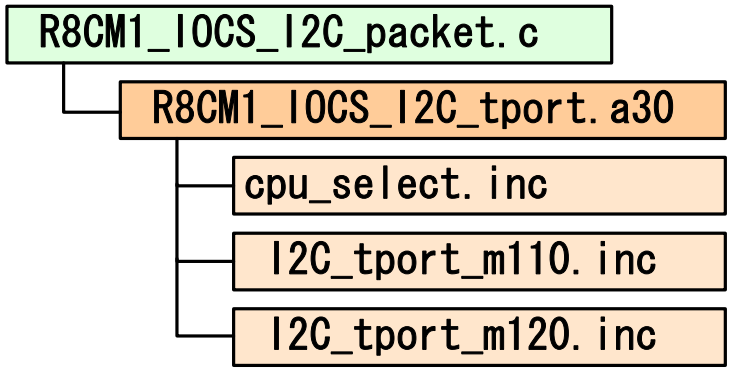
I2Cで、使用している I/O ポートの確認

R8C/Mシリーズの IOCSの I2Cに関わるソース プログラムは、以下の構成になります。

因みに IOCSは 当方で作成した R8C/Mシリーズ用の I/O周りの サブルーチン集です。



cpu_select.inc は R8C/M110Aか R8C/M120Aを 選択するための定義ファイルです。

その下の I2C_tport_m110.inc と I2C_tport_m120.inc が M110と M120で 異なる I/Oポートを 宣言したファイルです。

R8CM1_IOCS_I2C_tport.a30 先頭部分のコーディングです。 ここで、m120 か m110かを 切り替えています。

```
 .include  cpu_select.inc

.if  MPU_SEL==2
    .include  i2c_tport_m120.inc  ; M120A用マクロ
.ELSE
    .include  i2c_tport_m110.inc  ; M110A用マクロ
.ENDIF
```

次に I2C_tport_m110.inc と I2C_tport_m110.inc 内 先頭の マクロ定義を お見せします。

SCL_L　SCL信号を Lowに する。
SCL_H　SCL信号を Hiに する。
SDA_L　SDA信号を Lowに する。
SDA_H　SDA信号を Hiにする。
SDA_IN　SDAを 入力ポートにする。
SDA_OUT　SDAを 出力ポートにする。

```
;  R8C/M110A 用 I2C port宣言
; ------------------------------------
SCL_L .macro
      mov.b #00h, p3     ; SCL = 0 ( p3_7 = 0 )
      .endm

SCL_H .macro
      mov.b #80h, p3     ; SCL = 1 ( p3_7 = 1 )
      .endm

SDA_L .macro
      bclr  p1_7         ; SDA = 0 ( p1_7 = 0 )
      .endm

SDA_H .macro
      bset  p1_7         ; SDA = 1 ( p1_7 = 1 )
      .endm

SDA_IN      .macro
      mov.b #05Eh, pd1   ; p1_7 = In port
      .endm

SDA_OUT     .macro
      mov.b #0DEh, pd1   ; p1_7 = out port
      .endm
```

```
;  R8C/M120A 用 I2C port宣言
; ------------------------------------
SCL_L .macro
       bclr 5, rmch_p4    ; SCL = 0
       mov.b rmch_p4, p4 ; Port4 に 反映
      .endm
SCL_H .macro
       bset 5, rmch_p4    ; SCL = 1
       mov.b rmch_p4, p4 ; Port4 に 反映
      .endm
SDA_L .macro
       bclr 2, rmch_p4    ; SDA = 0
       mov.b rmch_p4, p4 ; Port4 に 反映
      .endm
SDA_H .macro
       bset 2, rmch_p4    ; SCL = 1
       mov.b rmch_p4, p4 ; Port4 に 反映
      .endm
SDA_IN .macro
      mov.b #20H, pd4    ; Dir.SCL=1 , Dir.SDA=0
      .endm
SDA_OUT .macro
      mov.b #24H, pd4    ; Dir.SCL=1 , Dir.SDA=1
      .endm
;    rmch_p4 は RAM上の Byte変数です。
```

という事で、M110Aの方が、RAM変数を使ってない分、シンプルですね。 それと、M110Aでは、I2C と SPI を 同時に使用しない。という事にしたので、I2C と SPIで、ポートが重複定義してないか、心配する必要も なかったですね。

