μITRON4.0仕様の徹底解説
パート3
「スタンダードプロファイルの機能」

株式会社日立製作所 半導体グループ
山田真二郎

タスク管理機能

タスクの状態を直接的に操作/参照する機能

[S] ***CRE_TSK***	タスクの生成(静的API)
cre_tsk	タスクの生成
acre_tsk	タスクの生成(ID番号自動割付け)
del_tsk	タスクの削除
[S] ***act_tsk, iact_tsk***	タスクの起動
[S] ***can_act***	タスク起動要求のキャンセル
sta_tsk	タスクの起動（起動コード指定）
[S] ext_tsk	自タスクの終了
exd_tsk	自タスクの終了と削除
ter_tsk	タスクの強制終了
[S] chg_pri	タスク優先度の変更
[S] ***get_pri***	タスク優先度の参照
ref_tsk	タスクの状態参照
ref_tst	タスクの状態参照(簡易版)

[S]：スタンダードプロファイル
赤文字イタリック：μITRON4.0で追加されたAPI

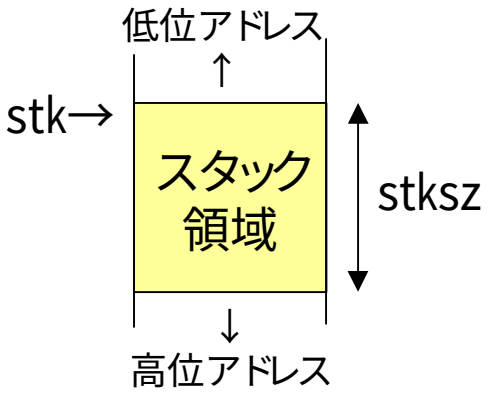
タスク生成のパラメータ

値を変更　　New

tskatr　タスク属性　((TA_HLNG┆TA_ASM) ┆ [TA_ACT])
　TA_ACT:生成後に実行可能状態とする
exinf　拡張情報　ref_tskでは参照できない　　Change
task　タスクの起動番地
itskpri　タスクの起動時優先度
stksz　タスクのスタックサイズ
New stk　タスクのスタック領域の先頭番地。NULL指定で自動割当て。
(以降、実装独自に拡張可能)

[スタンダードプロファイル]

- **stkにNULL以外を指定された場合の機能はサポート不要**
- **タスクの優先度は1～16以上 (TMIN_TPRI=1, TMAX_TPRI>16)**



#以降、図では上を低位アドレスとして記述

オブジェクトの生成と削除

タスクに限らず、全てのオブジェクトについてスタンダードプロファイルでは静的APIによる生成のみが要求される。
動的APIによる生成・削除は、スタンダードプロファイル外

acre_tsk：オブジェクトの生成(ID番号の自動割付け)

タスクに限らず、全てのオブジェクトについてID番号を自動的に割り付けるAPI（acre_yyy）を設けた。
　→ID空間を静的に決めておく必要が無い
カーネルは指定された内容でオブジェクトを生成し、生成したオブジェクトのID番号をリターン値として返す。

```
ER_ID objectid = acre_yyy(T_CYYY *pk_cyyy);
```

#acre_yyyはスタンダードプロファイル外

タスクの起動

[S]	act_tsk, iact_tsk	タスク起動(キューイング可,起動コード無し)
	sta_tsk	タスク起動(キューイング不可,起動コード有り)
[S]	can_act	タスク起動要求のキャンセル

act_tskでは、起動要求がキューイングされる。また、自タスクにも起動要求可能。キューイング数はext_tsk, ter_tskで減算される。can_act は、act_tskによる起動要求を無効にする。sta_tskによる起動要求はキャンセルできない。

周期的にタスクを実行させたい場合に、周期起動ではコンテキストの保存が不要。コンテキストは、スタックに保存する実装が多く、自動車プロファイルの制約タスクにおいては、スタックを1本で実現できる。

[スタンダードプロファイル]

- **起動要求キューイング数(TMAX_ACTCNT)は1以上**

タスクのC言語記述形式


```
void task(VP_INT  exinf )
{
      タスク本体処理

      ext_tsk();
}
```

act_tskでの起動 :タスクの拡張情報
sta_tskでの起動 :sta_tskで指定した起動コード

ext_tskを発行しなくても、タスクのメイン関数から
リターンすると、ext_tskと等価の振る舞いをする

起動要求キューイングの使用例

周期的にact_tskで起動されるタスクの場合、can_actを用いることで周期内に処理が完了したかを判断できる。

```
void CyclicTask(VP_INT exinf)
{
   周期的な処理
   if(can_act(myid) >0) // 既に次の要求が来ている
      NotInTime();      // 間に合わなかった場合の処理
}
```

タスク優先順位の厳密化(1)

原則： 同じ優先度の実行可能状態のタスク中では、最も早く実行可能状態となったタスクが最も高い優先順位を持つ。

◆解釈１

以下のケースでは、当該優先度の中でもっとも遅く実行可能状態になった、つまりその優先度の中では最も優先順位が低いとする。

(1)休止状態, 待ち状態, 強制待ち状態から実行可能状態に遷移したタスク
(2)chg_priの対象タスク
(3)rot_rdqの対象優先度の中で最高の優先順位を持っていたタスク
(4)起動要求がキューイングされている状態でext_tskを発行した、またはter_tskを発行されたタスク

◆解釈２

以下のケースでは、当該タスクは当該優先度の中で最高の優先順位を保つとする。

(1)プリエンプトされたタスク
(2)待ち状態に遷移するAPIを発行した時点で、既に条件が満たされていた場合(ポーリングを含む)

タスク優先順位の厳密化(2)

◆待ち状態の間の優先順位

待ち状態のタスクについては、待ち行列の順に待ち解除条件が評価されることとした。
set_flgでの一斉動作やオブジェクト削除時などには、待ち行列の順に待ちが解除され、実行可能状態に移行することになる。
また、chg_priでタスク優先度順の待ち行列につながれているタスクの優先度を変更した場合は、変更後の優先度と同じ優先度を持つタスクの中では、対象タスクを最後(最低順位)につなぐこととした。

なお、以下についてはそれらの中での待ち解除の順序は実装依存である。
(1)同時刻にタイムアウトしたタスクの待ち解除の順序
(2)ランデブポート削除時の、呼出し待ちタスクと受付待ちタスクの間の
　待ち解除順序

無駄を省いた状態参照サービスコール

◆get_pri：タスク優先度の参照

```
ER ercd = get_pri(ID tskid, PRI *p_tskpri);
```

送信するメッセージの優先度に自タスクの優先度を設定する場合などに有用

#現在優先度とベース優先度についてはミューテックスの説明を参照

◆ref_tst：簡易版タスク状態参照

```
ER ercd = ref_tst(ID tskid, T_RTST *pk_rtst);
```

ref_tskよりもオーバヘッドの小さいref_tstを新設。
タスク状態と待ち要因のみを参照可能。

 # ref_tstはスタンダードプロファイル外

タスク付属同期機能

[S]	slp_tsk	起床待ち
[S]	tslp_tsk	起床待ち(タイムアウト有り)
[S]	wup_tsk, iwup_tsk	タスクの起床
[S]	can_wup	タスク起床要求のキャンセル
[S]	rel_wai, irel_wai	待ち状態の強制解除
[S]	sus_tsk	強制待ち状態への移行
[S]	rsm_tsk	強制待ち状態からの再開
[S]	frsm_tsk	強制待ち状態からの強制再開
[S]	dly_tsk	自タスクの遅延

[スタンダードプロファイル]

- **起床要求キューイング数(TMAX_WUPCNT)は1以上**
- **強制待ち要求ネスト数(TMAX_SUSCNT)は1以上**

wup_tsk, sus_tskでの自タスク指定

[μITRON3.0] 自タスクへのsus_tsk, wup_tskは不可

↓

[μITRON4.0] 自タスクへのsus_tsk, wup_tskが可能

Javaスレッド等の自タスクによる待ちと他のタスクによる待ちを区別しないAPIのインタフェーサをカーネル上に効率的に実装するのを容易にするため、自タスクへのsus_tskを可能とした。
wup_tskもact_tskに合わせて自タスク指定を可能とした。

can_wup：リターン値の改善

使い勝手を考慮し、リターンパラメータを以下のようにサービスコールの返値として返すこととした。

```
ER_UINT wupcnt = can_wup(ID tskid)
```

得られる情報が正値に限定可能な以下のサービスコールにも適用

- acre_yyy [生成したID]
- can_act [起動キューイング数]
- can_wup [起床キューイング数]
- rcv_mbf, prcv_mbf, trcv_mbf [受信メッセージサイズ]
- cal_por, tcal_por [返答メッセージサイズ]
- acp_por, tacp_por [呼び出しメッセージサイズ]

