機能分散マルチプロセッサ向けリアルタイムカーネル仕様

作成者: 高田広章（名古屋大学／TOPPERSプロジェクト）
最終更新: 2005年5月27日

○ドキュメントの位置付け

このドキュメントは，μITRON仕様を機能分散マルチプロセッサ向けに拡張したリアルタイムカーネル仕様について記述したものである．TOPPERS/FDMPカーネルは，この仕様に準拠して実装している．

この仕様は，密結合（プロセッサ間の共有メモリがある）ではあるが，各プロセッサの役割が決まっている機能分散マルチプロセッサを対象としている．対称型マルチプロセッサ（SMP）向けのリアルタイムカーネル仕様については，別途検討が必要と考えている．また，メモリを共有しない疎結合のマルチプロセッサシステム（分散システム）については，TOPPERSプロジェクトでは「組込みコンポーネント仕様」の枠組みで扱うこととしており，この仕様の適用範囲外である．

機能分散マルチプロセッサ向けに拡張するベース仕様として，μITRON4.0仕様のスタンダードプロファイルを考える．このドキュメント中では，スタンダードプロファイル外の機能についても言及しているが，網羅的には検討していない．

なお，この仕様は，μITRON仕様を機能分散マルチプロセッサ向けに拡張する最初の試みであり，多くの検討課題が残っている．このドキュメント中で，「※」で始まる段落は検討課題を示している．これらの検討課題については，今後，TOPPERS/FDMPカーネルを適用した結果をフィードバックして検討していく計画である．

○カーネルオブジェクトのクラス分け（基本モデル）

共通の性質を持ったカーネルオブジェクト（以下，単にオブジェクトと呼ぶ）の集合を，クラスと呼ぶ．すべてのオブジェクトは，いずれか1つのクラスに所属する．

あるプロセッサに属するが，どのプロセッサからもアクセスできる（という共通の性質を持った）オブジェクトの集合を，そのプロセッサのローカルクラスと呼ぶ．ローカルクラスに所属する処理単位（タスク，周期ハンドラなど）は，それが属するプロセッサでのみ実行される．

TOPPERS/FDMPカーネルでは，プロセッサ毎に1つのローカルクラスがあるモデルのみをサポートしている．これを，機能分散マルチプロセッサ向けのリアルタイムカーネルの基本モデルと呼ぶ．基本モデルにおいては，クラスはプロセッサと1対1に対応するため，クラスという概念を導入する必然性はないが，後述の将来的な拡張を考えてクラスの概念を導入する．

基本モデルでは，ローカルクラスのみをサポートするため，オブジェクトが所属するクラスによって，オブジェクトに対するアクセス可能性に違いはない．すなわち，どのクラスに所属するオブジェクトであっても，同じようにアクセスすることができる．

また，処理単位となるオブジェクトは，それが属するプロセッサでのみ実行される．スタンダードプロファイルで，処理単位となるオブジェクトは次の通りである．この中で，タスク例外処理ルーチンは，タスクと同じクラスに所属するものと制約される．

・タスク
・タスク例外処理ルーチン
・周期ハンドラ
・割込みハンドラ
・CPU例外ハンドラ

クラスは，カーネルオブジェクトの1つであり，1から連続する正のID番号で識別する．TCLS_SELF（＝0）により，呼び出した処理単位が所属するのと同じクラスを指定することができる．

ID番号で識別されるオブジェクトは，ID番号を用いて所属するクラスを表す．具体的には，ID番号の上位ビットでクラスを表し，下位ビットでそのクラス内でのオブジェクトの識別番号を表す．TOPPERS/FDMPカーネルでは，ID番号を32ビットとし，上位16ビットをクラスID，下位16ビットをクラス内でのオブジェクトIDとしている．

スタンダードプロファイルでは，ID番号で識別されるオブジェクトは次の通りである．

・タスク
・セマフォ
・イベントフラグ
・データキュー
・メールボックス
・固定長メモリプール
・周期ハンドラ

これにより，これらのオブジェクトを操作するサービスコールは，そのままの形で機能分散マルチプロセッサに対応させることができる．

※ オブジェクトのID番号から，クラスIDを取り出すマクロと，クラス内での識別番号を取り出すマクロ，その逆にID番号を組み立てるマクロを標準化しておくべきであろう．

