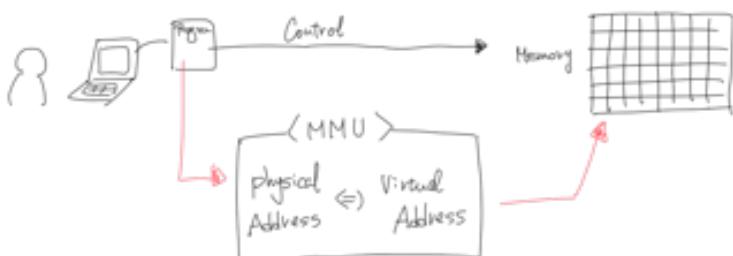
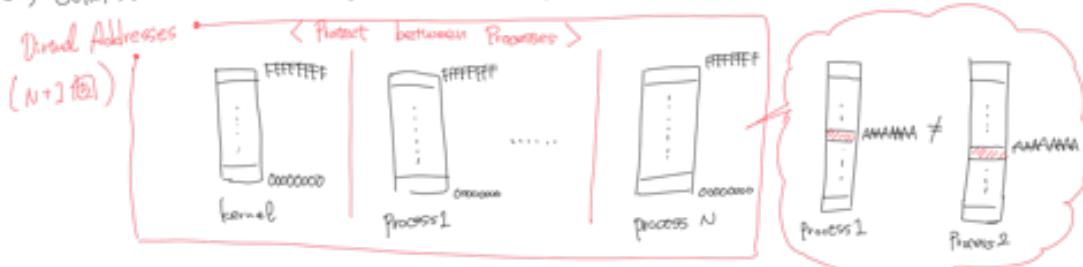


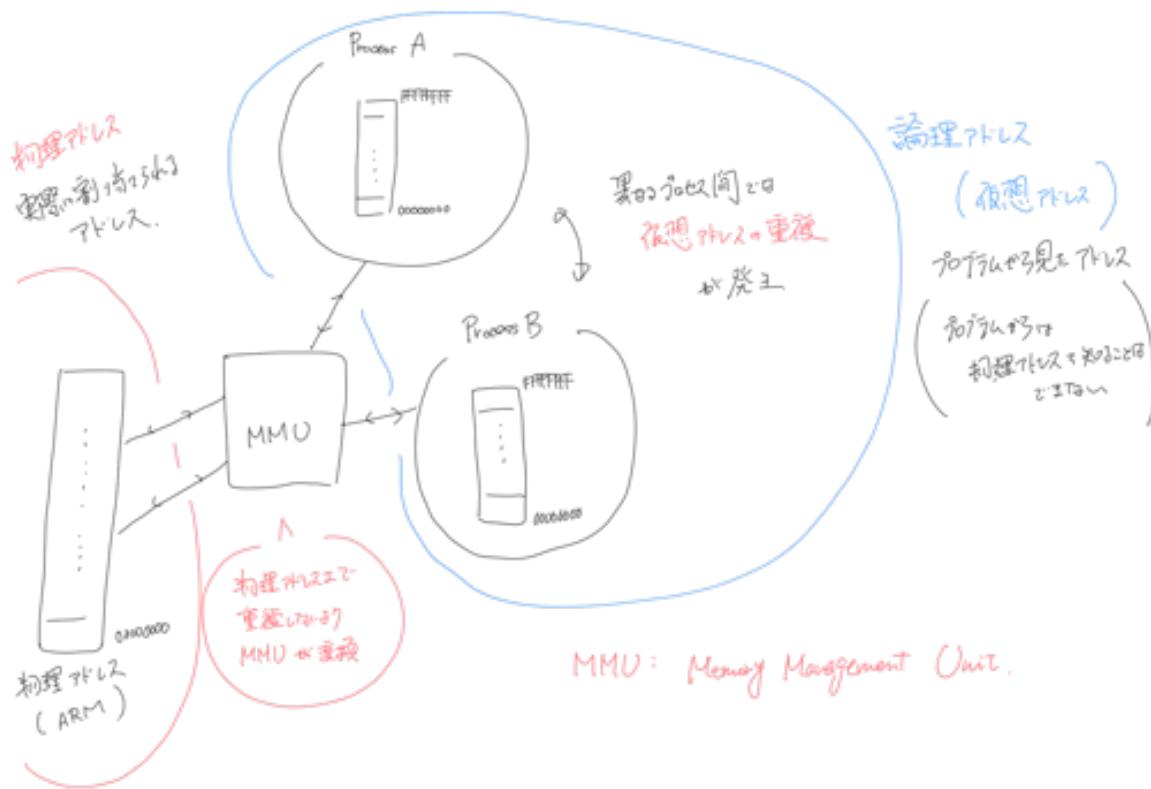
② 我々の作成したプログラムがメモリ上のアドレスにアクセスする時、直接物理アドレスは使わない。

↳ CPUは MMU という機能を持つので、物理アドレス ⇔ 虚擬アドレス変換を行う

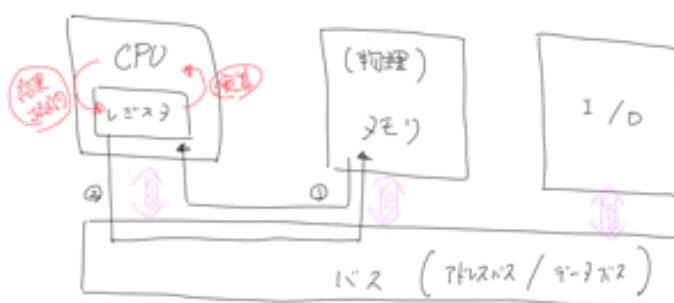


ex) 50個のプロセス ⇒ 51個 + (1+2+...+49) = 51個の虚擬記憶空間を有する.