タスク例外処理機能

UNIXのシグナル機能を簡略化したような機能で、μITRONで新規に導入。
主な用途は、

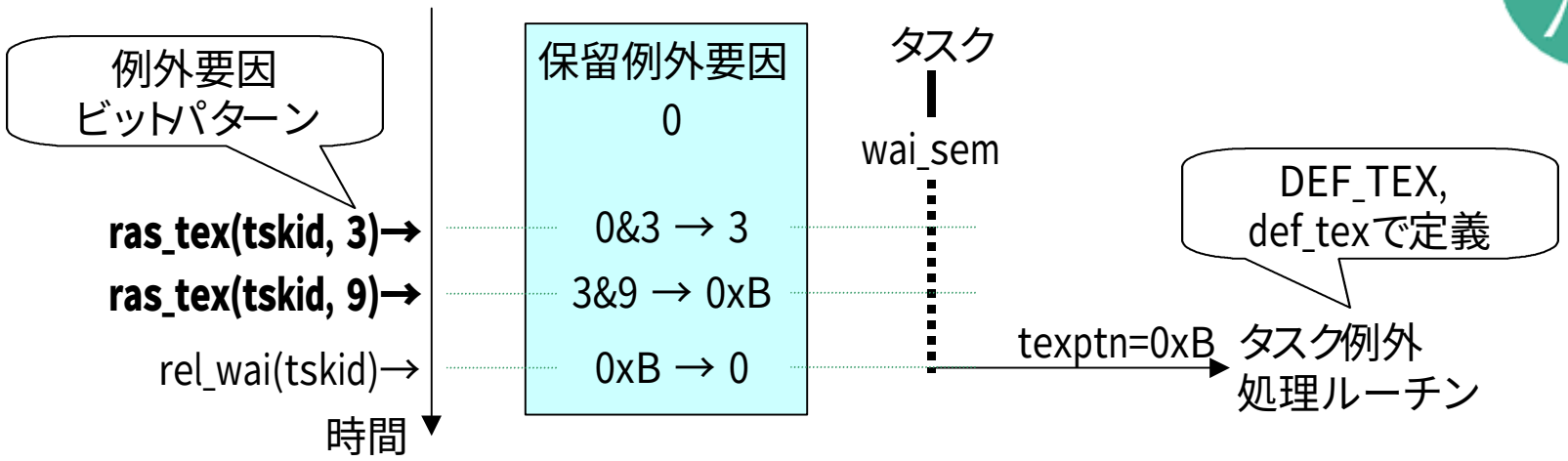
- ゼロ除算などのCPU例外をタスクに伝える
- 他タスクに終了要求を出す
- タスクにデッドラインが来たことを通知する

[S]	*DEF_TEX*	タスク例外処理ルーチンの定義(静的API)
	def_tex	タスク例外処理ルーチンの定義
[S]	*ras_tex, iras_tex*	タスク例外処理の要求
[S]	*dis_tex*	タスク例外処理の禁止
[S]	*ena_tex*	タスク例外処理の許可
[S]	*sns_tex*	タスク例外処理禁止状態の参照
	ref_tex	タスク例外処理の状態参照

[スタンダードプロファイル]

- **例外要因のビットパターン(TBIT_TEXPTN)は16ビット以上**

タスク例外の基本動作



- DEF_TEX., def_texで、タスク毎にひとつだけタスク例外処理ルーチンを定義可能。
- ras_texでタスク例外を要求。この時、例外の要因をビットパターンで指定。
- カーネルは、タスクに要求された例外要因のパターンの論理和を記憶 (保留例外要因のビットパターン)。
- タスク例外ルーチンは、タスクが実行状態になったときに起動。*
- タスク例外処理ルーチンには、保留例外要因が渡される。この時、保留例外要因は0クリアされる。
- タスク例外処理ルーチンは、タスクと同じコンテキストで動作。

** 厳密には後述*

タスク例外処理ルーチンのC記述形式


```
void texrtn(TEXPTN texptn, VP_INT exinf)
{

        タスク例外処理
        ルーチン本体

}
```

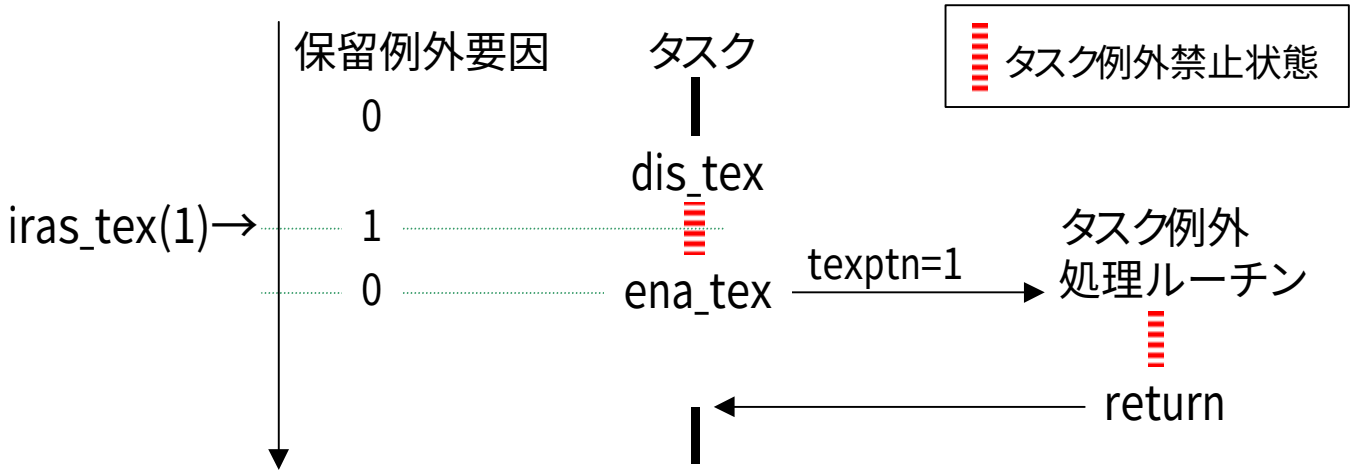
定義時に指定した拡張情報

受け付けた例外要因
のビットパターン

dis_tex, ena_tex：タスク例外処理の禁止,許可

[禁止状態]
- ◆タスク起動時
- ◆タスク例外処理ルーチンが未定義の時
- ◆タスク例外処理ルーチン起動時
- ◆dis_tex後

[許可状態]
- ◆タスク例外処理ルーチン終了直後
- ◆ena_tex後

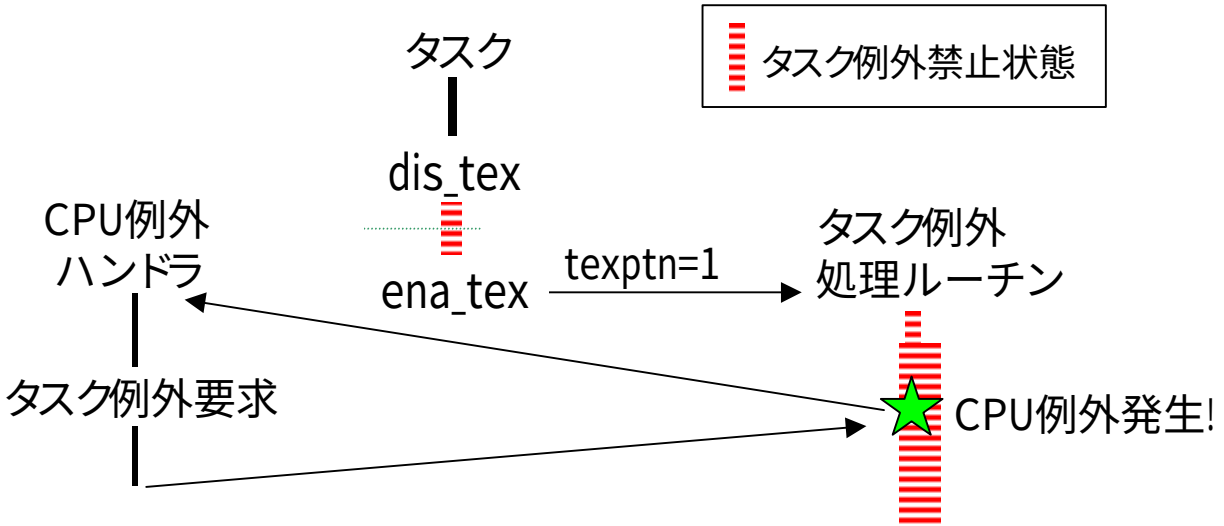


#タスク例外処理許可状態では、常に保留例外要因は0となる

dis_tex, ena_tex：タスク例外処理の禁止,許可

[禁止状態]
- タスク起動時
- タスク例外処理ルーチンが未定義の時
- タスク例外処理ルーチン起動時
- dis_tex後

[許可状態]
- タスク例外処理ルーチン終了直後
- ena_tex後



#タスク例外処理許可状態では、常に保留例外要因は0となる

タスク例外処理ルーチンの起動/終了

以下の条件が満たされると、タスクの制御はメインルーチンからタスク例外処理ルーチンへ移る。

- 対象タスクのタスク例外処理許可状態である
- 対象タスクの保留例外要因が0でない
- 対象タスクが実行状態
- 非タスクコンテキストまたはCPU例外ハンドラが実行されていない