【将来的な拡張について（拡張モデル）】

ローカルクラス以外のクラスとしては，次のようなクラスが考えられる．

プライベートクラス: あるプロセッサに属し，そのプロセッサからのみアクセスできる（他のプロセッサからはアクセスできない）オブジェクトの集合．プライベートクラスに所属する処理単位は，それが属するプロセッサでのみ実行され，他のプロセッサのオブジェクトにアクセスできない．

グローバルクラス: 特定のプロセッサに属さず，どのプロセッサからもアクセスできるオブジェクトの集合．グローバルに所属する処理単位は，どのプロセッサでも実行することができ，プロセッサ間をマイグレートすることもある．

さらに，複雑なアーキテクチャの場合には，いくつかのプロセッサで共有するオブジェクトのクラス（そこに所属する処理単位は，それらのプロセッサでのみ実行できる）などが考えられる．

※ TOPPERS/FDMPカーネルは基本モデルのみを実装しているため，拡張モデルへの拡張性について十分に検討していない．特に，各種の資源／状態を，プロセッサ毎に持つべきかクラス毎に持つべきかを十分に考察できていない．

○プロセッサ毎に持つシステム状態

シングルプロセッサ向けのカーネルにおけるシステム状態の多くは，機能分散

マルチプロセッサにおいては，プロセッサ毎に持つことになる．具体的には，スタンダードプロファイルにおいてプロセッサ毎に持つ状態は，以下の通りである．

・実行状態のタスク
・CPUロック状態
・ディスパッチ禁止状態

この結果，ディスパッチ保留状態もプロセッサ毎の状態となる．

あるプロセッサがCPUロック状態にある間は，そのプロセッサにおいて，（カーネルの管理外の割込みを除く）すべての割込みが禁止され，ディスパッチも起こらない．それに対して，他のプロセッサにおいては割込みやディスパッチが起こるため，CPUロック状態を使って他のプロセッサで実行される処理単位との排他制御を実現することはできない．

同様に，あるプロセッサがディスパッチ禁止状態にある間は，そのプロセッサにおいてディスパッチが起こらない．他のプロセッサにおいてはディスパッチが起こるため，ディスパッチ禁止状態を使って他のプロセッサで実行されるタスクとの排他制御を実現することはできない．

システム状態の意味の変更に伴って，これらのシステム状態を変更／参照する以下の機能は，呼び出した処理単位が実行されているプロセッサの状態を変更／参照するものとする．

・get_tid／iget_tid
・loc_cpu／iloc_cpu，unl_cpu／iunl_cpu
・dis_dsp，ena_dsp
・sns_loc，sns_dsp，sns_dpn，sns_tex

それぞれの具体的な扱いは次の通り．

get_tid／iget_tid: 呼び出した処理単位が実行されているプロセッサにおいて，実行状態となっているタスク（タスクコンテキストから呼び出された場合には，自タスクに一致する）のID番号を返す．

loc_cpu／iloc_cpu，unl_cpu／iunl_cpu: 呼び出した処理単位が実行されているプロセッサを，CPUロック状態，またはCPUロック解除状態に移行する．

dis_dsp，ena_dsp: 呼び出した処理単位が実行されているプロセッサを，ディスパッチ禁止状態，またはディスパッチ許可状態に移行する．

sns_loc，sns_dsp，sns_dpn: 呼び出した処理単位が実行されているプロセッサが，CPUロック状態，ディスパッチ禁止状態，またはディスパッチ保留状態であるかを参照する．

sns_tex: 呼び出した処理単位が実行されているプロセッサにおいて，実行状態となっているタスク（タスクコンテキストから呼び出された場合には，自タスクに一致する）が，タスク例外禁止状態であるかを参照する．

なお，sns_ctxは，呼び出した処理単位の属するコンテキストを参照するものであり，プロセッサ毎に持つシステム状態を参照するサービスコールではないため，シングルプロセッサの場合と意味が変わらない．

また，他のプロセッサで実行状態となっているタスクのID番号を参照するサービスコールとして，以下のサービスコールを用意する．これらのサービスコールでは，プロセッサを指定するために，そのプロセッサのローカルクラスのID番号を渡すこととする（このためだけにプロセッサIDを導入することを避ける

ため）．

```
ER ercd = mget_tid(ID clsid, ID *p_tskid)
ER ercd = imget_tid(ID clsid, ID *p_tskid)
```

