RL78/G15-FPB で遊んでみる(1)

RL78/G15-FPB で Arduino IDE を使ってスケッチを作成しようとしたのですが、どうやってもエラーとなってしまいました。

これ以上は進まないので、CS+環境でプログラムを作成することにしました。プログラムとしては、SPI(CSI)を使った LED 表示器の制御です。16MHz 動作で 4Mbps の通信を行うので、コード生成された割り込み処理プログラムでは、プログラムがフリーズしてしまうものです。SC の生成下コードでも割り込み処理プログラムは同じだったので、同様の問題が予想されます。この条件でも問題がない CSI 用のライブラリを作成して動作確認を行いました。

記載内容は以下のようになっています。

（1）はじめに

RL78/G22-FPB と同じころに RL78/G15-FPB も入手しましたが、ArduinoIDE で対応しているはずなのに、L チカすらコンパイルでエラーになってしまいました。調べてみると、以下のような制限条件が見つかり、ArduinoIDE の 1.8.19 でしか対応していないことが記載されていました。

Arduino公式と互換ボード

- 2022年6月、ルネサスは、Arduinoとパートナーシップを締結しております
  - https://www.renesas.com/about/press-room/renesas-announces-investment-popular-open-source-company-arduino-access-huge-developer-community
- これ以前はルネサスは互換ボードとしてArduino対応ボードを開発しておりました
- 本GitHubの掲載情報はこの互換ボードに関する情報となります
- Arduinoが開発したArduino公式ボードおよび対応するソフトウェア等の情報については以下一覧表をご参照ください
  - Arduino公式ボードとしてルネサスマイコン RA6M5 が搭載された第一弾ボードは Portenta C33 です

Product Name	MCU	product link	software link	purchase link	setup guide/datasheet
Portenta C33	RA6M5	link	link	link(now preparing)	link

対応ボード

- 以下の表にAruduino IDEに対応しているFast Prototyping Boardを示します。
- Fast Prototyping Boardによっては、「追加のボードマネージャのURL」の設定が必要になります。
- 「追加のボードマネージャのURL」の設定方法は、各ボードのクイックスタートガイドをご参照ください。

Board	IDE 2.0.4以降	IDE 1.8.19	追加のボードマネージャのURL
RL78/G23-64p Fast Prototyping Board	Library v2.0.0以降	Library v2.0.0以降	-
RL78/G22 Fast Prototyping Board	Library v2.0.0以降	Library v2.0.0以降	-
RL78/G16 Fast Prototyping Board	Library v1.0.0以降	Library v1.0.0以降	RL78/G16 ボードマネージャのURL
RL78/G15 Fast Prototyping Board	-	Library v1.0.0以降	RL78/G15 ボードマネージャのURL

注：RL78/G15 Fast Prototyping Boardを使用する場合は、Arduino IDE 1.8.19を使用してください。

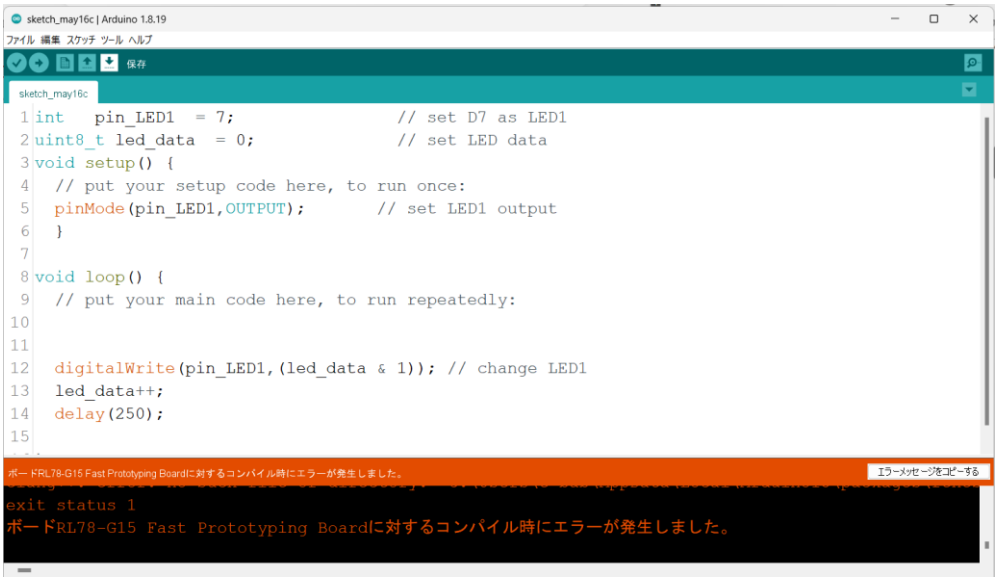
また、以下のリンクに RL78/G15-FPB を使った L チカの動画が見つかり、問題なくコンパイルできていました。

https://youtu.be/faAaMiONW7o

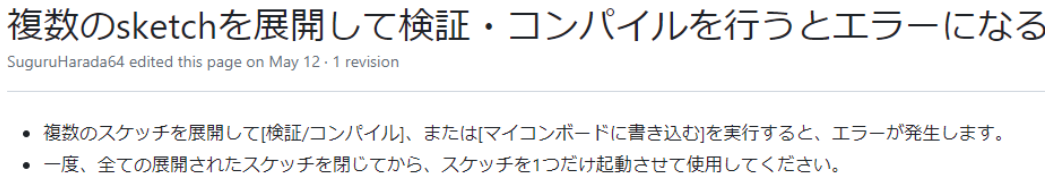
しかしながら、次ページに示すように、似たようなスケッチをコンパイルしたところエラーとなってしまいます。問題なくコンパイルできているはずのスケッチの内容を ArduinoIDE 1.8.19 でコンパイルしても同じくエラーとなってしまいます。