タスク例外処理ルーチンから復帰すると、タスクのメインルーチンの実行を継続する。

sns_tex：タスク例外処理禁止状態の参照

BOOL state = sns_tex();

state にTRUE(禁止状態)が返るケース

- 実行状態のタスクが例外禁止状態
- 実行状態のタスクの例外処理ルーチンが未定義
- 非タスクコンテキストから呼び出した場合で、実行状態のタスクが存在しない

タスクコンテキストから発行した場合は、実行状態のタスクは自タスクと同じ

使用例：ソフトウェア部品内で一時的にタスク例外を禁止する

```
ER f()
{
   ercd = E_CTX;
   if(sns_ctx() == FALSE) {  // タスクコンテキスト?
      tex = TRUE;
      if(sns_tex() == FALSE) {  // タスク例外を一時禁止
            dis_tex();
            tex = FALSE;
      }
      ソフトウェア部品の処理
      if(!tex)              // タスク例外許可状態に戻す
         ena_tex();
   }
   return ercd;
}
```

セマフォ

μITRON3.0仕様からの機能変更はない

[S]	***CRE_SEM***	セマフォの生成(静的API)
	cre_sem	セマフォの生成
	acre_sem	セマフォの生成(ID番号自動割付け)
	del_sem	セマフォの削除
[S]	sig_sem, isig_sem	セマフォ資源の返却
[S]	wai_sem	セマフォ資源の獲得
[S]	pol_sem　*	セマフォ資源の獲得(ポーリング)
[S]	twai_sem	セマフォ資源の獲得(タイムアウト有り)
	ref_sem	セマフォの状態参照

** μITRON3.0のpreq_semから改称*

[スタンダードプロファイル]

- **セマフォ最大資源数は65535以上(TMAX_MAXSEM)**

イベントフラグ

[S]	***CRE_FLG***	イベントフラグの生成(静的API)
	cre_flg	イベントフラグの生成
	acre_flg	イベントフラグの生成(ID番号自動割付け)
	del_flg	イベントフラグの削除
[S]	set_flg, iset_flg	イベントフラグのセット
[S]	clr_flg	イベントフラグのクリア
[S]	wai_flg	イベントフラグ待ち
[S]	pol_flg	イベントフラグ待ち(ポーリング)
[S]	twai_flg	イベントフラグ待ち(タイムアウト有り)
	ref_flg	イベントフラグの状態参照

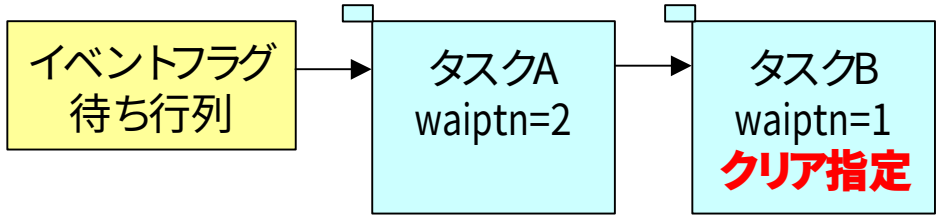
[スタンダードプロファイル]

- **イベントフラグは16ビット以上(TBIT_FLGPTN)**
- **複数タスクの待ちを許すTA_WMUL属性はサポート不要**

μITRON3.0のクリア指定の問題

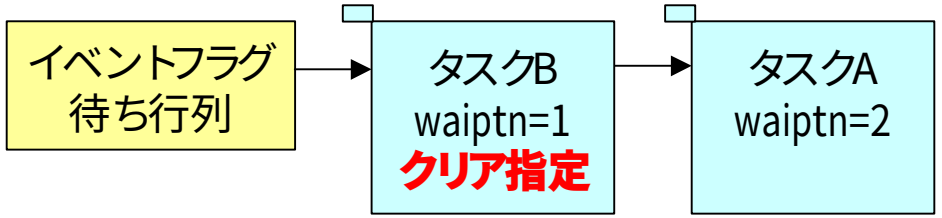
タスクA：waiptn=2, クリア指定無し
タスクB：waiptn=1, クリア指定有り

[タスクA,Bの順にwai_flgが発行された場合]



その後set_flg(setptn=3)→**タスクA,Bがともに待ち解除**

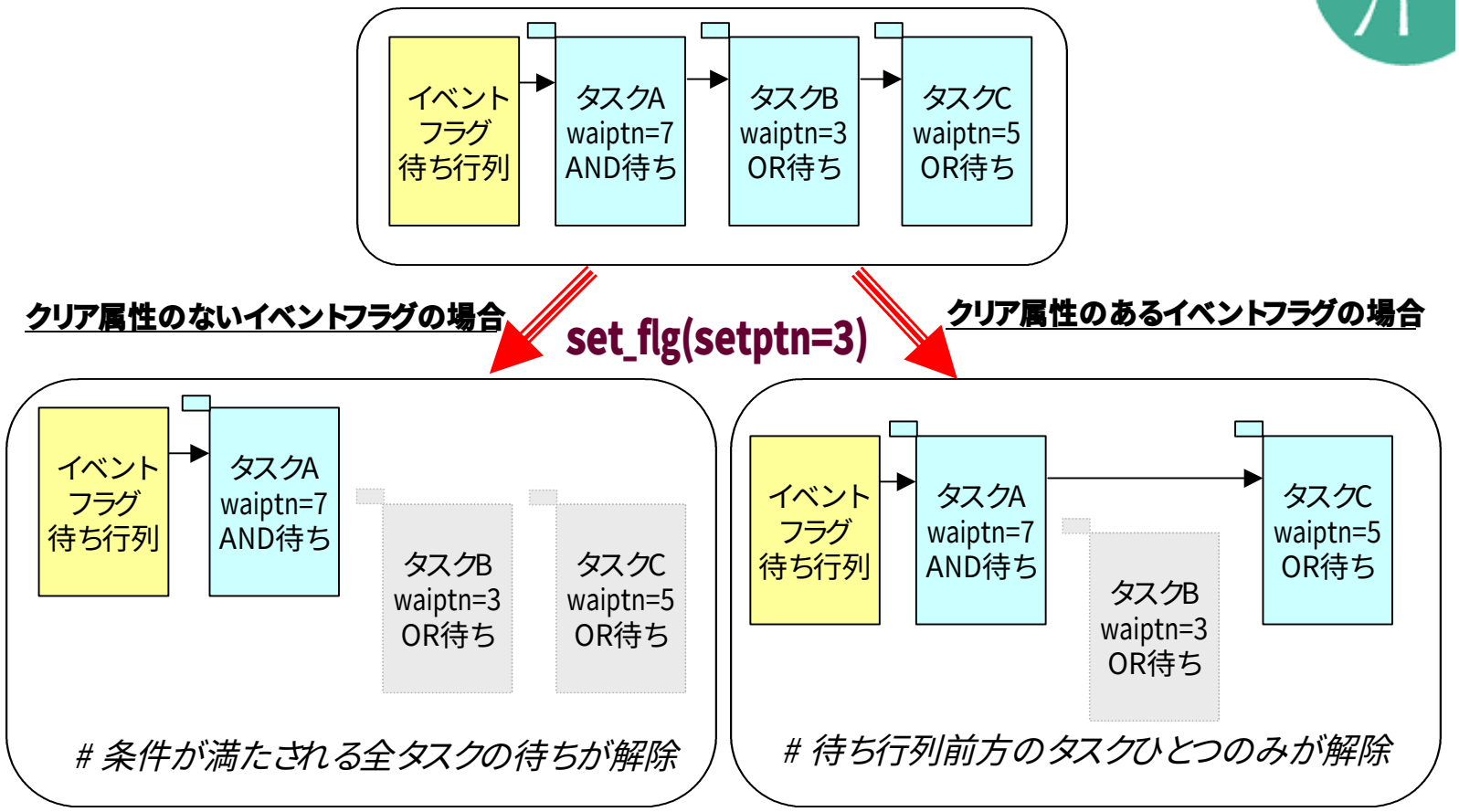
[タスクB,Aの順にwai_flgが発行された場合]



