H8S/2238 学習キットマニュアル　V1.0

１ はじめに

このキットを開発しようと考えた切っ掛けは、ネット上の掲示板などで「C 言語は分かるが、実際にマイコンで実行させ、動くものを作るにはどうしたらよいか分からない」、「マイコンを使った電子工作に興味はあるが、どこからはじめればよいか分からない」といった質問をよく目にしたからです。

実際に、マイコンを使った電子工作は、プログラミングの知識だけでなく、ハードウエアの知識、開発環境に関する知識など広範囲な知識を必要とするため、初心者にはハードルの高いものとなっています。

当キットは、ハードウエアや開発環境を提供することにより、電子工作のハードルを下げ、楽しみながら必要な知識を習得していただくことを目的に開発しました。

ターゲットのマイコンには、以下の条件を満たす H8S/2238 を採用しました。

- アーキテクチャが一般的
- C++のプログラムも実行できる程度のメモリーを持つ
- 汎用的なフリーの開発環境である gcc が使える
- 電池で動く

当キットの学習ではブレッドボードを使ってハードウエアを増設するのでハンダ付けは不要です。学習が終わった後に万能基板を使って、機能拡張できるよう CPU の入出力を 2.54mm のピン間隔（万能基板に装着可能なピン間隔）に引き出してあります。また、このマイコンは小型で電池で動くので、ロボットなどに組み込むことも容易です。

一人でも多くの方が、当キットでの学習を通して、マイコンを使った電子工作の扉を開くことができれば幸いです。

２キット内容

１） EE_Type_S 2238 マイコン　　×１
２） H8S/2238　開発環境 DVD　　×１
３） ミニブレッドボード　　×１
４） ブレッドボード・ジャンパコード（オス-オス）　100mm　40 本
５） ブレッドボード・ジャンパコード（メス-オス）　100mm　40 本
６） 電子部品

- 電池ボックス　　×１
- RS232C レベル変換モジュール　×１
- DIP スイッチ　　×１
- LED　　×２
- ボリューム　　×１
- DC モーター　　×１

・ トランジスタ　　×２
・ 圧電ブザー　　×１
・ 抵抗　　330Ω×２、1kΩ×１、2.2 kΩ×１

３学習に必要なもの

１）RS232C ケーブル（http://akizukidenshi.com/catalog/g/gC-00004/ など）または USB-シリアル変換機（http://akizukidenshi.com/catalog/g/gM-02746/　など）

２）単４電池３本

３）パソコン（WindowsXP 以降）

４当キットを使って学習するために必要な知識

当キットの学習は、下記の知識がある前提で解説します。

- プログラミング言語（C 言語、C++言語）の知識
- オームの法則など初歩的な電気回路の知識
- Linux 上の操作
- Windows 上の操作

５開発環境のインストール

５-１フリーソフトのダウンロード

以下のソフトウエアをダウンロードしてください

１）VMware　Player　ダウンロードページ　http://www.vmware.com/jp/products/player/

２）TeraTerm　http://ttssh2.sourceforge.jp/

３）FDT 無償評価版　http://japan.renesas.com/

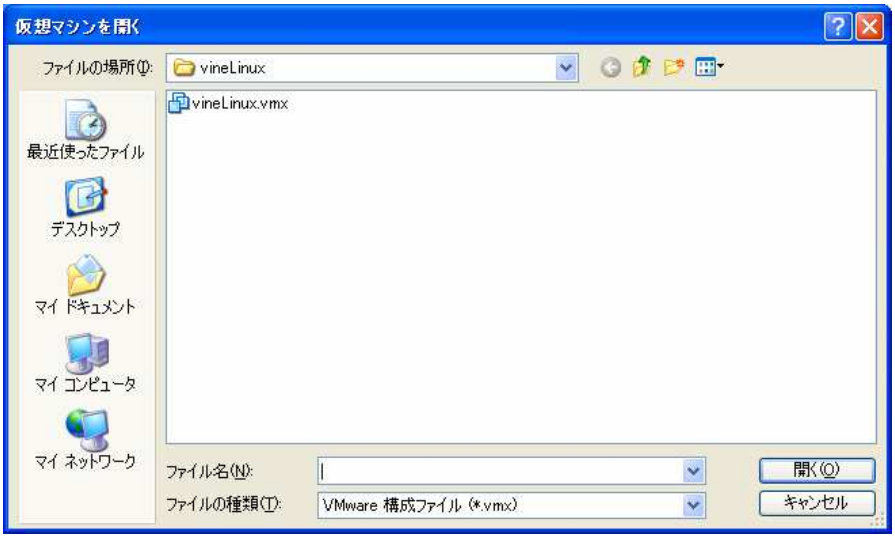
５-２VMware　Player のインストール

１）インストーラのガイドに従って VMware　Player をインストールする。

２）H8S/2238　開発環境 DVD の中の vineLinux.zip を PC の適当なフォルダに解凍する。

３）VMware　Player を起動し、仮想マシンを開くを選択する。

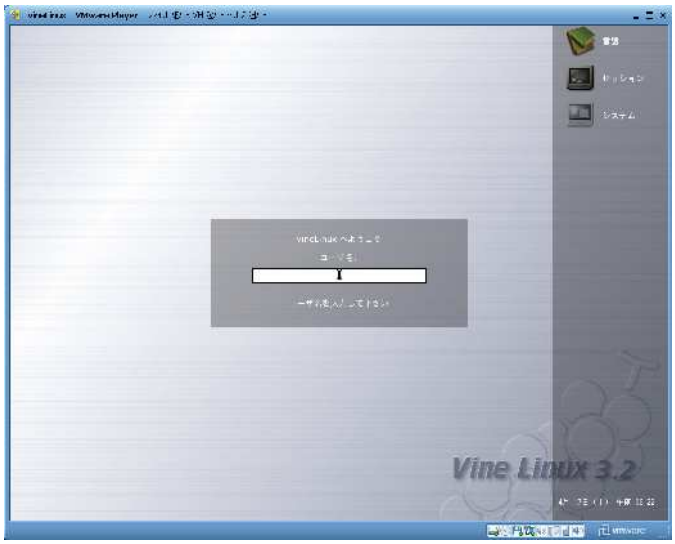
4）**2)**で解凍してできた vineLinux フォルダ中の vineLinux.vmx を選択する



5）VMware　Player の左のツリーに vineLinux が追加される

6）VMware　Player の仮想マシンの再生を選択し、vineLinux を起動する

7）vineLinux にログインします。



ユーザ gcc,パスワード vineLinux でログインしてください。

※　root のパスワードも vineLinux となっています。

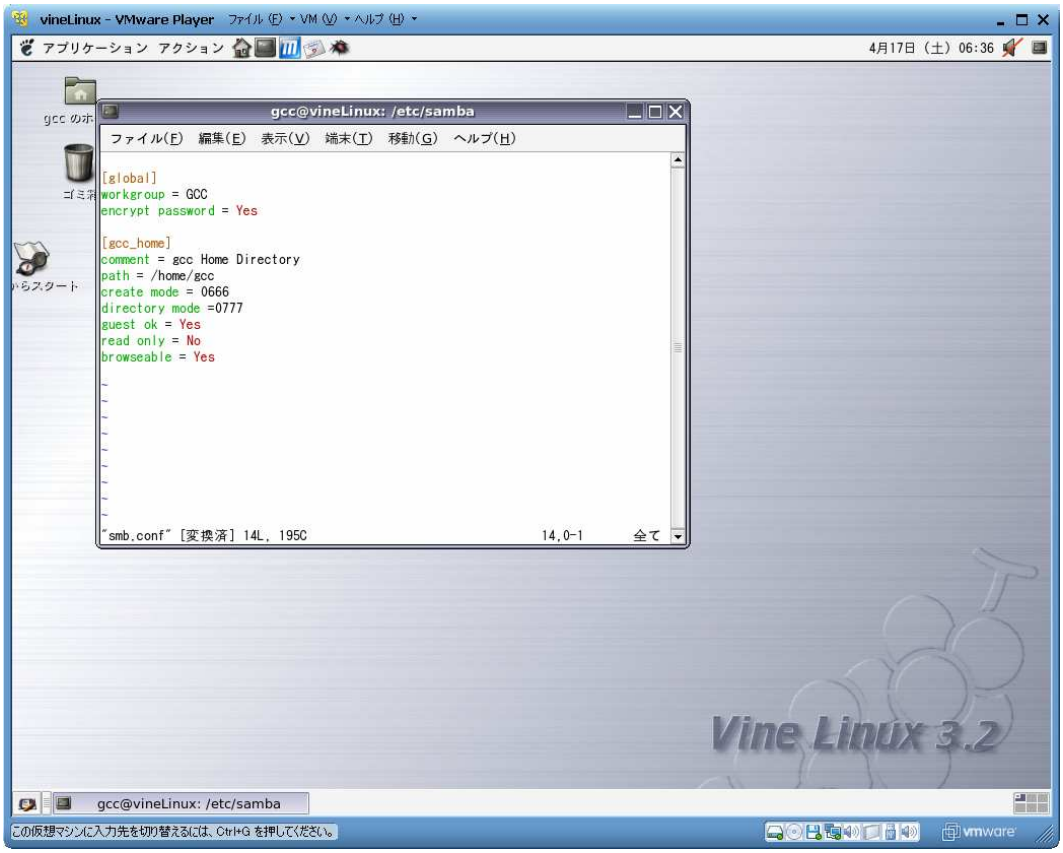
8）IP アドレスを設定する

[システムツール]-[ネットワークの設定]のダイアログで、ethr0 のプロパティを開いて仮想マシンの IP アドレスを設定してください。初期値は、192.168.0.70 になっています。

9）Windows から仮想マシンのファイルを参照できるように設定する

仮想マシンのユーザｇｃｃのホームディレクトリを PC のネットワークドライブに割り当て仮想マシンは、samba によりｇｃｃのホームディレクトリを Windows と共有するよう設定されています。

ワークグループはｇｃｃとなっています。お使いのネットワーク環境に合うよう、/etc/samba/smb.conf を修正してください。仮想マシンを再起動すると設定が有効になります。



samba のユーザも gcc、パスワード vineLinux となっています。

PC のファイル共有時に「異なるユーザ名で接続します」を指定して、ユーザにｇｃｃ、パスワードに vineLinux を設定してください。

5 - 3 TeraTerm のインストール

1）TeraTerm をインストーラのガイドに従ってインストールする。

2）TeraTerm を起動する。

※　シリアルポートを指定する。

3）[設定]-[端末]で開くダイアログで下記のように設定する。

４）[設定]-[シリアルポート]で開くダイアログで下記のように設定する。

５-４ FDT のインストール

FDT 無償評価版のインストーラ（fdtv408r01.exe など）をダウンロードして実行します。インストーラの指示に従ってインストールしてください。

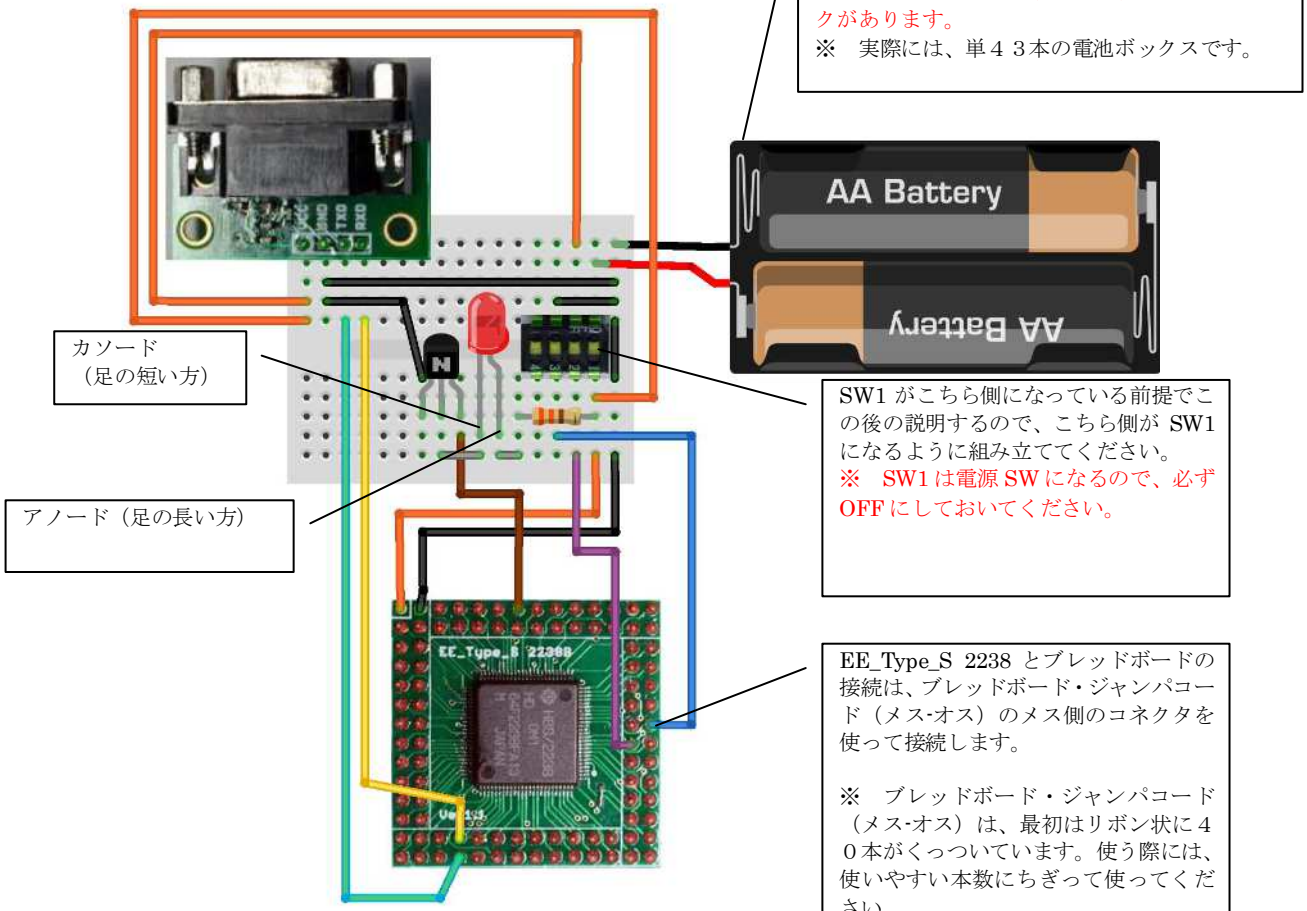
６　使ってみよう

まず、動作確認のために、LED を点滅させてみましょう。

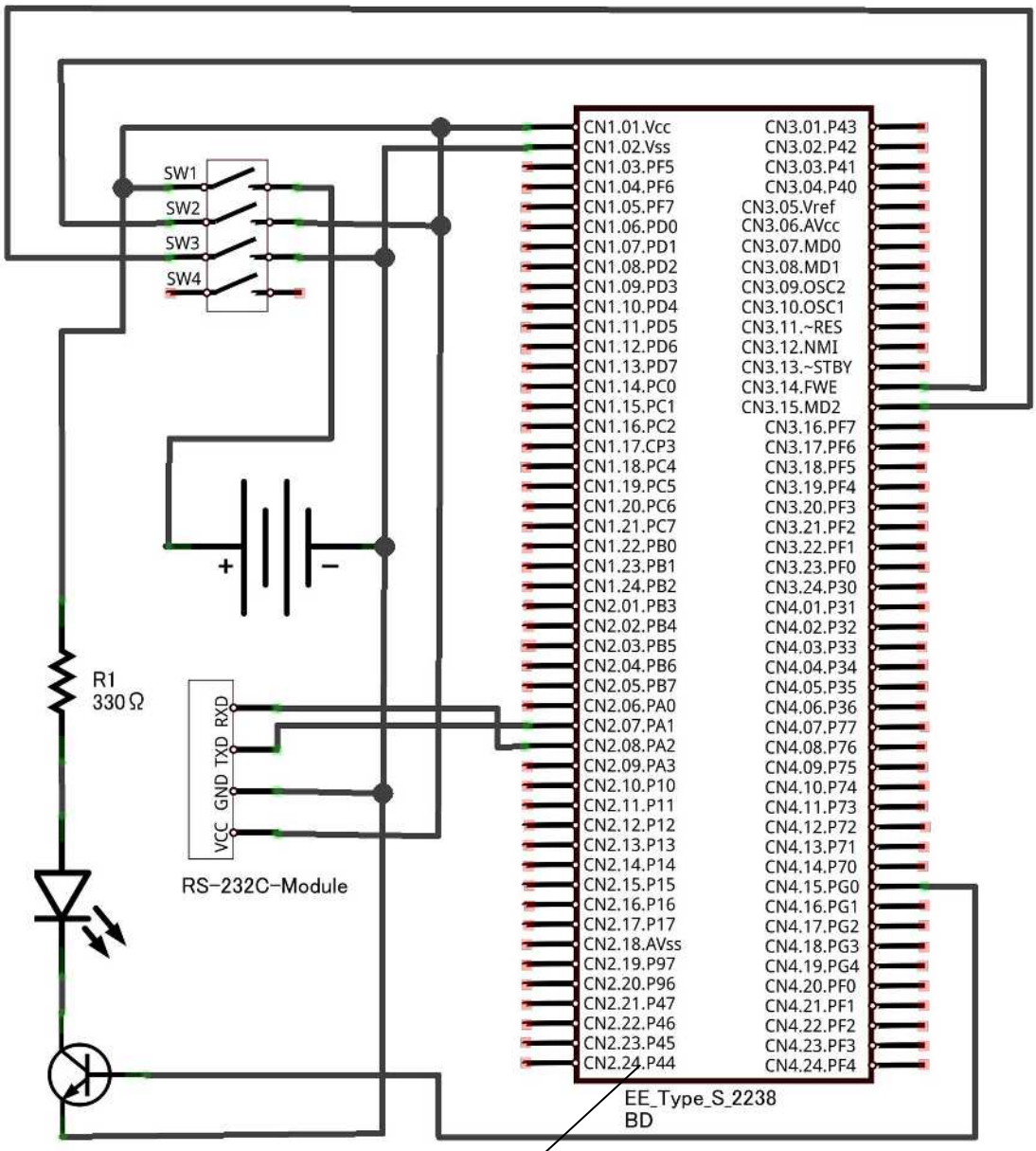
６-１　基本回路の組み立て

下記の回路を組み立ててください。

【実体配線図】

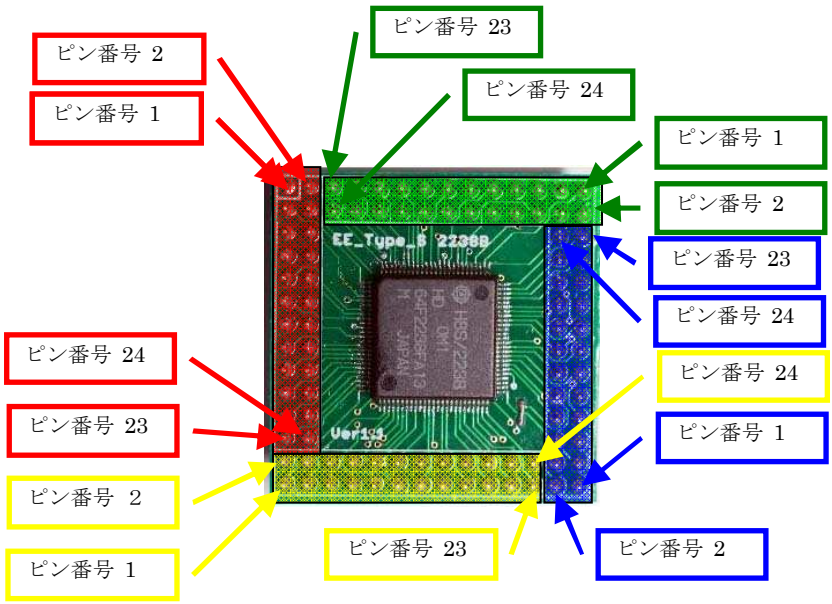


【回路図】



EE_Type_S 2238 の端子の名称は、「コネクタ No.ピン番号.信号名称」の構成になっています。
コネクタ No は１～４です。ピン番号は、コネクタ内の番号で１～２４です。
信号名は、H8S/2238 の端子の信号名を示しています。
たとえば、P14 であればポート１の４ビット目を指します。

【基盤のコネクタ番号とピン番号】



上図の赤・黄・青・緑がそれぞれ CN１・CN２・CN3・CN4 に対応します。
ピン番号は、基盤の外側が奇数、内側が偶数となるように上から下に付番しています。（ここで上から下という表現は、コネクタが左上にくるように基盤を回転させた状態でのルールである。例えば、上図では CN２の場合半時計周りに 90 度、CN３の場合反時計回りに 180 度、CN4 の場合反時計回りに 270 度回転させた状態での表現である）

６-２　プログラムの書き込み

DIP スイッチの SW1 が OFF になっていることを確認して、電池ボックスに電池を入れてください。

下記の手順でプログラムを書き込みます。

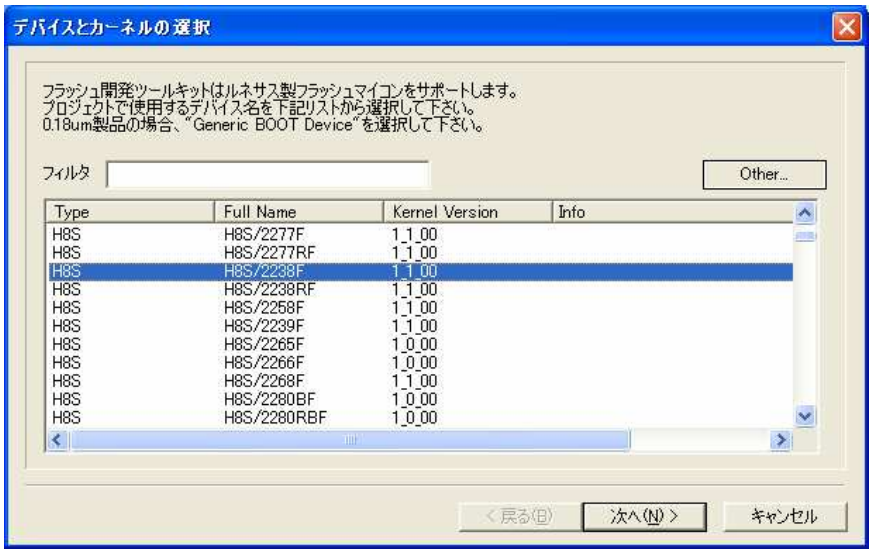
１）　RS-232C ケーブルで RS232C レベル変換モジュールとパソコンをつなぐ。