※ 他のプロセッサで実行状態となっているタスクは常に変化する可能性があるため，これらのサービスコールの用途は，デバッグ目的以外には無いようにも思われる．必要性については，今後再検討する．

○割込みハンドラとCPU例外ハンドラ

割込みハンドラとCPU例外ハンドラは，プロセッサ毎に定義する．そこで便宜上，割込みハンドラとCPU例外ハンドラは，そのプロセッサのローカルクラスに属するものと扱う（プライベートクラスをサポートする場合には，プライベートクラスに属するとする方が妥当であろう）．

割込みハンドラとCPU例外ハンドラは，（ID番号ではなく）オブジェクト番号で識別されるが，スタンダードプロファイルでは，それらを定義する静的API以外で使われないため，所属するクラスを表す方法は用意しない．

スタンダードプロファイル外では，割込みハンドラ／CPU例外ハンドラを定義するサービスコール（def_inh，def_exc）によって，他プロセッサの割込みハンドラ／CPU例外ハンドラの定義も許すことにするなら，ID番号と同様に，オブジェクト番号（この場合はハンドラ番号）の上位ビットでクラスを表す方法をとる必要がある．

○システム時刻とそれに依存して行う処理

システム時刻は，プロセッサ毎に管理する．言い換えると，プロセッサ毎にシステム時刻を持つ．カーネルは，各プロセッサのシステム時刻を同期させる機能を持たないこととする．

※ システム時刻の同期が必要な場合には，タイマ割込みをすべてのプロセッサに同期して入れるなど，ハードウェア的に実現する方が容易と思われるが，具体的な実現方法は今後の課題とする．

システム時刻は，シングルプロセッサ向けのカーネルでは，システムに唯一のオブジェクトであるため，識別する必要がない．そのため，システム時刻を操作するサービスコール（set_tim，get_tim，isig_tim）は，システム時刻の識別をパラメータに持たず，そのままでは複数のシステム時刻の内の1つを指定することができない．TOPPERS/FDMPカーネルでは，システム時刻の操作は，呼び出した処理単位が属するプロセッサのシステム時刻に対するものと限定し，他のプロセッサのシステム時刻を操作する機能は用意していない．

システム時刻に依存して行う処理は，どのシステム時刻に依存して行うか定める必要がある．スタンダードプロファイルでは，システム時刻に依存して行う処理として，以下の処理がある．

- タイムアウトに伴う待ち状態からの解除
- dly_tskによる時間経過待ち状態からの解除
- 周期ハンドラの起動

タイムアウトに伴う待ち状態と時間経過待ち状態からの解除は，待ち解除されるタスクが属するプロセッサのシステム時刻に依存して行う．周期ハンドラの起動は，周期ハンドラが属するプロセッサのシステム時刻に依存して行う．

○優先順位の回転

タスクの優先順位を回転させるサービスコール（rot_rdq，irot_rdq）は，呼び出した処理単位と同じクラスに所属するタスクの優先順位を回転する．

さらに，他のクラスに所属するタスクの優先順位を回転するために，クラスIDをパラメータに加えたサービスコール（mrot_rdq，imrot_rdq）を用意する．

```
ER ercd = mrot_rdq(ID clsid, PRI tskpri)
ER ercd = imrot_rdq(ID clsid, PRI tskpri)
```

mrot_rdqでは，tskpriにTPRI_SELFを指定することはできないものとする．

※ mrot_rdq，imrot_rdqが必要となる場面は少ないものと思われる．必要性については，今後再検討する．

○初期化ルーチン

初期化ルーチンも，静的APIにより，クラス毎に定義できるものとする．

○システム初期化手順

μITRON4.0仕様のシステム初期化手順（μITRON4.0仕様書3.7節）において，静的APIの処理とカーネルの動作開始の間に，プロセッサ間で同期を取る（他のプロセッサが静的APIの処理を終えるのを待つ）こととする．そのため，他のプロセッサのアプリケーション（タスク，割込みハンドラ等）から，静的APIの処理前ないしは処理途中の状態が見えることはない．

ただし，TOPPERS/FDMPカーネルにおいては，タスク属性にTA_ACTの指定されたタスクの起動処理と，周期ハンドラ属性にTA_STAの指定された周期ハンドラの動作開始処理は，プロセッサ間の同期後に行う．そのため，他のプロセッサのアプリケーションから，タスクの起動前や周期ハンドラ動作開始前の状態が見える可能性がある．

