UART でのボー・レート補正について

調歩同期通信では，送信側と受信側の通信速度（ボー・レート）が一致していないと正常に通信できません。細かく見ると，完全に同じではなく，通信速度の誤差がある程度の範囲に収まっていれば，通信は可能です。

このある程度の範囲は，通信する 1 キャラクタのビット長やパリティの有無で変化しますが，8 ビット程度のデータでは誤差が 4%程度以下（データの最後までいっても，サンプリングのタイミングがビットの中に納まる）なら通信は可能です。ただし，注意する必要があるのは，通信する信号波形の鈍りです。調歩同期通信は一つのボード内の通信ではなく，ボード間やセットの間の通信に使用することがほとんどで，信号線が長くなり，波形の立ち上がりや立下りで鈍りが発生したり，ひどい場合にはリンギングが発生して信号が安定するまでに時間がかかったりします。ここらを無視して，この範囲の誤差までは許容できると言った数値だけの計算は意味がありませんが，誤差はできるだけ少なくなるようにすべきです。この問題は，ハードウェアとして対策すべきことなので，今回の説明からは省きます。

RL78 を含めて，殆どの MCU の UART は内部のシステム・クロック（f_{CLK}）を分周して通信クロックを生成しています。f_{CLK} が正確ならば，そこから生成される通信クロックも正確に計算することが可能です。RL78 のマニュアルの目標ボー・レート誤差に記載された値は，あくまで f_{CLK} に誤差がないものとして計算してあるようです。これは，SAU としての説明の中なので仕方がないと言えます。

これまでのプログラムは，調歩同期通信でのボー・レート誤差を無視してきました。これは，波形の鈍りを無視すると，RL78/G13 の高速内蔵発振回路（HOCO）は電源電圧が 1.8V 以上なら±1%の誤差（温度範囲を広げても誤差は±1.5%）で，調歩同期通信でもほとんど気にならないレベルだったためです。

しかし，RL78/G10 では±2%，温度範囲が広がると±3%となり，気にする必要が出てきます。さらに，RL78/I1D や RL78/G11 の中速内蔵発振回路（MOCO）は，停止状態からの立ち上がりは早いのですが，精度は±12%とそのままではとても調歩同期通信には使えません。

幸いにして，内蔵発振回路の誤差は，個々のデバイスに依存する分が多く，温度で変動が少しある程度と考えることができるようなので，補正さえ行えば UART は使えます。

ボー・レートの補正方法

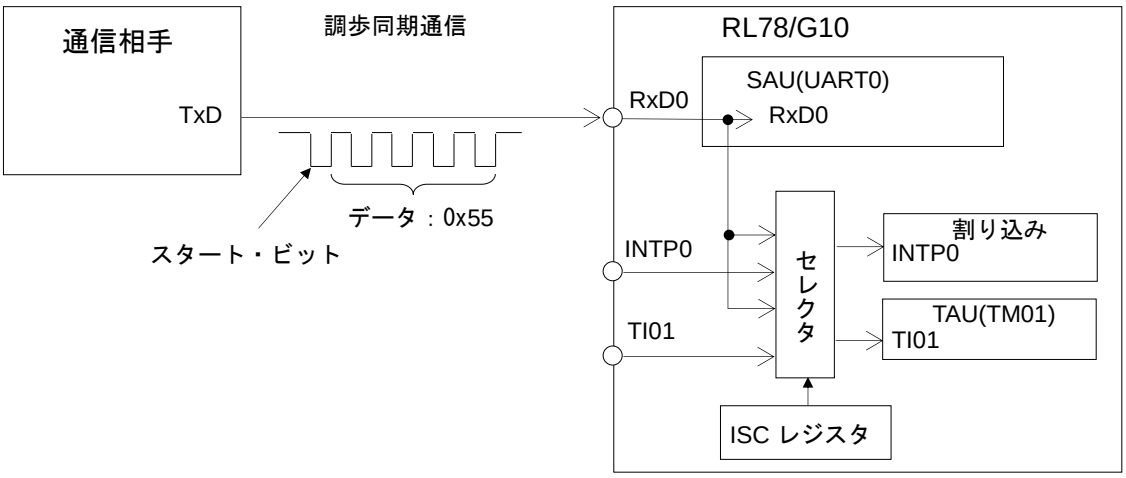
RL78 に内蔵された UART のボー・レートは fCLK をプリスケーラで 2^n 分周した SAU の動作クロック（f_{MCK}）を SDR レジスタの上位 7 ビットで指定された分周比で分周し，さらに 2 分周（これは，スタート・ビットを検出して，1/2 ビット遅らせることでビットの中央でサンプリングするためです）したものとなります。つまり，SDR レジスタの上位 7 ビットに設定する値を調整することで補正が可能になります。基準になる周波数の信号があれば，7 ビットで指定するので，ボー・レート誤差は 1%以下にはできる可能性があります。