まず、M110Aにて I2Cで使用する足ピンは、先ほどのソースを参考にすると
SCL = p3_7 で SDA = p1_7 です。
特に、高速性が必要な クロック信号は M110Aの場合 p3_7 を使うしか無いですよね。

よって SPIの場合も SCK信号は p3_7 になります。 あと、MOSI、MISO、SS0、SS1 は空いているピンに割り当てます。 特に、SS0、SS1は、早さの要求は 殆どないです。
それと、I2Cの様に 動的に信号の方向を 切り替える事は無いので、その分は 楽です。

R8C/M110A I2Cの場合		
Port	Pin	用途等
p1_1	14	空き
p1_2	13	空き
p1_3	12	空き
p1_4	11	TxD
p1_5	10	RxD
p1_6	9	PgmRxD
p1_7	8	SDA
p3_7	1	SCL

R8C/M110A SPIの場合		
Port	Pin	用途等
p1_1	14	MOSI
p1_2	13	SS0
p1_3	12	SS1
p1_4	11	TxD
p1_5	10	RxD
p1_6	9	PgmRxD
p1_7	8	MISO
p3_7	1	SCK

上の、R8C/M110Aの I/Oポート表は、左側がI2Cで、右側が SPIです。 SPI側もポート(端子)を 決めました。 当初の取り決め通り、SS信号は、SS0 と SS1 の 2つです。

SPIの場合 p1ポートには、4本の信号線を割り当てているので、RAM変数による、キャッシュが必要になります。

次は、M120Aの I/Oポート表を作成します。CPUのリセット端子、クロック入力端子は、I/Oポートの候補としては除外します。

CPU内蔵のクロックは、やや周波数が ぶれます。調歩同期のシリアル通信を行う場合はなるべく周波数が ぶれない方がいいです。 あと、リセット端子は、内蔵リセット回路があるのですが、電源 OFFから ONの時間間隔が 短いとリセット回路が、誤動作する事が あります。その現象を 過去に確認した事があるので、リセット端子には、CRと 逆流ダイオードを付けたリセット回路を付けています。

で、使えるポートのビットは Port.1が 8 bit、Port.3が 4 bit、Port.4が 2 bit で、計 14 bit です。 M120Aでは、調歩式シリアル通信、I2C、SPI の 3つを パラって使えるようにするので

一つのポート表に、3つのインタフェースの信号名を記入します。

Port	Pin	種別:用途等
p1_0	20	SPI: SS0
p1_1	19	SPI: SS1
p1_2	18	SPI: SS2
p1_3	17	空き
p1_4	16	TxD
p1_5	15	RxD
p1_6	14	PgmRxD
p1_7	13	空き
p3_7	2	タイマー割込みモニター
p3_5	9	SPI: SCK
p3_4	10	SPI: MOSI
p3_3	11	SPI: MISO
p4_5	12	I2C: SCL
p4_2	1	I2C: SDA

ピンアサインが、決まりました。

前ページにてRAM変数 という言葉を使いましたが、I/Oポートの キャッシュメモリのような使い方をします。

内蔵RAMメモリは、ウェイトサイクルは無いので、高速にアクセス出来ます。 それに対し I/Oポートを アクセスする場合は、ウェイトサイクルが 多少入ってきて、アクセス速度が遅くなります。

特に、I/Oポートに 1bitの書き込みを行う場合、CPUと I/Oポートの間は、8bit の バスラインで 接続されています。1bit 書き込むという事は物理的に出来ないのです。 過去の動画でも説明した事があると 思いますが、バスラインが 8bit である以上、8bit単位の読み出し、書き込みしか出来ないのです。

よって 1bit書き込む際は、一旦 その I/Oポートの状態を Byte単位で 読み出して レジスタに置きます。 そして 特定の bit指定データと 論理演算を行い その結果を、I/Oポートに Byte単位で書き込みます。

例えば、bit4 の ビット(10h)に 1 を立てる場合を 考えてみます。アセンブラソースで示します。

```
mov.b  p1, r0l   ; ポートp1 --> R0Lに読出し
or.b   #10h, r0l ; ビット演算
mov.b  r0l, p1   ; R0L --> ポートp1に書込み
```

という、3ステップになります。

ポートのアクセスが、2回生じるので ウェイトサイクルもその分増えて、遅くなります。

上の例では、間にビット演算を挟んで、2回I/Oアクセスを行っていますが、ビット演算は瞬時に終わるので、I/Oポートアクセスが 2回連続した状態になるので、この場合 特に遅くなります。

