On-Chip Embedded Network Solution

NORTi® Oceans

カーネル編

ユーザーズガイド

MiSPO

On-Chip Embedded Network Solution
NORTi Oceans ユーザーズガイド・カーネル編

はじめに

μITRON仕様OSの普及に最も貢献し、組み込みTCP/IPも広く浸透させたNORTiシリーズの究極形「NORTi Oceans（ノートアイ オーシャンズ）」は、ワンチップマイコン向けに、従来の「NORTi Version 4」よりコードサイズ約 6 割減、データサイズ約 8 割減という劇的な軽量化を実現したリアルタイムOSです。

コンフィグレーション不要で、ユーザープログラムとライブラリとをリンクするだけで OS の機能が利用できるようになる画期的手法を NORTi はもたらしましたが、NORTi Ocenas では、従来よりもっと手軽に組み込めることを目指し、MCU シリーズ別のパッケージとして、MCU 内蔵周辺機能にもきめ細かく対応しています。

開発者の人数に応じて、コンパイラと同等以下の低価格で導入でき、応用製品の制限なし、組込みロイヤリティフリーという NORTi シリーズの特長はそのままに、さらに、使い易さと品質に対する絶対的な自信から、保守費用も無料としました。

NORTi Oceans により、これまでリソース不足のために困難だったワンチップベースの高度な機器の設計が、低コストで可能となります。MCU の能力を最大限に引き出すことができる NORTi Oceans を、皆様方の新世代の組み込みシステム開発に、どうぞお役立てください。

本書について

本書「カーネル編」は、NORTi Oceans のリアルタイム・マルチタスク機能の共通マニュアルです。前半部で各機能の概要を、後半部で各システムコールの詳細な解説を行ってありす。MCU 固有の情報については、インストールした DOC フォルダにある補足説明書を参照してください。Ethernet コントローラを内蔵した MCU 向けに標準提供される TCP/IP プロトコルスタックについては、「TCP/IP 編」のユーザーズガイドを参照してください。

お問い合せ先

株式会社ミスポ宛てのご質問は、電子メールにて下記で承ります。
一般的なお問い合せ：sales@mispo.co.jp
技術サポートご依頼：norti@mispo.co.jp

目次

第１章 基本事項

1.1 特長

高速な応答性

NORTi シリーズはプリエンプティブなマルチタスク OS です。イベントの発生によって優先度を基準にしたスケジューリングが行われ、即座にタスクが切り替わります。十分に吟味されたコードでカーネルは構成されており、システムコール内部でスキャンすることなく一発で操作対象(オブジェクト)が選択されます。また、NORTi シリーズでは、システムコールの実行途中で一旦割り込を許可するという独自の高度な手法により、一般的なμITRON 仕様 OS の実装に比べて割り込み禁止時間が半減されていています。さらに、カーネルより高優先の(OS 管理外の)割り込みハンドラを共存させることができ、その割り込み禁止時間は、限りなくゼロです。

μITRON4.0 仕様サブセット

NORTi Oceans では、ワンチップマイコン向けの現実的な選択肢として、μITRON4.0 仕様から、小規模な組み込みシステムには過剰といえる機能を除いて実装してあります。具体的には、まず、使う機会が稀なタスク強制待ち、タスク例外処理、ランデブ用ポート、アラームハンドラ、オーバーランハンドラ、各オブジェクトでのタスク優先度順待ちの機能を削除してあります。また、オーバーヘッドが大きく他の機能で代用可能なミューテックス、メッセージバッファ、割り込みサービスルーチンも未実装です。そして、メモリ制約の厳しいプログラムでは論外の可変長メモリプールの機能や、動的な各オブジェクトの削除と再生成の機能も省いてあります。

C で設計されたカーネル

1991 年の発表以来、リアルタイム OS はアセンブリ言語で設計するものという常識を、NORTi シリーズは覆して来ました。レジスタの最適な割り付けや最小限の待避をコンパイラに任せ、コンパイラが待避するレジスタをカーネルでは二重に保存しないという発想と、コンパイラのコード展開を把握したコーディング技術とにより、アセンブリ言語より C 言語で設計する方が高速という事実を証明しました。さらに、カーネルのソースコードの大部分を共用できますので、新たな MCU 対応でも、リリース直後から高い信頼性を確保できます。

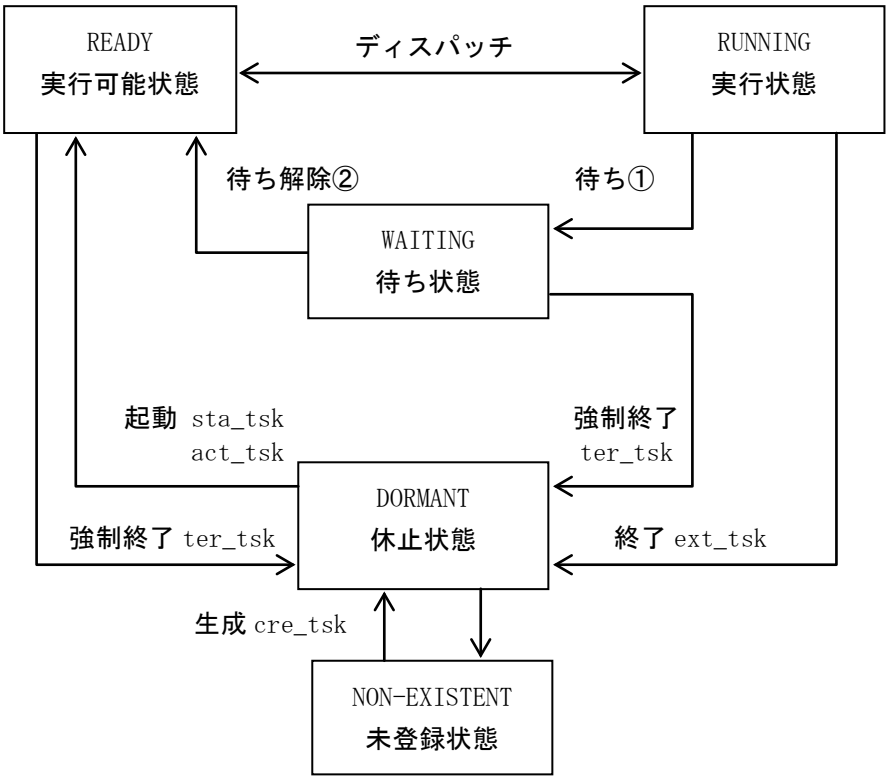
コンパクトなサイズ

システムコール単位でライブラリ化された NORTi Oceans では、使用しない機能がリンクされず、自動的に最小のコード構成となります。タスク管理ブロック(TCB)等の領域を、必要な数だけ詰めて動的に確保する仕組みにより、面倒なコンフィグレーションをすることなく最小のデータサイズとなります。ワンチップマイコンの貴重な内蔵メモリを、1 バイトたりとも無駄にしません。

1.2 タスクの状態

並列処理するプログラムの単位をタスクと呼び、タスクは、NON-EXISTENT, DORMANT, READY, RUNNING, WAITING の５つの状態のいずれかをとります。

タスクの状態遷移と、それを引き起こすシステムコールを図に示します。



① slp_tsk, tslp_tsk, wai_sem, twai_sem, wai_flg, twai_flg, rcv_mbx, trcv_mbx, get_mpf, tget_mpf, dly_tsk, snd_dtq, tsnd_dtq, rcv_dtq, trcv_dtq

② rel_wai, wup_tsk, sig_sem, set_flg, snd_mbx, rel_mpf, snd_dtq, psnd_dtq, tsnd_dtq

NORTi Oceans では、μITRON4.0 仕様にある他の２つの状態、強制待ち(SUSPENDED)、二重待ち(WAITINGSUSPENDED)、および、未登録状態へ戻る削除系のシステムコールが省かれています。

実行可能状態 (READY)

より優先度の高いタスクが実行中のため、実行を待たされている状態です。あるいは、同じ優先度のタスクが先に実行状態となっているため、実行を待たされている状態です。

実行状態 (RUNNING)

プロセッサを割り当てられて動作している状態です。RUNNING 状態のタスクは、同時にはひとつしか存在しません。タスクにとっては、READY 状態と RUNNING 状態には大差がなく、最優先 READY タスクの別名が RUNNING タスクともいえます。

待ち状態 (WAITING)

自ら発行したシステムコールにより、実行が止まっている状態です。時分割方式でなく事象駆

動(イベントドリブン)方式のマルチタスクでは、起動されたタスクは、ほどんどの期間をWAITING 状態で過ごします。そうでないと、優先度の高いタスクの待ちの間を利用して、優先度の低いタスクを実行させることができません。

WAITING 状態は、その要因によって次の様に分類されます。

起床待ち(slp_tsk, tslp_tsk)
時間待ち(dly_tsk)
イベントフラグ成立待ち(wai_flg, twai_flg)
セマフォ獲得待ち(wai_sem, twai_sem)
メールボックスでのメッセージ受信待ち(rcv_mbx, trcv_mbx)
データキューでのメッセージ送信待ち(snd_dtq, tsnd_dtq)
データキューでのメッセージ受信待ち(rcv_dtq, trcv_dtq)
固定長メモリブロック獲得待ち(get_mpf, tget_mpf)

休止状態（DORMANT）

DORMANT 状態は、タスクが起動されていない状態、あるいはタスクが終了した状態です。実行中のタスクが、自ら発行したシステムコールにより、DORMAT 状態になることもできますし、他タスクから強制的に DORMANT 状態にさせられることもできます。

未登録状態（NON-EXISTENT）

NON-EXISTENT 状態は、タスクが生成されていない状態です。

タスク切り替えの起きるタイミング

NORTi Oceans は、プリエンプティブなマルチタスク OS ですので、あるタスクの実行中に、それより優先度の高いタスクの実行が割り込みます。タスク切り替えの起きるタイミングとしては、次の 4 通りがあります。

(1) 実行中のタスクが、自分より高優先のタスクを起動、あるいは、待ち解除するようなシステムコールを発行した。
(2) 非タスクコンテキスト（割り込みハンドラ/ タイムイベントハンドラ） から、実行中タスクより高優先のタスクを起動、あるいは、待ち解除するようなシステムコールが発行された。
(3) 実行中タスクより高優先のタスクの待ち状態が、タイムアウトで解除された。
(4) 実行中のタスクが、自ら待ち状態に入った、優先度を下げた、あるいは、終了した。