２）　DIP スイッチの SW2 と SW3 を ON にした状態で、SW1 を ON にする。

３）　Flash Development Toolkit 4.08 Basic を Windows のスタートメニューから起動する。

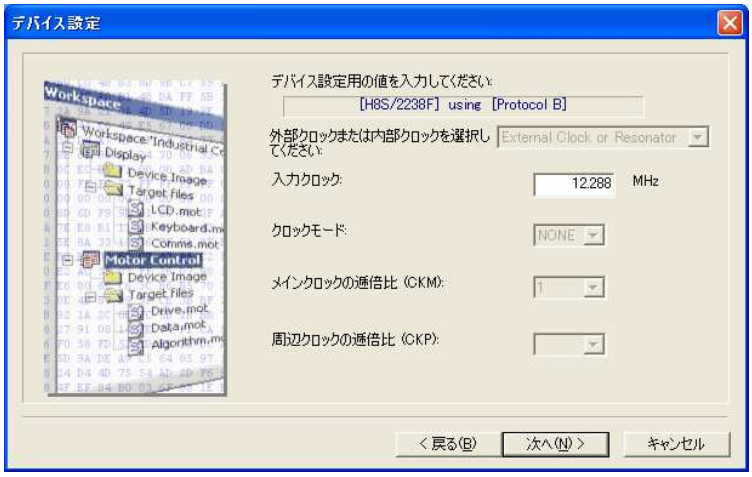
４）　各種パラメータを選択する。

①　H8S/2238F を選択する。

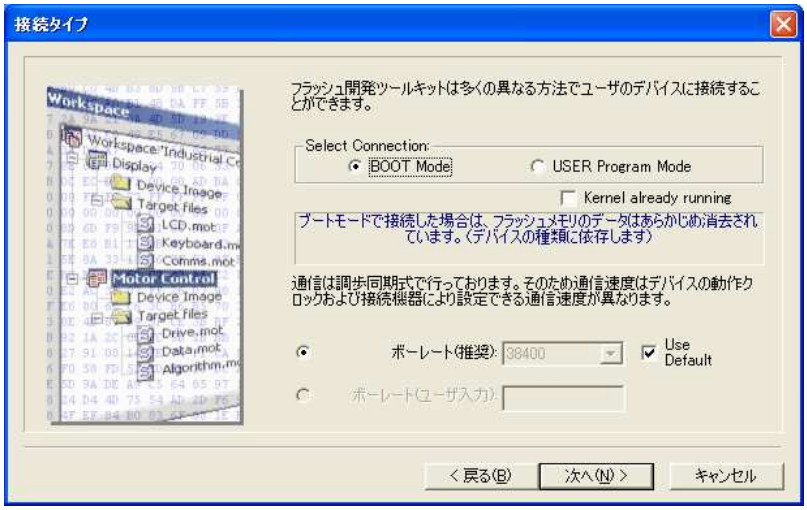


②　通信に使うシリアルポートを選択する。

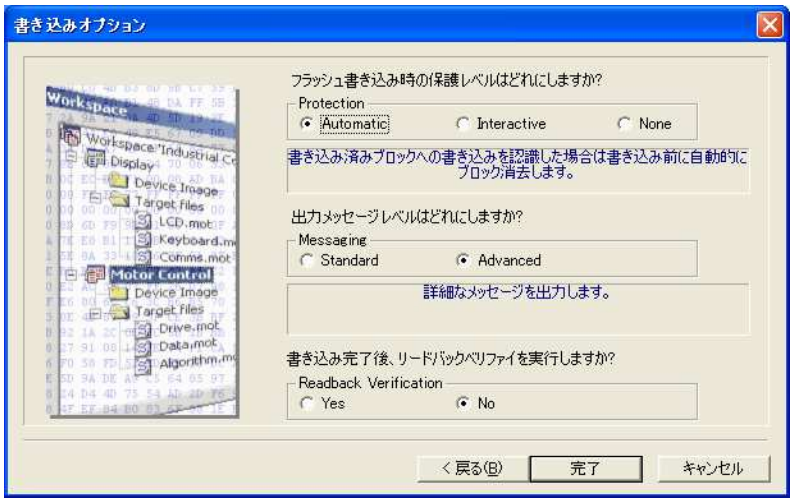
③CPU の入力クロックを選択する。



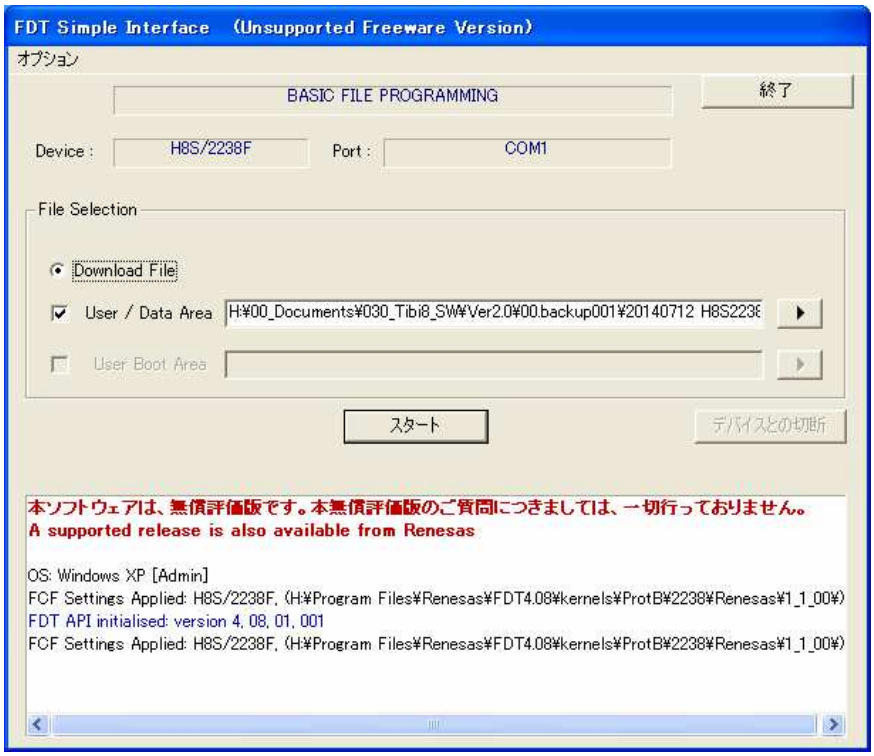
④接続タイプの設定（デフォルトのまま）



⑤書き込みオプションの設定（デフォルトのまま）



⑥　設定終了。次回の起動から以下の画面で起動します。



以降、設定したパラメータは保存されます。違う設定で書き込みたいときは、[オプション]-[新規設定]を使って、設定変更します。

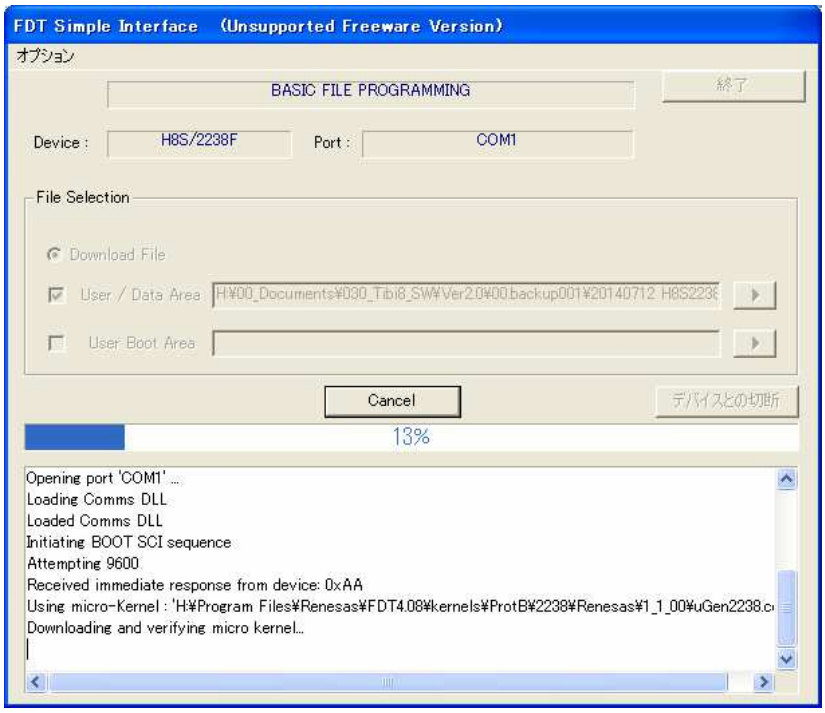
５）　書き込むファイルを選択する。

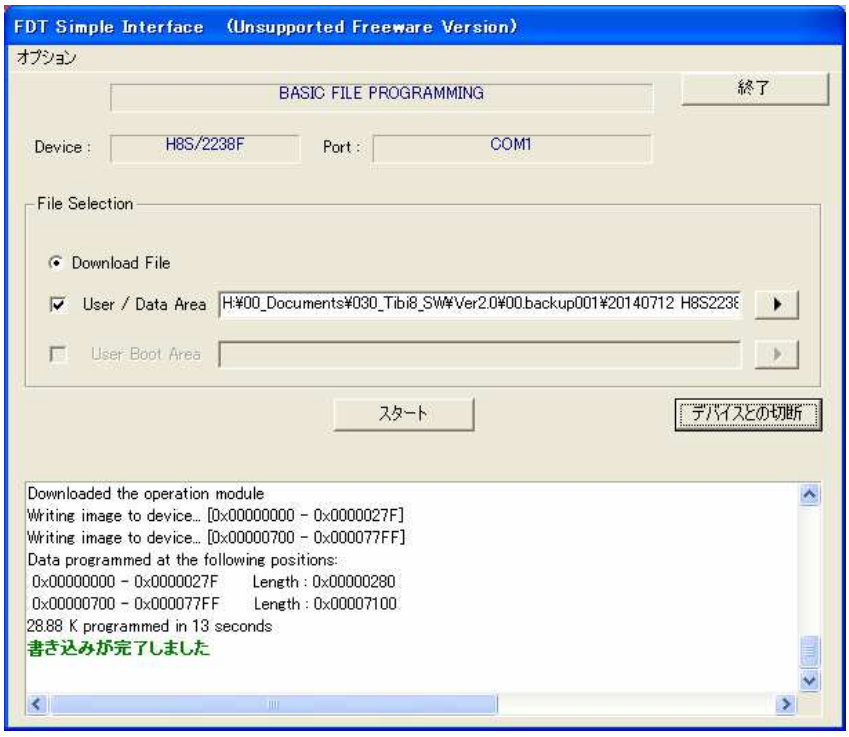
FDT の User/Data Area の右端のボタンを押して、書き込むファイルを選択します。

DVD のルートにある LED_TASK.mot を選択する。

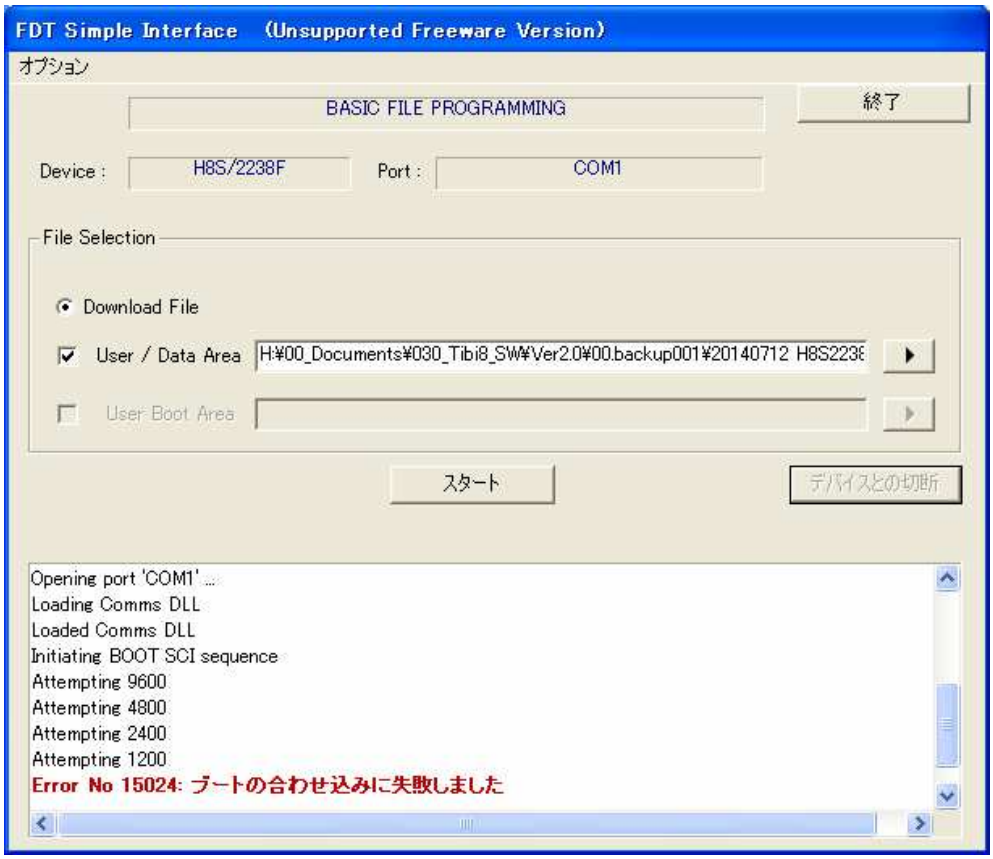
６）　スタートボタンを押し、書き込みを実行する。

下記のように、書き込みが完了しました。と表示されたら終了ボタンを押して FDT を終了する。


FDT Simple Interface (Unsupported Freeware Version)
オプション
BASIC FILE PROGRAMMING
終了
Device :
H8S/2238F
Port :
COM1
File Selection
Download File
User / Data Area
H:¥00_Documents¥030_Tibi8_SW¥Ver2.0¥00.backup001¥20140712 H8S223
User Boot Area
Cancel
デバイスとの切断
13%
Opening port 'COM1' ...
Loading Comms DLL
Loaded Comms DLL
Initiating BOOT SCI sequence
Attempting 9600
Received immediate response from device: 0xAA
Using micro-Kernel : 'H:¥Program Files¥Renesas¥FDT4.08¥kernels¥ProtB¥2238¥Renesas¥1_1_00¥uGen2238.c
Downloading and verifying micro kernel...


FDT Simple Interface (Unsupported Freeware Version)
オプション
BASIC FILE PROGRAMMING
終了
Device :
H8S/2238F
Port :
COM1
File Selection
Download File
User / Data Area
H:¥00_Documents¥030_Tibi8_SW¥Ver2.0¥00.backup001¥20140712 H8S223
User Boot Area
スタート
デバイスとの切断
Downloaded the operation module
Writing image to device... [0x00000000 - 0x0000027F]
Writing image to device... [0x00000700 - 0x000077FF]
Data programmed at the following positions:
0x00000000 - 0x0000027F Length : 0x00000280
0x00000700 - 0x000077FF Length : 0x00007100
28.88 K programmed in 13 seconds
書き込みが完了しました

※　以下のようなエラーとなる場合、PC とマイコンの通信ができていません。6-1 で組み立てた回路や PC との接続、電池切れなどを確認してください。



６-３　プログラムの実行

DIP スイッチの SW1 を OFF にし、マイコンの電源を切ります。

DIP スイッチの SW２，３を OFF にした状態で、DIP スイッチの SW1 を ON にします。

LED が１秒周期で点滅すれば、正常です。

プログラムが書き込めたのに、LED が点滅しない場合、LED をさす方向（長い端子と短い端子の方向）や DIP スイッチの SW２，３の接続を中心に 6-1 で組み立てた回路を確認してください。

7　H8S/2238 の周辺ハードウエアを理解しよう

この章では、電子工作でよく使う電子部品を、マイコンを使って制御する方法を紹介します。

7 - 1 CPU のハードウエアマニュアルをダウンロードする

ハードウエアマニュアルは、マイコンを使いこなすために最も重要なドキュメントの一つです。下記の手順で入手してください。

1）Renesas(http://japan.renesas.com/)のホームページで H8S/2238 を検索する。

ホームページの右上の検索ボタンの左の入力エリアに「HD64F2238」と入力し、検索ボタンを押す。

2）ハードウエアマニュアルをダウンロードする。

表示されたリストの一番上の行のドキュメント列をクリックする。

ユーザーハードウエアマニュアルを選択する。

H8S/2258 グループ、H8S/2239 グループ、H8S/2238 グループ、H8S/2237 グループ、H8S/2227 グループハードウェアマニュアルを選択する。

指示に従ってダウンロードする。

７-２IO ポートと LED の点滅

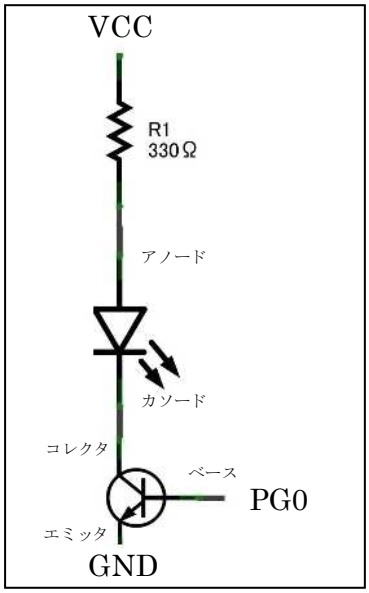
マイコンの Hello World ともいえる LED 点滅プログラムを作ってみましょう。

７-２-１ハードウエア

マイコンで LED などの電子機器を制御する場合、一般的に IO ポートの出力を用います。

ここでは、H8S/2238 マイコンボードの CN4 の１５番ピン（ポート G の０ビット目）に LED を接続して点滅させることにします。

回路は 6 章で作ったものをそのまま使いますので、この章では新たに結線する必要はありません。6 章で作った回路の内容を理解していきましょう。



６章で作った回路の LED に関連する部分は、右図のようになっています。PG0 が Hi になったときに、トランジスタのコレクタとエミッタ間が導通し、LED が点灯します。PG0 が Low になったときにトランジスタのコレクタとエミッタ間の抵抗が増加し、LED は消灯します。このトランジスタのスイッチング作用については、モーター制御の章で解説します。ここでは、トランジスタがスイッチの役目をはたして、PG0 の出力にしたがって LED のカソードが GND に接続したり、切断されたりすると理解してください。

今回使った LED（L-513LE1T）のデータシートを見ると、V_Fが 2.1V でした。このことから、LED 点灯時の LED のアノードとカソード間の電圧は 2.1V になることがわかります。また、I_Fの最大は 20mA と書かれていることから、今回は 10mA で LED を点灯させることとします。

当キットの電源電圧は、3.6V～５V 程度まで変動することが考えれれます。（新品のアルカリ電池を使った時には５V 程度になるでしょうし、ニッケル水素充電電池を使えば 3.6V を少し超える程度でしょう）ここでは電源電圧が５V として計算することにします。LED 点灯時に R1 にかかる電圧は 2.9V(=5-2.1)となります。R1 に 10mA が流れるように R1 の値を決めると、R1 の値は 290Ω(=2.9V/0.01A)となるので、入手性を考慮して R1 には 330Ω を使うことにしました。

次に CPU の IO ポートについて見ていきましょう。

ハードウエアマニュアルの 10.12 ポート G を一読してください。

下の表の丸で囲んだ状態に設定すれば、ポート G の０ビット目は出力ポートとして使用できることがわかります。つまり、IRQ6 を使わない（IER の IRQ6E を 0）かつ PG0DDR を１にすれば PG0 を出力ポートとして使えるということです。ハードウエアマニュアルの 5.3.2 を見ると IRQ6E は電源投入時には０なので、起動後に PG0DDR の設定を行えばよいことがわかります。

- PG0/$\overline{IRQ6}$

PG0DDR ビットにより次のように切り替わります。

PG0DDR	0	1
端子機能	PG0 入力端子	PG0 出力端子
	$\overline{IRQ6}$ 入力端子*	

【注】 * 外部割り込み端子として使用する場合は、他の機能として使用しないでください。

ハードウエアマニュアルの表 27.28 を見るとこの CPU では直接 LED を接続することは無理だということがわかります。LED の点灯には、10mA 程度の電流が必要ですが、IO ポートの Hi/Low 出力許容電流がともに 1 mA となっており（下表の赤丸の部分）10mA の電流を流すことができません。このため、6 章の回路では IO ポートに直接 LED を接続せずに、トランジスタを介して接続するようにしたのです。

表 27.28 出力許容電流

条件 A（F-ZTAT 版）：V_{CC}=3.0～5.5V、AV_{CC}=3.6～5.5V、V_{ref}=3.6V～AV_{CC}、V_{SS}=AV_{SS}=0V、T_a=-20～+75℃（通常仕様品）、T_a=-40～+85℃（広温度範囲仕様品）*1

条件 B（マスク ROM 版）：V_{CC}=2.7～5.5V、AV_{CC}=3.6～5.5V、V_{ref}=3.6V～AV_{CC}、V_{SS}=AV_{SS}=0V、T_a=-20～+75℃（通常仕様品）、T_a=-40～+85℃（広温度範囲仕様品）*1

項 目		記号	min	typ	max	単位
出力（1 端子当たり）Low レベル許容電流	SCL1、SCL0、SDA1、SDA0	I_{OL}	—	—	10	mA
	上記以外の出力端子		—	—	1.0	
出力 Low レベル許容電流（総和）	全出力端子の総和	ΣI_{OL}	—	—	60	mA
出力 High レベル許容電流（1 端子当たり）	全出力端子	$-I_{OH}$	—	—	1.0	mA
出力 High レベル許容電流（総和）	全出力端子の総和	$\Sigma -I_{OH}$	—	—	30	mA

【注】 LSI の信頼性を確保するため、出力電流値は表 27.28 の値を超えないようにしてください。

7-2-2 ソフトウエア

ハードウエアは完成したので、次はソフトウエアです。

この章では EE_LIB というハードウエアを隠蔽したライブラリを活用して、H8S/2238 の周辺回路の概念を理解していきたいと思います。

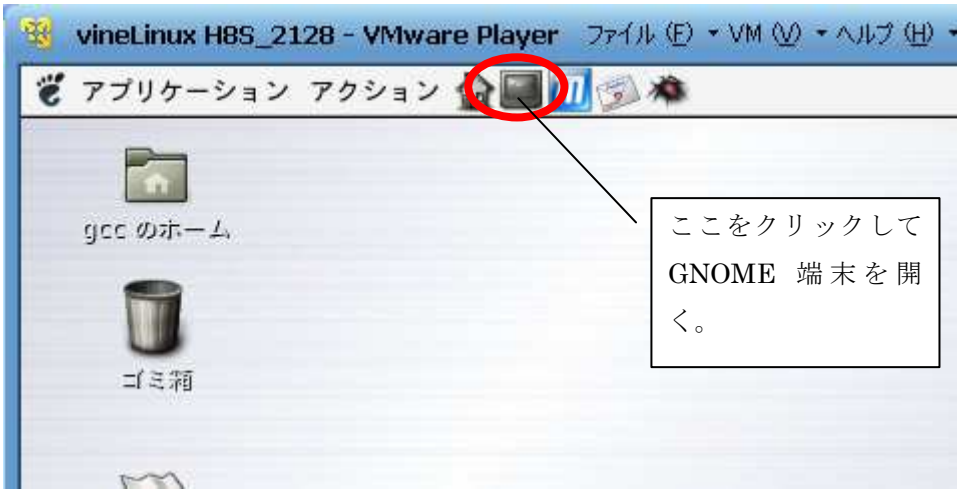
7-2-2-1 動かしてみる

まず、動かしてみましょう。

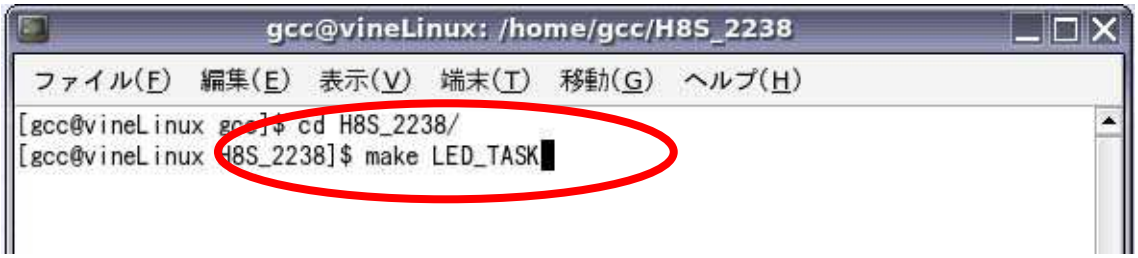
1）コンパイルする

仮想マシンにユーザー gcc でログインする。

GNOME 端末を開く。

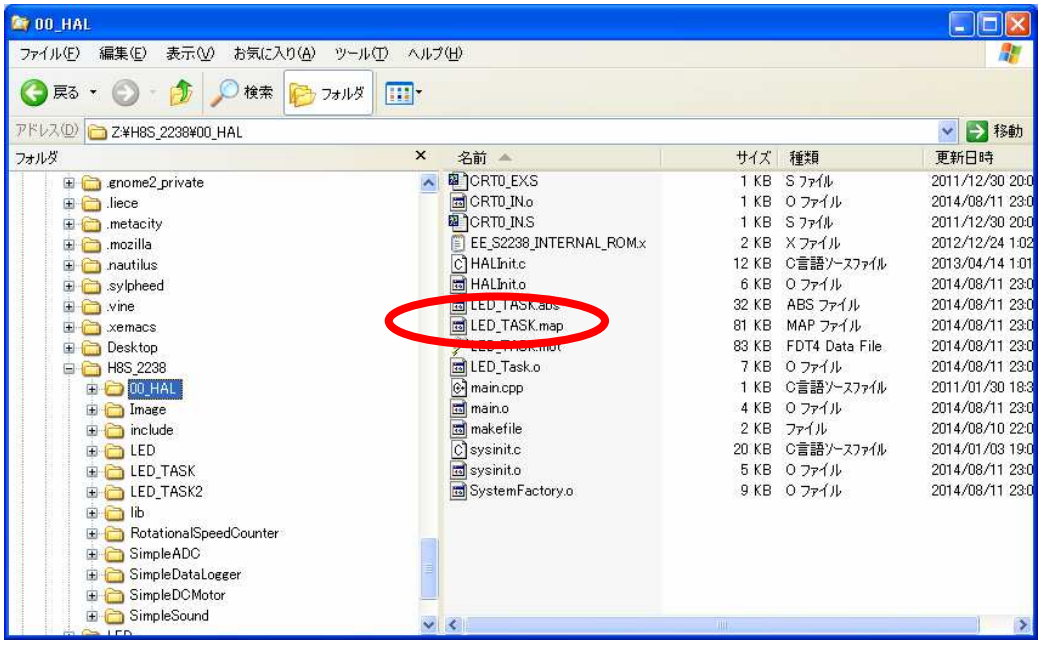


H8S_2238 にディレクトリを移動し、LED_TASK を make する。



エクスプローラで仮想マシンの H8S_2238/00_HAL フォルダの中身を確認する。

※　LED_TASK.mot ができていれば成功です。うまくできない場合、make clean を実行した後に、再度 make LED_TASK を実行してみてください。（うまく実行モジュールが生成できないときは、make clean を実行してみましょう）