という事で、ビット単位で I/Oポートを アクセスする場合に、I/Oポートアクセス回数を減らす手段として、I/Oポートの状態を記憶する RAM変数の キャッシュを用意します。

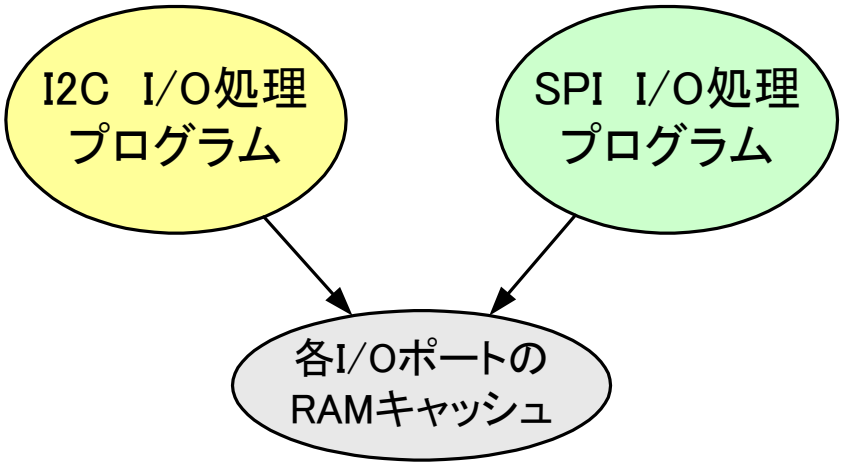
仮に ポート p1 のRAMキャッシュとして p1_rch という名前の変数を用意します。
最初の ポート初期化の時、p1の 初期化値を p1_rch にも 書き込んでおきます。

そして、実際に使用する場合を アセンブラで示します。

```
mov.b   p1_rch, r0l ; p1_rch --> R0Lに読出し
or.b    #10h, r0l     ; ビット演算
mov.b   r0l, p1       ; R0L --> ポートp1に書込み
mov.b   r0l, p1_rch ; R0L --> p1_rch に 書込み
```

