EveryStamp開発環境解説

エブリセンスジャパン株式会社

*EveryStamp*開発環境解説

エブリセンスジャパン株式会社

(2016)

EveryStamp開発環境解説

このドキュメントはEveryStampのソースコードにアクセスし、自分用のEveryStampを作るための方法について解説します。

このドキュメントを読むことで可能になることは、

- 配布ソースコードからEveryStamp用バイナリを生成し書き込む
- EveryStampの動作を理解する
- EveryStampが対応している以外のセンサーの対応コードを書き、組み込む

です。以下の点については、本ドキュメントの対象外とします。

- EveryStampを本来の目的以外にも使えるようにする
- バイナリで配布されている部分の動作の詳細の解説
- 独自開発したEveryStampバイナリを配布する

配布物の説明と開発環境について

EveryStamp開発キットは、以下のものから成ります。

- (配布ハードウェア)
- ソースコード
- バイナリライブラリ
- 本解説

また、以下のものが必要ですが、別途用意して下さい。

- ARM Cortex M3用ツールチェイン
- 上記の動く環境
- その他の開発ルール

本解説では、Intel/AMD 64bit環境でUbuntu 14.04を使った環境を前提としています。同様の環境はRaspberry Piの上でも用意することが可能です。

開発環境の構築

本解説では、Intel/AMD 64bit上のUbuntu 14.04を前提として解説します。これ以外の環境では自己責任の下、適宜読み替えをして下さい。

必要ハードウェア

CPUは本解説ではIntel/AMD 64bitを前提としていますが、32bitでも特に問題はありません。また、RaspberryPi(ARM Cortex A)でも特に大きな問題はありません。なお、RaspberryPiを使う場合は、電源容量と電源の安定には十分注意をして下さい。これらに問題があると、プログラム書き込みが正常に行えない場合があります。

メモリは1GB程度あれば十分です。

ハードディスクは全てを合わせて8GB程度あれば十分なようです。

VM上に構築することも可能です。この時、USBはゲストOSからアクセス出来るように設定して下さい。この場合、パススルーさせるデバイスは、「FTDI 2232」です。成功すれば、`/dev/ttyUSB`のデバイスが2つ作られます。

ツールチェインの入手

ツールチェインはLaunchpadのものを使っています。このため、PPAにLaunchPadのリポジトリを追加する必要があります。

```
$ sudo add-apt-repository ppa:terry.guo/gcc-arm-embedded
```

この後、

```
$ sudo apt-get update
$ sudo apt-get install gcc-arm-none-eabi
```

として、ツールチェインを取得します。

もし何らかの理由でUbuntu標準のARM用ツールチェインが入っていた場合、それらを削除してからPPAの追加以降の処理をして下さい。

ほとんどの場合、Ubuntu標準のARM用ツールチェインでも問題は起きませんが、いくつかの問題が確認されていますので、LaunchPadのものを使うようにして下さい。LaunchPadのツールチェインは他のARM CPUでの開発、例えばStampExpanderのソースをいじる場合にも有用です。

OpenOCDの入手

EveryStamp開発ボードを開発用パソコン(以下「母艦」)につなぐためには、USBインターフェイスを使います。USB経由で開発ボードを動かすために、OpenOCDが必要になります。

OpenOCDはUbuntu標準のものを使います。

```
$ sudo apt-get install openocd
```

で入手出来ます。

本ドキュメントではOpenOCDの使い方についての詳細には触れず、EveryStamp開発環境で必要なことのみを説明しますので、より詳しいことは他のドキュメントに当たって下さい。その際、Ubuntu標準のOpenOCDはVer 0.9.0以降の版であることに留意して下さい。世の中のOpenOCDの解説の多くは、Ver 0.8以前の版についてのものが多くあります。具体的には、開発ボードを動かすための設定ファイルの記述が、Ver 0.9以降とそれ以前ではディレクティブが異なります。

その他に必要なもの

その他に必要なものは、以下のとおりです。

- Python(Ubuntu標準)
- Ruby(Ubuntu標準)
- シリアル通信に使うターミナルソフト(たとえばminicom)

これらは特に細かい指定はありませんので、適当に好みのものを入れて下さい。

リリースファイルの入手と確認

リリースファイルの入手と開発環境が正しく行えたことの確認を行います。

リリースファイルの入手

リリースファイルは

```
https://developer.every-sense.com/everystamp/release/
```

から入手します。

現在のところ、ソースコードも含めて、全てがプロプライエタリです。そのため、

- EveryStamp用バイナリの作成
- センサーへのアクセスの参考

以外の用途には使うことも再配布することも出来ません。詳しいことは、ライセンスをご覧下さい。この制限は、将来変更される予定があります。

ファイルの展開とディレクトリ構造

ファイルの展開は、通常のtarと同じように、

```
$ tar xfz everystamp-????.tar.gz
```

のように行います。????の部分にはバージョン情報が入ります。

展開すると、以下のようなディレクトリが展開されます

```
--\
   +--\ bindist     バイナリで提供される実行形式
   +--\ libs        バイナリで提供されるライブラリとオブジェクト
   +--\ obj         オブジェクトの作られるディレクトリ(配布時には空です)
   +--\ src         ソースファイル
   +--\ tools       ツール類
   +--\ www         設定のウェブページで使うファイル
   +--- layout      EveryStampのフラッシュメモリのレイアウト情報
   +--- Makefile    Makefile
```