通常の CPU・モデルを実行（起動簡単版）



- (①) CPU がメモリからレジスターに値を読み込んで (load) 機械。  
 (②) レジスターの値をメモリに書き込む機関 (store) -> 書き込み。

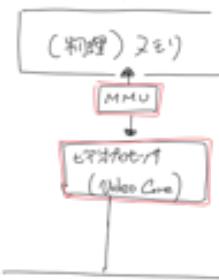
ବୁଦ୍ଧ ପାଳି,

三好市・ルクア前(2-テ空間)地図

カーネル内でもアレンamiento

27事。(記念、歴史=32、記念日=23指し=3保障の法)

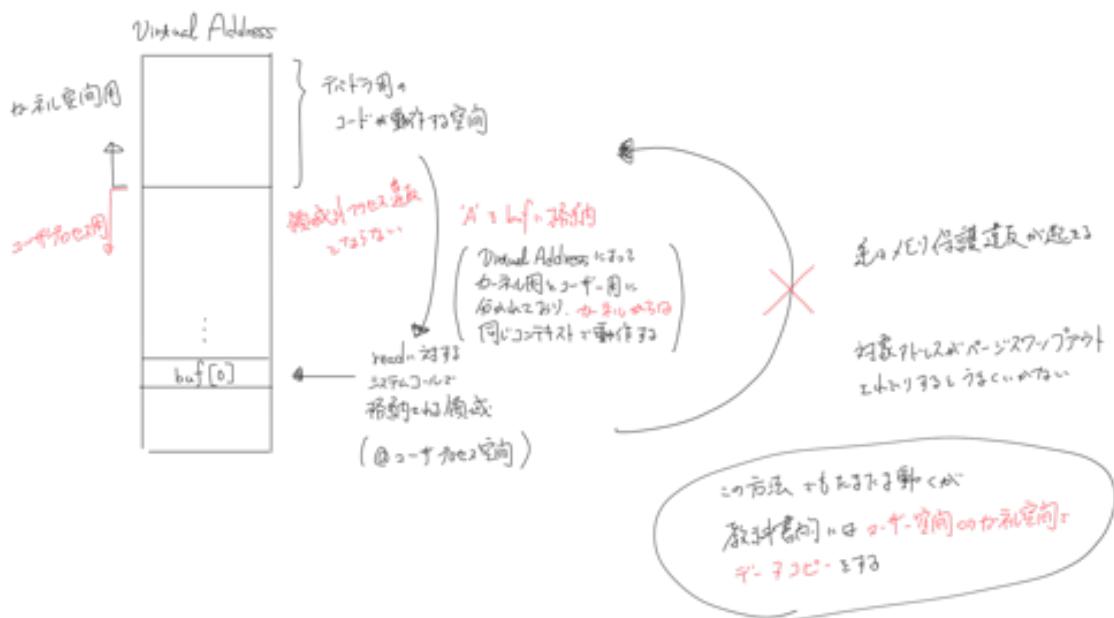
\* 実際はCPUからバス間にUCPが  
取扱う(どうみで、メモリと向かい面)



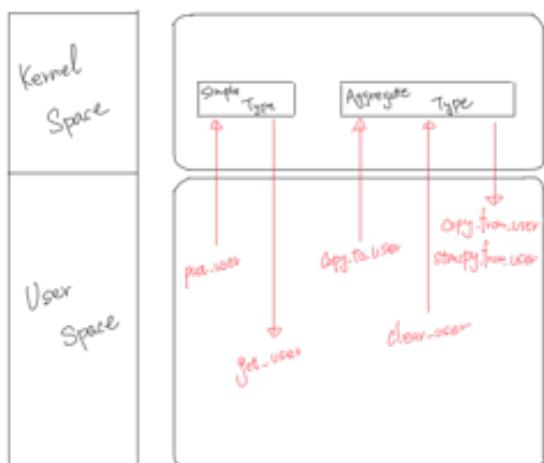
CPU がメモリにアクセスする門、  
CPU の動作を規制する  
とよばれる MMU(メモリマネージメントユニット)

二十、空间  $\leftrightarrow$  未来空间  $\rightarrow$  聚合モード

(三) 窗幕



书-条心 API



1-37-2 安全工場

( API \* p )

access.ok : 2-1-空间元组,如(2,7,9,西边,4,2,7)

Jst. 688r : 二千零四十三年正月五日

pot.user : 二、半空间。单纯形函数。进阶。

`clear-user` : `3.1.8@.java.sun.com` (no-ssl)

③ Copy to user : 如果是平-7, 那么2个空间人口点

Copy from user : 2-1-空间+2-2-运动和时间~10

strm[un-user] : -4- 留同, 2407-K, 75-402-頭稿.

$$\text{Starting from zero} : 2 \left( 2 \times 10^{-3} \right) = 2 + 2.6 = 2.6 \times 10^{-3}$$

卷四

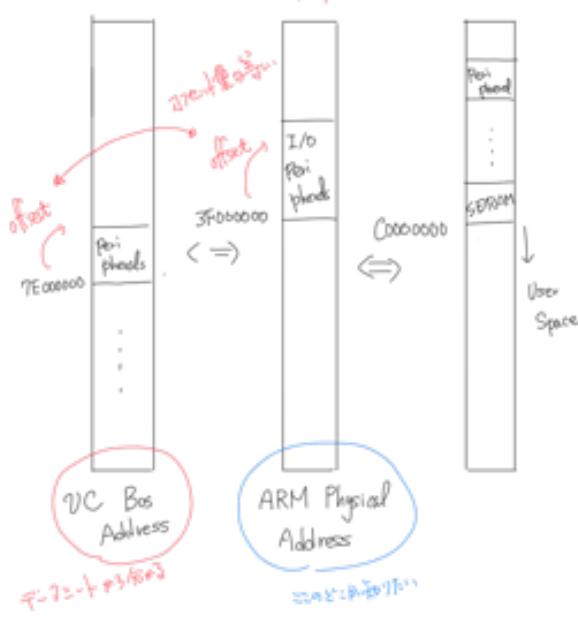
read 题： 七-九区间  $\rightarrow$  二-十区间  $\sim 7-13^{\circ}$