1行命令が増えますが、この方が、結果として早いです。 オシロスコープで確認してます。

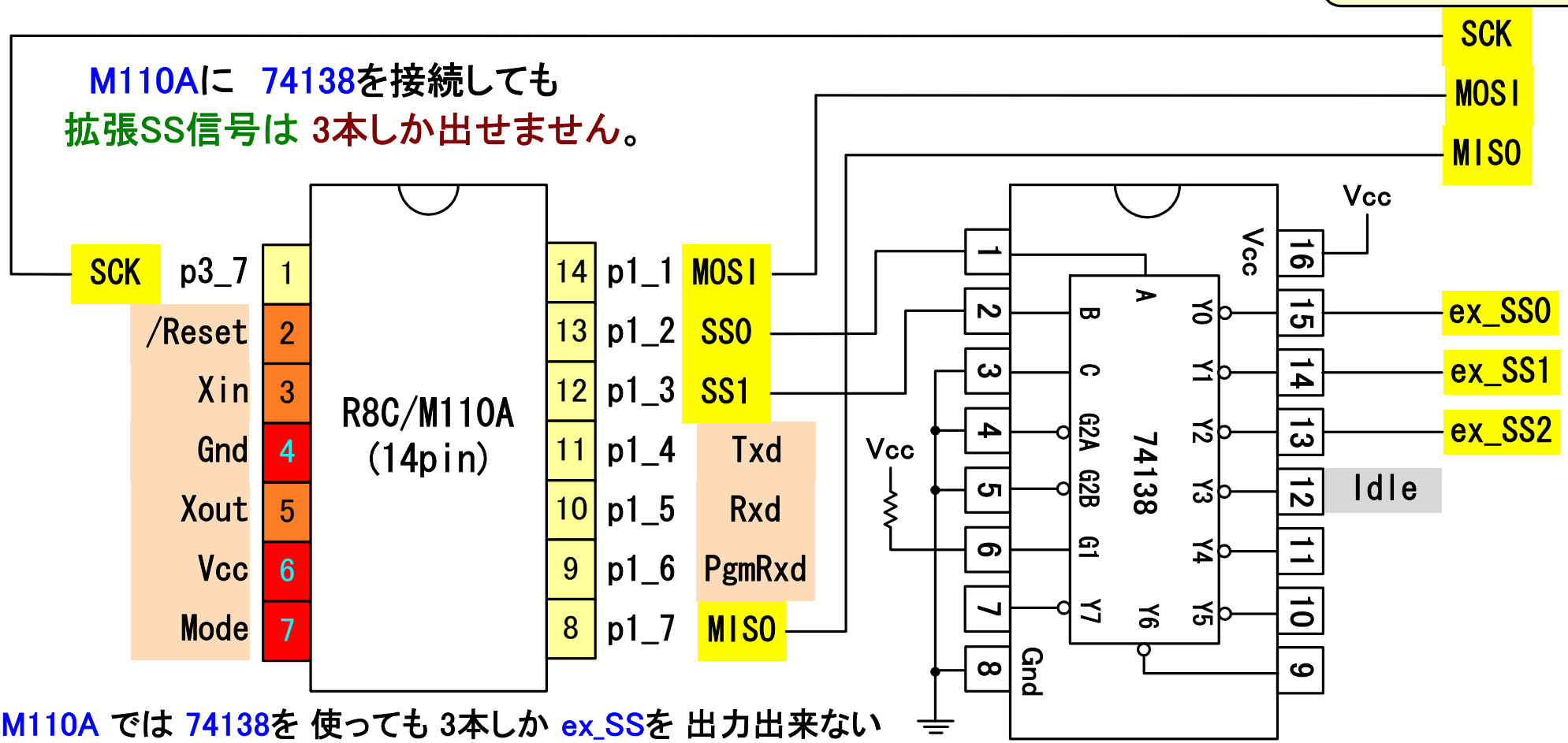
で、R8C/M120Aの場合 I/Oポートの RAMキャッシュを、I2Cと SPIの両方で使用する場合、各I/Oポートの RAMキャッシュを I2C と SPI にて 共通の物にしておかないと、I/Oポートのキャッシュが破綻します。
よって、I2C または SPIの I/O処理とは独立して、グローバルな変数として、I/Oポートの RAMキャッシュを 用意します。



R8C/M110Aマイコンと 138の接続図

デバイスに
向かう信号線

M110Aに　74138を接続しても
拡張SS信号は 3本しか出せません。



M110A では 74138を 使っても 3本しか ex_SSを 出力出来ない
ので、M110Aが 出力する 2本の SS信号で 間に合うなら　SS信号を直接デバイスに接続した方が いい。（注.1）

SPIの SS信号の拡張に関して

SPIの SS信号の 基本的 本数は
M110 が 2本で 、M120 が 3本です。

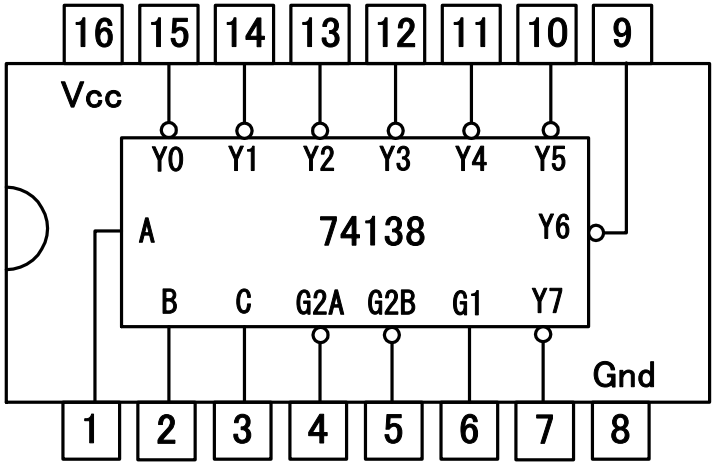
前々回、138の 動画で、チップセレクト信号の デコーダー ICの 74138を紹介しました。
確か、HC138 が あったと思うので、HC138 を 使用して、M120の 3本の SS信号を 7本に 増やしてみます。 138には A～C の 3bit バイナリ入力の3本の入力端子が 有ります。

ここに M120の 3本のSS信号を接続します。
SS0 = A端子、SS1=B端子、SS2=C端子です。

あと 138 には チップ選択の G端子が、あります。 これは、G1端子を Vcc に プルアップします。G2A端子と G2B端子を Gnd に落とします。これで、138は イネーブル状態に なります。

