シリアルデータ送信前の 文字列編集処理

今回は、シリアルデータ通信において、送信電文を組み立てるための 文字列編集処理の関数について使い方を説明します。

その前に電文とは 何かというと、遥か昔はホストと 端末という概念で データ通信は行われていました。 ホストは データを管理する中心となるコンピュータです。 端末は ホストの下に 複数接続され 電話回線にモデムを接続して 遠距離の通信が行われていました。

で、通信電文ですが その当時からテキスト(ASCII 文字列)で、構成された数十byteから最大で 200byteぐらいの 電文のやり取りを行っていました。 最大 200byteというのは、あまり電文が長いとその分 パルス性のノイズに当たる確率が 高くなるので 最大 200byteぐらいというのが あるようです。

それと、ある程度まとまったブロック Max 200byteを ひとつの電文として送ります。 その時 電文を、正常に受けたよ。 とか 受信データが壊れているので 再送信してくれ。とかの要求が受信側から送信側に 送られる場合があります。

これらは 回線制御コードとも呼ばれます。これらは、ASCIIコードの 00h ～ 1Fh の範囲に制御コード(文字として見る事の出来ないコード)として宣言してあります。 回線制御コードは EOT、ENQ、ACK、NAKとかが あります。 これらは 1byte単独で送る場合が 多いです。電文の中に含めるコードも あります。 SOH、STX、ETX が あります。 あとプリンター等の制御コードで CR(キャリッジリターン)、LF(ラインフィード)があります。 CR、LFは 1行改行する時に 使用します。 あと FFというのもあります。 これは、改ページコードです。

あと、最近は見ませんが EIA-232Cで パソコンと XYプロッタを接続する際に フロー制御で Xon、Xoff のコードを 使う事があります。

これも、ASCII コードの 00h～1Fh 内に あります。 ASCII コード表には Xon、Xoff では 記載されていません。 Xon が DC1 で、Xoff が DC3のようです。

で、何故 制御コードの話をしたかというと、データ通信を行う場合 データを 伝送するのが主な目的ですが、伝送を行う都合で 制御コードも 場合により必要になる。 ということです。

通常は、テキストで データ転送を行う事が多いのですが、長くても 数mの近距離で伝送効率を 上げるため 独自仕様で バイナリデータを 送る場合もあるようです。

その場合、一つ問題があり、データと 制御コードの区別が 難しくなります。 何故なら バイナリデータは 00h～ FFh まで全ての値を取ります。

よって単純にデータか 制御コードかの 区別が 出来なくなるということです

一般に それを 回避するため バイナリデータの場合は 事前に決められた固定長で バイナリデータを 転送します。それでも送受信の シーケンスがずれるとまずいので、バイナリデータの前と バイナリデータの後ろに 決められたコードを付ける事により、前後のコードが決められたコードと 一致した事により 正常にバイナリデータを受信したという判断を 行う事になります。

