dtb/RK3588-lubancat4.dtb を置き換えます。

uboot は起動時にこのディレクトリのデバイスファイルをメモリにロードし、カーネルに解析させます。開発ボードを再起動してください。

8.4.3.2 実験結果

デバイスツリーのデバイスノードはファイルシステム内に対応するファイルがあり、「/proc/device-tree」ディレクトリに位置しています。以下のように「/proc/device-tree」ディレクトリに移動します。

```
cat@lubancat:/proc/device-tree$ ls
'#address-cells'              dmac@fe550000                 i2s@fe430000
'#size-cells'                 dmc                           iep@fdef0000
 __symbols__                  dmc-fsp                       interrupt-controller@fd400000
 adc-keys                     dmc-opp-table                 interrupt-parent
 aliases                      dmcdbg                        iommu@fde4b000
 arm-pmu                      dsi@fe060000                  iommu@fdea0800
 audpwm@fe470000              dsi@fe070000                  iommu@fded0480
 backlight                    dwmmc@fe000000                iommu@fdee0800
 bus-npu                      dwmmc@fe2b0000                iommu@fdef0800
 bus-npu-opp-table            dwmmc@fe2c0000                iommu@fdf40f00
 cam-avdd                     ebc@fdec0000                  iommu@fdf80800
 cam-dovdd                    edp@fe0c0000                  iommu@fdfe0800
 cam-dvdd                     eink@fdf00000                 iommu@fdff1a00
 can@fe570000                 ethernet@fe010000             iommu@fe043e00
 can@fe580000                 ethernet@fe2a0000             jpegd@fded0000
 can@fe590000                 external-camera-clock         led_test
 chosen                       external-gmac0-clock          leds
 clock-controller@fdd00000    external-gmac1-clock          lpddr3-params
 clock-controller@fdd20000    fiq-debugger                  lpddr4-params
 codec-digital@fe478000       firmware                      lpddr4x-params
 compatible                   get_dts_info_test             mailbox@fe780000
 cpu0-opp-table               gpio-regulator                memory
 cpuinfo                      gpu@fde60000                  mini-pcie-3v3-regulator
 cpus                         hdmi@fe0a0000                 mipi-csi2@fdfb0000
```

次に led フォルダに入ると、led ノードで定義された属性とその子ノードを確認できます。

```
cat@lubancat:/proc/device-tree/get_dts_info_test$ ls
'#address-cells'  '#size-cells'   led@0xfdd60000   name   phandle
cat@lubancat:/proc/device-tree/get_dts_info_test$ cat name
get_dts_info_testcat@lubancat:/proc/device-tree/get_dts_info_test$
```

ノード属性には name が追加されていますが、これは led ノードで name 属性を定義していないにも関わらず生成されています。これは自動生成されたもので、ノード名を保存しています。

株式会社日昇テクノロジー

ここでの属性はファイルであり、子ノードはフォルダです。「led@0xfd5f8000」フォルダに再度入り、compatible、name、reg、status の 4 つの属性ファイルがあります。これらの属性ファイルを「cat」コマンドで確認できます。

```
cat@lubancat:/proc/device-tree/get_dts_info_test$ ls
'#address-cells'   '#size-cells'   led@0xfdd60000   name   phandle
cat@lubancat:/proc/device-tree/get_dts_info_test$
```

これにより、デバイスツリーに「get_dts_info_test」という名前のノードを成功裏に追加しました。

8.5 ドライバ内でノード属性を取得する実験

この実験の目的は、前のセクションで説明した of 関数を使用して、デバイスツリーからノード属性を取得する方法を示すことです。この実験を行う前に、デバイスツリーにデバイスノードを追加する実験を完了させる必要があります。なぜなら、この実験では、追加したノードからデバイスノード属性を取得するからです。ドライバはプラットフォームドライバに基づいています。

8.5.1 コード解説

この章のサンプルコードのディレクトリは「linux_driver/device_tree/get_dts_info.c」です。プログラムのソースコードは以下の通りですが、ここでは get_dts_info_probe 関数の内容のみをリストアップしています。完全な内容はこの章の補足ソースコードを参照してください。

リスト 33: ノード属性取得実験

```
1 /*get_dts_info_probe 関数*/
2 static int get_dts_info_probe(struct platform_device *pdev)
3 {
4     int error_status = -1;
5     pr_info("%s¥n", __func__);
6
7     led_test_device_node = of_find_node_by_path("/get_dts_info_test");
```

株式会社日昇テクノロジー

```
8    if(led_test_device_node == NULL)
9    {
10       printk(KERN_ALERT "¥n led_device_node  の取得に失敗しました！¥n");
11       return -1;
12   }
13   /* led_test_device_node  デバイスノード構造体からノードの基本情報を出力*/
14   printk(KERN_ALERT "name: %s", led_test_device_node->name); //ノード名を出力
15   printk(KERN_ALERT "child name: %s", led_test_device_node->child->name); //子ノードのノー
ド名を出力
16
17   /* led_device_node  の子ノードを取得*/
18   led_device_node = of_get_next_child(led_test_device_node, NULL);
19   if(led_device_node == NULL)
20   {
21       printk(KERN_ALERT "¥n led_device_node  の取得に失敗しました！¥n");
22       return -1;
23   }
24   printk(KERN_ALERT "name: %s", led_device_node->name); //ノード名を出力
25   printk(KERN_ALERT "parent name: %s", led_device_node->parent->name); //親ノードのノー
ド名を出力
26
27   /* led_device_node  ノードの「compatible」属性を取得*/
28   led_property = of_find_property(led_device_node, "compatible", &size);
```

株式会社日昇テクノロジー

```
29    if(led_property == NULL)
30    {
31        printk(KERN_ALERT "¥n led_property の取得に失敗しました！¥n");
32        return -1;
33    }
34    printk(KERN_ALERT "size = : %d", size); //実際に読み取られた長さ
35    printk(KERN_ALERT "name: %s", led_property->name); //属性名を出力
36    printk(KERN_ALERT "length: %d", led_property->length); //属性の長さを出力
37    printk(KERN_ALERT "value: %s", (char*)led_property->value); //属性値を出力
38
39    /*「reg」アドレス属性を取得*/
40    error_status = of_property_read_u32_array(led_device_node, "reg", out_values, 2);
41    if(error_status != 0)
42    {
43        printk(KERN_ALERT "¥n out_values の取得に失敗しました！¥n");
44        return -1;
45    }
46    printk(KERN_ALERT "0x%08X", out_values[0]);
47    printk(KERN_ALERT "0x%08X", out_values[1]);
48
49    return 0;
50 }
```

- 7-12 行：「of_find_node_by_path」関数を使用して「get_dts_info_test」デバイスノードを探し

株式会社日昇テクノロジー

ます。パラメーターは「get_dts_info_test」のデバイスノードパスです。

- 14-15 行：成功すると device_node 型の構造体ポインターが得られ、この構造体から必要なデータを取得できます。完全な属性情報を取得するには他の of 関数が必要になることがあります。

- 19-24 行：led_device_node の子ノードを取得します。ここでは「of_get_next_child」関数を使用して「led」ノードの子ノードを取得します。もちろん、「led」ノードの「デバイスノード構造体」から直接最初の子ノードを読み取ることもできます。

- 30-35 行：「of_find_property」関数を使用して「led」ノードの「compatible」属性を取得します。

- 42-49 行：「of_property_read_u32_array」関数を使用して reg 属性を取得します。

ドライバモジュールフォルダに移動してドライバモジュールをコンパイルします：

```
make
```

このフォルダには get_dts_info.ko ドライバモジュールが生成されます。

8.5.2 実験結果

コンパイル成功後、ドライバーの.ko ファイルを開発ボードにコピーし、insmod でドライバーモジュールをインストールすると以下のように表示されます：

```
sudo insmod get_dts_info.ko
```

```
[   34.619212] get_dts_info_probe
[   34.619517] name: get_dts_info_test
[   34.619520] child name: led
[   34.619564] name: led
[   34.619572] parent name: get_dts_info_test
[   34.619579] size = : 14
[   34.619585] name: compatible
[   34.619591] length: 14
[   34.619597] value : fire,led_test
[   34.619603] 0xFDD60000
[   34.619610] 0x00000100
```

get_dts_info_test ノードにマッチング

子ノード led の属性

株式会社日昇テクノロジー

上記から、ドライバープログラムがデバイスツリーで設定された属性値を正しく取得したことがわかります。

第 9 章 Linux デバイスツリー - LED ライト実験

前節の学習を通じて、簡単なデバイスツリーノードを作成し、デバイスツリーから必要なノードリソースを取得するための一般的な of 関数の使用方法を学びました。このセクションでは、デバイスツリーを使用してシンプルな LED ライトドライバープログラムを作成し、デバイスツリーへの理解を深めます。

9.1 実験説明

この実験では、Lubancat_RK 開発ボード上の LED ライト（システムライトを使用、またはピンを変更して GPIO を使用して LED に接続）を使用します。lubancat4 を例として、「文字デバイスドライバー実験 - LED ライトを点灯」セクションを参照できます。

9.2 実験コード解説

この章のサンプルコードディレクトリは：linux_driver/device_tree_led

9.2.1 プログラミングアプローチ

主なプログラミング内容は、LED デバイスノードをデバイスツリーに追加し、ドライバープログラムで of 関数を使用してデバイスノードから属性を取得し、テストアプリケーションを作成することです。

- まず、デバイスツリーに LED デバイスノードを追加します。

- 次に、プラットフォームデバイスドライバーフレームワークを作成します。主にドライバーエントリ関数、ドライバー登録解除関数、プラットフォームデバイス構造体定義の 3 つの部分を含む

株式会社日昇テクノロジー

- .probe 関数を実装して、LED のデバイス登録と初期化を行います。

- 文字デバイス操作関数セットを実装します。ここでは、.write 操作のみを実装します。

- 異なる値を入力して LED のオン/オフを制御するテストアプリケーションを作成します。

9.2.2 コード分析

9.2.2.1 RGB デバイスノードの追加

RGB ライトは実際には 1 つの GPIO ポートを使用しており、制御にはいくつかの制御レジスタが必要です。したがって、そのデバイスツリーノードも非常にシンプルです。

例として、 GPIO1_C7 を使用し、デバイスツリーの記述は以下の通りです：

リスト 1: デバイスノードの追加

```
1 /* led_test ノードを追加, */
2 led_test{
3    #address-cells = <1>;
4     #size-cells = <1>;
5          compatible = "fire,led_test";
6
7          //lubancat4 のシステム LED GPIO1_A7 の例
8          led@0xfd5f8004{ //データレジスタ（上位 16 ビット）のアドレス
9                    reg = <0xfd5f8004 0x00000004 0xfd5f800C 0x00000004>; // データレジスタとデータ
   方向レジスタ（上位 16 ビット）のアドレスと範囲
10                   status = "okay";
11    };
12
```

株式会社日昇テクノロジー

GPIO1_B5 を使用し、デバイスツリーの記述は以下の通りです：

リスト 2：デバイスノードの追加

```
1 /* led_test ノードを追加,*/
2 led_test{
3 #address-cells = <1>;
4 #size-cells = <1>;
5 compatible = "fire,led_test";
6
7 //lubancat4 のシステム LED GPIO4_B5
8 led@0xfec50000{ //データレジスタ（下位 16 ビット）のアドレスで、GPIO4 のベースアドレスで
もあります
9 reg = <0xfec50000 0x00000004 0xfec50008 0x00000004>; //データレジスタとデータ方向レジスタ
（下位 16 ビット）のアドレスと範囲
10 status = "okay";
11 };
12 };
```

LED ライトのデバイスノードはルートノードの下に追加されました。追加例については、本章のサポートコード linux_driver/device_tree_led/rk3568-lubancat4.dts を参照してください。

上記で追加されたデバイスツリーでは、LED ライトを制御するために使用されるレジスタが含まれています。疑問がある場合は、文字デバイスドライバ実験ー LED ライトを点灯させるセクションを参照してください。

株式会社日昇テクノロジー

• 2 行目：これが LED ライトのデバイスツリーノードで、ノード名「led_test」はルートノードの下にあるため、そのデバイスツリーパスは「/led_test」です。この「cells」は子ノードの reg 属性のスタイルを定義しています。「compatible」属性はドライバとのマッチングに使用され、ドライバには「compatible」と同じパラメータを設定し、これによってドライバがロードされると自動的にこのデバイスツリーノードとマッチングされます。

• 8-11 行目：led 子ノード、上記のように、レジスタを使用します。管理を容易にするため、「status = "okay"」は子ノードの状態を定義し、この子ノードを使用するために「okay」に設定されています。

9.2.2.2 ドライバープログラムの作成

デバイスツリーを基にしたドライバープログラムは、プラットフォームバスドライバーと非常に似ていますが、プラットフォームバスドライバーがプラットフォームデバイスとマッチングするのに対し、デバイスツリーベースのドライバーは対応するデバイスツリーノードとマッチングするだけです。

ドライバープログラムの主な内容には、プラットフォームデバイスドライバーフレームワークの作成、.probe 関数の実装、文字デバイス操作関数セットの実装、ドライバー登録解除が含まれます。ソースコードは linux_driver/device_tree_led/led_test.c にあります。

ドライバーエントリ関数

ドライバーエントリ関数は、単にプラットフォームドライバーを登録するだけです。以下に示します：

株式会社日昇テクノロジー

リスト 3: ドライバ初期化関数

```
1 /*
2 * ドライバ初期化関数
3 */
4 static int __init led_platform_driver_init(void)
5 {
6     int DriverState;
7     DriverState = platform_driver_register(&led_platform_driver);
8     printk(KERN_EMERG "¥tDriverState is %d¥n", DriverState);
9     return 0;
10 }
```

このエントリ関数全体で唯一、「platform_driver_register」関数を呼び出してプラットフォームドライバを登録しています。パラメータにはプラットフォームデバイス構造体を渡します。

プラットフォームデバイス構造体の定義

プラットフォームドライバを登録する際に使用されるプラットフォームデバイス構造体は、主にプラットフォームドライバの.probe 関数を指定し、プラットフォームドライバとマッチングするプラットフォームデバイス（デバイスツリーを使用した後は、プラットフォームドライバとマッチングするデバイスツリーノード）を指定する役割を持ちます。

株式会社日昇テクノロジー

リスト 4: プラットフォームデバイス構造体

```
1 static const struct of_device_id led_ids[] = {
2    {.compatible = "fire,led_test"},
3    {/* センチネル */}
4 };
5
6 /* プラットフォームデバイス構造体の定義 */
7 struct platform_driver led_platform_driver = {
8    .probe = led_probe,
9    .driver = {
10       .name = "leds-platform",
11       .owner = THIS_MODULE,
12       .of_match_table = led_ids,
13    }
14 };
```

- 1-4 行：マッチングテーブルの定義

- 8 行：定義されたプラットフォームデバイス構造体です。ここで「.probe = led_probe,」は.probe 関数を指定しています。.probe 関数は特別で、プラットフォームドライバとデバイスツリーノードがマッチングすると自動的に実行されます。RGB LED の初期化や文字デバイスの登録などはこの関数内で実装されます（もちろん他の関数で実装することも可能です）。

- 9-13 行：「.driver = { ...}」はドライバの属性を定義しています。名前、所有者などを含みますが、特に重要なのは「.of_match_table」属性です。これはドライバのマッチングテーブルを指定します。「.compatible = "fire,led_test"」というマッチング値を定義しており、このドライバはデ

株式会社日昇テクノロジー

バイスツリー内の「compatible = "fire,led_test"」のノードとマッチングします。

.probe 関数の実装

前述の通り、ドライバとデバイスツリーノードがマッチング成功すると、自動的に.probe 関数が実行されます。したがって、.probe 関数ではいくつかの初期化作業を行います。この実験では RGB の初期化と文字デバイスの初期化を.probe 関数内で実装しています。.probe 関数は長いですが、大量のシンプルで繰り返しの初期化コードを含んでいるため、非常に理解しやすいです。

リスト 5: .probe 関数

```
1 /* LED リソース構造体の定義、取得したノード情報と変換後の仮想レジスタアドレスを保存 */
2 struct led_resource
3 {
4     struct device_node *device_node; // LED のデバイスツリーノード
5     void __iomem *va_DR;
6     void __iomem *va_DDR;
7 };
8
9 static int led_probe(struct platform_device *pdv)
10 {
11     int ret = -1; // エラーステータスコードを保存
12     unsigned int register_data = 0;
13
14     printk(KERN_EMERG "¥t match successed ¥n");
15
```

株式会社日昇テクノロジー

```
16    /* led_test のデバイスツリーノードを取得 */
17    led_test_device_node = of_find_node_by_path("/led_test");
18    if (led_test_device_node == NULL)
19    {
20        printk(KERN_ERR "¥t led_test の取得に失敗しました！¥n");
21        return -1;
22    }
23
24    /* LED ノードを取得 */
25    led_res.device_node = of_find_node_by_name(led_test_device_node,"led");
26    if (led_res.device_node == NULL)
27    {
28        printk(KERN_ERR "¥n led_device_node の取得に失敗しました！¥n");
29        return -1;
30    }
31
32    /* reg 属性を取得し仮想アドレスに変換 */
33    led_res.va_DR = of_iomap(led_res.device_node, 0);
34    if(led_res.va_DR == NULL){
35        printk("of_iomap に失敗しました¥n");
36        return -1;
37    }
```

株式会社日昇テクノロジー

```
38
39    led_res.va_DDR = of_iomap(led_res.device_node, 1);
40    if(led_res.va_DDR == NULL){
41        printk("of_iomap に失敗しました¥n");
42        return -1;
43    }
44
45    /* モードレジスタの設定：出力モード */
46    register_data = readl(led_res.va_DDR); //  GPIO1_C7
47    register_data |= ((unsigned int)0X1 << (7)); // 低 16 ビットで GPIO の出力モードを制御
48    register_data |= ((unsigned int)0X1 << (23)); // 7+16、高 16 ビットで低 16 ビットの書き込み
 を有効化
49    writel(register_data,led_res.va_DDR);
50
51    /* 設定レジスタの設定：デフォルトで高電圧出力 */
52    register_data = readl(led_res.va_DR);
53    register_data |= ((unsigned int)0x1 << (7)); // 低 16 ビットで GPIO の電圧レベルを制御
54    register_data |= ((unsigned int)0x1 << (23)); // 7+16、高 16 ビットで低 16 ビットの書き込み
 を有効化
55    writel(register_data, led_res.va_DR);
56
57    /* 文字デバイス登録部分 */
```

株式会社日昇テクノロジー

```
58    // 第一ステップ
59    // 動的にデバイス番号を割り当て、メジャー番号を取得し、サブデバイス番号を 0 に、
 DEV_CNT を 1 に設定
60    ret = alloc_chrdev_region(&led_devno, 0, DEV_CNT, DEV_NAME);
61    if (ret < 0)
62    {
63        printk("led_devno の割り当てに失敗しました¥n");
64        goto alloc_err;
65    }
66    // 第二ステップ
67    // 文字デバイス構造体 cdev とファイル操作構造体 file_operations を関連付け
68    led_chr_dev.owner = THIS_MODULE;
69    cdev_init(&led_chr_dev, &led_chr_dev_fops);
70    // 第三ステップ
71    // cdev_map ハッシュテーブルにデバイスを追加
72    ret = cdev_add(&led_chr_dev, led_devno, DEV_CNT);
73    if (ret < 0)
74    {
75        printk("cdev の追加に失敗しました¥n");
76        goto add_err;
77    }
78
```

株式会社日昇テクノロジー

```
79    // 第四ステップ
80    /* クラスの作成 */
81    class_led = class_create(THIS_MODULE, DEV_NAME);
82
83    /* デバイスの作成 */
84    device = device_create(class_led, NULL, led_devno, NULL, DEV_NAME);
85
86    return 0;
87
88 add_err:
89    // デバイス追加に失敗した場合、デバイス番号を登録解除
90    unregister_chrdev_region(led_devno, DEV_CNT);
91    printk("¥n エラーが発生しました！¥n");
92 alloc_err:
93
94    return -1;
95 }
```

- 2-7 行：LED リソース構造体をカスタマイズして、取得したデバイスノード情報と変換後の仮想レジスタアドレスを保存します。

- 23-27 行：of_find_node_by_path 関数を使用してデバイスツリーノード「/led_test」を取得し、取得成功すると「/led_test」ノードのデバイスノード構造体が返されます。

- 26-57 行：LED ライトの初期化を行います。

- 65-89 行：文字デバイスを登録します。このプロセスは以前説明した文字デバイスドライバと非

株式会社日昇テクノロジー

常に似ています。

リスト 6: 文字デバイス登録に使用される構造体

```
1 static dev_t led_devno; // 文字デバイスのデバイス番号定義
2 static struct cdev led_chr_dev; // 文字デバイス構造体 chr_dev 定義
3 struct class *class_led; // 作成されたクラスを保存
4 struct device *device; // 作成されたデバイスを保存
5
6 static struct file_operations led_chr_dev_fops ={
7     .owner = THIS_MODULE,
8     .open = led_chr_dev_open,
9     .write = led_chr_dev_write,
10 };
```

- 65-70 行：「alloc_chrdev_region」を使用してメジャーデバイス番号を動的に割り当て、led_devno 構造体に保存します。

- 74 行：「cdev_init」を使用して文字デバイスを初期化します。

• 第 77-82 行：「cdev_add」を使用してシステムにキャラクターデバイスを追加します。デバイスノードを自動的に作成する必要がある場合は、クラスとデバイスを作成する必要があります。

• 第 89 行：「class_create」関数を使用してクラスを作成します。

• 第 86 行：「device_create」を使用してデバイスを作成します。パラメーターの「DEV_NAME」はデバイスノード名を指定するために使用され、この名前はアプリケーションで使用されます。

ドライバーとデバイスツリーノードが一致すると、システムは自動的に.probe 関数を実行します。上記のコードから、.probe 関数は RGB ライトの初期化とキャラクターデバイスの作成を行っています。次に、キャラクターデバイスの操作関数セットで RGB ライトを制御するだけです。

株式会社日昇テクノロジー

キャラクターデバイス操作関数セットの実装

プログラムの設計を簡素化するために、ここではキャラクターデバイス操作関数セットの.write 関数のみを実装しました。.write 関数は受信したメッセージに基づいて LED ライトのオン/オフを制御します。コードの紹介は以下の通りです：

リスト 7: .write 関数の実装

```
1 /* キャラクターデバイス操作関数セット、open 関数*/
2 static int led_chr_dev_open(struct inode *inode, struct file *filp)
3 {
4 printk("¥n led_chr_dev_open ¥n");
5 return 0;
6 }
7
8 /* キャラクターデバイス操作関数セット、write 関数*/
9 static ssize_t led_chr_dev_write(struct file *filp, const char __user *buf,
size_t cnt, loff_t *offt)
10 {
11
12 unsigned int register_data = 0; // 読み取ったレジスタデータを一時保存
13 unsigned char write_data; // 受け取ったデータを保存
14
15 int error = copy_from_user(&write_data, buf, cnt);
16 if (error < 0)
```

株式会社日昇テクノロジー

```
17 {
18 return -1;
19 }
20 /* led ピンの出力レベルを設定*/
21 if (write_data)
22 {
23 register_data |= ((unsigned int)0x1 << (7)); // 下位 16 ビットを 1 に設定
24 register_data |= ((unsigned int)0x1 << (23)); // 7+16、下位 16 ビットの書き込みを有効にする
25 writel(register_data, led_res.va_DR); // lubancat4 の GPIO1_C7 ピンを高レベル出力に設定、赤色
 LED を消灯
26 }
27 else
28 {
29 register_data &= ~((unsigned int)0x1 << (7)); // 下位 16 ビットを 0 に設定
30 register_data |= ((unsigned int)0x1 << (23)); // 7+16、下位 16 ビットの書き込みを有効にする
31 writel(register_data, led_res.va_DR); // lubancat4 の GPIO1_C7 ピンを低レベル出力に設定、赤色
 LED を点灯
32 }
33
34 return 0;
35
36 }
```

株式会社日昇テクノロジー

```
37

38 /* キャラクターデバイス操作関数セット*/

39 static struct file_operations led_chr_dev_fops =

40 {

41 .owner = THIS_MODULE,

42 .open = led_chr_dev_open,

43 .write = led_chr_dev_write,

44 };
```

2 つのキャラクターデバイス操作関数のみを実装しました。open は led_chr_dev_open 関数に対応しており、これは空関数です。.write は led_chr_dev_write 関数に対応しており、この関数はアプリケーションから送られてきたコマンドを受け取り、そのコマンドに基づいて LED ライトのオン/オフを制御します。

- 第 15-19 行：copy_from_user 関数を使用してユーザースペースのデータをカーネルスペースにコピーします。ここで渡されるデータは無符号整数データです。
- 第 21-32 行：取得した値を解析し、led のオン/オフを制御します。ここでの led のピンは GPIO1_C7 です。

9.2.2.3 テストアプリケーションの作成

ドライバプログラムでは、デバイスノードファイルの自動作成方法を採用し、キャラクターデバイスのデバイスノードを作成しました。ファイル名は自由に定義できます。テストアプリケーションを書く際には、ファイル名を覚えておいてください。この例ではデバイスノード名を「rgb_led」としています。テストプログラムは非常にシンプルで、ソースコードは以下のとおりです：

株式会社日昇テクノロジー

リスト 8: テストアプリケーション

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <fcntl.h>
4 #include <string.h>
5 int main(int argc, char *argv[])
6 {
7 printf("led test_app¥n");
8 /* 入力されたコマンドが正しいか判断*/
9 if(argc != 2)
10 {
11 printf(" command error ! ¥n");
12 printf(" usage : sudo test_app num [num can be 0 or 1]¥n");
13 return -1;
14 }
15
16 /* ファイルを開く*/
17 int fd = open("/dev/led_test", O_RDWR);
18 if(fd < 0)
19 {
20 printf("open file : %s failed !¥n", argv[0]);
21 return -1;
```

株式会社日昇テクノロジー

```
22 }
23
24 unsigned char command = atoi(argv[1]); // 受け取ったコマンド値を数字に変換;
25
26 /* コマンドを書き込む*/
27 int error = write(fd, &command, sizeof(command));
28 if(error < 0)
29 {
30 printf("write file error! ¥n");
31 close(fd);
32 /* ファイルが正常に閉じられたか判断*/
33 }
34
35 /* ファイルを閉じる*/
36 error = close(fd);
37 if(error < 0)
38 {
39 printf("close file error! ¥n");
40 }
41
42 return 0;
43 }
```

株式会社日昇テクノロジー

• 第 7-14 行：入力が正しいか簡単に判断します。このテストアプリケーションを実行する際、argc

は 2 でなければなりません。それはアプリケーションファイル名とコマンドで構成されます、例えば「./test_app <コマンド値>」。

- 第 17-22 行：デバイスファイルを開きます。
- 第 27 行：端末から入力されたコマンド値を数字に変換し、最終的に write 関数を使用します
- 第 35-39 行：デバイスファイルを閉じます。

9.3 ドライバプログラムのコンパイル

9.3.1 デバイスツリーのコンパイル

led_test ノードをデバイスツリーに追加し、カーネルソースコードディレクトリで以下のコマンドを実行します。

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- lubancat4_defconfig //ここでは RK358x の設定ファイルを例としています
make ARCH=arm64 -j4 CROSS_COMPILE=aarch64-linux-gnu- dtbs
```

コンパイル成功後、生成されたデバイスツリーファイル(.dtb)はソースコードディレクトリ下の arch/arm64/boot/dts/rockchip/にあり、ファイル名は「RK3588-lubancat.dtb」です。

9.3.2 ドライバとアプリケーションのコンパイル

make コマンドを実行します。Makefile は前の章と大体同じです。最終的に led_test.ko と test_app アプリケーションが生成されます。

9.4 プログラムの実行結果

Permission denied や類似のメッセージが表示された場合は、ユーザ権限に注意してください。ほとんどのハードウェア外部機器の操作機能には、root ユーザー権限が必要です。簡単な解決策は、コマンド実行前に sudo を追加するか、root ユーザーでプログラムを実行します。

株式会社日昇テクノロジー

9.4.1 実験操作

このセクションの実験では、lubancat のボードのシステムデバイスツリーでは、LED のデバイス機能がデフォルトで有効になっています。デバイスツリーの leds ノードを無効にするには、leds ノードの status = "okay"; を status = "disabled";に変更し、デバイスツリーをコンパイルして置き換えます。または、以下のコマンドを使用してボードで直接システム leds ドライバによる LED の制御を無効にすることができます：

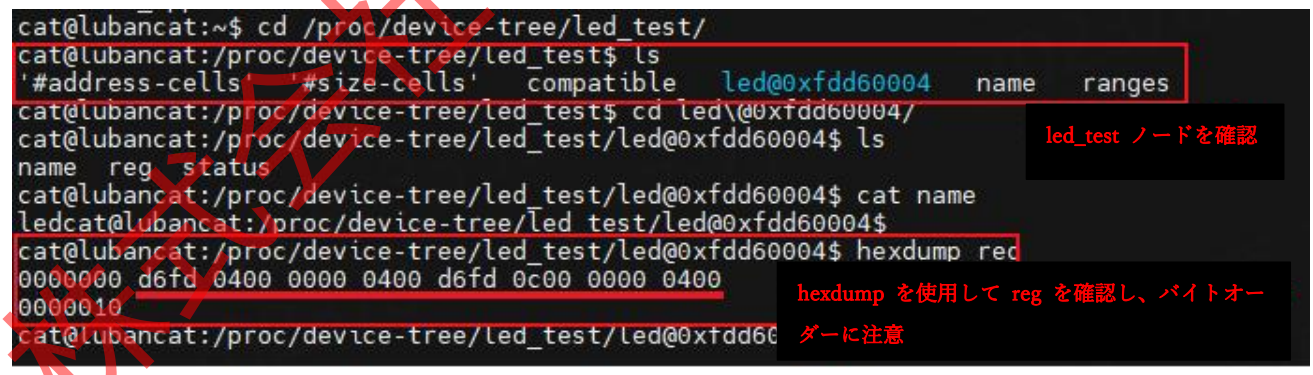
```
sudo sh -c 'echo 0 > /sys/class/leds/sys_status_led/brightness'
```

LED の明るさを 0 に設定すると、同時に LED のトリガー条件が自動的に none に変更され、leds ドライバによる LED の制御がキャンセルされます。

デバイスツリー、ドライバプログラム、アプリケーションを NFS や SCP などの方法で開発ボードにコピーします。

/boot/dtb/RK3588-lubancat.dtb の元のデバイスツリーを置き換えて開発ボードを再起動します。

再起動後、/proc/device-tree/ディレクトリ下で led_test を見つけることができます。制御されているピンは GPIO1_C7 です、以下のように表示されます：



以下のコマンドを実行してドライバをロードします：

```
sudo insmod led_test.ko
```

```
cat@lubancat:~$ sudo insmod led_test.ko

Message from syslogd@lubancat at Sep  5 16:47:44 ...
 kernel:[ 2123.668411]    match successed

Message from syslogd@lubancat at Sep  5 16:47:44 ...
 kernel:[ 2123.670906]  DriverState is 0
cat@lubancat:~$ ls /dev/led_test
/dev/led_test
cat@lubancat:~$
```

ドライバが正常にロードされたら、以下のようにアプリケーションを直接実行します。

コマンド：./test_app <コマンド>

コマンドは「unsigned char」型のデータで、1 を入力すると消灯、0 を入力すると点灯します。

実行結果は以下の通りです：

```
cat@lubancat:~$ sudo ./test_app 0
led test_app
cat@lubancat:~$ sudo ./test_app 1
led test_app
cat@lubancat:~$
```

LED ピンを低電圧で制御し、LED を点灯

LED ピンを高電圧で出力し、LED を消灯

同時に、ボードのハートビート LED を観察すると、LED が点灯または消灯しているのが見えます。

第 10 章 デバイスツリーオーバーレイ

Linux 4.4 以降で導入されたダイナミックデバイスツリーは、「デバイスツリーオーバーレイ」として翻訳されます。デバイスツリーオーバーレイは、メインデバイスツリーの「パッチ」として理解でき、システムに動的にロードされ、カーネルに認識されます。例えば、システムに RGB ドライバを追加したい場合は、そのハードウェアデバイス用のデバイスツリーオーバーレイを作成し、コンパイルしてシステムにロードするだけで、メインデバイスツリーを再コンパイルする必要はありません。

デバイスツリーオーバーレイは、デバイスツリーに追加される内容であり、以前に説明したデバイスツリーの文法はそのまま適用されます。実際には、以前に作成したデバイスツリーノードをデバイスツリーオーバーレイに直接コピーすることもできます。具体的な使用方法は以下の通りです。

株式会社日昇テクノロジー

10.1 デバイスツリーオーバーレイの形式

デバイスツリーオーバーレイには比較的固定された形式があり、デバイスノードに「シェル」を加えてコンパイルした後、カーネルがそれを動的にロードできるようになっています。具体的な形式は以下の通りで、具体的なノードは省略されています。

リスト 1: デバイスツリーオーバーレイの基本形式 1

```
1 /dts-v1/;
2 /plugin/;
3
4 / {
5   fragment@0 {
6     target-path = "/";
7     __overlay__ {
8       /* ここに挿入するノードを追加 */
9       .....
10     };
11   };
12
13   fragment@1 {
14     target = <&XXXXX>;
15     __overlay__ {
16       /* ここに挿入するノードを追加 */
17       .....
```

株式会社日昇テクノロジー

```
18    };
19  };
20  .....
21 };
```

・第 1 行：dts のバージョンを指定します。

・第 2 行：未定義の参照を許可し、それらを記録します。デバイスツリーオーバーレイでは、メインデバイスツリーのノードを参照できますが、これら「参照されたノード」はデバイスツリーオーバーレイにとって未定義ですので、「/plugin/」を加える必要があります。

・第 6 行：デバイスツリーオーバーレイのロード位置を指定します。デフォルトではルートノードにロードされますが、「target-path =“/”」、または「target = <&XXXXX>」を使用して、特定のノードにノードや属性を追加することができます。

・第 7-8 行：挿入するデバイスやノード、または参照（追加）するデバイスツリーノードを __overlay__ {…}内に置きます。メインデバイスツリーのノードを追加、変更、または上書きできます。

別のデバイスツリーオーバーレイ形式：

リスト 2: デバイスツリーオーバーレイの基本形式 2

```
1 /dts-v1/;
2 /plugin/;
3
4 &{/} {
5   /* ここではルートノード"/"に、挿入するノードや属性を追加 */
6 };
7
```

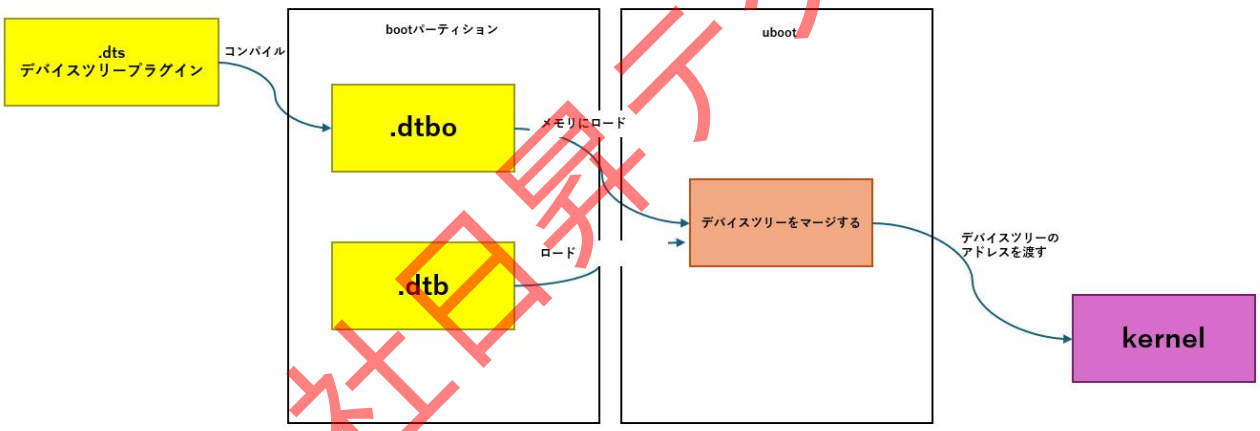
株式会社日昇テクノロジー

```
8 &XXXXX {
9   /* ここでは"XXXXX"ノードに、挿入するノードや属性を追加 */
10 };
```

これら 2 つの形式のデバイスツリーオーバーレイはどちらも使用できます。この章の実験では 2 番目の形式を例としています。コンパイルはカーネルソースコード内で行われます。単独でコンパイルする場合は、いくつかのライブラリの使用と DTC のバージョンに注意が必要です。

10.2 デバイスツリーオーバーレイのロード

例として lubancat4 (ubuntu20.04 イメージ)を使用します。デバイスツリーオーバーレイのロードは uboot を通じて行われ、プロセスは以下の通りです：



・デバイスツリーオーバーレイのソースファイルを作成し、DTC ツールを使用して.dtbo ファイルをコンパイルし、boot パーティションに保存します。

・boot パーティションのデバイスツリーオーバーレイをメモリにロードします。

・uboot 内で、デバイスツリーオーバーレイ dtbo ファイルとデバイスツリー dtb ファイルを 1 つのデバイスツリーにマージし、指定されたメモリアドレスに配置します。

・カーネルを起動し、デバイスツリーのメモリ内アドレスを渡します。

株式会社日昇テクノロジー

10.3 デバイスツリーオーバーレイ実験 1

10.3.1 ハードウェア紹介

このセクションの実験では、Lubancat_RK のボードを使用します。

10.3.2 デバイスツリーオーバーレイの作成とロード

この章のサンプルコードのディレクトリは、linux_driver/dynamic_device_tree です。

衝突を避けるために、前章でメインデバイスツリーに追加した led_test ノードを削除し、デバイスツリーオーバーレイの形式で変更する必要があります。カーネルソースコードの /arch/arm64/boot/dts/rockchip/overlays ディレクトリに lubancat-led-overlay.dts という名前のファイルを追加し、内容は以下の通りです：

lubancat4 を例に：

リスト 4: led デバイスツリーオーバーレイ

```
1 /dts-v1/;
2 /plugin/;
3
4 / {
5   fragment@0 {
6     target-path = "/";
7
8     __overlay__ {
9       /* led_test ノードを追加, */
10      led_test{
```

株式会社日昇テクノロジー

```
11        #address-cells = <2>;
12        #size-cells = <1>;
13        compatible = "fire,led_test";
14        ranges;
15
16        led@0xfec50000{
17          reg = <0xfec50000 0x00000004 0xfec50008 0x00000004>;
18          status = "okay";
19        };
20      };
21    };
22  };
23 };
```

この内容は前章と大きく変わらないが、デバイスツリーオーバーレイの書式に沿って修正されています。

・第 6 行：デバイスツリーオーバーレイのロード位置を指定し、ルートノードにロードします。

・第 8-21 行：挿入するデバイスおよびノード、または追加するデバイスツリーノードを__overlay__{…}内に配置し、前章のメインデバイスツリーの test_led ノードをここに追加します。

カーネルのディレクトリ/arch/arm64/boot/dts/rockchip/overlays 内の Makefile を編集し、編集したデバイスツリーオーバーレイを追加します。そして、Makefile と同じディレクトリにデバイスツリーオーバーレイファイルを配置して、デバイスツリーオーバーレイをコンパイルします。

株式会社日昇テクノロジー

次に、カーネルソースコードのトップディレクトリで以下のコマンドを実行し、デバイスツリーオーバーレイをコンパイルします：

rk3588 ボードの場合は以下のコマンドを実行します：

```
# 設定ファイルをロード
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- lubancat_linux_rk3588_defconfig
# dtbs オプションでデバイスツリーを個別にコンパイル
make ARCH=arm64 -j4 CROSS_COMPILE=aarch64-linux-gnu- dtbs
```

コンパイルされたデバイスツリーオーバーレイは、カーネルソースコードの /arch/arm64/boot/dts/rockchip/overlay/内にある lubancat-led-overlay.dtbo になります。このデバイスツリーオーバーレイを開発ボードの/boot/dtb/overlay/ディレクトリに転送し、/boot/uEnv/uEnv.txt にフォーマットに従ってデバイスツリーオーバーレイを追加してから開発ボードを再起動します。そうすると、システムはコンパイルしたデバイスツリーオーバーレイをロードします。

株式会社日昇テクノロジー

```
root@lubancat:/home/cat# cat /boot/uEnv/uEnv.txt
uname_r=5.10.160
size=0x1000000
cmdline="earlyprintk console=ttyFIQ0 console=tty1 consoleblank=0 loglevel=7
wait rw rootfstype=ext4"

enable_uboot_overlays=1
#overlay_start

#dtoverlay=/dtb/overlay/rk3588-lubancat-i2c2-m4-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-i2c3-m1-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-i2c5-m3-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-i2c6-m3-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-i2c8-m2-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-pwm3-ir-m3-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-pwm10-m2-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-pwm11-ir-m3-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-pwm14-m2-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-pwm15-ir-m3-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-spi0-m2-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-uart0-m2-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-uart4-m2-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-uart6-m1-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-uart7-m1-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-uart7-m2-overlay.dtbo
#dtoverlay=/dtb/overlay/rk3588-lubancat-uart0-m2-overlay.dtbo
dtoverlay=/dtb/overlay/lubancat-led-overlay.dtbo
```

10.3.3 ドライバコード

ドライバ部分は前章と全く同じで、ここでは詳細な説明を省略します。違いは前章ではデバイスツリーを使用し、この章ではデバイスツリーオーバーレイを使用している点ですが、原理は同じです。

10.3.4 LED のテスト

このセクションの実験では、lubancat のボードのシステムデバイスツリーでは、LED のデバイス機能がデフォルトで有効になっています。デバイスツリーの leds ノードを無効にするには、leds ノードの status = "okay";を status = "disabled";に変更し、デバイスツリーを再コンパイルして置き換えます。または、以下のコマンドを使用してボードで直接システム leds ドライバによる LED の制御を無効にすることができます：

```
sudo sh -c 'echo 0 > /sys/class/leds/sys_status_led/brightness'
```

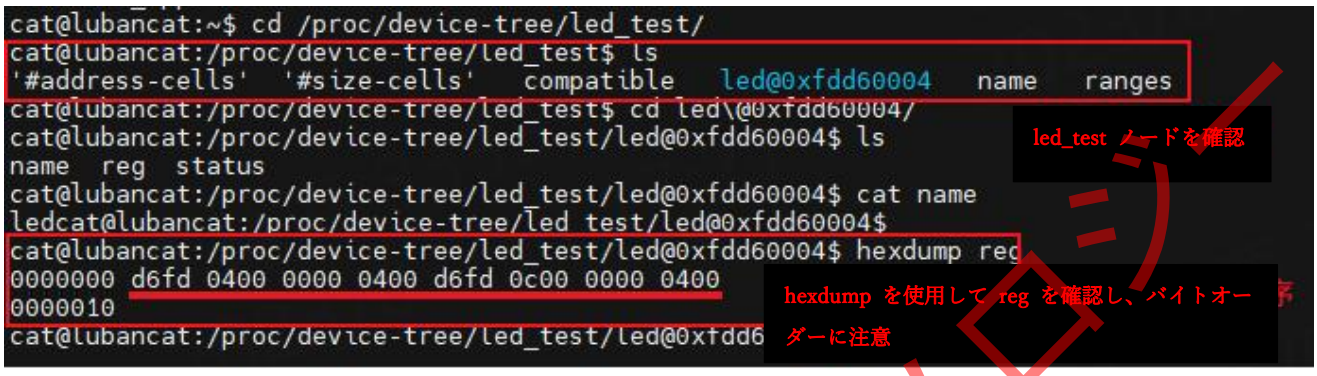
LED の明るさを 0 に設定すると、同時に LED のトリガー条件が自動的に none に変更され、leds ドライバによる LED の制御がキャンセルされます。

デバイスツリー、ドライバプログラム、アプリケーションを NFS や SCP などの方法で開発ボードにコ

株式会社日昇テクノロジー

ピーします。

再起動後、/proc/device-tree/ディレクトリ下で led_test を見つけることができます。制御されているピンは GPIO1_C7 です、以下のように表示されます：



以下のコマンドを実行してドライバをロードします：

```
sudo insmod led_test.ko
```

ドライバが正常にロードされたら、以下のようにアプリケーションを直接実行します。

コマンド：./test_app <コマンド>

コマンドは「unsigned char」型のデータで、1 を入力すると消灯、0 を入力すると点灯します。

実行結果は以下の通りです：

同時に、ボードのハートビート LED を観察すると、LED が点灯または消灯しているのが見えます。

第 11 章 Pinctrl 子系統と GPIO 子系統

前章で、レジスタを用いてキャラクタデバイスを書く経験をしました。このようにドライバコード内で直接レジスタマッピングを通じてペリフェラルを使用するプログラミング方法は、ドライバ開発者の観点から見れば災難と言えます。なぜなら、チップのレジスタが変更された場合、底層のドライバをほぼ書き直さなければならないからです。

この問題に対して、一歩進んで、デバイスツリーを用いてペリフェラルの様々な情報（例えばレジスタア

株式会社日昇テクノロジー

ドレス）を記述する方法を学びました。これにより、デバイス情報が変更されても、デバイスツリーのインターフェース関数を通じて、柔軟にデバイスの情報を取得できるようになりました。

では、ドライバ内で具体的なレジスタ操作に触れることなく、より一般的な方法はないのでしょうか？それには、ドライバフレームワークを使用することができます。カーネルは各種類のドライバに対して、成熟した、標準的な、典型的なドライバ実装を設計し、異なるメーカーの同類のハードウェアドライバの共通部分を抽出して自身で実装し、異なる部分を具体的なドライバ開発者が実装するためのインターフェースを提供します。標準化されたドライバ実装により、システムリソースを統一管理し、システムの安定性を維持します。

本章では、カーネルの pinctrl 子システムと GPIO 子システムの基本概念、主要なデータ構造、および rockchip の pinctrl コントローラーなどを簡単に紹介します。

11.1 Pinctrl 子システム

前章で、多くの SoC 内部には複数のピンコントローラーが含まれていることを学びました。ピンコントローラーのレジスタを通じて、一つまたは一群のピンの機能と特性を設定できます。Linux カーネルは、各 SoC メーカーのピン管理を統一するために、pinctrl 子システムを提供しています。このシステムの役割は以下の通りです：

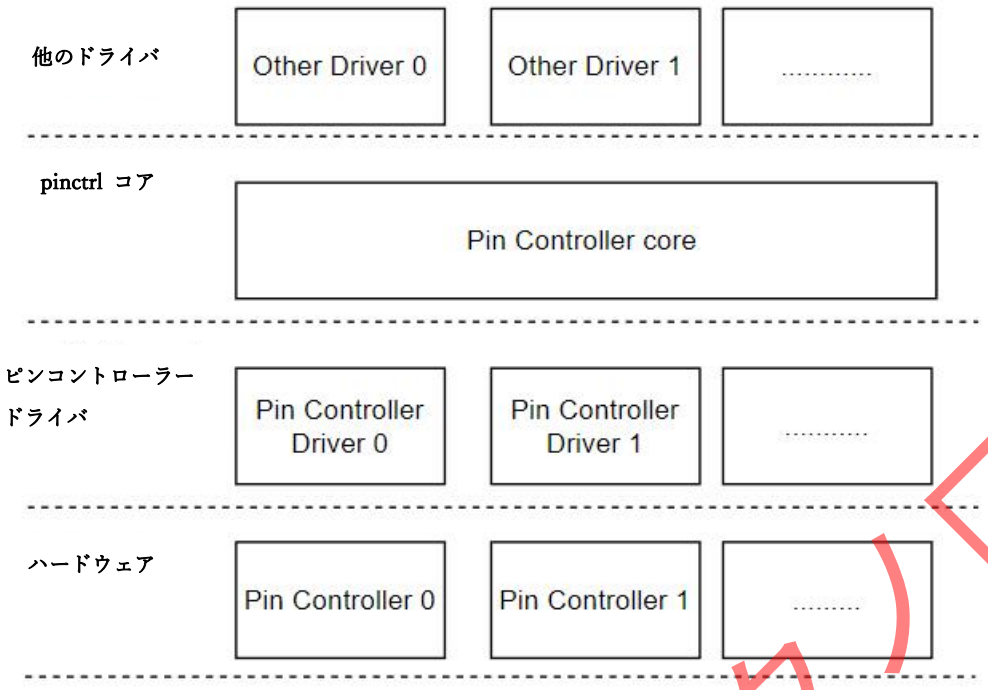
(1) 制御可能な全ピンを列挙し、システムの初期化時に全ての制御可能なピンを列挙し、これらのピンを識別します。

(2) ピンの機能多重化を設定します。例えば、GPIO や SPI などとして多重化します。

(3) ピンの設定、例えばプルアップ/プルダウン、ドライブ強度、デバウンスなどです。

pinctrl 子システムの構造は以下の通りです：

株式会社日昇テクノロジー



上図のように、pinctrl コア層はカーネルが抽象化したもので、下方には各 SoC ピンコントローラードライバに対して底層通信インターフェースの能力を提供し、上方には他のドライバにピン制御能力（例えばピン多重化、ピンの電気特性の設定）を提供し、同時に GPIO 子システムにピン操作も提供します。ピンコントローラードライバ層は、ピン操作メソッドを主に提供します。

pinctrl 子システムのソースファイルは、カーネルソースコードの/drivers/pinctrl ディレクトリにあり、主にコアファイル、他のカーネルドライバのインターフェースヘッダファイル、底層のピンコントローラードライバインターフェース、他のメーカーの Pinctrl ドライバファイルを含みます。

11.1.1 重要な概念

システムには具体的にどれだけのピンがあり、これらのピンはどのように記述されているのでしょうか？pinctrl コア層は、struct pinctrl_desc を提供し、具体的な SoC メーカーはこの抽象をもとにピンコントローラインスタンスを出力し、システム内の全ピンを記述し、インデックスを作成します。実際には、struct pinctrl_desc 構造内の pins と npins によってこれを完成します。それに応じて、ピンコントローラード

株式会社日昇テクノロジー

ライバがあります。

Pinctrl ドライバは、ピンの管理をピングループと関数の抽象化を通じて実現します。具体的には、RK3588-lubancat4.dts デバイスツリーを例にして説明します：

リスト 1: rk3588-lubancat4.dts

```
1 &pinctrl {
2 pmic {
3 pmic_int: pmic_int {
4 rockchip,pins =
5 <0 RK_PA3 RK_FUNC_GPIO &pcfg_pull_up>;
6 };
7
8 soc_slppin_gpio: soc_slppin_gpio {
9 rockchip,pins =
10 <0 RK_PA2 RK_FUNC_GPIO &pcfg_output_low_pull_down>;
11 };
12
13 soc_slppin_slp: soc_slppin_slp {
14 rockchip,pins =
15 <0 RK_PA2 RK_FUNC_1 &pcfg_pull_up>;
16 };
17
18 soc_slppin_rst: soc_slppin_rst {
```

株式会社日昇テクノロジー

```
19 rockchip,pins =
20 <0 RK_PA2 RK_FUNC_2 &pcfg_pull_none>;
21 };
22
23 spk_ctl_gpio: spk_ctl_gpio {
24 rockchip,pins = <3 RK_PC5 RK_FUNC_GPIO &pcfg_pull_up>;
25 };
26 };
27
28 /*.....*/
29
30 spi { //function
31 spi3_cs0: spi3-cs0 { //groups
32 rockchip,pins = <4 RK_PC6 RK_FUNC_GPIO &pcfg_pull_up_drv_level_1>;
33 };
34
35 spi3_cs1: spi3-cs1 {
36 rockchip,pins = <4 RK_PC4 RK_FUNC_GPIO &pcfg_pull_up_drv_level_1>;
37 };
38 };
39
40 /*.....*/
41 }
```

株式会社日昇テクノロジー

Pin groups

グループとは、上述の dts 内の spi3_cs0: spi3-cs0 のように、一連のピンを指し、これらのピンは統一された機能を表しています。たとえば、spi には 2 つの cs が必要であり、pmic には 5 つのピンが必要です。ピンを定義する際には、各ピンの電気特性の設定（例：プルアップ/プルダウン抵抗、駆動能力など）も提供されます。カーネル内では struct group_desc 構造体で記述されます。

function

function とは、ピンの機能群を指し、上述の dts 内の spi のように、現在そのピンが代表する機能を表します。あるグループが参照するピンの function が有効である場合のみ、それ以外はピンの function が衝突する原因となります。例えば、あるピンが通常の GPIO としても、SPI の cs ピンとしても機能する場合、そのピンは一つの function を代表することのみが可能です。つまり、通常の GPIO として機能するか、SPI の cs ピンとして機能するかのどちらかです。

pin state

デバイスがある状態で動作する際に使用するピンと function は一意に確定されます。固定された組み合わせが固定された状態を確認し、この固定状態は pinctrl 子システム内で「pin state」と呼ばれ、デバイスの可能性を列挙します。他のデバイスドライバが使用する際には、pinctrl 子システムが対応する pin state を設定するだけでよいです。デバイスツリー内での記述は、pinctrl-names で状態名を指定し、pinctrl-x でピン状態を指定します。

11.1.2 主要なデータ構造とインターフェース

オブジェクト指向の考え方から見ると、カーネルは pinctrl ドライバを pinctrl_desc オブジェクトとして抽象化し、具体的な soc メーカーの pinctrl ドライバはこのオブジェクトのインスタンスとなります。すべてのピン情報およびピンの制御インターフェースを pinctrl_desc にインスタンス化し、pinctrl_desc をカーネルに登録します。以下は pinctrl_desc の構造です：

株式会社日昇テクノロジー

リスト 2: pinctrl_desc

```
1 struct pinctrl_desc {
2   const char *name;
3   const struct pinctrl_pin_desc *pins; // ピンコントローラのピンを記述,
4   unsigned int npins; // そのコントローラが持つピンの数を記述
5   const struct pinctrl_ops *pctlops; // ピン操作関数、ピンの記述、ピンの取得など、全体の制御関数
6   const struct pinmux_ops *pmxops; // ピンの多重化に関する操作関数
7   const struct pinconf_ops *confops; // ピン設定に関する操作
8   struct module *owner;
9   #ifdef CONFIG_GENERIC_PINCONF
10  unsigned int num_custom_params;
11  const struct pinconf_generic_params *custom_params;
12  const struct pin_config_item *custom_conf_items;
13  #endif
14 };
```

一般的にコントローラドライバがデバイスとマッチし、probe を呼び出した後、最終的には pinctrl_register 関数を呼び出してカーネルに pinctrl を登録し、pinctrl_dev を生成します。この関数は以下のようになります：

リスト 3: pinctrl_register

```
1 struct pinctrl_dev *pinctrl_register(struct pinctrl_desc *pctldesc,
2                                      struct device *dev, void *driver_data);
```

ピンを記述する構造体 struct pinctrl_pin_desc：

株式会社日昇テクノロジー

リスト 4: pinctrl_pin_desc

```
1 struct pinctrl_pin_desc {
2   unsigned number;
3   const char *name;
4   void *drv_data;
5 };
```

多くのピンが一緒になって特定の機能を実現するために、struct group_desc を使用します：

リスト 5: group_desc

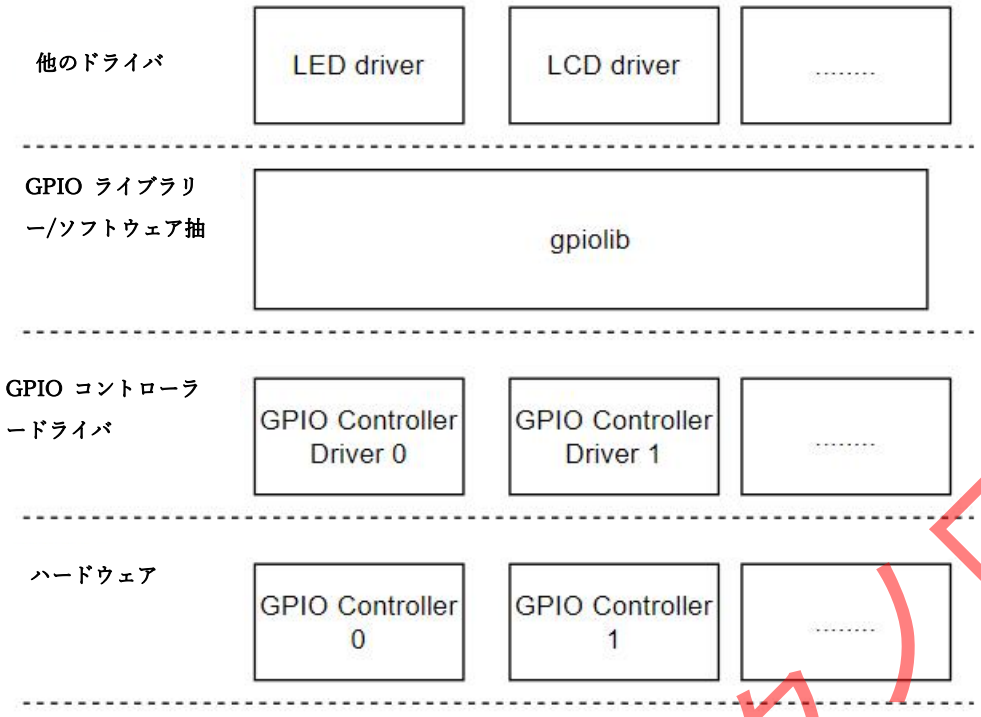
```
1 struct group_desc {
2   const char *name;
3   int *pins;
4   int num_pins;
5   void *data;
6 };
```

11.2 GPIO 子システム

pinctrl 子システムでピンを通常の GPIO として初期化した後、GPIO 子システムのインターフェースを使用して IO ポートの電圧レベル、割り込みなどを操作できます。ドライバ開発者はデバイスツリーに gpio に関する情報を追加し、その後、ドライバプログラム内で GPIO 子システムが提供する API 関数を使用して GPIO を操作できます。これにより、ドライバ開発者が GPIO を使用することが大幅に容易になります。gpio 子システムのコードはカーネルソースコードの/drivers/gpio/ディレクトリ下にあります。

GPIO 子システムの構造は以下の通り簡単に記述されます：

株式会社日昇テクノロジー



GPIO の核心は gpiolib フレームワークであり、上層に他のドライバに対していくつかの gpio インターフェースを提供し、下層に gpio リソースの登録関数を提供します。上層の他のドライバ、例えば LED ドライバは、gpiolib に対して gpio を申請し、設定および使用することができます。下層のコントローラドライバ（通常は SOC メーカーが作成）は、起動時に gpio リソースを gpiolib に登録します。例えば、ピン数、操作関数などです。

11.2.1 重要な構造体

GPIO 子システムにおける主要なデータ構造には gpio_device、gpio_chip、gpio_desc などがあり、gpio コントローラを記述するためにあります。これにはピン情報、割り込み情報、関連する操作関数などが含まれます。

株式会社日昇テクノロジー

リスト 6: gpio_device（カーネルソースコード/drivers/gpio/gpiolib.h 内）

```
1 struct gpio_device {
2   int id; //GPIO コントローラの ID、つまり第何番目か
3   struct device dev;
4   struct cdev chrdev;
5   struct device *mockdev;
6   struct module *owner;
7   struct gpio_chip *chip;
8   struct gpio_desc *descs;
9   int base; //カーネル内の GPIO の番号、GPIO を申請する際にこの番号を基に検索
10  u16 ngpio; //その GPIO コントローラが持つピンの数
11  const char *label; //ラベル
12  void *data;
13  struct list_head list;
14
15  #ifdef CONFIG_PINCTRL
16  /*
17   * CONFIG_PINCTRL が有効な場合、GPIO コントローラはオプションで
18   * SoC 内で提供される実際のピンレンジを記述できます。この情報は
19   * pinctrl サブシステムによって GPIO 使用のために対応するピンを設定するために使われます。
20   */
21  struct list_head pin_ranges;
22  #endif
23 };
```

株式会社日昇テクノロジー

リスト 7: gpio_chip（カーネルソースコード/include/linux/gpio/driver.h 内）

```
1 struct gpio_chip {
2   const char *label; //GPIO ポートの名前、ラベル
3   struct device *dev;
4   struct module *owner;
5
6   /* GPIO ポートの操作メソッド */
7   int (*request)(struct gpio_chip *chip,
8               unsigned offset);
9   void (*free)(struct gpio_chip *chip,
10              unsigned offset);
11  int (*direction_input)(struct gpio_chip *chip,
12                  unsigned offset);
13  int (*get)(struct gpio_chip *chip,
14          unsigned offset);
15  int (*direction_output)(struct gpio_chip *chip,
16                  unsigned offset, int value);
17  int (*set_debounce)(struct gpio_chip *chip,
18                  unsigned offset, unsigned debounce);
19  void (*set)(struct gpio_chip *chip,
20              unsigned offset, int value);
21  int (*to_irq)(struct gpio_chip *chip,
```

株式会社日昇テクノロジー

```
22              unsigned offset);
23  /*.....*/
24  int base; //カーネル内の GPIO の番号、申請する GPIO はこの番号に基づいて検索
25  u16 ngpio; //このコントローラの GPIO 数
26  const char *const *names;
27  unsigned can_sleep;
28  /*.....*/
29 };
```

リスト 8: gpio_desc（カーネルソースコード/drivers/gpio/gpiolib.h 内）

```
1 struct gpio_desc {
2  struct gpio_device *gdev; //gpio_device は GPIO 情報を記述、詳細はカーネルソースコード
/drivers/gpio/gpiolib.h を参照
3  unsigned long flags;
4  /* フラグシンボルはビット番号 */
5  #define FLAG_REQUESTED 0
6  #define FLAG_IS_OUT 1
7  #define FLAG_EXPORT 2 /* sysfs_lock によって保護 */
8  #define FLAG_SYSFS 3 /* /sys/class/gpio/経由でエクスポート */
9  #define FLAG_ACTIVE_LOW 6 /* 値がアクティブロー */
10 #define FLAG_OPEN_DRAIN 7 /* GPIO はオープンドレインタイプ */
11 #define FLAG_OPEN_SOURCE 8 /* GPIO はオープンソースタイプ */
12 #define FLAG_USED_AS_IRQ 9 /* GPIO は IRQ に接続 */
```

株式会社日昇テクノロジー

```
13  #define FLAG_IS_HOGGED 11 /* GPIO  は確保されている */
14  #define FLAG_TRANSITORY 12 /* GPIO  はスリープやリセットで値が失われる可能性がある */
15
16  /* 接続ラベル */
17  const char *label;
18  /* GPIO  の名前 */
19  const char *name;
20 };
```

gpio_device は一つの GPIO コントローラを表し、GPIO Controller 内の各ピンは gpio_desc で表され、ピンの関連操作関数と割り込み関連は gpio_chip 内にあります。

11.2.2 主要 API 関数解説

GPIO サブシステムには 2 つのインターフェースセットがあります。一つはディスクリプタベースのもので、関連する API 関数はすべて gpiod_で始まります。devm_が前に付く場合はデバイスリソース管理を意味し、自動的にリソースを解放するメカニズムです。もう一つは古いもので、gpio_で始まります。

1. GPIO 番号取得関数 of_get_named_gpio

ほとんどの GPIO サブシステム API 関数は GPIO 番号を使用します。GPIO 番号は、of_get_named_gpio 関数を使用してデバイスツリーから取得できます。

リスト 9: of_get_named_gpio 関数（カーネルソースコード include/linux/of_gpio.h 内）

```
1 static inline int of_get_named_gpio(struct device_node *np, const char *propname, int index)
```

パラメータ：

・np：デバイスノードを指定。

・propname：GPIO 属性名、デバイスツリーで定義された属性名と対応。

株式会社日昇テクノロジー

・index：ピンインデックス値、デバイスツリー内の一つのピン属性に複数のピンが含まれる場合、このパラメータはどのピンを指定するかを示します。

戻り値：

・成功：取得した GPIO 番号（ここでの GPIO 番号はピン属性から生成された非負の整数）。

・失敗：負の数を返します。

2. GPIO 申請関数 gpio_request

リスト 10: gpio_request 関数（カーネルソースコード drivers/gpio/gpiolib-legacy.c 内）

```
1 static inline int gpio_request(unsigned gpio, const char *label);
```

パラメータ：

・gpio: 申請したい GPIO 番号、この値は of_get_named_gpio 関数の戻り値です。

・label: ピンの名前、申請したピンに別名をつけることに相当します。

戻り値：

・成功: 0 を返します。

・失敗: 負の数を返します。

3. GPIO 解放関数

リスト 11: gpio_free 関数（カーネルソースコード drivers/gpio/gpiolib-legacy.c 内）

```
1 static inline void gpio_free(unsigned gpio);
```

gpio_free 関数は gpio_request と対の関数で、一方が申請、もう一方が解放を行います。GPIO は一度に一つの申請しかできませんので、特定のピンを使用しなくなった場合は、それを解放することを忘れないでください。

パラメータ：

・gpio：解放する GPIO の番号。

戻り値：なし

株式会社日昇テクノロジー

4. GPIO 出力設定関数 gpio_direction_output

ピンを出力モードに設定します。

リスト 12: gpio_direction_output 関数（カーネルソースコード include/asm-generic/gpio.h 内）

```
1 static inline int gpio_direction_output(unsigned gpio, int value);
```

関数パラメータ：

・gpio: 設定する GPIO の番号。

・value: 出力値、1 は高電圧、0 は低電圧を意味します。

戻り値：

・成功：0 を返します。

・失敗：負の数を返します。

5. GPIO 入力設定関数 gpio_direction_input

ピンを入力モードに設定します。

リスト 13: gpio_direction_input 関数（カーネルソースコード include/asm-generic/gpio.h 内）

```
1 static inline int gpio_direction_input(unsigned gpio);
```

関数パラメータ：

・gpio: 設定する GPIO の番号。

戻り値：

・成功：0 を返します。

・失敗：負の数を返します。

6. GPIO ピン値取得関数 gpio_get_value

ピンの現在の状態を取得します。ピンが出力モードでも入力モードでも、この関数でピンの現在の状態を取得できます。

リスト 14: gpio_get_value 関数（カーネルソースコード include/asm-generic/gpio.h 内）

株式会社日昇テクノロジー

```
1 static inline int gpio_get_value(unsigned gpio);
```

関数パラメータ：

・gpio: 取得する GPIO の番号。

戻り値：

・成功：取得したピンの状態を返します。

・失敗：負の数を返します。

7. GPIO 出力値設定関数 gpio_set_value

この関数は、出力モードに設定された GPIO にのみ使用します。

リスト 15: gpio_set_value 関数（カーネルソースコード include/asm-generic/gpio.h 内）

```
1 static inline void gpio_set_value(unsigned gpio, int value);
```

関数パラメータ

・gpio：設定する GPIO の番号。

・value：設定する出力値、1 は高電圧、0 は低電圧を意味します。

戻り値：

・成功：0 を返します。

・失敗：負の数を返します。

よく使われる新しいインターフェース関数：

表 1: よく使われる関数

関数	機能
gpiod_get	デバイスツリー内の gpio 属性が一組の値のみの場合に GPIO を取得
gpiod_get_index	デバイスツリー内の gpio が複数組の値を持つ場合に、INDEX を通して GPIO を取得
gpiod_get_direction	GPIO の状態（入力または出力）を取得
gpiod_direction_input	GPIO を入力に設定
gpiod_get_value	GPIO の論理値を取得
gpiod_set_value	GPIO ピンの値を設定

株式会社日昇テクノロジー

gpiod_put	GPIO リソースを解放、gpiod_get および gpiod_get_index で申請されたものに対して使用

詳細な使用方法については、カーネルソースコード内の Documentation/driver-api/gpio/board.rst、Documentation/driver-api/gpio/consumer.rst などを参照してください。

11.2.3 GPIO サブシステムと sysfs

sysfs の GPIO インターフェースも、GPIO の設定や管理に使用できます。ボードの/sys/class/gpio ディレクトリに移動すると、次のように表示されます：

```
cat@lubancat:/sys/class/gpio$ ls
export gpiochip0 gpiochip128 gpiochip32 gpiochip511 gpiochip64 gpiochip96 unexport
cat@lubancat:/sys/class/gpio/gpiochip0$ ls
base device label ngpio power subsystem uevent /* base はそのグループコントローラーの基本ピン番号、ngpio はそのコントローラーの GPIO ピン数、label はデバイスツリーのノードラベルを表します */
```

このディレクトリ内の各 gpiochipX は、一つの GPIO コントローラーを表しています。export と unexport については、export を使用してユーザースペースからそのファイルにピンインデックスを入力し、カーネルに GPIO をリクエストします（まだ申請されていなければ）。その後、ノードが作成されます。unexport はその逆の操作を行います。例えば以下のようにします：

```
cat@lubancat:/sys/class/gpio$ echo 6 | sudo tee export > /dev/null /* export に 6 を入力すると、現在のディレクトリに gpio6 ノードが作成されます */
cat@lubancat:/sys/class/gpio$ ls
export gpio6 gpiochip0 gpiochip128 gpiochip32 gpiochip511 gpiochip64 gpiochip96 unexport
cat@lubancat:/sys/class/gpio$ cd gpio6/ && ls
active_low device direction edge power subsystem uevent value /* direction に"out"を入力し、そのピンを出
```

株式会社日昇テクノロジー

力に設定した後、value に"1"を入力すると、その GPIO は高電圧を出力します */

cat@lubancat:/sys/class/gpio/gpio6$ echo out | sudo tee direction > /dev/null

cat@lubancat:/sys/class/gpio/gpio6$ cat direction

out

cat@lubancat:/sys/class/gpio$ echo 6 | sudo tee unexport > /dev/null /* unexport に 6 を入力すると、現在のディレクトリから gpio6 ノードが削除されます */

cat@lubancat:/sys/class/gpio$ ls

export gpiochip0 gpiochip128 gpiochip32 gpiochip511 gpiochip64 gpiochip96 unexport

重要：これは lubuncat2 ボードを例にしており、GPIO コントローラは 5 つあり、ピンのインデックスは pins = 32*bank + 8*group + x です。bank は 0～4、group は 0～3 で、A～D に対応します。例えば、GPIO1_A7 のピンは pins = 32*0 + 8*0 + 6 = 6 となります。

11.3 GPIO サブシステムと Pinctrl サブシステムの間の結合関係

Pinctrl サブシステムは全てのピンを管理し、GPIO サブシステムはこれらのピンの一つの用途です。GPIO サブシステムは Pinctrl サブシステムに対してピンを申請し、GPIO 機能に設定します。GPIO を申請する際に、その GPIO に対応するピンが他のドライバによって既に別の機能で申請されているかをチェックし、もし申請されていればエラーが報告されます。