具体的に RL78/I1D や RL78/G11 の UART 通信のアプリケーション・ノートを眺めてみると，±12%精度の MOCO での UART 動作でやっているように，±1%精度の HOCO を使って MOCO の周波数を求め，その値に応じた値を SDR レジスタに設定することで補正を行っています。これも，一つの方法ではありますが，個人的にはこのアプリケーション・ノートの方法（定期的に通信を停止して補正を行う）は好きではありません。この方法は，あくまで，起動時だけにすべきだと考えています。

2 つ目の方法は，起動時には上記の方法で初期設定を行っておき，その後は受信したデータそのものでボー・レート補正を行う方法です。調歩同期通信では，スタート・ビットが存在し，最後はストップ・ビット（正確には 1.5 ビットなんてのがあるのでビットとは言えないかもしれません）で終わります。つまり，受信データには必ずロウの期間が存在します。このロウの期間を測定することで，受信したデータのボー・レートを求め，それに合わせる方法です。ロウの期間はデータによって変化し，1～10（スタート・ビット＋データ＋パリティ）ビットになります。不定なロウ期間から正しいビット数を求めるには相対的な誤差が±5%である必要があります。このために，起動時に HOCO で±1%に設定する必要があります。この方法は，このような条件が付きますが，通信相手はそのままで，受信側だけで対応できるメリットがあります。つまり，既に存在しているシステムに比較的簡単に追加することができます。

3 つ目の方法は LIN バスで使われているような，通信の頭で特定のデータ（0x55）を送信し，受信側ではそのデータから送信側のボー・レートを算出し，送信側のボー・レートに合わせるものです。この方法は，ビットの配置が明確になっているので，わざわざビット長を求める必要がなく，ボー・レートをかなり正確に合わせ込むことが可能です。殆どの RL78 は LIN バスに対応した UART が搭載されているので，それらの機能を使用すると，送受信信号線だけを接続するだけで対応が可能です。LIN バスでは送受信を 1 本の信号線で行う半二重通信を用いていますが，2 本の信号性を用いた全二重通信でも同じことは可能です。

この方法を用いたボー・レート補正は RL78/G10 のアプリケーションノート（R01AN2473）があります（補正方法が準拠しているだけです）。RL78/G10 の構成を下図に示します。

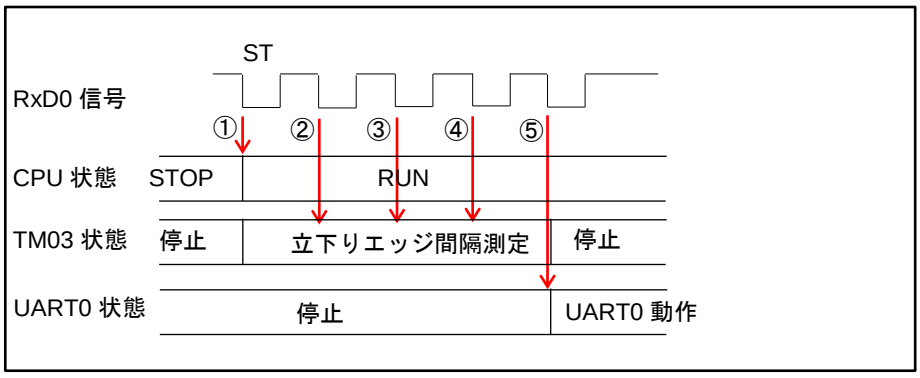


ボー・レートの補正の実現

ここでは，通信の最初に 0x55 を送る方法を少しモディファイしてみます。LIN バスでは，補正データである 0x55 の前に，補正を行うことを示すブレーク信号や，さらにその前にスタンバイを解除する処理が定義されていますが，それをデータ 0x55 だけで済ませます。

具体的には，0x55 のスタート・ビットでスタンバイを解除し，4 ビット分の時間を計測してボー・レートを計算し，UART のボー・レートを設定して再起動します。

ここでは，RL78/G11 の MOCO 動作をターゲットとして考えてみます（他のデバイスでは使用する内蔵周辺機能を変更する必要があります。もっとも，殆どの RL78 では補正そのものが不要ですが，マージンが大きくなるメリットはあります）。MOCO は周波数の精度が劣りますが，STOP 状態からの解除が高速なので，低消費電力のシステムでは有効かと思います。



①INTP0 で検出して STOP 解除し，TM03（立下りエッジ間隔測定）を起動（STOP モードを解除する時間で対応可能なボー・レートの上限が決まります。この制限をなくすには，LIN バスのように別の信号タイミングを準備する必要があります。）

②最初の INTTM03 は無視（パルス間隔測定では，タイマが起動した最初のデータは無意味）

③2 回目の INTTM03 でキャプチャ値を取り込み

④3 回目の INTTM03 でキャプチャ値を加算し，4 ビット分の時間からボー・レートを算出

⑤４回目の INTTM03 で UART0 を設定して起動（RL78 の UART は RxD 信号がロウの状態で起動させても，受信動作は行わないので，このタイミングで起動可能です。これにより，送信側が連続して次のデータ（コマンド等）を送信してきても対応可能になります。）

