RX220マイコンプログラミング３　概要

今回の動画から、RX220で使用する I/O処理の ライブラリモジュールを作る方向で、やって行こうと思います。 今回は初期化処理で、最も基本的なクロック周波数の切り替えと、インターバルタイマのモジュールを作成しました。

以前、R8Cマイコンにおいて I/O処理のライブラリモジュールの ソースを 複数作成し IOCS (インプット、アウトプット、コントロール、システム)という事で　名前を付けていました。

今回は RX22_IOCS という事で、同様に、順次機能を作成していこうと思います。
今回は、初回なのでまだ、3つのソースファイルだけです。

① RX22_iocs.h // 関数群のプロトタイプ宣言
② RX22_iocs_init.c // クロック周波数切り替え他
③ RX22_iocs_ivl_timer.c // インターバルタイマ

どのような関数を、実装しているかは、次のページで、ヘッダファイルの内容を紹介します。

関数の関数値や、引数部分の データ型に _UBYTE、_UWORD、_UINT を 使用しておりますが、これらの型は標準で入っている typedefine.h に、宣言してあった型です。

```
//  Source File : RX22_iocs_init.c  プロトタイプ宣言
// ------------------------------------------------------------------------
void    enable_irq( void );                 // CPU I Flag 割り込みを許可する
void    disable_irq( void );                // CPU I Flag 割り込みを禁止する
_UBYTE  chk_cpu_clock( void );              // クロック設定周波数 確認用
_UBYTE  setup_main_clk_20m( void );         // メインクロック 20MHz 切り替え
_UBYTE  setup_hoco_clk_32m( void );          // HOCOクロック 32MHz 切り替え
void    setup_wdt( void );                   // ウォッチドッグタイマー起動
void    refresh_wdt( void );                 // ウォッチドッグタイマー リフレッシュ
void    soft_cpu_reset( void );              // ソフトによる CPU リセット
void    nulls( _UBYTE *ptr, int  cnt );     // メモリブロックの Null 初期化

//  Source File : RX22_iocs_ivl_timer.c  プロトタイプ宣言
// ------------------------------------------------------------------------
void  setup_interval_timer( void );         // インターバルタイマー起動
_UINT get_free_ctr( void );                 // 1msフリーランタイマの読み出し
void  set_timer_1m1( _UWORD cnt );          // 1ms単位減算タイマー１ 初期値 設定
_UWORD      get_timer_1m1( void );          // 1ms単位減算タイマー１  現在の 残り時間 読み出し
void  set_timer_1m2( _UWORD cnt );          // 1ms単位減算タイマー２ 初期値 設定
_UWORD      get_timer_1m2( void );          // 1ms単位減算タイマー２  現在の 残り時間 読み出し
void  set_timer_10m1( _UWORD cnt );         // 10ms単位減算タイマー１ 初期値 設定
_UWORD      get_timer_10m1( void );         // 10ms単位減算タイマー１  現在の 残り時間 読み出し
void  set_timer_10m2( _UWORD cnt );         // 10ms単位減算タイマー２ 初期値 設定
_UWORD      get_timer_10m2( void );         // 10ms単位減算タイマー２  現在の 残り時間 読み出し
```

割り込み処理登録に 要注意

RX220マイコン発売開始から、わりと早い時期に、RX220マイコンのプログラミング記事をブログ等に Upしておられる諸先輩方の 資料を参考にして最初、試してみましたが、何か抜けてるようで、割り込みが動きませんでした。

それと不思議に思う事として、周辺回路の割り込み許可フラグの設定は、当然必要ですがCPUの PSWの Ｉフラグは、どこで設定しているのだろうと不思議に思いました。

最初、割り込みが動かないのは、CPUの Ｉフラグが 0 のまま(割り込み禁止)なのだろうと思い、Ｉフラグを セット、リセットする機能をまず、実装しようと 考えました。
因みに、CPUの PSWのフラグを
直接C言語で 操作する事は 出来ません。

CPUのPSWフラグは、アセンブラでしか操作出来ないので、まずはインラインアセンブラはどのように記述するのか調べる必要があります。

```
//*****************************************
//**  CPU Ｉ Flag 割り込み 許可にする     **
//**  スーパーバイザモードでのみ有効      **
//*****************************************
#pragma inline_asm enable_irq  // インライン宣言
void  enable_irq( void )
{
    setpsw  I;    // CPU 割り込みフラグ ( 有効 )
}
```

インラインアセンブラの宣言は、#pragma inline_asm enable_irq の下に 通常の 関数名(enable_irq)を 付けたC言語の関数のような 器の中に 1行の アセンブラステートメント setpsw I;（Ｉフラグを １にします ）を 入れます。複数行も入れられます。