その後set_flg(setptn=3)→**タスクBだけが待ち解除**

タスクA,Bの実行順序によって振る舞いが変わる！

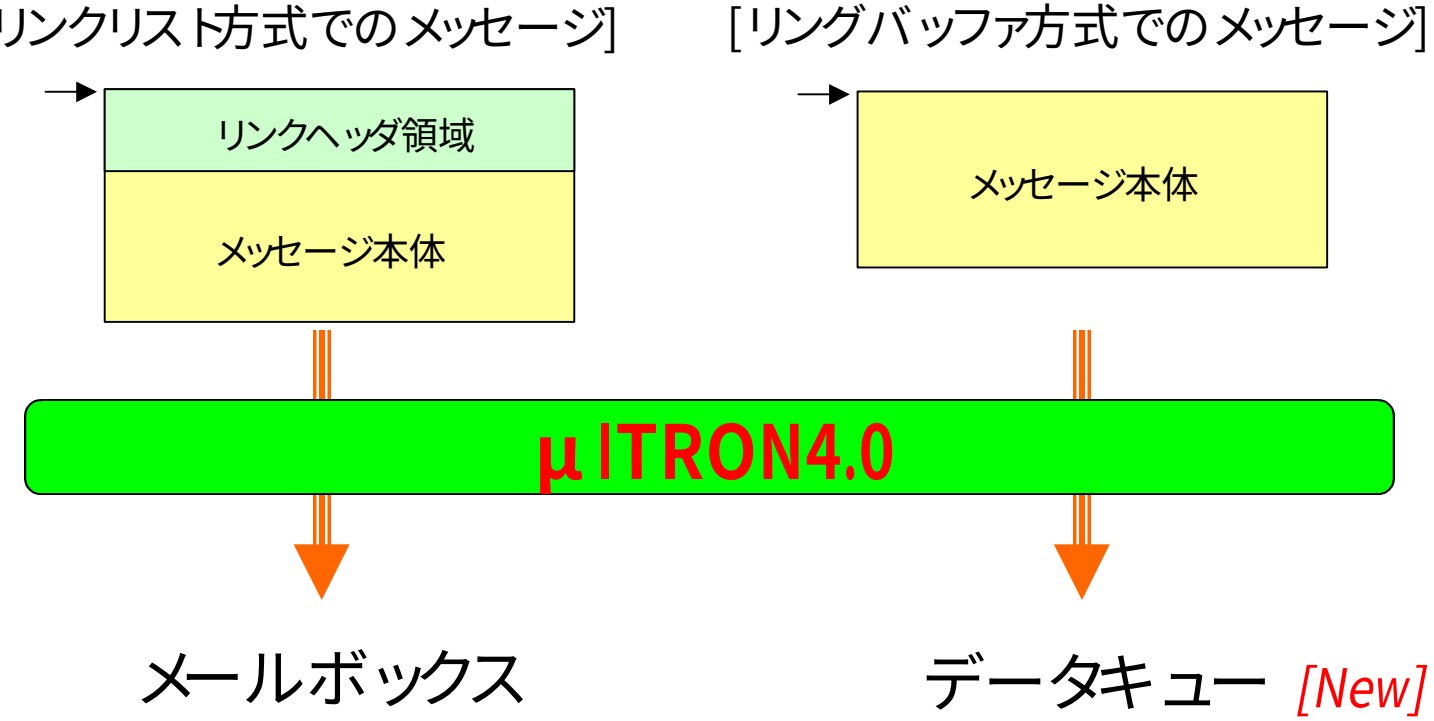
μITRON4.0のクリア属性による振る舞いの違い



 # クリア属性のイベントフラグでは、複数のタスクが同時に待ち解除となることはない

メールボックスとデータキュー

μITRON3.0では、メールボックスの実装方法を規定していなかったが、
実装方法によって扱うメッセージの構造が異なるため、互換性に問題

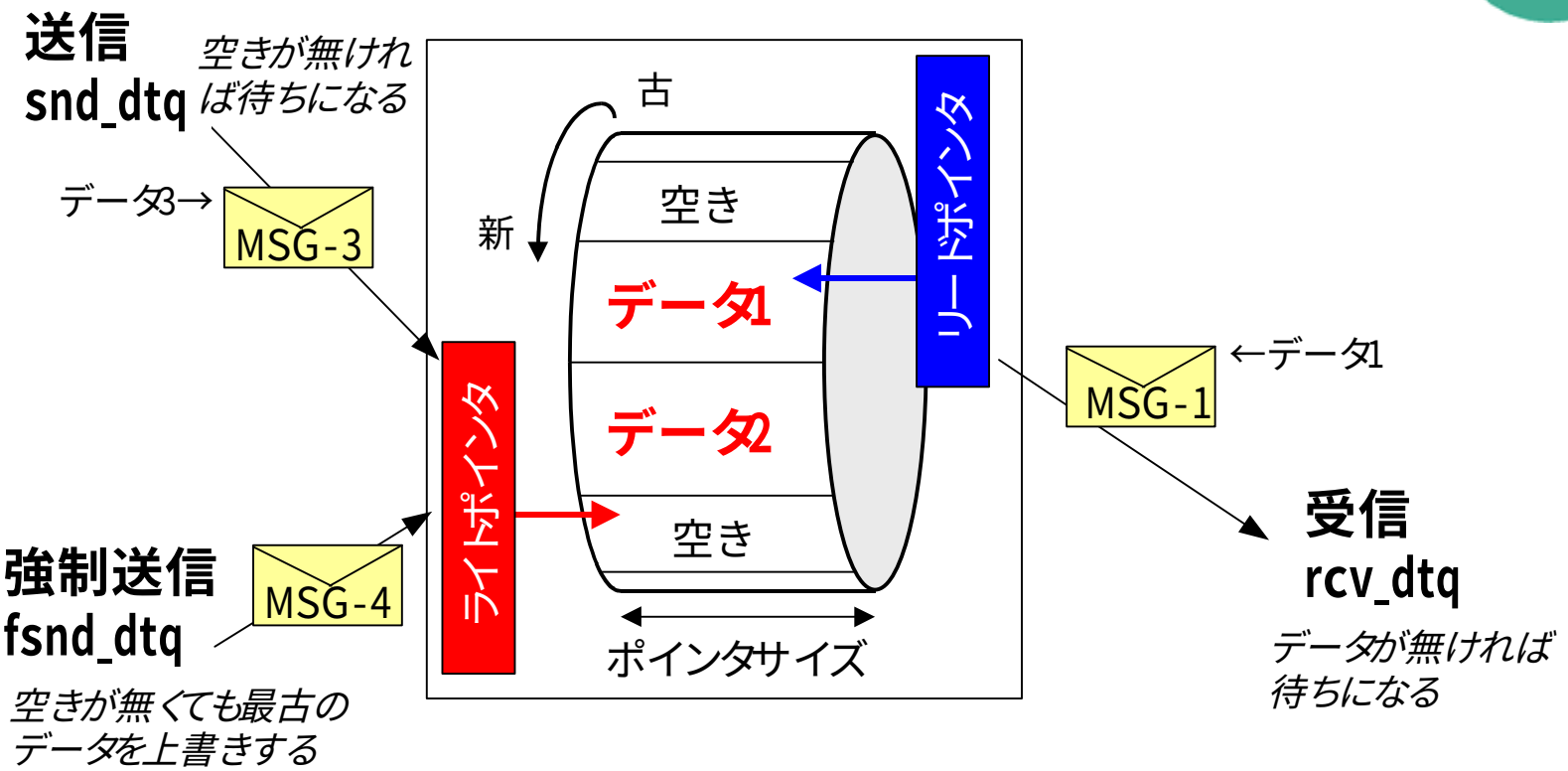


データキュー

1ワードのデータを受け渡しすることによって同期と通信を行うためのオブジェクト。μITRON4.0で新規導入。

[S]	CRE_DTQ	データキューの生成(静的API)
	cre_dtq	データキューの生成
	acre_dtq	データキューの生成(ID自動割付け)
	del_dtq	データキューの削除
[S]	snd_dtq	データキューへの送信
[S]	psnd_dtq, ipsnd_dtq	データキューへの送信(ポーリング)
[S]	tsnd_dtq	データキューへの送信(タイムアウト有り)
[S]	fsnd_dtq, ifsnd_dtq	データキューへの強制送信
[S]	rcv_dtq	データキューからの受信
[S]	prcv_dtq	データキューからの受信 (ポーリング)
[S]	trcv_dtq	データキューからの受信(タイムアウト有り)
	ref_dtq	データキューの状態参照