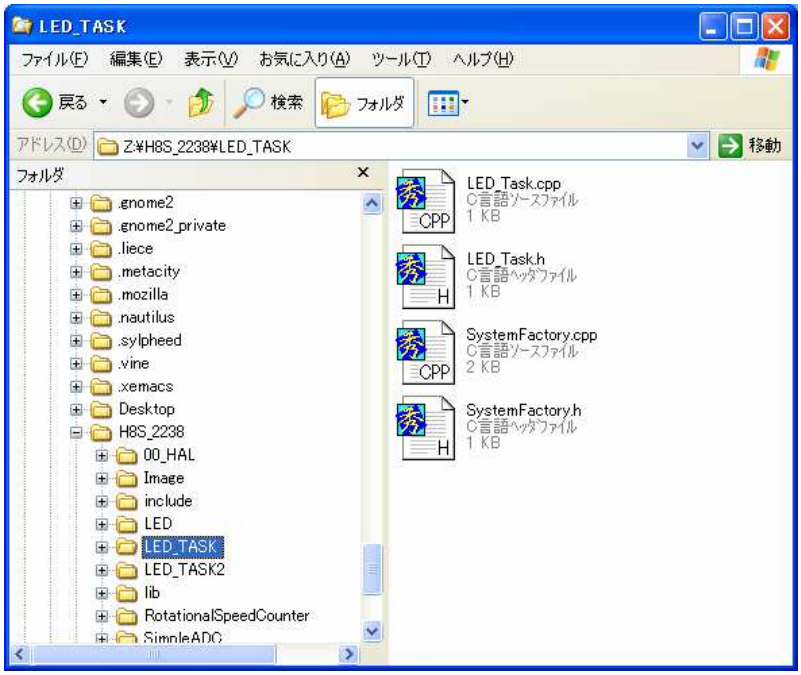
２）プログラムを書込み、実行する。

６章の要領で LED_TASK.mot を書き込み、実行する。

※　LED が１秒周期で点滅します。

７-２-２-２プログラムを理解しよう

仮想マシンの H8S_2238/LED_TASK の下に LED_TASK.mot 用のソースプログラムがあります。



SystemFactory と LED_Task ２つのクラスから構成されています。

１）SystemFactory クラス

このクラスは、startup メソッド以外実装はありません。メンバー変数もありません。

当キットのアプリケーションは、すべて SystemFactory::startup メソッドからプログラムを開始するルールにしています。

Startup メソッドは、下記のようになっています。

```
22 : void SystemFactory::startup()
23 : {
24 :         EE_LIB::HAL::IO::IOPin pin(EE_LIB::HAL::IO::IOPORTG,EE_LIB::HAL::IO::IOPIN0,EE_LIB::HAL::IO::IOPin::DIR_OUT);
25 :
26 :         EE_LIB::OSWrapper::Task::OSTimer osTimer;
27 :         EE_LIB::HAL::IntervalTimer::TimerWD
timer(EE_LIB::HAL::IntervalTimer::TimerWD::Timer0,1000000UL,EE_LIB::HAL::Interrupt::PRIORITY_LOW,&osTimer);          // 1ms周期割り込
み
28 :
29 :         LED_Task ledTaskHandler(&pin);
30 :         EE_LIB::OSWrapper::Task::TaskFactory::createTask(&ledTaskHandler,1000U,EE_LIB::OSWrapper::Task::Task::TP_Middle,&osTimer); //
1秒周期のタスクの生成
31 :
32 :         while(-1) {
33 :         }
34 : }
```

【解説】

このメソッドの機能は、以下の２つです。

・　ポート G のピン番号０を出力ポートに設定する。

・ LED_Task クラスの Perform メソッド１秒周期に呼び出す割り込みを発生させるよう、H8S/2238 の周辺機能を初期化する。

この機能を実現するために、以下のステップを実施しています。

24 行目は、IOPin クラスを生成しています。IOPin クラスは IO ポートの各ピンをプログラム制御したり、その状態読み取ったりするときに使うクラスです。この行では、ポート G(IOPORTG)のピン番号０（IOPIN0）を出力ポート（DIR_OUT）に設定しています。

26 行目は、OSTimer クラスを生成しています。OSTimer クラスは、後述タスクのスケジュールをするためのタイマーです。タスクを使うときは作るものと理解してください。

27 行目は、TimerWD クラスを生成しています。TimerWD クラスは、ウォッチドックタイマーを抽象化したクラスです。ウォッチドックタイマは H8S/2228 の周辺機能の一つで、ハードウエアマニュアルの 14 章に説明されています。ここでは、ウォッチドックタイマー０（TimerWD::Timer0）を使って、1ms（1000000 ns）周期に OSTimer クラスを呼び出すようにウォッチドックタイマを初期化しています。

29 行目は、このアプリケーションのもう一つのクラス LED_Task クラスを生成しています。

30 行目は、LED_Task クラスの Perform メソッドを１秒（1000 ms）周期で呼び出すよう設定しています。

32 行目からは無限ループすることによって、上記で生成したクラスが開放されないようにしています。

※　説明していることがソースコード上のどこでの表現されているかを示すため、ソースコードのパラメータを解説の（）の中に記入しました。

※　ここまでの説明に出てきた各クラスの詳細については、９章の EE_LIB リファレンスマニュアルを参照してください。

２）LED_Task クラス

このクラスの実装は、下記のようになっています。

【LED_TASK.h】

```
 8 : #if !defined(LED_TASK__INCLUDED_)
 9 : #define LED_TASK__INCLUDED_
10 :
11 : #include "EE_LIB/OSWrapper/Task/CyclicTaskPerformer.h"
12 : #include   "EE_LIB/HAL/IO/IOPin.h"
13 :
14 : namespace APPLI
15 : {
16 :                         class LED_Task: public EE_LIB::OSWrapper::Task::CyclicTaskPerformer
17 :                         {
18 :
19 :                         public:
20 :                                 LED_Task(EE_LIB::HAL::IO::IOPin* pin);
21 :                                 virtual ~LED_Task();
22 :
23 :                                 void perform(unsigned short interval);
24 :
25 :                         private:
26 :                                 unsigned char m_state;
27 :                                 EE_LIB::HAL::IO::IOPin* m_pin;
28 :                         };
29 :
30 : }
31 : #endif
```

【LED_TASK.cpp】

```
 8 : #include "LED_Task.h"
 9 :
10 : using APPLI::LED_Task;
11 :
```

```
12 : LED_Task::LED_Task(EE_LIB::HAL::IO::IOPin* pin):m_state(0),m_pin(pin){}
13 : LED_Task::~LED_Task(){}
14 :
15 : void LED_Task::perform(unsigned short interval)
16 : {
17 :             m_pin->set(m_state);
18 :             m_state ^=1;
19 : }
```

【解説】

このクラスの機能は、コンストラクタで指定された IOPin の出力を perform メソッドが呼び出されるたびに Hi/Low に切り替えることです。

○LED_TASK.h

8,9,31 行目は、このヘッダファイル複数回インクルードしたとしても 1 回しかコードに展開しないようプリプロセッサに処理させるための記述です。C++では常套手段です。

16 行目は、このクラスが CyclicTaskPerformer を継承したクラスであることを宣言しています。当キットでは周期的に実行する処理は、CyclicTaskPerformer の子クラスの perform メソッドに実装することにしています。

SystemFactory で perform メソッドの呼び出し間隔を定義し、その処理内容を CyclicTaskPerformer の子クラスの perform メソッドに書くわけです。

今回は 1 秒の周期処理だけですが、複数の実行タイミングの異なる周期処理を実装する場合に実行タイミングと実施する処理を分離して実装できることは、プログラムを書く上では大変便利です。

○LED_TASK.cpp

17 行目は、IOPin クラス（m_pin）の set メソッドを使って IO ポートの出力を m_state の値に変更しています。m_state の値が 0 の場合 Low レベル、0 でない場合 Hi レベルを出力します。

18 行目は、m_state が 0 の場合 m_state を 1 に、m_state が 1 の場合 m_state を 0 に設定しています。

7 - 3 AD コンバータ

センサーの出力をマイコンに取り込むときによく使用するのが AD コンバータです。電圧の変化を出力とするセンサーの出力を読み取る場合などに AD コンバータを使います。

今回はボリュームのつまみの角度を AD コンバータで読み取ってみたいと思います。

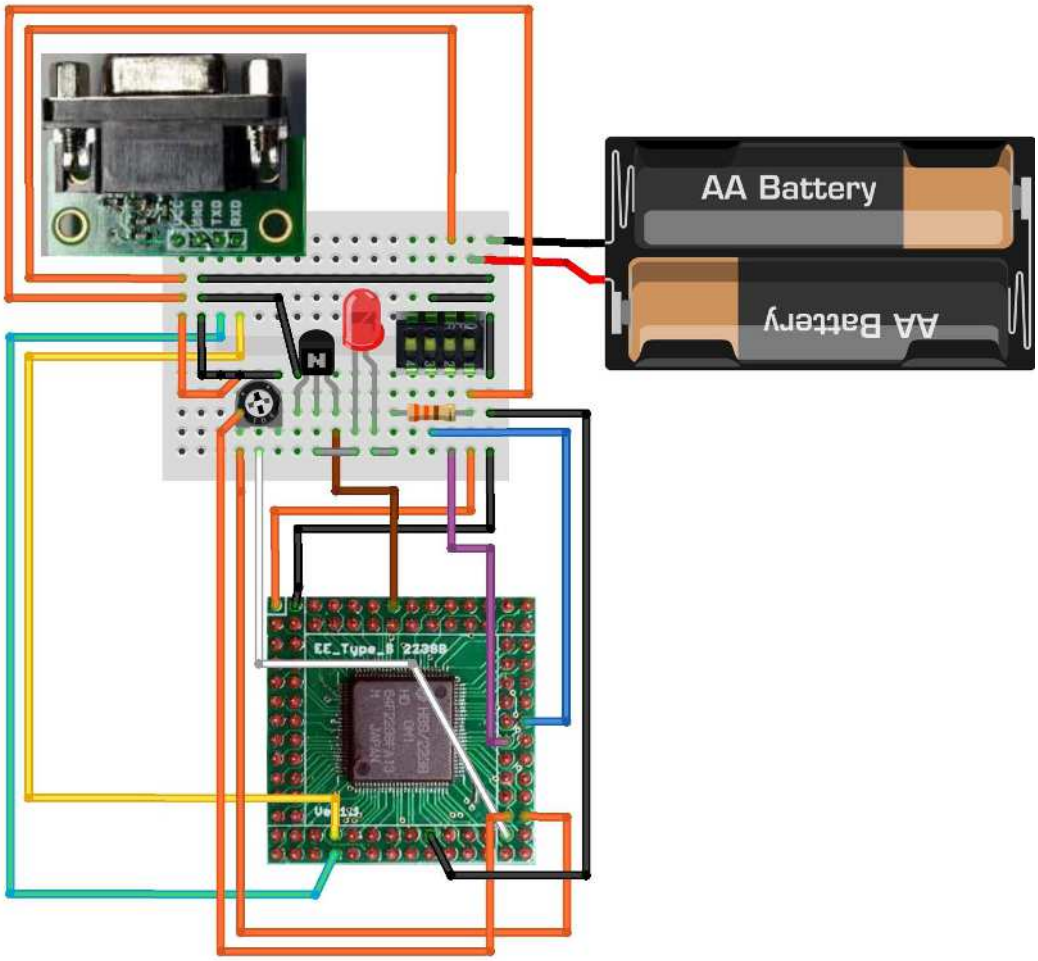
7 - 3 - 1 ハードウエア

H8S/2228 の AD コンバータは、10 ビットの分解能を持っているので、入力チャネルの電位が AVcc と同じとき 1023、GND と同じとき 0 に変換します。（詳細はハードウエアマニュアルの「17. A/D 変換器」を参照ください。）

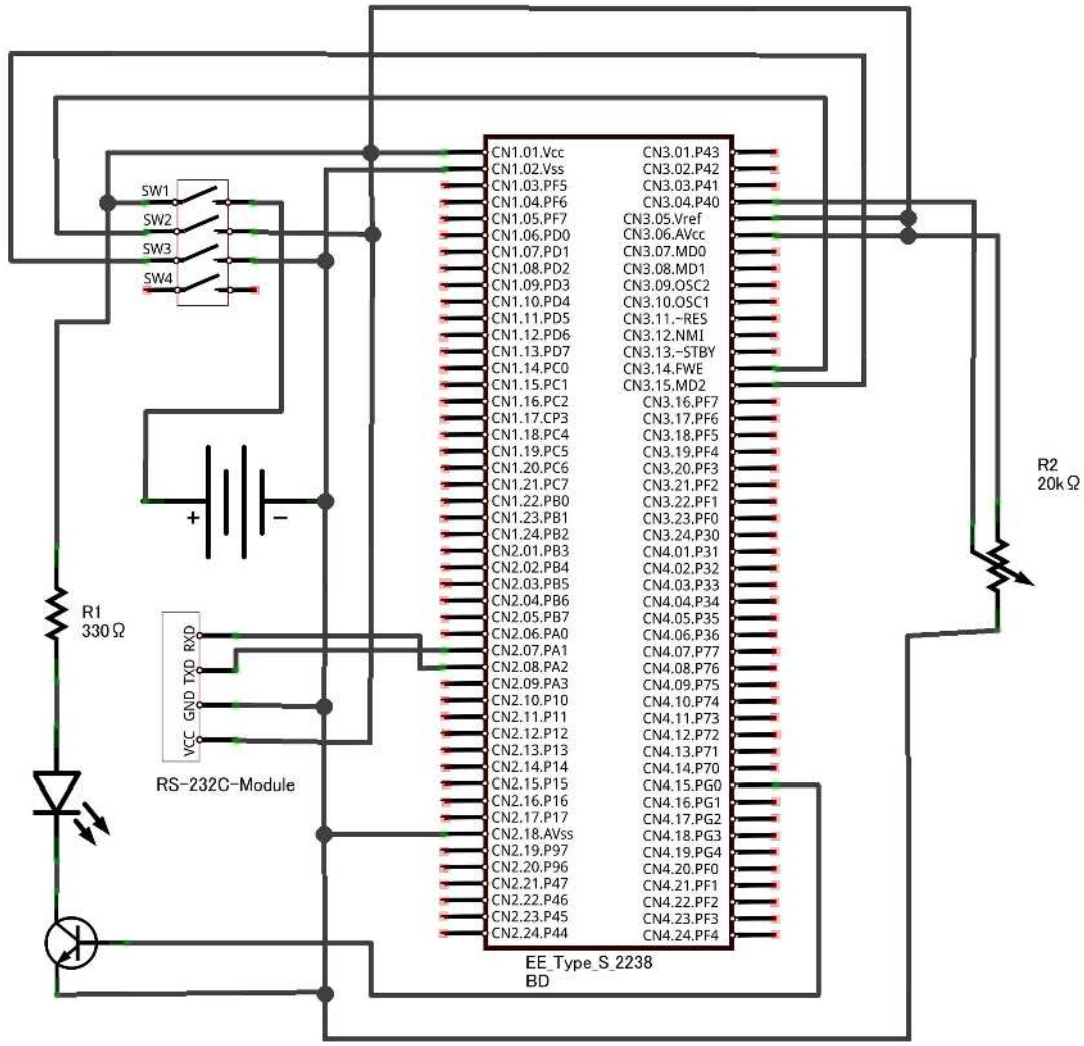
入力チャネルに印加できる電圧は最大で AVcc までで、AVcc には 7V まで印加できます。（ハードウエアマニュアル　表 27.26 を参照）

今回測定するのはボリュームのつまみの角度なので、下記のような回路になります。今回使用するボリュームは、3 つの端子を持つ最もポピュラーなものです。ボリュームの 3 つの端子のうち、

両端の２つの端子に電圧をかけると、真ん中の端子にはつまみの角度に応じて 0 から両端にかけた電圧までの電圧を出力します。
下記の回路図では、ボリュームの両端に 3.6V の電圧をかけているので、P40（AN0）を接続した端子の電位は、つまみの角度に応じて０～3.6V に変化します。
【実体配線図】



【回路図】



７-３-２ソフトウエア

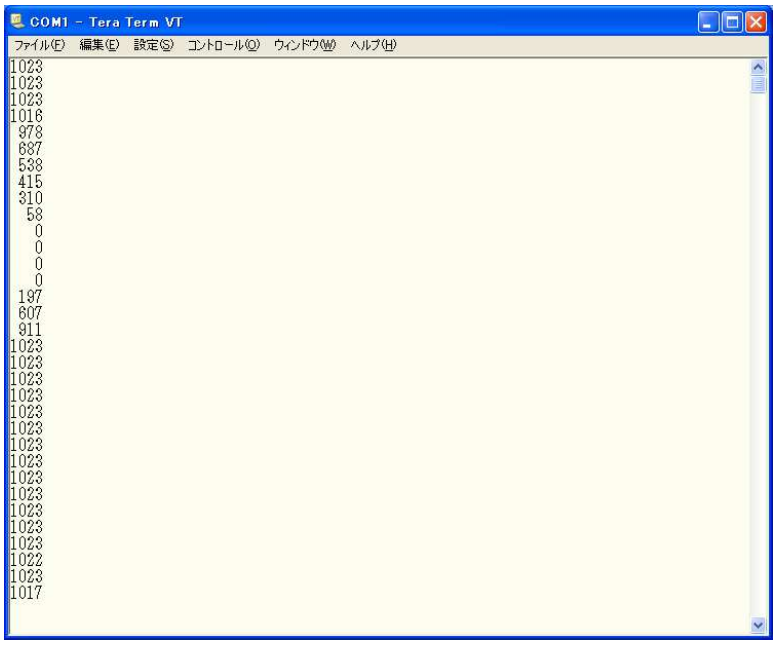
まず、動かしてみましょう。

H8S_2238 にディレクトリを移動し、SimpleADC を make する。

仮想マシンの H8S_2238/00_HAL の下に SimpleADC.mot ができるので、CPU に書き込んでください。

SimpleADC.mot は、ADC で読み取った値を RS-232C 経由で PC に送り、PC で表示するよう

プログラムされています。
H8S/2238 マイコンボードと PC を RS-232C ケーブルで接続し、PC 上で TeraTerm を起動しておいて、H8S/2238 マイコンボードの電源をいれると、TeraTerm 上に値が表示されます。
ボリュームのつまみの角度を変えると、TeraTerm 上の値が変化します。



７-３-２-１プログラムを理解しよう
仮想マシンの H8S_2238/ SimpleADC の下に SimpleADC.mot 用のソースプログラムがあります。
SystemFactory と ADC２つのクラスから構成されています。

３）SystemFactory クラス

```
 8 : #include "SystemFactory.h"
 9 : #include "ADC.h"
10 : #include <stdio.h>
11 : #include   "EE_LIB/HAL/IO/IOPin.h"
12 : #include   "EE_LIB/HAL/IntervalTimer/TimerWD.h"
13 : #include   "EE_LIB/HAL/Interrupt/Priority.h"
14 : #include   "EE_LIB/HAL/Communication/SCIUsingInterrupt.h"
15 : #include "EE_LIB/OSWrapper/Task/OSTimer.h"
16 : #include "EE_LIB/OSWrapper/Task/TaskFactory.h"
17 : #include "EE_LIB/Util/Primitive/Format.h"
18 : #include "EE_LIB/Unit/Communication/Com.h"
19 :
20 : using APPLI::SystemFactory;
21 :
22 : SystemFactory::SystemFactory(){}
23 : SystemFactory::~SystemFactory(){}
24 :
25 : void SystemFactory::wait()
26 : {
27 :             volatile int a,b,c,ii,jj;
28 :             for(a=10,b=20,jj=0; jj<7; jj++) {for(ii=0;ii<5000; ii++) {c=c*a+b;}}
29 : }
30 :
31 : void SystemFactory::startup()
32 : {
33 :             EE_LIB::HAL::IO::IOPin pin(EE_LIB::HAL::IO::IOPORTG,EE_LIB::HAL::IO::IOPIN0,EE_LIB::HAL::IO::IOPin::DIR_OUT);
34 :
```

```
 35 :        EE_LIB::OSWrapper::Task::OSTimer osTimer;
 36 :        EE_LIB::HAL::IntervalTimer::TimerWD
timer(EE_LIB::HAL::IntervalTimer::TimerWD::Timer0,500000UL,EE_LIB::HAL::Interrupt::PRIORITY_LOW,&osTimer); // 500uS 周期
 37 :        EE_LIB::HAL::Communication::SCI *sci;
 38 :
 39 :        //============ SCI の生成 ============
 40 :        {
 41 :                EE_LIB::HAL::Communication::SCI::CommunicationParam cp;
 42 :                EE_LIB::HAL::Communication::SCI::SCIId id=EE_LIB::HAL::Communication::SCI::SCI2;
 43 :                cp.cm=EE_LIB::HAL::Communication::SCI::SYNCHRONUS_TOTNOAL;
 44 :                cp.bps=EE_LIB::HAL::Communication::SCI::BPS_38400;
 45 :                cp.dl=EE_LIB::HAL::Communication::SCI::DL_8BIT;
 46 :                cp.sb=EE_LIB::HAL::Communication::SCI::SB1;
 47 :                cp.parity=EE_LIB::HAL::Communication::SCI::P_NONE;
 48 :                cp.mp=EE_LIB::HAL::Communication::SCI::MP_DISABLE;
 49 :                sci=new EE_LIB::HAL::Communication::SCIUsingInterrupt(id,cp);
 50 :        }
 51 :        // ========== 通信ポートの生成 ==============
 52 :        EE_LIB::Unit::Communication::Com com(sci,&osTimer);
 53 :        // ========== ADC タスクの生成  ==============
 54 :        EE_LIB::HAL::IO::ADC adc;
 55 :        ADC adcTask(&adc);
 56 :        EE_LIB::OSWrapper::Task::TaskFactory::createTask(&adcTask,10,EE_LIB::OSWrapper::Task::Task::TP_Middle,&osTimer); //  10ms 周期
のタスク
 57 :
 58 :        while(-1)
 59 :        {
 60 :                pin.set(0);
 61 :                wait();
 62 :                {
 63 :                        EE_LIB::Util::Primitive::Format format("%-4d",adc.get(0));
 64 :                        com.send(&format,true);
 65 :                }
 66 :                pin.set(1);
 67 :                wait();
 68 :        }
 69 : }
```

【解説】

このクラスの機能は、以下の４つです。

- 通信ポートを抽象化した Com クラスを使って通信できるようにする。
- ADC クラスの Perform メソッド 10ms 周期に呼び出す割り込みを発生させるよう、H8S/2238 の周辺機能を初期化する。
- EE_LIB::HAL::IO::ADC クラスのチャネル０の値を周期的に通信ポートに出力する。
- 本体の LED を周期的に点滅させる。

この機能を実現するために、以下のステップを実施しています。

8～18 行では、このクラスで使用する各クラスの定義をインクルードしています。EE_LIB のクラスを使うために必要なインクルードファイルについては、９章の EE_LIB リファレンスマニュアルを参照ください。

25～29 行は、wait という名前の通り、時間を稼ぐためのメソッドです。startup メソッドの最後の LED を点滅させる処理で、LED の点灯・消灯時間を作るために使います。

31 行以降は、startup メソッドで前述のとおり、電源投入後に最初に呼び出されるメソッドです。

33,35,36 行目では、7-2-2-2 と同様にそれぞれ IOPin・OSTimer・TimerWD クラスのインスタンスの生成を行っています。

41～48 行目では、RS-232C の通信方法（38400BPS、８ビット、ストップビット１、パリティなし）を構造体 cp に設定しています。

49 行目では、構造体 cp で指定した通信方法に従って SCI0 を使って通信する

SCIUsingInterrupt クラスを生成しています。SCIUsingInterrupt クラスは１バイトの送信と受信をサポートします。

52 行目では、SCIUsingInterrupt クラスを使って文字列など複数バイトの送受信を行う Com クラスを生成しています。SCIUsingInterrupt クラスは、送受信の単位が１バイト単位なので直接使うには不便なので、Com クラスが不足している機能を補います。

54～56 行目では、ADC クラスのコンストラクタに EE_LIB::HAL::IO::ADC の参照を渡して ADC クラスを生成し、ADC クラスの perform メソッドを１０ｍｓ周期で起動するために周期起動タスクとして登録しています。ここで、同じ名前の異なるクラス ADC が登場しました。ネームスペース「EE_LIB::HAL::IO::」がついている方は EE_LIB が提供する ADC クラスです。ネームスペースを指定しない ADC クラスは後述の今回のプログラム専用のクラスです。