この機能を呼び出す時は、通常の関数呼び出しと同様に enable_irq(); で呼び出せます。

割り込み禁止の場合は、以下の記述になります。

```
//****************************************
//**  CPU I Flag 割り込み 禁止にする     **
//**  スーパーバイザモードでのみ有効     **
//****************************************
#pragma inline_asm disable_irq
void  disable_irq( void )
{
    clrpsw  I;    // CPU 割り込みフラグ ( 禁止 )
}
```

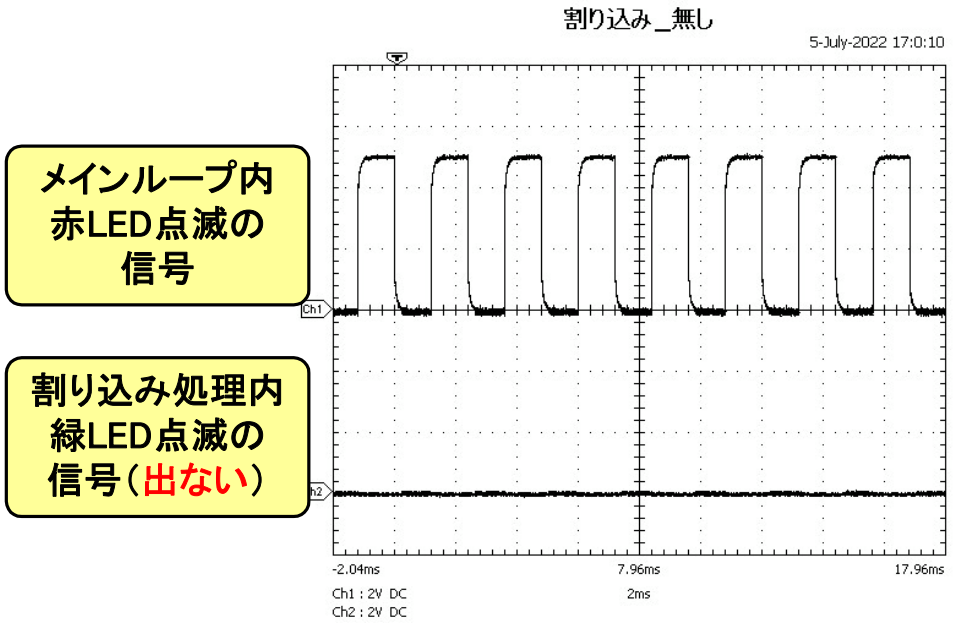
先ほどと、同様で 関数名を換えて setpsw I; を clrpsw I; に 換えるだけです。
この機能を呼び出す時は、通常の関数呼び出しと同様に disable_irq(); で呼び出せます。

この 2本の インラインアセンブラ関数は、RX22_iocs_init.c 内に実装してます。

左の関数上のコメントに スーパーバイザモードでのみ有効 と書いてますが、I フラグの操作はスーパーバイザモードでしか行えない。という事です。 RX220マイコンは起動直後は、スーパーバイザモードで、動いてます。 意図的に ユーザーモードに切り替える処理をしない限りスーパーバイザモードです。

しかし、enable_irq()を 呼び出しても割り込み処理は、走りませんでした。 今まで、私は、割り込み処理は、H8マイコンでも、R8Cマイコンでもアセンブラで記述していました。 割り込み可変ベクタテーブルが、アセンブラで記述されていた事もあり、自分にとっては、アセンブラの方が作りやすかったのです。ところが、RXになってプロジェクトのフォルダ内に、アセンブラらしき拡張子のファイルがありません。 よって C言語にて #pragma interrupt(割り込み処理の関数名)の宣言で、割り込み処理を 登録したつもりでしたが動きませんでした。

一応、どのようにして割り込み処理が、走らない事を確認したかというと、テストプログラムにてメイン関数のループ内にて、赤のLEDを 点滅させます。　割り込み処理内にて、緑のLEDを 割り込み処理に入った時で 点灯、割り込み処理から抜ける直前で 消灯していました。
オシロで 赤LEDの信号を ch.1 上側、
緑LEDの信号を ch.2 下側で観測してました。



#pragma interrupt() の 宣言箇所を示します。

```
#pragma interrupt( Int_CMT0_CMI0 )
void Int_CMT0_CMI0( void )
{
    割り込み処理内で行う処理を記述する
}
```

インターネットで 別の資料を見つけて読んでみると　プロジェクト生成時 自動的に生成される
intprg.c というＣソースファイル？が、あり
CMT0の割込み機能を使う場合は intprg.c 内の