計測するのは，4 ビット分で，8 ビットの半分でしかありません。それでも，4MHz の f_{CLK} で 38.4kbps のボー・レートの場合でも 416 カウント程度となります。SDR レジスタでの有効な設定値が 7 ビット（1%程度の精度）なので，計測時の 1～２カウント程度の誤差は十分な精度と考えられます。なお，単純に 7 ビットで計算すると，1/128 で 1%弱（1 ビット分）になりますが，これは 7 ビットの刻みをそのまま求めただけです。1 ビット分でも十分な精度ではありますが，実際の処理では，設定値を四捨五入して求めることで，誤差は半分の 1/2 ビット分となります。

この部分の実際の処理では INTP0 と INTTM03 を使用しますが，例として INTTM03 割り込み処理を以下に示します。

```
static void __near r_tau0_channel3_interrupt(void)
{
/*
    measured data(TDR03) is around 415 / bit
*/
    uint16_t work;

    g_tm03_flag--;                          /* INTTM03 回数カウント          */

    switch ( g_tm03_flag )
    {
        case  0x00 :                        /* UART0 起動タイミング          */
                TT0L = 0x08;                /* TM03 動作停止                 */
                g_cpu_mode ^= 0b01100000;   /* メイン関数への完了通知        */

                R_UART0_Start();            /* UART0 の起動                  */
                break;

        case  0x01 :                        /* キャプチャ完了                */
                g_time_acc += TDR03;        /* 2 回目のデータを積算          */
                g_time_acc = g_time_acc >> 2;   /* 1 ビット長を算出          */

                work = 0;                   /* プリスケーラの初期値設定      */
                while ( g_time_acc > 256 )  /* SDR での分周が 256 以下まで   */
                {                           /* プリスケーラで分周する        */
                    work++;
                    g_time_acc = g_time_acc >> 1;
                }

                SPS0 = work;                /* プリスケーラの設定            */
                g_time_acc--;               /* SDR 設定値の四捨五入用補正    */
                g_time_acc &= 0x00FE;       /* 四捨五入処理                  */
                SDR00 = ( g_time_acc << 8 );    /* 送信ボー・レート設定      */
                SDR01 = ( g_time_acc << 8 );    /* 受信ボー・レート設定      */
                break;

        case  0x02 :                        /* キャプチャ値積算開始          */
                g_time_acc += TDR03;        /* 1 回目のキャプチャ値積算      */
                break;

        default :                           /* 最初の INTTM03 ではデータ無視 */
                g_time_acc = 0x0004;        /* 積算値を初期化                */
     }
-
```

変数 g_tm03_flag が INTTM03 の割り込み回数をカウントするカウンタで，INTP0 割り込み（TM03 を起動し，INTP0 割り込みを禁止）で 4 に設定しています。変数 g_time_acc は TM03 でのキャプチャ値を積算するための変数で，初期値として 4 を設定しています。このうちの 2 はキャプチャ値が実際のカウント値―1 であることの補正用で，残りの 2 は，1 ビット分の時間計算時の四捨五入用です。

変数 g_tm03_flag が 0x01 になったら，2 回の計測が完了です。ボー・レート用の設定値を計算します。算出した値をプリスケーラと SDR レジスタに設定することで，ボー・レート補正を行います。

UART の通信速度を決定する分周比を指定する項目の関係を示します。

必要な分周比から，SDR レジスタ以降での分周比が 256 以下になるようにプリスケーラでの分周比（2^n）を求めます。残りを SDR レジスタに設定するときの補正（－1）と結果の四捨五入を合わせて処理します。得られた値を SPS と SDR レジスタに設定することで，ボー・レート補正が実現できます。

変数 g_tm03_flag が 0x00 になったら，UART を起動します。このようにタイミングを切り替えることで，最後のロウを UART がスタート・ビットと誤解するのを防いでいます。

基本的に，このボー・レート補正処理は全て割り込みで処理されます。変数 g_cpu_mode はメイン処理に対してボー・レート補正処理の状態を通知するために使用しています。ビット 6 が補正処理中を示し，ビット 5 が補正処理の完了を示しています。変数 g_cpu_mode のビット 5 がセットされたら，その後の処理（実際の通信処理）を行います。

また，ボー・レート補正の許可はメイン処理が INTP0 の割り込みのマスクを解除して，割り込み許可にすることで処理します（通常の通信中には，INTP0 の不要な割り込みをマスクする必要があります）。

まとめ

今回の方法は，比較的簡単な処理で通信相手のボー・レートに合わせた通信を行うことができます。調歩同期通信は，送信側と受信側でボー・レートを予め決めておかないと通信できないのですが，このボー・レート補正方法を使うと，受信側（この場合には RL78/G11 側）が勝手に送信側のボー・レートに合わせてくれるので，比較的簡単に通信を行うことが可能となります。