58～67 行目では、EE_LIB::HAL::IO::ACD クラスから取得した AD コンバータのチャネル AN0 の値を COM クラスをつかってシリアルポートに周期的に出力しています。Format クラスは数値を指定したフォーマットの文字列に変換するクラスで、ここでは 4 桁右詰の 10 進数に変換しています。

４）ADC クラス

このクラスの実装は、下記のようになっています。

【ADC.h】

```
 8 : #if !defined(ADC__INCLUDED_)
 9 : #define ADC__INCLUDED_
10 :
11 : #include "EE_LIB/OSWrapper/Task/CyclicTaskPerformer.h"
12 : #include "EE_LIB/HAL/IO/ADC.h"
13 :
14 : namespace APPLI
15 : {
21 :                         class ADC: public EE_LIB::OSWrapper::Task::CyclicTaskPerformer
22 :                         {
23 :
24 :                         public:
25 :                                 explicit ADC(EE_LIB::HAL::IO::ADC* pADC);
26 :                                 virtual ~ADC();
27 :
28 :                                 void perform(unsigned short interval);
29 :
30 :                         private:
31 :                                 EE_LIB::HAL::IO::ADC* m_pADC;
32 :                         };
33 :
34 : }
35 : #endif // !defined(ADC__INCLUDED_)
```

【ADC.cpp】

```
 8 : #include "ADC.h"
 9 :
10 : using APPLI::ADC;
11 :
12 : ADC::ADC(EE_LIB::HAL::IO::ADC* pADC):m_pADC(pADC){}
13 : ADC::~ADC(){}
14 :
15 : void ADC::perform(unsigned short interval)
16 : {
17 :         m_pADC->start(EE_LIB::HAL::IO::ADC::M_SINGLE,0);     // AN0 を 1 回 AD 変換する
18 : }
```

【解説】

このクラスの機能は、perform メソッドが呼び出されるたびに AN0 の AD 変換を実施することです。

○ADC.h

16 行目は、このクラスが CyclicTaskPerformer を継承したクラスであることを宣言しています。CyclicTaskPerformer の用法は LED_TASK と同じです。

○ADC.cpp

17 行目では、EE_LIB::HAL::IO::ADC クラスの start メソッドを呼び出して、AN0 の AD 変換を開始しています。

最後に実施した AD 変換の結果は、EE_LIB::HAL::IO::ADC クラスの get メソッドで取り出すことができます。（SystemFactory で取得しています。）

７-４ PWM 出力

DC モーターの回転数を制御してみたいと思います。今回は DC モーターの回転数を制御しますが、LED やランプの明るさなど電圧によって変化するものを制御する場合には同じ手法が使えます。電子工作にいろいろと応用できる知識ですので、しっかり理解しましょう。

７-４-１ ハードウエア

CPU でモーターの回転を制御するわけですが、IO ポートには前述のとおり 1mA 程度の電流しか流せないので、今回もトランジスタを使います。まず、下の回路を見てください。P10 は 3.2kΩの抵抗を介してトランジスタのベースに接続しています。モータは 3.6V 電源とトランジスタのコレクタに接続しています。トランジスタのエミッタは GND に接続しています。これはトランジスタのスイッチング回路です。

今回使用する NPN 型トランジスタの場合、ベースとエミッタ間に電圧をかけることによってベース-エミッタ間に電流を流すと、ベース-エミッタ間に流れる電流を増幅した電流がコレクタ-エミッタ間に流れるというトランジスタの性質を利用した回路です。

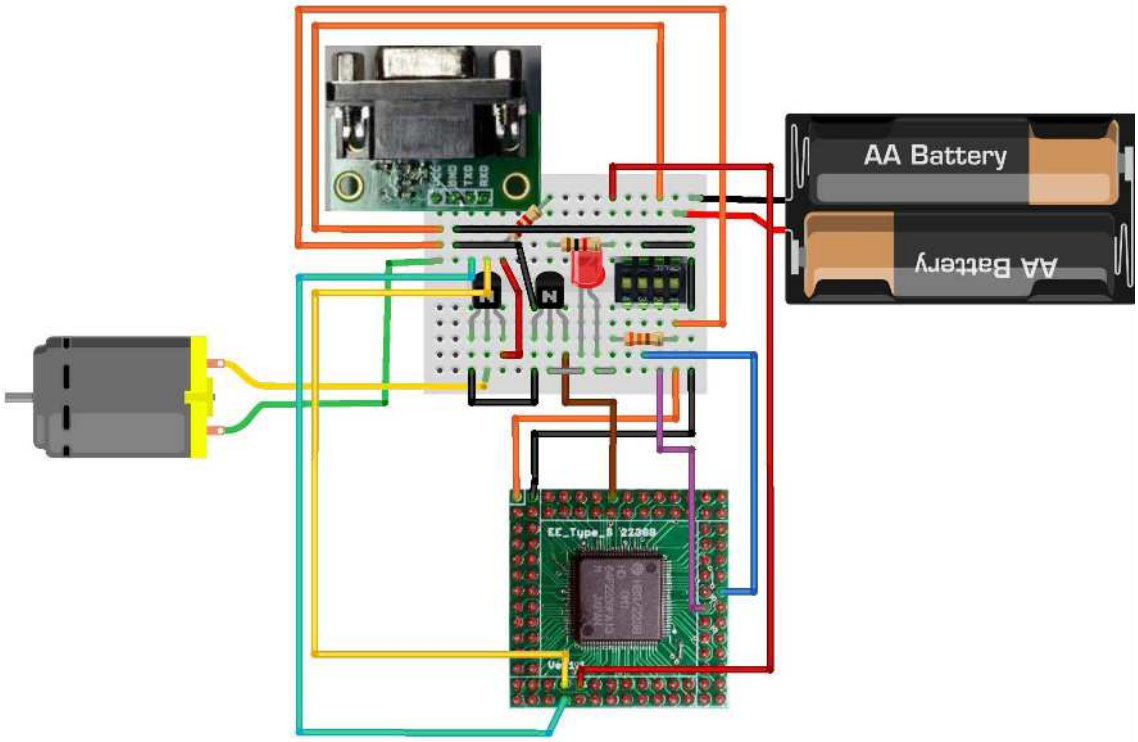
今回は、2SC1815 の Y ランクを使用しました。増幅率 hfe：120～240 です。今回の回路では、ベースに抵抗 3.2kΩを接続しているので、ベースに流れる電流は、0.94mA=(3.6-0.6)/3200×1000 です。ここで 3.6V は P10 の電圧、0.6V は 2SC1815 のベース-エミッタ間に電流が流れるときに生じるベース-エミッタ間の電圧です。倍率を 120 倍と仮定すると 113mA となり、小型のモータであれば回せる電流をコレクタ-エミッタ間に流すことができる回路だということが分かります。

7-2 章と同じように P10 を制御するプログラムを組めば、モータの ON/OFF することができますが、モーターの回転数を制御することはできません。

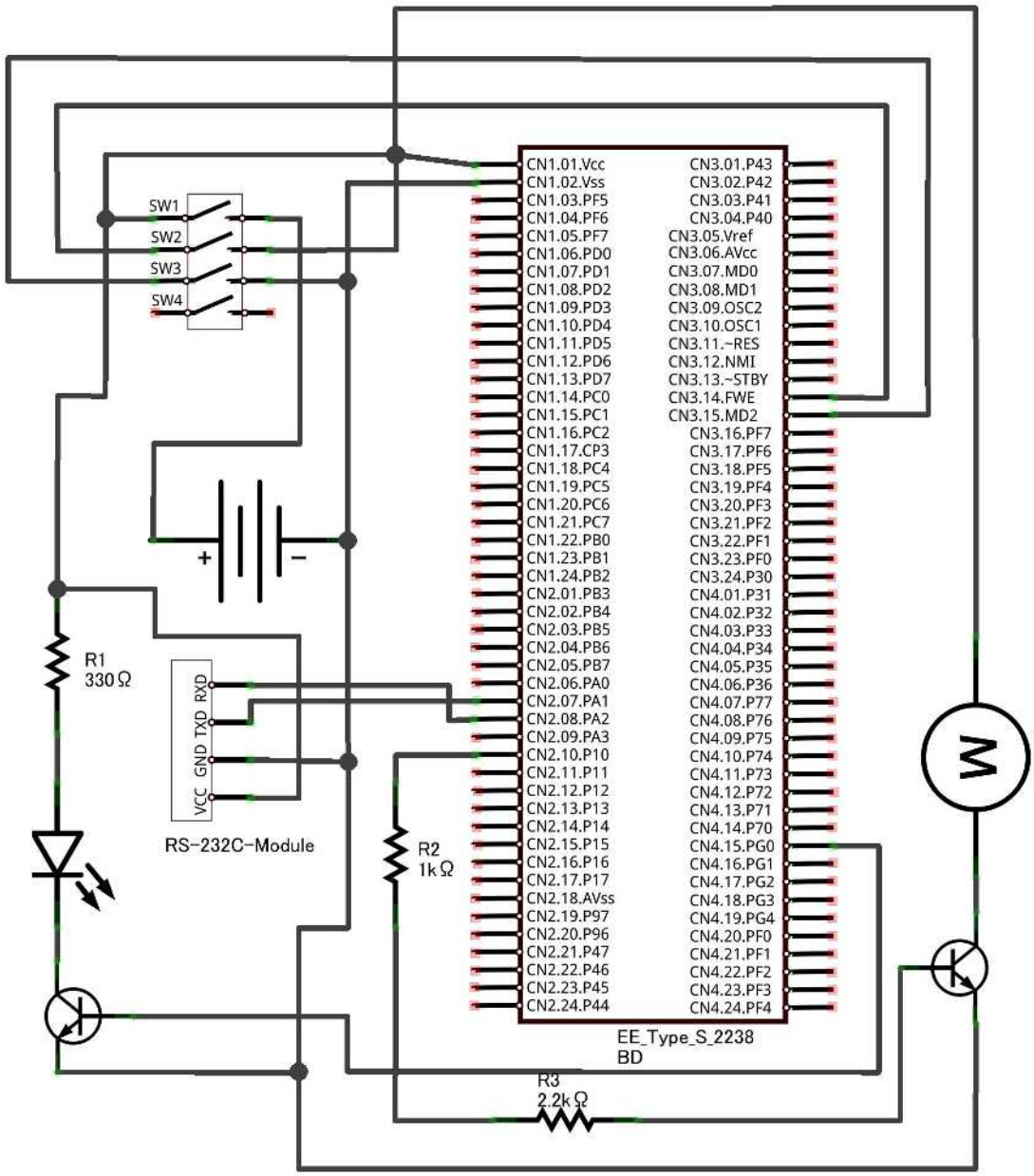
ここで登場するのが PWM(Pulse Width Modulation)です。簡単に言うと、ものすごく短い周期で ON/OFF を繰り返すのですが、１周期の中の ON している時間と OFF している時間の比率

を変えることによって、平均してかかる電圧を変化させるというものです。例えば、ON の時間が 70%、OFF の時間が 30%であれば、ON の時間にかかる電圧を 3.6V とすると平均すると 2.5V がかかることになります。
PWM を簡単に実現する周辺機能が H8S/2238 にはあります。詳しくは、ハードウエアマニュアルの 11 章　16 ビットタイマを参照ください。

【実体配線図】



【回路図】



７-４-２ソフトウエア

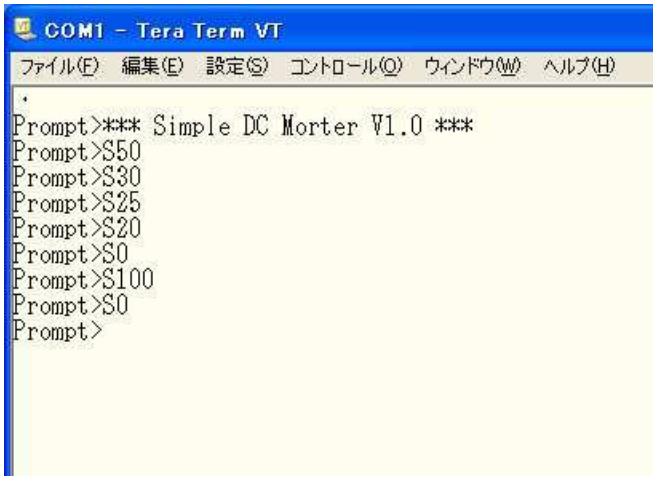
まず、動かしてみましょう。

H8S_2238 にディレクトリを移動し、SimpleDCMotor を make する。

仮想マシンの H8S_2238/00_HAL の下に SimpleDCMotor.mot ができるので、CPU に書き込んでください。

SimpleDCMotor.mot は、RS-232C 経由で支持される ON 時間の割合にしたがって、P10 の ON 時間の割合を変更するようプログラムされています。

H8S/2238 マイコンボードと PC を RS-232C ケーブルで接続し、PC 上で TeraTerm を起動しておいて、H8S/2238 マイコンボードの電源を入れてください。
TeraTerm 上で下記のように大文字 S に続けて 10 進数で ON 時間の割合(%)を入力すると、入力した数値にしたがって DC モーターの回転数が変わります。（50%程度からモーターが回り始め、25%程度まではモーターは回り続けます。）



７-４-２-１プログラムを理解しよう
仮想マシンの H8S_2238/ SimpleDCMotor の下に SimpleDCMotor.mot 用のソースプログラムがあります。
SystemFactory と SetSpeed２つのクラスから構成されています。

１）SystemFactory クラス

```
 8 : #include "SystemFactory.h"
 9 : #include "SetSpeed.h"
10 : #include <stdio.h>
11 : #include   "EE_LIB/HAL/IO/IOPin.h"
12 : #include   "EE_LIB/HAL/IntervalTimer/TimerWD.h"
13 : #include   "EE_LIB/HAL/Interrupt/Priority.h"
14 :
15 : #include   "EE_LIB/HAL/Communication/SCIUsingInterrupt.h"
16 :
17 : #include "EE_LIB/OSWrapper/Task/OSTimer.h"
18 : #include "EE_LIB/OSWrapper/Task/TaskFactory.h"
19 : #include "EE_LIB/Util/Primitive/Format.h"
20 : #include "EE_LIB/Unit/Communication/Com.h"
21 : #include "EE_LIB/Util/CommandInterpreter/CommandInterpreter.h"
22 : #include "EE_LIB/Util/CommandInterpreter/ExceptionCommand.h"
23 :
24 :
25 : using APPLI::SystemFactory;
26 :
27 : SystemFactory::SystemFactory(){}
28 : SystemFactory::~SystemFactory(){}
29 :
30 : void SystemFactory::wait()
31 : {
32 :             volatile int a,b,c,ii,jj;
33 :             for(a=10,b=20,jj=0; jj<7; jj++) {for(ii=0;ii<5000; ii++) {c=c*a+b;}}
34 : }
35 :
36 : void SystemFactory::startup()
37 : {
38 :             EE_LIB::HAL::IO::IOPin pin(EE_LIB::HAL::IO::IOPORTG,EE_LIB::HAL::IO::IOPIN0,EE_LIB::HAL::IO::IOPin::DIR_OUT);
39 :
```

```
 40 :        EE_LIB::OSWrapper::Task::OSTimer osTimer;
 41 :        EE_LIB::HAL::IntervalTimer::TimerWD
timer(EE_LIB::HAL::IntervalTimer::TimerWD::Timer0,500000UL,EE_LIB::HAL::Interrupt::PRIORITY_LOW,&osTimer); // 500uS 周期
 42 :        EE_LIB::HAL::Communication::SCI *sci;
 43 :
 44 :        //========== SCI の生成 ==========
 45 :        {
 46 :                EE_LIB::HAL::Communication::SCI::CommunicationParam cp;
 47 :                EE_LIB::HAL::Communication::SCI::SCIId id=EE_LIB::HAL::Communication::SCI::SCI2;
 48 :                cp.cm=EE_LIB::HAL::Communication::SCI::SYNCHRONUS_TOTNOAL;
 49 :                cp.bps=EE_LIB::HAL::Communication::SCI::BPS_38400;
 50 :                cp.dl=EE_LIB::HAL::Communication::SCI::DL_8BIT;
 51 :                cp.sb=EE_LIB::HAL::Communication::SCI::SB1;
 52 :                cp.parity=EE_LIB::HAL::Communication::SCI::P_NONE;
 53 :                cp.mp=EE_LIB::HAL::Communication::SCI::MP_DISABLE;
 54 :                sci=new EE_LIB::HAL::Communication::SCIUsingInterrupt(id,cp);
 55 :        }
 56 :        // ========== 通信ポートの生成 ==========
 57 :        EE_LIB::Unit::Communication::Com com(sci,&osTimer);
 58 :        // ========== コマンドインタラプタの生成 ==========
 59 :        EE_LIB::Util::CommandInterpreter::ExceptionCommand ec;
 60 :        EE_LIB::Util::CommandInterpreter::CommandInterpreter ci(&com,&ec,"Prompt>","*** Simple DC Morter V1.0 ***");
 61 :        SetSpeed setSpeed;
 62 :        ci.addCommand(&setSpeed);
 63 :        EE_LIB::OSWrapper::Task::TaskFactory::createTask(&ci,2,EE_LIB::OSWrapper::Task::Task::TP_Middle,&osTimer); // 2ms 周期のタスク
 64 :
 65 :
 66 :        while(-1)
 67 :        {
 68 :                pin.set(0);
 69 :                wait();
 70 :                pin.set(1);
 71 :                wait();
 72 :        }
 73 : }
```

【解説】

このクラスの機能は、以下の２つです。

- 通信ポートを抽象化した Com クラスを利用して、コマンドライン形式のユーザーインターフェイスを提供する CommandInterpreter に SetSpeed コマンドを登録する。
- 本体の LED を周期的に点滅させる。

この機能を実現するために、以下のステップを実施しています。

8～22 行では、このクラスで使用する各クラスの定義をインクルードしています。

30～34 行は、前の章と同じ wait 関数です。

36 行以降が startup メソッドです。(電源投入後に最初に呼び出されるメソッド)

38～41 行目では、IOPin・OSTimer・TimerWD クラスのインスタンスの生成を行っています。

44～57 行目では、38400BPS、８ビット、ストップビット１、パリティなしで通信する Com クラス生成しています。

59～63 行目では、コマンドインタプリタの生成をしています。

コマンドインタプリタは、TeraTerm などの通信ソフトから送信されるデータを、改行コードをデリミッタとして１行ごとに分解します。分解した１行はコマンドと呼びます。コマンドは適切な Command クラスの子クラス（以降コマンドクラスと呼びます）に処理を委譲して処理します。適切なコマンドクラスを決定する方法については、後述の SetSpeed クラスの解説時に説明します。コマンドは、addCommand メソッドで登録したコマンドクラスによって処理されます。

59 行目では、コマンドインタプリタの例外コマンドクラスに指定するために ExceptionCommand クラスを生成しています。例外コマンドクラスは、addCommand で登録したどのコマンドクラスでも処理されなかったコマンドを処理するコマンドクラスです。ここで

生成している ExceptionCommand クラスは、コマンドに対して何も処理せず、コマンドの後処理だけを行うクラスです。
60 行目では、CommandInterpreter クラスを生成しています。Com クラスを通信ポート、先ほど生成した ExceptionCommand クラスを例外コマンドクラス、Prompt>をプロンプト文字列、*** Simple DC Morter V1.0 ***を起動時に表示するメッセージに指定しています。
61～62 行目では、SetSpeed を生成して、コマンドクラスとして登録しています。
63 行目では、CommandInterpreter クラスをタスクとして生成しています。
66～72 行目は、マイコンボードの LED を点滅させる処理です。

２）SetSpeed クラス

このクラスの実装は、下記のようになっています。

【SetSpeed.h】

```
 1 : #if !defined(INCLUDED_SetSpeed)
 2 : #define INCLUDED_SetSpeed
 3 :
 4 : #include "EE_LIB/Util/CommandInterpreter/Command.h"
 5 : #include "EE_LIB/HAL/IntervalTimer/Timer16Bit.h"
 6 :
 7 :
 8 : namespace APPLI
 9 : {
10 :         class SetSpeed : public EE_LIB::Util::CommandInterpreter::Command
11 :         {
12 :         public:
13 :                 SetSpeed();
14 :                 virtual ~SetSpeed();
15 :                 /**
16 :                  * タイプされた行をこのクラスで処理するか判断する
17 :                  * @param command PCから入力した文字列（1行分）
18 :                  * @return true このクラスで処理する false このクラスでは処理しない
19 :                  */
20 :                 bool check(const EE_LIB::Util::Primitive::Bytes* command);
21 :                 /**
22 :                  * chaeck で trueを返すとこのメソッドがフレームワークから呼び出される
23 :                  * @param com 通信ポートクラスへのポインタ
24 :                  * @param command PCから入力した文字列（1行分）
25 :                  * @return なし
26 :                  */
27 :                 void execute(EE_LIB::Unit::Communication::Com* com, EE_LIB::Util::Primitive::Bytes* command);
28 :
29 :                 private:
30 :                         EE_LIB::HAL::IntervalTimer::PWM* m_pwm;
31 :
32 :         };
33 : }
34 : #endif
```

【SetSpeed.cpp】

```
 1 : #include "SetSpeed.h"
 2 : #include "EE_LIB/Util/Primitive/Format.h"
 3 : #include "EE_LIB/Util/Primitive/Bytes.h"
 4 : #include "EE_LIB/Unit/Communication/Com.h"
 5 :
 6 : using APPLI::SetSpeed;
 7 :
 8 :
 9 : SetSpeed::SetSpeed()
10 : {
11 :         m_pwm=new EE_LIB::HAL::IntervalTimer::Timer16Bit(EE_LIB::HAL::IntervalTimer::Timer16Bit::Timer0,204800UL,0);   // PWM0 周期
204.8uS Duty0
12 : }
13 : SetSpeed::~SetSpeed(){}
14 :
15 : bool SetSpeed::check(const EE_LIB::Util::Primitive::Bytes* command){
16 :         bool bRet=false;
17 :         if(command->size()>=1)
18 :         {               // 行の先頭に s（または S)がタイプされた行はこのクラスが処理する
19 :                         //  【コマンドの書式】
20 :                         //                                  S の後にモータの出力をパーセンテージで指定する。
21 :                         //                                  S100   100
22 :                         //                                  S20    20
23 :                         if( ( (*command)[0]=='s') || ((*command)[0]=='S'))
24 :                         { bRet=true; }
```

```
 25 :          }
 26 :          return bRet;
 27 : }
 28 :
 29 :
 30 : void SetSpeed::execute(EE_LIB::Unit::Communication::Com* com, EE_LIB::Util::Primitive::Bytes* command){
 31 :          long output;
 32 :          com->sendPrompt();                                    // プロンプトを表示する
 33 :          if(command->size() >=2)
 34 :          {
 35 :                    command->cutTheFront(1);                    // command の先頭 1 文字を削除する
 36 :                    if(command->decToValue(output)==true)       // command を 10 進文字列と仮定して数値に変換する　変換できたら True を返す
 37 :                    {
 38 :                              m_pwm->setDuty(output*100U);
 39 :                    }
 40 :          }
 41 :          com->releaseReceptionBuffer(command);                 // command を受信バッファとして再利用する
 42 : }
```

【解説】

このクラスは、Command クラスの子クラスです。

Command クラスは、check()と execute()２つの純粋仮想関数を定義したインターフェイスで、子クラスにこれら２つのメソッドの実装を強制します。

check()メソッドは、引数に指定された文字列がそのクラスで処理すべきコマンドの書式とあっている場合 True、そうでない場合 False を返すメソッドです。

execute()メソッドは、chedk()が True を返した時に、呼び出されるメソッドです。

要するに、コマンドインタラプタはコマンドを受信したときに、登録されたコマンドクラスの check()メソッド順に呼び出すことによって、コマンドを処理すべきコマンドクラスを決定し、そのクラスの execute()メソッドにコマンドを処理させるのです。

SetSpeed クラスは、S または s の次に ON 時間の割合を示す 10 進数が指定されるコマンドを処理するよう設計されています。

○SetSpeed.h

SetSpeed クラスが Command クラスの子クラスであることを宣言し、check()と execute()とメンバー変数の定義をしています。

○SetSpeed.cpp

9～12 行目は、コンストラクタです。

11 行目で、Timer16Bit クラスの使って、周期を 204800ns、デューティ比 0 で PWM 出力するよう 16 ビットタイマーのチャネル 0 を初期化しています。（Timer16Bit クラスを使った PWM 出力はコンペアマッチ A を使った出力となるので出力端子は TIOCA0(P10）となります。）