逆に言えば、全てのシステムコールでタスク切り替えが起きるわけではありません。実行中タスクより優先度が低いか同じタスクに対して起動や待ち解除の操作を行っても、即座にはタスク切り替えは起きません。上記の(4)で、操作されたタスクが最優先となるまで、タスク切り替えは待たされます。

優先度が同じ場合でも、rot_rdq と chg_pri では、実行中のタスクが、実行待ち行列の末尾に回ることで、同一優先度間でのタスク切り替えが起きます。

1.3 用語

オブジェクトと ID

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びます。オブジェクトを識別するための番号でユーザーが指定できるものを ID 番号と呼びます。ID 番号を持ったオブジェクトには、タスク、セマフォ、イベントフラグ、メールボックス、固定長メモリプール、データキュー、周期ハンドラがあります。その他に、ID 番号ではなく割り込み番号で識別されるオブジェクトとして、割り込みハンドラがあります。

コンテキスト

直訳は「文脈」ですが、システム内でのある時点のタスクの実行環境全体をそのタスクのコンテキストと言います。そして、タスクが切り替えられる時に保存／復元される物の総称としてコンテキストという用語を使いますが、具体的には CPU のレジスタと読み代えても構いません。

非タスクコンテキスト

割り込みハンドラと、周期ハンドラとを合わせて、非タスクコンテキストと呼びます。非タスクコンテキストのハンドラはタスクでは無いため、自タスクを対象とするシステムコールを発行することはできません。

なお、μITRON 仕様では、非タスクコンテキスト専用システムコールの先頭文字を i として区別していますが、NORTi シリーズの場合、システムコール内部でコンテキストを自動判別していて区別はありません。i 付きのシステムコールは i 無しのシステムコールと同じ実装になっていますが、他社のμITRON との互換のためには、i 付きのシステムコールを使用することを推奨します。

ディスパッチ

実行タスクを選択して切り替えることを、ディスパッチと呼びます。システムコールにはディスパッチの発生するものとそうでないものがあります。ディスパッチを発生させるシステムコールでも、新しく、READY となったタスクの優先度が、現在の RUNNING タスクの優先度より低ければ、タスクは切り替わりません。また、非タスクコンテキストで発行されたシステムコールによるディスパッチは、タスクコンテキストへ復帰する時にまとめて行われます。これを遅延ディスパッチと呼びます。

同期・通信機能

同期機能は、タスク間で待ち合わせを行うために使われます。通信機能は、タスク間でデータを渡すために使われます。通信では同期も伴うため、同期・通信機能とまとめて表現しています。

同期・通信機能を使わなくても、プログラマが慎重に設計すれば、共通変数を介して、タスク間の待ち合わせやデータの受け渡しが可能です。しかし、OS の機能を使う方が、楽でかつ安全です。

NORTi Oceans では、セマフォ、イベントフラグ、メールボックス、データキューという 4 種類の、それぞれ特徴のある同期・通信の機構が設けられています。

待ち行列

1 つのオブジェクトに対して、複数のタスクが要求を出した場合は、待ちタスクの行列ができます。セマフォ獲得待ち、メールボックスのメッセージ受信待ち、固定長メモリプールのメモリブロック獲得待ち、データキューの送信待ち／受信待ちで待ち行列がつくられます。

待ち行列の並びは、先着順（FIFO:First In First Out）のみをサポートしています。

キューイング

相手のタスクが直ぐに受け取れなくともエラーとせずに、要求をとっておくことをキューイングと言います。

タスクの起床要求とメールボックス／データキューのメッセージはキューイングされます。起床要求のキューイングは、要求回数のカウントで実現されます。メールボックスでのメッセージのキューイングは、ポインタでつないだ線形リストで実現されます。データキューでのメッセージのキューイングはリングバッファで実現されます。

イベントフラグはビットパターンを保持しているのみで、すなわち、事象の有無のみが記録されて回数や順序は記録されない点がキューイングと異なります。

ポーリングとタイムアウト

待ちの生じるシステムコールには、待ちなし（ポーリング）の機能と、指定時間の経過で中断（タイムアウト）する機能とが用意されています。ポーリングの場合、待ちが必要ならば、エラーとなります。

パラメータとリターンパラメータ

μITRON 仕様では、ユーザー側から渡すデータをパラメータと呼び、システムコール側から返るデータをリターンパラメータと呼びますが、本書では C で一般的な引き数と表現しています。

システムコールの戻り値は原則としてエラーコードであるため、それ以外の値が返る場合は、これを格納する場所へのポインタを、引数として指定します。

システムコールとサービスコール

アプリケーションからカーネルやソフトウェア部品を呼び出すインタフェース(API)をサービスコールと呼びます。カーネルのサービスコールを、特にシステムコールと呼びます。

排他制御

マルチタスクでは、同時にアクセスしてはいけないものに、複数のタスクからアクセスできてしまいます。リエントラントでない関数や、共有データなど、同時利用不可なものはたくさんあります。これらの資源が同時に利用されないよう管理することを排他制御といい、一般的にはセマフォが使われます。

ただし、タスクの優先度が同一で、資源アクセス中に競合するタスクへの切り換えが行われないならば、排他制御の必要はありません（優先度の統一は、排他制御を不要にする有効な手段です）。実は、セマフォには、高優先度のタスクが、低優先度のタスクのセマフォ返却を待たなければならない優先度逆転というやっかいな問題がありますから、競合する区間の優先度を一時的に上げる方が良い場合があります。排他制御するべき区間が短いなら一時的なディスパッチ禁止や割り込み禁止により排他制御するのが簡単です。

アイドルタスク

アイドルタスクは、他の全てのタスクが止まっている時に実行されます。カーネル内部にもアイドルタスク部がありますが、ユーザーが、最低優先度で無限ループするタスクを作成すれば、それが、アイドルタスクとなります。

アイドルタスクは何も実行しないタスクですが、重要な意味を持っています。事象駆動（イベントドリブン）方式のマルチタスクで、アイドルタスクに実行順序が回らないということは、CPU のパフォーマンス不足、あるいは、無駄に CPU パワーを消費しているタスクの存在を示唆しています。

静的なエラーと動的なエラー

システムコールから返るエラーは、静的なエラーと動的なエラーとに分類できます。静的なエラーとは、範囲外の ID 番号使用等のパラメータの異常で、システムの状態に関わらず必ず起こり、デバッグが終われば無くなる種類のものです。動的なエラーとは、待ち解除しようとしたタスクがまだ、待ちに入っていなかったとかのように、システムの状態やタイミングに依存する種類のものです。ポーリング失敗のように、動的エラーを積極的に利用するプログラミングもおこなわれます。

NORTi では、高速化のために、静的なパラメータエラーをチェックしないライブラリも用意されています。

コンテキストエラー

システムコールには、非タスクコンテキスト（割り込みハンドラやタイムイベントハンドラ）から発行できないものがあります。これに違反した場合は、システムコールからコンテキストエラーが返ります。これは静的なエラーですので、静的なパラメータをチェックしないライブラリでは、コンテキストエラーを検出しません。

1.4 共通原則

システムコールの名称

ITRON のシステムコール名は、基本的に xxx_yyy 型をしています。xxx が操作方法の省略名で、yyy が操作対象の省略名です。xxx_yyy から派生したシステムコールは、先頭に 1 文字追加して、zxxx_yyy 型になります。ポーリングするシステムコールの先頭文字は"p"、タイムアウト有りのシステムコールの先頭文字は"t"、独自システムコールは"v"です。

データタイプの名称

ITRON のデータタイプ（型）の名称としては、すべて大文字を使用します。ポインタ型は、～P の名称とします。構造体の型は、原則として、T～の名称とします。

引き数の名称

システムコールの説明で、引き数の名称には次のような原則を設けています。

p_～	データを格納する場所へのポインタ
pk_～	パケット(構造体)へのポインタ
ppk_～	パケット(構造体)へのポインタを格納する場所へのポインタ
～id	ID
～no	番号
～atr	属性
～cd	コード
～sz	サイズ(バイト数)
～cnt	個数
～ptn	ビットパターン
i～	初期値

ゼロと負数の扱い

システムコールの入出力で、多くの場合、0 は特別な意味を持ちます。タスク ID を例に挙げると、0 で「自タスク」を指定します。自タスクとは、そのシステムコールを発行したタスクのことです。0 に特別な意味を持たせるため、ID 番号や優先度等は 1 から始まっています。また、ITRON 仕様で負の値は「システム」を意味します。システムコールのエラーコードは負の値となっています。

なお、μITRON3.0 仕様以前では、システム用として負の ID 番号(-1)～(-4)が予約されていましたが、μITRON4.0 仕様で廃止され、NORTi Oceans でも使用していません。

1.5 データタイプ

ITRON では、このように再定義した型を規定していますが、ユーザープログラムでの使用を強制するものではありません。システムコールで受け渡されるデータにのみ使用することでも構いません。

汎用的なデータタイプ

typedef signed char B;	符号付き 8 ビット整数
typedef unsigned char UB;	符号なし 8 ビット整数
typedef short H;	符号付き 16 ビット整数
typedef unsigned short UH;	符号なし 16 ビット整数
typedef long W;	符号付き 32 ビット整数
typedef unsigned long UW;	符号なし 32 ビット整数
typedef char VB;	タイプ不定データ(8 ビットサイズ)
typedef short VH;	タイプ不定データ(16 ビットサイズ)
typedef long VW;	タイプ不定データ(32 ビットサイズ)
typedef void *VP;	タイプ不定データへのポインタ
typedef void (*FP)();	プログラムのスタートアドレス一般

ITRON に依存した意味を持つデータタイプ

typedef int INT;	符号付き整数
typedef unsigned int UINT;	符号なし整数
typedef int BOOL;	ブール値(FALSE(0)または TRUE(1))
typedef INT FN;	関数コード
typedef int ID;	オブジェクトの ID 番号
typedef unsigned int ATR;	オブジェクト属性
typedef int ER;	エラーコード
typedef int PRI;	タスク優先度
typedef long TMO;	タイムアウト
typedef long DLYTIME;	遅延時間
typedef int ER_ID;	エラーコードまたはオブジェクト ID 番号
typedef unsigned int STAT;	オブジェクトの状態
typedef unsigned int MODE;	サービスコールの動作モード
typedef unsigned int ER_UINT;	エラーコードまたは符号なし整数
typedef unsigned int TEXPTN;	タスク例外パターン
typedef unsigned int FLGPTN;	イベントフラグビットパターン
typedef unsigned int INHNO;	割り込みハンドラ番号
typedef unsigned int INTNO;	割り込み番号
typedef VP VP_INT;	タスクパラメータおよび拡張情報
typedef unsigned long SIZE;	メモリ領域のサイズ