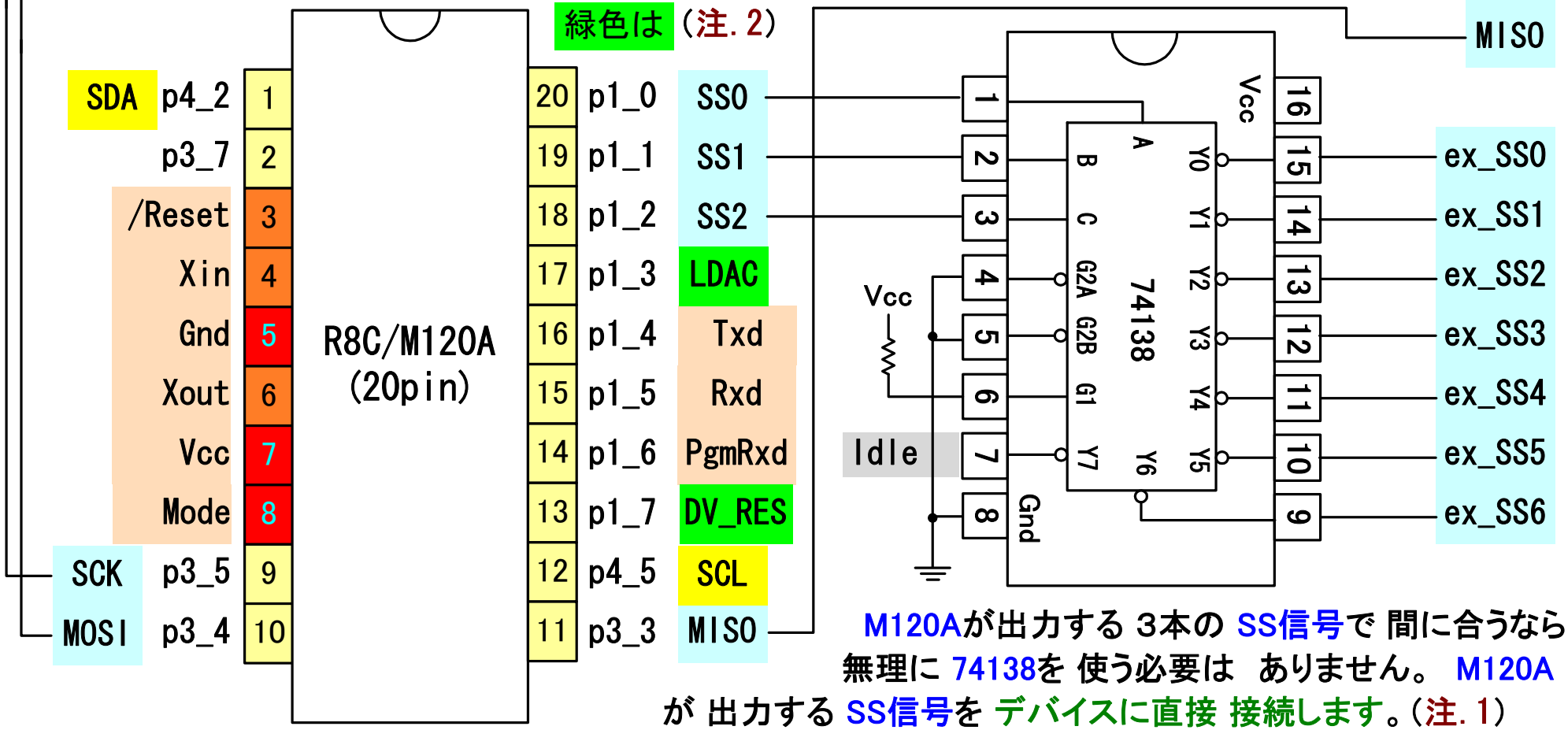
M110に 138を接続する場合は SS0 = A、SS1=B C端子を Gndに 接続します。

因みに、138の 出力端子は 8本あるのに、何故 7本のデバイスまでなのかというと、SS2,SS1,SS0 = 111 を どのデバイスにも接続してないアイドル状態として 使用する予定なので、Y0 ～ Y6 を SS信号として使用して、Y7には、何も 接続しないで下さい。



