ライントレーサ マニュアル V1.0

1　はじめに

1-1　キット内容

- ライントレーサ　　1台
- EE_TypeLT 開発環境 DVD　　1枚
- ダウンロードケーブル　　1本

1-2　1-1 以外に必要なもの

1） VMware Player3.0 (Windows 版) を実行できる PC

※ 仮想マシンに 512MB のメモリを割り当てるのでメモリは2GB 以上必要です

※ VMware Player3.0 は、Windows2000/XP/Server2003/Vista/7/XPx64/Server2003 x64/Vista x64/7 x64 で実行可能です。

※ PC にシリアルポートが少なくとも1ポート必要です。

2） アルカリ単3電池 または 充電式ニッケル水素単3電池 3本

1-3　ライントレーサのスペック

- CPU　　H8S/2398
- RAM　　CPU 内蔵 8KB　外部 128KB
- ROM　　外部 384KB
- 拡張ポート　　アナログ入力　　4チャンネル
- 動力　　130タイプ マブチモータ 2個
- ギアボックス　　タミヤ DOUBLE GEAR BOX ギア比 38.2:1
- ラインセンサ　　4個（8mm 間隔）
  検出可能な最小ライン幅 10mm
- ロータリエンコーダ　　インクリメンタル式 分解能 15.184mm

2　開発環境のインストール

2-1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/

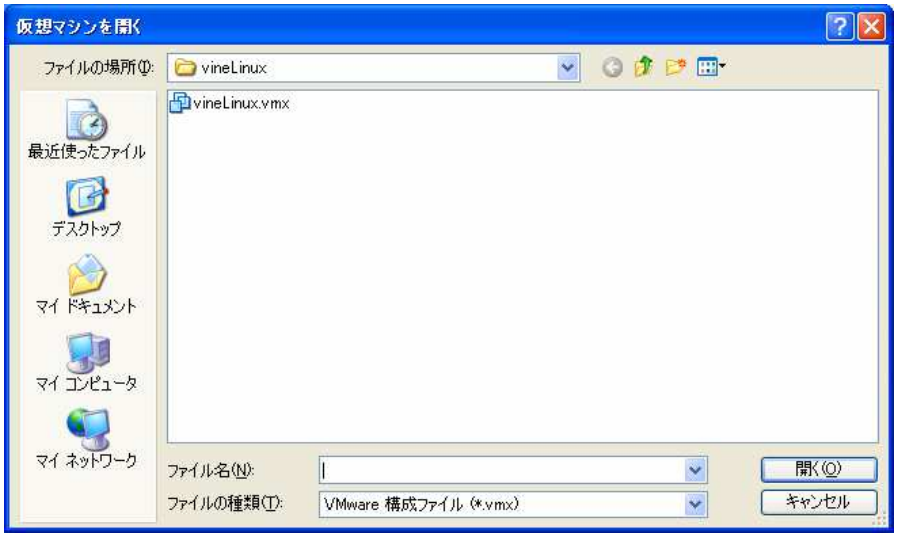
2-2　VMware　Player のインストール

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）EE_TypeLT　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。

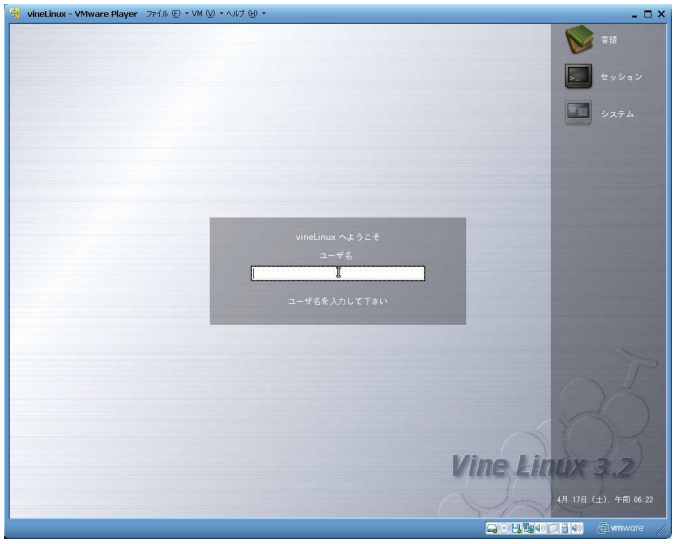
4）2)で解凍してできた vineLinux フォルダ中の vineLinux.vmx を選択する



５） VMware Player の左のツリーに vineLinux が追加されます。

６） VMware Player の仮想マシンの再生を選択し、vineLinux を起動します。

7） vineLinux にログインします。

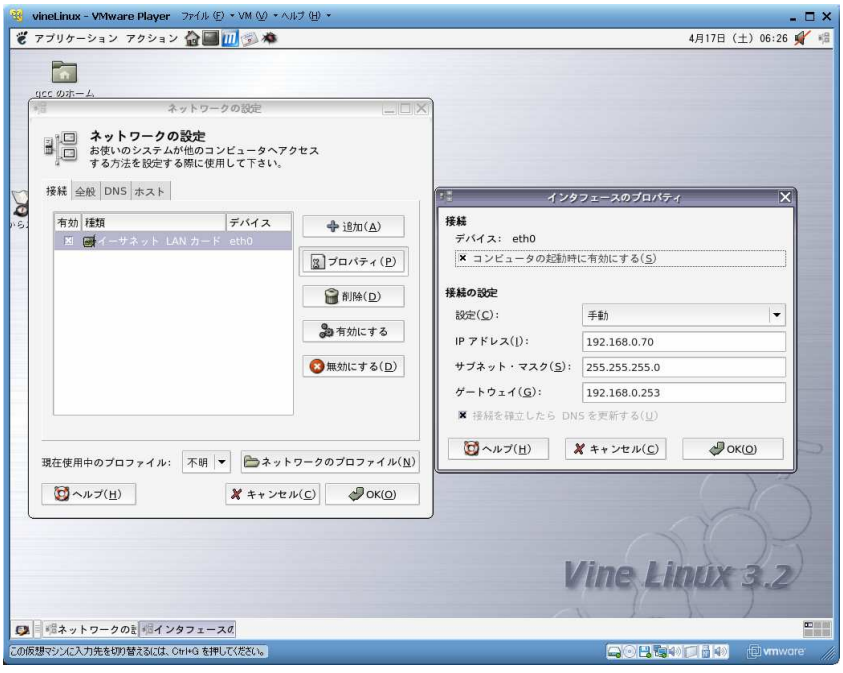


ユーザ gcc,パスワード vineLinux でログインしてください。

※　root のパスワードも vineLinux となっています。

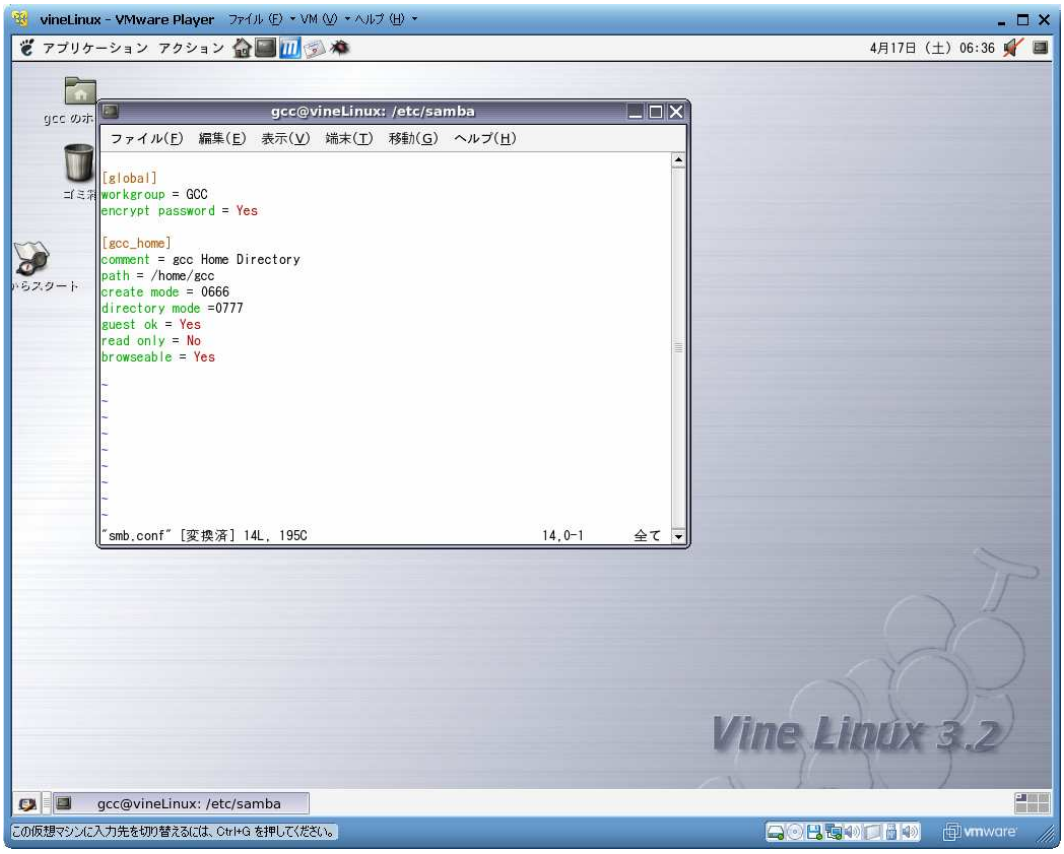
８）IP アドレスの設定

[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。



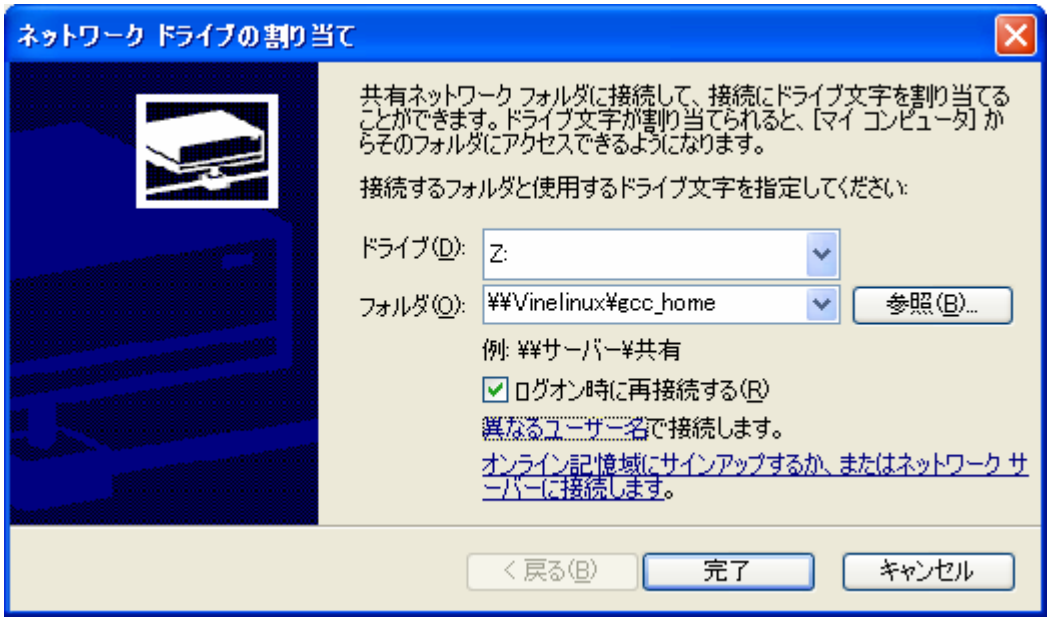
仮想マシンのユーザgccのホームディレクトリを PC のネットワークドライブに割り当て