→ 有機→無機空間へ→→

↳ copy-from-user 回收

アドレスの変換とアクセス

↳ peripheral Address = 3300



裸、書く際の I/O Peripheral Address

↳ CPU の物理地址

↓ MMU で変換

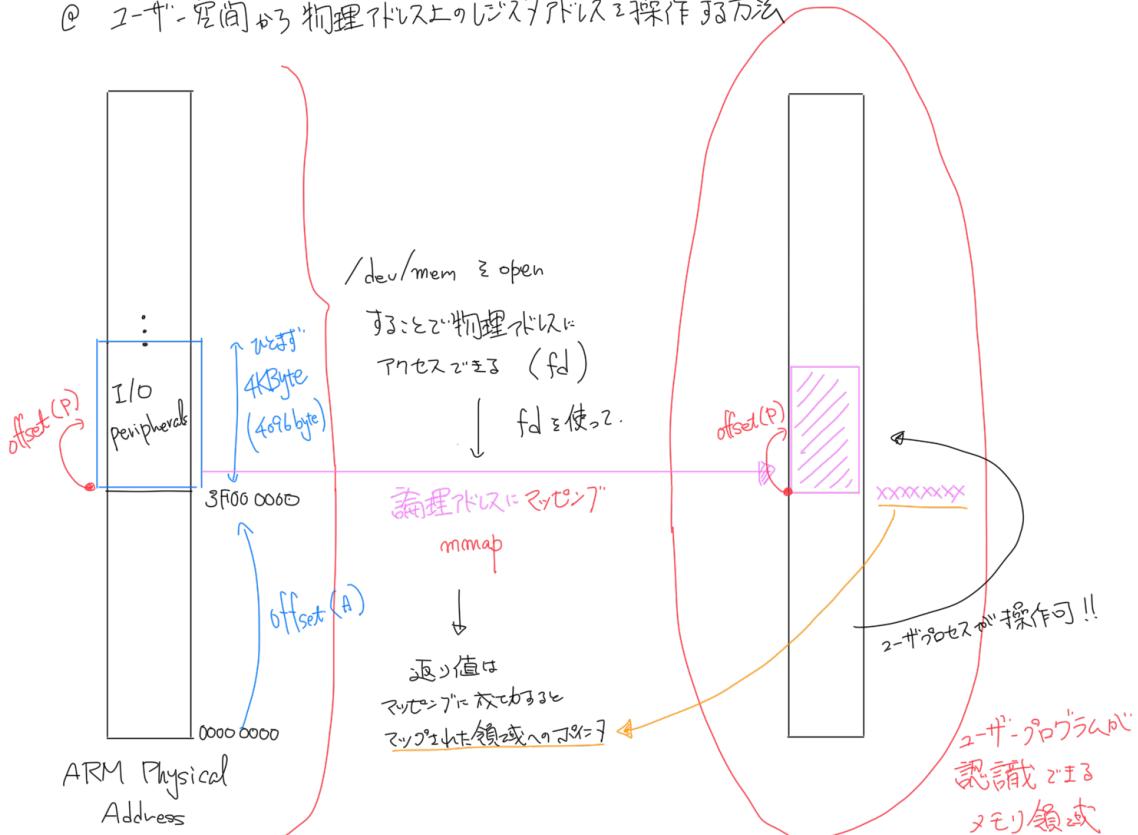
↑ I/O メモリ空間

↓ MMU で変換 (開始 07E000000)

↓ ( 物理ア  
    ドレッシング )

- ① I/O メモリ空間のアドレスを CPU が直接見るのは不可能  
peripheral の開始アドレス + フィルタリング + 値を直接見るのは不可能  
「物理アドレッシング」
- ② 物理アドレッシングは MMU で変換
- CPU の物理アドレッシング + peripheral のアドレス + フィルタリング
- ↳ bus host -> peripheral address の順序  
物理アドレッシングの位置はある?

## ② ユーザー空間から物理アドレス上のレジスタアドレスを操作する方法



/dev/mem - コンピュータのメモリ-イオ-キャッシュ-デバイスファイル  
このデバイスはシステムの便宜のために使われる

open(2), read / write / lseek で読み書きすると物理アドレスの読み書きができる

mmap ( NULL, 4096, PORT\_READ, PORT\_WRITE, MAP\_SHARED, fd, offset (A) )

