# TSOS カーネル(RH850 版)
# 説明書(初版)

**1. 概要**

TSOS カーネル(kernel.c/.h)は簡素で軽量ながら、状態遷移図との対比性に優れたコーディングができる特徴があります。

特徴

・ROM 0.8KB, RAM 96B(10 タスク時)と軽量です。スタックも 1 本だけで済みます。
・タスクを巡回的に呼出すという簡素な制御を行っています。
・各タスクは少なくとも一定周期(例えば 5ms)で 1 回は実行権が得られます。
・遅延時間を指定してのタスク起動や、割込みからのタスク起動もできます。
・処理の重いタスクを分割処理するためのサービスマクロを用意しています。
・状態遷移図の各箱をそれぞれ一つの関数に対比させてコーディングするためのサービスマクロを用意しています。

図 1-1. タスク実行例（6 つのタスク A～F）

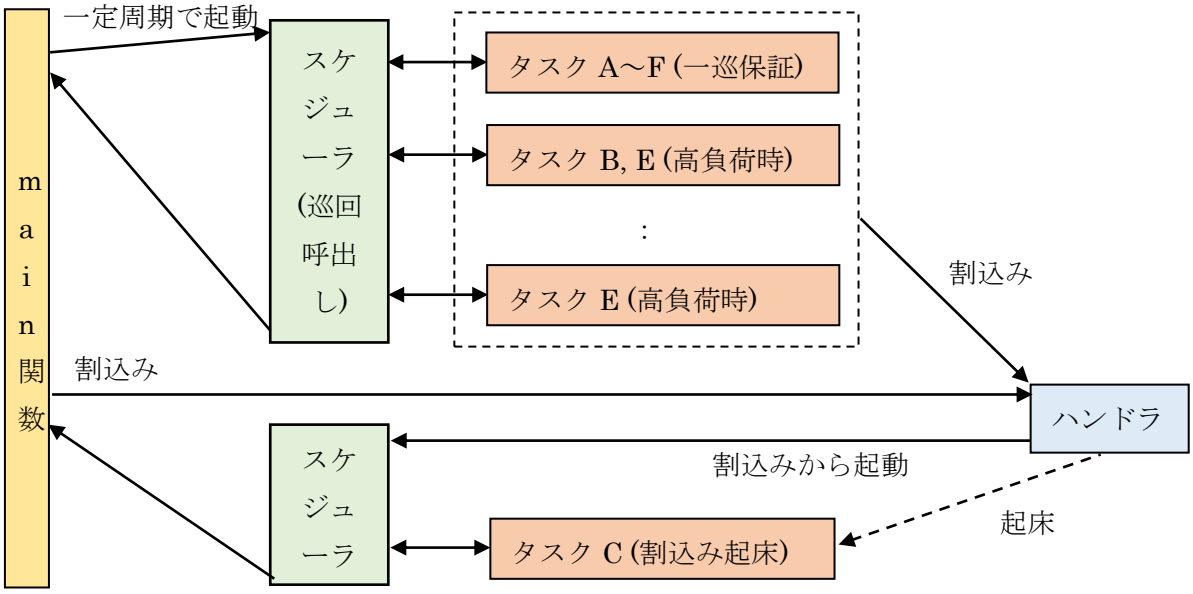


図 1-1 は 6 つのタスク A～F の実行例を示しています。

①一定周期(例えば 5ms)でスケジューラが起動され、A～F の順に状態を判定し、アクティブ状態のタスクを呼出します。つまり一定周期で一巡することを保証しています(全タスクの処理時間合計が一定周期以内の場合)。

②A～F 一巡後、処理未了のため再巡回を要請しているタスク(B,E タスク)が残っていれば再度 A～F を巡回し、それらを呼出します。ただし一定周期を超える場合は巡回を中止します。

③再巡回を要請しているタスク(E タスク)が終了すると巡回を終了します。

④一定周期でのタスク巡回が終了した後であっても割込みハンドラからスケジューラを呼出し再巡回させることができます。この例ではハンドラから起床された(アクティブにされた)タスク(C タスク)を呼出しています。

