Advanced

ラインレースカーによる開発実習

マイコン応用編

本テキストを使用する為に必要なモノ

- 実習機器
  - パソコン
  - PK-LTC
  - ドック
  - 応用編の部品一式
  - 単三電池×２
  - オーバルコース(A3サイズ)

推奨する動作環境

- OS　　: Windows7/8/8.1/10
- CPU　　: Intel Core i5以上(2.0GHz以上を推奨)
- RAM　　: 4GB以上
- I/F　　: USBポート(3個以上)
- 画面　　: HD1080(1920×1080)以上

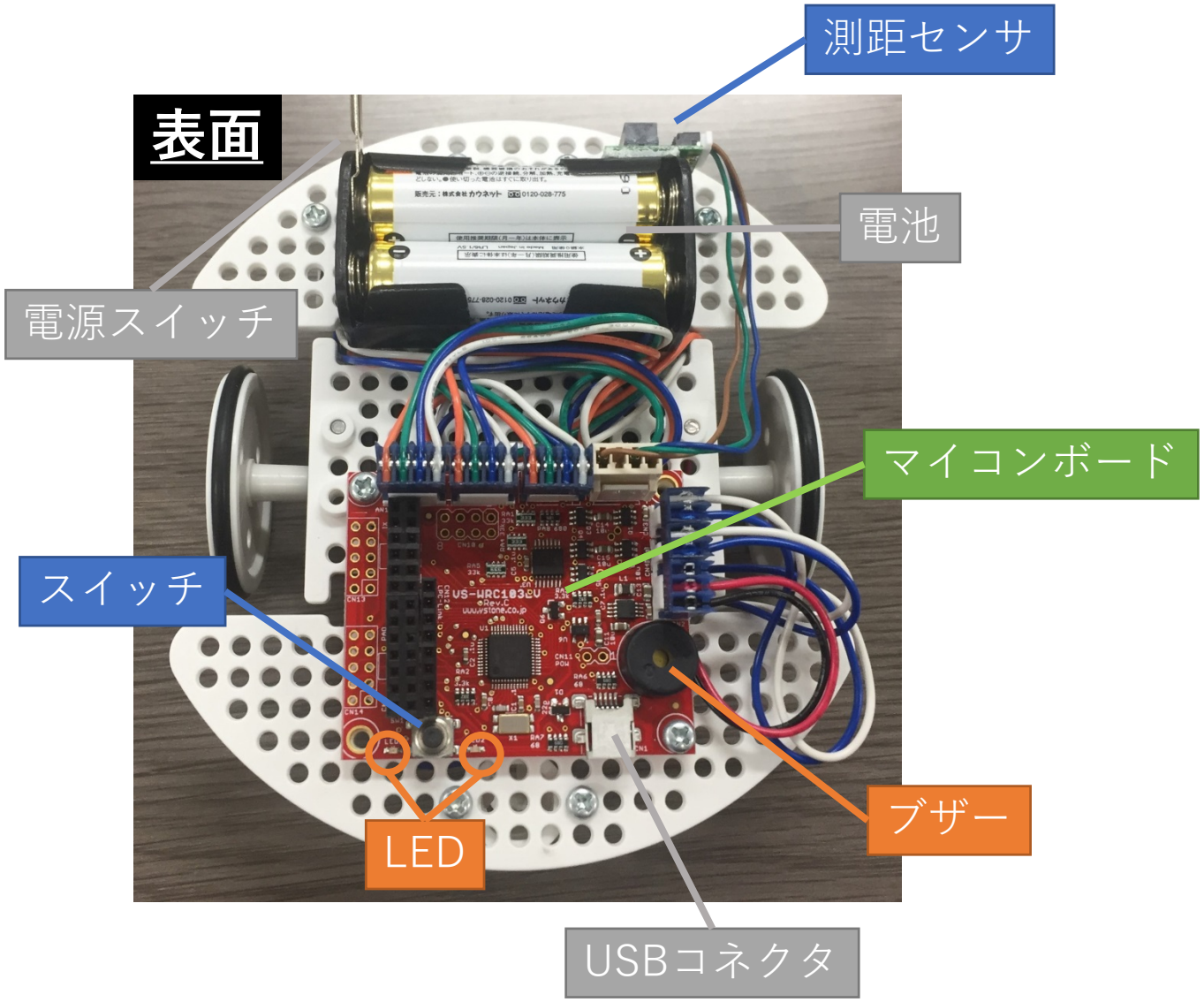
- 使用するソフトウェア
  - LPCXpresso IDE V8.2.2
  - TeraTerm V4.98

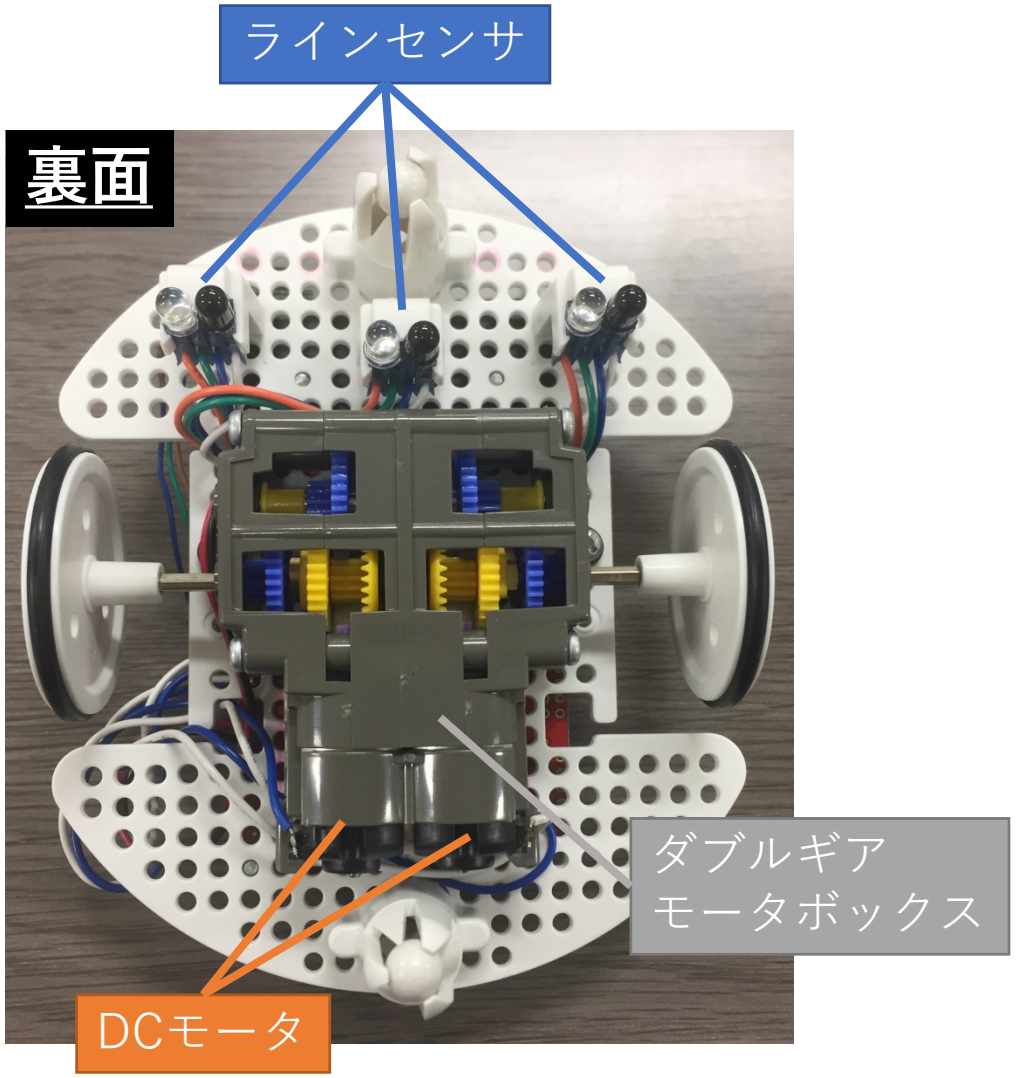
マイコン応用編の目的と目標

- 目的
  - マイコンの高機能なペリフェラルや、追加する高機能なハードウェアを使用し、マイコンに対する知識・技術を深めると共に実際の製品への発展を体感すること

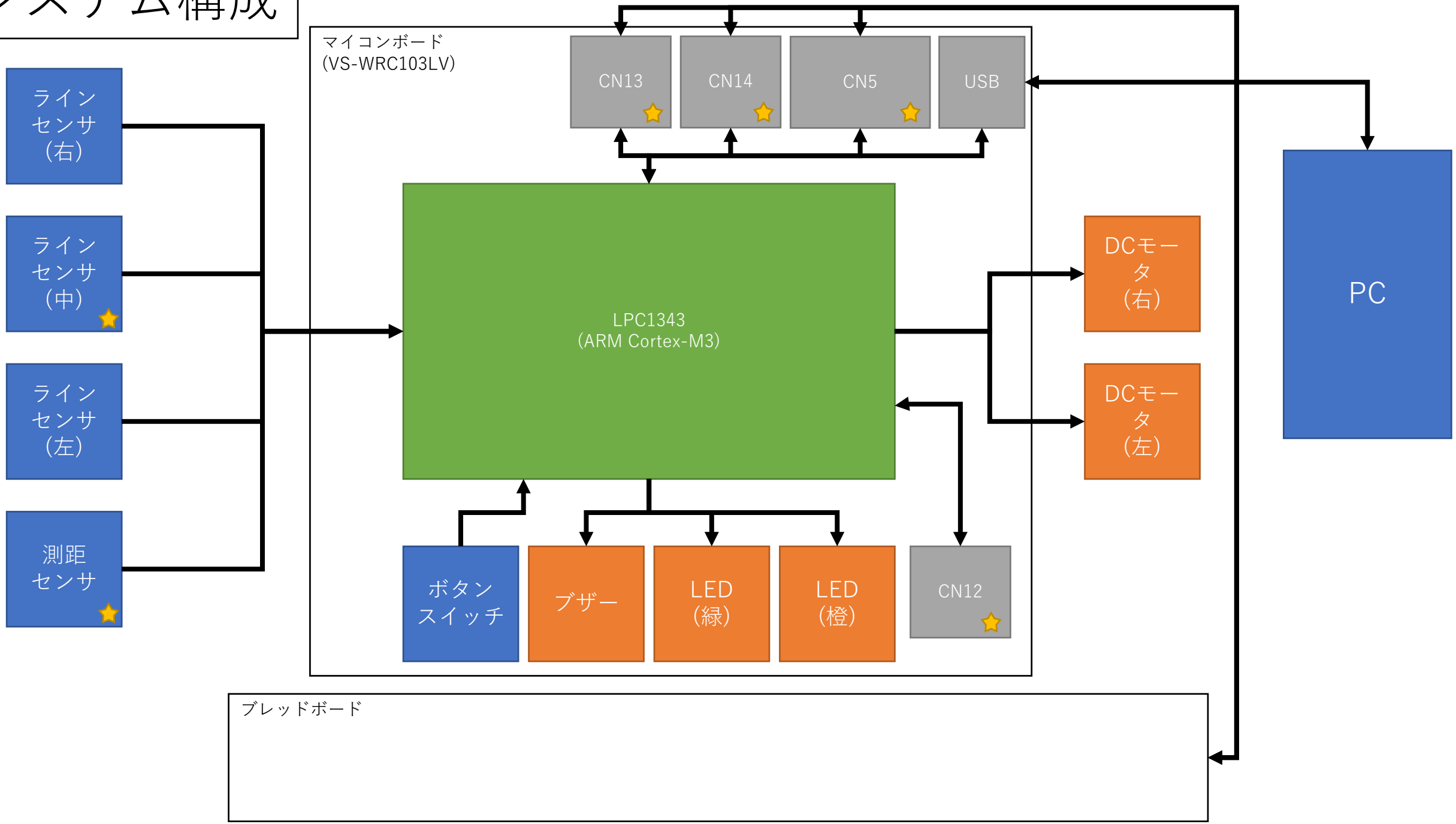
- 目標
  - デバッガの活用
  - 割込み機能の習得
  - 無線通信技術の活用
  - センサのノイズに対する対処の習得

PK-LTCの仕様


表面
測距センサ
電池
電源スイッチ
マイコンボード
スイッチ
LED
ブザー
USBコネクタ


裏面
ラインセンサ
ダブルギア
モータボックス
DCモータ

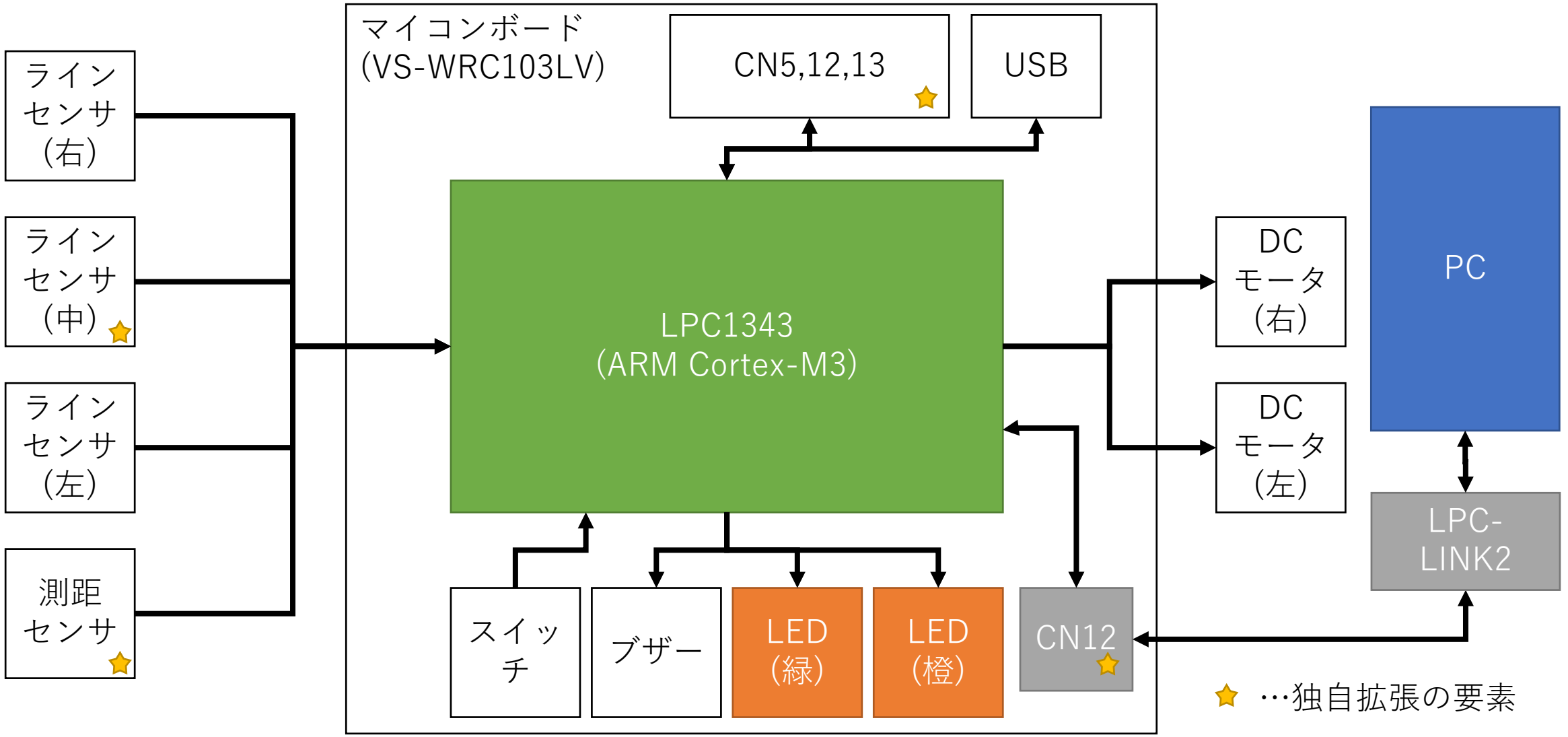
システム構成


マイコンボード
(VS-WRC103LV)
CN13
CN14
CN5
USB
LPC1343
(ARM Cortex-M3)
ラインセンサ(右)
ラインセンサ(中)
ラインセンサ(左)
測距センサ
ボタンスイッチ
ブザー
LED(緑)
LED(橙)
CN12
DCモータ(右)
DCモータ(左)
PC
ブレッドボード

応用編の前提

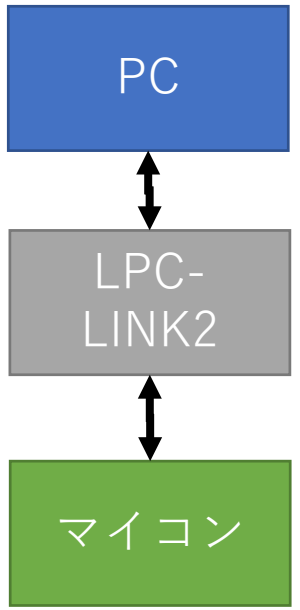
- 基礎編が完了している事を前提とします
  - LEDやモータ、センサなどの制御関数はpk_ltc.hおよびex_dock.hでそれぞれ管理しているものだとします
  - 基礎編のプログラムに自信が無い人は、test03.zipをインポートして使用してください
- 応用編では、test02プロジェクトをベースとしてtest03プロジェクトを作成し、こちらに各種ファイルを作成していきます