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。

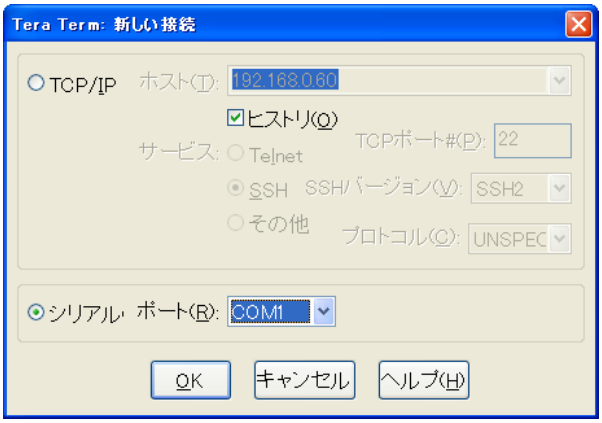


2-3 TeraTerm のインストール

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。

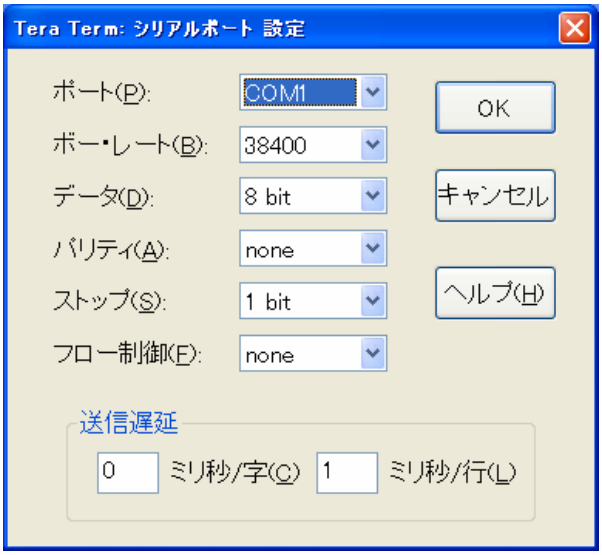
2） TeraTerm を起動する。

※ シリアルポートを指定する。



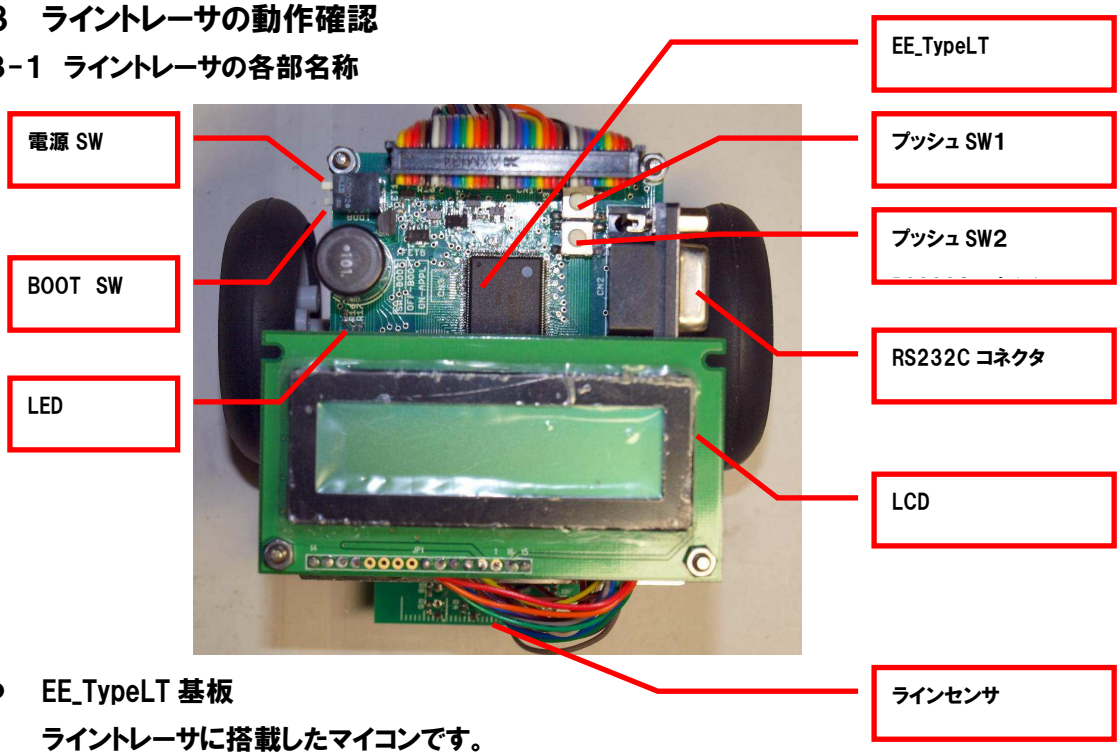
3） [設定] - [端末] で開くダイアログで下記のように設定する。

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。



3　ライントレーサの動作確認

3-1　ライントレーサの各部名称



- EE_TypeLT 基板
  ライントレーサに搭載したマイコンです。
- ラインセンサ
  ラインを読み取るセンサです
- 電源 SW
  ライントレーサの電源が入ります。（レバーが水平のとき OFF、下にたおしたとき ON です）
- BOOT SW
  電源投入時に、ユーザプログラムを起動するのか、BOOT プログラムを起動するのかを選択します。（レバーが水平のとき OFF、下にたおしたとき ON です）
  ON　　ユーザプログラムモード　電源投入によりユーザプログラムが起動します。
  OFF　　BOOT モード　　電源投入により BOOT プログラムが起動します。
- RS232C コネクタ
  PC とライントレーサをダウンロードケーブルでつなぐときに使うコネクタです。
- LED
  プログラムから点灯、消灯を操作できる LED です。BOOT プログラム起動時は点灯します。
- LCD
  16 文字、2行表示の LCD です。

3-2電池を入れる

LCD および EE_TypeLT をとめるナットをはずして、EE_TypeLT 一旦取り外した上で電池を入れてください。

3-3ライントレース

テストコースを机や床など凹凸のない場所にテープで固定し、黒いラインの上にライントレーサをおいて電源を入れてください。

ライントレーサは初期状態では、ライントレースプログラムがロードされているので、プッシュ SW1 を押すとラインに沿って走り始めます。

※ 電源を入れてもラインをトレースしない場合、ライントレーサを置く位置を変えて、プッシュ SW1 を再度押してください。

3-4 プログラムのダウンロード

１） PCとライントレーサの接続

PC とライントレーサをダウンロードケーブルで接続して、PC 上に TeraTerm を起動しておきます。
ライントレーサの BOOT SW を OFF にした状態で電源を入れてください。

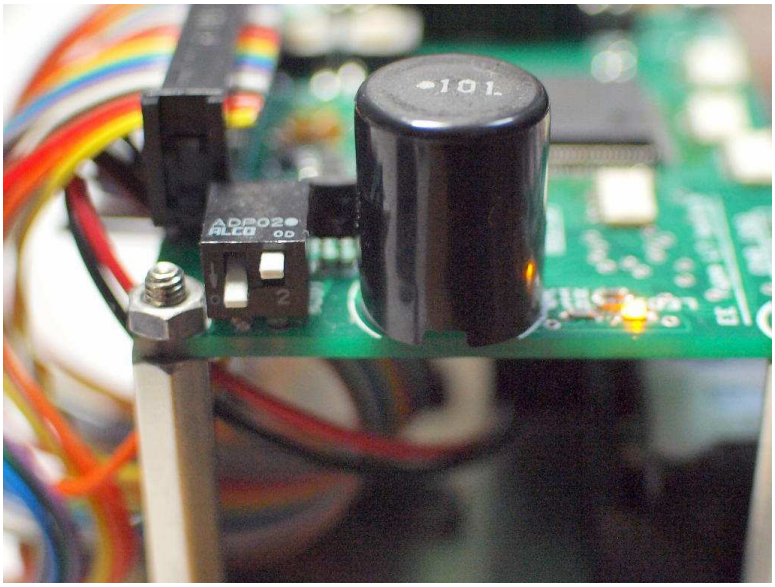
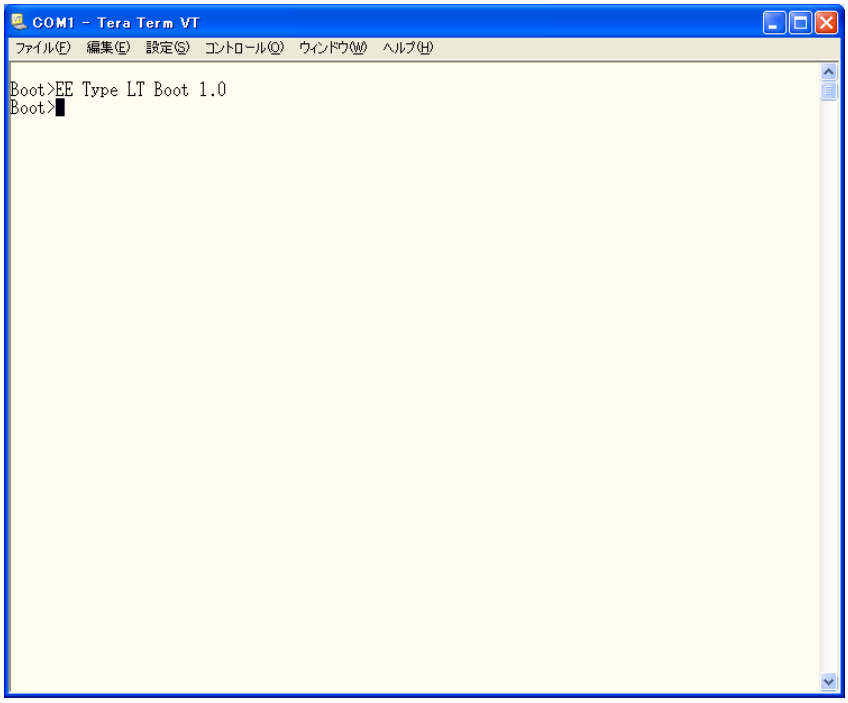


写真 3-2-1 ディップスイッチの状態(左:電源 SW ON 状態、右:BOOT SW OFF 状態）

TeraTerm に「Boot>EE Type LT Boot x.x」(x.x はバージョン番号をあらわす数値)というメッセージの後に Boot>のプロンプトが出ます。