### 1.1. タスクの定義・起動、タスク記述の概要

カーネル(kernel.c/.h)をプロジェクトに組込むと、以下のようにタスク定義・起動ができます。

（1）タスク定義

ID 名を記述するだけです。

task.h の enum TASK_ID の中に名称を記述します。

記述順序が巡回順序（タスク呼出し判定の順序）となります。

（2）タスク起動

start_task(タスク ID 名, 最初に実行する関数名);

これで起動します。

タスク実行に必要な初期設定(主にポートや周辺機能の初期化)は最初に実行する関数内で行っても良いですし、start_task 実行前に行っても良いです。

main 関数実行前にタスク起動したい場合は、以下のような初期設定およびタスク起動を記述したタスク初期化関数を作り、task.c の TASK_init()の中から呼出します。

```
void タスク名_init () {
  初期設定;
  start_task(タスク ID 名, 最初に実行する関数名);
}
```

TASK_init()は main 関数実行前に呼び出される INIT_init 関数(init.c)から呼出しています。

（3）タスク記述

タスクプログラムは、一定周期より十分短い時間で一旦処理を終了する必要があります。時間のかかる処理は分割して実行するようにします。

①状態遷移の無いタスク

単純なキー入力監視のように状態遷移の必要がないタスクは一つの関数だけ記述すればよいです(他の関数と同じくプロトタイプ宣言も必要です)。

```
void タスク名_main() {       //関数名をタスク名_xx にする(推奨。必須ではない)
    キー状態読み取り;
    チャタリング除去処理;
    delay_task(10);
}
```

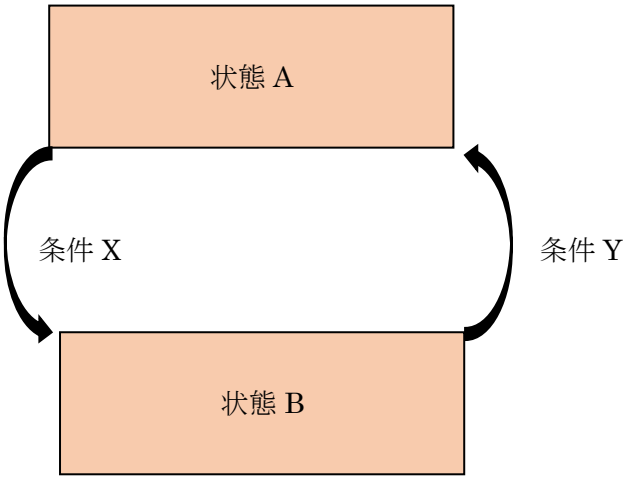
ここで delay_task(10)は、10ms 後にまた呼出してね、という意味になります。時間は ms 単位で指定できますが、実際に呼出されるのは一定周期の整数倍後です。例えば delay_task(6); と記述しても一定周期が 5ms なら実際には 10ms 後の呼出しとなります。どうしても 6ms 後に呼出して欲しい場合は一定周期を 2ms または 1ms にするか、もしくは後述の即時再呼出し記述と時間取得関数の組合わせを使用します。

delay_task()の代わりに return;を記述した場合や記述自体を省略した場合は、一定周期で呼出されるとともに他の要因でスケジューラが再巡回した時にも呼出されることになります。呼出される周期が特に気にならない(フラグチェックしてフラグが立っていたら処理するような)用途ではそのような記述でも構いません。

②状態遷移のあるタスク

状態遷移図の箱一つに対して一つのタスク関数を記述します。

以下、2 種類の状態 A,B の例を示します。



```
void タスク名_状態 A() {
    状態 A に関する制御処理;
    if (条件 X) next_task(タスク名_状態 B); //状態遷移判定
}

void タスク名_状態 B() {
    状態 B に関する制御処理;
    if (条件 Y) next_task(タスク名_状態 A); //状態遷移判定
}
```