10:デバッガ(LPC-LINK2)の活用

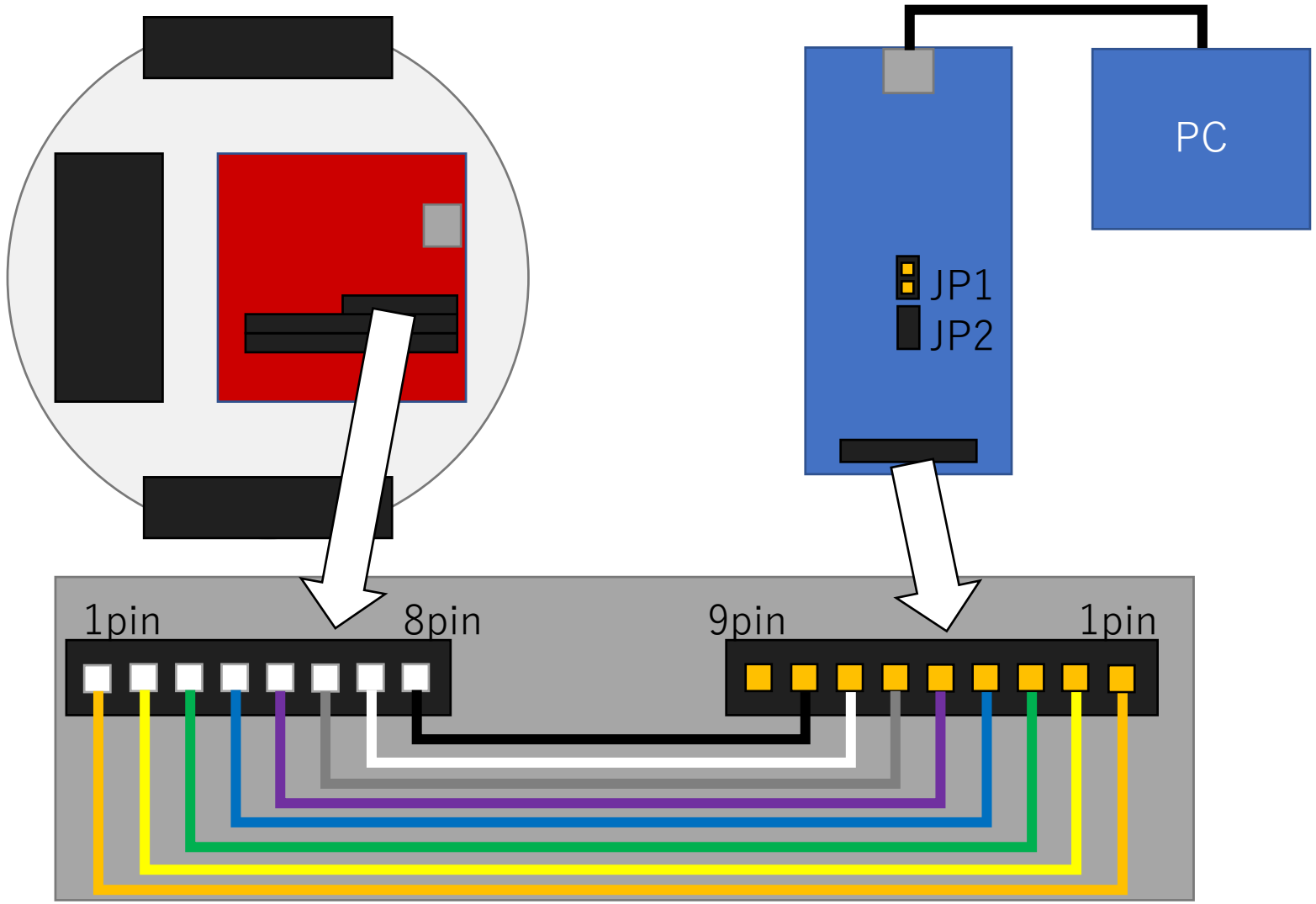


デバッガを活用するメリット

- 今回使用するのはNXP社のLPC-LINK2と呼ばれるデバッガ
  - CMSIS-DAPと呼ばれるARM社標準規格のデバッガ
    - CMSIS-DAPならば、ARM CortexMシリーズであればメーカー問わず対応する
  - いわゆるJTAGデバッガに相当
    - シリアル通信でICの内部回路と通信する仕組み
- デバッガ活用のメリット
  - ブレイクポイントを活用したデバッグが可能
    - さらにプログラムを「ステップ実行」で追う事も可能
  - 変数の値などを確認が可能



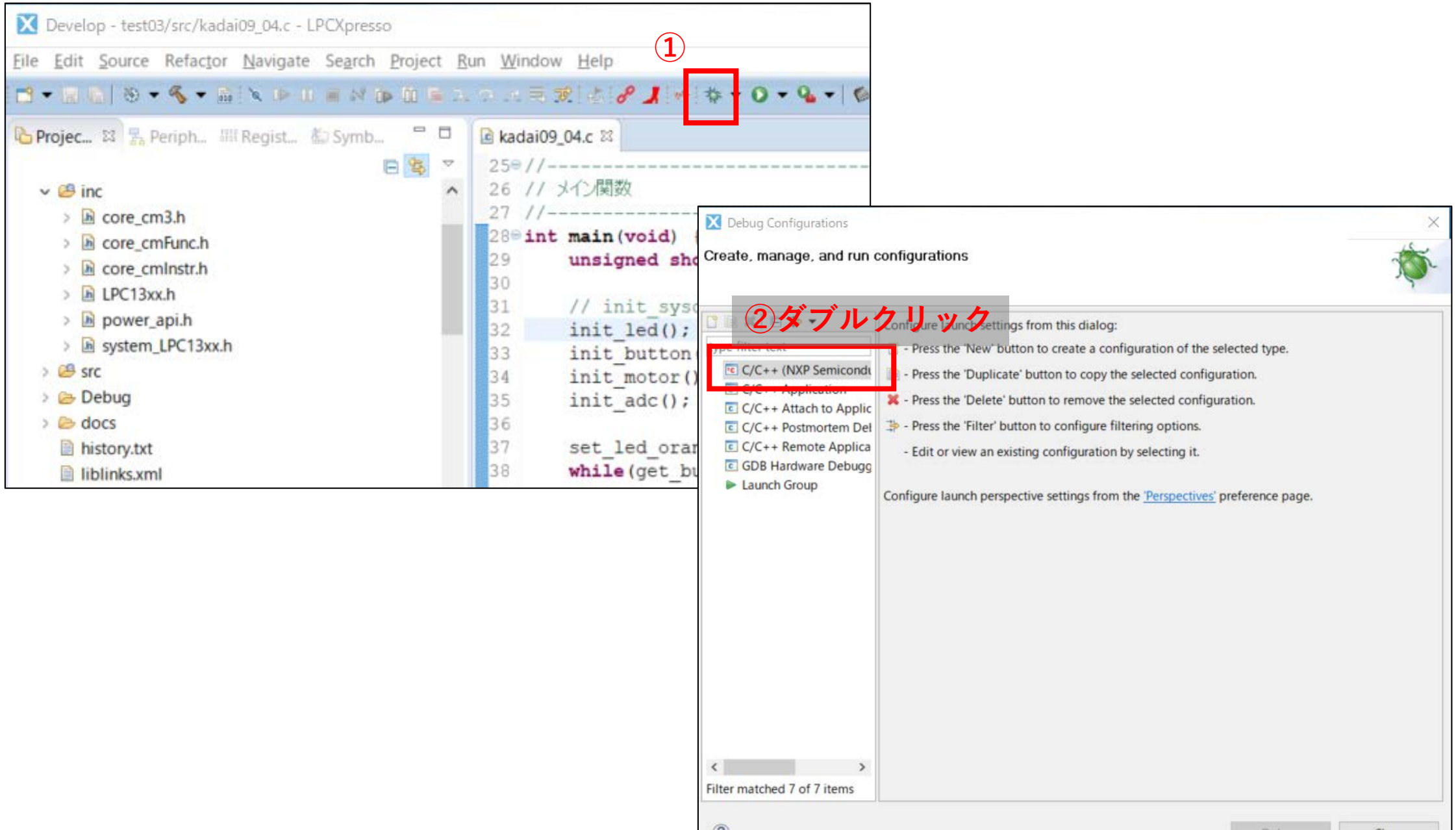
デバッガとの接続図

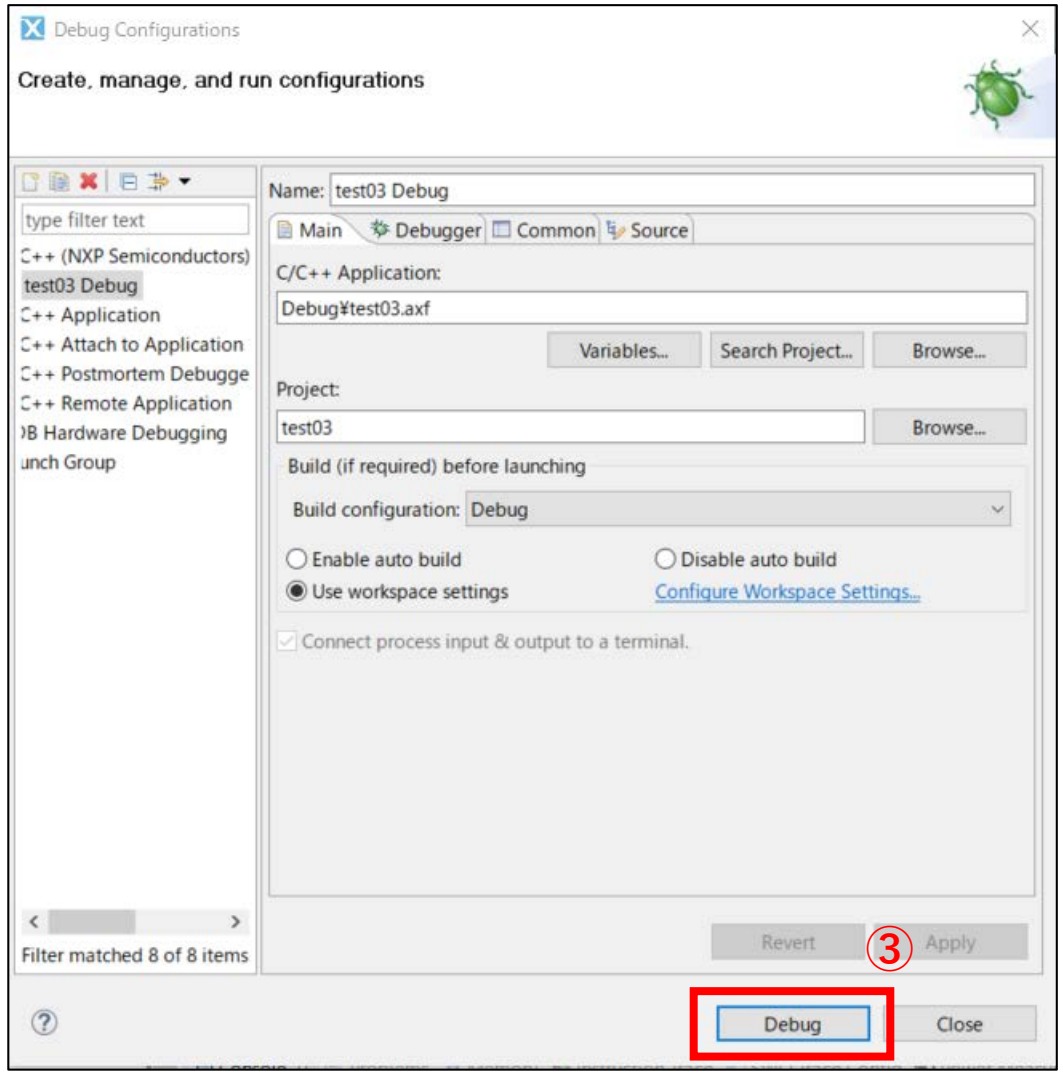


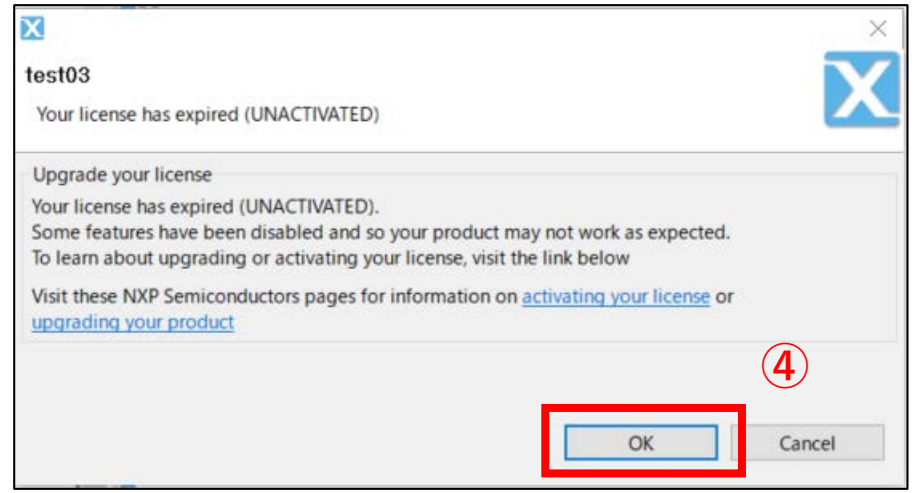
- ケーブルによる接続
  - 1pinの位置が逆である為、注意！！
    - LTC側は左端
    - デバッガ側は右端
- デバッガ
  - 9pinは空きピンとする
  - JP1は開放
  - JP2はショート
- PK-LTC
  - USBは使用しない
  - モータやフルカラーLEDを使用する際には電池ボックスのスナップスイッチをオンにする