R8C/M120Aマイコンと 138の接続図

M120Aに 74138を接続して 3本の SS信号を 7本の ex_SS に 拡張出来た。



M120Aが出力する 3本の SS信号で 間に合うなら 無理に 74138を 使う必要は ありません。 M120A が 出力する SS信号を デバイスに直接 接続します。(注.1)

（ 注.1 ）は、 SS信号の先に 74138が 入っているか、無いかを ドライバソフトに 認識させる必要が あります。 これは、その後 ソフトを開発を 行う際に 意識する必要が あります。

ここまで、ハード接続仕様を決めれば、あとは 基板作成が 出来ると思います。

と思っていましたが、過去の「058 RX220マイコン SPIデバイス側 基板作成」の動画を確認すると、デバイスによって、個別の信号を 若干用意する必要がある事を 思い出しました。

それが 前ページの 緑色は（ 注意.2 ）です。
17pin、p1_3 に LDAC信号と、13pin、p1_7 が DV_RES（ デバイス リセット ）信号です。

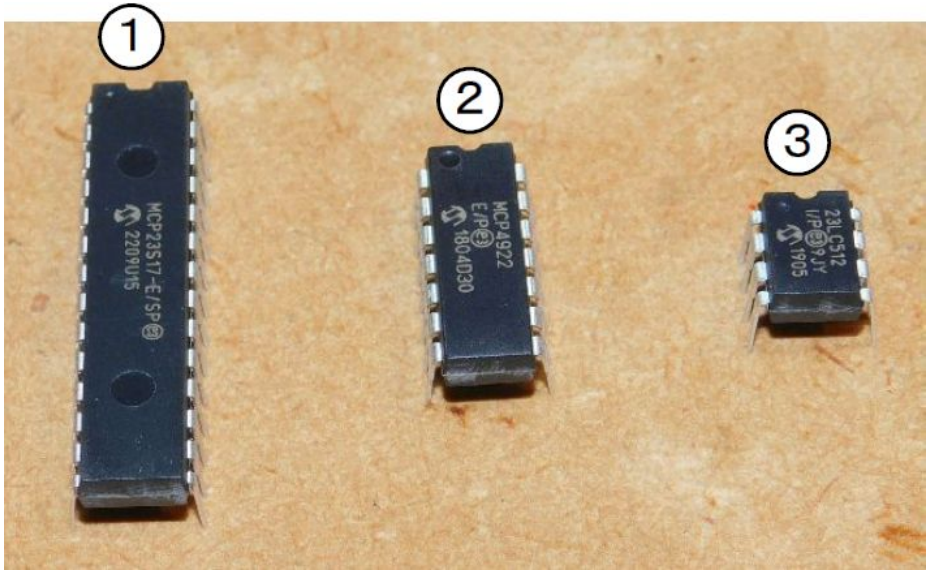
058の 3種の SPI デバイス基板には、

① 16bit I/O Expander／MCP23S17-E
② 12bit D/Aコンバータ／MCP4922
③ 512Kbit SPI Serial SRAM／23LC512

の、3つのデバイスが 実装されています。
このうち
①の MCP23S17-E 16bit I/O エクスパンダー が DV_RES信号（Low Active）を 必要とします。

②の MCP4922 12bit D/Aコンバータ にて LDAC信号（Low Active）が 必要になります。

③の 23LC512 512Kbit SPI Serial SRAM は 個別の信号は 特に必要ありません。



今回の R8C/Mマイコンの、SPIの デバイス アクセステストは、この 3つのデバイスを使用します。

特に、③の 23LC512の SRAMは、SPIのデバイスしかありません。 これを前から RAMの少ない R8C/Mマイコンに接続出来ないかと考えていました。 シリアルRAMなので 内蔵RAMの様に 自在には扱えませんが、64byte単位の補助記憶的な 使い方をすれば、結構 使えるのではないかと思います。

次に 過去の「 060 RX220 SPIアクセス ソフト編 」の一部を 引用して SCKと MOSIの 信号の オシロ波形をお見せします。 このオシロ波形を 出しているのは、RX220マイコンのSPI内蔵周辺回路です。 よって この信号は、ハードウェアにより生成されています。