ここで、next_task(遷移先関数名)は、次の巡回で遷移先関数を呼出してね、という意味になります。指定時間後に呼出しを希望する場合は、set_delay_task(ms 時間); の記述後に next_task を記述します。即時遷移を希望する場合は set_retry_task(); の記述後に next_task を記述します。

なお即時と言っても一巡保証のタスクが残っている場合はその後になります。また他に即時再起動や即時遷移を希望しているタスクがあれば、巡回順にそれらを再呼出しするため必ずしも短い時間が保証されているわけでは無いです。一定周期より短い時間内で呼出してもらえるという程度です。

③重負荷タスク

一定周期に対してかなりの処理時間を占めたり、一定周期内では処理が完結しない場合は処理分割が必要になります。特に各タスクの一巡保証する上では、分割の一巡目はなるべく短時間で終わるのが望ましいです。

単純に分割する場合は、以下のように記述します。

```
void タスク名_状態 1() {
    1 巡目処理;
    break_retry_task(タスク名_状態 2);
    2 巡目処理;
    break_retry_task(タスク名_状態 3);
    3 巡目処理;
    next_task(タスク名_状態 1);   //最後に遷移が必要
}
```

関数が一つだけのように見えますが、実際には void タスク名_状態 2()、void タスク名_状態 3()の関数も生成されるため、プロトタイプ宣言は 3 つとも必要です。

また break_retry_task()記述の前後で自動変数の値は引継げないため、関数外で静的変数を定義して使用します。例えば、ループ処理の場合は、以下のように記述します。（100 回ループの例)。

```
int loop_cnt;                    //ループ変数定義
void タスク名_状態 1() {
    1 巡目処理;
    loop_cnt = 0;                //ループ変数初期化
    break_retry_task(タスク名_状態 2);
    while (loop_cnt < 100) {
        ループ処理;
        loop_cnt ++;
        retry_task();
    }
    next_task(タスク名_状態 1);   //最後に遷移が必要
}
```

④割込みハンドラからのタスク起床

```
wakeup_task(タスク ID 名);
```

これをハンドラ内に記述すれば指定したタスクを起床できます。スケジューラが起動していなければ起動し、巡回呼出しします。指定したタスクは retry_task()で終了したのと同じ扱いになるので必ず呼出します。なお当該タスクの最後で delay_task(), sleep_task()を実行するのは禁止です。それらの実行タイミングの直前で割込みがかかった場合に wakeup されたものがキャンセルされてしまいます。従って wakeup するタスクの最後は return;(省略可)か next_task()または retry_task()で終了するタスクに限ります。

また wakeup_task()は、sleep_task()あるいは suspend_task()によって寝てしまったタスクを他のタスクまたは main 関数から起床するのにも使用できます。この場合は割込み干渉が無いので wakeup するタスクの最後に制限はありません。

⑤サブタスクの呼出しおよび復帰

関数呼出し／復帰と同じようなイメージで使用できるのがサブタスクです。関数と異なり戻り先のタスク名を呼出し時に明示する必要があります。また異なるタスク ID で共有するサブタスクは定義できません。

サブタスクを使用するには、まず関数の外で管理用のスタック変数定義が必要です。

```
set_stack_task(スタック深さ);
```

スタック深さは、サブタスクの中から他のサブタスクを呼ばなければ 1 です。ネスティングする場合はその段数を加算します。

サブタスクの呼出しは次のように記述します。

　　call_task(呼出すサブタスク関数名、戻り先のタスク/サブタスク名);

呼出し前に必要に応じて set_xxx_task による設定を行います(以下同様)。

サブタスクからの戻り先を続けて記述したい場合は次のようにします。

　　break_call_task(呼出すサブタスク関数名、戻り先のタスク/サブタスク名);

この行に続けて戻り先のタスク/サブタスクの内容を記述します (③と同様の記述)。

サブタスクから戻る場合は次のように記述します。

　　return_task();