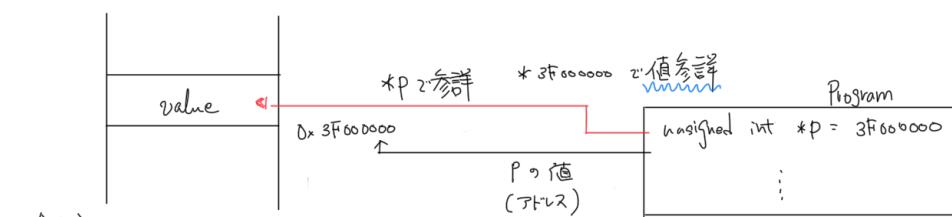
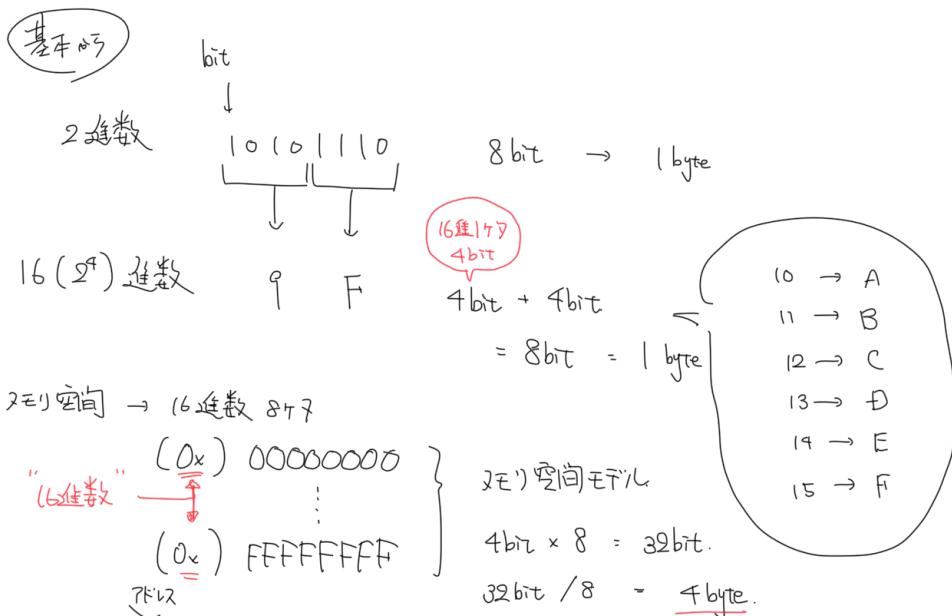
fd が参照しているファイルの offset (A) から開始して 4096 バイトのデータをマップ

マッピングのメモリ保護

{ リード可 書き込み可 }  
他のプロセスと共有される等

仮想アドレスのどこからでもアクセス可能  
NULL ならカーネルが決めてくる

シフト演算と用法とコスト計算.



（今回）

知らないのが、既存のアドレス 0x3F000000  
に入れる値 value.  
つまり  $\ast(0x3F000000)$  を参照している  
ただし、アドレス計算の実際で 0x3F000000 は  $\ast$   
使いつつも、変数 (unsigned int \*p) に入る ( $\ast$ )  
つまり、値の参照  $\ast p$  で行なう  
正確には  $\ast$  と  $\ast$  の間に、計算後のアドレスをポインタ変数に入れる

（通常）  
値 (int) と既存の変数 (a) がある。  
このアドレス ( $\&a$ ) はポインタ型 (int \*)  
の変数 (p) に入れた。  
 $p \rightarrow$  アドレス (16進数)  
 $\ast p \rightarrow$  アドレスの指し値 (a)  
 $\&p \rightarrow$  アドレスのポインテ (今日3送)

## GPIO制御

各レジ斯特を制御して、GPIOの制御を行う。

今回も、GPIO 4番ピンを出力で1にし、High / Low 制御する。

GPFSEL, GPSET, GPCR, レジ斯特等

GPLEV (リストも見てね)

動作は他のものの使い方が正しいから  
判断で2つ目がたぶん間違

② GPFSEL ( GPIO Function Select Register )

GPIOピンの機能設定を行う

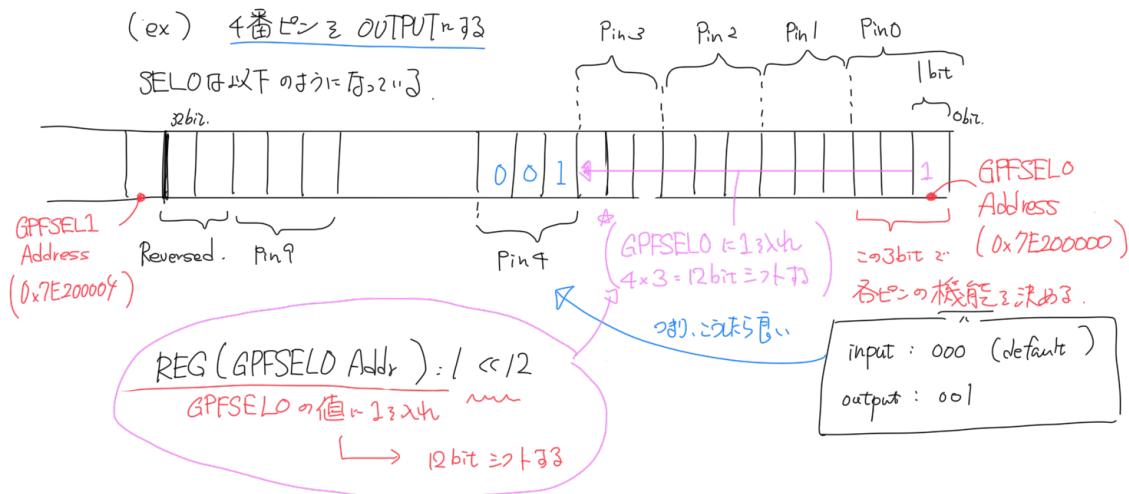
GPESEL0 ~ GPFSEL5 の6つのレジストが各4bit (32bit) あり。

目的ピンの設定をどこかで行う。 ( 4番ピンは SEL0, 6番ピンは SEL1 )

設定値は詳細は省略だが、ピンを OUTPUTにする時、0b001を設定する。

(ex) 4番ピンを OUTPUTにする

SEL0以下のようになります。



## ④ GPSET ( GPIO Pin Output Set Register )

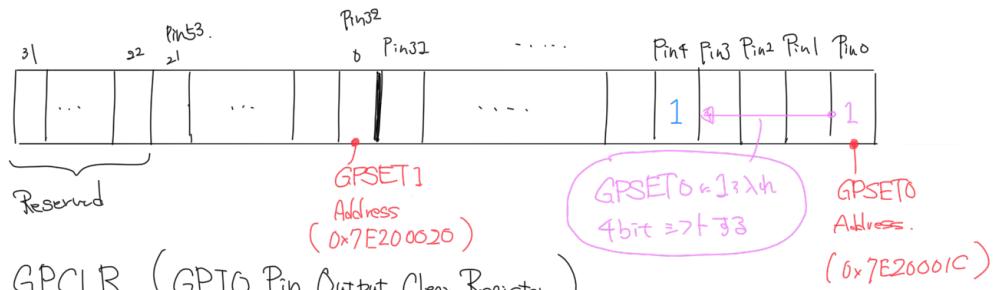
GPIO ピンの出力設定を行う。 ( High : 設定するものと思って書きとう )