※ TOPPERS/FDMPカーネルの方法で不都合がないかは，今後の検討課題である．

○静的APIの拡張

すべてのオブジェクトはいずれか1つのクラスに所属することから，オブジェクトの登録などを行う静的APIは，次のようにクラスの囲みの中に記述する．

```
local_class ＜プロセッサ名＞ {
        そのプロセッサのローカルクラスに属するオブジェクトの登録など
}
```

ここで＜プロセッサ名＞には，単一の識別子を記述する．同一のクラスの囲みを複数記述した場合には，それぞれの囲みの中に記述されている静的APIをマージして扱う．オブジェクトの登録を行う静的APIを，クラスの囲みの外側に記述することはできない．INCLUDEは，クラスの囲みの内側にも外側にも記述することができる．

コンフィギュレータは，プロセッサ毎のディレクトリに，それぞれkernel_cfg.cとkernel_id.hを生成する．＜プロセッサ名＞を，プロセッサ毎のディレクトリの名称とする方法を標準とする．

クラスIDの割付けは，コンフィギュレータが行う．コンフィギュレータは，kernel_id.h（または，そこからインクルードされるファイル）に，クラスIDのマクロ定義，すべてのオブジェクトのID番号（ID番号の上位16ビットにクラスIDを入れたもの）の定義，そのプロセッサのローカルクラスに属するオブジェクトのID番号の下位16ビット（言い換えると，上位16ビットをTCLS_SELFとし

たID番号）の定義を生成する．

※ local_class指定に＜プロセッサ名＞を記述する代わりに，＜プロセッサID＞を記述することとして，整数値の記述を許す（IDの自動割付けも許す）とする手も考えられる．この場合，ディレクトリ名をどうするかは要検討．

○ディスパッチ保留状態で実行中のタスクに対する強制待ち／強制終了

機能分散マルチプロセッサでは，ディスパッチ保留状態（ディスパッチャよりも優先順位の高い処理が実行されている間，CPUロック状態の間，およびディスパッチ禁止状態の間）で実行中のタスクに対して，他のプロセッサから強制待ち（sus_tsk）または強制終了（ter_tsk）した場合の扱いが問題になる．これについては，以下のように定める．

まず，μITRON4.0仕様では，ディスパッチ保留状態において，割込みハンドラから実行中のタスクを強制待ち（sus_tsk）または強制終了（ter_tsk）した場合の扱いについて，次のように定めている（μITRON4.0仕様書3.5.6節より引用）．

また，実装独自に，非タスクコンテキストから実行中のタスクを強制待ち状態や休止状態に移行させるサービスコールを追加した場合や，ディスパッチ禁止状態で自タスクを強制待ち状態に移行させるサービスコールを呼出し可能とした場合には，ディスパッチ保留状態の間のタスク状態について次のように定める．

ディスパッチ保留状態で，実行中のタスクを強制待ち状態や休止状態へ移行させようとした場合，タスクの状態遷移はディスパッチが起こる状態となるまで保留される．状態遷移が保留されている間は，それまで実行中であったタスクは過渡的な状態にあると考え，その具体的な扱いは実装依存とする．一方，ディスパッチが起こる状態となった後に実行すべきタスクは，実行可能状態である．

そこで，機能分散マルチプロセッサにおいて，ディスパッチ保留状態で実行中のタスクに対して，他プロセッサから強制待ち（sus_tsk）または強制終了（ter_tsk）した場合も，これに準じて扱うこととする．

※ μITRON4.0仕様からは外れるが，ディスパッチ禁止状態のタスクに対する強制終了（ter_tsk）は，強制終了を保留せずに，すぐに実行してしまう手もある．強制終了が，異常時の回復のためのものと考えれば，この仕様も妥当性がある．そうする場合でも，強制待ち（sus_tsk）は保留させるのが妥当であろう．

標準仕様という立場であれば，これ以上の具体的な扱いは実装定義でもよいと考えられるが，少なくともTOPPERS/FDMPカーネルではどのように取り扱うか定めておく必要がある．以下では，TOPPERS/FDMPカーネルにおける取扱いを述べる．