## 2. カーネル詳細

カーネル(kernel.c/.h)は以下の関数およびマクロから構成しています。

・スケジューラ関数(kernel_schedule)：タスクの周期制御、巡回呼出し
・一定周期タイマ関数(kernel_timer)：一定周期でスケジューラを呼出し
・一定周期タイマ関数 2(kernel_window_timer)：スケジュール禁止期間を設定
・カーネル初期化関数(KERNEL_init)：タイマの初期化と起動
・経過時間取得関数(KERNEL_time)：約 71 分周期の値を取得[μs 単位]
・経過時間取得関数(KERNEL_time_ll)：約 58 万年周期の値を取得[μs 単位]
・アイドリング関数(KERNEL_wait_next_period)：次の周期開始まで HALT する
・オーバーラン・タスク ID の取得関数(refer_error_task)：ID 取得とクリア
・その他サービスマクロ：起動、遅延、休止など各種タスク制御用

### 2.1. カーネルに必要な資源

ROM サイズ：1KB 未満 (現状 0.8KB 程度)
RAM サイズ：task.h に記述するタスク ID 数 x8 + 16 (10 タスクで 96 バイト)
タイマ　　：TCn(n は任意)を使用。どのタイマを使うかは device.h で指定。
割込み　　：タイマ割込み(TCn_IRQn)の他にスケジューラ用に任意の不使用割込みを 1 本使用。どの割込みを使うかは device.h で指定。

### 2.2. サービスマクロ、サービスコール

タスク制御用のサービスマクロおよびサービスコールの一覧を以下に示します。現版ではエラーチェックが無いので、タスク ID の指定を間違えないように注意が必要です。

| 形式 | パラメータ | 動作 |
|---|---|---|
| void start_task(id, funct) | id：タスク ID 名<br>funct：タスク関数名 | id で指定したタスクを実行可能な状態にします。次のスケジュール･タイミングで funct で指定した関数を実行します。 |
| void suspend_task(id) | id：タスク ID 名 | id で指定したタスクを強制的に起床待ち状態にします。 |
| void wakeup_task(id) | id：タスク ID 名 | id で指定したタスクを強制的に実行可能状態にします。 |

