TOPPERS基礎実装セミナー

（Raspberry Pi Pico(W)/Pico2(W)版：基本）

基礎編：2日目

TOPPERSプロジェクト

教育ワーキング・グループ

TOPPERS

本教材の利用条件

NEXCESS基礎コース01
組込みソフトウェア開発技術の基礎

Copyright (C) 2006-2007 by 名古屋大学 組込みソフトウェア技術者人材養成プログラム
Copyright (C) 2006-2007 by本田晋也

上記著作権者は，以下の(1)～(4)の条件を満たす場合に限り，本コンテンツ（本コンテンツを改変・翻訳したものを含む．以下同じ）を使用・複製・改変・翻訳・再配布（以下，利用と呼ぶ）することを無償で許諾する．

(1) この枠内の著作権表記等が，そのままの形でコンテンツ中に含まれていること．
(2) 本コンテンツを再配布する場合には，再配布の形態等を，以下のウェブサイトから報告すること．
http://www.nces.is.nagoya-u.ac.jp/NEXCESS/REPORT/
(3) 本コンテンツを改変・翻訳する場合には，コンテンツを改変・翻訳した旨の記述を，コンテンツ中に含めること．また，改変・翻訳者の著作権表記等は，この枠内の著作権表記等とは別に行うこと．
(4) 本コンテンツの利用により直接的または間接的に生じるいかなる損害からも，上記著作権者を免責すること．

※ 本コンテンツの一部は，文部科学省 科学技術振興調整費により，名古屋大学 組込みソフトウェア技術者人材養成プログラム（NEXCESS）の一環として作成しました．
※ 本コンテンツ中に記載されている商品名やサービス名などは，各社の商標または登録商標です．

本ドキュメントに関して

1. 著作権に関しての表記

<TOPPERS基礎実装セミナー(Raspberry Pi Pico(W)/Pico2(W)版:基本)2日目>

Copyright (C) 2006-2007 by 名古屋大学 組込みソフトウェア技術者人材養成プログラム
Copyright (C) 2006-2007 by 本田晋也　名古屋大学
Copyright (C) 2007-2025 by 竹内良輔 教育WG

上記著作権者は、以下の(1)～(3)の条件を満たす場合に限り、本ドキュメント(本ドキュメントを改変したものを含む。以下同じ)を使用・複製・改変・再配布(以下、利用と呼ぶ)することを無償で許諾する。
(1) 本ドキュメントを利用する場合には、上記の著作権表示、この利用条件および以下の無保証規定が、そのままの形でドキュメント中に含まれていること。
(2) 本ドキュメントを改変する場合には、ドキュメントを改変した旨の記述を、改変後のドキュメント中に含めること。ただし、改変後のドキュメントがTOPPERSプロジェクト指定の開発成果物である場合には、この限りではない。
(3) 本ドキュメントの利用により直接的または間接的に生じるいかなる損害からも、上記著作権者およびTOPPERSプロジェクトを免責すること。また、本ドキュメントのユーザまたはエンドユーザからのいかなる理由に基づく請求からも、上記著作者およびTOPPERSプロジェクトを免責すること。
本ドキュメントは、無保証で提供されているものである。上記著作権者およびTOPPERSプロジェクトは、本ドキュメントに関して、特定の使用目的に対する適合性も含めて、いかなる保障もしない。また、本ドキュメントの利用により直接的または間接的に生じたいかなる損害に関しても、その責任を負わない。

---

2. 本ドキュメントに関するご意見・ご提言・ご感想・ご質問等がありましたら、TOPPERSプロジェクト事務局までE-Mailにてご連絡ください。

3. 本ドキュメントの内容は、内容の改善や適正化の目的で予告無く改定することがあります。

本ドキュメントでは、Microsoft社のClip Art Galleryコンテンツを使用しています。

TRONは”The Real-time Operating system Nucleus”の略称です。ITRONは”Industrial TRON”の略称です。μITRONは”Micro Industrial TRON”の略称です。TOPPERS/JSPはToyohashi Open Platform for Embedded Real-Time System/Just Standard Profile Kernelの略称です。」

本ドキュメント中の商品名及び商標名は、各社の商標または登録商標です。

TOPPERS

スケジュール

- ２日目

1.	メモリマップドレジスタの操作方法の確認	0.5時間
2.	ポーリングプログラム	3.0時間
3.	割込みプログラム	2.0時間
4.	まとめ	0.5時間

概要

- マイコンボードを用いて
  組込みプログラムミングの基礎を学ぶ

- メモリマップドレジスタの操作方法の確認
  - デバッガを用いてデバイスレジスタを操作する
- ポーリングプログラミング
  - LED，スイッチ，タイマ，シリアルI/Oを操作する
    プログラムの作成
- 割込みプログラミング
  - スイッチ，タイマ，シリアルI/Oからの事象を割込みに
    より扱うプログラムの作成
- カップラーメンタイマの作成
  - スイッチ，タイマによりカップラーメンタイマの作成例

メモリマップドレジスタの操作方法の確認

1. ROMモニタの操作
2. LEDの接続
3. LEDの操作

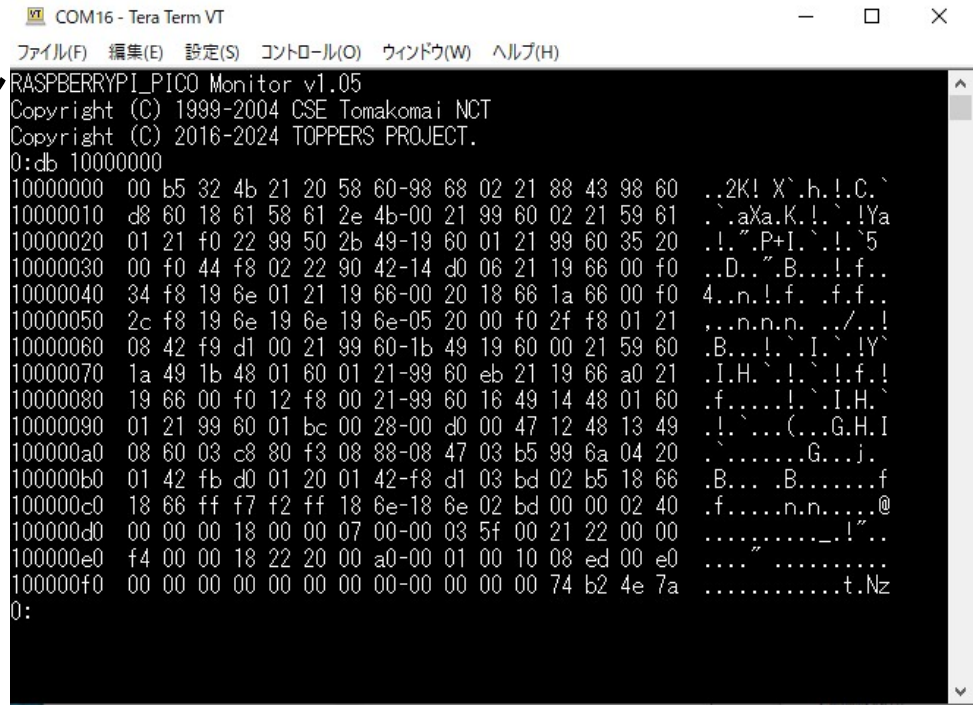
TOPPERS

メモリマップドレジスタの操作方法の確認：目的

LEDが接続されているプログラマブル入出力ポートを例に
メモリマップドレジスタの操作方法を学ぶ

- ROMモニタを用いると任意のアドレスのメモリの読み書きが可能
- CPUの周辺機能の制御レジスタはメモリ空間に配置されている（メモリマップドレジスタ）
  - ROMモニタからLEDが接続されているポートを操作可能

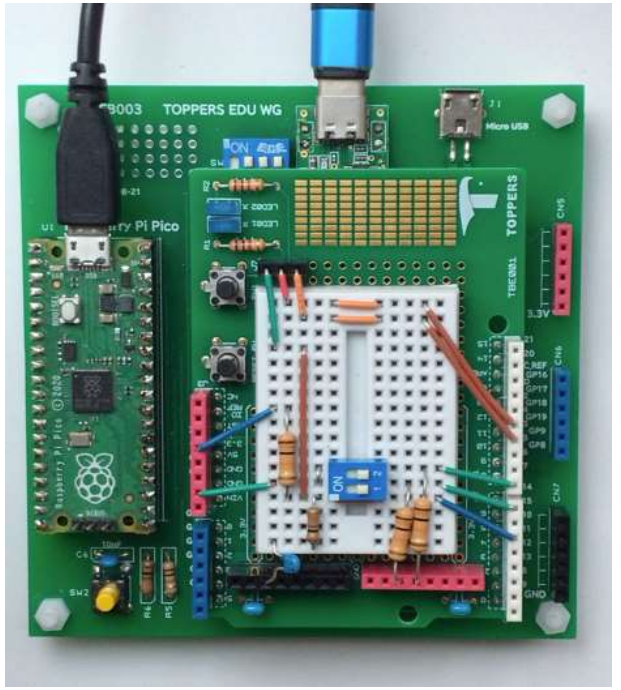
- ROMモニタの E(L)コマンドによるメモリの読み書き



TOPPERS

ROMモニタの操作方法

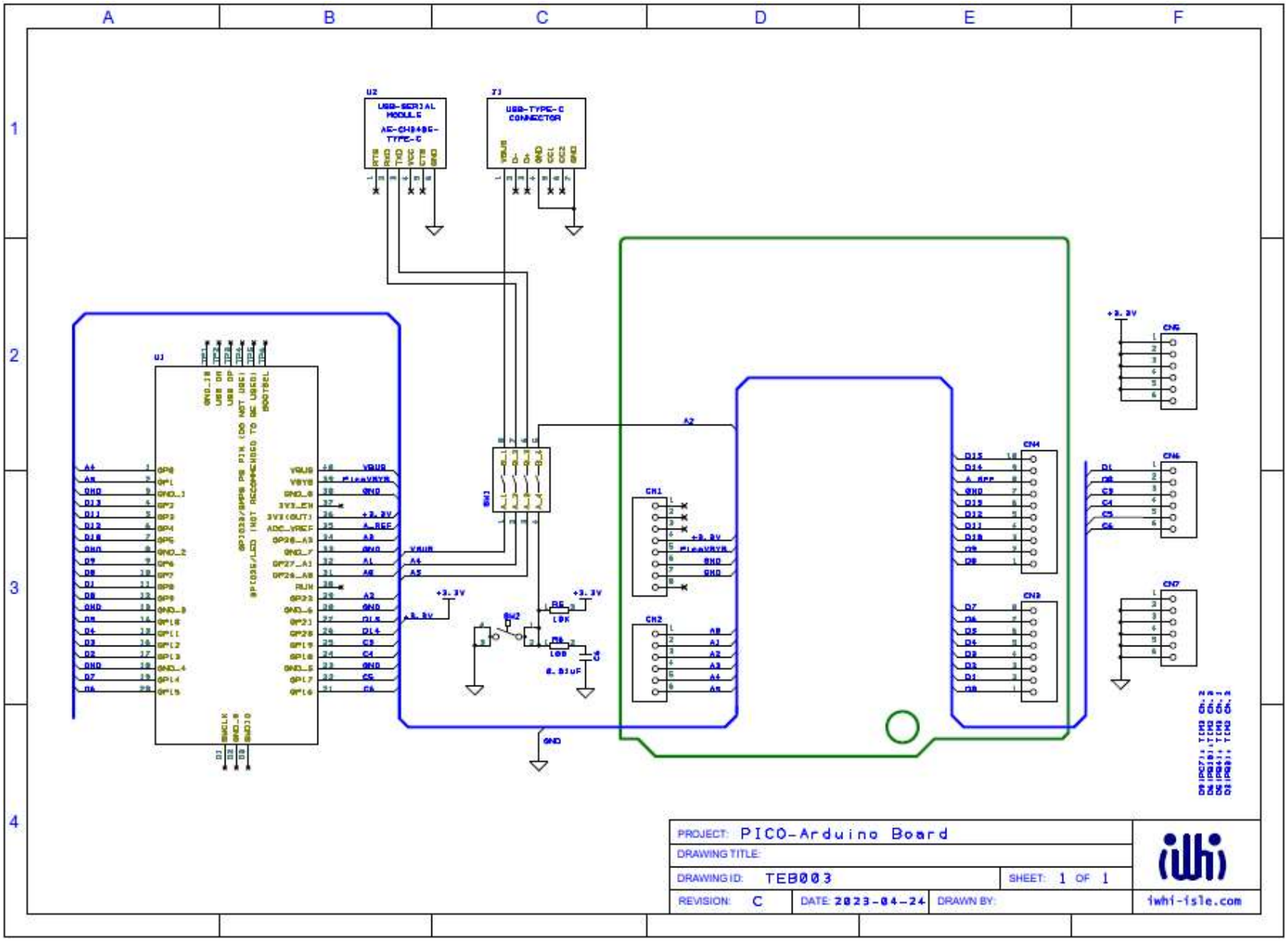
- TEB003ボードのUSBシリアルをパソコンに接続
- デバイスマネージャでCOMポートを確認
  - TeraTermを立ち上げ
  - TeraTermを設定



- Pico/Pico2の USBをケーブルでパソコンまたは5V電源に接続

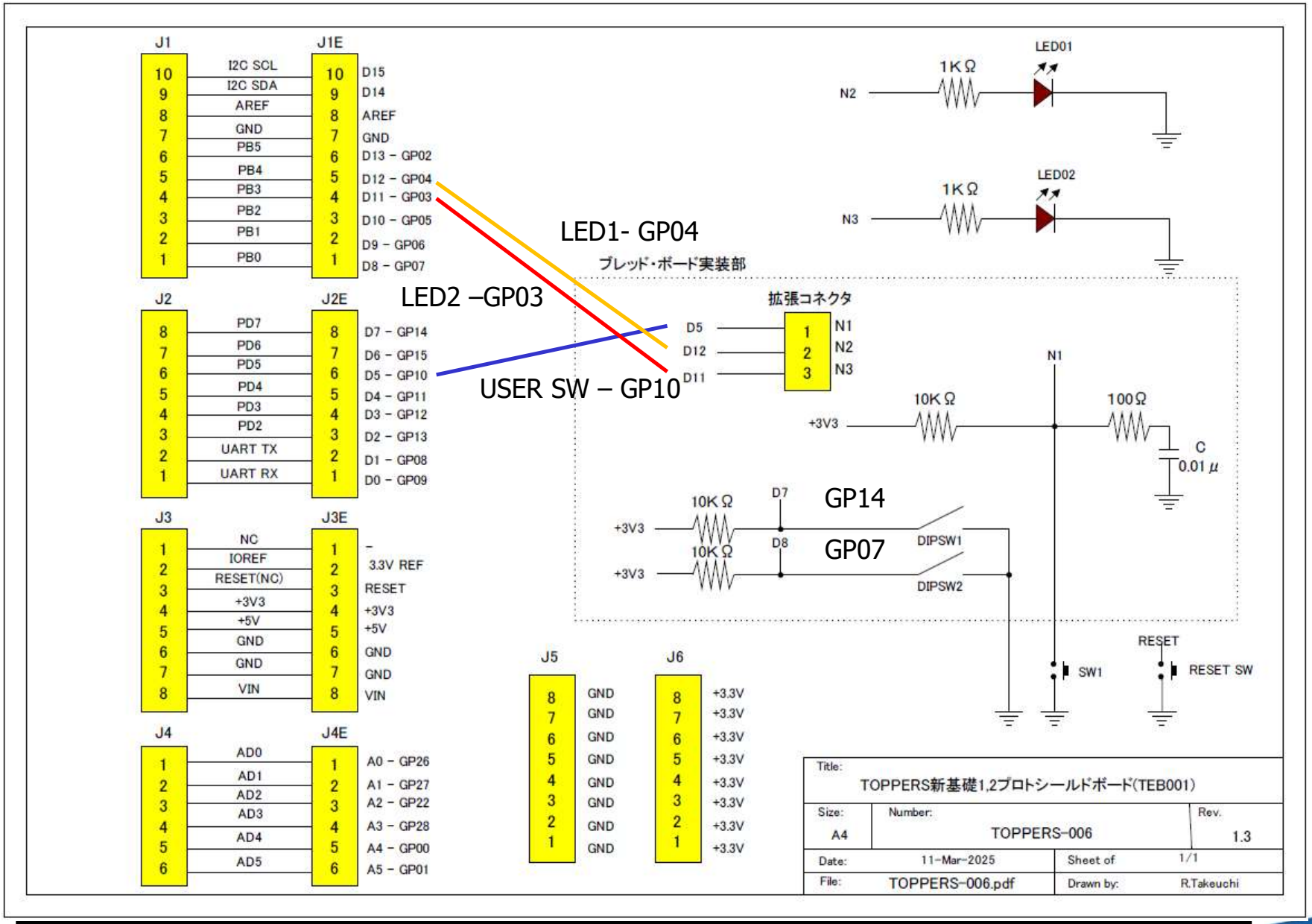
- コマンドを使ってモニタを操作する
  - ？　Enterで、使用可能なコマンドの一覧を表示する

TEB003回路図



TOPPERS

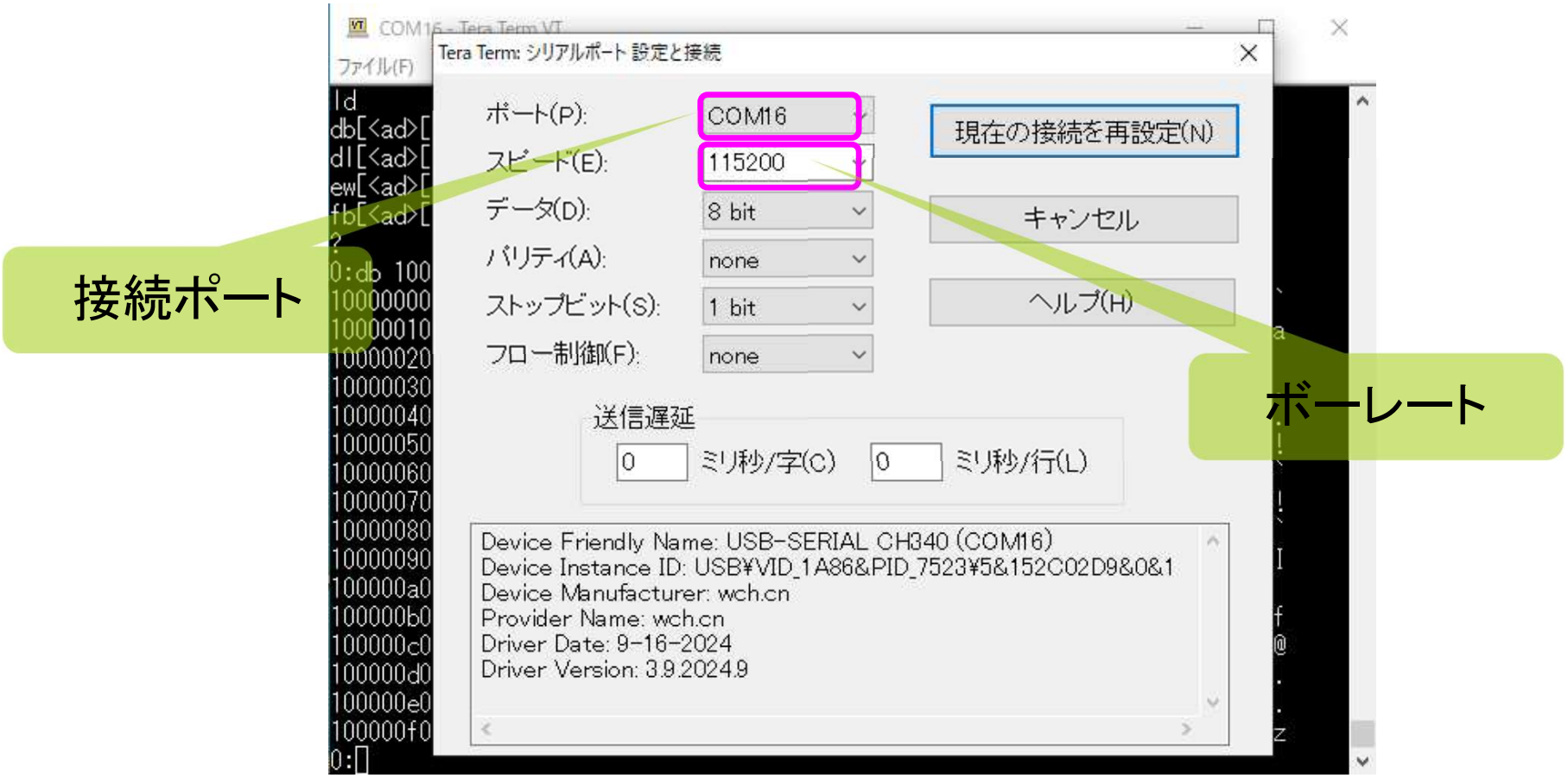
プロトタイピング・シールド回路図



TOPPERS

ROMモニタの操作方法：TeraTermシリアルポート

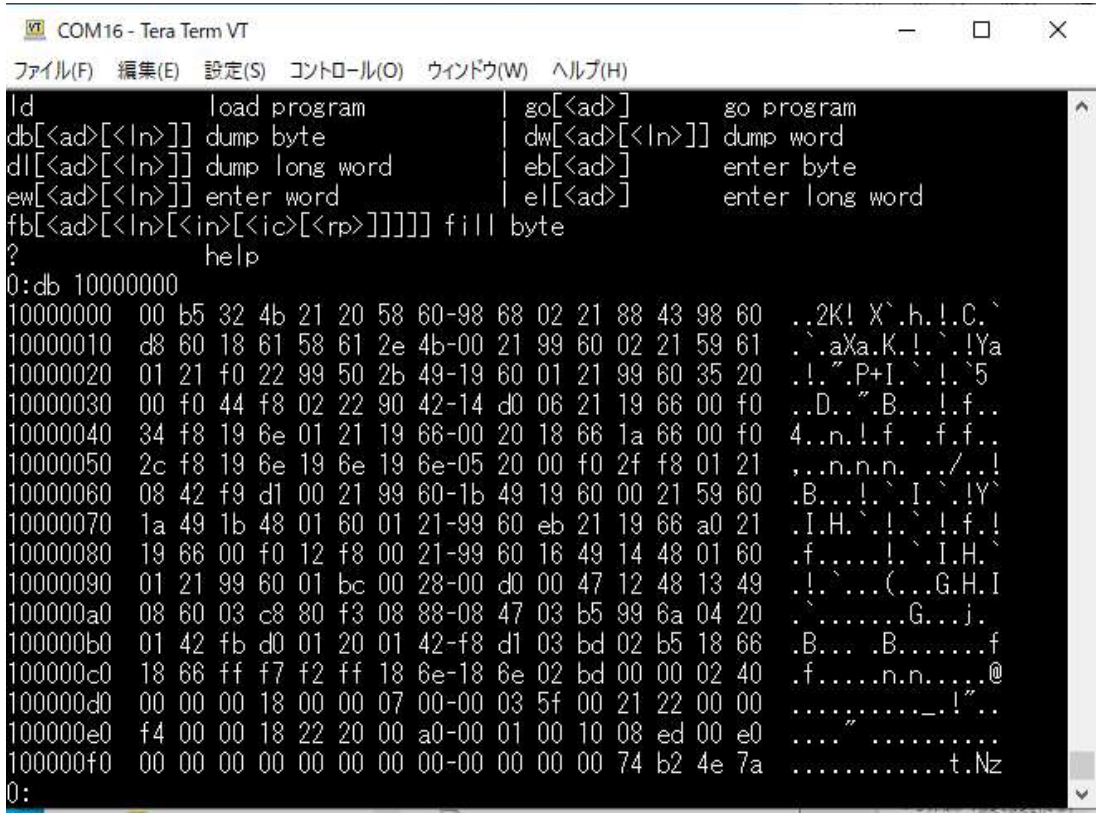
- ROMモニタはUARTを使用して、ボードに対して種々の設定や確認を行うことができます
- シリアルポート設定で設定を行います