動画で使っている環境がよくわかりませんが、「4 行目」や「9 行目」の Arduino IDE が自動生成しているコメントがないので、何か公開されている Arduino IDE とは異なる環境でやっているの

ではないかと想像していますが、全く進展がありません。



また、以下のような注意事項も確認して、Arduino IDE は 1 つしか開いていないのですが、症状は改善されませんでした。



ということで、RL78/G15-FPB は Arduino IDE では手詰まりとなってしまいました。

さすがに、このままでは示しがつかないので、4 年ほど前に、CSI 通信の評価用に Amazon で購入した MAX7219 を使った 8×8×4 桁の LED 表示器を CS+環境で動かすことにしました。
元々は RL78/G12 で CSI の高速連続送信を用いて対応していたのですが、コード生成で生成されたコードでは、システムクロックに対して通信速度が高いと、最後のデータに対して転送完了割り込みに切り替える処理が間に合わず、最後の割り込みが発生しなくなる問題がありました。これは、割り込み処理の中だけで対応することには、無理があります。つまり、転送完了をきちんとサポートするようなプログラム構成にすれば、簡単に対応可能です。当時は、この段階で留めていた（忘れていたとも言う）ものです。FPB では I2C（Arduino IDE ベースでは Wire）は結構使っていたのですが、SPI はやってなかったことに気付いて、RL78/G15-FPB で復活させることにしました。

使用する LED 表示器はカスケード接続することで、桁をどんどん増やすこともできますが、今回は 4 桁でとりあえず動作させることにします。

（2）CS+環境での使用設定

RL78/G15 はコード生成ではなく、SC（スマート・コンフィグレータ）での対応となります。例によって、初期設定しか使いたくないのですが、SC ではコンポーネントごとにそれが可能です。これを利用して、自由に、不要な API は組み込まないようにできます。RL78/G15 はメモリが少ないので、この機能を利用して、できるだけ使用するメモリを少なくします。

また、RL78/G15-FPB の Arduino コネクタで SPI を使う場合には、下の図に示す信号配置になることから、CSI00 を P03～P05 で使う必要があります。

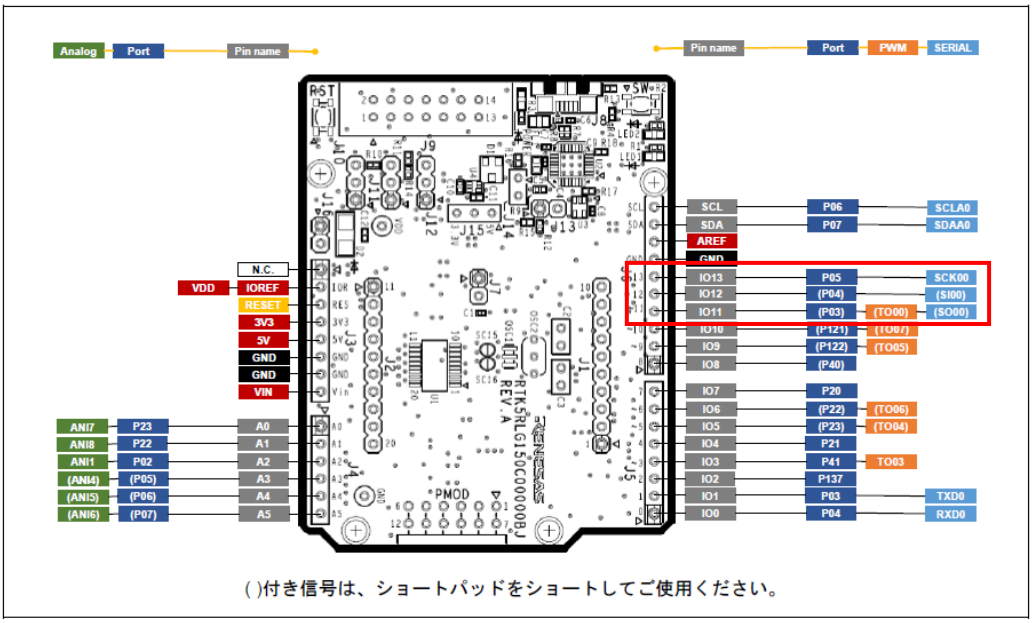
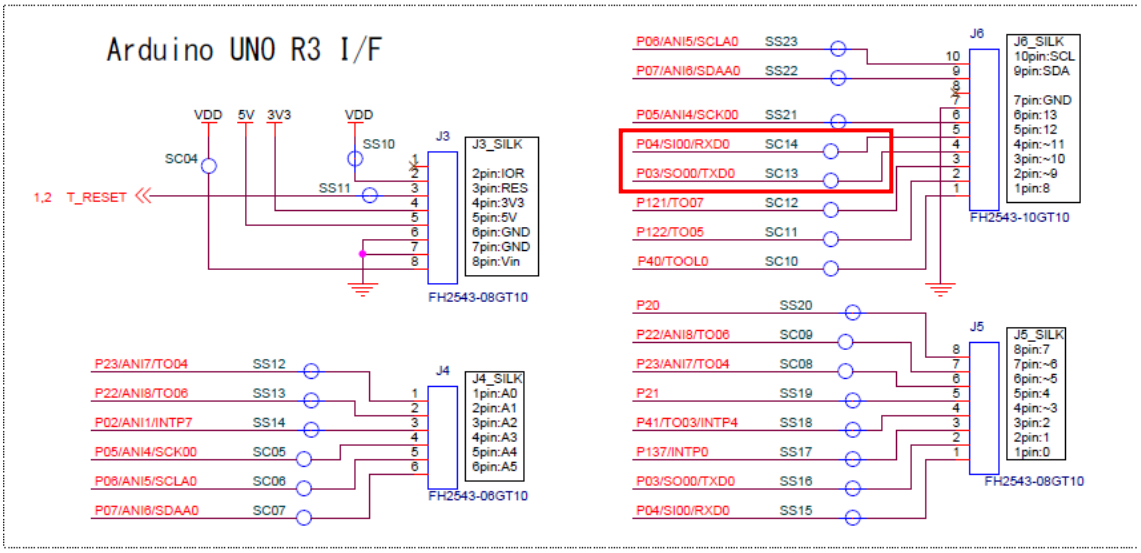