第 12 章 Pinctrl サブシステムと GPIO サブシステム — LED 実験

この章では、前章の解説を踏まえ、具体的なコードを書いて GPIO サブシステムを使用し LED ドライバを実装します。GPIO サブシステムは Pinctrl サブシステムを使用します。

12.1 Pinctrl サブシステム

Pinctrl サブシステムは主にチップのピンを管理するために使用されます。これには、ピンの多機能化、

株式会社日昇テクノロジー

ピンのプルアップ/プルダウン、ドライブ能力などが含まれます。rockchip チップには多数のオンチップ外部デバイスがあり、ほとんどの外部デバイスはチップのピンを介して外部デバイス（器具）に接続して対応する制御を実現する必要があります。例えば、よく知る I2C、SPI、LCD、USDHC などです。そして、（電源ピンと特定機能ピンを除く）使用可能なチップのピン数は限られています。チップの設計メーカーはハードウェアデザインの柔軟性を高めるために、一つのチップピンが複数のオンチップ外部デバイスの機能ピンとして機能するようにしています。RK3588 を例に取ると、

「Rockchip_RK3588_Datasheet_xxx」データシートを参照すると、

PIN Name	Func1	Func2	Func3	Func4	Func5
I2C3_SCL_M0/UART3_TX_M0/CAN1_TX_M0/AUDIOPWM_LOUT_N/ACODEC_ADC_CLK/GPIO1_A1_u	GPIO1_A1_u	I2C3_SCL_M0	UART3_TX_M0	CAN1_TX_M0	AUDIOPWM_LOUT_N
I2S1_MCLK_M0/UART3_RTSn_M0/SCR_CLK/PCIE30X1_PERSTn_M2/GPIO1_A2_d	GPIO1_A2_d	I2S1_MCLK_M0	UART3_RTSn_M0	SCR_CLK	PCIE30X1_PERSTn_M2
I2S1_SCLK_TX_M0/UART3_CTSn_M0/SCR_IO/PCIE30X1_WAKEn_M2/ACODEC_DAC_CLK/GPIO1_A3_d	GPIO1_A3_d	I2S1_SCLK_TX_M0	UART3_CTSn_M0	SCR_IO	PCIE30X1_WAKEn_M2
I2S1_SCLK_RX_M0/UART4_RX_M0/PDM_CLK1_M0/SPDIF_TX_M0/GPIO1_A4_d	GPIO1_A4_d	I2S1_SCLK_RX_M0	UART4_RX_M0	PDM_CLK1_M0	SPDIF_TX_M0
I2S1_LRCK_TX_M0/UART4_RTSn_M0/SCR_RST/PCIE30X1_CLKREQn_M2/ACODEC_DAC_SYNC/GPIO1_A5_d	GPIO1_A5_d	I2S1_LRCK_TX_M0	UART4_RTSn_M0	SCR_RST	PCIE30X1_CLKREQn_M2
I2S1_LRCK_RX_M0/UART4_TX_M0/PDM_CLK0_M0/AUDIOPWM_ROUT_P/GPIO1_A6_d	GPIO1_A6_d	I2S1_LRCK_RX_M0	UART4_TX_M0	PDM_CLK0_M0	AUDIOPWM_ROUT_P
I2S1_SDO0_M0/UART4_CTSn_M0/SCR_DET/AUDIOPWM_ROUT_N/ACODEC_DAC_DATAL/GPIO1_A7_d	GPIO1_A7_d	I2S1_SDO0_M0	UART4_CTSn_M0	SCR_DET	AUDIOPWM_ROUT_N
I2S1_SDO1_M0/I2S1_SDI3_M0/PDM_SDI3_M0/PCIE20_CLKREQn_M2/ACODEC_DAC_DATAR/GPIO1_B0_d	GPIO1_B0_d	I2S1_SDO1_M0	I2S1_SDI3_M0	PDM_SDI3_M0	PCIE20_CLKREQn_M2

GPIO1_A6 の機能ピンは GPIO だけでなく、複数の外部デバイスの機能ピンとしても使用できることがわかります。ハードウェアを設計する際には、必要に応じてこれらの中から一つを柔軟に選択できます。ハードウェアが設計されると、各ピンの機能は確定されます。たとえば、上記の二つのピンが他のシリアル制御外部デバイスに接続されている場合、これら二つのピンの機能は UART の受信、送信ピンとして機能します。プログラミングの過程で、ドライバでも、通常は最初にピンの多機能を設定し、ピンの PAD 属性（ドライブ能力、プルアップ/プルダウンなど）を設定します。ドライバプログラムでは、各ピンの多機能設定を手で行う必要がありますが、これは作業量を増やすだけでなく、書かれたドライバプログラムの移植性が悪く、再利用性が低いなどの問題があります。さらに悪いことに、ピンの統一管理が欠如しており、ピンの重複定義が容易に発生します。例えば、I2C のドライバで UART_RX_DATA ピンと UART_TX_DATA ピンを SCL と SDA として多機能化し、UART ドライバを書く際に UART_RX_DATA ピンと UART_TX_DATA ピンが既に使用されていることに気づかずに、再度

株式会社日昇テクノロジー

UART_RX と UART_TX として初期化した場合、IIC ドライバは正常に動作しなくなり、このようなエラーは発見が困難です。

Pinctrl サブシステムはチップメーカーによって実装され、簡単に言えばチップのピンを管理し、ピンの初期化を自動的に完了するのを助けるものです。行う必要があるのは、デバイスツリーで指定された形式に従って、希望する設定パラメータを記述するだけです。

12.1.1 Pinctrl サブシステムの記述形式とピン属性の詳細解説

12.1.1.1 Pinctrl デバイスツリーノード紹介

まず、カーネルソースコード/arch/arm64/boot/dts/rockchip/RK3588.dtsi ファイル内で、以下の定義を見ることができます。

リスト 1: RK3588.dtsi

```
1 pinctrl: pinctrl {
2 compatible = "rockchip,rk3568-pinctrl";
3 rockchip,grf = <&grf>;
4 rockchip,pmu = <&pmugrf>;
5 #address-cells = <2>;
6 #size-cells = <2>;
7 ranges;
8
9  GPIO1: gpio@fd5f8000 {
10 compatible = "rockchip,gpio-bank";
11 reg = <0x0 0xfd5f8000 0x0 0x100>;
12 interrupts = <GIC_SPI 33 IRQ_TYPE_LEVEL_HIGH>;
```

株式会社日昇テクノロジー

```
13 clocks = <&pmucru PCLK_ GPIO1>, <&pmucru DBCLK_ GPIO1>;
14
15 gpio-controller;
16 #gpio-cells = <2>;
17 gpio-ranges = <&pinctrl 0 0 32>;
18 interrupt-controller;
19 #interrupt-cells = <2>;
20 };
21 /* 以下略*/
22 };
```

- compatible：プラットフォームドライバとマッチングする名前を修飾します。ここでは、pinctrl サブシステムのプラットフォームドライバとマッチングします。

- reg：設定レジスタのベースアドレスを表します。

RK3588.dtsi ファイルは、チップメーカーがチップの共通部分を個別に抽出したデバイスツリーの設定です。soc ノードには、必要なピンの設定情報が集約され、pinctrl サブシステムが使用者のノード情報を保存します。

デバイスツリーの主な設定ファイルは、カーネルソースコードの /arch/arm64/boot/dts/rockchip/RK3588-lubancat 4 .dts にあります。RK3588-lubancat 4 .dts ファイルを開き、"&pinctrl"を検索して、デバイスツリー内で「pinctrl」ノードを参照する箇所を見つけます。以下のように表示されます。

株式会社日昇テクノロジー

リスト 2: RK3588-lubancat4.dts 内の&pinctrl 部分の内容

```
1 &pinctrl {
2 pmic {
3 pmic_int: pmic_int {
4 rockchip,pins =
5 <0 RK_PA3 RK_FUNC_GPIO &pcfg_pull_up>;
6 };
7
8 soc_slppin_gpio: soc_slppin_gpio {
9 rockchip,pins =
10 <0 RK_PA2 RK_FUNC_GPIO &pcfg_output_low_pull_down>;
11 };
12
13 soc_slppin_slp: soc_slppin_slp {
14 rockchip,pins =
15 <0 RK_PA2 RK_FUNC_1 &pcfg_pull_up>;
16 };
17
18 soc_slppin_rst: soc_slppin_rst {
19 rockchip,pins =
20 <0 RK_PA2 RK_FUNC_2 &pcfg_pull_none>;
21 };
```

株式会社日昇テクノロジー

```
22

23 spk_ctl_gpio: spk_ctl_gpio {

24 rockchip,pins = <3 RK_PC5 RK_FUNC_GPIO &pcfg_pull_up>;

25 };

26 };

27

28 headphone {

29 hp_det: hp-det {

30 rockchip,pins = <0 RK_PA5 RK_FUNC_GPIO &pcfg_pull_none>;

31 };

32 };

33

34 usb {

35 vcc5v0_usb20_host_en: vcc5v0-usb20-host-en {

36 rockchip,pins = <0 RK_PD5 RK_FUNC_GPIO &pcfg_pull_none>;

37 };

38

39 vcc5v0_usb30_host_en: vcc5v0-usb30-host-en {

40 rockchip,pins = <0 RK_PD6 RK_FUNC_GPIO &pcfg_pull_none>;

41 };

42

43 vcc5v0_otg_vbus_en: vcc5v0-otg-vbus-en {
```

株式会社日昇テクノロジー

```
44 rockchip,pins = <0 RK_PD3 RK_FUNC_GPIO &pcfg_pull_none>;
45 };
46 };
47 /* 以下略*/
48 };
```

ここでは、「&pinctrl」を使用して「pinctrl」ノードの下に内容を追加します。デバイスツリーソースコードの紹介は以下の通りです：

- 第 2 行：「pmic」ノードは、名前からそのノードが「pmic」の電源管理のピン機能を記述していることが分かります。その最初の子ノード「pmic_int」では、「rockchip,pins」を使用してピンの多機能性を指定しています。

- 第 29-30 行：「headphone」子ノードが使用するピンの電気特性を指定しています。

- その他のソースコードは、pinctrl の下の子ノードで、それぞれが使用する外部デバイスのピンとそれに対応する多機能性を記述しています。これらは特定のフォーマットに従って記述されています。

pinctrl をいつ使用するかについては、「&sdmmc1」の外部デバイスノードを例にとります。

```
1 &sdmmc1 {
2 pinctrl-names = "default", "opendrain", "sleep";
3 pinctrl-0 = <&sdmmc1_b4_pins_a>;
4 pinctrl-1 = <&sdmmc1_b4_od_pins_a>;
5 pinctrl-2 = <&sdmmc1_b4_sleep_pins_a>;
6 broken-cd;
7 st,neg-edge;
8 bus-width = <4>;
9 vmmc-supply = <&v3v3>;
```

株式会社日昇テクノロジー

```
10 status = "okay";

11 };
```

- pinctrl-names：sdmmc1 外部デバイスが使用する 3 種類のピン状態を記述します。それぞれ default、opendrain、sleep です。

- pinctrl-0：外部デバイスが default 状態にある場合、pinctrl-0 で参照されるピン設定 &sdmmc1_b4_pins_a を使用します。

- pinctrl-1：外部デバイスが opendrain 状態にある場合、pinctrl-1 で参照されるピン設定 &sdmmc1_b4_od_pins_a を使用します。

- pinctrl-2：外部デバイスが sleep 状態にある場合、pinctrl-2 で参照されるピン設定 &sdmmc1_b4_sleep_pins_a を使用します。

このようにして、この外部デバイスが使用するピンとその状態を指定します。

12.1.1.2 pinctrl の子ノードの記述フォーマット

「&pinctrl」の下のノードの記述形式に従って、特定の外部デバイスの pinctrl を自分で記述することもできます。

リスト 3: 例

```
&pinctrl {
    xxx: xxx {
        pins {
            rockchip,pins = <0 RK_PA6 RK_FUNC_GPIO &pcfg_pull_none>;
        };
    };
};
```

株式会社日昇テクノロジー

上記の外部デバイス xxx は、使用するピンが GPIO1_A7 で、「RK_FUNC_GPIO」で多機能性を GPIO として設定し、「&pcfg_pull_none」でプルアップ/プルダウンを設定していません。この参照ノードは、カーネルソースコードの arch/arm64/boot/dts/rockchip/rockchip-pinconf.dtsi ファイルを参照できます。

各チップメーカーの pinctrl 子ノードの記述フォーマットは異なり、これはデバイスツリーの規格ではなく、チップメーカーが独自に定義したものです。自分の pinctrl ノードを追加したい場合は、上記のフォーマットに従って記述すれば良いです。

pinctrl ノードの記述方法については、カーネルのドキュメントディレクトリ内でチップメーカーが提供するドキュメントを探すことができます。例えば、rockchip の公式 pinctrl ドキュメントは以下のディレクトリにあります：

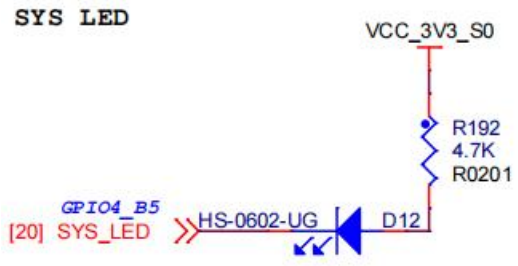
Documentation/devicetree/bindings/pinctrl/rockchip,pinctrl.txt

12.1.2 RGB ライトのピンを pinctrl サブシステムに追加

この小節では、回路図から始めて、LED ライトに使用されるピンを pinctrl サブシステムに一歩ずつ追加していきます。具体的なボードでピンが異なる場合は、実際のボードの回路図を参照してください。

12.1.2.1 LED ライトに使用されるピンの検索

lubuncat4 を例にすると、システム LED ライトに対応する回路図は以下の通りです。ネットワーク名からコアボード上の対応するピンが GPIO4_B5 であることが分かります。



株式会社日昇テクノロジー

12.1.2.2 pinctrl ノードに pinctrl 子ノードを追加

子ノードの追加は非常に簡単です。引脚情報を一定のフォーマットで対応するデバイスツリーファイルの pinctrl 子ノードに書き込むだけです。

lubuncat4 を例に、rk3588s-lubancat-4.dts に以下の内容を追加します。

リスト 5: pinctrl 子ノードの追加

```
&pinctrl {
  /*----------新規追加内容--------------*/
  led_test {
    led_test_pin: led_test_pin {
      rockchip,pins = <4 RK_PB5 RK_FUNC_GPIO &pcfg_pull_none>;
    };
  };
};
```

新しく追加されたノード名は「led_test」で、名前は任意（同一ノード内で重複しないこと）で、32 文字を超えない範囲でノードの情報を表すことが望ましいです。「led_test_pin」はノードタグで、「rockchip,pins」は固定のフォーマットですが、後ろの内容はカスタマイズされます。これにより、このタグを使用してこのノードを参照します。

pins の内容では、LED が使用する GPIO ピンの機能を設定しました。pinctrl は各チップメーカーごとに異なるため、ここでは詳しく説明しませんが、具体的には公式の Documentation/devicetree/bindings/pinctrl/rockchip,pinctrl.txt 文書を参照してください。pinctrl 子ノードを追加完了後、システムは追加した設定情報に基づいてピンを GPIO 機能として初期化します。これにより、pinctrl サブシステムの使用についての説明は終わりです。次に、GPIO サブシステムに関連する内

株式会社日昇テクノロジー

容を紹介します。

12.2 GPIO サブシステム

GPIO サブシステムを使用する前は、LED を点灯させたい場合には、まず LED に関連する設定レジスタを取得し、それらのレジスタを手で読み書きして LED を制御する必要がありました。GPIO サブシステムがあれば、この作業は GPIO サブシステムが代わりに行ってくれます。、GPIO サブシステムが提供する API 関数を呼び出すだけで GPIO の制御操作を完了できます。

RK3588.dtsi ファイルの pinctrl 子ノードには GPIO コントローラのレジスタアドレスが記録されています。ここでは GPIO1 を例にして、 GPIO1 サブノードに関連する内容を紹介します。

リスト 6: RK3588.dtsi 内の GPIOA ノードの内容

```
/ {
pinctrl: pinctrl {
compatible = "rockchip,rk3568-pinctrl";
rockchip,grf = <&grf>;
rockchip,pmu = <&pmugrf>;
#address-cells = <2>;
#size-cells = <2>;
ranges;
 GPIO1: gpio@fd5f8000 {
compatible = "rockchip,gpio-bank";
reg = <0x0 0xfd5f8000 0x0 0x100>;
interrupts = <GIC_SPI 33 IRQ_TYPE_LEVEL_HIGH>;
clocks = <&pmucru PCLK_ GPIO1>, <&pmucru DBCLK_ GPIO1>;
gpio-controller;
```

株式会社日昇テクノロジー

```
#gpio-cells = <2>;

gpio-ranges = <&pinctrl 0 0 32>;

interrupt-controller;

#interrupt-cells = <2>;

};

/* 以下略*/

    };

};
```

- compatible：GPIO サブシステムのプラットフォームドライバとのマッチングを示します。

- reg： GPIO1 デバイスレジスタのベースアドレス。gpioa の reg 属性では、GPIOA のレジスタグループのマップアドレスが fd5f8000 で、範囲は`0x100`です。

- interrupts：割り込みコントロール情報を表し、使用される割り込み番号はすべて SPI 割り込み源です。

- clocks：GPIO デバイスのクロック情報の初期化を表します。

- gpio-controller：gpioa が GPIO コントローラであることを示します。

- #gpio-cells：GPIO ピンを記述するためにいくつのセルが使用されるかを示し、ここでは 2 が使用されています。1 つのセルはどのピンかを記述し、もう 1 つのセルは有効レベルを記述します。

- gpio-ranges：GPIO ピンのマッピング範囲を示します。

- #interrupt-controller：割り込みコントローラであることを示します。

- #interrupt-cells：1 つの割り込みを記述するために何セルが使用されるかを示します。

この情報は一般的にチップメーカーによって完全に提供されます。 GPIO1 ノードは、 GPIO1 に関する全体を記述しています。GPIO サブシステムを使用する際には、デバイスツリーにデバイスノードを追加し、GPIO サブシステムが提供する API を使用して GPIO の制御を実現します。

株式会社日昇テクノロジー

12.2.1 デバイスツリーに RGB ライトのデバイスノードを追加

以前のデバイスツリー LED デバイスノード（GPIO サブシステムを使用していない）と比較して、GPIO 属性定義のみが必要です。GPIO サブシステムをベースにした led_test デバイスノードです。

lubuncat4 の例では、rk3588s-lubancat-4.dts のデバイスツリーのルートノードに以下の内容を追加します。

リスト 8: デバイスツリーに led_test ノードを追加

```
1 /* led_test ノードを追加 */
2 led_test: led_test {
3     status = "okay";
4     compatible = "fire,led_test";
5     default-state = "on";
6     gpios = <&gpio4 RK_PB5 GPIO_ACTIVE_HIGH>;
7     pinctrl-names = "default";
8     pinctrl-0 = <&led_test_pin>;
9 };
```

- 第 4 行：compatible 属性値を設定し、LED のプラットフォームドライバとマッチングします。

- 第 8 行：RGB ライトのピンの pinctrl 情報を指定します。前のセクションで pinctrl ノードを定義し、タグ led_test_pin を設定しました。ここでその pinctrl 情報を参照しています。

- 第 6 行：特定のピンを参照します。一般的に[name]-gpios 形式を使用し、どの GPIO コントローラ、またはどのピングループを使用するかを指定します。ピンのインデックスと有効レベルを指定します。

株式会社日昇テクノロジー

12.2.2 デバイスツリーで sys ライトのデバイスノードをコメントアウト

システム LED ライトはカーネルの組み込み LED ドライバを使用していますが、この実験と競合するため、それを無効にする必要があります。leds ノードを見つけて、`status`を`disabled`に設定し、leds ノードを無効にします。

リスト 9: leds の競合を無効化

```
1 leds: leds {
2    /*status = "okay";*/
3    status = "disabled";
4    compatible = "gpio-leds";
5
6    sys_status_led: sys-status-led {
7        label = "sys_status_led";
8        linux,default-trigger = "heartbeat";
9        default-state = "on";
10       gpios = <& GPIO1 RK_PC7 GPIO_ACTIVE_LOW>;
11       pinctrl-names = "default";
12       pinctrl-0 = <&sys_status_led_pin>;
13   };
14 };
```

または、対応するボードの dts に以下のように追加して、leds ノードを`disabled`に設定することもできます。

株式会社日昇テクノロジー

リスト 10: leds ノードの競合を追加で無効化

```
1 &leds {
2     status = "disabled";
3 };
```

12.2.3 デバイスツリーをコンパイル、ダウンロードして結果を検証

前の 2 セクションで、LED ライトに使用されるピンを pinctrl サブシステムに追加し、led_test デバイスノードをデバイスツリーに追加しました。このセクションでは、変更されたデバイスツリーをコンパイルしてダウンロードし、新しいデバイスツリーでシステムを起動して、led_test デバイスノードが生成されたかどうかを確認します。

カーネルのコンパイル時には自動的にデバイスツリーもコンパイルされますが、そのデメリットはコンパイル時間が長くなることです。カーネルディレクトリ下で以下のコマンドを実行し、デバイスツリーのみをコンパイルします：

rk3588 ボードの場合は以下のコマンドを実行します：

```
# 設定ファイルを読み込む
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- lubancat_linux_rk3588_defconfig
# dtbs パラメータを使用してデバイスツリーのみをコンパイル
make ARCH=arm64 -j4 CROSS_COMPILE=aarch64-linux-gnu- dtbs
```

コンパイルが成功すると、「カーネルソースコード arch/arm64/boot/dts/rockchip/」ディレクトリ下に対応する dtb ファイルが生成されます。これをボードの「/boot/dtb/」ディレクトリに置き換えて開発ボードを再起動します。

新しいデバイスツリーで再起動した後、通常は開発ボードの「/proc/device-tree」ディレクトリ下に「led_test」デバイスツリーノードが生成されます。以下のように表示されます。

株式会社日昇テクノロジー

```
root@lubancat:~# ls /proc/device-tree/led*
/proc/device-tree/led_test:
compatible      gpios  phandle     pinctrl-names
default-state   name   pinctrl-0   status

/proc/device-tree/leds:
compatible  name  phandle  status  sys-status-led
root@lubancat:~# cat /proc/device-tree/led_test/status
okayroot@lubancat:~#
root@lubancat:~# cat /proc/device-tree/leds/status
disabledroot@lubancat:~#
root@lubancat:~#
```

このように、「/proc/device-tree」ディレクトリ下に「led_test」と「leds」ノードが存在し、「led_test」ノードの状態は「okay」で、「leds」ノードの状態は「disabled」です。

これで、デバイスがシステムに追加されました。次に、LED デバイスを使用するためのドライバを書くことができます。しかし、その前に、前章の GPIO サブシステムのまとめでいくつかの関数を確認しておくと良いでしょう。

12.3 実験説明とコード解説

12.3.1 実験コード解説

この章のサンプルコードディレクトリは「linux_driver/gpio_subsystem_led」です。

プログラムには 2 つの C 言語ファイルが含まれています。1 つはドライバプログラムで、プラットフォームバスに基づいて書かれています。もう 1 つはドライバが正常に機能するかをテストするためのシンプルなテストプログラムです。

12.3.1.1 ドライバプログラム解説

ドライバプログラムは大きく 3 つの部分に分かれています。第一部分は、プラットフォームデバイスドライバのエントリーとエクジット関数を記述します。第二部分は、プラットフォームデバイスの.probe 関数を記述し、この関数内でキャラクタデバイスの登録と LED ライトの初期化を行います。第三部分では、キャラクタデバイスの関数セットを記述し、open 関数と write 関数を実装します。

プラットフォームドライバのエントリーとエクジット関数の実装

ソースコードは以下の通りです：

株式会社日昇テクノロジー

リスト 11: プラットフォームドライバフレームワーク

```
1 static const struct of_device_id led_ids[] = {
2     { .compatible = "fire,led_test"},
3     { /* センチネル */ }
4 };
5
6 /* プラットフォームデバイス構造体の定義 */
7 struct platform_driver led_platform_driver = {
8     .probe = led_probe,
9     .driver = {
10         .name = "leds-platform",
11         .owner = THIS_MODULE,
12         .of_match_table = led_ids,
13     }
14 };
15
16 /*
17 *ドライバ初期化関数
18 */
19 static int __init led_platform_driver_init(void)
20 {
21     int DriverState;
```

株式会社日昇テクノロジー

```
22
23    DriverState = platform_driver_register(&led_platform_driver);
24
25    printk(KERN_EMERG "¥tDriverState is %d¥n",DriverState);
26    return 0;
27 }
28
29 /*
30 * ドライバ登録解除関数
31 */
32 static void __exit led_platform_driver_exit(void)
33 {
34    printk(KERN_EMERG "led_test exit!¥n");
35    /* デバイスの削除 */
36    device_destroy(class_led, led_devno); // デバイスのクリーンアップ
37    class_destroy(class_led); // クラスのクリーンアップ
38    cdev_del(&led_chr_dev); // キャラクタデバイス番号のクリーンアップ
39    unregister_chrdev_region(led_devno, DEV_CNT); // キャラクタデバイスの登録解除
40
41    platform_driver_unregister(&led_platform_driver);
42 }
43
```

株式会社日昇テクノロジー

```
44 module_init(led_platform_driver_init);

45 module_exit(led_platform_driver_exit);

46

47 MODULE_LICENSE("GPL");
```

- 第 2-15 行：コードの第一部分では、.probe 関数と.driver のみを実装しています。ドライバとデバイスが成功裏にマッチした後に実行されるこの関数の実装は後ほど紹介します。.driver はこのドライバの属性を記述し、.name はドライバの名前、.owner はドライバの所有者、.of_match_table はドライバマッチングテーブルを指します。このマッチングテーブルには「led_test」という一つのマッチング値「.compatible = "fire,led_test"」が含まれています。この値はデバイスツリー内の led_test デバイスノードの`compatible`属性と一致する必要があります。

- 第 17-40 行：第二部分と第三部分はプラットフォームデバイスのエントリー関数とエクジット関数で、関数の実装は非常にシンプルです。エントリー関数ではプラットフォームドライバを登録し、エクジット関数ではプラットフォームドライバを登録解除します。

プラットフォームドライバの.probe 関数の実装

ドライバとデバイスがマッチした後に最初に実行されるのが probe 関数です。この関数内で RGB の初期化やキャラクタデバイスの登録を行います。その後、キャラクタデバイスの操作関数（open、write）内で RGB などの制御を実現します。関数のソースコードは以下の通りです。

リスト 12: probe 関数の実装

```
1 /*----------------プラットフォームドライバ関数集-----------------*/

2 static int led_probe(struct platform_device *pdv)

3 {

4     int ret = 0; // デバイス番号の申請結果を保存
```

株式会社日昇テクノロジー

```
5    printk("match successed¥n");
6
7    /* RGB のデバイスツリーノードを取得 */
8    led_device_node = of_find_node_by_path("/led_test");
9    if(led_device_node == NULL)
10   {
11       printk(KERN_EMERG "get led_test failed! ¥n");
12   }
13
14   led = of_get_named_gpio(led_device_node, "gpios", 0);
15   printk("led = %d ¥n", led);
16
17   /* GPIO を出力モードに設定し、高電圧を出力 */
18   gpio_direction_output(led, 1);
19
20   /*--------------------キャラクタデバイス登録部分-----------------*/
21
22   // 第一ステップ
23   // 動的にデバイス番号を割り当て、デバイス名は"rgb-leds"とし、`cat /proc/devices`で確認できる
24   // DEV_CNT は 1 で、現在は 1 つのデバイス番号のみを申請
25   ret = alloc_chrdev_region(&led_devno, 0, DEV_CNT, DEV_NAME);
26   if(ret < 0){
```

株式会社日昇テクノロジー

```
27      printk("fail to alloc led_devno¥n");
28      goto alloc_err;
29  }
30
31  // 第二ステップ
32  // キャラクタデバイス構造体 cdev とファイル操作構造体 file_operations を関連付ける
33  led_chr_dev.owner = THIS_MODULE;
34  cdev_init(&led_chr_dev, &led_chr_dev_fops);
35
36  // 第三ステップ
37  // デバイスを cdev_map ハッシュテーブルに追加
38  ret = cdev_add(&led_chr_dev, led_devno, DEV_CNT);
39  if(ret < 0)
40  {
41      printk("fail to add cdev¥n");
42      goto add_err;
43  }
44
45  // 第四ステップ
46  /* クラスを作成 */
47  class_led = class_create(THIS_MODULE, DEV_NAME);
48
```

株式会社日昇テクノロジー

```
49    /* デバイスを作成 */
50    device = device_create(class_led, NULL, led_devno, NULL, DEV_NAME);
51
52    return 0;
53
54 add_err:
55    // デバイスの追加に失敗した場合、デバイス番号を登録解除する
56    unregister_chrdev_region(led_devno, DEV_CNT);
57    printk("¥n error! ¥n");
58 alloc_err:
59    return -1;
60 }
```

- 第 10-14 行：of_find_node_by_path 関数を使用してデバイスツリー内の`led_test`ノードを取得します。"/led_test"は取得したいデバイスツリーノードのパスです。ノードが他の子ノードにネストされている場合は、そのノードが位置する完全なパスを指定する必要があります。

- 第 17-22 行：of_get_named_gpio 関数を使用して GPIO 番号を取得します。成功すると、読み取った GPIO 番号が返されます。"gpios"は GPIO の名前を指定し、このパラメータは`led_test`デバイスツリーノード内の GPIO 属性名と一致する必要があります。パラメータ"`0`"はピンのインデックスを指定し、属性に定義されたピンが 1 つしかないため`0`に設定されます。

- 第 25-27 行：GPIO を出力モードに設定し、デフォルトで高電圧を出力します。

- 第 32-65 行：キャラクタデバイス関連の内容です。この部分はキャラクタデバイスの章で詳しく説明されていますので、ここでは割愛します。

キャラクタデバイスの関数の実装では、open 関数と write 関数のみを実装する必要があります。関数の

株式会社日昇テクノロジー

ソースコードは以下の通りです。

リスト 13: open 関数と write 関数の実装

```
1 /*-----------------第一部分---------------*/
2 /* キャラクタデバイス操作関数集 */
3 static struct file_operations led_chr_dev_fops =
4 {
5     .owner = THIS_MODULE,
6     .open = led_chr_dev_open,
7     .write = led_chr_dev_write,
8 };
9
10 /*-----------------第二部分---------------*/
11 /* キャラクタデバイス操作関数集、open 関数 */
12 static int led_chr_dev_open(struct inode *inode, struct file *filp)
13 {
14     printk("open ¥n");
15     return 0;
16 }
17
18 /*-----------------第三部分---------------*/
18 /* キャラクタデバイス操作関数集、write 関数 */
20 static ssize_t led_chr_dev_write(struct file *filp, const char __user *buf, size_t cnt, loff_t *offt)
```

株式会社日昇テクノロジー

```
21 {
22     unsigned char write_data; // 受け取ったデータを保存
23
24     int error = copy_from_user(&write_data, buf, cnt);
25     if(error < 0) {
26         return -1;
27     }
28
29     /* LED のピン出力電圧を設定 */
30     if(write_data)
31    {
32        gpio_direction_output(led, 1); // ピン出力高電圧、LED オフ
33     }
34     else
35     {
36        gpio_direction_output(led, 0); // ピン出力低電圧、LED オン
37     }
38
39     return 0;
40 }
```

- コードの 3-8 行: キャラクタデバイスの操作関数セットを定義します。ここでは open 関数と write 関数を主に実装します。

- コード 12-16 行：open 関数を実装します。プラットフォームドライバの probe 関数で既に GPIO

株式会社日昇テクノロジー

が初期化されているため、ここでは何も操作する必要はありません。

- コード 20-39 行：write 関数の実装も非常にシンプルです。まず、アプリケーション層からのデータをカーネル層に「コピー」するために copy_from_user 関数を使用します。そして、コマンド値に基づいて gpio_direction_output 関数を使用して LED のオン/オフを制御します。

12.3.1.2 アプリケーションプログラムの説明

アプリケーションのプログラミングは比較的シンプルです。デバイスノードファイルを開いて、コマンドを書き込んでからデバイスノードファイルを閉じるだけです。ソースコードは以下の通りです。

リスト 14: Makefile ファイル

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <fcntl.h>
4 #include <string.h>
5
6 int main(int argc, char *argv[])
7 {
8     printf("led test¥n");
9     /* 入力コマンドの正当性をチェック */
10    if(argc != 2)
11    {
12        printf(" command error ! ¥n");
13        printf(" usage : sudo test_app num [num can be 0 or 1]¥n");
```

株式会社日昇テクノロジー

```
14      return -1;
15  }
16
17  /* ファイルを開く */
18  int fd = open("/dev/led_test", O_RDWR);
19  if(fd < 0)
20  {
21      printf("open file : %s failed !¥n", argv[0]);
22      return -1;
23  }
24
25  unsigned char command = atoi(argv[1]); //受け取ったコマンド値を数値に変換;
26
27  /* コマンドを書き込む */
28  int error = write(fd, &command, sizeof(command));
29  if(error < 0)
30  {
31      printf("write file error! ¥n");
32      close(fd);
33      /* 閉じる成功かどうかをチェック */
34  }
35
```

株式会社日昇テクノロジー

```
36    /* ファイルを閉じる */
37    error = close(fd);
38    if(error < 0)
39    {
40        printf("close file error! ¥n");
41    }
42
43    return 0;
44 }
```

コードの各部分についての説明は以下の通りです：

- コード 10-15 行：コマンドが有効かどうかを判断します。アプリケーションを実行する際には、制御コマンドを渡す必要があるため、パラメータの長さは 2 です。
- コード 19-24 行：デバイスファイルを開きます。`"/dev/led_test"` はデバイスノードファイルを指定し、このデバイスノード名はドライバプログラム内で設定されたものと一致させる必要があります。
- コード 26-43 行：main 関数から取得したパラメータは文字列なので、まずそれを数値に変換します。最後に write 関数を呼び出してコマンドを書き込んでからファイルを閉じます。

12.3.2 実験準備

ボード上の一部の GPIO はシステムによって使用されている可能性があるため、コードを実行すると「Device or resource busy」や実行コードがフリーズするなどの現象が発生する場合があります。デバイスツリープラグインをコメントアウトするか、デバイスツリーを変更してからシステムを再起動し、関連する GPIO ピンを解放する必要があります。

重要:「Permission denied」や類似のメッセージが表示された場合は、ユーザー権限に注意してください。

株式会社日昇テクノロジー

ほとんどのハードウェアデバイスを操作する機能は root ユーザー権限が必要です。簡単な解決策は、コマンドの前に sudo を追加するか、root ユーザーとしてプログラムを実行することです。

12.3.2.1 Makefile の変更説明

Makefile を変更してドライバプログラムをコンパイルします。

変更後の Makefile は以下の通りです。

リスト 15: Makefile ファイル

```
1 KERNEL_DIR=../../kernel/

2

3 ARCH=arm64

4 CROSS_COMPILE=aarch64-linux-gnu-

5 export ARCH CROSS_COMPILE

6

7 obj-m := led_test.o

8 out = led_app.o

9

10 all:

11 $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) modules

12 $(CROSS_COMPILE)gcc -o $(out) led_app.c

13

14 .PHONE:clean

15
```

株式会社日昇テクノロジー

```
16 clean:

17 $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) clean
```

- コード第 2 行：変数「KERNEL_DIR」にはカーネルのパスが保存されています。これは自分のカーネルが存在する場所に基づいて設定する必要があります。

- コード第 7 行：「obj-m := led_test.o」の「led_test.o」はドライバのソースコード名と対応させる必要があります。Makefile を修正した後、以下のコマンドを実行してドライバをコンパイルします。

対応するドライバディレクトリでコマンドを実行します：

```
make
```

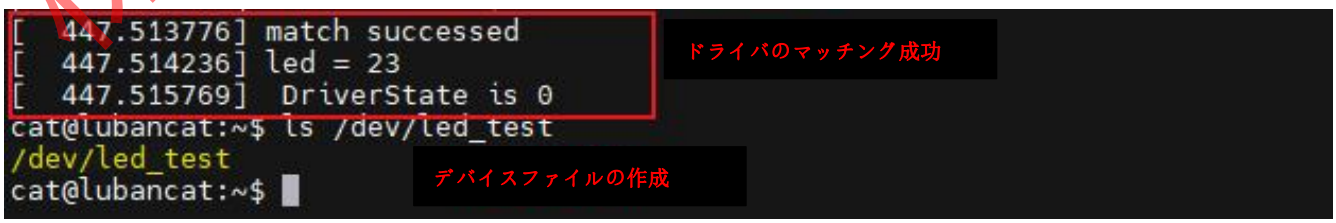
正常にコンパイルされると、現在のディレクトリに「led_test.ko」ドライバファイルと「led_app」アプリケーションが生成されます。

12.3.3 ダウンロードと検証

前の 2 セクションで、.ko ドライバとアプリケーションをコンパイルしました。ドライバとアプリケーションを開発ボードに追加します（scp や NFS 共有フォルダーの使用を推奨）。その後、以下のコマンドを実行してドライバをロードします：

```
insmod ./led_test.ko
```

ドライバ内で、プローブ関数でキャラクタデバイスを登録し、デバイスファイルを作成しています。デバイスとドライバが正常にマッチした場合、プローブ関数は既に実行されているので、「/dev/」ディレクトリ下に「led_test」デバイスノードが生成されているはずです。以下のように表示されます。

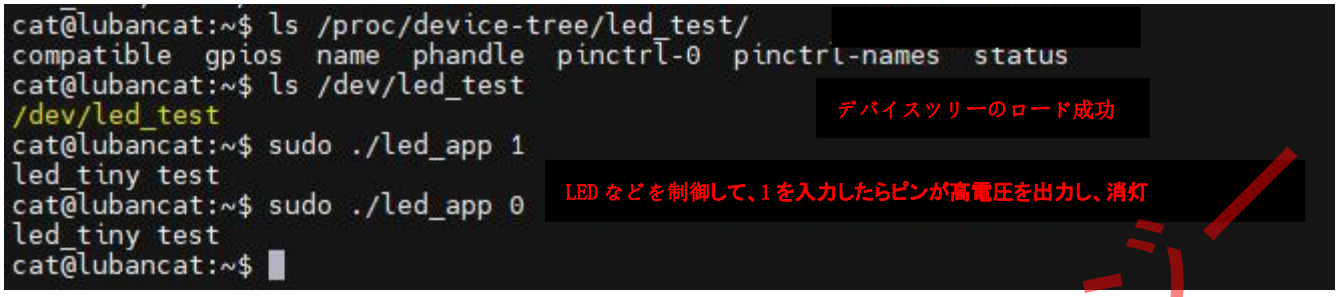


ドライバが正常にロードされた後、直接アプリケーションを実行します。

株式会社日昇テクノロジー

```
1 ./led_app <コマンド>
```

実行結果は以下の通りです：



コマンドは「unsigned char」型のデータで、0 を入力するとピンの出力が低電圧になり、ランプが点灯します。1 を入力するとピンの出力が高電圧になり、ランプが消灯します。

第 13 章 割り込みサブシステム

この章では、割り込みに関連する知識を説明し、カーネルの割り込みフレームワークと割り込みの概念を理解します。ARM の割り込みコントローラ（GIC v3）に関連する内容は、主にリファレンスマニュアルを参照して簡単に説明します。また、カーネルの割り込みコントローラの初期化ソースコードの簡単な分析と、割り込みドライバを作成する際によく使用される API についても説明します。

この章では、以下の部分を簡単に紹介します：

- 割り込みサブシステムフレームワーク
- GIC v3 割り込みコントローラ
- カーネル割り込みドライバのコアの簡単な分析
- 割り込み関連 API とデータ構造

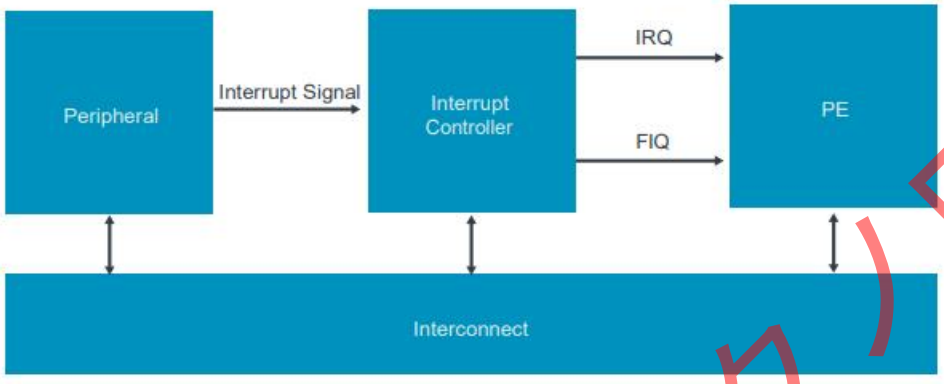
13.1 割り込みサブシステムフレームワーク

割り込みとは、CPU が通常のプログラムを実行している間に、内部または外部のイベント、または事前に設定されたイベントによって CPU が一時的に現在実行中のプログラムを停止し、その内部または外部のイベントのサービスプログラムに切り替え、サービスが完了した後に元のプログラムの実行を続けることを指します。

株式会社日昇テクノロジー

13.1.1 割り込みハードウェアの簡単な説明

割り込みハードウェアには主に 3 種類のデバイスが関与しています：各種外部デバイス、割り込みコントローラ、そして CPU です。これらの関係は以下の図で簡単に説明されます：



注：ARMv8 アーキテクチャでは、プロセッサがトランザクションを処理する抽象プロセスを PE（Processing Element）と定義しています。PE を単純にプロセッサコアとして理解することができます。

1. 外部デバイスは、割り込みイベントが発生すると、外部デバイス上の電気信号を介して割り込みコントローラに処理を要求します。

2. 割り込みが増え続けると、割り込みを管理するための専用デバイスが必要になります。割り込みコントローラはその役割を果たし、外部デバイスの割り込みシステムと CPU システムの橋渡しをします。外部からの割り込みを受け取り、処理した後、プロセッサに割り込み信号を報告します。例えば、ARM の場合、GIC 割り込みコントローラは外部からの割り込み信号を受け取り、FIQ や IRQ などに処理してから ARM コアに送信します。

3. CPU の主な機能は計算です。CPU は割り込みの優先順位を処理せず、割り込みコントローラからの割り込み情報のみを受け取り、割り込み処理を実行します。例えば、ARM では、CPU は IRQ や FIQ 信号を受け取り、それぞれ IRQ モードや FIQ モードに入ります。

株式会社日昇テクノロジー

13.1.2 ソフトウェアフレームワーク

Linux カーネルのソフトウェア部分では、割り込みサブシステムが 4 つの部分に分けられます：

1. ハードウェアに依存しないコード、これをカーネルの汎用割り込み処理モジュールと呼びます。CPU やコントローラの種類に関わらず、割り込み処理の過程にはいくつか共通の内容があり、これらの共通内容はハードウェアとは独立して抽象化されています。さらに、各外部デバイスのドライバコードでは、統一されたインターフェースを用いて IRQ の管理を実現し、これらのコードがカーネル割り込みサブシステムの核心部分を形成しています。

2. CPU アーキテクチャに関連する割り込み処理。特定の CPU アーキテクチャに関連します。

3. 割り込みコントローラのドライバコード。使用される割り込みコントローラに関連します。

4. 一般的な他のドライバ。これらのドライバは、カーネルの汎用割り込み処理モジュールの API を使用して独自のドライバロジックを実現します。

割り込みに関連するいくつかの一般的な概念を以下に挙げます：

- HW interrupt ID:

ハードウェア割り込み ID、つまり GIC ハードウェア上の irq ID で、GIC 標準で定義されています。GIC のセクションで説明します。

- IRQ number:

IRQ number はソフトウェア側で定義され、ハードウェアとは無関係です。CPU は各外部デバイス割り込みに番号を割り当てる必要があり、これを IRQ Number と呼びます。この IRQ number は、CPU が外部デバイス割り込みを識別するために使用する仮想の interrupt ID です。

- IRQ domain:

irq domain は、ある種のリソースを異なる領域に分割することを意味し、同じドメイン内ではいくつかの共通の属性が共有されます。domain は実際にはモジュール化の一形態であり、irq domain の生

株式会社日昇テクノロジー

成は GIC のカスケードサポートによるものです。実際には、カーネルは各 GIC に異なるドメインを設定していますが、その唯一の目的は GIC 内の hwirq と論理 irq のマッピングを担当することです。

- 割り込みベクターテーブル:

割り込みベクターテーブルは、割り込みベクターのリストで、メモリ内に格納され、テーブル内には割り込みソースに対応する割り込み処理プログラムのエントリアドレスが含まれています。

- 割り込み処理アーキテクチャ:

割り込みはカーネルプロセスの通常のスケジューリングと実行を割り込みします。より高いスループットを追求するシステムでは、割り込みサービスプログラムができるだけ短く効率的であることが要求されます。実際には、割り込みには処理が必要なプログラムが多くあります。一部の重い割り込みサービスプログラムについては、カーネルの割り込み処理プログラムが上半分と下半分の 2 つの部分に分割されます。上半分は簡単な緊急の機能（割り込みフラグのクリアなど）を処理し、主なタスクは下半分で処理されます。

13.2 GIC v3 割り込みコントローラーの概要

ARM マルチコアプロセッサで最も一般的に使用される割り込みコントローラーは GIC で、Generic Interrupt Controller の略です。柔軟で拡張可能な割り込み管理方法を提供し、単一コアシステムから数百コアの大規模なマルチチップ設計までをサポートします。主な機能は、ハードウェアからの割り込み信号を受け取り、特定の設定戦略を通じて、対応する CPU で処理するために配布することです。

GIC は ARM 社によって提案された設計仕様で、現在 4 つのバージョン、GIC v1～v4 があります。設計仕様で最も一般的に使用されているのは、V2.0、V3.1、V4.1 の 3 つのバージョンで、GICv3 バージョンは主に Armv8-A、Armv9-A などのアーキテクチャで動作します。ARM 社は実際のコントローラー設計の参照を提供しており、たとえば GIC-400（GIC v2 アーキテクチャをサポート）、GIC-500（GIC v3 アーキテクチャをサポート）、GIC-600（GIC v3 および GIC v4 アーキテクチャをサポート）などです。最終的に、チップメーカーは自分で GIC を実装するか、ARM が提供する設計を直接購入することができ

株式会社日昇テクノロジー

ます。

この章では、GIC V3 の基本構造と実装方法について簡単に説明します。より詳細な紹介は「Arm® Generic Interrupt Controller Architecture Specification」を参照してください。

13.2.1 GIC v3 割り込みタイプ

GIC v3 が処理する異なる種類の割り込み源には以下があります：

- SGI（Software Generated Interrupt）：ソフトウェアによって引き起こされる割り込み。GICD_SGIR レジスタに書き込むことでソフトウェアが割り込みイベントを引き起こすことができ、主にコア間通信に使用されます。カーネルの IPI（inter-processor interrupts）は SGI に基づいています。

- PPI（Private Peripheral Interrupt）：プライベートペリフェラル割り込み、特定のコアで処理される外部デバイスからの割り込みです。GIC はマルチコアをサポートしており、各コアには独自の割り込みがあります。

- SPI（Shared Peripheral Interrupt）：共有ペリフェラル割り込み、すべてのコアが共有する割り込みです。割り込みが発生すると、特定の CPU に配布されます。

- LPI（Locality-specific Peripheral Interrupt）：LPI は GICv3 で導入されたもので、他の三つのタイプの割り込みとは非常に異なるプログラミングモデルを持っています。LPI はメッセージベースの割り込みで、その設定はレジスタではなくテーブルに保存されます。

各割り込みには INTID と呼ばれる ID 番号があります。以下は ARM GIC v3 マニュアルの割り込み番号規定です：

表 1: 中断 ID と中断タイプ

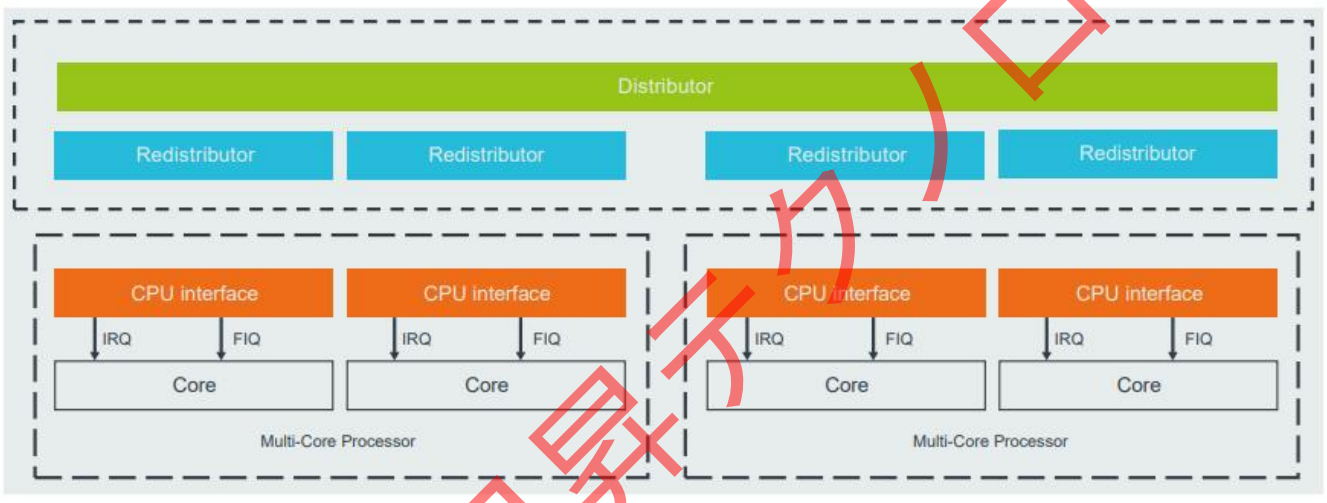
INTID	割り込みタイプ	説明
0-15	SGI	各 CPU コアにはそれぞれ 16 個があります
16-31	PPI	各 CPU コアにはそれぞれ 16 個があります
32-1019	SPI	SPI 例えば GPIO 中断、シリアルポート中断などの外部中断で、

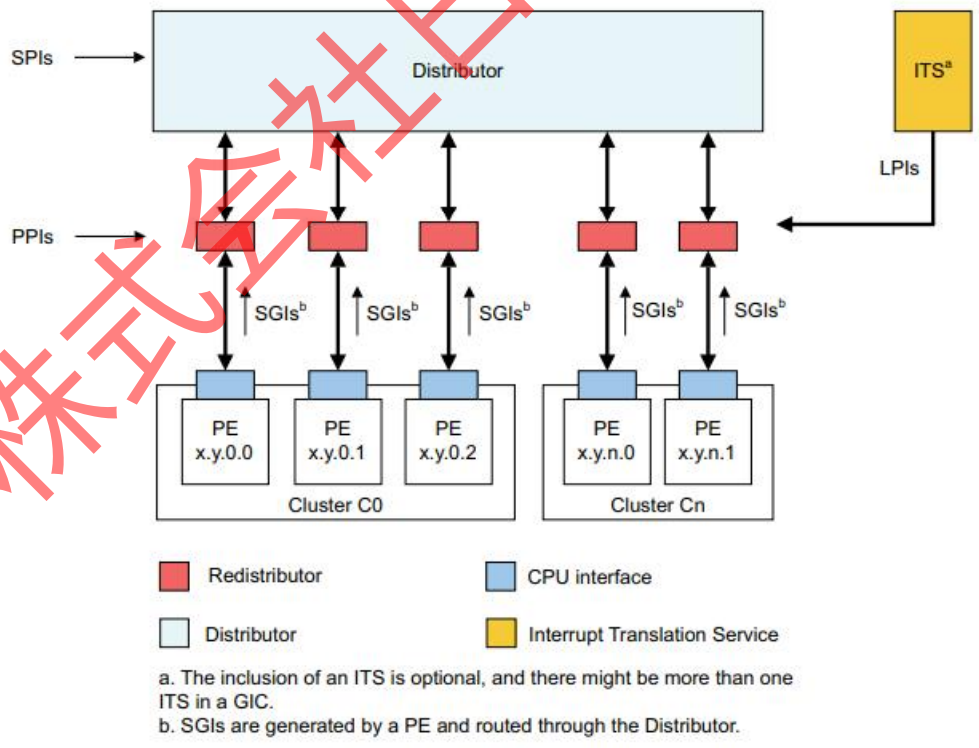
株式会社日昇テクノロジー

		具体的には SOC メーカーによって定義されます
1020-1023	•	特別な中断番号
1024 - 8191	•	予約済み
8192-MAX	LPI	• 13.2.2 GIC

13.2.2 GIC v3 の基本構造

GIC V3.0 の論理構造は以下の通りです。





a. The inclusion of an ITS is optional, and there might be more than one ITS in a GIC.
b. SGIs are generated by a PE and routed through the Distributor.

株式会社日昇テクノロジー

GIC v3 は、Distributor、CPU インターフェース、Redistributor、ITS の主要な部分から構成されています。GIC v3 では、CPU インターフェースが GIC から分離され、CPU 内に配置されています。CPU インターフェースは AXI Stream を介して GIC と通信します。GIC が割り込みを送信する際には、AXI Stream インターフェースを通じて CPU インターフェースに割り込みコマンドを送信し、CPU インターフェースは割り込みラインのマッピング設定に基づいて、IRQ または FIQ ピンを介して CPU に割り込みを送信します。

Distributor

Distributor は SPI（Shared Peripheral Interrupts）の管理に用いられ、仲裁と分配の役割を持ち、割り込みを Redistributor に送信します。

Distributor はいくつかのプログラミングインターフェース（寄存器）を提供し、これらのインターフェースを通じて以下の操作を実現できます。Distributor の主な機能には以下があります：

- CPU の割り込みを全体的に有効または無効にする。
- 任意の割り込みリクエストの有効または無効を制御する。
- 割り込みの優先度制御。
- 割り込みが発生した際に、どの CPU に割り込みリクエストを送信するかを指定する。
- 割り込み属性の設定、各「外部割り込み」のトリガーモード（エッジトリガーまたはレベルトリガー）を設定する。

CPU インターフェースの概要

CPU インターフェースは、GIC に接続されたプロセッサにインターフェースを提供し、Distributor と同様にいくつかのプログラミングインターフェースを提供します。CPU インターフェースを通じて以下の機能を実現できます（いくつかを列挙し、詳細は ARM マニュアルを参照）：

- CPU インターフェースが接続された CPU に割り込みイベントをアサートするかどうかを有効または無効にする。

株式会社日昇テクノロジー

- 割り込みの確認。

- 割り込み処理完了の通知。

- プロセッサに割り込み優先度マスクを設定する。

- プロセッサのプリエンプションポリシーを設定する。

- 最高優先度のペンディング割り込みリクエストを特定する。

簡単に言えば、CPU インターフェースは CPU への割り込みリクエストの有効または無効を制御でき、CPU の割り込みが有効になっている場合、割り込み優先度マスクよりも優先度の高い割り込みリクエストのみが CPU に送信されます。任意の時点で CPU は（CPU インターフェースのレジスタから）現在アクティブな最高優先度を読み取ることができます。

Redistributor の概要

GICv3 では、Redistributor は SGI、PPI、LPI の割り込みを管理し、それらを CPU インターフェースに送信します。Redistributor の機能には以下が含まれます：

- SGI と PPI の有効化と無効化。

- SGI と PPI の優先度設定。

- 各 PPI をエッジトリガーまたはレベルトリガーに設定する。

- 各 SGI と PPI を割り込みグループに割り当てる。

- SGI と PPI のステータスを制御する。

- メモリ内のデータ構造のベースアドレスを制御し、LPI に関連する割り込み属性とペンディングステータスをサポートする。

- 電源管理支持。

ITS（Interrupt translation service）

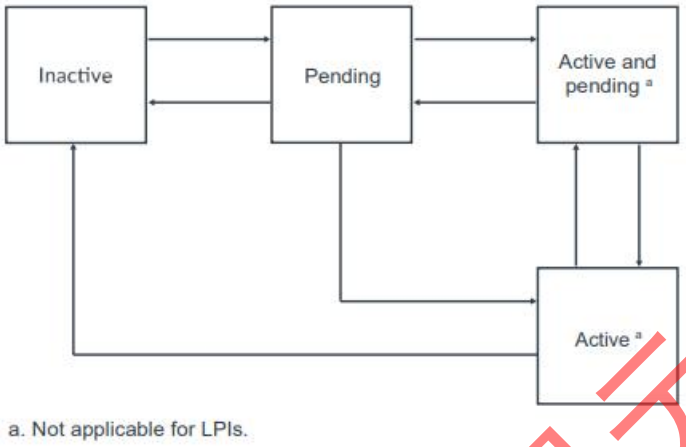
ITS は GIC v3 アーキテクチャにおける任意のハードウェアメカニズムで、メッセージベースの割り込みを LPI に変換するソフトウェアメカニズムを提供します。LPI をサポートする設定で任意にサポートさ

株式会社日昇テクノロジー

れます。ITS は LPI 割り込みを受信し、解析した後、対応するリディストリビュータに送信し、リディストリビュータは割り込み情報を CPU インターフェースに送信します。

13.2.3 割り込み状態と処理フロー

各割り込みは状態マシンを維持し、Inactive、Pending、Active、Active and pending をサポートします。割り込み処理の状態マシンは以下の通りです：



a. Not applicable for LPIs.

• Inactive：割り込み状態なし、つまり Pending でも Active でもありません。

• Pending：ハードウェアまたはソフトウェアが割り込みを引き起こし、その割り込みイベントがハードウェア信号によって GIC に通知され、GIC が割り当てた CPU が処理を待っています。レベルトリガモードでは、割り込みが発生すると同時に Pending 状態が保持されます。

• Active：CPU がその割り込み要求に応答し、処理中です。

• Active and pending：割り込み源が Active 状態にあるときに、同じ割り込み源が再び割り込みを引き起こし、Pending 状態に入り、保留状態になります。

簡単な割り込み処理プロセスは、周辺機器が割り込みを開始し、Distributor に送信され、Distributor はそれらの割り込み特性（優先度、有効化の有無など）に基づいて割り込みを処理し、適切な Redistributor に送信し、Redistributor は割り込み情報を CPU インターフェースに送信し、CPU インターフェースは適切な割り込み例外をプロセッサに生成し、プロセッサはその例外を受け取り、最後にソフトウェアがそ

株式会社日昇テクノロジー

の割り込みを処理します。

13.2.4 関連レジスタの紹介

レジスタは GIC v3 のリファレンスマニュアルの説明に従いますが、いくつかのレジスタは大まかに同じです。

重要：このセクションで説明されているレジスタは armv8 arch64 アーキテクチャ用で、レジスタ値には若干の差異があります。参考までに、実際のリファレンスマニュアルに準拠してください。

GIC v3 レジスタの分布図：

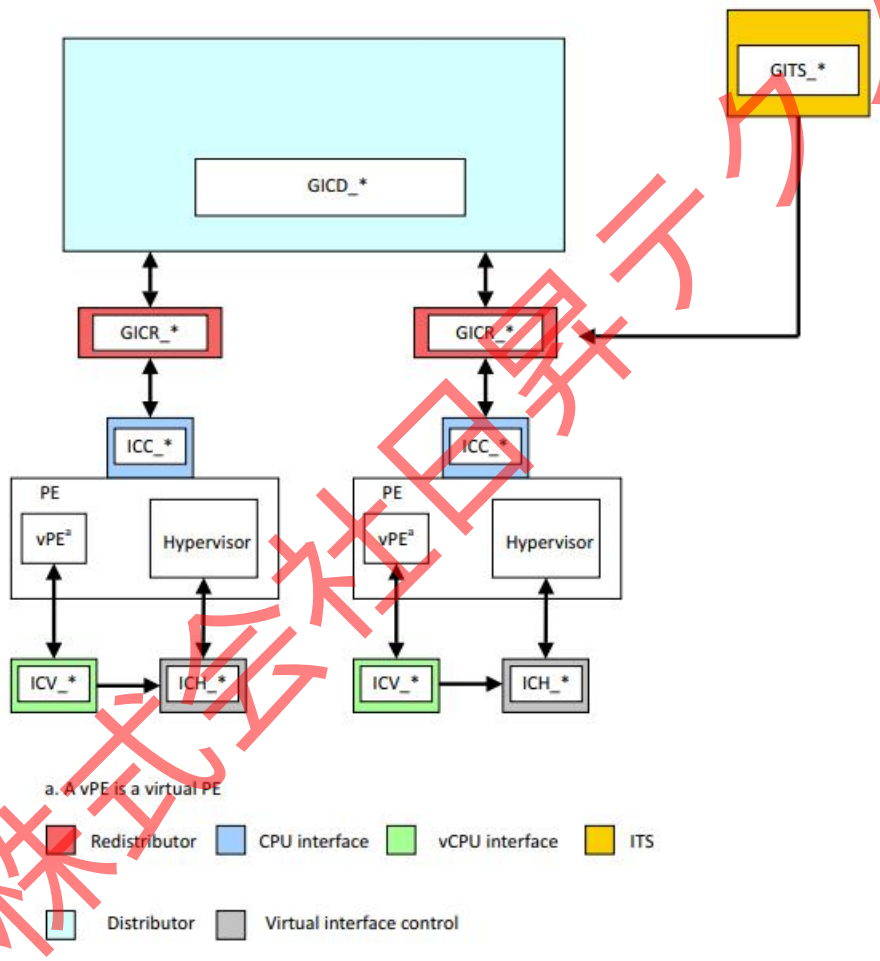


Figure 9-1 Register interfaces without legacy support (GICv3 only)

上の図のレジスタで、GICC が CPU インターフェースレジスタを示し、GICD は Distributor レジスタ、GITS は ITS レジスタ、GICR は Redistributor レジスタを示します。CPU インターフェースレジスタには、memory-mapped アクセスとシステムレジスタアクセスの 2 種類のアクセス方法があります。

株式会社日昇テクノロジー

ICC、ICV、ICH のプレフィックスが付いているのはシステムレジスタで、GICC、GICV などのプレフィックスが付いているのは memory-mapped レジスタです。

Distributor 関連レジスタ

前のセクションで述べたように、GIC Distributor はいくつかのプログラミングインターフェースを提供しており、「プログラミングインターフェース」はレジスタと考えることができます。ここではこれらのレジスタを簡単に紹介し、いくつかのレジスタを列挙します。詳細は ARM 割り込みマニュアルを参照してください。

GIC Distributor 表は以下の通りです（一部のみ示します）：

オフセットアドレス	レジスタ名	タイプ	デフォルト値	説明
0x000	GICD_CTLR	RW		ディストリビュータコントロールレジスタ
0x004	GICD_TYPER	RO		割り込みタイプコントロールレジスタ
0x008	GICD_IIDR	RO	保留	ディストリビュータバージョン情報レジスタ
0x080-0x0FC	GICD_IGROUPRn	RW	0x00000000	割り込みグループレジスタ
0x100-0x17C	GICD_ISENABLERn	RW		割り込み有効化レジスタ
0x180-0x1FC	GICD_ICENABLERn	RW	0x00000000	割り込み無効化レジスタ
0x200-0x27C	GICD_ISPENDRn	RW	0x00000000	割り込み保留設定レジスタ
0x280-0x2FC	GICD_ICPENDRn	RW	0x00000000	割り込み保留クリアレジスタ
0x300-0x37C	GICD_ISACTIVERn	RW	0x00000000	割り込みアクティブ状態設定レジスタ
0x380-0x3FC	GICD_ICACTIVERn	RW	0x00000000	割り込みアクティブ状態クリアレジスタ
0x400-0x7F8	GICD_IPRIORITYRn	RW		割り込み優先度設定レジスタ
0x800-0x81C	GICD_ITARGETSRn	RO	保留	保留中の割り込み処理対象 CPU レジスタ
0xC00-0xCFC	GICD_ICFGRn	RW	保留	保留中の割り込みタイプ（設定）レジスタ
0xE00-0xEFC	GICD_NSACRn	RW	0x00000000	非セキュアアクセス設定レジスタ
0xF00	GICD_SGIR	RO		ソフトウェア割り込み生成レジスタ

株式会社日昇テクノロジー

0xF10-0xF1C	GICD_CPENDSGIRn	RW	0x00000000	ソフトウェア割り込み保留レジスタ
0xF20-0xF2C	GICD_SPENDSGIRn	RW	0x00000000	ソフトウェア割り込み保留解除レジスタ
0xFFE8	GICD_PIDR2	RO	0x3B	ID 保存レジスタ

表では GIC Distributor の基底アドレスに対する各レジスタの相対アドレスのみが示されています。詳細な紹介は行いません。「デフォルト値」の項目に「保留中」とあるのは、元のマニュアルでは「IMPLEMENTATION DEFINED」であり、この表は《ARM® Generic Interrupt Controller》から抜粋されたもので、特定のチップに特化していないため、これらのレジスタのデフォルト値はチップメーカーによって決定されます。「アドレスオフセット」の項目の値が範囲で示されているのは、「割り込み有効化レジスタ」などのように、複数のレジスタが存在するためです。

表項目「アドレスオフセット」の部分の値は範囲であり、「割り込み有効化レジスタ」のアドレスオフセットが「0x100-0x17C」とされているのは、「割り込み有効化レジスタ」が多数存在するためで、アドレスオフセットの範囲が「0x100-0x17C」です。

一部のレジスタについて簡単に紹介します：

割り込み有効化レジスタ GICD_ISENABLERn

割り込み有効化レジスタと割り込み無効化レジスタ（GICD_ICENABLERn）は一対一で対応しており、GIC Distributor は割り込みの有効化と無効化を分けて設定します。

割り込み有効化レジスタは以下の通りです。

株式会社日昇テクノロジー

Field descriptions

The GICD_ISENABLER<n> bit assignments are:

31	0
Set_enable_bit<x>, bit [x], for x = 0 to 31	

Set_enable_bit<x>, bit [x], for x = 0 to 31

For SPIs and PPIs, controls the forwarding of interrupt number 32n + x to the CPU interfaces. Reads and writes have the following behavior:

`0b0` If read, indicates that forwarding of the corresponding interrupt is disabled. If written, has no effect.

`0b1` If read, indicates that forwarding of the corresponding interrupt is enabled.

《ARM® Generic Interrupt Controller》から分かるように、合計 1020 個（0～1019）の割り込み番号、つまり 1020 個の割り込みがあり、明らかに各割り込みを個別に制御するには、割り込み有効化レジスタ（GICD_ISENABLER）が 1 つ以上存在します。上述の表から、割り込み有効化レジスタのオフセットアドレスは 0x100-0x17C で、割り込み有効化レジスタは GICD_ISENABLER0 から GICD_ISENABLERn までこのアドレス空間に順に配置されます。

プログラムでは、割り込み番号で異なる割り込みを区別しますが、割り込み番号 m が既知の場合、どのようにして割り込み m を有効化または無効化するか？計算プロセスは以下の通りです（整数）：

(1) 設定する割り込み有効化レジスタ（n とする）を計算します。n = m / 32。例えば、割り込み番号 m = 34 の場合、割り込み有効化レジスタは GICD_ISENABLER1 です。レジスタのオフセットアドレスは（0x100 + (4*n)）です。

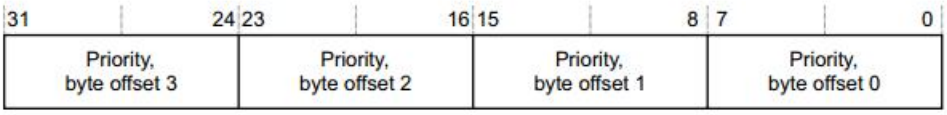
(2) 設定するビットを計算します。続けて、設定ビットを q とします。q = m % 32。m = 34 の場合、割り込み番号 34 の割り込みを有効にするには、GICD_ISENABLER1[2]を設定する必要があります。

割り込み有効化レジスタはビット単位での読み書きがサポートされており、読み出されたデータは端末の現在の状態です。0 ならば割り込みは無効、1 ならば割り込みは有効です。割り込み有効化レジスタに 1 を書き込むと割り込みが有効になり、0 を書き込んでも効果はありません。

株式会社日昇テクノロジー

割り込み優先度設定レジスタ GICD_IPRIORITYRn

割り込み有効化レジスタと同様に、GICD_IPRIORITYRn も一連のレジスタで、上述の表から、これらのレジスタは 0x400-0x7F8 のオフセットアドレスに位置しています。割り込み優先度設定レジスタは以下の通りです。



上図から、各割り込み番号は 8 ビットを占め、数値が小さいほど割り込み優先度が高いことが分かります。以下に、割り込み番号に基づいて対応する割り込み優先度設定レジスタを見つける方法を紹介します。割り込み番号が m、割り込み優先度設定レジスタが n、割り込み優先度設定ビットオフセットが offset の場合、n = m / 4。レジスタのオフセットアドレスは（0x400 + (4*n)）。レジスタ内のオフセットは offset= m%4 です。例として m = 65 の場合、n = 65/4 = 16 なので、割り込み優先度設定レジスタは GICD_IPRIORITYR16、offset(n) = 65%4 = 1、割り込み番号 65 に対応するレジスタは GICD_IPRIORITYR16[15:8]です。

13.2.4.1 CPU インターフェースレジスタの紹介

GIC v3 の CPU インターフェースモジュールもいくつかのプログラミングインターフェース、つまりいくつかのレジスタを提供しており、GIC V3 では新しいシステムレジスタ方式が提供されています。ここでは、いくつかの一般的なレジスタのみを紹介します。CPU インターフェース memory-mapped レジスタのリストは以下の通りです。

アドレスオフセット	レジスタ名	タイプ	リセット値	レジスタ説明
0x0000	GICC_CTLR	RW	0x00000000	CPU インタフェース制御レジスタ
0x0004	GICC_PMR	RW	0x00000000	割り込み優先度マスクレジスタ
0x0008	GICC_BPR	RW	0x0000000	割り込み優先度グループレジスタ
0x000C	GICC_IAR	RO	0x000003FF	割り込み確認レジスタ
0x0010	GICC_EOIR	WO	•	割り込み終了レジスタ

株式会社日昇テクノロジー

0x0014	GICC_RPR	RO	0x000000FF	実行優先度レジスタ
0x0018	GICC_HPPIR	RO	0x000003FF	最高優先度保留中断レジスタ
0x001C	GICC_ABPR	RW	0x0000000	GICC_BPR の別名レジスタ
0x0020	GICC_AIAR	RO	0x000003FF	GICC_IAR の別名レジスタ
0x0024	GICC_AEOIR	WO	•	GICC_EOIR の別名レジスタ
0x0028	GICC_AHPPIR	RO	0x000003FF	GICC_HPPIR の別名レジスタ
0x00D0-0x00DC	GICC_APRn	RW	0x00000000	アクティブな優先度レジスタ
0x00E0-0x00EC	GICC_NSAPRnc	RW	0x00000000	非セキュアなアクティブな優先度レジスタ
0x00FC	GICC_IIDR	RO	未定	CPU インタフェース識別レジスタ
0x1000	GICC_DIR	WO	•	割り込み無効レジスタ

上表の一般的な CPU インターフェースレジスタの紹介は以下の通りです：

割り込み優先マスクレジスタ GICC_PMR

前節では GIC ディストリビュータの割り込み優先度設定レジスタ GICD_IPRIORITYRn について説明しましたが、各割り込みは 8 ビットを占めます。ここでの割り込み優先マスクレジスタ GICC_PMR は、8 ビットで一つの割り込み閾値を表します。この優先度より高い割り込みのみが CPU に送られます。GICC_PMR レジスタは以下の通りです。

上図から GICC_PMR レジスタの後ろの 8 ビット（0～7）が優先度の設定に使用され、そのフォーマットは GICD_IPRIORITYR レジスタと同じです。設定が有効になると、この優先度以上の割り込みが CPU に送られます。注意が必要なのは、8 ビットレジスタのうち上位の 4 ビットのみが有効であることです。STM32 と同様に、これら 4 ビットはさらに「プリエンプション優先度」と「サブ優先度」に分けられます。優先度グループについて詳しく説明します。

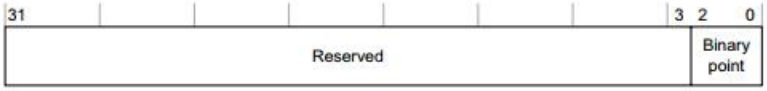
割り込み優先度グループレジスタ GICC_BPR

割り込み優先度グループレジスタは、8 ビットの優先度を 2 つの部分に分け、一部はプリエンプション

株式会社日昇テクノロジー

優先度、もう一部はサブ優先度を表し、これは STM32 の割り込み優先度グループと同じです。

GICC_BPR レジスタは以下の通りです。



割り込み優先度グループレジスタの後ろの 3 ビットは割り込み優先度グループの設定に使用され、以下の表の通りです。

割り込み優先度グループ表

GICC_BPR [2:0]	割り込み優先度値 PRI_N[7:4]	レベル			
	二進数点	プリエンプションレベルビット	サブプリオリティビット	メインプリオリティサ	ブプリオリティ
0b 001	0b xxxx	[7:4]	なし	16	0
0b 010	0b xxxx	[7:4]	なし	16	0
0b 011	0b xxxx	[7:4]	なし	16	0
0b 100	0b xxx.y	[7:5]	[4]	8	2
0b 101	0b xx.yy	[7:6]	[5:4]	4	4
0b 110	0b x.yyy	[7]	[6:4]	2	8
0b 111	0b .yyyy	None	[7:4]	None	16

各割り込みには 8 ビットの割り込み優先度設定ビットがありますが、上位の 4 ビットのみが有効です。したがって、上表の GICC_BPR [2:0]が 1 から 3 に設定されている場合は同じであり、つまり 16 レベルのプリエンプション優先度のみでサブ優先度はありません。

割り込み確認レジスタ GICC_IAR

割り込み確認レジスタ GICC_IAR は、現在保留中の最高優先度の割り込みを保存しており、レジスタの説明は以下の通りです。

株式会社日昇テクノロジー

GICC_IAR レジスタには 2 つのフィールドがあり、CPUID[10:12]は要求割り込みの CPU ID を保存します。複数コアの CPU において、割り込み処理時にこのビットに保存された情報を使用します。interrupt ID[0:9]は、現在保留中の最高優先度の割り込みを記録します。このレジスタを読み取ると、結果が 1023 であれば、現在利用可能な割り込みがないことを意味します。一般的ないくつかの状況は以下の通りです：

(1) GIC ディストリビュータで CPU への割り込み要求の送信が禁止されている。

(2) GIC の CPU インターフェースで CPU への割り込み要求の送信が禁止されている。

(3) CPU インターフェースに保留中の割り込みがない、または保留中の割り込みの優先度が GICC_PMR レジスタに設定された優先度以下である。

以下は CPU インターフェースのシステムレジスタです：

Table 12-21 Encodings for the AArch64 System registers

Register	Width (bits)	Access instruction encoding					Notes
		Op0	Op1	CRn	CRm	Op2	
ICC_PMR_EL1	32	3	0	4	6	0	RW
ICC_IAR0_EL1	32			12	8	0	RO
ICC_EOIR0_EL1	32					1	WO
ICC_HPPIR0_EL1	32					2	RO
ICC_BPR0_EL1	32					3	RW
ICC_AP0R<n>_EL1	32					4-7	RW, <n> = Op2-4
ICC_AP1R<n>_EL1	32				9	0-3	RW, <n> = Op2
ICC_DIR_EL1	32				11	1	WO
ICC_RPR_EL1	32					3	RO
ICC_SGI1R_EL1	64					5	WO
ICC_ASGI1R_EL1	64					6	WO
ICC_SGI0R_EL1	64					7	WO

CPU インターフェースは GIC 内部から分離され、PE（Processing Element）の内部で実装され、システムレジスタを介して割り込みの迅速な応答が可能になります。システムレジスタと memory-mapped レジスタは類似しています。さらに、上に挙げたレジスタの接尾辞にはすべて EL1 があり、これは GIC システムレジスタインターフェイスの例外レベルを示しています。armv8 では、例外レベルが EL0、EL1、

株式会社日昇テクノロジー

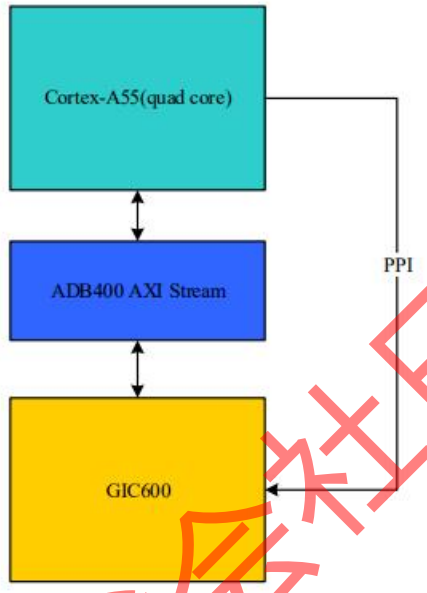
EL2、EL3 に分けられ、数字が大きいほどレベルが高くなります。

割り込み優先マスクレジスタ ICC_PMR_EL1

割り込み優先レベル機能を提供し、この値より高い優先レベルの割り込みのみが PE に通知されます。GICC_PMR と同様に、このレジスタの後ろの 8 ビット（0～7）が優先レベルの設定に使用され、上位 4 ビットのみが有効です。

13.2.5 GIC-600 の簡単な紹介

RK3588 の割り込みコントローラは GIC-600 で、GIC v3 バージョンをサポートする arm の実際のコントローラ設計参照です。RK3588 の GIC フローチャートは以下の通りです：



RK3588 の 256 個の SPI 割り込み番号の簡単な割り当て表（一部）：

株式会社日昇テクノロジー

Table 1-3 RK3588 Interrupt Connection List

Number	Source	Polarity	Number	Source	Polarity
0-21 PPI	NA	Low level	238	irq_fspi	High level

Copyright 2022 © Rockchip Electronics Co., Ltd.

Number	Source	Polarity	Number	Source	Polarity
22 PPI	cpu_ncommirq	Low level	239	irq_keylad	High level
23 PPI	cpu_npmuirq	Low level	240	irq_crypto_s	High level
24 PPI	cpu_ctiirq	Low level	241	irq_crypto_ns	High level
25 PPI	cpu_nvcpumntirq	Low level	242	irq_otp_s	High level
26 PPI	cpu_ncnthpirq	Low level	243	irq_otp_ns	High level
27 PPI	cpu_ncntvirq	Low level	244	irq_trng_chk	High level
28 PPI	cpu_ncnthvirq	Low level	245	irq_dcf	High level
29 PPI	cpu_ncntpsirq	Low level	246	irq_usb2host0_arb	High level
30 PPI	cpu_ncntpnsirq	Low level	247	irq_usb2host0_ehci	High level
31 PPI	NA	Low level	248	irq_usb2host0_ohci	High level
32	irq_dsu_nfaultirq0	High level	249	irq_usb2host1_arb	High level
33	irq_dsu_nfaultirq1	High level	250	irq_usb2host1_ehci	High level
34	irq_dsu_nfaultirq2	High level	251	irq_usb2host1_ohci	High level
35	irq_dsu_nfaultirq3	High level	252	irq_usb3otg0	High level

RK3588 のデバイストリーを組み合わせると、GIC-600 レジスタのアドレスを確認できます：

レジスタアドレス範囲

レジスタ	アドレス
Distributor	0xfd400000～0xfd410000
ITS	0xfd440000～0xfd460000
Redistributor	0xfd460000～0xfd520000

13.3 割り込みドライバの簡単な分析

13.3.1 デバイストリーの割り込み情報

前述のように、GIC コントローラの背景知識と一部のアセンブリコードを用いた割り込み処理の説明を通じて、割り込みの使用プロセスを理解しました。それらの低レベルの詳細は Linux システムではほとんどが既にためにカプセル化されており、このフレームワークの下で使用するだけです。以下では RK3588 の割り込みを例に挙げて、割り込みコントロールの初期化について簡単に説明します。カーネルソースコードの取得は環境構築の章を参照してください。

株式会社日昇テクノロジー

まず、デバイストリーがどのようにして全体の割り込みシステム情報を記述しているかを見てみましょう。カーネルソースコードの/arch/arm64/boot/dts/rockchip ディレクトリにある RK3588.dtsi デバイストリーファイルを開き、「interrupt-controller」ノードを見つけます。以下のようになります。

リスト 1：割り込み interrupt-controller ノード