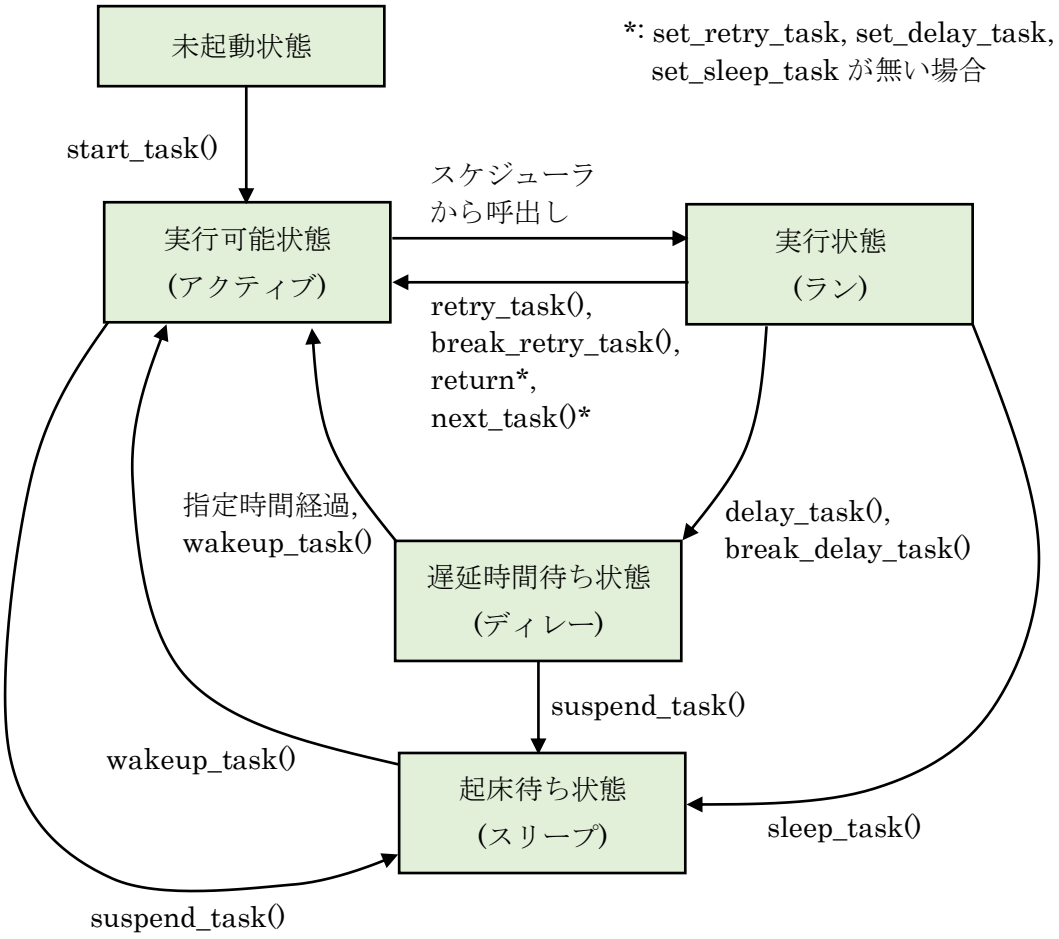
| 形式 | パラメータ | 動作 |
|---|---|---|
| int refer_task(id) | id：タスク ID 名 | id で指定したタスクの状態を取得します。戻り値は以下の通り。<br>== TASK_NONE //未起動状態<br>== TASK_WAIT //起床待ち状態<br>== TASK_ACTIVE //実行可能状態<br>> TASK_ACTIVE //遅延状態 |
| void retry_task() | - | 自タスクの即時再呼出しを設定した上で実行終了します |
| void set_retry_task() | - | 自タスクの即時再呼出しを設定します |
| void delay_task(t) | t：遅延時間 | 遅延時間後の自タスク呼出しを設定した上で実行終了します |
| void set_delay_task(t) | t：遅延時間 | 遅延時間後の自タスク呼出しを設定します |
| void sleep_task() | - | 自タスクを起床待ちに設定した上で実行終了します |
| void set_sleep_task() | - | 自タスクを起床待ちに設定します |
| return または関数終了位置の } | - | 実行終了します。set_xxx_task による設定がなければ、次の巡回時（一定周期後または他の要因でスケジューラが再巡回した時）に自タスクを呼出します |
| void next_task(funct) | funct：タスク関数名 | 次に呼出す関数名を funct に設定した上で実行終了(return と同等動作)します。 |
| void set_next_task(funct) | funct：タスク関数名 | 次に呼出す関数名を funct に設定します |
| void break_retry_task(funct) | funct：タスク関数名 | タスクの処理分割用記述です。次に呼出す関数名を funct に設定し、即時再呼出しを設定した上で実行終了します。この記述の次の行から funct の内容を記述します(1.1③参照) |

| 形式 | パラメータ | 動作 |
|---|---|---|
| void break_delay_task(funct, t) | funct：タスク関数名<br>t：遅延時間 | タスクの処理分割用記述です。次に呼出す関数名を funct に設定し、遅延時間後の呼出しを設定した上で実行終了します。この記述の次の行から funct の内容を記述します(1.1③参照) |
| void break_task(funct) | funct：タスク関数名 | タスクの処理分割用記述です。次に呼出す関数名を funct に設定し、実行終了(return と同等動作)します。この記述の次の行から funct の内容を記述します(1.1③参照) |
| void set_stack_task(depth) | depth：スタック深さ | サブタスクの管理変数定義です。深さは基本 1 で、ネスティングする場合はその段数を加算します。(1.1⑤参照) |
| void call_task(funct,<br>ret_funct) | funct：呼出すサブタスク関数名<br>ret_funct：戻り先のタスク／サブタスク関数名 | サブタスクの呼出しを設定し、実行終了(return と同等動作)します。(1.1⑤参照) |
| void break_call_task(<br>funct, ret_funct) | funct：呼出すサブタスク関数名<br>ret_funct：戻り先のタスク／サブタスク関数名 | タスクの処理分割用記述です。次に呼出すサブタスク関数名を funct に設定し、実行終了(return と同等動作)します。この記述の次の行から ret_funct の内容を記述します。サブタスクからの戻り先になります。(1.1⑤参照) |
| void return_task() | | 戻り先の関数名をスタックから取出して設定し、実行終了(return と同等動作)します。(1.1⑤参照) |