※ 表示が出ない場合、ケーブルの接続、TeraTerm のシリアルポートの設定を確認してください。

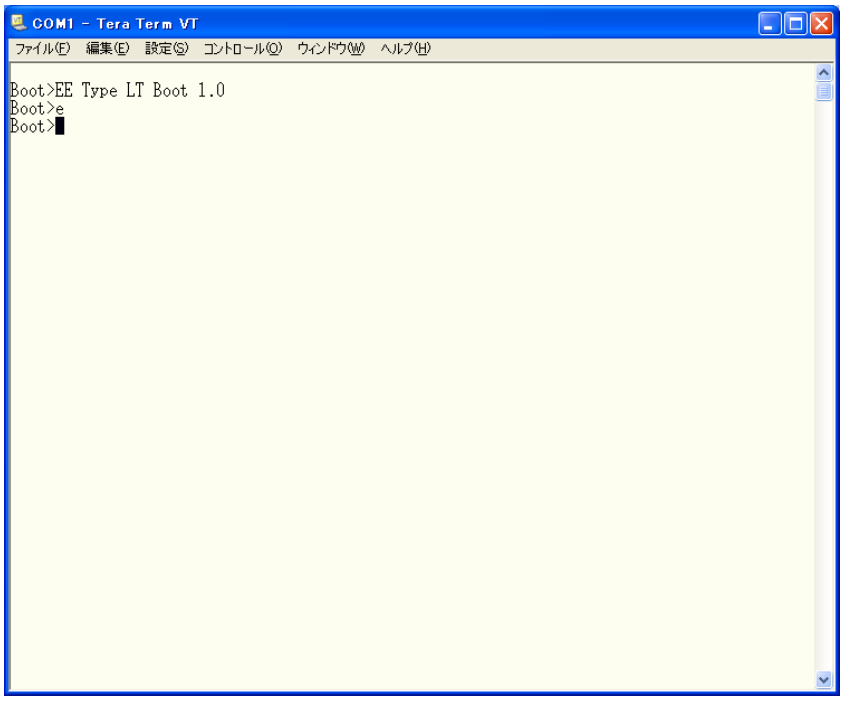


２）プログラムの消去

TeraTerm上でeリターンとタイプし、プログラムを消去します。

Boot> e ↓

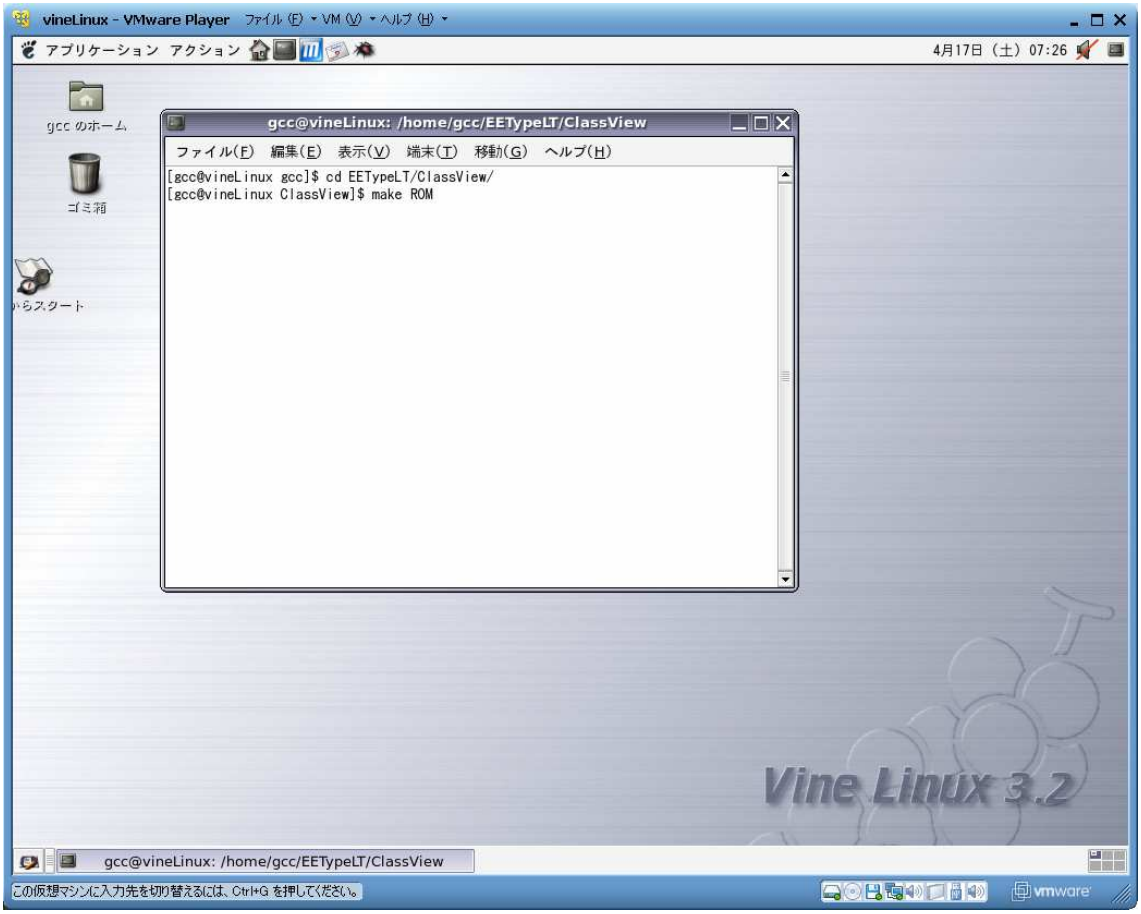
※　プロンプトが 5～10 秒程度で戻ってきます。



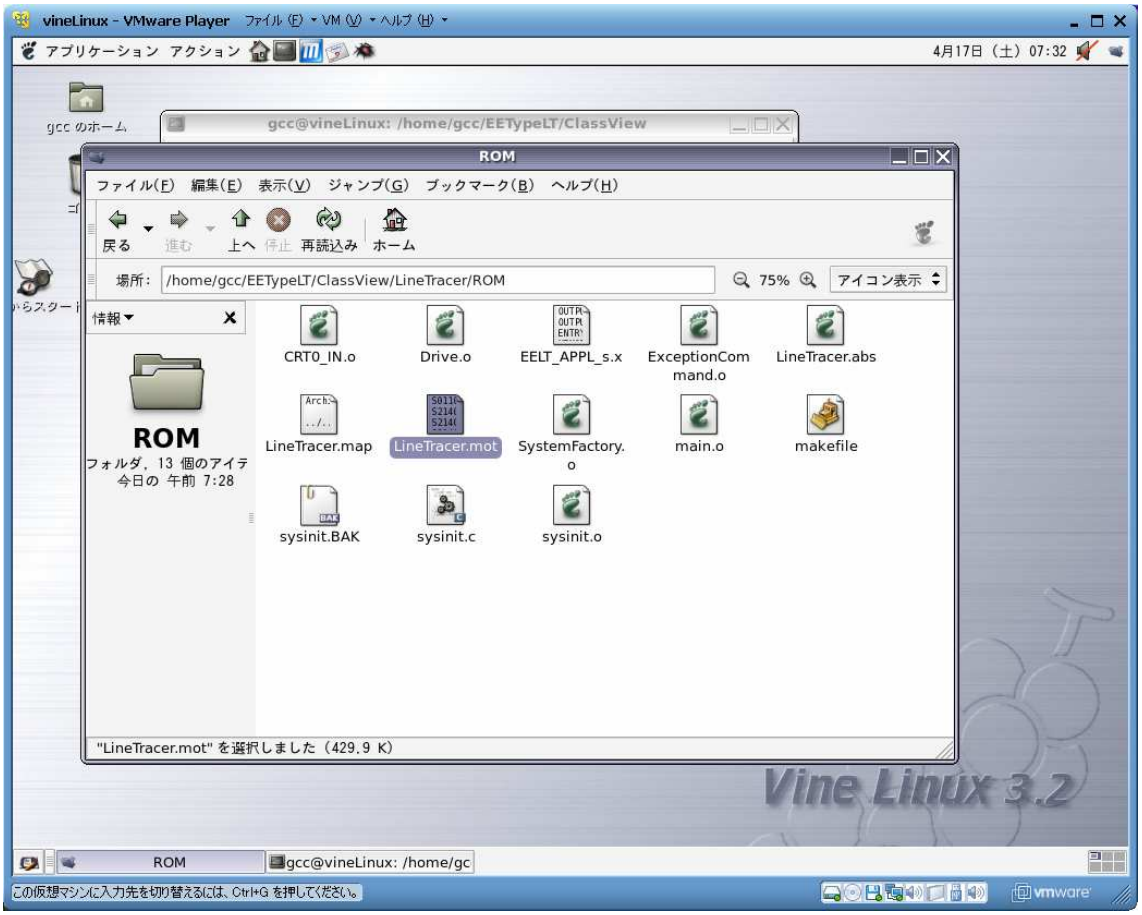
３）プログラムの make

仮想マシンにユーザgccでログインします。

GNOME 端末を開いて、ホームディレクトリの下の EETypeLT/ClassView の下に移動し、make ROM とタイプしてください。

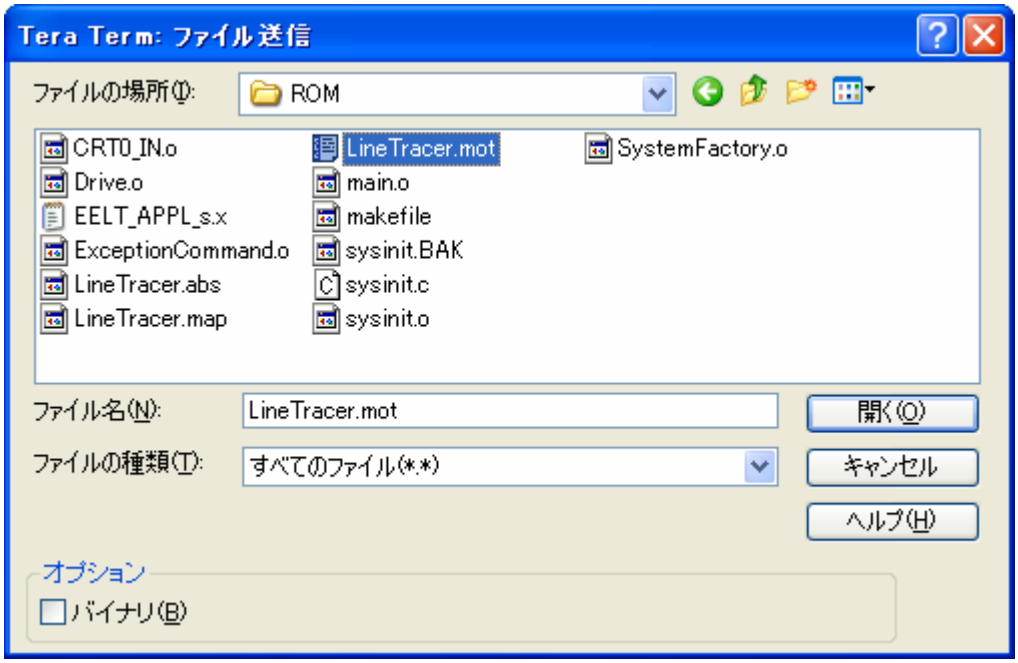


ホームディレクトリの EETypeLT/ClassView/LineTracer/ROM の下に LineTracer.mot ができたことを確認してください。

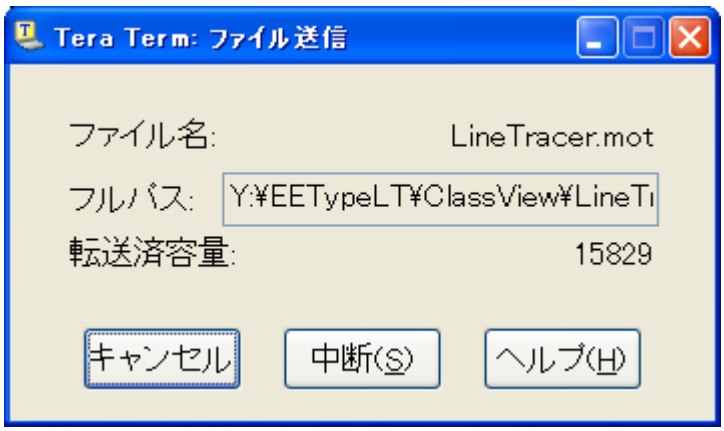

vineLinux - VMware Player
ファイル (F) VM (V) ヘルプ (H)
アプリケーション アクション
4月17日（土）07:32
gcc のホーム
gcc@vineLinux: /home/gcc/EETypeLT/ClassView
ROM
ファイル(F) 編集(E) 表示(V) ジャンプ(G) ブックマーク(B) ヘルプ(H)
戻る 進む 上へ 停止 再読込み ホーム
場所: /home/gcc/EETypeLT/ClassView/LineTracer/ROM
75% アイコン表示
情報
ROM
フォルダ，13 個のアイテ
今日の 午前 7:28
CRT0_IN.o
Drive.o
EELT_APPL_s.x
ExceptionCommand.o
LineTracer.abs
LineTracer.map
LineTracer.mot
SystemFactory.o
main.o
makefile
sysinit.BAK
sysinit.c
sysinit.o
"LineTracer.mot" を選択しました（429.9 K）
Vine Linux 3.2
ROM
gcc@vineLinux: /home/gc
この仮想マシンに入力先を切り替えるには、Ctrl+G を押してください。
vmware

４）プログラムのロード

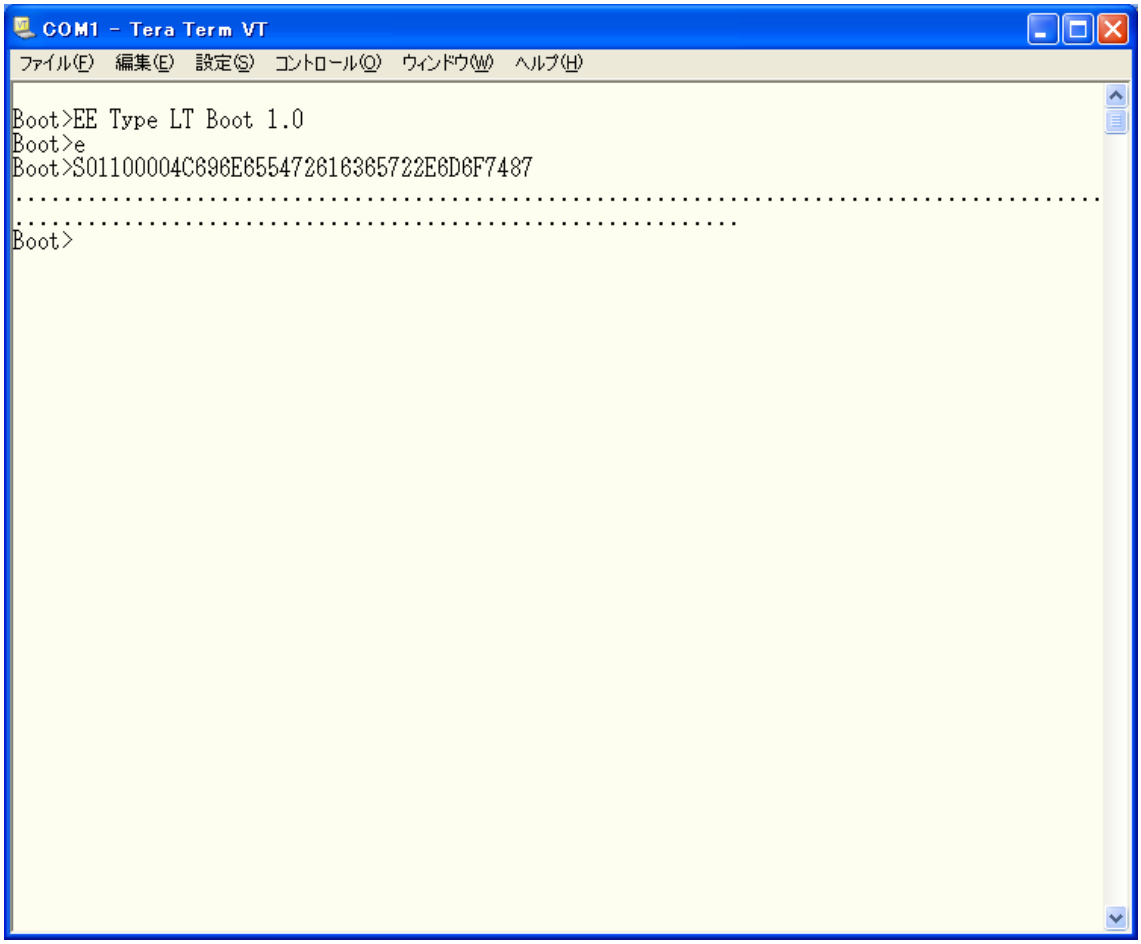
TeraTerm の［ファイル］－［ファイルの送信・・・］で３）で作った LineTracer.mot を選択します。