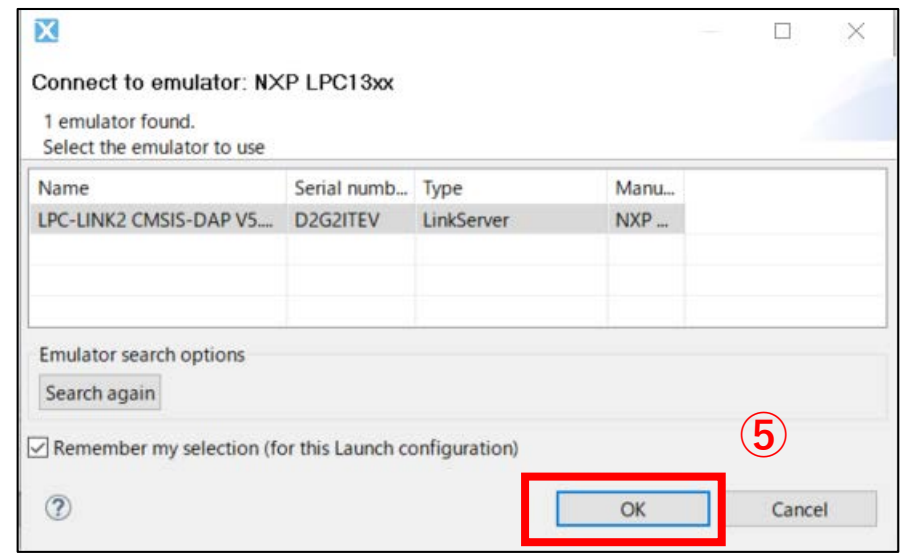
操作方法

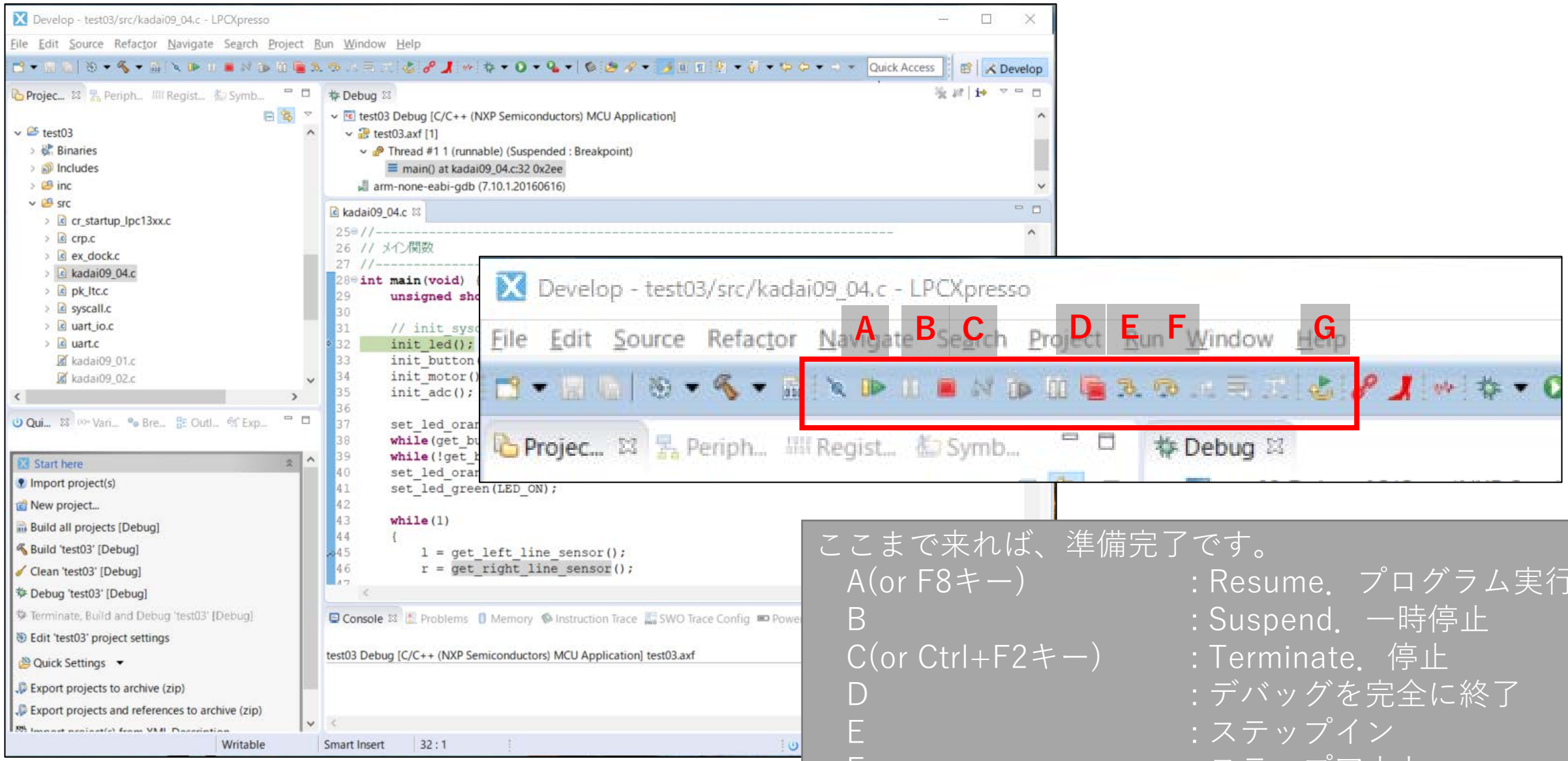
- ハードウェアの接続
  - デバッガのJP2がショートになっているか確認
  - PK-LTCとデバッガを接続する
  - デバッガをUSBケーブルでPCと接続する
- LPCXpressoの起動
- デバッグ対象のプロジェクト・ソースコードを起動
  - 今回はkadai09_04.cとする


Develop - test03/src/kadai09_04.c - LPCXpresso
File Edit Source Refactor Navigate Search Project Run Window Help
①
Projec... Periph... Regist... Symb...
inc
core_cm3.h
core_cmFunc.h
core_cmInstr.h
LPC13xx.h
power_api.h
system_LPC13xx.h
src
Debug
docs
history.txt
liblinks.xml
kadai09_04.c
25 //----------
26 // メイン関数
27 //----------
28 int main(void)
29 unsigned sho
30
31 // init_sys
32 init_led();
33 init_button
34 init_motor()
35 init_adc();
36
37 set_led_orar
38 while(get_bu
Debug Configurations
Create, manage, and run configurations
②ダブルクリック
C/C++ (NXP Semicondu
C/C++ Attach to Applic
C/C++ Postmortem Del
C/C++ Remote Applica
GDB Hardware Debugg
Launch Group
Configure launch settings from this dialog:
- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.
Configure launch perspective settings from the 'Perspectives' preference page.
Filter matched 7 of 7 items


Debug Configurations
Create, manage, and run configurations
type filter text
C++ (NXP Semiconductors)
test03 Debug
C++ Application
C++ Attach to Application
C++ Postmortem Debugge
C++ Remote Application
B Hardware Debugging
unch Group
Filter matched 8 of 8 items
Name: test03 Debug
Main
Debugger
Common
Source
C/C++ Application:
Debug¥test03.axf
Variables...
Search Project...
Browse...
Project:
test03
Browse...
Build (if required) before launching
Build configuration: Debug
Enable auto build
Disable auto build
Use workspace settings
Configure Workspace Settings...
Connect process input & output to a terminal.
Revert
Apply
③
Debug
Close


test03
Your license has expired (UNACTIVATED)
Upgrade your license
Your license has expired (UNACTIVATED).
Some features have been disabled and so your product may not work as expected.
To learn about upgrading or activating your license, visit the link below
Visit these NXP Semiconductors pages for information on activating your license or
upgrading your product
④
OK
Cancel


Connect to emulator: NXP LPC13xx
1 emulator found.
Select the emulator to use
Name
Serial numb...
Type
Manu...
LPC-LINK2 CMSIS-DAP V5....
D2G2ITEV
LinkServer
NXP ...
Emulator search options
Search again
Remember my selection (for this Launch configuration)
⑤
OK
Cancel



ここまで来れば、準備完了です。

A(or F8キー)	: Resume．プログラム実行
B	: Suspend．一時停止
C(or Ctrl+F2キー)	: Terminate．停止
D	: デバッグを完全に終了
E	: ステップイン
F	: ステップアウト
G	: RESTART

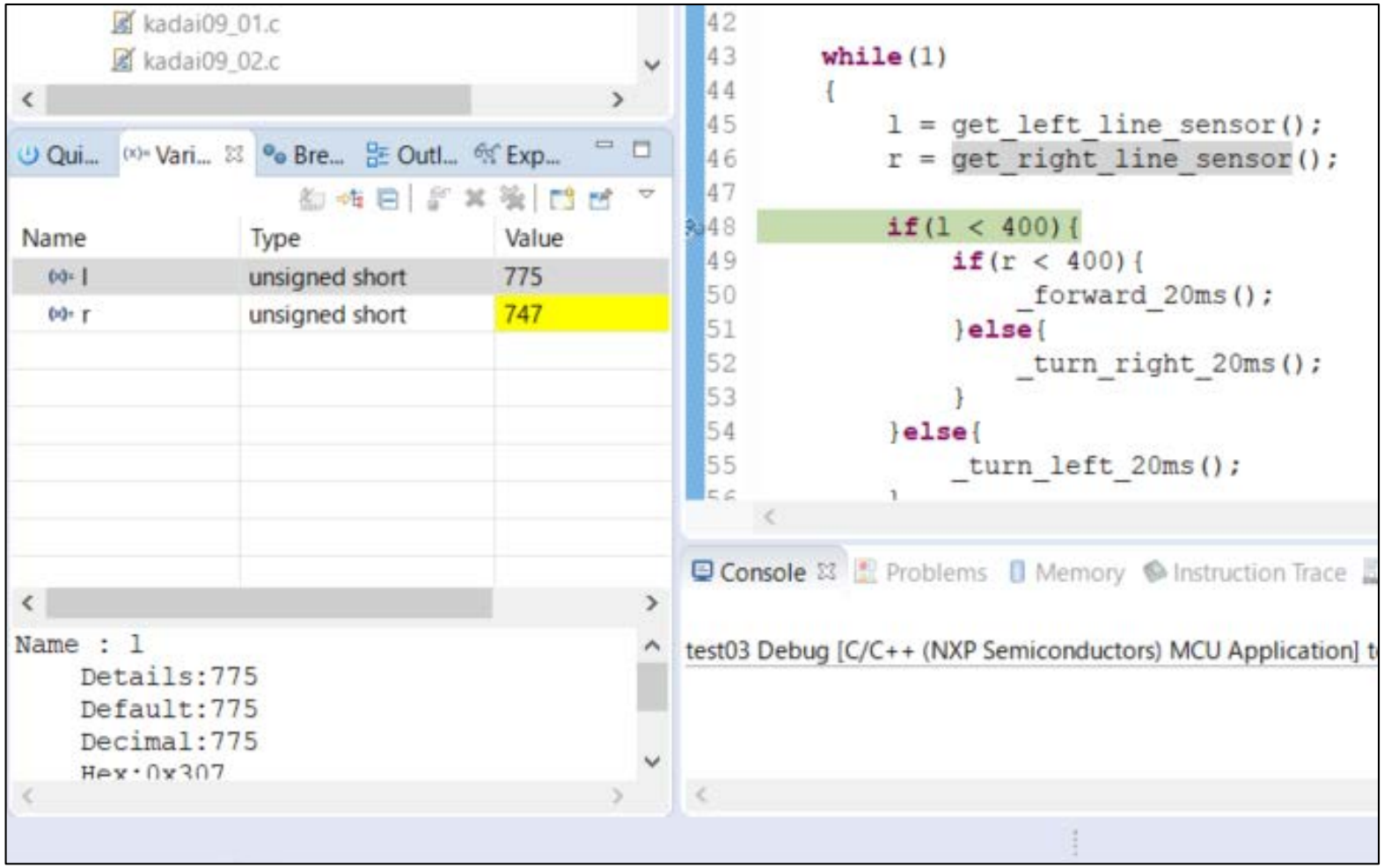
ブレイクポイントとステップ実行

kadai09_04.c

```
34      init_motor();
35      init_adc();
36
37      set_led_orange(LED_ON);
38      while(get_button_black());
39      while(!get_button_black());
40      set_led_orange(LED_OFF);
41      set_led_green(LED_ON);
42
43      while(1)
44      {
45          l = get_left_line_sensor();
46          r = get_right_line_sensor();
47
48          if(l < 400){
49              if(r < 400){
50                  _forward_20ms();
51              }else{
52                  _turn_right_20ms();
53              }
54          }else{
55              _turn_left_20ms();
```

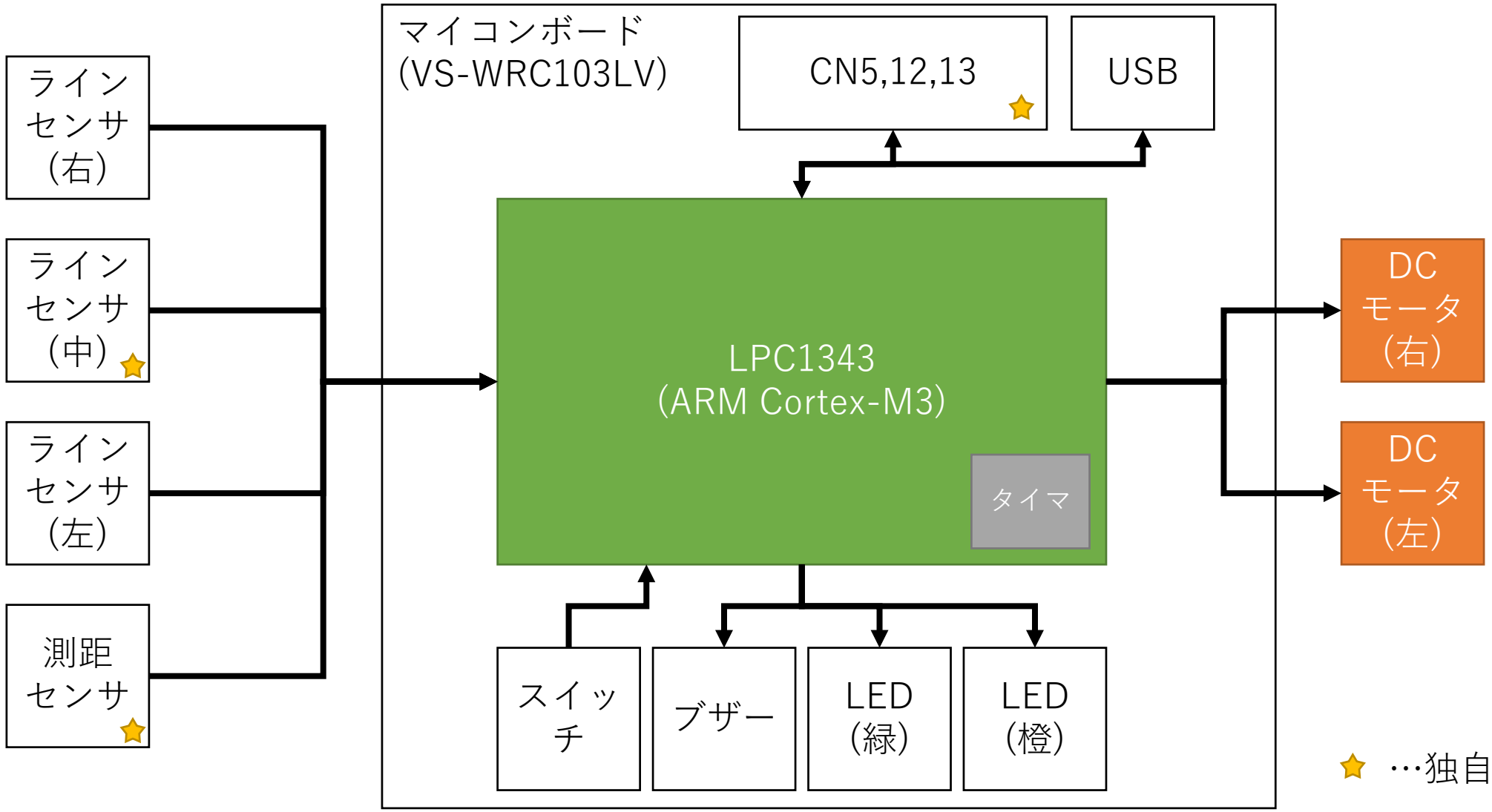
- ブレイクポイントの設置
  - 行数をダブルクリック
  - 実行時に該当箇所でプログラムを自動停止
    - そこからのステップ実行も可能に
  - プログラムの停止中に設置が可能
- ステップ実行
  - 1行ずつプログラムを実行すること

変数の値の確認



- Variableタブを選択すると現在の処理内の変数が表示
  - 型と保持している値が確認できる
  - 値は書き換えも可

11:タイマによるモータのPWM制御

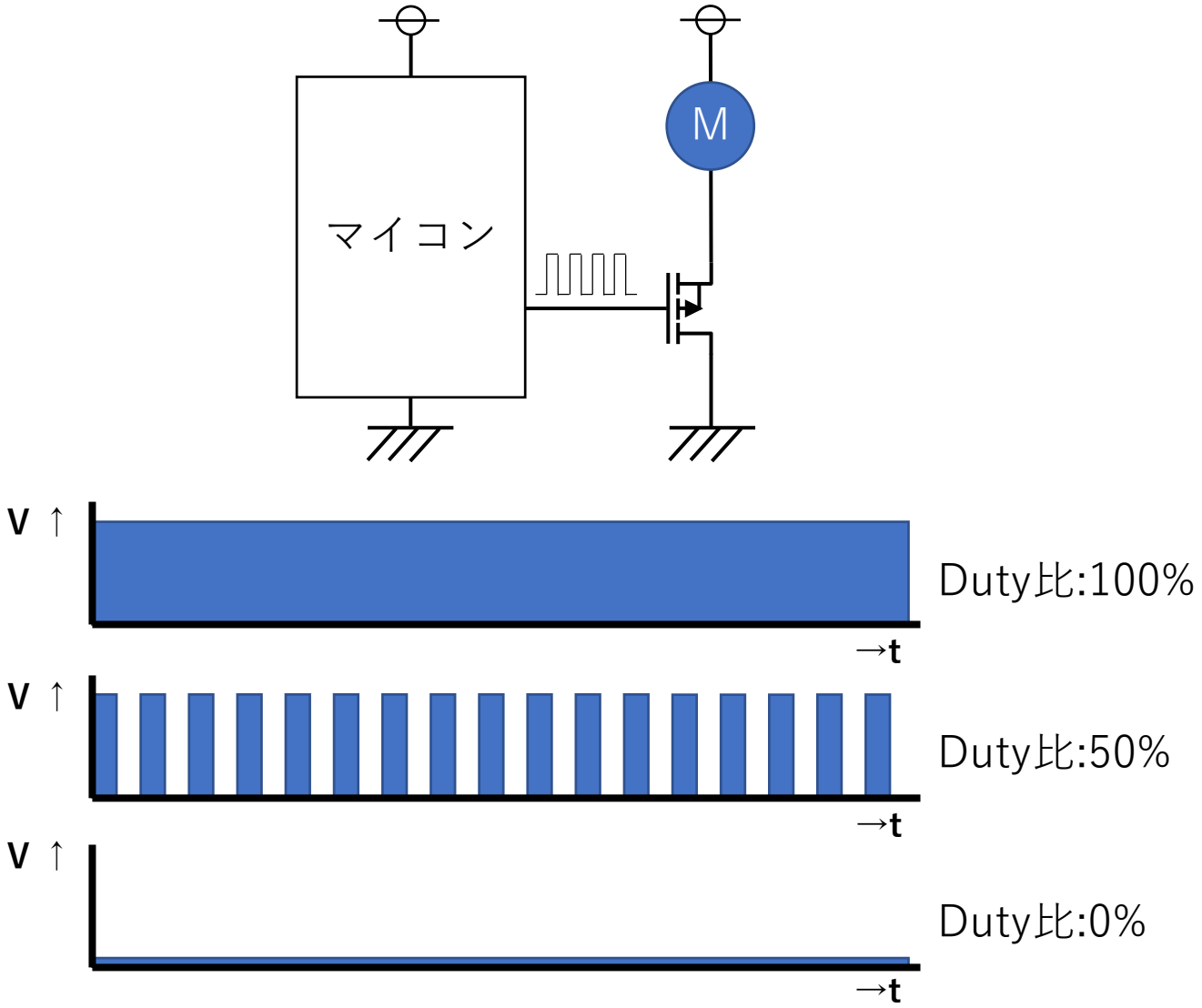


★ …独自拡張の要素

はじめに

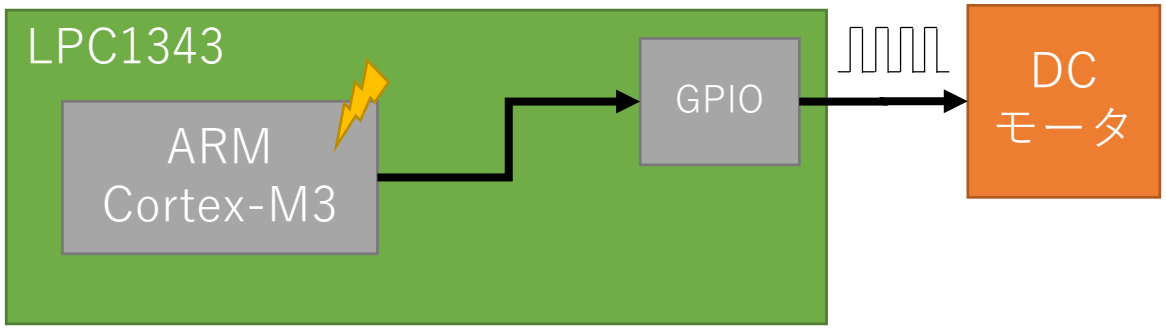
- GPIOとWait関数によるPWM制御の問題点を解消する為の1つの方法として「タイマのPWM機能」の活用があります。
- タイマを活用する事で、プロセッサが直接制御することなく、PWM波形をIOポートから出力する事が可能です。この機能は、多くのマイコンに内蔵されています。