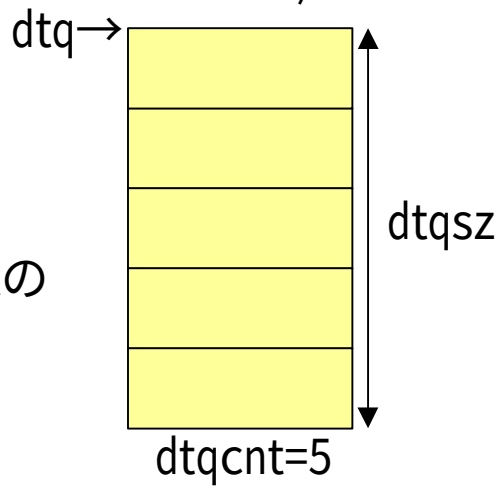
データキューの一般的な使用 -実装イメージ



#送受信するデータはメッセージアドレスに限定されない

データキューの生成パラメータ

dtqatr　データキュー属性
　　送信待ち行列　(TA_TFIFO||TA_TPRI)
dtqcnt　データキュー領域の容量(格納できるデータの個数)
dtq　データキュー領域のアドレス
　　NULL指定で自動割当て
(以降、実装独自に拡張可能)



カーネル構成マクロ TSZ_DTQ によって、dtqcntのデータキューに領域サイズを知ることができる。

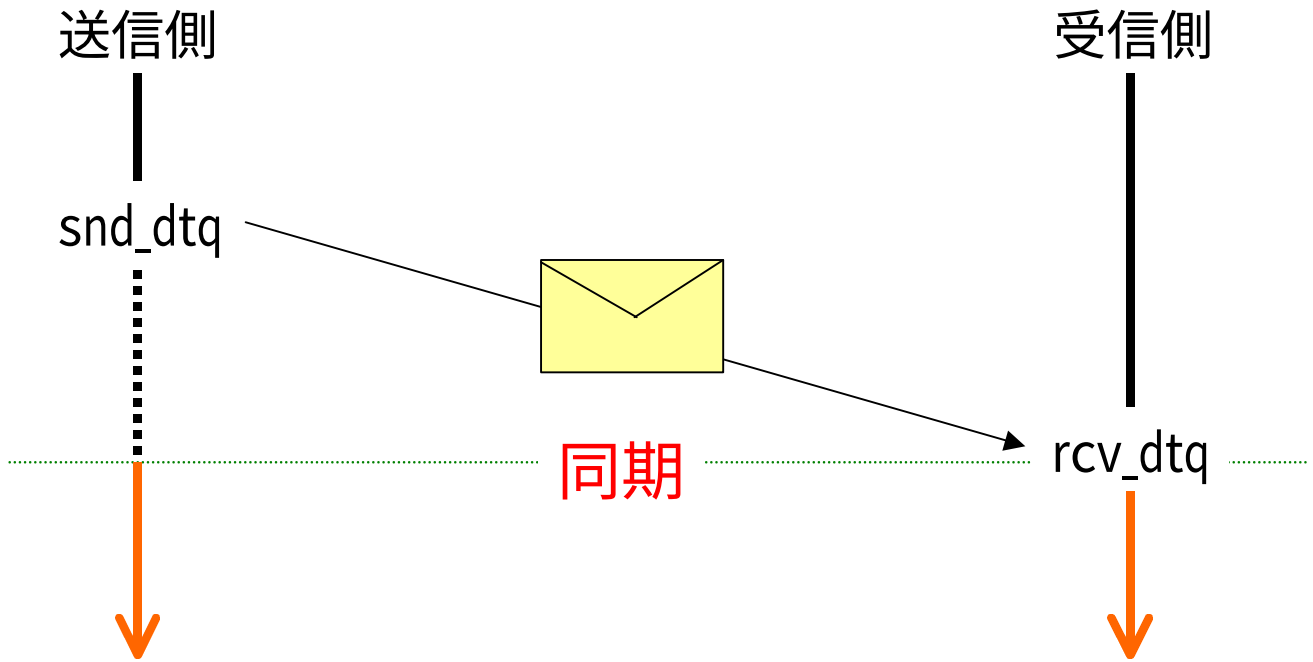
SIZE dtqsz = TSZ_DTQ(UINT dtqcnt)

[スタンダードプロファイル]

- **dtqにNULL以外を指定された場合の機能はサポート不要**

サイズ0のデータキュー

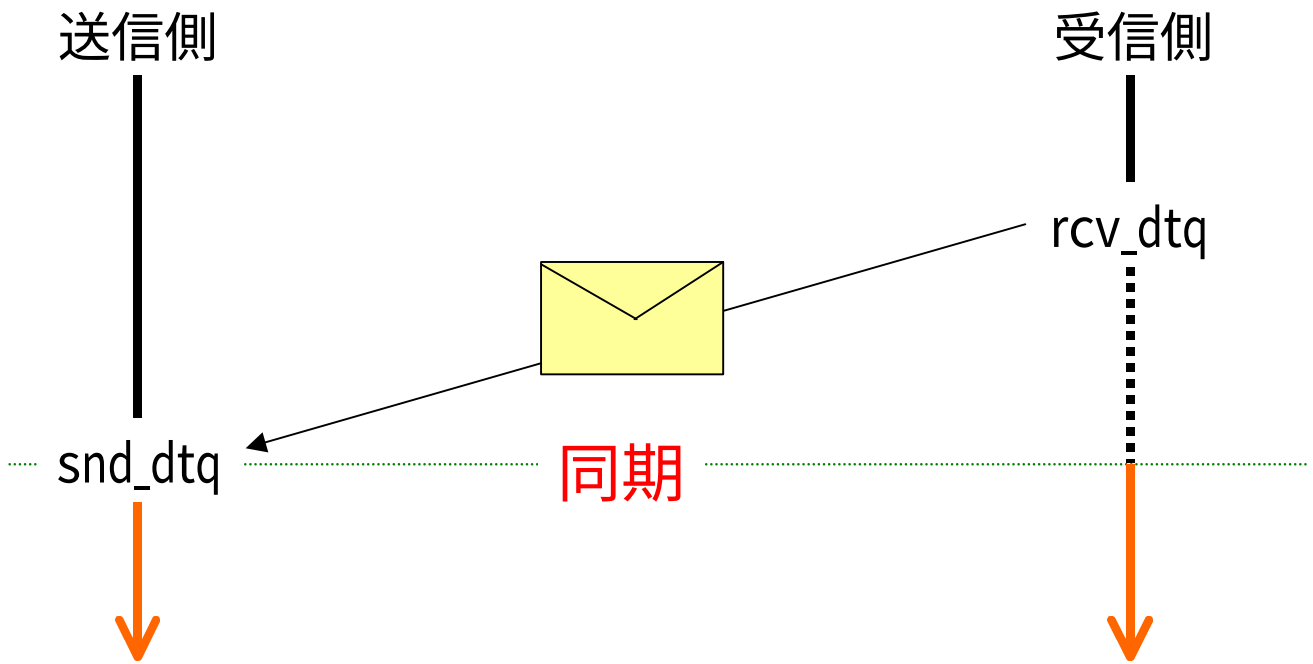
サイズ0のデータキューでは、送信側と受信側は常に同期



#サイズ0のデータキューには、強制送信(fsnd_dtq)はできない

サイズ0のデータキュー

サイズ0のデータキューでは、送信側と受信側は常に同期



#サイズ0のデータキューには、強制送信(fsnd_dtq)はできない

メールボックス

共有メモリ上に置かれたメッセージを受け渡しすることにより、同期と通信を行うためのオブジェクト

[S]	***CRE_MBX***	メールボックスの生成(静的API)
	cre_mbx	メールボックスの生成
	acre_mbx	メールボックスの生成(ID番号自動割付け)
	del_mbx	メールボックスの削除
[S]	snd_mbx *	メールボックスへの送信
[S]	rcv_mbx *	メールボックスからの受信
[S]	prcv_mbx *	メールボックスからの受信 (ポーリング)
[S]	trcv_mbx *	メールボックスからの受信(タイムアウト有り)
	ref_mbx	メールボックスの状態参照

** μITRON3.0のxxx_msgからxxx_mbxに改称*

メッセージ優先度(TA_MPRI属性)

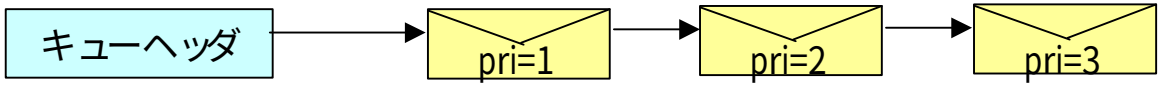
メールボックス生成時に、優先度毎のキューヘッダ領域のアドレスを指定する。
NULL指定時は、カーネルがキューヘッダ領域を確保する。
→カーネル構成マクロ TSZ_MPRIHD (キューヘッダ領域サイズを得る)

