本題に入る前に

実は、10日前ぐらいから一つ悩んでいた事がありました。 それは、割り込み処理をベクトルテーブルに登録するやり方です。

通常のC、C++アプリケーション開発としてHEWでプロジェクトを生成すると、自動生成されるソースにて、割り込み処理の登録に関わるソースは intprg.c しかありません。このソースは Cで出来ており、ベクトルテーブルの並びで、中身が空の ダミーの割り込み処理関数の集合体です。 ちょっと見は ベクトルテーブルと間違いそうですが、ベクトルテーブルでは ありません。 この intprg.cの 目的の割り込みベクトルに対応する 割り込み処理ルーチンの関数内に C言語で 割り込み処理を実装する事になります。 文章ばかりだと イメージが掴みにくいので 例を示します。

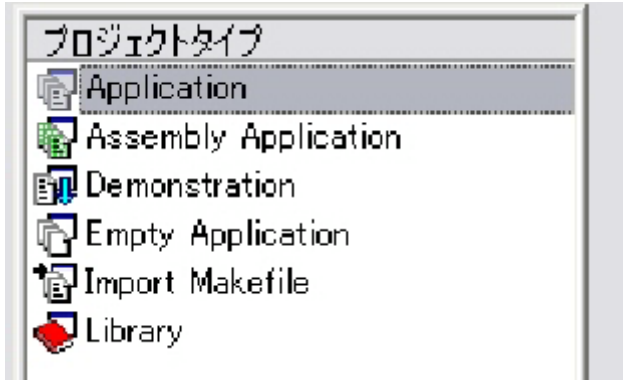
vector 23 ADI から vector 25 IMIB0の 割り込み処理のダミー関数です。 vector 24 IMIA0 に 割り込み処理を実装します。

```
//  vector 23 ADI
_interrupt(vect=23) void INT_ADI(void) {/* sleep(); */}
//  vector 24 IMIA0
_interrupt(vect=24) void INT_IMIA0(void) {/* sleep(); */}
//  vector 25 IMIB0
_interrupt(vect=25) void INT_IMIB0(void) {/* sleep(); */}
```

INT_MIA0 関数の 赤丸内の 緑の /* sleep(); */ を 書き変えます。

```
//  vector 24 IMIA0
_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--;
}
//  vector 25 IMIB0
_interrupt(vect=25) void INT_IMIB0(void) {/* sleep(); */}
```

で、前回、C言語で　メイン処理を作成して、アセンブラで、割り込み処理を　行うと言ってましたが、HEWで、新規プロジェクト作成時に表示されるプロジェクトタイプは　一番上の Application がデフォルトです。



通常、このデフォルト状態で　C、C++で プログラムを 作成して行きます。 で、一部アセンブラのサブルーチン関数を混ぜて開発する事も可能です。ところが、前ページの intprg.c 内の 割り込み関数スケルトン内に Cで　割り込み処理を記述する事は可能ですがアセンブラで　書き込む事は出来ません。

　逆に　新規作成時プロジェクトタイプで　上から2番目の　Assembly Application では　メインも含め、全てアセンブラで記述する事になっており、割り込み処理も　アセンブラで、記述出来ます。 という事で、前回やむなく この方法で タイマー割り込み処理プログラムを 作成しました。

　が、今となっては、アセンブラで全て記述する事は、まず やらない選択と 思います。

　メインをC言語で作って　割り込み処理部分をアセンブラで記述出来ないか、このHEWの H8環境では出来ないと判断しました。 何故かというと、HEWの H8環境で 通常のアプリケーション作成では、ベクターテーブルが　見当たらないのです。だいぶ探しましたが見つかりませんでした。

　実行する時は絶対必要なので　意図的に隠してる　としか思えません。 という事で　メインをC言語で　割り込み処理を アセンブラでやるのは断念しました。 割り込み処理は、前ページのやり方で　C言語で　作成します。

という事で、割り込み処理は　C言語で 行う事に した事もあり、気が楽になりました。

それと、アセンブラを　Cに置き変える事により、キー入力の タイプ量も減り、ソースが 約半分に減ります。 その分見やすくなるし、初心者の方にも　分かりやすいコーディングになると思います。