時間に関するデータタイプ

```
typedef struct t_systim ·····  システムクロックおよびシステム時刻
{
     H utime; ·····            上位 16bit
     UW ltime; ·····           下位 32bit
} SYSTIM;
typedef long RELTIM; ·····     相対時間
```

第２章 導入

2.1 インストール

インストールされた NORTi Oceans の標準的なフォルダ構成は、次の様になっています。XXX は MCU シリーズ名、BBB は評価ボード名、YYY は対応コンパイラ名(の略称)です。

/NORTi/NORTiOC/XXX/INC ･･･････インクルードファイル
/NORTi/NORTiOC/XXX/SMP/BBB ･･･サンプル
/NORTi/NORTiOC/XXX/LIB/YYY ･･･ライブラリ
/NORTi/NORTiOC/XXX/DOC ･･･････ドキュメント
ここで説明するファイル名の xxx の部分も、MCU に依存します。

インクルードファイル

INC フォルダには、次のヘッダファイルが収められています。

itron.h ･･･････････････ ITRON 標準ヘッダ
kernel.h ･･･････････････ カーネル標準ヘッダ
nocsys.h ･･･････････････ システム内部定義ヘッダ
noccfg.h ･･･････････････ コンフギュレーションヘッダ
nocrxxx.h ･･･････････････ CPU 差異定義ヘッダ
nocsio.h ･･･････････････ シリアル入出力関数ヘッダ
kernel.h は、NORTi Oceans を利用するすべてのソースファイルで#include してください。データタイプ、共通定数、関数プロトタイプ等、NORTi Oceans の機能を使用するために必要なすべての定義と宣言が記載されています。itron.h は、この kernel.h からインクルードされているので、ユーザーのソースファイルから#include する必要はありません。

noccfg.h には、最大タスク数等のコンフィグレーション用定数の標準値と、カーネル内部で使用する変数の実体が定義されています。ユーザープログラムの 1 つのファイルでのみ #include してください。

nocsys.h には、カーネルのすべての内部定義が記載されています。noccfg.h からインクルードされており、通常は、ユーザープログラムから#include する必要はありません。nocrxxx.h には、対応プロセッサによって異なる部分が定義されています。nocsys.h からインクルードされており、ユーザープログラムから#include する必要はありません。

ライブラリ

LIB フォルダには、NORTi Oceans のライブラリモジュールファイルが収められています。

n4exxx.lib ･･･････････････ パラメータチェック有りライブラリ
n4fxxx.lib ･･･････････････ パラメータチェック無しライブラリ
コンパイラによっては、ライブラリの拡張子が lib 以外の場合があります。

パラメータチェック無しライブラリとは、高速化のため、パラメータの静的なエラーチェックを省略したライブラリです。NORTi Oceans の SYSER 変数にエラーコードがセットされなくなったら、パラメータチェック無しライブラリに取り替えても良い目安となります。

サンプル

個々の MCU に依存する割り込み管理機能やデバイスドライバは、ライブラリに含まれません。サンプルとして付属しているソースファイルを(必要ならばカスタマイズして)、コンパイル、リンクしてしてください。

nocixxxx.c ・・・・・・・・・・・ **割り込み管理機能/周期タイマ割り込みハンドラソース**

nocsxxxx.c ・・・・・・・・・・・ **シリアル入出力ドライバソース**

nocsxxxx.h ・・・・・・・・・・・ **シリアル入出力ドライバヘッダ**

その他に、対応プロセッサの内蔵周辺機能のアドレスを定義したヘッダファイル、スタートアップルーチンの例、サンプルの main のソース、メイクやビルドファイル等が収められています。

2.2 カーネルコンフィグレーション

NORTi Oceans では、他のμITRON 仕様 OS のような面倒なコンフィグレーション手順はありません。ユーザープログラムのソースファイルの 1 つ、通常は、main 関数が含まれるファイルに、いくつかの#define と noccfg.h の#include を記述するだけで、コンフィグレーションは完了です。

ネットワーク等のソフトウェア部品を使用する場合には、ユーザープログラムで使用する ID 番号とソフトウェア部品が使用する ID 番号とが競合しないようにする必要があります。

標準値でのコンフィグレーション

次の様な標準値でよければ、#include ″noccfg.h″を記述するだけです。

タスク ID 上限 ····· 8
タイムイベントハンドラ ID 上限 ····· 1
他のオブジェクトの各 ID 上限 ····· 8
タスク優先度上限 ····· 8
割り込みハンドラのスタックサイズ ····· T_CTX 型の 4 倍サイズ(*1)
タイムイベントハンドラのスタックサイズ ·· T_CTX 型の 4 倍サイズ
システムメモリのサイズ ····· 0（スタック用メモリを使用）
メモリプール用メモリのサイズ ····· 0（スタック用メモリを使用）
スタック用メモリのサイズ ····· 0（デフォルトのスタックを使用）(*2)

(*1) T_CTX は、nocrxxx.h に定義されていて、通常そのサイズはスタックポインタ(SP)を除く CPU の全レジスタサイズの合計と同じです。(例外もあります)

(*2) デフォルトのスタックとは、通常、リンカで指定されるスタックセクションの先頭アドレスから、リセット時に SP に設定されるアドレスまでの領域を指します。

標準値以外でのコンフィグレーション

ID や優先度の上下限は、下記の通りです。

タスク ID / タイムイベントハンドラ ID ···· 1 ～ 253 (*3)
他のオブジェクトの ID ····· 1 ～ 999 (*4)
タスク優先度 ····· 1 ～ 14

(*3) この ID は、1 バイトで管理ており、255 と 254 は、内部で特別な意味に使われています。

(*4) その他 ID は、int で管理のためメモリ限界まで事実上無制限ですが、保証は 3 桁までとしています。

タスク優先度の上限については、なるべく小さな値を指定してください。優先度数が大きいと、最優先タスクを選ぶのに数命令ずつ余分な時間がかかります。

タスク優先度以外の定義では、上限を大きくしたことによる速度的なオーバヘッドはありません。ただし、ID 毎に内部でポインタを 1 個定義しますので、RAM 容量の少ないシステムでは、必要最小限にしてください。例を示します。

```
#define TSKID_MAX 16        タスク ID 上限
#define SEMID_MAX 4         セマフォ ID 上限
#define FLGID_MAX 5         イベントフラグ ID 上限
#define MBXID_MAX 3         メールボックス ID 上限
#define MPFID_MAX 3         固定長メモリプール ID 上限
#define DTQID_MAX 1         データキューID 上限
#define CYCNO_MAX 2         周期ハンドラ ID 上限
#define TPRI_MAX 4          タスク優先度上限
#include "noccfg.h"
```

タイマキューのサイズ

タイムアウトやタイムイベントハンドラを実現するために、2 種類タイマキューがあります。RAM に余裕がある場合は、各キューのサイズを 256 に変更してください。タイムアウト機能や時間管理機能の処理速度が大幅に改善されます。設定可能な値は、2 の階乗の数値（1,2,4,8,16,32,64,128,256）です。例を示します。

```
#define TMRQSZ 256          タスクのタイマキューサイズ
#define CYCQSZ 128          周期起動ハンドラのタイマキューサイズ
    :
#include "noccfg.h"
```

割り込みハンドラのスタックサイズ

割り込みハンドラのスタックサイズは、標準でコンテキスト型（T_CTX）の 4 倍サイズと定義されています。RAM 容量が不足する場合は、この値を慎重に削ってください。

割り込みハンドラのスタックは、システム初期化時に「スタック用メモリ」から動的に確保され、全ての割り込みハンドラで、このスタック領域を共有します。多重割り込みがあるならば、割り込みハンドラのスタックサイズに割り込みネストの分の追加が必要なことを考慮してください。例を示します。

```
#define ISTKSZ 400          割り込みハンドラのスタックサイズ
    :
#include "noccfg.h"
```

タイムイベントハンドラのスタックサイズ

タイムイベントハンドラ（周期起動ハンドラ）のスタックサイズは、標準でコンテキスト型（T_CTX）の 4 倍サイズと定義されています。RAM 容量が不足する場合は、この値を慎重に削ってください。

タイムイベントハンドラのスタックには、システム初期化時に main 関数が動作している「デフォルトのスタック」を使います。全てのタイムイベントハンドラで、このスタック領域を共有しますが、タイムイベントハンドラがネストすることはありません。

定義例を示します。

```
#define TSTKSZ 300          タイムイベントハンドラのスタックサイズ
    :
#include "noccfg.h"
```

システムメモリと管理ブロックのサイズ

タスクやセマフォやイベントフラグ等の管理ブロックは、全て、OS が用意する「システムメモリ」から動的に割り当てられます。次の表を元に必要なサイズを合計し、その値以上の数値を、システムメモリのサイズ SYSMSZ に定義してください。

オブジェクト	管理ブロックサイズ(1 オブジェクト毎)	
	計算式	ポインタ 32bit int 型 32bit (注 3)
タスク	sizoof(T_TCB)	40
セマフォ	sizeof(T_SEM)	4
イベントフラグ	sizeof(T_FLG)	8
メールボックス	sizeof(T_MBX)+ sizeof(void *) * 2 (注 1)	8+8
データキュー	sizeof(T_DTQ) (注 2)	24
固定長 メモリプール	sizeof(T_MPF)	16
周期ハンドラ	sizeof(T_CYC)	24

注 1：mprihd=NULL(行列ヘッダは管理ブロック内) の場合
注 2：dtqcnt=0、または、dtq!=NULL(バッファを指定) の場合
注 3: ポインタ 32bit,int 型 32bit プロセッサの場合(ColdFire 等)

システムメモリ使用量は生成するオブジェクト数で決まります。オブジェクト数の上限値の指定にかかわらず、実際に cre_???/acre_???システムコールで生成する分だけ確保すれば十分です。SYSMSZ の標準値は 0 で、この場合、「スタック用メモリ」からシステムメモリが割り当てられますので、スタック用メモリが十分にある場合、SYSMSZ を指定しない方が楽です。

定義例を示します。

```
#define SYSMSZ 2352          システムメモリのサイズ
    :
#include "noccfg.h"
```