SIZE mprihdsz = TSZ_MPRIHD(PRI maxmpri)

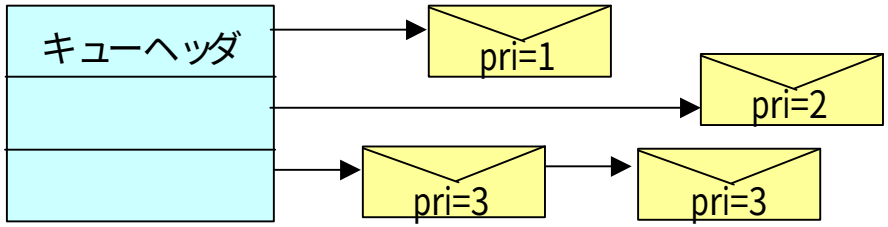
[スタンダードプロファイル]

- **メッセージ優先度は1～16以上(TMIN_MPRI=1, TMAX_MPRI>16)**
- **キューヘッダアドレスにNULL以外を指定された場合の機能はサポート不要**

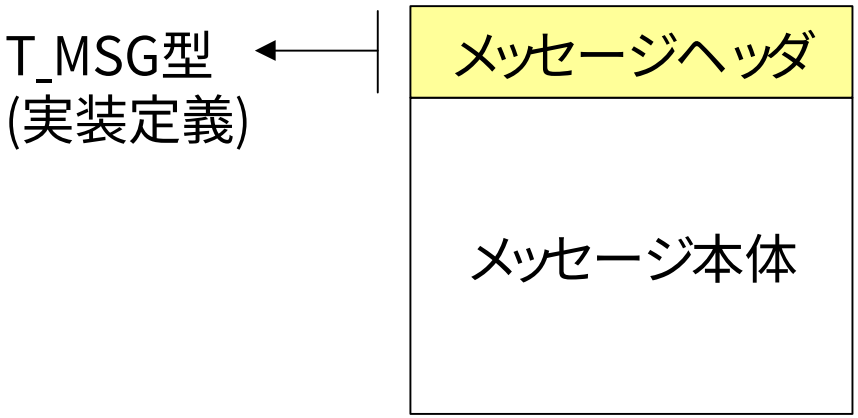
[常に1本のリストで管理する実装例]



[優先度毎のリストで管理する実装例]

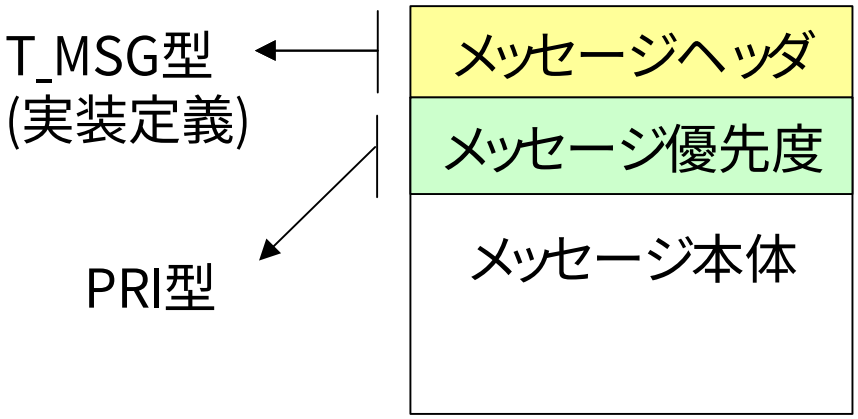


メッセージの構造



TA_MPRI(メッセージは優先度順)属性の場合はT_MSG_PRI型、
TA_MFIFO(メッセージはFIFO順)属性の場合はT_MSG型のメッセージを使う。
APIでは属性に関わらず、メッセージをT_MSG型として扱う。

メッセージの構造




```
typedef struct t_msg_pri {
    T_MSG   msgque;
    PRI     msgpri;
} T_MSG_PRI;
```

TA_MPRI(メッセージは優先度順)属性の場合はT_MSG_PRI型、
TA_MFIFO(メッセージはFIFO順)属性の場合はT_MSG型のメッセージを使う。
APIでは属性に関わらず、メッセージをT_MSG型として扱う。

固定長メモリプール

μITRON3.0仕様からの機能変更はない

[S]	***CRE_MPF***	固定長メモリプールの生成(静的API)
	cre_mpf	固定長メモリプールの生成
	acre_mpf	固定長メモリプールの生成(ID番号自動割付け)
	del_mpf	固定長メモリプールの削除
[S]	get_mpf *	固定長メモリブロックの獲得
[S]	pget_mpf *	固定長メモリブロックの獲得(ポーリング)
[S]	tget_mpf *	固定長メモリブロックの獲得(タイムアウト有り)
[S]	rel_mpf *	固定長メモリブロックの返却
	ref_mpf	固定長メモリプールの状態参照

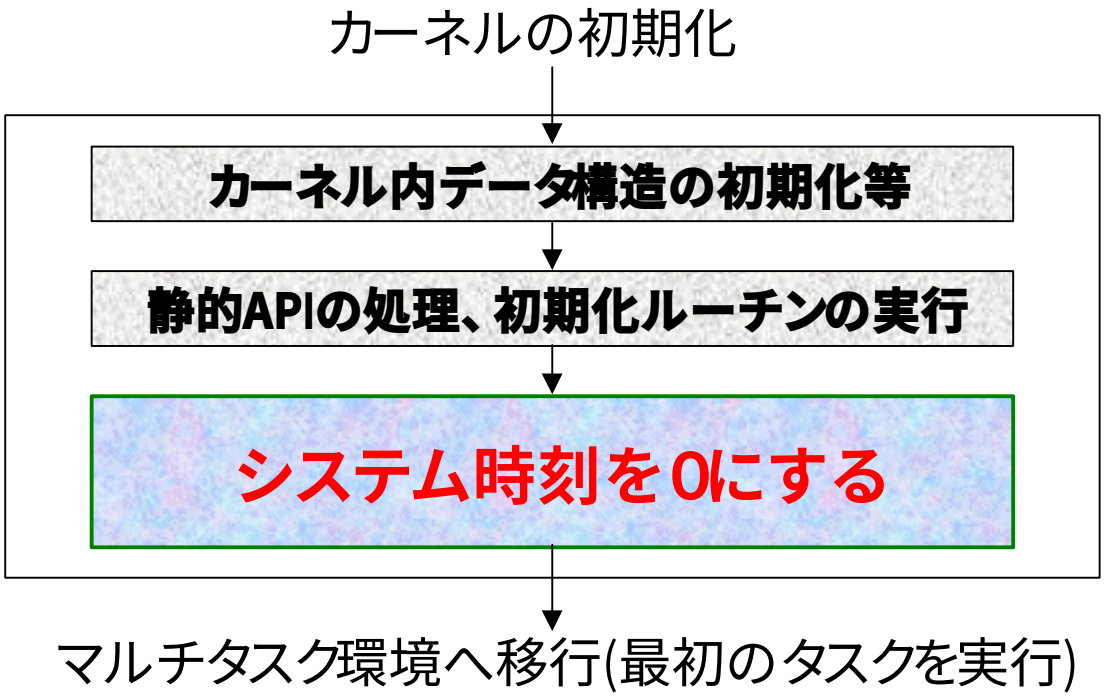
** μITRON3.0のxxx_blfからxxx_mpfに改称*

システム時刻管理

システム時刻を操作するための機能

[S]	set_tim	システム時刻の設定
[S]	get_tim	システム時刻の参照
[S]	*isig_tim*	タイムティックの供給

システム時刻の初期化


カーネルの初期化
カーネル内データ構造の初期化等
静的APIの処理、初期化ルーチンの実行
システム時刻を0にする
マルチタスク環境へ移行(最初のタスクを実行)

isig_tim：タイムティックの供給(1)

ハードウェアに依存せずにカーネルを供給可能とするために導入。

周期的なハードウェアタイマの割込みハンドラ(または割込みサービスルーチン)からisig_timを発行することで、カーネルは以下のような時間に関する処理を行う。

- システム時刻の更新
- タスクのタイムアウト処理
- タイムイベントハンドラの起動

#システム時刻を更新する機構をカーネル内部に持つ場合は、isig_timをサポートする必要はない。

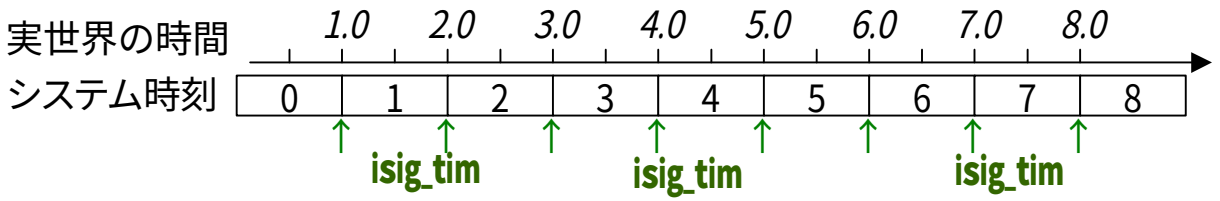
isig_timによってどれだけシステム時刻が更新されるかは実装定義
→カーネル構成定数：TIC_NUME, TIC_DENO (次紙参照)

isig_tim：タイムティックの供給(2)

$$\text{タイムティックの周期} = \frac{\text{TIC\_NUME(タイムティックの周期の分子)}}{\text{TIC\_DENO(タイムティックの周期の分母)}}$$

