クラシックサイズマイクロマウス
（プリント基板タイプ）
Pi:Co Classic 3

パート4 『取扱説明書』

ソフトウェア解説編

株式会社アールティ

2017 年 4 月

目次

1. ソフトウェアの解説 3

1.1. STEP1:LED を光らせよう 4
1.1.1. ポート方向レジスタ 5
1.1.2. 端子機能制御レジスタ 6
1.1.3. プロテクトの解除 8
1.1.4. ポートモードレジスタ 8
1.2. STEP2:ブザを鳴らそう 10
1.3. STEP3_1:スイッチを使おう 12
1.4. STEP3_2:モードを作ろう 15
1.5. STEP4:モードごとにブザの音程を変えよう 18
1.6. STEP5:センサの値を見よう 34
1.7. STEP6:モータを回そう 1 43
1.8. STEP6:モータを回そう 2 45
1.9. STEP7:フィードバック制御をしよう 55
1.10. STEP8:迷路を走破しよう 60
1.11. STEP9:迷路情報、センサ値を記憶しよう 62

2. 故障かな? と思ったら 65

3. 開発のヒント 66

3.1. 基板ポート一覧 66
3.2. 基板回路図 68
3.3. プログラムの詳解 69
3.4. シリアル通信(SCI)について 70

4. 著作権について 70

5. ソフトウェアについて 70

6. 改訂履歴 70

7. お問い合わせ 71

1. ソフトウェアの解説

Pi:Co Classic3 パート 2 取扱い説明書ハードウェア製作編、パート 3 取扱い説明書環境構築編でプログラムを書き込める環境ができました。ここでは 11 個のサンプルプログラムの解説を行います。サンプルプログラムの解説を読みながら、実際に Pi:Co Classic 3 を動作させて、解説どおりに動くことを確認してください。確認ができたら次は自分でプログラムを書き加えていくことにより、思い通りに動作させることができるようになります。

ここで注意して頂きたいことがあります。プログラムの入ったフォルダの名前はなるべく日本語を使わないでください。特に括弧”( )“とスペース” “を付けますと CS+でコンパイル時にエラーが出る場合があります。またプログラム名も同様です。なるべくプロジェクト名、プログラム名、それらの保存フォルダの名前には英語で、スペースの代わりにアンダーバを使ってください。

ここでは STEP ごとに以下の流れで進めていきます。

概要

大まかにその STEP で何をおこなうプログラムなのかを解説します。

プログラムソース

実際のプログラムの中からその STEP での重要なところを掲載します。ライブラリ化しているファイル名については末尾に一覧を掲載します。

動作確認と解説

プログラムを実際に書き込んでみてどのように動作するのかを解説していきます。

1.1. Step1：LED を光らせよう

概要

STEP1 はデバック用でもあり、モード確認用でもある LED を光らせてみましょう。このプログラムは LED を点滅させるプログラムになっています。RX マイコンのポート設定についても説明するので、付属の RX63N グループ、RX631 グループ ユーザーズマニュアル ハードウェア編(以降 RX631 ハードウェアマニュアルと省略)を参照しながらお読みください。

step1_1_LED_wait メインプログラムソース

```
#include "iodefine.h"
#include "portdef.h"
#include "init.h"
void main(void)
{
	int i;
	init_all();//LED の繋がっているポートを出力に設定
	while(1) //無限ループ
	{
		LED0 = LED1 = LED2 = LED3 = 1;  //LED 点灯
		for(i = 0; i < 0x7fffff; i++);		//時間待ち(空ループ)
		LED0 = LED1 = LED2 = LED3 = 0;  //LED 消灯
		for(i = 0; i < 0x7fffff; i++);		//時間待ち(空ループ)
	}
}
```

動作確認と解説

CS+を起動して「C:¥20170407Documents¥Sample_Program¥step1_1_LED_wait」フォルダ内の「PiCoClassic3.mtpj」を開き、ビルドします。ビルド方法は、別紙の PiCo_Classic_3 パート 3_取扱説明書_環境構築編 ver1.1 の 1.3 章サンプルプログラムのビルドを見直してください。そして生成した「C:¥20170407Documents¥Sample_Program¥step1_1_LED_wait¥DefaultBuild」フォルダ内の「PiCoClassic3.mot」というファイルを、Renesas Flash Programmer(これ以降 RFP と省略)で書き込みを行います。次からの STEP でもプロジェクトのフォルダの名前が違うだけで書き込む手順は同じです。

ここで初めてプログラムを修正して書き込みますが、ソフトウェア編の解説はC言語を知っている前提で話を進めていきます。最後の迷路を走行するサンプルプログラムでは、構造体(struct)、共用体(union)、ポインタ(*p)、列挙型(enum)、ビットフィールド(メンバ名 ： ビット数)、ビット演算など普段使わないようなところまで使用しています。

したがって、分からない事柄がありましたら随時C言語の参考書などを読んでください。出来る限りわかりやすくなるように努力しますのでよろしくお願いします。

では解説に移ります。

まずSTEP1ではLEDを点滅させていますがLEDを思い通りに光らせるにはマイコンのポートの出力と入力の切り替えと機能の切り替えができるようにならなければなりません。RX631では、以下の4点の機能のレジスタを設定する必要があります。

1.ポート方向レジスタ
2.端子機能制御レジスタ
3.書き込みプロテクトレジスタ
4.ポートモードレジスタ

1つ目のポート方向レジスタは、ポートの入出力を選択します。2つ目の端子機能制御レジスタは、ポートの機能を選択します。最近のマイコンは高機能であり、1つのポートにいくつもの機能を兼用として使用しています。たとえば、ポー

ト2のB1では、PWM、SCI(シリアルコミュニケーションインターフェース)、USBなどの機能が兼用されており、どれか1つを選択して使用することができます。3つ目の書き込みプロテクトレジスタは、2つ目の端子機能制御レジスタをプログラムのミスや暴走により安易に変更できないように保護する機能です。最後のポートレジスタは、汎用入出力ポートとして使用するか機能ポートとして使用するかの選択をします。各レジスタの設定について少し詳しく解説します。

1.1.1. ポート方向レジスタ

ポート方向レジスタは、ポートの入力か出力を決めるレジスタです。入出力を設定するには、RX631 ハードウェアマニュアルの" 21 I/O ポート"を参照してください。RX631 の初期設定では、全てのポートは入力(=0)になっています。ポート方向レジスタ(PDR)のレジスタで入出力を変更します。ポートを出力にしたい場合は、そのビットを"1"にします。

RX63Nグループ、RX631グループ　21. I/Oポート

21.3　レジスタの説明

21.3.1　ポート方向レジスタ（PDR）

アドレス　PORT0.PDR 0008 C000h、PORT1.PDR 0008 C001h、PORT2.PDR 0008 C002h、PORT3.PDR 0008 C003h、PORT4.PDR 0008 C004h、PORT5.PDR 0008 C005h、PORT6.PDR 0008 C006h、PORT7.PDR 0008 C007h、PORT8.PDR 0008 C008h、PORT9.PDR 0008 C009h、PORTA.PDR 0008 C00Ah、PORTB.PDR 0008 C00Bh、PORTC.PDR 0008 C00Ch、PORTD.PDR 0008 C00Dh、PORTE.PDR 0008 C00Eh、PORTF.PDR 0008 C00Fh、PORTG.PDR 0008 C010h、PORTJ.PDR 0008 C012h

	b7	b6	b5	b4	b3	b2	b1	b0
	B7	B6	B5	B4	B3	B2	B1	B0
リセット後の値	0	0	0	0	0	0	0	0

ビット	シンボル	ビット名	機能	R/W
b0	B0	Pm0方向制御ビット	0：入力（入力ポートとして機能） 1：出力（出力ポートとして機能）	R/W
b1	B1	Pm1方向制御ビット		R/W
b2	B2	Pm2方向制御ビット		R/W
b3	B3	Pm3方向制御ビット		R/W
b4	B4	Pm4方向制御ビット		R/W
b5	B5	Pm5方向制御ビット		R/W
b6	B6	Pm6方向制御ビット		R/W
b7	B7	Pm7方向制御ビット		R/W

m=0 ～ 9、A ～ G、J

LEDはポートAのB6(LED0)、ポートBのB0(LED1)、ポートAのB4(LED2)、ポートAのB0(LED3)にアサイン(割り当て)されています。RX631ハードウェアマニュアルではポートAのB6をPA6と表記しています。ポート方向レジスタの設定はこんな感じなります。init.cファイルの44行目あたりにあります。

```
PORTA.PDR.BIT.B6 = IO_OUT;//LED0
PORTB.PDR.BIT.B0 = IO_OUT;//LED1
PORTA.PDR.BIT.B4 = IO_OUT;//LED2
PORTA.PDR.BIT.B0 = IO_OUT;//LED3
```

このおまじないのような記述は、"iodefine.h" にレジスタにアクセスする記述が書いてあり、その正体は構造体(struct)です。また、同じレジスタに対してbyteアクセスとbitアクセスが可能のように共用体(union)でも定義されています。

"PORTA.PDR.BIT.B6 = 1;"ではないのか?と気が付いた方もいると思いますが、後からプログラムを見たときに、"1"は入力だったのか?出力だったのか?を分かり易くするため、static_parameters.hで宣言(define)しています。

```
#define IO_OUT    (1)
#define IO_IN     (0)
```

1.1.2. 端子機能制御レジスタ

端子機能制御レジスタは、使用する機能を選択するレジスタです。詳細は、RX631ハードウェアマニュアルの”22. マルチファンクションピンコントローラ(MPC)”を参照してください。このページには各ポートにどのような機能があるか説明されていますので基本的にポートの機能を設定する時はここを見ます。

ポートの機能を設定するときはマルチファンクションピンコントローラの端子機能制御レジスタで行います。リセット直後は、汎用入出力ポートとして機能します。今回光らせたいLEDはポートAのB6(LED0)、ポートBのB0(LED1)、ポートAのB4(LED2)、ポートAのB0(LED3)を使用していますが、機能ポートではく、汎用入出力ポートとして使用しているため設定する必要がありません。

もしポートAのB6を設定するのであれば、MPC.PA6PFS.BIT.PSEL = 0x0;となります。

RX63Nグループ、RX631グループ　　22. マルチファンクションピンコントローラ（MPC）

表22.25　64ピンLQFP、64ピンTFLGA　端子入出力機能レジスタ設定

PSEL[4:0]ビット設定値	端子				
	PA0	PA1	PA3	PA4	PA6
00000b（初期値）	Hi-Z				
00001b	MTIOC4A	MTIOC0B	MTIOC0D	MTIC5U	MTIC5V
00010b	—	MTCLKC	MTCLKD	MTCLKA	MTCLKB
00011b	TIOCA0	TIOCB0	TIOCD0	TIOCA1	TIOCA2
00100b	—	—	TCLKB	—	—
00101b	—	—	—	TMRI0	TMCI3
00110b	PO16	PO17	PO19	PO20	PO22
00111b	—	—	—	—	POE2#
01010b	—	SCK5	RXD5 SMISO5 SSCL5	TXD5 SMOSI5 SSDA5	—
01011b	—	—	—	—	CTS5# RTS5# SS5#
01101b	SSLA1	SSLA2	—	SSLA0	MOSIA

—：設定しないでください。

端子機能制御レジスタの該当箇所を”io.define.h”から抜粋したもの(2830行目あたり)が下記になります。struct st_mpcでマルチファンクションコントローラのレジスタを構造体で定義しています。構造体は、上から順番にメモリマップにアサインされるためマルチファンクションコントローラのレジスタのメモリマップの順番どおりに記述されています。C言語の場合、メモリの管理はリンカによってアサインされるため、開発者は何も考えずメモリ空間を使っていますが、レジスタにアクセスする場合、決まった番地にアクセスする必要があるため、構造体の先頭のアドレスを決める必要があります。それをしているのが、#define MPCのところです。これをすることで、PFCSEのレジスタが0x0008_C100に割り当

```
struct st_mpc {
        union {
                unsigned char BYTE;
                struct {
                        unsigned char CS7E:1;
                        unsigned char CS6E:1;
                        unsigned char CS5E:1;
                        unsigned char CS4E:1;
                        unsigned char CS3E:1;
                        unsigned char CS2E:1;
                        unsigned char CS1E:1;
                        unsigned char CS0E:1;
                } BIT;
        } PFCSE;
…
        union {
                unsigned char BYTE;
                struct {
                        unsigned char :3;
                        unsigned char PSEL:5;
                } BIT;
        } PA6PFS;
…
#define MPC             (*(volatile struct st_mpc       __evenaccess *)0x8C100)
```

てられ、PA6PFSが、0x0008_C196に割り当てられます。

0008 C100h	MPC	CS出力許可レジスタ	PFCSE	8	8	2～3PCLKB	2ICLK	MPC	750
0008 C102h	MPC	CS出力端子選択レジスタ0	PFCSS0	8	8	2～3PCLKB	2ICLK		751
0008 C103h	MPC	CS出力端子選択レジスタ1	PFCSS1	8	8	2～3PCLKB	2ICLK		752
0008 C104h	MPC	アドレス出力許可レジスタ0	PFAOE0	8	8、16	2～3PCLKB	2ICLK		753
0008 C105h	MPC	アドレス出力許可レジスタ1	PFAOE1	8	8、16	2～3PCLKB	2ICLK		754
0008 C106h	MPC	外部バス制御レジスタ0	PFBCR0	8	8、16	2～3PCLKB	2ICLK		755
0008 C107h	MPC	外部バス制御レジスタ1	PFBCR1	8	8、16	2～3PCLKB	2ICLK		756
0008 C10Eh	MPC	イーサネット制御レジスタ	PFENET	8	8	2～3PCLKB	2ICLK		757
0008 C114h	MPC	USB0制御レジスタ	PFUSB0	8	8	2～3PCLKB	2ICLK		758
0008 C115h	MPC	USB1制御レジスタ	PFUSB1	8	8	2～3PCLKB	2ICLK		759
0008 C11Fh	MPC	書き込みプロテクトレジスタ	PWPR	8	8	2～3PCLKB	2ICLK		720

表5.1　I/Oレジスタアドレス一覧（３３／４６）

アドレス	モジュールシンボル	レジスタ名	レジスタシンボル	幅（ビット）	アクセスサイズ	アクセスサイクル数		関連機能	参照ページ
						ICLK ≥ PCLKの場合	ICLK < PCLKの場合		
0008 C17Bh	MPC	P73端子機能制御レジスタ	P73PFS	8	8	2～3PCLKB	2ICLK	MPC	734
0008 C17Ch	MPC	P74端子機能制御レジスタ	P74PFS	8	8	2～3PCLKB	2ICLK		734
0008 C17Dh	MPC	P75端子機能制御レジスタ	P75PFS	8	8	2～3PCLKB	2ICLK		734
0008 C17Eh	MPC	P76端子機能制御レジスタ	P76PFS	8	8	2～3PCLKB	2ICLK		734
0008 C17Fh	MPC	P77端子機能制御レジスタ	P77PFS	8	8	2～3PCLKB	2ICLK		734
0008 C180h	MPC	P80端子機能制御レジスタ	P80PFS	8	8	2～3PCLKB	2ICLK		735
0008 C181h	MPC	P81端子機能制御レジスタ	P81PFS	8	8	2～3PCLKB	2ICLK		735
0008 C182h	MPC	P82端子機能制御レジスタ	P82PFS	8	8	2～3PCLKB	2ICLK		735
0008 C183h	MPC	P83端子機能制御レジスタ	P83PFS	8	8	2～3PCLKB	2ICLK		735
0008 C186h	MPC	P86端子機能制御レジスタ	P86PFS	8	8	2～3PCLKB	2ICLK		735
0008 C187h	MPC	P87端子機能制御レジスタ	P87PFS	8	8	2～3PCLKB	2ICLK		735
0008 C188h	MPC	P90端子機能制御レジスタ	P90PFS	8	8	2～3PCLKB	2ICLK		736
0008 C189h	MPC	P91端子機能制御レジスタ	P91PFS	8	8	2～3PCLKB	2ICLK		736
0008 C18Ah	MPC	P92端子機能制御レジスタ	P92PFS	8	8	2～3PCLKB	2ICLK		736
0008 C18Bh	MPC	P93端子機能制御レジスタ	P93PFS	8	8	2～3PCLKB	2ICLK		736
0008 C190h	MPC	PA0端子機能制御レジスタ	PA0PFS	8	8	2～3PCLKB	2ICLK		737
0008 C191h	MPC	PA1端子機能制御レジスタ	PA1PFS	8	8	2～3PCLKB	2ICLK		737
0008 C192h	MPC	PA2端子機能制御レジスタ	PA2PFS	8	8	2～3PCLKB	2ICLK		737
0008 C193h	MPC	PA3端子機能制御レジスタ	PA3PFS	8	8	2～3PCLKB	2ICLK		737
0008 C194h	MPC	PA4端子機能制御レジスタ	PA4PFS	8	8	2～3PCLKB	2ICLK		737
0008 C195h	MPC	PA5端子機能制御レジスタ	PA5PFS	8	8	2～3PCLKB	2ICLK		737
0008 C196h	MPC	PA6端子機能制御レジスタ	PA6PFS	8	8	2～3PCLKB	2ICLK		737
0008 C197h	MPC	PA7端子機能制御レジスタ	PA7PFS	8	8	2～3PCLKB	2ICLK		737
0008 C198h	MPC	PB0端子機能制御レジスタ	PB0PFS	8	8	2～3PCLKB	2ICLK		739
0008 C199h	MPC	PB1端子機能制御レジスタ	PB1PFS	8	8	2～3PCLKB	2ICLK		739
0008 C19Ah	MPC	PB2端子機能制御レジスタ	PB2PFS	8	8	2～3PCLKB	2ICLK		739
0008 C19Bh	MPC	PB3端子機能制御レジスタ	PB3PFS	8	8	2～3PCLKB	2ICLK		739
0008 C19Ch	MPC	PB4端子機能制御レジスタ	PB4PFS	8	8	2～3PCLKB	2ICLK		739
0008 C19Dh	MPC	PB5端子機能制御レジスタ	PB5PFS	8	8	2～3PCLKB	2ICLK		739

1.1.3. プロテクトの解除

RX631 のリセット後、機能の設定の保護のため、端子機能制御レジスタに直接アクセスできない仕組みになっています。端子機能制御レジスタ(マルチファンクションピンコントローラ)にアクセスするには、プロテクトを解除する必要があります。

22.2 レジスタの説明

パッケージの違いにより、端子がないレジスタ、ビットは予約です。該当するビットに値を書く場合は、リセット後の値を書いてください。

22.2.1 書き込みプロテクトレジスタ（PWPR）

アドレス 0008 C11Fh

	b7	b6	b5	b4	b3	b2	b1	b0
	B0WI	PFSWE	—	—	—	—	—	—
リセット後の値	1	0	0	0	0	0	0	0

ビット	シンボル	ビット名	機能	R/W
b5-b0	—	予約ビット	読むと“0”が読めます。書く場合、“0”としてください	R/W
b6	PFSWE	PFSレジスタ書き込み許可ビット	0：PFSレジスタへの書き込みを禁止 1：PFSレジスタへの書き込みを許可	R/W
b7	B0WI	PFSWEビット書き込み禁止ビット	0：PFSWEビットへの書き込みを許可 1：PFSWEビットへの書き込みを禁止	R/W

PFSWE ビット（PFS レジスタ書き込み許可ビット）

PFSWE ビットを“1”にしたときのみ、PmnPFS レジスタに対する書き込みが許可されます。

PFSWE ビットを“1”にする場合は、B0WI ビットに“0”を書いた後、PFSWE ビットを“1”にしてください。

B0WI ビット（PFSWE ビット書き込み禁止ビット）

B0WI ビットを“0”にしたときのみ、PFSWE ビットに対する書き込みが許可されます。

一度、MPC.PWPR.BIT.B0WI=0;を行った後、MPC.PWPR.BIT.PFSWE=1;をすることでプロテクトが解除されます。プロテクトをするときは、MPC.PWPR.BYTE=0x80;の1回で終了です。設定する場合はこんな感じになります。

```
MPC.PWPR.BIT.B0WI=0;
MPC.PWPR.BIT.PFSWE=1;
MPC.PA6PFS.BIT.PSEL=0x0;
MPC.PB0PFS.BIT.PSEL=0x0;
MPC.PA4PFS.BIT.PSEL=0x0;
MPC.PA0PFS.BIT.PSEL=0x0;
MPC.PWPR.BYTE=0x80;
```

1.1.4. ポートモードレジスタ

最後にポートモードレジスタ(PMR)があります。RX631ハードウェアマニュアルの”21.3.4 ポートモードレジスタ”を参照してください。このポートモードレジスタは、汎用ポートと機能ポートの切り替えをするものです。RX631の初期設定では、全てのPortは設定を変えない限り汎用入出力Portに設定されているので今回は設定を省いています。もし設定するのであれば、PORTA.PMR.BIT.B6= 0x0;となります。