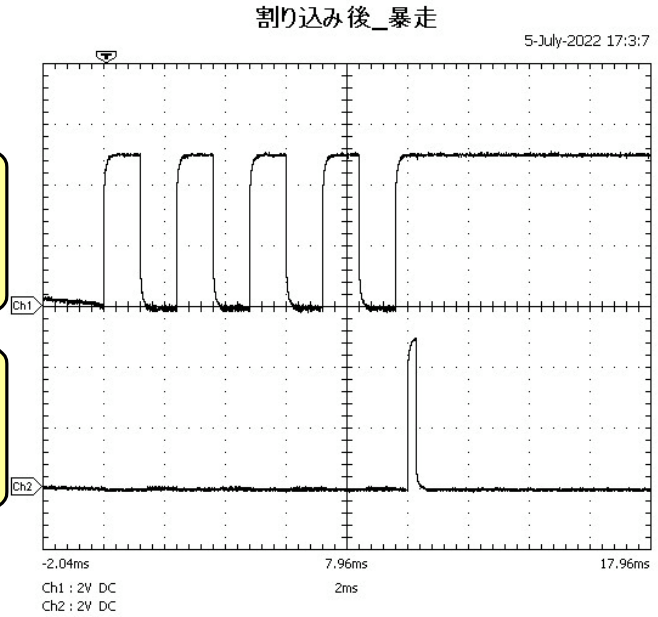
```
// CMTU0_CMT0
void Excep_CMTU0_CMT0(void){  }
```

の２行を見つけ {　}内に 関数呼び出しと同様に
割り込み処理関数を入れるとの事でした。

```
// CMTU0_CMT0
void Excep_CMTU0_CMT0(void){ Int_CMT0_CMI0(); }
```

これを行う事によりちょっと進展しました。
一回だけ、割り込みが走るのを 確認しました。

一回だけ、割り込みが走るのを 確認しました。
とは、どういう事かというと、これもオシロで確認しました。

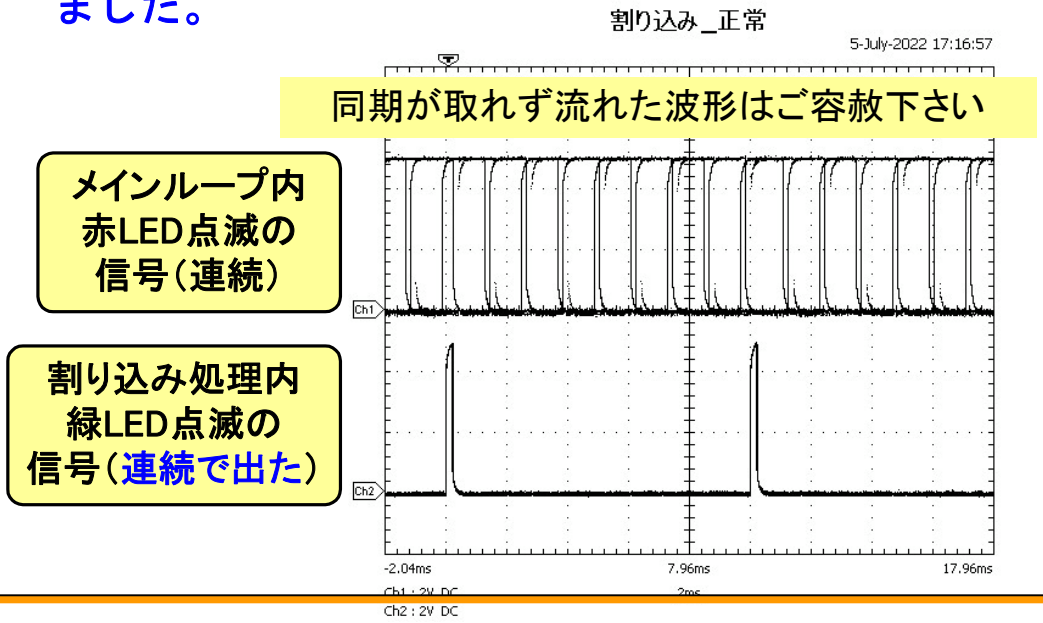


メインループ内
赤LED点滅の
信号

割り込み処理内
緑LED点滅の
信号（1回だけ）

初回、一回だけ、割り込み処理が走ってます。
その直後、メインループの点滅処理が止まってます。　これは、割り込み処理を抜けた時点でＣＰＵが、暴走した。　という症状です。