15～27 行目は、check()メソッドの実装です。

引数で渡された command の先頭 1 文字が S または s のとき True を返します。

15～27 行目は、execute()メソッドの実装です。

command には、check()メソッドに渡されたものと同じ文字列が指定されます。

35 行目で command の先頭の 1 文字（S または s）を削除しています。

35行目では、1 文字削除後の command が 10進数を示す文字列かをチェックしています。もし、

10 進数であればその数を変数 output に保存します。
38 行目で P10 の出力の ON 時間の割合を output の値に従って設定しています。
41 行目は、コマンドクラスの定型の処理で、command で指定した文字列を開放しています。コマンドクラスの execute()メソッドで、この処理を行わないとメモリーリークでフリーズするので必ずこのコマンドを実行する必要があります。（それならフレームワーク側で実施するべきでは？とお考えの方も多いと思いますが、command を再利用せずそのまま通信メッセージなどに使用する場合など次の受信バッファとして再利用できないケースもあるので、このようにしています。）

このようにコマンドインタラプタクラスとコマンドクラスを組み合わせると簡単にユーザーインターフェイスを構築できますので、パソコンからの入力に従った制御をマイコンで行うプログラムを簡単に組むことができます。

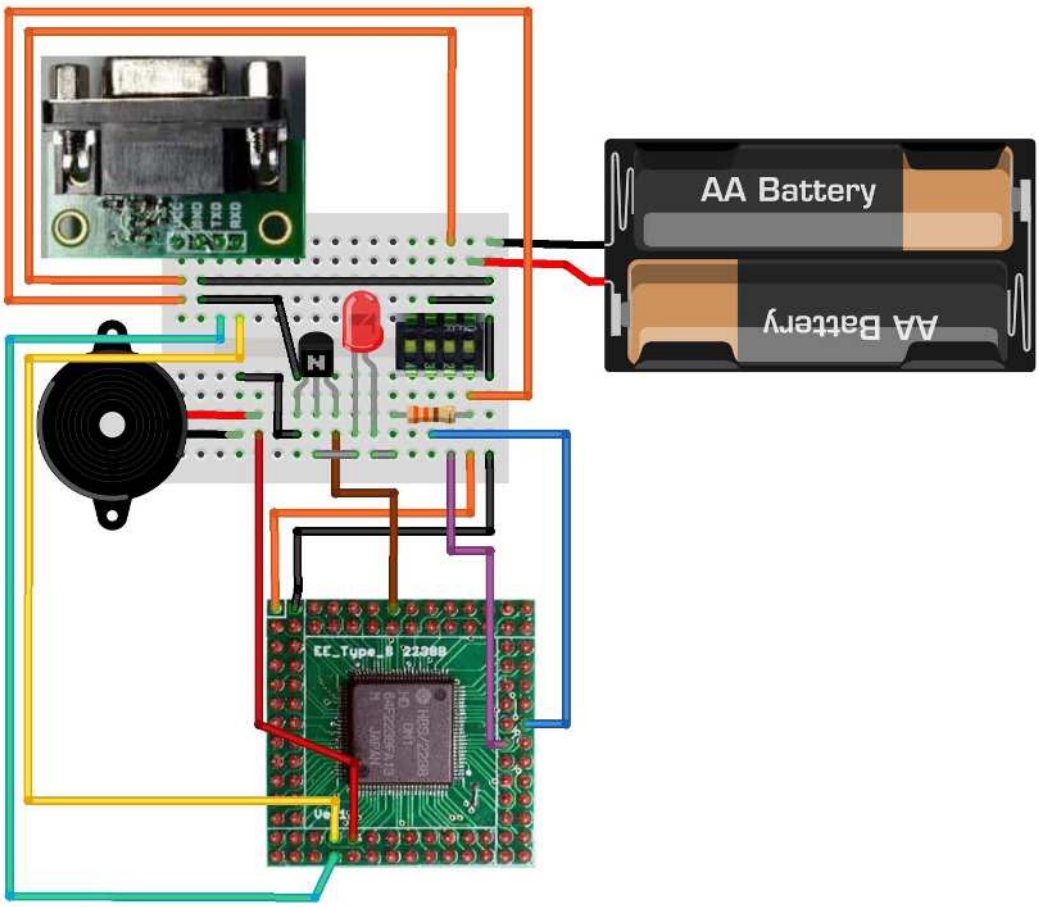
７-５ 電子音の出力

今度は、圧電ブザーを使って自動演奏をしてみよう。
モータの場合は、ON 時間と OFF 時間の比が重要で、パルスの周期は重要ではありませんでした。今回もパルスを出力するわけですが、制御対象が音なので今回はパルスの周期が重要な要素になります。

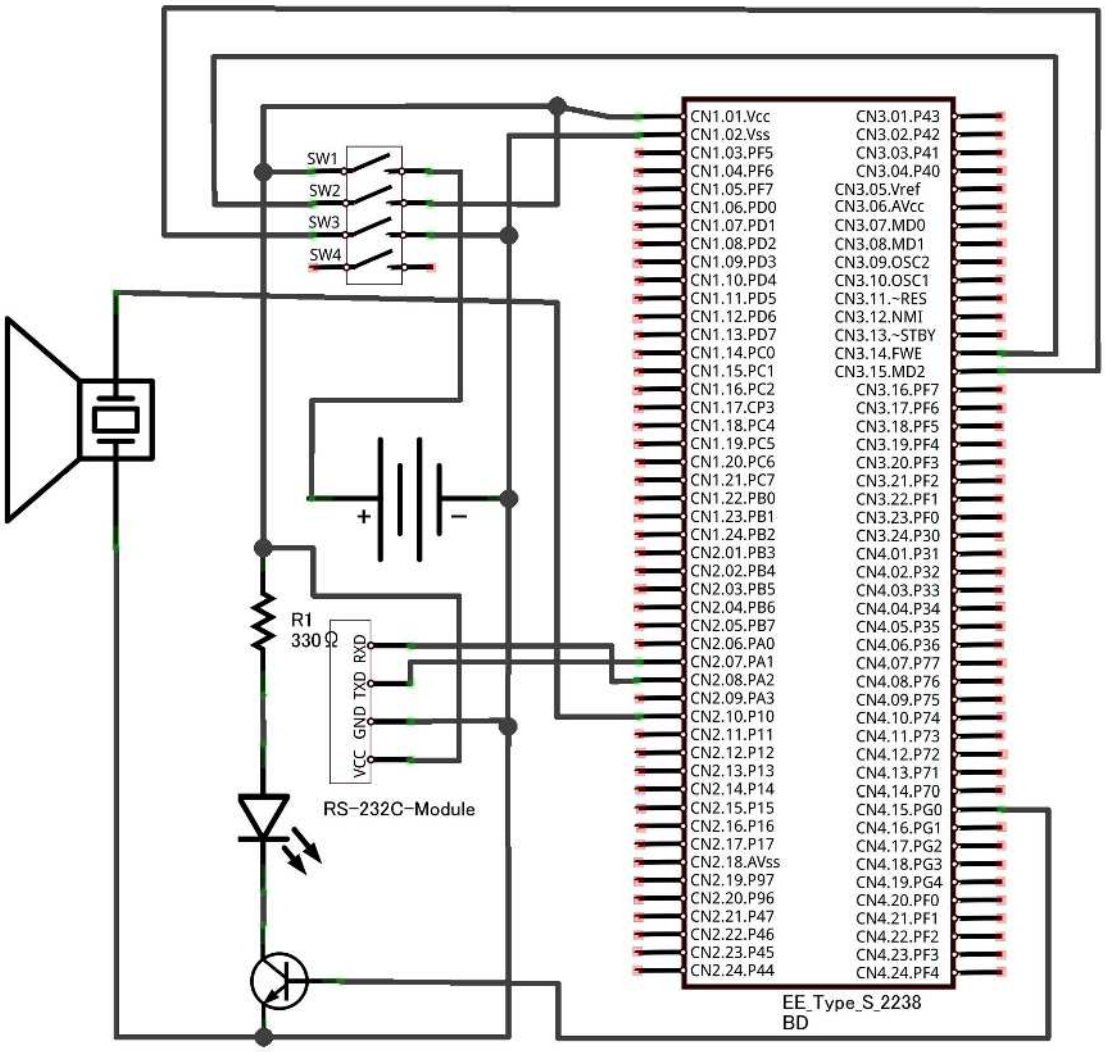
７-５-１ ハードウエア

ハードウエアマニュアルを見ると H8S/2238 は、16 ビットタイマ・8 ビットタイマ・ウォッチドックタイマの３種のタイマがあることが分かります。
音程を変えた音を出力するには、周期を変えたパルスを出力する必要がありますが、そのような機能を持つタイマは、16 ビットタイマ・8 ビットタイマの２種です。
今回は 16 ビットタイマを使って、自動演奏をしてみたいと思います。
ハードウエアマニュアルの 11 章を見ると、６チャネルの 16 ビットタイマがありますが、今回はチャネル０を使うことにします。コンペアマッチ A により、波形を出力するので、出力端子は、P10 になります。
H8S/2238 はもともと音楽専用の機能を持っていないのでゲーム機のように音色を変えたり、音量を変えたりしてきれいな音を出すことはできないのですが、パルスの周期を制御できるので音程だけは制御できます。パルス出力を音に変える電子部品に圧電ブザーというものがあります。圧電ブザーを P10 に直結するだけで、自動演奏の準備が完成します。

【実体配線図】



【回路図】



７-５-２ソフトウエア

まず、動かしてみましょう。

H8S_2238 にディレクトリを移動し、SimpleSound を make する。

仮想マシンの H8S_2238/00_HAL の下に SimpleSound.mot ができるので、CPU に書き込んでください。

電源を入れると「エリーゼのために」の自動演奏が始まります。

７-５-２-１プログラムを理解しよう

仮想マシンの H8S_2238/ SimpleSound の下に SimpleSound.mot 用のソースプログラムがあります。

SystemFactory のみから構成されています。

３）SystemFactory クラス

```
  8 : #include "SystemFactory.h"
  9 : #include <stdio.h>
 10 : #include   "EE_LIB/HAL/IO/IOPin.h"
 11 : #include   "EE_LIB/HAL/IntervalTimer/TimerWD.h"
 12 : #include   "EE_LIB/HAL/Interrupt/Priority.h"
 13 :
 14 : #include "EE_LIB/OSWrapper/Task/OSTimer.h"
 15 : #include "EE_LIB/OSWrapper/Task/TaskFactory.h"
 16 : #include "EE_LIB/OSWrapper/Task/Task.h"
 17 : #include "EE_LIB/HAL/IntervalTimer/Timer16Bit.h"
 18 : #include "EE_LIB/Unit/Sound/SoundPlayerUsingPWM.h"
 19 : #include "EE_LIB/Unit/Sound/Sound.h"
 20 :
 21 :
 22 : using APPLI::SystemFactory;
 23 :
 24 : SystemFactory::SystemFactory(){}
 25 : SystemFactory::~SystemFactory(){}
 26 :
 27 : void SystemFactory::wait()
 28 : {
 29 :         volatile int a,b,c,ii,jj;
 30 :         for(a=10,b=20,jj=0; jj<7; jj++) {for(ii=0;ii<5000; ii++) {c=c*a+b;}}
 31 : }
 32 :
 33 : void SystemFactory::startup()
 34 : {
 35 :         EE_LIB::HAL::IO::IOPin pin(EE_LIB::HAL::IO::IOPORTG,EE_LIB::HAL::IO::IOPIN0,EE_LIB::HAL::IO::IOPin::DIR_OUT);
 36 :
 37 :         EE_LIB::OSWrapper::Task::OSTimer osTimer;
 38 :         EE_LIB::HAL::IntervalTimer::TimerWD
timer(EE_LIB::HAL::IntervalTimer::TimerWD::Timer0,500000UL,EE_LIB::HAL::Interrupt::PRIORITY_LOW,&osTimer); // 500uS 周期
 39 :
 40 :         // ========== SoundPlayer の生成   ==============
 41 :         EE_LIB::HAL::IntervalTimer::Timer16Bit timer0(EE_LIB::HAL::IntervalTimer::Timer16Bit::Timer0,1000000UL,0);       //  PWM0  周 期
1ms Duty0
 42 :
 43 :         EE_LIB::Unit::Sound::SoundPlayerUsingPWM soundPlayer(&timer0);       // timer0 を PWM タイマに指定して SoundPlayer を生成
 44 :         EE_LIB::OSWrapper::Task::TaskFactory::createTask(&soundPlayer,10,EE_LIB::OSWrapper::Task::Task::TP_Middle,&osTimer);  //  10ms
周期のタスク
 45 :
 46 :
 47 : // For Elise
 48 :         EE_LIB::Unit::Sound::Sound
sound("V255T250_L16E5D#5E5D#5E5B4D5C5L8A4L16.C4E4A4L8B4L16.E4G#4B4L8C5L16.E4E5D#5E5D#5E5B4D5C5_L8A4L16.C4E4A4L8B4L16.
E4C5B4L4A4_L16E5D#5E5D#5E5B4D5C5L8A4L16.C4E4A4L8B4L16.E4G#4B4L8C5L16.E4E5D#5E5D#5E5B4D5C5_L8A4L16.C4E4A4L8B4L16.E4C
5B4L8A4L16.B4C5D5L8HE5L16G4F5E5_L8HD5L16F4E5D5L8HC5L16E4D5C5L8B4L16.E4E5..E5E6..D#5E5..D#5E5D5_E5B#5E5B4D5C5L8A4L16.C
4E4A4L8B4L16.E4G#4B4L8C5L16.E4E5D#5E5D#5E5B4D5C5_L8A4L16.C4E4A4L8B4L16.E4C5B4L8A4L16.");
 49 :
 50 :         soundPlayer.setSound(&sound,1);
 51 :
 52 :
 53 :         while(-1)
 54 :         {
 55 :                 pin.set(0);
 56 :                 wait();
 57 :                 pin.set(1);
 58 :                 wait();
 59 :                 if(soundPlayer.getStatus()==false)
 60 :                 {
 61 :                         // 演奏終了していたら、繰り返す
 62 :                         soundPlayer.setSound(&sound,1);
 63 :                 }
 64 :         }
 65 : }
```

【解説】

このクラスの機能は、以下の２つです。

- sound で定義した曲の自動演奏を繰り返す。
- 本体の LED を周期的に点滅させる。

この機能を実現するために、以下のステップを実施しています。

8～19 行では、このクラスで使用する各クラスの定義をインクルードしています。

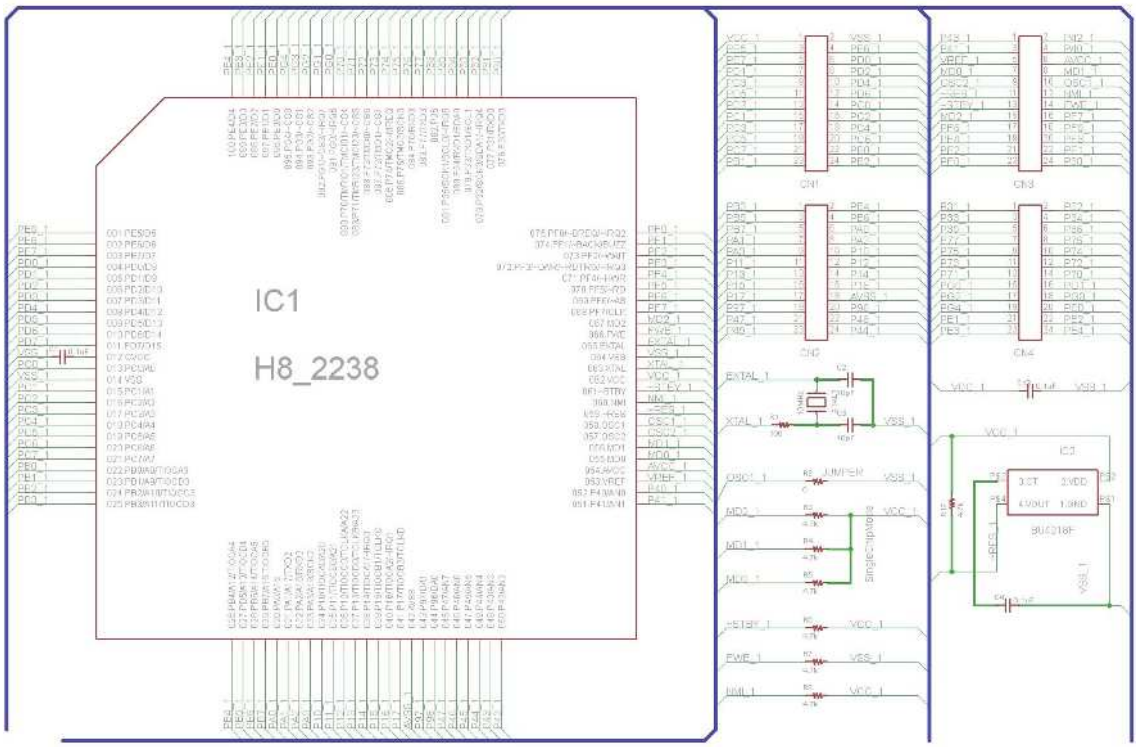
27～31 行は、前の章と同じ wait 関数です。

33 行以降が startup メソッドです。(電源投入後に最初に呼び出されるメソッド)

35～38 行目では、IOPin・OSTimer・TimerWD クラスのインスタンスの生成を行っています。
41 行目では、PWM モードの Timer16Bit クラスを生成しています。
43 行目では、41 行目で使ったタイマを使った SoundPlayerUsingPWM を生成しています。
44 行目では、SoundPlayerUsingPWM を 10ms 周期のタスクとして生成しています。
48 行目は、エリーゼのためにの曲データを持つ Sound クラスを生成しています。
50 行目では、SoundPlayerUsingPWM クラスを使って、Sound クラスの曲を 1 回目の再生を開始しています。
53～64 行目は、マイコンボードの LED を点滅させる処理と演奏が終了したら再度 Sound クラスを再生する処理を行っています。
53 行目以降は、音を出す処理をプログラム上は行っていないことがわかると思います。音を出す処理に関しては、タスクとして生成された SoundPlayerUsingPWM クラスがバックグラウンドで行っています。
このように EE_LIB は、ノンプリエンプション型ではありますが擬似的なマルチタスク環境を提供するのです。

7 - 6 H8S/2238 マイコンのハードウエア

本体のハードウエアについて解説します。
回路図は、下記の通りです。



以下の部分から構成されています。

・　発振回路

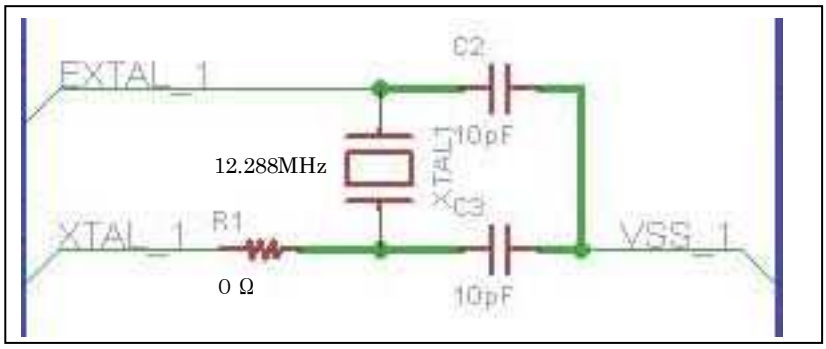
・　リセット回路

・　起動モードの設定回路

①　発振回路

右の部分が発振回路です。

これも H8S/2238 ハードウエアマニュアルの 23 章で推奨されている回路そのままです。

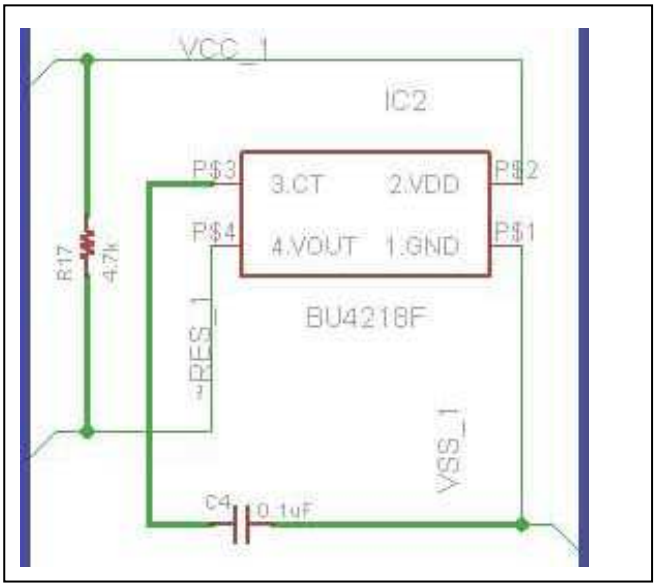


②　リセット回路

この回路も BU4218F のデータシートの推奨のパワーオンリセット回路です。

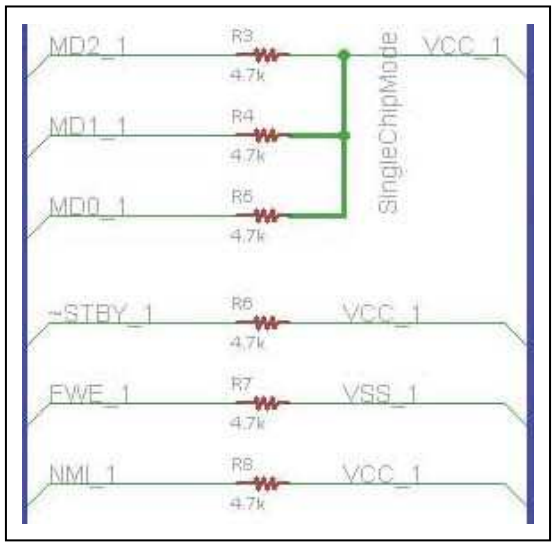
H8S/2238 のハードウエアマニュアルのリセットシーケンスでは、「電源投入時は最低 20ms の間、¯RES 端子を Low レベルに保持してください。」とあります。

Ct に 0.1uF のコンデンサを接続しているので、BU4218F のデータシートのグラフから遅延時間を読むと 500ms となり、十分な時間リセット端子を Low レベルに保持できます。



③　起動モードの設定回路

通常はシングルチップモード（動作モード７）で起動するよう端子を処理しています。（H8S/2238 ハードウエアマニュアルの３章を参照）
プログラムを書き込むときは、MD2 を０、FWE を１に設定することにより、ブートモードで起動します。（H8S/2238 ハードウエアマニュアルの 20.2 章を参照）



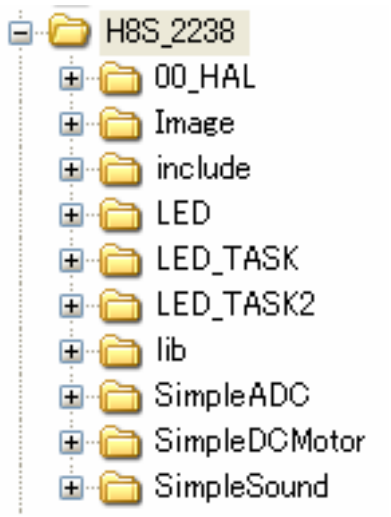
８ソフトウエアを理解しよう

マイコンで動くプログラムを作るためには､パソコンでプログラムを動かすための知識とは別の特殊な知識が必要です。この章では、そのようなマイコン開発特有の知識を紹介します。

８-１ EE_LIB を使った開発環境

８-１-１ディレクトリ構成

ユーザーgcc のホームディレクトリの下の H8S_2238 の下に EE_LIB を使った開発環境関連のファイルはあります。ディレクトリは下記のようになっています。