ファイルが送信中は下記のダイアログが表示され、TeraTerm 上に.（ピリオド）が３秒に１回程度の頻度で表示されていきます。

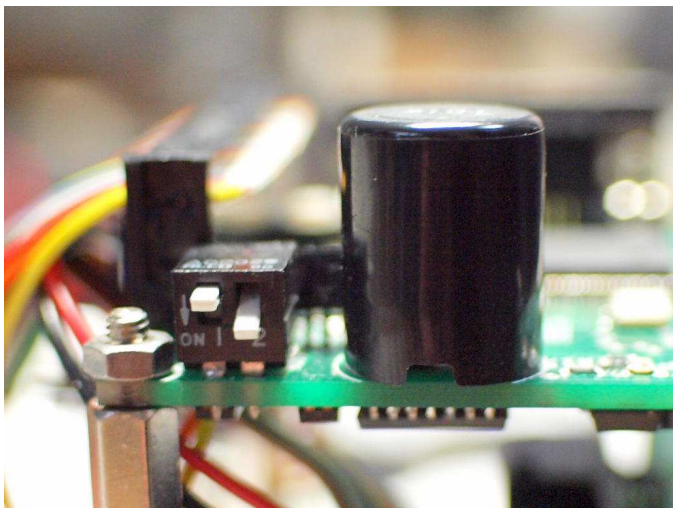


ダイアログが消え、下記のように TeraTerm に Boot>プロンプトがかえってきたらプログラムのロードが完了です。



５）プログラムの動作確認

ライントレーサの電源を切って、Boot SW を ON にします。



３－１と同じ手順でライントレーサの動作確認をしてください。

テストコースをライントレーサが走れば、プログラムのロードは成功です。

4　新しいプロジェクトを作ってみよう

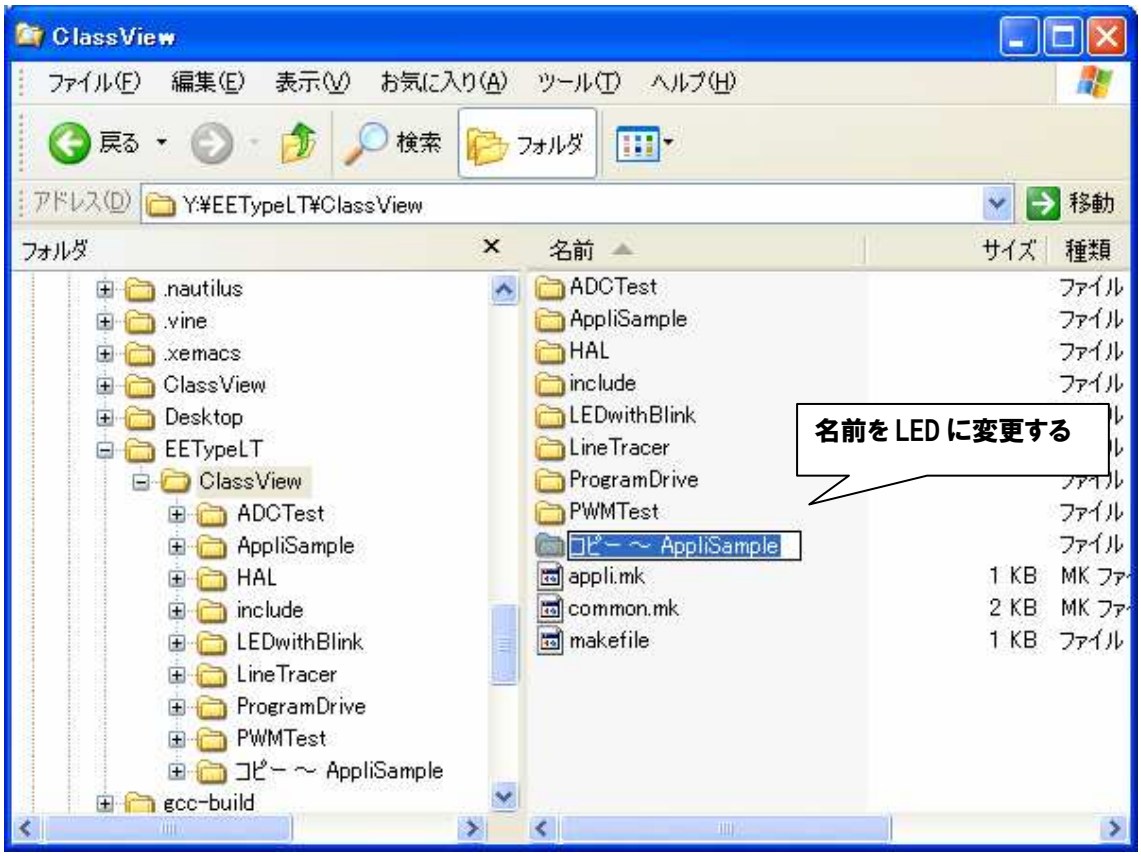
理屈を抜きにして、まず、プログラムを作って動かしてみましょう。

4-1 LED を点滅させるプログラム

１） 新しいプロジェクトを作る

① VineLinux を起動します。

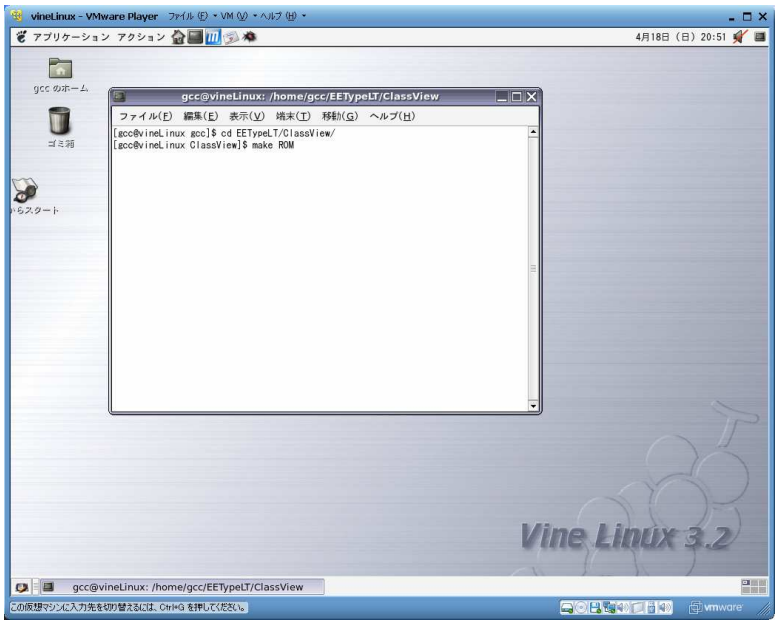
② ユーザgccのホームディレクトリの下の EETypeLT/ClassView/AppliSample を、ユーザgccのホームディレクトリの下の EETypeLT/ClassView にコピーし、そのディレクトリの名前を LED とします。



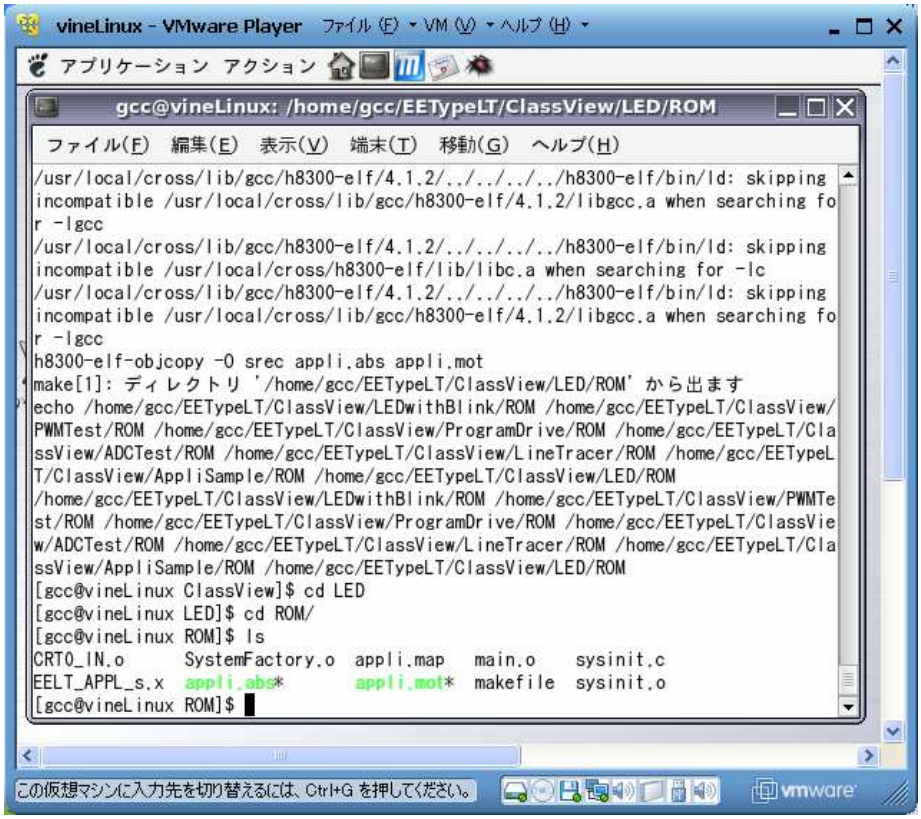
③ ユーザｇｃｃのホームディレクトリの下の EETypeLT/ClassView/makefile の中の Applications という変数の最後に LED を追加する。（LED の前の文字列との間はスペースで区切る）

```
Applications = LineTracer AppliSample LED
```

④ vineLinux にｇｃｃでログインして、GNOME 端末を開いて、ホームディレクトリの下の EETypeLT/ClassView の下に移動し、make ROM を実行する。



⑤ ユーザgccのホームディレクトリの下の EETypeLT/ClassView/LED/ROM の下に appli.mot ができていれば、プロジェクトの追加は成功です。



⑥ ホームディレクトリの下の EETypeLT/ClassView の下で make clean を実行し、確認のために作ったオブジェクトファイルをクリアしておきます。

２）プログラムする。

１)で作ったプロジェクトの SystemFactory.cpp を下記のように修正する。

※　太字の部分が追加するところです。

```
#include "SystemFactory.h"
#include "Unit/CPUBoard/CPUBoardFactory.h"
#include "Unit/CPUBoard/CPUBoard.h"
#include "Device/CPU/IOPortNo.h"
#include "Unit/Display/LED/LED.h"

using Appli::SystemFactory;

void SystemFactory::startup () {
    Unit::CPUBoard::CPUBoard* pCPUBoard;
    Unit::CPUBoard::CPUBoardFactory* pCpuBoardFactory;
    Device::CPU::IOPin* pin;
    bool bRet=false;
    // ============== CPUBoard の生成 ====================
    pCpuBoardFactory=Unit::CPUBoard::CPUBoardFactory::getInstance ();

bRet=pCpuBoardFactory->create (Unit::CPUBoard::CPUBoard::CBT_EE_TYPE_LT,Unit::CPUBoard::CPUBoard::BM_APP
LI);
    if (bRet==true) {
        // ============== IOpin の生成 ====================
        pCPUBoard=pCpuBoardFactory->getIF ();
        bRet=pCPUBoard->addIOPin (Device::CPU::IOPI_H8_PORT6,
                    Device::CPU::IOPin::PNOIOP_5,
                    Device::CPU::IOPin::DOIOP_OUT,
                    &pin);
        if (bRet==true)  {
        // ================ LED の生成 ====================
            Unit::Display::LED::LED led (pin);
            led.setPolarity (Unit::Display::LED::LED::P_REVERSE); // IOpin が H で消灯の回路なので極性を変更する
                while (-1)
                {
                    volatile int a,b,c,i,j;
                    led.turnOn ();    // LED 点灯
                    for (a=10,b=20,j=0; j<10; j++)  {for (i=0;i<1000; i++)  {c=c*a+b;} }
                    led.turnOff ();  // LED 消灯
                    for (a=10,b=20,j=0; j<10; j++)  {for (i=0;i<1000; i++)  {c=c*a+b;} }
                }
            }
    }
    while (-1)  {}
}

SystemFactory::SystemFactory () {}
SystemFactory::~SystemFactory () {}
```

３）プログラムの動作を確認する

3-4 と同様の手順でプログラムをダウンロードし、ライントレーサの LED が点滅するか確認してみましょう。

3-4 の手順と違うのは、ダウンロードするプログラムが、今回は EETypeLT/ClassView/LED/ROM の下の appli.mot になるという点のみです。

5　クラスライブラリ(EE_TypeLT　HAL)

クラスライブラリ　EETypeLT　HAL　は、アプリケーションプログラムからハードウエアを隠蔽することを目的に開発したものです。

5-1　EETypeLT クラス

OS の初期化やハードウエア部品の生成などを一手に引き受けるクラスです。EE_TypeLT ボード上でアプリケーションを作成する場合は、最初に生成するクラスです。

【使用例】

```
#include "Unit/CPUBoard/EETypeLT.h"
・・・
Unit::CPUBoard::EETypeLT* pCPUBoard;
//================ CPU ボードの生成と初期化 ================
pCPUBoard = new Unit::CPUBoard::EETypeLT();
pCPUBoard->init(Device::CPU::Timer::TI_H8_16BIT_5,5000000UL);      // OS タイマ：5ms
```

【操作の説明】

操作	ノート	パラメータ
Public EETypeLT()	コンストラクタ @return　なし	[in] CPUBoard::BootMode bootMode ブートモード(Appli or Boot)を指定する。デフォルトが Appli となっているため通常は指定しなくてよい。
Public ~EETypeLT()	デストラクタ	
Public UI::Switch* getSw1()	スイッチ1のインスタンスへのポインタを取得する @return NULL のとき、取得失敗。	
Public UI::Switch* getSw2()	スイッチ2のインスタンスへのポインタを取得する @return NULL のとき、取得失敗。	
Public Unit::Communication::Com* getCom()	Com　ポートのインスタンスへのポインタを取得する @return NULL のとき、取得失敗。	
Public Unit::Display::LED::LEDwithBlink* getLED()	オンボード LED のインスタンスへのポインタを取得する @return NULL のとき、取得失敗。	
Public UI::CommandInterpreter::CommandInterpreterFactory* getCIFactory()	通信インターフェイス　Factory(communicationInterfaceFactory)のインスタンスへのポインタを取得する @return NULL のとき、取得失敗。	
Public bool addIOPin()	IOPin を生成する @return true　成功　false　失敗	[in] Device::CPU::IOPortNo portNo ポート No を指定する [in] Device::CPU::IOPin::PinNoOfIOPort pinNo ピン番号を指定する [in] Device::CPU::IOPin::DirectionOfIOPort direction 入力、出力を指定する [in] Device::CPU::IOPin** pin 生成した IO ピンのインスタンスへのポインタを返す
Public bool	Timer　の追加と初期設定を行う　既に追	[in] Device::CPU::Timer::TimerId timerId

操作	ノート	パラメータ
addTimer ()	加されている場合は、Timer を返し、正常終了する。 Timer を持たない CPU は常に失敗を返す @return true 成功 false 失敗	追加する Timer を指定する [in] Common::TypeDef::UFourByte interval インターバル (ns) を指定する [in] Device::CPU::InterruptPriority ip H8 の場合、割り込み優先度を指定する IP_H8_LOW:非優先 IP_H8_HIGH:優先 [in] Device::CPU::Timer** timer 生成したタイマのインスタンスへのポインタを返す [in] Device::CPU::Timer::TimerMode tm タイマーの種類を指定する (TM_FREE_RUN or TM_PWM) [in] Device::CPU::TimerRegisterable* handler FREE_RUN タイマのとき、呼び出すハンドラを登録する。PWM の時は NULL を設定する
Public void enableInterrupt ()	割り込みを許可する @return なし	
Public void disableInterrupt ()	割り込みを禁止する @return なし	
Public void enableHighPriorityInterrupt ()	プライオリティの高い割り込みのみ割り込みを許可する。 このモデルでは、割り込みのプライオリティを２レベル（IP_H8_HIGH、IP_H8_LOW）設定できる。このメソッドにより、IP_H8_HIGH のレベルの割り込みのみを許可する。 @return なし	
Public Device::CPU::CPU* getCPU ()	CPU クラスのインスタンスへのポインタを取得する @return NULL のとき、取得失敗。	
Public Unit::MotorDriver::MotorDriver* getMotorDriver ()	モータドライバのインスタンスへのポインタを取得する @return NULL のとき、取得失敗。	
Public bool init ()	CPU ボードの初期化を行う @return true 成功 false 失敗	
Public bool init ()	CPU ボードの初期化を行う @return true 成功 false 失敗	[in] Device::CPU::Timer::TimerId timerId OSTimer として使用する Timer を指定する [in] Common::TypeDef::UFourByte cycle OSTimer のインターバル (ns) を指定する ノンプリエンプティブマルチタスク方式を採用しているため、すべてのタスクの処理時間の合計をこの時間より短く設計する必要がある。 タスクの実行時間がこの時間を越えると、周期実行タスクなどの呼び出し周期が保障されない。
Public Device::CPU::ADConverter* getADC ()	AD コンバータのインスタンスへのポインタを取得する @return NULL のとき、取得失敗。	

5-2 IOPin クラス

H8 の汎用ポートを抽象化したクラスです。

【使用例】

```
#include "Device/CPU/IOPin.h"
・・・
Device::CPU::IOPin* pRS;
//================ IOPin の初期化と生成 ======================
bRet=pCPUBoard->addIOPin (Device::CPU::IOPI_H8_PORT2,
                          Device::CPU::IOPin::PNOIOP_4,
                          Device::CPU::IOPin::DOIOP_OUT,
                          &pRS);
if (bRet==false)  goto ERROR;
・・・
pRS->setData (0); // Port2_4 を Low にする
pRS->setData (1); // Port2_4 を High にする
```