コーディングで一つ気になる箇所がありました。
intprg.c内の void Excep_CMTU0_CMT0(void){ }
関数は、これこそが 割り込み処理のエントリ関数で、この中で呼び出す関数は、通常の Ｃ関数でなければならないのではと、考えました。
であれば、#pragma interrupt(Int_CMT0_CMI0)は、必要無い。というか、むしろ 害があります。
試しに #pragma interrupt(Int_CMT0_CMI0)をコメント化してビルド、実行しました。動き出しました。



メインループ内
赤LED点滅の
信号（連続）

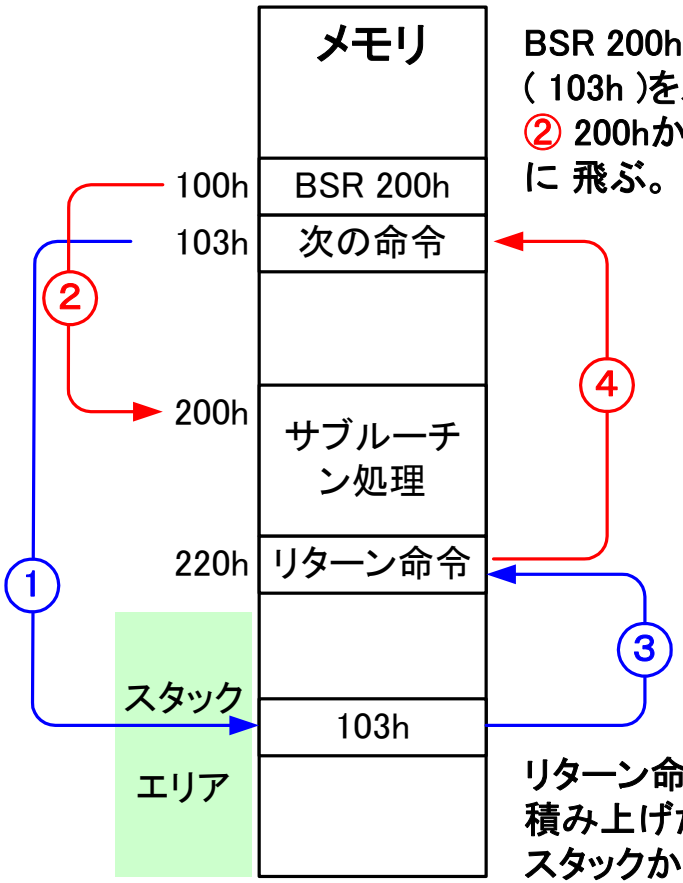
割り込み処理内
緑LED点滅の
信号（連続で出た）

では、割り込み処理エントリ関数と、通常のC関数は、何が違うのでしょうか。？

これは、メイン関数の中で、サブ関数を呼び出すとき CPUの命令レベルで何を 行っているかですが分岐命令というのがあって、BRA ブランチ命令とBSR ブランチサブルーチン命令があります。

まず、この 2つの命令の違いを理解する必要があります。 ブランチ命令は、別の所に飛ぶ事はできますが、戻る事が出来ません。 ブランチサブルーチン命令は、用が済んだら 呼び出した箇所の次の命令位置に戻る事が出来ます。 この戻る事を実現するため、戻り番地の値を スタックエリアに 積み上げてから、ブランチするのが、ブランチサブルーチン命令です。 スタックエリアの何番地に積み上げたかを管理しているのが、スタックポインタです。

サブルーチン側の最後に リターン命令があります。リターン命令は、スタックに積み上げた戻り番地を降ろし、PCに設定します。これにより呼び出し側の次の命令を読み出し実行します。



BSR 200h ① 次の命令の番地(103h)をスタックに積み上げる。
② 200hから始まるサブルーチンに 飛ぶ。

リターン命令に来たら、スタックに積み上げた 戻り番地(103h)をスタックから降ろし、③ PC(プログラムカウンタ)に、設定する。
PC = 103h に なっているので次の命令を取り込もうとすると④ 103Hの命令を取り込む事になる

では、割り込み処理エントリ関数と、通常のC関数は、何が違うのでしょうか。？　の答えが まだ出てないですよね。 実は、先ほどのページの説明で サブルーチンの 最後には リターン命令がありましたが このリターン命令が、複数あるのです。

　通常のC関数のリターン命令は RTS リターン　フロム　サブルーチンです。 で、割り込み処理エントリ関数の最後にあるリターン命令は、RTE リターン フロム イグゼプション なのです。

RTS命令は、戻り番地として 4byte スタックから値を降ろします。 それを、PCに設定します。

RTE命令は、まず 戻り番地として 4byte　スタックから値を降ろします。次に スタックから PSWの保存値として 値を降ろします。 計 8byte スタックから降ろします。