まず，ディスパッチ保留状態で実行中のタスクに対して，他のプロセッサから強制待ち（sus_tsk）した場合には，対象タスクは実行状態と強制待ち状態の中間的な状態となるものとする．この状態を，「強制待ち状態（実行継続中）」と呼ぶ．

また，ディスパッチ保留状態で実行中のタスクに対して，他のプロセッサから強制終了（ter_tsk）した場合には，対象タスクは実行状態と休止状態の中間的な状態となるものとする．この状態を，「休止状態（実行継続中）」と呼ぶ．さらに，休止状態（実行継続中）のタスクを起動した場合「実行可能状態（前回実行継続中）」に，それに対して強制待ち（sus_tsk）した場合「強制待ち

状態（前回実行継続中）」に遷移するものとする．

タスクがこれらの状態にある時に，（ディスパッチャよりも優先順位の高い処理からの復帰，CPUロック解除，またはディスパッチ許可により）ディスパッチできる状態になると，保留されていたタスク状態遷移が実行される．

具体的には，強制待ち状態（実行継続中）のタスクがディスパッチできる状態になると，即座にタスクディスパッチが発生し，当該タスクは強制待ち状態（実行開始後）に遷移する．また，休止状態（実行継続中）のタスクがディスパッチできる状態になると，当該タスクは即座に終了させられ，休止状態となる．

実行可能状態（前回実行継続中）または強制待ち状態（前回実行継続中）のタスクがディスパッチできる状態になると，現在実行中のタスクは即座に終了させられ，再起動後のタスクがそれぞれの状態になる．より具体的には，ディスパッチできる状態になると即座にタスクディスパッチが発生するが，その際に，それまで実行していたタスクのコンテキストは破棄される．

これらの中間的な状態のタスクの具体的な扱いは次の通りである．

(1) コンテキストを占めて実行を継続する

これらの状態のタスクは，プロセッサのコンテキストを占めて，そのまま継続して実行される．

(2) そのプロセッサにおいて実行状態のタスクであると扱う

これらの状態のタスクは，それを実行するプロセッサにおいて，実行状態のタスクであると扱う．すなわち，これらの状態のタスクが実行している時にget_tid／iget_tid／mget_tid／imget_tidを発行すると，実行状態のタスクとして，そのタスクのID番号が返る．sns_texについては後述する．

(3) 他に定める場合（(2),(4),(5)）を除いては，括弧外の状態（例えば，「強制待ち状態（実行継続中）」なら「強制待ち状態」）と扱う

例えば，強制待ち状態（実行継続中）のタスクに対して，再度強制待ち（sus_tsk）を行うと強制待ち要求ネスト数のオーバフローとなり（ネスト数の最大値が1の場合），再開（rsm_tsk）を行うと実行状態（ディスパッチ保留状態）に戻る．また，強制終了（ter_tsk）した場合には，休止状態（実行継続中）に移行する．

また，強制待ち状態（前回実行継続中）のタスクに対して，再度強制待ち（sus_tsk）を行うと強制待ち要求ネスト数のオーバフローとなり（ネスト数の最大値が1の場合），再開（rsm_tsk）を行うと実行可能状態（前回実行継続中）に戻る．また，強制終了（ter_tsk）した場合には，休止状態（実行継続中）に移行する．

ここで，強制待ち状態（実行継続中）または強制待ち状態（前回実行継続中）のタスクが，自タスクを指定して強制待ち状態からの再開（rsm_tsk）を行った場合でも，上記の通りのタスク状態遷移が発生し，エラーとはならない．これは，μITRON4.0仕様において強制待ち状態のタスクが自タスクを指定して再開（rsm_tsk）した場合の振舞いを実装定義としているのと整合する．

休止状態（実行継続中），強制待ち状態（実行継続中），強制待ち状態（前回実行継続中）のタスクの優先度を指定して，rot_rdqで優先順位を回転させる場合，当該タスクは回転の対象にはならない．それに対して，実行状態（ディスパッチ保留状態），実行可能状態（前回実行継続中）のタスクは，回転の対象となる．

ここで注意を要するのは，休止状態（実行継続中）のタスクが，自タスクを指定して呼び出すサービスコールにおいても，自タスクが休止状態と見なされる点である．例えば，自タスクに対するタスク例外処理の要求（ras_tex）や自タスクの優先度の変更（chg_pri）は，自タスクが休止状態と見なされるためにE_OBJエラーとなる．