メモリプール用メモリのサイズ

固定長メモリプールのメモリブロックは、OS が用意する「メモリプール用メモリ」から割り当てられます。アプリケーションに必要なサイズを定義してください。標準値は 0 で、この場合、「スタック用メモリ」からメモリプールが割り当てられますので、スタック用メモリが十分にある場合、MPLMSZ を指定しない方が楽です。

```
#define MPLMSZ 2048          メモリプール用メモリのサイズ
    :
#include "noccfg.h"
```

スタック用メモリのサイズ

cre_tsk でスタック領域を明示しない場合のタスクのスタックや、割り込みハンドラのスタックは、OS が用意する「スタック用メモリ」から割り当てられます。

さらに、SYSMSZ を 0 とした場合のシステムメモリ、MPLMSZ を 0 とした場合のメモリプール用メモリも、このスタック用メモリから割り当てられます。

スタック用メモリのサイズを定義する STKMSZ の標準値は 0 で、この場合、main 関数が使っている処理系のデフォルトのスタック領域（スタックセクション）を、OS のスタック用メモリとします。この場合の実際のスタックサイズは、リンカでのセクション設定とスタートアップルーチンでの初期スタックポインタ値で決まります。

なお、タイムイベントハンドラは、STKMSZ に 0 以外を定義した場合も、main 関数のスタックを引き継ぐために、処理系のデフォルトのスタック領域の方を使用します。

```
    :
#define STKMSZ 2048          スタック用メモリのサイズ
    :
#include "noccfg.h"
```

カーネルの割り込み禁止レベル

カーネル内部のクリティカルな区間では、一時的に割り込みを禁止しています。レベル割り込み機能のあるプロセッサでは、このカーネルの割り込み禁止レベルを選択できます。

システムコールを発行する割り込みハンドラの割り込み優先レベルは、カーネルの割り込み禁止レベル以下でなければなりません。カーネルの割り込み禁止レベルより高い優先度の割り込みハンドラは暴走の原因となりますので注意してください。

```
    :
#define KNL_LEVEL 6        カーネルの割り込み禁止レベル
    :
#include "noccfg.h"
```

ID の定義

μITRON 仕様では、ID を予め決めておく必要があります。全ての ID を#define してあるヘッダファイルを、ユーザープログラムの各ソースファイルから#include すればよいでしょう。

(例 1)

```
ーkernel_id.hー
#define ID_MainTsk 1
#define ID_KeyTsk 2
#define ID_ConSem 1
#define ID_KeyFlg 1
#define ID_ErrMbf 1
    :
```

```
ー各ソースー
#include "kernel.h"
#include "kernel_id.h"

    :
```

あるいは、ID をグローバル変数として定義すれば、ID 値が変更になった時に、全ファイルを再コンパイルしなくて済みます。

(例 2)

```
ーxxx_id.cー
#include "kernel.h"
ID ID_MainTsk = 1;
ID ID_KeyTsk = 2;
ID ID_ConSem = 1;
ID ID_KeyFlg = 1;
ID ID_ErrMbf = 1;
    :
```

```
ー各ソースー
#include "kernel.h"
extern ID ID_MainTsk;
extern ID ID_KeyTsk;
    :
```

ID の自動割り当て

acre_xxx システムコールによりオブジェクトを生成すると、空いていた ID 番号を戻り値として得ることができます。そのため、ID 番号を予め定義する必要がありません。この場合は、上記の(例 2)の様にグローバル変数として、ID 番号を参照すると良いでしょう。

空 ID 番号の検索は大きい方からですので、自動割り当てする ID 番号と、小さい方から#define した ID 番号との衝突が避けられます。

2.3 ユーザープログラムの作成例

2 つのタスクを使った簡単な例を挙げます。task1 の待ちを task2 が解除します。

```
#include "kernel.h"
#include "noccfg.h"

TASK task1(void)                        /* タスク 1 */
{
      FLGPTN ptn;
      for (;;) {
          tslp_tsk(100/MSEC)
          wai_sem(1);
          wai_sem(1);
          wai_flg(1, 0x01, TWF_ORW, &ptn);
      }
}

TASK task2(void)                        /* タスク 2 */
{
      for (;;) {
          wup_tsk(1);
          sig_sem(1);
          set_flg(1, 0x0001);
      }
}

const T_CTSK ctsk1 = {TA_HLNG, NULL, task1, 1, 512, NULL};
const T_CTSK ctsk2 = {TA_HLNG, NULL, task2, 2, 512, NULL};
const T_CSEM csem1 = {TA_TFIFO, 0, 1};
const T_CFLG cflg1 = {TA_CLR, 0};

void main(void)                         /* メイン */
{
      sysini();                         /* システム初期化 */
      cre_tsk(1, &ctsk1);               /* タスク 1 を生成 */
      cre_tsk(2, &ctsk2);               /* タスク 2 を生成 */
      cre_sem(1, &csem1);               /* セマフォ 1 を生成 */
      cre_flg(1, &cflg1);               /* イベントフラグ 1 を生成 */
      sta_tsk(1 ,0);                    /* タスク 1 を起動 */
      sta_tsk(2, 0);                    /* タスク 2 を起動 */
      intsta();                         /* 周期タイマ割り込み起動 */
      syssta();                         /* システム起動 */
}
```

第３章 タスクやハンドラの記述

システムを構成するソフトウェアは、OS 部分とユーザープログラム部分に分けることができます。一般にタスクはユーザープログラムに、ハンドラは OS 部分に分類されます。

この章では、ユーザーが記述しなければならないタスクとハンドラ類の具体的な記述形式を説明します。

3.1 タスクの記述

タスクの記述方法

タスクの記述に関しては次の 2 点を守るだけで、他は普通の C 関数と変わりません。

- TASK 型の関数とすること
- 引数は int 型か void とすること

タスクの記述例

終了するタイプのタスクは、次のように記述してください。

ext_tsk()は省略可能ですが、明示的にタスク終了を記述することを推奨します。

```
TASK task1(int stacd)
{
        :
        :
    ext_tsk();
}
```

繰り返すタイプのタスクは、次のように記述してください。

組込みシステムにおいて、ほとんどのタスクは、このタイプとなります。

```
TASK task1(int stacd)
{
    for (;;) {
        :
        :
    }
}
```

割り込みマスク状態

起動されたタスクは、割り込み許可状態です。

3.2 割り込みハンドラの記述

概要

μITRON 仕様の割り込み管理には、割り込みが発生すると、直接ユーザーの作成した「割り込みハンドラ」に制御が渡る方式と、一旦カーネル内で処理をおこなってからユーザーの作成した「割り込みサービスルーチンを呼び出す方式」があります。NORTi Oceans では割り込みハンドラだけをサポートしています。

割り込みハンドラでは、レジスタの待避と復元（NORTi Oceans では、ent_int と ret_int）をユーザーが記述する必要があります。

割り込みハンドラは、割り込み状態で実行されるため最小限の処理だけをおこなうようにし、後は割り込みを待っているタスクを起床して、実質的な割り込み処理を行わせるのが、一般的です。当然ですが、割り込み処理の中では、待ちとなるシステムコールを発行することはできません。また、動的なメモリ管理を伴うシステムコール（オブジェクトの生成）も発行できません。

割り込みハンドラの記述方法

割り込みハンドラの記述に関しては次の 2 点を守るとともに、普通の割り込みルーチン同様の配慮をおこなってください。

- **INTHDR 型の関数とすること**
- **ent_int で始め、ret_int システムコールで終了すること（カーネルの割り込み禁止レベルより高優先度の割り込みハンドラは除く）**

割り込みハンドラの記述例

```
INTHDR inthdr1(void)
{
      ent_int();
          :
          :
      ret_int();
}
```

ent_int システムコール

割り込みハンドラを全て C で記述できるようにするため、NORTi Oceans では独自の仕様として、割り込みハンドラの入口で呼ばれる ent_int システムコールを設けています。

ent_int では、レジスタの退避をおこなうと共に、スタックポインタを割り込みハンドラ専用のスタック領域に切り替えています。従って、各タスクのスタックには、割り込みハンドラが使う分を加算する必要がありません。

レジスタの多いプロセッサでは、ent_int で全レジスタを待避しません。コンパイラが待避せ

ずに使用するレジスタのみを待避します。残りのレジスタは、割り込み出口の ret_int システムコールで、ディスパッチが発生すると判断した時のみ待避されます。この処理により、ディスパッチが無い場合やネストして割り込んだ割り込みハンドラの処理時間を短縮しています。

ent_int 前の不要命令

ent_int システムコールの呼び出し前に、レジスタを破壊したりスタックポインタがずれるような命令が絶対に入ってはいけません。

第一の対策として、割り込みハンドラを含むソースファイルのコンパイルには、必ず最適化オプションを付けてください。デバッグオプションを付けてコンパイルすると、最適化が効かないコンパイラもありますので注意してください。

割り込みハンドラ関数の内容やコンパイラのバージョンやコンパイル条件によって、関数入り口で生成される不要な命令は変化するかもしれません。必ずアセンブルリストを出力させて、確認をおこなってください。RISC 系のプロセッサでは、ent_int だけでレジスタを待避できない場合があり、コンパイラが提供する interrupt 関数機能を使います。この場合には、ent_int の前にレジスタ待避命令が展開されるのが正常です。

auto 変数の禁止

割り込みハンドラ入り口で auto 変数を定義すると、スタックポインタが、ent_int 想定値よりずれてしまいます。static 変数とするか、割り込みハンドラからさらに関数を呼んで、そこに auto 変数を定義してください。ただし、auto 変数がスタック上ではなくレジスタ変数となることが分かっている場合は、auto 変数を使うことが可能です。

割り込みハンドラ関数で複雑な処理をおこなうと、やはり ent_int の前に予期しない命令が展開される場合があります。その場合も、割り込みハンドラからさらに関数を呼んで、そこで実際の処理をおこなってください。

インライン展開の抑制

割り込みハンドラからさらに関数を呼ぶように記述しても、コンパイラの最適化により、その関数が割り込みハンドラ内にインライン展開されてしまう場合があります。この場合は、インライン展開を禁止するオプションを付けてコンパイルしてください。

部分的なアセンブラによる記述

どうしても、ent_int 前の不要命令が抑制できない場合は、割り込みハンドラの入口と出口のみをアセンブラで記述し、そこから本体 C 関数を呼んでください。(アセンブラの展開方法は、個別の補足説明書を参照)