デバッガの操作方法：読み出し

- dl コマンドで4バイト単位のメモリDUMPができます
- db コマンドは1バイト単位、dw コマンドは2バイト単位

dl <address> enter

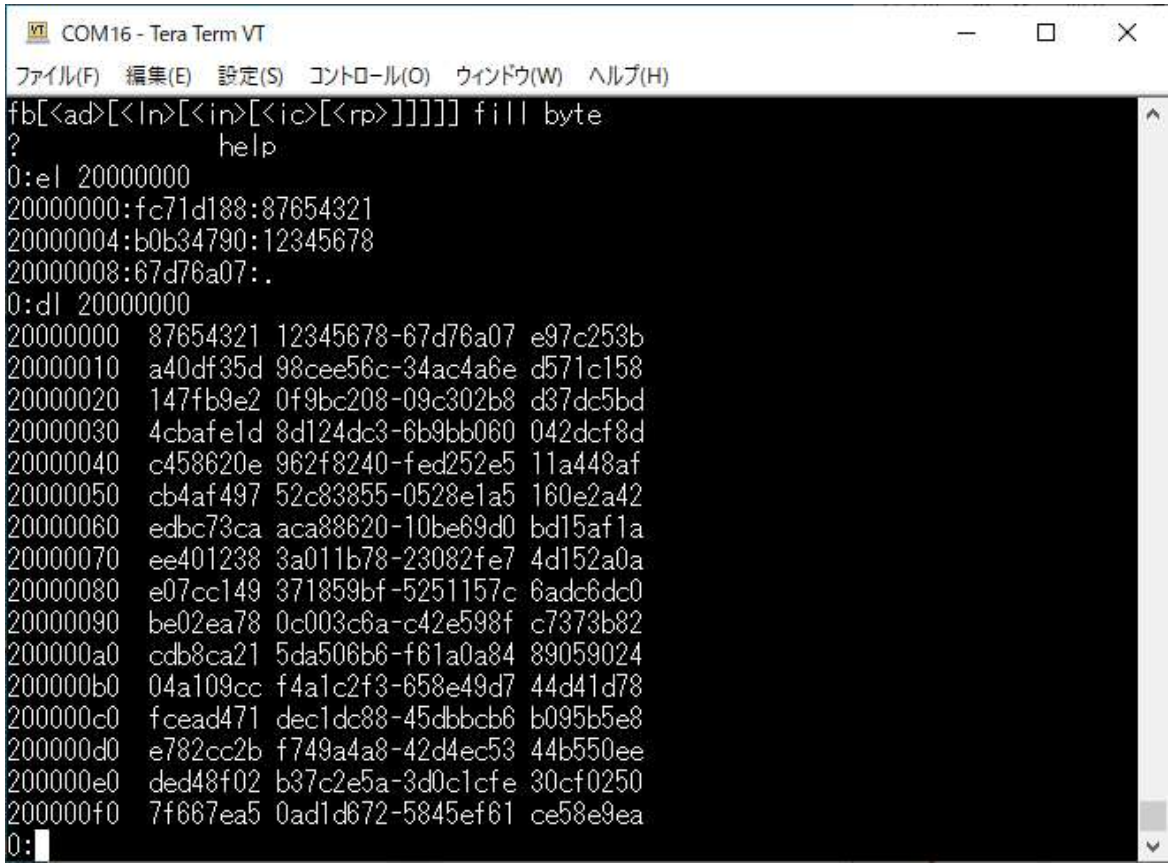


TOPPERS

デバッガの操作方法：書き込み

- el コマンドにて4バイト単位でのデータ書き込みができます
- el アドレスで、データ入力モードに移行し、データを入力すると設定アドレスにデータを書き込みます
- '.' enterにて、データ入力モードを終了します

RAM領域の
0x20000000番地と
0x20000004番地に
0x87654321と
0x12345678を書き込む



TOPPERS

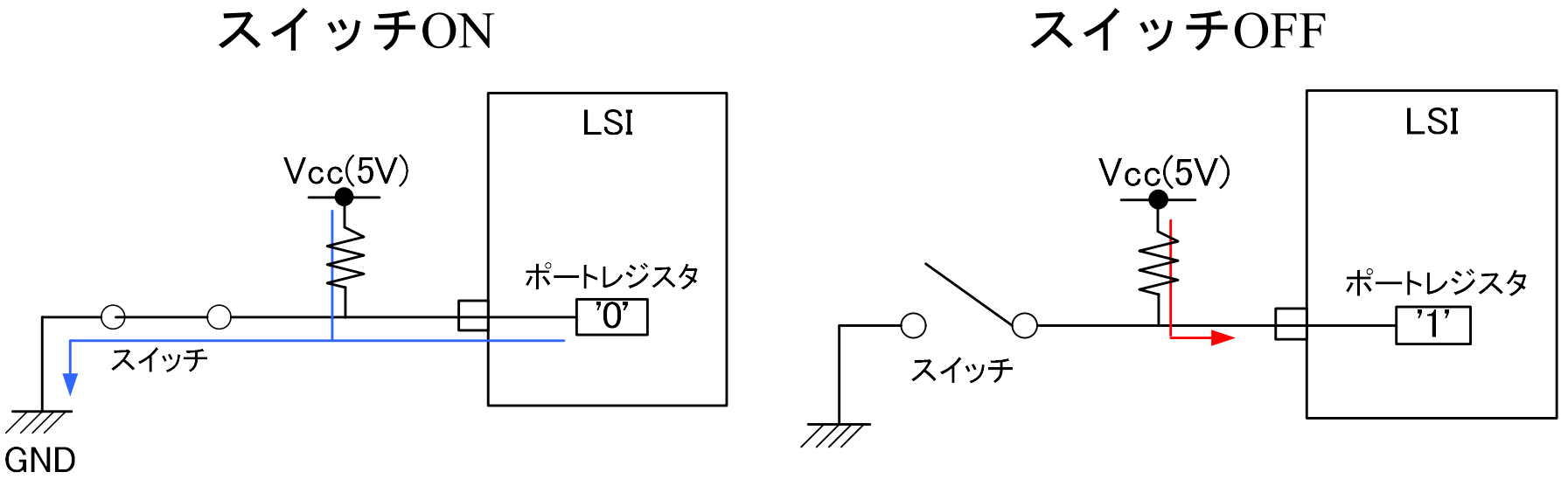
周辺デバイス：プルアップとプルダウン

- デジタル回路は電圧の高低（HiとLo）で’0’と’1’を判断する
  例）電源電圧が5vの場合，Hiが4～5v, Loは0～1v
- LSIへの入力はHiかLoの電圧になるようにしなければならない

プルアップとプルダウン

- GPIO入力ポートとしてスイッチ等を接続する場合，入力を電気的に安定させるための回路構成

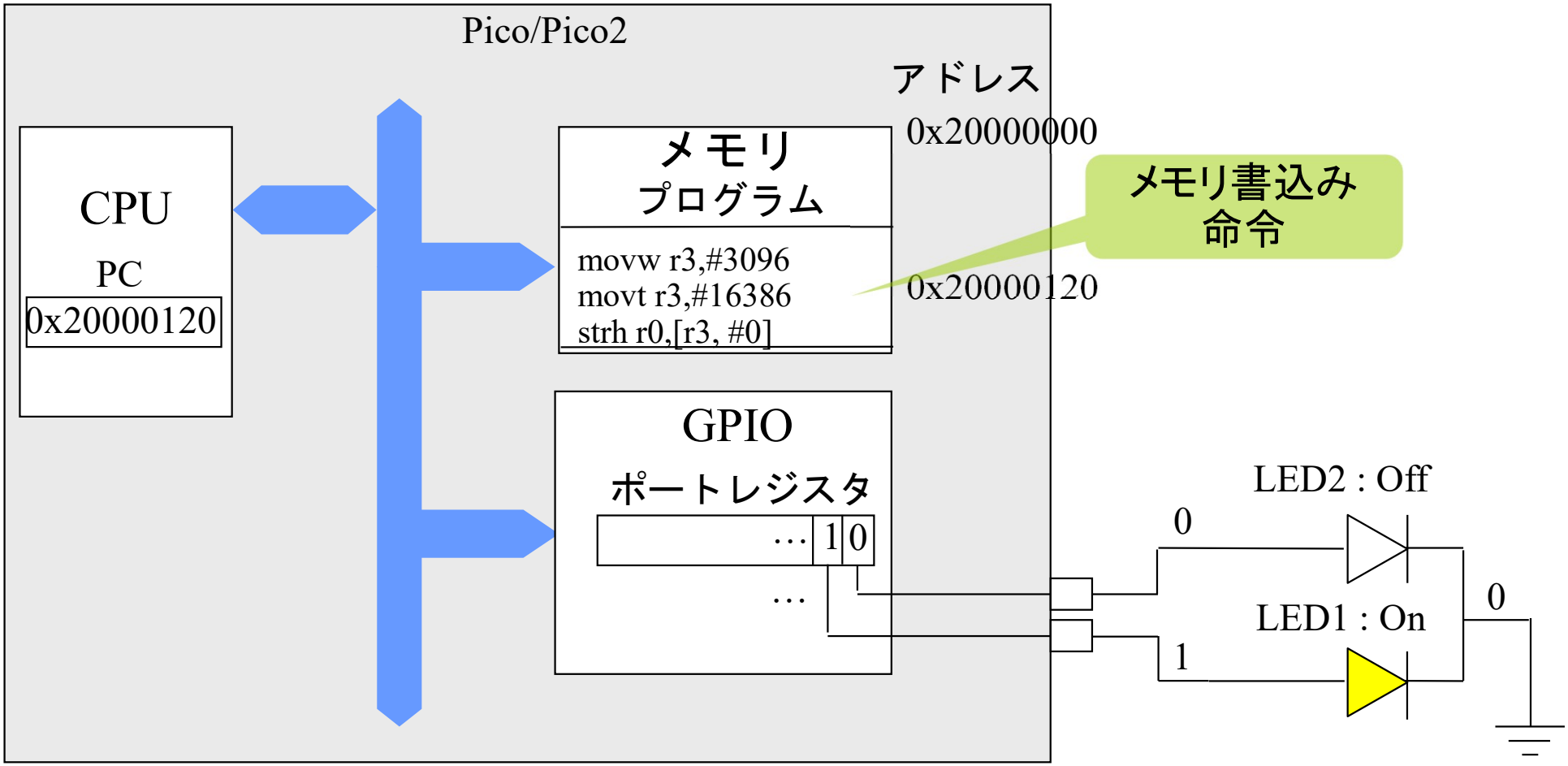
プルアップの例



周辺デバイス：プロセッサとLEDとの接続の関係

プロトタイピング・シールドの例

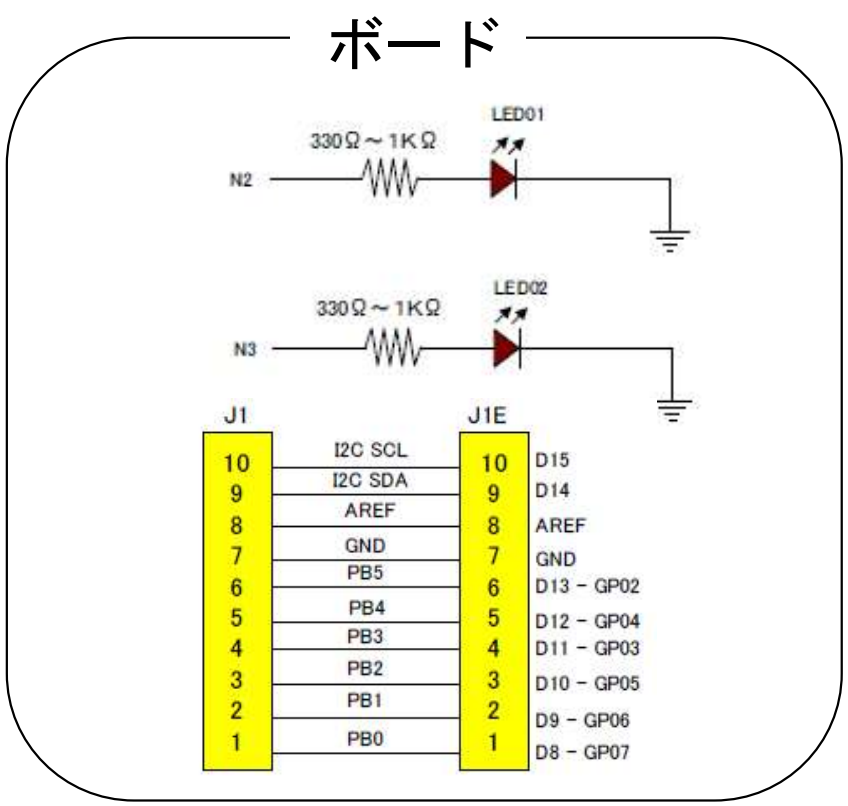
–ポートレジスタの値が1でLEDが点灯する

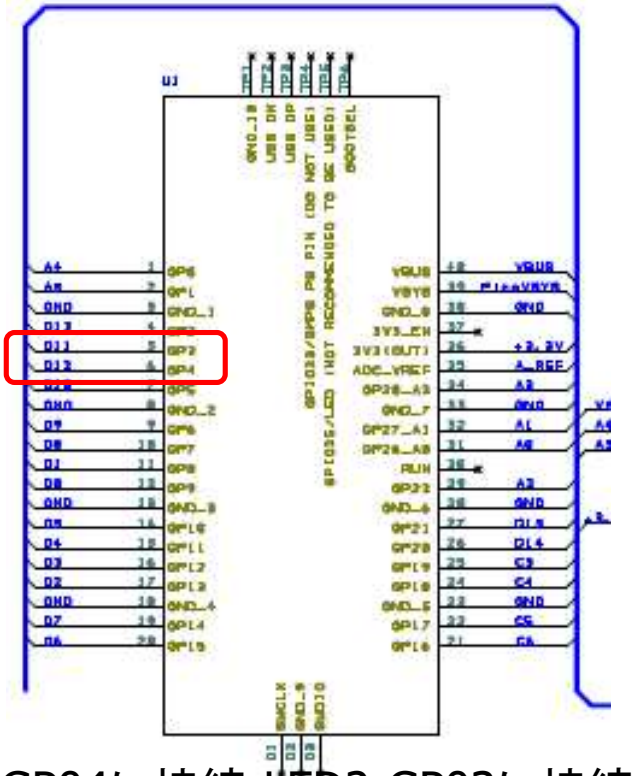


TOPPERS

LEDの接続：マニュアルによる確認

- LEDはプルアップされている
- 直接、マイコンのプログラマブル入出力ポートに接続されている
  - マイコンから”1”を出力すると点灯する





LED1-GP04に接続 LED2-GP03に接続

TOPPERS

LEDの接続：レジスタ構成

- LEDのピン設定
  - PDS_BANK0 GPIO
    - PDE 1 / PUE 0 / IE 1 / OD 0
  - Picoは初期値（0x00000056）のままなので、変更なし
- PicoとPico2ではマッピングアドレスが違う

Pioc	アドレス
PDS_BANK0 GP3	0x4001c010
PDS_BANK0 GP4	0x4001c014

Pioc2	アドレス
PDS_BANK0 GP3	0x40038010
PDS_BANK0 GP4	0x40038014

PADS_BANK0: GPIO0, GPIO1, ..., GPIO28, GPIO29 Registers

Offsets: 0x04, 0x08, ..., 0x74, 0x78

Description

Pad control register

Bits	Name	Description	Type	Reset
31:8	Reserved.	-	-	-
7	OD	Output disable. Has priority over output enable from peripherals	RW	0x0
6	IE	Input enable	RW	0x1
5:4	DRIVE	Drive strength. 0x0 → 2mA 0x1 → 4mA 0x2 → 8mA 0x3 → 12mA	RW	0x1
3	PUE	Pull up enable	RW	0x0
2	PDE	Pull down enable	RW	0x1
1	SCHMITT	Enable schmitt trigger	RW	0x1
0	SLEWFAST	Slew rate control. 1 = Fast, 0 = Slow	RW	0x0

TOPPERS

LEDの接続：レジスタ構成

- LEDのピン設定
  - IO BANK0のGPIO_CTRL
    - FUNCSELをGPIOに指定、GPIOはファンクション番号５
    - 設定値：0x00000005
- PicoとPico2ではマッピングアドレスが違う

Pioc	アドレス
IO_BANK0 CTRL GP3	0x4001401c
IO_BANK0 CTRL GP4	0x40014024

Pioc2	アドレス
IO_BANK0 CTRL GP3	0x4002801c
IO_BANK0 CTRL GP4	0x40028024

IO_BANK0: GPIO0_CTRL, GPIO1_CTRL, ..., GPIO28_CTRL, GPIO29_CTRL Registers

Offsets: 0x004, 0x00c, ..., 0x0e4, 0x0ec

Description

GPIO control including function select and overrides.

Table 296. GPIO0_CTRL, GPIO1_CTRL, ..., GPIO28_CTRL, GPIO29_CTRL Registers

Bits	Name	Description	Type	Reset
31:30	Reserved.	-	-	-
29:28	IRQOVER	0x0 → don't invert the interrupt 0x1 → invert the interrupt 0x2 → drive interrupt low 0x3 → drive interrupt high	RW	0x0
27:18	Reserved.	-	-	-
17:16	INOVER	0x0 → don't invert the peri input 0x1 → invert the peri input 0x2 → drive peri input low 0x3 → drive peri input high	RW	0x0
15:14	Reserved.	-	-	-
13:12	OEOVER	0x0 → drive output enable from peripheral signal selected by funcsel 0x1 → drive output enable from inverse of peripheral signal selected by funcsel 0x2 → disable output 0x3 → enable output	RW	0x0
11:10	Reserved.	-	-	-
9:8	OUTOVER	0x0 → drive output from peripheral signal selected by funcsel 0x1 → drive output from inverse of peripheral signal selected by funcsel 0x2 → drive output low 0x3 → drive output high	RW	0x0
7:5	Reserved.	-	-	-
4:0	FUNCSEL	Function select. 31 == NULL. See GPIO function table for available functions.	RW	0x1f

TOPPERS

LEDの接続:レジスタ構成

- GPIOの入出力設定
  - SIO GPIO_OE 対応のピンビット　0:input 1:output
  - GP3は (1<<3）で0x00000008
  - GP4は（1<<4）で0x00000010
- ビット操作となるので注意
- PicoとPico2ではマッピングアドレスが違う

Pioc	アドレス
SIO_GPIO_OE	0xd0000020

Pioc2	アドレス
SIO_GPIO_OE	0xd0000030

SIO: GPIO_OE Register

Offset: 0x020

Description

GPIO output enable

Table 25. GPIO_OE Register

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:0	Set output enable (1/0 → output/input) for GPIO0...29. Reading back gives the last value written. If core 0 and core 1 both write to GPIO_OE simultaneously (or to a SET/CLR/XOR alias), the result is as though the write from core 0 took place first, and the write from core 1 was then applied to that intermediate result.	RW	0x00000000

TOPPERS

LEDの接続:レジスタ構成

- GPIOの端子出力設定
  - SIO　GPIO_OUT　対応のピンビット　0:LOW,1:HIGH
  - 1を設定するには、SIO GPIO_OUT_SETの対応ビットに1をセット
  - 0を設定するには、SIO GPIO_OUT_CLRの対応ビットに1をセット
  - GP3は (1<<3）で0x00000008
  - GP4は（1<<4）で0x00000010
- PicoとPico2ではマッピングアドレスが違う

Pioc	アドレス
GPIO_OUT_SET	0xd0000014
GPIO_OUT_CLR	0xd0000018

Pioc2	アドレス
GPIO_OUT_SET	0xd0000018
GPIO_OUT_CLR	0xd0000024

Table 22. GPIO_OUT_SET Register

SIO: GPIO_OUT_SET Register

Offset: 0x014

Description

GPIO output value set

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:0	Perform an atomic bit-set on GPIO_OUT, i.e. `GPIO_OUT |= wdata`	RW	0x00000000

Table 23. GPIO_OUT_CLR Register

SIO: GPIO_OUT_CLR Register

Offset: 0x018

Description

GPIO output value clear

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:0	Perform an atomic bit-clear on GPIO_OUT, i.e. `GPIO_OUT &= ~wdata`	RW	0x00000000