これらのポートの設定はinit.cのファイルに記述されています。init.cには上記の説明のほかに、電池の電圧を監視するシステムの設定をしてあります。**10Vより低くなったらブザが鳴るように設定されており、電池を過放電しないようにしています。**必要ないからと消さないでください。

Main関数に戻り、while文の中で適当な時間でLEDを点滅させて、while(1)として無限ループさせています。

```
while(1) //無限ループ
{
        LED0 = LED1 = LED2 = LED3 = 1;  //LED 点灯
        for(i = 0; i < 0x7fffff; i++);          //時間待ち(空ループ)
        LED0 = LED1 = LED2 = LED3 = 0;  //LED 消灯
        for(i = 0; i < 0x7fffff; i++);          //時間待ち(空ループ)
}
```

while文内のLED0,LED1,LED2,LED3の表記がありますが、これはincludeしているportdef.hにてわかりやすいように宣言しています。portdef.hを開いてみてください。例えばLED0の場合

```
#define LED0        (PORTB.PODR.BIT.B0)
```

とBit単位で宣言しています。

それではプログラムを書き込んでみましょう。電源を入れたらLEDは光りましたか？
うまく光らせることができたら光らせ方を変えて遊んでみても良いでしょう。

おさらい
汎用ポートして使用する場合は、入出力の設定をし、機能ポートして使用する場合、
端子機能制御レジスタの設定(MPC.xxPFS.BIT.xxx)→ポートモードレジスタの設定(PORTx.PMR.BIT.xxx)
の順にやっていきましょう。

1.2. Step2：ブザを鳴らそう

概要

STEP2 では装飾用兼デバック用のブザを鳴らしてみましょう。ブザを鳴らす場合も LED のように出力すれば良いと思うかもしれませんが、実はブザはそれでは鳴りません。電流が流れるだけで鳴るものもありますが、Pi:Co Classic 3 に使っているブザは信号を入力しないと鳴りません。なので、STEP2 のプログラムでは STEP1 で行った LED の点滅を行っていた while 文を利用して鳴らします。

Step2_Buzzer_wait メインプログラムソース

```
void main(void)
{
        int i;
        init_all();                 //ブザで使用するポートを出力に設定
        while(1)//無限ループ
        {
                BUZZER = 1;                     //ブザに1を出力
                for(i = 0; i < 0x3000; i++);    //時間待ち(空ループ)
                BUZZER = 0;                     //ブザに0を出力
                for(i = 0; i < 0x3000; i++);    //時間待ち(空ループ)
        }
}
```

動作確認と解説

このサンプルの CS+プロジェクトは「C:¥20170407Documents¥Sample_Program¥step2_Buzzer_wait」フォルダ内の「PiCoClassic3.mtpj」です。それをビルドし、RFP を使って、「C:¥20170407Documents¥Sample_Program¥step2_Buzzer_wait¥DefaultBuild」フォルダ内の「PiCoClassic3.mot」を書き込みます。

while文の中身を見ていきましょう。

```
        while(1)//無限ループ
        {
                BUZZER = 1;                     //ブザに1を出力
                for(i = 0; i < 0x3000; i++);    //時間待ち(空ループ)
                BUZZER = 0;                     //ブザに0を出力
                for(i = 0; i < 0x3000; i++);    //時間待ち(空ループ)
        }
```

STEP1 と同様の処理ですが、この処理を電気的に考えると 3.3V と 0V が交互に切り替えられています。図にすると下記のようになります。 BUZZER はポート B の B3 にアサインしています。portdef.h にて” #define BUZZER (PORTB.PODR.BIT.B3)”で定義しています。

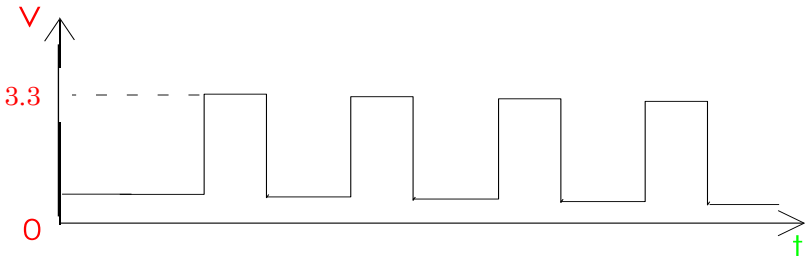


図　ポートから出ている電圧の変化

デジタル的に言うと0と1の繰り返しを矩形波（くけいは）といいます。このようにプログラムで矩形波を出力し、ブザを鳴らします。なぜ、矩形で音が鳴るかという、音の実体は、空気などの媒体が周期的かつ連続的に運動したときに、聴覚に伝わったものです。周期的かつ連続的に運動するものであれば、正弦波(sin)、余弦波(cos)でなくても、三角波、ノコギリ波でも音としてなります。音がなったら、待ち時間を変更して音の高さを変更して遊んでみてください。

それ以外はSTEP1と一緒なので、特に説明することはありません。

1.3. Step3_1：スイッチを使おう

概要

STEP3 ではモードの作成の際のモード選択用スイッチを使えるようにします。今回は今までの復習も兼ねて LED とブザを連動して動かしてみましょう。

Step3_1_Switch メインプログラムソース

```
void main(void)
{
	int i;
	int state_r, state_c, state_l;		//状態を示す変数
	state_r = state_c = state_l = 0;
	init_all();				//LEDの繋がっているポートを出力に設定
	while(1)  //無限ループ
	{
		while(SW_R & SW_C & SW_L);//何れかのスイッチが押されるまで待つ
		if(SW_R == 0)			//右スイッチが押されていた時
		{
			state_r++;		//状態を示す変数を更新
			if(state_r > 1)  state_r = 0;//状態を示す変数が0か1になるようにする
			if(state_r == 0)	//変数によってLEDを点灯/消灯させる
			{
				LED3 = 0;
			}else{
				LED3 = 1;
			}
		}

		if(SW_C == 0)			//中央スイッチが押されていた時
		{
			state_c++;		//状態を示す変数を更新
			if(state_c > 1)state_c = 0;//状態を示す変数が0か1になるようにする
			if(state_c == 0)	//変数によってLEDを点灯/消灯させる
			{
				LED1 = LED2 = 0;
			}else{
				LED1 = LED2 = 1;
			}
```

```
                }
                if(SW_L == 0)                   //左スイッチが押されていた時
                {
                        state_l++;              //状態を示す変数を更新
                        if(state_l > 1)state_l = 0;//状態を示す変数が0か1になるようにする
                        if(state_l == 0)        //変数によってLEDを点灯/消灯させる
                        {
                                LED0 = 0;
                        }else{
                                LED0 = 1;
                        }
                }
                for(i = 0; i < 0xffff; i++);        //チャタリング除去用時間待ち空ループ
                while( !(SW_R & SW_C & SW_L) );//スイッチが全て離されるまで待つ
                for(i = 0; i < 0xffff; i++);        //チャタリング除去用時間待ち空ループ
        }
}
```

動作確認と解説

このサンプルの CS+プロジェクトは「C:¥20170407Documents¥Sample_Program¥step3_1_Switch」フォルダ内の「PiCoClassic3.mtpj」です。それをビルドし、RFP を使って、「C:¥20170407Documents¥Sample_Program¥step3_1_Switch¥DefaultBuild」フォルダ内の「PiCoClassic3.mot」を書き込みます。

SW_R、SW_C、SW_Lはそれぞれポートの B2、ポートCのB3、ポート3のB1にアサインしています。portdef.hにて”#define SW_R (PORTC.PIDR.BIT.B2)、#define SW_C (PORTC.PIDR.BIT.B3)、#define SW_L (PORT3.PIDR.BIT.B1)”で定義しています。スイッチはSTEP1、STEP2のLED、ブザと違い入力になりますので、PIDR(ポート入力データレジスタ、詳細はRX631ハードウェアマニュアルの”21.3.3 ポート入力データレジスタ(PIDR)を参照)を経由してスイッチの状態を確認します。

まず34行目でスイッチの状態がわかるよう変数を下記のように用意します。

```
int state_r, state_c, state_l;   //状態を示す変数
```

スイッチの回路は抵抗にてプルアップしておりでスイッチが押されていない時には3.3Vが入り、スイッチが押されたらグラウンドにつながるようになっています。マイコンに取り込まれるのは3.3Vと0Vではなく、デジタルになるので、押されていない時は”1”、押されたときは”0”となります。

36行目でそれぞれのスイッチの状態を初期化(=0)しています。

```
state_r = state_c = state_l = 0;
```

43 行目の while 文で 3 つのスイッチのどれかが押されるまで待ちます。

```
while(SW_R & SW_C & SW_L); //何らかのスイッチが押されるまで待つ
```

45行目以下のif文でどのスイッチが押されたかの場合分けをしています。下記は右スイッチが押された場合の例です。

```
if(SW_R == 0)                    //右スイッチが押されていた時
{
        state_r++;               //状態を示す変数を更新
        if(state_r > 1)    state_r = 0;//状態を示す変数が0か1になるようにする
        if(state_r == 0)         //変数によってLEDを点灯/消灯させる
        {
                LED3 = 0;
        }else{
                LED3 = 1;
        }
}
```

上から見て右のスイッチが押されたら変数state_rに1が足され一回目はLEDが点灯し、二回目からは48行目のif文によってstate_rは0になるので、LEDは消灯します。他のスイッチを押すと同様の処理が行われ、光るLEDが変わるようになっています。

84行目から下の文はスイッチのチャタリング防止の処理になっています。チャタリングとは、スイッチを押す際に微細な機械的振動でオンオフが一瞬の間に複数回行われてしまうことをいいます。これにより、誤作動が起こる恐れがあります。

```
for(i = 0; i < 0xffff; i++)   ;          //チャタリング除去用時間待ち空ループ
while( !(SW_R & SW_C & SW_L) );          //スイッチが全て離されるまで待つ
for(i = 0; i < 0xffff; i++)   ;          //チャタリング除去用時間待ち空ループ
```

これでSTEP3_1のサンプルプログラムは終わりですが、ブザを使っていません。ここまでの成長を見るということで自分でブザを鳴らすプログラムを追加してみましょう。スイッチが押されたときの場合分けのif文内でSTEP2でのブザのパルスを作るプログラムを追加し、スイッチが離されたらbreakしてパルスを作るwhile文から抜け出すようにすればいいだけですので是非やってみてください。1つ注意点があります。電池の電圧が10V以下になったらブザがなる仕組みにしてあります。このままでは、STEP2のプログラムをコピーしても動きません。その理由は、ブザの端子の設定を汎用入出力ポートとではなく、機能ポートとして設定してあるからです。まず、機能ポートの設定を汎用入力ポートに切り替えてからSTEP2のブザのサンプルプログラムを参考にブザがなるようにしてみてください。わからなかったらSTEP1を確認しましょう。

1.4. Step3_2：モードを作ろう

概要

今回はモードを作るのですが、まずその際に必要な準備をしています。変数 mode を追加してその変数を足す、または引いて各モードに移れるようにしています。

Step3_2_Switch_Changing_mode メインプログラムソース

```
void _LED(int led_data)
{
	LED0 = led_data&0x01;
	LED1 = (led_data>>1)&0x01;
	LED2 = (led_data>>2)&0x01;
	LED3 = (led_data>>3)&0x01;
}

void main(void)
{
	int i,j,k,mode;
	mode = 1;		//初期化
	LED0 = 1;		//初期化
	init_all();		//LEDの繋がっているポートを出力に設定
	while(1)		//無限ループ
	{
		while(SW_R & SW_C & SW_L); //何れかのスイッチが押されるまで待つ
		if(SW_R == 0)			//右スイッチが押されていた時
		{
			mode--;		//右スイッチが押された時モードを減少させる
			if(mode < 1)		//モードの下限値を制限
			{
				mode = 1;
			}
		}
		if(SW_C == 0)				//中央スイッチが押されていた時
		{
			for(k = 0; k < mode; k++){		//modeの回数、ブザを鳴らす
				for(j = 0; j < 0x3ff; j++)
				{
```

```
                                        BUZZER = 1;
                                        for(i = 0; i < 0xfff; i++)    ;
                                        BUZZER = 0;
                                        for(i = 0; i < 0xfff; i++)    ;
                                }
                                for(j = 0; j < 0x1ff; j++)
                                {
                                        for(i = 0; i < 0xfff; i++);
                                }
                        }
                }

                if(SW_L == 0)               //左スイッチが押されていた時
                {
                        mode++;             //左スイッチが押されたらモードを増加させる
                        if(mode > 15)               //モードの上限値を制限
                        {
                                mode = 15;
                        }
                }
                _LED(mode);
                for(i = 0; i < 0xffff; i++)  ;//チャタリング除去用時間待ち空ループ
                while( !(SW_R & SW_C & SW_L) );//スイッチが全て離されるまで待つ
                for(i = 0; i < 0xffff; i++)  ;//チャタリング除去用時間待ち空ループ
        }
}
```

動作確認と解説

このサンプルの CS+プロジェクトは「C:¥20170407Documents¥Sample_Program¥step3_2_Switch_changing_mode」フォルダ内の「PiCoClassic3.mtpj」です。それをビルドし、RFP を使って、「C:¥20170407Documents¥Sample_Program¥step3_2_Switch_changing_mode¥DefaultBuild」フォルダ内の「PiCoClassic3.mot」を書き込みます。

Pi:Co Classic 3ではデバック用LEDが4つあり15通りの光らせ方を作れるのでモードの上限は15に設定しています。ここで気をつけなければいけないのは設定したモードの下限と上限を超えないようにすることです。もし、作ったモードが設定した数字を超えたり、下回った数字になったりすると誤作動を起こす恐れがあります。

52行目のif文で右のスイッチが押されるとmodeを-1されるようになっています。
また、55行目のif文でモードが1より少なくなることを防いでいます。

```
if(SW_R == 0)                   //右スイッチが押されていた時
{
        mode--;                 //右スイッチが押された時モードを減少させる
        if(mode < 1)            //モードの下限値を制限
        {
                mode = 1;
        }
  }
```

同様に78行目のif文で左のスイッチが押されるとmodeを+1進めます。
81行目のif文で15を超えるのを防いでいます。

```
if(SW_L == 0)                   //左スイッチが押されていた時
{
        mode++;                 //左スイッチが押されたらモードを増加させる
        if(mode > 15)           //モードの上限値を制限
        {
                mode = 15;
        }
  }
```

ここまでできましたらSwitch文で各モードに分けた処理を追加すれば良いだけです。
このサンプルは、モード数だけブサがなるようにしています。

次のStepとして、RX631ハードウェアマニュアルの”23. マルチファンクションタイマパルスユニット2(MTU2a)”（以下MTU2と省略）と”28. コンペアマッチタイマ(CMT)”(以下CMTと省略)の機能を使って、モードごとに違った周波数を出してブザを鳴らしましょう。
それでは実際に各モードの内容をつくり、音を鳴らしてみましょう。次のSTEP4に進んでください。

1.5. Step4：モードごとにブザの音程を変えよう

概要

今までのプログラムはスイッチを押すごとにモードが切り替わり、モードごとに LED が点灯していましたが、実際に動かす楽しさを重視するため、メインプログラムだけの説明でした。今回は動かしているマイコンがどのような設定になっているかも簡単に説明していきます。また前回の終わりにも書きましたが今回、新しく RX631 の MTU2 と CMT を使ってブザを鳴らす周波数を設定します。周波数はクロックとも言います。この MTU2 は設定が複雑なので説明が長くなります。諦めずに根気強く読んでいきましょう。

MTU2 とは？

様々な機能を持ったユニットです。このユニットはタイマ割り込みや、PWM 出力、トグル出力や位相計数などの機能があります。今回は PWM 出力に設定します。PWM については別紙のクラシックサイズマイクロマウス Pi:Co Classic3「部品説明+用語集」の PWM(Pulse Width Modulation)制御を参照してください。PWM の音源、電源回路、モータ制御などに使用されています。簡単な例として、PWM 制御で LED の明るさを変化させるサンプルプログラムを用意しました。C:¥20170407Documents¥Sample_Program¥step1_2_LED_PWM/PiCOClassic3.mtpj を開き、ビルドします。生成した PiCoClassic3.mot を RFP で書き込んで確認してみてください。10 段階で明るさが変わるようにしています。

CMT とは？

設定した周期ごとに割り込みを発生させることができます。割り込みとは main()など、通常の処理を中断して別の処理を割り込ませることです。割り込ませた処理が終わると元の処理を続けます。マイコンでは良く使われる機能です。今回は割り込みの周期を 1ms に設定します。

まずは各機能が設定されている初期化プログラムの init.c を見ていきます。

プログラムソース 1：init.c

```
#include "init.h"
#include "static_parameters.h"
#include "iodefine.h"

void init_clock(void)
{
        SYSTEM.PRCR.WORD = 0xa50b;          //クロックソース選択の保護の解除
        SYSTEM.PLLCR.WORD = 0x0F00;         /* PLL 逓倍×16 入力1分周*/
        SYSTEM.PLLCR2.BYTE = 0x00;          /* PLL ENABLE */
        SYSTEM.PLLWTCR.BYTE   = 0x0F;       /* 4194304cycle(Default) */
        SYSTEM.SCKCR.LONG = 0x21C21211;
        SYSTEM.SCKCR2.WORD = 0x0032;
        SYSTEM.BCKCR.BYTE = 0x01;           /* BCLK = 1/2 */
        SYSTEM.SCKCR3.WORD = 0x0400;        //PLL回路選
}
```

```
void init_io(void)
{
        //LED
        PORTB.PDR.BIT.B0 = IO_OUT;//LED0
        PORTA.PDR.BIT.B6 = IO_OUT;//LED1
        PORTA.PDR.BIT.B4 = IO_OUT;//LED2
        PORTA.PDR.BIT.B0 = IO_OUT;//LED3
        PORTB.PDR.BIT.B1 = IO_OUT;//BLED0
        PORTA.PDR.BIT.B3 = IO_OUT;//BLED1
        //Sensor
        PORT5.PDR.BIT.B4 = IO_OUT;//SLED_L
        PORT0.PDR.BIT.B5 = IO_OUT;//SLED_R
        PORT2.PDR.BIT.B7 = IO_OUT;//SLED_FL
        PORTB.PDR.BIT.B5 = IO_OUT;//SLED_FR

        //MOT_POWER
        PORT1.PDR.BIT.B5 = IO_OUT;//motor Enable
        //MOT_CWCCW
        PORTC.PDR.BIT.B5 = IO_OUT;//Rmotor
        PORTC.PDR.BIT.B6 = IO_OUT;//Lmotor
        //Buzzerのポートの初期化
        PORTB.PDR.BIT.B3 = IO_OUT;
        BUZZER = 1;
        MPC.PWPR.BIT.B0WI=0;
        MPC.PWPR.BIT.PFSWE=1;
        MPC.PB3PFS.BIT.PSEL=1;      //MTIOC0A
        MPC.P17PFS.BIT.PSEL=1;      //MTIOC3Aとして使用
        MPC.PE2PFS.BIT.PSEL=1;      //MTIOC4A
        MPC.PE0PFS.BIT.ASEL=1;      //A/D  電源
        MPC.PE1PFS.BIT.ASEL=1;      //A/D SEN_FR
        MPC.P44PFS.BIT.ASEL=1;      //A/D SEN_FL
        MPC.P46PFS.BIT.ASEL=1;      //A/D SEN_R
        MPC.P42PFS.BIT.ASEL=1;      //A/D SEN_L
        MPC.PWPR.BYTE=0x80;
        PORTB.PMR.BIT.B3=1; //SP PWM
        PORT1.PMR.BIT.B7=1;  //右PWM
        PORTE.PMR.BIT.B2=1; //左PWM
```

```
        PORTE.PMR.BIT.B0=1; //A/D
        PORTE.PMR.BIT.B1=1; //A/D
        PORT4.PMR.BIT.B4=1; //A/D
        PORT4.PMR.BIT.B6=1; //A/D
        PORT4.PMR.BIT.B2=1; //A/D
}

void init_cmt(void)
{
        SYSTEM.PRCR.WORD = 0xA502;
        MSTP(CMT0) = 0;
        MSTP(CMT1) = 0;
        SYSTEM.PRCR.WORD = 0xA500;

        //CMT0は制御割り込み用タイマとして使用
        CMT0.CMCR.BIT.CKS=1;        // PCLK/32 1.5MHz
        CMT0.CMCR.BIT.CMIE=1;       //割り込みを許可
        CMT0.CMCNT=0;               //カウンターのクリア
        CMT0.CMCOR=1500-1;          //1kHz

        IEN(CMT0,CMI0) = 1;   //割り込み要求を許可
        IPR(CMT0,CMI0) = 15;  //割り込み優先度 15が最高
        IR(CMT0,CMI0)=0;      //割り込みステータフラグをクリア

        //CMT1はセンサ制御用タイマとして使用
        CMT1.CMCR.BIT.CKS=1;        // PCLK/32 1.5MHz
        CMT1.CMCR.BIT.CMIE=1;       //割り込みを許可
        CMT1.CMCNT=0;               //カウンターのクリア
        CMT1.CMCOR=(1500/4)-1;      //4kHz

        IEN(CMT1,CMI1) = 1;   //割り込み要求を許可
        IPR(CMT1,CMI1) = 14;  //割り込み優先度を次点に設定
        IR(CMT1,CMI1)=0;      //割り込みステータフラグをクリア

        CMT.CMSTR0.BIT.STR0=1;      //カウントスタート
        CMT.CMSTR0.BIT.STR1=1;      //カウントスタート

}
```

```
void init_mtu(void)
{
        SYSTEM.PRCR.WORD = 0xA502;
        MSTP(MTU) = 0;                  //MTUモジュールON
        SYSTEM.PRCR.WORD = 0xA500;
        MTU.TSTR.BYTE=0;                //タイマ動作ストップ
        //スピーカ用MTU
        MTU0.TCR.BIT.CCLR=2;            //PWM TGRBのコンペアマッチでTCNTクリア
        MTU0.TCR.BIT.TPSC=1;            //PCLK/4 12MHz
        MTU0.TMDR.BIT.MD=2;             //PWM1
        MTU0.TIORH.BIT.IOA=5;           //コンベアマッチてlow初期はhigh
        MTU0.TIORH.BIT.IOB=2;           //コンベアマッチでhigh
        MTU0.TGRA = 6000;               //1kHz
        MTU0.TGRB = (12000-1);
        //右モータ用MTU設定
        MTU3.TCR.BIT.TPSC=1;            //PCLK/4 12MHz
        MTU3.TCR.BIT.CCLR=1;            //PWM TGRAのコンペアマッチでTCNTクリア
        MTU3.TIORH.BIT.IOA=1;           //初期出力0ンペアマッチ0出力
        MTU3.TIORH.BIT.IOB=2;           //初期出力0コンペアマッチ1出力
        MTU3.TGRA = 12000;
        MTU3.TGRB = 50;
        MTU3.TGRC = 48000;
        MTU3.TMDR.BIT.MD=2;             //PWM1
        MTU3.TMDR.BIT.BFA = 1;          //バッファモードに設定
        MTU3.TIER.BIT.TGIEB = 1;        //GRBコンペアマッチでの割り込み許可
        IEN(MTU3,TGIB3) = 1;            //割り込み要求を許可
        IPR(MTU3,TGIB3) = 13;           //割り込み優先度を次点に設定
        IR(MTU3,TGIB3)=0;               //割り込みステータフラグをクリア
        //左モータ用MTU設定
        MTU.TOER.BIT.OE4A=1;            //MTU出力端子を出力許可する
        MTU4.TCR.BIT.TPSC=1;            //PCLK/4 12MHz
        MTU4.TCR.BIT.CCLR=1;            //PWM TGRAのコンペアマッチでTCNTクリア
        MTU4.TIORH.BIT.IOA=1;           //初期出力0ンペアマッチ0出力
        MTU4.TIORH.BIT.IOB=2;           //初期出力0コンペアマッチ1出力
        MTU4.TGRA = 12000;
        MTU4.TGRB = 50;
        MTU4.TGRC = 48000;
```

```
        MTU4.TMDR.BIT.MD=2;         //PWM1
        MTU4.TMDR.BIT.BFA = 1;      //バッファモードに設定
        MTU4.TIER.BIT.TGIEB = 1;    //GRBコンペアマッチでの割り込み許可
        IEN(MTU4,TGIB4) = 1;        //割り込み要求を許可
        IPR(MTU4,TGIB4) = 12;       //割り込み優先度を次点に設定
        IR(MTU4,TGIB4)=0;           //割り込みステータフラグをクリア
        MTU.TSTR.BIT.CST0 = 0;      //タイマストップ
        MTU.TSTR.BIT.CST3 = 0;      //タイマストップ
        MTU.TSTR.BIT.CST4 = 0;      //タイマストップ
}
void init_all(void)
{
        init_clock();       //CPUの動作周波数を設定
        init_io();          //I/O(Input / Output)ポートを設定
        init_cmt();         //CMT(Compare Match Timer)を設定
        init_mtu();         //MTU(Multi Function Timer Pulse Unit)
        init_adc();         //ADC(Analog Digital Converter)初期化
}
```

動作確認と解説

このサンプルの CS+プロジェクトは「C:¥20170407Documents¥Sample_Program¥step4_Buzzer_do_re_mi_PWM」フォルダ内の「PiCoClassic3.mtpj」です。それをビルドし、RFP を使って、「C:¥20170407Documents¥Sample_Program¥step4_Buzzer_do_re_mi_PWM¥DefaultBuild」フォルダ内の「PiCoClassic3.mot」を書き込みます。

それでは説明していきます。

1. CPU クロックの設定

```
SYSTEM.PRCR.WORD = 0xa50b;   //クロックソース選択の保護の解除
SYSTEM.PLLCR.WORD = 0x0F00; /* PLL 逓倍×16 入力 1 分周 (12.000MHz * 16 = 192MHz)*/
SYSTEM.PLLCR2.BYTE = 0x00;    /* PLL ENABLE */
SYSTEM.PLLWTCR.BYTE       = 0x0F;                    /* 4194304cycle(Default) */
SYSTEM.SCKCR.LONG = 0x21C21211;
                    //FCK1/4 ICK1/2 BCLK 停止 SDCLK 停止 BCK1/4 PCLKA1/2 PCLKB1/4
SYSTEM.SCKCR2.WORD = 0x0032;          /* UCLK1/4 IEBCK1/4 */
SYSTEM.BCKCR.BYTE = 0x01;             /* BCLK = 1/2 */
SYSTEM.SCKCR3.WORD = 0x0400;//PLL 回路選択
```

この init_clock 関数で RX631 の CPU クロックの設定をしています。CPU クロックの数字が高ければ高いほどマイコンの処理速度は速くなります Pi:Co Classic 3 は、USB を使用する関係から内部クロックは 48MHz の倍数の 96MHz にしていますが、最大 100MHz で動作させることができます。マイコンに接続されているクロックは 12MHz です。つまり、内部で 8 逓倍しています。この逓倍の技術に PLL(Phase Locked Loop)が使われています。RX631 の起動時は 125kHz の低速オンチップオシレータにつながっています。125kHz でもマイクロマウスとしての動作は可能ですが、非常に動作が遅くなるため、クロックソースを PLL のクロックに切り替えて高速で動作させます。RX631 ハードウェアマニュアルの”9.クロック発生回路”を参照してください。システムクロックコントロールレジスタにより分周の設定、クロックソ

ースの選択をしますが、注 1 に PLL 選択時は 1 分周の設定は禁止されていることが記載されています。つまり、最低でも 2 分周にする必要があり、96MHz で動作させるには、PLL 後は 192MHz にする必要があります。

CPUや周辺モジュールのクロック設定一覧

クロック	分周、逓倍設定	接続されているモジュール
FCLK	4分周(192MHz/4=48MHz)	FlashIF
ICLK	2分周(192MHz/2=96MHz)	CPU、DMAC、DTC、ROM、RAM
BCLK	停止(4分周)	外部バスに供給するクロック
SDCLK	停止	SDRAMに供給するクロック
PCLKA	2分周(192MHz/2=96MHz)	ETHERC、EDMAC、DEU
PCLKB	4分周(192MHz/4=48MHz)	MTU、AD、SCI、I2C、CAN、SPI、PDC、DA
UCLK	4分周(192MHz/4=48MHz)	USB
IEBCLK	4分周(192MHz/4=48MHz)	IEBUS
PLL	16逓倍(12MHz×16=192MHz)	

9.2 レジスタの説明

9.2.1 システムクロックコントロールレジスタ（SCKCR）

アドレス 0008 0020h

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	FCK[3:0]				ICK[3:0]				PSTOP1	PSTOP0	—	—	BCK[3:0]			
リセット後の値	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	PCKA[3:0]				PCKB[3:0]				—	—	—	—	—	—	—	—
リセット後の値	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

ビット	シンボル	ビット名	機能	R/W
b3-b0	—	予約ビット	“0001b”を設定してください	R/W
b7-b4	—	予約ビット	“0001b”を設定してください	R/W
b11-b8	PCKB[3:0]	周辺モジュールクロックB（PCLKB）選択ビット(注1、注5)	b11 b8 0 0 0 0：1分周 0 0 0 1：2分周 0 0 1 0：4分周 0 0 1 1：8分周 0 1 0 0：16分周 0 1 0 1：32分周 0 1 1 0：64分周 上記以外は設定しないでください	R/W
b15-b12	PCKA[3:0]	周辺モジュールクロックA（PCLKA）選択ビット(注1、注5)	b11 b8 0 0 0 0：1分周 0 0 0 1：2分周 0 0 1 0：4分周 0 0 1 1：8分周	R/W

ビット	シンボル	ビット名	機能	R/W
b19-b16	BCK[3:0]	外部バスクロック（BCLK）選択ビット （注1、注2、注5）	b19 b16 0 0 0 0：1分周 0 0 0 1：2分周 0 0 1 0：4分周 0 0 1 1：8分周 0 1 0 0：16分周 0 1 0 1：32分周 0 1 1 0：64分周 上記以外は設定しないでください	R/W
b21-b20	—	予約ビット	読むと“0”が読めます。書く場合、“0”としてください	R/W
b22	PSTOP0	SDCLK端子出力制御ビット	0：SDCLK端子出力動作 1：SDCLK端子出力停止（High固定）	R/W
b23	PSTOP1	BCLK端子出力制御ビット（注3）	0：BCLK端子出力動作 1：BCLK端子出力停止（High固定）	R/W
b27-b24	ICK[3:0]	システムクロック（ICLK）選択ビット （注1、注2、注4、注5）	b27 b24 0 0 0 0：1分周 0 0 0 1：2分周 0 0 1 0：4分周 0 0 1 1：8分周 0 1 0 0：16分周 0 1 0 1：32分周 0 1 1 0：64分周 上記以外は設定しないでください	R/W
b31-b28	FCK[3:0]	FlashIFクロック（FCLK）選択ビット （注1、注4、注5）	b31 b28 0 0 0 0：1分周 0 0 0 1：2分周 0 0 1 0：4分周 0 0 1 1：8分周 0 1 0 0：16分周 0 1 0 1：32分周 0 1 1 0：64分周 上記以外は設定しないでください	R/W

注1.　PLL選択時は1分周は設定禁止です。
注2.　ICLKは外部バスクロックより低い周波数を設定しないでください。
注3.　外部バス有効時、BCLK端子と兼用しているP53は、I/Oポートとして使用できません。
注4.　低速動作モード2かつSCKCR3.CKSEL[2:0]ビットでサブクロック発振器選択時は、ICLKおよびFCLKの分周比は1分周のみ設定可能です。

このクロックの設定にはプロテクトがかかっていますので、プロテクトを解除してからクロックの設定をする必要があります。RX631ハードウェアマニュアルの”13.レジスタライトプロテクション機能”を参照してください。

13.1　レジスタの説明

13.1.1　プロテクトレジスタ（PRCR）

アドレス　0008 03FEh

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	PRKEY[7:0]								—	—	—	—	PRC3	—	PRC1	PRC0
リセット後の値	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

ビット	シンボル	ビット名	機能	R/W
b0	PRC0	プロテクトビット0	クロック発生回路関連レジスタへの書き込み許可 0：書き込み禁止 1：書き込み許可	R/W
b1	PRC1	プロテクトビット1	動作モード、消費電力低減機能、ソフトウェアリセット関連レジスタへの書き込み許可 0：書き込み禁止 1：書き込み許可	R/W
b2	—	予約ビット	読むと“0”が読めます。書く場合、“0”としてください	R/W
b3	PRC3	プロテクトビット3	LVD関連レジスタへの書き込み許可 0：書き込み禁止 1：書き込み許可	R/W
b7-b4	—	予約ビット	読むと“0”が読めます。書く場合、“0”としてください	R/W
b15-b8	PRKEY[7:0]	PRCキーコードビット	PRCRレジスタの書き換えの可否を制御します。 PRCRレジスタを書き換える場合、上位8ビットに“A5h”、下位8ビットに設定値を、16ビット単位で書いてください	R/(W) （注1）

注1.　書き込みデータは保持されません。

2. IO の設定

```
//Buzzerのポートの初期化
PORTB.PDR.BIT.B3 = IO_OUT;
BUZZER = 1;
MPC.PWPR.BIT.B0WI=0;
MPC.PWPR.BIT.PFSWE=1;
MPC.PB3PFS.BIT.PSEL=1;        //MTIOC0A
MPC.PWPR.BYTE=0x80;
PORTB.PMR.BIT.B3=1;           //SP PWM
```

この init_io 関数では、各ポートの入出力と機能の設定をしています。LED の設定に関しては STEP1 と同じなので省略します。STEP4 で初めて Buzzer を機能ポートして設定しています。

ポート B の Bit3 をブサのポートとしてアサインしています。この例では、PWM でブザ回路に信号を送り、音を鳴らすことをしています。その PWM は、周辺機能ポートとして設定する必要があり、69 行目の MPC.PB3PFS.BIT.PSEL=1;で周辺機能の MTU2 の出力(MTIOC0A)に設定し、80 行目の PORTB.PMR.BIT.B3=1;で周辺機能ポートに設定しています。PSEL=1 に関しては、RX631 ハードウェアマニュアルの”22.2.13 PBn 端子機能制御レジスタ (PBnPFS) (n=0～7)を参照してください。

最低限この設定だけで良いのですが、ブザを鳴らす/鳴らさないをタイマのカウンタのスタート/ストップで制御しているため、ストップするタイミングによってはブサ回路に電流が流れたままになります。それを回避するため、64 行目の PORTB.PDR.BIT.B3;で入出力を出力設定にした後、ブサ回路に電流を流さないように端子からの出力を”1”(ブサ回路のトランジスタはアクティブローです)に設定しています。

macro.h に定義している #define DISABLE_BUZZER PORTB.PMR.BIT.B3=0;MTU.TSTR.BIT.CST0=0 を使ってブサを停止するようにしています。この define によって、ブサを使用しないとき、周辺機能ポートから汎用入出力ポートに切り替えています。汎用入出力ポートに切り替えたとき、出力ポートに設定した”1”が有効となり、ブザのトランジスタが OFF する仕組みになっています。

3. CMT の設定

```
SYSTEM.PRCR.WORD = 0xA502;
MSTP(CMT0) = 0;
SYSTEM.PRCR.WORD = 0xA500;

//CMT0は制御割り込み用タイマとして使用
CMT0.CMCR.BIT.CKS=1;        // PCLK/32 1.5MHz
CMT0.CMCR.BIT.CMIE=1;       //割り込みを許可
CMT0.CMCNT=0;               //カウンターのクリア
CMT0.CMCOR=1500-1; //1kHz

IEN(CMT0,CMI0) = 1;   //割り込み要求を許可
IPR(CMT0,CMI0) = 15;  //割り込み優先度 15が最高
IR(CMT0,CMI0)=0;      //割り込みステータフラグをクリア

CMT.CMSTR0.BIT.STR0=1;      //カウントスタート
```

この init_cmt では、CMT の初期設定を行っています。RX631 ハードウェアマニュアルの”28. コンペアマッチタイマ(CMT)”を確認しながら進めましょう。

まずは CMT を動作するように設定します。今回使用する CMT のチャンネルは 0ch です。RX631 ハードウェアマニュアルの”11.2.2 モジュールストップコントロールレジスタ A(MSTPCRA)”（以降 MSTPCRA と省略)を見てみましょう。この MSTPCRA は RX631 の各機能の動作状態を設定するレジスタになっています。CMT0 のは bit15 の MSTPA15 を 0 にすることでモジュールストップを解除することができます。CS+に含まれる RX631 の iodefine.h で、モジュールストップコントロールレジスタ(MSTP)へアクセスを簡単にする細工が施されています。
SYSTEM.MSTPCRA.BIT.MSTPA15=0;するところを、MSTP(CMT0)=0;でモジュールストップを解除することができます。

このレジスタもプロテクトがかかっており、プロテクトを解除してから設定する必要があります。プロテクトに関するところは、RX631 ハードウェアマニュアルの”13. レジスタライトプロテクション機能”を参照してください。

11.2.2　モジュールストップコントロールレジスタ A（MSTPCRA）

アドレス 0008 0010h

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	ACSE	—	MSTPA29	MSTPA28	MSTPA27	—	—	MSTPA24	MSTPA23	—	—	—	MSTPA19	—	MSTPA17	—
リセット後の値	0	1	0	0	0	1	1	0	1	1	1	1	1	1	1	1

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	MSTPA15	MSTPA14	MSTPA13	MSTPA12	MSTPA11	MSTPA10	MSTPA9	—	—	—	MSTPA5	MSTPA4	—	—	—	—
リセット後の値	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

ビット	シンボル	ビット名	機能	R/W
b3-b0	—	予約ビット	読むと“1”が読めます。書く場合、“1”としてください	R/W
b4	MSTPA4	8ビットタイマ3、2（ユニット1）モジュールストップ設定ビット	対象モジュール：TMR3、TMR2 0：モジュールストップ状態の解除 1：モジュールストップ状態へ遷移	R/W
b5	MSTPA5	8ビットタイマ1、0（ユニット0）モジュールストップ設定ビット	対象モジュール：TMR1、TMR0 0：モジュールストップ状態の解除 1：モジュールストップ状態へ遷移	R/W
b8-b6	—	予約ビット	読むと“1”が読めます。書く場合、“1”としてください	R/W
b9	MSTPA9	マルチファンクションタイマパルスユニット2モジュールストップ設定ビット	対象モジュール：MTU（MTU0～MTU5） 0：モジュールストップ状態の解除 1：モジュールストップ状態へ遷移	R/W
b10	MSTPA10	プログラマブルパルスジェネレータ（ユニット1）モジュールストップ設定ビット	対象モジュール：PPG1 0：モジュールストップ状態の解除 1：モジュールストップ状態へ遷移	R/W
b11	MSTPA11	プログラマブルパルスジェネレータ（ユニット0）モジュールストップ設定ビット	対象モジュール：PPG0 0：モジュールストップ状態の解除 1：モジュールストップ状態へ遷移	R/W
b12	MSTPA12	16ビットタイマパルスユニット1（ユニット1）モジュールストップ設定ビット	対象モジュール：TPUユニット1（TPU6～TPU11） 0：モジュールストップ状態の解除 1：モジュールストップ状態へ遷移	R/W
b13	MSTPA13	16ビットタイマパルスユニット0（ユニット0）モジュールストップ設定ビット	対象モジュール：TPUユニット0（TPU0～TPU5） 0：モジュールストップ状態の解除 1：モジュールストップ状態へ遷移	R/W
b14	MSTPA14	コンペアマッチタイマ（ユニット1）モジュールストップ設定ビット	対象モジュール：CMTユニット1（CMT2、CMT3） 0：モジュールストップ状態の解除 1：モジュールストップ状態へ遷移	R/W
b15	MSTPA15	コンペアマッチタイマ（ユニット0）モジュールストップ設定ビット	対象モジュール：CMTユニット0 (CMT0、CMT1) 0：モジュールストップ状態の解除 1：モジュールストップ状態へ遷移	R/W

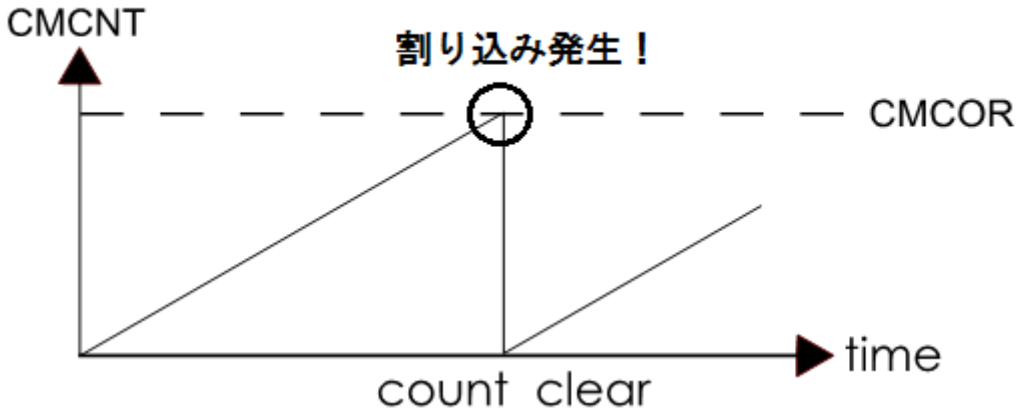
```
SYSTEM.PRCR.WORD = 0xA502;
MSTP(CMT0) = 0;
SYSTEM.PRCR.WORD = 0xA500;
```

これで CMT が動作する状態になりました。次に CMT の設定に移ります。

CMT の設定

```
CMT0.CMCR.BIT.CKS=1;        // PCLK/32 1.5MHz
CMT0.CMCR.BIT.CMIE=1;       //割り込みを許可
CMT0.CMCNT=0;               //カウンターのクリア
CMT0.CMCOR=1500-1;          //1kHz
```