インラインアセンブラが使える場合は、それで不要命令をキャンセルする方法も考えられます。例えば生成されてしまった push 命令を、インラインアセンブラの pop 命令で打ち消す等です。

割り込みマスク状態

割り込み禁止／許可の 2 値しかない CPU の場合、起動された割り込みハンドラは、割り込み禁

止状態です。多重割り込みを許す場合は、割り込みコントローラの操作により処理中の割り込みをマスクした上で、直接 CPU の割り込みマスクを変更して、割り込み許可にできます。

レベル割り込み機能を持った CPU の場合、ent_int()から復帰後の割り込みハンドラのレベルは、ハードウェアの割り込みのレベルと一致しています。より優先度の高い割り込みが発生した場合は、多重割り込みが受け付けられます。

3.3 タイムイベントハンドラの記述

概要

μITRON 4.0 仕様のタイムイベントハンドラには、繰り返し実行される周期ハンドラと、1 度だけ実行されるアラームハンドラ、指定タスクが指定した時間を超えて実行された場合に実行されるオーバーランハンドラの 3 種類があります。NORTi Oceans では周期ハンドラだけをサポートします。

周期ハンドラは、非タスクコンテキストとしてタスクより優先的に実行されるので、タスクと比較して、より正確な時間による制御が可能です。また、タスクと比較して管理ブロックやスタックに必要なメモリはより少なくて済むというメリットがあります。

ただし、周期ハンドラの中では、待ちを発生するシステムコールを発行することはできません。

周期ハンドラの記述方法

周期ハンドラの記述に関しては普通の割り込みルーチン同様の配慮をおこなってください。周期ハンドラは以下の様な C 関数として記述してください。exinf は周期ハンドラ生成時に指定した拡張情報です。

```
void cychdr(VP_INT exinf)
{
        :
        :
}
```

割り込みマスク状態

全ての周期ハンドラの処理が終わるまで、システムはディスパッチ禁止状態となっていますが、割り込みは許可状態です。

周期ハンドラ内で割り込み禁止にした場合は、必ず割り込み許可状態に戻してからリターンするようにしてください。

補足

周期ハンドラは、割り込みハンドラの次に優先的に実行されますので、処理は十分に短くし、なるべく最適化コンパイルをおこなってください。

なお、割り込みハンドラと異なり、auto 変数は自由に使用できます。

3.4 初期化ハンドラ

ITRON 仕様書では、システムの初期化方法については、処理系に依存するということで触れられていません。したがって本節の内容は、NORTi Oceans 独自のものです。

スタートアップルーチン

他のμITRON 仕様 OS の中には、専用のスタートアップルーチンを用意して、マルチタスクに必要な初期化をおこなった後、main 関数をタスクとして起動するタイプのものがあります。一方、NORTi Oceans では、特別なスタートアップルーチンを設けず、main 関数までは、通常のプログラムと同じように実行されます。

main 関数

NORTi Oceans では、main 関数をマルチタスクの初期化ハンドラとして位置づけています。main 関数では、システム初期化 sysini、I/O 等の初期化、1 個以上のタスク生成 cre_tsk、1 個以上のタスク起動 sta_tsk 、必要ならセマフォやイベントフラグ等のオブジェクトの生成 cre_xxx 、周期タイマ割り込みの起動とシステム起動 syssta 等をおこないます。

システム初期化

main 関数の先頭で sysini 関数を実行して、カーネルを初期化します。sysini からは、機種依存する割り込み関係の初期化をおこなうため intini 関数が呼び出されます。標準的な intini 関数は nocixxx.c に収められていますが、ユーザーのシステムに適合しない場合は、独自に作成してください。

I/O の初期化

マルチタスク動作の前に初期化しておきたい I/O 等が有る場合は、main 関数でそれらの初期化をおこないます。

オブジェクトの生成

タスクやセマフォやイベントフラグ等のオブジェクト生成は、main 関数でおこなっても、タスクでおこなっても構いません。ただし、オブジェクト生成にはメモリ管理が伴うため、リアルタイム性に劣ります。オブジェクトの生成は、なるべく main 関数でまとめて実行する方がよいでしょう。

タスクの起動

起動すべき全タスクを main 関数で起動しても構いません。1 つだけ（いわゆるメインタスク）のみを起動して、そのタスクで残るタスクを起動しても構いません。

周期タイマ割り込み起動

標準的には、intsta 関数で周期タイマ割り込みを起動します。機種依存する周期タイマ割り込み、および割り込み管理関係のモジュールは、ライブラリに含まれていませんので、付属の nocixxx.c をコンパイルしてリンクする必要があります。

システム起動

syssta 関数を実行すると、いよいよマルチタスク動作がスタートします。そして、syssta 関数は main 関数に戻って来ず、内部で無限ループします（この部分が、デフォルトのアイドルタスクになります）。ただし、syssta 関数を実行する前に呼ばれた cre_tsk や sta_tsk でエラーがあった場合は、マルチタスク動作をスタートせずに main 関数へ戻って来ます。

初期化ハンドラの記述例

```
#include "kernel.h"

/* コンフィグレーション */
#define TSKID_MAX     2               /*タスク ID 上限*/
#define SEMID_MAX     1               /*セマフォ ID 上限*/
#define FLGID_MAX     1               /*イベントフラグ ID 上限*/
#define TPRI_MAX      4               /*タスク優先度上限*/
#define TMRQSZ        256             /*タスクのタイマキューサイズ*/
#define ISTKSZ        256             /*割り込みハンドラのスタックサイズ*/
#define TSTKSZ        256             /*タイムイベントハンドラのスタックサイズ*/
#define SYSMSZ        256             /*システムメモリのサイズ*/
#define KNL_LEVEL     5               /*カーネルの割り込み禁止レベル*/
#include “noccfg.h”

/* ID の定義 */
#define ID_MainTSK    1
#define ID_KeyTsk     2
#define ID_ComSem     1
#define ID_KeyFlg     1

/* オブジェクト生成情報 */
extern TASK MainTsk(void);
extern TASK KeyTsk(void);
const T_CTSK ctsk1 = { TA_HLNG, NULL, task1, 1, 512, NULL };
const T_CTSK ctsk2 = { TA_HLNG, NULL, task2, 2, 512, NULL };
const T_CSEM csem1 = { TA_TFIFO, 0, 1 };
const T_CFLG cflg1 = { TA_CLR, 0 };

/* メイン（初期化ハンドラ） */
int main(void)
{
    sysini();                       /* システム初期化 */
    cre_tsk(ID_MainTsk, &ctsk1);    /* タスク生成 */
    cre_tsk(ID_KeyTsk, &ctsk2);     /* タスク生成 */
    cre_sem(ID_ConSem, &csem1);     /* セマフォ生成 */
    cre_flg(ID_KeyFlg, &cflg1);     /* イベントフラグ生成 */
    sta_tsk(ID_MainTsk, 0);         /* タスク起動 */
    intsta();                       /* 周期タイマ割り込みを起動 */
    return syssta();                /* マルチタスクへ移行 */
}
```

第４章 機能概説

4.1 タスク管理機能

概要

タスク生成は cre_tsk により、タスク起動は sta_tsk または act_tsk によりおこないます。act_tsk を使用した場合、指定タスクが既に ready 状態であれば起動要求がキューイングされます。タスク終了は、自タスクを終了する ext_tsk、他タスクを終了させる ter_tsk とに分かれています。起動要求がキューイングされているタスクを終了した場合、直ちに再起動されます。キューイングされている起動要求をキャンセルするには can_act を使用します。

優先度を変更する chg_pri、優先度を参照する get_pri、その他、タスクの状態を見る ref_tsk とその簡易版の ref_tst システムコールが、タスク管理機能に分類されています。

タスク管理ブロック

タスクの管理は、タスク管理ブロック（TCB）と呼ばれるデータテーブルの情報に基づいておこなわれています。

μITRON 仕様では、ユーザーが TCB やその他の管理ブロックに直接アクセスする方法は提供されていません。NORTi Oceans では、nocsys.h を#include すると、TCB 等に直接アクセスすることが可能ですが、TCB 等の構造はバージョンアップで変更される可能性があります。

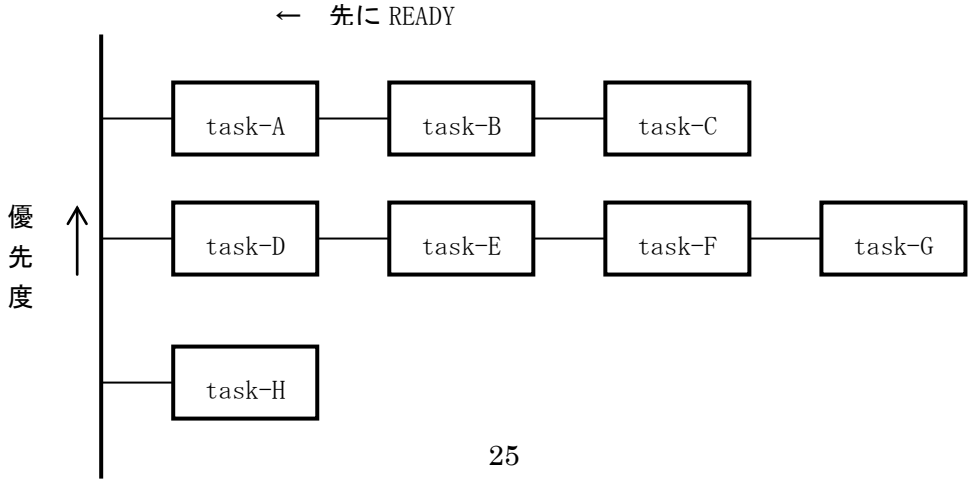
スケジューリングとレディキュー

タスクの実行順序を決めることをスケジューリングと言います。μITRON では、優先度ベースのスケジューリングがおこなわれます。

実行順序を管理する変数をレディキューと呼びます。レディキューには、優先度順で（同じ優先度なら先に READY になった順で）タスクがつながれています。

最優先の READY タスクが RUNNING 状態のタスクです（下図では task-A）。

このタスクが WAITING や DORMANT 状態になると、レディキューから外れ、次に優先度の高いタスク（下図では task-B）が、RUNNING 状態となります。



4.2 タスク付属同期機能

概要

タスク付属同期機能に分類されるのは、slp_tsk,tslp_tsk,wup_tsk,can_wup,rel_wai,dly_tsk システムコールです。

待ちと解除

タスクが自ら待ち状態 WAITING に移行するシステムコールとして、slp_tsk と tslp_tsk があります。tslp_tsk ではタイムアウト時間を指定できます。