GPSET0 , GPSET1 の 2つのレジ斯特リが各 32bit ( 4 バイト ) あり。

目的ピンの設定を手動で行う。 ( 4番ピンは SET0 , 40番ピンは SET1 )

各ビットが 1bit で対応している。 0 は no effect . 1 は High ( Set )

( ex ) 4番ピンの出力を High : ( 4番ピンを SET )



## ⑤ GPCLR ( GPIO Pin Output Clear Register )

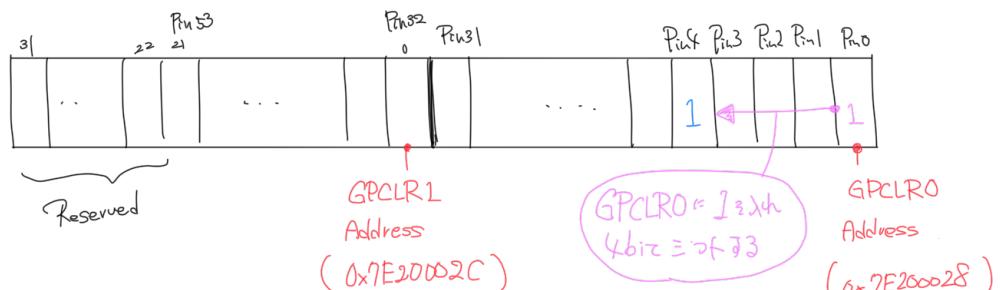
GPIO ピンの出力設定を行う ( Low : 設定するものと思って書きとう )

GPCLR0 , GPCLR1 の 2つのレジ斯特リが各 32bit ( 4 バイト ) あり。

目的ピンの設定を手動で行う ( 4番ピンは CLR0 , 40番ピンは CLR1 )

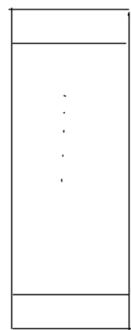
各ビットが 1bit で対応している。 0 は no effect . 1 は Low ( Clear )

( ex ) 4番ピンの出力を Low : ( 4番ピンを Clear )



## カーネル空間（カーネル仮想アドレス空間）

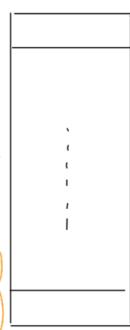
- OSのカーネルが存在する
- (Linuxでは)全カーネルスレッドが存在している



★ カーネル・ティベドラはここで動く  
 ★ 全てのメモリースペースにフルアクセス可能（システムによって可能な限り）  
 OS毎に異なるがシステムコールがある

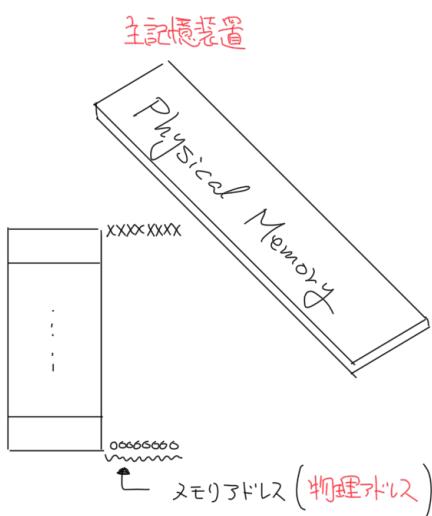
## ユーザ空間（ユーザ仮想アドレス空間）

- ユーザプログラムが動作する
- プロセス毎に割り当てられる
- /dev下にデバイスがあり、 ioctlも使う
- sysfsでデバイスがあり、 read/writeを使う