さらに，この規定に従うと，実行可能状態（前回実行継続中）または強制待ち状態（前回実行継続中）のタスクが，自タスクを指定して呼び出すサービスコールは，自タスクの次回実行に対する操作と見なされる．例えば，自タスクに対するタスク例外処理の要求（ras_tex）や自タスクの優先度の変更（chg_pri）は，自タスクの次回実行に対して有効となる．これにより，アプリケーションの作成が難しくなる可能性が考えられるため，次の例外規定を設ける．

(4) 実行可能状態（前回実行継続中）または強制待ち状態（前回実行継続中）のタスクが，自タスクを指定して呼び出すサービスコールにおいては，自タスクが休止状態であるとして扱う．can_actについては，自タスクを指定して呼び出した場合には，E_OBJエラーとする

スタンダードプロファイルにおいて，タスクIDをパラメータとするタスクコンテキスト用のサービスコールに対して，実行可能状態（前回実行継続中）または強制待ち状態（前回実行継続中）のタスクが自タスクを対象に呼び出した場合の具体的な振舞いは次の通りとする．

- act_tsk ... 特別な対処はしない（起動要求がキューイングされる）
- can_act ... E_OBJエラーにする（例外扱い．下で説明する）
- ter_tsk ... 特別な対処はしない（次回実行が強制終了される）
- chg_pri ... E_OBJエラーにする
- get_pri ... E_OBJエラーにする
- wup_tsk ... E_OBJエラーにする
- can_wup ... E_OBJエラーにする
- rel_wai ... 特別な対処はしない（E_OBJエラーになる）
- sus_tsk ... 特別な対処はしない（E_CTXになる）
- rsm_tsk ... E_OBJエラーにする
- frsm_tsk ... E_OBJエラーにする
- ras_tex ... E_OBJエラーにする

can_actは，休止状態のタスクに対しても発行できるサービスコールであり，E_OBJエラーが返ることのないサービスコールである．自タスクに対するcan_actは，自タスクが終了後に再起動されるのを防ぐことになるが，実行可能状態（前回実行継続中）または強制待ち状態（前回実行継続中）のタスクが発行した場合には，すでに次回の起動処理が行われた後で，キャンセルすることができない．そこで，E_OBJエラーを返して，キャンセルできないことをアプリケーションに通知する．

このようなケースは，タスクが他のプロセッサから強制終了される場合で，サービスコールをディスパッチ禁止状態で呼び出す場合のみに発生する．これに該当するケースでは，上記のサービスコールからE_OBJエラーが返ってくる場合に対処しなければならない．

(5) 休止状態（実行継続中），実行可能状態（前回実行継続中），強制待ち状態（前回実行継続中）の3状態においては，タスク例外処理ルーチンの起動を禁止する

この仕様は，主に実装上の都合によるものであるが，これらの3状態はタスクが強制終了されようとしている状態であり，タスク例外処理ルーチンが起動されないとして差し支えはないものと思われる．

具体的には，他のプロセッサから強制終了された時点で，タスク例外処理禁止状態に移行し，タスク例外処理の許可を許さないこととする．すなわち，タスクがこの3状態にある時にsns_texを呼び出すと，TRUEが返る．また，タスク例外処理の許可（ena_tex）を呼び出すと，E_OBJエラーを返すものとする．タスク例外処理の禁止（dis_tex）は許しても差し支えないが，整合性を考えて，E_OBJエラーを返すものとする．