図 5-1: Arduino™ コネクタピン配置

ここで、注意する必要があるのは、P04 と P03 は「(P04)」、「(P03)」書かれていることです。この()は、ハードウェア マニュアルでは「周辺 I/O リダイレクション・レジスタ 0-3（PIOR0-3）」でディフォルト状態から変更することを意味しています。ところが、RL78/G15-FPB ではそれだけではありません。

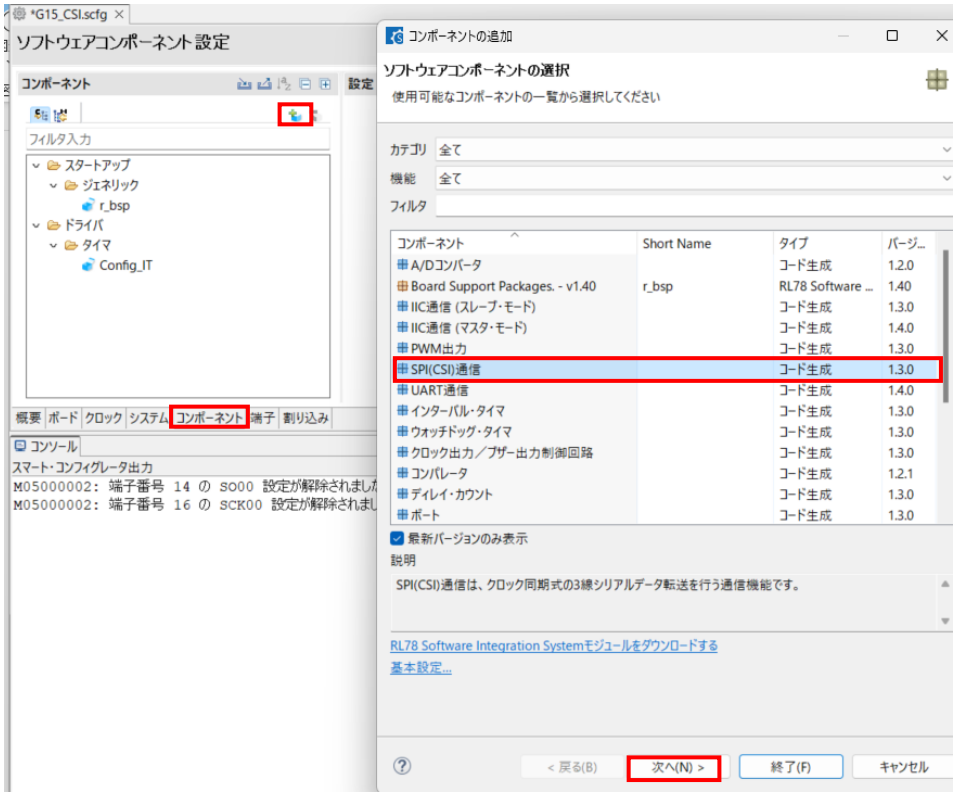
回路図に書かれているように、この２本の信号はArduinoコネクタには接続されていないので、

SC13 と SC14 を半田でショートさせる必要があります。



周辺 I/O リダイレクションについては、スマート・コンフィグレータで対応する必要があります。

CSI00 を使用するには、まず「コンポーネント」タグで、「コンポーネント追加」ボタンをクリックし、「通信の「Config_CSI00」を以下のように設定します。



「次へ(N)>」をクリックします。

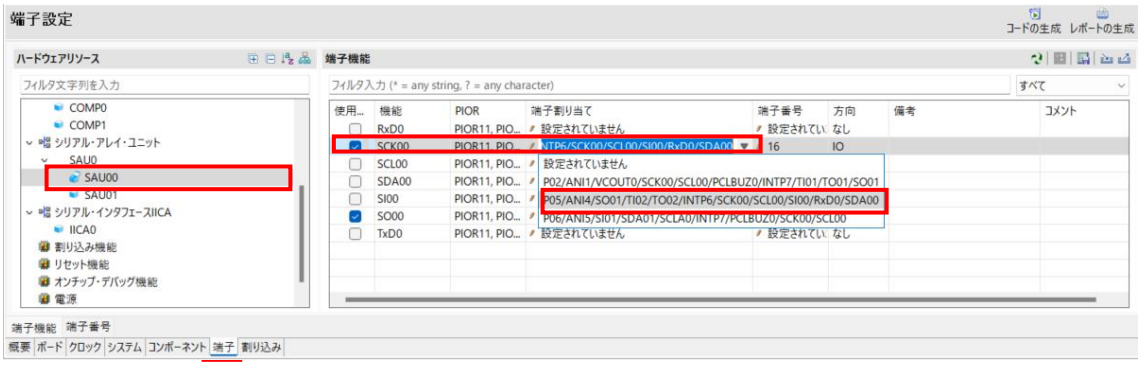
下の画面が表示されたら、「動作」を「送信」にして「終了(F)」をクリックします

CSI00 の設定は以下のように設定します。マスタ(送信)動作、「連続転送モード」、「8 ビット長」、「データ転送方向」は「MSB」ファースト、「データ送受信タイミング設定」は「タイプ 4」、「転送レート」は「4000000」bps、「割り込み」は「レベル 0(高優先順位)に設定しておきます。最後の「コールバック機能設定」は意味がないので無視しておきます。

