ITRON・μITRON 標準ハンドブック

坂村 健 監修

パーソナルメディア

ITRON・μITRON標準ハンドブック

仕様に関しては下記にお問い合わせください。

社団法人 トロン協会
〒105 東京都港区虎ノ門3-8-27
巴町アネックス2号館5F
電話 (03) 433-6741

TRONは、 The Realtime Operating system Nucleusの略称です。
ITRONは、 Industrial TRONの略称です。
BTRONは、 Business TRONの略称です。
CTRONは、 Central and Communication TRONの略称です。
MTRONは、 Macro TRONの略称です。

監修のことば

来たるべき21世紀には、社会を構成するあらゆる要素がコンピュータ化され、ネットワークで結ばれることになるであろう。TRONプロジェクトは、このような電脳社会を構築するための基礎を固めようとして始められたプロジェクトである。

TRONプロジェクトでは、コンピュータとセンサ、アクチュエータが一体となって組み込まれ、電脳社会の構成要素となるものをインテリジェントオブジェクト(知的物体)と呼んでいる。TRONプロジェクトを進める上では、今後社会のインテリジェントオブジェクト化が進むであろうということと、これらのインテリジェントオブジェクトがネットワーク化され、人間の住環境を快適にするために協調してはたらくであろうということが前提になっている。

TRONプロジェクトには、基礎プロジェクトと応用プロジェクトがある。応用プロジェクトは、インテリジェントオブジェクト化が進んだ未来の電脳社会を先取りしてシミュレーションすることを目的としたプロジェクトであり、電脳住宅プロジェクト、電脳ビルプロジェクト、電脳都市プロジェクト、電脳自動車網プロジェクトから成っている。一方、基礎プロジェクトは、応用プロジェクトからくる要求を満たすようなコンピュータを設計するためのプロジェクトであり、ITRON、BTRON、CTRON、MTRON、TRON仕様CHIPのサブプロジェクトに分かれている。

これらの基礎プロジェクトのうち、ITRONサブプロジェクトは最も長い歴史を持っている。ITRON仕様の設計は、1984年のTRONプロジェクト開始とともに着手され、着実な成果を上げてきた。その理由としては、インテリジェントオブジェクトを作る際の実行時オペレーティングシステムとしてITRON仕様のOSが必要であることに加えて、BTRON仕様OSなど他のオペレーティングシステムのカーネルとしてもITRON仕様OSを利用できるためである。すなわち、ITRONサブプロジェクトはTRONプロジェクト全体の中で極めて重要な位置を占めている。

現在では、ITRON仕様自体のシリーズ化や機能拡張も進んでいる。ITRON仕様のシリーズ化は、16ビット汎用マイクロプロセッサ向けのITRON1に始まり、現在では32ビットマイクロプロセッサ(特にTRONCHIP) 向けのITRON2、

およびシングルチップマイクロプロセッサ向けのμITRONへと発展してきた。もともとITRON仕様OSはいろいろな種類のプロセッサに実装することを前提として設計されたオペレーティングシステムであるが、こういったシリーズ化の手法により、さらにその適用範囲を広げることが可能となった。ITRON仕様OSは、これまでに数十のマイクロプロセッサの上に実際にインプリメントされ、その性能が確認されている。一方、ITRON仕様の機能拡張としては、標準入出力機能、BTRON仕様と互換性を持つファイル管理機能、ITRON仕様OS同士を接続するためのネットワーク機能などの仕様が追加されている。

本書では、このうち、32ビットマイクロプロセッサおよびネットワークへの対応を前提としたITRON2の仕様と、基本機能を重視し、8ビットマイクロプロセッサやシングルチップマイクロプロセッサの上でもインプリメントを可能としたμITRONの仕様を解説している。読者は本書を読むことにより、TRONプロジェクトの中におけるITRON2およびμITRON仕様の位置付け、これらのリアルタイムオペレーティングシステムの機能、詳細仕様などを知ることができよう。

ITRON2、μITRON仕様OSとも、現在では製品として入手可能になっており、多くのインプリメンタがTRON仕様CHIPを含めたいろいろなプロセッサの上で製品化を行っている。リアルタイムオペレーティングシステムの世界において、これだけ多くのインプリメンタが参加し、これだけ多くのプロセッサの上で動作するものは、ITRON仕様OS以外にはないと思われる。

本書がITRON仕様あるいはリアルタイムオペレーティングシステムに興味を持つ方々のお役に立てば幸いである。本書の利用によってITRON仕様OSのユーザやインプリメンタが増え、それがTRONプロジェクトの最終目標である電脳社会の構築に結び付くことを願ってやまない。

1990年12月

坂村 健
TRONプロジェクトリーダ

目次

図版目次

μITRONシステムコールの記述形式

本書のμITRON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なっている。

和文説明

［レベル］システムコール名

システムコール名称:　英文説明（名称の由来）

※ このうち、［レベル］で示されている数字は、インプリメントの必要性の度合いを示すものである。数字の小さい方が必要性の高いことを表わす。

※［レベル］の項の数字にA,Bの記号が付いているシステムコールは、インプリメンタの側で、A,Bのどちらかの仕様を選択してインプリメントすることを示す。

※［レベル］の項の数字に#の記号が付いているシステムコールは、他の方法で代替できる可能性のあるシステムコールを示す。例えば、def_intには#が付いているが、これは、割込みハンドラ定義の機能が他の方法（例えばOS初期化時の静的な定義）で代替できれば、そのシステムコールが不要であることを示す。

【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μITRONに渡す情報に関する説明を行なう

【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μITRONから返される情報（エラーコンディションを除く）に関する説明を行なう

【解説】

システムコールの機能の解説を行なう

※ 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)

※ μITRON では、【解説】に述べられている機能（システムコールの機能）のうち、一部の機能のみをインプリメントすることも許している。ただし、システムコールの一部の機能のみを提供しているという点については、マニュアルに明記して頂く必要がある。この時、そのシステムコールに対してサポートされていない機能を指定した場合には、E_NOSPT のエラーが返る。

※ 【解説】では、「そのシステムコールの実行中に割込みが発生し、同じオブジェクトが操作されることは無い」といった前提のもとで説明を行っている。これは、システムコール実行中またはその直前直後に割り込みが入ったケースまで考えると、システムコールの機能説明ができなくなってしまうためである。例えば、「資源数 = 3 のセマフォに対して sig_sem (rcnt = 1) のシステムコールを実行すると、資源数は 4 になる」といった説明も、システムコール実行中に割込みが入り、その中で sig_sem が実行されるケースを考えると正確ではない。【解説】は、途中の割込みがないという条件の下での動作を説明したものである。

【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なう

※ 以下のエラーコードについては、各システムコールに対するエラー

コードの説明の項には含めていない場合があるが、各システムコールにおいて共通して発生する可能性がある。

E_SYS, E_RSFN, E_NOSPT, E_XNOSPT, E_CTX

※ μITRONの場合、インプリメント方法の制約やそれによる性能の低下を避けるために、エラーチェックに関する細かい規定は設けず、エラーコードを大まかに標準化するだけにとどめている。したがって、この項で提示されるエラーコードの仕様も、厳密に守らなければならないものではない。インプリメントによっては、一部のエラーコードを検出しない場合、より多くのエラーコードを検出する場合、エラー検出の条件が多少異なっている場合などがある。

※ この項で説明されているエラーコードは、原則として、起こりうる可能性をもつエラーコードの最大限のものである。ここに出ているエラーを、実際にすべて発生しなければならないという意味ではない。インプリメントの違いやエラー検出方法の違いにより、実際には発生しないエラーがあっても構わない。例えば、wai_tsk の E_TNOSPT は、常にタイマが利用できれば発生する可能性の無いエラーである。

ITRON2システムコールの記述形式

本　書のITRON2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なっている。

和文説明

システムコール名称

システムコール名称:　英文説明（名称の由来）

【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ITRON2に渡す情報に関する説明を行なう

【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ITRON2から返される情報（エラーコンディションを除く）に関する説明を行なう

【解説】

システムコールの機能の解説を行なう

※ 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)

※【解説】では、「そのシステムコールの実行中に割込みが発生し、同じオブジェクトが操作されることは無い」といった前提のもとで説明を行っている。これは、システムコール実行中またはその直前直後に割り込みが入ったケースまで考えると、システムコールの機能説明ができなくなってしまうためである。例えば、「資源数 = 3 のセマフォに対して sig_sem (rcnt = 1) のシステムコールを実行すると、資源数は4になる」といった説明も、システムコール実行中に割込みが入り、その中で sig_sem が実行されるケースを考えると正確ではない。【解説】は、途中の割込みがないという条件の下での動作を説明したものである。

【補足事項】

仕様決定の理由や注意すべき点など、解説に対する補足事項を述べる

【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なう

※ 以下のエラーコードについては、各システムコールに対するエラーコードの説明の項には含めていない場合があるが、各システムコールにおいて共通して発生する可能性がある。

E_SYS, E_RSFN, E_NOSPT, E_XNOSPT, E_CTX

※ この項で説明されているエラーコードは、原則として、起こりうる可能性をもつエラーコードの最大限のものである。ここに出ているエラーを、実際にすべて発生しなければならないという意味ではない。インプリメントの違いやエラー検出方法の違いにより、実際には発生しないエラーがあっても構わない。例えば、wai_tsk の E_TNOSPT は、常にタイマが利用できれば発生する可能性の無いエラーである。

第一部
ITRONの基礎

第一部　第一章

TRONプロジェクトとITRON

TRONプロジェクトの特徴

マイクロプロセッサが誕生したのは1970年代初めであるが、その進歩は著しく、応用分野は今後もますます広がるであろう。数多くのマイクロプロセッサが人間の生活空間の細部に入り込み、それらが相互に通信し、分散処理を行ないながら、より快適な生活を人間に提供するようになると思われる。1990年代には50〜100万トランジスタ以上の集積度を持つVLSIチップ出現すると予測されているが、進んだ半導体技術を利用すれば、必要とされるマイクロプロセッサの処理能力に対応することは可能である。

しかしながら、現在のマイクロプロセッサは、まだ応用からの要求にとって十分な能力を提供しているとは言い難い。つまり、現在のマイクロプロセッサは、ハードウエア資源の乏しかった頃に作られたアーキテクチャをそのまま引きずっており、多くの部分で歪みを生じているということである。このまま半導体技術が進展すると、ハードウエア資源とアーキテクチャとのギャップがますます大きくなり、ハードウエアの能力を生かし切ることができなくなる恐れがある。応用分野からのトップダウンの要求が高まり、半導体技術の進歩によるボトムアップの性能向上がなされても、その中間に位置するアーキテクチャが古いままであると、システム全体としてのバランスが非常に悪くなってしまう。よりVLSIを意識し、応用からの要求に応え、90年代にターゲットを合わせた、新しいマイクロプロセッサのアーキテクチャ体系が欲しい。

TRONプロジェクトは、このような問題点を解決すべく、マイクロプロセッサから応用分野にわたる一貫したコンピュータ体系を、全く新しく作り直そうというものであり、1984年6月から始まった。TRONプロジェクトでは、将来のマイクロプロセッサの応用分野やユーザからくる要求と、1990年代のVLSIの技術予測とを調べ、従来のアーキテクチャからの互換性にとらわれない新しいアーキテクチャを提唱する。このアーキテクチャ体系は、リアルタイム特性に優れ、VLSIベース、分散処理指向といった特長をもち、1990年代における一つの標準となることを目指す。

TRONプロジェクトの基本的な考え方をまとめると、次のようになる。

・1990年代の技術水準をターゲットとした設計を進める。
現在のマイクロプロセッサのアーキテクチャは8ビット時代との互換性をいまだに引きずっているものが多く、VLSI化を始めから考慮したものではない。VLSI化で得られる豊富なハードウエア資源を、古いアーキテクチャに無理に付け加えたような形になっており、無駄が多い。TRONプロジェクトでは、始めからVLSIの存在を前提とした設計を行なう。

・リアルタイム処理を前提とする。
従来の計算機の使い方は、バッチ処理やTSS処理に見られるように、「人間の方が計算機を待つ」ものであった。しかし、今後は自然界を含めた外部環境を制御し、人間の生活をより快適にするために多くの計算機が使用される。この場合は、「計算機の方が人間（または外部環境からの刺激）を待つ」といった形態になる。これがリアルタイム性の本質である。リアルタイム性は、分散処理、通信などの応用においても必須のものである。しかし、リアルタイム処理では計算機の稼働率がさがるため、同じ処理能力を得るにはより高性能の計算機が必要になる。TRONプロジェクトではリアルタイム処理を必須のものと考え、始めからそれを考慮して設計を行なった。

・究極のノイマン型計算機を目指す。
TRONプロジェクトでの研究の対象は、あくまでも汎用的なノイマン型計算機である。非ノイマン型アーキテクチャの研究はさかんに行なわれており、中にはノイマン型計算機の可能性をいたずらに否定する意見もある。しかし、ノイマン型計算機の汎用性や技術の完成度などを見た場合、1990年代になっても社会の中心になって使われるのがノイマン型計算機であることは間違いない。TRONプロジェクトの目標は、VLSIの特徴を生かせるような究極のノイマン型計算機のモデルを提供することである。

・計算機の各階層の設計を同時に行ない、トータルアーキテクチャを目指す。
TRONプロジェクトでは、マイクロプロセッサからネットワークまでの設計を、同時に行なっている。したがって、OSでネックになる部分はプロセッサの命令で強化したり、ネットワークの構築のために必要なシステムコールをOSに導入したりといった、広い範囲にわたる設計のフィードバックが可能である。このように、設計段階において階層間にまたがるチューニングまで行なえるため、実質的な計算機の速度であるアプリケーションの実行速度を大幅に向上させることが可能である。

TRONのサブプロジェクト

TRONプロジェクトでは、5つのサブプロジェクトを同時に進行させている。プロジェクト全体の核となるのが、1990年代の半導体技術を生かせるようなアーキテクチャを持つマイクロプロセッサ、TRONCHIPである。TRONCHIPは、1990年代のアプリケーションやOSの実行に最も適したマイクロプロセッサとなる。次にOSの階層として、組み込み型制御用のリアルタイムOSであるITRON(Industorial-TRON)、ワークステーション向きのOSであるBTRON(Business-TRON)、大規模なデータ処理を行なうセンターマシンのためのOSであるCTRON(Central-TRON)がある。このうち、CTRONはマイクロコンピュータ用というよりはメインフレーム用のOSである。OSの階層を応用ニーズに応じて3つに分けたのは、組み込み用コンピュータ、ワークステーションコンピュータ、センターマシン用メインフレームのすべてを一つのOSでカバーするのは、無理があると判断したからである。さらに、複数のITRON、BTRON、CTRONをネートワークで有機的に結合し、人間の生活する環境全体を総合的に操作するのがMTRON(Macro-TRON)である。

TRONプロジェクトの特徴の一つとして、設計基準、すなわちアーキテクチャと実現手段、すなわちインプリメントを分離したことがある。ここで、アーキテクチャという言葉を定義しておこう。TRONプロジェクトの中では、G.M.Amdahlが定義した「アーキテクチャ」の意味を拡張し、「プログラマから見た命令セットのみならず、オペレーティングシステム、マンマシンインターフェースまで含めた、システムの各層の属性を規定するもの」といった意味で「アーキテクチャ」という言葉を用いている。具体的には、TRONCHIPの命令セットから、ITRONやBTRONのシステムコール、BTRONのマンマシンインターフェースまでのすべてのインタフェース仕様がTRONのアーキテクチャに含まれる。アーキテクチャとインプリメント手法の分離により、ある階層より下のインプリメンテーションが全く変わっても、それより上位の階層はそのまま使用できるという利点がある。すなわち、階層ごとに互換性が保たれている。また、TRONプロジェ

クトで重要なことは、アーキテクチャは規定するが性能に関する規定は行なわないことである。したがって、その部分がインプリメンタ間の自由競争の部分となる。ユーザから見ると、使い勝手は同じだが性能は異なるという製品が、メーカーの違いや時期の違いによって何種類も提供されるわけである。つまり、TRONプロジェクトは、統一化と同時に公平な技術競争を行なうための土俵を提供するものであると言える。

第一部 第二章

ITRONの基本方針

マルチタスクOSとリアルタイムOS

マイクロプロセッサを核として構成したコンピュータ、すなわちマイクロコンピュータは、小さい、安い、といった特徴を生かして、各種制御装置の置き替えというコンピュータの新しい応用分野を開拓した。すなわち、これまでリレーや機械仕掛け、アナログ回路などによって実現されていた制御装置を、小型のコンピュータによって置き替えることが可能になったのである。具体的には、電話交換機、工業用のロボット、NC機械などを始め、家電製品のエアコン、洗濯機、テレビ、など多くのものがマイコン制御となった。

こういった各種の制御装置に使用されるマイクロコンピュータの場合、メインフレームやパソコン、ワークステーションのようにそのコンピュータ自体に存在目的があるのではない。これらの制御用コンピュータは、より大きなシステムの一つの部品として使われている点に特色があり、そういった意味で、「組み込みコンピュータシステム（略して組み込みシステム）」と呼ばれる。

一般に、組み込みシステムでは外部で発生したいろいろな刺激や環境の変化をセンサーで感知して、それに対して応答の動作を起こしたり、周期的に外部に対して動作を起こしたりする能力が必要とされる。例えば、エアコンであれば外部の温度や湿度をセンサーで感知し、それに応じてエアコンの出力や風向きを制御するのである。組み込みシステムのプログラムは、刻々変化する外部環境との関係に遅れることなく追従し、それを制御しなければならないという点で、一般的なデータ処理のプログラムとはかなり異なった性質を持っている。

組み込みシステムのプログラムの生産性を上げ、工期を短縮するためには、プロセッサの命令をそのまま使うのではなく、共通の管理プログラム、すなわちオペレーティングシステム (OS) を利用し、その上にシステムに応じたアプリケーションプログラムを載せるのが有効である。この場合、OSといっても、汎用コンピュータのOSのようにユーザ間の保護やファイルの管理を主な目的とするものとは異なり、並行した処理や割込みによる外部

事象との同期などのプログラミングを容易にするのが目的のものである。

一般に、コンピュータ、あるいはその上のプログラムはシーケンシャルに走るものであり、同時に二つのことを行なう能力はない。ところが、組み込みシステムの場合には、センサーが複数存在し、それらを並行して監視しなければならないことが多い。監視するものが二つあれば、プログラムも二つに分け、それらが同時に並行して走ると考える方が自然であり、プログラミングも容易になる。このような要求から生まれたのがマルチタスクの考え方である。すなわち、全体の処理のうちで、並行した流れを持つ部分をタスクとして分割し、プログラムをタスクに分けることによって、わかりやすさやプログラムの生産性を上げようというものである。もちろん、裸のプロセッサではシーケンシャルに命令を実行する能力しかないので、マルチタスク機能はOSによって時分割で実現することになる。マルチタスク機能を持つOSをマルチタスクOSと呼び、これは組み込みシステム用のOSとして非常に有効なものである。

汎用コンピュータのOSも、複数のジョブ（プロセス、タスク）などを時分割で並行して実行するという意味ではマルチタスクOSである。しかし、汎用コンピュータの場合には、ジョブ（プロセス、タスク）どうしの独立性が強いのが普通である。例えば、同じ計算機である人が文書処理を行ない、ある人がコンパイルを行ない、ある人がエディタを使う、という意味ではマルチタスクで処理が行なわれているが、それぞれの人は自分の実行しているジョブ（プログラム）にのみ興味があるわけであり、並行してどのような処理が行なわれているかについては、特に関知しない。この場合、マルチタスク処理を行なう目的は、主にコスト・パフォーマンスの向上である。つまり、それぞれの人が専用のコンピュータを使用するのが理想であるにもかかわらず、コンピュータの台数が足りないために一つの計算機を共用する必要があり、その結果、マルチタスク処理になるのである。

一方、組み込みシステムの場合は、もともと一つの目的をもった作業を、プログラムの書きやすさという面から複数のタスクに分割したものである。したがって、それぞれのタスクが勝手に動いているわけではなく、タスク間の関係は極めて複雑である。組み込みシステムの場合は、その中あるタスクの動きや相互関係を常に見渡しているシステム全体の設計者がおり、複数のタスクが協力して一つの目的を達成するのである。こういった意味で、汎用コンピュータと組み込みシステムではマルチタスクの性格がかな

り異なっている。

組み込みシステム用のOSでもうひとつ重要な点は、リアルタイム性と呼ばれるものである。リアルタイム性（即時性）とは、抽象的に説明すると、コンピュータの処理が外部の状況の変化に追従できることであるが、具体的には、処理結果を出すまでの時間が制限されていることを指す。より簡単に言えば、応答速度が十分速いこと、あるいは応答速度の最悪の値が予測できることである。

従来の計算機の使い方は、バッチ処理やTSS処理に見られるように、「人間の方が計算機を待つ」ものであった。こういったケースでは、コンピュータで計算をすること自体が一つの目的となっており、ほかのシステムとの関連があるわけではない。したがって、処理時間が長くなっても、単に人間が待っていれば済むことが多い。処理が速いに越したことはないが、処理が遅くなっても致命的な影響を与えるわけではない。

一方、組み込みシステムの場合には、計算すること自体が目的なのではなく、計算結果を使って他の機器を制御することが目的である。したがって、処理時間に対する要求が厳しく、処理が遅れると、結果を出しても無意味になってしまうことが多い。例えば、新幹線の保安機器の制御であれば、異常を検出した場合に一定時間以内にブレーキをかけなければ、大事故につながる。また、自動車のエンジンのプラグ点火のタイミングを組み込みコンピュータシステムによって制御する場合、ちょうどエンジンの回転がその位置に達するまでに結果を出しておかないと、タイミングを計算した意味がなくなってしまう。つまり、処理結果がどうなったか、ということと同じように、いつ結果を得られるか、ということも重要なのであり、遅れて処理結果がでるのは処理結果を出さないのと同じなのである。

ところが、処理時間に対する厳しい要求を満たすためには、「計算機の方が外部環境からの刺激を待つ」といった形態にならざるを得ず、計算機の稼働率は低下する。つまり、計算機の使い方が贅沢になるのである。これは、今日のようにコンピュータが普及して安価になったからこそはじめて実現できたことである。リアルタイム、組み込みといった用途がコンピュータの応用分野として比較的新しいものであるのは、こういった理由による。

普通のプログラミングであれば、そのアルゴリズム、データ構造やおおまかな実行効率、計算量のオーダーを議論することはあっても、絶対的なプログラムの実行時間まで議論することは少ない。しかし、組み込みシス

テムへの応用では、上で述べたように、実行時間があらかじめ予測できないと困ることが多い。それも、個々のプログラム（ジョブ、プロセス、タスク）のCPUタイムではなく、他のタスクの実行状況などを含めて、システム全体の負荷まで計算に入れた絶対時間の見積りが必要なのである。これは、極めて難しい問題であり、大型機のような複雑なOSではまず不可能である。OS自体を軽く単純なものにしておかないと、OSのオーバーヘッドまで含めた時間の見積りはできない。

組み込みシステムに使用するOSは、以上のような意味で、一般的なデータ処理を行なうOSや開発用のOSとはかなり異なったものである。組み込みシステムのためのOSは、マルチタスク処理が可能で、リアルタイム性を持ったリアルタイム・マルチタスクOSでなければならない。

ITRONの特長

ITRON (Industrial - The Realtime Operating system Nucleus) は、産業用の組み込みシステムに使用することを目的として作られたリアルタイム・マルチタスクOSである。ITRONの大きな特長は、複数のプロセッサに実装するOSであるにもかかわらず、その処理性能を高くするために、必要以上の標準化、仮想化を行なっていないことである。また、もう一つの大きな特長は、豊富な機能を持つOSであるにもかかわらず、適応化を考慮し、実際のオブジェクトをコンパクトなものにすることができる点である。

標準化や仮想化を進めることは、ソフトウエアの互換性や教育の面からは望ましいことであるが、そのために基本性能が低下するという点ではマイナスである。また、豊富な機能を持つということは、プログラムの書きやすさや特定のケースにおける性能向上という点からは望ましいことであるが、そのためにオブジェクトコードのサイズが大きくなるという点ではマイナスである。OSを設計する場合には、この両者のバランスをとることが重要であり、その一つの答えとして出てきたのがITRONなのである。また、これは、TRONプロジェクト全体に共通する「弱い標準化」の考え方を、リアルタイムOSという分野で具体化したものであるとも言える。

こういったITRONの特長をまとめると、次のようになる。

・プロセッサの仮想化をしない。
ITRONは、複数のプロセッサ上に実装することを目的としたOSであるが、具体的な実装設計は、プロセッサ毎に独立して行なうことを基本方針としている。
標準OSといった場合に最初に考えられるのは、まずプロセッサ間で共通の仮想アーキテクチャを想定し、その上にITRONを実装することである。しかし、当然のことながら、この方法では、プロセッサのネイティブなアーキテクチャと想定された仮想アーキテクチャとの間のギャップが、そのまま性能の低下につながる。リアルタイムOSとしての最高の性能を得るためには、どうしてもプロセッサの能力を最大に生かすようなアー

キテクチャが必要であり、マシンやOSの極端な仮想化は避けなければならない。さらに、組み込み用という用途を考えると、オブジェクトの互換性に対する要求はあまり強くはない。むしろ、システムコールの名前や機能の統一といった、教育の面からの互換性が重要なことが多い。
そこで、ITRONでは、OSの仕様のうちで何が標準化できるかを検討し、プロセッサ間で共通の仕様とプロセッサに依存した仕様とを明確に分離することを行なった。そして、標準化の難しい部分を無理に標準化することは行なわず、標準化、仮想化による性能の低下を避けるようにした。

・豊富な機能を持つ。
ITRONでは、いろいろな機能を持つシステムコールが豊富に用意されている。例えば、ITRONでは同期手段としてイベントフラグ、セマフォ、および各タスクに付随した同期機構（slp_tsk, wup_tsk　システムコールで使用）など、考えられるほとんどのメカニズムを提供している。このため、応用に最適なメカニズムを選択することによる性能向上やプログラムの書きやすさが期待できる。

・高速な応答が可能である。
リアルタイム応用において応答速度に特に影響する部分は、タスクの切り替え（ディスパッチング）を行なう部分と割込みハンドラを起動する部分である。後で述べる適応化と関連するが、ITRONでは、ディスパッチ時に入れ換えを行なう汎用レジスタを指定することにより、高速のディスパッチングができるように考慮されている。
また、外部割込み発生時にはOSを介することなくユーザの定義した割込みハンドラを起動できるようになっており、この部分のオーバーヘッドは基本的にゼロである。ただし、割込みハンドラで使用するレジスタは、ユーザ側で退避する必要がある。

・適応化を考慮している。
ITRONは一つのOSアーキテクチャであるが、組み込み用ということを考慮し、柔軟で適応性の強いアーキテクチャになっている。システムコールの取捨選択、ディスパッチ時に入れ換えを行なうレジスタの指定、実行（可能）状態のタスクがない時の低消費電力モードの使用、システムコール実行におけるエラーチェックの有無、などの選択の自由度がユーザに開放されており、これを用いて適応化を行なうことができる。

例えば、メモリアドレスを指定するパラメータがあった場合、その値がセグメンテーションエラー、バスエラーなどのメモリ関係のエラーを起こす可能性があるかどうかをソフトウエアでチェックすると、システムコールの実行にとってかなりのオーバーヘッドになる。一方、メモリアドレスを指定するパラメータが静的なものであり、かつユーザの書いたプログラムが正しければ、パラメータがエラーを起こすことはないはずである。以上のような状況では、デバッグ時に多少のオーバーヘッドがあっても完全なエラーチェックを行なっておき、デバッグが終了してからエラーチェックをはずすようにするのが一番望ましい。ITRONアーキテクチャでは、エラーチェックの有無にも自由度を持たせているため、このようなことが可能になっている。

・教育を重視している。

ITRONの設計方針の一つとして、教育を重視し、教育も機能の一つと考えるということがあげられる。

ITRONのシステムコールの名前や機能はいくつかの基準に基づいて系統的に決められており、理解が容易になっている。例えば、ITRONのシステムコール名は 'xxx_yyy' 型の7文字、または 'zxxx_yyy' 型の8文字であり、'xxx' が操作の方法、'yyy' が操作の対象（オブジェクト）を表わすのが基本である［図1.1］。xxx,yyyの意味は［表1.1］のようになっている。'z' の付く8文字のシステムコールは、複数の機能を組み合せて実行する複合システムコール、あるいは'z' の付かない7文字のシステムコールから派生したシステムコールであることを表わす。

また、ITRONは複数のプロセッサ上で動作するので、ITRONを使用すれば、複数のプロセッサを使用する場合でも、OSに関する教育は一本化することができる。これも、実際のプログラム開発を考えると大きな利点である。

＊ITRON のシステムコールの名前は、'xxx_yyy' 型の７文字を使うのが原則である。このうち、'xxx' が操作の方法の省略名称を表わし、'yyy' が操作の対象（オブジェクト）の省略目名称を表わす。

［図1.1］ ITRONのシステムコール命名法

省略名	意味
tsk	Task
flg	Eventflag
sem	Semaphore
mbx	Mailbox
mpl	MemoryPool
msg	Message
int	Interrupt
exc	Exception
blk	Block
tim	Time
cre	Create
del	Delete
def	Define
can	Cancel
wup	Wakeup
wai	Wait
ret	Return

［表1.1］ ITRONにおける省略名称の生成規則

ITRONのシリーズ化

ITRONは標準OSであるが、その具体的な実装設計はプロセッサ毎に独立して行われる。そのため、対象となるプロセッサが増えるにしたがい、ITRON標準仕様の中でもいくつかのシリーズ化が行われている。具体的には、現在ITRON1(ITRON Ver 1.11)、ITRON2、μITRONの3つのシリーズが用意されている。

このうち、ITRON1は、インテルiAPX86、モトローラM68000、ナショセミNS32000、日本電気Vシリーズなどのプロセッサに実装されたITRONである。ITRON2は、TRONCHIPなどの新しい高性能プロセッサに載せることを意図したITRONであり、ITRON1と比較すると、機能の追加やITRON2相互間の互換性強化などが行われている。一方、μITRONは、ITRON2とは逆に、シングルチップコンピュータや8ビットプロセッサなどの低コストのプロセッサに実装することを意図したITRONである。

一般に、標準OSとしての互換性の追求と適応化による性能の向上とはトレードオフの関係になるため、両者のバランスをとる必要が生じる。ここで、既存の16ビットプロセッサ向けに両者のバランスをとったのがITRON1であり、新しい32ビットプロセッサ向けに両者のバランスをとったのがITRON2であり、シングルチップコンピュータや8ビットプロセッサ向けに両者のバランスをとったのがμITRONであるということになる。言い換えると、ITRON1の標準化の程度を上げて仕様拡張を行ったのがITRON2であり、ITRON1の適応化の程度を上げて仕様を簡略化したのがμITRONとなる。

第一部 第三章

ITRONの機能

基本概念

タスク

ITRONで言うタスクとは、プログラムの並行処理の単位である。すなわち、一つのタスクの中のプログラムは逐次的に実行が行なわれるのに対して、異なるタスクのプログラムは並行して実行が行なわれる。ただし、並行して実行が行なわれるというのは、アプリケーションプログラマ（OSのユーザ）から見た概念的な動作として、並行した実行が行なわれるということである。物理的には、OS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ことになる。

プロセッサの実行するタスクが切り替わるのは、タスクの状態を遷移させるシステムコールが発行された時や割込みが発生した時である。プロセッサの実行するタスクが切り替わることをディスパッチングと呼び、これを実現するOS内のメカニズムをディスパッチャと呼ぶ。

実行可能なタスクが複数存在した場合、何らかのアルゴリズムに従ってタスクの実行順序を決定する必要がある。OSによるタスクの実行順序の制御メカニズムをタスクのスケジューリングと呼ぶ。ITRONでは、タスクに与えられる優先度を基にした優先度ベースのスケジューリング方式を採用しており、数の小さい方が高い優先度を表わす。ITRONでは必ず優先度の高いタスクから実行され、自タスクよりも高優先度で実行可能状態のタスクが無ければ、自タスクは必ず実行状態になる（絶対優先度）。また、マルチプロセッサ構成でない場合は、優先度の高いタスクが実行中である限り、それより優先度の低いタスクは全く実行されない。

個々のタスクを区別するために、ITRONでは番号を用いる。これをタスクIDと呼ぶ。生成できるタスクの最大数は、ITRONのインプリメントに依存して、またターゲットシステムのハードウエア構成やコンフィグレーションテーブルの内容に依存して変化する。μITRONのタスクはシステム起動時に静的に生成されるのに対して、ITRON1およびITRON2のタスクは、タスクIDを用いて動的に生成することができる。生成後は、タスクIDまたは生成時に得

られるタスクアクセスキー（ITRON1の場合のみ）を用いてタスクを識別する。

ITRONが多くのタスクの実行を制御し、マルチタスキングを実現するためには、タスクを管理するための情報を集めたものが必要である。これをタスク制御ブロック(TCB)と呼ぶ。TCBの内容は原則としてユーザには見えないものであり、TCB内部の値の設定、変更はシステムコールを通じてITRONが行なう。ITRONの場合、TCBには普通次のような情報が含まれる。ただし、その詳細はインプリメントに依存して変化する。

タスクID番号
タスクの優先度
エントリポインタ（タスクスタートアドレス）
初期ユーザスタックポインタ
例外ハンドラのアドレス
タスクの使用するレジスタのセーブエリア
タスクの状態を示すフラグ群

資源

タスクの実行のために必要な各種の要素を資源と呼ぶ。資源には、プロセッサ、メモリ、入出力装置などのハードウエアやファイル、プログラムなどのソフトウエアが含まれる。大部分の資源は、複数のタスクから同時に使用することができないため、並行に動作する複数のタスクに対して、限られた資源を排他的に割り当てるようにOSが調整しなければならない。資源を排他的に割り当てる手段として、ITRONには各種の同期・通信機能が用意されている。

タスクの状態と動作

ITRONのタスク状態と、状態間の遷移を引き起こすシステムコールを［図1.2］に示す。ITRONのタスク状態は以下の5つに分けられている。

［1］実行 (RUN) 状態
現在そのタスクを実行中であるという状態。

［2］実行可能 (READY) 状態
タスク側の実行の準備は整っているが、そのタスクよりも優先度が高い（同じ場合もあるが）タスクが実行中であるため、そのタスクの実行はできない

という状態。つまり、プロセッサが使用可能になればいつでも実行できる状態。実行可能状態のタスクが複数個あった場合、これらのタスクはプロセッサが空くのを待って待ち行列を作る。これをレディキューと呼ぶ。

［3］待ち (WAIT) 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。すなわち、何らかの条件が満足されるのを待っている状態。

待ち状態をさらに分類すると、そのタスクがシステムコールを発行して自らイベント待ち状態に入った「待ち (WAIT) 状態」と、他タスクによって強制的にイベント待ち状態に入った「強制待ち (SUSPEND) 状態」と、待ち状態のタスクがさらに強制的にイベント待ち状態にされた場合の「二重待ち (WAIT-SUSPEND) 状態」とに分かれる。ITRONでは、「待ち (WAIT) 状態」と「強制待ち (SUSPEND) 状態」を明確に区別しており、自ら「強制待ち (SUSPEND) 状態」になることはできない。

待ち状態からの実行再開は中断した場所から行なわれ、プログラムカウンタやレジスタなどのプログラム実行状態を表現する情報（これをコンテキストと呼ぶ）はそのまま復元される。待ち状態では、セマフォなどの資源は占有したままである（プロセッサを除く）。

［4］休止 (DORMANT) 状態

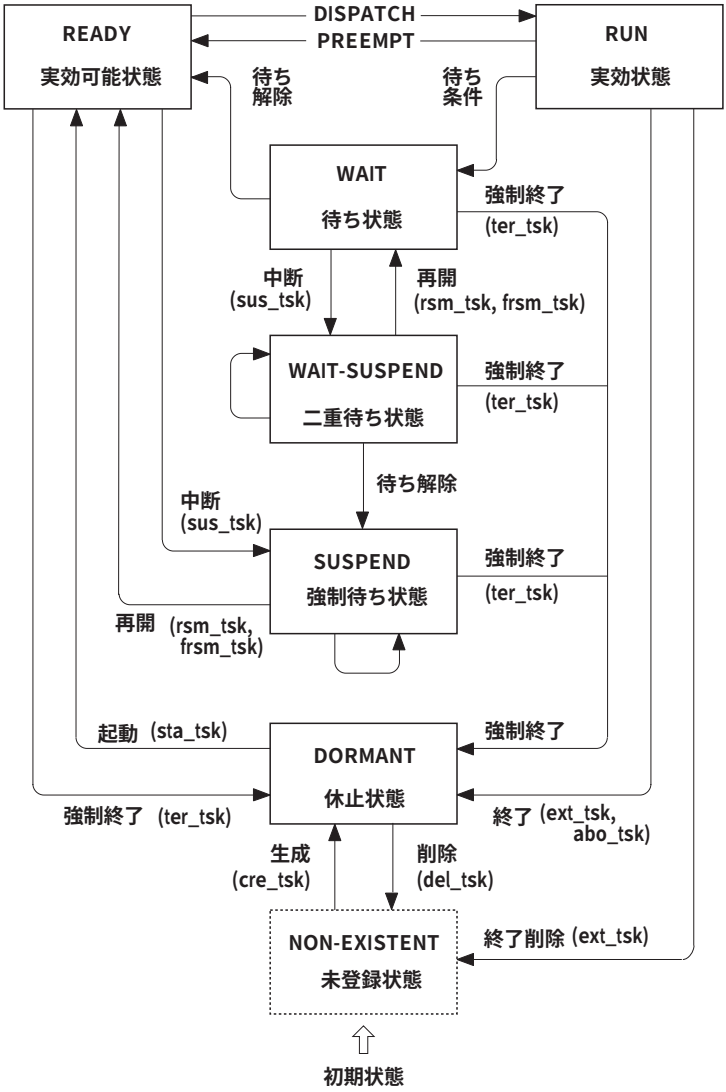
タスクがまだ起動されていない状態、または終了後の状態。タスクを新しく生成するとまずこの状態になる。

待ち状態との相違点は、資源をすべて解放している点、レジスタやプログラムカウンタなどのコンテキストが実行再開時に初期化される点、そのタスクに対する各種のタスク操作命令（wup_tsk,ter_tsk,sus_tsk,chg_pri システムコールなど）がエラーになる点である。

［5］未登録 (NON-EXITENT) 状態

タスクがまだ生成される前、または削除された後の、システムに登録されていない仮想的な状態である。

なお、μITRONの場合は、このうち、［5］の未登録 (NON-EXITENT) 状態を持たない。また、［3］の一部である強制待ち (SUSPEND) 状態、二重待ち (WAIT-SUSPEND) 状態と、［4］の休止 (DORMANT) 状態を持つかどうかはインプリメント依存となっている。ITRON1とITRON2では、［1］～［5］の全部の状



＊μITRONの場合、強制待ち状態(SUSPEND)、
二重待ち状態(WAIT-SUSPEND)、
休止状態 (DORMANT)はインプリメント依存である。
また、未登録状態(NON-EXISTENT)は持たない。

[図1.2] ITRONのタスク状態遷移図

	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	slp_tsk 実行状態 ⇩ 待ち状態	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	ext_tsk, abo_tsk 実行状態 ⇩ 休止状態	ter_tsk 実行可能、待ち状態 ⇩ 休止状態
タスクの消除	exd_tsk 実行状態 ⇩ 存在しない状態	del_tsk 休止状態 ⇩ 存在しない状態

＊ ITRONでは、タスクの状態遷移を明確にすることと、システムコールの理解を容易にすることを考え、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。これは、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1.2］自タスク、他タスクの区別と状態遷移図

態を持つことが必須となっている。

ITRON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1.2］。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。

タスクのスケジューリング

ITRONでのタスクスケジューリングは、タスクの優先度を基準にして行なう。また、同じ優先度のタスク間では、FCFS(First Come First Service)方式と呼ばれるスケジューリングを行なう。この様子を、［図1.3］の例で説明する。

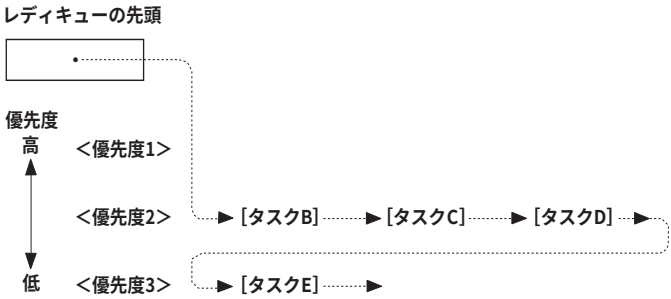
［図1.3(a)］は、優先度1のタスクA、優先度3のタスクE、優先度2のタス

クB、C、Dをこの順に起動した後の、レディキューの状態である。現在実行可能状態にある最も優先度の高いタスクはタスクAであるから、タスクAのみが実際に実行される。ITRONでは、タスクの優先度は絶対的な意味を持っており、タスクAが実行中であれば、優先度の低い他のタスクは全く実行されない（マルチプロセッサ構成の場合を除く）。

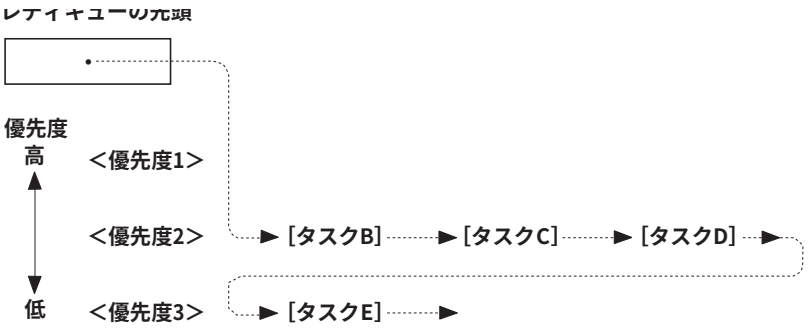
さらに、この状態で外部割込みが発生した場合、割込みハンドラから戻る時点で再度スケジューリングが行なわれる。しかし、割込みハンドラ内でタスクBよりも優先度の高いタスクが実行可能状態に移っていない限り、割込みハンドラから戻った後もやはりタスクBが実行される。つまり、外部割込みの発生によって、タスクBが実行権を奪われるということはない。このような仕様になっているのは、外部割込みの発生と、その時に実行中であったタスクとの間には、直接の関係がないためである。

しかし、一旦待ち状態または休止状態に移行したタスクが、再度実行可能状態に移行した場合には、そのタスクは、その優先度のレディキューの最後尾に付く。また、ITRONでは同一優先度のレディキューを回転させるシステムコールrot_rdqが用意されているので、それを発行することにより、タスクBをレディキューの最後尾にもっていくことも可能である。

同一優先度のタスクに対するスケジューリング方式を、FCFS方式ではなく、一定時間の経過により別のタスクを実行するラウンドロビン方式にしたい場合は、これをユーザ側で実現することもできる。そのためには、タイマ割込みを使って割込みハンドラを一定時間毎に起動し、その中でrot_rdqシステムコールを発行すればよい。



[図1.3(a)]　始めのレディキューの状態



[図1.3(b)] タスクAが待ち状態になった後のレディキューの状態

なお、タスクの優先度は、システムコールchg_priを用いて動的に変化させることができる。

非タスク部実行中のシステム状態

ITRONの上で動くタスクのプログラミングを行なう場合には、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、ITRONでは、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までユーザが行なう場合がある。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ITRONのシステム状態について説明を行なう。

ITRONのシステム状態は、[図1.4] のように分類される。

[図1.4] で示されている状態のうち、「過渡的な状態」には、OS実行中（システムコール実行中）の状態が含まれる。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。

「タスク独立部」と「準タスク部」は、各種のハンドラなど（タスク部として実行される例外ハンドラを除く）を含むものである。このうち、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」となり、外部割込みによって起動される割込みハンドラが「タスク独立部」となる。「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。

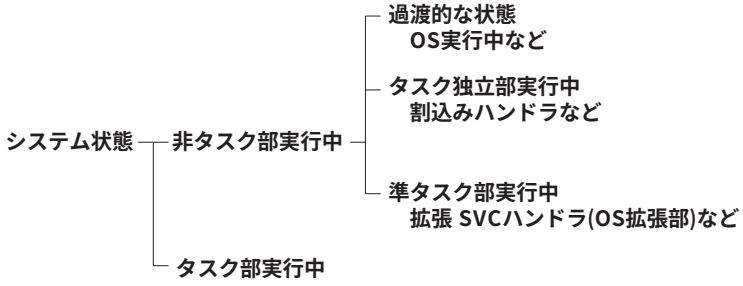
ITRONでは、「過渡的な状態」、「タスク独立部」、「準タスク部」を合わせて「非タスク部」と呼んでいる。これ以外で、普通にタスクのプログラムを実行している状態が「タスク部実行中」の状態である。

タスク独立部と準タスク部

ITRONの中で「タスク独立部」となるのは、割込みハンドラやタイマハンドラの部分である。タスク独立部の特徴は、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（ディスパッチング）は起らない。ディスパッチングは必要になっても、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なうと、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1.5］に示す。

［図1.5］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



［図1.4］ ITRONのシステム状態の分類

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、実質的に割込みハンドラが書けなくなってしまう。ITRONで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。

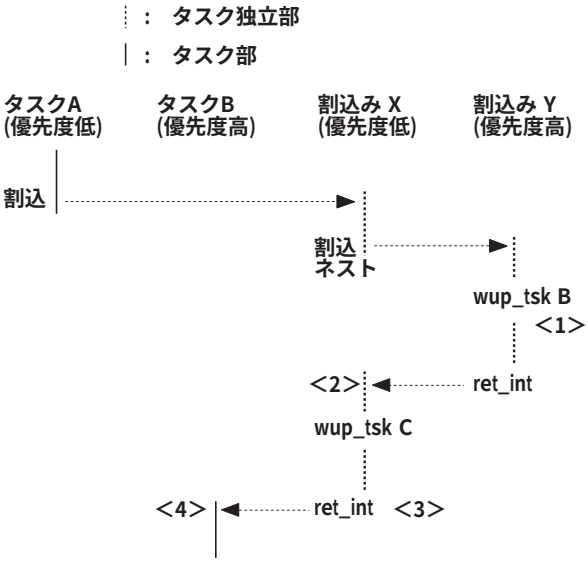
これに対して、ITRONの中で「準タスク部」となるのは、主にユーザがOSを拡張した部分である。具体的には、拡張SVCハンドラがこれに含まれる。ただし、プロセッサによっては、現在「準タスク部」を持たないものがあり、その場合は拡張SVCハンドラは「タスク独立部」になる。

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、割込みハンドラはすべてのタスクに優先して実行されるのに対して、OS拡張部などの準タスク部は、非タスクであるにもかかわらずタスク部の実行に優先して行なわれるとは限らない。

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、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。割込みハンドラの最後の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）の中でタスクB（優先度2＝高）に対する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、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



[図1.5] 割込みのネストと遅延ディスパッチ

スク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。

ID番号とアクセスキー

ITRONでは、タスクなどシステムコールの操作対象となるものを総称してオブジェクトと呼ぶ。オブジェクトには、タスクのほかにメモリプールやイベントフラグ、セマフォ、メイルボックスなどの同期、通信機構が含まれる。各オブジェクトに対して、そのオブジェクトの管理テーブル（タスクの場合はTCB）がOSによって用意される。オブジェクトの識別はID番号を用いて行なわれるので、オブジェクトを最初に生成する時は、ID番号をパラメータとしてオブジェクトを指定する。しかし、その後何度もそのオ

ブジェクトをアクセスする場合にID番号をパラメータとして参照すると、ID番号から管理テーブルアドレスへの変換が毎回生じるため、性能が落ちる場合がある。そこで、オブジェクトを生成した時に、そのオブジェクトをアクセスするためのキー値をOSのリターンパラメータとして返すようにし、以後、そのオブジェクトのアクセスをする場合には、生成時に返されたキー値をパラメータとして用いる方法がある。このような目的で用いられるキー値をITRONでは「アクセスキー」と呼ぶ。

通常、アクセスキーとしては、オブジェクト管理テーブルのベースアドレスが用いられることが多いため、アクセスキーは「アクセスアドレス」とも呼ばれる。ITRONでは、「アクセスキー」と「アクセスアドレス」を全く同義のことばとして用いている。ただし、アクセスキーとしてID番号をそのまま用いる場合や、オブジェクト管理ブロックのセグメント値を用いる場合、OSのデータ構造に依存したハッシュ値を用いる場合などもあるため、必ずしもアクセスキーがオブジェクト管理テーブルのアドレスを示すとは限らない。

なお、μITRONの場合は動的なオブジェクトの生成を行わず、必要となるオブジェクトは、すべてシステム起動時に静的に生成される。したがって、オブジェクトの識別にアクセスキーを用いる必要はなく、ID番号によってオブジェクトを指定する仕様になっている。また、ITRON2では、OSのインプリメントが違ってもアプリケーションの互換性を向上させるという意図から、やはり、ID番号によってオブジェクトの指定を行う仕様になっている。したがって、アクセスキーを用いているのはITRON1のみである。

ITRONでの標準化の方針と互換性

ITRONは、複数のプロセッサに実装される標準OSである。しかし、標準化や仮想化に伴う性能の低下を防ぐため、それぞれのプロセッサのアーキテクチャを生かす方向で、仕様にバリエーションを持たせている部分がある。また、ITRONの場合は、ターゲットシステムのハードウエアによって実行環境が大きく異なるので、OSのみに強い規定を設けても、オブジェクトレベルの互換性をとることは難しい。したがって、むしろOSでの規定は緩いものにし、インプリメント毎の効率向上を目指すという考え方をとっている。そのため、対象プロセッサやITRONのインプリメントが異なれば、ITRON1同士やμITRON同士であっても、完全な互換性は保証されない。仕

様の細かい点ではインプリメント依存の項目があり、ITRON2、ITRON1、μITRONの順でインプリメント依存項目が多くなっている。また、同様の理由により、ITRON1とITRON2や、ITRON1とμITRONも完全互換ではない。

ITRON1およびITRON2の仕様は、プロセッサ間で共通の部分、プロセッサに依存する部分、インプリメントに依存する部分の三段階に分けられている。ここでは、こういったITRONでの標準化のレベルについて説明する。

ITRONの仕様のうち、プロセッサ間共通の仕様となっているものには、次のようなものがある。

システムコールの機能と名称
パラメータの種類と名称
エラーコードの種類と名前、エラーコードの値
機能コード番号（システムコールを区別する番号）

ただし、これらの点についても、すべての仕様を厳密に実現する必要があるわけではなく、インプリメントの都合で、多少の仕様変更が行われる場合がある。また、ITRON1とITRON2の間では、上記のプロセッサ間共通の仕様に関しても、仕様の追加が行われていたり、仕様が多少違っていたりする場合がある。

これに対して、パラメータの渡し方やビット数、システムコール起動の方法などの仕様は、プロセッサのアーキテクチャの影響を強く受ける部分である。これらの仕様をITRON1あるいはITRON2して統一することも不可能ではないが、それに伴う性能の低下を考えると好ましいことではない。また、プロセッサが異なれば、もともとオブジェクトレベルの互換性が無いので、パラメータの渡し方などを統一してもあまり意味がない。

しかし、これらの仕様は、プロセッサが特定できれば標準化できる仕様である。そこで、ITRONでは、現在多く使用されているインテルiAPX86系、モトローラM68000系、ナショナルセミコンダクタNS32000系のプロセッサについて、具体的にITRONを実装した場合にどのようにするのが望ましいかを考え、その標準的な仕様を提示した。これらの仕様は、プロセッサに依存した点があるため、それを実装するプロセッサの名前をつけてITRON/86（ITRON1をインテルiAPX86にインプリメントしたもの）、ITRON/MMU286（ITRON1をインテルiAPX286にインプリメントしたもの）、ITRON/68K（ITRON1をモトローラM68000にインプリメントしたもの）

ITRON/32（ITRON1をナショセミNS32000にインプリメントしたもの）、ITRON/CHIP（ITRON2をTRONCHIPにインプリメントしたもの）、といった形で呼ばれる。

また、ITRONでは、基本的に外部仕様のみを規定し、インプリメント方法は全くの自由となっている。しかし、システムコール実行の細かな点まで考えると、どうしてもインプリメント方法が外部仕様にまで影響する点が出てくる。そこで、ITRONでは、こういった部分を「インプリメントに依存した仕様」とし、外部仕様ではあるが、ITRONの性能が最も引き出せるようにインプリメンタが自由に決めてよい部分を設けている。インプリメントに依存した仕様には、エラー検出のレベルなどが含まれる。

μITRONの場合には、プロセッサの違いに依存して、OSの仕様の違いもずっと大きくなる。そこで、μITRON仕様全体をガイドライン的なものとし、「プロセッサ間共通の仕様」「プロセッサ依存の仕様」「インプリメント依存の仕様」の三者を厳密に区別することは行っていない。「プロセッサ間共通の仕様」にあたるものが一応提示されているが、その仕様が強い拘束力を持つわけではなく、すべての点が「プロセッサ依存」でもあり「インプリメント依存」でもある。

なお、繰り返しになるが、ITRON仕様が完全互換性を保証するものではないにもかかわらず、ITRONとしてリアルタイムOSの標準仕様を提示している意義としては、次のような点がある。

・システムコール名や用語を統一することによって教育の一本化を図る。
・ITRONのインプリメンタが標準仕様との差を示すことにより、移植の際に変更の必要な部分が明確になる。

システムコールインタフェース

アセンブリ言語とのインタフェース

ITRONの仕様では、アセンブリ言語を使用している場合のシステムコール呼び出し方法、つまりアセンブリ言語とのインタフェース方法は、プロセッサ依存となっている。ただし、ITRON1およびITRON2では、以下のような点について、どのプロセッサにも当てはまる共通の原則を設けている。

・システムコールのパラメータやリターンパラメータは、原則としてレジスタに置く。このために4〜8個程度のレジスタを使用する。
・システムコールの種類を示す機能コードは、通常アキュムレータとして使われているレジスタに置く。
・パラメータの数が多い場合や、データの長さが長い場合は、メモリ上にパケットを作る。その先頭アドレスは、通常アドレスレジスタまたはインデクスレジスタとして使われているレジスタに置く。
・リターンパラメータとして使用しないレジスタはすべて保存されるのが望ましい。

ITRON/86(ITRON/MMU286)、ITRON/68K、ITRON/32、ITRON/CHIPは、これらの原則を踏まえた上で、iAPX86(iAPX286)、M68000、NS32000、TRONCHIPに具体的にITRONを載せたものである。

なお、μITRONの場合にも同じ原則を当てはめることは可能であるが、原則に合わせるかどうかよりも性能を優先する必要があるため、上記の原則はあまり意味を持たない。

言語Cとのインタフェース

ITRONでは、マシンの仮想化を行なっていないため、当然のことながら、プロセッサが異なればオブジェクトコードの互換性はない。異なるプロセッサ間で、ITRONのアプリケーションプログラムの移植性を確保するためには、高級言語を使用してプログラムを記述する必要がある。ITRONでは、現在、言語Cを標準的な高級言語として使用することにしており、言語Cか

らITRONの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。ITRONと言語Cとのインタフェースについては、以下のような原則が設けられている。

・システムコールはすべてCの関数として定義され、システムコールのエラーコンディションを関数の戻り値として返す。エラーコード以外のリターンパラメータは、原則としてポインタを用いて返される。
・パラメータの順序は、リターンパラメータへのポインタ、その他のパラメータの順となる。このほか、ITRON1では、命令オプションが最初のパラメータとして置かれる場合がある。

言語CでのITRONシステムコールインターフェースは、基本的にはプロセッサに依存しないものになっている。しかし、割込み処理や例外処理などシステムコールの機能自体がプロセッサによって違っている部分、およびITRON/86(ITRON/MMU286)において、iAPX86系のプロセッサのセグメントを使ったアドレッシングに関係している部分については、プロセッサによって多少仕様の異なる部分がある。もっとも、これはITRONの問題というよりも、プロセッサのアーキテクチャの問題である。また、ITRON1とITRON2の間でも、言語Cのシステムコールインターフェースに多少の相違がある。

第一部 第四章

μITRONとITRON2

μITRONの概要

家電製品や自動車などへの組み込みを目的とした制御用のシングルチップコンピュータでは、ROM容量やRAM容量の制限、量産時のコストダウンに対する要求などから、これまで標準OSを用いることは少なく、アプリケーション側でOSの機能まで包含してプログラミングを行うのが一般的であった。μITRONは、こういったケースで利用することを目的としたOSの体系であり、教育の一本化といった標準OSのメリットを生かしつつ、メモリ容量や性能の面では、標準化によるオーバーヘッドを回避することを可能としたものである。

μITRONの基本方針

最近では、いくつかの半導体メーカーが、制御用のシングルチップコンピュータに入れるプロセッサや、ASIC核となるプロセッサとして、オリジナルアーキテクチャの8〜16ビットプロセッサを作るケースが増えてきた。これらのプロセッサは、リアルタイム、組み込みの用途で使用されることが多いため、OSとしてITRONを載せたいという要求が強い。

しかし、ITRONは汎用の16ビットプロセッサを対象として設計されているため、シングルチップコンピュータや8ビットプロセッサに実装するには機能が多過ぎ、ROM容量やRAM容量に不足をきたす場合がある。ITRONでは、適応化によって不要な機能を削除できるようになっているが、これは、個々のアプリケーションに対する適応化や機能のサブセット化を意味するものであり、OS全体として勝手に機能を減らしても良いわけではない。OSでサポートされない機能があると互換性が失われてしまうため、ITRONでは、最低限インプリメントの必要な機能を定めているが、それだけでも多くの機能が含まれている。

また、適応化のために機能のサブセット化を行う場合には、システムコール全体を削除するだけではなく、システムコールの一部の機能のみを削除したいことがある。例えば、ITRONのセマフォは計数型のセマフォであるため、セマフォ待ちシステムコール wai_sem (Wait Semaphore) では、パラ

メータ rcnt により、資源獲得値を自由に指定できるようになっている。ここで、セマフォの機能全体を削除するのではなく、資源獲得値の指定機能のみを削除して機能のサブセット化を行い、資源獲得値を1に固定した場合、wai_sem システムコールの rcnt パラメータは不要になる。同様に、セマフォ待ちにおけるタイムアウトの機能を削除した場合には、タイムアウトの時間指定を行うパラメータ tmout も不要になる。パラメータを減らすことは、ROM容量やRAM容量の節約に効果があるので、ROM容量やRAM容量の制限の厳しいシングルチップコンピュータの場合に大きな意味がある。

ところが、パラメータの有無はシステムコールインタフェースに影響するので、高級言語による互換性を必要とする場合には、勝手にパラメータを削除するわけにはいかない。ITRONは、適応化を一つの特長としたOSであるが、一方では高級言語による互換性も考慮しているため、無制限の適応化を行うことができるわけではない。

μITRONは、こういったITRONの適応性の限界を越えることを狙ったアーキテクチャである。すなわち、ITRONでは、適応化よりも標準化に重点を置いた設計になっていたが、μITRONでは、標準化よりも適応化に重点を置くことにより、RAM容量やROM容量の極端に少ないシングルチップマイクロコンピュータやプロセッサの絶対能力が低い8ビットプロセッサでも、十分な性能を発揮できるようにした。

具体的には、ITRONで標準化を行っていた仕様のうち、システムコールインタフェースやパラメータの有無などいくつかの点について、推奨仕様あるいはインプリメント依存仕様に格下げを行い、インプリメントの際の自由度をずっと大きくした。また、OSレベルでの機能のサブセット化を許し、OSのインプリメンタが、プロセッサアーキテクチャに合った機能や必要性の高い機能を自由に選択できるようにした。

一般に、標準化と適応化はトレードオフの関係になる。μITRONでは、OSレベルでの強力な適応化ができる分だけ、標準化の程度は弱くなっている。OSによってインプリメントが行われていない機能があったり、システムコールインタフェースの詳細仕様に差異があったりするため、高級言語レベルでも完全な互換性は保障されない。

しかし、μITRONを実装するプロセッサは千差万別なので、互換性に関して言えば、μITRONの仕様の違いによる問題よりも、プロセッサのアー

キテクチャの違いの方がずっと大きな問題となる。プロセッサが異なると、ワード長やアドレス空間の大きさ、アドレッシング方式等の違いも生じるため、OSのシステムコールを全く利用しないユーザプログラムであっても、高級言語レベルの互換性が保障されているわけではない。また、組み込みのコントローラ等に使用するというμITRONの用途を考えると、接続されているハードウエアや入出力機器の差による影響も大きいため、プログラム移植の際には、プログラムの見直しや修正がどうしても必要になる。したがって、μITRONで高級言語レベルの互換性を保障しなくても、それほど大きな問題とはならない。

この場合、μITRONによる標準化のメリットは、システムコール名称と機能との対応といった教育面でのメリットに絞られることになる。しかし、プログラマの教育コストやプログラムの生産性向上を考えた場合、教育面での標準化の効果は大きなものがある。上で述べたように、μITRONの適用される用途では、必要に応じてプログラムの書き直しを行うことが多いため、プログラム自体の互換性よりも、教育面での互換性の方が重要になる。μITRONに準拠したOSであれば、どの名称のシステムコールでどういう機能を実現するかという点がITRONも含めて標準化されているので、プロセッサが変わっても、システムコールと機能との対応付けが容易である。また、ITRONを理解していればμITRONの理解が容易であり、逆にμITRONを理解していればITRONの理解も容易になる。

結局、μITRONは、一つのOSの仕様を指すものではなく、OSの仕様設計を行ない、システムコールの命名を行うためのガイドラインであると言える。μITRONでは、プロセッサ毎あるいはアプリケーション毎に、一つのガイドラインに沿った別々のOS仕様が存在しており、それらのOSがμITRONというOSのファミリを形成していると考えられる。μITRONは、複数のOSの仕様を包含したOS群とも呼ぶべきものであり、「μITRON準拠」であるとは、μITRONで決めたシステムコール名称を使っているという意味になる。

μITRON設計の具体化

μITRONは、基本的にはITRONで定めている機能をサブセット化し、仕様のインプリメント依存部分を多くしたものである。しかし、単にITRONの仕様の一部を抜き出すことによってサブセット化を行うと、機能がアン

バランスになったり、サポートしない機能に対するパラメータが存在して無駄が生じたりする。したがって、μITRONのシステムコールの名称やインタフェースは、必ずしもITRONと同一になっているわけではなく、必要な部分については仕様の見直しを行なっている。全体的な方向としては、仕様の簡素化を行い、汎用性よりも高速性を目指すようにしている。

μITRONの条件

μITRONの用途を考えると、μITRONとして必須の機能やシステムコールを決めても、結局中途半端なものになってしまう。そこで、μITRONは、「機能とシステムコール名称との対応付けのガイドライン」といった位置付けのものに撤することにし、μITRON必須の機能は次の2つだけとしている。

＜1＞RUN, READY, WAIT の3つのタスク状態を持つ。（マルチタスクOSである）
インプリメントによっては、このほかに DORMANT, SUSPEND の状態を持っても良い。

＜2＞割込みハンドラの定義が可能であり、割込みハンドラからシステムコールを発行することによって、タスクを起床する手段が存在する。（リアルタイムOSである）
割込みハンドラの定義は、静的に（システムジェネレーション時や初期化時に）、または def_int システムコールにより行う。

簡単に言えば、「システムコール名称をITRONに合わせたリアルタイムOS」はすべてμITRONということになる。極端な話、タスクを待ち状態にするシステムコール(slp_tsk)と、割込みハンドラから待ち状態のタスクを起床するシステムコール（wup_taskまたはret_wup）の2つのシステムコールしか持たないOSであっても、μITRONに含まれるわけである。

この場合、μITRONを利用する方から見ると、μITRON準拠のOSだからといって、どの機能が利用できるかが保障されているわけではない。しかし、前にも述べたように、μITRONであれば、システムコール名と機能との対応付けが保障されている。すなわち、ある機能が欲しい時にどういうシステムコールを使ったら良いかということや、プログラム中のシステムコールを見て何を行っているかということを容易に知ることができる。こういった「教育の互換性」は、実際にプログラミングを行う上で非常に重要なことである。

ITRON2の概要

ITRON2の仕様は、ITRONを高性能な32ビットプロセッサに実装することを前提として、従来のITRON(ITRON1)の仕様をバージョンアップしたものである。

ITRON2の基本方針

ITRON2では、ITRON間の互換性やアプリケーションプログラムの移植性を高めることを大きな目的としており、従来のITRONの基本方針に加えて、次のような点を考慮して設計されている。

・標準化のレベルアップ
プロセッサの性能が上がっているため、標準化、仮想化のレベルを上げ、互換性を高める。例えば、従来プロセッサ依存となっていた例外管理機能についても、標準仕様を設ける。
ただし、ITRON2になっても、バーチャルマシンアプローチはとらず、最高性能のものを求めるという方針はITRON1と同じである。

・TRONファミリのOSとしての統合性
名称や用語などについて、BTRON、CTRONとITRONとの整合性を強化する。

・システムコールのレベル分け
ITRON2では、ITRON1と比べてかなりの機能が追加されている。そこで、ユーザにとっての理解しやすさやITRON1からの移行のしやすさを考えて、システムコールのレベル分けを行なう。具体的には、基本機能≪I1≫、拡張機能≪I2≫、システム操作用機能の3段階に分けられている。このうち、基本機能≪I1≫だけを見ると、ほぼITRON1に相当する機能となる。

ITRON2設計の具体化

ここでは、ITRON2設計の際の考え方や、ITRON1とITRON2の具体的な相違点について、項目別に説明する。

・システムコールの分割と名称

ITRON2では、μITRONとの整合性強化、命令オプションの整理、機能追加、システムコールの取捨選択による適応化への対応などといった理由により、これまで一つのシステムコールで総称的(generic) に実現されていた機能が、複数のシステムコールに分割されている場合がある。

分割により生まれた派生的な意味を持つシステムコールの名称は、もとのシステムコール名称と関連を持たせるのが望ましい。しかし、従来のxxx_yyy の形のシステムコール名称では、わかりやすい名前を付けるのが難しい。

そこで、従来の形(xxx_yyy) の名称に加えて、zxxx_yyy の形の8文字のシステムコール名称も使うことにする。'xxx', 'yyy' の部分は、従来通り操作方法と対象（オブジェクト）を表わすが、'z' は、そこから派生したシステムコールであることを表わす。

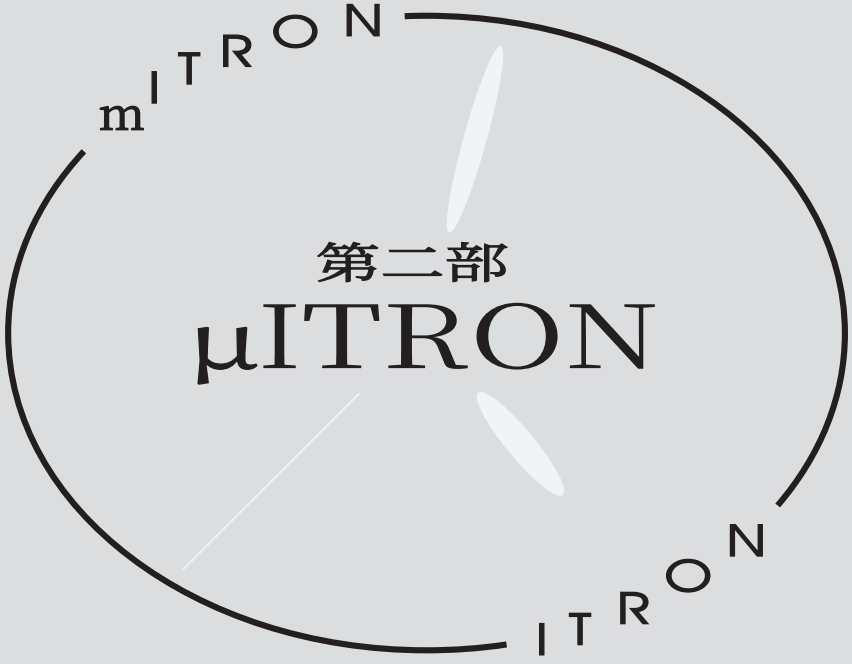
・オブジェクトのアクセス方法

ITRON2では、ITRON1のように、オブジェクト生成時に動的に得られたアクセスキーによってオブジェクトのアクセスを行なうのではなく、実行開始前に確定している値（ID番号など）によってオブジェクトのアクセスを行なう。これは、プログラムの書きやすさと互換性に対する配慮である。

・例外管理

例外のクラス分け、例外ハンドラの多重起動、拡張SVCハンドラに対する例外管理などの仕様を明確化し、例外処理のプログラムについても標準化を促進する。

また、ITRON2の拡張機能(≪I2≫)として、ソフトウエアにより他タスクの例外ハンドラを起動する機能（強制例外）を設ける。ITRON1の終了時処理ルーチンは、例外ハンドラの一つ（終了ハンドラ）として扱う。



第二部
μITRON

第二部 第一章

μITRON概説

この章では、μITRONの仕様設計の考え方、ITRON1とμITRONとの関係などに関して説明を行う。

μITRONの設計の考え方

最近では、いくつかの半導体メーカーが、制御用のシングルチップコンピュータに入れるプロセッサや、ASIC核となるプロセッサとして、オリジナルアーキテクチャの8〜16ビットプロセッサを作るケースが増えてきた。これらのプロセッサは、リアルタイム、組み込みの用途で使用されることが多いため、OSとしてITRONを載せたいという要求が強い。

しかし、ITRONは汎用の16ビットプロセッサを対象として設計されているため、シングルチップコンピュータや8ビットプロセッサに実装するには機能が多過ぎ、ROM容量やRAM容量に不足をきたす場合がある。ITRONでは、適応化によって不要な機能を削除できるようになっているが、これは、個々のアプリケーションに対する適応化や機能のサブセット化を意味するものであり、OS全体として勝手に機能を減らしても良いわけではない。OSでサポートされない機能があると互換性が失われてしまうため、ITRONでは、最低限インプリメントの必要な機能を定めているが、それだけでも多くの機能が含まれている。ITRONでの適応化の考え方を［図2.1］に示す。

また、適応化のために機能のサブセット化を行う場合には、システムコール全体を削除するだけではなく、システムコールの一部の機能のみを削除したいことがある。例えば、ITRONのセマフォは計数型のセマフォであるため、セマフォ待ちシステムコール wai_sem (Wait Semaphore) では、パラメータ rcnt により、資源獲得値を自由に指定できるようになっている。ここで、セマフォの機能全体を削除するのではなく、資源獲得値の指定機能のみを削除して機能のサブセット化を行い、資源獲得値を1に固定した場合、wai_sem システムコールの rcnt パラメータは不要になる。同様に、セマフォ待ちにおけるタイムアウトの機能を削除した場合には、タイムアウトの時間指定を行うパラメータ tmoutも不要になる。パラメータを減らすことは、ROM容量やRAM容量の節約に効果があるので、ROM容量やRAM容量の制限の厳しいシングルチップコンピュータの場合に大きな意味がある。

ところが、パラメータの有無はシステムコールインタフェースに影響するので、高級言語による互換性を必要とする場合には、勝手にパラメータ

［図2.1］ITRONにおける適応化の考え方

を削除するわけにはいかない。この様子を［図2.2］に示す。ITRONは、適応化を一つの特長としたOSであるが、一方では高級言語による互換性も考慮しているため、無制限の適応化を行うことができるわけではない。

μITRONは、こういったITRONの適応性の限界を越えることを狙ったアーキテクチャである。すなわち、ITRONでは、適応化よりも標準化に重点を置いた設計になっていたが、μITRONでは、標準化よりも適応化に重点を置くことにより、RAM容量やROM容量の極端に少ないシングルチップマイクロコンピュータやプロセッサの絶対能力が低い8ビットプロセッサでも、十分な性能を発揮できるようにした。

ITRON1 仕様におけるwai_semのインタフェース

```
ercd = wai_sem (option, a_sem, cnt, tmout) ;
        option :  タイムアウト検出の有無の指定
        a_sem : セマフォアクセスキー（セマフォID）
        cnt :     資源獲得値
        tmout :   タイムアウト時間
        ercd :    エラーコード
```

資源獲得値とタイムアウト指定が不要な場合のwai_semのインタフェース

（ITRON1仕様からは逸脱する）

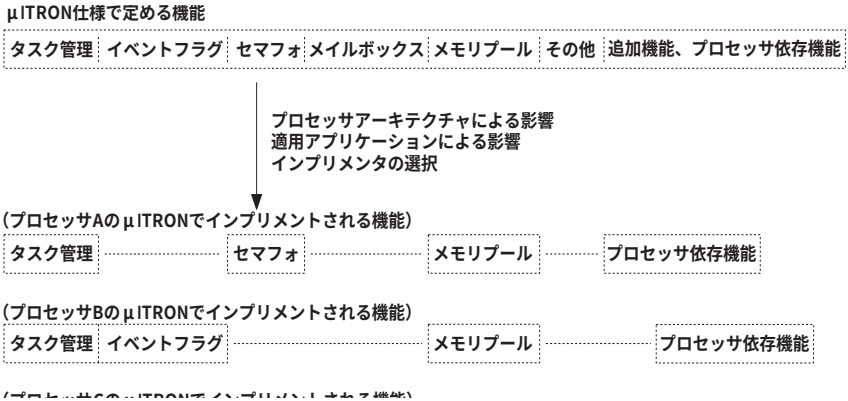
```
ercd = wai_sem (a_sem) ;
        a_sem : セマフォアクセスキー（セマフォID）
        ercd :    エラーコード
```

［図2.2］セマフォ待ちシステムコールwai_semのインタフェース例

具体的には、ITRONで標準化を行っていた仕様のうち、システムコールインタフェースやパラメータの有無などいくつかの点について、推奨仕様あるいはインプリメント依存仕様に格下げを行い、インプリメントの際の自由度をずっと大きくした。また、OSレベルでの機能のサブセット化を許し、OSのインプリメンタが、プロセッサアーキテクチャに合った機能や必要性の高い機能を自由に選択できるようにした。μITRONでの適応化の考え方を［図2.3］に示す。

一般に、標準化と適応化はトレードオフの関係になる。μITRONでは、OSレベルでの強力な適応化ができる分だけ、標準化の程度は弱くなっている。OSによってインプリメントが行われていない機能があったり、システムコールインタフェースの詳細仕様に差異があったりするため、高級言語レベルでも完全な互換性は保障されない。

しかし、μITRON を実装するプロセッサは千差万別なので、互換性に関して言えば、μITRONの仕様の違いによる問題よりも、プロセッサのアーキテクチャの違いの方がずっと大きな問題となる。プロセッサが異なると、ワード長やアドレス空間の大きさ、アドレッシング方式等の違いも生じるため、OSのシステムコールを全く利用しないユーザプログラムであっても、高級言語レベルの互換性が保障されているわけではない。また、組み込みのコントローラ等に使用するというμITRONの用途を考えると、接続されているハードウエアや入出力機器の差による影響も大きいため、プログラム移植の際には、プログラムの見直しや修正がどうしても必要になる。したがって、μITRONで高級言語レベルの互換性を保障しなくても、それほ



［図2.3］ μITRONにおける適応化の考え方

ど大きな問題とはならない。

この場合、μITRONによる標準化のメリットは、システムコール名称と機能との対応といった教育面でのメリットに絞られることになる。しかし、プログラマの教育コストやプログラムの生産性向上を考えた場合、教育面での標準化の効果は大きなものがある。上で述べたように、μITRONの適用される用途では、必要に応じてプログラムの書き直しを行うことが多いため、プログラム自体の互換性よりも、教育面での互換性の方が重要になる。μITRONに準拠したOSであれば、どの名称のシステムコールでどういう機能を実現するかという点がITRONも含めて標準化されているので、プロセッサが変わっても、システムコールと機能との対応付けが容易である。また、ITRONを理解していればμITRONの理解が容易であり、逆にμITRONを理解していればITRONの理解も容易になる。

結局、μITRONは、一つのOSの仕様を指すものではなく、OSの仕様設計を行ない、システムコールの命名を行うためのガイドラインであると言える。μITRONでは、プロセッサ毎あるいはアプリケーション毎に、一つのガイドラインに沿った別々のOS仕様が存在しており、それらのOSがμITRONというOSのファミリを形成していると考えられる。μITRONは、複数のOSの仕様を包含したOS群とも呼ぶべきものであり、「μITRON準拠」であるとは、μITRONで決めたシステムコール名称を使っているという意味になる。

本章の以下の部分では、μITRON設計の実際について具体的に述べ、またμITRONにおける標準化の意味について説明を行っている。

μITRON設計の実際

μITRONは、基本的にはITRONで定めている機能をサブセット化し、仕様のインプリメント依存部分を多くしたものである。しかし、単にITRONの仕様の一部を抜き出すことによってサブセット化を行うと、機能がアンバランスになったり、サポートしない機能に対するパラメータが存在して無駄が生じたりする。したがって、μITRONのシステムコールの名称やインタフェースは、必ずしもITRONと同一になっているわけではなく、必要な部分については仕様の見直しを行なっている。全体的な方向としては、仕様の簡素化を行い、汎用性よりも高速性を目指すようにしている。

以下では、μITRON設計の際の考え方や、ITRONとμITRONの具体的な相違点について、項目別に説明する。

タスク状態

μITRONは、ITRONと同様にプリエンプティブスケジューリングを行うリアルタイムOSである。タスクに優先度を付ける場合にも、ITRONと同様に、数字の小さい方を高い優先度とする。

μITRONでは、ITRONのようなオブジェクト（タスク、セマフォ等）の動的生成機能は持たないので、ITRONの未登録(NON-EXITENT)状態に相当するタスク状態は無い。それ以外の次の5つのタスク状態を持つ。

［1］実行(RUN)状態
現在そのタスクを実行中であるという状態。

［2］実行可能(READY)状態
タスク側の実行の準備は整っているが、そのタスクよりも優先度が高い（同じ場合もあるが）タスクが実行中であるため、そのタスクの実行はできないという状態。

［3］待ち(WAIT)状態
タスクの実行を継続できる条件が整わないため、実行が止まっている状態。［4］の強制待ち(SUSPEND)状態との違いは、自タスクの発行したシステ

ムコールにより実行が止まっているという点である。

［4］強制待ち(SUSPEND)状態
他タスクによって、強制的に実行を中断させられた状態。
なお、この状態は、［3］の待ち(WAIT)状態と重なる場合がある。
［3］と［4］が重なった状態を「二重待ち(WAIT-SUSPEND)状態」と呼ぶ。

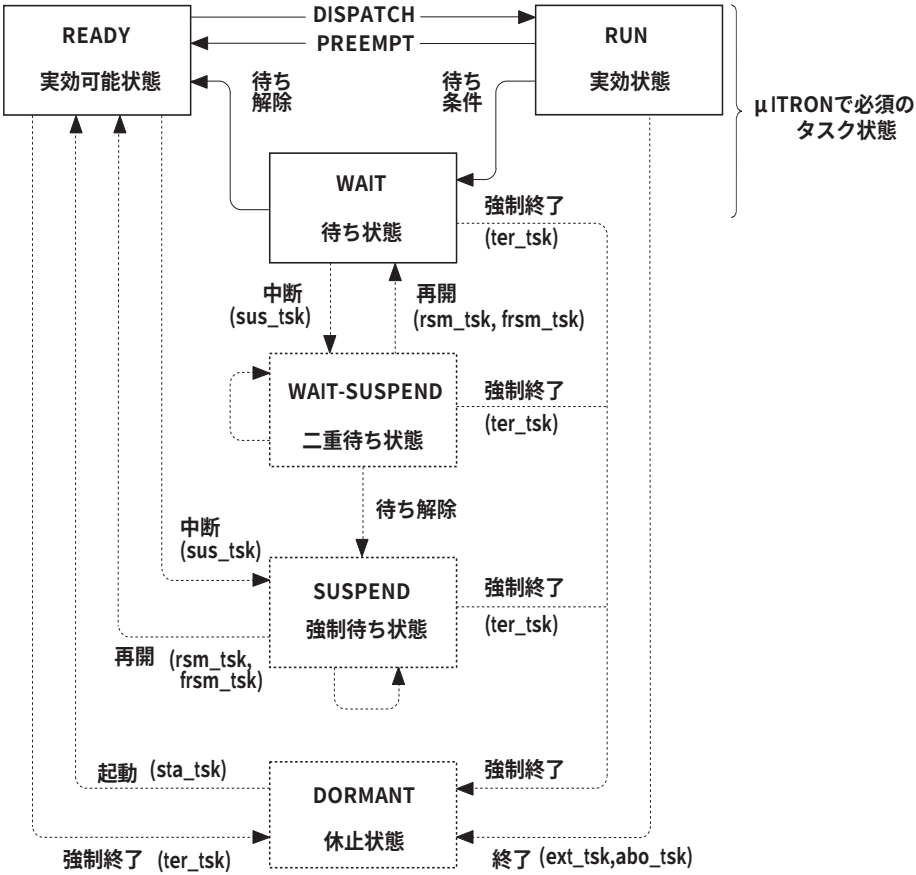
［5］休止(DORMANT)状態
タスクがまだ起動されていない状態、または終了後の状態。
待ち状態との相違点は、資源をすべて解放している点、レジスタやプログラムカウンタなどのコンテキストが実行再開時に初期化される点などである。

このうち、［4］の強制待ち(SUSPEND)状態と［5］の休止(DORMANT)状態は、μITRONでは必ずしもインプリメントしなくても良いタスク状態である。μITRONのタスク状態遷移図を［図2.4］に示す。μITRONのタスク状態は、ITRONのサブセットとなっている。

サブセット化が行われた機能

ITRONの持つ機能のうちの一部については、μITRONの用途に適合するように、サブセット化や仕様の簡素化が行われた。その内容を以下に示す。

・オブジェクトを動的に生成する機能
ITRONでは、オブジェクト（タスク、イベントフラグ、セマフォ、メイルボックス、メモリプール）を動的に生成、削除することが可能であったが、μITRONの場合、オブジェクトは初期起動時に静的に生成されるのが基本である。実行の途中でオブジェクトを生成、削除することは行わない。したがって、オブジェクトの生成や削除を行うシステムコールである cre_XXX(Create OBJECT) と del_XXX(Delete OBJECT) は、μITRONの仕様には含まれていない。これは、オブジェクトの動的な生成や削除によってメモリ管理等が複雑化し、インプリメントの負担がかなり増えるためである。なお、XXX は tsk, flg, sem, mbx, mpl のいずれかを表わす。
μITRONでは、オブジェクトの動的な生成を行わないので、ITRONのように、cre_XXXシステムコールによってID番号からアクセスキーを得、



＊点線の部分はインプリメント依存である。
＊タスクの初期状態は、システムジェネレーション時や初期化時に指定する。
＊待ち条件になるシステムコール：
slp_tsk, wai_tsk, wai_flg, cwai_flg, wai_sem, rcv_tmb, get_blk
＊待ち解除になるシステムコール：
wup_tsk, rel_wai, set_flg, sig_sem, snd_tmb, rel_blk, ret_wup,
iwup_tsk, irel_wai, iset_flg, isig_sem, isnd_msg, osnd_tmd

［図2.4］ μITRONのタスク状態遷移図

それを使ってオブジェクトのアクセスを行うことはできなくなる。アクセスキーとしては静的な値を持たなければならないので、原則として、ID番号をアクセスキーとしてそのまま使用する。
μITRONのID番号は1～2バイトで表現するが、最大数についてはインプリメント依存とする。また、使用できるID番号に制約を設けたり、リンカによってアクセスキーの割り当てを行なう（アセンブラやリンカがID番号からアクセスキーへの変換を行なう）といった方法をとっても構わない。いずれにしても、オブジェクト生成時に動的に得られたアクセスキーによってオブジェクトのアクセスを行なうのではなく、実行開始前に確定している値（ID番号など）によってオブジェクトのアクセスを行うことになる。

・キューイング機能
ITRONの場合、タスクを起床するシステムコール wup_tsk(Wakeup Task) が既に発行されているタスクに対して、もう一度 wup_tsk を発行しても、エラーにはならず、その要求が保持される。これを要求のキューイング機能と呼んでいる。この時、wup_tsk の対象タスクは、待ち状態移行のシステムコール slp_tsk(Sleep Task) を2回実行してもまだ待ち状態にはならず、3回目の slp_tsk の実行でやっと待ち状態になる。ITRONでは、wup_tsk～slp_tskシステムコールによるタスク付属の同期機構や、sus_tsk(Suspend Task)～rsm_tsk(ResumeTask)システムコールによるタスクの実行中断・再開の際に、複数回のキューイングが可能になっている。
しかし、キューイングの機能を実現するためには、要求のカウント数を保持する必要が生じる。これは、ITRONの場合には大きな問題ではないが、少しでもRAM容量を減らしたいμITRONの場合には、実現が難しくなる場合がある。そこで、μITRONでは、wup_tsk システムコールや sus_tsk システムコールにおける複数回のキューイング機能をインプリメント依存とし、1回のキューイングしかできなくても構わないことにした。

・タイムアウト指定機能
ITRONでは待ち状態に入るシステムコールにおいてタイムアウトの指定が可能であった。しかし、タイムアウトの処理はインプリメントの負担が大きいのでμITRONの仕様には含めない。

・イベントフラグ

ITRONのイベントフラグでは、1ワード（16〜32ビット）分のイベントフラグをグループ化して扱っており、指定ビットのセットをOR条件やAND条件で待つことができるようになっていた。しかし、この機能のために、イベントフラグ操作用システムコールのパラメータが増え、性能向上やメモリ容量での負担になる場合があった。そこで、μITRONでは、1ワード（8〜16ビット）のイベントフラグをグループ化する仕様のほかに、1ビットのイベントフラグの仕様を用意し、インプリメント側で選択できるようにした。1ビットのイベントフラグは、機能的には、1ワードのイベントフラグのサブセットになる。1ワードを扱うイベントフラグ操作と、1ビットを扱うイベントフラグ操作のパラメータやリターンパラメータの違いを［図2.5］に示す。

イベントフラグが1ビットの場合

システムコール	パラメータ				
wai_flg	flgid	-	-	-	クリア指定なし
cwai_flg	flgid	-	-	-	クリア指定あり

イベントフラグが1ワードの場合

システムコール	パラメータ			
wai_flg	flgid	waiptn	wfmode	flgptn

flgid :　イベントフラグID
waiptn :　待ち条件を示すビットパターン
wfmode : AND待ち/OR待ちの選択と待ち状態解除時のクリア指定
flgptn :　待ち状態解除時のイベントフラグのビットパターン

＊イベントフラグが1ワードになると、その機能を
生かすために、多くのパラメータが必要になる。

［図2.5］イベントフラグ待ちシステムコールのパラメータ

・セマフォの資源獲得値指定機能
ITRONのセマフォは計数型のセマフォであり、P命令を表わすシステムコール wai_sem (Wait Semaphore)やV命令を表わすシステムコール sig_sem (Signal Semaphore) では、資源獲得値や資源返却値を自由に指定できるようになっていた。しかし、μITRONでは、資源獲得値や資源返却値を1に固定することにより、システムコールのパラメータを減らした。

・可変長のメモリブロック管理機能
ITRONの仕様では、可変長（任意長）のメモリブロック管理機能と固定長のメモリブロック管理機能の両方をサポートしている。しかし、一般に可変長のメモリブロックの管理はインプリメントの負担が大きく、リアルタイム性も損なわれやすい。そこで、μITRONでは可変長のメモリブロック管理機能を含めず、固定長のみをサポートしている。

・例外管理機能
ITRONでは、システムコール実行のエラーにより例外ハンドラを起動するシステムコール例外の機能と、プロセッサの例外処理（ゼロ除算等）によってタスクの例外ハンドラを起動するCPU例外の機能があった。しかし、システムコール例外の機能はシステムコール実行のオーバーヘッドになりやすいので、μITRONの仕様には含めない。また、CPU例外の管理機能も、プロセッサのアーキテクチャに対する依存性が高いため、μITRONの標準仕様には含めない。
ただし、インプリメントによっては、デバッグサポート等の目的で、例外管理機能を独自に追加することも可能である。

システムコールインタフェース

ITRONのシステムコールには、かなり高機能なものや汎用的なものが含まれている。例えば、自ら待ち状態に入るシステムコールでは、必ずタイムアウトの指定ができるようになっている。タイムアウトの検出を行うかどうかは、パラメータの一つである命令オプションで指定する。このような仕様にすると、タイムアウトの有無にかかわらず同じシステムコールが使えるため、ユーザから見てシステムコールの選択が容易になり、OS全体がわかりやすいものになるというメリットがある。

しかし、複数の機能を一つのシステムコールで実現しているため、OSを

インプリメントする方から見ると、システムコール実行時におけるOS内部の判断や分岐が多くなり、性能面でデメリットを生じる場合がある。

また、システムコールの一つ一つが高機能であったり、いろいろな機能を包含していたりすると、システムコール単位で不要な機能を削除する場合でも、その効果が現われにくい。例えば、タイムアウトの検出を行うシステムコールと行わないシステムコールが別々になっていれば、タイムアウトの機能が不要な場合にタイムアウトの検出を行うシステムコールを削除し、プログラム容量を減らすことができる。しかし、タイムアウトの検出を行うシステムコールと行わないシステムコールが一つにまとまっていると、タイムアウトの機能が不要な場合にも、システムコールを削除することができない。この場合、タイムアウトの有無の指定が動的に（パラメータによって）決まるため、不要な機能であっても、その処理プログラムを削除することは難しくなる。

μITRONの場合、ITRON以上に性能面での要求やメモリ容量の制限が厳しく、より高度な適応化を行う必要がある。そこで、ITRONのシステムコールのうち、高度な機能を持つもの、複合した機能を持つもの、パラメータの多いものなどについて見直しを行い、システムコールの機能を分割して機能がプリミティブなものになるように変更を行った。システムコールの機能をプリミティブにすることにより、よりアプリケーションに適応した高性能なプログラムを書くことが可能になり、また、不要な機能を削除することによるメモリ容量の節約を、キメ細かく行うことが可能になった。

実際に分割の行われたシステムコールはあまり多くないが、分割の行われたシステムコールについても、μITRONとITRONとの間で整合性や連続性を確保できるようにする必要がある。すなわち、分割により生まれた派生的なシステムコールの名称は、分割を行う前のシステムコールの名称と関連を持たせるのが望ましい。しかし、ITRONと同じ XXX_YYY（XXXは操作方法、YYYは対象オブジェクト）の形のシステムコール名称だけでは、わかりやすい名前を付けるのが難しい。そこで、従来の形(XXX_YYY)の名称に加えて、μITRONでは、ZXXX_YYY の形の8文字のシステムコール名称も使うことにした。'XXX'、'YYY' の部分は、従来通り操作方法と対象オブジェクトを表わすが、'Z' は、そこから派生したシステムコールであることを表わす。なお、TRON体系全体としては、ITRONの複合システムコールやBTRONの外核でもZXXX_YYY の形のシステムコール名称が使用されている。

μITRONにおいて、具体的に分割やプリミティブ化が行われた機能としては、次のようなものがある。

・ポーリング機能

ITRON1では、資源獲得を行うシステムコールにおいて、資源が確保できなくても待ち状態に入りたくない場合、すなわち資源獲得のポーリングを行いたい場合は、タイムアウト時間を0にしてそのシステムコールを実行すれば良かった。この場合、正常終了により資源が獲得できたことを示し、タイムアウトエラーにより資源が獲得できなかったことを示す。ITRONでは、資源獲得待ち機能、ポーリング機能、タイムアウト機能を一つのシステムコールで実現していたことになる。

このうち、μITRONではタイムアウト機能が外されたが、残りの機能についてもプリミティブ化を行い、システムコールを分離した。分離したシステムコールのうち、資源獲得待ちを行うものはITRONと同じ名称を使用し、ポーリングを行うものは、poll を表わす 'p' の一文字を先頭に付けた名称を使用することにした。

例えば、ITRONやμITRONでは、メッセージの受信を待つシステムコールとして rcv_msg(Receive Message) が用意されているが、このシステムコールでは、メッセージが未着の場合に待ち状態に入ってしまう。それを避けたい場合、ITRONであればタイムアウトを0にして同じ rcv_msg システムコールを実行すれば良いのに対して、μITRONであれば、prcv_msg(Poll and Receive Message) システムコールを実行すれば良い。

ポーリング機能を実現するシステムコールとしては、次のようなものがある。

preq_sem（wai_sem の tmout=0 に相当）
prcv_msg（rcv_msg の tmout=0 に相当）
pget_blk（get_blk の tmout=0 に相当）

なお、ポーリング機能に類似の機能として、オブジェクトや資源の状態を見るシステムコール XXX_sts(Get OBJECT Status) が用意されている。しかし、XXX_sts では、資源が獲得できるかどうかを見るだけで、資源が獲得できる場合でも獲得を行わない。これに対して、ポーリング機能の場合は、資源を獲得できる場合は資源を獲得してしまう。この点で、単にオブジェクトの状態を見る機能と、資源のポーリングを行う機能とは異なっている。

・タスク独立部から発行するシステムコール

ITRONの仕様では、タスク部から発行するシステムコールとタスク独立部（割込みハンドラ）から発行するシステムコールを、特に区別しないようになっていた。ITRONでは、割込みハンドラ中で起ったディスパッチの要求が、割込みハンドラから抜ける時まで遅延させられる（遅延ディスパッチの原則）ため、システムコールがタスクから発行されたか割込みハンドラから発行されたかによって、OS内部の処理が変わってくる。しかし、システムコールはどちらの場合も共通であり、OS内部で、システムコールの発行されたコンテキストがタスクかタスク独立部かを判断するようになっていた。

これに対して、μITRONでは、OS内部でコンテキストを判断するオーバーヘッドを避けて性能を上げるため、タスク独立部（割込みハンドラ）から発行するシステムコールとタスクから発行するシステムコールを分離し、別システムコールとする仕様を用意した。μITRONでタスク独立部から発行するシステムコールを別名称とする場合には、interruptを表わす 'i' の一文字を先頭に付けた名称を使用する。

例えば、ITRONでは、タスクを起床するシステムコール wup_tskは、タスクからでもタスク独立部からでも発行可能である。しかし、μITRONでは、インプリメントによって、タスク独立部からも wup_tsk が発行できる場合と、wup_tsk の代わりに iwup_tsk を発行する場合とがある。OSの負担を減らし、タスク独立部から発行するシステムコールの処理を高速化するためには、タスク独立部から発行するシステムコール iwup_tskとタスク部から発行するシステムコール wup_tsk は分離されている方が良い。一方、ユーザから見ると、タスク部からもタスク独立部からも共通のシステムコール wup_tsk を発行できる方が便利である。μITRONでは、この両者の仕様の選択もインプリメント側で対応できるようにし、プロセッサやアプリケーションに対する適応性を強めた。

・周期起動ハンドラとアラームハンドラの機能

ITRONでは、指定時刻から一定周期でタスクの起床要求を出す機能、すなわち cyc_wup (Cyclic Wakeup)システムコールによるタスクの周期起床の機能が用意されていた。μITRONでは、この機能の代わりに、より仕様を簡素化、プリミティブ化したメカニズムとして、周期起動ハンドラ

とアラームハンドラ（指定時刻起動ハンドラ）の機能を設けた。前者は周期的に起動されるタスク独立のハンドラであり、後者は指定時刻に一回だけ起動されるタスク独立のハンドラである。μITRONにおいて、ITRONのタスク周期起床の機能を実現するためには、ユーザの側で周期起動ハンドラやアラームハンドラのプログラムを書き、その中でタスク起床のシステムコール wup_tsk(Wakeup Task) を発行すれば良い。
なお、実際に周期起動ハンドラやアラームハンドラの機能をサポートするかどうかは、μITRONではインプリメント依存になっている。

μITRONにおける標準化の意味

前に述べたように、μITRONの適用される用途では、OSの仕様に互換性を持たせたとしても、それ以外の部分（プロセッサアーキテクチャやハードウエア等）で互換性を阻害する要因が多い。したがって、μITRONでは、プログラムの完全互換性を確保することはあまり大きな意味を持たず、性能を上げたり、必要メモリ容量を小さくすることの方がずっと重要である。そのため、μITRONをインプリメントする際には、性能に影響しないところや無理のないところのみを標準仕様に合わせて頂くというのが基本的な考え方である。互換性を確保するために、性能を低下させるような無理な標準化を行う必要はない。互換性についての要求が高ければ、μITRONではなくITRONを使って頂けば良い。

以下では、μITRONで標準仕様を提示している項目と、標準仕様の位置付けについて述べる。

・機能セットとシステムコール

システムコールの機能とそれに対応する名称は、μITRONとして標準化されている。ただし、機能のサブセット化は自由である。μITRONの一番大きな目的が、システムコール名称の統一によって教育の互換性を確保することであるため、機能とシステムコール名の対応のみは強く標準化される。システムコール自体を取捨選択してサブセット化することは可能であるが、同じ機能に対しては同じシステムコール名を割り当てなければならない。

μITRONの一部のシステムコールでは、仕様が二本立てになっている部分がある。例えば、μITRONのイベントフラグの仕様では、1ビットのみを扱うイベントフラグの仕様と、1ワード（8〜16ビット）をグループ化して扱うイベントフラグの仕様とが両方用意されており、インプリメント側でどちらかを選択するようになっている。これは、次のような理由による。

現在のシングルチップマイクロコンピュータの規模では、イベントフラグを1ビットとするのが最適な仕様だと考えられる。しかし、デバイス技術の進歩により、今後はμITRONでも1ワード（8〜16ビット）のイベントフラグを実現できる可能性があるし、アプリケーションからの強い要求があれば、多少の無理があっても1ワードのイベントフラグを実現した方が良い場合がある。また、ITRONとの整合性という意味では、最終的には、イベントフラグを1ワード（8〜16〜32ビット）にできる方が望ましい。こういった点やμITRONの設計方針を考えると、μITRONのイベントフラグを1ビットに固定してしまうのはあまり意味がなく、むしろ、イベントフラグを1ビットにした時の仕様と1ワードにした時の仕様をそれぞれ標準化する方が望ましいと考えられる。イベントフラグの仕様が二本立てになっているのは、こういった理由による。

もちろん、ユーザにとっては、仕様が何通りもあるのは好ましいことではない。μITRONでも、仕様を二本立てにしてインプリメンタの選択項目としているのは、最小限の機能に限られている。

・パラメータの種類

原則として標準化し、同じシステムコールは同じパラメータを持つものとする。ただし、アプリケーションに応じて特定の機能を強化したいような場合は、一部のシステムコールについて、パラメータの追加が必要になることがある。例えば、セマフォにおける資源獲得値指定機能は、μITRONでサブセット化されている機能であり、サポートしない方が原則である。しかし、特定のアプリケーションでどうしてもこの機能が必要な場合は、この機能を追加しても構わない。その場合、セマフォ管理システムコールのパラメータにも追加を生じ、μITRON標準のシステムコールパラメータとは変わってくる可能性があるが、そうなってもμITRONの範囲から逸脱するわけではない。このような場合は、標準仕様との相違点について、インプリメント毎に明らかにして頂ければ良い。

・アセンブラインタフェース（パラメータの渡し方）

プロセッサのアーキテクチャやコンパイラの影響を強く受ける部分なので、μITRONでも標準化は行わない。プロセッサ毎に仕様を定めるものとする。

・高級言語（言語C）インタフェース
システムコール名とパラメータが標準化されているので、言語Cインタフェースも原則として標準化する。ただし、パラメータの項で述べたように、特殊な用途ではパラメータの有無が変わる場合があり、そうなると言語Cのインタフェースも変わることになる。標準仕様は提示されるが厳密なものではなく、「弱い標準化」になる。

・機能コード
ITRONでは、システムコール発行時にシステムコールの種類を区別する番号を、機能コードと呼んでいる。μITRONで使用する機能コードについても、ITRONと同様に標準を設けるが、やはり「弱い標準」であり、厳密なものではない。
また、μITRONの場合、プロセッサアーキテクチャや開発環境との関係によっては、トラップ命令（ソフトウエア割込み命令）ではなく、サブルーチンジャンプ命令によって、OS内のシステムコール処理ルーチンに直接ジャンプするケースがある。そのような場合には、機能コードを使用しなくても構わない。

・エラーコード
エラーのニモニックやエラーコードの値についても、μITRONでは「弱い標準化」を行ない、一応の標準仕様を提示している。
一般には、エラーコードを標準化しても、性能向上やオブジェクトサイズ縮小の障害になることは少ないと考えられる。しかし、エラーチェックの厳密さや、複数のエラーが発生した場合の検出順序について細かい規定を行うと、インプリメント方法を制約し、性能にも影響が出る可能性がある。したがって、μITRONでは、エラーチェックに関する細かい規定は設けず、エラーコードを大まかに標準化するだけにとどめている。
なお、BTRONおよびITRON2との整合性を強化するため、μITRONのエラーコードの標準値としては、負の値を使用している。ITRON2のエラーコードの値とμITRONのエラーコードの標準値は共通になっている。

・負の数の扱い
TRONファミリ全体での方針（一種の「TRON作法」）として、負の数はシステム用の数、正の数はユーザ用の数と考えるという原則がある。また、0は特殊な意味を表わすものとしている。例として次のようなものがある。

― ITRONでは、tskid = 0 で自タスクの指定になる。
― ITRON/MMUでは、srcid = (-1) により共通空間を示す。
― BTRONでは、システムコールの関数値が負になったことによりエラーを示す。
― BTRONでは、process_id = 0 により自プロセスを、process_id = (-1) により親プロセスを示す。
― TRONCHIPでは、負のアドレスが SS（共通半空間）、正のアドレスが US（個別半空間）となっている。
― CTRONでは、tmout = (-1) でタイムアウト無しを示す。

μITRONの場合は、個々のプロセッサアーキテクチャの制約が強いため、必ずしもこの原則を適用できるとは限らない。しかし、オブジェクトのID番号や優先度を割り当てる際には、性能を落とさない範囲で、できるだけこの原則に合わせるのが望ましい。例えば、μITRONのカーネルと共に入出力用のタスクを提供する場合は、タスクIDを符号付きの数として扱い、負のID番号（例えば -8〜-5）を入出力用のタスクに割り当てるのが望ましい。

なお、上記のうち、必ずしも標準仕様に合わせなくて良い項目であっても、μITRONのインプリメンタは、どの部分が標準仕様と違っているかということを明確に示す必要がある。これは、プログラムの移植に対して、どこを修正すべきかをはっきりさせるためである。

また、プロセッサによっては、レジスタバンクの切り換え機能や、アイドル状態での低消費電力モードなどの特殊機能が用意されている場合がある。そのような機能は、μITRONでも積極的に利用するのが望ましい。そういった追加機能をサポートするためには、プロセッサやインプリメントに依存した特殊なシステムコールが必要となることがあるので、μITRONでは、そのための機能コードの範囲を決めている。

μITRONの範囲

これまで述べてきたように、μITRONは、非常にインプリメント依存部分の多いOSである。以下の章におけるμITRONのシステムコール説明では、システムコールの必要性に関するレベルが示されているが、これは絶対的なものではない。したがって、インプリメンタ側としては、どういう条件を満たせばμITRONと呼べるのか、といった問題を生じる。すなわち、μITRONと呼べるOSの範囲について明確にする必要がある。

一つの方法として考えられるのは、μITRONでサポートを必須とするシステムコールを決めることである。しかし、μITRONはいろいろな応用分野を持つOSであり、μITRONで提供すべき機能も、アプリケーション毎にかなり異なっている。実際、どのようなアプリケーションでも絶対に使用するシステムコールというのは、一つも存在しない。

例えば、割込み処理の機能はリアルタイムOSとして必須の機能であるが、割込みハンドラの定義のために、必ずしも割込みハンドラ定義のシステムコール(def_int)を使う必要はない。OSのジェネレーション時や初期化時に、静的な方法で割込みハンドラを定義することも可能である。したがって、システムコールとしては必須のものが無い。タスク管理機能についても同様である。休止(DORMANT)状態が無いケースを考えると、タスクの起動(sta_tsk)や終了(ext_tsk)のシステムコールも不可欠ではない。

また、同期・通信の機能はリアルタイムOSとして不可欠の機能であるが、μITRONではタスク付属同期機能(wup_tsk)、イベントフラグ、セマフォ、メイルボックスなどいくつかの同期・通信機能が用意されているため、最低限どれか一つの機能があれば十分である。プロセッサのアーキテクチャやアプリケーションが決まっていれば、どの同期・通信機能が最適かということが決まるので、最適なものを必須の機能としておけば良いが、μITRONの用途を考えると無理である。例えば、セマフォの機能をμITRONの必須機能としても、セマフォよりメイルボックスの方が適したアプリケーションは常に存在し、メイルボックスを使えばセマフォは不要になるかもしれない。したがって、セマフォをμITRONの必須機能とするのは、あ

まり意味がない。

こういった点を考えると、μITRONとして必須の機能やシステムコールを決めても、結局中途半端なものになってしまう。そこで、μITRONは、「機能とシステムコール名称との対応付けのガイドライン」といった位置付けのものに撤することにし、μITRON必須の機能は次の2つだけとした。

＜1＞RUN, READY, WAIT の3つのタスク状態を持つ。（マルチタスクOSである）
インプリメントによっては、このほかに DORMANT, SUSPEND の状態を持っても良い。

＜2＞割込みハンドラの定義が可能であり、割込みハンドラからシステムコールを発行することによって、タスクを起床する手段が存在する。（リアルタイムOSである）
割込みハンドラの定義は、静的に（システムジェネレーション時や初期化時に）、または def_int システムコールにより行う。

簡単に言えば、「システムコール名称をITRONに合わせたリアルタイムOS」はすべてμITRONということになる。極端な話、タスクを待ち状態にするシステムコール(slp_tsk)と、割込みハンドラから待ち状態のタスクを起床するシステムコール（wup_taskまたはret_wup）の2つのシステムコールしか持たないOSあっても、μITRONに含まれるわけである。

この場合、μITRONを利用する方から見ると、μITRON準拠のOSだからといって、どの機能が利用できるかが保障されているわけではない。しかし、前にも述べたように、μITRONであれば、システムコール名と機能との対応付けが保障されている。すなわち、ある機能が欲しい時にどういうシステムコールを使ったら良いかということや、プログラム中のシステムコールを見て何を行っているかということを容易に知ることができる。こういった「教育の互換性」は、実際にプログラミングを行う上で非常に重要なことである。特に、リアルタイムOSの分野では、UNIXやMS-DOSのように実質標準として広く使われているOSが無いため、システムコール名を揃えたというだけでも大きな効果がある。

TRON体系全体から見ると、教育の互換性のみを達成するのがμITRONであり、実際に動作させた時のプログラム互換性まで考慮しているのがITRONということになる。

μITRONの追加機能

μITRONでは、ITRONよりもさらに適応性を高めるため、考えられる機能はできるだけ広く用意するという方針をとっている。そのため、μITRONの仕様では、ITRON1の仕様に比べて次のような機能が追加された。なお、これらの追加機能の一部はITRON2と共通になっている。また、これらの機能を実際に導入するかどうかは、インプリメント依存である。

・周期起動ハンドラ、指定時刻起動ハンドラの機能
「システムコールインタフェース」の項でも述べたが、μITRONでは、ITRON1の周期起床(cyc_wup)の代替機能として、周期起動ハンドラおよび指定時刻起動ハンドラ（アラームハンドラ）の機能を設けた。前者は周期的に起動されるタスク独立のハンドラであり、後者は指定時刻に起動されるタスク独立のハンドラである。

・オブジェクトの状態を見る機能
タスク、イベントフラグ、セマフォ、メイルボックス、メモリプールなどのオブジェクトについて、状態参照システムコール(XXX_sts) を設けた。オブジェクトの状態参照は、ITRON1では、XXX_adr（アクセスアドレスを得るシステムコール）を利用して管理ブロックの内部を直接見るという方針であった。しかし、互換性を高めるためには、XXX_adr を使用するよりも、オブジェクトの状態を直接参照できるシステムコールを用意する方が望ましい。そこで、ITRON1のXXX_adr のシステムコールは廃止し、代わりに XXX_sts のシステムコールを導入した。

・タスク付属メイルボックスの機能
ITRONでは、一つ一つのタスクの仕様を軽くするため、タスク従属の機能（タスク付属の機能）はできるだけ入れない方針である。そのため、ITRONのメイルボックスの機能は、タスクとは独立したものになっている。すなわち、メッセージの送信先はタスクではなくメイルボックスであり、その時にそのメイルボックスに対してメッセージを待っていたタ

スクが、送信されたメッセージを受け取るという仕様である。
しかし、メッセージを受信するタスクがあらかじめ分かっている場合には、メッセージ通信を高速化するために、メイルボックスをタスク付属のものにした方が良い場合がある。μITRONのタスク付属メイルボックスは、このために導入された機能である。ここに送られたメッセージを受信できるのは、そのメイルボックスの属しているタスクに限られるため、メッセージを待つタスクの待ち行列が無くなり、メッセージ通信を高速化できる。
μITRONにおいて、タスク従属の機能は、t〜 という名称で区別する方針とする。したがって、タスク付属メイルボックスのオブジェクトの名称は、

tmb　　Task Mailbox

となる。

・タスクリスタート機能
タスクリスタート機能は、割込み処理からの戻り時に、指定タスクを指定アドレスから実行再開する機能である。この機能は、実行再開を指定されたタスクから見ると、一種の例外処理（ソフトウエアによる強制例外）に相当するが、簡単な仕様にして高速化することを狙うため、μITRONでは例外処理と別扱いにしている。
この機能は、例えば、フォールトトレラントの目的で、実行プログラムをチェックポイントまで戻したいような場合に利用する。

・その他
μITRONの仕様を今後の半導体技術の進歩に対応させていくためには、機能の絞り込みを行うだけではなく、一旦絞った機能を再度拡大できるように、仕様に柔軟性を持たせることが必要である。特に、μITRONの場合には、その上位のOSとしてITRON1やITRON2があり、μITRONを機能拡大する際は、できるだけこれらのOSと同じ仕様にすることが目標になる。そういった意味で、μITRONの仕様書は、一つのOSの仕様を表わすものではなく、μITRONの仕様がITRON1やITRON2の仕様からどの程度異なっても良いかということを示すガイドラインだと考えられる。
本書の以下の部分では、μITRONの標準システムコールの説明が行われているが、これは、μITRONで提供することができる機能の上限を定め

たものではない。必要があれば、この仕様書に無いシステムコールを追加して設けることも可能である。ただし、追加した機能がITRON2でサポートする機能に含まれるものであれば、ITRON2のシステムコール名やシステムコールインタフェースと矛盾の無い仕様にしなければならない。

第二部 第二章

μITRONシステムコール

この章では、μITRONで提供している各システムコールに関して説明を行う。

タスク管理機能

タスクを起動する

[2] sta_tsk

sta_tsk: Start Task

【パラメータ】

tskid	TaskIdentifier	タスクID

【リターンパラメータ】

なし

【解説】

tskidで示されたタスクを起動する。つまり、DORMANT状態からREADY 状態へと移す。

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なわない。したがって、対象タスクが DORMANT 状態にない場合に発せられた起動要求に対しては、発行タスクにエラー E_NODMT が戻る。

μITRONの場合、タスクの生成はシステム起動時に静的に行う。タスク生成時に必要なタスクスタートアドレス (stadr)、初期優先度 (itskpri) などのパラメータも、システム起動時に静的に指定する。

μITRONのタスク状態は、ITRONのサブセットとなる。ITRONのタスク状態のうち、NON-EXISTENT を除く

●RUN ●READY ●WAIT ●DORMANT ●SUSPEND ●WAIT-SUSPEND

の状態がμITRONに含まれる。ただし、インプリメントによっては、このうちDORMANT, SUSPEND(WAIT-SUSPEND) 状態を持たない場合がある。

【エラーコード(ercd)】

E_OK	正常終了
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
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_NODMT	タスクがDORMANTでない（tskidのタスクがDORMANTでない）

自タスクを正常終了する

[2] ext_tsk

ext_tsk:　　Exit Task

【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

【解説】

自タスクを正常終了させ、DORMANT 状態へと移行させる。

なお、ext_tsk では、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。

ret_XXX および ext_tsk システムコールでは、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そのチェックを行なっていなければ、プログラムが暴走する。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のが望ましい。

【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、インプリメント依存となる。

E_CTX　　コンテキストエラー（タスク独立部から実行）

他タスクを強制的に異常終了させる

[3] ter_tsk

ter_tsk: Terminate Task

【パラメータ】

tskid	TaskIdentifier	タスクID

【リターンパラメータ】

なし

【解説】

tskid で示されたタスクを強制的に異常終了させる。

ter_tsk によりタスクを終了する場合でも、ext_tsk の場合と同様に、対象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わけではない。

対象タスクが待ち状態に入り、何らかの待ち行列につながれていた場合には、ter_tskの実行によってその待ち行列から削除される。すなわち、対象タスクは待ち解除となってから終了する。

本システムコールでは、自タスクの指定はできない。

【エラーコード(ercd)】

E_OK	正常終了
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
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_DMT	タスクがDORMANTである（tskidのタスクがDORMANT）
E_SELF	自タスク、自プロセスの指定（tskidが自タスク、タスク部の発行でtskid=0）

タスク優先度を変更する

[3] chg_pri

タスク優先度を変更する（タスク独立部専用）

[#3] ichg_pri

chg_pri:　Change Task Priority
ichg_pri:　Change Task Priority (for Interrupt Handler)

【パラメータ】

tskid	TaskIdentifier	タスクID
tskpri	TaskPriority	優先度

【リターンパラメータ】

なし

【解説】

chg_pri, ichg_pri では、tskidで示されたタスクの現在の優先度を、tskpriで示される値に変更する。tskid = TSK_SELF によって自タスクの指定になる。

タスクの優先度は、数の小さい方が高い優先度となる。対象タスクが何らかの待ち行列につながれていた場合には、このシステムコールにより待ち行列の順番が変わることがある。

インプリメントによっては、tskpri = TPRI_INI (0) の指定により、システム起動時に指定された初期優先度（タスクの固有優先度）に戻ることのできる場合がある。この機能は、不可分の処理を行なうために一時的にタスクの優先度を高くし、その後タスク優先度を元に戻す場合などに使用する。

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 DORMANT 状態になると、終了前のタスク優先度は捨てられ、次にタスクが起動された時のタスク優先度は、システム起動時に指定された初期優先度 itskpri になる。

μITRONのタスク優先度は、1バイト程度で表現する（例えば 0〜255）のが原則であるが、インプリメントやアプリケーションによっては、優先度を数個（例えば1〜4）に絞ることによって高速化するという選択ができてもよい。

μITRONでは、OS内部でコンテキストを判断するオーバーヘッドを避けて性能を上げるため、タスク独立部（割込みハンドラ）から発行するシステムコールを、一般のタスクから発行するシステムコール chg_pri から分離して別システムコール ichg_pri とすることができる。OSの負担を減らし、タスク独立部から発行するシステムコールの処理を高速化するためには、タスク独立部から発行するシステムコール ichg_pri とタスク部から発行するシステムコール chg_pri は分離されている方が良い。一方、ユーザから見ると、タスク部からもタスク独立部からも共通のシステムコール chg_pri を発行できる方が便利である。どちらの仕様を用いるかはインプリメント側の選択項目となっている。ichg_pri をサポートする方がOSの機能が高いというわけではない。

【エラーコード(ercd)】

E_OK	正常終了
E_CTX	コンテキストエラー（タスク部からichg_priを実行）
E_TPRI	不正タスク優先度（tskpriが不正）
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
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_DMT	タスクがDORMANTである（tskidのタスクがDORMANT）

タスクのレディキューを回転する　[3] rot_rdq

タスクのレディキューを回転する（タスク独立部専用）　[#3] irot_rdq

rot_rdq:　Rotate Ready Queue
irot_rdq:　Rotate Ready Queue (for Interrupt Handler)

【パラメータ】

tskpri　TaskPriority　優先度

【リターンパラメータ】

なし

【解説】

tskpri で示される優先度のレディキューを回転する。すなわち、その優先度のレディキューの先頭につながれているタスクをレディキューの最後尾につなぎかえ、同一優先度のタスクの実行を切り換える。このシステムコールを一定時間間隔で発行することにより、ラウンドロビン・スケジューリングを行なうことが可能となる。

tskpri = TPRI_RUN (0) により、その時実行状態にあるタスクを含むレディキュー（最高優先度のレディキュー）を回転させるものとする。一般のタスクから発行される rot_rdq では、これは自タスクの持つ優先度のレディキューを回転するのと同じ意味になるが、このような仕様にしておけば、周期起動ハンドラなどのタスク独立部から rot_rdq (tskpri=TPRI_RUN) を発行することも可能である。

rot_rdq の tskpri として TPRI_RUN または自タスクの優先度を指定した場合には、自タスクがそのレディキューの後ろにまわることになる。つまり、自ら実行権を放棄するために、rot_rdq を発行することができる。（ここで言う「レディキュー」は、実行状態のタスクも含んだものと考えている。）

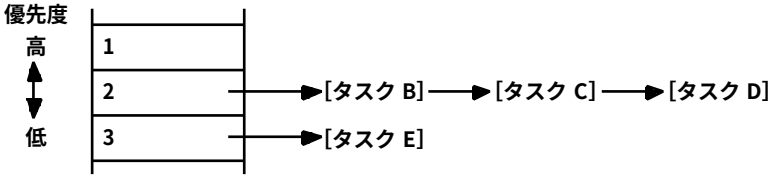
指定した優先度のレディキューにタスクがない場合は何もしないが、エラーとはならない。

rot_rdq の実行例を［図2.6(a)］,［図2.6(b)］に示す。［図2.6(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、新しいレディキューの状態は［図2.6(b)］のようになり、次に実行されるのはタスクCとなる。

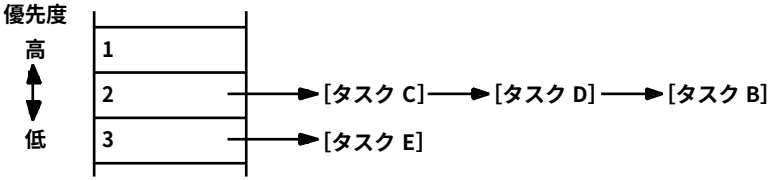
μITRONでは、OS内部でコンテキストを判断するオーバーヘッドを避けて性能を上げるため、タスク独立部（割込みハンドラ）から発行するシステムコールを、一般のタスクから発行するシステムコール rot_rdq から分離して別システムコール irot_rdq とすることができる。OSの負担を減らし、タスク独立部から発行するシステムコールの処理を高速化するためには、タスク独立部から発行するシステムコール irot_rdq とタスク部から発行するシステムコール rot_rdq は分離されている方が良い。一方、ユーザから見ると、タスク部からもタスク独立部からも共通のシステムコール rot_rdq を発行できる方が便利である。どちらの仕様を用いるかはインプリメント側の選択項目となっている。irot_rdq をサポートする方がOSの機能が高いというわけではない。

【エラーコード (ercd)】

E_OK	正常終了
E_CTX	コンテキストエラー（タスク部からirot_rdqを実行）
E_TPRI	不正タスク優先度（tskpriが不正）



［図2.6(a)］rot_rdq実行前のレディキューの状態



［図2.6(b)］rot_rdq(tskpri=2)実行後のレディキューの状態

タスクの待ち状態を強制解除する　[5] rel_wai

タスクの待ち状態を強制解除する（タスク独立部専用）　[#5] irel_wai

rel_wai:　Release Wait

【パラメータ】

tskid　TaskIdentifier　タスクID

【リターンパラメータ】

なし

【解説】

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

本システムコール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。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。

本システムコールでは、待ち状態解除要求のキューイングは行わない。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_NOWAI 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_NOWAI のエラーとなる。

本システムコールでは、SUSPEND状態の解除は行わない。二重待ち状態(WAIT-SUSPEND) のタスクを対象としてこのシステムコールを発行すると、対象タスクはSUSPEND状態となる。SUSPEND状態も解除する必要がある場合には、別に frsm_tsk を発行する必要がある。

rel_wai (irel_wai) と wup_tsk (iwup_tsk) との違いをまとめると、次のようになる。

— wup_tsk は slp_tsk, wai_tsk による待ち状態のみを解除するが、rel_wai ではそれ以外の要因（wai_flg, wai_sem, rcv_msg, 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、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、rel_wai による待ち状態の解除はエラー(E_RLWAI) である。
— wup_tsk の場合は、対象タスクがまだ slp_tsk, wai_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、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エラーとなる。

μITRONでは、OS内部でコンテキストを判断するオーバーヘッドを避けて性能を上げるため、タスク独立部（割込みハンドラ）から発行するシステムコールを、一般のタスクから発行するシステムコール rel_wai から分離して別システムコール irel_wai とすることができる。OSの負担を減らし、タスク独立部から発行するシステムコールの処理を高速化するためには、タスク独立部から発行するシステムコール irel_wai とタスク部から発行するシステムコール rel_wai は分離されている方が良い。一方、ユーザから見ると、タスク部からもタスク独立部からも共通のシステムコール rel_wai を発行できる方が便利である。どちらの仕様を用いるかはインプリメント側の選択項目となっている。irel_wai をサポートする方がOSの機能が高いというわけではない。

【エラーコード(ercd)】

E_OK	正常終了
E_CTX	コンテキストエラー（タスク部からirel_waiを実行）
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
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_DMT	タスクがDORMANTである（tskidのタスクがDORMANT）
E_NOWAI	タスクが待ち状態でない（tskidが自タスクの場合を含む）

自タスクのIDを得る

[3] get_tid

get_tid: Get Task Identifier

【パラメータ】

なし

【リターンパラメータ】

tskid	TaskIdentifier	タスクID

【解説】

自タスクのID番号を得る。タスク独立部から発行された場合は、tskid として FALSE (0) が返る。

【エラーコード(ercd)】

E_OK	正常終了

タスクの状態を見る

[3] tsk_sts

tsk_sts : Get Task Status

【パラメータ】

tskid	TaskIdentifier	タスクID

【リターンパラメータ】

tskpri	TaskPriority	現在の優先度
tskstat	TaskState	タスク状態

【解説】

tsk_sts では、tskid で示された対象タスクの各種の状態を参照し、対象タスクの現在の優先度 tskpri、タスク状態 tskstat 等をリターンパラメータとして返す。

このシステムコールのインタフェース形式としては、パケットを利用して tskpri, tskstat 等の情報をまとめて読み出す場合と、個々の情報を別々に読み出す場合とがある。

このうち、tskstat は次のような値をとるのが標準である。

tskstat :

TTS_RUN	RUN状態
TTS_RDY	READY状態
TTS_WAI	WAIT状態
TTS_SUS	SUSPEND状態
TTS_WAS	WAIT-SUSPEND状態
TTS_DMT	DORMANT状態

なお、tskstat として TTS_RUN, TTS_WAIT 等の値が定義されているが、これは、同じ値を TCB に入れなければならないという意味ではない。TCB 内でのタスク状態の表現方法はインプリメント依存であり、tsk_sts 実行時に、内部表現のタスク状態を TTS_RUN, TTS_WAIT 等の標準値に変換すれば良いわけである。

インプリメントによっては、tskpri, tskstat 以外に次のような追加情報を参照できる場合がある。

```
tskatr          /* タスク属性 */
itskpri         /* 初期優先度 */
suscnt          /* SUSPEND要求カウント */
wupcnt          /* 起床要求カウント */
```

tskid = TSK_SELF によって自タスクの指定になるが、このシステムコールでは自タスクのIDはわからない。自タスクのIDを知りたい場合は、get_tid を利用する。

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

【エラーコード(ercd)】

E_OK	正常終了
E_RSID	予約ID番号（インプリメント依存）
E_ILADR	不正アドレス（リターンパラメータ用のパケットアドレスが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
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)

タスク付属同期機能

タスクを強制待ち状態へ移行する　[3] sus_tsk

タスクを強制待ち状態へ移行する（タスク独立部専用）　[#3] isus_tsk

sus_tsk :　Suspend Task
isus_tsk :　Suspend Task (for Interrupt Handler)

【パラメータ】

tskid	TaskIdentifier	タスクID

【リターンパラメータ】

なし

【解説】

tskid で示されたタスクの実行を中断させ、強制待ち状態に移す。強制待ち状態は、rsm_tsk, frsm_tsk システムコールの発行によって解除される。強制待ち状態は、他タスクの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。

sus_tsk の要求のキューイングの有無、およびその回数の制限はインプリメント依存である。すなわち、既に強制待ち状態のタスクに対して sus_tsk を発行した場合に、エラー E_QOVR となるか、多重の強制待ち状態になるかはインプリメント依存である。後者の場合、sus_tsk が発行された回数 (suscnt) だけ rsm_tsk を発行すると、必ずもとの状態に戻るため、sus_tsk～rsm_tsk の対をネストすることが可能である。

WAIT-SUSPEND 状態のタスクに対しても、WAIT状態のタスクと同じように資源の割り当てが行なわれる。資源が割当てられることにより、そのタスクは SUSPEND 状態に移行する。WAIT-SUSPEND 状態のタスクに対する特別措置（資源割り当て遅延など）は取らない。これは、以下のような理由による。

・システムコールの意味的な問題
　　SUSPEND状態は、他の処理とは直交した関係のものであり、

SUSPEND の時だけ勝手に別の仕様に変えるのは良くないという考えによる。

・必要な場合は使い方の問題で対処可能であること

SUSPEND 状態のタスクの優先度を一時的に下げたいのであれば、ユーザタスクのレベルで、sus_tsk,rsm_tsk に chg_pri を組み合せて発行すれば済む。

μITRONでは、OS内部でコンテキストを判断するオーバーヘッドを避けて性能を上げるため、タスク独立部（割込みハンドラ）から発行するシステムコールを、一般のタスクから発行するシステムコール sus_tsk から分離して別システムコール isus_tsk とすることができる。OSの負担を減らし、タスク独立部から発行するシステムコールの処理を高速化するためには、タスク独立部から発行するシステムコール isus_tsk とタスク部から発行するシステムコール sus_tsk は分離されている方が良い。一方、ユーザから見ると、タスク部からもタスク独立部からも共通のシステムコール sus_tsk を発行できる方が便利である。どちらの仕様を用いるかはインプリメント側の選択項目となっている。isus_tsk をサポートする方がOSの機能が高いというわけではない。

【エラーコード(ercd)】

E_OK	正常終了
E_CTX	コンテキストエラー（タスク部からisus_tskを実行）
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
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_DMT	タスクがDORMANTである（tskidのタスクがDORMANT）
E_SELF	自タスク、自プロセスの指定（tskidが自タスク、タスク部の発行でtskid=0）
E_QOVR	キューイングのオーバーフロー（キューイング不可能、suscntのオーバーフロー）

強制待ち状態のタスクを再開する　[3] rsm_tsk

強制待ち状態のタスクを再開する（タスク独立部専用）　[#3] irsm_tsk

強制待ち状態のタスクを強制再開する　[4] frsm_tsk

rsm_tsk :　Resume Task
irsm_tsk :　Resume Task (for Interrupt Handler)
frsm_tsk :　Force Resume Task

【パラメータ】

tskid	TaskIdentifier	タスクID

【リターンパラメータ】

なし

【解説】

rsm_tsk, irsm_tsk, frsm_tsk システムコールでは、tskid で示された対象タスクが sus_tsk によって強制待ち状態に入っている場合に、その強制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。

rsm_tsk (irsm_tsk) システムコールでは、1回分の sus_tsk 要求 (suscnt) を解除する。したがって、対象タスクに対して2回以上の sus_tskが発行されていた場合 (suscnt≧2) には、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のままである。一方、frsm_tsk システムコールの場合は、対象タスクに対して2回以上の 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して suscnt = 0 となる。すなわち、強制待ち状態は必ず解除され、対象タスクは待ち状態（二重待ち状態）でない限り実行を再開できる。

μITRONでは、OS内部でコンテキストを判断するオーバーヘッドを避け

て性能を上げるため、タスク独立部（割込みハンドラ）から発行するシステムコールを、一般のタスクから発行するシステムコール rsm_tsk から分離して別システムコール irsm_tsk とすることができる。OSの負担を減らし、タスク独立部から発行するシステムコールの処理を高速化するためには、タスク独立部から発行するシステムコール irsm_tsk とタスク部から発行するシステムコール rsm_tsk は分離されている方が良い。一方、ユーザから見ると、タスク部からもタスク独立部からも共通のシステムコール rsm_tsk を発行できる方が便利である。どちらの仕様を用いるかはインプリメント側の選択項目となっている。irsm_tsk をサポートする方がOSの機能が高いというわけではない。

【エラーコード(ercd)】

E_OK	正常終了
E_CTX	コンテキストエラー（タスク部からirsm_tskを実行）
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
E_SELF	自タスク、自プロセスの指定（インプリメント依存）
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_DMT	タスクがDORMANTである（tskidのタスクがDORMANT）
E_NOSUS	タスクがSUSPENDでない（インプリメント依存）

タスクを待ち状態へ移行する
[1] slp_tsk

タスクを一定時間待ち状態に移行する
[3] wai_tsk

slp_tsk:　Sleep Task
wai_tsk:　Wait for Wakeup Task

【パラメータ】

＜1＞tmout　Timeout　タイムアウト指定
［＜1＞wai_tsk のみ］

【リターンパラメータ】

なし

【解説】

slp_tsk システムコールでは、自タスクを実行状態から待ち状態に移す。この待ち状態は、本タスクを対象として発行された wup_tsk (iwup_tsk) システムコールまたは rel_wai (irel_wai) システムコールにより解除される。

wai_tsk システムコールでは、自タスクを一定時間だけ実行状態から待ち状態に移す。この待ち状態は、本タスクを対象とした wup_tsk (iwup_tsk) システムコールの発行、本タスクを対象とした rel_wai (irel_wai) システムコールの発行、あるいは tmout で指定した時間の経過により解除される。wup_tsk が発行された場合は正常終了、時間経過の場合はタイムアウトエラー E_TMOUT としてリターンする。このシステムコールは、slp_tsk システムコールにタイムアウト機能を付けたものであり、タスクの単純な遅延に用いることができる。

wai_tsk の tmout のビット数やデータタイプ（符号付き、符号無し）はインプリメント依存である。また、インプリメントによっては、tmout = TMO_FEVR によりタイムアウト指定が行なわれないことを示す。この場合は、wup_tsk が発行されるまで永久に待ち状態になるため、slp_tsk と同じ動作をする。

自タスクの指定のみであるため、slp_tsk, wai_tsk のネスティングは有り

得ないが、slp_tsk, wai_tsk により待ち状態となっている時に、他のタスクから sus_tsk が実行されることはある。この場合はwup_tsk により待ち状態が解除されてもまだ強制待ち状態であり、rsm_tsk (irsm_tsk, frsm_tsk) の発行までタスクの実行は再開されない。

【エラーコード(ercd)】

E_OK	正常終了
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_ILTIME	不正時間指定（インプリメント依存）
E_CTX	コンテキストエラー（タスク独立部から実行）
E_TMOUT	タイムアウト
E_RLWAI	待ち状態強制解除（待ちの間にrel_waiを受け付け）

待ち状態のタスクを起床する　[1] wup_tsk

待ち状態のタスクを起床する（タスク独立部専用）　[#1] iwup_tsk

wup_tsk :　Wakeup Task
iwup_tsk :　Wakeup Task (for Interrupt Handler)

【パラメータ】

tskid	TaskIdentifier	タスクID

【リターンパラメータ】

なし

【解説】

wup_tsk, iwup_tsk システムコールでは、slp_tsk または wai_tsk システムコールの実行により待ち状態になっていたタスクを、待ち状態から実行可能状態に移す。対象タスクは、tskid で示され、自タスクを指定することはできない。

対象タスクが slp_tsk または wai_tsk を実行しておらず、待ち状態でない場合には、このwai_tsk要求はキューイングされる。その場合、この wup_tsk 要求は、後に対象タスクが slp_tsk または wai_tsk システムコールを実行した時に有効となる。具体的には、wup_tsk を実行すると対象タスクの起床要求カウントがプラス1され、対象タスクが slp_tsk または wai_tsk を実行すると、対象タスクの起床要求カウントがマイナス1される。起床要求カウントが負になろうとした時に、はじめて待ち状態になる。

起床要求 (wupcnt) のキューイング数の最大値はインプリメント依存であるが、最小限1回のキューイングはできるものとする。2回以上のキューイングができるかどうかは、インプリメントによって異なる。

μITRONでは、OS内部でコンテキストを判断するオーバーヘッドを避けて性能を上げるため、タスク独立部（割込みハンドラ）から発行するシステムコールを、一般のタスクから発行するシステムコール wup_tsk から分

離して別システムコール iwup_tsk とすることができる。OSの負担を減らし、タスク独立部から発行するシステムコールの処理を高速化するためには、タスク独立部から発行するシステムコール iwup_tsk とタスク部から発行するシステムコール wup_tsk は分離されている方が良い。一方、ユーザから見ると、タスク部からもタスク独立部からも共通のシステムコール wup_tsk を発行できる方が便利である。どちらの仕様を用いるかはインプリメント側の選択項目となっている。iwup_tsk をサポートする方がOSの機能が高いというわけではない。

【エラーコード(ercd)】

E_OK	正常終了
E_CTX	コンテキストエラー（タスク部からiwup_tskを実行）
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
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_DMT	タスクがDORMANTである（tskidのタスクがDORMANT）
E_SELF	自タスク、自プロセスの指定（tskidが自タスク、タスク部の発行でtskid=0）
E_QOVR	キューイングのオーバーフロー（wupcntのオーバーフロー）

タスクの起床要求を無効にする　[3] can_wup

can_wup:　Cancel Wakeup Task

【パラメータ】

tskid	TaskIdentifier	タスクID

【リターンパラメータ】

wupcnt	Wakeup Count	キューイングされていた起床要求回数

【解説】

can_wup システムコールでは、tskid で示された対象タスクにキューイングされていた起床要求回数(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解除する。tskid = TSK_SELF によって自タスクの指定になる。

このシステムコールは、周期的にタスクを起床する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（前の起床要求に対する slp_tsk が実行される前に、次の起床要求が発生していないか）を判定するために利用できる。can_wup のリターンパラメータである wupcnt が 0 でなければ、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がわかるので、それに対して何らかの処置をすることができる。

【エラーコード(ercd)】

E_OK	正常終了
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
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_DMT	タスクがDORMANTである（tskidのタスクがDORMANT）

同期・通信機能

1ワードイベントフラグをセットする　［3A］set_flg

1ワードイベントフラグをセットする（タスク独立部専用）　［#3A］iset_flg

1ワードイベントフラグをクリアする

set_flg:　Set EventFlag
iset_flg:　Set EventFlag (for Interrupt Handler)
clr_flg:　Clear EventFlag

【パラメータ】

flgid	EventFlagIdentifier	イベントフラグID
＜1＞setptn	SetBitPattern	セットするビットパターン
＜2＞clrptn	ClearBitPattern	クリアするビットパターン

［＜1＞set_flg のみ］
［＜2＞clr_flg のみ］

【リターンパラメータ】

なし

【解説】

set_flg では、flgid で示される1ワード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1ワード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、clr_flg では、flgid で示される1ワードイベントフラグのうち、対応する clrptn が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。

set_flg において、イベントフラグ値の変更の結果、そのイベントフラグを待っていたタスクの待ち解除条件を満たすようになれば、そのタスクが実行可能状態へと移行する。イベントフラグでは、ビット対応のイベントの OR や AND を条件として待つことができるが、そのほか、イベントフラ

グの全ビットを使用すれば、1ワードの簡単なメッセージ転送（キューイング無し）を行なうこともできる。

clr_flg では、その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。

set_flg で setptn の全ビットを0とした場合や、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、その場合でもエラーとはならない。

同一イベントフラグに対する複数タスクの待ちも可能である。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 set_flg で複数のタスクが待ち解除となることがある。

μITRONの場合、イベントフラグの生成はシステム起動時に静的に行う。また、μITRONの仕様では、1ワードイベントフラグの仕様と1ビットイベントフラグの仕様の両方が用意されており、インプリメンタの側で必要な方を選択してサポートすることができる。高機能でITRON1やITRON2の仕様に近いものをインプリメントするという意味では1ワードイベントフラグが良く、高速で必要メモリ量の少ないものをインプリメントするという意味では1ビットイベントフラグが良い。

μITRONでは、OS内部でコンテキストを判断するオーバーヘッドを避けて性能を上げるため、タスク独立部（割込みハンドラ）から発行するシステムコールを、一般のタスクから発行するシステムコール set_flg から分離して別システムコール iset_flg とすることができる。OSの負担を減らし、タスク独立部から発行するシステムコールの処理を高速化するためには、タスク独立部から発行するシステムコール iset_flg とタスク部から発行するシステムコール set_flg は分離されている方が良い。一方、ユーザから見ると、タスク部からもタスク独立部からも共通のシステムコール set_flg を発行できる方が便利である。どちらの仕様を用いるかはインプリメント側の選択項目となっている。iset_flg をサポートする方がOSの機能が高いというわけではない。

【エラーコード (ercd)】

E_OK	正常終了
E_CTX	コンテキストエラー（タスク部からiset_flgを実行）
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
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）

1ワードイベントフラグを待つ [3A] wai_flg

1ワードイベントフラグを得る [3A] pol_flg

wai_flg:　Wait EventFlag
pol_flg:　Poll EventFlag

【パラメータ】

flgid	EventFlagIdentifier	イベントフラグID
waiptn	WaitBitPattern	待ちビットパターン
wfmode	WaitEventFlagMode	待ちモード

【リターンパラメータ】

flgptn	EventFlagBitPattern	待ち解除時のビットパターン

【解説】

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

wfmode では、次のような指定を行う。

wfmode : = (TWF_ANDW ‖ TWF_ORW) ｜［TWF_CLR］

TWF_ANDW	AND待ち
TWF_ORW	OR待ち
TWF_CLR	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。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 で指定したビットがすべてセットされるのを待つ。

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。一方、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）を0にクリアする。

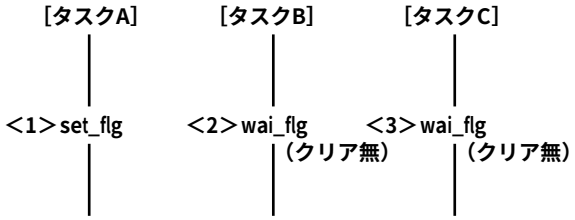
μITRONの wai_flg ではタイムアウトの指定はできず、条件が満足されるまで永久に待つことになる。一方、待ち条件が満たされない場合でも即時にリターンするシステムコールとして、pol_flg が用意されている。pol_flg システムコールのパラメータ等の意味は wai_flg と同じである。pol_flg では、待ち条件が満たされていれば正常終了となるが、待ち条件が満たされていない場合には、待ち状態に入らずにエラー E_PLFAIL を返す。TWF_CLR の指定によってフラグのクリアが行われるのは、pol_flg が正常終了した場合のみである。

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 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。

同一イベントフラグに対する複数タスクの待ちも可能である。この場合、一回の set_flg で複数のタスクが待ち解除となることがある。イベントフラグでもタスクの待ち行列が定義され、次のような動作をする。

― 待ち行列の順番は FIFO 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ため、待ち解除とはならない。

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＞set_flg を実行するまで、タスクB、タスクCを＜2＞＜3＞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2.7］。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wai_flgがエラーになる。



[図2.7] 1ワードイベントフラグに対する複数タスク待ちの機能

waiptn を0とした場合は、set_flg により待ち状態から抜けることができなくなるため、パラメータエラー E_PAR とする。したがって、set_flg で全ビットをセットした場合、待ち行列の先頭にあるタスクであれば、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解除となることが保証される。

【エラーコード(ercd)】

E_OK	正常終了
E_RSMD	予約モード、予約オプション（wfmodeが不正）
E_RSID	予約ID番号（インプリメント依存）
E_PAR	一般的なパラメータエラー（waiptn = 0）
E_IDOVR	ID範囲外（インプリメント依存）
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_CTX	コンテキストエラー（タスク独立部から実行）
E_RLWAI	待ち状態強制解除（待ちの間にrel_waiを受け付け）
E_PLFAIL	ポーリング失敗

1ワードイベントフラグ状態を参照する　［4A］flg_sts

flg_sts:　Get EventFlag Status

【パラメータ】

flgid	EventFlagIdentifier	イベントフラグID

【リターンパラメータ】

wtskid	WaitTaskID	待ちタスクID
flgptn	EventFlagBitPattern	イベントフラグのビットパターン

【解説】

flg_sts では、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、対象イベントフラグの現在のフラグ値 flgptn、待ちタスクID番号 wtskid 等をリターンパラメータとして返す。

このシステムコールのインタフェース形式としては、パケットを利用して flgptn, wtskid 等の情報をまとめて読み出す場合と、個々の情報を別々に読み出す場合とがある。

wtskid には、待ち行列の先頭のタスクのIDが返る。すなわち、現在このイベントフラグを待っているタスクのうち、最も早く待ち状態になったタスクのIDが返る。待ちタスクが無い場合は FALSE(0) が返る。

インプリメントによっては、flgptn, wtskid 以外の追加情報も参照できる場合がある。

flg_sts で、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。

【エラーコード(ercd)】

E_OK	正常終了
E_RSID	予約ID番号（インプリメント依存）
E_ILADR	不正アドレス（リターンパラメータ用のパケットアドレスが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
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）

1ビットイベントフラグをセットする [3B] set_flg

1ビットイベントフラグをセットする（タスク独立部専用） [#3B] iset_flg

1ビットイベントフラグをクリアする [3B] clr_flg

set_flg:　Set EventFlag
iset_flg:　Set EventFlag (for Interrupt Handler)
clr_flg:　Clear EventFlag

【パラメータ】

flgid	EventFlagIdentifier	イベントフラグID

【リターンパラメータ】

なし

【解説】

set_flg では、flgid で示される1ビットイベントフラグが1にセットされる。また、clr_flg では、flgid で示される1ビットイベントフラグが0にクリアされる。

set_flg の場合、wai_flg, cwai_flg によってそのイベントフラグを待っているタスクがあれば、そのタスクの待ちを解除して実行可能状態へと移す。一方、clr_flg の場合は、その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。

1ビットイベントフラグの場合には、ビット対応に事象を表現し、それらのOR待ちやAND待ちを行うことはできない。しかし、単純な事象のOR待ちやAND待ちを行うのであれば、set_flgを複数回組み合わせたり、wai_flg や cwai_flg を複数回組み合わせたりすることによって実現できる場合がある。

同一イベントフラグに対する複数タスクの待ちも可能である。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 set_flg で複数のタスクが待ち解除となることがある。

μITRONの場合、イベントフラグの生成はシステム起動時に静的に行う。また、μITRONの仕様では、1ワードイベントフラグの仕様と1ビットイベントフラグの仕様の両方が用意されており、インプリメンタの側で必要な方を選択してサポートすることができる。高機能でITRON1やITRON2の仕様に近いものをインプリメントするという意味では1ワードイベントフラグが良く、高速で必要メモリ量の少ないものをインプリメントするという意味では1ビットイベントフラグが良い。

μITRONでは、OS内部でコンテキストを判断するオーバーヘッドを避けて性能を上げるため、タスク独立部（割込みハンドラ）から発行するシステムコールを、一般のタスクから発行するシステムコール set_flg から分離して別システムコール iset_flg とすることができる。OSの負担を減らし、タスク独立部から発行するシステムコールの処理を高速化するためには、タスク独立部から発行するシステムコール iset_flg とタスク部から発行するシステムコール set_flg は分離されている方が良い。一方、ユーザから見ると、タスク部からもタスク独立部からも共通のシステムコール set_flg を発行できる方が便利である。どちらの仕様を用いるかはインプリメント側の選択項目となっている。iset_flg をサポートする方がOSの機能が高いというわけではない。

【エラーコード (ercd)】

E_OK	正常終了
E_CTX	コンテキストエラー（タスク部からiset_flgを実行）
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
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）

1ビットイベントフラグを待つ（クリア無）　[3B] wai_flg

1ビットイベントフラグを待つ（クリア有）　[3B] cwai_flg

1ビットイベントフラグを得る（クリア無）　[3B] pol_flg

1ビットイベントフラグを得る（クリア有）　[3B] cpol__flg

wai_flg:　Wait EventFlag
cwai_flg:　Wait and Clear EventFlag
pol_flg:　Poll EventFlag
cpol_flg:　Poll and Clear EventFlag

【パラメータ】

flgid	EventFlagIdentifier	イベントフラグID

【リターンパラメータ】

なし

【解説】

wai_flg, cwai_flg では、flgidで示されるイベントフラグが1にセットされるのを待つ。

wai_flg の場合には、イベントフラグがセット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セットされたままである。一方、cwai_flg の場合には、イベントフラグがセット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が0にクリアされる。

μITRONの wai_flg, cwai_flg ではタイムアウトの指定はできず、イベントフラグがセットされるまで永久に待つことになる。一方、イベントフラグがセットされていなくても即時にリターンするシステムコールとして、

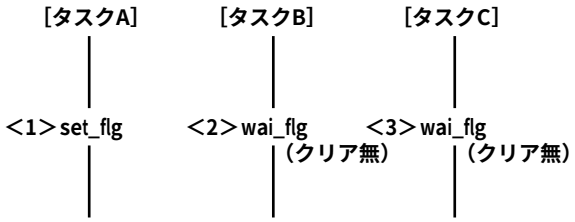
pol_flg, cpol_flg が用意されている。pol_flg, cpol_flg システムコールのパラメータ等の意味は wai_flg, cwai_flg と同じである。pol_flg, cpol_flg では、イベントフラグがセットされていれば正常終了となるが、イベントフラグがセットされていない場合には、待ち状態に入らずにエラー E_PLFAIL を返す。cpol_flg の場合にフラグのクリアが行われるのは、正常終了した場合のみである。なお、cpol_flg ではイベントフラグがセットされていた場合にそれをクリアするのに対して、pol_flg では単にイベントフラグの値の参照のみを行う。したがって、pol_flg の機能は次の flg_sts で代用可能であるが、パラメータの整理と対称性のために別システムコールとなっている。

同一イベントフラグに対する複数タスクの待ちも可能である。この場合、一回の set_flg で複数のタスクが待ち解除となることがある。イベントフラグでもタスクの待ち行列が定義され、次のような動作をする。

— 待ち行列の順番は FIFO である。1ビットイベントフラグの場合には、必ず行列先頭のタスクから待ち解除になる。
— 待ち行列中にクリア指定のタスク（cwai_flgで待っているタスク）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ため、待ち解除とはならない。

1ビットイベントフラグでは、set_flg が実行された場合に、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＞set_flg を実行するまで、タスクB、タスクCを＜2＞＜3＞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2.8］。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wai_flgがエラーになる。



[図2.8] 1ビットイベントフラグに対する複数タスク待ちの機能

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
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_CTX	コンテキストエラー（タスク独立部から実行）
E_RLWAI	待ち状態強制解除（待ちの間にrel_waiを受け付け）
E_PLFAIL	ポーリング失敗

1ビットイベントフラグ状態を参照する ［4B］flg_sts

flg_sts:　Get EventFlag Status

【パラメータ】

flgid	EventFlagIdentifier	イベントフラグID

【リターンパラメータ】

wtskid	WaitTaskID	待ちタスクID
flgptn	EventFlagBitPattern	イベントフラグのビットパターン

【解説】

flg_sts では、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、対象イベントフラグの現在のフラグ値 flgptn（1ビット）、待ちタスクID番号 wtskid 等をリターンパラメータとして返す。

このシステムコールのインタフェース形式としては、パケットを利用して flgptn, wtskid 等の情報をまとめて読み出す場合と、個々の情報を別々に読み出す場合とがある。

wtskid には、待ち行列の先頭のタスクのIDが返る。すなわち、現在このイベントフラグを待っているタスクのうち、最も早く待ち状態になったタスクのIDが返る。待ちタスクが無い場合は FALSE (0) が返る。

インプリメントによっては、flgptn, wtskid 以外の追加情報も参照できる場合がある。

flg_sts で、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。

【エラーコード(ercd)】

E_OK	正常終了
E_RSID	予約ID番号（インプリメント依存）
E_ILADR	不正アドレス（リターンパラメータ用のパケットアドレスが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
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）

セマフォに対する信号操作（V命令）　[1] sig_sem

セマフォに対する信号操作（タスク独立部専用）　[#1] isig_sem

sig_sem:　Signal Semaphore
isig_sem:　Signal Semaphore (for Interrupt Handler)

【パラメータ】

semid	SemaphoreIdentifier	セマフォID

【リターンパラメータ】

なし

【解説】

semidで示されたセマフォに対して待っているタスクがあれば、待ち行列の先頭のタスクを実行可能状態にする。セマフォに対して待っているタスクが無ければ、そのセマフォのカウント値 (semcnt) を1だけ増やす。

セマフォのカウント値の増加により、カウント値がシステム起動時のカウント値 (isemcnt) を越えた場合にも、エラーとはしない。これは、資源管理ではなく、同期の目的（wup_tsk～slp_tsk と同様）でセマフォを使用することを想定しているためである。また、セマフォを一旦生成した後で、資源数を動的に再設定したい場合にも、カウント値が初期値を越える場合がある。なお、セマフォのカウント値 (semcnt) がインプリメントの都合による上限値を越えた場合には、エラー E_QOVR を返す。

μITRONの場合、セマフォの生成はシステム起動時に静的に行う。セマフォ生成時に必要なセマフォ初期値 (isemcnt) などのパラメータも、システム起動時に静的に指定する。

μITRONのセマフォの場合、待ちタスクのキューイング方法はFIFO(TA_TFIFO)とするのが標準である。ただし、インプリメントによっては、タスク優先度順 (TA_TPRI) のキューイング方法を選択できるようになっていても良い。また、μITRONの wai_sem では資源要求数 (rcnt) が1に固

定されているので、ITRON2における TA_FIRST（行列先頭のタスクを優先扱い）と TA_CNT（要求数の少ないタスクを優先扱い）の区別は意味を持たない。

μITRONでは、OS内部でコンテキストを判断するオーバーヘッドを避けて性能を上げるため、タスク独立部（割込みハンドラ）から発行するシステムコールを、一般のタスクから発行するシステムコール sig_sem から分離して別システムコール isig_sem とすることができる。OSの負担を減らし、タスク独立部から発行するシステムコールの処理を高速化するためには、タスク独立部から発行するシステムコール isig_sem とタスク部から発行するシステムコール sig_sem は分離されている方が良い。一方、ユーザから見ると、タスク部からもタスク独立部からも共通のシステムコール sig_sem を発行できる方が便利である。どちらの仕様を用いるかはインプリメント側の選択項目となっている。isig_sem をサポートする方がOSの機能が高いというわけではない。

isig_sem のシステムコールをタスク部から発行した場合には、システムコールの意味的な間違いであるという意味で E_CTX のエラーを返すのが望ましい。また、isig_sem のシステムコール全体が実装されていない場合には、E_RSFN のエラーを返すのが望ましい。具体的には、インプリメントに依存して［表2.1］の＜1＞～＜4＞のいずれかの動作をすることになる。iset_flg, isnd_msg なども同様である。

	タスク部から発行	タスク独立部から発行
sig_sem	＜1＞E_RSFN	＜1＞E_RSFN
	＜2＞実行可能	＜2＞実行可能
		＜3＞E_XNOSPT
		＜4＞E_CTX
isig_sem	＜1＞E_RSFN	＜1＞E_RSFN
	＜2＞E_CTX	＜2＞実行可能

［表2.1］sig_sem,isig_semの返すエラー

【エラーコード (ercd)】

E_OK	正常終了
E_CTX	コンテキストエラー（タスク部からisig_semを実行）
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
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	キューイングのオーバーフロー（semcntのオーバーフロー）

セマフォに対する待ち操作（P命令） [1] wai_sem

セマフォ資源を得る [1] preq_sem

wai_sem: Wait on Semaphore
preq_sem: Poll and Request Semaphore

【パラメータ】

semid	SemaphoreIdentifier	セマフォID

【リターンパラメータ】

なし

【解説】

wai_sem では、semid で示されたセマフォのカウント値 (semcnt) が1以上の場合、セマフォのカウント値を1だけ減じて、本システムコールの発行タスクは実行を継続する。セマフォのカウント値が0の場合は、セマフォのカウント値は変更せず、本システムコールを発行したタスクはそのセマフォに対する待ち行列につながれる。待ち行列へのつながれ方はFIFOが標準である。

μITRONの wai_sem ではタイムアウトの指定はできず、条件が満足されるまで永久に待つことになる。一方、待ち条件が満たされない場合でも即時にリターンするシステムコールとして、preq_sem が用意されている。preq_sem システムコールでは、semid で示されたセマフォのカウント値が1以上の場合、セマフォのカウント値を1だけ減じて、本システムコールの発行タスクは正常終了する。セマフォのカウント値が0の場合は、待ち状態に入らずにエラー E_PLFAIL を返す。この場合、セマフォのカウント値は0のままである。

preq_sem の機能は、ITRON2では wai_sem (tmout=TMO_POL) の指定により実現される機能であるが、μITRONの場合は wai_sem でタイムアウトの指定ができないため、別システムコール preq_sem に分離している。なお、

sem_sts が単にセマフォ値を参照するシステムコールであるのに対して、preq_sem の場合は、資源を獲得できる時は獲得してしまう（セマフォのカウント値が正であれば1だけ減じる）という点が異なっている。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
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_CTX	コンテキストエラー（タスク独立部から実行）
E_RLWAI	待ち状態強制解除（待ちの間にrel_waiを受け付け）
E_PLFAIL	ポーリング失敗

セマフォ状態を参照する
[4] sem_sts

sem_sts: Get Semaphore Status

【パラメータ】

semid	SemaphoreIdentifier	セマフォID

【リターンパラメータ】

wtskid	WaitTaskID	待ちタスクID
semcnt	SemaphoreCount	現在のセマフォカウント値

【解説】

sem_sts では、semid で示された対象セマフォの各種の状態を参照し、対象セマフォの現在のカウント値semcnt、待ちタスクID番号 wtskid等をリターンパラメータとして返す。

このシステムコールのインタフェース形式としては、パケットを利用して semcnt, wtskid 等の情報をまとめて読み出す場合と、個々の情報を別々に読み出す場合とがある。

wtskid には、待ち行列の先頭のタスクのIDが返る。すなわち、待ち行列へのつながれ方がFIFOの場合には、現在このセマフォを待っているタスクのうち、最も早く待ち状態になったタスクのIDが返る。待ちタスクの無い時は FALSE (0) が返る。semcnt と wtskid の少なくとも一方は0となる。

インプリメントによっては、semcnt, wtskid 以外の追加情報も参照できる場合がある。

sem_sts で、対象セマフォが存在しない場合には、エラーE_NOEXS となる。

【エラーコード(ercd)】

E_OK	正常終了
E_RSID	予約ID番号（インプリメント依存）
E_ILADR	不正アドレス（リターンパラメータ用のパケットアドレスが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
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）

メイルボックスへ送信する　[2] snd_msg

メイルボックスへ送信する（タスク独立部専用）　[#2] isnd_msg

snd_msg:　Send Message to Mailbox
isnd_msg:　Send Message to Mailbox (for Interrupt Handler)

【パラメータ】

mbxid	MailboxIdentifier	メイルボックスID
pk_msg	MessagePacket	送信メッセージの先頭アドレス

【リターンパラメータ】

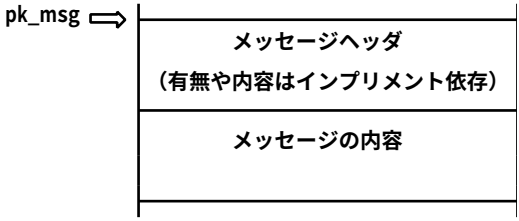
なし

【解説】

mbxidで示された対象メイルボックスに、pk_msg のアドレスに入っているメッセージを送信する。メッセージの内容はコピーされず、受信時にはアドレス（pk_msgの値）のみが渡される。

対象メイルボックスでメッセージを待つタスクがない時に snd_msg が発行されても、snd_msg 発行タスクは待ち状態とはならない。このシステムコールでは、メッセージをメイルボックスに入れ、メッセージキューと呼ぶ待ち行列につなぐだけで、タスクはその先の実行を続ける。つまり、非同期のメッセージ送信を行なう。また、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（キュー）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。

メッセージキューをリングバッファによって実現するか、線形リストによって実現するかはインプリメント依存である。メッセージキューをリングバッファによって実現した場合に、リングバッファが一杯になってメッセージのキューイングができなくなっても、snd_msg 発行タスクは待ち状態とはならない。この場合、snd_msg 発行タスクは E_QOVR のエラーを返



［図2.9］μITRONにおけるメッセージの形式

して即座にリターンする。

一方、メッセージキューを線形リストによって実現した場合、メッセージブロックの中に、OSで用いるキューのリンク用エリアが必要となる。したがって、OSの管理用として、ユーザの用意したメッセージブロックの一部を使用することがある。この部分を、メッセージヘッダと呼ぶ。この場合、pk_msg は、メッセージヘッダをも含めたメッセージブロックの先頭アドレスを表わす［図2.9］。ユーザが実際にメッセージを入れることができるのは、pk_msg の直後からではなく、メッセージヘッダの後の部分からとなる。メッセージヘッダの有無や大きさはインプリメント依存である。

μITRONの場合、メイルボックスの生成はシステム起動時に静的に行う。メイルボックス生成時に必要なパラメータ（メイルボックス属性など）がある場合にも、システム起動時に静的に指定する。

μITRONのメイルボックスの場合、待ちタスクとメッセージのキューイング方法はFIFO (TA_TFIFO, TA_MFIFO) とするのが標準である。ただし、インプリメントによっては、タスク優先度順 (TA_TPRI) やメッセージ優先度順(TA_MPRI)のキューイング方法を選択できるようになっていても良い。これらの選択を行う場合には、システム起動時のメイルボックス生成の際に指定するメイルボックス属性 (mbxatr) を利用する。メッセージ優先度順の属性とした場合には、メッセージヘッダの中で各メッセージの優先度 (msgpri) の指定を行う。

μITRONでは、OS内部でコンテキストを判断するオーバーヘッドを避けて性能を上げるため、タスク独立部（割込みハンドラ）から発行するシステ

ムコールを、一般のタスクから発行するシステムコール snd_msg から分離して別システムコール isnd_msg とすることができる。OSの負担を減らし、タスク独立部から発行するシステムコールの処理を高速化するためには、タスク独立部から発行するシステムコール isnd_msg とタスク部から発行するシステムコール snd_msg は分離されている方が良い。一方、ユーザから見ると、タスク部からもタスク独立部からも共通のシステムコール snd_msg を発行できる方が便利である。どちらの仕様を用いるかはインプリメント側の選択項目となっている。isnd_msg をサポートする方がOSの機能が高いというわけではない。

【エラーコード (ercd)】

E_OK	正常終了
E_CTX	コンテキストエラー（タスク部からisnd_msgを実行）
E_RSID	予約ID番号（インプリメント依存）
E_ILADR	不正アドレス（pk_msgが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
E_ILMSG	不正メッセージ形式（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（mbxidのメイルボックスが存在しない）
E_QOVR	キューイングのオーバーフロー（メッセージキューのオーバーフロー）

メイルボックスからの受信を待つ [2] rcv_msg

メイルボックスから受信する [2] prcv_msg

rcv_msg:　Receive Message from Mailbox
prcv_msg:　Poll and Receive Message from Mailbox

【パラメータ】

mbxid　MailboxIdentifier　メイルボックスID

【リターンパラメータ】

pk_msg　MessagePacket　受信メッセージの先頭アドレス

【解説】

rcv_msg では、mbxid で示されたメイルボックスからメッセージを受信し、受信したメッセージの先頭アドレスをリターンパラメータとして返す。まだそのメイルボックスにメッセージが到着していない場合には、本システムコール発行タスクはメッセージ到着を待つ待ち行列につながれる。待ち行列へのつながれ方はFIFOが標準である。

pk_msg は、メッセージヘッダ（OSで使用する予約領域など）をも含めたメッセージブロックの先頭アドレスである。ユーザが実際にメッセージを入れることができるのは、メッセージヘッダの後の部分であるが、メッセージヘッダの有無や大きさはインプリメント依存である。

μITRONの rcv_msg ではタイムアウトの指定はできず、メッセージが受信できるまで永久に待つことになる。一方、まだメッセージが到着していない場合にも即時にリターンするシステムコールとして、prcv_msg が用意されている。prcv_msg システムコールでは、対象メイルボックスに既にメッセージが有る場合には、そのメッセージを受信して正常終了する。メッセージが無い場合には、待ち状態に入らずにエラー E_PLFAIL を返す。

prcv_msg の機能は、ITRON2では rcv_msg (tmout=TMO_POL) の指定により実現される機能であるが、μITRONの場合は rcv_msg でタイムアウトの

指定ができないため、別システムコール prcv_msg に分離している。なお、mbx_sts を利用しても次に受信されるべきメッセージを知ることは可能であるが、prcv_msg の場合は、メッセージが受信できる時には受信してしまう（先頭のメッセージをメッセージキューから削除する）という点が異なっている。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（mbxidのメイルボックスが存在しない）
E_CTX	コンテキストエラー（タスク独立部から実行）
E_RLWAI	待ち状態強制解除（待ちの間にrel__waiを受け付け）
E_PLFAIL	ポーリング失敗

メイルボックス状態を参照する [4] mbx_sts

mbx_sts: Get Mailbox Status

【パラメータ】

mbxid	MailboxIdentifier	メイルボックスID

【リターンパラメータ】

wtskid	WaitTaskID	待ちタスクID
pk_msg	MessagePacket	次のメッセージの先頭アドレス

【解説】

mbx_sts では、mbxid で示された対象メイルボックスの各種の状態を参照し、次に受信されるメッセージの先頭アドレス pk_msg、待ちタスクID番号 wtskid 等をリターンパラメータとして返す。

このシステムコールのインタフェース形式としては、パケットを利用して pk_msg, wtskid 等の情報をまとめて読み出す場合と、個々の情報を別々に読み出す場合とがある。

wtskid には、待ち行列の先頭のタスクのIDが返る。すなわち、待ち行列へのつながれ方がFIFOの場合には、現在このメイルボックスでメッセージを待っているタスクのうち、最も早く待ち状態になったタスクのIDが返る。待ちタスクの無い時は FALSE (0) が返る。

pk_msg には、次に受信されるメッセージの先頭アドレスが返る。直後に rcv_msg を実行した場合には、同じ値が pk_msgとして戻される。メッセージが無い時は、pk_msg は NADR (-1) となる。また、どんな場合でも、pk_msg=NADR と wtskid=FALSE の少なくても一方は必ず成り立つ。

インプリメントによっては、pk_msg, wtskid 以外の追加情報も参照できる場合がある。また、pk_msg の代りに、まだ受信されていないメッセージの個数を参照できる場合もある。

mbx_sts で、対象メイルボックスが存在しない場合には、エラー E_NOEXS となる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（インプリメント依存）
E_ILADR	不正アドレス（リターンパラメータ用のパケットアドレスが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（mbxidのメイルボックスが存在しない）

タスク付属メイルボックスへ送信する [5] snd_tmb

タスク付属メイルボックスへ送信する（タスク独立部専用） [#5] isnd_tmb

snd_tmb: Send Message to Task Mailbox
isnd_tmb: Send Message to Task Mailbox (for Interrupt Handler)

【パラメータ】

tskid	TaskIdentifier	送信先タスクID
pk_msg	MessagePacket	送信メッセージの先頭アドレス

【リターンパラメータ】

なし

【解説】

tskidで指定したタスクに付属するメイルボックスに、pk_msg のアドレスに入っているメッセージを送信する。メッセージの内容はコピーされず、受信時にはアドレス（pk_msgの値）のみが渡される。送信されたメッセージは、FIFO でキューイングされる。

メッセージのキューをリングバッファによって実現するか、線形リストによって実現するかはインプリメント依存である。メッセージキューをリングバッファによって実現した場合に、リングバッファが一杯になってメッセージのキューイングができなくなっても、snd_tmb 発行タスクは待ち状態とはならない。この場合、snd_tmb発行タスクは E_QOVR のエラーを返して即座にリターンする。

タスク付属メイルボックスの機能は、ITRON1やITRON2ではサポートしていない。μITRONの独自機能である。

μITRONでは、OS内部でコンテキストを判断するオーバーヘッドを避けて性能を上げるため、タスク独立部（割込みハンドラ）から発行するシステムコールを、一般のタスクから発行するシステムコール snd_tmb から分離して別システムコール isnd_tmb とすることができる。OSの負担を減らし、

タスク独立部から発行するシステムコールの処理を高速化するためには、タスク独立部から発行するシステムコール isnd_tmb とタスク部から発行するシステムコール snd_tmb は分離されている方が良い。一方、ユーザから見ると、タスク部からもタスク独立部からも共通のシステムコール snd_tmb を発行できる方が便利である。どちらの仕様を用いるかはインプリメント側の選択項目となっている。isnd_tmb をサポートする方がOSの機能が高いというわけではない。

【エラーコード (ercd)】

E_OK	正常終了
E_CTX	コンテキストエラー（タスク部からisnd_tmbを実行）
E_RSID	予約ID番号（インプリメント依存）
E_ILADR	不正アドレス（pk_msgが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
E_ILMSG	不正メッセージ形式（インプリメント依存）
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_QOVR	キューイングのオーバーフロー（メッセージキューのオーバーフロー）

タスク付属メイルボックスからの受信を待つ [5] rcv_tmb

タスク付属メイルボックスから受信する [5] prcv_tmb

rcv_tmb:　Receive Message from Task Mailbox
prcv_tmb:　Poll and Receive Message from Task Mailbox

【パラメータ】

なし

【リターンパラメータ】

pk_msg	MessagePacket	受信メッセージの先頭アドレス

【解説】

rcv_tmb では、タスク付属メイルボックスからメッセージを受信し、受信したメッセージの先頭アドレスをリターンパラメータとして返す。タスク付属メイルボックスにまだメッセージが到着していない場合には、本システムコール発行タスクはメッセージ待ち状態となる。なお、本システムコールでメッセージを待つメイルボックスはタスク付属のものなので、それに対して待ち状態となるタスクは、高々一つだけである。したがって、rcv_tmb では、メッセージ待ちのタスクが待ち行列を作ることはない。

rcv_tmb ではタイムアウトの指定はできず、メッセージが受信できるまで永久に待つことになる。その代り、まだメッセージが到着していない場合にも即時にリターンするシステムコールとして、prcv_tmb が用意されている。prcv_tmb システムコールでは、既にメッセージが到着している場合には、そのメッセージを受信して正常終了する。メッセージが無い場合には、待ち状態に入らずにエラー E_PLFAIL を返す。

なお、tmb_sts を利用しても次に受信されるべきメッセージを知ることは可能であるが、prcv_tmb の場合は、メッセージが受信できる時には受信してしまう（先頭のメッセージをメッセージキューから削除する）という点が本質的に異なっている。

【エラーコード (ercd)】

E_OK	正常終了
E_CTX	コンテキストエラー（タスク独立部から実行）
E_RLWAI	待ち状態強制解除（待ちの間にrel_waiを受け付け）
E_PLFAIL	ポーリング失敗

タスク付属メイルボックスの状態を参照する [5] tmb_sts

tmb_sts: Get Task Mailbox Status

【パラメータ】

tskid	TaskIdentifier	タスクID

【リターンパラメータ】

tmbwait	Task Mailbox Wait	タスク付属メイルボックス待ち状態
pk_msg	MessagePacket	次のメッセージの先頭アドレス

【解説】

tmb_sts では、tskid で示されたタスクに付属するメイルボックスの各種の状態を参照し、待ちタスクの有無 tmbwait、次に受信されるメッセージの先頭アドレス pk_msg 等をリターンパラメータとして返す。

tmbwait は、ブール値を返すリターンパラメータである。tskid で示されるタスクがタスク付属メイルボックスを待っている状態の時には、tmbwaitにTRUE（0でない値、標準は1）が返り、そうでない時には FALSE(0) が返る。タスク付属メイルボックスの場合、待ちタスクのIDを知ることには意味が無いので、他のシステムコール（sem_sts,mbx_sts など）の wtskid に相当するリターンパラメータがブール値の tmbwait となっている。

pk_msg には、次に受信されるメッセージの先頭アドレスが返る。tskid で示されるタスクが直後に rcv_tmbを実行した場合には、同じ値が pk_msg として戻される。メッセージが無い時は、pk_msg は NADR (-1) となる。また、どんな場合でも、pk_msg = NADR と tmbwait = FALSE の少なくても一方は必ず成り立つ。

このシステムコールのインタフェース形式としては、パケットを利用して tmbwait, pk_msg 等の情報をまとめて読み出す場合、個々の情報を別々に読み出す場合のほか、tmbwait の情報をpk_msg やリターン値（エラーコード）にエンコードして読み出す場合がある。

インプリメントによっては、tmbwait, pk_msg 以外の追加情報も参照でき

る場合がある。また、pk_msg の代りに、まだ受信されていないメッセージの個数を参照できる場合もある。

【エラーコード(ercd)】

E_OK	正常終了
E_RSID	予約ID番号（インプリメント依存）
E_ILADR	不正アドレス（リターンパラメータ用のパケットアドレスが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
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）

割込み管理機能

割込みハンドラを定義する
[#1] def_int

def_int:　Define Interrupt Handler

【パラメータ】

intno	InterruptNumber	割込み番号
inthdr	InterruptHandlerAddress	割込みハンドラアドレス

【リターンパラメータ】

なし

【解説】

割込みハンドラを定義し、使用可能にする。すなわち、inthdr で示される割込みハンドラのアドレスと、intno で示される割込み番号との対応付けを行なう。

intnoの具体的な意味はプロセッサ依存であり、割込みベクトル番号、割込みデバイス番号などを表わす。また、プロセッサによっては、これ以外のパラメータ（割込みハンドラの属性やハンドラ起動時の割込みマスクの値など）の指定も行う場合がある。

ITRONやμITRONでは、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、ハードウエアの割込み処理メカニズムにより、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、割込みハンドラで使用するレジスタの退避と復帰は、ユーザが責任をもって行なう必要がある。

割込みハンドラは、ret_int 等のシステムコール、またはアセンブラの割込みリターン命令により終了する。

割込みハンドラ内のシステムコール発行によりタスクのディスパッチを生じた場合は、割込みハンドラが終了して ret_int 等のシステムコールによりOSにコントロールが戻ってくるまで、ディスパッチを遅らせるのが原則である。したがって、割込みハンドラ内でディスパッチを生ずる可能性がある場合、一般には、アセンブラの割込みリターン命令ではなく、必ず ret_int システムコールを使って割込みハンドラから復帰する必

要がある。

inthdr = NADR (-1) 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になる。ただし、def_intのシステムコールインタフェースがパケットを使っており、しかも、定義解除の対象を指定するパラメータ (intno) がパケットに含まれていない場合には、パケットアドレス= NADR (-1) によって定義解除を行うこともある。

def_int によって割込みハンドラを再定義する場合でも、あらかじめ前のハンドラの定義解除を行なっておく必要はない。既に割込みハンドラが定義された intno に対して直接新しいハンドラを定義しても、エラーにはならない。

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、待ちとなるシステムコールや、tskid = TSK_SELF によって自タスクの指定を意味するシステムコールは実行することはできない。

割込みハンドラ内において使用可能なシステムコールは、インプリメント依存である。また、μITRONの場合は、インプリメント依存の仕様として、次のような制限を設けることも許している。

— あるシステムコールをタスク独立部（割込みハンドラ）のみから発行できるように制限する。
— タスク部から発行するシステムコールとタスク独立部（割込みハンドラ）から発行するシステムコールを分離することによって、性能を上げる。すなわち、ITRONではタスク部発行とタスク独立部発行のシステムコールで同一の名称を使用しているのに対して、μITRONでは別名称を使っても良い。この場合、タスク部から発行するシステムコールの名称が XXX_YYY であれば、タスク独立部から発行するシステムコールの名称は iXXX_YYY となる。

μITRONでは、割込みハンドラを定義する機能が必須である。しかし、割込みハンドラの定義は、このシステムコールを使って動的に行う必要はなく、オブジェクトの生成と同様に、システム起動時に静的に行うこともできる。つまり、他の方法で割込みハンドラを定義することが可能であれば、必ずしもこのシステムコールをサポートする必要は無い。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足（インプリメント依存） ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSATR	予約属性（インプリメント依存）
E_PAR	一般的なパラメータエラー（インプリメント依存）
E_ILADR	不正アドレス（inthdrが使用できない値）
E_VECN	不正ベクトル番号（インプリメント依存）

割込みハンドラから復帰する
［#1］ret_int

ret_int:　　Return from Interrupt Handler

【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

【解説】

割込みハンドラからのリターンの際に発行するシステムコールである。

ITRONやμITRONでは、原則として、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、このシステムコールが発行されて割込みハンドラを抜けるまで、ディスパッチが遅延させられる。ただし、μITRONの場合、プロセッサのアーキテクチャや多重割込みの扱い方によっては、必ずしも遅延ディスパッチの原則が適用されるとは限らない。

このシステムコールを呼び出した時の状態（スタック、レジスタ等）は、割込みハンドラへ入った時の状態と同じでなければいけない。すなわち、割込みハンドラで使用するレジスタの退避や復帰は、ユーザが行わなければならない。

また、レジスタの復帰をユーザが行なう関係で、このシステムコールでは機能コードを使用できないことがある。したがって、このシステムコールの呼び出しの際、一般には、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る。

割込みハンドラから戻っても、必ず同じタスクが実行を継続することがわかっている場合には、ret_intではなくアセンブラのリターン命令によって割込みハンドラから戻っても構わない。

【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、インプリメント依存となる。

E_CTX　　コンテキストエラー（タスク部から実行）

割込処理復帰とタスク起床を行う　[3] ret_wup

ret_wup:　Return and Wakeup Task

【パラメータ】

tskid　TaskIdentifier　起床タスクID

【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

【解説】

割込みハンドラからリターンし、同時に tskid で示される割込み処理タスク（slp_tsk, wai_tsk で待っているタスク）を起床する。

ret_wup において、対象タスクがまだ slp_tsk, wai_tsk を実行していない場合には、要求がキューイングされる。

なお、ITRONやμITRONにおいて、ハンドラの終わりを示すシステムコールは、すべて 'ret' を付けるということで統一している。また、元のコンテキストに戻らないシステムコールは、すべて ret_XXX または ext_XXX (exd_XXX) の名称となっている。

【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、インプリメント依存となる。

E_CTX	コンテキストエラー（タスク部から実行）
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
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_DMT	タスクがDORMANTである（tskidのタスクがDORMANT）
E_QOVR	キューイングのオーバーフロー（wupcntのオーバーフロー）

タスクリスタートアドレスを定義する ［#5］def_rst

def_rst:　Define TaskRestartAddress

【パラメータ】

tskid	TaskIdentifier	タスクID
rstadr	RestartAddress	タスクリスタートアドレス

【リターンパラメータ】

なし

【解説】

tskidで指定したタスクのリスタートアドレスを定義する。ここで定義したアドレスは、ret_rst システムコール実行時に使用される。

タスクリスタート機能は、割込み処理からの戻り時に ret_rst を実行することにより、指定タスクを指定アドレスから実行再開する機能である。この機能は、実行再開を指定されたタスクから見ると、一種の例外処理（ソフトウエアによる強制例外）に相当するが、簡単な仕様にして高速化することを狙うため、μITRONでは例外処理と別扱いにしている。

この機能は、例えば、フォールトトレラントの目的で、実行プログラムをチェックポイントまで戻したいような場合に利用する。

対象タスクがDORMANT状態であっても、タスクリスタートアドレスを定義することが可能である。

タスクリスタートの機能は、ITRON1やITRON2ではサポートしていない。μITRONの独自機能である。

タスクリスタートの機能を提供する場合でも、タスクリスタートアドレスの定義は、このシステムコールを使って動的に行う必要はなく、オブジェクトの生成と同様に、システム起動時に静的に行う（タスクの属性の一つとして指定する）ことができる。つまり、他の方法でタスクリスタートアドレスを定義することが可能であれば、このシステムコールをサポートする必要は無い。この点は、def_int 等と同様である。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足（インプリメント依存） ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
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_ILADR	不正アドレス（rstadrが使用できない値）

割込処理復帰とタスクリスタートを行う　［5］ret_rst

ret_rst:　Return and Restart Task

【パラメータ】

tskid	TaskIdentifier	リスタートタスクID

【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

【解説】

割込みハンドラからリターンし、同時に tskid で指定されたタスクをリスタートさせる。

tskidで指定された対象タスクは、これ以前に、システム起動時の静的な指定あるいは def_rst システムコールによって、リスタートアドレス rstadr を定義している。このシステムコールでは、対象タスクのプログラムカウンタを rstadr に変更する。

tskidで指定されたタスクが待ち状態であった場合には、待ち状態が強制解除される。これは、割込みハンドラが、slp_tsk 等による割込み待ちタスクに対して、ret_wup による実行再開と、ret_rst による実行再開の二つの処理を指示する可能性があるためである。

【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、インプリメント依存となる。

E_CTX	コンテキストエラー（タスク部から実行）
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
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_DMT	タスクがDORMANTである（tskidのタスクがDORMANT）

割込みを禁止する

[2A] dis_int

dis_int:　Disable Interrupt

【パラメータ】

intno	InterruptNumber	割込み番号

【リターンパラメータ】

なし

【解説】

割込み番号 intno で指定される割込みを禁止する。

intnoの具体的な意味はプロセッサ依存であり、割込みベクトル番号、割込みデバイス番号などを表わす。

【エラーコード (ercd)】

E_OK	正常終了
E_PAR	一般的なパラメータエラー（インプリメント依存）
E_VECN	不正ベクトル番号（インプリメント依存）

割込みを許可する

［2A］ena_int

ena_int:　Enable Interrupt

【パラメータ】

intno	InterruptNumber	割込み番号

【リターンパラメータ】

なし

【解説】

割込み番号 intno で指定される割込みを許可する。

intnoの具体的な意味はプロセッサ依存であり、割込みベクトル番号、割込みデバイス番号などを表わす。

【エラーコード (ercd)】

E_OK	正常終了
E_PAR	一般的なパラメータエラー（インプリメント依存）
E_VECN	不正ベクトル番号（インプリメント依存）

割込みマスク（レベル, 優先度）を変更する　［2B］chg_iXX

chg_iXX:　Change Interrupt XXXX

【パラメータ】

iXXXX	Interrupt XXXX	割込みマスク（レベル、優先度）

【リターンパラメータ】

なし

【解説】

プロセッサの割込みマスク（レベル、優先度）を、iXXXX で指定した値に変更する。

システムコール名称およびパラメータ名称の一部の XXXX には、対象プロセッサのアーキテクチャに応じて適当な名前を入れる。具体的なシステムコール名は、例えば chg_ims, chg_ipl, chg_ilv 等となる。また、プロセッサによっては、このシステムコールが、iXXXX に加えて intno 等のパラメータを持つ場合がある。

通常、このシステムコールは、ユーザタスクの一部を割込み禁止にするために用いられる。

なお、このシステムコールにより割込みの禁止を行なっている間は、割込みハンドラ実行中と同じように、割込みマスクが解除されるまでディスパッチを遅延するのが原則である。これは、割込みを禁止したままでディスパッチが行なわれると、割込み禁止の状態が中断したり、命令の実行に矛盾が起こったりするためである。

【エラーコード (ercd)】

E_OK	正常終了
E_PAR	一般的なパラメータエラー（インプリメント依存）
E_VECN	不正ベクトル番号（インプリメント依存）
E_IMS	不正IMASK（インプリメント依存）

割込みマスク（レベル, 優先度）を参照する ［3B］iXX_sts

iXX_sts:　Get Interrupt XXXX Status

【パラメータ】

なし

【リターンパラメータ】

iXXXX	Interrupt XXXX	割込みマスク（レベル、優先度）

【解説】

プロセッサの割込みマスク（レベル、優先度）を参照し、リターンパラメータ iXXXX として返す。

システムコール名称およびパラメータ名称の一部の XXXX には、対象プロセッサのアーキテクチャに応じて適当な名前を入れる。具体的なシステムコール名は、例えば ims_sts, ipl_sts, ilv_sts 等となる。また、プロセッサによっては、このシステムコールが intno 等のパラメータを持つ場合がある。

【エラーコード (ercd)】

E_OK	正常終了
E_PAR	一般的なパラメータエラー（インプリメント依存）
E_VECN	不正ベクトル番号（インプリメント依存）

例外管理機能

ITRONでは、システムコール実行のエラーにより例外ハンドラを起動するシステムコール例外の機能と、プロセッサの例外処理（ゼロ除算等）によってタスクの例外ハンドラを起動するCPU例外の機能があった。しかし、システムコール例外の機能はシステムコール実行のオーバーヘッドになりやすいので、μITRON標準仕様には含めない。また、CPU例外の管理機能も、プロセッサのアーキテクチャに対する依存性が高いため、μITRONの標準仕様には含めない。μITRONの例外管理機能は、すべてプロセッサ依存あるいはインプリメント依存となっている。

μITRONの用途では、必ずしも例外管理の機能は必要ではないが、デバッグサポートなどのために、インプリメントに依存した独自の例外管理機能を設けることは可能である。

メモリプール管理機能

この章の「メモリプール管理機能」で対象としているのは、ソフトウエアによるメモリプールやメモリブロックの管理機能である。MMU に関する操作（MMUサポート機能）は、ここには含まれない。

μITRONでは、タスク間で共有される固定長メモリブロックのみを扱うのが普通である。

固定長メモリブロックの獲得待ちを行う [4] get_blk

固定長メモリブロックを獲得する [3] pget_blk

get_blk:　Get Memory Block
pget_blk:　Poll and Get Memory Block

【パラメータ】

mplid	MemoryPoolIdentifier	メモリプールID

【リターンパラメータ】

blk	BlockStartAddress	メモリブロックの先頭アドレス

【解説】

mplidで示されるメモリプールから、共有メモリブロック（固定長）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。獲得されるメモリブロックのサイズは、システム起動時にメモリブロック毎に指定した値 (blksz) となる。

get_blk で獲得したメモリのゼロクリアは特に行われない。獲得されたメモリブロックの内容は不定である。

get_blk の場合、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、get_blk 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、獲得できるようになるまで待つ。一方、メモリブロックが獲得できない場合でも即座にリターンするシステムコールとして、pget_blk が用意されている。pget_blk システムコールでは、mplid で示されたメモリプールから即座にメモリブロックが獲得できる場合には、メモリブロックの獲得を行って、システムコールは正常終了する。即座にメモリブロックが獲得できない場合には、待ち状態に入らずにエラー E_PLFAIL を返す。

pget_blk の機能は、ITRON2では get_blk (tmout=TMO_POL) の指定により実現される機能であるが、μITRONの場合は get_blk でタイムアウトの指定ができないため、別システムコール pget_blk に分離している。なお、

リアルタイムOSの場合は、メモリ獲得待ちでタスクの実行を止めるということはあまり無い。したがって、get_blk よりも pget_blk の方が必要性が高い。

μITRONの場合、メモリプールの生成はシステム起動時に静的に行う。メモリプール生成時に必要なメモリブロックサイズ (blksz) やメモリプール全体のサイズ(mplsz) などのパラメータも、システム起動時に静的に指定する。

μITRONのメモリプールの場合、get_blk における待ちタスクのキューイング方法はFIFO (TA_TFIFO) とするのが標準である。ただし、インプリメントによっては、タスク優先度順 (TA_TPRI) のキューイング方法を選択できるようになっていても良い。また、μITRONの get_blkでは固定長のメモリブロックのみを扱うのが標準なので、ITRON2における TA_FIRST（行列先頭のタスクを優先扱い）と TA_CNT（要求数の少ないタスクを優先扱い）の区別は意味を持たない。

μITRONでは固定長のメモリブロックしかサポートしないので、メモリプールが1つしかないとすると、1種類のサイズのメモリブロックしか確保できないことになる。そこで、μITRONでも get_blk, pget_blk システムコールのパラメータとして mplid を含めることにより、複数のメモリプールを持つことが可能となるようにしている。

資源獲得を行うシステムコールのうち、pget_blk に限っては、次のような理由により、例外的にタスク独立部からも実行できる場合がある。(実際に実行可能かどうかはインプリメント依存である。)

— 一般的には、セマフォカウント値やメモリブロックなどの「資源」は、タスクに与えられるものであるということになっている。したがって、割込みハンドラ等のタスク独立部では、資源を確保することができず、タスク独立部から wai_sem 等を発行することはできない。
— しかし、タスク独立部が資源を確保したとしても、リターンの前に資源を返してしまう（あるいは他のタスクやメイルボックスに渡してしまう）ことにすれば、特に矛盾は生じない。実際、割込みハンドラの中からメッセージを送る場合には、その前に get_blk を実行する必要が生じる。もちろん、割込みハンドラでは待ち状態になることはできないので、get_blkではなく、pget_blk を発行しなければならない。
— pget_blk 以外の pol_flg, preq_sem, prsv_msg などをタスク独立部から実行

する必要性はない。pget_blk 以外のシステムコールに関しては、やはり E_CTX のエラーとする。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（mplidのメモリプールが存在しない）
E_CTX	コンテキストエラー（タスク独立部から実行、インプリメント依存）
E_RLWAI	待ち状態強制解除（待ちの間にrel_waiを受け付け）
E_PLFAIL	ポーリング失敗

固定長メモリブロックを返却する [3] rel_blk

rel_blk: Release Memory Block

【パラメータ】

mplid	MemoryPoolIdentifier	メモリプールID
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】

なし

【解説】

blk で示されるメモリブロックを、mplidで示されるメモリプールへ返却する。

メモリブロックの返却を行うメモリプールは、メモリブロックの獲得を行った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_ILBLK のエラーとなる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（インプリメント依存）
E_IDOVR	ID範囲外（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（mplidのメモリプールが存在しない）
E_ILBLK	不正メモリブロックの返却、操作

メモリプールの状態を参照する　[4] mpl_sts

mpl_sts:　Get MemoryPool Status

【パラメータ】

mplid	MemoryPoolIdentifier	メモリプールID

【リターンパラメータ】

wtskid	WaitTaskID	待ちタスクID
frbcnt	FreeBlockCount	空き領域のブロック数

【解説】

mpl_sts では、mplid で示された対象メモリプールの各種の状態を参照し、対象メモリプールの現在の空きブロック数 frbcnt、待ちタスクID番号 wtskid 等をリターンパラメータとして返す。

このシステムコールのインタフェース形式としては、パケットを利用して frbcnt, wtskid 等の情報をまとめて読み出す場合と、個々の情報を別々に読み出す場合とがある。

wtskid には、待ち行列の先頭のタスクのIDが返る。すなわち、待ち行列へのつながれ方がFIFOの場合には、現在このメモリプールからメモリブロックの獲得を待っているタスクのうち、最も早く待ち状態になったタスクのIDが返る。待ちタスクの無い時は FALSE (0) が返る。frbcntと wtskid の少なくとも一方は0となる。

インプリメントによっては、frbcnt, wtskid 以外の追加情報（メモリプール全体のサイズmplszやブロックサイズblkszなど）も参照できる場合がある。

mpl_sts で、対象メモリプールが存在しない場合には、エラー E_NOEXS となる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（インプリメント依存）
E_ILADR	不正アドレス（リターンパラメータ用のパケットアドレスが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（mplidのメモリプールが存在しない）

時間管理・タイマハンドラ機能

システムクロックを設定する [2] set_tim

set_tim: Set Time

【パラメータ】

time	CurrentDateTime	現在の年月日時刻データ

【リターンパラメータ】

なし

【解説】

システムが保持しているシステムクロックの現在の値を、time で示される値に設定する。

システムクロックのビット数（timeのビット数）はインプリメント依存であるが、48ビットを推奨する。timeとして必要なビット数のデータを一度に扱えないプロセッサの場合には、パラメータの time を上位 (utime) と下位 (ltime) あるいは上位 (utime) と中位 (mtime) と下位 (ltime) に分けて扱うのが普通である。

システムクロックの意味や単位もインプリメント依存であるが、他の制約が無い場合には、1985年1月1日0時 (GMT) からの通算のミリ秒数とすることを推奨する。

システムの動作中に set_tim を使ってシステムクロックを変更した場合には、それまで時間待ちをしていたタスクや、起動を待っていたハンドラ（周期起動ハンドラやアラームハンドラ）の動作のタイミングが狂う可能性がある。このような場合の動作はインプリメント依存である。

【エラーコード (ercd)】

E_OK	正常終了
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_ILTIME	不正時間指定（インプリメント依存）

システムクロックの値を読み出す　[2] get_tim

get_tim:　Get Time

【パラメータ】

なし

【リターンパラメータ】

time	CurrentDateTime	現在の年月日時刻データ

【解説】

システムが保持しているシステムクロックの現在の値を読み出し、リターンパラメータ time に返す。

システムクロックのビット数（timeのビット数）はインプリメント依存であるが、48ビットを推奨する。time として必要なビット数のデータを一度に扱えないプロセッサの場合には、リターンパラメータの time を上位 (utime) と下位 (ltime) あるいは上位 (utime) と中位 (mtime) と下位 (ltime) に分けて扱うのが普通である。

システムクロックの意味や単位もインプリメント依存であるが、他の制約が無い場合には、1985年1月1日0時 (GMT) からの通算のミリ秒数とすることを推奨する。

【エラーコード (ercd)】

E_OK正常終了

E_TNOSPT　タイマ関係の未サポート機能（タイマが利用できない）
※ このエラーが発生するかどうかはインプリメント依存である。

周期起動ハンドラを定義する ［#4］def_cyc

def_cyc:　Define Cyclic Handler

【パラメータ】

cyhno	CyclicHandlerNumber	周期起動ハンドラ指定番号
cychdr	CyclicHandlerAddress	周期起動ハンドラアドレス
cyhact	CyclicHandlerActivation	周期起動ハンドラ活性状態
cytime	CycleTime	周期起動時間間隔

【リターンパラメータ】

なし

【解説】

周期起動ハンドラは、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。このシステムコールでは、周期起動ハンドラを定義する。

cyhno は複数の周期起動ハンドラを区別するための番号であり、act_cyc, cyh_sts 等で使用される。また、cycadr は周期起動ハンドラの先頭アドレス、cytime は周期起動の時間間隔、cyhact は周期起動ハンドラの一時的なON/OFFの選択（活性状態）を表わす。cyhact では、次のような値を指定する。

cyhact : = (TCY_OFF ‖ TCY_ON)

TCY_OFF	周期起動ハンドラは起動されない
TCY_ON	周期起動ハンドラは起動される

cyhact = TCY_OFF は周期起動ハンドラの一時的な中断を意味し、この間はハンドラが起動されない。

def_cyc の実行により周期カウントがクリアされるため、def_cyc を実行してからちょうど指定周期経った後に、最初のハンドラ起動が起こることになる。

cychdr = NADR(-1) とした場合には、前に定義した周期起動ハンドラの定義解除になる。ただし、def_cyc のシステムコールインタフェースがパケットを使っており、しかも、定義解除の対象を指定するパラメータ (cyhno) が

パケットに含まれていない場合には、パケットアドレス= NADR (-1) によって定義解除を行うこともある。

def_cyc では回数の指定が無いので、act_cyc によって cyhact を TCY_OFF にするか、ハンドラを定義解除するまで周期起動が繰り返される。また、既に定義されている周期起動ハンドラの cyhno に対して、再度def_cyc が実行された場合には、前の定義がすべて取り消され、新しい定義が有効になる。def_cyc によって周期起動ハンドラを再定義する場合でも、あらかじめ前のハンドラの定義解除を行なっておく必要はない。これは、def_int などと同様である。

周期起動ハンドラでのレジスタの退避と復帰はユーザの責任である。

周期起動ハンドラはタスク独立部として実行されるため、周期起動ハンドラの中では、待ちとなるシステムコールや、tskid = TSK_SELF によって自タスクの指定を意味するシステムコールは実行することはできない。また、周期起動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、ret_tmr システムコールの発行により周期起動ハンドラを抜けるまで、ディスパッチを遅延するのが原則である。

周期起動ハンドラやアラームハンドラをタスク独立部と考えているのは、周期起動ハンドラやアラームハンドラの定義が有効な間に、def_cyc, def_alm を発行したタスクが終了、削除される可能性があるためである。

周期起動ハンドラはオブジェクトではないため、ID番号ではなく、cyhno により複数ハンドラの区別を行なっている。

周期起動ハンドラの機能を提供する場合でも、周期起動ハンドラの定義は、このシステムコールを使って動的に行う必要はなく、オブジェクトの生成と同様に、システム起動時に静的に行うことができる。つまり、他の方法で周期起動ハンドラを定義することが可能であれば、このシステムコールをサポートする必要は無い。この点は、def_int 等と同様である。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足（インプリメント依存） ※ このエラーが発生するかどうかはインプリメント依存である。
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSMD	予約モード、予約オプション（cyhactが不正）
E_RSATR	予約属性（インプリメント依存）
E_PAR	一般的なパラメータエラー（cyhnoが不正）
E_ILADR	不正アドレス（cychdrが使用できない値）
E_ILTIME	不正時間指定（インプリメント依存）

周期起動ハンドラの活性制御を行なう　[4] act_cyc

act_cyc:　Activate Cyclic Handler

【パラメータ】

cyhno	CyclicHandlerNumber	周期起動ハンドラ指定番号
cyhact	CyclicHandlerActivation	周期起動ハンドラ活性状態

【リターンパラメータ】

なし

【解説】

周期起動ハンドラの活性状態 (cyhact) を変更する。
cyhact の具体的な指定方法は次のようになる。

cyhact : = (TCY_OFF ‖ TCY_ON) | [TCY_INI]

TCY_OFF	周期起動ハンドラは起動されない
TCY_ON	周期起動ハンドラは起動される
TCY_INI	周期起動ハンドラのカウントが初期化される

cyhact = TCY_ON の指定を行うと、周期の経過とは独立に活性状態を ON にする。OS内部の動作としては、活性状態が OFF の間も指定周期のカウントを行っているため、act_cyc を実行してから最初に周期起動ハンドラが実行されるまでの時間は一定しない。

一方、cyhact = (TCY_ON | TCY_INI) の指定を行うと、活性状態を ON にするのと同時に周期カウントをクリアする。したがって、act_cyc を実行してからちょうど指定周期経った後に、最初のハンドラ起動が起こることになる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSMD	予約モード、予約オプション（cyhactが不正）
E_NOEXS	オブジェクトが存在していない（cyhnoの周期起動ハンドラが存在しない）

周期起動ハンドラの状態を参照する [4] cyh_sts

cyh_sts: Get Cyclic Handler Status

【パラメータ】

cyhno	CyclicHandlerNumber	周期起動ハンドラ指定番号

【リターンパラメータ】

cyhact	CyclicHandlerActivation	周期起動ハンドラ活性状態
lftime	LeftTime	次のハンドラ起動までの残り時間

【解説】

cyh_sts では、cyhno で示された周期起動ハンドラの状態を参照し、周期起動ハンドラ活性状態 cyhact、次のハンドラ起動までの残り時間 lftime 等をリターンパラメータとして返す。このうち、cyhact では、TCY_ONと TCY_OFF の区別のみが返され、TCY_INI の情報が返ることはない。

このシステムコールのインタフェース形式としては、パケットを利用して cyhact, lftime 等の情報をまとめて読み出す場合と、個々の情報を別々に読み出す場合とがある。

インプリメントによっては、cyhact, lftime 以外の追加情報（周期cytimeなど）も参照できる場合がある。

cyh_sts で、cyhno の周期起動ハンドラが存在しない場合には、エラー E_NOEXS となる。

【エラーコード (ercd)】

E_OK	正常終了
E_PAR	一般的なパラメータエラー（インプリメント依存）
E_ILADR	不正アドレス（リターンパラメータ用のパケットアドレスが使用できない値）
E_NOEXS	オブジェクトが存在していない（cyhnoの周期起動ハンドラが存在しない）

アラームハンドラを定義する　[#4] def_alm

def_alm:　Define Alarm Handler

【パラメータ】

alhno	AlarmHandlerNumber	アラームハンドラ指定番号
almhdr	AlarmHandlerAddress	アラームハンドラアドレス
time	AlarmTime	ハンドラ起動時刻
tmmode	TimeMode	初期時刻指定モード

【リターンパラメータ】

なし

【解説】

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。このシステムコールでは、指定時刻起動ハンドラを定義する。

alhnoは複数のアラームハンドラを区別するための番号である。almadr は起動されるアラームハンドラの先頭アドレスを表わす。また、time により、アラームハンドラの起動時刻を指定する。tmmode により、次のようにして絶対時刻と相対時刻の指定が可能である。

tmmode : = (TTM_ABS ‖ TTM_REL)

TTM_ABS	絶対時刻での指定
TTM_REL	相対時刻での指定

指定した時刻が過去の時刻であった場合には、エラー E_ILTIME となる。

time のビット数はインプリメント依存であるが、システムクロックのビット数と同じく48ビットを使用するのが望ましい。time として必要なビット数のデータを一度に扱えないプロセッサの場合には、パラメータの time を上位 (utime) と下位 (ltime) あるいは上位 (utime) と中位 (mtime) と下位 (ltime) に分けて扱うのが普通である。

almhdr = NADR (-1) とした場合には、前に定義したアラームハンドラの

定義解除になる。ただし、def_alm のシステムコールインタフェースがパケットを使っており、しかも、定義解除の対象を指定するパラメータ (alhno) がパケットに含まれていない場合には、パケットアドレス= NADR (-1) によって定義解除を行うこともある。

def_alm によってアラームハンドラを再定義する場合でも、あらかじめ前のハンドラの定義解除を行なっておく必要はない。既にアラームハンドラが定義された alhno に対して直接新しいハンドラを定義しても、エラーにはならず、前の定義が取り消されて新しい定義が有効になる。これは、def_int などと同様である。

アラームハンドラでのレジスタの退避と復帰はユーザの責任である。

アラームハンドラはタスク独立部として実行されるため、アラームハンドラの中では、待ちとなるシステムコールや、tskid = TSK_SELF によって自タスクの指定を意味するシステムコールは実行することはできない。また、アラーム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、ret_tmr システムコールの発行によりアラームハンドラを抜けるまで、ディスパッチを遅延するのが原則である。

アラームハンドラはオブジェクトではないため、ID番号ではなく、alhno により複数ハンドラの区別を行なっている。

アラームハンドラの機能を提供する場合でも、アラームハンドラの定義は、このシステムコールを使って動的に行う必要はなく、オブジェクトの生成と同様に、システム起動時に静的に行うことができる。つまり、他の方法でアラームハンドラを定義することが可能であれば、このシステムコールをサポートする必要は無い。この点は、def_int 等と同様である。

アラームハンドラの中で rel_wai, irel_wai を発行し、待ち状態のタスクを待ち解除とすることにより、タイムアウトに似た機能を実現することができる。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足（インプリメント依存） ※ このエラーが発生するかどうかはインプリメント依存である。
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSMD	予約モード、予約オプション（tmmode が不正）
E_RSATR	予約属性（インプリメント依存）
E_PAR	一般的なパラメータエラー（alhno が不正）
E_ILADR	不正アドレス（almhdr が使用できない値）
E_ILTIME	不正時間指定（インプリメント依存）

アラームハンドラの状態を参照する [4] alh_sts

alh_sts: Get Alarm Handler Status

【パラメータ】

alhno	AlarmHandlerNumber	アラームハンドラ指定番号

【リターンパラメータ】

lftime	LeftTime	次のハンドラ起動までの残り時間

【解説】

alh_sts では、alhno で示されたアラームハンドラの状態を参照し、次のハンドラ起動までの残り時間lftime 等をリターンパラメータとして返す。

インプリメントによっては、lftime 以外の追加情報も参照できる場合がある。

このシステムコールのインタフェース形式としては、パケットを利用して lftime やそれ以外の追加情報をまとめて読み出す場合と、個々の情報を別々に読み出す場合とがある。また、lftime に必要なビット数のデータを一度に扱えないプロセッサの場合には、リターンパラメータの lftime を上位 (lfutime) と下位 (lfltime) あるいは上位 (lfutime) と中位 (lfmtime) と下位 (lfltime) に分けて扱うのが普通である。

alh_sts で、alhno のアラームハンドラが存在しない場合には、エラー E_NOEXS となる。

【エラーコード (ercd)】

E_OK	正常終了
E_PAR	一般的なパラメータエラー（インプリメント依存）
E_ILADR	不正アドレス（リターンパラメータ用のパケットアドレスが使用できない値）
E_NOEXS	オブジェクトが存在していない（alhnoのアラームハンドラが存在しない）

タイマハンドラから復帰する
［#4］ret_tmr

ret_tmr:　　Return from Timer Handler

【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

【解説】

タイマハンドラ（周期起動ハンドラ、アラームハンドラ）から戻る際に発行するシステムコールである。

このシステムコールでは、ret_int と同様に、タイマハンドラの中で遅延させられていたディスパッチの起こる可能性がある。

プロセッサのアーキテクチャやOSのインプリメントによっては、サブルーチンリターン命令や割込みリターン命令によって、このシステムコールを実現する場合がある。

なお、ret_tmr を、同じタスク独立部からの戻りシステムコールである ret_int と統合しないのは、ハンドラ起動時にOSが介入するかどうかという点で違いがあるためである。すなわち、ret_int がハードウエアで直接起動されたタスク独立のハンドラからのリターンを表わすのに対して、ret_tmr はOSから起動されたタスク独立のハンドラからのリターンを表わす。

【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、インプリメント依存となる。

E_CTX　　コンテキストエラー（タスク部から実行）

システム管理機能

ITRON, μITRONのバージョン番号を得る [1] get_ver

get_ver:　Get Version NO

【パラメータ】

pk_ver	Packet of Version Numbers	バージョン管理情報を返すパケットの先頭アドレス

【リターンパラメータ】

なし

【解説】

現在実行中のITRON仕様OSやμITRON仕様OSのスペックの概要、OSの形式番号、バージョン番号などを得る。

このシステムコールは、OSのメーカーおよびバージョン番号を、プログラムから読み出すことを目的としたものである。

バージョンの読み出しは、次のようなパケットを用いて行なわれる。なお、以下で、UH は 16ビット符号無し整数のデータタイプを表わす。

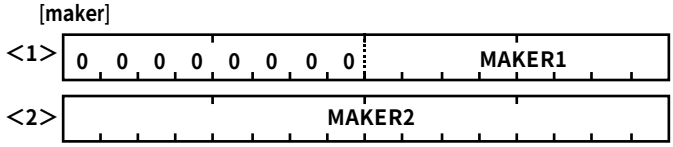
```
typedef struct t_ver {
    UH      maker;          /* メーカー */
    UH      id;             /* 形式番号 */
    UH      spver;          /* 仕様書バージョン */
    UH      prver;          /* 製品バージョン */
    UH      prno[4];        /* 製品管理情報 */
    UH      cpu;            /* CPU情報 */
    UH      var;            /* バリエーション記述子 */
} T_VER;

T_VER   *pk_ver;    /* バージョン管理ブロックへのポインタ */
```

パケットの形式や構造体の各メンバのフォーマットは、プロセッサ間で、あるいはITRON1、ITRON2、μITRON、BTRONの間でほぼ共通になっている。以下では、各メンバのフォーマットや意味について説明を行う。

makerでは、この製品 (ITRON、BTRON、TRONCHIP) を作ったメーカーを表わす。maker のフォーマットを［図2.10(a)］に示す。

ITRON、BTRON、TRONCHIPのうちの複数のプロジェクトに参加しているメーカーに対しては、協議の上［図2.10 (a)］＜1＞のフォーマットを割り当てる。この場合、MAKER1 のコードは、ITRON、BTRON、TRONCHIP で共通になる。それ以外のメーカーの場合は、［図2.10 (a)］＜2＞のフォーマットを使用し、B'0000000100000001～ のコードを割り当てる。MAKER2 のコードは登録制となる。MAKER2のコードは、ITRON、BTRON、TRONCHIP の間で異なったものになる。

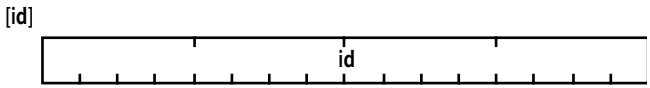
なお、MAKER1 のコードを持つメーカーの子会社や関連会社の場合は、そのメーカーの判断によって、MAKER1のコード番号を使うか、新しく登録する MAKER2のコードを使うかということを決める。



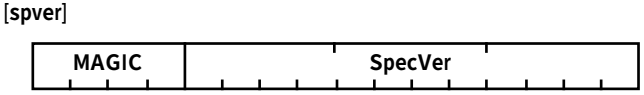
MAKER1 : メーカー番号

B' 00000000	バージョンなし〔実験システムなど〕
B' 00000001	University of TOKYO
...	
	＊以下はABC順
B' 00001001	FUJITSU
B' 00001010	HITACHI
B' 00001011	MATSUSHITA
B' 00001100	MITSUBISHI
B' 00001101	NEC
B' 00001110	OKI
B' 00001111	TOSHIBA
B' 00010000～B' 11111111	
	reserved

[図2.10 (a)] get_verで得られるmakerのフォーマット



[図2.10 (b)] get_verで得られるidのフォーマット



MAGIC: TRONのシリーズを区別する番号

H' 0	TRON共通(TAD等)
H' 1	ITRON
H' 2	BTRON
H' 3	CTRON
H' 4	reserved
H' 5	μITRON
H' 6	μBTRON

SpecVer:この製品のものになった東京大学のTRON仕様書のバージョン番号。3桁のバックト型式BCDコードで入れる。

[図2.10 (c)] get_verで得られるspverのフォーマット

id は、OSやVLSIの種類を区別する番号である。id のフォーマットを［図2.10(b)］に示す。1つのメーカーの中で、MMU対応版とMMU非対応版のような複数の種類の製品を作った場合、それらを区別するためにidを用いることができる。

idの番号の付け方はメーカーの自由とする。ただし、製品の区別はあくまでもこの番号のみで行なうので、各メーカー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。

spver では、ITRON, μITRON, BTRON, CTRON, TRONCHIPの区別と、この製品のもとになった仕様書のバージョン番号を表わす。spver のフォーマットを［図2.10(c)］に示す。

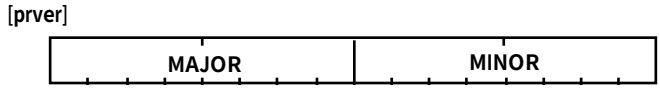
μITRONは、ITRON2との整合性が高いという意味で、最初のバージョンが 1.00 ではなく 2.00 となる。たとえば、μITRONの最初のバージョンに対応する spver は次のようになる。

MAGIC = H'5 (μITRON)
SpecVer = H'200 (Ver 2.00)
spver = H'5200

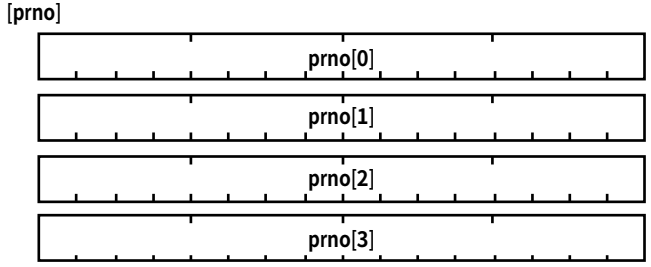
また、μITRON Ver 2.56.xx.xx の仕様書をインプリメントした製品の場合、
spver = H'5256
となる。

prver では、内部インプリメント上のバージョン番号を表わす。prver のフォーマットを［図2.10 (d)］に示す。MAJOR はバージョン番号の大きな区別、MINOR はバージョン番号の小さな区別を表わす。番号の付け方はメーカーの自由である。

prno は、製品管理情報や製品番号などを入れるために、各メーカーが自由に使用してもよい部分である。prno のフォーマットを

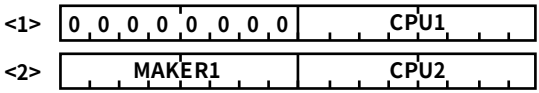


［図2.10 (d)］get_verで得られるprverのフォーマット



［図2.10 (e)］get_verで得られるprnoのフォーマット

[cpu]



CPU1:

B'00000000 (H'00) CPUを特定しない、CPU情報を持たない
B'00000001 (H'01) TRONCHIP32共通、メーカーを特定しない
B'00000010 (H'02) reserved
B'00000011 (H'03) reserved
B'00000100 (H'04) reserved
B'00000101 (H'05) reserved (≪L1R≫仕様のTRONCHIIP)
B'00000110 (H'06) reserved (≪L1≫仕様の仕様のTRONCHIIP)
B'00000111 (H'07) reserved (LSIDの機能を持つTRONCHIIP)
＊ B' 00000000～B' 00000111 の場合、CPUのメーカーを特定しない。

B'00001000 (H'08) reserved
B'00001001 (H'09) GMICRO/100
B'00001010 (H'0a) GMICRO/200
B'00001011 (H'0b) GMICRO/300
B'00001100 (H'0c) reserved
B'00001101 (H'0d) TX1
B'00001110 (H'0e) reserved
B'00001111 (H'0f) TX3

B'00010000 (H'10) reserved
B'00010001 (H'11) reserved
B'00010010 (H'12) reserved
B'00010011 (H'13) 032

B'00010100 (H'14) reserved
B'00010101 (H'15) MN10400
B'00010110 (H'16) reserved
B'00010111 (H'17) reserved

B' 00011000～B' 00111111 (H'18～H'3f) - reserved
＊GMICRO拡張用、TXシリーズ拡張用、TRONCHIP64用

[図2.10 (f)] get_verで得られるcpuのフォーマット（前半）

```
B'01000000 (H'40) 68000
B'01000001 (H'41) 68010
B'01000010 (H'42) 68020
B'01000011 (H'43) 68030
B'01000000～B'01001111 (H'40～H'4f) - 68000 系

B'01010000 (H'50) 32032
B'01010000～B'01011111(H'50～H'5f) - NS32000 系

B'01100000 (H'60) 8086, 8088
B'01100001 (H'61) 80186
B'01100010 (H'62) 80286
B'01100011 (H'63) 80386
B'01100000～B'01101111(H'60～H'6f) - 86 系

B'01110000～B'01111111(H'70～H'7f) - NEC V シリーズ
                    割り当てはVシリーズ関連メーカーが決める。

B' 10000000～B' 11111111 (H' 80～H' ff) - reserved
```

[図2.10 (f)] get_verで得られるcpuのフォーマット（後半）

［図2.10 (e)］に示す。

cpu は、このITRONを実行するプロセッサを表わす。cpu のフォーマットを［図2.10 (f)］に示す。TRONCHIP、モトローラM68000系、インテルiAPX86系、ナショセミNS32000系、NEC-Vシリーズのプロセッサの場合は［図2.10 (f)］＜1＞のフォーマットを使用し、それ以外の各社独自アーキテクチャのプロセッサの場合には、［図2.10 (f)］＜2＞のフォーマットを使用する。

［図2.10 (f)］＜1＞のフォーマットにおける CPU1 のコード割当ては、ITRONとBTRONで共通である。また、BTRON / CHIP 標準オブジェクトフォーマットのCPUタイプの部分にも、これと同じ番号が入る。

一方、［図2.10 (f)］＜2＞のフォーマットは、主としてμITRON で用いられるものである。この場合、MAKER1 のコードの割当ては、maker の項で説明したものと同じになる。また、CPU2 のコードの割当ては MAKER1のメーカーが決める。

var では、このITRON2、μITRONで利用できる機能の概要を表わす。var のフォーマットを［図2.10 (g)］に示す。

なお、製品の種類の区別はあくまでも idの部分で行なう。var, cpu による表示は便宜的なものに過ぎない。すなわち、var, cpu が異なり、かつ id の等しい製品が存在してはいけない。

μITRONの場合、M, V, P のフィールドは 0 となる。

【エラーコード (ercd)】

E_OK	正常終了
E_ILADR	不正アドレス（pk_verが使用できない値）

[var]

–: reserved（0 が返る）

LEV: カーネル仕様のレベル分け
- B' 000 μITRON
- B' 001 reserved
- B' 010 reserved
- B' 011 ITRON2基本仕様（≪I1≫仕様）
- B' 100 ITRON2一部拡張仕様（一部≪I2≫仕様）
- B' 101 ITRON2拡張仕様（≪I2≫仕様）
- B' 110 reserved
- B' 111 reserved

M=1: マルチプロセッササポート
M=0: シングルプロセッサ用
V=1: 仮想記憶サポート
P=1: MMU対応版
FIL: ファイル仕様のレベル分け
- B' 000 サポートなし
- B' 001 ≪FI0≫仕様
- B' 010 ≪FI1≫仕様
- B' 011～B' 111

IO: 入出力仕様
- B' 00 サポートなし
- B' 01 標準仕様
- B' 10 reserved
- B' 11 reserved

［図2.10 (g)］get_verで得られるvarのフォーマット

第二部 第三章

μITRON標準インタフェース

この章では、μITRON仕様で推奨しているシステムコールインタフェースに関して説明を行う。具体的には、機能コード、エラーコード、言語Cインタフェースなどの説明がここに含まれる。

アセンブラインタフェース

μITRONの各システムコールのアセンブラインタフェースは、プロセッサ毎に定める。ただし、機能コードやフラグ変化など一部の点については、μITRON全体として、この節で述べるような原則を設けている。他の制約が無い場合には、できるだけこれらの原則に合わせて頂けるのが望ましい。

システムコール実行後のフラグ変化

システムコール実行後のフラグの値については、特に規定せず、不定値になるものとする。

ITRON1では、エラーの有無に応じてゼロフラグのセットを行うことを推奨していた。しかし、OSがせっかくフラグをセットしても、必ずしも参照されるとは限らないし、言語Cを使えば全く不要な機能である。また、フラグセットのためのオーバーヘッドも無視できない。

プロセッサによっては、OSで特にフラグの操作をしない場合に、トラップ命令のハードウエア処理によって自然にフラグを保存できることがある。しかし、将来システムコールをハードウエア化することがあれば、フラグのセットを行う方が望ましいということになる可能性もあり、必ずしもフラグを保存するように規定するのが良いとは言えない。そこで、現段階では、システムコール実行後のフラグ値に関して規定を設けないことにする。

機能コード

ITRONでは、システムコールの種類を区別する番号として、「機能コード」を用いている。機能コードをあるレジスタに置き、それからトラップ命令を発行することによって、対応するシステムコールが実行される。

μITRONの場合、プロセッサアーキテクチャや開発環境との関係によっては、トラップ命令（ソフトウエア割込み命令）ではなく、サブルーチンジャンプ命令によって、OS内のシステムコール処理ルーチンに直接ジャンプするケースがある。そのような場合には、機能コードを使用しないことがある。しかし、μITRONでも、機能コードを使ったシステムコールインタ

フェースを行うのであれば、特に他の制約が無い限り、機能コードの値を標準化しておくのが望ましい。

ITRON2およびμITRONにおける標準機能コードの割当方針を以下に示す。

【機能コードの割当て方針】

― TRONの全体方針に合わせて、機能コードの標準値には負の値を使用している。
― 8の倍数に対するアライメントや、共通機能（cre_XXX, XXX_sts, uXXX_YYY, sXXX_YYY など）に対する共通ビットパターンを考慮した。
― 必ずしも、仕様書の出現順に機能コードを割り当てるわけではない。ただし、仕様書の機能グループ毎には、まとまった範囲の機能コードを割当てるようにする。（強制例外などシステムコール数が少ないものについては、他のグループにまとめられる場合もある。）
また、仕様書での説明の順序は、機能コードとは直接関係しない。
― μITRONで使用する iXXX_YYY, pXXX_YYY は、それぞれ一つのグループを作るので、一般のシステムコールと混在させるのではなく、番号を飛ばした別グループとしている。

μITRONの標準機能コードはITRON2とも共通になっている。実際の機能コードの値（ITRON2とμITRONで共通）を、巻末のレファレンスに示す。

言語Cインタフェースとニモニック

μITRONやITRON2では、ニモニックの付与に関して、以下のような共通原則を設けている。

・データタイプの名称
データタイプの名称としては、構造体も含めてすべて大文字を使用する。
ポインタのデータタイプは ～P の名称とする。
構造体については、T_～ の名称とする。

・定数のニモニック
エラーコードは E_～ の名称とする。
特定のシステムコールやパラメータでのみ使用するモードなどの名称は、Txy_～ とする。xy の部分は、システムコールやパラメータに応じて変化する。

・パラメータ、リターンパラメータの名称
パラメータ、リターンパラメータの名称に関しては、次のような原則を設ける。

p_	リターンパラメータを入れる領域へのポインタ
pk_	パケットアドレス
～cd	～コード
ar_	配列へのポインタ
i～	初期値
～sz	サイズ

・パラメータ名が同じであれば、原則として同じデータタイプを持つようにする。

また、μITRONの言語Cインタフェースでは、次のような共通原則を設けている。

・関数としての戻り値は、原則としてシステムコールのエラーコード (ercd)

になる。正常終了の場合には ercd は 0 となり、エラーが発生した場合には、ercd は0以外の値（標準は負の値）になる。

・エラーコード以外のリターンパラメータ(XXX)を返す場合には、リターンパラメータを入れる領域へのポインタ(p_XXX)をパラメータとして指定する。

・リターンパラメータへのポインタは、パラメータよりも先に置くものとする。
アセンブラインタフェースでの指定がポインタ（パケットアドレス）になっているものについても、意味的にリターンパラメータに相当するものであれば、この原則が適用される（XXX_sts など）。

μITRONで使用する標準データタイプ、および言語Cインタフェースの標準仕様を巻末のレファレンスに示す。また、システムで共通に使用する定数のニモニックや標準値、および構造体のパケット形式を巻末のレファレンスに示す。

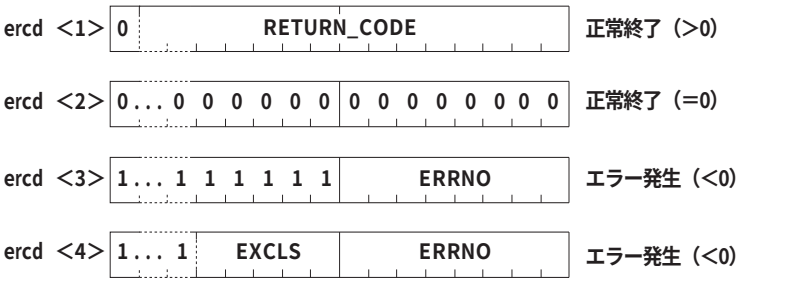
エラーコードのフォーマット

μITRONでは、各システムコールで発生するエラーに対して、それを表わすニモニックとエラーコード値の標準を設けている。また、これらの標準は、μITRONとITRON2で共通化されている。

エラーコードの標準値は負の数になっており、その中に ercls（例外クラス）と errno の情報を含んでいる。例外クラスは、例外の重要度や分類を示すものである。μITRONの場合、例外クラスの違いが直接OSの動作に影響するわけではないが、ITRON2においては、システムコール例外の起動の有無を決めるために例外クラスが利用されている。

μITRONで使用するエラーコードのフォーマットを［図2.11］に示す。

— excls (4≦excls≦31) により、発生するシステムコール例外の例外クラスを表わす。
— errno (1≦errno≦255) により、同じ例外クラスに属する複数のエラーを区別する。



＊EXCLSはexclsの1の補数、ERRNOはerrnoの2の補数である。

［図2.11］ μITRONにおけるエラーコードのフォーマット

— エラー発生の場合の ercd の値は (- (excls ≪ 8) - errno) によって表わされ、原則として ercd<4>のフォーマットを使用する。ただし、'≪' は左シフトである。
— excls が異なるのに errno が等しくなる組み合せが生じないようにする。つまり、excls の部分が無くても、errno のみで ercd の識別を可能とする。(errno には、excls と独立にシーケンシャルな番号を与える。)
— ercd<1>のフォーマットは、μITRONでは使用していない。ITRON2では、ファイル管理で ercd<1>のフォーマットを使用している。
— デバッガ等で ercd の値を直接見る場合に、errno の部分のみの値を知りたければ、ercd を1バイトの符号付きの数として扱えば良い。
— μITRONで8ビットの制約がある場合には、excls の情報を省略し、errno の部分のみを用いても良い（ercd<3>のフォーマット)。

ITRON2およびμITRONで使用する例外クラス(excls)を、巻末のレファレンスに示す。

μITRONやITRON2では、エラーコードのニモニックの付与に関して、以下のような共通原則を設けている。

— 汎用的に使用されるエラーは汎用的な名称とする。特殊な意味を持つエラーは、誤解が起きないように、特殊な意味を持つことが想像できるようにする。

— 他のオブジェクトでも発生することが予想されるが、実際は一種類のオブジェクトにのみ関連して発生するエラーがある。このような場合は、誤解を防ぐために、エラー名称にそのオブジェクトの名称を含めるようにする。
例えば、ファイルのアクセス権違反を示すエラーは、メモリのアクセス権違反と区別するために 'F'の名称を含める。

— ITRON1で使っていたエラーコードとの連続性、整合性を考慮する。

—「不正エラー（意味的に間違った操作)」と「予約機能エラー（将来の機能追加によりエラーとならなくなるもの)」の区別がはっきりしている部分は、その区別を行う。

ITRON2およびμITRONで使用するエラーコードのニモニックとその標

準値を、巻末のレファレンスに示す。

μITRONにおいて、インプリメント依存のエラーコード、およびユーザ用（拡張SVCなど）のエラーコードとしては、次の値を使用する。

インプリメント依存:	errno = 225〜255, ERRNO = B'000XXXXX ercd = B'11...111YYYYY000XXXXX
ユーザ用:	errno = 193〜223, ERRNO = B'001XXXXX ercd = B'11...111YYYYY001XXXXX

個々のエラーに対応するエラークラス (ercls) については、エラーの種類や意味に応じて、0または適当な値を入れる。


mITRON
第三部
ITRON2
ITRON

第三部 第一章

ITRON2概説

この章では、ITRON1とITRON2との関係、ITRON2の追加機能などに関して説明を行う。

ITRON2設計の実際

ここでは、ITRON2設計の際の考え方や、ITRON1とITRON2との関係、両者の具体的な相違点などについて、項目別に説明する。

・豊富な機能
豊富な機能の取捨選択による適応化、といった観点から、考えられる機能はできるだけ用意するという方針にする。これは、ITRON1と同様である。

・同期・通信機能
ITRONでは TCB を軽くするという意味で、slp_tsk～wup_tsk 以外の同期・通信機能は、原則としてタスク独立となっている。ITRON2では、メッセージバッファ（内容をコピーするメッセージ）やランデブの機能が拡張機能として追加されているが、タスク独立の原則は、ITRON2で新規に追加された機能に対しても適用される。

・例外管理
例外のクラス分け、例外ハンドラの多重起動、拡張SVCハンドラに対する例外管理などの仕様を明確化し、例外処理のプログラムについても標準化を促進する。
また、ソフトウエアによって他タスクの例外ハンドラを起動する機能（強制例外）を設ける。ITRON1の終了時処理ルーチンは、例外ハンドラの一つ（終了ハンドラ）という扱いにする。

・メモリプール管理
タスクがローカルに使用するメモリブロックと、タスク間で共用されるメモリブロックについては、non-MMU 版でもシステムコールレベルで区別するものとする。これは、MMU 対応版への移行をスムーズにするためである。

・オブジェクトのアクセス方法
ITRON2では、ITRON1のように、オブジェクト生成時に動的に得られたアクセスキーによってオブジェクトのアクセスを行なうのではなく、実

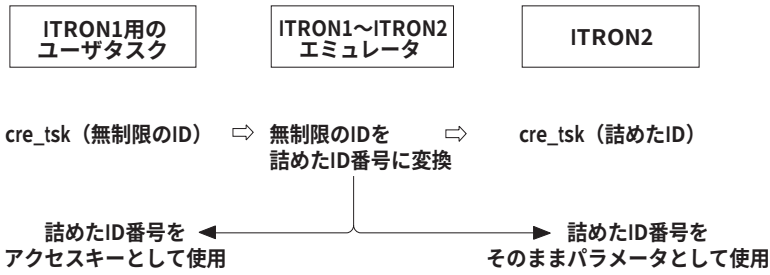
行開始前に確定している値（ID番号など）によってオブジェクトのアクセスを行なう。これは、プログラムの書きやすさと互換性に対する配慮である。

ID番号を使った仕様で十分な性能を出すためには、ID番号を内部のアクセスアドレスに変換する際に、ID番号をインデクスとする配列を用いるのが一般的である。この場合、システム構築時に、ユーザに最大、最小のID番号を指定させ、ID番号をできるだけ詰めて使ってもらうようにしても構わない。

なお、ITRON1で無制限の大きさのID番号値を使用していた場合との互換性については、以下のように考えることができる［図3.1］。μITRONでITRON1をエミュレーションする際にも、同じ考え方が適用できる。

―オブジェクト生成システムコール (cre_XXX) では、パラメータで指定された無制限の大きさのID番号を、ITRON2で使用する詰めたID番号に変換する。

―変換後の詰めたID番号は、ITRON1をエミュレーションするためのアクセスキーとして使用する。また、同じ値を、ITRON2のオブジェクト操作システムコールに対するパラメータとして使用する。



［図3.1］ID番号に関する互換性

・エラーコード

BTRONのシステムコールでは、システムコールの戻り値が負の場合にエラーを表わし、0または正の場合に正常終了を表わす。したがって、エラーコードは負の値になっている。戻り値が正の場合は、何らかの意味のある情報（ファイルオープン時のファイルディスクリプタ等）を表わす。

BTRONとの整合性強化のため、ITRON2では、エラーコードとして負の値を使用する。また、μITRONでも、ITRON2に合わせてエラーコードの値を負にするのが望ましい。この場合、正の値を使えば、システムコールの関数値としてエラー以外の情報を返すことができる。ただし、現在ITRONで正の値を返すシステムコールは、ファイル関連のシステムコールのみとなっている。

これに伴って、エラーコードのニモニックもBTRONと合わせて整理が行なわれる。

・負の数の扱い

TRONファミリ全体での方針（一種の「TRON作法」）として、負の数はシステム用の数、正の数はユーザ用の数と考えるという原則がある。また、0は特殊な意味を表わすものとしている。例えば、TRON全体として次のような例がある。

― ITRONでは、tskid = 0 で自タスクの指定になる。
― ITRON / MMUでは、srcid = (-1) により共通空間を示す。
― BTRONでは、システムコールの関数値が負になったことによりエラーを示す。
― BTRONでは、process_id = 0 により自プロセスを、process_id = (-1) により親プロセスを示す。
― TRONCHIPでは、負のアドレスがSS（共通半空間）、正のアドレスがUS（個別半空間）となっている。
― CTRONでは、tmout = (-1) でタイムアウト無しを示す。

これに合わせて、ITRON2でも負のID番号を利用できるようにし、負のID番号を持つタスクはシステムタスクとして扱う。例えば、ITRON2のカーネルと共に入出力用のタスクを提供する場合には、負のID番号（例えば -8 ～ -5）を入出力用のタスクに割り当てるのが望ましいということになる。

また、ITRON2の標準システムコール用の機能コードとしては、負の値を使用する。
左記のうち、自タスクや共通空間など特殊な意味を持つ値については、次のようなニモニックを使用する。

タスクIDに対する指定 (tskid)

```
TSK_SELF    0       /* 自タスク指定 */
TSK_CMN     (-1)    /* タスク間共通の例外ハンドラの定義 */
TSK_INDP    (-2)    /* タスク独立部の指定 */
```

タイムアウト指定 (tmout)

```
TMO_POL     0       /* ポーリング */
TMO_FEVR    (-1)    /* 永久待ち */
```

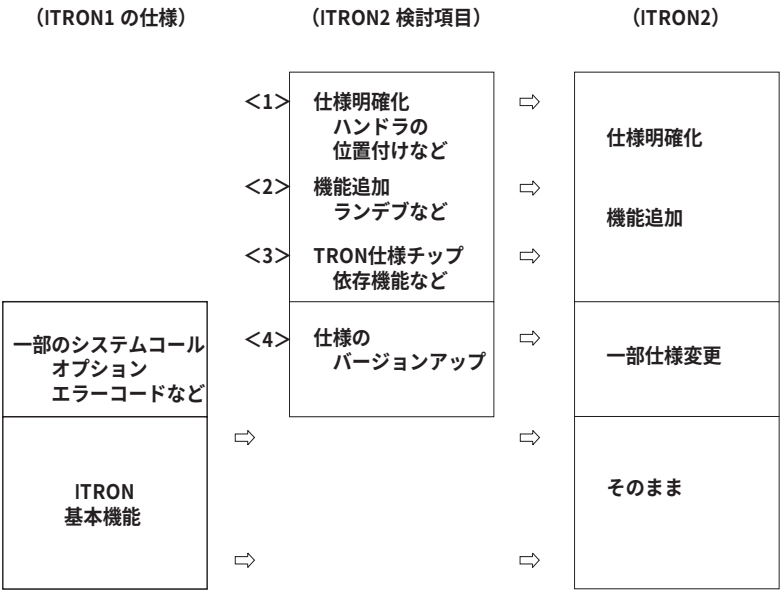
拡張SVCでの機能コード指定 (s_fncd)

```
TFN_CMN     (-1)    /* 拡張SVC共通の例外ハンドラの定義 */
```

スペースIDの指定(spcid)

```
TSP_SELF    0       /* 自タスクの属する空間 */
TSP_CMN     (-1)    /* 共通空間 */
```

・ITRON1との互換性
ITRON2は、機能の増強や整理、BTRONやCTRONとの整合性強化のため、ITRON1とは完全互換にならない部分がある。それでも、ITRON1との連続性を確保するため、互換性を保証するためのツールやライブラリを提供できるようにする。
ITRON1(ITRON Ver 1.11) とITRON2との関係を［図3.2］に示す。



＊ ITRON1 の内容だけでは、<1><2><3>の部分が含まれない。

［図3.2］ ITRON1とITRON2との関係

ITRON2の追加機能

ITRON2では、ITRON1と比べてかなりの機能が追加されている。そこで、ユーザにとっての理解しやすさやITRON1からの移行のしやすさを考えて、システムコールのレベル分けを行なっている。具体的には、基本機能≪I1≫、拡張機能≪I2≫、システム操作用機能の3段階に分けられている。

ここでは、ITRON2で追加された機能に関して説明を行う。

・ランデブ機能

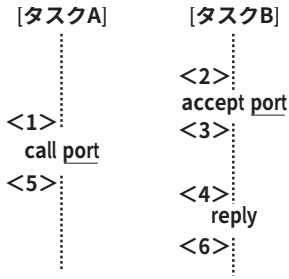
ランデブ機能は、ADA の言語仕様に導入されている同期/通信機能であり、その元になっているのは、Hoare の提唱した CSP (Communicating Sequential Processes) である。ランデブ機能は、メッセージ機能を組み合せて実現することもできる。しかし、アクノレッジを伴う通信を行なうのであれば、ランデブを利用する方が無駄が少ないため、ITRON2で機能を追加する。

ランデブでは、call, accept, reply の3つのプリミティブを利用し、次のような動作を行なう。

［図3.3］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、call port 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 は待ち状態になる。逆に、タスクB が先に＜2＞に到達し、accept port 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 は待ち状態になる。

タスクAが call port を実行し、タスクB が accept port を実行した時点でランデブが成立し、タスクAが待ち状態のままタスクBの＜3＞～＜4＞が実行される。＜4＞の reply が実行された時点で、タスクAが待ち状態から解放され、タスクAは＜5＞より、タスクB は＜6＞より、再び実行を始める。

＜3＞～＜4＞の間ではタスクAとタスクB が同時に動かないということが保証されるので、タスク間で共有するデータを操作することができる。また、インプリメントの観点から見ると、データの授受が済むまで双方



[図3.3] ランデブの動作

のタスクを待たせておくことになるため、データ（メッセージ）のキューイングやバッファリングが不要になるという利点がある。

・資源管理サポート機能

資源管理サポート機能とは、資源の獲得や返却と同時（不可分）に、指定されたメモリの内容を変更する機能である。この機能により、タスクが自分で資源管理することを容易にし、強制終了や強制待ちに対するクリティカルセクションを無くする。

資源管理サポート機能は、具体的には、次のような場合に使用する。一般に、複数の操作を不可分に行なう場合は、[図3.4] のように、セマフォ等によるクリティカルセクション (C.S.) を設ける。これ以外に、割込み禁止にする方法もあるが、割込みに対する応答性が悪くなるため、あまり望ましく無い。

ところが、このタスクが強制終了されたような場合、このタスクが C.S. 内にいたかどうかによって、終了ハンドラの処理内容が変わってくる。つまり、図の＜1＞以前や＜4＞以後にいたのであれば、この C.S. やセマフォに関して終了処理の必要はないが、＜2＞～＜3＞にいたのであれば、wai_sem で獲得したセマフォ資源を返却しておく必要が生じる。したがって、終了ハンドラでは、タスクが持っていた資源を知ることが重要になる。「資源管理サポート機能」は、このような場合に利用する機能である。

マルチユーザのOSであれば、タスク強制終了時の資源返却はOSが行なうのが普通である。しかし、ITRONの場合は、OSを軽くするためタスク終了時の自動的な資源返却を行なわない。その代わり、それをサポートする機能を提供することにより、終了ハンドラの中でユーザが容易に資源

返却のプログラムを書けるようにする、という方針である。それを実現するのが資源管理サポート機能である。
レジスタではなくメモリに情報を入れるのは、例外ハンドラ（終了ハンドラ）からも容易にその情報にアクセスするためである。
資源管理サポート機能のシステムコールの名称は、～ with memory update の意味で u～ とする。

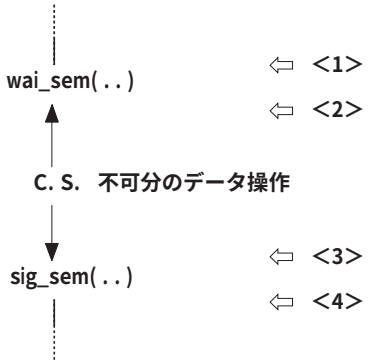
（例） uwai_sem
usig_sem

ITRON1のシステムコールのみを使って、ユーザ側で資源の割り当て状況を厳密に管理しようとした場合には、資源獲得を示すテーブルの更新命令と wai_sem や sig_sem の操作を不可分にしなければならず、この間を割込み禁止にする必要が生じていた。

・メッセージバッファ（コピー）機能
メッセージブロックをリンクでつなぐ従来のITRONの方式は、効率優先で決められた仕様であるが、以下の点で問題がある。

― ITRON/MMU（特にページングマシン）では非常に使いにくい。
ページ単位でないとメモリの共有ができない。

― 異常終了に対する対応が難しい。
タスクが異常終了したり、メイルボックスが削除されたりしたよ



［図3.4］セマフォによるクリティカルセクションの確保

うな場合に、それに合わせてメッセージブロックを解放するのが難しい。（資源管理サポート機能を使えば、不可能ではない。）

これを解決するため、現在の仕様に加えて、内容のコピーによりメッセージのやりとりをする機能（メッセージバッファ機能）を追加する。

・オブジェクトの状態を見る機能

タスク、イベントフラグ、セマフォ、メイルボックス、メモリプールなどのオブジェクトについて、状態参照システムコール (XXX_sts) を設ける。オブジェクトの状態参照は、ITRON1では、アクセスアドレスを得るシステムコール (XXX_adr) を使って管理ブロックの内部を直接見るという方針であった。しかし、互換性を高めるためには、XXX_adr よりも、オブジェクトの状態を見るシステムコールを用意する方が望ましい。そこで、XXX_adr のシステムコールは廃止し、代わりに XXX_sts のシステムコールを導入する。

・オブジェクトの属性設定機能

各オブジェクトの属性として、コプロセッサの有無や待ち行列の作り方などシステム (OS) でサポートしている属性以外に、ユーザが自由に定義できる属性を設ける。具体的には、全オブジェクトに対して属性を示すワード (XXXatr) を設け、その最上位バイトをユーザ用、第二上位バイトをインプリメンタ用、下位2バイトをシステム用（ITRON1の命令オプション機能を含む）とする。XXXatr は、XXX_sts システムコールで読み出すことができる。

例えば、タスクのユーザ定義属性であれば、大規模なOSには入っているがITRONには入っていない機能を、ユーザがシミュレートするような場合に利用できる。具体的には、タスクに親子関係を導入したり、タスクをグループ化したりするために利用することが考えられる。

・周期起動ハンドラ、アラームハンドラ（指定時刻起動ハンドラ）の機能

ITRON1では、周期起床 (cyc_wup) の機能により、時間に関係する処理を行なうようになっていた。ITRON2では、それに加えて、周期起動ハンドラ、指定時刻起動ハンドラ（アラームハンドラ）の機能を設ける。前者は周期的に起動されるタスク独立のハンドラであり、後者は指定時刻に起動されるタスク独立のハンドラである。これらの機能は、ITRON1の周期起床の機能と比較すると、よりプリミティブで自由度の高いメカニズムとなる。

以上のような機能追加により、ITRON2のOS全体の仕様は、かなり大きなものになる。しかし、追加した仕様の多くは≪I2≫仕様（拡張仕様）になるので、≪I1≫仕様（基本仕様）の部分だけを見れば、ITRON1と同程度の規模である。

ITRONの場合は、適応化によってターゲットシステム上の最終的なオブジェクトを小さくすることが可能なので、適応化前の仕様としては、できるだけ多くのものを包含するという考え方になっている。従来のITRONもそのような方針に基づいて設計されたものであるが、ITRON2では、その考え方をさらに推し進めたものになっている。

第三部 第二章

ITRON2基本機能

この章では、ITRON2でサポートを必須としている機能（≪I1≫レベルのシステムコール）に関する説明を行う。このレベルの機能は、ほぼITRON1でサポートしていた機能に相当するものである。

タスク管理機能

タスクを生成する

cre_tsk

cre_tsk: Create Task

【パラメータ】

tskid	TaskIdentifier	タスクID
tskatr	TaskAttribute	タスク属性
stadr	TaskStartAddress	タスクスタートアドレス
itskpri	InitialTaskPriority	初期優先度
stksz	UserStackSize	ユーザスタックのサイズ

【リターンパラメータ】

なし

【解説】

cre_tsk では、tskid で指定された ID 番号を持つタスクを生成する。次に、生成されたタスクに対してTCBを割り付け、パラメータで与えた情報 (itskpri, stadr, stksz) をもとにその初期設定を行う。

itskpriは2バイトで表現されるが、その取りうる値は (-16)〜255 に制限されている。一般に、負の優先度はシステム用として使用される。対象タスクは生成後、DORMANT 状態となる。

ID 番号が (-4)〜0 のタスクは生成できない。また、負のID番号のタスクは、システムタスクである。

tskatr のサイズは標準で4バイトである。tskatr のフォーマットを［図3.5］に示す。このうち、最上位バイト（bit0〜bit7 あるいは 2^31〜2^24 のビット）はユーザ属性を表わし、第二上位バイト（bit8〜bit15 あるいは2^23〜2^16 のビット）はインプリメント依存の属性を表わし、下位ハーフワード（bit16〜bit31 あるいは 2^15〜2^0 のビット）はシステム属性を表わす。ユーザは、タスクに関する情報を入れておくために、tskatr の最上位バイトを自由に使用することができる。

第二上位バイトのインプリメント依存属性の機能は、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、TRONCHIPのデバッグモード DB の値を設定したりするために利用できる。

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_COP0] ¦ [TA_COP1] ¦ [TA_COP2] ¦ [TA_COP3]
        ¦ [TA_COP4] ¦ [TA_COP5] ¦ [TA_COP6] ¦ [TA_COP7]
```

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_COPn	対象タスクが第n番目のコプロセッサを使用する（浮動小数点演算用のコプロセッサを含む）

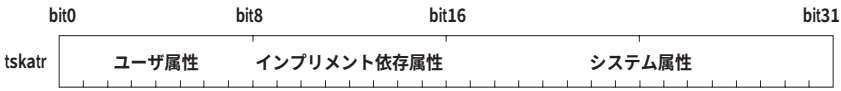
TA_HLNG の指定を行った場合には、タスク起動時に直接 stadr のアドレスにジャンプするのではなく、高級言語の環境設定プログラム（高級言語対応ルーチン）を通してから stadr のアドレスにジャンプする。

tskatr のシステム属性の部分のうち、この他の reserved の部分は、将来マルチプロセッサ属性の指定などを行なうために利用できる。

cre_tsk で指定した tskatr は、tsk_sts により読み出すことができる。

タスクが DORMANT 状態の時は、すべての状態（wupcnt, suscnt を含む）がリセットされるというのが原則である。ただし、DORMANT 状態でも例外的にリセットされないものとしては、例外マスク状態、例外ハンドラの定義状態などがある。これらの状態は、タスクの環境設定といった意味合いのものなので、DORMANT 状態の他タスクに対しても指定可能としている。

cre_tsk で生成されたタスクの例外マスク（emsptn）のデフォルト値は、



［図3.5］tskatrのフォーマット

~(ECM_CEX ¦ ECM_TER) となっている。すなわち、ter_tek, ras_ter による終了要求とCPU例外のみが受け付けられ、その他の例外はマスクされた状態となっている。タスク起動時に終了要求をマスクしておきたい場合は、cre_tsk の直後に set_ems(setptn=ECM_TER) を実行しておけば良い。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足(管理ブロック用の領域が確保できない) ※ このエラーが発生するかどうかはインプリメント依存である。
E_NOMEM	メモリ不足 (ユーザスタック用の領域が確保できない) ※ E_NOSMEMとE_NOMEMとの厳密な区別はインプリメント依存である。
E_RSID	予約ID番号 (-4≦tskid≦0)
E_RSATR	予約属性 (tskatrの下位3バイトが不正)
E_PAR	一般的なパラメータエラー (stkszが不正)
E_ILADR	不正アドレス (stadrが奇数、あるいは使用できない値)
E_IDOVR	ID範囲外 (tskidがシステムで利用できる範囲を越えた)
E_TPRI	不正タスク優先度 (itskpriが不正)
E_EXS	オブジェクトが既に存在している (同一ID番号のタスクが存在)
E_OACV	オブジェクトアクセス権違反 (拡張SVC以外のユーザタスクからの発行でtskid＜(-4))

タスクを起動する

sta_tsk

sta_tsk:　Start Task

【パラメータ】

tskid	TaskIdentifier	タスクID
stacd	TaskStartCode	タスク起動コード

【リターンパラメータ】

なし

【解説】

tskid で示されたタスクを起動する。つまり、DORMANT 状態からREADY状態へと移す。

stacd により、タスクのスタート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。

スタート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初期優先度となる。

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なわない。したがって、対象タスクが DORMANT 状態にない場合に発せられた起動要求に対しては、発行タスクにエラー E_NODMT が戻る。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦tskid≦0）
E_IDOVR	ID範囲外（インプリメント依存）
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_NODMT	タスクがDORMANTでない（tskidのタスクがDORMANTでない）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行で tskid＜(-4)）

タスクを削除する

del_tsk

del_tsk: Delete Task

【パラメータ】

tskid	TaskIdentifier	タスクID

【リターンパラメータ】

なし

【解説】

tskid で示されたタスクを削除する。つまり、DORMANT 状態から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、スタック領域を解放する。 DORMANT 状態にでないタスクの削除はエラー E_NODMT となる。その後はこのタスクのID番号と同じID番号で、新しいタスクを生成することが許可される。

この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、自タスクが DORMANT状態ではないため、E_NODMT 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、exd_tskシステムコールを発行する。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号 (-4≦tskid≦0)
E_IDOVR	ID範囲外 (インプリメント依存)
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_NODMT	タスクがDORMANTでない (tskidのタスクがDORMANTでない)
E_OACV	オブジェクトアクセス権違反 (拡張SVC以外のユーザタスクからの発行で tskid＜(-4))

自タスクを正常終了する

ext_tsk

ext_tsk: Exit Task

【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

【解説】

自タスクを正常終了させ、DORMANT 状態へと移行させる。正常終了であるので、終了ハンドラは実行されない。本システムコールは、終了ハンドラの最後でも発行されることがある。

ext_tsk では、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。

ret_XXX および ext_tsk, exd_tsk, abo_tsk システムコールでは、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そのチェックを行なっていなければ、プログラムが暴走する。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でも、決してシステムコール発行元へは戻らないものとする。万一エラーを検出した場合には、メッセージバッファへのロギングやコンソールへのエラーメッセージの表示を行うのが望ましいが、詳細な動作はインプリメント依存となる。

このシステムコールを準タスク部（拡張SVCハンドラ）から発行した場合は、拡張SVCを呼んだタスクに対する終了ハンドラが起動されないまま、タスクが終了してしまう。したがって、拡張SVCハンドラからこのシステムコールを発行するのは望ましくなく、E_CTX のエラーとするべきである。しかし、このシステムコールは元のコンテキストに戻らないシステムコールなので、エラーチェックを行ったとしても、エラーコードを返すことができない。そこで、拡張SVCハンドラの中からこのシステムコールを発行した場合の動作は、インプリメント依存となっている。なお、拡張SVCハ

ンドラの中でタスクを終了する必要が生じた場合、通常は abo_tsk を使用する。abo_tsk を発行すれば、タスクを終了する前に、拡張SVCハンドラに対する終了ハンドラやタスクに対する終了ハンドラを起動することができる。

タスクが終了して DORMANT 状態になるときには、emsptn がタスク生成直後と同じ値、すなわち

emsptn = ~(ECM_CEX | ECM_TER | ECM_ABO | ECM_SUS)

にリセットされる。これは、タスク終了直前の emsptn の値とは無関係である。同様に、タスクに対する例外ハンドラも、タスクが終了して DORMANT 状態になるときにはリセットされる（未定義の状態になる）。ただし、これはタスクが他の状態から DORMANT 状態に変わる時（すなわちタスクの生成時またはタスクの終了時）にemsptn と例外ハンドラが一旦リセットされるという意味であり、既に DORMANT 状態になっている他タスクに対して、emsptn の変更や例外ハンドラの定義を行うことは可能である。タスクの起動前に emsptn の設定や例外ハンドラの定義を行っておく必要があれば、必要に応じて、sta_tsk の前に clr_ems や def_Xex を実行すればよい。

これに対して、tskpri（タスク優先度）、wupcnt, suscnt などのタスク属性は、他の状態から DORMANT 状態に変わる時にリセットされるだけではなく、タスクが DORMANT 状態の間は一切変更できない。これらの属性を変更しようとするシステムコールは、E_DMT のエラーになる。

【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には、メッセージバッファへのロギングや、コンソールへのエラーメッセージの表示などを行うのが望ましいが、詳細な動作はインプリメント依存となる。

E_CTX	コンテキストエラー（拡張SVCハンドラ、タスク独立部あるいはディスパッチ遅延中のタスクから発行）

自タスクを正常終了後、削除する exd_tsk

exd_tsk: Exit and Delete Task

【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

【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、システムに存在しない状態へと移行させる。正常終了であるので、終了ハンドラは実行されない。本システムコールは、終了ハンドラの最後でも発行されることがある。

なお、exd_tsk で、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。

このシステムコールを準タスク部（拡張SVCハンドラ）から発行した場合は、拡張SVCを呼んだタスクに対する終了ハンドラが起動されないまま、タスクが終了してしまう。したがって、拡張SVCハンドラからこのシステムコールを発行するのは望ましくなく、E_CTX のエラーとするべきである。しかし、このシステムコールは元のコンテキストに戻らないシステムコールなので、エラーチェックを行ったとしても、エラーコードを返すことができない。そこで、拡張SVCハンドラの中からこのシステムコールを発行した場合の動作は、インプリメント依存となっている。なお、拡張SVCハンドラの中でタスクを終了する必要が生じた場合、通常は abo_tsk を使用する。abo_tsk を発行すれば、タスクを終了する前に、拡張SVCハンドラに対する終了ハンドラやタスクに対する終了ハンドラを起動することができる。

【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には、メッセージバッファへのロギングや、コンソールへのエラーメッセージの表示などを行うのが望ましいが、詳細な動作はインプリメント依存となる。

E_CTX	コンテキストエラー（拡張SVCハンドラ、タスク独立部あるいはディスパッチ遅延中のタスクから発行）

自タスクを異常終了させる

abo_tsk

abo_tsk: Abort Task

【パラメータ】

exccd	ExceptionCode	例外コード

【リターンパラメータ】

なし

【解説】

自タスクに対して終了要求を出し、自タスクを異常終了させる。

本システムコールの発行タスクや拡張SVCハンドラに終了ハンドラが定義されていた場合には、タスクを直ちに DORMANT 状態に移行させるのではなく、制御を終了ハンドラに移すだけである。この時、終了ハンドラへのパラメータを exccd として与えることができる。

一般に、拡張SVCハンドラの中から拡張SVCハンドラを呼んだタスクを異常終了したい場合には、このシステムコールを使用する。拡張SVCハンドラの中から abo_tsk を発行した場合には、まず拡張SVCハンドラに対する終了ハンドラが起動され、タスクに戻ってからタスクに対する終了ハンドラが起動される。この時の動作は、例外マスク関連の条件を除き、拡張SVCハンドラ実行中のタスクを対象として ter_tsk が発行された場合と同じである。これに対して、拡張SVCハンドラの中から ext_tsk, exd_tsk を発行すると、タスクに対する終了ハンドラが実行されないままタスクを終了してしまう。タスクの終了処理を含むような拡張SVCハンドラを書いた場合は、タスクに対する終了ハンドラを実行してからタスクを終了するために、ext_tsk や exd_tsk ではなく、abo_tsk を発行しなければならない。

終了ハンドラが定義されていない場合には、本システムコールにより直ちにタスクや拡張SVCハンドラを終了する。その動作は ext_tsk, ret_svc と同じである。拡張SVCハンドラの実行中に abo_tsk が発行されたが拡張SVCに対する終了ハンドラが未定義だった場合、拡張SVCを呼んだ側に返すエラーコードは、abo_tsk で指定した exccd ではなく、E_TER になる。すなわち、

このケースでは、abo_tsk によって ret_svc (s_ercd=E_TER) 相当の処理を行うことになる。

abo_tsk では、終了ハンドラのマスクの状態（ECM_TER がクリアされているか、あるいはクリア可能かどうかといった状態）に関係なく、即座に終了ハンドラが起動されるものとする。すなわち、abo_tsk を使えば、ECM_TER がセットされていても、終了ハンドラが起動される場合がある。ECM_TER は、他のタスクからの終了要求 (ter_tsk, ras_ter) に対するマスクであり、自タスクからの終了要求（abo_tsk, ハンドラ未定義の例外）に対するマスクを意味するものではない。また、例外ハンドラ（CPU例外ハンドラ、システムコール例外ハンドラ、強制例外ハンドラ）が未定義でマスクされておらず、これらの例外が終了要求に置き換わった場合にも、abo_tskと同様の動作をする。すなわち、終了ハンドラのマスク状態に関係なく、即座に終了ハンドラが起動される。

終了ハンドラの起動をマスクした状態で abo_tsk を実行し、終了ハンドラが起動された場合（ハンドラ未定義の例外が終了要求に置き換わった場合を含む）、終了ハンドラ起動前の環境の emsptn の ECM_TER はセットされたままであると考える。すなわち、abo_tsk によって終了ハンドラを起動する際には ECM_TER の状態を無視するが、この時 ECM_TER がクリアされるわけではない。したがって、起動された終了ハンドラで hdr_sts を実行し、終了ハンドラを呼び出した環境の emsptn（すなわち ar_hdrs [1].emsptn & ECM_TER）をチェックすることにより、終了マスク中に起動された終了ハンドラかどうかを調べることができる。終了マスク中に起動された終了ハンドラでは、クリティカルセクションに対する処置が変ってくる可能性があるが、このような形で必要な情報を得ることができる。

終了マスク中に abo_tsk で即座に起動されるのは、現在の環境（タスク、拡張SVCハンドラ）に対する終了ハンドラのみである。たとえば、タスク [A] が終了マスク中に呼んだ拡張SVCハンドラ [B] の中で abo_tsk が発行された場合、拡張SVCハンドラ [B] の終了マスクの状態にかかわらず、[B] に対する終了ハンドラは即座に起動される。しかし、[A] に対する終了ハンドラは、[B] に対する終了ハンドラから戻った後で即座に起動されるわけではない。[A] に対する終了ハンドラは、拡張SVCハンドラ [B] からタスク [A] の実行に戻り、終了マスクがクリアされた後で起動される。

abo_tsk, ter_tsk, ras_ter による複数の終了要求のキューイングは行わない。

すなわち、これらのシステムコールが複数回発行されても、終了ハンドラは1回しか起動されない。ただし、同一タスクに対して abo_tsk,ter_tsk, ras_ter の発行が複数回行なわれた場合は、各システムコールで指定された exccd の論理和がとられていき、pk_exc の exccd ではこの論理和の値が返る。したがって、exccd をビット対応で使うことにより、疑似的に複数の要求を区別することができる。たとえば、例外コードとしての情報が1ビットで良ければ、32種類の独立した要求を区別することができる。起動された終了ハンドラでは、exccd を見て複数の起動要求があるかどうかを判断し、内部の処理を振り分ければ良い。

なお、デフォルト例外ハンドラからの終了要求による例外コードや、将来システムで定義される予定の例外コードでは、例外コードの上位ビット（上位1〜2バイト）を使用する予定である。したがって、ユーザが abo_tskで利用する例外コードとしては、下位ビット（下位2バイト）を使用するのが望ましい。

終了ハンドラ実行中（終了ハンドラ実行中に起動された例外ハンドラを含む）に abo_tsk を実行した場合（終了ハンドラ実行中に例外が発生し、そのハンドラが未定義だった場合を含む）には、終了ハンドラを起動するのではなく、即座にタスクや拡張SVCハンドラを終了するものとする。すなわち、ext_tsk あるいは ret_svc (s_ercd=E_TER) 相当の動作をする。これは、終了ハンドラが再帰的に起動されるのを防ぐためである。

上記の仕様を明確に表現するため、emsptn の中に ECM_ABO というビットを設け、このビットによってabo_tsk の動作を選択するように考える。ECM_ABO は終了ハンドラ実行中の間セットされており、この場合に abo_tsk が実行されると、拡張SVCハンドラあるいはタスクの終了を行う。一方、終了ハンドラ実行中でない場合は ECM_ABO がクリアされており、この時 abo_tsk が実行されると、終了マスクの状態にかかわらず終了ハンドラの起動を行う。ECM_ABO は、abo_tsk の動作に影響するビットであると同時に、終了ハンドラ実行中かどうかを示すビットでもある。ECM_ABO のビット配置は次のようになる。

ECM_ABO H'00000002

ECM_ABO は、タスク起動時や拡張SVCハンドラ起動時にはクリアされている。終了ハンドラ起動時には、ECM_ABO が自動的にセットされる。一方、例外ハンドラ起動時には ECM_ABO の値は変わらない。たとえば、

終了ハンドラから呼ばれた例外ハンドラでは、ECM_ABO はセットされたままである。すなわち、終了ハンドラから呼ばれた例外ハンドラの中でabo_tsk が実行されても、再度終了ハンドラが起動されるわけではなく、そのタスクあるいは拡張SVCハンドラを終了する。なお、clr_ems, set_ems では、ECM_ABO のビットを操作（変更）できないものとする。clr_ems, set_ems により ECM_ABO のビットを操作しようとした場合には、E_PAR のエラーになる。

終了要求が出された場合、その対象タスク（abo_tskの場合は自タスク）は、ECM_TER のマスク状態や拡張SVCハンドラ実行中かどうかにかかわらず、直ちに最高のタスク優先度になる。すなわち、abo_tsk の処理の中にchg_pri の処理を含むと考えることができる。また、終了マスクのため、拡張SVCに対する終了ハンドラとタスクに対する終了ハンドラが連続して実行されない場合でも、その間で実行されるタスク部（ECM_TER をマスクした状態）の優先度は、最高の状態のままになっている。これは、終了要求の緊急性を考慮したためである。

【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には、メッセージバッファへのロギングや、コンソールへのエラーメッセージの表示などを行うのが望ましいが、詳細な動作はインプリメント依存となる。

E_CTX	コンテキストエラー（タスク独立部あるいはディスパッチ遅延中のタスクから発行）

他タスクを強制的に異常終了させる
ter_tsk

他タスクに対して終了要求を出す
ras_ter

ter_tsk:　Terminate Task
ras_ter:　Raise Terminate

【パラメータ】

tskid	TaskIdentifier	タスクID
exccd	ExceptionCode	例外コード

【リターンパラメータ】

なし

【解説】

tskidで示されたタスクに対して終了処理要求を出し、対象タスクを強制的に異常終了させる。

ter_tsk, ras_ter の対象タスクが終了ハンドラを定義していた場合には、対象タスクにそれを実行させる。すなわち、対象タスクが直ちに DORMANT 状態に移行するのではなく、対象タスクの制御を終了ハンドラに移す。この時、終了ハンドラへのパラメータを exccd として与えることができる。 exccd は、対象タスクのスタックを通じて、対象タスクから参照することができる。

ter_tsk, ras_ter の対象タスクが終了処理要求をマスクしていた場合は、マスクが解除されるまで終了ハンドラの起動が遅延させられる。この機能は、対象タスクがクリティカルセクション中で強制終了させられるのを防ぐためのものである。

ter_tsk の場合は、対象タスクが待ち状態に入り、何らかの待ち行列につながれていても、その待ち状態を強制解除する。これは、終了ハンドラのマスク状態には関係しない。終了ハンドラの起動をマスクしたまま待ち状態に入っていたタスクに対して ter_tsk が発行されると、待ち状態は解除されて E_RLWAI のエラーが返るが、終了ハンドラは起動されない。

また、ter_tsk では、対象タスクが強制待ち状態（SUSPEND状態）であっても、強制的に実行を再開させる。すなわち、ter_tsk の処理は、rel_wai の処理と frsm_tsk の処理を含んでいる。ter_tsk では SUSPEND状態が解除されるため、当然 suscnt もクリアされて0になる。

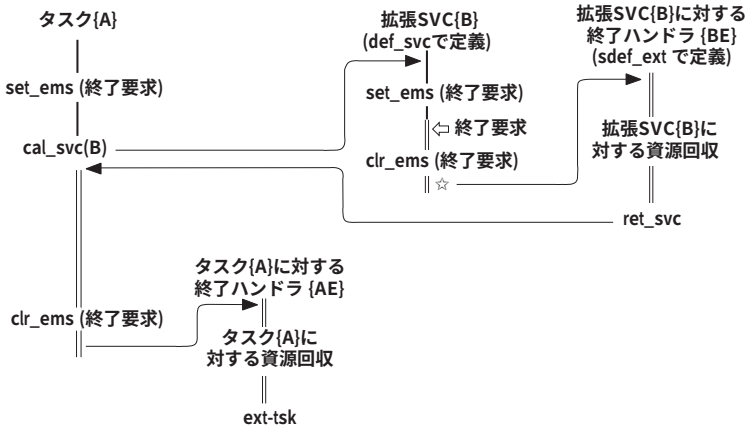
一方、ras_ter では、対象タスクの待ち状態や強制待ち状態を解除することはない。待ち状態のタスクに対して ras_ter が発行された場合、終了ハンドラが起動されるのは他の要因によって待ち状態が解除された後になる。

ter_tsk で対象タスクの終了ハンドラが定義されておらず、かつ終了ハンドラの起動がマスクされていない場合には、本システムコールにより対象タスクが直ちに終了する。この場合、ext_tsk 等と同様に、対象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）は自動的に解放されないので、注意が必要である。終了ハンドラの起動がマスクされ、かつ終了ハンドラが定義されていない場合には、マスクが解除されるまでタスクの終了が遅らされる。対象タスクが待ち状態や強制待ち状態であった場合でも、ter_tsk によりそれらの待ち状態が解除されるため、上記と同じ動作をする。ras_ter の動作も ter_tsk とほぼ同様であるが、対象タスクが待ち状態だった場合には、他の要因により待ち状態が解除された後でタスクが終了する。

対象タスクが拡張SVC実行中であった場合には、タスクに対する終了ハンドラを起動する代りに拡張SVCに対する終了ハンドラが起動される。拡張SVCに対する終了ハンドラが定義されていない場合は、タスクを終了する代りに、即座に拡張SVCハンドラからリターンする。拡張SVCハンドラ実行中も、終了ハンドラの起動や拡張SVCハンドラの終了（拡張SVCハンドラからのリターン）をマスクすることが可能であるが、例外マスクの値はタスク実行中とは異なったものが使用される。(例外管理の説明を参照)。

ter_tsk, ras_ter 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は E_SELF のエラーとなる。

終了要求が出された場合、その対象タスクは、ECM_TER のマスク状態や拡張SVCハンドラ実行中かどうかにかかわらず、直ちに最高のタスク優先度になる。すなわち、ter_tsk, ras_ter の処理の中に chg_pri の処理を含むと考えることができる。また、終了マスクのため、拡張SVCに対する終了ハンドラとタスクに対する終了ハンドラが連続して実行されない場合でも、その間で実行されるタスク部（ECM_TER をマスクした状態）の優先度は、最高の状態のままになっている。これは、終了要求の緊急性を考慮したためである。



＊ '‖' は最高の優先度で実行される区間を示す。

［図3.6］終了ハンドラとタスク優先度の変化

拡張SVCハンドラ実行中に終了要求を受け付け、拡張SVCハンドラに対する終了ハンドラが起動された場合のタスク優先度の変化を［図3.6］に示す。

なお、終了要求の受け付けにより最高の優先度になった後も、優先度の変更は可能である。終了要求によるタスク優先度の変更は終了要求のシステムコール発行時に行われるだけであり、その後のタスク優先度の変更は自由である。例えば、［図3.6］の{BE}の中でタスク優先度が変更された場合、変更後の優先度は、タスク{A}のcal_svc (B)の後の部分や、{AE}の中でも有効である。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦tskid≦-1、タスク独立部の発行でtskid=0）
E_IDOVR	ID範囲外（インプリメント依存）
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_DMT	タスクがDORMANTである（tskidのタスクがDORMANT）
E_SELF	自タスク、自プロセスの指定（tskidが自タスク、タスク部や準タスク部の発行でtskid=0）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でtskid＜(-4)）

タスク優先度を変更する

chg_pri

chg_pri: Change Task Priority

【パラメータ】

tskid	TaskIdentifier	タスクID
tskpri	TaskPriority	優先度

【リターンパラメータ】

なし

【解説】

chg_pri では、tskid で示されたタスクの現在の優先度を、tskpri で示される値に変更する。tskid = TSK_SELF によって自タスクの指定になる。

タスクの優先度は、数の小さい方が高い優先度となる。対象タスクが何らかの待ち行列につながれていた場合には、このシステムコールにより待ち行列の順番が変わることがある。

tskpri = TPRI_INI (0) の指定により、タスク生成時に指定された優先度（タスクの固有優先度）に戻るものとする。

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（あるいは終了ハンドラが起動されるまで）有効である。タスクが DORMANT 状態になると、終了前のタスク優先度は捨てられ、次にタスクが起動された時のタスク優先度は、タスク生成時に指定された優先度 itskpri になる。

【エラーコード (ercd)】

E_OK	正常終了
E_TPRI	不正タスク優先度（tskpriが不正）
E_RSID	予約ID番号（-4≦tskid≦-1、タスク独立部の発行でtskid=0）
E_IDOVR	ID範囲外（インプリメント依存）
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_DMT	タスクがDORMANTである（tskidのタスクがDORMANT）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でtskid＜(-4)）

タスクのレディキューを回転する rot_rdq

rot_rdq: Rotate Ready Queue

【パラメータ】

tskpri TaskPriority 優先度

【リターンパラメータ】

なし

【解説】

tskpri で示される優先度のレディキューを回転する。すなわち、その優先度のレディキューの先頭につながれているタスクをレディキューの最後尾につなぎかえ、同一優先度のタスクの実行を切り換える。このシステムコールを一定時間間隔で発行することにより、ラウンドロビン・スケジューリングを行なうことが可能となる。

tskpri = TPRI_RUN(0) により、その時実行状態にあるタスクを含むレディキュー（最高優先度のレディキュー）を回転させるものとする。一般のタスクから発行される rot_rdq では、これは自タスクの持つ優先度のレディキューを回転するのと同じ意味になるが、このような仕様にしておけば、周期起動ハンドラなどのタスク独立部から rot_rdq (tskpri=TPRI_RUN) を発行することも可能である。

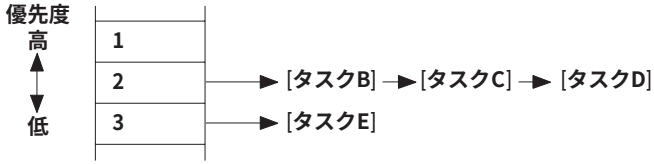
rot_rdq の tskpri として TPRI_RUN または自タスクの優先度を指定した場合には、自タスクがそのレディキューの後ろにまわることになる。つまり、自ら実行権を放棄するために、rot_rdq を発行することができる。（ここで言う「レディキュー」は、実行状態のタスクも含んだものと考えている。）

指定した優先度のレディキューにタスクがない場合は何もしないが、エラーとはならない。

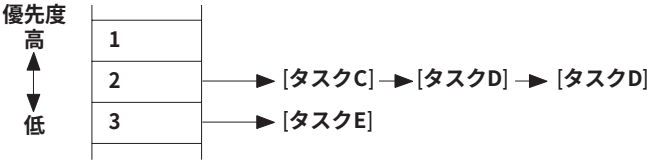
rot_rdq の実行例を［図3.7 (a)］,［図3.7 (b)］に示す。［図3.7 (a)］の状態で、tskpri = 2をパラメータとしてこのシステムコールが呼ばれると、新しいレディキューの状態は［図3.7 (b)］のようになり、次に実行されるのはタスクCとなる。

【エラーコード (ercd)】

E_OK	正常終了
E_TPRI	不正タスク優先度（tskpriが不正）



[図3.7 (a)] rot_rdq 実行前のレディキューの状態



＊次に実行されるのはタスクCである。

[図3.7 (b)] rot_rdq (tskpri = 2) 実行後のレディキューの状態

タスクの待ち状態を強制解除する rel_wai

rel_wai: Release Wait

【パラメータ】

tskid TaskIdentifier タスクID

【リターンパラメータ】

なし

【解説】

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

本システムコール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。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。

本システムコールでは、待ち状態解除要求のキューイングは行わない。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_NOWAI 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_NOWAI のエラーとなる。

本システムコールでは、SUSPEND状態の解除は行わない。二重待ち状態(WAIT-SUSPEND) のタスクを対象としてこのシステムコールを発行すると、対象タスクはSUSPEND状態となる。SUSPEND状態も解除する必要がある場合には、別に frsm_tsk を発行する必要がある。

rel_wai と wup_tsk との違いをまとめると、次のようになる。

— wup_tsk は slp_tsk, wai_tsk による待ち状態のみを解除するが、rel_wai ではそれ以外の要因（wai_flg, wai_sem, rcv_msg, 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、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、rel_wai による待ち状態の

解除はエラー(E_RLWAI) である。

― wup_tsk の場合は、対象タスクがまだ slp_tsk, wai_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、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エラーとなる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦tskid≦0）
E_IDOVR	ID範囲外（インプリメント依存）
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_DMT	タスクがDORMANTである（tskidのタスクがDORMANT）
E_NOWAI	タスクが待ち状態でない（tskidが自タスクの場合を含む）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でtskid＜(-4)）

自タスクのIDを得る

get_tid

get_tid:　Get Task Identifier

【パラメータ】

なし

【リターンパラメータ】

tskid	TaskIdentifier	タスクID

【解説】

自タスクのID番号を得る。タスク独立部から発行された場合は、タスク独立部のため tskid が得られなかったという意味で FALSE = 0が返る。

【エラーコード (ercd)】

E_OK	正常終了

タスクの状態を見る

tsk_sts

tsk_sts: Get Task Status

【パラメータ】

tskid	TaskIdentifier	タスクID
pk_tsks	Packet of TaskStatus	タスク状態を返すパケットの先頭アドレス

【リターンパラメータ】

なし

【解説】

tsk_sts では、tskid で示されたタスクの状態を読み出し、その結果を pk_tsks 以下の領域に返す。

pk_tsks に返される情報としては、次のようなものがある。

```
tskatr      /* タスク属性 */
spcid       /*スペースID、MMU未サポートや共有空間の
              場合は TSP_CMN */
tskstat     /* タスク状態 */
epndptn     /* ペンディング中の終了要求と強制例外 */
tskwait     /* 待ち要因 */
wid         /* 待ちオブジェクトID */
suscnt      /* SUSPEND要求カウント、0≦suscnt＜2^31 */
wupcnt      /* 起床要求カウント、0≦wupcnt＜2^31 */
stadr       /* タスクスタートアドレス */
isp         /* 初期SP値 */
itskpri     /* 初期優先度 */
tskpri      /* 現在優先度 */
```

このうち、tskstat は次のような値をとる。

tskstat:

TTS_RUN	RUN状態
TTS_RDY	READY状態
TTS_WAI	WAIT状態
TTS_SUS	SUSPEND状態
TTS_WAS	WAIT-SUSPEND状態
TTS_DMT	DORMANT状態
TTS_DBG	デバッグ待ち状態

上記のうち、TTS_DBG は他のすべての状態と複合することができる。また、TTS_SUS と TTS_WAI を複合した状態が TTS_WAS である。tskstat が TTS_SUS (TTS_WAS を含む) の場合は、必ず suscnt＞0 となる。

tskstat が TTS_WAI (TTS_WAS を含む) の場合、tskwait, wid は［表3.1］のような値をとる。

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。

なお、tskstat, tskwait として上記のような値が定義されているが、これは、同じ値を TCB に入れなければならないという意味ではない。TCB 内での表現方法はインプリメント依存であり、tsk_sts 実行時に、TTS_RUN, TTS_WAI 等の標準値に変換すれば良いわけである。

tskid = TSK_SELF によって自タスクの指定になるが、このシステムコールでは自タスクのIDはわからない。自タスクのIDを知りたい場合は、get_tid を利用する。

tsk_sts で、対象タスクが存在しないタスクであれば、エラー E_NOEXS とする。

タスクが DORMANT 状態の時は、すべての状態 (wupcnt, suscnt を含む) がリセットされるというのが原則である。すなわち、DORMANT 状態のタスクでは wupcnt=0, suscnt=0 であり、また、tskstat = (TTS_DMT ¦ TTS_SUS) といった複合状態は無い。

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 tsk_sts を実行した場合、tskstat として RUN状態 (TTS_RUN) を返すものとする。これは、割込みハンドラが、タスクとは別のレベルで動くものと考えられるためである。また、もしこの場合に実行可能状態を返すと、この時点で実行状態

のタスクが一つも無くなるため、変則的な状況になる。(強いて言えば割込みハンドラが「実行状態」ということであるが、タスク独立部はタスク状態を持たないので、これは不合理である。)

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号 (-4≦tskid≦-1、タスク独立部の発行でtskid=0)
E_ILADR	不正アドレス (pk_tsksが使用できない値)
E_IDOVR	ID範囲外 (インプリメント依存)
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_OACV	オブジェクトアクセス権違反 (拡張SVC以外のユーザタスクからの発行でtskid＜(-4))

tskwait	意味	wid
TTW_SLP	slp_tsk, wai_tsk による待ち	0
TTW_DLY	dly_tsk による待ち	0
TTW_FLG	wai_flg による待ち	待ち対象のflgid
TTW_SEM	wai_sem, uwai_sem による待ち	待ち対象のsemid
TTW_MBX	rcv_msg, urcv_msg による待ち	待ち対象のmbxid
TTW_MPL	get_blk, uget_blk による待ち	待ち対象のmplid
TTW_MBF	rcv_mbf による待ち	待ち対象のmblid
TTW_CAL	ランデブ呼出待ち	待ち対象のporid
TTW_ACP	ランデブ受付待ち	待ち対象のporid
TTW_RPL	ランデブ終了待ち	待ち対象のporid
TTW_LMP	get_lbl, uget_lbl による待ち	待ち対象のlmpid

[表3.1] tskwaitとwidの値

ハンドラ実行環境を参照する
hdr_sts

hdr_sts: Get Handler Status

【パラメータ】

tskid	TaskIdentifier	タスクID
ar_hdrs	Array of Handlers	ハンドラ実行環境の情報を返す配列へのポインタ
hdstart	StartLevel of Handlers	実行環境の参照を開始するハンドラのネスト段数
hdcnt	Count of Hanldlers	実行環境の参照を行うハンドラの段数

【リターンパラメータ】

なし

【解説】

tskidで示されたタスクがハンドラ（例外ハンドラ、拡張SVCハンドラ）を実行中かどうかを調べ、その情報（ハンドラ実行環境）を ar_hdrs 以下の領域に返す。tskid = TSK_SELF により自タスクの指定になる。ハンドラがネストしていた場合には、ネストしたハンドラの各段に関するハンドラ実行環境の情報を、ar_hdrs 以下の領域に配列として返す。

ar_hdrs は、以下のような情報を含む構造体 (T_HDRS) の配列である。

```
emsptn      /* 例外マスク */
pk_exc      /* 例外情報を示すパケットへのポインタ */
pk_regs     /*ハンドラ起動時のレジスタを入れたパケットへのポインタ */
pk_eit      /*ハンドラ起動時の PC, PSW を入れたパケットへのポインタ */
```

ネストした各段のハンドラに対して、これらの情報を含んだ一つの構造体が対応する。現在実行中のハンドラから数えて hdstart 段目より hdcnt 段

	exccd	eclsptn
終了要求	ter_tsk(ras_ter)で指定したexccd	ECM_TER
強制例外	ras_fexで指定したexccd	ECM_FEX
CPU例外	CPU例外に関する情報(EITINF等)	ECM_CEX
システムコール例外	ercd(ret_svcで指定したs_ercd)	ercd(s_ercd)に依存

＊システムコール例外の場合の eclsptn は、ercd(s_ercd) 中のercls フィールドによって決まる。したがって、eclsptn の情報は、exccd の情報と重複している。

［表3.2］例外の種類とexccd, eclsptnとの関係

数のハンドラに関して、上記の情報を含む構造体を配列にしたもの（要素数はhdcnt個）が ar_hdrs に返される。

ar_hdrs に返される情報のうち、emsptn は、その例外ハンドラ、あるいは拡張SVCハンドラの実行環境における例外マスクの値を示す。

pk_exc は、例外ハンドラ（終了ハンドラ、強制例外ハンドラを含む）の場合にのみ有効な値を持ち、拡張SVCハンドラの場合には NADR となる。例外ハンドラの場合、pk_exc の指す領域には以下のような情報を含んでいる。

```
eclsptn     /* 例外クラスビットパターン */
exccd       /* 例外コード（エラーコード） */
fncd        /* 例外発生システムコール機能コード */
その他      /* インプリメント依存追加情報 */
```

この中の eclsptn の内容を調べることにより、このハンドラが終了ハンドラ、強制例外ハンドラ、CPU例外ハンドラ、システムコール例外ハンドラのいずれであるかを知ることができる。また、上記のうち、fncd はシステムコール例外ハンドラの場合にのみ意味を持ち、他の場合には0となる。発生した例外の種類と、pk_exc の中の exccd, eclsptn との関係は、［表3.2］のようになる。

pk_regs は、ハンドラ起動時の汎用レジスタ（このハンドラが起動されたことにより退避された汎用レジスタ）を含むパケットへのポインタである。また、pk_eit は、ハンドラ起動時の PC, PSW（このハンドラが起動されたことにより退避された PC, PSW）を含むパケットへのポインタである。ここで、pk_eit の指す T_EIT 構造体には、pc, psw 以外に eitinf, expc などの情報

も含まれているが、このシステムコールで返される pk_eit ではそれらの情報が入るべき領域は使用していない。eitinf, expc など未使用の領域を読み出した場合には、インプリメント依存の不定値が返る。また、拡張SVCハンドラ実行中の状態の場合は、pk_regs に含まれているレジスタの値（機能コードを入れるレジスタの値、ITRON/CHIPの場合はr0）を調べることにより、実行中の拡張SVCの機能コードを知ることができる。

ハンドラのネストの段数が (hdstart + hdcnt) よりも少なく、hdstart 以上であった場合には、ar_hdrs の配列のその次の要素の pk_eit の値が NADR となる。すなわち、ハンドラのネストの深さをNとすると、hdstart≦N＜(hdstart + hdcnt) の場合に ar_hdrs[N-hdstart].pk_eit が NADR となる。特に、タスクが全くハンドラを呼んでいない状態(N=0) において、hdstart=0, hdcnt＞0 としてこのシステムコールを発行すると、ar_hdrs [0].pk_eit が NADR となる

この場合、ar_hdrs の配列のそれ以下の要素に対する pk_eit、すなわち ar_hdrs[N-hdstart+1].pk_eit,ar_hdrs[N-hdstart+2].pk_eit, ... ar_hdrs[hdcnt-1].pk_eit の値は不定である。また、ar_hdrs[N-hdstart].pk_exc, ar_hdrs[N-hdstart+1].pk_exc, … ar_hdrs[hdcnt-1].pk_exc の値や、ar_hdrs[N-hdstart].pk_regs, ar_hdrs[N-hdstart+2].pk_regs, ... ar_hdrs[hdcnt-1].pk_regs の値も不定である。

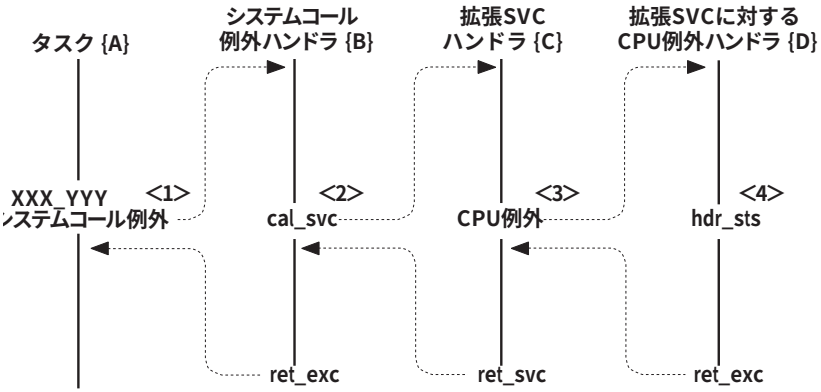
hdstart＞N の場合は、hdstart = N の場合と同じ動作をするものとする。すなわち、hdstart＞N, hdcnt＞0の場合は ar_hdrs [0].pk_eit が NADR になり、hdstart＞N, hdcnt = 0 の場合は ar_hdrs の領域は無変化で、単に E_OK が返る。

ハンドラのネストの段数が (hdstart + hdcnt) で指定した数よりも多い場合には、用意した配列の長さが短すぎるということであるから、E_AROVR のエラーとする。なお、エラー発生の場合にはできるだけ元の状態に戻すという一般原則とは異なり、このエラーの場合は、hdcnt 段までのハンドラの情報が ar_hdrs に返される。また、hdcnt が 0 の場合にも同様の動作をするが、この場合は、エラーコードが E_AROVR かどうかによって、ハンドラのネストの段数が hdstart より多いかどうかをチェックすることになる。hdcnt が 0 であれば、ar_hdrs の指す領域の内容は変化しない。

pk_exc, pk_regs, pk_eit は、例外ハンドラの起動時にパラメータとして渡される情報と同じものである。

【hdr_sts の実行例】

［図3.8］は、タスク{A}で XXX_YYY のシステムコールを発行したが、XXX_YYY でエラーが検出されたためシステムコール例外ハンドラ{B}が起動され、{B}の中でさらに拡張SVCハンドラ{C}を呼び、{C}の中で CPU例外が発生したためCPU例外ハンドラ{D}が起動された様子を示したものである。このような状況で hdr_sts (hdstart = 0, hdcnt = 5) が発行された場合に、ar_hdrs に戻される情報は次のようになる。



［図3.8］ hdr_stsの実行環境の例

ar_hdrs[0].emsptn = 例外ハンドラ{D}の環境の例外マスク
ar_hdrs[0].pk_exc->eclsptn = ECM_CEX
ar_hdrs[0].pk_exc->exccd = {C}で発生したCPU例外のEITINF
(CPU例外の例外コード)
ar_hdrs[0].pk_exc->fncd = 0
ar_hdrs[0].pk_regs = CPU例外発生直前(<3>)の{C}のレジスタ値
ar_hdrs[0].pk_eit = CPU例外発生直前(<3>)の{C}のPC,PSW

ar_hdrs[1].emsptn = 拡張SVCハンドラ{C}の環境の例外マスク
ar_hdrs[1].pk_exc = NADR
ar_hdrs[1].pk_regs = 拡張SVC呼びだし直前(<2>)の{B}のレジスタ値
ar_hdrs[1].pk_regs->r0 = {C}の拡張SVCの機能コード
ar_hdrs[1].pk_eit = 拡張SVC呼びだし直前(<2>)の{B}のPC,PSW

ar_hdrs[2].emsptn = 例外ハンドラ{B}の環境の例外マスク
ar_hdrs[2].pk_exc->eclsptn = {A}で発生したシステムコール例外の例外クラス
(XXX_YYYで戻されたエラーコードの例外クラス)
ar_hdrs[2].pk_exc->exccd = XXX_YYYで戻されたエラーコード
(システムコール例外の例外コード)
ar_hdrs[2].pk_exc->fncd = XXX_YYYの機能コード
ar_hdrs[2].pk_regs = XXX_YYY実行直後の{A}のレジスタ値
ar_hdrs[2].pk_regs->r0 = XXX_YYYで戻されたエラーコード
(ar_hdrs[2].pk_exc->exccdと同じ)
ar_hdrs[2].pk_eit = XXX_YYY実行直後の{A}のPC,PSW

ar_hdrs[3].emsptn = タスク{A}の環境の例外マスク
ar_hdrs[3].pk_exc = 不定
ar_hdrs[3].pk_regs = 不定
ar_hdrs[3].pk_eit = NADR

ar_hdrs[4].emsptn = 不定
ar_hdrs[4].pk_exc = 不定
ar_hdrs[4].pk_regs = 不定
ar_hdrs[4].pk_eit = 不定

{D}のCPU例外ハンドラの起動時に渡される pk_exc, pk_regs, pk_eit の値は、上記の ar_hdrs[0].pk_exc,ar_hdrs [0].pk_regs, ar_hdrs[0].pk_eit の値と同じものになる。また、{B}のシステムコール例外ハンドラの起動時に渡される pk_exc, pk_regs, pk_eit の値は、上記の ar_hdrs [2].pk_exc, ar_hdrs [2].pk_regs,ar_hdrs[2].pk_eit の値と同じものになる。

なお、hdr_sts では、{D}の環境のレジスタ値や PSW 値を参照することはできない。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦tskid≦-1、タスク独立部の発行でtskid=0）
E_PAR	一般的なパラメータエラー（hdstart＜0、hdcnt＜0）
E_ILADR	不正アドレス（ar_hdrsが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
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_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でtskid＜(-4)）
E_AROVR	用意した領域のサイズが小さすぎる

タスク付属同期機能

タスクを強制待ち状態へ移行する sus_tsk

sus_tsk: Suspend Task

【パラメータ】

tskid	TaskIdentifier	タスクID

【リターンパラメータ】

なし

【解説】

tskid で示されたタスクの実行を中断させ、強制待ち状態に移す。強制待ち状態は、rsm_tsk システムコールの発行によって解除される。強制待ち状態は、他タスクの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。

sus_tsk の要求はキューイングされる。すなわち、対象タスクが既に待ち状態や強制待ち状態であっても、sus_tsk の要求は無視されず、多重の待ち状態となる。sus_tsk が発行された回数だけ rsm_tsk を発行すると、必ずもとの状態に戻るため、sus_tsk 〜 rsm_tsk の対をネストすることが可能である。

WAIT-SUSPEND 状態のタスクに対しても、WAIT状態のタスクと同じように資源の割り当てが行なわれる。資源が割当てられることにより、そのタスクは SUSPEND 状態に移行する。WAIT-SUSPEND 状態のタスクに対する特別措置（資源割り当て遅延など）は取らない。これは、以下のような理由による。

— システムコールの意味的な問題
SUSPEND 状態は、他の処理とは直交した関係のものであり、SUSPEND の時だけ勝手に別の仕様に変えるのは良くないという考えによる。

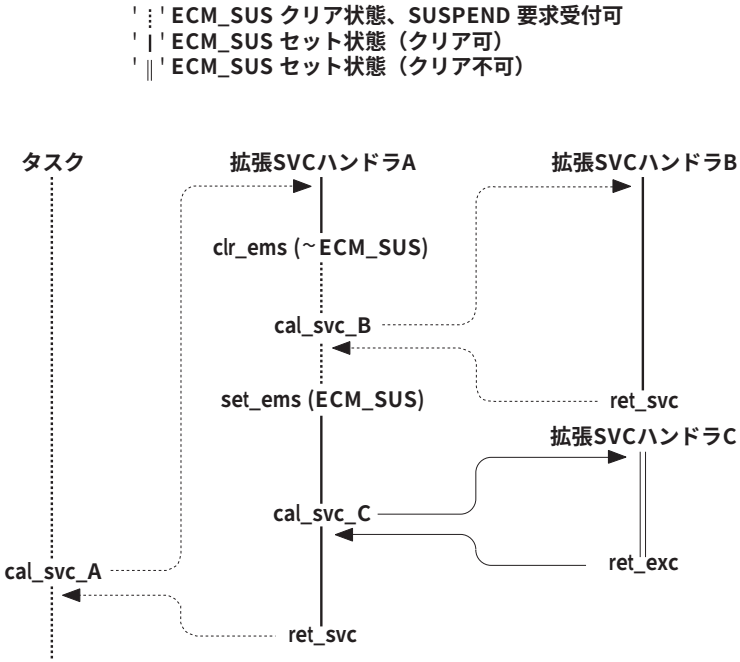
— 必要な場合は使い方の問題で対処可能であること
SUSPEND 状態のタスクの優先度を一時的に下げたいのであれば、ユーザタスクのレベルで、sus_tsk,rsm_tsk に chg_pri を組み合せて発行すれば済む。

サスペンド要求 (suscnt) のキューイング数の最大値は、標準で (2^31-1) である。ただし、'^' はべき乗を表わす。

拡張SVCハンドラ実行中は SUSPEND 要求を遅延（拒否）させることができる。具体的には、SUSPEND 要求を遅延させるかどうかの選択を例外マスク環境（例外管理の章参照）に含め、emsptn の最下位ビット (ECM_SUS) を SUSPEND 要求拒否ビットとしている。ECM_SUS がセットされている間は、SUSPEND 要求が遅延させられる。

拡張SVCハンドラ起動時のデフォルト値は、ECM_SUS がセットされた状態であり、そのままの状態では、拡張SVCハンドラが終了するまで SUSPEND 要求の遅延が行われる。拡張SVCハンドラが動き出してから、clr_ems (clrptn=~ECM_SUS) を実行することにより、それ以後は SUSPEND 要求を受け付けることができるようになる。

一方、タスク実行中は、ECM_SUS がクリアされた状態となっており、SUSPEND 要求は常に受け付け可能である。タスク実行中に set_ems (setptn = ECM_SUS) を実行した場合は、E_PAR のエラーとなる。



[図3.9] ECM_SUSをクリアできないケース

タスク、あるいは拡張SVCハンドラに対する例外ハンドラが起動された場合、例外マスク環境の中の ECM_SUSの値は、例外ハンドラ起動前の値をそのまま引き継ぐ。

SUSPEND 要求を遅延させている間は、その中からネストして呼ばれた拡張SVCハンドラや例外ハンドラの実行中も SUSPEND 要求を遅延させる必要がある。したがって、拡張SVCハンドラを呼んだ側の環境で SUSPEND 要求を遅延していた場合には、呼ばれた側の拡張SVCハンドラでは遅延状態を解除できないものとする。具体的には、clr_ems (clrptn=~ECM_SUS) を実行した場合に E_PAR のエラーになる。

拡張SVCハンドラの中で ECM_SUS をクリアできないケースについて、具体的な動作例を［図3.9］に示す。

sus_tsk の対象タスクが ECM_SUS をセットした状態になっていた場合でも、SUSPEND 要求が受け付けられないのではなく、遅延させられるだけである。したがって、sus_tsk を発行した側にはエラーが返らない。また、SUSPEND 要求が遅延させられている間も、tsk_sts で返る suscnt の情報はこのタスクに対する sus_tsk の実行を反映したものになっている。SUSPEND 要求の遅延中は、suscnt＞0かつ tskstat≠TTS_SUSとなることがある。

インプリメント上は、suscnt＞0 の場合に SUSPEND 状態とするのではなく、suscnt＞0.and. (emsptn &ECM_SUS) = 0 の場合に SUSPEND 状態とすれば良いことになる。また、例外マスク環境が変る時（clr_ems,ret_exc, ret_svc 等）には、遅延されている SUSPEND 要求が無いかどうかのチェックをする必要が生じる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦tskid≦-1、タスク独立部の発行でtskid = 0）
E_IDOVR	ID範囲外（インプリメント依存）
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_DMT	タスクがDORMANTである（tskidのタスクがDORMANT）
E_SELF	自タスク、自プロセスの指定（tskidが自タスク、タスク部や準タスク部の発行でtskid = 0）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でtskid＜(-4)）
E_QOVR	キューイングのオーバーフロー（suscntのオーバーフロー）

強制待ち状態のタスクを再開する
rsm_tsk

強制待ち状態のタスクを強制再開する
frsm_tsk

rsm_tsk:　Resume Task
frsm_tsk:　Force Resume Task

【パラメータ】

tskid	TaskIdentifier	タスクID

【リターンパラメータ】

なし

【解説】

tskid で示されたタスクが sus_tsk システムコールによって中断されている場合、対象タスクの強制待ち状態を解除し、実行を再開させる。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。

rsm_tsk システムコールでは、一回分の sus_tsk 要求を解除する。また、frsm_tsk システムコールでは、sus_tsk 要求が複数回キューイングされていても、それらをすべて解除する。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号(-4≦tskid≦-1、タスク独立部の発行でtskid = 0)
E_IDOVR	ID範囲外(インプリメント依存)
E_SELF	自タスク、自プロセスの指定(tskidが自タスク、タスク部や準タスク部の発行でtskid=0)
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_DMT	タスクがDORMANTである(tskidのタスクがDORMANT)
E_NOSUS	タスクがSUSPENDでない
E_OACV	オブジェクトアクセス権違反(拡張SVC以外のユーザタスクからの発行でtskid＜(-4))

タスクを待ち状態へ移行する
slp_tsk

タスクを一定時間待ち状態に移行する
wai_tsk

slp_tsk:　Sleep Task
wai_tsk:　Wait for Wakeup Task

【パラメータ】

＜1＞tmout　Timeout　タイムアウト指定
［＜1＞wai_tsk のみ］

【リターンパラメータ】

なし

【解説】

slp_tsk システムコールでは、自タスクを実行状態から待ち状態に移す。この待ち状態は、本タスクを対象として発行された wup_tsk システムコールにより解除される。

wai_tsk システムコールでは、自タスクを一定時間だけ実行状態から待ち状態に移す。この待ち状態は、本タスクを対象とした wup_tsk システムコールの発行、本タスクを対象とした ter_tsk, ras_fex システムコールの発行、あるいは tmout で指定した時間の経過により解除される。wup_tsk が発行された場合は正常終了、時間経過の場合はタイムアウトエラー E_TMOUT としてリターンする。このシステムコールは、slp_tsk システムコールにタイムアウト機能を付けたものである。

wai_tsk の tmout は正の値のみが有効である。(-2^31)〜(-2) の値の指定は reserved であり、指定した場合には E_ILTIME のエラーとなる。また、tmout = TMO_FEVR によりタイムアウト指定が行なわれないことを示す。この場合は、wup_tsk が発行されるまで永久に待ち状態になるため、slp_tsk と同じ動作をする。また、wai_tsk で tmout = TMO_POL を利用することも可能である。wai_tsk (tmout = TMO_POL) では、起床要求が出ているかどう

かをチェックして、起床要求が出ている場合にはそれを1つ減らして正常終了し、起床要求が出ていない場合には E_TMOUT のエラーになる。

自タスクの指定のみであるため、slp_tsk, wai_tsk のネスティングは有り得ないが、slp_tsk, wai_tsk により待ち状態となっている時に、他のタスクから sus_tsk が実行されることはある。この場合はwup_tsk により待ち状態が解除されてもまだ強制待ち状態であり、rsm_tsk の発行までタスクの実行は再開されない。

なお、タスクの単純な遅延を行うのであれば、wai_tsk 以外に、dly_tsk も利用できる。

【エラーコード (ercd)】

E_OK	正常終了
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_ILTIME	不正時間指定（tmout≦-2）
E_CTX	コンテキストエラー（タスク独立部あるいはディスパッチ遅延中のタスクから発行）
E_TMOUT	タイムアウト
E_RLWAI	待ち状態強制解除（待ちの間にter_tsk,rel_waiを受け付け）

待ち状態のタスクを起床する wup_tsk

wup_tsk: Wakeup Task

【パラメータ】

tskid	TaskIdentifier	タスクID

【リターンパラメータ】

なし

【解説】

wup_tsk システムコールでは、slp_tsk または wai_tsk システムコールの実行により待ち状態になっていたタスクを、待ち状態から実行可能状態に移す。対象タスクは、tskid で示され、自タスクを指定することはできない。

対象タスクが slp_tsk または wai_tsk を実行しておらず、待ち状態でない場合には、この wup_tsk 要求はキューイングされる。その場合、この wup_tsk 要求は、後に対象タスクが slp_tsk または wai_tsk システムコールを実行した時に有効となる。具体的には、wup_tsk を実行すると対象タスクの起床要求カウントがプラス1され、対象タスクが slp_tsk または wai_tsk を実行すると、対象タスクの起床要求カウントがマイナス1される。起床要求カウントが負になろうとした時に、はじめて待ち状態になる。

起床要求(wupcnt)のキューイング数の最大値は、標準で (2^31-1) である。ただし、'^' はべき乗を表わす。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号(-4≦tskid≦-1、タスク独立部の発行でtskid = 0)
E_IDOVR	ID範囲外（インプリメント依存）
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_DMT	タスクがDORMANTである（tskidのタスクがDORMANT）
E_SELF	自タスク、自プロセスの指定（tskidが自タスク、タスク部や準タスク部の発行でtskid=0）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でtskid＜(-4)）
E_QOVR	キューイングのオーバーフロー(wupcntのオーバーフロー)

タスクの起床要求を無効にする can_wup

can_wup: Cancel Wakeup Task

【パラメータ】

tskid	TaskIdentifier	タスクID

【リターンパラメータ】

wupcnt	Wakeup Count	キューイングされていた起床要求回数

【解説】

can_wup システムコールでは、tskid で示された対象タスクにキューイングされていた起床要求回数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解除する。tskid = TSK_SELF によって自タスクの指定になる。

このシステムコールは、周期起床において、時間内に処理が終わっているかどうか（前の起床要求に対するslp_tsk が実行される前に、次の起床要求が発生していないか）を判定する必要がある場合に利用できる。can_wup のリターンパラメータである wupcnt が 0でなければ、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がわかるので、それに対して何らかの処置をすることができる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦tskid≦-1、タスク独立部の発行でtskid=0）
E_IDOVR	ID範囲外（インプリメント依存）
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_DMT	タスクがDORMANTである（tskidのタスクがDORMANT）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でtskid＜(-4)）

同期・通信機能

イベントフラグを生成する
cre_flg

cre_flg:　Create EventFlag

【パラメータ】

flgid	EventFlagIdentifier	イベントフラグID
flgatr	EventFlagAttribute	イベントフラグ属性

【リターンパラメータ】

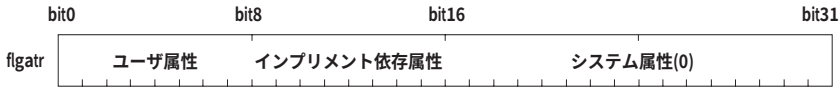
なし

【解説】

cre_flg では、flgid で指定された ID 番号を持つイベントフラグを生成する。次に、生成された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 (flgptn) を0とする。一つのイベントフラグで、プロセッサの1ワード（32ビット）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。

ID 番号が (-4)〜0 のイベントフラグは生成できない。また、負の ID 番号のイベントフラグは、システム用のものである。

flgatr のサイズは標準で4バイトである。flgatr のフォーマットを［図3.10］に示す。このうち、最上位バイト（bit0〜bit7 あるいは 2^31〜2^24 のビット）はユーザ属性を表わし、第二上位バイト（bit8〜bit15 あるいは 2^23〜2^16 のビット）はインプリメント依存の属性を表わし、下位ハーフワード（bit16〜bit31 あるいは 2^15〜2^0 のビット）はシステム属性を表わす。ユーザは、イベントフラグに関する情報を入れておくために、flgatr の最上位バイトを自由に使用することができる。



［図3.10］flgatrのフォーマット

cre_flg で指定した flgatr は、flg_sts により読み出すことができる。
flgatr のシステム属性の部分では、次のような指定を行うことができる。

flgatr: = (TA_WMUL ‖ TA_WSGL)

TA_WMUL	複数タスクの待ちを許す (Wait Multiple Task)
TA_WSGL	複数タスクの待ちを許さない (Wait Single Task)

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足 ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSID	予約ID番号（-4≦flgid≦0）
E_RSATR	予約属性（flgatrの下位3バイトが不正）
E_IDOVR	ID範囲外（flgidがシステムで利用できる範囲を越えた）
E_EXS	オブジェクトが既に存在している（同一ID番号のイベントフラグが存在）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でflgid＜(-4)）

イベントフラグを削除する

del_flg

del_flg:　Delete EventFlag

【パラメータ】

flgid	EventFlagIdentifier	イベントフラグID

【リターンパラメータ】

なし

【解説】

flgid で示されるイベントフラグを削除する。そのイベントフラグにおいてイベント発生を待っているタスクがある場合にも本システムコールは正常終了するが、イベント発生を待っていたタスクにはエラー E_DLT が返される。

本システムコール発行後は、同じID番号のイベントフラグを再び新しく生成することができる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦flgid≦0）
E_IDOVR	ID範囲外（インプリメント依存）
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_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でflgid＜(-4)）

イベントフラグをセットする

set_flg

イベントフラグをクリアする

clr_flg

set_flg: Set EventFlag
clr_flg: Clear EventFlag

【パラメータ】

flgid	EventFlagIdentifier	イベントフラグID
＜1＞setptn	SetBitPattern	セットするビットパターン
＜2＞clrptn	ClearBitPattern	クリアするビットパターン

［＜1＞set_flg のみ］
［＜2＞clr_flg のみ］

【リターンパラメータ】

なし

【解説】

set_flg では、flgid で示される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、clr_flg では、flgid で示されるイベントフラグのうち、対応する clrptn が 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。

set_flg において、イベントフラグ値の変更の結果、そのイベントフラグを待っていたタスクの待ち解除条件を満たすようになれば、そのタスクが実行可能状態へと移行する。イベントフラグでは、ビット対応のイベントの OR や AND を条件として待つことができるが、そのほか、イベントフラグの全ビットを使用すれば、1ワードの簡単なメッセージ転送（キューイング無し）を行なうこともできる。

同一イベントフラグに対する複数タスクの待ちも可能である。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 set_flg で複数のタスクが待ち解除となることがある。

clr_flg では、その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。

set_flg で setptn の全ビットを 0 とした場合や、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、その場合でもエラーとはならない。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦flgid≦0）
E_IDOVR	ID範囲外（インプリメント依存）
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_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でflgid＜(-4)）

イベントフラグを待つ

wai_flg

wai_flg: Wait EventFlag

【パラメータ】

flgid	EventFlagIdentifier	イベントフラグID
waiptn	WaitBitPattern	待ちビットパターン
tmout	Timeout	タイムアウト指定
wfmode	WaitEventFlagMode	待ちモード

【リターンパラメータ】

flgptn	EventFlagBitPattern	待ち解除時のビットパターン

【解説】

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

wfmode では、次のような指定を行う。

wfmode : = (TWF_ANDW ‖ TWF_ORW) ｜ [TWF_CLR]

TWF_ANDW	AND待ち
TWF_ORW	OR待ち
TWF_CLR	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。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 で指定したビットがすべてセットされるのを待つ。

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。一方、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）を 0 にクリアする。

tmout により待ち時間のタイムアウト指定を行なうことができる。tmout としては、正の値のみを指定することができる。タイムアウト指定

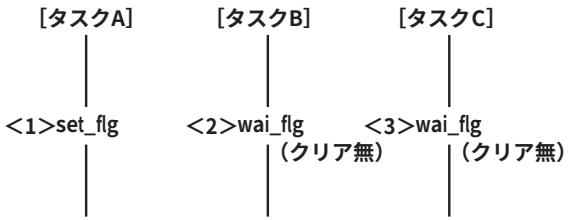
が行なわれた場合、条件が満足されぬまま tmout 時間が経過すると、タイムアウトエラー E_TMOUT としてエラーリターンする。また、tmout として TMO_POL を指定した場合は、待ち条件が満たされない場合でも即時にリターンする。この場合、待ち条件が満たされていれば正常終了に、待ち条件が満たされていなければタイムアウトエラー E_TMOUT になる。TWF_CLR の指定によってフラグのクリアが行われるのは、正常終了した場合のみである。さらに、tmout = TMO_FEVR によりタイムアウト指定が行なわれないことを示す。この場合は条件が満足されるまで永久に待つ。

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 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。

TA_WMUL 属性のイベントフラグの場合は、同一イベントフラグに対する複数タスクの待ちも可能である。この場合、一回の set_flg で複数のタスクが待ち解除となることがある。TA_WMUL 属性のイベントフラグでは、タスクの待ち行列が定義され、次のような動作をする。

— 待ち行列の順番は FIFO 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ため、待ち解除とはならない。

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＞set_flg を実行するまで、タスクB、タスクC を＜2＞＜3＞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3.11］。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 wai_flg がエラーになる。



[図3.11] イベントフラグに対する複数タスク待ちの機能

waiptn を0とした場合は、set_flg により待ち状態から抜けることができなくなるため、パラメータエラー E_PAR とする。したがって、set_flg で全ビットをセットした場合、待ち行列の先頭にあるタスクであれば、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解除となることが保証される。

一方、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 wai_flg を実行することはできない。ただし、これに違反した場合にエラーチェックを行って E_OBJ のエラーを返すかどうかはインプリメント依存である。インプリメントによっては、TA_WSGLに対するエラーチェックが行われない場合がある。互換性の高いプログラミングを行うためには、TA_WSGL 属性のイベントフラグに対する複数のタスク待ちが起こらないようにしなければならない。

【エラーコード (ercd)】

E_OK	正常終了
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSMD	予約モード、予約オプション（wfmodeが不正）
E_RSID	予約ID番号（-4≦flgid≦0）
E_PAR	一般的なパラメータエラー（waiptn = 0）
E_IDOVR	ID範囲外（インプリメント依存）
E_ILTIME	不正時間指定（tmout≦-2）

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_OBJ	オブジェクト状態に関するその他のエラー（TA_WSGL属性のイベントフラグに対する複数タスクの待ち,インプリメント依存）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でflgid＜(-4)）
E_CTX	コンテキストエラー（タスク独立部あるいはディスパッチ遅延中のタスクから発行）
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_TMOUT	タイムアウト
E_RLWAI	待ち状態強制解除（待ちの間にter_tsk,rel_waiを受け付け）

イベントフラグ状態を参照する flg_sts

flg_sts: Get EventFlag Status

【パラメータ】

flgid	EventFlagIdentifier	イベントフラグID
pk_flgs	Packet of EventFlagStatus	イベントフラグ状態を返すパケットの先頭アドレス

【リターンパラメータ】

なし

【解説】

flgid で示されたイベントフラグの状態を読み出し、その結果を pk_flgs 以下の領域に返す。

pk_flgs に返される情報としては、次のようなものがある。

```
flgatr      /* イベントフラグ属性 */
wtskid      /* 待ち行列先頭のタスクID */
flgptn      /* 現在のフラグ値 */
```

wtskid には、待ち行列の先頭のタスクのIDが返る。すなわち、現在このイベントフラグを待っているタスクのうち、最も早く待ち状態になったタスクのIDが返る。待ちタスクが無い場合は FALSE = 0 が返る。

対象となるイベントフラグは、既に生成されたものでなければならない。

なお、ITRONでは、cre_XXX で生成したオブジェクトや def_XXX で定義したハンドラの状態を見るシステムコールの名称を XXX_sts としている。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦flgid≦0）
E_ILADR	不正アドレス（pk_flgsが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
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_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でflgid＜(-4)）

セマフォを生成する

cre_sem

cre_sem: Create Semaphore

【パラメータ】

semid	SemaphoreIdentifier	セマフォID
sematr	SemaphoreAttribute	セマフォ属性
isemcnt	InitialSemaphoreCount	セマフォの初期値

【リターンパラメータ】

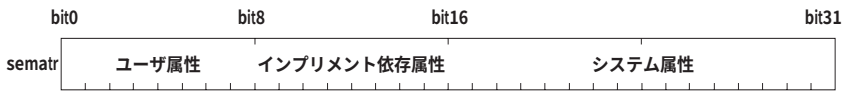
なし

【解説】

cre_sem では、semid で指定されたID番号を持つセマフォを生成する。次に、生成されたセマフォに対して管理ブロックを割り付け、そのセマフォ初期値を isemcnt とする。

ID 番号が (-4)〜0 のセマフォは生成できない。また、負の ID 番号のセマフォは、システム用のものである。

sematrのサイズは標準で4バイトである。sematrのフォーマットを［図3.12］に示す。このうち、最上位バイト（bit0〜bit7 あるいは 2^31〜2^24 のビット）はユーザ属性を表わし、第二上位バイト（bit8〜bit15 あるいは 2^23〜2^16 のビット）はインプリメント依存の属性を表わし、下位ハーフワード（bit16〜bit31 あるいは 2^15〜2^0 のビット）はシステム属性を表わす。ユーザは、セマフォに関する情報を入れておくために、sematrの最上位バイトを自由に使用することができる。



［図3.12］sematrのフォーマット

cre_sem で指定した sematr は、sem_sts により読み出すことができる。
sematr のシステム属性の部分では、次のような指定を行うことができる。

sematr: = (TA_TFIFO ‖ TA_TPRI) ｜ (TA_FIRST ‖ TA_CNT)
　　｜ (TA_VAR ‖ TA_FIX)

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	行列先頭のタスクを優先扱い
TA_CNT	要求数の少ないタスクを優先扱い
TA_VAR	要求/返却資源数を指定可能
TA_FIX	要求/返却資源数は1のみ

TA_VAR の属性を指定したセマフォに対しては、sig_sem, wai_sem の要求/返却資源数 (rcnt) として、任意の正の値を指定することができる。一方、TA_VAR の属性を指定したセマフォに対しては、sig_sem, wai_sem の要求 / 返却資源数 (rcnt) は 1 に固定される。

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。

また、TA_CNT と TA_FIRST の属性指定により、要求カウント数の少ないタスクを優先扱いにするか、行列先頭のタスクを優先扱いにするかが指定できる。TA_CNT の属性を指定した場合には、要求カウント数によって、行列途中のタスクから先に待ち解除になる場合がある。例えば、あるセマフォに対して 要求カウント=5 のタスク A と要求カウント = 1 のタスク B がこの順で待っており、sig_sem によりカウントが 1 になった場合、行列先頭にあるタスク A は要求カウント数が多いので、行列の後にあるタスク B の方が先に待ち解除になる。

TA_FIRST の属性を指定した場合には、カウント数にかかわらず、必ず行列先頭のタスクから資源が割り当てられる。これは、カウント要求数よりも行列の順序や優先度を重視した考え方である。上の例では、sig_sem 実行後も タスク B は待ち解除にならず、さらに sig_sem が実行されてセマフォのカウント数が 5 以上になった時にはじめてタスク A が待ち解除になり、その後タスク B が待ち解除になる。

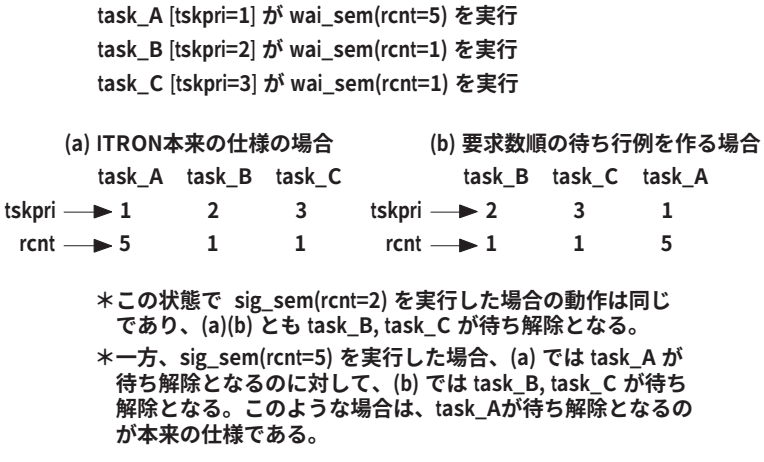
セマフォやメモリプールの待ち行列の属性としては、

TA_TFIFO または TA_TPRI と、
TA_FIRST または TA_CNT

を組み合わせて指定できるようになっている。このうち、TA_TPRI と TA_TFIFO の属性は、待ち行列の作り方を指定するものであり、wai_sem のシステムコール発行時にのみ参照され、sig_sem の処理では参照されない。

一方、TA_CNT と TA_FIRST の指定は、待ち行列の順序には影響しない。TA_CNT を指定した場合でも、資源要求数 (rcnt) の少ない順に待ち行列を構成するわけではない。待ち行列の構成順序は、常にタスク優先度順 (TA_TPRI) または先着順 (TA_TFIFO) のいずれかであり、資源要求数は待ち行列の構成順序には関係しない。

もし、TA_CNT 指定時に資源要求数順の待ち行列を作るものとすると、［図3.13］のような場合の動作が異なってくる。



［図3.13］セマフォの待ち行列の作り方による動作の違い

なお、TA_FIX 属性の場合は TA_CNT と TA_FIRST の属性の違いが無くなってしまうが、これに関しては何もチェックしない。(TA_FIX | TA_CNT) の指定と (TA_FIX | TA_FIRST) の指定はどちらもエラーにはならず、同じ動作を行なう。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足 ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSID	予約ID番号（-4≦semid≦0）
E_RSATR	予約属性（sematrの下位3バイトが不正）
E_PAR	一般的なパラメータエラー（isemcntが負）
E_IDOVR	ID範囲外（semidがシステムで利用できる範囲を越えた）
E_EXS	オブジェクトが既に存在している（同一ID番号のセマフォが存在）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でsemid＜(-4)）

セマフォを削除する

del_sem

del_sem: Delete Semaphore

【パラメータ】

semid	SemaphoreIdentifier	セマフォID

【リターンパラメータ】

なし

【解説】

semid で示されるセマフォを削除する。

そのセマフォにおいて sig_sem を待っているタスクがある場合にも本システムコールは正常終了するが、イベント発生を待っていたタスクにはエラー E_DLT が返される。

本システムコール発行後は、そのID番号のセマフォを再び新しく生成することができる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦semid≦0）
E_IDOVR	ID範囲外（インプリメント依存）
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_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でsemid＜(-4)）

セマフォに対する信号操作（V命令）　sig_sem

sig_sem:　Signal Semaphore

【パラメータ】

semid	SemaphoreIdentifier	セマフォID
rcnt	ResourceCount	増カウント値

【リターンパラメータ】

なし

【解説】

semid で示されたセマフォのカウント値を rcnt だけ増やす。その後、そのセマフォの待ち行列の中から、減カウント値がセマフォのカウンタ値以下であるタスクを選んで実行可能状態に移行させ、セマフォのカウンタ値を減カウント値だけ減らす。実行可能状態にするタスクの選び方は、cre_sem 時に指定する sematr (TA_CNT,TA_FIRST) で指定される。sig_sem で指定した増カウント値と、待ち行列中のタスクの減カウント値との関係によって、実行可能状態に移行するタスクが無い場合や、複数のタスクが実行可能状態に移行する場合が生じる。

セマフォのカウント値の増加により、カウント値が cre_sem で指定した isemcnt の値を越えた場合にも、エラーとはしない。これは、資源管理ではなく、同期の目的（wup_tsk〜slp_tsk と同様）でセマフォを使用することを想定しているためである。また、セマフォを一旦生成した後で、資源数を動的に再設定したい場合にも、カウント値が isemcnt を越える場合がある。

TA_FIX 属性のセマフォに対して、rcnt≠1 の sig_sem, wai_sem を発行することはできない。ただし、これに違反した場合に rcnt=1 と同じ動作をするか、指定した rcnt がそのまま有効になる（TA_VAR属性のセマフォと同じ動作をする）かはインプリメント依存である。互換性の高いプログラミングを行うためには、TA_FIX属性のセマフォに対して rcnt≠1 の sig_sem, wai_sem を発行してはならない。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号 (-4≦semid≦0)
E_PAR	一般的なパラメータエラー (rcnt≦0)
E_IDOVR	ID範囲外 (インプリメント依存)
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_OACV	オブジェクトアクセス権違反 (拡張SVC以外のユーザタスクからの発行でsemid＜(-4))
E_QOVR	キューイングのオーバーフロー(semcntのオーバーフロー)

セマフォに対する待ち操作（P命令）　wai_sem

wai_sem:　Wait on Semaphore

【パラメータ】

semid	SemaphoreIdentifier	セマフォID
rcnt	ResourceCount	減カウント値
tmout	Timeout	タイムアウト指定

【リターンパラメータ】

なし

【解説】

semid で示されたセマフォのカウント値が rcnt 以上であれば、セマフォのカウント値を rcnt だけ減じて、本システムコールの発行タスクは実行を継続する。セマフォのカウント値が rcnt より小さければ、セマフォ値は変更せず、このシステムコールを発行したタスクはそのセマフォの待ち行列につながれる。待ち行列へのつながれ方は、セマフォ生成時に sematr で指定されたセマフォ属性により、FIFO または タスク優先度順のいずれかが選択される。

tmout により待ち時間のタイムアウト指定を行なうことができる。tmout としては、正の値のみ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、条件が満足されぬまま tmout 時間が経過すると、タイムアウトエラー E_TMOUT としてエラーリターンする。また、tmout として TMO_POL を指定した場合は、資源が獲得できない場合でも即時にリターンする。この場合、資源が獲得できれば正常終了に、資源が獲得できなければタイムアウトエラー E_TMOUT になる。さらに、tmout = TMO_FEVR によりタイムアウト指定が行なわれないことを示す。この場合は条件が満足されるまで永久に待つ。

TA_FIX 属性のセマフォに対して、rcnt≠1 の sig_sem, wai_sem を発行することはできない。ただし、これに違反した場合に rcnt=1 と同じ動作をするか、指定した rcnt がそのまま有効になる（TA_VAR属性のセマフォと同じ

動作をする）かはインプリメント依存である。互換性の高いプログラミングを行うためには、TA_FIX 属性のセマフォに対して rcnt≠1 の sig_sem, wai_sem を発行してはならない。

セマフォやメモリブロックなどの資源を要求するシステムコール (wai_sem, get_blk) では、

(a) 資源要求時に資源要求タスクを一旦待ち行列につなぎ、
(b) それから資源が割り当て可能かどうか（タスクが本当に待ち状態になるかどうか）をチェックする、

と考えて処理を行う。この場合、(a)の処理の部分では、TA_FIRST/TA_CNT の属性は関係せず、TA_TFIFO/TA_TPRIの属性のみが関係する。一方、(b)の処理の部分では、TA_TFIFO/TA_TPRIの属性は関係せず、TA_FIRST/TA_CNT の属性のみが関係する。

【エラーコード (ercd)】

E_OK	正常終了
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSID	予約ID番号（-4≦semid≦0）
E_PAR	一般的なパラメータエラー（rcnt≦0）
E_IDOVR	ID範囲外（インプリメント依存）
E_ILTIME	不正時間指定（tmout≦-2）
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_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でsemid＜(-4)）
E_CTX	コンテキストエラー（タスク独立部あるいはディスパッチ遅延中のタスクから発行）
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_TMOUT	タイムアウト
E_RLWAI	待ち状態強制解除（待ちの間にter_tsk,rel_waiを受け付け）

セマフォ状態を参照する
sem_sts

sem_sts:　Get Semaphore Status

【パラメータ】

semid	SemaphoreIdentifier	セマフォID
pk_sems	Packet of SemaphoreStatus	セマフォ状態を返すパケットの先頭アドレス

【リターンパラメータ】

なし

【解説】

semid で示されたセマフォの状態を読み出し、その結果を pk_sems 以下の領域に返す。

pk_sems に返される情報としては、次のようなものがある。

```
sematr      /* セマフォ属性 */
wtskid      /* 待ち行列先頭のタスクID */
semcnt      /* 現在のカウント値 */
```

wtskid には、待ち行列の先頭のタスクのIDが返る。すなわち、セマフォ属性が TA_TFIFO であれば、現在このセマフォを待っているタスクのうち、最も早く待ち状態になったタスクのIDが返る。また、セマフォ属性が TA_TPRI であれば、現在このセマフォを待っているタスクのうち、最も優先度の高いタスクのIDが返る。待ちタスクの無い時は FALSE = 0 が返る。

wai_sem で待ちタスクの指定している減カウント値が、セマフォの現在のカウント値よりも大きければ、semcnt＞0かつ wtskid≠0となる場合がある。

対象となるセマフォは、既に生成されたものでなければならない。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号 (-4≦semid≦0)
E_ILADR	不正アドレス (pk_semsが使用できない値)
E_IDOVR	ID範囲外(インプリメント依存)
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_OACV	オブジェクトアクセス権違反 (拡張SVC以外のユーザタスクからの発行でsemid＜(-4))

メイルボックスを生成する

cre_mbx

cre_mbx:　Create Mailbox

【パラメータ】

mbxid	MailboxIdentifier	メイルボックスID
mbxatr	MailboxAttribute	メイルボックス属性

【リターンパラメータ】

なし

【解説】

cre_mbx では、mbxid で指定された ID 番号を持つメイルボックスを生成する。次に、生成されたメイルボックスに対して管理ブロックを割り付ける。メイルボックスは、そのメイルボックスに入れられたメッセージのキューと、それを待つタスクの待ち行列とから構成される。メッセージのキューは、リングバッファではなく、メッセージを線形リストでつないだものである。

ID 番号が (-4)〜0 のメイルボックスは生成できない。また、負の ID 番号のメイルボックスは、システム用のものである。

mbxatrのサイズは標準で4バイトである。mbxatrのフォーマットを［図3.14］に示す。このうち、最上位バイト（bit0〜bit7 あるいは 2^31〜2^24 のビット）はユーザ属性を表わし、第二上位バイト（bit8〜bit15あるいは 2^23〜2^16 のビット）はインプリメント依存の属性を表わし、下位ハーフワード（bit16〜bit31 あるいは 2^15〜2^0 のビット）はシステム属性を表わす。ユーザは、メイルボックスに関する情報を入れておくために、mbxatr の最上位バイトを自由に使用することができる。

cre_mbx で指定した mbxatr は、mbx_sts により読み出すことができる。

mbxatr のシステム属性の部分では、次のような指定を行うことができる。

mbxatr: = (TA_TFIFO ‖ TA_TPRI) ｜ (TA_MFIFO ‖ TA_MPRI)

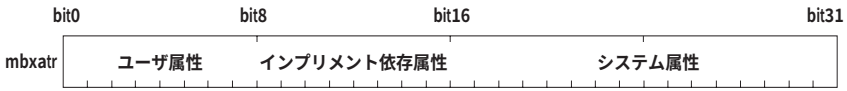
TA_TFIFO	待ちタスクのキューイングは FIFO
TA_TPRI	待ちタスクのキューイングは優先度順

TA_MFIFO	メッセージはFIFO
TA_MPRI	メッセージは優先度順

TA_TFIFO の属性を指定した場合には、メッセージを待つタスクは FIFO の待ち行列を作り、TA_TPRI の属性を指定した場合には、メッセージを待つタスクはタスクの優先度順の待ち行列を作る。また、TA_MFIFO の属性を指定した場合には、メッセージは FIFO のキューを作り、TA_MPRI の属性を指定した場合には、メッセージはメッセージに与えられた優先度 msgpri にしたがってキューを作る。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足 ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSID	予約ID番号（-4≦mbxid≦0）
E_RSATR	予約属性（mbxatrの下位3バイトが不正）
E_IDOVR	ID範囲外（mbxidがシステムで利用できる範囲を越えた）
E_EXS	オブジェクトが既に存在している（同一ID番号のメイルボックスが存在）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でmbxid＜(-4)）



［図3.14］ mbxatrのフォーマット

メイルボックスを削除する

del_mbx

del_mbx: Delete Mailbox

【パラメータ】

mbxid	MailboxIdentifier	メイルボックスID

【リターンパラメータ】

なし

【解説】

mbxid で示されるメイルボックスを削除する。

そのメイルボックスに対してメッセージの到着を待っているタスクがある場合、本システムコールは正常終了するが、メッセージの到着を待っていたタスクにはエラー E_DLT が返される。

本システムコールでは、対象メイ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、メイルボックスの削除が行なわれる。

本システムコール発行後は、そのID番号のメイルボックスを再び新しく生成することができる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号 (-4≦mbxid≦0)
E_IDOVR	ID範囲外 (インプリメント依存)
E_NOEXS	オブジェクトが存在していない (mbxidのメイルボックスが存在しない)
E_OACV	オブジェクトアクセス権違反 (拡張SVC以外のユーザタスクからの発行でmbxid＜(-4))

メイルボックスへ送信する snd_msg

snd_msg: Send Message to Mailbox

【パラメータ】

mbxid	MailboxIdentifier	メイルボックスID
pk_msg	MessagePacket	送信メッセージの先頭アドレス

【リターンパラメータ】

なし

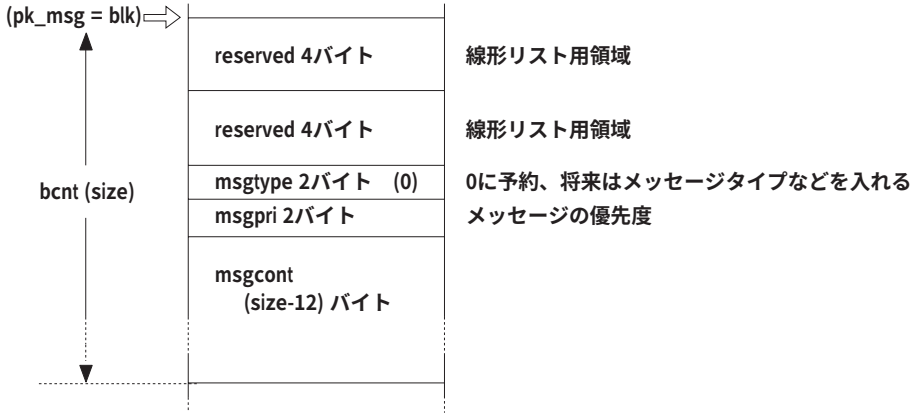
【解説】

mbxid で示された対象メイルボックスに、pk_msg のアドレスに入っているメッセージを送信する。メッセージの内容はコピーされず、受信時にはアドレス（pk_msgの値）のみが渡される。

対象メイルボックスでメッセージを待つタスクがない時に snd_msg が発行されても、snd_msg 発行タスクは待ち状態とはならない。このシステムコールでは、メッセージをメイルボックスに入れ、メッセージキューと呼ぶ待ち行列につなぐだけで、タスクはその先の実行を続ける。つまり、非同期のメッセージ送信を行なう。また、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（キュー）、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。

一般に、線形リストによりメッセージのキューを作る場合、メッセージブロックの中に、OSで用いるキューのリンク用エリアが必要となる。また、対象メイルボックスのメッセージの並び方の属性がメッセージ優先度順となっている場合には、メッセージの内部にメッセージの優先度を入れておく必要がある。したがって、OSの管理用として、ユーザの用意したメッセージブロックの一部を使用することになる。この部分を、メッセージヘッダと呼ぶ。

pk_msg は、メッセージヘッダをも含めたメッセージブロックの先頭アドレスである。実際にユーザがメッセージを入れることができるのは、ヘッダの後の部分である。



＊pk_msgは、メッセージの先頭アドレスであり、snd_msgのパラメータで指定される値である。メッセージとして、メモリプールから確保したメモリブロックをそのまま使用する場合にはget_blkで得られるメモリブロックのアドレス (blk)をpk_msgとして指定すれば良い。
＊可変長メモリブロックの場合には、一般に各メモリブロックに対応してmplid,bcnt（サイズ）等の情報をOSが管理するための領域が必要になる。しかし、その領域はメッセージヘッダとは別になっており、上記のメッセージブロック（＝メモリブロック）の図には含まれていない。

［図3.15］ITRON2におけるメッセージの形式

メッセージの具体的な形式は［図3.15］のようになる。このうち、ユーザが自由に使用できるのは、msgcont の部分である。

メッセージとしては、get_blk でメモリプールから動的に確保されたメモリブロックを使用することも可能であるし、静的に確保された領域を使用することも可能である。しかし、一般には、メモリプールから動的に確保されたメモリブロックをメッセージとして使用することが多い。この場合、メモリブロックのサイズをそのままメッセージのサイズと考えることにより、ユーザが改めてメッセージサイズをセットする必要は無くなる。受信した側でサイズを参照したい場合には、blk_sts システムコール（メモリブロックアドレスblkからブロック数bcntを返す）を利用すれば良い。

TA_MPRI 属性のメイルボックスにおけるメッセージ優先度 msgpri は、符号付きの数として扱う。メッセージ優先度としては、タスク優先度と同じ範囲（(-16)〜255, ただし(-4)〜0を除く）の数を利用できることが必要である。ただし、範囲に関するエラーチェックを行うことによってオーバーヘッドを生じる可能性もあるので、上記の範囲外のメッセージ優先度を指定した場合の動作はインプリメント依存とする。具体的には、次のような仕様になる。

msgpri が (-16)〜(-5)の場合 :
　実行可能

msgpri が 1〜255 の場合 :
　実行可能

msgpri が 上記の範囲以外の場合 :
　インプリメントに依存して、実行可能な場合とエラー(E_ILMSG) になる場合とがある。(-16)〜(-5), 1〜255 以外のメッセージ優先度を使ったアプリケーションでは、プログラムの互換性が保証されない。

互換性の高いプログラミングを行うためには、(-16)〜(-5), 1〜255 以外のメッセージ優先度を使わないようにする必要がある。

一方、TA_MFIFO 属性のメイルボックスに送ったメッセージの場合は、メッセージ優先度 msgpri の入る領域を別の目的で使用している可能性があるので、この領域がいかなる値であってもエラーとはしない。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦mbxid≦0）
E_ILADR	不正アドレス（pk_msgが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
E_ILMSG	不正メッセージ形式（msgtype≠0）
E_NOEXS	オブジェクトが存在していない（mbxidのメイルボックスが存在しない）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でmbxid＜(-4)）

メイルボックスから受信する
rcv_msg

rcv_msg: Receive Message from Mailbox

【パラメータ】

mbxid	MailboxIdentifier	メイルボックスID
tmout	Timeout	タイムアウト指定

【リターンパラメータ】

pk_msg	MessagePacket	受信メッセージの先頭アドレス

【解説】

mbxid で示されたメイルボックスからメッセージを受信し、受信したメッセージの先頭アドレスをリターンパラメータとして返す。まだそのメイルボックスにメッセージが到着していない場合には、本システムコール発行タスクはメッセージ到着を待つ待ち行列につながれる。待ち行列へのつながれ方は、メールボックス生成時にmbxatr で指定した方法によるものであり、FIFO または タスク優先度のいずれかである。

pk_msg は、メッセージヘッダをも含めたメッセージブロックの先頭アドレスである。メッセージヘッダには、メッセージの優先度やOSで使用する予約領域が含まれており、実際にユーザがメッセージを入れることができるのは、ヘッダの後の部分である。メッセージヘッダの構成については、snd_msg の項を参照。

tmout により待ち時間のタイムアウト指定を行なうことができる。tmout としては、正の値のみ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、メッセージを受信できないまま tmout 時間が経過すると、タイムアウトエラー E_TMOUT としてエラーリターンする。また、tmout として TMO_POL を指定した場合は、メッセージが受信できない場合でも即時にリターンする。この場合、メッセージが受信できれば正常終了に、メッセージが無ければタイムアウトエラー E_TMOUT になる。さらに、tmout = TMO_FEVR によりタイムアウト指定が行なわれないことを示す。この場合は条件が満足されるまで永久に待つ。

【エラーコード (ercd)】

E_OK	正常終了
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSID	予約ID番号（-4≦mbxid≦0）
E_IDOVR	ID範囲外（インプリメント依存）
E_ILTIME	不正時間指定（tmout≦-2）
E_NOEXS	オブジェクトが存在していない（mbxidのメイルボックスが存在しない）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でmbxid＜(-4)）
E_CTX	コンテキストエラー（タスク独立部あるいはディスパッチ遅延中のタスクから発行）
E_DLT	待ちオブジェクトが削除された（待ちの間に対象メイルボックスが削除）
E_TMOUT	タイムアウト
E_RLWAI	待ち状態強制解除（待ちの間にter_tsk,rel_waiを受け付け）

メイルボックス状態を参照する　mbx_sts

mbx_sts:　Get Mailbox Status

【パラメータ】

mbxid	MailboxIdentifier	メイルボックスID
pk_mbxs	Packet of MailboxStatus	メイルボックス状態を返すパケットの先頭アドレス

【リターンパラメータ】

なし

【解説】

mbxid で示されたメイルボックスの状態を読み出し、その結果を pk_mbxs 以下の領域に返す。

pk_mbxs に返される情報としては、次のようなものがある。

```
mbxatr      /* メイルボックス属性 */
wtskid      /* 待ち行列先頭のタスクID */
pk_msg      /* 次のメッセージのアドレス */
```

wtskid には、待ち行列の先頭のタスクのIDが返る。すなわち、属性が TA_TFIFO であれば、現在このメイルボックスを待っているタスクのうち、最も早く待ち状態になったタスクのIDが返る。また、属性が TA_TPRI であれば、現在このメイルボックスを待っているタスクのうち、最も優先度の高いタスクのIDが返る。待ちタスクの無い時は FALSE = 0 が返る。

pk_msg には、次に受信されるメッセージの先頭アドレスが返る。直後に rcv_msg を実行した場合には、同じ値が pk_msg として戻される。メッセージが無い時は、pk_msg は NADR(-1) となる。また、どんな場合でも、pk_msg=NADR と wtskid=FALSE の少なくても一方は必ず成り立つ。

対象となるメイルボックスは、既に生成されたものでなければならない。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦mbxid≦0）
E_ILADR	不正アドレス（pk_mbxsが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（mbxidのメイルボックスが存在しない）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でmbxid＜(-4)）

割込み管理機能

ITRONの割込みハンドラは、タスク独立部として扱われる。タスク独立部でも、タスク部や準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。しかし、タスク独立部では現在実行中のタスクを特定できないので、タスクの切り換え（ディスパッチング）は起らない。したがって、ディスパッチングが必要になっても、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。

また、暗黙に自タスクを指定するシステムコール（自タスクを待ち状態にするシステムコールを含む） は、タスク独立部から発行することができず、E_CTX のエラーになる。これ以外のシステムコールでも、インプリメントの制限によってタスク独立部から発行できない場合がある。

タスク独立部から発行可能なシステムコール

ITRON2では、性能面を考え、割り込み禁止時間を減らすため、タスク独立部から発行できる最低限のシステムコールを次の5個としている。

wup_tsk, ret_int, ret_tmr, psw_sts, get_tid

この5個のシステムコールについては、タスク独立部からも発行できることが保証されている。これ以外のシステムコールがタスク独立部から発行可能かどうかは、インプリメント依存である。

拡張SVCを使ったユーザ定義のシステムコールについても、タスク独立部から利用できるかどうかはインプリメント依存である。これは、タスク独立部から発行できるシステムコールが少ない場合に、いろいろなシステムコールを含んだ拡張SVCハンドラをタスク独立部から発行するのはあまり意味が無いためである。また、ITRONでは割込みハンドラの起動時にOSが介入しないため、割込みハンドラと拡張SVCハンドラや例外ハンドラのコンテキストが混在した場合でも、OS側でその区別を行うことができない。そのため、タスク独立部から拡張SVCを発行できるようにした場合、オーバーヘッドが大きくなったりする危険がある。

割込み処理とシステムコールの不可分性

一般のシステムコールでは、各システムコールの処理がユーザから見て不可分に行われるというのが原則である。これに対して、割込みハンドラから発行されたシステムコールでは、システムコールの不可分性を保証できない（保証しない方が明らかに性能が上がる）場合がある。

割込みハンドラから発行可能なシステムコールがインプリメント依存となっているので、割込みハンドラから発行されたシステムコールと、タスクから発行されたシステムコールとの間の不可分性に関しても、ある程度はインプリメント依存とならざるを得ない。これはやむを得ない問題だと考えられるので、割込みハンドラから発行したシステムコールに関しては、最低限発行可能なシステムコール（wup_tsk等）を除き、必ずしも不可分性を保証しなくても良いことにする。もちろん、どのような場合にシステムコールの不可分性が保てないかということについては、インプリメント毎に明らかにして頂く必要がある。

結局、割込みハンドラから発行可能なシステムコールについての考え方は次のようになる。

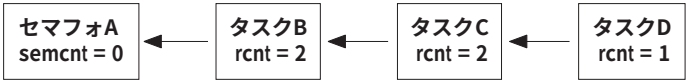
— wup_tsk, get_tid, psw_sts, ret_int, ret_tmr 以外のシステムコールの発行の可否はインプリメント依存である。

— 割込みハンドラから上記以外のシステムコールを発行した場合には、システムコールの不可分性を保てない場合がある。この問題については、割込み禁止時間を短くするためにやむを得ない面があるので、インプリメント毎に状況を明らかにして頂く。例えば、次の（例1）（例2）程度であれば、許容範囲内であろう。

（例1） TA_VAR, TA_CNT 属性のセマフォA に対して、タスクBがwai_sem(rcnt=2) を発行、タスクCが wai_sem(rcnt=2) を発行、タスクDがwai_sem(rcnt=1) を発行し、この順に待ち状態にあるものとする［図3.16］。セマフォA の現在カウント値 (semcnt) は 0 である。

ここで、タスクEがセマフォA に対して sig_sem (rcnt=3) を発行した場合、次のような動作をする。

(1) semcnt =3 とする。
(2) タスクBを待ち解除にして semcnt = 1 とする。



［図3.16］セマフォの動作例とシステムコールの不可分性

(3) 待ち行列の次のタスクはタスクCであるが、タスクCは rcnt = 2 なので、rcnt＞semcnt であり、待ち解除にはできない。一方、セマフォA の属性は TA_CNT なので、待ち行列の次のタスクであるタスクD をチェックすることになる。タスクDは rcnt = 1 なので、タスクD を待ち解除とし、semcnt = 0 とする。

さて、この状況で、上記の sig_sem (rnct=3) の前後に割込みが入り、割込みハンドラの中で sig_sem (rcnt=1) のシステムコールが発行されたものとする。この場合、次のような動作をすることが考えられる。

a. sig_sem(rcnt=3) が発行される直前に割込みが入り、そこで sig_sem (rcnt=1) が発行された場合。

割込みハンドラ中の sig_sem (rcnt = 1) により、タスクDが待ち解除となる。

その後 sig_sem (rcnt = 3) が実行された場合、タスクBは待ち解除となり、semcnt = 1 となるが、タスクC は rcnt＞semcnt なので待ち解除にできない。

最終的な状態は、タスクC (rcnt = 2) が待ち状態のままであり、semcn=1 となっている。

b. sig_sem (rcnt = 3) が発行された直後に割込みが入り、そこで sig_sem (rcnt = 1) が発行された場合。

上記の (1)〜(3) の動作の後でsig_sem (rcnt = 1) が実行され、semcnt = 1 となる。しかし、やはりタスクC は rcnt＞semcnt なので待ち解除にできない。

最終的な状態は、タスクC (rcnt = 2) が待ち状態のままであり、semcnt = 1 となっている。これは a. と同じである。

c. 上記の (1)〜(3) の処理時間が長くなるため、(2)と(3) の間で一旦割込みを許可することにした場合を考える。たまたま、(2)と(3) の間で割込み

ハンドラが起動され、そこで sig_sem(rcnt=1) が実行されると、次のような動作をする。

割込みハンドラ中の sig_sem (rcnt = 1) によって semcnt =2 となるので、（タスクCの rcnt）=semcnt となる。すなわち、(3) の処理を行なう代わりに、タスクC を待ち解除にできる。タスクC を待ち解除にすると、semcnt = 0 となる。

最終的な状態は、タスクD (rcnt = 1) が待ち状態のままであり、semcnt=0 となっている。これは、a. b. と異なった状態である。

すなわち、かなり細かいケースではあるが、割込み禁止時間を短くすると、システムコールの不可分性を保証できない場合がある。

（例2） tsk_sts をタスク独立部から発行できるシステムコールに含めた場合、このシステムコールの処理自体は特に難しくないが、そこで得られる情報の一貫性を正しく保とうとすると処理が複雑化する。

たとえば、タスクを SUSPEND 状態に入れる場合を考えると、カーネルの中で suscnt を増やし、tskstat を SUSPEND にするという処理が必要である。しかし、この2つの操作の間で割り込みが入り、そこでtsk_sts が実行された場合には、そこで返る情報に矛盾が生じることになる。

それを防ぐためには、2つの操作の間を割込み禁止にしなければならないが、それでは割込み禁止時間が増大し、性能低下の危険がある。

したがって、割込みハンドラから tsk_sts を発行することが可能であるとしても、返す情報の一貫性は保証できない場合がある。

割込みハンドラを定義する
def_int

def_int:　Define Interrupt Handler

【パラメータ】

intvec	InterruptVector	割込みベクトル番号
inhatr	InterruptHandlerAttribute	割込みアラームハンドラ属性
inthdr	InterruptHandlerAddress	割込みハンドラアドレス
imask	InterruptMask	割込み処理開始時の割込みマスク

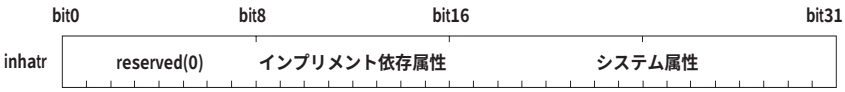
【リターンパラメータ】

なし

【解説】

割込みハンドラを定義し、使用可能にする。すなわち、inthdr で示される割込みハンドラのアドレスと、intvec で示される割込みベクトル番号との対応付けを行なう。

inhatr のサイズは標準で4バイトである。inhatr のフォーマットを［図3.17］に示す。このうち、最上位バイト（bit0〜bit7 あるいは 2^31〜2^24 のビット）は未使用であり、第二上位バイト（bit8〜bit15 あるいは 2^23〜2^16 のビット）はインプリメント依存の属性を表わし、下位ハーフワード（bit16〜bit31 あるいは 2^15〜2^0 のビット）はシステム属性を表わす。



［図3.17］inhatrのフォーマット

inhatr のシステム属性の部分では、次のような指定を行うことができる。

inhatr : = (TA_ASM ‖ TA_HLNG)

TA_ASM	ハンドラがアセンブラで書かれている
TA_HLNG	ハンドラが高級言語で書かれている

TA_HLNG の指定を行った場合には、ハンドラ起動時に直接 inthdr 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 inthdr のアドレスにジャンプする。

TA_HLNG の指定を行わない場合には、割込みハンドラの起動時にOSが介入しない。すなわち、ハードウエアの割込み処理メカニズムにより、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、割込みハンドラで使用するレジスタの退避と復帰は、ユーザが責任をもって行なう必要がある。

割込みハンドラは、ret_int 等のシステムコール、またはアセンブラの割込みリターン命令により終了する。

割込みハンドラ内のシステムコール発行によりタスクのディスパッチを生じた場合は、割込みハンドラが終了して ret_int 等のシステムコールによりOSにコントロールが戻ってくるまでディスパッチが遅らされる。したがって、割込みハンドラ内でディスパッチを生ずる可能性がある場合、一般には、アセンブラの割込みリターン命令ではなく、必ず ret_int システムコールを使って割込みハンドラから復帰する必要がある。

inthdr = NADR (-1) の指定により、前に定義されていた割込みハンドラの定義が解除され、未定義の状態になる。ただし、前に定義されていた割込みハンドラの定義解除を行わずに、直接新しいハンドラを定義しても、エラーにはならない。また、もともとハンドラが定義されていない状態（定義が解除された状態）で、inthdr=NADRの指定による定義解除の操作を行っても、エラーにはならない。後者の場合、このシステムコールは正常終了するが、割込みハンドラは未定義のままである。

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、待ちとなるシステムコールや、tskid = TSK_SELF によって自タスクの指定を意味するシステムコールは実行することはできない。

imask は、割込みハンドラ起動時におけるプロセッサの割込みマスクの値を指定するものである。ただし、intvec や imask の詳細な意味は、プロセ

ッサ依存となる。また、プロセッサによっては、imask の指定が意味を持たない場合がある。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足 ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSATR	予約属性（inhatrが不正）
E_ILADR	不正アドレス（inthdrが奇数、あるいは使用できない値）
E_VECN	不正ベクトル番号（intvecが不正）
E_IMS	不正IMASK（imaskが不正）

割込みハンドラから復帰する
ret_int

ret_int: Return from Interrupt Handler

【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

【解説】

割込みハンドラからのリターンの際に発行するシステムコールである。

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、このシステムコールが発行されて割込みハンドラを抜けるまで、ディスパッチが遅延させられる。

このシステムコールを呼び出した時の状態（スタック、レジスタ等）は、割込みハンドラへ入った時の状態と同じでなければいけない。すなわち、割込みハンドラで使用するレジスタの退避や復帰は、ユーザが行わなければならない。

また、レジスタの復帰をユーザが行なう関係で、このシステムコールでは機能コードを使用できない。したがって、このシステムコールの呼び出しの際、一般には、他のシステムコールと別ベクトルのトラップ命令を用いる。

【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には、メッセージバッファへのロギングや、コンソールへのエラーメッセージの表示などを行うのが望ましいが、詳細な動作はインプリメント依存となる。

E_CTX コンテキストエラー（タスク部、準タスク部から実行）

割込みマスクを変更する chg_ims

chg_ims: Change Interrupt Mask

【パラメータ】

imask	InterruptMask	割込みマスク

【リターンパラメータ】

なし

【解説】

プロセッサの割込みマスクを、imask で指定した値に変更する。

通常、このシステムコールは、ユーザタスクの一部を割込み禁止にするために用いられる。

chg_ims により割込みの禁止を行なっている間は、割込みハンドラ実行中と同じように、割込みマスクが解除されるまでディスパッチを遅延する。これは、割込みを禁止したままでディスパッチが行なわれると、割込み禁止の状態が中断したり、命令の実行に矛盾が起こったりするためである。ただし、割込み禁止状態のタスクは、ディスパッチを遅延した状態にはなっているが、タスク独立部の扱いにはしない。「ディスパッチ遅延区間」と「タスク独立部」は別のものとして考える。たとえば、割込み禁止状態のタスクが get_tid を実行した場合、0ではなくそのタスクのタスクIDが返る。

【エラーコード (ercd)】

E_OK	正常終了
E_IMS	不正IMASK

例外管理機能

例外の種類と例外クラス

ITRON2では、次のような例外事象に対して、その例外ハンドラを定義し、適切な処理を行なうことができる。例外ハンドラは、例外が発生した時に非同期に実行されるプログラムであり、例外発生前と同じリングレベルで実行される。例外ハンドラには、タスク終了時に資源解放のために実行される終了ハンドラも含まれている。

― 終了要求
他タスクが自タスクに対して ter_tsk あるいは ras_ter を発行することにより生じる例外が終了要求である。
終了要求に対する例外ハンドラを、特に終了ハンドラと呼ぶ。
タスクに対する終了ハンドラは、def_ext により定義される。ハンドラ内でタスクが終了するため、ハンドラから戻ることはない。

― 強制例外（タスク外から）
他タスクが自タスクに対して ras_fex システムコールを発行することによって生じる例外が強制例外である。
タスクに対する強制例外ハンドラは、def_fex により定義され、ret_exc によりハンドラから戻る。
なお、強制例外の機能は拡張仕様である。

― CPU例外
機械命令レベルのエラー（ゼロ除算など）が発生した場合に生じる例外がCPU例外である。
タスクに対する CPU例外ハンドラは、def_cex により定義され、ret_exc によりハンドラから戻る。CPU例外は、マスクすることができない。

― システムコール例外（拡張SVCによるエラーを含む）
システムコール実行に伴うエラー（パラメータエラーなど）が発生した場合に生じる例外がシステムコール例外である。

タスクに対するシステムコール例外ハンドラは、def_sex により定義され、ret_exc によりハンドラから戻る。

なお、システムコール例外の場合は、CPU例外と違って、システムコールのリターン値として個別にエラーチェックが可能であり、例外ハンドラの必要性が少ない。また、高級言語への対応やハンドラ起動時に渡す情報が CPU例外と異なっている。CPU 例外とシステムコール例外を分けるのは、このためである。

上記の分類は、次に述べる例外クラスの分類に対応したものである。例外と終了要求は、合わせて次のようにクラス分けされる。CPU例外を除き、それぞれの例外クラスに対して例外ハンドラの起動をマスク(あるいは遅延)させることができる。

EC_TER　　終了要求クラス

他タスクの発行した ter_tsk, ras_ter により発生する。

これに対するハンドラ（終了ハンドラ）は優先度を上げて実行される。

この例外がタスク実行中に発生した場合の例外ハンドラは def_ext で定義され、この例外が拡張SVC中で発生した場合の例外ハンドラは sdef_ext で定義される。

原則として元のプログラムに戻らないが、拡張SVCハンドラ中で起動された終了ハンドラでは、戻る必要が生じる場合もある。

他の例外とは異なり、この終了要求は、現在実行中のハンドラに対する要求ではなく、タスク全体（そのタスクとタスクから起動されて実行中のハンドラ全部）に対する要求となる。例えば、タスクAが拡張SVCハンドラBを呼び、それを準タスクとして実行中に終了要求が受け付けられた場合、まず拡張SVCハンドラBに対する終了ハンドラが実行され、拡張SVCハンドラからタスクに戻ってから、タスクに対する終了ハンドラが実行されることになる。すなわち、終了要求は、現在実行中の拡張SVCハンドラに影響を与えるだけではなく、それを呼んだタスクにも伝達されていく。

EC_FEX　　強制例外クラス

他タスクの発行した ras_fex により発生する。

強制例外ハンドラは、タスクに対するもののみが存在する。拡張SVCハンドラに対する強制例外ハンドラは無い。拡張SVCハンドラの実行中に

強制例外を受けた場合、拡張SVCハンドラの中では強制例外ハンドラの起動が抑制されており、タスクに戻った時にタスクに対する強制例外ハンドラが起動される。これは、強制例外がタスクに対して要求されるものであり、ras_fex を実行した時にたまたまそのタスクが拡張SVCを実行中だったとしても、それは直接関係しないという理由による。
タスクに対する強制例外ハンドラは、def_fex によって定義される。

EC_CEX　　CPU例外クラス
ゼロ除算など、プロセッサによる自タスクのハードウエア的な例外により発生する。
この例外がタスク実行中に発生した場合の例外ハンドラは def_cex で定義され、この例外が拡張SVC中で発生した場合の例外ハンドラは sdef_cex で定義される。

※ 以下はシステムコール実行中のエラーに対する例外（システムコール例外）を表わす例外クラスである。これらの例外がタスク実行中に発生した場合の例外ハンドラは def_sex で定義され、この例外が拡張SVC中で発生した場合の例外ハンドラは sdef_sex で定義される。（EC_SYS を除く）

EC_SYS　　システムエラー例外クラス
原因不明のエラー、システム全体に影響する重大なエラーなど
システムコールのエラーのうち、このエラーが発生した場合に限り、idef_sex で定義されたタスク独立部に対する例外ハンドラが起動される。

EC_NOMEM　　メモリ不足エラー例外クラス

EC_RSV　　予約機能エラー例外クラス

EC_PAR　　パラメータエラー例外クラス
主に、エラーがスタティックに（実行前に）判定できるものを含む。

EC_OBJ 不正オブジェクト指定例外クラス

EC_ACV　　アクセス権違反例外クラス

EC_CTX　　コンテキストエラー例外クラス

EC_EXEC　　実行時エラー例外クラス

EC_EXCMP　　実行終了時エラー例外クラス

EC_RLWAI　　待ち状態強制解除例外クラス

EC_RSLT　　処理実行結果を示す例外クラス

これらのニモニックは例外クラスを 0〜31 の番号で示したものであるが、例外ハンドラ起動のマスク等を指定するためには、これらの例外クラスをビット対応の表現にしたものを利用する。例外クラスをビット対応の表現にしたものは ECM_xxxx のニモニックで表わされ、EC_xxxx との関係は次のようになる。

ECM_xxxx = (H'80000000 ≫ EC_xxxx)
※ '≫' は論理右シフトを示す。

例外ハンドラの独立性と例外ハンドラ定義環境

例外の処理を行う例外ハンドラは、その例外を発生する可能性のあるプログラムと組にして提供されるべきである。例えば、入出力処理を行う拡張SVCハンドラのプログラムがあった場合、その拡張SVCハンドラに対する例外ハンドラを作成するためには、拡張SVCハンドラの内部構成がよく分かっていなければならない。しかし、拡張SVCハンドラのユーザの側では、単にそれを使って入出力ができれば良い。ユーザの側では、拡張SVCハンドラの内部仕様やインプリメント方法は知らないのが普通であり、また知る必要もないので、拡張SVCハンドラに対する例外ハンドラを作成することはできない。したがって、拡張SVCハンドラに対する例外ハンドラは、拡張SVCハンドラの作成と同時に行われ、同時に提供されなければならない。同じように、拡張SVCハンドラを利用する側のタスクに対する例外ハンドラも、タスクの内部仕様やインプリメント方法を知らないと作成できないのが普通である。ところが、拡張SVCハンドラとタスクは全く独立して作成されることが多いので、それぞれに対する例外ハンドラも独立して定義できなければならない。また、拡張SVCハンドラ同士であっても、機能コードが違えば、全く独立して作成されているかもしれない。そのため、異なる機能コードの拡張SVCハンドラに対する例外ハンドラも独立して定義できなければならない。もし、システム全体で一つの例外ハンドラしか定義できないものとすると、入出力処理の拡張SVCで例外ハンドラを

使ってしまった場合には、他のタスクや拡張SVCハンドラでは例外ハンドラが使えない（入出力処理の拡張SVCに対する例外ハンドラが動いても無意味）ということになってしまう。

以上のような要求に対応するため、ITRON2では、例外発生時のコンテキスト（タスク実行中、拡張SVCハンドラ実行中など）によって、どの例外ハンドラが起動されるかを切り換えるようになっている。具体的には、次のような例外ハンドラを独立して定義することができる。

A. タスクに対する例外ハンドラ

A-e. タスクに対する終了ハンドラ　（def_ext により定義）
A-f. タスクに対する強制例外ハンドラ　（def_fex により定義）
A-c. タスクに対するCPU例外ハンドラ　（def_cex により定義）
A-s. タスクに対するシステムコール例外ハンドラ　（def_sex により定義）

上記のハンドラは、各タスク毎に独立して定義することができる。また、各タスクに対するハンドラ以外に、全タスクで共通に使用されるハンドラを定義することができる。各タスクに対するハンドラが未定義の場合は、全タスク共通のハンドラが起動される。

B. 拡張SVCに対する例外ハンドラ

B-e. 拡張SVCに対する終了ハンドラ　(sdef_ext により定義)
B-c. 拡張SVCに対するCPU例外ハンドラ　(sdef_cex により定義)
B-s. 拡張SVCに対するシステムコール例外ハンドラ　(sdef_sex により定義)

上記のハンドラは、各拡張SVCハンドラ（機能コード）毎に独立して定義することができる。また、各機能コードに対するハンドラ以外に、全拡張SVCハンドラ（全機能コード）で共通に使用されるハンドラを定義することができる。各機能コードに対するハンドラが未定義の場合は、全機能コード共通のハンドラが起動される。

拡張SVCハンドラ実行中は強制例外の起動が抑制されるため、拡張SVCハンドラに対する強制例外ハンドラはない。

C. タスク独立部やOS内部（カーネル）に対する例外ハンドラ

C-e. タスク独立部に対する終了ハンドラ　（idef_ext により定義）
C-c. タスク独立部に対するCPU例外ハンドラ　（idef_cex により定義）
C-s. タスク独立部に対するシステムコール例外ハンドラ（idef_sex により定義）

例外ハンドラを定義するシステムコールがこのように分離されているのは、パラメータ等が異なるためである。拡張SVCハンドラに対する例外ハンドラの定義機能は、システム操作機能に分類されている。

上記のように、「現在例外が発生した場合にどこで定義された例外ハンドラが起動されるか」を区別する環境を例外ハンドラ定義環境と呼ぶ。例外ハンドラ定義環境は、タスク毎に異なり、また、タスクから呼ばれた拡張SVC毎にも異なっている。逆に、タスクや拡張SVCから例外ハンドラが起動された場合には、例外ハンドラ定義環境は変わらない。また、異なるタスクから呼ばれた同一の拡張SVCでも、例外ハンドラ定義環境は同じになる。例えば、次のような状況では、bとc、fとd の例外ハンドラ定義環境が同じであり、a, b, d, e の例外ハンドラ定義環境はそれぞれ異なっている。

a. タスクA
b. タスクAから呼ばれた拡張SVCのB
c. タスクAから呼ばれた拡張SVCのBの中で起動された例外ハンドラC
d. 例外ハンドラCの中から呼ばれた拡張SVCのD
e. タスクE
f. タスクBから呼ばれた拡張SVCのD

※ この場合、a, b=c, d=f, e の異なる 4 つの例外ハンドラ定義環境が存在する。

例外ハンドラを起動する際は、例外ハンドラ定義環境が変わらないため、タスク中で終了要求を受け付けた場合と、タスクに対するCPU例外ハンドラの中で終了要求を受け付けた場合では、同じ終了ハンドラが起動されることになる。特に、例外の種類が同じである場合、例えば、システムコール例外のハンドラの中で再度システムコール例外が発生したような場合には、同じハンドラが再帰的に呼び出されることになる。ただし、例外ハンドラの起動により、例外ハンドラの起動を許可するかどうかを示す例外マスク環境が別のものになるため、起動されるべき例外ハンドラは同じものであるが、実際には例外が発生しても例外ハンドラが起動されない場合がある。例外マスク環境については次の項で説明する。

例外マスク環境

例外ハンドラの実行中は、同じ種類の例外が発生しても例外ハンドラの再帰的な起動を行わないのが普通である。すなわち、例外ハンドラ実行中のみ、一時的に例外ハンドラ起動の有無を変更したい場合がある。また、

例外ハンドラは、各タスクや個々の拡張SVCに対して別々のものが定義される。したがって、拡張SVCの要求タスクが例外ハンドラの起動を許可していても、拡張SVCの内部で一時的に例外をマスクしたい場合や、その逆の場合がある。したがって、例外ハンドラ起動の有無を決める例外マスクは、こういった状況に応じて自動的に切り換わるものが望ましい。

そこで、「例外ハンドラ起動の有無」を表わす環境を「例外マスク環境」と呼ぶことにし、例外マスク環境は各タスクや各タスクから呼ばれた各ハンドラ（拡張SVCハンドラ、例外ハンドラ）に対して独立して存在するものとする。したがって、例外マスク環境は、タスク毎に異なり、タスクから呼ばれた拡張SVC毎にも異なり、タスクや拡張SVCから呼ばれた例外ハンドラ毎にも異なっている。例えば、次のような状況では、a, b, c, d, e, fの例外マスク環境はすべて異なったものになる。

a. タスクA
b. タスクAから呼ばれた拡張SVCのB
c. タスクAから呼ばれた拡張SVCのBの中で起動された例外ハンドラC
d. 例外ハンドラCの中から呼ばれた拡張SVCのD
e. タスクE
f. タスクBから呼ばれた拡張SVCのD

※ この場合、a, b, c, d, e, f の異なる 6 つの例外マスク環境が存在する。

例外マスクは、例外マスク環境毎に、emsptn (Exception Mask Pattern) というビットパターンによって表現される。emsptn は、例外クラス別に例外ハンドラ起動の禁止 / 許可を表わしたものである。

なお、ここでは、タスク部や準タスク部に対する例外マスク環境についての説明を行っている。タスク独立部に対する例外マスク環境や emsptn も考えられるが、それについては後で説明を行う。

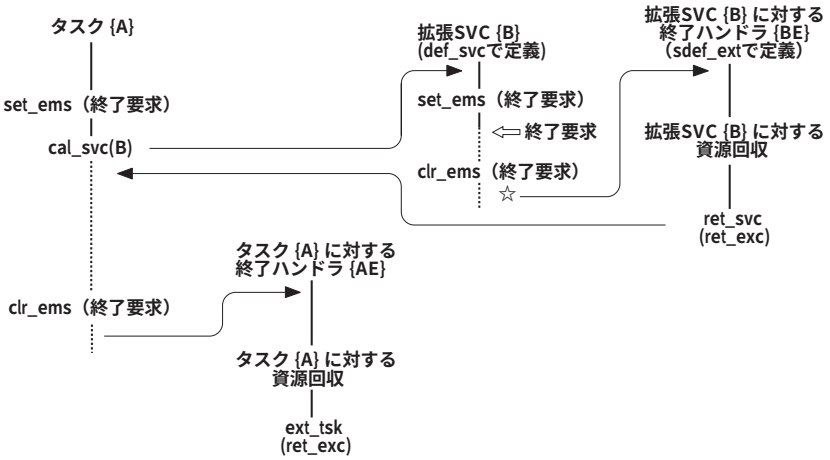
例外（終了処理要求、強制例外を含む）が発生した場合、発生した例外の例外クラスが、現在の例外処理環境の emsptn によってマスクされていれば、例外ハンドラは起動されない。この時、発生した例外がシステムコール例外であれば、例外ハンドラ起動要求はキャンセルされる。この場合は、エラーコードを参照することによってのみ、エラーがあったということを知ることができる。

一方、発生した例外が終了処理要求や強制例外であり、emsptn によって

それがマスクされていた場合は、例外ハンドラ起動要求がキャンセルされるのではなく、マスクが解除されるまで例外ハンドラの起動が遅延させられる。この時、タスクが例外ハンドラ起動要求を覚えておく必要があるが、そのために次の項の「例外ペンディング環境」が利用される。

現在の例外マスク環境、すなわち自タスクや自ハンドラの emsptn は、set_ems, clr_ems を使って動的に変更することができる。clr_ems では、ペンディングになっていた終了処理要求、強制例外の有無がチェックされ、その結果によって、遅らせられていた終了ハンドラや強制例外ハンドラの起動が行なわれる。

拡張SVCハンドラ実行中に終了要求を受け付け、拡張SVCハンドラに対する終了ハンドラが起動された場合の動作例を［図3.18］に示す。



＊'┆' は、例外マスクの機能のために終了ハンドラの起動が遅延させられている区間を示す。
拡張SVCハンドラに対する終了ハンドラ、タスクに対する終了ハンドラとも、例外マスクによって起動が遅らされている。

＊要求タスクが例外をマスクしていても、そこから呼ばれた拡張SVCでは例外マスクを解除したい場合がある。

＊拡張SVCハンドラに対する終了ハンドラ {BE} で ret_exc が実行された場合には、ret_svc が実行された場合と同様に、拡張SVCを呼んだ側のコンテキスト（終了ハンドラを起動したコンテキスト＝☆ではない）に戻る。この時の s_ercd は E_TER になる。

［図3.18］拡張SVCハンドラに対する終了ハンドラ

この場合、タスク{A}からみると、終了要求がマスクされている間に拡張SVC{B}に対する終了ハンドラ{BE}が動いていることになる。しかし、{BE}　で操作するオブジェクトは、拡張SVC{B}で使用していたオブジェクトに限られ、タスク{A}で操作していたオブジェクトを直接操作することは無いはずである。また、たとえ操作しても、矛盾の無い形に戻すようにプログラミングすることが可能なはずである。したがって、タスク{A}が終了要求をマスクしていても、これはタスク{A}に対する終了ハンドラ{AE}の起動の抑止を意味するものであり、拡張SVC{B}に対する終了ハンドラ{BE}の起動を抑止する必然性は無い。タスクに対する終了ハンドラ起動の例外マスクと、拡張SVCに対する終了ハンドラ起動の例外マスクとを別々に指定できるのは、こういった理由による。

また、上記のように、拡張SVCに対する終了ハンドラ{BE}とタスクに対する終了ハンドラ{AE}は、連続して動くとは限らない。その意味では、「拡張SVCに対する終了ハンドラ」を「拡張SVC中断ハンドラ」と考えることもできる。拡張SVCに対する終了ハンドラ{BE}の最後では、「終了要求により拡張SVCが中断した」といった意味のエラーを返すのが一般的である。

CPU例外はマスクすることができないため、emsptn のうち、CPU例外の例外クラスに対するビットは常にクリアされている。これは、タスクに対する例外マスク環境、拡張SVCハンドラに対する例外マスク環境、例外ハンドラに対する例外マスク環境とも同様である。また、拡張SVCハンドラ実行中は強制例外ハンドラの起動が遅延させられるため、拡張SVCハンドラに対する例外マスク環境、および拡張SVCハンドラから起動された例外ハンドラに対する例外マスク環境では、強制例外の例外クラスに対する emsptn のビットは常にセットされている。

タスク生成直後の例外マスク環境の emsptn の値は次のようになる。

終了処理(EC_TER)	可
強制例外(EC_FEX)	マスク
CPU例外(EC_CEX)	可（マスクをセットできない）
その他（システムコール例外）	全部マスク

すなわち、

emsptn = ~(ECM_CEX ¦ ECM_TER ¦ ECM_ABO ¦ ECM_SUS)

※ '&' はビット対応の論理積、'¦' はビット対応の論理和、
'~'はビット対応の否定

と書くことができる。タスク起動時に終了要求をマスクしておきたい場合は、cre_tsk の実行後、sta_tsk の実行前に set_ems (setptn = ECM_TER) を実行しておけば良い。なお、タスク終了時には emsptn の値がリセットされ、タスク生成直後と同じ値に戻る。

拡張SVCハンドラ起動時の例外マスク環境の emsptn の値は次のようになる。

終了処理(EC_TER)	マスク
強制例外(EC_FEX)	マスク（マスクをクリアできない）
CPU例外(EC_CEX)	可（マスクをセットできない）
その他（システムコール例外）	全部マスク

すなわち、

emsptn = ~ (ECM_CEX ¦ ECM_ABO)

※ '&' はビット対応の論理積、'¦' はビット対応の論理和、
'~'はビット対応の否定

と書くことができる。拡張SVCハンドラ起動時の emsptn の値は、ハンドラ起動前の emsptn の値（拡張SVCハンドラを呼んだ側の emsptn の値）には関係しない。

タスク部、準タスク部で例外（終了）ハンドラが起動された時の emsptn の値は、ハンドラ起動前の例外マスク環境の emsptn の値に、その例外（終了）ハンドラの起動要因となった例外の例外クラスのマスクを加えた（論理和をとった）ものである。たとえば、システムコール例外ハンドラ起動時の emsptn は、システムコール例外発生前の emsptn と、発生したシステムコール例外の例外クラスビットパターンとの論理和になる。このような仕様になっているのは、同じ種類の例外の二重起動を禁止するためである。違う種類の例外については、例外ハンドラの二重起動は可能であり、同じ種類の例外については、例外ハンドラの二重起動は原則として禁止している。ただし、終了ハンドラの起動時には、終了要求に対するマスクだけではなく、強制例外に対するマスクも自動的にセットされる。これは、終了要求を受け付けた場合、強制例外の処理を行うことは無意味になることが多いためである。したがって、終了ハンドラ起動時の emsptn は、終了処理要求の受け付け前の emsptn と (ECM_TER¦ ECM_FEX) との論理和になる。一方、CPU例外はマスクすることができないので、CPU例外ハンドラ起動時の emsptn は、CPU例外発生前の emsptn と同じである。

ハンドラの起動では、元の例外マスク環境の emsptn は変化しない。また、ハンドラから戻る際には、emsptn も元の例外マスク環境のものに戻る。ハンドラから戻る際のハンドラ側の emsptn の値は、単に捨てられる。

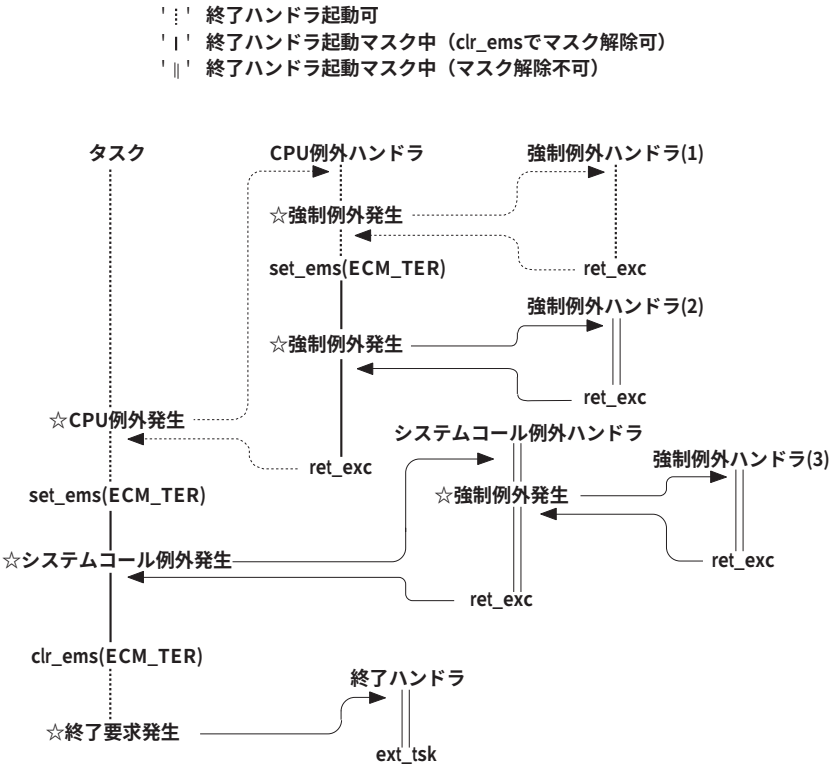
例外（終了）ハンドラ起動時に終了処理要求や強制例外（起動が遅延させられる例外）に対する例外マスクがセットされていた場合には、例外（終了）ハンドラの中でその例外マスクを解除することはできない。例えば、終了ハンドラの起動をマスクしている間にCPU例外が発生した場合、そこで起動されたCPU例外ハンドラの中では、終了処理要求に対する例外マスクをクリアすることができない。これは、CPU例外ハンドラの実行中も終了ハンドラに対するクリティカルセクションが続いており、終了ハンドラを起動すると、CPU例外発生直前に行われていた処理に関して矛盾を生じるためである。ここで、もしCPU例外ハンドラ実行中に ter_tsk による終了要求を受けた場合には、CPU例外ハンドラから元のコンテキストに戻り、終了処理要求に対する例外マスクが解除された後で終了ハンドラが起動される。

また、終了ハンドラの起動時には、自動的に終了ハンドラの再起動に対するマスク (ECM_TER) がセットされるが、終了ハンドラの中でこのマスクを解除することはできない。これに対して、終了ハンドラや強制例外ハンドラの起動時に自動的にセットされる強制例外のマスク (ECM_FEX) は、終了ハンドラや強制例外ハンドラの中で、必要に応じて解除することが可能である。

例外ハンドラの中で例外マスクをクリアできないケースについて、具体的な動作例を［図3.19］に示す。

一方、拡張SVCハンドラに関しては、終了ハンドラの起動がマスクされている間に呼び出された拡張SVCハンドラの中でも、終了処理要求に対する例外マスクを解除することが可能である。拡張SVCハンドラ実行中に終了要求を受けた場合は、拡張SVCに対する終了ハンドラが起動され、タスクに対する終了ハンドラが起動されるわけではないので、クリティカルセクションの問題は発生しない。この時、タスクに対する終了ハンドラは、拡張SVCハンドラあるいは拡張SVCに対する終了ハンドラから元のタスクに戻り、終了処理要求に対する例外マスクが解除された後で起動される。

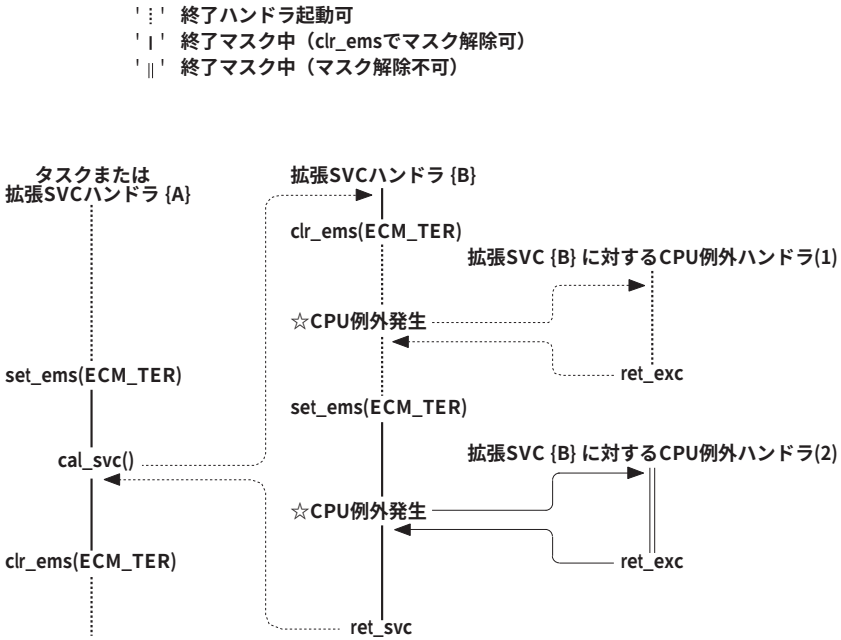
また、拡張SVCハンドラがネストしていた場合も同様である。すなわち、拡張SVC{A}が終了ハンドラの起動をマスクしている間に拡張SVC{B}を呼び出した場合、拡張SVC{B}の中で終了処理要求に対する例外マスクを解除



＊強制例外ハンドラ(1) (2) (3) は、途中で再定義が行われない限り、
プログラムとしては同じものが動く。

［図3.19］例外マスクをクリアできないケース

することが可能である。拡張SVC{B}の実行中に終了要求を受けた場合は、拡張SVC{B}に対する終了ハンドラが起動され、拡張SVC{A}に対する終了ハンドラが起動されるわけではないので、やはりクリティカルセクションの問題は発生しない。この時、拡張SVC{A}に対する終了ハンドラは、拡張SVC{B}あるいは拡張SVC{B}に対する終了ハンドラから拡張SVC{A}に戻り、終了処理要求に対する拡張SVC{A}の例外マスクが解除された後で起動される。



［図3.20］拡張SVCハンドラと例外マスク

ただし、拡張SVCに対する例外（終了）ハンドラの起動時に、終了処理要求や強制例外に対する拡張SVCの例外マスクがセットされていた場合には、拡張SVCに対する例外（終了）ハンドラの中で、それらの例外マスクを解除することはできない。これは、タスクの場合と同様である。

拡張SVCハンドラの起動と終了要求に対する例外マスクとの関係について、具体的な動作例を［図3.20］に示す。

例外ペンディング環境

拡張SVC処理中にシステムコール例外や CPU例外が発生した場合は、拡張SVCに対する例外ハンドラのみが起動されれば良く、要求タスクに影響を与える必要はない。エラーの原因やそれに対する処理が、拡張SVCだけで閉じたものになるからである。一方、拡張SVC処理中に強制例外が発生した場合は、拡張SVCからタスクに戻るまで、その要求を保持しておく必要がある。また、拡張SVC処理中に終了処理要求が発生した場合は、まず拡張SVC処理に対する終了ハンドラを起動し、そこから戻った後に要求タスクの終了ハンドラも起動する必要がある。これは、拡張SVCハンドラがネストした場合にも同様である。

そこで、「終了要求や強制例外要求の有無」を表わす環境を「例外ペンディング環境」と呼ぶことにし、この中の epndptn (Pending ExceptionClass Pattern) というビットパターンによってペンディング中の例外を表現することにする。epndptn で対象となる例外クラスは終了処理要求と強制例外に対する例外クラスのみであり、ECM_TER と ECM_FEX の2ビットで表現される。上記のような理由により、例外ペンディング環境はハンドラ毎に存在するのではなく、各タスクに対してただ一つのみ存在する。例えば、次のような状況では、a, b, c, d の例外ペンディング環境がすべて同じであり、a, e, f の例外ペンディング環境は異なったものになる。

a. タスクA
b. タスクAから呼ばれた拡張SVCのB
c. タスクAから呼ばれた拡張SVCのBの中で起動された例外ハンドラC
d. 例外ハンドラCの中から呼ばれた拡張SVCのD
e. タスクE
f. タスクBから呼ばれた拡張SVCのD
※ この場合、a=b=c=d, e, f の異なる 3 つの例外ペンディング環境が存在する。

ECM_TER に対する epndptn は、ter_tsk や ras_ter により一度セットされると、タスクが終了するまでクリアされない。一方、ECM_FEX に対する epndptn は、ras_fex によりセットされた後、強制例外に対するマスクが解除されて強制例外ハンドラが起動されると、クリアされてしまう。これは、両者の性質が次のように異なるためである。

終了ハンドラ (ECM_TER):

— 終了要求受付時にネストした状態の拡張SVCハンドラを実行している場合は、実行中の各レベルの拡張SVCハンドラに対する終了ハンドラと、タスクレベルに対する終了ハンドラをすべて起動する必要がある。つまり、1回の ter_tsk, ras_ter に対して起動されるハンドラは複数であり、1つの終了ハンドラを起動しても、要求をクリアすることができない。

— 終了要求が一度受け付けられた後は、最終的にタスクを終了するしかあり得ないので、終了要求をクリアする必要はない。

強制例外ハンドラ (ECM_FEX):

— 要求が起きる度に何度でも起動される。強制例外ハンドラからリターンした後は、完全に元の状態に戻る。したがって、強制例外ハンドラを起動した後は要求をクリアしておかないと、次の強制例外を受け付けられる体制にならない。

— 拡張SVCハンドラに対する強制例外ハンドラが無いため、1回の ras_fex に対して起動されるハンドラはタスクレベルに対する強制例外ハンドラのみである。つまり、1回発行された ras_fex に対して、起動される強制例外ハンドラは1つだけである。そのため、強制例外ハンドラ起動直後に要求をクリアしても、問題はない。

タスク起動時の epndptn の値は、ECM_TER と ECM_FEX とも要求無しになっている。epndptn は DORMANT 状態になる毎にクリアされる。

epndptn は、各タスクおよび各例外（終了と強制例外）に対して1ビットずつしか用意されていないため、終了要求同士、あるいは強制例外起動要求同士の複数キューイングはできない。たとえば、同じタスクを対象とした ter_tsk が2回発行されても、終了ハンドラは1回しか起動されない。ただし、終了ハンドラが受け取る例外コード exccd は、それぞれの ter_tsk で渡された例外コードの論理和をとったものになる。また、この場合、先の強制例外（終了要求）を起こした ter_tsk, ras_ter, ras_fex にも、後の強制例外（終了要求）を起こしたter_tsk, ras_ter, ras_fex にもエラーは返らない。

例外関係の各機能の処理概要

以下に、例外関係のシステムコール等の処理方法を述べ、仕様の確認およびインプリメントのヒントとする。なお、以下の説明はあくまでも処理の概要を述べたものであり、詳細な点や例外が多重に発生した場合の処理、SUSPEND 要求の遅延などは考慮していない。

― ras_fex の処理

```
if（対象タスク/ハンドラの emsptn ->ECM_FEXがクリア
   and. 対象タスクが待ち状態でない）
then
        対象タスクの強制例外ハンドラを起動;
else
        対象タスクのepndptn->ECM_FEXをセット;
endif
```

― ras_ter の処理

```
対象タスクのepndptn->ECM_TERをセット;
if（対象タスク/ハンドラの emsptn->ECM_TERがクリア
   .and. 対象タスクが待ち状態でない） then
        対象タスク/ハンドラの終了ハンドラを起動;
endif
```

― ter_tsk の処理

```
対象タスクのepndptn->ECM_TERをセット;
if（対象タスク/ハンドラのemsptn->ECM_TERがクリア）then
        対象タスク/ハンドラの終了ハンドラを起動;
            ※ 待ち状態の時もそうでない時も含む
else if（対象タスクが待ち状態）then
        待ち状態を解除しE_RLWAIを返す;
endif
```

― 待ち解除時の処理

```
if（対象タスクの epndptn->ECM_TERがセット
  .and. 対象タスク/ハンドラのemsptn->ECM_TERがクリア）
then
        対象タスク/ハンドラの終了ハンドラを起動;
```

```
endif
if（対象タスクのepndptn->ECM_FEXがセット
  .and. 対象タスク/ハンドラのemsptn->ECM_FEXがクリア）
then
        対象タスクのepndptn->ECM_FEXをクリア;
        対象タスクの強制例外ハンドラを起動;
endif
```

— ret_svc の処理

```
if（epndptn->ECM_TERがセット
  .and. 戻り先の環境のemsptn->ECM_TERがクリア）
then
        戻り先の環境の終了ハンドラを起動;
endif
if （epndptn->ECM_FEXがセット
  .and. 戻り先の環境のemsptn->ECM_FEXがクリア）
then
        epndptn->ECM_FEXをクリア;
        戻り先の環境の強制例外ハンドラを起動;
endif
```

— ret_exc の処理

```
/* 例外マスク環境を戻す。後の処理は clr_ems と同じ */
```

— clr_ems の処理

```
パラメータに応じてemsptnを変更;
if （epndptn->ECM_TERがセット .and. emsptn->ECM_TERがクリア）
then
        終了ハンドラを起動;
endif
if（epndptn->ECM_FEXがセット .and. emsptn->ECM_FEXがクリア）
then
        epndptn->ECM_FEXをクリア;
        強制例外ハンドラを起動;
endif
```

なお、拡張SVCハンドラの実行中は強制例外ハンドラが起動されない。しかし、emsptn->ECM_FEX がクリアされた状態であれば、このタスクは拡張SVC以外の部分を実行中 (ring1〜3) であることが保証されるので、拡張SVCハンドラ実行中かどうかというチェックを別に行う必要はない。

例外ハンドラが未定義の場合の動作

例外ハンドラは、個々のタスクあるいは拡張SVC毎に定義することが可能である。しかし、こういった個別の例外ハンドラを定義したくない場合には、タスク間あるいは拡張SVC間で共通に使用される例外ハンドラを定義することができる。タスクあるいは拡張SVCに対する個別の例外ハンドラが未定義で、かつタスク間あるいは拡張SVC間で共通の例外ハンドラが定義されていれば、例外発生時にそれが起動される。

また、個別の例外ハンドラも共通の例外ハンドラも未定義で、しかも対応する例外クラスのマスクがセットされていない場合には、システムデフォルトの例外ハンドラを起動すると考える。システムデフォルトの例外ハンドラの中では、次のような処理を行うことを推奨しているが、詳細はインプリメント依存である。互換性の高いプログラムを書くためには、システムデフォルトの例外ハンドラの機能を利用せず、例外ハンドラの起動をマスクしておくか、例外ハンドラのプログラムをきちんと定義しておくのが望ましい。

例外マスクがクリアされており、かつ対応する例外ハンドラが未定義であった場合において、例外発生時の実行環境や発生した例外の種類とそれに対する動作（推奨仕様）との関係は次のようになる。強制例外の場合を除けば、例外ハンドラ未定義の場合は終了要求に置換する（終了ハンドラ未定義の場合は即座にタスクやハンドラを終了する）というのが原則である。

また、発生した例外の種類、例外マスクの有無、例外ハンドラ定義の有無と、その時の動作との関係をまとめて整理すると、次のようになる。

システムコール例外ハンドラ、あるいはCPU例外ハンドラが未定義で例外が発生し、それが終了要求に置き換わった場合は、システムデフォルトの例外ハンドラから abo_tsk を発行すると考える。したがって、この場合は終了マスクが無視される。この時の例外コード（abo_tskのパラメータである例外コードexccdに相当する値）としては、次の値が使用される。

```
CPU例外の場合                ECM_CEX   H'10000000
システムコール例外の場合 ECM_SEX   H'08000000  /* ECM_SYS と同じ値 */
```

実行環境 →	タスク実行中 タスクから呼ばれた例外ハンドラ実行中	拡張SVCハンドラ実行中 拡張SVCから呼ばれた例外ハンドラ実行中
終了要求	タスク終了 [ext_tsk]	拡張SVCハンドラ終了 [ret_svc]
強制例外	何もしない	何もしない
CPU例外	終了要求に置換 [abo_tsk]	終了要求に置換 [abo_tsk]
システムコール例外	終了要求に置換 [abo_tsk]	終了要求に置換 [abo_tsk]

＊[～] 内は、例外ハンドラの中で [～] 内のシステムコールを
　発行するのと等価の動作をするということを示す。
＊例外発生時のデバッガの起動については、別の方法で指定する。

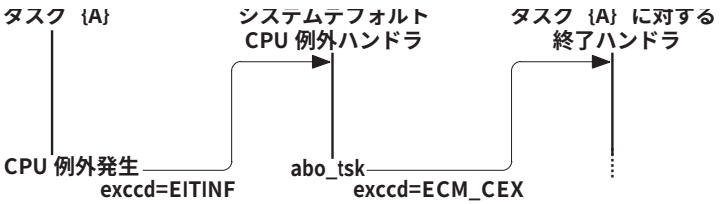
［表3.3］例外ハンドラ未定義の場合の動作

マスク → 例外ハンドラ → 実行環境 →	有り 定義／未定義 タスク内／ハンドラ内	無し 定義 タスク内／ハンドラ内	無し 未定義 タスク内	無し 未定義 拡張SVCハンドラ内
終了要求	ペンディング	終了ハンドラ起動	タスク終了	拡張SVCハンドラ終了
強制例外	ペンディング	強制例外ハンドラ起動	何もしない	何もしない
CPU例外	（マスク不可）	CPU例外ハンドラ起動	終了要求に置換	終了要求に置換
システムコール例外	キャンセル	システムコール例外ハンドラ起動	終了要求に置換	終了要求に置換

［表3.4］例外発生時の動作と例外マスクや例外ハンドラ定義状態との関係

結局、システムデフォルトの例外ハンドラは次のように書くことができる。

```
/* 強制例外に対するデフォルトのハンドラ */
   EXCHDR fex_hdr( T_EXC *pk_exc, T_REGS *pk_regs, T_EIT *pk_eit ){
         /* 何もしない */
         ret_exc( );
   }
/* CPU例外に対するデフォルトのハンドラ */
   EXCHDR cex_hdr( T_EXC *pk_exc, T_REGS *pk_regs, T_EIT *pk_eit ){
         abo_tsk( ECM_CEX );
   }
/* システムコール例外に対するデフォルトのハンドラ */
   EXCHDR sex_hdr( T_EXC *pk_exc, T_REGS *pk_regs, T_EIT *pk_eit ){
         abo_tsk( ECM_SEX );
   }
```



[図3.21] システムデフォルトのCPU例外ハンドラ

たとえば、CPU例外ハンドラが未定義でCPU例外が発生した場合には、システムデフォルトの例外ハンドラにより、[図3.21] のような動作を行うことになる。

システムコール例外の場合に、エラーコードをそのまま終了要求の例外コードとして使用するのではなく、例外クラスビットパターン（対応する例外クラスを2のべき乗のパターンにしたもの）を例外コードとして使用しているのは、複数の終了要求が重なった場合に、例外コードがビットパターンの論理和をとる仕様になっているからである。また、システムコール例外の場合に、例外クラスに対応した別々の例外クラスビットパターンではなく、ECM_SEX を共通の例外コードとして使用するのは、デフォルトの例外ハンドラで使用する例外コードの範囲を狭くし、ユーザの使用する例外コードと競合する可能性を少なくするためである。ユーザ側では、ter_tsk,ras_ter などに対する終了コードとして、ECM_CEX, ECM_SEX 以外のビットを利用することにより、ビット対応にすべての要求を区別することが可能になる。ただし、将来システムで定義される例外コードが増えた場合には、上位ビット（上位1〜2バイト）を使用する予定なので、ユーザが利用する終了コードとしては、下位ビット（下位2バイト）を使用するのが望ましい。

なお、上記のケースで、終了ハンドラからCPU例外やシステムコール例外に関する詳しい情報を知りたければ、hdr_sts を使って、システムデフォルトの例外ハンドラ（上記の cex_hdr, sex_hdr）の環境に対する例外コード (pk_exc->exccd) を参照すればよい。また、例外発生時のレジスタの値（パラメータ）などを知りたい場合にも、hdr_sts を使って、上記の cex_hdr, sex_hdr の環境に対する pk_regs を参照すればよい。

一方、タスクに対する終了ハンドラが未定義で終了要求が発生した場合は、即座にタスクを終了する。また、拡張SVCハンドラに対する終了ハンドラが未定義で終了要求が発生した場合は、即座に拡張SVCハンドラからリターンする。この場合のエラーコードは E_TER となる。

結局、システムデフォルトの終了ハンドラは次のように書くことができる。

```
/* タスクに対するデフォルトの終了ハンドラ */
  EXCHDR ext_hdr( T_EXC *pk_exc, T_REGS *pk_regs, T_EIT *pk_eit ){
      /* タスク終了 */
      ext_tsk( );
  }
/* 拡張SVCハンドラに対するデフォルトの終了ハンドラ */
  EXCHDR sext_hdr( T_EXC *pk_exc, T_REGS *pk_regs, T_EIT *pk_eit ){
      /* 拡張SVCハンドラから戻る */
      ret_svc(s_ercd=E_TER);
  }
```

複数の例外が同時に発生した場合の動作

ITRON2で同時に発生する可能性のある例外は、他タスクから要求の出る終了要求、強制例外、および自タスクから要求の出るシステムコール例外の3つである。これらの例外は、すべてOSレベルで発生する例外である。これに対して、CPU例外は機械命令レベルで発生し、即座に起動されるため、上記の例外と同時に検出されることはない。また、OSレベルの例外であっても、複数のシステムコール例外が同時に発生することはない。(システムコールの複数のエラーが同時に検出される可能性はあるが、それらの間の優先度はエラーの検出順序といった問題で処理されるため、複数のシステムコール例外が発生するわけではない。システムコールから返ってくるエラーコードはただ一つである。)

終了ハンドラを起動する時には、終了要求に対するマスクだけではなく、強制例外に対するマスクも自動的にセットされる。したがって、終了要求と強制例外が同時に受け付けられた場合でも、タスクに対する終了ハンドラのスタックフレームを形成した後は、強制例外ハンドラの起動は行われず、スタックフレームも形成されない。強制例外については PENDING 状態のままとなる。

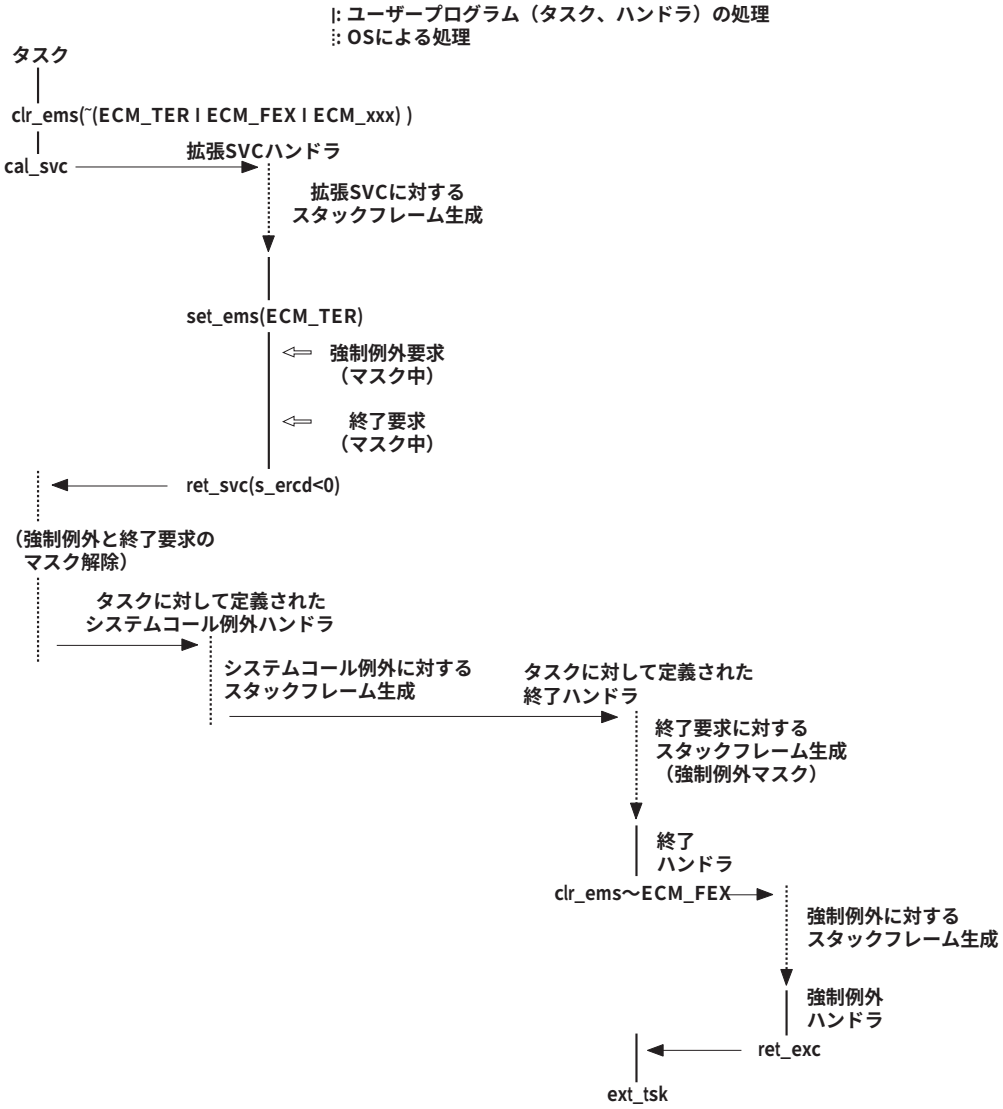
強制例外、終了、システムコール例外をマスクしていないタスクが拡張SVCを呼びだし、その拡張SVCが強制例外と終了をマスクしている間に強制例外と終了要求を受け付け、両者の要求が PENDING 状態のまま ret_svc(s_ercd＜0) を実行した場合、終了要求、強制例外、システムコール例外の3つの例外が重なることになる。この場合、「システムコール例外→終了要求→強制例外の順にネストしたスタックフレームを作り、その逆順にハンドラが起動される」というのが推奨仕様である。

たとえば、強制例外、終了、システムコール例外をマスクしていないタスクが拡張SVCを呼びだし、その拡張SVCが強制例外と終了をマスクしている間に強制例外と終了要求を受け付け、両者の要求が PENDING 状態のまま ret_svc (s_ercd＜0) を実行した場合のOSの推奨動作は次のようになる。

1) タスクのシステムコール例外ハンドラに対するスタックフレームを形成
2) タスクの終了ハンドラに対するスタックフレームを形成

この後は、タスクに対して定義された終了ハンドラが起動され、その中でタスクを終了してしまうため、システムコール例外ハンドラは起動されないことになる。また、終了ハンドラの中で clr_ems (ECM_FEX) を実行することにより、強制例外ハンドラが起動される。これらの動作を図で示すと［図3.22］のようになる。

なお、［図3.22］のような場合、拡張SVCに対する終了ハンドラは起動されない。拡張SVCから戻る前の環境では終了ハンドラの起動がマスクされており、また拡張SVCから戻った後の環境では、例外ハンドラ定義環境も元のコンテキスト（タスク）のものに戻っている。例外マスク環境と例外ハンドラ定義環境の切り換えは不可分に行われるため、拡張SVCに対する終了ハンドラが起動されることはない。



[図3.22] 終了要求と強制例外とシステムコール例外が重なった場合の動作

タスク独立部に対する例外処理

タスク独立部（割込みハンドラ）に対しても、idef_cex, idef_sex システムコールによって例外ハンドラを定義することができる。ただし、性能面への配慮から、多重割込み（外部割り込み）の入り口や出口で例外マスク環境や例外ハンドラ定義環境を切り換えることは無理がある。したがって、割込みハンドラ同士であれば、優先度が異なっていても、すべて共通の例外ハンドラと例外マスク環境を持つことになる。また、割込みハンドラ中から起動された例外ハンドラや拡張SVCハンドラについても、その中でさらに優先度の高い割込みを受け付けるケースを考えると、別々の例外ハンドラ定義環境や例外マスク環境を持つことはできない。結局、割込みハンドラ（タスク独立部）は、その中から呼ばれた拡張SVCハンドラや例外ハンドラを含めて、全体で共通の例外ハンドラ定義環境と例外マスク環境を持つということになる。

たとえば、割込みハンドラから拡張SVCハンドラAが呼ばれ、そこで例外が発生した場合でも、拡張SVCハンドラAに対する例外ハンドラではなく、idef_Xexで定義されたタスク独立部に対する例外ハンドラが起動される。また、割込みハンドラに対するシステムコール例外の各例外クラスに対するマスクは、割込みハンドラ全体として共通の値を取ることになる。したがって、タスク独立部から発行される拡張SVCとタスク部から発行される拡張SVCとでは、システムコール例外の処理について区別して扱うことが必要になる。

タスク独立部に対する例外マスク環境の設定は、clr_ems, set_ems で tskid=TSK_INDP(-2) を指定することによって行う。タスク独立部に対する emsptn の初期値は次のようになる。

終了処理 (ECM_TER)	可（マスクをセットできない）
強制終了 (ECM_FEX)	マスク（マスクをクリアできない）
CPU 例外 (ECM_CEX)	可（マスクをセットできない）
システムコール例外	全部マスク

なお、タスク独立部に対する終了例外や強制例外の機能は無いし、CPU 例外はマスク不可なので、タスク独立部に対する例外マスク環境が意味を持つのは、システムコール例外に限られる。また、タスク独立部に対するシステムコール例外の機能は、必要性が少なく、インプリメントの負担も

大きいので、インプリメント依存の機能とする。

タスク独立部に対する例外マスク環境は一つ（システム共通）なので、割込みが多重に発生しても、例外マスクは共通に使用される。したがって、タスク独立部でシステムコール例外が発生した場合には、タスク部あるいは準タスク部でシステムコール例外が発生した場合とは異なり、対応する例外マスクはセットされないものとする。タスク独立部に対する例外マスクの値が変化するのは、明示的に clr_ems(tskid=TSK_INDP), set_ems(tskid=TSK_INDP) を実行した場合のみ（システムダウンの前のタスク独立部用終了ハンドラの起動時に終了要求に対する例外マスクがセットされるケースを除く）である。もし、割込みハンドラの中でシステムコール例外が発生した場合に例外マスクをセットする仕様にすると、割込みハンドラAの実行中に別の割込みが発生して割込みハンドラBが起動された場合に、割込みハンドラAに対するシステムコール例外ハンドラを実行中かどうかによって、割込みハンドラBに対する例外マスクの値が変わってくることになる。しかし、割込みハンドラBは割込みハンドラAとは全く無関係なので、これは不合理である。なお、タスク独立部（割込みハンドラ）に対するシステムコール例外の機能はインプリメント依存であり、この仕様が適用されるのは、タスク独立部に対するシステムコール例外の機能がインプリメントされた場合に限られている。

タスク独立部に対する例外マスクがクリアされており、かつ例外ハンドラが未定義の状態でタスク独立部の例外が発生した場合の動作は、タスク部/準タスク部の動作に準じたものとすることを推奨する。具体的には、上記の場合に、タスク独立部に対する終了ハンドラ（idef_extで定義したハンドラ）が定義されていれば、それが起動された後にシステムダウンとなる。タスク独立部に対する終了ハンドラが定義されていなければ、即座にシステムダウンとなる。

タスク用終了ハンドラの定義
def_ext

タスク用CPU例外ハンドラの定義
def_cex

タスク用システムコール例外ハンドラの定義
def_sex

def_ext:　Define ExitHandler
def_cex:　Define CPUExceptionHandler
def_sex:　Define SystemcallExceptionHandler

【パラメータ】

tskid	TaskIdentifier	タスクID
exhatr	ExceptionHandlerAttribute	例外ハンドラ属性
＜1＞exthdr	ExitHandlerAddress	終了ハンドラアドレス
＜2＞cexhdr	CPUExceptionHandlerAddress	CPU例外ハンドラアドレス
＜3＞sexhdr	SystemCallExceptionHandlerAddress	システムコール例外ハンドラアドレス
	［＜1＞def_ext のみ］	
	［＜2＞def_cex のみ］	
	［＜3＞def_sex のみ］	

【リターンパラメータ】

なし

【解説】

自タスクの例外ハンドラ、終了ハンドラを定義する。あるいは、DORMANT 状態の他タスクの例外ハンドラ、終了ハンドラを定義する。

例外ハンドラでは、例外の原因となった情報を、スタックを通じて例外コード exccd として受け取ることができる。exccd の内容は、終了ハンドラの場合は ter_tsk, ras_ter, abo_tsk システムコールで指定したexccd であり、

CPU例外の場合はプロセッサの例外処理の際に得られる情報（ベクトル番号など）であり、システムコール例外の場合はシステムコールのリターン値（エラーコード）である。

exhatr のサイズは標準で4バイトである。exhatr のフォーマットを［図3.23］に示す。このうち、最上位バイト（bit0～bit7 あるいは 2^31～2^24 のビット）は未使用であり、第二上位バイト（bit8～bit15 あるいは 2^23～2^16 のビット）はインプリメント依存の属性を表わし、下位ハーフワード（bit16～bit31 あるいは 2^15～2^0 のビット）はシステム属性を表わす。

exhatr のシステム属性の部分では、次のような指定を行うことができる。

exhatr : = (TA_ASM ‖ TA_HLNG)

TA_ASM	ハンドラがアセンブラで書かれている
TA_HLNG	ハンドラが高級言語で書かれている

TA_HLNG の指定を行った場合には、ハンドラ起動時に直接 exthdr, cexhdr, sexhdr 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 exthdr, cexhdr, sexhdrのアドレスにジャンプする。

exthdr,cexhdr,sexhdr = NADR(-1) の指定により、前に定義されていた例外（終了）ハンドラの定義が解除され、未定義の状態になる。ただし、前に定義されていた例外（終了）ハンドラの定義解除を行わずに、直接新しいハンドラを定義しても、エラーにはならない。また、もともとハンドラが定義されていない状態（定義が解除された状態）で、exthdr,cexhdr,sexhdr=NADR の指定による定義解除の操作を行っても、エラーにはならない。後者の場合、このシステムコールは正常終了するが、例外（終了）ハンドラは未定義のままである。

［図3.23］exhatrのフォーマット

tskid = TSK_SELF により、自タスクのハンドラを指定する。また、tskid=TSK_CMN によりタスク間で共通の例外（終了）ハンドラを指定する。タスク実行中に例外が発生した場合には、OSはタスク固有の例外ハンドラが定義してあるかどうかを調べ、定義してあればそれを実行する。定義してなければ、次にタスク間共通の例外ハンドラが定義してあるかどうかを調べ、定義してあればそれを実行する。タスク間共通の例外ハンドラのコードは、システム全体に一つ（終了要求、強制例外、CPU例外、システムコール例外に対して一つずつ）である。

tskid が他タスクを示し（tskid≠TSK_SELF かつ tskid≠自タスクのID）、かつ tskid のタスクが DORMANT 状態でない場合には、E_NODMT のエラーとなる。

例外ハンドラ起動の際は、OS側で汎用レジスタの退避を行う。OSから例外ハンドラに渡す情報としては、次のようなものがある。

発生した例外の例外クラス、例外コード（エラーコード）
例外の発生したアドレス（PC）
例外発生時のレジスタの内容（例外を発生したシステムコールのパラメータを含む）
その他例外に応じた情報

他タスクの例外（終了）ハンドラは、対象タスクが DORMANT 状態の場合にのみ設定可能である。

終了ハンドラも例外ハンドラの一つとして扱われるため、例外コードの引き渡し、例外のマスクによるハンドラ起動の遅延、終了要求がネストした場合の動作、拡張SVCハンドラ実行中の終了ハンドラ起動などのメカニズムは、例外処理と同様である。（例外処理全体の説明参照）

終了ハンドラの場合、終了要求の受け付け後は自動的にタスクの優先度が高くなる（ter_tskの説明参照）。したがって、終了ハンドラは優先的に実行されることになる。タスクに対する終了ハンドラの最後では、ext_tskまたは exd_tsk システムコールの発行により、自タスクを正常終了させる。タスクに対する終了ハンドラでret_exc を発行した場合には、ext_tsk 相当の動作をする。

例外ハンドラ、終了ハンドラは、例外を起こしたコンテキスト（あるいは終了要求を受け付けたコンテキスト）の実行モード（リング、レベル）と、

同じ実行モード（リング、レベル）で実行される。これは、タスク間共通の例外ハンドラを実行する場合も同様である。タスク間共通の例外ハンドラを実行する場合に、自動的に実行モード（リング、レベル）が上がるというわけではない。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足 ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSID	予約ID番号（-4≦tskid≦-2、タスク独立部の発行でtskid=0）
E_IDOVR	ID範囲外（インプリメント依存）
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_NODMT	タスクがDORMANTでない（tskidが他タスクの場合）
E_RSATR	予約属性（exhatrが不正）
E_ILADR	不正アドレス（exthdr,cexhdr,sexhdrが奇数、あるいは使用できない値）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でtskid＜(-4)）

例外ハンドラから復帰

ret_exc

ret_exc: Return from ExceptionHandler

【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

【解説】

例外ハンドラ（終了ハンドラを除く）から例外発生前のコンテキストに復帰する。

例外ハンドラから戻る際には、自動的にレジスタの復帰を行なう。したがって、機能コードを使ったインタフェースが可能である。

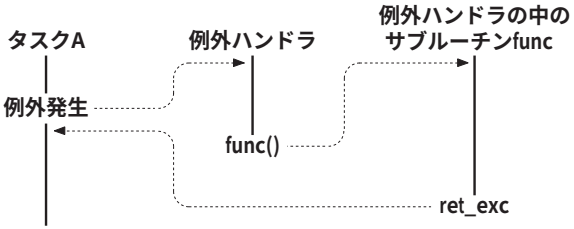
このシステムコールにより、実行モード（リング、レベル）は不変である。

例外ハンドラや拡張SVCハンドラの中でサブルーチンを呼び、そのサブルーチンの中で直接 ret_exc やret_svc を実行した場合でも、エラーとはならず、そのまま例外ハンドラや拡張SVCハンドラ起動前のコンテキストに復帰するものとする。すなわち、サブルーチンに対するスタックを残したまま、ret_exc や ret_svc によって直接例外ハンドラや拡張SVCハンドラから戻ることが可能である。この場合、ret_exc や ret_svc の処理の中で、自動的にサブルーチンに対するスタックフレームの解放が行われることになる。

例外ハンドラからサブルーチンを呼び、その中で ret_exc を実行した場合の動作は、[図3.24] のようになる。

なお、タスク独立部である割込みハンドラやタイマハンドラの場合は、トラップ命令ではなくサブルーチンリターン命令を使ったインタフェースとなっているため、このようなことはできない。スタックフレームを残したままリターンできるのは、拡張SVCハンドラと例外（終了）ハンドラに限られる。

タスクに対する終了ハンドラの中で ret_exc が実行された場合は、ext_tsk と同等の処理を行う。また、拡張SVCハンドラに対する終了ハンドラの中



[図3.24] サブルーチン中で実行したret_exc

で ret_exc が実行された場合には、ret_svc (s_ercd=E_TER) と同等の処理を行う。いずれの場合も、ret_exc によって終了ハンドラが起動された場所に戻るわけではない。

元のコンテキストに戻らない ret_XXX 系のシステムコールでエラーが発生した場合、システムコールの例外ハンドラが定義され、それがマスクされていなければ、原則としてそれを起動する。システムコールに対する例外ハンドラによって、元のコンテキストに戻らないシステムコールに対するエラーの処理もある程度可能になる。

システムコールの例外ハンドラから戻った後の動作、あるいはシステムコールの例外ハンドラを起動しなかった時の動作は、厳密にはインプリメント依存である。ただし、エラーの被害をできるだけ局所化するという意味で、タスクであれば終了要求を発生し、タスク独立部であればその状態から抜ける（ret_int相当）のが望ましいと考える。また、これとは別に、メッセージバッファへのロギングやコンソールへのエラーの表示などを行うことを推奨している。

【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には、メッセージバッファへのロギングや、コンソールへのエラーメッセージの表示などを行うのが望ましいが、詳細な動作はインプリメント依存となる。

E_CTX　　コンテキストエラー（例外ハンドラ実行中ではない）

例外ハンドラの終了

end_exc

end_exc: End ExceptionHandler

【パラメータ】

なし

【リターンパラメータ】

なし

【解説】

例外ハンドラを終了し、元の環境に復帰する。ただし、このシステムコールでは、例外処理環境の復帰やスタックのクリアを行なうだけで、プログラムカウンタはそのままである。したがって、ret_exc とは異なり、システムコールを発行したコンテキストに戻ってくる。

end_exc と ret_exc との違いは、システムコール後の実行環境（プログラムカウンタを含む）が例外発生点に戻る（ret_exc）か、システムコール(end_exc) 発生点に戻るかの違いだけである。ネストした例外ハンドラの中で end_exc を実行した場合にも、ret_exc と同様に、1段分の例外ハンドラ環境のみが復帰する。

この機能は、元の場所（例外発生点）に戻りたくない場合に使用する機能であり、終了処理やエラー回復などの場合に使用される。このシステムコールの発行後は、アセンブラのジャンプ命令や C言語の longjmp ( ) によって、例外発生タスクやハンドラの例外発生点以外の場所（初期化や処理再実行を行う部分など）にジャンプするのが普通である。

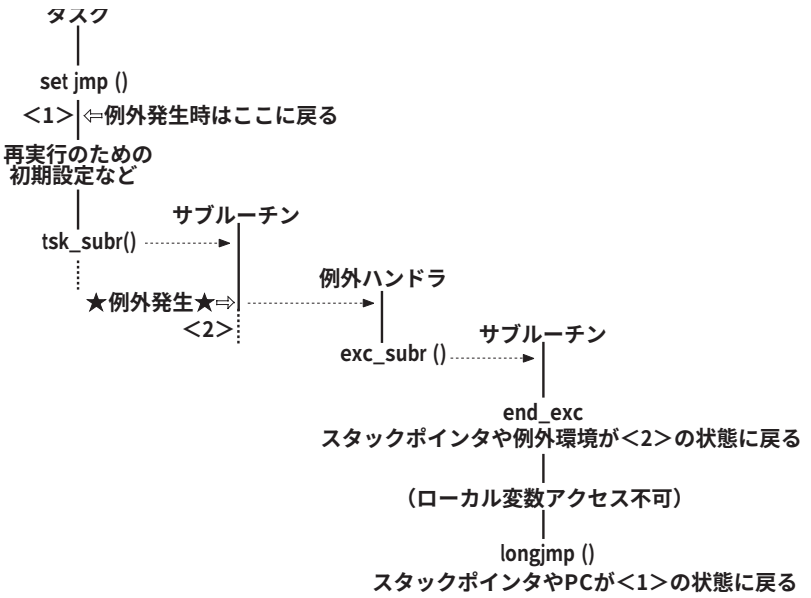
このシステムコールは、タスクに対する例外ハンドラでも、拡張SVCハンドラに対する例外ハンドラでも利用することができる。

end_exc の利用イメージを［図3.25］に示す。

end_exc を実行すると例外ハンドラに対するスタックフレームが削除されるため、スタックポインタの値は変更され、例外発生前の値に戻る。したがって、end_exc 実行以後は、ローカル変数のアクセスはできなくなる。これはITRONの問題というよりも言語環境の問題である。また、同じ理由

により、end_exc の高級言語インタフェースルーチンではスタック上にリターンアドレスを保持することができないため、リターンアドレスを保持するために何らかの工夫が必要になる。たとえば、高級言語の関数呼び出しにおいて値の保存を保証しなくてもよいレジスタ（R1など-コンパイラにより異なる）にリターンアドレスを退避しておく、といった方法が考えられる。

また、end_exc の場合、ret_exc とは異なり、例外ハンドラの起動時に保存していたレジスタ内容の復帰は行わない。end_exc 実行後のレジスタ（SPを除く）の内容は、例外ハンドラ起動前のレジスタの内容に戻るのではなく、end_exc 実行前のレジスタの内容がそのまま保存される。例外ハンドラ起動前のレジスタ（PCを含む）の内容は、捨てられる。



※この例では、例外発生前にサブルーチン tsk_subr() を呼んだり、例外ハンドラの中でサブルーチン exc_subr() を呼んだりしているが、tsk_subr() や exc_subr() のサブルーチン呼びだしが無くても同様の動作になる。

［図3.25］end_excの利用イメージ

end_exc は例外環境を切り替えるので、end_exc の発行により、ペンディングになっていた例外が起動される場合がある。例えば、次のようなプログラムでは、end_exc と longjmp の間で強制例外ハンドラが起動されることがある。この場合、end_exc と longjmp が連続して実行されるとは限らない。

```
EXCHDR exchdr( ){
        ...
    set_ems(ECM_FEX);          /* 強制例外マスク */
      ...                  ←他タスクによる強制例外要求
                             （マスク中なのでペンディングになる）
    end_exc( );
                           →ここで強制例外ハンドラ起動
                             （exchdr 起動前の例外環境で ECM_FEX がクリアされていた場合）
    longjmp(buf,val);
}
```

終了ハンドラの中で end_exc を実行した場合には、E_CTX のエラーとなる。

【エラーコード (ercd)】

E_OK	正常終了
E_CTX	コンテキストエラー（例外ハンドラ実行中ではない、あるいは終了ハンドラ実行中である）

例外処理マスククリア

clr_ems

clr_ems: Clear ExceptionMask

【パラメータ】

tskid	TaskIdentifier	タスクID
clrptn	ClearExceptionMaskPattern	クリアする例外マスクのビットパターン

【リターンパラメータ】

なし

【解説】

自タスク、あるいは、DORMANT 状態の他タスクの例外処理環境の emsptn をクリアし、対応する例外クラスの例外（終了）ハンドラの起動を許可する。

具体的な動作としては、現在の emsptn に対して、clrptn の論理積をとる。その後、ペンディングになっていた終了処理要求、強制例外の有無がチェックされ、その結果によって、遅らせられていた例外ハンドラの起動が行なわれる。

tskid = TSK_SELF の指定により自タスクの例外マスクを操作する。自タスクが現在例外ハンドラあるいは拡張SVCハンドラを実行中である場合には、自タスクのハンドラ起動前の例外マスクではなく、自タスクから呼ばれている現在実行中のハンドラに対する例外マスクをセットする。拡張SVCを呼んだ側や例外を起こした側の例外マスクは無変化である。

tskidが他タスクを示し（tskid≠TSK_SELF かつ tskid≠自タスクのID)、かつ tskid のタスクが DORMANT 状態でない場合には、E_NODMT のエラーとなる。このシステムコールの対象となる他タスクは必ず DORMANT 状態なので、対象タスクがハンドラ実行中であるということはない。

clr_ems, set_ems で tskid=TSK_INDP(-2) を指定することにより、タスク独立部(Task Independent Portion) に対する例外マスクの操作を行うことができる。この機能により、タスク独立部の例外マスク環境を操作することが可

能になる。ただし、タスク独立部に対する例外マスクが意味を持つのは、システムコール例外に限られている。また、タスク独立部に対するシステムコール例外の機能はインプリメント依存である。tskid = TSK_INDP を指定した clr_ems や set_ems は、タスク部や準タスク部から発行することも可能である。

clr_ems によってクリアできない例外マスクをクリアしようとした場合や、set_ems によってセットできない例外マスクをセットしようとした場合には、E_PAR のエラーとなる。具体的には、以下の操作を行った場合に E_PAR のエラーとなる。

clr_emsでE_PARとなる場合 :

ECM_NONE	すべての場合
ECM_FEX	拡張SVCハンドラ内での自タスク指定
	強制例外がマスクされた状態から起動された例外（終了）ハンドラ内での自タスク指定
	タスク独立部に対する指定
ECM_TER	終了ハンドラ内での自タスク指定
	終了要求がマスクされた状態から起動された例外（終了）ハンドラ内での自タスク指定
ECM_ABO	終了ハンドラ実行中での自タスク指定
ECM_SUS	SUSPEND要求がマスクされた状態から起動された例外（終了）あるいは拡張SVCハンドラ内での自タスク指定

一方、clr_ems や set_ems が何も意味のない操作を行う場合（ECM_CEXをクリアしようとした場合、終了ハンドラ内でECM_TERをセットしようとした場合、clrptnの全ビットを1としてclr_emsを実行しようとした場合、setptnの全ビットを0としてset_emsを実行しようとした場合など）には、特にエラーとはしない。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦tskid≦-3、tskid=(-1)、タスク独立部の発行でtskid=0）
E_PAR	一般的なパラメータエラー（クリアできないビットの指定）
E_IDOVR	ID範囲外（インプリメント依存）
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_NODMT	タスクがDORMANTでない（tskidが他タスクの場合）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でtskid＜(-4)）

例外処理マスクセット

set_ems

set_ems: Set ExceptionMask

【パラメータ】

tskid	TaskIdentifier	タスクID
setptn	SetExceptionMaskPattern	セットする例外マスクのビットパターン

【リターンパラメータ】

なし

【解説】

自タスク、あるいは、DORMANT 状態の他タスクの例外処理環境の emsptn をセットし、対応する例外クラスの例外（終了）ハンドラの起動を禁止する。

具体的な動作としては、現在の emsptn に対して、setptn の論理和をとる。

tskid = TSK_SELF の指定により自タスクの例外マスクを操作する。自タスクが現在例外ハンドラあるいは拡張SVCハンドラを実行中である場合には、自タスクのハンドラ起動前の例外マスクではなく、自タスクから呼ばれている現在実行中のハンドラに対する例外マスクをセットする。拡張SVCを呼んだ側や例外を起こした側の例外マスクは無変化である。

tskid が他タスクを示し（tskid≠TSK_SELF かつ tskid≠自タスクのID）、かつ tskid のタスクが DORMANT 状態でない場合には、E_NODMT のエラーとなる。このシステムコールの対象となる他タスクは必ず DORMANT 状態なので、対象タスクがハンドラ実行中であるということはない。

clr_ems, set_ems で tskid=TSK_INDP (-2) を指定することにより、タスク独立部 (Task Independent Portion) に対する例外マスクの操作を行うことができる。この機能により、タスク独立部の例外マスク環境を操作することが可能になる。ただし、タスク独立部に対する例外マスクが意味を持つのは、システムコール例外に限られている。また、タスク独立部に対するシステムコール例外の機能はインプリメント依存である。tskid=TSK_INDP を指定した

clr_ems や set_ems は、タスク部や準タスク部から発行することも可能である。

このシステムコールを使っても、CPU例外に対するマスクを行うことはできない。

clr_ems によってクリアできない例外マスクをクリアしようとした場合や、set_ems によってセットできない例外マスクをセットしようとした場合には、E_PAR のエラーとなる。具体的には、以下の操作を行った場合に E_PAR のエラーとなる。

set_emsでE_PARとなる場合 :

ECM_TER	タスク独立部に対する指定
ECM_CEX	すべての場合
ECM_ABO	終了ハンドラ実行中でない場合の自タスク指定 他タスク（DORMANT状態）に対する指定
ECM_SUS	タスク部での自タスク指定 他タスク（DORMANT状態）に対する指定

一方、clr_ems や set_ems が何も意味のない操作を行う場合（ECM_CEXをクリアしようとした場合、終了ハンドラ内でECM_TERをセットしようとした場合、clrptnの全ビットを1としてclr_emsを実行しようとした場合、setptnの全ビットを0としてset_emsを実行しようとした場合など）には、特にエラーとはしない。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦tskid≦-3、tskid=(-1)、タスク独立部の発行でtskid=0）
E_PAR	一般的なパラメータエラー(セットできないビットの指定)
E_IDOVR	ID範囲外（インプリメント依存）
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_NODMT	タスクがDORMANTでない（tskidが他タスクの場合）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でtskid＜(-4)）

タスク独立部用終了ハンドラの定義　idef_ext

タスク独立部用CPU例外ハンドラの定義　idef_cex

タスク独立部用システムコール例外ハンドラの定義　idef_sex

idef_ext:　Define ExitHandler for TaskIndependentPortion
idef_cex:　Define CPUExceptionHandler for TaskIndependentPortion
idef_sex:　Define SystemcallExceptionHandler for TaskIndependentPortion

【パラメータ】

exhatr	ExceptionHandlerAttribute	例外ハンドラ属性
＜1＞exthdr	ExitHandlerAddress	終了ハンドラアドレス
＜2＞cexhdr	CPUExceptionHandlerAddress	CPU例外ハンドラアドレス
＜3＞sexhdr	SystemCallExceptionHandlerAddress	システムコール例外ハンドラアドレス

［＜1＞idef_ext のみ］
［＜2＞idef_cex のみ］
［＜3＞idef_sex のみ］

【リターンパラメータ】

なし

【解説】

タスク独立部（割込みハンドラ、タイマハンドラ、OS）に対する例外ハンドラ、終了ハンドラを定義する。

各システムコールでは、次のようなハンドラを定義する。

・idef_ext により定義されるハンドラ
— タスク独立部から発行したシステムコールにおいて、システムを終了せざるをえないような重大なエラーが発生した場合に起動される。タスク独立部でCPU例外が発生し、CPU例外に対するハンドラが未定義の場合もこれに含まれる。ハンドラを実行した後は、システムダウンとなる。
— OSの内部で、システムを終了せざるをえないような原因不明の重大なエラーが発生した場合に起動される。ハンドラを実行した後は、システムダウンとなる。

・idef_cex により定義されるハンドラ
— タスク独立部でCPU例外が発生した場合に起動される。ハンドラを実行した後は、元のコンテキスト（タスク独立部）に戻れる場合がある。

・idef_sex により定義されるハンドラ
—（エラーコードが E_SYS 以外の場合）タスク独立部から発行したシステムコールにおいて、システムコール例外が発生した場合に起動される。ハンドラを実行した後は、元のコンテキスト（タスク独立部）に戻れる場合がある。
なお、タスク独立部でシステムコール例外が発生した場合には、タスク部あるいは準タスク部でシステムコール例外が発生した場合とは異なり、対応する例外マスクはセットされない。システムコール例外ハンドラ実行中の emsptn の値は、例外発生前と同じ値をとる。
—（エラーコードが E_SYS の場合）OSの内部で、回復できるかもしれない程度の原因不明のエラーが発生した場合に起動される。OSの内部で、OSだけでは対処できないようなCPU例外が発生した場合も、これに含まれる。ハンドラを実行した後は、元のコンテキスト(OS)に戻れる場合がある。

このうち、idef_ext で定義される「タスク独立部に対する終了ハンドラ」は、システム全体に対する終了ハンドラを意味するものであり、システムダウン時に一度だけ起動されるハンドラである。タスク独立部に対する例外ハンドラが未定義のため一般の例外が終了要求に置き換わった場合や、その他の異常によるシステムダウンの場合には、このシステム終了ハンドラを実行した後でシステムを停止する。このハンドラにより、メッセージ

［図3.26］exhatrのフォーマット

の表示や状況の解析がある程度可能になる。

exhatrのサイズは標準で4バイトである。exhatrのフォーマットを［図3.26］に示す。このうち、最上位バイト（bit0～bit7 あるいは 2^31～2^24 のビット）は未使用であり、第二上位バイト（bit8～bit15 あるいは 2^23～2^16 のビット）はインプリメント依存の属性を表わし、下位ハーフワード（bit16～bit31 あるいは 2^15～2^0 のビット）はシステム属性を表わす。

exhatr のシステム属性の部分では、次のような指定を行うことができる。

exhatr : = (TA_ASM ‖ TA_HLNG)

TA_ASM	ハンドラがアセンブラで書かれている
TA_HLNG	ハンドラが高級言語で書かれている

TA_HLNG の指定を行った場合には、ハンドラ起動時に直接 exthdr, cexhdr, sexhdr 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 exthdr, cexhdr, sexhdrのアドレスにジャンプする。

exthdr,cexhdr,sexhdr = NADR (-1) の指定により、前に定義されていた例外（終了）ハンドラの定義が解除され、未定義の状態になる。ただし、前に定義されていた例外（終了）ハンドラの定義解除を行わずに、直接新しいハンドラを定義しても、エラーにはならない。また、もともとハンドラが定義されていない状態（定義が解除された状態）で、exthdr,cexhdr,sexhdr=NADR の指定による定義解除の操作を行っても、エラーにはならない。後者の場合、このシステムコールは正常終了するが、例外（終了）ハンドラは未定義のままである。

これらの例外ハンドラや終了ハンドラは、例外を起こした実行モード（リング、レベル）と同じ実行モード（リング、レベル）で実行される。

idef_XXX ではOS内部で発生したエラーに対する例外ハンドラを定義で

きるが、ここで定義される例外ハンドラは、OS自身がエラー回復を行うことを妨げるものではなく、OS自身が何らかの処置を行い、それでもエラーを回復できない場合に起動されるハンドラという意味である。当然のことながら、OS内部でエラーが発生した場合には、idef_XXX で定義されるハンドラの有無にかかわらず、できる限りOS内部でエラーの回復に努めるべきである。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足 ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSATR	予約属性（exhatrが不正）
E_ILADR	不正アドレス（exthdr,cexhdr,sexhdrが奇数、あるいは使用できない値）

メモリプール管理機能

この章の「メモリプール管理機能」で対象としているのは、ソフトウエアによるメモリプールやメモリブロックの管理機能である。MMU に関する操作（MMUサポート機能）は、ここには含まれない。

ITRON2では、MMUの有無にかかわらず、タスクがローカルに使用するメモリブロックを扱うシステムコールと、タスク間で共有されるメモリブロックを扱うシステムコールとを完全に分離している。これは、MMU 版への移行をスムーズにするためであり、共有メモリブロックに対するメモリプール（共有メモリプール）とローカルメモリブロックに対するメモリプール（ローカルメモリプール）は別のオブジェクトとして扱われる。このうち、この章では、タスク間で共有される共有メモリプールについての説明を行う。

タスクがローカルに使用するローカルメモリプールについては、拡張機能の「ローカルメモリプール管理機能」の項で説明を行う。ただし、ローカルメモリプール管理機能がインプリメントされない場合には、本章で説明する共有メモリプール管理機能により、ローカルメモリプールの機能を代用することになる。

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。両者はメモリプールの属性として区別される。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、基本サイズの整数倍といった形でブロックサイズを指定することができる。

mplid=TMPL_OS (-4) のメモリプールは、OS（システム）用のメモリプールとして扱い、余っているメモリはそこに所属するものと考える。mplid = TMPL_OS (-4) のメモリプールに対して状態参照のシステムコール (mpl_sts) を発行することにより、OSの残りのメモリ容量などを知ることができる。

このほか、mplid=(-3)〜(-2) のメモリプールを、インプリメント依存のOS用メモリプール（スタック専用のメモリプールなど）として利用することができる。この機能は、残りのメモリ容量に関してより詳しい情報

を提供する時に利用できる機能であるが、機能の有無や詳細はインプリメント依存である。mplid = (-3)〜(-2) のメモリプールを細かく使い分ければ、より詳細な情報をユーザに提供することができる。なお、mplid = (-1), mplid = 0 は reserved である。

メモリプールの生成

cre_mpl

cre_mpl: Create MemoryPool

【パラメータ】

mplid	MemoryPoolIdentifier	メモリプールID
mplatr	MemoryPoolAttribute	メモリプール属性
mplsz	MemoryPoolSize	メモリプール全体のサイズ
blksz	MemoryBlockSize	メモリブロック基本サイズ

【リターンパラメータ】

なし

【解説】

cre_mpl では、mplid で指定された ID 番号を持つメモリプールを生成する。次に、生成されたメモリプールに対して管理ブロックを割り付ける。

メモリプールの生成時には、mplatr によって、固定長メモリプール/可変長メモリプールの区別を指定する。

メモリプール全体の大きさは、mplsz によって指定する。メモリプール生成に必要なメモリ領域の確保は、システム生成時などに行なわれる。

可変長のメモリプールの場合、blksz により、メモリ獲得、解放の最小単位が決められる。固定長のメモリプールであれば、blkszの大きさでのメモリ獲得、解放しかできない。

ID 番号が (-4)〜0 のメモリプールは生成できない。また、負の ID 番号のメモリプールは、システム用のものである。

mplatr のサイズは標準で4バイトである。mplatr のフォーマットを［図3.27］に示す。このうち、最上位バイト（bit0〜bit7 あるいは 2^31〜2^24 のビット） はユーザ属性を表わし、第二上位バイト（bit8〜bit15 あるいは 2^23〜2^16 のビット）はインプリメント依存の属性を表わし、下位ハーフワード（bit16〜bit31 あるいは 2^15〜2^0 のビット）はシステム属性を表わす。ユーザは、メモリプールに関する情報を入れておくために、mplatr の最上位バイトを自由に使用することができる。

［図3.27］mplatrのフォーマット

cre_mpl で指定した mplatr は、mpl_sts により読み出すことができる。
mplatr のシステム属性の部分では、次のような指定を行うことができる。

mplatr: = (TA_TFIFO ‖ TA_TPRI) ｜ (TA_FIRST ‖ TA_CNT)
　　｜ (TA_VAR ‖ A_FIX)

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	行列先頭のタスクを優先扱い
TA_CNT	要求数の少ないタスクを優先扱い
TA_VAR	可変長メモリブロック用のメモリプール
TA_FIX	固定長メモリブロック用のメモリプール

TA_TPRI, TA_TFIFO の属性により、タスクがメモリ獲得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。

さらに、TA_CNT と TA_FIRST の属性指定により、要求メモリ数の少ないタスクを優先扱いにするか、メモリ待ち行列先頭のタスクを優先扱いにするかが指定できる。TA_CNT の属性を指定した場合には、要求メモリ数によって、行列途中のタスクから先に待ち解除になる場合がある。例えば、ある可変長メモリプールに対して要求ブロック数 = 5 のタスク A と要求ブロック数=1 のタスクB がこの順で待っており、rel_blk により空きブロックの数が 1 になった場合、行列先頭にあるタスクA は要求ブロック数が多いので、行列の後にあるタスクB の方が先にメモリを獲得する。

一方、TA_FIRST の属性を指定した場合には、要求ブロック数にかかわらず、必ず行列先頭のタスクからメモリが割り当てられる。これは、要求ブロック数よりも優先度を重視した考え方である。上の例では、rel_blk 実

行後もタスクB は待ち解除にならず、さらに rel_blk　が実行されて連続した空きブロックの数が 5 以上になった時にはじめてタスクA にメモリが割り当てられる。タスクB には、その後の rel_blk によってメモリが割り当てられる。

なお、TA_FIX 属性（固定長メモリプール）の場合は TA_CNT と TA_FIRST の属性の違いが無くなってしまうが、これに関しては何もチェックしない。(TA_FIX | TA_CNT) の指定と (TA_FIX | TA_FIRST) の指定はどちらもエラーにはならず、同じ動作を行なう。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足(管理ブロック用の領域が確保できない) ※ このエラーが発生するかどうかはインプリメント依存である。
E_NOMEM	メモリ不足（メモリプール用の領域が確保できない） ※ E_NOSMEMとE_NOMEMとの厳密な区別はインプリメント依存である。
E_RSID	予約ID番号（-4≦mplid≦0）
E_RSATR	約属性（mplatrの下位3バイトが不正）
E_PAR	一般的なパラメータエラー（mplsz,blkszが不正）
E_IDOVR	ID範囲外（mplidがシステムで利用できる範囲を越えた）
E_EXS	オブジェクトが既に存在している（同一ID番号のメモリプールが存在）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でmplid＜(-4)）

メモリプールを削除する

del_mpl

del_mpl: Delete MemoryPool

【パラメータ】

mplid	MemoryPoolIdentifier	メモリプールID

【リターンパラメータ】

なし

【解説】

mplid で示される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に対しては、何の保証もしない。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。

この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る場合にも、本システムコールは正常終了するが、メモリ獲得を待っていたタスクにはエラー E_DLT が返される。

本システムコール発行後は、そのID番号のメモリプールを再び新しく生成することができる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号 (-4≦mplid≦0)
E_IDOVR	ID範囲外 (インプリメント依存)
E_NOEXS	オブジェクトが存在していない (mplidのメモリプールが存在しない)
E_OACV	オブジェクトアクセス権違反 (拡張SVC以外のユーザタスクからの発行でmplid＜(-4))

共有メモリブロックを獲得する
get_blk

get_blk:　Get Shared Memory Block

【パラメータ】

mplid	MemoryPoolIdentifier	メモリプールID
bcnt	ContinuousBlockCount	連続領域のブロック数
tmout	Timeout	タイムアウト指定

【リターンパラメータ】

blk	BlockStartAddress	メモリブロックの先頭アドレス

【解説】

mplid で示されるメモリプールから、共有メモリブロック（固定長/可変長）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。

get_blk で獲得したメモリのゼロクリアは特に行われない。獲得されたメモリブロックの内容は不定である。

可変長メモリプールを対象とした場合は、このシステムコールにより、（cre_mpl で指定した blksz）×bcnt の大きさのメモリブロックが確保される。また、固定長メモリプールを対象とした場合は、bcnt の指定は無視され、常に（cre_mpl で指定した blksz）の大きさのメモリブロックが確保される。

指定したメモリプールから、指定した大きさのメモリブロックがすぐに確保できない場合には、タスクはそのメモリプールのメモリ獲得待ち行列につながれ、確保できるようになるまで待つ。

tmout により待ち時間のタイムアウト指定を行なうことができる。tmout としては、正の値のみ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、メモリブロックを獲得できないまま tmout 時間が経過すると、タイムアウトエラー E_TMOUT としてエラーリターンする。また、tmout として TMO_POL を指定した場合は、メモリブロックが獲得できない場合でも即時にリターンする。この場合、メモリブロックが獲得できれば正常終了になり、メモリブロックが獲得できなければタイムアウトエラー E_TMOUT になる。さらに、tmout =TMO_FEVR によりタイムアウト指定が行なわれないことを示す。この場合は条件が満足されるまで永久に待つ。

【補足事項】

資源獲得を行うシステムコールのうち、get_blk (TMO_POL) に限っては、次のような理由により、例外的にタスク独立部からも実行可能な場合がある。(実際に実行可能かどうかはインプリメント依存である)

― 一般的には、セマフォカウント値やメモリブロックなどの「資源」は、タスクに与えられるものであるということになっている。したがって、割込みハンドラ等のタスク独立部では、資源を確保することができず、タスク独立部から wai_sem 等を発行することはできない。
― しかし、タスク独立部が資源を確保したとしても、リターンの前に資源を返してしまう(あるいは他のタスクやメイルボックスに渡してしまう)ことにすれば、特に矛盾は生じない。実際、割込みハンドラの中からメッセージを送る場合には、その前に get_blk を実行する必要が生じる。もちろん、割込みハンドラでは待ち状態になることはできないので、tmout = TMO_POL を指定しなければならない。
― get_blk 以外の wai_flg(TMO_POL), wai_sem(TMO_POL), rsv_msg(TMO_POL) などをタスク独立部から実行する必要性は無い。get_blk 以外のシステムコールに関しては、やはり E_CTX のエラーとする。

OS用メモリプール(TMPL_OS) に対して直接 get_blk, rel_blk を行う機能を提供することが考えられるが、以下の理由により、この機能は提供していない。mplid=TMPL_OS に対して get_blk, rel_blk を実行した場合には、E_RSID のエラーとなる。

― 機能が増えて仕様が複雑化する。また、メモリプールの位置付けが曖昧になる。
「メモリブロックは必ずメモリプールから確保する」という原則が崩れる。また、OS用メモリプール (TMPL_OS) のブロックサイズをユーザ側から自由に指定することはできないので、ブロックサイズの値に関しては、ユーザ側で特殊な扱いをする必要が生じる。
たとえば、システム(初期化時の設定値など)とユーザプログラムとの間でブロックサイズの値を合意しておくか、必要メモリ容量(バイト数)と mpl_sts の実行結果から、ユーザプログラムの側で動的に要求ブロック数 bcnt を計算する必要がある。

— MMU をサポートした場合には、この機能を導入することによってインプリメントの負担が増える可能性がある。

空きメモリがOS用メモリプール(TMPL_OS) に属するという考え方は、メモリ全体に対するメモリプールの管理方法と、各メモリプール内におけるメモリブロックの管理方法とを一本化するというインプリメント方針に基づいたものである。

しかし、MMU をサポートした場合には、マッピング（ページテーブル）の設定などが必要になるので、メモリプールあるいは空きメモリの管理と、メモリブロックの管理とを一本化するのは無理がある。すなわち、TMPL_OS に対して get_blk, rel_blk を行う機能を導入した場合、それだけのために OS 側の特殊な処理が増えてしまう。

なお、mplid = TMPL_OS (-4) のメモリプールに対して状態参照のシステムコール (mpl_sts) を発行することは可能である。

【エラーコード (ercd)】

E_OK	正常終了
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSID	予約ID番号（-4≦mplid≦0）
E_PAR	一般的なパラメータエラー（bcnt≦0）
E_IDOVR	ID範囲外（インプリメント依存）
E_ILTIME	不正時間指定（tmout≦-2）
E_NOEXS	オブジェクトが存在していない（mplidのメモリプールが存在しない）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でmplid＜(-4)）
E_CTX	コンテキストエラー（タスク独立部あるいはディスパッチ遅延中のタスクから発行,インプリメント依存）
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_TMOUT	タイムアウト
E_RLWAI	待ち状態強制解除（待ちの間にter_tsk,rel_waiを受け付け）

共有メモリブロックを返却する rel_blk

rel_blk: Release Shared Memory Block

【パラメータ】

mplid	MemoryPoolIdentifier	メモリプールID
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】

なし

【解説】

blk で示されるメモリブロックを、mplidで示されるメモリプールへ返却する。

メモリブロックの返却を行うメモリプールは、メモリブロックの獲得を行ったメモリプールと同じものでなければならない。メモリブロックを、前と異なるメモリプールへ返却していることが検出された場合には、E_ILBLK のエラーとなる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦mplid≦0）
E_IDOVR	ID範囲外（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（mplidのメモリプールが存在しない）
E_ILBLK	不正メモリブロックの返却、操作
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でmplid＜(-4)）

メモリプールの状態を参照する　mpl_sts

mpl_sts:　Get MemoryPool Status

【パラメータ】

mplid	MemoryPoolIdentifier	メモリプールID
pk_mpls	Packet of MemoryPoolStatus	メモリプール状態を返すパケットの先頭アドレス

【リターンパラメータ】

なし

【解説】

mplid で示されたメモリプールの状態を読み出し、その結果を pk_mpls 以下の領域に返す。

pk_mpls に返される情報としては、次のようなものがある。

```
mplatr      /* メモリプール属性 */
wtskid      /* 待ち行列先頭のタスクID */
frbcnt      /* 空き領域全体のブロック数 */
maxbcnt     /* 最大の連続空き領域のブロック数 */
mplsz       /* メモリプール全体のサイズ */
blksz       /* メモリブロックのサイズ */
fragcnt     /* 空きブロックのフラグメント数 */
```

wtskid には、待ち行列の先頭のタスクのIDが返る。すなわち、属性が TA_TFIFO であれば、現在このメモリプールを待っているタスクのうち、最も早く待ち状態になったタスクのIDが返る。また、属性が TA_TPRI であれば、現在このメモリプールを待っているタスクのうち、最も優先度の高いタスクのIDが返る。

frbcnt はメモリの空き領域の合計ブロック数であり、maxbcnt は空き領域のうちで最大の連続領域のブロック数である。固定長メモリプールの場合は、maxbcnt は意味を持たず、常に1が返される。また、mplsz, blkszには、cre_mpl で指定した値がそのまま返される。

fragcnt は、メモリ空きブロックのフラグメント数（空きブロックがいくつの連続領域に分かれているか）を示す情報である。この情報は、対象メモリプールからどの程度の大きさの連続ブロックが取れるかということを知るための目安になる。固定長メモリプールに対する fragcnt の値としては、frbcnt と同じ値がセットされる。

対象となるメモリプールは、既に生成されたものでなければならない。

mplid = TMPL_OS (-4) のメモリプールに対して mpl_sts を発行することにより、OSの残りのメモリ容量などを知ることができる。この場合、pk_mpls には次のような値を返す。

mplatr	不定（意味がない）
wtskid	不定 （意味がない）
frbcnt	空き領域全体のブロック数 frbcnt*blksz により空き領域全体のバイト数になる
maxbcnt	最大の連続空き領域のブロック数 maxbcnt*blksz により最大の連続空き領域のバイト数になる
mplsz	OS用メモリプール全体のバイト数 ITRON2で管理しているメモリ全体のバイト数
blksz	インプリメント依存 MMUをサポートしない場合は、1あるいはOSのインプリメントで使用している具体的なメモリブロックサイズになる。 MMUをサポートしている場合は、ページサイズになる。
fragcnt	空き領域のフラグメント数

blksz がインプリメント依存であるため、ユーザ側でメモリの空き領域のサイズを調べるためには、必ず(frbcnt×blksz) の演算を行う必要がある。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号 (-3≦mplid≦0、ただし(-3)≦mplid≦(-2)はインプリメント依存)
E_ILADR	不正アドレス（pk_mplsが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（mplidのメモリプールが存在しない）

E_OACV　オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でmplid＜(-4)）
※ mplid=TMPL_OS(-4) に対する get_blk, rel_blk, mpl_sts をユーザタスクから発行することは可能である。

共有メモリブロックの状態参照

blk_sts

blk_sts: Get Shared Memory Block Status

【パラメータ】

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】

mplid	MemoryPoolIdentifier	メモリプールID
bcnt	ContinuousBlockCount	連続領域のブロック数

【解説】

blk で示されるメモリブロックの状態を参照する。具体的には、そのメモリブロックの属するメモリプールID mplid と、メモリブロックの大きさ（連続ブロック数）bcnt を返す。

固定長メモリプールから得られたメモリブロックに対して blk_sts が実行された場合には、bcnt として1が返る。

【エラーコード (ercd)】

E_OK	正常終了
E_ILBLK	不正メモリブロックの返却、操作

時間管理機能

システムクロックを設定する

set_tim

set_tim: Set Time

【パラメータ】

utime	UpperCurrentDateTime	現在の年月日時刻データ(上位)
ltime	LowerCurrentDateTime	現在の年月日時刻データ(下位)

【リターンパラメータ】

なし

【解説】

システムが保持しているシステムクロックの現在の値を、utime, ltime で示される値に設定する。システムクロックは、48 ビット(utime:16 bit + ltime :32 bit) で表現されるのが標準である。

システムクロックの意味や単位としては、1985年1月1日0時 (GMT) からの通算のミリ秒数とすることを推奨する。

32ビットの時間指定パラメータ（タイムアウト等）、48ビットの時間指定パラメータ（絶対時間等）とも、符号付きの数として扱われる。したがって、set_tim の utime で負の数を指定した場合には E_ILTIME のエラーとなる。

システムの動作中に set_tim を使ってシステムクロックを変更した場合には、それまで時間待ちをしていたタスクや、起動を待っていたハンドラ（周期起動ハンドラやアラームハンドラ）の動作のタイミングが狂う可能性がある。このような場合の動作はインプリメント依存である。これは、OS 内部の時間管理で相対時間を使っているような場合に、set_tim でこれらの時間の補正を行ったり、過ぎた時刻を待つタスクを待ち解除にしたりするのは難しい可能性があるためである。

【エラーコード (ercd)】

E_OK	正常終了
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_ILTIME	不正時間指定（utime,ltimeが不正）

システムクロックの値を読み出す get_tim

get_tim: Get Time

【パラメータ】

なし

【リターンパラメータ】

utime	UpperCurrentDateTime	現在の年月日時刻データ（上位）
ltime	LowerCurrentDateTime	現在の年月日時刻データ（下位）

【解説】

システムが保持しているシステムクロックの現在の値を読み出し、utime, ltime に返す。システムクロックは、48 ビット (utime:16bit+ltime:32bit) で表現されるのが標準である。

システムクロックの意味や単位としては、1985年1月1日0時 (GMT) からの通算のミリ秒数とすることを推奨している。

【エラーコード (ercd)】

E_OK	正常終了
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。

タスクの遅延を行う

dly_tsk

dly_tsk: Delay Task

【パラメータ】

dtime	DelayTime	遅延時間

【リターンパラメータ】

なし

【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、dtime により指定される。

このシステムコールは、wai_tsk とは異なり、遅延して終了する方が正常終了になる。また、遅延期間中にwup_tsk が実行されても、待ち解除とはならない。指定時間以前に dly_tsk が終了するのは、ter_tsk やrel_wai が発行された場合のみである。

このシステムコールを発行したタスクがSUSPEND状態（WAIT-SUSPEND状態）になっている間も、時間経過のカウントは行われる。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足 ※ このエラーが発生するかどうかはインプリメント依存である。
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_ILTIME	不正時間指定(dtime＜0)
E_CTX	コンテキストエラー（タスク独立部あるいはディスパッチ遅延中のタスクから発行）
E_RLWAI	待ち状態強制解除（待ちの間にter_tsk,rel_waiを受け付け）

タスクを周期起床する

cyc_wup

cyc_wup:　Cyclic Wakeup Task

【パラメータ】

tskid	TaskIdentifier	タスクID
rptcnt	RepeatCount	起床回数
cytime	CycleTime	周期起床時間間隔
utime	UpperInitialTime	初期起床時刻（上位）
ltime	LowerInitialTime	初期起床時刻（下位）
tmmode	TimeMode	初期時刻指定モード

【リターンパラメータ】

なし

【解説】

tskid で示されたタスクの周期起床を行なう。すなわち、対象タスクに対して、初期指定時刻から一定時間毎に起床要求（wup_tsk と同じもの）を発行する。tskid = TSK_SELF により自タスクの指定になる。

utime, ltime により、最初に起床要求を出す時間を指定する。標準では、utime16ビット、ltime 32ビットの計48ビットを使って時刻の指定が行われる。tmmode により、次のようにして絶対時刻と相対時刻の指定が可能である。

tmmode : = (TTM_ABS ‖ TTM_REL)

TTM_ABS	絶対時刻での指定
TTM_REL	相対時刻での指定

指定した時刻が過去の時刻であった場合には、エラー E_ILTIME となる。

rptcnt = 0 で無限回数の指定となる。この場合は、can_cyc により要求が解除されるか、対象タスクが削除されるまで、永久に起床要求を出し続ける。rptcnt < 0 は reserved であり、指定した場合にはエラー E_PARとなる。

cytime は周期起床の時間間隔であり、標準では 32 ビットで表現される。cytime ≦ 0 も reserved である。cytime ≦ 0 の指定を行った場合には、エラー E_ILTIME となる。

周期的なタスクの実行は、本システムコールと slp_tsk システムコールを

組み合わることによって実現される。

rptcnt = 1 により、指定時刻に一度だけタスクを起床するという遅延起床の機能が実現できる。この場合、cytime のパラメータは意味を持たないことになるが、E_ILTIME のエラーを避けるため、何らかの正の値を指定しておく必要がある。

cyc_wup システムコールは、時間が経過してから実際の動作をするので、wup_tsk とは異なり、システムコール発行時に対象タスクが DORMANT 状態であったとしてもエラーとはしない。また、時間が経過して実際に起床要求が出る時になって、対象タスクが DORMANT 状態や存在しない状態であったり、起床要求カウントのオーバーフローエラー E_QOVR が検出されたりしても、エラーの通知先がない。cyc_wup により、こういった通知先のないエラーが発生した場合、システムのエラーログ用のメッセージバッファが用意されていれば、そこにエラーが記録される。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足 ※ このエラーが発生するかどうかはインプリメント依存である。
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSMD	予約モード、予約オプション（tmmodeが不正）
E_RSID	予約ID番号（-4≦tskid≦-1、タスク独立部の発行でtskid=0）
E_PAR	一般的なパラメータエラー(rptcnt＜0)
E_IDOVR	ID範囲外（インプリメント依存）
E_ILTIME	不正時間指定（cytime≦0、utime,ltimeが不正）
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_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でtskid＜(-4)）

※ 以下のエラーは、cyc_wup の実行時にすぐに検出されるものではなく、時間が経過してから発生する可能性のあるエラーである。

E_DMT	タスクがDORMANTである（tskidのタスクがDORMANT）
E_QOVR	キューイングのオーバーフロー（wupcntのオーバーフロー）

タスクに出された周期起床要求を解除する can_cyc

can_cyc: Cancel Cyclic Wakeup Task

【パラメータ】

tskid	TaskIdentifier	タスクID

【リターンパラメータ】

なし

【解説】

tskid で示されるタスクに対して発せられている周期起床要求を取り消す。

対象タスクに複数の周期起床要求が出ていても、それらはすべて取り消される。周期起床要求の出ていないタスクに対してこのシステムコールを発行すると、エラー E_NOCYC となる。

tskid = TSK_SELF により自タスクの指定になる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦tskid≦-1、タスク独立部の発行でtskid=0）
E_IDOVR	ID範囲外（インプリメント依存）
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_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でtskid＜(-4)）
E_NOCYC	タスクに対してcyc_wupが発行されていない

システム管理機能

拡張SVCハンドラを定義する

def_svc

def_svc:　Define SupervisorCall Handler

【パラメータ】

s_fncd	SVCFunctionCode	拡張機能コード
svhatr	SVCHandlerAttribute	拡張SVCハンドラ属性
svchdr	SVCHandlerAddress	拡張SVCハンドラアドレス

【リターンパラメータ】

なし

【解説】

ユーザが書いた拡張システムコールハンドラ（拡張SVCハンドラと呼ぶ）を定義する。拡張SVCハンドラは、ITRONのシステムコールを拡張する働きを持つ。

def_svc では、ユーザ定義のサービスルーチンである拡張SVCハンドラをOSに登録し、svchdr で示される拡張SVCハンドラのアドレスと、s_fncd で示される拡張機能コードとの対応付けを行なう。def_svc 実行の結果、ITRONのシステムコールを呼び出すのと同様のインタフェースで、その拡張SVCハンドラが実行されるようになる。

svhatr のサイズは標準で4バイトである。svhatr のフォーマットを［図3.28］に示す。このうち、最上位バイト（bit0～bit7 あるいは 2^31～2^24 のビット）は未使用であり、第二上位バイト（bit8～bit15 あるいは 2^23～2^16 のビット）はインプリメント依存の属性を表わし、下位ハーフワード（bit16～bit31 あるいは 2^15～2^0 のビット）はシステム属性を表わす。

svhatr のシステム属性の部分では、次のような指定を行うことができる。

svhatr : = (TA_ASM ‖ TA_HLNG)

TA_ASM	ハンドラがアセンブラで書かれている
TA_HLNG	ハンドラが高級言語で書かれている

TA_HLNG の指定を行った場合には、拡張SVCハンドラ起動時に直接 svchdr 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

［図3.28］svhatrのフォーマット

ム（高級言語対応ルーチン）を通してから svchdr のアドレスにジャンプする。

svchdr = NADR (-1) の指定により、前に定義されていた拡張SVCハンドラの定義が解除され、未定義の状態になる。ただし、前に定義されていた拡張SVCハンドラの定義解除を行わずに、直接新しいハンドラを定義しても、エラーにはならない。また、もともとハンドラが定義されていない状態（定義が解除された状態）で、svchdr=NADR の指定による定義解除の操作を行っても、エラーにはならない。後者の場合、このシステムコールは正常終了するが、拡張SVCハンドラは未定義のままである。

拡張SVCハンドラ起動時には、常に終了ハンドラの起動がマスクされた状態になっている。拡張SVCハンドラの中で終了要求を受けた時に、sdef_ext で定義した終了ハンドラを起動したい場合には、拡張SVCハンドラの中で、終了要求に対する clr_ems を実行しておく必要がある。また、拡張SVCハンドラ起動時には、強制例外ハンドラの起動もマスクされた状態になっている。拡張SVCハンドラ実行中は、clr_ems を使っても強制例外に対する例外マスクを解除することはできず、強制例外ハンドラの起動は拡張SVCハンドラを抜けるまで遅らされる。

なお、SVCハンドラといった名称は、このシステムコールで定義されるサービスルーチンに限って用いられており、普通のシステムコールは SVCハンドラとは呼ばない。これは、ユーザの定義する拡張システムコールと、ITRONの仕様書に載っている標準システムコールを別の名称で扱うためである。

一般に、標準システムコールは、ユーザにとってそれが不可分に実行されるように見えなくてはならない。例えば、システムコール実行中に割込みが入った場合でも、システムコールの実行前、または実行後に割り込まれたのと同じ動作をする必要がある。これに対して、拡張システムコールでは内部が全部ユーザに見えているので、「不可分の実行」というわけにはいかない。また、標準システムコールの処理はOSが行なうため、OS内部のデータ構造（レディキュー、ウエイトキューなど）を自由に操作できるのに

対して、拡張SVCハンドラではこれらのデータを直接操作することはできず、OS内部の状態を変更するには、必ず標準システムコールを通す必要がある。標準システムコールと拡張システムコールでは、この2つの点で大きな違いがあるので、注意する必要がある。

拡張SVCハンドラの運用上の問題として、内部で待ち状態に入る拡張SVCハンドラは、タスク独立部から呼び出された場合に、E_CTX のエラーを返すのが望ましい。タスク独立部から拡張SVCハンドラが呼ばれ、その中で待ちに入るシステムコールを発行した場合には、自然に E_CTX のエラーが返るので、拡張SVCハンドラ内で E_CTX のエラーが起きた場合にそれをそのまま要求元に返せば、結果的にこのようなチェックが行われることになる。なお、タスク独立部から拡張SVCハンドラを呼び出す機能はインプリメント依存である。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足 ※ このエラーが発生するかどうかはインプリメント依存である。
E_ILFN	不正機能コード番号（s_fncd≦0）
E_RSATR	予約属性（svhatrが不正）
E_ILADR	不正アドレス（svchdrが奇数、あるいは使用できない値）

拡張SVCハンドラから復帰する

ret_svc

ret_svc: Return from SupervisorCall Handler

【パラメータ】

s_ercd	SVCErrorCode	SVCからのエラーコード

【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

【解説】

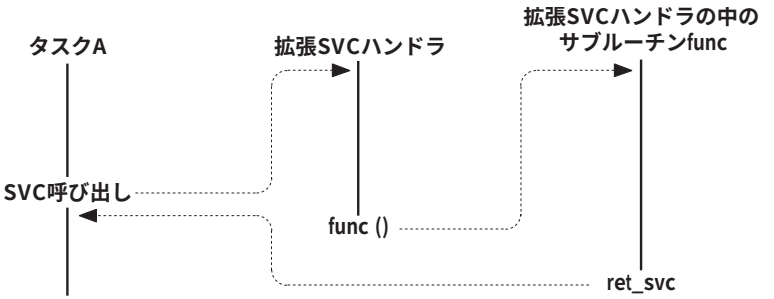
拡張SVCハンドラから、要求元のタスクまたはハンドラに戻る。要求元には、リターンコード（エラーコード）として s_ercd の値が返される。s_ercd は、正の値でも良い。

拡張SVCハンドラの中から、拡張システムコールを実行することも可能である。すなわち、拡張SVCハンドラはネストすることができる。

ret_svc で要求元のシステムコール例外を起動したい場合には、s_ercd の 2^8～2^12 のビットに、起動すべきシステムコール例外の例外クラス番号の補数を設定する。この場合、拡張SVCハンドラを呼んだコンテキストの例外マスク環境において、s_ercdで指定した例外クラスに対するマスクがクリアされていれば、システムコール例外が起動される。s_ercd が正の場合には、システムコール例外は起動されない。(エラーコードの説明の章を参照)

また、s_ercd で指定したエラーコードの例外クラスが EC_TER, EC_FEX, EC_CEX であった場合にも、システムコール例外は起動されない。すなわち、終了例外、強制例外、CPU例外に相当するエラーコードを使ってシステムコール例外を発生することはできない。たとえば、拡張SVCを呼んだ側の例外マスクの ECM_TER がクリアされた状態で ret_svc(s_ercd=E_TER) を実行しても、システムコール例外は発生しない。

拡張SVCハンドラや例外（終了）ハンドラの中でサブルーチンを呼び、そのサブルーチンの中で直接 ret_svc や ret_exc を実行した場合でも、エラーとはならず、そのまま拡張SVCハンドラや例外（終了）ハンドラ起動前のコンテキストに復帰するものとする。すなわち、サブルーチンに対するスタ



［図3.29］サブルーチンの中で実行したret_svc

ックを残したまま、ret_svc やret_exc によって直接拡張SVCハンドラや例外ハンドラから戻ることが可能である。この場合、ret_svc やret_exc の処理の中で、自動的にサブルーチンに対するスタックフレームの解放が行われることになる。

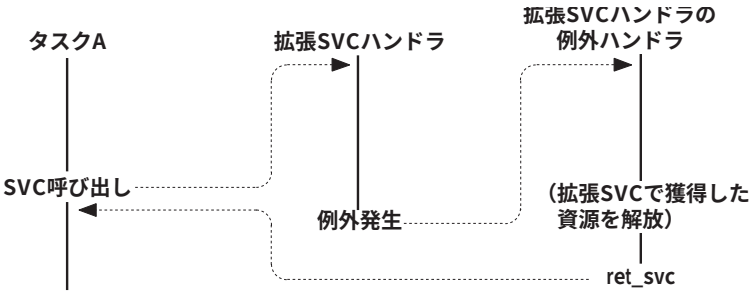
拡張SVCハンドラからサブルーチンを呼び、その中で ret_svc を実行した場合の動作は、［図3.29］のようになる。

また、拡張SVCに対する例外（終了）ハンドラで ret_svc を発行した場合は、まず例外（終了）ハンドラから拡張SVCハンドラへの環境復帰が行われ、次に元のコンテキストへの環境復帰が行われる。エラーとはならない。この場合、ret_svc のみで、end_exc+ret_svc と同等の動作を行うことになる。この機能は、拡張SVC中で回復不可能なエラーが発生し、拡張SVCを続行しても意味が無いので元のタスクに戻る場合や、拡張SVC実行中に終了要求を受け付けたので、拡張SVCを中断して元のタスクに戻る場合（この時タスクの終了ハンドラが起動される）などに有効である。

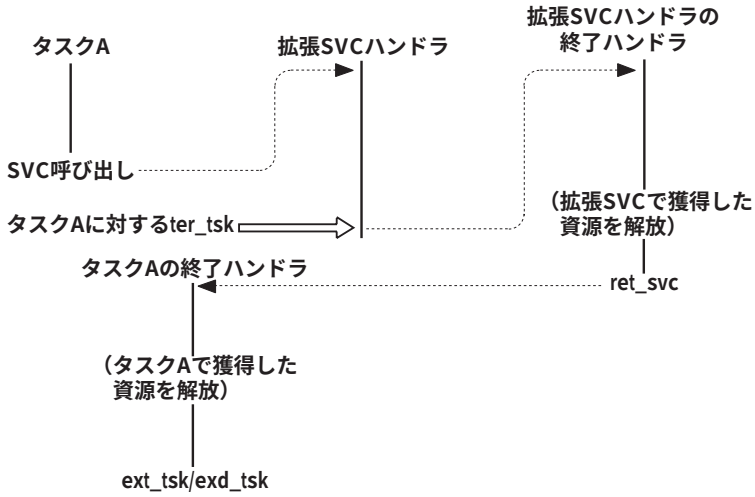
拡張SVCハンドラ実行中に例外が発生し、拡張SVCハンドラに対する例外ハンドラの中で ret_svc を実行した場合の動作は、［図3.30］のようになる。

また、拡張SVCハンドラ実行中に終了要求が受け付けられ、拡張SVCハンドラに対する終了ハンドラの中でret_svc を実行した場合の動作は、［図3.31］のようになる。なお、タスクA は終了ハンドラの起動をマスクしていない（ECM_TER をクリアしている）ものとする。

一方、拡張SVCが終了ハンドラの起動をマスクしている間に終了要求を受け付け、終了要求がペンディング状態のまま ret_svc を実行した場合、



[図3.30] 拡張SVCハンドラに対する例外ハンドラ中で実行したret_svc



[図3.31] 拡張SVCハンドラに対する終了ハンドラ中で実行したret_svc

ret_svc の実行後も拡張SVCハンドラに対する終了ハンドラは起動されない。

この場合、拡張SVCから戻る前の環境では終了ハンドラの起動がマスクされており、また拡張SVCから戻った後の環境では、例外ハンドラ定義環境も元のコンテキスト（タスク）のものに戻っている。したがって、例外マスク環境と例外ハンドラ定義環境の切り換えが不可分に行われると考えれば、拡張SVCに対する終了ハンドラが起動されることはない。終了要求をマスクしておけば、abo_tsk 発行時を除いて、その環境に対する終了ハンドラが起動されることは無いということになる。

【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には、メッセージバッファへのロギングや、コンソールへのエラーメッセージの表示などを行うのが望ましいが、詳細な動作はインプリメント依存となる。

E_CTX	コンテキストエラー（拡張SVCハンドラ実行中ではない）
E_PAR	一般的なパラメータエラー（s_ercdが不正）

ITRONのバージョン番号を得る get_ver

get_ver: Get Version NO

【パラメータ】

pk_ver	Packet of Version Numbers	バージョン管理情報を返すパケットの先頭アドレス

【リターンパラメータ】

なし

【解説】

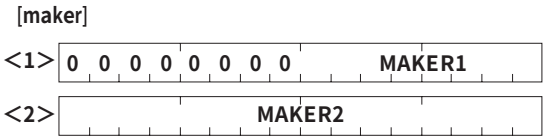
現在実行中のITRON仕様OSやμITRON仕様OSのスペックの概要、OSの形式番号、バージョン番号などを得る。

このシステムコールは、OSのメーカーおよびバージョン番号を、プログラムから読み出すことを目的としたものである。

バージョンの読み出しは、次のようなパケットを用いて行なわれる。なお、以下で、UH は 16ビット符号無し整数のデータタイプを表わす。

```
typedef struct t_ver {
        UH      maker;          /* メーカー */
        UH      id;             /* 形式番号 */
        UH      spver;          /* 仕様書バージョン */
        UH      prver;          /* 製品バージョン */
        UH      prno[4];        /* 製品管理情報 */
        UH      cpu;            /* CPU情報 */
        UH      var;            /* バリエーション記述子 */
} T_VER;

T_VER           *pk_ver; /* バージョン管理ブロックへのポインタ */
```

パケットの形式や構造体の各メンバのフォーマットは、プロセッサ間で、あるいはITRON1、ITRON2、μITRON、BTRONの間でほぼ共通になっている。以下では、各メンバのフォーマットや意味について説明を行う。



MAKER : メーカー番号

B' 00000000	バージョンなし （実験システムなど）
B' 00000001	University of TOKYO
...	＊以下はABC順
B' 00001001	FUJITSU
B' 00001010	HITACHI
B' 00001011	MATSUSHITA
B' 00001100	MITSUBISHI
B' 00001101	NEC
B' 00001110	OKI
B' 00001111	TOSHIBA
B' 00010000〜B' 11111111	reserved

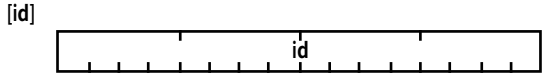
[図3.32 (a)] get_verで得られるmakerのフォーマット

maker では、この製品 (ITRON、BTRON、TRONCHIP) を作ったメーカーを表わす。maker のフォーマットを［図3.32 (a)］に示す。

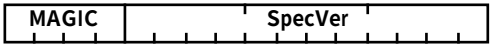
ITRON、BTRON、TRONCHIPのうちの複数のプロジェクトに参加しているメーカーに対しては、協議の上［図3.32 (a)］＜1＞のフォーマットを割り当てる。この場合、MAKER1 のコードは、ITRON、BTRON、TRONCHIP で共通になる。それ以外のメーカーの場合は、［図3.32 (a)］＜2＞のフォーマットを使用し、B'0000000100000001〜 のコードを割り当てる。MAKER2 のコードは登録制となる。MAKER2 のコードは、ITRON、BTRON、TRONCHIPの間で異なったものになる。

なお、MAKER1 のコードを持つメーカーの子会社や関連会社の場合は、そのメーカーの判断によって、MAKER1のコード番号を使うか、新しく登録する MAKER2 のコードを使うかということを決める。

id は、OS や VLSI の種類を区別する番号である。id のフォーマットを［図3.32 (b)］に示す。1つのメーカーの中で、MMU対応版とMMU非対応版のような複数の種類の製品を作った場合、それらを区別するためにidを用いることができる。



[図3.32 (b)] get_verで得られるidのフォーマット



MAGIC : TRONのシリーズを区別する番号

H' 0	TRON共通（TAD等）
H' 1	ITRON1, ITRON2
H' 2	BTRON
H' 3	CTRON
H' 4	reserved
H' 5	μITRON
H' 6	μBTRON

SpecVer : この製品のもとになったTRON仕様書のバージョン番号。
3桁のパックト形式コードで入れる。

[図3.32 (c)] get_verで得られるspverのフォーマット

idの番号の付け方はメーカーの自由とする。ただし、製品の区別はあくまでもこの番号のみで行なうので、各メーカー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。

spverでは、ITRON, μITRON, BTRON, CTRON, TRONCHIPの区別と、この製品のもとになった仕様書のバージョン番号を表わす。spver のフォーマットを［図3.32 (c)］に示す。

ITRON2の最初のバージョンに対応する spver は次のようになる。

```
MAGIC   = H'1              (ITRON)
SpecVer = H'200            (Ver 2.00)
spver   = H'1200
```

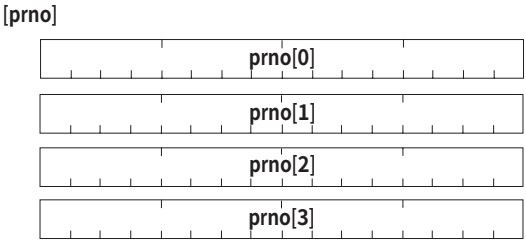
また、たとえば ITRON2 Ver 2.34.xx.xx の仕様書をインプリメントした製品の場合、

```
spver = H'1234
```

となる。

prverでは、内部インプリメント上のバージョン番号を表わす。prver のフ

[図3.32 (d)] get_verで得られるprverのフォーマット



[図3.32 (e)] get_verで得られるprnoのフォーマット

ォーマットを［図3.32 (d)］に示す。MAJOR はバージョン番号の大きな区別、MINOR はバージョン番号の小さな区別を表わす。番号の付け方はメーカーの自由である。

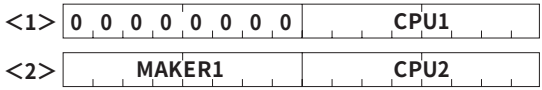
prno は、製品管理情報や製品番号などを入れるために、各メーカーが自由に使用してもよい部分である。prnoのフォーマットを［図3.32 (e)］に示す。

cpu は、このITRONを実行するプロセッサを表わす。cpu のフォーマットを［図3.32 (f)］に示す。TRONCHIP、モトローラM68000系、インテルiAPX86系、ナショセミNS32000系、NEC-Vシリーズのプロセッサの場合は［図3.32(f)］＜1＞のフォーマットを使用し、それ以外の各社独自アーキテクチャのプロセッサの場合には、［図3.32(f)］＜2＞のフォーマットを使用する。

［図3.32(f)］＜1＞のフォーマットにおける CPU1 のコード割当ては、ITRONとBTRONで共通である。また、BTRON/CHIP標準オブジェクトフォーマットのCPUタイプの部分にも、これと同じ番号が入る。

一方、［図3.32(f)］＜2＞のフォーマットは、主としてμITRON で用いられるものである。この場合、MAKER1 のコードの割当ては、maker の項で説明したものと同じになる。また、CPU2 のコードの割当ては MAKER1 のメーカーが決める。

[cpu]



CPU1:

B'00000000 (H'00) CPUを特定しない、CPU情報を持たない
B'00000001 (H'01) TRONCHIP32共通
B'00000010 (H'02) reserved
B'00000011 (H'03) reserved
B'00000100 (H'04) reserved
B'00000101 (H'05) reserved（《L1R》仕様のTRONCHIIP）
B'00000110 (H'06) reserved（《L1》仕様のTRONCHIIP）
B'00000111 (H'07) reserved（LSIDの機能を持つTRONCHIIP）

＊B' 00000000～00000111 の場合、CPUのメーカーを特定しない。

B'00001000 (H'08) rreserved
B'00001001 (H'09) GMICRO/100
B'00001010 (H'0a) GMICRO/200
B'00001011 (H'0b) GMICRO/300
B'00001100 (H'0c) reserved
B'00001101 (H'0d) TX1
B'00001110 (H'0e) reserved
B'00001111 (H'0f) TX3

B'00010000 (H'10) reserved
B'00010001 (H'11) reserved
B'00010010 (H'12) reserved
B'00010011 (H'13) 032

B'00010100 (H'14) reserved
B'00010101 (H'15) MN10400
B'00010110 (H'16) reserved
B'00010111 (H'17) reserved

B'00011000～B'00111111(H'18～H'3f) - reserved

＊GMICRO拡張用、TXシリーズ拡張用、TRONCHIP64用

［図3.32 (f)］get_verで得られるcpuのフォーマット（前半）

```
B'01000000 (H'40) 68000
B'01000001 (H'41) 68010
B'01000010 (H'42) 68020
B'01000011 (H'43) 68030
B'01000000～B'01001111(H'40～H'4f) - 6800 系


B'01010000 (H'50) 32032
B'01010000～B'01001111(H'50～H'5f) - NS32000 系


B'01100000 (H'60) 8086. 8088
B'01100001 (H'61) 80186
B'01100010 (H'62) 80286
B'01100011 (H'63) 80386
B'01100000～B'01101111(H'60～H'6f) - 86 系


B'01110000～B'01111111(H'70～H'7f) - NEC Vシリーズ
        ＊割り当てはVシリーズ関連メーカーが決める。


B'10000000～B'11111111(H'80～H'ff) - reserved
```

［図3.32 (f)］get_verで得られるcpuのフォーマット（後半）

var では、このITRON2、μITRONで利用できる機能の概要を表わす。varのフォーマットを［図3.32 (g)］に示す。

なお、製品の種類の区別はあくまでも idの部分で行なう。var, cpu による表示は便宜的なものに過ぎない。すなわち、var, cpu が異なり、かつ id の等しい製品が存在してはいけない。

【エラーコード (ercd)】

E_OK	正常終了
E_ILADR	不正アドレス（pk_verが使用できない値）

[var]

－ : reserved（0 が返る）
LEV: カーネル仕様のレベル分け
B' 000 μITRON
B' 001 reserved
B' 010 reserved
B' 011 ITRON2基本仕様（≪I1≫仕様）
B' 100 ITRON2一部拡張仕様（一部≪I2≫仕様）
B' 101 ITRON2拡張仕様（≪I2≫仕様）
M=1: マルチプロセッササポート
B' 110 reserved
M=0: シングルプロセッサ用
B' 111 reserved
V=1: 仮想記憶サポート
P=1: MMU対応版
B' 000 サポートなし
FIL: ファイル仕様のレベル分け
B' 001 ≪FI0≫仕様
B' 010 ≪FI1≫仕様
B' 011～111 reserved
B' 00 サポートなし
IO: 入出力仕様
B' 01 標準仕様
B' 10 reserved
B' 11 reserved

[図3.32 (g)] get_verで得られるvarのフォーマット

プロセッサ状態語を参照する
psw_sts

psw_sts: Get Processor Status Word

【パラメータ】

なし

【リターンパラメータ】

psw	Processor Status Word	プロセッサ状態語

【解説】

CPU のプロセッサ状態語 (PSW) を参照する。

一般に、このシステムコールにより、次のような情報を得ることができる。

― マスクされている割込みの優先度（割込みマスク、割込みレベル）
― 現在実行中のリングやモード（スーパバイザモード等）

また、これらの情報により、現在実行中のコンテキストがタスク独立部なのか、準タスク部なのか、タスク部なのかを知ることができる。

プロセッサによっては、PSW のうちの一部のフィールドしか意味を持たない場合や、PSW のうちの一部のフィールドしか参照できない場合がある。

【エラーコード (ercd)】

E_OK	正常終了

第三部 第三章

ITRON2拡張機能

この章では、ITRON2で拡張、追加された機能（≪I2≫レベルのシステムコール）に関する説明を行う。

拡張同期・通信機能

メッセージバッファを生成する cre_mbf

cre_mbf: Create MessageBuffer

【パラメータ】

mbfid	MessageBufferIdentifier	メッセージバッファID
mbfatr	MessageBufferAttribute	メッセージバッファ属性
bufsz	BufferSize	メッセージバッファのサイズ
maxbmsz	MaxBufferedMessageSize	メッセージの最大長

【リターンパラメータ】

なし

【解説】

cre_mbf では、mbfid で指定されたID番号を持つメッセージバッファを生成する。次に、生成された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。

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り、リングバッファにより構成される。メイルボックス (mbx) との大きな違いは、送信時と受信時にメッセージの内容がコピーされるということである。リアルタイム性確保のため、生成時にメッセージの最大長を指定し、それ以上のサイズのメッセージを送信した場合はエラー E_SZOVR とする。リングバッファの大きさは bufsz で指定される。メッセージの順番は FIFO のみである。

ID 番号が (-4)〜0 のメッセージバッファは生成できない。また、負の ID 番号のメッセージバッファは、システム用のものである。このうち、特に mbfid=TMBF_OS (-4) のメッセージバッファは、システム (OS) のエラーログ用（どこにも通知できないエラーの記録用）として用いられる。また、mbfid = TMBF_DB (-3) のメッセージバッファは、デバッグサポート機能との情報交換のために用いられる。

mbfatrのサイズは標準で4バイトである。mbfatrのフォーマットを［図3.33］に示す。このうち、最上位バイト（bit0〜bit7 あるいは 2^31〜2^24 のビット）

はユーザ属性を表わし、第二上位バイト（bit8〜bit15 あるいは 2^23〜2^16 のビット）はインプリメント依存の属性を表わし、下位ハーフワード（bit16〜bit31 あるいは 2^15〜2^0 のビット）はシステム属性を表わす。ユーザは、メッセージバッファに関する情報を入れておくために、mbfatr の最上位バイトを自由に使用することができる。

cre_mbf で指定した mbfatr は、mbf_sts により読み出すことができる。

mbfatr のシステム属性の部分では、次のような指定を行うことができる。

mbfatr : = (TA_TFIFO ‖ TA_TPRI)

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_TFIFO の属性を指定した場合には、メッセージを待つタスクは FIFO の待ち行列を作り、TA_TPRI の属性を指定した場合には、メッセージを待つタスクはタスクの優先度順の待ち行列を作る。メッセージの待ち行列は FIFO のみである。

ITRONの基本方針として、TCB を軽くするために、できるだけタスク従属の機能を設けないという考え方がある。メッセージバッファをタスク従属にしないのは、このためである。また、エラーログの目的では、タスク独立にした方が良い。

また、メイルボックスとメッセージバッファは、似た用途で使用されるが、別のオブジェクトとして扱っている。これは、両者の内部構成、使い方、システムコールのパラメータなどがかなり異なっているためである。

［図3.33］mbfatrのフォーマット

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足（管理ブロック用の領域が確保できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_NOMEM	メモリ不足（リングバッファ用の領域が確保できない） ※ E_NOSMEMとE_NOMEMとの厳密な区別はインプリメント依存である。
E_RSID	予約ID番号（-4≦mbfid≦0）
E_RSATR	予約属性（mbfatrの下位3バイトが不正）
E_PAR	一般的なパラメータエラー（bufsz,maxbmszが不正）
E_IDOVR	ID範囲外（mbfidがシステムで利用できる範囲を越えた）
E_EXS	オブジェクトが既に存在している（同一ID番号のメッセージバッファが存在）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でmbfid＜(-4)）

メッセージバッファを削除する del_mbf

del_mbf: Delete MessageBuffer

【パラメータ】

mbfid	MessageBufferIdentifier	メッセージバッファID

【リターンパラメータ】

なし

【解説】

mbfid で示されるメッセージバッファを削除する。

そのメッセージバッファへのメッセージの到着を待っているタスクがある場合にも本システムコールは正常終了するが、メッセージの到着を待っていたタスクにはエラー E_DLT が返される。

また、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にも、del_mbf ではメッセージバッファの削除が行われ、中にあったメッセージは消滅する。

本システムコール発行後は、そのID番号のメッセージバッファを再び新しく生成することができる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号 (-4≦mbfid≦0)
E_IDOVR	ID範囲外 (インプリメント依存)
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_OACV	オブジェクトアクセス権違反 (拡張SVC以外のユーザタスクからの発行でmbfid＜(-4))

メッセージバッファへ送信する snd_mbf

snd_mbf: Send Message to MessageBuffer

【パラメータ】

mbfid	MessageBufferIdentifier	メッセージバッファID
bmsgsz	BufferedMessageSize	送信メッセージのサイズ
pk_bmsg	BufferdMessagePacket	送信メッセージの先頭アドレス

【リターンパラメータ】

なし

【解説】

mbfid で示されたメッセージバッファに、pk_bmsg 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 bmsgsz で指定される。すなわち、pk_bmsg 以下の bmsgsz バイトが、mbfid で指定されたメッセージバッファにコピーされる。

bmsgszが、cre_mbfで指定したmaxbmszよりも大きい場合は、エラーE_SZOVRとなる。また、メッセージバッファが一杯の時は、即座にエラーE_QOVRとしてリターンする。このシステムコールで待ち状態になることはない。

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

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号 (-4≦mbfid≦0)
E_PAR	一般的なパラメータエラー (bmsgsz≦0)
E_ILADR	不正アドレス (pk_bmsgが使用できない値)
E_IDOVR	ID範囲外 (インプリメント依存)
E_SZOVR	パラメータのサイズが制限を越えた (bmsgsz＞maxbmsz)

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_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でmbfid＜(-4)）
E_QOVR	キューイングのオーバーフロー（リングバッファのオーバーフロー）

メッセージバッファから受信する rcv_mbf

rcv_mbf: Receive Message from MessageBuffer

【パラメータ】

mbfid	MessageBufferIdentifier	メッセージバッファID
tmout	Timeout	タイムアウト指定
pk_bmsg	BufferedMessagePacket	受信メッセージを入れるアドレス

【リターンパラメータ】

bmsgsz	BufferedMessageSize	受信したメッセージのサイズ

【解説】

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、pk_bmsg 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先頭のメッセージが、pk_bmsg 以下の bmsgsz バイトにコピーされる。受信時にもメッセージのコピーが行われるため、pk_bmsg はリターンパラメータではなくパラメータとなっている。

mbfid で示されたメッセージバッファにまだメッセージが到着していない場合には、本システムコール発行タスクはメッセージ到着を待つ待ち行列につながれる。待ち行列へのつながれ方は、メッセージバッファ生成時に mbfatr で指定した方法によるものであり、FIFO または タスク優先度のいずれかである。

tmout により待ち時間のタイムアウト指定を行うことができる。tmout としては、正の値のみ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、条件が満足されぬまま tmout 時間が経過すると、タイムアウトエラー E_TMOUT としてエラーリターンする。また、tmout として TMO_POL を指定した場合は、メッセージが受信できない場合でも即時にリターンする。この場合、メッセージが受信できれば正常終了に、メッセージが無ければタイムアウトエラー E_TMOUT になる。さらに、tmout =TMO_FEVR によりタイムアウト指定が行われないことを示す。この場合は条件が満足されるまで永久に待つ。

【エラーコード (ercd)】

E_OK	正常終了
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSID	予約ID番号（-3≦mbfid≦0、ただし mbfid = (-3) はインプリメント依存）
E_ILADR	不正アドレス（pk_bmsgが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
E_ILTIME	不正時間指定（tmout≦-2）
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_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行で mbfid＜(-3)） ※ mbfid = TMBF_OS (-4) に対する rcv_mbf をユーザタスクから発行することはできない。
E_CTX	コンテキストエラー（タスク独立部あるいはディスパッチ遅延中のタスクから発行）
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_TMOUT	タイムアウト
E_RLWAI	待ち状態強制解除（待ちの間にter_tsk,rel_waiを受け付け）

メッセージバッファ状態を参照する mbf_sts

mbf_sts: Get MessageBuffer Status

【パラメータ】

mbfid	MessageBufferIdentifier	メッセージバッファID
pk_mbfs	Packet of MessageBufferStatus	メッセージバッファ状態を返すパケットの先頭アドレス

【リターンパラメータ】

なし

【解説】

mbfidで示されたメッセージバッファの状態を読み出し、その結果をpk_mbfs 以下の領域に返す。

pk_mbfs に返される情報としては、次のようなものがある。

```
mbfatr      /* メッセージバッファ属性 */
wtskid      /* 待ち行列先頭のタスクID */
bmsgsz      /* 次のメッセージのサイズ */
maxbmsz     /* メッセージの最大サイズ */
frbufsz     /* 空きバッファサイズ */
bufsz       /* バッファ全体のサイズ */
```

wtskid には、待ち行列の先頭のタスクのIDが返る。すなわち、属性がTA_TFIFO であれば、現在このメッセージバッファを待っているタスクのうち、最も早く待ち状態になったタスクのIDが返る。また、属性がTA_TPRI であれば、現在このメッセージバッファを待っているタスクのうち、最も優先度の高いタスクのIDが返る。待ちタスクの無い時は FALSE=0が返る。

bmsgsz には、次に受信されるメッセージのサイズが返る。メッセージが無い時は 0 が返る。bmsgsz とwtskid の少なくても一方は必ず 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。

frbufsz は、メッセージバッファ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が受け付けられるかを知る手掛かりになる。

mbfatr, maxbmsz, bufsz には、cre_mbf で指定された値がそのまま返る。

対象となるメッセージバッファは、既に生成されたものでなければならない。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号 (-3≦mbfid≦0、ただしmbfid = (-3) はインプリメント依存)
E_ILADR	不正アドレス (pk_mbfsが使用できない値)
E_IDOVR	ID範囲外 (インプリメント依存)
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_OACV	オブジェクトアクセス権違反 (拡張SVC以外のユーザタスクからの発行でmbfid＜(-3)) ※ mbfid = TMBF_OS (-4) に対する mbf_sts をユーザタスクから発行することはできない。

ランデブ用のポートを生成する cre_por

cre_por: Create Port for Rendezvous

【パラメータ】

porid	PortIdentifier	ポートID
poratr	PortAttribute	ポート属性

【リターンパラメータ】

なし

【解説】

cre_por では、porid で指定された ID 番号を持つポートを生成する。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。次に、生成されたポートに対して管理ブロックを割り付ける。

ID 番号が (-4)〜0 のポートは生成できない。また、負の ID 番号のポートは、システム用のものである。

poratr のサイズは標準で4バイトである。poratr のフォーマットを［図3.34］に示す。このうち、最上位バイト（bit0〜bit7 あるいは 2^31〜2^24 のビット）はユーザ属性を表わし、第二上位バイト（bit8〜bit15 あるいは 2^23〜2^16 のビット）はインプリメント依存の属性を表わし、下位ハーフワード（bit16〜bit31 あるいは 2^15〜2^0 のビット）はシステム属性を表わす。ユーザは、ポートに関する情報を入れておくために、poratr の最上位バイトを自由に使用することができる。

なお、poratr のシステム属性の部分は、現在はすべて reserved となってい

［図3.34］poratrのフォーマット

るため、TA_NULL (0) を入れておく必要がある。cre_por で指定した poratr は、por_sts により読み出すことができる。

ITRON2のポートにより実現されるランデブは、ADA のランデブとは異なり、タスク独立である。すなわち、cal_por によるランデブ呼び出し時には、相手タスクの指定は行わず、ポートの指定のみを行う。また、一つのポートで複数のランデブを同時に行うことも可能である。すなわち、前に成立したランデブの rpl_por が実行される前に、同じポートに対して別のタスクが acp_por を実行することも可能である。さらに、ITRON2のランデブでは、呼び出し側 (cal_por) のタスク、受け付け側 (acp_por) のタスクがともに待ち行列を作ることができる。

呼び出し側の待ち行列、受け付け側の待ち行列はともに FIFO であるが、後述する条件付きランデブの機能により、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。同一条件のタスク間でのみ FIFO の規則が適用される。条件付きランデブ機能は、ADA の選択受付機能 (select) を実現するためのものである。

ランデブの機能は、他の同期・通信機能の組み合せで実現することも可能だが、サーバ等の用途に用いた場合には、他の同期・通信機能を組み合せるよりも効率を上げることができると考えられる。

ポートをタスク独立としたのは、ITRONにおいて、タスク付属の機能をあまり入れないという方針に基づいている。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足 ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSID	予約ID番号 (-4≦porid≦0)
E_RSATR	予約属性 (poratrの下位3バイトが不正)
E_IDOVR	ID範囲外 (poridがシステムで利用できる範囲を越えた)
E_EXS	オブジェクトが既に存在している (同一ID番号のポートが存在)
E_OACV	オブジェクトアクセス権違反 (拡張SVC以外のユーザタスクからの発行でporid＜(-4))

ランデブ用のポートを削除する del_por

del_por: Delete Port for Rendezvous

【パラメータ】

porid	PortIdentifier	ポートID

【リターンパラメータ】

なし

【解説】

porid で示されるランデブ用のポートを削除する。

そのポートに対して、受付待ち (acp_por) や呼出待ち (cal_por) を行っているタスクがある場合にも、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。

del_por により、現在ランデブ中のタスクが存在するポートを削除した場合、ランデブ終了待ちのタスクには E_DLT が返り、ランデブを受け付けた側のタスク（待ち状態ではない）には何も通知されないものとする。この場合、ランデブを受け付けた側のタスクには、rpl_por を実行した段階で E_NORDV のエラーが返ることになる。

本システムコール発行後は、そのID番号のランデブ用ポートを再び新しく生成することができる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦porid≦0）
E_IDOVR	ID範囲外（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（poridのポートが存在しない）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でporid＜(-4)）

ポートに対するランデブの呼び出し cal_por

cal_por: Call Port for Rendezvous

【パラメータ】

porid	PortIdentifier	ポートID
calptn	CallBitPattern	呼出側選択条件を表わすビットパターン
pk_rmsg	RendezvousMessagePacket	メッセージアドレス
maxrmsz	Max RendezvousMessageSize	返答メッセージ受信領域のサイズ
tmout	Timeout	タイムアウト指定

【リターンパラメータ】

なし

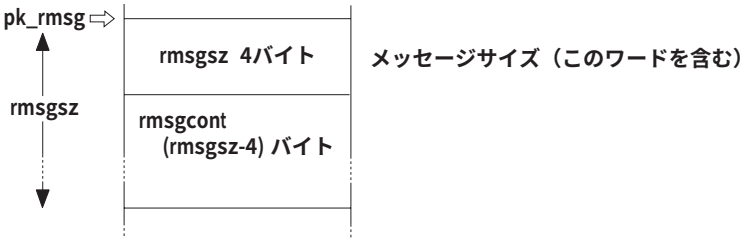
【解説】

ポートに対するランデブ呼び出しを行う。

具体的な動作は次のようになる。porid で指定したポートの受け付け待ちを行っているタスクと、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、受け付け待ちだったタスクは実行可能状態となり、cal_por 発行タスクはランデブ処理終了待ちの状態になる。また、porid で指定したポートの受け付け待ち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には、cal_por 発行タスクは呼び出し側待ち行列の最後に入り、ランデブ呼出待ちの状態となる。

ランデブ成立条件は、受付側タスクの acpptn と呼び出し側タスクの 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。

ランデブ成立時には、呼び出し側タスクから受付側タスクに対してメッセージを送ることができる。メッセージは、［図3.35］のような形式であり、cal_por ではその先頭アドレス pk_rmsg を指定する。



[図3.35] ランデブで送られるメッセージの形式

ランデブが成立した場合は、受付側タスクが acp_por の pk_rmsg パラメータで指定した領域に対して、上記のメッセージがコピーされる。

逆に、ランデブ終了時には、受付側タスクから呼びだし側タスクに対してメッセージを送ることができる。具体的には、rpl_por の pk_rmsg パラメータ以下のアドレスに置かれたメッセージが、cal_por の pk_rmsg パラメータ以下の領域コピーされる。メッセージの形式はランデブ成立時と同じく上記の形式である。結局、cal_por で指定した pk_rmsg 以下のメッセージ領域は、rpl_por 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。

maxrmsz により、ランデブ終了時にメッセージを受け取る領域（pk_rmsg 以下の空き領域）のサイズを指定する。ランデブ終了時にこれを越えるサイズのメッセージが送られてきた場合（cal_porのmaxrmsz＜rpl_porのrmsgsz）には、rpl_porのpk_rmsg で指定されたメッセージのうち、maxrmsz のサイズまでの分のみがコピーされ、それ以降はコピーされない。この場合、cal_por を発行したタスク（rpl_por を発行したタスクではない）には E_AROVR のエラーが返る。E_AROVR のエラーになった場合でも、ランデブの終了はエラーの無い場合と同じように行われる。また、cal_por で E_AROVR になった場合（cal_porのmaxrmsz＜rpl_porのrmsgsz）、pk_rmsg->rmsgsz に格納されるサイズは rpl_por で指定された rmsgsz（元のメッセージサイズ）であり、cal_por で指定された maxrmsz ではない。

tmoutにより待ち時間のタイムアウト指定を行うことができる。tmout としては、正の値のみ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、ランデブが成立しないまま tmout 時間が経過すると、タイムアウトエラー E_TMOUT としてエラーリターンする。

tmoutとして TMO_POL を指定した場合は、cal_por 実行時に既にランデブ成立条件が満たされていないと、E_TMOUT として即時にリターンする。tmout として TMO_POL を指定し、cal_por 実行時に既にランデブ成立条件が満たされていれば、cal_por を実行したタスクはランデブ受け付けタスクが rpl_por を実行するまでランデブ終了待ちの状態になる。この場合は、タイムアウト指定が無い時と同様に、ランデブ受け付けタスクの rpl_por の実行により cal_por は正常終了する。

tmout = TMO_FEVR により、タイムアウト指定が行われないことを示す。この場合はランデブ成立条件が満足されるまで永久に待つ。

いずれの場合も、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。

【エラーコード (ercd)】

E_OK	正常終了
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSID	予約ID番号（-4≦porid≦0）
E_PAR	一般的なパラメータエラー（rmsgsz＜4）
E_ILADR	不正アドレス（pk_rmsgが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
E_ILTIME	不正時間指定（tmout≦-2）
E_NOEXS	オブジェクトが存在していない（poridのポートが存在しない）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でporid＜(-4)）
E_CTX	コンテキストエラー（タスク独立部あるいはディスパッチ遅延中のタスクから発行）

E_DLT	待ちオブジェクトが削除された（ランデブ呼出待ち、あるいは終了待ちの間に対象ポートが削除）
E_AROVR	用意した領域のサイズが小さすぎる（rpl_porのpk_rmsg中のrmsgsz＞maxrmsz）
E_TMOUT	タイムアウト
E_RLWAI	待ち状態強制解除（待ちの間にter_tsk,rel_waiを受け付け）

ポートに対するランデブの受け付け
acp_por

acp_por: Accept Port for Rendezvous

【パラメータ】

porid	PortIdentifier	ポートID
acpptn	AcceptBitPattern	受付側選条件を表わすビットパターン
pk_rmsg	RendezvousMessagePacket	メッセージアドレス
maxrmsz	Max RendezvousMessageSize	メッセージ受信領域のサイズ
tmout	Timeout	タイムアウト指定

【リターンパラメータ】

ctskid	CallTaskIdentifier	呼出タスクID

【解説】

ポートに対するランデブ受け付けを行う。

具体的な動作は次のようになる。poridで指定したポートの呼び出し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、呼び出し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（成立待ち）状態からランデブ処理終了待ちの状態に変わる。acp_por の発行タスクは、実行を継続する。

porid で指定したポートの呼び出し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は、acp_por発行タスクがそのポートに対する受け付け待ち行列につながれる。porid で指定したポートに対して、受け付け待ちタスクが既に存在した場合でも、エラーとはならない。また、ITRON2では、一つのポートで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ため、poridで指定したポートで別のタスクがランデブ中の場合（前のランデブに関する rpl_por がまだ実行されてない場合）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 と呼び出し側タスクの 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。calptn, 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、呼び出し側と受け付け側で完全に対称である。

この機能により、選択受け付け（ADA の select）の処理が実現可能となる。具体的には、次のようにして実現される。

≪ADA でのプログラム例≫

```
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;
```

≪ITRONのランデブ機能による上記プログラムの実現方法≫

― entry_A, entry_B, entry_C がそれぞれ一つのポートになるのではなく、select 文全体が一つのポートに対応する。

― 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。

―上記の 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;
```

acp_por(acpptn :=ptn);

―もし、上記の文以外に、select 無しの単なる entry_A の accept があれば、
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、
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 call であれば
cal_por (calptn : = 2^0);
を実行し、entry_C の call であれば
cal_por (calptn : = 2^2);
を実行すれば良い。

ADA であれば、選択機能は受け付け側にしか用意されていないが、ITRONのランデブでは、calptn で複数のビットを指定することにより、呼び出し側に選択機能を持たせることも可能である。

acp_por のリターンパラメータのうち、ctskid は、ランデブの成立した相手タスク（呼び出し側タスク）のタスクID である。一つのタスクが複数のランデブを同時に行うことが可能であるため、rpl_por でランデブを終了する際には、ctskid の情報を指定する必要がある。ctskid は、ランデブID（ポートIDではなく、その上で成立した複数のランデブを区別するID）であると考えることができる。

acp_por の pk_rmsg 以下の領域には、cal_por の pk_rmsg 以下の領域で指定したメッセージがコピーされる。メッセージのサイズは、このメッセージの先頭ワードに置かれている。

maxrmsz により、ランデブ成立時にメッセージを受け取る領域（pk_rmsg 以下の空き領域）のサイズを指定する。ランデブ成立時にこれを越えるサイズのメッセージが送られてきた場合（acp_porのmaxrmsz＜cal_porのrmsgsz）には、cal_por の pk_rmsg で指定されたメッセージのうち、maxrmsz のサイズまでの分のみがコピーされ、それ以降はコピーされない。この場合、acp_por を発行したタスク（cal_por を発行したタスクではない）には E_AROVR のエラーが返る。E_AROVR のエラーになった場合でも、ランデブの終了はエラーの無い場合と同じように行われる。また、acp_por で

E_AROVR になった場合（acp_porのmaxrmsz＜cal_porのrmsgsz）、pk_rmsg->rmsgsz に格納されるサイズは cal_por で指定された rmsgsz（元のメッセージサイズ）であり、acp_por で指定された maxrmsz ではない。

tmoutにより待ち時間のタイムアウト指定を行うことができる。tmout としては、正の値のみ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、条件が満足されぬまま tmout 時間が経過すると、タイムアウトエラー E_TMOUT としてエラーリターンする。また、tmout として TMO_POL を指定した場合は、ランデブ成立条件が満たされていない場合でも即時にリターンする。この場合、ランデブが成立していれば正常終了に、ランデブ成立条件が満たされていなければタイムアウトエラー E_TMOUT になる。さらに、tmout = TMO_FEVR によりタイムアウト指定が行われないことを示す。この場合は条件が満足されるまで永久に待つ。

ランデブ受け付け側のタスクが、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、acp_por によりあるランデブを受け付けたタスクが、rpl_por を実行する前に、もう一度 acp_por を実行しても構わない。また、同時に成立している複数のランデブは、同じポートを対象としたものであっても、異なるポートを対象としたものであっても構わない。

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（rpl_por実行前）に異常終了したような場合は、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終了ハンドラで rpl_por または 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通知しておく必要がある。

【エラーコード (ercd)】

E_OK	正常終了
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSID	予約ID番号（-4≦porid≦0）
E_PAR	一般的なパラメータエラー（maxrmsz＜4,
E_ILADR	不正アドレス（pk_rmsgが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）

E_ILTIME	不正時間指定（tmout≦-2）
E_NOEXS	オブジェクトが存在していない（poridのポートが存在しない）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でporid＜(-4)）
E_CTX	コンテキストエラー（タスク独立部あるいはディスパッチ遅延中のタスクから発行）
E_DLT	待ちオブジェクトが削除された（待ちの間に対象ポートが削除）
E_AROVR	用意した領域のサイズが小さすぎる（cal_porあるいはfwd_porのpk_rmsg中の rmsgsz＞maxrmsz）
E_TMOUT	タイムアウト
E_RLWAI	待ち状態強制解除（待ちの間にter_tsk,rel_waiを受け付け）

ポートに対するランデブの回送 fwd_por

fwd_por: Forward Port for Rendezvous

【パラメータ】

porid	PortIdentifier	ポートID
ctskid	CallTaskIdentifier	呼出タスクID
calptn	CallBitPattern	呼出側選択条件を表わすビットパターン
pk_rmsg	RendezvousMessagePacket	メッセージアドレス

【リターンパラメータ】

なし

【解説】

一旦受け付けたランデブを別のポートに回送する。

このシステムコールを発行したタスク（タスクX）は、あるポート（ポートA）において ctskidで示されるタスク（タスクY）のランデブ呼び出しを受け付け、現在ランデブ中の状態でなければならない。その状態でこのシステムコールを実行すると、ポートAにおけるタスクXとタスクYとのランデブ状態が解除され、porid で示される別のポート（ポートB）に対してタスクYがランデブ呼び出しを行ったのと同じ状況になる。

fwd_por の具体的な動作は次のようになる。

1. ctskid で示されるタスクとのランデブ状態を解除する。
2. ctskid で示されるタスクを、porid のポートに対してランデブ呼びだし待ちの状態にする。この時、ランデブ成立条件の calptn は、cal_por で指定したものではなく、fwd_por で指定したものが使用される。ctskid のタスクから見ると、ランデブ終了待ちの状態からランデブ受付待ちの状態に戻ることになる。
3. その後、porid の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 ctskid のタスクとの間でランデブ成立となる。もちろん、ランデブ成立条件が合っていれば、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、cal_por で指定したものではなく、fwd_por で指定したものになる。fwd_por を実行したタスクは、この新しいランデブとは全く関係しない。

4. 新しいランデブが終了した際に rpl_por で返されるメッセージは、fwd_por で指定した pk_rmsg 以下の領域ではなく、cal_por で指定した pk_rmsg 以下の領域にコピーされる。これは、ランデブを呼び出した側に対して、ランデブが回送されたことを見せないようにするために必要な仕様である。

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。

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。

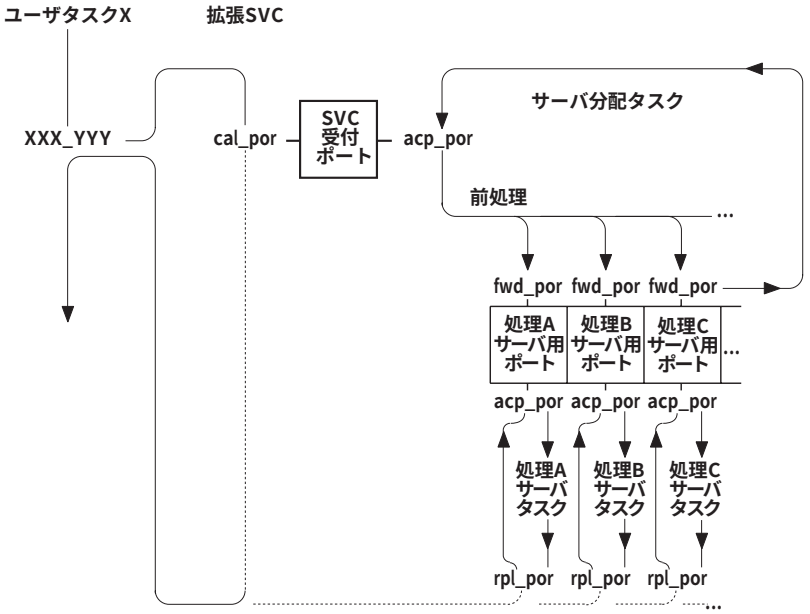
fwd_por を使ったサーバタスクの動作イメージを以下の図に示す。

fwd_por を実行する際、fwd_por で送信されるメッセージは cal_por の pk_rmsg で指定されたメッセージ領域にコピーされる。したがって、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、cal_por の pk_rmsg に置かれたメッセージが受付側に渡される。すなわち、ランデブが回送された場合でも、ランデブが成立するまで、cal_por で指定したメッセージ領域をメッセージバッファとして使うことになる。このため、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、fwd_por の pk_rmsg 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になる。

fwd_por で指定したメッセージのサイズ（fwd_porのpk_rmsg->rmsgsz）が、cal_por を実行した側で確保しているメッセージ領域の大きさ（cal_porのmaxrmsz）よりも大きい場合は、cal_por の maxrmsz の大きさまでのメッセージがコピーされ、ランデブの回送は行われるが、fwd_por に対しては E_AROVR のエラーが返るものとする。

タスク独立部から fwd_por, rpl_por を発行した場合には E_CTX のエラーになるが、ディスパッチ遅延中（割込み禁止中）のタスクから fwd_por, rpl_por を発行することは可能である。この機能は、fwd_por や rpl_por と不可分に何らかの処理を行う場合に利用できる。

fwd_por により、ランデブ終了待ち状態であったタスクAがランデブ成立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) として扱われるものとする。これは、タスクAが cal_por でタイムアウト指定を行っていた場合でも同様である。



＊太枠内はポート（ランデブエントリ）を表わす。
＊ ┆ は待ちを表わす。
＊fwd_por の代りに cal_por を使うことも可能であり、その場合は
ランデブがネストすることになる。しかし、処理A～C のサーバタ
スクの処理終了後にそのままユーザタスクX の実行を再開して良け
＝桝ﾓ枠内はポート（ランデブエントリ）を表わす。
＊ は待ちを表わす。
＊ の代り

[図3.36] fwd_porを使ったサーバタスクの動作イメージ

【補足事項】

fwd_por で送信するメッセージを cal_por で指定したメッセージ領域にコピーするのは、次のような理由による。

一般に、fwd_por を実行するのはサーバ分配タスクなどであるため、メッセージ領域を再利用して別の回送などを行いたいことが多い。ところが、fwd_por を実行した側では、回送したランデブがいつ成立するか分からない。そのため、fwd_por で指定したメッセージを別の場所にコピーせず、ポインタだけを保持する仕様にしておくと、fwd_por を実行した側では、pk_rmsg

で指定したメッセージ領域がいつ再利用（変更）可能になるかということが分からない。したがって、fwd_por を実行したサーバ分配タスクは、別の要求の回送を行うことができない。

一方、cal_por を実行したタスクはランデブ終了まで待ち状態なので、cal_por のメッセージ領域をランデブ成立までのバッファとして使っても、特に問題はない。

メッセージのサイズが大きすぎる場合、cal_por → acp_por, rpl_por → cal_por では、いずれもメッセージを受け取る側に E_AROVR のエラーを返している。しかし、fwd_por では、メッセージを送る側（fwd_porを実行した側）が E_AROVR のエラーとなることもある。このような仕様になっているのは、fwd_por で送られるメッセージが、一旦他の領域にコピーされるためである。

fwd_por により回送されたランデブが成立するまでのタイムアウトを常に永久待ちとして扱うのは、次のような理由による。cal_por で指定するタイムアウトは、最初にランデブが成立するまでのタイムアウトであり、ランデブ終了までのタイムアウトを意味しているわけではない。ランデブ終了までのタイムアウトは、常に永久待ちとして扱われている。fwd_por 実行後のランデブ成立待ちの時間は、cal_por を実行したタスクから見た場合、ランデブが成立するまでの時間ではなく、ランデブが終了するまでの時間に含めて考えるのが自然である。したがって、fwd_por 実行後のランデブ成立待ちのタイムアウトも、永久待ちとして扱う。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦porid≦0、-4≦ctskid≦0）
E_PAR	一般的なパラメータエラー（rmsgsz＜4,
E_ILADR	不正アドレス（pk_rmsgが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（poridのポートが存在しない、ctskidのタスクが存在しない）
E_DMT	タスクがDORMANTである（ctskidのタスクがDORMANT）
E_NORDV	現在ランデブ中のタスクではない（ctskidで現在ランデブ中ではないタスクを指定）

E_CTX	コンテキストエラー（タスク独立部から発行）
E_AROVR	用意した領域のサイズが小さすぎる （fwd_porのpk_rmsg中のrmsgsz＞cal_porのmaxrmsz）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でporid＜(-4)） ※ ctskid に関しては E_OACV をチェックしない

ポートに対するランデブの返答

rpl_por

rpl_por: Reply Port for Rendesvous

【パラメータ】

ctskid	CallTaskIdentifier	呼び出しタスクID
pk_rmsg	RendezvousMessagePacket	メッセージアドレス

【リターンパラメータ】

なし

【解説】

ランデブ中のポート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）は、あるポート（ポートA）において ctskid で示されるタスク（タスクY）のランデブ呼び出しを受け付け、現在ランデブ中の状態でなければならない。その状態でこのシステムコールを実行すると、ポートAにおけるタスクXとタスクYと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を実行可能状態に移す。

このシステムコールを使ってランデブを終了する時、受付側タスクXから呼出側タスクYに対してメッセージを送ることができる。rpl_por の pk_rmsg のパラメータでメッセージの先頭アドレスを指定することにより、そのメッセージが cal_por の pk_rmsg のパラメータで指定した領域にコピーされる。メッセージの形式はcal_por で指定したものと同じである。

タスク独立部から fwd_por, rpl_por を発行した場合には E_CTX のエラーになるが、ディスパッチ遅延中（割込み禁止中）のタスクから fwd_por, rpl_por を発行することは可能である。この機能は、fwd_por や rpl_por と不可分に何らかの処理を行う場合に利用できる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦ctskid≦0）
E_PAR	一般的なパラメータエラー（rmsgsz＜4）

E_ILADR	不正アドレス（pk_rmsgが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（ctskidのタスクが存在しない）
E_DMT	タスクがDORMANTである（ctskidのタスクがDORMANT）
E_NORDV	現在ランデブ中のタスクではない（ctskidで現在ランデブ中ではないタスクを指定）
E_CTX	コンテキストエラー（タスク独立部から発行） ※ ctskid に関しては E_OACV をチェックしない

ポート状態を参照する

por_sts

por_sts: Get Port Status

【パラメータ】

porid	PortIdentifier	ポートID
pk_pors	Packet of PortStatus	ポート状態を返すパケットの先頭アドレス

【リターンパラメータ】

なし

【解説】

porid で示されたポートの状態を読み出し、その結果を pk_pors 以下の領域に返す。

pk_pors に返される情報としては、次のようなものがある。

```
poratr      /* ポート属性 */
wtskid      /* 呼出側待ち行列先頭のタスクID */
atskid      /* 受付側待ち行列先頭のタスクID */
```

wtskid には、呼出側待ち行列の先頭のタスクのIDが返る。また、atskid には、受付側待ち行列の先頭のタスクのIDが返る。待ち行列にタスクが無い時には、それぞれ FALSE = 0 が返る。

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。wtskid, atskid で示されるタスクは、このポートに関してまだランデブが成立していないタスクに限られる。現在ランデブ中のタスクに関する情報を知りたい場合には、rdv_sts を利用する。

対象となるポートは、既に生成されたものでなければならない。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号 (-4≦porid≦0)
E_ILADR	不正アドレス (pk_porsが使用できない値)

E_IDOVR	ID範囲外（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（poridのポートが存在しない）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でporid＜(-4)）

ランデブ状態を参照する

rdv_sts

rdv_sts: Get Rendesvous Status

【パラメータ】

porid	PortIdentifier	ポートID
ar_rdvs	Array of Rendesvous	ランデブ情報を返す配列へのポインタ
rvstart	StartEntry of Rendesvous	状態参照を開始するランデブの順番
rvcnt	Count of Rendesvous	状態参照を行うランデブの個数

【リターンパラメータ】

なし

【解説】

porid で示されたポートでランデブ中のタスク（複数）に関する情報を読み出し、その結果を ar_rdvs 以下の領域に返す。

porid で起こっている個々のランデブに対しては、ランデブ成立順に順番が付いている。この順番は、このシステムコールでのみ意味を持つものである。rdv_sts では、このうちの (rvstart+1) 番目から rvcnt 個のランデブに関する情報を配列にして、ar_rdvs 以下の領域に返す。

個々のランデブに対して、このシステムコールにより得られる情報としては、次のようなものがある。

```
ctskid      /* ランデブ中の呼出側タスクのID */
atskid      /* ランデブ中の受付側タスクのID */
pk_rmsg     /* 返答時に渡されるメッセージを受取るアドレス */
maxrmsz     /* 受け取ることのできるメッセージの最大長 */
```

※ pk_rmsg は、rpl_por で送られるメッセージを入れるアドレスである。したがって、一旦回送されたランデブの場合でも、pk_rmsg は fwd_por で指定した値ではなく、cal_por で指定した値になる。

一つのランデブに対して、これらの情報を含んだ一つの構造体が対応する。

その構造体を配列にしたもの（要素数はrvcnt個）が ar_rdvs に返される。

成立中のランデブの個数が (rvstart+rvcnt) よりも少なかった場合には、ar_rdvs の配列のその次の要素の ctskid が FALSE = 0 となる。すなわち、ランデブ成立の個数を N とすると、ar_rdvs[N-rvstart].ctskid が FALSE = 0 となる。この場合、ar_rdvs の配列のそれ以下の要素に対する ctskid、すなわち ar_rdvs[N-rvstart+1].ctskid, ar_rdvs[N-rvstart+2].ctskid, ... ar_rdvs[rvcnt-1].ctskid の値は不定である。

一方、(rvstart+rvcnt) で指定した数よりも成立中のランデブの個数が多い場合には、用意した配列の長さが短すぎるということであるから、E_AROVR のエラーとする。なお、エラー発生の場合にはできるだけ元の状態に戻すという一般原則とは異なり、このエラーの場合は、rvcnt 個までのランデブの情報が ar_rdvs に返される。また、rvcnt が 0 の場合にも同様の動作をするが、この場合は、エラーコードが E_AROVR かどうかによって、ランデブの個数が rvstart より多いかどうかをチェックすることになる。rvcnt が 0 であれば、ar_rdvsの指す領域の内容は変化しない。

rdv_sts で得られる ctskid, atskid, pk_rmsg の情報は、既にランデブ中のタスクに関する情報であり、ランデブ呼出待ちやランデブ受付待ち状態のタスクに関する情報ではない。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦porid≦0）
E_PAR	一般的なパラメータエラー（rvstart＜0、rvcnt＜0）
E_ILADR	不正アドレス（ar_rdvsが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（poridのポートが存在しない）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でporid＜(-4)）
E_AROVR	用意した領域のサイズが小さすぎる

強制例外機能

タスク用強制例外ハンドラの定義 def_fex

def_fex: Define ForcedExceptionHandler

【パラメータ】

tskid	TaskIdentifier	タスクID
exhatr	ExceptionHandlerAttribute	例外ハンドラ属性
fexhdr	ForcedExceptionHandlerAddress	強制例外ハンドラアドレス

【リターンパラメータ】

なし

【解説】

タスクの強制例外ハンドラを定義する。

強制例外ハンドラでは、例外の原因となった情報を、スタックを通じて例外コード exccd として受け取ることができる。強制例外ハンドラの場合の exccd の内容は、ras_fex で指定した exccd である。

exhatrのサイズは標準で4バイトである。exhatrのフォーマットを［図3.37］に示す。このうち、最上位バイト（bit0〜bit7 あるいは 2^31〜2^24 のビット）は未使用であり、第二上位バイト（bit8〜bit15 あるいは 2^23〜2^16 のビット）はインプリメント依存の属性を表わし、下位ハーフワード（bit16〜bit31 あるいは 2^15〜2^0 のビット）はシステム属性を表わす。

［図3.37］exhatrのフォーマット

exhatr のシステム属性の部分では、次のような指定を行うことができる。

exhatr : = (TA_ASM ‖ TA_HLNG)

TA_ASM	ハンドラがアセンブラで書かれている
TA_HLNG	ハンドラが高級言語で書かれている

TA_HLNG の指定を行った場合には、ハンドラ起動時に直接 fexhdr 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 fexhdr のアドレスにジャンプする。

fexhdr = NADR (-1) の指定により、前に定義されていた強制例外ハンドラの定義が解除され、未定義の状態になる。ただし、前に定義されていた強制例外ハンドラの定義解除を行わずに、直接新しいハンドラを定義しても、エラーにはならない。また、もともとハンドラが定義されていない状態（定義が解除された状態）で、fexhdr=NADR の指定による定義解除の操作を行っても、エラーにはならない。後者の場合、このシステムコールは正常終了するが、強制例外ハンドラは未定義のままである。

tskid = TSK_SELF により、自タスクのハンドラを指定する。また、tskid = TSK_CMN によりタスク間で共通の強制例外ハンドラを指定する。タスク実行中に強制例外が発生した場合には、OSはタスク固有の例外ハンドラが定義してあるかどうかを調べ、定義してあればそれを実行する。定義してなければ、次にタスク間共通の強制例外ハンドラが定義してあるかどうかを調べ、定義してあればそれを実行する。

強制例外ハンドラは、強制例外を受けたタスク（ras_fexの対象タスク）の一部として、強制例外を受けたタスクの実行モード（リング、レベル）と同じ実行モード（リング、レベル）で実行される。強制例外ハンドラの実行モード（リング、レベル）は、強制例外を起こしたタスク（ras_fexを実行したタスク）の実行モード（リング、レベル）には関係しない。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足 ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSID	予約ID番号（-4≦tskid≦-2、タスク独立部の発行でtskid=0）

E_IDOVR	ID範囲外（インプリメント依存）
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_RSATR	予約属性（exhatrが不正）
E_ILADR	不正アドレス（fexhdrが奇数、あるいは使用できない値）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でtskid＜(-4)）

他タスクに対する強制例外の発生 ras_fex

ras_fex: Raise ForcedException

【パラメータ】

tskid	TaskIdentifier	強制例外発生タスクID
exccd	ExceptionCode	例外コード

【リターンパラメータ】

なし

【解説】

他タスクに対して強制例外を発生する。

ras_fex では、tskid として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_SELF のエラーとなる。

強制例外では、待ち状態や強制待ち状態（SUSPEND状態）の解除は行わない。ras_fex の対象タスクが待ち状態であった場合、強制例外ハンドラが起動されるのは、待ち状態が解除された後になる。待ち状態を強制的に解除して強制例外ハンドラを起動する必要がある場合には、ras_fex と rel_wai や frsm_tsk を併用する必要がある。

ras_fex による複数の強制例外起動要求のキューイングは行わない。すなわち、強制例外ハンドラの起動前に、同じタスクを対象とした ras_fex が複数回発行されても、強制例外ハンドラは1回しか起動されない。ただし、この場合は、各 ras_fex で指定された exccd の論理和がとられていき、pk_exc の exccd ではこの論理和の値が返る。したがって、exccd をビット対応で使うことにより、疑似的に複数の要求を区別することができる。たとえば、例外コードとしての情報が1ビットで良ければ、32種類の独立した要求を区別することができる。起動された強制例外ハンドラでは、exccd を見て複数の起動要求があるかどうかを判断し、内部の処理を振り分ければ良い。

強制例外ハンドラの起動時には、次の強制例外ハンドラ起動に備えて強制例外要求 epndptn がクリアされるが、同様に、それまで論理和をとっていた例外コード exccd も0 にクリアされる。（強制例外ハンドラ中でpk_exc の中の exccd を参照した場合には、0 ではなく、クリア直前の値が返る。）

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦tskid≦-1、タスク独立部の発行でtskid=0）
E_IDOVR	ID範囲外（インプリメント依存）
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_DMT	タスクがDORMANTである（tskidのタスクがDORMANT）
E_SELF	自タスク、自プロセスの指定（tskidが自タスク、タスク部や準タスク部の発行で tskid = 0）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でtskid＜(-4)）

ローカルメモリプール管理機能

ここでは、タスクがローカルに使用するローカルメモリブロックのためのローカルメモリプールについて説明を行う。ローカルメモリプール管理のシステムコールは、基本機能のメモリプール管理（共有メモリプール管理）のシステムコールとほぼ同じになっている。ただし、ローカルメモリプールの場合は、MMUのページング機能を生かすため、ブロックサイズの値に制限を設けることがある。

ローカルメモリプールのオブジェクト名称は lmp、またローカルメモリブロックの名称は lbl となる。いずれも、共有メモリプールや共有メモリブロックの名称の最初に 'l' を付けたものになっている。

1タスクのみ、あるいは1つの論理空間のみが使用するローカルメモリプールというのはあまり意味がないので、ローカルメモリプールであっても、ローカルメモリプール自体は複数のタスクや論理空間から共用されるものと考える。すなわち、一つのローカルメモリプールから得られたいくつかのローカルメモリブロックが、複数の論理空間に分かれて使用される可能性がある。ローカルメモリプールであっても、それがどの論理空間に属するものかをあらかじめ指定しておく必要はない。インプリメント上は、ローカルメモリプール生成時に物理メモリの確保のみを行い、メモリブロック獲得時に、要求タスクの論理空間のページテーブルを操作して、獲得したメモリブロックを要求タスクの論理空間に割り付けるということになる。

メモリプールからメモリを確保する際には、タスクがローカルで使用するローカルメモリか、タスク間で共有する共有メモリかによって、システムコールが分かれることになる。この区別は、non-MMU 版ではインプリメント上の意味を持たず、プログラミング作法としての意味だけになるが、MMU 版への移行をスムーズにするために役立つ。

ローカルメモリプールには、共有メモリプールと同様に、固定長メモリプールと可変長メモリプールがある。両者はローカルメモリプールの属性として区別される。

ローカルメモリプールの生成 cre_lmp

cre_lmp: Create LocalMemoryPool

【パラメータ】

lmpid	LocalMemoryPoolIdentifier	ローカルメモリプールID
lmpatr	LocalMemoryPoolAttribute	ローカルメモリプール属性
lmpsz	LocalMemoryPoolSize	ローカルメモリプール全体のサイズ
blksz	MemoryBlockSize	メモリブロック基本サイズ

【リターンパラメータ】

なし

【解説】

cre_lmp では、lmpid で指定された ID 番号を持つローカルメモリプールを生成する。次に、生成されたローカルメモリプールに対して管理ブロックを割り付ける。

ローカルメモリプールの生成時には、lmpatr によって、固定長ローカルメモリプール / 可変長ローカルメモリプールの区別を指定する。

ローカルメモリプール全体の大きさは、lmpsz によって指定する。ローカルメモリプール生成に必要なメモリ領域の確保は、システム生成時などに行われる。

可変長のローカルメモリプールの場合、blksz により、メモリ獲得、解放の最小単位が決められる。固定長のローカルメモリプールであれば、blksz の大きさでのメモリ獲得、解放しかできない。

ID 番号が (-4)〜0 のローカルメモリプールは生成できない。また、負の ID 番号のローカルメモリプールは、システム用のものである。

lmpatrのサイズは標準で4バイトである。lmpatrのフォーマットを［図3.38］に示す。このうち、最上位バイト（bit0〜bit7 あるいは 2^31〜2^24 のビット）はユーザ属性を表わし、第二上位バイト（bit8〜bit15 あるいは 2^23〜2^16 のビット）はインプリメント依存の属性を表わし、下位ハーフワード（bit16〜bit31 あるいは 2^15〜2^0 のビット）はシステム属性を表わす。ユー

［図3.38］lmpatrのフォーマット

ザは、ローカルメモリプールに関する情報を入れておくために、lmpatr の最上位バイトを自由に使用することができる。

cre_lmp で指定した lmpatr は、lmp_sts により読み出すことができる。

lmpatr のシステム属性の部分では、次のような指定を行うことができる。

lmpatr : = (TA_TFIFO ‖ TA_TPRI) ｜ (TA_FIRST ‖ TA_CNT)
　　｜ (TA_VAR ‖ TA_FIX)

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	行列先頭のタスクを優先扱い
TA_CNT	要求数の少ないタスクを優先扱い
TA_VAR	可変長メモリブロック用のローカルメモリプール
TA_FIX	固定長メモリブロック用のローカルメモリプール

TA_TPRI, TA_TFIFO の属性により、タスクがメモリ獲得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。

さらに、TA_CNT と TA_FIRST の属性指定により、要求メモリ数の少ないタスクを優先扱いにするか、メモリ待ち行列先頭のタスクを優先扱いにするかが指定できる。TA_CNT の属性を指定した場合には、要求メモリ数によって、行列途中のタスクから先に待ち解除になる場合がある。例えば、ある可変長ローカルメモリプールに対して要求ブロック数=5 のタスク A と要求ブロック数=1 のタスクBがこの順で待っており、rel_lbl により空きブロックの数が1になった場合、行列先頭にあるタスクA は要求ブロック数が多いので、行列の後にあるタスクB の方が先にメモリを獲得する。

一方、TA_FIRST の属性を指定した場合には、要求ブロック数にかかわらず、必ず行列先頭のタスクからメモリが割り当てられる。これは、要求ブロック数よりも優先度を重視した考え方である。上の例では、rel_lbl 実行後もタスクB は待ち解除にならず、さらに rel_lbl が実行されて連続した空きブロックの数が 5 以上になった時にvはじめてタスクA にメモリが割り当てられる。タスクB には、その後の rel_lbl によってメモリが割り当てられる。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足（管理ブロック用の領域が確保できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_NOMEM	メモリ不足（ローカルメモリプール用の領域が確保できない） ※ E_NOSMEMとE_NOMEMとの厳密な区別はインプリメント依存である。
E_RSID	予約ID番号（-4≦lmpid≦0）
E_RSATR	予約属性（lmpatrの下位3バイトが不正）
E_PAR	一般的なパラメータエラー（lmpsz,blkszが不正）
E_IDOVR	ID範囲外（lmpidがシステムで利用できる範囲を越えた）
E_EXS	オブジェクトが既に存在している（同一ID番号のローカルメモリプールが存在）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でlmpid＜(-4)）

ローカルメモリプールを削除する del_lmp

del_lmp: Delete LocalMemoryPool

【パラメータ】

lmpid	LocalMemoryPoolIdentifier	ローカルメモリプールID

【リターンパラメータ】

なし

【解説】

lmpid で示されるローカルメモリプールを削除する。

このローカルメモリプールからメモリを獲得しているタスクに対しては、何の保証もしない。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。

このローカル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る場合にも、本システムコールは正常終了するが、メモリ獲得を待っていたタスクにはエラー E_DLT が返される。

本システムコール発行後は、そのID番号のローカルメモリプールを再び新しく生成することができる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦lmpid≦0）
E_IDOVR	ID範囲外（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（lmpidのローカルメモリプールが存在しない）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でlmpid＜(-4)）

ローカルメモリブロックを獲得する get_lbl

get_lbl: Get Local Memory Block

【パラメータ】

lmpid	LocalMemoryPoolIdentifier	ローカルメモリプールID
bcnt	ContinuousBlockCount	連続領域のブロック数
tmout	Timeout	タイムアウト指定

【リターンパラメータ】

blk	BlockStartAddress	メモリブロックの先頭アドレス

【解説】

lmpid で示されるローカルメモリプールから、ローカルメモリブロック（固定長 / 可変長）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。

get_lbl で獲得したメモリのゼロクリアは特に行われない。獲得されたメモリブロックの内容は不定である。

可変長ローカルメモリプールを対象とした場合は、このシステムコールにより、（cre_lmpで指定したblksz）×bcntの大きさのローカルメモリブロックが確保される。また、固定長ローカルメモリプールを対象とした場合は、bcnt の指定は無視され、常に（cre_lmp で指定した blksz）の大きさのローカルメモリブロックが確保される。

指定したローカルメモリプールから、指定した大きさのメモリブロックがすぐに確保できない場合には、タスクはそのローカルメモリプールのメモリ獲得待ち行列につながれ、確保できるようになるまで待つ。

tmout により待ち時間のタイムアウト指定を行うことができる。tmout としては、正の値のみ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、メモリブロックを獲得できないまま tmout 時間が経過すると、タイムアウトエラー E_TMOUT としてエラーリターンする。また、tmout として TMO_POL を指定した場合は、メモリブロックが獲得できない場合でも即時にリターンする。この場合、メモリブロックが獲得できれば正常終了

になり、メモリブロックが獲得できなければタイムアウトエラーE_TMOUT になる。さらに、tmout=TMO_FEVR によりタイムアウト指定が行われないことを示す。この場合は条件が満足されるまで永久に待つ。

【エラーコード (ercd)】

E_OK	正常終了
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSID	予約ID番号（-4≦lmpid≦0）
E_PAR	一般的なパラメータエラー（bcnt≦0）
E_IDOVR	ID範囲外（インプリメント依存）
E_ILTIME	不正時間指定（tmout≦-2）
E_NOEXS	オブジェクトが存在していない（lmpidのローカルメモリプールが存在しない）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でlmpid＜(-4)）
E_CTX	コンテキストエラー（タスク独立部あるいはディスパッチ遅延中のタスクから発行）
E_DLT	待ちオブジェクトが削除された（待ちの間に対象ローカルメモリプールが削除）
E_TMOUT	タイムアウト
E_RLWAI	待ち状態強制解除（待ちの間にter_tsk,rel_waiを受け付け）

ローカルメモリブロックを返却する rel_lbl

rel_lbl: Release Local Memory Block

【パラメータ】

lmpid	LocalMemoryPoolIdentifier	ローカルメモリプールID
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】

なし

【解説】

blk で示されるメモリブロックを、lmpid で示されるローカルメモリプールへ返却する。

メモリブロックの返却を行うローカルメモリプールは、メモリブロックの獲得を行ったローカルメモリプールと同じものでなければならない。メモリブロックを異なるローカルメモリプールへ返却していることが検出された場合には、E_ILBLK のエラーとなる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦lmpid≦0）
E_IDOVR	ID範囲外（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（lmpidのローカルメモリプールが存在しない）
E_ILBLK	不正メモリブロックの返却、操作
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でlmpid＜(-4)）

ローカルメモリプールの状態を参照する　lmp_sts

lmp_sts:　Get LocalMemoryPool Status

【パラメータ】

lmpid	LocalMemoryPoolIdentifier	ローカルメモリプールID
pk_lmps	Packet of LocalMemoryPoolStatus	ローカルメモリプール状態を返すパケットの先頭アドレス

【リターンパラメータ】

なし

【解説】

lmpid で示されたローカルメモリプールの状態を読み出し、その結果を pk_lmps 以下の領域に返す。

pk_lmps に返される情報としては、次のようなものがある。

```
lmpatr      /* ローカルメモリプール属性 */
wtskid      /* 待ち行列先頭のタスクID */
frbcnt      /* 空き領域全体のブロック数 */
maxbcnt     /* 最大の連続空き領域のブロック数 */
lmpsz       /* ローカルメモリプール全体のサイズ */
blksz       /* メモリブロックのサイズ */
fragcnt     /* 空きブロックのフラグメント数 */
```

wtskid には、待ち行列の先頭のタスクのIDが返る。すなわち、属性が TA_TFIFO であれば、現在このローカルメモリプールを待っているタスクのうち、最も早く待ち状態になったタスクのIDが返る。また、属性が TA_TPRI であれば、現在このローカルメモリプールを待っているタスクのうち、最も優先度の高いタスクのIDが返る。

frbcnt はメモリの空き領域の合計ブロック数であり、maxbcnt は空き領域のうちで最大の連続領域のブロック数である。固定長ローカルメモリプールの場合は、maxbcnt は意味を持たず、常に1が返される。また、lmpsz,

blksz には、cre_lmp で指定した値がそのまま返される。

fragcnt は、メモリ空きブロックのフラグメント数（空きブロックがいくつの連続領域に分かれているか）を示す情報である。この情報は、対象のローカルメモリプールからどの程度の大きさの連続ブロックが取れるかということを知るための目安になる。固定長ローカルメモリプールに対するfragcnt の値としては、frbcnt と同じ値がセットされる。

対象となるローカルメモリプールは、既に生成されたものでなければならない。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦lmpid≦0）
E_ILADR	不正アドレス（pk_lmpsが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（lmpidのローカルメモリプールが存在しない）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でlmpid＜(-4)）

ローカルメモリブロックの状態参照 lbl_sts

lbl_sts: Get Local Memory Block Status

【パラメータ】

blk	BlockStartAddress	ローカルメモリブロックの先頭アドレス

【リターンパラメータ】

lmpid	LocalMemoryPoolIdentifier	ローカルメモリプールID
bcnt	ContinuousBlockCount	連続領域のブロック数

【解説】

blk で示されるローカルメモリブロックの状態を参照する。具体的には、そのローカルメモリブロックの属するローカルメモリプールID lmpid と、メモリブロックの大きさ（連続ブロック数）bcnt を返す。

固定長メモリプールから得られたメモリブロックに対して lbl_sts が実行された場合には、bcnt として1が返る。

【エラーコード (ercd)】

E_OK	正常終了
E_ILBLK	不正メモリブロックの返却、操作

資源管理サポート機能

「資源管理サポート機能」では、資源操作と不可分にメモリの更新を行う。ここで言うシステムコールの不可分性には、「A. 他のタスクに対する不可分性」と「B. 割込みハンドラに対する不可分性」とが考えられ、B.は A. よりも強い意味を持つ。A. は、資源管理サポート機能として当然必要な機能である。

ところが、B. を完全に保証しようとするとどうしても割込み禁止時間が長くなってしまうし、割込みハンドラから発行できるシステムコール自体もインプリメント依存となっている。また、資源管理サポート機能は、主に終了ハンドラとの不可分性を保つために導入された機能であり、割込みハンドラとの不可分性について強い要求があったわけではない。

そこで、「B. 割込みハンドラに対する不可分性」については、インプリメント依存の扱いとする。割込みハンドラからのシステムコール発行を許すオブジェクトであっても、割込みハンドラに対する uXXX_YYY の不可分性は保証されない場合がある。もちろん、どのような場合に uXXX_YYY の不可分性が確保できないかということについては、インプリメント毎に明らかにして頂く必要がある。

セマフォに対する信号操作（メモリ更新有） usig_sem

usig_sem: Signal Semaphore with Memory Update

【パラメータ】

semid	SemaphoreIdentifier	セマフォID
rcnt	ResourceCount	増カウント値
p_rcnt	Pointer to ResourceCount	不可分のカウント値操作を行うメモリアドレス

【リターンパラメータ】

なし

【解説】

sig_sem の処理を行い、それと不可分に、アドレス p_rcnt で指定したメモリの内容（標準4バイト）からrcnt で指定したカウント数を減らす。

この機能は、ユーザレベルで資源割り当て状況を管理したい場合に使用する。usig_sem の前後で割込み処理に入ったり、タスクが強制終了させられたりした場合でも、p_rcnt の内容を見ることによって、usig_sem の処理が終わったかどうかを知ることができる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦semid≦0）
E_PAR	一般的なパラメータエラー（rcnt≦0）
E_ILADR	不正アドレス（p_rcntが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
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_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でsemid＜(-4)）
E_QOVR	キューイングのオーバーフロー(semcntのオーバーフロー)

セマフォに対する待ち操作（メモリ更新有） uwai_sem

uwai_sem: Wait on Semaphore with Memory Update

【パラメータ】

semid	SemaphoreIdentifier	セマフォID
rcnt	ResourceCount	減カウント値
tmout	Timeout	タイムアウト指定
p_rcnt	Pointer to ResourceCount	不可分のカウント値操作を行うメモリアドレス

【リターンパラメータ】

なし

【解説】

wai_sem の処理を行い、それと不可分に、アドレス p_rcnt で指定したメモリの内容（標準4バイト）に rcntで指定したカウント数を加える。資源獲得待ち状態の間は、p_rcnt の内容は変化しない。

この機能は、ユーザレベルで資源割り当て状況を管理したい場合に使用する。uwai_sem の前後で割込み処理に入ったり、タスクが強制終了させられたりした場合でも、p_rcnt の内容を見ることによって、uwai_sem の資源獲得処理が終わったかどうかを知ることができる。

【エラーコード (ercd)】

E_OK	正常終了
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSID	予約ID番号（-4≦semid≦0）
E_PAR	一般的なパラメータエラー（rcnt≦0）
E_ILADR	不正アドレス（p_rcntが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
E_ILTIME	不正時間指定（tmout≦-2）

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_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でsemid＜(-4)）
E_CTX	コンテキストエラー（タスク独立部あるいはディスパッチ遅延中のタスクから発行）
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_TMOUT	タイムアウト
E_RLWAI	待ち状態強制解除（待ちの間にter_tsk,rel_waiを受け付け）

メイルボックスへ送信する（メモリ更新有） usnd_msg

usnd_msg: Send Message to Mailbox with Memory Update

【パラメータ】

mbxid	MailboxIdentifier	メイルボックスID
ppk_msg	Pointer to MessagePacket	送信メッセージの先頭アドレスへのポインタ

【リターンパラメータ】

なし

【解説】

アドレス ppk_msg の指す領域の内容をメッセージアドレスとして、snd_msg の処理を行い、それと不可分に、アドレス ppk_msg の指す領域の内容を NADR (-1) とする。

この機能は、ユーザレベルで資源割り当て状況を管理したい場合に利用するものであり、メモリプールから獲得したメモリブロックをメッセージとして使用する場合に効果がある。usnd_msg の前後で割込み処理に入ったり、タスクが強制終了させられたりした場合でも、ppk_msg の内容を見ることによって、usnd_msg の処理が終わったかどうかを知ることができる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦mbxid≦0）
E_ILADR	不正アドレス（ppk_msg、あるいはppk_msgの内容が使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
E_ILMSG	不正メッセージ形式（msgtype≠0）
E_NOEXS	オブジェクトが存在していない（mbxidのメイルボックスが存在しない）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でmbxid＜(-4)）

メイルボックスから受信する（メモリ更新有） urcv_msg

urcv_msg: Receive Message from Mailbox with Memory Update

【パラメータ】

mbxid	MailboxIdentifier	メイルボックスID
tmout	Timeout	タイムアウト指定
ppk_msg	Pointer to MessagePacket	受信メッセージの先頭アドレスへのポインタ

【リターンパラメータ】

なし

【解説】

rcv_msg の処理を行い、それと不可分に、受信したメッセージの先頭アドレスを ppk_msg の指す領域に設定する。メッセージ待ち状態の間は、ppk_msg の内容は変化しない。

この機能は、ユーザレベルで資源割り当て状況を管理したい場合に利用するものであり、メモリプールから獲得したメモリブロックをメッセージとして使用する場合に効果がある。urcv_msg の前後で割込み処理に入ったり、タスクが強制終了させられたりした場合でも、ppk_msg の内容を見ることによって、urcv_msg の処理が終わったかどうかを知ることができる。

【エラーコード (ercd)】

E_OK	正常終了
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSID	予約ID番号（-4≦mbxid≦0）
E_ILADR	不正アドレス（ppk_msgが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
E_ILTIME	不正時間指定（tmout≦-2）

E_NOEXS	オブジェクトが存在していない（mbxidのメイルボックスが存在しない）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でmbxid＜(-4)）
E_CTX	コンテキストエラー（タスク独立部あるいはディスパッチ遅延中のタスクから発行）
E_DLT	待ちオブジェクトが削除された（待ちの間に対象メイルボックスが削除）
E_TMOUT	タイムアウト
E_RLWAI	待ち状態強制解除（待ちの間にter_tsk,rel_waiを受け付け）

共有メモリブロックを獲得する（メモリ更新有）　uget_blk

ローカルメモリブロックを獲得する（メモリ更新有）　uget_lbl

uget_blk:　Get Shared Memory Block with Memory Update
uget_lbl:　Get Local Memory Block with Memory Update

【パラメータ】

＜1＞mplid	MemoryPoolIdentifier	メモリプールID
＜2＞lmpid	LocalMemoryPoolIdentifier	ローカルメモリプールID
bcnt	ContinuousBlockCount	連続領域のブロック数
tmout	Timeout	タイムアウト指定
p_blk	Pointer to BlockStartAddress	メモリブロックの先頭アドレスへのポインタ

［＜1＞uget_blk のみ］
［＜2＞uget_lbl のみ］

【リターンパラメータ】

なし

【解説】

get_blk, get_lbl の処理を行い、それと不可分に、獲得したメモリブロックの先頭アドレスをp_blk の指す領域に設定する。メモリブロック獲得待ち状態の間は、p_blk の内容は変化しない。

この機能は、ユーザタスク側で資源管理を行う場合に利用できる。

【エラーコード (ercd)】

E_OK	正常終了
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSID	予約ID番号（-4≦mplid≦0、-4≦lmpid≦0）

E_PAR	一般的なパラメータエラー（bcnt≦0）
E_ILADR	不正アドレス（p_blkが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）
E_ILTIME	不正時間指定（tmout≦-2）
E_NOEXS	オブジェクトが存在していない（mplid,lmpidのメモリプールが存在しない）
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でmplid,lmpid＜(-4)）
E_CTX	コンテキストエラー（タスク独立部あるいはディスパッチ遅延中のタスクから発行）
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_TMOUT	タイムアウト
E_RLWAI	待ち状態強制解除（待ちの間にter_tsk,rel_waiを受け付け）

共有メモリブロックを返却する（メモリ更新有）　urel_blk

ローカルメモリブロックを返却する（メモリ更新有）　urel_lbl

urel_blk:　Release Shared Memory Block with Memory Update
urel_lbl:　Release Local Memory Block with Memory Update

【パラメータ】

＜1＞mplid	MemoryPoolIdentifier	メモリプールID
＜2＞lmpid	LocalMemoryPoolIdentifier	ローカルメモリプールID
p_blk	Pointer to BlockStartAddress	メモリブロックの先頭アドレスへのポインタ

［＜1＞urel_blk のみ］
［＜2＞urel_lbl のみ］

【リターンパラメータ】

なし

【解説】

p_blk の指す領域の内容を返却すべきメモリブロックの先頭アドレスとして rel_blk, rel_lbl の処理を行い、それと不可分に、p_blk の指す領域を NADR (-1) とする。メモリブロック未返却の間は、p_blk の内容は変化しない。この機能は、ユーザタスク側で資源管理を行う場合に利用できる。

【エラーコード (ercd)】

E_OK	正常終了
E_RSID	予約ID番号（-4≦mplid≦0、-4≦lmpid≦0）
E_ILADR	不正アドレス（p_blkが使用できない値）
E_IDOVR	ID範囲外（インプリメント依存）

E_NOEXS	オブジェクトが存在していない（mplid,lmpidのメモリプールが存在しない）
E_ILBLK	不正メモリブロックの返却、操作
E_OACV	オブジェクトアクセス権違反（拡張SVC以外のユーザタスクからの発行でmplid,lmpid＜(-4)）

タイマハンドラ機能

周期起動ハンドラを定義する
def_cyc

def_cyc: Define Cyclic Handler

【パラメータ】

cyhno	CyclicHandlerNumber	周期起動ハンドラ指定番号
cyhatr	CyclicHandlerAttribute	周期起動ハンドラ属性
cychdr	CyclicHandlerAddress	周期起動ハンドラアドレス
cyhact	CyclicHandlerActivation	周期起動ハンドラ活性状態
cytime	CycleTime	周期起動時間間隔

【リターンパラメータ】

なし

【解説】

周期起動ハンドラは、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。このシステムコールでは、周期起動ハンドラを定義する。

cyhno は複数の周期起動ハンドラを区別するための番号であり、act_cyc, cyh_sts 等で使用される。また、cychdr は周期起動ハンドラの先頭アドレス、cytime は周期起動の時間間隔、cyhact は周期起動ハンドラの一時的な ON / OFF の選択（活性状態）を表わす。cyhact では、次のような値を指定する。

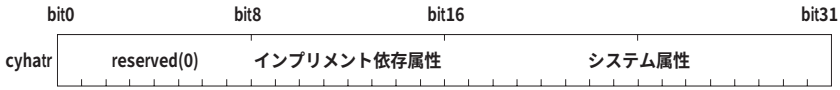
cyhact : = (TCY_OFF ‖ TCY_ON)

TCY_OFF	周期起動ハンドラは起動されない
TCY_ON	周期起動ハンドラは起動される

cyhact = TCY_OFF は周期起動ハンドラの一時的な中断を意味し、この間はハンドラが起動されない。

def_cyc の実行により周期カウントがクリアされるため、def_cyc を実行してからちょうど指定周期経った後に、最初のハンドラ起動が起こることになる。

cyhatr のサイズは標準で4バイトである。cyhatrのフォーマットを［図3.39］に示す。このうち、最上位バイト（bit0～bit7 あるいは 2^31～2^24 のビット）



［図3.39］cyhatrのフォーマット

は未使用であり、第二上位バイト（bit8～bit15 あるいは 2^23～2^16 のビット）はインプリメント依存の属性を表わし、下位ハーフワード（bit16～bit31 あるいは 2^15～2^0 のビット）はシステム属性を表わす。

def_cyc で指定した cyhatr は、cyh_sts により読み出すことができる。

cyhatr のシステム属性の部分では、次のような指定を行うことができる。

cyhatr : = (TA_ASM ‖ TA_HLNG)

TA_ASM	ハンドラがアセンブラで書かれている
TA_HLNG	ハンドラが高級言語で書かれている

TA_HLNG の指定を行った場合には、ハンドラ起動時に直接 cychdr 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 cychdr のアドレスにジャンプする。

なお、cyhatr では、def_cyc 実行後に変更されない静的な情報を扱う。これに対して、cyhact では動的な情報を扱うという点が異なっている。

cychdr=NADR (-1) の指定により、前に定義されていた周期起動ハンドラの定義が解除され、未定義の状態になる。ただし、前に定義されていた周期起動ハンドラの定義解除を行わずに、直接新しいハンドラを定義しても、エラーにはならない。また、もともとハンドラが定義されていない状態（定義が解除された状態）で、cychdr=NADR の指定による定義解除の操作を行っても、エラーにはならない。後者の場合、このシステムコールは正常終了するが、周期起動ハンドラは未定義のままである。

def_cyc では回数の指定が無いので、act_cyc によって cyhact を TCY_OFF にするか、ハンドラを定義解除（あるいは同じcyhnoに対して別の周期起動ハンドラを定義）するまで周期起動が繰り返される。

周期起動ハンドラでのレジスタ退避、復帰はユーザの責任である。

周期起動ハンドラやアラームハンドラをタスク独立部と考えているのは、

周期起動ハンドラやアラームハンドラの定義が有効な間に、def_cyc, def_alm を発行したタスクが終了、削除される可能性があるためである。

周期起動ハンドラはオブジェクトではないため、IDではなく、cyhno により複数ハンドラの区別を行っている。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足 ※ このエラーが発生するかどうかはインプリメント依存である。
E_TNOSPT	タイマ関係の未サポート機能（タイマが利用できない） ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSMD	予約モード、予約オプション（cyhactが不正）
E_RSATR	予約属性（cyhatrが不正）
E_PAR	一般的なパラメータエラー（cyhnoが不正）
E_ILADR	不正アドレス（cychdrが奇数、あるいは使用できない値）
E_ILTIME	不正時間指定（cytimeが不正）

周期起動ハンドラの活性制御を行う
act_cyc

act_cyc:　Activate Cyclic Handler

【パラメータ】

cyhno	CyclicHandlerNumber	周期起動ハンドラ指定番号
cyhact	CyclicHandlerActivation	周期起動ハンドラ活性状態

【リターンパラメータ】

なし

【解説】

周期起動ハンドラの活性状態 (cyhact) を変更する。

cyhact の具体的な指定方法は次のようになる。

cyhact : = (TCY_OFF ‖ TCY_ON) ｜ [TCY_INI]

TCY_OFF	周期起動ハンドラは起動されない
TCY_ON	周期起動ハンドラは起動される
TCY_INI	周期起動ハンドラのカウントが初期化される

cyhact = TCY_ON の指定を行うと、周期の経過とは独立に活性状態を ON にする。OS内部の動作としては、活性状態が OFF の間も指定周期のカウントを行っているため、act_cyc を実行してから最初に周期起動ハンドラが実行されるまでの時間は一定しない。

一方、cyhact = (TCY_ON ¦ TCY_INI) の指定を行うと、活性状態を ON にするのと同時に周期カウントをクリアする。したがって、act_cyc を実行してからちょうど指定周期経った後に、最初のハンドラ起動が起こることになる。

cyhact = (TCY_OFF ¦ TCY_INI) の指定も可能である。この機能を使うことは少ないかもしれないが、次にTCY_ON を指定してから周期起動ハンドラが起動されるまでのタイミングを調整するために利用できる可能性がある。

【エラーコード (ercd)】

E_OK	正常終了
E_RSMD	予約モード、予約オプション（cyhactが不正）
E_PAR	一般的なパラメータエラー（インプリメント依存）
E_NOEXS	オブジェクトが存在していない（cyhnoの周期起動ハンドラが存在しない）

周期起動ハンドラの状態を参照する cyh_sts

cyh_sts:　Get Cyclic Handler Status

【パラメータ】

cyhno	CyclicHandlerNumber	周期起動ハンドラ指定番号
pk_cyhs	Packet of CyclicHandlerStatus	周期起動ハンドラ状態を返すパケットの先頭アドレス

【リターンパラメータ】

なし

【解説】

cyhno で指定した周期起動ハンドラの状態を参照し、その結果を pk_cyhs 以下の領域に返す。

pk_cyhs に返される情報としては、次のようなものがある。

```
cyhatr      /* 周期起動ハンドラ属性 */
cyhact      /* 周期起動ハンドラ活性状態 */
cytime      /* 周期 */
lftime      /* 次の周期起動ハンドラ起動までの残り時間 */
```

このうち、cyhact では、TCY_ON と TCY_OFF の区別のみが返され、TCY_INI の情報が返ることはない。

【エラーコード (ercd)】

E_OK	正常終了
E_PAR	一般的なパラメータエラー（インプリメント依存）
E_ILADR	不正アドレス（pk_cyhsが使用できない値）
E_NOEXS	オブジェクトが存在していない（cyhnoの周期起動ハンドラが存在しない）

アラームハンドラを定義する
def_alm

def_alm: Define Alarm Handler

【パラメータ】

alhno	AlarmHandlerNumber	アラームハンドラ指定番号
alhatr	AlarmHandlerAttribute	アラームハンドラ属性
almhdr	AlarmHandlerAddress	アラームハンドラアドレス
utime	Upper AlarmTime	ハンドラ起動時刻（上位）
ltime	Lower AlarmTime	ハンドラ起動時刻（下位）
tmmode	TimeMode	初期時刻指定モード

【リターンパラメータ】

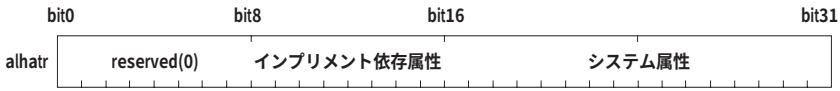
なし

【解説】

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。このシステムコールでは、指定時刻起動ハンドラを定義する。

alhno は複数のアラームハンドラを区別するための番号である。almadr は起動されるアラームハンドラの先頭アドレスを表わす。

alhatr のサイズは標準で4バイトである。alhatr のフォーマットを［図3.40］に示す。このうち、最上位バイト（bit0〜bit7 あるいは 2^31〜2^24 のビット）は未使用であり、第二上位バイト（bit8〜bit15 あるいは 2^23〜2^16 のビット）はインプリメント依存の属性を表わし、下位ハーフワード（bit16〜bit31 あるいは 2^15〜2^0 のビット）はシステム属性を表わす。



［図3.40］ alhatrのフォーマット

def_alh で指定した alhatr は、alh_sts により読み出すことができる。

alhatr のシステム属性の部分では、次のような指定を行うことができる。

alhatr : = (TA_ASM ‖ TA_HLNG)

TA_ASM	ハンドラがアセンブラで書かれている
TA_HLNG	ハンドラが高級言語で書かれている

TA_HLNG の指定を行った場合には、ハンドラ起動時に直接 almhdr 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 almhdr のアドレスにジャンプする。

utime, ltime により、アラームハンドラの起動時刻を指定する。標準では、utime 16 ビット、ltime 32 ビットの計 48 ビットを使って時刻の指定が行われる。tmmode により、次のようにして絶対時刻と相対時刻の指定が可能である。

tmmode : = (TTM_ABS ‖ TTM_REL)

TTM_ABS	絶対時刻での指定
TTM_REL	相対時刻での指定

指定した時刻が過去の時刻であった場合には、エラー E_ILTIME となる。

def_alm で相対時刻を指定し、時刻 0 を指定した場合には、即座にアラームハンドラが呼び出される。また、def_alm で指定した時刻が現在時刻に近い場合の def_alm の動作は、即座にアラームハンドラを起動するか（同時刻あるいはわずかに未来の時刻を指定した場合）、def_alm が E_ILTIME のエラーになるか（わずかに過去の時刻を指定した場合）のどちらかである。def_alm が正常終了した場合には、必ずアラームハンドラが起動されなければならない。

almhdr = NADR (-1) の指定により、前に定義されていたアラームハンドラの定義が解除され、未定義の状態になる。ただし、前に定義されていたアラームハンドラの定義解除を行わずに、直接新しいハンドラを定義しても、エラーにはならない。また、もともとハンドラが定義されていない状態（定義が解除された状態）で、almhdr = NADR の指定による定義解除の操作を行っても、エラーにはならない。後者の場合、このシステムコールは正常終了するが、アラームハンドラは未定義のままである。

アラームハンドラでのレジスタ退避、復帰はユーザの責任である。

アラームハンドラの中で rel_wai を発行することにより、タイムアウトに似た機能を実現することができる。

アラームハンドラが起動された後は、自動的にアラームハンドラの定義が解除されるものとする。定義の解除はアラームハンドラ起動時に行われ、アラームハンドラの実行中は、既に定義が解除された状態になっている。たとえば、アラームハンドラの中で自分自身の alhno に対して alh_sts を実行すると、E_NOEXS のエラーになる。(自分自身の alhno は、アラームハンドラ起動時のパラメータとして知ることができる。)

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足 ※ このエラーが発生するかどうかはインプリメント依存である。
E_TNOSPT	タイマ関係の未サポート機能 (タイマが利用できない) ※ このエラーが発生するかどうかはインプリメント依存である。
E_RSMD	予約モード、予約オプション (tmmodeが不正)
E_RSATR	予約属性 (alhatrが不正)
E_PAR	一般的なパラメータエラー (alhnoが不正)
E_ILADR	不正アドレス (almhdrが奇数、あるいは使用できない値)
E_ILTIME	不正時間指定 (utime,ltimeが不正)

アラームハンドラの状態を参照する alh_sts

alh_sts: Get Alarm Handler Status

【パラメータ】

alhno	AlarmHandlerNumber	アラームハンドラ指定番号
pk_alhs	Packet of AlarmHandlerStatus	アラームハンドラ状態を返すパケットの先頭アドレス

【リターンパラメータ】

なし

【解説】

alhno で指定したアラームハンドラの状態を参照し、その結果を pk_alhs 以下の領域に返す。

pk_alhs に返される情報としては、次のようなものがある。

```
alhatr      /* アラームハンドラ属性 */
lfutime     /* アラームハンドラ起動までの残り時間（上位） */
lfltime     /* アラームハンドラ起動までの残り時間（下位） */
```

【エラーコード (ercd)】

E_OK	正常終了
E_PAR	一般的なパラメータエラー（インプリメント依存）
E_ILADR	不正アドレス（pk_alhsが使用できない値）
E_NOEXS	オブジェクトが存在していない（alhnoのアラームハンドラが存在しない）

タイマハンドラから復帰する
ret_tmr

ret_tmr: Return from Timer Handler

【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

【解説】

タイマハンドラ（周期起動ハンドラ、アラームハンドラ）から戻る。ret_int 同様、ディスパッチの起こる可能性がある。

同じタスク独立部からの戻りシステムコールである ret_int と統合しないのは、ハンドラ起動時にOSが介入するかどうかという点で違いがあるためである。すなわち、ret_int がハードウエアで直接起動されたタスク独立のハンドラからのリターンを表わすのに対して、ret_tmr はOSから起動されたタスク独立のハンドラからリターンを表す。

【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には、メッセージバッファへのロギングや、コンソールへのエラーメッセージの表示などを行うのが望ましいが、詳細な動作はインプリメント依存となる。

E_CTX コンテキストエラー（タスク部、準タスク部から実行）

第三部　第四章

ITRON2システム操作機能

この章では、ITRON2のシステム操作機能に関する説明を行う。システム操作機能とは、実行環境の整備などのために必要となる機能である。システム操作機能は、一般ユーザには非公開とするのが普通である。

拡張SVCサポート機能

拡張SVC用終了ハンドラの定義
sdef_ext

拡張SVC用CPU例外ハンドラの定義
sdef_cex

拡張SVC用システムコール例外ハンドラの定義
sdef_sex

sdef_ext:　Define ExitHandler for SVC
sdef_cex:　Define CPUExceptionHandler for SVC
sdef_sex:　Define SystemcallExceptionHandler for SVC

【パラメータ】

s_fncd	SVCFunctionCode	拡張SVC機能コード
exhatr	ExceptionHandlerAttribute	例外ハンドラ属性
＜1＞exthdr	ExitHandlerAddress	終了ハンドラアドレス
＜2＞cexhdr	CPUExceptionHandlerAddress	CPU例外ハンドラアドレス
＜3＞sexhdr	SystemCallExceptionHandlerAddress	システムコール例外ハンドラアドレス

［＜1＞sdef_ext のみ］
［＜2＞sdef_cex のみ］
［＜3＞sdef_sex のみ］

【リターンパラメータ】

なし

【解説】

拡張SVCハンドラに対する例外ハンドラや終了ハンドラを定義する。

拡張SVCハンドラに対する例外ハンドラや終了ハンドラの実行モード（リング、レベル）は、元の拡張SVCハンドラの実行モード（リング、レベル）と同じである。

exhatrのサイズは標準で4バイトである。exhatrのフォーマットを［図3.41］に示す。このうち、最上位バイト（bit0～bit7 あるいは 2^31～2^24 のビット）は未使用であり、第二上位バイト（bit8～bit15 あるいは 2^23～2^16 のビット）はインプリメント依存の属性を表わし、下位ハーフワード（bit16～bit31 あるいは 2^15～2^0 のビット）はシステム属性を表わす。

exhatr のシステム属性の部分では、次のような指定を行うことができる。

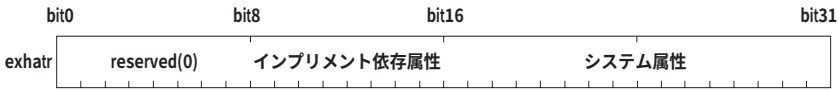
exhatr : = (TA_ASM ‖ TA_HLNG)

TA_ASM	ハンドラがアセンブラで書かれている
TA_HLNG	ハンドラが高級言語で書かれている

TA_HLNG の指定を行った場合には、ハンドラ起動時に直接 exthdr, cexhdr, sexhdr 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 exthdr, cexhdr, sexhdrのアドレスにジャンプする。

exthdr, cexhdr, sexhdr = NADR (-1) の指定により、前に定義されていた例外（終了）ハンドラの定義が解除され、未定義の状態になる。ただし、前に定義されていた例外（終了）ハンドラの定義解除を行わずに、直接新しいハンドラを定義しても、エラーにはならない。また、もともとハンドラが定義されていない状態（定義が解除された状態）で、exthdr, cexhdr, sexhdr = NADR の指定による定義解除の操作を行っても、エラーにはならない。後者の場合、このシステムコールは正常終了するが、例外（終了）ハンドラは未定義のままである。

s_fncd=TFN_CMN により、拡張SVCハンドラに対する共通の例外ハンドラが定義される。



［図3.41］exhatrのフォーマット

【補足事項】

拡張SVCハンドラに対する例外ハンドラを機能コード別に定義しているのは、メーカー提供の拡張SVCハンドラ（入出力、ファイルなどを含む）に対する例外ハンドラと、ユーザ定義の拡張SVCハンドラに対する例外ハンドラを分けて定義できるようにするためである。拡張SVCハンドラに対する例外ハンドラを一つにまとめてしまった場合、例えば、入出力管理の拡張SVCで例外ハンドラを使ってしまうと、ユーザ定義の拡張SVCでは例外ハンドラが使えない、ということになってしまう。

【エラーコード (ercd)】

E_OK	正常終了
E_NOSMEM	システムメモリ不足 ※ このエラーが発生するかどうかはインプリメント依存である。
E_PAR	一般的なパラメータエラー（インプリメント依存）
E_ILFN	不正機能コード番号（s_fncd=0, s_fncd≦-2）
E_NOEXS	オブジェクトが存在していない(s_fncdの拡張SVCが未定義)
E_RSATR	予約属性（exhatrが不正）
E_ILADR	不正アドレス（exthdr, cexhdr, sexhdrが奇数、あるいは使用できない値）

第三部 第五章

ITRON2標準インタフェース

この章では、ITRON2のシステムコールインタフェースのうち、プロセッサに依存しない部分の仕様に関する説明を行う。具体的には、機能コード、エラーコード、言語Cインタフェースなどの説明がここに含まれる。

タスク管理情報とID番号

ITRON2で使用するID番号のサイズは、アセンブラや言語Cのインタフェース上は4バイト（32ビット）とする。ID番号を4バイトとしたのは、ITRON2が32ビットのプロセッサを対象としていること、将来の拡張性を考慮したこと等による。ただし、ユーザが利用できるID番号の範囲をターゲット毎に制限することは構わないので、一般には、ID番号の内部表現は2バイトで済む。ID番号の内部表現を2バイトとした場合、リターンパラメータとしてID番号を返すシステムコール（get_tidなど）では、2バイトのID番号を符号拡張して4バイトとすれば良い。

ITRON2では、タスクID番号の割り当て方法について、次のような約束を設けておく。

［最小値］～(-5) ---　システムタスク用ID番号
(-4)～(-1)　---　reserved
0 以外にも reserved が欲しいため。
0　---　自タスクを表わす(TSK_SELF)
1～［最大値］　---　ユーザタスク用ID番号

※［最小値］,［最大値］の具体的な値は、インプリメントやターゲットに依存して決まる。普通は、OSのジェネレーション時にこれらの値を指定する。

ITRON2のタスク優先度は2バイトで表現され、

(-16)～255　［(-4)～0 は reserved］

を使用可能な優先度とする。数字の小さい方が高い優先度になる。(-16)～255 という値は、(-16)～15 の32個の優先度を別扱いして高速化することを考慮したものであるが、実際のインプリメント方法はもちろん自由である。

また、タスク優先度の割り当て方法についても、タスクIDと同じように次のような約束を設けておく。

(-32768)〜(-17)	---	reserved
(-16)〜(-5)	---	システム用優先度
(-4)〜(-1)	---	reserved
		0 以外にも reserved が欲しいため。
0	---	自タスクの優先度を表わす
1〜255	---	ユーザ用優先度
256〜32767	---	reserved

ここで、優先度もタスクIDと同様に負の値がシステム用となっているが、これは一応の目安（運用方法のガイドライン）に過ぎないものである。ユーザが負の優先度を使っても構わない。優先度は、ID番号と違って大小関係に意味を持っており、ユーザタスクの優先度が、必ずしもシステムタスクの優先度より低いとは限らないためである。一般のユーザタスクは正の優先度を使用し、入出力タスク、ファイル管理タスク、デバッガタスク等のシステムタスクは絶対値の小さい負の優先度を使用し、ユーザタスクで特にシステムタスクよりも優先度を高くしたいものは絶対値の大きな負の優先度を使用する、という優先度の割当て方を行えば良い。

アセンブラインタフェース

アセンブラインタフェース共通原則

・ITRON2のアセンブラインタフェースは、原則としてレジスタを使ったインタフェースとし、パケットを使用しない。これは、以下の理由による。
— コンパイラやOSのインプリメントでの最適化に成功すれば、最高の性能が出せる可能性を持っている。
— パケット領域の確保が要らないので、使いやすい場合が多い。
— ITRONのシステムコールのパラメータはあまり多くないため、パラメータをレジスタに置いても、レジスタが足りなくなるということはない。cre_tsk, cyc_wup, def_cyc など一部のシステムコールでは、パラメータが多いものもあるが、パラメータが多いといっても 5〜6個なので、レジスタ渡しの仕様とする。ただし、XXX_sts, get_ver のシステムコールで返す情報については、すべてパケットを使用するものとする。

・レジスタの使い方は、できる限り次の原則に合わせる。

R0: 機能コード(fncd)、エラーコード(ercd =)
R1: ID番号(XXXid)、識別コード(XXXno,intvec) など
R2: パケットアドレス(pk_XXX)、属性(XXXatr)、ビットパターン(XXXptn)、コード(XXXcd) など
R3: ハンドラアドレス(XXXhdr)、スタートアドレス、タイムアウト(tmout) など

ここで、R0, R1, R2... は一般的な汎用レジスタの名称を表わしたものであるが、当然のことながら、正確な名称や数はプロセッサによって異なっている。最もパラメータの多いシステムコール(cyc_wup, def_alm) では、パラメータを置くレジスタとして、R0〜R6 まで使用することがある。

システムコール実行後のフラグ変化

システムコール実行後のフラグの値については、特に規定せず、不定値になるものとする。

ITRON1では、エラーの有無に応じてゼロフラグのセットを行うことを推奨していた。しかし、OSがせっかくフラグをセットしても、必ずしも参照されるとは限らないし、言語Cを使えば全く不要な機能である。また、フラグセットのためのオーバーヘッドも無視できない。

プロセッサによっては、OSで特にフラグの操作をしない場合に、トラップ命令のハードウエア処理によって自然にフラグを保存できることがある。しかし、将来システムコールをハードウエア化することがあれば、フラグのセットを行う方が望ましいということになる可能性もあり、必ずしもフラグを保存するように規定するのが良いとは言えない。そこで、現段階では、システムコール実行後のフラグ値に関して規定を設けないことにする。

機能コード

ITRONでは、システムコールの種類を区別する番号として、「機能コード」を用いている。機能コードをあるレジスタ（通常はR0）に置き、それからトラップ命令を発行することによって、対応するシステムコールが実行される。

ITRON2およびμITRONにおける機能コードの割当方針を以下に示す。

【機能コードの割当て方針】

- ― TRONの全体方針に合わせて、機能コードの標準値には負の値を使用している。
- ― 8の倍数に対するアライメントや、共通機能(cre_XXX, XXX_sts, uXXX_YYY, sXXX_YYY など) に対する共通ビットパターンを考慮した。
- ― 必ずしも、仕様書の出現順に機能コードを割り当てるわけではない。ただし、仕様書の機能グループ毎には、まとまった範囲の機能コードを割当てるようにする。(強制例外などシステムコール数が少ないものについては、他のグループにまとめられる場合もある。)
  また、仕様書での説明の順序は、機能コードとは直接関係しない。
- ― μITRONで使用する iXXX_YYY, pXXX_YYY は、それぞれ一つのグループを作るので、一般のシステムコールと混在させるのではなく、番号を飛ばした別グループとしている。

ITRON2の機能コードは、μITRONに適用することも可能である。実際の機能コードの値（ITRON2とμITRONで共通）を、巻末のレファレンスに示す。

言語Cインタフェースとニモニック

μITRONやITRON2では、ニモニックの付与に関して、以下のような共通原則を設けている。

・データタイプの名称
データタイプの名称としては、構造体も含めてすべて大文字を使用する。
ポインタのデータタイプは ～P の名称とする。
構造体については、T_～ の名称とする。

・定数のニモニック
エラーコードは E_～ の名称とする。
例外クラスの番号(0～31)は EC_～ の名称、例外クラスのマスク（2のべき乗を使ったビットパターン）は ECM_～ の名称とする。
オブジェクト、ハンドラの属性は、TA_～ の名称とする。
特定のシステムコールやパラメータでのみ使用するモードなどの名称は、Txy_～ とする。xy の部分は、システムコールやパラメータに応じて変化する。

・パラメータ、リターンパラメータの名称
パラメータ、リターンパラメータの名称に関しては、次のような原則を設ける。

p_	リターンパラメータを入れる領域へのポインタ
pk_	パケットアドレス
～cd	～コード
ar_	配列へのポインタ
i～	初期値
～sz	サイズ

・パラメータ名が同じであれば、原則として同じデータタイプを持つようにする。

また、ITRON2の言語Cインタフェースでは、次のような共通原則を設けている。

・ITRON2では、プログラムの書きやすさを考えて、できるだけパケットを使用しない方針とする。これは、アセンブラインタフェースをレジスタ主導型とすることにも対応している。

・関数としての戻り値は、原則としてシステムコールのエラーコード(ercd)となる。正常終了の場合には ercd は 0 となり、エラーが発生した場合には ercd は負となる。

・エラーコード以外のリターンパラメータ(XXX)を返す場合には、リターンパラメータを入れる領域へのポインタ(p_XXX)をパラメータとして指定する。

・リターンパラメータへのポインタは、パラメータよりも先に置くものとする。
アセンブラインタフェースでの指定がポインタ（パケットアドレス）になっているものについても、意味的にリターンパラメータに相当するものであれば、この原則が適用される（XXX_sts など）。
また、uXXX_YYY のシステムコールで使用するメモリ更新用のポインタについても、この原則が適用される。そのため、例えば、rcv_msg と urcv_msg や get_blk と uget_blk の言語Cインタフェースが全く同じになる。

・リターンパラメータへのポインタ、およびパラメータの中での順番は、原則としてアセンブラインターフェースのレジスタ番号(R0, R1...) の順番と同じになるようにする。
ただし、次の原則が適用される場合には、そちらの方を優先する。
— tmout は最後のパラメータとする。(wai_flg, cal_por, rcv_mbf)
— メッセージのアドレスとサイズを両方指定する場合は、アドレスを先に、サイズを後にする。(snd_mbf, rcv_mbf)

ITRON2で使用する標準データタイプ、および言語Cインタフェースの標準仕様を巻末のレファレンスに示す。また、システムで共通に使用する定数のニモニックや標準値、それらの共通定数の適用されるシステムコール、および構造体のパケット形式を巻末のレファレンスに示す。

ITRON1において、タスクや例外ハンドラをすべて高級言語で書こうとした場合には、OSを作る際に、特定の高級言語の実行環境（関数のパラメータの渡し方など）を仮定しておく必要があった。それと異なった環境をもつ高級言語（例えば他社製のCコンパイラ）を利用しようとすると、高級言語の関数名を直接 cre_tsk や def_extのパラメータとすることは難しくなる。場合によっては、ユーザに高級言語のスタートアップ処理ルーチンを見せて、タスクやハンドラ毎に個別に対応してもらわなければならない。これに対する対策として、高級言語対応ルーチンが用意されていたが、あまり活用されていなかった。

そこで、ITRON2では、高級言語に対応するためにオブジェクト属性やハンドラ属性の機能を利用する方針とした。具体的には、タスクやハンドラの属性として、タスクやハンドラが高級言語で書かれているかアセンブラで書かれているかの区別を行う。後者であれば、タスクやハンドラをスタートアドレスから直接起動するが、前者であれば、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、その中からユーザの定義したタスクスタートアドレスやハンドラアドレスに間接ジャンプすることになる。この場合、OSから見ると、タスクスタート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。こういった方法により、異なる言語環境にも容易に対応できるようにした。

また、ITRON2では、タスクやハンドラを言語Cの関数として書いた場合に、単なる関数のリターン（return や "}"）を行うだけでも、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が自動的に実行されるものとしている。このため、最後の ext_tsk や ret_XXX を書き忘れたとしても、できるだけ正常な実行を続けることが可能になっている。インプリメント上は、高級言語対応ルーチン(h_XXXhdr) において、高級言語で書かれたユーザ定義ハンドラ(XXXhdr) の呼び出しをサブルーチンジャンプとし、そこから戻ってきた後でret_XXX や ext_tsk を実行するようにすれば良い。

なお、アセンブラの場合はオーバーヘッドを避けることが重要なので、ret_XXX を発行するトラップ命令(TRAPA) の代わりにサブルーチンリターン命令(RTS) を書いたとしても、正常なリターンは行われない。

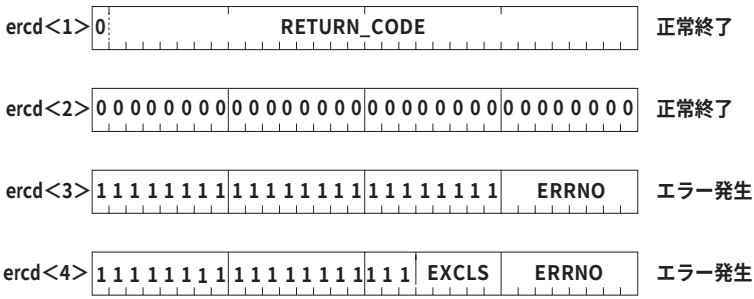
エラーコードとシステムコール例外

ITRON2では、各システムコールで発生するエラーに対して、それを表わすニモニックとエラーコード値を標準化している。また、これらの標準は、μITRONとITRON2で共通化されている。

エラーコードの標準値は負の数になっており、その中に ercls（例外クラス）と errno の情報を含んでいる。例外クラスは、例外の重要度や分類を示すものである。ITRON2においては、システムコール例外の起動の有無を決めるために例外クラスが利用されている。

ITRON2で使用するエラーコードのフォーマットを［図3.42］に示す。

— excls (4≦excls≦31) により、発生するシステムコール例外の例外クラスを表わす。
— errno (1≦errno≦255) により、同じ例外クラスに属する複数のエラーを区別する。
— エラー発生の場合の ercd の値は (- (excls ≪ 8) - errno) によって表わされ、原則として ercd<4>のフォーマットを使用する。ただし、'≪' は左シフトである。



＊EXCLSはexclsの1の補数、ERRNOはerrnoの2の補数である。

［図3.42］ITRON2におけるエラーコードのフォーマット

— excls が異なるのに errno が等しくなる組み合せが生じないようにする。つまり、excls の部分が無くても、errno のみで ercd の識別を可能とする。（errno には、excls と独立にシーケンシャルな番号を与える。）
— ercd＜1＞のフォーマットは、μITRONでは使用していない。ITRON2では、ファイル管理で ercd＜1＞のフォーマットを使用している。
— デバッガ等で ercd の値を直接見る場合に、errno の部分のみの値を知りたければ、ercd を1バイトの符号付きの数として扱えば良い。
— システムコール例外が発生するのは、ercd＜4＞の場合のみである。
— ret_svc のパラメータで指定する s_ercd も、同じフォーマットに従う。すなわち、s_ercd の excls 部分のフィールドで指定した値によって、拡張SVC終了時に起動されるシステムコール例外の例外クラスが決まる。なお、システムコール例外を発生しない ercd＜3＞のフォーマットは、ITRON2の場合、拡張SVCからの戻り値 (s_ercd) としてのみ使用されている。

ITRON2およびμITRONで使用する例外クラス (excls) を、巻末のレファレンスに示す。

ret_svc で指定した s_ercd の excls が EC_NONE の場合（EXCLSフィールドがすべて1の場合）には、s_ercd全体の値が 0 でなくても、拡張SVCから戻る際のシステムコール例外は発生しない。すなわち、EC_NONE の例外クラスのエラー (-256≦s_ercd≦-1) に対する emsptn は、clr_ems 実行の有無にかかわらず、常にマスクされた状態となっている。また、ret_svc で指定した s_ercd の excls が、EC_TER, EC_FEX, EC_CEX など特別な意味をもった例外クラスであった場合にも、システムコール例外は発生しない。

拡張SVCから返されるエラーコード s_ercd で示される例外クラスと、例外マスクの値、システムコール例外の発生の有無との関係は、［表3.5］のようになる。

μITRONやITRON2では、エラーコードのニモニックの付与に関して、以下のような共通原則を設けている。

— 汎用的に使用されるエラーは汎用的な名称とする。特殊な意味を持つエラーは、誤解が起きないように、特殊な意味を持つことが想像できるようにする。— 他のオブジェクトでも発生することが予想されるが、実際は一種類のオブジェクトにのみ関連して発生するエラーがある。

s_ercd の例外クラス	対応する例外マスク (emsptn) の意味	ret_svc によるシステムコール例外
EC_NONE [0]	ECM_NONEはクリア不可	発生しない
EC_TER [1]	ECM_TERは終了例外に対するマスク	発生しない
EC_FEX [2]	ECM_FEXは強制例外に対するマスク	発生しない
EC_CEX [3]	ECM_CEXはCPU例外に対するマスク、セット不可	発生しない
EC_SYS [4]	ECM_SYSはシステムコール例外に対するマスク （EC_SYSのシステムコール例外の場合に限り タスク独立部に対する例外ハンドラを起動）	ECM_SYSに依存して発生
EC_NOMEM [5]	ECM_NOMEMはシステムコール例外に対するマスク	ECM_NOMEMに依存して発生
EC_RSV [6]	ECM_RSVはシステムコール例外に対するマスク	ECM_RSVに依存して発生
EC_PAR [7]	ECM_PARはシステムコール例外に対するマスク	ECM_PARに依存して発生
EC_FILE [20]	ECM_FILEはシステムコール例外に対するマスク	ECM_FILEに依存して発生
[21]	システムコール例外に対するマスク	例外マスクに依存して発生
	（今後の拡張のために予約された例外クラス）	
[23]	システムコール例外に対するマスク	例外マスクに依存して発生
[24]	システムコール例外に対するマスク	例外マスクに依存して発生
	（インプリメンタが自由に利用できる例外クラス）	
[27]	システムコール例外に対するマスク	例外マスクに依存して発生
[28]	システムコール例外に対するマスク	例外マスクに依存して発生
	（ユーザが自由に利用できる例外クラス）	
[29]	システムコール例外に対するマスク	例外マスクに依存して発生
[30]	ECM_ABOは終了ハンドラ実行中を示す	発生しない
[31]	ECM_SUSはサスペンド遅延要求を示す	発生しない

＊EC_SYSについてもシステムコール例外を発生可能としている。
この場合は、idef_sexで定義されたタスク独立部に対する例外
ハンドラが起動される。
＊例外クラス 21〜23 は今後のために予約しておき、24〜27 を
インプリメンタに解放する例外クラス、28〜29をユーザに解
放する例外クラスとした。

［表3.5］拡張SVCから返されるエラーとシステムコール例外

このような場合は、誤解を防ぐために、エラー名称にそのオブジェクトの名称を含めるようにする。

例えば、ファイルのアクセス権違反を示すエラーは、メモリのアクセス権違反と区別するために 'F'の名称を含める。

— ITRON1で使っていたエラーコードとの連続性、整合性を考慮する。

— 「不正エラー（意味的に間違った操作）」と「予約機能エラー（将来の機能追加によりエラーとならなくなるもの）」の区別がはっきりしている部分は、その区別を行う。

— 例外クラスについては、エラーの内容や意味だけではなく、そのエラーに対する対応方法（例えばシステムコール例外ハンドラを起動することに意味があるかどうか）まで考慮して分類を行う。

ITRON2およびμITRONで使用するエラーコードのニモニックとその標準値を、巻末のレファレンスに示す。

ITRON2において、インプリメント依存のエラーコード、およびユーザ用（拡張SVC専用）のエラーコードとしては、次の値を使用する。

インプリメント依存 :

errno = 225〜255, ERRNO = B'000XXXXX
ercd = B'11111111111111111111YYYYY000XXXXX

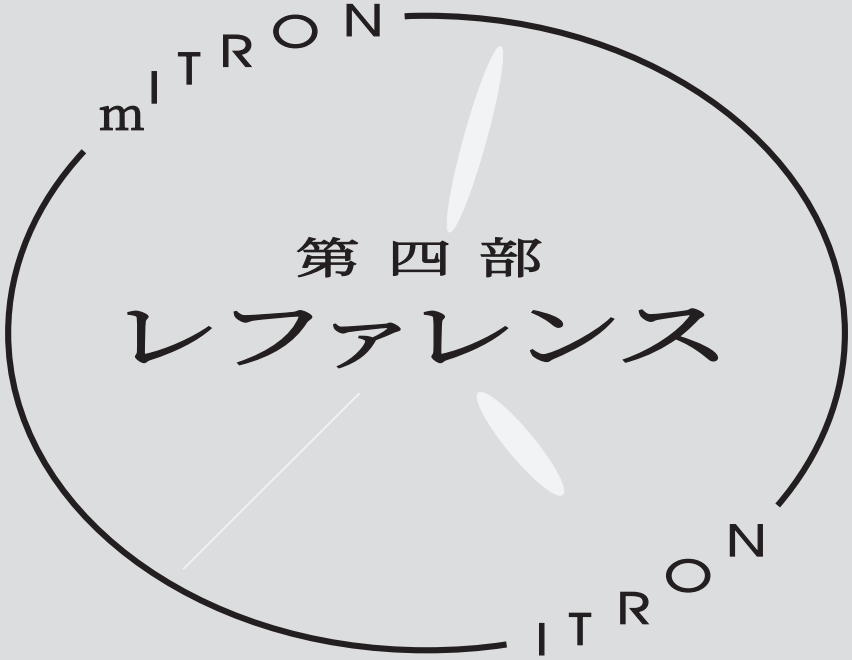
ユーザ用（拡張SVC用）:

errno = 193〜223, ERRNO = B'001XXXXX
ercd = B'11111111111111111111YYYYY001XXXXX

拡張SVC用として 193〜223 の errno を割当ててあるが、拡張SVCでは、これ以外のエラーコード（1〜96の標準エラーコードを含む）を返しても構わない。193〜223 の番号の割当ては、拡張SVC用以外の目的ではこの番号を使わないという意味である。

個々のエラーに対応するエラークラス (ercls) については、エラーの種類や意味に応じて、0 または適当な値を入れる。

97〜128, 129〜192 の errno については、標準入出力機能、MMUサポート機能、ファイル管理機能などのために予約されている。



第四部
レファレンス

第四部　第一章

μITRON・ITRON2共通レファレンス

機能コード

【機能コードの範囲】

reserved:	H'fe00～H'fe7f	(-512)～(-385)	128個
ファイル管理、デバイス管理:	H'fe80～H'febf	(-384)～(-321)	64個
デバッグサポート:	H'fec0～H'feff	(-320)～(-257)	64個
システム構成用:	H'ff00～H'ff1f	(-256)～(-225)	32個
標準システムコール（I/O,MMU含）:	H'ff20～H'fffb	(-224)～(-5)	220個
reserved:	H'fffc～H'0000	(-4)～0	5個
ユーザ用（拡張SVC）:	H'0001～	1～	

【機能コードの標準値】

※ '-' は、予約または機能コード不要の可能性があるシステムコールである。

※ 'μ' は、ITRON2で使用せず、μITRONでのみ使用するシステムコールである。

※「システム構成用」の［公開］と［非公開］の区別は、厳密なものではない。

	+H'0				+H'4				+H'8				+H'c			
H'ff00	システム構成用、インプリメント依存 [公開]															
H'ff10	システム構成用、インプリメント依存 [非公開]															
H'ff20	（タスク独立部専用システムコール iXXX_YYY）															
	µi snd tmb	-	µi snd mbf	-	µi snd msg	µi sig sem	µi set flg	µi wup tsk	-	µi rsm tsk	-	µi sus tsk	µi rei wai	-	µi rot rdq	µi chg pri
H'ff30	（ポーリング機能 pXXX_YYY）								（タスク付属メッセージ XXX_tmb）				（割込み）			
	µp rcv tmb	µp get tbl	µp rcv mbf	µp get blk	µp rcv msg	µp req sem	µ pol flg	µc pol flg	-	µ snd tmb	-	µ rcv tmb	µ tmb sts	-	µ ret rst	µ def rst
H'ff40	標準入出力管理												reserved			
	put lin	get lin	put chr	get chr	-	-	cio ctl	cio sts	req gio	-	-	-	-	-	-	-
H'ff50	reserved															
	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
H'ff60	MMUサポート															
	-	-	sha map	trf map	map sts	set prt	del map	cre map	-	mov dat		set dat	spc sts	-	del spc	cre spc
H'ff70	ローカルメモリプール管理								ランデブ（ポート）							
	u rel lbl	rel lbl	u get lbl	get lbl	lmp sts	lbl sts	del lmp	cre lmp	fwd	rpl	acp	cal	por sts	rdv sts	del	cre
H'ff80	メッセージバッファ								拡張SVCサポート							
	-	snd	-	rcv	mbf sts	-	del	cre	-	-	-s end	-s ret	-s def fex	s def ext	s def sex	s def cex
H'ff90	例外管理、強制例外															
	set ems	clr ems	- ref exd	-	- ems sts	i def ext	i def sex	i def cex	ras fex	-	end	ret	def fex	def ext	def sex	def cex
H'ffa0	タイマハンドラ								時間管理							
	-	-	act cyc	- ret tmr	cyh sts	alh sts	def cyc	def alm	can cyc	cyc wup	-	dly tsk	get tim	set tim	-	-
H'ffb0	メモリプール管理								割込み管理							
	u rel blk	rel blk	u get blk	get blk	mpl sts	blk sts	del	cre	µ dis int	µ ena int	- ret wup	- ret int	µ iXX sts	chg ims	-	def int
H'ffc0	メイルボックス								セマフォ							
	u sng msg	snd msg	u rcv msg	rcv msg	mbx sts	-	del	cre	u sig	sig	u wai	wai	sem sts	-	del	cre
H'ffd0	イベントフラグ								タスク付属同期							
	set	clr	wai	µc wai	flg sts	-	del	cre	can wup	wup	slp	wai	f rsm	rsm	-	sus
H'ffe0	タスク管理															
	ras ext	rel wai	-	-	rot rdq	chg pri	abo	ter	get tid	sta	exd	ext	tsk sts	hdr sts	del	cre
H'fff0	システム管理、拡張SVCサポート								reserved							
	get ver	-	-	ret svc	psw sts	-	s def svc	def svc	-	-	-	-	=	=	=	=

例外クラス

excls:

```
EC_NONE     0    /* システムコール例外を発生しないクラス */
EC_TER      1    /* 終了要求クラス */
EC_FEX      2    /* 強制例外クラス */
EC_CEX      3    /* CPU例外クラス */
EC_SYS      4    /* システムエラー例外クラス */
EC_NOMEM    5    /* メモリ不足エラー例外クラス */
EC_RSV      6    /* 予約機能エラー例外クラス */
EC_PAR      7    /* パラメータエラー例外クラス */
EC_OBJ      8    /* 不正オブジェクト指定例外クラス */
EC_ACV      9    /* アクセス権違反例外クラス */
EC_CTX      10   /* コンテキストエラー例外クラス */
EC_EXEC     11   /* 実行時エラー例外クラス */
EC_EXCMP    12   /* 実行終了時エラー例外クラス */
EC_RLWAI    13   /* 待ち状態強制解除例外クラス */
EC_RSLT     14   /* 処理実行結果を示す例外クラス */
EC_IO       15   /* 標準入出力(GIO/CIO)関連例外クラス */
EC_MMU      16   /* MMUサポート関連例外クラス */
EC_xxxx     17   /* 予備（ネットワーク関連など）*/
EC_xxxx     18   /* 予備（ネットワーク関連など）*/
EC_xxxx     19   /* 予備（ネットワーク関連など）*/
EC_FILE     20   /* ファイル管理関連例外クラス
                    （デバイス管理を含む）*/
```

eclsptn:

```
/* excls を2のべき乗のパターンにしたもの */
/* 以下、'≫' は論理右シフトを示す */
ECM_TER      ((UW) H'80000000 ≫ EC_TER)      /* H'40000000 */
ECM_FEX      ((UW) H'80000000 ≫ EC_FEX)      /* H'20000000 */
ECM_CEX      ((UW) H'80000000 ≫ EC_CEX)      /* H'10000000 */
ECM_SYS      ((UW) H'80000000 ≫ EC_SYS)      /* H'08000000 */
ECM_NOMEM ((UW) H'80000000 ≫ EC_NOMEM)  /* H'04000000 */
ECM_RSV      ((UW) H'80000000 ≫ EC_RSV)      /* H'02000000 */
ECM_PAR      ((UW) H'80000000 ≫ EC_PAR)      /* H'01000000 */
ECM_OBJ      ((UW) H'80000000 ≫ EC_OBJ)      /* H'00800000 */
ECM_ACV      ((UW) H'80000000 ≫ EC_ACV)      /* H'00400000 */
ECM_CTX      ((UW) H'80000000 ≫ EC_CTX)      /* H'00200000 */
ECM_EXEC     ((UW) H'80000000 ≫ EC_EXEC)     /* H'00100000 */
ECM_EXCMP ((UW) H'80000000 ≫ EC_EXCMP)  /* H'00080000 */
ECM_RLWAI ((UW) H'80000000 ≫ EC_RLWAI)  /* H'00040000 */
ECM_RSLT     ((UW) H'80000000 ≫ EC_RSLT)     /* H'00020000 */
ECM_IO       ((UW) H'80000000 ≫ EC_IO)       /* H'00010000 */
ECM_MMU      ((UW) H'80000000 ≫ EC_MMU)      /* H'00008000 */
ECM_FILE     ((UW) H'80000000 ≫ EC_FILE)     /* H'00000800 */

ECM_SEX          H'08000000 =  ECM_SYS
ECM_ABO          H'00000002
ECM_SUS          H'00000001
```

※ EC_NONE に対応する eclsptn (ECM_NONE = H'80000000) は、必要性が無いので用意されていない。

エラーコード

●システムコール例外を発生しないクラス (EC_NONE=0)　　errno: 0

ニモニック	ercd	説明
E_OK	0	正常終了 ITRON1の TE_OK に対応する。

●終了要求に対応する例外クラス (EC_TER=1)　　errno: 1

※ 通常のシステムコールでこのエラーコードを返すことはない。終了要求の発生などによって、拡張SVCハンドラから強制的に戻るような場合にこのエラーコードを返すことがある。

※ 拡張SVCハンドラからこのエラーコードを返した場合、ECM_TER がクリアされていても、このエラーコードに対するシステムコール例外は発生しない。

ニモニック	ercd	説明
E_TER	-H'101 (-(EC_TER≪8)-1)	終了要求発生

●CPU例外に対応する例外クラス (EC_CEX=3)　　errno: 3

※ 通常のシステムコールでこのエラーコードを返すことはない。CPU例外ハンドラが未定義の状態でCPU例外が発生し、終了ハンドラが起動された場合に、終了ハンドラへのパラメータがこのエラーコードになる。

※ 拡張SVCハンドラからこのエラーコードを返した場合でも、このエラーコードに対するシステムコール例外は発生しない。

ニモニック	ercd	説明
E_CEX	-H'303 (-(EC_CEX≪8)-3)	CPU例外発生

●システムエラー例外クラス (EC_SYS=4)　　errno: 5〜8

ニモニック	ercd	説明
E_SYS	-H'405	システムエラー
	(-(EC_SYS≪8)-5)	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。 ITRON2では、このエラーが発生した場合に限り、idef_sex で定義されたタスク独立部に対する例外ハンドラが起動される。 ITRON1の TE_SYSに対応する。

●メモリ不足エラー例外クラス (EC_NOMEM=5)　　errno: 9〜16

ニモニック	ercd	説明
E_NOSMEM	-H'509	システムメモリ不足
	(-(EC_NOMEM≪8)-9)	オブジェクト管理ブロック確保時などの一般的なメモリ不足 (no system memory) ITRON1の TE_MEM に対応する。
E_NOMEM	-H'50a	メモリ不足
	(-(EC_NOMEM≪8)-10)	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獲得時のメモリ不足 (no memory) ITRON1の TE_MEM, (TE_MPLSZ) に対応する。

●予約機能エラー例外クラス (EC_RSV=6)　　errno: 17〜32

ニモニック	ercd	説明
E_NOSPT	-H'611	未サポート機能
	(-(EC_RSV≪8)-17)	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_RSMD, E_RSATR または E_NOSPTのエラーを発生する。

E_RSMD, E_RSATR に該当しない場合には、E_NOSPT のエラーとなる。
μITRONの場合、システムコールの一部の機能のみをインプリメントすることも許しているため、このエラーを発生する可能性がある。μITRONやデバッグ関連機能のようにインプリメント依存部分の多いシステムコールにおいて、仕様書で定義できないようなエラーを検出した場合に、このエラーを返す。
ITRON2の基本機能や拡張機能では、E_RSFN, E_RSMD, E_RSATR などのエラーコードを使えば十分なので、原則としてE_NOSPT のエラーを返す必要はない。

E_INOSPT -H'612 (-(EC_RSV≪8)-18)
ITRON/FILEでの未サポート機能
ITRON1の E_INOSPT に対応する。

E_TNOSPT -H'613 (-(EC_RSV≪8)-19)
タイマ関係の未サポート機能
ITRON1の TE_NOTMR に対応する。

E_RSFN -H'614 (-(EC_RSV≪8)-20)
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。
なお、このエラーはシステムコール全体がサポートされていない場合に発生するエラーである。システムコールの一部の機能のみがサポートされていない場合には、E_RSMD, E_RSATR,E_NOSPT などのエラーを発生する。
ITRON1の TE_UDF に対応する。

E_RSSFN (-(EC_RSV≪8)-21)	-H'615	予約サブ機能コード/予約デバイス番号
E_RSMD (-(EC_RSV≪8)-22)	-H'616	予約モード、予約オプション ITRON1の TE_OPT に対応する。
E_RSID (-(EC_RSV≪8)-23)	-H'617	予約ID番号 0～(-4)のID番号を指定した場合などに発生する。 ITRON1の TE_IDOVR に対応する。
E_RSATR (-(EC_RSV≪8)-24)	-H'618	予約属性 未定義のシステム用オブジェクト属性を指定した場合などに発生する。
E_XNOSPT (-(EC_RSV≪8)-25)	-H'619	このコンテキストでの実行はサポートしていない。 インプリメントの都合により、タスク独立部（割込みハンドラ等）から発行できなくなっているシステムコールを発行しようとした場合に、このエラーが返る。各システムコールにおいて、E_XNOSPT が発生するかどうかはインプリメント依存である。 なお、自タスクを待ち状態にするようなシステムコールがタスク独立部から発行できないのは、インプリメントの都合ではなく意味的な問題なので、このエラーではなく E_CTX が返る。

●パラメータエラー例外クラス (EC_PAR=7)　　errno: 33～48

※ ほぼ静的にチェック可能なエラーである。

ニモニック	ercd	説明
E_PAR (-(EC_PAR≪8)-33)	-H'721	一般的なパラメータエラー ITRON1の TE_PAR、(TE_MPLSZ)に対応する。

E_ILADR -H'722 (-(EC_PAR≪8)-34)	不正アドレス スタートアドレスが奇数の場合など ITRON1の TE_MA,TE_SA,TE_PA に対応する。
E_IDOVR -H'723 (-(EC_PAR≪8)-35)	ID範囲外 オブジェクトを指定するIDが、システムで利用できる最大値を越えた。 なお、E_IDOVR や E_RSID 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ため、割込みベクトル番号やハンドラ番号に関してはこれらのエラーが発生することはない。割込みベクトル番号やハンドラ番号に関して、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のエラーを返す。 ITRON1の TE_IDOVR（オブジェクト生成時）に対応する。
E_SZOVR -H'724 (-(EC_PAR≪8)-36)	パラメータのサイズが制限を越えた。
E_PPRI -H'725 (-(EC_PAR≪8)-37)	不正プロセス優先度
E_TPRI -H'726 (-(EC_PAR≪8)-38)	不正タスク優先度 ITRON1の TE_TPRI に対応する。
E_ILTIME -H'727 (-(EC_PAR≪8)-39)	不正時間指定 ltime, utime, cytime, tmout の指定におけるエラー ITRON1の TE_PAR に対応する。
E_ILFN -H'728 (-(EC_PAR≪8)-40)	不正機能コード番号 拡張SVC定義などの際に発生する ITRON1の TE_SVC に対応する。

E_ILMSG -H'729
(-(EC_PAR≪8)-41)
不正メッセージ形式
メッセージの msgtype が 0 でない場合などに発生する。

E_VECN -H'72a
(-(EC_PAR≪8)-42)
不正ベクトル番号
ITRON1の TE_VECN に対応する。

E_IMS -H'72b
(-(EC_PAR≪8)-43)
不正IMASK
ITRON1の TE_IPRI に対応する。

E_ILSFN -H'72c
(-(EC_PAR≪8)-44)
不正サブ機能コード番号
拡張SVC定義の際のサブ機能コードが不正であった場合に発生する。
なお、E_ILSFN は定義時に、E_RSSFN は実行時に発生するエラーである。

●不正オブジェクト指定例外クラス (EC_OBJ=8)　　errno: 49〜64

ニモニック	ercd	説明
E_SELF	-H'831 (-(EC_OBJ≪8)-49)	自タスク、自プロセスの指定 ITRON1の TE_SELF に対応する。
E_NOPRC	-H'832 (-(EC_OBJ≪8)-50)	プロセスがまだ存在していない。
E_EXS	-H'833 (-(EC_OBJ≪8)-51)	オブジェクトが既に存在している ITRON1の TE_EXSに対応する。
E_NOEXS	-H'834 (-(EC_OBJ≪8)-52)	オブジェクトが存在していない sdef_sexで指定した拡張SVCが未定義の場合なども含まれる。 なお、オブジェクトを操作する一般のシステムコールにおいて対象オブジェクトが存在していない時は、E_NOEXS のエラーになる場合と E_IDOVR のエラーになる場合とがある。E_IDOVRと E_NOEXS の違いは、E_IDOVR が静的に検出可能なエラーであるのに対して、E_NOEXS は動的な検出を必要とするエ

ラーであるという点である。ただし、利用可能な ID 番号の範囲にインプリメント依存の点があるので、両者の厳密な区別もインプリメント依存である。
ITRON1の TE_NOEXS に対応する。

E_DMT -H'835 (-(EC_OBJ≪8)-53)
タスクがDORMANTである。
ITRON1の TE_DMT に対応する。

E_NODMT -H'836 (-(EC_OBJ≪8)-54)
タスクがDORMANTでない。
ITRON1の TE_NODMT に対応する。

E_NOCYC -H'837 (-(EC_OBJ≪8)-55)
タスクに対してcyc_wupが発行されていない。
ITRON1の TE_NOCYC に対応する。

E_NOSUS -H'838 (-(EC_OBJ≪8)-56)
タスクがSUSPENDでない。
ITRON1の TE_NOSUS に対応する。

E_ILSEM -H'839 (-(EC_OBJ≪8)-57)
セマフォに対する不正操作

E_ILAKEY -H'83a (-(EC_OBJ≪8)-58)
セマフォアクセスキーが不正

E_ILBLK -H'83b (-(EC_OBJ≪8)-59)
不正メモリブロックの返却、操作
ITRON1の TE_ILBLK に対応する。

E_NORDV -H'83c (-(EC_OBJ≪8)-60)
現在ランデブ中のタスクではない

E_NOWAI -H'83e (-(EC_OBJ≪8)-62)
タスクが待ち状態でない。

E_OBJ -H'83f (-(EC_OBJ≪8)-63)
オブジェクト状態に関するその他のエラー

●アクセス権違反例外クラス (EC_ACV=9) **errno: 65〜68**

※ Access Violation

※ オブジェクト、メモリのアクセス違反を含む。ファイルのアクセス違反は除く。

※ μITRONでは発生しない。

ニモニック	ercd	説明
E_MACV	-H'941 (-(EC_ACV≪8)-65)	メモリアクセス権違反 バスアクセスエラー等も含む。 ITRON1の TE_ACCES に対応する。
E_OACV	-H'942 (-(EC_ACV≪8)-66)	オブジェクトやハンドラのアクセス権違反 ユーザタスク（拡張SVCハンドラ内を除く）がシステムオブジェクトを操作した場合など
E_ULEVV	-H'943 (-(EC_ACV≪8)-67)	ユーザレベル違反

●コンテキストエラー例外クラス (EC_CTX=10)　　errno: 69〜72

ニモニック	ercd	説明
E_CTX	-H'a45 (-(EC_CTX≪8)-69)	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 E_CTX のエラーが発生する。 ITRON2の場合、E_CTX の発生条件にインプリメント依存性はなく、仕様書の各システムコールの説明の項に E_CTX の記述が無いシステムコールでは、このエラーが発生することはない。一方、μITRONの場合はエラーコードの返し方にインプリメント依存性がある。そのため、意味的な間違いではなく、インプリメントの都合でタスク独立部から発行できなくなっているシステムコールを発行しようとした場合にも、E_XNOSPT のエラーの代わりに E_CTX のエラーを返す場合がある。

ITRON1の TE_CTX に対応する。

●実行時エラー例外クラス (EC_EXEC=11)　　errno: 73〜80

ニモニック	ercd	説明
E_QOVR	-H'b49	キューイングのオーバーフロー
	(-(EC_EXEC≪8)-73)	ITRON1の TE_QOVR, TE_MBOVR に対応する。
E_OOVR	-H'b4a	オブジェクト数のオーバーフロー
	(-(EC_EXEC≪8)-74)	

●実行終了時エラー例外クラス (EC_EXCMP=12)　　errno: 81〜84

※ Execution Completed

ニモニック	ercd	説明
E_DLT	-H'c51	待ちオブジェクトが削除された。
	(-(EC_EXCMP≪8)-81)	ITRON1の TE_DLT に対応する。
E_INIHDR	-H'c52	初期化ハンドラエラー
	(-(EC_EXCMP≪8)-82)	

●待ち状態強制解除例外クラス (EC_RLWAI=13)　　errno: 85〜88

※ Release Wait

ニモニック	ercd	説明
E_TMOUT	-H'd55	タイムアウト
	(-(EC_RLWAI≪8)-85)	ITRON1の TE_TMOUT に対応する。
E_RLWAI	-H'd56	待ち状態強制解除
	(-(EC_RLWAI≪8)-86)	

●処理実行結果を示す例外クラス (EC_RSLT=14)　　errno: 89〜96

ニモニック	ercd	説明
E_PLFAIL	-H'e59	ポーリング失敗（メッセージ/セマフォが
	(-(EC_RSLT≪8)-89)	無い）(Polling Fail)

E_AROVR	-H'e5a (-(EC_EXEC≪8)-90)	用意した領域のサイズが小さすぎる (Area Over)
E_MPURG	-H'e5b (-(EC_RSLT≪8)-91)	メモリブロックが削除された
E_MRELOC	-H'e5c (-(EC_RSLT≪8)-92)	メモリブロックが再配置された

第四部　第二章

μITRONレファレンス

システムコール一覧（μITRON）

'=' の記号のついているものは、リターンパラメータである。

■ タスク管理機能

レベル	システムコール	パラメータ/リターンパラメータ（=印）	機能説明
2	sta_tsk	tskid	タスクを起動する
2	ext_tsk		自タスクを正常終了する
3	ter_tsk	tskid	他タスクを強制的に異常終了させる
3	chg_pri	tskid tskpri	タスク優先度を変更する
#3	ichg_pri	tskid tskpri	タスク優先度を変更する（タスク独立部専用）
3	rot_rdq	tskpri	タスクのレディキューを回転する
#3	irot_rdq	tskpri	タスクのレディキューを回転する（タスク独立部専用）
5	rel_wai	tskid	タスクの待ち状態を強制解除する
#5	irel_wai	tskid	タスクの待ち状態を強制解除する（タスク独立部専用）
3	get_tid	tskid=	自タスクのIDを得る
3	tsk_sts	tskid tskpri= tskstat=	タスクの状態を見る

■ タスク付属同期機能

レベル	システムコール	パラメータ/リターンパラメータ（=印）	機能説明
3	sus_tsk	tskid	タスクを強制待ち状態へ移行する
#3	isus_tsk	tskid	タスクを強制待ち状態へ移行する（タスク独立部専用）
3	rsm_tsk	tskid	強制待ち状態のタスクを再開する
#3	irsm_tsk	tskid	強制待ち状態のタスクを再開する（タスク独立部専用）

4	frsm_tsk	tskid		強制待ち状態のタスクを強制再開する
1	slp_tsk			タスクを待ち状態へ移行する
3	wai_tsk	tmout		タスクを一定時間待ち状態に移行する
1	wup_tsk	tskid		待ち状態のタスクを起床する
#1	iwup_tsk	tskid		待ち状態のタスクを起床する（タスク独立部専用）
3	can_wup	tskid	wupcnt=	タスクの起床要求を無効にする

■ 同期・通信機能

レベル	システムコール	パラメータ/リターンパラメータ (=印)			機能説明
3A	set_flg	flgid	setptn		1ワードイベントフラグをセットする
#3A	iset_flg	flgid	setptn		1ワードイベントフラグをセットする（タスク独立部専用）
3A	clr_flg	flgid	clrptn		1ワードイベントフラグをクリアする
3A	wai_flg	flgid	waiptn	flgptn= wfmode	1ワードイベントフラグを待つ
3A	pol_flg	flgid	waiptn	flgptn= wfmode	1ワードイベントフラグを得る
4A	flg_sts	flgid	wtskid=	flgptn=	1ワードイベントフラグ状態を参照する
3B	set_flg	flgid			1ビットイベントフラグをセットする
#3B	iset_flg	flgid			1ビットイベントフラグをセットする（タスク独立部専用）
3B	clr_flg	flgid			1ビットイベントフラグをクリアする
3B	wai_flg	flgid			1ビットイベントフラグを待つ（クリア無）
3B	cwai_flg	flgid			1ビットイベントフラグを待つ（クリア有）
3B	pol_flg	flgid			1ビットイベントフラグを得る（クリア無）
3B	cpol_flg	flgid			1ビットイベントフラグを得る（クリア有）

4B	flg_sts	flgid	wtskid= flgptn=	1ビットイベントフラグ状態を参照する
1	sig_sem	semid		セマフォに対する信号操作（V命令）
#1	isig_sem	semid		セマフォに対する信号操作（タスク独立部専用）
1	wai_sem	semid		セマフォに対する待ち操作（P命令）
1	preq_sem	semid		セマフォ資源を得る
4	sem_sts	semid	wtskid= semcnt=	セマフォ状態を参照する
2	snd_msg	mbxid	pk_msg	メイルボックスへ送信する
#2	isnd_msg	mbxid	pk_msg	メイルボックスへ送信する（タスク独立部専用）
2	rcv_msg	mbxid	pk_msg=	メイルボックスからの受信を待つ
2	prcv_msg	mbxid	pk_msg=	メイルボックスから受信する
4	mbx_sts	mbxid	wtskid= pk_msg=	メイルボックス状態を参照する
5	snd_tmb	tskid	pk_msg	タスク付属メイルボックスへ送信する
#5	isnd_tmb	tskid	pk_msg	タスク付属メイルボックスへ送信する（タスク独立部専用）
5	rcv_tmb	pk_msg=		タスク付属メイルボックスからの受信を待つ
5	prcv_tmb	pk_msg=		タスク付属メイルボックスから受信する
5	tmb_sts	tskid	tmbwait= pk_msg=	タスク付属メイルボックスの状態を参照する

■ 割込み管理機能

レベル	システムコール	パラメータ/リターンパラメータ(=印)		機能説明
#1	def_int	intno	inthdr	割込みハンドラを定義する
#1	ret_int			割込みハンドラから復帰する
3	ret_wup	tskid		割込処理復帰とタスク起床を行う
#5	def_rst	tskid	rstadr	タスクリスタートアドレスを定義する
5	ret_rst	tskid		割込処理復帰とタスクリスタートを行う
2A	dis_int	intno		割込みを禁止する
2A	ena_int	intno		割込みを許可する
2B	chg_iXX	iXXXX		割込みマスク（レベル,優先度）を変更する
3B	iXX_sts	iXXXX=		割込みマスク（レベル,優先度）を参照する

■ メモリプール管理機能

レベル	システムコール	パラメータ/リターンパラメータ(=印)		機能説明
4	get_blk	mplid	blk=	固定長メモリブロックの獲得待ちを行う
3	pget_blk	mplid	blk=	固定長メモリブロックを獲得する
3	rel_blk	mplid	blk	固定長メモリブロックを返却する
4	mpl_sts	mplid	wtskid= frbcnt=	メモリプールの状態を参照する

■ 時間管理・タイマハンドラ機能

レベル	システムコール	パラメータ/リターンパラメータ(=印)				機能説明
2	set_tim	time				システムクロックを設定する
2	get_tim	time=				システムクロックの値を読み出す
#4	def_cyc	cyhno	cychdr	cyhact	cytime	周期起動ハンドラを定義する
4	act_cyc	cyhno	cyhact			周期起動ハンドラの活性制御を行なう

4	cyh_sts	cyhno	cyhact= lftime=	周期起動ハンドラの状態を参照する
#4	def_alm	alhno	almhdr time tmmode	アラームハンドラを定義する
4	alh_sts	alhno	lftime=	アラームハンドラの状態を参照する
#4	ret_tmr			タイマハンドラから復帰する

■ システム管理機能

レベル	システムコール	パラメータ/リターンパラメータ (=印)	機能説明
1	get_ver	pk_ver	ITRON, μITRONのバージョン番号を得る

※「レベル」の項に示されている数字は、インプリメントの必要性の度合いを示すものである。数字の小さい方が必要性の高いことを表わす。

※「レベル」の項の数字に A, B の記号が付いているシステムコールは、インプリメンタの側で、A, B のどちらかの仕様を選択してインプリメントすることを示す。なお、可能であれば、A, B 両方の仕様をインプリメントしても構わない。

※「レベル」の項の数字に # の記号が付いているシステムコールは、他の方法で代替できる可能性のあるシステムコールを示す。例えば、def_int には # が付いているが、これは、割込みハンドラ定義の機能が他の方法（例えばシステム起動時の静的な定義）で代替できれば、そのシステムコールが不要であることを示す。

※割込みマスク（レベル,優先度）の変更や参照を行うシステムコール chg_iXX, iXX_sts では、プロセッサのアーキテクチャに合わせて 'XX' の部分の名称を決める。

※オブジェクトの状態を見るシステムコール XXX_sts のリターンパラメータは、インプリメントによってかなり違いがある。この一覧表に書かれているインタフェースは、1つの例である。

※必要があれば、この一覧表に無いシステムコールを追加して設けることも可能である。ただし、追加した機能がITRON2でサポートする機能に含まれるものであれば、ITRON2のシステムコール名やシステムコールインタフェースと矛盾の無い仕様にしなければならない。

データタイプ (μITRON)

```
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 long            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 long   UW;     /* 符号無し32ビット整数 */
typedef long            VW;     /*データタイプが一定しないもの
                                (32ビットサイズ) */
typedef short           VH;     /*データタイプが一定しないもの
                                (16ビットサイズ) */
typedef char            VB;     /*データタイプが一定しないもの
                                (8ビットサイズ) */
typedef void            *VP;    /*データタイプが一定しないものへの
                                  ポインタ */
typedef void            (*FP)( ); /* プログラムのスタートアドレス一般 */
```

※ VB, VH, VW と B, H, W との違いは、前者がビット数のみがわかっており、データタイプの中味がわからないものを表すのに対して、後者は整数を表しているという点である。例えば、メッセージの中身やメモリブロックはバイトの配列であるが、中に入れるデータの実際のデータタイプは場合によって異なっているので、VB のデータタイプを使っている。また、パケットの中でリザーブ領域を確保する場合は、データのサイズはわかっているが整数であるとは言えないため、VH, VW のデータタイプを使っている。

```
/* パラメータの意味を明確にし、また対象プロセッサのビット数に依しない記述を行うため、出現頻度の高いデータタイプや特殊な意味を持つデータタイプとして、以下のような名称を使用する */
BOOL        /* ブール値、FALSE (0 - 偽) または TRUE (1 - 真) */
ID          /* オブジェクトのID (XXXid) */
```

```
HNO         /* ハンドラ番号 */
ER          /* エラーコード - 一般には符号付きである */
TMO         /* タイムアウト */
TIME        /* 時間指定 - 実際には上位と下位に分けて扱うことが多い */
FN          /* 機能コード */
TPRI        /* タスク優先度 */
T_MSG       /* メッセージパケットの構造体 */
INT         /* 符号付き整数（プロセッサのビット幅）*/
UINT        /* 符号無し整数（プロセッサのビット幅）*/
```

※ これらのデータタイプのビット数、値の範囲、符号の有無などは、対象プロセッサおよびインプリメントに依存する。

※ μITRONでは対象プロセッサのビット数を固定していないため、言語Cインタフェースの説明の中では、ビット数を明確にした W, H, UW, UH のような記述はあまり使用せず、INT, UINT を使用している場合が多い。ただし、μITRONの個々のインプリメントでは対象プロセッサのビット数がわかっているので、μITRONの製品のマニュアル等において、INT, UINT の代りに W, H, UW, UH 等を使用することは構わない。

※ 例えば、16ビットのプロセッサを対象とした場合には、INT と H、および UINT と UH が同じ意味を表わす。つまり、以下のように書くことができる。

```
typedef H       INT;    /* 16ビットプロセッサの場合 */
typedef UH      UINT;   /* 16ビットプロセッサの場合 */
```

※ 上記のデータタイプと実際のビット数との対応については、インプリメント毎に明らかにしておく必要がある。

【記述例】

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

言語Cインタフェース(μITRON)

●タスク管理機能

```
ER ercd = sta_tsk ( ID tskid );
void       ext_tsk ( );
ER ercd = ter_tsk ( ID tskid );
ER ercd = chg_pri ( ID tskid, TPRI tskpri );
ER ercd = ichg_pri ( ID tskid, TPRI tskpri );
ER ercd = rot_rdq ( TPRI tskpri );
ER ercd = irot_rdq ( TPRI tskpri );
ER ercd = rel_wai ( ID tskid );
ER ercd = irel_wai ( ID tskid );
ER ercd = get_tid ( ID *p_tskid );
ER ercd = tsk_sts ( UINT *p_tskstat, TPRI *p_tskpri, ID tskid );
```

※ tsk_sts のインタフェースはパケットを使用する場合もある

●タスク付属同期機能

```
ER ercd = sus_tsk ( ID tskid );
ER ercd = isus_tsk ( ID tskid );
ER ercd = rsm_tsk ( ID tskid );
ER ercd = irsm_tsk ( ID tskid );
ER ercd = frsm_tsk ( ID tskid );
ER ercd = slp_tsk ( );
ER ercd = wai_tsk ( TMO tmout );
ER ercd = wup_tsk ( ID tskid );
ER ercd = iwup_tsk ( ID tskid );
ER ercd = can_wup ( INT *p_wupcnt, ID tskid );
```

●同期・通信機能

【1ワードイベントフラグの場合】

```
ER ercd = set_flg ( ID flgid, UINT setptn );
ER ercd = iset_flg ( ID flgid, UINT setptn );
ER ercd = clr_flg ( ID flgid, UINT clrptn );
ER ercd = wai_flg ( UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode );
ER ercd = pol_flg ( UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode );
ER ercd = flg_sts ( ID *p_wtskid, UINT *p_flgptn, ID flgid );
```

※ flg_sts のインタフェースはパケットを使用する場合もある

【1ビットイベントフラグの場合】

```
ER ercd = set_flg ( ID flgid );
ER ercd = iset_flg ( ID flgid );
ER ercd = clr_flg ( ID flgid );
ER ercd = wai_flg ( ID flgid );
ER ercd = cwai_flg ( ID flgid );
ER ercd = pol_flg ( ID flgid );
ER ercd = cpol_flg ( ID flgid );
ER ercd = flg_sts ( ID *p_wtskid, BOOL *p_flgptn, ID flgid );
```

※ flg_sts のインタフェースはパケットを使用する場合もある

```
ER ercd = sig_sem ( ID semid );
ER ercd = isig_sem ( ID semid );
ER ercd = wai_sem ( ID semid );
ER ercd = preq_sem ( ID semid );
ER ercd = sem_sts ( ID *p_wtskid, INT *p_semcnt, ID semid );
```

※ sem_sts のインタフェースはパケットを使用する場合もある

```
ER ercd = snd_msg ( ID mbxid, T_MSG *pk_msg);
ER ercd = isnd_msg ( ID mbxid, T_MSG *pk_msg);
ER ercd = rcv_msg ( T_MSG **ppk_msg, ID mbxid );
ER ercd = prcv_msg ( T_MSG **ppk_msg, ID mbxid );
ER ercd = mbx_sts ( ID *p_wtskid, T_MSG **ppk_msg, ID mbxid );
```

※ mbx_sts のインタフェースはパケットを使用する場合もある

```
ER ercd = snd_tmb ( ID tskid, T_MSG *pk_msg);
ER ercd = isnd_tmb ( ID tskid, T_MSG *pk_msg);
ER ercd = rcv_tmb ( T_MSG **ppk_msg );
ER ercd = prcv_tmb ( T_MSG **ppk_msg );
ER ercd = tmb_sts ( BOOL *p_tmbwait, T_MSG **ppk_msg, ID tskid );
```

※ tmb_sts のインタフェースはパケットを使用する場合もある

●割込み管理機能

```
ER ercd = def_int ( UINT intno, FP inthdr );
void       ret_int ( );
void       ret_wup ( ID tskid );
ER ercd = def_rst ( ID tskid, FP rstadr );
void       ret_rst ( ID tskid );
ER ercd = dis_int ( UINT intno );
ER ercd = ena_int ( UINT intno );
ER ercd = chg_iXX ( UINT iXXXX );
ER ercd = iXX_sts ( UINT *p_iXXXX );
```

※ intno, iXXXX のデータタイプは UINT にならない場合もある

●メモリプール管理機能

```
ER ercd = get_blk ( VP *p_blk, ID mplid );
ER ercd = pget_blk ( VP *p_blk, ID mplid );
ER ercd = rel_blk ( ID mplid, VP blk );
ER ercd = mpl_sts ( ID *p_wtskid, INT *p_frbcnt, ID mplid );
```

※ mpl_sts のインタフェースはパケットを使用する場合もある

●時間管理・タイマハンドラ機能

```
ER ercd = set_tim ( TIME time );
ER ercd = get_tim ( TIME *p_time );
```

※ set_tim, get_tim のインタフェースはパケットを使用する場合もある

```
ER ercd = def_cyc ( HNO cyhno, FP cychdr, UINT cyhact, TIME cytime );
ER ercd = act_cyc ( HNO cyhno, UINT cyhact );
```

```
ER ercd = cyh_sts ( UINT *p_cyhact, TIME *p_lftime, HNO cyhno );
ER ercd = def_alm ( HNO alhno, FP almhdr, TIME time, UINT tmmode );
ER ercd = alh_sts ( TIME *p_lftime, HNO alhno );
```

※ def_cyc, cyh_sts, def_alm, alh_sts のインタフェースはパケットを使用する場合もある

```
void        ret_tmr ( );
```

●システム管理機能

```
ER ercd = get_ver ( T_VER *pk_ver );
```

共通定数と構造体のパケット形式（μITRON）

```
/*----------------------------------------------------------------------------*/
   （共通に使用する定数）
全体:
   NADR        (-1)      /* アドレスやポインタ値が無効 */
   TRUE        1         /* 真 */
   FALSE       0         /* 偽 */
tskid, wtskid:
   TSK_SELF    0         /* 自タスク指定 */
   FALSE       0         /* タスク独立部、あるいは待ちタスク無しを
                            表わす（リターンパラメータのみ）*/
tskpri:
   TPRI_INI    0         /* タスク起動時の初期優先度を指定 (chg_pri) */
   TPRI_RUN    0         /* その時実行中の最高優先度を指定 (rot_rdq) */
tmmode:
   TTM_ABS     0         /* 絶対時刻での指定 */
   TTM_REL     1         /* 相対時刻での指定 */
tskstat:
   TTS_RUN     H'00000001        /* RUN */
   TTS_RDY     H'00000002        /* READY */
   TTS_WAI     H'00000004        /* WAIT */
   TTS_SUS     H'00000008        /* SUSPEND */
   TTS_WAS     H'0000000c        /* WAIT-SUSPEND */
   TTS_DMT     H'00000010        /* DORMANT */
```

※ tskstat 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、WAIT または READY であることを判定する）場合に便利である。

```
cyhact:
    TCY_OFF     0               /* 周期起動ハンドラが起動されない */
    TCY_ON      1               /* 周期起動ハンドラが起動される */
    TCY_INI     2               /* 周期のカウントが初期化される */
/*----------------------------------------------------------------------------------------*/
    typedef struct t_msg {
                ...
    /* インプリメントによっては、ここにOSの予約領域が入る */
                ...
            VB      msgcont[ ];
    } T_MSG;
/*----------------------------------------------------------------------------------------*/
    typedef struct t_ver {
            UH      maker;          /* メーカー */
            UH      id;             /* 形式番号 */
            UH      spver;          /* 仕様書バージョン */
            UH      prver;          /* 製品バージョン */
            UH      prno[4];        /* 製品管理情報 */
            UH      cpu;            /* CPU情報 */
            UH      var;            /* バリエーション記述子 */
    } T_VER;
/*----------------------------------------------------------------------------------------*/
```

第四部　第三章

ITRON2レファレンス

システムコール一覧 (ITRON2)

'=' の記号のついているものは、リターンパラメータである。

ITRON2基本機能

■ タスク管理機能

システムコール	パラメータ/リターンパラメータ (=印)	機能説明
cre_tsk	tskid tskatr stadr itskpri stksz	タスクを生成する
sta_tsk	tskid stacd	タスクを起動する
del_tsk	tskid	タスクを削除する
ext_tsk		自タスクを正常終了する
exd_tsk		自タスクを正常終了後、削除する
abo_tsk	exccd	自タスクを異常終了させる
ter_tsk	tskid exccd	他タスクを強制的に異常終了させる
ras_ter	tskid exccd	他タスクに対して終了要求を出す
chg_pri	tskid tskpri	タスク優先度を変更する
rot_rdq	tskpri	タスクのレディキューを回転する
rel_wai	tskid	タスクの待ち状態を強制解除する
get_tid	tskid=	自タスクのIDを得る
tsk_sts	tskid pk_tsks	タスクの状態を見る
hdr_sts	tskid ar_hdrs hdstart hdcnt	ハンドラ実行環境を参照する

■ タスク付属同期機能

システムコール	パラメータ/リターンパラメータ (=印)	機能説明
sus_tsk	tskid	タスクを強制待ち状態へ移行する

rsm_tsk	tskid	強制待ち状態のタスクを再開する
frsm_tsk	tskid	強制待ち状態のタスクを強制再開する
slp_tsk		タスクを待ち状態へ移行する
wai_tsk	tmout	タスクを一定時間待ち状態に移行する
wup_tsk	tskid	待ち状態のタスクを起床する
can_wup	tskid wupcnt=	タスクの起床要求を無効にする

■ 同期・通信機能

システムコール	パラメータ/リターンパラメータ（=印）	機能説明
cre_flg	flgid flgatr	イベントフラグを生成する
del_flg	flgid	イベントフラグを削除する
set_flg	flgid setptn	イベントフラグをセットする
clr_flg	flgid clrptn	イベントフラグをクリアする
wai_flg	flgid waiptn flgptn= tmout wfmode	イベントフラグを待つ
flg_sts	flgid pk_flgs	イベントフラグ状態を参照する
cre_sem	semid sematr isemcnt	セマフォを生成する
del_sem	semid	セマフォを削除する
sig_sem	semid rcnt	セマフォに対する信号操作（V命令）
wai_sem	semid rcnt tmout	セマフォに対する待ち操作（P命令）
sem_sts	semid pk_sems	セマフォ状態を参照する
cre_mbx	mbxid mbxatr	メイルボックスを生成する
del_mbx	mbxid	メイルボックスを削除する
snd_msg	mbxid pk_msg	メイルボックスへ送信する
rcv_msg	mbxid pk_msg= tmout	メイルボックスから受信する
mbx_sts	mbxid pk_mbxs	メイルボックス状態を参照する

■ 割込み管理機能

システムコール	パラメータ/リターンパラメータ（=印）	機能説明
def_int	intvec inhatr inthdr imask	割込みハンドラを定義する
ret_int		割込みハンドラから復帰する
chg_ims	imask	割込みマスクを変更する

■ 例外管理機能

システムコール	パラメータ/リターンパラメータ（=印）	機能説明
def_ext	tskid exhatr exthdr	タスク用終了ハンドラの定義
def_cex	tskid exhatr cexhdr	タスク用CPU例外ハンドラの定義
def_sex	tskid exhatr sexhdr	タスク用システムコール例外ハンドラの定義
ret_exc		例外ハンドラから復帰
end_exc		例外ハンドラの終了
clr_ems	tskid clrptn	例外処理マスククリア
set_ems	tskid setptn	例外処理マスクセット
idef_ext	exhatr exthdr	タスク独立部用終了ハンドラの定義
idef_cex	exhatr cexhdr	タスク独立部用CPU例外ハンドラの定義
idef_sex	exhatr sexhdr	タスク独立部用システムコール例外ハンドラの定義

■ メモリプール管理機能

システムコール	パラメータ/リターンパラメータ（=印）	機能説明
cre_mpl	mplid mplatr mplsz blksz	メモリプールの生成
del_mpl	mplid	メモリプールを削除する
get_blk	mplid bcnt blk= tmout	共有メモリブロックを獲得する
rel_blk	mplid blk	共有メモリブロックを返却する
mpl_sts	mplid pk_mpls	メモリプールの状態を参照する
blk_sts	mplid= blk bcnt=	共有メモリブロックの状態参照

■ 時間管理機能

システムコール	パラメータ/リターンパラメータ（=印）	機能説明
set_tim	utime ltime	システムクロックを設定する
get_tim	utime= ltime=	システムクロックの値を読み出す
dly_tsk	dtime	タスクの遅延を行う
cyc_wup	tskid rptcnt cytime utime ltime tmmode	タスクを周期起床する
can_cyc	tskid	タスクに出された周期起床要求を解除する

■ システム管理機能

システムコール	パラメータ/リターンパラメータ（=印）	機能説明
def_svc	s_fncd svhatr svchdr	拡張SVCハンドラを定義する
ret_svc	s_ercd	拡張SVCハンドラから復帰する
get_ver	pk_ver	ITRONのバージョン番号を得る
psw_sts	psw=	プロセッサ状態語を参照する

ITRON2拡張機能

■ 拡張同期・通信機能

システムコール	パラメータ/リターンパラメータ（=印）	機能説明
cre_mbf	mbfid mbfatr bufsz maxbmsz	メッセージバッファを生成する
del_mbf	mbfid	メッセージバッファを削除する
snd_mbf	mbfid bmsgsz pk_bmsg	メッセージバッファへ送信する
rcv_mbf	mbfid bmsgsz= tmout pk_bmsg	メッセージバッファから受信する
mbf_sts	mbfid pk_mbfs	メッセージバッファ状態を参照する
cre_por	porid poratr	ランデブ用のポートを生成する
del_por	porid	ランデブ用のポートを削除する

cal_por	porid calptn pk_rmsg maxrmsz tmout	ポートに対するランデブの呼び出し
acp_por	porid acpptn ctskid = pk_rmsg maxrmsz tmout	ポートに対するランデブの受け付け
fwd_por	porid ctskid calptn pk_rmsg	ポートに対するランデブの回送
rpl_por	ctskid pk_rmsg	ポートに対するランデブの返答
por_sts	porid pk_pors	ポート状態を参照する
rdv_sts	porid ar_rdvs rvstart rvcnt	ランデブ状態を参照する

■ 強制例外機能

システムコール	パラメータ/リターンパラメータ（=印）	機能説明
def_fex	tskid exhatr fexhdr	タスク用強制例外ハンドラの定義
ras_fex	tskid exccd	他タスクに対する強制例外の発生

■ ローカルメモリプール管理機能

システムコール	パラメータ/リターンパラメータ（=印）	機能説明
cre_lmp	lmpid lmpatr lmpsz blksz	ローカルメモリプールの生成
del_lmp	lmpid	ローカルメモリプールを削除する
get_lbl	lmpid bcnt blk= tmout	ローカルメモリブロックを獲得する
rel_lbl	lmpid blk	ローカルメモリブロックを返却する
lmp_sts	lmpid pk_lmps	ローカルメモリプールの状態を参照する
lbl_sts	lmpid= blk bcnt=	ローカルメモリブロックの状態参照

■ 資源管理サポート機能

システムコール	パラメータ/リターンパラメータ（=印）	機能説明
usig_sem	semid rcnt p_rcnt	セマフォに対する信号操作（メモリ更新有）

uwai_sem	semid rcnt tmout p_rcnt	セマフォに対する待ち操作（メモリ更新有）
usnd_msg	mbxid ppk_msg	メイルボックスへ送信する（メモリ更新有）
urcv_msg	mbxid tmout ppk_msg	メイルボックスから受信する（メモリ更新有）
uget_blk	mplid bcnt tmout p_blk	共有メモリブロックを獲得する（メモリ更新有）
uget_lbl	lmpid bcnt tmout p_blk	ローカルメモリブロックを獲得する（メモリ更新有）
urel_blk	mplid p_blk	共有メモリブロックを返却する（メモリ更新有）
urel_lbl	lmpid p_blk	ローカルメモリブロックを返却する（メモリ更新有）

■ タイマハンドラ機能

システムコール	パラメータ/リターンパラメータ（=印）	機能説明
def_cyc	cyhno cyhatr cychdr cyhact cytime	周期起動ハンドラを定義する
act_cyc	cyhno cyhact	周期起動ハンドラの活性制御を行なう
cyh_sts	cyhno pk_cyhs	周期起動ハンドラの状態を参照する
def_alm	alhno alhatr almhdr utime ltime tmmode	アラームハンドラを定義する
alh_sts	alhno pk_alhs	アラームハンドラの状態を参照する
ret_tmr		タイマハンドラから復帰する

ITRON2システム操作機能

■ 拡張SVCサポート機能

システムコール	パラメータ/リターンパラメータ（=印）			機能説明
sdef_ext	s_fncd	exhatr	exthdr	拡張SVC用終了ハンドラの定義
sdef_cex	s_fncd	exhatr	cexhdr	拡張SVC用CPU例外ハンドラの定義
sdef_sex	s_fncd	exhatr	sexhdr	拡張SVC用システムコール例外ハンドラの定義

データタイプ (ITRON2)

```
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 long              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 long     UW;    /* 符号無し32ビット整数 */
typedef long              VW;    /* データタイプが一定しないもの
                                    （32ビットサイズ） */
typedef short             VH;    /* データタイプが一定しないもの
                                    （16ビットサイズ） */
typedef char              VB;    /* データタイプが一定しないもの
                                    （8ビットサイズ） */
typedef void              *VP;   /* データタイプが一定しないものへの
                                    ポインタ */
typedef void              (*FP)( ); /* プログラムのスタートアドレス一般 */
```

※ VB, VH, VW と B, H, W との違いは、前者がビット数のみがわかっており、データタイプの中味がわからないものを表すのに対して、後者は整数を表しているという点である。例えば、メッセージの中身やメモリブロックはバイトの配列であるが、中に入れるデータの実際のデータタイプは場合によって異なっているので、VB のデータタイプを使っている。また、パケットの中でリザーブ領域を確保する場合は、データのサイズはわかっているが整数であるとは言えないため、VH, VW のデータタイプを使っている。

```
/* 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タイプや特
   殊な意味を持つデータタイプをあらかじめ用意しておく */
typedef W                 BOOL;  /* ブール値、FALSE（0 - 偽）または
                                    TRUE（1 - 真） */
```

```
typedef W       ID;             /* オブジェクトのID (XXXid) */
        /* 値の範囲はシステムに依存して決まる */
        /* 値の範囲は2バイト以下だが、データタイプは4バイトである */
        (理由) 2バイトとすると、ポインタを使う時に紛らわしくなる。
                将来は4バイトへの拡張を考えることができる。
typedef W       HNO;            /* ハンドラ番号 */
        /* 値の範囲は2バイトだが、データタイプは4バイトである。*/
typedef W       ER;             /* エラーコード */
        /* 0 か負の値しかとらないので、正の値を持つ他のタイプと合成可
           能である */
        /* 値の範囲は2バイトだが、データタイプは4バイトである */
typedef W       TMO;            /* タイムアウト */
        /* 範囲は (-1)〜(2^31-1)、0でポーリング、(-1)で永久待ち */
typedef UW      ATR;            /* オブジェクト属性 */
typedef UW      HATR;           /* ハンドラ属性 */

/* 以下は4バイトへの拡張の必要性が少ないことと、パケットに入ること
   があるため、2バイトとする */
typedef H       FN;             /* 機能コード */
typedef H       TPRI;           /* タスク優先度、範囲は (-16)〜255 */
        /* 値の範囲は (1バイト+α) だが、データタイプは2バイトである */

/* タスクやハンドラのデータタイプ */
        /* ハンドラやタスクを表わす関数に対して、戻り値の有無やハンド
           ラ終了 (戻り) システムコールとの対応に着目し、以下のようなデー
           タタイプを設ける。*/
typedef void    TASK;           /* タスク - ext_tsk, exd_tsk で終了 */
typedef void    INTHDR;         /* 割込みハンドラ - ret_int で終了 */
typedef ER      SVCHDR;         /* 拡張SVCハンドラ - ret_svc で終了 */
typedef void    TMRHDR;         /* タイマハンドラ - ret_tmr で終了 */
typedef void    EXCHDR;         /* 終了,例外ハンドラ - ret_exc で終了 */
typedef void    INIHDR;         /* 初期化ハンドラ */
```

※ 初期化ハンドラは、カーネルが起動される際に1回だけ実行されるハンドラである。このハンドラを定義することで、初期オブジェクトの生成や

例外環境の定義などを行なうことができる。初期化ハンドラの有無や実行コンテキスト(タスク部、準タスク部、タスク独立部) の区別はインプリメント依存である。

```
typedef TASK        (*TASKP)( );      /* タスクスタートアドレス */
typedef INTHDR      (*INTHDRP)( );  /* 割込みハンドラスタートアドレス */
typedef SVCHDR      (*SVCHDRP)( ); /*拡張SVCハンドラスタートアドレス */
typedef TMRHDR      (*TMRHDRP)( );/* タイマハンドラスタートアドレス */
typedef EXCHDR      (*EXCHDRP)( ); /* 例外ハンドラスタートアドレス */
typedef INIHDR      (*INIHDRP)( );   /* 初期化ハンドラスタートアドレス */

/* プロセッサのアーキテクチャに関連したデータタイプ */
typedef UW                PSW;          /* PSW */
```

【記述例】

```
ER  ercd = sta_tsk ( ID tskid, W stacd );
```

言語Cインタフェース (ITRON2)

●動的機能コードによるシステムコール呼び出し

```
ER ercd = sys_cal   ( FN fncd, VW par1, VW par2, VW par3, VW par4 );
```

※ インタフェースルーチンを用意せずに拡張SVCを実行する場合などに利用する。

※ パラメータは最大4個で制限する。parN が Reg N に入る。パラメータがそれより多い場合は、パケットを利用する必要がある。また、パケット以外のリターンパラメータは戻せない。

※ サブ機能コード(sfncd) がある場合には、par1 が sfncd を表わす。

●タスク管理機能

```
ER ercd = cre_tsk  ( ID tskid, ATR tskatr, TASKP stadr, TPRI itskpri, W stksz );
ER ercd = sta_tsk  ( ID tskid, W stacd );
ER ercd = del_tsk  ( ID tskid );
void       ext_tsk  ( );
void       exd_tsk ( );
void       abo_tsk ( UW exccd );
ER ercd = ter_tsk  ( ID tskid, UW exccd );
ER ercd = ras_ter  ( ID tskid, UW exccd );
ER ercd = chg_pri  ( ID tskid, TPRI tskpri );
ER ercd = rot_rdq  ( TPRI tskpri );
ER ercd = rel_wai  ( ID tskid );
ER ercd = get_tid  ( ID *p_tskid );
ER ercd = tsk_sts  ( T_TSKS *pk_tsks, ID tskid );
ER ercd = hdr_sts  ( T_HDRS *ar_hdrs, ID tskid, W hdstart, W hdcnt );
```

●タスク付属同期機能

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

```
ER ercd = rsm_tsk ( ID tskid );
ER ercd =frsm_tsk ( ID tskid );
ER ercd = slp_tsk  ( );
ER ercd = wai_tsk ( TMO tmout );
ER ercd = wup_tsk( ID tskid );
ER ercd = can_wup( W *p_wupcnt, ID tskid );
```

●同期・通信機能

```
ER ercd = cre_flg  ( ID flgid, ATR flgatr );
ER ercd = del_flg  ( ID flgid );
ER ercd = set_flg  ( ID flgid, UW setptn );
ER ercd = clr_flg  ( ID flgid, UW clrptn );
ER ercd = wai_flg ( UW *p_flgptn, ID flgid, UW waiptn, UW wfmode,
                    TMO tmout );
ER ercd = flg_sts  ( T_FLGS *pk_flgs, ID flgid );
ER ercd = cre_sem ( ID semid, ATR sematr, W isemcnt );
ER ercd = del_sem ( ID semid );
ER ercd = sig_sem ( ID semid, W rcnt );
ER ercd = wai_sem( ID semid, W rcnt, TMO tmout );
ER ercd = sem_sts ( T_SEMS *pk_sems, ID semid );
ER ercd = cre_mbx ( ID mbxid, ATR mbxatr );
ER ercd = del_mbx ( ID mbxid );
ER ercd = snd_msg ( ID mbxid, T_MSG *pk_msg);
ER ercd = rcv_msg ( T_MSG **ppk_msg, ID mbxid, TMO tmout );
ER ercd = mbx_sts ( T_MBXS *pk_mbxs, ID mbxid );
```

●割込み管理機能

```
ER ercd = def_int  ( UH intvec, HATR inhatr, INTHDRP inthdr, PSW imask );
void       ret_int  ( );
ER ercd = chg_ims( PSW imask );
```

●例外管理機能

```
ER ercd = def_ext  ( ID tskid, HATR exhatr, EXCHDRP exthdr );
ER ercd = def_cex ( ID tskid, HATR exhatr, EXCHDRP cexhdr );
```

```
ER ercd = def_sex ( ID tskid, HATR exhatr, EXCHDRP sexhdr );
void       ret_exc  ( );
ER ercd = end_exc( );
ER ercd = clr_ems ( ID tskid, UW clrptn );
ER ercd = set_ems ( ID tskid, UW setptn );
ER ercd =idef_ext  ( HATR exhatr, EXCHDRP exthdr );
ER ercd =idef_cex ( HATR exhatr, EXCHDRP cexhdr );
ER ercd =idef_sex ( HATR exhatr, EXCHDRP sexhdr );
```

●メモリプール管理機能

```
ER ercd = cre_mpl ( ID mplid, ATR mplatr, W mplsz, W blksz );
ER ercd = del_mpl ( ID mplid );
ER ercd = get_blk  ( VP *p_blk, ID mplid, W bcnt, TMO tmout );
ER ercd = rel_blk  ( ID mplid, VP blk );
ER ercd = mpl_sts ( T_MPLS *pk_mpls, ID mplid );
ER ercd = blk_sts  ( ID *p_mplid, W *p_bcnt, VP blk );
```

●時間管理機能

```
ER ercd = set_tim  ( H utime, UW ltime );
ER ercd = get_tim  ( H *p_utime, UW *p_ltime );
ER ercd = dly_tsk  ( W dtime );
ER ercd = cyc_wup    ( ID tskid, W rptcnt, W cytime, H utime, UW ltime,
                       UW tmmode);
ER ercd = can_cyc ( ID tskid );
```

●システム管理機能

```
ER ercd = def_svc ( FN s_fncd, HATR svhatr, SVCHDRP svchdr );
void       ret_svc  ( ER s_ercd );
ER ercd = get_ver  ( T_VER *pk_ver );
ER ercd = psw_sts ( PSW *p_psw );
```

●拡張同期・通信機能

```
ER ercd = cre_mbf ( ID mbfid, ATR mbfatr, W bufsz, W maxbmsz );
ER ercd = del_mbf ( ID mbfid );
ER ercd = snd_mbf( ID mbfid, VP pk_bmsg, W bmsgsz );
```

```
ER ercd = rcv_mbf( VP pk_bmsg, W *p_bmsgsz, ID mbfid, TMO tmout );
ER ercd = mbf_sts ( T_MBFS *pk_mbfs, ID mbfid );
ER ercd = cre_por  ( ID porid, ATR poratr );
ER ercd = del_por  ( ID porid );
ER ercd = cal_por  ( T_RMSG *pk_rmsg, ID porid, UW calptn, W maxrmsz,
                     TMO tmout );
ER ercd = acp_por ( ID *p_ctskid, T_RMSG *pk_rmsg, ID porid, UW acpptn,
                     W maxrmsz, TMO tmout );
ER ercd = fwd_por( ID porid, ID ctskid, T_RMSG *pk_rmsg, UW calptn );
ER ercd = rpl_por  ( ID ctskid, T_RMSG *pk_rmsg );
ER ercd = por_sts  ( T_PORS *pk_pors, ID porid );
ER ercd = rdv_sts  ( T_RDVS *ar_rdvs, ID porid, W rvstart, W rvcnt );
```

●強制例外機能

```
ER ercd = def_fex  ( ID tskid, HATR exhatr, EXCHDRP fexhdr );
ER ercd = ras_fex  ( ID tskid, UW exccd );
```

●ローカルメモリプール管理機能〕

```
ER ercd = cre_lmp ( ID lmpid, ATR lmpatr, W lmpsz, W blksz );
ER ercd = del_lmp ( ID lmpid );
ER ercd = get_lbl   ( VP *p_blk, ID lmpid, W bcnt, TMO tmout );
ER ercd = rel_lbl   ( ID lmpid, VP blk );
ER ercd = lmp_sts ( T_LMPS *pk_lmps, ID lmpid );
ER ercd = lbl_sts   ( ID *p_lmpid, W *p_bcnt, VP blk );
```

●資源管理サポート機能

```
ER ercd =usig_sem  ( W *p_rcnt, ID semid, W rcnt );
ER ercd =uwai_sem ( W *p_rcnt, ID semid, W rcnt, TMO tmout );
ER ercd =usnd_msg ( T_MSG **ppk_msg, ID mbxid );
ER ercd =urcv_msg ( T_MSG **ppk_msg, ID mbxid, TMO tmout );
ER ercd =uget_blk   ( VP *p_blk, ID mplid, W bcnt, TMO tmout );
ER ercd =uget_lbl    ( VP *p_blk, ID lmpid, W bcnt, TMO tmout );
ER ercd =urel_blk    ( VP *p_blk, ID mplid );
ER ercd =urel_lbl    ( VP *p_blk, ID lmpid );
```

●タイマハンドラ機能

```
ER ercd = def_cyc ( HNO cyhno, HATR cyhatr, TMRHDRP cychdr,
                    UW cyhact, W cytime );
ER ercd = act_cyc  ( HNO cyhno, UW cyhact );
ER ercd = cyh_sts  ( T_CYHS *pk_cyhs, HNO cyhno );
ER ercd = def_alm ( HNO alhno, HATR alhatr, TMRHDRP almhdr, H utime,
                    UW ltime, UW tmmode );
ER ercd = alh_sts   ( T_ALHS *pk_alhs, HNO alhno );
void        ret_tmr  ( );
```

●拡張SVCサポート機能

```
ER ercd =sdef_ext ( FN s_fncd, HATR exhatr, EXCHDRP exthdr );
ER ercd =sdef_cex ( FN s_fncd, HATR exhatr, EXCHDRP cexhdr );
ER ercd =sdef_sex ( FN s_fncd, HATR exhatr, EXCHDRP sexhdr );
```

共通定数のニモニックと標準値(ITRON2)

```
/*-----------------------------------------------------------------------------------------*/
全体:
    NADR        (-1)    /* アドレスやポインタ値が無効 */
    TRUE        1       /* 真 */
    FALSE       0       /* 偽 */
tmout:
    TMO_POL     0       /* ポーリング */
    TMO_FEVR    (-1)    /* 永久待ち */
tskid, wtskid:
    TSK_SELF    0       /* 自タスク指定 */
    TSK_CMN     (-1)    /* タスク間共通の例外ハンドラの定義 */
    TSK_INDP    (-2)    /* タスク独立部の指定 */
    FALSE       0       /*タスク独立部、あるいは待ちタスク無しを
                          表わす(リターンパラメータのみ) */
tskpri:
    TPRI_INI    0       /* タスク起動時の初期優先度を指定 (chg_pri) */
    TPRI_RUN    0       /* その時実行中の最高優先度を指定 (rot_rdq) */
mplid:
    TMPL_OS     (-4)    /* OS用のメモリプール */
mbfid:
    TMBF_OS     (-4)    /* OSのエラーログ用のメッセージバッファ */
    TMBF_DB     (-3)    /* デバッグ用のメッセージバッファ */
tmmode:
    TTM_ABS     0       /* 絶対時刻での指定 */
    TTM_REL     1       /* 相対時刻での指定 */
s_fncd:
    TFN_CMN     (-1)    /* 拡張SVC共通の例外ハンドラの定義 */
```

spcid:
　TSP_SELF　　0　　/* 自タスクの属する空間 */
　TSP_CMN　　(-1)　　/* 共通空間 */
poratr:
　TA_NULL　　0　　/* 特別な属性を指定しない */
※ TA_NULLは、属性などの機能を OFF にする場合に、0 の代わりに使用するニモニックである。

inhatr,exhatr,cyhatr,alhatr,svhatr:
　TA_HLNG　　H'00000100　　/* 高級言語によるプログラム */
　TA_ASM　　H'00000000　　/* アセンブラによるプログラム */
※ TA_ASM は、タスクやハンドラがアセンブラレベルから直接起動されることを表わす。TA_ASM は TA_HLNG と反対の意味を持つ属性である。

```
/*-----------------------------------------------------------------------------------------*/
/* システムコール xxx_yyy を呼び出すための機能コードをニモニックで表わす場合には、TFN_XXX_YYY のニモニックを使用する。
   代表的なシステムコールの機能コードに対するニモニックは次のようになる。*/
   TFN_CRE_TSK (-17)     H'ffef     タスクを生成する
   TFN_STA_TSK (-23)     H'ffe9     タスクを起動する
         ...
   TFN_DEF_INT  (-65)    H'ffbf     割込みハンドラを定義する
         ...

/*-----------------------------------------------------------------------------------------*/
```

共通定数の適用システムコール(ITRON2)

前項のレファレンスに出てくる定数と、それを適用できるシステムコールとの関係を以下に示す。○はその組み合せが可能である（意味がある）ことを示し、E_xxxx は表記のエラーが発生することを示す。ただし、発生するエラーコードの詳細にはインプリメント依存の部分がある。

●tmout:	TMO_POL	TMO_FEVR
wai_tsk	○	○
wai_flg	○	○
wai_sem	○	○
rcv_msg	○	○
get_blk	○	○
rcv_mbf	○	○
cal_por	○	○
acp_por	○	○
get_lbl	○	○
uwai_sem	○	○
urcv_msg	○	○
uget_blk	○	○
uget_lbl	○	○

●tskid:	TSK_SELF	TSK_CMN	（参考）自タスクのIDを指定
cre_tsk	E_RSID	E_RSID	E_EXS
sta_tsk	E_RSID	E_RSID	E_NODMT
del_tsk	E_RSID	E_RSID	E_NODMT
ter_tsk	E_SELF	E_RSID	E_SELF
ras_ter	E_SELF	E_RSID	E_SELF
chg_pri	○	E_RSID	○
rel_wai	E_RSID	E_RSID	E_NOWAI
tsk_sts	○	E_RSID	○

sus_tsk	E_SELF	E_RSID	E_SELF
rsm_tsk	E_SELF	E_RSID	E_SELF
frsm_tsk	E_SELF	E_RSID	E_SELF
wup_tsk	E_SELF	E_RSID	E_SELF
can_wup	○	E_RSID	○
def_ext	○	○	○
def_cex	○	○	○
def_sex	○	○	○
clr_ems	○	E_RSID	○
set_ems	○	E_RSID	○
cyc_wup	○	E_RSID	○
can_cyc	○	E_RSID	○
def_fex	○	○	○
ras_fex	E_SELF	E_RSID	E_SELF

※ 上記のうち、E_SELF は、自タスク操作と他タスク操作を区別するという方針に基づいたエラーを示す。E_SELF は、自タスクのIDを指定した場合にも TSK_SELF を指定した場合にも共通して発生する。

●tskpri:	TPRI_INI	TPRI_RUN
chg_pri	○	
rot_rdq		○

※ TPRI_INI と TPRI_RUN は値が同じである。

●mplid:	TMPL_OS
cre_mpl	E_RSID
del_mpl	E_RSID
get_blk	○
rel_blk	○
mpl_sts	○
uget_blk	○
urel_blk	○

●mbfid:	TMBF_OS
cre_mbf	E_RSID
del_mbf	E_RSID

```
    snd_mbf             E_RSID
    rcv_mbf             ○
    mbf_sts             ○

●tmmode:                TTM_ABS     TTM_REL
    cyc_wup             ○           ○
    def_alm             ○           ○

●s_fncd:                TFN_CMN
    def_svc             E_ILFN
    sdef_ext            ○
    sdef_cex            ○
    sdef_sex            ○

●XXhatr:                TA_HLNG
    def_int             ○
    def_ext             ○
    def_cex             ○
    def_sex             ○
    idef_ext            ○
    idef_cex            ○
    idef_sex            ○
    def_svc             ○
    def_fex             ○
    def_cyc             ○
    def_alm             ○
    sdef_ext            ○
    sdef_cex            ○
    sdef_sex            ○
/*-------------------------------------------------------------------------------------------*/
```

構造体のパケット形式 (ITRON2)

```
typedef struct t_tsks {
    ATR     tskatr;     /* タスク属性 */
    ID      spcid;      /* スペースID、共通空間では TSP_CMN(-1) */
    UW      tskstat;    /* タスク状態 */
    UW      epndptn;    /* ペンディング中の終了要求と強制例外 */
    UW      tskwait;    /* 待ち要因 */
    ID      wid;        /* 待ちオブジェクトID */
    W       suscnt;     /* SUSPEND要求カウント、0≦ suscnt＜ 2^31 */
    W       wupcnt;     /* 起床要求カウント、0≦ wupcnt＜ 2^31 */
    TASKP   stadr;      /* タスクスタートアドレス */
    VW      isp;        /* 初期SP値 */
    TPRI    itskpri;    /* 初期優先度 */
    TPRI    tskpri;     /* 現在優先度 */
} T_TSKS;

tskatr:
    TA_ASM      H'00000000      /* アセンブラによるプログラム */
    TA_HLNG     H'00000100      /* 高級言語によるプログラム */
    TA_COP0     H'00000080      /* ID=0   のコプロセッサを使用 */
    TA_COP1     H'00000040      /* ID=1   のコプロセッサを使用 */
    TA_COP2     H'00000020      /* ID=2   のコプロセッサを使用 */
    TA_COP3     H'00000010      /* ID=3   のコプロセッサを使用 */
    TA_COP4     H'00000008      /* ID=4   のコプロセッサを使用 */
    TA_COP5     H'00000004      /* ID=5   のコプロセッサを使用 */
    TA_COP6     H'00000002      /* ID=6   のコプロセッサを使用 */
    TA_COP7     H'00000001      /* ID=7   のコプロセッサを使用 */

tskstat:
    TTS_RUN     H'00000001      /* RUN */
```

```
    TTS_RDY     H'00000002      /* READY */
    TTS_WAI     H'00000004      /* WAIT */
    TTS_SUS     H'00000008      /* SUSPEND */
    TTS_WAS     H'0000000c      /* WAIT-SUSPEND */
    TTS_DMT     H'00000010      /* DORMANT */
    TTS_DBG     H'00000020      /* デバッグ待ち */
tskwait:
    TTW_SLP     H'00000001      /* slp_tsk, wai_tsk による待ち */
    TTW_DLY     H'00000002      /* dly_tsk による待ち */
    TTW_FLG     H'00000010      /* wai_flg による待ち */
    TTW_SEM     H'00000020      /* wai_sem, uwai_sem による待ち */
    TTW_MBX     H'00000040      /* rcv_msg, urcv_msg による待ち */
    TTW_MPL     H'00000080      /* get_blk, uget_blk による待ち */
    TTW_MBF     H'00000100      /* rcv_mbf による待ち */
    TTW_CAL     H'00000200      /* ランデブ呼出待ち */
    TTW_ACP     H'00000400      /* ランデブ受付待ち */
    TTW_RPL     H'00000800      /* ランデブ終了待ち */
    TTW_LMP     H'00001000      /* get_lbl, uget_lbl による待ち */
```

※ tskstat, tskwait 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、WAIT または READY であることを判定する）場合に便利である。

```
typedef struct t_hdrs {         /* ハンドラ実行環境 */
    UW      emsptn;             /* 例外マスク */
    T_EXC   *pk_exc;            /* 例外情報 */
    T_REGS  *pk_regs;           /* ハンドラ起動時のレジスタ */
    T_EIT   *pk_eit;            /* ハンドラ起動時の PC, PSW */
} T_HDRS;

typedef struct t_exc {          /* 例外情報 */
    UW      eclsptn;            /* 例外クラスビットパターン */
    UW      exccd;              /* 例外コード（エラーコード）*/
    FN      fncd;               /* 例外発生システムコール機能コード */
    VW      _Wrsv_10;           /* インプリメント依存追加情報 */
```

```
        ...
} T_EXC;
```

※ システムコール例外でない場合は、fncd は 0 となる。

※ fncd は、拡張SVCからの戻り時に発生するシステムコール例外の時にも有効である。例えば、拡張SVC{S} から ret_svc で戻る際には、s_ercd の値により、戻り先のコンテキストでシステムコール例外を発生する場合がある。この時、そのシステムコール例外ハンドラに対して渡される情報(T_EXC)の中の fncd　の値は、{S}の機能コードになる。なお、この時の s_ercd の値は、exccd あるいは pk_regs の中の r0 を参照することによって得ることができる。

```
typedef struct t_regs {
    VW    r0;
    VW    r1;
    VW    r2;
    VW    r3;
    VW    r4;
    VW    r5;
    VW    r6;
    VW    r7;
    VW    r8;
    VW    r9;
    VW    r10;
    VW    r11;
    VW    r12;
    VW    r13;
    VW    r14;
    VW    sp;
} T_REGS;
```

※ システムコール例外の発生時にこのパケットを参照した場合、r0 は ercd なので pk_exc のexccd と等しくなる。

※ システムコール例外の発生時にこのパケットを参照した場合、r1～ の内容が例外を発生したシステムコールのパラメータとなる。

※ 一部のシステムコールでは、r2 などでリターンパラメータとパラメータを兼ねている場合があり、リターンパラメータを返すとパラメータの情報が失われる場合がある。しかし、エラー発生の際は、できるだけエラー発生前の状態に戻すのが原則である。つまり、エラーコード以外のリターンパラメータは返さず、システムコール実行前のレジスタの値をそのまま残すのが望ましい。そうなれば、システムコールのパラメータはレジスタ上に保存されることになるので問題はない。(インプリメントとの関係で検討を要する。どうしても問題がある場合は、pk_exc の後ろの追加情報を利用する。)

なお、r0 では ercd により fncd が必ず破壊されるので、fncd の情報のみは pk_exc で別に用意している。

```
typedef struct t_eit {          /* ハンドラ起動時の PC,PSW（EIT情報） */
    PSW     psw;
    EITINF  eitinf;             /* 終了例外や強制例外の場合は不定値 */
    FP      pc;
    FP      expc;               /* 無い場合もあり */
    IOINF   ioinf;              /* 無い場合もあり */
    VP      eraddr;             /* 無い場合もあり */
    VW      erdata;             /* 無い場合もあり */
} T_EIT;
/*-----------------------------------------------------------------------------------------------*/
typedef struct t_flgs {
    ATR   flgatr;               /* イベントフラグ属性 */
    ID    wtskid;               /* 待ち行列先頭のタスクID */
    UW    flgptn;               /* 現在のフラグ値 */
} T_FLGS;

flgatr:
    TA_WMUL     H'00000000      /* 複数タスクの待ちを許す
                                   (Wait Multiple Task) */
    TA_WSGL     H'00000008      /* 複数タスクの待ちを許さない
                                   (Wait Single Task) */
wfmode:
    TWF_CLR     1               /* クリア指定 */
```

```
    TWF_ANDW    0           /* AND待ち */
    TWF_ORW     2           /* OR待ち */
/*-----------------------------------------------------------------------------------------*/
typedef struct t_sems {
    ATR   sematr;           /* セマフォ属性 */
    ID    wtskid;           /* 待ち行列先頭のタスクID */
    W     semcnt;           /* 現在のカウント値 */
} T_SEMS;

sematr:
    TA_TFIFO    H'00000000      /* 待ちタスクのキューイングはFIFO */
    TA_TPRI     H'00000001      /* 待ちタスクのキューイングは優先度順 */
    TA_FIRST    H'00000000      /* 行列先頭のタスクを優先扱い */
    TA_CNT      H'00000002      /* 要求数の少ないタスクを優先扱い */
    TA_VAR      H'00000000      /* 要求/返却資源数を指定可能 */
    TA_FIX      H'00000004      /* 要求/返却資源数は1のみ */
/*-----------------------------------------------------------------------------------------*/
typedef struct t_mbxs {
    ATR         mbxatr;         /* メイルボックス属性 */
    ID          wtskid;         /* 待ち行列先頭のタスクID */
    T_MSG       *pk_msg;        /* 次に受信されるメッセージのアドレス */
} T_MBXS;

mbxatr:
    TA_TFIFO    H'00000000      /* 待ちタスクはFIFO */
    TA_TPRI     H'00000001      /* 待ちタスクは優先度順 */
    TA_MFIFO    H'00000000      /* メッセージはFIFO */
    TA_MPRI     H'00000002      /* メッセージは優先度順 */

typedef struct t_msg {
    VP    msgsent;
    VP    _Wrsv_04;
    UH    msgtype;          /* 現在は reserved、0 を入れる必要あり */
    H     msgpri;           /* メッセージ優先度 */
    VB    msgcont[ ];
```

```
} T_MSG;
/*-----------------------------------------------------------------------------------------*/
typedef struct t_mpls {
    ATR   mplatr;           /* メモリプール属性 */
    ID    wtskid;           /* 待ち行列先頭のタスクID */
    W     frbcnt;           /* 空き領域全体のブロック数 */
    W     maxbcnt;          /* 最大の連続空き領域のブロック数 */
    W     mplsz;            /* メモリプール全体のサイズ */
    W     blksz;            /* メモリブロックのサイズ */
    W     fragcnt;          /* 空きブロックのフラグメント数 */
} T_MPLS;

mplatr:
    TA_TFIFO    H'00000000    /* 待ちタスクはFIFO */
    TA_TPRI     H'00000001    /* 待ちタスクは優先度順 */
    TA_FIRST    H'00000000    /* 行列先頭のタスクを優先扱い */
    TA_CNT      H'00000002    /* 要求数の少ないタスクを優先扱い */
    TA_VAR      H'00000000    /* 可変長メモリブロック用のメモリプール */
    TA_FIX      H'00000004    /* 固定長メモリブロック用のメモリプール */
/*-----------------------------------------------------------------------------------------*/
typedef struct t_mbfs {
    ATR   mbfatr;           /* メッセージバッファ属性 */
    ID    wtskid;           /* 待ち行列先頭のタスクID */
    W     bmsgsz;           /* 次に受信されるメッセージのサイズ */
    W     maxbmsz;          /* メッセージの最大サイズ */
    W     frbufsz;          /* 空きバッファサイズ */
    W     bufsz;            /* バッファ全体のサイズ */
} T_MBFS;

mbfatr:
    TA_TFIFO    H'00000000    /* 待ちタスクはFIFO */
    TA_TPRI     H'00000001    /* 待ちタスクは優先度順 */
/*-----------------------------------------------------------------------------------------*/
typedef struct t_pors {
    ATR     poratr;         /* ポート属性 */
```

```
    ID          wtskid;         /* 呼出側待ち行列先頭のタスクID */
    ID          atskid;         /* 受付側待ち行列先頭のタスクID */
} T_PORS;
typedef struct t_rdvs {
    ID          ctskid;         /* ランデブ中の呼出側タスクのID */
    ID          atskid;         /* ランデブ中の受付側タスクのID */
    T_RMSG      *pk_rmsg;       /* 返答時に渡されるメッセージを受取るアドレス */
    W           maxrmsz;        /* 受け取ることのできるメッセージの最大長 */
} T_RDVS;
typedef struct t_rmsg {
    W           rmsgsz;         /* メッセージサイズ */
    VB          rmsgcont[ ];    /* メッセージの中身 */
} T_RMSG;
/*-----------------------------------------------------------------------------------------*/
typedef struct t_lmps {
    ATR         lmpatr;         /* ローカルメモリプール属性 */
    ID          wtskid;         /* 待ち行列先頭のタスクID */
    W           frbcnt;         /* 空き領域全体のブロック数 */
    W           maxbcnt;        /* 最大の連続空き領域のブロック数 */
    W           lmpsz;          /* メモリプール全体のサイズ */
    W           blksz;          /* メモリブロックのサイズ */
    W           fragcnt;        /* 空きブロックのフラグメント数 */
} T_LMPS;
lmpatr:
    TA_TFIFO    H'00000000      /* 待ちタスクはFIFO */
    TA_TPRI     H'00000001      /* 待ちタスクは優先度順 */
    TA_FIRST    H'00000000      /* 行列先頭のタスクを優先扱い */
    TA_CNT      H'00000002      /* 要求数の少ないタスクを優先扱い */
    TA_VAR      H'00000000      /* 可変長メモリブロック用のメモリプール */
    TA_FIX      H'00000004      /* 固定長メモリブロック用のメモリプール */
/*-----------------------------------------------------------------------------------------*/
typedef struct t_cyhs {
    HATR        cyhatr;             /* 周期起動ハンドラ属性 */
```

```
    W           cytime;         /* 周期 */
    VW          _Wrsv_08;
    W           lftime;         /* 次の起動までの残り時間(LeftTime) */
    UW          cyhact;         /* 活性状態 */
} T_CYHS;

cyhatr:
    TA_ASM      H'00000000      /* アセンブラによるプログラム */
    TA_HLNG     H'00000100      /* 高級言語によるプログラム */

cyhact:
    TCY_OFF     0               /* 周期起動ハンドラが起動されない */
    TCY_ON      1               /* 周期起動ハンドラが起動される */
    TCY_INI     2               /* 周期のカウントが初期化される */

typedef struct t_alhs {
    HATR        alhatr;         /* アラームハンドラ属性 */
    VW          _Wrsv_04;
    VH          _Hrsv_08;
    H           lfutime;        /* 次の起動までの残り時間（上位） */
    UW          lfltime;        /* 次の起動までの残り時間（下位） */
} T_ALHS;

alhatr:
    TA_ASM      H'00000000      /* アセンブラによるプログラム */
    TA_HLNG     H'00000100      /* 高級言語によるプログラム */
/*-----------------------------------------------------------------------------------------*/
typedef struct t_ver {
    UH    maker;            /* メーカー */
    UH    id;               /* 形式番号 */
    UH    spver;            /* 仕様書バージョン */
    UH    prver;            /* 製品バージョン */
    UH    prno[4];          /* 製品管理情報 */
    UH    cpu;              /* CPU情報 */
    UH    var;              /* バリエーション記述子 */
} T_VER;
/*-----------------------------------------------------------------------------------------*/
```

μITRONシステムコール索引

この索引は、本書で説明されるμITRONシステムコールのアルファベット順索引である。

この索引は、本書で説明されるITRON2システムコールのアルファベット順索引である。

ITRON・μITRON標準ハンドブック

1990年12月15日　初版1刷発行

監　　修　　坂村 健

発 行 所　　パーソナルメディア株式会社
〒142　東京都品川区平塚1-7-7 MYビル
電 話　東京(03)5702-0502
振 替　東京4-105703

印刷／製本　　倉敷印刷

Printed in Japan　　ISBN4-89362-079-7　C3055
定価はカバーに記載