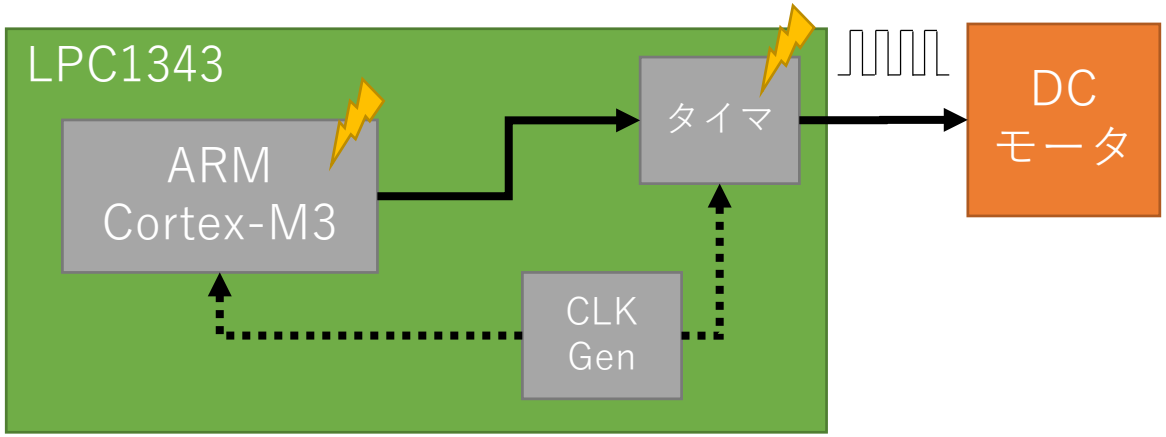
速度制御とその仕組み



- PWM制御による速度制御を実現
- モータに流す電流を細かくON/OFFさせてTotalで流れる電流量を調整することで速度変化が生じる

タイマ機能の活用によるPWM出力

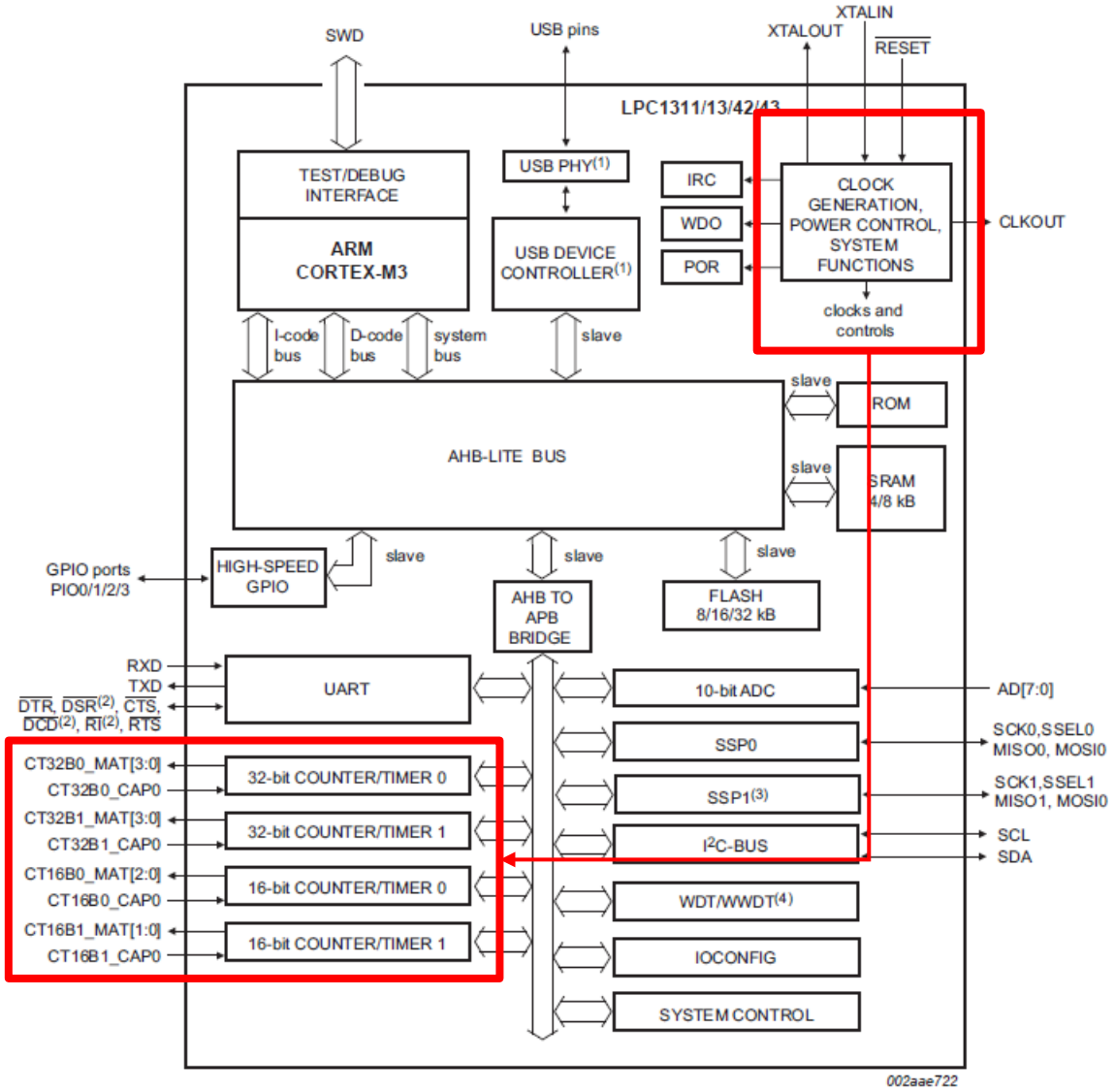




- 基礎編ではGPIOのON/OFF制御をプログラムから直接行う事でPWM制御を実現しました
- CPUがPWM制御にかかりきりになってしまう点が問題

- タイマと呼ばれる周辺ペリフェラルを活用し、タイマから任意のデューティ比の矩形波を出力する

タイマ機能の活用によるPWM出力



- LPC1343にはタイマが4本ある
  - 16bitタイマ×2本
  - 32bitタイマ×2本
- 各タイマには2～4本のPWM出力用のピンが割り当て
  - GPIOなどと兼用ピン

例題：rei11_01.c(ソースコード提供)

```c
int main(void) {
    int vol;

    init_syscall();
    init_adc();
    init_motor2();

    while(1) {
        vol = get_volume();
        vol = (vol - 512) * 64;
        drive_motor(vol, -vol);

        //printf("%d¥n", vol);
    }
    return 0 ;
}
//-------------------------------------------------------------
// 関数本体
//-------------------------------------------------------------
void init_motor2(){
    // GPIO
    LPC_GPIO2->DIR |= 0x000F;
    // LPC_GPIO0->DIR |= 0x0100;
    // LPC_GPIO1->DIR |= 0x0200;

    //Timer16B0,B1
    //use MTR1
    LPC_IOCON->PIO0_8   &= 0xE7;// PullUp disable,CT16B0_MAT0
    LPC_IOCON->PIO0_8   |= 0x02;// Selects function

    //use MTR2
    LPC_IOCON->PIO1_9   &= 0xE7;// PullUp disable,CT16B1_MAT0
    LPC_IOCON->PIO1_9   |= 0x01;// Selects function

init_timer16();
}
```

```c
void drive_motor(short m1,short m2){
short mta1, mta2;
unsigned int duty1, duty2;
volatile unsigned int gpio2_tmp;

gpio2_tmp = LPC_GPIO2->DATA;
gpio2_tmp &= ~0x000F;

mta1 = m1;
mta2 = m2;

// 反転制御(取付向き的にCWが反対になる)
if(mta1 == 0x8000){
mta1 = 0;
}
if(mta2 == 0x8000){
mta2 = 0;
}
mta2 = -mta2;

// 回転方向の制御・速度制御
if(mta1 > 0){
duty1 = (unsigned int)(~(mta1*2));
gpio2_tmp |= 1;
}
else if(mta1 < 0){
duty1 = (unsigned int)(~(-mta1*2));
gpio2_tmp |= 2;
}
else{
duty1 = 0;
}
```

```
if(mta2 > 0){
duty2 = (unsigned int)(~(mta2 * 2));
gpio2_tmp |= 1 << 1 * 2;
}
else if(mta2 < 0){
duty2 = (unsigned int)(~(-mta2 * 2));
gpio2_tmp |= 2 << 1 * 2;
}
else{
duty2 = 0;
}

LPC_TMR16B0->MR0 = duty1 & 0x0000FFFF;
LPC_TMR16B1->MR0 = duty2 & 0x0000FFFF;

LPC_GPIO2->DATA = gpio2_tmp;
}
```

```
void init_timer16(void){
/* Enable AHB clock to the CT16B0,CT16B1.
*/
LPC_SYSCON->SYSAHBCLKCTRL |=
((1<<9)|(1<<8)|(1<<7));

//Timer stop
LPC_TMR16B0->TCR &= ~0x01;
LPC_TMR16B1->TCR &= ~0x01;

//set PWMmode
LPC_TMR16B0->PWMC =
(0<<3|0<<2|0<<1|1<<0);//MAT0 enable,
MAT1,MAT2,MAT3 disable
LPC_TMR16B1->PWMC =
(0<<3|0<<2|0<<1|1<<0);//MAT0 enable,
MAT1,MAT2,MAT3 disable

//interrupt flag clear
LPC_TMR16B0->IR = 0;
LPC_TMR16B1->IR = 0;

//interrupt
LPC_TMR16B0->MCR = 0;
LPC_TMR16B1->MCR = 0;

//maximum value for the Prescale Counter.
Divide Mainclk
LPC_TMR16B0->PR = 0;
LPC_TMR16B1->PR = 0;

//Set Duty
LPC_TMR16B0->MR0 = ~0x0000;//MTR1
LPC_TMR16B1->MR0 = ~0x0000;//MTR2

//Timer start
LPC_TMR16B0->TCR |= 0x01;
LPC_TMR16B1->TCR |= 0x01;

return;
}
```

- 今回の関数
  - init_motor2()
  - drive_motor()
  - init_timer16()
- 以下を確認
  - サンプルコードを書き込みボリュームによりモータの速度調整ができる事
  - PWM波形をオシロスコープで確認
  - Main関数内のWhileループに緑LEDを200ms周期で点滅するコードを追加。モータの動作への影響を見る

特定処理の関数化

- drive_motor関数は、直進・後進・右旋回・左旋回など、プログラムにて直感的に記述できません。
- そこで、関数化することでより直感的に容易にPK-LTCが制御できるようにしたいと思います。

作成する関数:kadai11_01.c

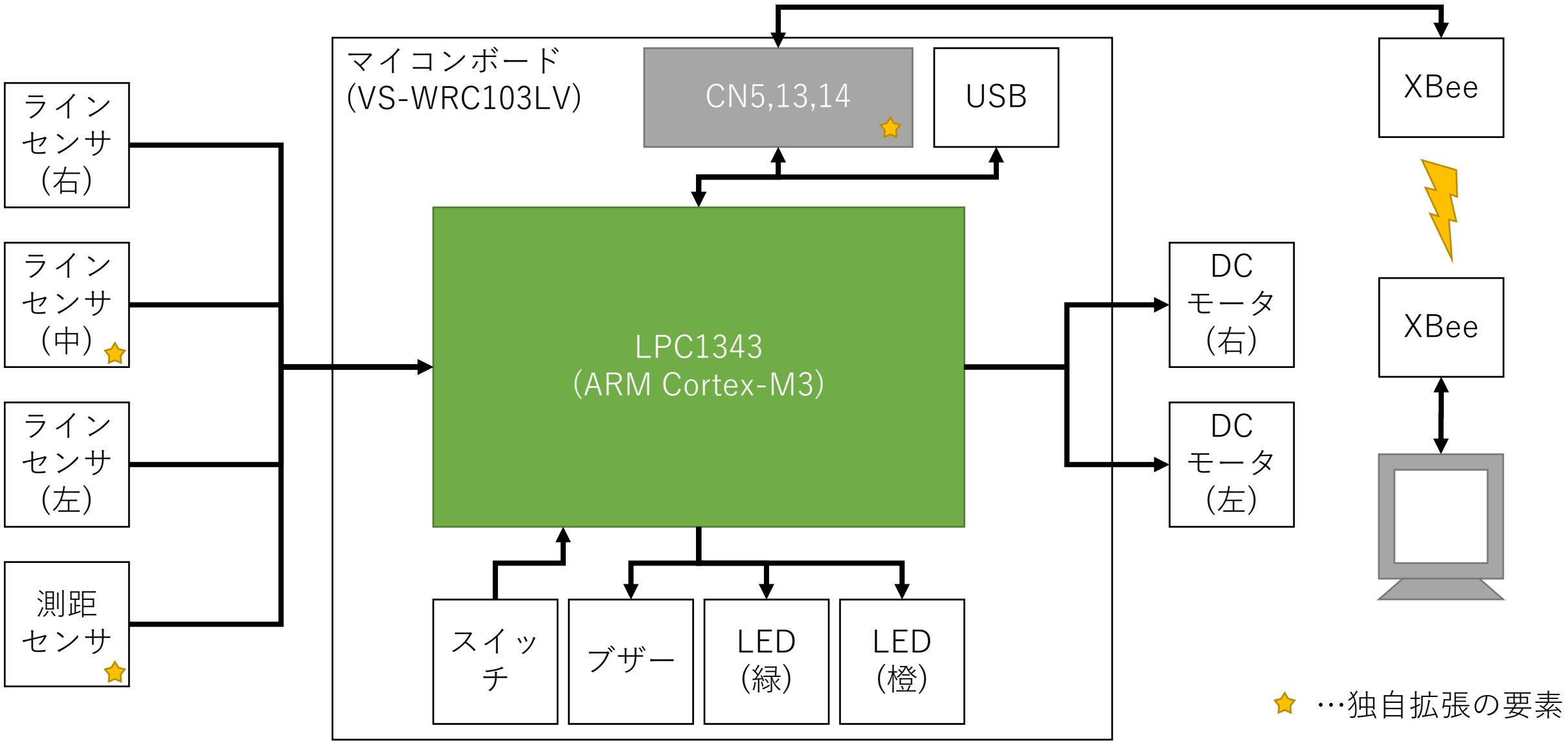
- void forward(short speed); 
  - PK-LTCが前進する関数
  - 引数：モータの制御値
    - 0 ：フリー（ブレーキ）
    - 0x8000 ：フリー（ブレーキ）
    - 時計回り最大値 ：0x7FFF (32767)
    - 反時計回り最大値：0x8001 (-32767)
  - 戻り値：なし
- void back(short speed); 
  - PK_LTCが後進する関数
- void turn_right(short speed); 
  - PK_LTCが右旋回する関数
- void turn_left(short speed); 
  - PK_LTCが左旋回する関数
- void stop(); 
  - PL_LTCを停止する関数
- void forward_t(short speed, unsigned int msec);
  - PK-LTCが指定秒数、前進する関数
  - 引数１：モータの制御値
    - 0 ：フリー（ブレーキ）
    - 0x8000 ：フリー（ブレーキ）
    - 時計回り最大値 ：0x7FFF (32767)
    - 反時計回り最大値：0x8001 (-32767)
  - 引数２：指定した時間(msec)の間、モータが動作する
  - 戻り値：なし
- void back_t(short speed, unsigned int msec);
  - PK-LTCが指定秒数、後進する関数
- void turn_right_t(short speed, unsigned int msec);
  - PK-LTCが指定秒数、右旋回する関数
- void turn_left_t(short speed, unsigned int msec);
  - PK-LTCが指定秒数、左旋回する関数