```
1 gic: interrupt-controller@fd400000 {
2 compatible = "arm,gic-v3";
3 #interrupt-cells = <3>;
4 #address-cells = <2>;
5 #size-cells = <2>;
6 ranges;
7 interrupt-controller;
8
9 reg = <0x0 0xfd400000 0 0x10000>, /* GICD */
10 <0x0 0xfd460000 0 0xc0000>; /* GICR */
11 interrupts = <GIC_PPI 9 IRQ_TYPE_LEVEL_HIGH>;
12 its: interrupt-controller@fd440000 {
13 compatible = "arm,gic-v3-its";
14 msi-controller;
15 #msi-cells = <1>;
16 reg = <0x0 0xfd440000 0x0 0x20000>; /*ITS の物理アドレス*/
17 };
18 };
```

- compatible：compatible 属性はプラットフォームデバイスドライバのマッチングに使用されます。

株式会社日昇テクノロジー

- reg：reg は割り込みコントローラ関連レジスタのアドレスとサイズを指定します。GICD は Distributor レジスタ、GICR は Redistributor レジスタを指します。

- interrupt-controller：このデバイストリーノードが割り込みコントローラであることを宣言します。

- #interrupt-cells：この割り込みコントローラを使用するノードが 1 つの割り込みを記述するために何個の cells を使用するかを指定します。これは、1 つの割り込み情報を記述するために何個のパラメータを使用するかを理解するのに役立ちます。この場合、intc ノードの子ノードは割り込みを記述するために 3 個のパラメータを使用します。

- interrupts：割り込み情報を記述します。ここでは 3 つの u32 で記述され、これは前述の #interrupt-cells によって指定されたものです。最初の u32 は割り込みタイプを、2 番目は割り込み番号を、3 番目はトリガタイプを指定します。

- its: GIC デバイスノードの下には、子デバイスノード its があります。ITS デバイスはメッセージシグナル割り込み（MSI）を CPU にルーティングするために使用されます。

- msi-controller: このデバイスが MSI コントローラであることを示します。

- #msi-cells: 1 でなければなりません。MSI デバイスの DeviceID です。

前に学んだ内容を理解している方なら、GIC 割り込みコントローラについては既に馴染みがあるでしょう。GIC アーキテクチャは Distributor、Redistributor、CPU Interface に分かれており、上述のデバイストリーノードは全 GIC コントローラを記述するために使用されます。

株式会社日昇テクノロジー

GIC 割り込みコントローラの使用例として、uart3 を例に取ります（RK3588.dtsi）：

```
1 / {
2 compatible = "rockchip,rk3568";
3
4 interrupt-parent = <&gic>;
5 #address-cells = <2>;
6 #size-cells = <2>;
7
8 /*.....*/
9 uart3: serial@fe670000 {
10 compatible = "rockchip,rk3568-uart", "snps,dw-apb-uart";
11 reg = <0x0 0xfe670000 0x0 0x100>;
12 interrupts = <GIC_SPI 119 IRQ_TYPE_LEVEL_HIGH>;
13 clocks = <&cru SCLK_UART3>, <&cru PCLK_UART3>;
14 clock-names = "baudclk", "apb_pclk";
15 reg-shift = <2>;
16 reg-io-width = <4>;
17 dmas = <&dmac0 6>, <&dmac0 7>;
18 pinctrl-names = "default";
19 pinctrl-0 = <&uart3m0_xfer>;
20 status = "disabled";
21 };
```

株式会社日昇テクノロジー

```
22 /*.....*/

23 };
```

uart3 はルートノードの下の子ノードで、ルートノードは interrupt-parent として gic を指定しています。したがって、uart3 子ノードも GIC 割り込みコントローラを継承し、使用するリソースは interrupts で記述されます。

• interrupts: 具体的な割り込み記述情報で、このノードが使用する割り込みコントローラは gic で、gic ノード内の「#interrupt-cells = <3>」が子コントローラの情報を記述するために 3 つの cells を使用することを規定しています。3 つのパラメータの意味は以下の通りです：

最初のパラメータは割り込みタイプを指定し、GIC の割り込みタイプには 3 種類（SPI 共有割り込み、PPI プライベート割り込み、SGI ソフトウェア割り込み）があります。使用する外部割り込みはすべて SPI 割り込みタイプに属します。

2 番目のパラメータは割り込み番号を設定し、範囲は最初のパラメータに依存します。PPI 割り込みの範囲は[0-15]、SPI 割り込みの範囲は[0-256]です。

3 番目のパラメータは割り込みトリガ方式を指定し、パラメータは u32 型で、後ろの 4 ビット[0-3]が割り込みトリガタイプの設定に使用されます。各ビットはトリガ方式を表し、組み合わせが可能です。システムは対応するマクロを提供しており、以下のように直接使用できます：

リスト 2: 割り込みトリガ方式設定（irq.h）

```
1 #define IRQ_TYPE_NONE 0

2 #define IRQ_TYPE_EDGE_RISING 1

3 #define IRQ_TYPE_EDGE_FALLING 2

4 #define IRQ_TYPE_EDGE_BOTH (IRQ_TYPE_EDGE_FALLING | IRQ_TYPE_EDGE_RISING)

5 #define IRQ_TYPE_LEVEL_HIGH 4

6 #define IRQ_TYPE_LEVEL_LOW 8
```

株式会社日昇テクノロジー

3 番目のパラメータの[8-15]ビットは、PPI 割り込みにおいて「CPU マスク」の設定にも使用されます。マルチコアシステムでは、この 8 ビットは PPI 割り込みをどの CPU に送信するかを設定するために使用され、1 ビットが 1 つの CPU を表し、1 ならば PPI 割り込みを CPU0 に送信し、そうでなければマスクします。

以下の例：

```
1 timer {
2 compatible = "arm,armv8-timer";
3 interrupts = <GIC_PPI 13 (GIC_CPU_MASK_SIMPLE(4) | IRQ_TYPE_LEVEL_HIGH)>,
4 <GIC_PPI 14 (GIC_CPU_MASK_SIMPLE(4) | IRQ_TYPE_LEVEL_HIGH)>,
5 <GIC_PPI 11 (GIC_CPU_MASK_SIMPLE(4) | IRQ_TYPE_LEVEL_HIGH)>,
6 <GIC_PPI 10 (GIC_CPU_MASK_SIMPLE(4) | IRQ_TYPE_LEVEL_HIGH)>;
7 arm,no-tick-in-suspend;
8 };
```

13.3.2 GIC v3 割り込みコントローラのコード

割り込みコントローラは IRQCHIP_DECLARE マクロを通じて__irqchip_of_table に登録されます。

リスト 3: カーネルソースコード/drivers/irqchip/irq-gic-v3.c

```
1 IRQCHIP_DECLARE(gic_v3, "arm,gic-v3", gicv3_of_init); // __irqchip_of_table に静的な const struct of_device_id を初期化し配置する
```

システムの起動初期化段階で、of_irq_init 関数はデバイスノード情報を検索し、「arm,gic-v3」に基づいて__irqchip_of_table セグメント内の「gic」デバイストリーノードをマッチさせ、デバイス情報を取得します。最終的に IRQCHIP_DECLARE 宣言のコールバック関数 gicv3_of_init が実行されます。

カーネルソースコード include/linux/irqchip.h ファイル内で IRQCHIP_DECLARE マクロが定義されて

株式会社日昇テクノロジー

います：

リスト 4: カーネルソースコード include/linux/irqchip.h

```
1 #define IRQCHIP_DECLARE(name, compat, fn) OF_DECLARE_2(irqchip, name, compat, fn)

2

3 #define OF_DECLARE_2(table, name, compat, fn) ¥

4 _OF_DECLARE(table, name, compat, fn, of_init_fn_2)

5

6 #define _OF_DECLARE(table, name, compat, fn, fn_type) ¥

7 static const struct of_device_id __of_table_##name ¥

8 __used __section(__##table##_of_table) ¥

9 __aligned(__alignof__(struct of_device_id)) ¥

10 = { .compatible = compat, ¥

11 .data = (fn == (fn_type)NULL) ? fn : fn }
```

このマクロは struct of_device_id の静的定数を初期化し、__irqchip_of_table セグメントに配置します。

株式会社日昇テクノロジー

リスト 5: カーネルソースコード/drivers/irqchip/irq-gic-v3.c

```
1 static int __init gicv3_of_init(struct device_node *node, struct device_node *parent)
2 {
3     void __iomem *dist_base;
4     struct redist_region *rdist_regs;
5     u64 redist_stride;
6     u32 nr_redist_regions;
7     int err, i;
8
9     dist_base = of_iomap(node, 0); // GICD のレジスタアドレス空間をマッピング
10    if (!dist_base) {
11        pr_err("%pOF: unable to map gic dist registers¥n", node);
12        return -ENXIO;
13    }
14
15    err = gic_validate_dist_version(dist_base); // GIC ハードウェアのバージョンが GICv3 または
 GICv4 であることを検証
16    if (err) {
17        pr_err("%pOF: no distributor detected, giving up¥n", node);
18        goto out_unmap_dist;
19    }
20
21    /* デバイスツリーノードから"redistributor-regions"の値を読み取り、存在しない場合は 1 とする */
```

株式会社日昇テクノロジー

```
22    if (of_property_read_u32(node, "#redistributor-regions", &nr_redist_regions))
23        nr_redist_regions = 1;
24
25    rdist_regs = kcalloc(nr_redist_regions, sizeof(*rdist_regs), GFP_KERNEL);
26    if (!rdist_regs) {
27        err = -ENOMEM;
28        goto out_unmap_dist;
29    }
30
31    for (i = 0; i < nr_redist_regions; i++) { // GICR ドメインのベースアドレスを割り当てる
32        struct resource res;
33       int ret;
34
35        ret = of_address_to_resource(node, 1 + i, &res);
36        rdist_regs[i].redist_base = of_iomap(node, 1 + i);
37        if (ret || !rdist_regs[i].redist_base) {
38            pr_err("%pOF: couldn't map region %d¥n", node, i);
39            err = -ENODEV;
40            goto out_unmap_rdist;
41        }
42        rdist_regs[i].phys_base = res.start;
43    }
44
```

株式会社日昇テクノロジー

```
45    /* DTS から"redistributor-regions"の値を読み取り、存在しない場合は 0 とする */
46    if (of_property_read_u64(node, "redistributor-stride", &redist_stride))
47        redist_stride = 0;
48
49    /* GIC v3 の初期化の核心的な作業 */
50    err = gic_init_bases(dist_base, rdist_regs, nr_redist_regions, redist_stride, &node->fwnode);
51    if (err)
52        goto out_unmap_rdist;
53
54    gic_populate_ppi_partitions(node); // PPI のアフィニティを設定
55
56    if (static_branch_likely(&supports_deactivate_key))
57        gic_of_setup_kvm_info(node);
58
59    return 0;
60
61 out_unmap_rdist:
62    for (i = 0; i < nr_redist_regions; i++)
63        if (rdist_regs[i].redist_base)
64            iounmap(rdist_regs[i].redist_base);
65    kfree(rdist_regs);
66
```

株式会社日昇テクノロジー

```
67 out_unmap_dist:
68     iounmap(dist_base);
69     return err;
70 }
```

gicv3_of_init 関数は上記の通りです。この関数は GIC レジスタのベースアドレスをマッピングし、GIC のバージョン情報を取得し（GICD_PIDR2 レジスタを読み、レジスタのビット[7:4]が 0x3 であれば GIC v3 と判断）、デバイスツリーの属性値を取得し、gic_init_bases 関数を呼び出します。

リスト 6: カーネルソースコード/drivers/irqchip/irq-gic-v3.c

```
1 static int __init gic_init_bases(void __iomem *dist_base, struct redist_region *rdist_regs, u32
nr_redist_regions, u64 redist_stride, struct fwnode_handle *handle)
2 {
3     u32 typer;
4     int gic_irqs;
5     int err;
6
7     if (!is_hyp_mode_available())


8         static_branch_disable
9
10 (&supports_deactivate_key);
11
12     if (static_branch_likely(&supports_deactivate_key))
```

株式会社日昇テクノロジー

```
13      pr_info("GIC: Using split EOI/Deactivate mode¥n");
14
15  /* GIC v3 ハードウェアデバイス関連のデータ構造を初期化 */
16   gic_data.fwnode = handle; // 一部のコールバックメソッド
17   gic_data.dist_base = dist_base; // Distributor メモリ領域のアドレス
18   gic_data.redist_regions = rdist_regs; // gic の Redistributor ドメイン情報
19   gic_data.nr_redist_regions = nr_redist_regions; // Redistributor ドメインの数
20   gic_data.redist_stride = redist_stride; // Redistributor ドメイン間の間隔
21
22  /*
23   * サポートされている割り込みの数を調べる。
24   * GIC は最大で 1020 の割り込みソース（SGI+PPI+SPI）のみをサポートしている
25   */
26   typer = readl_relaxed(gic_data.dist_base + GICD_TYPER);
27   gic_data.rdists.gicd_typer = typer;28   gic_irqs = GICD_TYPER_IRQS(typer); // bit[4:0]の値を取得
 し、SPI の数を計算
28   if (gic_irqs > 1020)
29      gic_irqs = 1020;
30   gic_data.irq_nr = gic_irqs;
31
32  /* システムに GIC V3 の irq domain データ構造を登録 */
33   gic_data.domain = irq_domain_create_tree(handle, &gic_irq_domain_ops, &gic_data);
```

株式会社日昇テクノロジー

```
34    irq_domain_update_bus_token(gic_data.domain, DOMAIN_BUS_WIRED);
35    gic_data.rdists.rdist = alloc_percpu(typeof(*gic_data.rdists.rdist));
36    gic_data.rdists.has_vlpis = true;
37    gic_data.rdists.has_direct_lpi = true;
38
39    if (WARN_ON(!gic_data.domain) || WARN_ON(!gic_data.rdists.rdist)) {
40        err = -ENOMEM;
41        goto out_free;
42    }
43
44    gic_data.has_rss = !!(typer & GICD_TYPER_RSS);
45    pr_info("Distributor has %sRange Selector support¥n", gic_data.has_rss ? "" : "no ");
46
47    if (typer & GICD_TYPER_MBIS) {
48        err = mbi_init(handle, gic_data.domain);
49        if (err)
50            pr_err("Failed to initialize MBIs¥n");
51    }
52    set_handle_irq(gic_handle_irq); // 割り込みコールバック関数を設定し、どの割り込みであるかを調べ
て処理する
53    gic_update_vlpi_properties(); /* Redistributor 関連の属性を更新 */
54    if (IS_ENABLED(CONFIG_ARM_GIC_V3_ITS) && gic_dist_supports_lpis())
```

株式会社日昇テクノロジー

```
55        its_init(handle, &gic_data.rdists, gic_data.domain); // ITS を初期化
56
57    gic_smp_init(); // コア間通信などを設定
58    gic_dist_init(); // Distributor を初期化
59    gic_cpu_init();
60    gic_cpu_pm_init(); // GIC の電源管理を初期化
61
62    return 0;
63
64 out_free:
65   if (gic_data.domain)
66        irq_domain_remove(gic_data.domain);
67    free_percpu(gic_data.rdists.rdist);
68    return err;
69 }
```

gic_init_bases 関数は上記の通りで、上のコードコメントでいくつかの解説を見ることができます。ソースコード分析はここまでで、主に割り込みコントローラドライバに関連するソースコードを分析しました。詳細な IRQ Domain マッピング、irq イベントの詳細な処理プロセスは、カーネルソースコードを参照してください。

13.4 割り込み API と重要なデータ構造

次章でドライバを書く前に、カーネルが提供する割り込みの一般的なインターフェイス関数を理解する必要があります。

株式会社日昇テクノロジー

13.4.1 request_irq 割り込みの申請と解放関数

リスト 7: 割り込み申請関数

```
1 static inline int __must_check request_irq(unsigned int irq, irq_handler_t handler,
2 unsigned long flags, const char *name, void *dev);
3
4 int devm_request_irq(struct device *dev, unsigned int irq, irq_handler_t handler,
5 unsigned long irqflags, const char *devname, void *dev_id);
```

request_irq()関数

関数のパラメータ：

- irq：「カーネル割り込み番号」を指定するために使用され、このパラメータはデバイスツリーから取得または変換されます。カーネル空間では、それは一意の割り込み番号を代表します。

- handler：割り込み処理関数を指定するために使用され、割り込みが発生した後、この関数にジャンプして実行されます。

- flags：割り込みトリガ条件であり、上昇エッジトリガ、下降エッジトリガなどのトリガ方式を「|」で組み合わせて指定します（この設定はデバイスツリーのデフォルト設定を上書きします）、マクロ定義は以下の通りです：

株式会社日昇テクノロジー

```
1 #define IRQF_TRIGGER_NONE 0x00000000
2 #define IRQF_TRIGGER_RISING 0x00000001
3 #define IRQF_TRIGGER_FALLING 0x00000002
4 #define IRQF_TRIGGER_HIGH 0x00000004
5 #define IRQF_TRIGGER_LOW 0x00000008
6 #define IRQF_TRIGGER_MASK (IRQF_TRIGGER_HIGH | IRQF_TRIGGER_LOW | ¥
7 IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING)
8 #define IRQF_TRIGGER_PROBE 0x00000010
9
10 #define IRQF_SHARED 0x00000080
11 /*----------以下のマクロ定義は省略----------*/
```

- name：割り込みの名前で、割り込みが正常に申請された後、「/proc/interrupts」ディレクトリで対応するファイルを見ることができます。

- dev：**IRQF_SHARED**マクロを使用した場合、共有割り込みを開始します。「共有割り込み」とは、複数のドライバプログラムが同じ割り込みを共有することを意味します。共有割り込みを開始した後、割り込みが発生すると、カーネルはこれらのドライバの「割り込みサービス関数」を順番に呼び出します。これにより、割り込みサービス関数内で割り込みが本ドライバから発生したかどうかを判断する必要があります。ここで dev パラメータを使用して割り込みを判断できます。ドライバを解除する際にも、カーネルは dev パラメータに基づいてどの割り込みサービス関数を削除するかを決定するため、dev パラメータを追加する必要があります。

戻り値：

- 成功：0 を返します

- 失敗：負の数を返します。

株式会社日昇テクノロジー

devm_request_irq()関数

この関数は request_irq()との違いは、devm_で始まる API がカーネルの「managed」リソースを申請することであり、一般的にエラーハンドリングや remove()インターフェースで明示的にリリースする必要はありません。

リスト 8: 割り込み解放関数

```
1 void free_irq(unsigned int irq, void *dev);
```

free_irq()関数：

- irq：デバイスツリーから取得または変換された割り込み番号。
- dev：request_irq 関数で渡された dev パラメータと一致します。

13.4.2 割り込み処理関数

割り込みを申請する際には、割り込み処理関数を指定する必要があります。書式は以下の通りです。

リスト 9: 割り込みサービス関数の形式

```
1 irqreturn_t (*irq_handler_t)(int irq, void *dev);
```

パラメータ：

- irq：「カーネル割り込み番号」を指定します。
- dev：共有割り込みで、割り込みが発生したドライバがどれかを判断するために使用されます。dev パラメータはカーネルから「持ち帰り」ます。共有割り込みを使用している場合は、dev が持ち帰るハードウェア情報に基づいて割り込みが本ドライバから来たかどうかを判断する必要があります。そうでなければ、割り込みサービス関数から直ちに抜け出すべきです。そうでない場合は、通常通り割り込みサービス関数を実行します。

戻り値：

- irqreturn_t 型：列挙型変数で、以下の通りです。

株式会社日昇テクノロジー

リスト 10: 割り込みサービス関数の戻り値タイプ

```
1 enum irqreturn {

2 IRQ_NONE = (0 << 0),

3 IRQ_HANDLED = (1 << 0),

4 IRQ_WAKE_THREAD = (1 << 1),

5 };

6

7 typedef enum irqreturn irqreturn_t;
```

「共有割り込み」を使用しており、割り込みサービス関数で割り込みが本ドライバからではないことがわかった場合は IRQ_NONE を返すべきです。共有割り込みを使用していない場合、または使用していて割り込みが本ドライバから来た場合は IRQ_HANDLED を返し、割り込み要求が正常に処理されたことを示します。第三のパラメータは、後ほど説明する割り込みサービス関数の「上半部」および「下半部」に関連します。割り込みサービス関数が「上半部」と「下半部」の実装を使用している場合は、IRQ_WAKE_THREAD を返すべきです。

13.4.3 割り込みの有効化と無効化関数

関数を使用して特定の割り込みを有効または無効にします。

リスト 11: 割り込みの無効化と有効化

```
1 void enable_irq(unsigned int irq)

2 void disable_irq(unsigned int irq)
```

パラメータ：

- irq：指定された「カーネル割り込み番号」

戻り値：なし

株式会社日昇テクノロジー

リスト 12: 本 CPU 内のすべての割り込みを無効化または復元

```
1 local_irq_enable()

2 local_irq_disable()

3 local_irq_save(flags)

4 local_irq_restore(flags)
```

「グローバル割り込み」の特性により、通常、割り込みを無効にする前に local_irq_save を使用して現在の割り込み状態を保存し、割り込みを有効にした後に local_irq_restore マクロを使用して以前の状態を復元します。flags は unsigned long 型のデータです。

armv8-arch64 アーキテクチャでは、local_irq_disable()は daif フラグビットを操作して現在の CPU の例外を無効にするだけで、具体的な割り込みコントローラとは関係ありません。一方、disable_irq()は割り込みコントローラを制御して割り込みを無効にします。

上記の関数の機能を理解すれば、割り込みドライバプログラムを書くことができます。コードの紹介中に何か漏れがあれば、次章の実験で参照します。

第 14 章 割り込みサブシステムの実験

この章では、前章で説明した割り込み API を組み合わせて、キー割り込みドライバを書き、Linux の割り込みプログラミング方法を理解し、割り込みの有効化と無効化、割り込みのボトムハーフ tasklet、ワークキュー、ソフト割り込みメカニズムおよび threaded_irq などの概念を理解します。

14.1 キー割り込みプログラムの実験

14.1.1 デバイストリープラグインの実装

前述の割り込みコントローラの内容は、製造業者が提供しています。行う必要があるのは、書かれた割り込みコントローラの親ノードを参照し、割り込み情報を設定することだけです。

ここでは、デバイストリープラグインの形式で書くことにします（デバイストリーも使用可能です）。

株式会社日昇テクノロジー

リスト 1: button キー割り込みのデバイストリープラグイン

```
1 /dts-v1/;

2 /plugin/;

3

4 #include <dt-bindings/gpio/gpio.h>

5 #include <dt-bindings/pinctrl/rockchip.h>

6 #include <dt-bindings/interrupt-controller/irq.h>

7

8 &{/} {

9 button_interrupt: button_interrupt {

10 status = "okay";

11 compatible = "button_interrupt";

12 button-gpios = <& GPIO1 RK_PB0 GPIO_ACTIVE_LOW>;;

13 pinctrl-names = "default";

14 pinctrl-0 = <&button_interrupt_pin>;

15 interrupt-parent = <& GPIO1>;

16 interrupts = <RK_PB0 IRQ_TYPE_LEVEL_LOW>;

17 };

18 };

19

20 &{/pinctrl} {

21 pinctrl_button {
```

株式会社日昇テクノロジー

```
22 button_interrupt_pin: button_interrupt_pin {

23 rockchip,pins = <0 RK_PB0 RK_FUNC_GPIO &pcfg_pull_none>;

24 };

25 };

26 };
```

ここでは、割り込みに関連する部分を主に紹介します。このノードは割り込みコントローラではなく、割り込みコントローラを使用するため、「interrupt-controller」タグはありません。

- 第 4-6 行：デバイストリープラグインでいくつかのマクロ定義を使用しています。ここでは、対応するヘッダーファイルを含める必要があります。

- 第 8-9 行、新たに追加された button_interrupt ノード、

- 第 12 行、ボタンに使用される GPIO を定義し、ここでは lubuncat2 を例に、40pin の GPIO1_B0 を使用しています。実際には、異なるボードでボタンがない場合があり、他のピンに置き換えて外部ボタンを接続するか、高低電圧でボタンをシミュレートできます。

- 第 14 行、ピンのマルチプレクス情報、つまり pinctrl。

- 第 15-16 行、割り込み関連情報を追加し、interrupt-parent は親割り込みコントローラノードが GPIO1 であることを示し、interrupts は割り込みピンとトリガ方式を示します。

14.1.2 キー割り込みドライバプログラムの実装

デバイストリー（デバイストリープラグイン）を使用していますが、ドライバプログラムはシンプルなキャラクタデバイスドライバであり、デバイストリー内のノードとマッチするわけではありません。マッチするかどうかに関わらず、「デバイストリーを読む」こととは関係ありません。ドライバのソースコードは、ドライバのエントリとエグジット関数の実装、キャラクタデバイス操作関数の実装の 2 つの部分に大きく分けられます。ソースコードと組み合わせて以下に紹介します：

株式会社日昇テクノロジー

14.1.2.1 ドライバエントリとエグジット関数の実装

ドライバのエントリ関数ではキャラクタデバイスの登録を行い、エグジット関数ではキャラクタデバイスを解除します。一部のコードは以下のようになります：

リスト 2: ドライバエントリとエグジット関数の実装

```
1 /*
2 * ドライバ初期化関数
3 */
4 static int __init button_driver_init(void)
5 {
6    int error = -1;
7    /* 動的割り当て方式でデバイス番号を取得、サブデバイス番号は 0、*/
8    error = alloc_chrdev_region(&button_devno, 0, DEV_CNT, DEV_NAME);
9    if (error < 0)
10   {
11       printk("fail to alloc button_devno¥n");
12       goto alloc_err;
13   }
14   /* cdev 構造体と file_operations 構造体を関連付け*/
15   button_chr_dev.owner = THIS_MODULE;
16   cdev_init(&button_chr_dev, &button_chr_dev_fops);
17
18   /* cdev_map ハッシュテーブルにデバイスを追加*/
```

株式会社日昇テクノロジー

```
19    /*------------以下のコード省略---------------*/
20 }
21
22 /*
23  * ドライバ解除関数
24  */
25 static void __exit button_driver_exit(void)
26 {
27     pr_info("button_driver_exit¥n");
28     /* デバイスの削除*/
29     device_destroy(class_button, button_devno); //デバイスをクリア
30     class_destroy(class_button); // クラスをクリア
31    cdev_del(&button_chr_dev); // デバイス番号をクリア
32     unregister_chrdev_region(button_devno, DEV_CNT); // キャラクタデバイスの登録解除
33 }
34
35 module_init(button_driver_init);
36 module_exit(button_driver_exit);
37
38 MODULE_LICENSE("GPL");
39 MODULE_DESCRIPTION("embedfire lubuncat2_RK, interrupt ");
```

キャラクタデバイスの登録と解除は以前のキャラクタデバイスの章で説明しましたが、読みやすくするために一部のコードをここに示します。完全な内容はこのセクションの付随コードを参照してください。

株式会社日昇テクノロジー

14.1.2.2 .open 関数の実装

open 関数はボタンの初期化作業を行います。コードは以下の通りです：

リスト 3: open 関数の実装

```
1 static int button_open(struct inode *inode, struct file *filp)
2 {
3    int error = -1;
4
5    /* ボタンのデバイスツリーノードを取得*/
6    button_device_node = of_find_node_by_path("/button_interrupt");
7    if(NULL == button_device_node)
8    {
9       printk("of_find_node_by_path error!");
10       return -1;
11   }
12
13   /* ボタン使用の GPIO を取得*/
14   button_GPIO_number = of_get_named_gpio(button_device_node, "button-gpios", 0);
15   if(0 == button_GPIO_number)
16   {
17       printk("of_get_named_gpio error");
18       return -1;
19   }
```

株式会社日昇テクノロジー

```
20
21    /* GPIO  の申請、解放を忘れずに*/
22    error = gpio_request(button_GPIO_number, "button_gpio");
23    if(error < 0)
24    {
25        printk("gpio_request error");
26        gpio_free(button_GPIO_number);
27        return -1;
28    }
29
30    error = gpio_direction_input(button_GPIO_number);
31
32    /* 割り込み番号を取得*/
33    interrupt_number = irq_of_parse_and_map(button_device_node, 0);
34    printk("¥n irq_of_parse_and_map! = %d ¥n",interrupt_number);
35
36    /* 割り込みの申請、解放を忘れずに*/
37    error = request_irq(interrupt_number, button_irq_handler, IRQF_TRIGGER_RISING,
"button_interrupt", NULL);
38    if(error != 0)
39    {
40        printk("request_irq error");
```

株式会社日昇テクノロジー

```
41        free_irq(interrupt_number, NULL);
42        return -1;
43    }
44
45    return 0;
46 }
```

- 第 10 行、ボタンのデバイスツリーノードを取得します。以前説明したように、ドライバがデバイスツリーノードとマッチする方法を採用していないとしても、ノードパスが正しい限り他のデバイスツリーノードを取得することが影響されません。

- 第 18 行、使用される GPIO を取得します。詳細は「GPIO サブシステム章」を参照してください。

- 第 26 行、GPIO を登録します。

- 第 34 行、GPIO を入力モードに設定します。

- 第 37 行、関数 irq_of_parse_and_map を使用して割り込み関数を解析しマップします。関数のプロトタイプは以下の通りです：

- 第 41 行、割り込みを申請します。この関数は本章の初めにすでに紹介されていますが、ここでは共有割り込みを使用していませんが、dev パラメータをキャラクタデバイス構造体ポインタとして設定しています。もちろん、NULL や他の値を設定することも可能です。

リスト 4: 割り込み関数の解析とマッピング

```
1 unsigned int irq_of_parse_and_map(struct device_node *dev, int index)
```

この関数はデバイスツリーから特定の割り込みを取得し、割り込み ID を Linux カーネルの仮想 IRQ 番号に変換する機能を持ちます。IRQ 番号は割り込み ID を区別するために使用されます。

パラメータ：

- dev：デバイスノードを指定します。

株式会社日昇テクノロジー

- index：解析・マッピングする割り込みを指定します。デバイスツリーノードには複数の割り込みが含まれる可能性がありますので、ここで何番目の割り込みを指定するかを指定します。インデックスは 0 から始まります。

戻り値：

- 成功：解析・マッピングされたカーネル割り込み番号

- 失敗：0 を返す

14.1.2.3 割り込みサービス関数の実装

open 関数で割り込みを要求する際に、割り込みサービス関数を指定する必要があります。シンプルな割り込みサービス関数の例は以下の通りです。

リスト 5: 割り込みサービス関数の実装

```
1 atomic_t button_status = ATOMIC_INIT(0); //整数型のアトミック変数を定義し、ボタンの状態を保存、初期値を 0 に設定
2 static irqreturn_t button_irq_handler(int irq, void *dev_id)
3 {
4    /* ボタンの状態をインクリメント */
5    atomic_inc(&button_status);
6    return IRQ_HANDLED;
7 }
```

上記のコードから、整数型のアトミック変数を使用してボタンの状態を保存していることがわかります。割り込みが発生した後、アトミック変数がインクリメントされます。アトミック変数が 0 より大きい場合、ボタンが押されたことを意味します。

株式会社日昇テクノロジー

14.1.2.4 .read と.release 関数の実装

.read 関数はユーザースペースにボタンの状態値を返し、.release 関数は終了前にクリーンアップ作業を行います。関数の実装は以下の通りです。

リスト 6: .read と.release 関数の実装

```
1 static int button_read(struct file *filp, char __user *buf, size_t cnt, loff_t *offt)
2 {
3   int error = -1;
4    int button_counter = 0;
5
6    /* ボタンの状態値を読み取る */
7    button_counter = atomic_read(&button_status);
8
9    /* 結果をユーザースペースにコピー */
10    error = copy_to_user(buf, &button_counter, sizeof(button_counter));
11    if(error < 0)
12    {
13        printk("copy_to_user error");
14        return -1;
15    }
16
17    /* ボタンの状態値をクリア */
18    atomic_set(&button_status,0);
```

株式会社日昇テクノロジー

```
19     return 0;
20 }
21
22 /* .release 関数の実装 */
23 static int button_release(struct inode *inode, struct file *filp)
24 {
25     /* open 関数で要求されたピンと割り込みを解放 */
26     gpio_free(button_GPIO_number);
27     free_irq(interrupt_number, device_button);
28     return 0;
29 }
```

- 第 1-20 行、button_read 関数ではボタンの状態値を読み取り、copy_to_user を使用してユーザースペースにコピーした後、ボタンの状態値を 0 に設定します。

- 第 23-29 行、button_release 関数はシンプルで、.open 関数で要求された割り込みと GPIO を解放します。

14.1.3 テストアプリケーションの実装

テストアプリケーションはボタンの状態を読み取り、その状態をプリントするだけです。ソースコードは以下の通りです。

リスト 7: テストアプリケーション

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <fcntl.h>
```

株式会社日昇テクノロジー

```
4 #include <string.h>
5 #include <stdlib.h>
6
7 int main(int argc, char *argv[])
8 {
9    int error = -20;
10    int button_status = 0;
11
12    /* ファイルを開く */
13    int fd = open("/dev/button", O_RDWR);
14   if (fd < 0)
15    {
16       printf("open file : /dev/button error!¥n");
17       return -1;
18    }
19
20    printf("wait button down... ¥n");
21
22    do
23    {
24       /* ボタンの状態を読み取る */
25       error = read(fd, &button_status, sizeof(button_status));
```

株式会社日昇テクノロジー

```
26      if (error < 0)
27      {
28          printf("read file error! ¥n");
29      }
30      usleep(100 * 1000); //100 ミリ秒遅延
31  } while (0 == button_status);
32  printf("button Down !¥n");
33
34  /* ファイルを閉じる */
35  error = close(fd);
36  if (error < 0)
37  {
38      printf("close file error! ¥n");
38  }
40  return 0;
41 }
```

テストアプリケーションはドライバが正常に機能するかどうかのみをテストするために使用されます。ファイルを開いて、状態を読み取り、ファイルを閉じるだけです。開いた後は閉じる必要があります。そうしないと、2 回目の開きが GPIO と割り込みがまだ解放されていないために失敗する可能性があります。

14.1.4 実験準備

ボード上の一部の GPIO はシステムに使用されている可能性があり、ピンが使用されている場合、デバイストリーを再度ロードしたり、ドライバ内で対応するリソースを要求したりすることができなくなる場

株式会社日昇テクノロジー

合があります。たとえば、「Device or resource busy」というエラーが出たり、コードの実行がフリーズしたりするなどです。他のデバイストリープラグインを使用している場合はコメントアウトしてください。

カーネルディレクトリ/arch/arm64/boot/dts/rockchip/overlays の Makefile を編集して、編集したデバイストリープラグインを追加します。そして、Makefile と同じディレクトリレベルにデバイストリープラグインファイルを置き、デバイストリープラグインのコンパイルを行います。

カーネルのルートディレクトリで以下のコマンドを実行するだけです：

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- lubancat4_defconfig
make ARCH=arm64 -j4 CROSS_COMPILE=aarch64-linux-gnu- dtbs
```

生成された.dtbo ファイルはカーネルルートディレクトリの「arch/arm64/boot/dts/rockchip/overlays」ディレクトリにあります。この章でのデバイストリープラグインは「lubancat-button-overlay.dts」で、コンパイル後にカーネルソースコード/arch/arm64/boot/dts/rockchip/overlays ディレクトリに同名の lubancat-button-overlay.dtbo ファイルが生成されます。.dtbo を取得した後、次のステップはそれをシステムにロードすることです。

14.1.4.1 デバイストリープラグインファイルの追加

前のセクションでコンパイルした lubancat-button-overlay.dtbo は、動的にシステムにロードすることができます。lubancat4 ボードの uboot はデバイストリープラグインをロードします。詳細は環境構築の章を参照してください。2 つの簡単なステップを完了するだけです：

-1、ロードする.dtbo ファイルをボードの/boot/dtb/overlays/ディレクトリに置きます。

-2、対応するデバイストリープラグインのロード設定を uEnv.txt 設定ファイルに書き込みます。システム起動時に uEnv.txt からロードするデバイストリープラグインを自動的に読み取ります。

「/boot/uEnv/」ディレクトリの uEnv.txt ファイルを開き、vim や nano エディタを使用してファイルを開き、「dtoverlay=<デバイストリープラグインパス>」の形式でデバイストリープラグインを

株式会社日昇テクノロジー

uEnv.txt に記述します。

追加した後、開発ボードを再起動し、コマンド ls /proc/device-tree/を使用して、button_interrupt ディレクトリがあるかどうかを確認します。あれば、正常にロードされたことになります。

14.1.4.2 ドライバプログラムとテストプログラムのコンパイル

このセクションの実験で使用する Makefile は以下の通りです：

リスト 8: Makefile(../linux_driver/button_interrupt/interrupt に位置)

```
1 KERNEL_DIR=../../kernel/
2
3 ARCH=arm64
4 CROSS_COMPILE=aarch64-linux-gnu-
5 export ARCH CROSS_COMPILE
6
7 obj-m := interrupt.o
8 out = test_app
9
10 all:
11     $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) modules
12     $(CROSS_COMPILE)gcc -o $(out) test_app.c
13
14 .PHONY:clean
15 clean:
```

株式会社日昇テクノロジー

```
16    $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) clean

17    rm test_app
```

ドライバコード、例えば：interrupt をカーネルと同じレベルのディレクトリに配置し、ドライバディレクトリで以下のコマンドを入力して、ドライバモジュールとテストプログラムをコンパイルします：

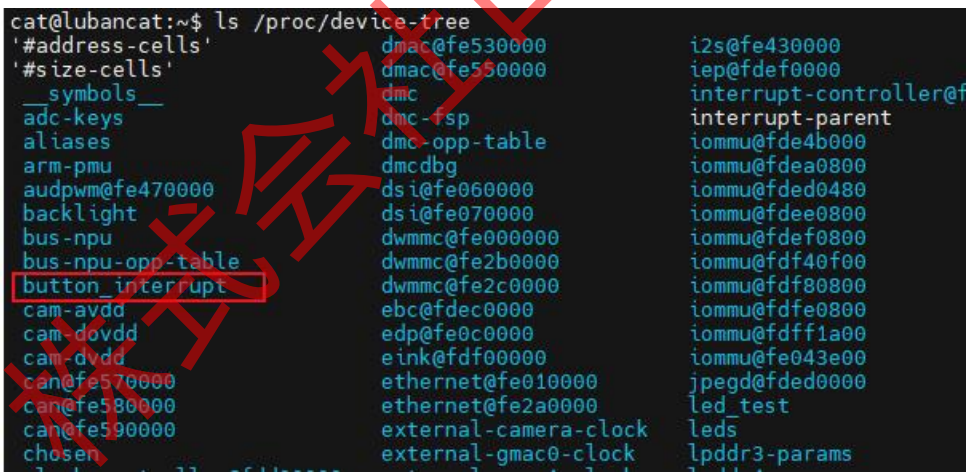
```
make
```

14.1.5 実験現象

コンパイルされたドライバ、アプリケーション、デバイストリープラグインを開発ボードにコピーします。この部分の内容は既に前の章で詳しく説明されているので、ここでは省略します。

モジュールをロードする前に、/boot/uEnv.txt ファイルでボードの元の KEY に関連するデバイストリープラグインがロードされているかを確認します。KEY に関連するデバイストリープラグインが有効な場合は、'#'を追加して KEY に関連するデバイストリープラグインをコメントアウトします。そして、ボタン割り込みのデバイストリープラグインを追加した後、開発ボードを再起動します。

```
ls /proc/device-tree
```



カーネルモジュールをロードし、テストプログラムを実行します：

```
cat@lubancat:~$ sudo insmod interrupt.ko
cat@lubancat:~$
cat@lubancat:~$
cat@lubancat:~$ dmesg |tail^C
cat@lubancat:~$
cat@lubancat:~$
cat@lubancat:~$
cat@lubancat:~$ sudo ./test_app
wait button down...
button Down !
```

キーがない場合は、40pin の GND と 3.3V をデュポン線で順次ショートしてシミュレートします。ピンの電圧レベルを高くしたり低くしたりします。ただし、IO を入力に設定した後にショートし、3.3V 以上の電源に接続しないように注意してください。そうすると、メインコントローラが損傷する可能性があります。

test_app を実行するときに、以下のコマンドを使用して、ドライバが正常にロードされ、初期化されたかどうかを確認できます：

```
cat /proc/interrupts
```

```
er
 85:          0          0          0          0          0          0          0          0     GICv3 120 Level     fea30000.dma-controll
er
 86:          0          0          0          0          0          0          0          0     GICv3 121 Level     fea30000.dma-controll
er
 87:        127          0          0          0          0          0          0          0     GICv3 350 Level     fea90000.i2c
 88:          0          0          0          0          0          0          0          0     GICv3 347 Edge      feaf0000.watchdog
 89:       9281          0          0          0          0          0          0          0     GICv3 360 Level     feb20000.spi
 90:          0          1          0          0          0          0          0          0     GICv3 378 Level     rk_pwm_irq
 91:          0          0          0          0          0          0          0          0     GICv3 379 Level     rk_pwm_pwr_irq
 92:          0          0          0          0          0          0          0          0     GICv3 429 Level     rockchip_thermal
 93:       1100          0          0          0          0          0          0          0     GICv3 430 Level     fec10000.saradc
 94:        142          0          0          0          0          0          0          0     GICv3 356 Level     fec90000.i2c
 95:          0          0          0          0          0          0          0          0     GICv3 122 Level     fed10000.dma-controll
er
 96:          0          0          0          0          0          0          0          0     GICv3 123 Level     fed10000.dma-controll
er
102:          0          0          0          0          0          0          0          0     GICv3 455 Edge      debug-signal
103:          1          0          0          0          0          0          0          0     GICv3 365 Level     debug
104:          0          0          0          0          0          0          0          0  rockchip_gpio_irq   7 Level     rk806
105:          0          0          0          0          0          0          0          0     rk806   0 Edge      rk805_pwrkey_fall
106:          0          0          0          0          0          0          0          0     rk806   1 Edge      rk805_pwrkey_rise
107:          0          0          0          0          0          0          0          0     rk806   7 Level     rk806_vb_low
108:          0          0          0          0          0          0          0          0     GICv3 252 Level     dwc3
109:          0          0          0          0          0          0          0          0     GICv3 254 Level     xhci-hcd:usb5
119:          0          0          0          0          0          0          0          0   ITS-MSI 570425352 Edge      PCIe PME
120:          0          0          0          0          0          0          0          0  rockchip_gpio_irq   8 Level     hym8563
121:          0          0          0          0          0          0          0          0  rockchip_gpio_irq  22 Level     fsc_interrup
t_int_n
122:          0          0          0          0          0          0          0          0  rockchip_gpio_irq  16 Edge      headset_dete
ct
123:          0       3217          0          0          0          0          0          0   ITS-MSI 570949632 Edge      rtw88_pci
IPI0:      4343       7855       8144       8963        824       1667        932       1393       Rescheduling interrupts
IPI1:      6325       3247       3274       2906      16296      11139      11892      10304       Function call interrupts
IPI2:         0          0          0          0          0          0          0          0       CPU stop interrupts
IPI3:         0          0          0          0          0          0          0          0       CPU stop (for crash dump) interrupts
IPI4:      1149       1961       2299       2290       1287        916        928        920       Timer broadcast interrupts
IPI5:      3393       2345        946       1755       3001       1424       1902       1543       IRQ work interrupts
IPI6:         0          0          0          0          0          0          0          0       CPU wake-up interrupts
Err:          0
cat@lubancat:~$
```

株式会社日昇テクノロジー

登録した割り込みなどの情報を確認できます。

14.2 割り込みの高度な使用

Linux の割り込みについて、以下の 2 点を知っておく必要があります：

- 1、Linux は割り込みのネストをサポートしません。

- 2、割り込みサービス関数の実行時間はできるだけ短くすべきです。つまり、迅速に入って迅速に出るべきです。

しかし、割り込みが発生した後に長時間処理が必要な場合もあります。例えば、ネットワーク伝送によって発生する割り込みでは、ネットワーク伝送の割り込みが発生した後に受信または送信データを処理するのに長い時間がかかります。Linux では割り込みがネストできないため、このような状況で他の割り込みが発生すると、即時に対応できなくなります。この問題を解決するために、Linux では「割り込みの上半部」と「割り込みの下半部」という概念を導入しました。割り込みの上半部では割り込みの簡単な処理のみを行い、時間がかかる処理は割り込みの下半部で行います。これにより、他の割り込みに対して即時に対応でき、システムのリアルタイム性が向上します。この概念は割り込みの階層化とも呼ばれます：

- 「上半部」は割り込みサービス関数内で実行されるコード部分を指します。

- 「下半部」は本来割り込みサービス関数内で実行されるべきだが、何らかの方法でそれらを割り込みサービス関数の外で実行させる部分を指します。

すべての割り込み処理に「上半部」と「下半部」が必要なわけではありません。先に記述したボタン割り込みプログラムのように、比較的時間がかからない処理であれば、割り込みの「上半部」だけで処理できます。「下半部」を実現するメカニズムには、ソフト割り込み、tasklet、ワークキュー、スレッド irq があります。

割り込みの階層化「下半部」を模擬するために、時間がかかる操作を加えてみましょう。

14.2.1 ソフト割り込みと tasklet

tasklet はソフト割り込みに基づいて実装されており、多くの類似点があります。これらを一緒に紹介します。

株式会社日昇テクノロジー

14.2.1.1 ソフト割り込み(softirq)

Linux 4.x は限られた数のソフト割り込みしかサポートしておらず、バージョンによって異なりますが、通常は 10 個です。Linux カーネルでは、enum 型の変数で使用可能なすべてのソフト割り込みがリストアップされています。

リスト 9: ソフト割り込みの種類(include/linux/interrupt.h)

```
1 enum
2 {
3 HI_SOFTIRQ=0,
4 TIMER_SOFTIRQ,
5 NET_TX_SOFTIRQ,
6 NET_RX_SOFTIRQ,
7 BLOCK_SOFTIRQ,
8 IRQ_POLL_SOFTIRQ,
9 TASKLET_SOFTIRQ,
10 SCHED_SOFTIRQ,
11 HRTIMER_SOFTIRQ, /* Unused, but kept as tools rely on the numbering. Sigh! */
```

ハード割り込みに例えると、この enum 型はソフト割り込みの割り込み番号をリストアップしており、「登録」する際やソフト割り込みをトリガする際に使用します。

ソフト割り込み「登録」関数は以下の通りです：

株式会社日昇テクノロジー

リスト 10: ソフト割り込み登録関数

```
1 void open_softirq(int nr, void (*action)(struct softirq_action *))
2 {
3 softirq_vec[nr].action = action;
4 }
```

パラメータ：

- nr：「登録」するソフトウェア割り込みの割り込み番号を指定します
- action：ソフトウェア割り込みの割り込みサービス関数を指定します

戻り値：なし

関数実装では、softirq_vec 変数への単純な代入が行われます。この変数はカーネルソースコード内で見つけることができます。

リスト 11: ソフト割り込み「割り込みベクタテーブル」

```
1 static struct softirq_action softirq_vec[NR_SOFTIRQS]
```

これは NR_SOFTIRQS の長さを持つ softirq_action 型の配列で、NR_SOFTIRQS はソフト割り込みの「割り込み番号」enum で定義されている長さ、つまり 10 です。この配列は全体として硬割り込みの割り込みベクタテーブルに相当します。次に配列の型「struct softirq_action」を見てみましょう。

リスト 12: ソフト割り込み構造体

```
1 struct softirq_action
2 {
3 void (*action)(struct softirq_action *);
4 };
```

それはパラメータが 1 つだけで、ソフトウェア割り込み関数を登録するパラメータ open_softirq です。これにより、配列 softirq_vec がソフトウェア割り込みの割り込みベクターテーブルであることがわかりま

株式会社日昇テクノロジー

す。所謂のソフトウェア割り込み関数の登録とは、割り込み番号に基づいて割り込みサービス関数のアドレスを softirq_vec 配列の対応する位置に書き込むことです。

ソフト割り込み登録後、ソフト割り込みを「トリガ」する関数を呼び出してソフト割り込みのサービス関数を実行する必要があります。

リスト 13: 割り込み interrupt-controller ノード

```
1 void raise_softirq(unsigned int nr);
```

パラメータ：

- nr：トリガーするソフトウェア割り込み

戻り値：なし

14.2.1.2 tasklet

tasklet はソフト割り込みに基づいて実装されています。特にパフォーマンスに特別な要件がない場合は、tasklet を使用して割り込みの階層化を実装することをお勧めします。なぜなら、ソフト割り込みサービス関数は全体で共有される配列になっており、マルチ CPU システムではすべての CPU がアクセスできるため、並行性や再入可能性などを自分で管理する必要があり、プログラミングの負担が増えるからです。

ソフト割り込みリ

ソースは非常に限られており、特定のデバイス用に予約されているものもあります（NET_TX_SOFTIRQ, NET_RX_SOFTIRQ など）。これらのケースでは、ソフト割り込みを使用して割り込みの階層化を実装します。

tasklet_struct 構造体

ドライバでは、tasklet_struct 構造体を使用して tasklet を表します。構造体の定義は以下の通りです：

株式会社日昇テクノロジー

リスト 14: ソフト割り込み

```
1 struct tasklet_struct

2 {

3 struct tasklet_struct *next;

4 unsigned long state;

5 atomic_t count;

6 void (*func)(unsigned long);

7 unsigned long data;

8 };
```

パラメータの紹介は以下の通りです：

• next：リストの次の tasklet_struct を指す、このパラメータは自分で設定する必要はありません。

• state：tasklet の状態を保存します。0 の場合、tasklet はまだスケジュールされていません。

TASKLET_STATE_SCHED の場合、tasklet がスケジュールされ、実行準備ができています。

TASKLET_STATE_RUN の場合、実行中を意味します。

• count：参照カウンターで、0 の場合は tasklet が利用可能で、それ以外の場合は tasklet が禁止されています。

• func：tasklet の処理関数を指定します。

• data：tasklet の処理関数のパラメータを指定します。

tasklet 初期化関数

関数プロトタイプは以下の通りです：

株式会社日昇テクノロジー

リスト 15: tasklet 初期化関数

```
1 void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long), unsigned long data)
2 {
3     t->next = NULL;
4     t->state = 0;
5     atomic_set(&t->count, 0);
6     t->func = func;
7     t->data = data;
8 }
```

パラメータ：

- t：初期化する tasklet_struct 構造体を指定します。
- func：tasklet 処理関数を指定します。これは割り込み内の割り込みサービス関数に相当します。
- data：tasklet 処理関数のパラメータを指定します。関数の実装は、設定されたパラメータに基づいて tasklet_struct 構造体を充填します。

戻り値：なし

tasklet のトリガ

ソフト割り込みと同様に、tasklet をトリガするための関数が必要です。関数の定義は以下の通りです：

リスト 16: tasklet トリガ関数

```
1 static inline void tasklet_schedule(struct tasklet_struct *t)
2 {
3     if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
4         __tasklet_schedule(t);
5 }
```

株式会社日昇テクノロジー

パラメータ：

- t：tasklet_struct 構造体。

戻り値：なし

14.2.1.3 tasklet を用いた割り込みの階層化実験

実験はボタン割り込みプログラムを基に行われます。ボタン割り込みは元々割り込みの階層化を必要としませんが、ここではそれを例に tasklet の具体的な使用方法を簡単に紹介します。tasklet の使用は非常にシンプルで、tasklet 構造体の定義、定義した tasklet 構造体の初期化、tasklet 割り込み処理関数の実装、tasklet 割り込みのトリガーが主なステップです。

以下のソースコードは「ボタン割り込みプログラム」に tasklet 関連のコードを追加したものです。ここでは tasklet 関連のコードのみをリストアップしています。参照ソースコードは linux_driver/button_interrupt/interrupt_tasklet ディレクトリにあります。

リスト 17: tasklet 関連のコード

```
1 /*--------------第一部分---------------*/
2 struct tasklet_struct button_tasklet; //グローバルに tasklet_struct 型構造体を定義
3
4 /*--------------第二部分-----------------*/
5 void button_tasklet_handler(unsigned long data)
6
7    int counter = 1;
8    mdelay(200);
9    printk(KERN_ERR "button_tasklet_handler counter = %d ¥n", counter++);
10    mdelay(200);
```

株式会社日昇テクノロジー

```
11    printk(KERN_ERR "button_tasklet_handler counter = %d ¥n", counter++);
12    mdelay(200);
13    printk(KERN_ERR "button_tasklet_handler counter = %d ¥n", counter++);
14    mdelay(200);
15    printk(KERN_ERR "button_tasklet_handler counter = %d ¥n", counter++);
16    mdelay(200);
17    printk(KERN_ERR "button_tasklet_handler counter = %d ¥n", counter++);
18 }
19
20 /*--------------第三部分-----------------*/


21 static int button_open(struct inode *inode, struct file *filp)
22 {
23    /* 初期化コード省略 */
24    /* button_tasklet  の初期化 */
25    tasklet_init(&button_tasklet, button_tasklet_handler, 0);
26
27    return 0;
28 }
29
30
31 static irqreturn_t button_irq_handler(int irq, void *dev_id)
```

株式会社日昇テクノロジー

```
32 {
33     printk(KERN_ERR "button_irq_handler----------enter");
34     /* ボタンの状態をインクリメント */
35     atomic_inc(&button_status);
36
37     tasklet_schedule(&button_tasklet);
38
39     printk(KERN_ERR "button_irq_handler-----------exit");
40     return IRQ_RETVAL(IRQ_HANDLED);
41 }
```

コードの各部分は以下のように紹介されます：

- 第 2 行：tasklet_struct 型の構造体を定義します。
- 第 5-18 行：tasklet の「割り込みサービス関数」を定義します。ここでは tasklet の割り込みサービス関数内で遅延と printk ステートメントを使用して時間のかかる操作をシミュレートします。
- 第 21-28 行：元のコードの基盤の上に tasklet_init 関数を呼び出して tasklet_struct 型の構造体を初期化します。
- 第 37 行：割り込みサービス関数内で tasklet_schedule 関数を呼び出して tasklet 割り込みをトリガします。ボタンの割り込みサービス関数の開始と終了にプリントステートメントを追加します。通常、プログラムは最初にボタンの割り込みサービス関数を実行し、割り込みサービス関数から抜けた後に割り込みの下半部、すなわち tasklet の「割り込みサービス関数」を実行します。

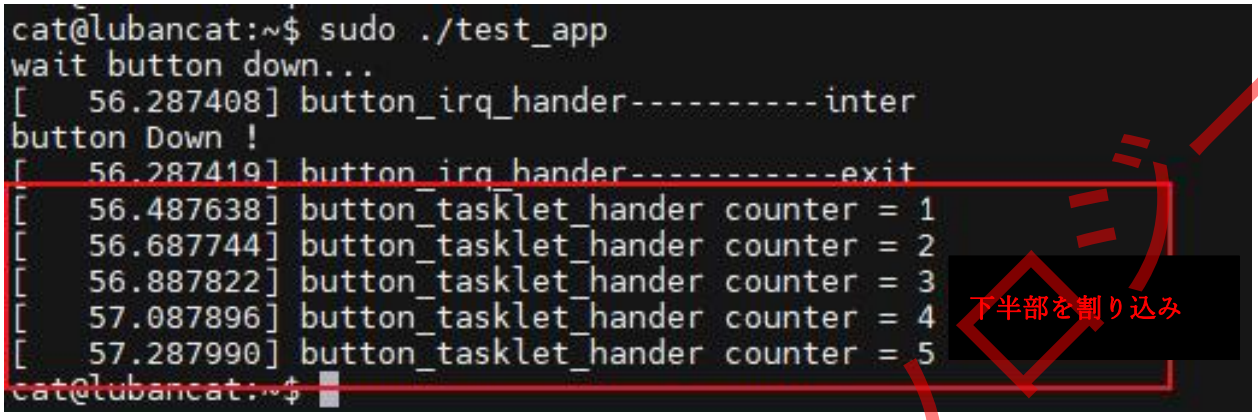
14.2.1.4 ダウンロードと検証

デバイストリープラグインのロード方法は前の章で何度も触れられているので、ここでは省略します。疑

株式会社日昇テクノロジー

問がある場合は、以前のセクションを再度確認してください。

修正されたドライバプログラムをコンパイルし、開発ボードにダウンロードし、insmod を使用してドライバをロードした後、以下に示すようにテストアプリケーションを実行します。



14.2.2 ワークキュー

ソフト割り込みや tasklet とは異なり、ワークキューはカーネルスレッド上で実行され、再スケジュールやスリープが可能です。tasklet と比較して、より高いリアルタイム性を持ちます。割り込みの下半部が再スケジュールやスリープを受け入れることができる場合は、ワークキューの使用をお勧めします。

ワークキューの使用は tasklet に似ており、主にワーク構造体の定義、ワークの初期化、ワークのトリガーが含まれます。

14.2.2.1 ワーク構造体

「ワークキュー」は「キュー」ですが、ユーザーは「キュー」やキューを処理するカーネルスレッドについて心配する必要はありません。これらはカーネルが処理します。ユーザーは具体的なワークを定義し、ワークを初期化するだけで済みます。ドライバ内で一つのワーク構造体は一つのワークを表します。ワーク構造体は以下のように示されます：

株式会社日昇テクノロジー

リスト 18: work_struct 構造体

```
1 struct work_struct {
2     atomic_long_t data;
3     struct list_head entry;
4     work_func_t func;
5     #ifdef CONFIG_LOCKDEP
6     struct lockdep_map lockdep_map;
7     #endif
8 };
```

重要なパラメータは「work_func_t func;」で、これは「ワーク」の処理関数を指定します。

リスト 19：ワーク関数

```
1 void (*work_func_t)(struct work_struct *work);
```

14.2.2.2 ワークの初期化関数

カーネルは以下の初期化マクロを提供しています：

リスト 20：ワーク初期化マクロ定義

```
1 #define INIT_WORK(_work, _func)
```

このマクロは 2 つのパラメータを持ち、_work は初期化するワーク構造体を指定し、_func はワークの処理関数を指定します。

14.2.2.3 ワークのトリガー関数

ドライバのワーク関数が実行されると、関連するカーネルスレッドはワーク構造体に指定された処理関数を実行します。ドライバ関数は以下の通りです：

リスト 21: ワークをトリガーする関数

株式会社日昇テクノロジー

```
1 static inline bool schedule_work(struct work_struct *work)
2 {
3     return queue_work(system_wq, work);
4 }
```

この関数はワーク構造体のパラメータのみを持ちます。

14.2.2.4 ワークキューの実験

ワークキューの実験もボタン割り込みプログラムを基に実装されます。ここではワークキュー関連のコードのみをリストアップしています。完全な内容は、このセクションに付随するドライバプログラムを参照してください。（ここではドライバプログラムのみを変更し、他の内容は変更しません）。参照ソースコードは、linux_driver/button_interrupt/interrupt_work ディレクトリにあります。

interrupt_work ディレクトリ下。

リスト 22：ワークキュー関連関数

```
1 /*--------------第一部分-----------------*/
2 struct work_struct button_work;
3
4 /*--------------第二部分-----------------*/
5 void work_hander(struct work_struct *work)
6 {
7 int counter = 1;
8 mdelay(200);
9 printk(KERN_ERR "work_hander counter = %d ¥n", counter++);
10 mdelay(200);
```

株式会社日昇テクノロジー

```
11 printk(KERN_ERR "work_hander counter = %d ¥n", counter++);

12 mdelay(200);

13 printk(KERN_ERR "work_hander counter = %d ¥n", counter++);

14 mdelay(200);

15 printk(KERN_ERR "work_hander counter = %d ¥n", counter++);

16 mdelay(200);

17 printk(KERN_ERR "work_hander counter = %d ¥n", counter++);

18 }

19

20 /*--------------第三部分-----------------*/

21 static int button_open(struct inode *inode, struct file *filp)



22 {

23 /*---------------省略----------------*/

24 /* button_work  の初期化*/

25 INIT_WORK(&button_work, work_hander);

26 return 0;

27 }

28

29 /*--------------第四部分-----------------*/

30 static irqreturn_t button_irq_hander(int irq, void *dev_id)

31 {
```

株式会社日昇テクノロジー

```
32 /* ボタンの状態をインクリメント*/
33 atomic_inc(&button_status);
34 schedule_work(&button_work);
35 return IRQ_HANDLED;
36 }
```

- 2 行目：work_struct 型の構造体を定義します。
- 5-18 行目：ワークキューの「割り込みサービス関数」を定義し、遅延と printk 文を使って時間のかかる操作を模擬します。
- 21-27 行目：元のコードの基礎上で INIT_WORK マクロを呼び出して work_struct 型構造体と割り込みの下半分関数を初期化します。
- 34 行目：割り込みサービス関数内で schedule_work 関数を呼び出し、割り込みの下半分をトリガーします。

tasklet での割り込み階層の実装と似ており、使用方法はほぼ同じですので、ここでは詳細は省略します。テストは以下の図のように表示されます：