　ここで、また　intprg.c 内の

void Excep_CMTU0_CMT0(void){ Int_CMT0_CMI0(); }

の　記述ですが、Excep_CMTU0_CMT0() は 割り込み処理関数で　最後の命令は RTE のはずです。

Excep_CMTU0_CMT0() 関数内で 呼び出している Int_CMT0_CMI0() 関数は 通常の関数のはずです。

それを、#pragma interrupt(Int_CMT0_CMI0)で、割り込み処理関数として宣言していると 最後のリターン命令が RTE になるので、スタック上の データを 4byte余分に 降ろしてしまうため、呼び出した Excep_CMTU0_CMT0() の 戻りアドレスが　スタックから消滅しているので、CPUが、暴走する訳です。

実際に、#pragma interrupt を 関数に付けた場合と付けない場合とで、Cコンパイラで リスティングファイルを出力して、内容を確認したら、やはり　#pragma interrupt を 付けた場合が、最後のリターン命令が RTE で、#pragma interrupt を 付けない場合の リターン命令が　RTS である事を 確認しました。

リスティングファイルは、通常 出ないので、Hewの メインメニューの ビルド - RX Standard tool chainの コンパイラ タブの カテゴリ:リスト で リスト出力で チェックを入れて下さい。

#pragma interrupt
をコメント化してコンパイル
したリストファイルです。
茶色の文字列は、元のＣソース
です。　RTSは　サブルーチン
からの復帰時のに使用するリ
ターン命令で、PC(プログラム
カウンタ)だけを 復帰します。

```
00000056 FB4E11C008                ;      54 //*****************************************
0000005B F14120                    ;      55 //**  ★  CMT0 定周期タイマー割り込処理     **
                                   ;      56 //*****************************************
0000005E FB5AD007                  ;      57 //#pragma interrupt( Int_CMT0_CMI0 )
00000062                           ;      58
00000062 6015                      ;      59 voidInt_CMT0_CMI0( void )
00000064 21rr                                   .glb        _Int_CMT0_CMI0
00000066                           ;      60 {
                                   ;      61     int   t;
00000066 F14920                    ;      62
00000069 02                        ;      63     PORTH.PODR.BIT.B1 = 1;        // 緑 点灯
                                                MOV.L       #0008C011H,R4
                                                BSET        #01H,20H[R4]
                                   ;      64     for( t=0; t<2000; t++ );
                                                MOV.L       #000007D0H,R5
                                   L12:
                                                SUB         #01H,R5
                                                BNE         L12
                                   L13:
                                   ;      65     PORTH.PODR.BIT.B1 = 0;        // 緑 消灯
                                                BCLR        #01H,20H[R4]
                                                RTS
                                   ;      66 }
```

#Pragma interrupt
を有効化してコンパイルしたリストファイルです。
茶色の文字列は、元のＣソースです。

```
00000056 6E45                                 PUSHM      R4-R5
00000058                            L11:
                                    ;   60 {
                                    ;   61     int   t;
                                    ;   62
                                    ;   63     PORTH.PODR.BIT.B1 = 1;      // 緑 点灯
                                          MOV.L      #0008C011H,R4
                                          BSET       #01H,20H[R4]
                                    ;   64     for( t=0; t<2000; t++ );
                                          MOV.L      #000007D0H,R5
00000064                            L12:
00000064 6015                             SUB        #01H,R5
00000066 21rr                             BNE        L12
00000068                            L13:
                                    ;   65     PORTH.PODR.BIT.B1 = 0;      // 緑 消灯
00000068 F14920                           BCLR       #01H,20H[R4]
0000006B 6F45                             POPM       R4-R5
0000006D 7F95                             RTE
                                    ;   66 }
```

Source header of the listing:

```
54 //****************************************
55 //**  ★  CMT0 定周期タイマー割り込み処理   **
56 //****************************************
57 #pragma interrupt( Int_CMT0_CMI0 )
59 void Int_CMT0_CMI0( void )
      .glb        _Int_CMT0_CMI0
      .STACK      _Int_CMT0_CMI0=16
```

#pragma interruptを指定した事で、最後のリターン命令は RTE が 生成されています。
RTE は、割り込み処理からの復帰時に 使用するリターン命令で、PCと PSWの２つを 復帰します。

この関数は 内部で使用されている R4 と R5 レジスタを入口側で PUSHM R4-R5で、スタックに退避して、出口側で、POPM R4-R5 で 復帰させています。