課題：kadai11_02.c

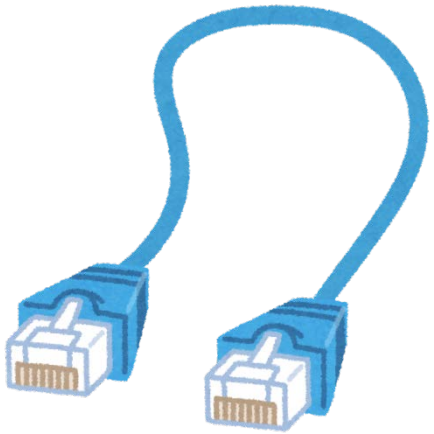
- Kadai09_01.cで開発したライントレースのプログラムを今回作成した関数で作り変えよ

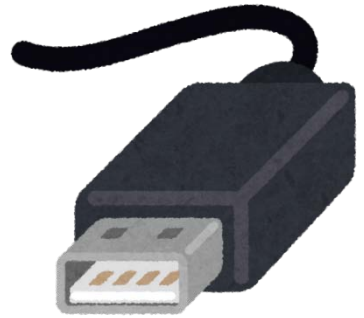
- 作り変えたプログラムに、緑LEDを200ms周期で点滅させる処理を追加し、モータの動作に影響を与えるか確認せよ

12:無線通信「XBee」を活用したデバッグ

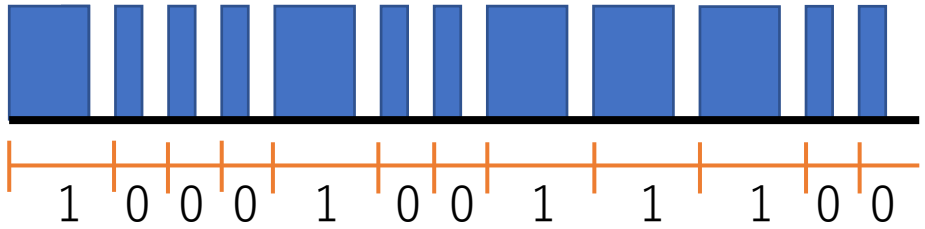


シリアル通信とは





- シリアル通信には主に2つの意味で使われる
- 広義のシリアル通信
  - １本の信号線に対してデータを直列に送信する通信のこと



  - 例：USB, Ethernet, RS-232, MIDI, SerialATA, UART, SPI, I2Cなど
- 狭義のシリアル通信
  - RS232C通信ないし、UARTによる通信のこと
  - FAや組込み業界のエンジニアの中では一般的な呼び方

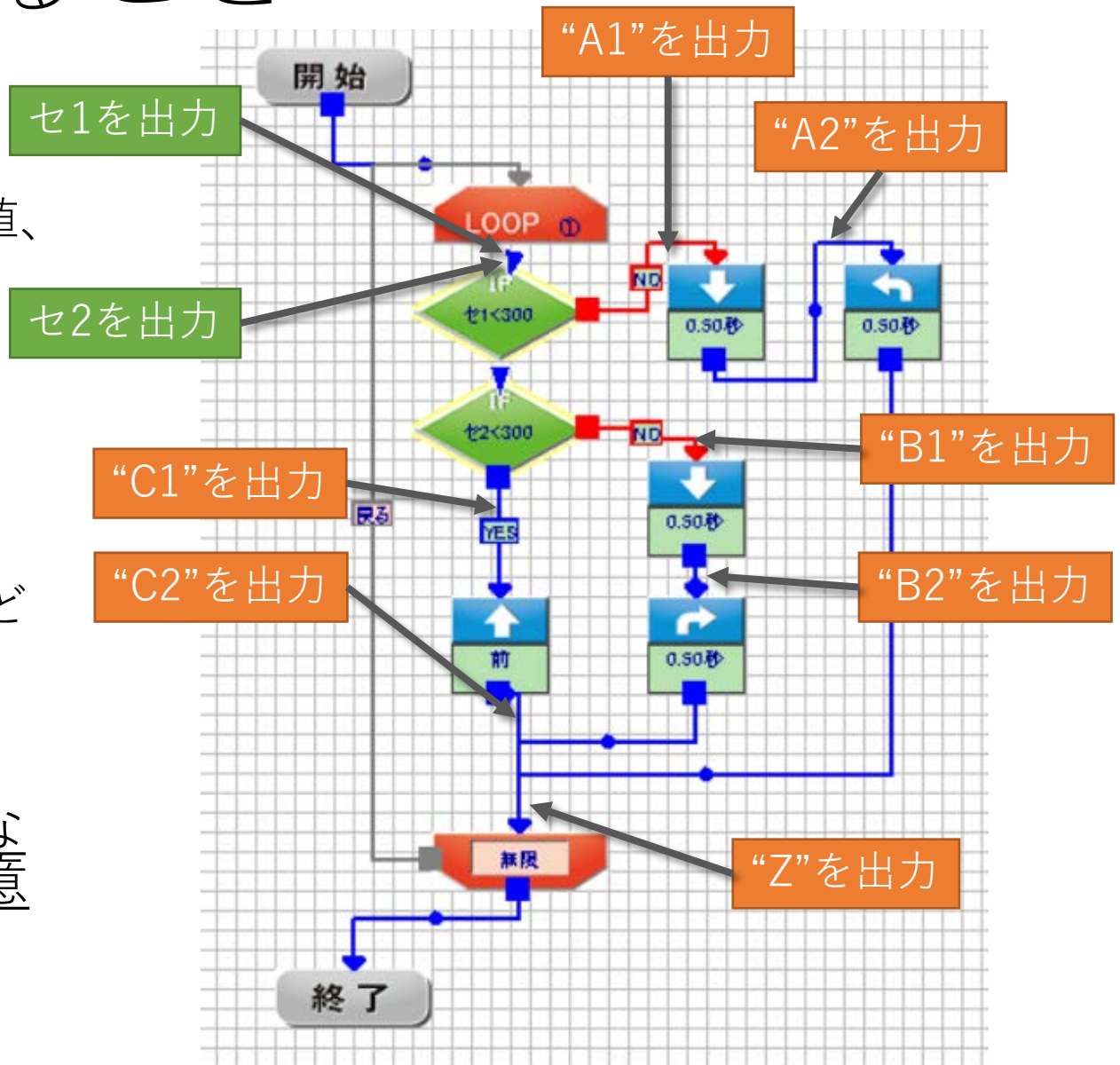
LTCの開発における課題

- PK-LTCを実際に走行させながらのデバッグが難しいこと
  - 実際のセンサの値が拾えない
  - 実際に走行中の処理の流れ・遷移を追いづらい

- デバッグの基本はトライ＆エラーの繰り返し
  - エラーが出た際に、何が原因であるか？を考え推論することが大事
  - ロボットの実際の挙動を目で追うだけでは情報が少ない
    - プログラム内部の情報が欲しい

printfデバッグで出来ること

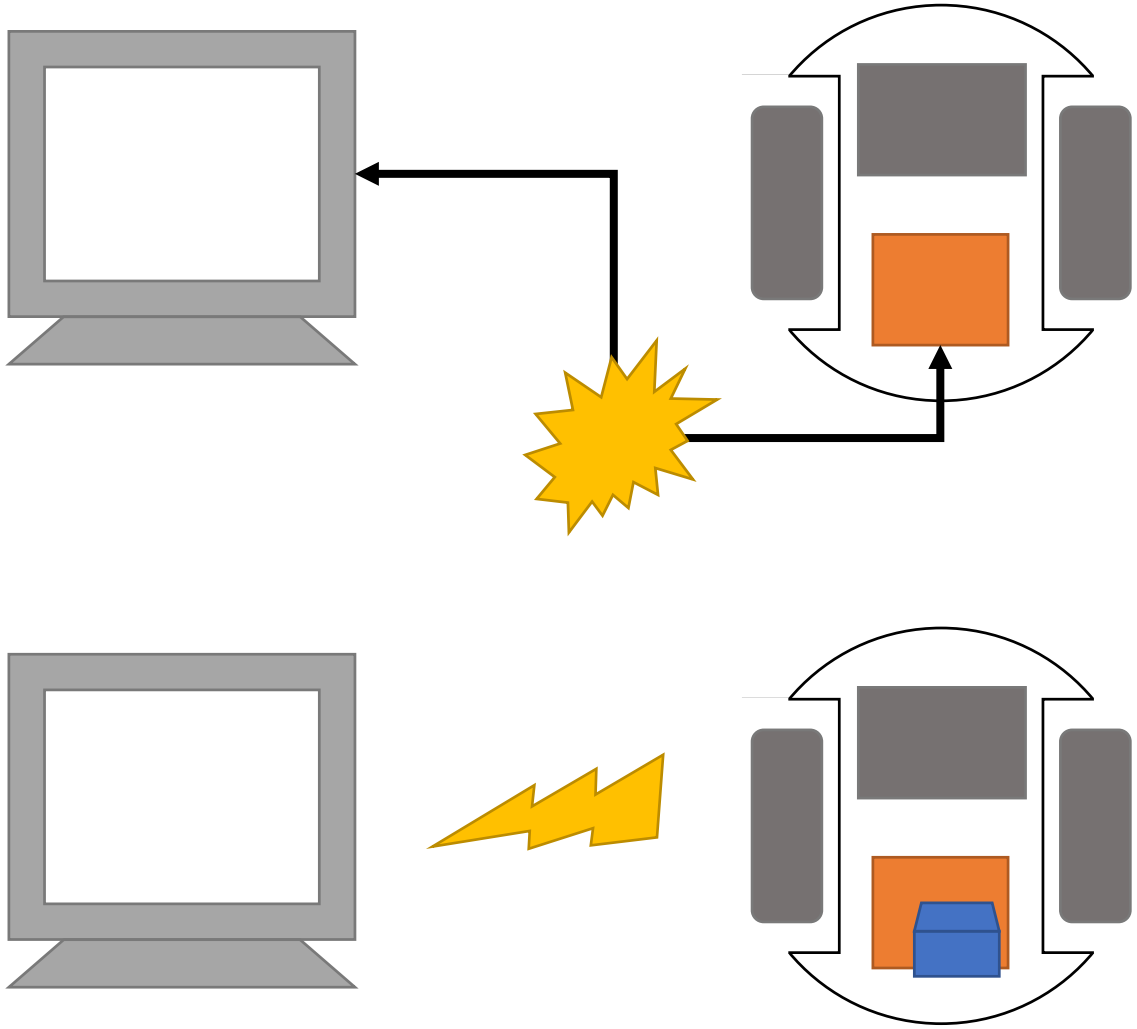
- 変数の中身を確認する
  - 例えば、ADから取得したラインセンサ値、距離センサ値の確認
  - カウンタの値の確認
  - 変化しやすい値は、その値によって処理が分岐することが主である
- 内部フローを確認する
  - 特定のif文の中に入ったら'a'を表示する、if文に入らなかったら'z'を表示する。など
  - 自身の作成したプログラムが意図したタイミングで分岐や繰り返しをしているかを判断
- ただし、printf関数はLED点灯/消灯などに比べて処理時間がかかる点に注意



無線通信モジュールのメリット

- 有線通信(USB-Serialケーブルなど)
  - 実際に走行させながらデータを拾うのは難しい

- 無線で通信できれば解決！！

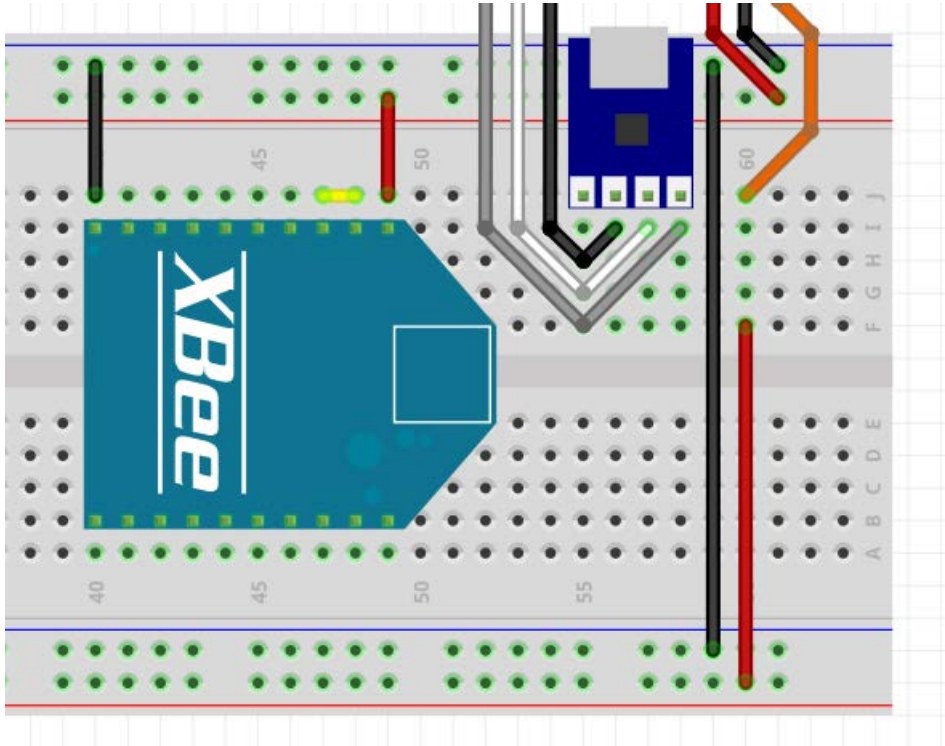


XBeeの透過シリアル機能の活用

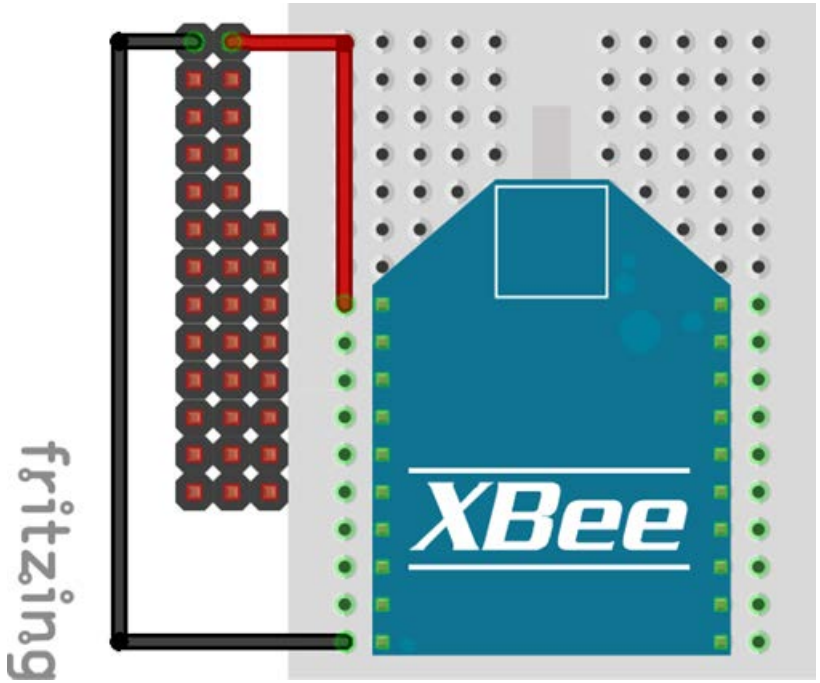
- PC側、マイコン側から見ると、通常のシリアル通信(UART)
- 間に入っているXBeeもジュールが上位のレイヤのプロトコルへUART流し込んだデータを変換する

動作確認(ループバック)