```
cat@lubancat:~$ sudo ./test_app
wait button down...
button Down !
[  181.213177] work_hander counter = 1
cat@lubancat:~$ [  181.413341] work_hander counter = 2
[  181.613426] work_hander counter = 3
[  181.813504] work_hander counter = 4
[  182.013586] work_hander counter = 5
```

14.2.3 スレッド化された IRQ

割り込みをスレッド化すると、割り込みがカーネルスレッドとして実行され、異なるリアルタイム優先度を持つことができます。負荷が高い時には、割り込みスレッドが割り込みされて、より高優先度のリアルタイムタスクが適時に応答できない状況を避けることができます。

株式会社日昇テクノロジー

スレッド化された IRQ は devm_request_threaded_irq()によって要求され、この関数の使用は request_irq()や devm_request_irq()に似ており、例で直接置き換えることができます。

リスト 23: ワークキュー関連の関数

```
1 request_threaded_irq(unsigned int irq, irq_handler_t handler,
2 irq_handler_t thread_fn,
3 unsigned long flags, const char *name, void *dev);
```

- unsigned int irq：割り込み番号、要求された割り込みベクタ

- irq_handler_t handler：割り込み処理関数。ドライバでは、このパラメータは通常 NULL です。NULL の場合、デフォルトの処理を使用します。これは割り込みの「上半部」に相当します。

- irq_handler_t thread_fn：割り込みスレッド。割り込みが発生すると、handler が NULL であれば、thread_fn をカーネルスレッドで直接実行します。

- flags：割り込みの属性やトリガー方式を指定します。

- name：割り込みの名前。

- dev：割り込み処理プログラムへ渡されるパラメータ。共有割り込みを登録する場合、このパラメータは NULL にすることはできません。このパラメータを使用して共有割り込み時に割り込みを区別することができます。また、デバイスの情報を保存する構造体変数にこのパラメータを渡すことで、割り込み処理関数がそのデバイスの情報を取得できるようにすることもできます。

第 15 章 入力サブシステム

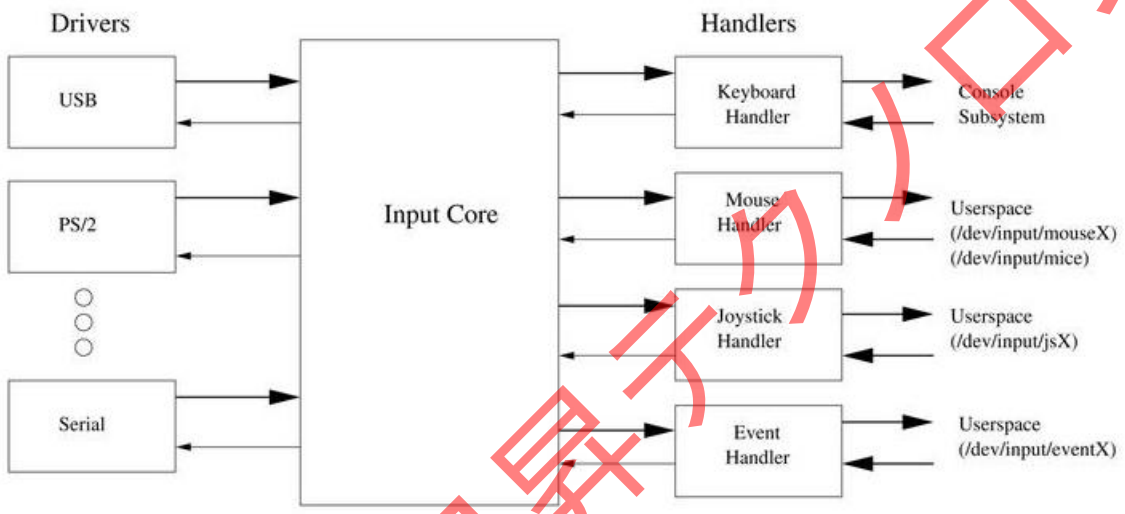
コンピュータの入力デバイスには、キーボード、マウス、タッチスクリーン、ゲームコントローラーなど多岐にわたるものがあります。Linux カーネルは、すべての入力デバイスを統一的に管理するために入力サブシステムを設計しました。これにより、上位のアプリケーションに対して統一された抽象層が提供され、各入力デバイスのドライバプログラムは発生した入力イベントを報告するだけでよくなります。

株式会社日昇テクノロジー

ここでは、ボタン入力イベント（GPIO を模倣してもよい）を例に入力サブシステムの使用方法を説明します。この章の対応するソースコードとデバイスツリープラグインは「/linux_driver/input_sub_system」ディレクトリにあります。

15.1 入力サブシステムの概要

Linux は、さまざまな入力デバイスを統一するために、入力サブシステムを Drivers（ドライバ層）、Input Core（入力サブシステムのコア層）、Handlers（イベント処理層）の 3 つの部分に分けました。



- Drivers は、ハードウェアデバイスへの読み書きアクセス、割り込みの設定、およびハードウェアが生成するイベントを Input Core が定義する規格に変換して Handlers に送ることを主に担います。

- Input Core は、Drivers に規格とインターフェイスを提供し、Handlers にイベント処理を通知する役割を果たします。

- Handlers は、ハードウェアに関する具体的な操作を行わず、純粋なソフトウェア層です。キー、キーボード、マウス、ゲームコントローラーなどの異なるソリューションを含みます。

最終的に、すべての入力デバイスからの入力情報は、以下の構造体に抽象化されます：

株式会社日昇テクノロジー

リスト 1: struct input_event 構造体（カーネルソースコード/include/uapi/linux/input.h 内）

```
1 // 入力イベント
2 struct input_event{
3     struct timeval time; // イベントが発生した時間
4     __u16 type; // 入力デバイスのタイプ、マウス、キーボード、タッチスクリーン
5     __u16 code;
6     __s16 value;
7 }
```

- time：イベントが発生した時間。

- type：入力デバイスのタイプ。

- code：デバイスのタイプに応じて異なる意味を持ちます。デバイスタイプがキーの場合、code はキー値（例：どのキーかなど）を表します。

- value：デバイスのタイプに応じて異なる意味を持ちます。デバイスタイプがキーの場合、value はリリースまたはプレスを表します。

この章の目的は、入力サブシステムと割り込みを基にしたキーボードドライバプログラムを作成することであり、Input Core が提供するインターフェースを理解し、どのようにキー情報をイベントとして報告するかを理解することに重点を置いています。

入力サブシステムの Input Core 実装コードは「カーネルソースコード/drivers/input/input.c」と「カーネルソースコード/include/linux/input.h」の 2 つのファイルによって提供されており、これらの API を操作することで、入力イベントの登録、初期化、報告、解除などの作業を行うことができます。以下では、入力サブシステムでよく使われる API インターフェースとデータ構造について紹介します。

株式会社日昇テクノロジー

15.1.1 input_dev 構造体

どのタイプの入力デバイスであっても、どのタイプのイベントを送信しても、カーネル内の入力デバイスは struct input_dev のインスタンスとして表され、具体的な入力デバイスを表します。この構造体は後で具体的なデバイスに応じて初期化されます。構造体のメンバーは以下の通りです（input_dev には多くのパラメータがありますが、手で設定する必要がないものもあるため、ここではよく使われるパラメータのみをリストアップし、紹介します。完全な内容は input.h ファイルにあります）。

リスト 2: input_dev 構造体（カーネルソースコード/include/linux/input.h）

```
1 struct input_dev {
2     const char *name; //ユーザーに提供される入力デバイスの名前
3     const char *phys; //開発者に提供されるデバイスノードの名前
4     const char *uniq; //一意の ID 番号を指定
5     struct input_id id; //入力デバイス識別 ID
6
7     unsigned long propbit[BITS_TO_LONGS(INPUT_PROP_CNT)];
8
9     unsigned long evbit[BITS_TO_LONGS(EV_CNT)]; //デバイスがサポートするイベントタイプを指定
10     unsigned long keybit[BITS_TO_LONGS(KEY_CNT)]; //サポートされるキー値を記録
11     unsigned long relbit[BITS_TO_LONGS(REL_CNT)]; //サポートされる相対座標ビットマップを記録
12     unsigned long absbit[BITS_TO_LONGS(ABS_CNT)]; //サポートされる絶対座標ビットマップを記
録
13     unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)];
14     unsigned long ledbit[BITS_TO_LONGS(LED_CNT)];
```

株式会社日昇テクノロジー

```
15    unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];
16    unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];
17    unsigned long swbit[BITS_TO_LONGS(SW_CNT)];
18    /*----------以下の構造体メンバーは省略----------------*/
19 };
```

構造体のメンバーで最も重要なのは evbit、keybit、relbit などの配列で、これらの配列によってデバイスの入力イベントのタイプとキー値が設定されます。

- evbit: サポートされるイベントタイプを指定します。これは実際の入力デバイスが生成できるイベントに基づいて選択する必要があります。選択可能なオプションは以下の通りです。

リスト 3: 入力サブシステムのイベントタイプ（カーネルソースコード /include/uapi/linux/input-event-codes.h）

```
1 #define EV_SYN 0x00 //同期イベント
2 #define EV_KEY 0x01 //キーボード、ボタン、または他の類似のキーデバイスを記述するために使用されます。
3 #define EV_REL 0x02 //相対位置の変化を記述するために使用されます、例えばマウスの動き
4 #define EV_ABS 0x03 //絶対位置の変化を記述するために使用されます、例えばタッチスクリーンのタッチポイント座標
5 #define EV_MSC 0x04 //その他のイベントタイプ
6 #define EV_SW 0x05 //二進開閉デバイスを記述するために使用されます、例えばダイヤルスイッチ。
7
8 #define EV_SND 0x12
9 #define EV_REP 0x14
```

株式会社日昇テクノロジー

```
10 #define EV_FF 0x15

11 #define EV_PWR 0x16

12 #define EV_FF_STATUS 0x17

13 #define EV_MAX 0x1f

14 #define EV_CNT (EV_MAX+1)
```

上記のコードで最初の数個のマクロ定義は、より一般的な入力イベントタイプを記述しています。完全なイベントリストの紹介はカーネルソースコードディレクトリの「~/Documentation/input/event-codes.rst」カーネルドキュメントを参照してください。明らかに、この章で使用するキーのイベントタイプは EV_KEY を使用すべきです。

- keybit：サポートされるキー値を記録します。「キー値」はプログラム内で異なるキーを区別するために使用されます。選択可能な「キー値」は以下の通りです。

リスト 4: 入力サブシステム - キー値（カーネルソースコード /include/uapi/linux/input-event-codes.h）

```
1 #define KEY_RESERVED 0

2 #define KEY_ESC 1

3 #define KEY_1 2

4 #define KEY_2 3

5 #define KEY_3 4

6 #define KEY_4 5

7 /*-----------以下略-------------*/
```

「キー値」はいくつかの数字です。実際のデバイスとキーが対応していれば良いです。

- relbit、absbit：これら二つのパラメータは上述の keybit と evbit に関連しています。もし evbit で EV_KEY のみを選択した場合、relbit（相対座標）や absbit（絶対座標）やその他省略されたメンバー

株式会社日昇テクノロジー

を設定する必要はありません。これらは使用時に詳しく説明します。異なる入力イベントタイプを使用する場合は、異なる設定が必要です。

要するに、input_dev 構造体には多くのメンバーがありますが、具体的な入力デバイスに対しては、使用するいくつかの属性のみを設定する必要があります。

15.1.2 input_dev 構造体の割り当てと解放

input_dev 構造体は、入力デバイスを表し、入力サブシステムのサブデバイス番号を占有します。入力サブシステムは、input_dev 構造体を割り当てて解放するための関数を提供しています。input_dev 構造体のメンバーが多く、初期化プロセスが比較的複雑であるため、通常は入力サブシステムが提供するインターフェース関数を使用して input_dev 構造体を割り当てて解放します。以下はその例です。

リスト 5: input_dev 割り当て関数（カーネルソースコード/drivers/input/input.c）

```
1 struct input_dev *input_allocate_device(void)
2 {
3     static atomic_t input_no = ATOMIC_INIT(-1);
4     struct input_dev *dev;
5
6     dev = kzalloc(sizeof(*dev), GFP_KERNEL); //動的メモリ割り当て
7     if (dev) {
8         dev->dev.type = &input_dev_type;
9         dev->dev.class = &input_class; //dev->dev は struct device 型構造体
10        device_initialize(&dev->dev); //dev->dev 構造体内部メンバーの初期化
11        mutex_init(&dev->mutex); //ミューテックスの初期化
12        spin_lock_init(&dev->event_lock); //スピンロックの初期化
```

株式会社日昇テクノロジー

```
13      timer_setup(&dev->timer, NULL, 0); //タイマーの初期化
14      INIT_LIST_HEAD(&dev->h_list); //handle リストノードの初期化
15      INIT_LIST_HEAD(&dev->node); //入力デバイスリストノードの初期化
16
17      dev_set_name(&dev->dev, "input%lu", (unsigned long)atomic_inc_return(&input_no)); //デバイス
名の設定
18
19      __module_get(THIS_MODULE);
20    }
21
22    return dev;
23 }
24 EXPORT_SYMBOL(input_allocate_device);
```

パラメータ：なし

戻り値：

- 成功：struct input_dev 型のポインタ
- 失敗：NULL

この関数をどのように呼び出すかを知るだけで十分です。さらに深く学びたい方は、入力サブシステムの実装コードの分析を試みることができますが、入力サブシステムのコード分析だけで長い記事が書けるため、ここでは詳細なソースコード分析には立ち入りません。

リスト 6: input_dev 解放関数（カーネルソースコード/drivers/input/input.c）

```
1 void input_free_device(struct input_dev *dev)
```

パラメータ：dev：struct input_dev 型ポインタ

株式会社日昇テクノロジー

戻り値：なし

割り当てと解放関数のインターフェイスは比較的シンプルです。割り当て関数 input_allocate_device が成功すると、割り当てられた`input_dev`構造体のアドレスを返し、失敗すると NULL を返します。解放関数 input_free_device には、解放する input_dev 構造体を指定する dev パラメータのみがあります。

15.1.3 input_dev 構造体の登録と解除

input_dev が割り当てられた後、実際の入力デバイスに応じて input_dev 構造体を設定する必要があります。具体的な設定は実験コードの作成部分で詳しく説明されます。設定が完了した後、登録と解除関数を使用して input_dev を入力サブシステムに登録する必要があります。登録と解除関数は以下のとおりです：

リスト 7: input_dev 登録関数（カーネルソースコード/drivers/input/input.c）

```
1 int input_register_device(struct input_dev *dev)
```

パラメータ：dev：struct input_dev 型ポインタ

戻り値：

- 成功：0
- 失敗：非 0 の値

input_register_device 関数は、入力デバイス（input_dev）を入力サブシステムのコア層に登録します。この関数を使用する際には、次の点に注意してください：

- この関数で登録される input_dev は、input_allocate_device 関数で割り当てられたものでなければなりません。
- 登録前に、実際の入力デバイスに応じて input_dev 構造体を設定する必要があります。
- 登録が失敗した場合は、input_free_device 関数を呼び出して input_dev 構造体を解放する必要があります。

株式会社日昇テクノロジー

- 登録が成功した場合は、関数の終了時に input_unregister_device 関数を使用して input_dev 構造体を登録解除するだけで、input_free_device 関数を呼び出して input_dev 構造体を解放する必要はありません。

リスト 8: input_dev 登録解除関数（カーネルソースコード/drivers/input/input.c）

```
1 void input_unregister_device(struct input_dev *dev)
```

パラメータ：dev：struct input_dev 型ポインタ

戻り値：なし

input_unregister_device は登録解除関数で、入力サブシステムのリソースは限られているため、使用しない場合は登録解除するべきです。input_unregister_device 関数を呼び出した後は、input_free_device 関数を呼び出して input_dev を解放する必要はありません。

15.1.4 イベント報告関数と報告終了関数

キーが押された後、報告関数を使用して入力サブシステムのコア層にキーイベントを報告し、報告後には報告終了情報を送信する必要があります。関数の定義は以下の通りです。

リスト 9: 一般的なイベント報告関数（カーネルソースコード/drivers/input/input.h）

```
1 void input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value);
```

input_event 関数はイベントを報告するために使用され、以下の 4 つのパラメータがあります。

パラメータ：

- dev：入力デバイス（input_dev 構造体）を指定します。

- type：イベントタイプ。実際の入力デバイスに応じて input_dev 構造体を設定する際に input_dev->evbit パラメータを設定し、「生成可能な」イベントタイプ（複数可）を設定します。イベントを報告する際には、これらのイベントタイプから選択する必要があります。

- code：コード。キーの例として、キーのコードは設定されたキー値です。

株式会社日昇テクノロジー

- value：イベントの値を指定します。

戻り値：なし

リスト 10: キーイベントの報告と報告終了イベントの送信（カーネルソースコード /drivers/input/input.h）

```
1 static inline void input_sync(struct input_dev *dev)
2 {
3     input_event(dev, EV_SYN, SYN_REPORT, 0);
4 }
5
6 static inline void input_report_key(struct input_dev *dev, unsigned int code, int value)
7 {
8     input_event(dev, EV_KEY, code, !!value);
9 }
```

入力サブシステムは、異なる入力イベントに対して異なる関数インターフェイスを提供しています。これらの関数インターフェイスは、input_event 関数の単純なラッパーに過ぎません。input_report_key はキーイベントを報告するために使用され、input_sync は同期信号を送信して報告が終了したことを示します。

15.2 入力サブシステム実験

このセクションでは、キー（または通常の GPIO）を例にして、入力サブシステムの具体的な使用方法を紹介します。プログラミングの考え方は、第一に入力デバイスをポーリングする方法です。これは GPIO に基づいており、IRQ にマッピングされていません。カーネルは GPIO をポーリングし、GPIO の電位を取得して、GPIO の状態に応じてキーの押下とリリースを判断します。第二に、ドライバプログラムが GPIO 割り込みを使用して、入力イベントをカーネルに送信する方法です。割り込みが発生した際に GPIO を検出し、キーが押されたか放されたかを判断します。

株式会社日昇テクノロジー

この章では、割り込みを使用した方法を採用し、ソースコードを組み合わせて説明します（対応するソースコードとデバイスツリープラグインは「/linux_driver/input_sub_system」ディレクトリにあります）。

15.2.1 デバイスツリープラグインの実装

デバイスツリープラグインは、前章の「割り込み実験」で使用されたデバイスツリープラグインとほぼ同じですが、割り込みタイプを「立ち上がりエッジと立ち下がりエッジの両方でトリガーする」に変更するだけです。変更部分は以下の通りです。

リスト 11: キー（GPIO）デバイスツリープラグインの記述

```
1 /dts-v1/;
2 /plugin/;
3
4 #include <dt-bindings/gpio/gpio.h>
5 #include <dt-bindings/pinctrl/rockchip.h>
6 #include <dt-bindings/interrupt-controller/irq.h>
7
8 &{/} {
9     input_button: input_button {
10        status = "okay";
11        compatible = "input_button";
12        button-gpios = <& GPIO1 RK_PB0 GPIO_ACTIVE_HIGH>; // 実際のボードに応じてピンを変更できます。ここでは GPIO1_B0 を例としています
13        pinctrl-names = "default";
14        pinctrl-0 = <&input_button_pin>;
```

株式会社日昇テクノロジー

```
15        interrupt-parent = <& GPIO1>;
16        interrupts = <RK_PB0 IRQ_TYPE_EDGE_BOTH>;
17    };
18 };
19
20 &{/pinctrl} {
21    pinctrl_button {
22        input_button_pin: input_button_pin {
23            rockchip,pins = <0 RK_PB0 RK_FUNC_GPIO &pcfg_pull_none>;
24        };
25    };
26 };
```

- button-gpios: GPIO を指定し、後でドライバが gpiod_get を使用して GPIO を取得します。

変更内容はシンプルで、割り込みのトリガー方式を両エッジトリガーに変更しただけで、他のデバイスツリーの内容は前章の割り込み実験と同じです。

15.2.2 ドライバプログラムの実装

15.2.2.1 ドライバのエントリ関数

リスト 12: ドライバエントリ関数

```
1 static int button_probe(struct platform_device *pdev)
2 {
3    struct button_data *priv;
```

株式会社日昇テクノロジー

```
4    struct gpio_desc *gpiod;
5    struct input_dev *i_dev;
6    int ret;
7
8    pr_info("button_probe¥n");
9    priv = devm_kzalloc(&pdev->dev, sizeof(*priv), GFP_KERNEL);
10    if (!priv)
11        return -ENOMEM;
12
13    i_dev = input_allocate_device();
14    if (!i_dev)
15        return -ENOMEM;
16
17    i_dev->open = btn_open;
18    i_dev->close = btn_close;
19    i_dev->name = "key input";
20    i_dev->dev.parent = &pdev->dev;
21    priv->button_input_dev = i_dev;
22    priv->pdev = pdev;
23
24    set_bit(EV_KEY, i_dev->evbit); /* 使用する入力イベントタイプを設定 */
25    set_bit(BTN_0, i_dev->keybit); /* イベント値、ボタン 0 を設定 */
```

株式会社日昇テクノロジー

```
26
27    gpiod = gpiod_get(&pdev->dev, "button", GPIOD_IN); /* GPIO を取得し、入力として設定 */
28    if (IS_ERR(gpiod))
29        return -ENODEV;
30
31    /* IRQ を取得 */
32    priv->irq = gpiod_to_irq(gpiod);
33    priv->button_input_gpiod = gpiod;
34
35    /* input デバイスを登録 */
36    ret = input_register_device(priv->button_input_dev);
37    if (ret) {
38        pr_err("Failed to register input device¥n");
39        goto err_input;
40    }
41
42    /* プラットフォームドライバデータを設定 */
43    platform_set_drvdata(pdev, priv);
44
45    /* GPIO 割り込みを要求 */
46    ret = request_any_context_irq(priv->irq, button_input_irq_handler, IRQF_TRIGGER_FALLING |
IRQF_TRIGGER_RISING, "input-button", priv);
```

株式会社日昇テクノロジー

```
47    if (ret < 0) {
48        dev_err(&pdev->dev, "GPIO 割り込み要求失敗¥n");
49        goto err_btn;
50    }
51
52    return 0;
53
54 err_btn:
55    gpiod_put(priv->button_input_gpiod);
56 err_input:
57    input_free_device(priv->button_input_dev);
58    return ret;
59 }
```

ドライバエントリ関数は基本的な初期化作業を完了し、コードの各部分を以下に説明します：

- 第 8 行：devm_kzalloc を使用してメモリを割り当てます。この関数で割り当てられたメモリは、デバイスがアンロードされる際に自動的に解放されます。

- 第 12 行：input_allocate_device()を使用して入力サブシステムの構造体を割り当てます。割り当てられた`input_dev`構造体は、入力デバイスを表します。

- 第 16-24 行：定義された button_data 構造体を埋め、入力デバイスの名前、関数などを設定し、set_bit 関数を使用して入力イベントタイプと値を設定します。input_dev のパラメータは多いですが、最も重要なのはイベントタイプとイベントに対応するコードです。evbit の各ビットはイベントタイプを表し、1 の場合はサポートされ、0 の場合はサポートされません。例えば、"キー"イベントをサポートする場合は、EV_KEY(0x01 に等しい)のビットを 1 にします。カーネルは BIT_MASK マクロを

株式会社日昇テクノロジー

提供して、特定の"イベント"を有効にするのに役立ちます。

- 第 26 行：gpiod_get 関数を使用して GPIO を取得し、入力として設定します。

リスト 13: gpiod_get 関数（カーネルソースコード/drivers/gpio/gpiolib.c）

```
1 struct gpio_desc *__must_check gpiod_get(struct device *dev, const char *con_id
2
3                                  , enum gpiod_flags flags)
4 {
5     return gpiod_get_index(dev, con_id, 0, flags);
6 }
```

gpiod_get 関数は、dev デバイスの con_id の 0 番目のピン情報を取得し、flags で初期化します。パラメータは以下の通りです。

パラメータ：

- dev device pointer, からピン情報を取得します。

- con_id はピンのグループ名です（プレフィックスは含まれません）。例えば、ピンのグループ名が button-gpios なら、con_id は"button"です。

- flags は初期化のフラグで、GPIO が入力か出力か、出力が高か低かを設定できます。フラグの指定については以下の通りです：

株式会社日昇テクノロジー

リスト 14: カーネルソースコード/include/gpio/consumer.h

```
1 enum gpiod_flags {
2 GPIOD_ASIS = 0, /* GPIO を初期化しない、方向は他の関数で設定する必要がある*/
3 GPIOD_IN = GPIOD_FLAGS_BIT_DIR_SET, /* GPIO ピンを入力として初期化*/
4 GPIOD_OUT_LOW = GPIOD_FLAGS_BIT_DIR_SET | GPIOD_FLAGS_BIT_DIR_OUT, /* GPIO ピ
ンを出力として初期化、出力は 0*/
5 GPIOD_OUT_HIGH = GPIOD_FLAGS_BIT_DIR_SET | GPIOD_FLAGS_BIT_DIR_OUT |
GPIOD_FLAGS_BIT_DIR_VAL,
6 GPIOD_OUT_LOW_OPEN_DRAIN = GPIOD_OUT_LOW | GPIOD_FLAGS_BIT_OPEN_DRAIN, /*
GPIO ピンをオープンドレイン出力として初期化、出力は 0*/
7 GPIOD_OUT_HIGH_OPEN_DRAIN = GPIOD_OUT_HIGH | GPIOD_FLAGS_BIT_OPEN_DRAIN,
8 };
```

戻り値：gpio_desc の構造体ポインタ、失敗した場合は-ENOENT。

15.2.2.2 ドライバの終了関数

終了関数は主にドライバが終了する前のクリーンアップを行います。非常にシンプルで、コードは以下の通りです：

リスト 15: ドライバの終了関数

```
1 static int button_remove(struct platform_device *pdev)
2 {
3 struct button_data *priv;
4 priv = platform_get_drvdata(pdev);
```

株式会社日昇テクノロジー

```
5

6 input_unregister_device(priv->button_input_dev);

7 input_free_device(priv->button_input_dev);

8 free_irq(priv->irq, priv);

9 gpiod_put(priv->button_input_gpiod);

10 return 0;

11

12 }
```

- 6 行目：input デバイスを登録解除する

- 7 行目：input デバイスを解放する

- 8-9 行目：irq と要求された gpio を解放する

15.2.2.3 割り込みサービス関数

割り込みサービス関数では、ボタンの入力ピンの状態を読み取り、ボタンが押されたか放されたかを判断します。コードは以下の通りです。

リスト 16: ボタンの割り込み処理関数

```
1 static irqreturn_t button_input_irq_hander(int irq, void *dev_id)

2 {

3 struct button_data *priv = dev_id;

4 int button_status;

5

6 /* ボタンのピンの電位を読み取り、読み取った結果に基づいてボタンの状態を入力する*/

7 button_status = (gpiod_get_value(priv->button_input_gpiod) & 1);
```

株式会社日昇テクノロジー

```
8 if(button_status)
9 {
10 input_report_key(priv->button_input_dev, BTN_0, 1);
11 input_sync(priv->button_input_dev);
12 }
13 else
14 {
15 input_report_key(priv->button_input_dev, BTN_0, 0);
16 input_sync(priv->button_input_dev);
17 }
18
19 return IRQ_HANDLED;
20 }
```

- 7 行目：ボタンに対応するピンの電位を読み取る。

- 8-17 行目：ボタンのピンの状態に基づいてシステムにボタンイベントを報告する。

15.2.3 テストアプリケーションの実装

テストアプリケーションでは、ボタンのキー値を読み取り、ボタンの状態をプリントします。具体的なコードは以下の通りです。

リスト 17: テストアプリケーションの実装

```
1 struct input_event button_input_event;
2
3 main(int argc, char *argv[])
```

株式会社日昇テクノロジー

```
4 {

5 int error = -20;

6

7 /* 実際のデバイスファイルに応じて変更が必要な、開くべきファイル/dev/input/event4 */

8 int fd = open("/dev/input/event4", O_RDONLY);

9 if (fd < 0)

10 {

11 printf("open file : /dev/input/event1 error!¥n");

12 return -1;

13 }

14

15 printf("wait button down... ¥n");

16

17 do

18 {

19 /* ボタンの状態を読み取る*/

20 error = read(fd, &button_input_event, sizeof(button_input_event));

21 if (error < 0)

22 {

23 printf("read file error! ¥n");

24 }

25 /* ボタンの状態を判断し、プリントする*/
```

株式会社日昇テクノロジー

```
26 if((button_input_event.type == 1) && (button_input_event.code == 0x100))
27 {
28 if(button_input_event.value == 0)
29 {
30 printf("button up¥n");
31 }
32 else if(button_input_event.value == 1)
33 {
34 printf("button down¥n");
35 }
36 }
37 } while (1);
38
39 printf("button Down !¥n");
40
41 /* ファイルを閉じる*/
42 error = close(fd);
43 if (error < 0)
44 {
45 printf("close file error! ¥n");
46 }
47 return 0;
48 }
```

株式会社日昇テクノロジー

- 1 行目：input_event 型の構造体変数を宣言する。これは、この章の冒頭で話したように、すべての入力デバイスがイベントの形で情報を報告するということです。

- 8 行目：ここで開くファイル**/dev/input/event1**は、入力サブシステムによって生成された入力デバイス、つまり使用するボタンです。実際のデバイスファイルに応じて変更する必要があります。

- 21 行目：ボタン情報を読み取り、入力イベントが報告されるまで read 関数は待機します。

- 27-37 行目：読み取った情報に基づいてボタンの状態を判断します。

テストアプリケーションの内容は非常にシンプルで、基本的にはファイルを開いて、状態を読み取り、状態を判断してから状態をプリントするという流れです。

15.2.4 実験の準備

ボード上の一部の GPIO はシステムによって使用されている可能性があります。ピンが使用中の場合、デバイスツリーは再びロードできないかもしれませんし、ドライバーでは対応するリソースを再度要求できないかもしれません。例えば、「Device or resource busy」やコードの実行がフリーズするなどの現象が発生する場合、使用する GPIO が他のドライバーに使用されていないことを確認する必要があります。

Permission denied やそれに類似するメッセージが表示された場合は、ユーザー権限に注意してください。ほとんどのハードウェア外部デバイスの操作機能は、root ユーザー権限が必要です。簡単な解決策は、コマンド実行前に sudo を追加するか、root ユーザーでプログラムを実行することです。

カーネルディレクトリ/arch/arm64/boot/dts/rockchip/overlays の Makefile を編集し、編集済みのデバイスツリープラグイン(本章のデバイスツリープラグインソースコードは lubancat-button-input-overlay.dts)を追加し、Makefile と同じディレクトリレベルにデバイスツリープラグインファイルを配置して、デバイスツリープラグインのコンパイルを行います。

カーネルのルートディレクトリで以下のコマンドを実行します：

株式会社日昇テクノロジー

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- lubancat4_defconfig //lubancat4 を例に

make ARCH=arm64 -j4 CROSS_COMPILE=aarch64-linux-gnu- dtbs
```

生成された.dtbo は、カーネルのルートディレクトリにある「arch/arm64/boot/dts/rockchip/overlays」ディレクトリにあります。

この章のデバイスツリープラグインは「lubancat-button-input-overlay.dts」で、コンパイル後、カーネルソースコード/arch/arm64/boot/dts/rockchip/overlays ディレクトリに同名の lubancat-button-input-overlay.dtbo ファイルが生成され、次のステップはそれをシステムにロードすることです。

15.2.4.1 デバイスツリープラグインファイルの追加

前のセクションで lubancat-button-input-overlay.dtbo をコンパイルしました。このファイルはシステムに動的にロードすることができます。lubancat4 ボードの uboot でデバイスツリープラグインをロードする例を見てみましょう。詳細は環境設定の章を参照してください。

まず、コンパイルしたデバイスツリープラグインファイルを開発ボードにアップロードします。uboot で書かれたデバイスツリープラグインをロードするには、2 つの簡単なステップだけです：

- 1、必要な.dtbo ファイルをボードの/boot/dtb/overlays/ディレクトリに置きます。
- 2、対応するデバイスツリープラグインのロード設定を uEnv.txt 設定ファイルに書き込みます。システム起動時に uEnv.txt からロードするデバイスツリープラグインを自動的に読み取ります。

  「/boot/uEnv/」ディレクトリにある uEnv.txt ファイルを開き、デバイスツリープラグインを uEnv.txt に書き込みます。vim や nano エディタを使用してファイルを開き、「dtoverlay=<デバイスツリープラグインパス>」という形式で書きます。

追加後、開発ボードを再起動し、コマンド ls /proc/device-tree/を使用して input_button ディレクトリがあるかどうかを確認します。あれば、ロードに成功しています。

株式会社日昇テクノロジー

15.2.4.2 ドライバプログラムとテストプログラムのコンパイル

このセクションの実験で使用する Makefile は以下のとおりです：

リスト 18: Makefile（../linux_driver/button_interrupt/interrupt に位置）

```
1 KERNEL_DIR=../../kernel/

2

3 ARCH=arm64

4 CROSS_COMPILE=aarch64-linux-gnu-

5 export ARCH CROSS_COMPIL

6

7 obj-m := input_sub_system.o

8 out = test_app

9

10 all:

11 $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) modules
```

付属のドライバーコードのルートディレクトリをカーネルと同じディレクトリレベルに配置し、その後、input_sub_system サブディレクトリに入り、ドライバーディレクトリ内で以下のコマンドを入力してドライバーモジュールとテストプログラムをコンパイルします：

```
make
```

15.2.5 ドライバの検証とロード

コンパイルしたドライバ、アプリケーション、デバイスツリープラグインを開発ボードにコピーします。この部分については、前のセクションで詳しく説明されていますので、ここでは詳細には触れません。

再起動後、insmod input_sub_system.ko コマンドを使用してドライバをロードします。デバイスがマッチ

株式会社日昇テクノロジー

すると、「/dev/input」ディレクトリにデバイスファイルが生成されます。この実験では

「/dev/input/event4」が生成されます。

```
cat@lubancat:~$ sudo insmod input_sub_system.ko
cat@lubancat:~$ ls /dev/in
initctl  input/
cat@lubancat:~$ ls -al /dev/input/by-path/
total 0
drwxr-xr-x 2 root root 140 Sep 20 10:06 .
drwxr-xr-x 3 root root 160 Sep 20 10:06 ..
lrwxrwxrwx 1 root root   9 Apr 21 20:54 platform-adc-keys-event -> ../event2
lrwxrwxrwx 1 root root   9 Apr 21 20:54 platform-fdd40000.i2c-platform-rk805-pwrkey-event -> ../event1
lrwxrwxrwx 1 root root   9 Apr 21 20:54 platform-fe6e0030.pwm-event -> ../event0
lrwxrwxrwx 1 root root   9 Sep 20 10:06 platform-input_button-event -> ../event4
lrwxrwxrwx 1 root root   9 Apr 21 20:54 platform-rk-headset-event -> ../event3
cat@lubancat:~$
```

cat /proc/interrupts コマンドを使用して、ドライバが要求した割り込みを確認できます：

```
 76:        259          0          0          0     GICv3 150 Level     debug
 78:          0          0          0          0     GICv3 202 Level     xhci-hcd:usb5
 79:          0          0          0          0     gpio0   3 Level     rk817
 80:          0          0          0          0     rk817   0 Edge      rk805_pwrkey_fall
 81:          0          0          0          0     rk817   1 Edge      rk805_pwrkey_rise
 85:          0          0          0          0     rk817   5 Edge      RTC alarm
105:          0          0          0          0     gpio0   5 Level     headset input
106:          0          0          0          0     gpio0  23 Edge      input-button
IPI0:     26696     115298      26837      33189       Rescheduling interrupts
IPI1:     45926      54049      55365      37709       Function call interrupts
IPI2:         0          0          0          0       CPU stop interrupts
IPI3:         0          0          0          0       CPU stop (for crash dump) interrupts
IPI4:      4275       4474       3187       3223       Timer broadcast interrupts
IPI5:     23831      83049      44495      22531       IRQ work interrupts
IPI6:         0          0          0          0       CPU wake-up interrupts
Err:          0
```

udevadm info /dev/input/event4 コマンドを使用してデバイス情報を確認します：

```
cat@lubancat:~$ udevadm info /dev/input/event4
P: /devices/platform/input_button/input/input5/event4
N: input/event4
L: 0
S: input/by-path/platform-input_button-event
E: DEVPATH=/devices/platform/input_button/input/input5/event4
E: DEVNAME=/dev/input/event4
E: MAJOR=13
E: MINOR=68
E: SUBSYSTEM=input
E: USEC_INITIALIZED=46468293842
E: ID_INPUT=1
E: ID_PATH=platform-input_button
E: ID_PATH_TAG=platform-input_button
E: DEVLINKS=/dev/input/by-path/platform-input_button-event
```

ドライバが正常にロードされた後、テストアプリケーションコマンド「./test_app」を直接実行してください。プログラムを実行した後、ボタンを押す（またはピンの電圧レベルを下げる/上げる）。ターミナルにはボタンの状態が表示されます。

```
cat@lubancat:~$ sudo ./test_app
wait button down...
button down
button up
button down
button up
button down
button up
```

また、以下の evtest コマンドを使用してテストすることもできます。コマンドを使用した後にボタンを押すか、ピンの電圧レベルを上げる/下げると：

sudo evtest /dev/input/event4 /* /dev/input/event4 は、自分がロードしたドライバーによって追加されたデバイスファイルに基づいています*/

```
^Ccat@lubancat:~$ sudo evtest /dev/input/event4
Input driver version is 1.0.1
Input device ID: bus 0x0 vendor 0x0 product 0x0 version 0x0
Input device name: "key input"
Supported events:
  Event type 0 (EV_SYN)
  Event type 1 (EV_KEY)
    Event code 256 (BTN_0)
Properties:
Testing ... (interrupt to exit)
Event: time 1663640623.140909, type 1 (EV_KEY), code 256 (BTN_0), value 1
Event: time 1663640623.140909, -------------- SYN_REPORT ------------
Event: time 1663640623.143024, type 1 (EV_KEY), code 256 (BTN_0), value 0
Event: time 1663640623.143024, -------------- SYN_REPORT ------------
Event: time 1663640623.160824, type 1 (EV_KEY), code 256 (BTN_0), value 1
Event: time 1663640623.160824, -------------- SYN_REPORT ------------
Event: time 1663640623.163075, type 1 (EV_KEY), code 256 (BTN_0), value 0
Event: time 1663640623.163075, -------------- SYN_REPORT ------------
Event: time 1663640623.180783, type 1 (EV_KEY), code 256 (BTN_0), value 1
Event: time 1663640623.180783, -------------- SYN_REPORT ------------
Event: time 1663640623.183069, type 1 (EV_KEY), code 256 (BTN_0), value 0
Event: time 1663640623.183069, -------------- SYN_REPORT ------------
Event: time 1663640623.200704, type 1 (EV_KEY), code 256 (BTN_0), value 1
Event: time 1663640623.200704, -------------- SYN_REPORT ------------
```

キーの種類　　伝送されたキー値

第 16 章 PWM 子システム

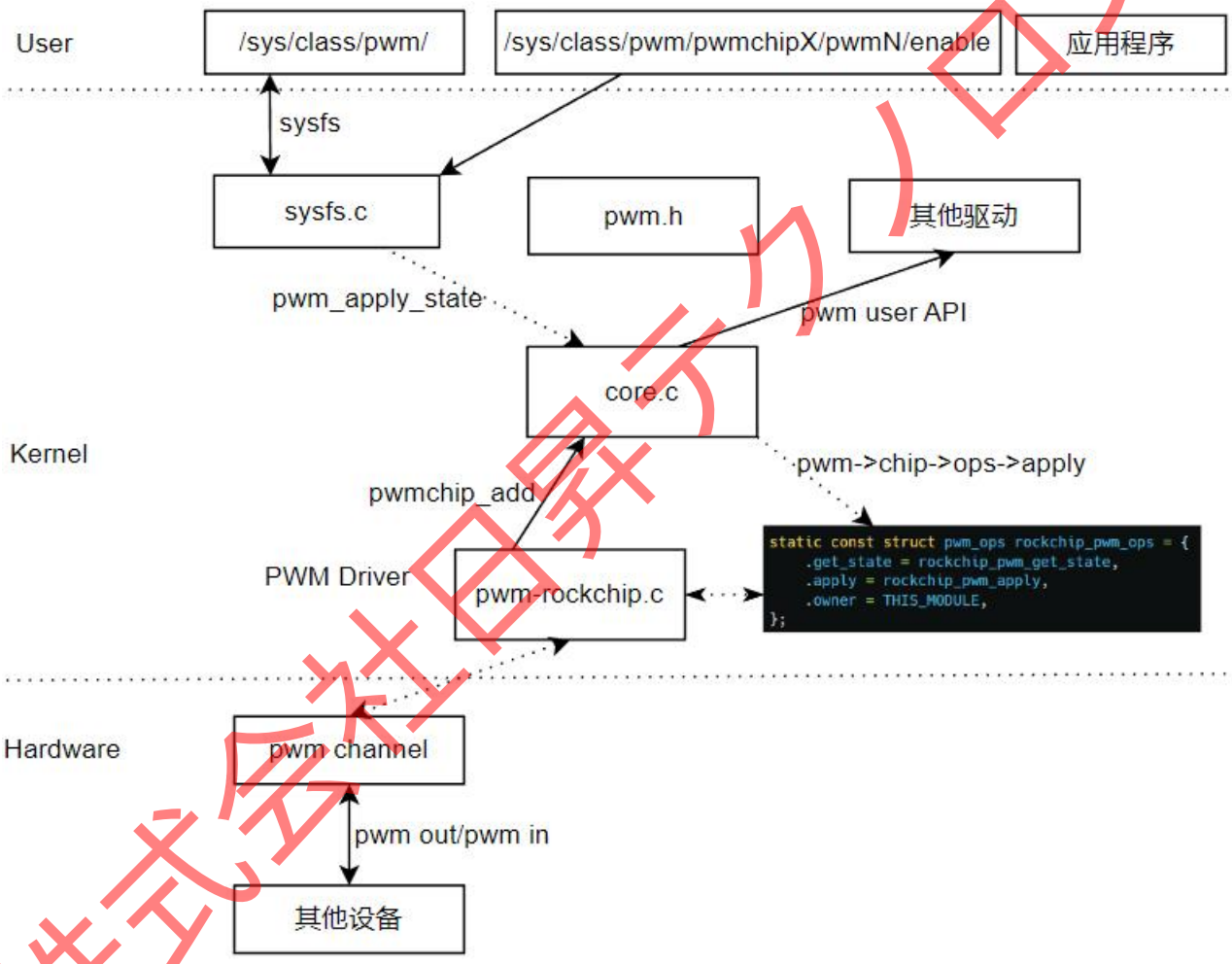
PWM 子システムは PWM 波の出力を管理するために使用され、これまでに学習した他の子システムと同様に、PWM の具体的な実装コードはチップメーカーによって提供され、デフォルトでカーネルにコンパイルされています。しかし、カーネル（PWM 子システム）が提供するいくつかのインターフェース関数を使用して、具体的な機能を実現することができます。例えば、PWM 波を使用してディスプレイのバックライトを制御したり、無源蜂鳴器、サーボモーター、電圧調整などを制御することができます。

株式会社日昇テクノロジー

PWM 子システムの機能は単一であり、単独で使用されることは少ないですが、この章では簡単な PWM 子システムドライバを通じて PWM 子システムについて簡単に紹介し、LED ライト実験（または PWM 波形出力実験）をテストします。

16.1 PWM 子システムの紹介

PWM（Pulse Width Modulation）、パルス幅変調。カーネル内の PWM ドライバは比較的シンプルですが、基本的なフレームワークは完全です。PWM フレームワークの簡単な紹介は以下の通りです：



カーネルの PWM コアは、実際の PWM コントローラドライバに対して、pwm_chip、SoC メーカーのコントローラドライバに対し、構造体を登録し、private_data を設定し、pwm_ops 操作をインスタンス化し、具体的な関数を記述するだけで済みます。他のドライバやユーザープログラムには統一されたインターフェースを提供し、pwm_device を介して pwm_chip に関連付けられ、他のドライバやユーザープログラムはこのインターフェースを通じて pwm_device 構造体を操作します。

株式会社日昇テクノロジー

PWM コントローラドライバは SoC メーカーがすでに作成しており、行うべきことは、デバイスツリー（またはデバイスツリープラグイン）内でコントローラノードを有効にし、PWM デバイスノードを記述し、その後、カーネル PWM が提供するインターフェースを呼び出して、PWM ドライバ制御を実現することです。完全なプログラムは、付属のソースコード「linux_driver/pwm_sub_system」ディレクトリ下のファイルを参照してください。

16.1.1 PWM デバイス構造体

PWM を提供するドライバを使用する場合、ドライバ内で pwm_device 構造体は PWM デバイスを表します。構造体のプロトタイプは以下のとおりです：

リスト 1: pwm 関連構造体（カーネルソースコード/include/linux/pwm.h）

```
1 struct pwm_device {
2 const char *label;
3 unsigned long flags;
4 unsigned int hwpwm;
5 unsigned int pwm;
6 struct pwm_chip *chip;
7 void *chip_data;
8
9 struct pwm_args args;
10 struct pwm_state state;
11 };
12
13 struct pwm_args {
```

株式会社日昇テクノロジー

```
14 u64 period;
15 enum pwm_polarity polarity;
16 };
17
18 struct pwm_state {
19 u64 period;
20 u64 duty_cycle;
21 enum pwm_polarity polarity;
22 enum pwm_output_type output_type;
23 struct pwm_output_pattern *output_pattern;
24 #ifdef CONFIG_PWM_ROCKCHIP_ONESHOT
25 u64 oneshot_count;
26 #endif /* CONFIG_PWM_ROCKCHIP_ONESHOT */
27 bool enabled;
28 };
```

PWM 関連構造体のいくつかの重要なパラメータは以下の通りです：

- label: PWM デバイス名

- flags: 関連フラグ

- pwm: グローバルな PWM デバイスインデックス

- chip: PWM コントローラの抽象

- state: 現在の PWM チャネルの状態

- period: PWM の周期を設定します。ここでは単位はナノ秒(ns)です。例えば、1MHz の PWM 波を出力する場合、period は 1000 に設定する必要があります。

株式会社日昇テクノロジー

- duty_cycle: 占有率を設定します。通常の出力極性の場合、このパラメータは PWM 波の周期内の高レベル持続時間を指定し、単位は ns です。明らかに duty_cycle は period より大きくなることはできません。出力反転を設定する場合、このパラメータは周期内の低レベル持続時間を指定するために使用されます。

- polarity: 出力極性を指定するパラメータで、PWM 出力が反転するかどうかを指定します。これは列挙型で、以下の通りです。

リスト 2: pwm_polarity 列挙型

```
1 enum pwm_polarity {
2 PWM_POLARITY_NORMAL,
3 PWM_POLARITY_INVERSED,
4 };
```

- PWM_POLARITY_NORMAL: 正常モードを表し、反転しません。

- PWM_POLARITY_INVERSED: 出力を反転させます。

16.1.2 PWM の申請と解放関数

PWM を使用する前には申請が必要で、使用しない時は速やかに解放する必要があります。申請と解放の関数は多く、4 つのグループに分かれています。以下に紹介します：

リスト 3: 第一グループの PWM 申請と解放関数

```
1 struct pwm_device *pwm_request(int pwm, const char *label);
2 void pwm_free(struct pwm_device *pwm);
```

これは旧システムで使用された PWM の申請と解放の関数で、現在は使用されていません。認識のみで良いです。

リスト 4: 第二グループの PWM 申請と解放関数

株式会社日昇テクノロジー

```
1 struct pwm_device *pwm_get(struct device *dev, const char *con_id);

2 void pwm_put(struct pwm_device *pwm);
```

- pwm_get: PWM の申請関数

- pwm_put: PWM の解放関数

- パラメータ dev: どのデバイスから PWM を取得するか、カーネルは dev デバイスのデバイスツリーノード内でパラメータ「con_id」に基づいて検索し、判断基準は con_id がデバイスツリーノードの「pwm-names」と同じであることです。

- パラメータ con_id: デバイス内で PWM が一つしか使用されない場合、con_id パラメータを NULL に設定でき、デバイスツリーノードに「pwm-names」属性を設定する必要はありません。

• 戻り値：取得に成功した場合は取得した pwm を返します。失敗した場合は NULL を返します。

• pwm デバイスを使用しない場合は、pwm_put を使用して pwm を解放します。パラメータは pwm_get で取得した pwm_device 構造体型のポインタです。

リスト 5：第三グループの pwm の申請と解放関数

```
1 struct pwm_device *devm_pwm_get(struct device *dev, const char *con_id)

2 void devm_pwm_put(struct device *dev, struct pwm_device *pwm)
```

このグループの関数は、前のグループの関数のラッパーであり、使用方法は第二グループと同様ですが、ドライバが削除される際に申請した pwm を自動的に解放するという利点があります。

リスト 6：第四組の PWM の申請と解放関数

```
1 /*-----第四グループ-----*/

2 struct pwm_device *of_pwm_get(struct device_node *np, const char *con_id)

3 struct pwm_device *devm_of_pwm_get(struct device *dev, struct device_node *np, const char *con_id)
```

• of_pwm_get 関数：指定されたデバイスツリーノードから PWM を取得します。

• パラメータ np は、どのデバイスノードから PWM を取得するかを指定します。

株式会社日昇テクノロジー

• パラメータ con_id の役割は、前のグループの関数と同じです。

• 戻り値は、取得した PWM で、失敗した場合は NULL を返します。

devm_of_pwm_get 関数は of_pwm_get 関数のラッパーであり、3 つのパラメータを持ちます。パラメータ dev は、どのデバイスが PWM を取得するかを指定し、他の 2 つのパラメータは of_pwm_get 関数と同じです。この関数の利点は、ドライバが削除される前に申請した pwm を自動的に解放することです。

16.1.3 pwm の設定関数と有効/無効化関数

取得に成功した後、pwm の周波数とデューティサイクルを設定し、出力を有効にするだけで、設定されたピンで PWM 波を出力することができます。関数は非常にシンプルで、以下のようになります。

リスト 7：pwm の設定関数と有効化/無効化関数

```
1 int pwm_config(struct pwm_device *pwm, int duty_ns, int period_ns)
2 int pwm_set_polarity(struct pwm_device *pwm, enum pwm_polarity polarity)
3 int pwm_enable(struct pwm_device *pwm)
4 void pwm_disable(struct pwm_device *pwm)
```

• pwm_config 関数は、PWM の周波数とデューティサイクルを設定するために使用されます。ここでは、PWM の 1 周期の時間と高レベル時間を設定することによって、PWM の周波数とデューティサイクルを設定します。単位はすべて ns です。

• pwm_set_polarity 関数は、PWM の極性を設定するために使用されます。ここで、PWM を負の極性に設定した場合、pwm_config 関数のパラメータ duty_ns は、周期内の低レベル時間を設定します。

• pwm_enable と pwm_disable 関数を使用して、pwm を有効化および無効化します。

16.2 pwm 出力実験

PWM 子システムは、単独で使用されることが少ないため、ここでは極めてシンプルな例示ドライバプログラムを用いて、PWM 子システムの使用方法を紹介します。ここでは、lubancat4 ボードを例に、ハート

株式会社日昇テクノロジー

ビート LED ピン GPIO1_B5 を PWM 出力として使用します（他の IO も使用可能）。ドライバ内で周期とデューティサイクルの設定を変更することで、異なる明るさのライトを観察できますし、オシロスコープを使用して波形を確認することもできます。

lubancat のボードは類似していますが、具体的に使用可能な PWM 出力ピンを確認する必要があります。

サンプルプログラムは主に 2 つの部分を含みます：一つ目は、対応するデバイスツリーノードを追加すること（ここではデバイスツリープラグインを使用）、二つ目は、テストドライバプログラムを記述することです。

16.2.1 PWM 関連のデバイスツリープラグインを追加

まず、デバイスツリー内の PWM 関連の内容について簡単に紹介します。「RK3588.dtsi」ファイルを開き、「pwm0」を検索すると、以下のような内容が見つかります：

リスト 8：pwm0 ノード

```
1 pwm0: pwm@fdd70000 {
2 compatible = "rockchip,rk3568-pwm", "rockchip,rk3328-pwm";
3 reg = <0x0 0xfdd70000 0x0 0x10>;
4 #pwm-cells = <3>;
5 pinctrl-names = "active";
6 pinctrl-0 = <&pwm0m0_pins>;
7 clocks = <&pmucru CLK_PWM0>, <&pmucru PCLK_PWM0>;
8 clock-names = "pwm", "pclk";
9 status = "disabled";
10 };
```

株式会社日昇テクノロジー

リスト 9: rk3568-pinctrl.dtsi

```
1 pwm0 {
2 pwm0m0_pins: pwm0m0-pins {
3 rockchip,pins =
4 /* pwm0_m0 */
5 <0 RK_PB7 1 &pcfg_pull_none>;
6 };
7
8 pwm0m1_pins: pwm0m1-pins {
9 rockchip,pins =
10 /* pwm0_m1 */
11 <0 RK_PC7 2 &pcfg_pull_none>;
12 };
13 };
```

これは PWM ドライバに対応するデバイスツリーノードであり、PWM 子システムの制御ノードです。「RK3588-pinctrl.dtsi」には、pwm0m0_pins の記述があります。今回は pwm0m1_pins を使用するため、デバイスツリーの作成時に修正が必要です。

PWM を使用する際には、このデバイスツリーノードを参照し、いくつかの属性情報を追加する必要があります。以下のようになります：

リスト 10：pwm 属性情報

```
1 pwms = <&PWMn id period_ns PWM_POLARITY_INVERTED>;
2 pwm-names = "name";
```

- pwms：必須の属性です

株式会社日昇テクノロジー

• &PWMn：どの PWM を指定するか、RK3588.dtsi ファイルで定義されています。参照時には pwm1、pwm2 などの別名を使用できます

• id：PWM の id、通常は 0 に設定されます。

• PWM_POLARITY_INVERTED：オプションで、PWM の極性を示します。0 は正の極性、1 は反転極性です。

• period_ns：PWM 信号の周期を設定します。単位は ns です。

• pwm-names：PWM デバイスの名前を定義します。

本実験では、gpio デバイスツリープラグインのソースコードは以下の通りです：

リスト 11：デバイスツリープラグイン

```
1 /*
2 * Copyright (C) 2022 - All Rights Reserved by
3 * EmbedFire LubanCat
4 */
5 /dts-v1/;
6 /plugin/;
7
8 #include <dt-bindings/gpio/gpio.h>
9 #include <dt-bindings/pinctrl/rockchip.h>
10 #include <dt-bindings/clock/rk3568-cru.h>
11 #include <dt-bindings/pwm/pwm.h>
12
13 &pwm0 {
```

株式会社日昇テクノロジー

```
14 status = "okay";
15 pinctrl-names = "active";
16 pinctrl-0 = <&pwm0m1_pins>;
17 };
18
19 &{/} {
20 pwm_demo: pwm_demo {
21 status = "okay";
22 compatible = "pwm_demo";
23
24 back {
25 pwm-names = "pwm-demo";
26 pwms = <&pwm0 0 10000 1>;
27 duty_ns = <5000>;
28 };
29 };
30 };
```

• 13 行目、pwm0 の status を"okay"に設定し、この pwm0 コントローラを有効にします。

• 15 行目、pinctrl-0 を設定し、 GPIO1_C7 ピンを PWM 機能として使用します。

• 18-28 行目、このセクションの内容はルートノードに挿入され、pwm_demo ノードが定義されています。compatible は"pwm_demo"で、自分たちが記述したドライバとマッチするためのものです。このノードには"back"子ノードが含まれ、子ノード内で PWM 属性情報が定義されています。ここではPWM0 のチャネルを使用し、デューティサイクルを 5000、周期を 10000、反転極性に設定していま

株式会社日昇テクノロジー

す。

注意：このデバイスツリープラグインの一部内容は lubancat4 ボードのデバイスツリーの leds ノードと衝突する可能性があります。実験前にデバイスツリー内の leds の使用を無効にすること、つまり status を"disabled"に設定することを確認してください。

16.2.2 ドライバプログラムの実装

ドライバプログラムの具体的なコードは以下の通りです：

リスト 12：プラットフォームデバイスの登録

```
1 static const struct of_device_id of_pwm_leds_match[] = {
2 {.compatible = "pwm_demo"},
3 {},
4 };
5
6 static struct platform_driver pwm_demo_driver = {
7 .probe = pwm_demo_probe_new,
8 .remove = pwm_demo_remove,
9 .driver = {
10 .name = "pwm_demo",
11 .of_match_table = of_pwm_leds_match,
12 },
13 };
14
15 module_platform_driver(pwm_demo_driver);
```

株式会社日昇テクノロジー

• 1-4 行目、デバイスツリーノードのマッチ情報を設定します。

• 6-13 行目、platform_driver 構造体を埋めます。

• 15 行目、登録プラットフォームデバイスの方法を採用して、ドライバプログラムを登録します。

プラットフォームデバイスとデバイスノードが正常にマッチすると、デバイスツリーから情報を容易に取得できます。もちろん、デバイスツリーノードから情報を直接取得するために of 関数を使用する方法もあります。

.prob 関数内で、PWM を申請し、設定し、有効化する具体的なコードは以下の通りです：

リスト 13：prob 関数

```
1 static int pwm_demo_probe(struct platform_device *pdev)
2 {
3 int ret = 0;
4 struct device_node *child; // 子ノードを保存
5 struct device *dev = &pdev->dev;
6 printk("マッチ成功 ¥n");
7
8 /*-------------第一部分-----------------*/
9 child = of_get_next_child(dev->of_node, NULL);
10 if (child)
11 {
12 /*-------------第二部分-----------------*/
13 pwm_test = devm_of_pwm_get(dev, child, NULL);
14 if (IS_ERR(pwm_test))
```

株式会社日昇テクノロジー

```
15 {
16 printk(KERN_ERR" pwm_test、PWM の取得に失敗しました!!¥n");
17 return -1;
18 }
19 }
20 else
21 {
22 printk(KERN_ERR" pwm_test of_get_next_child に失敗しました!!¥n");
23 return -1;
24 }
25
26 /*--------------第三部分-----------------*/
27 pwm_config(pwm_test, 1000, 5000);
28 pwm_set_polarity(pwm_test, PWM_POLARITY_INVERSED);
29 pwm_enable(pwm_test);
30
31 return ret;
32 }
33
34 static int pwm_demo_remove(struct platform_device *pdev)
35 {
36 pwm_config(pwm_test, 0, 5000);
```

株式会社日昇テクノロジー

```
37 pwm_free(pwm_test);
38 return 0;
39 }
```

• 9 行目、子ノードを取得します。デバイスツリープラグインでは、PWM 関連の情報を pwm_demo の子ノードに保存しているため、ここで最初に子ノードを取得します。

• 13 行目、子ノードの取得に成功した後、devm_of_pwm_get 関数を使用して PWM を取得します。ノード内で PWM が 1 つしかないため、最後のパラメータを NULL に直接設定します。

• 27-29 行目、pwm_config、pwm_set_polarity、pwm_enable 関数を順に呼び出して PWM を設定し、出力極性を設定し、PWM 出力を有効にします。ここでの極性設定は負の極性であり、そのため pwm_config 関数では周期(period_ns)を 5000、有効時間(duty_ns)を 1000 として設定しています。

16.2.3 実験準備

ボード上の一部の GPIO はシステムによって占有されている可能性があり、ピンが占用されると、デバイスツリーが再びロードされないか、ドライバー内で対応するリソースを再度要求できない可能性があります。たとえば、「Device or resource busy」やコードの実行がフリーズするなどの現象が発生する場合、使用する GPIO が他のドライバーによって使用されていないことを確認する必要があります。

「Permission denied」や類似のメッセージが表示された場合は、ユーザー権限に注意してください。ほとんどのハードウェア外部デバイスの操作機能は、root ユーザー権限が必要です。簡単な解決策は、コマンドの実行前に sudo を追加するか、root ユーザーでプログラムを実行することです。

16.2.3.1 カーネルツールを使用してデバイスツリープラグインをコンパイル

デバイスツリープラグインはデバイスツリーと同様に DTC ツールを使用してコンパイルされますが、デバイスツリーは.dtb にコンパイルされ、デバイスツリープラグインは.dtbo にコンパイルする必要があ

株式会社日昇テクノロジー

ります。DTC コンパイルコマンドを使用して.dtbo をコンパイルすることができますが、これは比較的煩雑でエラーが発生しやすいです。

カーネルディレクトリ/arch/arm/boot/dts/overlays の Makefile を編集し、編集したデバイスツリープラグインを追加し、Makefile と同じディレクトリレベルにデバイスツリープラグインファイルを配置してデバイスツリープラグインのコンパイルを行います。詳細は環境構築の章を参照してください。

カーネルのルートディレクトリで以下のコマンドを実行するだけです：

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- lubancat4_defconfig //lubancat4 を例に
make ARCH=arm64 -j4 CROSS_COMPILE=aarch64-linux-gnu- dtbs
```

生成された.dtbo はカーネルのルートディレクトリ下の「arch/arm64/boot/dts/rockchip/overlays」ディレクトリにあります。

この章の PWM 子システムのデバイスツリープラグインは「lubancat-pwm0-m1-demo-overlay.dts」で、コンパイル後に「arch/arm64/boot/dts/rockchip/overlays」ディレクトリ下に同名の「lubancat-pwm0-m1-demo-overlay.dtbo」ファイルが生成されます。

.dtbo を取得したら、次のステップはそれをシステムにロードすることです。

16.2.3.2 デバイスツリープラグインファイルを追加

前のセクションで「lubancat-pwm0-m1-demo-overlay.dtbo」をコンパイルしました。このファイルはシステムに動的にロードすることができます。lubancat4 ボードの uboot でデバイスツリープラグインをロードする例を見てみましょう。詳細は環境構築の章を参照してください。

まず、コンパイルしたデバイスツリープラグインファイルを開発ボードにアップロードします。uboot で書かれたデバイスツリープラグインをロードするには、2 つの簡単なステップが必要です：

- 1、必要な.dtbo ファイルをボードの/boot/dtb/overlays/ディレクトリに入れます。
- 2、対応するデバイスツリープラグインのロード設定を uEnv.txt 設定ファイルに書き込み、システム

株式会社日昇テクノロジー

の起動時に自動的に uEnv.txt からロードするデバイスツリープラグインを読み取ります。lubancat4 を例に、「/boot/uEnv/」ディレクトリ下の uEnvLubancat4.txt ファイルを開き、デバイスツリープラグインを uEnvLubancat4.txt に書き込みます。vim や nano エディタを使用してファイルを開き、「dtoverlay=<デバイスツリープラグインのパス>」という形式で書き込みます。

追加後、開発ボードを再起動し、コマンド「ls /proc/device-tree/」を使用して「pwm_demo」ディレクトリが存在するかどうかを確認します。存在する場合は、ロードに成功しています。

16.2.3.3 ドライバプログラムとテストプログラムをコンパイル

リスト 14：Makefile（linux_driver/pwm_sub_system に位置）

```
1 KERNEL_DIR=../../kernel/

2

3 ARCH=arm64

4 CROSS_COMPILE=aarch64-linux-gnu-

5 export ARCH CROSS_COMPIL

6

7 obj-m := pwm_sub_system.o

8

9 all:

10 $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) modules

11

12 .PHONY:clean

13 clean:

14 $(MAKE) -C $(KERNEL_DIR) M=$(CURDIR) clean
```

株式会社日昇テクノロジー

付属のドライバーコード、例えば linux_driver/をカーネルと同じディレクトリレベルに配置し、pwm_sub_system ドライバディレクトリで以下のコマンドを入力してドライバーモジュールとテストプログラムをコンパイルします：

```
make
```

16.2.4 ダウンロードと検証

コンパイルしたドライバ、アプリケーション、デバイスツリープラグインを開発ボードにコピーします。この部分は以前の章で詳しく説明されていますので、ここでは詳細には触れません。

再起動後、直接「insmod」コマンドを使用してドライバをロードします。

以下のコマンドを使用すると、システムの現在の PWM 状態を確認できます：