すなわち単純な時間待ちにも利用できますが、基本的には単純な時間待ちには dly_tsk を使うべきです。

tslp_tsk は、指定時間経過後 E_TMOUT を返しますが、dly_tsk は E_OK を返します。tslp_tsk が E_OK を返すのは wup_tsk された場合です。

wup_tsk はキューイング機能があるため、tslp_tsk を呼び出す前に wup_tsk されていると、タスクは WAITING 状態に入らずに直ちに E_OK を返します。したがって、tslp_tsk ではタスクが指定された時間 WAITING するとは限りません。

なお、slp_tsk,tslp_tsk の他、wai_flg,wai_sem,rcv_mbx 等のシステムコールでも、待ち状態 WAITING へ移行します。これらの待ち状態にあるタスクに対しては、wup_tsk ではなく rel_wai を発行することにより、強制的に待ちを解除することができます。

4.3 同期・通信機能（セマフォ）

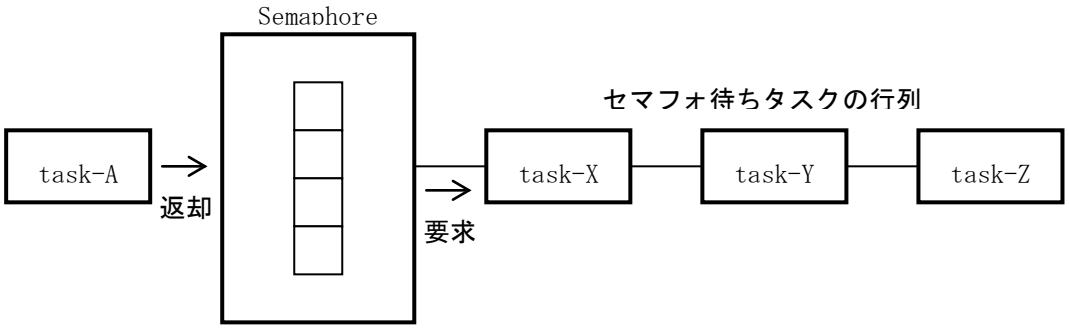
概要

セマフォは、資源の排他制御に使用します。非同期に動作する複数のタスクが、同時利用不可の資源（関数やデータや入出力等々）を共有する場合は、セマフォで資源の獲得と返却をおこなって排他制御する必要があります。セマフォは、排他制御すべき資源ごとに設けます。

セマフォの生成は cre_sem、acre_sem でおこないます。資源の返却をおこなう sig_sem に対し、資源の獲得待ちをおこなう wai_sem、待たずにポーリングをおこなう pol_sem、タイムアウト付きで待つ twai_sem があります。その他に、セマフォの状態を参照する ref_sem システムコールがあります。

セマフォ待ち行列

複数のタスクが同じセマフォを待つことができ、先に要求した順番で待ち行列をつくります。



セマフォのカウント値

sig_sem を実行した時、セマフォを待っているタスクが有る場合は、待ち行列の先頭のタスクを READY 状態にします。待ちタスクが無い場合は、セマフォのカウント値を+1 します。

wai_sem を実行した時、セマフォのカウント値が 1 以上の場合、カウント値を-1 して、タスクは実行を継続します。カウント値が 0 の場合、タスクは WAITING 状態になります。

一般的な用途ではセマフォカウントは 0 と 1 だけで十分ですから、セマフォ生成時に、セマフォ最大値 1 を指定することを推奨します。

4.4 同期・通信機能（イベントフラグ）

概要

イベントフラグは、事象の有無だけを相手のタスクに知らせたい場合に使用します。

イベントフラグの生成は cre_flg、acre_flg でおこないます。イベントフラグをセットする set_flg に対し、イベントフラグ成立を待つ wai_flg、待たずにポーリングする pol_flg、タイムアウト付きで待つ twai_flg があります。その他に、イベントフラグをクリアする clr_flg 、イベントフラグの状態を参照する ref_flg システムコールがあります。

イベントフラグ待ち行列

同じイベントフラグを、同時に複数のタスクが待つことはできません。したがって待ち行列も生成されません。

待ちモード

イベントフラグでは、複数ビットのフラグ群を用いていますので、待ち条件をビットパターンの AND, OR で指定することができます。AND 待ちでは、パラメータで指定したビットのすべてが、イベントフラグ上にセットされるのを待ちます。OR 待ちでは、指定したビットのいずれかが、イベントフラグ上にセットされるのを待ちます。

クリア指定

wai_flg、pol_flg、twai_flg システムコールでは、パラメータの指定により、イベントフラグが成立した時、自動的にイベントフラグをクリアすることができます。生成時にクリア指定をした場合は常にクリアされます。

クリアは全てのビットに対しておこなわれます。

4.5 同期・通信機能（データキュー）

概要

データキューは、リングバッファで実装されたメールボックスです。バッファを使用するため送信時にも待ちが発生します。

データキューの生成は cre_dtq、acre_dtq でおこないます。データを送信する snd_dtq、ポーリングで送信を試みる psnd_dtq、タイムアウト付きで送信をおこなう tsnd_dtq に対し、メッセージの受信待ちをおこなう rcv_dtq、待たずにポーリングをおこなう prcv_dtq、タイムアウト付きで待つ trcv_dtq があります。また強制的にデータを送信する fsnd_dtq があります。その他に、データキューの状態を参照する ref_dtq システムコールがあります。

待ち行列

データキューは、送信待ち行列、受信待ち行列、リングバッファから構成されます。送信時にバッファがフルの場合、送信しようとしたタスクは、データがバッファから取り除かれるまで送信待ち行列につながれます。受信時にバッファが空の場合、受信しようとしたタスクはデータが送信されるまで受信待ち行列につながれます。

リングバッファサイズを 0 にすることもできます。この場合、送信タスクと受信タスクはお互いに待ちあうことになり同期を取ることができます。

送信および受信待ち行列は常に到着順になります。

データ順

データに優先度を付けることはできません。しかし、fsnd_dtq を使用することで snd_dtq で送信されたデータより先に受信されることがあります。

fsnd_dtq により送信した時、バッファフルの場合にはバッファの先頭のデータを抹消してそこにデータを格納します。

4.6 同期・通信機能（メールボックス）

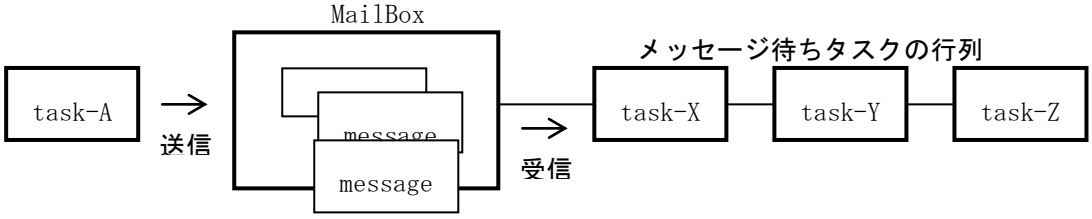
概要

メールボックスはタスク間での比較的大きなデータの受け渡しに使用します。実際に送信されるのは、メッセージと呼ばれるデータパケットへのポインタだけで、メッセージの内容そのものがコピーされる訳ではありません。コピーされないので、メッセージサイズに依存せず高速です。また、ユーザー領域の送信メッセージを線形リスト化し、管理するため送信待ちが発生しません。メールボックスにおける待ち行列は、メッセージ行列と受信待ちタスク行列です。

メールボックスの生成は cre_mbx、acre_mbx でおこないます。メッセージを送信する snd_mbx に対し、メッセージの受信待ちをおこなう rcv_mbx 、待たずにポーリングをおこなう prcv_mbx 、タイムアウト付きで待つ trcv_mbx があります。その他に、メールボックスの状態を参照する ref_mbx システムコールがあります。

メッセージ待ち行列

複数のタスクが同じメールボックスで待つことができ、先に要求した順番で待ち行列をつくります。



この図には両方が描かれていますが、メッセージ待ちタスクとキューイングされたメッセージが同時に存在することはありません。

メッセージキュー

メッセージは、受信タスクの有無にかかわらず、随時送信することができます。メッセージパケットの先頭部分が、次のメッセージを指すポインタとして使われます。したがって、ROM 上のデータをメッセージパケットとすることができません。メッセージは送信された順番で行列をつくります。

メッセージパケット領域

メッセージはいつ受信タスクに引き取られるか分かりません。したがって、メッセージパケットを auto 変数にとることは危険です。また、メッセージ領域を静的に定義しても、空いたかのチェックをおこなって再利用するのは面倒です。まだキューイングされているメッセージを再度送信した場合のシステム動作は保証できません。そこで、通常は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パケットに用います。

メールボックスは、メッセージパケットのサイズを関知しません。すなわち、メッセージ長

に制限はありません。しかし、固定長メモリプールのみをサポートするNORTi Oceansでは、必然的にメッセージパケットのサイズが使用しているメモリブロックのサイズに固定されます。

4.7 割り込み管理機能

概要

割り込み管理機能に分類されるのは、chg_ims、get_ims、ent_int、ret_int システムコールです。def_inh、dis_int、ena_int は機種依存（ユーザーカスタマイズ）システムコールです。

割り込みハンドラの定義

割り込みハンドラを定義する def_inh システムコールでは、割り込みベクタの設定をおこないます。しかし、割り込みの設定方法はシステムにより異なりますので、カーネルにはこのシステムコールを含めていません。付属の nocixxx.c に定義されているこのシステムコールが適合しない場合は、ユーザー側で、独自の機能を設計してください。

特定の割り込みの禁止／許可

μITRON 仕様にある、特定の割り込みを禁止／許可する dis_int、ena_int システムコールは、完全に機種依存しますので、NORTi Oceans では、ほとんどのプロセッサに対してサポートしていません。（汎用的な dis_int、ena_int が作成可能なプロセッサでは、サンプルに含まれている場合があります。）

割り込みハンドラの起動

割り込みハンドラより先に、カーネルが割り込みをフックすることはしていません。直接、ユーザーの記述した割り込みハンドラへ飛んできます。

そして NORTi Oceans では、割り込みハンドラを全て C で記述できるようにするため、独自の仕様として、割り込みハンドラの先頭で呼ばれる ent_int システムコールを設けています。ent_int では、レジスタの退避を行うと共に、スタックポインタを割り込みハンドラ専用のスタック領域に切り替えています。

RISC プロセッサの割り込み

ARM、MIPS、PowerPC、SH-3/4 等の RISC 系プロセッサでは、全ての外部割り込みが一ヶ所のエントリを共有しています。この場合、def_inh システムコールでは、割り込みベクタテーブルの代わりに、nocixxx.c に定義してある配列へ、割り込みハンドラのアドレスを設定するようにしています。そして、割り込み要因を判別し、この配列を参照してジャンプするプログラムが、vecxxx.asm にサンプルとして記述されています。したがって、RISC 系のプロセッサでも、割り込みベクタテーブルがあるかのごとく、プログラミングすることが可能です。システムコールを発行しない、カーネルの割り込み禁止レベルより高優先度の割り込みルーチンは ent_int, ret_int が不要であることは CISC プロセッサと同様です。

カーネルより高優先の割り込みルーチン

カーネルの割り込み禁止レベルより高いレベルの割り込みルーチンを使うことができます。この割り込みルーチンにとって、カーネル内部の割り込み禁止区間は割り込み許可と同じことになり、非常に高速な割り込み応答が要求されるシステムでも NORTi Oceans を応用することができるようになります。

ただし、カーネルより高優先の割り込みルーチンでは、システムコールを発行できません。割り込みの入り口と出口のレジスタ待避・復元も、ent_int()と ret_int()ではなく、コンパイラが interrupt 関数として提供する機能か、あるいは、独自にアセンブラで行ってください。

カーネルより高優先の割り込みルーチンでは、タスクとの同期や通信を行うことができません。一連の割り込みの区切りでタスクと同期・通信すれば良い場合、高優先の割り込みルーチンからカーネルのレベル以下の割り込みハンドラを起動して、そこでシステムコールを発行するテクニックを使ってください。

4.8 メモリプール管理機能

概要

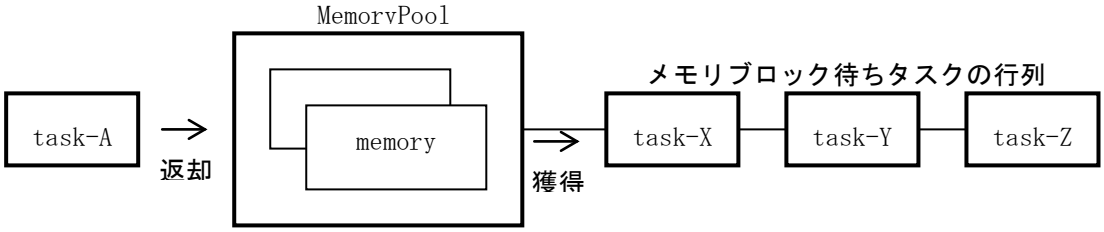
メモリ管理機能は、固定長のメモリブロックの機能を提供します。タスクは、メモリが必要になるとメモリプールからメモリブロックを獲得し、そのメモリが不要になると同じメモリプールへメモリブロックを返却するようにプログラムしてください。タスク間で共有されるメモリ領域は、メモリプールと呼ばれる単位で管理され、1 つのメモリプールは、複数のメモリブロックから構成されます。

メモリプールの機能は、C の標準ライブラリの malloc/free 関数と似ています。malloc/free 関数との違いは、メモリを解放した時に他タスクのメモリ獲得待ちを解除することやリエントラントであることなどの、マルチタスクに適した機能が備わっていることです。

メモリプールの生成は cre_mpf, acre_mpf でおこないます。メモリブロックを返却する rel_mpf に対し、メモリブロックの獲得待ちをおこなう get_mpf、待たずにポーリングをおこなう pget_mpf、タイムアウト付きで待つ tget_mpf があります。その他に、固定長メモリプールの状態を参照する ref_mpf システムコールがあります。

メモリブロック待ち行列

複数のタスクが同じメモリプールで待つことができ、先に要求した順番で、待ち行列をつくります。



この図には両方が描かれていますが、固定長メモリプールでは、メモリブロック待ちタスクとメモリブロックが同時に存在することはありません。

メッセージ送受信との組み合わせ

一般的に、メールボックス機能のメッセージパケット領域にはメモリプールのメモリブロックを利用します。メッセージ送信側でメモリブロックを獲得し、メッセージ受信側でそのメモリブロックを返却するようにプログラムしてください。

複数のメモリプール

用途別に複数のメモリプールを設けることを推奨します。1 つだけのメモリプールを様々なタスクから使うと、メモリプールを使い切った時にデッドロックの恐れがあります。すなわち、1 個所の遅れがシステム全体に波及して処理が破綻する場合があり得ます。

例えば、タスク A とタスク B とタスク C が、メモリプールを組み合わせたメッセージ送受信により協調して動作する場合で説明します。処理の流れとしては、タスク A が指令メッセージをタスク B へ送り、それを受けたタスク B がさらに指令メッセージをタスク C へ送り、それを受けたタスク C が、返答メッセージをタスク B に送り返す場合を考えます。もし、タスク C の処理が遅いと、タスク A から B へのメッセージが次々とキューイングされ、やがてメモリブロックを使い果たします。すると処理の終わったタスク C は返答メッセージを返すためのメモリブロックが獲得できなくなり、この返答を待つタスク B も永久に止ってしまいます。

一方、用途別にメモリプールを分ければ、メモリプールが空になるのを積極的に利用して、処理のキューイング数を制御することができます。

4.9 時間管理機能

概要

時間管理機能に分類されるのは、set_tim、get_tim、cre_cyc、acre_cyc、sta_cyc、stp_cyc、ref_cyc、isig_tim システムコールです。

システム時刻とシステムクロック

システムクロックはシステム起動時に 0 クリアされ、以後周期割り込みごとにカウントアップされます。

システム時刻は set_tim により任意の値に初期化され、以後周期割り込みごとにカウントアップされます。このシステム時刻値は、get_tim システムコールで読み出すことができます。set_tim するまでシステム時刻は不定です。

タイムイベントハンドラはシステムクロックをベースに起動されます。したがってシステム時刻を変更してもすでに設定してある動作に影響はありません。

システムコール内部での乗除算のオーバヘッドを避けるため、時間の単位はタイマ割り込み周期を使用しています。

周期ハンドラ

指定した時間間隔で起動実行されるタイムイベントハンドラです。時間的な正確さが要求されるデータのサンプリングや、rot_rdq 発行によるラウンドロビンケジューリング等に応用できます。

周期ハンドラは、cre_cyc、acre_cyc システムコールにより登録します。その他、ハンドラの活性制御をおこなう sta_cyc、stp_cyc、活性状態や次の起動までの時間を調べる ref_cyc システムコールがあります。

4.10 システム状態管理機能

概要

システム状態管理機能はシステムの状態参照／変更をするための機能で、レディキューを回転するための rot_rdq、自タスクのタスク ID を得るための get_tid、vget_tid、CPU をロック／アンロックするための loc_cpu、unl_cpu、ディスパッチを禁止／許可するための dis_dsp、ena_dsp、システム状態を参照するための sns_loc、sns_ctx、sns_dsp、sns_dpn、ref_sys が含まれます。

タスクの実行順制御

ディスパッチ禁止 dis_dsp と許可 ena_dsp により、複数のシステムコールを発行した後でまとめてタスク切り替えをおこなうことができます。レディキューを回転する rot_rdq により、同一優先度のタスクに実行権を渡したり、ラウンドロビンのような実行順制御が可能になります。CPU をロックすることで一時的に割り込みを禁止することもできます。

4.11 システム構成管理機能

システム管理機能に分類されるシステムコールは、OS のバージョンを得る ref_ver、コンフィグレーション情報を参照する ref_cfg です。

4.12 未サポート機能

μITRON4.0 フル仕様の NORTi Version 4 (CPU 例外ハンドラ定義 def_exc を除く、ほぼ全てのシステムコールを実装)と比較した場合、以下の機能／システムコールが省かれています。

オブジェクト削除 (del_tsk, exd_tsk, del_sem, del_flg, del_dtq, del_mbx, del_mpf, del_cyc)
タスク強制待ち (sus_tsk, rsm_tsk, frsm_tsk)
タスク例外処理 (def_tex, ras_tex, iras_tex, dis_tex, ena_tex, sns_tex, ref_tex)
ミューテックス (cre_mtx, acre_mtx, del_mtx, unl_mtx, loc_mtx, pol_mtx, tloc_mtx, ref_mtx)
メッセージバッファ (cre_mbf, acre_mbf, del_mbf, snd_mbf, psnd_mbf, tsnd_mbf, rcv_mbf, prcv_mbf, trcv_mbf, ref_mbf)
ランデブ用ポート (cre_por, acre_por, del_por, cal_por, tcal_por, acp_por, pacp_por, tacp_por, fwd_por, rpl_rdv, ref_por, ref_rdv)
可変長メモリプール (cre_mpl, acre_mpl, del_mpl, get_mpl, pget_mpl, tget_mpl rel_mpl, ref_mpl)
アラームハンドラ (cre_alm, acre_alm, del_alm, sta_alm, stp_alm, ref_alm)
オーバーランハンドラ (def_ovr, sta_ovr, stp_ovr, ref_ovr)
割り込みサービスルーチン (cre_isr, acre_isr, del_isr, ref_isr)
特定割り込みの禁止・許可 (dis_int, ena_int)
拡張サービスコール (def_svc, cal_svc)
NORTi3 (μITRON3.0) 互換モード

※ 次ページ以降のエラーの分類表記について

次章のシステムコール解説で、戻値に記載されている*と**マークは、次の様なエラーの分類を示します。

* 標準のライブラリでのみ検出される静的なエラーで、SYSER 変数に記録される。

** パラメータチェック無しライブラリでも検出され、SYSER 変数に記録される。

マーク無しのエラーは、いずれのライブラリでも検出されますが、SYSER 変数へは記録されません。

第５章 システムコール解説

5.1 タスク管理機能

cre_tsk

機能 タスク生成

形式 ER cre_tsk(ID tskid, const T_CTSK *pk_ctsk);
tskid タスク ID
pk_ctsk タスク生成情報パケットへのポインタ

解説 tskid で指定されたタスクを生成します。すなわち、システムメモリから、タスク管理ブロック(TCB)を動的に割り当てます。タスク生成情報パケットのスタック領域先頭番地(stk)が NULL の場合にスタック用メモリから、スタック領域を動的に確保します。生成した結果、対象タスクは NON-EXISTENT 状態から DORMANT 状態へ遷移します。