| 形式 | パラメータ | 動作 |
|---|---|---|
| int refer_error_task() | - | 一定周期のタイミングを超えて実行終了したタスクの ID 番号を取得します。複数のタスクがあった場合は最後に実行したタスクの ID となります。また取得後に記録をクリアします。無ければ EOF(-1)が戻ります。 |
| ulong KERNEL_time() | - | カーネル起動からの経過時間をμs 単位で示します(32bit 符号なし整数, 約 71 分周期)。 |
| ulonglong KERNEL_time_ll() | - | カーネル起動からの経過時間をμs 単位で示します(64bit 符号なし整数, 約 58 万年周期)。 |
| void KERNEL_wait_next_period() | - | main 関数(アイドリングタスクを兼用)の中から呼び出します。次の一定周期まで HALT(CPU 停止)します。ただし実行可能タスクがあればその終了後に HALT します。 |
| void KERNEL_init() | - | カーネル初期化関数です。一度だけ呼出します。 |

### 2.3. タスク状態遷移

サービスマクロによるタスクの状態遷移を下図に示します。




①set_retry_task(), set_delay_task(),set_sleep_task()を複数実行した場合は最後に実行したものが有効です。ただし最後に return または next_task()を実行しないとキャンセルされます。

②set_next_task()の後に break_xxx_task()を実行すると set_next_task()の設定はキャンセルされます。

③未起動状態のタスクに対して wakeup_task()を実行すると暴走します。
スタート時に起床待ち状態にする場合は以下の例のように割込み禁止にして start_task()と suspend_task()を実行します。

```
__DI();
start_task(対象タスク ID);
suspend_task(対象タスク ID);
__EI();
```

### 2.4. 一定周期

TSOS カーネルはスケジューラを一定周期で呼出しています。周期時間は device.h の TASK_PERIOD 定数で指定します(ms 単位)。1, 2, 5, 10 のいずれかを選択します。デフォルトは 5ms です。

### 2.5. タスク割当て時間

一定周期の内､タスク処理に割当てる時間を何%にするかを device.h の TASK_WINDOW 定数で指定します(1%単位)。デフォルトは 80%です。

main 関数をアイドル・タスク(タスク処理が無い時に CPU を停止するタスク)としてのみ使用する場合は 100%を指定します。

main 関数でもある程度の処理をしたい場合は、タスク割当て時間を減らします。割当て時間内に開始したタスクが時間外で終了する可能性も考慮して決めます。

### 2.6. システム時刻

一定周期用のタイマを使用してシステム時刻 (カーネル起動からの経過時間) を管理しています。KERNEL_time_ll()によって 64 ビット符号なし整数(1μs 単位 58 万年周期)の時刻を得ることができます。71 分周期で十分であれば、KERNEL_time ()によって下位の 32 ビット符号なし整数部分のみを得ることもできます。

### 2.7. main 関数の役割

main 関数はタスクを実行していない時に実行します。一定周期時間に縛られずに処理ができるので、非リアルタイム系のバックグランド処理やコマンド応答のような逐次処理に適しています。

タスクを実行していない時に CPU 動作を止めて低消費電力化を図りたい場合は次のように常に__HALT()を実行するループにします。

```
while(1) {
    __HALT();
}
```

一定周期に同期して動かしたい場合は KERNEL_wait_next_period()を使用します。

```
while (1) {
    KERNEL_wait_next_period();  //次の一定周期開始を待ちます
    同期処理;                   //タスク終了後に実行します。
}
```

ただし同期処理が長すぎて途中で一定周期を超えた場合は、そこでタスクが動作するため一周期ずれてしまいます。

## 2.8. スタック返却

タスクは一定周期より十分短い時間内に実行状態(ラン)から抜ける必要があります(抜けるために実行するサービスマクロ等は「タスク状態遷移」を参照)。実行状態から抜けるとスタックも返却されて次のタスクに使用されてしまうので、タスク内で値を引継ぐ必要がある変数は静的変数として定義します。