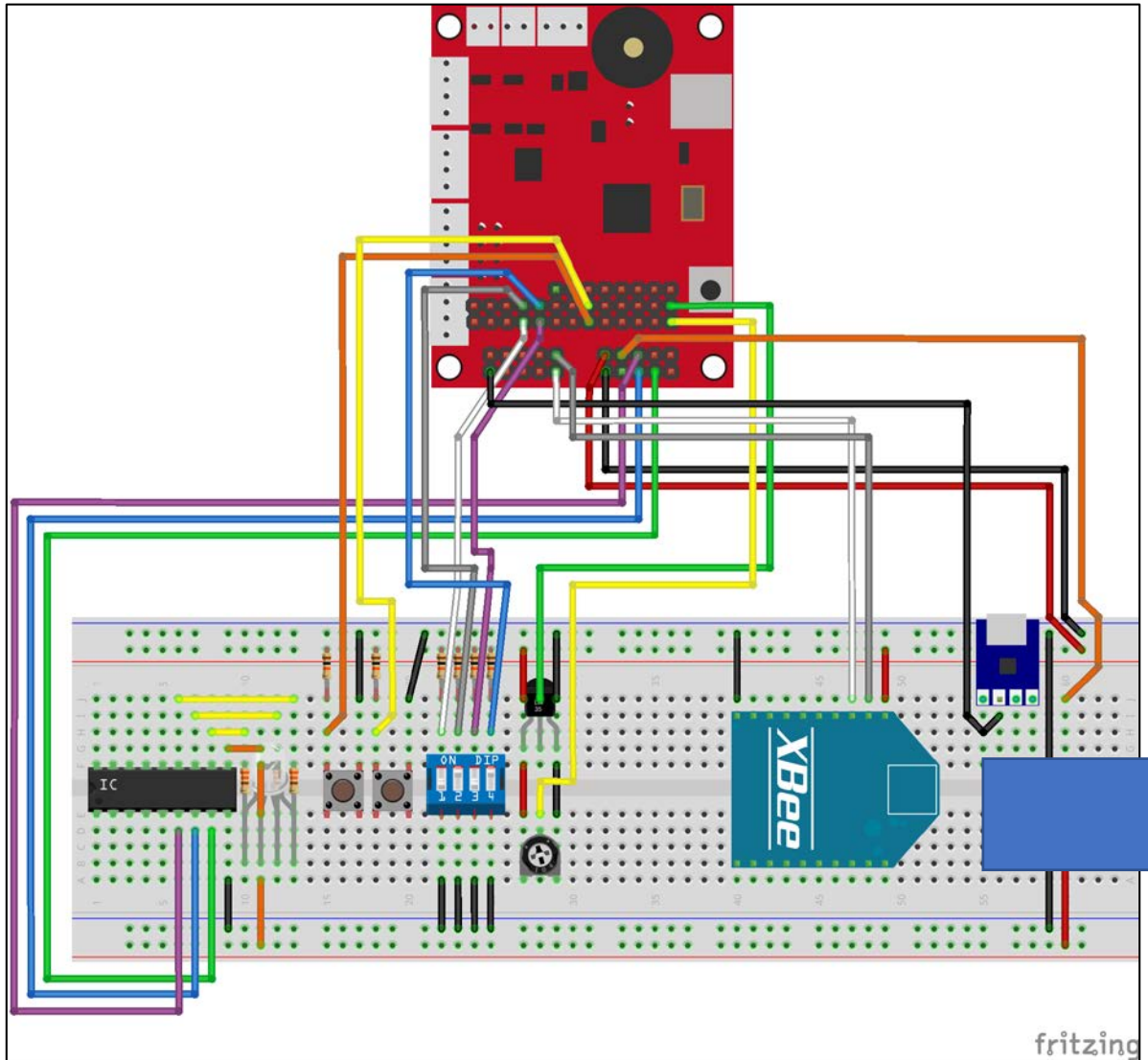
ドック上で確認する場合



拡張ボード上で確認する場合

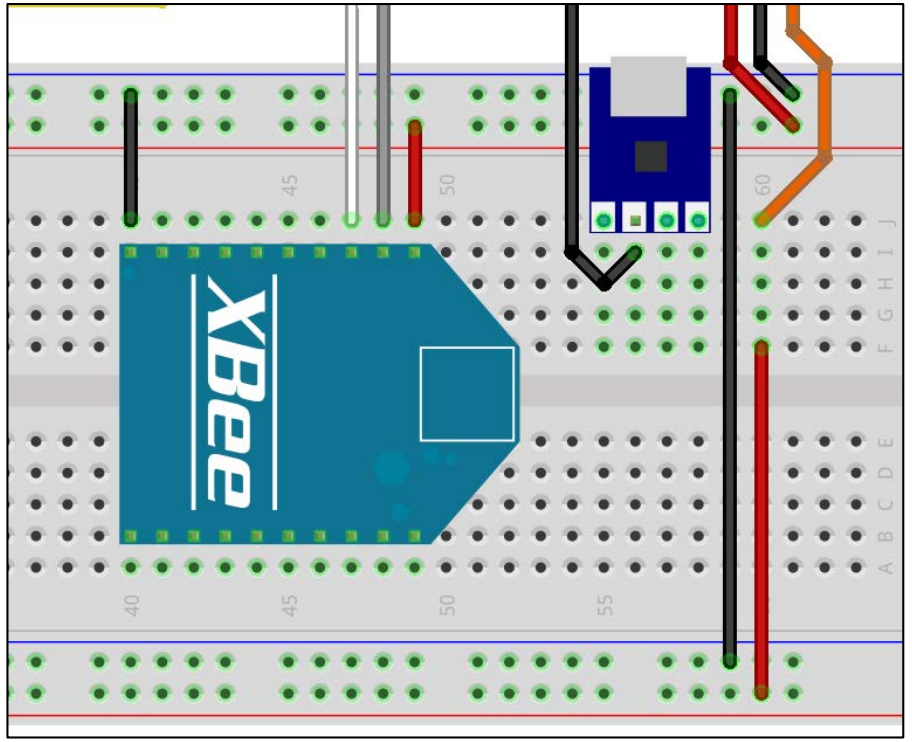


実体配線図(ドックの場合)

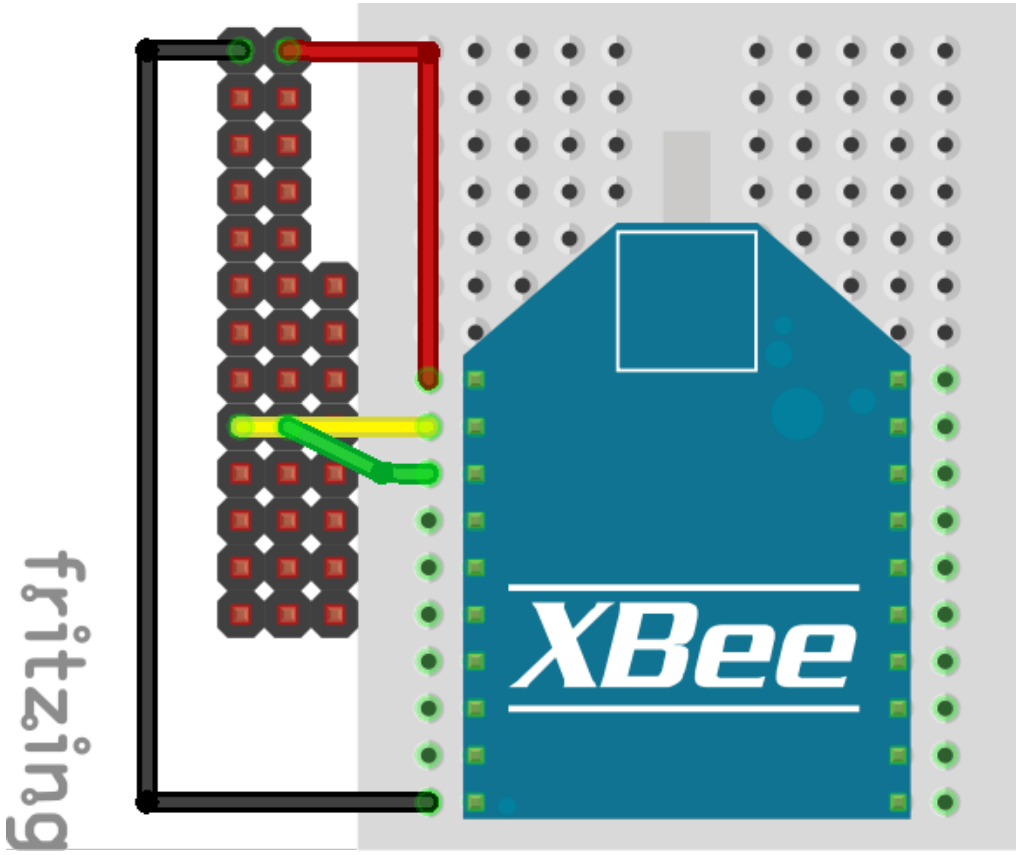


USBシリアルからTxDおよびRxDの信号線を繋ぎ変えるだけでOK！！

拡大図

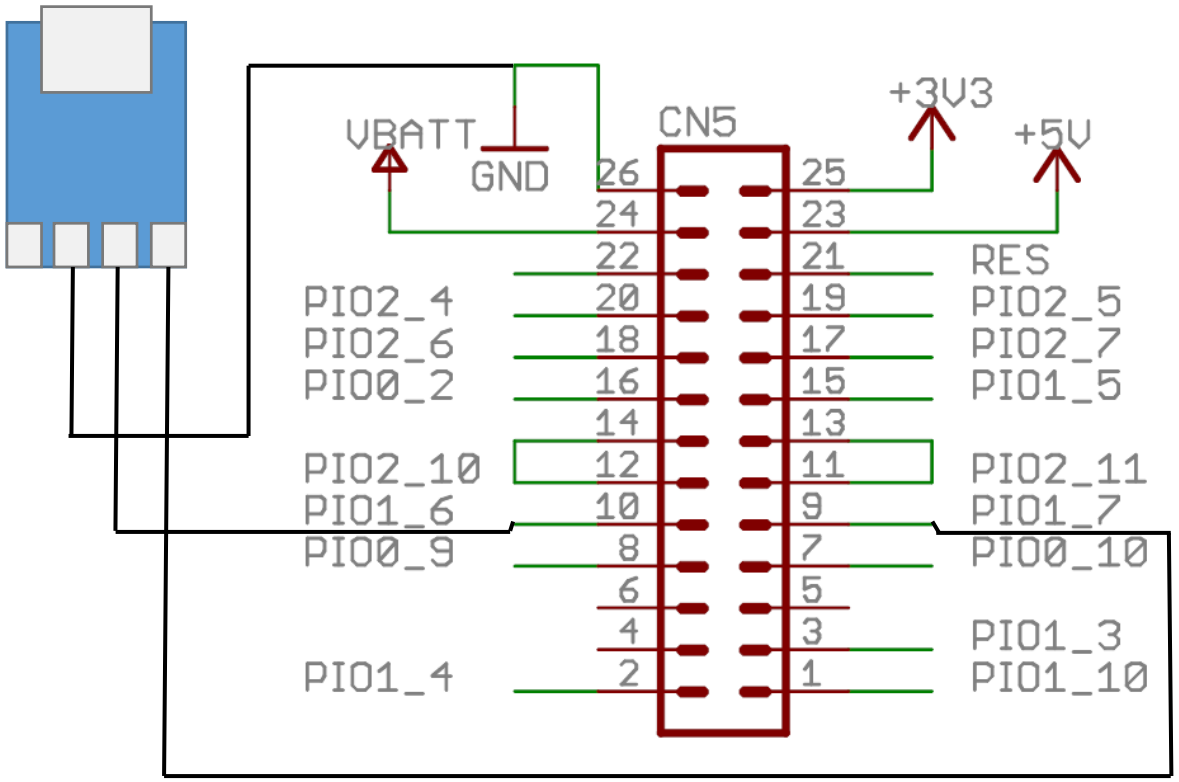


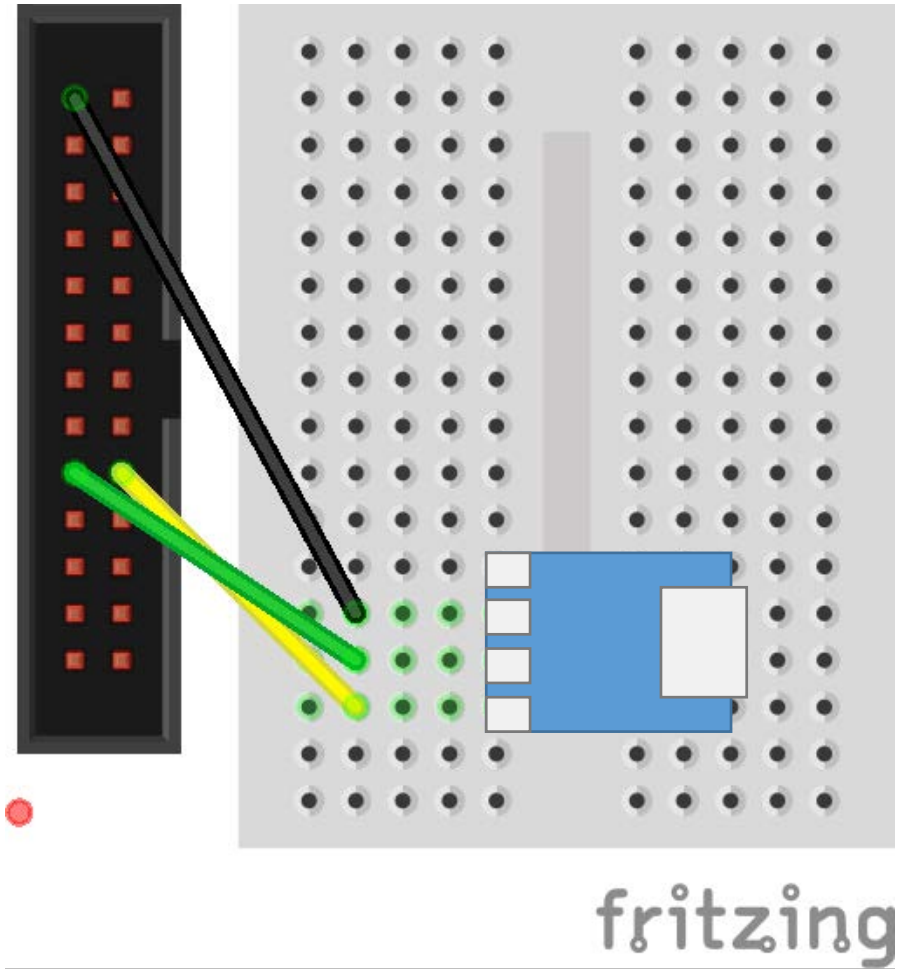
実装配線図(拡張ボードの場合)



- USB-Serial変換基板からXBeeモジュールへハードウェア的に作り変える
- Xbee
  - 2pin:Xbee TxD
  - 3pin:Xbee RxD
- LTC(CN5)
  - PIO1_6はRxDと兼用ピン
  - PIO1_7はTxDと兼用ピン

【参考】有線の場合：回路図＆実装配線図





- PIO1_6はRxDと兼用ピン
- PIO1_7はTxDと兼用ピン

例題:rei12_01.c

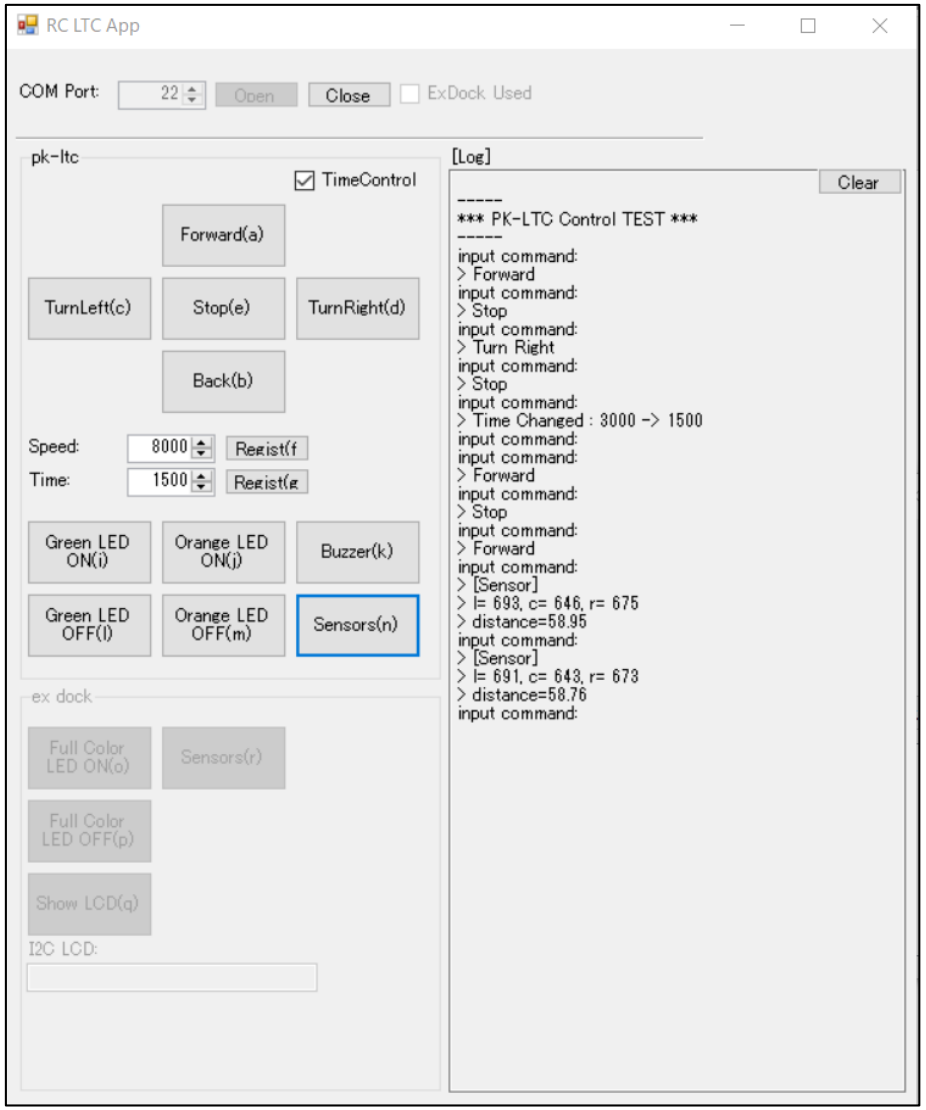
```
int main(void) {
    char c;

    init_syscall();
    init_led();

    while(1)
    {
        set_led_orange(LED_ON);
        scanf("%c", &c);
        set_led_orange(LED_OFF);
        printf("%c", c);
    }
    return 0 ;
}
```