```
cat /sys/kernel/debug/pwm
```

オシロスコープを使用すると、設定された PWM 波（例程の設定を変更しない場合、PWM 周波数は 200KHz、デューティサイクルは 80%）を確認できます。

第 17 章 I2C 子システム–mpu6050 ドライバ実験

I2C(Inter-Integrated Circuit)は同期型、半二重通信バスで、組み込みシステム内でよく使用され、シリアル EEPROM、RTC チップ、GPIO エクスパンダー、温度センサーなどを接続するのに使われます。この章では、オンボード MPU6050 を例にして、i2c ドライバプログラムの書き方を説明します。主に以下の 5 部分から成ります。

- 第一部分、i2c の基礎知識、i2c 物理バスと基本通信プロトコルの復習。
- 第二部分、Linux 下の i2c ドライバフレームワーク。
- 第三部分、i2c バスドライバコードの解析。

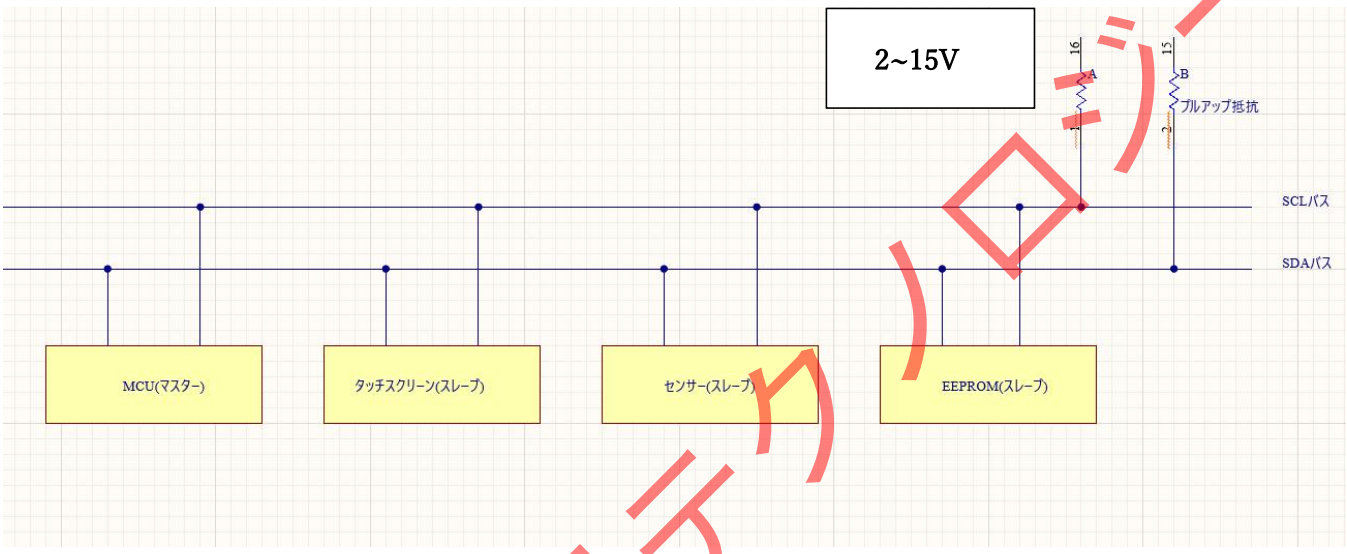
株式会社日昇テクノロジー

- 第四部分、i2c デバイスドライバのコア関数。

- 第五部分、MPU6050 ドライバおよびテストプログラム。

17.1 i2c の基礎知識

17.1.1 i2c 物理バス



上図に示すように、i2c は一主多従をサポートし、各デバイスは独立したアドレスを持ちます。標準モードの転送速度は 100kbit/s、高速モードは 400kbit/s です。バスはプルアップ抵抗を介して電源に接続されます。I2C デバイスがアイドル状態の時、高インピーダンスを出力し、すべてのデバイスがアイドルで高インピーダンスを出力すると、プルアップ抵抗がバスを高電圧に引っ張ります。

I2C 物理バスは、SCL と SDA の 2 本のバスラインを使用します。

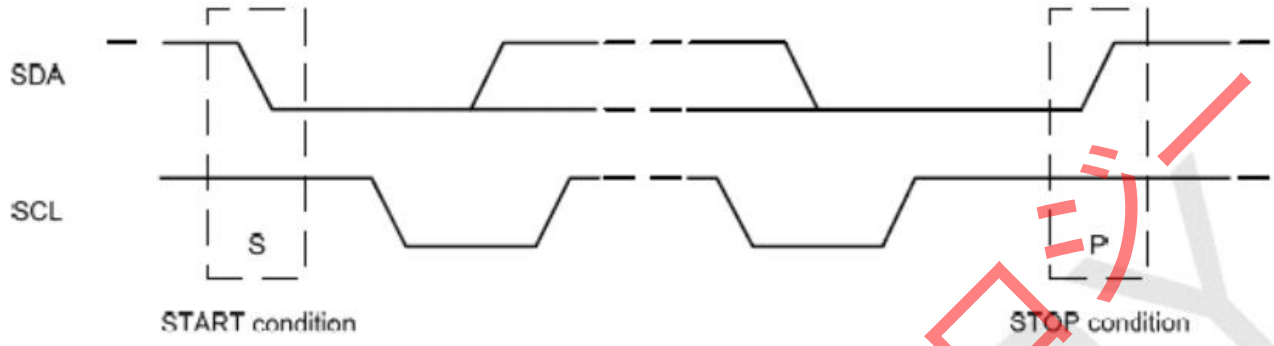
- SCL：クロックライン、データ送受信の同期

- SDA：データライン、具体的なデータの伝送

株式会社日昇テクノロジー

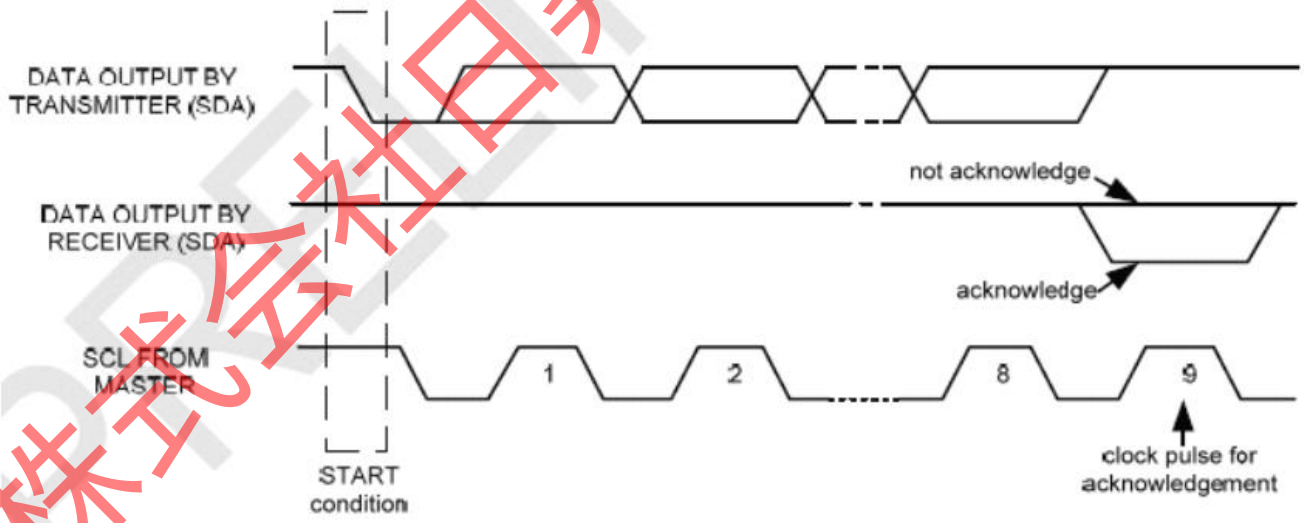
17.1.2 i2c 基本通信プロトコル

17.1.2.1 スタート信号(S)とストップ信号(P)



SCL ラインが高電圧の時に、SDA ラインが高から低への下降エッジで、伝送開始のシグナル(S)となります。メインデバイスが終了シグナル(P)を出すまで、バスの状態は常にビジーです。終了シグナル(P)は、SCL ラインが高電圧の時に、SDA ラインが低から高への上昇エッジです。

17.1.2.2 データ形式と応答信号(ACK/NACK)

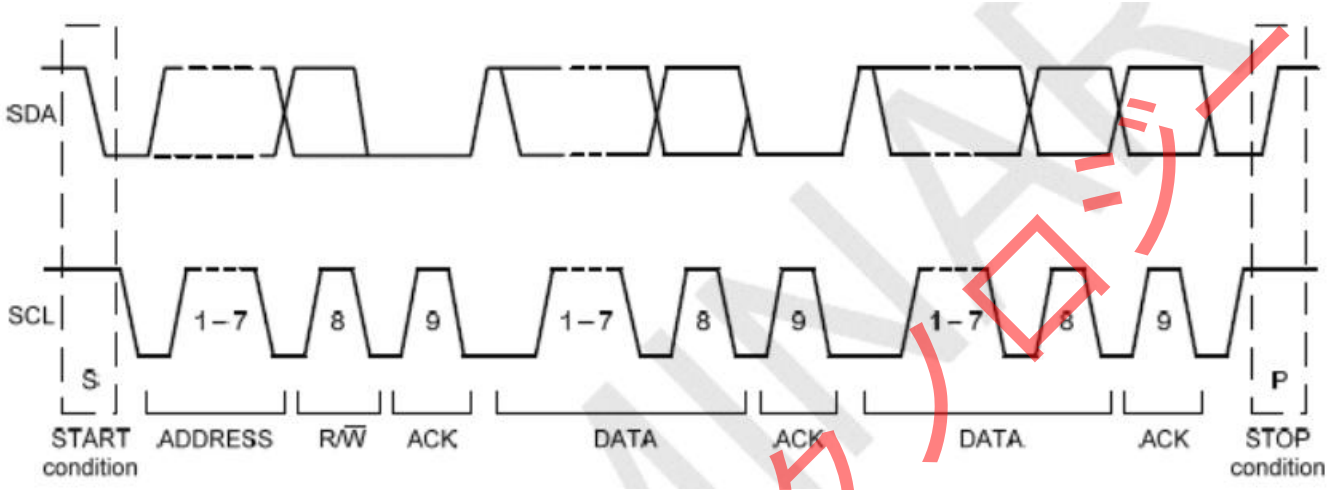


i2c のデータバイトは 8 ビットの長さで定義され、各伝送の総バイト数に制限はありませんが、各伝送には応答(ACK)シグナルが必要で、そのクロックはメインデバイスが提供し、実際の応答シグナルはスレーブデバイスが出します。クロックが高の時に、SDA の値を低く引っ張って保持することによって実現します。スレーブデバイスがビジーの場合、SCL を低電圧に保つことができ、これによりメインデバイスを

株式会社日昇テクノロジー

待機状態に強制します。スレーブデバイスがアイドルになり、クロックラインを解放すると、元のデータ転送が続行されます。

17.1.2.3 メインデバイスと



スタートシグナル(S)を出した後、メインデバイスは 7 ビットのスレーブアドレスを送信し、その後に第 8 ビットとして Read/Write ビットを続けます。R/W ビットは、メインデバイスがスレーブデバイスからデータを受け取るのか、それともスレーブデバイスにデータを書き込むのかを示します。その後、メインデバイスは SDA ラインを解放し、スレーブデバイスからの応答シグナル(ACK)を待ちます。各バイトの転送には応答ビットが続きます。応答が生成されると、スレーブデバイスは SDA ラインを低く引っ張り、SCL が高電圧の間、低く保持します。データ転送は常にストップシグナル(P)で終了し、通信ラインが解放されます。しかし、メインデバイスは終了シグナルを出さずに、別のスレーブデバイスを操作するためにリピートスタートシグナルを生成することもできます。すべての SDA シグナルの変更は、SCL クロックが低電圧の時に行われる必要がありますが、スタートとストップシグナルを除きます。

17.1.2.4 i2c で mpu6050 にデータを読み書きする

シングルバイト書き込み

Master	S	AD+W		RA		DATA		P
Slave			ACK		ACK		ACK	

株式会社日昇テクノロジー

連続バイト書き込み

Master	S	AD+W		RA		DATA		DATA		P
Slave			ACK		ACK		ACK		ACK	

MPU6050 に対して書き込み操作を行う際、メインデバイスはスタートシグナル(S)と書き込みアドレス(アドレスビットに R/W ビット、0 は書き込み)を送信します。MPU6050 は応答シグナルを生成します。その後、メインデバイスはレジスタアドレス(RA)を送信し、応答を受け取った後、レジスタデータを送信し、再び応答シグナルが必要です。連続して複数のバイトを書き込む場合も同様です。

シングルバイト読み出し

Master	S	AD+W		RA		S	AD+R			NACK	P
Slave			ACK		ACK			ACK	DATA		

連続バイト読み出し

Master	S	AD+W		RA		S	AD+R		ACK		NACK	P
Slave			ACK		ACK			ACK	DATA	DATA		

MPU6050 に対して読み出し操作を行う際、メインデバイスはスタートシグナル(S)と読み出しアドレス(アドレスビットに R/W ビット、1 は読み出し)を送信します。MPU6050 からの応答シグナルを待ちます。その後、レジスタアドレスを送信して、どのレジスタを読み出すかを MPU6050 に伝えます。応答シグナルを受け取った後、メインデバイスはもう一度スタートシグナルを送信し、スレーブデバイスの読み出しアドレスを送信します。MPU6050 は応答シグナルを生成し、レジスタデータの送信を開始します。通信は、メインデバイスが生成する否定応答シグナル(NACK)とストップシグナル(P)で終了します。

17.2 i2c ドライバフレームワーク

マイコンのベアメタル i2c ドライバを書くとき、i2c プロトコルに従って手動で i2c コントロールレジスタを設定し、スタートシグナル、ストップシグナル、データ情報などを出力する必要があります。

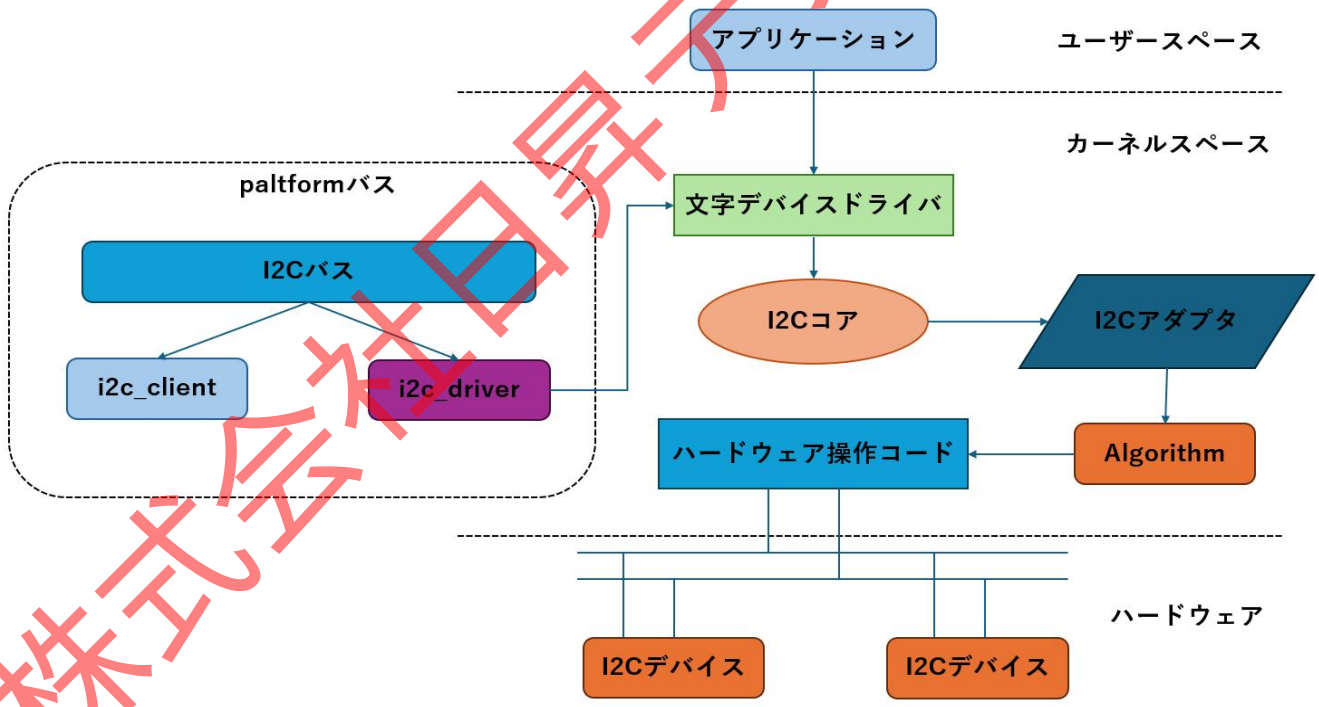
Linux システムでは、バス、デバイス、ドライバモデルが採用されています。以前説明したプラットフ

株式会社日昇テクノロジー

ォームデバイスもこのモデルを採用していますが、プラットフォームバスは仮想のバスです。

i2c（例えば i2c1）には、MPU6050、i2c インタフェースの OLED ディスプレイ、カメラ（カメラは i2c インタフェースを介して制御情報を送信）など、複数の i2c デバイスを接続できます。これらのデバイスは同じ i2c を共有しています。この i2c のドライバを i2c バスドライバと呼びます。具体的なデバイス、例えば mpu6050 のドライバは i2c デバイスドライバです。このように、mpu6050 を使用するには「2 つのドライバ」が必要です。一つは i2c バスドライバ、もう一つは mpu6050 デバイスドライバです。

- i2c バスドライバはチップメーカーが提供します（ドライバが複雑で、公式によってテストされたドライバを直接使用します）。
- mpu6050 デバイスドライバは、mpu6050 チップメーカーから入手することができます（確実ではありません）、または手で記述することができます。



上述の通り、i2c ドライバフレームワークは i2c バスドライバと特定のデバイスドライバを含みます。

i2c バスには i2c デバイス(i2c_client)と i2c ドライバ(i2c_driver)が含まれ、Linux にデバイスやドライバを登録する際、i2c バスのマッチングルールに従ってペアリングされます。ペアリングに成功すると、i2c_driver 内の.prob 関数を通じて具体的なデバイスドライバを作成できます。現代の Linux では、i2c

株式会社日昇テクノロジー

デバイスは手で作成する必要はなく、デバイスツリーメカニズムを使用して導入されます。デバイスツリーノードは platform バスと協力して使用されるため、i2c バスを一層の platform バスでラップし、デバイスツリーノードを platform バスデバイスに変換した後、それを i2c デバイスに変換し、i2c バスに登録する必要があります。

デバイスドライバが正常に作成されると、デバイスのファイル操作インターフェース(file_operations)も実装する必要があります。file_operations 内では、i2c コア関数（i2c システムが既に実装している関数で、ドライバエンジニアが使用するために公開されている関数）を使用します。これらの関数を使用すると、i2c アダプタ（i2c コントローラ）に関わることになります。i2c コントローラにはさまざまな設定があるため、Linux はそれぞれの i2c コントローラを i2c アダプタオブジェクトに抽象化します。このオブジェクトには、Algorithm という非常に重要なメンバ変数があります。Algorithm 内には、実際のハードウェア操作コードを指す一連の関数ポインタが存在します。

17.2.1 主要なデータ構造

i2c ドライバフレームワークのソースコードを解析する前に、いくつかの重要なオブジェクトを理解しておく必要があります。

struct i2c_adapter

i2c_adapter は、一つの i2c コントローラに対応し、物理的な i2c バスを識別し、それにアクセスするために必要なアルゴリズムを表す構造体です。

リスト 1: i2c_adapter 構造体（カーネルソースコード/include/linux/i2c.h）

```
1 /*
2 * i2c_adapter is the structure used to identify a physical i2c bus along
3 * with the access algorithms necessary to access it.
4 */
```

株式会社日昇テクノロジー

```
5 struct i2c_adapter {

6 struct module *owner;

7 unsigned int class; /* classes to allow probing for */

8 const struct i2c_algorithm *algo; /* the algorithm to access the bus */

9 void *algo_data;

10

11 /* data fields that are valid for all devices */

12 struct rt_mutex bus_lock;

13

14 int timeout; /* in jiffies */

15 int retries;

16 struct device dev; /* the adapter device */

17

18 int nr;

19 char name[48];

20 struct completion dev_released;

21

22 struct mutex userspace_clients_lock;

23 struct list_head userspace_clients;

24

25 struct i2c_bus_recovery_info *bus_recovery_info;

26 const struct i2c_adapter_quirks *quirks;

27 };
```

株式会社日昇テクノロジー

- algo：struct i2c_algorithm 構造体、バスにアクセスするためのアルゴリズム。

- dev：struct device 構造体、コントローラを示し、これがデバイスであることを示します。

struct i2c_algorithm

i2c_algorithm は、i2c 通信方法の抽象インターフェースで、異なるチップ上の i2c デバイスが i2c バスモデルを使用できるようにします。

struct i2c_algorithm 構造体は、バス（i2c）にアクセスするためのアルゴリズムを指定し、構造体内にはいくつかの関数ポインタメンバが含まれます。異なるメーカーは自社のハードウェアの特性に応じて、自身の i2c 転送機能を実装します。

つまり、i2c デバイス（例：mpu6050、i2c インターフェースの OLED スクリーンなど）は、これらの関数インターフェースを介して i2c バスを使用してデータの送受信を行います。i2c バスドライバ内でこれらの関数（の一部）が実装されます。

リスト 2: i2c_algorithm 構造体（カーネルソースコード/include/linux/i2c.h）

```
1 struct i2c_algorithm {
2 /* If an adapter algorithm can't do I2C-level access, set master_xfer
3 to NULL. If an adapter algorithm can do SMBus access, set
4 smbus_xfer. If set to NULL, the SMBus protocol is simulated
5 using common I2C messages */
6 /* master_xfer should return the number of messages successfully
7 processed, or a negative value on error */
8 int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs,
9 int num);
10 int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr,
```

株式会社日昇テクノロジー

```
11 unsigned short flags, char read_write,
12 u8 command, int size, union i2c_smbus_data *data);
13
14 /* To determine what the adapter supports */
15 u32 (*functionality) (struct i2c_adapter *);
16
17 #if IS_ENABLED(CONFIG_I2C_SLAVE)
18 int (*reg_slave)(struct i2c_client *client);
19 int (*unreg_slave)(struct i2c_client *client);
20 #endif
21 };
```

- master_xfer：マスターデバイスとして送信する関数で、成功したメッセージ数を返すか、エラー時には負の値を返します。

- smbus_xfer：smbus は i2c プロトコルの一種で、ハードウェアがサポートしていれば、このインターフェースを実装できます。

struct i2c_client

i2c のスレーブデバイスを表します。

リスト 3: i2c_client 構造体（カーネルソースコード/include/linux/i2c.h）

```
1 struct i2c_client {
2 unsigned short flags; /* div., see below    */
3 unsigned short addr; /* chip address - NOTE: 7bit  */
4
```

株式会社日昇テクノロジー

```
5 char name[I2C_NAME_SIZE];

6 struct i2c_adapter *adapter; /* the adapter we sit on  */

7 struct device dev; /* the device structure  */

8 int init_irq; /* irq set at initialization  */

9 int irq; /* irq issued by device  */

10 struct list_head detected;

11 #if IS_ENABLED(CONFIG_I2C_SLAVE)

12 i2c_slave_cb_t slave_cb; /* callback for slave mode */

13 #endif

14 };
```

- flags：I2C_CLIENT_TEN はデバイスが 10 ビットのチップアドレスを使用することを示し、I2C_CLIENT_PEC は SMBus データパケットエラーチェックを使用することを示します。

- addr：親アダプター上の I2C バスで使用されるアドレス。

- name：デバイスの型を示し、通常はチップ名です。

- adapter：struct i2c_adapter 構造体、この I2C デバイスをホストするバスを管理します。

- dev：ドライバーモデルデバイスノード。

- irq：このデバイスが生成する割り込み番号。

- detected：struct list_head i2c のメンバー_ドライバープログラム。クライアントリストまたは i2c コアのユーザースペースデバイスリスト。

- slave_cb：アダプターの I2C スレーブモードを使用する際のコールバック。アダプターはこれを使用して、スレーブイベントをスレーブドライバに渡します。i2c_client は、i2c バスに接続された個々のデバイス（つまり、チップ）を識別します。Linux 下で公開される動作は、デバイスを管理するドライバーによって定義されます。

株式会社日昇テクノロジー

struct i2c_driver

i2c デバイスドライバ

リスト 4: i2c_driver 構造体（カーネルソースコード/include/linux/i2c.h）

```
1 struct i2c_driver {
2 unsigned int class;
3
4 int (*probe)(struct i2c_client *, const struct i2c_device_id *);
5 int (*remove)(struct i2c_client *);
6
7 struct device_driver driver;
8 const struct i2c_device_id *id_table;
9
10 int (*detect)(struct i2c_client *, struct i2c_board_info *);
11
12 const unsigned short *address_list;
13 struct list_head clients;
14
15 ...
16 };
```

- probe: i2c デバイスと i2c ドライバがマッチした後、この関数ポインタがコールバックされます。

- id_table: struct i2c_device_id でマッチするスレーブデバイス情報。

- address_list: デバイスアドレス。

- clients: デバイスのリスト。

株式会社日昇テクノロジー

- detect: デバイス検出関数。

17.3 i2c バスドライバ

i2c バスドライバはチップメーカーから提供され、ST 公式の Linux カーネルを使用する場合、i2c バスドライバは既にカーネルに含まれており、デフォルトでカーネルにコンパイルされています。

以下は、ソースコードを組み合わせて i2c バスの動作メカニズムを簡単に紹介します。

- 1. I2C バスを登録します。

- 2. I2C ドライバを I2C バスのドライバリストに追加します。

- 3. I2C バス上のデバイスリストを走査し、i2c_device_match 関数によってマッチングを行い、マッチした場合に i2c_device_probe 関数を呼び出します。

- 4. i2c_device_probe 関数は、I2C ドライバの probe 関数を呼び出します。

i2c バス定義

リスト 5: i2c バス定義（カーネルソース/drivers/i2c/i2c-core-base.c）

```
1 struct bus_type i2c_bus_type = {
2     .name = "i2c",
3     .match = i2c_device_match,
4     .probe = i2c_device_probe,
5     .remove = i2c_device_remove,
6     .shutdown = i2c_device_shutdown,
7 };
```

i2c バスは、I2C ドライバと I2C デバイスの二つのリストを維持し、I2C デバイスと I2C ドライバのマッチングや削除などを管理します。

i2c バス登録

Linux が起動した後、デフォルトで i2c_init が実行されます。

株式会社日昇テクノロジー

リスト 6: i2c バス登録（カーネルソース/drivers/i2c/i2c-core-base.c）

```
1 static int __init i2c_init(void)
2 {
3 int retval;
4 ...
5 retval = bus_register(&i2c_bus_type);
6 if (retval)
7 return retval;
8
9 is_registered = true;
10 ...
11 retval = i2c_add_driver(&dummy_driver);
12 if (retval)
13 goto class_err;
14
15 if (IS_ENABLED(CONFIG_OF_DYNAMIC))
16 WARN_ON(of_reconfig_notifier_register(&i2c_of_notifier));
17 if (IS_ENABLED(CONFIG_ACPI))
18 WARN_ON(acpi_reconfig_notifier_register(&i2c_acpi_notifier));
19
20 return 0;
21 ...
22 }
```

株式会社日昇テクノロジー

- 5 行目: bus_register で i2c_bus_type バスを登録します。バス定義は上記の通りです。

- 11 行目: i2c_add_driver でデバイス dummy_driver を登録します。

i2c デバイスと i2c ドライバのマッチングルール

リスト 7: i2c デバイスと i2c ドライバのマッチングルール（カーネルソース/drivers/i2c/i2c-core-base.c）

```
1 static int i2c_device_match(struct device *dev, struct device_driver *drv)
2 {
3 struct i2c_client *client = i2c_verify_client(dev);
4 struct i2c_driver *driver;
5
6 /* Attempt an OF style match */
7 if (i2c_of_match_device(drv->of_match_table, client))
8 return 1;
9
10 /* Then ACPI style match */
11 if (acpi_driver_match_device(dev, drv))
12 return 1;
13
14 driver = to_i2c_driver(drv);
15
16 /* Finally an I2C match */
17 if (i2c_match_id(driver->id_table, client))
18 return 1;
```

株式会社日昇テクノロジー

```
19

20 return 0;

21 }
```

- of_driver_match_device: デバイスツリーのマッチング方法。I2C デバイスノードの compatible 属性と of_device_id の compatible 属性を比較します。

- acpi_driver_match_device: ACPI マッチング方式。

- i2c_match_id: i2c バスの伝統的なマッチング方式。I2C デバイス名と i2c ドライバの id_table->name フィールドが等しいかどうかを比較します。

i2c バスドライバコードの重要な部分のみを簡単に紹介しました。詳細については、完全な i2c ドライバソースコードを自身で読むことができます。通常、ドライバプログラムを見る際には、ドライバのエントリーとエグジット関数を最初に探す必要があります。これらは通常、ドライバの最後に位置しています（例: RK3588）。

リスト 8: ドライバエントリーとエグジット関数（カーネルソース/drivers/i2c/busses/i2c-rk3x.c）

```
1 static struct platform_driver rk3x_i2c_driver = {

2 .probe = rk3x_i2c_probe,

3 .remove = rk3x_i2c_remove,

4 .driver = {

5 .name = "rk3x-i2c",

6 .of_match_table = rk3x_i2c_match,

7 .pm = &rk3x_i2c_pm_ops,

8 },

9 };
```

株式会社日昇テクノロジー

10 module_platform_driver(rk3x_i2c_driver);

ドライバ登録関数 module_platform_driver（カーネルソース/include/linux/platform_device.h に定義されている）の詳細は、以前の動的デバイスツリーの章で説明しました。i2c ドライバがプラットフォームドライバであることがわかります。プラットフォームドライバ構造体は「rk3x_i2c_driver」であることがわかります。

リスト 9: プラットフォームデバイスドライバ構造体（カーネルソース/drivers/i2c/busses/i2c-rk3x.c）

```
1 static const struct of_device_id rk3x_i2c_match[] = {
2 {
3 .compatible = "rockchip,rv1108-i2c",
4 .data = &rv1108_soc_data
5 },
6 {
7 .compatible = "rockchip,rv1126-i2c",
8 .data = &rv1126_soc_data
9 },
10 {
11 .compatible = "rockchip,rk3066-i2c",
12 .data = &rk3066_soc_data
13 },
14 {
15 .compatible = "rockchip,rk3188-i2c",
16 .data = &rk3188_soc_data
17 },
```

株式会社日昇テクノロジー

```
18 {
19 .compatible = "rockchip,rk3228-i2c",
20 .data = &rk3228_soc_data
21 },
22 {
23 .compatible = "rockchip,rk3288-i2c",
24 .data = &rk3288_soc_data
25 },
26 {
27 .compatible = "rockchip,rk3399-i2c",
28 .data = &rk3399_soc_data
29 },
30 {},
31 };
32 MODULE_DEVICE_TABLE(of, rk3x_i2c_match);
33
34 static struct platform_driver rk3x_i2c_driver = {
35 .probe = rk3x_i2c_probe,
36 .remove = rk3x_i2c_remove,
37 .driver = {
38 .name = "rk3x-i2c",
39 .of_match_table = rk3x_i2c_match,
```

株式会社日昇テクノロジー

```
40 .pm = &rk3x_i2c_pm_ops,
41 },
42 };
```

- 1-5 行目: i2c ドライバのマッチングテーブルで、デバイスツリーノードとのマッチングに使用されます。

- 8-16 行目: 初期化されたプラットフォームデバイス構造体。この構造体から.prob 関数を見つけることができます。.prob 関数の役割はよく知られており、通常、デバイスの基本初期化を行うために使用されます。

以下は、.probe 関数の内容です。

リスト 10: RK3588 の i2c コントローラドライバの.probe 関数（カーネルソースコード /drivers/i2c/busses/i2c-rk3x.c）

```
1 static int rk3x_i2c_probe(struct platform_device *pdev)
2 {
3 struct device_node *np = pdev->dev.of_node;
4 const struct of_device_id *match;
5 struct rk3x_i2c *i2c;
6 struct resource *mem;
7 int ret = 0;
8 u32 value;
9 int irq;
10 unsigned long clk_rate;
11
```

株式会社日昇テクノロジー

```
12 i2c = devm_kzalloc(&pdev->dev, sizeof(struct rk3x_i2c), GFP_KERNEL); // rk3x_i2c 構造体のための
メモリ空間を割り当て
13 if (!i2c)
14 return -ENOMEM;
15
16 match = of_match_node(rk3x_i2c_match, np); // 対応するデバイスノードの of_device_id を見つける
17 i2c->soc_data = match->data; // of_device_id から.data メンバー、つまり.data = &rk3399_soc_data
を取得
18
19 /* I2C タイミングプロパティを取得するための共通インターフェースを使用 */
20 i2c_parse_fw_timings(&pdev->dev, &i2c->t, true);
21
22 strlcpy(i2c->adap.name, "rk3x-i2c", sizeof(i2c->adap.name));
23 i2c->adap.owner = THIS_MODULE;
24 i2c->adap.algo = &rk3x_i2c_algorithm;
25 i2c->adap.retries = 3;
26 i2c->adap.dev.of_node = np;
27 i2c->adap.algo_data = i2c;
28 i2c->adap.dev.parent = &pdev->dev;
29
30 i2c->dev = &pdev->dev;
31
```

株式会社日昇テクノロジー

```
32 spin_lock_init(&i2c->lock);

33 init_waitqueue_head(&i2c->wait);

34

35 i2c->i2c_restart_nb.notifier_call = rk3x_i2c_restart_notify;

36 i2c->i2c_restart_nb.priority = 128;

37 ret = register_pre_restart_handler(&i2c->i2c_restart_nb);

38 if (ret) {

39 dev_err(&pdev->dev, "failed to setup i2c restart handler.¥n");

40 return ret;

41 }

42

43 mem = platform_get_resource(pdev, IORESOURCE_MEM, 0);

44 i2c->regs = devm_ioremap_resource(&pdev->dev, mem);

45 if (IS_ERR(i2c->regs))

46 return PTR_ERR(i2c->regs);

47

48 /*

49 * Switch to new interface if the SoC also offers the old one.

50 * The control bit is located in the GRF register space.

51 */

52 if (i2c->soc_data->grf_offset >= 0) {

53 struct regmap *grf;
```

株式会社日昇テクノロジー

```
54
55 grf = syscon_regmap_lookup_by_phandle(np, "rockchip,grf");
56 if (!IS_ERR(grf)) {
57 int bus_nr;
58
59 /* Try to set the I2C adapter number from dt */
60 bus_nr = of_alias_get_id(np, "i2c");
61 if (bus_nr < 0) {
62 dev_err(&pdev->dev, "rk3x-i2c needs i2cX alias");
63 return -EINVAL;
64 }
65
66 if (i2c->soc_data == &rv1108_soc_data && bus_nr == 2)
67 /* rv1108 i2c2 set grf offset-0x408, bit-10 */
68 value = BIT(26) | BIT(10);
69 else if (i2c->soc_data == &rv1126_soc_data &&
70 bus_nr == 2)
71 /* rv1126 i2c2 set pmugrf offset-0x118, bit-4 */
72 value = BIT(20) | BIT(4);
73 else
74 /* rk3xxx 27+i: write mask, 11+i: value */
75 value = BIT(27 + bus_nr) | BIT(11 + bus_nr);
```

株式会社日昇テクノロジー

```
76
77 ret = regmap_write(grf, i2c->soc_data->grf_offset,
78 value);
79 if (ret != 0) {
80 dev_err(i2c->dev, "Could not write to GRF: %d¥n",
81 ret);
82 return ret;
83 }
84 }
85 }
86
87 /* IRQ setup */
88 irq = platform_get_irq(pdev, 0);
89 if (irq < 0) {
90 dev_err(&pdev->dev, "cannot find rk3x IRQ¥n");
91 return irq;
92 }
93
94 ret = devm_request_irq(&pdev->dev, irq, rk3x_i2c_irq,
95 0, dev_name(&pdev->dev), i2c);
96 if (ret < 0) {
97 dev_err(&pdev->dev, "cannot request IRQ¥n");
```

株式会社日昇テクノロジー

```
98 return ret;

99 }

100

101 platform_set_drvdata(pdev, i2c);

102

103 if (i2c->soc_data->calc_timings == rk3x_i2c_v0_calc_timings) {

104 /* Only one clock to use for bus clock and peripheral clock */

105 i2c->clk = devm_clk_get(&pdev->dev, NULL);

106 i2c->pclk = i2c->clk;

107 } else {

108 i2c->clk = devm_clk_get(&pdev->dev, "i2c");

109 i2c->pclk = devm_clk_get(&pdev->dev, "pclk");

110 }

111

112 if (IS_ERR(i2c->clk)) {

113 ret = PTR_ERR(i2c->clk);

114 if (ret != -EPROBE_DEFER)

115 dev_err(&pdev->dev, "Can't get bus clk: %d¥n", ret);

116 return ret;

117 }

118 if (IS_ERR(i2c->pclk)) {

119 ret = PTR_ERR(i2c->pclk);
```

株式会社日昇テクノロジー

```
120 if (ret != -EPROBE_DEFER)
121 dev_err(&pdev->dev, "Can't get periph clk: %d¥n", ret);
122 return ret;
123 }
124
125 ret = clk_prepare(i2c->clk);
126 if (ret < 0) {
127 dev_err(&pdev->dev, "Can't prepare bus clk: %d¥n", ret);
128 return ret;
129 }
130 ret = clk_prepare(i2c->pclk);
131 if (ret < 0) {
132 dev_err(&pdev->dev, "Can't prepare periph clock: %d¥n", ret);
133 goto err_clk;
134 }
135
136 i2c->clk_rate_nb.notifier_call = rk3x_i2c_clk_notifier_cb;
137 ret = clk_notifier_register(i2c->clk, &i2c->clk_rate_nb);
138 if (ret != 0) {
139 dev_err(&pdev->dev, "Unable to register clock notifier¥n");
140 goto err_pclk;
141 }
```

株式会社日昇テクノロジー

```
142

143 clk_rate = clk_get_rate(i2c->clk);

144 rk3x_i2c_adapt_div(i2c, clk_rate);

145

146 ret = i2c_add_adapter(&i2c->adap);

147 if (ret < 0)

148 goto err_clk_notifier;

149

150 return 0;

151

152 err_clk_notifier:

153 clk_notifier_unregister(i2c->clk, &i2c->clk_rate_nb);

154 err_pclk:

155 clk_unprepare(i2c->pclk);

156 err_clk:

157 clk_unprepare(i2c->clk);

158 return ret;

159 }
```

株式会社日昇テクノロジー

- メモリ空間 rk3x_i2c 構造体の割り当て

- 対応するデバイスノードの of_device_id を見つける

- of_device_id を通じて、.data メンバを取得し、.data = &rk3399_soc_data をセットする

- 12-14 行目：rk3x_i2c 構造体のためのスペースを確保し、後述します。

- 43-44 行目：reg プロパティを取得し、カーネルが提供する「platform_get_resource」を使用します。これは、of 関数を使用して reg プロパティを取得する機能と同様です。ここで、i2c のベースアドレスを取得し、「devm_ioremap_resource」を使用して仮想アドレスに変換します。

- 87-99 行目：割り込み番号を取得します。i2c のデバイスツリーノードで定義された割り込みをここで取得し、割り込みを要求する際に使用します。取得関数は、カーネルが提供する「irq_of_parse_and_map」を使用します。

- 122-144 行目：クロック設定を取得し、設定します。

残りは PM パワーマネージメント、アダプターの設定で、比較的簡単です。最終的には「i2c_add_adapter」関数を使用して、プラットフォームドライバをバスに登録します。

以下は、rk3x_i2c 構造体についてです。これは、メーカーの i2c コントローラと Linux プラットフォームを関連付けるための橋渡しとなる構造体です。

リスト 11: rk3x_i2c 構造体

```
1 struct rk3x_i2c {
2 struct i2c_adapter adap;
3 struct device *dev;
4 const struct rk3x_i2c_soc_data *soc_data;
5
6 /* Hardware resources */
7 void __iomem *regs;
8 struct clk *clk;
9 struct clk *pclk;
10 struct notifier_block clk_rate_nb;
11
```

株式会社日昇テクノロジー

```
12 /* Settings */
13 struct i2c_timings t;
14
15 /* Synchronization & notification */
16 spinlock_t lock;
17 wait_queue_head_t wait;
18 bool busy;
19
20 /* Current message */
21 struct i2c_msg *msg;
22 u8 addr;
23 unsigned int mode;
24 bool is_last_msg;
25
26 /* I2C state machine */
27 enum rk3x_i2c_state state;
28 unsigned int processed;
29 int error;
30 unsigned int suspended:1;
31
32 struct notifier_block i2c_restart_nb;
33 bool system_restarting;
34 };
```

株式会社日昇テクノロジー

rk3x_i2c 構造体はメンバーが多く、メーカーの I2C コントローラー情報およびバスに登録される予定のアダプターに関して記述しています。この構造体を通じて、Linux 下の I2C バスモデルとメーカーのチップドライバ機能を関連付けることができます。

- adap：バスに登録されるアダプタ
- irq：i2c の割り込み番号を保存
- clk：クロック関連情報を保存する clk 構造体
- busy：イベント待機の条件

上記の probe 関数では、rk3x_i2c 構造体内の adap メンバを初期化します。特に注目すべきは、「i2c->adap.algo = &rk3x_i2c_algorithm」という部分で、これは「i2c バスへのアクセスアルゴリズム」を初期化するために使用されます。「rk3x_i2c_algorithm」の定義は以下の通りです。

リスト 12: i2c_algorithm 構造体インスタンス rk3x_i2c_algorithm

```
1 static const struct i2c_algorithm rk3x_i2c_algorithm = {
2     .master_xfer = rk3x_i2c_xfer,
3     .functionality = rk3x_i2c_func,
4 };
```

st_i2c_algo 構造体には、i2c バスを外部からアクセスするための 2 つの関数が指定されています：

- 関数 rk3x_i2c_func は、I2C バスが提供する機能を返すだけです。
- 関数 rk3x_i2c_xfer は、i2c 外部デバイスへのアクセスを実際に実装し、データ転送を行います。

rk3x_i2c_xfer 関数の定義は以下の通りです

リスト 13: rk3x_i2c_xfer 関数

```
1 static int rk3x_i2c_xfer(struct i2c_adapter *adap,
2 struct i2c_msg *msgs, int num)
```

株式会社日昇テクノロジー

```
3 {
4 struct rk3x_i2c *i2c = (struct rk3x_i2c *)adap->algo_data;
5 unsigned long timeout, flags;
6 u32 val;
7 int ret = 0;
8 int i;
9
10 if (i2c->suspended)
11 return -EACCES;
12
13 spin_lock_irqsave(&i2c->lock, flags);
14
15 clk_enable(i2c->clk);
16 clk_enable(i2c->pclk);
17
18 i2c->is_last_msg = false;
19
20 /*
21 * Process msgs. We can handle more than one message at once (see
22 * rk3x_i2c_setup()).
23 */
24 for (i = 0; i < num; i += ret) {
```

株式会社日昇テクノロジー

```
25 ret = rk3x_i2c_setup(i2c, msgs + i, num - i);

26

27 if (ret < 0) {

28 dev_err(i2c->dev, "rk3x_i2c_setup() failed¥n");

29 break;

30 }

31

32 if (i + ret >= num)

33 i2c->is_last_msg = true;

34

35 rk3x_i2c_start(i2c);

36

37 spin_unlock_irqrestore(&i2c->lock, flags);

38

39 timeout = wait_event_timeout(i2c->wait, !i2c->busy,

40 msecs_to_jiffies(WAIT_TIMEOUT));

41

42 spin_lock_irqsave(&i2c->lock, flags);

43

44 if (timeout == 0) {

45 dev_err(i2c->dev, "timeout, ipd: 0x%02x, state: %d¥n",

46 i2c_readl(i2c, REG_IPD), i2c->state);
```

株式会社日昇テクノロジー

```
47

48 /* Force a STOP condition without interrupt */

49 rk3x_i2c_disable_irq(i2c);

50 val = i2c_readl(i2c, REG_CON) & REG_CON_TUNING_MASK;

51 val |= REG_CON_EN | REG_CON_STOP;

52 i2c_writel(i2c, val, REG_CON);

53

54 i2c->state = STATE_IDLE;

55

56 ret = -ETIMEDOUT;

57 break;

58 }

59

60 if (i2c->error) {

61 ret = i2c->error;

62 break;

63 }

64 }

65

66 rk3x_i2c_disable_irq(i2c);

67 rk3x_i2c_disable(i2c);

68
```

株式会社日昇テクノロジー

```
69 clk_disable(i2c->pclk);

70 clk_disable(i2c->clk);

71

72 spin_unlock_irqrestore(&i2c->lock, flags);

73

74 return ret < 0 ? ret : num;

75 }
```

デバイスドライバ、例えば mpu6050 のドライバを記述する際に、「i2c_transfer」関数を使用してデータ転送を実行します。「i2c_transfer」関数は、最終的に「rk3x_i2c_xfer」関数を呼び出して具体的な送受信作業を実行します。これについては後ほど「i2c_transfer」関数の使用法を詳しく説明します。

rk3x_i2c_xfer 内で、実際の送受信は rk3x_i2c_setup を通じて完了されます。関数の実装は以下の通りです：

リスト 14: rk3x_i2c_setup 関数

```
static int rk3x_i2c_setup(struct rk3x_i2c *i2c, struct i2c_msg *msgs, int num)

{

u32 addr = (msgs[0].addr & 0x7f) << 1;

int ret = 0;

/*

* The I2C adapter can issue a small (len < 4) write packet before

* reading. This speeds up SMBus-style register reads.

* The MRXADDR/MRXRADDR hold the slave address and the slave register

* address in this case.

*/
```

株式会社日昇テクノロジー

```
if (num >= 2 && msgs[0].len < 4 &&

!(msgs[0].flags & I2C_M_RD) && (msgs[1].flags & I2C_M_RD)) {

u32 reg_addr = 0;

int i;

dev_dbg(i2c->dev, "Combined write/read from addr 0x%x¥n",

addr >> 1);

/* Fill MRXRADDR with the register address(es) */

for (i = 0; i < msgs[0].len; ++i) {

reg_addr |= msgs[0].buf[i] << (i * 8);

reg_addr |= REG_MRXADDR_VALID(i);

}

/* msgs[0] is handled by hw. */

i2c->msg = &msgs[1];

i2c->mode = REG_CON_MOD_REGISTER_TX;

i2c_writel(i2c, addr | REG_MRXADDR_VALID(0), REG_MRXADDR);

i2c_writel(i2c, reg_addr, REG_MRXRADDR);

ret = 2;

} else {

/*

* We'll have to do it the boring way and process the msgs

* one-by-one.

*/
```

株式会社日昇テクノロジー

```
if (msgs[0].flags & I2C_M_RD) {

addr |= 1; /* set read bit */

/*

* We have to transmit the slave addr first. Use

* MOD_REGISTER_TX for that purpose.

*/

i2c->mode = REG_CON_MOD_REGISTER_TX;

i2c_writel(i2c, addr | REG_MRXADDR_VALID(0),

REG_MRXADDR);

i2c_writel(i2c, 0, REG_MRXRADDR);

} else {

i2c->mode = REG_CON_MOD_TX;

}

i2c->msg = &msgs[0];

ret = 1;

}

i2c->addr = msgs[0].addr;

i2c->busy = true;

i2c->processed = 0;

i2c->error = 0;

rk3x_i2c_clean_ipd(i2c);

return ret;

}
```

株式会社日昇テクノロジー

ここでは、外部デバイスレジスタへの操作に当てますが、操作内容は簡単な説明はコード内のコメントを参照してください。

以上で、i2c プラットフォームドライバについての説明を終えます。probe 関数は i2c の基本的な初期化を行い、システムに追加します。また、ドライバは i2c 外部インターフェース関数も実装しています。i2c_adapter 構造体の初期化時に、バスアクセスアルゴリズム構造体 i2c_algorithm も初期化されました。

それでは、probe 関数全体をまとめると、主に 2 つの作業が完了しています。第一に、I2C ハードウェアの初期化、第二に、I2C 外部デバイスにアクセス可能な i2c_adapter 構造体を初期化し、それをシステムに追加することです。

17.4 i2c デバイスドライバのコア関数

i2c_add_adapter()

Linux システムに i2c アダプタを登録します。

リスト 15: i2c アダプタの登録（カーネルソースコード/drivers/i2c/i2c-core-base.c）

```
1 // Linux システムが自動的に i2c アダプタ番号を設定します（adapter->nr）
2 int i2c_add_adapter(struct i2c_adapter *adapter)
3 // i2c アダプタ番号を手で設定します（adapter->nr）
4 int i2c_add_numbered_adapter(struct i2c_adapter *adapter)
```

パラメータ

- adapter: i2c 物理コントローラに対応するアダプタ

戻り値:

- 成功: 0

- 失敗: 負の数

i2c_add_driver() マクロ

リスト 16: i2c ドライバの登録（カーネルソースコード/include/linux/i2c.h）

株式会社日昇テクノロジー

```
1 #define i2c_add_driver(driver)
```

このマクロ関数の本質は、i2c_register_driver()関数を呼び出すことです。

i2c_register_driver() 関数

リスト 17: i2c ドライバの登録（カーネルソースコード/drivers/i2c/i2c-core-base.c）

```
1 int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
```

パラメータ

- owner: 通常は THIS_MODULE

- driver: 登録する i2c_driver

戻り値:

- 成功: 0

- 失敗: 負の数

i2c_transfer() 関数

i2c_transfer()関数は、最終的に前述の rk3x_i2c_xfer()関数を呼び出してデータ転送を実現します。

リスト 18: i2c メッセージの送受信（カーネルソースコード/drivers/i2c/i2c-core-base.c）

```
1 int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)
```

パラメータ

- adap: struct i2c_adapter 構造体、メッセージを送受信するために使用される i2c アダプタ。

i2c_client は、対応する i2c_adapter を保存します。

- msgs: struct i2c_msg 構造体、送信する一つまたは複数の i2c メッセージ

- num: メッセージの数、つまり msgs の数

戻り値:

- 成功: 送信された msgs の数

- 失敗: 負の数

株式会社日昇テクノロジー

i2c_msg 構造体

リスト 19: i2c メッセージの記述（カーネルソースコード/include/uapi/linux/i2c.h）

```
1 struct i2c_msg {
2    __u16 addr; /* スレーブアドレス */
3    __u16 flags;
4    ...
5    __u16 len; /* メッセージ長 */
6    __u8 *buf; /* メッセージデータへのポインタ */
7 };
```

- addr: i2c デバイスアドレス

- flags: メッセージの送信方向と特性。I2C_M_RD は読み取りメッセージを示し、0 は送信メッセージを示す。

- len: メッセージデータの長さ

- buf: メッセージのデータを格納する文字配列、メッセージのバッファとして機能

i2c_master_send()関数

リスト 20: i2c メッセージの送信（カーネルソースコード/include/linux/i2c.h）

```
1 static inline int i2c_master_send(const struct i2c_client *client,
2                const char *buf, int count)
3 {
4    return i2c_transfer_buffer_flags(client, (char *)buf, count, 0);
5 };
```

i2c_master_recv()関数

リスト 21: i2c メッセージの受信（カーネルソースコード/include/linux/i2c.h）

株式会社日昇テクノロジー

```
1 static inline int i2c_master_recv(const struct i2c_client *client,
2                       char *buf, int count)
3 {
4     return i2c_transfer_buffer_flags(client, buf, count, I2C_M_RD);
5 };
```

i2c_transfer_buffer_flags()関数

リスト 22: i2c メッセージの送信（カーネルソースコード/drivers/i2c/i2c-core-base.c）

```
1 int i2c_transfer_buffer_flags(const struct i2c_client *client, char *buf,
2 int count, u16 flags)
3 {
4     int ret;
5     struct i2c_msg msg = {
6         .addr = client->addr,
7         .flags = flags | (client->flags & I2C_M_TEN),
8         .len = count,
9         .buf = buf,
10    };
11    ret = i2c_transfer(client->adapter, &msg, 1);
12    /*
13    * If everything went ok (i.e., 1 msg transferred), return #bytes
14    * transferred, else error code.
15    */
```

株式会社日昇テクノロジー

```
16    return (ret == 1) ? count : ret;

17 }
```

以下では、mpu6050 を例に i2c デバイスドライバの書き方について説明します。

17.5 mpu6050 ドライバ実験

17.5.1 ハードウェア紹介

このセクションの実験では、lubancat4 ボードおよび「公式 MPU6050 モジュール」を使用します。

MPU6050 は、3 軸 MEMS ジャイロスコープと 3 軸 MEMS 加速度計を統合したモーション処理センサーです。

mpu6050 と lubancat4 のピン接続対応表:

MPU6050 ピン（モジュールのシルクプリント）	説明	lubancat4 ボードピン
SCL	SCL ピン	I2C3_SCL
SDA	SDA ピン	I2C3_SDA
XDA	未使用	
XCL	未使用	
AD0	GND に接続	GND
int	GND に接続	
GND	GND	GND
VCC	電源	3.3V

ヒント: lubancat4 ボードのピンは「LubanCat-RK ボードクイックスタートガイド」の 40 ピン対応表を参照してください。

MPU6050 は i2c で開発ボードに接続され、センサー上の SDA と SCL は開発ボードの i2c3 に接続されます。開発ボードが MPU6050 を制御するためには、これら 2 つのピンを i2c コントローラピンとしてマルチプレクスする必要があります。

MPU チップマニュアルによると、MPU6050 のスレーブアドレスは b110100X で、7 ビット長で、最下位ビット X は AD0 ピンのロジックレベルによって決定されます。回路図では、AD0 は GND に接続

株式会社日昇テクノロジー

されているため、アドレスは b1101000、つまり 0x68 です。また、割り込みピン「INT」は使用されていません。

17.5.1.1 デバイスツリー

Lubancat4 ボードでは、i2c3 m0 を介して mpu6050 と通信します。以下は、i2c3 コントローラのデバイスツリーコードです:

リスト 23: RK3588.dtsi

```
1 i2c3: i2c@fe5c0000 {
2     compatible = "rockchip,rk3399-i2c";
3     reg = <0x0 0xfe5c0000 0x0 0x1000>;
4     clocks = <&cru CLK_I2C3>, <&cru PCLK_I2C3>;
5     clock-names = "i2c", "pclk";
6     interrupts = <GIC_SPI 49 IRQ_TYPE_LEVEL_HIGH>;
7     pinctrl-names = "default";
8     pinctrl-0 = <&i2c3m0_xfer>; //デフォルトは GPIO1_A1 と GPIO1_A0 が I2C3 ピンとしてマルチプレクスされています
9     #address-cells = <1>;
10     #size-cells = <0>;
11     status = "disabled";
12 };
```

株式会社日昇テクノロジー

リスト 24: RK3588-pinctrl.dtsi

```
1 i2c3 {
2 i2c3m0_xfer: i2c3m0-xfer {
3 rockchip,pins =
4 /* i2c3_sclm0 */
5 <1 RK_PA1 1 &pcfg_pull_none_smt>,
6 /* i2c3_sdam0 */
7 <1 RK_PA0 1 &pcfg_pull_none_smt>;
8 };
9
10 i2c3m1_xfer: i2c3m1-xfer {
11 rockchip,pins =
12 /* i2c3_sclm1 */
13 <3 RK_PB5 4 &pcfg_pull_none_smt>,
14 /* i2c3_sdam1 */
15 <3 RK_PB6 4 &pcfg_pull_none_smt>;
16 };
17 };
```

株式会社日昇テクノロジー

リスト 25: mpu6050 デバイスツリープラグイン

```
1 /*
2  * Copyright (C) 2022 - All Rights Reserved by
3  * EmbedFire LubanCat
```

```
4 */
5
6 /dts-v1/;
7 /plugin/;
8
9 #include <dt-bindings/gpio/gpio.h>
10 #include <dt-bindings/pinctrl/rockchip.h>
11 #include <dt-bindings/clock/RK3588-cru.h>
12 #include <dt-bindings/interrupt-controller/irq.h>
13
14 &i2c3 {
15     status = "okay";
16     pinctrl-names = "default";
17     pinctrl-0 = <&i2c3m0_xfer>; // GPIO1_A1 と GPIO1_A0 を I2C3 ピンとしてマルチプレクス
18     #address-cells = <1>;
19     #size-cells = <0>;
20
21     mpu6050@68 {
22         compatible = "fire,i2c_mpu6050";
23         //compatible = "invensense,mpu6050"
24         reg = <0x68>;
25         status = "okay";
```

株式会社日昇テクノロジー

```
26    };
27 };
```

- 15 行目：i2c3 ノードを有効にする

- 17 行目：i2c3 のピンとして i2c3m1 を使用

- 21 行目：MPU6050 サブノードを追加

- 22-23 行目：MPU6050 サブノードのプロパティを"fire,i2c_mpu6050"に設定し、ドライバと一致させます。"invensense,mpu6050"をコメントアウトしています。このプロパティは、カーネルに組み込まれている mpu6050 ドライバを使用できますが、この組み込みドライバは iio サブシステムを使用して実装されています。興味がある場合は、自分で調べることができます。

17.5.2 実験コードの解説

17.5.2.1 プログラミングアプローチ

i2c_mpu6050 ドライバの実験プログラミングアプローチは以下の通りです：

- ハードウェアの回路図を分析し、mpu6050 のデバイスツリープラグインを作成します（既に実装済み）。

- mpu6050 ドライバプログラムを作成します。

- シンプルなテストアプリケーションを作成します。

17.5.2.2 mpu6050 ドライバの実装

rockchip 公式から提供されている i2c のバスドライバがあるため、mpu6050 デバイスドライバは非常にシンプルになります。以下は、コードを組み合わせて mpu6050 デバイスドライバの実装を紹介します。

mpu6050 ドライバプログラムは、以下のような構造になります：

株式会社日昇テクノロジー

リスト 26: mpu6050 ドライバプログラムの構造

```
1 static int i2c_write_mpu6050(struct i2c_client *mpu6050_client, u8 address, u8 data)
2 {
3     return 0;
4 }
5
6 static int i2c_read_mpu6050(struct i2c_client *mpu6050_client, u8 address, void *data, u32 length)
7 {
8     return 0;
9 }
10
11 static int mpu6050_init(void)
12 {
13     return 0;
14 }
15
16 /* キャラクタデバイス操作関数集、.open 関数の実装 */
17 static int mpu6050_open(struct inode *inode, struct file *filp)
18 {
19     return 0;
20 }
21
```

株式会社日昇テクノロジー

```
22 /* キャラクタデバイス操作関数集、.read 関数の実装 */
23 static ssize_t mpu6050_read(struct file *filp, char __user *buf, size_t cnt, loff_t *off)
24 {
25     return 0;
26 }
27
28 /* キャラクタデバイス操作関数集、.release 関数の実装 */
29 static int mpu6050_release(struct inode *inode, struct file *filp)
30 {
31     return 0;
32 }
33
34 /* キャラクタデバイス操作関数集 */
35 static struct file_operations mpu6050_chr_dev_fops =
36 {
37     .owner = THIS_MODULE,
38     .open = mpu6050_open,
39     .read = mpu6050_read,
40     .release = mpu6050_release,
41 };
42
43 /* i2c バスデバイス関数集 */
```

株式会社日昇テクノロジー

```
44 static int mpu6050_probe(struct i2c_client *client, const struct i2c_device_id *id)
45 {
46     return 0;
47 }
48
49 static int mpu6050_remove(struct i2c_client *client)
50 {
51     /* デバイスの削除 */
52     return 0;
53 }
54
55 /* i2c  バスデバイス構造体の定義 */
56 struct i2c_driver mpu6050_driver = {
57     .probe = mpu6050_probe,
58     .remove = mpu6050_remove,
59     .id_table = gtp_device_id,
60 };
61
62 /*
63  * ドライバの初期化関数
64  */
65 static int __init mpu6050_driver_init(void)
```

株式会社日昇テクノロジー

```
66 {
67     return 0;
68 }
69
70 /*
71  * ドライバの終了関数
72  */
73 static void __exit mpu6050_driver_exit(void)
74 {
75
76 }
77
78 module_init(mpu6050_driver_init);
79 module_exit(mpu6050_driver_exit);
80
81 MODULE_LICENSE("GPL");
```

ドライバプログラムは以下の四部分に分けられます（下から上に見てください）：

- 49-73 行目：i2c バスデバイス構造体を定義し、i2c バスデバイスの登録と登録解除関数を実装します。これは、ドライバプログラムのエントリとエグジット関数です。

- 38-47 行目：i2c バスデバイス構造体で定義された操作関数を実装します。主に、.probe マッチング関数で、ここでキャラクタデバイスを追加・登録し、mpu6050 の具体的な機能を実装するキャラクタデバイスを使用します。

株式会社日昇テクノロジー

- 14-36 行目：キャラクタデバイス操作関数集を定義し実装します。アプリケーションからの open や read 操作がカーネルに到達すると、これらの関数が実行され、mpu6050 の初期化や結果読み取りを実際に実行します。

- 1-12 行目：mpu6050 の読み書き関数を具体的に実装します。これらは第三部の関数によって呼び出され、ユーザーが自由に定義します。

次に、これら四部分の内容に従って、mpu6050 デバイスドライバプログラムの実装を紹介します。

17.5.2.2.1 ドライバのエントリーとエグジット関数の実装

ドライバのエントリーとエグジット関数は、i2c デバイスドライバの登録と解除のためだけに使用されます。以下がそのコードです：

リスト 27: mpu6050 ドライバのエントリーとエグジット関数の実装

```
1 /* ID マッチングテーブルの定義 */
2 static const struct i2c_device_id gtp_device_id[] = {
3     {"fire,i2c_mpu6050", 0},
4     {}};


5 /* デバイスツリーのマッチングテーブルの定義 */
6 static const struct of_device_id mpu6050_of_match_table[] = {
7     {.compatible = "fire,i2c_mpu6050"},
8     {/* sentinel */}};
9
10 /* i2c デバイス構造体の定義 */
11 struct i2c_driver mpu6050_driver = {
```

株式会社日昇テクノロジー

```
12    .probe = mpu6050_probe,
13    .remove = mpu6050_remove,
14     .id_table = gtp_device_id,
15     .driver = {
16         .name = "fire,i2c_mpu6050",
17         .owner = THIS_MODULE,
18         .of_match_table = mpu6050_of_match_table,
19     },
20 };
21
22 /*
23  * ドライバの初期化関数
24  */
25 static int __init mpu6050_driver_init(void)
26 {
27     int ret;
28     pr_info("mpu6050_driver_init¥n");
29     ret = i2c_add_driver(&mpu6050_driver);
30     return ret;
31 }
32
33 /*
```

株式会社日昇テクノロジー

```
34 * ドライバの終了関数
35 */
36 static void __exit mpu6050_driver_exit(void)
37 {
38     pr_info("mpu6050_driver_exit¥n");
39     i2c_del_driver(&mpu6050_driver);
40 }
41
42 module_init(mpu6050_driver_init);
43 module_exit(mpu6050_driver_exit);
44
45 MODULE_LICENSE("GPL");
```

- 1-9 行目：デバイスツリーのマッチングテーブルの定義。

- 13-14 行目：.probe と .remove は i2c デバイスの操作関数です。.probe 関数はマッチング成功後に実行され、デバイスが解除される前に .remove 関数が実行されます。これら二つの関数の実装は後述します。

- 12-21 行目：i2c デバイスドライバ構造体 mpu6050_driver の定義。これは、以前に学んだプラットフォームデバイスドライバと似ており、「構造体」が一つのデバイスを表します。主なメンバーは「.id_table」と「.of_match_table」で、デバイスツリーノードとのマッチングに使用されます。

- 26-41 行目：ドライバのエントリーとエグジット関数。エントリー関数では「i2c_add_driver」関数を呼び出して i2c デバイスドライバを追加し、エグジット関数では「i2c_del_driver」関数を呼び出して i2c デバイスドライバを削除します。

株式会社日昇テクノロジー

17.5.2.2.2 .probe 関数と.remove 関数の実装

通常、.probe 関数は初期化作業を行い、.remove 関数は終了前のクリーンアップを行います。mpu6050 に必要な初期化は少なく、文字デバイスの.open 関数内で実施されます。.probe 関数では文字デバイスの追加と登録のみを行います。以下がそのソースコードです：

リスト 28: mpu6050 ドライバの.probe と.remove 関数の実装

```
1 static int mpu6050_probe(struct i2c_client *client, const struct i2c_device_id *id)
2 {
3    int ret = -1; // エラーステータスコードの保存
4    printk(KERN_EMERG "¥t match successed ¥n");
5    // 動的割り当て方式でデバイス番号を取得、サブデバイス番号は 0
6    ret = alloc_chrdev_region(&mpu6050_devno, 0, DEV_CNT, DEV_NAME);
7    if (ret < 0)
8    {
9       printk("fail to alloc mpu6050_devno¥n");
10       goto alloc_err;
11    }
12    ...
13 }
14
15 static int mpu6050_remove(struct i2c_client *client)
16 {
17    /* デバイスの削除 */
```

株式会社日昇テクノロジー

```
18    device_destroy(class_mpu6050, mpu6050_devno); // デバイスのクリア
19    class_destroy(class_mpu6050); // クラスのクリア
20    cdev_del(&mpu6050_chr_dev); // デバイス番号のクリア
21    unregister_chrdev_region(mpu6050_devno, DEV_CNT); // キャラクタデバイスの登録解除
22    return 0;
23 }
```

- .probe 関数は文字デバイスを単に登録しています。文字デバイスの登録は以前のドライバプログラムで何度も使用されているため、ここでは詳細は述べません。

- .remove 関数は文字デバイスの登録を解除します。

17.5.2.2.3 文字デバイス操作関数の実装

.prob 関数で文字デバイスを追加したので、mpu6050 の初期化と結果読み取りは、この文字デバイスの操作関数で実装されます。主に、.open と.read 関数です。以下がこれら二つの関数の実装です。

.open 関数の実装（.open 関数で mpu6050 を設定します）：

リスト 29: open 関数の実装

```
1 /* 文字デバイス操作関数集、open 関数実装*/
2 static int mpu6050_open(struct inode *inode, struct file *filp)
3 {
4 // printk("¥n mpu6050_open ¥n");
5 /* mpu6050 に設定データを送信し、mpu6050 を正常動作状態にする*/
6 mpu6050_init();
7 return 0;
8 }
```

株式会社日昇テクノロジー

```
9

10 /* i2c の初期化

11 * 戻り値、成功時は 0。失敗時は-1

12 */

13 static int mpu6050_init(void)

14 {

15 int error = 0;

16 /* mpu6050 を設定する*/

17 error += i2c_write_mpu6050(mpu6050_client, PWR_MGMT_1, 0X00);

18 error += i2c_write_mpu6050(mpu6050_client, SMPLRT_DIV, 0X07);

19 error += i2c_write_mpu6050(mpu6050_client, CONFIG, 0X06);

20 error += i2c_write_mpu6050(mpu6050_client, ACCEL_CONFIG, 0X01);

21

22 if (error < 0)

23 {

24 /* 初期化エラー*/

25 printk(KERN_DEBUG "¥n mpu6050_init error ¥n");

26 return -1;

27 }

28 return 0;

29 }

30
```

株式会社日昇テクノロジー

```
31

32 /* i2c を介して mpu6050 にデータを書き込む

33 *mpu6050_client：mpu6050 の i2c_client 構造体。

34 *address, 書き込み先のアドレス

35 *data, 書き込むデータ

36 * 戻り値、エラー時は-1。成功時は 0

37 */

38 static int i2c_write_mpu6050(struct i2c_client *mpu6050_client, u8 address,␣

,→u8 data)

39 {

40 int error = 0;

41 u8 write_data[2];

42 struct i2c_msg send_msg; // 送信するデータの構造体

43

44 /* 送信するデータを設定*/

45 write_data[0] = address;

46 write_data[1] = data;

47

48 /* 書き込み先のアドレス reg を iic で送信*/

49 send_msg.addr = mpu6050_client->addr; // iic バス上の mpu6050 のアドレス

50 send_msg.flags = 0; // データ送信のマーク

51 send_msg.buf = write_data; // 書き込むデータの先頭アドレス
```

株式会社日昇テクノロジー

```
52 send_msg.len = 2; // reg の長さ

53

54 /* 送信を実行*/

55 error = i2c_transfer(mpu6050_client->adapter, &send_msg, 1);

56 if (error != 1)

57 {

58 printk(KERN_DEBUG "¥n i2c_transfer error ¥n");

59 return -1;

60 }

61 return 0;

62 }
```

- 第 2 行：open 関数では、自分で作成した mpu6050_init 関数のみを呼び出しています。

- 第 13-29 行：mpu6050 に制御パラメータを送信し、これらのパラメータはチップのマニュアルを参照してください。i2c_write_mpu6050 関数の実装に注目します。

- 第 33 行：mpu6050_client は i2c_client 型の構造体で、mpu6050 デバイスに対応する i2c_client 構造体を指定します。

- 第 34 行：address パラメータは、mpu6050 の内部アドレスに書き込むアドレスを指定します。

- 第 35 行：data パラメータは、書き込むデータを指定します。

- 第 42 行：struct i2c_msg 構造体を定義し、送信するデータを格納します。

- 第 45-46 行：データを書き込む際には、先にアドレスを送信し、その後でデータを送信します。ここでは、アドレスとデータを格納するために 2 つの要素を持つ配列を使用しています。

- 第 49-52 行：i2c_msg 構造体に i2c バス上のアドレス、データ送信のフラグ、先頭アドレス、レジスタ長を設定します。

株式会社日昇テクノロジー

- 第 55 行：i2c_write_mpu6050 関数は、i2c_transfer 関数をラップするもので、i2c_transfer はシステムが提供する i2c デバイスドライバの送信関数です。この関数は、以前説明した通り、最終的には i2c バスドライバ内の関数を呼び出し、送受信作業を実行します。

リスト 30: read 関数の実装

(linux_driver/I2c_MPU6050/i2c_mpu6050.c)

```
1 /* 文字デバイス操作関数集、.read 関数実装*/
2 static ssize_t mpu6050_read(struct file *filp, char __user *buf, size_t cnt, loff_t *off)
3 {
4
5 char data_H;
6 char data_L;
7 int error;
8 short mpu6050_result[6]; // mpu6050 から変換された生データを保存
9
10 i2c_read_mpu6050(mpu6050_client, ACCEL_XOUT_H, &data_H, 1);
11 i2c_read_mpu6050(mpu6050_client, ACCEL_XOUT_L, &data_L, 1);
12 mpu6050_result[0] = data_H << 8;
13 mpu6050_result[0] += data_L;
14
15 i2c_read_mpu6050(mpu6050_client, ACCEL_YOUT_H, &data_H, 1);
16 i2c_read_mpu6050(mpu6050_client, ACCEL_YOUT_L, &data_L, 1);
17 mpu6050_result[1] = data_H << 8;
```

株式会社日昇テクノロジー

```
18 mpu6050_result[1] += data_L;

19

20 i2c_read_mpu6050(mpu6050_client, ACCEL_ZOUT_H, &data_H, 1);

21 i2c_read_mpu6050(mpu6050_client, ACCEL_ZOUT_L, &data_L, 1);

22 mpu6050_result[2] = data_H << 8;

23 mpu6050_result[2] += data_L;

24

25 i2c_read_mpu6050(mpu6050_client, GYRO_XOUT_H, &data_H, 1);

26 i2c_read_mpu6050(mpu6050_client, GYRO_XOUT_L, &data_L, 1);

27 mpu6050_result[3] = data_H << 8;

28 mpu6050_result[3] += data_L;

29

30 i2c_read_mpu6050(mpu6050_client, GYRO_YOUT_H, &data_H, 1);

31 i2c_read_mpu6050(mpu6050_client, GYRO_YOUT_L, &data_L, 1);

32 mpu6050_result[4] = data_H << 8;

33 mpu6050_result[4] += data_L;

34

35 i2c_read_mpu6050(mpu6050_client, GYRO_ZOUT_H, &data_H, 1);

36 i2c_read_mpu6050(mpu6050_client, GYRO_ZOUT_L, &data_L, 1);

37 mpu6050_result[5] = data_H << 8;

38 mpu6050_result[5] += data_L;

39
```

株式会社日昇テクノロジー

```
40 /* 読み取ったデータをユーザースペースにコピーする*/

41 error = copy_to_user(buf, mpu6050_result, cnt);

42

43 if(error != 0)

44 {

45 printk("copy_to_user error!");

46 return -1;

47 }

48 return 0;

49 }
```

.read 関数は非常にシンプルで、主に以下の二つの部分に分けられます。焦点は i2c_read_mpu6050 関数の実装にあります。

- 第 10-38 行：i2c_read_mpu6050 関数を呼び出して mpu6050 の変換結果を読み取ります。
- 第 41 行：copy_to_user 関数を呼び出して、変換で得られたデータをユーザースペースにコピーします。

リスト 31: i2c_read_mpu6050 関数の実装

```
1 static int i2c_read_mpu6050(struct i2c_client *mpu6050_client, u8 address,   void *data, u32 length)

2 {

3 int error = 0;

4 u8 address_data = address;

5 struct i2c_msg mpu6050_msg[2];

6
```

株式会社日昇テクノロジー

```
7 /* 読み取り位置の i2c_msg を設定 */

8 mpu6050_msg[0].addr = mpu6050_client->addr; // mpu6050 の iic バス上のアドレス

9 mpu6050_msg[0].flags = 0; // データ送信をマーク

10 mpu6050_msg[0].buf = &address_data; // 書き込みの先頭アドレス

11 mpu6050_msg[0].len = 1; // 書き込みの長さ

12

13 /* 読み取りの i2c_msg */

14 mpu6050_msg[1].addr = mpu6050_client->addr; // mpu6050 の iic バス上のアドレス

15 mpu6050_msg[1].flags = I2C_M_RD; // データ読み取りをマーク

16 mpu6050_msg[1].buf = data; // 読み取ったデータの保存場所

17 mpu6050_msg[1].len = length; // 読み取りの長さ

18

19 error = i2c_transfer(mpu6050_client->adapter, mpu6050_msg, 2);

20

21 if (error != 2)

22 {

23 printk(KERN_DEBUG "¥n i2c_read_mpu6050 error ¥n");

24 return -1;

25 }

26 return 0;

27 }
```

これは、以前に説明した i2c_write_mpu6050 関数と非常に似ており、ソースコードと組み合わせて以下のように紹介されます：

株式会社日昇テクノロジー

- 第 1 行：i2c_read_mpu6050 関数のパラメータには、mpu6050 デバイスに対応する i2c_client 構造体、読み取りたい mpu6050 の内部アドレス、読み取りデータを保存する場所、読み取りの長さ（バイト単位）が含まれます。

- 第 3-5 行：読み取り作業に使用される変数と i2c_msg 構造体を定義します。読み取り作業は、読み取りたいアドレスを先に書き込んでから実際の読み取りを行う必要があります。

- 第 8-17 行：i2c_msg 構造体を初期化します。2 つの構造体を用意し、1 つ目で読み取りたいアドレスを指定し、2 つ目で実際のデータ読み取りを行います。特に、1 つ目の i2c_msg 構造体のフラグは 0（または I2C_M_RD | I2C_M_REV_DIR_ADDR）に設定し、2 つ目の i2c_msg 構造体のフラグは I2C_M_RD に設定します。

- 第 19 行：i2c_transfer 関数を呼び出し、最終的に i2c バスドライバによって送受信作業が実行されます。

17.5.2.3 mpu6050 テストアプリケーションの実装

ここでは、ドライバが正常に機能しているかをテストするために、シンプルなテストアプリケーションを作成します。非常に簡単で、開く、読み取る、印刷するだけです。

テストコードは以下の通りです。

リスト 32: mpu6050 テストプログラム

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <fcntl.h>
4 #include <string.h>
5 #include <stdlib.h>
6 int main(int argc, char *argv[])
```

株式会社日昇テクノロジー

```
7 {
8 short receive_data[6];
9 printf("led_tiny test¥n");
10
11 /* ファイルを開く */
12 int fd = open("/dev/I2C1_mpu6050", O_RDWR);
13 if(fd < 0)
14 {
15 printf("ファイルのオープンに失敗しました：%s ¥n", argv[0]);
16 return -1;
17 }
18
19 /* データを読み取る */
20 int error = read(fd, receive_data, 12);
21 if(error < 0)
22 {
23 printf("ファイル読み取りエラー！¥n");
24 close(fd);
25 /* 閉じることが成功したかどうかを判断 */
26 }
27
28 printf("AX=%d, AY=%d, AZ=%d ", (int)receive_data[0], (int)receive_data[1],(int)receive_data[2]);
```