1） H8S_2238

1 階層目の makefile とそのパラメータを格納しています。

2） 00_HAL

２階層目の makefile とリンカスクリプト、スタートアップルーチンを格納しています。

3） Image

サンプルプログラムの実行イメージ（mot 形式のファイル）と map ファイルを格納しています。

4） include

EE_LIB のヘッダーファイルを格納しています。

5） lib

EE_LIB を格納しています。

6） その他のディレクトリ

サンプルプログラムを格納しています。

8-1-2プログラムの作成

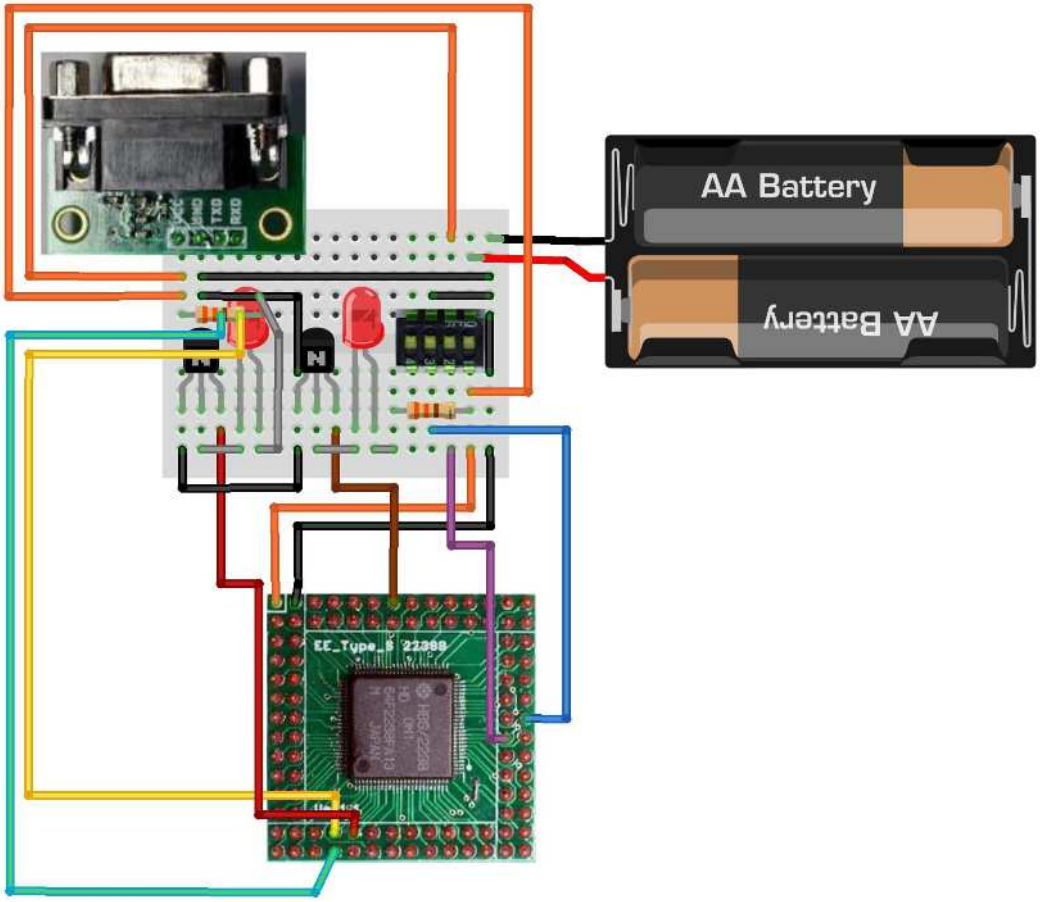
これまでにサンプルプログラムの解説を行ってきましたが、今回はプログラムを作ってみましょう。

7-2 では LED を１秒周期で点滅させるプログラムを紹介しましたが、今回は LED をもう一つ追加して 400ms 周期で点滅させましょう。

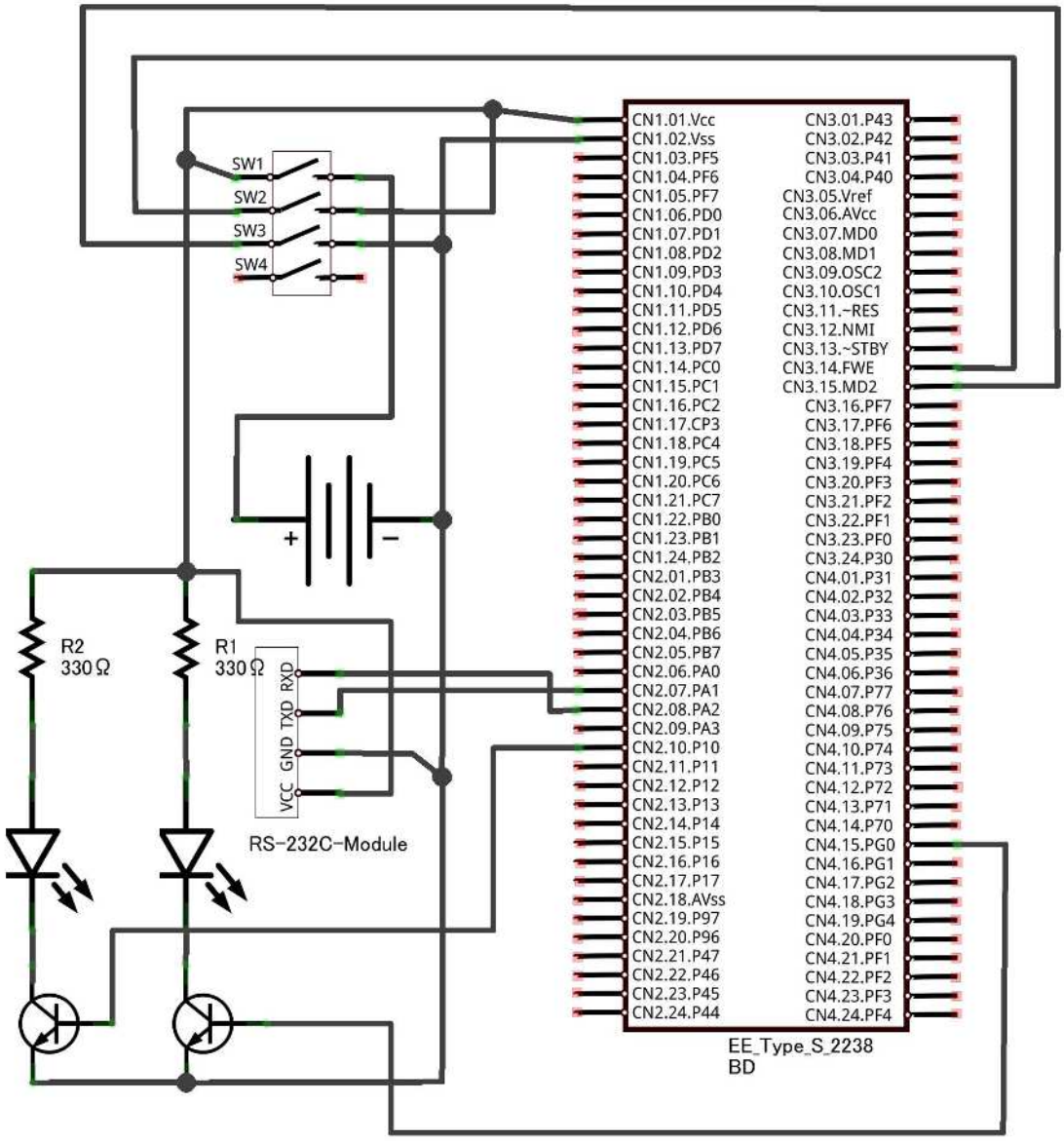
８-１-２-１ ハードウエア

今回は LED の出力端子に P10 を使います。LED と抵抗の接続は PG0 と同じです。

【実体配線図】



【回路図】



8-1-2-2 ソフトウエア

EE_LIB を使ったプログラムの追加は、下記のステップで実施します。

1） Step1

H8S_2238 の下にディレクトリを作り、SystemFactory クラスの startup メソッドから始まるプログラムを記述する。

2） Step2

H8S_2238 の下の makefile を修正する。

3） Step3

H8S_2238/ 00_HAL の下の makefile を修正する。

８-１-２-３　Step1

今回は、LED_TASK ディレクトリの下のファイルをコピーして、LED_TASK２というディレクトリの下に格納してください。

さらに、SystemFactory.cpp を下記のリストのように修正します。（25,32,34 行目を追加してください。）

これで、P10 に接続した LED を 400ms 周期で点滅するタスクの追加が完了しました。

```
  8 : #include "SystemFactory.h"
  9 : #include   "EE_LIB/HAL/IO/IOPin.h"
 10 : #include   "EE_LIB/HAL/IntervalTimer/TimerWD.h"
 11 : #include   "EE_LIB/HAL/Interrupt/Priority.h"
 12 : #include "EE_LIB/OSWrapper/Task/OSTimer.h"
 13 : #include "EE_LIB/OSWrapper/Task/CyclicTask.h"
 14 : #include "EE_LIB/OSWrapper/Task/TaskFactory.h"
 15 : #include   "LED_Task.h"
 16 :
 17 : using APPLI::SystemFactory;
 18 :
 19 : SystemFactory::SystemFactory(){}
 20 : SystemFactory::~SystemFactory(){}
 21 :
 22 : void SystemFactory::startup()
 23 : {
 24 :         EE_LIB::HAL::IO::IOPin pin(EE_LIB::HAL::IO::IOPORTG,EE_LIB::HAL::IO::IOPIN0,EE_LIB::HAL::IO::IOPin::DIR_OUT);
 25 :         EE_LIB::HAL::IO::IOPin pin1(EE_LIB::HAL::IO::IOPORT1,EE_LIB::HAL::IO::IOPIN0,EE_LIB::HAL::IO::IOPin::DIR_OUT);
 26 :
 27 :
 28 :         EE_LIB::OSWrapper::Task::OSTimer osTimer;
 29 :         EE_LIB::HAL::IntervalTimer::TimerWD
timer(EE_LIB::HAL::IntervalTimer::TimerWD::Timer0,1000000UL,EE_LIB::HAL::Interrupt::PRIORITY_LOW,&osTimer);          // 1ms周期割り込み
 30 :
 31 :         LED_Task ledTaskHandler(&pin);
 32 :         LED_Task ledTaskHandler1(&pin1);
 33 :         EE_LIB::OSWrapper::Task::TaskFactory::createTask(&ledTaskHandler,1000U,EE_LIB::OSWrapper::Task::Task::TP_Middle,&osTimer); //
1 秒周期のタスクの生成
 34 :         EE_LIB::OSWrapper::Task::TaskFactory::createTask(&ledTaskHandler1,400U,EE_LIB::OSWrapper::Task::Task::TP_Middle,&osTimer); //
400ms 周期のタスクの生成
 35 :
 36 :         while(-1) {
 37 :         }
 38 : }
```

８-１-２-４　Step2

H8S_2238 の下の makefile を以下のように修正しターゲットを追加します。

１）　Applications にプログラム名を追加する。

48 行目の最後にスペースを追加し、その後に今回追加するプログラム名（LED_TASK2）を追加します。

２）　依存関係を追記する。

アプリケーションの依存関係を、99～103 行目のように追記します。

```
100 : LED_TASK2::APPLI=LED_TASK2
101 : LED_TASK2::COPT += -DWDT_M
102 : LED_TASK2:COMMON_MK
103 :     $(MAKE) --directory=$(HOME_DIRECTORY)/00_HAL LED_TASK2
```

依存関係は、３つのパートで記述します。

最初と２番目のパートは、１）で追加したプログラム名に::を付けた文字列からスタートします。

最後のパートは、追加したプログラム名に:を付けた文字列からスタートします。
最初のパート（100 行目）は、プログラム名を APPLI という変数に代入します。APPLI という変数は、共通なので別のプログラムを追加するときのこのパートの記述で変わるのはプログラム名のみです。
2 番目のパート（101 行目）は、プログラムが使用する CPU の周辺機能を定義します。
COPT += の後ろに使用する CPU の周辺機能をスペースで区切って並べます。選択肢は、下記の通りです。LED_TASK2 ではウォッチドックタイマー（TimerWD クラス）を使っているので WDT_M を指定します。

- SCI_M　　SCI
- ADC_M　　ADC
- WDT_M　　ウォッチドックタイマー
- T8_M　　8bit タイマー
- T16_M　　16bit タイマー
- INTERRUPT_M　　割り込み

最後のパート（102、103 行目）は、プログラムを格納したディレクトリ名を指定します。
別のプログラムを追加するときに、変更する必要のあるこのパートの記述はディレクトリ名のみです。

102 : **LED_TASK2**:COMMON_MK
103 :　　$(MAKE) --directory=$(HOME_DIRECTORY)/00_HAL **LED_TASK2**

【修正後の makefile】

```
 45 :
 46 : ############################## Target ###################################
 47 : # EDIT When Application add
 48 : Applications = LED LED_TASK SimpleDCMotor SimpleADC RotationalSpeedCounter SimpleSound SimpleDataLogger LED_TASK2
 49 :
 50 : ##############################   Directorys ##############################
 51 : Appl :=$(addprefix $(HOME_DIRECTORY)/,$(Applications))
 52 : Appl_ROM :=$(HOME_DIRECTORY)/00_HAL
 53 :
 54 : .PHONY: clean $(Applications) COMMON_MK
 55 : ############################## Dependence ###############################
 56 : # EDIT When Application add
 57 :
 58 : ########################
 59 : LED::APPLI=LED
 60 : LED:COMMON_MK
 61 :　　$(MAKE) --directory=$(HOME_DIRECTORY)/00_HAL LED
 62 :
 63 : ########################
 64 : LED_TASK::APPLI=LED_TASK
 65 : LED_TASK::COPT += -DWDT_M
 66 : LED_TASK:COMMON_MK
 67 :　　$(MAKE) --directory=$(HOME_DIRECTORY)/00_HAL LED_TASK
 68 :
 69 : ########################
 70 : SimpleDCMotor::APPLI=SimpleDCMotor
 71 : SimpleDCMotor::COPT += -DWDT_M -DSCI_M
 72 : SimpleDCMotor:COMMON_MK
 73 :　　$(MAKE) --directory=$(HOME_DIRECTORY)/00_HAL SimpleDCMotor
 74 :
 75 : ########################
 76 : SimpleADC::APPLI=SimpleADC
 77 : SimpleADC::COPT +=　-DWDT_M -DSCI_M -DADC_M
 78 : SimpleADC:COMMON_MK
 79 :　　$(MAKE) --directory=$(HOME_DIRECTORY)/00_HAL SimpleADC
 80 :
　　～～～～～～～～～～～～　省略　～～～～～～～～～～～～～
 98 :
 99 : ########################
100 : LED_TASK2::APPLI=LED_TASK2
101 : LED_TASK2::COPT += -DWDT_M
102 : LED_TASK2:COMMON_MK
103 :　　$(MAKE) --directory=$(HOME_DIRECTORY)/00_HAL LED_TASK2
104 :
105 :
106 : #----------------------- clean ---------------------------------
107 : clean:
108 :　　for d in $(Appl_ROM) ; ¥
```

```
109 :            do ¥
110 :            ( $(MAKE) --directory=$$d clean ); ¥
111 :            done
112 :
113 : #------------------------ libraries ---------------------------------
114 : COMMON_MK:
115 :            cp common.org.mk common.mk
116 :            sed -e "s/CFLAGS =/CFLAGS = $(COPT)/" common.org.mk>tmp.mk
117 :            sed -e "s/APPLICATION = /APPLICATION = $(APPLI)/" tmp.mk>tmp1.mk
118 :            sed -e "s/H8_TARGET :=/H8_TARGET := $(H8_TARGET)/" tmp1.mk>common.mk
119 :            rm -f tmp.mk tmp1.mk
120 :
121 :
```

8-1-2-5　Step3

H8S_2238/ 00_HAL の下の makefile を以下のように修正しターゲットを追加します。

33 行目から 35 行目のように、プログラム名とプログラムを構成するソースファイル名の関連を定義します。

別のプログラムを追加するときに、この makefile に追加する内容で変更すべき箇所は、２箇所です。

１つ目は、33 行目のプログラム名（LED_TASK2 に相当する文字列）です。

２つ目は、34 行目の main.o sysinit.o HALInit.o 以降です。この行は、OBJECTS という変数にプログラムを構成するソースファイル名を代入しているのですが、main、sysinit、HALInit はデフォルトで必要です。その後ろに、追加したプログラムのディレクトリに作成したソースファイル名の拡張子 cpp から o に変更して、スペースで区切ってつなげます。

```
33：ifeq "$(APPLICATION)" "LED_TASK2"
34：OBJECTS                = main.o sysinit.o HALInit.o    SystemFactory.o LED_Task.o
35：endif
```

【修正後の makefile】

```
 1 : include ../common.mk
 2 : include ../appli.mk
 3 :
 4 : INCLUDE_OPT += -I./ -I../$(APPLICATION)/
 5 :
 6 : # ************ OBJECTS **************
 7 : ifeq "$(APPLICATION)" "LED"
 8 : OBJECTS                                = main.o sysinit.o SystemFactory.o HALInit.o
 9 : endif
10 : ifeq "$(APPLICATION)" "LED_TASK"
11 : OBJECTS                                = main.o sysinit.o SystemFactory.o LED_Task.o HALInit.o
12 : endif
13 : ifeq "$(APPLICATION)" "SimpleDCMotor"
14 : OBJECTS                                = main.o sysinit.o SystemFactory.o SetSpeed.o HALInit.o
15 : endif
16 :
17 : ifeq "$(APPLICATION)" "SimpleADC"
18 : OBJECTS                                = main.o sysinit.o SystemFactory.o ADC.o HALInit.o
19 : endif
     ～～～～～～～～～～～～　省略　～～～～～～～～～～～～～～
32 :
33 : ifeq "$(APPLICATION)" "LED_TASK2"
34 : OBJECTS                                = main.o sysinit.o HALInit.o SystemFactory.o LED_Task.o
35 : endif
36 : # ************ Target define **************
37 :  ifeq "$(findstring EE_S2238,$(CFLAGS))" "EE_S2238"
38 : LDSCRIPT                = EE_S2238_INTERNAL_ROM.x
39 : OBJECTS += CRT0_IN.o
40 : endif
41 : # ************** vpath *******************
42 : vpath x ../$(APPLICATION)/
43 : vpath xpp ../$(APPLICATION)/
44 : vpath   ../$(APPLICATION)/
45 : vpath   ../$(APPLICATION)/
46 :
47 : # *********** Dependance *****************
48 : $(APPLICATION).mot : $(APPLICATION).abs
49 :            $(OBJCOPY) -O srec $(APPLICATION).abs $(APPLICATION).mot
```

```
50 :
51 : $(APPLICATION).abs : $(OBJECTS)
52 :             $(CPP) -o $(APPLICATION).abs $(LDFLAGS) $(LIBS) $(LIBS) $(LIBS) -T$(LDSCRIPT) $^ $(LIBS) $(LIBS) $(LIBS)
53 :
54 : # ****************** clean *********************
55 : .PHONY:clean
56 : clean:
57 :             rm *.abs *.mot *.o *.map *.BAK
58 : # ****************** APPLI *********************
59 : .PHONY:$(APPLICATION)
60 : $(APPLICATION) : $(APPLICATION).mot
61 : $(APPLICATION) : $(APPLICATION).mot
```

８-１-２-６ 実行モジュールの生成

下記を実行すると、実行モジュールが H8S_2238/ 00_HAL の下にできます。

make clean

make LED_TASK2

８-２ EE_LIB を使わない開発環境

マイコンを使った電子工作では、CPU 外付けのハードウエアや CPU 内の周辺機能の理解とともに、ソフトウエアの理解が必要になります。簡単なものでも、このように広範囲の知識の習得が必要になるので、動くものを作るまでに時間を要し、習得できないことも多いのです。また、知識が少ない状態だと、作ったものが動かなかったときに、ハードウエアに問題があるのか、ソフトウエアに問題があるのかを切り分けるのも大変です。

このような課題を解決するために、（8-1 を含め）ここまでは EE_LIB を使ってソフトウエアは簡単に作ることによって、ハードウエアや CPU の周辺機能を手っ取り早く動かし、理解することを目的に解説してきました。

ハードウエアの理解や動くものを作ることが目的の方には、目的にあった内容だと思いますが、一方、ソフトウエアでハードウエアを制御することを目的とされる方には、物足りない内容だったかもしれません。（今までの解説内容は、ハードウエアの動作を確認する手段として活用いただければ幸いです。）

この章では、いよいよ EE_LIB を使わずにプログラミングする方法を解説します。

8-1　LED 点滅プログラム

PG0 に接続した LED を点滅させるプログラムを見てみましょう。ハードウエアは、7-2 章で使ったものと同じものを使います。

LED というディレクトリの下に実行モジュールの生成に必要なファイル一式があります。

```
Desktop/  H8S_2238/  LED/  upgrade-log
[gcc@vineLinux gcc]$ cd LED
[gcc@vineLinux LED]$ ls
CRT0_IN.S  EE_S2238_INTERNAL_ROM.x  main.c  makefile  sysinit.c
```

・ CRT0_IN.S　　スタートアップメインルーチン

・ H8S_2238_INTERNAL_ROM.x リンカスクリプト
・ main.c ユーザープログラム（LED 点滅のソースファイルです）
・ makefile make ファイルです
・ sysinit.c スタートアップサブルーチン

それぞれのファイルの中身を見ると、main.c が今回実現したいことをプログラムしているソースファイルですが、それ以外（main.c を動かすための仕組み）の部分のボリュームが大きいことに気づくと思います。

このようにマイコンのプログラミングでは、実現したいことをプログラムする以外にそのプログラムを動かす仕組みを用意する必要があり、この部分が入門のハードルをあげている一つの要因です。

以降、それぞれのファイルについて解説します。

８-２-１リンカスクリプト

パソコンなど OS 上で動作するプログラムと（OS を持たない）マイコン上で動作するプログラムの一つの違いが、マイコン上で動作するプログラムにはリンカスクリプトとスタートアップルーチンが必要だということです。

リンカスクリプトは、プログラムコードやデータのメモリへの配置方法をリンカに指示するものです。当キットでは、コンパイラに GCC を採用しているので、GCC のリンカである LD 用のリンカスクリプトを書くことになります。

7-6 章でこのキットの H8S/2238 はアドバンスモード（モード７）で起動するようハードウエアが設計されていると解説しました。ハードウエアマニュアルの 3.4 章を見てください。

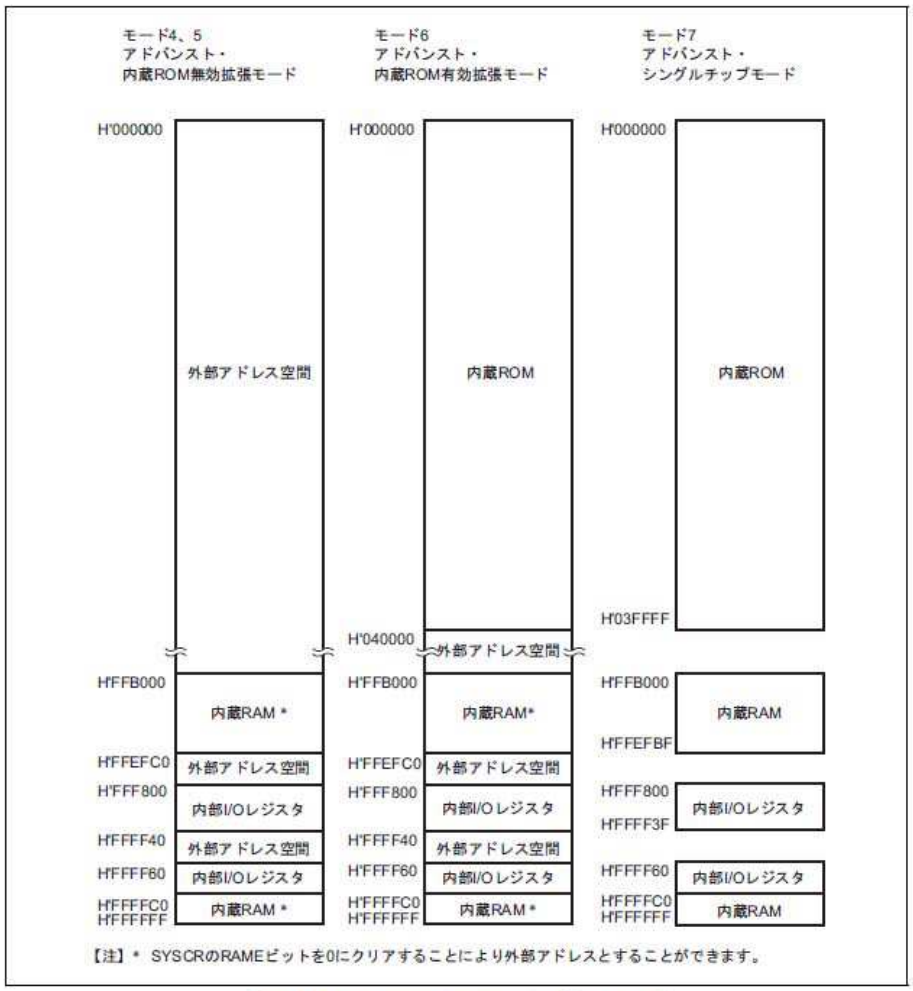


図 3.4　H8S/2238B、H8S/2238R のアドレスマップ

これを見ると、0～3FFFFF 番地が ROM、FFB000～FFEFBF 番地および FFFFC0～FFFFFF 番地が RAM であることが分かります。

一方、GCC ではプログラムをコンパイルするときに、その構成要素を text、data、bss、stack の４つに分類します。text プログラムコード、data は初期値を持った変数、bss は初期値を持たない変数、stack はオート変数です。

リンカスクリプトは、text を ROM に、data、bss、stack を RAM に配置するよう CPU のメモリーマップをリンカに教える役目をします。

リンカスクリプトは、メモリーマップが変わらない限り、修正する必要性が低いものなので、ここで文法など詳しい解説はしません。マイコンのプログラムを実行モジュールに変換する際に、このような機能を持つファイルが必要だということだけ覚えておいてください。（RAM を増設するなどメモリーマップを変更したくなった時に、LD のマニュアルを参照してください）

当キット用にプログラムする場合、リンカスクリプトは下記を使ってください。

【H8S_2238_INTERNAL_ROM.x】

```
1 : OUTPUT_FORMAT("elf32-h8300")
```

```
 2 : OUTPUT_ARCH(h8300s)
 3 : ENTRY("_start")
 4 : /* BootMode:Mode7 with EE_Type_S/2238 */
 5 : MEMORY
 6 : {
 7 :             vectors(r)    : o = 0x000000, l = 0x000200
 8 :             rom2(rwx)     : o = 0x200, l = (0x500)
 9 :             rom(rx)       : o = (0x200+0x500)     , l = (0x40000-0x200-0x500)
10 :             ram2(rwx)     : o = 0xffb000, l = (0x500)
11 :             ram3(rwx)     : o = (0xffb000+0x500), l = (0x3fc0-0x500-0x100)
12 :             stack1(rw)    : o = 0xffefbc, l = 0x000004
13 :             ram4(rwx)     : o = 0xffffc0, l = 0x40
14 : }
15 : SECTIONS
16 : {
17 : .vectors : { *(.vect) }   > vectors
18 :
19 : /* text : program code */
20 : .text : {
21 :             *(.text)
22 :             *(.text.*)
23 :               *(handler)
24 :             *(.rodata)
25 :             ___ctors = . ;
26 :             *(.ctors)
27 :             ___ctors_end = . ;
28 :             ___dtors = . ;
29 :             *(.dtors)
30 :             *(.rodata.*)
31 :             *(.gnu.linkonce.t*)
32 :             *(.gnu.linkonce.r*)
33 :             *(.comment*)
34 :             *(.strings)
35 :             *(.gcc_except_table*)
36 :             ___dtors_end = . ;
37 :               _etext = . ;
38 :             } > rom
39 : /* bss : Variable without initialization */
40 : .stack : {
41 :               _stack_external = . ;
42 :             }   > stack
43 : .data : AT ( 0x200   ){
44 :               _handler_begin = .;
45 :               _handler_end = .;
46 : /* data : Initialized variable */
47 :             _data_begin = .;
48 :             ___data = . ;
49 :             *(.data)
50 :             *(.tiny)
51 :             *(.gnu.linkonce.d*)
52 :             *(.gnu.linkonce.s*)
53 :               _edata = .;
54 :             _data_end = .;
55 :             } > ram2
56 : .bss   : {
57 :               _bss_begin = .;
58 :               *(handler_RAM)
59 :             *(.bss)
60 :             *(.gnu.linkonce.b*)
61 :             *(COMMON)
62 :               _bss_end     = .;
63 :             /* for stdlib :When the memory is used from the heap,this symbol(_end) is used. */
64 :             _end = .;
65 :             }   > ram3
66 : .heap ALIGN(4) : {
67 :             _heap_start = . ;
68 :             _heap_end = (. + 0x2000) ;
69 :             } > ram3
70 : .stack1 : {
71 :               _stack_internal = . ;
72 :             }   > stack1
73 : }
```

８-２-２スタートアップルーチン

C 言語を習った方はプログラムを書くときは、main からプログラムがスタートすると習ったと思います。C 言語の言語仕様は確かにその通りなのですが、マイコンではその仕様を構築する仕組みを自前で用意しなければなりません。

その役割を果たすのが、スタートアップルーチンです。

当キットのマイコンは電源を入れるとリセット回路により￣RES 端子が Low レベルを保持し

た後に High になります。これにより、ハードウエアマニュアルの 4.3.2 章のリセット例外処理が開始され、0～3 番地に記述されたアドレスからプログラムが開始されます。

vectors[]の最初の要素が 0～3 番地に書き込まれるよう sysinit.c でプログラムしています。（sysinit.c の 24 行目の__attribute__ ((section (".vectors")))がこの配列を.vectors セクションに割り当てており、.vectors セクションはリンカスクリプトで０～200 番地に配置するよう指定しているためです）vectors の 1 番目の要素には、start 関数の番地格納されています。
start 関数は、CRT0_IN.S でアセンブリ言語で記述されています。_start と表現されているシンボルが start 関数の開始位置です。

以上の仕組みにより、当キットのマイコンに電源を入れると CRT0_IN.S の_start からプログラムが実行されます。

CRT0_IN.S を見ると sysinit()を呼び出した後、main()を実行していることが分かります。sysinit()の機能は、bss と data に定義した変数の初期化です。

当キットのスタートアップルーチンは、CRT0_IN.S と sysinit.c を使ってください。

【CRT0_IN.S】

```
1 :             .h8300s
2 :             .section .text
3 :             .global         _start
4 : _start:
5 :             mov.l           #_stack_internal,sp             ; スタックの初期化
6 :             jsr             @__sysinit                      ; システムの初期化（セクション初期化）
7 :             mov.l           #_stack_internal,sp             ; スタックの初期化
8 :             jsr             @_main
9 : loop:
10 :            bra             loop
11 :            .end
```

【sysinit.c】

```
1 : extern void start(void);
2 :
3 : extern    unsigned char *    bss_begin , bss_end ;
4 : extern    unsigned char * handler_begin ,data_end;
5 :
6 : /* 割り込みベクタテーブル */
7 : typedef unsigned long   fp;
8 : #define                 FLASH_ROM_RAM2_START                    0x200UL
9 :
10 : void _sysinit(void)
11 : {
12 :            unsigned char *dst,*src;
13 :            /* 未初期化 .bss セクション初期化 */
14 :            for( dst = (unsigned char *)&bss_begin ; dst < (unsigned char *)&bss_end ; dst ++ )  { *dst = 0 ; }
15 :
16 :            // 初期化済み変数の値を ROM から RAM にコピーする
17 :            for( dst = (unsigned char *)&handler_begin ; dst<= (unsigned char *)&data_end ; dst++)
18 :            {
19 :                            src=dst-(unsigned long)&handler_begin+FLASH_ROM_RAM2_START;
20 :                            *dst=*src;
21 :            }
22 : }
23 :
24 : fp vectors[] __attribute__ ((section (".vectors")))={
25 :     (fp)start               ,(fp)start          ,(fp)start   ,(fp)start,
26 :     (fp)start               ,(fp)start          ,(fp)start   ,(fp)start,
27 :     (fp)start               ,(fp)start          ,(fp)start   ,(fp)start,
28 :     (fp)start               ,(fp)start          ,(fp)start   ,(fp)start,
29 :
30 :     (fp)start               ,(fp)start          ,(fp)start   ,(fp)start,
31 :     (fp)start               ,(fp)start          ,(fp)start   ,(fp)start,
32 :     (fp)start               ,(fp)start          ,(fp)start   ,(fp)start,
33 :     (fp)start               ,(fp)start          ,(fp)start   ,(fp)start,
34 :
35 :     (fp)start               ,(fp)start          ,(fp)start   ,(fp)start,
36 :     (fp)start               ,(fp)start          ,(fp)start   ,(fp)start,
37 :     (fp)start               ,(fp)start          ,(fp)start   ,(fp)start,
```

```
38 :     (fp)start                ,(fp)start       ,(fp)start   ,(fp)start,
39 :
40 :     (fp)start                ,(fp)start       ,(fp)start   ,(fp)start,
41 :     (fp)start                ,(fp)start       ,(fp)start   ,(fp)start,
42 :     (fp)start                ,(fp)start       ,(fp)start   ,(fp)start,
43 : };
44 :
```

８-２-３ユーザープログラム

EE_LIB を使ってプログラムするときは、PortG の１ビット目などのように指定していましたが、今回はハードウエア制御用のレジスタを指定して IO ポートを操作します。

H8S/2238 のハードウエアマニュアルの 10.12 にポート G を制御するためのレジスタが書いてあります。

H8S の IO ポートへの出力には、下記の２つレジスタを使います。

・ディレクションレジスタ　IO ポートを出力に使うか、入力に使うかを指定します。

・データレジスタ　出力に使う場合、IO ポートに出力する状態を書き込みます。

　　入力に使う場合、IO ポートの状態を読み取ります。

ハードウエアマニュアルの 10.12 章および 26 章に一通り目を通してください。

PG0 を出力に指定するには、FE3F 番地を１に設定すればよいことが分かると思います。

PG0 の状態を HIGH にするには、FF0F 番地の LSB（最下位ビット）を１にすればよいことが分かります。

PG0 の状態を LOW にするには、FF0F 番地の LSB（最下位ビット）を０にすればよいことが分かります。

このことを踏まえて下記のプログラムを見ると、LED を点滅させていることが分かると思います。FE3F や FF0F が FF FE3F、FFFF0F と表現されているのは、CPU のモードと関係します。当キットは H8S/2238 をアドバンスモードで使用するため、レジスタを指定するときに上位に FF をつけて、24 ビットに拡張して指定します。

```
 1 : #define PGDDR_ADDR ((volatile unsigned char *)0xfffe3fUL)
 2 : #define PGDDR      (*PGDDR_ADDR)
 3 : #define PGDR_ADDR  ((volatile unsigned char *)0xffff0fUL)
 4 : #define PGDR       (*PGDR_ADDR)
 5 :
 6 : void wait()
 7 : {
 8 :         volatile int a,b,c,ii,jj;
 9 :         for(a=10,b=20,jj=0; jj<7; jj++) {for(ii=0;ii<5000; ii++) {c=c*a+b;}}
10 : }
11 :
12 : main()
13 : {
14 :         PGDDR=0x1;                  // PG0 を出力ポートに設定する
15 :         PGDR=0;                     // PG0 の状態を LOW に設定する
16 :         while(-1){
17 :                         PGDR |= 0x1; // PG0 の状態を HIGH に設定する
18 :                         wait();
19 :                         PGDR &=(~1); // PG0 の状態を LOW に設定する
20 :                         wait();
21 :         }
22 :         return 0;
23 : }
```

８-２-４make ファイル

make コマンドを使って実行モジュールを生成していたと思います。make にプログラムを構成するファイルやコンパイル手順を指示するのが makefile です。
makefile の文法については、GNU　Make のマニュアルを参照してください。
別のプログラムを作るときに、makefile の修正が必須の箇所は２箇所です。

- APPLICATION　　生成する mot ファイルの名称を指定します。
- OBJECTS　　実行モジュールを生成するために必要なソースファイルをスペースで区切って羅列します。
  このときに拡張子は.o を指定してください。
  スタートアップルーチンも含めて記述します。

【makefile】

```
 1 : INCLUDE_OPT += -I./
 2 :
 3 : # ************ Target define **************
 4 : APPLICATION          = LED
 5 : OBJECTS              = CRT0_IN.o sysinit.o main.o
 6 : LDSCRIPT             = H8S_2238_INTERNAL_ROM.x
 7 :
 8 : # *********** Commands ********************
 9 : PREFIX               := h8300-elf
10 : CPP                  = $(PREFIX)-g++
11 : CC                   = $(PREFIX)-gcc
12 : LD                   = $(PREFIX)-ld
13 : OBJCOPY                        = $(PREFIX)-objcopy
14 : STRIP                = $(PREFIX)-strip
15 : AR                   = $(PREFIX)-ar
16 :
17 : # *********** PATH ************************
18 : GCC_VERSION          = 4.1.2
19 : LIBPATH1 = /usr/local/cross/h8300-elf/lib
20 : LIBPATH11 = $(LIBPATH1)/h8300s
21 : LIBPATH12 = $(LIBPATH1)/h8300s/normal
22 : LIBPATH13 = $(LIBPATH1)/h8300s/int32
23 : LIBPATH14 = $(LIBPATH1)/h8300s/normal/int32
24 :
25 : LIBPATH2 = /usr/local/cross/lib/gcc/h8300-elf/$(GCC_VERSION)
26 : LIBPATH21 = $(LIBPATH2)/h8300s
27 : LIBPATH22 = $(LIBPATH2)/h8300s/normal
28 : LIBPATH23 = $(LIBPATH2)/h8300s/int32
29 : LIBPATH24 = $(LIBPATH2)/h8300s/normal/int32
30 :
31 : # ************ options *********************
32 : LIBDIR                 = -L$(LIBPATH1) ¥
33 :            -L$(LIBPATH1) ¥
34 :            -L$(LIBPATH11) ¥
35 :            -L$(LIBPATH12) ¥
36 :            -L$(LIBPATH13) ¥
37 :            -L$(LIBPATH14) ¥
38 :            -L$(LIBPATH21) ¥
39 :            -L$(LIBPATH22) ¥
40 :            -L$(LIBPATH23) ¥
41 :            -L$(LIBPATH24)
42 :
43 : # *********** FLAGS *****************
44 : CFLAGS = -O3 -ms -mrelax -mno-s2600 -Wall
45 : LDFLAGS                = -nostartfiles -Wl,--strip-all $(LIBDIR) -Wl,-Map,$(APPLICATION).map
46 :
47 : # *********** Dependance *****************
48 : $(APPLICATION).mot : $(APPLICATION).abs
49 :            $(OBJCOPY) -O srec $(APPLICATION).abs $(APPLICATION).mot
50 :
51 : $(APPLICATION).abs : $(OBJECTS)
52 :            $(CPP) -o $(APPLICATION).abs $(LDFLAGS) -T$(LDSCRIPT) $^
53 :
54 : # ****************** clean **********************
55 : .PHONY:clean
56 : clean:
57 :            rm *.abs *.mot *.o *.map
58 :
59 : # *********** Suffix rule ****************
60 : .SUFFIXES:
61 : .SUFFIXES: .o .s .c .cpp .S
62 :
63 : .c.o:
```

```
64 :            $(CC) -c $(CFLAGS) $<
65 :
66 : .cpp.o:
67 :            $(CPP) -c $(CFLAGS) $<
68 :
69 : .s.o:
70 :            $(CC) -c $(CFLAGS) $<
71 :
72 : .S.o:
73 :            $(CC) -c $(CFLAGS) $<
```

9　EE_LIB リファレンスマニュアル

9-1 クラス一覧

EE_LIB がサポートするクラスの一覧です。

インクルードの列は、そのクラスを使用する場合にインクルードする必要のあるヘッダファイルです。

デストラクト可否の列が否と書かれたクラスは、クラスの解放ができません。これは、電源投入時に作成し電源を切るまで機能し続けることを期待される用途と想定されるクラスです。ROM 容量削減のためデストラクタを実装していないのです。

パッケージ	サブパッケージ	クラス	説明	インクルード	抽象クラス	デストラクト可否
HAL	Communication	SCI	SCI を使って 1 文字単位で送受信を行う。	EE_LIB/HAL/Communication/SCI.h	○	否
		SCIUsingInterrupt	SCI を使って 1 文字単位で送受信を行う。 このクラスのインスタンスを作成し、クラスを使用するときは、SCI クラスを使ってこのクラスにアクセスするとよい。（DMA をサポートする CPU で SCIUsingDMA タイプを利用する場合は、クラスのインスタンス生成部のみを修正するとプログラムがそのまま使えるため）	EE_LIB/HAL/Communication/SCIUsingInterrupt.h	－	否
	IO	ADC	ADC を使って AD 変換を行う。	EE_LIB/HAL/IO/ADC.h	－	否
		IOpin	IO ポートを使って入出力する。	EE_LIB/HAL/IO/IOPinNo.h EE_LIB/HAL/IO/IOPortNo.h EE_LIB/HAL/IO/IOPin.h	－	否
	Interrupt	CriticalSection	タスクスイッチや割り込みを禁止する。タスクスイッチを禁止したい間、このクラスを生成する。	EE_LIB/HAL/Interrupt/CriticalSection.h	－	可
	IntervalTimer	IntervalTimer	周期的に割り込みを発生させる。 割り込みにより TimerRegisterable クラスの Perform メソッドを呼び出す。	EE_LIB/HAL/IntervalTimer/IntervalTimer.h	○	否
		PWM	PWM を行う。	EE_LIB/HAL/IntervalTimer/PWM.h	○	否
		Timer16Bit	16 ビットタイマを使って以下のことを行う。 PWM モード：PWM を行う。	EE_LIB/HAL/IntervalTimer/Timer16Bit.h	－	否

			インターバルタイマモード : 周期的に割り込みを発生させる。	EE_LIB/HAL/Interrupt/Priority.h		
		TimerRegisterable	割り込みハンドラを実装する。 このクラスを継承したクラスの Perform メソッドに割り込みハンドラを実装する。	EE_LIB/HAL/IntervalTimer/TimerRegisterable.h	○	否
		TimerWD	TimerWD を使って周期的に割り込みを発生させる。	EE_LIB/HAL/IntervalTimer/TimerWD.h EE_LIB/HAL/Interrupt/Priority.h	－	否
OSWrapper	Task	CyclicTask	周期実行タスクを実現する仕組み。	EE_LIB/OSWrapper/Task/CyclicTask.h	－	否
		CyclicTaskPerformer	周期タスクの処理を実装する。 このクラスを継承したクラスを作成し、perform メソッドの中に周期処理を実装する。	EE_LIB/OSWrapper/Task/CyclicTaskPerformer.h	○	否
		OSTimer	周期実行タスクを管理方法を指定する。 周期実行タスクの実行周期の分解能とタスクを実行に使用する割り込みのプライオリティおよび使用する CPU のタイマを定義する。	EE_LIB/OSWrapper/Task/OSTimer.h	－	否
		TaskFactory	周期実行タスクを生成する。	EE_LIB/OSWrapper/Task/TaskFactory.h EE_LIB/OSWrapper/Task/Task.h	－	可
Unit	Communication	Com	指定した SCI を使ってバイト列を送受信する	EE_LIB/Unit/Communication/Com.h	－	否
	Sound	Sound	楽譜を文字列で定義する	EE_LIB/Unit/Sound/Sound.h	－	可
		SoundPlayer	Sound クラスを再生する	EE_LIB/Unit/Sound/SoundPlayer.h	○	否
		SoundPlayerUsingPWM	PWM タイマを使って Sound クラスを再生する	EE_LIB/Unit/Sound/SoundPlayerUsingPWM	－	否
Util	CommandInterpreter	Command	コマンドを定義する。 このクラスを継承したクラスに、コマンドのフォーマットの検査方法とコマンドの処理を実装する。	EE_LIB/Util/CommandInterpreter/Command.h	○	否
		CommandInterpreter	TeraTerm などの通信ソフトを使って１行送信されるとごに、その文字列のフォーマットを分析し、期待していたフォーマットだった場合に特定の処理う。	EE_LIB/Util/CommandInterpreter/CommandInterpreter.h	－	否
		ExceptionCommand	CommandInterpreter のコンストラクタに指定する ExceptionCommand クラスのデフォルトのクラス。 受信した文字列がどのコマンドのフォーマットでもない場合、受信した文字列を開放する処理だけを実装したクラス。	EE_LIB/Util/CommandInterpreter/ExceptionCommand.h	－	否
	Primitive	Bytes	文字列やバイト列を格納するクラス	EE_LIB/Util/Primitive/Bytes.h	－	可
		Format	数値や文字列を書式にしたがって編集した文字列を格納するクラス	EE_LIB/Util/Primitive/Format.h	－	可

９-２ HAL

９-２-１ Communication

9-2-1-1 SCI

操作	ノート	パラメータ
Public void **echoOff()**	エコーを無効にする	
Public void **echoOn()**	エコーを有効にす。SCI を生成したときには、エコーは有効となっている。	
Public bool Pure **receive()**	1byte 受信する。受信したデータがない場合は、false を返す return true 成功 false 失敗	[in] unsigned char* **data** 受信データ
Public **SCI()**	コンストラクタ SCI0、1 どちらを使用するかを SCIId に指定し、cp で以下の通信条件を指定する。 通信方式 (0:調歩同期式 , 1:クロック同期式) ボーレート(bps) データ長 (1:7bit , 0:8bit) ストップビット(0:1bit , 1:2bit) パリティ(0:なし , 1:奇数 ,2:偶数) マルチプロセッサ (0:マルチプロセッサ無効 1:マルチプロセッサ有効)	[in] EE_LIB::HAL::Communication::SCI::SCIId **SCIId** SCI の ID [in] EE_LIB::HAL::Communication::SCI::CommunicationParam& **cp** 通信パラメータ
Public **~SCI()**	デストラクタ 実装していない	なし
Public bool Pure **send()**	1byte 送信する。送信バッファフル時は、false を返す。 return true 成功 false 失敗	[in] unsigned char **data** 送信データ

9-2-1-2 SCIUsingInterrupt

操作	ノート	パラメータ
Public bool **receive()**	1byte 受信する。受信したデータがない場合は、false を返す return true 成功 false 失敗	[in] unsigned char* **data** 受信データ
Public **SCIUsingInterrupt()**	コンストラクタ SCI0、1 どちらを使用するかを SCIId に指定し、cp で以下の通信条件を指定する。 通信方式 (0:調歩同期式 , 1:クロック同期式) ボーレート(bps) データ長 (1:7bit , 0:8bit) ストップビット(0:1bit , 1:2bit) パリティ(0:なし , 1:奇数 ,2:偶数) マルチプロセッサ (0:マルチプロセッサ無効 1:マルチプロセッサ有効)	[in] EE_LIB::HAL::Communication::SCI::SCIId **SCIId** SCI の ID [in] EE_LIB::HAL::Communication::SCI::CommunicationParam& **cp** 通信パラメータ [in] unsigned int **receptionBufferSize** 受信バッファのサイズ(byte) オプション（初期値：48byte） [in] unsigned int **transmissionBufferSize** 送信バッファのサイズ(byte) オプション

操作	ノート	パラメータ
		（初期値：8byte）
Public **~SCIUsingInterrupt()**	デストラクタ 実装していない	なし
Public bool **send()**	1byte 送信する。送信バッファフル時は、false を返す。 return true 成功 false 失敗	[in] unsigned char **data** 送信データ

9-2-2IO

9-2-2-1 ADC

操作	ノート	パラメータ
Public **ADC()**	コンストラクタ	なし
Public **~ADC()**	デストラクタ 実装していない。	なし
Public unsigned short **get()**	AD変換後のデータを取得する	[in] unsigned char **c** チャネル No （0～7）
Public void **start()**	スキャン対象のチャネルの AD 変換結果を、クラス内に取り込む。取り込んだ値は、getData で取り出す。	[in] EE_LIB::HAL::IO::ADC::Mode **mode** M_SINGLE : 指定したチャネルの AD 変換を行う M_SCAN: c で指定したスキャン対象のチャネルを 1 回ずつ AD 変換する。 [in] unsigned char **c** M_SINGLE のとき、変換するチャネルを指定する。（0～7 は、AN0～AN7 に対応する） M_SCAN:c の数値により下記のようにスキャン対象を指定する。※ 外部トリガのとき 4 以上は指定不可 0: AN0 1: AN0,AN1 2: AN0-AN2 3: AN0-AN3 4: AN0-AN4 5: AN0-AN5 6: AN0-AN6 7: AN0-AN7 [in] EE_LIB::HAL::IO::ADC::Trig **trig** トリガによる AD 変換が必要なときに、トリガ要因を指定する（オプション） TRIG_T8：8bit タイマ割り込み TRIG_EXT：ADTrig 端子

9-2-2-2 IOPin

操作	ノート	パラメータ

操作	ノート	パラメータ
Public unsigned char **get()**	IO ピンの状態を取得する Hi：1　Low：0　（polarity の影響は受けない）	なし
Public **IOPin()**	コンストラクタ ポートNo、ピンNo、入出力を指定してインスタンスを生成する。	[in] IOPortNo **portNo** IO ポート No [in] IOPinNo **pinNo** IO ピン No [in] Direction **dir** ０：入力　０以外：出力
Public **~IOPin()**	デストラクタ 実装していない	なし
Public void **set()**	IO ピンの状態を設定する	[in] unsigned char **data** polarity＝０のとき 0:Low 0 以外：Hi polarity が０以外のとき 0:HI 0 以外：LOW を出力する。
Public void **setDirection()**	IO ピンの入力/出力を切り替える	[in] Direction **dir** ０：入力　０以外：出力
Public void **setPolarity()**	極性を設定する。	[in] unsigned char **polarity** 正論理のとき０、負論理のとき１を設定する。 インスタンス生成時は、正論理になっている。

9-2-3 Interrupt

9-2-3-1 CriticalSection

操作	ノート	パラメータ
Public **CriticalSection()**	コンストラクタ このクラスのインスタンスを生成している間は、タスクスイッチや割り込みが禁止される。 割り込みハンドラやタスクの処理中に、２つの変数を不可分に書き換えたい場合は、このクラスのインスタンスを生成するとよい。	なし
Public **~CriticalSection()**	デストラクタ このクラスを生成した時点で、割り込み禁止となっている場合は、このオブジェクト破棄後も割り込み禁止を継続する。	なし

9-2-4 IntervalTimer

9-2-4-1 IntervalTimer

操作	ノート	パラメータ
Public **IntervalTimer()**	コンストラクタ	なし
Public **~IntervalTimer()**	デストラクタ	なし
Public void	インターバルを変更する。	[in]