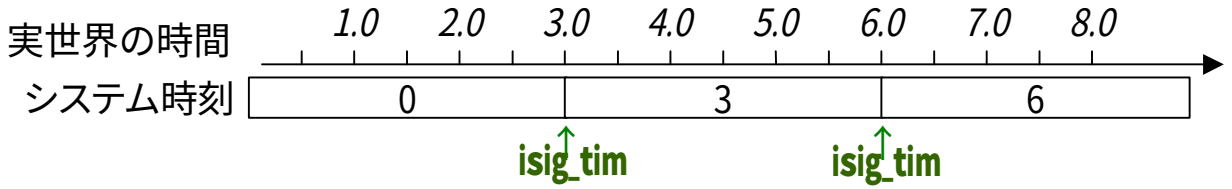
アプリケーションは、このカーネル構成定数を使ってシステム時刻に依存したカーネル機能の時間精度を知ることができる。

例1：1msec周期のタイマ割込みを使ってシステム時刻の単位時間をmsecとする



この場合、TIC_NUME=TIC_DENO=1

例2：3msec周期のタイマ割込みを使ってシステム時刻の単位時間をmsecとする



この場合、TIC_NUME=3, TIC_DENO=1

時間の単位

APIで使用する時間パラメータの単位時間は実装定義。
APIで使用する時間パラメータのデータ型は、以下の4種類。

TMO：タイムアウト
RELTIM：相対時間
SYSTIM：システム時刻
OVRTIM：プロセッサ時間

[スタンダードプロファイル]

- **TMO, REL_TIM, SYSTIMの単位はすべて1msecで、16bit以上**
  (OVRTIMはスタンダードプロファイル外であるオーバランハンドラ機能でのみ使用)

タイムイベントハンドラ

タイムイベントハンドラは時間をきっかけとして起動される処理で、非タスクコンテキストで動作する。

■タイムイベントハンドラの種類

- ◆周期ハンドラ[スタンダードプロファイル]

  指定した周期で起動

- ◆アラームハンドラ

  指定したシステム時刻に一度だけ起動

- ◆オーバーランハンドラ

  タスクが設定された時間を超えてプロセッサを使用したときに起動

周期ハンドラ

周期ハンドラ：一定周期で起動されるタイムイベントハンドラ

	API名称	機能	μITRON3.0との対応
[S]	***CRE_CYC***	周期ハンドラの生成 (静的API)	---
	cre_cyc	周期ハンドラの生成	def_cycによる定義
	acre_cyc	周期ハンドラの生成 (ID番号自動割付け)	---
	del_cyc	周期ハンドラの削除	def_cycによる定義解除
[S]	***sta_cyc***	周期ハンドラの動作開始	act_cycによる活性化
[S]	***stp_cyc***	周期ハンドラの動作停止	act_cycによる非活性化
	ref_cyc	周期ハンドラの状態参照	ref_cyc

周期ハンドラの生成パラメータ

cycatr　ハンドラ属性　((TA_HLNG||TA_ASM) [¦ TA_STA] [¦ TA_PHS])
　　TA_STA :定義後にハンドラを稼動状態にする
　　TA_PHS :位相を保存(スタンダードプロファイル外)
exinf　拡張情報
cychdr　周期ハンドラの起動番地
cyctim　起動周期
cycphs　起動位相
(以降、実装独自に拡張可能)

周期ハンドラのC言語記述形式

```
void cychdr(VP_INT exinf)
{
        周期ハンドラ
        本体の処理

}
```

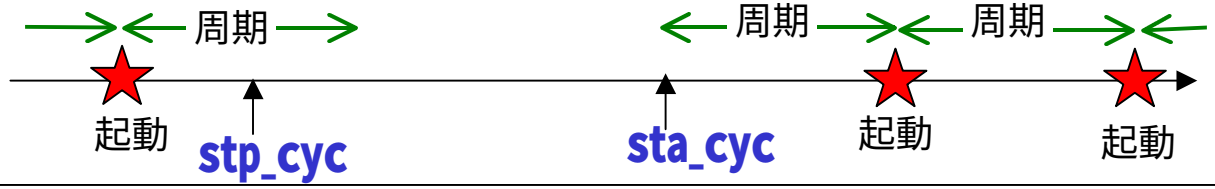
周期ハンドラの拡張情報

起動位相

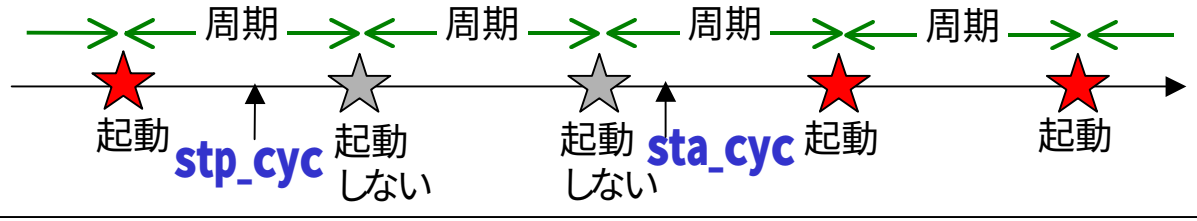
システム時刻0で、周期=100, 起動位相30で周期ハンドラを生成した場合
(#静的APIによる生成では、時刻0で生成したと扱われる)

位相の保存(TA_PHS属性)　(スタンダードプロファイル外)

例1 : 位相を保存しない場合(TA_PHS属性無し)

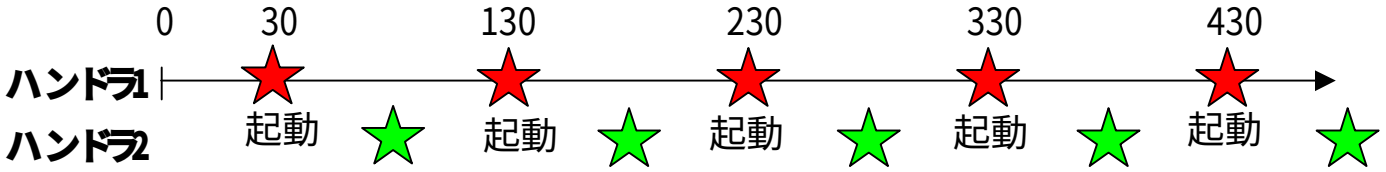


例2 : 位相を保存する場合(TA_PHS属性有り)



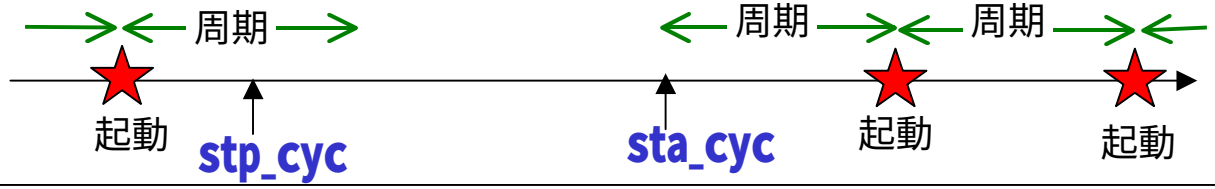
起動位相

システム時刻0で、周期=100, 起動位相30で周期ハンドラを生成した場合
(#静的APIによる生成では、時刻0で生成したと扱われる)

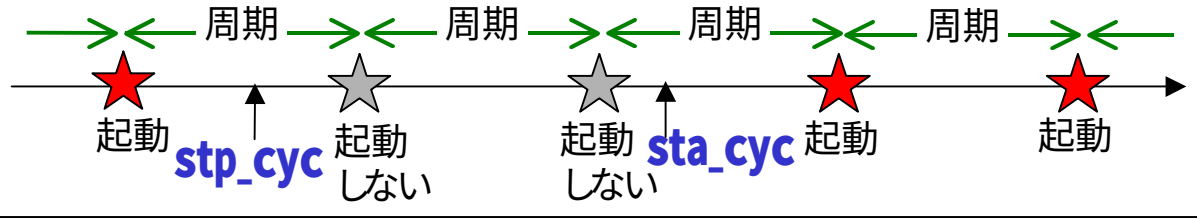


位相の保存(TA_PHS属性)　(スタンダードプロファイル外)