因みにアセンブラで作成した 16bitタイマー0の 初期化処理を　C言語に 作り直しました。
参考のため お見せします。

```
        .global  _init_1ms_timer
_init_1ms_timer:
        bclr     #0, @h8_t16_tstr     ; TSTR = 0  カウンタを 一旦停止させる
        mov.w    #0, r0
        mov.w    r0, @h8_t16_cnt0h    ; TCNT0 = 0  カウンタを ゼロクリア
        mov.b    #H'23, r0l           ; tcr_0 = 0010 0011
        mov.b    r0l, @h8_t16_tcr0    ; GRAコンペアマッチ、P_Edge、φ/8
        mov.w    #3125, r0
        mov.w    r0, @h8_t16_gra0h    ; GRA0 = 3125  1ms 分周値設定
        bset     #4, @h8_t16_tisra    ; IMIEA0 = 1  GRA0によるコンペア
                                      ;  マッチ割り込みを出す
         bset     #0, @h8_t16_tstr    ; TSTR = 1  カウンタを 走らせる
        rts                           ; リターン ツゥ サブルーチ ン
```

```
void    init_1ms_timer( void )
{
    ITU.TSTR.BIT.STR0 = 0;   // TSTR = 0  カウンタを 一旦停止
    ITU0.TCNT = 0;           // TCNT0 = 0  カウンタを ゼロクリア
    ITU0.TCR.BYTE = 0x23;    // GRAコンペアマッチ、P_Edge、φ/8
    ITU0.GRA = 3125;         // GRA0 = 3125       1ms 分周値設定
    ITU.TISRA.BIT.IMIEA0 = 1;  // IMIEA0 = 1  GRA0によるコンペアマッチ
                               //   で 割り込みを出す
    ITU.TSTR.BIT.STR0 = 1;   // TSTR = 1  カウンタを 走らせる
}
```

やっと本題に

インターバルタイマーに限った事では ありませんが、C言語から使いやすい(と思われる) I/O処理サブルーチンを 今後 少しずつ用意して行きます。

今、さしあたり用意した I/O処理関数の ヘッダーファイルを お見せします。凡そ、過去に作ったR8Cマイコン用の IOCS ルーチンに 似た物になります。

まずは、右側のプロトタイプ宣言を読み上げます。

文字が小さくて申し訳ありません。

```
//    CPU周辺  基本機能
// ------------------------------------------------------
int  get_cpu_mode( void ); // CPU Mode取り出し(有効 最下位 3bit)
void  enable_interrupt( void );   // 割り込み許可
void  disable_interrupt( void );  // 割り込み禁止
void  setup_wdt( void );     // ウォッチドッグタイマ有効化
void  stop_wdt( void );      // ウォッチドッグタイマ停止
void  refresh_wdt( void );   // ウォッチドッグタイマ リフレッシュ
int   check_reset( void );   // CPU リセット要因 確認
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
// ------------------------------------------------------
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
_UWORD  get_timer_1m4( void );   // 1msタイマー４現在値読み出し
```

CPU周辺 基本機能　割り込み 許可、禁止

この分類の機能は　CPUコア周辺の　ごく基本的な機能を　サポートします。
一部、特別な命令を使用する関数もあるのでアセンブラで構成されています。

① int　get_cpu_mode(void); // CPU Mode取り出し（有効 最下位 3bit）この関数は MDCRというレジスタ値を　読み出しています。 値は下位 3bitのみ有効で、この値は 書き込みモード、実行モードを決める モード端子から読み込んだ値を そのまま出しているようで、モード設定の DIP-SWを 読み出しているという事です。
よって実行時　モード 5 なので　5 しか出てきません。あまり意味のない読み出しですね。

② void　enable_interrupt(void);　// 割り込み許可　です。

NMI以外の　全ての割り込みを許可します。
電源ON 直後は、割り込み禁止状態になってます。 通常エンドレスループに入る前に　各 I/O ポート、各周辺回路の初期化を行います。 周辺回路の中には割り込み出力を出す物もあります。それらを含め一通り 初期化処理が終われば enable_interrupt 関数で 割り込みを許可します。
割り込みを許可したら、メインのエンドレスループに 入ります。