これに加えて、今回は 2 つの設定を行います。

次ページに示すように、「Config_CSI00」を右クリックして、プルダウンメニューを表示し、「初期化 API のみ出力」にチェック入れます。これは、コード生成のころから文句を言ってきた通信関係の API の問題点を避けるために面倒な初期設定だけを使用するように指定するものです。スマート・コンフィグレータでは、個々の API 単位で設定できるようなっているので、これを利用することにします。

続いて、周辺 I/O リダイレクションについての設定を行います。まず、「端子」タグをクリックします。「ハードウェリソース」は、「シリアル・アレイ・ユニット」で SAU00 を選択します。すると、「端子機能」には、関係する端子機能が表示され、その中で使用するところにチェックが入っています。(下の画面イメージ)

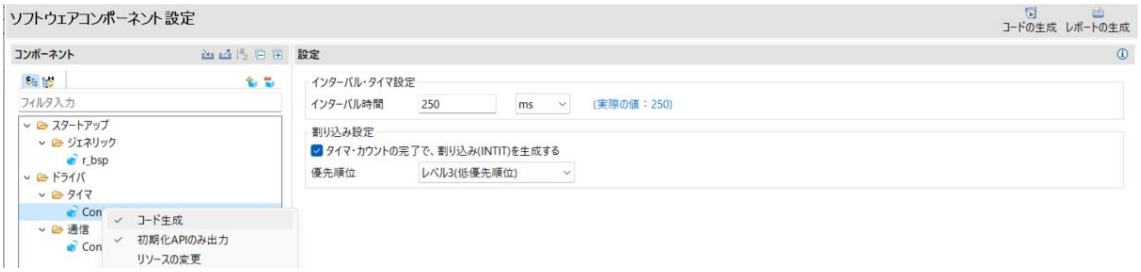


「機能」の「SCK00」をクリックして、右側の「端子割り当て」をクリックすると、選択肢が表示されるので、「P05・・・・・」を選択します。

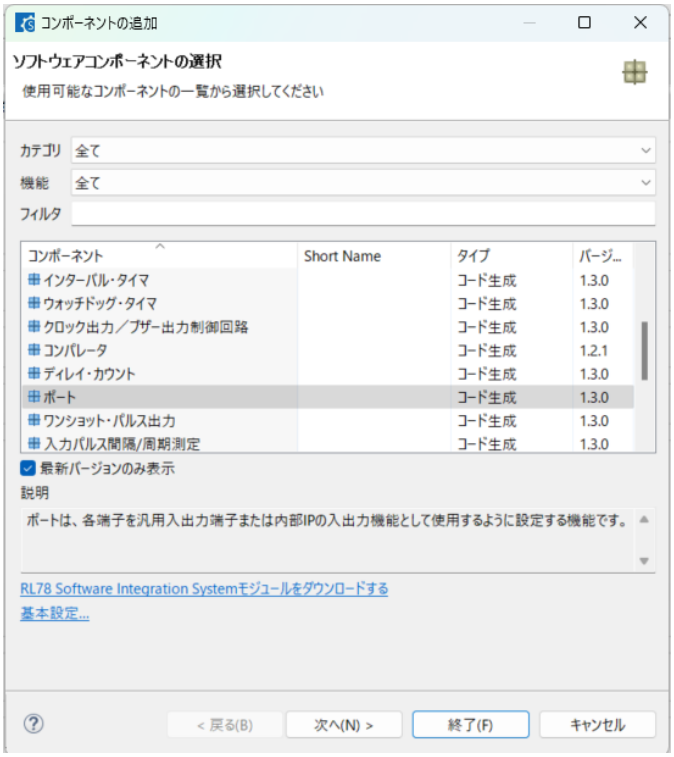
同様に「SO00」も「P03・・・・・・」を選択します(SCK00 を「P05・・・・・」にすると変更されているはずです)。

これで、SCK00 信号が端子番号 16、SO00 信号が端子番号 14 になっていることが分かります。受信はしないので、設定は以上です。

CSI00 以外に使用するのは、12 ビット・インターバルタイマとポートです。12 ビット・インターバルタイマは 250msに設定します。



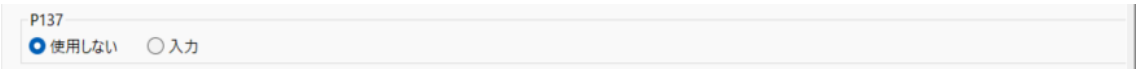
最後はポートの設定です。



ポートとしては、MAX7219 の CS 信号として P04（CSI00 の SI00 信号の兼用端子）を使用します。

RL78/G15-FPB のオンボード LED（P20、P21）と SW（P137）も使用します。
出力は初期値を 1 に設定しておきます。

P137 はあえて入力には設定していません。そもそも、P137 は設定するレジスタは存在しません。ここを入力に設定すると、INTP0 として使うときに問題が起こります。ここを設定しなくてもちゃんと入力として使用できます。端子を INTP0 で使用して、INTP0 割り込みが入った時にその時の端子の状態を P137 で確認することができます。



最後に CSI00 の割り込みでは、処理の高速化のためにレジスタバンク 3 を使用するように設定していますがここでの設定は使われません(初期設定以外は使用しないので、ここの設定も無意味です)。

実際のプログラムでは、割り込み関数は以下のように宣言されています。

```
21 /***********************************************************************↓
22 Pragma directive↓
23 ***********************************************************************/
24 #pragma interrupt r_csi00_interrupt(vect=INTCSI00,bank=RB1)↓
```