TOPPERS

LEDの接続：設定値

レジスタを操作して全てのLEDを点灯させる

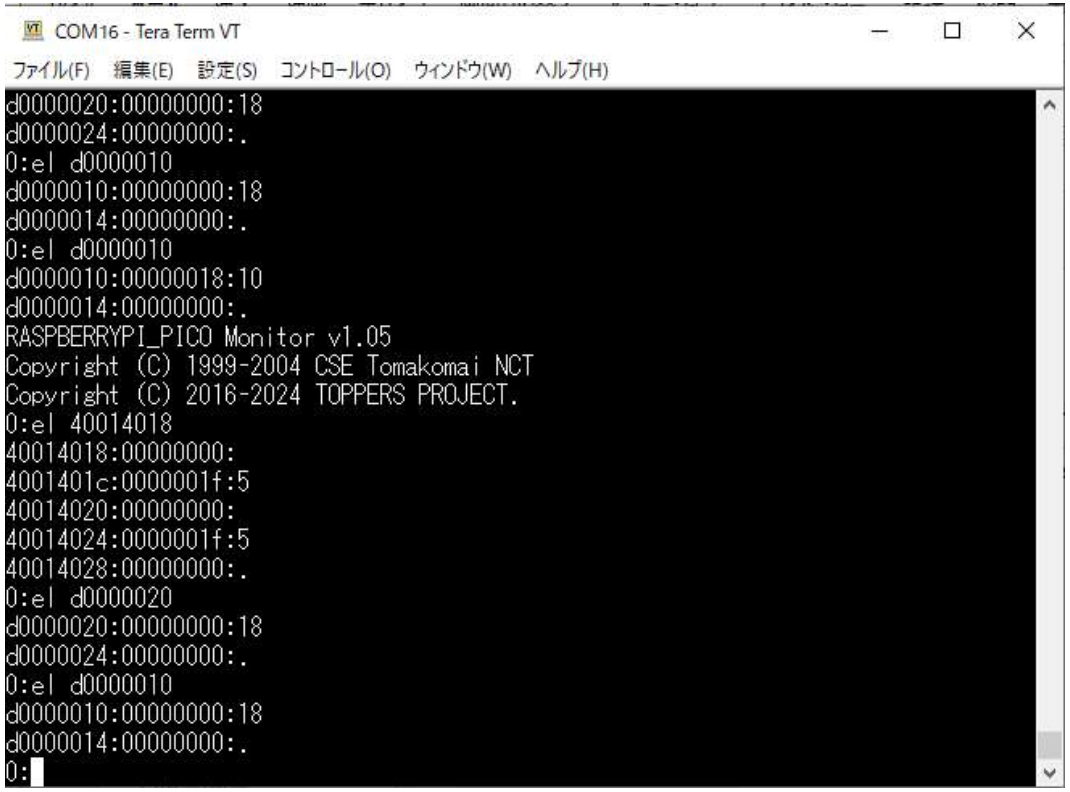
- ポートを構成するレジスタとLEDとの接続を整理すると全てのLEDを点灯させるには以下の設定が必要
  - PDS_BANK0 GP3とGP4にピン属性を設定
  - IO_BANK0 CTRL GP3のFUNCSELに0x5を設定
  - IO_BANK0 CTRL GP4のFUNCSELに0x5を設定
  - SIO GPIO_OEのピン3とピン4ビットに1を設定
- Picoの場合、レジスタへ書き込む値（?は変更させない）
  - 0x401c0010: 0x00000056(元の値：0x00000056)→変更不要
  - 0x401c0014: 0x00000056(元の値：0x00000056)→変更不要
  - 0x4014001c : 0x??????05（“???? ???? ???? ???? ???0 0101"）
  - 0x40140024 : 0x??????05(“???? ???? ???? ???? ???0 0101”)
  - 0xd0000020 : 0x??????18（“???????? ???? ???? ???1 1???"）

TOPPERS

LEDの操作：任意のLEDの点灯

- PicoのROMモニタのコマンド設定でLEDの初期化を行った
- 最後の0xd0000010の設定はSIO GPIO_OUTに直接0x18を書き込みLEDを点灯させた

Pico2の場合はアドレスが違うので注意



TOPPERS

LEDの操作：任意のLEDの点灯

- 次のパターンでLEDを点灯させるためのポートレジスタへ書き込む値を考え，実際に確認せよ

1. LED2:OFF LED1:OFF

2. LED2:ON LED1:ON

3. LED2:OFF LED1:ON

4. LED2:ON LED1:OFF

LEDの操作 : 任意のLEDの点灯

1. LED2:OFF LED1:OFF
   - RESET(0xd0000018):0x0018 : “0000 0000 0001 1000"
2. LED2:ON LED1:ON
   - SET(0xd0000014):0x00018 : “0000 0000 0001 1000“
3. LED2:OFF LED1:ON
   - SET(0xd0000014):0x0010 : "0000 0000 0001 0000“
   - RESET(0xd0000018):0x0008 : “0000 0000 0000 1000”
4. LED2:ON LED1:OFF
   - SET(0xd0020014):0x0008 : “0000 0000 0000 1000“
   - RESET(0xd0000018):0x0010 : “0000 0000 001 0000”

ポーリングプログラム

1. LEDプログラム
   ・プロジェクトの作成方法
2. スイッチプログラム
3. タイマプログラム
4. シリアルI/Oプログラム

ポーリングプログラミング：目的

周辺デバイスを操作するプログラムの書き方を学ぶ

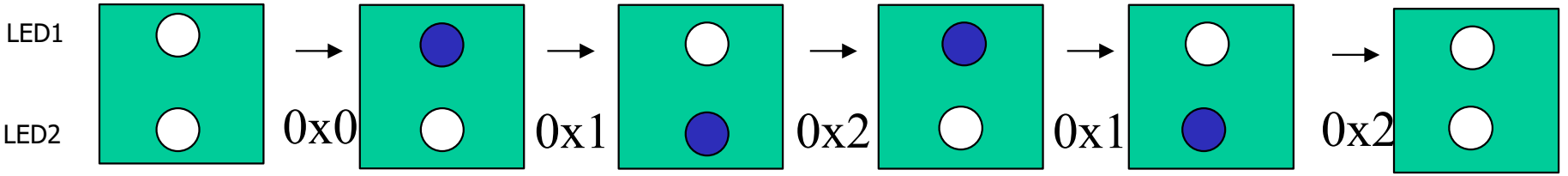
- 初期設定，データの入出力，事象の待ち方
  - スイッチのONやタイマのタイムアウトなどの外部事象はポーリングによって扱う
- プログラミング対象の周辺デバイス
  - LED（プロマブル入出力ポート出力）
  - スイッチ（プロマブル入出力ポート入力）
  - タイマ
  - シリアルI/O

ポーリングプログラミング：プログラムファイル

- プログラムファイルの置き場所
  - 教材ディレクトリ/base1/program
  - MSYS2のホームディレクトリにtoppersを作り、base1をtoppersの下にコピーしてください
- プログラム一覧
  - sample　　：表示確認サンプルプログラム
  - led_shift　　：LEDシフト点灯プログラム
  - int_switch_2：割込みPUSHスイッチプログラム
  - int_uart　　：割込みUARTプログラム
  - cup_timer　　：カップラーメンタイマプログラム

LEDプログラム : led_shift 概要

- 次のパターンでLEDを点灯させる
  - LED全てOFF → LED1 ON → LED2 ON → LED1 ON → LED2 ON → LED全てOFF



- 学習内容
  - プログラムの全体像の理解
  - ビット操作（セット，クリア）
  - デバイスレジスタ操作（様々な記法）
    ポートからの出力
  - スタートアップルーチン，セクション
- 構成ファイル
  - led_shift.c

led_shift : メイン関数

– 後述するスタートアップルーチンから呼び出される

LED接続ポートの初期化

ウェイト

処理内容は同一であるが記法が異なる

```
void
main(void){
    uint32_t count = 0;
    led_init();
    for (;;) {
        sys_ptintf("count=%d¥n", count);
        count++;
        led_out_no_macro(0x00);
        busy_wait();
        led_out_macro1(LED1);
        busy_wait();
        led_out_macro2(LED2);
        busy_wait();
        led_out_struct(LED1);
        busy_wait();
        led_out_sil(LED2);
        busy_wait();
    }
}
```

LED全消灯

LED1点灯

LED2点灯

LED1点灯

LED2点灯

TOPPERS

led_shift : ポートレジスタへの書き込み

- LED点灯・消灯制御
  - SIOのGPIO_OUT_SET/CLRレジスタ（0xd0000014,0xd0000018）への書き込み
  - 特定アドレスの読み書きが必要
- アセンブリコード

```
                ldr   r2,  [pc, #12]  /* 2000026c */
                str    r1, [r2, #0] /* 書込み */
                :
2000026c  :  d0000018
```

- C言語
  - ポインタを使用
  - アドレスをポインタ変数にキャスト

```
書き込み    : *((uint32_t *)(0xd0000014)) = 0x00000018;
読み込み    :  tmp = *((uint32_t *)(0xd0000010));
```

led_shift : volatile修飾子

- volatile
  - ポインタを用いてレジスタをアクセスする場合は，最適化を抑制するため，常にvolatile指定をつける必要ある

```
書き込み　: *((volatile uint32_t *)(0xd0000014)) = 0x00000018;
読み込み　:  tmp = *((volatile uint32_t *)(0xd0000010));
```

- volatile修飾子の場所
  - 以下の記述でもコンパイルは通るが，最適化は抑制できないので注意が必要である（ポインタ変数がvolatile）
  - ポインタ変数が指し示す先をvolatileにしなければならない

```
tmp = *((uint32_t * volatile )(0xd0000010));
```


TOPPERS

LEDの接続:レジスタ構成

- GPIOの出力イネーブル設定
  - SIO　GPIO_OE 対応のピンビット　0:LOW,1:HIGH
  - 1を設定するには、SIO GPIO_OE_SETの対応ビットに1をセット
  - 0を設定するには、SIO GPIO_OE_CLRの対応ビットに1をセット
  - GPOEは (1<<3）で0x00000008
  - GPOEは（1<<4）で0x00000010
- PicoとPico2ではマッピングアドレスが違う

Pioc	アドレス
GPIO_OUT_OE_SET	0xd0000024
GPIO_OUT_OE_CLR	0xd0000028

Pioc2	アドレス
GPIO_OUT_OE_SET	0xd0000038
GPIO_OUT_OE_CLR	0xd0000040

SIO: GPIO_OE_SET Register

Offset: 0x024

Description

GPIO output enable set

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:0	Perform an atomic bit-set on GPIO_OE, i.e. `GPIO_OE |= wdata`	RW	0x00000000

SIO: GPIO_OE_CLR Register

Offset: 0x028

Description

GPIO output enable clear

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:0	Perform an atomic bit-clear on GPIO_OE, i.e. `GPIO_OE &= ~wdata`	RW	0x00000000

led_shift : ポートレジスタ書き込み関数

- LED点灯・消灯制御
  - SIO GPIO_OUT_SET/CLRレジスタへの書き込み
  - アクセスサイズが4bytesなので，uint32_t 型のポインタへキャストして書き込む

```
void led_out(uint32_t led_data){
    uint32_t reg1 = ~led_data;
    uint32_t reg2 = led_data;
    *((volatile uint32_t *) 0xD0000018) = reg1;    /* 書き込み*/
    *((volatile uint32_t *) 0xD0000014) = reg2;
}
```

今回はこれで十分だが、

再利用性を考えると不十分なコード?

- 将来的にGPIOのLEDが接続されているビット以外のビットに他の回路が接続された場合、このコードの実行により発生する不具合は?

led_shift : 変更値以外の保存

- 再利用性を考えると操作対象のビット以外は変更するべきではない
  - 操作対象のビット以外は変更前の値を書き込む
- 変更前の値をどこから読むか?
  - ポートレジスタから?

通常のMemory Mapped IOの場合、ポートレジスタに直接データを書き込む、この場合、周辺回路のレジスタはメモリマッピングされているが、メモリとは異なり書き込んだ値が読み込めるとは限らないため注意が必要

Pico/Pico2の場合、ROMモニタの設定で使用したSIO_GPIO_OUTを使えば直接outputポートにデータ出力できる
  - このレジスタは、設定値の保存や読み出しができます

- ポートレジスタが値を保持しない場合はグローバル変数に値を保持する

led_shift : 特定ビットのみの変更

- 変更前の値はポートレジスタから取得可能
- 取得した値のうち、LEDが接続されているビット4-3にのみ引数 led_data の値を反映させる
- 例えば..
  - ポートレジスタの値(LED1:ON): 0x4168 (“0100 0001 0110 1000")
  - 引数led_data(LED2:ON) : 0x0010 (“0000 0000 0001 0000")
  - 書き込みたい値は : 0x4190 (“0100 0001 0111 0000")
- プログラムによる0x4190の生成手順
  1. ポートレジスタの値のビット4,3をクリア
     (0x4168 (“0100 0001 0110 1000”))
  2. led_dataのビット4,3のみを取り出す
  3. 1.と2.で生成した値の論理和を生成
     (“0100 0001 0110 0000”|“0000 0000 0001 0000”
     = “0100 0001 0111 0000”)
- 特定ビットのクリアや特定ビットの取り出しが必要

  ➡ ビット演算により実現

TOPPERS

led_shift : led_outの変更

- ポートレジスタのビット4,3以外は元の値を保持

Pico/Pico2の場合、SET、RESETが別ポートであり,0を書き込んだビットは変化しないため、ビット演算の必要はない

```
void
led_out_no_macro(uint32_t led_data){
    uint32_t reg1, reg2;

    reg1 = ~led_data & 0x00000018;
    reg2 = led_data & 0x00000018;

    *((volatile uint32_t *)0xD0000018) = reg1;
    *((volatile uint32_t *)0xD0000014) = reg2;
}
```

1. オフ(0)にするビットを取り出す

2. オン(1)にするビットを取り出す

3. GPIO_CLRにオフにするデータを書き込む

4. GPIO_SETにオフにするデータを書き込む

TOPPERS

led_shift : マクロによる記述1

定数はマクロ（記号定数）で記述する

- レジスタのアドレスや接続ビットの値を直接プログラム中に記述すると可読性が悪くなる（マジックナンバー）
- レジスタ値や接続ビットの変更に対応が可能

- SIO_GPIOレジスタのマクロ定義

```
#define TADR_SIO_GPIO_OUT_CLR  (TADR_SIO_BASE+TOFF_SIO_GPIO_OUT_CLR)
#define TADR_SIO_GPIO_OUT_SET  (TADR_SIO_BASE+TOFF_SIO_GPIO_OUT_SET)
```

- GPIOのLEDの接続ビット定義

```
#define PINPOSITION3    3
#define PINPOSITION4    4

#define    LED1          (1<<PINPOSITION3)          /* LED1ビット定義 */
#define    LED2          (1<<PINPOSITION4)          /* LED2ビット定義 */
#define    LED_MASK      (LED1 | LED2)
```


TOPPERS

led_shift : マクロによる記述2

- マクロ化すると、参照のマップドIOの定義ファイルを変更することで、自動的にアドレスを変更している

Picoのマップドの定義:rp2040.h

```
#define TADR_SIO_BASE           0xD0000000
#define TOFF_SIO_CPUID          0x0000      /* (R) Processor core identifier  */
#define TOFF_SIO_GPIO_IN        0x0004      /* (R)  Input value for GPIO pins */
#define TOFF_SIO_GPIO_HI_IN     0x0008      /* (R)  Input value for QSPI pins */
#define TOFF_SIO_GPIO_OUT       0x0010      /* (RW) GPIO output value */
#define TOFF_SIO_GPIO_OUT_SET   0x0014   /* (W)  GPIO output value set */
#define TOFF_SIO_GPIO_OUT_CLR   0x0018   /* (W)  GPIO output value clear */
#define TOFF_SIO_GPIO_OUT_XOR   0x001C /* (W)  GPIO output value XOR */
```

Pico2のマップドの定義:rp2350.h

```
#define TADR_SIO_BASE           0xD0000000
#define TADR_SIO_NONSEC_BASE    0xD0020000
#define TOFF_SIO_CPUID          0x0000      /* (R) Processor core identifier  */
#define TOFF_SIO_GPIO_IN        0x0004      /* (R)  Input value for GPIO pins */
#define TOFF_SIO_GPIO_HI_IN     0x0008      /* (R)  Input value for QSPI pins */
#define TOFF_SIO_GPIO_OUT       0x0010      /* (RW) GPIO output value */
#define TOFF_SIO_GPIO_HI_OUT    0x0014   /* (RW) QSPI output value */
#define TOFF_SIO_GPIO_OUT_SET   0x0018 /* (W)  GPIO output value set */
#define TOFF_SIO_GPIO_HI_OUT_SET  0x001C /* (W)  QSPI output value set */
#define TOFF_SIO_GPIO_OUT_CLR   0x0020 /* (W)  GPIO output value clear */
```


TOPPERS

led_shift : マクロによるled_outの書き換え1

- 何をしているプログラムなのか分かりやすくなる

```
void
led_out_macro1(uint32_t led_data){
    uint32_t reg1, reg2;

    reg1 = ~led_data & LED_MASK;
    reg2 = led_reg & LED_MASK;

   *((volatile uint32_t *)TADR_SIO_GPIO_CLR) = reg1;
   *((volatile uint32_t *)TADR_SIO_GPIO_SET) = reg2;
}
```

led_shift : マクロによる記述

デバイスレジスタでは，ポインタへのキャストの部分も含めてマクロ化する場合が多い

- 型の指定間違いを防ぐ
  - デバイスレジスタにはアクセスサイズがある
- 記述が簡素になる
- マクロ化の方法や命名規則はプログラムやプロジェクト単位で一定化しておく

- GPIOレジスタ

```
#define TREG_SIO_GPIO_CLR   ((volatile uint32_t*)TADR_SIO_GPIO_CLR)
#define TREG_SIO_GPIO_SET  ((volatile uint32_t*)TADR_SIO_GPIO_SET)
```

- ポインタ部までマクロ化(プログラム例はない)

```
#define TREG_SIO_GPIO_CLR *((volatile uint32_t*)TADR_SIO_GPIO_CLR)
#define TREG_SIO_GPIO_SET *((volatile uint32_t*)TADR_SIO_GPIO_SET)
```

led_shift : マクロによるled_outの書き換え2

```
void
led_out_macro2(uint32_t led_data){
    uint32_t reg1, reg2;

    reg1 = ~led_data & LED_MASK;
    reg2 = led_data & LED_MASK;

   *TREG_SIO_GPIO_CLR = reg1;
   *TREG_SIO_GPIO_SET = reg2;
}
```

- 1行での記述も可能

```
*TREG_SIO_GPIO_CLR  = ~led_data & LED_MASK;
```


TOPPERS

led_shift : 構造体による記述

構造体を用いてデバイスレジスタのアドレスを指定する

- ビット単位の読み書きが容易に記述可能
  - コンパイラ依存であり，利用できないコンパイラもある
- 記述方法もコンパイラによって異なる

```
/* ビット構造体 */
struct bit_def {
  char  b0:1;
  char  b1:1;
  char  b2:1;
  char  b3:1;
  char  b4:1;
  char  b5:1;
  char  b6:1;
  char  b7:1;
};
```

ビット
フィールド

```
struct byte_def{
    struct bit_def bit0;
    struct bit_def bit1;
    struct bit_def bit2;
    struct bit_def bit3;
};
```

```
/* ワード構造体 */
union word_def{
    struct byte_def  byte;
    uint32_t word;
};
```


TOPPERS

led_shift : 構造体による書き換え

バイトアクセスによる記述

ビットアクセスによる記述

```
void
led_out_struct(uint32_t led_data){
    union word_def  reg1, reg2;

    reg1.word = 0;
    reg2.word = 0;
    reg1.byte.bit0.b4 = ((LED2 & led_data) == 0)? 1 : 0;
    reg2.byte.bit0.b4 = ((LED2 & led_data) != 0)? 1 : 0;
    reg1.byte.bit0.b3 = ((LED1 & led_data) == 0)? 1 : 0;
    reg2.byte.bit0.b4 = ((LED1 & led_data) != 0)? 1 : 0;
    *TREG_SIO_GPIO_CLR = reg1.word;
    *TREG_SIO_GPIO_SET = reg2.word;
}
```


TOPPERS

led_shift : デバイスアクセス関数による記述

デバイスへのアクセスを専用の関数により実現

- プログラムのハードウェア依存性が弱まる
- シミュレーション環境への対応が容易

例）デバイスドライバ設計ガイドラインのアクセスインタフェース

- ハーフデータの読み込みと書き込み

```
uint32_t
sil_rew_mem(uint32_t addr){
    return *((volatile uint32_t *)addr);
}
```

```
void
sil_wrw_mem(uint32_t addr, uint32_t bdata){
    *((volatile uint32_t *)addr) = bdata;
}
```


TOPPERS

led_shift : デバイスアクセス関数による書き換え

- アクセスサイズによって呼び出す関数が異なる

```
void
led_out_sil(uint32_t led_data){
    uint32_t reg1, reg2;

    reg1 = ~led_data & LED_MASK;
    reg2 = led_data  & LED_MASK;

    sil_wrw_mem((uint32_t *)TADR_SIO_GPIO_CLR, reg1);
    sil_wrw_mem((uint32_t *)TADR_SIO_GPIO_SET, reg2);
}
```

led_shift : 初期化関数 led_init()

- ポートを出力方向に設定
- GPIO_IEはLED接続ビット以外を変更しないようにする

設定内容は、ROMモニタの実習と同じ

```
void
led_init(void){
    uint8_t  pinpos;

    for(pinpos = PINPOSITION3 ; pinpos <= PINPOSITION4 ; pinpos++){
        /* PDS_BANK0_GPIO設定 */
        TREG_PADS_BANK0_GPIO[pinpos] = 0x00000056;;
        /* IO_BANK0_CTROL_GPIO設定 */
        TREG_IO_BANK0_GPIO_CTRL[pinpos * 2 + 1] = 0x00000005;
        /* SIO_GPIO_OE設定 */
        *TREG_SIO_GPIO_OE_SET = 1 << pinpos;
    }
}
```

LEDの位置で LOOP

TOPPERS

led_shift : ビジーウェイト関数 busy_wait()

- forループにより任意時間待ちを生成
- ループ回数は1msをベース
- ループ変数はint型
  - 1msをベースに32ビット長の時間に対応するため
  - 引数1に対して、1ナノ秒のソフト待ちを行うsil_dly_nseを用意

```
#define DELAY_LOOP (250*1000)

void
busy_wait(void){
    volatile int i;
    for(i = 0; i < DELAY_LOOP; i++)
            sil_dly_nse(1000);
}
```

long(32bit)指定

TOPPERS

led_shift : startup_rp2040.S

アセンブリ言語で記述されたファイル

- 各プログラムで共通に使用できる
- セクションの配置
  - ROM実行の場合、ROM上に配置
  - RAM実行の場合、RAM上に配置
- ベクタテーブル
  - ROM実行の場合、ROM上に配置
  - RAM実行の場合、RAM上に配置
    - ベクタテーブルの設定アドレスを切り替え
- スタートアップルーチン
  - セクションの初期化
  - main関数の呼び出し