【操作の説明】

操作	ノート	パラメータ
Public IOPin ()	コンストラクタ @return なし	
Public ~IOPin ()	デストラクタ @return なし	
Public bool Pure setPinNo ()	ポート No と PinNo を設定する。 @return true 成功 false 失敗	[in] Device::CPU::IOPortNo portNo ポート No を指定する [in] PinNoOfIOPort pinNo PinNo を指定する。
Public PinNoOfIOPort Pure getPinNo ()	PinNo を取得する。 @return PinNo	
Public DirectionOfIOPort Pure getDirection ()	方向(入力/出力)を取得する @return 方向	
Public bool Pure setDirection ()	入力、出力を切り替える @return true 成功 false 失敗	[in] DirectionOfIOPort direction 方向(入力:DOIOP_IN、出力:DOIOP_OUT)
Public void Pure setData ()	ポートに出力する。(direction に DOIOP_OUT を設定したときに使用してください) @return なし	[in] Common::TypeDef::UOneByte data 0:Low レベル 0 以外:High レベル
Public Common::TypeDef::UOneByte Pure getData ()	Pin のレベルを取得する。(direction に DOIOP_IN を設定したときに使用してください) @return Low レベルのとき0、High レベルのとき1を返す	

5-3 LCD クラス

LCD を抽象化したクラスです。

【使用例】

```
#include "Unit/Display/LCD/LCD.h"
・・・
```

```
Unit::Display::LCD::LCD* pLCD;
//================ LCD の初期化と生成 ======================
pLCD = new Unit::Display::LCD::LCD(16U,2U,pRS,pRW,pE,pDB4,pDB5,pDB6,pDB7);
if(pLCD==NULL) goto ERROR;
pLCD->init();
pLCD->returnHome();
pLCD->print("Hello!!");
```

【操作の説明】

操作	ノート	パラメータ
Public LCD()	コンストラクタ @return なし	[in] Common::TypeDef::UTwoByte width 横方向の表示文字数 [in] Common::TypeDef::UTwoByte hight 縦方向の表示文字数 [in] Device::CPU::IOPin* pRS LCD コネクタの RS 信号に接続した IOPin を指定する [in] Device::CPU::IOPin* pRW LCD コネクタの RW 信号に接続した IOPin を指定する [in] Device::CPU::IOPin* pE LCD コネクタの E 信号に接続した IOPin を指定する [in] Device::CPU::IOPin* pDB4 LCD コネクタの DB4 信号に接続した IOPin を指定する [in] Device::CPU::IOPin* pDB5 LCD コネクタの DB5 信号に接続した IOPin を指定する [in] Device::CPU::IOPin* pDB6 LCD コネクタの DB6 信号に接続した IOPin を指定する [in] Device::CPU::IOPin* pDB7 LCD コネクタの DB7 信号に接続した IOPin を指定する
Public ~LCD()	デストラクタ @return なし	
Public void init()	LCD を初期化する @return なし	
Public void clear()	画面をクリアする @return なし	
Public void returnHome()	カーソルをフォームに戻す @return なし	
Public void print()	カーソルで指定した位置に文字列を表示する @return なし	[in] char* str 文字列を指定する(文字列は NULL 終端すること)
Public void print()	カーソルで指定した位置に文字列を表示する @return なし	[in] Common::BasicType::Format& format 文字列を指定する(文字列は NULL 終端すること)
Public void cursor()	カーソルを移動する 右上を原点(0,0)とし、右に1文字移動するとx座標を＋1、下に1文字移動するとy座標を＋1する座	[in] Common::TypeDef::UTwoByte x x座標を指定する。

	標系で座標を指定する @return なし	[in] Common::TypeDef::UTwoByte y y 座標を指定する。

5-4 LED クラス

LED を抽象化したクラスです。

【使用例】

```
#include "Unit/Display/LED/LED.h"
・・・
Device::CPU::IOPin* pin;
・・・
Unit::Display::LED::LED led(pin);  // pin は生成済みの IOpin へのポインタ
led.setPolarity(Unit::Display::LED::LED::P_REVERSE);
while(-1)
{
        volatile int a,b,c,i,j;
        led.turnOn();
        for(a=10,b=20,j=0; j<10; j++) {for(i=0;i<1000; i++) {c=c*a+b;}}
        led.turnOff();
        for(a=10,b=20,j=0; j<10; j++) {for(i=0;i<1000; i++) {c=c*a+b;}}
}
```

【操作の説明】

操作	ノート	パラメータ
Public LED()	コンストラクタ @return なし	[in] Device::CPU::IOPin* ioPin LED を制御する IOPin を指定する
Public ~LED()	デストラクタ @return なし	
Public void setPolarity()	極性(turnOn 時に IOPin を High にするか、Low にするか)を指定する。 @return なし	[in] Polarity polarity P_NORMAL:turnOn 時に IOPin を High にする、P_REVERSE:turnOn 時に IOPin を Low にする
Public Polarity getPolarity()	極性を取得する @return 極性を返す	
Public void turnOn()	LED を点灯する	
Public void turnOff()	LED を消灯する	

5-5 LEDwithBlink クラス

ブリンク機能付き LED クラスです。

【使用例】

```
#include "Unit/Display/LED/LEDwithBlink.h"
・・・
Unit::Display::LED::LEDwithBlink * pLED;
// ============== LED を EE_Type_LT から取得する ====================
pLED=pCPUBoard->getLED();
pLED->turnOn();        // LED を点灯
pLED->setLightingTime(500);                    // 点灯時間 0.5 秒
pLED->setLightingInterval(500);                // 消灯時間 0.5 秒
pLED->startBlink();                            // 点滅開始
```

【操作の説明】

操作	ノート	パラメータ
Public LEDwithBlink()	コンストラクタ @return なし	[in] Device::CPU::IOPin* ioPin LED を制御する IOPin を指定する
Public ~LEDwithBlink()	デストラクタ @return なし	
Public bool init()	LED を初期化する @return なし	
Public void startBlink()	点滅を開始する @return なし	
Public void stopBlink()	点滅を停止する @return なし	
Public bool setLightingTime()	点灯する時間(ms)を設定する	[in] Common::TypeDef::UFourByte time time
Public bool setLightingInterval()	消灯する時間(ms)を設定する	[in] Common::TypeDef::UFourByte interval interval
Public Common::TypeDef::UFourByte getLightingTime()	点灯する時間(ms)を取得する	
Public Common::TypeDef::UFourByte getLightingInterval()	消灯する時間(ms)を取得する	

5-6 ADConverter クラス

AD コンバータを抽象化したクラスです。

【使用例】

```
#include "Device/CPU/ADConverter.h"
・・・
Device::CPU::ADConverter* pADC;
// ============== ADC を EE_Type_LT から取得する ====================
pADC=pCPUBoard->getADC();
if(pADC==NULL) goto ERROR;
bRet=pADC->addChannel(Device::CPU::ADConverter::C_AN0);
```

```
if (bRet==true) { bRet=pADC->addChannel (Device::CPU::ADConverter::C_AN1); }
if (bRet==true) { bRet=pADC->addChannel (Device::CPU::ADConverter::C_AN2); }
if (bRet==true) { bRet=pADC->addChannel (Device::CPU::ADConverter::C_AN3); }
if (bRet==true) { pADC->start (Device::CPU::ADConverter::M_SINGLE); }
・・・          // 1ms 程度  時間をあける
data [0] =pADC->getData (Device::CPU::ADConverter::C_AN0);
data [1] =pADC->getData (Device::CPU::ADConverter::C_AN1);
data [2] =pADC->getData (Device::CPU::ADConverter::C_AN2);
data [3] =pADC->getData (Device::CPU::ADConverter::C_AN3);
```

【操作の説明】

操作	ノート	パラメータ
Public ADConverter ()	コンストラクタ @return なし	
Public ~ADConverter ()	デストラクタ @return なし	
Public bool Pure addChannel ()	AD 変換する対象のチャネルを指定する @return 成功:true 失敗:false	[in] Channel c チャネルを指定する
Public void Pure start ()	AD 変換を開始する。 @return なし	[in] Mode mode M_SINGLE:1チャネル分の AD 変換を行う。addChannel で指定したチャネルが4チャネルの場合、1回で4チャネル分の変換を行う。 複数のチャネルが指定されている場合、変換するチャネルを自動的に順に変更する。 M_SCAN:addChannel で指定したチャネルの AD 変換を stop が呼び出されるまで常に行う。
Public void Pure stop ()	AD 変換を停止する。 @return なし	
Public Common::TypeDef::UFourByte Pure getData ()	AD 変換後のデータを取得する @return AD 変換後のデータ	[in] Channel c チャネルを指定する

5-7 SwitchListener インターフェイス

Switch パッケージのインターフェイスです。

【使用例】

```
・・・
PushSW1* pPSW1;                    // PushSW1 は、SwitchListener インターフェイスを継承したクラス
PushSW2* pPSW2;                    // PushSW2は、SwitchListener インターフェイスを継承したクラス
//=============== Switch クラスに SwitchListener を登録する ===============
pPSW1 = new PushSW1 (pDsFSM);
pPSW2 = new PushSW2 (pDsFSM);

pCPUBoard->getSw1 () ->setListener (pPSW1);
pCPUBoard->getSw2 () ->setListener (pPSW2);
```

【操作の説明】

操作	ノート	パラメータ
Public SwitchListener ()	コンストラクタ @return なし	
Public ~SwitchListener ()	デストラクタ @return なし	
Public void Pure OnTurnOn ()	IOPin の状態が Low から High に変化したときに、Switch クラスから呼び出されるメソッド。 ユーザーはこのメソッドにスイッチが On になったときの処理を書く @return なし	
Public void Pure OnTurnOff ()	IOPin の状態が High から Low に変化したときに、Switch クラスから呼び出されるメソッド。 ユーザーはこのメソッドにスイッチが Off になったときの処理を書く @return なし	

5-8 CommandInterpreter クラス、Command インターフェイス、Com クラス

SCI を使ってコマンドの処理を行うときに便利なクラス群です。

【使用例】

```
UI::CommandInterpreter::CommandInterpreterFactory* pCIF;
UI::CommandInterpreter::CommandInterpreter* pCI;
・・・
//================ Com ポートと CommandInterpreter の初期化 ================
pCIF=pCPUBoard->getCIFactory();              // EE_Type_LT から CommandInterpreterFactory を取得する
pCI=pCIF->getCommandInterpreter();           // CommandInterpreter を取得する。ここまではおまじないのようなもの。
pDumpCommand = new DumpCommand(pLineSensor);      // コマンドとして DumpCommand を登録する。
                                                  // Commad インターフェイスを継承したクラスである。
pCI->setPrompt("LineTracer>");               // プロンプトを設定
pCIF->init("LineTracer 1.0");                // すべてのコマンドとプロンプトの設定が完了した後に、初期化する。
pCPUBoard->getCom()->send("OSTimerInterval ",OsWrapper::OSFactory::getInstance()->getInterval(),10,false,true);
```

【CommandInterpreter の操作の説明】

操作	ノート	パラメータ
Public CommandInterpreter ()	コンストラクタ @return なし	[in] Unit::Communication::Com* com 通信ポートを指定する [in] UI::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。
Public ~CommandInterpreter ()	デストラクタ @return なし	
Public void perform ()	文字列受信時に呼び出され、登録したコマンド check メソッドが true を返す場合、そのコマンドの execute メソッドを呼び出す。 コマンドの execute メソッドでは、最低限受信バッファの開放をする必要がある。	[in] OsWrapper::Task::Task* pTask pTask

	@return なし	
Public bool addCommand ()	コマンドを追加する @return true 成功 false 失敗	[in] UI::CommandInterpreter::Command* command 追加するコマンドを指定する
Public void setPrompt ()	プロンプトを設定する @return なし	[in] char* msg プロンプトへのポインタを指定する。

【Command の操作の説明】

操作	ノート	パラメータ
Public Command ()	コンストラクタ @return なし	
Public ~Command ()	デストラクタ @return なし	
Public bool Pure check ()	commandに渡された文字列が、処理すべきコマンドであるとき True を返す。 例えば、 dump が処理すべきコマンドの場合、ユーザは Command インターフェイスを継承した dump コマンドを処理するためのクラス（DumpCommand）を作成し、dump という文字列が command の先頭から4文字と一致する場合に check 関数が True を返すように記述する。 そのクラスを CommandInterpreter の addCommand メソッドを使ってコマンドインタラプタに登録しておくと LineTracer>dump 10 とタイプしたとき（LineTracer>はプロンプト） Command には dump 10 が渡されてくるので、chek メソッドは True を返すこととなり、DumpCommand の execute メソッドが CommandInterpreter から呼び出されることになる。	[in] Common::BasicType::Bytes* command
Public void Pure execute ()	Command の実行処理を記述するメソッドである。 このメソッドは、最低限 command の開放をする必要がある。 Check の説明の中の例の場合、command には、dump 10 が渡されてくるので、ユーザーは DumpCommand クラスのこのメソッドに、そのコマンドを処理するための処理を記述する。	[in] Unit::Communication::Com* com [in] Common::BasicType::Bytes* command

【Com クラスの操作の説明】

操作	ノート	パラメータ
Public Com ()	コンストラクタ @return なし	[in] Unit::Communication::Receiver* pr Receiver クラスへのポインタを指定する [in] Unit::Communication::Transmitter* pt

		Transmitter クラスへのポインタを指定する
Public ~Com ()	デストラクタ @return なし	
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を指定する [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] Common::BasicType::Format& format フォーマット付き文字列への参照を指定する [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] Common::BasicType::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
Public OsWrapper::Task::MessageQueue* getReceptionQueue ()	受信キューを取得する @return 受信キューへのポインタを返す	
Public void setNewLine ()	改行文字を設定する @return なし	[in] Common::TypeDef::UOneByte 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 sendNewLine ()	改行を送信する @return なし	
Public void sendPrompt ()	プロンプトを送信する @return なし	
Public void releaseReceptionBuffer ()	受信した文字列を開放する。 通信ポートから受信した文字列は、このメソッドを使って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 を使って送り返すか、または、ユーザーが delete しなければならない。 @return なし	[in] Common::BasicType::Bytes* pReceptionBuffer 受信した文字列へのポインタ