(3) CSI の連続転送(送信)処理

最初の方で触れたように、ここでは、CSI00 での連続転送の具体的な処理例を示します。前でも書いたように、転送終了の確認までを含めた処理としています。

CSI の初期化以外の制御関数は" r_CSI_lib.c"に纏められています。サポートしている API 関数の一覧を示します。

関数名	概要
R_CSI00_Start	CSI00 の動作開始
R_CSI00_Stop	CSI00 の動作停止
R_CSI_Tx	CSI00 データ送信処理を起動し、完了する
R_CSI_Tx_start	CSI00 データ送信の起動処理を行う.
R_check_status	CSI00 の通信ステータスを確認する
R_CSI_Init	CSI00 のパラメータ(ステータス)を初期化
R_CSI_wait_empty	CSI00 の送信完了を待つ

API 関数の仕様を以下に示します。

[関数名]　R_CSI00_Start

関数概要	CSI00 の動作開始のための処理を行う
プロトタイプ宣言	void R_CSI00_Start(void):
機能概要説明	SCK00 のレベルをローに設定し、CSI00 の動作を起動し、割り込み関係を初期化する。
引数	なし
リターン値	なし
概要	なし

[関数名]　R_CSI00_Stop

関数概要	CSI00 の動作を停止する
プロトタイプ宣言	void R_CSI00_Stop(void):
機能概要説明	CSI00 の動作を停止し、出力を停止し、割り込みを禁止する。
引数	なし
リターン値	なし
概要	なし

[関数名]　R_CSI00_Stop

関数概要	CSI00 の動作を停止する
プロトタイプ宣言	void R_CSI00_Stop(void):
機能概要説明	CSI00 の動作を停止し、出力を停止し、割り込みを禁止する。
引数	なし
リターン値	なし
概要	なし

［関数名］　R_CSI_Tx

関数概要	CSI00 でデータを送信する
プロトタイプ宣言	MD_STATUS　R_CSI_Tx(uint8_t　*　const　tx_buf,　uint16_t tx_num);
機能概要説明	CSI00 から、第 1 パラメータで示されたバッファから第 2 パラメータで示された数のデータを送信する。送信が完了したら、戻る。
引数	第 1 引数　　送信データを格納したバッファのアドレス 第 2 引数　　送信するデータのバイト数
リターン値	0x00:MD_OK　　送信を正常完了 0x80:MD_ERROR　　オーバーラン・エラー 0x81:MD_ARGERROR　引数(データ数)エラー
概要	この関数では、MD_ERROR は発生しない

［関数名］　R_CSI_Tx_start

関数概要	CSI00 でデータの送信を開始する
プロトタイプ宣言	MD_STATUS　R_CSI_Tx_start(uint8_t　*　const　tx_buf,　uint16_t tx_num);
機能概要説明	CSI00 から、第 1 パラメータで示されたバッファから第 2 パラメータで示された数のデータの送信を開始したら戻る。この関数で開始した送信が完了する前に、再度送信を行おうとすると、オーバーラン・エラーを戻す。
引数	第 1 引数　　送信データを格納したバッファのアドレス 第 2 引数　　送信するデータのバイト数
リターン値	0x00:MD_OK　　送信を正常完了 0x80:MD_ERROR　　オーバーラン・エラー 0x81:MD_ARGERROR　引数(データ数)エラー
概要	

［関数名］　R_check_status

関数概要	CSI00 の通信状況(ステータス)を確認する
プロトタイプ宣言	uint8_t R_check_status(void);
機能概要説明	CSI00 の通信状況を戻す
引数	なし
リターン値	0x00:MD_OK　　送信を正常完了 0x01:ビジー　　CSI00は送信中 0x80:MD_ERROR　　オーバーラン・エラー 0x81:MD_ARGERROR　引数(データ数)エラー
概要	この関数では、MD_ERROR は発生しない

［関数名］　R_CSI_Init

関数概要	CSI00 を初期化する
プロトタイプ宣言	void R_CSI_Init(void);
機能概要説明	CSI00 の通信状況(ステータス:g_status)をクリアする
引数	なし
リターン値	なし
概要	なし

［関数名］　R_CSI_wait_ready

関数概要	CSI00 送信完了を待つ
プロトタイプ宣言	void R_CSI_wait_ready(void);
機能概要説明	CSI00 の通信状況(ステータス:g_status)をチェックして、通信が完了するのを待つ。残り送信データ数が 0 になり、CSI00が通信中でなければ、CS信号をネゲートし、バッファ空き割り込みに設定し、通信ステータスを通信完了にして戻る。
引数	なし
リターン値	なし
概要	なし

実際に通信を行う場合には、以下の手順で行います。

①CSI00の動作開始(R_CSI00_Start()を呼び出す。)
②ベクタ割り込みを許可する。
この状態で、
③送信したいデータを引数にして R_CSI_Tx()を呼び出す。

この処理方法では手順は簡単ですが、送信が開始すると、割り込み処理以外は実行できません。送信中に別の処理も行いたい場合には③のところを、以下のようにします。

③ ‘送信したいデータを引数にして R_CSI_Tx_start()を呼び出し、正常に送信開始したことを確認する。
④他の処理を実行する。
⑤R_check_status()を呼び出して、戻り値の LSB が 1 なら、④に戻る。
⑥戻り値の LSB が0ならば、R_CSI_wait_ready()を呼び出して、通信を完了する。

（4）コード生成、SC の生成コードの問題点への対応

コード生成等の通信関係の API では、ベアメタル処理用（？）に通信は起動するだけで、その後のケアとしては、割り込みでのコールバック処理をユーザが指定できるようになっているだけです。そのために、初心者が混乱しており、かふぇルネでもトラブルの報告（スレッド）が結構見受けられました。

このギャップを埋めるために、”r_CSI_lib.c”では、同様な’R_CSI_Tx_start()’関数に加えて、通信の完了検出をサポートする’R_CSI_wait_ready ()’関数等の手段を準備しています。しかも、この関数が起動するだけであることを明確にするために名前に”start”を付けています。