よって、この関数は 必要最低限のレジスタしか、退避、復帰して無いので、割り込み処理に使用すれば応答は速いと思います。

intprog.c の
リストファイルの一部です

```
                                63 // CMTU0_CMT0
                                64 void Excep_CMTU0_CMT0(void){ Int_CMT0_CMI0(); }
                                        .glb    _Excep_CMTU0_CMT0
00000016                    _Excep_CMTU0_CMT0:                  ; function: Excep_CMTU0_CMT0
                                        .STACK  _Excep_CMTU0_CMT0=36
                                        .RVECTOR 28,_Excep_CMTU0_CMT0
00000016 6E15                           PUSHM   R1-R5    // R1～R15 レジスタまで 使用可能な
00000018 6EEF                           PUSHM   R14-R15 // レジスタ全てを退避してます。
0000001A                    L20:
0000001A 05rrrrrr           A           BSR     _Int_CMT0_CMI0  // 普通のサブルーチン
                                                                // 呼び出しです

0000001E 6FEF                           POPM    R14-R15 // スタックに積み上げた R1～R15までの
00000020 6F15                           POPM    R1-R5   // 全レジスタを 復帰してます。
00000022 7F95                           RTE             // 割り込み処理のリターン命令です
```

タイマーCMT0 の割り込み処理登録 関数部分です。
関数内入口側で、15本の 32bit レジスタ値を スタックに積み上げ 退避しています。 関数内出口側で、退避していた 15本の32bitレジスタ値を スタックから 降ろして、レジスタ値を元に戻しています。 無難では、ありますが レジスタ退避、復帰の処理に、時間的オーバーヘッドは、やや あるでしょうね。

それと 最初参考にしたソースに 初期化時に CPU の I フラグを 割り込み許可状態にする命令が、無い事を書きました。　その後調べてみると　main()関数が、呼び出された時点で既に　I　フラグ = 1 で　割り込みが 受付可能になっていました。

何で こうなっているの。？　初心者にとって割り込み許可フラグ I を 有効にする方法は厄介で悩むのでは、と考え 最初から 割り込み許可状態で、main() 関数を呼ぶようにしたのでしょうか。？

単純に考えると便利にも思えますが、処理内容によっては、割り込み処理に入って欲しくないタイミングが、一時的に発生する場合があります。
クリティカルパスと呼びます。 通常 この　クリティカルパスのタイミングに入る前に 割り込み禁止の手続きをして、クリティカルパスのタイミングを 抜け出したら、また割り込み許可の手続きを行う処理をします。

よって、CPUの I フラグを 操作して 割り込み禁止、割り込み許可を 行う事が出来る関数が、必要になります。

マイコンにとって、割り込み処理は 重要な機能と考えますので、今回 ページを割いて説明しました。
レジスタの退避、復帰で、やや時間的オーバーヘッドは　ありますが、 割り込み処理の登録方法は、今後 当面は intprg.c を 使う方法で行う事にします。　　今回調べた事で、
RXシリーズ　コンパイラの 割り込みに関わる舞台裏が、多少見えて来た感じがします。

後は、今回作成した RX22_iocs_init.c と RX22_iocs_ivl_timer.c に関して概要を 説明します。

今回作成した IOCS 関数に関して

まず、RX22_iocs_init.c ですが、RX220マイコンは、起動直後は、遅いLOCO 125KHzを CPUクロックとして動作してます。 これを高速のクロックに切り替える機能として、

① _UBYTE setup_main_clk_20m(void);
// メインクロック 20MHz 切り替え
外部水晶発振子 20MHz で動作します。
発振周波数精度は 高いです。

② _UBYTE setup_hoco_clk_32m(void);
// HOCOクロック 32MHz 切り替え
CPU内部の HOCO発振器(多分 RC OSC)で
発振周波数精度は 32MHz ±1% です。

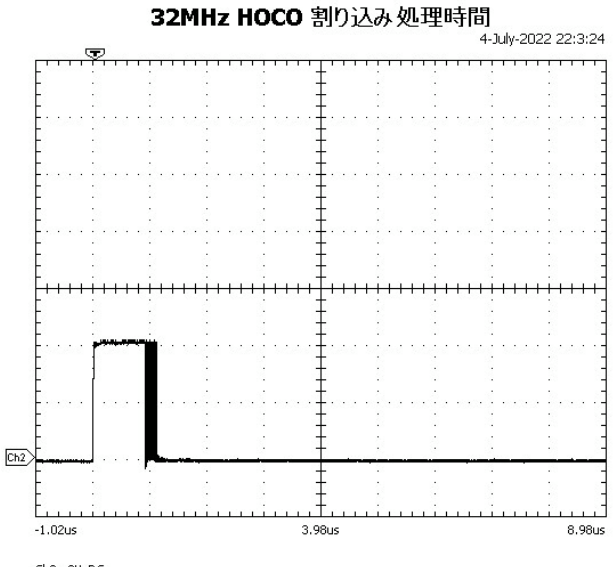
①より ②の方が クロック周波数換算で 1.6倍 早いです。 でも、時間精度を要求する用途では、水晶発振器の ①の方が有利です。