5-9 Counter クラス

呼び出された回数をカウントするクラスです。時間の計測に使うことを想定しています。

【使用例】

```
#include "Unit/Clock/Counter.h"

Unit::Clock::Counter *pCounter;
//================ ms カウンタの生成 ======================
pCounter=new Unit::Clock::Counter();
if(pCounter==NULL) goto ERROR;
bRet=pCPUBoard->addTimer(Device::CPU::Timer::TI_H8_8BIT_0, 992000UL,Device::CPU::IP_H8_HIGH,&pTimer
                         ,Device::CPU::Timer::TM_FREE_RUN,pCounter);      // 約 1ms 周期で Perform を実行
if(bRet==false) goto ERROR;
```

【Counter の操作の説明】

操作	ノート	パラメータ
Public Counter()	コンストラクタ @return なし	
Public ~Counter()	デストラクタ @return なし	
Public void perform()	Timer から一定周期で呼び出される。呼び出されるたびに、1カウントする。 @return なし	
Public Common::TypeDef::UFourByte getCount()	カウントを取得する。経過した時間は、getInterval() ×getCount( )ms となる @return なし	
Public void setInterval()	Inteval を設定する TimerBase::init から呼び出される @return なし	[in] Common::TypeDef::UFourByte interval (ns)
Public Common::TypeDef::UTwoByte getInterval()	Inteval を返す @return interval (ms)	

5-10 Cyclometer クラスと CyclometerListener インターフェイス

【Cyclometer クラス】

2 相のインクリメンタル型のロータリエンコーダのパルスをカウントする。

ロータリエンコーダの進み側、遅れ側2つのパルスの排他論理和をとった信号の割り込みハンドラとして登録することを前提に設計されている。

【CyclometerListener インターフェイス】

Increment メソッド,decrement メソッドは、Cyclmeter クラスがカウントアップするタイミングで呼び出される。ユーザは、このインターフェイスを継承したクラスのそれぞれのメソッドに、カウントアップのタイミングで実行したい処理を記述することができる。

【使用例】

```
#include "Unit/Cyclometer/Cyclometer.h"
#include "Unit/Cyclometer/CyclometerListener.h"
・・・
Unit::Cyclometer::Cyclometer* pLeft;
```

```
Unit::Cyclometer::Cyclometer* pRight;
PathRecorder* pPRLeft;              // CyclometerListener インターフェイスを継承したクラス
PathRecorder* pPRRight;             // CyclometerListener インターフェイスを継承したクラス

//================ 回転計の初期化と生成 ======================
pLeft=new Unit::Cyclometer::Cyclometer(pRLead,pRLag,pPRRight);
if(pLeft==NULL) goto ERROR;
pCPUBoard->getCPU() ->setInterruptHandler(Device::CPU::CPU::I_H8_IRQ4,pLeft,Device::CPU::CPU::IRQS_BOTH_EDGE);

if(bRet==false) goto ERROR;
pRight=new Unit::Cyclometer::Cyclometer(pLLead,pLLag,pPRLeft);
if(pRight==NULL) goto ERROR;
pCPUBoard->getCPU() ->setInterruptHandler(Device::CPU::CPU::I_H8_IRQ5,pRight,Device::CPU::CPU::IRQS_BOTH_EDGE);
```

【Cyclometer の操作の説明】

操作	ノート	パラメータ
Public Cyclometer()	コンストラクタ @return なし	[in] Device::CPU::IOPin* pLead 2 相のインクリメンタル型の進み側のパルス入力に接続した IOPin を指定する。 [in] Device::CPU::IOPin* pLag 2 相のインクリメンタル型の遅れ側のパルス入力に接続した IOPin を指定する。 [in] Unit::Cyclometer::CyclometerListener* pListener リスナへのポインタを指定する。正のカウント、負のカウントごとにそれぞれ Listener の increment、decrement メソッドを呼び出す。
Public ~Cyclometer()	デストラクタ @return なし	
Public Common::TypeDef::FourByte getCount()	カウントを取得する @return 現在のカウントを返す。	
Private void handler()		
Private Common::TypeDef::OneByte getPhase()		[in] Common::TypeDef::UOneByte lead [in] Common::TypeDef::UOneByte lag

【CyclometerListener の操作の説明】

操作	ノート	パラメータ
Public CyclometerListener()	コンストラクタ @return なし	
Public ~CyclometerListener()	デストラクタ @return なし	
Public void Pure increment()	Cyclometer がカウンタを＋1するときに呼び出される @param なし	
Public void Pure decrement()	Cyclometer がカウンタを－1するときに呼び出される	

	@param なし	

5-11 SpeedMeter クラス

Cyclometer クラスのカウント結果を一定周期で読み取り、走行速度を算出します。

【使用例】

```
#include "Unit/Cyclometer/SpeedMeter.h"

Unit::Cyclometer::SpeedMeter* pSMLeft::
//================ 速度計の生成 ======================
pSMLeft=new Unit::Cyclometer::SpeedMeter(pLeft,15184); // 1 カウント 15.184mm=π×58/12
if(pSMLeft==NULL) goto ERROR;
bRet=pCPUBoard->addTimer(Device::CPU::Timer::TI_H8_16BIT_3,50UL*100000UL,Device::CPU::IP_H8_HIGH,&pTimer,Devi
ce::CPU::Timer::TM_FREE_RUN,pSMLeft);          // 5ms 周期で Perform を実行
if(bRet==false) goto ERROR;
```

【Counter の操作の説明】

操作	ノート	パラメータ
Public SpeedMeter()	SpeedMeter のコンストラクタ @return なし @param meter 回転計	[in] Unit::Cyclometer::Cyclometer* pMeter [in] Common::TypeDef::UTwoByte unitLength (um) 1/1000mm の単位で1カウントあたり進む距離
Public ~SpeedMeter()	デストラクタ @return なし	
Public void perform()		
Public Common::TypeDef::TwoByte getSpeed()	速度を取得する @return 現在のスピード(mm/s) 現在のスピードとは、最後にカウントが入ったときから、今回周期までの平均速度である。前回周期 2カウント、今回周期 4カウントの場合、Speed=4 カウント分の移動距離/1 周期の時間 前々回周期 2カウントの場合、前回周期 1カウント、今回周期 0カウントの場合 Speed=1 カウント分の移動距離/2 周期の時間	
Public void setInterval()	Inteval を設定する TimerBase::init から呼び出される @return なし	[in] Common::TypeDef::UFourByte interval (ns)

5-12 MotorDriver クラス

左右のモータの出力を PWM 制御により制御するクラスです。

【使用例】

```
#include "Unit/MotorDriver/MotorDriver.h"

Unit::MotorDriver::MotorDriver* pMd;
//================ Drive の生成 ========================
pMd=pCPUBoard->getMotorDriver();
m_pMd->right(300);                  // 右モータの出力を PWM 制御により30%にする
m_pMd->left(500);                   // 左モータの出力を PWM 制御により50%にする
```

【Counter の操作の説明】

操作	ノート	パラメータ
Public MotorDriver()	コンストラクタ @return なし	[in] Device::CPU::PWM* pPWM PWM クラスへのポインタを指定する [in] Device::CPU::Timer::PWMOutputPin rightForward 右車輪用 DC モータが前進方向に回転させる H ブリッジの入力信号を PWM 出力端子 ID で指定する。 [in] Device::CPU::Timer::PWMOutputPin rightBackward 右車輪用 DC モータが後進方向に回転させる H ブリッジの入力信号を PWM 出力端子 ID で指定する。 [in] Device::CPU::Timer::PWMOutputPin leftForward 左車輪用 DC モータが前進方向に回転させる H ブリッジの入力信号を PWM 出力端子 ID で指定する。 [in] Device::CPU::Timer::PWMOutputPin leftBackward 左車輪用 DC モータが後進方向に回転させる H ブリッジの入力信号を PWM 出力端子 ID で指定する。 [in] Common::TypeDef::UTwoByte maxDuty
Public ~MotorDriver()	デストラクタ @return なし	
Public void right()	右モータの出力を設定。 @return なし	[in] Common::TypeDef::TwoByte duty 1 to 1000 : 前進　duty -1 to -1000: 後進　0:停止
Public void left()	左モータの出力を設定。 @return なし	[in] Common::TypeDef::TwoByte duty 1 to 1000 : 前進　duty -1 to -1000: 後進　0:停止
Public Common::TypeDef::TwoByte getRight()	右モータの出力を取得。 @return モータの出力	
Public Common::TypeDef::TwoByte getLeft()	左モータの出力を取得。 @return 左モータの出力	
Private void getOutPut()		[in] Common::TypeDef::TwoByte duty [in] Common::TypeDef::UTwoByte& forward [in] Common::TypeDef::UTwoByte& backward
Private		[in]

Common::TypeDef::TwoByte getDuty ()		Common::TypeDef::UTwoByte forward [in] Common::TypeDef::UTwoByte backward

5-13 Format クラス

Printf 形式の文字列の整形をするクラスです。

【使用例】

```
#include "Common/BasicType/Format.h"

Common::BasicType::Format *pFormat;
//================ LCD への速度の表示 ========================
pFormat=new Common::BasicType::Format ("SpeedR%-4d L%-4d");
pFormat->push_back (pSMRight->getSpeed ());
pFormat->push_back (pSMLeft->getSpeed ());
pLCD->cursor (0,1);
pLCD->print (*pFormat);
```

【Format の操作の説明】

操作	ノート	パラメータ
Public Format ()	このクラスは、コンストラクタで format を指定した後、%x,%d,%s で指定した場所に埋める値を push_back で指定する。 その後に getStrig で format 後の文字列を取り出す @return なし @param format フォーマットを printf 形式のサブセットで指定する。 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 数値または文字列がフィールド幅に満たない場合スペースを埋める また、数値または文字列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（フラグに-指定がないかぎり左詰め）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） 変換文字 d:10 進数に変換する x:16 進数に変換する s:文字列を出力する	[in] char* pFormat
Public ~Format ()	デストラクタ @return なし	
Public void push_back ()	コンストラクタで指定した format の%x,%d で指定した場所に埋める値を指定する。 @return なし	[in] Common::TypeDef::UFourByte data %x,%d で指定した場所に表示したい数値を指定する
Public void push_back ()	コンストラクタで指定した format の%s で指定した場所に埋める値を指定する。 @return なし	[in] char* data %s で指定した場所に表示したい文字列を指定する
Public void	フォーマット後の文字列を取得する	[in]

getString ()	@return なし	Common::BasicType::Bytes& str フォーマット後の文字列への参照
Public Common::TypeDef::UTwoByte 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	
Public void clear ()	push_back したデータをクリアする @return なし	

5-14 周期起動タスクと IperformerインターフェイスIperformer

周期起動タスクの実装方法です。

【使用例】

```
#include "OsWrapper/Task/IPerformer.h"

LineSensor* pLineSensor;              // Iperformer インターフェイスを継承したクラス
//  周期起動タスクの生成
pLineSensor = new LineSensor(pADC);
OsWrapper::OSFactory::getInstance()->createCyclicTask(0,pLineSensor,OsWrapper::Task::Task::TP_Blink,5);// 5ms 間隔
```

【OSFactory の操作の説明】

操作	ノート	パラメータ
Public bool createCyclicTask ()	周期起動タスクを生成する @return true 成功 false 失敗 指定した周期で IPerformer の perform を呼び出す。 複数のタスクが同時に呼び出される場合、プライオリティの高いタスクの perform から呼び出す。 ノンプリエンプティブの擬似マルチタスクなので、タスクの実装にあたってはすべてのタスクの perform 関数の実行が EE_Type_LT の init 関数で指定した OS タイマの時間以内に完了するよう注意する必要がある。 実行時間の長い処理は、別途 TimerRegisterable として実装する必要がある。	[in] Common::TypeDef::UFourByte id [in] OsWrapper::Task::IPerformer* pIPerformer [in] Common::TypeDef::UTwoByte priority 【プライオリティ】 TP_TimeCritical 高いプライオリティのタスク TP_General 一般のタスク TP_Blink 低いプライオリティのタスク [in] Common::TypeDef::UFourByte cycle ms で起動周期を指定する。
Public OSFactory* getInstance ()	OSFactory のインスタンスを取得する	

【IPerformer の操作の説明】

操作	ノート	パラメータ
Public IPerformer ()	コンストラクタ @return なし	

Public ~IPerformer ()	デストラクタ @return なし	
Public void Pure perform ()	ユーザーは、このインターフェイスを継承したクラスのこのメソッドに周期起動する処理を記述する	[in] OsWrapper::Task::Task* pTask pTask

5-15 TimerRegisterable インターフェイス

タスクよりも厳密に周期実行したい処理や処理時間が OSTimer の周期以内に収まらない処理を記述する方法です。

内部では CPU の Timer の割り込みハンドラとして perform メソッドを登録しています。

【使用例】

```
#include "Device/CPU/TimerRegisterable.h"

pDrive = new Drive(pDsFSM,pLineSensor,pMd,pSMLeft,pSMRight);        // TimerRegisterable を継承したクラス
bRet=pCPUBoard->addTimer(Device::CPU::Timer::TI_H8_16BIT_2,50UL*300000UL,Device::CPU::IP_H8_LOW,&pTimer,Devic
e::CPU::Timer::TM_FREE_RUN,pDrive);                 // 15ms 周期で Perform を実行
```

【TimerRegisterable の操作の説明】

操作	ノート	パラメータ
Public TimerRegisterable ()	コンストラクタ @return なし	
Public ~TimerRegisterable ()	デストラクタ @return なし	
Public void Pure perform ()	ユーザーは、このインターフェイスを継承したクラスのこのメソッドに周期起動する処理を記述する	
Public void Pure setInterval ()	Inteval を設定する TimerBase::init から呼び出される @return なし	[in] Common::TypeDef::UFourByte interval (ns)

5-16 Bytes クラス

Byte 列を格納するクラスです。文字列の格納にも使います。

【使用例】

```
#include "Common/BasicType/Bytes.h"
・・・・
bool DumpCommand::check(const Common::BasicType::Bytes* command) {
        bool bRet=false;
        if(command->size()>=1)
        {
                if( ( (*command)[0]=='d') || ((*command)[0]=='D') )
```

```
                    {
                    bRet=true;
                    }
            }
            return bRet;
}
```

【TimerRegisterable の操作の説明】

操作	ノート	パラメータ
Public ~Bytes ()	デストラクタ @return なし	
Public Bytes ()	コンストラクタ @return なし	[in] unsigned int size [in] bool bResizeable True:リサイズを許可する False:リサイズを許可しない
Public Bytes ()	コピーコンストラクタ @return なし	[in] Bytes& rhs
Public Bytes ()	コンストラクタ @return なし	[in] char* str
Public Common::TypeDef::UTwoByte size ()	格納した Byte 列のサイズを取得する	
Public Common::TypeDef::UOneByte& operator [] ()	Bytes クラスに格納した Byte 列を、1Byte の配列とみなし [] でアクセスする	[in] Common::TypeDef::UTwoByte idx
Public Bytes& operator= ()	代入演算子	[in] Bytes& rhs
Public void clear ()	格納した Byte 列をクリアする	
Public bool resize ()	resize 成功 true resize 失敗 false 格納領域を拡張する。	[in] Common::TypeDef::UTwoByte size
Public bool push_back ()	1byte を最後に追加する @return true 成功 false 失敗	[in] Common::TypeDef::UOneByte data
Public bool push_back ()	byts 型を連結する @return true 成功 false 失敗	[in] Bytes& data
Public bool push_back ()	文字列を連結する	[in] char* str [in] Common::TypeDef::UTwoByte size
Public	最初に見つかったインデックスを返す 見	[in]

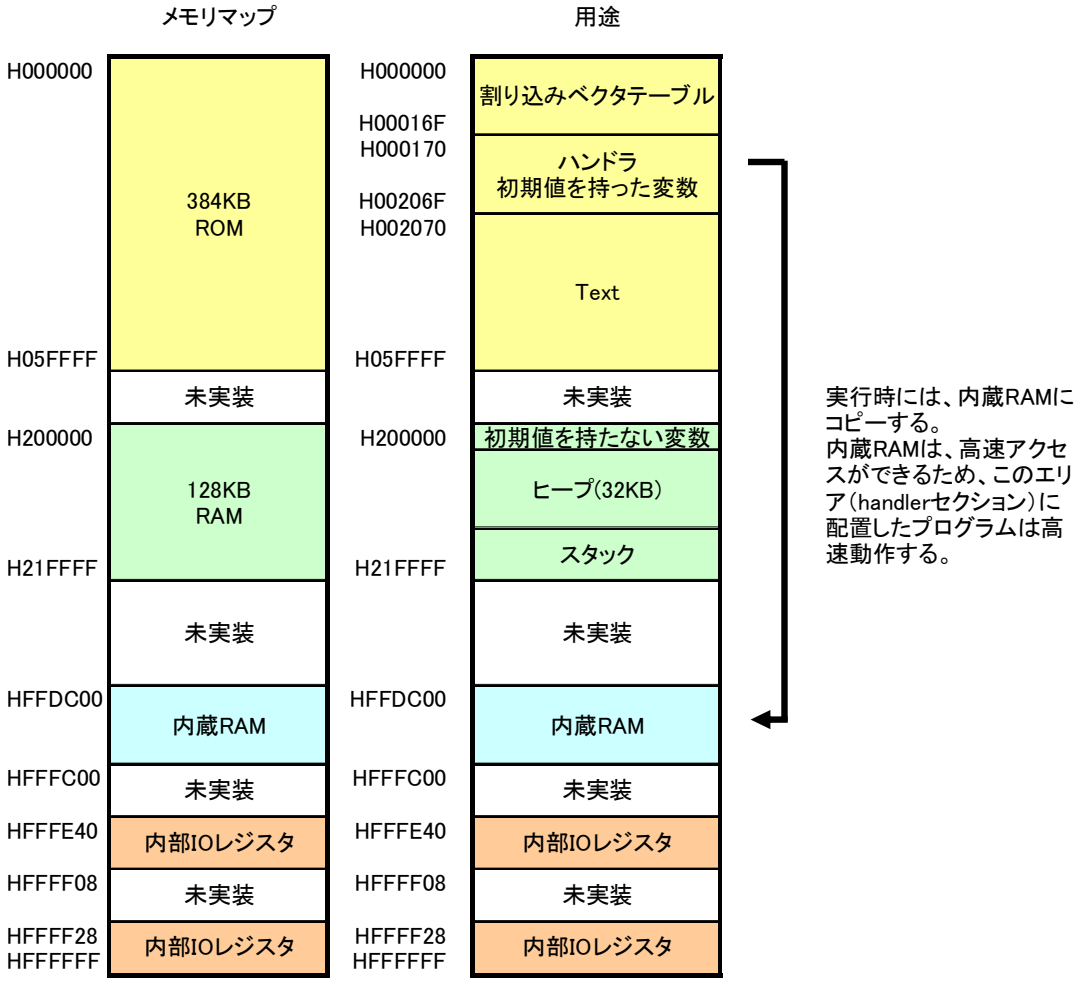
Common::TypeDef::UTwoByte find ()	つからなかったときは、0xffff を返す	Common::TypeDef::UOneByte data [in] Common::TypeDef::UTwoByte firstIdx
Public Common::TypeDef::UTwoByte find ()	最初に見つかったインデックスを返す 見つからなかったときは、0xffff を返す	[in] Bytes& rhs
Public Common::TypeDef::UTwoByte count ()	dataで指定した値がいくつあるかを数える @return data で指定した値の数	[in] Common::TypeDef::UOneByte data
Public void cutTheBack ()	前から index 個のデータを残し、後ろの要素を捨てる。	[in] Common::TypeDef::UTwoByte index
Public void cutTheFront ()	前から index 個のデータを捨てる。	[in] Common::TypeDef::UTwoByte index
Public bool hexToValue ()	index から始まる文字列を 16 進数とみなし値に変換する	[in] Common::TypeDef::UFourByte& value [in] Common::TypeDef::UTwoByte index [in] Common::TypeDef::UOneByte maxDigits
Public bool decToValue ()	index から始まる文字列を 10 進数とみなし値に変換する	[in] Common::TypeDef::UFourByte& value [in] Common::TypeDef::UTwoByte index [in] Common::TypeDef::UOneByte maxDigits
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] Common::TypeDef::UTwoByte index 調査対象の要素のインデックス