スタックサイズは以下の合計サイズで決めます。

・各タスクの中で一番大きなサイズを要求するタスクのスタックサイズ
・main 関数で消費するスタックサイズ
・割込みで必要なスタックサイズ(多重割込みの場合は各レベル最大値の合計)

スタックの実際の消費量を取得するには、device.c/.h で提供されている DEVICE_stack 関数を使用します。現版では最小 128 バイトから 4 バイト単位で所得できます。取得値の最大値はリンカ・オプションで指定した値となりますので、その値を取得した場合の実消費量はオプション指定の値を超えている可能性があります。

## 2.9. 割込み優先順位

スケジューラおよびタスク　　：優先コード 14、多重割込み許可
スケジューラを起動するタイマ：優先コード 12、多重割込み許可

優先コードは低いほど優先度が高くなります。従って、一般的な割込みは 14 未満を指定する必要があります。

device.h に割込み優先順位の割り当てを記載してあります。15 は準アイドルタスク(タスク処理が無い時に main より優先して処理)とし、必ず多重割込み許可にします。

## 2.10. 割込みベクタの動的制御

本版では割込みベクタを RAM に持っているため、割込みを許可する前に必ずベクタ設定が必要です。ベクタ設定および割込み関数名変更には以下の関数を使用します。

int NVIC_set_vector(割込みレジスタ名, 割込みチャネル番号, 当初の割込み関数名)
int NVIC_chg_vector(割込みチャネル番号, 変更後の割込み関数名)
　　戻り値が EOF なら失敗

割込み関数名を動的に変更できるため、処理状況に応じて最短の割込み時間にすることが可能です。

## 3. タスクの時間配分設計

各タスクの許容処理時間の考え方と確認方法を以下に述べます。

### 3.1. 割込みの時間算定

タスク割当て時間の内、実際にタスク処理に使える時間は割込みを除いた時間となります。そのためまずは割込み時間合計を算出します。

割込み時間合計 = Σ(割込み要因ごとの頻度 x 処理時間)

スケジューラ用タイマ割込みのように 1 周期で 1 回程度の頻度であれば例えば適当に合算して 100µs 程度(10µs x 10 要因)と考えて良い場合も多いです。しかし高頻度の割込み要因については十分吟味が必要です。例えば DMA を使用しない 460.8kbps 通信であれば 4ms 中に 185 回の割込みが生じますので、1 回の処理時間を 4µs とすると 740µs を占有することになります。

### 3.2. 各タスクへの時間割当て

タスク割当て時間から割込み時間合計を除いた時間を各タスクに割振ります。例えばタスク割当て時間 4ms, 割込み時間合計 1.7ms であれば、2.3ms を割振ります。

①まずは 1 周期内で必ず実行完了しなければならないタスクの合計時間を算出します。
例えば 7 タスク合計で 1.1ms のように 2.3ms より十分短ければ問題ありません。

②次に 1 周期内で必ずしも実行完了しなくて良いタスク(分割処理タスク)へ残り時間を割付けます。このとき分割処理の 1 回の処理時間はオーバーラン(タスク割当て時間内から起動したタスクが次の周期で終了する事)を避けるために例えば 200µs のように短い時間にします。なるべく長くしたい場合でも以下の限界時間以内とします。

限界時間 =(一定周期 － タスク割当て時間) x
(1 － 割込み時間合計 / タスク割当て時間)

例えば一定周期 5ms, タスク割当て時間 4ms, 割込み時間合計 1.7ms であれば、
575µs が限度となります(割込み頻度,処理時間が変わらないと仮定して)。
今 200µs とすると割当の残り時間 1.2ms / 0.2ms = 6 スロット時間を分割処理に充てることができます。残りタスク数が 6 以下であれば一巡保証ができます。一巡保証が必要ない(数周期以内に処理着手すればよい)場合はそれ以上のタスク数であっても構いませんが一般的ではありませんので、足りない場合は