操作	ノート	パラメータ
Pure **setInterval()**	@return なし	unsigned long **interval** インターバル(ns) ※ HW の制限で設定できない場合、interval で指定した時間に最も近い時間を採用する。
Public void Pure **start()**	カウントを開始する。	なし
Public void Pure **stop()**	カウントを停止する。	なし

9-2-4-2 PWM

操作	ノート	パラメータ
Public **PWM()**	コンストラクタ	なし
Public **~PWM()**	デストラクタ	なし
Public void Pure **setDuty()**	デューティ比を変更する。	[in] unsigned short **duty** デューティ比(%)を 100 倍した値（10000-0 10000 のときすべて１、0 のときすべて０を出力する）
Public void Pure **setDuty()**	デューティ比を変更する。	[in] unsigned short **duty** デューティ比(%)を 100 倍した値（10000-0 10000 のときすべて１、0 のときすべて０を出力する） [in] unsigned long **interval** インターバル(ns) ※ HW の制限で設定できない場合、interval で指定した時間に最も近い時間を採用する。
Public void Pure **start()**	PWM 出力を再開する。PWM モードで周期数を制限した場合のみ有効。	なし

9-2-4-3 Timer16Bit

操作	ノート	パラメータ
Public void **setDuty()**	デューティ比を変更する。（PWM モード時のみ使用可）	[in] unsigned short **duty** デューティ比(%)を 100 倍した値（10000-0）
Public void **setDuty()**	デューティ比を変更する。（PWM モード時のみ使用可）	[in] unsigned short **duty** デューティ比(%)を 100 倍した値（10000-0） [in] unsigned long **interval** インターバル(ns) ※ HW の制限で設定できない場合、interval より一段大きい設定可能な時間をインターバルとする。
Public void **setInterval()**	インターバルを変更する。	[in] unsigned long **interval**

操作	ノート	パラメータ
		インターバル(ns) ※ HW の制限で設定できない場合、interval で指定した時間に最も近い時間を採用する。
Public void **start()**	インターバルタイマモードのとき、カウントを開始する。 PWM モードのとき、PWM 出力を再開する。PWM モードで周期数を制限した場合のみ有効。	なし
Public void **stop()**	カウンタを停止する。	なし
Public **Timer16Bit()**	インターバルタイマモードで初期化する。HW の制限で設定できない場合、interval より一段大きい設定可能な時間をインターバルとする。 TimerRegisterable の perform()メソッドを指定した interval で呼び出す。	[in] TimerId **timerId** タイマーID [in] unsigned long **interval** インターバル(ns) [in] EE_LIB::HAL::Interrupt::Priority **ip** プライオリティ [in] TimerRegisterable* **timer** TimerRegisterable へのポインタ [in] bool bStart true のとき、生成時からカウンタをスタートさせる。 (オプション デフォルト：true)
Public **~Timer8Bit()**	デストラクタ 実装していない	なし
Public **Timer16Bit()**	PWM モードで初期化する。interval で指定した時間に最も近い時間を採用する。 dutyで指定する期間は、周期の後半部分の時間である。 ↓ Dutyで指定する |　　|~~~~| |　　|　　| |____|　　|	[in] TimerId **timerId** タイマーID [in] unsigned long **interval** インターバル(ns) [in] unsigned short **duty** デューティ比(%)を 100 倍した値（10000-0）

9-2-4-4 TimerRegisterable

操作	ノート	パラメータ
Public void Pure **init()**	IntervalTimerクラスのインスタンスを生成したタイミングで１回だけ呼び出される。 引数で渡される情報をメンバー変数に保存するなど、クラスの初期化を行うのが主な用途である。	[in] unsigned long **interval** perform()を呼び出す周期(ns) [in] EE_LIB::HAL::Interrupt::Priority **ip** 割り込みの優先度(ns) [in]

操作	ノート	パラメータ
		IntervalTimer* **it** インターバルタイマへのポインタ
Public void Pure **perform()**	Timerクラスから周期的に呼び出される。 ユーザーは周期実行したい処理をこのクラスの子クラスにこのメソッドをオーバーライドして実装する。	なし
Public **TimerRegisterable()**	コンストラクタ	なし
Public **~TimerRegisterable()**	デストラクタ	なし

9 - 2 - 4 - 5 TimerWD

操作	ノート	パラメータ
Public void **setInterval()**	インターバルを変更する。	[in] unsigned long **interval** インターバル(ns) ※ HW の制限で設定できない場合、interval で指定した時間に最も近い時間を採用する。
Public void **start()**	カウントを開始する。	なし
Public void **stop()**	カウントを停止する。	なし
Public **TimerWD()**	インターバルタイマを初期化する。interval で指定した時間に最も近い時間をインターバルを採用する。	[in] TimerId **timerId** タイマーID [in] unsigned long **interval** インターバル(ns) [in] EE_LIB::HAL::Interrupt::Priority **ip** プライオリティ [in] TimerRegisterable* **timer** TimerRegisterable へのポインタ [in] bool bStart true のとき、生成時からカウンタをスタートさせる。(オプション デフォルト：true)
Public **~TimerWD()**	デストラクタ 実装していない	なし

9 - 3 OSWrapper

9 - 3 - 1 Task

9 - 3 - 1 - 1 CyclicTask

操作	ノート	パラメータ
Public **CyclicTask()**	コンストラクタ	[in] EE_LIB::OSWrapper::Task::CyclicTaskPerformer* **pPerformer** CyclicTaskPerformer へのポインタ

操作	ノート	パラメータ
		[in] unsigned short **interval** タスクの起動間隔(ms) 有効範囲：1 から 4294 [in] unsigned short **priority** タスクプライオリティ 小さい方が優先度が高い
Public **~CyclicTask()**	デストラクタ 実装していない	なし
Public unsigned short **getPriority()**	プライオリティの取得 return プライオリティ 小さい方が優先度が高い	なし
Public void **perform()**	OSTimer クラスから周期的に呼び出される。 return なし	なし
Public void **setInterval()**	perform を呼び出す周期の設定（OSTimer が addTask 時に呼び出す） return なし	[in] unsigned long **interval** perform()を呼び出す周期(ns)

9-3-1-2 CyclicTaskPerformer

操作	ノート	パラメータ
Public **CyclicTaskPerformer()**	コンストラクタ	なし
Public **~CyclicTaskPerformer()**	デストラクタ 実装していない	なし
Public void Pure **perform()**	CyclicTaskr クラスから周期的に呼び出される。 return なし	[in] unsigned short **interval** perform()を呼び出す周期(ms)

9-3-1-3 OSTimer

操作	ノート	パラメータ
Public bool **addTask()**	タスクを追加する。 追加後、m_interval 周期で OS タイマはタスクの perform を呼び出す return 成功-true 失敗-false	[in] EE_LIB::OSWrapper::Task::Task* **task** task
Public void **init()**	周期実行タスクの実行周期の分解能とタスクを実行に使用する割り込みのプライオリティおよび使用する CPU のタイマを定義する。 return なし	[in] unsigned long **interval** 周期実行タスクの実行周期の分解能(ns) [in] EE_LIB::HAL::Interrupt::Priority **ip** 割り込みの優先度(ns) [in] EE_LIB::HAL::IntervalTimer::IntervalTimer* **it** インターバルタイマへのポインタ
Public **OSTimer()**	コンストラクタ	なし

操作	ノート	パラメータ
Public **~OSTimer()**	デストラクタ 実装していない	なし
Public void **perform()**	Timer クラスから周期的に呼び出される。 return なし	なし

9 - 3 - 1 - 4 TaskFactory

操作	ノート	パラメータ
Public void **createTask()**	周期実行タスクを生成する。 return なし 周期実行する処理は `CyclicTaskPerformer` を継承したクラスのperformメソッドに実装する。	[in] EE_LIB::OSWrapper::Task::CyclicTaskPerformer* **performer** CyclicTaskPerformer へのポインタ [in] unsigned short **interval** perform()を呼び出す周期(ms) [in] unsigned short **priority** [in] EE_LIB::OSWrapper::Task::OSTimer* **osTimer** osTimer へのポインタ
Public **TaskFactory()**	コンストラクタ	なし
Public **~TaskFactory()**	デストラクタ	なし

9 - 4 Unit

9 - 4 - 1 Communication

9 - 4 - 1 - 1 Com

操作	ノート	パラメータ
Public **Com()**	指定した SCI を使ってバイト列を送受信する通信ポートを生成する。	[in] EE_LIB::HAL::Communication::SCI* **sci** SCI クラスへのポインタ [in] EE_LIB::OSWrapper::Task::OSTimer* **osTimer** OSTimer クラスへのポインタ [in] unsigned int **receptionQueueNumber** 受信キュー件数（オプション） [in] unsigned int **receptionBufferSize** 受信バッファのサイズ（byte）（オプション） [in] unsigned int **transmittionQueueNumber** 送信キューの件数（オプション） [in] unsigned long **cycle** 送受信タスクの実行周期(ms) （オプション）

操作	ノート	パラメータ
Public **~Com()**	デストラクタ 実装していない	なし
Public void **DebugPrint()**	最後に生成した Com ポートに message で指定した文字列を送信する。Com を生成していない段階では、文字列の出力は行わない。 return なし	[in] char* **message** 文字列へのポインタ message の開放はユーザが行う
Public void **DebugPrint()**	最後に生成した Com ポートに message で指定した文字列を送信する。Com を生成していない段階では、文字列の出力は行わない。 return なし	[in] EE_LIB::Util::Primitive::Format* **format** フォーマット付き文字列へのポインタ format の開放はユーザが行う
Public void **echoOff()**	エコーを無効にする return なし	なし
Public void **echoOn()**	エコーを有効にする。com を生成したときには、エコーは有効となっている。 return なし	なし
Public Com* **getInstance()**	Debug Print 用に com ポートを取得する。com ポートを生成するわけではなく、すでに生成済みの最後に生成した com ポートへのポインタを返す return 最後に生成した Com ポートへのポインタ。1 つも Com ポートを生成していない場合、Null を返す。	なし
Public EE_LIB::Util::Primitive::Bytes* **receive()**	メッセージを受信する。 return 受信したメッセージへのポインタを返す。メッセージがない場合、NULL を返す。	なし
Public void **releaseReceptionBuffer()**	受信バッファを再利用する。 受信したメッセージは、このメソッドを使って com サブシステム内で再利用することが可能である。 このメソッドを使って再利用しない場合、send を使って送信するか、または、ユーザーが delete しなければならない。 return なし	[in] EE_LIB::Util::Primitive::Bytes* **pReceptionBuffer** 受信した文字列へのポインタ
Public void **send()**	通信ポートに文字列を送信する return なし	[in] char* **message** 文字列へのポインタ message の開放はユーザが行う [in] bool **withNewLine** true を指定すると最後に改行をつける
Public void **send()**	通信ポートに文字列を送信する return なし	[in] EE_LIB::Util::Primitive::Format* **format** フォーマット付き文字列へのポインタ format の開放はユーザが行う

操作	ノート	パラメータ
		[in] bool **withNewLine** true を指定すると最後に改行をつける
Public void **send()**	通信ポートに文字列を送信する return なし	[in] EE_LIB::Util::Primitive::Bytes* **message** 文字列へのポインタを指定する [in] bool **withNewLine** true を指定すると最後に改行をつける ※注意！引数に Bytes を用いるこのバージョンの send は、 message の delete は com サブシステムが行うため、ユーザーは Delete しないこと （通信のパフォーマンス向上のため）
Public void **sendNewLine()**	改行を送信する return なし	なし
Public void **sendPrompt()**	プロンプトを送信する return なし	なし
Public void **setNewLine()**	改行文字を設定する return なし	[in] unsigned char **newLine** 改行文字の文字コードを指定する
Public void **setPrompt()**	プロンプトを設定する return なし	[in] char* **msg** プロンプトとして表示する文字列へのポインタを指定する
Public void **setTransmitNewLine()**	改行文字を設定する return なし	[in] EE_LIB::Util::Primitive::Bytes& **newLine**

9-4-2 Sound

9-4-2-1 Sound

操作	ノート	パラメータ
Public bool **getNext()**	次の音符(またはコマンド）を取得する return 取得成功:true 取得失敗：false（楽譜終わり）	[in] unsigned char& **kind** NOP,VOLUME,LENGTH, 音 符 (0-119) [in] unsigned long& **value** kind が音量のとき音量、kind が音の長さのとき長さ(ms) kind が左記以外の場合 0 固定
Public void **seekFirst()**	音符の先頭から再生するよう初期化する。(同じ Sound オブジェクトを使いまわすときに使う） return なし	なし
Public **Sound()**	コンストラクタ 【楽譜のフォーマット】 ○音程	[in] char* **pSound** 楽譜を定義した文字列

操作	ノート	パラメータ
	ドレミファソラシ → CDEFGAB 半音上げるときは#をつける C#→ド# オクターブ 音程の後に 0～9 をつけて表現する 数字が大きいほど高い音を表す ○テンポ T の後に 16 分音符が 1 分間に入る数を指定する ○音の長さ L の後に音符の種類を指定する。L4 であれば 4 部音符。付点は L4H のように後ろに H をつける。 ○音量 V の後に 0～255 の数字をつけて音の大きさを表す。数字が大きいほど音が大きい。0 は消音。（H8S/2238 では音量のコントロールは働かない。0 が消音、それ以外は音を出す。） ※　音量と音の長さは、一旦指定するとその後に続く音程に引き継がれる。 ○休符 ピリオード.で表す。 ○区切り文字 記号と記号の間の_は無視。記号とパラメータの間の_は NG（L_8 などは NG）。 例）音量200でT=640で4分音符のドレミファソラシドを表現する文字列 ： T640V200L4C4D4E4F4G4A4B4C5 return なし	
Public **~Sound()**	デストラクタ	なし

9-4-2-2 SoundPlayer

操作	ノート	パラメータ
Public bool **getStatus()**	player の状態を取得する return true:再生中 false:待機中	なし
Public void Pure **setSound()**	Sound クラスを再生する return なし	[in] EE_LIB::Unit::Sound::Sound* **pSound** 音へのポインタ [in] unsigned short **playCount** 再生回数 65535 を指定するとずっと繰り返す
Public **SoundPlayer()**	コンストラクタ	なし
Public **~SoundPlayer()**	デストラクタ 実装していない	なし

9-4-2-3 SoundPlayerUsingPWM

操作	ノート	パラメータ
Public void **setSound()**	Sound クラスを再生する return なし	[in] EE_LIB::Unit::Sound::Sound* **pSound** 音へのポインタ [in] unsigned short **playCount** 再生回数 65535 を指定するとずっと繰り返す
Public **SoundPlayerUsingPWM()**	コンストラクタ 使用するPWMを指定する。	[in] EE_LIB::HAL::IntervalTimer::PWM* **pPWM** PWM タイマーへのポインタ
Public **~SoundPlayerUsingPWM()**	デストラクタ 実装していない	なし

9 - 5 Util

9 - 5 - 1 CommandInterpreter

9 - 5 - 1 - 1 Command

操作	ノート	パラメータ
Public bool Pure **check()**	引数の Command が、このクラスで処理すべきコマンドかをチェックする。(このクラスを継承したクラスのこのメソッドでチェックを行う。) CommandInterpreter クラスに登録した後、CommandInterpreter クラスが１行受信するたびに、その文字列を引数 Command に指定してこのメソッドを呼び出す。 Command で渡された文字列がこのクラスで処理するコマンドのフォーマットに合致していれば True を返す。 CommandInterpreter は、このメソッドの戻り値が True の場合に、そのクラスの execute メソッドを呼び出す。 TeraTerm などの通信ソフトを使って１行送信されるとごに、その文字列のフォーマットを分析し、期待していたフォーマットだった場合に特定の処理を行うときに CommandInterpreter クラスと組み合わせて当クラスを利用する。 return このクラスで処理すべきコマンドのとき Trueを返す。	[in] EE_LIB::Util::Primitive::Bytes* **command** コマンド文字列へのポインタ
Public **Command()**	コンストラクタ	なし
Public **~Command()**	デストラクタ 実装していない	なし
Public void Pure **execute()**	コマンドの処理内容をこのメソッドに記述する。(このクラスを継承したクラスのこのメソッドでコマンドを処理する。) このメソッドで最低限行わなくてはならないことは、**command** で指定された文字列の開放である。解放の方法には下記の３種がある。 １） Com クラスの releaseReceptionBuffer メソッドを使って受信バッファとして再利用する。 ２） Com クラスの Send メソッドを使い Command で指定した文字列を送信する。 ３） delete により直接開放する。	[in] EE_LIB::Unit::Communication::Com* **com** 通信ポートへのポインタ [in] EE_LIB::Util::Primitive::Bytes* **command** コマンド文字列へのポインタ (**check** メソッドに指定された文字列と同じ文字列)

9 - 5 - 1 - 2 CommandInterpreter

操作	ノート	パラメータ
Public bool **addCommand()**	コマンドを追加する return true 成功 false 失敗	[in] EE_LIB::Util::CommandInterpreter::Command * **command** 追加するコマンドを指定する

操作	ノート	パラメータ
Public **CommandInterpreter()**	コンストラクタ return なし	[in] EE_LIB::Unit::Communication::Com* **com** 通信ポートへのポインタ [in] EE_LIB::Util::CommandInterpreter::Command * **exception** 登録したどのコマンドでもない場合、呼び出されるコマンドを指定する。このコマンドは、最低限受信バッファの開放をする必要がある。 [in] char* **prompt** プロンプトとして表示する文字列へのポインタ [in] char* **msg** CommandInterpreter 初期化完了時に表示する文字列へのポインタ
Public **~CommandInterpreter()**	デストラクタ 実装していない	なし
Public void **sendMessage()**	文字列を表示する return なし	[in] char* **msg** 文字列へのポインタ [in] bool **withNewLine** true-改行文字を最後に出力する false-改行文字を最後に出力しない
Public void **setPrompt()**	プロンプトを設定する return なし	[in] char* **msg** プロンプトへのポインタを指定する。

9-5-1-3 ExceptionCommand

操作	ノート	パラメータ
Public bool **check()**	何も処理しない。	[in] EE_LIB::Util::Primitive::Bytes* **command** command
Public **ExceptionCommand()**	コンストラクタ	なし
Public **~ExceptionCommand()**	デストラクタ 実装していない	なし
Public void **execute()**	**command** で指定した文字列を開放する。	[in] EE_LIB::Unit::Communication::Com* **com** [in] EE_LIB::Util::Primitive::Bytes* **command** command

9-5-2 Primitive

9-5-2-1 Bytes

操作	ノート	パラメータ
Public	デストラクタ	なし

操作	ノート	パラメータ
~Bytes()	return なし	
Public **Bytes()**	size で指定したサイズのバイト列を生成する。 bResizeable を false にすると、コピーコンストラクタや代入、push_back 時に自動的なサイズの変更を行わない。 代入、push_back は size で指定したサイズ分まで行われ、サイズ外については中断される。 return なし	[in] unsigned int **size** サイズ（byte） [in] bool **bResizeable** true:自動的にリサイズする false:自動的にリサイズしない
Public **Bytes()**	rhs で指定したバイト列をコピーした、リサイズ可能なバイト列を生成する。 return なし	[in] Bytes& **rhs** コピー元 Byte 列
Public **Bytes()**	str で指定した文字列を格納した、リサイズ可能なバイト列を生成する。 return なし	[in] char* **str** 文字列
Public **Bytes()**	str で指定した文字列を格納した、size で指定したサイズのメモリを予め確保した、リサイズ可能なバイト列を生成する。 size で指定したサイズが str より小さい場合もサイズを拡張して、str を格納する。 return なし	[in] char* **str** 文字列 [in] unsigned short **length** サイズ(byte)
Public void **clear()**	格納した要素を消去する return なし	なし
Public unsigned short **count()**	data で指定した値がいくつあるかを数える return data で指定した値の数	[in] unsigned char **data** 検索する値
Public void **cutTheBack()**	前から index 個の要素を残し、後ろの要素を捨てる。 return なし	[in] unsigned short **index** 切断する要素のインデックス
Public void **cutTheFront()**	前から index 個の要素を捨てる。 return なし	[in] unsigned short **index** 切断する要素のインデックス
Public bool **decToValue()**	10 進数とみなし値に変換する return true 成功 false 失敗	[in] long& **value** 変換後の値 [in] unsigned short **index** 変換を開始する要素のインデックス [in] unsigned char **maxDigits** 最大桁数
Public unsigned short **find()**	検索開始要素に格納された data で指定した値を持つ要素を見つける return 最初に見つかったインデックスを返す。見つからなかったときは、0xffff を返す。	[in] unsigned char **data** 検索する値 [in] unsigned short **firstIdx** 検索開始要素のインデックス

操作	ノート	パラメータ
Public unsigned short **find()**	rhs と一致するバイト列を見つける return 最初に見つかったインデックスを返す。 見つからなかったときは、0xffff を返す	[in] Bytes& **rhs** バイト列
Public bool **hexToValue()**	16 進数とみなし値に変換する return true 成功 false 失敗	[in] unsigned long& **value** 変換後の値 [in] unsigned short **index** 変換を開始する要素のインデックス [in] unsigned char **maxDigits** 最大桁数
Public bool **isDecChar()**	index で指定した要素が数値なのか調べる return '0'-'9'のとき：true それ以外のとき：false	[in] unsigned short **index** 調査対象の要素のインデックス
Public unsigned char & **operator[]()**	idx で指定した要素を参照する return 要素の値	[in] unsigned short **idx** インデックス（C の配列と同じ）
Public Bytes& **operator=()**	代入演算子 return バイト列の参照	[in] Bytes& **rhs** 右辺
Public bool **push_back()**	1byte を最後に追加する return true 成功 false 失敗	[in] unsigned char **data** 追加する要素の値
Public bool **push_back()**	byts 型を連結する return true 成功 false 失敗	[in] Bytes& **data** 連結するバイト列
Public bool **push_back()**	指定した文字列の先頭から size バイトを連結する return true 成功 false 失敗	[in] char* **str** 連結する文字列 [in] unsigned short **size** サイズ(byte)
Public bool **push_back()**	指定した文字列を連結する return true 成功 false 失敗	[in] char* **str** 連結する文字列（NULL 終端文字列）
Public bool **resize()**	サイズを変更する return true 成功 false 失敗	[in] unsigned short **size** サイズ（byte）
Public unsigned short **size()**	サイズを取得する return 格納した要素の数	なし

9-5-2-2 Format

操作	ノート	パラメータ
Private void	push_back したデータを	なし

操作	ノート	パラメータ
clear()	クリアする。 return なし	
Public **Format()**	書式を指定可能な文字列クラス。 return なし	[in] char* **pFormat** 書式（printf 形式のサブセットで書式を指定する） 書式のフォーマット ：%[フラグ][フィールド幅][変換文字] ○ フラグ -: 右詰 ○ フィールド幅 変換文字に d を指定した場合、1～10 が有効範囲（無効の場合は無視） 変換文字に x を指定した場合、1～8 が有効範囲（無効の場合は無視） 数値または文字列がフィールド幅に満たない場合、以下のルールに従う。 ・ フィールド幅の先頭の数字が 0 の場合、フラグの指定に関係なく右詰めとする。(足りない桁は 0 で埋める) ・ フィールド幅の先頭の数字が 0 でない、かつ、右詰めの場合、右詰とする。（足りない桁はスペースで埋める） ・ フィールド幅の先頭の数字が 0 でない、かつ、左詰めの場合、 左詰めとする。（足りない桁はスペースで埋める） 数値の桁数または文字列の文字数がフィールド幅を超える場合、左からフィールド幅で切断する ○ 変換文字 d,D：10 進数に変換する d-short D-long x,X：16 進数に変換する x-short X-long s,S：文字列を出力する s-char* S-Bytes* [in] format の後に続く引数 書式の指定に従って編集する数値や文字列を書式で指定した順に、指定した個数指定する。
Public **~Format()**	デストラクタ @return なし	なし
Public unsigned short **getExpectationSize()**	フォーマット後の文字列長の予測値を取得する。（必ずこのサイズより小さくなるとは限らない。あくまで目安） return フォーマット後の文字列長の予測値を返す	なし
Public void **getString()**	フォーマット後の文字列を取得する return なし	[in] EE_LIB::Util::Primitive::Bytes& **str** フォーマット後の文字列への参照
Public Format& **operator=()**	代入算子 return フォーマットへの参照	[in] Format& **rhs** 代入元クラスの参照

１０参考文献

１） OOP　Foundations 「組込みUML」翔泳社、2002年

２） RENESAS 「ルネサス 16 ビットシングルチップマイクロコンピュータハードウエアマニュアル H8S/2258 グループ、H8S/2239 グループ、H8S/2238 グループ、H8S/2237 グループ、H8S/2227 グループ」RENESAS、2010年3月18日 Rev 6.00