例1：位相を保存しない場合(TA_PHS属性無し)

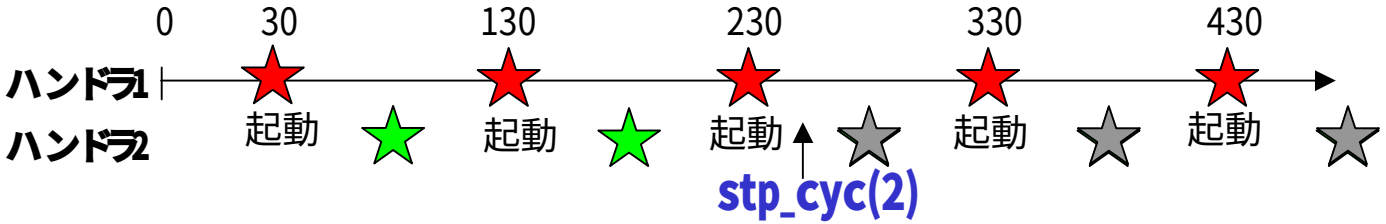


例2：位相を保存する場合(TA_PHS属性有り)



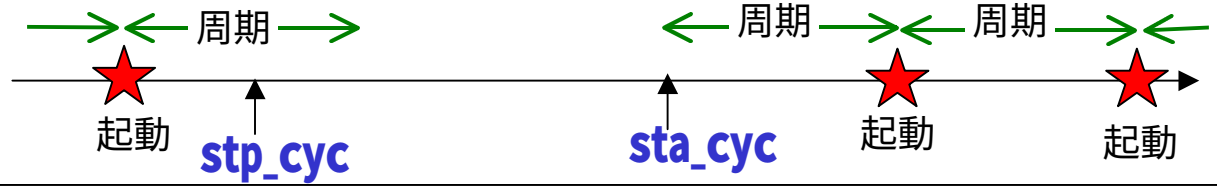
起動位相

システム時刻0で、周期=100, 起動位相30で周期ハンドラを生成した場合
(#静的APIによる生成では、時刻0で生成したと扱われる)

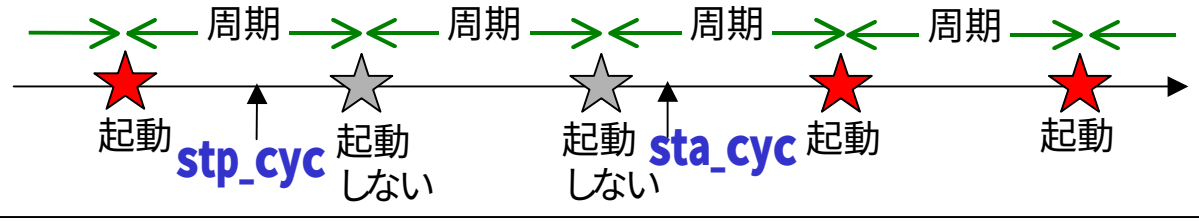


位相の保存(TA_PHS属性)　(スタンダードプロファイル外)

例1：位相を保存しない場合(TA_PHS属性無し)



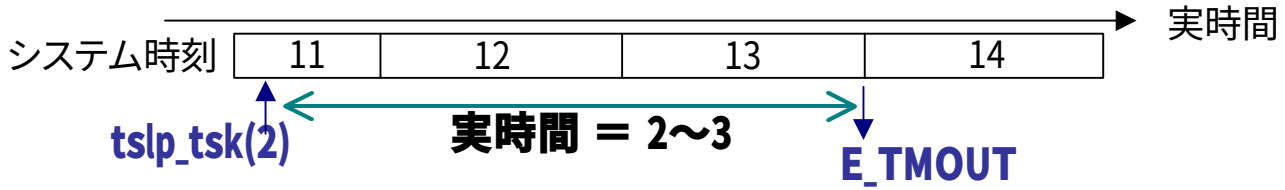
例2：位相を保存する場合(TA_PHS属性有り)



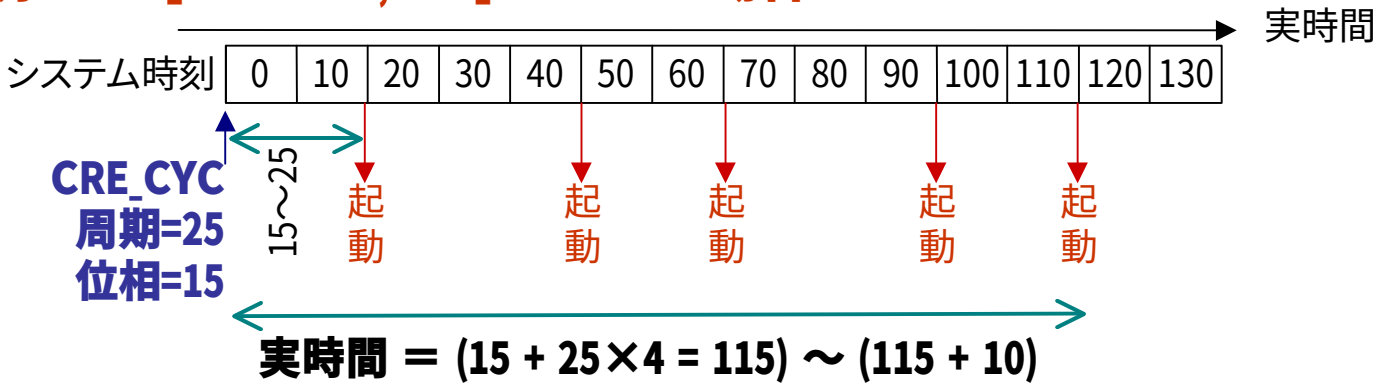
時間管理の厳密化

タイムアウトなどの時間に関するイベントが発生するのは、サービスコール時点から指定されたパラメータに相当する実時間が経過した後の最初のタイムティック供給(isig_tim)時点と規定

例1 : TIC_NUME=1, TIC_DENO=1の場合



例2 : TIC_NUME=10, TIC_DENO=1の場合



システム状態管理機能

システムの状態を変更/参照する機能

[S]	rot_rdq, irot_rdq	タスク優先順位の回転
[S]	get_tid, iget_tid	実行状態のタスクIDの参照
[S]	loc_cpu, iloc_cpu	CPUロック状態への移行
[S]	unl_cpu, iunl_cpu	CPUロック状態の解除
[S]	dis_dsp	ディスパッチ禁止
[S]	ena_dsp	ディスパッチ許可
[S]	***sns_ctx***	コンテキストの参照
[S]	***sns_loc***	CPUロック状態の参照
[S]	***sns_dsp***	ディスパッチ禁止状態の参照
[S]	***sns_dpn***	ディスパッチ保留状態の参照
	ref_sys	システム状態の参照

 # loc_cpu以降のサービスコールはパート2を参照

rot_rdq

[μITRON3.0] TPRI_RUN(0:現在実行中のタスクの優先度)指定の機能

↓

[μITRON4.0] TPRI_SELF(0:自タスクの優先度)指定の機能
非タスクコンテキストからのTSK_SELF指定はエラー

get_tid

[μITRON3.0] get_tidは、自タスクIDを得る機能。
非タスクコンテキストからの発行はFALSE(0)が返る

↓

[μITRON4.0] get_tidは、実行状態のタスクIDを得る機能。
非タスクコンテキストからの発行では、非タスクコンテキストに移行する前に実行していたタスクIDが返る

#非タスクコンテキストに移行する前に実行していたタスクを知りたいケースが多い

システム構成管理機能

[S]	DEF_EXC	CPU例外ハンドラの定義(静的API)
	def_exc	CPU例外ハンドラの定義
	ref_cfg	コンフィギュレーション情報の参照
	ref_ver *	バージョン情報の参照
[S]	ATT_INI	初期化ルーチンの追加(静的API)

** μITRON3.0のget_verから改称*

#CPU例外ハンドラについてはパート2を参照

ATT_INI：初期化ルーチンの追加

```
ATT_INI({ATR iniatr, VP_INT exinf, FP inirtn});
```

属性
(TA_HLNG ‖ TA_ASM)

拡張情報

初期化ルーチン
の起動番地

システムコンフィギュレーションファイル

```
    ...
ATT_INI({TA_HLNG,INF1,init1});
ATT_INI({TA_HLNG,INF2,init2});
ATT_INI({TA_HLNG,INF3,init3});
    ...
```