6　ハードウエア/ソフトウエアについて

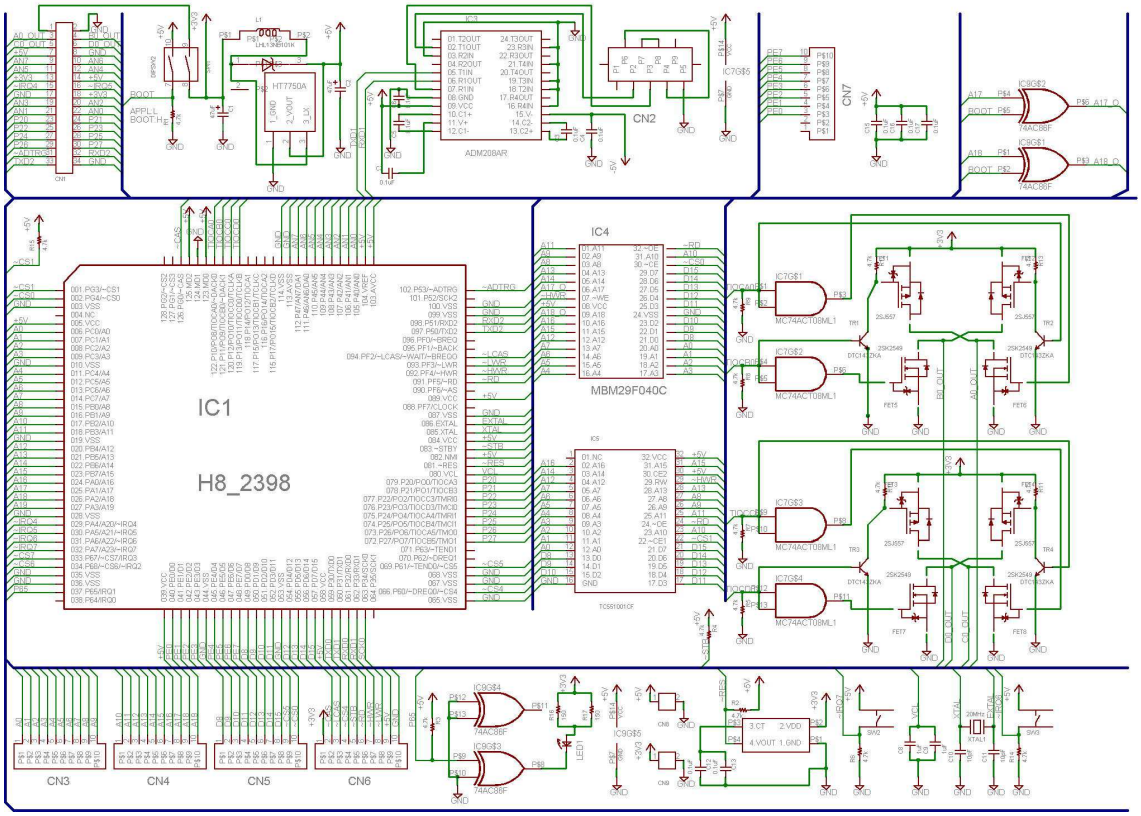
6-1　メモリマップ

EE_TypeLT では、H8S　2398 をモード5で使っています。

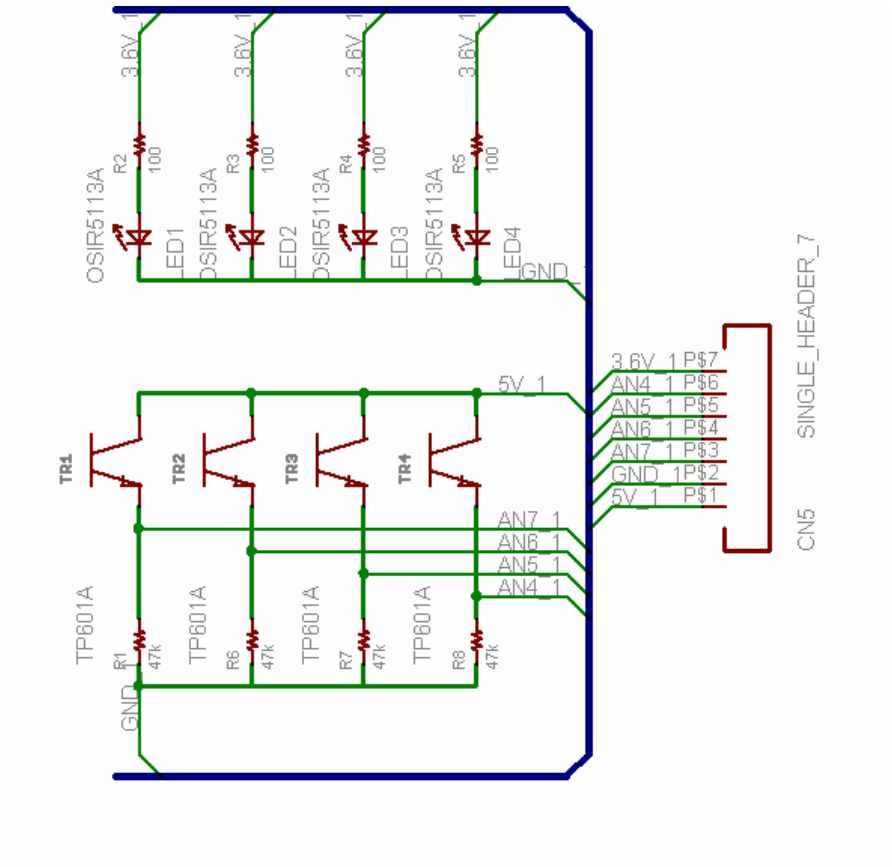
ユーザモードで起動した時のメモリマップを以下に示します。



6-2 回路図



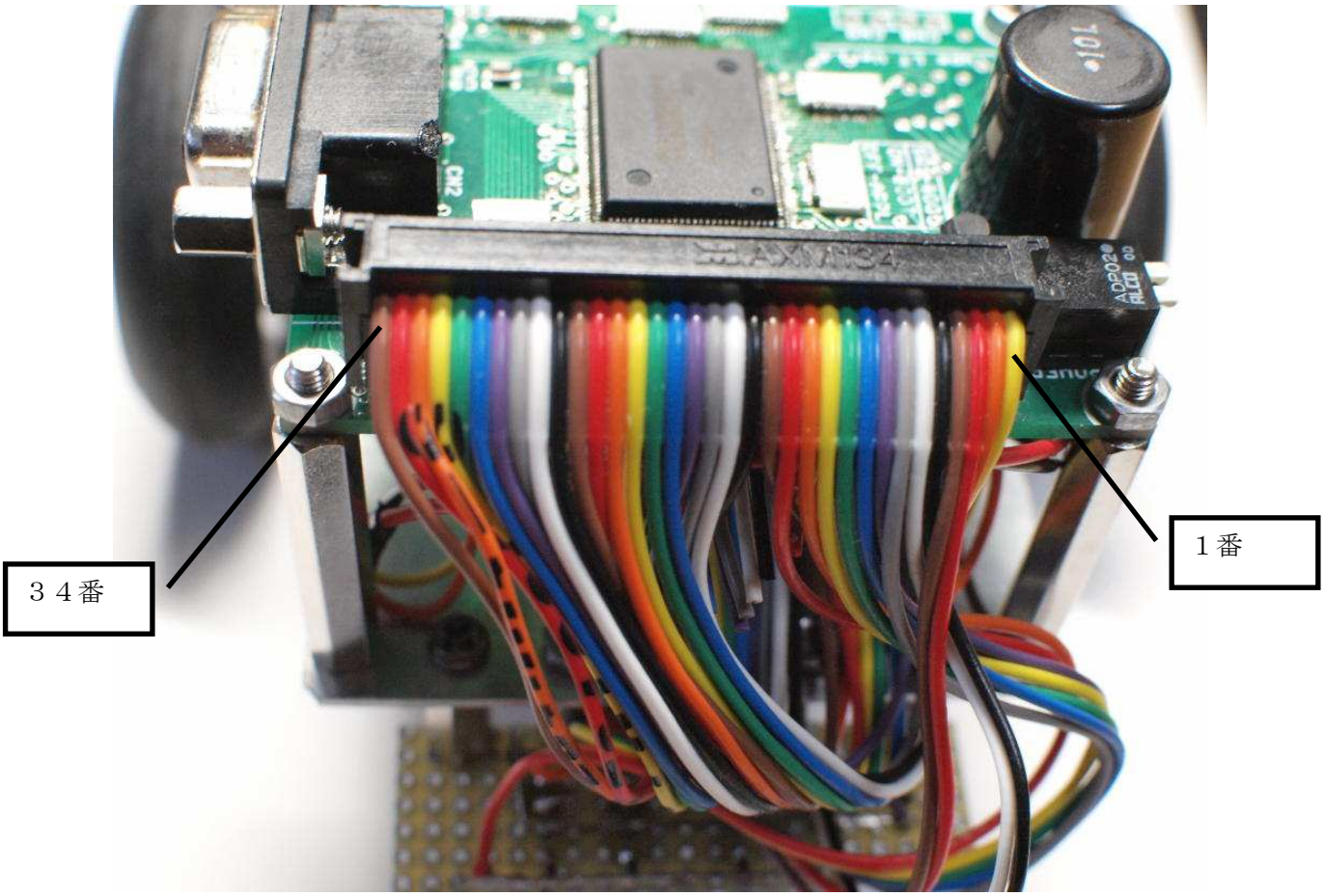
EE_TypeLT の回路図



ラインセンサの回路図

6-3 配線

EE_TypeLT とライントレーサは、下記のように配線します。

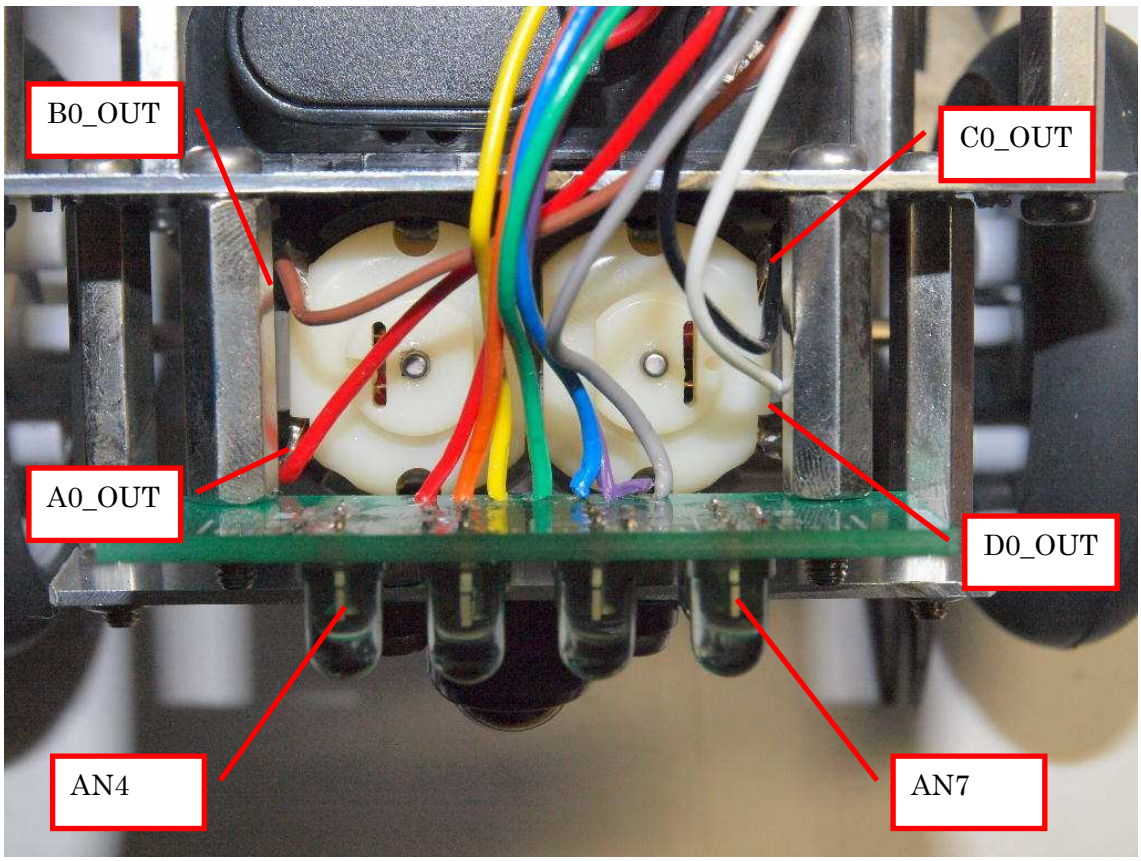


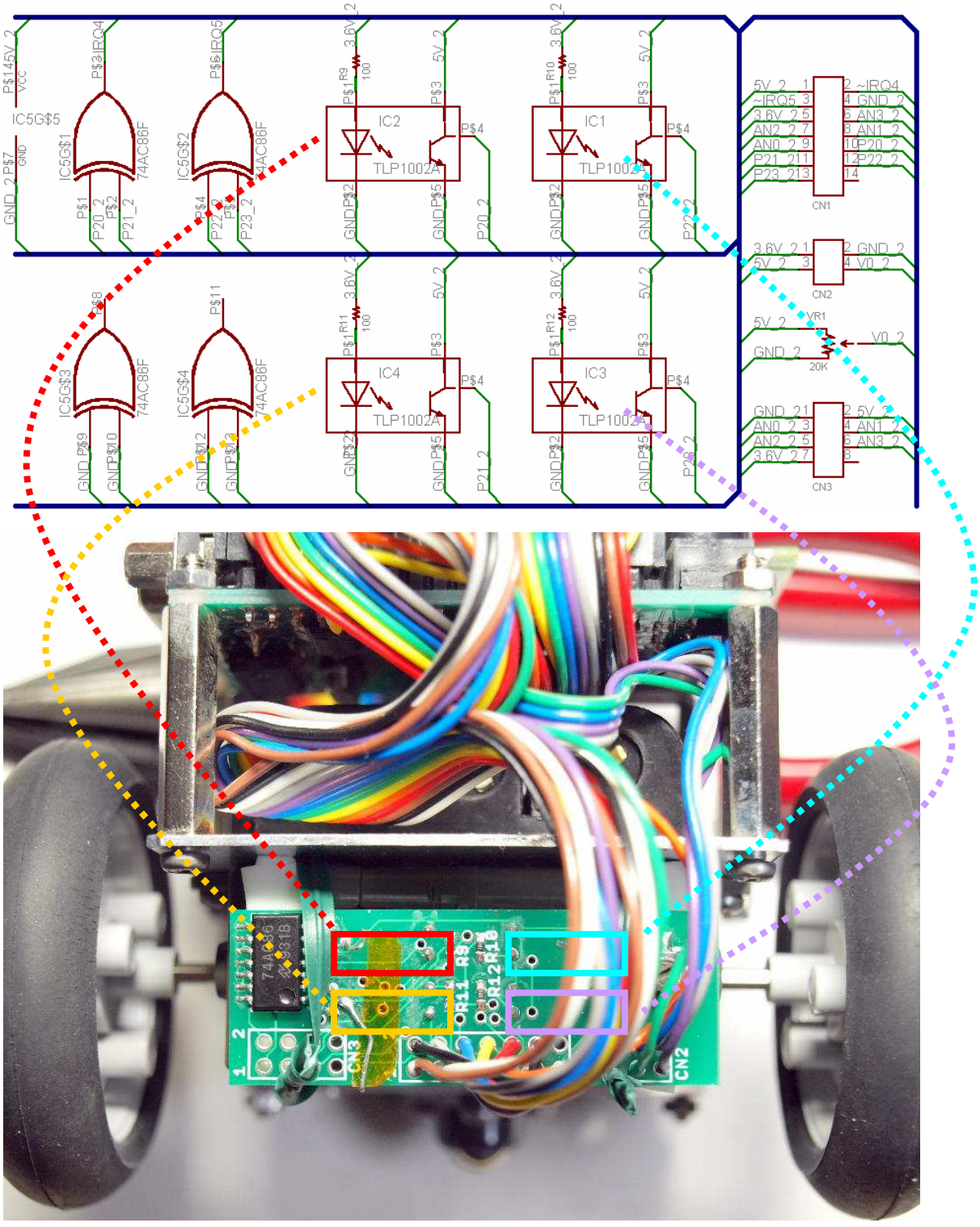
CN1 の写真です。

右側が1番、左側が34番ピンになります。

No	配線の色	信号	No	配線の色	信号
1	黄	5V(入力)	18	紫	3.6V
2	橙	GND(入力)	19	青	AN3/$P4_3$
3	赤	AO_OUT	20	緑	AN2/$P4_2$
4	茶	BO_OUT	21	黄	AN1/$P4_1$
5	黒	CO_OUT	22	橙	AN0/$P4_0$
6	白	DO_OUT	23	赤	$P2_0$
7	灰	5V	24	茶	$P2_1$
8	紫	GND	25	黒	$P2_2$
9	青	AN7	26	白	$P2_3$
10	緑	AN6	27	灰	$P2_4$
11	黄	AN5	28	紫	$P2_5$
12	橙	AN4	29	青	$P2_6$

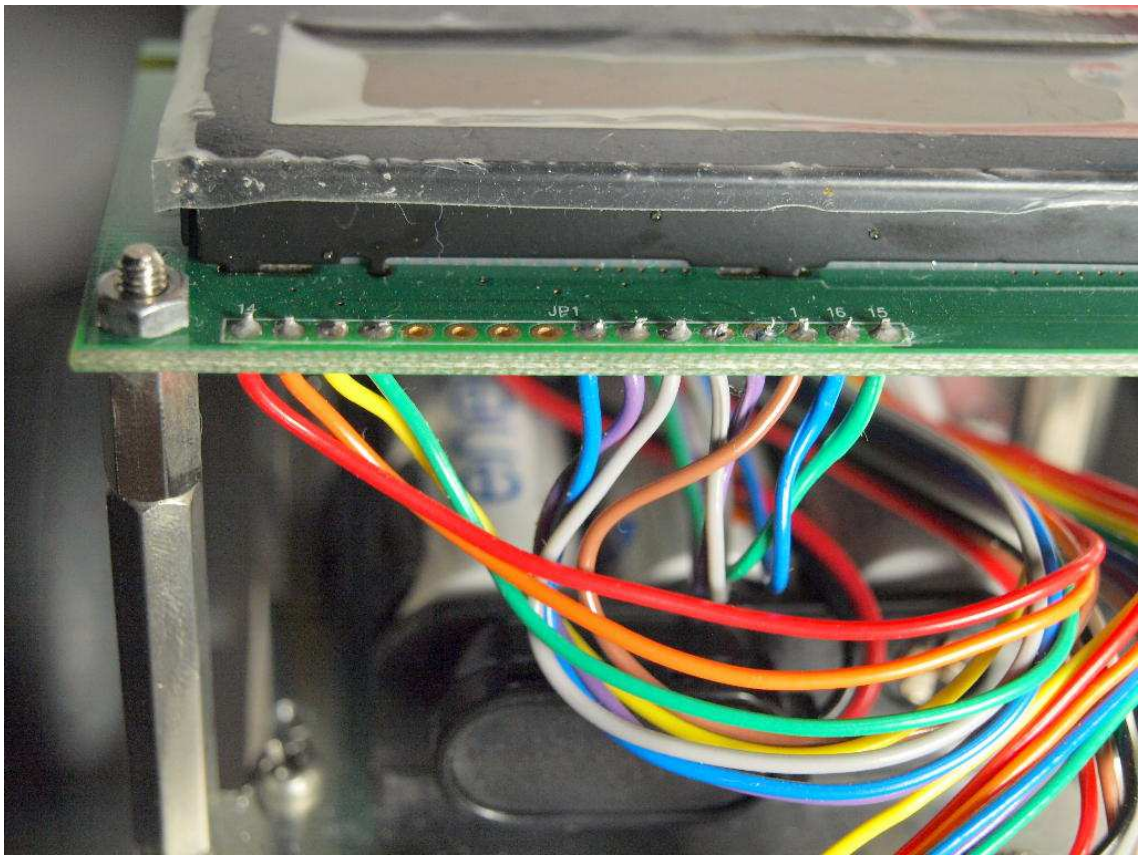
13	赤	3.6V	30	緑	$P2_7$
14	茶	5V	31	黄	~ADTRG/ $P5_3$
15	黒	~IRQ4	32	橙	RXD2/ $P5_1$
16	白	~IRQ5	33	赤	TXD2/ $P5_0$
17	灰	GND	34	茶	GND




IC5G$5
IC5G$1
IC5G$2
IC5G$3
IC5G$4
74AC86F
IC2
IC1
IC4
IC3
TLP1002A
CN1
CN2
CN3
VR1
20K
R11
R12
CN3
CN2

● LCD（LC4216 ： ST7066U 使用標準キャラクタディスプレイ）

PinNo	LCD の信号名	配線	PinNo	LCD の信号名	配線
1	Vss	GND	10	DB3	NC
2	VDD	5V	11	DB4	$P2_7$
3	Vo	コントラスト(VR と結線)	12	DB5	$P5_3$
4	RS	$P2_4$	13	DB6	$P5_1$
5	R/W	$P2_5$	14	DB7	$P5_0$
6	E	$P2_6$	15	VLED+	3.6V
7	DB0	NC	16	VLED－	GND
8	DB1	NC	BL1	VLED+	NC
9	DB2	NC	BL2	VLED-	NC



7　参考文献

1．OOP Foundations 「組込み UML」翔泳社、2002年

２．RENESAS 「H8S/2357 グループ、H8S/2357F-ZTAT TM、H8S/2398F-ZTAT TM ハードウェアマニュアル」RENESAS、２００４年９月１７日