TOPPERS

led_shift : startup_rp2040.S（ROM実行）

- リンクファイル：rp2040_rom.ldで指定

```
; ------- RAM 領域 -------------
    ORG          0x20000000
    SECTION  .data
    SECTION  .bss
```

```
; ------- ROM 領域 -------------
    ORG          0x10000000
    SECTION  .boot2
    SECTION  .text
    SECTION  .data
```

セクション

- .boot2　　：rp2040パディングデータ、書き換え不可
- .text　　：割込みベクタ、プログラム、書き換え不可データ
- .data　　：初期値ありデータ

  ROM化される場合、ROM上の初期値LOADADDR(.data)

  RAM上の参照領域ADDR(.data)に分けられる

  スタートアップルーチンでROM上のデータをRAM上にコピーする
- .bss　　：初期値なしデータ

  スタートアップルーチンでゼロに初期化

TOPPERS

led_shift : startup_rp2040.S（RAM実行）

- リンクファイルstm32f4xx_ram.ldで指定

```
; ------- RAM 領域 -------------
    ORG          0x20000000
    SECTION  .boot2
    SECTION  .text
    SECTION  .data
    SECTION  .bss
```

セクション

- すべてのデータはROMモニタのldコマンドでダウンロード
- .vector　　: rp2040パディングデータ、書き換え不可
- .text　　:割込みベクタ、プログラム、書き換え不可データ
- .data　　: 初期値ありデータ
  RAM上にダウンロードするため、ROM領域からの.dataのコピーは不要
- .bss　　: 初期値なしデータ
  ROMモニタのgoコマンド実行後、ゼロに初期化

led_shift : startup_rp2040.S 固定ベクタ

- rp2040はリセット後、ROM-BIOSが起動し，フラッシュROMを参照して0x10000104番地のベクタから実行を始める
- ROM実行の場合、0x10000104にリセット関数(Reset_Handler)へのポインタを記載する
- 0x10000100番地にリセット時のmspのアドレスを置く

```
g_pfnVectors:
  .word  _estack
  .word  Reset_Handler
  .word  NMI_Handler
  .word  HardFault_Handler
  .word  MemManage_Handler
  .word  BusFault_Handler
  .word  UsageFault_Handler
  .word  0
  .word  0
  .word  0
  .word  0
  .word  SVC_Handler
  .word  DebugMon_Handler
```

RESET時のスタックポインタ

リセット時の関数ポインタ

Pico2:Cortex-M33はベクタのアドレスが違う、Pico2:RISC-Vは起動手順が違う

TOPPERS

startup_rp2040.S スタートアップルーチン

- アセンブリ言語で記述
  - 設定しているレジスタをマニュアルでチェック
  - C言語と同様に定数はマクロ化

```
 Reset_Handler:

#ifdef RP2040_CORE1_RESET
  :
#endif

  ldr   r0, =_estack
  mov   sp, r0          /* set stack pointer */

/* Copy the data segment initializers from flash to SRAM */
  movs r1, #0
  b LoopCopyDataInit

CopyDataInit:
  ldr r3, =_sidata
  ldr r3, [r3, r1]
  str r3, [r0, r1]
  adds r1, r1, #4
```

_sdataから_edataまでに
_sidataをコピー

TOPPERS

startup_rp2040.S セクションの初期化

```
LoopCopyDataInit:
  ldr r0, =_sdata
  ldr r3, =_edata
  adds r2, r0, r1
  cmp r2, r3
  bcc CopyDataInit
  ldr r2, =_sbss
  b LoopFillZerobss
/* Zero fill the bss segment. */
FillZerobss:
  movs r3, #0
  str  r3, [r2]
  adds r2, r2, #4

LoopFillZerobss:
  ldr r3, = _ebss
  cmp r2, r3
  bcc FillZerobss

/* Call the clock system intitialization function.*/
  bl  SystemInit
```

_sbssから_ebss
までをゼロクリア

C言語のスタートアップ
ルーチンSystemInitを実行

TOPPERS

led_shift : start.S main関数の呼び出し

- ハードウェアの初期化，data,bssの初期化が終了後，main関数を呼び出す

```
/* Call the application's entry point.*/
start_0:
  bl  main
LoopForever:
  b  LoopForever
```

Cortex-M0+アセンブラでのmain関数呼び出し

```
  /* argc = argv = 0 */
  li a0, 0
  li a1, 0
  call main
1:
  j 1b
```

RISC-Vアセンブラでのmain関数呼び出し

TOPPERS

ポーリングプログラム

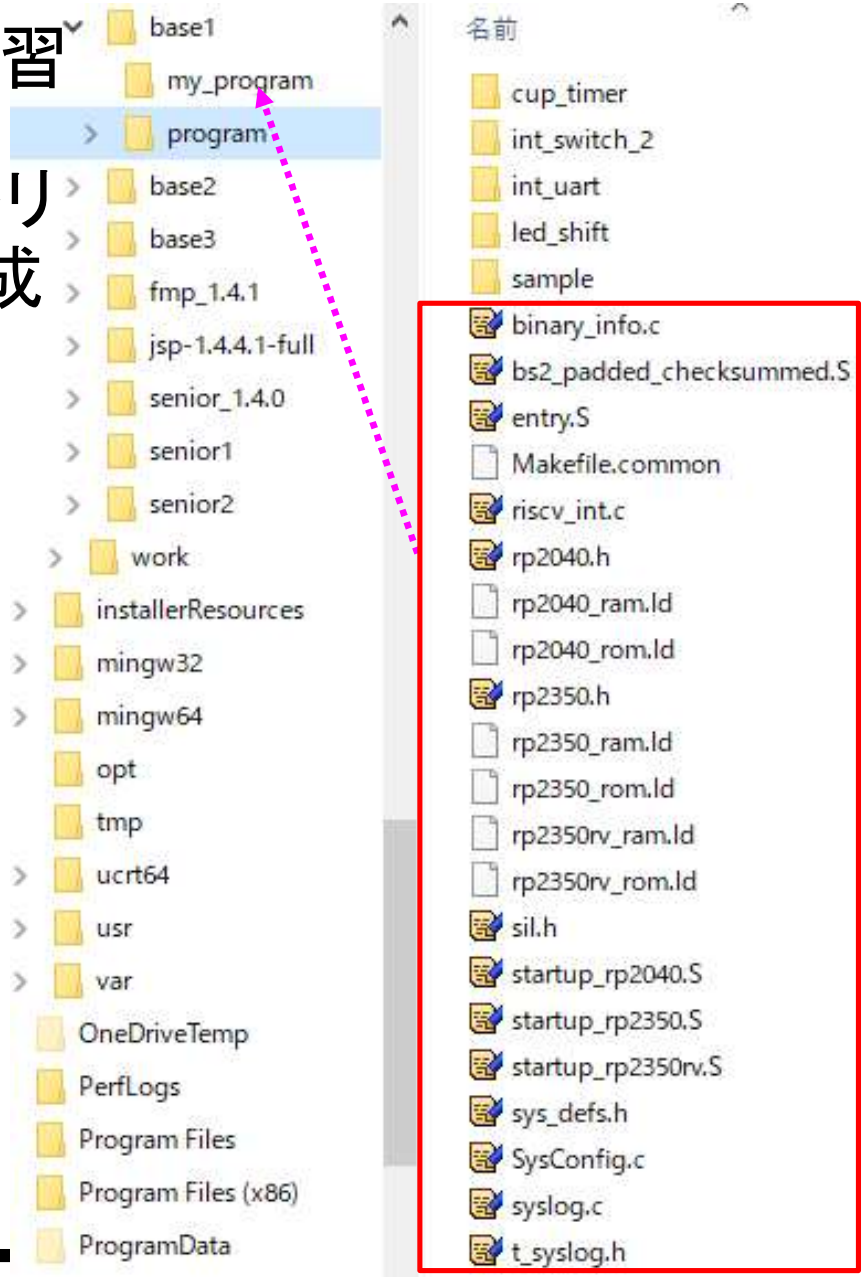
1. LEDプログラム
   - プロジェクトの作成方法
2. スイッチプログラム
3. タイマプログラム
4. シリアルI/Oプログラム

プロジェクトの作成方法

新規プロジェクトの作成方法を学習

- プロジェクトのためのディレクトリ（my_program/led_count）を作成
  - 作成場所に制約はない
- 教材ディレクトリ /programの直下 にある以下のファイルをmy_program フォルダにコピーする

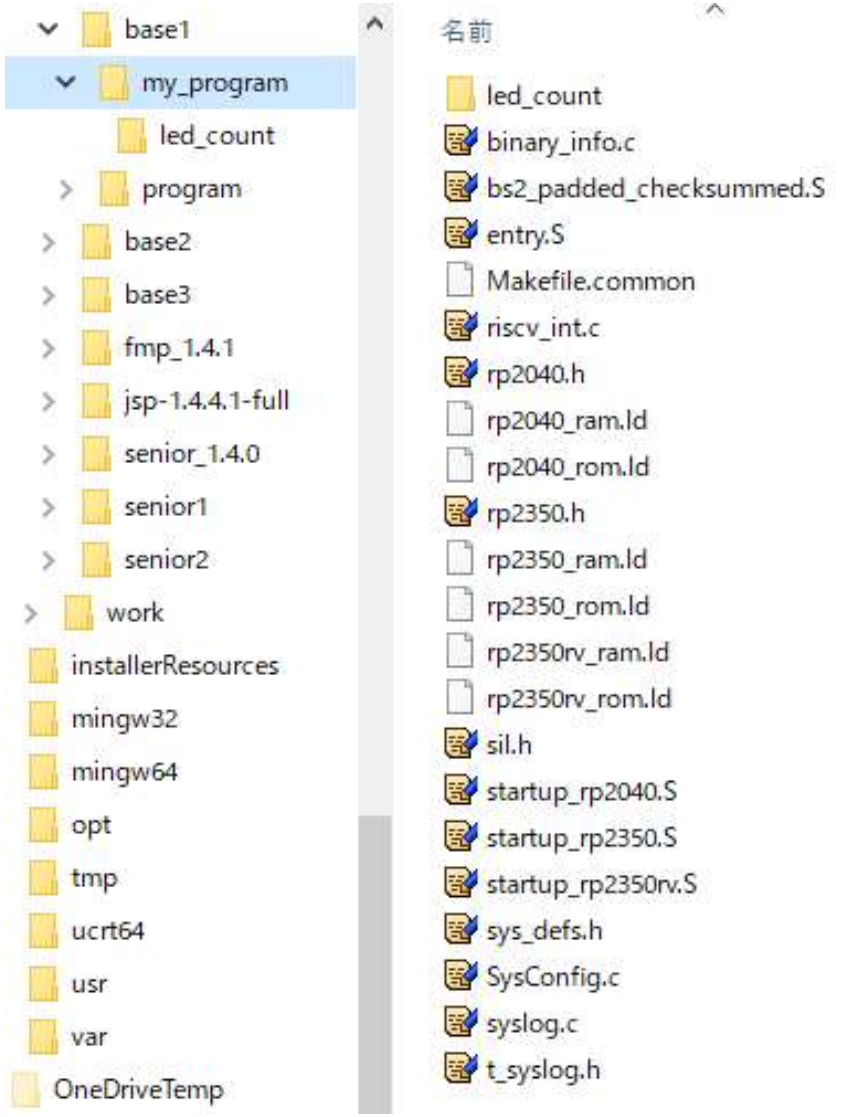
binary_info.c
bs2_padding_checksummed.S
entry.S
Makefile.commn
riscv_int.c
:
t_syslog.h



TOPPERS

プロジェクトの作成方法：C言語プログラム

- コンパイルするC言語プログラムを用意する
- program/led_shift/led_shift.cの内容をコピーして，led_countのディレクトリにled_count.c として作成する
- Makefileをそのままコピーする



Makefileの書き換え

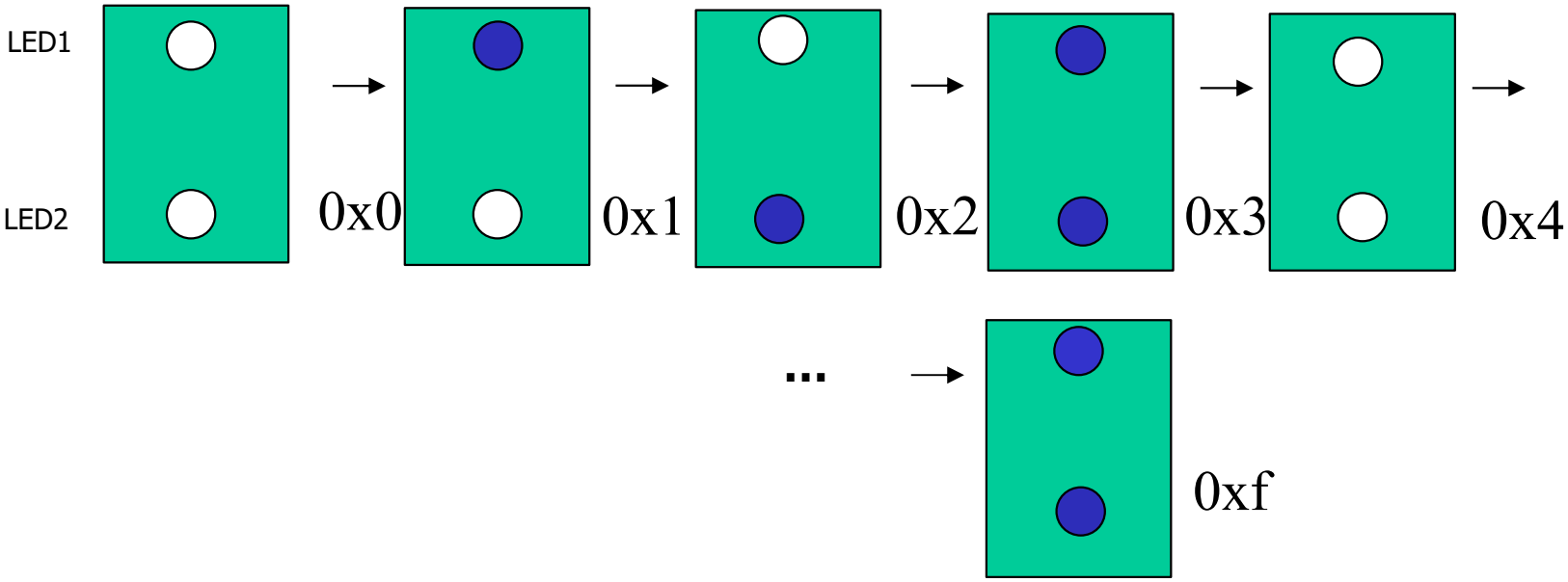
- コピーしたMakefileをエデッタで開く
- 17行目のOBJNAMEをled_shiftからled_countに変更する

```
#
#  オブジェクトファイル名の定義
#
OBJNAME = led_count
#ifdef OBJEXT
        OBJFILE = $(OBJNAME).$(OBJEXT)
#else
        OBJFILE = $(OBJNAME)
```


TOPPERS

LEDプログラム : led_count 概要

- LEDを2進数の表示器とみなし，一定時間毎にカウンタの0,1ビットを表示する（カウントアップ）



- 学習内容
  - プロジェクトの作成方法
  - 出力値のエンコード方法
- 作成方法
  - led_shiftをベースに作成

led_count : メイン関数

- 無限ループで一定時間毎に変数を引数にLEDの表示を更新する関数を呼び出し，変数をインクリメントする
- led_out_bdigit()は引数の下位2bitをLEDに反映させる

```
void
main(void){
    uint32_t count = 0;

    led_init();
    for (;;) {
        led_out_bdigit(count++);
        busy_wait();
    }
}
```

ハードウェア初期化

LED設定

TOPPERS

led_count : 作成

プログラム led_count を作成せよ

- 手順
  - led_out_bdigit()を作成
    - 引数の下位2bitをそのままポートに書き込むと正しく表示されないため（LED1はビット3に接続），引数見て，対応するビットをONにする必要がある
    - LED接続ポートへの書き込みは，led_out_xxx() 関数を用いる

led_count : led_out_bdigit()

- 引数の各ビットをチェックし，0ビット目が’1’ならLED1をON，1ビット目が’1’ならLED2をONとなるようにled_dataを設定する
- led_dataの初期値は，全てのLED OFF とする0x00にしておく

```
void
led_out_bdigit(uint32_t data){
    uint32_t led_data = 0x00;
    if ((data & 0x01) == 0x01) {
        led_data = led_data | LED1;
    }
    if ((data & 0x02) == 0x02) {
        led_data = led_data | LED2;
    }
    led_out(led_data);
}
```

初期値

LED1設定

LED2設定

LED設定
led_out以下の5つの関数から選ぶ
①led_out_no_macro
②led_out_macro1
③led_out_macro2
④led_out_struct
⑤led_out_sil

TOPPERS

デバッグ表示の記載例

- 正しくLEDが表示されない場合
- led_out_bdigitの引数が正しくLEDの設定値になっているか確認のためにsys_printf文を入れてデバッグする（make DEBUG=1でビルド）

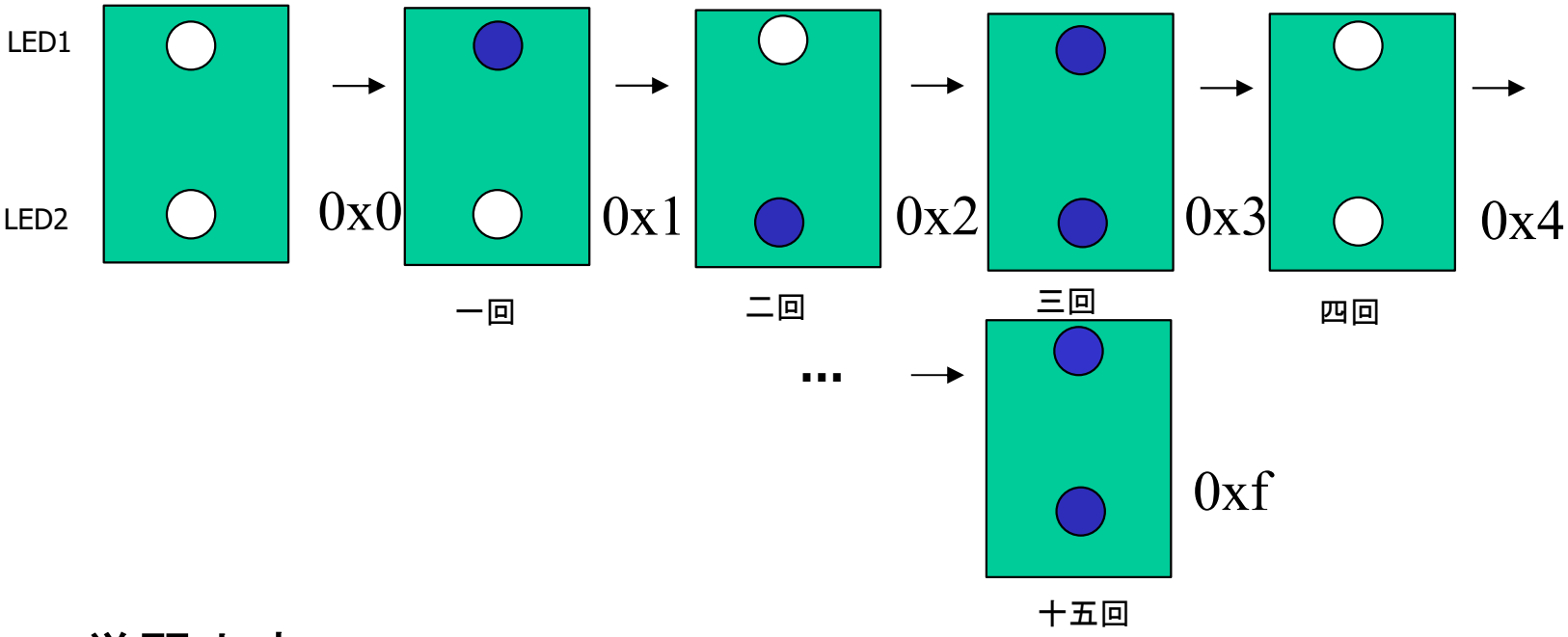
```
void
led_out_bdigit(unsigned char data){
    ...省略...
    led_out(led_data);
    sys_printf("led_out_bdight data[%08x]¥n", led_data);
}
```

ポーリングプログラム

1. LEDプログラム
   ・プロジェクトの作成方法
2. スイッチプログラム
3. タイマプログラム
4. シリアルI/Oプログラム

スイッチプログラム：switch_push 概要

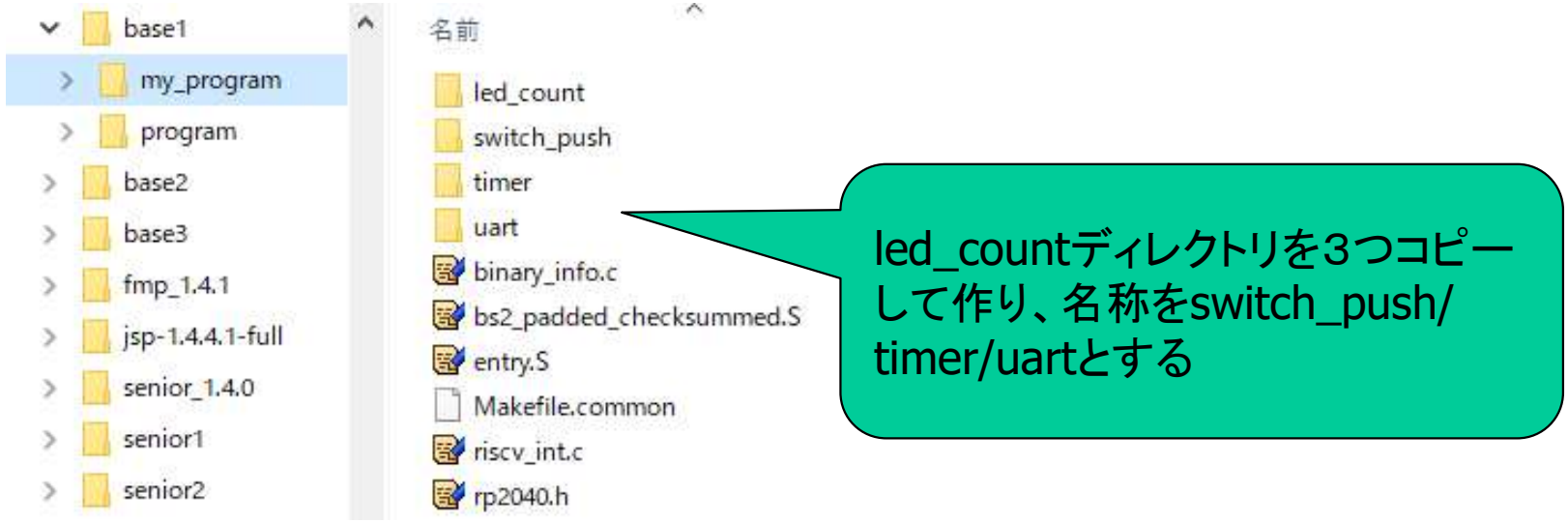
- プッシュスイッチの押下回数を、LEDにバイナリ表示する



- 学習内容
  - ポートからの読み込み
- 作成方法
  - led_countをmy_programにコピーし、ディレクトリ名を書き換えて作成（次のスライドで説明）

led_countデレクトリのコピー

- 前の演習で作成したmy_program中のled_countディレクトリのプログラムを元に以下の演習を行います
  - switch_push(この演習)
  - timer(ポーリングタイマ)
  - uart(ポーリングUART)
- lcd_countをmy_program中に3つコピーしてそれぞれのディレクトリ名を、上記の3つに変更します