確認

開発環境の準備が完了しファイルの展開が出来たら、それらが正しいかどうかの確認を行います。確認のために、実際にEveryStampのバイナリを作成し、EveryStampに書き込んでみます。

まず、`make`します。

```
$ cd everystamp-????
$ make
```

正しく環境が作られていれば、makeは問題なく終了します。

次に、開発ボードとパソコンとの接続を確認します。

開発ボードをUSBでパソコンと接続して電源を入れ、

```
$ minicom -D /dev/ttyUSB1 115200
```

のように入力します。

開発ボードが正しく接続されていれば、画面上にいろいろなメッセージが流れているはずです。もしそうでなければ、

- 電源は入っているか(ボード上のスライドスイッチは左、EveryStampボード上のスライドスイッチは右がONです。正しくONになると、LEDが点灯します)
- USBに他のシリアルデバイスはつながっていないか(その場合、`/dev/ttyUSB`の後の数字が別のものになります)
- 開発ボードは認識されているか(dmesg等を見て下さい)

などを確認してみて下さい。

その次に、JTAGの書き込みを確認します。

LEDが点灯している状態で、

```
$ sudo make everystamp_ramload
```

と入力します。

```
openocd -s ?????/everystamp-????/tools -f interface/ftdi.cfg -f openocd.cfg -c \
init -c "load bin/everystamp.axf `arm-none-eabi-readelf -h bin/everystamp.axf \
| grep "Entry point" | awk '{print $4 }'`" -c shutdown
Open On-Chip Debugger 0.9.0-dev-00268-ga9c90a0 (2015-02-05-12:00)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.sourceforge.net/doc/doxygen/bugs.html
adapter speed: 5000 kHz
adapter_nsrst_delay: 100
Error: session transport was not selected. Use 'transport select <transport>'
Info : session transport was not selected, defaulting to JTAG
jtag_ntrst_delay: 100
cortex_m reset_config sysresetreq
sh_load
Info : clock speed 5000 kHz
Info : JTAG tap: mc200.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver:
0x4)
Info : mc200.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : JTAG tap: mc200.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver:
0x4)
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00000fb8 msp: 0x20010400
requesting target halt and executing a soft reset
Warn : soft_reset_halt is deprecated, please use 'reset halt' instead.
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00000fb8 msp: 0x20010400
194900 bytes written at address 0x00100000
2364 bytes written at address 0x20000000
downloaded 197264 bytes in 1.427483s (134.951 KiB/s)
verified 197264 bytes in 0.228720s (842.255 KiB/s)
shutdown command invoked
```

このように表示されれば成功です。何らかのエラーの場合は、接続等確認してみて下さい。

ここまで出来ていれば、EveryStampの開発環境の構築は完了です。

APIとサンプルコードの解説

以下では、配布されたEveryStampのコードを元に、センサードライバの書き方について解説します。

EveryStampソフトウェアの構造

EveryStampのソフトウェアはだいたい以下のような構造をしています。

ほとんどの処理は、「イベントループ」のスレッドで行われますが、「省電力モード」でない場合の時計処理は専用のスレッドで行われます。

EveryStampが起動すると、各種初期設定の後、「センサードライバ」のセンサー初期化関数が呼び出されます。これにより、センサーの初期化等を行います。実際のセンサーが装備されていない場合、センサーの初期化は失敗しますが、これによって当該センサーが存在しないことを検知します。センサーを検知したかどうかは、「設定モード」の「センサーの状態」で確認することが出来ます。

「計測時刻(設定項目にあります)」が来た場合、「イベントループ」から「センサードライバ」が呼び出され、計測が行われます。取得されたデータはメモリ上に保存され、「送信時刻(設定項目にあります)」が来た時にネットワークに接続され送信されます。省電力モードで動作している場合は、送信と同じタイミングで時計合わせが行われます。

「イベントループ」と「センサードライバ」の情報のやりとりは、「SensorInfo」という構造体で行われます。

省電力モードの場合、「イベントループ」は一定時間以上動作するべきイベントが発生しないことがわかれば、ハードウェアを省電力状態にして次のイベント(計測または送信)を待ちます。ハードウェアが省電力状態になる前には「センサードライバ」の省電力のための関数を呼び出して、センサーを省電力状態にすることもあります。