さらに、Arduino API のようにブロッキングで処理する”R_CSI_Tx ()”関数も準備しています。

また、CSI 送信でのバッファ空き割り込みによる高速な通信に対して、最後の送信データに対する割り込み処理が間に合わないと、送信完了を検出できない問題があります。これは、単に割り込み処理の中だけでは完全に対応することはできません。これは、CPU の割り込み処理速度と CSI の通信速度の相対的な関係で変わってきます。しかも、CPU の割り込み処理速度には他の処理による CSI の割り込みの遅延も含めて考えないといけません。このように、CSI 通信割り込み処理プログラムだけでは決まらないようなことの影響を割り込みの中だけで対応するのは不可能ではないにしても困難です。そのような処理を割り込みの中でやることは、割り込み処理の時間を不要に占有してしまい、システム全体の効率を悪化させるだけです。

現状のコード生成された通信処理の割り込みでのコールバック処理には、ある危険性があります。それは、「割り込み処理では処理時間はできるだけ短くする」のが常識ですが、コールバック処理では、結構な処理時間が必要な処理をやろうとしているのを見かけることがあります。これは、絶対にやってはいけないことです。この処理を行っている間は、他の割り込みが受け付けられなくなることが考えられます。特に通信割り込みの割り込み優先順位は通常は高く設定するので、多重割り込みも受け付けない可能性があります。実際に、今回のプログラムでも CSI00 の割り込みは最高優先順位に設定してあります。通信した結果の処理は、処理の優先度が割り込みほど要求されないはずです。つまり、通信割り込みのコールバックでは通信完了フラグをセットする程度としておき、さっさと割り込み処理を終了させるべきです。最近は、コード生成への要望が受け入れてもらえなくて、フラストレーションがたまっているのか（おそらく年齢のせいかもしれませんが）、愚痴ばかりになってしまいました。

今回のプログラムですが、CSI00 の通信速度は 4Mbps です。これに対して、システムクロックは 16MHz で通信速度の 4 倍でしかありません。つまり、1 バイトの転送に対して 32 クロックしか処理時間が割り当てられません。

また、RL78/G15 では、割り込みの受付けまでに必要な時間は、次ページのハードウェア マニュアルの記載内容を示すように最低でも 9 クロックかかります。これは、1 クロックで実行可能な命令

を実行している場合の時間です。RL78 はほとんど 1 クロックで処理できますが、分岐関係の命令では 6 クロック程度かかることがあるので、その場合には 14 クロックとほぼ最大時間かかる可能性もあります。

14.4.1　マスカブル割り込み要求の受け付け動作

マスカブル割り込み要求は、割り込み要求フラグがセット（1）され、その割り込み要求のマスク（MK）フラグがクリア（0）されていると受け付けが可能な状態になります。ベクタ割り込み要求は、割り込み許可状態（IE フラグがセット（1）されているとき）であれば受け付けます。ただし、優先順位の高い割り込みを処理中に低い優先順位に指定されている割り込み要求は受け付けられません。

マスカブル割り込み要求が発生してからベクタ割り込み処理が行われるまでの時間は**表 14－4** のようになります。

割り込み要求の受け付けタイミングについては、**図 14－8**、**図 14－9** を参照してください。

表 14－4　マスカブル割り込み要求発生から処理までの時間

	最小時間	最大時間[注1]
処理時間	9 クロック	16 クロック

残りは、18 クロックになります。さらに、割り込み禁止期間があると、その分は待たされますし、下に示すような、割り込み要求が保留される命令があると、その分＋1 クロックは待たされるので、注意が必要です。

14.4.4　割り込み要求の保留

命令の中には、その命令実行中に割り込み要求が発生しても、その次の命令の実行終了まで割り込み要求の受け付けを保留するものがあります。このような命令（割り込み要求の保留命令）を次に示します。

それに対して、ここで使用している CSI00 割り込みの処理は以下のようになっています。

```
static void __near r_csi00_interrupt(void)
{

    if (g_tx_number > 0x00)          // 送信データ数確認
    {   // 残り送信データあり
        SIO00 = *gp_tx_data;         // 送信データ設定
        gp_tx_data++;                // 送信データ・ポインタ更新
        g_tx_number--;               // 残り送信データ数カウント
    }
    else
    {   // 送信データ書き込み完了

        if ( (SSR00L & TSFmn) )      // 送信状況を確認
        {       // 最終データ送信中
            SMR00 &= 0xFFFE;         // 割り込みタイミングを転送完了に
        }
        else
        {       // 最終データ完了
            CS_PORT = 1;             // MAX7219のCS/LOAD信号をネゲート
            SMR00 |= 0x01;           // バッファ空き割り込みに戻す
            g_status = 0x00;         // CSI00の通信完了フラグを設定

        }
    }
}
```

これをコンパイルしてシミュレータで逆アセンブル結果を見ると次ページのような命令が使われて

います。赤で囲んだ部分が割り込みタイミングの変更に関わる部分です。最初の部分がデータ数をチェックして条件分岐している部分です。残りデータ数が 0 なので、ここでは条件分岐することになります。すると合計 7 クロックかかることになります。下側の赤で囲んだ部分がその後の処理です。最初の if 文で 5 命令 6 クロックかかります。その後の SMR00 レジスタの書き換えで 5 クロックかかります。これで、ちょうど合計 18 クロックとぴったりになります。