あと データは長くなりますが バイナリデータを 16進文字列データとして 送る方法もあります。俗にいう ヘキサファイルです。

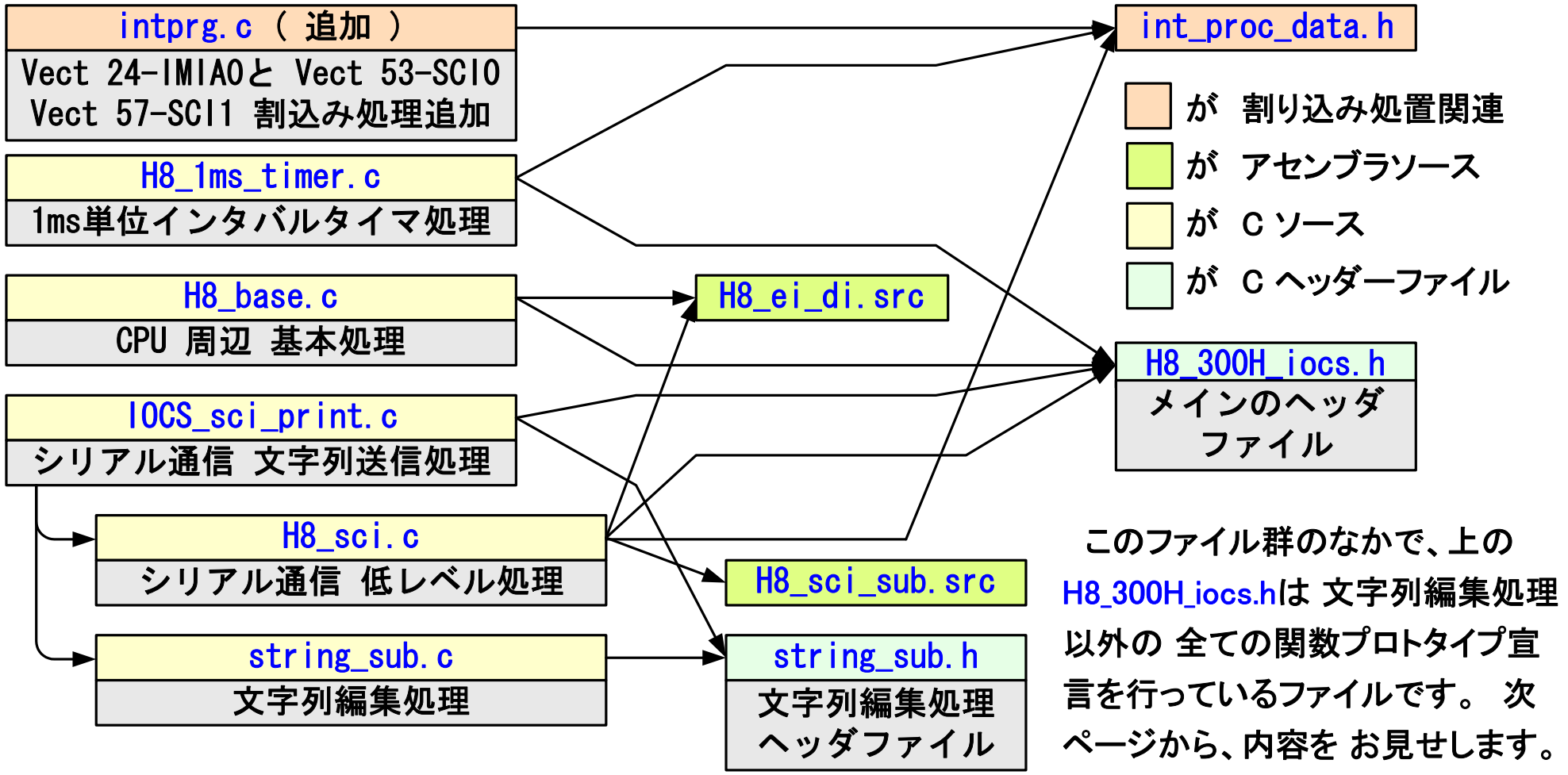
マイコンから、パソコンのテラタームに　マイコンの動作確認のため、文字列を表示するのであれば、ASCII 文字列と最後に　CrLf の付いた文字列を　送るだけで十分な気がします。

　マイコンと　パソコンの プログラム、あるいは複数のマイコン間にて　データ転送を行う場合は、最低限　ACK、NAK ぐらいは必要と思います。

　ちょっと制御コードの話で、長くなりましたね。本題に入ります。 まず、今回のサブルーチンのソースファイルの構成について説明します。

　次のページに　全体のソースファイル構成を描きます。

H8_IOCS関数の ファイル構成



このファイル群のなかで、上の H8_300H_iocs.hは 文字列編集処理以外の 全ての関数プロトタイプ宣言を行っているファイルです。 次ページから、内容を お見せします。