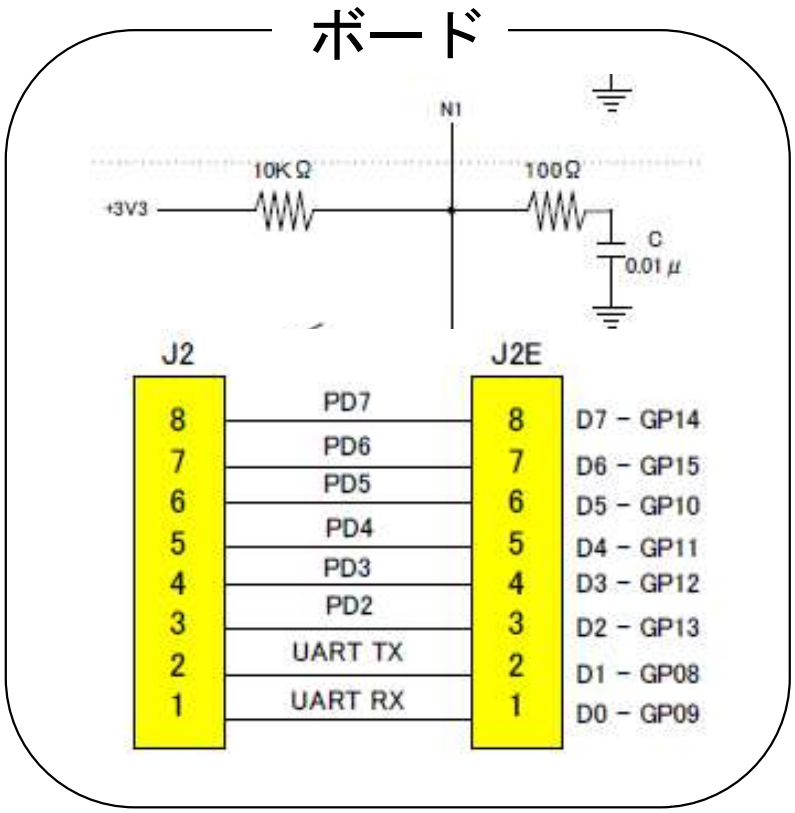
switch_push : 作成

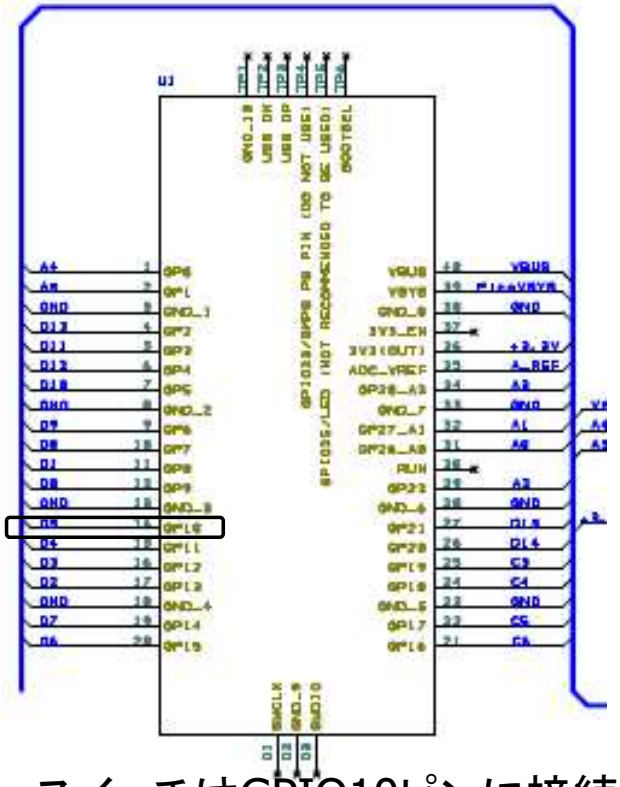
プログラム switch_push を作成

- 手順
  - マニュアルによるプッシュスイッチの接続の確認
  - ソース・ファイルとMakefileをswitch_pushに書き換え
  - マクロの定義
  - 初期化関数 switch_push_init() の作成
    - ポートを入力モードへ
  - プッシュスイッチ状態取得関数 switch_push_sense() の作成
    - ポートの読込み
  - メイン関数
    - スイッチの変化を捉える(OFFからONへ)

switch_push : プッシュスイッチの接続

- マニュアルにより確認する
- マイコンのGP10ピンに接続されている
- ONで電流が流れ、GP10にプッシュスイッチSW1の状態が取り込める
  - マイコンからはONで’0’が，OFFで’1’が読み込める





プッシュスイッチはGPIO10ピンに接続している

TOPPERS

switch_push : プッシュスイッチの接続ポート

- ポートGP10に接続
  - プッシュスイッチ: ビット10
- ポートGPIOの構成レジスタと初期化
  - PDS_BANK0 GP10 : pico:0x4001c02c/pico2:0x4003802c
    - PDE 0 / PUE 0 / IE 1 / OD なので、設定値 0x00000052
  - IO_BANK0 CTRL GPIO10 : pico:0x40100050/pico2:0x40028050
    - FUNCSELをGPIOに設定、設定値 0x00000005
  - SIOのOE設定を0に設定 : 10ビットの場所を0に設定
- スイッチ状態の取得
  - SIO GPIO_IN（0xd0000004）の10ビット目を参照

SIO: GPIO_IN Register

Offset: 0x004

Description

Input value for GPIO pins

Table 19. GPIO_IN Register

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:0	Input value for GPIO0..29	RO	0x00000000

TOPPERS

switch_push : マクロ定義

- ポインタ型へのキャストまで含めてマクロ化する（方針）

- SIO GPIO_IN/SIO_GPIO_OE_CLRレジスタのマクロ定義

```
#define TADR_SIO_GPIO_OE_CLR  (TADR_SIO_BASE+TOFF_SIO_GPIO_OE_CLR)
#define TADR_SIO_GPIO_IN      (TADR_SIO_BASE+TOFF_SIO_GPIO_IN)
#define TREG_SIO_GPIO_OE_CLR  ((volatile uint32_t *)(TADR_SIO_GPIO_OE_CLR))
#define TREG_SIO_GPIO_IN      ((volatile uint32_t *)(TADR_SIO_GPIO_IN))
```

- GPIOポートのプッシュスイッチの接続ビット定義

```
#define PSW1            (1 << PINPOSITION10)
#define PSW_MASK        (PSW1)
#define PSW1_PINNO      PINPOSITION10
```

switch_push : 初期化関数

- GPIOポートのビット0を入力モードへ
  - GPIOポート方向レジスタのビット4を’00’に設定する

```
void
switch_push_init(void){
    uint8_t  pinpos = PSW1_PINNO;

    /* PDS_BANK0_GPIO設定 */
    TREG_PADS_BANK0_GPIO[pinpos] = 0x00000052;
    /* IO_BANK0_CTROL_GPIO設定 */
    TREG_IO_BANK0_GPIO_CTRL[pinpos * 2 + 1] = 0x00000005;
    /* SIO_GPIO_OE設定 */
    *TREG_SIO_GPIO_OE_CLR = 1 << pinpos;
}
```

switch_push : プッシュスイッチ状態取得関数

- switch_push_sense()
- GPIOのビット10の値を取得

```
uint32_t
switch_push_sense(void){
    uint32_t reg;

    reg = *TREG_SIO_GPIO_IN ^ PSW1;
    return reg  & PSW_MASK;
}
```

ビット10のみ有効に

TOPPERS

switch_push : main()関数

- カウント用変数led_dataの用意
- ハードウェアの初期化
- 無限ループ内でプッシュスイッチの状態をチェックしてSW1がOFFからONに変化した場合は，led_dataをインクリメントする
- 状態の変化の捉え方
  - プッシュスイッチの以前の状態を保存して比較
  - ローカル変数を用意して前回の状態取得時の状態を保存する
- スイッチの状態（ON,OFF）はマクロ化する

```
#define SW_ON     1
#define SW_OFF    0
```

switch_push : main()関数

```
void  main(void){
    uint32_t count = 0;
    uint32_t sw_data;
    uint8_t sw1_state = SW_OFF;
    led_init();
    switch_push_init();
    for (;;) {
        sw_data = switch_push_sense();
        if(((sw_data & PSW1) == PSW1) && (sw1_state == SW_OFF)){
            led_data++;
        }
       sw1_state = ((sw_data & PSW1) == PSW1)? SW_ON : SW_OFF;
       led_out_bdigit(count);
   }
}
```

SW1状態保存用変数

プッシュスイッチ状態取得

SW1がOFFからONになったか

SW1の状態を保存

LED設定

TOPPERS

ポーリングプログラム

1. LEDプログラム
   ・プロジェクトの作成方法
2. スイッチプログラム
3. タイマプログラム
4. シリアルI/Oプログラム

タイマプログラム：timer 概要

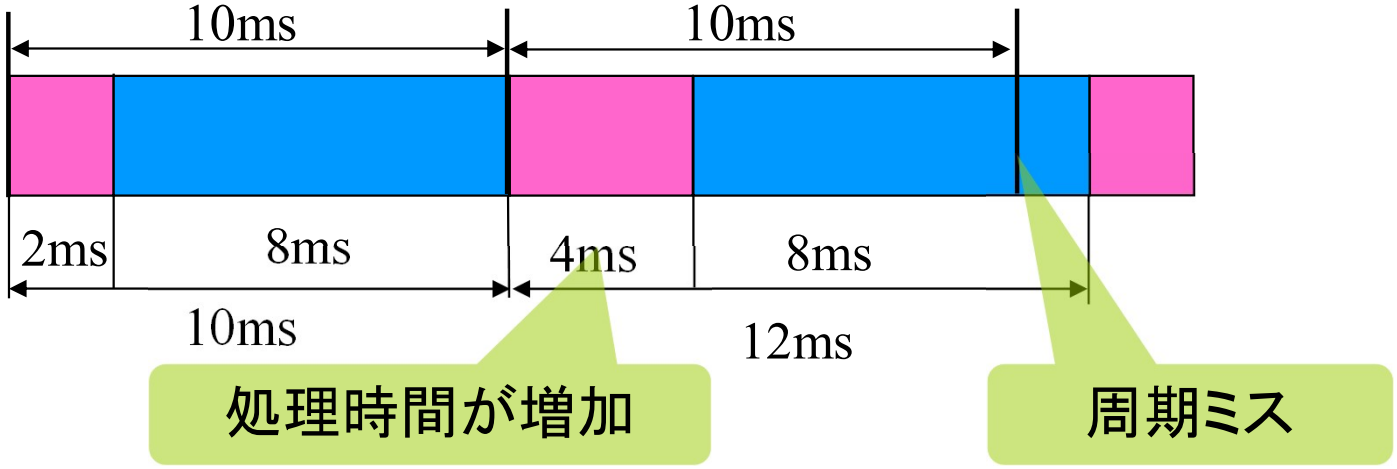
- ハードウェアタイマにより正確に1秒間隔でLEDの点灯をカウントアップする

- 学習内容
  - ハードウェアタイマの使い方
  - カウントソース，カウント値の決定方法

- 作成方法
  - led_count をコピーし、ディレクトリ名をtimerに書き換えて作成

timer : ハードウェアタイマによる時間生成

- led_countでは，時間生成にbusy_wait()関数を使用
  - for()ループにより一定時間を生成
- ループでは正確な時間が生成できない
  - そもそもループでは正確な時間は作れない
  - 正確な周期処理を実現が困難
  - 周期 = 周期処理の実行時間 + ループの実行時間
    - 周期処理の実行時間が変わると，それに合わせてループの長さを変更する必要がある



timer : ハードウェアタイマの概要

- Picoのハードウェアタイマ
  - ６４ビットタイマを１個（Pico2は２個）
  - １個のタイマに、４つのコンペアレジスタがあり、４つの割込みを発生できる
  - タイマ・ソースは1μSECのカウントアップ

- タイマ0を使用
  - ６４ビットタイマのLOW（３２ビット）を参照
  - 時間の経過は1μSEC

timer : レジスタ

- TIMERAWH:TIMERAWLを読み出せば、リセット後の1μSEC時間が取り出せる

名前	アドレス	説明
TIMEHW	0x40054000 0x400b0000	63:32のtime書込み
TIMELW	0x40054004 0x400b0004	31:0のtime書込み
TIMEHR	0x40054008 0x400b0008	63:32のtime読み出し timelrを先に読むこと
TIMELR	0x4005400c 0x400b000c	31:0のtime読み出し
ALARM0	0x40054010 0x400b0010	割込み用のマッチカウンタ
TIMERAWH	0x40054024 0x400b0024	63:32のtime読み出し 副作用なし
TIMERAWL	0x40054028 0x400b0028	31:0のtime読み出し 副作用なし

timer : タイマ操作の流れ

- 初期化は最初のターゲット時間を変数として保存
- TIMERAWLと比較して、ターゲット時間を超えれば、タイムアップする
- タイムアップ時、次のターゲット時間を変数として保存

timer : 作成

プログラム timer を作成

- 手順
  - ソース・ファイルとMakefileをtimerに書き換え
  - カウント値の決定
    - TIMERAWLを読み出し、その値＋250msをタイムアップ値とする
  - タイムアップ時
    - タイムアップ値に250msを加算する
  - プログラミング
    - マクロの定義
    - 初期化関数
    - 計時関数
    - メイン関数2

timer : マクロの定義

- タイマ関連のレジスタ

```
#ifndef TADR_TIMER_BASE
#define TADR_TIMER_BASE    TADR_TIMER0_BASE
#endif
#define TADR_TIMER_TIMERAWL  (TADR_TIMER_BASE+TOFF_TIMER_TIMERAWL
#define TREG_TIMER_TIMERAWL ((volatile uint32_t *)(TADR_TIMER_TIMERAWL))
```

Pico2にはTADR_TIMER_BASEの定義がない

- タイマ関連の定義
  - タイムアップのカウントは250 *1000であり、DELAY_LOOPで代用可能

timer : 初期化関数

- 最初のタイムアップ・カウントを決める

```
uint32_t timeup_value;
void
timer_init(void){
    timeup_value = *REG_TIMER_TIMERAWL + DELAY_LOOP;
}
```

タイムアップ・カウントを保存

TOPPERS

timer : 計時関数

- タイムアップするまで待って、次のタイムアップ・カウントを設定する

```
void
timer_wait(void){
    while(timeup_vlaue > *REG_TIMER_TIMERAWL ){
    }
    timeup_value += DELAY_LOOP;
}
```

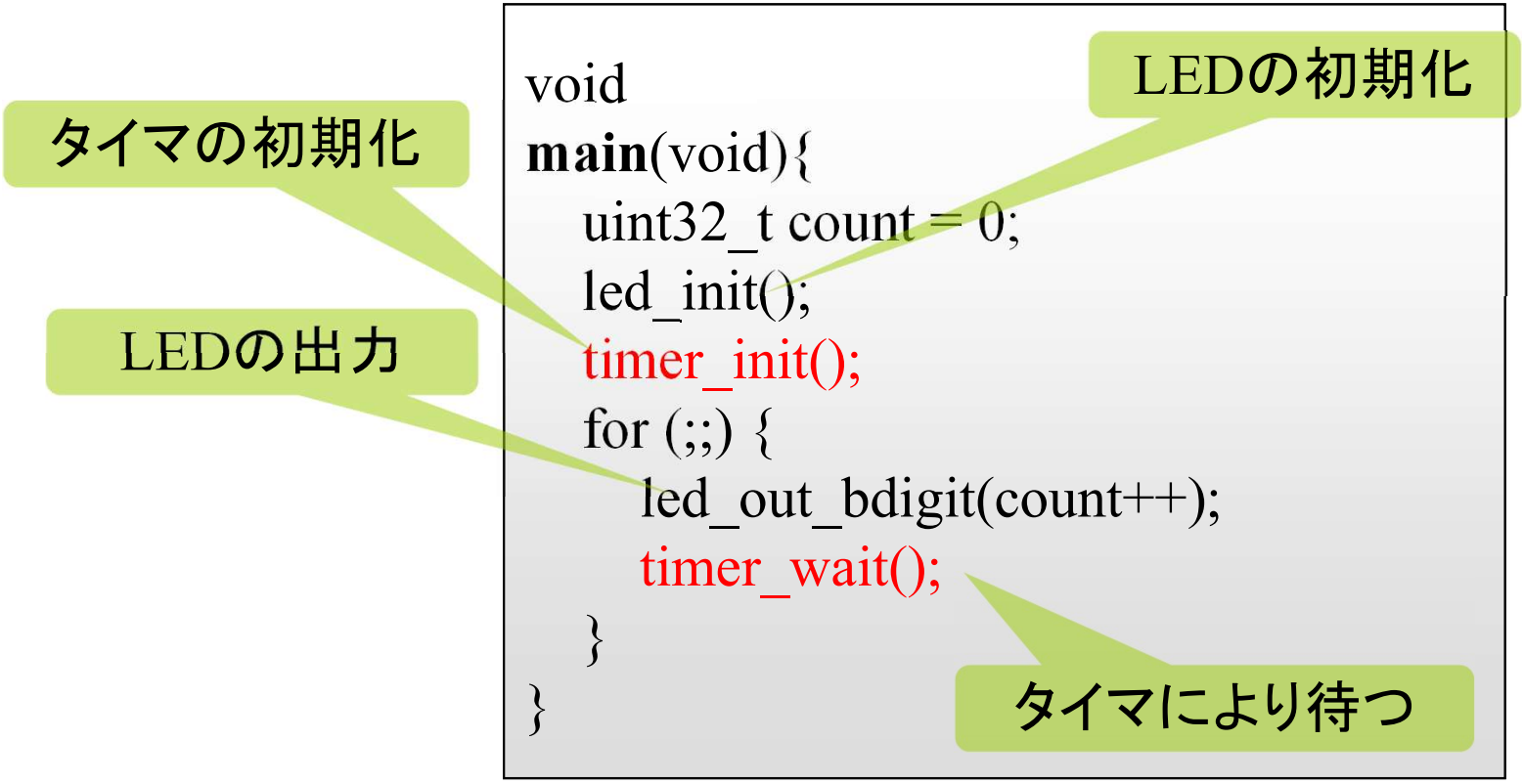
タイムアップするまで待つ

次のタイムアップ・カウントを設定

TOPPERS

timer : メイン関数

- led_countのメイン関数を変更
  - busy_wait() を timer_wait()に変更




```
void
main(void){
    uint32_t count = 0;
    led_init();
    timer_init();
    for (;;) {
        led_out_bdigit(count++);
        timer_wait();
    }
}
```

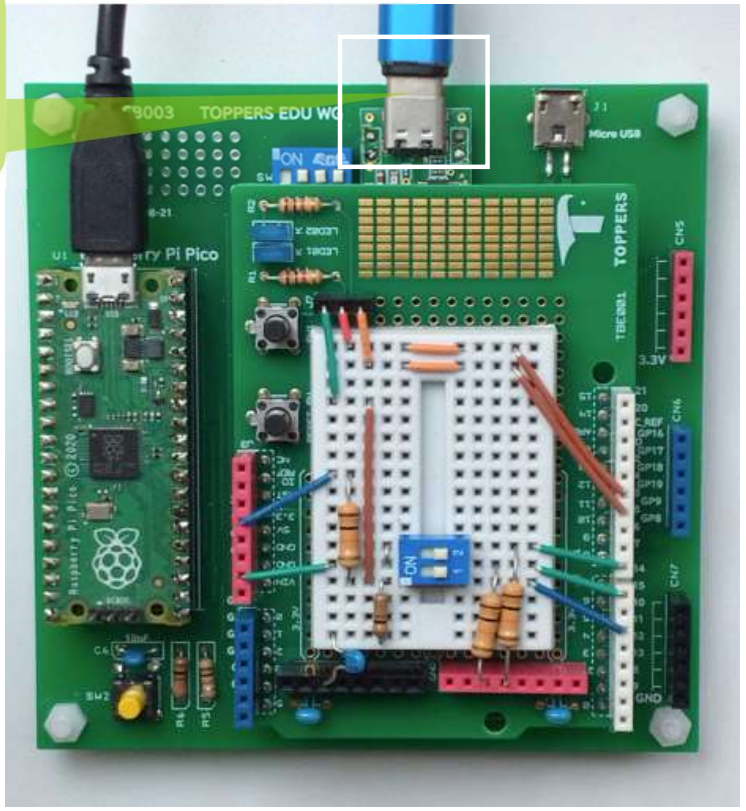
ポーリングプログラム

1. LEDプログラム
   ・プロジェクトの作成方法
2. スイッチプログラム
3. タイマプログラム
4. シリアルI/Oプログラム

シリアルI/Oプログラム : uart 概要

- シリアルI/Oからデータを受信すると，LEDの点灯をカウントアップし，受け取ったデータをシリアルI/Oに送信する（ポーリング）

UART0 TX/RX
COMでUSBに対応

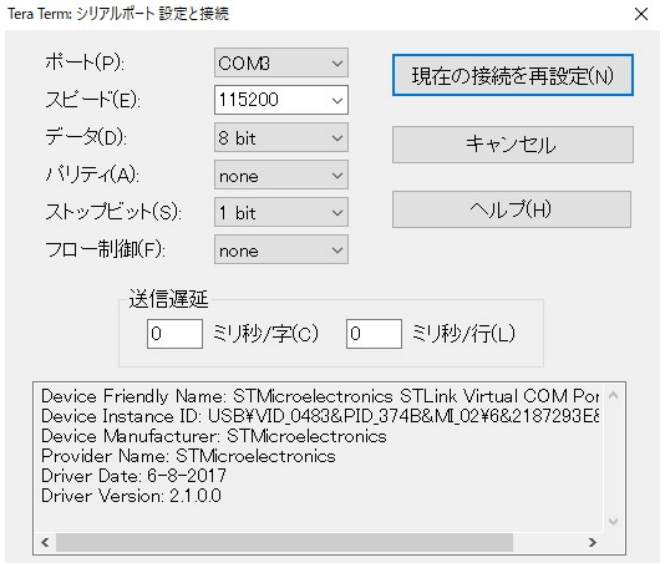


- 学習内容
  - シリアルI/Oの使い方
- 作成方法
  - led_countをコピーし、ディレクトリ名をuartに書き換えて作成
- ボードの設定
  - UART0のTX/RXをUSBシリアルのCOMに接続する