EveryStampの通常動作は、この省電力モードですが、装備しているセンサーや計測の性質によっては省電力モードになるのが好ましくない

場合があります。この場合は、「連続計測モード」にすることも可能です。ただし、連続計測モードでは、

- 消費電力の増大(電池の持ちが悪くなります)
- 発熱量の増大(温度センサーの計測値が正しくなくなります)

という問題がありますので、その辺は注意して下さい。

SensorInfo

既に述べたように、「イベントループ」と「センサードライバ」の間のインターフェイスは、SensorInfoという構造体です。src/os_sensors.hの中で以下のように定義されています。

```
#ifndef _INC_SENSOR_IO_H_
#define _INC_SENSOR_IO_H_

typedef Bool    (*sensor_init_func)(void);
typedef uint8_t *(*sensor_get_func)(void);
typedef Bool    (*sensor_parse_func)(uint8_t *buff, void *data);
typedef size_t  (*sensor_build_func)(char *p, uint8_t *buff);
typedef char    *(*sensor_human_value)(void);
typedef void    (*sensor_stop_func)(int sec);

typedef struct  {
        char    *name;
        char    *unit;
        uint8_t dimension;
        uint8_t precision;
        uint8_t size;
        sensor_init_func        init;
        sensor_get_func         get;
        sensor_parse_func       parse;
        sensor_build_func       build;
        sensor_human_value      human_value;
        sensor_stop_func        stop;
        Bool    fValid;
}       SensorInfo;

extern  SensorInfo      Sensors[];
extern  int                     nSensors;
extern  void            *SensorWork;

extern  void    InitSensorIO(Bool fInit);

#endif
```

センサードライバを作ることは、この構造体のメンバを埋めるという意味になります。

各フィールドについて概要は以下のとおりです。

name
: センサーの名前です。EverySenseシステムでのセンサー名と同じです。

unit
: 単位の文字列です。EverySenseシステムで通常使われる単位名と同じです。

dimension
: 計測データの次元です。温度や湿度のような1回の計測で得るデータが1つのものは'1'、空間ベクトルを表現するもののように一度に多数のデータを得るものは、その数を指定します。

precision
: 計測データの小数点以下の桁数です。計測データの精度はセンサーによって決まりますから、無闇に桁数を増やしても意味がありません。

size

1回の計測で得たデータを保存するのに必要なデータの大きさ(バイト数)です。計測時刻は別途保存されるので、この数字には含まれません。

init
: センサーおよびセンサードライバを初期化するための関数です。成功すればTRUE, 失敗の場合はFALSEを返します。失敗した場合、当該センサーは存在しないということになり、計測は行われません。

get
: 計測を行い、その結果を保存するための関数です。返り値は取得したデータが保持されている領域のポインタです。

parse
: getで取得したデータをunitで表現される値に変換するための関数です。

build
: センサー値を計測データに変換するための関数です。JSONのコード片を生成します。

human_value
: 「設定モード」の「センサーの状態」で表示するための文字列を生成する関数です。

stop
: センサーおよびセンサードライバを停止させるための関数です。存在しない(必要でない)場合はNULLにします。

fValid
: センサーが有効な場合はTRUEになります。これはセンサードライバの初期化の時に作られますので、センサードライバでは特に何かする必要はありません。

以下では、実際のコードを用いてどのようにこれがを実装するかについて解説します。

BH1750のセンサードライバの解説

以下では、照度センサーであるBH1750の実際のセンサードライバのコードを元に、センサードライバについて説明を行います。

BH1750のSensorInfo

以下はBH1750のSensorInfoのコードです。

```
SensorInfo      BH1750 = {      "BH1750",
                                LIGHT_UNIT,
                                1,
                                0,
                                2,
                                (sensor_init_func)init,
                                (sensor_get_func)get,
                                (sensor_parse_func)parse,
                                (sensor_build_func)build,
                                (sensor_human_value)human_value,
                                NULL
};
```

このうち、LIGHT_UNITは、src/unit.hの中で定義されていますが、

```
#define LIGHT_UNIT                      "lx"
```

となっています。

initの実装

initにはセンサー及びセンサードライバの初期化コードを書きます。

```
static  Bool
init(void)
{
        I2C_device      *i2c_dev;
        Bool    ret;
        uint8_t *buff = SensorWork;

ENTER_FUNC;
        i2c_dev = I2C_open(I2C_ADDRESS_BH1750);

        I2C_write_uint8(i2c_dev, 0x01); //      power on
        sleep_ms(1);
        I2C_write_uint8(i2c_dev, 0x07); //      reset
        sleep_ms(1);
        I2C_write_uint8(i2c_dev, 0x10); //      continuously h-resolution mode
        sleep_ms(10);
        if      ( I2C_read(i2c_dev, buff, 2) == 2 )     {
                ret = TRUE;
        } else {
                ret = FALSE;
        }
        I2C_close(i2c_dev);
LEAVE_FUNC;
        return  (ret);
}
```

I2C_openはI2Cデバイスのオープンの関数で、引数はI2C slaveのアドレスを指定します。この例ではマクロ定義されていますが、src/i2c.hの中で、

```
#define I2C_ADDRESS_BH1750              0x23
```

と定義されています。

I2C_write_uint8は、I2Cデバイスにデータを出力する関数です。

sleep_msは指定した時間(ms単位)sleepします。一般のセンサーは、コマンドを送った後にコマンドを処理する時間が必要です。この時間が経過しないで次のコマンドを送信すると、処理が正しく行われなかったりハングアップすることがあります。これを避けるために、適当にsleepしながらコマンドを送ります。たいていのセンサーは、データシートに必要な時間が書かれていますが、ない場合はcut&tryすることになります。

I2C_readはI2Cデバイスからデータを読む関数です。このコードでは、データが読み込めたら初期化は成功したことにしています。

EveryStampでは、I2C通信のタイムアウトを送信受信共に100msにしています。100ms経過して通信が完了しなかった場合は通信エラーとなります。I2Cデバイスが装備されていない場合、通信は完了しませんので、これを利用してデバイスの有無を検知します。

ENTER_FUNC, LEAVE_FUNCは、printf debugのためのマクロです。デバッグのフラグをつけてコンパイルすると、関数に入ったこと出たことをが、コンソール出力されます。

buffの初期値としてSensorWorkが設定されています。SensorWorkは関数内で有効な128バイトのメモリです。通常は一時的な通信バッファ等として使います。

getの実装

getには、センサーからデータを取得する処理を書きます。

```
static  uint8_t *
get(void)
{
        I2C_device      *i2c_dev;
        uint8_t *buff = SensorWork;

        i2c_dev = I2C_open(I2C_ADDRESS_BH1750);
        I2C_read(i2c_dev, buff, 2);
        dbgprintf("read = %02X %02X", (int)buff[0], (int)buff[1]);
        I2C_close(i2c_dev);

        return  buff;
}
```

BH1750の場合、単にデータを読めばセンサー値が得られます。

SensorWorkから借りて来たbuffを返り値にしています。これはgetの返り値は取得したデータを格納した領域へのポインタだからです。なお、取得したデータの大きさはSensorInfoのsizeというメンバに定義しておきます。現在のところ、この大きさは固定です。

parseの実装

parseには、getで取得したデータを実際のセンサー値に変換する処理を書きます。

```
static  size_t
parse(
        uint8_t *buff,
        float   *data)
{
        *data = (float)unsigned2signed(buff, TRUE) * 0.8333333333;
        dbgprintf("BH1750 %s lx (%02X %02X)", ftos(*data, 0, -1), (int)buff[0],
(int)buff[1]);
        return  (2);
}
```

BH1750の場合、取得した2バイトのデータを2バイトの符号なし整数として読んだものに、0.83333333を乗じたものが、lx(ルクス)となります。

返り値の'2'は、処理したデータの大きさですが、これは現在のところ取得したデータの大きさ(size)と同じですが、必ず返すようにして下さい。

BH1750はdimensionが'1'のセンサーですが、'1'以外のセンサーもあります。たとえば、以下はMPU9250の地磁気センサーのparseです。

```
static  size_t
parse_mag(
        uint8_t *buff,
        float   *data)
{
        float   res;

        res = 1000.0 * 4219.0 / 32760.0;

        data[0] = unsigned2signed(buff + 0, FALSE) * res;
        data[1] = unsigned2signed(buff + 2, FALSE) * res;
        data[2] = unsigned2signed(buff + 4, FALSE) * res;

        return  (18);
}
```

dimensionが'1'以外の場合、このような返し方になります。とは言えparseが呼び出されるのは、現在のところ通常は次に説明するbuildからだけなので、この間でツジツマが合っていれば基本的には問題はありません(保証があるわけでもありませんが)。

buildの実装

buildは送信データの組み立てを行います。

```
static  size_t
build(
        char    *p,
        uint8_t *buff)
{
        float   data;

        parse(buff, &data);

#ifdef  DATA_IS_JSON
        return  sprintf(p, "\"value\":\"%s\",\"unit\":\"%s\"", ftos(data,0,0),
LIGHT_UNIT);
#endif
#ifdef  DATA_IS_CSV
        return  sprintf(p, "%s", ftos(data,0,0));
#endif
}
```

単にJSONのコード片をsprintfで作っているだけです。ここで作っているのは、センサーデータの"data"フィールドのうち、時刻以外の部分です。

上記のコードには、DATA_IS_CSVとして、CSVのコード片を作る処理も書かれていますが、これはなくても構いません。

返り値は組み立てたコード片の長さですが、これはsprintfの返り値と同じです。このため、このコードではsprintfの返り値をそのままreturnするようになっています。

human_valueの実装

human_valueは「設定モード」の「センサーの状態」で表示するための文字列を生成します。

```
static  char    *
```

```
human_value(void)
{
        float   data;
        char    *out = SensorWork;

        parse(get(),&data);
        sprintf(out,"%s</td><td>%s", ftos(data, 10,2), LIGHT_UNIT);
        return  (out);
}
```

だいたいはbuildと同じですが、parseの引数がバッファではなくてget()の返り値そのままである点と、結果が長さではなくて文字列そのものである点が異なります。

BH1750の場合、センサーのデータを取得してその値を表示することに特に困難さがあるわけではないので、センサー値を取得して表示していますが、センサーによっては「設定モード」ではうまく動いてくれないものや、値を表示することが困難なものもあります。その場合は単に「センサーが存在する」という意味の出力をしても構いません。以下はStampExpanderの場合です。

```
static  char    *
human_value(void)
{
        char    *out = SensorWork;

        sprintf(out,"sensor exist");
        return  (out);
}
```

単に"sensor exist"と返すだけになっています。

センサードライバの組み込み

作ったセンサードライバは、イベントループから呼び出されるようにする必要があります。

build_sensors.yamlへの追加

EveryStampで利用するセンサードライバは、`src/build_sensors.yaml`に記述する必要があります。

以下は、Ver 1.0時点での`src/build_sensors.yaml`です。

```
BH1750:
  - BH1750
SHT25:
  - SHT25_Hum
  - SHT25_Temp
BMP280:
  - BMP280_Barometric
  - BMP280_Temp
MPU9250:
  - MPU9250_Accel
  - MPU9250_Gyro
  - MPU9250_Mag
GSU120:
  - GSU120_Location
#  - GSU120_Altitude
Si1145:
  - Si1145_UV
#  - Si1145_IR
StampExpander:
  - StampExpander
SWITCH:
  - Switch
```

第一階層に書かれているのは、センサードライバのソースファイル名、第二階層の配列の内容は、`SensorInfo`構造体で定義された変数名です。行頭に'#'があるのは、コメント行です。

このファイルの形式であるYAMLは一般的なYAMLと同じなので、詳しいことはそちらを参照して下さい。

この情報を元に、`tools/define_sensors.rb`が`src/build_sensors.c`を生成しますが、これはMakefileの中で自動的に行われるようになっています。

Makefileへの追加

センサードライバのコンパイルについては、Makefileに直接追加します。

以下は、Ver 1.0時点でのMakefileの当該箇所(53～60行目)です。

```
#-------------------------------------------------------------------------------

EMCS =  header.c left_menu.c index.c wifi.c network.c sensors.c basic.c emc.c

SENSORS = build_sensors.c i2c.c BH1750.c SHT25.c BMP180.c Si1145.c BMP280.c MPU9250.c
WDT.c
SENSORS += uart.c GSU120.c AQM0802.c
SENSORS += StampExpander.c SWITCH.c

```

`SENSORS`に追加するように、ソースファイル名を書いて下さい。順番は特に意味はありませんが、60行目の空行に

```
SENSORS += 新しいセンサードライバ
```

とでも書いておくと良いと思います。

以上でセンサードライバの組み込みは完了です。「設定モード」への追加も`build_sensors.yaml`に追加することで完了します。

デバッグのしかた

本開発キットでは、以下の2つのデバッグ方法を提供しています。

- printfデバッグ
- gdbによるデバッグ

printfデバッグのしかた

EveryStampのソースには、printfデバッグ用のマクロがいくつか定義されています。これらのマクロが有効になると、UARTにデバッグメッセージが表示されます。現在リリースしているEveryStampのファームウェアは、いくつかの理由で、一部のモジュールはデバッグマクロが有効な状態になっていますので、単にターミナルソフトを起動するだけで、メッセージを見ることが出来ます。

デバッグメッセージを見るためには、

```
$ minicom -D /dev/ttyUSB0 115200
```

のようにします。`/dev/ttyUSB0`の部分は、お使いの環境に依存しますので、適当に探して下さい。詳しくは「確認」を見て下さい。

自分の好きなところでこのメッセージを出力させるためには、まず出力させたいモジュールの先頭を見ます。たとえば、以下は`src/BMP280.c`の冒頭の部分です。

```
#include        "config.h"
//#define       DEBUG
//#define       TRACE
#include        "types.h"
#include        "os_support.h"
#include        "debug.h"
#include        "os_sensors.h"
#include        "i2c.h"
#include        "unit.h"
#ifdef  HAVE_BMP280
#include        "BMP280.h"
```

ここでコメントアウトされている、

```
//#define       DEBUG
//#define       TRACE
```

のコメントを解除して、これらのマクロを有効にします。

この状態でコンパイルしてEveryStampで実行すると、このソースにあらかじめ組み込まれているデバッグメッセージが出力されるようになります。

あらかじめ組み込まれている以上のメッセージを出したい場合には、メッセージ出力用のコードを追加します。EveryStampのソースで用意されているデバッグ出力用のマクロは、

- dbgmsg
- dbgprintf
- ENTER_FUNC
- LEAVE_FUNC

です。これらが何をするかは、リファレンスを参照して下さい。

デバッグ出力が必要でなくなった場合は、元のようにコメントアウトします。

dbgによるデバッグ

gdbを使ったデバッグも可能です。

gdbでデバッグを行うには、openocdを使います。このための定義は既にMakefileに書かれていますので、

```
$ sudo make debug
```

とするだけで利用出来ます。

まず上記のようにして、openocdを起動します。その次に、

```
$ arm-none-eabi-gdb xxxx.axf
```

のようにして、gdbを起動します。

gdbを起動したら、

```
(gdb) target remote localhost:3333
(gdb) set remote hardware-breakpoint-limit 6
(gdb) set remote hardware-watchpoint-limit 4
```

のようにコマンドを入れます。

このうち、重要なのは最初にあるtargetの指定です。ここでlocalhostのポート3333に接続するように指示していますが、これはMakefileの中に定義されたdebugターゲットの

```
debug:
        openocd -s $(TOOL_DIR) -f interface/ftdi.cfg -f openocd.cfg -c "gdb_port 3333"
```

の"gdb_port 3333"に対応します。何かの事情で3333が衝突する場合は、ここを変更して下さい。

以後は通常のgdbの操作と同じです。

なお、make debugした場合、通常は既にプログラムは動いてしまっています。このため、うまくgdbを使うことは出来ないと思います。

そこで、gdbによるデバッグを円滑に行うためには、あらかじめソースの中に

```
__asm__ ( "bkpt 0" );
```

のようなコードを入れてコンパイルしておくのが良いでしょう。

センサードライバから呼び出せる関数(API)

センサードライバを記述するにあたって利用出来る関数のリファレンスです。

I2C

I2Cは以下の基本APIがあります。

- I2C_open
- I2C_close
- I2C_read
- I2C_write

また、以下の応用APIもあります。

- I2C_reg_read_uint8
- I2C_reg_read_int8
- I2C_reg_read_uint16
- I2C_reg_read_int16
- I2C_reg_write_uint8
- I2C_reg_read_bytes
- I2C_read_uint8
- I2C_read_int8
- I2C_read_uint16
- I2C_read_int16
- I2C_write_uint8
- I2C_write_uint16

I2C_open

```
I2C_device      *I2C_open(uint8_t addr);
```

addr
: slaveのアドレスを指定します。7bit形式のアドレスです

返り値
: I2Cデバイスを表現するポインタです

いわゆるオープンです。

I2Cバスは、`I2C_open`から`I2C_close`までの間占有されます。そのため、操作を行う都度open/closeする必要があります。

I2C_close

```
void            I2C_close(I2C_device *dev);
```

dev
: I2Cデバイスを表現するポインタ

いわゆるクローズです。I2Cバスを解放します。

I2C_read

```
size_t          I2C_read(I2C_device *dev, uint8_t *buff, size_t size);
```

dev
: I2Cデバイスを表現するポインタ

buff
: データバッファ

size
: 読み込むデータのバイト数

返り値
: 読み込んだバイト数

I2Cからデータを指定バイト数読み込みます。

size分読み込むまでブロックしますが、タイムアウト(100ms)過ぎると解放されます。

I2C_write

```
size_t          I2C_write(I2C_device *dev, uint8_t *buff, size_t size);
```

dev
: I2Cデバイスを表現するポインタ

buff
: データバッファ

size
: 書き込むデータのバイト数

返り値
: 書き込んだバイト数

I2Cからデータを指定バイト数書き込みます。

I2Cのレジスタから読み込む関数

```
uint8_t         I2C_reg_read_uint8(I2C_device *dev, byte addr);
int8_t          I2C_reg_read_int8(I2C_device *dev, byte addr);
uint16_t        I2C_reg_read_uint16(I2C_device *dev, byte addr);
int16_t         I2C_reg_read_int16(I2C_device *dev, byte addr);
```

dev
: I2Cデバイスを表現するポインタ

addr
: レジスタのアドレス

返り値
: 読み込んだデータ

I2Cデバイスのレジスタからデータを読み込みます。

レジスタは1バイト長です。複数バイト読み込む関数は、内部でアドレスをインクリメントしています。

I2Cのレジスタに書き込む関数

```
void            I2C_reg_write_uint8(I2C_device *dev, byte addr, uint8_t data);
void            I2C_reg_read_bytes(I2C_device *dev, byte addr, byte *buff, size_t size);
```

dev
: I2Cデバイスを表現するポインタ

addr
: レジスタのアドレス

data, buff
: 書き込むデータ

size
: 書き込むバイト数

I2Cデバイスのレジスタにデータを書き込みます。

レジスタは1バイト長です。複数バイト書き込む関数は、内部でアドレスをインクリメントしています。

その他のI2Cから読み込む関数

```
uint8_t         I2C_read_uint8(I2C_device *dev);
int8_t          I2C_read_int8(I2C_device *dev);
uint16_t        I2C_read_uint16(I2C_device *dev);
int16_t         I2C_read_int16(I2C_device *dev);
```

dev
: I2Cデバイスを表現するポインタ

返り値
: 読み込んだデータ

I2Cからデータを読み込みます。

その他のI2Cに書き込む関数

```
size_t  I2C_write_uint8(I2C_device *dev, uint8_t val);
size_t  I2C_write_uint16(I2C_device *dev, uint16_t val);
```

dev
: I2Cデバイスを表現するポインタ

val
: 書き込むデータ

返り値
: 書き込んだバイト数

I2Cにデータを書き込みます。

UART

UARTには以下の基本APIがあります。

- UART_open
- UART_close
- UART_read
- UART_write
- UART_flash

また、以下の応用APIがあります。

- UART_puts
- UART_putc
- UART_getc
- UART_gets

UARTは読み込み書き込み共に永久ブロックをします。実用的には、UARTデバイスのセンサードライバは、独立したスレッドで動かすようにします。

UART_open

```
UART_device     *UART_open(int speed);
```

speed
: UARTの速度です

いわゆるオープンです。

UARTには基本的に1つのデバイスしかつながりません。openを一度だけ行えば十分です。

UART_close

```
void            UART_close(UART_device *dev);
```

dev
: UARTデバイスを表現するポインタ

いわゆるクローズです。実用的にはなくても構いません。

UART_read

```
size_t          UART_read(UART_device *dev, uint8_t *buff, size_t size);
```

dev
: UARTデバイスを表現するポインタ

buff
: データバッファ

size
: 読み込むデータのバイト数

返り値
: 読み込んだバイト数

UARTからデータを指定バイト数読み込みます。

size分読み込むまでブロックします。解放はされません。

UART_write

```
size_t          UART_write(UART_device *dev, uint8_t *buff, size_t size);
```

dev
: UARTデバイスを表現するポインタ

buff
: データバッファ

size
: 書き込むデータのバイト数

返り値
: 書き込んだバイト数

UARTにデータを指定バイト数書き込みます。

size分書き込むまでブロックします。解放はされません。

UART_flush

```
void            UART_flush(UART_device *dev);
```

dev
: UARTデバイスを表現するポインタ

UARTのバッファをflushします。

その他のUARTの読み書き関数

```
size_t  UART_puts(UART_device *dev, char *str);
void    UART_putc(UART_device *dev, int c);
int             UART_getc(UART_device *dev);
size_t  UART_gets(UART_device *dev, char *buff);
```

dev
: UARTデバイスを表現するポインタ

str
: 文字列

c
: 文字

buff
: データバッファ

UARTに対する、`puts`, `putc`, `getc`, `gets`です。

GPIO

GPIOには以下の応用APIがあります。

- GPIO_is_sw1_on
- GPIO_set_sw1_callback
- GPIO_is_input_on
- GPIO_set_input_callback
- GPIO_output_pin

他にもいくつか定義されていますが、実際にセンサードライバで利用出来るのは、この5つです。

GPIO_is_sw1_on, GPIO_is_input_on

```
Bool    GPIO_is_sw1_on(void);
Bool    GPIO_is_input_on(void);
```

返り値
: GPIOの状態

GPIOの状態を得ます。`GPIO_is_sw1_on`は本体スイッチの状態、`GPIO_is_input_on`は、センサーモジュールインターフェイスコネクタのGPIO入力線です。

GPIO_set_sw1_callback, GPIO_set_input_callback

```
void    GPIO_set_sw1_callback(void (*func)(void));
void    GPIO_set_input_callback(void (*func)(void));
```

func
: コールバック関数

GPIOの状態変化があった時に通知するコールバックです。

`GPIO_set_sw1_callback`は押された時に発動するコールバックを設定します。

`GPIO_set_input_callback`は、オンになった時にもオフになった時にも発動するコールバックを設定します。区別が必要な場合は、`GPIO_is_input_on`を併せて使います。

GPIO_output_pin

```
void    GPIO_output_pin(Bool on);
```

on
: GPIOの状態

センサーモジュールインターフェイスコネクタの出力用GPIO線をオンオフします。

スレッド

以下のスレッド操作が可能になっています。

- THREAD_create
- THREAD_delete

THREAD_create

```
THREAD  THREAD_create(char *name, void (*func)(void *), void *arg, size_t size_stack, int
prio);
```

name
: スレッド名

func
: スレッドにする関数

arg
: funcに渡す引数

size_stack
: スレッドが使うスタックの大きさ

prio
: スレッドの優先度。OS_PRIO_0からOS_PRIO_4まであり、数字が小さい程優先度は高くなっています。

返り値
: スレッドを表現するデータ

funcで指定したスレッドを起動します。

スタックサイズは、あまり大きいと取得出来ないので、必要最低限にします。

THREAD_delete

```
void    THREAD_delete(THREAD thread);
```

thread
: スレッドを表現するデータ

スレッドを削除します。

EveryStampのメモリ管理は非常に貧弱なので、スレッドの起動停止はあまり頻繁には出来ません。出来れば停止しないで、動いたままにして下さい。

mutex

排他制御のためのmutexの操作は、以下のものがあります。

- MUTEX_create
- MUTEX_get
- MUTEX_put

MUTEX_create

```
MUTEX   MUTEX_create(char *name, int flags);
```

name
: mutexの名前

flags
: mutexの継承。OS_MUTEX_INHERITとOS_MUTEX_NO_INHERITがあります。

返り値
: mutexを意味するデータ

mutexを作ります。

MUTEX_get

```
Bool    MUTEX_get(MUTEX mutex, uint32_t wait);
```

mutex
: mutexを意味するデータ

wait
: 獲得するまでの待ち時間。単位はticksです。実際の時間からticksに変換するには、msec_to_ticksを使います。0の場合は待ちません。OS_WAIT_FOREVERの場合は永久に待ちます。

返り値
: 獲得に成功したかどうか

mutexを獲得します。

成功しても失敗しても復帰しますが、失敗した場合は'FALSE'が返ります。

MUTEX_put

```
void    MUTEX_put(MUTEX mutex);
```

mutex
: mutexを意味するデータ

mutexを解放します。

ユーティリティルーチン

以下のユーティリティルーチンとマクロがあります。

- appln_critical_error_handler
- sleep, sleep_ms
- msec_to_ticks
- ftos
- unsigned2signed
- memclear
- strlcmp
- xmalloc, xfree
- dbgmsg, dbgprintf
- ENTER_FUNC, LEAVE_FUNC

appln_critical_error_handler

回復不能のエラーが発生した場合に呼び出します。

```
void    appln_critical_error_handler(void *data);
```

data
: メッセージ

現在のモードのままリブートします。

sleep, sleep_ms

現在のスレッドが指定した時間だけsleepします。

```
void    sleep(int val);
void    sleep_ms(int val);
```

val
: sleepする時間。sleepの単位は秒、sleep_msの単位はミリ秒です。

msec_to_ticks

`MUTEX_get`等で使われる、ticks単位の時間を求めるための関数です。

```
unsigned long   msec_to_ticks(unsigned long msec);
```

msec
: ミリ秒

返り値
: msecをticksに変換した値

ftos

`float`型を出力用の文字列に変換します。

EveryStampでは、`printf`に浮動小数点が使えないため、この関数で文字列にします。

```
char    *ftos(float val, int length, int prec);
```

val
: 変換するべき値

length
: 変換結果の文字列長。0を指定すると、システムの用意したバッファの長さになります。

prec
: 小数点以下の桁数。-1を指定すると、変換結果がlengthの桁数いっぱいになるまで指定したことになります。

処理のためのバッファは固定領域で使い回されますので、使い方に留意する必要があります。

---

unsigned2signed

---

uint8_tの配列をintに変換します。

```
int     unsigned2signed(uint8_t *buff, Bool rev)
```

buff
: 変換対象の配列

rev
: エンディアン変換の有無

2バイトが前提です。

---

memclear

メモリ領域を0で埋めます。

```
#define memclear(buff,size)             memset((buff),0,(size))
```

buff
: 処理対象の領域

size
: 領域の大きさ(バイト単位)

---

strlcmp

文字列の前方一致

```
#define strlcmp(p,q)                    strncmp((p),(q),strlen(q))
```

p, q
: 比較するべき文字列

qの長さ分だけ比較します。

---

xmalloc, xfree

メモリの獲得と解放

通常のmalloc, freeと同じですが、デバッグ時にメッセージを出力します。

---

dbgmsg, dbgprintf

---

printfデバッグに使う、デバッグ時のみ有効な出力です。

単メッセージを出力するだけなら`dbgmsg`を使い、`printf`をしたい場合は`dbgprintf`を使います。

マクロ、'`TRACE`'が無効の場合は、コード自体がなくなります。

ENTER_FUNC, LEAVE_FUNC

---

printfデバッグに使う、デバッグ時のみ有効な出力です。

関数に入ったか出たかを表示します。

マクロ、'`TRACE`'が無効の場合は、コード自体がなくなります。