③ void　disable_interrupt(void);　// 割り込み禁止　が、あります。 これは、排他制御的な用途で、処理の途中で割り込んで欲しくない時に、一時的（ 極一瞬 ） disable_interrupt 関数で　割り込みを禁止して　フラグや カウンタの更新を行い、速やかに enable_interrupt 関数で　割り込み受付を 再開します。 NMI は 禁止出来ません。

ウォッチドッグタイマー機能

ウォッチドッグタイマーとは、制御用のソフトウェアが 正常に動作しているか確認するための検証機能です。ウォッチドッグタイマーは無くても、本来のプログラム機能を 実行する事は可能です。 むしろ、デバッグ中は 一時的に止めたりすると CPUリセットが かかり煩わしいので外しておいた方が いいです。

よって、ウォッチドッグタイマーを 実装する場合は プログラムが完全に出来上がってからウォッチドッグタイマーを 実装する 事になります。 業務で使用するシステムで 信頼性を高める必要がある場合に使用します。

④ void setup_wdt(void); // ウォッチドッグタイマ有効化

⑤ void stop_wdt(void); // ウォッチドッグタイマ停止

⑥ void refresh_wdt(void); // ウォッチドッグタイマ リフレッシュ

3本の関数がありますが、stop_wdt関数は 一度も使った事がありません。 ウォッチドッグタイマを 途中で 一時的に止めて、また再開するような使い方は まず しません。

そもそも、ウォッチドッグタイマーとは 何なのかというと、setup_wdt 関数を呼び出すと ウォッチドッグタイマーは カウント動作を始めます。

その後、そのまま放置すると 約 40ms の 時間経過で いきなりCPU リセットが かかります。

よって setup_wdt 関数を呼び出し後、40ms以内の周期で 継続的に refresh_wdt 関数を 呼び出し続ける必要が あります。 よってメインループ処理のあちこちに refresh_wdt 関数を 配置する必要が あります。 割り込み処理の中には 入れてはいけません。

CPUリセットと CPUスリープ処理

⑦ int check_reset(void); // CPU リセット要因 確認

⑧ void soft_reset(void); // ソフトによる CPU リセット

⑨ void sleep_func(void); // CPU スリープに移行させる

check_reset 関数は CPUリセットが かかった要因を 関数値で 返します。関数値 = 0 で あれば、パワーONリセット、リセットスイッチによるリセットで、関数値 = 0x80 で あれば wdtの タイムアップによるリセットです。 よって 起動時 ウォッチドッグタイマーによる リセットが かかったのか 確認出来ます。

soft_reset 関数は、ソフトウェアによる CPU リセット処理です。
割り込みを 禁止して 0 番地の CPUリセットの ベクトルアドレスを 読み出して、その番地に Jump します。 その後、check_reset 関数で ステータスを 読み出すと 0 です。

sleep_func 関数は CPUを スリープモードに 移行します。 スリープモードは CPUの レジスタ、及び RAMは 状態を そのまま保持して CPUが停止した状態になる事です。

電源の OFF、ON リセット、割り込み受付けによって、ウェイクアップするようです。 が、私はまだ、スリープモードの動作確認を やった事が ありません。 近いうちやっておきます。

1ms　timer処理

1ms taimer処理とは　1ms分解能のタイマー処理という事です。

⑩　void　init_1ms_timer(void);　// 1ms インターバルタイマー初期化

⑪　_UWORD get_counter_1m(void);　// 1msフリーランカウンタ読み出し

⑫　void　set_timer_1m1(_UWORD n); // 1msタイマー1初期値設定

⑬　_UWORD　get_timer_1m1(void);　// 1msタイマー 1 現在値読み出し

⑭　void　set_timer_1m2(_UWORD n); // 1msタイマー2初期値設定

⑮　_UWORD　get_timer_1m2(void);　// 1msタイマー2現在値読み出し

⑯　void　set_timer_1m3(_UWORD n); // 1msタイマー3初期値設定

⑰　_UWORD　get_timer_1m3(void);　// 1msタイマー3現在値読み出し

⑱　void　set_timer_1m4(_UWORD n); // 1msタイマー4初期値設定

⑲　_UWORD　get_timer_1m4(void);　// 1msタイマー4現在値読み出し

まず、init_1ms_timer 関数が　インターバルタイマーの初期化処理で　最初に呼び出します。

次の、get_counter_1m 関数は、init_1ms_timer 関数が、呼び出されたタイミングで　1ms周期でインクリメントが　継続的に　行われます。
符号なし単精度整数なので　65,535 まで　カウントすると　次は ゼロに戻り カウントは 続けます。