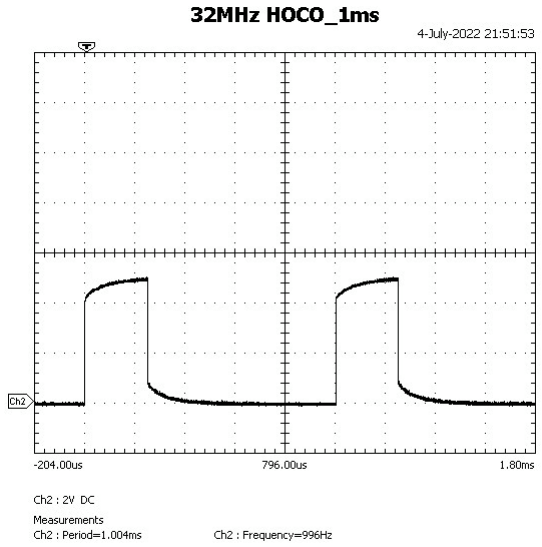
で、CPUクロックの周波数が、早い遅いは、単純に処理速度だけの問題ではなくて、例えば今回作成したインターバルタイマーの周期や、シリアル通信のボーレイト等に影響します。 で、クロック周波数の切り替えが、周辺回路に影響を及ぼさないようにする必要があります。 RX22_iocs_init.c 内に
_UBYTE chk_cpu_clock(void);
// クロック設定周波数 確認用
を、用意してます。
20MHzの場合は 関数値 = 20 で、
32MHzの場合は、関数値 = 32 です。
この関数を、各周辺回路の 初期化関数で、読み込み クロックの分周値を 調整します。

これにより、20MHzでも、32MHzでも インターバルタイマーの 時間分解能は、1ms に 出来ます。

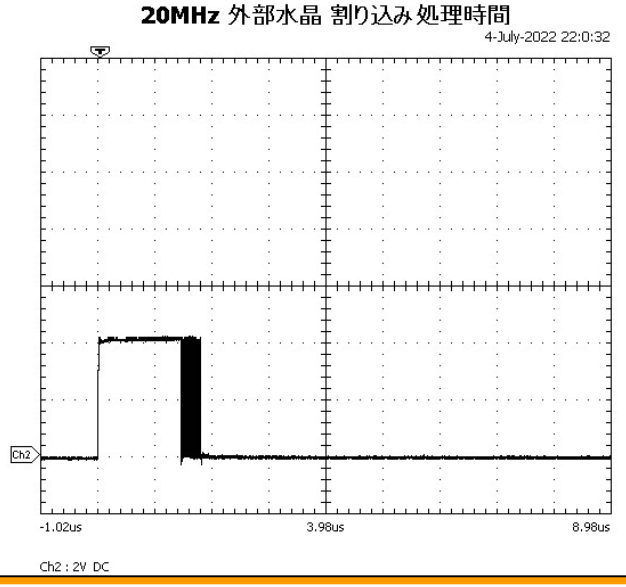
まだ作ってませんが、シリアル通信のボーレイトも、同様に クロック周波数に影響を受けないようにする予定です。

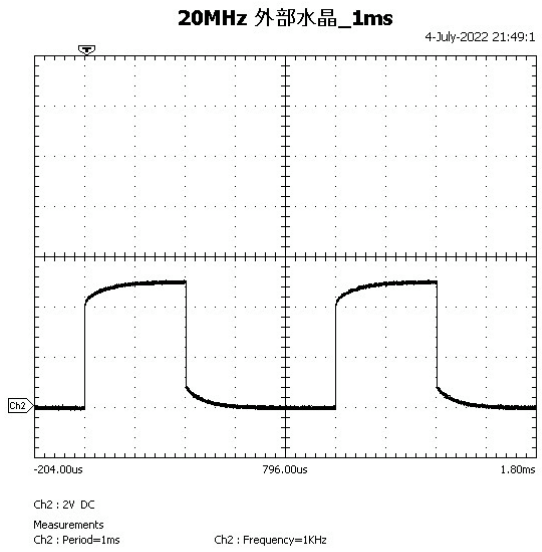
HOCO 32MHz 使用時
割り込み処理時間：Max 1.15us
タイマー割り込み
周期： 1.004ms
周波数： 996Hz