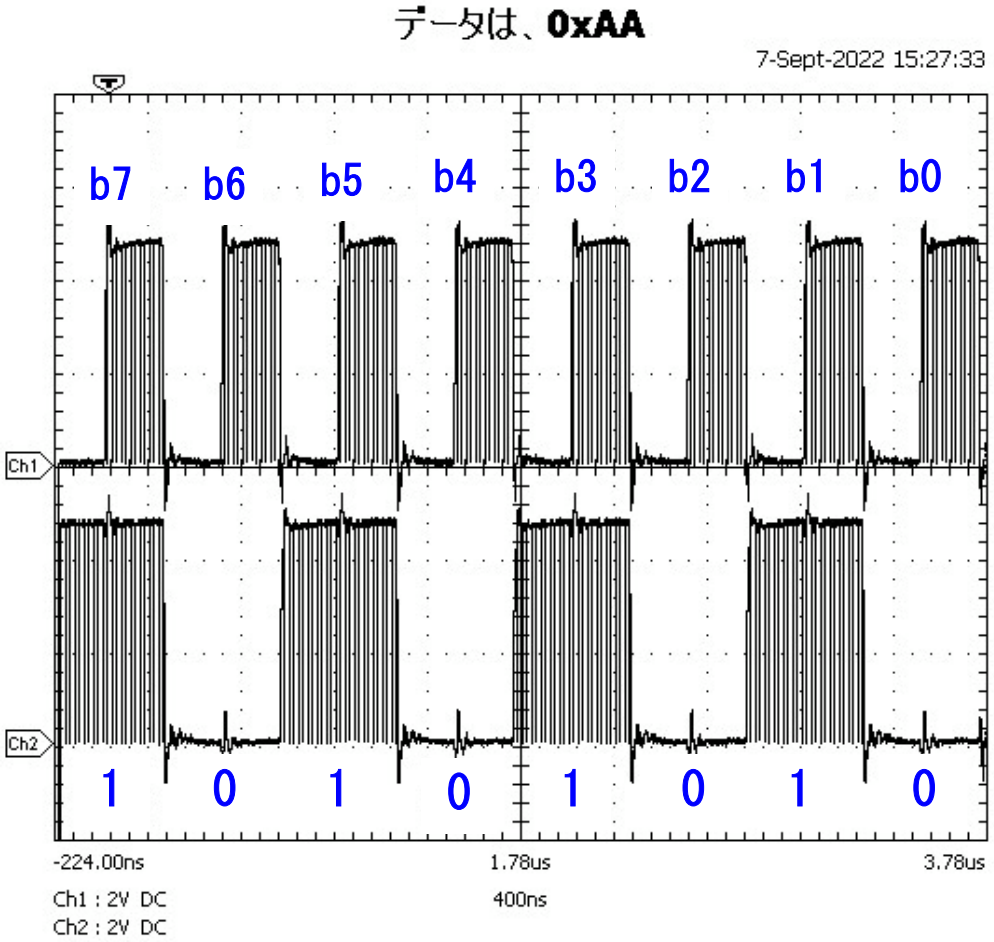
オシログラフの下に 時間軸の メモリと、経過時間の記載があったので、8 bit 転送するのに 3.78usなので、1 bit 転送するのに 0.4725us で 約 0.5us / bit になります。 よって 2Mbit/秒で転送してます。

この速さは、ソフトによる SPI では 多分 無理です。

次のページで、RX220の SPI信号グラフを お見せします。

SPI信号波形の確認　1

　右のオシログラフは、上側の信号が SCK 下側の信号が、MOSI マスター出力データです。
　データは、1バイト分のデータ転送時の波形です。 データは、0xAA を転送してます。
　データは、正論理で　左から MSB b7 です。このデータは、マスタが送信していますのでスレーブ側では、上側の SCKの 立ち上がりエッジで、下側の MOSI信号の 取り込みを行う事になってます。
　そして、SCK信号の　立ち下がりエッジでデータの信号を、次の信号に入れ替えています。
b0のデータ信号を出した後は、しばらくの間、b0のデータ信号レベルを 維持しているようです。



SPI信号波形の確認　2

　右のオシログラフは、上側の信号が SCK 下側の信号が、MOSI マスター出力データです。 このオシログラフでは 4バイトのデータを送信しています。
データは、0x00、0x03、0x0F、0x3Fの 順に送信しています。 右のグラフで、色を付けている四角は、青が 0 の領域、赤が 1 の領域です。
　SPI信号の 伝送時の波形のイメージが多少なりと掴めていただければ幸いです。