- マイコン側でのループバック
- 動作が確認できれば、ハード・ソフト両方とも正常な動作と言える

課題

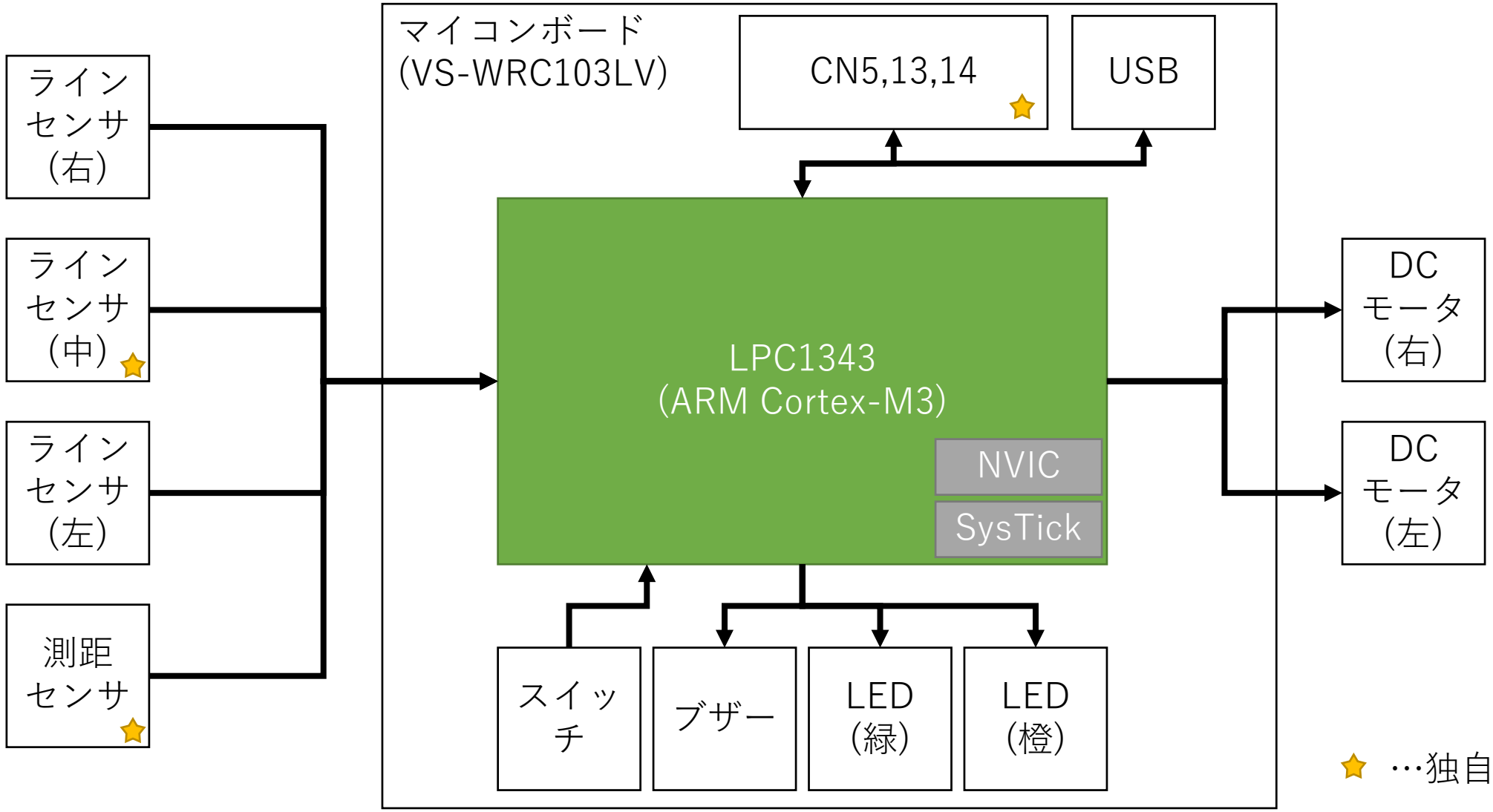
- Kadai12_01.c
  - Kadai11_02.cにラインセンサ3つと測距センサの値をprintfで出力するコードを追加し、XBeeによる無線通信でPCのTeraterm上で確認できる事を確認せよ。
- Kadai12_02.c
  - シリアル通信で文字列を送信するWindowsアプリ「RTC LTC App」に合わせて機能を実装せよ
  - Kadai11_01.cをベースに改変してください

課題(続き)



- 基本概念
  - ボタンを押すと(x)の'x'の1文字がシリアル通信でPCからマイコンへ転送
  - マイコンからprintfした文字列はPCが受信後にLogに表示
- Kadai11_1.cの時点でモータ制御は実装済み(a～g, A～D)
- 課題順序
  - LED制御(I,j,l,m)を実装
  - Sensors(n)ボタンを実装
    - 3つのラインセンサと測距センサの値を返し、Logに表示します
  - Ex dockのボタンの機能を実装
- 注意点
  - Buzzer(k)とShow LCD(q)は未実装として下さい

13:割込み入門

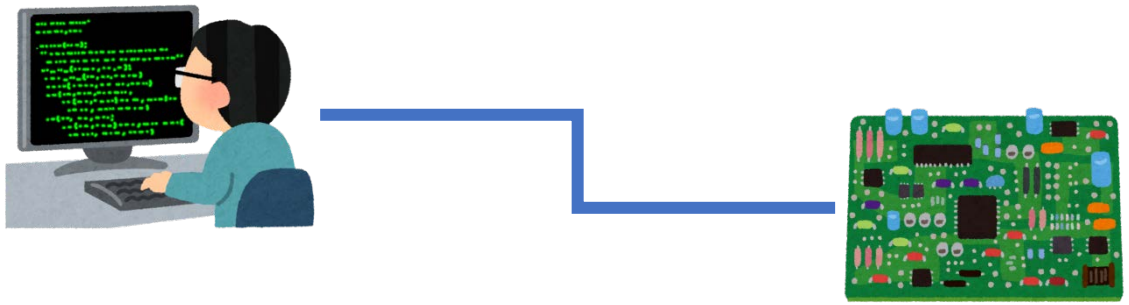

マイコンボード
(VS-WRC103LV)
CN5,13,14
USB
ライン
センサ
(右)
ライン
センサ
(中)
ライン
センサ
(左)
測距
センサ
LPC1343
(ARM Cortex-M3)
NVIC
SysTick
DC
モータ
(右)
DC
モータ
(左)
スイッ
チ
ブザー
LED
(緑)
LED
(橙)
…独自拡張の要素

割り込みとポーリング

- 従来の制御はポーリングによる制御
  - 定期的にイベントを監視して処理を行う
    - mainループでスイッチやセンサの値を監視
      → 条件を満たせばアクチュエータを制御
- ポーリングの問題点
  - 監視の周期がのびるとイベントの発生から検出までに遅延
  - 最終的なアクチュエータの制御にも遅延
- 割り込みを用いてイベント発生時に処理を実施
  - イベントの発生時に、現在の処理を中断して発生したイベントに関する処理を実行

はじめに

- ある特定の処理を実行中に、優先度・緊急度の高い別の処理を行う必要のある場面がある。
- 1つのマイコンで複数の機能を並行して動作させる事は難しい。複数の機能間で密に結合しやすい。
- タイマ割り込みなどを活用する事で、機能を分離し疎結合なシステムに近づける事が可能である。
- また、外部入力割込みを活用することで、緊急度の高い処理を優先して処理を行うことが可能である。



お手軽な割込みSysTickを試す

- SysTickとは
  - ARM-CortexM系のマイコンには、SysTickと呼ばれるシステムタイマがコア内に存在します。
- 特徴
  - 24bitの自動再ロードダウンカウンタ
  - カウントダウンの終了時に割込み発生機能が付属
  - CMSIS-COREを活用すると、僅か2行でタイマ割込みの設定が可能
    - レジスタの細かい設定を簡略化！

タイマ割込みのメリット

- Mainループとは別に並行して周期的な処理を実現できる
  - Mainループでブザーでメロディ演奏、タイマ割込み内でライントレース。といった事も可能
    - 複数の機能を1つのマイコンで並行して動作させる場面に便利
  - 割込みではあるが、ポーリングとして活用する場面が多い
- 時間間隔が正確な周期的な処理を実現可能
  - Mainループでは分岐や繰り返しの回数次第で、ループ一周にかかる処理時間が変化しやすい
  - タイマ割込みは基本的に設定した周期で呼び出される
    - 割り込み待ちや、タイマ関数の内部処理が設定した周期以上に時間が掛かった場合は除く
  - バラツキを避け、一定時間ごとにセンサ値を取得したい計測機器やデータロガーなどの製品の実装時に有用である

例題：rei13_01.c

```
int main(void) {
    unsigned char result = 0;

    init_led();

    SystemCoreClockUpdate();
    result = SysTick_Config(SystemCoreClock / 10);// 100mse周期
    if(result == 1){
        // 失敗
        set_led_green(LED_ON);
        set_led_orange(LED_ON);
while(1);
}

    while(1){
        set_led_green(LED_ON);
        wait_ms(250);
        set_led_green(LED_OFF);
        wait_ms(250);
    }
    return 0 ;
}

void SysTick_Handler(void){
    static unsigned char flg = 0;

    if(flg == 0){
        set_led_orange(LED_ON);
        flg = 1;
    }else{
        set_led_orange(LED_OFF);
        flg = 0;
    }
}
```

- 割り込みを使う時に必要な記述
  - 割り込みの初期化処理
  - 割り込み関数
  - 割込み関数の割り込みベクタテーブルへの登録
- SysTickの割込みを活用する場合は…
  - 1は既に専用の初期化関数が存在する為、大幅に簡略化
  - 2は自身で作成
  - 3はcr_startup_lpc13xx.c内にて登録済み

例題：rei13_02.c

```c
int main(void)
{
    init_led();
    init_TIMER32_1_IRQHandler();//タイマ割込み初期化

   while(1){
       set_led_orange(LED_ON);
       wait_ms(250);
       set_led_orange(LED_OFF);
       wait_ms(250);
   }

    return 0;
}

// 10msec周期のタイマ割込みの初期化
void init_TIMER32_1_IRQHandler(){    // 略    }

// 10msec周期のタイマ割込み
void TIMER32_1_IRQHandler(void) {
    static unsigned char flg = 0;

    if(flg == 0){
        set_led_green(LED_ON);
        flg = 1;
    }else{
        set_led_green(LED_OFF);
        flg = 0;
    }

    LPC_TMR32B1->IR=0x08; //タイマ割込みのフラグOFF
}
```

- SysTickと同様の処理を32bitタイマで実現した場合の事例

- 割り込みを使う時に必要な記述
  - 割り込みの初期化処理
  - 割り込み関数
  - 割込み関数の割り込みベクタテーブルへの登録
- 32bitタイマの割込みを活用する場合は…
  - **1は自身で作成**
  - 2は自身で作成
  - 3はcr_startup_lpc13xx.c内にて登録済み

割り込みの初期化処理(32bitタイマ)

```
// 10msec周期のタイマ割込みの初期化
void init_TIMER32_1_IRQHandler()
{
    NVIC_EnableIRQ(TIMER_32_1_IRQn);        //32bitタイマ割込みの有効化

    // 10msec周期のタイマ割込みを設定
    LPC_SYSCON->SYSAHBCLKCTRL |=0x400;      // Timer32B1 Turn ON
    LPC_TMR32B1->PR =7200-1;                // 分周設定 ：10kHz(Max 32bit dec:4294967295)
    LPC_TMR32B1->MCR =0x600;                // MR3 as Period of 32B1 and interrupt
    LPC_TMR32B1->MR0 =100;                  //
    //LPC_TMR32B1->MR3 =100;                // カウンタマッチ：10k/100=100Hz
    LPC_TMR32B1->MR3 =1000;                 // カウンタマッチ：10k/1000=10Hz
    LPC_TMR32B1->TCR =2;                    // カウンタリセット&停止
    LPC_TMR32B1->TCR =1;                    // カウントスタート
}
```

- タイマ割込みを行う為に必要なレジスタを設定する
  - 周期に関わるのは**PR**と**MR3**レジスタ
- Systickの設定関数の中身も上記のような処理が記述されている

タイマユニットの活用

- ARM-Cortex系のマイコンであればSysTickの活用が便利ですが、こちらは24bitタイマが1本だけとなります。
- LPC1343には16bitタイマが2本と、32bitタイマが2本が内蔵
  - 16bitタイマ→モータ制御に使用済み
  - 32bitタイマ→2本共に未使用
    - Rei13_02.cでは32bitタイマの0,1の2つのうちの1を利用

課題

- Kadai13_01.c
  - Rei13_01.cを改変し、Systickの割込み周期を25msec周期に変更してください。確認はオシロで行う事
- Kadai13_02.c
  - 確認後、1msec周期に変更してください。確認はオシロで行う事
- Kadai13_03.c
  - 確認後、1msec周期はそのままにLEDの点滅周期は100msec周期にしてください。(ON時間は50msec, OFF時間は50msec)

課題②

- [発展]Mainループ側の処理で距離センサが10cm以内に物体があると反応したら、タイマ割込み側でLEDを点灯させる。というプログラムを実現しましょう

- Kadai13_04.c：stop watchの実現
  - StopWatch関数群を作成し、以下の処理に掛かる時間を計測してprintfで出力してください。

```
printf(“Hello World!!¥n”);
```

ブザー制御とタイマ割込み

- 圧電ブザーには自励式と他励式の2種類があり、今回実装されているのは他励式になる
  - 自励式：発振回路を内蔵。電圧印加で鳴動
  - 他励式：発振回路は非搭載。外部よりパルスを与える事で鳴動
- 他励式の場合、パルスの周期で音程を制御が可能。
  - 加えてデューティ比でボリュームを若干変更可能
- マイコン内蔵のタイマによるPWM機能はデューティ比の容易な変更は可能なものの、周期の変更は手間である為、タイマ割込みを活用して実装する
- 今回は、ボードの開発元のサンプルコードを移植して使用する

例題：rei13_03.c

```
int main(void)
{
    init_button();
    LPC_GPIO1->DIR |= 0x0100;

    //ループ
    while(1){
        if(!get_button_black()){
            LPC_GPIO1->DATA |= 0x0100;
            wait_ms(1);
            LPC_GPIO1->DATA &= ~0x0100;
            wait_ms(1);
        }
    }

    return 0;
}
```

- 最もシンプルな構成で制御する場合、Main関数からブザーに繋がるPIO1_8をON/OFF制御する事で鳴動させることが可能である
- しかし、Mainループ内で一定周期を保持する必要がある為、他のHWの制御を入れ辛い
  - そこで、タイマ割込みを活用したブザーの制御を行う

ブザー制御関数群

- 以下の2つのファイルをプロジェクトに追加すること
  - buzzer.h
  - buzzer.c
- 制御関数は以下の通り
  - void init_buzzer(void);
    - ブザー制御を行う為にGPIOおよび、タイマの初期化を行う
  - void set_buzzer(unsigned char pitch, unsigned char vol);
    - ブザーの音程(pitch)および、ボリューム(vol)の設定
  - void start_buzzer(void);
    - ブザーの鳴動開始。停止関数を実行するまで鳴動。
  - void stop_buzzer(void);
    - ブザーの鳴動停止。
  - unsigned char is_buzzer(void);
    - ブザーが鳴動状態であるか確認。戻り値が0の時は非鳴動、1の時は鳴動。