RX631 ハードウェアマニュアルの”28.2.3 コンペアマッチタイマコントロールレジスタ(CMCR)”を見てください。このレジスタの 6bit 目の値を 1 にすることによりコンペアマッチ割り込みが許可されます。コンペアマッチとは、CMT 内でクロック(CKS で設定)をカウント(CMCNT)し、設定した値(CMCOR)とカウントが一致した状態のことを言います。ます。コンペアマッチ割り込みは、コンペアマッチが起こったときに発生する割り込みのことを言います。参考までにコンペアマッチのイメージ図を下に記します。



次に CKS で CMT の動作クロックを設定します。CMT の動作周波数は供給される 48MHz を何分の 1 かにすることで決定します。Pi:Co Classic 3 では 32 分周にして使用しています。CMT のクロックとしては、48MHz/32 = 1.5MHz になります。

分周比の設定ができたので、次に割り込みを発生させる周期を決めます。RX631 ハードウェアマニュアルの”28.2.5 コンペアマッチタイマコンスタントレジスタ(以下 CMCOR)を見てください。今回 1ms ごとに割り込みを発生させるので

CMCOR = 動作周期 × 48MHz ÷ 32
= 1ms × 48MHz ÷32 = 1500

となり、1500(−1)にすれば 1ms になります。−1 するのは、0 からカウントするためです。

次は割り込み優先度を決めます。 RX631 ハードウェアマニュアルの”15.2.3 割り込み要因プライオリティレジスタ n(IPRn)”を見てください。割り込み優先度とは複数の割り込みが発生している状態でどの割り込みを優先して処理するかを決めます。

```
IEN(CMT0,CMI0) = 1;   //割り込み要求を許可
IPR(CMT0,CMI0) = 15;  //割り込み優先度 15が最高
IR(CMT0,CMI0)=0;      //割り込みステータフラグをクリア
```

今回は 0ch 一つだけなので、優先度を最高に設定しています。

モジュールストップコントロールレジスタと同様に CS+に含まれる RX631 の iodefine.h で、割り込みに関するレジスタに簡単にアクセスできるようになっています。本来ならば、CMT0 の CMI0 の割り込み番号を調べる必要がありますが、割り込み要因の CMI0 さえ分かれば、IPR(CMT0,CMI0)=15 で割り込みの優先度を設定することができます。また、上記の IEN(割り込み要求許可)と IPR(割り込み優先度)と IR(割り込みステータス)の設定はセットで行うものと思ってください。CMT0.CMCR.BIT.CMIE で割り込みを許可しているのに IEN で再度割り込みの許可をしているのは、周辺モジュール(CMT など)からの割り込みが 割り込みコントローラー(ICUb) に入り、優先度など確認したあと CPU や DTC などに割り込みを行うシステムになっているからです。

RX63N グループ、RX631 グループ　　15. 割り込みコントローラ（ICUb）

15.2.3　割り込み要因プライオリティレジスタ n（IPRn）（n = 000 ～ 253）

アドレス 0008 7300h～0008 73FDh

	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	—	—	IPR[3:0]			
リセット後の値	0	0	0	0	0	0	0	0

ビット	シンボル	ビット名	機能	R/W
b3-b0	IPR[3:0]	割り込み優先レベル設定ビット	b3 b0 0 0 0 0：レベル0（割り込み禁止）（注1） 0 0 0 1：レベル1 0 0 1 0：レベル2 0 0 1 1：レベル3 0 1 0 0：レベル4 0 1 0 1：レベル5 0 1 1 0：レベル6 0 1 1 1：レベル7 1 0 0 0：レベル8 1 0 0 1：レベル9 1 0 1 0：レベル10 1 0 1 1：レベル11 1 1 0 0：レベル12 1 1 0 1：レベル13 1 1 1 0：レベル14 1 1 1 1：レベル15（最高）	R/W
b7-b4	—	予約ビット	読むと"0"が読めます。書く場合、"0"としてください	R/W

注1.　高速割り込みに設定している場合は、レベル0であっても割り込みの発行が可能です。

最後に CMT　0ch の動作開始の設定をします。

```
CMT.CMSTR0.BIT.STR0=1;
```

RX631 ハードウェアマニュアルの"28.2.1 コンペアマッチタイマスタートレジスタ 0(CMSTR0)を見てください。このレジスタの 0bit 目を 1 にすると動作開始になります。次は MTU2 の設定です。今回使用する MTU2 のチャンネルは 0ch です。

4.　MTU2 の設定

MTU2 で PWM を出力させます。

```
SYSTEM.PRCR.WORD = 0xA502;
MSTP(MTU) = 0;                  //MTUモジュールON
SYSTEM.PRCR.WORD = 0xA500;
MTU.TSTR.BYTE=0;                //タイマ動作ストップ
```

CMT と同じでまずは MTU2 を動作する状態にします。MTU2 の動作設定をする場合、動作を停止させなければなり

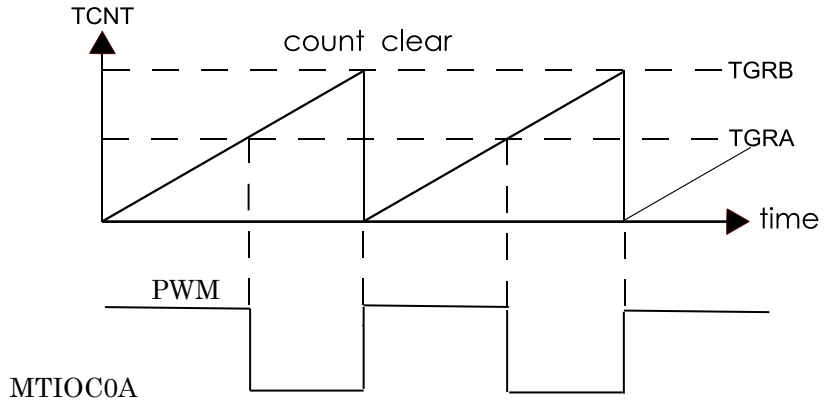
ません。RX631 ハードウェアマニュアルの”23.2.14 タイマスタートレジスタ(TSTR)”を見てください。この 8bit のレジスタを 0 にすれば動作が停止します。
ここから先で出てくる「MTUx」は、MTU2 のチャンネル x を表します。MTU1 なら ch1、MTU2 なら ch2 となります。

```
MTU0.TCR.BIT.CCLR=2;
MTU0.TCR.BIT.TPSC=1;
```

RX631 ハードウェアマニュアルの”23.2.1 タイマコントロールレジスタ(TCR)”を見てください。表 23.4 を見るとコンペアマッチでタイマをクリアするかの設定が書かれています。タイマジェネラルレジスタ A(以下 TGRA)、タイマジェネラルレジスタ B(以下 TGRB)、タイマジェネラルレジスタ C(以下 TGRC)、タイマジェネラルレジスタ D(以下 TGRD)から選択でき、今回は TGRB を設定しています。TGR はコンペアマッチするタイミングを設定するレジスタです。
続いて表 23.6 を見ると MTU2 の内部クロックの分周する設定が書かれています。Pi:Co Classic 3 では MTU2 を 4 分周して 12MHz で動作しています。

```
MTU0.TIORH.BIT.IOA=5;        //コンベアマッチてlow初期はhigh
MTU0.TIORH.BIT.IOB=2;        //コンベアマッチでhigh
```

RX631 ハードウェアマニュアルの”23.2.3 タイマ I/O コントロールレジスタ(TIOR)”を見てください。タイマ I/O コントロールレジスタ(以下 TIOR)は TGR でマッチしたタイミングで出力(I/O)を制御するレジスタです。MTU2 の ch0 を使うので表 23.12、表 23.20 を見てください。TGRA を初期出力 1、コンペアマッチで 0 出力、TGRB を初期出力 0、コンペアマッチで 1 出力に設定しています。文字だけでは解り辛いので、下図を見てください。



このような PWM を出力します。

```
MTU0.TGRA = 6000;            //1kHz
MTU0.TGRB = (12000-1);
```

そして TGRA を 6000、TGRB を 12000(−1)に設定しています。TGRA が PWM の duty 比であり TGRB が周波数の設定になっています。今回は周波数が 1kHz の duty 比が 50%になっています。

```
MTU0.TMDR.BIT.MD=2;
```

RX631 ハードウェアマニュアルの”23.2.2 タイマモードレジスタ(TMDR)”を見てください。このレジスタは各チャンネルの動作モードの設定を行っています。表 23.11 に動作モードの設定がまとめられています。今回は PWM モード 1 に設定します。PWM1 と PWM2 両方とも PWM 出力ができますが、同一周期で複数の PWM が必要な時や MTIOCxB、MTIOCxD から PWM を出力したいときに PWM2 を使用します。周期がことなる PWM が必要な場合は、PWM1 を選択します。PWM1 の場合、TGRA と TGRB の組み合わせで MTIOCxA のポート、TGRC と TGRD の組み合わせで MTIOCxC のポートを PWM 出力することができます。PWM2 の場合、TGRD を周期とした場合、MTIOCxA、MTIOCxB、MTIOCxC

のポートに PWM を出力することができます。
　これでブザの MTU2 の初期設定は終わりです。

```
void init_all(void)
{
        init_clock();       //CPUの動作周波数を設定
        init_io();          //I/O(Input / Output)ポートを設定
        init_cmt();         //CMT(Compare Match Timer)を設定
        init_mtu();         //MTU(Multi Function Timer Pulse Unit)
  }
```

　Init_adc()と init_sci()が init_all にありますが、ブザを鳴らす設定には関係ないため、ここでは、説明を省略しています。Init_adc()に関しては step5 のセンサの値を見ようのところで解説します。
　初期化プログラムの最後に各機能の初期化を全てまとめた init_all 関数を作りまとめて初期化できるようにしてあります。

　これで初期化プログラムの説明を終わります。
　次にメインプログラムの説明に入ります。

プログラムソース 2：PiCoClassic3.c

```
#define FREQ_C          523     //ドの周波数
#define FREQ_D          587     //レの周波数
#define FREQ_E          659     //ミの周波数

void exec_by_mode(int mode)     //モード番号に従って実行する
{
        switch(mode)
        {
                case 1:
                        SET_BUZZER_FREQ(FREQ_C);//ブザの周波数をドに設定
                        ENABLE_BUZZER;          //ブザを発振させる
                        wait_ms(1000);          //1秒間　ドの音を鳴らす
                        DISABLE_BUZZER;         //ブザの発振を停止させる
                        break;

                case 2:
                        SET_BUZZER_FREQ(FREQ_D);//ブザの周波数をレに設定
                        ENABLE_BUZZER;          //ブザを発振させる
                        wait_ms(1000);          //1秒間　レの音を鳴らす
                        DISABLE_BUZZER;         //ブザーの発振を停止させる
                        break;

                case 3:
                        SET_BUZZER_FREQ(FREQ_E); //ブザの周波数をミに設定
                        ENABLE_BUZZER;          //ブザを発振させる
                        wait_ms(1000);          //1秒間　ミの音を鳴らす
                        DISABLE_BUZZER;         //ブザーの発振を停止させる
                        break;
```

動作確認と解説

今までのステップで解説したものは省いて、各モードの中身について説明します。
まずはブザの音程用周波数を扱いやすいように define で宣言しています。

```
#define FREQ_C          523     //ドの周波数
#define FREQ_D          587     //レの周波数
#define FREQ_E          659     //ミの周波数
```

モード 1 の中身は以下のようになっています。

```
case 1:
        SET_BUZZER_FREQ(FREQ_C);//ブザの周波数をドに設定
        ENABLE_BUZZER;          //ブザを発振させる
        wait_ms(1000);          //1秒間　ドの音を鳴らす
        DISABLE_BUZZER;         //ブザの発振を停止させる
        break;
```

始めにブザの周波数を設定しているのですが、この SET_BUZZER_FREQ(FREQ_C)は macro.h に以下のように定義されています。

macro.h より抜粋

```
#define SET_BUZZER_FREQ(f)  MTU0.TGRB=
(unsigned short)(12000000/(f));MTU0.TGRA=
(unsigned short)(6000000/(f))
```

SET_BUZZER_FREQ(f)の f に定義したブザの周波数を代入しています。分子の定数が TGRB と TGRA で 2:1 にしているので、分母の f がどう変わろうと duty 比は 50%で固定です。

ENABLE_BUZZER と DISABLE_BUZZER も macro.h に発振(1)と停止(0)で定義されていています。

wait_ms 関数は misc.c で宣言されていて以下のようになっています。

misc.c より抜粋

```
extern volatile unsigned int timer;
void wait_ms(int wtime)             //mS単位で待ち時間を生成する
{
        unsigned int start_time;
        start_time = timer;
        while( (timer - start_time) < wtime)        ;
}
```

extern で宣言することにより外部のプログラムに宣言した変数を呼び出して使っています。今回は CMT で割り込み処理により 1ms ごとに timer 変数をカウントアップしています。これにより人に解りやすい時間単位で待機時間を扱っています。また、volatile で宣言するとコンパイラの最適化を防止することができます。CMT に関する処理は interrupt.c にまとめられています。

Interrupt.c

```
#include "iodefine.h"
#include "portdef.h"
#include "static_parameters.h"
#include "macro.h"

  volatile unsigned int timer = 0;   //1mS ごとにカウントアップされる変数.
void int_cmt0(void)
{
        battery_save(getBatteryVolt());     //バッテリ監視
        timer++;                            //1mSごとにカウントアップ
  }
```

関数 int_cmt()はタイマ割り込み用の関数ですが、RX631 にもともと用意されたところに int_cmt0()関数をコールするように追加しています。 Intprg.c の 58 行目を見てください。

Intprg.c から抜粋

```
// CMT0 CMI0
void Excep_CMT0_CMI0(void){
        int_cmt0();
  }
```

void Excep_CMT0_CMI0(void)の関数は、vect.h で定義されています。vect.h の 83 行目を見てください。

vect.h から抜粋

```
// CMT0 CMI0
#pragma interrupt (Excep_CMT0_CMI0(vect=28))
  void Excep_CMT0_CMI0(void);
```

この#pragma interrupt というのは割り込み用関数という意味で、CS+のコンパイラ指示のコマンドです。割り込み番号 28(vect=28)の関数として Excep_CMT0_CMI0 という関数名で宣言するという意味です。

この割り込みで 1ms ごとに timer 変数を足しています。

これで STEP4 の説明は終わりです。このあとのモード 2、モード 3 はブザ発振用周波数の値が違うだけで内容は同じなので説明は省きます。

1.6. Step5：センサの値を見よう

概要

ステップ 5 はタイマ割り込みと AD 変換を組み合わせたセンサ値を取得するプログラムになっています。Pi:Co Classic 3 には赤外線センサが 4 つあります。このセンサを使って迷路の壁情報や姿勢制御を行います。AD 変換については、別紙のクラシックサイズマイクロマウス Pi:Co Classic3「部品説明+用語集」の AD 変換を参照してください。このプログラムの動作確認としてセンサの調整をしていきます。

前回のステップで解説したところは省いて、追加したプログラムの説明をしていきますので、忘れてしまった場合は前回の解説を読みなおしてください。

プログラムソース 1：init.c からの抜粋

```
void init_io(void)
{
…
        //Sensor
        PORT5.PDR.BIT.B4 = IO_OUT;//SLED_L
        PORT0.PDR.BIT.B5 = IO_OUT;//SLED_R
        PORT2.PDR.BIT.B7 = IO_OUT;//SLED_FL
        PORTB.PDR.BIT.B5 = IO_OUT;//SLED_FR
…
        MPC.PWPR.BIT.B0WI=0;
        MPC.PWPR.BIT.PFSWE=1;
        MPC.PB3PFS.BIT.PSEL=1;      //MTIOC0A
        MPC.P17PFS.BIT.PSEL=1;      //MTIOC3Aとして使用
        MPC.PE2PFS.BIT.PSEL=1;      //MTIOC4A

        MPC.PE0PFS.BIT.ASEL=1;      //A/D SEN_FL
        MPC.PE1PFS.BIT.ASEL=1;      //A/D SEN_NL
        MPC.P44PFS.BIT.ASEL=1;      //A/D SEN_R
        MPC.P46PFS.BIT.ASEL=1;      //A/D SEN_NR
        MPC.P42PFS.BIT.ASEL=1;      //A/D SEN_FR
        MPC.PWPR.BYTE=0x80;
…
        PORTE.PMR.BIT.B0=1; //A/D
        PORTE.PMR.BIT.B1=1; //A/D
        PORT4.PMR.BIT.B4=1; //A/D
        PORT4.PMR.BIT.B6=1; //A/D
        PORT4.PMR.BIT.B2=1; //A/D
}
void init_cmt(void)
{
        SYSTEM.PRCR.WORD = 0xA502;
        MSTP(CMT0) = 0;
        MSTP(CMT1) = 0;
        SYSTEM.PRCR.WORD = 0xA500;
```

```
//CMT0は制御割り込み用タイマとして使用
        CMT0.CMCR.BIT.CKS=1;        // PCLK/32 1.5MHz
        CMT0.CMCR.BIT.CMIE=1;       //割り込みを許可
        CMT0.CMCNT=0;               //カウンターのクリア
        CMT0.CMCOR=1500-1; //1kHz
        IEN(CMT0,CMI0) = 1;   //割り込み要求を許可
        IPR(CMT0,CMI0) = 15;  //割り込み優先度 15が最高
        IR(CMT0,CMI0)=0;      //割り込みステータフラグをクリア
        //CMT1はセンサ制御用タイマとして使用
        CMT1.CMCR.BIT.CKS=1;        // PCLK/32 1.5MHz
        CMT1.CMCR.BIT.CMIE=1;       //割り込みを許可
        CMT1.CMCNT=0;               //カウンタのクリア
        CMT1.CMCOR=(1500/4)-1;      //4kHz
        IEN(CMT1,CMI1) = 1;   //割り込み要求を許可
        IPR(CMT1,CMI1) = 14;  //割り込み優先度を次点に設定
        IR(CMT1,CMI1)=0;      //割り込みステータフラグをクリア
        CMT.CMSTR0.BIT.STR0=1;      //カウントスタート
        CMT.CMSTR0.BIT.STR1=1;      //カウントスタート
}
void init_adc(void)
{
        SYSTEM.PRCR.WORD = 0xA502;
        MSTP(S12AD) = 0;
        SYSTEM.PRCR.WORD = 0xA500;
        S12AD.ADCER.BIT.ADRFMT=0;//右づめ
        S12AD.ADCSR.BIT.CKS=0x03;//PCLKの分周なし
}
void init_all(void)
{
        init_clock();       //CPUの動作周波数を設定
        init_io();          //I/O(Input / Output)ポートを設定
        init_cmt();         //CMT(Compare Match Timer)を設定
        init_mtu();         //MTU(Multi Function Timer Pulse Unit)
        init_adc();         //ADC(Analog Digital Converter)初期化
        init_usb();
}
```

動作確認と解説

このサンプルの CS+プロジェクトは「C:¥20170407Documents¥Sample_Program¥step5_sensor_check_usb」フォルダ内の「PiCoClassic3.mtpj」です。それをビルドし、RFP を使って、「C:¥20170407Documents¥Sample_Program¥step5_sensor_check_usb¥DefaultBuild」フォルダ内の「PiCoClassic3.mot」を書き込みます。

前回に解説した CMT の設定にもう一つチャンネルを追加して、センサの値を取得するための割り込みを設定します。

```
CMT1.CMCR.BIT.CKS=1;        // PCLK/32 1.5MHz
CMT1.CMCR.BIT.CMIE=1;       //割り込みを許可
CMT1.CMCNT=0;               //カウンタのクリア
CMT1.CMCOR=(1500/4)-1;      //4kHz
IEN(CMT1,CMI1) = 1;         //割り込み要求を許可
IPR(CMT1,CMI1) = 14;        //割り込み優先度を次点に設定
IR(CMT1,CMI1)=0;            //割り込みステータフラグをクリア
CMT.CMSTR0.BIT.STR1=1;      //カウントスタート
```

今回追加したチャンネルは 1ch です。割り込み周期の設定を 1/4ms に変更し、割り込み優先度を 0ch の次に優先するように変更しています。他の設定は 0ch と同じです。次は AD 変換の設定についてです。

```
SYSTEM.PRCR.WORD = 0xA502;
MSTP(S12AD) = 0;
SYSTEM.PRCR.WORD = 0xA500;
```

CMT や MTU2 で始めに設定したように ADC を動作状態にします。今回使う ADC のチャンネルは 12bit の方です。ADC の設定で共通な個所を設定します。

```
S12AD.ADCER.BIT.ADRFMT=0;//右づめ
S12AD.ADCSR.BIT.CKS=0x03;//PCLK の分周なし
```

RX631 ハードウェアマニュアルの”42.2.7 A/D コントロール拡張レジスタ(ADCER)”を見てください。ADC の結果は 12bit であり、16bit でアクセスすると上位または、下位の 4bit が 0 になります。見やすいように上位 4bit を 0 にするため右づめにします。初期値と同じですが、あとで、見たときにわかるようにあえて、コメントと設定をしています。

42.2.7　A/D コントロール拡張レジスタ（ADCER）

アドレス 0008 900Eh

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	ADRFMT	—	—	—	—	—	—	—	—	—	ACE	—	—	—	—	—
リセット後の値	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

ビット	シンボル	ビット名	機能	R/W
b4-b0	—	予約ビット	読むと“0”が読めます。書く場合、“0”としてください	R/W
b5	ACE	自動クリアイネーブルビット	0：自動クリアを禁止 1：自動クリアを許可	R/W
b14-b6	—	予約ビット	読むと“0”が読めます。書く場合、“0”としてください	R/W
b15	ADRFMT	A/Dデータレジスタフォーマット選択ビット	0：A/Dデータレジスタのフォーマットを右詰めにする 1：A/Dデータレジスタのフォーマットを左詰めにする	R/W

RX631 ハードウェアマニュアルの”42.2.1 A/D コントロールレジスタ(ADCSR)”を見てください。ここでは、ADC のクロックを設定しています。初期値が 0 であり、PCLK/8 になっています。高速に処理させるため、分周をしない CKS=0x03 に設定します。

42.2　レジスタの説明

42.2.1　A/D コントロールレジスタ（ADCSR）

アドレス 0008 9000h

	b7	b6	b5	b4	b3	b2	b1	b0
	ADST	ADCS	—	ADIE	CKS[1:0]		TRGE	EXTRG
リセット後の値	0	0	0	0	0	0	0	0

ビット	シンボル	ビット名	機能	R/W
b0	EXTRG	トリガ選択ビット（注1）	0：同期トリガ（MTU、TPU、TMR）によるA/D変換の開始を選択 1：非同期トリガ（ADTRG0#）によるA/D変換の開始を選択	R/W
b1	TRGE	トリガ開始許可ビット	0：同期、非同期トリガによるA/D変換の開始を禁止 1：同期、非同期トリガによるA/D変換の開始を許可	R/W
b3-b2	CKS[1:0]	A/D変換クロック選択ビット	b3 b2 0 0：PCLK/8 0 1：PCLK/4 1 0：PCLK/2 1 1：PCLK	R/W
b4	ADIE	スキャン終了割り込み許可ビット	0：スキャン終了後のS12ADI0割り込み発生の禁止 1：スキャン終了後のS12ADI0割り込み発生の許可	R/W
b5	—	予約ビット	読むと“0”が読めます。書く場合、“0”としてください	R/W
b6	ADCS	スキャンモード選択ビット	0：シングルスキャンモード 1：連続スキャンモード	R/W
b7	ADST	A/D変換スタートビット	0：A/D変換停止 1：A/D変換開始	R/W

それでは割り込み処理の説明をしていきます。

プログラムソース 2:interrupt.c

```
#include "iodefine.h"
#include "portdef.h"
#include "static_parameters.h"
#include "macro.h"
#include "parameters.h"
#include "interrupt.h"

void int_cmt0(void)
{
}

void int_cmt1(void)                    //センサ読み込み用り込み
{
        static int state = 0;          //読み込むセンサのローテーション管理用変数
        int i;

        switch(state)
        {
                case 0:                //右センサ読み込み
                        SLED_R = 1;                                     //LED点灯
                        for(i = 0; i < WAITLOOP_SLED; i++);//フォトトランジスタの応答待ちループ
                        S12AD.ADANS0.BIT.ANS0=0x0040;   //AN006
                        S12AD.ADCSR.BIT.ADST=1;         //AD変換開始
                        while(S12AD.ADCSR.BIT.ADST);    //AD変換終了まで待つ
                        SLED_R = 0;                                     //LED消灯
                        sen_r.p_value = sen_r.value;    //過去の値を保存
                        sen_r.value = S12AD.ADDR6;      //値を保存
                        break;
                case 1:                //前左センサ読み込み
                        SLED_FL = 1;                                    //LED点灯
                        for(i = 0; i < WAITLOOP_SLED; i++);//フォトトランジスタの応答待ちループ
                        S12AD.ADANS0.BIT.ANS0=0x0010;   //AN004
                        S12AD.ADCSR.BIT.ADST=1;         //AD変換開始
                        while(S12AD.ADCSR.BIT.ADST);    //AD変換終了まで待つ
                        SLED_FL = 0;                                    //LED消灯
                        sen_fl.p_value = sen_fl.value;  //過去の値を保存
                        sen_fl.value = S12AD.ADDR4;     //値を保存
                        break;
                case 2:                //前右センサ読み込み
                        SLED_FR = 1;                                    //LED点灯
                        for(i = 0; i < WAITLOOP_SLED; i++);//フォトトランジスタの応答待ちループ
                        S12AD.ADANS0.BIT.ANS0=0x0200;   //AN009
                        S12AD.ADCSR.BIT.ADST=1;         //AD変換開始
                        while(S12AD.ADCSR.BIT.ADST);    //AD変換終了まで待つ
                        SLED_FR = 0;                                    //LED消灯
```

```
                    sen_fr.p_value = sen_fr.value;      //過去の値を保存
                    sen_fr.value = S12AD.ADDR9;     //値を保存
                    break;
            case 3:             //左センサ読み込み
                    SLED_L = 1;                                 //LED点灯
                    for(i = 0; i < WAITLOOP_SLED; i++);//フォトトランジスタの応答待ちループ
                    S12AD.ADANS0.BIT.ANS0=0x0004;       //AN002
                    S12AD.ADCSR.BIT.ADST=1;             //AD変換開始
                    while(S12AD.ADCSR.BIT.ADST);        //AD変換終了まで待つ
                    SLED_L = 0;                                 //LED消灯
                    sen_l.p_value = sen_l.value;                //過去の値を保存
                    sen_l.value = S12AD.ADDR2;                  //値を保存
                    break;
    }
    state++;                                                    //４回ごとに繰り返す
    if(state > 3)
    {
            state = 0;
    }

}
```

動作確認と解説

今回は割り込みを switch 文で 4 つにわけ、1/4ms の割り込みを 4 回行っています。この 1 サイクルが終わると 1ms ごとに 1 つ 1 つの割り込み処理が行われるので、疑似的に 1ms の割り込みを 4 つ行うことになります。各割り込み処理の終わりに state 変数をインクリメントし、state 変数の値によって case0、case1、case2、case3 と処理が進んでいきます。そして state 変数が 3 よりも大きいときに 0 になりまた case0 から処理が行われます。

```
    state++;                    //４回ごとに繰り返す
    if(state > 3)
    {
            state = 0;
    }
```

センサの一連の処理について解説します。

```
SLED_R = 1;                                  //LED点灯
for(i = 0; i < WAITLOOP_SLED; i++);//フォトトランジスタの応答待ちループ
S12AD.ADANS0.BIT.ANS0=0x0040;                //AN006
S12AD.ADCSR.BIT.ADST=1;                      //AD変換開始
while(S12AD.ADCSR.BIT.ADST);                 //AD変換終了まで待つ
SLED_R = 0;                                  //LED消灯
sen_r.p_value = sen_r.value;                 //過去の値を保存
sen_r.value = S12AD.ADDR6;                   //値を保存
```

始めにセンサ用 LED を点灯させます。次に AD 変換のチャンネルを選択します。case0 の処理は右センサの処理なので 6ch を選択しています。赤外線センサが LED の光を受光し、アナログ電圧が立ち上がるまで待ちます。立ち上がったアナログ電圧を AD 変換でデジタル信号に変換し変換終了まで待ちます。AD 変換が終わりましたらセンサ用 LED を消灯します。これがセンサからの信号を取得する 1 サイクルです。このあとに取得した値を保存します。

```
sen_r.p_value = sen_r.value;                 //過去の値を保存
sen_r.value = S12AD.ADDR6;                   //値を保存
```

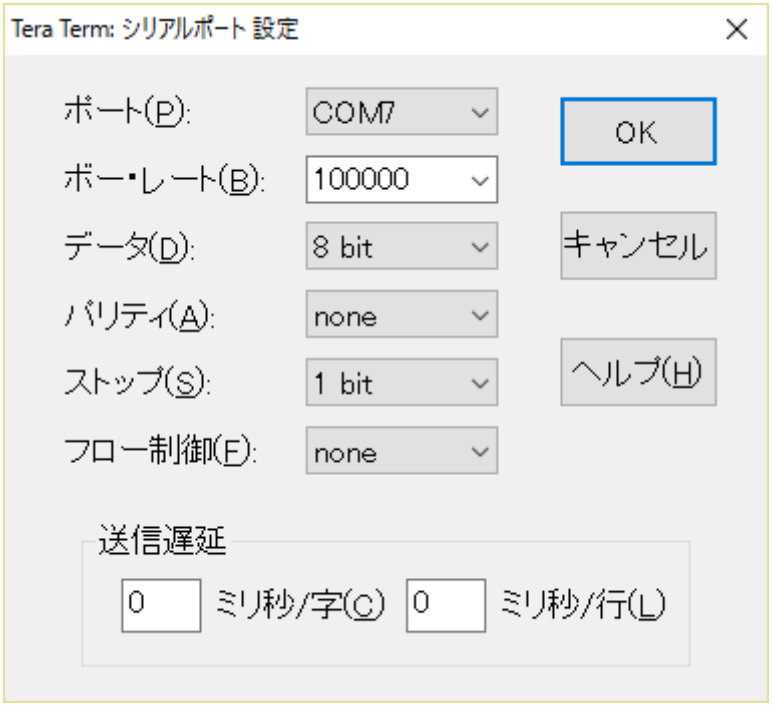
姿勢制御の処理のために過去の値を保存し、先ほど取得した値を sen_r.value に保存します。

ADC のチャネルに関しては、RX631 ハードウェアマニュアルの”42.2.2 A/D チャネル選択レジスタ 0(ADANS0)”を参照してください。

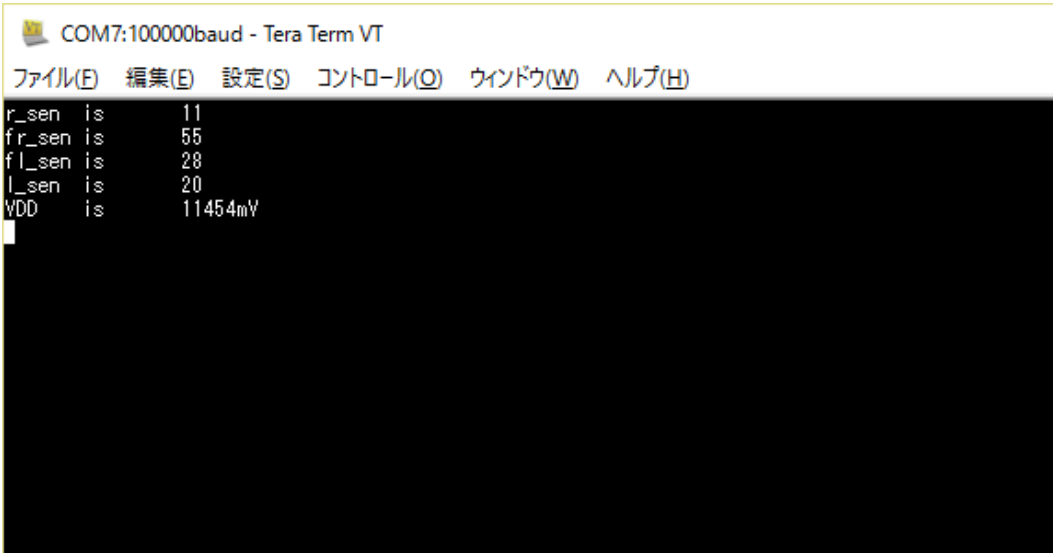
これでセンサからの値が取得、保存できました。メインプログラムは電源を入れたらセンサの値を取得するようになっています。実際にセンサ値を見ながら調整していきましょう。

電源を入れる前に PC につないだ通信用のケーブルを差し込んでください。Pi:Co Classic3 のモードが RUN になっていることを確認してください。

次に PC でターミナルエミュレータ（Tera Term）を起動し、Pi:Co の電源を入れてください。シリアルケーブルをさしている com ポート番号を選択してください。（メニューバー -> 設定 -> シリアルポート）ほかの項目は初期設定のままで大丈夫です。下図設定例を参照してください。



画面にセンサの値は出ましたか？センサの前に手をかざすなどして反応がありましたら（値が変化していたら）正常に動作しています。マウスを迷路の中央に置いて調整をしましょう。まずはセンサ用 LED と受光器の向いている方向（光軸）を合わせましょう。なるべく前センサと横センサそれぞれのスポットを左右対象になるようにしましょう。横センサは少し角度をつけると良いでしょう。前壁に対して 15°～45°くらい傾けると良いでしょう。



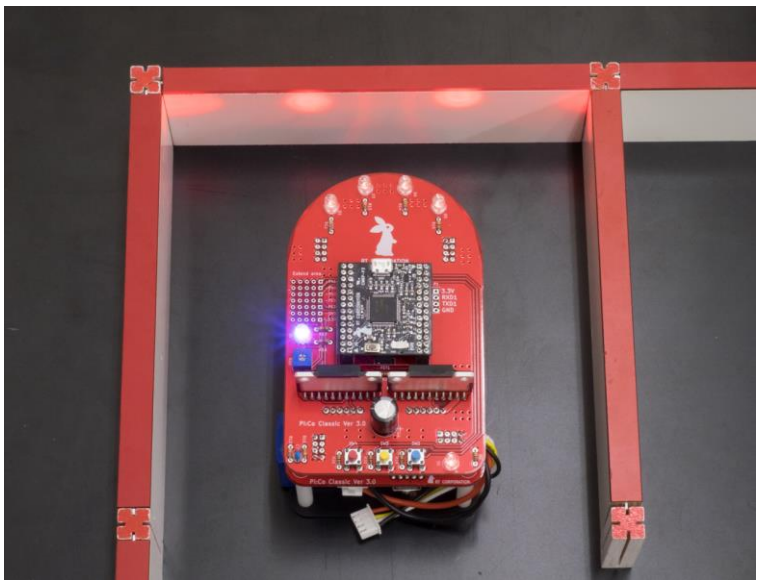


図 1-6-1 上から見た図

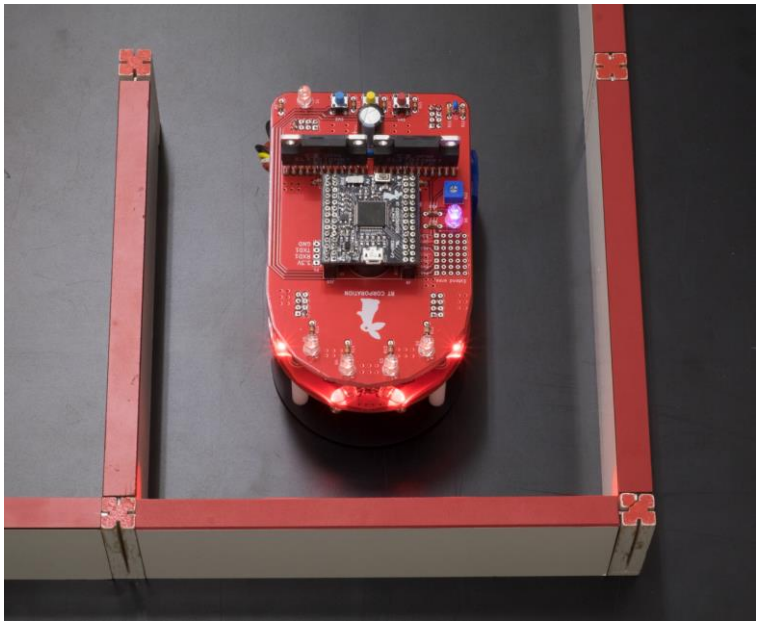


図 1-6-2 前から見た図

前センサは少し左右に広げるくらいが良いです。(LED の光が広がるため)ただ、内側に狭めるのはやめましょう。

1.7. Step6：モータを回そう 1

概要

今回は、モータを回します。モータはステッピングモータを使用します。詳しくは、別紙のクラシックマイクロマウス Pi:Co Classic3 パート 1「基礎知識編」の 2.7 ステッピングモータ仕様を見てください。

実際のモータ制御には PWM を使用してモータを回していますが、この章ではモータドライバーIC とブザにパルスを送り、モータが回ることとブザなることが同じ原理であることを確認します。

Step6_1_run_wait メインプログラムソース

```
void main(void)
{
        int i,j=0;
        init_all();                     //モータで使用するポートを出力に設定
        MOT_CWCCW_R = MOT_CWCCW_L = MOT_FORWARD;//前方に進む
        MOT_POWER_ON;
        for(i = 0; i < 0x7fffff; i++);
        while(j < LEN2STEP(180))
        {
                BUZZER = MOTOR_R = MOTOR_L = 1; //モータclockに1を出力
                for(i = 0; i < 0x3000; i++);            //時間待ち(空ループ)
                BUZZER = MOTOR_R = MOTOR_L = 0; //モータclockに0を出力
                for(i = 0; i < 0x3000; i++);            //時間待ち(空ループ)
                j = j + 2;
        }
        for(i = 0; i < 0x7fffff; i++);
        MOT_POWER_OFF;
        while(1);
  }
```

動作確認と解説

このサンプルの CS+プロジェクトは「C:¥20170407Documents¥Sample_Program¥step6_1_run_wait」フォルダ内の「PiCoClassic3.mtpj」です。それをビルドし、RFP を使って、「C:¥20170407Documents¥Sample_Program¥step6_1_run_wait¥DefaultBuild」フォルダ内の「PiCoClassic3.mot」を書き込みます。

使用しているモータドライバの IC SLA7070 は、回転方向、モータイネーブル、クロックの三種類で制御しています。まず、41 行目モータの回転方向を決め、そのあと、43 行目でモータを励磁しています。MOT_CWCCW_R などの宣言は portdef.h を見てください。

```
MOT_CWCCW_R = MOT_CWCCW_L = MOT_FORWARD;
MOT_POWER_ON;
```

44 行目の”for(i = 0; i < 0x7fffff; i++);”は、いきなりモータを回すと脱調することがあるため、モータを回す前に少し励磁させています。

48 行目から 52 行目でモータにクロック(パルス)をいれてモータを回しています。同じパルスをブザにいれてパルスで動作していることを確認しています。この例では、LEN2STEP(180)で 180mm 進むのに必要なパルスを求め、そのパルス数だけモータに送るようにしています。

```
while(j < LEN2STEP(180))
{
        BUZZER = MOTOR_R = MOTOR_L = 1; //モータclockに1を出力
        for(i = 0; i < 0x3000; i++);          //時間待ち(空ループ)
        BUZZER = MOTOR_R = MOTOR_L = 0; //モータclockに0を出力
        for(i = 0; i < 0x3000; i++);          //時間待ち(空ループ)
        j = j + 2;
}
```

LENSTEP()は”macro.h”に記述しています。

```
#define TIRE_CIRCUIT  (PI*TIRE_DIAMETER)
#define LEN2STEP(l)     (2*400*(l)/TIRE_CIRCUIT)
```

PI は”static_parameters.h”に TIRE_DIAMETER は”parameters.h”で定義しています。少し LEN2STEP を解説しますと LEN2STEP は右のモータのパルスと左モータのパルスを合わせたものになります。これは、姿勢制御で、右のモータの走行距離と左の走行距離がかけ離れていたとき、右の走行距離で判断すると行きすぎたり、足りなかったりすることがあります。これを少しでも目標値に近づけるため、右と左の平均を使って走行距離を判断するようにしています。Pi:Co Classic3 で使用しているステッピングモータは一回転 400 パルス、車輪の直径が 48mm ですので、1 パルスが 48×PI/400 となり、距離 Lmm に必要なパルスは、L×400/(48×PI)となります。

55 行目で慣性モーメントを抑えて、最後に 56 行目で励磁を off にしています。

指定した距離を進むことを確認したら、次の STEP では while 文内で作ったクロックを PWM に置き換えて、加減速してみます。

1.8. Step6：モータを回そう 2

概要

Step6:モータを回そう 1 でパルスを送ることでモータが回ることが分かったと思います。ここでは、モータの制御を PWM に変更し、台形駆動をしてみます。台形加速は別紙のクラシックサイズマイクロマウス Pi:Co Classic3 パート 1「基礎知識編」の 2.7 ステッピングモータ仕様 図 6 のような速度変化させることです。前回のステップで解説したところは省いて、追加したプログラムの説明をしていきますので、忘れてしまった場合は、前回の解説を読み直してください。

プログラムソース 1: init.c からの抜粋

```
void init_io(void)
  {
  …
        //MOT_POWER
        PORT1.PDR.BIT.B5 = IO_OUT;//motor Enable
        //MOT_CWCCW
        PORTC.PDR.BIT.B5 = IO_OUT;//Rmotor
        PORTC.PDR.BIT.B6 = IO_OUT;//Lmotor
        MPC.PWPR.BIT.B0WI=0;
        MPC.PWPR.BIT.PFSWE=1;
        MPC.PB3PFS.BIT.PSEL=1;      //MTIOC0A
        MPC.P17PFS.BIT.PSEL=1;      //MTIOC3Aとして使用
        MPC.PE2PFS.BIT.PSEL=1;      //MTIOC4A
  …
        MPC.PWPR.BYTE=0x80;
        PORTB.PMR.BIT.B3=1; //SP PWM
        PORT1.PMR.BIT.B7=1;  //右PWM
        PORTE.PMR.BIT.B2=1; //左 PWM
  …
  }
void init_mtu(void)
{
…
        //右モータ用MTU設定
        MTU3.TCR.BIT.TPSC=2;//PCLK/16 3MHz
        MTU3.TCR.BIT.CCLR=1;//PWM TGRAのコンペアマッチでTCNTクリア
        MTU3.TIORH.BIT.IOA=1;//初期出力0ンペアマッチ0出力
        MTU3.TIORH.BIT.IOB=2;//初期出力0コンペアマッチ1出力
        MTU3.TGRA = 12000;
        MTU3.TGRB = 50;
        MTU3.TGRC = 48000;
        MTU3.TMDR.BIT.MD=2;         //PWM1
```

```
        MTU3.TMDR.BIT.BFA = 1;       //バッファモードに設定
        MTU3.TIER.BIT.TGIEB = 1;//GRBコンペアマッチでの割り込み許可
        IEN(MTU3,TGIB3) = 1;  //割り込み要求を許可
        IPR(MTU3,TGIB3) = 13;//割り込み優先度を次点に設定
        IR(MTU3,TGIB3)=0;     //割り込みステータフラグをクリア
        //左モータ用MTU設定
        MTU.TOER.BIT.OE4A=1;         //MTU出力端子を出力許可する
        MTU4.TCR.BIT.TPSC=2;         //PCLK/16 3MHz
        MTU4.TCR.BIT.CCLR=1;         //PWM TGRAのコンペアマッチでTCNTクリア
        MTU4.TIORH.BIT.IOA=1;        //初期出力0ンペアマッチ0出力
        MTU4.TIORH.BIT.IOB=2;        //初期出力0コンペアマッチ1出力
        MTU4.TGRA = 12000;
        MTU4.TGRB = 50;
        MTU4.TGRC = 48000;
        MTU4.TMDR.BIT.MD=2;          //PWM1
        MTU4.TMDR.BIT.BFA = 1;       //バッファモードに設定
        MTU4.TIER.BIT.TGIEB = 1;//GRBコンペアマッチでの割り込み許可
        IEN(MTU4,TGIB4) = 1;  //割り込み要求を許可
        IPR(MTU4,TGIB4) = 12;//割り込み優先度を次点に設定
        IR(MTU4,TGIB4)=0;     //割り込みステータフラグをクリア
        MTU.TSTR.BIT.CST0 = 0;       //タイマストップ
        MTU.TSTR.BIT.CST3 = 0;       //タイマストップ
        MTU.TSTR.BIT.CST4 = 0;       //タイマストップ
}
```

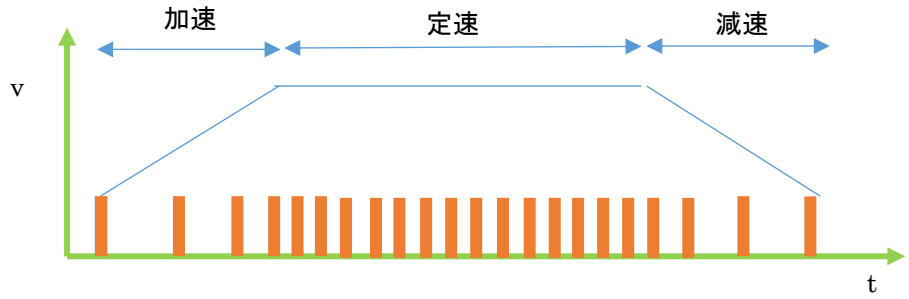
動作確認と解説

このサンプルの CS+プロジェクトは「C:¥20170407Documents¥Sample_Program¥step6_2_run_ST_PWM」フォルダ内の「PiCoClassic3.mtpj」です。それをビルドし、RFP を使って、「C:¥20170407Documents¥Sample_Program¥step6_2_run_ST_PWM¥DefaultBuild」フォルダ内の「PiCoClassic3.mot」を書き込みます。

モード 1 にし、真ん中のスイッチを押すと指定した距離を走行します。

Init_io 関数では、各ポートの入出力と機能の設定をしています。今回モータを PWM で回すため、汎用ポートから機能ポートに設定しています(ファイル init.c の 70、71 行目、81、82 行目)。

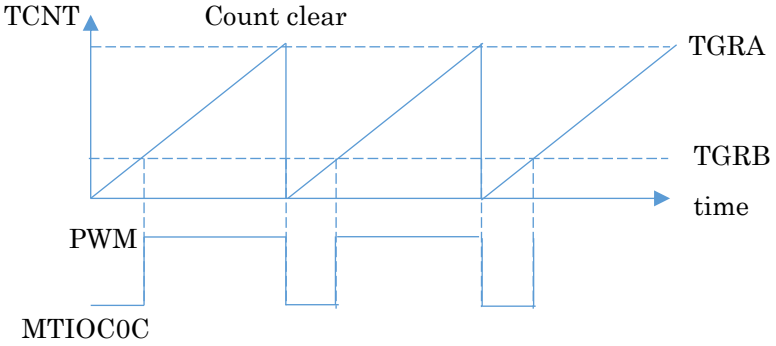
Init_mtu 関数で PWM の設定をしています。STEP4 のブザで同じ PWM1 モードを使用しますが、モータの PWM は周期を可変するため、少し設定が違っています。概要にもありますように今回は台形制御をします。台形制御するには、パルスの間隔を短くすると速度が上がり、間隔を短くすると速度が遅くなります。イメージとして次頁になります。オレンジ色がパルスです。加速時は、パルスの間隔が短くなります。逆に減速はパルスの間隔が長くなります。定速は、パルスの間隔が一定です。



ブザの PWM は、割り込みを使用していません。モータの PWM には、割り込みとバッファ動作の設定をしています。

```
MTU3.TCR.BIT.TPSC=2;//PCLK/16 3MHz
MTU3.TCR.BIT.CCLR=1;//PWM TGRAのコンペアマッチでTCNTクリア
MTU3.TIORH.BIT.IOA=1;//初期出力0ンペアマッチ0出力
MTU3.TIORH.BIT.IOB=2;//初期出力 0 コンペアマッチ 1 出力
```

設定の内容については、ブザのところを見てください。TGRA が周期レジスタ、TGRB がパルス幅になります。このような PWM 出力をします。




```
MTU3.TMDR.BIT.BFA = 1;//バッファモードに設定
```

ファイル init.c の 150 行目の MTU3.TMDR.BIT.BFA=1;のバッファモードとは、TGRA のコンペアマッチのタイミングで TGRC の値を TGRA にコピーしています。詳しくは RX631 ハードウェアマニュアルの”23.2.2 タイマモードレジスタ(TMDR)”と”23.3.3 バッファ動作”の(2)を見てください。

バッファモードの場合、コンペアマッチに指定した TGRA のペアの TGRC を変更することで、次の TGRA のコンペアマッチを変更します。Pi:Co Classic3 では、周期を変更しています。バッファ動作をせずに周期を変更するとどのような現状が起きると思いますか?

(2)　バッファ動作例

(a)　TGR がアウトプットコンペアレジスタの場合

MTU0 を PWM モード 1 に設定し、TGRA と TGRC をバッファ動作に設定した場合の動作例を図 23.17 に示します。TCNT はコンペアマッチ B によりクリア、出力はコンペアマッチ A で High 出力、コンペアマッチ B で Low 出力に設定した例です。この例では、TBTM の TTSA ビットは “0” に設定しています。

バッファ動作が設定されているため、コンペアマッチ A が発生すると出力を変化させると同時に、バッファレジスタ TGRC の値がタイマジェネラルレジスタ TGRA に転送されます。この動作は、コンペアマッチ A が発生する度に繰り返されます。

PWM モードについては、「23.3.5　PWM モード」を参照してください。

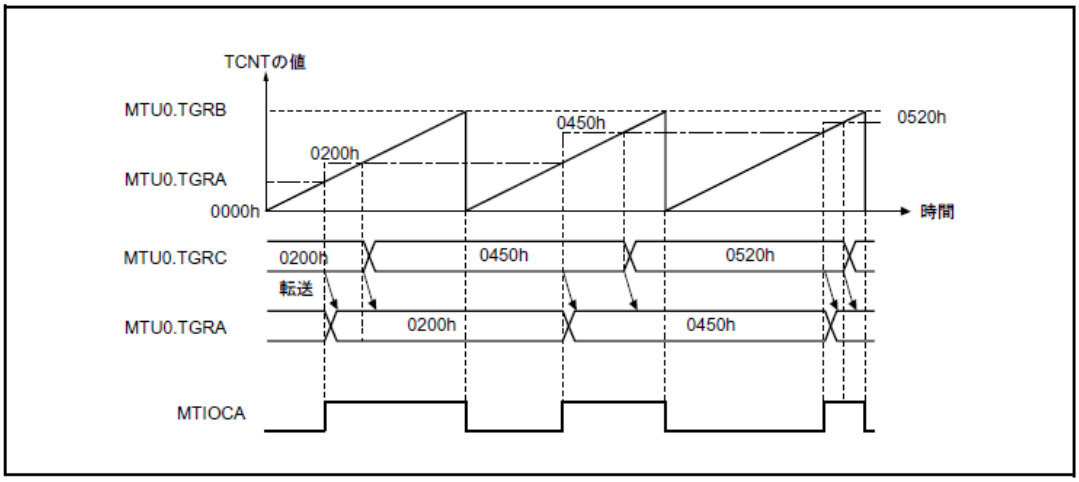
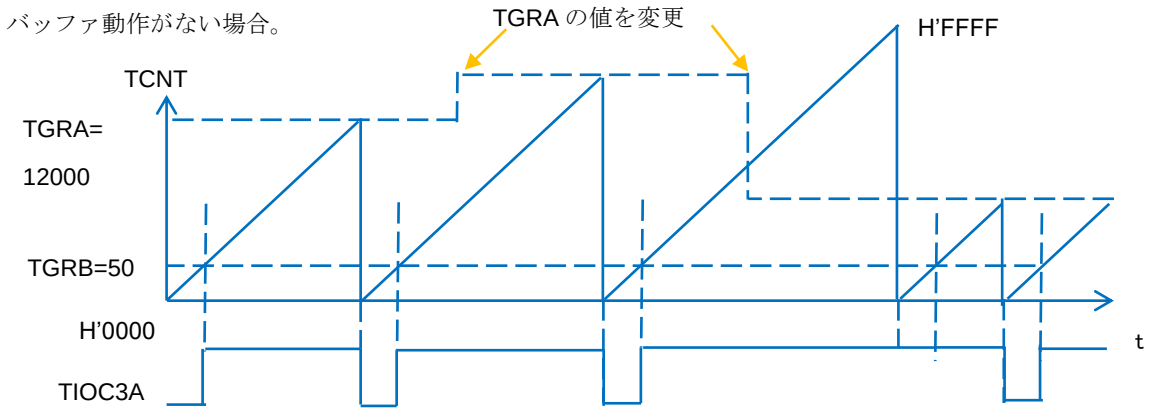
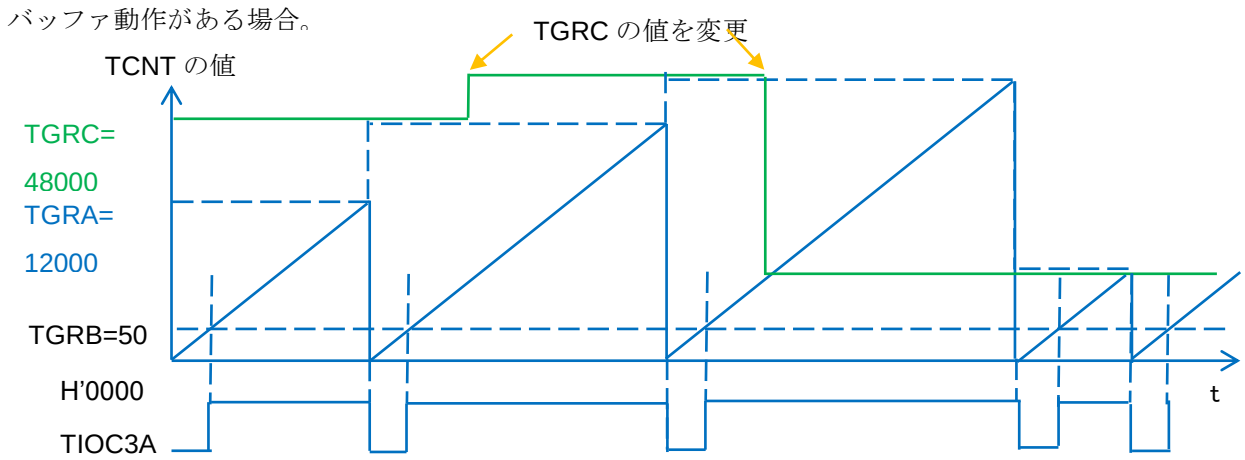


図 23.17　バッファ動作例（1）



TGRA の値を TCNT の値より小さくすると、TGRA のコンペアマッチのタイミングがなくなり、TCNT がカウントクリアできなくなります。そのままカウントするとオーバーフローして 0 になりますが、意図したパルスとかけ離れてしまうため、モータが脱調してしまう可能性があります。



バッファ動作がある場合、ない場合にくらべ、TCNT のオーバーフローはなくなり、一周期後に必ず更新されています。

```
MTU3.TIER.BIT.TGIEB = 1;//GRB コンペアマッチでの割り込み許可
```

PWM の立ち上がりのタイミングで割り込みを発生し、パルスをカウントします。ステッピングモータはパルスをカウントすることで走行の距離が分かります。

```
MTU.TOER.BIT.OE4A=1;          //MTU 出力端子を出力許可する
```

左のモータに使用している PWM MTU4 はタイムアウトプットマスタ許可レジスタで出力を許可する必要があります。そのほかは、右のモータの MTU3 と同じ設定ですので解説を省略します。タイムアウトプットマスタ許可レジスタに関しては、RX631 ハードウェアマニュアルの”23.2.17 タイムアウトプットマスタ許可レジスタ(TOER)を見てください。

23.2.17　タイマアウトプットマスタ許可レジスタ（TOER）

アドレス　MTU.TOER 0008 860Ah

	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	OE4D	OE4C	OE3D	OE4B	OE4A	OE3B
リセット後の値	1	1	0	0	0	0	0	0

ビット	シンボル	ビット名	機能	R/W
b0	OE3B	マスタ許可MTIOC3Bビット	0：MTU出力禁止（注1） 1：MTU出力許可	R/W
b1	OE4A	マスタ許可MTIOC4Aビット	0：MTU出力禁止（注1） 1：MTU出力許可	R/W
b2	OE4B	マスタ許可MTIOC4Bビット	0：MTU出力禁止（注1） 1：MTU出力許可	R/W
b3	OE3D	マスタ許可MTIOC3Dビット	0：MTU出力禁止（注1） 1：MTU出力許可	R/W
b4	OE4C	マスタ許可MTIOC4Cビット	0：MTU出力禁止（注1） 1：MTU出力許可	R/W
b5	OE4D	マスタ許可MTIOC4Dビット	0：MTU出力禁止（注1） 1：MTU出力許可	R/W
b7-b6	—	予約ビット	読むと“1”が読めます。書く場合、“1”としてください	R/W

注1.　MTU出力禁止を設定したときに、各端子から非アクティブレベルを出力する場合は、I/Oポートのデータ方向レジスタ（PDR）、ポート出力データレジスタ（PODR）にあらかじめ汎用入出力ポートに非アクティブレベルを出力する設定をした上で、ポートモードレジスタ（PMR）で汎用入出力ポート使用に切り替えてください。

プログラムソース 2: interrupt.c からの抜粋

```
void int_cmt0(void)
{
        float spd_r, spd_l;         //最終的な速度
        speed+=r_accel;             //加速処理
        if(speed > max_speed)       //最高速度を制限
        {
                speed = max_speed;
        }
        if(speed < MIN_SPEED){//最低速度を制限(MTUの周期設定レジスタの値が～0xffffまでであるため)
                speed = MIN_SPEED;
        }
        spd_r = spd_l = speed;
        MTU3.TGRC = SPEED2GREG(spd_r);//右速度(mm/s)からジェネラルレジスタの値に変換して、バッファレジスタに代入
        MTU4.TGRC = SPEED2GREG(spd_l);//左速度(mm/s)からジェネラルレジスタの値に変換して、バッファレジスタに代入
        timer++;                                    //1mSごとにカウントアップ
  }
```

```
void int_mot_r(void)      //右モータが１ステップ進む毎の割り込み
{
        step_r++;         //ステップ数をカウント
}
void int_mot_l(void)      //左モータが１ステップ進む毎の割り込み
{
        step_l++;         //ステップ数をカウント
}
```

動作確認と解説

ファイル interrupt.c の Int_cmt0()の関数では、モータの加減速を行っています。23 行目の speed += r_accel でスピードの加減速をしています。r_accel は、PiCoClassic3.c の exec_by_mode 関数内で値を設定し、加速、減速を run.c の straight 関数内で設定しています。
高校の物理で

速度 v = 加速度 a × 時間 t + 初速度 v'

であると習ったことでしょう。その公式が 23 行目になっています。23 行目は、 speed = r_accel + speed;と書き直すことができ、時間 t が "1"だとすると、先ほど示した公式と同じになります。時間 t を"1"としましたが、CMT0 の割り込み周期はいくつに設定していましたでしょうか?1ms に設定していましたよね。公式に戻って単位をつけて表すと、

v[m/s] = a[m/s/s] × t[s] + v' [m/s]

になります。時間 t[s]を t[ms]にした場合、このように書き直すことができます。

V[mm/s] = a[m/s/s] × t[ms] + v' [mm/s]

お気づきだと思いますが、割り込み周期を 1ms にすると時間の係数が"1"になりますので、speed += r_accel;で加減速できるという事になります。
25 行目から 28 行目は最高速度の制限をしています。速度が速くなると制御しにくくなるため、制御できるスピードまでに抑えています。30 行目から 32 行目は、最低速度を制限しています。周期レジスタの最大値は 0xFFFF と決まっていますので、それを超えない値にしています。
SPEED2GREG( )で速度を TGRC に変換しています。この関数は、macro.h に
#define SPEED2GREG(v) (7500/(((v)/TIRE_CIRCUIT)))と定義しています。
この式は最適化された後であり、何が何だかわからないと思います。わからなくなったら、公式に当てはめて計算します。使う公式は
時間 = 距離÷速度
です。まず、距離を求めます。ステッピングモータは割り込みごとに 1pulse 進むので、1pulse の距離を求めます。
TIRE_DIAMETER×PI/400 = TIRE_CIRCUIT/400[mm]
次に時間を求めます。MTU2 のクロックは、3MHz で、TGRC 値の時間、つまり割り込み時間は
TGRC÷3MHz[t]
公式に当てはめると、
TGRC÷3MHz = TIRE_CIRCUIT/400 /V
TGRC = TIRE_CIRCUIT /400 /v ×3MHz
TGRC = 7500 × TIRE_CIRCUIT ÷ v
と define で定義された式と同じになりますが、7500×TIRE_CIRCUIT を計算すると 7500 ×48 ×3.14 = 1130400 となり、16bit を超えてしまいます。そこで、先に TIRE_CIRCUIT/v を先に計算することでオーバーフローを抑えています。
int_mot_r と int_mot_r は先ほどタイムアウトプットマスタ許可レジスタで解説したとおり、PWM の立ち上がりの割り込み先が、int_mot_r と int_mot_r となり、パルスをカウントして、距離を測定しています。

プログラムソース 3: run.c からの抜粋

```
void straight(int len, int tar_speed) //直線走行
{
        int obj_step;                   //目標ステップ数
        r_accel = accel;                //加速度(後で指定方法を変更すること)
        step_r = step_l = 0;            //ステップ数カウントのリセット
        obj_step = LEN2STEP(len);       //目標ステップ数を算出
        MOT_CWCCW_R = MOT_CWCCW_L = MOT_FORWARD;//前方に進む
        MTU.TSTR.BIT.CST3 = MTU.TSTR.BIT.CST4 = 1;      //カウントスタート
        //目標速度が最低速度を下回らないようにする
        if(tar_speed < MIN_SPEED)
        {
                tar_speed = MIN_SPEED;
        }
//減速を開始し始めるところまで待つ(あと走るべき距離が減速するべき距離より短くなったら・・・)
        while( (len - STEP2LEN(step_r + step_l) ) > ( ((tar_speed*tar_speed) -(speed*speed) ) /
(-2*1000*accel) ));
        r_accel = -accel; //減速する
        while((step_r + step_l) < obj_step);        //目標地点まで走行
        MTU.TSTR.BIT.CST3 = 0;                      //モータのカウントをストップ
        MTU.TSTR.BIT.CST4 = 0;                      //モータのカウントをストップ
  }
```

動作確認と解説

この straight()という関数に走りたい距離(mm)と目標速度を入れ、最高速度(max_speed[mm/s])と加速度(accel[m/s])を与えると走らせることができます。

18 行目の step_r = step_l = 0;で距離=パルス数を初期化しています。

24 行目の MTU.TSTR.BIT.CST3 = MTU.TSTR.BIT.CST4 = 1;で PWM を動作しています。

40、41 行目の MTU.TSTR.BIT.CST3 = 0;で PWM を停止しています。

34 行目でコメントにあるように減速を開始し始めるところまで加速、または、指定した max_speed で走行します。

36 行目で、加速度をマイナスにし減速方向にます。

38 行目で目標値まで tar_speed 以下にならないところまで減速しながら走行します。

34 行目を少し解説します。

len は straight 関数の引数として宣言されています。オリジナルでは 180 が入っています。STEP2LEN(step_r+step_l)で走行したパルス数から距離を算出しています。STEP2LEN は"macro.h"に

#define STEP2LEN(s)　　(TIRE_CIRCUIT*(s)/(2*400))と定義しています。算出方法は LEN2STEP の逆数なので説明は省略します。

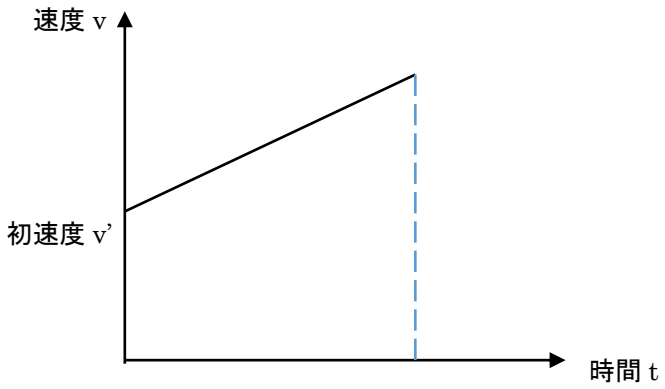
(speed*speed) − (tar_speed*tar_speed)) / (2*1000*accel)は、加速度と速度と距離の関係式になります。

この式には、時間が含まれておらず、あまり目にすることはないと思います。高校の物理でよく見かける公式は、

速度 v = 加速度 a × 時間 t + 初速度 v'

ですが、$v^2-v'^2=2aL$ という速度と距離の関係式があり、これを知っているとすぐに納得できると思います。直感的にはわからないので、v=at+v'の公式から距離を求めてみます。v=at+v'の式をグラフにすると次頁の図のようになりしま

す。



上記の台形の面積を求めると距離になります。台形の面積の公式は、(上底+下底)×高さ÷2 から

$$距離 = \frac{1}{2}(v + v')t$$

となります。速度の式 v=at+v' から t=求めると

$$t = \frac{v - v'}{a}$$

この式を距離の式に代入すると

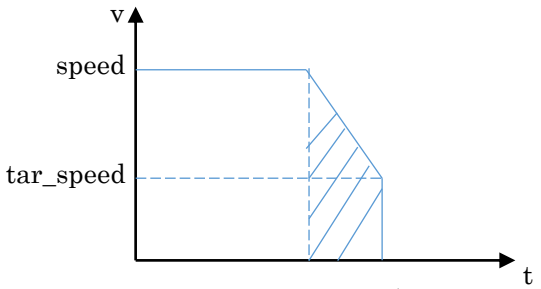
$$距離 = \frac{1}{2}(v + v') \cdot \frac{v - v'}{a} = \frac{v^2 - v'^2}{2a} \cdots ①$$

v を speed、v' を tartget_speed と置き換えるとこれで 34 行目の不思議な式になると思います。
34 行目の式

```
while( (len - STEP2LEN(step_r + step_l) ) > ( ((tar_speed*tar_speed) -(speed*speed) ) / (-2*1000*accel) ));
```

1000で割っているのは単位を合わせるために1000で割っています。speedとtar_speedは[mm/s]、accelは[m/s/s]なので、accelの単位を[mm/s/s]にするためaccelに1000をかけています。

コメントにもありますようにこの条件は減速を開始するタイミングを求めています。減速ですので、加速度はマイナス(−)にすると①の公式から初速度が−v'²なのでspeed、今の速度がtartgetとなります。



こんなグラフになるかと思います。ハッチングされた面積が減速に必要な距離となります。これで、減速を開始するタイミングであることが分かったと思います。

プログラムソース 4: PiCoClassic3 からの抜粋

```
case 1:
    max_speed = 2000.0;
    accel = 1.5;
    r_accel = 1.5;
    straight(180,0);
    wait_ms(1000);
    break;
```

動作確認と解説

straight 関数を呼び出す前に最高速度 max_speed と加速度 accel を設定しています。

straight 関数の目標速度の引数が 0 なのかというと、この引数 tar_speed は目標位置での速度を表しています。目標位置で止まりたい場合はここに 0 を入れるようにしてください。0 でなくとも止まりますが、急激に止まることになるため、目標の位置とずれてしまいます。目標の位置に正確に止まるには、緩やかに止まるようにするのがベストです。

この例では、一区画入るプログラムになっています。実際に走らせて動作を確認してみてください。

1.9. Step7：フィードバック制御をしよう

概要

今回はセンサでフィードバック制御をしながら走行し、迷路の真ん中を走らせるプログラムになっています。

プログラムソース:interrupt.c からの抜粋

```
void int_cmt0(void)
{
…
//センサ制御
if(con_wall.enable == true)//壁制御が許可されているかチェック
{
        con_wall.p_error = con_wall.error;          //過去の偏差を保存
//左右のセンサが、それぞれ使える状態であるかどうかチェックして、姿勢制御の偏差を計算
        if( ( sen_r.is_control == true ) && ( sen_l.is_control == true ) )
        {//両方とも有効だった場合の偏差を計算
                con_wall.error = sen_r.error - sen_l.error;
        }
        else//片方もしくは両方のセンサが無効だった場合の偏差を計算
        {
                con_wall.error = 2.0 * (sen_r.error - sen_l.error);
                //片方しか使用しないので2倍する
        }
        //P制御計算
        con_wall.diff = con_wall.error - con_wall.p_error;  //偏差の微分値を計算
        con_wall.sum += con_wall.error;                     //偏差の積分値を計算
        if(con_wall.sum > con_wall.sum_max)     //偏差の積分値の最大値を制限
        {
                con_wall.sum = con_wall.sum_max;
        }
        else if(con_wall.sum < (-con_wall.sum_max))//偏差の積分値の最低値を制限
        {
                con_wall.sum = -con_wall.sum_max;
        }
        con_wall.control = 0.001 * speed * con_wall.kp * con_wall.error;//操作量を計算
        spd_r = speed + con_wall.control; //制御を左右モータの速度差という形で反映
        spd_l = speed - con_wall.control;  //制御を左右モータの速度差という形で反映
```

```
}else{
        spd_r = speed;
        spd_l = speed;
}
        MTU3.TGRC = SPEED2GREG(spd_r);//右速度(mm/s)からジェネラルレジスタの値に変換して、バッファレジスタに代入
        MTU4.TGRC = SPEED2GREG(spd_l);//左速度(mm/s)からジェネラルレジスタの値に変換して、バッファレジスタに代入
        timer++;                        //1mSごとにカウントアップ
}
void int_cmt1(void)             //センサ読み込み用り込み
{
        static int state = 0;       //読み込むセンサのローテーション管理用変数
        int i;
        switch(state)
        {
                case 0:             //右センサ読み込み

                        SLED_R = 1;                     //LED点灯
                        for(i = 0; i < WAITLOOP_SLED; i++);//フォトトランジスタの応答待ちループ
                        S12AD.ADANS0.BIT.ANS0=0x0040;//AN006
                        S12AD.ADCSR.BIT.ADST=1;     //AD変換開始
                        while(S12AD.ADCSR.BIT.ADST);//AD変換終了まで待つ
                        SLED_R = 0;                     //LED消灯
                        sen_r.p_value = sen_r.value;        //過去の値を保存
                        sen_r.value = S12AD.ADDR6;      //値を保存
                        if(sen_r.value > sen_r.th_wall)     //壁の有無を判断
                        {
                                sen_r.is_wall = true;       //右壁あり
                        }
                        else
                        {
                                sen_r.is_wall = false;      //右壁なし
                        }

                        if(sen_r.value > sen_r.th_control)//制御をかけるか否かを判断
                        {
                                sen_r.error = sen_r.value - sen_r.ref;//制御をかける場合は偏差を計算
                                sen_r.is_control = true;    //右センサを制御に使う
                        }
```

```
                else
                {
                        sen_r.error = 0;//制御に使わない場合は偏差を0にしておく
                        sen_r.is_control = false;   //右センサを制御に使わない
                }
                break;
```

動作確認と解説

このサンプルの CS+プロジェクトは「C:¥20170407Documents¥Sample_Program¥step7_P_control_run_usb」フォルダ内の「PiCoClassic3.mtpj」です。それをビルドし、RFP を使って、「C:¥20170407Documents¥Sample_Program¥step7_P_control_run_usb¥DefaultBuild」フォルダ内の「PiCoClassic3.mot」を書き込みます。

今回はモータの割り込みと前回のセンサ割り込みにセンサからのフィードバック制御を追加しています。具体的には壁があるかないか判定し、ある場合その壁からの自分の位置を測定し、PID 制御の P 制御をかけています。

制御についての解説に入ります。

今回は P 制御しか入れていないので、制御式は

・偏差＝目標値 － 現在値

操作量=(偏差)＊ゲイン数

という制御式になっています。プログラムでは下記のようになっています。

sen_r.error = sen_r.value (現在値)－ sen_r.ref(目標値)

これは右モータ用制御式です。なぜ目標値と現在地の符号が逆になっているかというと、

モータ用 PWM のジェネラルレジスタ(MTU3.TGRC、MTU4.TGRC)は値が小さくなればなるほど周期が速くなります。

PWM の周波数を決めるジェネラルレジスタに代入する値の式の分母に操作量を加えた速度の値が入るので、操作量が増えれば増えるほど速度が速くなります。具体的には以下のようになっています。

```
con_wall.error = sen_r.error － sen_l.error;
//両方とも有効だった場合の偏差を計算

con_wall.control = 0.001 * speed * con_wall.kp * con_wall.error;
//操作量を計算

spd_r = speed + con_wall.control;
//制御を左右モータの速度差という形で反映

MTU3.TGRC = SPEED2GREG(spd_r);
//右速度(mm/s)からジェネラルレジスタの値に変換して、バッファレジスタに代入

SPEED2GREG(v) = (7500/(((v)/TIRE_CIRCUIT)))
//スピードからジェネラルレジスタの値を計算
```

ここで、v = spd_r となっています。

この場合、真ん中より右寄りになると右センサの現在値が目標値より大きくなり偏差値はプラスになります。その偏

差値を操作量に計算して、ジェネラルレジスタを決める式に代入することにより下図のように右モータの速度が速くなるので真ん中に戻ろうとします。これは目標値と現在値との間に偏差がなくなるまで続けられます。このように目標とする値と現在値の差を縮めていく制御をフィードバック制御といいます。

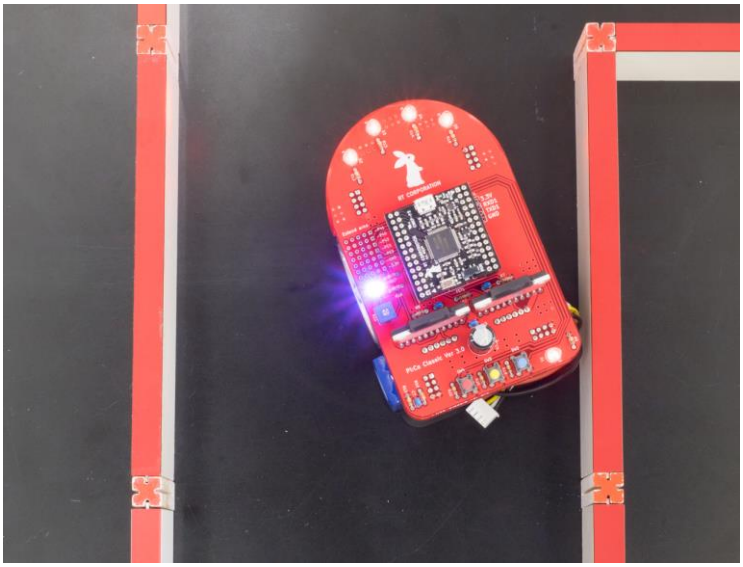


図 7-1

ちなみに左モータ用制御式の場合、左の速度に制御量を加える式が

```
spd_l = speed  ー  con_wall.control;
```

と右モータ用制御式と比べて符号が逆になっているので、制御を左右モータの速度差という形で反映しています。

【コラム】PID 制御について

Pi:Co Classic3 ではセンサからのフィードバックによる比例(P)制御を導入しています。しかし、比例制御だけですと目標値に緩やかに近づくだけで、反応がやや遅れ気味になります。この問題を解決するのに微分項(D 項)を追加する方法があります。微分項を追加することにより反応が P 制御だけの時よりも反応が早くなります。しかし、微分項のゲインを大きくしすぎると環境の変化に敏感になりすぎてしまいます。例えば、壁の切れ目に反応しやすくなったり、センサのノイズに反応しやすくなったりします。

ただし、PD 制御を導入しても目標値にすばやく近づくことはできるのですが、厳密には目標値にはなりません。この目標値と現在地との偏差を定常偏差(オフセット)といいます。この問題を解決するためには積分項(I 項)を追加します。現在 Pi:Co Classic 3 で使用している反射式光センサ(可視光センサ)では定常偏差は制御するのに気にならない程度の影響しかでないので積分項を追加する必要はないと思われます。以下にプログラム上での PID 制御の式を記します。

$$制御量 = 現在の偏差 * K_P + \left(\sum 偏差\right) * K_I + \left(現在の偏差 - 一つ前の偏差\right) * K_D$$

ここで、

$K_P = 比例ゲイン$

$K_I = 積分ゲイン$

$K_D = 微分ゲイン$

となります。

intrrupt.c の 57 行目から 67 行目までのプログラムで偏差の微分値 con_wall.diff と偏差の積分値である con_wall.sum を計算しています。これらの値を P 制御式に前述の式のように追加すれば PID 制御式がつくれます。ただし、積分項を追加する場合、積分値に上限と下限を設定しなければなりません。設定しなければ偏差が大きすぎる場合などに値が際限なく大きくなってしまい、誤作動を起こす原因になります。特に DC モータを使用するマウスやロボ

ットを製作、制御する場合にこの設定をしないと最悪モータが焼き切れる場合があるので注意してください。

動作確認とセンサ用変数の設定方法

別紙 クラシックサイズマイクロマウス Pi:Co Classic3 操作説明書「本体組立完成後 まとめ」の 3-2-3 センサ用パラメータの設定を見てセンサのパラメータを設定してください。
センサの調整モードに移行するには、モード 15(表示用 LED を左のステッチを押して 4 つすべて点灯)に、真ん中のスイッチを押すと teraTerm にセンサの値を表示します。

モード 1 に実際に走らせる処理を入れてあるので参照してください。
迷路区間の真ん中を走りましたか？挙動がおかしい場合は parameters.h 内の比例制御のゲイン数 CON_WALL_KP の値を調整してください。あまり値を大きくしすぎると機体が不安定になってしまうので、気をつけましょう。

1.10. Step8：迷路を走破しよう

概要

ついに迷路を走破する時がきました。ここまでの調整やプログラムはこのためにあったと言っても過言ではありません。迷路走破のためのプログラムを見ていきましょう。

またバッテリの過放電対策のため、バッテリがある電圧（10V 程度）以下になると電源が切れるまでブザを鳴らし、LED を消灯させるプログラムが組み込まれていますので注意してください。

動作確認

このサンプルの CS+プロジェクトは「C:¥20170407Documents¥Sample_Program¥step8_running_maze_usb」フォルダ内の「PiCoClassic3.mtpj」です。それをビルドし、RFP を使って、「C:¥20170407Documents¥Sample_Program¥step8_running_maze_usb¥DefaultBuild」フォルダ内の「PiCoClassic3.mot」を書き込みます。

プログラムの解説

init.c：初期設定全般が入ったファイル

今回は今までのユニットの初期設定の他に迷路情報の初期化が追加されています。マイクロマウス競技のルールに則って外枠の壁及び自分の前以外の壁(初期位置の右の壁)の情報を更新しています。

intrrupt.c：割り込み処理全般が入ったファイル

今までのモータ割り込みとセンサの値取得割り込み、そしてフィードバック制御の割り込みが入っています。

parameters.h：各種パラメータを設定ファイル

走行用パラメータやセンサ用パラメータ、ゴール座標の設定などをするためのプログラムです。走行速度を変えたいときやゴール座標を変えたい場合はこのプログラムを見ましょう。

search.c：迷路探索アルゴリズムの入ったファイル

迷路を走破するための肝になっているプログラムです。歩数マップ生成関数、壁情報の保存、優先度の取得、足立法、左手法の関数などが入っています。詳しくは、別紙の”クラシックサイズマイクロマウス Pi:Co Classic3 パート 1「基礎知識編」の 3.迷路解析の手法と考え方”を見てください。

fast.c：最短走行用ファイル

最短走行用関数 fast_run が入ったファイルです。詳しくは、別紙の”クラシックサイズマイクロマウス Pi:Co Classic3 パート 1「基礎知識編」の 3.7 最短経路の求め方”を見てください。

PiCoClassic3.c：main 関数の入っているファイルです

モード 1 に左手法、モード 2 に足立法でゴールを目指し、帰りにまた足立法でスタートまで帰り探索をするプログラムになっています。モード 3 には最短走行プログラムになっています。

```
case 1:     //左手法
    search_lefthand();
    break;

case 2:     //足立法で行って帰ってくる
    mypos.x = mypos.y = 0;              //座標を初期化
    mypos.dir = north;                  //方角を初期化
    search_adachi(GOAL_X,GOAL_Y);       //ゴールまで足立法
    rotate(right,2);                    //ゴールしたら180度回転する
    mypos.dir = (mypos.dir+6) % 4;      //方角を更新
    goal_appeal();                      //ゴールしたことをアピール
    search_adachi(0,0);                 //スタート地点まで足立法で帰ってくる
    rotate(right,2);                    //帰ってきたら180度回転
    break;

case 3:     //最短走行
    mypos.x = mypos.y = 0;              //座標を初期化
    mypos.dir = north;                  //方角を初期化
    accel = 2.0;                        //加速度を設定
    fast_run(GOAL_X,GOAL_Y);            //ゴールまで最短走行
    mypos.dir = (mypos.dir+6) % 4;      //方角を更新
    rotate(right,2);                    //ゴールしたら180度回転
    goal_appeal();                      //ゴールしたことをアピール
    fast_run(0,0);                      //スタート地点まで最短走行
    rotate(right,2);                    //帰ってきたら180度回転
    break;
```

迷路を走らせるには

Pi:Co Classic 3 のプログラムには左手法か足立法でゴールする二種類の関数が用意されています。このうち足立法でゴールする場合は main 関数のモード 2 に書いてあるように

1. スタート地点の座標を入力(座標の初期化)
2. スタート地点で自分の向いている方向を入力(方角の初期化)
3. 足立法関数の呼び出し

この手順で設定して頂ければマウスは足立法で迷路のゴールを目指してくれます。

上記のプログラムではゴールからの帰りにも探索をするように探索用関数 search_adaci()で目指す座標をスタートの 0,0 にすることでゴールからスタートまで探索しながら帰ってきます。

また最短走行の場合は最短走行関数 fast_run()を使用して、加速度を与える以外は探索時と同じ手順で最短走行ができます。

モード 15 でセンサの値の確認、距離の確認などができるようになっています。そのため、今までのようにモード 15 に入ったらセンサの値がすぐに表示するのではなく、モード 15 のモード 1 でセンサの値が表示するようになっています。これまでと同じ感覚でセンサの値を見るには、モード 15 にして真ん中のスイッチを 2 回押す必要があります。

詳細は別紙の「クラシックサイズマイクロマウス Pi:Co Classic 3 操作説明書」に操作法が書いてあります。

これで全ての解説を終わります。マイクロマウスは迷路をゴールしてからが始まりです。

これからはアルゴリズムを改良したり、より速く、安定して走れるようにするなど自分なりに工夫してマイクロマウスを楽しんでください。できることならば、この Pi:Co Classic 3 で学んだことを生かし新しく自分で設計、製作したマイクロマウスで大会に参加するなどして、楽しんで頂けたら幸いです。

1.11. Step9：迷路情報、センサ値を記憶しよう

概要

Step8 で迷路の走破ができたと思います。しかし、やっとの思いでゴールまでたどり着いたの帰り道でぶつかってしまい、電源を OFF にせざるを得ず、探索から始めるということを経験した方もいらっしゃるでしょう。Pi:Co Classic3 で採用している RX631 はデータフラッシュメモリを搭載しており、そのデータフラッシュメモリに書き込んだデータは電源を OFF にしてもデータが消えない特徴があります。この特徴を生かし、ゴールにたどり着いたときにそれまでの走行したマップをデータフラッシュメモリに書き込むようにしました、この機能を実装したことで、ゴールにさえ入ってしまえば、帰りにぶつかって電源を OFF にしても再度探索をせずに最短走行をすることができます。

また、センサのキャリブレーション時に PC に接続して値を確認し CS+でパラメータを書き直し->プログラムをビルド->プログラムの書き込みをしていましたが、モード 15 でセンサの値を確認しながらデータフラッシュに記憶することができるようにしてあります。

動作確認

このサンプルの CS+プロジェクトは「C:¥20170407Documents¥Sample_Program¥step9_running_maze_Dataflash_usb」フォルダ内の「PiCoClassic3.mtpj」です。それをビルドし、RFP を使って、「C:¥20170407Documents¥Sample_Program¥step9_running_maze_Dataflash_usb¥DefaultBuild」フォルダ内の「PiCoClassic3.mot」を書き込みます。

プログラムの解説

モード 2 の足立法で探索し、ゴールに入ると LED が点滅すると思います。ゴールに入る goal_appeal()関数が呼ばれ、その関数内で LED の点滅をさせています。その関数内で”map_write();”の関数を呼び出し、探索したマップデータをデータフラッシュメモリに書く作業をしています。

DataFlash.c から抜粋

```
void map_write(void)
{
        short i;
        unsigned short *map_add;
        map_add = (unsigned short *)&wall;

        //DataFlashイレース
        for(i=0;i<8;i++){
                erase((unsigned short *)(MAP_ADD+i*32));
        }
        //マップデータをDataFlashに書き込む
        for(i=0;i<128;i++){
                write_eeflash((unsigned short *)(MAP_ADD+i*2),*map_add);
                map_add++;
        }
  }
```

データフラッシュの詳細は、RX631 ハードウェアマニュアルの”47. フラッシュメモリ”を見てください。RX631 のフラッシュメモリには ROM と E2 データフラシュの二種類あります。ROM の方にはマイクロマウスのプログラムを書き込んで使用しています。E2 データフラッシュは、電源を OFF にしてもデータを保存したいものを書き込む領域です。この E2 データフラッシュ(以降データフラッシュと省略)の書き込みは 2 バイト、イレース(消去)は 32 バイトと小さいサイズで書き込み、イレースが可能となっています。迷路の map データは最大 16x16=256 バイトの領域が必要です。このデータをすべてデータフラッシュに書き込むには、書き込みでは 128 回(256/2byte)、イレースでは 8 回(256/32byte)必要です。データを書き換えるには、一度イレースしてから書き込む必要があります。
USB ブートモード(USB でプログラムを書き換えるモード)では、データフラッシュに書き込んだデータをプロテクトすることができません。USB ブートモードにするとデータフラッシュ内がすべてイレースされます。データフラッシュにあるデータをプロテクトするには、UART によるブートモードにする必要があります。

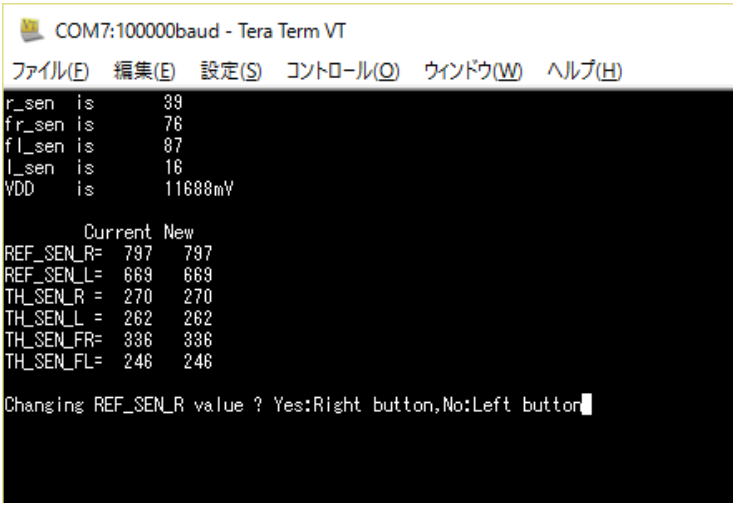
データフラッシュに保存したマップ情報は最短走行時(モード 3)の最初に map_copy()関数でデータフラッシュから RAM にあるマップ情報に上書きされます。

DataFlash.c から抜粋

```
void map_copy(void)
{
	short i;
	unsigned short *map_add;
	map_add = (unsigned short *)&wall;
	//マップデータをRAMにコピー
	for(i=0;i<128;i++){
		*map_add = *(unsigned short *)(MAP_ADD+i*2);
		map_add++;
	}
}
```

センサ調整時の値を保存するサンプルプログラムは adjust.c にあります。

モード 15 でセンサの値を確認すると以下のような画面になります。



一番下に Changing REF_SEN_R value ? Yes:Right Button,No:Left Button ののメッセージが出力されていると思います。何もしなければ、REF_SEN_R、REF_SEN_L、TH_SEN_R、TH_SEN_L、TH_SEN_FR、TH_SEN_FL の値を書き換えません。書き換えたいときに、マイクロマウスをその場所において右のボタンを押してください。書き換えの問い合わせの順番は、REF_SEN_R、REF_SEN_L、TH_SEN_R、TH_SEN_L、TH_SEN_FR、TH_SEN_FL になります。たとえば、REF_SEN_R の値を変更する必要がなく、REF_SEN_L の値を変更したい場合は、REF_SEN_R の問い合わせの時に左ボタンを押すと、値を変更せずに次の REF_SEN_L の問い合わせになりますので、マイクロマウスを迷路の真ん中に置いて右ボタンを押してください。まだ、この状態ではデータフラッシュに書き込まれていません。"Changing value write? Yes:Right button,No:Left button"のメッセージが出て、右ボタンを押したときにはじめてデータフラッシュに書き込みます。

2. 故障かな? と思ったら

以下に主に起こりうる故障についての解決策をあげます。どうしても直らない、解決しない場合は弊社までお問い合わせください。

●電源スイッチを入れても何も反応しない	
電池をコネクタにきちんと接続していますか？	意外と忘れてしまうときがあります。落ち着いてもう一度接続してみましょう。
電池のコネクタがとれていませんか？	何度もコネクタの抜き差しを行うと取れてしまう場合があります。
電池の容量は充分にありますか？	充電をしなおしてみてください。
電源基板と他の基板の接続はしっかりしていますか？	基板をさしなおしてみてください。
●センサの調子がおかしい	
基板間の接続はしっかりしていますか？	基板をさしなおしてみてください。
プログラムにミスはないですか？	絶対に入らないような論理式を立てていたりしていないか、無限ループに入っていないかなどを確認しましょう。
閾値が高すぎたり、低すぎたりしていませんか？	まずはシリアル通信でセンサの情報を調べてみましょう。
発光部・受光部の方向が変な方向に向いていませんか？	直接衝撃が加わった時などにおこります。ずれた場合は調整が必要です。
●まっすぐ進まない、動かない	
基板間の接続はしっかりしていますか？	基板をさしなおしてみてください。
基盤を支えるスペーサーが緩んでいませんか？	ねじが緩んでいると正確に力が伝わらないことがあります。
ゴムタイヤがずれていませんか？	ゴムリングが車輪からずれてしまうことがあります。確認してみてください。
姿勢制御や走行距離のパラメータは適正ですか？	さまざまな要因によりずれが生じてしまうことがあります。パラメータを調整してみましょう。
プログラムにミスはないですか？	コンパイルが成功していたとしても、動作しない場合があります。もう一度プログラムを確認してみましょう。
テフロンシートの貼りつけ位置は合っていますか？	土台の一番下に張っているテフロンシートの位置を貼り直して調整してみてください。

3. 開発のヒント

ここでは Pi:Co Classic 3 を使用してこれから新たに何か回路を積むなどの開発を行う人のために Pi:Co Classic 3 の情報を公開します。自分だけの新しいマイクロマウスの開発のヒントになれば幸いです。

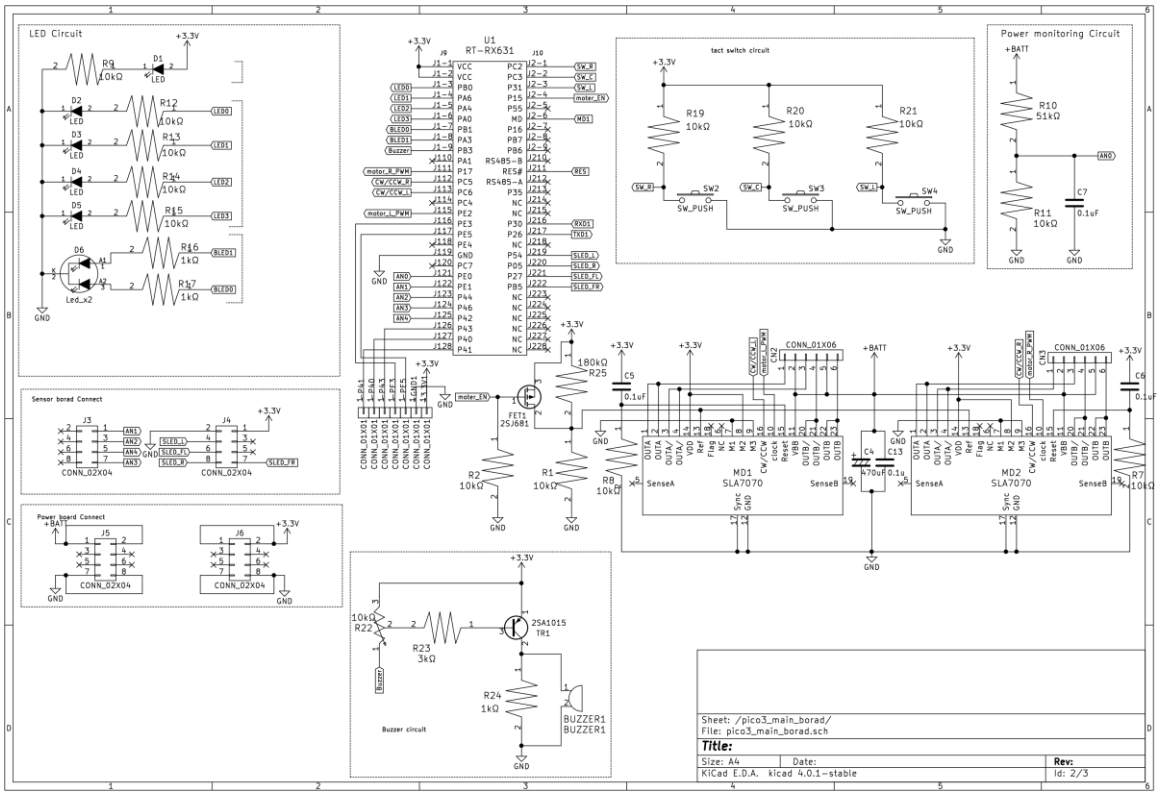
3.1. 基板ポート一覧

J9		
No.	信号名	機能
1	VDD	3.3V
2	VDD	3.3V
3	RSPCKA/PB0	確認用 LED
4	MOSIA/MTCLKB/PA6	確認用 LED
5	SSLA0/MTCLKA/PA4	確認用 LED
6	SSLA1/PA0	確認用 LED
7	MTIOCOC/PB1	バッテリ監視用 LED
8	MTIOCOD/MTCLKD/PA3	バッテリ監視用 LED
9	MTIOCOA/PB3	ブザ PWM
10	MTIOCOB/MTCLKC/PA1	
11	MTIOC3A/MISOA/P17	右モータ PWM
12	MTIOC3B/RSPCKA/PC5	右モータ CW/CCW
13	MTIOC3C/MOSIA/PC6	左モータ CW/CCW
14	MTIOC3D/SSLA0/PC4	
15	MTIOC4A/MOSIB/PE2	左モータ PWM
16	MTIOC4B/MSIOB/PE3	増設ポート
17	MTIOC4C/PE5	増設ポート
18	MTIOC4D/SSLB0/PE4	
19	GND	GND
20	PC7	
21	AN008/SSLB1/PE0	電圧監視
22	AN009/RSPCKB/PE1	右前センサ
23	AN004/P44	左前センサ
24	AN006/P46	右センサ
25	AN002/P42	左センサ
26	AN003/P43	増設 A/D
27	AN000/P40	増設 A/D
28	AN001/P41	増設 A/D

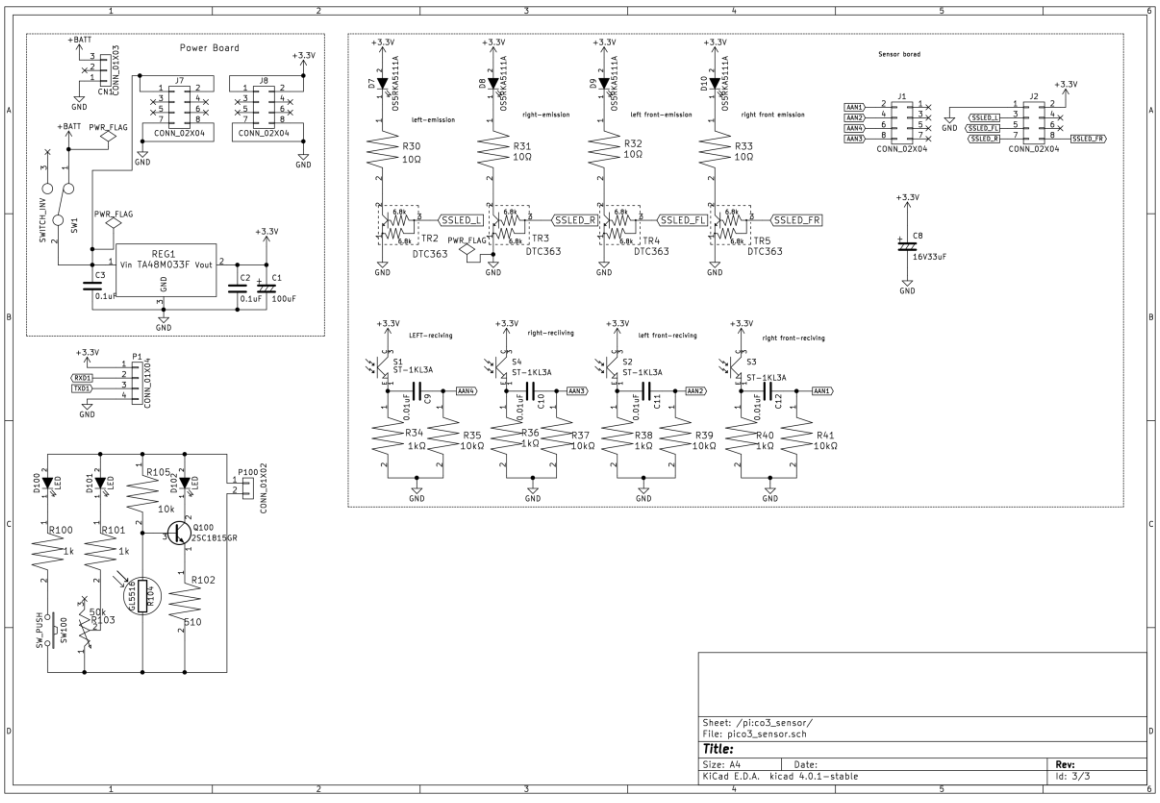
J10		
No.	信号名	機能
1	SSLA3/PC2	タクトスイッチ R
2	MTIOC4D/PC3	タクトスイッチ C
3	MTIOC4D/SSLB0/P31	タクトスイッチ L
4	MTIOC0B/P15	モータ ON/OFF 切り替え (アクティブ H)
5	MTIOC4D/P55	
6	MD/FINED	書き込みモード切替
7	P16	
8	TXD9/PB7	
9	RXD9/PB6	
10	RS485	
11	RES#	リセット(アクティブ L)
12	RS485	
13	P35	
14	OPEN	
15	OPEN	
16	RXD1/P30	シリアル通信
17	TXD1/P26	シリアル通信
18	OPEN	
19	MTIOC4B/P54	センサ LED 左
20	DA1/P05	センサ LED 右
21	MTIOC2B/RSPCKB/P27	センサ LED 左前
22	MTIOC1B/PB5	センサ LED 右前
23	OPEN	
24	OPEN	
25	OPEN	
26	OPEN	
27	OPEN	
28	OPEN	

3.2. 基板回路図

メイン基板回路図



センサ基板回路図、電源基板回路図



3.3. プログラムの詳解

ダウンロードフォルダにあるサンプルプログラムのフォルダ構成は次のとおりです。

¥Sample_program
- Step0_Check
- Step1_1_LED_wait
- Step1_2_LED_PWM
- Step2_Buzzer_wait
- Step3_1_Switch
- Step3_2_Switch_changing_mode
- Step4_Buzzer_do_le_mi_PWM
- Step5_sensor_check_usb
- Step6_1_run_wait
- Step6_2_run_ST_PWM
- Step6_3_run_ROT_PWM
- Step7_P_control_run_usb
- Step8_running_maze_usb
- Step9_running_maze_Dataflash_usb

各サンプルソフトのフォルダの中にある“PiCoClassic3”フォルダ内に本体のプログラムソースが入っています。構成としては以下のとおりです。

¥Sample_Program ¥Step9_running_maze_Dataflash¥PiCoClassic3
- adjust.c
- DataFlash.c
- DataFlash.h
- dbsct.c
- demo.c
- fast.c
- glob_var.c
- glob_var.h
- init.c
- init.h
- interrupt.c
- interrupt.h
- intprg.c
- iodefine.h
- macro.h
- misc.c
- mytypedef.h
- parameters.h
- PiCoClassic3.c
- portdef.h
- resetprg.c
- run.c
- run.h
- sbrk.c
- sbrk.h
- sci.c
- sci.h
- search.c
- stacksct.h
- static_parameters.h
- typedefine.h
- usb.c
- usb.h
- usb_define.h
- usb_int.c
- usb_int.h
- vect.h
- vecttbl.c

同フォルダ内に hwsetup.c、lowlvl.src、lowsrc.c、lowsrc.h のファイルは使用しておりません。RX631 の環境作成時に作成できるファイルになります。

3.4. シリアル通信(SCI)について

マイコンボードの右に 3.3V、RXD1、TXD1、GND と記述されたポートがあります。このポートを利用してほかの機器とシリアル通信を行うことができます。
サンプルプログラムでは使用しませんが、step9_running_maze_Dataflash_usb の環境において
通信速度 : 38400bps
パリティ : なし
stop ビット : 1
データ : 8bit
で通信できるように初期化されています。簡易的な printf(SCI_printf)も用意してあります。シリアル通信を用いるといろいろいなデバイスにつなぐことができます。たとえば、USB では、デバックが難しい場合、このシリアル通信のポートに無線で飛ばせるデバイスを接続すれば、走行しながら内部の情報を知ることができます。
また、USB を使用しなくとも、シリアルポートを使ってプログラムの書き換えも可能です。その場合、ブートモードを USB からシリアルに変換する必要があり、J9 の 19 と 20 をショート(短絡)させるとシリアル通信のブートモードになります。

4. 著作権について

本取扱説明書で紹介、または記載されている会社名、製品名は、各社の登録商標または商標です。

本取扱説明書に掲載している文書、写真、イラストなどの著作物は、日本の著作権法及び国際条約により、著作権の保護を受けています。

5. ソフトウェアについて

ルネサスエレクトロニクス製ソフトウェアについて

ダウンロードしたルネサスエレクトロニクス製ソフトウェアは、サポート対象製品ではありません。サポートは一切行われませんので、あらかじめご了承ください。

すべての収録ファイルについて

ダウンロードしたすべての収録ファイル対して，その使用にあたって生じたトラブル等は、ルネサスエレクトロニクス(株)、および(株)アールティは一切の責任を負いません。

インターネット等の公共ネットワーク、構内ネットワーク等へのアップロードなどは、ルネサスエレクトロニクス(株)および(株)アールティの許可無く行うことはできません。

6. 改訂履歴

発行日	ページ	改訂内容
2016/6	-	新規発行
2017/4		
	全頁	サンプルプログラムのフォルダー名を 20160601 から 20170407 に変更
	57	PID の言葉の定義が違っていたので修正。制御量 → 操作量

7. お問い合わせ

本製品に関するお問い合わせは、下記までお願いします。

株式会社アールティ

〒101-0021　東京都千代田区外神田 3-2-13 山口ビル 3F

E-mail: shop@rt-net.jp

URL :　http://www.rt-net.jp/