株式会社日昇テクノロジー

```
29 printf("GX=%d, GY=%d, GZ=%d ¥n ¥n", (int)receive_data[3], (int)receive_data[4],

(int)receive_data[5]);

30

31 /* ファイルを閉じる */

32 error = close(fd);

33 if(error < 0)

34 {

35 printf("ファイルを閉じる際のエラー！¥n");

36 }

37

38 return 0;

39 }
```

- 第 8 行：mpu6050 から得た生データを保存します。順に AX（x 軸の角度）、AY、AZ、GX（x 軸の加速度）、GY、GZ です。

- 第 12-17 行：MPU6050 デバイスファイルを開きます。

- 第 20-29 行：センサーからデータを読み取り、表示します。

このテストアプリケーションはシンプルで、ドライバの.read 関数で毎回 6 つの short 型データ（AX, AY, AZ、GX, GY, GZ）を読み取るため、アプリケーションでも同じ量を読み取る必要があります。

17.5.3 実験準備

17.5.3.1 デバイスツリープラグインのコンパイル

linux_driver/I2c_MPU6050/lubancat-mpu6050-overlay.dts をカーネルソースコードの /arch/arm64/boot/dts/rockchip/overlays/ディレクトリにコピーします。

株式会社日昇テクノロジー

rk3588 の場合は以下のコマンドでデバイスツリープラグインをコンパイルします。

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- lubancat_linux_rk3588_defconfig

make ARCH=arm64 -j4 CROSS_COMPILE=aarch64-linux-gnu- dtbs
```

コンパイル成功後、生成されたデバイスツリープラグインファイル(lubancat-mpu6050-overlay.dtbo)はソースコードディレクトリのカーネルソースコード/arch/arm64/boot/dts/rockchip/overlays/に位置します。

17.5.3.2 ドライバプログラムとアプリケーションのコンパイル

linux_driver/ をカーネルソースコードと同レベルのディレクトリにコピーし、I2c_MPU6050/ディレクトリに移動して make コマンドを実行します。これにより、i2c_mpu6050.ko と 6050_test_app が生成されます。

17.5.4 プログラムの実行結果

17.5.4.1 デバイスツリープラグインのロード

SCP、NFS、または sftp を使用してコンパイル済みのデバイスツリープラグインを開発ボードにコピーします。デバイスツリープラグイン lubancat-mpu6050-overlay.dtbo を/boot/dtb/overlay/ディレクトリにコピーします。

この実験では lubancat4 を例に、/boot/ueEnv ディレクトリの uEnvLubancat4.txt ファイルを開き、以下の内容を変更します。

株式会社日昇テクノロジー

```
uname_r=4.19.232
size=0x1000000
#dtb=rk3568-lubancat2.dtb

enable_uboot_overlays=1
#overlay_start

#dtoverlay=/dtb/overlay/lubancat-i2c3-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-i2c5-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm8-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm9-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm10-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm14-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-spi3-m1-gpio-cs-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-spi3-m1-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-uart3-m1-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-button-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-button-input-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm0-m1-demo-overlay.dtbo
dtoverlay=/dtb/overlay/lubancat-mpu6050-overlay.dtbo   添加

#overlay_end
```

開発ボードを再起動すると、デバイスツリープラグインが正常にロードされます。

```
cat@lubancat:/proc/device-tree/i2c@fe5c0000$ ls
'#address-cells'   clocks       mpu6050@68   pinctrl-0       status
'#size-cells'      compatible   name         pinctrl-names
 clock-names       interrupts   phandle      reg
cat@lubancat:/proc/device-tree/i2c@fe5c0000$ cat status
okaycat@lubancat:/proc/device-tree/i2c@fe5c0000$
cat@lubancat:/proc/device-tree/i2c@fe5c0000$
```

17.5.4.2 テスト結果

以前にコンパイルした i2c_mpu6050.ko ドライバとテストアプリケーションを開発ボードにアップロードします。

i2c_mpu6050.ko をロードし、mpu6050 の姿勢を変更して 6050_test_app を実行すると、以下のような結果が表示されます。

```
cat@lubancat:~$ sudo insmod i2c_mpu6050.ko
[  176.703420]    match successed
cat@lubancat:~$ sudo ./6050_test_app
AX=0, AY=0, AZ=0      GX=0, GY=0, GZ=217

cat@lubancat:~$ sudo ./6050_test_app
AX=4590, AY=-5308, AZ=-14556      GX=-425, GY=-33, GZ=-115

cat@lubancat:~$ sudo ./6050_test_app
AX=10584, AY=-4392, AZ=-11182      GX=-210, GY=644, GZ=34

cat@lubancat:~$ sudo ./6050_test_app
AX=-8642, AY=-5200, AZ=-12562      GX=517, GY=121, GZ=909
```

株式会社日昇テクノロジー

注意：ここで取得されるのは生データなので、大きな変動があるのは正常です。

第 18 章 SPI サブシステム - OLED スクリーン実験

シリアルペリフェラルインターフェース（Serial Peripheral Interface）は、SPI と略され、高速で、全二重、同期通信バスです。そして、チップのピン上でわずか 4 本の線を使用するため、チップのピンを節約します。この章では、SPI に関連する基本知識、カーネルの SPI フレームワーク、および SPI インターフェースの OLED ディスプレイを例に、SPI ドライバプログラムの作成について説明します。

この章は主に 5 つの部分から成ります。

- 第一部分、SPI ドライバの基本知識、SPI 物理バス、タイミング、モードについて簡単に説明します。
- 第二部分、SPI ドライバフレームワークと後に使用する主要なデータ構造を分析します。
- 第三部分、SPI バスドライバ、SPI コア層、および SPI コントローラーを分析します。
- 第四部分、ドライバの作成時に使用される関数、例えば同期、非同期などについて説明します。
- 第五部分、実験、SPI ドライバ OLED 液晶ディスプレイ。

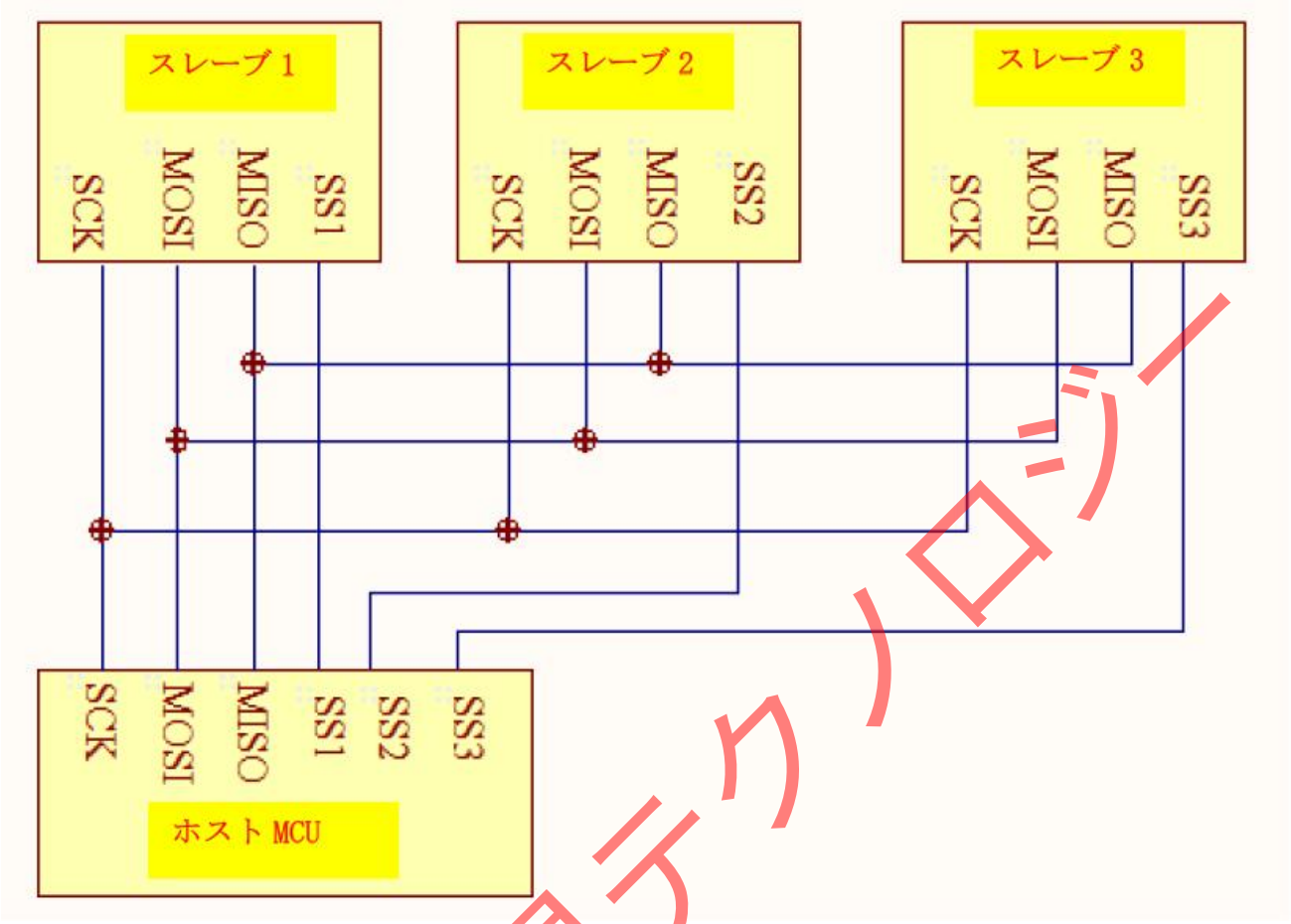
18.1 SPI 基本知識

18.1.1 SPI 物理バス

SPI バスは複数のデバイスを接続でき、標準の 1 マスター多スレーブ、全二重半二重通信などをサポートします。4 本の制御線は以下を含みます：

- SCK：クロック線、データ送受信の同期
- MOSI：データ線、マスターデバイスからのデータ送信、スレーブデバイスからのデータ受信
- MISO：データ線、スレーブデバイスからのデータ送信、マスターデバイスからのデータ受信
- NSS：チップセレクト信号線

株式会社日昇テクノロジー



I2C は I2C デバイスアドレスで通信デバイスを選択しますが、SPI はチップセレクトピンで通信デバイスを選択します。

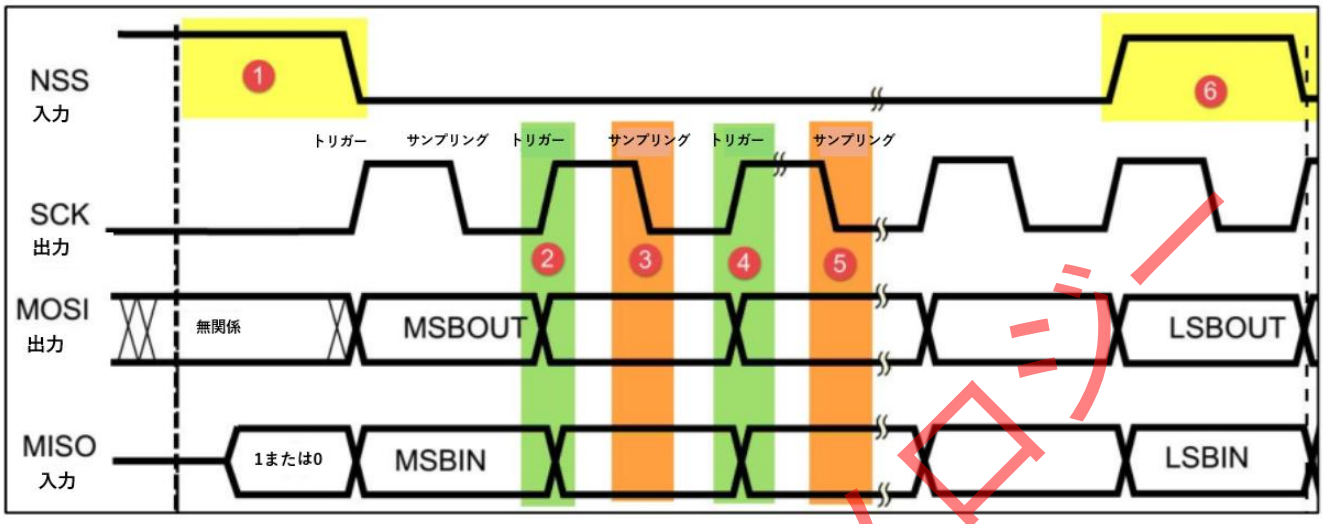
SPI インターフェースは複数のチップセレクトピンをサポートし、複数の SPI スレーブデバイスを接続できます。また、外部 GPIO を使用して SPI デバイスの数を拡張することもできます。この場合、SPI インターフェースで接続できるデバイス数はチップセレクトピンの数によって決まります。

- SPI インターフェースが提供するチップセレクトピンを使用する場合、SPI バスドライバがいつ SPI デバイスを選択するかを適切に処理します。
- 外部 GPIO をチップセレクトピンとして使用する場合は、SPI デバイスドライバ内でいつ SPI を選択するかを設定する必要があります（または SPI を設定する際に使用するチップセレクトピンを指定します）。

通常、特別な要件がない限り、SPI インターフェースが提供するチップセレクトピンを使用します。

株式会社日昇テクノロジー

18.1.2 SPI タイミング



- 開始信号：NSS 信号線が高から低に変わります。

- 停止信号：NSS 信号が低から高に変わります。

- データ転送：SCK の各クロックサイクルで MOSI と MISO が同時に 1 ビットのデータを転送します。ビットの高/低の転送には硬い規定はありません。

- 伝送単位：8 ビットまたは 16 ビット

- 単位数量：無限のデータ転送が許可されます

18.1.3 SPI 通信モード

バスがアイドル状態の時の SCK のクロック状態とデータのサンプリング時刻に基づき、SPI の動作モードは 4 種類に分けられます：

SPI モード	CPOL	CPHA	アイドル時の SCK クロック	サンプリングタイミング
0	0	0	低電圧レベル	奇数エッジ
1	0	1	低電圧レベル	偶数エッジ
2	1	0	高電圧レベル	奇数エッジ
3	1	1	高電圧レベル	偶数エッジ

株式会社日昇テクノロジー

- クロック極性 CPOL：SPI 通信デバイスがアイドル状態にある時の SCK 信号線の電平信号：

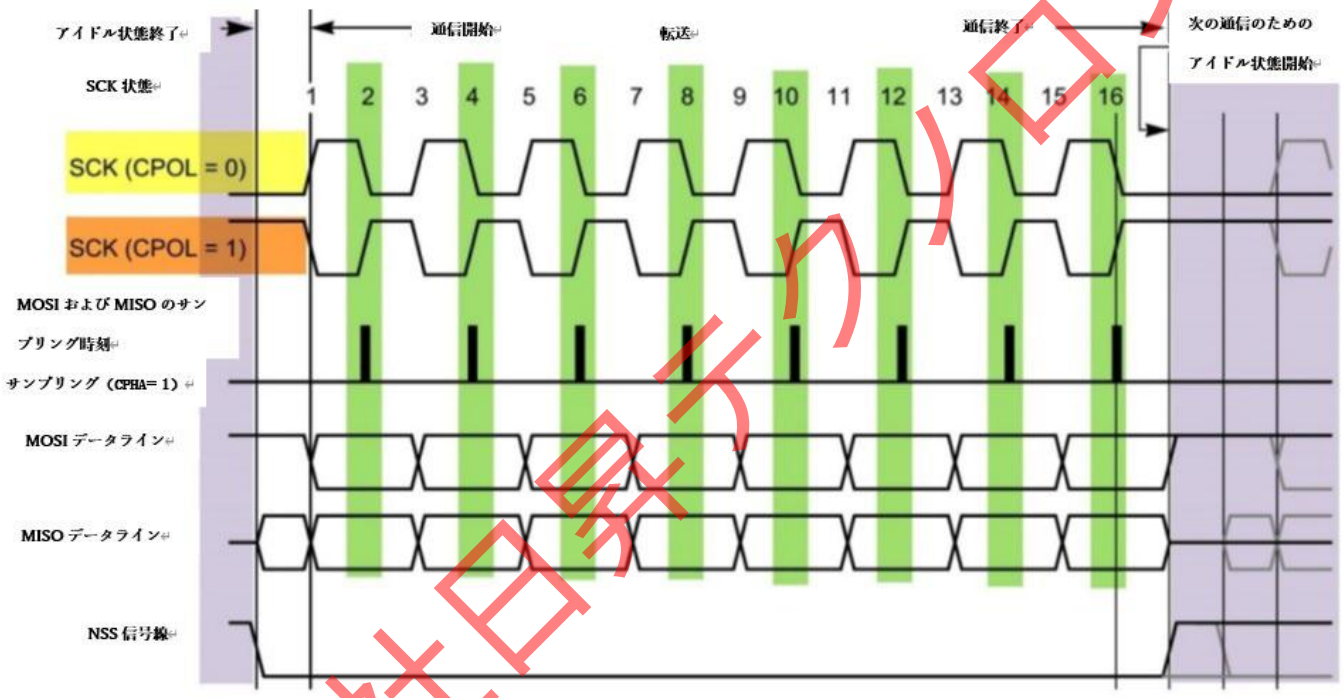
  - CPOL=0 の時、SCK はアイドル状態で低電平です。

  - CPOL=1 の時、SCK はアイドル状態で高電平です。

- クロック相位 CPHA：データのサンプル時刻：

  - CPHA=0 の時、データは SCK クロック線の「奇数エッジ」でサンプルされます。

  - CPHA=1 の時、データは SCK クロック線の「偶数エッジ」でサンプルされます。



上の図のように：

SCK 信号線がアイドル状態で低電位の場合、CPOL=0；アイドル状態で高電位の場合、CPOL=1 です。

CPHA=0 の場合、データは SCK クロック線の「奇数エッジ」でサンプリングされます。CPOL=0 の場合、クロックの奇数エッジは上昇エッジで、CPOL=1 の場合は下降エッジです。

Linux カーネルでは、これらの通信モードが定義されており、一般的にモード 0 とモード 3 がよく使用されます。

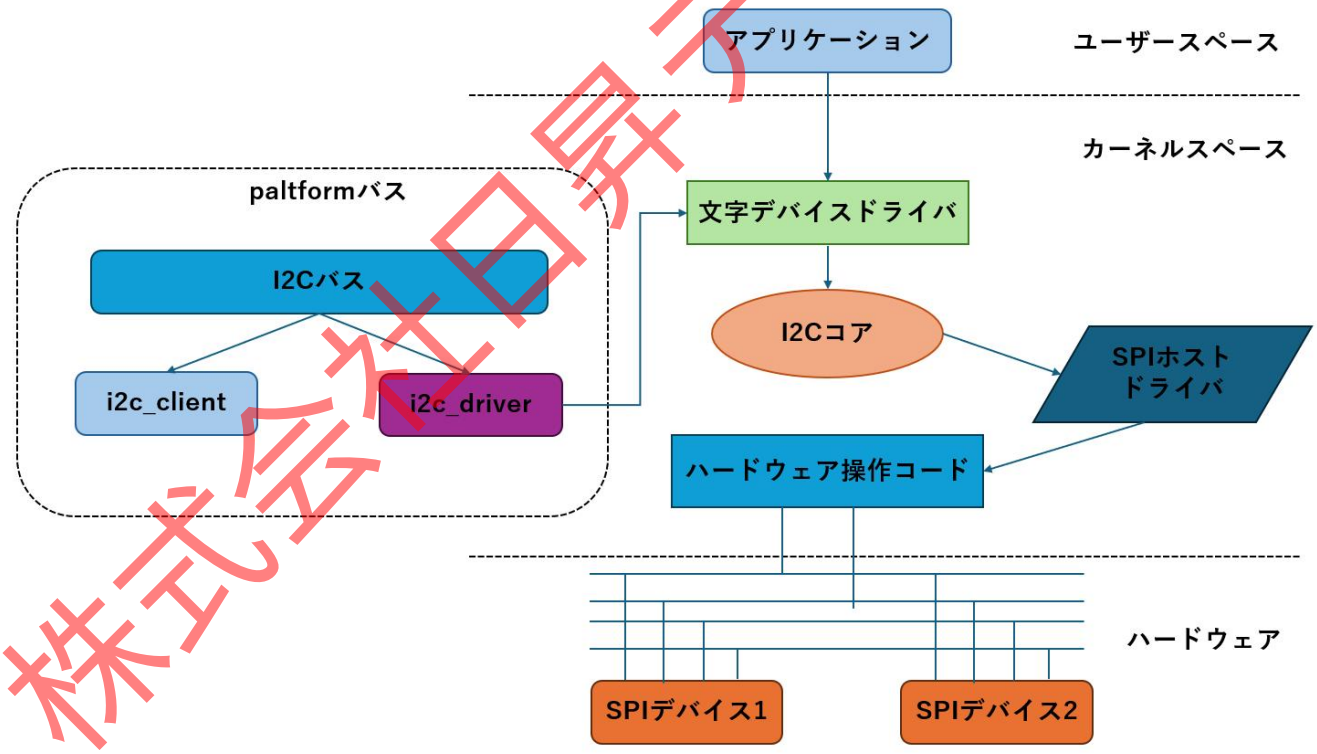
株式会社日昇テクノロジー

リスト 1: (カーネルソースコード/include/linux/spi/spi.h)

```
1 #define SPI_CPHA 0x01 /* clock phase */
2 #define SPI_CPOL 0x02 /* clock polarity */
3 #define SPI_MODE_0 (0|0) /* (original MicroWire) */
4 #define SPI_MODE_1 (0|SPI_CPHA)
5 #define SPI_MODE_2 (SPI_CPOL|0)
6 #define SPI_MODE_3 (SPI_CPOL|SPI_CPHA)
```

18.2 SPI ドライバフレームワーク

SPI デバイスドライバと I2C デバイスドライバは非常に似ており、比較学習が可能です。このセクションでは、SPI ドライバフレームワークおよび主要な構造体について紹介します。



フレームワーク図に示されているように、SPI は SPI バスドライバと SPI デバイスドライバに分けられます。SPI バスドライバはすでにチップメーカーによって提供されているため、その実装メカニズムを適切に理解する必要があります。一方、SPI デバイスドライバは自身で作成する必要があり、その原理を理

株式会社日昇テクノロジー

解する必要があります。SPI デバイスドライバには、キャラクタデバイスドライバ、SPI コア層、SPI ホストドライバが関連しています。具体的な機能は以下の通りです。

- SPI コア層: SPI コントローラドライバとデバイスドライバの登録方法、登録解除方法を提供します。SPI コアは操作インターフェース関数を提供し、SPI マスター、SPI ドライバ、SPI デバイスが SPI コア内で登録され、退出時に登録解除されるようにします。また、上位の API インターフェースも提供します。

- SPI ホストドライバ（SPI コントローラドライバ、SPI マスタードライバ）: SPI ハードウェアアーキテクチャ内のアダプター（SPI コントローラー）の制御を含み、SPI バスのハードウェアアクセス操作を実装します。

- SPI デバイスドライバ: SPI デバイス側のドライバプログラムで、SPI ホストドライバを介して CPU とデータを交換します。

18.2.1 主要なデータ構造

このセクションでは、SPI ドライバフレームワークに関連する主要なデータ構造を列挙します。まずはこれらのデータ構造を概観し、後でこれらのデータ構造に遭遇した際に、詳細な定義を再度確認してください。

18.2.1.1 spi_master

spi_master は SPI コントローラインターフェースで、実際には spi_controller 構造体です。

リスト 2：spi_controller（カーネルソースコード/include/linux/spi/spi.h、カーネルバージョン 4.19 内）

```
1 #define spi_master spi_controller
```

株式会社日昇テクノロジー

18.2.1.2 spi_controller

spi_controller 構造体の一部のメンバ変数は省略されていますが、以下に spi_controller 構造体の主要なメンバ変数を列挙します：

リスト 3：spi_controller 構造体（カーネルソースコード/include/linux/spi/spi.h）

```
1 struct spi_controller {
2 struct device dev;
3 ...
4 struct list_head list;
5 s16 bus_num;
6 u16 num_chipselect;
7 ...
8 struct spi_message *cur_msg;
9 ...
10 int (*setup)(struct spi_device *spi);
11 int (*transfer)(struct spi_device *spi,
12 struct spi_message *mesg);
13 void (*cleanup)(struct spi_device *spi);
14 struct kthread_worker kworker;
15 struct task_struct *kworker_task;
16 struct kthread_work pump_messages;
17 struct list_head queue;
18 struct spi_message *cur_msg;
```

株式会社日昇テクノロジー

```
19

20 ...

21 int (*transfer_one)(struct spi_controller *ctlr, struct spi_device *spi,struct spi_transfer *transfer);

22 int (*prepare_transfer_hardware)(struct spi_controller *ctlr);

23 int (*transfer_one_message)(struct spi_controller *ctlr,struct spi_message *mesg);

24 void (*set_cs)(struct spi_device *spi, bool enable);

25 ...

26 int *cs_gpios;

27 }
```

spi_controller には、さまざまな関数ポインタが含まれており、これらの関数ポインタは SPI コア層で使用されます。

- list: チェーンリストノード、複数の spi コントローラが存在する可能性があります。

- bus_num: SPI コントローラ番号

- num_chipselect: SPI チップセレクト信号の数、異なるスレーブデバイスを区別します。

- cur_msg: spi_message 構造体タイプ、送信されるメッセージはこの構造体に封印されます。

- transfer: データをコントローラのメッセージキューに追加するために使用されます。

- cleanup: spi_master が解放されたときに、クリーンアップ作業を完了します。

- kworker: カーネルスレッドワーカー、SPI は非同期転送方法でデータを送信できます。

- pump_messages: 具体的な転送作業を行います。

- queue: すべての転送待ちのメッセージキューがこのリストに掛けられます。

- transfer_one_message: 一つの spi メッセージを送信します。

- cs_gpios: SPI 上の具体的なチップセレクト信号を記録します。

株式会社日昇テクノロジー

18.2.1.3 spi_driver 構造体

リスト 4：spi_driver 構造体（カーネルソースコード/include/linux/spi/spi.h）

```
1 struct spi_driver {
2 const struct spi_device_id *id_table;
3 int (*probe)(struct spi_device *spi);
4 int (*remove)(struct spi_device *spi);
5 void (*shutdown)(struct spi_device *spi);
6 struct device_driver driver;
7 };
```

- id_table：spi とのペアリングに使用します。
- .probe：spi デバイスと spi ドライバが正しくマッチした後、この関数ポインタがコールバックされます

spi デバイスドライバ構造体は、以前に話した i2c デバイスドライバ構造体 i2c_driver やプラットフォームデバイスドライバ構造体 platform_driver と同じ構造を持ち、使用方法も同じです。

18.2.1.4 spi_device デバイス構造体

spi デバイス構造体は、具体的な spi デバイスを表し、その spi デバイスの詳細情報（設定情報）を保存します。ドライバとデバイスが正常にマッチした後（例えば、デバイスツリーノード）、spi_device 構造体は.prob 関数のパラメータとして取得できます。

リスト 5: spi_driver 構造体（カーネルソースコード/include/linux/spi/spi.h）

```
1 struct spi_device {
2 struct device dev;
3 struct spi_controller *controller;
```

株式会社日昇テクノロジー

```
4 struct spi_controller *master; /* compatibility layer */

5 u32 max_speed_hz;

6 u8 chip_select;

7 u8 bits_per_word;

8 u16 mode;

9 #define SPI_CPHA 0x01 /* clock phase */

10 #define SPI_CPOL 0x02 /* clock polarity */

11 #define SPI_MODE_0 (0|0) /* (original MicroWire) */

12 #define SPI_MODE_1 (0|SPI_CPHA)

13 #define SPI_MODE_2 (SPI_CPOL|0)

14 #define SPI_MODE_3 (SPI_CPOL|SPI_CPHA)

15 #define SPI_CS_HIGH 0x04 /* chipselect active high? */

16 #define SPI_LSB_FIRST 0x08 /* per-word bits-on-wire */

17 #define SPI_3WIRE 0x10 /* SI/SO signals shared */

18 #define SPI_LOOP 0x20 /* loopback mode */

19 #define SPI_NO_CS 0x40 /* 1 dev/bus, no chipselect */

20 #define SPI_READY 0x80 /* slave pulls low to pause */

21 #define SPI_TX_DUAL 0x100 /* transmit with 2 wires */

22 #define SPI_TX_QUAD 0x200 /* transmit with 4 wires */

23 #define SPI_RX_DUAL 0x400 /* receive with 2 wires */

24 #define SPI_RX_QUAD 0x800 /* receive with 4 wires */

25 int irq;
```

株式会社日昇テクノロジー

```
26 void *controller_state;

27 void *controller_data;

28 char modalias[SPI_NAME_SIZE];

29 int cs_gpio; /* chip select gpio */

30

31 /* the statistics */

32 struct spi_statistics statistics;

33 };
```

- dev: device 型の構造体。これはデバイス構造体で、SPI デバイス構造体、I2C デバイス構造体、プラットフォームデバイス構造体などはすべて「デバイス構造体」から「継承」されています。それぞれの特性に応じて独自のメンバを追加しています。SPI デバイスが追加するメンバは後ほど紹介されます。

- controller: 現在の SPI デバイスがどの SPI コントローラに接続されているか。

- master: spi_master 型の構造体。バスドライバ内で、spi_master は SPI バスを代表し、このパラメータは SPI デバイスがどの SPI バスに接続されているかを指定するために使用されます。

- max_speed_hz: SPI 通信の最大周波数を指定します。

- chip_select: SPI のチップセレクトは、異なる SPI デバイスを区別するために使用される番号で、SPI デバイスのチップセレクトピンと混同してはいけません。チップセレクトピンを指定するメンバは後述されます。

- bits_per_word: SPI 通信時の 1 バイトあたりのビット数、つまり伝送単位を指定します。

- mode: SPI の動作モード。上記コードで定義されているマクロとして、時計極性、ビット幅などを含む。これらのマクロはビット OR「|」で組み合わせて使用できます。これらのマクロは SPI プロトコルで詳しく紹介されていますが、ここでは詳細は割愛します。

株式会社日昇テクノロジー

- irq: 割り込みを使用する場合、割り込み番号を指定します。

- cs_gpio: チップセレクトピン。デバイスツリーでチップセレクトピンが設定されている場合、ドライバとデバイスツリーノードがマッチした後に自動的にチップセレクトピンを取得します。また、ドライバ内でこのパラメータを設定してチップセレクトピンをカスタム設定することも可能です。

- statistics: SPI の名前を記録し、spi_driver とマッチングするために使用されます。

18.2.1.5 spi_transfer 構造体

SPI デバイスドライバプログラム内で、spi_transfer 構造体は送信するデータを指定するために使用され、後で「転送構造体」と呼ばれます：

リスト 6: spi_transfer 構造体

```
1 struct spi_transfer {
2 /* it's ok if tx_buf == rx_buf (right?)
3 * for MicroWire, one buffer must be null
4 * buffers must work with dma_*map_single() calls, unless
5 * spi_message.is_dma_mapped reports a pre-existing mapping
6 */
7 const void *tx_buf;
8 void *rx_buf;
9 unsigned len;
10
11 dma_addr_t tx_dma;
12 dma_addr_t rx_dma;
13 struct sg_table tx_sg;
```

株式会社日昇テクノロジー

```
14 struct sg_table rx_sg;

15

16 unsigned cs_change:1;

17 unsigned tx_nbits:3;

18 unsigned rx_nbits:3;

19 #define SPI_NBITS_SINGLE 0x01 /* 1bit transfer */

20 #define SPI_NBITS_DUAL 0x02 /* 2bits transfer */

21 #define SPI_NBITS_QUAD 0x04 /* 4bits transfer */

22 u8 bits_per_word;

23 u16 delay_usecs;

24 u32 speed_hz;

25

26 struct list_head transfer_list;

27 };
```

転送構造体のメンバは多く、自分で設定する必要があるものは少ないため、ここではよく使用される設定項目のみ紹介します。

- tx_buf: 送信バッファ、送信するデータのアドレスを指定します。

- rx_buf: 受信バッファ、受信したデータを保存します。受信しない場合は設定しないか NULL に設定します。

- len: 送受信する長さで、SPI の特性により送信、受信の長さは等しいです。

- tx_dma、rx_dma: DMA を使用する場合、tx または rx の DMA アドレスを指定します。

- bits_per_word、speed_hz: それぞれ 1 バイトあたりのビット数、送信周波数を設定します。これら

株式会社日昇テクノロジー

のパラメータを設定しない場合、デフォルトの設定、つまり初期化時に設定したパラメータが使用されます。

18.2.1.6 spi_message 構造体

基本的に、spi_transfer 構造体は送信（または受信）するデータを保存し、SPI デバイスドライバ内でデータは「メッセージ」の形で送信されます。spi_message はメッセージ構造体であり、メッセージの送信は 4 ステップで行われます。具体的には、メッセージ構造体の定義、メッセージ構造体の初期化、「送信するデータ（つまり、初期化済みの spi_transfer 構造体）をバインドする」、送信の実行です。

spi_message 構造体の定義は以下の通りです：

リスト 7: spi_message 構造体

```
1 struct spi_message {
2 struct list_head transfers;
3
4 struct spi_device *spi;
5
6 unsigned is_dma_mapped:1;
7
8 /* REVISIT: we might want a flag affecting the behavior of the
9 * last transfer ... allowing things like "read 16 bit length L"
10 * immediately followed by "read L bytes". Basically imposing
11 * a specific message scheduling algorithm.
12 *
13 * Some controller drivers (message-at-a-time queue processing)
```

株式会社日昇テクノロジー

```
14 * could provide that as their default scheduling algorithm. But

15 * others (with multi-message pipelines) could need a flag to

16 * tell them about such special cases.

17 */

18

19 /* completion is reported through a callback */

20 void (*complete)(void *context);

21 void *context;

22 unsigned frame_length;

23 unsigned actual_length;

24 int status;

25

26 /* for optional use by whatever driver currently owns the

27 * spi_message ... between calls to spi_async and then later

28 * complete(), that's the spi_master controller driver.

29 */

30 struct list_head queue;

31 void *state;

32 };
```

株式会社日昇テクノロジー

spi_message 構造体のメンバは少し見慣れないかもしれませんが、具体的な送信の詳細を考慮しない場合、これらのメンバの意味を理解する必要はありません。なぜなら、spi_message の初期化および spi_transfer 転送構造体の「バインド」はカーネル関数によって実装されるからです。ただし、唯一の例外として、spi_message の二番目のメンバ「spi」は spi_device 型のポインタであり、spi_device 構造体を説明したと

きに、一つの SPI デバイスに対応する一つの spi_device 構造体があると述べました。このメンバは、メッセージがどのデバイスから来たかを指定するために使用されます。

18.2.2 SPI コア層

18.2.2.1 SPI バス登録

Linux システムは起動時に自動的に SPI バスを登録します。

リスト 8: SPI バス登録（カーネルソースコード/drivers/spi/spi.c）

```
1 static int __init spi_init(void)
2 {
3 int status;
4 ...
5 status = bus_register(&spi_bus_type);
6 ...
7 status = class_register(&spi_master_class);
8 ...
9 }
```

バスが登録されると、sys/bus 下に SPI バスが生成され、システムに新しいデバイスクラスが追加されます。その結果、sys/class/ディレクトリ下に spi_master クラスが見つかります。

18.2.2.2 SPI バス定義

spi_bus_type バスの定義は、SPI バスの登録時に使用されます。

株式会社日昇テクノロジー

リスト 9: SPI バス定義

```
1 struct bus_type spi_bus_type = {

2 .name = "spi",

3 .dev_groups = spi_dev_groups,

4 .match = spi_match_device,

5 .uevent = spi_uevent,

6 };
```

.match 関数ポインタは、SPI デバイスと SPI ドライバのマッチングルールを設定しています。具体的には spi_match_device のようになります。

18.2.2.3 spi_match_device() 関数

リスト 10: SPI バスの登録(カーネルソースコード/drivers/spi/spi.c)

```
1 static int spi_match_device(struct device *dev, struct device_driver *drv)

2 {

3 const struct spi_device *spi = to_spi_device(dev);

4 const struct spi_driver *sdrv = to_spi_driver(drv);

5

6 /* Attempt an OF style match */

7 if (of_driver_match_device(dev, drv))

8 return 1;

9

10 /* Then try ACPI */

11 if (acpi_driver_match_device(dev, drv))
```

株式会社日昇テクノロジー

```
12 return 1;

13

14 if (sdrv->id_table)

15 return !!spi_match_id(sdrv->id_table, spi);

16

17 return strcmp(spi->modalias, drv->name) == 0;

18 }
```

この関数は 4 つのマッチング方法を提供しています。デバイスツリーのマッチング方法、ACPI のマッチング方法、そして ID テーブルのマッチング方法です。これら 3 つの方法でマッチングに成功しなかった場合は、デバイス名でのペアリングを行います。

18.2.3 SPI コントローラードライバ

このセクションでは、RK3588 のコントローラーについて簡単に紹介します。RK3588 チップは 4 つの SPI コントローラーを持っており、それぞれのデバイスツリーには 4 つのノードが存在します。

リスト 11: spi3 デバイスツリーノード(RK3588.dtsi)

```
1 spi3: spi@fe640000 {

2

3 compatible = "rockchip,rk3066-spi";

4 reg = <0x0 0xfe640000 0x0 0x1000>;

5 interrupts = <GIC_SPI 106 IRQ_TYPE_LEVEL_HIGH>;

6 #address-cells = <1>;

7 #size-cells = <0>;

8 clocks = <&cru CLK_SPI3>, <&cru PCLK_SPI3>;
```

株式会社日昇テクノロジー

```
9 clock-names = "spiclk", "apb_pclk";

10 dmas = <&dmac0 26>, <&dmac0 27>;

11 dma-names = "tx", "rx";

12 pinctrl-names = "default", "high_speed";

13 pinctrl-0 = <&spi3m0_cs0 &spi3m0_cs1 &spi3m0_pins>;

14 pinctrl-1 = <&spi3m0_cs0 &spi3m0_cs1 &spi3m0_pins_hs>;

15 status = "disabled";

16 };
```

- reg：spi3 レジスタグループの開始アドレスは 0xfe640000 で、レジスタの長さは 0x1000 です。

- compatible 属性の値は、ホストドライバーとマッチします。RK3588 の SPI コントローラードライバーファイルはカーネルソースコード/drivers/spi/spi-rockchip.c で見つけることができます。

リスト 12: (カーネルソースコード/drivers/spi/spi-rockchip.c)

```
1 static const struct of_device_id rockchip_spi_dt_match[] = {

2 { .compatible = "rockchip,px30-spi", },

3 { .compatible = "rockchip,rv1108-spi", },

4 { .compatible = "rockchip,rv1126-spi", },

5 { .compatible = "rockchip,rk3036-spi", },

6 { .compatible = "rockchip,rk3066-spi", },

7 { .compatible = "rockchip,rk3188-spi", },

8 { .compatible = "rockchip,rk3228-spi", },

9 { .compatible = "rockchip,rk3288-spi", },

10 { .compatible = "rockchip,rk3368-spi", },
```

株式会社日昇テクノロジー

```
11 { .compatible = "rockchip,rk3399-spi", },

12 { },

13 };
```

ドライバーコントローラーは以下の module_platform_driver()によって登録されます：

リスト 13: (カーネルソースコード/drivers/spi/spi-rockchip.c)

```
1 static struct platform_driver rockchip_spi_driver = {

2 .driver = {

3 .name = DRIVER_NAME,

4 .pm = &rockchip_spi_pm,

5 .of_match_table = of_match_ptr(rockchip_spi_dt_match),

6 },

7 .probe = rockchip_spi_probe,

8 .remove = rockchip_spi_remove,

9 };

10

11 module_platform_driver(rockchip_spi_driver);
```

コントローラードライバーのソースコードでは、module_platform_driver()マクロ：

リスト 14: module_platform_driver() マクロ(カーネルソースコード

/include/linux/platform_device.h)

```
1 #define module_platform_driver(__platform_driver) ¥

2 module_driver(__platform_driver, platform_driver_register, ¥

3 platform_driver_unregister)
```

module_driver()は以下のように展開されます：

株式会社日昇テクノロジー

リスト 15: module_driver(カーネルソースコード/include/linux/device.h)

```
1 #define module_driver(__driver, __register, __unregister, ...) ¥
2 static int __init __driver##_init(void) ¥
3 { ¥
4 return __register(&(__driver) , ##__VA_ARGS__); ¥
5 } ¥
6 module_init(__driver##_init); ¥
```

- __driver：これは module_platform_driver()マクロの__platform_driver、つまり rockchip_spi_driver です。

- __register：platform_driver_register

- __unregister：platform_driver_unregister

- ##__VA_ARGS__：可変パラメータ

platform_driver 構造体型の spi_imx_driver 構造体変数を関数に渡し、module_platform_driver(rockchip_spi_driver)を間接的に呼び出し、platform_driver_register および platform_driver_unregister を実行し、プラットフォームドライバー関数の登録と登録解除を実現します。

“rockchip,rk3066-spi”にマッチした場合、rockchip_spi_probe 関数を呼び出して初期化を行い、デバイスツリーノード情報を取得し、SPI クロック、DMA、割り込みなどを初期化し、最後にコントローラーを登録します(カーネルソースコード/drivers/spi/spi-rockchip.c を詳細に読む)。

コードは以下の通りです(一部省略されています)：

リスト 16: rockchip_spi_probe 関数(カーネルソースコード/drivers/spi/spi-rockchip.c)

```
1 static int rockchip_spi_probe(struct platform_device *pdev)
2 {
```

株式会社日昇テクノロジー

```
3 int ret;

4 struct rockchip_spi *rs;

5 struct spi_controller *ctlr;

6 struct resource *mem;

7 struct device_node *np = pdev->dev.of_node;

8 u32 rsd_nsecs;

9 bool slave_mode;

10 struct pinctrl *pinctrl = NULL;

11

12 slave_mode = of_property_read_bool(np, "spi-slave"); // SPI がマスターデバイスかスレーブデバイス
 かを確認

13

14 if (slave_mode)

15 ctlr = spi_alloc_slave(&pdev->dev,

16 sizeof(struct rockchip_spi));

17 else

18 ctlr = spi_alloc_master(&pdev->dev,

19 sizeof(struct rockchip_spi)); // SPI マスターを割り当て

20

21 if (!ctlr)

22 return -ENOMEM;

23
```

株式会社日昇テクノロジー

```
24 platform_set_drvdata(pdev, ctlr); // ドライバデータを設定、割り当てたメモリポインタを保存
25
26 rs = spi_controller_get_devdata(ctlr);
27 ctlr->slave = slave_mode; // スレーブデバイスかどうかをマーク
28
29 /*.....*/
30
31 spi_enable_chip(rs, false);
32
33 ret = platform_get_irq(pdev, 0);
34 if (ret < 0)
35 goto err_disable_spiclk;
36
37 ret = devm_request_threaded_irq(&pdev->dev, ret, rockchip_spi_isr, NULL,
38 IRQF_ONESHOT, dev_name(&pdev->dev), ctlr);
39 if (ret)
40 goto err_disable_spiclk;
41
42 /*.....*/
43
44 pm_runtime_set_active(&pdev->dev);
45 pm_runtime_enable(&pdev->dev);
```

株式会社日昇テクノロジー

```
46
47 ctlr->auto_runtime_pm = true;
48 ctlr->bus_num = pdev->id;
49 ctlr->mode_bits = SPI_CPOL | SPI_CPHA | SPI_LOOP | SPI_LSB_FIRST | SPI_CS_HIGH;
50 if (slave_mode) {
51 ctlr->mode_bits |= SPI_NO_CS;
52 ctlr->slave_abort = rockchip_spi_slave_abort;
53 } else {
54 ctlr->flags = SPI_MASTER_GPIO_SS;
55 }
56 ctlr->num_chipselect = ROCKCHIP_SPI_MAX_CS_NUM;
57 ctlr->dev.of_node = pdev->dev.of_node;
58 ctlr->bits_per_word_mask = SPI_BPW_MASK(16) | SPI_BPW_MASK(8) | SPI_BPW_MASK(4);
59 ctlr->min_speed_hz = rs->freq / BAUDR_SCKDV_MAX;
60 ctlr->max_speed_hz = min(rs->freq / BAUDR_SCKDV_MIN, MAX_SCLK_OUT);
61
62 ctlr->set_cs = rockchip_spi_set_cs;
63 ctlr->setup = rockchip_spi_setup;
64 ctlr->cleanup = rockchip_spi_cleanup;
65 ctlr->transfer_one = rockchip_spi_transfer_one;
66 ctlr->max_transfer_size = rockchip_spi_max_transfer_size;
67 ctlr->handle_err = rockchip_spi_handle_err;
```

株式会社日昇テクノロジー

```
68

69 ctlr->dma_tx = dma_request_chan(rs->dev, "tx");

70 if (IS_ERR(ctlr->dma_tx)) {

71 /* ドライバのプロービングを遅延させる必要があるかどうかをチェック */

72 if (PTR_ERR(ctlr->dma_tx) == -EPROBE_DEFER) {

73 ret = -EPROBE_DEFER;

74 goto err_disable_pm_runtime;

75 }

76 dev_warn(rs->dev, "TX DMA チャンネルのリクエストに失敗しました¥n");

77 ctlr->dma_tx = NULL;

78 }

79

80 ctlr->dma_rx = dma_request_chan(rs->dev, "rx");

81 if (IS_ERR(ctlr->dma_rx)) {

82 if (PTR_ERR(ctlr->dma_rx) == -EPROBE_DEFER) {

83 ret = -EPROBE_DEFER;

84 goto err_free_dma_tx;

85 }

86 dev_warn(rs->dev, "RX DMA チャンネルのリクエストに失敗しました¥n");

87 ctlr->dma_rx = NULL;

88 }

89
```

株式会社日昇テクノロジー

```
90 if (ctlr->dma_tx && ctlr->dma_rx) {

91 rs->dma_addr_tx = mem->start + ROCKCHIP_SPI_TXDR;

92 rs->dma_addr_rx = mem->start + ROCKCHIP_SPI_RXDR;

93 ctlr->can_dma = rockchip_spi_can_dma;

94 }

95

96 /*.....*/

97

98 pinctrl = devm_pinctrl_get(&pdev->dev); // pinctrl を取得

99 if (!IS_ERR(pinctrl)) {

100 rs->high_speed_state = pinctrl_lookup_state(pinctrl, "high_speed"); // このピンの high_speed 状態を

 検索

101 if (IS_ERR_OR_NULL(rs->high_speed_state)) {

102 dev_warn(&pdev->dev, "high_speed pinctrl 状態がありません¥n");

103 rs->high_speed_state = NULL;

104 }

105 }

106

107 ret = devm_spi_register_controller(&pdev->dev, ctlr);

108 if (ret < 0) {

109 dev_err(&pdev->dev, "コントローラの登録に失敗しました¥n");

110 goto err_free_dma_rx;
```

株式会社日昇テクノロジー

```
111 }
112
113 return 0;
114
115 /*.....*/
116 }
```

- 第 19 行：メモリを割り当て、マスターをインスタンス化；
- 第 33-40 行：割り込み番号を取得し、割り込み関数を設定；
- 第 47-94 行：コントローラー、DMA、およびランタイム電源管理などを初期化；
- 第 107 行：devm_spi_register_controller 関数を使用して、SPI サブシステムに SPI コントローラーを登録します。

18.2.4 SPI デバイスドライバ

SPI バスドライバは、ハードウェアサプライヤーによって提供されます。その原理を理解し、学習するだけで十分です。以下に関連する関数は、SPI デバイスドライバ内で使用されます。

SPI デバイスの登録と登録解除関数は、ドライバのエントリーポイントとエグジットポイントでそれぞれ呼び出されます。これは、プラットフォームデバイスドライバや I2C デバイスドライバと同様です。

SPI デバイスの登録と登録解除関数は以下の通りです：

リスト 17: SPI デバイスの登録と登録解除関数

```
1 int spi_register_driver(struct spi_driver *sdrv)
2 static inline void spi_unregister_driver(struct spi_driver *sdrv)
```

I2C デバイスの登録と登録解除関数と比較すると、「spi」を「i2c」に置き換えるだけで I2C デバイスの登録と登録解除関数になり、使用法は同じです。

株式会社日昇テクノロジー

パラメータ：

- spi_driver 型の構造体（SPI デバイスドライバ構造体）。spi_driver 構造体は 1 つの SPI デバイスドライバを表します。

返り値：

- 成功：0
- 失敗：その他の値はエラーコード

18.2.4.1 spi_setup() 関数

この関数は SPI デバイスのチップセレクト信号、転送単位、最大転送速率などを設定します。関数内で SPI コントローラのメンバーである controller->setup()、つまり master->setup が呼び出されます。以前の関数 rockchip_spi_probe()で「ctlr->setup = rockchip_spi_setup;」と初期化されました。

リスト 18: spi_setup 関数（カーネルソースコード/drivers/spi/spi.c）

```
1 int spi_setup(struct spi_device *spi)
```

パラメータ：

- spi：spi_device 型の SPI デバイス構造体

返り値：

- 成功：0
- 失敗：その他の値はエラーコード

18.2.4.2 spi_message_init() 関数

spi_message を初期化します。

株式会社日昇テクノロジー

リスト 19: spi_message_init 関数（カーネルソースコード/include/linux/spi/spi.h）

```
1 static inline void spi_message_init(struct spi_message *m)
2 {
3     memset(m, 0, sizeof *m);
4     spi_message_init_no_memset(m);
5 }
```

パラメータ：

- m：spi_message 構造体へのポインタ。spi_message 構造体の定義と説明は、前述のキーデータ構造で見つけることができます。

返り値：なし。

18.2.4.3 spi_message_add_tail() 関数

spi_transfer 構造体を spi_message キューの末尾に追加します。

リスト 20: spi_message_init 関数（カーネルソースコード/include/linux/spi/spi.h）

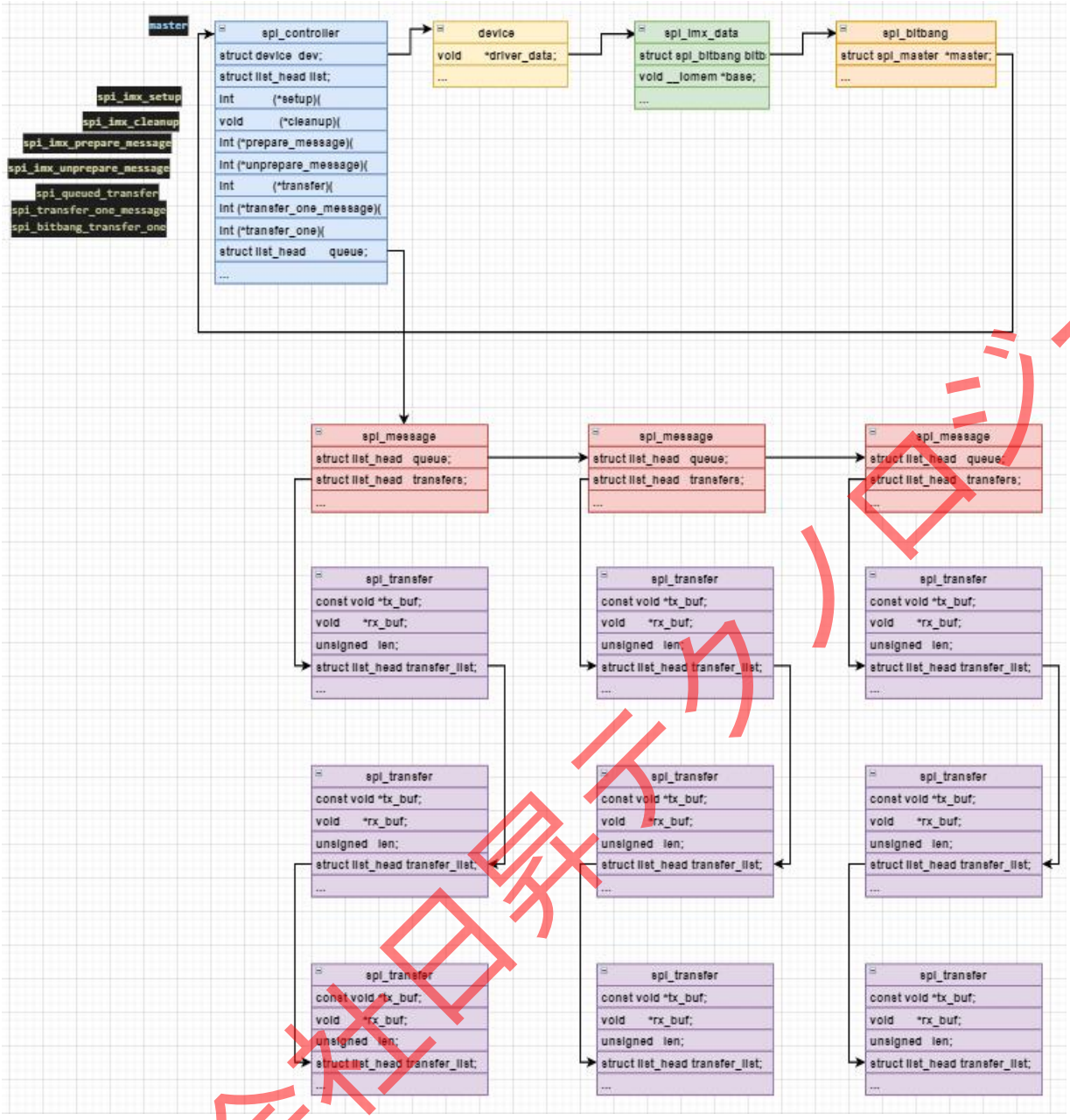
```
1 static inline void spi_message_add_tail(struct spi_transfer *t, struct spi_message *m)
2 {
3     list_add_tail(&t->transfer_list, &m->transfers);
4 }
```

この関数は、spi_transfer 構造体を spi_message キューの末尾に追加するだけです。

18.2.5 SPI 同期と排他

spi_message は、メンバー変数 queue を介して一連の spi_message を連結します。最初の spi_message は struct list_head queue の下に掛けられ、spi_message には struct list_head transfers メンバー変数もあり、transfer も連結されています。

株式会社日昇テクノロジー



18.2.5.1 SPI 同期データ転送

現在のスレッドをブロックしてデータ転送を行い、spi_sync()内部では__spi_sync()関数が呼ばれます。

mutex_lock()と mutex_unlock()は、排他ロックのロック取得と解放を行います。

株式会社日昇テクノロジー

リスト 21: spi_sync()関数（カーネルソースコード/drivers/spi/spi.c）

```
1 int spi_sync(struct spi_device *spi, struct spi_message *message)
2 {
3 int ret;
4
5 mutex_lock(&spi->controller->bus_lock_mutex);
6 ret = __spi_sync(spi, message);
7 mutex_unlock(&spi->controller->bus_lock_mutex);
8
9 return ret;
10 }
```

__spi_sync() 関数の実装は以下の通りです：

リスト 22: __spi_sync() 関数（カーネルソースコード/drivers/spi/spi.c）

```
1 static int __spi_sync(struct spi_device *spi, struct spi_message *message)
2 {
3 int status;
4 struct spi_controller *ctlr = spi->controller;
5 unsigned long flags;
6
7 status = __spi_validate(spi, message);
8 if (status != 0)
9 return status;
```

株式会社日昇テクノロジー

```
10
11 message->complete = spi_complete;
12 message->context = &done;
13 message->spi = spi;
14 ...
15 if (ctlr->transfer == spi_queued_transfer) {
16 spin_lock_irqsave(&ctlr->bus_lock_spinlock, flags);
17
18 trace_spi_message_submit(message);
19
20 status = __spi_queued_transfer(spi, message, false);
21
22 spin_unlock_irqrestore(&ctlr->bus_lock_spinlock, flags);
23 } else {
24 status = spi_async_locked(spi, message);
25 }
26
27
28 if (status == 0) {
29 ...
30 wait_for_completion(&done);
31 status = message->status;
```

株式会社日昇テクノロジー

```
32 }
33 message->context = NULL;
34 return status;
35 }
```

- 第 7-9 行：関数内部では最初に__spi_validate を呼び出して SPI の各通信パラメータを検証します。

- 第 11-13 行：message 構造体を初期化します。ここで、第 11 行では、メッセージ送信が完了した後に spi_complete コールバック関数が実行されます。

- 第 30 行：現在のスレッドをブロックし、メッセージの送信が完了したときにブロックを解除します。

18.2.5.2 SPI 非同期データ転送

リスト 23: spi_async() 関数（カーネルソースコード/drivers/spi/spi.c）

```
1 int spi_async(struct spi_device *spi, struct spi_message *message)
2 {
3 ...
4 ret = __spi_async(spi, message);
5 ...
6 }
```

ドライバープログラムで async を呼び出した時、現在のプロセスはブロックされず、現在の message 構造体を現在の SPI コントローラーのメンバー queue の末尾に追加します。その後、カーネル内に新しいワークを追加し、この message 構造体を処理する内容です。

株式会社日昇テクノロジー

リスト 24: __spi_async() 関数（カーネルソースコード/drivers/spi/spi.c）

```
1 static int __spi_async(struct spi_device *spi, struct spi_message *message)
2 {
3 struct spi_controller *ctlr = spi->controller;
4 ...
5 return ctlr->transfer(spi, message);
6 }
```

18.3 OLED スクリーンドライバ実験

spi_oled ドライバは、前節で紹介した i2c_mpu6050 デバイスドライバと非常に似ており、比較学習が推奨されます。i2c_mpu6050 ドライバから学習することをお勧めします。

本章の付属ソースコードとデバイスツリープラグインは linux_driver/SPI_OLED に位置しています。

18.3.1 ハードウェア紹介

本実験では lubancat4 ボードを例に挙げますが、他の Lubancat_RK のボードでも同様です。使用する SPI インターフェースを自身で見つける必要があります。oled モジュールは「公式【OLED スクリーン_SPI_0.96 インチ】モジュール資料」を使用しています。

18.3.1.1 ハードウェア接続

oled ドライバでは lubuncat2 の spi3 を使用します。RK3588 のデータシートを参照すると、SPI で使用されるピンは以下の通りです。

GPIO4_C2_d	PWM14_M1	SPI3_CLK_M1	CAN1_RX_M1	PCIE30X2_CLKREQn_M2	I2S3_MCLK_M1
GPIO4_C3_d	PWM15_IR_M1	SPI3_MOSI_M1	CAN1_TX_M1	PCIE30X2_WAKEn_M2	I2S3_SCLK_M1
GPIO4_C4_d	EDP_HPDIN_M0	SPDIF_TX_M2	SATA2_ACT_LED	PCIE30X2_PERSTn_M2	I2S3_LRCK_M1
GPIO4_C5_d	PWM12_M1	SPI3_MISO_M1	SATA1_ACT_LED	UART9_TX_M1	I2S3_SDO_M1
GPIO4_C6_d	PWM13_M1	SPI3_CS0_M1	SATA0_ACT_LED	UART9_RX_M1	I2S3_SDI_M1

oled スクリーンとボードのピンの対応接続は以下の表の通りです：

株式会社日昇テクノロジー

OLED 表示画面ピン	対応ボード GPIO	説明	ボードピン（ピンヘッダー）
MOSI	GPIO4_C3	MOSI ピン	MOSI
未使用		MISO ピン	
CLK	GPIO4_C2	SPI クロックピン	MCLK
D/C	GPIO3_A7	データ/コマンド制御ピン	GPIO3_A7
CS	GPIO4_C6	チップセレクトピン	CS0
GND			GND
VCC			3.3V

ヒント: lubancat4 ボードのピンアウトは「LubanCat-RK ボードクイックスタートガイド」の 40 ピンピン対照図を参照してください。

18.3.1.2 デバイスツリープラグイン

デバイスツリープラグインの記述フォーマットは変わりません。ここでは spi_oled デバイスノードに焦点を当てて説明します。

リスト 25: spi_oled デバイスツリープラグイン

(linux_driver/SPI_OLED/lubancat-spi3-m1-oled-overlay.dts)

```
1 /*
2 * Copyright (C) 2022 - All Rights Reserved by
3 * EmbedFire LubanCat
4 */
5 /dts-v1/;
6 /plugin/;
7
8 #include <dt-bindings/gpio/gpio.h>
9 #include <dt-bindings/pinctrl/rockchip.h>
```

株式会社日昇テクノロジー

```
10 #include <dt-bindings/clock/rk3568-cru.h>

11 #include <dt-bindings/interrupt-controller/irq.h>

12

13 &spi3{

14 status = "okay";

15 pinctrl-names = "default", "high_speed";

16 pinctrl-0 = <&spi3m1_cs0 &spi3m1_pins>;

17 pinctrl-1 = <&spi3m1_cs0 &spi3m1_pins_hs>;

18 cs-gpios = <&gpio4 RK_PC6 GPIO_ACTIVE_LOW>;

19

20 spi_oled@0 {

21 status = "okay";

22 compatible = "fire,spi_oled";

23 reg = <0>; //chip select 0:cs0 1:cs1

24 spi-max-frequency = <24000000>; //spi output clock

25 dc_control_pin = <&gpio3 RK_PA7 GPIO_ACTIVE_HIGH>;

26 pinctrl-names = "default";

27 pinctrl-0 = <&spi_oled_pin>;

28 };

29 };

30

31 &pinctrl {
```

株式会社日昇テクノロジー

```
32 spi_oled {
33 spi_oled_pin: spi_oled_pin {
34 rockchip,pins = <3 RK_PA7 RK_FUNC_GPIO &pcfg_pull_none>;
35 };
36 };
37 };
```

- 第 14-18 行: spi3 を有効にし、使用する pinctrl ノードを指定します。つまり、spi3 が使用するピンを指定します。詳細は以下のとおりです：

リスト 26: デバイスツリー pinctrl の説明（カーネルソースコード /arch/arm64/boot/dts/rockchip/RK3588-pinctrl.dtsi）

```
1 spi3 {
2 /*.............*/
3 spi3m1_pins: spi3m1-pins {
4 rockchip,pins =
5 /* spi3_clkm1 */
6 <4 RK_PC2 2 &pcfg_pull_none>,
7 /* spi3_misom1 */
8 <4 RK_PC5 2 &pcfg_pull_none>,
9 /* spi3_mosim1 */
10 <4 RK_PC3 2 &pcfg_pull_none>;
11 };
12
```

株式会社日昇テクノロジー

```
13 spi3m1_cs0: spi3m1-cs0 {

14 rockchip,pins =

15 /* spi3_cs0m1 */

16 <4 RK_PC6 2 &pcfg_pull_none>;

17 };

18

19 spi3m1_cs1: spi3m1-cs1 {

20 rockchip,pins =

21 /* spi3_cs1m1 */

22 <4 RK_PD1 2 &pcfg_pull_none>;

23 };

24 };

25

26 spi3-hs {

27 /*.....*/

28 spi3m1_pins_hs: spi3m1-pins {

29 rockchip,pins =

30 /* spi3_clkm1 */

31 <4 RK_PC2 2 &pcfg_pull_up_drv_level_1>,

32 /* spi3_misom1 */

33 <4 RK_PC5 2 &pcfg_pull_up_drv_level_1>,

34 /* spi3_mosim1 */
```

株式会社日昇テクノロジー

```
35 <4 RK_PC3 2 &pcfg_pull_up_drv_level_1>;

36 };

37

38 spi3m1_cs0_hs: spi3m1-cs0 {

39 rockchip,pins =

40 /* spi3_cs0m1 */

41 <4 RK_PC6 2 &pcfg_pull_up_drv_level_1>;

42 };

43

44 spi3m1_cs1_hs: spi3m1-cs1 {

45 rockchip,pins =

46 /* spi3_cs1m1 */

47 <4 RK_PD1 2 &pcfg_pull_up_drv_level_1>;

48 };

49 };
```

- 第 18 行: 使用するチップセレクトピンを指定します。ここでは GPIO4_C6 を使用しています。

- 第 21 行: spi3 ノードに spi_oled デバイスノードを追加します。

- 第 23 行: reg 属性を 0 に設定し、spi_oled を spi3 のチャネル 0 に接続します。

- 第 24 行: SPI 転送の最大周波数を設定します。実際の oled デバイスに基づいています。

- 第 25 行: spi_oled が使用する D/C 制御ピンを指定します。ドライバプログラムでは、このピンを制御して、送信するのがコマンドかデータかを設定します。

pinctrl サブシステムへのピンの追加についての具体的な内容は、GPIO サブシステムのセクションを参照してください。デバイスツリーのいくつかのピンは、spi_oled ディスプレイとのピン対応関係、ピンの

株式会社日昇テクノロジー

機能、および開発ボード上の位置を前述の表で示しています。spi_oled ディスプレイには MISO ピンがないことに注意し、直接空けておくことができます。spi_oled ディスプレイには、D/C を接続するための追加のピンが必要です。これは、SPI がデータ送信かコントロールコマンド送信かを制御するためです（高電位はデータ、低電位はコントロールコマンド）。

18.3.2 実験コード解説

18.3.2.1 プログラミングの考え方

spi_oled ドライバは、デバイスツリープラグイン方式で開発されます。主に三つの部分で構成されています。

- 第一に、spi_oled のデバイスツリープラグインを記述し、spi3 を有効にし、spi3 デバイスノードに spi_oled ノードを追加します。（ハードウェア部分は既に説明されています）。
- 第二に、spi_oled ドライバプログラムを記述します。これには、ドライバのエントリとエグジット関数の実装、.probe 関数の実装、file_operations 関数セットの実装が含まれます。
- 第三に、簡単なテストアプリケーションを記述します。

18.3.2.2 ドライバのエントリとエグジット関数の実装

ドライバのエントリとエグジット関数は、I2C_mpu6050 ドライバと似ており、i2c を spi に置き換えるだけです。ソースコードは以下の通りです。

リスト 27: ドライバエントリ関数の実装（linux_driver/SPI_OLED/spi_oled.c）

```
1 /* ID マッチングテーブルを指定 */
2 static const struct spi_device_id oled_device_id[] = {
3 {"fire,spi_oled", 0},
4 {}};
```

株式会社日昇テクノロジー

```
5

6 /* デバイスツリーマッチングテーブルを指定 */

7 static const struct of_device_id oled_of_match_table[] = {

8 {.compatible = "fire,spi_oled"},

9 {}};

10

11 /* SPI バスデバイス構造体 */

12 struct spi_driver oled_driver = {

13 .probe = oled_probe,

14 .remove = oled_remove,

15 .id_table = oled_device_id,

16 .driver = {

17 .name = "spi_oled",

18 .owner = THIS_MODULE,

19 .of_match_table = of_match_ptr(oled_of_match_table),

20 },

21 };

22

23 /*

24 * ドライバ初期化関数

25 */

26 static int __init oled_driver_init(void)
```

株式会社日昇テクノロジー

```
27 {
28 int error;
29 int ret = -1; // エラーステータスコードを保存
30 pr_info("oled_driver_init¥n");
31
32 /*--------------------キャラクタデバイス登録部分-----------------*/
33
34 // 動的にデバイス番号を割り当てる方法を使用し、サブデバイス番号は 0、
35 // デバイス名は spi_oled、cat /proc/devices コマンドで確認できます
36 // DEV_CNT は 1、現在は 1 つのデバイス番号のみを申請しています
37 ret = alloc_chrdev_region(&oled_devno, 0, DEV_CNT, DEV_NAME);
38 if (ret < 0)
39 {
40 printk("oled_devno の割り当てに失敗¥n");
41 goto alloc_err;
42 }
43
44 // キャラクタデバイス構造体 cdev とファイル操作構造体 file_operations を関連付ける
45 oled_chr_dev.owner = THIS_MODULE;
46 cdev_init(&oled_chr_dev, &oled_chr_dev_fops);
47
48 // cdev_map ハッシュテーブルにデバイスを追加
```

株式会社日昇テクノロジー

```
49 ret = cdev_add(&oled_chr_dev, oled_devno, DEV_CNT);

50 if (ret < 0)

51 {

52 printk("cdev の追加に失敗¥n");

53 goto add_err;

54 }

55

56 /* クラスを作成 */

57 class_oled = class_create(THIS_MODULE, DEV_NAME);

58

59 /* デバイス DEV_NAME を指定してデバイスを作成、 */

60 device_oled = device_create(class_oled, NULL, oled_devno, NULL, DEV_NAME);

61

62 error = spi_register_driver(&oled_driver);

63 if (error < 0) {

64 device_destroy(class_oled, oled_devno); // デバイスをクリア

65 class_destroy(class_oled); // クラスをクリア

66 cdev_del(&oled_chr_dev); // デバイス番号をクリア

67 unregister_chrdev_region(oled_devno, DEV_CNT); // キャラクタデバイスの登録を解除

68 }

69 return error;

70
```

株式会社日昇テクノロジー

```
71 add_err:
72 // デバイスの追加に失敗した場合、デバイス番号を解除する必要がある
73 unregister_chrdev_region(oled_devno, DEV_CNT);
74 printk(" エラー! ¥n");
75 alloc_err:
76
77 return -1;
78 }
79
80 /*
81 * ドライバ登録解除関数
82 */
83 static void __exit oled_driver_exit(void)
84 {
85 pr_info("oled_driver_exit¥n");
86
87 spi_unregister_driver(&oled_driver);
88 gpio_free(oled_control_pin_number);
89
90 /* デバイスを削除 */
91 device_destroy(class_oled, oled_devno); // デバイスをクリア
92 class_destroy(class_oled); // クラスをクリア
```

株式会社日昇テクノロジー

```
93 cdev_del(&oled_chr_dev); // デバイス番号をクリア
94 unregister_chrdev_region(oled_devno, DEV_CNT); // キャラクタデバイスの登録を解除
95 }
```

- 第 2-9 行：2 つのマッチングテーブルを定義しています。最初のものは従来の ID マッチングテーブル（省略可能）。二番目のものはデバイスツリーノードとマッチングするテーブルで、デバイスツリーノードの.compatible 属性と同じ値を保証する必要があります。

- 第 12-21 行：spi_driver 型の構造体を定義しています。この構造体は、i2c_driver や platform_driver と類似しています。

- 第 26-95 行：ドライバのエントリとエグジット関数です。エントリ関数では、spi ドライバとキャラクターデバイスを登録するだけで、エグジット関数ではそれらを登録解除します。

18.3.2.3 .probe 関数の実装

.prob 関数の実装では、主に 2 つの重要な作業を行います。D/C 制御用の GPIO ピンのリクエストと、SPI の初期化です。

リスト 28: .prob 関数の実装（linux_driver/SPI_OLED/spi_oled.c）