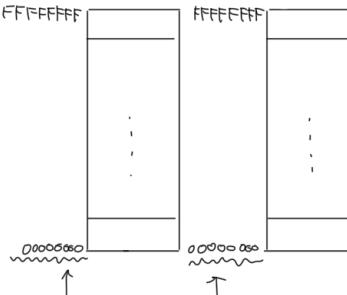
物理メモリ直接アクセスできない。

物理的な主記憶装置に加え補助記憶装置と併用すれば物理的な主記憶装置よりも大きな仮想メモリを提供可能また、スケーラ化も容易



仮想記憶 各アドレス空間に専用の主記憶装置があるようにOSが提供する

### アドレス空間（独立）

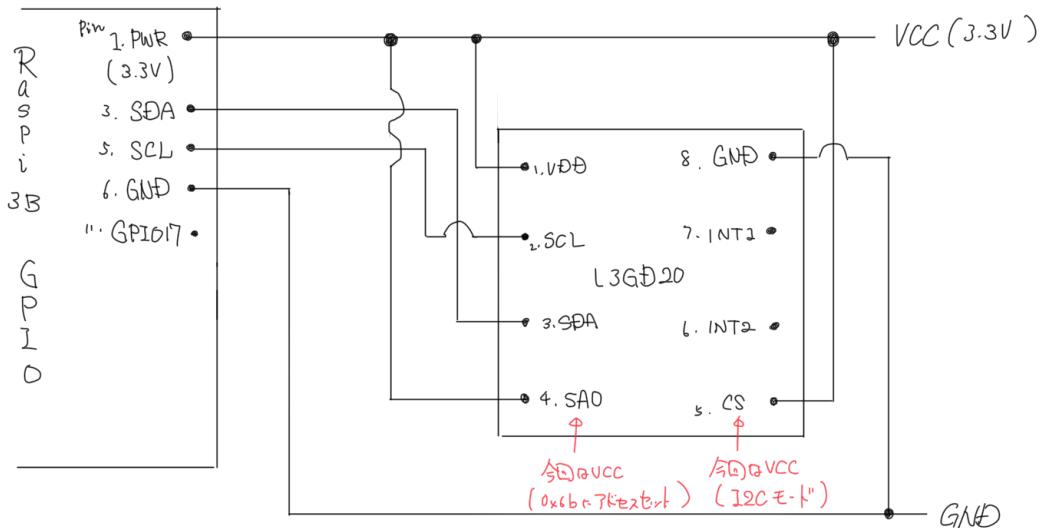


それらに対し物理メモリがどのように見えた

- メモリアドレス（仮想アドレス、論理アドレス）
- コンピュータのメモリ内の物理的位置を識別
  - 同じでも異なるアドレス空間内のアドレスならば、異なる位置を指す

I2C : Raspi 3B + L3GD20 (ジヤイロセンサ - 動き検出)

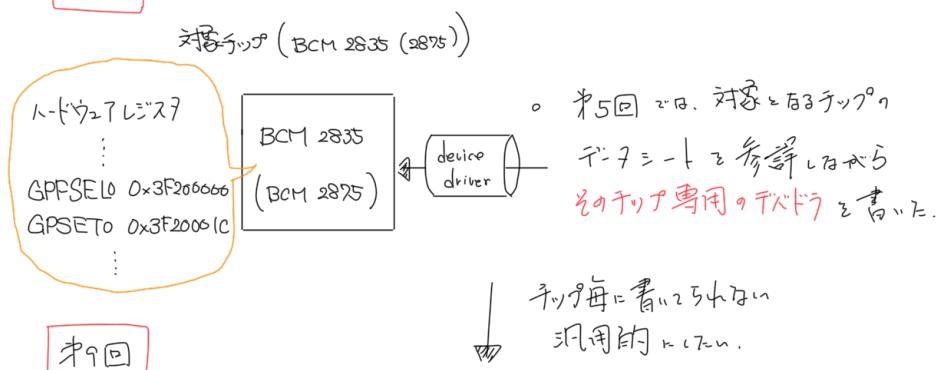
接続図



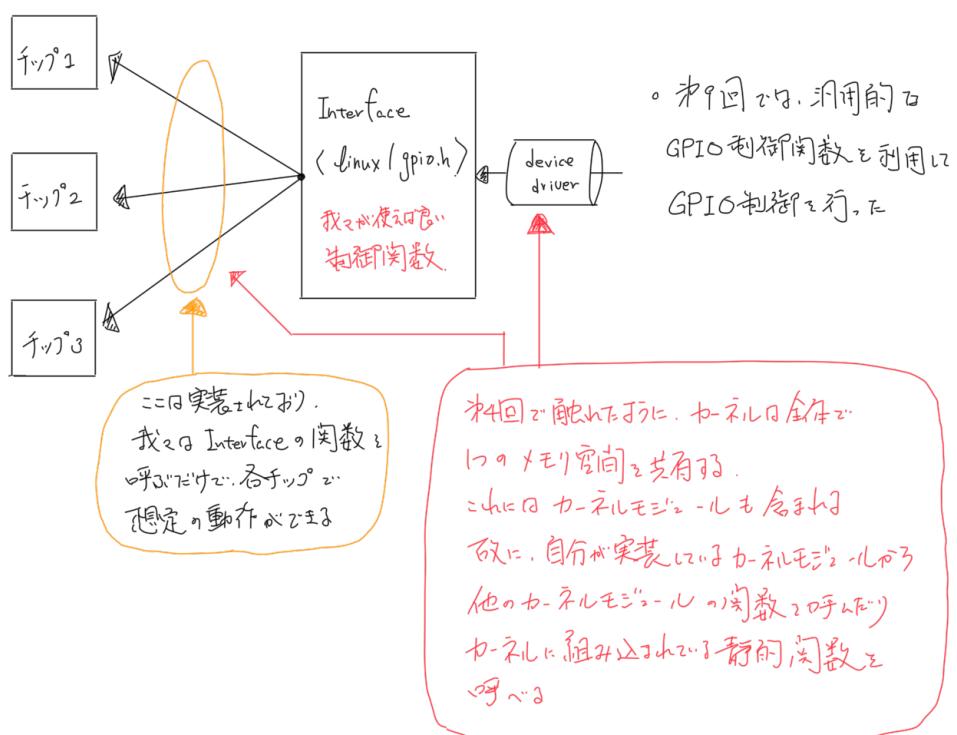
復習：これまでの GPIO 制御

★ GPIO の時は対象のラズパイのデバッガを書いて（その後、汎用関数を書き直す）

第5回



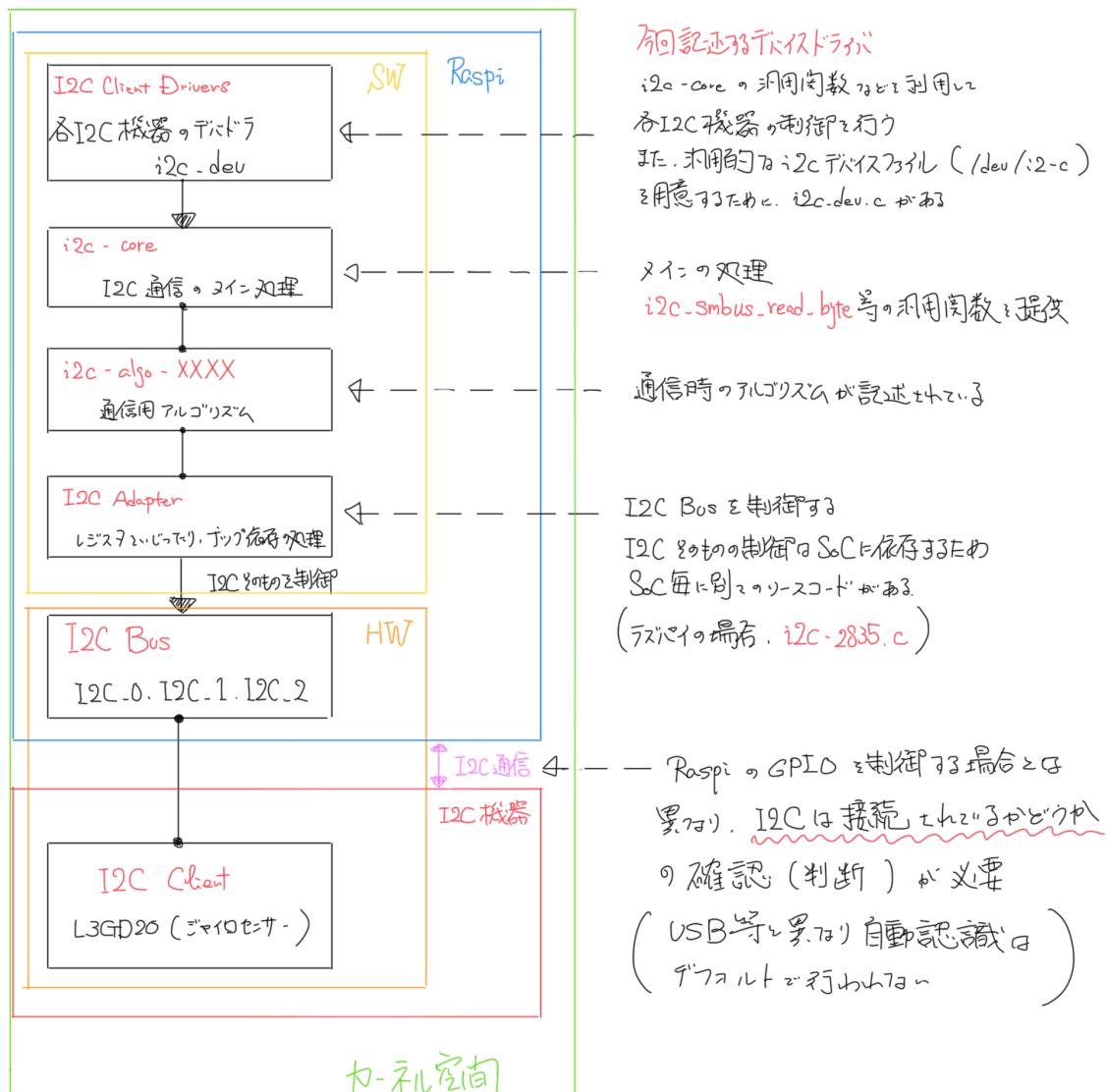
第9回



## I2C 周りの仕組み

GPIO の時と同様に汎用関数のコール 1つで簡単制御が可能。

I2C は少し複雑である。（I2C 自体のみなら I2C を構成するデバイスのデータを扱う）



## ① I2C の デバイドライバを作成

i2c\_driver 構造体

```c

```
struct i2c_driver {
    ...  

    int (*probe)(struct i2c_client *, const struct i2c_driver_id *);  

    int (*remove)(struct i2c_client *);  

    const struct i2c_device_id *id_table;  

    struct device_driver driver;  

    ...  

};
```

☞ この構造体を作成.

insmod 時 → 登録 : i2c\_add\_driver ( i2c\_driver の構造体のアドレス )

rmmod 時 → 削除 : i2c\_del\_driver ( i2c\_driver の構造体のアドレス )

カーネルモジュールの作成 (make)

・ カーネルモジュール読み込み後，“手動” I2C の接続 (認識)

を行うと，接続時ハンドラがコールされ 出力 = dmesg で確認できる。

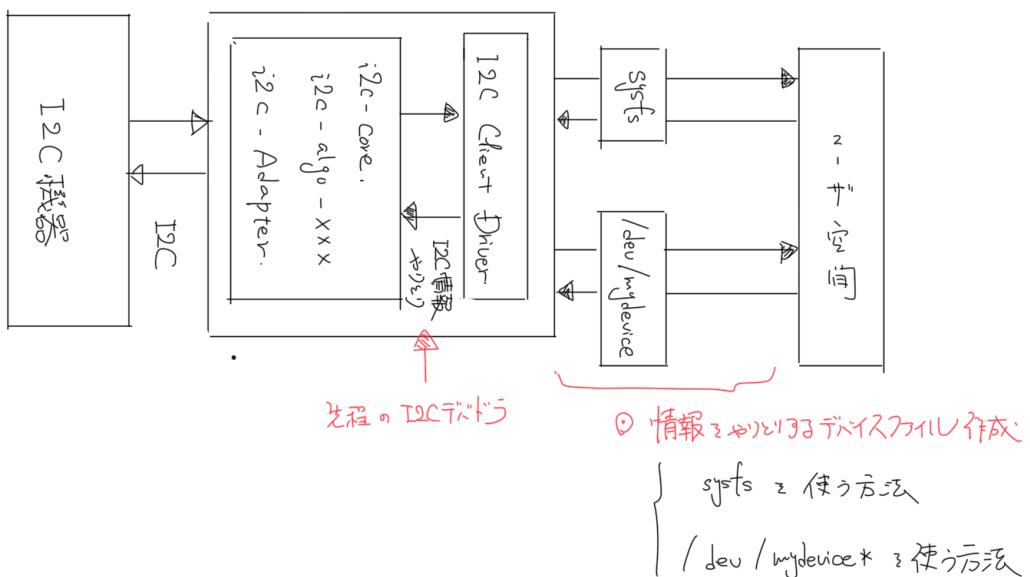
“手動” I2C の接続解除すると，取出し時ハンドラがコールされ。  
出力 = dmesg で確認できる。

\* 以上の行為は全 “カーネル空間” で行われる。

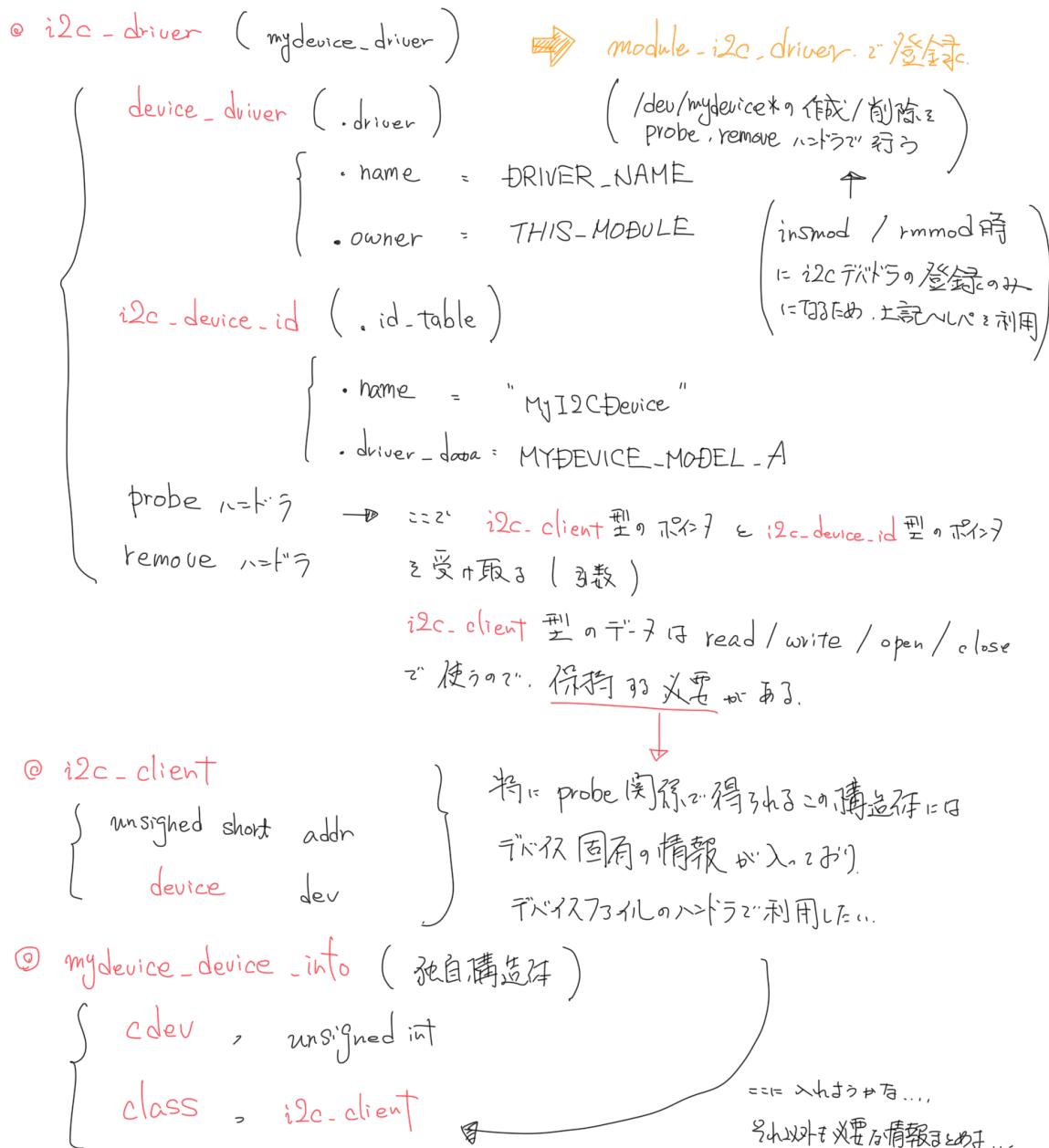
② I<sup>2</sup>C のデータを得た情報は - 手間に認識する。

今日は先に `dmesg` を確認して I<sup>2</sup>C 機器情報を - 手間に見る。

└ GPIO の時行方、データストリームの作成を行なう。



## 本コードの構造体概観 (構造体説明)



- ① probe ハンドラの引数に得られる (i2c-client) \* client は。  
 デバイス固有の情報を保持するため、カーネル-2-サブ間で /dev/mydevice0  
 デバイスファイルのハンドラ内で渡される ... しかし、専用処理ハンドラで 7 時間かかる。
- ↓ 独自の構造体を作成するのではなく  
諸々データを比較して、メモリに保持して、
- ② probe ハンドラ内で、カーネル空間のメモリ領域を確保して、値を入れる。  
i2c\_set\_clientdata で デバイス固有のプライベートデータ領域 (?) を保存。
- 正直、この関数のやるべき事は良くないね。 ...  
 となると、領域確保 → 保存という流れがデータをストアするが重要。
- ③ デバイスコントローラとして成る。システムコントローラハンドラブルの構成。  
 二つ、Open / Close 関数と、引数の inode 通り。この cdev を  
 参照しているため、Container\_of(cdev) で受け取ったメモリの構造体。  
 (mydevice\_device\_info)  
 先頭アドレス。つまり、probe時の dev\_info の先頭アドレスを得る。
- file → private\_data = &dev\_info, read / write  
 ハンドラ内で利用する
- dev\_info → client で probe時に得た i2c-client の指
  - アリデバイス固有の値が代入
- ④ cdev が I2C の probe / remove 時に作成 / 削除される。  
 insmod, rmmod の処理は I2C ハンドラの登録・削除 → module\_i2c\_driver.