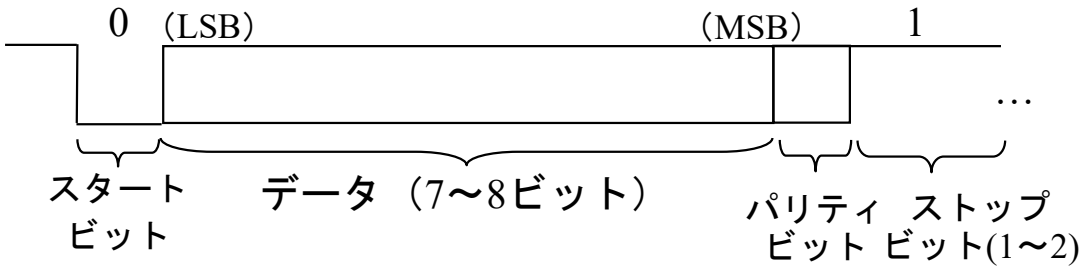
uart : シリアルI/Oのフォーマット

- 通信する双方で同じ設定にする必要がある
  - ボーレート（BPS）　: 9600, 14400, 38400, 57600, 115200
  - データサイズ　　: 7bit, 8bit
  - パリティ　　: なし, even, odd
  - ストップビット　　: 1bit, 2bit
  - フローコントロール　: なし, Xon/Xoff, hardware

・ターミナルソフトの設定例



・データフォーマット



TOPPERS

uart:ピン・クロック設定

- UARTとGPIO(TX/RX)のクロック設定
  - ペリフェラル・クロック(CLK_PERI)を使用
  - スタートアップルーチンで初期化済み
- TX/RXピンのGPIOピン設定
  - TX/RXピン対応のGPIOのピン設定 TX:GP0/RX:GP1
  - PDS_BABK0 GPIO
    - PDE 1 / PUE 0 / IE 1 / OD 0 (設定値:0x00000056)
  - IO_BANK0 GPIO CTRL
    - FUNCSELはUARTファンクション:0x00000002を設定

レジスタ	Pico アドレス	Pico2 アドレス
PDS_BANK0 GP0	0x4001c004	0x40038004
PDS_BANK0 GP1	0x4001c008	0x40038008
IO_BANK0 CTRL GP0	0x40014004	0x40028004
IO_BANK0 CTRL GP1	0x4001400c	0x40028008

uart:デバイスのリセット

- RESETモジュールでuart0のリセットを行う
  - ビット単位でリセットデバイスを指定
  - Picoのuart0デバイスビットは22ビット:0x00400000
  - Pico2のuart0デバイスビットでは26ビット:0x04000000
- リセットの手順
  - RESETS_RESETレジスタにuart0のデバイスビットを1に設定
  - RESETS_RESETレジスタにuart0のデバイスビットを0に戻す
  - RESETS_RESET_DONEレジスタを読み込みuart0のデバイスビットが0に戻れば、リセット終了

レジスタ	Pico アドレス	Pico2 アドレス
RESETS_RESET	0x4000c000	0x40002000
RESETS_RESET_DONE	0x4000c008	0x40002008

uart : 送受信フォーマットとレジスタ

- フォーマット
  - ボーレート(BPS) : 115200, データサイズ : 8bit, パリティ: なし, ストップビット: 1bit, フローコントロール : なし
- uart制御用レジスタは、以下の通り

名前	Picoアドレス	Pico2アドレス	説明
UARTDR	0x40034000	0x40070000	データレジスタ
UARTFR	0x40034018	0x40070018	フラグレジスタ
UARTIBRD	0x40034024	0x40070024	ボーレートレジスタ(整数)
UARTFBRD	0x40034028	0x40070028	ボーレートレジスタ(分数)
UARTLCR_H	0x4003402c	0x4007002c	ラインコントロールレジスタ
UARTCR	0x40034030	0x40070030	コントローラレジスタ
UARTIFLS	0x40034034	0x40070034	FIFOレベル設定レジスタ
UARTIMSC	0x40034038	0x40070038	割込みマスクレジスタ
UARTRIS	0x4003403c	0x4007003c	割込みステータスレジスタ
UARTDMACR	0x40034048	0x40070048	DMAコントロールレジスタ

TOPPERS

uart : 作成

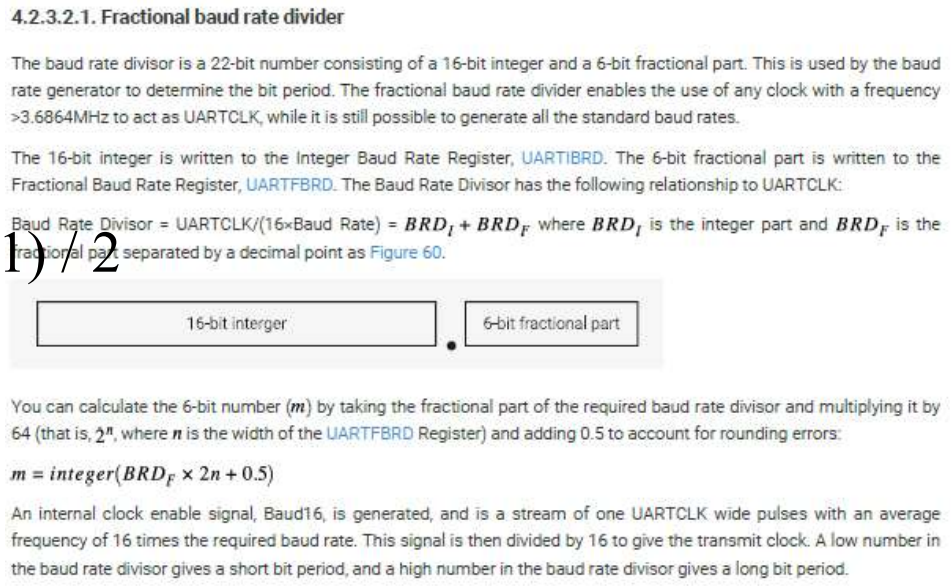
プログラム uart を作成

- 手順
  - ソース・ファイルとMakefileをuartに書き換え
  - 転送速度レジスタの設定値の計算
    - ボーレートが115200bps(DEFAULT_SPEED)になるよう設定
  - その他のレジスタの設定値の決定
  - プログラミング
    - マクロの定義
    - 初期化関数
    - 受信関数
    - 送信関数
    - メイン関数

uart : 転送速度レジスタの設定値の計算

- 転送速度は 115200bps、Picoのクロックは125,000,000
- 125,000,000 / (16 * 1152000) = 67.8186402777
- UARTIBRD = 67
- UARTFBRD = 0.8186402777 * 64 = 52.2777 なので52

- 小数部を7ビット取る計算を行う
  - clock * 128 / (16 * 1152000)
  - r = clock * 8 / 115200
  - UARTIBRD = r >> 7
  - UARTFBRD = ((r & 0x7F) + 1) / 2



4.2.3.2.1. Fractional baud rate divider

The baud rate divisor is a 22-bit number consisting of a 16-bit integer and a 6-bit fractional part. This is used by the baud rate generator to determine the bit period. The fractional baud rate divider enables the use of any clock with a frequency >3.6864MHz to act as UARTCLK, while it is still possible to generate all the standard baud rates.

The 16-bit integer is written to the Integer Baud Rate Register, UARTIBRD. The 6-bit fractional part is written to the Fractional Baud Rate Register, UARTFBRD. The Baud Rate Divisor has the following relationship to UARTCLK:

Baud Rate Divisor = UARTCLK/(16×Baud Rate) = $BRD_I + BRD_F$ where BRD_I is the integer part and BRD_F is the fractional part separated by a decimal point as Figure 60.

You can calculate the 6-bit number (m) by taking the fractional part of the required baud rate divisor and multiplying it by 64 (that is, 2^n, where n is the width of the UARTFBRD Register) and adding 0.5 to account for rounding errors:

$m = integer(BRD_F \times 2n + 0.5)$

An internal clock enable signal, Baud16, is generated, and is a stream of one UARTCLK wide pulses with an average frequency of 16 times the required baud rate. This signal is then divided by 16 to give the transmit clock. A low number in the baud rate divisor gives a short bit period, and a high number in the baud rate divisor gives a long bit period.

TOPPERS

uart : その他のレジスタの設定値

UARTLCR_Hレジスタにフォーマットに従って設定を決定する

- 1ストップ : STP2を0
- パリティなし : PENを0、EPSを0
- データ8ビット : WLENを3
- その他のビットは変更せずなので以下の設定をすればよい
  - ~(WLEN|STP2|PEN|EPS)でマスク
  - (8-5) << 5をORする
  - ~FENでマスク

UART: UARTLCR_H Register

Offset: 0x02c

Description

Line Control Register, UARTLCR_H

Table 443. UARTLCR_H Register

Bits	Name	Description	Type	Reset
31:8	Reserved.	-	-	-
7	SPS	Stick parity select. 0 = stick parity is disabled 1 = either: * if the EPS bit is 0 then the parity bit is transmitted and checked as a 1 * if the EPS bit is 1 then the parity bit is transmitted and checked as a 0. This bit has no effect when the PEN bit disables parity checking and generation.	RW	0x0
6:5	WLEN	Word length. These bits indicate the number of data bits transmitted or received in a frame as follows: b11 = 8 bits b10 = 7 bits b01 = 6 bits b00 = 5 bits.	RW	0x0
4	FEN	Enable FIFOs: 0 = FIFOs are disabled (character mode) that is, the FIFOs become 1-byte-deep holding registers 1 = transmit and receive FIFO buffers are enabled (FIFO mode).	RW	0x0
3	STP2	Two stop bits select. If this bit is set to 1, two stop bits are transmitted at the end of the frame. The receive logic does not check for two stop bits being received.	RW	0x0
2	EPS	Even parity select. Controls the type of parity the UART uses during transmission and reception: 0 = odd parity. The UART generates or checks for an odd number of 1s in the data and parity bits. 1 = even parity. The UART generates or checks for an even number of 1s in the data and parity bits. This bit has no effect when the PEN bit disables parity checking and generation.	RW	0x0
1	PEN	Parity enable: 0 = parity is disabled and no parity bit added to the data frame 1 = parity checking and generation is enabled.	RW	0x0
0	BRK	Send break. If this bit is set to 1, a low-level is continually output on the UARTTXD output, after completing transmission of the current character. For the proper execution of the break command, the software must set this bit for at least two complete frames. For normal use, this bit must be cleared to 0.	RW	0x0

TOPPERS

uart : その他のレジスタの設定値

UARTLCRレジスタに通信有効化設定

- UART、送信、受信を有効化
  - UARTE | TXE | RXEをセット

UART: UARTCR Register

Offset: 0x030

Description

Control Register, UARTCR

Table 444. UARTCR Register

Bits	Name	Description	Type	Reset
31:16	Reserved.	-	-	-
15	CTSEN	CTS hardware flow control enable. If this bit is set to 1, CTS hardware flow control is enabled. Data is only transmitted when the nUARTCTS signal is asserted.	RW	0x0
14	RTSEN	RTS hardware flow control enable. If this bit is set to 1, RTS hardware flow control is enabled. Data is only requested when there is space in the receive FIFO for it to be received.	RW	0x0
13	OUT2	This bit is the complement of the UART Out2 (nUARTOut2) modem status output. That is, when the bit is programmed to a 1, the output is 0. For DTE this can be used as Ring Indicator (RI).	RW	0x0
12	OUT1	This bit is the complement of the UART Out1 (nUARTOut1) modem status output. That is, when the bit is programmed to a 1 the output is 0. For DTE this can be used as Data Carrier Detect (DCD).	RW	0x0
11	RTS	Request to send. This bit is the complement of the UART request to send, nUARTRTS, modem status output. That is, when the bit is programmed to a 1 then nUARTRTS is LOW.	RW	0x0
10	DTR	Data transmit ready. This bit is the complement of the UART data transmit ready, nUARTDTR, modem status output. That is, when the bit is programmed to a 1 then nUARTDTR is LOW.	RW	0x0
9	RXE	Receive enable. If this bit is set to 1, the receive section of the UART is enabled. Data reception occurs for either UART signals or SIR signals depending on the setting of the SIREN bit. When the UART is disabled in the middle of reception, it completes the current character before stopping.	RW	0x1
8	TXE	Transmit enable. If this bit is set to 1, the transmit section of the UART is enabled. Data transmission occurs for either UART signals, or SIR signals depending on the setting of the SIREN bit. When the UART is disabled in the middle of transmission, it completes the current character before stopping.	RW	0x1
7	LBE	Loopback enable. If this bit is set to 1 and the SIREN bit is set to 1 and the SIRTEST bit in the Test Control Register, UARTTCR is set to 1, then the nSIROUT path is inverted, and fed through to the SIRIN path. The SIRTEST bit in the test register must be set to 1 to override the normal half-duplex SIR operation. This must be the requirement for accessing the test registers during normal operation, and SIRTEST must be cleared to 0 when loopback testing is finished. This feature reduces the amount of external coupling required during system test. If this bit is set to 1, and the SIRTEST bit is set to 0, the UARTTXD path is fed through to the UARTRXD path. In either SIR mode or UART mode, when this bit is set, the modem outputs are also fed through to the modem inputs. This bit is cleared to 0 on reset, to disable loopback.	RW	0x0
6:3	Reserved.	-	-	-
2	SIRLP	SIR low-power IrDA mode. This bit selects the IrDA encoding mode. If this bit is cleared to 0, low-level bits are transmitted as an active high pulse with a width of 3 / 16th of the bit period. If this bit is set to 1, low-level bits are transmitted with a pulse width that is 3 times the period of the IrLPBaud16 input signal, regardless of the selected bit rate. Setting this bit uses less power, but might reduce transmission distances.	RW	0x0
1	SIREN	SIR enable: 0 = IrDA SIR ENDEC is disabled. nSIROUT remains LOW (no light pulse generated), and signal transitions on SIRIN have no effect. 1 = IrDA SIR ENDEC is enabled. Data is transmitted and received on nSIROUT and SIRIN. UARTTXD remains HIGH, in the marking state. Signal transitions on UARTRXD or modem status inputs have no effect. This bit has no effect if the UARTEN bit disables the UART.	RW	0x0
0	UARTEN	UART enable: 0 = UART is disabled. If the UART is disabled in the middle of transmission or reception, it completes the current character before stopping. 1 = the UART is enabled. Data transmission and reception occurs for either UART signals or SIR signals depending on the setting of the SIREN bit.	RW	0x0

TOPPERS

uart : その他のレジスタの設定値

UARTDMACRレジスタにDMA有効化設定

- 送信、受信DMAを有効化
  - TXDMAE | RXDMAEをセット

UART: UARTDMACR Register

Offset: 0x048

Description

DMA Control Register, UARTDMACR

Table 450. UARTDMACR Register

Bits	Name	Description	Type	Reset
31:3	Reserved.	-	-	-
2	DMAONERR	DMA on error. If this bit is set to 1, the DMA receive request outputs, UARTRXDMASREQ or UARTRXDMABREQ, are disabled when the UART error interrupt is asserted.	RW	0x0
1	TXDMAE	Transmit DMA enable. If this bit is set to 1, DMA for the transmit FIFO is enabled.	RW	0x0
0	RXDMAE	Receive DMA enable. If this bit is set to 1, DMA for the receive FIFO is enabled.	RW	0x0

TOPPERS

uart : その他のレジスタの設定値

UARTIMSCレジスタに割込み無効化設定

UARTIFLSレジスタのFIFOレベルを0に設定

- 割込み無効化
  - UARTIMSCに0をセット
- FIFOレベルを0
  - UARTIFLSに0をセット

UART: UARTIFLS Register

Offset: 0x034

Description

Interrupt FIFO Level Select Register, UARTIFLS

Table 445. UARTIFLS Register

Bits	Name	Description	Type	Reset
31:6	Reserved.	-	-	-
5:3	RXIFLSEL	Receive interrupt FIFO level select. The trigger points for the receive interrupt are as follows: b000 = Receive FIFO becomes >= 1 / 8 full b001 = Receive FIFO becomes >= 1 / 4 full b010 = Receive FIFO becomes >= 1 / 2 full b011 = Receive FIFO becomes >= 3 / 4 full b100 = Receive FIFO becomes >= 7 / 8 full b101-b111 = reserved.	RW	0x2
2:0	TXIFLSEL	Transmit interrupt FIFO level select. The trigger points for the transmit interrupt are as follows: b000 = Transmit FIFO becomes <= 1 / 8 full b001 = Transmit FIFO becomes <= 1 / 4 full b010 = Transmit FIFO becomes <= 1 / 2 full b011 = Transmit FIFO becomes <= 3 / 4 full b100 = Transmit FIFO becomes <= 7 / 8 full b101-b111 = reserved.	RW	0x2

UART: UARTIFLS Register

Offset: 0x034

Description

Interrupt FIFO Level Select Register, UARTIFLS

Table 445. UARTIFLS Register

Bits	Name	Description	Type	Reset
31:6	Reserved.	-	-	-
5:3	RXIFLSEL	Receive interrupt FIFO level select. The trigger points for the receive interrupt are as follows: b000 = Receive FIFO becomes >= 1 / 8 full b001 = Receive FIFO becomes >= 1 / 4 full b010 = Receive FIFO becomes >= 1 / 2 full b011 = Receive FIFO becomes >= 3 / 4 full b100 = Receive FIFO becomes >= 7 / 8 full b101-b111 = reserved.	RW	0x2
2:0	TXIFLSEL	Transmit interrupt FIFO level select. The trigger points for the transmit interrupt are as follows: b000 = Transmit FIFO becomes <= 1 / 8 full b001 = Transmit FIFO becomes <= 1 / 4 full b010 = Transmit FIFO becomes <= 1 / 2 full b011 = Transmit FIFO becomes <= 3 / 4 full b100 = Transmit FIFO becomes <= 7 / 8 full b101-b111 = reserved.	RW	0x2

uart : マクロ定義1: UART0関連のレジスタ

- 使用するレジスタのアドレス宣言

```
#define TADR_RESETS_RESET      (TADR_RESETS_BASE+TOFF_RESETS_RESET)
#define TADR_RESETS_RESET_DONE  (TADR_RESETS_BASE+TOFF_RESETS_RESET_DONE)
#define TADR_UART0_UARTDR      (TADR_UART0_BASE+TOFF_UART_UARTDR)
#define TADR_UART0_UARTFR      (TADR_UART0_BASE+TOFF_UART_UARTFR)
#define TADR_UART0_UARTIBRD     (TADR_UART0_BASE+TOFF_UART_UARTIBRD)
#define TADR_UART0_UARTFBRD     (TADR_UART0_BASE+TOFF_UART_UARTFBRD)
#define TADR_UART0_UARTLCR_H    (TADR_UART0_BASE+TOFF_UART_UARTLCR_H)
#define TADR_UART0_UARTCR      (TADR_UART0_BASE+TOFF_UART_UARTCR)
#define TADR_UART0_UARTIFLS     (TADR_UART0_BASE+TOFF_UART_UARTIFLS)
#define TADR_UART0_UARTIMSC     (TADR_UART0_BASE+TOFF_UART_UARTIMSC)
#define TADR_UART0_UARTRIS     (TADR_UART0_BASE+TOFF_UART_UARTRIS)
#define TADR_UART0_UARTDMACR    (TADR_UART0_BASE+TOFF_UART_UARTDMACR)
```

uart : マクロ定義2: UART0関連のレジスタ

- 使用するレジスタのレジスタ宣言
- Picoにはアドレスエリアスがあり、エリアスのオフセットで特殊設定が可能
  - TREG_RESETS_RESET_SET : 書き込んだ1のビットのみセット(1)する
  - TREG_RESETS_RESET_CLR : 書き込んだ1のビットのみクリア(0)する
  - TREG_UART0_UARTLCR_HDMY : 0書込みでデータ変化なしの書き込み設定

```
#define TREG_RESETS_RESET_SET   ((volatile uint32_t *)(TADR_RESETS_RESET+REG_ALIAS_SET))
#define TREG_RESETS_RESET_CLR   ((volatile uint32_t *)(TADR_RESETS_RESET+REG_ALIAS_CLR))
#define TREG_RESETS_RESET_DONE  ((volatile uint32_t *)(TADR_RESETS_RESET_DONE))
#define TREG_UART0_UARTDR       ((volatile uint32_t *)(TADR_UART0_UARTDR))
#define TREG_UART0_UARTFR       ((volatile uint32_t *)(TADR_UART0_UARTFR))
#define TREG_UART0_UARTIBRD     ((volatile uint32_t *)(TADR_UART0_UARTIBRD))
#define TREG_UART0_UARTFBRD     ((volatile uint32_t *)(TADR_UART0_UARTFBRD))
#define TREG_UART0_UARTLCR_H    ((volatile uint32_t *)(TADR_UART0_UARTLCR_H))
#define TREG_UART0_UARTLCR_HDMY ((volatile uint32_t *)
                                 (TADR_UART0_UARTLCR_H+REG_ALIAS_SET))
#define TREG_UART0_UARTCR       ((volatile uint32_t *)(TADR_UART0_UARTCR))
#define TREG_UART0_UARTIFLS     ((volatile uint32_t *)(TADR_UART0_UARTIFLS))
#define TREG_UART0_UARTIMSC     ((volatile uint32_t *)(TADR_UART0_UARTIMSC))
#define TREG_UART0_UARTRIS      ((volatile uint32_t *)(TADR_UART0_UARTRIS))
#define TREG_UART0_UARTDMACR    ((volatile uint32_t *)(TADR_UART0_UARTDMACR))
```