スロット時間(max) = 残り時間 / 残りタスク数

とします。

### 3.3. タスクの順序

タスクの呼出し順序は task.h の enum TASK_ID の中に記述した ID 順となります。

個々のタスク処理時間は固定とは限らないため後ろに記述したタスクは起動時間にぶれが生じます。そのためなるべく起動周期を一定に保ちたいタスクを先に記述します。またもしタスク内の処理が一定部分と可変部分から構成されているのであればタスク分割して 1 巡目の処理時間をなるべく一定になるようにします。

なお割込みの有無による変動もあるため、厳格な周期性を要求される場合はタイマ割込みを使用します。

### 3.4. 処理時間の評価方法

①オーバーラン検出

タスク実行が次の周期まで続いてしまった場合は、オーバーランタスクとしてタスク番号が TSOS 内部に登録されます（タスク番号は 0 から始まるタスク ID 順の番号)。この番号は refer_error_task 関数で取得できます。取得すると記録はクリアされます。

なおタスクから抜けなくなるとウォッチドッグ割込みが発生しリセットされます。ウォッチドッグ時間は device.c の DEVICE_init 関数内で設定しています(現在 1 秒)。また、ウォッチドッグによるリセットかどうかは、DEVICE_det_wdt 関数で判断できます(戻り値が EOF ならウォッチドッグによるリセット)。

②一巡保証確認

一定周期で少なくとも 1 回は全タスクを呼出し可能であることを確認するには、一番最後のタスクとして周期連続性を確認する監視タスクを置きます。

例

```
int prev_cnt, total_cnt, error_cnt;  //前回時刻,呼出し回数,エラー回数
void TASK_chk() {
    int t;
    total_cnt++;
    t = KERNEL_time() / (TASK_PERIOD * 1000);
    if (prev_cnt > 0 && t > prev_cnt) {
        if (++prev_cnt != t) error_cnt++;
    }
    prev_cnt = t;
    delay_task(TASK_PERIOD);
}
```

③タスク処理時間の計測

開始時と終了時に KERNEL_time 関数で時刻を取得して差分を算出すればよいです。

ただし純粋なタスク処理時間としては割込み時間を推測して差引く必要があります。割込み頻度が少ないならば何回か計測して最小値を処理時間としても良いでしょう。

④main 関数の動作可能時間もしくはアイドリング時間

main 関数の中で 64 ビットカウントループを作り、カウント値をデバッガで確認します。カウント値 x ループ時間／計測時間が main 関数の動作可能時間比率になります。

例

```
static ulonglong volatile free_cnt;
while(1) [
  free_cnt++;
]
```

例えば、10 秒間に 600 万カウントの結果で、1 ループ 1μs なら 60%となります。

消費電流を測定しておおまかに推測する方法もあります。上の例でコアがフルに動作している時の電流を測り、次の例でアイドリングさせた時の電流を測り、差分の電流をコアの計算上の電流から推測します。

例

```
while(1) [
  __HALT();
]
```

計算式

main 動作可能時間比率 = (フル動作電流測定値－ アイドリング時電流測定値) ／
(フル動作コア電流カタログ値 － IDLE0 コア電流カタログ値)

例えば

フル動作電流測定値=8.0mA, アイドリング時電流測定値=6.2mA

フル動作コア電流カタログ値=4.7mA, IDLE0 コア電流カタログ値=1.8mA

であれば、main 動作可能時間比率 = 62%

### 4. タスク間通信、main 関数との通信

TSOS ではタスクの実行途中で他のタスクが割り込むことはないので、タスク間で受渡すデータやハンドシェーク用フラグの実行順序を気にする必要はありません。

一方で main 関数とタスク間では、main 実行中にタスクが割込む可能性があるため、データ受渡準備が整ってからハンドシェーク用フラグを立てると言った考慮が必要です。

改訂履歴

| 版 | 日付 | 記事 |
|---|---|---|
| 初版 | 2024/3/5 | 新規作成 |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |