T-Kernel 仕様書

T-Kernel 1.00.01
2010 年 10 月

T-Kernel 仕様書 (Ver.1.00.01)

本仕様書に関しては、下記にお問い合わせください。
T-Engine フォーラム事務局
〒141-0031　東京都品川区西五反田 2-20-1　第 28 興和ビル
YRP ユビキタス・ネットワーキング研究所内
TEL：03-5437-0572　　FAX：03-5437-2399
E-mail：office@t-engine.org

図版目次

システムコールの記述形式

本仕様書のシステムコール説明の部分では、システムコールごとに、以下のような形式で仕様の説明を行っている。

システムコール名称　　説明

-システムコール名称: フルスペル-

【C 言語インタフェース】
-システムコールの C 言語インタフェースを示す-

【パラメータ】
-システムコールのパラメータ、すなわちシステムコールを発行するときに OS に渡す情報に関する説明を行う-

【リターンパラメータ】
-システムコールのリターンパラメータ、すなわちシステムコールの実行が終ったときに OS から返される情報に関する説明を行う-

【エラーコード】
- システムコールで発生する可能性のあるエラーに関して説明を行う -
※以下のエラーコードについては、各システムコールのエラーコードの説明の項には含めていないが、各システムコールにおいて共通に発生する可能性がある。
E_SYS, E_NOSPT, E_RSFN, E_MACV, E_OACV
※エラーコード E_CTX については、このエラーの発生条件が明確な場合にのみ(待ち状態に入るシステムコールなど)、各システムコールのエラーコードの説明に含めている。しかし、実装の制限により、それ以外のシステムコールにおいても E_CTX のエラーが発生する可能性がある。実装依存で発生する E_CTX については、各システムコールのエラーコードの説明の項には含めていない。

【解説】
- システムコールの機能の解説を行う -
※いくつかの値を選択して設定するようなパラメータの場合には、以下のような記述方法によって仕様説明を行っている。

(x || y || z)	-	x, y, z のいずれか一つを選択して指定する。
x | y	-	x と y を同時に指定可能である。 (同時に指定する場合は x と y の論理和をとる)
[x]	-	x は指定しても指定しなくても良い。

例:
wfmode := (TWF_ANDW || TWF_ORW) | [TWF_CLR] の場合、
wfmode の指定は次の 4 種のいずれかになる。

```
TWF_ANDW
TWF_ORW
(TWF_ANDW | TWF_CLR)
(TWF_ORW | TWF_CLR)
```

【補足事項】
-特記事項や注意すべき点など、解説に対する補足事項を述べる-

【仕様決定の理由】
- 仕様決定の理由を述べる -

T-Kernel/OS システムコール索引

この索引は、本仕様書で説明される T-Kernel/OS のシステムコールのアルファベット順索引である。

T-Kernel/SM 拡張 SVC・ライブラリ索引

この索引は、本仕様書で説明される T-Kernel/SM の T-Kernel/SM 拡張 SVC・ライブラリのアルファベット順索引である。

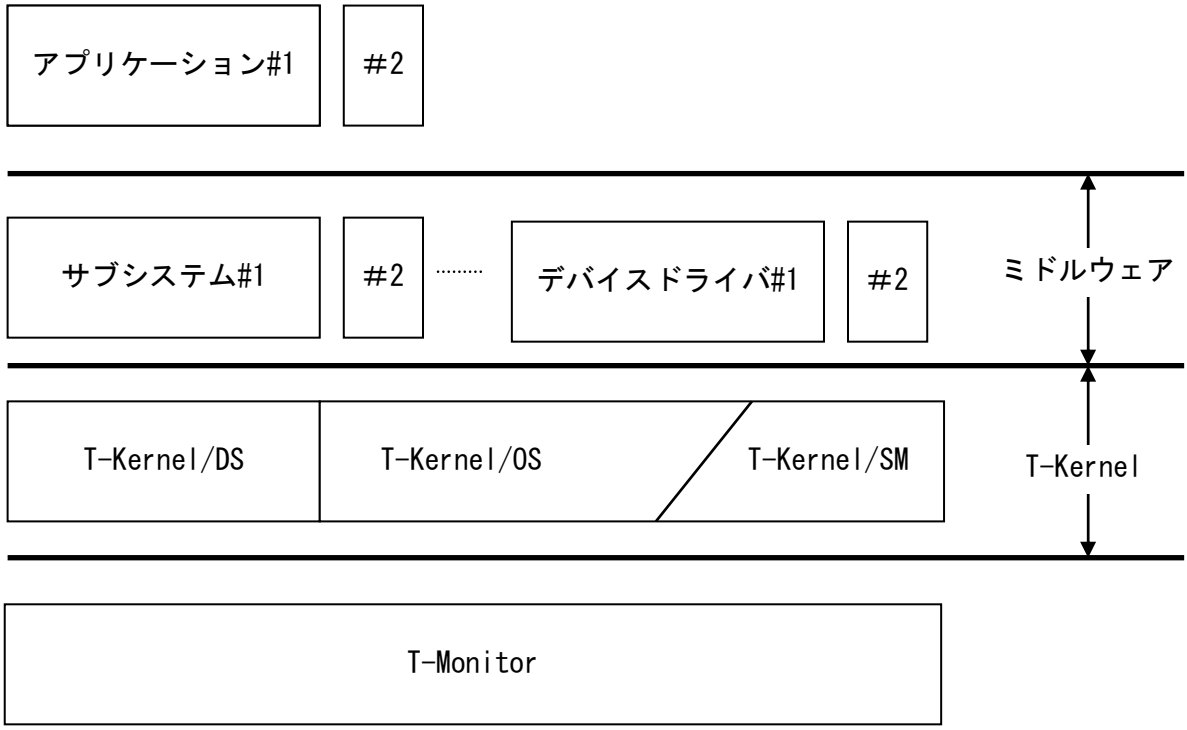
T-Kernel/DS システムコール索引

この索引は、本仕様書で説明される T-Kernel/DS のシステムコールのアルファベット順索引である。

第 1 章　T-Kernel の概要

1.1 全体的な位置付け

T-Kernel の T-Engine システム全体における位置付けを[図 1]に示す。



[図 1] T-Kernel の位置付け

T-Kernel は、T-Kernel/OperatingSystem(T-Kernel/OS)、T-Kernel/SystemManager(T-Kernel/SM)、および T-Kernel/DebuggerSupport(T-Kernel/DS)を含む全体を指すが、T-Kernel/OS のみを(狭義の)T-Kernel と呼ぶ場合もある。

T-Kernel/OperatingSystem(T-Kernel/OS)は以下のような機能を提供する。

- タスク制御機能
- タスク間同期通信機能
- メモリ管理機能
- 例外／割込み制御機能
- 時間管理機能
- サブシステム管理機能

T-Kernel/SystemManager(T-Kernel/SM)は以下のような機能を提供する。

- システムメモリ管理機能
- アドレス空間管理機能
- デバイス管理機能
- 割込み管理機能
- I/O ポートアクセスサポート機能
- 省電力機能
- システム構成情報管理機能

T-Kernel/DebuggerSupport(T-Kernel/DS)はデバッガ専用に以下のような機能を提供する。

- カーネルの内部状態の参照
- 実行のトレース

1.2 適応化

T-Kernel は、組み込みシステム用のリアルタイムカーネルとして、小規模から大規模のさまざまなシステムをターゲットとしており、デバイスドライバやミドルウェアの流通性の向上をねらっている。

T-Kernel は規模の大きなシステムにも対応できるように仕様が策定されている。そのため、小規模なシステム向けには必須とはされない機能も含まれているが、サブセット仕様を定めてしまうと、デバイスドライバやミドルウェア等の流通性や移植性を妨げることになる。また、ターゲットによって必要な機能はさまざまであり、一様にサブセット仕様を定めることも難しい。

T-Kernel では、レベル分けなどのサブセット化のための仕様は定めない。原則として、すべての T-Kernel 仕様 OS はすべての仕様を実装しなければならない。ただし、ターゲットシステムにおいてハードウェア上の制約で実現不可能な機能は、簡易な実装として構わない。

「簡易な実装」とは、仕様通りの機能は持っていないが、その機能を呼び出すことで動作が異常になるようなことがない実装ということができる。つまり、大規模のシステムをターゲットとして作成されたミドルウェアが、そのまま実行できるような環境を提供することが重要となる。例えば、MMU が使用できないシステムでは、T-Kernel/SM の LockSpace()は次のように実装しても構わない。

```
#define LockSpace(addr, len)    ( E_OK )
```

しかし、MMU が使用できないシステムだからといって、LockSpace()を実装しなかったり、E_NOSPT を返すような実装としてはいけない。

逆に、ミドルウェアの作成においても、MMU を使用できないシステムだからといって、LockSpace() を使用しない実装をしてしまうと、MMU を使用したシステムには対応できなくなってしまう。

T-Kernel をターゲットシステムに実装する際に、必要とされない機能を省いたり、機能を変更することは構わない。ただし、その T-Kernel は、改変された T-Kernel とみなされる。

ミドルウェア提供者は、以下の点に注意しなくてはならない。

- T-Kernel が提供しているすべての要件を満たすように、ミドルウェアは作成されなければならない。つまり、特定のターゲットシステムに限定するのではなく、さまざまな規模のシステムにも対応できる必要がある。
- 利用者が不要な機能を省けるような仕組を提供してもよい。

第 2 章　T-Kernel 仕様の概念

2.1 基本的な用語の意味

(1) タスクと自タスク

プログラムの並行実行の単位を「タスク」と呼ぶ。すなわち、一つのタスク中のプログラムは逐次的に実行されるのに対して、異なるタスクのプログラムは並行して実行が行われる。ただし、並行して実行が行われるというのは、アプリケーションから見た概念的な動作であり、実装上はカーネルの制御のもとで、それぞれのタスクが時分割で実行される。

また、システムコールを呼出したタスクを「自タスク」と呼ぶ。

(2) ディスパッチとディスパッチャ

プロセッサが実行するタスクを切り替えることを「ディスパッチ」(または「タスクディスパッチ」)と呼ぶ。また、ディスパッチを実現するカーネル内の機構を「ディスパッチャ」(または「タスクディスパッチャ」)と呼ぶ。

(3) スケジューリングとスケジューラ

次に実行すべきタスクを決定する処理を「スケジューリング」(または「タスクスケジューリング」)と呼ぶ。また、スケジューリングを実現するカーネル内の機構を「スケジューラ」(または「タスクスケジューラ」)と呼ぶ。スケジューラは、一般的な実装では、システムコール処理の中やディスパッチャの中で実現される。

(4) コンテキスト

一般に、プログラムの実行される環境を「コンテキスト」と呼ぶ。コンテキストが同じというためには、少なくとも、プロセッサの動作モードが同一で、用いているスタック空間が同一(スタック領域が一連)でなければならない。ただし、コンテキストはアプリケーションから見た概念であり、独立したコンテキストで実行すべき処理であっても、実装上は同一のプロセッサ動作モードで同一のスタック空間で実行されることもある。

(5) 優先順位

処理が実行される順序を決める順序関係を「優先順位」と呼ぶ。優先順位の低い処理を実行中に、優先順位の高い処理が実行できる状態になった場合、優先順位の高い処理を先に実行するのが原則である。

【補足事項】

優先度は、タスクやメッセージの処理順序を制御するために、アプリケーションによって与えられるパラメータである。それに対して優先順位は、仕様中で処理の実行順序を明確にするために用いる概念である。

タスク間の優先順位は、タスクの優先度に基づいて定められる。

2.2 タスク状態とスケジューリング規則

2.2.1 タスク状態

タスク状態は、大きく次の 5 つに分類される。この内、広義の待ち状態は、さらに 3 つの状態に分類される。また、実行状態と実行可能状態を総称して、実行できる状態と呼ぶ。

(a) 実行状態 (RUNNING)

現在そのタスクを実行中であるという状態。ただし、タスク独立部を実行している間は、別に規定されている場合を除いて、タスク独立部の実行を開始する前に実行していたタスクが実行状態であるものとする。

(b) 実行可能状態 (READY)

そのタスクを実行する準備は整っているが、そのタスクよりも優先順位の高いタスクが実行中であるために、そのタスクを実行できない状態。言い換えると、実行できる状態のタスクの中で最高の優先順位になればいつでも実行できる状態。

(c) 広義の待ち状態
　そのタスクを実行できる条件が整わないために、実行ができない状態。言い換えると、何らかの条件が満たされるのを待っている状態。タスクが広義の待ち状態にある間、プログラムカウンタやレジスタなどのプログラムの実行状態を表現する情報は保存されている。タスクを広義の待ち状態から実行再開する時には、プログラムカウンタやレジスタなどを広義の待ち状態になる直前の値に戻す。広義の待ち状態は、さらに次の 3 つの状態に分類される。

(c.1) 待ち状態 (WAITING)
　何らかの条件が整うまで自タスクの実行を中断するシステムコールを呼出したことにより、実行が中断された状態。

(c.2) 強制待ち状態 (SUSPENDED)
　他のタスクによって、強制的に実行を中断させられた状態。

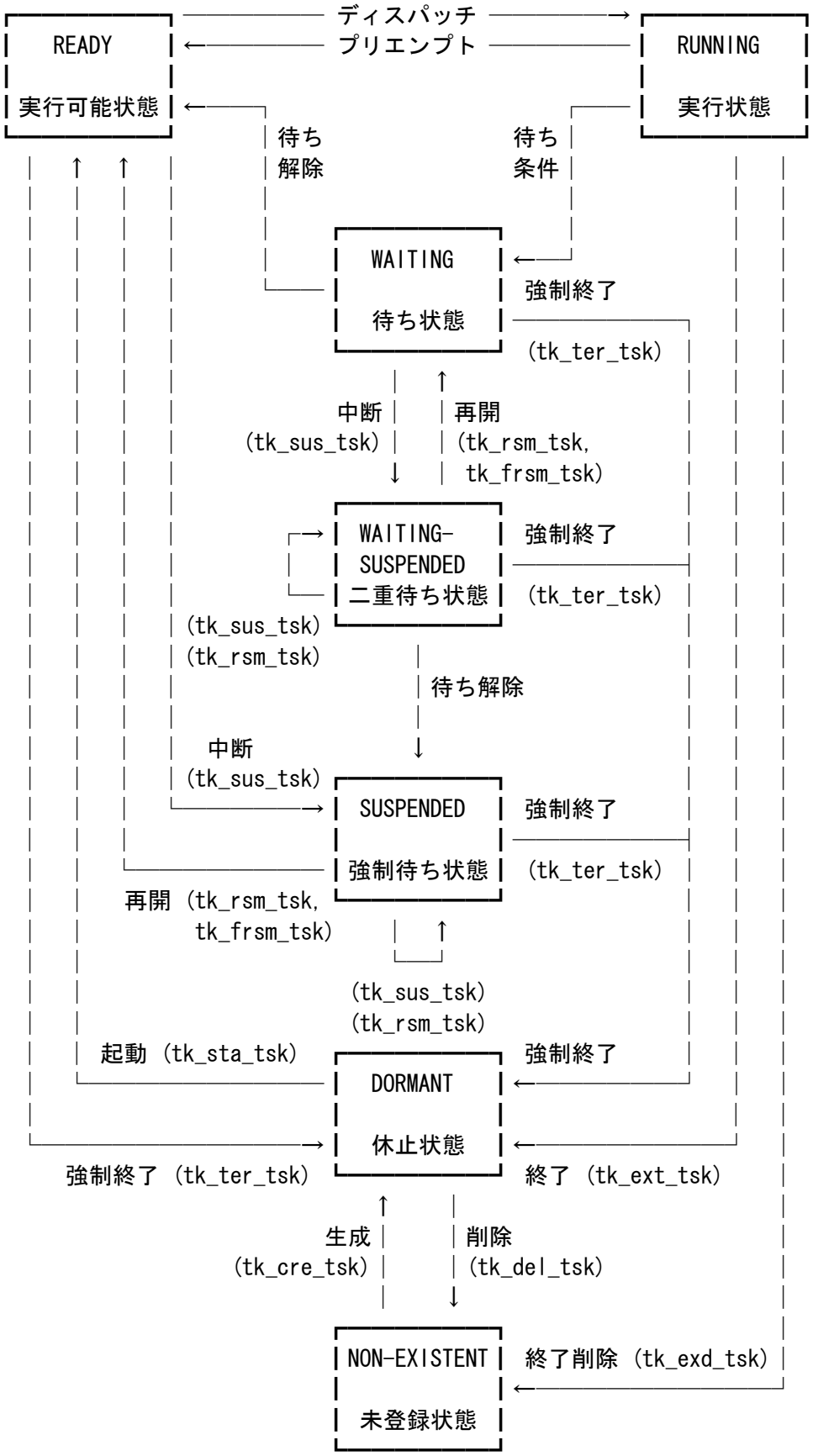
(c.3) 二重待ち状態 (WAITING-SUSPENDED)
　待ち状態と強制待ち状態が重なった状態。待ち状態にあるタスクに対して、強制待ち状態への移行が要求されると、二重待ち状態に移行させる。
　T-Kernel では「待ち状態(WAITING)」と「強制待ち状態(SUSPENDED)」を明確に区別しており、タスクが自ら強制待ち状態(SUSPENDED)になることはできない。

(d) 休止状態 (DORMANT)
　タスクがまだ起動されていないか、実行を終了した後の状態。タスクが休止状態にある間は、実行状態を表現する情報は保存されていない。タスクを休止状態から起動する時には、タスクの起動番地から実行を開始する。また、別に規定されている場合を除いて、レジスタの内容は保証されない。

(e) 未登録状態 (NON-EXISTENT)
　タスクがまだ生成されていないか、削除された後の、システムに登録されていない仮想的な状態。

実装によっては、以上のいずれにも分類されない過渡的な状態が存在する場合がある(2.5 節参照)。

実行可能状態に移行したタスクが、現在実行中のタスクよりも高い優先順位を持つ場合には、実行可能状態への移行と同時にディスパッチが起こり、即座に実行状態へ移行する場合がある。この場合、それまで実行状態であったタスクは、新たに実行状態へ移行したタスクにプリエンプトされたという。また、システムコールの機能説明などで、「実行可能状態に移行させる」と記述されている場合でも、タスクの優先順位によっては、即座に実行状態に移行させる場合もある。
タスクの起動とは、休止状態のタスクを実行可能状態に移行させることをいう。このことから、休止状態と未登録状態以外の状態を総称して、起動された状態と呼ぶことがある。タスクの終了とは、起動された状態のタスクを休止状態に移行させることをいう。
タスクの待ち解除とは、タスクが待ち状態の時は実行可能状態に、二重待ち状態の時は強制待ち状態に移行させることをいう。また、タスクの強制待ちからの再開とは、タスクが強制待ち状態の時は実行可能状態に、二重待ち状態の時は待ち状態に移行させることをいう。
一般的な実装におけるタスクの状態遷移を[図 2]に示す。実装によっては、この図にない状態遷移を行う場合がある。



[図 2] タスク状態遷移図

T-Kernel の特色として、自タスクの操作をするシステムコールと他タスクの操作をするシステムコールを明確に分離しているということがある[表 1]。これは、タスクの状態遷移を明確にし、システムコールの理解を容易にするためである。自タスク操作と他タスク操作のシステムコールを分離しているということは、言い換えると、実行状態からの状態遷移とそれ以外の状態からの状態遷移を明確に分離しているという意味にもなる。

［表 1］自タスク、他タスクの区別と状態遷移図

	自タスクに対する操作 (実行状態からの遷移)	他タスクに対する操作 (実行状態以外からの遷移)
タスクの待ち状態 (強制待ち状態を含む) への移行	tk_slp_tsk 実行状態 ↓ 待ち状態	tk_sus_tsk 実行可能、待ち状態 ↓ 強制待ち、二重待ち状態
タスクの終了	tk_ext_tsk 実行状態 ↓ 休止状態	tk_ter_tsk 実行可能、待ち状態 ↓ 休止状態
タスクの削除	tk_exd_tsk 実行状態 ↓ 未登録状態	tk_del_tsk 休止状態 ↓ 未登録状態

【補足事項】

待ち状態と強制待ち状態は直交関係にあり、強制待ち状態への移行の要求は、タスクの待ち解除条件には影響を与えない。言い換えると、タスクが待ち状態にあるか二重待ち状態にあるかで、タスクの待ち解除条件は変化しない。そのため、資源獲得のための待ち状態(セマフォ資源の獲得待ち状態やメモリブロックの獲得待ち状態など)にあるタスクに強制待ち状態への移行が要求され、二重待ち状態になった場合にも、強制待ち状態への移行が要求されなかった場合と同じ条件で資源の割付け(セマフォ資源やメモリブロックの割付けなど)が行われる。

【仕様決定の理由】

T-Kernel 仕様において待ち状態(自タスクによる待ち)と強制待ち状態(他のタスクによる待ち)を区別しているのは、それらが重なる場合があるためである。それらが重なった状態を二重待ち状態として区別することで、タスクの状態遷移が明確になり、システムコールの理解が容易になる。それに対して、待ち状態のタスクはシステムコールを呼び出せないため、複数の種類の待ち状態(例えば、起床待ち状態とセマフォ資源の獲得待ち状態)が重なることはない。T-Kernel 仕様では、他のタスクによる待ちには一つの種類(強制待ち状態)しかないため、強制待ち状態が重なった状況を強制待ち要求のネストと扱うことで、タスクの状態遷移を明確にしている。

2.2.2 タスクのスケジューリング規則

T-Kernel 仕様においては、タスクに与えられた優先度に基づくプリエンプティブな優先度ベーススケジューリング方式を採用している。同じ優先度を持つタスク間では、FCFS (First Come First Served)方式によりスケジューリングを行う。具体的には、タスクのスケジューリング規則はタスク間の優先順位を用いて、タスク間の優先順位はタスクの優先度によって、それぞれ次のように規定される。実行できるタスクが複数ある場合には、その中で最も優先順位の高いタスクが実行状態となり、他は実行可能状態となる。タスク間の優先順位は、異なる優先度を持

つタスク間では、高い優先度を持つタスクの方が高い優先順位を持つ。同じ優先度を持つタスク間では、先に実行できる状態（実行状態または実行可能状態)になったタスクの方が高い優先順位を持つ。ただし、システムコールの呼出しにより、同じ優先度を持つタスク間の優先順位が変更される場合がある。

最も高い優先順位を持つタスクが替わった場合には、ただちにディスパッチが起こり、実行状態のタスクが切り替わる。ただし、ディスパッチが起こらない状態になっている場合には、実行状態のタスクの切替えは、ディスパッチが起こる状態となるまで保留される。

【補足事項】

T-Kernel 仕様のスケジューリング規則では、優先順位の高いタスクが実行できる状態にある限り、それより優先順位の低いタスクは全く実行されない。すなわち、最も高い優先順位を持つタスクが待ち状態に入るなどの理由で実行できない状態とならない限り、他のタスクは全く実行されない。この点で、複数のタスクを公平に実行しようという TSS (Time Sharing System)のスケジューリング方式とは根本的に異なっている。

ただし、同じ優先度を持つタスク間の優先順位は、システムコールを用いて変更することが可能である。アプリケーションがそのようなシステムコールを用いて、TSS における代表的なスケジューリング方式であるラウンドロビン方式を実現することができる。

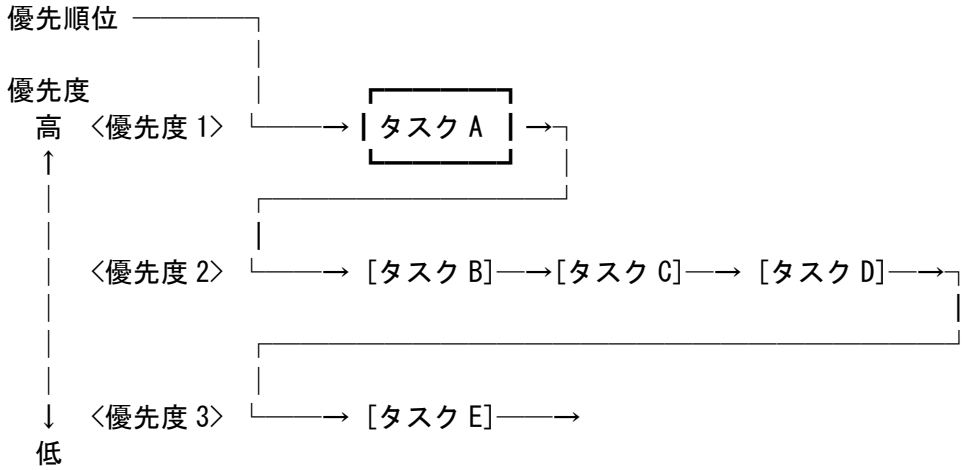
同じ優先度を持つタスク間では、先に実行できる状態(実行状態または実行可能状態)になったタスクの方が高い優先順位を持つことを、[図 3]の例を用いて説明する。[図 3(a)]は、優先度 1 のタスク A、優先度 3 のタスク E、優先度 2 のタスク B、タスク C、タスク D がこの順序で起動された後のタスク間の優先順位を示す。この状態では、最も優先順位の高いタスク A が実行状態となっている。

ここでタスク A が終了すると、次に優先順位の高いタスク B を実行状態に遷移させる[図 3(b)]。その後タスク A が再び起動されると、タスク B はプリエンプトされて実行可能状態に戻るが、この時タスク B は、タスク C とタスク D のいずれよりも先に実行できる状態になっていたことから、同じ優先度を持つタスクの中で最高の優先順位を持つことになる。すなわち、タスク間の優先順位は[図 3(a)]の状態に戻る。

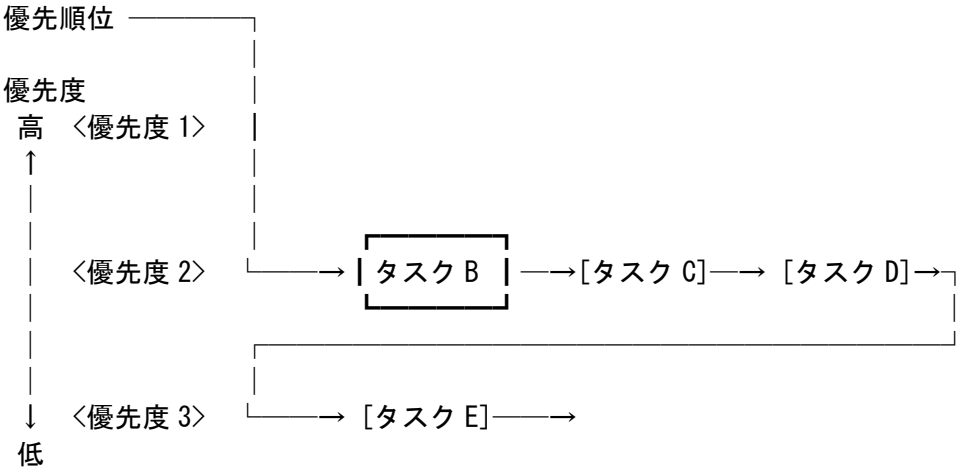
次に、[図 3(b)]の状態でタスク B が待ち状態になった場合を考える。タスクの優先順位は実行できるタスクの間で定義されるため、タスク間の優先順位は[図 3(c)]の状態となる。その後タスク B が待ち解除されると、タスク B はタスク C とタスク D のいずれよりも後に実行できる状態になったことから、同じ優先度を持つタスクの中で最低の優先順位となる[図 3(d)]。

以上を整理すると、実行可能状態のタスクが実行状態になった後に実行可能状態に戻った直後には、同じ優先度を持つタスクの中で最高の優先順位を持っているのに対して、実行状態のタスクが待ち状態になった後に待ち解除されて実行できる状態になった直後には、同じ優先度を持つタスクの中で最低の優先順位となる。

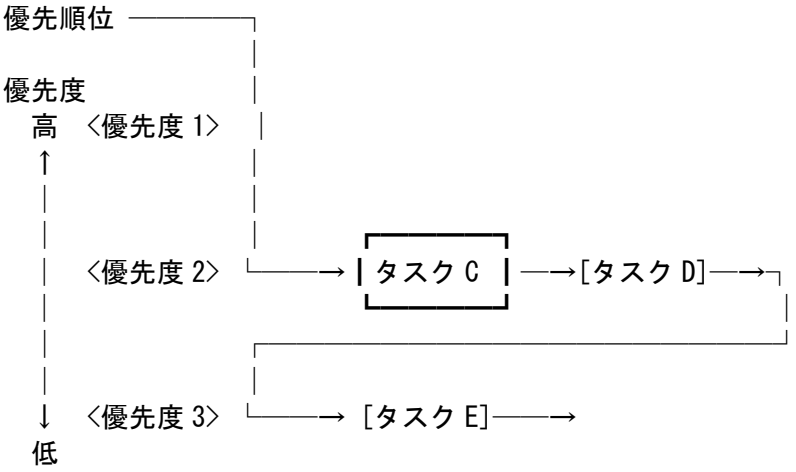
なお、タスクが強制待ち状態(SUSPENDED)から実行できる状態になった直後にも、同じ優先度を持つタスクの中で最低の優先順位となる。仮想記憶システムにおいては、ページイン待ちを強制待ち(SUSPENDED)によって行うため、このようなシステムにおいては、ページイン待ちによってタスクの優先順位が変化する。



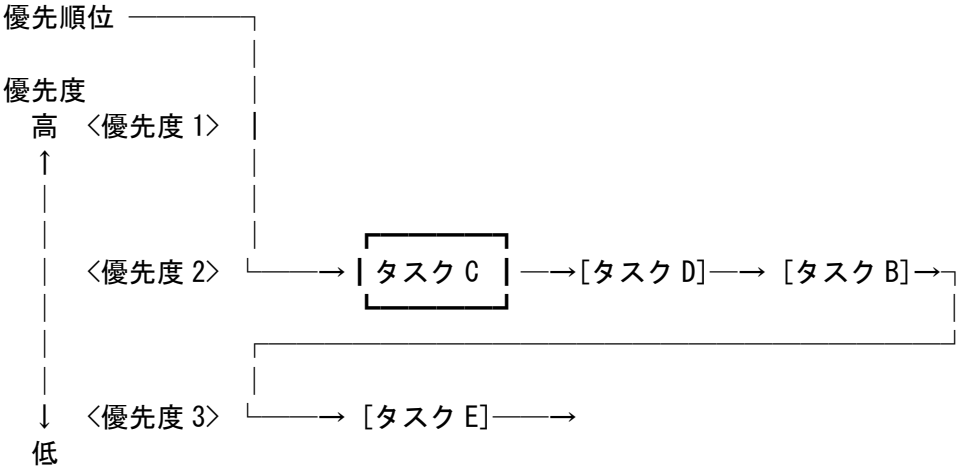
[図 3(a)] 最初の状態の優先順位



[図 3(b)] タスク B が実行状態になった後の優先順位



[図 3(c)] タスク B が待ち状態になった後の優先順位



[図 3(d)] タスク B が待ち解除された後の優先順位

2.3 割込み処理

T-Kernel 仕様では、割込みは、デバイスからの外部割込みと、CPU 例外による割込みの両方を含む。一つの割込みハンドラ番号に対して一つの割込みハンドラを定義できる。割込みハンドラの記述方法としては、基本的に OS が介入せずに直接起動する方法と、高級言語対応ルーチンを経由して起動する方法の 2 通りがある。

2.4 タスク例外処理

T-Kernel 仕様では、例外処理のための機能として、タスク例外処理の機能を規定する。なお CPU 例外については、割込みとして扱うこととする。

タスク例外処理機能は、タスクを指定してタスク例外処理を要求するシステムコールを呼び出すことで、指定したタスクに実行中の処理を中断させ、タスク例外ハンドラを実行させるための機能である。タスク例外ハンドラの実行は、タスクと同じコンテキストで行われる。タスク例外ハンドラからリターンすると、中断された処理の実行が継続される。

タスク毎に一つのタスク例外ハンドラを、アプリケーションで登録することができる。

2.5 システム状態

2.5.1 非タスク部実行中のシステム状態

T-Kernel の上で動くタスクのプログラミングを行う場合には、タスク状態遷移図を見て、各タスクの状態の変化を追っていけばよい。しかし、割込みハンドラや拡張 SVC ハンドラなど、タスクより核に近いレベルのプログラミングもユーザが行う。この場合は、非タスク部、すなわちタスク以外の部分を実行している間のシステム状態についても考慮しておかないと、正しいプログラミングができない。ここでは、T-Kernel のシステム状態について説明を行う。

システム状態は、[図 4]のように分類される。

[図 4]で示されている状態のうち、「過渡的な状態」は、OS 実行中(システムコール実行中)の状態に相当する。ユーザから見ると、ユーザの発行したそれぞれのシステムコールが不可分に実行されるということが重要なのであり、システムコール実行中の内部状態はユーザからは見えない。OS 実行中の状態を「過渡的な状態」と考え、その内部をブラックボックス的に扱うのは、こういった理由による。

しかし、次の場合、過渡的な状態が不可分に実行されない。

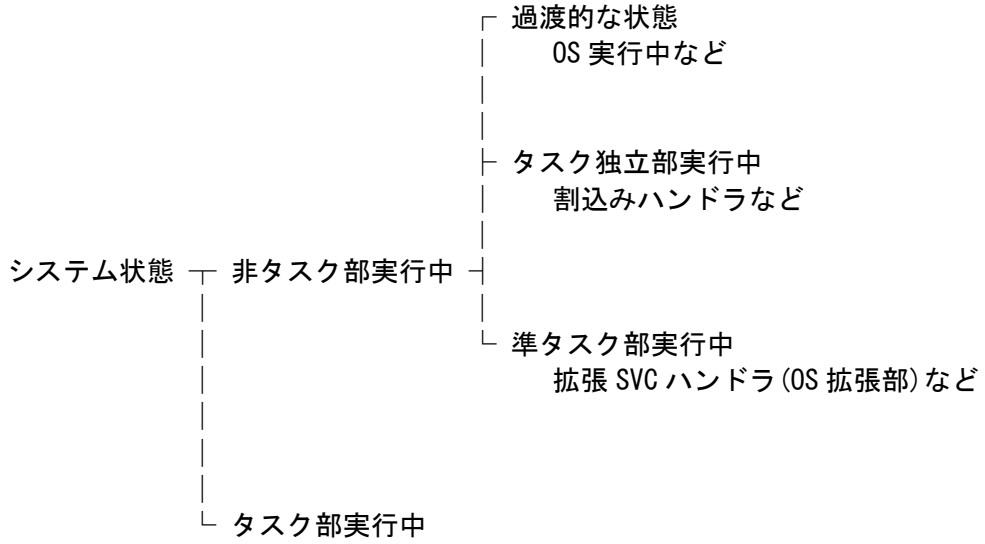
- メモリの獲得・解放を伴うシステムコールで、メモリの獲得・解放を行っている間。(T-Kernel/SM のシステムメモリ管理機能を呼出している間)
- 仮想記憶システムにおいて、システムコールの処理中に非常駐メモリのアクセスを行った場合。

このような、過渡的な状態にあるタスクに対して、タスクの強制終了(tk_ter_tsk)を行った場合の動作は保証されない。また、タスクの強制待ち(tk_sus_tsk)も過渡的な状態のまま停止することになり、それによりデッドロック等を引き起こす可能性がある。

したがって、tk_ter_tsk, tk_sus_tsk は原則として使用できない。これらを使用するのは、仮想記憶システムやデバッガのような OS にごく近い、OS の一部といえるようなサブシステム内のみとするべきである。

「タスク独立部」「準タスク部」は、ハンドラ実行中の状態に相当する。ハンドラのうち、タスクのコンテキストを持つ部分が「準タスク部」であり、タスクとは独立したコンテキストを持つ部分が「タスク独立部」である。具体的には、ユーザの定義した拡張システムコールの処理を行う拡張 SVC ハンドラが「準タスク部」であり、外部割込みによって起動される割込みハンドラやタイムイベントハンドラが「タスク独立部」である。「準タスク部」では、一般のタスクと同じようにタスクの状態遷移を考えることができ、待ち状態に入るシステムコールも発行可能である。

「過渡的な状態」、「タスク独立部」、「準タスク部」を合わせて「非タスク部」と呼ぶ。これ以外で、普通にタスクのプログラムを実行している状態が「タスク部実行中」の状態である。



[図 4] システム状態の分類

2.5.2 タスク独立部と準タスク部

タスク独立部(割込みハンドラやタイムイベントハンドラ)の特徴は、タスク独立部に入る直前に実行中だったタスクを特定することが無意味であり、「自タスク」の概念が存在しないことである。したがって、タスク独立部からは、待ち状態に入るシステムコールや、暗黙で自タスクを指定するシステムコールを発行することはできない。また、タスク独立部では現在実行中のタスクが特定できないので、タスクの切り換え（ディスパッチ)は起らない。ディスパッチが必要になっても、それはタスク独立部を抜けるまで遅らされる。これを遅延ディスパッチ(delayed dispatching)の原則と呼ぶ。

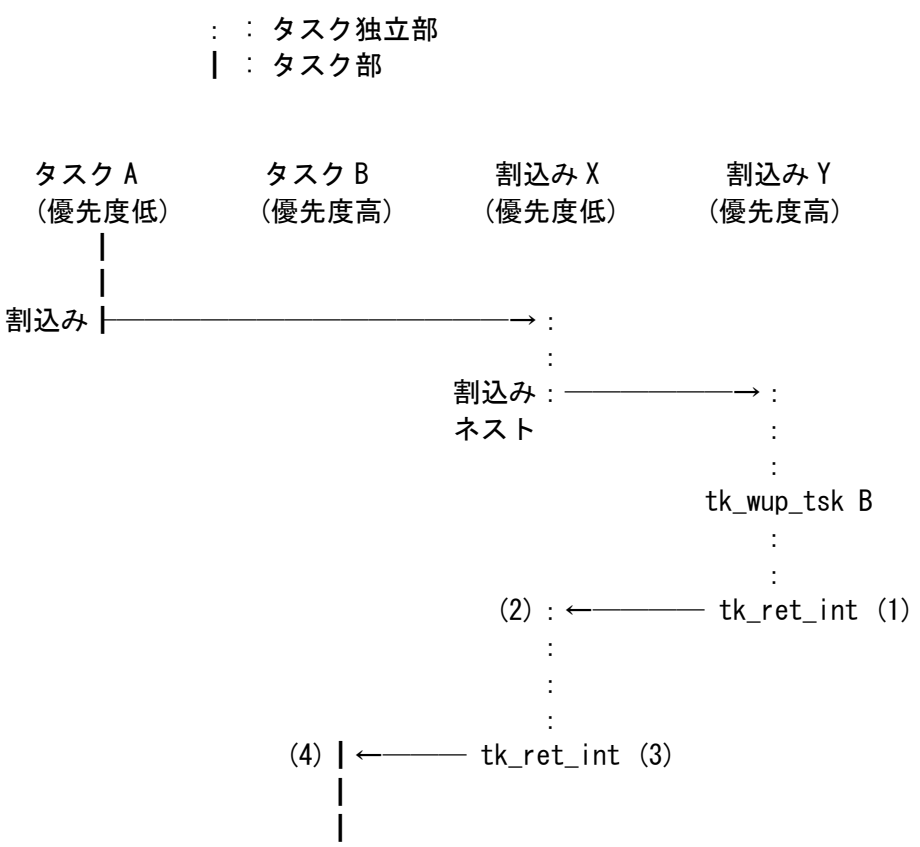
もし､タスク独立部である割込みハンドラの中でディスパッチを行うと､割込みハンドラの残りの部分の実行が、そこで起動されるタスクよりも後回しになるため、割込みがネストしたような場合に問題が起こる。この様子を[図5]に示す。

[図 5]は、タスク A の実行中に割込み X が発生し、その割込みハンドラ中でさらに高優先度の割込み Y が発生した状態を示している。この場合、(1)の割込み Y からのリターン時に即座にディスパッチを起こしてタスク B を起動すると、割込み X の(2)～(3)の部分の実行がタスク B よりも後回しになり、タスク A が実行状態になった時にはじめて(2)～(3)が実行されることになる。これでは、低優先度の割込み X のハンドラが、高優先度の割込みばかりではなく、それによって起動されたタスク B にもプリエンプトされる危険を持つことになる。したがって、割込みハンドラがタスクに優先して実行されるという保証がなくなり、割込みハンドラが書けなくなってしまう。遅延ディスパッチの原則を設けているのは、こういった理由による。

それに対して、準タスク部の特徴は、準タスク部に入る直前に実行中だったタスク(要求タスク)を特定することが可能であること、タスク部と同じようにタスクの状態が定義されており、準タスク部の中で待ち状態に入ることも可能なことである。したがって、準タスク部の中では、通常のタスク実行中の状態と同じようにディスパッチングが起きる。その結果、OS 拡張部などの準タスク部は、非タスク部であるにもかかわらず、常にタスク部に優先して実行されるとは限らない｡これは､割込みハンドラがすべてのタスクに優先して実行されるのとは対照的である。

次の二つの例は、タスク独立部と準タスク部の違いを示すものである。

- タスク A(優先度 8=低)の実行中に割込みがかかり、その割込みハンドラ(タスク独立部である)の中でタスク B(優先度 2=高)に対する tk_wup_tsk が実行された。しかし、遅延ディスパッチの原則により、ここではまだディスパッチが起きず、tk_wup_tsk 実行後はまず割込みハンドラの残りの部分が実行される。割込みハンドラの後の tk_ret_int によって、はじめてディスパッチングが起り、タスク B が実行される。
- タスク A(優先度 8=低)の中で拡張システムコールが実行され、その拡張 SVC ハンドラ(準タスク部とする)の中のタスク B(優先度 2=高)に対する tk_wup_tsk が実行された｡この場合は遅延ディスパッチの原則が適用されないので、tk_wup_tsk の中でディスパッチングが行なわれ、タスク A が準タスク部内での実行可能状態に、タスク B が実行状態になる。したがって、拡張 SVC ハンドラの残りの部分よりもタスク B の方が先に実行される。拡張 SVC ハンドラの残りの部分は、再びディスパッチングが起ってタスク A が実行状態となった後で実行される。



※ (1)でディスパッチを行うと、割込み X のハンドラの残りの部分((2)～(3))の実行が後回しになってしまう。

［図 5］割込みのネストと遅延ディスパッチ

2.6 オブジェクト

カーネルが操作対象とする資源をオブジェクトと総称する。オブジェクトには、タスクのほかに、メモリプールや、セマフォ、イベントフラグ、メールボックスなどの同期・通信機構、およびタイムイベントハンドラ(周期ハンドラおよびアラームハンドラ)が含まれている。

オブジェクト生成時には、原則として属性を指定することができる。属性はオブジェクトの細かな動作の違いやオブジェクトの初期状態を定める。属性に TA_XXXXX が指定されている場合、そのオブジェクトを「TA_XXXXX 属性のオブジェクト」と呼ぶ。特に指定すべき属性がない場合には、TA_NULL(＝0)を指定する。オブジェクト登録後に属性を読み出すインタフェースは、一般には用意されない。

オブジェクトやハンドラの属性は、下位側がシステム属性を表し、上位側が実装独自属性を表す。どのビット位置を境に上位側/下位側と区別するかは特に定めない。基本的には、標準仕様で定義されていないビットは実装独自属性として使用できる。ただし、原則としてシステム属性は LSB(最下位ビット)から MSB(最上位ビット)に向かって順に割当てる。実装独自属性では MSB から LSB に向かって割当てるものとする。属性の未定義のビットは 0 クリアされていなければならない。

オブジェクトは拡張情報を持つ場合がある。拡張情報はオブジェクト登録時に指定し、オブジェクトが実行を始める時にパラメータとして渡される情報で、カーネルの動作には影響を与えない。拡張情報はオブジェクトの状態参照システムコールで読み出すことができる。

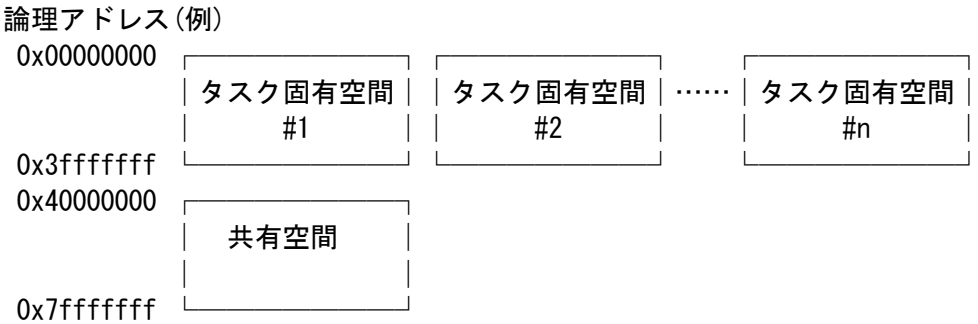
オブジェクトは ID 番号により識別される。T-Kernel では、ID 番号はオブジェクトの生成時に自動的に割当てられる。利用者が ID 番号を指定することはできない。このため、デバッグの際にオブジェクトを識別することが困難となる。そこで、各オブジェクトの生成の際に、デバッグ用のオブジェクト名称を指定することができる。この名称はあくまでデバッグ用であり、T-Kernel/DS の機能からのみ参照できる。また、名称に関するチェックは、T-Kernel では一切行われない。

2.7 メモリ

2.7.1 アドレス空間

メモリのアドレス空間は、共有空間とタスク固有空間に区別される。共有空間は、すべてのタスクから同じようにアクセスできる空間で、タスク固有空間はそのタスク固有空間に属しているタスクからのみアクセスできる[図6]。1つのタスク固有空間に複数のタスクが属している場合もある。

タスク固有空間と共有空間の論理アドレス空間は、CPU(およびMMU)の制限に依存するため実装依存となるが、タスク固有空間を低位アドレス、共有空間を高位アドレスにおくことを原則とする。



[図 6] アドレス空間

割込みハンドラなどのタスク独立部は、タスクでないためタスク固有空間を所有しない。そのため、タスク独立部では、タスク独立部に入る直前に実行していたタスクのタスク固有空間に属するものとする。これは、tk_get_tidで返される現在実行状態のタスクのタスク固有空間と一致する。実行状態のタスクがない時は、タスク固有空間は不定である。

アドレス空間の生成や管理などは、T-Kernel では行わない。通常は、アドレス空間の管理などを行うサブシステムが T-Kernel に組み合わされる。

MMU のない(または MMU を使用しない)システムにおいては、タスク固有空間が存在しないものと考える。

2.7.2 非常駐メモリ

メモリには常駐メモリと非常駐メモリがある。

非常駐メモリは、そのメモリをアクセスすることにより、ディスク等からメモリへデータの転送などが行われる。そのため、デバイスドライバによるディスクアクセスなどの複雑な処理が必要になる。したがって、非常駐メモリをアクセスする時には、デバイスドライバなどが動作できる状況でなければならない。そのため、ディスパッチ禁止中や割込み禁止中、タスク独立部実行中などの状況でアクセスすることはできない。

同様に、OS 内部の処理においても、クリティカルセクション内で非常駐メモリにアクセスしないようにする必要がある。特に、システムコールのパラメータとして渡されるメモリアドレスが、非常駐メモリを指している場合が考えられる。システムコールのパラメータが非常駐メモリにあることを許すか否かは、実装依存とする。

非常駐メモリのアクセスによるディスク等からの転送は、T-Kernel では行わない。通常は、仮想記憶管理などを行うサブシステムが T-Kernel に組み合わされる。

仮想記憶を使用しないシステムにおいて、システムコール等で非常駐メモリが指定された場合は、単に無視してすべて常駐メモリとして扱えばよい。

2.7.3 保護レベル

T-Kernel では、0～3 の 4 段階の保護レベルを仮定する。

- 0 が最も高い特権を持ち、3 が最も低い。
- 現在実行中の保護レベルと同じか低い保護レベルのメモリにのみアクセスできる。
- 保護レベルの遷移は、システムコールまたは拡張 SVC の呼出し、および割込み・CPU 例外により行われる。

各保護レベルは次のような用途に使用する。

保護レベル	
0	OS、サブシステム、デバイスドライバなど
1	システムアプリケーションタスク
2	(予約)
3	ユーザアプリケーションタスク

非タスク部(タスク独立部、準タスク部など)は保護レベル 0 で実行する。保護レベル 1～3 で実行できるのはタスク部のみである。保護レベル 0 でもタスク部は実行可能である。

MMU によっては、特権とユーザの 2 段階の保護レベルしかない場合もある。そのような場合は、保護レベル 0～2 を特権、保護レベル 3 をユーザとして 4 段階に見立てて扱う。MMU のないシステム、または MMU を使用しないシステムでは、保護レベル 0～3 をすべて同一に扱えばよい。

第 3 章　T-Kernel 仕様共通規定

3.1 データ型

3.1.1 汎用的なデータ型

```
typedef char            B;          /* 符号付き  8 ビット整数 */
typedef short           H;          /* 符号付き 16 ビット整数 */
typedef int             W;          /* 符号付き 32 ビット整数 */
typedef unsigned char   UB;         /* 符号無し  8 ビット整数 */
typedef unsigned short  UH;         /* 符号無し 16 ビット整数 */
typedef unsigned int    UW;         /* 符号無し 32 ビット整数 */

typedef char            VB;         /* 型が一定しない  8 ビットのデータ */
typedef short           VH;         /* 型が一定しない 16 ビットのデータ */
typedef int             VW;         /* 型が一定しない 32 ビットのデータ */
typedef void           *VP;         /* 型が一定しないデータへのポインタ */

typedef volatile          B          _B;     /* volatile 宣言付 */
typedef volatile          H          _H;
typedef volatile          W          _W;
typedef volatile         UB          _UB;
typedef volatile         UH          _UH;
typedef volatile         UW          _UW;

typedef int              INT;        /* プロセッサのビット幅の符号付き整数 */
typedef unsigned int     UINT;       /* プロセッサのビット幅の符号無し整数 */

typedef INT              ID;         /*   ID 一般 */
typedef INT              MSEC;       /*   時間一般(ミリ秒) */

typedef void             (*FP)();    /*  関数アドレス一般 */
typedef INT              (*FUNCP)(); /* 関数アドレス一般 */

#define LOCAL            static      /* ローカルシンボル定義 */
#define EXPORT                       /* グローバルシンボル定義 */
#define IMPORT           extern      /* グローバルシンボル参照 */

/*
 * ブール値
 *      TRUE = 1 と定義するが、0 以外はすべて真(TRUE)である。
 *      したがって、bool == TRUE の様な判定をしてはいけない。
 *      bool != FALSE の様に判定すること。
 */
typedef INT              BOOL;
#define TRUE             1              /* 真 */
#define FALSE            0              /* 偽 */

/*
 * TRON 文字コード
 */
typedef UH               TC;            /* TRON 文字コード */
#define TNULL            ((TC)0)        /* TRON コード文字列の終端 */
```

※VB,VH,VW と B,H,W との違いは、前者はビット数のみが分かっており、データ型の中身が分からないものを表すのに対して、後者は整数を表すことがはっきりしているという点である。

※プロセッサのビット幅は 32 ビット以上に限定する。したがって、INT, UINT は必ず 32 ビット以上の幅がある。

※BOOL は、TRUE を 1 と定義するが、0 以外はすべて真である。したがって、ブール値==TRUE の様な判定を行ってはいけない。ブール値!=FALSE の様に判定する。

【補足事項】

stksz, wupcnt, メッセージサイズなど、明らかに負の数にならないパラメータも、原則として符号付き整数(INT)のデータ型となっている。これは、整数はできるだけ符号付きの数として扱うという TRON 全般のルールに基づいたものである。また、タイムアウト(TMO tmout)のパラメータでは、これが符号付きの整数であることを利用し、TMO_FEVR(＝-1)を特殊な意味に使っている。符号無しのデータ型を持つパラメータは、ビットパターンとして扱われるもの(オブジェクト属性やイベントフラグなど)である。

3.1.2 意味が定義されているデータ型

パラメータの意味を明確にするため、出現頻度の高いデータ型や特殊な意味を持つデータ型に対して、以下のような名称を使用する。

```
typedef INT     FN;     /* 機能コード */
typedef INT     RNO;    /* ランデブ番号 */
typedef UINT    ATR;    /* オブジェクト/ハンドラ属性 */
typedef INT     ER;     /* エラーコード */
typedef INT     PRI;    /* 優先度 */
typedef INT     TMO;    /* タイムアウト指定 */
typedef UINT    RELTIM; /* 相対時間 */

typedef struct systim { /* システム時刻 */
    W       hi;             /* 上位 32 ビット */
    UW      lo;             /* 下位 32 ビット */
} SYSTIM;

/*
 * 共通定数
 */
#define NULL      0       /* 無効ポインタ */
#define TA_NULL   0       /* 特別な属性を指定しない */
#define TMO_POL   0       /* ポーリング */
#define TMO_FEVR  (-1)    /* 永久待ち */
```

※複数のデータ型を複合したデータ型の場合は、その内の最も主となるデータ型で代表する。例えば、tk_cre_tsk の戻値はタスク ID かエラーコードであるが、主となるのはタスク ID なので、データ型は ID となる。

3.2 システムコール

3.2.1 システムコールの形式

T-Kernel では、C 言語を標準的な高級言語として使用することにしており、C 言語からシステムコールを実行する場合のインタフェース方法を標準化している。
一方､アセンブラレベルのインタフェース方法は実装定義とする｡アセンブラでプログラムを作成する場合でも、C 言語のインタフェースを利用して呼び出す方法を推奨する。これにより、アセンブラで作成したプログラムも同一 CPU であれば OS が変わっても移植性が確保できる。
システムコールインタフェースでは、次のような共通原則を設けている。

- システムコールはすべて C の関数として定義される。
- 関数としての戻値は、0 または正の値が正常終了、負の値がエラーコードとなる。

システムコールインタフェースは、すべてライブラリとして提供される。C 言語のマクロやインライン関数、インラインアセンブラなどは用いない。これは、マクロやインライン関数などでは C のプログラムからしか利用することができないためである。また、インライン関数やインラインアセンブラは C 言語の標準機能ではないため、コンパイラごとにその機能が異なっている場合が多く移植性が低い。

3.2.2 タスク独立部から発行できるシステムコール

次のシステムコールは、タスク独立部およびディスパッチ禁止状態から発行できなくてはならない。
これら以外のシステムコールがタスク独立部およびディスパッチ禁止状態から発行できるか否かは実装依存である。

tk_sta_tsk	タスクの起動
tk_wup_tsk	タスクの起床
tk_rel_wai	タスク待ちの強制解除
tk_sus_tsk	タスクの強制待ち
tk_sig_sem	セマフォへの資源返却
tk_set_flg	イベントフラグのセット
tk_sig_tev	タスクイベントの送信
tk_rot_rdq	タスクの優先順位の回転
tk_get_tid	実行状態タスクの ID 参照
tk_sta_cyc	周期ハンドラの動作開始
tk_stp_cyc	周期ハンドラの動作停止
tk_sta_alm	アラームハンドラの動作開始
tk_stp_alm	アラームハンドラの動作停止
tk_ref_tsk	タスク状態の参照
tk_ref_cyc	周期ハンドラ状態の参照
tk_ref_alm	アラームハンドラ状態の参照
tk_ref_sys	システム状態の参照
tk_ret_int	割込みハンドラからの復帰

3.2.3 システムコールの呼出し制限

システムコールを呼び出すことのできる保護レベルを制限することができる。この場合、指定した保護レベルより低い保護レベルで動作しているタスク(タスク部)からシステムコールを発行した場合に、E_OACV エラーとする。
拡張 SVC の呼出しは制限されない。
例えば、保護レベル 1 より低い保護レベルからのシステムコールの呼出しを禁止した場合、保護レベル 2,3 で実行しているタスクからはシステムコールは発行できなくなる。つまり、保護レベル 2,3 で動作しているタスクは、拡張 SVC しか発行できないということになり、サブシステムの機能のみを使ってプログラムすることになる。
これは、T-Kernel を T-Kernel Extension と組み合わせる場合、T-Kernel Extension の仕様に基づくタスクから T-Kernel の機能を直接操作させないために使用する。T-Kernel をマイクロカーネルとして使用するための機能である。
システムコール呼出し制限をする保護レベルは、システム構成情報管理機能により設定する。システム構成情報管理機能については、5.7 節を参照のこと。

3.2.4 パラメータパケット形式の変更

システムコールへ渡すパラメータのいくつかは、パケット形式になっている。このパケット形式のパラメータには、システムコールへ情報を渡す入力パラメータとなるもの(T_CTSK など)とシステムコールから情報を返される出力パラメータとなるもの(T_RTSK など)がある。
これらパケット形式のパラメータには、実装独自の情報を追加することができる。ただし、標準仕様で定義されているデータの型および順序を変更してはならず、削除してもいけない。実装独自の情報を追加する場合は、標準仕様で定義されているものの後ろに追加しなければならない。
システムコールへの入力パラメータとなるパケット(T_CTSK など)では、実装独自で追加された情報が未初期化(メモリの内容が不定)のまま、システムコールを呼出した場合でも正常に動作するようにしなければならない。
通常は、追加した情報に有効な値が入っていることを示すフラグを標準仕様に含まれている属性フラグの実装独自領域に定義する。そのフラグがセットされている(1)場合にのみ追加した情報を使用し、セットされていない(0)場合にはその追加情報は未初期化(メモリの内容が不定)であるとして、デフォルト値を適用する。
これは、標準仕様の範囲内で作成されたプログラムを、再コンパイルのみで実装独自の機能拡張を行われた OS 上で動作させるための規定である。

3.2.5 機能コード

機能コードは、システムコールを識別するために、各システムコールに割り付けられる番号である。
システムコールの機能コードは特に定めない。すべて実装定義とする。
拡張 SVC の機能コードについては、tk_def_ssy を参照。

3.2.6 エラーコード

システムコールの戻値は原則として符号付きの整数で、エラーが発生した場合には負の値のエラーコード、処理を正常に終了した場合は E_OK(＝0)または正の値とする。正常終了した場合の戻値の意味はシステムコール毎に規定する。この原則の例外として、呼び出されるとリターンすることのないシステムコールがある。
リターンすることのないシステムコールは、C 言語 API では戻値を持たないもの(すなわち void 型の関数)として宣言する。
エラーコードは、メインエラーコードとサブエラーコードで構成される。エラーコードの下位 16 ビットがサブエラーコード、残りの上位ビットがメインエラーコードとなる。メインエラーコードは、検出の必要性や発生状況などにより、エラークラスに分類される。T-Kernel/OS ではサブエラーコードは使用せず、常に 0 となる。

```
#define MERCD(er)       ( (ER)(er) >> 16 )    /* メインエラーコード */
#define SERCD(er)       ( (H)(er) )           /* サブエラーコード */
#define ERCD(mer, ser)  ( (ER)(mer) << 16 | (ER)(UH)(ser) )
```

3.2.7 タイムアウト

待ち状態に入る可能性のあるシステムコールには、タイムアウトの機能を持たせる。タイムアウトは、指定された時間が経過しても処理が完了しない場合に、処理をキャンセルしてシステムコールからリターンするものである(この時、システムコールは E_TMOUT エラーを返す)。

そのため、「システムコールがエラーコードを返した場合には、システムコールを呼出したことによる副作用はない」という原則より、タイムアウトした場合には、システムコールを呼出したことで、システムの状態は変化していないのが原則である。ただし、システムコールの機能上、処理のキャンセル時に元の状態に戻せない場合は例外とし、システムコールの説明でその旨を明示する。

タイムアウト時間を 0 に設定すると、システムコールの中で待ち状態に入るべき状況になっても、待ち状態には入らない。そのため、タイムアウト時間を 0 としたシステムコール呼出しでは、待ち状態に入る可能性がない。タイムアウト時間を 0 としたシステムコール呼出しを、ポーリングと呼ぶ。すなわち、ポーリングを行うシステムコールでは、待ち状態に入る可能性がない。

各システムコールの説明では、タイムアウトがない(言い換えると、永久待ちの)場合の振舞いを説明するのを原則とする。システムコールの説明で「待ち状態に入る」ないしは「待ち状態に移行させる」と記述されている場合でも、タイムアウト時間を指定した場合には、指定時間経過後に待ち状態が解除され、メインエラーコードを E_TMOUT としてシステムコールからリターンする。また、ポーリングの場合には、待ち状態に入らずにメインエラーコードを E_TMOUT としてシステムコールからリターンする。

タイムアウト指定(TMO 型)は、正の値でタイムアウト時間、TMO_POL(＝0)でポーリング、TMO_FEVR(＝-1)で永久待ちを指定する。タイムアウト時間が指定された場合、タイムアウトの処理は、システムコールが呼び出されてから、指定された以上の時間が経過した後に行うことを保証しなければならない。

【補足事項】

ポーリングを行うシステムコールでは待ち状態に入らないため、それを呼出したタスクの優先順位は変化しない。

一般的な実装においては、タイムアウト時間に 1 が指定されると、システムコールが呼び出されてから 2 回めのタイムティックでタイムアウト処理を行う。タイムアウト時間に 0 を指定することはできないため(0 は TMO_POL に割り付けられている)、このような実装では、システムコールが呼び出された後の最初のタイムティックでタイムアウトすることはない。

3.2.8 相対時間とシステム時刻

イベントの発生する時刻を、システムコールを呼出した時刻などからの相対値で指定する場合には、相対時間(RELTIM 型)を用いる。相対時間を用いてイベントの発生時刻が指定された場合、イベントの処理は、基準となる時刻から指定された以上の時間が経過した後に行うことを保証しなければならない。イベントの発生間隔など、イベントの発生する時刻以外を指定する場合にも、相対時間(RELTIM 型)を用いる。その場合、指定された相対時間の解釈方法は、それぞれの場合毎に定める。時刻を絶対値で指定する場合には、システム時刻(SYSTIM 型)を用いる。カーネル仕様には現在のシステム時刻を設定する機能が用意されているが、この機能を用いてシステム時刻を変更した場合にも、相対時間を用いて指定されたイベントが発生する実世界の時刻(これを実時刻と呼ぶ)は変化しない。言い換えると、相対時間を用いて指定されたイベントが発生するシステム時刻は変化することになる。

- SYSTIM　システム時刻
  基準時間 1 ミリ秒、64 ビット符号付整数

```
typedef struct systim {
    W       hi;     /* 上位 32 ビット */
    UW      lo;     /* 下位 32 ビット */
} SYSTIM;
```

- RELTIM　相対時間
  基準時間 1 ミリ秒、32 ビット以上の符号なし整数(UINT)

```
typedef UINT    RELTIM;
```

- TMO　タイムアウト時間
  基準時間 1 ミリ秒、32 ビット以上の符号付整数(INT)

```
typedef INT     TMO;
```

TMO_FEVR＝(-1) で永久待ちを指定できる。

【補足事項】

RELTIM, TMO で指定された時間は、指定された時間以上経過した後にタイムアウト等が起こることを保証しなければならない。例えば、タイマー割込み周期が 1 ミリ秒間隔で、タイムアウト時間として 1 ミリ秒が指定された場合、システムコール呼出後の 2 回目のタイマー割込みでタイムアウトする。(1 回目のタイマー割込みでは 1 ミリ秒未満である。)

3.3 高級言語対応ルーチン

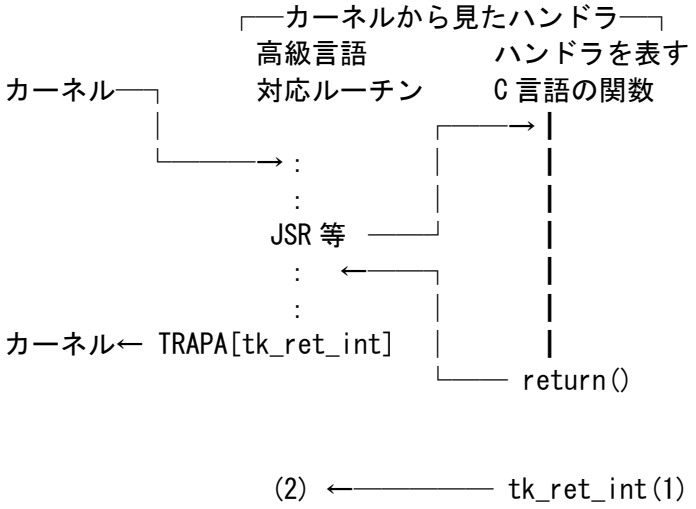
タスクやハンドラを高級言語で記述した場合にもカーネルに関する処理と言語環境に関する処理を分離できるように、高級言語対応ルーチンの機能を用意している。高級言語対応ルーチンの利用の有無は、オブジェクト属性やハンドラ属性の一つ(TA_HLNG)として指定される。

TA_HLNG の指定が無い場合は、tk_cre_tsk や tk_def_???のパラメータで指定されたスタートアドレスから直接タスクやハンドラを起動するが、TA_HLNG の指定がある場合は、最初に高級言語のスタートアップ処理ルーチン(高級言語対応ルーチン)を起動し、その中から tk_cre_tsk や tk_def_???のパラメータで指定されたタスク起動アドレスやハンドラアドレスに間接ジャンプする。この場合、OS から見ると、タスク起動アドレスやハンドラアドレスは高級言語対応ルーチンに対するパラメータとなる。こういった方法をとることにより、カーネルに関する処理と言語環境に関する処理が分離され、異なる言語環境にも容易に対応できる。

また、高級言語対応ルーチンを利用すれば、タスクやハンドラを C 言語の関数として書いた場合に、単なる関数のリターン(return や "}") を行うだけで、タスクを終了するシステムコールやハンドラから戻るシステムコールを自動的に実行することが可能となる。

しかし、MMU を持つシステムにおいては、OS と同じ保護レベルで動作する割込みハンドラなどでは比較的容易に高級言語対応ルーチンを実現できるのに対し、OS と異なる保護レベルで動作するタスクやタスク例外ハンドラなどは高級言語対応ルーチンを実現することが難しい。そのため、タスクの場合は高級言語対応ルーチンを使用したとしても関数からのリターンによるタスクの終了は保証しない。タスクの関数からリターンした場合の動作は未定義とする。また、タスク例外ハンドラの高級言語対応ルーチンはソースコードで提供することとし、ユーザプログラムに組み込むものとしている。

高級言語対応ルーチンの内部動作は[図 7]のようになる。



[図 7] 高級言語対応ルーチンの動作

第 4 章　T-Kernel/OS の機能

この章では、T-Kernel/OperatingSystem(T-Kernel/OS)で提供しているシステムコールの詳細について説明を行う。

4.1　タスク管理機能

タスク管理機能は、タスクの状態を直接的に操作／参照するための機能である。タスクを生成／削除する機能、タスクを起動／終了する機能、タスクに対する起動要求をキャンセルする機能、タスクの優先度を変更する機能、タスクの状態を参照する機能が含まれる。タスクは ID 番号で識別されるオブジェクトである。タスクの ID 番号をタスク ID と呼ぶ。タスク状態とスケジューリング規則については、2.2 節を参照すること。

タスクは、実行順序を制御するために、ベース優先度と現在優先度を持つ。単にタスクの優先度といった場合には、タスクの現在優先度を指す。タスクのベース優先度は、タスクの起動時にタスクの起動時優先度に初期化する。ミューテックス機能を使わない場合には、タスクの現在優先度は常にベース優先度に一致している。そのため、タスク起動直後の現在優先度は、タスクの起動時優先度になっている。ミューテックス機能を使う場合に現在優先度がどのように設定されるかについては、4.5.1 節で述べる。

カーネルは、タスクの終了時に、ミューテックスのロック解除を除いては、タスクが獲得した資源(セマフォ資源、メモリブロックなど)を解放する処理を行わない。タスク終了時に資源を解放するのは、アプリケーションの責任である。

タスク生成 tk_cre_tsk

tk_cre_tsk:Create Task

【C 言語インタフェース】
```
ID tskid = tk_cre_tsk ( T_CTSK *pk_ctsk ) ;
```

【パラメータ】

T_CTSK*	pk_ctsk	Packet to Create Task	タスク生成情報

pk_ctsk の内容

VP	exinf	ExtendedInformation	拡張情報
ATR	tskatr	TaskAttribute	タスク属性
FP	task	TaskStartAddress	タスク起動アドレス
PRI	itskpri	InitialTaskPriority	タスク起動時優先度
INT	stksz	StackSize	スタックサイズ(バイト数)
INT	sstksz	SystemStackSize	システムスタックサイズ(バイト数)
VP	stkptr	UserStackPointer	ユーザスタックポインタ
VP	uatb	Address of Task Space Page Table	タスク固有空間ページテーブル
INT	lsid	LogicalSpaceID	論理空間 ID
ID	resid	ResourceID	リソース ID
UB	dsname[8]	DS Object name	DS オブジェクト名称

――(以下に実装独自に他の情報を追加してもよい)――

【リターンパラメータ】

ID	tskid	TaskID	タスク ID
	または	ErrorCode	エラーコード

【エラーコード】

E_NOMEM	メモリ不足(管理ブロックやスタック用の領域が確保できない)
E_LIMIT	タスクの数がシステムの制限を超えた
E_RSATR	予約属性(tskatr が不正あるいは利用できない)、指定のコプロセッサは存在しない
E_NOSPT	未サポート機能(TA_USERSTACK,TA_TASKSPACE が未サポートの場合)
E_PAR	パラメータエラー
E_ID	リソース ID(resid)が不正
E_NOCOP	指定のコプロセッサが使用できない(動作中のハードウェアには搭載されていない、または動作異常が検出された)

【解説】

タスクを生成しタスク ID 番号を割当てる。具体的には、生成するタスクに対して TCB(Task Control Block)を割り付け、itskpri, task, stksz などの情報をもとにその初期設定を行う。

対象タスクは生成後、休止状態(DORMANT)となる。

itskpri によって、タスクが起動する時の優先度の初期値を指定する。タスク優先度としては、1～140 の値を指定することができ、数の小さい方が高い優先度となる。

exinf は、対象タスクに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、タスクに起動パラメータとして渡される他、tk_ref_tsk で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。OS では exinf の内容について関知しない。

tskatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。tskatr のシステム属性の部分では、次のような指定を行う。

```
tskatr := (TA_ASM || TA_HLNG)
   | [TA_SSTKSZ] | [TA_USERSTACK] | [TA_TASKSPACE] | [TA_RESID] | [TA_DSNAME]
   | (TA_RNG0 || TA_RNG1 || TA_RNG2 || TA_RNG3)
   | [TA_COP0] | [TA_COP1] | [TA_COP2] | [TA_COP3] | [TA_FPU]
```

TA_ASM	対象タスクがアセンブラで書かれている
TA_HLNG	対象タスクが高級言語で書かれている
TA_SSTKSZ	システムスタックサイズを指定する
TA_USERSTACK	ユーザスタックポインタを指定する
TA_TASKSPACE	タスク固有空間を指定する
TA_RESID	所属リソースグループを指定する
TA_DSNAME	DS オブジェクト名称を指定する
TA_RNGn	対象タスクは保護レベル n で実行する
TA_COPn	対象タスクが第 n 番目のコプロセッサを使用する(浮動小数点演算用コプロセッサや DSP を含む)
TA_FPU	対象タスクが浮動小数点演算用コプロセッサを使用する(TA_COPn による指定の内､特に浮動小数点演算を使用するための、CPU に依存しない汎用的な指定である)

実装独自属性の機能は､例えば､被デバッグ対象のタスクであることを指定したりするために利用できる｡また､システム属性の残りの部分は、将来マルチプロセッサ属性の指定などを行うために利用できる。

```
#define TA_ASM          0x00000000      /* アセンブラによるプログラム */
#define TA_HLNG         0x00000001      /* 高級言語によるプログラム */
#define TA_SSTKSZ       0x00000002      /* システムスタックサイズを指定 */
#define TA_USERSTACK    0x00000004      /* ユーザスタックポインタを指定 */
#define TA_TASKSPACE    0x00000008      /* タスク固有空間を指定 */
#define TA_RESID        0x00000010      /* 所属リソースグループを指定 */
#define TA_DSNAME       0x00000040      /* DS オブジェクト名称を指定 */
#define TA_RNG0         0x00000000      /* 保護レベル 0 で実行 */
#define TA_RNG1         0x00000100      /* 保護レベル 1 で実行 */
#define TA_RNG2         0x00000200      /* 保護レベル 2 で実行 */
#define TA_RNG3         0x00000300      /* 保護レベル 3 で実行 */
#define TA_COP0         0x00001000      /* ID=0 のコプロセッサを使用 */
#define TA_COP1         0x00002000      /* ID=1 のコプロセッサを使用 */
#define TA_COP2         0x00004000      /* ID=2 のコプロセッサを使用 */
#define TA_COP3         0x00008000      /* ID=3 のコプロセッサを使用 */
```

TA_HLNG の指定を行った場合には、タスク起動時に直接 task のアドレスにジャンプするのではなく、高級言語の環境設定プログラム(高級言語対応ルーチン)を通してから task のアドレスにジャンプする｡TA_HLNG 属性の場合のタスクは次の形式となる。

```
void task( INT stacd, VP exinf )
{
     /*
           処理
      */
    tk_ext_tsk(); または tk_exd_tsk(); /* タスクの終了 */
}
```

タスクの起動パラメータとして､tk_sta_tsk で指定するタスク起動コード stacd､および tk_cre_tsk で指定する拡張情報 exinf を渡す。

関数からの単純なリターン(return)でタスクを終了することはできない(してはいけない)。その場合の動作は不定(実装依存)である。

TA_ASM 属性の場合のタスクの形式は実装依存とする。ただし、起動パラメータとして stacd, exinf を渡さなければならない。

タスクは、TA_RNGn で指定された保護レベルで動作する。システムコールや拡張 SVC を呼び出すことで保護レベル 0 に移行し、システムコールや拡張 SVC から戻ると元の保護レベルに復帰する。

各タスクはシステムスタックとユーザスタックの 2 本のスタックを持つ。ユーザスタックは TA_RNGn で指定した保護レベルで使用される。システムスタックは保護レベル 0 で使用される。システムコールや拡張 SVC を呼び出すことにより保護レベルが遷移したときに使用するスタックが切り替えられる。

なお、TA_RNG0 を指定したタスクでは、保護レベルの遷移が起きないためスタックの切替も起きない。TA_RNG0 の場合は、ユーザスタックサイズとシステムスタックサイズの合計を 1 本のスタックとし、ユーザスタック兼システムスタックとして使用する。

TA_SSTKSZ を指定した場合に sstksz が有効になる。TA_SSTKSZ を指定しなかった場合は、sstksz は無視されデフォルトサイズが適用される。

TA_USERSTACK を指定した場合に stkptr が有効になる。この場合、ユーザスタックは OS で用意しない。ユーザスタックは呼びだし側で用意する。stksz には 0 を設定しなければならない。TA_USERSTACK を指定しなかった場合は、stkptr は無視される。ただし、TA_RNG0 の場合は、TA_USERSTACK を指定することはできない。TA_RNG0 と TA_USERSTACK を同時に指定した場合は E_PAR が発生する。

TA_TASKSPACE を指定した場合に uatb, lsid が有効となり、タスク固有空間として設定される。TA_TASKSPACE を指定しなかった場合は、uatb, lsid は無視され、タスク固有空間は不定となる。タスク固有空間が不定の間は、共有空間のみアクセスが許される。固有空間にはアクセスしてはいけない。TA_TASKSPACE を指定した場合も指定しなかった場合も、タスク生成後にタスク固有空間を変更することができる。なお、タスク固有空間を変更した場合、タスクが DORMANT 状態に戻ってもタスク生成時に指定したタスク固有空間に戻ることはなく、最後に設定されたタスク固有空間のままとなる。

TA_RESID を指定した場合に resid が有効となり、タスクの所属するリソースグループとして resid のリソースグループが設定される。TA_RESID を指定しなかった場合は resid は無視され、システムリソースグループに所属するよう設定される。なお、所属リソースグループを変更した場合、タスクが DORMANT 状態に戻ってもタスク生成時に指定したリソースグループに戻ることはなく、最後に設定されたリソースグループのままとなる。(tk_cre_res 参照)

TA_DSNAME を指定した場合に dsname が有効となり、DS オブジェクト名称として設定される。DS オブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DS のシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname、td_set_dsname を参照のこと。TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

【補足事項】

タスクは、TA_RNGn で指定した保護レベルと保護レベル 0 のいずれかでのみ動作する。例えば、TA_RNG3 を指定したタスクが保護レベル 1 および 2 で動作することはない。

割込みスタックが分離されていないシステムでは、割込みハンドラもシステムスタックを使用する。割込みハンドラは保護レベル 0 で動作する。

システムスタックのデフォルトサイズは、システムコールの実行により消費するサイズおよび割込みスタックが分離されていないシステムでは割込みハンドラにより消費されるサイズを考慮して決定する。

システムスタックは保護レベル 0 の共有空間の常駐メモリとなる。TA_USERSTACK が指定されなかった場合、ユーザスタックは、TA_RNGn で指定した保護レベルの共有空間の常駐メモリとなる。TA_USERSTACK が指定された場合、ユーザスタックのメモリの属性は呼出し側で任意に決定する。タスク固有空間の非常駐メモリとしてもよい。

TA_COPn の定義は、CPU などのハードウェアに依存して決められるため移植性はない。

TA_FPU は、TA_COPn の定義の内、浮動小数点演算の使用に関してのみ移植性のある指定方法として用意される。例えば、浮動小数点コプロセッサが TA_COP0 の場合は、TA_FPU=TA_COP0 となる。浮動小数点演算を行うのに特にコプロセッサの使用を指定する必要がない場合は、TA_FPU=0 となる。

MMU のないシステムにおいても、移植性確保のために TA_RNGn などすべての属性を受け付けなければならない。例えば、TA_RNGn の指定はすべて TA_RNG0 相当として処理してもよいが、エラーとはしない。

ただし、TA_USERSTACK と TA_TASKSPACE については MMU なしでは対応不可能な場合が多いと思われるため、E_NOSPT としてよい。

タスク削除 tk_del_tsk

tk_del_tsk:Delete Task

【C 言語インタフェース】
ER ercd = tk_del_tsk (ID tskid) ;

【パラメータ】

ID	tskid	TaskID	タスク ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが休止状態(DORMANT)でない)

【解説】
tskid で示されたタスクを削除する。
具体的には、tskid で指定されたタスクを休止状態(DORMANT)から未登録状態(NON-EXISTENT)(システムに存在しない状態)へと移行させ、それに伴って TCB およびスタック領域を解放する。また、タスク ID 番号も解放される。休止状態(DORMANT)でないタスクに対してこのシステムコールを実行すると、E_OBJ のエラーとなる。
このシステムコールで自タスクの指定はできない。自タスクを指定した場合には、自タスクが休止状態(DORMANT)ではないため、E_OBJ のエラーとなる。自タスクを削除するには、本システムコールではなく、tk_exd_tsk システムコールを発行する。

タスク起動 tk_sta_tsk

tk_sta_tsk:Start Task

【C 言語インタフェース】
```
ER ercd = tk_sta_tsk ( ID tskid, INT stacd ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID
INT	stacd	TaskStartCode	タスク起動コード

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが休止状態(DORMANT)でない)

【解説】

tskid で示されたタスクを起動する。具体的には、休止状態(DORMANT)から実行可能状態(READY)へと移す。

stacd により、タスクの起動時にタスクに渡すパラメータを設定することができる。このパラメータは、対象タスクから参照することができ、簡単なメッセージ通信の目的で利用できる。

タスク起動時のタスク優先度は、対象タスクが生成された時に指定されたタスク起動時優先度(itskpri)となる。

このシステムコールによる起動要求のキューイングは行わない。すなわち、対象タスクが休止状態(DORMANT)でないのにこのシステムコールが発行された場合、このシステムコールは無視され、発行タスクに E_OBJ のエラーが返る。

自タスク終了 tk_ext_tsk

tk_ext_tsk:Exit Task

【C 言語インタフェース】
```
void tk_ext_tsk ( void ) ;
```

【パラメータ】
なし

【リターンパラメータ】
※ システムコールを発行したコンテキストには戻らない

【エラーコード】
※ 次のようなエラーを検出する可能性があるが、エラーを検出した場合でも、システムコールを発行したコンテキストには戻らない。したがって、システムコールのリターンパラメータとして直接エラーコードを返すことはできない。万一エラーを検出した場合の動作は、実装依存となる。

E_CTX　　コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

【解説】
自タスクを正常終了させ、休止状態(DORMANT)へと移行させる。

【補足事項】
tk_ext_tsk によるタスクの終了時に、終了するタスクがそれ以前に獲得した資源(メモリブロック、セマフォなど) を自動的に解放するということはない。タスク終了前に資源を解放しておくのは、ユーザの責任である。
tk_ext_tsk は発行元のコンテキストに戻らないシステムコールである。したがって、何らかのエラーを検出した場合にエラーコードを返しても、これらのシステムコールを呼んだ側ではエラーのチェックを行っていないのが普通であり、プログラムが暴走する可能性がある。そこで、これらのシステムコールでは、エラーを検出した場合にも、システムコール発行元へは戻らないものとする。
タスクが休止状態(DORMANT)に戻る時は、タスク優先度など TCB に含まれている情報もリセットされるというのが原則である。たとえば、tk_chg_pri によりタスク優先度を変更されているタスクが、tk_ext_tsk により終了した時、タスク優先度は tk_cre_tsk で指定したタスク起動時優先度(itskpri)に戻る。tk_ext_tsk 実行時のタスク優先度になるわけではない。
元のコンテキストに戻らないシステムコールは、すべて tk_ret_???または tk_ext_???(tk_exd_???)の名称となっている。

自タスクの終了と削除 tk_exd_tsk

tk_exd_tsk:Exit and Delete Task

【C 言語インタフェース】
```
void tk_exd_tsk ( void ) ;
```

【パラメータ】
なし

【リターンパラメータ】
※ システムコールを発行したコンテキストには戻らない

【エラーコード】
※ 次のようなエラーを検出する可能性があるが、エラーを検出した場合でも、システムコールを発行したコンテキストには戻らない。したがって、システムコールのリターンパラメータとして直接エラーコードを返すことはできない。万一エラーを検出した場合の動作は、実装依存となる。

E_CTX　コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

【解説】
自タスクを正常終了させ、さらに自タスクを削除する。すなわち、自タスクを未登録状態(NON-EXISTENT)(システムに存在しない状態)へと移行させる。

【補足事項】
tk_exd_tsk によるタスクの終了時に、終了するタスクがそれ以前に獲得した資源(メモリブロック、セマフォなど) を自動的に解放するということはない。タスク終了前に資源を解放しておくのは、ユーザの責任である。
tk_exd_tsk は発行元のコンテキストに戻らないシステムコールである。したがって、何らかのエラーを検出した場合にエラーコードを返しても、これらのシステムコールを呼んだ側ではエラーのチェックを行っていないのが普通であり、プログラムが暴走する可能性がある。そこで、これらのシステムコールでは、エラーを検出した場合にも、システムコール発行元へは戻らないものとする。

他タスク強制終了 tk_ter_tsk

tk_ter_tsk:Terminate Task

【C 言語インタフェース】
```
ER ercd = tk_ter_tsk ( ID tskid ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが休止状態(DORMANT)または自タスク)

【解説】

tskid で示されたタスクを強制的に終了させる。すなわち、tskid で示された対象タスクを休止状態(DORMANT)に移行させる。

対象タスクが待ち状態(強制待ち状態(SUSPENDED)を含む)にあった場合でも、対象タスクは待ち解除となって終了する。また、対象タスクが何らかの待ち行列(セマフォ待ちなど)につながれていた場合には、tk_ter_tsk の実行によってその待ち行列から削除される。

本システムコールでは、自タスクの指定はできない。自タスクを指定した場合には、E_OBJ のエラーとなる。

tk_ter_tsk の対象タスクの状態と実行結果との関係についてまとめたものを[表 2]に示す。

[表 2] tk_ter_tsk の対象タスクの状態と実行結果

対象タスク状態	tk_ter_tsk の ercd	処理
実行できる状態(RUNNING,READY)(自タスク以外)	E_OK	強制終了処理
実行状態(RUNNING)(自タスク)	E_OBJ	何もしない
待ち状態(WAITING)	E_OK	強制終了処理
強制待ち状態(SUSPENDED)	E_OK	強制終了処理
二重待ち状態(WAITING-SUSPENDED)	E_OK	強制終了処理
休止状態(DORMANT)	E_OBJ	何もしない
未登録状態(NON-EXISTENT)	E_NOEXS	何もしない

【補足事項】

tk_ter_tsk によるタスクの終了時に、終了するタスクがそれ以前に獲得した資源(メモリブロック、セマフォなど)を自動的に解放するということはない。タスク終了前に資源を解放しておくのは、ユーザの責任である。

タスクが休止状態(DORMANT)に戻る時は、タスク優先度など TCB に含まれている情報もリセットされるというのが原則である。たとえば、tk_chg_pri によりタスク優先度を変更されているタスクが、tk_ter_tsk により終了した時、タスク優先度は tk_cre_tsk で指定したタスク起動時優先度(itskpri)に戻る。tk_sta_tsk によって再度タスクを起動した場合、tk_ter_tsk を実行して強制終了された時のタスク優先度になるわけではない。

他タスクの強制終了は、デバッガなどの OS に密接に関連したごく一部でのみ使用することを原則とする。一般のアプリケーションやミドルウェアでは、他タスクの強制終了は原則として使用してはいけない。これは次のような理由による。

強制終了は、対象タスクの実行状態に関係なく行われる。例えば、タスクがあるミドルウェアの機能を呼出しているとき、そのタスクを強制終了するとミドルウェアの実行途中でタスクが終了してしまうことになる。そのよう

な状況になれば、ミドルウェアの正常動作は保証できなくなる。
　このように、タスクの状態（何を実行中か）が不明な状況で、そのタスクを強制終了させることはできない。したがって、一般にタスクの強制終了は使用してはならない。

タスク優先度変更 tk_chg_pri

tk_chg_pri:Change Task Priority

【C 言語インタフェース】
```
ER ercd = tk_chg_pri ( ID tskid, PRI tskpri ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID
PRI	tskpri	TaskPriority	タスク優先度

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(tskpri が不正あるいは利用できない値)
E_ILUSE	不正使用(上限優先度違反)

【解説】

tskid で指定されるタスクのベース優先度を、tskpri で指定される値に変更する。それに伴って、タスクの現在優先度も変更する。

タスク優先度としては、1～140 の値を指定することができ、数の小さい方が高い優先度となる。

tskid に TSK_SELF(＝0)が指定されると、自タスクを対象タスクとする。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF を指定した場合には、E_ID のエラーとなる。また、tskpri に TPRI_INI(＝0)が指定されると、対象タスクのベース優先度を、タスクの起動時優先度(itskpri)に変更する。

このシステムコールで変更した優先度は、タスクが終了するまで有効である。タスクが休止状態(DORMANT)に戻る時､終了前のタスクの優先度は捨てられ､タスク生成時に指定されたタスク起動時優先度(itskpri)になる｡ただし、休止状態(DORMANT)中に変更した優先度は有効である｡次にタスクを起動したときは､その変更された優先度で起動される。

このシステムコールを実行した結果､対象タスクの現在優先度がベース優先度に一致している場合(ミューテックス機能を使わない場合には、この条件は常に成り立つ)には、次の処理を行う。

対象タスクが実行できる状態である場合、タスクの優先順位を、変更後の優先度にしたがって変化させる。変更後の優先度と同じ優先度を持つタスクの間では、対象タスクの優先順位を最低とする。

対象タスクが何らかのタスク優先度順の待ち行列につながれている場合にも、その待ち行列の中での順序を、変更後の優先度にしたがって変化させる。変更後の優先度と同じ優先度を持つタスクの間では、対象タスクを最後につなぐ。

対象タスクが TA_CEILING 属性のミューテックスをロックしているか、ロックを待っている場合で、tskpri で指定されたベース優先度が、それらのミューテックスのいずれかの上限優先度よりも高い場合には、E_ILUSE エラーを返す。

【補足事項】

このシステムコールを呼出した結果、対象タスクのタスク優先度順の待ち行列の中での順序が変化した場合、対象タスクないしはその待ち行列で待っている他のタスクの待ち解除が必要になる場合がある(メッセージバッファの送信待ち行列、および可変長メモリプールの獲得待ち行列)。

対象タスクが、TA_INHERIT 属性のミューテックスのロック待ち状態である場合、このシステムコールでベース優先度を変更したことにより、推移的な優先度継承の処理が必要になる場合がある。

ミューテックス機能を使わない場合には、対象タスクに自タスク、変更後の優先度に自タスクのベース優先度を指定してこのシステムコールが呼び出されると､自タスクの実行順位は同じ優先度を持つタスクの中で最低となる。そのため、このシステムコールを用いて、実行権の放棄を行うことができる。

タスクスライスタイム変更 tk_chg_slt

tk_chg_slt:Change Task Slicetime

【C 言語インタフェース】
```
ER ercd = tk_chg_slt ( ID tskid, RELTIM slicetime ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID
RELTIM	slicetime	SliceTime	スライスタイム(ミリ秒)

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(slicetime が不正)

【解説】

tskid で示されたタスクのスライスタイムを、slicetime で示される値に変更する。

スライスタイムはタスクのラウンドロビンスケジューリングのための機能である。タスクが slicetime 以上の時間、連続して実行されると、同じ優先度を持つタスクの中で最低の優先順位となり、自動的に実行権を次のタスクに譲る。

slicetime=0 は無制限を示し､自動的に実行権を譲ることはない｡タスク生成時には slicetime=0 に設定される。

tskid＝TSK_SELF＝0 によって自タスクの指定を行うことができる。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF＝0 を指定した場合には、E_ID のエラーとなる。

このシステムコールで変更したスライスタイムは、タスクが終了するまで有効である。タスクが休止状態(DORMANT)に戻る時、終了前のタスクのスライスタイムは捨てられ、タスク生成時の値(slicetime=0)になる。ただし、休止状態(DORMANT)中に変更したスライスタイムは有効である。次にタスクを起動したときは、その変更されたスライスタイムが適用される。

【補足事項】

より優先度の高いタスクによって実行権が奪われている間は連続実行時間としてカウントされない。また、より高い優先度のタスクによって実行権が奪われても、不連続と扱わない。つまり、より高い優先度のタスクによって実行権が奪われている間は無視して、実行時間をカウントする。

同一優先度に一つしか実行できるタスクがなければ、スライスタイムは実質的に意味はなく、そのタスクが連続して実行される。

同一優先度のタスクの中に、slicetime=0 のタスクが含まれていると、そのタスクが実行権を得た時点でラウンドロビンスケジューリングは停止することになる。

実行時間のカウント方法は実装依存だが、それほど高い精度は要求されない。逆に、アプリケーションは高い精度を期待してはいけない。

タスク固有空間の参照 tk_get_tsp

tk_get_tsp:Get Task Space

【C 言語インタフェース】
```
ER ercd = tk_get_tsp ( ID tskid, T_TSKSPC *pk_tskspc ) ;
```

【パラメータ】

ID	tskid	TaskID	対象タスクの ID
T_TSKSPC*	pk_tskspc	Packet of Task Space	タスク固有空間の情報を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_tskspc の内容
```
  typedef struct t_tskspc {
        VP      uatb;  Address of Task Space Page Table  タスク固有空間ページテーブルのアドレス
        INT     lsid;  LogicalSpaceID                    タスク固有空間 ID (論理空間 ID)
  } T_TSKSPC;
```

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(pk_tskspc が不正)

【解説】

tskid のタスクの現在のタスク固有空間情報を取得する。

tskid＝TSK_SELF＝0 で自タスクを指定することができる。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF＝0 を指定した場合には、E_ID のエラーとなる。

【補足事項】

T_TSKSPC の内容(uatb, lsid)の正確な定義は、実装定義である。ただし、できる限り上記の定義に沿うよう実装すること。

タスク固有空間の設定 tk_set_tsp

tk_set_tsp:Set Task Space

【C 言語インタフェース】
```
ER ercd = tk_set_tsp ( ID tskid, T_TSKSPC *pk_tskspc ) ;
```

【パラメータ】

ID	tskid	TaskID	対象タスクの ID
T_TSKSPC*	pk_tskspc	Packet of Task Space	タスク固有空間の情報

pk_tskspc の内容
```
typedef struct t_tskspc {
    VP      uatb;  Address of Task Space Page Table  タスク固有空間ページテーブルのアドレス
    INT     lsid;  LogicalSpaceID                    タスク固有空間 ID (論理空間 ID)
} T_TSKSPC;
```

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(pk_tskspc が不正)

【解説】

tskid のタスクのタスク固有空間を設定する。

tskid＝TSK_SELF＝0 で自タスクを指定することができる。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF＝0 を指定した場合には、E_ID のエラーとなる。

タスクの固有空間を変更したことによる影響は、OS は関知しない。例えば、タスク固有空間を実行中のタスクに対してタスク固有空間を変更すると、そのタスクが暴走するなどの可能性がある。このような問題が起きないようにするのは、呼出側の責任である。

【補足事項】

T_TSKSPC の内容(uatb, lsid)の正確な定義は、実装定義である。ただし、できる限り上記の定義に沿うよう実装すること。

タスクの所属リソースグループの参照 tk_get_rid

tk_get_rid:Get Task Resource ID

【C 言語インタフェース】
```
ID resid = tk_get_rid ( ID tskid ) ;
```

【パラメータ】

ID	tskid	TaskID	対象タスクの ID

【リターンパラメータ】

ID	resid	ResourceID	リソース ID
	または	ErrorCode	エラーコード

【エラーコード】

E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	所属リソースグループが存在しない

【解説】

tskid のタスクが現在所属しているリソースグループのリソース ID を返す。

tskid＝TSK_SELF＝0 で自タスクを指定することができる。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF＝0 を指定した場合には、E_ID のエラーとなる。

【補足事項】

所属しているリソースグループが削除されていた場合にも、削除されたリソース ID を返す場合がある。エラー(E_OBJ)を返すか否かは実装に依存する。(tk_cre_res, tk_del_res 参照)

本関数は、サブシステムで利用する。サブシステムは、リソース ID によってプロセスを認識している。しかしながら、アプリケーションから拡張 SVC を発行してサブシステムに処理を実行させる場合、リソース ID を指定することはできない。そこでサブシステムでは本関数を利用してリソース ID を取得する。

タスクの所属リソースグループの設定 tk_set_rid

tk_set_rid:Set Task Resource ID

【C 言語インタフェース】
```
ID oldid = tk_set_rid ( ID tskid, ID resid ) ;
```

【パラメータ】

ID	tskid	TaskID	対象タスクの ID
ID	resid	ResourceID	新リソース ID

【リターンパラメータ】

ID	oldid	OldResourceID	旧リソース ID
	または	ErrorCode	エラーコード

【エラーコード】

E_ID	不正 ID 番号(tskid, resid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid, resid のオブジェクトが存在しない)

【解説】

tskid のタスクが現在所属しているリソースグループを resid のリソースグループに変更する。戻値に変更前のリソースグループのリソース ID を返す。

tskid＝TSK_SELF＝0 で自タスクを指定することができる。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF＝0 を指定した場合には、E_ID のエラーとなる。

【補足事項】

resid がすでに削除されたリソースグループであっても、エラーとならない場合がある。エラー(E_NOEXS)となるか否かは実装依存である。原則として、削除されたリソースグループを指定しないようにするのは、呼出し側の責任となる。

タスクレジスタの取得 tk_get_reg

tk_get_reg:Get Task Registers

【C 言語インタフェース】

```
ER ercd = tk_get_reg ( ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs ) ;
```

【パラメータ】

ID	tskid	TaskID	対象タスクの ID
T_REGS*	pk_regs	Packet of Registers	汎用レジスタの値を返す領域へのポインタ
T_EIT*	pk_eit	Packet of EIT Registers	例外時に保存されるレジスタの値を返す領域へのポインタ
T_CREGS*	pk_cregs	Packet of Control Registers	制御レジスタの値を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

T_REGS, T_EIT, T_CREGS の内容は、CPU および実装ごとに定義する。

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正（対象タスクが自タスク）
E_CTX	コンテキストエラー（タスク独立部からの発行）

【解説】

tskid のタスクの現在のレジスタの内容を参照する。
pk_regs, pk_eit, pk_cregs にそれぞれ NULL を指定すると、対応するレジスタは参照されない。
参照されたレジスタの値が、タスク部実行中のものであるとは限らない。
自タスクに対して本システムコールを発行することはできない。(E_OBJ)

【補足事項】

参照可能なレジスタは、タスクのコンテキストに含まれるすべてのレジスタを原則とする。また、CPU に物理的に存在するレジスタ以外に、OS が仮想的にレジスタとして扱っているものがあればそれも含まれる。

タスクレジスタの設定 tk_set_reg

tk_set_reg:Set Task Registers

【C 言語インタフェース】
```
ER ercd = tk_set_reg ( ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs ) ;
```

【パラメータ】

ID	tskid	TaskID	対象タスクの ID
T_REGS*	pk_regs	Packet of Registers	汎用レジスタ
T_EIT*	pk_eit	Packet of EIT Registers	例外時に保存されるレジスタ
T_CREGS*	pk_cregs	Packet of Control Registers	制御レジスタ

T_REGS, T_EIT, T_CREGS の内容は、CPU および実装ごとに定義する。

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが自タスク)
E_CTX	コンテキストエラー(タスク独立部からの発行)
E_PAR	設定するレジスタ値が不正(実装依存)

【解説】

tskid のタスクのレジスタを指定の内容に設定する。

pk_regs, pk_eit, pk_cregs にそれぞれ NULL を指定すると、対応するレジスタは設定されない。

設定するレジスタの値が、タスク部実行中のものであるとは限らない。レジスタの値を設定したことによる影響には、OS は関知しない。

ただし、OS の動作上変更が許されないレジスタやレジスタ内の一部のビットが変更できないようになっている場合がある。(実装依存)

自タスクに対して本システムコールを発行することはできない。(E_OBJ)

コプロセッサのレジスタの取得 tk_get_cpr

tk_get_cpr:Get Task Coprocessor Registers

【C 言語インタフェース】

```
ER ercd = tk_get_cpr ( ID tskid, INT copno, T_COPREGS *pk_copregs ) ;
```

【パラメータ】

ID	tskid	TaskID	対象タスクの ID
INT	copno	CoprocessorNumber	コプロセッサ番号(0～3)
T_COPREGS*	pk_copregs	Packet of Coprocessor Registers	コプロセッサレジスタの値を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_copregs の内容

```
typedef union {
    T_COP0REG       cop0;    コプロセッサ番号 0 のレジスタ
    T_COP1REG       cop1;    コプロセッサ番号 1 のレジスタ
    T_COP2REG       cop2;    コプロセッサ番号 2 のレジスタ
    T_COP3REG       cop3;    コプロセッサ番号 3 のレジスタ
} T_COPREGS;
```

T_COPnREG の内容は、CPU および実装ごとに定義する。

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正 (対象タスクが自タスク)
E_CTX	コンテキストエラー(タスク独立部からの発行)
E_PAR	パラメータエラー(copno が不正または指定のコプロセッサは存在しない)

【解説】

tskid のタスクの copno で指定したコプロセッサの現在のレジスタ内容を参照する。
参照されたレジスタの値が、タスク部実行中のものであるとは限らない。
自タスクに対して本システムコールを発行することはできない。(E_OBJ)

【補足事項】

参照可能なレジスタは、タスクのコンテキストに含まれるすべてのレジスタを原則とする。また、CPU に物理的に存在するレジスタ以外に、OS が仮想的にレジスタとして扱っているものがあればそれも含まれる。

コプロセッサのレジスタの設定 tk_set_cpr

tk_set_cpr:Set Task Coprocessor Registers

【C 言語インタフェース】

```
ER ercd = tk_set_cpr ( ID tskid, INT copno, T_COPREGS *pk_copregs ) ;
```

【パラメータ】

ID	tskid	TaskID	対象タスクの ID
INT	copno	CoprocessorNumber	コプロセッサ番号(0～3)
T_COPREGS*	pk_copregs	Packet of Coprocessor Registers	コプロセッサのレジスタ

pk_copregs の内容

```
typedef union {
    T_COP0REG       cop0;    コプロセッサ番号 0 のレジスタ
    T_COP1REG       cop1;    コプロセッサ番号 1 のレジスタ
    T_COP2REG       cop2;    コプロセッサ番号 2 のレジスタ
    T_COP3REG       cop3;    コプロセッサ番号 3 のレジスタ
} T_COPREGS;
```

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正 (対象タスクが自タスク)
E_CTX	コンテキストエラー(タスク独立部からの発行)
E_PAR	パラメータエラー(copno が不正または指定のコプロセッサは存在しない)、 設定するレジスタ値が不正(実装依存)

【解説】

tskid のタスクの copno で指定したコプロセッサのレジスタに指定の内容を設定する。

設定するレジスタの値が、タスク部実行中のものであるとは限らない。レジスタの値を設定したことによる影響には、OS は関知しない。

ただし、OS の動作上変更が許されないレジスタやレジスタ内の一部のビットが変更できないようになっている場合がある。(実装依存)

自タスクに対して本システムコールを発行することはできない。(E_OBJ)

タスク統計情報参照 tk_inf_tsk

tk_inf_tsk:Get Task Information

【C 言語インタフェース】

```
ER ercd = tk_inf_tsk ( ID tskid, T_ITSK *pk_itsk, BOOL clr ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID
T_ITSK*	pk_itsk	Packet to Information of Task Statistics	タスク統計情報を返す領域へのポインタ
BOOL	clr	Clear	タスク統計情報のクリアの有無

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_itsk の内容

RELTIM	stime	SystemTime	累積システムレベル実行時間(ミリ秒)
RELTIM	utime	UserTime	累積ユーザレベル実行時間(ミリ秒)

――(以下に実装独自に他の情報を追加してもよい)――

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(pk_itsk が不正)

【解説】

tskid で示された対象タスクの統計情報を参照する。

clr＝TRUE≠0 の場合は、統計情報を取り出した後、累積時間をリセット(0 クリア)する。

tskid＝TSK_SELF＝0 によって自タスクの指定を行うことができる。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF＝0 を指定した場合には、E_ID のエラーとなる。

【補足事項】

システムレベル実行時間は TA_RNG0 で実行していた時間、ユーザレベル実行時間は TA_RNG0 以外で実行していた時間である。したがって、TA_RNG0 で生成されたタスクは、すべてシステムレベル実行時間としてカウントされることになる。

実行時間のカウント方法は実装依存だが、それほど高い精度は要求されない。逆に、アプリケーションは高い精度を期待してはいけない。

タスク状態参照 tk_ref_tsk

tk_ref_tsk:Refer Task Status

【C 言語インタフェース】

```
ER ercd = tk_ref_tsk ( ID tskid, T_RTSK *pk_rtsk ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID
T_RTSK*	pk_rtsk	Packet to Refer Task	タスク状態を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_rtsk の内容

VP	exinf	ExtendedInformation	拡張情報
PRI	tskpri	TaskPriority	現在の優先度
PRI	tskbpri	TaskBasePriority	ベース優先度
UINT	tskstat	TaskState	タスク状態
UINT	tskwait	TaskWaitFactor	待ち要因
ID	wid	WaitingObjectID	待ちオブジェクト ID
INT	wupcnt	WakeupCount	起床要求キューイング数
INT	suscnt	SuspendCount	強制待ち要求ネスト数
RELTIM	slicetime	SliceTime	最大連続実行時間(ミリ秒)
UINT	waitmask	WaitMask	待ちを禁止されている待ち要因
UINT	texmask	TaskExceptionMask	許可されているタスク例外
UINT	tskevent	TaskEvent	発生しているタスクイベント

――(以下に実装独自に他の情報を追加してもよい)――

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(pk_rtsk が不正)

【解説】

tskid で示された対象タスクの各種の状態を参照する。
tskstat は次のような値をとる。

tskstat:

TTS_RUN	0x0001	実行状態(RUNNING)
TTS_RDY	0x0002	実行可能状態(READY)
TTS_WAI	0x0004	待ち状態(WAITING)
TTS_SUS	0x0008	強制待ち状態(SUSPENDED)
TTS_WAS	0x000c	二重待ち状態(WAITING-SUSPENDED)
TTS_DMT	0x0010	休止状態(DORMANT)
TTS_NODISWAI	0x0080	待ち禁止拒否状態

TTS_RUN, TTS_WAI などによるタスク状態の表現はビット対応になっているため、和集合の判定を行う(例えば、実行状態(RUNNING)または実行可能状態(READY)であることを判定する)場合に便利である。なお、上記の状態のうち、

TTS_WAS は TTS_SUS と TTS_WAI が複合したものであるが、TTS_SUS がこれ以外の状態(TTS_RUN,TTS_RDY,TTS_DMT)と複合することはない。

TTS_WAI(TTS_WAS 含む)の場合、tk_dis_wai による待ち禁止を拒否している状態であれば、TTS_NODISWAI がセットされる。TTS_WAI 以外と TTS_NODISWAI が組み合わされることはない。

割込みハンドラの中から、割り込まれたタスクを対象とした tk_ref_tsk を実行した場合は、tskstat として実行状態(RUNNING)(TTS_RUN)を返す。

tskstat が TTS_WAI(TTS_WAS を含む)の場合、tskwait,wid は[表 3]のような値をとる。

[表 3] tskwait と wid の値

tskwait	値	意味	wid
TTW_SLP	0x00000001	tk_slp_tsk による待ち	0
TTW_DLY	0x00000002	tk_dly_tsk による待ち	0
TTW_SEM	0x00000004	tk_wai_sem による待ち	待ち対象の semid
TTW_FLG	0x00000008	tk_wai_flg による待ち	待ち対象の flgid
TTW_MBX	0x00000040	tk_rcv_mbx による待ち	待ち対象の mbxid
TTW_MTX	0x00000080	tk_loc_mtx による待ち	待ち対象の mtxid
TTW_SMBF	0x00000100	tk_snd_mbf による待ち	待ち対象の mbfid
TTW_RMBF	0x00000200	tk_rcv_mbf による待ち	待ち対象の mbfid
TTW_CAL	0x00000400	ランデブ呼出待ち	待ち対象の porid
TTW_ACP	0x00000800	ランデブ受付待ち	待ち対象の porid
TTW_RDV	0x00001000	ランデブ終了待ち	0
(TTW_CAL|TTW_RDV)	0x00001400	ランデブ呼出または終了待ち	0
TTW_MPF	0x00002000	tk_get_mpf による待ち	待ち対象の mpfid
TTW_MPL	0x00004000	tk_get_mpl による待ち	待ち対象の mplid
TTW_EV1	0x00010000	タスクイベント#1 待ち	0
TTW_EV2	0x00020000	タスクイベント#2 待ち	0
TTW_EV3	0x00040000	タスクイベント#3 待ち	0
TTW_EV4	0x00080000	タスクイベント#4 待ち	0
TTW_EV5	0x00100000	タスクイベント#5 待ち	0
TTW_EV6	0x00200000	タスクイベント#6 待ち	0
TTW_EV7	0x00400000	タスクイベント#7 待ち	0
TTW_EV8	0x00800000	タスクイベント#8 待ち	0

tskstat が TTS_WAI(TTS_WAS を含む)でない場合は、tskwait, wid はともに 0 となる。

waitmask は、tskwait と同じビット並びとなる。

休止状態(DORMANT)のタスクでは wupcnt=0,suscnt=0,tskevent=0 である。

tskid＝TSK_SELF＝0 によって自タスクの指定を行うことができる。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF＝0 を指定した場合には、E_ID のエラーとなる。

tk_ref_tsk で、対象タスクが存在しない場合には、エラーE_NOEXS となる。

【補足事項】

このシステムコールで tskid＝TSK_SELF を指定した場合でも、自タスクの ID は分からない。自タスクの ID を知りたい場合には、tk_get_tid を利用する。

4.2 タスク付属同期機能

タスク付属同期機能は、タスクの状態を直接的に操作することによって同期を行うための機能である。タスクを起床待ちにする機能とそこから起床する機能、タスクの起床要求をキャンセルする機能、タスクの待ち状態を強制解除する機能、タスクを強制待ち状態へ移行する機能とそこから再開する機能、自タスクの実行を遅延する機能、タスクイベントに関する機能、タスクの待ち状態を禁止する機能が含まれる。

タスクに対する起床要求は、キューイングされる。すなわち、起床待ち状態でないタスクを起床しようとすると、そのタスクを起床しようとしたという記録が残り、後でそのタスクが起床待ちに移行しようとした時に、タスクを起床待ち状態にしない。タスクに対する起床要求のキューイングを実現するために、タスクは起床要求キューイング数を持つ。タスクの起床要求キューイング数は、タスクの起動時に 0 にクリアする。

タスクに対する強制待ち要求は、ネストされる。すなわち、すでに強制待ち状態（二重待ち状態を含む）になっているタスクを再度強制待ち状態に移行させようとすると、そのタスクを強制待ち状態に移行させようとしたという記録が残り、後でそのタスクを強制待ち状態（二重待ち状態を含む）から再開させようとした時に、強制待ちからの再開を行わない。タスクに対する強制待ち要求のネストを実現するために、タスクは強制待ち要求ネスト数を持つ。タスクの強制待ち要求ネスト数は、タスクの起動時に 0 にクリアする。

自タスクを起床待ち状態へ移行 tk_slp_tsk

tk_slp_tsk:Sleep Task

【C 言語インタフェース】

```
ER ercd = tk_slp_tsk ( TMO tmout ) ;
```

【パラメータ】

TMO	tmout	Timeout	タイムアウト指定

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_PAR	パラメータエラー(tmout≦(-2))
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

【解説】

tk_slp_tsk システムコールでは、自タスクを実行状態(RUNNING)から起床待ち状態(tk_wup_tsk を待つ状態)に移す。

tmout で指定した時間が経過する前にこのタスクを対象とした tk_wup_tsk が発行された場合は、このシステムコールは正常終了する。一方、tmout で指定した時間が経過する間に tk_wup_tsk が発行されなかった場合は、タイムアウトエラー E_TMOUT となる。なお、tmout＝TMO_FEVR＝(-1)により、タイムアウトまでの時間が無限大であることを示す。この場合は、tk_wup_tsk が発行されるまで永久に待ち状態になる。

【補足事項】

tk_slp_tsk は自タスクを待ち状態に移すシステムコールであるため､tk_slp_tsk がネストすることはあり得ない。しかし、tk_slp_tsk によって待ち状態になっているタスクに対して、他のタスクから tk_sus_tsk が実行される可能性はある。この場合、このタスクは二重待ち状態(WAITING-SUSPENDED)となる。

タスクの単純な遅延(時間待ち)を行うのであれば、tk_slp_tsk ではなく、tk_dly_tsk を用いるべきである。

タスク起床待ちはアプリケーションでの利用を前提とし、ミドルウェアでは原則として使用してはいけない。これは、次の理由による。

同一タスクにおいて、2 ヵ所以上でタスク起床待ちによる同期を行うと、起床要求が混同してしまい誤動作することになる。例えば、アプリケーションとミドルウェアの双方で起床待ちによる同期を利用していた場合、ミドルウェア内で起床待ちしている間に、アプリケーションが起床要求してしまうことがありえる。このような状況になれば、ミドルウェアもアプリケーションも正常な動作はできなくなる。

このように、どこで起床待ちしているかなどの状況がわからないと、正しいタスクの同期ができなくなる。タスク起床待ちによるタスクの同期は簡便な方法としてよく利用されるため、アプリケーションで自由に利用できることを確保するために、ミドルウェアでは使用しないことを原則とする。

他タスクの起床 tk_wup_tsk

tk_wup_tsk:Wakeup Task

【C 言語インタフェース】

```
ER ercd = tk_wup_tsk ( ID tskid ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが自タスクまたは休止状態(DORMANT))
E_QOVR	キューイングまたはネストのオーバーフロー(キューイング数 wupcnt のオーバーフロー)

【解説】

tskid で示されるタスクが tk_slp_tsk の実行による待ち状態であった場合に、その待ち状態を解除する。

本システムコールでは、自タスクを指定することはできない。自タスクを指定した場合には、E_OBJ のエラーとなる。

対象タスクが tk_slp_tsk を実行しておらず、待ち状態でない場合には、tk_wup_tsk による起床要求はキューイングされる。すなわち、対象タスクに対して tk_wup_tsk が発行されたという記録が残り、この後で対象タスクが tk_slp_tsk を実行した場合にも、待ち状態にはならない。これを起床要求のキューイングと呼ぶ。

起床要求のキューイングの動作は、具体的には次のようになる。各タスクは、TCB の中に起床要求キューイング数(wupcnt)という状態を持っており、その初期値(tk_sta_tsk 実行時の値)は 0 である。起床待ち状態でないタスクに対して tk_wup_tsk を実行することにより、対象タスクの起床要求キューイング数がプラス 1 される。一方、タスクが tk_slp_tsk を実行することにより、そのタスクの起床要求キューイング数がマイナス 1 される。そうして、起床要求キューイング数＝0 のタスクが tk_slp_tsk を実行した時に、起床要求キューイング数がマイナスになる代わりに、そのタスクが待ち状態になる。

tk_wup_tsk を 1 回キューイングすること(wupcnt＝1)は常に可能であるが、起床要求キューイング数(wupcnt)の最大値は実装依存であり、1 以上の適当な値をとる。すなわち、待ち状態でないタスクに対して tk_wup_tsk を 1 回発行してもエラーとはならないが、2 回目以降の tk_wup_tsk がエラーとなるかどうかは実装依存である。

起床要求キューイング数(wupcnt)の最大値の制限を越えて tk_wup_tsk を発行した場合には、E_QOVR のエラーとなる。

タスクの起床要求を無効化 tk_can_wup

tk_can_wup:Cancel Wakeup Task

【C 言語インタフェース】

```
INT wupcnt = tk_can_wup ( ID tskid ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID

【リターンパラメータ】

INT	wupcnt	WakeupCount	キューイングされていた起床要求回数
	または	ErrorCode	エラーコード

【エラーコード】

E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが休止状態(DORMANT))

【解説】

tskid で示されたタスクの起床要求キューイング数(wupcnt)をリターンパラメータとして返し、同時にその起床要求をすべてキャンセルする。すなわち、対象タスクの起床要求キューイング数(wupcnt)を 0 にする。

tskid＝TSK_SELF＝0 によって自タスクの指定になる。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF＝0 を指定した場合には、E_ID のエラーとなる。

【補足事項】

このシステムコールは、周期的にタスクを起床して動かすような処理を行う場合に、時間内に処理が終わっているかどうかを判定するために利用できる。すなわち、前の起床要求に対する処理が終了して tk_slp_tsk を発行する前に、それを監視するタスクが tk_can_wup を発行し、そのリターンパラメータである wupcnt が 1 以上の値であった場合、前の起床要求に対する処理が時間内に終了しなかったということを示す。したがって、処理の遅れに対して何らかの処置をとることができる。

他タスクの待ち状態解除 tk_rel_wai

tk_rel_wai:Release Wait

【C 言語インタフェース】

```
ER ercd = tk_rel_wai ( ID tskid ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが待ち状態ではない(自タスクや休止状態(DORMANT)の場合を含む))

【解説】

tskid で示されるタスクが何らかの待ち状態(強制待ち状態(SUSPENDED)を除く)にある場合に、それを強制的に解除する。

tk_rel_wai により待ち状態が解除されたタスクに対しては、エラー E_RLWAI が返る。

tk_rel_wai では、待ち状態解除要求のキューイングは行わない。すなわち、tskid で示される対象タスクが既に待ち状態であればその待ち状態を解除するが、対象タスクが待ち状態でなければ、発行元にエラーE_OBJ が返る。本システムコールで自タスクを指定した場合にも、同様に E_OBJ のエラーとなる。

tk_rel_wai では、強制待ち状態(SUSPENDED)の解除は行わない。二重待ち状態(WAITING-SUSPENDED)のタスクを対象として tk_rel_wai を発行すると、対象タスクは強制待ち状態(SUSPENDED)となる。強制待ち状態(SUSPENDED)も解除する必要がある場合には、別に tk_rsm_tsk または tk_frsm_tsk を発行する。

tk_rel_wai の対象タスクの状態と実行結果との関係についてまとめたものを[表 4]に示す。

[表 4] tk_rel_wai の対象タスクの状態と実行結果

対象タスク状態	tk_rel_wai の ercd	処理
実行できる状態(RUNNING,READY)(自タスク以外)	E_OBJ	何もしない
実行状態(RUNNING)(自タスク)	E_OBJ	何もしない
待ち状態(WAITING)	E_OK	待ち解除 ※1
強制待ち状態(SUSPENDED)	E_OBJ	何もしない
二重待ち状態(WAITING-SUSPENDED)	E_OK	SUSPENDED 状態に移行
休止状態(DORMANT)	E_OBJ	何もしない
未登録状態(NON-EXISTENT)	E_NOEXS	何もしない

※ 1 対象タスクには E_RLWAI のエラーが返る。対象タスクは、資源を確保せずに(待ち解除の条件が満たされないまま)待ち解除となったことが保証される。

【補足事項】

アラームハンドラ等を用いて、あるタスクが待ち状態に入ってから一定時間後にこのシステムコールを発行することにより、タイムアウトに類似した機能を実現することができる。

tk_rel_wai と tk_wup_tsk とは次のような違いがある。

- tk_wup_tsk は tk_slp_tsk による待ち状態のみを解除するが、tk_rel_wai ではそれ以外の要因(tk_wai_flg, tk_wai_sem, tk_rcv_msg, tk_get_blk 等)による待ち状態も解除する。
- 待ち状態に入っていたタスクから見ると、tk_wup_tsk による待ち状態の解除は正常終了(E_OK)であるのに対して、tk_rel_wai による待ち状態の解除はエラー(E_RLWAI)である。
- tk_wup_tsk の場合は、対象タスクがまだ tk_slp_tsk を実行していなくても、要求がキューイングされる。一方、tk_rel_wai の場合は、対象タスクが既に待ち状態に無い場合には、E_OBJ のエラーとなる。

他タスクを強制待ち状態へ移行 tk_sus_tsk

tk_sus_tsk:Suspend Task

【C 言語インタフェース】

```
ER ercd = tk_sus_tsk ( ID tskid ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが自タスクまたは休止状態(DORMANT))
E_CTX	ディスパッチ禁止状態で実行状態のタスクを指定した
E_QOVR	キューイングまたはネストのオーバーフロー(ネスト数 suscnt のオーバーフロー)

【解説】

tskid で示されたタスクを強制待ち状態(SUSPENDED)に移し、タスクの実行を中断させる。

強制待ち状態(SUSPENDED)は、tk_rsm_tsk,tk_frsm_tsk システムコールの発行によって解除される。

tk_sus_tsk の対象タスクが既に待ち状態であった場合には、tk_sus_tsk の実行により、対象タスクは待ち状態と強制待ち状態が複合した二重待ち状態(WAITING-SUSPENDED)となる。その後、このタスクの待ち解除の条件が満たされると、対象タスクは強制待ち状態(SUSPENDED)に移行する。一方、このタスクに対して tk_rsm_tsk が発行されると、対象タスクは前と同じ待ち状態に戻る。

強制待ち状態(SUSPENDED)は他タスクの発行したシステムコールによる中断状態を意味するものなので､本システムコールで自タスクを指定することはできない。自タスクを指定した場合には、E_OBJ のエラーとなる。

タスク独立部からの本システムコールの発行において､ディスパッチ禁止状態で実行状態(RUNNING)のタスクを指定した場合は E_CTX のエラーとなる。

あるタスクに対して複数回の tk_sus_tsk が発行された場合、そのタスクは多重の強制待ち状態(SUSPENDED)になる｡これを強制待ち要求のネストと呼ぶ｡この場合､tk_sus_tsk が発行された回数(suscnt)と同じ回数の tk_rsm_tsk を発行することにより、対象タスクが元の状態に戻る。したがって、tk_sus_tsk～tk_rsm_tsk の対をネストすることが可能である。

強制待ち要求のネストの機能(同一タスクに対して 2 回以上の tk_sus_tsk を発行する機能)の有無およびその回数の制限値は、実装依存である。

強制待ち要求をネストできないシステムで複数の tk_sus_tsk が発行された場合や、ネスト回数の制限値を越えた場合には、E_QOVR のエラーとなる。

【補足事項】

あるタスクが資源獲得のための待ち状態(セマフォ待ちなど)で、かつ強制待ち状態(SUSPENDED)の場合でも、強制待ち状態(SUSPENDED)でない時と同じ条件によって資源の割当て(セマフォの割当てなど)が行われる｡強制待ち状態(SUSPENDED)であっても、資源割当ての遅延などが行われるわけではなく、資源割当てや待ち状態の解除に関する条件や優先度は全く変わらない。すなわち、強制待ち状態(SUSPENDED)は、他の処理やタスク状態と直交関係にある。

強制待ち状態(SUSPENDED)のタスクに対して資源割当ての遅延(一時的な優先度の低下) を行いたいのであれば、ユーザ側で、tk_sus_tsk, tk_rsm_tsk に tk_chg_pri を組み合せて発行すれば良い。

タスク強制待ちは、仮想記憶システムにおけるページフォルト処理やデバッガのブレークポイント処理等の OS に密接に関連したごく一部でのみ使用することを原則とする。一般のアプリケーションおよびミドルウェアでは原

則として使用してはいけない。これは、次のような理由による。

タスク強制待ちは、対象のタスクの実行状態に関係なく行われる。例えば、タスクがあるミドルウェアの機能を呼出しているとき、そのタスクを強制待ち状態にするとミドルウェアの内部でタスクが停止することになる。ミドルウェアでは、リソース管理などで排他制御等を行っている場合があり、あるタスクがミドルウェア内でリソースを獲得したまま停止してしまったことにより､他のタスクがそのミドルウェアを使用できなくなる状況が起きうる。このようにして連鎖的にタスクが停止してしまい、システム全体が停止(デッドロック)することもありうる。

このように、タスクの状態(何を実行中か)が不明な状況で、そのタスクを停止させることはできない。したがって、一般にタスク強制待ちは使用してはならない。

強制待ち状態のタスクを再開 tk_rsm_tsk
強制待ち状態のタスクを強制再開 tk_frsm_tsk

tk_rsm_tsk:Resume Task
tk_frsm_tsk:Force Resume Task

【C 言語インタフェース】

```
ER ercd = tk_rsm_tsk ( ID tskid ) ;
ER ercd = tk_frsm_tsk ( ID tskid ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが強制待ち状態(SUSPENDED)ではない(自タスクや休止状態(DORMANT)の場合を含む))

【解説】

tskid で示されたタスクの強制待ち状態(SUSPENDED)を解除する。すなわち、対象タスクが以前に発行された tk_sus_tsk によって強制待ち状態(SUSPENDED)に入り、その実行が中断している場合に、その状態を解除し、実行を再開させる。

対象タスクが待ち状態(WAITING)と強制待ち状態(SUSPENDED)の複合した二重待ち状態(WAITING-SUSPENDED)であった場合には、tk_rsm_tsk の実行により強制待ち状態(SUSPENDED)のみが解除され、対象タスクは待ち状態となる。

本システムコールでは、自タスクを指定することはできない。自タスクを指定した場合には、E_OBJ のエラーとなる。

tk_rsm_tsk では、強制待ち要求のネスト(suscnt)を 1 回分だけ解除する。したがって、対象タスクに対して 2 回以上の tk_sus_tsk が発行されていた場合(suscnt≧2)には、tk_rsm_tsk の実行が終わった後も、対象タスクはまだ強制待ち状態(SUSPENDED)のままである。一方、tk_frsm_tsk の場合は、対象タスクに対して 2 回以上の tk_sus_tsk が発行されていた場合(suscnt≧2)でも、それらの要求をすべて解除(suscnt＝0)する。すなわち、強制待ち状態(SUSPENDED)は必ず解除され、対象タスクが二重待ち状態(WAITING-SUSPENDED)でない限り実行を再開できる。

【補足事項】

実行状態(RUNNING)もしくは実行可能状態(READY)のタスクが tk_sus_tsk により強制待ち状態(SUSPENDED)になった後、tk_rsm_tsk や tk_frsm_tsk によって実行を再開した場合、そのタスクは同じ優先度を持つタスクの中で最低の優先順位となる。

たとえば、同じ優先度の task_A と task_B に対して以下のシステムコールを実行した場合、次のような動作をする。

```
tk_sta_tsk (tskid＝task_A, stacd_A);
tk_sta_tsk (tskid＝task_B, stacd_B);
 /* この時は FCFS の原則により、優先順位は task_A→task_B となっている */

tk_sus_tsk (tskid＝task_A);
tk_rsm_tsk (tskid＝task_A);
 /* この時に優先順位は task_B→task_A となる */
```

タスク遅延 tk_dly_tsk

tk_dly_tsk:Delay Task

【C 言語インタフェース】

```
ER ercd = tk_dly_tsk ( RELTIM dlytim ) ;
```

【パラメータ】

RELTIM	dlytim	DelayTime	遅延時間

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_NOMEM	メモリ不足
E_PAR	パラメータエラー(dlytim が不正)
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除

【解説】

自タスクの実行を一時的に停止し、時間経過待ちの状態に入る。タスクの実行を停止する時間は、dlytim により指定される。

時間経過待ちの状態は待ち状態の一つであり、tk_rel_wai により時間経過待ちを解除することができる。

このシステムコールを発行したタスクが強制待ち状態(SUSPENDED)または二重待ち状態(WAITING-SUSPENDED)になっている間も、時間経過のカウントは行われる。

dlytim の基準時間(時間の単位)はシステム時刻の基準時間(＝1 ミリ秒)と同じである。

【補足事項】

このシステムコールは、tk_slp_tsk とは異なり、遅延して終了した場合に正常終了となる。また、遅延時間中に tk_wup_tsk が実行されても、待ち解除とはならない。遅延時間が経過する前に tk_dly_tsk が終了するのは、tk_ter_tsk や tk_rel_wai が発行された場合に限られる。

タスクイベントの送信 tk_sig_tev

tk_sig_tev:Signal Task Event

【C 言語インタフェース】

```
ER ercd = tk_sig_tev ( ID tskid, INT tskevt ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID
INT	tskevt	TaskEvent	タスクイベント番号(1～8)

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが休止状態(DORMANT))
E_PAR	パラメータエラー(tskevt が不正)

【解説】
tskid のタスクへ tskevt で指定したタスクイベントを送信する。
タスクイベントは、タスクごとに保持される 8 種類のイベントであり、1～8 の番号で指定する。
タスクイベントの発生回数は保持されず、発生の有無のみが保持される。
tskid＝TSK_SELF＝0 によって自タスクの指定になる。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF＝0 を指定した場合には、E_ID のエラーとなる。

【補足事項】
タスクイベントは、tk_slp_tsk～tk_wup_tsk と同様のタスク付属の同期機能であるが、次の点が異なる。

- 起床要求(タスクイベント)の回数は保持されない。
- 8 種類のイベントタイプで起床要求を分類できる。

同一タスクにおいて、同じイベントタイプを使用して 2 ヵ所以上で同期を行うと混乱することになる。イベントタイプの割当ては明確に行うべきである。
タスクイベントは、ミドルウェアでの利用を前提とし、アプリケーションでは原則として使用しない。アプリケーションでは、tk_slp_tsk～tk_wup_tsk の利用を推奨する。

タスクイベント待ち tk_wai_tev

tk_wai_tev:Wait Task Event

【C 言語インタフェース】

```
INT tevptn = tk_wai_tev ( INT waiptn, TMO tmout ) ;
```

【パラメータ】

INT	waiptn	WaitEventPattern	待ちタスクイベントのパターン
TMO	tmout	Timeout	タイムアウト指定

【リターンパラメータ】

INT	tevptn	TaskEventPattern	待ち解除時のタスクイベント発生状況
	または	ErrorCode	エラーコード

【エラーコード】

E_PAR	パラメータエラー(waiptn,tmout が不正)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

【解説】

waiptn で指定したタスクイベントのいずれかが発生するまで待つ。タスクイベントの発生で待ちが解除された場合、waiptn で指定したタスクイベントはクリア(発生中のタスクイベント&=~waiptn)される。また、戻値(tevptn)に待ちが解除された時に発生していたタスクイベントの状態(クリア前の状態)を返す。

waiptn および tevptn は、各タスクイベントを 1<<(タスクイベント番号-1)のビット値として論理和(OR)した値である。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。待ち解除の条件が満足されない(tk_sig_tev が実行されない)まま tmout の時間が経過すると、タイムアウトエラーE_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1 ミリ秒)と同じである。

tmout として TMO_POL＝0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、タスクイベントが発生していなくても待ちに入らず E_TMOUT を返す。また、tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずにタスクイベントが発生するまで待ち続ける。

タスク待ち状態の禁止 tk_dis_wai

tk_dis_wai:Disable Task Wait

【C 言語インタフェース】

```
INT tskwait = tk_dis_wai ( ID tskid, UINT waitmask ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID
UINT	waitmask	WaitMask	タスク待ち禁止設定

【リターンパラメータ】

INT	tskwait	TaskWait	タスク待ち禁止後のタスク待ち状態
	または	ErrorCode	エラーコード

【エラーコード】

E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(waitmask が不正)

【解説】

tskid のタスクに対して、waitmask で指定した待ち要因による待ちに入ることを禁止する。タスクがすでに waitmask で指定した待ち要因の待ちに入っている場合は、その待ちを解除する。

waitmask には、以下の待ち要因を任意に組み合わせて論理和(OR)で指定する。

```
#define TTW_SLP   0x00000001  /* 起床待ちによる待ち */
#define TTW_DLY   0x00000002  /* タスクの遅延による待ち */
#define TTW_SEM   0x00000004  /* セマフォ待ち */
#define TTW_FLG   0x00000008  /* イベントフラグ待ち */
#define TTW_MBX   0x00000040  /* メールボックス待ち */
#define TTW_MTX   0x00000080  /* ミューテックス待ち */
#define TTW_SMBF  0x00000100  /* メッセージバッファ送信待ち */
#define TTW_RMBF  0x00000200  /* メッセージバッファ受信待ち */
#define TTW_CAL   0x00000400  /* ランデブ呼出待ち */
#define TTW_ACP   0x00000800  /* ランデブ受付待ち */
#define TTW_RDV   0x00001000  /* ランデブ終了待ち */
#define TTW_MPF   0x00002000  /* 固定長メモリプール待ち */
#define TTW_MPL   0x00004000  /* 可変長メモリプール待ち */
#define TTW_EV1   0x00010000  /* タスクイベント＃１待ち */
#define TTW_EV2   0x00020000  /* タスクイベント＃２待ち */
#define TTW_EV3   0x00040000  /* タスクイベント＃３待ち */
#define TTW_EV4   0x00080000  /* タスクイベント＃４待ち */
#define TTW_EV5   0x00100000  /* タスクイベント＃５待ち */
#define TTW_EV6   0x00200000  /* タスクイベント＃６待ち */
#define TTW_EV7   0x00400000  /* タスクイベント＃７待ち */
#define TTW_EV8   0x00800000  /* タスクイベント＃８待ち */
#define TTX_SVC   0x80000000  /* 拡張 SVC 呼出禁止 */
```

TTX_SVC はタスクの待ちではないが、拡張 SVC の呼出を禁止する特殊な指定となる。TTX_SVC が指定されている場合、そのタスクが拡張 SVC を呼び出そうとすると、拡張 SVC を呼び出さずに E_DISWAI を返す。すでに呼出している拡張 SVC を終了させるような効果はない。

戻値(tskwait)には、tk_dis_wai によって待ち禁止処理が行われた後のタスクの待ち状態を返す。この値は、tk_ref_tsk の tskwait と同じ値となる。なお、tskwait には TTX_SVC に関する情報は返されない。tskwait が 0 であれば、タスクは待ち状態に入っていない(または待ちが解除された)ことを示す。tskwait が 0 でなければ、waitmask に指定した以外の要因で待っていることになる。

タスクの待ちが tk_dis_wai により解除された場合、または待ち禁止状態で新たに待ちに入ろうとした場合は、E_DISWAI が返される。

待ち禁止状態でその待ちに入る可能性があるシステムコールを呼出したとき、待ちに入らずに処理できる場合であっても E_DISWAI が返される。例えば、メッセージバッファに空きがあり、待ちに入ることなく送信できるような状況で、メッセージバッファへ送信(tk_snd_mbf)を行った場合も、メッセージの送信は行われずに E_DISWAI が返される。

拡張 SVC の実行中に設定された待ち禁止は、拡張 SVC から呼出元に戻る際に自動的に解除される。また、拡張 SVC を呼出した時も自動的に解除され、拡張 SVC から戻ってきたときに元の設定に復帰する。

タスクが休止状態(DORMANT)に戻るときにも、自動的に解除される。ただし、休止状態(DORMANT)であるときの設定は有効であり、次にタスクが起動したときはその待ち禁止が適用される。

セマフォなど主なオブジェクトでは、オブジェクト生成時に TA_NODISWAI を指定することができる。TA_NODISWAI を指定して生成されたオブジェクトでは、tk_dis_wai による待ち禁止は拒否され、待ちは禁止されない。

tskid＝TSK_SELF＝0 によって自タスクの指定になる。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF＝0 を指定した場合には、E_ID のエラーとなる。

【補足事項】

待ち禁止機能は、拡張 SVC ハンドラの実行を中止するために用意された機能で、一般的にはブレーク関数で使用される。(その用途に限定されるものではない。)

ランデブの場合の待ち禁止は、他に比べて複雑である。基本的には、ランデブの待ち状態が変化するときに待ち禁止が検出されると待ちが解除される。

いくつか具体例を挙げる。

TTW_CAL の待ちが禁止されておらず、TTW_RDV の待ちが禁止されている場合、ランデブ呼出の待ちには入るが、ランデブが受け付けられてランデブ終了待ちになる時点で待ちが解除され E_DISWAI を返す。このとき、メッセージは受付タスクに送られ、受付タスクはメッセージを受け取りランデブ成立状態となる。受付タスクは返答(tk_rpl_rdv)を行った時点で相手タスクがないことがわかりエラー(E_OBJ)となる。

ランデブの回送の場合も待ち禁止は適用される。このとき、回送先のランデブポートの属性に従うことになる。つまり、回送先のランデブポートに TA_NODISWAI 属性の指定があれば、待ち禁止は拒否される。

ランデブ終了待ち状態となったあと TTW_CAL の待ちが禁止された場合、その状態で回送を行うと、回送によっていったんランデブ呼出待ちの状態になるため、TTW_CAL の指定により待ちが禁止される。このとき、ランデブ呼出側(tk_cal_por)と回送側(tk_fwd_por)の双方に E_DISWAI が返される。

タスク待ち禁止の解除 tk_ena_wai

tk_ena_wai:Enable Task Wait

【C 言語インタフェース】

```
ER ercd = tk_ena_wai ( ID tskid ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)

【解説】

tskid のタスクに tk_dis_wai によって設定された待ち禁止をすべて解除する。

tskid＝TSK_SELF＝0 によって自タスクの指定になる。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF＝0 を指定した場合には、E_ID のエラーとなる。

4.3 タスク例外処理機能

タスク例外処理機能は、タスクに発生した例外事象に対する処理を、割込み的にタスクのコンテキストで行うための機能である。

タスク例外ハンドラは、タスク例外が発生したタスクのコンテキストで、かつタスク生成時に指定した保護レベルで、そのタスクの一部として実行される。また、タスク例外ハンドラ内では、タスク例外に関する状態を除き、通常のタスク部の実行中と同じ状態であり、使用できるシステムコールも同等である。

タスク例外ハンドラは、対象タスクがタスク部を実行しているときにのみ起動される。対象タスクがタスク部以外を実行中のときにタスク例外が発生した場合は、タスク部へ戻ってくるまで待ってタスク例外ハンドラが起動される。準タスク部(拡張 SVC)を実行中にタスク例外が発生した場合は、その拡張 SVC に対応するブレーク関数が呼び出される。ブレーク関数によって拡張 SVC の処理が中止されタスク部へ戻ってくることになる。

発生したタスク例外要求は、タスク例外ハンドラが呼び出された(タスク例外ハンドラが実行を開始した)時点でクリアされる。

タスク例外は、0～31 の 32 種類のタスク例外コードにより指定され、0 が最も優先度が高く 31 が低い。また、タスク例外コード 0 は特殊な扱いとなる。

タスク例外コード 1～31 の動作:

- タスク例外ハンドラはネストして実行されない。タスク例外ハンドラの実行中に発生したタスク例外はペンディングされる。(タスク例外コード 0 の場合を除く)
- タスク例外ハンドラからリターンすることで、タスク例外によって割り込まれた位置に復帰する。
- タスク例外ハンドラからリターンせずに、longjmp()などによりタスク内の任意の位置にジャンプすることもできる。

タスク例外コード 0 の動作:

- タスク例外コード 1～31 のタスク例外ハンドラを実行中でもネストして実行される。タスク例外コード 0 のタスク例外ハンドラの実行中はネストされない。
- ユーザスタックポインタをタスク起動時の初期値に設定してから、タスク例外ハンドラが実行される。ただし、ユーザスタックとシステムスタックが分離されていないシステムでは、スタックポインタは初期値に戻されない。
- タスク例外ハンドラから復帰することは出来ない。必ずタスクを終了しなければならない。

タスク例外ハンドラの定義 tk_def_tex

tk_def_tex:Define Task Exception Handler

【C 言語インタフェース】

```
ER ercd = tk_def_tex ( ID tskid, T_DTEX *pk_dtex ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID
T_DTEX*	pk_dtex	Packet to Define Task Exception	タスク例外ハンドラ定義情報

pk_dtex の内容

ATR	texatr	TaskExceptionAttribute	タスク例外ハンドラ属性
FP	texhdr	TaskExceptionHandler	タスク例外ハンドラアドレス
――(以下に実装独自に他の情報を追加してもよい)――			

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(tskid のタスクは TA_RNG0 属性)
E_RSATR	予約属性(texatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_dtex が不正)

【解説】

tskid で指定したタスクに対するタスク例外ハンドラを定義する。タスク例外ハンドラはタスクに対して 1 つのみ定義可能で、すでに定義されていた場合は、後から定義した関数が有効となる。pk_dtex＝NULL の時は定義を解除する。

タスク例外ハンドラを定義または解除すると、ペンディングされているタスク例外要求はクリアされ、すべてのタスク例外は禁止状態となる。

texatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。texatr のシステム属性には、現在のバージョンでは割当てがなく、システム属性は使われていない。

タスク例外ハンドラは、下記のような形式になる。

```
void texhdr( INT texcd )
 {
   /*
       タスク例外の処理
    */

   /* タスク例外ハンドラの終了 */
   if ( texcd == 0 ) {
         tk_ext_tsk() または tk_exd_tsk();
   } else {
         tk_end_tex();
         return または longjmp();
   }
 }
```

タスク例外ハンドラは、TA_ASM 属性相当のみで高級言語対応ルーチンを経由しての呼出は行われない。したがって、タスク例外ハンドラのエントリー部分はアセンブラで作成する必要がある。OS 提供者は、上記の C 言語のタスク例外ハンドラを呼び出すためのエントリールーチンのアセンブラソースを提供しなければならない。つまり、高級言語対応ルーチンに相当するソースコードを提供しなければならない。

タスク生成時の保護レベルが TA_RNG0 のタスクに対して、タスク例外を使用することはできない。

【補足事項】

タスク生成時にはタスク例外ハンドラは定義されておらず、タスク例外も禁止されている。

タスクが休止状態(DORMANT)に戻る時には自動的にタスク例外ハンドラは解除され、タスク例外は禁止される。また、ペンディングされていたタスク例外はクリアされる。ただし、休止状態(DORMANT)であるときに、タスク例外ハンドラを定義することはできる。

タスク例外は、tk_ras_tex によって発生するソフトウェア的なもので、CPU の例外と直接の関連はない。

タスク例外の許可 tk_ena_tex
タスク例外の禁止 tk_dis_tex

tk_ena_tex:Enable Task Exception
tk_dis_tex:Disable Task Exception

【C 言語インタフェース】

```
ER ercd = tk_ena_tex ( ID tskid, UINT texptn ) ;
ER ercd = tk_dis_tex ( ID tskid, UINT texptn ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID
UINT	texptn	TaskExceptionPattern	タスク例外パターン

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない、タスク例外ハンドラが定義されていない)
E_PAR	パラメータエラー(texptn が不正あるいは利用できない)

【解説】
tskid のタスクへのタスク例外の発生を許可または禁止する。
texptn は、タスク例外コードを 1<<タスク例外コードのビット値として、論理和(OR)した値である。
tk_ena_tex は、texptn で指定したタスク例外を許可する。tk_dis_tex は、texptn で指定したタスク例外を禁止する。現在のタスク例外の許可状態を texmask とすると次のようになる。

```
許可：texmask |= texptn
禁止：texmask &= ~texptn
```

texptn の全ビットを 0 とした場合は、texmask に対して何の操作も行わないことになる。ただし、この場合でもエラーとはならない。
禁止されているタスク例外は無視され、ペンディングもされない。ペンディングされているタスク例外がある状態で、そのタスク例外が禁止された場合は、タスク例外要求が捨てられる(ペンディング状態がクリアされる)。
タスク例外ハンドラが定義されていないタスクのタスク例外を許可することはできない。
休止状態(DORMANT)のタスクに対しても適用される。

タスク例外を発生 tk_ras_tex

tk_ras_tex:Raise Task Exception

【C 言語インタフェース】

```
ER ercd = tk_ras_tex ( ID tskid, INT texcd ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID
INT	texcd	TaskExceptionCode	タスク例外コード(0～31)

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない、タスク例外ハンドラが定義されていない)
E_OBJ	オブジェクトの状態が不正(tskid のタスクは休止状態(DORMANT))
E_PAR	パラメータエラー(texcd が不正あるいは利用できない)
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

【解説】

tskid のタスクに対して texcd のタスク例外を発生させる。

tskid のタスクがタスク例外ハンドラの実行中であれば、タスク例外はペンディングされる。ペンディングされる場合、対象タスクが拡張 SVC の実行中であってもブレーク関数は実行されない。

ただし、texcd=0 の場合は、対象タスクが例外ハンドラを実行中であってもペンディングされない。対象タスクが、タスク例外コード 1～31 の例外に対するタスク例外ハンドラの実行中であればタスク例外は受け付けられ、拡張 SVC 実行中であればブレーク関数が呼び出される。対象タスクが、タスク例外コード 0 の例外に対するタスク例外ハンドラを実行中の場合は、タスク例外の発生は無視される。

tskid＝TSK_SELF＝0 によって自タスクの指定を行うことができる。

タスク独立部から発行することはできない。(E_CTX)

【補足事項】

対象タスクが拡張 SVC を実行中の場合には､その拡張 SVC に対応するブレークハンドラが tk_ras_tex を呼出したタスクのコンテキストで実行される。したがって、このような場合、ブレーク関数の実行が終わるまで tk_ras_tex から戻ってこない。また、ブレーク関数実行中に tk_ras_tex を呼出したタスクに発生したタスク例外は、ブレーク関数の終了まで保留される。

タスク例外ハンドラの終了 tk_end_tex

tk_end_tex:End Task Exception Handler

【C 言語インタフェース】

```
INT texcd = tk_end_tex ( BOOL enatex ) ;
```

【パラメータ】

BOOL	enatex	EnableTaskException	タスク例外ハンドラ呼出の許可

【リターンパラメータ】

INT	texcd	TaskExceptionCode	発生しているタスク例外コード(0～31)
	または	ErrorCode	エラーコード

【エラーコード】

E_CTX　コンテキストエラー(タスク例外ハンドラ以外またはタスク例外コード 0 (検出は実装依存))

【解説】

タスク例外ハンドラを終了し、新たなタスク例外ハンドラの呼出を許可する。ペンディングされているタスク例外がある場合は、その内の最も優先度の高いタスク例外コードを戻値に返す。ペンディングされているタスク例外がなければ 0 を返す。

enatex＝FALSE の場合、ペンディングされているタスク例外があると新たなタスク例外ハンドラの呼出は許可されない。この場合、tk_end_tex から戻った時点で、戻値に返された texcd の例外ハンドラを実行している状態になっている。ペンディングされているタスク例外がない場合は新たなタスク例外ハンドラの呼出が許可される。

enatex＝TRUE の場合は、ペンディングされているタスク例外の有無に関係なく、新たなタスク例外ハンドラの呼出を許可する。ペンディングされているタスク例外があっても、タスク例外ハンドラを終了した状態になる。

タスク例外ハンドラの終了は tk_end_tex を呼び出すこと以外にはない。タスク例外ハンドラが起動されてから tk_end_tex を呼び出すまでがタスク例外ハンドラの実行中となる。tk_end_tex を呼び出さずにタスク例外ハンドラからリターンしたとしても、リターン先はまだタスク例外ハンドラの実行中となる。同様に、tk_end_tex を呼び出さずに longjmp によりタスク例外ハンドラから抜けたとしても、そのジャンプ先はタスク例外ハンドラの実行中である。

タスク例外がペンディングされている状態で tk_end_tex を呼び出すことにより、ペンディングされていたタスク例外が新たに受け付けられる。この時、tk_end_tex を拡張 SVC ハンドラ内から呼出した場合でも、tk_end_tex を呼出した拡張 SVC ハンドラに対するブレーク関数は呼び出されない。拡張 SVC をネストして呼出していた場合は、拡張 SVC のネストが 1 段浅くなるときに戻先の拡張 SVC に対応するブレーク関数が呼び出される。タスク例外ハンドラが呼び出されるのは、タスク部に戻ってからとなる。

タスク例外コード 0 の場合、タスク例外ハンドラを終了することはできないため、tk_end_tex を発行することはできない。タスク例外コード 0 で発行した場合の動作は不定(実装依存)である。

タスク例外ハンドラ以外から発行することはできない。タスク例外ハンドラ以外から発行した場合の動作は不定(実装依存)である。

【補足事項】

tk_end_tex(TRUE)とすると、ペンディングされたタスク例外がある場合、tk_end_tex の直後にさらにタスク例外ハンドラが呼び出されることになる。そのため、スタックが戻されないままにタスク例外ハンドラが呼び出されることとなり、スタックオーバーフローの可能性がある。

通常は tk_end_tex(FALSE)を利用し、次のようにタスク例外が残っている間繰り返し処理するようにするとよい。

```
void texhdr( INT texcd )
{
        if ( texcd == 0 ) tk_exd_tsk();
        do {
             /*
                      タスク例外処理
              */
        } while ( (texcd = tk_end_tex(FALSE)) > 0 );
}
```

厳密には、tk_end_tex で 0 が返されループを終了してから texhdr を抜けるまでの間にタスク例外が発生した場合には、スタックが戻されずに texhdr に再入する可能性はある。しかし、タスク例外はソフトウェア的なものであり、タスクの実行と無関係に発生することは通常ないため、実用上は問題ないと思われる。

タスク例外の状態参照 tk_ref_tex

tk_ref_tex:Refer Task Exception Status

【C 言語インタフェース】

```
ER ercd = tk_ref_tex ( ID tskid, T_RTEX *pk_rtex ) ;
```

【パラメータ】

ID	tskid	TaskID	タスク ID
T_RTEX*	pk_rtex	Packet to Refer Task Exception	タスク例外状態を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_rtex の内容

UINT	pendtex	PendingTaskException	発生しているタスク例外
UINT	texmask	TaskExceptionMask	許可されているタスク例外

――(以下に実装独自に他の情報を追加してもよい)――

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(pk_rtex が不正)

【解説】

tskid で示された対象タスクのタスク例外の状態を参照する。

pendtex は現在発生しているタスク例外を示す。タスク例外の発生からタスク例外ハンドラが呼び出されるまでの間、pendtex に示される。

texmask は許可されているタスク例外を示す。

pendtex, texmask とも、1<<タスク例外コード のビット値として表した値である。

tskid＝TSK_SELF＝0 によって自タスクの指定を行うことができる。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF＝0 を指定した場合には、E_ID のエラーとなる。

4.4 同期・通信機能

同期・通信機能は、タスクとは独立したオブジェクトにより、タスク間の同期・通信を行うための機能である。セマフォ、イベントフラグ、メールボックスの各機能が含まれる。

4.4.1 セマフォ

セマフォは、使用されていない資源の有無や数量を数値で表現することにより、その資源を使用する際の排他制御や同期を行うためのオブジェクトである。セマフォ機能には、セマフォを生成／削除する機能、セマフォの資源を獲得／返却する機能、セマフォの状態を参照する機能が含まれる。セマフォは ID 番号で識別されるオブジェクトである。セマフォの ID 番号をセマフォ ID と呼ぶ。

セマフォは、対応する資源の有無や数量を表現する資源数と、資源の獲得を待つタスクの待ち行列を持つ。資源を m 個返却する側(イベントを知らせる側)では、セマフォの資源数を m 個増やす。一方、資源を n 個獲得する側(イベントを待つ側)では、セマフォの資源数を n 個減らす。セマフォの資源数が足りなくなった場合(具体的には、資源数を減らすと資源数が負になる場合)、資源を獲得しようとしたタスクは、次に資源が返却されるまでセマフォ資源の獲得待ち状態となる。セマフォ資源の獲得待ち状態になったタスクは、そのセマフォの待ち行列につながれる。

また、セマフォに対して資源が返却され過ぎるのを防ぐために、セマフォ毎に最大資源数を設定することができる。最大資源数を越える資源がセマフォに返却されようとした場合(具体的には、セマフォの資源数を増やすと最大資源数を越える場合)には、エラーを報告する。

セマフォ生成 tk_cre_sem

tk_cre_sem:Create Semaphore

【C 言語インタフェース】

```
ID semid = tk_cre_sem ( T_CSEM *pk_csem ) ;
```

【パラメータ】

T_CSEM* pk_csem	Packet to Create Semaphore		セマフォ生成情報
pk_csem の内容			
VP	exinf	ExtendedInformation	拡張情報
ATR	sematr	SemaphoreAttribute	セマフォ属性
INT	isemcnt	InitialSemaphoreCount	セマフォの初期値
INT	maxsem	MaximumSemaphoreCount	セマフォの最大値
UB	dsname[8]	DS Object name	DS オブジェクト名称
――(以下に実装独自に他の情報を追加してもよい)――			

【リターンパラメータ】

ID	semid	SemaphoreID	セマフォ ID
	または	ErrorCode	エラーコード

【エラーコード】

E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_LIMIT	セマフォの数がシステムの上限を超えた
E_RSATR	予約属性(sematr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_csem が不正、isemcnt, maxsem が負または不正)

【解説】

セマフォを生成しセマフォ ID 番号を割当てる。具体的には、生成するセマフォに対して管理ブロックを割り付け、その初期値を isemcnt、最大値(上限値)を maxsem とする。なお、maxsem には少なくとも 65535 が指定できなくてはならない。65536 以上の値が指定できるかは実装に依存する。

exinf は、対象セマフォに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、tk_ref_sem で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。OS では exinf の内容について関知しない。

sematr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。sematr のシステム属性の部分では、次のような指定を行う。

```
sematr:= (TA_TFIFO || TA_TPRI) | (TA_FIRST || TA_CNT) | [TA_DSNAME] | [TA_NODISWAI]
```

TA_TFIFO	待ちタスクのキューイングは FIFO
TA_TPRI	待ちタスクのキューイングは優先度順
TA_FIRST	待ち行列先頭のタスクを優先
TA_CNT	要求数の少ないタスクを優先
TA_DSNAME	DS オブジェクト名称を指定する
TA_NODISWAI	tk_dis_wai による待ち禁止を拒否する

TA_TFIFO, TA_TPRI では、タスクがセマフォの待ち行列に並ぶ際の並び方を指定することができる。属性が TA_TFIFO であればタスクの待ち行列は FIFO となり、属性が TA_TPRI であればタスクの待ち行列はタスクの優先度

順となる。

TA_FIRST,TA_CNT では、資源獲得の優先順を指定する。TA_FIRST および TA_CNT の指定によって待ち行列の並び順が変わることはない。待ち行列の並び順は TA_TFIFO,TA_TPRI によってのみ決定される。

TA_FIRST では、要求カウントに関係なく待ち行列の先頭のタスクから順に資源を割当てる。待ち行列の先頭のタスクが要求分の資源を獲得できない限り、待ち行列の後ろのタスクが資源を獲得することはない。

TA_CNT では、要求カウント分の資源が獲得できるタスクから順に割当てる。具体的には、待ち行列の先頭のタスクから順に要求カウント数を検査し、要求カウント数分の資源が割当てられるタスクに割当てる。要求カウント数の少ない順に割当てる訳ではない。

TA_DSNAME を指定した場合に dsname が有効となり、DS オブジェクト名称として設定される。DS オブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DS のシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname、td_set_dsname を参照のこと。

TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

```
#define TA_TFIFO        0x00000000      /* 待ちタスクを FIFO で管理 */
#define TA_TPRI         0x00000001      /* 待ちタスクを優先度順で管理 */
#define TA_FIRST        0x00000000      /* 待ち行列先頭のタスクを優先 */
#define TA_CNT          0x00000002      /* 要求数の少ないタスクを優先 */
#define TA_DSNAME       0x00000040      /* DS オブジェクト名称を指定 */
#define TA_NODISWAI     0x00000080      /* 待ち禁止拒否 */
```

セマフォ削除 tk_del_sem

tk_del_sem:Delete Semaphore

【C 言語インタフェース】

```
ER ercd = tk_del_sem ( ID semid ) ;
```

【パラメータ】

ID	semid	SemaphoreID	セマフォ ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(semid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(semid のセマフォが存在しない)

【解説】

semid で示されたセマフォを削除する。

本システムコールの発行により、対象セマフォの ID 番号および管理ブロック用の領域は解放される。

対象セマフォにおいて条件成立を待っているタスクがあった場合にも、本システムコールは正常終了するが、待ち状態にあったタスクにはエラー E_DLT が返される。

セマフォ資源返却 tk_sig_sem

tk_sig_sem:Signal Semaphore

【C 言語インタフェース】

```
ER ercd = tk_sig_sem ( ID semid, INT cnt ) ;
```

【パラメータ】

ID	semid	SemaphoreID	セマフォ ID
INT	cnt	Count	資源返却数

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(semid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(semid のセマフォが存在しない)
E_QOVR	キューイングまたはネストのオーバーフロー(カウント値 semcnt のオーバーフロー)
E_PAR	パラメータエラー(cnt≦0)

【解説】

semid で示されたセマフォに対して、cnt 個の資源を返却する操作を行う。対象セマフォに対して既に待っているタスクがあれば、要求カウントを確認して可能であれば資源を割当てる。資源を割当てられたタスクを実行可能状態(READY)に移す。条件によっては、複数のタスクに資源が割当てられ実行可能状態(READY)になる場合がある。

セマフォのカウント値の増加により、その値がセマフォの最大値(maxsem)を越えようとした場合は、E_QOVR のエラーとなる。この場合、資源の返却は一切行われずカウント値(semcnt)も変化しない。

【補足事項】

カウント値(semcnt)がセマフォ初期値(isemcnt)を越えた場合にも、エラーとはならない。排他制御ではなく、同期の目的(tk_wup_tsk～tk_slp_tsk と同様)でセマフォを使用する場合には、セマフォのカウント値(semcnt)が初期値(isemcnt)を越えることがある。一方、排他制御の目的でセマフォを使う場合は、セマフォ初期値(isemcnt)とセマフォ最大値(maxsem)を等しい値にしておくことにより、カウント値の増加によるエラーをチェックすることができる。

セマフォ資源獲得 tk_wai_sem

tk_wai_sem:Wait on Semaphore

【C 言語インタフェース】

```
ER ercd = tk_wai_sem ( ID semid, INT cnt, TMO tmout ) ;
```

【パラメータ】

ID	semid	SemaphoreID	セマフォ ID
INT	cnt	Count	資源要求数
TMO	tmout	Timeout	タイムアウト指定

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(semid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(semid のセマフォが存在しない)
E_PAR	パラメータエラー(tmout≦(-2), cnt≦0)
E_DLT	待ちオブジェクトが削除された(待ちの間に対象セマフォが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

【解説】

semid で示されたセマフォから、cnt 個の資源を獲得する操作を行う。資源が獲得できれば、本システムコールの発行タスクは待ち状態に入らず、実行を継続する。この場合、そのセマフォのカウント値(semcnt)は cnt 分減算される。資源が獲得できなければ、本システムコールを発行したタスクは待ち状態に入る。すなわち、そのセマフォに対する待ち行列につながれる。この場合、そのセマフォのカウント値(semcnt)は不変である。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。待ち解除の条件が満足されない(tk_sig_sem が実行されない)まま tmout の時間が経過すると、タイムアウトエラーE_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1 ミリ秒)と同じである。

tmout として TMO_POL＝0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、資源を獲得できなくても待ちに入らず E_TMOUT を返す。また、tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずに資源が獲得できるまで待ち続ける。

セマフォ状態参照 tk_ref_sem

tk_ref_sem:Refer Semaphore Status

【C 言語インタフェース】

```
ER ercd = tk_ref_sem ( ID semid, T_RSEM *pk_rsem ) ;
```

【パラメータ】

ID	semid	SemaphoreID	セマフォ ID
T_RSEM*	pk_rsem	Packet to Refer Semaphore	セマフォ状態を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_rsem の内容

VP	exinf	ExtendedInformation	拡張情報
ID	wtsk	WaitTaskInformation	待ちタスクの有無
INT	semcnt	SemaphoreCount	現在のセマフォカウント値

――(以下に実装独自に他の情報を追加してもよい)――

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(semid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(semid のセマフォが存在しない)
E_PAR	パラメータエラー(pk_rsem が不正)

【解説】

semid で示された対象セマフォの各種の状態を参照し、リターンパラメータとして現在のセマフォカウント値(semcnt)、待ちタスクの有無(wtsk)、拡張情報(exinf)を返す。

wtsk は、このセマフォで待っているタスクの ID を示す。複数のタスクが待っている場合には、待ち行列の先頭のタスクの ID を返す。待ちタスクが無い場合は wtsk＝0 となる。

対象セマフォが存在しない場合には、エラーE_NOEXS となる。

4.4.2 イベントフラグ

イベントフラグは、イベントの有無をビット毎のフラグで表現することにより、同期を行うためのオブジェクトである。イベントフラグ機能には、イベントフラグを生成／削除する機能、イベントフラグをセット／クリアする機能、イベントフラグで待つ機能、イベントフラグの状態を参照する機能が含まれる。イベントフラグは ID 番号で識別されるオブジェクトである。イベントフラグの ID 番号をイベントフラグ ID と呼ぶ。

イベントフラグは、対応するイベントの有無をビット毎に表現するビットパターンと、そのイベントフラグで待つタスクの待ち行列を持つ。イベントフラグのビットパターンを、単にイベントフラグと呼ぶ場合もある。イベントを知らせる側では、イベントフラグのビットパターンの指定したビットをセットないしはクリアすることが可能である。一方、イベントを待つ側では、イベントフラグのビットパターンの指定したビットのすべてまたはいずれかがセットされるまで、タスクをイベントフラグ待ち状態にすることができる。イベントフラグ待ち状態になったタスクは、そのイベントフラグの待ち行列につながれる。

イベントフラグ生成 tk_cre_flg

tk_cre_flg:Create EventFlag

【C 言語インタフェース】

```
ID flgid = tk_cre_flg ( T_CFLG *pk_cflg ) ;
```

【パラメータ】

T_CFLG* pk_cflg	Packet to Create EventFlag		イベントフラグ生成情報

pk_cflg の内容

VP	exinf	ExtendedInformation	拡張情報
ATR	flgatr	EventFlagAttribute	イベントフラグ属性
UINT	iflgptn	InitialEventFlagPattern	イベントフラグの初期値
UB	dsname[8]	DS Object name	DS オブジェクト名称
――(以下に実装独自に他の情報を追加してもよい)――			

【リターンパラメータ】

ID	flgid	EventFlagID	イベントフラグ ID
	または	ErrorCode	エラーコード

【エラーコード】

E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_LIMIT	イベントフラグの数がシステムの上限を超えた
E_RSATR	予約属性(flgatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_cflg が不正)

【解説】

イベントフラグを生成しイベントフラグ ID 番号を割当てる。具体的には、生成するイベントフラグに対して管理ブロックを割り付け、その初期値を iflgptn とする。一つのイベントフラグで、プロセッサの 1 ワード分のビットをグループ化して扱う。操作はすべて 1 ワード分を単位とする。

exinf は、対象イベントフラグに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、tk_ref_flg で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。OS では exinf の内容について関知しない。

flgatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。flgatr のシステム属性の部分では、次のような指定を行う。

```
flgatr:= (TA_TFIFO || TA_TPRI) | (TA_WMUL || TA_WSGL) | [TA_DSNAME] | [TA_NODISWAI]
```

TA_TFIFO	待ちタスクのキューイングは FIFO
TA_TPRI	待ちタスクのキューイングは優先度順
TA_WSGL	複数タスクの待ちを許さない (Wait Single Task)
TA_WMUL	複数タスクの待ちを許す (Wait Multiple Task)
TA_DSNAME	DS オブジェクト名称を指定する
TA_NODISWAI	tk_dis_wai による待ち禁止を拒否する

TA_WSGL を指定した場合は、複数のタスクが同時に待ち状態になることを禁止する。TA_WMUL を指定した場合は、同時に複数のタスクが待ち状態となることが許される。

TA_TFIFO, TA_TPRI では、タスクがイベントフラグの待ち行列に並ぶ際の並び方を指定することができる。属性が TA_TFIFO であればタスクの待ち行列は FIFO となり、属性が TA_TPRI であればタスクの待ち行列はタスクの優先

度順となる。ただし、TA_WSGL を指定した場合は待ち行列を作らないため、TA_TFIFO,TA_TPRI のどちらを指定しても動作に変わりはない。

複数のタスクが待っている場合、待ち行列の先頭から順に待ち条件が成立しているか検査し、待ち条件が成立しているタスクの待ちを解除する。したがって、待ち行列の先頭のタスクが必ずしも最初に待ちが解除される訳ではない。また、待ち条件が成立したタスクが複数あれば、複数のタスクの待ちが解除される。

TA_DSNAME を指定した場合に dsname が有効となり、DS オブジェクト名称として設定される。DS オブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DS のシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname、td_set_dsname を参照のこと。

TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

```
#define TA_TFIFO        0x00000000    /* 待ちタスクを FIFO で管理 */
#define TA_TPRI         0x00000001    /* 待ちタスクを優先度順で管理 */
#define TA_WSGL         0x00000000    /* 複数タスクの待ちを許さない */
#define TA_WMUL         0x00000008    /* 複数タスクの待ちを許す */
#define TA_DSNAME       0x00000040    /* DS オブジェクト名称を指定 */
#define TA_NODISWAI     0x00000080    /* 待ち禁止拒否 */
```

イベントフラグ削除 tk_del_flg

tk_del_flg:Delete EventFlag

【C 言語インタフェース】

```
ER ercd = tk_del_flg ( ID flgid ) ;
```

【パラメータ】

ID	flgid	EventFlagID	イベントフラグ ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(flgid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(flgid のイベントフラグが存在しない)

【解説】
flgid で示されたイベントフラグを削除する。
本システムコールの発行により、対象イベントフラグの ID 番号および管理ブロック用の領域は解放される。
対象イベントフラグにおいて条件成立を待っているタスクがあった場合にも、本システムコールは正常終了するが、待ち状態にあったタスクにはエラーE_DLT が返される。

イベントフラグのセット tk_set_flg
イベントフラグのクリア tk_clr_flg

tk_set_flg:Set EventFlag
tk_clr_flg:Clear EventFlag

【C 言語インタフェース】

```
ER ercd = tk_set_flg ( ID flgid, UINT setptn ) ;
ER ercd = tk_clr_flg ( ID flgid, UINT clrptn ) ;
```

【パラメータ(tk_set_flg の場合)】

ID	flgid	EventFlagID	イベントフラグ ID
UINT	setptn	SetBitPattern	セットするビットパターン

【パラメータ(tk_clr_flg の場合)】

ID	flgid	EventFlagID	イベントフラグ ID
UINT	clrptn	ClearBitPattern	クリアするビットパターン

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(flgid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(flgid のイベントフラグが存在しない)

【解説】

tk_set_flg では、flgid で示される 1 ワードのイベントフラグのうち、setptn で示されているビットがセットされる。すなわち、flgid で示されるイベントフラグの値に対して、setptn の値で論理和がとられる。また、tk_clr_flg では、flgid で示される 1 ワードのイベントフラグのうち、対応する clrptn の 0 になっているビットがクリアされる。すなわち、flgid で示されるイベントフラグの値に対して、clrptn の値で論理積がとられる。

tk_set_flg によりイベントフラグの値が変更された結果、tk_wai_flg でそのイベントフラグを待っていたタスクの待ち解除の条件を満たすようになれば、そのタスクの待ち状態が解除され、実行状態(RUNNING)または実行可能状態(READY)(待っていたタスクが二重待ち状態(WAITING-SUSPENDED)であった場合には強制待ち状態(SUSPENDED))へと移行する。

tk_clr_flg では、対象イベントフラグを待っているタスクが待ち解除となることはない。すなわち、ディスパッチは起らない。

tk_set_flg で setptn の全ビットを 0 とした場合や、tk_clr_flg で clrptn の全ビットを 1 とした場合には、対象イベントフラグに対して何の操作も行わないことになる。ただし、この場合でもエラーとはならない。

TA_WMUL の属性を持つイベントフラグに対しては、同一のイベントフラグに対して複数のタスクが同時に待つことができる。したがって、イベントフラグでもタスクが待ち行列を作ることになる。この場合、一回の tk_set_flg で複数のタスクが待ち解除となることがある。

イベントフラグ待ち tk_wai_flg

tk_wai_flg:Wait EventFlag

【C 言語インタフェース】

```
ER ercd = tk_wai_flg ( ID flgid, UINT waiptn, UINT wfmode, UINT *p_flgptn, TMO tmout ) ;
```

【パラメータ】

ID	flgid	EventFlagID	イベントフラグ ID
UINT	waiptn	WaitBitPattern	待ちビットパターン
UINT	wfmode	WaitEventFlagMode	待ちモード
UINT*	p_flgptn	Pointer to EventFlag Bit Pattern	リターンパラメータ flgptn を返す領域へのポインタ
TMO	tmout	Timeout	タイムアウト指定

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード
UINT	flgptn	EventFlagBitPattern	待ち解除時のビットパターン

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(flgid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(flgid のイベントフラグが存在しない)
E_PAR	パラメータエラー(waiptn=0, wfmode が不正, tmout≦(-2))
E_OBJ	オブジェクトの状態が不正(TA_WSGL 属性のイベントフラグに対する複数タスクの待ち)
E_DLT	待ちオブジェクトが削除された(待ちの間に対象イベントフラグが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

【解説】

tk_wai_flg では、wfmode で示される待ち解除の条件にしたがって、flgid で示されるイベントフラグがセットされるのを待つ。

flgid で示されるイベントフラグが、既に wfmode で示される待ち解除条件を満たしている場合には、発行タスクは待ち状態にならずに実行を続ける。

wfmode では、次のような指定を行う。

```
wfmode := (TWF_ANDW || TWF_ORW) | [TWF_CLR || TWF_BITCLR]
```

TWF_ANDW	0x00	AND 待ち
TWF_ORW	0x01	OR 待ち
TWF_CLR	0x10	全クリア指定
TWF_BITCLR	0x20	条件ビットのみクリア指定

TWF_ORW を指定した場合には、flgid で示されるイベントフラグのうち、waiptn で指定したビットのいずれかがセットされるのを待つ(OR 待ち)。また、TWF_ANDW を指定した場合には、flgid で示されるイベントフラグのうち、waiptn で指定したビットのすべてがセットされるのを待つ(AND 待ち)。

TWF_CLR の指定が無い場合には、条件が満足されてこのタスクが待ち解除となった場合にも、イベントフラグの値はそのままである。TWF_CLR の指定がある場合には、条件が満足されてこのタスクが待ち解除となった場合、イベントフラグの値(全部のビット)が 0 にクリアされる。TWF_BITCLR の指定がある場合には、条件が満足されてこの

タスクが待ち解除となった場合、イベントフラグの待ち解除条件に一致したビットのみが 0 クリアされる。(イベントフラグ値 &=~待ち解除条件)

flgptn は、本システムコールによる待ち状態が解除される時のイベントフラグの値(TWF_CLR または TWF_BITCLR 指定の場合は、イベントフラグがクリアされる前の値)を示すリターンパラメータである。flgptn で返る値は、このシステムコールの待ち解除の条件を満たすものになっている。なお、タイムアウト等で待ちが解除された場合は、flgptn の内容は不定となる。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。待ち解除の条件が満足されないまま tmout の時間が経過すると、タイムアウトエラー E_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1 ミリ秒)と同じである。

tmout として TMO_POL＝0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、条件を満たしていなくても待ちに入らず E_TMOUT を返す。また、tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずに条件が成立するまで待ち続ける。

タイムアウトした場合は、TWF_CLR または TWF_BITCLR の指定があってもイベントフラグのクリアは行われない。

waiptn を 0 とした場合は、パラメータエラー E_PAR になる。

既に待ちタスクの存在する TA_WSGL 属性のイベントフラグに対して、別のタスクが tk_wai_flg を実行することはできない。この場合は、後から tk_wai_flg を実行したタスクが待ち状態に入るかどうか(待ち解除条件が満たされているかどうか)にかかわらず、後から tk_wai_flg を実行したタスクは E_OBJ のエラーとなる。

一方、TA_WMUL の属性を持つイベントフラグに対しては、同一のイベントフラグに対して複数のタスクが同時に待つことができる。したがって、イベントフラグでもタスクが待ち行列を作ることになる。この場合、一回の tk_set_flg で複数のタスクが待ち解除となることがある。

TA_WMUL 属性を持つイベントフラグに対して複数のタスクが待ち行列を作った場合、次のような動作をする。

- 待ち行列の順番は FIFO またはタスク優先度順である。(ただし、waiptn や wfmode との関係により、必ずしも行列先頭のタスクから待ち解除になるとは限らない。)
- 待ち行列中にクリア指定のタスクがあれば、そのタスクが待ち解除になる時に、フラグをクリアする。
- クリア指定を行っていたタスクよりも後ろの待ち行列にあったタスクは、既にクリアされた後のイベントフラグを見ることになる。

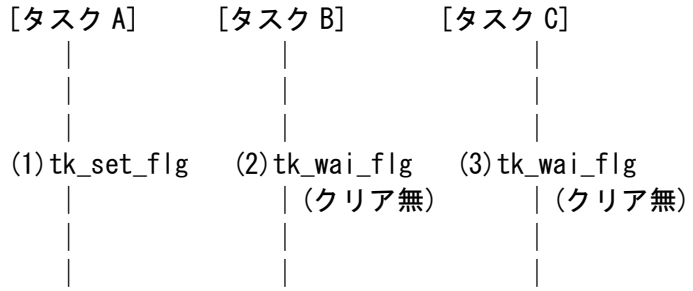
tk_set_flg によって同じ優先度を持つ複数のタスクが同時に待ち解除となる場合、待ち解除後のタスクの優先順位は、元のイベントフラグの待ち行列の順序を保存する。

【補足事項】

tk_wai_flg の待ち解除条件として全ビットの論理和を指定すれば(waiptn＝0xfff...ff, wfmode＝TWF_ORW)、tk_set_flg と組み合わせることによって、1 ワードのビットパターンによるメッセージ転送を行うことができる。ただし、この場合、全部のビットが 0 というメッセージを送ることはできない。また、前のメッセージが tk_wai_flg で読まれる前に tk_set_flg により次のメッセージが送られると、前のメッセージは消えてしまう。すなわち、メッセージのキューイングはできない。

waiptn＝0 の指定は E_PAR のエラーになるので、イベントフラグで待つタスクの waiptn は 0 でないことが保証されている。したがって、tk_set_flg で全ビットをセットすれば、どのような条件でイベントフラグを待つタスクであっても、待ち行列の先頭にあるタスクは必ず待ち解除となる。

イベントフラグに対する複数タスク待ちの機能は、次のような場合に有効である。例えば、タスク A が(1)の tk_set_flg を実行するまで、タスク B、タスク C を(2),(3)の tk_wai_flg で待たせておく場合に、イベントフラグの複数待ちが可能であれば、(1)(2)(3)のどのシステムコールが先に実行されても結果は同じになる[図 8]。一方、イベントフラグの複数待ちができなければ、(2)→(3)→(1)の順でシステムコールが実行された場合に、(3)の tk_wai_flg が E_OBJ のエラーになる。



[図 8]イベントフラグに対する複数タスク待ちの機能

【仕様決定の理由】

waiptn＝0 の指定を E_PAR のエラーとしたのは、waiptn＝0 の指定を許した場合に、それ以後イベントフラグがどのような値に変わっても待ち状態から抜けることができなくなるためである。

イベントフラグ状態参照 tk_ref_flg

tk_ref_flg:Refer EventFlag Status

【C 言語インタフェース】

```
ER ercd = tk_ref_flg ( ID flgid, T_RFLG *pk_rflg ) ;
```

【パラメータ】

ID	flgid	EventFlagID	イベントフラグ ID
T_RFLG*	pk_rflg	Packet to Refer Eventflag	イベントフラグ状態を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_rflg の内容

VP	exinf	ExtendedInformation	拡張情報
ID	wtsk	WaitTaskInformation	待ちタスクの有無
UINT	flgptn	EventFlagBitPattern	イベントフラグのビットパターン

――(以下に実装独自に他の情報を追加してもよい)――

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(flgid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(flgid のイベントフラグが存在しない)
E_PAR	パラメータエラー(pk_rflg が不正)

【解説】

flgid で示された対象イベントフラグの各種の状態を参照し、リターンパラメータとして現在のフラグ値(flgptn)、待ちタスクの有無(wtsk)、拡張情報(exinf)を返す。

wtsk は、このイベントフラグで待っているタスクの ID を示す。このイベントフラグで複数のタスクが待っている場合(TA_WMUL 属性のときのみ)には、待ち行列の先頭のタスクの ID を返す。待ちタスクが無い場合は wtsk＝0 となる。

対象イベントフラグが存在しない場合には、エラーE_NOEXS となる。

4.4.3 メールボックス

メールボックスは、共有メモリ上に置かれたメッセージを受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である。メールボックス機能には、メールボックスを生成／削除する機能、メールボックスに対してメッセージを送信／受信する機能、メールボックスの状態を参照する機能が含まれる。メールボックスは ID 番号で識別されるオブジェクトである。メールボックスの ID 番号をメールボックス ID と呼ぶ。

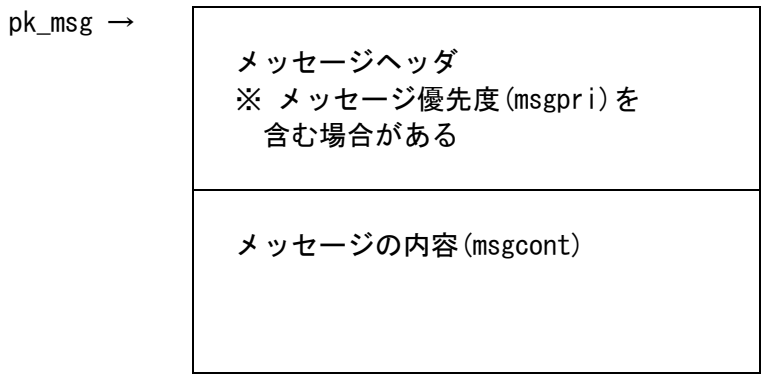
メールボックスは、送信されたメッセージを入れるためのメッセージキューと、メッセージの受信を待つタスクの待ち行列を持つ。メッセージを送信する側(イベントを知らせる側)では、送信したいメッセージをメッセージキューに入れる。一方、メッセージを受信する側(イベントを待つ側)では、メッセージキューに入っているメッセージを一つ取り出す。メッセージキューにメッセージが入っていない場合は、次にメッセージが送られてくるまでメールボックスからの受信待ち状態になる。メールボックスからの受信待ち状態になったタスクは、そのメールボックスの待ち行列につながれる。

メールボックスによって実際に送受信されるのは、送信側と受信側で共有しているメモリ上に置かれたメッセージの先頭番地のみである。すなわち、送受信されるメッセージの内容のコピーは行わない。カーネルは、メッセージキューに入っているメッセージを、リンクリストにより管理する。アプリケーションプログラムは、送信するメッセージの先頭に、カーネルがリンクリストに用いるための領域を確保しなければならない。この領域をメッセージヘッダと呼ぶ。また、メッセージヘッダと、それに続くアプリケーションがメッセージを入れるための領域をあわせて、メッセージパケットと呼ぶ。メールボックスへメッセージを送信するシステムコールは、メッセージパケットの先頭番地(pk_msg)をパラメータとする。

また、メールボックスからメッセージを受信するシステムコールは、メッセージパケットの先頭番地をリターンパラメータとして返す。

メッセージキューをメッセージの優先度順にする場合には、メッセージの優先度を入れるための領域(msgpri)も、メッセージヘッダ中に持つ必要がある。[図 9]

ユーザが実際にメッセージを入れることができるのは、メッセージ先頭アドレスの直後からではなく、メッセージヘッダの後の部分から(図の msgcont の部分)である。



[図 9] メールボックスで使用されるメッセージの形式

カーネルは、メッセージキューに入っている(ないしは、入れようとしている)メッセージのメッセージヘッダ(メッセージ優先度のための領域を除く)の内容を書き換える。一方、アプリケーションは、メッセージキューに入っているメッセージのメッセージヘッダ(メッセージ優先度のための領域を含む)の内容を書き換えてはならない。アプリケーションによってメッセージヘッダの内容が書き換えられた場合の振舞いは未定義である。この規定は、アプリケーションプログラムがメッセージヘッダの内容を直接書き換えた場合に加えて、メッセージヘッダの番地をカーネルに渡し、カーネルにメッセージヘッダの内容を書き換えさせた場合にも適用される。したがって、すでにメッセージキューに入っているメッセージを再度メールボックスに送信した場合の振舞いは未定義となる。

【補足事項】

メールボックス機能では、メッセージヘッダの領域をアプリケーションプログラムで確保することとしているため、メッセージキューに入れることができるメッセージの数には上限がない。また、メッセージを送信するシステムコールで待ち状態になることもない。

メッセージパケットとしては、固定長メモリプールまたは可変長メモリプールから動的に確保したメモリブロックを用いることも、静的に確保した領域を用いることも可能であるが、タスク固有空間上に置くことはできない。

一般的な使い方としては、送信側のタスクがメモリプールからメモリブロックを確保し、それをメッセージパケットとして送信し、受信側のタスクはメッセージの内容を取り出した後にそのメモリブロックを直接メモリプールに返却するという手順をとることが多い。

メールボックス生成 tk_cre_mbx

tk_cre_mbx:Create Mailbox

【C 言語インタフェース】

```
ID mbxid = tk_cre_mbx ( T_CMBX *pk_cmbx ) ;
```

【パラメータ】

T_CMBX*	pk_cmbx	Packet to Create Mailbox	メールボックス生成情報

pk_cmbx の内容

VP	exinf	ExtendedInformation	拡張情報
ATR	mbxatr	MailboxAttribute	メールボックス属性
UB	dsname[8]	DS Object name	DS オブジェクト名称

――(以下に実装独自に他の情報を追加してもよい)――

【リターンパラメータ】

ID	mbxid	MailboxID	メールボックス ID
	または	ErrorCode	エラーコード

【エラーコード】

E_NOMEM	メモリ不足(管理ブロックなどの領域が確保できない)
E_LIMIT	メールボックスの数がシステムの上限を超えた
E_RSATR	予約属性(mbxatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_cmbx が不正)

【解説】

メールボックスを生成しメールボックス ID 番号を割当てる。具体的には、生成するメールボックスに対して管理ブロックなどを割り付ける。

exinf は、対象メールボックスに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、tk_ref_mbx で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。OS では exinf の内容について関知しない。

mbxatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。mbxatr のシステム属性の部分では、次のような指定を行う。

```
mbxatr:= (TA_TFIFO || TA_TPRI) | (TA_MFIFO || TA_MPRI) | [TA_DSNAME] | [TA_NODISWAI]
```

TA_TFIFO	待ちタスクのキューイングは FIFO
TA_TPRI	待ちタスクのキューイングは優先度順
TA_MFIFO	メッセージのキューイングは FIFO
TA_MPRI	メッセージのキューイングは優先度順
TA_DSNAME	DS オブジェクト名称を指定する
TA_NODISWAI	tk_dis_wai による待ち禁止を拒否する

TA_TFIFO,TA_TPRI では、メッセージを受信するタスクがメールボックスの待ち行列に並ぶ際の並び方を指定することができる。属性が TA_TFIFO であればタスクの待ち行列は FIFO となり、属性が TA_TPRI であればタスクの待ち行列はタスクの優先度順となる。

一方、TA_MFIFO,TA_MPRI では、メッセージがメッセージキュー(受信されるのを待つメッセージの待ち行列)に入る際の並び方を指定することができる。属性が TA_MFIFO であればメッセージキューは FIFO となり、属性が TA_MPRI

であればメッセージキューはメッセージの優先度順となる。メッセージの優先度は、メッセージパケットの中の特定領域で指定する。メッセージ優先度は正の値で、1 が最も優先度が高く、数値が大きくなるほど優先度は低くなる。PRI 型で表せる最大の正の値が最も低い優先度となる。同一優先度の場合は FIFO となる。

TA_DSNAME を指定した場合に dsname が有効となり、DS オブジェクト名称として設定される。DS オブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DS のシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname、td_set_dsname を参照のこと。

TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

```
#define TA_TFIFO     0x00000000     /* 待ちタスクを FIFO で管理 */
#define TA_TPRI      0x00000001     /* 待ちタスクを優先度順で管理 */
#define TA_MFIFO     0x00000000     /* メッセージを FIFO で管理 */
#define TA_MPRI      0x00000002     /* メッセージを優先度順で管理 */
#define TA_DSNAME    0x00000040     /* DS オブジェクト名称を指定 */
#define TA_NODISWAI  0x00000080     /* 待ち禁止拒否 */
```

【補足事項】

メールボックスで送受信されるメッセージの本体は共有メモリ上に置かれており、実際に送受信されるのはその先頭アドレスのみである。そのため、タスク固有空間上にメッセージを置くことはできない。

メールボックス削除 tk_del_mbx

tk_del_mbx:Delete Mailbox

【C 言語インタフェース】

```
ER ercd = tk_del_mbx ( ID mbxid ) ;
```

【パラメータ】

ID	mbxid	MailboxID	メールボックス ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mbxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbxid のメールボックスが存在しない)

【解説】

mbxid で示されたメールボックスを削除する。

本システムコールの発行により、対象メールボックスの ID 番号および管理ブロック用の領域などが解放される。

対象メールボックスにおいてメッセージを待っているタスクがあった場合にも、本システムコールは正常終了するが、待ち状態にあったタスクにはエラーE_DLT が返される。また、対象メールボックスの中にメッセージが残っている場合でも、エラーとはならず、メールボックスの削除が行なわれる。

メールボックスへ送信 tk_snd_mbx

tk_snd_mbx:Send Message to Mailbox

【C 言語インタフェース】

```
ER ercd = tk_snd_mbx ( ID mbxid, T_MSG *pk_msg ) ;
```

【パラメータ】

ID	mbxid	MailboxID	メールボックス ID
T_MSG*	pk_msg	Packet of Message	メッセージパケットの先頭アドレス

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mbxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbxid のメールボックスが存在しない)
E_PAR	パラメータエラー(pk_msg が不正、msgpri≦0)

【解説】

mbxid で示された対象メールボックスに、pk_msg を先頭アドレスとするメッセージパケットを送信する。メッセージパケットの内容はコピーされず、受信時には先頭アドレス(pk_msg の値)のみが渡される。

対象メールボックスで既にメッセージを待っているタスクがあった場合には、待ち行列の先頭のタスクの待ち状態が解除され、tk_snd_mbx で指定した pk_msg がそのタスクに送信されて、tk_rcv_mbx のリターンパラメータとなる。一方、対象メールボックスでメッセージを待っているタスクが無ければ、送信されたメッセージは、メールボックスの中のメッセージキュー(メッセージの待ち行列)に入れられる。どちらの場合にも、tk_snd_mbx 発行タスクは待ち状態とはならない。

pk_msg は、メッセージヘッダを含めたメッセージパケットの先頭アドレスである。メッセージヘッダは次の形式となる。

```
typedef struct t_msg {
   ?          ?            /* 内容は実装依存(ただし、固定長) */
} T_MSG;
typedef struct t_msg_pri {
     T_MSG    msgque;     /* メッセージキューのためのエリア */
     PRI      msgpri;     /* メッセージ優先度 */
} T_MSG_PRI;
```

メッセージヘッダは、TA_MFIFO の場合は T_MSG、TA_MPRI の場合は T_MSG_PRI となる。いずれの場合も、メッセージヘッダのサイズは固定長で、sizeof(T_MSG) または sizeof(T_MSG_PRI)で取得する。

実際にメッセージを格納できるのは、メッセージヘッダより後ろの領域となる。メッセージ本体部分のサイズには制限はなく、可変長でもよい。

【補足事項】

tk_snd_mbx によるメッセージ送信は、受信側のタスクの状態とは無関係に行われる。すなわち、非同期のメッセージ送信が行われる。待ち行列につながれるのは、そのタスクの発行したメッセージであって、タスクそのものではない。すなわち、メッセージの待ち行列(メッセージキュー)や受信タスクの待ち行列は存在するが、送信タスクの待ち行列は存在しない。

メールボックスから受信 tk_rcv_mbx

tk_rcv_msg:Receive Message from Mailbox

【C 言語インタフェース】

```
ER ercd = tk_rcv_mbx ( ID mbxid, T_MSG **ppk_msg, TMO tmout ) ;
```

【パラメータ】

ID	mbxid	MailboxID	メールボックス ID
T_MSG**	ppk_msg	Pointer to Packet of Message	リターンパラメータ pk_msg を返す領域へのポインタ
TMO	tmout	Timeout	タイムアウト指定

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード
T_MSG*	pk_msg	Packet of Message	メッセージパケットの先頭アドレス

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mbxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbxid のメールボックスが存在しない)
E_PAR	パラメータエラー(tmout≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象メールボックスが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

【解説】

tk_rcv_mbx では、mbxid で示されたメールボックスからメッセージを受信する。

対象メールボックスにまだメッセージが送信されていない場合(メッセージキューが空の場合)には、本システムコールを発行したタスクは待ち状態となり、メッセージの到着を待つ待ち行列につながれる。一方、対象メールボックスに既にメッセージが入っている場合には、メッセージキューの先頭にあるメッセージを 1 つ取り出して、それをリターンパラメータ pk_msg とする。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。タイムアウト指定が行なわれた場合、待ち解除の条件が満足されない(メッセージが到着しない)まま tmout の時間が経過すると、タイムアウトエラー E_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1 ミリ秒)と同じである。

tmout として TMO_POL＝0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、メッセージがない場合も待ちに入らず E_TMOUT を返す。また、tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずにメッセージが到着するまで待ち続ける。

【補足事項】

pk_msg は、メッセージヘッダを含めたメッセージパケットの先頭アドレスである。メッセージヘッダは TA_MFIFO の場合は T_MSG、TA_MPRI の場合は T_MSG_PRI となる。

メールボックス状態参照 tk_ref_mbx

tk_ref_mbx:Refer Mailbox Status

【C 言語インタフェース】

```
ER ercd = tk_ref_mbx ( ID mbxid, T_RMBX *pk_rmbx ) ;
```

【パラメータ】

ID	mbxid	MailboxID	メールボックス ID
T_RMBX*	pk_rmbx	Packet to Refer Mailbox	メールボックス状態を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_rmbx の内容

VP	exinf	ExtendedInformation	拡張情報
ID	wtsk	WaitTaskInformation	待ちタスクの有無
T_MSG*	pk_msg	Packet of Message	次に受信されるメッセージパケットの先頭アドレス

――(以下に実装独自に他の情報を追加してもよい)――

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mbxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbxid のメールボックスが存在しない)
E_PAR	パラメータエラー(pk_rmbx が不正)

【解説】

mbxid で示された対象メールボックスの各種の状態を参照し、リターンパラメータとして次に受信されるメッセージ(メッセージキューの先頭のメッセージ)、待ちタスクの有無(wtsk)、拡張情報(exinf)を返す。

wtsk は、このメールボックスで待っているタスクの ID を示す。このメールボックスで複数のタスクが待っている場合には、待ち行列の先頭のタスクの ID を返す。待ちタスクが無い場合は wtsk＝0 となる。

対象メールボックスが存在しない場合には、エラーE_NOEXS となる。

pk_msg は、次に tk_rcv_mbx を実行した場合に受信されるメッセージである。メッセージキューにメッセージが無い時は、pk_msg＝NULL となる。また、どんな場合でも、pk_msg＝NULL と wtsk＝0 の少なくとも一方は成り立つ。

4.5 拡張同期・通信機能

拡張同期・通信機能は、タスクとは独立したオブジェクトにより、タスク間の高度な同期・通信を行うための機能である。ミューテックス、メッセージバッファ、ランデブポートの各機能が含まれる。

4.5.1 ミューテックス

ミューテックスは、共有資源を使用する際にタスク間で排他制御を行うためのオブジェクトである。ミューテックスは、排他制御に伴う上限のない優先度逆転を防ぐための機構として、優先度継承プロトコル(priority inheritance protocol)と優先度上限プロトコル(priority ceiling protocol)をサポートする。

ミューテックス機能には、ミューテックスを生成／削除する機能、ミューテックスをロック／ロック解除する機能、ミューテックスの状態を参照する機能が含まれる。ミューテックスは ID 番号で識別されるオブジェクトである。ミューテックスの ID 番号をミューテックス ID と呼ぶ。

ミューテックスは、ロックされているかどうかの状態と、ロックを待つタスクの待ち行列を持つ。また、カーネルは、各ミューテックスに対してそれをロックしているタスクを、各タスクに対してそれがロックしているミューテックスの集合を管理する。タスクは、資源を使用する前に、ミューテックスをロックする。ミューテックスが他のタスクにロックされていた場合には、ミューテックスがロック解除されるまで、ミューテックスのロック待ち状態となる。ミューテックスのロック待ち状態になったタスクは、そのミューテックスの待ち行列につながれる。タスクは、資源の使用を終えると、ミューテックスのロックを解除する。

ミューテックスは、ミューテックス属性に TA_INHERIT(＝0x02)を指定することにより優先度継承プロトコルを、TA_CEILING(＝0x03)を指定することにより優先度上限プロトコルをサポートする。TA_CEILING 属性のミューテックスに対しては、そのミューテックスをロックする可能性のあるタスクの中で最も高いベース優先度を持つタスクのベース優先度を、ミューテックス生成時に上限優先度として設定する。TA_CEILING 属性のミューテックスを、その上限優先度よりも高いベース優先度を持つタスクがロックしようとした場合、 E_ILUSE エラーとなる。また、TA_CEILING 属性のミューテックスをロックしているかロックを待っているタスクのベース優先度を、tk_chg_pri によってそのミューテックスの上限優先度よりも高く設定しようとした場合、tk_chg_pri が E_ILUSE エラーを返す。

これらのプロトコルを用いた場合、上限のない優先度逆転を防ぐために、ミューテックスの操作に伴ってタスクの現在優先度を変更する。優先度継承プロトコルと優先度上限プロトコルに厳密に従うなら、タスクの現在優先度を、次に挙げる優先度の最高値に常に一致するように変更する必要がある。これを、厳密な優先度制御規則と呼ぶ。

- タスクのベース優先度
- タスクが TA_INHERIT 属性のミューテックスをロックしている場合、それらのミューテックスのロックを待っているタスクの中で、最も高い現在優先度を持つタスクの現在優先度
- タスクが TA_CEILING 属性のミューテックスをロックしている場合、それらのミューテックス中で、最も高い上限優先度を持つミューテックスの上限優先度

ここで、TA_INHERIT 属性のミューテックスを待っているタスクの現在優先度が、ミューテックス操作か tk_chg_pri によるベース優先度の変更に伴って変更された場合、そのミューテックスをロックしているタスクの現在優先度の変更が必要になる場合がある。これを推移的な優先度継承と呼ぶ。さらにそのタスクが、別の TA_INHERIT 属性のミューテックスを待っていた場合には、そのミューテックスをロックしているタスクに対して推移的な優先度継承の処理が必要になる場合がある。

T-Kernel 仕様では、上述の厳密な優先度制御規則に加えて、現在優先度を変更する状況を限定した優先度制御規則(これを簡略化した優先度制御規則と呼ぶ)を規定し、どちらを採用するかは実装定義とする。具体的には、簡略化した優先度制御規則においては、タスクの現在優先度を高くする方向の変更はすべて行うのに対して、現在優先度を低くする方向の変更は、タスクがロックしているミューテックスがなくなった時にのみ行う(この場合には、タスクの現在優先度をベース優先度に戻すことになる)。より具体的には、次の状況でのみ現在優先度を変更する処理を行えばよい。

- タスクがロックしている TA_INHERIT 属性のミューテックスを、そのタスクよりも高い現在優先度を持つタスクが待ち始めた時
- タスクがロックしている TA_INHERIT 属性のミューテックスを待っているタスクが、前者のタスクよりも高い現在優先度に変更された時
- タスクが、そのタスクの現在優先度よりも高い上限優先度を持つ TA_CEILING 属性のミューテックスをロックした時
- タスクがロックしているミューテックスがなくなった時

ミューテックスの操作に伴ってタスクの現在優先度を変更した場合には、次の処理を行う。
優先度を変更されたタスクが実行できる状態である場合、タスクの優先順位を、変更後の優先度にしたがって変化させる。変更後の優先度と同じ優先度を持つタスクの間での優先順位は、実装依存である。優先度が変更されたタスクが何らかのタスク優先度順の待ち行列につながれている場合にも、その待ち行列の中での順序を、変更後の優先度にしたがって変化させる。変更後の優先度と同じ優先度を持つタスクの間での順序は、実装依存である。タスクが終了する時に、そのタスクがロックしているミューテックスが残っている場合には、それらのミューテックスをすべてロック解除する。ロックしているミューテックスが複数ある場合には、それらをロック解除する順序は実装依存である。ロック解除の具体的な処理内容については、tk_unl_mtx の機能説明を参照すること。

【補足事項】
TA_TFIFO 属性または TA_TPRI 属性のミューテックスは、最大資源数が 1 のセマフォ(バイナリセマフォ)と同等の機能を持つ。ただし、ミューテックスは、ロックしたタスク以外はロック解除できない、タスク終了時に自動的にロック解除されるなどの違いがある。
ここでいう優先度上限プロトコルは、広い意味での優先度上限プロトコルで、最初に優先度上限プロトコルとして提案されたアルゴリズムではない。厳密には、highest locker protocol などと呼ばれているアルゴリズムである。
ミューテックスの操作に伴ってタスクの現在優先度を変更した結果、優先度を変更されたタスクのタスク優先度順の待ち行列の中での順序が変化した場合、優先度を変更されたタスクないしはその待ち行列で待っている他のタスクの待ち解除が必要になる場合がある。

【仕様決定の理由】
ミューテックスの操作に伴ってタスクの現在優先度を変更した場合に、変更後の優先度と同じ優先度を持つタスクの間での優先順位を実装依存としたのは、次の理由による。アプリケーションによっては、ミューテックス機能による現在優先度の変更が頻繁に発生する可能性があり、それに伴ってタスク切替えが多発するのは望ましくない(同じ優先度を持つタスクの間での優先順位を最低とすると、不必要なタスク切替えが起こる)。理想的には、タスクの優先度ではなく優先順位を継承するのが望ましいが、このような仕様にすると実装上のオーバヘッドが大きくなるため、実装依存とすることにした。

ミューテックス生成 tk_cre_mtx

tk_cre_mtx:Create Mutex

【C 言語インタフェース】

```
ID mtxid = tk_cre_mtx ( T_CMTX *pk_cmtx ) ;
```

【パラメータ】

T_CMTX*	pk_cmtx	Packet to Create Mutex	ミューテックス生成情報

pk_cmtx の内容

VP	exinf	ExtendedInformation	拡張情報
ATR	mtxatr	MutexAttribute	ミューテックス属性
PRI	ceilpri	Ceiling Priority of Mutex	ミューテックスの上限優先度
UB	dsname[8]	DS Object name	DS オブジェクト名称

――(以下に実装独自に他の情報を追加してもよい)――

【リターンパラメータ】

ID	mtxid	MutexID	ミューテックス ID
	または	ErrorCode	エラーコード

【エラーコード】

E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_LIMIT	ミューテックスの数がシステムの上限を超えた
E_RSATR	予約属性(mtxatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_cmtx, ceilpri が不正)

【解説】

ミューテックスを生成しミューテックス ID 番号を割当てる。具体的には、生成するミューテックスに対して管理ブロックなどを割り付ける。

exinf は、対象ミューテックスに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、tk_ref_mtx で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。OS では exinf の内容について関知しない。

mtxatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。mtxatr のシステム属性の部分では、次のような指定を行う。

```
mtxatr:=  (TA_TFIFO || TA_TPRI || TA_INHERIT || TA_CEILING)
          | [TA_DSNAME] | [TA_NODISWAI]
```

TA_TFIFO	待ちタスクのキューイングは FIFO
TA_TPRI	待ちタスクのキューイングは優先度順
TA_INHERIT	優先度継承プロトコル
TA_CEILING	優先度上限プロトコル
TA_DSNAME	DS オブジェクト名称を指定する
TA_NODISWAI	tk_dis_wai による待ち禁止を拒否する

TA_TFIFO の場合、ミューテックスのタスクの待ち行列は FIFO となる。TA_TPRI,TA_INHERIT,TA_CEILING では、タスクの優先度順となる。TA_INHERIT では優先度継承プロトコル、TA_CEILING では優先度上限プロトコルが適用される。

TA_CEILING の場合のみ ceilpri が有効となり、ミューテックスの上限優先度を設定する。

TA_DSNAME を指定した場合に dsname が有効となり、DS オブジェクト名称として設定される。DS オブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DS のシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname、td_set_dsname を参照のこと。TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

```
#define TA_TFIFO        0x00000000      /* 待ちタスクを FIFO で管理 */
#define TA_TPRI         0x00000001      /* 待ちタスクを優先度順で管理 */
#define TA_INHERIT      0x00000002      /* 優先度継承プロトコル */
#define TA_CEILING      0x00000003      /* 優先度上限プロトコル */
#define TA_DSNAME       0x00000040      /* DS オブジェクト名称を指定 */
#define TA_NODISWAI     0x00000080      /* 待ち禁止拒否 */
```

ミューテックス削除 tk_del_mtx

tk_del_mtx:Delete Mutex

【C 言語インタフェース】

```
ER ercd = tk_del_mtx ( ID mtxid ) ;
```

【パラメータ】

ID	mtxid	MutexID	ミューテックス ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mtxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mtxid のミューテックスが存在しない)

【解説】

mtxid で示されたミューテックスを削除する。

本システムコールの発行により、対象ミューテックスの ID 番号および管理ブロック用の領域は解放される。

対象ミューテックスにおいてロック待ちしているタスクがあった場合にも､本システムコールは正常終了するが､待ち状態にあったタスクにはエラーE_DLT が返される。

ミューテックスが削除されると、そのミューテックスをロックしているタスクにとっては、ロックしているミューテックスが減ることになる。したがって、削除されるミューテックスが TA_INHERIT または TA_CEILING 属性の場合には、ロックしていたタスクの優先度が変更される場合がある。

ミューテックスのロック tk_loc_mtx

tk_loc_mtx:Lock Mutex

【C 言語インタフェース】

```
ER ercd = tk_loc_mtx ( ID mtxid, TMO tmout ) ;
```

【パラメータ】

ID	mtxid	MutexID	ミューテックス ID
TMO	tmout	Timeout	タイムアウト指定

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mtxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mtxid のミューテックスが存在しない)
E_PAR	パラメータエラー(tmout≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象ミューテックスが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)
E_ILUSE	不正使用(多重ロック、上限優先度違反)

【解説】

mtxid のミューテックスをロックする。ミューテックスがロックできれば、本システムコールの発行タスクは待ち状態に入らず、実行を継続する。この場合、そのミューテックスはロック状態になる。ロックできなければ、本システムコールを発行したタスクは待ち状態に入る。すなわち、そのミューテックスに対する待ち行列につながれる。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。待ち解除の条件が満足されない(ロックが解除されない)まま tmout の時間が経過すると、タイムアウトエラーE_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1 ミリ秒)と同じである。

tmout として TMO_POL＝0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、ロックできなくても待ちに入らず E_TMOUT を返す。また、tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずにロックできるまで待ち続ける。

自タスクがすでに対象ミューテックスをロックしていた場合には、E_ILUSE(多重ロック)を返す。

対象ミューテックスが TA_CEILING 属性の場合、自タスクのベース優先度(*1)が対象ミューテックスの上限優先度より高い場合には E_ILSUE(上限優先度違反)を返す。

(*1)ベース優先度：ミューテックスによって自動的に引き上げられる前のタスクの優先度を示す。最後(ミューテックスのロック中も含む)に tk_chg_pri によって設定された優先度、または tk_chg_pri を一度も発行していない場合はタスク生成時に指定したタスク優先度が、ベース優先度である。

【補足事項】

・TA_INHERIT 属性のミューテックスの場合

自タスクがロック待ち状態になる場合、そのミューテックスをロックしているタスクの現在優先度が自タスクより低ければ、ロックしているタスクの優先度を自タスクと同じ優先度まで引き上げる。ロックを待っているタスクがロックを獲得せずに待ちを終了した場合(タイムアウトなど)、そのミューテックスをロック中のタスクの優先度を、次の内の最も高い優先度まで引き下げる。ただし、この優先度の引き下げを行うか否かは実装依存である。

(a) そのミューテックスでロック待ちしているタスクの現在優先度の内の最も高い優先度。
(b) そのミューテックスをロック中のタスクがロックしている他のすべてのミューテックスの内の最も高い優先度。
(c) ロック中のタスクのベース優先度。

・TA_CEILING 属性のミューテックスの場合

自タスクがロックを獲得した場合、自タスクの現在優先度がミューテックスの上限優先度より低ければ、自タスクの優先度をミューテックスの上限優先度まで引き上げる。

ミューテックスのアンロック tk_unl_mtx

tk_unl_mtx:UnLock Mutex

【C 言語インタフェース】

```
ER ercd = tk_unl_mtx ( ID mtxid ) ;
```

【パラメータ】

ID	mtxid	MutexID	ミューテックス ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mtxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mtxid のミューテックスが存在しない)
E_ILUSE	不正使用(自タスクがロックしたミューテックスではない)

【解説】
mtxid のミューテックスのロックを解除する。
ロック待ちしているタスクがあれば、待ち行列の先頭のタスクの待ちを解除し、そのタスクをロック獲得状態にする。
自タスクがロックしていないミューテックスを指定した場合、E_ILUSE を返す。

【補足事項】
ロック解除したミューテックスが TA_INHERIT または TA_CEILING 属性の場合、次のようにタスク優先度を引き下げる必要がある。
ロックを解除することにより、自タスクがロックしているミューテックスがすべてなくなった場合は、自タスクの優先度をベース優先度まで引き下げる。
自タスクがロック中のミューテックスが残っている場合、自タスクの優先度を次の内の最も高い優先度まで引き下げる。

(a)ロックしているすべてのミューテックスの内の最も高い優先度
(b)ベース優先度

ただし、ロック中のミューテックスが残っている場合の優先度の引き下げを行うか否かは実装依存である。
ミューテックスをロックした状態でタスクを終了した(休止状態(DORMANT)または未登録状態(NON-EXISTENT)になった)場合、当該タスクがロックしているすべてのミューテックスは T-Kernel によって自動的にロック解除される。

ミューテックス状態参照 tk_ref_mtx

tk_ref_mtx:Refer Mutex Status

【C 言語インタフェース】

```
ER ercd = tk_ref_mtx ( ID mtxid, T_RMTX *pk_rmtx ) ;
```

【パラメータ】

ID	mtxid	Mutex ID	ミューテックス ID
T_RMTX*	pk_rmtx	Packet to Refer Mutex	ミューテックス状態を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	Error Code	エラーコード
pk_rmtx の内容			
VP	exinf	ExtendedInformation	拡張情報
ID	htsk	LockingTaskID	ロックしているタスクの ID
ID	wtsk	LockWaitTaskID	ロック待ちタスクの ID
――(以下に実装独自に他の情報を追加してもよい)――			

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mtxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mtxid のミューテックスが存在しない)
E_PAR	パラメータエラー(pk_rmtx が不正)

【解説】

mtxid で示された対象ミューテックスの各種の状態を参照し、リターンパラメータとして
ロック中のタスク(htsk)、ロック待ちタスク(wtsk)、拡張情報(exinf)を返す。

htsk は、このミューテックスをロックしているタスクの ID を示す。ロックしているタスクがない場合は htsk＝0 となる。

wtsk は、このミューテックスで待っているタスクの ID を示す。複数のタスクが待っている場合には、待ち行列の先頭のタスクの ID を返す。待ちタスクが無い場合は wtsk＝0 となる。

対象ミューテックスが存在しない場合には、エラーE_NOEXS となる。

4.5.2 メッセージバッファ

メッセージバッファは、可変長のメッセージを受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である。メッセージバッファ機能には、メッセージバッファを生成／削除する機能、メッセージバッファに対してメッセージを送信／受信する機能、メッセージバッファの状態を参照する機能が含まれる。メッセージバッファは ID 番号で識別されるオブジェクトである。メッセージバッファの ID 番号をメッセージバッファ ID と呼ぶ。

メッセージバッファは、メッセージの送信を待つタスクの待ち行列(送信待ち行列)とメッセージの受信を待つタスクの待ち行列(受信待ち行列)を持つ。また、送信されたメッセージを格納するためのメッセージバッファ領域を持つ。メッセージを送信する側(イベントを知らせる側)では、送信したいメッセージをメッセージバッファにコピーする。メッセージバッファ領域の空き領域が足りなくなった場合、メッセージバッファ領域に十分な空きができるまでメッセージバッファへの送信待ち状態になる。

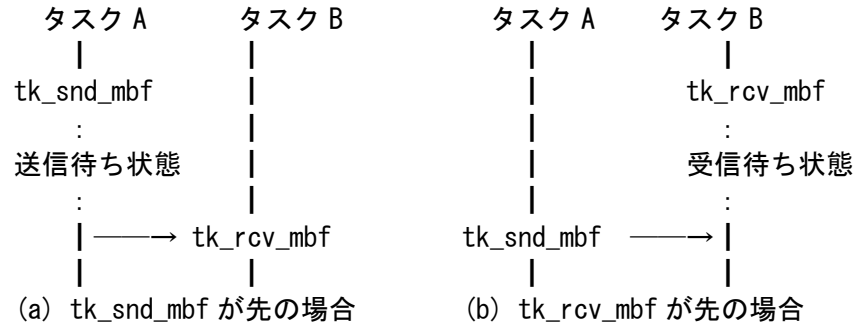
メッセージバッファへの送信待ち状態になったタスクは､そのメッセージバッファの送信待ち行列につながれる。一方、メッセージを受信する側(イベントを待つ側)では、メッセージバッファに入っているメッセージを一つ取り出す。メッセージバッファにメッセージが入っていない場合は、次にメッセージが送られてくるまでメッセージバッファからの受信待ち状態になる。メッセージバッファからの受信待ち状態になったタスクは、そのメッセージバッファの受信待ち行列につながれる。

メッセージバッファ領域のサイズを 0 にすることで、同期メッセージ機能を実現することができる。すなわち、送信側のタスクと受信側のタスクが、それぞれ相手のタスクがシステムコールを呼び出すのを待ち合わせ、両者がシステムコールを呼出した時点で、メッセージの受渡しが行われる。

【補足事項】

メッセージバッファ領域のサイズを 0 にした場合の､メッセージバッファの動作を[図 10]の例を用いて説明する。この図で、タスク A とタスク B は非同期に実行しているものとする。

- もしタスク A が先に tk_snd_mbf を呼出した場合には、タスク B が tk_rcv_mbf を呼び出すまでタスク A は待ち状態となる。この時タスク A は、メッセージバッファへの送信待ち状態になっている[図 10(a)]。
- 逆にタスク B が先に tk_rcv_mbf を呼出した場合には､タスク A が tk_snd_mbf を呼び出すまでタスク B は待ち状態となる。この時タスク B は、メッセージバッファからの受信待ち状態になっている[図 10(b)]。
- タスク A が tk_snd_mbf を呼出し、タスク B が tk_rcv_mbf を呼出した時点で、タスク A からタスク B へメッセージの受渡しが行われる。その後は、両タスクとも実行できる状態となる。



[図 10] メッセージバッファによる同期通信

メッセージバッファへの送信を待っているタスクは、待ち行列につながれている順序でメッセージを送信する。例えば、あるメッセージバッファに対して 40 バイトのメッセージを送信しようとしているタスク A と、10 バイトのメッセージを送信しようとしているタスク B が、この順で待ち行列につながれている時に、別のタスクによるメッセージの受信により 20 バイトの空き領域ができたとする。このような場合でも、タスク A がメッセージを送信するまで、タスク B はメッセージを送信できない。

メッセージバッファは、可変長のメッセージをコピーして受渡しする。メールボックスとの違いは、メッセージをコピーすることである。

メッセージバッファは、リングバッファで実装することを想定している。

メッセージバッファ生成 tk_cre_mbf

tk_cre_mbf:Create MessageBuffer

【C 言語インタフェース】

```
ID mbfid = tk_cre_mbf ( T_CMBF *pk_cmbf ) ;
```

【パラメータ】

T_CMBF* pk_cmbf　Packet to Create MessageBuffer　メッセージバッファ生成情報

pk_cmbf の内容

VP	exinf	ExtendedInformation	拡張情報
ATR	mbfatr	MessageBufferAttribute	メッセージバッファ属性
INT	bufsz	BufferSize	メッセージバッファのサイズ(バイト数)
INT	maxmsz	MaxMessageSize	メッセージの最大長(バイト数)
UB	dsname[8]	DS Object name	DS オブジェクト名称

――(以下に実装独自に他の情報を追加してもよい)――

【リターンパラメータ】

ID	mbfid	MessageBufferID	メッセージバッファ ID
	または	ErrorCode	エラーコード

【エラーコード】

E_NOMEM	メモリ不足(管理ブロックやリングバッファ用の領域が確保できない)
E_LIMIT	メッセージバッファの数がシステムの上限を超えた
E_RSATR	予約属性(mbfatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_cmbf が不正, bufsz, maxmsz が負または不正)

【解説】

メッセージバッファを生成しメッセージバッファ ID 番号を割当てる。具体的には、生成するメッセージバッファに対して管理ブロックを割り付ける。また、bufsz の情報を元に、メッセージキュー(受信されるのを待つメッセージの待ち行列)として利用するためのリングバッファの領域を確保する。

メッセージバッファは、可変長メッセージの送受信の管理を行うオブジェクトである。メールボックス(mbx)との違いは、送信時と受信時に可変長のメッセージ内容がコピーされるということである。また、バッファが一杯の場合に、メッセージ送信側も待ち状態に入る機能がある。

exinf は、対象メッセージバッファに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、tk_ref_mbf で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。OS では exinf の内容について関知しない。

mbfatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。mbfatr のシステム属性の部分では、次のような指定を行う。

```
mbfatr:= (TA_TFIFO || TA_TPRI) | [TA_DSNAME] | [TA_NODISWAI]
```

TA_TFIFO	送信待ちタスクのキューイングは FIFO
TA_TPRI	送信待ちタスクのキューイングは優先度順
TA_DSNAME	DS オブジェクト名称を指定する
TA_NODISWAI	tk_dis_wai による待ち禁止を拒否する

TA_TFIFO, TA_TPRI では、バッファが一杯の場合にメッセージを送信するタスクがメッセージバッファの待ち行列に並ぶ際の並び方を指定することができる。属性が TA_TFIFO であればタスクの待ち行列は FIFO となり、属性が TA_TPRI であればタスクの待ち行列はタスクの優先度順となる。なお、メッセージキューの順序は FIFO のみである。

メッセージ受信待ちのタスクの待ち行列の順序は FIFO のみである。

TA_DSNAME を指定した場合に dsname が有効となり、DS オブジェクト名称として設定される。DS オブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DS のシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname、td_set_dsname を参照のこと。TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

```
#define TA_TFIFO      0x00000000    /* 送信待ちタスクを FIFO で管理 */
#define TA_TPRI       0x00000001    /* 送信待ちタスクを優先度順で管理 */
#define TA_DSNAME     0x00000040    /* DS オブジェクト名称を指定 */
#define TA_NODISWAI   0x00000080    /* 待ち禁止拒否 */
```

【補足事項】

送信待ちのタスクが複数あった場合、バッファの空きができて送信待ちが解除されるのは常に待ち行列の順となる。

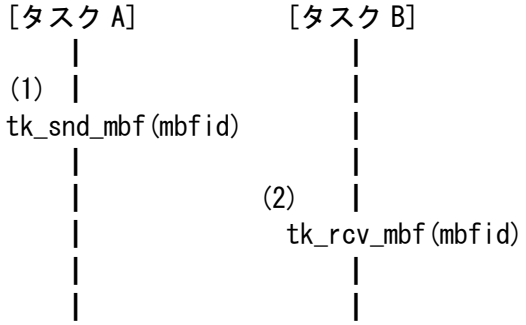
例えば、30 バイトのメッセージを送信しようとしているタスク A と、10 バイトのメッセージを送信しようとしているタスク B が A - B の順で待っていた場合、メッセージバッファに 20 バイトの空きができても A のタスクを追い越して B のタスクが先に送信することはない。

メッセージキューを入れるリングバッファの中には、一つ一つのメッセージを管理する情報も入るため、bufsz で指定されたリングバッファのサイズとキューに入るメッセージのサイズの合計とは、一般には一致しない。後者の方が小さい値をとるのが普通である。その意味で、bufsz の情報は厳密な意味をもつものではない。

bufsz＝0 のメッセージバッファを生成することは可能である。この場合、このメッセージバッファでは送受信側が完全に同期した通信を行うことになる。すなわち、tk_snd_mbf と tk_rcv_mbf の一方のシステムコールが先に実行されると、それを実行したタスクは待ち状態となる。もう一方のシステムコールが実行された段階で、メッセージの受け渡し(コピー)が行われ、その後双方のタスクが実行を再開する。

bufsz＝0 のメッセージバッファの場合、具体的な動作は次のようになる。

1) [図 11]で、タスク A とタスク B は非同期に動いている。もし、タスク A が先に(1)に到達し、tk_snd_mbf(mbfid)を実行した場合には、タスク B が(2)に到達するまで タスク A はメッセージ送信待ち状態になる。この状態のタスク A を対象として tk_ref_tsk を発行すると、tskwait＝TTW_SMBF となる。逆に、タスク B が先に(2)に到達し、tk_rcv_mbf(mbfid)を実行した場合には、タスク A が(1)に到達するまで タスク B はメッセージ受信待ち状態になる。この状態のタスク B を対象として tk_ref_tsk を発行すると、tskwait＝TTW_RMBF となる。
2) タスク A が tk_snd_mbf(mbfid)を実行し、かつタスク B が tk_rcv_mbf(mbfid)を実行した時点でタスク A からタスク B にメッセージが送信され、どちらのタスクも待ち解除となって実行を再開する。



(1)で待ちに入る場合がメッセージ送信待ち(TTW_SMBF)
(2)で待ちに入る場合がメッセージ受信待ち(TTW_RMBF)

[図 11] bufsz＝0 のメッセージバッファを使った同期式通信

メッセージバッファ削除 tk_del_mbf

tk_del_mbf:Delete MessageBuffer

【C 言語インタフェース】

```
ER ercd = tk_del_mbf ( ID mbfid ) ;
```

【パラメータ】

ID	mbfid	MessageBufferID	メッセージバッファ ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mbfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbfid のメッセージバッファが存在しない)

【解説】

mbfid で示されたメッセージバッファを削除する。
本システムコールの発行により､対象メッセージバッファの ID 番号および管理ブロック用の領域およびメッセージを入れるバッファ領域は解放される。
対象メッセージバッファにおいてメッセージ受信またはメッセージ送信を待っているタスクがあった場合にも、本システムコールは正常終了するが、待ち状態にあったタスクにはエラーE_DLT が返される。また、対象メッセージバッファの中にメッセージが残っている場合でも、エラーとはならず、メッセージバッファの削除が行なわれ、中にあったメッセージは消滅する。

メッセージバッファへ送信 tk_snd_mbf

tk_snd_mbf:Send Message to MessageBuffer

【C 言語インタフェース】

```
ER ercd = tk_snd_mbf ( ID mbfid, VP msg, INT msgsz, TMO tmout ) ;
```

【パラメータ】

ID	mbfid	MessageBufferID	メッセージバッファ ID
INT	msgsz	SendMessageSize	送信メッセージのサイズ(バイト数)
VP	msg	SendMessage	送信メッセージの先頭アドレス
TMO	tmout	Timeout	タイムアウト指定

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mbfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbfid のメッセージバッファが存在しない)
E_PAR	パラメータエラー(msgsz≦0, msgsz＞maxmsz, msg が不正、 tmout≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象メッセージバッファが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

【解説】

tk_snd_mbf では、mbfid で示されたメッセージバッファに対して、msg のアドレスに入っているメッセージを送信する。メッセージのサイズは msgsz で指定される。すなわち、msg 以下の msgsz バイトが、mbfid で指定されたメッセージバッファのメッセージキューにコピーされる。メッセージキューは、リングバッファによって実現されることを想定している。

msgsz が、tk_cre_mbf で指定した maxmsz よりも大きい場合は、エラー E_PAR となる。

バッファの空き領域が少なく、msg のメッセージをメッセージキューに入れられない場合、本システムコールを発行したタスクはメッセージ送信待ち状態となり、バッファの空きを待つための待ち行列(送信待ち行列)につながれる。待ち行列の順序は tk_cre_mbf 時の指定により FIFO またはタスク優先度順となる。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。タイムアウト指定が行なわれた場合、待ち解除の条件が満足されない(バッファに十分な空き領域ができない)まま tmout の時間が経過すると、タイムアウトエラーE_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1 ミリ秒)と同じである。

tmout として TMO_POL＝0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、バッファに十分な空きがない場合は待ちに入らず E_TMOUT を返す。また、tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずにバッファに空きができるまで待ち続ける。

長さが 0 のメッセージは送信することができない。msgsz≦0 の場合には、エラーE_PAR となる。

タスク独立部やディスパッチ禁止状態から実行した場合はエラーE_CTX となるが、tmout＝TMO_POL の場合は、実装によってはタスク独立部やディスパッチ禁止状態から実行することができる場合がある。

メッセージバッファから受信 tk_rcv_mbf

tk_rcv_mbf:Receive Message from MessageBuffer

【C 言語インタフェース】

```
INT msgsz = tk_rcv_mbf ( ID mbfid, VP msg, TMO tmout ) ;
```

【パラメータ】

ID	mbfid	MessageBufferID	メッセージバッファ ID
VP	msg	ReceiveMessage	受信メッセージを入れるアドレス
TMO	tmout	Timeout	タイムアウト指定

【リターンパラメータ】

INT	msgsz	ReceiveMessageSize	受信したメッセージのサイズ(バイト数)
	または	ErrorCode	エラーコード

【エラーコード】

E_ID	不正 ID 番号(mbfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbfid のメッセージバッファが存在しない)
E_PAR	パラメータエラー(msg が不正、tmout≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象メッセージバッファが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

【解説】

tk_rcv_mbf では、mbfid で示されたメッセージバッファからメッセージを受信し、msg で指定した領域に入れる。すなわち、mbfid で指定されたメッセージバッファのメッセージキューの先頭のメッセージの内容を、msg 以下の msgsz バイトにコピーする。

mbfid で示されたメッセージバッファにまだメッセージが送信されていない場合(メッセージキューが空の場合)には、本システムコールを発行したタスクは待ち状態となり、メッセージの到着を待つ待ち行列(受信待ち行列)につながれる。受信待ちタスクの待ち行列は FIFO のみである。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。タイムアウト指定が行なわれた場合、待ち解除の条件が満足されない(メッセージが到着しない)まま tmout の時間が経過すると、タイムアウトエラー E_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1 ミリ秒)と同じである。

tmout として TMO_POL＝0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、メッセージがない場合にも待ちに入らず E_TMOUT を返す。また、tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずにメッセージが到着するまで待ち続ける。

メッセージバッファ状態参照 tk_ref_mbf

tk_ref_mbf:Get MessageBuffer Status

【C 言語インタフェース】

```
ER ercd = tk_ref_mbf ( ID mbfid, T_RMBF *pk_rmbf ) ;
```

【パラメータ】

ID	mbfid	MessageBufferID	メッセージバッファ ID
T_RMBF*	pk_rmbf	Packet to Refer MessageBuffer	メッセージバッファ状態を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_rmbf の内容

VP	exinf	ExtendedInformation	拡張情報
ID	wtsk	WaitTaskInformation	受信待ちタスクの有無
ID	stsk	SendTaskInformation	送信待ちタスクの有無
INT	msgsz	MessageSize	次に受信されるメッセージのサイズ(バイト数)
INT	frbufsz	FreeBufferSize	空きバッファのサイズ(バイト数)
INT	maxmsz	MaximumMessageSize	メッセージの最大長(バイト数)

――(以下に実装独自に他の情報を追加してもよい)――

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mbfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbfid のメッセージバッファが存在しない)
E_PAR	パラメータエラー(pk_rmbf が不正)

【解説】

mbfid で示された対象メッセージバッファの各種の状態を参照し、リターンパラメータとして送信待ちタスクの有無(stsk)、次に受信されるメッセージのサイズ(msgsz)、空きバッファのサイズ(frbufsz)、メッセージの最大長(maxmsz)、受信待ちタスクの有無(wtsk)、拡張情報(exinf)を返す。

wtsk は、このメッセージバッファで受信待ちしているタスクの ID を示す。また、stsk は送信待ちしているタスクの ID を示す。このメッセージバッファで複数のタスクが待っている場合には、待ち行列の先頭のタスクの ID を返す。待ちタスクが無い場合は 0 となる。

対象メッセージバッファが存在しない場合には、エラー E_NOEXS となる。

msgsz には、メッセージキューの先頭のメッセージ(次に受信されるメッセージ)のサイズが返る。メッセージキューにメッセージが無い場合には、msgsz＝0 となる。なお、サイズが 0 のメッセージを送ることはできない。

どんな場合でも、msgsz＝0 と wtsk＝0 の少なくとも一方は成り立つ。

frbufsz は、メッセージキューを構成するリングバッファの空き領域のサイズを示すものである。この値は、あとどの程度の量のメッセージを送信できるかを知る手掛かりになる。

maxmsz には、tk_cre_mbf で指定したメッセージの最大長が返される。

4.5.3 ランデブポート

ランデブ機能は、タスク間で同期通信を行うための機能で、あるタスクから別のタスクへの処理依頼と、後者のタスクから前者のタスクへの処理結果の返却を、一連の手順としてサポートする。双方のタスクが待ち合わせるためのオブジェクトを、ランデブポートと呼ぶ。ランデブ機能は、典型的にはクライアント／サーバモデルのタスク間通信を実現するために用いられるが、クライアント／サーバモデルよりも柔軟な同期通信モデルを提供するものである。

ランデブ機能には、ランデブポートを生成／削除する機能、ランデブポートに対して処理の依頼を行う機能(ランデブの呼出し)、ランデブポートで処理依頼を受け付ける機能(ランデブの受付)、処理結果を返す機能(ランデブの終了)、受け付けた処理依頼を他のランデブポートに回送する機能(ランデブの回送)ランデブポートおよびランデブの状態を参照する機能が含まれる。ランデブポートは ID 番号で識別されるオブジェクトである。ランデブポートの ID 番号をランデブポート ID と呼ぶ。

ランデブポートに対して処理依頼を行う側のタスク(クライアント側のタスク)は、ランデブポートとランデブ条件、依頼する処理に関する情報を入れたメッセージ(これを呼出しメッセージと呼ぶ)を指定して、ランデブの呼出しを行う。一方、ランデブポートで処理依頼を受け付ける側のタスク(サーバ側のタスク)は、ランデブポートとランデブ条件を指定して、ランデブの受付を行う。

ランデブ条件は、ビットパターンで指定する。あるランデブポートに対して、呼出したタスクのランデブ条件のビットパターンと、受け付けたタスクのランデブ条件のビットパターンをビット毎に論理積をとり、結果が 0 以外の場合にランデブが成立する。ランデブを呼出したタスクは、ランデブが成立するまでランデブ呼出し待ち状態となる。逆に、ランデブを受け付けるタスクは、ランデブが成立するまでランデブ受付待ち状態となる。

ランデブが成立すると、ランデブを呼出したタスクから受け付けたタスクへ、呼出しメッセージが渡される。ランデブを呼出したタスクはランデブ終了待ち状態へ移行し、依頼した処理が完了するのを待つ。一方、ランデブを受け付けたタスクは待ち解除され、依頼された処理を行う。ランデブを受け付けたタスクが依頼された処理を完了すると、処理結果を返答メッセージの形で呼出したタスクに渡し、ランデブを終了する。この時点で、ランデブを呼出したタスクが、ランデブ終了待ち状態から待ち解除される。

ランデブポートは、ランデブ呼出し待ち状態のタスクをつなぐための呼出し待ち行列と、ランデブ受付待ち状態のタスクをつなぐための受付待ち行列を持つ。それに対して、ランデブが成立した後は、ランデブした双方のタスクはランデブポートから切り離される。すなわち、ランデブポートは、ランデブ終了待ち状態のタスクをつなぐための待ち行列は持たない。また、ランデブを受け付け、依頼された処理を実行しているタスクに関する情報も持たない。

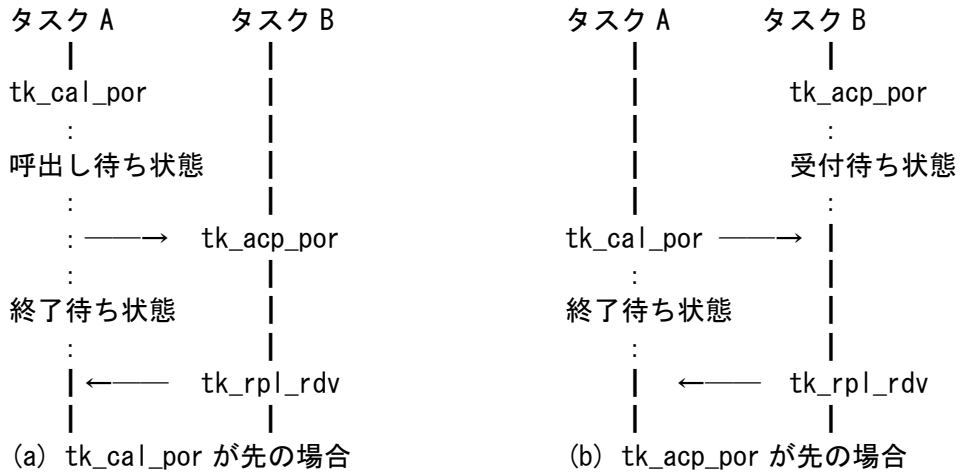
カーネルは、同時に成立しているランデブを識別するために、オブジェクト番号を付与する。ランデブのオブジェクト番号をランデブ番号と呼ぶ。ランデブ番号の付与方法は実装依存であるが、少なくとも、ランデブを呼出したタスクを指定するための情報を含んでいなければならない。また、同じタスクが呼出したランデブであっても、1 回目のランデブと 2 回目のランデブで異なるランデブ番号を付与するなど、できる限りユニークにしなければならない。

【補足事項】

ランデブ機能は、ADA の言語仕様に導入されている同期／通信機能であり、その元になっているのは CSP(Communicating Sequential Processes)である。ただし、ADA のランデブ機能は言語仕様の一部であり、リアルタイムカーネル仕様である T-Kernel 仕様が提供するランデブ機能とは位置付けが異なる。具体的には、リアルタイムカーネルが提供するランデブ機能は、言語のランデブ機能を実現するためのプリミティブに位置付けられるものであるが、ADA のランデブ機能と T-Kernel 仕様のランデブ機能にはいくつかの相違があり、ADA のランデブを実現するために、T-Kernel 仕様のランデブ機能を使えるとは限らない。

ランデブの動作を[図 12]の例を用いて説明する。この図で、タスク A とタスク B は非同期に実行しているものとする。

- もしタスク A が先に tk_cal_por を呼出した場合には、タスク B が tk_acp_por を呼び出すまでタスク A は待ち状態となる。この時タスク A は、ランデブの呼出し待ち状態になっている[図 12(a)]。
- 逆にタスク B が先に tk_acp_por を呼出した場合には、タスク A が tk_cal_por を呼び出すまでタスク B は待ち状態となる。この時タスク B は、ランデブの受付待ち状態になっている[図 12(b)]。
- タスク A が tk_cal_por を呼出し、タスク B が tk_acp_por を呼出した時点でランデブが成立し、タスク A を待ち状態としたまま、タスク B が待ち解除される。この時タスク A は、ランデブの終了待ち状態になっている。
- タスク B が tk_rpl_rdv を呼出した時点で、タスク A は待ち解除される。その後は、両タスクとも実行できる状態となる。



［図 12］ランデブの動作

ランデブ番号の具体的な付与方法の例として､ランデブ番号の下位ビットをランデブを呼出したタスクの ID 番号、上位ビットをシーケンシャルな番号とする方法がある。

【仕様決定の理由】

ランデブ機能は、他の同期・通信機能を組み合わせて実現することも可能であるが、返答を伴う通信を行う場合には、それ専用の機能を用意した方が、アプリケーションプログラムが書きやすく、他の同期・通信機能を組み合わせるよりも効率を上げることができると考えられる。一例として、ランデブ機能は、メッセージの受渡しが終わるまで双方のタスクを待たせておくために、メッセージを格納するための領域が必要ないという利点がある。

同じタスクが呼出したランデブであっても、ランデブ番号をできる限りユニークにしなければならないのは、次の理由による。ランデブが成立してランデブ終了待ち状態となっているタスクが、タイムアウトや待ち状態の強制解除などにより待ち解除された後、再度ランデブを呼出してランデブが成立した場合を考える。この時、最初のランデブのランデブ番号と、後のランデブのランデブ番号が同一の値であると、最初のランデブを終了させようとした時に、ランデブ番号が同一であるために後のランデブが終了してしまう。 2 つのランデブに異なるランデブ番号を付与し、ランデブ終了待ち状態のタスクに待ち対象のランデブ番号を記憶しておけば、最初のランデブを終了させようとした時にエラーとすることができる。

ランデブポート生成 tk_cre_por

tk_cre_por:Create Port for Rendezvous

【C 言語インタフェース】

```
ID porid = tk_cre_por ( T_CPOR *pk_cpor ) ;
```

【パラメータ】

T_CPOR* pk_cpor		Packet to Create Port	ランデブポートの生成情報

pk_cpor の内容

VP	exinf	ExtendedInformation	拡張情報
ATR	poratr	PortAttribute	ランデブポート属性
INT	maxcmsz	MaxCallMessageSize	呼出時のメッセージの最大長(バイト数)
INT	maxrmsz	MaxReplyMessageSize	返答時のメッセージの最大長(バイト数)
UB	dsname[8]	DS Object name	DS オブジェクト名称

――(以下に実装独自に他の情報を追加してもよい)――

【リターンパラメータ】

ID	porid	PortID	ランデブポート ID
	または	ErrorCode	エラーコード

【エラーコード】

E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_LIMIT	ランデブポートの数がシステムの上限を超えた
E_RSATR	予約属性(poratr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_cpor が不正, maxcmsz, maxrmsz が負または不正)

【解説】

ランデブポートを生成しランデブポート ID 番号を割当てる。具体的には、生成されたランデブポートに対して管理ブロックを割り付ける。ランデブポートは、ランデブを実現するためのプリミティブとなるオブジェクトである。

exinf は、対象ランデブポートに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、tk_ref_por で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。OS では exinf の内容について関知しない。

poratr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。poratr のシステム属性の部分では、次のような指定を行う。

```
poratr:= (TA_TFIFO || TA_TPRI) | [TA_DSNAME] | [TA_NODISWAI]
```

TA_TFIFO	呼出し待ちタスクのキューイングは FIFO
TA_TPRI	呼出し待ちタスクのキューイングは優先度順
TA_DSNAME	DS オブジェクト名称を指定する
TA_NODISWAI	tk_dis_wai による待ち禁止を拒否する

TA_TFIFO, TA_TPRI では、ランデブ呼出し待ちのタスクの待ち行列の並び順を指定する。ランデブ受け付け待ちのタスクの待ち行列は FIFO のみである。

TA_DSNAME を指定した場合に dsname が有効となり、DS オブジェクト名称として設定される。DS オブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DS のシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname、td_set_dsname を参照のこと。

TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとな

る。

```
#define TA_TFIFO     0x00000000  /* 待ちタスクを FIFO で管理 */
#define TA_TPRI      0x00000001  /* 待ちタスクを優先度順で管理 */
#define TA_DSNAME    0x00000040  /* DS オブジェクト名称を指定 */
#define TA_NODISWAI  0x00000080  /* 待ち禁止拒否 */
```

maxcmsz にはランデブの呼出し時に渡すメッセージの最大サイズ(バイト数)を指定する。maxcmsz に 0 を指定することも可能である。ただし、maxcmsz に 0 を指定した場合は、ランデブの呼出し時に渡すメッセージのサイズは 0 に限定され、メッセージ無しでの同期にのみ使用されることになる。

maxrmsz にはランデブの返答時に渡すメッセージの最大サイズ(バイト数)を指定する。maxrmsz に 0 を指定することも可能である。ただし、maxrmsz に 0 を指定した場合はランデブの返答時に渡すメッセージのサイズは 0 に限定されることになる。

ランデブポート削除 tk_del_por

tk_del_por:Delete Port for Rendezvous

【C 言語インタフェース】

```
ER ercd = tk_del_por ( ID porid ) ;
```

【パラメータ】

ID	porid	PortID	ランデブポート ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(porid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(porid のランデブポートが存在しない)

【解説】

porid で示されたランデブポートを削除する。

本システムコールの発行により、対象ランデブポートの ID 番号および管理ブロック用の領域は解放される。

対象ランデブポートにおいてランデブ受付(tk_acp_por)や呼出(tk_cal_por)を待っているタスクがあった場合にも、本システムコールは正常終了するが、待ち状態にあったタスクにはエラーE_DLT が返される。

tk_del_por によりランデブポートが削除されても、既にランデブ成立済のタスクに対しては、影響を与えない。ランデブ受付側タスク(待ち状態ではない)には何も通知されないし、ランデブ呼出側タスク(ランデブ終了待ち状態)の状態もそのままである。ランデブ受付側タスクが tk_rpl_rdv を行う時に、ランデブ成立に使ったランデブポートが既に削除されていても、tk_rpl_rdv は正常に実行される。

ランデブポートに対するランデブの呼出 tk_cal_por

tk_cal_por:Call Port for Rendezvous

【C 言語インタフェース】

```
INT rmsgsz = tk_cal_por ( ID porid, UINT calptn, VP msg, INT cmsgsz, TMO tmout ) ;
```

【パラメータ】

ID	porid	PortID	ランデブポート ID
UINT	calptn	CallBitPattern	呼出側選択条件を表すビットパターン
VP	msg	Message	メッセージを入れるアドレス
INT	cmsgsz	CallMessageSize	呼出メッセージのサイズ(バイト数)
TMO	tmout	Timeout	タイムアウト指定

【リターンパラメータ】

INT	rmsgsz	ReplyMessageSize	返答メッセージのサイズ(バイト数)
	または	ErrorCode	エラーコード

【エラーコード】

E_ID	不正 ID 番号(porid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(porid のランデブポートが存在しない)
E_PAR	パラメータエラー(cmsgsz＜0, cmsgsz＞maxcmsz, calptn＝0, msg が不正, tmout≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象ランデブポートが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

【解説】

ランデブポートに対するランデブ呼出を行う。

tk_cal_por の具体的な動作は次のようになる。porid で指定したランデブポートにおいてランデブ受付待ち状態のタスクがあり、そのタスクと tk_cal_por 発行タスクとの間でランデブ成立条件が満たされた場合は、ランデブ成立となる。この場合、ランデブ受付待ちだったタスクは実行可能状態(READY)となり、tk_cal_por 発行タスクはランデブ終了待ちの状態になる。ランデブ終了待ちの状態になったタスクは、ランデブの相手のタスク(ランデブ受付タスク)が tk_rpl_rdv を実行することにより待ち状態が解除される。この時点で tk_cal_por のシステムコールが終了する。

porid で指定したランデブポートに受付待ちタスクが無かった場合や、受付待ちタスクがあってもランデブ成立条件が満たされなかった場合には、tk_cal_por 発行タスクはこのランデブポートの呼出側待ち行列に並びランデブ呼出待ちの状態となる。ランデブ呼出し待ち行列の順序は、tk_cre_por の指定により FIFO またはタスク優先度順となる。

ランデブ成立条件は、受付側タスクの acpptn と呼出側タスクの calptn との論理積が 0 かどうかによって判定される。論理積が 0 でない場合にランデブ成立となる。calptn が 0 の場合は、決してランデブが成立しなくなるので、パラメータエラーE_PAR とする。

ランデブ成立時には、呼出側タスクから受付側タスクに対してメッセージ(呼出メッセージ)を送ることができる。呼出メッセージのサイズは cmsgsz で指定される。具体的には、呼出側タスクが tk_cal_por で指定した msg 以下の領域の cmsgsz バイトが、受付側タスクが tk_acp_por で指定した msg 以下の領域にコピーされる。

逆に、ランデブ終了時には、受付側タスクから呼出側タスクに対してメッセージ(返答メッセージ)を送ることができる。具体的には、受付側タスクが tk_rpl_rdv で指定した返答メッセージの内容が、呼出側タスクが tk_cal_por で指定した msg 以下の領域にコピーされる。また、返答メッセージのサイズ rmsgsz は、tk_cal_por のリターンパラメータとなる。結局、tk_cal_por の msg パラメータで指定されたメッセージ領域は、tk_rpl_rdv 実行の際に送ら

れてくるメッセージによって破壊されることになる。
なお、ランデブが回送された場合は、tk_cal_por で指定した msg のアドレスから最大で maxrmsz の領域をバッファとして使用するため、その内容を破壊する可能性がある。したがって、tk_cal_por で要求したランデブが回送される可能性がある場合は、期待する返答メッセージのサイズにかかわらず、msg 以下に少なくとも maxrmsz のサイズの領域を確保しておかなければならない。(詳細は tk_fwd_por の項を参照)
cmsgsz が、tk_cre_por で指定した maxcmsz よりも大きい場合は、エラーE_PAR となる。このエラーはランデブ呼出待ち状態に入る前にチェックされ、エラーの場合、tk_cal_por を実行したタスクはランデブ呼出待ち状態には入らない。
tmout により待ち時間の最大値(タイムアウト値)を指定することができる。タイムアウト指定が行なわれた場合、待ち解除の条件が満足されない(ランデブが成立しない)まま tmout の時間が経過すると、タイムアウトエラーE_TMOUT となってシステムコールが終了する。
tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(=1 ミリ秒)と同じである。
tmout として TMO_POL=0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、対象ランデブポートにランデブ受付待ちタスクが無いか、ランデブ成立条件が満たされない場合には、待ちに入らず E_TMOUT を返す。
tmout として TMO_FEVR=(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトすることなくランデブが成立するまで待ち続ける。
いずれにしても、tmout の指定はランデブ成立までの時間に関するタイムアウトを意味するものであり、ランデブ成立からランデブ終了までの時間には関係しない。

ランデブポートに対するランデブ受付 tk_acp_por

tk_acp_por:Accept Port for Rendezvous

【C 言語インタフェース】

```
INT cmsgsz = tk_acp_por ( ID porid, UINT acpptn, RNO *p_rdvno, VP msg, TMO tmout ) ;
```

【パラメータ】

ID	porid	PortID	ランデブポート ID
UINT	acpptn	AcceptBitPattern	受付側選択条件を表すビットパターン
RNO*	p_rdvno	Pointer to Rendezvous Number	リターンパラメータ rdvno を返す領域へのポインタ
VP	msg	Packet of CallMessage	呼出しメッセージを入れるアドレス
TMO	tmout	Timeout	タイムアウト指定

【リターンパラメータ】

RNO	rdvno	Rendezvous Number	ランデブ番号
INT	cmsgsz	CallMessageSize	呼出しメッセージのサイズ(バイト数)
	または	ErrorCode	エラーコード

【エラーコード】

E_ID	不正 ID 番号(porid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(porid のランデブポートが存在しない)
E_PAR	パラメータエラー(acpptn＝0, msg が不正, tmout≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象ランデブポートが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

【解説】

ランデブポートに対するランデブ受付を行う。

tk_acp_por の具体的な動作は次のようになる。porid で指定したランデブポートの呼出側待ち行列に入っているタスクと、このタスクとの間で、ランデブ成立条件が満たされた場合は、ランデブ成立となる。この場合、呼出側待ち行列にあったタスクは行列から外れ、ランデブ呼出待ち(ランデブ成立待ち)の状態からランデブ終了待ちの状態に変わる。tk_acp_por の発行タスクは、実行を継続する。

porid で指定したランデブポートの呼出側待ち行列にタスクが無かった場合や、タスクがあってもランデブ成立条件が満たされなかった場合、tk_acp_por の発行タスクはそのランデブポートに対するランデブ受付待ち状態になる。この時、既に別のタスクがランデブ受付待ち状態であったとしても、エラーとはならず、tk_acp_por を発行したタスクはランデブ受付待ち行列につながれる。また、一つのランデブポートを使って、複数のタスクが同時にランデブを行うことが可能である。そのため、porid で指定したランデブポートで別のタスクがランデブを行っている間に(前に成立したランデブに関する tk_rpl_rdv が実行される前に)に次のランデブを行っても、エラーとはならない。

ランデブ成立条件は、受付側タスクの acpptn と呼出側タスクの calptn との論理積が 0 かどうかによって判定される。論理積が 0 でない場合にランデブ成立となる。先頭のタスクが条件を満たさなければ、待ち行列の次のタスクについて順にチェックを行う。calptn と acpptn に 0 以外の同じ値を指定すれば、条件が無い(無条件)のと同じになる。また、acpptn が 0 の場合は、決してランデブが成立しなくなるので、パラメータエラーE_PAR とする。ランデブ成立までの処理に関しては、ランデブ呼出側とランデブ受付側で完全に対称である。

ランデブ成立時には、呼出側タスクから受付側タスクに対して呼出メッセージを送ることができる。呼出側タスクが指定した呼出メッセージの内容は、受付側タスクが tk_acp_por で指定した msg 以下の領域にコピーされる。また、呼出メッセージのサイズ cmsgsz は、tk_acp_por のリターンパラメータとなる。

ランデブ受付側のタスクが、同時に複数のランデブを行うことも可能である。具体的には、tk_acp_por によりあるランデブを受け付けたタスクが、tk_rpl_rdv を実行する前に、もう一度 tk_acp_por を実行しても構わない。また、この時の tk_acp_por は、前と異なるランデブポートを対象としたものであっても、前と同じランデブポートを対象としたものであっても構わない。特殊な例であるが、ランデブ中のタスクが同じランデブポートに対してもう一度 tk_acp_por を実行してランデブが成立した場合、同一のタスクが同一のランデブポートに対して複数の(多重の)ランデブを行っているという状態になる。もちろん、この場合にランデブの相手(呼出側タスク)は異なっている。

tk_acp_por のリターンパラメータとして返される rdvno は、同時に成立している複数のランデブを区別するための情報であり、ランデブ終了時に tk_rpl_rdv のパラメータとして使用する。また、ランデブ回送時には tk_fwd_por のパラメータとして使用する。rdvno の具体的な内容は実装依存であるが、ランデブ成立相手の呼出側タスクを指定するための情報を含んでいるはずである。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。タイムアウト指定が行われた場合、待ち解除の条件が満足されない(ランデブが成立しない)まま tmout の時間が経過すると、タイムアウトエラー E_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1 ミリ秒)と同じである。

tmout として TMO_POL＝0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、対象ランデブポートにランデブ呼出待ちタスクが無いか、ランデブ成立条件が満たされない場合には、待ちに入らず E_TMOUT を返す。また、tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトすることなくランデブの成立まで待ち続ける。

【補足事項】

ランデブ受付側のタスクも待ち行列を作る機能は、同じ処理をするサーバを複数個並列に走らせるような時に有用である。また、ランデブポートをタスク独立にしたという特徴を活かすことができる。

ランデブを受け付けたタスクが、何らかの理由でランデブ終了前(tk_rpl_rdv 実行前)に異常終了したような場合は、tk_cal_por を実行したランデブ呼出側のタスクがランデブ終了待ち状態から解放されないまま残ることになる。このようなケースを避けるためには、ランデブ受付側タスクの異常終了時に tk_rpl_rdv または tk_rel_wai を実行し、ランデブがエラーで終了したことをランデブ呼出側のタスクにも通知しておく必要がある。

rdvno にはランデブ成立相手の呼出側タスクを指定する情報を含むが、その番号の割当て方法は、できるだけユニークになるように配慮しなければならない。同一タスク間のランデブであっても、1 回目のランデブと 2 回目のランデブでは異なる rdvno を割当てる必要がある。この配慮により、以下のような問題を回避できる。

tk_cal_por を実行してランデブ終了待ちのタスクが、tk_rel_wai や tk_ter_tsk＋tk_sta_tsk 等によって強制的に待ち解除となり、再度 tk_cal_por を実行してランデブが成立した場合を考える。最初のランデブに対する rdvno と後のランデブに対する rdvno が同じ値であると、最初のランデブに対する tk_rpl_rdv によって後のランデブが終了してしまう。rdvno の割当てをユニークにし、ランデブ終了待ちタスクの側で期待する rdvno を覚えておけば、最初のランデブに対する tk_rpl_rdv のエラーを検出することができる。

rdvno の具体的な実現方法としては、たとえば、rdvno の下位バイトを呼出側タスクの ID、上位バイトをシーケンシャルな番号とすればよい。

calptn, acpptn によるランデブ成立条件の機能を使えば、ランデブの選択受付(ADA の select に相当)の機能が実現可能となる。ADA の select 文の例[図 13(a)]に相当する具体的な処理方法を[図 13(b)]に示す。

```
select
    when condition_A
        accept entry_A do ... end;
or
    when condition_B
        accept entry_B do ... end;
or
    when condition_C
        accept entry_C do ... end;
end select;
```

[図 13(a)] select 文を使った ADA のプログラム例

- entry_A, entry_B, entry_C がそれぞれ一つのランデブポートに対応するのではなく、select 文全体が一つのランデブポートに対応する。

- entry_A, entry_B, entry_C を、calptn, acpptn の 2^0, 2^1, 2^2 のビットに対応させる。

- ADA のプログラム例の中の select 文は、次のようになる。

```
ptn := 0;
if conditon_A then ptn := ptn + 2^0 endif;
if conditon_B then ptn := ptn + 2^1 endif;
if conditon_C then ptn := ptn + 2^2 endif;
tk_acp_por(acpptn := ptn);
```

- もし、プログラム例中の select 文以外に、select 無しの単なる entry_A の accept があれば、
```
tk_acp_por(acpptn := 2^0);
```
を実行すれば良い。また、entry_A, entry_B, entry_C を無条件に OR で待ちたい時は、
```
tk_acp_por(acpptn := 2^2+2^1+2^0);
```
を実行すれば良い。

- 一方、これを呼び出す側は、entry_A の呼出しであれば
```
tk_cal_por(calptn := 2^0);
```
を実行し、entry_C の呼出しであれば
```
tk_cal_por(calptn := 2^2);
```
を実行すれば良い。

[図 13(b)] ランデブによる ADA の select 機能の実現方法

ADA の選択機能は受け付け側にしか用意されていないが、ランデブでは、calptn で複数のビットを指定することにより、呼出側に選択機能を持たせることも可能である。

【仕様決定の理由】
ランデブ成立条件に関して呼出側と受付側が全く対称であるにもかかわらず、tk_cal_por と tk_acp_por が別システムコールとなっているのは、ランデブ成立後の処理が異なるからである。すなわち、呼出側はランデブ成立後に待ち状態になるのに対して、受付側はランデブ成立後に実行可能状態(READY)となる。

ランデブポートに対するランデブ回送 tk_fwd_por

tk_fwd_por:Forward Rendezvous to Other Port

【C 言語インタフェース】

```
ER ercd = tk_fwd_por ( ID porid, UINT calptn, RNO rdvno, VP msg, INT cmsgsz ) ;
```

【パラメータ】

ID	porid	PortID	回送先のランデブポート ID
UINT	calptn	CallBitPattern	呼出側選択条件を表すビットパターン
RNO	rdvno	RendezvousNumber	回送前のランデブ番号
VP	msg	CallMessage	回送するメッセージを入れるアドレス
INT	cmsgsz	CallMessageSize	回送するメッセージのサイズ(バイト数)

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(porid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(porid のランデブポートが存在しない)
E_PAR	パラメータエラー(cmsgsz＜0, cmsgsz＞回送後の maxcmsz, cmsgsz＞回送前の maxrmsz, calptn＝0, msg が不正)
E_OBJ	オブジェクトの状態が不正(rdvno が不正, 回送後の maxrmsz＞回送前の maxrmsz)
E_CTX	コンテキストエラー(タスク独立部から発行, 実装依存)
E_DISWAI	待ち禁止による待ち解除

【解説】

一旦受け付けたランデブを別のランデブポートに回送する。

このシステムコールを発行したタスク(タスク X とする)は、現在ランデブ中の状態(tk_acp_por を実行した後の状態)でなければならない。また、ランデブ相手の呼出側タスクはタスク Y、tk_acp_por のリターンパラメータとして返されたランデブ番号は rdvno であるものとする。その状態で tk_fwd_por を実行すると、タスク X とタスク Y との間のランデブ状態が解除され、その後 porid で示される別のランデブポート(ランデブポート B)に対してタスク Y がランデブ呼出を行ったのと同じ状況になる。

tk_fwd_por の具体的な動作は次のようになる。

1. rdvno で示されるランデブを解除する。
2. タスク Y を、porid のランデブポートに対してランデブ呼出待ちの状態にする。この時、ランデブ成立のための呼出側選択条件を表すビットパターン calptn は、タスク Y が tk_cal_por で指定したものではなく、タスク X が tk_fwd_por で指定したものが使用される。タスク Y から見ると、ランデブ終了待ちの状態からランデブ呼出待ちの状態に戻ることになる。
3. その後、porid のランデブポートに対するランデブが受け付けられれば、それを受け付けたタスクとタスク Y との間でランデブ成立となる。もちろん、porid のランデブポートに既にランデブ受付待ちタスクが存在し、ランデブ成立条件が満たされていれば、tk_fwd_por の実行により即座にランデブ成立となることもある。ここで、ランデブ成立時に受付側に送られるメッセージは、タスク Y が tk_cal_por で指定したものではなく、タスク X が tk_fwd_por で指定したものが使用される(calptn と同様)。
4. 新しいランデブが終了した際に tk_rpl_rdv で返されるメッセージは、タスク X が tk_fwd_por で指定した msg 以下の領域ではなく、タスク Y が tk_cal_por で指定した msg 以下の領域にコピーされる。

基本的には、
「tk_cal_por(porid=portA, calptn=ptnA, msg=mesA)の後で
tk_fwd_por(porid=portB, calptn=ptnB, msg=mesB)が実行された状態」
と、
「tk_cal_por(porid=portB, calptn=ptnB, msg=mesB)が実行された状態」
とは全く同じ状態になる。結果的に、カーネルとしてはランデブ回送の履歴を覚えておく必要はないことになる。

tk_fwd_por によってランデブ呼出待ち状態に戻ったタスクに対して tk_ref_tsk を実行した場合、tskwait は TTW_CAL となる。また、wid も回送先のランデブポートの ID 番号になる。

tk_fwd_por の実行は即座に終了する。このシステムコールで待ち状態になることはない。また、tk_fwd_por の実行が終わった後の tk_fwd_por 発行タスクは、回送前のランデブが成立したランデブポート、回送後のランデブポート(porid のランデブポート)、それらの上でランデブを行ったタスクのいずれとも無関係になる。

cmsgsz が、回送後のランデブポートの maxcmsz よりも大きい場合は、エラー E_PAR となる。このエラーはランデブの回送を行う前にチェックされる。エラーの場合、ランデブの回送は行われず、rdvno で示されるランデブも解除されない。

tk_fwd_por で指定した送信メッセージは、tk_fwd_por 実行時に他の領域(たとえば tk_cal_por で指定したメッセージ領域)にコピーされる。したがって、回送されたランデブが成立する前に tk_fwd_por の msg で示されるメッセージ領域の内容が変更されても、回送されたランデブはその影響を受けない。

tk_fwd_por でランデブを回送する場合、回送後のランデブポート(porid のランデブポート)の maxrmsz は、回送前のランデブの成立したランデブポートの maxrmsz と等しいか、それよりも小さくなければならない。回送後のランデブポートの maxrmsz が回送前のランデブポートの maxrmsz よりも大きかった場合は、回送先のランデブポートが不適当であるという意味で、E_OBJ のエラーとなる。ランデブ呼出側では、回送前のランデブポートの maxrmsz に合わせて返答メッセージ受信領域を用意しているため、ランデブの回送によって返答メッセージの最大サイズが大きくなると、呼出側に対して予期せぬ大きさの返答メッセージを返す可能性を生じ、問題を起こす。maxrmsz の大きなランデブポートにランデブを回送できないのは、このような理由による。

また、tk_fwd_por で送信するメッセージのサイズ cmsgsz についても、回送前のランデブの成立したランデブポートの maxrmsz と等しいか、それよりも小さくなければならない。これは、tk_fwd_por の実装方法として、tk_cal_por で指定したメッセージ領域を送信メッセージのバッファとして使うことを想定しているためである。cmsgsz が回送前のランデブポートの maxrmsz より大きかった場合には、E_PAR のエラーとなる。(詳細は【補足事項】を参照)

タスク独立部から tk_fwd_por, tk_rpl_rdv を発行する必要はないが、ディスパッチ禁止中あるいは割込み禁止中のタスクから tk_fwd_por, tk_rpl_rdv を発行することは可能である。この機能は、tk_fwd_por や tk_rpl_rdv と不可分に何らかの処理を行う場合に利用できる。なお、タスク独立部から tk_fwd_por, tk_rpl_rdv が発行された場合のエラーチェックは実装依存である。

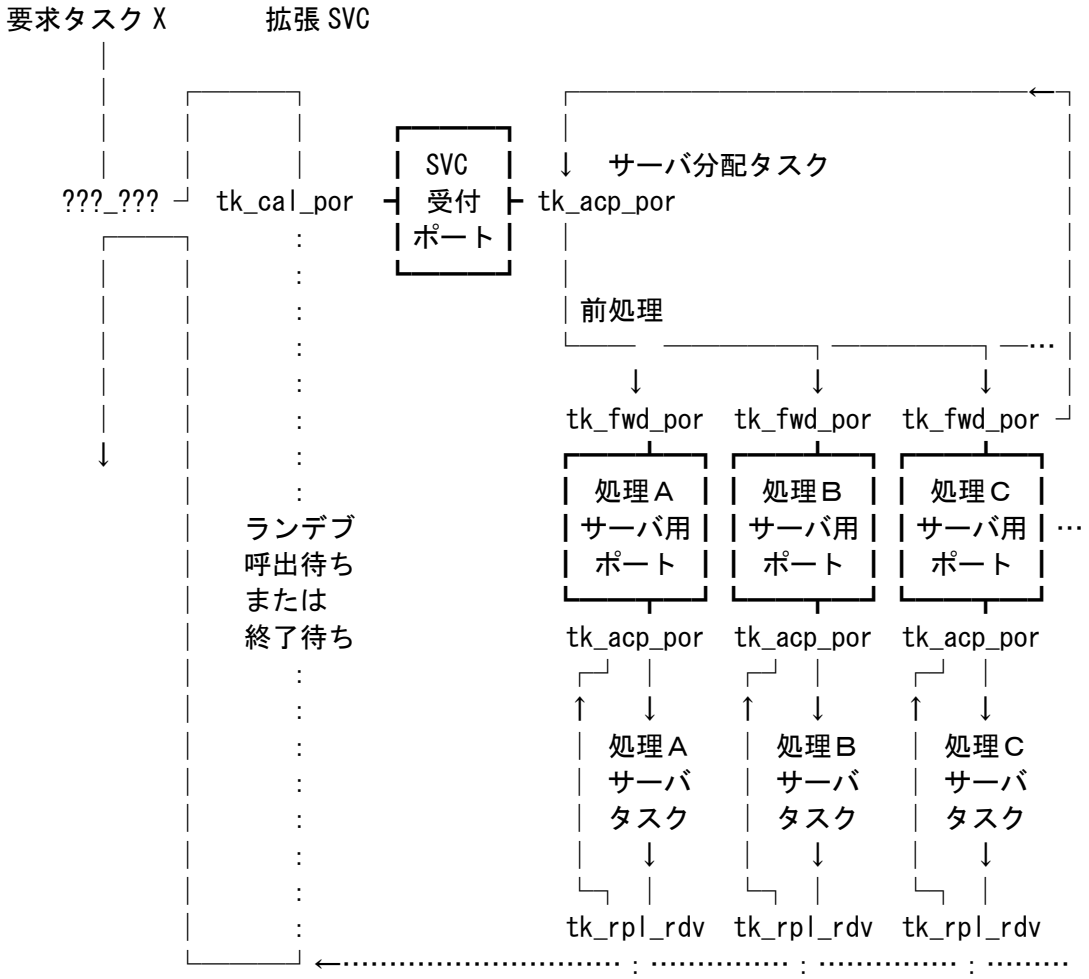
tk_fwd_por により、ランデブ終了待ち状態であったタスク Y がランデブ呼出待ちの状態に戻った場合、次にランデブが成立するまでのタイムアウトは、常に永久待ち(TMO_FEVR)として扱われる。

回送先のランデブポートは、前のランデブに使っていたランデブポート(rdvno のランデブが成立したランデブポート)と同じランデブポートであっても構わない。この場合は、tk_fwd_por によって、一旦受け付けたランデブの受付処理をとりやめることになる。ただし、この場合でも、呼出メッセージや calptn は、呼出側タスクが tk_cal_por で指定したものではなく、受付側タスクが tk_fwd_por で指定したものに変更される。

一旦回送されてきたランデブを、さらに回送することも可能である。

【補足事項】

tk_fwd_por を使ったサーバタスクの動作イメージを[図 14]に示す。



※ 太枠内はランデブポート(ランデブエントリー)を表す。

※ tk_fwd_por の代わりに tk_cal_por を使うことも可能であるが、その場合はランデブがネストする。処理 A～C のサーバタスクの処理終了後にそのまま要求タスク X の実行を再開して構わないのであれば、tk_fwd_por を利用することによってランデブのネストが不要となり、効率良い動作をすることができる。

[図 14] tk_fwd_por を使ったサーバタスクの動作イメージ

一般に、tk_fwd_por を実行するのは、[図 14]に示されるようなサーバ分配タスク(サーバの受け付けた処理を別のタスクに分配するためのタスク)である。したがって、tk_fwd_por を実行したサーバ分配タスクは、回送したランデブの成立の可否にかかわらず、次の要求を受け付ける処理に移らなければならない。その場合、tk_fwd_por のメッセージ領域は次の要求を処理するために利用されるので、メッセージ領域の内容を変更しても、前に処理したランデブの回送には影響しないようにしなければならない。このため、tk_fwd_por の実行後は、回送されたランデブの成立前であっても、tk_fwd_por の msg で示されるメッセージ領域の内容を変更することが可能でなければならない。

実装上は、この仕様を実現するために、tk_cal_por で指定したメッセージ領域をバッファとして使うことが許される。すなわち、tk_fwd_por の処理では、tk_fwd_por で指定した呼出メッセージを tk_cal_por の msg で指定したメッセージ領域にコピーし、tk_fwd_por 発行タスクがメッセージ領域の内容を変更しても構わないようにする。ランデブ成立時には、ランデブが回送されたものかどうかにかかわらず、tk_cal_por のメッセージ領域に置かれていたメッセージが受付側に渡される。

このような実装方法を適用できるように、以下のような仕様を設けている。

- tk_cal_por で要求したランデブが回送される可能性がある場合は、期待する返答メッセージのサイズにかかわらず、tk_cal_por の msg 以下に少なくとも maxrmsz のサイズの領域を確保しておかなければならない。
- tk_fwd_por で送信するメッセージのサイズ cmsgsz は、回送前のランデブポートの maxrmsz と等しいか、それよりも小さくなければならない。
- tk_fwd_por によりランデブが回送される場合に、回送後のランデブポートの maxrmsz が回送前のランデブポートの maxrmsz より大きくなることはない。等しい値をとるか、小さくなっていく。

【仕様決定の理由】

システム全体で持つべき状態の数を減らすため、tk_fwd_por の仕様は、ランデブ回送の履歴を保存しないという前提で設計されている。ランデブ回送の履歴を覚える必要がある用途では、tk_fwd_por ではなく、tk_cal_por～tk_acp_por の組をネストして使えばよい。

ランデブ返答 tk_rpl_rdv

tk_rpl_rdv:Reply Rendezvous

【C 言語インタフェース】

```
ER ercd = tk_rpl_rdv ( RNO rdvno, VP msg, INT rmsgsz ) ;
```

【パラメータ】

RNO	rdvno	RendezvousNumber	ランデブ番号
VP	msg	ReplyMessage	返答メッセージを入れるアドレス
INT	rmsgsz	ReplyMessageSize	返答メッセージのサイズ(バイト数)

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_PAR	パラメータエラー(rmsgsz＜0, rmsgsz＞maxrmsz, msg が不正)
E_OBJ	オブジェクトの状態が不正(rdvno が不正)
E_CTX	コンテキストエラー(タスク独立部から発行,実装依存)

【解説】

ランデブの相手のタスク(呼出側タスク)に対して返答を返し、ランデブを終了する。

このシステムコールを発行したタスク(タスク X とする)は、ランデブ中の状態(tk_acp_por を実行した後の状態)でなければならない。また、ランデブ相手の呼出側タスクはタスク Y、tk_acp_por のリターンパラメータとして返されたランデブ番号は rdvno であるものとする。その状態で tk_rpl_rdv を実行すると、タスク X とタスク Y との間のランデブ状態が解除され、ランデブ終了待ち状態にあった呼出側タスク Y は実行可能状態(READY)に戻る。

tk_rpl_rdv でランデブを終了する時には、受付側タスク X から呼出側タスク Y に対して返答メッセージを送ることができる。受付側タスクが指定した返答メッセージの内容は、呼出側タスクが tk_cal_por で指定した msg 以下の領域にコピーされる。また、返答メッセージのサイズ rmsgsz は、tk_cal_por のリターンパラメータとなる。

rmsgsz が、tk_cre_por で指定した maxrmsz よりも大きい場合は、エラー E_PAR となる。このエラーが検出された場合、ランデブは終了せず、tk_cal_por を実行したタスクのランデブ終了待ち状態は解除されない。

タスク独立部から tk_fwd_por, tk_rpl_rdv を発行することはできないが、ディスパッチ禁止中あるいは割込み禁止中のタスクから tk_fwd_por, tk_rpl_rdv を発行することは可能である。この機能は、tk_fwd_por や tk_rpl_rdv と不可分に何らかの処理を行う場合に利用できる。なお、タスク独立部から tk_fwd_por, tk_rpl_rdv が発行された場合のエラーチェックは実装依存である。

【補足事項】

ランデブを呼出したタスクが、何らかの理由でランデブ終了前(tk_rpl_rdv 実行前)に異常終了したような場合にも、ランデブ受付側のタスクは直接それを知ることができない。この場合は、ランデブ受付側のタスクが tk_rpl_rdv を実行する時に E_OBJ のエラーとなる。

ランデブ成立後は、原則としてタスクとランデブポートとが切り離されてしまう(相互に情報を参照する必要がない)が、tk_rpl_rdv のメッセージ長のチェックに使用する maxrmsz のみは、ランデブポートに依存した情報であるため、ランデブ中のタスクがどこかに覚えておく必要がある。実装上の工夫としては、待ち状態になっている呼出側タスクの TCB、あるいは TCB から参照可能な領域(スタック等)に入れておくことが考えられる。

【仕様決定の理由】

tk_rpl_rdv や tk_fwd_por のパラメータでは、成立中のランデブを区別する情報として rdvno を指定するが、ランデブ成立に利用したランデブポートの ID(porid)は指定しない。これは、ランデブ成立後のタスクがランデブポートとは無関係になるという方針に基づいたものである。

rdvno が不正値の場合のエラーコードとして、E_PAR ではなく E_OBJ を使用している。これは、rdvno の実体が呼出側タスクを示しているという理由による。

ランデブポート状態参照 tk_ref_por

tk_ref_por:Refer Port Status

【C 言語インタフェース】

```
ER ercd = tk_ref_por ( ID porid, T_RPOR *pk_rpor ) ;
```

【パラメータ】

ID	porid	PortID	ランデブポート ID
T_RPOR*	pk_rpor	Packet to Refer Port	ランデブポート状態を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード
pk_rpor の内容			
VP	exinf	ExtendedInformation	拡張情報
ID	wtsk	WaitTaskInformation	呼出待ちタスクの有無
ID	atsk	AcceptTaskInformation	受付待ちタスクの有無
INT	maxcmsz	MaximumCallMessageSize	呼出時のメッセージの最大長(バイト数)
INT	maxrmsz	MaximumReplyMessageSize	返答時のメッセージの最大長(バイト数)
――(以下に実装独自に他の情報を追加してもよい)――			

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(porid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(porid のランデブポートが存在しない)
E_PAR	パラメータエラー(pk_rpor が不正)

【解説】

porid で示された対象ランデブポートの各種の状態を参照し、リターンパラメータとして受付待ちタスクの有無(atsk)、呼出待ちタスクの有無(wtsk)、メッセージの最大長(maxcmsz, maxrmsz)、拡張情報(exinf)を返す。

wtsk は、このランデブポートでランデブ呼出待ちになっているタスクの ID を示す。ランデブ呼出待ちタスクが無い場合は wtsk＝0 となる。一方、atsk は、このランデブポートでランデブ受付待ちになっているタスクの ID を示す。ランデブ受付待ちタスクが無い場合は atsk＝0 となる。

このランデブポートで複数のタスクが呼出待ち状態または受付待ち状態になっている場合には、それぞれ呼出待ち行列または受付待ち行列の先頭のタスクの ID を返す。

対象ランデブポートが存在しない場合には、エラーE_NOEXS となる。

【補足事項】

このシステムコールでは、現在ランデブ中のタスクに関する情報を知ることはできない。

4.6 メモリプール管理機能

「メモリプール管理機能」は、ソフトウェアによってメモリプールの管理やメモリブロックの割当てを行う機能である。

メモリプールには、固定長メモリプールと可変長メモリプールがある。両者は別のオブジェクトであり、操作のためのシステムコールが異なっている。固定長メモリプールから獲得されるメモリブロックはサイズが固定されているのに対して、可変長メモリプールから獲得されるメモリブロックでは、任意のブロックサイズを指定することができる。

メモリプール管理機能によって管理されるメモリは共有空間のメモリのみである。タスク固有空間のメモリを管理する機能はない。

4.6.1 固定長メモリプール

固定長メモリプールは、固定されたサイズのメモリブロックを動的に管理するためのオブジェクトである。固定長メモリプール機能には、固定長メモリプールを生成／削除する機能、固定長メモリプールに対してメモリブロックを獲得／返却する機能、固定長メモリプールの状態を参照する機能が含まれる。固定長メモリプールは ID 番号で識別されるオブジェクトである。固定長メモリプールの ID 番号を固定長メモリプール ID と呼ぶ。

固定長メモリプールは、固定長メモリプールとして利用するメモリ領域(これを固定長メモリプール領域、または単にメモリプール領域と呼ぶ)と、メモリブロックの獲得を待つタスクの待ち行列を持つ。固定長メモリプールからメモリブロックを獲得するタスクは、メモリプール領域に空きがなくなった場合、次にメモリブロックが返却されるまで固定長メモリブロックの獲得待ち状態となる。固定長メモリブロックの獲得待ち状態になったタスクは、その固定長メモリプールの待ち行列につながれる。

【補足事項】

固定長メモリプールの場合、何種類かのサイズのメモリブロックが必要となる場合には、サイズ毎に複数のメモリプールを用意する必要がある。

固定長メモリプール生成 tk_cre_mpf

tk_cre_mpf:Create Fixed-size MemoryPool

【C 言語インタフェース】

```
ID mpfid = tk_cre_mpf ( T_CMPF *pk_cmpf ) ;
```

【パラメータ】

T_CMPF* pk_cmpf　　Packet to Create MemoryPool　　固定長メモリプール生成情報

pk_cmpf の内容

VP	exinf	ExtendedInformation	拡張情報
ATR	mpfatr	MemoryPoolAttribute	メモリプール属性
INT	mpfcnt	MemoryPoolBlockCount	メモリプール全体のブロック数
INT	blfsz	MemoryBlockSize	固定長メモリブロックサイズ(バイト数)
UB	dsname[8]	DS Object name	DS オブジェクト名称

――(以下に実装独自に他の情報を追加してもよい)――

【リターンパラメータ】

ID	mpfid	MemoryPoolID	固定長メモリプール ID
	または	ErrorCode	エラーコード

【エラーコード】

E_NOMEM	メモリ不足(管理ブロックやメモリプール用の領域が確保できない)
E_LIMIT	固定長メモリプールの数がシステムの上限を超えた
E_RSATR	予約属性(mpfatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_cmpf が不正, mpfcnt, blfsz が負または不正)

【解説】

固定長メモリプールを生成し固定長メモリプール ID を割当てる。具体的には、mpfcnt,blfsz の情報を元に、メモリプールとして利用するメモリ領域を確保する。また、生成したメモリプールに対して管理ブロックを割り付ける。ここで生成されたメモリプールに対して tk_get_mpf システムコールを発行することにより、blfsz のサイズ(バイト数)をもつメモリブロックを獲得することができる。

exinf は、対象メモリプールに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、tk_ref_mpf で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。OS では exinf の内容について関知しない。

mpfatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。mpfatr のシステム属性の部分では、次のような指定を行う。

```
mpfatr:= (TA_TFIFO || TA_TPRI) | [TA_DSNAME] | [TA_NODISWAI]
       | (TA_RNG0 || TA_RNG1 || TA_RNG2 || TA_RNG3)
```

TA_TFIFO	メモリ獲得待ちタスクのキューイングは FIFO
TA_TPRI	メモリ獲得待ちタスクのキューイングは優先度順
TA_RNGn	メモリのアクセス制限を保護レベル n とする
TA_DSNAME	DS オブジェクト名称を指定する
TA_NODISWAI	tk_dis_wai による待ち禁止を拒否する

```
#define TA_TFIFO        0x00000000      /* 待ちタスクを FIFO で管理 */
#define TA_TPRI         0x00000001      /* 待ちタスクを優先度順で管理 */
#define TA_DSNAME       0x00000040      /* DS オブジェクト名称を指定 */
#define TA_NODISWAI     0x00000080      /* 待ち禁止拒否 */
#define TA_RNG0         0x00000000      /* 保護レベル 0 */
#define TA_RNG1         0x00000100      /* 保護レベル 1 */
#define TA_RNG2         0x00000200      /* 保護レベル 2 */
#define TA_RNG3         0x00000300      /* 保護レベル 3 */
```

TA_TFIFO, TA_TPRI では、タスクがメモリ獲得のためにメモリプールの待ち行列に並ぶ際の並び方を指定することができる。属性が TA_TFIFO であればタスクの待ち行列は FIFO となり、属性が TA_TPRI であればタスクの待ち行列はタスクの優先度順となる。

TA_RNGn では、メモリのアクセスを制限する保護レベルを指定する。指定された保護レベルと同じかより高い保護レベルで実行しているタスクからのみアクセス可能である。低いレベルで実行しているタスクがアクセスすると CPU の保護違反の例外が発生する。例えば、TA_RNG1 を指定して作成したメモリプールから獲得したメモリは、TA_RNG0,1 で動作しているタスクからはアクセス可能だが、TA_RNG2,3 で動作しているタスクからはアクセスできない。

作成されるメモリプールは、共有空間上の常駐メモリとなる。タスク固有空間上にメモリプールを作成する機能はない。

TA_DSNAME を指定した場合に dsname が有効となり、DS オブジェクト名称として設定される。DS オブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DS のシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname、td_set_dsname を参照のこと。

TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

【補足事項】

固定長メモリプールの場合、ブロックサイズを変えるためには別のメモリプールを用意しなければならない。すなわち、何種類かのメモリブロックサイズが必要となる場合は、サイズごとに複数のメモリプールを設ける必要がある。

MMU のないシステムにおいても、移植性確保のために TA_RNGn 属性を受け付けなければならない。例えば、TA_RNGn の指定はすべて TA_RNG0 相当として処理してもよいが、エラーとはしない。

固定長メモリプール削除 tk_del_mpf

tk_del_mpf:Delete Fixed-size MemoryPool

【C 言語インタフェース】

```
ER ercd = tk_del_mpf ( ID mpfid ) ;
```

【パラメータ】

ID	mpfid	MemoryPoolID	固定長メモリプール ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mpfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mpfid の固定長メモリプールが存在しない)

【解説】

mpfid で示される固定長メモリプールを削除する。

このメモリプールからメモリを獲得しているタスクが存在しても、そのチェックや通知は行われない。すべてのメモリブロックが返却されていなくても、このシステムコールは正常終了する。

本システムコールの発行により､対象メモリプールの ID 番号および管理ブロック用の領域やメモリプール本体の領域は解放される。

対象メモリプールにおいてメモリ獲得を待っているタスクがあった場合にも、本システムコールは正常終了するが、待ち状態にあったタスクにはエラーE_DLT が返される。

固定長メモリブロック獲得 tk_get_mpf

tk_get_mpf:Get Fixed-size Memory Block

【C 言語インタフェース】

```
ER ercd = tk_get_mpf ( ID mpfid, VP *p_blf, TMO tmout ) ;
```

【パラメータ】

ID	mpfid	MemoryPoolID	固定長メモリプール ID
VP*	p_blf	Pointer to BlockStartAddress	リターンパラメータ blf を返す領域へのポインタ
TMO	tmout	Timeout	タイムアウト指定

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード
VP	blf	BlockStartAddress	メモリブロックの先頭アドレス

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mpfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mpfid の固定長メモリプールが存在しない)
E_PAR	パラメータエラー(tmout≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象メモリプールが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

【解説】

mpfid で示される固定長メモリプールからメモリブロックを獲得する。獲得したメモリブロックの先頭アドレスが blf に返される。獲得されるメモリブロックのサイズは、固定長メモリプール生成時に blfsz パラメータで指定された値となる。

獲得したメモリのゼロクリアは行われず、獲得されたメモリブロックの内容は不定となる。

指定したメモリプールからメモリブロックが獲得できなければ、tk_get_mpf 発行タスクがそのメモリプールのメモリ獲得待ち行列につながれ、メモリを獲得できるようになるまで待つ。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。タイムアウト指定が行なわれた場合、待ち解除の条件が満足されない(空きメモリができない)まま tmout の時間が経過すると、タイムアウトエラー E_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1 ミリ秒)と同じである。

tmout として TMO_POL＝0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、メモリが獲得できなかった場合も待ちに入ることなく E_TMOUT を返す。

tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずにメモリが獲得できるまで待ち続ける。

メモリブロック獲得待ちを行う場合の待ち行列の順序は、メモリプールの属性によって、FIFO またはタスク優先度順のいずれかとなる。

固定長メモリブロック返却 tk_rel_mpf

tk_rel_mpf:Release Fixed-size Memory Block

【C 言語インタフェース】

```
ER ercd = tk_rel_mpf ( ID mpfid, VP blf ) ;
```

【パラメータ】

ID	mpfid	MemoryPoolID	固定長メモリプール ID
VP	blf	BlockStartAddress	メモリブロックの先頭アドレス

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mpfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mpfid の固定長メモリプールが存在しない)
E_PAR	パラメータエラー(blf が不正、 異なるメモリプールへの返却)

【解説】

blf で示されるメモリブロックを、mpfid で示される固定長メモリプールへ返却する。

tk_rel_mpf の実行により、mpfid のメモリプールでメモリを待っていた別のタスクがメモリを獲得し、そのタスクの待ち状態が解除される場合がある。

メモリブロックの返却を行う固定長メモリプールは、メモリブロックの獲得を行った固定長メモリプールと同じものでなければならない。メモリブロックの返却を行うメモリプールが、メモリブロックの獲得を行ったメモリプールと異なっていることが検出された場合には、E_PAR のエラーとなる。ただし、エラーを検出するか否かは実装依存である。

固定長メモリプール状態参照 tk_ref_mpf

tk_ref_mpf:Refer Fixed-size MemoryPool Status

【C 言語インタフェース】

```
ER ercd = tk_ref_mpf ( ID mpfid, T_RMPF *pk_rmpf ) ;
```

【パラメータ】

ID	mpfid	MemoryPoolID	固定長メモリプール ID
T_RMPF*	pk_rmpf	Packet to Refer MemoryPool	メモリプール状態を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_rmpf の内容

VP	exinf	ExtendedInformation	拡張情報
ID	wtsk	WaitTaskInformation	待ちタスクの有無
INT	frbcnt	FreeBlockCount	空き領域のブロック数

――(以下に実装独自に他の情報を追加してもよい)――

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mpfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mpfid の固定長メモリプールが存在しない)
E_PAR	パラメータエラー(pk_rmpf が不正)

【解説】

mpfid で示された対象固定長メモリプールの各種の状態を参照し、リターンパラメータとして現在の空きブロック数 frbcnt、待ちタスクの有無(wtsk)、拡張情報(exinf)を返す。

wtsk は、この固定長メモリプールで待っているタスクの ID を示す。この固定長メモリプールで複数のタスクが待っている場合には、待ち行列の先頭のタスクの ID を返す。待ちタスクが無い場合は wtsk＝0 となる。

tk_ref_mpf で、対象固定長メモリプールが存在しない場合には、エラーE_NOEXS となる。

どんな場合でも、frbcnt＝0 と wtsk＝0 の少なくとも一方は成り立つ。

【補足事項】

tk_ref_mpl の frsz ではメモリの空き領域の合計サイズがバイト数で返るのに対して、tk_ref_mpf の frbcnt では空きブロックの数が返る。

4.6.2 可変長メモリプール

可変長メモリプールは、任意のサイズのメモリブロックを動的に管理するためのオブジェクトである。可変長メモリプール機能には、可変長メモリプールを生成／削除する機能、可変長メモリプールに対してメモリブロックを獲得／返却する機能、可変長メモリプールの状態を参照する機能が含まれる。可変長メモリプールは ID 番号で識別されるオブジェクトである。可変長メモリプールの ID 番号を可変長メモリプール ID と呼ぶ。

可変長メモリプールは、可変長メモリプールとして利用するメモリ領域(これを可変長メモリプール領域、または単にメモリプール領域と呼ぶ)と、メモリブロックの獲得を待つタスクの待ち行列を持つ。可変長メモリプールからメモリブロックを獲得するタスクは、メモリプール領域の空き領域が足りなくなった場合、十分なサイズのメモリブロックが返却されるまで可変長メモリブロックの獲得待ち状態となる。可変長メモリブロックの獲得待ち状態になったタスクは、その可変長メモリプールの待ち行列につながれる。

【補足事項】

可変長メモリプールでメモリブロックの獲得を待っているタスクは、待ち行列につながれている順序でメモリブロックを獲得する。例えば、ある可変長メモリプールに対して 400 バイトのメモリブロックを獲得しようとしているタスク A と、100 バイトのメモリブロックを獲得しようとしているタスク B が、この順で待ち行列につながれている時に、別のタスクからのメモリブロックの返却により 200 バイトの連続空きメモリ領域ができたとする。このような場合でも、タスク A がメモリブロックを獲得するまで、タスク B はメモリブロックを獲得できない。

可変長メモリプール生成 tk_cre_mpl

tk_cre_mpl:Create Variable-size MemoryPool

【C 言語インタフェース】

```
ID mplid = tk_cre_mpl ( T_CMPL *pk_cmpl ) ;
```

【パラメータ】

T_CMPL* pk_cmpl　Packet to Create MemoryPool　可変長メモリプール生成情報

pk_cmpl の内容

VP	exinf	ExtendedInformation	拡張情報
ATR	mplatr	MemoryPoolAttribute	メモリプール属性
INT	mplsz	MemoryPoolSize	メモリプール全体のサイズ(バイト数)
UB	dsname[8]	DS Object name	DS オブジェクト名称

――(以下に実装独自に他の情報を追加してもよい)――

【リターンパラメータ】

ID	mplid	MemoryPoolID	可変長メモリプール ID
	または	ErrorCode	エラーコード

【エラーコード】

E_NOMEM	メモリ不足(管理ブロックやメモリプール用の領域が確保できない)
E_LIMIT	可変長メモリプールの数がシステムの上限を超えた
E_RSATR	予約属性(mplatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_cmpl が不正、mplsz が負または不正)

【解説】

可変長メモリプールを生成し可変長メモリプール ID を割当てる。具体的には、mplsz の情報を元に、メモリプールとして利用するメモリ領域を確保する。また、生成したメモリプールに対して管理ブロックを割り付ける。

exinf は、対象メモリプールに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、tk_ref_mpl で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。OS では exinf の内容について関知しない。

mplatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。mplatr のシステム属性の部分では、次のような指定を行う。

```
mplatr:= (TA_TFIFO || TA_TPRI) | [TA_DSNAME] | [TA_NODISWAI]
       | (TA_RNG0 || TA_RNG1 || TA_RNG2 || TA_RNG3)
```

TA_TFIFO	メモリ獲得待ちタスクのキューイングは FIFO
TA_TPRI	メモリ獲得待ちタスクのキューイングは優先度順
TA_RNGn	メモリのアクセス制限を保護レベル n とする
TA_DSNAME	DS オブジェクト名称を指定する
TA_NODISWAI	tk_dis_wai による待ち禁止を拒否する

```
#define TA_TFIFO      0x00000000  /* 待ちタスクを FIFO で管理 */
#define TA_TPRI       0x00000001  /* 待ちタスクを優先度順で管理 */
#define TA_DSNAME     0x00000040  /* DS オブジェクト名称を指定 */
#define TA_NODISWAI   0x00000080  /* 待ち禁止拒否 */
```

```
#define TA_RNG0         0x00000000  /* 保護レベル 0 */
#define TA_RNG1         0x00000100  /* 保護レベル 1 */
#define TA_RNG2         0x00000200  /* 保護レベル 2 */
#define TA_RNG3         0x00000300  /* 保護レベル 3 */
```

TA_TFIFO,TA_TPRI では、タスクがメモリ獲得のためにメモリプールの待ち行列に並ぶ際の並び方を指定することができる。属性が TA_TFIFO であればタスクの待ち行列は FIFO となり、属性が TA_TPRI であればタスクの待ち行列はタスクの優先度順となる。

タスクがメモリ獲得待ちの行列を作った場合は、待ち行列先頭のタスクに優先してメモリを割当てる。待ち行列の２番目以降により少ないメモリサイズを要求しているタスクがあった場合も、そのタスクが先にメモリを獲得することはない。例えば、ある可変長メモリプールに対して要求メモリサイズ＝400 のタスク A と要求メモリサイズ＝100 のタスク B がこの順で待っており、別のタスクの tk_rel_mpl によりメモリサイズ＝200 の連続空きメモリ領域ができたとする。このとき、行列の先頭ではないが要求サイズの少ないタスク B が先にメモリを獲得することはない。

TA_RNGn では、メモリのアクセスを制限する保護レベルを指定する。指定された保護レベルと同じかより高い保護レベルで実行しているタスクからのみアクセス可能である。低いレベルで実行しているタスクがアクセスすると CPU の保護違反の例外が発生する。例えば、TA_RNG1 を指定して作成したメモリプールから獲得したメモリは、TA_RNG0,1 で動作しているタスクからはアクセス可能だが､TA_RNG2,3 で動作しているタスクからはアクセスできない。

作成されるメモリプールは、共有空間上の常駐メモリとなる。タスク固有空間上にメモリプールを作成する機能はない。

TA_DSNAME を指定した場合に dsname が有効となり、DS オブジェクト名称として設定される。DS オブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DS のシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname、td_set_dsname を参照のこと。

TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

【補足事項】

メモリ獲得待ち行列の先頭のタスクの待ちが強制解除されたり、タスク優先度が変更されるなどで待ち行列の順序が変化した場合は、新たに待ち行列の先頭になったタスクに対してメモリ割当てが試みられる。メモリを割当てることができれば、そのタスクの待ちは解除される。したがって、tk_rel_mpl によるメモリの解放がなくても、状況によってはメモリの獲得が行われ、待ちが解除される場合がある。

MMU のないシステムにおいても､移植性確保のために TA_RNGn 属性を受け付けなければならない｡例えば､TA_RNGn の指定はすべて TA_RNG0 相当として処理してもよいが、エラーとはしない。

【仕様決定の理由】

複数個の可変長メモリプールを設ける機能は、エラー処理時や緊急時などにメモリを確保するためのメモリプールを分離しておくために利用できる。

可変長メモリプール削除 tk_del_mpl

tk_del_mpl:Delete Variable-size MemoryPool

【C 言語インタフェース】

```
ER ercd = tk_del_mpl ( ID mplid ) ;
```

【パラメータ】

ID	mplid	MemoryPoolID	可変長メモリプール ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mplid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mplid の可変長メモリプールが存在しない)

【解説】

mplid で示される可変長メモリプールを削除する。

このメモリプールからメモリを獲得しているタスクが存在しても、そのチェックや通知は行われない。すべてのメモリブロックが返却されていなくても、このシステムコールは正常終了する。

本システムコールの発行により､対象メモリプールの ID 番号および管理ブロック用の領域やメモリプール本体の領域は解放される。

対象メモリプールにおいてメモリ獲得を待っているタスクがあった場合にも、本システムコールは正常終了するが、待ち状態にあったタスクにはエラーE_DLT が返される。

可変長メモリブロック獲得 tk_get_mpl

tk_get_mpl:Get Variable-size Memory Block

【C 言語インタフェース】

```
ER ercd = tk_get_mpl ( ID mplid, INT blksz, VP *p_blk, TMO tmout ) ;
```

【パラメータ】

ID	mplid	MemoryPoolID	可変長メモリプール ID
INT	blksz	MemoryBlockSize	メモリブロックサイズ(バイト数)
VP*	p_blk	Pointer to BlockStartAddress	リターンパラメータ blk を返す領域へのポインタ
TMO	tmout	Timeout	タイムアウト指定

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード
VP	blk	BlockStartAddress	メモリブロックの先頭アドレス

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mplid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mplid の可変長メモリプールが存在しない)
E_PAR	パラメータエラー(tmout≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象メモリプールが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

【解説】

mplid で示される可変長メモリプールから、blksz で指定されるサイズ(バイト数)のメモリブロックを獲得する。獲得したメモリブロックの先頭アドレスが blk に返される。

獲得したメモリのゼロクリアは行われず、獲得されたメモリブロックの内容は不定となる。

メモリが獲得できなければ、本システムコールを発行したタスクは待ち状態に入る。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。タイムアウト指定が行なわれた場合、待ち解除の条件が満足されない(空きメモリができない)まま tmout の時間が経過すると、タイムアウトエラー E_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1 ミリ秒)と同じである。

tmout として TMO_POL＝0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、メモリが獲得できなかった場合も待ちに入ることなく E_TMOUT を返す。

tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずにメモリが獲得できるまで待ち続ける。

メモリブロック獲得待ちを行う場合の待ち行列の順序は、メモリプールの属性によって、FIFO またはタスク優先度順のいずれかとなる。

可変長メモリブロック返却 tk_rel_mpl

tk_rel_mpl:Release Variable-size Memory Block

【C 言語インタフェース】

```
ER ercd = tk_rel_mpl ( ID mplid, VP blk ) ;
```

【パラメータ】

ID	mplid	MemoryPoolID	可変長メモリプール ID
VP	blk	BlockStartAddress	メモリブロックの先頭アドレス

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mplid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mplid の可変長メモリプールが存在しない)
E_PAR	パラメータエラー(blk が不正, 異なるメモリプールへの返却)

【解説】

blk で示されるメモリブロックを、mplid で示される可変長メモリプールへ返却する。

tk_rel_mpl の実行により、mplid のメモリプールでメモリを待っていた別のタスクがメモリを獲得し、そのタスクの待ち状態が解除される場合がある。

メモリブロックの返却を行う可変長メモリプールは、メモリブロックの獲得を行った可変長メモリプールと同じものでなければならない。メモリブロックの返却を行うメモリプールが、メモリブロックの獲得を行ったメモリプールと異なっていることが検出された場合には、E_PAR のエラーとなる。ただし、エラーを検出するか否かは実装依存である。

【補足事項】

複数のタスクが待っている可変長メモリプールに対してメモリを返却する場合は､要求メモリ数との関係により、複数のタスクが同時に待ち解除となることがある。この場合の待ち解除後のタスクの優先順位は、同じ優先度を持つタスクの間では待ち行列に並んでいたときと同じ順序となる。

可変長メモリプール状態参照 tk_ref_mpl

tk_ref_mpl:Refer Variable-size MemoryPool Status

【C 言語インタフェース】

```
ER ercd = tk_ref_mpl ( ID mplid, T_RMPL *pk_rmpl ) ;
```

【パラメータ】

ID	mplid	MemoryPoolID	可変長メモリプール ID
T_RMPL*	pk_rmpl	Packet to Refer MemoryPool	メモリプール状態を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_rmpl の内容

VP	exinf	ExtendedInformation	拡張情報
ID	wtsk	WaitTaskInformation	待ちタスクの有無
INT	frsz	FreeMemorySize	空き領域の合計サイズ(バイト数)
INT	maxsz	MaxMemorySize	最大の空き領域のサイズ(バイト数)

――(以下に実装独自に他の情報を追加してもよい)――

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(mplid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mplid の可変長メモリプールが存在しない)
E_PAR	パラメータエラー(pk_rmpl が不正)

【解説】

mplid で示された対象可変長メモリプールの各種の状態を参照し、リターンパラメータとして現在の空き領域の合計サイズ frsz、すぐに獲得可能な最大の空き領域のサイズ maxsz、待ちタスクの有無(wtsk)、拡張情報(exinf)を返す。

wtsk は、この可変長メモリプールで待っているタスクの ID を示す。この可変長メモリプールで複数のタスクが待っている場合には、待ち行列の先頭のタスクの ID を返す。待ちタスクが無い場合は wtsk＝0 となる。

tk_ref_mpl で、対象可変長メモリプールが存在しない場合には、エラーE_NOEXS となる。

4.7 時間管理機能

時間管理機能は、時間に依存した処理を行うための機能である。システム時刻管理、周期ハンドラ、アラームハンドラの各機能が含まれる。
周期ハンドラとアラームハンドラを総称して、タイムイベントハンドラと呼ぶ。

4.7.1 システム時刻管理

システム時刻管理機能は、システム時刻を操作するための機能である。システム時刻を設定／参照する機能、システム稼働時間を参照する機能が含まれる。

システム時刻設定 tk_set_tim

tk_set_tim:Set Time

【C 言語インタフェース】

```
ER ercd = tk_set_tim ( SYSTIM *pk_tim ) ;
```

【パラメータ】

SYSTIM*	pk_tim	Packet of CurrentTime	現在時刻を示すパケット

pk_tim の内容

W	hi	high 32bits	システム時刻設定用現在時刻の上位 32 ビット
UW	lo	low 32bits	システム時刻設定用現在時刻の下位 32 ビット

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_PAR	パラメータエラー(pk_tim が不正、設定時間が不正)

【解説】
システム時刻の値を pk_tim で示される値に設定する。
システム時刻は、1985 年 1 月 1 日 0 時(GMT)からの通算のミリ秒数とする。

【補足事項】
システムの動作中に tk_set_tim を使ってシステム時刻を変更した場合にも、RELTIM や TMO で指定された相対時間は変化しない。例えば、60 秒後にタイムアウトする様に指定した場合、タイムアウト待ちの間に tk_set_tim で時間を 60 秒進めてもそこでタイムアウトすることはなく、60 秒後にタイムアウトする。したがって、tk_set_tim によってタイムアウトするシステム時刻は変化することになる。

tk_get_tim:Get Time

【C 言語インタフェース】

```
ER ercd = tk_get_tim ( SYSTIM *pk_tim ) ;
```

【パラメータ】

SYSTIM*	pk_tim	Packet of CurrentTime	現在時刻を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_tim の内容

W	hi	high 32bits	システムの現在時刻の上位 32 ビット
UW	lo	low 32bits	システムの現在時刻の下位 32 ビット

【エラーコード】

E_OK	正常終了
E_PAR	パラメータエラー(pk_tim が不正)

【解説】
システム時刻の現在の値を読み出し、リターンパラメータ pk_tim に返す。
システム時刻は、1985 年 1 月 1 日 0 時(GMT)からの通算のミリ秒数とする。

システム稼働時間参照 tk_get_otm

tk_get_otm:Get Operating Time

【C 言語インタフェース】

```
ER ercd = tk_get_otm ( SYSTIM *pk_tim ) ;
```

【パラメータ】

SYSTIM*	pk_tim	Packet of Operating Time	稼働時間を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_tim の内容

W	hi	high 32bits	システム稼働時間の上位 32 ビット
UW	lo	low 32bits	システム稼働時間の下位 32 ビット

【エラーコード】

E_OK	正常終了
E_PAR	パラメータエラー(pk_tim が不正)

【解説】

システム稼働時間を取得する。

システム稼働時間はシステム時刻(時刻)と異なり、システム起動時からの単純増加する稼働時間を表す。tk_set_tim による時刻設定に影響されない。

システム稼働時間はシステム時刻と同じ精度でなければならない。

4.7.2 周期ハンドラ

周期ハンドラは、一定周期で起動されるタイムイベントハンドラである。周期ハンドラ機能には、周期ハンドラを生成／削除する機能、周期ハンドラの動作を開始／停止する機能、周期ハンドラの状態を参照する機能が含まれる。周期ハンドラは ID 番号で識別されるオブジェクトである。周期ハンドラの ID 番号を周期ハンドラ ID と呼ぶ。

周期ハンドラの起動周期と起動位相は、周期ハンドラの生成時に、周期ハンドラ毎に設定することができる。カーネルは、周期ハンドラの操作時に、設定された起動周期と起動位相から、周期ハンドラを次に起動すべき時刻を決定する。周期ハンドラの生成時には、周期ハンドラを生成した時刻に起動位相を加えた時刻を、次に起動すべき時刻とする。周期ハンドラを起動すべき時刻になると、その周期ハンドラの拡張情報(exinf)をパラメータとして、周期ハンドラを起動する。またこの時、周期ハンドラの起動すべき時刻に起動周期を加えた時刻を、次に起動すべき時刻とする。また、周期ハンドラの動作を開始する時に、次に起動すべき時刻を決定しなおす場合がある。

周期ハンドラの起動位相は、起動周期以下であることを原則とする。起動位相に、起動周期よりも長い時間が指定された場合の振舞いは、実装依存とする。

周期ハンドラは、動作している状態か動作していない状態かのいずれかの状態をとる。周期ハンドラが動作していない状態の時には、周期ハンドラを起動すべき時刻となっても周期ハンドラを起動せず、次に起動すべき時刻の決定のみを行う。周期ハンドラの動作を開始するシステムコール(tk_sta_cyc)が呼び出されると、周期ハンドラを動作している状態に移行させ、必要なら周期ハンドラを次に起動すべき時刻を決定しなおす。周期ハンドラの動作を停止するシステムコール(tk_stp_cyc)が呼び出されると、周期ハンドラを動作していない状態に移行させる。周期ハンドラを生成した後にどちらの状態になるかは、周期ハンドラ属性によって決めることができる。

周期ハンドラの起動位相は、周期ハンドラを生成するシステムコールが呼び出された時刻を基準に、周期ハンドラを最初に起動する時刻を指定する相対時間と解釈する。周期ハンドラの起動周期は、周期ハンドラを(起動した時刻ではなく)起動すべきであった時刻を基準に、周期ハンドラを次に起動する時刻を指定する相対時間と解釈する。そのため、周期ハンドラが起動される時刻の間隔は、個々には起動周期よりも短くなる場合があるが、長い期間で平均すると起動周期に一致する。

周期ハンドラの生成 tk_cre_cyc

tk_cre_cyc:Create Cyclic Handler

【C 言語インタフェース】

```
ID cycid = tk_cre_cyc ( T_CCYC *pk_ccyc ) ;
```

【パラメータ】

```
T_CCYC* pk_ccyc      Packet to Create CyclicHandler  周期ハンドラ定義情報
```

pk_ccyc の内容

VP	exinf	ExtendedInformation	拡張情報
ATR	cycatr	CyclicHandlerAttribute	周期ハンドラ属性
FP	cychdr	CyclicHandlerAddress	周期ハンドラアドレス
RELTIM	cyctim	CycleTime	周期起動時間間隔(ミリ秒)
RELTIM	cycphs	CyclePhase	周期起動位相(ミリ秒)
UB	dsname[8]	DS Object name	DS オブジェクト名称

――(以下に実装独自に他の情報を追加してもよい)――

【リターンパラメータ】

ID	cycid	CyclicHandlerID	周期ハンドラ ID
	または	ErrorCode	エラーコード

【エラーコード】

E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_LIMIT	周期ハンドラの数がシステムの制限を超えた
E_RSATR	予約属性(cycatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_ccyc, cychdr, cyctim, cycphs が不正あるいは利用できない)

【解説】

周期ハンドラを生成し周期ハンドラ ID を割当てる。具体的には、生成された周期ハンドラに対して管理ブロックを割り付ける。

周期ハンドラは、指定した時間間隔で動くタスク独立部のハンドラである。

exinf は、対象となる周期ハンドラに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、周期ハンドラにパラメータとして渡される他、tk_ref_cyc で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。OS では exinf の内容について関知しない。

cycatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。cycatr のシステム属性の部分では、次のような指定を行う。

```
cycatr := (TA_ASM || TA_HLNG) | [TA_STA] | [TA_PHS] | [TA_DSNAME]
    TA_ASM          対象ハンドラがアセンブラで書かれている
    TA_HLNG         対象ハンドラが高級言語で書かれている
    TA_STA          周期ハンドラ生成後直ちに起動する
    TA_PHS          起動位相を保存する
    TA_DSNAME       DS オブジェクト名称を指定する

#define TA_ASM      0x00000000  /* アセンブラによるプログラム */
#define TA_HLNG     0x00000001  /* 高級言語によるプログラム */
#define TA_STA      0x00000002  /* 周期ハンドラ起動 */
```

```
#define TA_PHS      0x00000004  /* 周期ハンドラ起動位相を保存 */
#define TA_DSNAME   0x00000040  /* DS オブジェクト名称を指定 */
```

cychdr は周期ハンドラの先頭アドレス、cyctim は周期起動の時間間隔、cycphs は起動位相を表す。

TA_HLNG 属性の場合は、高級言語対応ルーチンを経由して周期ハンドラを起動する。高級言語対応ルーチンによって、レジスタの退避と復帰が行われる。周期ハンドラからは、単純な関数からのリターンによって終了する。TA_HLNG 属性の場合の周期ハンドラは次の形式となる。

```
void cychdr( VP exinf )
{
    /*
        処理
     */

    return; /* 周期ハンドラの終了 */
}
```

TA_ASM 属性の場合の周期ハンドラの形式は実装定義とする。ただし、起動パラメータとして exinf を渡さなければならない。

cycphs は tk_cre_cyc によって周期ハンドラを生成してから最初の周期ハンドラの起動までの時間を表す。その後は、cyctim 間隔で周期起動を繰り返す。cycphs に 0 を指定した場合は、周期ハンドラの生成直後に周期ハンドラが起動されることになる。cyctim に 0 を指定することはできない。

周期ハンドラの n 回目の起動は、周期ハンドラを生成してから cycphs + cyctim * (n - 1)以上の時間が経過した後に行う。

TA_STA を指定した場合は、周期ハンドラの生成時から周期ハンドラは動作状態となり、前述の時間間隔で周期ハンドラが起動される。TA_STA を指定しなかった場合は、起動周期の計測は行われるが、周期ハンドラは起動されない。

TA_PHS が指定されている場合は、tk_sta_cyc によって周期ハンドラが活性化されても、起動周期はリセットされず前述のように周期ハンドラの生成時から計測している周期を維持する。TA_PHS が指定されていない場合は、tk_sta_cyc によって起動周期がリセットされ、tk_sta_cyc 呼出時から cyctim 間隔で周期ハンドラが起動される。tk_sta_cyc によるリセットでは、cycphs は適用されない。この場合、tk_sta_cyc から n 回目の周期ハンドラの起動は、tk_sta_cyc 呼出時から cyctim * n 以上の時間が経過した後となる。

周期ハンドラの中でシステムコールを発行することにより､それまで実行状態(RUNNING)であったタスクがその他の状態に移行し､代わりに別のタスクが実行状態(RUNNING)となった場合でも､周期ハンドラ実行中はディスパッチ(実行タスクの切り替え)が起こらない。ディスパッチングが必要になっても、まず周期ハンドラを最後まで実行することが優先され、周期ハンドラを終了する時にはじめてディスパッチが行われる。すなわち、周期ハンドラ実行中に生じたディスパッチ要求はすぐに処理されず、周期ハンドラ終了までディスパッチが遅らされる。これを遅延ディスパッチの原則と呼ぶ。

周期ハンドラはタスク独立部として実行される。したがって、周期ハンドラの中では、待ち状態に入るシステムコールや、自タスクの指定を意味するシステムコールを実行することはできない。

TA_DSNAME を指定した場合に dsname が有効となり、DS オブジェクト名称として設定される。DS オブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DS のシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname、td_set_dsname を参照のこと。

TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

【補足事項】

tk_cre_cyc では回数の指定が無いので、一旦定義された周期ハンドラは、tk_stp_cyc によって停止するか、周期起動ハンドラを削除するまで周期起動が繰り返される。

複数のタイムイベントハンドラや割込みハンドラが同時に動く場合に、それらのハンドラがシリアルに起動される(1 つのハンドラの実行を終了してから別のハンドラの実行を始める)か、ネストして起動される(1 つのハンドラの実行を中断して別のハンドラを実行し､そのハンドラが終了した後で前のハンドラの続きを実行する)かということは実装依存である。いずれにしても、タイムイベントハンドラや割込みハンドラはタスク独立部なので、遅延ディスパッチの原則が適用される。

周期ハンドラの削除 tk_del_cyc

tk_del_cyc:Delete Cyclic Handler

【C 言語インタフェース】

```
ER ercd = tk_del_cyc ( ID cycid ) ;
```

【パラメータ】

ID	cycid	CyclicHandlerID	周期ハンドラ ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(cycid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(cycid の周期ハンドラが存在しない)

【解説】
周期ハンドラを削除する。

周期ハンドラの動作開始 tk_sta_cyc

tk_sta_cyc:Start Cyclic Handler

【C 言語インタフェース】

```
ER ercd = tk_sta_cyc ( ID cycid ) ;
```

【パラメータ】

ID	cycid	CyclicHandlerID	周期ハンドラ ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(cycid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(cycid の周期ハンドラが存在しない)

【解説】

周期ハンドラを動作状態にする。

TA_PHS 属性を指定している場合は、周期ハンドラの起動周期はリセットされず、周期ハンドラを動作状態にする。すでに動作状態であれば何もせずに動作状態を維持する。

TA_PHS 属性を指定していない場合は、起動周期をリセットして、周期ハンドラを動作状態にする。すでに動作状態であれば起動周期のリセットをした上で、動作状態を維持する。したがって、次に周期ハンドラが起動されるのは、cyctim 後となる。

周期ハンドラの動作停止 tk_stp_cyc

tk_stp_cyc:Stop Cyclic Handler

【C 言語インタフェース】

```
ER ercd = tk_stp_cyc ( ID cycid ) ;
```

【パラメータ】

ID	cycid	CyclicHandlerID	周期ハンドラ ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(cycid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(cycid の周期ハンドラが存在しない)

【解説】
周期ハンドラを動作していない状態にする。すでに動作していない状態であれば何もしない。

周期ハンドラ状態参照 tk_ref_cyc

tk_ref_cyc:Refer Cyclic Handler Status

【C 言語インタフェース】

```
ER ercd = tk_ref_cyc ( ID cycid, T_RCYC *pk_rcyc ) ;
```

【パラメータ】

ID	cycid	CyclicHandlerID	周期ハンドラ ID
T_RCYC*	pk_rcyc	Packet to Refer CyclicHandler	周期ハンドラの状態を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_rcyc の内容

VP	exinf	ExtendedInformation	拡張情報
RELTIM	lfttim	LeftTime	次のハンドラ起動までの残り時間
UINT	cycstat	CyclicHandlerStatus	周期ハンドラの状態

――(以下に実装独自に他の情報を追加してもよい)――

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(cycid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(cycid の周期ハンドラが存在していない)
E_PAR	パラメータエラー(pk_rcyc が不正)

【解説】

cycid で示された周期ハンドラの状態を参照し、周期ハンドラの状態 cycstat、次のハンドラ起動までの残り時間 lfttim、拡張情報 exinf をリターンパラメータとして返す。

cycstat には次の情報が返される。

```
cycstat:= (TCYC_STP | TCYC_STA)

#define TCYC_STP    0x00    /* 周期ハンドラが動作していない */
#define TCYC_STA    0x01    /* 周期ハンドラが動作している */
```

lfttim には、次に周期ハンドラが起動される予定の時刻までの残り時間(ミリ秒)が返される。周期ハンドラが現在起動中か停止中かは関係しない。

exinf には、周期ハンドラを生成する際のパラメータとして指定された拡張情報が返される。exinf は周期ハンドラに引数として渡される。

tk_ref_cyc で、cycid の周期ハンドラが存在していない場合には、エラーE_NOEXS となる。

4.7.3 アラームハンドラ

アラームハンドラは、指定した時刻に起動されるタイムイベントハンドラである。アラームハンドラ機能には、アラームハンドラを生成／削除する機能、アラームハンドラの動作を開始／停止する機能、アラームハンドラの状態を参照する機能が含まれる。アラームハンドラは ID 番号で識別されるオブジェクトである。アラームハンドラの ID 番号をアラームハンドラ ID と呼ぶ。

アラームハンドラを起動する時刻(これをアラームハンドラの起動時刻と呼ぶ)は、アラームハンドラ毎に設定することができる。アラームハンドラの起動時刻になると、そのアラームハンドラの拡張情報(exinf)をパラメータとして、アラームハンドラを起動する。

アラームハンドラの生成直後には、アラームハンドラの起動時刻は設定されておらず、アラームハンドラの動作は停止している。アラームハンドラの動作を開始するシステムコール(tk_sta_alm)が呼び出されると、アラームハンドラの起動時刻を、システムコールが呼び出された時刻から指定された相対時間後に設定する。アラームハンドラの動作を停止するシステムコール(tk_stp_alm)が呼び出されると、アラームハンドラの起動時刻の設定を解除する。また、アラームハンドラを起動する時にも、アラームハンドラの起動時刻の設定を解除し、アラームハンドラの動作を停止する。

アラームハンドラの生成 tk_cre_alm

tk_cre_alm:Create Alarm Handler

【C 言語インタフェース】

```
ID almid = tk_cre_alm ( T_CALM *pk_calm ) ;
```

【パラメータ】

```
T_CALM* pk_calm    Packet to Create AlarmHandler    アラームハンドラ定義情報
```

pk_calm の内容

VP	exinf	ExtendedInformation	拡張情報
ATR	almatr	AlarmHandlerAttribute	アラームハンドラ属性
FP	almhdr	AlarmHandlerAddress	アラームハンドラアドレス
UB	dsname[8]	DS Object name	DS オブジェクト名称

――(以下に実装独自に他の情報を追加してもよい)――

【リターンパラメータ】

ID	almid	AlarmHandlerID	アラームハンドラ ID
	または	ErrorCode	エラーコード

【エラーコード】

E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_LIMIT	アラームハンドラの数がシステムの制限を超えた
E_RSATR	予約属性(almatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_calm, almatr, almhdr が不正あるいは利用できない)

【解説】

アラームハンドラを生成しアラームハンドラ ID を割当てる。具体的には、生成されたアラームハンドラに対して管理ブロックを割り付ける。

アラームハンドラ(指定時刻起動ハンドラ)は、指定した時刻に起動されるタスク独立部のハンドラである。

exinf は、対象となるアラームハンドラに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、アラームハンドラにパラメータとして渡される他、tk_ref_alm で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。OS では exinf の内容について関知しない。

almatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。almatr のシステム属性の部分では、次のような指定を行う。

```
almatr := (TA_ASM || TA_HLNG) | [TA_DSNAME]
```

TA_ASM	対象ハンドラがアセンブラで書かれている
TA_HLNG	対象ハンドラが高級言語で書かれている
TA_DSNAME	DS オブジェクト名称を指定する

```
#define TA_ASM          0x00000000  /* アセンブラによるプログラム */
#define TA_HLNG         0x00000001  /* 高級言語によるプログラム */
#define TA_DSNAME       0x00000040  /* DS オブジェクト名称を指定 */
```

almhdr は起動されるアラームハンドラの先頭アドレスを表す。

TA_HLNG 属性の場合は、高級言語対応ルーチンを経由してアラームハンドラを起動する。高級言語対応ルーチン

によって、レジスタの退避と復帰が行われる。アラームハンドラからは、単純な関数からのリターンによって終了する。TA_HLNG 属性の場合のアラームハンドラは次の形式となる。

```
void almhdr( VP exinf )
{
      /*
              処理
      */

      return; /* アラームハンドラの終了 */
}
```

TA_ASM 属性の場合のアラームハンドラの形式は実装定義とする。ただし、起動パラメータとして exinf を渡さなければならない。

アラームハンドラの中でシステムコールを発行することにより､それまで実行状態(RUNNING)であったタスクがその他の状態に移行し、代わりに別のタスクが実行状態(RUNNING)となった場合でも、アラームハンドラ実行中はディスパッチ(実行タスクの切り替え)が起こらない。ディスパッチングが必要になっても、まずアラームハンドラを最後まで実行することが優先され、アラームハンドラを終了する時にはじめてディスパッチが行われる。すなわち、アラームハンドラ実行中に生じたディスパッチ要求はすぐに処理されず、アラームハンドラ終了までディスパッチが遅らされる。これを遅延ディスパッチの原則と呼ぶ。

アラームハンドラはタスク独立部として実行される。したがって、アラームハンドラの中では、待ち状態に入るシステムコールや、自タスクの指定を意味するシステムコールを実行することはできない。

TA_DSNAME を指定した場合に dsname が有効となり、DS オブジェクト名称として設定される。DS オブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DS のシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname、td_set_dsname を参照のこと。

TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

【補足事項】

複数のタイムイベントハンドラや割込みハンドラが同時に動く場合に、それらのハンドラがシリアルに起動される(1 つのハンドラの実行を終了してから別のハンドラの実行を始める)か、ネストして起動される(1 つのハンドラの実行を中断して別のハンドラを実行し､そのハンドラが終了した後で前のハンドラの続きを実行する)かということは実装依存である。いずれにしても、タイムイベントハンドラや割込みハンドラはタスク独立部なので、遅延ディスパッチの原則が適用される。

アラームハンドラの削除 tk_del_alm

tk_del_alm:Delete Alarm Handler

【C 言語インタフェース】

```
ER ercd = tk_del_alm ( ID almid ) ;
```

【パラメータ】

ID	almid	AlarmHandlerID	アラームハンドラ ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(almid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(almid のアラームハンドラが存在しない)

【解説】
アラームハンドラを削除する。

アラームハンドラの動作開始 tk_sta_alm

tk_sta_alm:Start Alarm Handler

【C 言語インタフェース】

```
ER ercd = tk_sta_alm ( ID almid, RELTIM almtim ) ;
```

【パラメータ】

ID	almid	AlarmHandlerID	アラームハンドラ ID
RELTIM	almtim	AlarmTime	アラームハンドラ起動時刻

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(almid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(almid のアラームハンドラが存在しない)

【解説】

アラームハンドラの起動時刻を設定して、動作状態にする。almtim は相対時刻で、tk_sta_alm が呼び出された時刻から almtim で指定した時間が経過した後にアラームハンドラが起動される。すでにアラームハンドラの起動時刻が設定されて動作状態であった場合には、その設定を解除した後、新たに起動時刻を設定し動作状態とする。

almtim=0 の場合には、起動時刻設定直後にアラームハンドラが起動される。

アラームハンドラの動作停止 tk_stp_alm

tk_stp_alm:Stop Alarm Handler

【C 言語インタフェース】

```
ER ercd = tk_stp_alm ( ID almid ) ;
```

【パラメータ】

ID	almid	AlarmHandlerID	アラームハンドラ ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(almid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(almid のアラームハンドラが存在しない)

【解説】
アラームハンドラの起動時刻を解除し、動作していない状態にする。すでに動作していない状態であれば何もしない。

アラームハンドラ状態参照 tk_ref_alm

tk_ref_alm:Refer Alarm Handler Status

【C 言語インタフェース】

```
ER ercd = tk_ref_alm ( ID almid, T_RALM *pk_ralm ) ;
```

【パラメータ】

ID	almid	AlarmHandlerID	アラームハンドラ ID
T_RALM*	pk_ralm	Packet to Refer AlarmHandler	アラームハンドラの状態を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_ralm の内容

VP	exinf	ExtendedInformation	拡張情報
RELTIM	lfttim	LeftTime	ハンドラ起動までの残り時間
UINT	almstat	Alarm Handler Status	アラームハンドラの状態

――(以下に実装独自に他の情報を追加してもよい)――

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(almid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(almid のアラームハンドラが存在しない)
E_PAR	パラメータエラー(pk_ralm が不正)

【解説】

almid で示されたアラームハンドラの状態を参照し、ハンドラ起動までの残り時間 lfttim、拡張情報 exinf をリターンパラメータとして返す。

almstat には次の情報が返される。

```
almstat:= (TALM_STP | TALM_STA)

#define TALM_STP   0x00   /* アラームハンドラが動作していない */
#define TALM_STA   0x01   /* アラームハンドラが動作している */
```

アラームハンドラが動作している(TALM_STA)場合、lfttim には次にアラームハンドラが起動されるまでの相対時間が返される。tk_sta_alm で指定した almtim ≧ lfttim ≧ 0 の範囲の値となる。lfttim はタイマー割込みごとに減算されるため、次のタイマー割込みでアラームハンドラが起動される場合に lfttim= 0 となる。

exinf には、アラームハンドラを生成する際のパラメータとして指定された拡張情報が返される。exinf はアラームハンドラに引数として渡される。

アラームハンドラが動作していない(TALM_STP)場合は、lfttim は不定である。

tk_ref_alm で、almid のアラームハンドラが存在していない場合には、エラーE_NOEXS となる。

4.8 割込み管理機能

割込み管理機能は、外部割込みおよび CPU 例外に対するハンドラの定義などの操作を行う機能である。

割込みハンドラは、タスク独立部として扱われる。タスク独立部でも、タスク部と同じ形式でシステムコールを発行することが可能であるが、タスク独立部で発行できるシステムコールには以下のような制限がつく。

- 暗黙に自タスクを指定するシステムコールや自タスクを待ち状態にするシステムコールは発行することができない。E_CTX のエラーとなる。

タスク独立部の実行中はタスクの切り換え(ディスパッチ)は起らず、システムコールの処理の結果ディスパッチの要求が出されても、タスク独立部を抜けるまでディスパッチが遅らされる。これを遅延ディスパッチ(delayed dispatching)の原則と呼ぶ。

割込みハンドラ定義 tk_def_int

tk_def_int:Define Interrupt Handler

【C 言語インタフェース】

```
ER ercd = tk_def_int ( UINT dintno, T_DINT *pk_dint ) ;
```

【パラメータ】

UINT	dintno	InterruptDefineNumber	割込みハンドラ番号
T_DINT*	pk_dint	Packet to Define InterruptHandler	割込みハンドラ定義情報

pk_dint の内容

ATR	intatr	InterruptHandlerAttribute	割込みハンドラ属性
FP	inthdr	InterruptHandlerAddress	割込みハンドラアドレス

――(以下に実装独自に他の情報を追加してもよい)――

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_RSATR	予約属性(intatr が不正あるいは利用できない)
E_PAR	パラメータエラー(dintno, pk_dint, inthdr が不正あるいは利用できない)

【解説】

割込みには、デバイスからの外部割込みと、CPU 例外による割込みの両方を含む。

dintno で示される割込みハンドラ番号に対して割込みハンドラを定義し、割込みハンドラを使用可能にする。すなわち、dintno で示される割込みハンドラ番号と割込みハンドラのアドレスや属性との対応付けを行う。

dintno の具体的な意味は実装ごとに定義されるが、一般に割込みベクトル番号などを表す。

intatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。intatr のシステム属性の部分では、次のような指定を行う。

```
intatr := (TA_ASM || TA_HLNG)
```

TA_ASM	対象ハンドラがアセンブラで書かれている
TA_HLNG	対象ハンドラが高級言語で書かれている

```
#define TA_ASM     0x00000000  /* アセンブラによるプログラム */
#define TA_HLNG    0x00000001  /* 高級言語によるプログラム */
```

TA_ASM 属性の場合、原則として、割込みハンドラの起動時には OS が介入しない。割込み発生時には、CPU ハードウェアの割込み処理機能(実装によっては T-Monitor による処理が含まれる場合もある)により、このシステムコールで定義した割込みハンドラが直接起動される。したがって、割込みハンドラの先頭と最後では、割込みハンドラで使用するレジスタの退避と復帰を行う必要がある。割込みハンドラは、tk_ret_int システムコールまたは CPU の割込みリターン命令(またはそれに相当する手段)によって終了する。

tk_ret_int を使用せずに OS を介すことなく割込みハンドラから復帰する手段は必須である。ただし、tk_ret_int を使用しない場合は、遅延ディスパッチが行われなくてもよい。

tk_ret_int による割込みハンドラからの復帰も必須であり、この場合は遅延ディスパッチが行われなければならない。

TA_HLNG 属性の場合は、高級言語対応ルーチンを経由して割込みハンドラを起動する。高級言語対応ルーチンによって、レジスタの退避と復帰が行われる。割込みハンドラは、C 言語関数からのリターンによって終了する。TA_HLNG 属性の場合の割込みハンドラは次の形式となる。

```
void inthdr( UINT dintno )
{
      /*
              割込み処理
      */
      return; /* 割込みハンドラの終了 */
}
```

割込みハンドラに渡される dintno は、発生した割込みの割込みハンドラ番号で、tk_def_int で指定するものと同一である。実装によっては、dintno 以外にも発生した割込みに関する情報が渡される場合がある。このような情報がある場合は、割込みハンドラの 2 番目以降のパラメータとして実装ごとに定義する。

TA_HLNG 属性の場合、割込み発生から割込みハンドラが呼び出されるまで、CPU の割込みフラグは割込み禁止状態であるものとする。つまり、割込み発生直後から多重割込みが禁止された状態であり、多重割込みが禁止された状態のまま割込みハンドラが呼び出される。多重割込みを許可する場合は、割込みハンドラ内で CPU の割込みフラグを操作して許可する必要がある。

また、TA_HLNG 属性の場合は、割込みハンドラに入った時点で、システムコールの呼出が可能な状態となっていなければならない。ただし、上述の機能を標準とした上で、多重割込みを許可した状態で割込みハンドラに入る機能を追加するなどの拡張は許される。

TA_ASM 属性の場合、割込みハンドラに入った時点の状態は実装ごとに定義される。割込みハンドラに入った時点のスタックやレジスタの状態、システムコールの呼出の可否、システムコールを呼び出せるようにする方法、および割込みハンドラから OS を介さずに復帰する方法などが明確に定義されていなければならない。

TA_ASM 属性の場合、実装によっては、割込みハンドラ実行中もタスク独立部として判定されない場合がある。タスク独立部として判定されない場合は、次の点に注意が必要である。

- 割込みを許可すると、タスクディスパッチが発生する可能性がある。
- システムコールを呼出した場合、タスク部または準タスク部から呼出したものとして処理される。

なお、割込みハンドラ内で何らかの操作を行ってタスク独立部として判定させる方法がある場合は、実装ごとにその方法を示すものとする。

TA_HLNG, TA_ASM 属性のいずれの場合も、割込みハンドラに入った時点では、割込みが発生した時点の論理空間を維持している。また、割込みハンドラからの復帰時に論理空間を割込み発生時点の状態に戻す処理は行われない。割込みハンドラ内での論理空間の切替は禁止されないが、論理空間を切り替えた場合の影響も OS では関知しない。

割込みハンドラの中でシステムコールを発行することにより、それまで実行状態(RUNNING)であったタスクがその他の状態に移行し、代わりに別のタスクが実行状態(RUNNING)となった場合でも、割込みハンドラ実行中はディスパッチ(実行タスクの切り替え)が起こらない。ディスパッチングが必要になっても、まず割込みハンドラを最後まで実行することが優先され、割込みハンドラを終了する時にはじめてディスパッチが行われる。すなわち、割込みハンドラ実行中に生じたディスパッチ要求はすぐに処理されず、割込みハンドラ終了までディスパッチが遅らされる。これを遅延ディスパッチの原則と呼ぶ。

割込みハンドラはタスク独立部として実行される。したがって、割込みハンドラの中では、待ち状態に入るシステムコールや、自タスクの指定を意味するシステムコールを実行することはできない。

pk_dint＝NULL とした場合には、前に定義した割込みハンドラの定義解除を行う。割込みハンドラの定義解除状態では、T-Monitor の定義するデフォルトハンドラが設定される。

既に定義済みの割込みハンドラ番号に対して、割込みハンドラを再定義することも可能である。再定義のときにも、あらかじめその番号のハンドラの定義解除を行っておく必要はない。既に割込みハンドラが定義された dintno に対して新しいハンドラを再定義しても、エラーにはならない。

【補足事項】

TA_ASM 属性に関する種々の規定は、主に割込みのフックを行うためのものである。例えば、不正アドレスのアクセスによる例外の場合、通常は上位のプログラムで定義した割込みハンドラによって例外を検出してエラー処理を行うが、デバッグ中の場合は上位のプログラムでエラー処理を行う代りに T-Monitor の割込みハンドラに処理させてデバッガを起動するというようなことを行う。この場合、上位のプログラムで定義した割込みハンドラは、T-Monitor の割込みハンドラをフックする形になる。そして、状況に応じて T-Monitor に割込み処理を引き渡すか、そのまま自身で処理することになる。

割込みハンドラから復帰 tk_ret_int

tk_ret_int:Return from Interrupt Handler

【C 言語インタフェース】

```
void tk_ret_int ( void ) ;
```

C 言語インタフェースは形式として定義されるが、高級言語対応ルーチンを使用した場合には呼び出されることがない。

【パラメータ】

なし

【リターンパラメータ】

※ システムコールを発行したコンテキストには戻らない

【エラーコード】

※ 次のようなエラーを検出する可能性があるが、エラーを検出した場合でも、システムコールを発行したコンテキストには戻らない。したがって、システムコールのリターンパラメータとして直接エラーコードを返すことはできない。万一エラーを検出した場合の動作は、実装依存となる。

E_CTX	コンテキストエラー(割込みハンドラ以外から発行,実装依存)

【解説】

割込みハンドラを終了する。

割込みハンドラの中でシステムコールを実行してもディスパッチは起らず、tk_ret_int によって割込みハンドラを終了するまでディスパッチが遅延させられる(遅延ディスパッチの原則)。そのため、tk_ret_int では、割込みハンドラの中から発行されたシステムコールによるディスパッチ要求がまとめて処理される。

tk_ret_int は TA_ASM 属性の割込みハンドラの場合にのみ呼び出す。TA_HLNG 属性の割込みハンドラの場合は、高級言語対応ルーチン内で暗黙に tk_ret_int 相当の機能が実行されるので、tk_ret_int を明示的には呼び出さない(呼出してはいけない)。

TA_ASM 属性の場合、原則として、割込みハンドラの起動時には OS が介入しない。割込み発生時には、CPU ハードウェアの割込み処理機能により、割込みハンドラが直接起動される。そのため、割込みハンドラで使用するレジスタの退避や復帰は割込みハンドラの中で行わなければならない。

また、この理由により、tk_ret_int を発行した時のスタックやレジスタの状態は、割込みハンドラへ入った時と同じでなければならず、この関係で tk_ret_int では機能コードを使用できないことがある。この場合は、他のシステムコールとは別ベクトルのトラップ命令を用いて tk_ret_int を実現する。

【補足事項】

tk_ret_int は発行元のコンテキストに戻らないシステムコールである。したがって、何らかのエラーを検出した場合にエラーコードを返しても、これらのシステムコールを呼んだ側ではエラーのチェックを行っていないのが普通であり、プログラムが暴走する可能性がある。そこで、これらのシステムコールでは、エラーを検出した場合にも、システムコール発行元へは戻らないものとする。

割込みハンドラから戻っても、ディスパッチが起こらない(必ず同じタスクが実行を継続する)ことがはっきりしている、またはディスパッチしなくても構わない場合には、tk_ret_int ではなく、アセンブラの割込みリターン命令によって割込みハンドラを終了しても構わない。

また、CPU のアーキテクチャや OS の構成法によっては、アセンブラの割込みリターン命令によって割込みハンドラを終了しても、遅延ディスパッチが可能となる場合がある。このような場合は、アセンブラの割り込みリターン命令がそのまま tk_ret_int のシステムコールであると解釈しても構わない。

タイムイベントハンドラから tk_ret_int を呼んだ場合の E_CTX のエラーチェックは実装依存である。実装によっては、種類の違うハンドラからそのままリターンしてしまう場合がある。

4.9 システム状態管理機能

システム状態管理機能は、システムの状態を変更／参照するための機能である。タスクの優先順位を回転する機能、実行状態のタスク ID を参照する機能、タスクディスパッチを禁止／解除する機能、コンテキストやシステム状態を参照する機能、省電力モードを設定する設定、カーネルのバージョンを参照する機能が含まれる。

タスクの優先順位の回転 tk_rot_rdq

tk_rot_rdq:Rotate Ready Queue

【C 言語インタフェース】

```
ER ercd = tk_rot_rdq ( PRI tskpri ) ;
```

【パラメータ】

PRI	tskpri	TaskPriority	タスク優先度

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_PAR	パラメータエラー(tskpri が不正)

【解説】

tskpri で指定される優先度のタスクの優先順位を回転する。すなわち、対象優先度を持った実行できる状態のタスクの中で、最も高い優先順位を持つタスクを、同じ優先度を持つタスクの中で最低の優先順位とする。

tskpri＝TPRI_RUN＝0 により、その時実行状態(RUNNING)にあるタスクの優先度のタスクの優先順位を回転する。一般のタスクから発行される tk_rot_rdq では、これは自タスクの持つ優先度のタスクの優先順位を回転するのと同じ意味であるが、周期ハンドラなどのタスク独立部から tk_rot_rdq(tskpri=TPRI_RUN)を発行することも可能である。

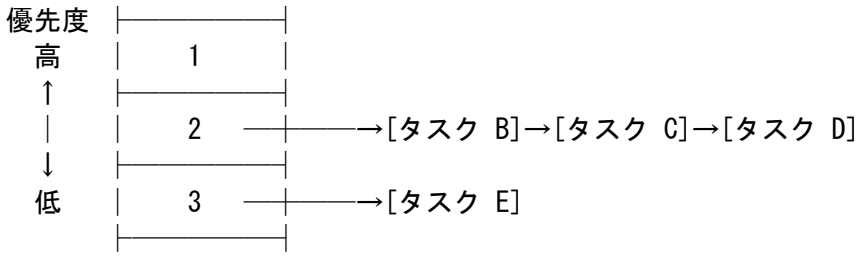
【補足事項】

対象優先度を持った実行できる状態のタスクがない場合や、一つしかない場合には、何もしない(エラーとはしない)。

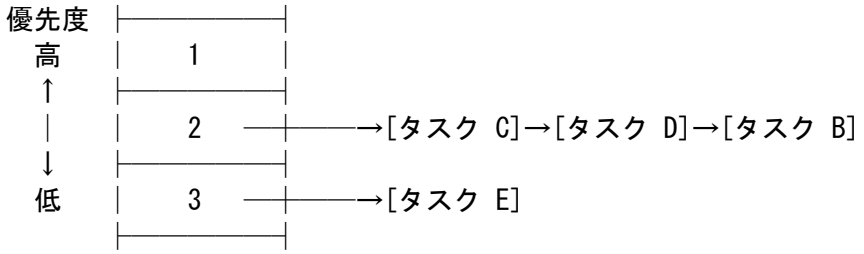
ディスパッチ許可状態で、対象優先度に TPRI_RUN または自タスクの現在優先度を指定してこのシステムコールが呼び出されると、自タスクの実行順位は同じ優先度を持つタスクの中で最低となる。そのため、このシステムコールを用いて、実行権の放棄を行うことができる。

ディスパッチ禁止状態では、同じ優先度を持つタスクの中で最高の優先順位を持ったタスクが実行されているとは限らないため、この方法で自タスクの実行順位が同じ優先度を持つタスクの中で最低となるとは限らない。

tk_rot_rdq の実行例を[図 15(a)],[図 15(b)]に示す。[図 15(a)]の状態で、tskpri＝2 をパラメータとしてこのシステムコールが呼ばれると、新しい優先順位は[図 15(b)]のようになり、次に実行されるのはタスク C となる。



[図 15(a)] tk_rot_rdq 実行前の優先順位



※ 次に実行されるのはタスク C である。

[図 15(b)] tk_rot_rdq(tskpri=2)実行後の優先順位

実行状態タスクのタスク ID 参照 tk_get_tid

tk_get_tid:Get Task Identifier

【C 言語インタフェース】

```
ID tskid = tk_get_tid ( void ) ;
```

【パラメータ】

なし

【リターンパラメータ】

ID	tskid	TaskIdentifier	実行状態タスクの ID

【エラーコード】

なし

【解説】
現在実行状態にあるタスクの ID 番号を得る。タスク独立部の実行中を除けば、現在実行状態にあるタスクは自タスクである。
現在実行状態のタスクがない場合は、0 が返される。

【補足事項】
tk_get_tid で返されるタスク ID は、tk_ref_sys で返される runtskid と同一である。

ディスパッチ禁止 tk_dis_dsp

tk_dis_dsp:Disable Dispatch

【C 言語インタフェース】

```
ER ercd = tk_dis_dsp ( void ) ;
```

【パラメータ】

なし

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_CTX	コンテキストエラー(タスク独立部から発行)

【解説】

タスクのディスパッチを禁止する。これ以後 tk_ena_dsp が実行されるまでの間はディスパッチ禁止状態となり、自タスクが実行状態(RUNNING)から実行可能状態(READY)に移ることはなくなる。また、待ち状態に移ることもできなくなる。ただし、外部割込みは禁止しないので、ディスパッチ禁止状態であっても割込みハンドラは起動される。ディスパッチ禁止状態においては、実行中のタスクが割込みハンドラによってプリエンプト(CPU の実行権の横取り)される可能性はあるが、他のタスクによってプリエンプトされる可能性はない。

ディスパッチ禁止状態の間は、具体的には次のような動作をする。

- 割込みハンドラあるいは tk_dis_dsp を実行したタスクから発行されたシステムコールによって、tk_dis_dsp を実行したタスクより高い優先度を持つタスクが実行可能状態(READY)となっても、そのタスクにはディスパッチされない。優先度の高いタスクへのディスパッチは、ディスパッチ禁止状態が終了するまで遅延される。
- tk_dis_dsp を実行したタスクが、自タスクを待ち状態に移す可能性のあるシステムコール(tk_slp_tsk、tk_wai_sem など)を発行した場合には、E_CTX のエラーとなる。
- tk_ref_sys によってシステム状態を参照した場合、sysstat として TSS_DDSP が返る。

既にディスパッチ禁止状態にあるタスクが tk_dis_dsp を発行した場合は、ディスパッチ禁止状態がそのまま継続するだけで、エラーとはならない。ただし、tk_dis_dsp を何回か発行しても、その後 tk_ena_dsp を 1 回発行するだけでディスパッチ禁止状態が解除される。したがって、tk_dis_dsp～tk_ena_dsp の対がネストした場合の動作は、必要に応じてユーザ側で管理しなければならない。

【補足事項】

ディスパッチ禁止状態では、実行状態のタスクが休止状態(DORMANT)や未登録状態(NON-EXISTENT)に移行することはできない。割込みおよびディスパッチ禁止状態で、実行状態のタスクが tk_ext_tsk または tk_exd_tsk を発行した場合には、E_CTX のエラーを検出する。ただし、tk_ext_tsk や tk_exd_tsk は元のコンテキストに戻らないシステムコールなので、これらのシステムコールのリターンパラメータとしてエラーを通知することはできない。

ディスパッチ許可 tk_ena_dsp

tk_ena_dsp:Enable Dispatch

【C 言語インタフェース】

```
ER ercd = tk_ena_dsp ( void ) ;
```

【パラメータ】

なし

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_CTX	コンテキストエラー(タスク独立部から発行)

【解説】

タスクのディスパッチを許可する。すなわち、tk_dis_dsp によって設定されていたディスパッチ禁止状態を解除する。

ディスパッチ禁止状態ではないタスクが tk_ena_dsp を発行した場合は、ディスパッチを許可した状態がそのまま継続するだけで、エラーとはならない。

システム状態参照 tk_ref_sys

tk_ref_sys:Refer System Status

【C 言語インタフェース】

```
ER ercd = tk_ref_sys ( T_RSYS *pk_rsys ) ;
```

【パラメータ】

T_RSYS*	pk_rsys	Packet to Refer System	システム状態を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_rsys の内容

INT	sysstat	SystemState	システム状態
ID	runtskid	RunningTaskID	現在実行状態にあるタスクの ID
ID	schedtskid	ScheduledTaskID	実行状態にすべきタスクの ID

――(以下に実装独自に他の情報を追加してもよい)――

【エラーコード】

E_OK	正常終了
E_PAR	パラメータエラー(pk_rsys が不正)

【解説】

実行状態を参照し、ディスパッチ禁止中、タスク独立部実行中などといった情報をリターンパラメータとして返す。

sysstat は次のような値をとる。

```
sysstat :=  ( TSS_TSK  | [TSS_DDSP] | [TSS_DINT] )
            || ( TSS_QTSK | [TSS_DDSP] | [TSS_DINT] )
            || ( TSS_INDP )
```

TSS_TSK	0	タスク部実行中
TSS_DDSP	1	ディスパッチ禁止中
TSS_DINT	2	割込み禁止中
TSS_INDP	4	タスク独立部実行中
TSS_QTSK	8	準タスク部実行中

runtskid には現在実行中のタスクの ID、schedtskid には実行状態にすべきタスクの ID が返される。通常は runtskid=schedtskid となるが、ディスパッチ禁止中により優先度の高いタスクが起床された場合などに runtskid ≠ schedtskid となる場合がある。なお、該当タスクがない場合は 0 を返す。

割込みハンドラやタイムイベントハンドラからも発行できなければならない。

【補足事項】

tk_ref_sys で返される情報は、OS の実装によっては、必ずしも常に正しい情報を返すとは限らない。

省電力モード設定 tk_set_pow

tk_set_pow:Set Power Mode

【C 言語インタフェース】

```
ER ercd = tk_set_pow ( UINT powmode ) ;
```

【パラメータ】

UINT	powmode	PowerMode	省電力モード

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_PAR	パラメータエラー(powmode が使用できない値)
E_QOVR	低消費電力モード切替禁止カウントのオーバーフロー
E_OBJ	低消費電力モード切替禁止カウントが 0 の状態で更に TPW_ENALOWPOW を要求した

【解説】
省電力機能は、次の 2 つの機能からなる。

● アイドル(無負荷)中の消費電力低減
実行すべきタスクがない状態のとき、ハードウェアに用意されている低消費電力モードに切り替える。
低消費電力モードは、タイマー割込みから次のタイマー割込みの間のようにごく短い時間の電力消費を抑えるための機能で、CPU のクロック周波数を下げることなどにより行われる。したがって、ソフトウェアによる複雑なモード切替を必要とせず、主にハードウェアの機能によって実現される。

● 自動電源オフ
オペレータが何も操作しない状態が一定時間以上続いた場合、自動的に電源を切りサスペンド状態に移行する。周辺機器からの起動要求(割込みなど)またはオペレータが電源を入れることによりリジュームされ、電源が切れたときの状態に復帰する。
また、バッテリー切れなどの電源異常によっても、自動的に電源を切りサスペンド状態に移行する。
サスペンド中は、周辺機器や周辺回路および CPU の電源が切れる。しかし、メインメモリの内容は保持される。

tk_set_pow は、省電力モードの設定を行う。

```
powmode:= ( TPW_DOSUSPEND || TPW_DISLOWPOW || TPW_ENALOWPOW )

#define TPW_DOSUSPEND   1   サスペンド状態へ移行
#define TPW_DISLOWPOW   2   低消費電力モード切替禁止
#define TPW_ENALOWPOW   3   低消費電力モード切替許可(デフォルト)
```

・TPW_DOSUSPEND
すべてのタスク及びハンドラの実行を停止し、周辺回路(タイマーや割込みコントローラ)を停止し、電源を切る(サスペンドする)。(off_pow を呼び出す)
電源オンされたら、周辺回路の再起動をし、すべてのタスク及びハンドラの実行を再開して、電源を切る前の状態に復帰(リジューム)し、リターンする。
何らかの理由によりリジュームに失敗したときには、通常(リセット時)のスタートアップ処理を行い、新たにシステムを立ち上げなおす。

・TPW_DISLOWPOW
　ディスパッチャ内で行われる低消費電力モードへの切替を禁止する。(low_pow を呼び出さない)

・TPW_ENALOWPOW
　ディスパッチャ内で行われる低消費電力モードへの切替を許可する。(low_pow を呼び出す)

起動時のデフォルトは切替許可(TPW_ENALOWPOW)となる。
TPW_DISLOWPOW が指定された場合、その要求回数がカウントされる。TPW_DISLOWPOW が要求された回数と同じだけ TPW_ENALOWPOW が要求されないと、低消費電力モードは許可されない。要求カウント数の最大値は実装定義だが、少なくとも 255 回以上カウントできなければならない。

【補足事項】
off_pow, low_pow は T-Kernel/SM の機能である。詳細は 5.6 節を参照のこと。
T-Kernel では、電源異常などのサスペンド移行要因の検出は行わない。また、実際にサスペンドするためには、各周辺機器(デバイスドライバ)のサスペンド処理も必要である。サスペンドするには tk_set_pow を直接呼び出すのではなく、T-Kernel/SM のサスペンド機能を利用する。

バージョン参照 tk_ref_ver

tk_ref_ver:Refer Version Information

【C 言語インタフェース】

```
ER ercd = tk_ref_ver ( T_RVER *pk_rver ) ;
```

【パラメータ】

T_RVER*	pk_rver	Packet to Refer Version	バージョン情報を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_rver の内容

UH	maker	MakerCode	カーネルのメーカコード
UH	prid	ProductID	カーネルの識別番号
UH	spver	SpecificationVersion	仕様書バージョン番号
UH	prver	ProductVersion	カーネルのバージョン番号
UH	prno[4]	ProductNumber	カーネル製品の管理情報

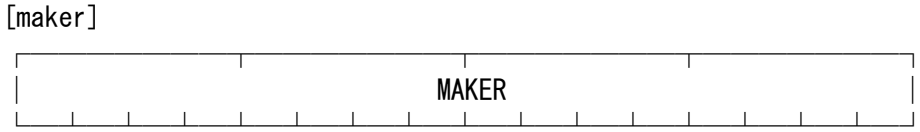
【エラーコード】

E_OK	正常終了
E_PAR	パラメータエラー(pk_rver が不正)

【解説】

使用しているカーネルのバージョン情報を参照し、pk_rver で指定されるパケットに返す。具体的には、次の情報を参照することができる。

maker は、このカーネルを実装した T-Kernel 提供者のコードである。maker のフォーマットを[図 16(a)]に示す。



[図 16(a)] maker のフォーマット

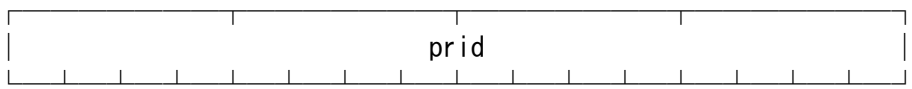
prid は、カーネルの種類を区別するための番号である。prid のフォーマットを[図 16(b)]に示す。

prid の具体的な値の割付けは、カーネルを実装した T-Kernel 提供者に任される。ただし、製品の区別はあくまでもこの番号のみで行うので、各 T-Kernel 提供者において番号の付け方を十分に検討した上、体系づけて使用するようにしなければならない。したがって、maker と prid の組でカーネルの種類を一意に識別することができる。

T-Kernel のオリジナルは、T-Engine フォーラムから提供され、その maker と prid は次のようになる。

```
maker      = 0x0000
prid       = 0x0000
```

[prid]



[図 16(b)] prid のフォーマット

spver では、上位 4 ビットで OS 仕様の種類と、下位 12 ビットでカーネルが準拠する仕様のバージョン番号をあらわす。spver のフォーマットを[図 16(c)]に示す。

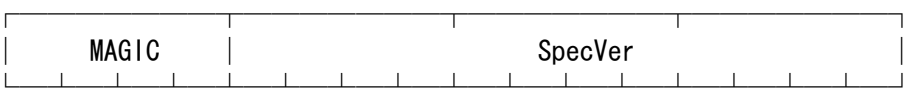
たとえば T-Kernel の Ver 1.02.xx の仕様書に対応する spver は次のようになる。

```
MAGIC      = 0x7      (T-Kernel)
SpecVer    = 0x102    (Ver 1.02)
spver      = 0x7102
```

また、T-Kernel 仕様書のドラフト版である Ver 1.B0.xx の仕様書に対応する spver は次のようになる。

```
MAGIC      = 0x7      (T-Kernel)
SpecVer    = 0x1B0    (Ver 1.B0)
spver      = 0x71B0
```

[spver]



MAGIC: OS のシリーズを区別する番号
- 0x0 TRON 共通 (TAD 等)
- 0x1 reserved
- 0x2 reserved
- 0x3 reserved
- 0x4 AMP T-Kernel
- 0x5 SMP T-Kernel
- 0x6 μT-Kernel
- 0x7 T-Kernel

SpecVer: この製品のもとになった仕様書のバージョン番号。3 桁のパックト形式 BCD コードで入れる。ドラフトの仕様書の場合は、上から 2 桁目が A,B,C となる場合もある。この場合は、対応する 16 進数の A,B,C を入れる。

[図 16(c)] spver のフォーマット

prver は、カーネル実装上のバージョン番号をあらわす。prver の具体的な値の割付けは、カーネルを実装した T-Kernel 提供者に任される。

prno は、カーネル製品の管理情報や製品番号などを入れるために使用するためのリターンパラメータである。

prno の具体的な値の意味は、カーネルを実装した T-Kernel 提供者に任される。

【補足事項】

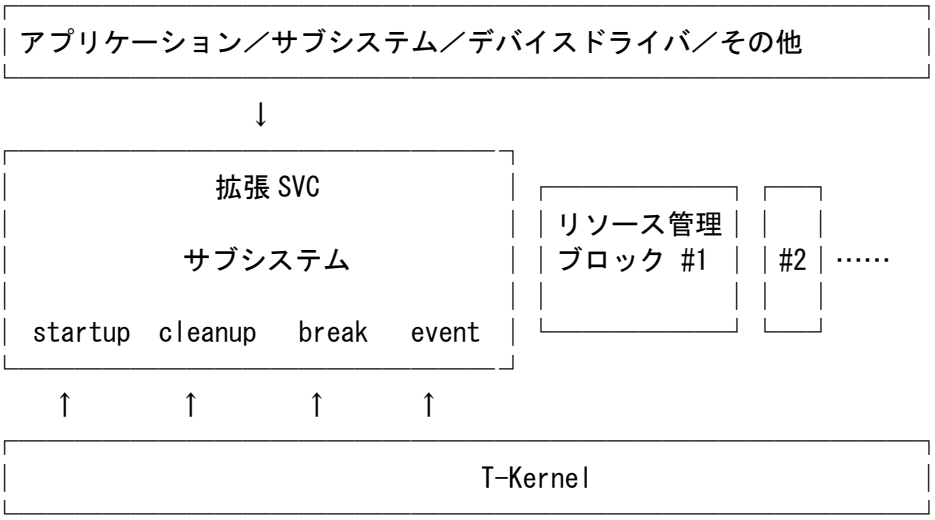
バージョン情報を得るためのパケットの形式や構造体の各メンバのフォーマットは、各種 TRON 仕様の間でほぼ共通になっているが、CPU 情報とバリエーション記述子が省略されている。

tk_ref_ver の SpecVer として得られるのは仕様書のバージョン番号の上位 3 桁であるが、これより下位の桁はミスプリントの修正などといった表記上の変更を表す桁なので、tk_ref_ver で得られる情報には含めていない。仕様書の内容との対応という意味では、仕様書のバージョン番号の上位 3 桁を知ることで必要十分である。

ドラフト版の仕様書を対象として実装された OS では、SpecVer の 2 桁目が A,B,C となることがある。この場合、仕様書のリリース順序と SpecVer の大小関係が必ずしも一致しないので、注意が必要である。仕様書のリリース順は、たとえば Ver 1.A1 → Ver 1.A2 → Ver 1.B1 → Ver 1.C1 → Ver 1.00 → Ver 1.01 → ...となるが、SpecVer の大小関係はドラフト版から正式版に移る部分(Ver 1.Cx → Ver 1.00 の部分)で逆転する。

4.10 サブシステム管理機能

サブシステムは、アプリケーションなどからの要求を受け付けるための拡張 SVC ハンドラ、OS からの要求を受け付けるためのブレーク関数/スタートアップ関数/クリーンアップ関数/イベント処理関数、およびリソース管理ブロックから構成される[図 17]。



[図 17] サブシステム概要

サブシステム定義 tk_def_ssy

tk_def_ssy:Define Sub-System

【C 言語インタフェース】

```
ER ercd = tk_def_ssy ( ID ssid, T_DSSY *pk_dssy ) ;
```

【パラメータ】

ID	ssid	SubsystemID	サブシステム ID
T_DSSY*	pk_dssy	Packet to Define Subsystem	サブシステム定義情報

pk_dssy の内容

ATR	ssyatr	SubsystemAttributes	サブシステム属性
PRI	ssypri	SubsystemPriority	サブシステム優先度
FP	svchdr	Extended SVC handler address	拡張 SVC ハンドラアドレス
FP	breakfn	BreakFunctionAddress	ブレーク関数アドレス
FP	startupfn	StartupFunctionAddress	スタートアップ関数アドレス
FP	cleanupfn	CleanupFunctionAddress	クリーンアップ関数アドレス
FP	eventfn	EventHandlingFunctionAddress	イベント処理関数アドレス
INT	resblksz	ResourceControlBlockSize	リソース管理ブロックサイズ(バイト数)

――(以下に実装独自に他の情報を追加してもよい)――

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(ssid が不正あるいは利用できない)
E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_RSATR	予約属性(ssyatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_dssy が不正あるいは利用できない)
E_OBJ	ssid はすでに定義済みである。(pk_dssy≠NULL の時)
E_NOEXS	ssid は定義されていない。(pk_dssy＝NULL の時)

【解説】

ssid のサブシステムを定義する。

1つのサブシステムに1つのサブシステム ID を､他のサブシステムと重複しないように割当てなければならない。OS に自動割当ての機能はない。

サブシステム ID は 1～9 が T-Kernel 用に予約されている｡10～255 がミドルウェア等で使用できる番号となる。ただし、使用可能なサブシステム ID の最大値は実装定義であり、255 より小さい場合がある。

ssyatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。ssyatr のシステム属性には、現在のバージョンでは割当てがなく、システム属性は使われていない。

ssypri は、サブシステムの優先度で、スタートアップ関数、クリーンアップ関数、イベント処理関数が優先度順に呼び出される。同一優先度の場合の呼出順は不定である。サブシステム優先度は、1 が最も優先度が高く、数値が大きくなるにしたがって優先度が下がる。指定できる優先度の範囲は実装定義だが、少なくとも 1～16 が指定できなければならない。

breakfn, startupfn, cleanupfn, eventfn はそれぞれ NULL を指定することができる。NULL を指定した場合、対応する関数は呼び出されない。

pk_dssy=NULL とした場合は、サブシステムの定義を抹消する。このとき、ssid のサブシステムのリソース管理ブロックもすべて削除される。

・リソース管理ブロック

リソース(資源)をグループ分けし、その所属などを管理するためのメモリブロックである。resblksz で指定したサイズのメモリが、リソースグループごとに 1 つずつ用意される。resblksz=0 を指定すれば、リソース管理ブロックは割当てられない。しかし、その場合もリソース ID(tk_cre_res 参照)は割当てられる。

タスクは、いずれかのリソースグループに所属する。あるタスクからサブシステムに対して要求があり、サブシステム内のリソースがそのタスクに割当てられた場合に、その割当て情報をリソース管理ブロックに登録する。どのようなリソースをリソース管理ブロックに登録するか、またどのようにして登録するかは、サブシステム側で決める。

リソース管理ブロックの内容については OS は関知しないので、サブシステム側で自由に利用できる。ただし、resblksz はできる限り小さなサイズとする。より大きなメモリブロックが必要であれば、メモリブロックは別途サブシステム側で用意し、そのアドレスをリソース管理ブロックに登録するようにする。

リソース管理ブロックは、システム空間の常駐メモリである。

・拡張 SVC ハンドラ

アプリケーションなどからの要求の受け付け口となる。サブシステムの API となる部分である。システムコールと同様の方法で呼び出すことが可能で、一般的にはトラップ命令などで呼び出す。

拡張 SVC ハンドラは次の形式となる。

```
INT svchdr( VP pk_para, FN fncd )
{
     /*
             fncd により分岐して処理
      */
    return retcode;  /* 拡張 SVC ハンドラの終了 */
}
```

fncd は機能コードである。機能コードの下位 8 ビットにはサブシステム ID が入る。残りの上位ビットは、サブシステム側で任意に使用できる。通常は、サブシステム内の機能コードとして使用する。ただし、機能コードは正の値でなければならないため、最上位ビットは必ず 0 となる。

pk_para は呼出側から渡されたパラメータをパケット形式にしたものである。パケットの形式は、サブシステム側で任意に決めることができる。一般的には、C 言語で関数に引数を渡すときのスタックの形式と同じになる。これは、多くの場合 C 言語の構造体の形式と同じである。

拡張 SVC ハンドラからの戻値は、そのまま呼出元へ関数の戻値として戻される。原則として、負の値をエラー値、0 または正の値を正常時の戻値とする。なお、何らかの理由で拡張 SVC の呼出に失敗した場合は、拡張 SVC ハンドラは呼び出されずに、呼出元には OS のエラーコード(負の値)が返されるため、それと混同しないようにしておくことが望ましい。

拡張 SVC の呼出側の形式は OS の実装に依存する。ただし、サブシステムの API としては C 言語の関数の形式で、OS の実装に依存しないように規定しなければならない。サブシステムは、C 言語の関数の形式から OS 依存の拡張 SVC の呼出形式に変換するための、インタフェースライブラリを用意しなければならない。

拡張 SVC ハンドラは、準タスク部として実行される。

タスク独立部からも呼出が可能で、タスク独立部から呼び出された場合は、拡張 SVC ハンドラもタスク独立部として実行される。

・ブレーク関数

ブレーク関数は、拡張 SVC ハンドラの実行中のタスクに、タスク例外が発生した場合に呼び出される関数である。
ブレーク関数が呼び出されたら、タスク例外が発生した現在実行中の拡張 SVC ハンドラの処理を速やかに中止し、拡張 SVC ハンドラから呼出元に戻らなければならない。現在実行中の拡張 SVC ハンドラの処理を中止するための処理をブレーク関数で行う。
ブレーク関数は次の形式となる。

```
void breakfn( ID tskid )
{
      /*
            拡張 SVC ハンドラの実行中止処理
       */
}
```

tskid はタスク例外が発生したタスクの ID である。
ブレーク関数は、tk_ras_tex によりタスク例外が発生されたときに呼び出される。また、拡張 SVC ハンドラがネストして呼び出されていた場合は、拡張 SVC ハンドラから戻ってネストが１段浅くなったときに、戻った先の拡張 SVC ハンドラに対応するブレーク関数が呼び出される。
ブレーク関数は、１回のタスク例外で、１つの拡張 SVC ハンドラに対して１回のみ実行される。
タスク例外が発生している状態で、さらにネストして拡張 SVC を呼出した場合、呼出先の拡張 SVC ハンドラに対してはブレーク関数は呼び出されない。
ブレーク関数は準タスク部として実行されるが、そのタスクコンテキストは、tk_ras_tex を発行したタスク(tk_ras_tex が呼び出されたときのブレーク関数の実行)またはタスク例外の発生したタスク(拡張 SVC ハンドラを実行しているタスク)(拡張 SVC のネストが１段浅くなったときのブレーク関数の実行)のいずれかとなる。したがって、ブレーク関数の実行タスクと拡張 SVC ハンドラの実行タスクが異なっている場合がある。このような場合、ブレーク関数と拡張 SVC ハンドラがタスクスケジューリングにしたがって同時に実行されることになる。
そのため、ブレーク関数の実行が終了する前に拡張 SVC ハンドラから呼出元に戻るケースが考えられるが、そのような場合には拡張 SVC ハンドラから呼出元に戻る直前の状態でブレーク関数が終了するまで待たされる。この待ち状態がタスク状態遷移のどの状態になるかは実装依存とするが、READY 状態のまま(RUNNING 状態になれない READY 状態)とすることが望ましい。また、ブレーク関数の終了を待つ間、タスクの優先順位が変化することがあるが、タスクの優先順位がどのようになるかは実装依存とする。
同様に、ブレーク関数の実行が終了するまで拡張 SVC ハンドラから拡張 SVC を呼び出すことはできず、ブレーク関数の終了まで待たされる。
つまり、タスク例外が発生してからブレーク関数が終了するまでの間、タスク例外が発生したときの拡張 SVC ハンドラから抜けないようにしなくてはならない。
ブレーク関数と拡張 SVC ハンドラが異なるタスクで実行される場合、ブレーク関数のタスク優先度が拡張 SVC ハンドラのタスク優先度より低い場合は、ブレークハンドラの実行期間だけ、拡張 SVC ハンドラのタスク優先度と同じ優先度までブレーク関数のタスク優先度が引き上げられる。逆に、ブレーク関数のタスク優先度が同じかより高い場合には、優先度は変更されない。変更される優先度は現在優先度で、ベース優先度は変更されない。
優先度変更はブレーク関数へ入る直前のみで、その後に拡張 SVC ハンドラ側の優先度が変更されても追従してブレーク関数側の優先度が変更されることはない。ブレーク関数実行中にブレーク関数側の優先度を変更した場合も、拡張 SVC ハンドラ側の優先度が変更されることはない。また、このときブレーク関数を実行中であることにより優先度の変更が制限されることはない。
ブレーク関数終了時には現在優先度をベース優先度に戻す。ただし、ミューテックスをロックしていた場合には、ミューテックスにより調整された優先度に戻す。(つまり、ブレーク関数の入口と出口で現在優先度を調整する機能があるのみで、それ以外については通常のタスク実行中と同じである。)

・スタートアップ関数

tk_sta_ssy の呼出によって呼び出される。
リソース管理ブロックの初期化処理を行う。
スタートアップ関数は次の形式となる。

```
void startupfn( ID resid, INT info )
{
	/*
		リソース管理ブロックの初期化処理
	 */
}
```

resid は初期化対象のリソースグループ ID、info は任意のパラメータで、いずれも tk_sta_ssy で指定されたものである。
何らかの理由でリソース管理ブロックの初期化に失敗した場合も、スタートアップ関数は正常に終了させなければならない。リソース管理ブロックの初期化ができなかった場合、それによって正常に実行できない API(拡張 SVC)が呼び出された時点で、その API の戻値をエラーとする。
スタートアップ関数は tk_sta_ssy を呼出したタスクのコンテキストで、準タスク部として実行される。

・クリーンアップ関数

tk_cln_ssy の呼出によって呼び出される。
リソースの解放処理を行う。
クリーンアップ関数は次の形式となる。

```
void cleanupfn( ID resid, INT info )
{
	/*
		リソース解放処理
	 */
}
```

resid はリソース解放の対象となるリソースグループ ID、info は任意のパラメータで、いずれも tk_cln_ssy で指定されたものである。
何らかの理由でリソースの解放に失敗した場合でも、クリーンアップ関数は正常に終了させなければならない。エラーログを残すなどのエラー処理は、サブシステムで独自に決める。
クリーンアップ関数が終了した後、自動的にリソース管理ブロックは 0 クリアされる。また、クリーンアップ関数が定義されなかった場合(cleanupfn=NULL)にも、tk_cln_ssy によりリソース管理ブロックは 0 クリアされる。
クリーンアップ関数は、tk_cln_ssy を呼出したタスクのコンテキストで、準タスク部として実行される。

・イベント処理関数

tk_evt_ssy の呼出によって呼び出される。
サブシステムに対する各種要求を処理する。
なお、すべての要求がすべてのサブシステムで処理されなければならないという性質のものではない。処理する必要がなければ何もせず単に E_OK を返せばよい。
イベント処理関数は次の形式となる。

```
ER eventfn( INT evttyp, ID resid, INT info )
{
	/*
		各イベントに対応する処理
	 */
	return ercd;
}
```

evttyp は要求の種別、resid は対象となるリソースグループの ID、info は任意のパラメータで、いずれも tk_evt_ssy で指定されたものである。特定のリソースグループを対象としない場合は、resid=0 となる。
正常に処理した場合は戻値に E_OK を返す。異常があった場合はエラーコード(負の値)を返す。
evttyp には次のものがある。詳細は 5.3 節を参照のこと。

```
#define TSEVT_SUSPEND_BEGIN  1    /* デバイスサスペンド開始前 */
#define TSEVT_SUSPEND_DONE   2    /* デバイスサスペンド完了後 */
#define TSEVT_RESUME_BEGIN   3    /* デバイスリジューム開始前 */
#define TSEVT_RESUME_DONE    4    /* デバイスリジューム完了後 */
#define TSEVT_DEVICE_REGIST  5    /* デバイス登録通知 */
#define TSEVT_DEVICE_DELETE  6    /* デバイス抹消通知 */
```

イベント処理関数は、tk_evt_ssy を呼出したタスクのコンテキストで、準タスク部として実行される。

【補足事項】
拡張SVCハンドラおよびブレーク関数/スタートアップ関数/クリーンアップ関数/イベント関数はTA_HLNG属性相当のみで、高級言語対応ルーチンを経由して呼び出される。TA_ASM 属性を指定する機能はない。
スタートアップ関数によりリソース管理ブロックが初期化される以前、およびクリーンアップ関数によりリソースが解放されたあとに、そのリソースグループに属するタスクから拡張 SVC が呼び出された場合の動作は、サブシステムの実装に依存する。OS ではこのような呼出を特に禁止しない。基本的には、スタートアップ関数の呼出前およびクリーンアップ関数の呼出後に、拡張 SVC を呼び出さないようにする必要がある。
何らかの理由で、スタートアップ関数が呼び出されないまま、ブレーク関数/クリーンアップ関数/イベント処理関数が呼び出される場合がある。このような場合も、誤動作せずに実行されなければならない。リソース管理ブロックは、リソース管理ブロックが最初に生成された時および tk_cln_ssy によるクリーンアップ処理時に 0 クリアされる。したがって、スタートアップ関数により正常に初期化されていなければ、リソース管理ブロックは 0 クリアされているはずである。

スタートアップ/クリーンアップ関数呼出

tk_sta_ssy
tk_cln_ssy

tk_sta_ssy:Call StartUp Function of Sub-System
tk_cln_ssy:Call CleanUp Function of Sub-System

【C 言語インタフェース】

```
ER ercd = tk_sta_ssy ( ID ssid, ID resid, INT info ) ;
ER ercd = tk_cln_ssy ( ID ssid, ID resid, INT info ) ;
```

【パラメータ】

ID	ssid	SubsystemID	サブシステム ID
ID	resid	ResourceID	リソース ID
INT	info	Information	任意パラメータ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(ssid, resid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(ssid のサブシステムが定義されていない)
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

【解説】

ssid のサブシステムのスタートアップ関数/クリーンアップ関数を呼び出す。

ssid=0 を指定したときは、現在定義されているすべてのサブシステムのスタートアップ関数/クリーンアップ関数を呼び出す。この場合、次の順序で各サブシステムのスタートアップ関数/クリーンアップ関数を呼び出す。

tk_sta_ssy ：サブシステム優先度の高いものから順に呼び出す。
tk_cln_ssy ：サブシステム優先度の低いものから順に呼び出す。

同一優先度の場合の呼出順は不定である。

したがって、複数のサブシステム間に依存関係がある場合、サブシステム優先度をその依存関係にしたがって設定する必要がある。例えば、サブシステム B がサブシステム A の機能を利用しているような場合、サブシステム A の優先度をサブシステム B より高くしなければならない。

スタートアップ関数またはクリーンアップ関数が定義されていないサブシステムに対して実行しても、単にスタートアップ関数/クリーンアップ関数を呼び出さないだけでエラーとはならない。

スタートアップ関数/クリーンアップ関数を実行中に、tk_sta_ssy, tk_cln_ssy を呼出したタスクにタスク例外が発生した場合、タスク例外はスタートアップ関数/クリーンアップ関数が終了するまで保留される。

イベント処理関数呼出 tk_evt_ssy

tk_evt_ssy:Call Event Function of Sub-System

【C 言語インタフェース】

```
ER ercd = tk_evt_ssy ( ID ssid, INT evttyp, ID resid, INT info ) ;
```

【パラメータ】

ID	ssid	SubsystemID	サブシステム ID
INT	evttyp	EventType	イベント要求種別
ID	resid	ResourceID	リソース ID
INT	info	Information	任意パラメータ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(ssid, resid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(ssid のサブシステムが定義されていない)
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)
その他	イベント処理関数の返したエラー値

【解説】

ssid のサブシステムのイベント処理関数を呼び出す。

ssid=0 を指定したときは、現在定義されているすべてのサブシステムのイベント処理関数を呼び出す。この場合、次の順序で各サブシステムのイベント処理関数を呼び出す。

evttyp が奇数のとき：サブシステム優先度の高い方から順に呼び出す。
evttyp が偶数のとき：サブシステム優先度の低い方から順に呼び出す。

同一優先度の場合の呼出順は不定である。

イベント処理関数が定義されていないサブシステムに対して実行しても、単にイベント処理関数を呼び出さないだけでエラーとはならない。

特定のリソースに対する処理要求でない場合は、resid=0 とする。

イベント処理関数がエラーを返した場合、システムコールの戻値としてそのエラー値をそのまま返す。ssid=0 の場合、イベント処理関数がエラーを返した場合も、すべてのサブシステムのイベント処理関数が呼び出される。システムコールの戻値には、エラーを返したイベント処理関数の内の１つのエラー値のみ返す。どのサブシステムのイベント処理関数が返したエラーかはわからない。

イベント処理関数を実行中に、tk_evt_ssy を呼出したタスクにタスク例外が発生した場合、タスク例外はイベント処理関数が終了するまで保留される。

サブシステム定義情報の参照 tk_ref_ssy

tk_ref_ssy:Refer Sub-System Status

【C 言語インタフェース】

```
ER ercd = tk_ref_ssy ( ID ssid, T_RSSY *pk_rssy ) ;
```

【パラメータ】

ID	ssid	SubsystemID	サブシステム ID
T_RSSY*	pk_rssy	Packet to Refer Subsystem	サブシステム定義情報を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_rssy の内容

PRI	ssypri	サブシステム優先度
INT	resblksz	リソース管理ブロックサイズ(バイト数)

――(以下に実装独自に他の情報を追加してもよい)――

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(ssid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(ssid のサブシステムが定義されていない)
E_PAR	パラメータエラー(pk_rssy が不正)

【解説】

ssid で示された対象サブシステムの各種の情報を参照する。
resblksz には、tk_def_ssy で指定したリソース管理ブロックサイズが返される。
ssid のサブシステムが定義されていない場合は、E_NOEXS となる。

リソースグループの生成 tk_cre_res

tk_cre_res:Create Resource Group

【C 言語インタフェース】

```
ID resid = tk_cre_res ( void ) ;
```

【パラメータ】

なし

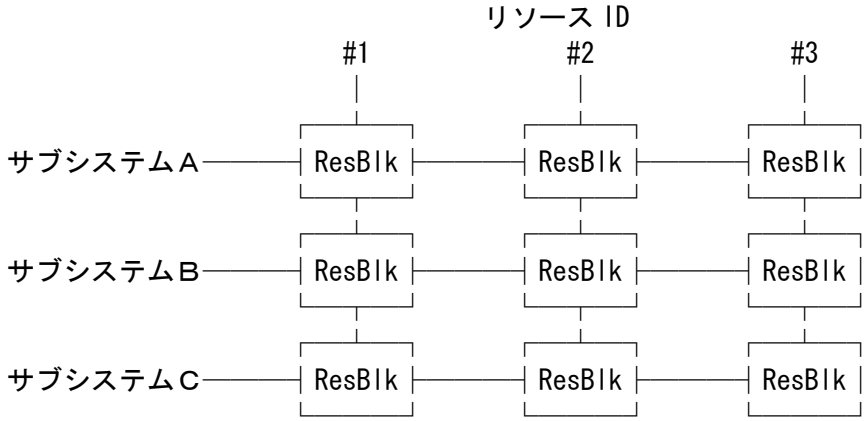
【リターンパラメータ】

ID	resid	ResourceID	リソース ID
	または	ErrorCode	エラーコード

【エラーコード】

E_LIMIT	リソースグループの数がシステムの制限を超えた
E_NOMEM	メモリ不足(管理ブロックの領域が確保できない)

【解説】
新規のリソースグループを生成し、リソース管理ブロックとリソース ID を割当てる。
リソース ID は全サブシステムに対して共通で割当てられ、リソース管理ブロックが各サブシステムごとに生成される[図 18]。



[図 18] サブシステムとリソースグループの関係

すでにリソースグループが生成されている状態で、新たにサブシステムが定義される場合がある。この場合も、新たに登録されたサブシステムに対して、すでに存在するリソースグループのリソース管理ブロックが生成されなければならない。つまり、リソース管理ブロックの生成は tk_def_ssy でも行わなければならないケースがありうる。
例えば、[図 18]のような状態のとき新たにサブシステム D が定義された場合、自動的にサブシステム D のためのリソース ID#1,#2,#3 のリソース管理ブロックが生成されなければならない。

【補足事項】

リソース ID は論理空間 ID(lsid)としても使用される場合がある。そのため、リソース ID は論理空間 ID として直接利用可能な値、もしくは簡単な変換で論理空間 ID として利用可能な値とすることが望ましい。

特殊なリソースグループとして、システムリソースグループが必ず存在する。システムリソースグループは、tk_cre_res で生成するまでもなくシステム起動時から必ず 1 つ存在しているリソースグループである。システムリソースグループは削除することはできない。必ず存在していることを除けば、システムリソースグループが他のリソースグループと異なる部分はない。

リソース管理ブロックの生成は、次の 2 種類の実装が考えられる。

(A) サブシステム定義時(tk_def_ssy)に最大リソースグループ数分のリソース管理ブロックを生成しておき、tk_cre_res では単にそれを割当てる。
(B) tk_cre_res で全サブシステム分のリソース管理ブロックを生成して、それを割当てる。

最初にリソース管理ブロックを生成した時に、リソース管理ブロックを 0 クリアするという仕様としているため、(A)と(B)ではリソース管理ブロックの 0 クリアのタイミングが異なる。これによる影響は少ないと考えられるが、(A)の方が 0 クリアされるケースが少なくなるため、サブシステムは(A)を前提として実装しなければならない。なお、OS の実装としても(A)を推奨する。

リソースグループの削除 tk_del_res

tk_del_res:Delete Resource Group

【C 言語インタフェース】

```
ER ercd = tk_del_res ( ID resid ) ;
```

【パラメータ】

ID	resid	ResourceID	リソース ID

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(resid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(resid のリソースが存在しない)

【解説】
resid のリソースグループのリソース管理ブロックを削除し、リソース ID を解放する。
全サブシステムのリソース管理ブロックが削除される。

【補足事項】
削除対象のリソースに属しているタスクが残っていても、リソースは削除される。原則として、削除対象のリソースに属しているすべてのタスクを終了・削除した後に、リソースを削除しなければならない。削除対象リソースに属しているタスクが残っている状態で、かつそのタスクがサブシステム(拡張 SVC)を呼出している状態で、リソースを削除した場合の動作は保証されない。また、削除されたリソースに属するタスクが、サブシステム(拡張 SVC)を呼出した場合の動作も保証されない。
リソース管理ブロックが実際に削除されるタイミングは実装に依存する。(tk_cre_res 参照)
システムリソースグループの削除はできない。(E_ID)

リソース管理ブロックの取得 tk_get_res

tk_get_res:Get Resource Management Block

【C 言語インタフェース】

```
ER ercd = tk_get_res ( ID resid, ID ssid, VP *p_resblk ) ;
```

【パラメータ】

ID	resid	ResourceID	リソース ID
ID	ssid	SubsystemID	サブシステム ID
VP*	p_resblk	ResourceControlBlock	リターンパラメータ resblk を返す領域へのポインタ

【リターンパラメータ】

VP	resblk	ResourceControlBlock	リソース管理ブロック
ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(resid, ssid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(resid, ssid のリソースが存在しない)
E_PAR	パラメータエラー(p_resblk が不正)

【解説】
ssid のサブシステムの resid のリソース管理ブロックのアドレスを取得する。

【補足事項】
削除されたリソース IDに対してもE_OKとなる場合がある。エラー(E_NOEXS)が返されるか否かは実装依存である。

第 5 章　T-Kernel/SM の機能

この章では、T-Kernel/SystemManager(T-Kernel/SM)で提供している機能の詳細について説明を行う。

全般的な注意・補足事項

- T-Kernel/SM では原則として、tk_～は拡張 SVC、それ以外はライブラリ関数(インライン関数を含む)、または C 言語のマクロである。
- ライブラリまたはマクロであっても､間接的に拡張SVC やシステムコールを呼び出している場合がある。
- 常に発生する可能性のあるエラーE_PAR, E_MACV, E_NOMEM などは、特に説明を必要とする場合以外は省略している。
- T-Kernel/SM の拡張 SVC およびライブラリは、特に明記されているものを除き、タスク独立部およびディスパッチ禁止中・割込み禁止中状態から呼び出すことはできない(E_CTX)。
- T-Kernel/SM の拡張 SVC およびライブラリは、特に明記されているものを除き、T-Kernel/OS のシステムコールの呼び出し可能な保護レベルより低い保護レベル(TSVCLimit より低い保護レベル)から呼び出すことはできない(E_OACV)。
- T-Kernel/SM の拡張 SVC およびライブラリは、特に明記されているものを除き、再入可能(reentrant)である。ただし、内部で排他制御を行っている場合がある。
- E_PAR, E_MACV, E_CTX の検出は実装依存でありエラーとして検出されない場合もあるため、このようなエラーを発生する可能性のある呼び出しを行ってはいけない。

5.1 システムメモリ管理機能

システムメモリ管理機能では、T-Kernel が動的に割当てるすべてのメモリ(システムメモリ)を管理する。T-Kernel 内部で使用しているメモリやタスクのスタック、メッセージバッファ、メモリプールなどもここから割当てる。

システムメモリは、ブロック単位で管理される。ブロックサイズは、通常 MMU で定義されるページサイズとなる。MMU を使用しないシステムでは任意のサイズでよいが、1KB～4KB 程度を推奨する。ブロックサイズは、tk_ref_smb により取得できる。

システムメモリは、共有空間のメモリである。T-Kernel ではタスク固有空間のメモリの管理は行わない。

システムメモリ管理機能は拡張 SVC により呼び出す。T-Kernel 内部で利用する他、アプリケーションやサブシステム、デバイスドライバなどからも利用可能である。なお、T-Kernel 内部での利用は拡張 SVC を経由しない方法でもよく、実装定義とする。

5.1.1 システムメモリ割当て

```
ER tk_get_smb ( VP *addr, INT nblk, UINT attr )
```

nblk で指定したブロック数分の連続したメモリ領域を attr で指定した属性で割当てる。割当てたメモリの先頭アドレスを addr に返す。

```
attr := (TA_RNG0 || TA_RNG1 || TA_RNG2 || TA_RNG3) | [TA_NORESIDENT]

#define TA_NORESIDENT    0x00000010      /* 非常駐 */
#define TA_RNG0          0x00000000      /* 保護レベル 0 */
#define TA_RNG1          0x00000100      /* 保護レベル 1 */
#define TA_RNG2          0x00000200      /* 保護レベル 2 */
#define TA_RNG3          0x00000300      /* 保護レベル 3 */
```

獲得したメモリは、リソースグループに属さない。メモリを割当てられなかったときは、E_NOMEM を返す。

```
ER tk_rel_smb ( VP addr )
```

addr で指定したメモリを解放する。addr は、tk_get_smb()で得たアドレスでなければならない。

```
ER tk_ref_smb ( T_RSMB *pk_rsmb )
```

システムメモリに関する情報を取得する。

```
typedef struct t_rsmb {
    INT         blksz;          /* ブロックサイズ(バイト数) */
    INT         total;          /* 全ブロック数 */
    INT         free;           /* 残りブロック数 */
    /* 以下に実装独自の情報が追加される場合がある */
} T_RSMB;
```

仮想記憶を行っている場合、全ブロック数、残りブロック数が一意に決定できない場合がある。そのような場合の total, free の内容は実装依存とするが、free ÷ total が残りメモリ容量の割合の参考値となるような値とすることが望ましい。

5.1.2 メモリ割当てライブラリ

システムメモリの割当てはブロック単位となるため、それを細分化して利用するためのライブラリを提供する。

```
void* Vmalloc ( size_t size )

void* Vcalloc ( size_t nmemb, size_t size )

void* Vrealloc ( void *ptr, size_t size )

void  Vfree ( void *ptr )

void* Kmalloc ( size_t size )

void* Kcalloc ( size_t nmemb, size_t size )

void* Krealloc ( void *ptr, size_t size )

void  Kfree ( void *ptr )
```

C 言語標準ライブラリの malloc/calloc/realloc/free と同等の機能である。V～ は非常駐メモリ、K～は常駐メモリを対象とし、いずれも TSVCLimit の保護レベルのメモリとなる。

タスク独立部およびディスパッチ禁止中、割り込み禁止中に呼び出すことはできない。呼び出した場合の動作は不定となる。(システムダウンの可能性もある)

5.2 アドレス空間管理機能

メモリのアクセス権は、アクセス権情報としてタスクごとに保持される。アクセス権情報は、基本的に拡張 SVC を呼び出す直前の保護レベルでアクセスできる権利を示している。例えば、保護レベル 3 で実行中のタスクが拡張 SVC を呼び出した場合、アクセス権情報としては保護レベル 3 でアクセスできる権利を示している。拡張 SVC を実行しているときは保護レベル 0 であるため、拡張 SVC からさらに拡張 SVC をネストして呼び出した場合、ネストして呼び出した拡張 SVC でのアクセス権情報は保護レベル 0 でアクセスできる権利を示すことになる。

メモリのアクセス権情報は次のように設定される。

- タスク起動直後は、そのタスクの生成時に指定された保護レベルのアクセス権が設定される。
- 拡張 SVC を呼び出すと、呼び出す直前に実行していた保護レベルのアクセス権が設定される。
- 拡張 SVC から戻ると、呼び出す直前のアクセス権に戻る。
- SetTaskSpace()を行うと、その時点の対象タスクのアクセス権が自タスクに複写される。

5.2.1 アドレス空間設定

```
ER SetTaskSpace ( ID tskid )
```

tskid で指定したタスクのタスク固有空間およびアクセス権情報が自タスクに設定される。これにより、tskid のタスクと同じアドレス空間とアクセス権を持つことになる。

なお、これは現時点で tskid と同じになるだけであり、tskid のタスクが別のアドレス空間に切り替わったりアクセス権が変わっても、それに追従して自タスクのアドレス空間やアクセス権が切り替わることはない。また、自タスクが拡張 SVC の呼び出し中の場合、拡張 SVC から戻るとアクセス権は拡張 SVC 呼び出し前の状態に戻る。ただし、タスク固有空間は戻らない。

tskid に自タスクのタスク ID を指定することはできない。ただし、TSK_SELF により自タスクを指定した場合は、現在実行中の保護レベルのアクセス権が設定される。このとき、タスク固有空間は切り替わらない。

E_ID	tskid が不正
E_NOEXS	tskid のタスクが存在しない
E_OBJ	TSK_SELF 以外で自タスクを指定した

5.2.2 アドレス空間チェック

現在のアクセス権情報にしたがって、指定されたメモリ領域へのアクセスが許可されているか検査する。アクセスできない(権利がない、またはメモリが存在しない)場合は、E_MACV が返される。

～R　読込みアクセス権があるか検査する。
～RW　読込みおよび書込みアクセス権があるか検査する。
～RE　読込みアクセス権および実行権があるか検査する。

```
ER ChkSpaceR ( VP addr, INT len )
```

```
ER ChkSpaceRW ( VP addr, INT len )
```

```
ER ChkSpaceRE ( VP addr, INT len )
```

addr から len バイトの領域に対するアクセス権があるか検査する。

```
INT ChkSpaceBstrR ( UB *str, INT max )
```

```
INT ChkSpaceBstrRW ( UB *str, INT max )
```

str から文字列の終端('¥0')に達するか max 文字目(バイト数)に達するまでのメモリ領域に対するアクセス権があるか検査する。max=0 の場合は、max は無視して文字列終端まで検査する。
アクセス可能であれば、その文字列の長さ(バイト数)を返す。max 文字目までに文字列の終端があった場合は'¥0'の直前までの長さ、文字列の終端より先に max 文字に達した場合は max を返す。

```
INT ChkSpaceTstrR ( TC *str, INT max )
```

```
INT ChkSpaceTstrRW ( TC *str, INT max )
```

```
typedef     UH    TC;                 /* TRON 文字コード */
#define     TNULL    ((TC)0)          /* TRON コード文字列の終端 */
```

str から TRON コード文字列の終端(TNULL)に達するか max 文字目(TC 数)に達するまでのメモリ領域に対するアクセス権があるか検査する。max=0 の場合は、max は無視して文字列終端まで検査する。
アクセス可能であれば、その文字列の長さ(TC 数)を返す。max 文字目までに文字列の終端があった場合は TNULL の直前までの長さ、文字列の終端より先に max 文字に達した場合は max を返す。
str は、偶数アドレスでなければならない。

5.2.3 アドレス空間ロック

一般に常駐/非常駐はページ単位で行われ、管理もページ単位となる。そのため、ロックとアンロックの領域の一致を OS 側で検査することはしない場合が多い。ロックとアンロックに同じ領域を指定するのは呼び出し側の責任である。

```
ER LockSpace ( VP addr, INT len )
```

addr から len バイトのメモリ領域をロック(常駐化)する。

E_MACV　メモリが存在しない

```
ER UnlockSpace ( VP addr, INT len )
```

addr から len バイトのメモリ領域をアンロック(常駐解除)する。ただし、同じ領域に対して複数回ロックした場合は、同じ回数アンロックしないと解除されない。
なお、ロックした領域の一部だけをアンロックすることはできない(してはいけない)。

5.2.4 物理アドレスの取得

```
INT CnvPhysicalAddr ( VP vaddr, INT len, VP *paddr )
```

vaddr	LogicalAddress	論理アドレス
len	Length	領域サイズ(バイト数)
paddr	PhysicalAddress	物理アドレスを返す
戻値	Size of Physical Address Contiguous Space	物理アドレスの連続領域サイズ(バイト数)
	または ErrorCode	エラーコード

論理アドレス vaddr に対応する物理アドレスを求め、paddr へ返す。また、vaddr から len バイトの領域の内、物理アドレスが連続している分のサイズを戻値に返す。したがって、paddr から戻値に返されたサイズ分までの領域のみ有効である。

なお、物理アドレスを取得する領域は常駐化(ロック)しておかなければならない。

物理アドレスを取得した領域(paddr から戻値に返されたサイズ分の領域)は、DMA 転送を行う前提で、メモリキャッシュがオフになる。常駐を解除(アンロック)することで、キャッシュはオンに戻る。

ただし、ハードウェアの制限から部分的なメモリのキャッシュのオフができない場合には、キャッシュのフラッシュ(ライトバックして無効化)を行うものとする。

E_MACV　メモリが存在しない

5.2.5 メモリのマップ

```
ER MapMemory ( VP paddr, INT len, UINT attr, VP *laddr )
```

物理アドレス paddr から len バイトの領域を論理空間にマップし、その論理アドレスを*laddr に返す。論理アドレスを指定することはできず、自動的に割り当てられる。

paddr=NULL を指定した場合には、物理アドレスの連続した任意のメモリを自動的に割り当て、その領域を論理空間にマップする。

マップされた論理空間は、attr で指定された属性となる。

```
attr := (MM_USER || MM_SYSTEM) | [MM_READ] | [MM_WRITE] | [MM_EXECUTE] | [MM_CDIS]
```

MM_USER	ユーザーレベルアクセス可
MM_SYSTEM	システムレベルアクセス可
MM_READ	読み込みアクセス可
MM_WRITE	書き込みアクセス可
MM_EXECUTE	プログラム実行可
MM_CDIS	キャッシュ禁止

これらのシンボルの値は機種によって異なる。したがって、必ずこれらのシンボルを使用する必要がある。また、機種によっては上記の属性以外の指定がある場合がある。

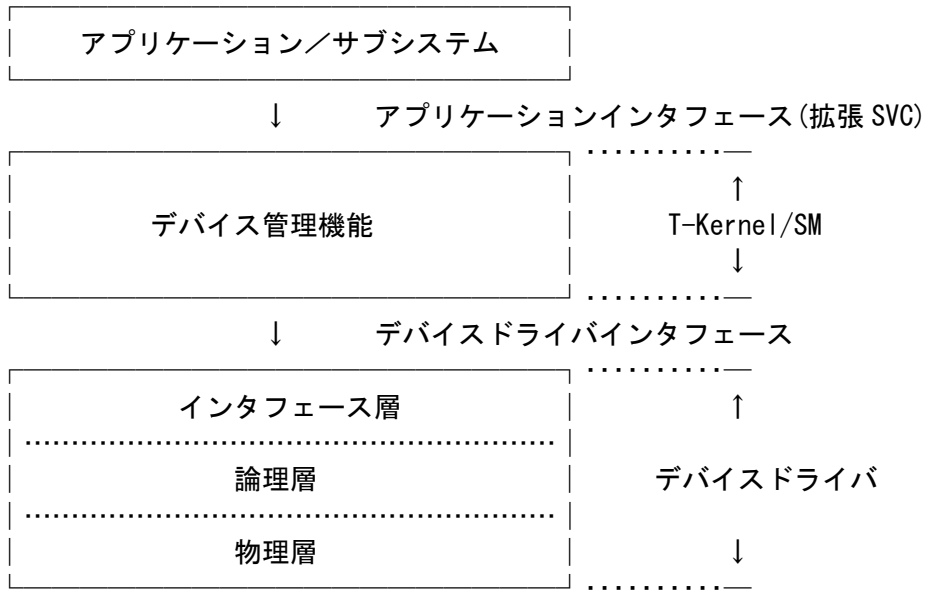
E_LIMIT	マップする論理空間が不足した。
E_NOMEM	メモリが不足した。

```
ER UnmapMemory ( VP laddr )
```

MapMemory()で割り当てられた論理空間を解放する。laddr には、MapMemory()で得た論理アドレスを指定しなければならない。MapMemory()でメモリの割り当てを行っていた場合は、そのメモリも解放される。

5.3 デバイス管理機能

5.3.1 デバイスの基本概念



［図 19］デバイス管理機能

(1) デバイス名 (UB*型)
　デバイス名は最大 8 文字の文字列で、次の要素により構成される。

```
#define L_DEVNM 8    /* デバイス名の長さ */
```

種別	デバイスの種別を示す名前 使用可能な文字は a～z A～Z
ユニット	物理的なデバイスを示す英字 1 文字 a～z でユニットごとに a から順に割当てる
サブユニット	論理的なデバイスを示す数字最大 3 文字 0～254 でサブユニットごとに 0 から順に割当てる

　デバイス名は、種別＋ユニット＋サブユニットの形式で表すが、ユニット、サブユニットはデバイスによっては存在しない場合もあり、その場合はそれぞれのフィールドは存在しない。
　特に、種別＋ユニットの形式を物理デバイス名と呼ぶ。また、種別＋ユニット＋サブユニットを論理デバイス名と呼び物理デバイス名と区別することがある。サブユニットがない場合は、物理デバイス名と論理デバイス名は同じになる。単にデバイス名と言ったときは、論理デバイス名を指す。
　サブユニットは、一般的にはハードディスクの区画(パーティション)を表すが、それ以外でも論理的なデバイスを表すために使用できる。

(例)	hda	ハードディスク(ディスク全体)
	hda0	ハードディスク(先頭の区画)
	fda	フロッピーディスク
	rsa	シリアルポート
	kbpd	キーボード／ポインティングデバイス

(2) デバイス ID(ID 型)

デバイス(デバイスドライバ)を T-Kernel/SM に登録することにより、デバイス(物理デバイス名)に対してデバイス ID (>0)が割当てられる。デバイス ID は物理デバイスごとに割当てられ、論理デバイスのデバイス ID は物理デバイスに割当てられたデバイス ID にサブユニット番号+1(1～255)を加えたものとなる。

devid: 登録時に割当てられたデバイス ID

devid	物理デバイス
devid + n+1	n 番目のサブユニット(論理デバイス)

(例)	hda	devid	ハードディスク全体
	hda0	devid + 1	ハードディスクの先頭の区画
	hda1	devid + 2	ハードディスクの２番目の区画

(3) デバイス属性 (ATR 型)

各デバイスの特徴を表しデバイスの種類分けを行うため、デバイス属性を次のように定義する。

```
IIII IIII IIII IIII PRxx xxxx KKKK KKKK
```

上位 16 ビットは、デバイス依存属性で、デバイスごとに定義される。下位 16 ビットは、標準属性で、下記のように定義される。

```
#define TD_PROTECT        0x8000   /* P: 書込み禁止 */
#define TD_REMOVABLE      0x4000   /* R: メディアの取り外し可能 */

#define TD_DEVKIND        0x00ff   /* K: デバイス/メディア種別 */
#define TD_DEVTYPE        0x00f0   /*デバイスタイプ */

                                   /* デバイスタイプ */
#define TDK_UNDEF         0x0000   /* 未定義/不明 */
#define TDK_DISK          0x0010   /* ディスクデバイス */

/* ディスク種別 */
#define TDK_DISK_UNDEF    0x0010   /* その他のディスク */
#define TDK_DISK_RAM      0x0011   /* RAM ディスク (主メモリ使用) */
#define TDK_DISK_ROM      0x0012   /* ROM ディスク (主メモリ使用) */
#define TDK_DISK_FLA      0x0013   /* Flash ROM、その他のシリコンディスク */
#define TDK_DISK_FD       0x0014   /* フロッピーディスク */
#define TDK_DISK_HD       0x0015   /* ハードディスク */
#define TDK_DISK_CDROM    0x0016   /* CD-ROM */
```

上記以外のデバイスタイプは、デバイスドライバ仕様書で別途定義される。未定義のデバイスは、未定義 TDK_UNDEF とする。なお、デバイスタイプは、アプリケーションがデバイス/メディアの種類によって処理内容を変える必要がある場合など、デバイスを利用する側で区別が必要となる場合に定義する。特に明確な区別をする必要がないデバイスに関しては、デバイスタイプを割当てる必要はない。

デバイス依存属性については、各デバイスドライバの仕様書を参照のこと。

(4) デバイスディスクリプタ (ID 型)

デバイスをアクセスするための識別子で、デバイスをオープンしたときに T-Kernel/SM によりデバイスディスクリプタ(>0)が割当てられる。

デバイスディスクリプタは、リソースグループに所属する。デバイスディスクリプタを使用した操作は、そのデバイスディスクリプタと同じリソースグループに所属するタスクからしかできない。異なるリソースグループに所属するタスクからの要求はエラーとなる(E_OACV)。

(5) リクエスト ID (ID 型)

デバイスに対して、入出力を要求したときにその要求の識別子としてリクエスト ID (>0)が割当てられる。このリクエスト ID により入出力の完了を待つことができる。

(6) データ番号 (INT 型)

デバイスのデータはデータ番号により指定する。データは次の 2 種類に大別される。

固有データ : データ番号≧0

デバイス固有のデータで、データ番号はデバイスごとに定義される。

(例)

ディスク	データ番号＝物理ブロック番号
シリアル回線	データ番号は 0 のみ使用

属性データ : データ番号＜0

ドライバやデバイスの状態の取得や設定、特殊機能などを指定する。

データ番号のいくつかは共通で定義されているが、デバイス独自にも定義できる。詳細は後述。

5.3.2 アプリケーションインタフェース

アプリケーションインタフェースには下記の関数があり、拡張 SVC により呼び出す。これらの関数は、タスク独立部およびディスパッチ禁止中、割り込み禁止中に呼び出すことはできない(E_CTX)。

```
ID  tk_opn_dev( UB *devnm, UINT omode )
ER  tk_cls_dev( ID dd, UINT option )
ID  tk_rea_dev( ID dd, INT start, VP buf, INT size, TMO tmout )
ER  tk_srea_dev( ID dd, INT start, VP buf, INT size, INT *asize )
ID  tk_wri_dev( ID dd, INT start, VP buf, INT size, TMO tmout )
ER  tk_swri_dev( ID dd, INT start, VP buf, INT size, INT *asize )
ID  tk_wai_dev( ID dd, ID reqid, INT *asize, ER *ioer, TMO tmout )
INT tk_sus_dev( UINT mode )
ID  tk_get_dev( ID devid, UB *devnm )
ID  tk_ref_dev( UB *devnm, T_RDEV *rdev )
ID  tk_oref_dev( ID dd, T_RDEV *rdev )
INT tk_lst_dev( T_LDEV *ldev, INT start, INT ndev )
INT tk_evt_dev( ID devid, INT evttyp, VP evtinf )
```

```
ID tk_opn_dev ( UB *devnm, UINT omode )
```

devnm	DeviceName	デバイス名
omode	OpenMode	オープンモード
戻値	DeviceDescriptor	デバイスディスクリプタ
または	ErrorCode	エラーコード

devnm で指定したデバイスを omode で指定したモードでオープンし、デバイスへのアクセスを準備する。戻値に、デバイスディスクリプタを返す。

```
omode := (TD_READ || TD_WRITE || TD_UPDATE) | [TD_EXCL || TD_WEXCL || TD_REXCL]
         | [TD_NOLOCK]

#define TD_READ    0x0001  /* 読込み専用 */
#define TD_WRITE   0x0002  /* 書込み専用 */
#define TD_UPDATE  0x0003  /* 読込みおよび書込み */
#define TD_EXCL    0x0100  /* 排他 */
#define TD_WEXCL   0x0200  /* 排他書込み */
#define TD_REXCL   0x0400  /* 排他読込み */
#define TD_NOLOCK  0x1000  /* ロック(常駐化)不要 */
```

TD_READ　読込み専用
TD_WRITE　書込み専用
TD_UPDATE　読込みおよび書込み
アクセスモードを指定する。
TD_READ の場合は、tk_wri_dev()は使用できない。
TD_WRITE の場合は、tk_rea_dev()は使用できない。

TD_EXCL　排他
TD_WEXCL　排他書込み
TD_REXCL　排他読込み
排他モードを指定する。
TD_EXCL は、一切の同時オープンを禁止する。
TD_WEXCL は、書込みモード(TD_WRITE または TD_UPDATE)による同時オープンを禁止する。
TD_REXCL は、読込みモード(TD_READ または TD_UPDATE)による同時オープンを禁止する。

［表 5］同じデバイスを同時にオープンしようとしたときの可否

現在オープンモード		同時オープンモード								
		排他指定なし			TD_WEXCL			TD_REXCL			TD_EXCL		
		R	U	W	R	U	W	R	U	W	R	U	W
排他指定なし	R	○	○	○	○	○	○	×	×	×	×	×	×
	U	○	○	○	×	×	×	×	×	×	×	×	×
	W	○	○	○	×	×	×	○	○	○	×	×	×
TD_WEXCL	R	○	×	×	○	×	×	×	×	×	×	×	×
	U	○	×	×	×	×	×	×	×	×	×	×	×
	W	○	×	×	×	×	×	○	×	×	×	×	×
TD_REXCL	R	×	×	○	×	×	○	×	×	×	×	×	×
	U	×	×	○	×	×	×	×	×	×	×	×	×
	W	×	×	○	×	×	×	×	×	○	×	×	×
TD_EXCL	R	×	×	×	×	×	×	×	×	×	×	×	×
	U	×	×	×	×	×	×	×	×	×	×	×	×
	W	×	×	×	×	×	×	×	×	×	×	×	×

R = TD_READ　W = TD_WRITE　U = TD_UPDATE
○ = オープン可　× = オープン不可(E_BUSY)

TD_NOLOCK　ロック(常駐化)不要
　入出力時(tk_rea_dev/tk_wri_dev)に指定したメモリ(buf)の領域は、呼出側でロック(常駐化)済みで、デバイスドライバ側ではロック不要であることを指示する。この場合、デバイスドライバではロックを行わない(行ってはいけない)。この指定は、仮想記憶システムにおいてページイン／ページアウトのためのディスクアクセスを行う場合などに使用する。一般的には指定する必要はない。
　デバイスディスクリプタは、オープンしたタスクのリソースグループに所属する。
　なお、物理デバイスをオープンした場合、その物理デバイスに属する論理デバイスをすべて同じモードでオープンしたのと同様に扱い、排他オープンの処理が行われる。

E_BUSY	デバイスは使用中(排他オープン中)
E_NOEXS	デバイスは存在しない
E_LIMIT	オープン可能な最大数を超えた
その他	デバイスドライバから返されたエラーコード

```
ER tk_cls_dev ( ID dd, UINT option )
```

dd	DeviceDescriptor	デバイスディスクリプタ
option	CloseOption	クローズオプション
戻値	ErrorCode	エラーコード

　dd のデバイスディスクリプタをクローズする。処理中の要求があった場合は、その処理を中止させてクローズする。

```
option := [TD_EJECT]
#define TD_EJECT      0x0001      /* メディア排出 */
```

TD_EJECT　メディア排出
　同一デバイスが他からオープンされていなければ、メディアを排出する。ただし、メディアの排出ができないデバイスでは無視される。

　サブシステムのクリーンアップ処理(tk_cln_ssy)により、対象リソースグループに属するデバイスディスクリプタはすべてクローズされる。

E_ID	dd が不正またはオープンされていない
その他	デバイスドライバから返されたエラーコード

```
ID tk_rea_dev ( ID dd, INT start, VP buf, INT size, TMO tmout )
```

dd	DeviceDescriptor	デバイスディスクリプタ
start	StartLocation	読込み開始位置（≧0:固有データ, ＜0:属性データ）
buf	Buffer	読み込んだデータを格納するバッファ
size	ReadSize	読み込むサイズ
tmout	Timeout	要求受付待ちタイムアウト時間(ミリ秒)
戻値	RequestID	リクエスト ID
または	ErrorCode	エラーコード

デバイスから固有データまたは属性データの読込みを開始する。読込みを開始するのみで、読込み完了を待たずに呼び出し元へ戻る。読込みが完了するまで、buf を保持しなければならない。読込み完了は tk_wai_dev()により待つ。読込み開始のための処理にかかる時間はデバイスドライバにより異なる。必ずしも即座に戻るとは限らない。

固有データの場合、start および size の単位はデバイスごとに決められる。属性データの場合、start は属性データ番号、size はバイト数となり、start のデータ番号の属性データを読み込む。通常、size は読み込む属性データのサイズ以上でなければならない。複数の属性データを一度に読み込むことはできない。size=0 を指定した場合、実際の読込みは行わず、現時点で読込み可能なサイズを調べる。

読込みまたは書込みの動作中である場合、新たな要求を受け付けられるか否かはデバイスドライバによる。新たな要求を受け付けられない状態の場合、要求受付待ちとなる。要求受付待ちのタイムアウト時間を tmout に指定する。tmout には TMO_POL または TMO_FEVR を指定することもできる。なお、タイムアウトするのは要求受付までである。要求が受け付けられた後にはタイムアウトしない。

E_ID	dd が不正またはオープンされていない
E_OACV	オープンモードが不正(読込みが許可されていない)
E_LIMIT	最大リクエスト数を超えた
E_TMOUT	他の要求を処理中で受け付けられない
E_ABORT	処理中止
その他	デバイスドライバから返されたエラーコード

```
ER tk_srea_dev ( ID dd, INT start, VP buf, INT size, INT *asize )
```

同期読込み。以下と等価である。

```
ER tk_srea_dev( ID dd, INT start, VP buf, INT size, INT *asize )
{
        ER  er, ioer;

        er = tk_rea_dev(dd, start, buf, size, TMO_FEVR);
        if ( er > 0 ) {
                er = tk_wai_dev(dd, er, asize, &ioer, TMO_FEVR);
                if ( er > 0 ) er = ioer;
        }
        return er;
}
```

```
ID tk_wri_dev ( ID dd, INT start, VP buf, INT size, TMO tmout )
```

dd	DeviceDescriptor	デバイスディスクリプタ
start	StartLocation	書込み開始位置(≧0:固有データ, <0:属性データ)
buf	Buffer	書き込むデータを格納したバッファ
size	WriteSize	書き込むサイズ
tmout	Timeout	要求受付待ちタイムアウト時間(ミリ秒)
戻値	RequestID	リクエスト ID
または	ErrorCode	エラーコード

デバイスへ固有データまたは属性データの書込みを開始する。書込みを開始するのみで、書込み完了を待たずに呼び出し元へ戻る。書込みが完了するまで、buf を保持しなければならない。書込み完了は tk_wai_dev()により待つ。書込み開始のための処理にかかる時間はデバイスドライバにより異なる。必ずしも即座に戻るとは限らない。

固有データの場合、start および size の単位はデバイスごとに決められる。属性データの場合、start は属性データ番号、size はバイト数となり、start のデータ番号の属性データに書き込む。通常、size は書き込む属性データのサイズと同じでなければならない。複数の属性データを一度に書き込むことはできない。size=0 を指定した場合、実際の書込みは行わず、現時点で書込み可能なサイズを調べる。

読込みまたは書込みの動作中である場合、新たな要求を受け付けられるか否かはデバイスドライバによる。新たな要求を受け付けられない状態の場合、要求受付待ちとなる。要求受付待ちのタイムアウト時間を tmout に指定する。tmout には TMO_POL または TMO_FEVR を指定することもできる。なお、タイムアウトするのは要求受付までである。要求が受け付けられた後にはタイムアウトしない。

E_ID	dd が不正またはオープンされていない
E_OACV	オープンモードが不正(書込みが許可されていない)
E_RONLY	書き込めないデバイス
E_LIMIT	最大リクエスト数を超えた
E_TMOUT	他の要求を処理中で受け付けられない
E_ABORT	処理中止
その他	デバイスドライバから返されたエラーコード

```
ER tk_swri_dev ( ID dd, INT start, VP buf, INT size, INT *asize )
```

同期書込み。以下と等価である。

```
ER tk_swri_dev( ID dd, INT start, VP buf, INT size, INT *asize )
{
	ER  er, ioer;

	er = tk_wri_dev(dd, start, buf, size, TMO_FEVR);
	if ( er > 0 ) {
		er = tk_wai_dev(dd, er, asize, &ioer, TMO_FEVR);
		if ( er > 0 ) er = ioer;
	}

	return er;
}
```

```
ID tk_wai_dev ( ID dd, ID reqid, INT *asize, ER *ioer, TMO tmout )
```

dd	DeviceDescriptor	デバイスディスクリプタ
reqid	RequestID	リクエスト ID
asize	Read/WriteSize	読込み/書込みサイズを返す
ioer	I/O Error	入出力エラーを返す
tmout	Timeout	タイムアウト時間(ミリ秒)
戻値	RequestID	完了したリクエスト ID
または	ErrorCode	エラーコード

dd に対する reqid の要求の完了を待つ。reqid=0 の場合は、dd に対する要求の内のいずれかが完了するのを待つ。なお、現時点で処理中の要求のみが完了待ちの対象となる。tk_wai_dev()の呼出後に要求された処理は完了待ちの対象とならない。

複数の要求を同時に処理している場合、その要求の完了の順序は必ずしも要求した順序ではなく、デバイスドライバに依存する。ただし、要求した順序で処理した場合と結果が矛盾しないような順序で処理されることは保証される。例えば、ディスクからの読込みの場合、次のような処理順序の変更が考えられる。

要求順ブロック番号　1 4 3 2 5
処理順ブロック番号　1 2 3 4 5

このように順序を入れ替えて処理することにより、シークや回転待ちを減らすことができ、より効率的にディスクアクセスができる。

tmout に完了待ちのタイムアウト時間を指定する。TMO_POL または TMO_FEVR を指定することもできる。タイムアウト(E_TMOUT)した場合は要求された処理を継続中なので、再度 tk_wai_dev()により完了を待つ必要がある。reqid>0 かつ tmout=TMO_FEVR の場合はタイムアウトすることはなく、必ず処理が完了する。

要求された処理の結果のエラー(入出力エラーなど)は、戻値ではなく ioer に格納される。戻値には、要求の完了待ちが正しくできなかった場合にエラーを返す。戻値にエラーが返された場合、ioer の内容は無意味である。また、戻値にエラーが返された場合は処理を継続中なので、再度 tk_wai_dev()により完了を待つ必要がある。

tk_wai_dev()で完了待ち中にタスク例外が発生すると、reqid の要求を中止して処理を完了させる。中止した処理の結果がどのようになるかは、デバイスドライバに依存する。ただし、reqid=0 の場合は、要求を中止することなくタイムアウトと同様に扱われる。この場合は、E_TMOUT ではなく E_ABORT を返す。

同じリクエスト ID に対して、複数のタスクから同時に完了待ちすることはできない。reqid=0 で待っているタスクがあれば、同じ dd に対して他のタスクは完了待ちできない。同様に、reqid>0 で待っているタスクがあれば、他のタスクで reqid=0 の完了待ちはできない。

E_ID	dd が不正またはオープンされていない reqid が不正または dd に対する要求ではない
E_OBJ	reqid の要求は他のタスクで完了待ちしている
E_NOEXS	処理中の要求はない(reqid=0 の場合のみ)
E_TMOUT	タイムアウト(処理継続中)
E_ABORT	処理中止
その他	デバイスドライバから返されたエラーコード

```
INT tk_sus_dev ( UINT mode )
```

mode	Mode	モード
戻値	Suspend disable request count	サスペンド禁止要求カウント数
または	ErrorCode	エラーコード

mode にしたがって処理を行い、その処理を行ったあとのサスペンド禁止要求カウント数を戻値に返す。

mode := ((TD_SUSPEND | [TD_FORCE]) || TD_DISSUS || TD_ENASUS || TD_CHECK)

```
#define TD_SUSPEND    0x0001  /* サスペンド */
#define TD_DISSUS     0x0002  /* サスペンドを禁止 */
#define TD_ENASUS     0x0003  /* サスペンドを許可 */
#define TD_CHECK      0x0004  /* サスペンド禁止要求カウント取得 */
#define TD_FORCE      0x8000  /* 強制サスペンド指定 */
```

TD_SUSPEND　　サスペンド
サスペンド許可状態であればサスペンドする。
サスペンド禁止状態であれば E_BUSY を返す。

TD_SUSPEND|TD_FORCE　　強制サスペンド
サスペンド禁止状態であってもサスペンドする。

TD_DISSUS　　サスペンド禁止
サスペンドを禁止する。

TD_ENASUS　　サスペンド許可
サスペンドを許可する。
自リソースグループでサスペンド禁止した回数以上に許可した場合は何もしない。

TD_CHECK　　サスペンド禁止カウント取得
サスペンド禁止要求を行っている回数の取得のみ行う。

サスペンドは、次の手順によって行われる。

1. 各サブシステムのサスペンド開始前の処理
   tk_evt_ssy(0, TSEVT_SUSPEND_BEGIN,0,0)
2. 各ディスクデバイス以外のサスペンド処理
3. 各ディスクデバイスのサスペンド処理
4. 各サブシステムのサスペンド完了後の処理
   tk_evt_ssy(0, TSEVT_SUSPEND_DONE,0,0)
5. サスペンド状態へ移行
   tk_set_pow(TPW_DOSUSPEND)

リジューム(サスペンドからの復帰)は、次の手順によって行われる。

1. サスペンド状態から復帰
   tk_set_pow(TPW_DOSUSPEND)から戻る
2. 各サブシステムのリジューム開始前の処理
   tk_evt_ssy(0,TSEVT_RESUME_BEGIN,0,0)
3. 各ディスクデバイスのリジューム処理
4. 各ディスクデバイス以外のリジューム処理
5. 各サブシステムのリジューム完了後の処理
   tk_evt_ssy(0,TSEVT_RESUME_DONE,0,0)

サスペンド禁止は、その要求回数がカウントされる。同じ回数だけサスペンド許可を要求しないとサスペンドは許可されない。システム起動時はサスペンド許可(サスペンド禁止要求カウント＝0)である。サスペンド禁止要求カ

ウントはシステム全体で 1 つだが、その要求がどのリソースグループから行われたかを管理する。他のリソースグループで行ったサスペンド禁止要求を解除することはできない。なお、リソースグループのクリーンアップ処理が行われることによって、そのリソースグループで行ったサスペンド要求はすべて解除され、サスペンド禁止要求カウントは減算される。サスペンド禁止要求カウントの上限は実装依存だが、最低 255 回まではカウントできるものとする。上限を超えた場合は E_QOVR を返す。

E_BUSY　　サスペンド禁止中
E_QOVR　　サスペンド禁止要求カウントオーバー

```
ID tk_get_dev ( ID devid, UB *devnm )
```

devid	DeviceID	デバイス ID
devnm	DeviceName	デバイス名の格納領域
戻値	Device ID of Physical Device	物理デバイスのデバイス ID
または	ErrorCode	エラーコード

devid で示すデバイスのデバイス名を取得し devnm に格納する。
devid は物理デバイスのデバイス ID または論理デバイスのデバイス ID である。
devid が物理デバイスであれば、devnm には物理デバイス名が格納される。
devid が論理デバイスであれば、devnm には論理デバイス名が格納される。
devnm は L_DEVNM+1 バイト以上の領域が必要である。

戻値には、devid のデバイスが属する物理デバイスのデバイス ID を返す。

E_NOEXS　　devid のデバイスは存在しない

```
ID tk_ref_dev ( UB *devnm, T_RDEV *rdev )
```

```
ID tk_oref_dev ( ID dd, T_RDEV *rdev )
```

devnm	DeviceName	デバイス名
dd	DeviceDescriptor	デバイスディスクリプタ
rdev	DeviceInformation	デバイス情報
戻値	DeviceID	デバイス ID
または	ErrorCode	エラーコード

```
typedef struct t_rdev {
        ATR     devatr;     /* デバイス属性 */
        INT     blksz;      /* 固有データのブロックサイズ (-1:不明) */
        INT     nsub;       /* サブユニット数 */
        INT     subno;      /* 0:物理デバイス 1～nsub:サブユニット番号+1 */
        /* 以下に実装独自の情報が追加される場合がある */
} T_RDEV;
```

devnm または dd で示すデバイスのデバイス情報を取得し、rdev に格納する。rdev=NULL とした場合には、デバイス情報は格納されない。

nsub は、devnm または dd で示すデバイスが属する物理デバイスのサブユニット数である。

戻値に devnm のデバイスのデバイス ID を返す。

E_NOEXS devnm のデバイスは存在しない

```
INT tk_lst_dev ( T_LDEV *ldev, INT start, INT ndev )
```

ldev	ListOfDevices	登録デバイス情報の格納領域(配列)
start	StartingNumber	開始番号
ndev	NumberOfDevices	取得数
戻値	RemainingDevices	残りの登録数
または	ErrorCode	エラーコード

```
typedef struct t_ldev {
    ATR     devatr;             /* デバイス属性 */
    INT     blksz;              /* 固有データのブロックサイズ (-1:不明) */
    INT     nsub;               /* サブユニット数 */
    UB      devnm[L_DEVNM];     /* 物理デバイス名 */
    /* 以下に実装独自の情報が追加される場合がある */
} T_LDEV;
```

登録済みのデバイスの情報を取得する。登録デバイスは物理デバイス単位で管理される。したがって、登録デバイス情報も物理デバイス単位で取得される。

登録デバイスの数が N の時、登録デバイスに 0～N-1 の連番を振る。この連番にしたがって、start 番目から ndev 個の登録情報を取得し、ldev に格納する。ldev は ndev 個の情報を格納するのに十分な大きさの領域でなければならない。戻値には、start 以降の残りの登録数(N - start)を返す。

start 以降の残りが ndev 個に満たない場合は、残りのすべてを格納する。戻値≦ndev であれば、すべての登録情報が取得できたことを示す。なお、この連番はデバイスの登録・抹消があると変化する。したがって、複数回に分けて取得すると、正確な情報が得られない場合がある。

E_NOEXS start が登録数を超えている

```
INT tk_evt_dev ( ID devid, INT evttyp, VP evtinf )
```

devid	DeviceID	イベント送信先デバイス ID
evttyp	EventType	ドライバ要求イベントタイプ
evtinf	EventInformation	イベントタイプ別の情報
戻値	ReturnCode	デバイスドライバからの戻値
または	ErrorCode	エラーコード

devid のデバイス(デバイスドライバ)に、ドライバ要求イベントを送信する。

ドライバ要求イベントタイプには下記のものがある。

```
#define TDV_CARDEVT    1    /* PC カードイベント (カードマネージャ参照) */
#define TDV_USBEVT     2    /* USB イベント  (USB マネージャ参照) */
```

ドライバ要求イベントの機能(処理内容)および evtinf の内容はイベントタイプごとに定義される。

E_NOEXS	devid のデバイスは存在しない
E_PAR	デバイス管理内部イベント(evttyp＜0)は指定できない

5.3.3 デバイス登録

以下のデバイス登録情報により、デバイスを登録する。デバイスは、物理デバイスごとに登録する。

```
typedef struct t_ddev {
     VP  exinf;       /* 拡張情報 */
     ATR drvatr;      /* ドライバ属性 */
     ATR devatr;      /* デバイス属性 */
     INT nsub;        /* サブユニット数 */
     INT blksz;       /* 固有データのブロックサイズ（-1:不明）*/
     FP  openfn;      /* オープン関数 */
     FP  closefn;     /* クローズ関数 */
     FP  execfn;      /* 処理開始関数 */
     FP  waitfn;      /* 完了待ち関数 */
     FP  abortfn;     /* 中止処理関数 */
     FP  eventfn;     /* イベント関数 */
     /* 以下に実装独自の情報が追加される場合がある */
} T_DDEV;
```

exinf は、任意の情報の格納に使用できる。この値は、各処理関数に渡される。内容に関してデバイス管理では関知しない。

drvatr は、デバイスドライバの属性に関する情報を設定する。下位側がシステム属性を表し、上位側が実装独自属性を表す。実装独自属性は、T_DDEV に実装独自データを追加する場合に有効フラグを定義するためなどに使用する。

```
drvatr := [TDA_OPENREQ]
#define TDA_OPENREQ      0x0001  /* 毎回オープン/クローズ */
```

TDA_OPENREQ

デバイスが多重オープンされた場合、通常は最初のオープン時に openfn が呼び出され、最後のクローズ時に closefn が呼び出される。TDA_OPENREQ を指定した場合、多重オープンの場合でもすべてのオープン/クローズ時に openfn/closefn が呼び出される。

devatr は、デバイス属性を設定する。デバイス属性の詳細については前述。

nsub は、サブユニット数を設定する。サブユニットがない場合は 0 とする。

blksz は、固有データのブロックサイズをバイト数で設定する。ディスクデバイスの場合は、物理ブロックサイズとなる。シリアル回線などは 1 バイトとなる。固有データのないデバイスでは 0 とする。未フォーマットのディスクなど、ブロックサイズが不明の場合は-1 とする。blksz<=0 の場合は、固有データにアクセスできない。tk_rea_dev,tk_wri_dev で固有データをアクセスする場合に、size＊blksz がアクセスする領域サイズ、つまり buf のサイズとならなければならない。

openfn, closefn, execfn, waitfn, abortfn, eventfn は、処理関数のエントリーアドレスを設定する。処理関数の詳細については後述。

```
ID tk_def_dev ( UB *devnm, T_DDEV *ddev, T_IDEV *idev )
```

devnm	PhysicalDeviceName	物理デバイス名
ddev	DefineDevice	デバイス登録情報
idev	InitialDeviceInformation	デバイス初期情報が返される
戻値	DeviceID	デバイス ID
または	ErrorCode	エラーコード

devnm のデバイス名でデバイスを登録する。devnm のデバイスがすでに登録されているときは、新しい登録情報で更新する。更新の場合は、デバイス ID は変更されない。ddev=NULL の場合は、devnm のデバイス登録を抹消する。
idev にデバイス初期情報が返される。ここには、デバイスドライバ起動時のデフォルトとして設定するための情報などが返されるので、必要に応じて利用する。idev=NULL とした場合には、デバイス初期情報は格納されない。

```
typedef struct t_idev {
    ID        evtmbfid;       /* 事象通知用メッセージバッファ ID */
    /* 以下に実装独自の情報が追加される場合がある */
} T_IDEV;
```

evtmbfid は、システムデフォルトの事象通知用メッセージバッファ ID である。システムデフォルトの事象通知用メッセージバッファがない場合は 0 が設定される。
デバイス登録および抹消が行われたとき、各サブシステムに対して次のように通知が行われる。devid は登録・抹消された、物理デバイスのデバイス ID である。

デバイス登録・更新時	tk_evt_ssy(0, TSEVT_DEVICE_REGIST, 0, devid)
デバイス抹消時	tk_evt_ssy(0, TSEVT_DEVICE_DELETE, 0, devid)

E_LIMIT	登録可能な最大数を超えた
E_NOEXS	devnm のデバイスは存在しない(ddev＝NULL の場合)

```
ER tk_ref_idv ( T_IDEV *idev )
```

idev	InitialDeviceInformation	デバイス初期情報が返される
戻値	ErrorCode	エラーコード

デバイス初期情報を取得する。tk_def_dev()で得られるものと同じ内容である。

5.3.4 デバイスドライバインタフェース

デバイスドライバインタフェースは、デバイス登録時に指定した処理関数群によって構成される。これらの処理関数は、デバイス管理によって呼び出され、準タスク部として実行される。これらの処理関数は、再入可能(reentrant)でなければならない。また、処理関数が排他的に呼び出されることは保証されない。例えば、複数のタスクから同じデバイスに対して同時に要求があった場合、それぞれのタスクが同時に処理関数を呼び出すことがある。デバイスドライバ側は、必要に応じて排他制御などを行う必要がある。

デバイスドライバへの入出力要求は、リクエスト ID に対応した下記の要求パケットにより行う。

```
typedef struct t_devreq {
    struct t_devreq *next;      /* I:要求パケットのリンク (NULL:終端) */
    VP          exinf;          /* X:拡張情報 */
    ID          devid;          /* I:対象デバイス ID */
    INT         cmd:4;          /* I:要求コマンド */
    BOOL        abort:1;        /* I:中止要求を行った時 TRUE */
    BOOL        nolock:1;       /* I:ロック(常駐化)不要の時 TRUE */
    INT         rsv:26;         /* I:予約(常に 0) */
    T_TSKSPC    tskspc;         /* I:要求タスクのタスク固有空間 */
    INT         start;          /* I:開始データ番号 */
    INT         size;           /* I:要求サイズ */
    VP          buf;            /* I:入出力バッファアドレス */
    INT         asize;          /* O:結果サイズ */
    ER          error;          /* O:結果エラー */
    /* 以下に実装独自の情報が追加される場合がある */
} T_DEVREQ;
```

I は入力パラメータ、O は出力パラメータで、入力パラメータはデバイスドライバで変更してはならない。入力パラメータ(I)以外は、デバイス管理により最初に 0 クリアされる。その後はデバイス管理は変更しない。

next は、要求パケットをリンクするために使用する。デバイス管理内の要求パケットの管理に使用される他、完了待ち関数(waitfn),中止処理関数(abortfn)でも使用する。

exinf は、デバイスドライバ側で任意に使用できる。デバイス管理では内容には関知しない。

devid は、要求対象のデバイス ID が設定される。

cmd は、要求コマンドが設定される。

```
cmd := (TDC_READ || TDC_WRITE)

#define TDC_READ    1   /* 読込み要求 */
#define TDC_WRITE   2   /* 書込み要求 */
```

abort は中止処理を行う場合、中止処理関数(abortfn)を呼び出す直前に TRUE を設定する。abort は中止処理を要求したことを示すフラグであり、処理が中止されたことを示すものではない。また、中止処理関数(abortfn)を呼び出さない場合でも abort が TRUE に設定される場合がある。abort が TRUE に設定された要求がデバイスドライバに渡された場合は、中止処理を行う。

nolock は、buf の領域が呼出側ですでにロック(常駐化)されており、デバイスドライバ側ではロック不要であることを示す。この場合、デバイスドライバはロックを行ってはいけない。(デバイスドライバでロックを行うと正しく動作しなくなる可能性がある状況で nolock が指定される。しがたって、nolock=TRUE の場合はロックを行ってはいけない。)

tskspc は、要求タスクのタスク固有空間が設定される。処理関数は要求タスクのコンテキストで呼び出されるため、tskspc は処理関数のタスク固有空間と同じである。しかし、実際の入出力処理(buf の領域の読込み/書込み)をデバイスドライバ内の別タスクで行う場合は、入出力を行うタスクのタスク固有空間を要求タスクのタスク固有空間に切り替える必要がある。

start, size は、tk_rea_dev(), tk_wri_dev()で指定された start, size がそのまま設定される。

buf は、tk_rea_dev(), tk_wri_dev()で指定された buf がそのまま設定される。buf の指す領域は、非常駐である場合やタスク固有空間である場合がある。そのため、次の点に注意する必要がある。

- 非常駐のメモリは､タスク独立部やディスパッチ禁止･割込み禁止の状態でアクセスすることはできない。
- タスク固有空間のメモリは、異なるタスク固有空間からはアクセスできない。

そのため、必要に応じてタスク固有空間の切り替えや常駐化を行わなければならない。特に割込みハンドラから直接アクセスする場合には注意が必要である。一般的には、buf の領域に割込みハンドラから直接アクセスしない方がよい。また、buf の領域をアクセスする前に、buf の領域の有効性をアドレス空間チェック機能(ChkSpace～)により検査する必要がある。(ChkSpace～については後述)

asize は、tk_wai_dev()の asize に返す値をデバイスドライバが設定する。

error は、tk_wai_dev()の戻値として返すエラーコードをデバイスドライバが設定する。正常であれば E_OK を設定する。

オープン関数：ER openfn(ID devid, UINT omode, VP exinf)

devid	DeviceID	オープンするデバイスのデバイス ID
omode	OpenMode	オープンモード(tk_opn_dev と同じ)
exinf	ExtendedInformation	デバイス登録時に指定した拡張情報
戻値	ErrorCode	エラーコード

tk_opn_dev()が呼び出されたときに openfn が呼び出される。

openfn では、デバイスの使用開始のための処理を行う。処理内容はデバイスに依存し、何もする必要がなければ何もしなくてもよい。また、オープンされているか否かをデバイスドライバで記憶する必要もなく、オープンされていない(openfn が呼び出されていない)という理由だけで、他の処理関数が呼び出されたときエラーにする必要もない。オープンされていない状態で他の処理関数が呼び出された場合でも、デバイスドライバの動作に問題がなければ要求を処理してしまって構わない。

openfn でデバイスの初期化等を行う場合でも、待ちを伴うような処理は原則として行わない。できる限り速やかに処理を行い openfn から戻らなければならない。例えば、シリアル回線のように通信モードを設定する必要があるようなデバイスでは、tk_wri_dev()により通信モードが設定されたときにデバイスの初期化を行えばよい。openfn ではデバイスの初期化を行う必要はない。

同じデバイスが多重オープンされた場合、通常は最初のオープン時のみ呼び出されるが、デバイス登録時にドライバ属性として TDA_OPENREQ が指定された場合は、すべてのオープン時に呼び出される。

多重オープンやオープンモードに関する処理はデバイス管理で行われるため、openfn ではそれらに関する処理は特に必要ない。omode も参考情報として渡されるだけで、omode に関する処理を行う必要はない。

クローズ関数：ER closefn(ID devid, UINT option, VP exinf)

devid	DeviceID	クローズするデバイスのデバイス ID
option	CloseOption	クローズオプション(tk_cls_dev と同じ)
exinf	ExtendedInformation	デバイス登録時に指定した拡張情報
戻値	ErrorCode	エラーコード

tk_cls_dev()が呼び出されたときに closefn が呼び出される。

closefn では、デバイスの使用終了のための処理を行う。処理内容はデバイスに依存し、何もする必要がなければ何もしなくてもよい。

メディアの取り外しが可能なデバイスの場合､option に TD_EJECT が指定されていたならメディアの排出を行う。

closefn でデバイスの終了処理やメディアの排出を行う場合でも、待ちを伴うような処理は原則として行わない。できる限り速やかに処理を行い closefn から戻らなければならない。メディアの排出に時間がかかる場合、排出の完了を待たずに closefn から戻っても構わない。

同じデバイスが多重オープンされていた場合、通常は最後のクローズ時のみ呼び出されるが、デバイス登録時にドライバ属性として TDA_OPENREQ が指定された場合は、すべてのクローズ時に呼び出される。ただし、この場合も最後のクローズにしか option に TDA_EJECT が指定されることはない。

多重オープンやオープンモードに関する処理はデバイス管理で行われるため、closefn ではそれらに関する処理は特に必要ない。

処理開始関数：ER execfn(T_DEVREQ *devreq, TMO tmout, VP exinf)

devreq	DeviceRequestPacket	要求パケット
tmout	Timeout	要求受付待ちタイムアウト時間(ミリ秒)
exinf	ExtendedInformation	デバイス登録時に指定した拡張情報
戻値	ErrorCode	エラーコード

tk_rea_dev()または tk_wri_dev()が呼び出されたときに execfn が呼び出される。

devreq の要求の処理を開始する。処理を開始するのみで、完了を待たずに呼び出し元へ戻る。処理開始のためにかかる時間はデバイスドライバに依存する。必ずしも即座に完了するとは限らない。

新たな要求を受け付けられない状態のときは、要求受付待ちとなる。tmout に指定した時間以内に新たな要求を受け付けられなければ、タイムアウトする。tmout には、TMO_POL または TMO_FEVR を指定することもできる。タイムアウトした場合、execfn の戻値に E_TMOUT を返す。要求パケットの error は変更しない。タイムアウトするのは要求を受け付けるまでで、要求を受け付けた後はタイムアウトしない。

execfn の戻値にエラーを返した場合は、要求が受け付けられなかったものとして、要求パケットは消滅する。

処理を中止する場合、その要求の受け付け前(処理開始前)であれば execfn の戻値に E_ABORT を返す。この場合、要求パケットは消滅する。要求受け付け後(処理開始後)の場合は、E_OK を返す。この場合、要求パケットは waitfn を実行し処理完了が確認されるまで消滅しない。

中止要求があった場合、execfn からできる限り速やかに戻らなければならない。処理を中止しなくてもすぐに終るのであれば中止しなくてもよい。

完了待ち関数：INT waitfn(T_DEVREQ *devreq, INT nreq, TMO tmout, VP exinf)

devreq	DeviceRequestPacket	要求パケットのリスト
nreq	NumberOfRequest	要求パケットの数
tmout	Timeout	タイムアウト時間(ミリ秒)
exinf	ExtendedInformation	デバイス登録時に指定した拡張情報
戻値	PacketNumber	完了した要求パケットの番号
または	ErrorCode	エラーコード

tk_wai_dev()が呼び出されたときに waitfn が呼び出される。

devreq は devreq->next で接続された要求パケットのリストで、devreq から nreq 個分の要求パケットについて、その内のいずれかが完了するのを待つ。リストの最後の next は必ずしも NULL になっているとは限らないため、必ず nreq の指定に従う。戻値に完了した要求パケットの番号(devreq から何番目か)を返す。最初が 0 番目で最後が nreq-1 番目となる。なお、完了とは、正常終了/異常(エラー)終了/中止のいずれかである。

tmout に完了待ちのタイムアウト時間を指定する。TMO_POL または TMO_FEVR を指定することもできる。タイムアウトした場合は、要求された処理は継続中である。タイムアウトの場合、waitfn の戻値に E_TMOUT を返す。要求パケットの error は変更しない。なお、要求された処理を継続中で waitfn から戻るときは、waitfn の戻値に必ずエラーを返す。戻値にエラーを返したにもかかわらず処理が完了していてはいけないし、処理継続中であればエラー以外を返してもいけない。waitfn の戻値にエラーが返されている限り、その要求は処理中として要求パケットは消滅しない。waitfn の戻値に処理を完了した要求パケットの番号が返されたとき、その要求の処理は完了したものとして要求パケットは消滅する。

入出力エラーなどデバイスに関するエラーを、要求パケットの error に格納する。waitfn の戻値には、完了待ちが正しくできなかった場合のエラーを返す。waitfn の戻値が tk_wai_dev の戻値となり、要求パケットの error が ioer に戻される。

単一要求の完了待ち(nreq=1)の場合と、複数要求の完了待ち(nreq>1)の場合では、中止の処理が異なる。単一要求の完了待ちの時は、処理中の要求を中止する。複数要求の完了待ちでは、要求の処理自体は中止せずに完了待ちのみを中止(待ち解除)する。複数要求待ちの中止(待ち解除)の場合、waitfn の戻値に E_ABORT を返す。

完了待ち要求の中には、中止要求が要求パケットの abort にセットされている場合がある。このような場合、単一要求の完了待ちではその要求の中止処理を行わなければならない。複数要求の完了待ちでも中止処理を行うのが望ましいが、abort フラグを無視しても構わない。

なお、中止処理では waitfn からできる限り速やかに戻ることが重要であり、処理を中止しなくてもすぐに処理が終るのであれば中止しなくてもよい。

処理が中止された場合は、要求パケットの error に E_ABORT を返すことを原則とするが、そのデバイスの特性に合わせて E_ABORT 以外のエラーを返してもよい。また、中止される直前までの処理を有効として E_OK としてもよい。なお、中止要求があっても正常に最後まで処理したのなら E_OK を返す。

中止処理関数：ER abortfn(ID tskid, T_DEVREQ *devreq, INT nreq, VP exinf)

tskid	TaskID	execfn, waitfn を実行しているタスクのタスク ID
devreq	DeviceRequestPacket	要求パケットのリスト
nreq	NumberOfRequestPackets	要求パケットの数
exinf	ExtendedInformation	デバイス登録時に指定した拡張情報
戻値	ErrorCode	エラーコード

abortfn は、指定された要求を実行中の execfn, waitfn から速やかに戻らせる。通常は、処理中の要求を中止して戻らせる。ただし、中止しなくてもすぐに処理が終るのであれば、必ずしも中止しなくてもよい。重要なのは、できる限り速やかに execfn, waitfn から戻ることである。

tskid は、devreq で指定した要求を実行中のタスクである。つまり、execfn, waitfn を実行しているタスクである。devreq, nreq は、execfn, waitfn の引数として指定したものと同じである。ただし、execfn の場合は常に nreq=1 である。

abortfn は、execfn, waitfn を実行しているタスクとは別のタスクから呼び出される。両者は並行して実行されるため、必要に応じて排他制御等を行う必要がある。また、execfn, waitfn の呼出直前や execfn, waitfn から戻る途中に abortfn が呼び出される可能性もある。このような場合においても正しく動作するように配慮する必要がある。abortfn を呼び出す前に、中止処理対象の要求パケットの abort フラグに TRUE が設定されるので、execfn, waitfn はこれにより中止要求の有無を知ることもできる。また、abortfn では、任意のオブジェクトに対する tk_dis_wai()を使用することができる。

複数要求待ち(nreq>1)の waitfn 実行中の場合は、特別な扱いとなり他の場合とは次の点で異なる。

- 要求の処理を中止することはせず、完了待ちのみ中止(待ちを解除)する。
- 要求パケットの abort フラグはセットされない(abort=FALSE のまま)。

なお、execfn, waitfn の実行中でないときに要求を中止させる場合には、abortfn を呼び出すことなく、要求パケットの abort フラグがセットされる。abort フラグがセットされた状態で execfn が呼び出されたときは、要求を受け付けない。waitfn が呼び出されたときは、abortfn が呼び出された場合と同様の中止処理を行う。

execfn により処理を開始した要求が、waitfn による完了待ちでない状態で中止された場合は、後で waitfn が呼び出されたときに中止されて処理が完了したことを知らせる。処理が中止されても、waitfn により完了が確認されるまでは要求自体は消滅しない。

abortfn は中止処理を開始するのみで、中止が完了するまで待たずに速やかに戻る。

abortfn は、次のような場合に呼び出される。

- タスク例外の発生によるブレーク関数の実行時に、タスク例外の発生したタスクによる処理中の要求があった場合、そのタスクによる処理中の要求を中止する。
- tk_cls_dev()およびサブシステムのクリーンアップ処理によるデバイスクローズ時に、クローズするデバイスディスクリプタによる処理中の要求があった場合、そのデバイスディスクリプタによる処理中の要求を中止する。

イベント関数：INT eventfn(INT evttyp, VP evtinf, VP exinf)

evttyp	EventType	ドライバ要求イベントタイプ
evtinf	EventInformation	イベントタイプ別の情報
exinf	ExtendedInformation	デバイス登録時に指定した拡張情報
戻値	ReturnCode	イベントタイプ別に定義された戻値
または	ErrorCode	エラーコード

ドライバ要求イベントタイプには下記のものがあり、正の値が tk_evt_dev()の呼び出しによるもの、負の値がデバイス管理内部からの呼び出しによるものである。

```
#define TDV_SUSPEND    (-1)     /* サスペンド */
#define TDV_RESUME     (-2)     /* リジューム */
#define TDV_CARDEVT    1        /* PC カードイベント    (カードマネージャ参照) */
#define TDV_USBEVT     2        /* USB イベント         (USB マネージャ参照) */
```

イベント関数で行う処理は、各イベントタイプごとに定義される。サスペンド/リジュームについては後述。
tk_evt_dev()による呼び出しの場合、eventfn の戻値はそのまま tk_evt_dev()の戻値となる。
イベント関数への要求は､他の要求の処理中であっても受け付け､できる限り速やかに処理しなければならない。

5.3.5 属性データ

属性データは大きく次の 3 つに分類される。

- 共通属性
  すべてのデバイス(デバイスドライバ)に共通に定義する属性
- デバイス種別属性
  同じ種類に分類されるデバイス(デバイスドライバ)に共通に定義する属性
- デバイス個別属性
  各デバイス(デバイスドライバ)ごとに独自に定義される属性

デバイス種別属性およびデバイス個別属性については、各デバイスごとの仕様書を参照のこと。ここでは、共通属性のみ定義する。
共通属性の属性データ番号は-1～-99 の範囲となる。共通属性のデータ番号はすべてのデバイスで共通となるが、すべてのデバイスが必ずしもすべての共通属性に対応しているとは限らない。対応していないデータ番号を指定されたときは、エラーE_PAR とする。

```
#define TDN_EVENT       (-1)    /* RW:事象通知用メッセージバッファ ID */
#define TDN_DISKINFO    (-2)    /* R-:ディスク情報 */
#define TDN_DISPSPEC    (-3)    /* R-:表示デバイス仕様 */
```

RW: 読込み(tk_rea_dev)/書込み(tk_wri_dev)可能
R-: 読込み(tk_rea_dev)のみ可能

TDN_EVENT : 事象通知用メッセージバッファ ID

データ形式　ID

デバイス事象通知用のメッセージバッファの ID である。デバイス登録時にシステムデフォルトのメッセージバッファ ID が渡されるので、ドライバ起動時の初期設定としてその ID を設定する。
0 が設定されている場合は、デバイス事象通知を行わない。デバイス事象通知については後述。

TDN_DISKINFO : ディスク情報

データ形式　DiskInfo

```
typedef enum {
    DiskFmt_STD     = 0,        /* 標準 (HD など) */
    DiskFmt_2DD     = 1,        /* 2DD 720KB */
    DiskFmt_2HD     = 2,        /* 2HD 1.44MB */
    DiskFmt_CDROM   = 4,        /* CD-ROM 640MB */
} DiskFormat;

typedef struct {
    DiskFormat format;          /* フォーマット形式 */
    UW   protect:1;             /* プロテクト状態 */
    UW   removable:1;           /* 取り外し可否 */
    UW   rsv:30;                /* 予約 (常に 0) */
    W    blocksize;             /* ブロックバイト数 */
    W    blockcount;            /* 総ブロック数 */
} DiskInfo;
```

詳細は、ディスクドライバ仕様書を参照のこと。

TDN_DISPSPEC : 表示デバイス仕様

データ形式　DEV_SPEC

```
typedef struct {
    H    attr;                  /* デバイス属性 */
    H    planes;                /* プレーン数 */
    H    pixbits;               /* ピクセルビット数(境界/有効) */
    H    hpixels;               /* 横のピクセル数 */
    H    vpixels;               /* 縦のピクセル数 */
    H    hres;                  /* 横の解像度 */
    H    vres;                  /* 縦の解像度 */
    H    color[4];              /* カラー情報 */
    H    resv[6];               /* 予約 */
} DEV_SPEC;
```

詳細は、スクリーンドライバ仕様書を参照のこと。

5.3.6 デバイス事象通知

デバイスドライバは、デバイスで発生した事象を、事象通知用メッセージバッファ(TDN_EVENT)にデバイス事象通知として送信する。事象通知用メッセージバッファは、システムデフォルトのものがデバイス登録時に指定されるが、後から変更することも可能である。システムデフォルトの事象通知用メッセージバッファは、システム構成情報の TDEvtMbfSz により定義される。

事象タイプには以下のものがある。

```
typedef enum tdevttyp {
    TDE_unknown      = 0,        /* 未定義 */
    TDE_MOUNT        = 0x01,     /* メディア挿入 */
    TDE_EJECT        = 0x02,     /* メディア排出 */
    TDE_ILLMOUNT     = 0x03,     /* メディア不正挿入 */
    TDE_ILLEJECT     = 0x04,     /* メディア不正排出 */
    TDE_REMOUNT      = 0x05,     /* メディア再挿入 */
    TDE_CARDBATLOW   = 0x06,     /* カードバッテリ残量警告 */
    TDE_CARDBATFAIL  = 0x07,     /* カードバッテリ異常 */
    TDE_REQEJECT     = 0x08,     /* メディア排出要求 */
    TDE_PDBUT        = 0x11,     /* PD ボタン状態の変化 */
    TDE_PDMOVE       = 0x12,     /* PD 位置移動 */
    TDE_PDSTATE      = 0x13,     /* PD の状態変化 */
    TDE_PDEXT        = 0x14,     /* PD 拡張事象 */
    TDE_KEYDOWN      = 0x21,     /* キーダウン */
    TDE_KEYUP        = 0x22,     /* キーアップ */
    TDE_KEYMETA      = 0x23,     /* メタキー状態の変化 */
    TDE_POWEROFF     = 0x31,     /* 電源スイッチオフ */
    TDE_POWERLOW     = 0x32,     /* 電源残量警告 */
    TDE_POWERFAIL    = 0x33,     /* 電源異常 */
    TDE_POWERSUS     = 0x34,     /* 自動サスペンド */
    TDE_POWERUPTM    = 0x35,     /* 時計更新 */
    TDE_CKPWON       = 0x41      /* 自動電源オン通知 */
} TDEvtTyp;
```

デバイス事象通知の形式は以下のようになる。事象通知の内容は事象タイプごとに異なり、サイズも異なる。

```
typedef struct t_devevt {
    TDEvtTyp         evttyp; /* 事象タイプ */
    /* 以下に事象タイプ別の情報が付加される */
} T_DEVEVT;
```

デバイス ID 付のデバイス事象通知の形式は次のようになる。

```
typedef struct t_devevt_id {
    TDEvtTyp        evttyp;         /* 事象タイプ */
    ID              devid;          /* デバイス ID */
    /* 以下に事象タイプ別の情報が付加される */
} T_DEVEVT_ID;
```

事象の詳細は、各デバイスドライバの仕様を参照のこと。

事象通知用メッセージバッファが一杯で事象通知を送信できない場合は、その事象が通知されないことで事象通知の受信側の動作に悪影響が出ないようにしなければならない。メッセージバッファが空くまで待ってから事象通知を行ってもよいが、その場合も原則として事象通知以外のデバイスドライバの処理が滞ってはならない。なお、事象通知の受信側は、できる限りメッセージバッファが溢れることがないように処理しなければならない。

5.3.7 各デバイスのサスペンド/リジューム処理

イベント関数(eventfn)へのサスペンド/リジューム(TDV_SUSPEND/TDV_RESUME)イベントの発行により、各デバイスドライバはデバイスのサスペンド/リジューム処理を行う。サスペンド/リジュームイベントは、物理デバイスに対してのみ発行される。

サスペンド (TDV_SUSPEND)

```
evttyp = TDV_SUSPEND
evtinf = NULL (なし)
```

次のような手順でサスペンド処理を行う。

1. 現在処理中の要求があれば、完了するまで待つか、中断または中止する。どの方法を選択するかはデバイスドライバの実装に依存する。ただし、できるだけ速やかにサスペンドする必要があるため、完了までに時間がかかる場合は中断または中止としなければならない。
   サスペンドイベントは物理デバイスに対してのみ発行されるが､そのデバイスに含まれるすべての論理デバイスに対しても、同様に処理する。
   中断：処理を一時的に中断し、リジューム後に続きを行う。
   中止：中止処理関数(abortfn)による中止と同様に、処理を中止する。リジューム後も再開されない。
2. リジュームイベント以外の新たな要求を受け付けないようにする。
3. デバイスの電源を切るなどのサスペンド処理を行う。

中止はアプリケーションへの影響が大きいと考えられるため極力避けたい。シリアル回線からの長期の入力待ちなどで､かつ中断とするのが難しい場合以外は中止としない｡通常は､終了まで待つか､可能であれば中断とする。
サスペンド期間中にデバイスドライバへ来た要求は､リジュームまで待たせてリジューム後に受け付け処理する。ただし、デバイスへのアクセスを伴わない処理など、サスペンド中でも可能な処理は受け付けてもよい。

リジューム (TDV_RESUME)

```
evttyp = TDV_RESUME
evtinf = NULL (なし)
```

次のような手順でリジューム処理を行う。

1. デバイスの電源を入れデバイスの状態を復帰させるなどのリジューム処理を行う。
2. 中断していた処理があれば再開する。
3. 要求受け付けを再開する。

5.3.8 ディスクデバイスの特殊性

仮想記憶システムにおいて、ディスクデバイスは特別なものとなる。仮想記憶を行うにあたり、メモリとディスク間でのデータ転送を行うため、OS がディスクドライバを呼び出す必要がある。

OS がディスクとのデータ転送を行う必要があるのは、非常駐メモリにアクセスされたことにより、そのメモリの内容をディスクから読み出す(ページイン)ような場合である。このような場合に、OS はディスクドライバを呼び出す。

もし、ディスクドライバ内で非常駐メモリにアクセスした場合、OS はやはりディスクドライバを呼び出さなければならなくなる。このようなケースでは、非常駐メモリにアクセスしたことでページイン待ちとされているディスクドライバに対して、OS が再びディスクアクセスの要求を出す可能性がある。このような場合でも、ディスクドライバは後から来た OS からの要求を実行できなければならない。

同じ様なことが､サスペンド処理時にも起きる可能性がある｡サスペンド処理中に非常駐メモリがアクセスされ、ディスクドライバが呼び出されたとき、そのディスクドライバがすでにサスペンドされているとページインができないことになる。このようなケースを避けるため、サスペンド時にはディスクデバイス以外を先にサスペンドした後、ディスクデバイスをサスペンドする。しかし、複数のディスクデバイスがあれば、そのディスクデバイスのサスペンド順序は不定である。したがって、ディスクドライバでは、サスペンド処理中に非常駐メモリにアクセスしてはならない。

このような制限から、ディスクドライバでは非常駐メモリを使用しない(アクセスしない)ようにしなければならない。ただし、tk_rea_dev(), tk_wri_dev()で指定される入出力バッファ(buf)に関しては、呼び出し側で指定したメモリ領域であるため非常駐メモリである可能性がある。そのため、入出力バッファに関しては、入出力を行う間は常駐化(LockSpace 参照)しておくなどの対応が必要である。

5.4 割込み管理機能

割込み関係はハードウェア依存度が高く、システムごとに異なっているため共通化することが難しい。下記を標準仕様として定めるが、システムによってはこの通りに実装することが難しい場合がある。できる限り標準仕様に合わせた実装を求めるが、実装不可能なものは実装しなくてもよい。標準仕様とは別の機能を追加することも許されるが、その場合は関数名などは標準仕様と異なるものでなければならない。ただし、DI(), EI(), isDI()は標準仕様にしたがって、必ず実装しなければならない。

割込み管理機能は、ライブラリ関数またはC言語のマクロで提供され、これらはタスク独立部およびディスパッチ禁止・割込み禁止の状態から呼び出すことができる。

5.4.1 CPU 割込み制御

CPU の外部割込みフラグを制御する。一般的には、割込みコントローラに対しては何もしない。
DI(), EI(), isDI()は、C言語のマクロである。

```
DI ( UINT intsts )
```

intsts　　InterruptStatus　　CPU 割込みの状態(内容は実装定義)　※ポインタではない
すべての外部割込みを禁止する。割込みを禁止する前の状態を intsts に保存する。

```
EI ( UINT intsts )
```

intsts　　InterruptStatus　　CPU 割込みの状態(内容は実装定義)

すべての外部割込みを許可する。正確には、intsts の状態に戻す。つまり、DI()で割込み禁止する前の状態に戻す。したがって、DI()で割込みを禁止する以前から割込み禁止状態であれば、EI()を実行しても割込みは許可されない。ただし、intsts として0を指定した場合は、必ず割込みを許可する。

intsts は、DI()で保存した値または0のいずれかでなければならない。それ以外の値を指定した場合の動作は保証されない。

```
BOOL isDI ( UINT intsts )
```

intsts　　InterruptStatus　　CPU 割込みの状態(内容は実装定義)
戻値 TRUE(0以外の値):割込み禁止状態　FALSE:割込み許可状態

intsts に保存されている外部割込み禁止状態を調べる。T-Kernel/OS で割込み禁止と判断される状態を割込み禁止状態とする。

intsts は、DI()で保存した値でなければならない。それ以外の値を指定した場合の動作は保証されない。

(使用例)

```
void foo()
{
    UINTintsts;

    DI(intsts);

    if ( isDI(intsts) ) {
    /* この関数が呼び出された時点で既に割込み禁止であった */
    } else {
    /* この関数が呼び出された時点では割込み許可であった */
    }

    EI(intsts);
}
```

5.4.2 割込みコントローラ制御

割込みコントローラを制御する。一般的には、CPU の割込みフラグに対しては何もしない。

```
typedef UINT    INTVEC;     /* 割込みベクタ */
```

割込みベクタ(INTVEC)の具体的な内容は、実装定義である。ただし、tk_def_int()で指定する割込みハンドラ番号と同じか、簡単な方法で割込みハンドラ番号と相互に変換できる様なものであることが望ましい。

```
  UINT DINTNO( INTVEC intvec )
```

割込みベクタから割込みハンドラ番号へ変換する。

```
  void EnableInt( INTVEC intvec )
```

```
  void EnableInt( INTVEC intvec, INT level )
```

intvec の割込みを許可する。割込み優先度レベルを指定可能なシステムでは、level により割込み優先度レベルを指定する。level の具体的な意味は実装定義である。

level あり、または level なしの、いずれか一方を提供する。

```
  void DisableInt( INTVEC intvec )
```

intvec の割込みを禁止する。一般的には、割込み禁止中の割込みはペンディングされ、EnableInt()により許可した時に割込みが発生する。割込み禁止中に発生した割込みを無効にしたい場合は、ClearInt()を行う必要がある。

```
  void ClearInt( INTVEC intvec )
```

intvec の割込みが発生していればクリアする。

```
  void EndOfInt ( INTVEC intvec )
```

割込みコントローラに EOI(End Of Interrupt)を発行する。intvec は EOI 発行対象の割込みでなければならない。一般的には、割込みハンドラの最後で実行する必要がある。

```
  BOOL CheckInt ( INTVEC intvec )
```

intvec の割込みが発生しているか調べる。intvec の割込みが発生していれば TRUE(0 以外の値)、発生していなければ FALSE を返す。

5.5 I/O ポートアクセスサポート機能

I/O ポートアクセスサポート機能は、ライブラリ関数または C 言語のマクロで提供され、これらはタスク独立部およびディスパッチ禁止・割込み禁止の状態から呼び出すことができる。

5.5.1 I/O ポートアクセス

I/O 空間とメモリ空間が独立しているシステムでは、I/O ポートアクセス関数は I/O 空間のアクセスとなる。メモリマップド I/O のみのシステムでは、I/O ポートアクセス関数はメモリ空間のアクセスとなる。メモリマップド I/O のシステムにおいても、これらの関数を利用することにより、ソフトウェアの移植性や可読性が向上する。

～_w ワード(32bit)単位
～_h ハーフワード(16bit)単位
～_b バイト(8bit)単位

```
void out_w ( INT port, UW data )
```

```
void out_h ( INT port, UH data )
```

```
void out_b ( INT port, UB data )
```

port　　I/O ポートアドレス
data　　書き込むデータ

I/O ポートに書き込む。

```
UW in_w ( INT port )
```

```
UH in_h ( INT port )
```

```
UB in_b ( INT port )
```

port　　I/O ポートアドレス
戻値　　読み込んだデータ

I/O ポートから読み込む。

5.5.2 微小待ち

```
void WaitUsec ( UINT usec )
```

```
void WaitNsec ( UINT nsec )
```

usec　　MicroSeconds　　待ち時間(マイクロ秒)
nsec　　NanoSeconds　　待ち時間(ナノ秒)

指定された時間分の微小待ちを行う。
これらの待ちは通常ビジーループで行われる。そのため、RAM 上で実行する場合、ROM 上で実行する場合、メモリキャッシュがオンの場合、オフの場合など、実行環境の影響を受けやすい。したがって、これらの待ち時間はあまり正確ではない。
これらの待ちは OS の待ち状態ではない。実行状態のままである。

5.6 省電力機能

T-Kernel/OS から呼び出される機能である。tk_set_pow()参照。

これらの関数の呼び出し方法は実装定義である。単純な関数呼び出しでもよいし、トラップを使用してもよい。ただし、拡張 SVC などの OS の機能を利用するものは使用できない。また、T-Monitor にこれらの機能を用意してもよい。

なお、ここに示した low_pow(), off_pow()は参考仕様である。これらは、T-Kernel 内部でのみ使用するものであるため、独自仕様としても構わない。より高度な省電力機能を実現するために、まったく別の仕様としても構わない。ただし、同等程度の機能であるなら、参考仕様に合わせる方が望ましい。

```
void low_pow ( void )
```

低消費電力モードに切り替え、割込み発生を待つ。

タスクディスパッチャ内から呼び出される。次のような処理を行う。

1. 低消費電力モードへ移行する。
2. 外部割込みの発生を待つ。
3. 外部割込みが発生したら、通常モードに復帰して呼び出し元へ戻る。

割込み禁止状態で呼び出される。割込みを許可してはいけない。処理速度が割込み応答速度に影響する。できる限り高速であることが要求される。

```
void off_pow ( void )
```

システムをサスペンド状態に移行する。リジューム要因の発生を待ちリジュームする。

tk_set_pow()から呼び出される。次のような処理を行う。

1. ハードウェアをサスペンド状態へ移行させる。
2. リジューム要因の発生を待つ。
3. リジューム要因が発生したら、サスペンド状態から復帰して呼び出し元へ戻る。

割込み禁止状態で呼び出される。割込みを許可してはいけない。周辺機器など各デバイスは、デバイスドライバによってサスペンド状態への移行および復帰を行う。

5.7 システム構成情報管理機能

システム構成に関する情報(最大タスク数など)およびその他の任意の情報を保持・管理し、提供する。システム実行時に情報を追加・変更する機能はない。

システム構成情報をどの様に保持するかについては規定しないが、一般的にはメモリ(ROM/RAM)に保持する。したがって、あまり大量の情報を保持する目的には使用しない。

システム構成情報は、一部標準定義として定めるが、それ以外にもアプリケーションやサブシステム、デバイスドライバで任意に定義して利用できる。

システム構成情報の形式は、名称と定義データの組である。

- 名称
  最大 16 文字の文字列で表される。
  使用可能な文字(UB) a～z A～Z 0～9 _
- 定義データ
  数値(整数)の並びまたは文字列により定義される。
  使用可能な文字(UB) 0x00～0x1F, 0x7F, 0xFF(文字コード)を除く文字

(例)

名称	定義データ
SysVer	1 0
SysName	T-Kernel Version 1.00

5.7.1 システム構成情報の取得

システム構成情報取得機能は、拡張 SVC により呼び出す。T-Kernel 内部で使用する他、アプリケーションやサブシステム、デバイスドライバなどからも利用可能である。なお、T-Kernel 内部での利用は拡張 SVC を経由しない方法でもよく、実装定義とする。

```
INT tk_get_cfn ( UB *name, INT *val, INT max )
```

name	Name	名称
val	Value	数値列を格納する配列
max	MaximumCount	val の配列の要素数
戻値	定義されている数値情報の個数 またはエラー	

システム構成情報から数値列の情報を取得する。name の名称で定義されている数値列の情報を、最大 max 個まで取得し val へ格納する。戻値に定義されている数値列情報の数を返す。戻値>max であれば、すべての情報を格納しきれていないことを示す。max=0 を指定することで、val に格納せずに数値列の数を知ることができる。

name の名称の情報が定義されていないときは E_NOEXS を返す。name の名称で定義されている情報が文字列であった場合の動作は不定である。

この機能は、T-Kernel/OS のシステムコールの呼び出し可能な保護レベルの制限をうけずに任意の保護レベルから呼び出すことができる。

```
INT tk_get_cfs ( UB *name, UB *buf, INT max )
```

name	Name	名称
buf	Buffer	文字列を格納する配列
max	MaximumLength	buf の最大長(バイト数)
戻値	定義されている文字列情報の長さ(バイト数) またはエラー	

システム構成情報から文字列の情報を取得する。name の名称で定義されている文字列の情報を、最大 max 文字まで取得し、buf へ格納する。格納した文字列が max 文字未満であれば、最後に'¥0'を格納する。戻値に定義されている文字列情報の長さ('¥0'は含まない)を返す。戻値>max であれば、すべての情報を格納しきれていないことを示す。max=0 を指定することで、buf に格納せずに文字列の長さを知ることができる。

name の名称で情報が定義されていないときは E_NOEXS を返す。name の名称で定義されている情報が数値列であった場合の動作は不定である。

この機能は、T-Kernel/OS のシステムコールの呼び出し可能な保護レベルの制限をうけずに任意の保護レベルから呼び出すことができる。

5.7.2 標準システム構成情報

下記の定義情報を標準として定める。標準定義の名称はＴを前置する。
N:　数値列情報
S:　文字列情報

・製品情報
S: TSysName システム名称(製品名称)

・各オブジェクトの最大数
N: TmaxTskId　　最大タスク数
N: TmaxSemId　　最大セマフォ数
N: TmaxFlgId　　最大イベントフラグ数
N: TmaxMbxId　　最大メールボックス数
N: TmaxMtxId　　最大ミューテックス数
N: TmaxMbfId　　最大メッセージバッファ数
N: TmaxPorId　　最大ランデブポート数
N: TmaxMpfId　　最大固定長メモリプール数
N: TmaxMplId　　最大可変長メモリプール数
N: TmaxCycId　　最大周期起動ハンドラ数
N: TmaxAlmId　　最大アラームハンドラ数
N: TmaxResId　　最大リソースグループ数
N: TmaxSsyId　　最大サブシステム数
N: TmaxSsyPri　　最大サブシステム優先度数

・その他
N: TsysStkSz　　デフォルトシステムスタックサイズ(バイト数)
N: TSVCLimit　　システムコールの呼び出し可能な最も低い保護レベル
N: TTimPeriod　　タイマー割り込み間隔(ミリ秒)

・デバイス管理
N: TMaxRegDev　　最大デバイス登録数
N: TMaxOpnDev　　最大デバイスオープン数
N: TMaxReqDev　　最大デバイスリクエスト数
N: TDEvtMbfSz　　事象通知用メッセージバッファのサイズ(バイト数)
事象通知のメッセージ最大長(バイト数)
TDEvtMbfSz が定義されなかった場合、またはメッセージバッファのサイズに負数が設定された場合は、事象通知用メッセージバッファを使用しない。

上記の数値列情報で複数の数値が定義されるものは、説明の順序と同じ順序で配列に格納される。

(例)　　tk_get_cfn("TDEvtMbfSz", val, 2)
val[0] = 事象通知用メッセージバッファのサイズ
val[1] = 事象通知のメッセージ最大長

5.8 サブシステムおよびデバイスドライバの起動

サブシステムおよびデバイスドライバは次のようなエントリーを持つ。

```
ER main( INT ac, UB *av[] )
{
	if ( ac >= 0 ) {
		/* サブシステム/デバイスドライバ起動処理 */
	} else {
		/* サブシステム/デバイスドライバ終了処理 */
	}
	return ercd;
}
```

このエントリールーチンは、サブシステムおよびデバイスドライバの起動処理または終了処理を行うだけで、実際のサービスの提供は行わない。起動処理または終了処理が済んだら直ちに呼び出し元に戻らなければならない。エントリールーチンは、できるだけ速やかに実行し、呼び出し元に戻らなければならない。

エントリールーチンは、通常はシステムの起動時および終了時にシステムリソースグループに属するタスクから呼び出され、OS の起動処理タスクまたは終了処理タスクのコンテキスト(保護レベル 0)で実行される。なお、OS の実装によっては、準タスク部として実行される場合がある。また、サブシステムやデバイスドライバの動的なロードをサポートしているシステムにおいては、システムの起動時および終了時以外に呼び出される場合もある。

複数のサブシステムおよびデバイスドライバがあるとき、システムの起動時および終了時のエントリールーチンの呼び出しは、一つずつ順番に行われる。複数のエントリールーチンが異なるタスクから同時に呼び出されることはない。したがって、サブシステムまたはデバイスドライバ間で初期化順序に依存関係があるような場合、エントリールーチンから戻る前に必要な処理をすべて終わらせておくことで、初期化の順序関係を維持することができる。

エントリールーチンの関数名は通常は main であるが、OS とリンクする場合など main が使用できない場合は任意の名称としてよい。

エントリールーチンを OS に指示する(登録する)方法、パラメータの指定方法、およびエントリールーチンの呼び出し順の指定方法は、OS の実装ごとに定義される。

・起動処理

ac　　パラメータの数(≧0)
av　　パラメータ(文字列)
戻値　エラーコード

ac≧0 の場合、起動処理となる。サブシステムおよびデバイスドライバの初期化処理を行った後、サブシステムおよびデバイスドライバを登録する。

戻値として負の値(エラー)を返した場合は起動処理の失敗とみなされる。OS の実装によっては、このサブシステムまたはデバイスドライバをメモリから削除してしまう場合があるので、サブシステムおよびデバイスドライバの登録を行った状態でエラーを戻してはいけない。エラーを戻すときは、必ず登録を抹消しておかなければならない。また、獲得したリソースも解放しておく必要がある。自動的には解放されない。

ac, av は C 言語の標準的な main() の引数と同じである。ac にパラメータ数、av は ac+1 個分のポインタの配列で、文字列のパラメータを指している。配列の終端(av[ac])は NULL である。

av[0]は、サブシステムまたはデバイスドライバの名称である。一般的には、サブシステムまたはデバイスドライバが格納されたファイル名などになるが、どのような名称が格納されるかは実装定義である。名称なし(空文字列"")であってもよい。

av[1]以降のパラメータは、各サブシステムおよびデバイスドライバごとに定義される。

av の指す文字列の領域は、エントリールーチンから抜けると消滅するため、必要に応じてパラメータは別の場所に保存しなければならない。

・終了処理

ac　　-1
av　　NULL
戻値　エラーコード

ac＜0 の場合、終了処理となる。サブシステムおよびデバイスドライバの登録を抹消した後、獲得したリソースなどを解放する。終了処理中にエラーが発生した場合も、終了処理を中止してはいけない。可能な限り処理する。一部が正常に終了処理できなかった場合は、戻値にエラーを返す。

サブシステムおよびデバイスドライバに対する要求を処理中の状態で、終了処理が呼び出された場合の動作は、サブシステムおよびデバイスドライバの実装に依存する。一般的には、終了処理はシステム終了時に呼び出されることを前提とするため、処理中の要求がないのが普通である。そのため、処理中の要求があった場合の動作は保証されないのが普通である。

第 6 章 T-Kernel/DS の機能

この章では、T-Kernel/DebuggerSupport(T-Kernel/DS)で提供している機能の詳細について説明を行う。
T-Kernel/DS は、デバッガが T-Kernel の内部状態の参照や実行のトレースを行うための機能を提供するものである。T-Kernel/DS の提供する機能は、デバッガ専用であり、一般のアプリケーション等からは使用しない。

全般的な注意・補足事項

- T-Kernel/DS のシステムコール(td_～)は、特に明記されているものを除き、タスク独立部およびディスパッチ禁止中・割込み禁止中からも呼び出すことができる。
  ただし、実装によっては機能が制限される場合がある。
- T-Kernel/DS のシステムコール(td_～)を割込み禁止状態で呼出した場合、割込み許可されることなく処理される。同様に、OS のその他の状態も変化させない。割込み許可状態やディスパッチ許可状態で呼び出された場合は、OS の動作も継続されるため、OS の状態は変化する場合がある。
- T-Kernel/DS のシステムコール(td_～)は、T-Kernel/OS のシステムコールの呼出し可能な保護レベルより低い保護レベル(TSVCLimit より低い保護レベル)から呼び出すことはできない(E_OACV)。
- 常に発生する可能性のあるエラーE_PAR, E_MACV, E_CTX などは、特に説明を必要とする場合以外は省略している。
- E_PAR, E_MACV, E_CTX の検出は実装依存でありエラーとして検出されない場合もあるため、このようなエラーを発生する可能性のある呼出を行ってはいけない。

6.1 カーネル内部状態取得機能

この機能は、デバッガがカーネルの内部状態を取得するための機能である。オブジェクトの一覧を取得する機能、タスクの優先順位を取得する機能、待ち行列に並んだタスクの並び順を取得する機能、オブジェクトやシステムやタスクレジスタの状態を取得する機能、および時刻を取得する機能が含まれる。

td_lst_tsk, td_lst_sem, td_lst_flg, td_lst_mbx,
td_lst_mtx, td_lst_mbf, td_lst_por, td_lst_mpf,
td_lst_mpl, td_lst_cyc, td_lst_alm, td_lst_ssy

各オブジェクトの ID のリスト参照

【C 言語インタフェース】

```
INT ct = td_lst_tsk ( ID list[], INT nent ) ;  /* タスク */
INT ct = td_lst_sem ( ID list[], INT nent ) ;  /* セマフォ */
INT ct = td_lst_flg ( ID list[], INT nent ) ;  /* イベントフラグ */
INT ct = td_lst_mbx ( ID list[], INT nent ) ;  /* メールボックス */
INT ct = td_lst_mtx ( ID list[], INT nent ) ;  /* ミューテックス */
INT ct = td_lst_mbf ( ID list[], INT nent ) ;  /* メッセージバッファ */
INT ct = td_lst_por ( ID list[], INT nent ) ;  /* ランデブポート */
INT ct = td_lst_mpf ( ID list[], INT nent ) ;  /* 固定長メモリプール */
INT ct = td_lst_mpl ( ID list[], INT nent ) ;  /* 可変長メモリプール */
INT ct = td_lst_cyc ( ID list[], INT nent ) ;  /* 周期ハンドラ */
INT ct = td_lst_alm ( ID list[], INT nent ) ;  /* アラームハンドラ */
INT ct = td_lst_ssy ( ID list[], INT nent ) ;  /* サブシステム */
```

【パラメータ】

ID	list[]	List	オブジェクト ID を格納する領域
INT	nent	NumberOfListEntries	list に格納可能な最大数

【リターンパラメータ】

INT	ct	Count	使用されているオブジェクトの個数
	または	ErrorCode	エラーコード

【解説】

現在使用されているオブジェクトの ID のリストを取得し、list へ最大 nent 個分の ID を格納する。戻値に、使用されているオブジェクトの個数を返す。戻値>nent であれば、すべての ID は取得できていないことを示す。

td_rdy_que

タスクの優先順位の参照

【C 言語インタフェース】

```
INT ct = td_rdy_que ( PRI pri, ID list[], INT nent ) ;
```

【パラメータ】

PRI	pri	TaskPriority	対象タスク優先度
ID	list[]	TaskIDList	タスク ID を格納する領域
INT	nent	NumberOfListEntries	list に格納可能な最大数

【リターンパラメータ】

INT	ct	Count	優先度 pri の実行できる状態のタスクの個数
	または	ErrorCode	エラーコード

【解説】

実行できる状態(実行可能状態および実行状態)のタスクのうち、優先度が pri であるものを、優先順位の高い順に一列に並べ、その並び順にタスク ID のリストを取得する。

list には、優先順位が最高のものを先頭としてその並び順に、タスク ID が最大 nent 個まで格納される。

戻値には、pri の優先度を持つ実行できる状態のタスクの個数を返す。戻値>nent であれば、すべてのタスク ID は取得できていないことを示す。

td_sem_que, td_flg_que, td_mbx_que, td_mtx_que, td_smbf_que, td_rmbf_que, td_cal_que, td_acp_que, td_mpf_que, td_mpl_que
待ち行列の参照

【C 言語インタフェース】
```
INT ct = td_sem_que ( ID semid, ID list[], INT nent ) ; /* セマフォ */
INT ct = td_flg_que ( ID flgid, ID list[], INT nent ) ; /* イベントフラグ */
INT ct = td_mbx_que ( ID mbxid, ID list[], INT nent ) ; /* メールボックス */
INT ct = td_mtx_que ( ID mtxid, ID list[], INT nent ) ; /* ミューテックス */
INT ct = td_smbf_que ( ID mbfid, ID list[], INT nent ) ; /* メッセージバッファ送信 */
INT ct = td_rmbf_que ( ID mbfid, ID list[], INT nent ) ; /* メッセージバッファ受信 */
INT ct = td_cal_que ( ID porid, ID list[], INT nent ) ; /* ランデブ呼出 */
INT ct = td_acp_que ( ID porid, ID list[], INT nent ) ; /* ランデブ受付 */
INT ct = td_mpf_que ( ID mpfid, ID list[], INT nent ) ; /* 固定長メモリプール */
INT ct = td_mpl_que ( ID mplid, ID list[], INT nent ) ; /* 可変長メモリプール */
```

【パラメータ】

ID	～id	ObjectID	対象オブジェクト ID
ID	list[]	TaskIDList	待ちタスクのタスク ID を格納する領域
INT	nent	NumberOfListEntries	list に格納可能な最大数

【リターンパラメータ】

INT	ct	Count	待ちタスクの個数
	または	ErrorCode	エラーコード

【エラーコード】

E_ID	ID 番号が不正
E_NOEXS	対象オブジェクトが存在しない

【解説】

～id で指定したオブジェクトの待ち行列に並んでいるタスクの ID のリストを取得する。list には、待ち行列の先頭からその並び順にタスク ID が、最大 nent 個まで格納される。戻値には、待ち行列に並んでいるタスクの個数を返す。戻値>nent であれば、すべてのタスク ID は取得できていないことを示す。

タスクの状態参照 td_ref_tsk

【C 言語インタフェース】
```
ER ercd = td_ref_tsk ( ID tskid, TD_RTSK *rtsk );
```

【パラメータ】

ID	tskid	TaskID	対象タスク ID(TSK_SELF 可)
TD_RTSK*	rtsk	Packet to Refer Task State	タスク状態を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	ID 番号が不正
E_NOEXS	対象オブジェクトが存在しない

【解説】

タスクの状態を参照する。tk_ref_tsk()と同等だが、タスク起動アドレスおよびスタックに関する情報が追加されている。

```
typedef struct td_rtsk {
    VP      exinf;      /* 拡張情報 */
    PRI     tskpri;     /* 現在の優先度 */
    PRI     tskbpri;    /* ベース優先度 */
    UINT    tskstat;    /* タスク状態 */
    UINT    tskwait;    /* 待ち要因 */
    ID      wid;        /* 待ちオブジェクト ID */
    INT     wupcnt;     /* 起床要求キューイング数 */
    INT     suscnt;     /* SUSPEND 要求ネスト数 */
    RELTIM  slicetime;  /* 最大連続実行時間（ミリ秒）*/
    UINT    waitmask;   /* 待ちを禁止されている待ち要因 */
    UINT    texmask;    /* 許可されているタスク例外 */
    UINT    tskevent;   /* 発生しているタスクイベント */
    FP      task;       /* タスク起動アドレス */
    INT     stksz;      /* ユーザスタックサイズ(バイト）*/
    INT     sstksz;     /* システムスタックサイズ(バイト）*/
    VP      istack;     /* ユーザスタックポインタ初期値 */
    VP      isstack;    /* システムスタックポインタ初期値 */
} TD_RTSK;
```

スタック領域は、スタックポインタ初期値の位置から低位アドレス(値の小さい方)へ向かって、スタックサイズ分となる。

```
istack  - stksz  ≦ ユーザスタック領域   < istack
isstack - sstksz ≦ システムスタック領域 < isstack
```

なお、スタックポインタ初期値(istack, isstack)は、スタックポインタの現在位置ではない。タスク起動前の状態であっても、スタック領域は使用されている場合がある。スタックポインタの現在位置を得るには、td_get_reg()を用いる。

td_ref_sem , td_ref_flg, td_ref_mbx, td_ref_mtx,
td_ref_mbf, td_ref_por, td_ref_mpf, td_ref_mpl,
td_ref_cyc, td_ref_alm, td_ref_ssy

各オブジェクトの状態参照

【C 言語インタフェース】

```
ER ercd = td_ref_sem ( ID semid, TD_RSEM *rsem );  /* セマフォ */
ER ercd = td_ref_flg ( ID flgid, TD_RFLG *rflg );  /* イベントフラグ */
ER ercd = td_ref_mbx ( ID mbxid, TD_RMBX *rmbx );  /* メールボックス */
ER ercd = td_ref_mtx ( ID mtxid, TD_RMTX *rmtx );  /* ミューテックス */
ER ercd = td_ref_mbf ( ID mbfid, TD_RMBF *rmbf );  /* メッセージバッファ */
ER ercd = td_ref_por ( ID porid, TD_RPOR *rpor );  /* ランデブポート */
ER ercd = td_ref_mpf ( ID mpfid, TD_RMPF *rmpf );  /* 固定長メモリプール */
ER ercd = td_ref_mpl ( ID mplid, TD_RMPL *rmpl );  /* 可変長メモリプール */
ER ercd = td_ref_cyc ( ID cycid, TD_RCYC *rcyc );  /* 周期ハンドラ */
ER ercd = td_ref_alm ( ID almid, TD_RALM *ralm );  /* アラームハンドラ */
ER ercd = td_ref_ssy ( ID  ssid, TD_RSSY *rssy );  /* サブシステム */
```

【パラメータ】

ID	～id	ObjectID	対象オブジェクト ID
TD_R～*	r～	Packet to Refer Status	各オブジェクトの状態を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	ID 番号が不正
E_NOEXS	対象オブジェクトが存在しない

【解説】

各オブジェクトの状態を参照する。tk_ref_～() と同等。
リターンパケットの定義を以下に示す。

```
/*
 * セマフォ状態情報 td_ref_sem
 */
typedef struct td_rsem {
    VP  exinf;  /* 拡張情報 */
    ID  wtsk;   /* 待ちタスクの ID */
    INT semcnt; /* 現在のセマフォカウント値 */
} TD_RSEM;

/*
 * イベントフラグ状態情報    td_ref_flg
 */
typedef struct td_rflg {
    VP   exinf;  /* 拡張情報 */
    ID   wtsk;   /* 待ちタスクの ID */
    UINT flgptn; /* 現在のイベントフラグパターン */
} TD_RFLG;

/*
 * メールボックス状態情報    td_ref_mbx
 */
typedef struct td_rmbx {
    VP    exinf;       /* 拡張情報 */
    ID    wtsk;        /* 待ちタスクの ID */
    T_MSG *pk_msg;     /* 次に受信されるメッセージ */
} TD_RMBX;

/*
 * ミューテックス状態情報    td_ref_mtx
 */
typedef struct td_rmtx {
    VP  exinf;  /* 拡張情報 */
    ID  htsk;   /* ロックしているタスクの ID */
    ID  wtsk;   /* ロック待ちタスクの ID */
} TD_RMTX;

/*
 * メッセージバッファ状態情報    td_ref_mbf
 */
typedef struct td_rmbf {
    VP  exinf;  /* 拡張情報 */
    ID  wtsk;   /* 受信待ちタスクの ID */
    ID  stsk;   /* 送信待ちタスクの ID */
    INT msgsz;  /* 次に受信されるメッセージのサイズ(バイト) */
    INT frbufsz;/* 空きバッファのサイズ(バイト) */
    INT maxmsz; /* メッセージの最大長(バイト) */
} TD_RMBF;
```

```
/*
 * ランデブポート状態情報    td_ref_por
 */
typedef struct td_rpor {
    VP  exinf;  /* 拡張情報 */
    ID  wtsk;   /* 呼出待ちタスクの ID */
    ID  atsk;   /* 受付待ちタスクの ID */
    INT maxcmsz;/* 呼出メッセージの最大長(バイト) */
    INT maxrmsz;/* 返答メッセージの最大長(バイト) */
} TD_RPOR;

/*
 * 固定長メモリプール状態情報    td_ref_mpf
 */
typedef struct td_rmpf {
    VP  exinf;  /* 拡張情報 */
    ID  wtsk;   /* 待ちタスクの ID */
    INT frbcnt; /* 空き領域のブロック数 */
} TD_RMPF;

/*
 * 可変長メモリプール状態情報    td_ref_mpl
 */
typedef struct td_rmpl {
    VP  exinf;  /* 拡張情報 */
    ID  wtsk;   /* 待ちタスクの ID */
    INT frsz;   /* 空き領域の合計サイズ(バイト) */
    INT maxsz;  /* 最大の連続空き領域のサイズ(バイト) */
} TD_RMPL;

/*
 * 周期ハンドラ状態情報 td_ref_cyc
 */
typedef struct td_rcyc {
    VP  exinf;      /* 拡張情報 */
    RELTIM  lfttim; /* 次のハンドラ起動までの残り時間 */
    UINT cycstat;   /* 周期ハンドラ状態 */
} TD_RCYC;

/*
 * アラームハンドラ状態情報 td_ref_alm
 */
typedef struct td_ralm {
    VP  exinf;      /* 拡張情報 */
    RELTIM  lfttim; /* ハンドラ起動までの残り時間 */
    UINT almstat;   /* アラームハンドラ状態 */
} TD_RALM;

/*
 * サブシステム状態情報 td_ref_ssy
 */
typedef struct td_rssy {
    PRI ssypri;     /* サブシステム優先度 */
    INT resblksz;   /* リソース管理ブロックサイズ(バイト) */
} TD_RSSY;
```

タスク例外の状態参照 td_ref_tex

【C 言語インタフェース】
```
ER ercd = td_ref_tex ( ID tskid, TD_RTEX *pk_rtex );
```

【パラメータ】

ID	tskid	TaskID	対象タスク ID (TSK_SELF 可)
TD_RTEX*	pk_rtex	Packet to Refer Task Exception Status	タスク例外状態を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_rtex の内容

UINT	pendtex	発生しているタスク例外
UINT	texmask	許可されているタスク例外

【エラーコード】

E_OK	正常終了
E_ID	ID 番号が不正
E_NOEXS	対象オブジェクトが存在しない

【解説】
タスク例外の状態を参照する。tk_ref_tex()と同等。

タスク統計情報参照 td_inf_tsk

【C 言語インタフェース】
```
ER ercd = td_inf_tsk ( ID tskid, TD_ITSK *pk_itsk, BOOL clr );
```

【パラメータ】

ID	tskid	TaskID	対象タスク ID (TSK_SELF 可)
TD_ITSK*	pk_itsk	Packet to Refer Task Statistics	タスク統計情報を返す領域へのポインタ
BOOL	clr	Clear	タスク統計情報のクリアの有無

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_itsk の内容

RELTIM	stime	累積システムレベル実行時間(ミリ秒)
RELTIM	utime	累積ユーザレベル実行時間(ミリ秒)

【エラーコード】

E_OK	正常終了
E_ID	ID 番号が不正
E_NOEXS	対象オブジェクトが存在しない

【解説】

タスク統計情報を参照する。tk_inf_tsk()と同等。clr=TRUE(≠0)の場合は、統計情報を取り出した後、累積時間をリセット(0 クリア)する。

タスクレジスタの参照 td_get_reg

【C 言語インタフェース】

```
ER ercd = td_get_reg ( ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs ) ;
```

【パラメータ】

ID	tskid	TaskID	対象タスクの ID (TSK_SELF 不可)
T_REGS*	pk_regs	Packet of Registers	汎用レジスタを返す領域へのポインタ
T_EIT*	pk_eit	Packet of EIT Registers	例外時に保存されるレジスタを返す領域へのポインタ
T_CREGS*	pk_cregs	Packet of Control Registers	制御レジスタを返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

T_REGS, T_EIT, T_CREGS の内容は、CPU および実装ごとに定義する。

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正 (対象タスクが現在実行状態のタスク)

【解説】

タスクのレジスタを参照する。tk_get_reg と同等。

現在実行状態にあるタスクは参照することはできない。タスク独立部の実行中を除けば、現在実行状態のタスクは自タスクである。

regs, eit, cregs は、それぞれ NULL を指定すると、対応するレジスタは参照されない。

T_REGS, T_EIT, T_CREGS の内容は実装定義である。

タスクレジスタの設定 td_set_reg

【C 言語インタフェース】

```
ER ercd = td_set_reg ( ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs ) ;
```

【パラメータ】

ID	tskid	TaskID	対象タスクの ID (TSK_SELF 不可)
T_REGS*	pk_regs	Packet of Registers	汎用レジスタ
T_EIT*	pk_eit	Packet of EIT Registers	例外時に保存されるレジスタ
T_CREGS*	pk_cregs	Packet of Control Registers	制御レジスタ

T_REGS, T_EIT, T_CREGS の内容は、CPU および実装ごとに定義する。

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_ID	不正 ID 番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正 (対象タスクが現在実行状態のタスク)

【解説】

タスクのレジスタを設定する。tk_set_reg と同等。

現在実行状態にあるタスクに設定することはできない。タスク独立部の実行中を除けば、現在実行状態のタスクは自タスクである。

regs, eit, cregs は、それぞれ NULL を指定すると、対応するレジスタは設定されない。

T_REGS, T_EIT, T_CREGS の内容は実装定義である。

システム状態参照 td_ref_sys

【C 言語インタフェース】
```
ER ercd = td_ref_sys ( TD_RSYS *pk_rsys ) ;
```

【パラメータ】

TD_RSYS*	pk_rsys	Packet to Refer System Status	システム状態を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

pk_rsys の内容

INT	sysstat	システム状態
ID	runtskid	現在実行状態にあるタスクの ID
ID	schedtskid	実行状態にすべきタスクの ID

【エラーコード】

E_OK	正常終了

【解説】
システムの状態を参照する。tk_ref_sys()と同等。

システム時刻参照 td_get_tim

【C 言語インタフェース】
```
ER ercd = td_get_tim ( SYSTIM *tim, UINT *ofs ) ;
```

【パラメータ】

SYSTIM*	tim	Time	現在時刻(ミリ秒)を返す領域へのポインタ
UINT*	ofs	Offset	リターンパラメータ ofs を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード
SYSTIM	tim	Time	現在時刻(ミリ秒)
UINT	ofs	Offset	tim からの相対的な経過時間(ナノ秒)

tim の内容　現在時刻(ミリ秒)

W	hi	high 32bits	システムの現在時刻の上位 32 ビット
UW	lo	low 32bits	システムの現在時刻の下位 32 ビット

【エラーコード】

E_OK	正常終了

【解説】

現在時刻(1985 年 1 月 1 日 0 時(GMT)からの通算のミリ秒数)を取得する。tim に返される値は tk_get_tim()と同じである。tim は、タイマ割込み間隔(周期)の分解能となるが、さらに細かい精度の時刻情報として tim からの経過時間を ofs にナノ秒単位で取得する。ofs の分解能は実装依存であるが、一般にはタイマハードウェアの分解能となる。

tim は、タイマ割込みによってカウントされる時刻であるため、割込み禁止期間中にタイマ割込み周期が来た場合、タイマ割込みハンドラが起動されず(起動が遅らされ)時刻が更新されないことがある。このような場合、tim には前回のタイマ割込みによって更新された時刻が返され、ofs には前回のタイマ割込みからの経過時間を返す。したがって、ofs はタイマ割込み間隔より長い時間となる場合がある。ofs がどの程度まで長い経過時間を計測できるかはハードウェアなどに依存するが、少なくともタイマ割込み間隔の 2 倍未満(0 ≦ ofs ＜ タイマ割込み間隔の 2 倍)の範囲まで計測できることが望ましい。

なお、tim および ofs に返される時刻は、td_get_tim()を呼出してから戻るまでにかかった時間範囲の中のどこかの時点の時刻となる。td_get_tim()を呼出した時点の時刻でも、td_get_tim()から戻った時点の時刻でもない。したがって、より正確な情報を得たい場合は、割込み禁止状態で呼び出すべきである。

システム稼動時間参照 td_get_otm

【C 言語インタフェース】
```
ER ercd = td_get_otm ( SYSTIM *tim, UINT *ofs ) ;
```

【パラメータ】

SYSTIM*	tim	Time	稼働時間(ミリ秒)を返す領域へのポインタ
UINT*	ofs	Offset	リターンパラメータ ofs を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード
SYSTIM	tim	Time	稼働時間(ミリ秒)
UINT	ofs	Offset	tim からの相対的な経過時間(ナノ秒)

tim の内容　稼働時刻(ミリ秒)

W	hi	high 32bits	システム稼働時間の上位 32 ビット
UW	lo	low 32bits	システム稼働時間の下位 32 ビット

【エラーコード】

E_OK	正常終了

【解説】

システム稼働時間(システム起動時からの積算ミリ秒数)を取得する。tim に返される値は tk_get_otm()と同じである。tim は、タイマ割込み間隔(周期)の分解能となるが、さらに細かい精度の時刻情報として tim からの経過時間を ofs にナノ秒単位で取得する。ofs の分解能は実装依存であるが、一般にはタイマハードウェアの分解能となる。

tim は、タイマ割込みによってカウントされる時刻であるため、割込み禁止期間中にタイマ割込み周期が来た場合、タイマ割込みハンドラが起動されず(起動が遅らされ)時刻が更新されないことがある。このような場合、tim には前回のタイマ割込みによって更新された時刻が返され、ofs には前回のタイマ割込みからの経過時間を返す。したがって、ofs はタイマ割込み間隔より長い時間となる場合がある。ofs がどの程度まで長い経過時間を計測できるかはハードウェアなどに依存するが、少なくともタイマ割込み間隔の 2 倍未満(0 ≦ ofs ＜ タイマ割込み間隔の 2 倍)の範囲まで計測できることが望ましい。

なお、tim および ofs に返される時刻は、td_get_otm()を呼出してから戻るまでにかかった時間範囲の中のどこかの時点の時刻となる。td_get_otm()を呼出した時点の時刻でも、td_get_otm()から戻った時点の時刻でもない。したがって、より正確な情報を得たい場合は、割込み禁止状態で呼び出すべきである。

DS オブジェクト名称の参照 td_ref_dsname

【C 言語インタフェース】
```
ER ercd = td_ref_dsname( UINT type, ID id, UB *dsname );
```

【パラメータ】

UINT	type	ObjectType	対象オブジェクトタイプ
ID	id	ObjectID	対象オブジェクト ID
UB*	dsname	DS Object Name	DS オブジェクト名称を返す領域へのポインタ

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

dsname の内容
　オブジェクト生成時、または td_set_dsname()で設定された DS オブジェクト名称

【エラーコード】

E_OK	正常終了
E_PAR	オブジェクトタイプ不正
E_NOEXS	対象オブジェクトが存在しない
E_OBJ	DS オブジェクト名称未使用

【解説】
　オブジェクト生成時に設定した DS オブジェクト名称(dsname)を参照する。対象となるオブジェクトは、オブジェクトタイプ(type)とオブジェクト ID(id)で指定する。
　指定可能なオブジェクトタイプ(type)は、以下の通りである。

```
TN_TSK 0x01     /* タスク */
TN_SEM 0x02     /* セマフォ */
TN_FLG 0x03     /* イベントフラグ */
TN_MBX 0x04     /* メールボックス */
TN_MBF 0x05     /* メッセージバッファ */
TN_POR 0x06     /* ランデブポート */
TN_MTX 0x07     /* ミューテックス */
TN_MPL 0x08     /* 可変長メモリプール */
TN_MPF 0x09     /* 固定長メモリプール */
TN_CYC 0x0a     /* 周期ハンドラ */
TN_ALM 0x0b     /* アラームハンドラ */
```

　DS オブジェクト名称は、オブジェクトの属性に、TA_DSNAME が指定された場合に有効となる。オブジェクト生成後に、td_set_dsname()で DS オブジェクト名称を再設定した場合は、この名称が参照される。
　DS オブジェクト名称は、

　　使用可能文字(UB)a～z, A～Z, 0～9
　　名称長　8byte（満たない場合は NULL で埋める）

とするが、文字コードのチェックは、T-Kernel では行わない。

DS オブジェクト名称の設定 td_set_dsname

【C 言語インタフェース】
```
ER ercd = td_set_dsname( UINT type, ID id, UB *dsname );
```

【パラメータ】

UINT	type	ObjectType	対象オブジェクトタイプ
ID	id	ObjectID	対象オブジェクト ID
UB*	dsname	DS Object Name	設定する DS オブジェクト名称

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【エラーコード】

E_OK	正常終了
E_PAR	オブジェクトタイプ不正
E_NOEXS	対象オブジェクトが存在しない
E_OBJ	DS オブジェクト名称未使用

【解説】

オブジェクト生成時に設定した DS オブジェクト名称(dsname)を再設定する。対象となるオブジェクトは、オブジェクトタイプ(type)とオブジェクト ID(id)で指定する。

指定可能なオブジェクトタイプ(type)は、td_ref_dsname()と同様である。

なお、設定可能な DS オブジェクト名称は、

使用可能文字(UB)a～z, A～Z, 0～9
名称長　8byte（満たない場合は NULL で埋める）

とするが、文字コードのチェックは、T-Kernel では行わない。

DS オブジェクト名称は、オブジェクトの属性に、TA_DSNAME が指定されている場合に有効である。TA_DSNAME 属性が指定されていないオブジェクトを対象とした場合は、エラーE_OBJ となる。

6.2 実行トレース機能

この機能はデバッガがプログラムの実行をトレースするための機能である。実行トレースはフックルーチンを設定することによって行う。

- フックルーチンでは、各種の状態をフックルーチンが呼び出された時点の状態に戻してから、フックルーチンから戻らなければならない。ただし、レジスタに関しては C 言語の関数の保存規則にしたがって復帰すればよい。
- フックルーチン内では、各種の状態を制限の緩い方へ変更してはいけない。例えば、割込み禁止状態で呼び出された場合は、割込みを許可してはいけない。
- フックルーチンは、保護レベル 0 で呼び出される。
- フックルーチンは、フックした時点のスタックをそのまま継承している。したがって、あまり多くスタックを消費するとスタックオーバーフローを引き起こす可能性がある。どの程度のスタックが使用可能であるかは、フックされた時点の状況により異なるため不確定である。フックルーチン内で独自スタックに切り替えれば、より安全である。

システムコール・拡張 SVC のフックルーチン定義 td_hok_svc

【C 言語インタフェース】
```
ER ercd = td_hok_svc ( TD_HSVC *hsvc ) ;
```

【パラメータ】

TD_HSVC*	hsvc	SVC Hook Routine	フックルーチン定義情報

hsvc の内容

FP	enter	呼出前のフックルーチン
FP	leave	呼出後のフックルーチン

【リターンパラメータ】

ER	ercd	ErrorCode	エラーコード

【解説】

システムコールおよび拡張 SVC の呼出前後に、フックルーチンを設定する。hsvc に NULL を指定することによりフックルーチンを解除する。

トレースの対象となるのは、T-Kernel/OS のシステムコール(tk_～)、および拡張 SVC である。ただし、実装によるが、一般に tk_ret_int はトレースの対象とならない。

T-Kernel/DS のシステムコール(td_～)は、トレースの対象とならない。

フックルーチンは、システムコール・拡張 SVC を呼び出したコンテキストで呼び出される。例えば、フックルーチン内での自タスクは、システムコール・拡張 SVC を呼び出したタスクと同じである。

システムコール内でタスクディスパッチや割込みが起こる場合があるため、enter()と leave()が常にペアで連続して呼び出されるとは限らない。また、システムコールから戻らない場合は、leave()は呼び出されない。

```
VP enter( FN fncd, TD_CALINF *calinf, ... )

    fncd    機能コード
        <  0 システムコール
        >= 0 拡張 SVC
    calinf  呼出元情報
    ...パラメータ(可変個数)
    戻値    leave() に引き渡す任意の値

typedef struct td_calinf {
      システムコール・拡張 SVC の呼出元(アドレス)を特定するための情報で、
      スタックのバックトレースを行うための情報が含まれることが望ましい。内容は実装定義となる。
      一般的には、スタックポインタやプログラムカウンタなどのレジスタの値である。
} TD_CALINF;
```

システムコールまたは拡張 SVC を呼び出す直前に呼び出される。

戻値に返された値は、そのまま対応する leave()に渡される。これにより、enter()と leave()のペアの確認や任意の情報の受け渡しを行うことができる。

```
exinf = enter(fncd, &calinf, ... )
ret = システムコール・拡張 SVC の実行
leave(fncd , ret, exinf)
```

・システムコールの場合
パラメータは、システムコールのパラメータと同じとなる。

(例) tk_wai_sem(ID semid, INT cnt, TMO tmout) の場合
enter(TFN_WAI_SEM, &calinf, semid, cnt, tmout)

・拡張 SVC の場合
パラメータは、拡張 SVC ハンドラに渡されるパケットの状態となる。
fncd も拡張 SVC ハンドラに渡されるものと同一である。
enter(FN fncd, TD_CALINF *calinf, VP pk_para)

void leave(FN fncd, INT ret, VP exinf)

fncd	機能コード
ret	システムコールまたは拡張 SVC の戻値
exinf	enter()で戻された任意の値

システムコールまたは拡張 SVC から戻った直後に呼び出される。
システムコールまたは拡張 SVC が呼び出された後(システムコールまたは拡張 SVC の実行中)にフックルーチンが設定された場合、enter()が呼び出されずに leave()のみ呼び出される場合がある。このような場合、exinf には NULL が渡される。
逆に、システムコールまたは拡張 SVC が呼び出された後フックルーチンが解除された場合、enter()が呼び出されて、leave()が呼び出されない場合がある。

タスクディスパッチのフックルーチン定義 td_hok_dsp

【C 言語インタフェース】
```
ER ercd = td_hok_dsp ( TD_HDSP *hdsp ) ;
```

【パラメータ】

TD_HDSP*	hdsp	Dispatcher Hook Routine	フックルーチン定義情報

hdsp の内容

FP	exec	実行開始時のフックルーチン
FP	stop	実行停止時のフックルーチン

【リターンパラメータ】

ER	ercd	Error Code	エラーコード

【解説】

タスクディスパッチャに、フックルーチンを設定する。hdsp に NULL を指定することによりフックルーチンを解除する。

フックルーチンは、ディスパッチ禁止状態で呼び出される。フックルーチンでは、T-Kernel/OS のシステムコール(tk_～)および拡張 SVC を呼び出してはいけない。T-Kernel/DS のシステムコール(td_～)は呼び出すことができる。

```
void exec( ID tskid, INT lsid )
```

tskid	実行を開始・再開するタスクのタスク ID
lsid	tskid のタスクの論理空間 ID

タスクの実行が開始・再開されるときに呼び出される。exec()が呼び出された時点で、すでに tskid のタスクは RUN 状態となっており、論理空間も切り替えられている。ただし、tskid のタスクのプログラムコードが実行されるのは、exec()から戻った後である。

```
void stop( ID tskid, INT lsid, UINT tskstat )
```

tskid	実行を停止したタスクのタスク ID
lsid	tskid のタスクの論理空間 ID
tskstat	tskid のタスクの状態

タスクが実行を停止した時に呼び出される。tskstat には、停止後のタスクの状態が示され、以下のいずれかとなる。

TTS_RDY	READY 状態 (実行可能状態)
TTS_WAI	WAIT 状態 (待ち状態)
TTS_SUS	SUSPEND 状態 (強制待ち状態)
TTS_WAS	WAIT-SUSPEND 状態
TTS_DMT	DORMANT 状態 (休止状態)
0	NON-EXISTENT 状態 (未登録状態)

stop()が呼び出された時点で、すでに tskid のタスクは tskstat で示した状態となっている。論理空間は不定である。

割込みハンドラのフックルーチン定義 td_hok_int

【C 言語インタフェース】
```
ER ercd = td_hok_int ( TD_HINT *hint ) ;
```

【パラメータ】

TD_HINT*	hint	Interruption Handler Hook Routine	フックルーチン定義情報

hint の内容

FP	enter	ハンドラ呼出前のフックルーチン
FP	leave	ハンドラ呼出後のフックルーチン

【リターンパラメータ】

ER	ercd	Error Code	エラーコード

【解説】

割込みハンドラの呼出前後に、フックルーチンを設定する。フックルーチンの設定は、例外・割込み要因ごとに独立に行うことはできない。すべての例外・割込み要因で共通のフックルーチンを 1 つのみ設定できる。

hint に NULL を指定することによりフックルーチンを解除する。

フックルーチンはタスク独立部(割込みハンドラの一部)として呼び出される。したがって、フックルーチンからはタスク独立部から発行可能なシステムコールのみ呼び出すことができる。

なお、フックルーチンを設定できるのは、tk_def_int で TA_HLNG 属性を指定して定義された割込みハンドラのみである。TA_ASM 属性の割込みハンドラのフックはできない。TA_ASM 属性の割込みハンドラをフックしたい場合は、例外･割込みベクタテーブルを直接操作してフックするなどの方法があるが､それらの方法は実装によって異なる。

```
void enter( UINT dintno )
void leave( UINT dintno )

dintno  割込みハンドラ番号
```

enter() および leave() に渡される引数は、例外・割込みハンドラに渡される引数と同じものである。実装によっては、dintno 以外の情報も渡される場合がある。

フックルーチンは、高級言語対応ルーチンから次のようにして呼び出される。

```
enter(dintno);
inthdr(dintno); /* 例外・割込みハンドラ */
leave(dintno);
```

enter()は割込み禁止状態で呼び出されることになる。また、割込みを許可してはいけない。leave()は、inthdr()から戻ったときの状態となるため、割込み禁止状態は不確定である。

enter()では、inthdr()で得ることのできる情報と同じだけの情報を得ることができる。逆に、inthdr()で得ることのできない情報は enter()でも得ることはできない。enter()および inthdr()で得ることのできる情報としては、dintno が仕様としては保証されているが、それ以外の情報については実装定義である。なお、leave()では割込み禁止状態など各種の状態が変化している場合があるため、enter()や inthdr()と同じだけの情報を得ることができるとは限らない。

第7章　レファレンス

7.1 C言語インタフェース一覧

□ T-Kernel/OS

● タスク管理機能

```
ID tskid = tk_cre_tsk ( T_CTSK *pk_ctsk );
ER ercd = tk_del_tsk ( ID tskid );
ER ercd = tk_sta_tsk ( ID tskid, INT stacd );
void tk_ext_tsk ( void );
void tk_exd_tsk ( void );
ER ercd = tk_ter_tsk ( ID tskid );
ER ercd = tk_chg_pri ( ID tskid, PRI tskpri );
ER ercd = tk_chg_slt ( ID tskid, RELTIM slicetime );
ER ercd = tk_get_tsp ( ID tskid, T_TSKSPC *pk_tskspc );
ER ercd = tk_set_tsp ( ID tskid, T_TSKSPC *pk_tskspc );
ID resid = tk_get_rid ( ID tskid );
ID oldid = tk_set_rid ( ID tskid, ID resid );
ER ercd = tk_get_reg ( ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs );
ER ercd = tk_set_reg ( ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs );
ER ercd = tk_get_cpr ( ID tskid, INT copno, T_COPREGS *pk_copregs );
ER ercd = tk_set_cpr ( ID tskid, INT copno, T_COPREGS *pk_copregs );
ER ercd = tk_inf_tsk ( ID tskid, T_ITSK *pk_itsk, BOOL clr );
ER ercd = tk_ref_tsk ( ID tskid, T_RTSK *pk_rtsk );
```

● タスク付属同期機能

```
ER ercd = tk_slp_tsk ( TMO tmout );
ER ercd = tk_wup_tsk ( ID tskid );
INT wupcnt = tk_can_wup ( ID tskid );
ER ercd = tk_rel_wai ( ID tskid );
ER ercd = tk_sus_tsk ( ID tskid );
ER ercd = tk_rsm_tsk ( ID tskid );
ER ercd =tk_frsm_tsk ( ID tskid );
ER ercd = tk_dly_tsk ( RELTIM dlytim );
ER ercd = tk_sig_tev ( ID tskid, INT tskevt );
INT tevptn = tk_wai_tev ( INT waiptn, TMO tmout );
INT tskwait = tk_dis_wai ( ID tskid, UINT waitmask );
ER ercd = tk_ena_wai ( ID tskid );
```

● タスク例外処理機能

```
ER ercd = tk_def_tex ( ID tskid, T_DTEX *pk_dtex );
ER ercd = tk_ena_tex ( ID tskid, UINT texptn );
ER ercd = tk_dis_tex ( ID tskid, UINT texptn );
ER ercd = tk_ras_tex ( ID tskid, INT texcd );
INT texcd = tk_end_tex ( BOOL enatex );
ER ercd = tk_ref_tex ( ID tskid, T_RTEX *pk_rtex );
```

● 同期・通信機能

```
ID semid = tk_cre_sem ( T_CSEM *pk_csem );
ER ercd = tk_del_sem ( ID semid );
ER ercd = tk_sig_sem ( ID semid, INT cnt );
ER ercd = tk_wai_sem ( ID semid, INT cnt, TMO tmout );
ER ercd = tk_ref_sem ( ID semid, T_RSEM *pk_rsem );
ID flgid = tk_cre_flg ( T_CFLG *pk_cflg );
ER ercd = tk_del_flg ( ID flgid );
ER ercd = tk_set_flg ( ID flgid, UINT setptn );
ER ercd = tk_clr_flg ( ID flgid, UINT clrptn );
ER ercd = tk_wai_flg ( ID flgid, UINT waiptn, UINT wfmode, UINT *p_flgptn, TMO tmout );
ER ercd = tk_ref_flg ( ID flgid, T_RFLG *pk_rflg );
ID mbxid = tk_cre_mbx ( T_CMBX* pk_cmbx );
ER ercd = tk_del_mbx ( ID mbxid );
ER ercd = tk_snd_mbx ( ID mbxid, T_MSG *pk_msg );
ER ercd = tk_rcv_mbx ( ID mbxid, T_MSG **ppk_msg, TMO tmout );
ER ercd = tk_ref_mbx ( ID mbxid, T_RMBX *pk_rmbx );
```

● 拡張同期・通信機能

```
ID mtxid = tk_cre_mtx ( T_CMTX *pk_cmtx );
ER ercd = tk_del_mtx ( ID mtxid );
ER ercd = tk_loc_mtx ( ID mtxid, TMO tmout );
ER ercd = tk_unl_mtx ( ID mtxid );
ER ercd = tk_ref_mtx ( ID mtxid, T_RMTX *pk_rmtx );
ID mbfid = tk_cre_mbf ( T_CMBF *pk_cmbf );
ER ercd = tk_del_mbf ( ID mbfid );
ER ercd = tk_snd_mbf ( ID mbfid, VP msg, INT msgsz, TMO tmout );
INT msgsz = tk_rcv_mbf ( ID mbfid, VP msg, TMO tmout );
ER ercd = tk_ref_mbf ( ID mbfid, T_RMBF *pk_rmbf );
ID porid = tk_cre_por ( T_CPOR *pk_cpor );
ER ercd = tk_del_por ( ID porid );
INT rmsgsz = tk_cal_por ( ID porid, UINT calptn, VP msg, INT cmsgsz, TMO tmout );
INT cmsgsz = tk_acp_por ( ID porid, UINT acpptn, RNO *p_rdvno, VP msg, TMO tmout );
ER ercd = tk_fwd_por ( ID porid, UINT calptn, RNO rdvno, VP msg, INT cmsgsz );
ER ercd = tk_rpl_rdv ( RNO rdvno, VP msg, INT rmsgsz );
ER ercd = tk_ref_por ( ID porid, T_RPOR *pk_rpor );
```

● メモリプール管理機能

```
ID mpfid = tk_cre_mpf ( T_CMPF *pk_cmpf );
ER ercd = tk_del_mpf ( ID mpfid );
ER ercd = tk_get_mpf ( ID mpfid, VP *p_blf, TMO tmout );
ER ercd = tk_rel_mpf ( ID mpfid, VP blf );
ER ercd = tk_ref_mpf ( ID mpfid, T_RMPF *pk_rmpf );
ID mplid = tk_cre_mpl ( T_CMPL *pk_cmpl );
ER ercd = tk_del_mpl ( ID mplid );
ER ercd = tk_get_mpl ( ID mplid, W blksz, VP *p_blk, TMO tmout );
ER ercd = tk_rel_mpl ( ID mplid, VP blk );
ER ercd = tk_ref_mpl ( ID mplid, T_RMPL *pk_rmpl );
```

● 時間管理機能

```
ER ercd = tk_set_tim ( SYSTIM *pk_tim );
ER ercd = tk_get_tim ( SYSTIM *pk_tim );
ER ercd = tk_get_otm ( SYSTIM *pk_tim );
ID cycid = tk_cre_cyc ( T_CCYC *pk_ccyc );
ER ercd = tk_del_cyc ( ID cycid );
ER ercd = tk_sta_cyc ( ID cycid );
ER ercd = tk_stp_cyc ( ID cycid );
ER ercd = tk_ref_cyc ( ID cycid, T_RCYC *pk_rcyc );
ID almid = tk_cre_alm ( T_CALM *pk_calm );
ER ercd = tk_del_alm ( ID almid );
ER ercd = tk_sta_alm ( ID almid, RELTIM almtim );
ER ercd = tk_stp_alm ( ID almid );
ER ercd = tk_ref_alm ( ID almid, T_RALM *pk_ralm );
```

● 割込み管理機能

```
ER ercd = tk_def_int ( UINT dintno, T_DINT *pk_dint );
void tk_ret_int ( void );
```

● システム状態管理機能

```
ER ercd = tk_rot_rdq ( PRI tskpri );
ID tskid = tk_get_tid ( void );
ER ercd = tk_dis_dsp ( void );
ER ercd = tk_ena_dsp ( void );
ER ercd = tk_ref_sys ( T_RSYS *pk_rsys );
ER ercd = tk_set_pow ( UINT powmode );
ER ercd = tk_ref_ver ( T_RVER *pk_rver );
```

● サブシステム管理機能

```
ER ercd = tk_def_ssy ( ID ssid, T_DSSY *pk_dssy );
ER ercd = tk_sta_ssy ( ID ssid, ID resid, INT info );
ER ercd = tk_cln_ssy ( ID ssid, ID resid, INT info );
ER ercd = tk_evt_ssy ( ID ssid, INT evttyp, ID resid, INT info );
ER ercd = tk_ref_ssy ( ID ssid, T_RSSY *pk_rssy );
ID resid = tk_cre_res ( void );
ER ercd = tk_del_res ( ID resid );
ER ercd = tk_get_res ( ID resid, ID ssid, VP *p_resblk );
```

□ T-Kernel/SM

● システムメモリ管理機能

```
ER tk_get_smb( VP *addr, INT nblk, UINT attr );
ER tk_rel_smb( VP addr );
ER tk_ref_smb( T_RSMB *pk_rsmb );
void* Vmalloc( size_t size );
```

```
void* Vcalloc( size_t nmemb, size_t size );
void* Vrealloc( void *ptr, size_t size );
void  Vfree( void *ptr );
void* Kmalloc( size_t size );
void* Kcalloc( size_t nmemb, size_t size );
void* Krealloc( void *ptr, size_t size );
void  Kfree( void *ptr );
```

● アドレス空間管理機能

```
ER SetTaskSpace( ID tskid );
ER ChkSpaceR( VP addr, INT len );
ER ChkSpaceRW( VP addr, INT len );
ER ChkSpaceRE( VP addr, INT len );
INT ChkSpaceBstrR( UB *str, INT max );
INT ChkSpaceBstrRW( UB *str, INT max );
INT ChkSpaceTstrR( TC *str, INT max );
INT ChkSpaceTstrRW( TC *str, INT max );
ER LockSpace( VP addr, INT len );
ER UnlockSpace( VP addr, INT len );
INT CnvPhysicalAddr( VP vaddr, INT len, VP *paddr );
ER MapMemory( VP paddr, INT len, UINT attr, VP *laddr );
ER UnmapMemory( VP laddr );
```

● デバイス管理機能

```
ID tk_opn_dev( UB *devnm, UINT omode );
ER tk_cls_dev( ID dd, UINT option );
ID tk_rea_dev( ID dd, INT start, VP buf, INT size, TMO tmout );
ER tk_srea_dev( ID dd, INT start, VP buf, INT size, INT *asize );
ID tk_wri_dev( ID dd, INT start, VP buf, INT size, TMO tmout );
ER tk_swri_dev( ID dd, INT start, VP buf, INT size, INT *asize );
ID tk_wai_dev( ID dd, ID reqid, INT *asize, ER *ioer, TMO tmout );
INT tk_sus_dev( UINT mode );
ID tk_get_dev( ID devid, UB *devnm );
ID tk_ref_dev( UB *devnm, T_RDEV *rdev );
ID tk_oref_dev( ID dd, T_RDEV *rdev );
INT tk_lst_dev( T_LDEV *ldev, INT start, INT ndev );
INT tk_evt_dev( ID devid, INT evttyp, VP evtinf );
ID tk_def_dev( UB *devnm, T_DDEV *ddev, T_IDEV *idev );
ER tk_ref_idv( T_IDEV *idev );
```

● 割込み管理機能

```
DI( UINT intsts );
EI( UINT intsts );
BOOL isDI( UINT intsts );
UINT DINTNO( INTVEC intvec );
void EnableInt( INTVEC intvec [, INT level] );
void DisableInt( INTVEC intvec );
void ClearInt( INTVEC intvec );
void EndOfInt( INTVEC intvec );
```

```
BOOL CheckInt( INTVEC intvec );
```

● I/O ポートアクセスサポート機能

```
void out_w( INT port, UW data );
void out_h( INT port, UH data );
void out_b( INT port, UB data );
UW in_w( INT port );
UH in_h( INT port );
UB in_b( INT port );
void WaitUsec( UINT usec );
void WaitNsec( UINT nsec );
```

● 省電力機能

```
void low_pow( void );
void off_pow( void );
```

● システム構成情報管理機能

```
INT tk_get_cfn( UB *name, INT *val, INT max );
INT tk_get_cfs( UB *name, UB *buf, INT max );
```

□ T-Kernel/DS

● カーネル内部状態取得機能

```
INT ct = td_lst_tsk ( ID list[], INT nent );
INT ct = td_lst_sem ( ID list[], INT nent );
INT ct = td_lst_flg ( ID list[], INT nent );
INT ct = td_lst_mbx ( ID list[], INT nent );
INT ct = td_lst_mtx ( ID list[], INT nent );
INT ct = td_lst_mbf ( ID list[], INT nent );
INT ct = td_lst_por ( ID list[], INT nent );
INT ct = td_lst_mpf ( ID list[], INT nent );
INT ct = td_lst_mpl ( ID list[], INT nent );
INT ct = td_lst_cyc ( ID list[], INT nent );
INT ct = td_lst_alm ( ID list[], INT nent );
INT ct = td_lst_ssy ( ID list[], INT nent );
INT ct = td_rdy_que ( PRI pri, ID list[], INT nent );
INT ct = td_sem_que ( ID semid, ID list[], INT nent );
INT ct = td_flg_que ( ID flgid, ID list[], INT nent );
INT ct = td_mbx_que ( ID mbxid, ID list[], INT nent );
INT ct = td_mtx_que ( ID mtxid, ID list[], INT nent );
INT ct = td_smbf_que ( ID mbfid, ID list[], INT nent );
INT ct = td_rmbf_que ( ID mbfid, ID list[], INT nent );
INT ct = td_cal_que ( ID porid, ID list[], INT nent );
INT ct = td_acp_que ( ID porid, ID list[], INT nent );
INT ct = td_mpf_que ( ID mpfid, ID list[], INT nent );
INT ct = td_mpl_que ( ID mplid, ID list[], INT nent );
```

```
ER ercd = td_ref_tsk ( ID tskid, TD_RTSK *rtsk );
ER ercd = td_ref_sem ( ID semid, TD_RSEM *rsem );
ER ercd = td_ref_flg ( ID flgid, TD_RFLG *rflg );
ER ercd = td_ref_mbx ( ID mbxid, TD_RMBX *rmbx );
ER ercd = td_ref_mtx ( ID mtxid, TD_RMTX *rmtx );
ER ercd = td_ref_mbf ( ID mbfid, TD_RMBF *rmbf );
ER ercd = td_ref_por ( ID porid, TD_RPOR *rpor );
ER ercd = td_ref_mpf ( ID mpfid, TD_RMPF *rmpf );
ER ercd = td_ref_mpl ( ID mplid, TD_RMPL *rmpl );
ER ercd = td_ref_cyc ( ID cycid, TD_RCYC *rcyc );
ER ercd = td_ref_alm ( ID almid, TD_RALM *ralm );
ER ercd = td_ref_ssy ( ID  ssid, TD_RSSY *rssy );
ER ercd = td_ref_tex ( ID tskid, TD_RTEX *pk_rtex );
ER ercd = td_inf_tsk ( ID tskid, TD_ITSK *pk_itsk, BOOL clr );
ER ercd = td_get_reg ( ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs );
ER ercd = td_set_reg ( ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs );
ER ercd = td_ref_sys ( TD_RSYS *pk_rsys );
ER ercd = td_get_tim ( SYSTIM *tim, UNIT *ofs );
ER ercd = td_get_otm ( SYSTIM *tim, UINT *ofs );
ER ercd = td_ref_dsname( UINT type, ID id, UB *dsname );
ER ercd = td_set_dsname( UINT type, ID id, UB *dsname );
```

● 実行トレース機能

```
ER ercd = td_hok_svc ( TD_HSVC *hsvc );
ER ercd = td_hok_dsp ( TD_HDSP *hdsp );
ER ercd = td_hok_int ( TD_HINT *hint );
```

7.2 エラーコード一覧

――― 正常終了のエラークラス (0) ―――

E_OK　0　正常終了

――― 内部エラークラス (5～8) ―――

E_SYS　ERCD(-5, 0) システムエラー
　原因不明のエラーであり、システム全体に影響するエラーである。

E_NOCOP　ERCD(-6, 0) コプロセッサ使用不可
　現在動作中のハードウェアに指定のコプロセッサが搭載されていない。または、コプロセッサの動作異常を検出した。

――― 未サポートエラークラス (9～16) ―――

E_NOSPT　ERCD(-9, 0) 未サポート機能
　システムコールの一部の機能がサポートされていない場合に、その機能を指定すると、E_RSATR または E_NOSPT のエラーを発生する。E_RSATR に該当しない場合には、E_NOSPT のエラーとなる。

E_RSFN　ERCD(-10, 0)予約機能コード番号
　予約機能コード(未定義の機能コード)を指定してシステムコールを実行しようとした場合に、このエラーが発生する。未定義の拡張 SVC ハンドラを実行しようとした場合(機能コードが正の場合)にも、このエラーが発生する。

E_RSATR　ERCD(-11, 0)予約属性
　未定義やサポートしていないオブジェクト属性を指定した場合に発生する。
　システム依存の適応化を行う場合、このエラーのチェックは省略されることがある。

――― パラメータエラークラス (17～24) ―――

E_PAR　ERCD(-17, 0)パラメータエラー
　システム依存の適応化を行う場合、このエラーのチェックは省略されることがある。

E_ID　ERCD (-18, 0)不正 ID 番号
　E_ID は ID 番号を持つオブジェクトに対してのみ発生するエラーである。
　割込みハンドラ番号などの範囲外や予約番号といった静的なエラーが検出された場合には、E_PAR のエラーが発生する。

――― 呼出コンテキストエラークラス (25～32) ―――

E_CTX　ERCD(-25, 0)コンテキストエラー
　このシステムコールを発行できるコンテキスト(タスク部/タスク独立部の区別やハンドラ実行状態)にはないということを示すエラーである。
　自タスクを待ち状態にするシステムコールをタスク独立部から発行した場合のように、システムコールの発行コンテキストに関して意味的な間違いのある場合には、必ずこのエラーが発生する。また、それ以外のシステムコールであっても、実装の制約のため、あるコンテキスト(割込みハンドラなど)からそのシステムコールを発行できない場合に、このエラーが発生する。

E_MACV　ERCD(-26, 0)メモリアクセス不能,メモリアクセス権違反
　エラーの検出は実装依存である。

E_OACV　ERCD(-27, 0)オブジェクトアクセス権違反
　ユーザタスクがシステムオブジェクトを操作した場合に発生する。
　システムオブジェクトの定義およびエラーの検出は実装依存である。

E_ILUSE　ERCD(-28, 0)システムコール不正使用

――― 資源不足エラークラス (33～40) ―――――――――――――――――――――――――――――――

E_NOMEM　ERCD(-33, 0)メモリ不足
　オブジェクト管理ブロック領域、ユーザスタック領域、メモリプール領域、メッセージバッファ領域などを獲得する時のメモリ不足 (no memory)

E_LIMIT　ERCD(-34, 0)システムの制限を超過
　オブジェクト数の上限を超えてオブジェクトを生成しようとした場合など。

――― オブジェクト状態エラークラス (41～48) ―――――――――――――――――――――――――――

E_OBJ　　ERCD(-41, 0)オブジェクトの状態が不正

E_NOEXS　ERCD(-42, 0)オブジェクトが存在していない

E_QOVR　ERCD(-43, 0)キューイングまたはネストのオーバーフロー

――― 待ち解除エラークラス (49～56) ―――――――――――――――――――――――――――――――

E_RLWAI　ERCD(-49, 0)待ち状態強制解除

E_TMOUT　ERCD(-50, 0)ポーリング失敗またはタイムアウト

E_DLT　　ERCD(-51, 0)待ちオブジェクトが削除された

E_DISWAI ERCD(-52, 0)待ち禁止による待ち解除

――― デバイスエラークラス (57～64) (T-Kernel/SM) ―――――――――――――――――――――――

E_IO　　ERCD(-57, 0)入出力エラー
※ E_IO のサブエラーコードには、デバイスごとにエラー状態等を示す値が定義される場合がある。

E_NOMDA　ERCD(-58, 0)メディアがない

――― 各種状態エラークラス (65～72) (T-Kernel/SM) ―――――――――――――――――――――――

E_BUSY　ERCD(-65, 0)ビジー状態

E_ABORT　ERCD(-66, 0)中止した

E_RONLY　ERCD(-67, 0)書込み禁止