TOPPERS

uart : マクロ定義３： レジスタの設定値

- ビット定義はrp2040.h/rp2350.hに定義

```
#define RX0_PIN                 PINPOSITION0
#define TX0_PIN                 PINPOSITION1
#define UART_MODE_MASK (UART_UARTLCR_H_WLEN |
    UART_UARTLCR_H_STP2 | UART_UARTLCR_H_PEN |
    UART_UARTLCR_H_EPS)
#define UART_MODE_VALUE ((8-5) << 5)
```

uart : マクロ定義４： レジスタの設定値

- ビット定義はrp2040.h/rp2350.hの定義を使用する

uart bit	rp2040/rp2350に記載の定義
WLEN	UART_UARTLCR_H_WLEN
STP2	UART_UARTLCR_H_STP2
PEN	UART_UARTLCR_H_PEN
EPS	UART_UARTLCR_H_EPS
FEN	UART_UARTLCR_H_FEN
UARTEN	UART_UARTCR_UARTEN
TXE	UART_UARTCR_TXE
REX	UART_UARTCR_RXE
TXDMAE	UART_UARTDMACR_TXDMAE
RXDMAE	UART_UARTDMACR_RXDMAE

uart:初期化設定関数

- RXピンとTXピン設定、uart0リセット

```
void
uart0_init(void){
    uint32_t clock, reg;
    uint32_t baud_rate_div, baud_ibrd, baud_fbrd;
    uint32_t reset = RESETS_RESET_UART0;
    /*  RXピン設定 */
    TREG_PADS_BANK0_GPIO[RX0_PIN] = 0x00000056;
    TREG_IO_BANK0_GPIO_CTRL[RX0_PIN * 2 + 1] = 0x00000002;
    /* TXピン設定 */
    TREG_PADS_BANK0_GPIO[TX0_PIN] = 0x00000056;
    TREG_IO_BANK0_GPIO_CTRL[TX0_PIN * 2 + 1] = 0x00000002;

    *TREG_RESETS_RESET_SET = reset;
    *TREG_RESETS_RESET_CLR = reset;
    while((*TREG_RESETS_RESET_DONE & reset) == 0);
```


TOPPERS

uart：初期化設定関数

- uart0リセット、クロックを取得、ボーレートdivied値計算と設定

```
if((clock = get_Clock(CLK_PERI)) == 0)
        return;
    baud_rate_div = (8 * clock / DEFAULT_SPEED);
    baud_ibrd = baud_rate_div >> 7;
    if(baud_ibrd == 0){
        baud_ibrd = 1;
        baud_fbrd = 0;
    }
    else if(baud_ibrd >= 65535){
        baud_ibrd = 65535;
        baud_fbrd = 0;
     }
     else
        baud_fbrd = ((baud_rate_div & 0x7f) + 1) / 2;
    /* Baudrate Dividor設定 */
    *TREG_UART0_UARTIBRD = baud_ibrd;
    *TREG_UART0_UARTFBRD = baud_fbrd;
```

ペリフェラルのクロックは
get_Clock関数で取得

DEFAILT_SPEEDは
sys_defs.hで定義

TOPPERS

uart : 初期化関数 uart0_init

- 各制御レジスタを初期化し，送受信を許可する

```
    /* ラインコントロールレジスタにダミー設定 */
    *TREG_UART0_UARTLCR_HDMY = 0;
    /* 通信モード設定/FIFO無効化 */
    reg  = *TREG_UART0_UARTLCR_H & ~(UART_MODE_MASK |
                UART_UARTLCR_H_FEN);
    *TREG_UART0_UARTLCR_H = reg | UART_MODE_VALUE;
    /* 送受信有効化 */
    *TREG_UART0_UARTCR = (UART_UARTCR_UARTEN |
                UART_UARTCR_TXE | UART_UARTCR_RXE);
    /* RX/TX DMA設定 */
    *TREG_UART0_UARTDMACR = (UART_UARTDMACR_TXDMAE |
                UART_UARTDMACR_RXDMAE);
    /* 割込み停止設定  */
    *TREG_UART0_UARTIMSC = 0;
    *TREG_UART0_UARTIFLS = 0;
}
```


TOPPERS

uart : 受信関数

- データを受信すると，受信制御レジスタのRXRISビット（受信完了フラグ）がセットされる
- ポーリングでセットされるのを待ち、受信データを取り込む

```
uint8_t
uart0_pol_getc(void){
   /* 受信バッファにデータが入るのを待つ */
   while((*TREG_UART0_UARTRIS & UART_UARTRIS_RXRIS) == 0);
  /* データ受信 */
  return (uint8_t)*TREG_UART0_UARTDR;
}
```

uart : 送信関数

- 送信データを書き込める状態であれば，受信制御レジスタのUART_UARTFR_TXFF（送信バッファ空フラグ）がセットされる
- ポーリングで空きを待ち、送信データを書き込む

```
void
uart0_pol_putc(unsigned char c){
    /* 送信バッファが空くのを待つ */
    while((*TREG_UART0_UARTFR & UART_UARTFR_TXFF) != 0);
    /* データ送信 */
   *TREG_UART0_UARTDR = c;
}
```

uart : メイン関数

- 受信関数でデータを受信後，LEDをカウントアップし，送信関数で受信したデータを送信する

```
void
main(void){
    uint32_t count = 0;
    uint8_t c;

    led_init();
    uart0_init();

    for (;;) {
        c = uart0_pol_getc();
        led_out_bdigit(count++);
        uart0_pol_putc(c);
    }
}
```

データ受信

LEDカウントアップ

データ送信

TOPPERS

割込みプログラミング

1. Cortex-Mの割込みアーキテクチャ
2. RP2350:RISC-Vの割込みアーキテクチャ
3. スイッチ割込みプログラム
4. 割込みタイマ

目的

割込みを用いたプログラミングを学ぶ

- 割込みによる事象待ちのために必要な知識
  - 割込みハンドラ
  - 割込みベクタテーブル
  - 割込み関連のレジスタ
  - 割込み禁止・許可
  - 割込み優先度

- プログラム対象の周辺デバイス
  - プッシュスイッチ
  - タイマ
  - シリアルI/O

Cortex-Mの割込みアーキテクチャ

Cortex-Mはネスト型割込みコントローラ(NVIC)を標準搭載

- NVICは最大240の割込みをサポート
- 各割込みには最大256レベルの優先度を動的に設定可能
- NVICへの完全なアクセスは特権モードから可能
- レベル割込みとパルス割込み
  - レベル割込みは、デバイスにアクセスするISRによってクリアさせるまで、アサートされ続ける
  - パルス割込みが、エッジモデルの一種で、パルスのサンプリングは非同期ではなく、Cortex-MのクロックFCLKの立ち上がりでエッジされることを保証する必要がある

NVICレジスタ

NVICレジスタ設定に割込みの有効無効、割込み優先度を設定する

- NVIC_ISER割込みの有効化
- NVIC_ICER割込みの無効化
- NVIC_IPR割込み優先度設定

アドレス	名前	タイプ	リセット時の値	説明
0xE000E004	ICTR	RO	-	*「割り込みコントローラタイプレジスタ、ICTR」*
0xE000E100～0xE000E11C	NVIC_ISER0～NVIC_ISER7	RW	0x00000000	割り込みイネーブルセット レジスタ
0xE000E180～0xE000E19C	NVIC_ICER0～NVIC_ICER7	RW	0x00000000	割り込みイネーブルクリア レジスタ
0xE000E200～0xE000E21C	NVIC_ISPR0～NVIC_ISPR7	RW	0x00000000	割り込み保留セットレジスタ
0xE000E280～0xE000E29C	NVIC_ICPR0～NVIC_ICPR7	RW	0x00000000	割り込み保留クリアレジスタ
0xE000E300～0xE000E31C	NVIC_IABR0～NVIC_IABR7	RO	0x00000000	割り込みアクティブビット レジスタ
0xE000E400～0xE000E4EC	NVIC_IPR0～NVIC_IPR59	RW	0x00000000	割り込み優先度レジスタ

TOPPERS

Cortex-Mの割込みアーキテクチャ

- IRQ割込みはstartup_rp2040.S/startup_rp2350.Sにベクタテーブルがすでに定義されている

```
.section  .isr_vector,"a",%progbits
 .type  g_pfnVectors, %object
 .size  g_pfnVectors, .-g_pfnVectors
g_pfnVectors:
.word  _estack
.word  Reset_Handler
.word  NMI_Handler
.word  HardFault_Handler
 :
.word  SysTick_Handler
.word  Timer0_IRQHandler            /* 0  -  TIMER_IRQ_0  */
.word  Timer1_IRQHandler            /* 1  -  TIMER_IRQ_1  */
.word  Timer2_IRQHandler            /* 2  -  TIMER_IRQ_2  */
.word  Timer3_IRQHandler            /* 3  -  TIMER_IRQ_3  */
.word  PWM_WRAP_IRQHandler     /* 4  -  PWM_IRQ_WRAP */
 :
.word  IO_BANK0_IRQHandler      /* 13  -  IO_IRQ_BANK0 */
```

Cortex-Mの割込みアーキテクチャ

- IO_BANK0用の割込みハンドラ（IO_BANK0_IRQHandler）はstartup_rp2040.S内で.weak設定で定義されており、Default_Handlerにリンクしている
- この定義は仮定義で、globalにIO_BANK0_Handler関数をリンクすれば、その関数が正しい関数として扱われる

```
.weak     PIO_1_WKUP_IRQHandler
.thumb_set PIO_1_IRQHandler,Default_Handler

.weak     DMA0_IRQHandler
.thumb_set DMA0_IRQHandler,Default_Handler

.weak     DMA1_IRQHandler
.thumb_set DMA1_IRQHandler,Default_Handler

.weak     IO_BANK0_IRQHandler
.thumb_set IO_BANK0_IRQHandler,Default_Handler
```

割込みプログラミング

1. Cortex-Mの割込みアーキテクチャ
2. RP2350:RISC-Vの割込みアーキテクチャ
3. スイッチ割込みプログラム
4. 割込みタイマ

RISC-V:hazard3の割込みアーキテクチャ

- rp2350:RISC-Vの割込み制御はNVICと同じ制御手順ですが、設定方法はCSR制御レジスタを用いた設定となっている
- 割込み有効、無効と優先度設定は4つのCSRを使用する
- 設定領域は16ビット単位で設定する
  - 上位16ビットが設定値
  - 下位16ビットがオフセット

CSR番号	番号	機能
RVCSR_MEIEA	0x0BE0	割込みを有効にする
RVCSR_MEIPA	0x0BE1	割込みペンディング
RVCSR_MEIFA	0x0BE2	割込みフォースペンディング
RVCSR_MEIPRA	0x0BE3	割込み優先度を設定

RISC-V:hazard3の割込みアーキテクチャ

- rp2350:RISC-Vの割込み制御はNVICと同じような制御手順ですが、設定方法はCSR制御レジスタを用いた設定と

```
#define hazard3_irqarray_set(csr, index, data) (set_csr(csr, (index) | ((uint32_t)(data) << 16)))
#define hazard3_irqarray_clear(csr, index, data) (clear_csr(csr, (index) | ((uint32_t)(data) << 16)))

void hazard3_enable_interrupt(uint8_t source)
{
    hazard3_irqarray_clear(RVCSR_MEIFA, (source / 16), (1 << (source % 16)));
    hazard3_irqarray_set(RVCSR_MEIEA, (source / 16), (1 << (source % 16)));
}
uint32_t hazard3_disable_interrupt(uint8_t source)
{
    return hazard3_irqarray_clear(RVCSR_MEIEA, (source / 16), (1 << (source % 16)));
}
void hazard3_set_priority(uint8_t source, uint8_t priority)
{
    hazard3_irqarray_clear(RVCSR_MEIPRA, source / 4, 0xFU << (4 * (source % 4)));
    hazard3_irqarray_set(RVCSR_MEIPRA, source / 4, (priority & 0xFU) << (4 * (source % 4)));
}
```

割込み許可

割込み禁止

優先度設定

TOPPERS

RISC-V:CSR制御レジスタ設定

- 標準的なRISC-Vプロセッサでは、encoding.hインクルードファイル供給されプロセッサで使用可能なCSRレジスタとC言語での設定マクロが記載され、設定や読み出しが可能となっている

C言語マクロ	アセンブラ記述	機能
read_csr(#num, val)	cssr a0, #num	
write_csr(#num, val)	csrw #num, a5	
set_csr(#reg, bit)	csrrs a0, #num, a5	割込みペンディング
clear_csr(#reg, bit)	csrrc a0, #num, a5	割込みフォースペンディング

valやbitの値が32未満の場合、レジスタを使わないアセンブル記述がある

割込みプログラミング

1. Cortex-Mの割込みアーキテクチャ
2. RP2350:RISC-Vの割込みアーキテクチャ
3. スイッチ割込みプログラム
4. 割込みタイマ

スイッチ割込みプログラム1：int_switch_1 概要

- プッシュスイッチを押すと発生する割込み（IO_BANK0）により，LEDをカウントアップする

- 学習内容
  - 割り込みプログラムの全体像の理解
  - 割込みベクタテーブル
  - 割込み制御レジスタ
  - 割込みハンドラ

- 動作確認
  - switch_pushをコピーして、ディレクトリ名をint_switch_1に修正して作成する

int_switch_1 : 割込みプログラムの全体像

- 割込みをプログラムの視点から見ると
  - メイン関数を実行中に割込みが入ると、一旦メイン関数中の処理が中断され，割込みに対応した関数（割込みハンドラ）が実行される
  - 割込みハンドラの実行が終了すると，メイン関数の処理が再開される

- プログラムで必要となる処理
  - 割込み要因ハードウェアの初期化
  - 割込みハンドラの用意
  - 割込みベクタテーブルの用意
  - 割込み関連の初期化

int_switch_1 : 割込み要因ハードウェア

プッシュスイッチ（SW1）を割込み要因として用いる

- 接続
  - GP10ピンにGPIO入力モードで接続
  - ボタンを押すとゼロ(LOW)が入力される
- 割込み番号、優先度（NVIC_IPRを参照）
  - 割込み番号INT_VECTOR_IO_BANK0：21番に設定済
  - 割込みハンドラ、レベルを設定
- 外部割込みのモード設定
  - IO_BANK0_PROC0_INTEレジスタに立下りエッジに設定する
    - キーを押し、離した時点で割込みが発生

int_switch_1 : 割込み要因ハードウェアの初期化

ポート関連のレジスタと
割込み制御関連のレジスタの初期化が必要

- ポート関連レジスタの初期化
  - ポーリングプログラミング switch_push の初期化関数を流用

- 割込み制御関連のレジスタ
  - 割込み制御レジスタ
    - 割込みレベルや割込みエッジの設定
  - 割込み要因選択レジスタ
    - 割込み極性切り替え（両エッジか片エッジか）

int_switch_1 : IO_BANK0_PROC0_INTE設定

- GPIOピン1つに付き、4つの割込み設定が可能
  - EDGE_HIGH
  - EDGE_LOW
  - LEVEL_HIGH
  - LEVEL_LOW
- GP10のEDGE_LOWビットをオンにする

IO_BANK0: PROC0_INTE1 Register

Offset: 0x104

Description

Interrupt Enable for proc0

Table 302. PROC0_INTE1 Register

Bits	Name	Description	Type	Reset
31	GPIO15_EDGE_HIGH		RW	0x0
30	GPIO15_EDGE_LOW		RW	0x0
29	GPIO15_LEVEL_HIGH		RW	0x0
28	GPIO15_LEVEL_LOW		RW	0x0
27	GPIO14_EDGE_HIGH		RW	0x0
26	GPIO14_EDGE_LOW		RW	0x0
25	GPIO14_LEVEL_HIGH		RW	0x0
24	GPIO14_LEVEL_LOW		RW	0x0
23	GPIO13_EDGE_HIGH		RW	0x0
22	GPIO13_EDGE_LOW		RW	0x0
21	GPIO13_LEVEL_HIGH		RW	0x0
20	GPIO13_LEVEL_LOW		RW	0x0
19	GPIO12_EDGE_HIGH		RW	0x0
18	GPIO12_EDGE_LOW		RW	0x0
17	GPIO12_LEVEL_HIGH		RW	0x0
16	GPIO12_LEVEL_LOW		RW	0x0
15	GPIO11_EDGE_HIGH		RW	0x0
14	GPIO11_EDGE_LOW		RW	0x0
13	GPIO11_LEVEL_HIGH		RW	0x0
12	GPIO11_LEVEL_LOW		RW	0x0
11	GPIO10_EDGE_HIGH		RW	0x0
10	GPIO10_EDGE_LOW		RW	0x0
9	GPIO10_LEVEL_HIGH		RW	0x0
8	GPIO10_LEVEL_LOW		RW	0x0
7	GPIO9_EDGE_HIGH		RW	0x0
6	GPIO9_EDGE_LOW		RW	0x0
5	GPIO9_LEVEL_HIGH		RW	0x0
4	GPIO9_LEVEL_LOW		RW	0x0
3	GPIO8_EDGE_HIGH		RW	0x0
2	GPIO8_EDGE_LOW		RW	0x0
1	GPIO8_LEVEL_HIGH		RW	0x0
0	GPIO8_LEVEL_LOW		RW	0x0

TOPPERS

int_switch_1 : NVIC制御関連のレジスタの定義

- 割込み制御レジスタのアドレス定義
  - アドレスはrp2040.h/rp2350.hで定義

```
#define TADR_IO_BANK0_INTR     (TADR_IO_BANK0_BASE+TOFF_IO_BANK0_INTR)
#define TADR_IO_BANK0_PROC0_INTE  (TADR_IO_BANK0_BASE
                          +TOFF_IO_BANK0_PROC0_INTE)
#define TADR_IO_BANK0_PROC0_INTS  (TADR_IO_BANK0_BASE
                          +TOFF_IO_BANK0_PROC0_INTS)
#if BOARDNO == 0
#define TADR_SCB_AIRCR        (TADR_PPB_BASE+TOFF_M0PLUS_AIRCR)
#define TADR_NVIC_ISER        (TADR_PPB_BASE+TOFF_M0PLUS_NVIC_ISER)
#define TADR_NVIC_ICER        (TADR_PPB_BASE+TOFF_M0PLUS_NVIC_ICER)
#define TADR_NVIC_IP          (TADR_PPB_BASE+TOFF_M0PLUS_NVIC_IPR)
#else
#define TADR_SCB_AIRCR        (TADR_PPB_BASE+TOFF_M33_AIRCR)
#define TADR_NVIC_ISER        (TADR_PPB_BASE+TOFF_M33_NVIC_ISER)
#define TADR_NVIC_ICER        (TADR_PPB_BASE+TOFF_M33_NVIC_ICER)
#define TADR_NVIC_IP          (TADR_PPB_BASE+TOFF_M33_NVIC_IPR)
#endif
```

int_switch_1 : NVIC制御関連のレジスタの定義

- 割込み制御レジスタを定義

```
#define TREG_IO_BANK0_INTR      ((volatile uint32_t *)(TADR_IO_BANK0_INTR))
#define TREG_IO_BANK0_PROC0_INTE
                                ((volatile uint32_t *)(TADR_IO_BANK0_PROC0_INTE))
#define TREG_IO_BANK0_PROC0_INTS
                                ((volatile uint32_t *)(TADR_IO_BANK0_PROC0_INTS))
#define TREG_SCB_AIRCR          ((volatile uint32_t *)(TADR_SCB_AIRCR))
#define TREG_NVIC_ISER          ((volatile uint32_t *)(TADR_NVIC_ISER))
                                                    /* NVIC INT SET ENABLEレジスタ */
#define TREG_NVIC_ICER          ((volatile uint32_t *)(TADR_NVIC_ISER))
                                                    /* NVIC INT CLR ENABLEレジスタ */
#define TREG_NVIC_IP            ((volatile uint32_t *)(TADR_NVIC_IP))
                                                    /* NVIC IPレジスタ */
```

int_switch_1：Cortex-Mの割込み制御関数

```
static inline uint8_t
get_machine_priority(uint8_t lvl){
    uint8_t spri = 0x0F;
    uint8_t tmppriority = 0x00, tmpsub = 0x0F, tmpgroup;

    tmpgroup = (*TREG_SCB_AIRCR & 0x700) >> 8;
    tmpsub = (tmpgroup + TBITW_IPRI) < 7 ? 0 : (tmpgroup - 7) + TBITW_IPRI;
    tmppriority = lvl << tmpsub;
    tmppriority |=  (spri & ((1 << tmpsub)-1));
    tmppriority = tmppriority << (8 - TBITW_IPRI);
    return tmppriority;
}
static inline void
enable_interrupt(uint8_t no){
    TREG_NVIC_ISER[no >> 0x05] = 0x01 << (no & 0x1F);
}
static inline void
disable_interrupt(uint8_t no){
    TREG_NVIC_ICER[no >> 0x05] = 0x01 << (no & 0x1F);
}
static inline void
set_priority(uint8_t no, uint8_t priority){
    uint32_t val = TREG_NVIC_IP[no>>2];
    val &= ~(0xFF << (8 * (no & 0x03)));
    val |= (get_machine_priority(priority) << (8 * (no & 0x03)));
    TREG_NVIC_IP[no>>2] = val;
}
```

割込み許可

割込み禁止

優先度設定

TOPPERS

int_switch_1：RISC-Vの割込み制御関数

```
/*
 *  RISC-V割込み設定関数
 */
#define hazard3_irqarray_set(csr, index, data) (set_csr(csr, (index) | ((uint32_t)(data) << 16)))
#define hazard3_irqarray_clear(csr, index, data) (clear_csr(csr, (index) | ((uint32_t)(data) << 16)))

extern void hazard3_set_vector(uint8_t no, function_ptr_t func);
void IO_BANK0_IRQHandler(void);

static inline void
enable_interrupt(uint8_t no){
    hazard3_irqarray_clear(RVCSR_MEIFA, (no / 16), (1 << (no % 16)));
    hazard3_irqarray_set(RVCSR_MEIEA, (no / 16), (1 << (no % 16)));
}
static inline void
disable_interrupt(uint8_t no){
    hazard3_irqarray_clear(RVCSR_MEIEA, (no / 16), (1 << (no % 16)));
}
static inline void
set_priority(uint8_t no, uint8_t priority){
    hazard3_irqarray_clear(RVCSR_MEIPRA, no / 4, 0xFU << (4 * (no % 4)));
    hazard3_irqarray_set(RVCSR_MEIPRA, no / 4, (no & 0xFU) << (4 * (no % 4)));
}
```