```
1 static int oled_probe(struct spi_device *spi)
2 {
3     struct device_node *node = spi->dev.of_node;
4
5     printk(KERN_EMERG "match successed ¥n");
6
7     /* OLED の D/C 制御ピンを取得し、出力に設定、デフォルトは高レベル */
8     oled_control_pin_number = of_get_named_gpio(node, "dc_control_pin", 0);
```

株式会社日昇テクノロジー

```
9
10    printk("oled_control_pin_number = %d,¥n ", oled_control_pin_number);
11
12    gpio_request(oled_control_pin_number, "dc_control_pin");
13    gpio_direction_output(oled_control_pin_number, 1);
14
15    /* SPI の初期化 */
16    oled_spi_device = spi;
17    oled_spi_device->mode = SPI_MODE_0;
18    oled_spi_device->max_speed_hz = 2000000;
19    spi_setup(oled_spi_device);
20
21    /* oled_spi_device の一部内容をプリント */
22    printk("max_speed_hz = %d¥n", oled_spi_device->max_speed_hz);
23    printk("chip_select = %d¥n", (int)oled_spi_device->chip_select);
24    printk("bits_per_word = %d¥n", (int)oled_spi_device->bits_per_word);
25    printk("mode = %02X¥n", oled_spi_device->mode);
26    printk("cs_gpio = %02X¥n", oled_spi_device->cs_gpio);
27
28    return 0;
29
30 }
```

.prob 関数の紹介は以下の通りです：

株式会社日昇テクノロジー

- 第 8-13 行：spi_oled の D/C 制御ピンを取得し、高レベルに設定します。

- 第 16 行：.prob 関数が返す spi_device 構造体を使用して、前述した通り、この構造体は SPI デバイスを表し、ここで設定された内容はデバイスツリーノードで設定された内容を上書きします。

- 第 17 行：SPI モードを SPI_MODE_0 に設定します。

- 第 18 行：最大速度を 2000000 に設定します。デバイスツリーでもこの属性が設定されていれば、ここで設定された速度が最終的な値になります。

18.3.2.4 文字デバイス操作関数セットの実装

文字デバイス操作関数セットは、ドライバーの外部インターフェースであり、これらの関数内で spi_oled の初期化、書き込み、クローズなどの作業を実装します。ここでは、.open 関数は spi_oled の初期化、.write 関数は spi_oled へのディスプレイデータの書き込み、.release 関数は spi_oled のクローズを実装します。

.open 関数の実装：

spi_oled の初期化を open 関数で行います。コードは以下の通りです。

リスト 29: .open 関数の実装（linux_driver/SPI_OLED/spi_oled.c）

```
1 /* キャラクタデバイス操作関数セット、open 関数の実装 */
2 static int oled_open(struct inode *inode, struct file *filp)
3 {
4 spi_oled_init(); //表示画面の初期化
5 return 0;
6 }
7
8 /* oled 初期化関数 */
```

株式会社日昇テクノロジー

```
9 void spi_oled_init(void)

10 {

11 /* oled を初期化 */

12 oled_send_commands(oled_spi_device, oled_init_data, sizeof(oled_init_data));

13

14 /* 画面クリア */

15 oled_fill(0x00);

16 }

17

18 static int oled_send_command(struct spi_device *spi_device, u8 *commands, u16 length)

19 {

20 int error = 0;

21 struct spi_message *message; //送信するメッセージを定義

22 struct spi_transfer *transfer; //転送構造体を定義

23

24 /* スペースを申請 */

25 message = kzalloc(sizeof(struct spi_message), GFP_KERNEL);

26 transfer = kzalloc(sizeof(struct spi_transfer), GFP_KERNEL);

27 /* D/C ピンを低電位に設定 */

28 gpio_direction_output(oled_control_pin_number, 0);

29 /* message と transfer 構造体を埋める */

30 transfer->tx_buf = commands;
```

株式会社日昇テクノロジー

```
31 transfer->len = length;

32 spi_message_init(message);

33 spi_message_add_tail(transfer, message);

34 error = spi_sync(spi_device, message);

35

36 kfree(message);

37 kfree(transfer);

38 if (error != 0)

39 {

40 printk("spi_sync エラー！¥n");

41 return -1;

42 }

43 return error;

44 }
```

上記のコードに示されているように、open 関数はカスタムの spi_oled_init 関数を呼び出し、spi_oled_init 関数では最終的に oled_send_command 関数を呼び出して spi_oled を初期化し、画面をクリアします。ここでは oled_send_command 関数に焦点を当てます：

- 第 21、22 行：spi_message 構造体と spi_transfer 構造体を定義します。

- 第 25、26 行：カーネルスタックのスペースを節約するために、これらの構造体にスペースを割り当てるために kzalloc を使用します。これらの構造体は約 100 バイトを占めるため、このような実装が推奨されます。

- 第 28 行：D/C ピンを低レベルに設定します。以前説明したように、spi_oled の D/C ピンは、送信されるのがコマンドかデータかを制御するために使用されます。低レベルではコマンドが送信され

株式会社日昇テクノロジー

ます。

- 第 30-34 行：ここでは以前説明した送信プロセスを順に実行します。spi_transfer 構造体に送信するデータを指定し、メッセージ構造体を初期化し、メッセージ構造体をキューの末尾に追加し、spi_sync 関数を呼び出して同期送信を行います。

- 第 36-43 行：スペースを解放します。

.write 関数の実装：

.write 関数は、アプリケーションからのデータを受け取り、これらのデータを表示します。関数の実装は以下の通りです。

リスト 30: .write 関数の実装（linux_driver/SPI_OLED/spi_oled.c）

```
1 /* キャラクタデバイス操作関数セット、.write 関数の実装 */
2 static int oled_write(struct file *filp, const char __user *buf, size_t cnt, loff_t *off)
3 {
4 int copy_number=0;
5 /* メモリを確保 */
6 oled_display_struct *write_data;
7 write_data = (oled_display_struct*)kzalloc(cnt, GFP_KERNEL);
8 copy_number = copy_from_user(write_data, buf, cnt);
9 oled_display_buffer(write_data->display_buffer, write_data->x, write_data->y, write_data->length);
10 /* メモリを解放 */
11 kfree(write_data);
12 return 0;
13 }
```

株式会社日昇テクノロジー

```
14

15 static int oled_display_buffer(u8 *display_buffer, u8 x, u8 y, u16 length)

16

17 /* データ送信構造体 */

18 typedef struct oled_display_struct

19 {

20 u8 x;

21 u8 y;

22 u32 length;

23 u8 display_buffer[];

24 } oled_display_struct;
```

コードの説明は以下の通りです：

- 第 2 行：.write 関数は、特に buf パラメータに注目します。これはアプリケーションからのデータのアドレスを保存しており、これらのデータをカーネル空間にコピーして使用する必要があります。cnt パラメータはデータの長さを指定します。

- 第 6 行：oled_display_struct 構造体を定義し、ユーザースペースからのデータを保存します。

- 第 7-8 行：kzalloc を使用して oled_display_struct 構造体にスペースを割り当て、アプリケーションで使用される同じ構造体に基づいています。スペースの割り当てに成功した後、copy_from_user を実行します。

- 第 9 行：自定義関数 oled_display_buffer を呼び出し、データを表示します。

- 第 11 行：空間を解放します。

- 第 15 行：関数のプロトタイプは第 4 部で示されており、`display_buffer`は表示するドットマトリックスデータを指定し x、y は表示開始位置を指定し、length は表示長を指定します。具体的な関数

株式会社日昇テクノロジー

の実装は非常にシンプルであり、ここでは詳述しません。

- 第 25 行：oled_display_struct 構造体は、カスタムの可変長構造体です。x、y パラメータはデータ表示位置を指定し、length パラメータはデータの長さを指定し、フレキシブルアレイ display_buffer[] はユーザースペースからの表示データを保存します。

.release 関数の実装:

.release 関数は、spi_oled ディスプレイに表示オフコマンドを送信する機能のみを持ちます。ソースコードは以下の通りです。

リスト 31: .release 関数の実装（linux_driver/SPI_OLED/spi_oled.c）

```
1 /* 文字デバイス操作関数セット、.release 関数の実装 */
2 static int oled_release(struct inode *inode, struct file *filp)
3 {
4     oled_send_command(oled_spi_device, 0xae); // 表示オフ
5     return 0;
6 }
```

18.3.2.5 テストアプリケーションの作成:

テストアプリケーションは主に、ドライバのテストと、oled ディスプレイでの画面更新、テキスト表示、画像表示を行います。テストプログラムは、文字と画像のドットマトリックスデータを oled_code_table.c ファイルに保存し、管理を容易にするために簡単な makefile ファイルを作成し、プログラムのコンパイルを容易にします。

その makefile ファイルは以下の通りですが、直接コマンドを使用してもコンパイルできます：

株式会社日昇テクノロジー

リスト 32: Makefile（linux_driver/SPI_OLED/test_app/Makefile.c）

```
1 out_file_name = "test_app"

2

3 all: test_app.c oled_code_table.c

4 aarch64-linux-gnu-gcc $^ -o $(out_file_name)

5

6 .PHONY: clean

7 clean:

8 rm $(out_file_name)
```

以下はテストプログラムのソースコードです：

リスト 33: テストアプリケーション Makefile（linux_driver/SPI_OLED/test_app/test_app.c）

```
1 /* ドットマトリックスデータ */

2 extern unsigned char F16x16[];

3 extern unsigned char F6x8[][6];

4 extern unsigned char F8x16[][16];

5 extern unsigned char BMP1[];

6

7 int main(int argc, char *argv[])

8 {

9     int error = -1;

10    /* ファイルオープン */

11    int fd = open("/dev/spi_oled", O_RDWR);
```

株式会社日昇テクノロジー

```
12  if (fd < 0)
13  {
14      printf("open file : %s failed !¥n", argv[0]);
15      return -1;
16  }
17
18  while(1)
19  {
20      /* 画像表示 */
21      show_bmp(fd, 0, 0, BMP1, X_WIDTH*Y_WIDTH/8);
22
23      sleep(2);
24      oled_fill(fd, 0, 0, 127, 7, 0x00); // 画面クリア
25
26      oled_show_F16X16_letter(fd,0, 0, F16x16, 4); // 漢字表示
27      oled_show_F8X16_string(fd,0,2,"F8X16:THIS IS SPI TEST APP");
28      oled_show_F6X8_string(fd, 0, 6,"F6X8:THIS IS SPI TEST APP");
29
30      sleep(2);
31      oled_fill(fd, 0, 0, 127, 7, 0x00);// 画面クリア
32
33      oled_show_F8X16_string(fd,0,0,"Testing is completed");
```

株式会社日昇テクノロジー

```
34
35      sleep(2);
36      oled_fill(fd, 0, 0, 127, 7, 0x00); // 画面クリア
37  }
38
39  /* ファイルクローズ */
40  error = close(fd);
41  if(error < 0)
42  {
43      printf("close file error! ¥n");
44  }
45
46  return 0;
47 }
```

テストプログラムは非常にシンプルで、完全なコードは対応するサンプルプログラムを参照してください。コードに関する簡単な説明は以下の通りです：

- 第 2-5 行：テストプログラムで使用されるドットマトリックスデータ。画像や文字を表示する前に、これらをドットマトリックスデータに変換する必要があります。spi_oled モジュールの付属資料には変換ツールと使用説明が提供されています。

- 第 11 行：spi_oled のデバイスノードを開きます。これは使用するドライバによって異なりますが、ここでは使用されているドライバソースコードのパスです。

- 第 18 行：画像表示テスト。表示開始位置の`x`座標は 0 に設定すべきで、これによりループ表示時に表示が乱れません。表示長はディスプレイのピクセル数を 8 で割った値であるべきです。なぜなら、

株式会社日昇テクノロジー

各バイトの 8 ビットが 8 ピクセルを制御するためです。

- 第 22-25 行：漢字やさまざまなサイズの文字の表示をテストします。

- 第 28-33 行：テスト終了のプロンプトを表示します。

18.3.3 実験準備

18.3.3.1 デバイスツリープラグインのコンパイル

デバイスツリープラグインを記述した後、コンパイルを行います。

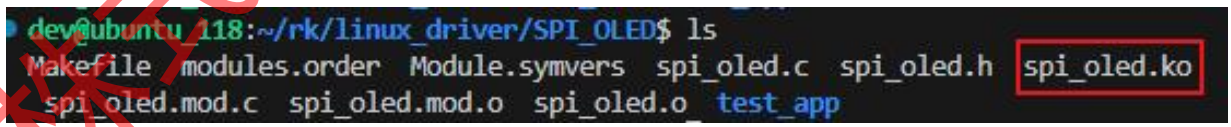
rk3588 では、カーネルソースのルートディレクトリで以下のコマンドを実行します：

```
1 make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- lubancat_linux_rk3588_defconfig
2
3 make ARCH=arm64 -j4 CROSS_COMPILE=aarch64-linux-gnu- dtbs
```

カーネルソース/arch/arm64/boot/dts/rockchip/overlays ディレクトリ下で、lubancat-spi3-m1-oled-overlay.dtbo が生成されます。最終的に生成される lubancat-spi3-m1-oled-overlay.dtbo ファイルが、OLED スクリーンのデバイスツリープラグインになります。

18.3.3.2 ドライバプログラムのコンパイル

linux_driver/SPI_OLED/をカーネルソースと同じディレクトリレベルに配置し、内部の MakeFile を実行して spi_oled.ko を生成します。



18.3.3.3 アプリケーションのコンパイル

linux_driver/SPI_OLED/test_app ディレクトリ内で MakeFile を実行し、test_app を生成します。

株式会社日昇テクノロジー

```
dev@ubuntu_118:~/rk/linux_driver/SPI_OLED/test_app$ make
aarch64-linux-gnu-gcc test_app.c oled_code_table.c -o "test_app"
dev@ubuntu_118:~/rk/linux_driver/SPI_OLED/test_app$ ls
makefile  oled_code_table.c  test_app  test_app.c  test_app.h
```

18.3.4 プログラム実行結果

生成されたデバイスツリープラグイン、ドライバプログラム、アプリケーションを scp などの方法で開発ボードにコピーします。

18.3.4.1 デバイスツリーとドライバファイルのロード

Lubancat4 ボード上で、デバイスツリープラグインをボードの/boot/dtb/overlay/ディレクトリにコピーし、/boot/uEnv ディレクトリ下の uEnvLubanCat4.txt ファイルを開いて dtoverlay=/dtb/overlay/lubancat-spi3-m1-oled-overlay.dtbo を追加し、保存後に再起動します。

```
uname_r=4.19.232
size=0x1000000
#dtb=rk3568-lubancat2.dtb

enable_uboot_overlays=1
#overlay_start

#dtoverlay=/dtb/overlay/lubancat-i2c3-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-i2c5-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm8-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm9-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm10-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-pwm14-m0-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-spi3-m1-gpio-cs-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-spi3-m1-overlay.dtbo
#dtoverlay=/dtb/overlay/lubancat-uart3-m1-overlay.dtbo
dtoverlay=/dtb/overlay/lubancat-spi3-m1-oled-overlay.dtbo

#overlay_end
~
~
~
"/boot/uEnv/uEnvLubanCat2.txt" 20L, 669C                    18,0-1
```

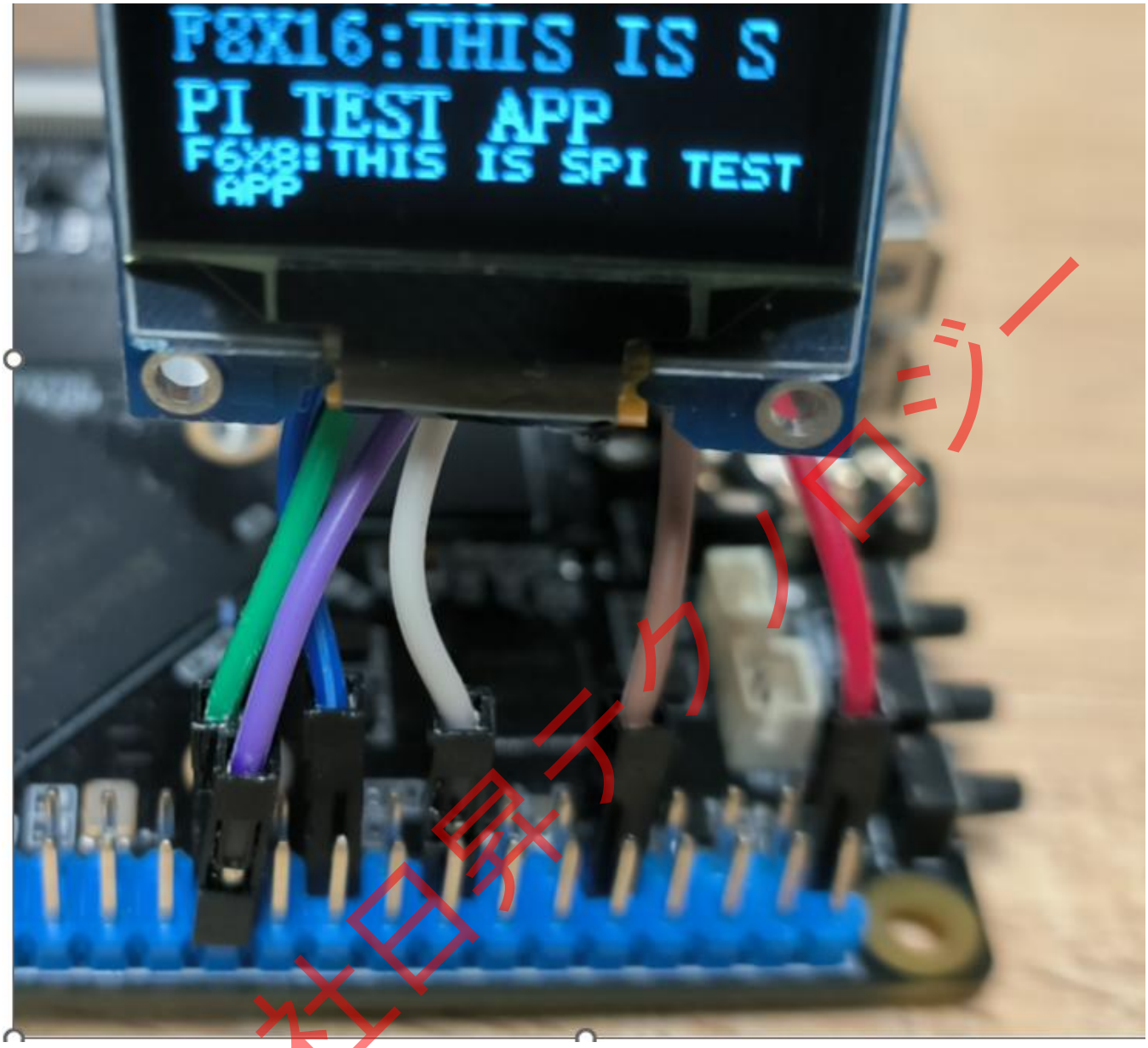
ドライバプログラムをロードするには sudo insmod spi_oled.ko コマンドを使用し、"match successed"および SPI に関連する情報が表示されます。

```
cat@lubancat:~$ sudo insmod spi_oled.ko
[43520.447026]  match successed
cat@lubancat:~$ dmesg | tail
[40372.100311] oled_driver_exit
[43520.445916] oled_driver_init
[43520.447026]  match successed
[43520.447127] oled_control_pin_number = 103,

[43520.447258] max_speed_hz = 2000000
[43520.447261] chip_select = 0
[43520.447263] bits_per_word = 8
[43520.447265] mode = 00
[43520.447266] cs_gpio = 96
cat@lubancat:~$
```

18.3.4.2 テスト結果

ドライバが正常にロードされた後、テストアプリケーションを sudo ./test_app で直接実行します。正常な場合、ディスプレイは設定した内容を表示し、自動的に切り替わります。

株式会社日昇テクノロジー



株式会社日昇テクノロジー

第 19 章 Linux 電源管理

Linux カーネルにおける電源管理（Power Management）は、システムの電源を管理する広大なシステムで、通常は使用していないときに電源を切るか、システムを低消費電力状態に切り替えます。具体的には電源状態管理と省電力管理に分けられ、電源供給、電源状態管理、実行時の電源管理、省電力管理、低消費電力などを含みます。

電源の状態管理とは、通常の起動、シャットダウン、再起動などを指します。具体的には、睡眠（Sleep）または Suspend to RAM（STR）、システムの状態情報をメモリに保存し、メモリには電源を供

給し、他を断電します。休眠（Hibernate）または Suspend to Disk（STD）は、システムの状態情報をディスクに保存し、システム全体を断電します。

再起動（Restart）やシャットダウン（Shutdown）は、システムを使用しない場合や再起動が必要な場合に使用され、reboot、halt、poweroff などのコマンドで行います。これらのコマンドは reboot システムコールを発行し、操作を実行します。

電源の省電力管理には、CPU の動的周波数変更（CPUFreq）、デバイスの動的周波数変更（DevFreq）、CPUIdle、CPU Hotplug、Runtime PM、PM QoS などがあります。CPUFreq と DevFreq は、周波数を下げることで省電力を実現し、性能と負荷を調整します。CPUIdle は、特定の CPU 上でスケジュール可能なプロセスがないときに、その CPU の電源を一時的に切ることができます。

CPU Hotplug は、特定の CPU をホットアンプラグして、その CPU にタスクが割り当てられないようにし、完全に電源をオフにできます。Runtime PM は、システム内の多くのデバイスが常に使用されているわけではないため、使用していないときにデバイスの電源を切って消費電力を削減します。

本章では、システムの Suspend と Regulator Framework について簡単に紹介します。

19.1 Suspend

ユーザー視点では、システムをスリープ（sleep）または休眠（Hibernate）させることができ、実際にはシステムコンテキストを保存してシステムをサスペンド（suspend）し、その後ウェイクアップ（wakeup）が必要です。サスペンドの詳細なプロセスは複雑で、多くのモジュールに関連していますが、ここではサスペンドに関連するユーザーインターフェースについて簡単に説明します。

Linux カーネルがサポートする休眠方式は、以下のコマンドで確認できます：

```
1 cat /sys/power/state
```

/sys/power/state ファイルは、システムを特定の電源状態（freeze, standby, mem, disk）に設定するために使用されます。これらの状態の説明は以下の通りです：

- **freeze**：

株式会社日昇テクノロジー

I/O デバイスを凍結し、低消費電力状態に設定し、プロセッサをアイドル状態にします。S2Idle 状態では、デバイスの割り込みによってウェイクアップできます。

- Standby：

I/O デバイスの凍結に加え、システムを一時停止します。コアロジックユニットが電源供給され続けるため、状態の損失は発生せず、以前の状態に容易に復帰できます。Standby 状態では、プラットフォームによってウェイクアップソースを設定する必要があります。

- mem：

動作状態データをメモリに保存し、外部デバイスをオフにして待機モードに入ります。Memory は自己リフレッシュを行いデータを保持する必要があり、他のすべてのデバイスは低消費電力状態になります。これは STR（Suspend to RAM）です。Standby で行われる操作に加え、プラットフォーム関連の操作も必要です。電源が切れるため、復帰時に再設定が必要で、ウェイクアッププロセスは遅くなります。STR 状態では、プラットフォームによってウェイクアップソースを設定する必要があります。

- disk：

実行時のコンテキストをディスクなどの不揮発性記憶デバイスに保存し、電源をオフにします。これは STD（Suspend-to-Disk）です。例えば、電源ボタンを押してウェイクアップし、その後復帰します。ウェイクアッププロセスは最も遅いです。

これら 4 つの状態では、省電力効果が順に強化され、同時にウェイクアップにかかる時間も増加します。

19.2 Regulator Framework

Regulator（調節器）は、電圧調節器（voltage regulator）と電流調節器（current regulator）に分けられ、電源管理の下層基盤の一つです。Linux における Regulator Framework は、システム内の特定のデバイスの電圧や電流供給を制御し、システムが動作している間に Regulator の出力を動的に変更して省電力を目指す設計です。このフレームワークは、電源を使用するデバイス（コンシューマ）に統一されたインターフェイスを提供し、電圧の取得、制限、有効化、無効化などの操作を可能にし、Regulator ドライバインタ

株式会社日昇テクノロジー

ーフェイスを提供して電源提供者（プロバイダ）の登録やカーネルへの操作関数の提供を可能にします。

Linux の Regulator Framework は、machine（Regulator のハードウェア制約やマッピング関係など）、regulator（理解すると Regulator ドライバ）、consumer（Regulator の使用者、サービス対象）、sys-class-regulator（ユーザースペースインターフェイス）の 4 部分に大別されます。

19.2.1 Regulator ドライバ

Regulator ドライバは、主に電源提供者（プロバイダ）の登録と関連操作関数の提供に関連します。電源提供者（プロバイダ）は、PMIC などです。以下に登録インターフェイス関数とデータ構造を紹介します：

Linux では、regulator Framework はシステム内の一部のデバイスの電圧/電流供給を制御し、システムが動作している間に regulators の出力を動的に変更して、省エネルギーを目指すために設計されています。この Regulator フレームワークは、電源を使用する各種デバイス（consumer）に統一インターフェースを提供し、電圧の取得、電圧の制限、電源の有効化と無効化などの操作を可能にします。また、Regulator ドライバーインターフェースも提供し、電源提供者（provider）の登録や、内核への操作関数の提供を可能にします。

Linux Regulator Framework は大きく 4 つの部分に分かれています。それぞれ machine（regulator のハードウェア制約、マッピング関係など）、regulator（regulator ドライバーとして理解）、consumer（regulator の使用者、サービス対象）、sys-class-regulator（ユーザースペースインターフェース）です。

19.2.1 Regulator ドライバー

Regulator ドライバーは主に電源提供者（provider）の登録と関連する操作関数を通じて、電源提供者（provider）は PMIC などです。以下に登録インターフェース関数とデータ構造について紹介します：

struct regulator_desc 構造体は、PMIC が提供する各レギュレータを記述し、各レギュレータの静的な説明を提供します。

株式会社日昇テクノロジー

リスト 1: struct regulator_desc（カーネルソースコード/include/linux/regulator/driver.h）

```
1 struct regulator_desc {

2 const char *name;

3 const char *supply_name;

4 const char *of_match;

5 const char *regulators_node;

6 /*.....*/

7 int id;

8

9 unsigned n_voltages;

10 const struct regulator_ops *ops;

11 int irq;

12 enum regulator_type type;

13 struct module *owner;

14 unsigned int min_uV;

15 unsigned int uV_step;

16 unsigned int ramp_delay;

17 /*.....*/

18 };
```

株式会社日昇テクノロジー

詳細はカーネルソースコードの/include/linux/regulator/driver.h を参照してください。詳細なコメントがあります。

- name: レギュレーターの名前

- supply_name: そのレギュレーターの親の名前。カスケード時に使用されます。

- of_match: デバイスツリー内のレギュレーター名とのマッチング

- regulators_node: DTS から init_data を自動解析

- id: レギュレーターの識別子

- n_voltages: レギュレーターが利用可能なセレクタ出力の数。固定出力電圧の場合、n_voltages を 1 に設定します。

- ops: 一連の操作関数。電源管理の操作や、レギュレーター資源の登録に使用されます。

- type: レギュレーターが電圧レギュレーターか電流レギュレーターかを示します。

- min_uV: レギュレーターの出力最小電圧

- ramp_delay: 電圧変更後に安定するまでの時間

リスト 2: 登録および登録解除インターフェース

```
1 struct regulator_dev *regulator_register(struct regulator_desc *regulator_desc,
2 const struct regulator_config *config);
3
4 void regulator_unregister(struct regulator_dev *rdev);
```

regulator_register 関数はレギュレーターを登録するインターフェースです。regulator_desc と regulator_config 二つの構造体をパラメータとして受け取り、regulator_desc はレギュレーターと関連する操作関数を記述し、regulator_config はレギュレーターの説明に関する変更可能な要素と一部の制約などを含みます。この関数は`regulator_dev`構造体を返します。この構造体はレギュレーターの抽象的な記述です。

リスト 3: regulator_dev（カーネルソースコード/include/linux/regulator/driver.h）

```
1 struct regulator_dev {
2 const struct regulator_desc *desc;
```

株式会社日昇テクノロジー

```
3 int exclusive;

4 u32 use_count;

5 u32 open_count;

6 u32 bypass_count;

7

8 /* lists we belong to */

9 struct list_head list; /* list of all regulators */

10

11 /* lists we own */

12 struct list_head consumer_list; /* consumers we supply */

13

14 struct coupling_desc coupling_desc;

15

16 struct blocking_notifier_head notifier;

17 struct mutex mutex; /* consumer lock */

18 struct task_struct *mutex_owner;

19 int ref_cnt;

20 struct module *owner;

21 struct device dev;

22 struct regulation_constraints *constraints;

23 struct regulator *supply; /* for tree */

24 const char *supply_name;
```

株式会社日昇テクノロジー

```
25 struct regmap *regmap;

26

27 struct delayed_work disable_work;

28 int deferred_disables;

29 void *reg_data; /* regulator_dev data */

30 /*.....*/

31 };
```

- list: レギュレーターリスト

- consumer_list: そのレギュレーターのすべてのコンシューマー

- notifier: レギュレーターの通知チェーン。イベント通知用。

- constraints: regulation_constraints 構造体。レギュレーターに課される一連の制約。安全制約です。

リスト 4: regulation_constraints（カーネルソースコード/include/linux/regulator/machine.h）

```
1 struct regulation_constraints {

2

3 const char *name;

4

5 /* 電圧出力範囲 */

6 int min_uV;

7 int max_uV;

8

9 int uV_offset;

10
```

株式会社日昇テクノロジー

```
11 /* 電流出力範囲 */

12 int min_uA;

13 int max_uA;

14 int ilim_uA;

15

16 int system_load;

17

18 /* カップリングされたレギュレーター用 */

19 int max_spread;

20

21 /* この  regulator  に有効な操作モードに対するフラグ */

22 unsigned int valid_modes_mask;

23

24 /* regulator  に有効な操作 */

25 unsigned int valid_ops_mask;

26

27 /* regulator  の入力電圧 - 供給源が別のレギュレーターの場合のみ */

28 int input_uV;

29

30 /*.....*/

31

32 /* 制約フラグビット */
```

株式会社日昇テクノロジー

```
33 unsigned always_on:1; /* システムがオンの時、regulator はオフにならない */
34 unsigned boot_on:1; /* bootloader/firmware が有効にした regulator */
35 unsigned apply_uV:1; /* 電圧の最大値が最小値と等しい場合、制約を適用し、固定電圧にする */
36 unsigned ramp_disable:1; /* ランプディレイを無効にする */
37 unsigned soft_start:1; /* 電圧をゆっくりと上昇させる */
38 unsigned pull_down:1; /* レギュレーターがオフの時にプルダウン抵抗を使用 */
39 unsigned over_current_protection:1; /* 過電流時に自動で無効化 */
40 }
```

この構造体は、電圧レギュレーターなどに適用される制約で、例えば電圧の出力範囲を制限します。これらは一般的にデバイスツリーで記述され、例えば`min_uA`はデバイスツリーの regulator-min-microvolt 属性に対応します。

19.2.2 コンシューマインターフェース関数

コンシューマはレギュレーターが提供するサービスの対象、使用者です。各コンシューマには以下の構造体があります。

リスト 5: regulator（カーネルソースコード/drivers/regulator/internal.h）

```
1 struct regulator {
2 struct device *dev;
3 struct list_head list;
4 unsigned int always_on:1;
5 unsigned int bypass:1;
6 int uA_load;
7 struct regulator_voltage voltage[REGULATOR_STATES_NUM];
```

株式会社日昇テクノロジー

```
8 const char *supply_name;

9 struct device_attribute dev_attr;

10 struct regulator_dev *rdev; //関連の  regulator

11 struct dentry *debugfs;

12 };
```

よく使われるコンシューマインターフェース関数:

```
1 /* 取得と解放 */

2 struct regulator *regulator_get(struct device *dev, const char *id);

3

4 void regulator_put(struct regulator *regulator);

5

6 /* 有効化と無効化 */

7 int regulator_enable(struct regulator *regulator);

8

9 int regulator_disable(struct regulator *regulator);

10

11 /* レギュレーターの電圧を設定し、レギュレーターの電圧状態を取得 */

12 int regulator_set_voltage(struct regulator *regulator, int min_uV, int max_uV);

13 int regulator_get_voltage(struct regulator *regulator);

14

15 int regulator_set_current_limit(struct regulator *regulator, int min_uA, int max_uA);

16
```

株式会社日昇テクノロジー

```
17 /* 操作モード制御と状態 */

18 int regulator_set_optimum_mode(struct regulator *regulator, int load_uA);

19

20 int regulator_set_mode(struct regulator *regulator, unsigned int mode);

21 unsigned int regulator_get_mode(struct regulator *regulator);
```

19.2.3 ユーザースペースの sysfs インターフェース

lubancat システムにログインし、/sys/class/regulator ディレクトリに移動します。

```
cat@lubancat:/sys/class/regulator$ ls -al
total 0
drwxr-xr-x  2 root root 0 Apr 21 20:54 .
drwxr-xr-x 69 root root 0 Apr 21 20:54 ..
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.0 -> ../../devices/platform/reg-dummy/regulator/regulator.0
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.1 -> ../../devices/platform/dc-5v/regulator/regulator.1
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.10 -> ../../devices/platform/cam-dovdd/regulator/regulator.10
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.11 -> ../../devices/platform/cam-avdd/regulator/regulator.11
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.12 -> ../../devices/platform/cam-dvdd/regulator/regulator.12
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.13 -> ../../devices/platform/gpio-regulator/regulator/regulator.13
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.14 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-001c/regulator/regulator.14
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.15 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-0020/regulator/regulator.15
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.16 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-0020/regulator/regulator.16
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.17 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-0020/regulator/regulator.17
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.18 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-0020/regulator/regulator.18
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.19 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-0020/regulator/regulator.19
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.2 -> ../../devices/platform/vcc5v0-sys/regulator/regulator.2
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.20 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-0020/regulator/regulator.20
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.21 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-0020/regulator/regulator.21
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.22 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-0020/regulator/regulator.22
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.23 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-0020/regulator/regulator.23
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.24 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-0020/regulator/regulator.24
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.25 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-0020/regulator/regulator.25
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.26 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-0020/regulator/regulator.26
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.27 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-0020/regulator/regulator.27
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.28 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-0020/regulator/regulator.28
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.29 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-0020/regulator/regulator.29
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.3 -> ../../devices/platform/vcc3v3-sys/regulator/regulator.3
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.30 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-0020/regulator/regulator.30
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.4 -> ../../devices/platform/vcc5v0-usb20-host-regulator/regulator/regulator.4
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.5 -> ../../devices/platform/vcc5v0-usb30-host-regulator/regulator/regulator.5
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.6 -> ../../devices/platform/vcc5v0-otg-vbus-regulator/regulator/regulator.6
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.7 -> ../../devices/platform/mini-pcie-3v3-regulator/regulator/regulator.7
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.8 -> ../../devices/platform/mipi-dsi0-lcd-power-regulator/regulator/regulator.8
lrwxrwxrwx  1 root root 0 Apr 21 20:54 regulator.9 -> ../../devices/platform/vdd-cam-5v-regulator/regulator/regulator.9
```

このディレクトリでは、登録されたレギュレーターを確認できます。これらはすべてリンクファイルであり、プラットフォームデバイスの具体的なファイルを指します。regulator.14(tcs4525)ディレクトリに移動すると、一連のファイルを確認できます。

```
cat@lubancat:/sys/class/regulator/regulator.14$ ls
cpu0-cpu        of_node                 suspend_disk_state
device          opmode                  suspend_mem_microvolts
max_microvolts  power                   suspend_mem_state
microvolts      regulator.14-SUPPLY     suspend_standby_microvolts
min_microvolts  state                   suspend_standby_state
name            subsystem               type
num_users       suspend_disk_microvolts uevent
```

- state: そのレギュレーターの状態を表し、enabled、disabled、unknown のいずれかです。

- type: レギュレーターのタイプで、電圧や電流などがあります。つまり、voltage、current、unknown です。

- microvolts: 出力される電圧を示します。min_microvolts と max_microvolts は出力可能な電圧範囲を示します。

- num_users: そのレギュレーターを使用するコンシューマーの数を示します。

- opmode: 操作モードで、通常は'normal'です。

19.3 ソースコード簡単解析

Lubancat4 を例に、RK3588 の電源供給について簡単に解説します。Lubancat4 の回路図を参照すると、RK3588 の電源供給 PMIC は rk809 と tcs4525 で、i2c で制御されます。デバイスツリーの記述は以下の通りです（一部省略）：

リスト 6: tcs4525 デバイスツリー記述（カーネルソースコード /arch/arm64/boot/dts/rockchip/RK3588-lubancat4.dts）

```
1 &i2c0 {
2     status = "okay";
3
4     vdd_cpu: tcs4525@1c {
5         compatible = "tcs,tcs452x";
6         reg = <0x1c>; //I2c アドレス
7         vin-supply = <&vcc5v0_sys>; //その親レギュレーターノード
8         regulator-compatible = "fan53555-reg"; //廃止された属性
9         regulator-name = "vdd_cpu"; //レギュレーターの名前
```

株式会社日昇テクノロジー

```
10        regulator-min-microvolt = <712500>; //最小出力電圧
11        regulator-max-microvolt = <1390000>; //最大出力電圧
12        regulator-ramp-delay = <2300>; //電圧変更後の安定化時間
13        fcs,suspend-voltage-selector = <1>; //suspend 時の電圧選択
14        regulator-boot-on; //ブートローダー/ファームウェアで有効化されたレギュレーター
15        regulator-always-on; //レギュレーターを無効化しない
16        regulator-state-mem {
17            regulator-off-in-suspend; //システム suspend 時にレギュレーターをオフにする
18        };
19    };
20    /* その他の設定... */
21 }
```

I2C0 ノード下に tcs4525 デバイスが記述され、レギュレーターとして定義されています。regulator-という接頭辞を持つフィールドは、レギュレーター固有のものです。compatible 属性は、i2c ドライバ module_i2c_driver(fan53555_regulator_driver)とマッチし、fan53555_regulator_probe()関数を呼び出して初期化し、最終的に devm_regulator_register()関数を使用してレギュレーターを登録します（詳細はカーネルソースコード/drivers/regulator/fan53555.c を参照）。

リスト 7: rk809 デバイスツリー記述（カーネルソースコード /arch/arm64/boot/dts/rockchip/RK3588-lubancat4.dts）

```
1 &i2c0 {
2 /*.....*/
3 rk809: pmic@20 {
```

株式会社日昇テクノロジー

```
4 compatible = "rockchip,rk809";

5 reg = <0x20>; //i2c アドレス

6 interrupt-parent = <& GPIO1>; //割り込み親ノード

7 interrupts = <3 IRQ_TYPE_LEVEL_LOW>; //割り込み番号割り込みタイプ

8

9 pinctrl-names = "default", "pmic-sleep",

10 "pmic-power-off", "pmic-reset";

11 pinctrl-0 = <&pmic_int>;

12 pinctrl-1 = <&soc_slppin_slp>, <&rk817_slppin_slp>;

13 pinctrl-2 = <&soc_slppin_gpio>, <&rk817_slppin_pwrdn>;

14 pinctrl-3 = <&soc_slppin_gpio>, <&rk817_slppin_rst>;

15

16 /*.....*/

17 vcc1-supply = <&vcc3v3_sys>;

18 vcc2-supply = <&vcc3v3_sys>;

19 vcc3-supply = <&vcc3v3_sys>;

20 vcc4-supply = <&vcc3v3_sys>;

21 vcc5-supply = <&vcc3v3_sys>;

22 vcc6-supply = <&vcc3v3_sys>;

23 vcc7-supply = <&vcc3v3_sys>;

24 vcc8-supply = <&vcc3v3_sys>;

25 vcc9-supply = <&vcc3v3_sys>;
```

株式会社日昇テクノロジー

```
26
27 pwrkey { //
28 status = "okay";
29 };
30
31 /*.............*/
32 regulators { //5 路の大電流 BUCK、9 つの LDO、2 つの SWITCH
33 vdd_logic: DCDC_REG1 {
34 regulator-always-on;
35 regulator-boot-on;
36 regulator-min-microvolt = <500000>;
37 regulator-max-microvolt = <1350000>;
38 regulator-init-microvolt = <900000>;
39 regulator-ramp-delay = <6001>;
40 regulator-initial-mode = <0x2>;
41 regulator-name = "vdd_logic";
42 regulator-state-mem {
43 regulator-off-in-suspend;
44 };
45 };
46
47 /*.............*/
```

株式会社日昇テクノロジー

```
48 }
49 rk809_codec: codec { //オーディオコーデック
50 #sound-dai-cells = <0>;
51 compatible = "rockchip,rk809-codec", "rockchip,rk817-codec";
52 clocks = <&cru I2S1_MCLKOUT_TX>;
53 clock-names = "mclk";
54 assigned-clocks = <&cru I2S1_MCLKOUT_TX>, <&cru I2S1_MCLK_TX_IOE>;
55 assigned-clock-rates = <12288000>;
56 assigned-clock-parents = <&cru CLK_I2S1_8CH_TX>, <&cru I2S1_MCLKOUT_TX>;
57 pinctrl-names = "default";
58 pinctrl-0 = <&i2s1m0_mclk>;
59 hp-volume = <20>;
60 spk-volume = <3>;
61 //mic-in-differential;
62 status = "okay";
63 };
64 };
65 };
```

- vcc1-supply: この一連のノードは PMIC の入力電源で、すべて vcc3v3_sys を使用しています。vcc1 は DCDC_REG1 レギュレーターの supply_name（そのレギュレーターの親ノードを指定）で、vcc3v3_sys ノードは以下のようになります：

株式会社日昇テクノロジー

リスト 8: regulator-fixed ノード（カーネルソースコード

/arch/arm64/boot/dts/rockchip/RK3588-lubancat4.dts）

```
1 dc_5v: dc-5v {
2     compatible = "regulator-fixed";
3     regulator-name = "dc_5v";
4     regulator-always-on;
5     regulator-boot-on;
6     regulator-min-microvolt = <5000000>;
7     regulator-max-microvolt = <5000000>;
8 };
9
10 vcc5v0_sys: vcc5v0-sys {
11     compatible = "regulator-fixed";
12     regulator-name = "vcc5v0_sys";
13     regulator-always-on;
14     regulator-boot-on;
15     regulator-min-microvolt = <5000000>;
16     regulator-max-microvolt = <5000000>;
17     vin-supply = <&dc_5v>;
18 };
19
20 vcc3v3_sys: vcc3v3-sys {
```

株式会社日昇テクノロジー

```
21     compatible = "regulator-fixed";
22     regulator-name = "vcc3v3_sys";
23     regulator-always-on;
24     regulator-boot-on;
25     regulator-min-microvolt = <3300000>;
26     regulator-max-microvolt = <3300000>;
27     vin-supply = <&vcc5v0_sys>;
28 };
```

RK809 は高性能な PMIC で、レギュレータ、RTC、pwrkey、コーデックなどの機能を持っています。そのため、このノードには pwrkey ノード、regulators ノード、rk809_codec ノードなど、多くのサブノードがあります。regulators サブノードには compatible 属性がなく、一連のレギュレータがどのように登録されるかが疑問になります。

compatible = "rockchip,rk809"は i2c ドライバ module_i2c_driver(rk808_i2c_driver)にマッチし rk808_probe 関数を呼び出します。この関数は i2c デバイスの初期化などを行い、devm_mfd_add_devices 関数を使用して MFD（マルチファンクションデバイス）を追加します。これにより、rk808-regulator プラットフォームデバイスが追加され、プラットフォームドライバ rk808_regulator_driver とマッチし、rk808_regulator_probe 関数でレギュレータが登録されます（詳細はカーネルソースコード drivers/mfd/rk808.c および drivers/regulator/rk808-regulator.c を参照）。

前述の regulator を登録したなら、使用者が存在します。rk3568 には、電源管理ユニット(PMU)があり、rk3568 内の電源リソースを制御し、低消費電力モードへの入力、複数の電圧源と電源ドメインのサポートなどが可能です。その中の IO 電源ドメインは、IO 出力の電圧レベルを管理し、IO 電源ドメインは一般的に PMU 電源チップの異なるグループの LDO に接続されます。この IO 電源ドメインと PMIC ハードウェアの接続については、回路図を見て、デバイスツリーの記述は以下の通りです：

株式会社日昇テクノロジー

リスト 9: （カーネルソースコード/arch/arm64/boot/dts/rockchip/RK3588-lubancat4.dts）

```
1 &pmu_io_domains {

2 status = "okay";

3 pmuio1-supply = <&vcc3v3_pmu>;

4 pmuio2-supply = <&vcc3v3_pmu>;

5 vccio1-supply = <&vccio_acodec>;

6 vccio3-supply = <&vccio_sd>;

7 vccio4-supply = <&vcc_1v8>;

8 vccio5-supply = <&vcc_3v3>;

9 vccio6-supply = <&vcc_1v8>;

10 vccio7-supply = <&vcc_3v3>;

11 };
```

- pmuio1-supply は、vcc3v3_pmu、つまり rk809 の LDO_REG6 レギュレータに接続され、固定の 3.3V 電圧を供給します。pmuio2-supply も同様です。
- vccio1-supply vccio1 電源ドメインの供給は、RK809 の LDO_REG4 に接続され、3.3V です。
- vccio3-supply は vccio_sd で、rk809 の LDO_REG5 に接続され、他のものも同様です。

システムに登録された regulator と consumer 間の関係、または regulator 間の関係は、/sys/kernel/debug/regulator/regulator_summary ファイルで確認できます。以下の画像は IO 電源ドメインと regulator 間の関係を示しています：

株式会社日昇テクノロジー

```
vdd_npu                 1    2      0   825mV     0mA   500mV  1350mV
   fde40000.npu-rknpu                                    825mV  1350mV
   vdd_npu                                                 0mV     0mV
vcc_1v8                 1    3      0  1800mV     0mA  1800mV  1800mV
   fdc20000.syscon:io-domains-vccio6
   fdc20000.syscon:io-domains-vccio4
   vcc_1v8                                                 0mV     0mV
vdda0v9_image           1    1      0   900mV     0mA   900mV   900mV
   vdda0v9_image                                           0mV     0mV
vdda_0v9                1    1      0   900mV     0mA   900mV   900mV
   vdda_0v9                                                0mV     0mV
vdda0v9_pmu             1    1      0   900mV     0mA   900mV   900mV
   vdda0v9_pmu                                             0mV     0mV
vccio_acodec            1    2      0  3300mV     0mA  3300mV  3300mV
   fdc20000.syscon:io-domains-vccio1
   vccio_acodec                                            0mV     0mV
vccio_sd                1    3      0  1800mV     0mA  1800mV  3300mV
   fe2b0000.dwmmc-vqmmc                                  1800mV  1950mV
   fdc20000.syscon:io-domains-vccio3
   vccio_sd                                                0mV     0mV
vcc3v3_pmu              1    3      0  3300mV     0mA  3300mV  3300mV
   fdc20000.syscon:io-domains-pmuio2
   fdc20000.syscon:io-domains-pmuio1
   vcc3v3_pmu                                              0mV     0mV
vcca_1v8                1    2      0  1800mV     0mA  1800mV  1800mV
   fe720000.saradc-vref                                     0mV     0mV
   vcca_1v8                                                0mV     0mV
vcca1v8_pmu             1    1      0  1800mV     0mA  1800mV  1800mV
   vcca1v8_pmu                                             0mV     0mV
vcca1v8_image           1    1      0  1800mV     0mA  1800mV  1800mV
   vcca1v8_image                                           0mV     0mV
vcc_3v3                 1    3      0  3300mV     0mA     0mV     0mV
   fdc20000.syscon:io-domains-vccio7
   fdc20000.syscon:io-domains-vccio5
   vcc_3v3                                                 0mV     0mV
vcc3v3_sd               1    2      0  3300mV     0mA     0mV     0mV
```

19.4 実験

以下では、500000μV から 1350000μV の電圧範囲を持つ regulators をカーネルに登録する簡単なドライバを書きます。対応するソースコードとデバイスツリープラグインは linux_driver/power_management にあります。

19.4.1 ドライバコード

リスト 10: regulator_test.c（linux_driver/power_management/下）

```
1 static int regulator_driver_probe(struct platform_device *pdev)

2 {

3 struct regulator_config config = { };

4 int ret;
```

株式会社日昇テクノロジー

```
5

6 config.dev = &pdev->dev;

7 config.init_data = &my_regulator_initdata;

8

9 my_regulator_test_rdev = regulator_register(&my_regulator_desc, &config);

10 if (IS_ERR(my_regulator_test_rdev)) {

11 ret = PTR_ERR(my_regulator_test_rdev);

12 pr_err("Failed to register regulator: %d¥n", ret);

13 return ret;

14 }

15

16 return 0;

17 }

18

19 static struct platform_driver my_regulator_driver = {

20 .probe = regulator_driver_probe,

21 .driver = {

22 .name = "my_regulator",

23 .owner = THIS_MODULE,

24 },

25 };

26
```

株式会社日昇テクノロジー

```
27 static struct platform_device *regulator_pdev;

28

29 static int my_regulator_test_init(void)

30 {

31 int ret;

32

33 regulator_pdev = platform_device_alloc("my_regulator", -1);

34 if (!regulator_pdev) {

35 pr_err("Failed to allocate dummy regulator device¥n");

36 return -1;

37 }

38

39 ret = platform_device_add(regulator_pdev);

40 if (ret != 0) {

41 pr_err("Failed to register dummy regulator device: %d¥n", ret);

42 platform_device_put(regulator_pdev);

43 return -1;

44 }

45

46 ret = platform_driver_register(&my_regulator_driver);

47 if (ret != 0) {

48 pr_err("Failed to register dummy regulator driver: %d¥n", ret);
```

株式会社日昇テクノロジー

```
49 platform_device_unregister(regulator_pdev);

50 return -1;

51 }

52

53 return 0;

54 }

55

56 static void my_regulator_test_exit(void)

57 {

58 regulator_unregister(my_regulator_test_rdev);

59 platform_device_unregister(regulator_pdev);

60 platform_driver_unregister(&my_regulator_driver);

61 }

62

63 module_init(my_regulator_test_init);

64 module_exit(my_regulator_test_exit);

65 MODULE_LICENSE("GPL");
```

- 第 9 行で regulator を登録、
- 第 19 行でプラットフォームドライバ my_regulator_driver を定義、
- 第 29-54 行でプラットフォームドライバとプラットフォームデバイスを登録し追加、
- 第 56-60 行でプラットフォームデバイスとプラットフォームドライバおよび regulator を登録解除。

株式会社日昇テクノロジー

19.4.2 テスト結果

デバイスツリープラグインとカーネルモジュールのコンパイルは、前の環境構築セクションを参照してください。コンパイル後に regulator_test.ko が得られます。

デバイスツリープラグインを読み込んだ後、ドライバをロードするコマンド sudo insmod regulator_test.ko を実行します。ロード後、/sys/class/regulator 下に regulator.31 のディレクトリ（このサフィックスの数字は実際のボードによって異なる）が生成されます。

```
lrwxrwxrwx  1 root root 0 Oct 11 11:22 regulator.30 -> ../../devices/platform/fdd40000.i2c/i2c-0/0-0020/regulator/regulator.30
lrwxrwxrwx  1 root root 0 Oct 11 11:28 regulator.31 -> ../../devices/platform/my_regulator/regulator/regulator.31
lrwxrwxrwx  1 root root 0 Oct 11 11:22 regulator.4 -> ../../devices/platform/vcc5v0-usb20-host-regulator/regulator/regulator.4
lrwxrwxrwx  1 root root 0 Oct 11 11:22 regulator.5 -> ../../devices/platform/vcc5v0-usb30-host-regulator/regulator/regulator.5
lrwxrwxrwx  1 root root 0 Oct 11 11:22 regulator.6 -> ../../devices/platform/vcc5v0-otg-vbus-regulator/regulator/regulator.6
lrwxrwxrwx  1 root root 0 Oct 11 11:22 regulator.7 -> ../../devices/platform/mini-pcie-3v3-regulator/regulator/regulator.7
lrwxrwxrwx  1 root root 0 Oct 11 11:22 regulator.8 -> ../../devices/platform/mipi-dsi0-lcd-power-regulator/regulator/regulator.8
lrwxrwxrwx  1 root root 0 Oct 11 11:22 regulator.9 -> ../../devices/platform/vdd-cam-5v-regulator/regulator/regulator.9
```

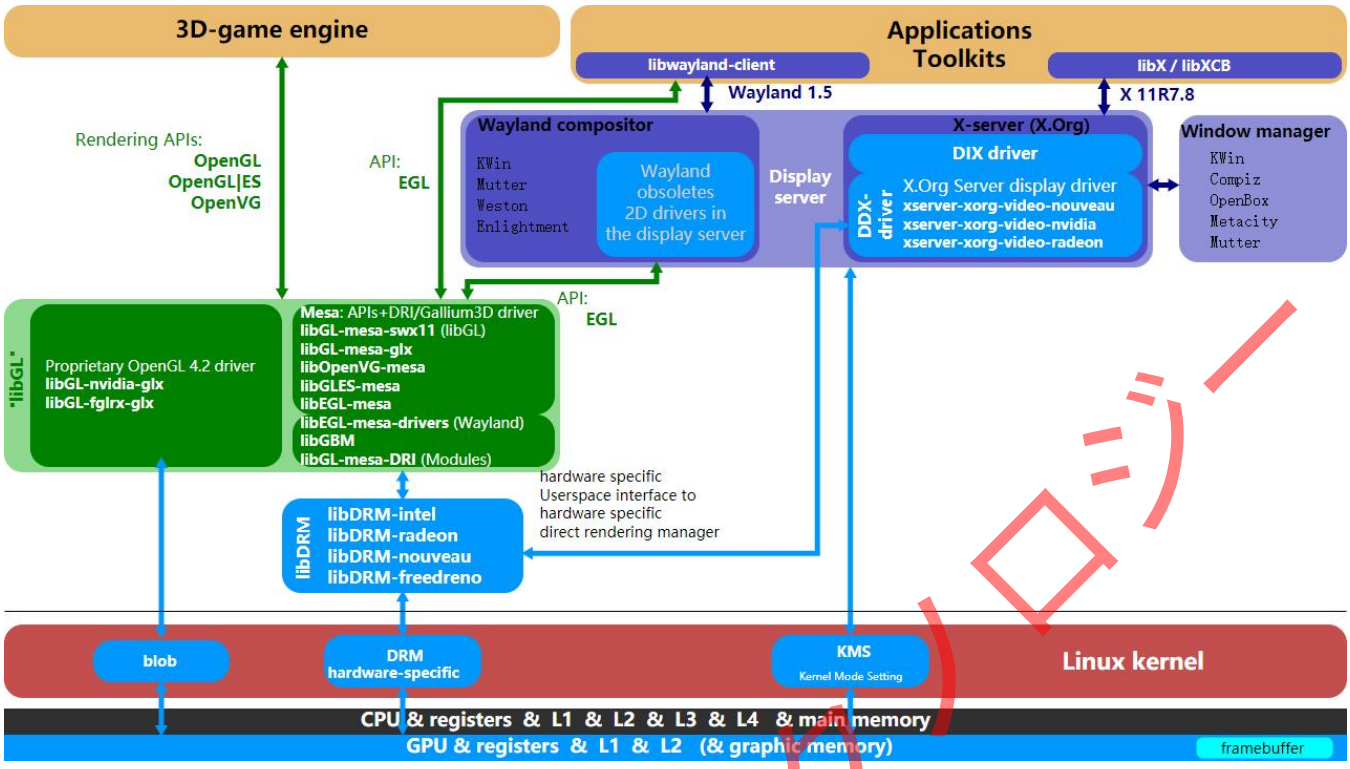
/sys/kernel/debug/regulator/regulator_summary ファイルには、システムの regulator と consumer 間の関係が記録されます。

```
pcie30_3v3                     0    2      0  3300mV     0mA   100mV  3300mV
   3c0800000.pcie-vpcie3v3                                        0mV     0mV
   pcie30_3v3                                                     0mV     0mV
my_regulator                   0    1      0     0mV     0mA   500mV  1350mV
   my_regulator                                                   0mV     0mV
```

第 20 章 DRM グラフィックス表示フレームワーク

Linux システムの画像サブシステムのソフトウェアフレームワークは複雑で、GUI（グラフィカルユーザーインターフェイス）、3D アプリケーション、DRM/KMS、ハードウェアなどが関係しています。以下の画像で簡単に理解できます（DRI フレームワーク、Direct Rendering Infrastructure についても興味があれば調べてみてください）：

株式会社日昇テクノロジー



Linux の表示デバイスドライバ開発では、通常、FBDEV（Framebuffer Device）、DRM/KMS サブシステムに注目します。Framebuffer Device ドライバフレームワークでは、簡単な表示ドライバを迅速に開発することができます。しかし、チップの表示デバイスの性能が向上し、3D レンダリングや GPU が導入されるにつれて、Framebuffer フレームワークは少し時代遅れに見えます。一番の表れは、従来のフレームワークでは、多くのチップ表示デバイスの新機能（例：表示オーバーレイ、GPU 加速、ハードウェアカーソルなど）が十分にサポートされていないことです。さらに、Framebuffer フレームワークでは、ユーザースペースの/dev/fb インターフェイスを通じて、下層のビデオメモリをユーザーに露出してしまいます。これにより、異なるアプリケーションがビデオメモリにアクセスする際に、アクセス競合が発生しやすく、このような方法は安全ではないように思えます。

この背景の下で、これらの問題を解決するための現代的なグラフィックス表示フレームワークが必要となり、そこで DRM（Direct Rendering Manager、直接描画マネージャ）が誕生しました。

20.1 DRM フレームワーク概要

DRM グラフィックス表示フレームワークは、Framebuffer フレームワークが直面する困難をどのように

株式会社日昇テクノロジー

解決するのでしょうか？DRM は、現代の表示領域で関係する操作を分層し、これらのモジュールを独立させます。例えば、上層アプリケーションがビデオメモリにアクセスしたり、表示効果や GPU を操作したい場合、いくつかのフレームワークの制約の下で行う必要があります。

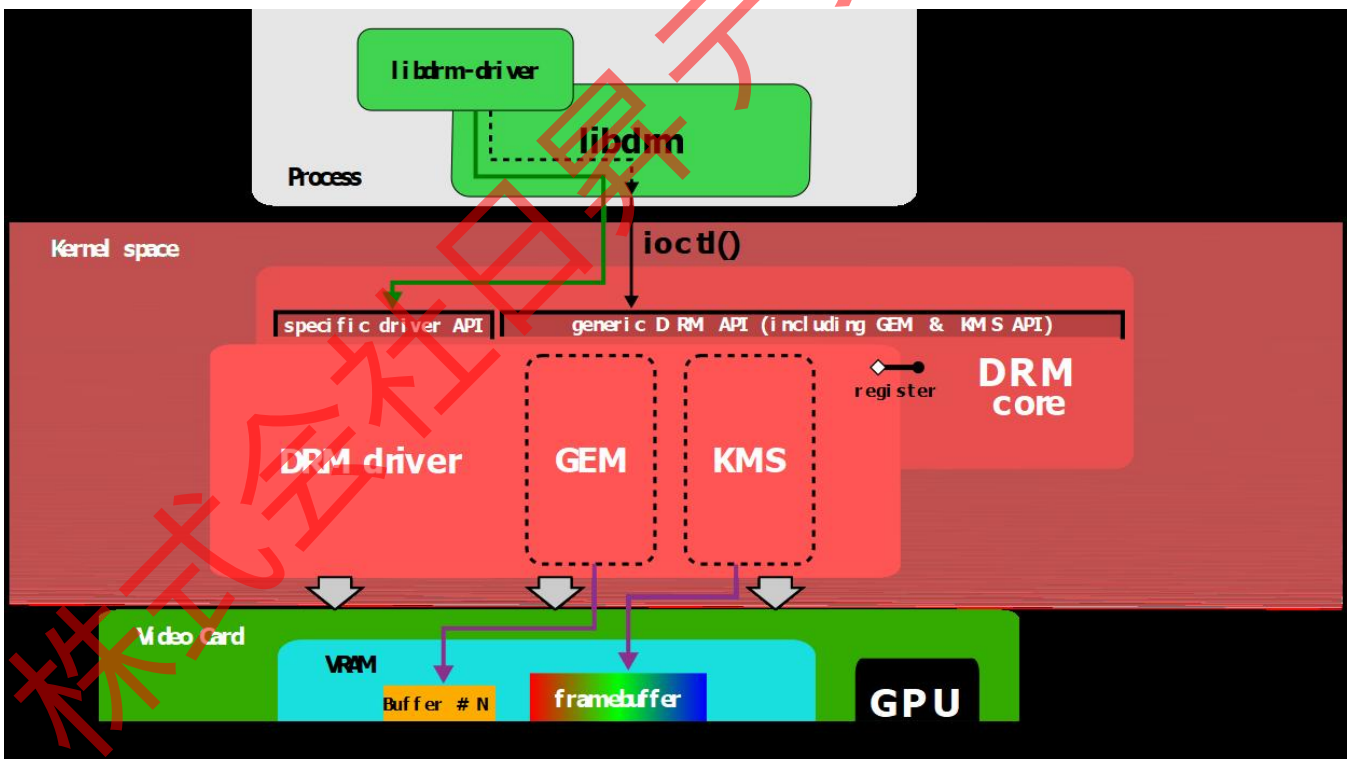
DRM フレームワークを、ユーザースペースとカーネルスペースの 2 つの観点から理解することができます：

ユーザースペース（libdrm ドライバ）：

- Libdrm（ユーザースペースの DRM フレームワーク用ライブラリ）

カーネルスペース（DRM ドライバ）：

- KMS（Kernel Mode Setting、カーネルモード設定）
- GEM（Graphic Execution Manager、グラフィック実行マネージャ）



通常、DRM/KMS は DRM サブシステム全体を指すために使用されますが、KMS と DRM ドライバは DRM サブシステムの中の 2 つの部分に過ぎません。

株式会社日昇テクノロジー

20.1.1 Libdrm

ユーザースペースで提供される DRM フレームワークの Libdrm は、下層インターフェースをラップし、主に様々な IOCTL インターフェースをカプセル化して、上層に一般的な API インターフェースを提供します。ユーザーまたはアプリケーションがユーザースペースで libdrm が提供するライブラリ関数を呼び出すことで、表示リソースにアクセスし、表示リソースを管理および使用することができます。

20.1.2 KMS（Kernel Mode Setting）

KMS は DRM フレームワークの大きなモジュールの一つで、主に表示パラメータの設定と表示画面の制御の 2 つの機能を担当します。これら 2 つの基本機能は、表示ドライバが持つべき基本的な能力です。DRM フレームワークの下では、これら 2 部分を現代の表示デバイスのロジックに合わせて適応させるために、さらにいくつかのサブモジュールに分けられています。

20.1.2.1 DRM FrameBuffer

DRM FrameBuffer は、ハードウェアに依存しないソフトウェアの抽象であり、レイヤー表示内容の情報（幅、高さ、ピクセルフォーマット、ピッチなど）を記述します。

20.1.2.2 Planes

基本的な表示制御ユニットで、各画像には Planes があります。Planes の属性によって、画像の表示領域、画像の反転、色の混合方法などが制御されます。最終的に、複数の画像の混合表示や個別表示などの機能が CRTC コンポーネントを通じて実現されます。

20.1.2.3 CRTC

CRTC の役割は、表示する画像を底層ハードウェアの具体的なタイミング要求に変換することであり、フレームの切り替え、電源制御、色の調整なども担当します。複数の Encoder に接続でき、スクリーンの複製機能を実現します。

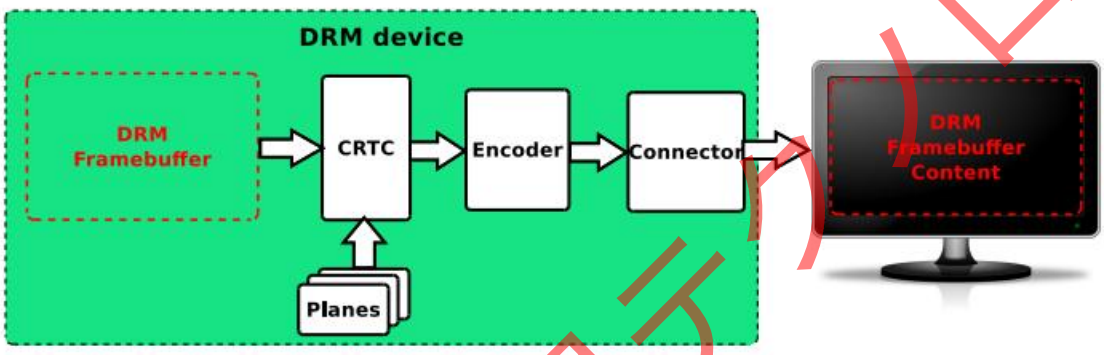
株式会社日昇テクノロジー

20.1.2.4 Encoder

変換器であり、電源管理や、メモリ内のピクセルをディスプレイが必要とする信号に変換する異なる信号変換器の管理を担当します。

20.1.2.5 Connector

コネクタは、ハードウェアデバイスの接続を管理し、HDMI、VGA などを扱います。デバイスの EDID や DPMS 接続状態などを取得できます。



上述のこれらのコンポーネントは、DRM 表示制御プロセスを最終的に完成させます。これらの CRTC、Planes、Encoder、Connector はハードウェアの抽象化であり、実際のハードウェアが存在しなくても、ソフトウェアドライバ内でこれらを実装する必要があります。さもなければ、DRM サブシステムは正常に動作しません。

20.1.3 GEM（Generic DRM Memory-Management）

言わば、GEM は DRM が使用するメモリ（例えばビデオメモリ）の管理を担当するソフトウェアの抽象です。

GEM フレームワークが提供する機能には、以下が含まれます：

- メモリの割り当てと解放
- コマンドの実行
- コマンド実行時の管理

株式会社日昇テクノロジー

20.2 ドライバ概要

DRM ドライバのフレームワークを簡単に説明し、DRM フレームワーク下での表示機能の実装方法について簡単に紹介しました。実際のコードの詳細は、ここで紹介した内容よりもはるかに複雑です。フレームワークコンポーネントの機能を紹介することは、入門としての役割を果たします。

表示機能を担当するチップ上のドライバは、通常、チップメーカー（例えば Rockchip）によって実装され、DRM ホスト、主に drivers/gpu/drm/xxx/ディレクトリ下に配置されます。ここで xxx はチップメーカー（例えば ST、NXP）を指します。詳細なソースコードに興味がある場合は、カーネルソースコードの drivers/gpu/drm ディレクトリで具体的なドライバ実装を確認できます。

Rockchip の表示ドライバはコンポーネントフレームワークを使用し、表示ドライバをマスターとして、その下のデバイスをコンポーネントとしています。display_subsystem デバイスノードの ports ノードは関連するコンポーネントを指し、デバイスツリーのソースコードでは vop_out ノードを指します（RK3588 の VOP はさまざまな出力画像のインターフェースです）。このノードには 3 つのポートがあり、3 つのビデオ信号を同時に出力できるはずです。デバイスツリーは以下の通りです（具体的には arch/arm64/boot/dts/rockchip/RK3588.dtsi を参照してください）：

リスト 1: arch/arm64/boot/dts/rockchip/RK3588.dtsi

```
1 display_subsystem: display-subsystem {
2 compatible = "rockchip,display-subsystem";
3 memory-region = <&drm_logo>, <&drm_cubic_lut>;
4 memory-region-names = "drm-logo", "drm-cubic-lut";
5 ports = <&vop_out>;
6 devfreq = <&dmc>;
7
```

株式会社日昇テクノロジー

```
8 route {

9 route_dsi0: route-dsi0 {

10 status = "disabled";

11 logo,uboot = "logo.bmp";

12 logo,kernel = "logo_kernel.bmp";

13 logo,mode = "center";

14 charge_logo,mode = "center";

15 connect = <&vp0_out_dsi0>;

16 /*.................*/

17 };

18 };

19 /*.................*/

20 vop: vop@fe040000 {

21 compatible = "rockchip,rk3568-vop";

22 reg = <0x0 0xfe040000 0x0 0x3000>, <0x0 0xfe044000 0x0 0x1000>;

23 reg-names = "regs", "gamma_lut";

24 rockchip,grf = <&grf>;

25 interrupts = <GIC_SPI 148 IRQ_TYPE_LEVEL_HIGH>;

26 clocks = <&cru ACLK_VOP>, <&cru HCLK_VOP>, <&cru DCLK_VOP0>, <&cru DCLK_VOP1>,

  <&cru DCLK_VOP2>;

27 clock-names = "aclk_vop", "hclk_vop", "dclk_vp0", "dclk_vp1", "dclk_vp2";

28 iommus = <&vop_mmu>;
```

株式会社日昇テクノロジー

```
29 power-domains = <&power RK3568_PD_VO>;
30 status = "disabled";
31
32 vop_out: ports {
33 #address-cells = <1>;
34 #size-cells = <0>;
35 vp0: port@0 {
36 /*.....*/
37 vp1: port@1 {
38 /*.....*/
39 vp2: port@2 {
40 /*.....*/
41 };
42 };
```

プラットフォームドライバ rockchip-drm はデバイスツリーにマッチし、ports ノードと iommus ノードを検索し、component_master_add_with_match 関数を使用してコンポーネントフレームワークに自身を登録します。それは rockchip_drm_ops を設定し、コンポーネントは component_add 関数によって追加されます。マスターがすべてのコンポーネントにマッチした後、マスターの bind コールバック関数が呼び出され、最終的に drm_dev_register()関数によって DRM コアに登録されます。一部のドライバソースコードは以下の通りです（詳細は drivers/gpu/drm/rockchip/rockchip_drm_drv.c を参照）：

株式会社日昇テクノロジー

リスト 2: drivers/gpu/drm/rockchip/rockchip_drm_drv.c

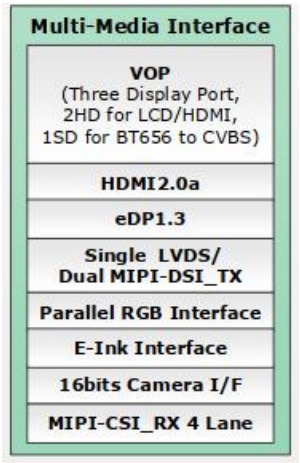
```
1 static const struct component_master_ops rockchip_drm_ops = {

2 .bind = rockchip_drm_bind,

3 .unbind = rockchip_drm_unbind,

4 };

5

6 static int rockchip_drm_platform_probe(struct platform_device *pdev)

7 {

8 struct device *dev = &pdev->dev;

9 struct component_match *match = NULL;

10 int ret;

11

12 ret = rockchip_drm_platform_of_probe(dev);