H8_300H_iocs.h　IOCS関数の宣言ファイル

```
//   CPU周辺  基本機能 ( H8_base.c , H8_ei_di.src )                    ( 1/5 )
// ------------------------------------------------------
char  get_cpu_mode( void );           // CPU Mode取り出し（有効なのは 最下位 8bit)
void  enable_interrupt( void );       // 割り込み許可 ( H8_ei_di.src )
void  disable_interrupt( void );      // 割り込み禁止 ( H8_ei_di.src )
void  soft_reset( void );             // ソフトによる CPUリセット
void  sleep_func( void );             // CPU スリープに移行させる

//   1ms_tamer処理 ( H8_1ms_timer.c )
// ------------------------------------------------------
void     wait_ms( int n );            // ソフトによる 1ms単位  時間待ち処理
void    init_1ms_timer( void );       // 1ms インターバルタイマー初期化
_UWORD  get_counter_1m( void );       // 1msフリーランカウンタ読み出し
void    set_timer_1m1( _UWORD n );    // 1msタイマー１初期値設定
_UWORD  get_timer_1m1( void );        // 1msタイマー１現在値読み出し
void    set_timer_1m2( _UWORD n );    // 1msタイマー２初期値設定
_UWORD  get_timer_1m2( void );        // 1msタイマー２現在値読み出し
void    set_timer_1m3( _UWORD n );    // 1msタイマー３初期値設定
_UWORD  get_timer_1m3( void );        // 1msタイマー３現在値読み出し
```

```
void    set_timer_1m4( _UWORD n );  // 1msタイマー４初期値設定                    ( 2/5 )
_UWORD  get_timer_1m4( void );      // 1msタイマー４現在値読み出し

//   シリアル通信基本処理 ( H8_sci.c )
// ------------------------------------------------------------
int  open_sci_0( _UWORD bps, char *fmt ); // SCI.0 オープン処理
void close_sci_0( void );             // SCI.0 クローズ処理
void send_sci_0( char dt );           // SCI.0  1byte 送信
int  get_ring0_cnt( void );           // 受信用リングバッファ0  データ格納数 取り出し
int  get_ring0_data( void );          // SCI0 リング格納データ 1byte 取り出し

int  open_sci_1( _UWORD bps, char *fmt ); // SCI.1 オープン処理
void close_sci_1( void );             // SCI.1 クローズ処理
void send_sci_1( char dt );           // SCI.1  1byte 送信
int  get_ring1_cnt( void );           // 受信用リングバッファ1  データ格納数 取り出し
int  get_ring1_data( void );          // SCI1 リング格納データ 1byte 取り出し

//   シリアル通信基本サブ処理 ( H8_sci_sub.src )
// ------------------------------------------------------------
void  init_sci0_sub( void );          // SCI0 初期化の 一部分
void  send_sci0_check( void );        // SCI0  1byte 送信時の  レディ確認
```

```
void  init_sci1_sub( void );          // SCI1 初期化の 一部分                    ( 3/5 )
void  send_sci1_check( void );        // SCI1  1byte 送信時の レディ確認

//   シリアル通信 文字列 送信処理 ( IOCS_sci_print.c )
// ------------------------------------------------------------
//   ★  SCIチャネル 0 側 :
void  sci_prin_space_0( int n );      // スペースコード出力
void  sci_beep_0( void );             // ビープ音を 端末側で鳴らす
void  sci_put_crlf_0( void );         // 改行コード出力
void  sci_prin_0( char *tx );         // 文字列のみの出力
void  sci_print_0( char *tx );        // 文字列 + CrLf 出力
void  sci_put_bin_0( unsigned char *buf, int cnt );  // バイナリデータの出力
void  sci_prin_ascii_0( char *asc, int len );        // ASCII文字以外は '.'に置き換え出力
void  sci_prin_ascii_16_0( char *asc, int len ); // 16文字出力 ASCII文字以外は '.'に置換え出力
void  sci_prin_byte_hex1_0( unsigned char c ); // 1byteデータ 下位 4bit: 16進文字 1文字出力
void  sci_prin_byte_hex2_0( unsigned char c );  // 1byteデータ: 16進文字列 2文字出力
void  sci_prin_byte_bcd2_0( unsigned char c );  // 1byteデータ: BCD文字列 2文字出力
void  sci_prin_word_hex4_0( unsigned int dt );     // 1Wordデータ: 16進文字列 4文字出力
void  sci_prin_dword_hex6_0( unsigned long dt );  // Dwordデータ: 16進文字列 6文字出力
void  sci_prin_dword_hex8_0( unsigned long dt );  // Dwordデータ: 16進文字列 8文字出力
void  sci_prin_byte_dec_0( unsigned char dat ); // byteデータ :10進数 3桁 文字列出力
```

```
void  sci_prin_word_dec_0( int dat, _UBYTE sw );  // Wordデータ：10進数 文字列出力     ( 4/5 )
void  sci_prin_long_dec_0( long dat, _UBYTE sw ); // longデータ：10進数 文字列出力
void  sci_prin_byte_bit8_0( _UBYTE dt );          // byteデータ --> 2進数 4bit+' '+4bit出力
int   sci_recv_wait_0( void );                          // １文字 受信待ち
int   sci_txt_input_0( char *ttl, char txt[], int len ); // 文字列の入力処理
// ------------------------------------------------------------
//   ★  SCIチャネル 1 側：
void  sci_prin_space_1( int n );     // スペースコード出力
void  sci_beep_1( void );            // ビープ音を 端末側で鳴らす
void  sci_put_crlf_1( void );        // 改行コード出力
void  sci_prin_1( char *tx );        // 文字列のみの出力
void  sci_print_1( char *tx );       // 文字列 + CrLf 出力
void  sci_put_bin_1( unsigned char *buf, int cnt ); // バイナリデータの出力
void  sci_prin_ascii_1( char *asc, int len );       // ASCII文字以外は '.'に置き換え出力
void  sci_prin_ascii_16_1( char *asc, int len ); // 16文字出力 ASCII文字以外は '.'に置き換え出力
void  sci_prin_byte_hex1_1( unsigned char c ); // 1byteデータ下位 4bit: 16進文字 1文字出力
void  sci_prin_byte_hex2_1( unsigned char c ); // 1byteデータ: 16進文字列 2文字出力
void  sci_prin_byte_bcd2_1( unsigned char c ); // 1byteデータ: BCD文字列 2文字出力
void  sci_prin_word_hex4_1( unsigned int dt );   // 1Wordデータ: 16進文字列 4文字出力
void  sci_prin_dword_hex6_1( unsigned long dt ); // Dwordデータ: 16進文字列 6文字出力
void  sci_prin_dword_hex8_1( unsigned long dt ); // Dwordデータ: 16進文字列 8文字出力
```

```
void  sci_prin_byte_dec_1( unsigned char dat );      // byteデータ：10進数 3桁 文字列出力
void  sci_prin_word_dec_1( int dat, _UBYTE sw );     // Wordデータ：10進数 文字列出力
void  sci_prin_long_dec_1( long dat, _UBYTE sw );    // longデータ：10進数 文字列出力
void  sci_prin_byte_bit8_1( _UBYTE dt );             // byteデータ --> 2進数 4bit+' '+4bit出力
int   sci_recv_wait_1( void );                          // １文字 受信待ち
int  sci_txt_input_1( char *ttl, char txt[], int len );  // 文字列の入力処理
```

以上が、現時点での H8_300H_iocs.h の 内容です。
今後、H8/3069基板裏側に実装されている 2MBの D-RAM アクセスや、
I2Cのアクセス関数を 追加していく予定です。

ちなみに、H8マイコンは I2Cインタフェースの周辺回路を 持ちません。
よって I/Oポートを 細かく Hi、Low させるソフトウェアで 実現します。それと、H8マイコンは 5Vで動作しますが、I2Cデバイスは 3.3Vで動作するデバイスも多いです。 よって信号線の 電圧変換も 必要となります。

で、今回は、手始めに intprg.c の 追加割り込み処理の説明と 使用する構造体データの型宣言をしている int_proc_data.h の説明を 行います。

それと もう一つ H8_sci.c の説明を行います。

int_proc_data.h　構造体データ型宣言ファイル

```
#define    SBUF_SIZ   256   // シリアル通信リングサイズ
typedef    struct     {     // タイマー、カウンター構造体
     _UWORD     ctr;        // フリーラン カウンタ
     _UWORD     tm1;        // 1ms タイマー変数.1
     _UWORD     tm2;        // 1ms タイマー変数.2
     _UWORD     tm3;        // 1ms タイマー変数.3
     _UWORD     tm4;        // 1ms タイマー変数.4
} TCU160_CTR;
typedef    struct     {     // シリアル通信用リングバッファ
     _UBYTE     buf[SBUF_SIZ];  // リングバッファ
     _UBYTE     wp;         // 書き込み位置ポインタ
     _UBYTE     rp;         // 読み出し位置ポインタ
     _UBYTE     len;        // 書き込みデータ長( byte )
} SERIAL_RING;
```

一番上の SBUF_SIZは シリアル通信 受信用のリングバッファの最大格納文字数です。 受信は、相手の都合で いつ送ってくるか 分からないので、送られてきた複数文字をリングバッファ内に蓄えておきます。
TCU160_CTRは 1msのインターバルタイマ用のカウンタ変数 5個です。 ctr が 1ms毎に 常時インクリメントするカウンタ変数です。
tm1～tm4 は 通常 0 では カウント動作を行いません。 値がセットされたとき、値が 0 でなければ、1ms毎に デクリメントを 行います。

SERIAL_RING は、シリアルデータ受信用のリングバッファです。受信割り込みで 受信データを 取り込み リングバッファの wpポインタで指す位置に データを書き込み、wpと lenを +1 します。リングからデータを取り出す時は、len が 1以上である事を確認して rpで指す位置の リングデータを取り出し rp を +1して、lenが 0 以上であれば len -1 を 行います。

intprg.c 割り込み処理関数テーブルファイル

(1/4)

```
#include "IntProc_data.h" // 割り込み処理内で使用する構造体データ

//**  内部データ領域
// ------------------------------------------------------------------
    volatile  TCU16O_CTR   Tcu_cn;   // TCU16_0 カウンタ変数
    volatile  SERIAL_RING  Sr0, Sr1; // 受信用リングバッファ 0と 1

//==================================================================
#pragma section IntPRG
//  vector 1 Reserved
```

intprg.cは 標準で 自動生成されるファイルですが このソースは 何もしないと 全てのベクター番号の空の割り込み処理で構成されています。

割り込み処理を記述する場合は、空の割り込み処理の中にコードを入れて行きます。

赤四角で囲った部分は、前のページで 型宣言した 構造体の 変数を用意してます。
Tcu_cn は インターバルタイマ用の カウント変数です。
Sr0、Sr1 は シリアル通信周辺回路の SCI0、SCI1用の リングバッファ構造体です。

(2/4)

```
//__interrupt(vect=24) void INT_IMIA0(void) {/* sleep(); */}
__interrupt(vect=24) void INT_IMIA0(void)
{
      ITU.TISRA.BIT.IMFA0 = 0;   // 16bitタイマー IMFA0 クリア
      Tcu_cn.ctr++;
      if( Tcu_cn.tm1 > 0 ) Tcu_cn.tm1--;
      if( Tcu_cn.tm2 > 0 ) Tcu_cn.tm2--;
      if( Tcu_cn.tm3 > 0 ) Tcu_cn.tm3--;
      if( Tcu_cn.tm4 > 0 ) Tcu_cn.tm4--;
}
```

ベクター 24 は 16bitタイマー周辺回路 0 の ベクターです。
元々は コメントにした 赤の上の一行です。右の {/* sleep(); */} を下に降ろして、/* sleep(); */ の ところを 青の6行に 変更した。という事です。

ITU.TISRA.BIT.IMFA0 = 0; は 右のコメントの通りですが、ITUというタイマー周辺回路の IMFA0 というフラグを クリアしないと 次の割り込みが正常に受け付けられなくなるので、ハード的な お約束と思って下さい。 Tcu_cn.ctr++; は フリーランカウンタを インクリメントしています。 その下の4行は Tcu_cn.tm1 ～ Tcu_cn.tm4 にて 同様の処理を行っていますが 例として tm1 で 説明すると tm1 が 0 より大きければ tm1を デクリメントする という事です。
このタイマー処理は 初期化で 1ms周期で 割り込みが 継続してかかるようにしています。よって、tm1 は 1ms毎に カウントダウンされます。 tm1に 1000 を セットすれば、1秒後に tm1は 0 に なります。

```
__interrupt(vect=53) void INT_RXI0(void)
{
     //***  SCI0の データ受信割り込み  ***
volatile  _UBYTE     dt, sts;

     SCI0.SCR.BIT.RIE = 0;      // SCI0 一旦 受信割り込み 停止
     dt = SCI0.RDR;             // 受信データを取り出す
     sts = SCI0.SSR.BYTE;       // ダミー読み出し
     SCI0.SSR.BIT.RDRF = 0;     // 受信フラグを クリア

     Sr0.buf[Sr0.wp] = dt;      // 受信データを リングバッファに書き込む
     if( Sr0.len < SBUF_SIZ -1 )   // 書き込みデータ数 満杯より小さいか ?
     {
          Sr0.wp++;             //  Write Pointer 更新 255の次 0に 折り返す
          Sr0.len++;            //  格納個数 +1 更新
     }
     SCI0.SCR.BIT.RIE = 1;      // SCI0 受信割り込み 再開
}
```

```
__interrupt(vect=57) void INT_RXI1(void)
{
     //***  SCI1の データ受信割り込み  ***
volatile  _UBYTE     dt, sts;

     SCI1.SCR.BIT.RIE = 0;      // SCI1 一旦 受信割り込み 停止
     dt = SCI1.RDR;             // 受信データを取り出す
     sts = SCI1.SSR.BYTE;       // ダミー読み出し
     SCI1.SSR.BIT.RDRF = 0;     // 受信フラグを クリア

     Sr1.buf[Sr1.wp] = dt;      // 受信データを リングバッファに書き込む
     if( Sr1.len < SBUF_SIZ -1 )   // 書き込みデータ数 満杯より小さいか ?
     {
          Sr1.wp++;             //  Write Pointer 更新 255の次 0に 折り返す
          Sr1.len++;            //  格納個数 +1 更新
     }
     SCI1.SCR.BIT.RIE = 1;      // SCI1 受信割り込み 再開
}
```

```
//    シリアル通信基本処理 ( H8_sci.c )
int  open_sci_0( _UWORD bps, char *fmt ); // SCI.0 オープン処理
void close_sci_0( void );      // SCI.0 クローズ処理
void send_sci_0( char dt );  // SCI.0  1 _UBYTE 送信
int  get_ring0_cnt( void );   // 受信用リングバッファ0  データ格納数 取り出し
int  get_ring0_data( void );  // SCI0 リング格納データ 1_UBYTE 取り出し

int  open_sci_1( _UWORD bps, char *fmt ); // SCI.1 オープン処理
void close_sci_1( void );      // SCI.1 クローズ処理
void send_sci_1( char dt );   // SCI.1  1 _UBYTE 送信
int  get_ring1_cnt( void );    // 受信用リングバッファ1 データ格納数 取り出し
int  get_ring1_data( void );   // SCI1 リング格納データ 1_UBYTE 取り出し
```

アプリ側から 呼び出すシリアル通信の 基本関数です。 sci_0 と sci_1 は 同様の使い方なので sci_0 で 説明します。 まず int open_sci_0(_UWORD bps, char *fmt); ですが 引数 bps ですがボーレイトです。 設定範囲は 150、300、600、1200、2400、4800、9600、19200、38400 です。 プリスケーラの分解能の関係で 最高速度 38400 です。 引数 *fmt は 3文字構成で 1文字目が パリティで "N"、"E"、"O" です。 2文字目が データ長で "7"、"8"です。 3文字目が Stop bit長で "1"、"2" です。 よく使用するのは "N81" Non Parity、8bit Data Length、1stop bit です。 send_sci_0関数の引数は 送信文字です。 get_ring0_cnt関数の関数値は 0 ～ 255 です。 get_ring0_data関数の関数値は データが無い時は -1で、データがある時はそのデータの値 0 ～ 255 を 返します。 H8_sci.c内の 関数の頭にも コメントで 引数の説明を 付けています。

今回、新たに分かった事

今まで、ルネサスの開発環境は 本来有償の開発環境であって、評価版の開発環境は、いくつかの制約があり 標準的なライブラリの提供は 外されている。と、思い込んで いました。

評価版に明らかに存在する制約は リンク後の 実行プログラムサイズが 64Kバイト以内である事。です。その他 製品版には あるけど評価版で外されている機能が あるかもしれません。 遥か昔の環境は 制約が多かった事もあり、そう思い込んでいたのかも しれません。

そう思い込んでいたので、文字列編集のような機能も、自前で用意する必要がある。 と、思い 自作していました。

それが、もしかして sprintf 関数が使えるのでは、ないかと 急に思い立って 試してみたら sprintf 関数が使えました。 条件としては、ソース先頭で #include <stdio.h> を 入れる事です。

で、もう一つ sprintf 関数にて 浮動小数点データも使える事を 確認しました。 但し、4byte 浮動小数点の float 変数のみの サポートです。

で、開発途中で sprintf関数や float変数を 使い始めると ビルドにて 一旦、オブジェクトファイルが 全て削除されます。 これは、浮動小数点の必要が無い場合は、整数のみの演算ライブラリが リンクされて小さい実行形式が作られます。

浮動小数点を扱う事になる場合は、整数と、浮動小数点の演算ライブラリが リンクされるようで一旦、オブジェクトファイルを 消す必要があるのでしょう。そして実行形式のサイズも 大きいです。

私が、小さいプログラムでテストした時は
整数だけのプログラムでは 実行形式が 9K byteでしたが、整数と実数のプログラムでは 30Kbyte でした。 約 3倍に なります。

でも、64Kbyte以内であれば、開発出来るので、まだ 34Kbyteぐらいは 余裕が あります。
巨大な実行形式は 作れませんが sprintf 関数は、積極的に使ってもいいと 思います。

それと、H8マイコンには 浮動小数点演算コプロセッサが ありません。 ソフトで浮動小数点演算を 行います。
ソフトで 浮動小数点演算を 行うと、浮動小数点演算コプロセッサを 使う時に比べ 演算処理速度が 1/20 ～ 1/30 ぐらいに落ちます。
遥か昔、初期の PC-9801 16bitで、浮動小数点演算コプロセッサ i8087を 抜き差しして 実験しました。

浮動小数点演算が 出来れば 便利な場合もありますが、H8は 基本 16bitマイコンなので 浮動小数点の 演算処理は、遅いです。

あと、指数関数、対数関数、三角関数 などの 超越関数は 使えません。 あくまで、加減乗除だけのようです。

あと、floatの浮動小数点演算では 有効数字は 10進数で 6桁ぐらいです。
浮動小数点が使えても 超越関数が 使えないのであれば、魅力半減という感じですね。
以上でした。

あと、簡単な文字表示の実験動画を お見せします。