(6) ref_tsk／ref_tstでどちらと扱うかは要検討

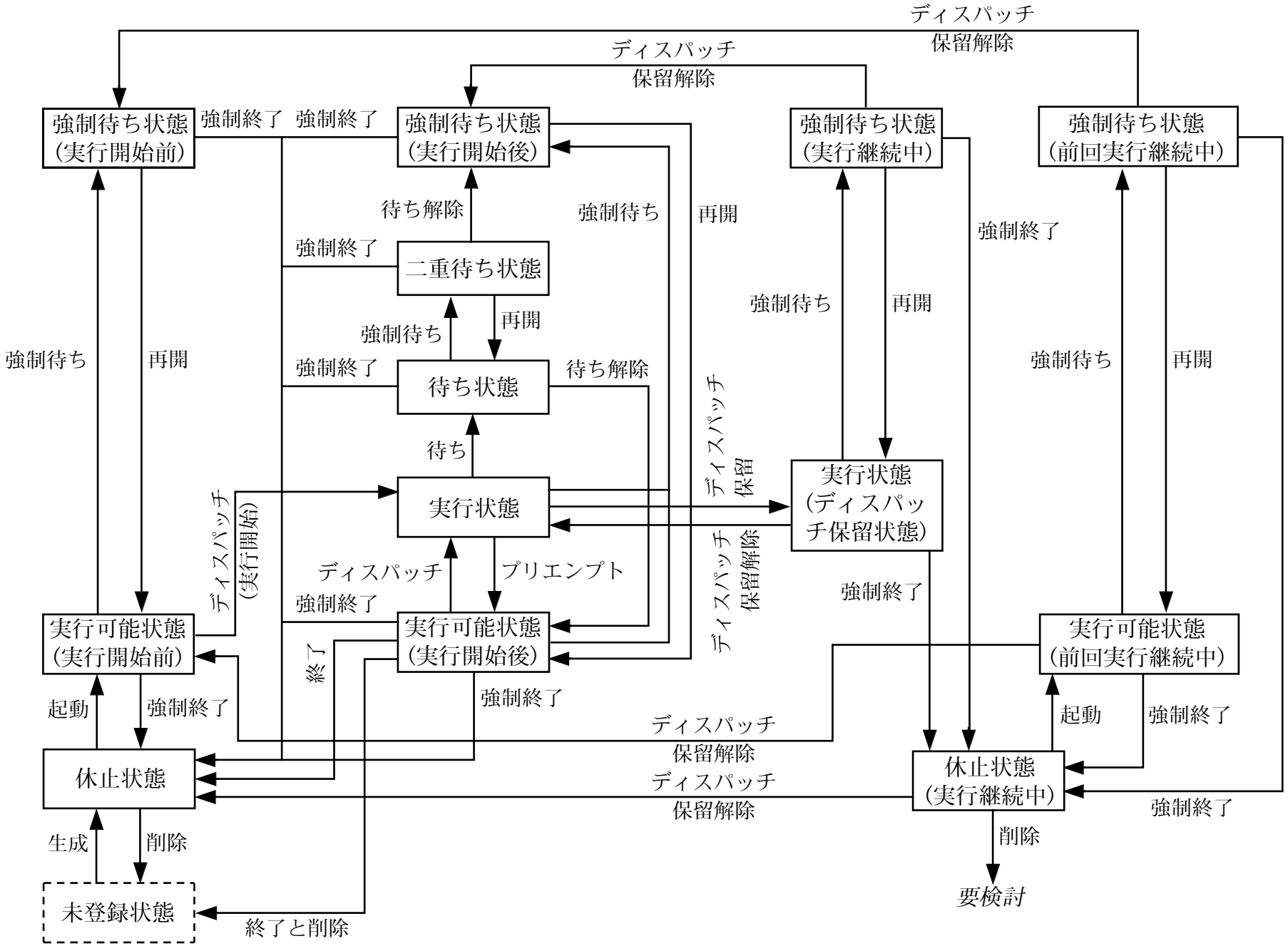
これらの状態のタスクをref_tsk／ref_tstで参照した場合に，タスク状態として何を返すか決める必要がある．スタンダードプロファイル外の機能なので，現時点では決定を保留する．

(7) 休止状態（実行継続中）のタスクを削除した場合の扱いも要検討

スタンダードプロファイル外の機能では，休止状態（実行継続中）のタスクを削除した場合にどうするかという問題が残っている．未登録状態においては，TCBのみならずスタック領域まで解放されるため，ディスパッチできる状態となるまで削除を遅延するのが妥当であると思われるが，実装が難しくなる．スタンダードプロファイル外の機能なので，現時点では決定を保留する．

※ 「次回実行」「再起動後」という用語を厳密に定義しておらず，曖昧な記述になっていると思われる．厳密に規定するには，「タスクのインスタンス」という概念を導入する必要があるだろう．

以上



※ ディスパッチ保留：CPUロックまたはディスパッチ禁止
ディスパッチ保留解除：CPUロック解除状態でディスパッチ許可，
またはディスパッチ許可状態でCPUロック解除