水晶発振子 20MHz 使用時
割り込み処理時間：Max 1.8us
タイマー割り込み
周期： 1.000 ms
周波数： 1.000 KHz





あと、RX22_iocs_init.c には、細かい処理の関数をいくつか作ってます。 CPUの割り込み フラグ I をセットしたり、リセットする関数。

```
void    enable_irq( void );
void    disable_irq( void );
```

ウォッチドックタイマー機能 :
これは、setup_wdt()で起動して、その後 0.8秒以内に refresh_wdt() を 継続的に呼び出し続けないと、CPUリセットが、かかる機能です。
機器の信頼性を、高める用途で使用します。

```
void    setup_wdt( void );
void    refresh_wdt( void );
```

ソフトによる CPUリセットを 行う機能です。

```
void    soft_cpu_reset( void );
```

メモリブロックを 00h で 埋め尽くす機能です。
cntは、バイト単位の メモリサイズ指定です。

```
void    nulls( _UBYTE *ptr, int  cnt );
```

RX22_iocs_ivl_timer.c 内の関数:
1ms単位 インターバルタイマー起動

```
void setup_interval_timer( void );
```

1msフリーランタイマの読み出し

```
_UINTget_free_ctr( void );
```

1ms単位減算タイマー1 初期値 設定

```
void set_timer_1m1( _UWORD cnt );
```

1ms単位減算タイマー1 現在の 残り時間 読み出し

```
_UWORD    get_timer_1m1( void );
```

同様の仕様で、1ms単位減算タイマー2と
分解能を10ms にした、10ms単位減算タイマー1と2が あります。
減算し続けて 0 になったら、減算は停止します。
多少確認処理が遅れても 減算タイマー値が 0 になった事を確認し損なう心配はありません。

インターバルタイマ機能を使った簡易テスト(1/2)

```
//************************
//**   初期化処理          **
//************************
void  init_proc( void )
{
//    CPU クロック初期化
// ---------------------------------------------------------
      setup_main_clk_20m();    ① // メインクロック 20MHz 切り替え
//    setup_hoco_clk_32m();       // HOCOクロック 32MHz 切り替え

//  I/Oポート 初期化
// ---------------------------------------------------------
      PORTH.PODR.BYTE = 0x00;  ② // ポートH 出力データ初期値 = 0
      PORTH.PDR.BYTE  = 0xFF;  ② // ポートH 各ビットの入出力指定( 1=出力 )
      setup_interval_timer();  ③ // インターバルタイマー起動

//    disable_irq();           // CPU I Flag 割り込みを禁止する
//    enable_irq();            // CPU I Flag 割り込みを許可する
}
```

① CPUクロックは、20MHzを選択。

② 秋月電子RX220ベース基板の赤と緑のLEDを点灯出来るように ポートHを 初期化する。

③ インターバルタイマーの起動

インターバルタイマ機能を使った簡易テスト(2/2)

```
void  main( void )
{
    init_proc();                          // 総合 初期化処理
    set_timer_1m1( 100 );         // タイマー1m1／初期値=100 設定
    set_timer_1m2( 99 );          // タイマー1m2／初期値= 99 設定
    while( 1 )
    {
      if( get_timer_1m1() == 0 )       // タイマー1m1が 0 になったら
      {
          set_timer_1m1( 100 ); // タイマー1m1／初期値=100 再設定
          if( PORTH.PODR.BIT.B0 == 0 )  // 赤LEDポートは 0 か？
            PORTH.PODR.BIT.B0 =1;        // 赤LED = 1
          else
            PORTH.PODR.BIT.B0 =0;        // 赤LED = 0
      }
      if( get_timer_1m2() == 0 )       // タイマー1m2が 0 になったら
      {
          set_timer_1m2( 99 );  // タイマー1m2／初期値= 99  再設定
          if( PORTH.PODR.BIT.B1 == 0 )  // 緑LEDポートは 0 か？
            PORTH.PODR.BIT.B1 =1;        // 緑LED = 1
          else
            PORTH.PODR.BIT.B1 =0;        // 緑LED = 0
      }
    }
}
```

この、テストプログラムでは、
タイマー1m1の周期を 100ms に設定し
タイマー1m2の周期を 99ms に設定しています。 これにより
赤LEDは 点灯(100ms)->消灯(100ms)を繰り返します。
緑LEDは 点灯(99ms)->消灯(99ms)を繰り返します。
赤LEDに 比べ緑LEDは、僅かに周期が短いため、オシロで観測すると 赤LED信号に比べ、緑LED信号の位相が、進み続けます。