13 #if !IS_ENABLED(CONFIG_DRM_ROCKCHIP_VVOP)

14 if (ret)

15 return ret;

16 #endif

17

18 match = rockchip_drm_match_add(dev);

19 if (IS_ERR(match))

20 return PTR_ERR(match);

21
```

株式会社日昇テクノロジー

```
22 ret = component_master_add_with_match(dev, &rockchip_drm_ops, match);
23 if (ret < 0) {
24 rockchip_drm_match_remove(dev);
25 return ret;
26 }
27 dev->coherent_dma_mask = DMA_BIT_MASK(64);
28
29 return 0;
30 }
31
32 static struct platform_driver rockchip_drm_platform_driver = {
33 .probe = rockchip_drm_platform_probe,
34 .remove = rockchip_drm_platform_remove,
35 .shutdown = rockchip_drm_platform_shutdown,
36 .driver = {
37 .name = "rockchip-drm",
38 .of_match_table = rockchip_drm_dt_ids,
39 .pm = &rockchip_drm_pm_ops,
40 },
41 };
```

次に、lubancat4 を例にして、RK3588 がサポートする様々な表示インターフェイスについて見てみましょう：

株式会社日昇テクノロジー



Lubancat4 は MIPI DSI と HDMI インターフェイスを提供しています。HDMI 表示インターフェイスを例にとると、デフォルトのデバイスツリー（RK3588-lubancat4.dts）では HDMI が有効になっており、HDMI に関するデバイスツリーの記述は以下の通りです：

リスト 3: arch/arm64/boot/dts/rockchip/RK3588.dtsi

```
1 hdmi: hdmi@fe0a0000 {
2 compatible = "rockchip,rk3568-dw-hdmi";
3 reg = <0x0 0xfe0a0000 0x0 0x20000>; /* レジスタの物理ベースアドレスは 0xfe0a0000、メモリマッピングの長さは 0x20000 */
4 interrupts = <GIC_SPI 45 IRQ_TYPE_LEVEL_HIGH>;
5 clocks = <&cru PCLK_HDMI_HOST>,
6 <&cru CLK_HDMI_SFR>,
7 <&cru CLK_HDMI_CEC>,
8 <&pmucru PLL_HPLL>,
9 <&cru HCLK_VOP>;
10 clock-names = "iahb", "isfr", "cec", "ref", "hclk";
11 power-domains = <&power RK3568_PD_VO>;
```

株式会社日昇テクノロジー

```
12 reg-io-width = <4>; /* レジスタの読み書きアクセス幅は 4 バイト */

13 rockchip,grf = <&grf>;

14 #sound-dai-cells = <0>;

15 pinctrl-names = "default";

16 pinctrl-0 = <&hdmitx_scl &hdmitx_sda &hdmitxm0_cec>;

17 status = "disabled";

18

19 ports {

20 #address-cells = <1>;

21 #size-cells = <0>;

22

23 hdmi_in: port { /* vop2 の hdmi インターフェースのエンドポイントにバインド */

24 reg = <0>;

25 #address-cells = <1>;

26 #size-cells = <0>;

27

28 hdmi_in_vp0: endpoint@0 {

29 reg = <0>;

30 remote-endpoint = <&vp0_out_hdmi>;

31 status = "disabled";

32 };

33 hdmi_in_vp1: endpoint@1 {
```

株式会社日昇テクノロジー

```
34 reg = <1>;
35 remote-endpoint = <&vp1_out_hdmi>;
36 status = "disabled";
37 };
38 };
39 };
40 };
```

HDMI のプラットフォームドライバは rockchip_drm_init 中で登録され、デバイスツリーにマッチする際に該当するドライバが呼び出され、dw_hdmi_rockchip_probe 関数を実行し、最終的に component_add 関数によってコンポーネントを登録し、component_ops の操作関数を設定します。マスターの bind コールバックの後、各コンポーネントの bind コールバックが呼び出されます。これは、ここでの dw_hdmi_rockchip_bind 関数にあたります。一部のコードは以下の通りです：

リスト 4: drivers/gpu/drm/rockchip/dw_hdmi-rockchip.c

```
1 static const struct component_ops dw_hdmi_rockchip_ops = {
2 .bind = dw_hdmi_rockchip_bind,
3 .unbind = dw_hdmi_rockchip_unbind,
4 };
5
6 static int dw_hdmi_rockchip_probe(struct platform_device *pdev)
7 {
8 pm_runtime_enable(&pdev->dev);
9 pm_runtime_get_sync(&pdev->dev);
```

株式会社日昇テクノロジー

```
10
11 return component_add(&pdev->dev, &dw_hdmi_rockchip_ops);
12 }
13 /*.....*/
14 struct platform_driver dw_hdmi_rockchip_pltfm_driver = {
15 .probe = dw_hdmi_rockchip_probe,
16 .remove = dw_hdmi_rockchip_remove,
17 .shutdown = dw_hdmi_rockchip_shutdown,
18 .driver = {
19 .name = "dwhdmi-rockchip",
20 .of_match_table = dw_hdmi_rockchip_dt_ids,
21 .pm = &dw_hdmi_pm_ops,
22 },
23 };
```

dw_hdmi_rockchip_bind 関数内で、HDMI を初期化し、CRTC を検索し、Encoder などを初期化します。

20.3 画面表示テスト

DRM ドライバフレームワーク下でのドライバ効果をテストするために、HDMI 画面のテストを行います。RK3588-lubancat4.dts デバイスツリーを使用して、デフォルトで HDMI が有効になっています。

20.3.1 Libdrm

libdrm のテストプログラムを作成するのは比較的複雑ですので、ここでは libdrm の公式テストツールを使用してテストを行います。こちらからソースコードをダウンロードし、交差コンパイルして開発ボード上で使用するテストツールを作成できます：libdrm。

株式会社日昇テクノロジー

新しいバージョンの libdrm は meson+ninja のビルド方式を採用しており、旧バージョンの autotools とは異なります。基礎がない方が新しいバージョンの libdrm を構築するのは少し苦労するかもしれません。我々がコンパイルしたテストプログラムを直接使用することをお勧めします。テストプログラムは linux_driver/framework_drm/modetest にあります。

自分で libdrm をコンパイルする場合は、以下のコマンドを参考にしてください：

```
git clone https://gitlab.freedesktop.org/mesa/drm

sudo apt -y install python3-pip cmake git ninja-build

python3 -m pip install meson /* インストール後、ボードを再起動*/

meson . build && ninja -C build
```

コンパイル後、build/tests/modetest/に modetest プログラムがあります。libdrm テストプログラムに興味がある場合は、libdrm ソースコードをダウンロードして解凍し、/drm/tests/modetest/ディレクトリで modetest.c ファイルを見ることができます。これはテストプログラムのソースコードです。

20.3.2 実験操作

上記の modetest テストプログラムを開発ボードにアップロードし、chmod で実行可能な権限を追加するか、直接使用してください。グラフィカルインターフェイスを停止するには以下のコマンドを使用します：

```
# グラフィカルインターフェイスを停止するコマンド

sudo systemctl set-default multi-user.target

sudo reboot

# テスト終了後、グラフィカルインターフェイスを有効にするには以下のコマンドを使用します：

sudo systemctl set-default graphical.target
```

./modetest コマンドを使用して modetest プログラムを実行します：

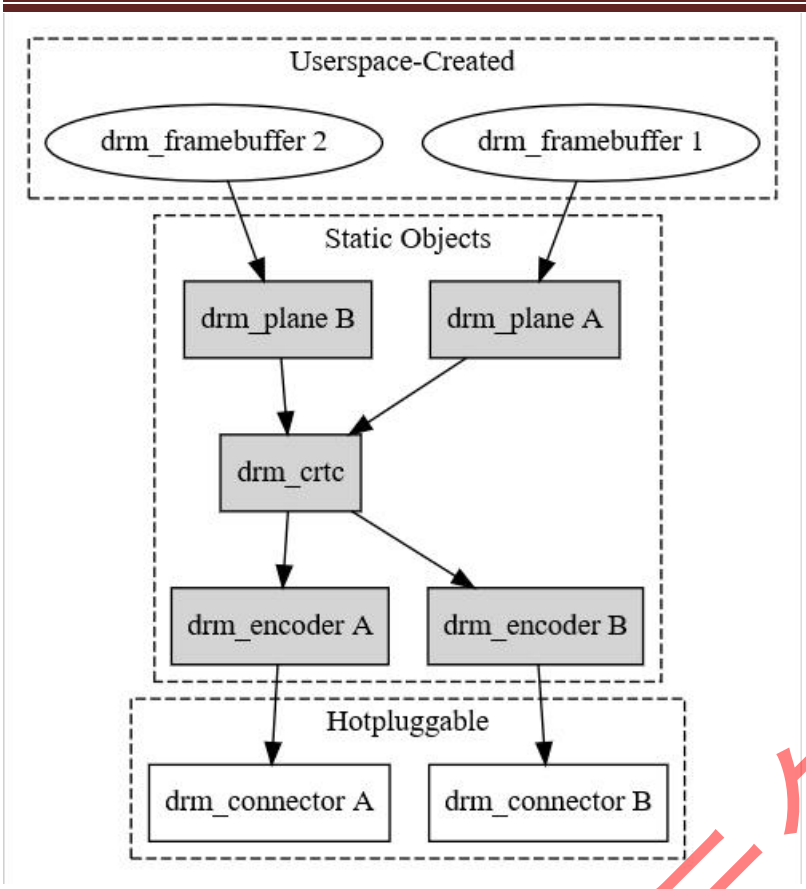
株式会社日昇テクノロジー

```
trying to open device 'i915'...failed
trying to open device 'amdgpu'...failed
trying to open device 'radeon'...failed
trying to open device 'nouveau'...failed
trying to open device 'vmwgfx'...failed
trying to open device 'omapdrm'...failed
trying to open device 'exynos'...failed
trying to open device 'tilcdc'...failed
trying to open device 'msm'...failed
trying to open device 'sti'...failed
trying to open device 'tegra'...failed
trying to open device 'imx-drm'...failed
trying to open device 'rockchip'...done
Encoders:
id      crtc    type    possible crtcs  possible clones
147     0       Virtual 0x00000001      0x00000000
149     0       TMDS    0x00000001      0x00000000

Connectors:
id      encoder status          name            size (mm)       modes   encoders
150     0       connected       HDMI-A-1        600x340         31      149
  modes:
        index name refresh (Hz) hdisp hss hse htot vdisp vss vse vtot
  #0 1920x1080 60.00 1920 2008 2052 2200 1080 1084 1089 1125 148500 flags: phsync, pvsync; type: p
  #1 1920x1080 59.94 1920 2008 2052 2200 1080 1084 1089 1125 148352 flags: phsync, pvsync; type: d
  #2 1920x1080i 30.00 1920 2008 2052 2200 1080 1084 1094 1125 74250 flags: phsync, pvsync, interla
  #3 1920x1080i 29.97 1920 2008 2052 2200 1080 1084 1094 1125 74176 flags: phsync, pvsync, interla
  #4 1920x1080 50.00 1920 2448 2492 2640 1080 1084 1089 1125 148500 flags: phsync, pvsync; type: d
  #5 1920x1080i 25.00 1920 2448 2492 2640 1080 1084 1094 1125 74250 flags: phsync, pvsync, interla
  #6 1680x1050 59.88 1680 1728 1760 1840 1050 1053 1059 1080 119000 flags: phsync, nvsync; type: d
  #7 1280x1024 75.02 1280 1296 1440 1688 1024 1025 1028 1066 135000 flags: phsync, pvsync; type: d
  #8 1280x1024 60.02 1280 1328 1440 1688 1024 1025 1028 1066 108000 flags: phsync, pvsync; type: d
  #9 1440x900 59.90 1440 1488 1520 1600 900 903 909 926 88750 flags: phsync, nvsync; type: driver
  #10 1280x960 60.00 1280 1376 1488 1800 960 961 964 1000 108000 flags: phsync, pvsync; type: dri
  #11 1280x720 60.00 1280 1390 1430 1650 720 725 730 750 74250 flags: phsync, pvsync; type: driver
  #12 1280x720 59.94 1280 1390 1430 1650 720 725 730 750 74176 flags: phsync, pvsync; type: driver
  #13 1280x720 50.00 1280 1720 1760 1980 720 725 730 750 74250 flags: phsync, pvsync; type: driver
  #14 1024x768 75.03 1024 1040 1136 1312 768 769 772 800 78750 flags: phsync, pvsync; type: driver
  #15 1024x768 70.07 1024 1048 1184 1328 768 771 777 806 75000 flags: nhsync, nvsync; type: driver
  #16 1024x768 60.00 1024 1048 1184 1344 768 771 777 806 65000 flags: nhsync, nvsync; type: driver
  #17 832x624 74.55 832 864 928 1152 624 625 628 667 57284 flags: nhsync, nvsync; type: driver
  #18 800x600 75.00 800 816 896 1056 600 601 604 625 49500 flags: phsync, pvsync; type: driver
  #19 800x600 72.19 800 856 976 1040 600 637 643 666 50000 flags: phsync, pvsync; type: driver
  #20 800x600 60.32 800 840 968 1056 600 601 605 628 40000 flags: phsync, pvsync; type: driver
  #21 800x600 56.25 800 824 896 1024 600 601 603 625 36000 flags: phsync, pvsync; type: driver
  #22 720x576 50.00 720 732 796 864 576 581 586 625 27000 flags: nhsync, nvsync; type: driver
  #23 720x480 60.00 720 736 798 858 480 489 495 525 27027 flags: nhsync, nvsync; type: driver
  #24 720x480 59.94 720 736 798 858 480 489 495 525 27000 flags: nhsync, nvsync; type: driver
  #25 640x480 75.00 640 656 720 840 480 481 484 500 31500 flags: nhsync, nvsync; type: driver
  #26 640x480 72.81 640 664 704 832 480 489 492 520 31500 flags: nhsync, nvsync; type: driver
  #27 640x480 66.67 640 704 768 864 480 483 486 525 30240 flags: nhsync, nvsync; type: driver
  #28 640x480 60.00 640 656 752 800 480 490 492 525 25200 flags: nhsync, nvsync; type: driver
  #29 640x480 59.94 640 656 752 800 480 490 492 525 25175 flags: nhsync, nvsync; type: driver
  #30 720x400 70.08 720 738 846 900 400 412 414 449 28320 flags: nhsync, pvsync; type: driver
  props:
        1 EDID:
                flags: immutable blob
```

プログラムが実行された後、開発ボード上の DRM フレームワーク下の表示デバイスがリストアップされます。Encoders、Connectors、CRTCs などの用語は、前述の説明から少し印象に残っているはずです。テストコマンドを使用してテストを行います：

DRM フレームワーク下の表示プロセスは以下の図のようになります：

株式会社日昇テクノロジー



以前の端末で出力された HDMI スクリーンに対応する connectors、CRTCs の ID を見つけるためにコマンドを使用します：

```
./modetest -M rockchip -e

./modetest -M rockchip -c
```

```
Encoders:
id      crtc    type    possible crtcs  possible clones
147     0       Virtual 0x00000001      0x00000000
149     71      TMDS    0x00000001      0x00000000

Connectors:
id      encoder status          name            size (mm)       modes   encoders
150     149     connected       HDMI-A-1        600x340         31      149
  modes:
        index name refresh (Hz) hdisp hss hse htot vdisp vss vse vtot
```

```
CRTCs:
id      fb      pos     size
71      160     (0,0)   (1920x1080)
  #0 1920x1080 60.00 1920 2008 2052 2200 1080 1084 1089 1125 148500 flags: phsync,
  props:
        54 ACLK:
                flags: range
                values: 0 4294967295
                value: 500000
```

図にある状態が connected の connectorsID は 150 で、名前は HDMI-A-1 です。CRTCs オブジェクトの中で、唯一 id が 71 の CRTC があり、そのパラメータも私たちの HDMI スクリーンのパラメータで

株式会社日昇テクノロジー

す。

したがって、以下のコマンドを実行してテストすることができます：

```
./modetest -M rockchip -s 150@71:1920x1080
```

ここで 150、71 はそれぞれ画面の Connectors ID、CRTCs ID です。テストの現象は画像に示されています。

端末でエンターキーを押してテストを終了します。テストに関するさらに多くの情報については、Linux DRM/KMS テストツール modetest を参照してください。

記事の最後にちょっとした知識を追加しますが、DRM 機能は現代の表示装置のニーズに合致していますが、多くの古い装置やソフトウェアは Framebuffer のサポートが必要です。そのため、DRM フレームワークの下で、Framebuffer デバイスを模倣するためのコードが一部あります。

rockchip が提供する表示ドライバーコードの中にも、Framebuffer デバイスを模倣する関連コードがあります。drivers/gpu/drm/rockchip/rockchip_drm_fb.c ファイルを参照してください。最終的な効果は、デバイスディレクトリ下に、おなじみの/dev/fb0 が現れることです。

株式会社日昇テクノロジー

伝統的な FrameBuffer デバイスのテスト方法を使って、以下のコマンドでテストすることができます：

```
# 画面乱れのような効果が得られます

cat /dev/random > /dev/fb0
```

第 21 章 SMP（Symmetrical Multi-Processing）

21.1 プロセッサの発展過程

コアコンポーネントであるプロセッサは、入力データの分析と処理を担当し、出力を行います。プロセッサの性能を測る主な 2 つの側面は、各クロックサイクルで実行できる命令数（IPC: Instruction Per Clock）とプロセッサのクロック速度です。シングルコアの時代には、同一アーキテクチャの改良による IPC の向上幅は限られており、性能を向上させるには CPU のクロック速度を上げるしかありませんでした。しかし、CPU のクロック速度が上がるにつれて、プロセッサの性能が顕著に向上しないばかりか、CPU の消費電力が大幅に増加しました。主周波数が 1GHz 上昇するごとに消費電力が約 25 ワット上昇し、チップの消費電力が 150 ワットを超えると、既存の空冷冷却システムでは冷却ニーズを満たせなくなりました。この時点で、「クロック速度至上主義」の時代は終わりを告げ、人々は別の方法を探さざるを得ませんでした。そこで、マルチコアプロセッサが登場しました。

マルチコアプロセッサは、2 つ以上の独立したプロセッサを単一の集積回路（IC）に封入したものです。マルチコアのハードウェア実装方法には 2 つの種類があります。一つは、すべて同じアーキテクチャの CPU を統合する同型マルチコアで、システムリソースを共有し、ほとんどのデスクトップやノート PC の CPU プロセッサはこのアーキテクチャを採用しています。もう一つは異型マルチコアで、一般的には汎用プロセッサ、DSP、FPGA などを同時に統合し、ロボットのピックアンドプレイスアセンブリラインなど複雑でリアルタイム性が高いアプリケーションシナリオに主に使用されます。汎用プロセッサだけではすぐに過負荷になり、指定された時間内に処理を完了できなくなる可能性があります。

初期のスマートフォン SoC は、ARM 社によって提案された「Big.Little」アーキテクチャを採用してい

株式会社日昇テクノロジー

ました。これは、2 種類の異なるタイプの ARM コアを単一の集積回路に統合したもので、高性能コア（big core）と低性能コア（little core）から成り、高負荷タスクを担当する big core と、スマートフォンの大部分の作業負荷を処理する little core があります。モバイルデバイスの普及に伴い、モバイルデバイスの性能に対する要求が高まり、それに伴い消費電力も増加しました。「Big.Little」アーキテクチャは CPU の大小コアリソースの合理的な呼び出しにより、高性能と低消費電力を両立させ、スマートフォンバッテリーの持続時間を大幅に向上させました。

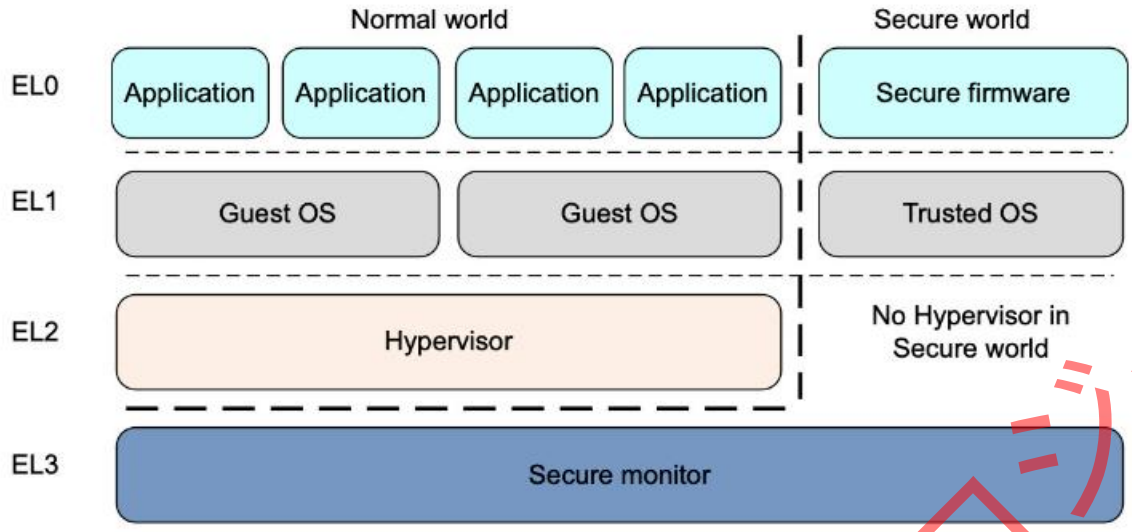
DynamIQ は、big.LITTLE を基に拡張および設計されたもので、big.LITTLE 技術の進化版と見なされますが、big.LITTLE は DynamIQ 技術がサポートする多くの機能の一つに過ぎません。大きな CPU クラスタと小さな CPU クラスタを統合し、大小の CPU を完全に統合した CPU クラスタを形成するアプローチを取ります。

21.2 関連知識紹介

以下の小節での起動プロセス分析を始める前に、いくつかの関連知識を理解しておきましょう：

- ARMv8 の例外レベル、ARMv8 では例外レベルが権限レベルを決定し、EL0~3 に分けられます。数字が小さいほど、例外レベルが低く、権限が低くなります。

株式会社日昇テクノロジー



- TF-A（Trusted Firmware-A）は、ARM Profile A（ARMv8-a、ARMv7-a）の信頼できるファームウェアの参照実装です。簡単に言うと、セキュリティ問題を処理し、ファームウェアの起動には署名認証があり、起動トラストチェーン（Trust Chain）が確立されています。

- PSCI（Power State Coordination Interface）は、ARM によって定義された電源管理インターフェースの仕様で、通常はファームウェアによって実装されます。Linux システムは smc/hvc 命令を通じて異なる Exception Level に入り、対応する実装を呼び出します。ARMv8 アーキテクチャは Virtualization、Security などの概念を導入しました。CPU の起動、サスペンド/レジュームなどの操作は従来のように単純ではなく、底層のファームウェアインターフェースを呼び出す命令が必要です。

21.3 RK3588 プロセッサの基本紹介

Lubancat4 を例にとると、このボードはマルチコアプロセッサで、内部には 4 つの Cortex-A55 CPU が統合されており、最高クロック速度は 2GHz に達します。以下のコマンドを使用して CPU の関連情報を確認できます：

```
1 lscpu
```

CPU の情報を表示

株式会社日昇テクノロジー

```
root@lubancat:/home/cat# lscpu
Architecture:                    aarch64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              1
Core(s) per socket:              2
Socket(s):                       3
Vendor ID:                       ARM
Model:                           0
Model name:                      Cortex-A55
Stepping:                        r2p0
CPU max MHz:                     2352.0000
CPU min MHz:                     408.0000
BogoMIPS:                        48.00
L1d cache:                       256 KiB
L1i cache:                       256 KiB
L2 cache:                        1 MiB
L3 cache:                        3 MiB
Vulnerability Itlb multihit:     Not affected
Vulnerability L1tf:              Not affected
Vulnerability Mds:               Not affected
Vulnerability Meltdown:          Not affected
Vulnerability Mmio stale data:   Not affected
Vulnerability Retbleed:          Not affected
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl
Vulnerability Spectre v1:        Mitigation; __user pointer sanitization
Vulnerability Spectre v2:        Vulnerable: Unprivileged eBPF enabled
Vulnerability Srbds:             Not affected
Vulnerability Tsx async abort:   Not affected
Flags:                           fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrdm lrcpc dcp
```

• Architecture：プロセッサが使用するアーキテクチャを表します。一般的なものには x86、MIPS、PowerPC、ARM などがあります。

• Byte Order：プロセッサがビッグエンディアンモードかリトルエンディアンモードかを表します；

• CPU(s)：プロセッサが持つコアの数を表します。ここでは 4 つあり、それぞれ 0~3 に対応しています；

• On-line CPU(s) list：現在正常に稼働している CPU の番号で、現在システム内の 4 つのコアがすべて正常に稼働している状態であることがわかります；

• Socket(s)：ソケットで、現在のボード上にある rk3588 チップの数と理解できます；

• Core(s) per socket：チップメーカーは複数のコアを 1 つのソケットに集成します。ここではチップ上に 4 つのコアがあることを示します；

• Thread(s) per core：プロセスはプログラムの実行インスタンスであり、それは各スレッドの分業協力に依存しています。そのため、Intel はハイパースレッディング技術を開発しました。この技術により、Intel は物理的な CPU 内に 2 つの論理スレッドを提供し、シングルプロセッサでもスレッドレベルの並列計算を使用できるようにしました。

株式会社日昇テクノロジー

• Vendor ID：チップメーカー ID で、例えば GenuineIntel、ARM、AMD などがあります；

• Model name：CPU のモデル名で、ここでは Cortex-A55 に対応しています；

• Stepping：一連の CPU の設計または製造バージョンを識別するために使用されます。ステッピングバージョンは、このシリーズの CPU の製造プロセスの改善、バグの解決、または機能の追加に伴い変更されます；

• CPU min MHz、CPU max MHz：CPU がサポートする最小および最大の周波数で、システムは実際の状況に応じて CPU の動作周波数を調整しますが、最大サポート周波数を超えることはありません；

• BogoMIPS：MIPS は millions of instructions per second（百万命令/秒）の略で、この値は関数によって計算され、プロセッサの性能を大まかに算出するためにのみ使用できます。非常に正確とは言えません。

• Flags：CPU の特徴を表すパラメータです。

21.4 Linux SMP 起動プロセス

現在、マルチコアプロセッサをサポートするリアルタイムオペレーティングシステムのアーキテクチャには、SMP（Symmetric Multi-Processing）アーキテクチャと AMP（Asymmetric Multi-Processing）アーキテクチャの 2 種類があります。AMP モードでは、各 CPU コア上で 1 つのオペレーティングシステム（必ずしも同じである必要はありません）が実行され、各オペレーティングシス

テムは自分専用のメモリを持ち、限定された共有メモリを介して相互に通信します。SMP モードでは、1 つのオペレーティングシステムインスタンスがすべての CPU コアを制御し、すべての CPU コアがメモリと周辺機器リソースを共有します。AMP モードに比べ、SMP モードのオペレーティングシステムは共有メモリ、高いパフォーマンスと消費電力比、および負荷分散の容易さなどの利点を持ち、マルチコアプロセッサのハードウェアの利点をよりよく活かすことができます。

SMP モードではすべての CPU コアがメモリと周辺機器リソースを共有しているものの、起動段階では

株式会社日昇テクノロジー

彼らの地位が異なります。core0 は主 CPU（ブートプロセッサ）であり、他は従 CPU（または補助プロセッサ）です。ブート CPU はブートローダー（uboot など）の実行およびカーネルの初期化を担当し、システムの初期化が完了した後に他のプロセッサを起動します。

メインプロセッサが他のプロセッサを起動する方法にはいくつかあり、ARM プロセッサでは一般的に spin-table（スピンテーブル）と psci（Power State Coordination Interface）の 2 つの方法が使用されます。ARM64 では psci がより多く使用されます。

Linux カーネルのコンパイル時には、CONFIG_SMP 設定項目がカーネルが SMP をサポートするかどうかを制御します。デフォルトは`#define CONFIG_SMP 1`です。Linux システムでの SMP モードの起動プロセスは、リセット後に CPU0 が ROM コードを実行し、その後 ROM コードが CPU0 に Bootloader（TF-A および uboot を含む）のコードおよびカーネルコードの実行を指示します。

もし以前にカーネルの起動出力に注意を払っていたら、以下のようなプリント情報が表示されているのを見つけるかもしれません。これは現在のカーネルが CPU0 上で実行されていることを示唆しています。

```
Starting kernel ...
[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x412fd050]
```

上記は起動プロセスを簡単に紹介しただけですが、実際には、カーネルはチップ内にいくつの CPU コアがあるかをどのように識別するのでしょうか？CPU0 は他の CPU をどのように起動するのでしょうか？まず、現在のシステム内の各 CPU コアの動作状態を記述するために、カーネルは kernel/cpu.c 内にいくつかの cpumask 型の構造体変数を定義しています。

株式会社日昇テクノロジー

リスト 1: cpumask 型の構造体変数（kernel/cpu.c に位置）

```
1 struct cpumask __cpu_possible_mask __read_mostly;
2 EXPORT_SYMBOL(__cpu_possible_mask);
3
4 struct cpumask __cpu_online_mask __read_mostly;
5 EXPORT_SYMBOL(__cpu_online_mask);
6
7 struct cpumask __cpu_present_mask __read_mostly;
8 EXPORT_SYMBOL(__cpu_present_mask);
9
10 struct cpumask __cpu_active_mask __read_mostly;
11 EXPORT_SYMBOL(__cpu_active_mask);
12
13 struct cpumask __cpu_isolated_mask __read_mostly;
14 EXPORT_SYMBOL(__cpu_isolated_mask);
```

cpumask 型の構造体にはメンバ変数が 1 つだけあり、データ型は unsigned long の 1 次元配列です。1 つの CPU コアは配列要素の 1 ビットで表され、マクロ定義 BITS_TO_LONGS(bits)が配列の長さを計算する責任を持っています。現在 33 個の CPU(NR_CPUS=33)があると仮定すると、BITS_TO_LONGS の変換後、必要な配列の長さが 2 であることがわかります。

株式会社日昇テクノロジー

リスト 2: struct cpumask 構造体（include/linux/cpumask.h に位置）

```
1 /* Don't assign or return these: may not be this big! */
2 typedef struct cpumask { DECLARE_BITMAP(bits, NR_CPUS); } cpumask_t;
3
4 #define DECLARE_BITMAP(name,bits) ¥
5 unsigned long name[BITS_TO_LONGS(bits)]
```

これらの cpumask 型の変数は以下のような役割を果たします：

- __cpu_possible_mask：物理的に存在し、アクティブ化される可能性のある CPU コアの番号を記録します。デバイスツリーから CPU ノードを解析して取得します。
- __cpu_online_mask：現在のシステムで実行中の CPU コアの番号を記録します。ハードウェア上で接続されている CPU です。
- __cpu_present_mask：現在のシステム内のすべての CPU コアの番号を動的に記録します。CONFIG_HOTPLUG_CPU がカーネルに設定されている場合、これらの CPU コアがすべて実行中であるとは限りません。なぜなら、一部の CPU コアはホットプラグされている可能性があるからです。
- __cpu_active_mask：現在のシステムでタスクスケジューリングに使用できる CPU コアを記録します。

/sys/devices/system/cpu ディレクトリ内には、システム内のすべての CPU コアと上記の各変数の内容が記録されています。例えば、ファイル present は__cpu_present_mask 変数に対応しています。以下のコマンドを実行すると、現在のシステム内のすべての CPU コアの番号を確認できます。

```
1 cat /sys/devices/system/cpu/present
```

さらに、/sys/devices/system/cpu/cpu1/online ファイルを通じて、ユーザースペースから特定の CPU

株式会社日昇テクノロジー

コアをオンまたはオフにすることができます。

```
1 # CPU1 をオフにする

2 echo 0 > /sys/devices/system/cpu/cpu1/online

3 # CPU1 をオンにする

4 echo 1 > /sys/devices/system/cpu/cpu1/online
```

次に、カーネルが CPU 間の関係をどのように確立するかを見てみましょう。デバイスツリーのルートノードには/cpus という子ノードがあり、その内容は以下の通りです：

リスト 3: /cpus ノードと/psci ノード（arch/arm64/boot/dts/rockchip/RK3588.dtsi に位置）

```
1 cpus {

2 #address-cells = <2>;

3 #size-cells = <0>;

4

5 cpu0: cpu@0 {

6 device_type = "cpu";

7 compatible = "arm,cortex-a55";

8 reg = <0x0 0x0>;

9 enable-method = "psci";

10 clocks = <&scmi_clk 0>;

11 operating-points-v2 = <&cpu0_opp_table>;

12 cpu-idle-states = <&CPU_SLEEP>;

13 #cooling-cells = <2>;

14 dynamic-power-coefficient = <187>;

15 };
```

株式会社日昇テクノロジー

```
16
17 cpu1: cpu@100 {
18 device_type = "cpu";
19 compatible = "arm,cortex-a55";
20 reg = <0x0 0x100>;
21 enable-method = "psci";
22 clocks = <&scmi_clk 0>;
23 operating-points-v2 = <&cpu0_opp_table>;
24 cpu-idle-states = <&CPU_SLEEP>;
25 };
26
27 cpu2: cpu@200 {
28 device_type = "cpu";
29 compatible = "arm,cortex-a55";
30 reg = <0x0 0x200>;
31 enable-method = "psci";
32 clocks = <&scmi_clk 0>;
33 operating-points-v2 = <&cpu0_opp_table>;
34 cpu-idle-states = <&CPU_SLEEP>;
35 };
36
37 cpu3: cpu@300 {
```

株式会社日昇テクノロジー

```
38 device_type = "cpu";

39 compatible = "arm,cortex-a55";

40 reg = <0x0 0x300>;

41 enable-method = "psci";

42 clocks = <&scmi_clk 0>;

43 operating-points-v2 = <&cpu0_opp_table>;

44 cpu-idle-states = <&CPU_SLEEP>;

45 };

46

47 /*.....*/

48 };

49

50 psci {

51 compatible = "arm,psci-1.0"; // PSCI に対応し、psci_0_2_init を使用

52 method = "smc"; // smc 命令を使用

53 };
```

この cpus ノードは、現在のハードウェア上に 4 つの CPU が存在することを記述しており、それぞれが CPU0、CPU1、CPU2、CPU3 です。カーネルコードはこのノードを解析することで、現在のシステムの CPU コア数を得ることができます。enable-method 属性が psci であることから、サブプロセッサの起動は psci 方式で行われることがわかります。

psci ノードは使用されるバージョンと EL3 例外レベルへのトラップ方法を示しています。バージョンは PSCI v0.2 を使用しており、これは起動情報からも確認できます。

株式会社日昇テクノロジー

```
[ 0.000000] psci: probing for conduit method from DT.
[ 0.000000] psci: PSCIv1.1 detected in firmware.
[ 0.000000] psci: Using standard PSCI v0.2 function IDs
[ 0.000000] psci: Trusted OS migration not required
[ 0.000000] psci: SMC Calling Convention v1.2
```

さらに、cpus ノードには「operating-points-v2」属性も含まれており、cpu0_opp_table ノードを指しています。このノードは、CPU コアがサポートする周波数などを設定するために使用されます。

リスト 4: /cpu0_opp_table ノード（arch/arm64/boot/dts/rockchip/RK3588.dtsi に位置）

```
1 cpu0_opp_table: cpu0-opp-table {
2 compatible = "operating-points-v2";
3 opp-shared;
4
5 mbist-vmin = <825000 900000 950000>;
6 nvmem-cells = <&cpu_leakage>, <&core_pvtm>, <&mbist_vmin>;
7 nvmem-cell-names = "leakage", "pvtm", "mbist-vmin";
8 rockchip,pvtm-voltage-sel = <
9 0 84000 0
10 84001 91000 1
11 91001 100000 2
12 >;
13 rockchip,pvtm-freq = <408000>;
14 rockchip,pvtm-volt = <900000>;
```

株式会社日昇テクノロジー

```
15 rockchip,pvtm-ch = <0 5>;

16 rockchip,pvtm-sample-time = <1000>;

17 rockchip,pvtm-number = <10>;

18 rockchip,pvtm-error = <1000>;

19 rockchip,pvtm-ref-temp = <40>;

20 rockchip,pvtm-temp-prop = <26 26>;

21 rockchip,thermal-zone = "soc-thermal";

22 rockchip,temp-hysteresis = <5000>;

23 rockchip,low-temp = <0>;

24 rockchip,low-temp-adjust-volt = <

25 /* MHz MHz uV */

26 0 1608 75000

27 >;

28

29 opp-408000000 {

30 opp-hz = /bits/ 64 <408000000>;

31 opp-microvolt = <850000 850000 1150000>;

32 opp-microvolt-L0 = <850000 850000 1150000>;

33 opp-microvolt-L1 = <825000 825000 1150000>;

34 opp-microvolt-L2 = <825000 825000 1150000>;

35 clock-latency-ns = <40000>;

36 };
```

株式会社日昇テクノロジー

```
37 opp-600000000 {
38 opp-hz = /bits/ 64 <600000000>;
39 opp-microvolt = <850000 825000 1150000>;
40 opp-microvolt-L0 = <850000 850000 1150000>;
41 opp-microvolt-L1 = <825000 825000 1150000>;
42 opp-microvolt-L2 = <825000 825000 1150000>;
43 clock-latency-ns = <40000>;
44 };
45 opp-816000000 {
46 opp-hz = /bits/ 64 <816000000>;
47 opp-microvolt = <850000 850000 1150000>;
48 opp-microvolt-L0 = <850000 850000 1150000>;
49 opp-microvolt-L1 = <825000 825000 1150000>;
50 opp-microvolt-L2 = <825000 825000 1150000>;
51 clock-latency-ns = <40000>;
52 opp-suspend;
53 };
54 /*.....*/
55 };
```

株式会社日昇テクノロジー

OPP ドライバはチップ内部のバージョン番号に基づいて、CPU コアの動作電圧と動作周波数を設定します。

カーネルが現在のシステムの CPU0 関連情報を理解した後、次に他の CPU をカーネルの管理下に置き、

作業を開始させる必要があります。SMP 初期化の前に、カーネルは present_mask を初期化し、その後 present_mask の内容に従って対応する CPU を起動します。具体的な実装方法は、possible_mask の値を present_mask にコピーすることです。

リスト 5: present_mask の初期化（arch/arm64/kernel/smp.c に位置）

```
1 void __init smp_prepare_cpus(unsigned int max_cpus)
2 {
3 int err;
4 unsigned int cpu;
5 unsigned int this_cpu;
6
7 init_cpu_topology(); //現在の/cpus ノードを解析し、CPU のトポロジー構造を取得
8
9 this_cpu = smp_processor_id();
10 store_cpu_topology(this_cpu); //DTS から CPU トポロジーを成功裏に取得できなかった場合、
ARM64 のレジスタから関連情報を取得
11 numa_store_cpu_info(this_cpu);
12 numa_add_cpu(this_cpu);
13
14 /*
15 * "nosmp"によって UP（シングルプロセッサ）が指定されている場合（これは"maxcpus=0"を意味す
る）、
16 * セカンダリ CPU をプレゼントとして設定しない。
17 */
```

株式会社日昇テクノロジー

```
18 if (max_cpus == 0)
19 return;
20
21 /*
22 * 現在実際に搭載されている CPU のセットを記述する present マップを初期化し、
23 * ブートローダーからセカンダリを解放する。
24 */
25 for_each_possible_cpu(cpu) {
26
27 per_cpu(cpu_number, cpu) = cpu;
28
29 if (cpu == smp_processor_id())
30 continue;
31
32 if (!cpu_ops[cpu])
33 continue;
34
35 err = cpu_ops[cpu]->cpu_prepare(cpu);
36 if (err)
37 continue;
38
39 set_cpu_present(cpu, true); //__cpu_present_mask を設定
```

株式会社日昇テクノロジー

```
40 numa_store_cpu_info(cpu);

41 }

42 }
```

リスト 6: smp_init 関数と cpu_up 関数（kernel/smp.c、kernel/cpu.c に位置）

```
1 /* ブートプロセッサによって残りをアクティブにするために呼び出されます。*/

2 void __init smp_init(void)

3 {

4 /* 一部のコードを省略 */

5 cpuhp_threads_init(); //各コアに"cpuhp/%u"カーネルスレッドなどを作成、現在の cpu の"cpuhp/%u"
スレッド："cpuhp/0"

6

7 pr_info("サブ CPU を起動中...¥n");

8 /* FIXME: これはユーザースペースで行うべきです --RR */

9 for_each_present_cpu(cpu) {

10 if (num_online_cpus() >= setup_max_cpus)

11 break;

12 if (!cpu_online(cpu))

13 cpu_up(cpu);

14 }

15 /*.....*/

16 }

17
```

株式会社日昇テクノロジー

```
18 int cpu_up(unsigned int cpu)
19 {
20 return do_cpu_up(cpu, CPUHP_ONLINE); //CPUHP_ONLINE は、cpu が到達すべき状態を示すパラメータです
21 }
```

smp_init()関数は present_mask 内の cpu を反復処理し、cpu がオンラインでない場合は cpu_up()関数を呼び出します。この関数は SMP 起動プロセスの最も重要な部分であり、その後 PSCI の操作関数 cpu_psci_ops を呼び出し、最終的に smc を使用して EL3 にトラップし、対応するサブプロセッサの起動サービスを要求します。これにより、サブプロセッサはカーネル（EL3->EL1）に戻り、secondary_entry で指定された命令を実行し、サブプロセッサの起動が完了します。

SMP システムが起動する過程で、すなわち電源が入った直後は、カーネルの初期化を行うために 1 つの CPU のみが使用されます。この CPU は「ブートプロセッサ」、つまり BP と呼ばれ、残りのプロセッサは一時停止状態であり、「アプリケーションプロセッサ」、つまり AP と呼ばれます。コードのコメントには、BP と AP の間の初期化プロセスの概要が示されています。左側は CPU の電源投入プロセスで、OFFLINE->BRINGUP_CPU->AP_OFFLINE->AP_ONLINE->AP_ACTIVE のプロセスを経る必要があります。

リスト 7: CPU の状態値の列挙（include/linux/cpuhotplug.h に位置）

```
1 /*
2 * CPU-up CPU-down
3 *
4 * BP AP BP AP
5 *
```

株式会社日昇テクノロジー

```
6 * OFFLINE OFFLINE
7 * | ^
8 * v |
9 * BRINGUP_CPU->AP_OFFLINE BRINGUP_CPU <- AP_IDLE_DEAD (idle thread/play_dead)
10 * |     AP_OFFLINE
11 * v (IRQ-off) ,---------------^
12 * AP_ONLNE | (stop_machine)
13 * | TEARDOWN_CPU <- AP_ONLINE_IDLE
14 * | ^
15 * v |
16 * AP_ACTIVE AP_ACTIVE
17 */
18
19 enum cpuhp_state {
20 CPUHP_INVALID = -1,
21 CPUHP_OFFLINE = 0,
22 /* …略…*/
23 CPUHP_AP_ONLINE_DYN_END = CPUHP_AP_ONLINE_DYN + 30,
24 CPUHP_AP_X86_HPET_ONLINE,
25 CPUHP_AP_X86_KVM_CLK_ONLINE,
26 CPUHP_AP_ACTIVE,
27 CPUHP_ONLINE,
28 };
```

株式会社日昇テクノロジー

カーネルは kernel/cpu.c 内で cpuhp_step 型の配列 cpuhp_hp_states を提供しており、配列内には初期化のコールバック関数がいくつか組み込まれています。これらのコールバック関数は初期化プロセスの各状態に対応しています。

リスト 8: cpuhp_hp_states 配列（kernel/cpu.c に位置）

```
1 static struct cpuhp_step cpuhp_hp_states[] = {
2 [CPUHP_OFFLINE] = {
3 .name = "offline",
4 .startup.single = NULL,
5 .teardown.single = NULL,
6 },
7 #ifdef CONFIG_SMP
8
9 [CPUHP_CREATE_THREADS]= {
10 .name = "threads:prepare",
11 .startup.single = smpboot_create_threads, //回调函数
12 .teardown.single = NULL,
13 .cant_stop = true,
14 },
15 [CPUHP_PERF_PREPARE] = {
16 .name = "perf:prepare",
17 .startup.single = perf_event_init_cpu,
18 .teardown.single = perf_event_exit_cpu,
```

株式会社日昇テクノロジー

```
19 },
20 /*.....*/
21 };
```

以下に、OFFLINE->BRINGUP_CPU のプロセスを見てみましょう。前述の do_cpu_up 関数は_cpu_up 関数を呼び出し、パラメータ target に CPUHP_ONLINE（cpuhp_state の最大値）を渡します。コードの 4 行目は CPUHP_ONLINE と CPUHP_BRINGUP_CPU を比較し、より小さい値、つまり CPUHP_BRINGUP_CPU を返します。

リスト 9: _cpu_up 関数（kernel/smp.c に位置）

```
1 static int _cpu_up(unsigned int cpu, int tasks_frozen, enum cpuhp_state target)
2 {
3 /* 略*/
4 target = min((int)target, CPUHP_BRINGUP_CPU);
5 ret = cpuhp_up_callbacks(cpu, st, target);
6 out:
7 cpus_write_unlock();
8 arch_smt_update();
9 cpu_up_down_serialize_trainwrecks(tasks_frozen);
10 return ret;
11 }
```

cpuhp_up_callbacks 関数の役割は、関数名が示すように、cpuhp_hp_states 配列に提供されている初期化関数を呼び出すことです。

株式会社日昇テクノロジー

リスト 10: cpuhp_up_callbacks 関数（kernel/cpu.c に位置）

```
1 static int cpuhp_up_callbacks(unsigned int cpu, struct cpuhp_cpu_state *st,
2 enum cpuhp_state target)
3 {
4 enum cpuhp_state prev_state = st->state;
5 int ret = 0;
6
7 while (st->state < target) {
8 st->state++;
9 ret = cpuhp_invoke_callback(cpu, st->state, true, NULL, NULL); //各ステップの切り替え時には、この
 関数が呼び出されて完了します。
10 if (ret) {
11 if (can_rollback_cpu(st)) {
12 st->target = prev_state;
13 undo_cpu_up(cpu, st);
14 }
15 break;
16 }
17 }
18 return ret;
19 }
```

st->state は、現在の AP コアの状態を記録しており、デフォルトでは AP は CPUHP_OFFLINE 状態にあります。そのため、cpuhp_up_callbacks 関数は cpuhp_hp_states 配列に提供されている

株式会社日昇テクノロジー

（CPUHP_OFFLINE+1）から CPUHP_BRINGUP_CPU までのすべてのフェーズのコールバック関数を実行し、現在の AP コアの起動を行います。BRINGUP_CPU の状態を経た後、現在の AP コアはアイドルタスクを実行し、同時に BP コアは cpuhp/0 プロセスを起動し、CPUHP_AP_ONLINE_IDLE->CPUHP_ONLINE のプロセスを完了します。具体的な実装プロセスは以下の通りです：

リスト 11: cpuhp_thread_fun 関数（kernel/smp.c に位置）

```
1 static void cpuhp_thread_fun(unsigned int cpu)
2 {
3 if (cpuhp_is_atomic_state(state)) {
4 local_irq_disable();
5 st->result = cpuhp_invoke_callback(cpu, state, bringup, st->node, &st->last);
6 local_irq_enable();
7 WARN_ON_ONCE(st->result);
8 } else {
9 st->result = cpuhp_invoke_callback(cpu, state, bringup, st->node, &st->last);
10 }
11
12 }
```

このプロセスでは、以前に述べた cpuhp_invoke_callback 関数が再び呼び出され、最終的に AP コアが CPUHP_ONLINE の状態に達し、カーネルによってスケジュールされ、BP コアと一緒に作業負荷を負担します。上記のプロセスは単一の AP コアの起動プロセスを分析したものですが、現在のシステムに複数の AP コアがある場合、カーネルは各 AP コアに対して同様の操作を実行し、すべての AP コアが成功裏に起動するまで続けます。

株式会社日昇テクノロジー

第 22 章 Sysfs

前の章でドライバを学習した際、テスト時によく/sys ディレクトリ下のファイルを使用しました。この章では Sysfs ファイルシステムについて簡単に紹介します。

Sysfs はメモリファイルシステム（configfs、debugfs なども含む）で、カーネルオブジェクトをユーザースペースにエクスポートする方法を提供し、kobject オブジェクト階層のビューを表示します。ユーザーは一部のカーネルパラメータを表示または設定できます。kobject 階層は、システムのハードウェア関連のデバイス、ドライバ、バスなどの階層関係に対して、Linux 統一デバイスモデルとして管理されます。

22.1 Sysfs のディレクトリ構造

よく知っているディスクファイルシステムと比べて、Sysfs ファイルシステムは仮想ファイルシステムの一種です。つまり、ファイルシステム内のファイルはディスク上のどのファイルにも対応せず、メモリ内に存在します。このファイルシステムは初期化時にデフォルトで/sys にマウントされ、以下のコマンドを使用してマウントすることができます（他の場所にもマウント可能です）。

```
mount -t sysfs sysfs /sys
```

一般的な状況での/sys のディレクトリ構造：

```
cat@lubancat:/$ ls -l /sys

total 0

drwxr-xr-x 2 root root 0 Oct 19 09:49 block

drwxr-xr-x 33 root root 0 Oct 19 09:49 bus

drwxr-xr-x 69 root root 0 Oct 19 09:49 class

drwxr-xr-x 4 root root 0 Oct 19 09:49 dev

drwxr-xr-x 11 root root 0 Oct 19 09:49 devices

drwxr-xr-x 3 root root 0 Oct 19 09:49 firmware
```

株式会社日昇テクノロジー

```
drwxr-xr-x 8 root root 0 Oct 19 09:49 fs

drwxr-xr-x 12 root root 0 Oct 19 09:49 kernel

drwxr-xr-x 151 root root 0 Oct 19 09:49 module

drwxr-xr-x 3 root root 0 Oct 19 09:49 power
```

これらのディレクトリは、サブシステムがカーネル内で kobject を登録する際に、システムの起動時に生成されます。初期化後、それらは各自のディレクトリ内で登録されたオブジェクトを探索します。kobject に対応するディレクトリが 1 つあり、含まれるオブジェクト属性がファイルに対応します。ファイルはディレクトリ、通常のファイル（テキストまたはバイナリファイル）、シンボリックリンクファイルの 3 種類のみをサポートします。

22.1.1 block ディレクトリ

ここには、システム内のすべての現在のブロックデバイスがあります。機能的には/sys/class の下に置く方が適切ですが、歴史的な遺産のために/sys/block に存在しています。ブロックデバイスは現在移動されています。

```
cat@lubancat:/sys/block$ ls -l

total 0

lrwxrwxrwx 1 root root 0 Oct 19 10:12 loop0 -> ../devices/virtual/block/loop0

lrwxrwxrwx 1 root root 0 Oct 19 10:12 loop1 -> ../devices/virtual/block/loop1

lrwxrwxrwx 1 root root 0 Oct 19 10:12 loop2 -> ../devices/virtual/block/loop2

lrwxrwxrwx 1 root root 0 Oct 19 10:12 loop3 -> ../devices/virtual/block/loop3

lrwxrwxrwx 1 root root 0 Oct 19 10:12 loop4 -> ../devices/virtual/block/loop4

lrwxrwxrwx 1 root root 0 Oct 19 10:12 loop5 -> ../devices/virtual/block/loop5

lrwxrwxrwx 1 root root 0 Oct 19 10:12 loop6 -> ../devices/virtual/block/loop6
```

株式会社日昇テクノロジー

```
lrwxrwxrwx 1 root root 0 Oct 19 10:12 loop7 -> ../devices/virtual/block/loop7
lrwxrwxrwx 1 root root 0 Oct 19 10:12 mmcblk0 -
 > ../devices/platform/fe310000.sdhci/mmc_host/mmc0/mmc0:0001/block/mmcblk0
lrwxrwxrwx 1 root root 0 Oct 19 10:12 mmcblk0boot0 -
> ../devices/platform/fe310000.sdhci/mmc_host/mmc0/mmc0:0001/block/mmcblk0/mmcblk0boot0
lrwxrwxrwx 1 root root 0 Oct 19 10:12 mmcblk0boot1 -
> ../devices/platform/fe310000.sdhci/mmc_host/mmc0/mmc0:0001/block/mmcblk0/mmcblk0boot1
lrwxrwxrwx 1 root root 0 Oct 19 10:12 mmcblk1 -
> ../devices/platform/fe2b0000.dwmmc/mmc_host/mmc1/mmc1:aaaa/block/mmcblk1
lrwxrwxrwx 1 root root 0 Oct 19 10:12 ram0 -> ../devices/virtual/block/ram0
lrwxrwxrwx 1 root root 0 Oct 19 10:12 zram0 -> ../devices/virtual/block/zram0
```

22.1.2 bus、devices、class ディレクトリ

これらのディレクトリは、Linux 統一デバイスモデルと密接な関係があり、Linux 統一デバイスモデルの一部を構成しています。

bus ディレクトリ内の各サブディレクトリは、登録されたバスタイプです。これは、デバイスがバスタイプによって分類されるディレクトリ構造です。各サブディレクトリ（バスタイプ）内には、devices と drivers の 2 つのサブディレクトリがあります。devices 内にはそのバスタイプ下のすべてのデバイスがあり、これらのデバイスはシンボリックリンクであり、それぞれが/sys/devices/下の実際のデバイスを指しています。drivers 内には、このバス上に登録されたすべてのドライバがあり、各ドライバのサブディレクトリ内にはドライバパラメータを観察および変更できるものがあります。

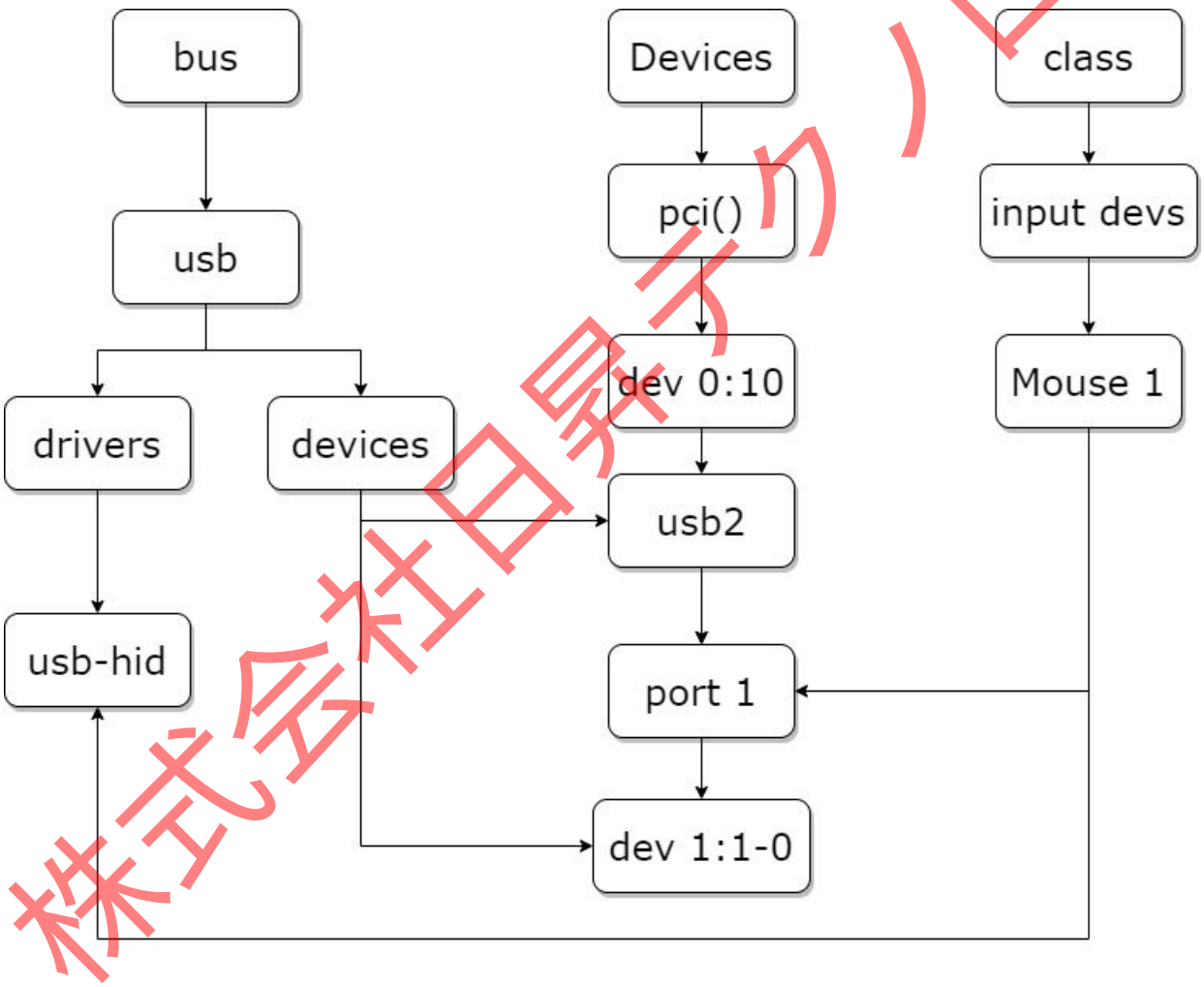
devices ディレクトリ内には、各種バス上に登録された物理デバイスが全て含まれています。一般的に、すべての物理デバイスは、バス上のトポロジーに従って表示されます。/sys/devices は、カーネルがシステ

株式会社日昇テクノロジー

ム内のすべてのデバイスを階層的に表現するモデルであり、/sys ファイルシステム内でデバイスを管理するための最も重要なディレクトリ構造です。

class ディレクトリ内には、カーネル内に登録されたすべてのデバイスタイプが含まれています。これは、デバイス機能によるデバイスの分類です。各種デバイスには特定の機能があります。たとえば、マウスは人間とコンピュータ間のインタラクションの入力として機能します。デバイス機能による分類では、それがどのバスに接続されているかに関係なく、すべてのマウスは/sys/class/input 下に分類されます。

以下の画像はこれらの関連性を直感的に示しています：



株式会社日昇テクノロジー

22.1.3 firmware ディレクトリ

このディレクトリには、ファームウェアオブジェクトと属性を含むサブディレクトリが含まれています。カーネルのファームウェアのロードとファームウェアドライバに関心がある場合は、自分で調べてみると

良いでしょう。このディレクトリには：

```
cat@lubancat:/sys/firmware$ ls -l
total 0
drwxr-xr-x 3 root root 0 Oct 19 11:34 devicetree
-r-------- 1 root root 163840 Oct 19 11:34 fdt
```

devicetree フォルダがあり、デバイスツリー情報のロードを記述します。ルートノードは base ディレクトリに対応し、各デバイスツリーノードは一つのディレクトリに対応し、各属性は一つのファイルに対応します。fdt ファイルは元の dtb ファイルで、uboot からカーネルに渡されるデバイスツリーファイルです。hexdump -C で見ることができます。

22.1.4 fs ディレクトリ

このディレクトリは、システム内のすべてのファイルシステム、ファイルシステム自体、およびファイルシステムに分類されたマウントポイントを記述するために使用されます。登録されたファイルシステムのビューを記述しますが、現在は fuse、ext4 など少数のファイルシステムのみが sysfs インターフェースをサポートしています。一部の従来の仮想ファイルシステム（VFS）の階層制御パラメータは依然として sysctl(/proc/sys/fs)インターフェースにあります。ディレクトリ構造は以下の通りです：

```
cat@lubancat:/sys/fs$ ls -l
total 0
drwx------T 2 root root 0 Oct 19 14:41 bpf
drwxr-xr-x 10 root root 280 Oct 19 14:41 cgroup
drwxr-xr-x 5 root root 0 Oct 19 16:58 ext4
drwxr-xr-x 3 root root 0 Oct 19 14:41 fuse
drwxr-x--- 2 root root 0 Oct 19 14:41 pstore
```

株式会社日昇テクノロジー

```
drwxr-xr-x 3 root root 0 Oct 19 16:58 xfs
```

cgroup や bpf ファイルに興味がある場合は、調べてみると良いでしょう。

22.1.5 kernel ディレクトリ

このディレクトリは、調整可能なすべてのカーネルパラメータがある場所です。いくつかのカーネル調整可能パラメータは依然として sysctl(/proc/sys/kernel)インターフェースにあります。

22.1.6 module ディレクトリ

このディレクトリには、システム内のすべてのモジュールの情報が含まれています。これらのモジュールがカーネルイメージファイル(vmlinuz)にインラインでコンパイルされているか、外部モジュール(.ko ファイル)としてコンパイルされているかにかかわらず、/sys/module ディレクトリに表示されることがあります。

外部モジュール(.ko ファイル)は、ロード後に/sys/module/に対応するモジュールフォルダが表示され、このフォルダにはいくつかの属性ファイルと属性ディレクトリが表示され、外部モジュールの情報（バージョン、ロード状態、提供されるドライバなど）が表示されます。

カーネルにコンパイルされたモジュールは、非 0 属性のモジュールパラメータがある場合にのみ、対応する/sys/module/に表示されます。これらのモジュールの利用可能なパラメータは、/sys/modules/parameters/に表示されます。例えば、/sys/module/printk/parameters/time この読み書き可能なパラメータは、インラインモジュール printk がカーネルメッセージを出力する際に時間プレフィックスを付けるかどうかを制御します。

```
cat@lubancat:/sys/module/printk/parameters$ ls -al

total 0

drwxr-xr-x 2 root root 0 Oct 19 14:34 .
```

株式会社日昇テクノロジー

```
drwxr-xr-x 3 root root 0 Oct 18 15:12 ..
-rw-r--r-- 1 root root 4096 Oct 19 14:36 always_kmsg_dump
-rw-r--r-- 1 root root 4096 Oct 19 14:36 console_suspend
-rw-r--r-- 1 root root 4096 Oct 19 14:36 ignore_loglevel
-rw-r--r-- 1 root root 4096 Oct 19 14:36 time
cat@lubancat:/sys/module/printk/parameters$ cat time
Y
cat@lubancat:/sys/module/printk/parameters$ echo 0 | sudo tee time
0
cat@lubancat:/sys/module/printk/parameters$ cat time
N
cat@lubancat:/sys/module/printk/parameters$
```

すべてのインラインモジュールのパラメータは、カーネル起動時のパラメータに“.=”の形式で記述することもできます。例えば、カーネルを起動する際に“printk.time=1”というパラメータを追加すると、“/sys/module/printk/parameters/time”に 1 を書き込むのと同じ効果が得られます。非 0 属性パラメータを持たないインラインモジュールはここには表示されません。

22.1.7 power ディレクトリ

このディレクトリには、システムの電源オプションが含まれており、電源管理サブシステムが提供する統一インターフェースファイルが含まれています。一部の属性ファイルは、シャットダウン、再起動などの操作を行うために書き込むことができます。

22.2 Sysfs の使用

sysfs モデルの中心は kobject です。kobject オブジェクトをディレクトリエントリに密接に関連付け、

株式会社日昇テクノロジー

kobject をディレクトリ上にマッピングします。struct Kobject は、<linux/Kobject.h>で定義されており、カーネルオブジェクト（例えばデバイスなど）を表し、sysfs ファイルシステムでディレクトリとして表示される内容です。

ここでは、デバイスモデルの中核となるデータ構造 kobject について簡単に紹介します：

リスト 1: kobject（include/linux/kobject.h に定義されている）

```
1 struct kobject {
2 const char *name;
3 struct list_head entry;
4 struct kobject *parent;
5 struct kset *kset;
6 struct kobj_type *ktype;
7 struct kernfs_node *sd; /* sysfs directory entry */
8 struct kref kref;
9 /*....*/
10 }
```

- name: kobject の名前。/sys ディレクトリ内でのディレクトリ名はこれを使って作成されます。

- parent: 親 kobject ノード。/sys ディレクトリ内でこの親ディレクトリ下にディレクトリが作成されます。

- kset: kobject を分類するために使用されます。同じ属性を持つ kobject は同じセットに属します。

- ktype: kobject 関連の属性とオブジェクトの操作方法を定義します。この構造体の詳細は、カーネルの include/linux/kobject.h で確認できます。

- kref: 参照カウント。ゼロになるとデストラクタ関数が呼び出されます。

- sd: kobject_create_and_add 関数を使用する際に、sysfs_dirent と関連付けられ、マッピングを実現し

株式会社日昇テクノロジー

ます。

22.2.1 ディレクトリの作成

```
struct kobject * kobject_create_and_add (const char * name, struct kobject * parent);
```

- name: 作成する kobject の名前。

- parent: 親 kobject を指定します。これにより、特定のディレクトリ下に新しいディレクトリが作成されます。例えば kernel_kobj を指定すると、/sys/kernel ディレクトリ下にディレクトリが作成されます。NULL を指定すると、/sys ディレクトリ下に作成されます。

22.2.2 ファイルの作成

int sysfs_create_file (struct kobject * kobj, const struct attribute * attr);

- kobj: 作成した kobject。

- attr: 属性の説明。

より多くの関数はカーネルソースコードの include/linux/sysfs.h ファイルで参照できます。

22.3 簡単な実験

前節の関数インターフェースに基づいて、ドライバプログラムを簡単に作成してみます。/sys ディレクトリ下に sysfs_test ディレクトリを作成し、そのディレクトリ下にファイルを作成します。

サンプルコードディレクトリ: linux_driver/Sysfs

22.3.1 プログラムソースコード

リスト 2: linux_driver/Sysfs/sys_test.c

```
1 #include <linux/kernel.h>

2 #include <linux/init.h>

3 #include <linux/module.h>
```

株式会社日昇テクノロジー

```
4 #include <linux/fs.h>

5 #include<linux/sysfs.h>

6 #include<linux/kobject.h>

7 #include <linux/err.h>

8

9 volatile int test_value = 0;

10 struct kobject *kobj_test;

11

12 /* この関数は sysfs ファイルが読み込まれる時に呼ばれます*/

13 static ssize_t sysfs_show(struct kobject *kobj,

14 struct kobj_attribute *attr, char *buf)

15 {

16 pr_info("sysfs を読み込みます！¥n");

17 return sprintf(buf, "test_value = %d¥n", test_value);

18 }

19

20 /* この関数は sysfs ファイルが書き込まれる時に呼ばれます*/

21 static ssize_t sysfs_store(struct kobject *kobj,

22 struct kobj_attribute *attr,const char *buf, size_t count)

23 {

24 pr_info("sysfs に書き込みます！¥n");

25 sscanf(buf,"%d",&test_value);
```

株式会社日昇テクノロジー

```
26 return count;

27 }

28

29 /* __ATTR マクロを使用して sysfs_test_attr 構造体を初期化します、このマクロは
include/linux/sysfs.h に定義されています*/

30 struct kobj_attribute sysfs_test_attr = __ATTR(test_value, 0664, sysfs_show, sysfs_store);

31

32 /* モジュール初期化関数*/

33 static int __init sysfs_test_driver_init(void)

34 {

35 /* /sys 下にディレクトリを作成します*/

36 kobj_test = kobject_create_and_add("sysfs_test",NULL);

37

38 /* sysfs_test ディレクトリ下にファイルを作成します*/

39 if(sysfs_create_file(kobj_test,&sysfs_test_attr.attr)){

40 pr_err("sysfs ファイルの作成に失敗しました.....¥n");

41 goto error_sysfs;

42 }

43

44 pr_info("ドライバモジュールの初期化完了！¥n");

45 return 0;

46
```

株式会社日昇テクノロジー

```
47 error_sysfs:
48 kobject_put(kobj_test);
49 sysfs_remove_file(kernel_kobj, &sysfs_test_attr.attr);
50 return -1;
51
52 }
53
54 /* モジュール終了関数*/
55 static void __exit sysfs_test_driver_exit(void)
56 {
57 kobject_put(kobj_test);
58 sysfs_remove_file(kernel_kobj, &sysfs_test_attr.attr);
59 pr_info("デバイスドライバモジュールが削除されました！¥n");
60 }
61
62 module_init(sysfs_test_driver_init);
63 module_exit(sysfs_test_driver_exit);
64
65 MODULE_LICENSE("GPL");
66 MODULE_AUTHOR("Embedfire");
67 MODULE_DESCRIPTION("sysfs を使用する簡単なドライバプログラム");
```

実際の上層ドライバモデル（cdev、bus、device など）では、kobject 構造体が内蔵されており、直接使用されます。カーネルは対応するディレクトリ内にファイルを作成します。

株式会社日昇テクノロジー

22.3.2 テスト

クロスコンパイラを使用してコンパイルし、sys_test.ko モジュールを生成し、ボードに転送した後、コマンド sudo insmod sys_test.ko を使用してカーネルモジュールをロードします。その後、/sys/sysfs_test ディレクトリ内で作成されたファイルの読み書きを行います：

```
root@lubancat:/home/cat#
root@lubancat:/home/cat#
root@lubancat:/home/cat#
root@lubancat:/home/cat# cd /sys
root@lubancat:/sys# ls
block  class  devices   fs       module  rk8xx        system_monitor
bus    dev    firmware  kernel   power   sysfs_test
root@lubancat:/sys# cd sysfs_test/
root@lubancat:/sys/sysfs_test# ls
test_value
root@lubancat:/sys/sysfs_test# cat test_value
test_value = 0
root@lubancat:/sys/sysfs_test# echo 2 > test_value
root@lubancat:/sys/sysfs_test# cat test_value
test_value = 2
root@lubancat:/sys/sysfs_test# ls -al
total 0
drwxr-xr-x  2 root root    0 Oct 20 10:11 .
dr-xr-xr-x 15 root root    0 Oct 20 10:11 ..
-rw-rw-r--  1 root root 4096 Oct 20 10:11 test_value
root@lubancat:/sys/sysfs_test#
```

22.4 まとめ

デバイスドライバプログラムがユーザーレベルとインターフェースを必要とする場合、一般的に複数の方法があります。仮想キャラクターデバイスファイルを登録し、ユーザースペースは read/write/ioctl を介してインタラクションできます；proc インターフェイスをサポートして登録し、これはオペレーティングシステム自体とアプリケーション間の安全なインターフェースを提供し、主にプロセス情報とカーネルの各サブシステムの情報を表示しますが、現在は一般的にデバイスドライバには使用されません；sysfs 属性を登録し、直接ディレクトリと属性ファイルを追加し、ユーザーレイヤーがスクリプトまたはコマンドを介して直接操作できます。

/sys ディレクトリ下のファイルに関する詳細な説明は、カーネルソースコードの Documentation/ABI/ 下のドキュメントを参照してください。これらのドキュメントは、/sys 下のファイルの属性と使用方法を

株式会社日昇テクノロジー

記述しています。

以上。

株式会社日昇テクノロジー