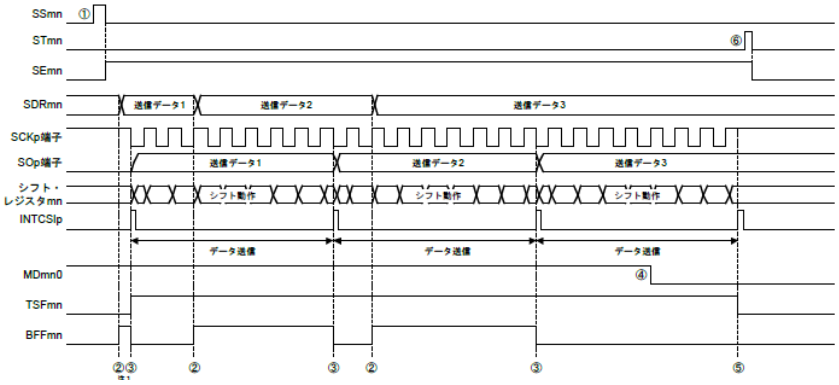
```
static void __near r_csi00_interrupt(void)
_r_csi00_interrupt@1:
00142       61df          SEL             RB1
    if (g_tx_number > 0x00)              // 送信データ数確認
00144       af04fb        MOVW            AX,!_g_tx_number@2
00147       6168          OR              A,X
00149       dd0e          BZ              $_r_csi00_interrupt@1+0x17
      SIO00 = *gp_tx_data;               // 送信データ設定
0014b       eb02fb        MOVW            DE,!_gp_tx_data@1
0014e       89            MOV             A,[DE]
0014f       9d10          MOV             SIO00,A
      gp_tx_data++;                      // 送信データ・ポインタ更新
00151       a202fb        INCW            !_gp_tx_data@1
      g_tx_number--;                     // 残り送信データ数カウント
00154       b204fb        DECW            !_g_tx_number@2
00157       61fc          RETI
      if ( (SSR00L & TSFmn) )            // 送信状況を確認
00159       8f0001        MOV             A,!SSR00L
0015c       318e          SHRW            AX,8H
0015e       31ad          SHLW            AX,0AH
00160       de0b          BNC             $_r_csi00_interrupt@1+0x2b
00162       341001        MOVW            DE,#110H
        SMR00 &= 0xFFFE;                 // 割り込みタイミングを転送完
00165       a9            MOVW            AX,[DE]
00166       08            XCH             A,X
00167       5cfe          AND             A,#0FEH
00169       08            XCH             A,X
0016a       b9            MOVW            [DE],AX
0016b       61fc          RETI
```

しかし、SMR00 レジスタへの書き戻しには3段のパイプライン処理を考えると、20 クロックと考えないといけないでしょう。

ところで、ハードウェア マニュアルの連続送信モードでのタイミングチャートは以下のようになっています。

(4) 処理フロー（連続送信モード時）

図 12－27 マスタ送信（連続送信モード時）のタイミング・チャート（タイプ 1：DAPmn=0, CKPmn=0）



このタイミングチャートの下には、注意として以下のように書かれています。

注意　シリアル・モード・レジスタ mn（SMRmn）の MDmn0 ビットは、動作中でも書き換えることができます。ただし、最後の送信データの転送完了割り込みに間に合わせるために、最終ビットの転送開始前までに書き換えてください。

つまり、ここで一気に 4 クロック分が減ってしまうことになり、SMR00 例スタの書き換えが間に合わない可能性が高くなってしまうということです。

参考までに、SC の生成する CSI00 の割り込み処理の内容を以下に示します。

```
/***********************************************************************************************************************
* Function Name: r_Config_CSI00_interrupt
* Description  : This function is INTCSI00 interrupt service routine.
* Arguments    : None
* Return Value : None
***********************************************************************************************************************/
static void __near r_Config_CSI00_interrupt(void)
{
    if (g_csi00_tx_count > 0U)
    {
        SIO00 = *gp_csi00_tx_address;
        gp_csi00_tx_address++;
        g_csi00_tx_count--;
    }
    else
    {
        if (0U == (SSR00 & _0040_SAU_UNDER_EXECUTE))
        {
            r_Config_CSI00_callback_sendend();    /* complete send */
        }
        else
        {
            SMR00 &= (uint16_t)~_0001_SAU_BUFFER_EMPTY;
        }
    }
}
```

殆ど変わりませんが、赤で囲んだ部分の判定条件が逆になっています。つまり SMR00 を書き換えるときには条件分岐で分岐が必要だということです。そのため、書き換えは条件分岐命令で 2 クロック余計に必要になるということです。細かなことですが、”r_CSI_lib.c”では、そこまで考えてプログラムを作成しています。古いハードウェアの人間なので、ついつい細かなことまで気にしてしまっています(参考までに)。

さて、SMR00 レジスタの書き換えが間に合わなくて、最後の割り込みが間に合わないことへの対策ですが、それが” R_CSI_wait_ready()”関数による送信完了待ちです。

ここで、CSI00 の送信完了条件をチェックしています。送信完了条件は2つあります。1 つ目は、残り送信データ数が 0 であることで、もう 1 つはバッファにもシフトレジスタにもデータが残っていないことです。これにより、最後の割り込みが発生しなくても対応できるようになります。実際のプログラムを次ページに示します。