後、⑫から ⑲まで 関数が ありますが、これは 2つの関数で一組の インターバルタイマー処理が 4チャネルあります。
4つのチャネルは 全く同じ仕様です。

チャネル1が、⑫ set_timer_1m1関数と ⑬ get_timer_1m1関数
チャネル2が、⑭ set_timer_1m2関数と ⑮ get_timer_1m2関数
チャネル3が、⑯ set_timer_1m3関数と ⑰ get_timer_1m3関数
チャネル4が、⑱ set_timer_1m4関数と ⑲ get_timer_1m4関数

4つのチャネルは、全く同じ仕様なので set_timer_1m1関数と get_timer_1m1関数で、動作を説明します。 通常この2つの関数でアクセスするカウンタ変数は 0 で、0 の場合、何もしません。0 を 維持してます。 set_timer_1m1(1000); を 行うと、カウンタ変数に 1000が 入ります。 そして 1msのタイマー割り込みで、1ms毎に デクリメント(-1)されます。 そして 1秒経過後カウンタ変数は 0 に なります。
そして、get_timer_1m1関数は、関数を呼び出した瞬間の カウンタ変数の瞬時値を 返します。

このような仕様にすると何が便利なのかというと、例えば、Arduinoの環境では Delay関数があります。 これは Delay(200); と設定すると 0.2秒経過後に 元の呼び出し側に CPUの制御権が戻ってきます。 単に時間待ちだけであれば、これでいいのですが 0.2秒の待ち時間の間に別の仕事をしたい。という場合は 0.2秒 経過しないと Delay関数から抜け出して来ないので出来ません。
よって、最初に監視時間を設定し、そして 1/1000秒単位で 瞬時値の 経過時間を確認できればその間に 別の小さな処理が出来ます。 という事です。

用途によっては、シリアル通信を行う上で 通信制御を行う場合が、あります。 一まとまりのデータ転送の監視時間が 1分で、その中で 1ブロックの 電文を送信して 相手側から、ACK、NAKを 待ち受ける場合、数秒の時間監視を行う事になると思います。

今回は、もっと簡単な例で、マルチで時間監視出来るプログラムを 作ってみます。 今回のH8/3069F基板には LEDを 3個実装しているので、3個のLEDを バラバラの周期で パラって点灯させてみます。

一応 各LEDの点滅周期は
赤LED 0.5 秒 周期
青LED 0.3 秒 周期
黄LED 0.75 秒 周期で
点滅を 繰り返します。
右のソースは 今回の C_test_4.c です。

```
init_1ms_timer(); // インターバルタイマー初期化
enable_interrupt();  // 割り込み許可
while( 1 )           // 無限ループ
{
    if( get_timer_1m1() == 0 )
    {
        set_timer_1m1( 500 );  // 赤LED 点滅設定
        if( PBDR.BIT.B5 == 0 )   PBDR.BIT.B5 = 1;
        else                     PBDR.BIT.B5 = 0;
    }
    if( get_timer_1m2() == 0 )
    {
        set_timer_1m2( 300 );  // 青LED 点滅設定
        if( PBDR.BIT.B7 == 0 )   PBDR.BIT.B7 = 1;
        else                     PBDR.BIT.B7 = 0;
    }
    if( get_timer_1m3() == 0 )
    {
        set_timer_1m3( 750 );  // 黄LED 点滅設定
        if( PBDR.BIT.B6 == 0 )   PBDR.BIT.B6 = 1;
        else                     PBDR.BIT.B6 = 0;
    }
}
```