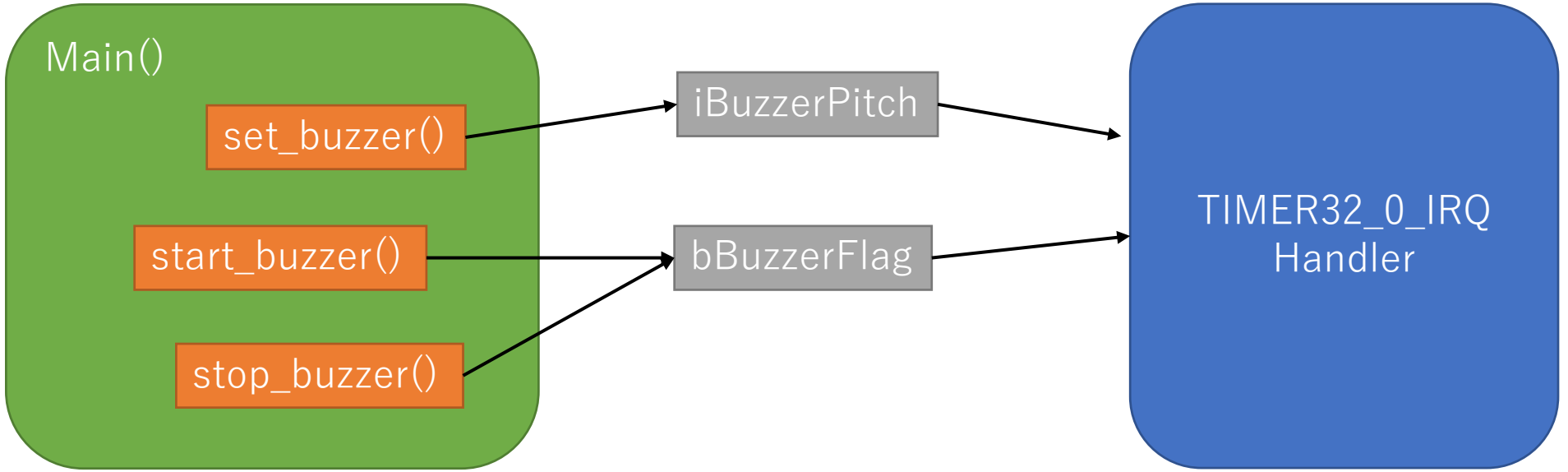
例題：rei13_04.c

```
#ifdef __USE_CMSIS
#include "LPC13xx.h"
#endif

// 略
#include "buzzer.h"

int main(void)
{
    init_button();
    init_buzzer();

    //ループ
    while(1){
        if(!get_button_black()){
            set_buzzer(C1, 32);
            start_buzzer();
        }else{
            stop_buzzer();
        }
    }

    return 0;
}
```

- SW1を押下時にブザーが鳴動するプログラム
- start_buzzer()やstop_buzzer()などのブザーの制御関数とタイマ割込み関数とを大域変数で橋渡ししている設計

割込み関数とMain関数との連携・橋渡し

- 割込み関数とMain関数との間でデータのやりとりや、処理の通知などを行う方法の良い事例が今回のブザー関数群である
- 基本的に大域変数(グローバル変数)を活用する
  - データ、フラグなどを用いて、割込み関数とMain関数間のやりとりに利用
  - Main関数から直接、グローバル変数を触らせるのではなく、関数で制限をかける形が好ましい設計になる



課題

- Kadai13_05.c
  - Rei13_04.cを改造して、ボタンを押したらド・レ・ミ・ファ・ソ・ラ・シ・ドを順番に鳴らしてください。1つ音の長さは500msecとします。
- Kadai13_06.c　：電子テルミンの実現
  - 測距センサより障害物との距離を取得し、距離に応じて音階を変化させたブザー音を鳴らしてください。
  - ただし、ブザーを鳴らす条件として「SW1押下時のみ」とします。

以上	未満	出力する音階
	10.0cm	ド
10.0cm	15.0cm	レ
15.0cm	20.0cm	ミ
20.0cm	25.0cm	ファ
25.0cm	30.0cm	ソ
30.0cm	35.0cm	ラ
35.0cm	40.0cm	シ
40.0cm		ド

[Appendix] Main関数も割込みの一種？

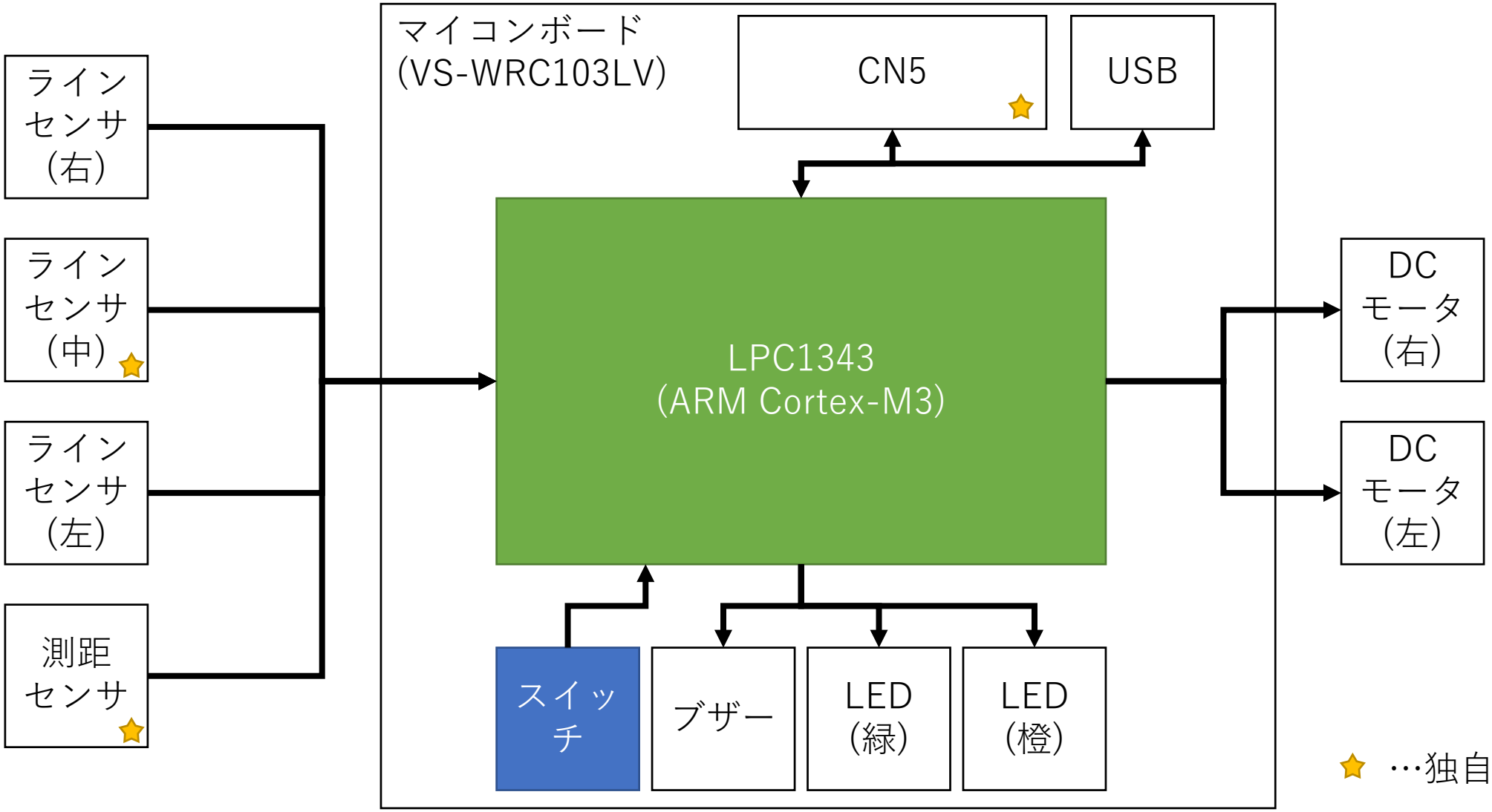
- 皆さんが何気なく使用しているMain関数ですが、一体、どこから呼び出されているのでしょうか？
  - Main関数は「リセット割込み」が起因の処理になる
- リセット割込みとは？
  - マイコンに電源を投入した時 or リセットピンに信号を入れた時に発生する割込み
  - 多くのマイコンは、このリセット割込みから処理がスタートする
    1. 電源投入 or リセット信号入力
    2. リセット割込みの発生
    3. スタートアップルーチンの動作
       1. スタックの設定やプログラムに必要なデータの設定、ハードウェアの初期化など
       2. 今回の環境の場合は、cr_startup_lpc13xx.cの「ResetISR関数」がスタートアップルーチン相当に該当
    4. Main関数の呼び出し

割り込みを使う時の注意点

- 割り込みを活用してシステム設計する際には、いくつか考慮しておくべき事があります。すべてを網羅的に挙げる事は難しいですが、いくつか代表的なポイントを列挙しておきます。

  - **共通資源に対する排他制御**
    - Mainループと割込み関数から同じハードウェアに対してアクセスをかける事は避けるor 競合すること無いよう注意して制御する
  - **割り込み内で「長い処理」は避ける**
    - 割り込みが発生している間、基本的に他の処理は実行されない為、割込み内で「長い処理」が発生すると、他の機能に大きな影響を及ぼす可能性がある
      - 事例）Mainループでライントレース処理中に、タイマ割込み内でscanf関数を実行するなど
      - 「長い処理」の代表例）printf系の関数、wait系の関数など
  - **CPUリソースの配分を意識する**（主にタイマ割込み利用時）
    - 10msec周期で発生するタイマ割込みの中で、9msecかかる処理を行うと他の割込みやMainループの処理を実行する時間が無くなってしまう。

14:外部入力割込みの活用



外部入力割込みとは？

- マイコンの外部入力割込み対応のピンに立ち上がり/立下りの信号を入力することで発生する割込み
  - 通常、マイコンの特定のピンのみ外部入力割込みとして利用できることが大半である
- LPC1343の場合、全GPIOピン全てが割込みピンとして活用可

例題：rei14_01.c

```c
int main(void)
{
    init_led();
    init_irq_button_black();
    init_syscall();

    while(1){
        if(push_count % 2 == 0){
            set_led_orange(LED_ON);
            printf("%d¥n", push_count);
        }else{
            set_led_orange(LED_OFF);
            printf("%d¥n", push_count);
        }
    }
    return 0;
}

void init_irq_button_black(void){
    LPC_GPIO0->DIR &= ~0x0002;
    LPC_GPIO0->IS &= ~(1 << 1);     // エッジ検出設定
    LPC_GPIO0->IBE &= ~(1 << 1);    // GPIOnIEV設定
    LPC_GPIO0->IEV |= (1 << 1);     // 立ち上がりエッジでトリガ
    LPC_GPIO0->IE |= (1 << 1);      // PIO0_1ポートの割り込みマスク無し
    NVIC_EnableIRQ(EINT0_IRQn);
}

void PIOINT0_IRQHandler(void) {
    wait_ms(10);// チャタリング対策．割り込み内でのwaitは本来は好ましくない
    push_count++;

    // 割り込みロジッククリア
    if (LPC_GPIO0->MIS & (1 << 1)) {
        LPC_GPIO0->IC |= (1 << 1);
    }
}
```

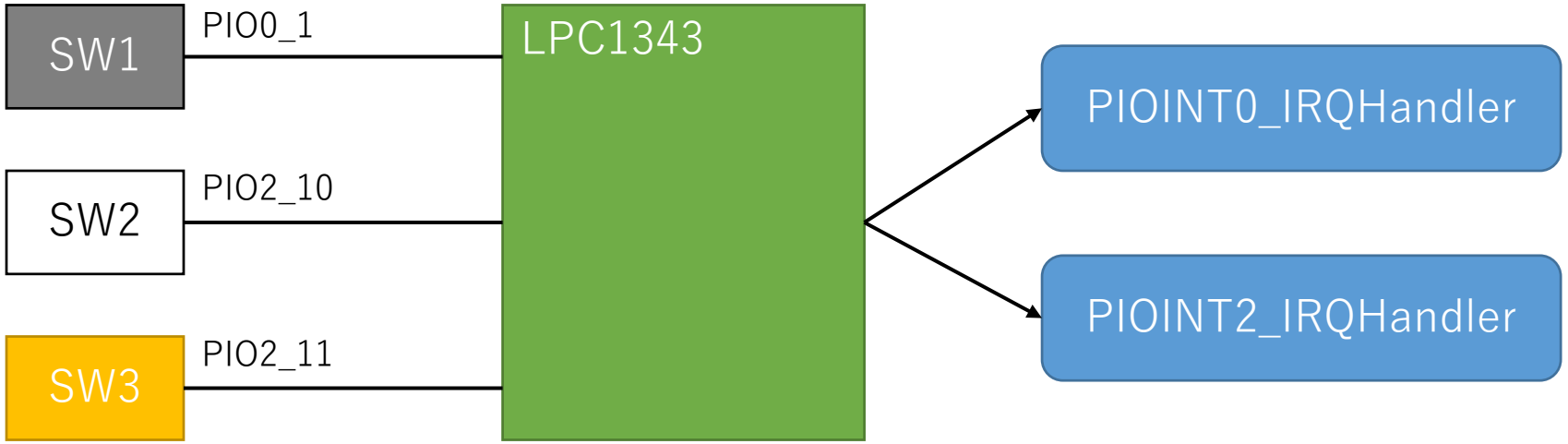
- 外部入力割り込みのサンプル
  - モーメンタリスイッチのオルタネイトスイッチ化
  - SW1を押下すると、LEDが点灯。もう1度押下するとLEDが消灯

スイッチのチャタリング問題

- 機械接点のスイッチを利用すると、必ず発生する問題
- スイッチの接点を切り替えるタイミングで、機械接点の影響から必ずON/OFFが繰り返されるタイミングが発生する
  - この現象を「チャタリング」と呼ぶ

ポート別の割込み関数

- LPC1343の外部入力割込みの割込み関数はポートの各ピン別ではなく、ポートごとに割り当てられる。
- よって、ポート0の0bit目で入った割込みでも7bit目で入った割込みでも同一の割込み関数へ処理が遷移する



[Appendix] GPIOによるオルタネイト化

例題14-02

```
int main(void){
    unsigned char last_state;
    unsigned char now_state;

    init_led();
    init_button();
    init_syscall();

    now_state = get_button_black();
    last_state = now_state;

    while(1){
        last_state = now_state;
        now_state = get_button_black();
        if(last_state == 0 && now_state == 1){       // 立ち上がりを検出
            wait_ms(10);
            push_count++;
        }

        if(push_count % 2 == 0){
            set_led_orange(LED_ON);
            printf("%d¥n", push_count);
        }else{
            set_led_orange(LED_OFF);
            printf("%d¥n", push_count);
        }
    }

    return 0;
}
```

- 外部入力割込みを使用せず、GPIO入力によりモーメンタリをオルタネイト化する方法
- 簡易な実装は左のコードの通り。
  - 前回の値と今回の値をチェックするのがポイント
  - 今回の事例では信号の立ち上がりを検出
- チャタリング対策はwaitによる簡易実装

[Appendix] GPIOによるオルタネイト化

例題14-03

```c
int main(void){
    unsigned char last_state;
    unsigned char now_state;
    unsigned char up_edge_flag = 0;
    unsigned char chatt_count = 0;

    init_led();
    init_button();
    init_syscall();

    now_state = get_button_black();
    last_state = now_state;

    while(1){
        // 立ち上がりを検出
        //(ボタン押下後の指離しを想定)
        last_state = now_state;
        now_state = get_button_black();
        if(last_state == 0 && now_state == 1){
            up_edge_flag = 1;
        }

        // チャタリング後の信号の安定の検出
        if(up_edge_flag == 1 && now_state ==
1){
            chatt_count++;
        }else {
            up_edge_flag = 0;
            chatt_count = 0;
        }
        // チャタリング終了後に(1回だけ)行う処理
        // (10msec間、信号が安定していれば
        //  チャタリングが終わったと判断)
        if(10 <= chatt_count){
            push_count++;

            // フラグやカウンタをクリア
            up_edge_flag = 0;
            chatt_count = 0;
        }

        if(push_count % 2 == 0){
            set_led_orange(LED_ON);
            printf("%d¥n", push_count);
        }else{
            set_led_orange(LED_OFF);
            printf("%d¥n", push_count);
        }

        wait_1ms();// チャタリングチェック用の周期
    }

    return 0;
}
```

- 前段のGPIOによるモーメンタリをオルタネイト化する方法の改良版。チャタリング対策を丁寧に記述した場合。
- チャタリングによる信号のバタつきを監視。バタつきがなくなった時点で「スイッチが切り替わった」と判断。

課題

- Kadai14_01.c：モード切替
  - SW1の押下回数でモードを切り替えながら動作するプログラム
    - 0回押下：電源投入直後は橙LEDを100ms周期で点滅
    - 1回押下：電子テルミンモードへ遷移
    - 2回押下：ライントレースモードへ遷移
    - 3回押下：0回目(電源投入直後)へ戻る。
- Kadai14_02.c
  - 01の課題をSW1からドックの白ボタンへ変えて実現する

関連リンクほか

- ビュートローバーARM（ヴイストン社）
  - https://www.vstone.co.jp/products/beauto_rover/index.html
  - 本テキストに関する問い合わせについて、ヴイストン社は受け付けておりませんのでご注意ください。
- LPCXPresso(NXP社)
  - http://www.nxp-lpc.com/lpc_boards/lpcxpresso/
- ARMマイコンによる組込みプログラミング入門(オーム社)
  - http://shop.ohmsha.co.jp/shop/shopdetail.html?brandcode=000000001461&search=ARM%A5%DE%A5%A4%A5%B3%A5%F3

- Special Thanks
  - SSEST5(Summer School on Embedded System Technologies)