割込み許可

割込み禁止

優先度設定

TOPPERS

int_switch_1 : 割込み制御関連のレジスタの初期化

- 割込み設定を行うinit_int関数を作成する

```
/*
 * 割込み初期化
 */
void
init_int(uint8_t no, uint8_t mpri, bool_t active){

    if(active){
        set_priority(no, mpri);
        enable_interrupt(no);
    }
    else{
        disable_interrupt(no);
    }
}
```


TOPPERS

int_switch_1：SW1用の割込み設定

- ポーリング型のSWITCH初期化関数を割込み型に改造
- プッシュボタンを押した際に割込みが入るよう立ち下がりエッジを選択

```
void
switch_push_init(uint8_t no, uint8_t lvl){
    uint8_t  shift = (PSW1_PINNO & 0x7) * 4;
    uint32_t reg;
    /* PDS_BANK0_GPIO設定 */
    TREG_PADS_BANK0_GPIO[PSW1_PINNO] = 0x00000052;
    /* IO_BANK0_CTROL_GPIO設定 */
    TREG_IO_BANK0_GPIO_CTRL[PSW1_PINNO * 2 + 1] = 0x00000005;
    /* SIO_GPIO_OE設定 */
    *TREG_SIO_GPIO_OE_CLR = 1 << PSW1_PINNO;

    reg = TREG_IO_BANK0_PROC0_INTE[PSW1_PINNO>>3] & ~(0xF << shift);
    TREG_IO_BANK0_PROC0_INTE[PSW1_PINNO>>3]
                = reg | (IO_BANK0_INTR0_GPIO_EDGE_LOW << shift);

#ifdef __riscv
    hazard3_set_vector(no, IO_BANK0_IRQHandler);
#endif
   init_int(no, lvl, true);
}
```

引数に割込み条件を追加する

フォーリング・エッジ

RISC-Vでは割込みハンドラを登録しなければならない

割込み許可

TOPPERS

int_switch_1 : 割込みハンドラ

- グローバル変数をカウントアップしてLEDに出力

```
void
IO_BANK0_IRQHandler(void){
    static uint32_t led_count = 0;
    uint32_t ine = TREG_IO_BANK0_PROC0_INTE[PSW1_PINNO >> 3];
    uint32_t ins = TREG_IO_BANK0_PROC0_INTS[PSW1_PINNO >> 3];
    uint32_t emask = 0xF << ((PSW1_PINNO & 7) * 4);
    uint32_t event;

    event = ine & ins;
    if(event == 0)
        return;
    if((event & emask) != 0){
        TREG_IO_BANK0_INTR[PSW1_PINNO>>3] = (event & emask);
        /* 表示更新  */
        led_out_bdigit(++led_count);
    }
}
```

int_switch_1 : メインルーチン

- 初期化後，割込みを許可して無限ループで割込みを待つ

```
/*
 * main関数
 */
void
main(void){
    /* ハードウェア初期化 */
    led_init();
    switch_push_init(INT_VECTOR_IO_BANK0, 3);

    for (;;) {
    }
}
```

RISC-Vでは割込みハンドラを登録しなければならない

何も行わない無限ループ

TOPPERS

スイッチ割込みプログラム2：int_switch_2 概要

- メイン関数でLEDを一定周期でカウントアップし，SW1割込みが発生するとカウント値をクリアする

- 学習内容
  - メインルーチンと割込みハンドラ間でのデータの共有
  - 割込み禁止の必要性（クリティカルセクション）

- 動作確認
  - 作成済みのプログラムを実行し動作を確認する

int_switch_2 : データ共有と割込みハンドラ

メインルーチンと割込みハンドラ間でのデータの共有は，
グローバル変数（共有変数）で実現する

- LEDのカウントデータはグローバル変数とする

```
uint32_t led_count;
```

- 割込みハンドラではカウントデータをクリアする

```
void
IO_BANK0_IRQHandler(void){
    uint32_t ine = TREG_IO_BANK0_PROC0_INTE[PSW1_PINNO >> 3];
    uint32_t ins = TREG_IO_BANK0_PROC0_INTS[PSW1_PINNO >> 3];
    uint32_t emask = 0xF << ((PSW1_PINNO & 7) * 4);
    uint32_t event;
    event = ine & ins;
    if(event == 0)
        return;
    if((event & emask) != 0){
        TREG_IO_BANK0_INTR[PSW1_PINNO>>3] = (event & emask);
        /* LEDクリア */
        led_count = 0;
        /* 表示更新  */
        led_out_bdigit(led_count);
    }
}
```

外部割込み要因のクリア

データのクリア

TOPPERS

int_switch_2 : メイン関数

- led_dataのカウントアップは外部関数で行う
- 外部関数の戻り値（インクリメントした値）を led_data に入れる

```
void
main(void){
    led_init();
    switch_push_init(INT_VECTOR_IO_BANK0, 3);
    led_count = 0;

    for (;;) {
        led_count = inc_data(led_count);
        led_out_bdigit(led_count);
        busy_wait();
    }
}
```

LEDのカウントアップ

TOPPERS

int_switch_2 : カウントアップ関数

- 引数の値をインクリメントして返す
- busy_wait()により故意に時間を消費している
  - 関数の実行によりある一定の処理時間が必要であるという想定を模している
  - 外部関数がライブリラリで提供されていると処理時間は分からない場合がある
  - busy_wait()は以前のプログラムの2倍の時間待つ

```
uint32_t
inc_data(unsigned char data){
    busy_wait();
    busy_wait();
    return (data + 1);
}
```

ダミーウェイト

TOPPERS

int_switch_2 : プログラムの問題点

前述のプログラムには問題があり正しく動作しない

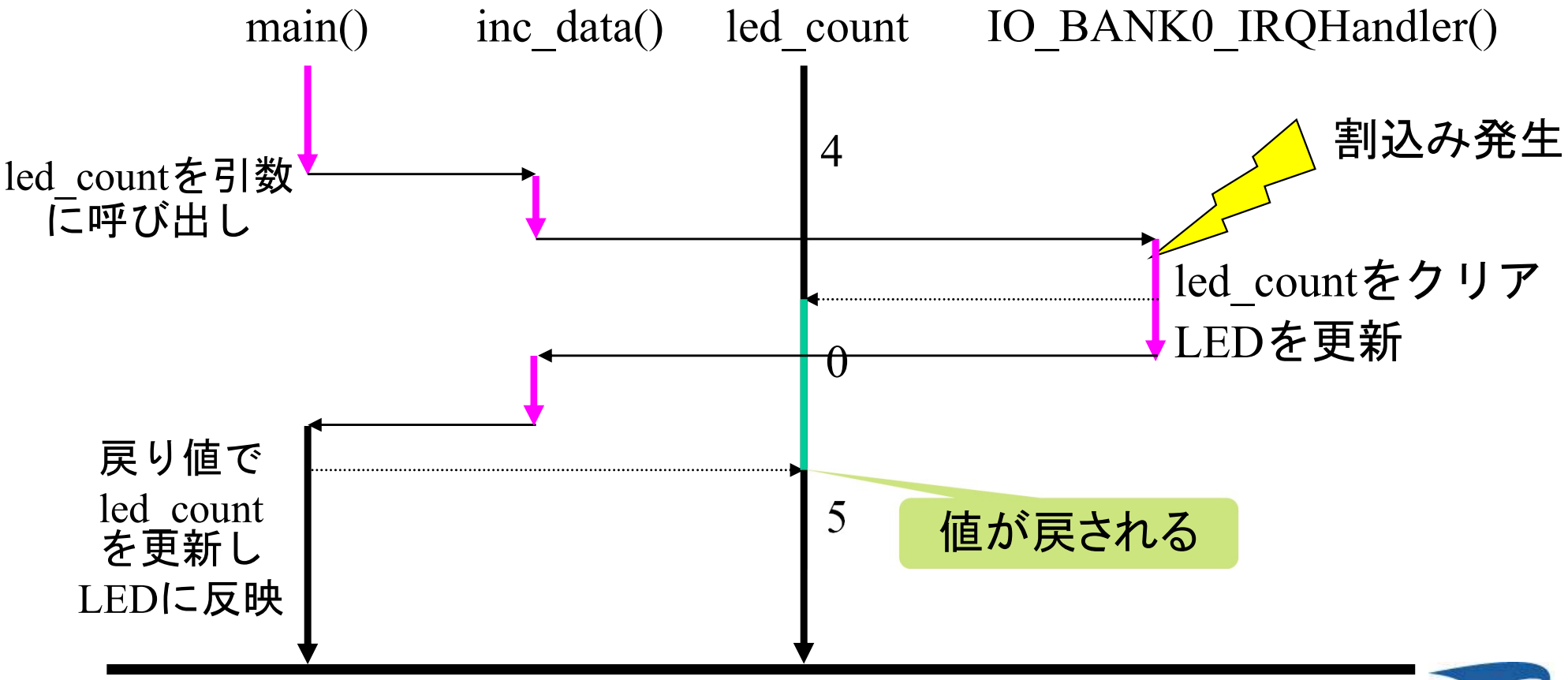
- SW1を押して割込みハンドラでLEDをクリアしても，次のカウントのタイミングで’1’ではなく，前のカウント値をインクリメントした値が表示される

実際に動作させて問題点を確認する

- メイン関数ではグローバル変数 led_data を引数に inc_data() を呼び出し，戻り値を led_data に戻している
  - ➡ グローバル変数の値を読み込んでから書き戻すまでに時間が必要

int_switch_2 : led_countの更新

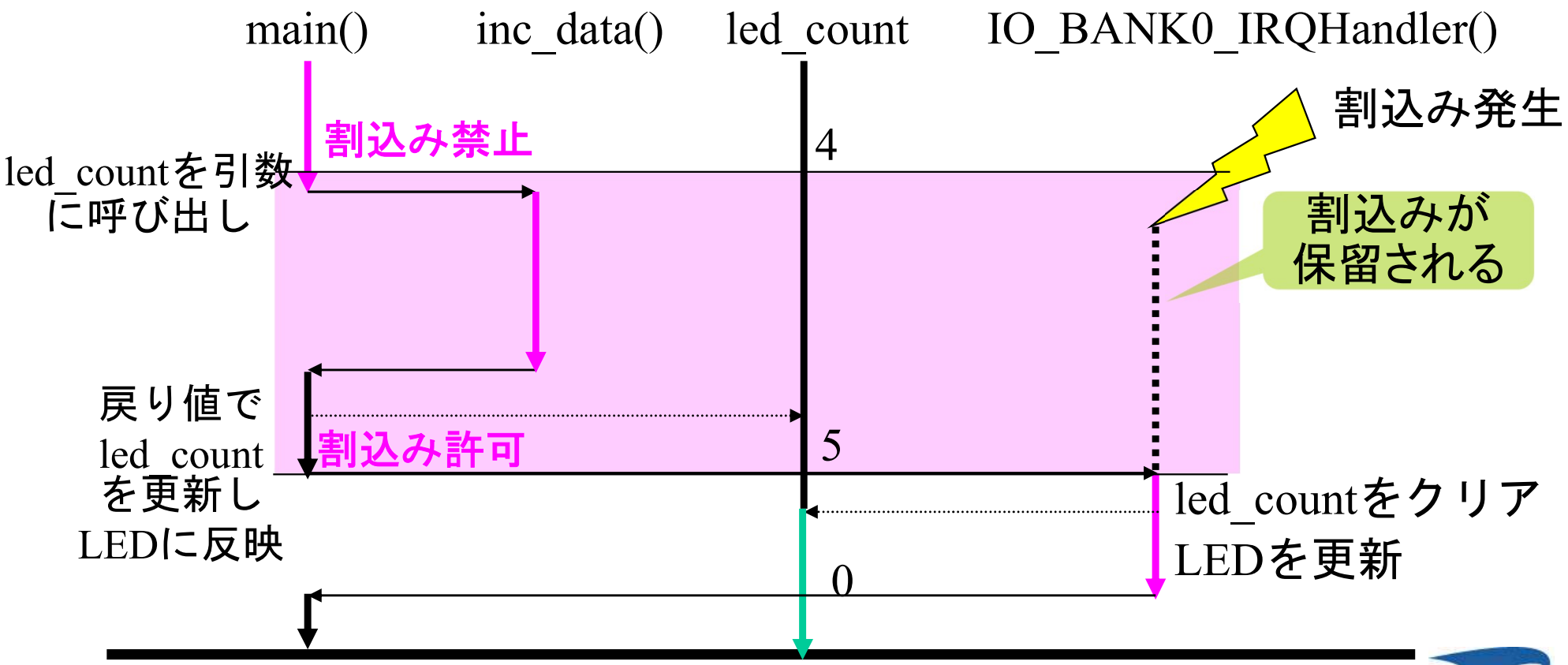
- main()関数でled_countを読み込んでinc_data()を実行している間に，割込みが発生し，割込みハンドラでled_countをクリアすると，その後main()関数によって上書きされてしまう



int_switch_2 : 割込みの禁止

- 共有データ(led_count)を排他的に扱えば問題は発生しない
- main()関数でled_countを更新する間はSW1の割込みがプロセッサによって受け付けられないようにすればよい

➡ 割込み禁止により実現



TOPPERS

int_switch_2：割込み許可禁止設定

- 割込み許可禁止関数を追加
  - Enable(); 割込み許可
  - Disable(); 割込み禁止

```
void Enable(void)
{
#ifndef __riscv
    asm volatile ("cpsie i");
#else
    set_csr(mstatus, MSTATUS_MIE);
#endif
}
void Disable(void)
{
#ifndef __riscv
    asm volatile ("cpsid i");
#else
    clear_csr(mstatus, MSTATUS_MIE);
#endif
}
```

Cortex-M（上段）、RISC-V（下段）のどちらかを記載すればよい

int_switch_2 : 割込みの禁止 プログラムでの実現

- 割込み禁止・許可命令で実現する
  - プロセッサ毎に実現方法は異なる
  - 割込みマスクで実現する方法も（後述）

割込み禁止

```
void
main(void){
    ...省略...
    for (;;) {
        Disable();
        led_count = inc_data(led_count);
        led_out_bdigit(led_count);
        Enable();
        busy_wait();
    }
}
```

割込みが発生しても受け付けられず，割込みハンドラと排他的に実行される

TOPPERS

int_switch_2 : クリティカルセクション

排他的に動作する（しなければならない）区間

- 実現方法
  - メイン関数と割込みハンドラ間や割込みハンドラ間は，割込み禁止・許可命令を組み合わせて実現
- クリティカルセクションの実行時間
  - 長くなると，割込みが発生してから応答するまでの時間（割り込み応答時間）が長くなり，リアルタイムシステムでは問題となる場合がある
  - 修正したプログラムでは，スイッチを押してからクリアされるまでのタイムラグが発生する

割込みプログラミング

1. Cortex-Mの割込みアーキテクチャ
2. RP2350:RISC-Vの割込みアーキテクチャ
3. スイッチ割込みプログラム
4. 割込みタイマ

タイマ割込みプログラム : int_timer 概要

- タイマ4の割込みを用いて1秒間隔でLEDをカウントアップする

- 学習内容
  - タイマ割込みの扱い方

- 作成方法
  - ポーリングプログラム timer をコピーして、ディレクトリ名をint_timerに修正して作成する

int_timer : 作成

プログラム int_timer を作成する

- 手順
  - 割込み要因ハードウエアの初期化（タイマ0）
    - カウントソース，カウント値を適切に設定してスタートさせる
  - 割込み制御レジスタの初期化
    - 割込み優先度の設定
  - 割込みハンドラ
    - LEDをカウントアップする

int_timer: init_int関数の取り込み

- int_switch_1で作成したinit_init関数をint_timer.cに取り込む
- PicoとPico2では、timer0の割込み番号と割込み関数名が違うため#ifndefを追加する
  - Pico INT_VECTOR_TIMER0 Timer0_IRQHandler
  - Pico2 INT_VECTOR_TIMER0_0 Timer0_0_IRQHandler

```
#ifndef INT_VECTOR_TIMER0
#define INT_VECTOR_TIMER0   INT_VECTOR_TIMER0_0
#define Timer0_IRQHandler         Timer0_0_IRQHandler
#endif
```

int_timer: NVIC制御関連のレジスタの定義

- 割込み制御レジスタ

```
#define TADR_TIMER_ALARM    (TADR_TIMER_BASE+TOFF_TIMER_ALARM0)
#define TADR_TIMER_TIMERAWL  (TADR_TIMER_BASE+TOFF_TIMER_TIMERAWL)
#define TADR_TIMER_INTR        (TADR_TIMER_BASE+TOFF_TIMER_INTR)
#define TADR_TIMER_INTE        (TADR_TIMER_BASE+TOFF_TIMER_INTE)
#define TADR_TIMER_INTS        (TADR_TIMER_BASE+TOFF_TIMER_INTS)

#define TREG_TIMER_ALARM
            ((volatile uint32_t *)(TADR_TIMER_ALARM+ALARMNO*4))
#define TREG_TIMER_TIMERAWL     ((volatile uint32_t *)(TADR_TIMER_TIMERAWL))
#define TREG_TIMER_INTR        ((volatile uint32_t *)(TADR_TIMER_INTR))
#define TREG_TIMER_INTE        ((volatile uint32_t *)(TADR_TIMER_INTE))
#define TREG_TIMER_INTS        ((volatile uint32_t *)(TADR_TIMER_INTS))
```

int_timer: NVIC制御関連のレジスタの定義

- timr0中のアラーム番号を設定する
- 割込みハンドラのプロトタイプ宣言を追加する

```
#define ALARMNO                 0

void Timer0_IRQHandler(void);
```

- 変数led_countを追加する

```
uint32_t timeup_value, led_count;
```

int_timer : TIMER4ハードウェアの初期化

- オートリロード設定を行えば，一度スタートさせれば，設定周期で割込みが発生する

```
void
timer_init(uint8_t no, uint8_t lvl){
    uint32_t inte = *TREG_TIMER_INTE;

    *TREG_TIMER_INTE = inte | (1<<ALARMNO);
    timeup_value = *TREG_TIMER_TIMERAWL + DELAY_LOOP;
    *TREG_TIMER_ALARM = timeup_value;

    /*
     *  割込みコントローラ設定
     */
#ifdef __riscv
    hazard3_set_vector(no, Timer0_IRQHandler);
#endif
    init_int(no, lvl, true);
}
```

ARARM0の割込み設定

割込み発生カウントを設定

TOPPERS

int_timer : 割込みハンドラ

- 呼び出されると，LEDをカウントアップ

```
void
Timer0_IRQHandler(void){
    static uint32_t led_count = 0;
    uint32_t ints = *TREG_TIMER_INTS;

    /*
     * 割込みクリア
     */
    *TREG_TIMER_INTR = ints;

    /*
     * リロード時間設定
     */
    timeup_value += DELAY_LOOP;
    *TREG_TIMER_ALARM = timeup_value;
    sys_printf("COUNT(%d)¥n", led_count);
    led_out_bdigit(led_count++);
}
```

int_timer：MAIN関数

- メイン関数ではled_countをグローバル変数に修正を行った対応

```
void
main(void){
    /* ハードウェア初期化 */
    led_init();
    timer_init(INT_VECTOR_TIMER0+ALARMNO, 3);
    for (;;) {
    }
}
```

シリアルI/Oの割込み対応

- Appendexとしてデバイスの割込み演習を用意しました
  - シリアルI/Oを割込みで動作させるint_uart
- program/int_uartを参考にしてください

まとめ

小規模組込み開発のまとめ

- 小規模な組込み開発では少数の開発者でファームウェア、ソフトウェアの開発を行うため、ハードウェアの知識や開発環境を構築する知識も必要となる
- 開発言語はC言語を用いることが多いが、C言語で記述できない部分は、アセンブラ言語で開発する必要がある
- 開発環境はプロセッサメーカーの開発環境を用いることが多く、開発前にノウハウの蓄積が必要
- ハードウェアの制御は、ポーリングを用いるのもの、割込みを用いて制御を行うものがあり、適切な方式を選択する必要がある
- 割込み制御はプロセッサ毎に異なる。特別な設定を行う必要があり、処理を熟知して開発を行う必要がある

小規模組込み開発のまとめ

- アプリケーション部分の開発については、オブジェクト指向やモデル設計を用いれば、可視的なソフトウェア開発ができる。
- このような開発は実行ROM、RAMサイズの増加を伴うことが多いので注意が必要
- C言語で記載すれば、アプリケーションレベルのプログラムは、ほぼない
- NPO法人　組込みソフトウェア管理者・技術者育成研究会では、いろいろは組込みソフトウェア開発手法の紹介を行ってますので、組込み開発プロセスの参考にしてください。http://www.sessame.jp/

講座終了の確認

- my_programには、以下のディレクトリができ正しくプログラムが実行されているはずです

実習済プログラム	
int_switch_1	: 割込みによるプッシュスイッチプログラム
int_switch_2	: 割込みプッシュスイッチクリティカルセクション検証
int_timer	: タイマー割込みLEDカウントプログラム
led_count	: ソフトウェアタイマーLEDカウントプログラム
led_shift	: LEDシフトプログラム（コピー）
sample	: サンプルプログラム（コピー）
switch_push	: ポーリングによるスイッチ検知
timer	: ハードウェアタイマーLEDカウントプログラム
uart	: ポーリング方式UART通信

謝辞

本教材の開発において、NPO法人組込みソフトウェア管理者・技術者育成研究会およびNPO法人TOPPERSプロジェクトの作成した以下の教材を参考としました.

- OpenSESSAME Seminar組込みソフトウェア技術者・管理者向けセミナー初級者向けテキスト第5版
- TOPPERS初級実装セミナー教材

両団体および上記教材の作者の皆様へ感謝します.

本教材の開発において、NEXCESS初級コースおよび指導者養成コースの受講者の皆様のコメントを参考としました.
両コースの受講者の皆様へ感謝します.

著者リスト

- メモリマップドレジスタの操作方法の確認
  - 本田 晋也（名古屋大学）
- ポーリングプログラミング
  - 本田 晋也（名古屋大学）
- 割込みプログラミング
  - 本田 晋也（名古屋大学）
- Pico/Pico2用のプログラム作成、テキストの修正
  - 竹内良輔