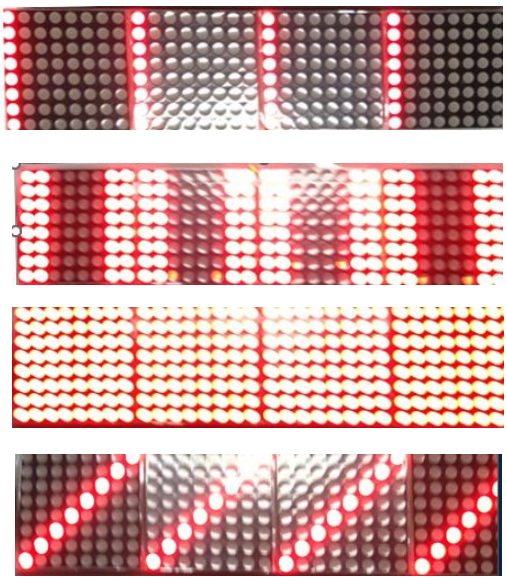
ここでは、送信完了を確認したら、次の通信のための準備(243～245 行目)も行っています。コールバック処理で逃げるのではなく、このような処理にしてもらうとユーザは楽になると思います。

```
232 void R_CSI_wait_ready(void)↓
233 {↓
234 ↓
235     while ((g_status & 0x0F) == 0x01)   // g_statusの下位をチェック↓
236     {↓
237         if ( 0x0000 == g_tx_number )    // 残りデータ数をチェック↓
238         {   // 残り送信データ数がない場合↓
239 ↓
240             if ( ( SSR00L & 0b01100000 ) == 0x00 )↓
241             {   // CSI00に送信中のデータがない場合↓
242 ↓
243                 CS_PORT = 1;            // MAX7219のLOAD信号をネゲート↓
244                 SMR00 |= 0x01;          // バッファ空き割り込みに戻す↓
245                 g_status = 0x00;        // CSI00の通信完了フラグを設定↓
246 ↓
247             }↓
248 ↓
249         }↓
250 ↓
251     }/* End of 送信完了 */↓
252 ↓
253     NOP();↓
254 ↓
255 }↓
```

（5）プログラムについて

最後になりましたが、今回のプログラムについて説明します。

今回のプログラムは、単なる遊びです。ドット・マトリクス LED に動く簡単なパターンを表示していくだけです。次は、きちんと使える（参考になる）プログラムにしますが、今回はあくまで CSI の動作確認用なので、ご容赦ください。一部のパターンを下に示します。



main 関数がスタートすると、1 秒待ちして SW が押されるのを待ちます。これは、RL78/G15 は 2.7V 程度で起動しますが MAX7219 は 5V 動作のためにスレーブ側の準備が整うのに時間がかかります。これを待つために、このような起動処理になっています。スレーブとの通信を行う場合には、このような待ち合わせを行わないと、うまく通信できなくなる場合があります。

```
152 void main(void)↓
153 {↓
154 ↓
155     R_MAIN_UserInit();↓
156 ↓
157   {↓
158 ↓
159     uint8_t loop = 0x00;          // LOOPカウンタ↓
160     uint8_t work;↓
161     uint8_t i;↓
162 ↓
163     R_wait_1s(1);                 // 1s待ち↓
164     R_wait_SW();                  // SWの押下を待つ↓
165 ↓
166 /*-------------------------------------------------↓
167     MAX7219の初期化を行う↓
168         ①DispT：ディスプレイテストモード解除↓
169         ②ShutD：スタンバイモードを解除↓
170         ③DecMode：デコードモードを解除↓
171         ④Intend：LEDの輝度を設定↓
172         ⑤ScanL：スキャン範囲を設定↓
173         ⑥No_OP：複数のMAX7219の初期化用↓
174 -------------------------------------------------*/↓
175 ↓
```

SW が押されると、MAX7219 の初期化が行われます。具体的には、const で定義されている初期化コマンドを送信していくだけです。ここではアドレスとデータの組み合わせデータを 9 回送信するだけです。

```
175 ↓
176     for ( i = 0 ; i < 9 ; i++ )↓
177     {↓
178 ↓
179         // 7219の初期化コマンド送信↓
180         work = R_CSI_Tx( (uint8_t *)&INIT1[i][0], 0x02 );↓
181 ↓
182 ↓
183         NOP();↓
184 ↓
185         while ( work )↓
186         {   // ありえないが、エラー時の処理↓
187 ↓
188             LED1 =0;              // エラーならLED1を点灯↓
189             HALT();               // エラーならHALTで停止↓
190 ↓
191         }↓
192 ↓
193     }/* End of 7219 Initialise  */↓
194 ↓
195     NOP();↓
196 ↓
```

エラーが発生することはないのですが、一応エラー処理(LED1 を点灯して HALT します。初期化

が完了すると、while で無限ループに入り、表示データを送信します。ここでは全ての MAX7219 には同じデータパターンを点灯させるようにしています。

```
    while (1U)↓
    {↓
↓
/*------------------------------------------------------↓
        表示データ送信↓
------------------------------------------------------*/↓
        for ( i = 0 ; i < 8 ; i++ )↓
        {↓
            work = R_CSI_Tx( (uint8_t *)&led_data[i][0],0x02 );↓
            ↓
            NOP();↓
            ↓
            while ( work )↓
            {↓
                LED1 =0;↓
                HALT();          // エラーならHALTで停止（ありえないが）↓
            }↓
        }↓
↓
        NOP();↓
        for ( i = 0 ; i < 4 ; i++ )↓
        {                              // NOPにより表示データをシフト↓
            work = R_CSI_Tx( (uint8_t *)&INIT1[5][0],0x02 );↓
            ↓
            NOP();↓
↓
        }↓
↓
    R_wait_250ms();↓
↓
↓
        for ( i = 0x00 ; i < 8 ; i++ )↓
        {                              // 次の表示データを更新する↓
            led_data[i][1] = SET_DATA1[loop][i];↓
        }↓
        loop++;                        // ループ・カウンタを更新する↓
        loop &= 0x0F;↓
↓
    }/* End of while loop   */↓
```

表示データは CSI00 に接続された MAX7219 から順に変更されていきます。プログラム中の NOP();にブレークポイントを設定して、ブレークポイントを有効にして実行すれば、後段の LED モジュール(MAX7219)に順にデータが動いていくのが分かります。

ブレークポイントなしで実行させると、250ms で表示データを書き換えることで、表示が変わっていくのが確認できます。

ということで、今回使用した CSI00 の制御ルーチンは、4Mbps の通信速度でも、問題なく動作していることが確認できました。

次回は、少し実用的なプログラムにします。