タスク生成情報パケットの構造は次の通りです。

```
typedef struct t_ctsk {
    ATR tskatr;             タスク属性
    VP_INT exinf;           拡張情報
    FP task;                タスクとする関数へのポインタ
    PRI itskpri;            タスク起動時優先度
    SIZE stksz;             スタックサイズ(バイト数)
    VP stk;                 スタック領域先頭番地
    const char *name;       タスク名へのポインタ（省略可）
} T_CTSK;
```

exinf の値は act_tsk によるタスク起動時にタスクパラメータとしてタスクに渡されます。exinf は ref_tsk で参照できます。

tskatr には、タスクが高級言語で記述されていることを示す TA_HLNG を入れてください。また、タスク生成後 DORMANT 状態から READY 状態とする場合は TA_ACT を入れてください。

name には、タスク名文字列を入れてください。対応デバッガ用で OS が使用することはありません。名前を指定しない場合には""か NULL を入れてください。T_CTSK 構造体を初期値付きで定義する場合には、name を省略しても構いません。

スタック領域をユーザープログラム内に確保した場合は、その先頭番地を stk に、サイズを stksz にそれぞれ設定してください。

戻値

E_OK	正常終了
E_PAR	優先度が範囲外*
E_ID	タスク ID が範囲外*
E_OBJ	タスクが既に生成されている
E_CTX	割り込みハンドラから発行*
E_SYS	管理ブロック用のメモリが確保できない**
E_NOMEM	スタック用のメモリが確保できない**

例

```
#define ID_task2 2
const T_CTSK ctsk2 = {TA_HLNG, NULL, task2, 8, 512, NULL};
TASK task1(void)
{
    ER ercd;
        :
    ercd = cre_tsk(ID_task2, &ctsk2);
        :
}
```

acre_tsk

機能　タスク生成(ID 自動割り当て)

形式　ER_ID acre_tsk(const T_CTSK *pk_ctsk);
pk_ctsk　タスク生成情報パケットへのポインタ

解説　未生成タスクの ID を、大きな方から検索して割り当てます。タスク ID が割り当てられない場合は、E_NOID エラーを返します。それ以外は、cre_tsk と同じです。

戻値　正の値　割り当てられたタスク ID
E_PAR　優先度が範囲外*
E_NOID　タスク ID が不足
E_CTX　割り込みハンドラから発行*
E_SYS　管理ブロック用のメモリが確保できない**
E_NOMEM　スタック用のメモリが確保できない**

例
```
ID ID_task2;
const T_CTSK ctsk2 = {TA_HLNG, NULL, task2, 8, 512, NULL};
TASK task1(void)
{
    ER_ID ercd;
        :
    ercd = acre_tsk(&ctsk2);
    if (ercd > 0)
        ID_task2 = ercd;
        :
}
```

act_tsk, iact_tsk

機能 タスク起動

形式
```
ER act_tsk(ID tskid);
ER iact_tsk(ID tskid);
tskid       タスク ID
```

解説 tskid で指定されたタスクを起動します。iact_tsk はμITRON 仕様と互換性を取るためのマクロによる act_tsk の再定義です。対象タスクは DORMANT 状態から READY 状態へ遷移します(現在の RUNNING タスクより高優先なら RUNNING 状態へ遷移)。対象タスクが DORMANT 状態でない場合、このシステムコールにより起動要求のキューイングがおこなわれます。タスク起動時にタスク生成情報に含まれる拡張情報が渡されます。

tskid に TSK_SELF を指定すると自タスクに対する起動要求になりキューイングされます。

戻値

E_OK	正常終了
E_ID	タスク ID が範囲外*
E_NOEXS	タスクが生成されていない
E_QOVR	キューイングオーバーフロー

例
```
#define ID_task2 2
#define ID_task3 3
const T_CTSK ctsk2 = {TA_HLNG, 1, task2, 8, 512, NULL};
const T_CTSK ctsk3 = {TA_HLNG, NULL, task3, 8, 512, NULL};
TASK task2(int exinf)
{
    if (exinf == 1)
        :
}
TASK task3(void) /* exinf を使用しない場合 */
{
        :
}
```

```
TASK task1(void)
{
        :
    cre_tsk(ID_task2, &ctsk2);
    cre_tsk(ID_task3, &ctsk3);
        :
    act_tsk(ID_task2);
    act_tsk(ID_task3);
        :
}
```

can_act

機能　タスク起動要求のキャンセル

形式　ER_UINT can_act(ID tskid);
tskid　タスク ID

解説　tskid で指定されたタスクに対する起動要求をキャンセルし 0 にします。キャンセルする前の起動要求キューイング数(actcnt)を、関数の戻値として返します。

tskid = TSK_SELF で自タスクを指定できます。

戻値　正または 0 キューイングされていた起動要求数
E_ID　タスク ID が範囲外*
E_NOEXS　タスクが生成されていない

例
```
#define ID_task2 2
const T_CTSK ctsk2 = {TA_HLNG, 1, task2, 8, 512, NULL};
TASK task2(int exinf)
{
        :
}
TASK task1(void)
{
    cre_tsk(ID_task2, &ctsk2);
        :
    act_tsk(ID_task2);
        :
    can_act(ID_task2);
        :
}
```

sta_tsk

機能　タスク起動

形式　ER sta_tsk(ID tskid, VP_INT stacd);
tskid　タスク ID
stacd　タスク起動コード

解説　tskid で指定されたタスクを起動し、stacd を渡します(stacd を使用しない場合は 0 を推奨)。対象タスクは DORMANT 状態から READY 状態へ遷移します(現在の RUNNING タスクより高優先なら RUNNING 状態へ遷移)。

このシステムコールによる起動要求のキューイングはおこなわれません。したがって、対象タスクが DORMANT 状態でない場合は、エラーとなります。

戻値　E_OK　正常終了
E_ID　タスク ID が範囲外*
E_OBJ　自タスク指定(tskid = TSK_SELF)*
E_NOEXS　タスクが生成されていない
E_OBJ　タスクが既に起動されている

例
```
#define ID_task2 2
#define ID_task3 3
TASK task2(int stacd)
{
    if (stacd ==1)
        :
}
TASK task3(void) /* stacd を使用しない場合 */
{
        :
}
TASK task1(void)
{
        :
    sta_tsk(ID_task2, 1);
    sta_tsk(ID_task3, 0);
        :
}
```

ext_tsk

機能 **自タスク終了**

形式 void ext_tsk(void);

解説 タスク自ら終了します。タスクは起動要求がキューイングされてなければ RUNNING 状態から DORMANT 状態へ遷移します。起動要求がキューイングされていた場合は、キューイング数から 1 を減じて再起動します。再起動時にはタスクの内部状態は初期化されます。すなわち、タスクの優先度・起床要求数・スタックが初期状態になります。

再起動された場合、初期優先度レディーキューの最後につながります。

戻値 **なし(呼び出し元に戻りません)**

補足 内部的には次のエラーを検出しています。

E_CTX **非タスクコンテキストまたは、ディスパッチ禁止状態で実行***

注意 タスクが獲得していた資源(セマフォやメモリブロック)は自動的に解放されません。ユーザーの責任において、タスク終了前に資源を解放してください。

例

```
TASK task2(void)
{
        :
    ext_tsk();
}
```

このように明示的に呼び出さなくともメインルーチンからのリターンで自動的に呼び出されます。

ter_tsk

機能 他タスク強制終了

形式 ER ter_tsk(ID tskid);
tskid タスク ID

解説 tskid で指定されたタスクを終了させます。終了させた結果、対象タスクは READY または WAITING 状態から DORMANT 状態へ遷移します。起動要求がキューイングされている場合は再起動されます。対象タスクが何等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、対象タスクはその待ち行列から外されます。このシステムコールでは、自タスクは指定できません。

戻値

E_OK	正常終了
E_ID	タスク ID が範囲外*
E_ILUSE	自タスク指定(tskid = TSK_SELF)*
E_NOEXS	タスクが生成されていない
E_OBJ	タスクが起動されていない

注意 タスクが獲得していたミューテックス以外の資源（セマフォやメモリブロック）は自動的に解放されません。ユーザーの責任において、タスク終了前に資源を解放してください。

例

```
#define ID_task2 2
TASK task1(void)
{
        :
    ter_tsk(ID_task2);
        :
}
```

chg_pri

機能 タスク現在優先度変更

形式 ER chg_pri(ID tskid, PRI tskpri);
tskid タスク ID
tskpri 優先度

解説 tskid で指定されたタスクの現在優先度を tskpri の値とします。タスクの優先度は、数の小さい方が高優先です。優先度には初期優先度と現在優先度があります。初期優先度はタスク生成情報に指定(itskpri)した優先度で、タスク起動時に現在優先度にコピーされます。タスクは現在優先度で走行し、chg_pri は現在優先度を変更します。

tskid = TSK_SELF で自タスクを指定できます。tskpri = TPRI_INI で初期優先度、TMIN_TPRI で最高優先度、TMAX_TPRI で最低優先度とすることができます。

対象タスクがレディーキューにつながれていた場合、優先度の変更により、待ち行列のつなぎ替えが起こります。

READY 状態である対象タスクの優先度を、このシステムコールを発行したタスクより高くした場合、このシステムコールを発行したタスクは RUNNING 状態から READY 状態へ遷移し、対象タスクは RUNNING 状態へ遷移します。

自タスクの優先度を他の READY タスクより低くした場合、自タスクは RUNNING 状態から READY 状態へ遷移し、他の READY タスクの中で最も優先度の高いタスクが RUNNING 状態へ遷移します。

現在と同じ優先度を指定した場合、他に同じ優先度のタスクがあると、対象タスクはその優先度の待ち行列の最後に回ります。

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す。次にタスクが起動した時には、初期優先度に戻ります。

戻値

E_OK	正常終了
E_PAR	優先度が範囲外*
E_ID	タスク ID が範囲外*
	非タスクコンテキストで TSK_SELF を指定 *
E_NOEXS	タスクが生成されていない
E_OBJ	タスクが起動されていない

例
```
TASK task1(void)
{
        :
    chg_pri(TSK_SELF, TMIN_TPRI);/* 一時的に最高優先度へ */
        :
    chg_pri(TSK_SELF, TPRI_INI); /* 優先度を戻す */
        :
}
```

get_pri

機能　タスク現在優先度参照

形式　ER get_pri(ID tskid, PRI *tskpri);
tskid　タスク ID
tskpri　対象タスクの現在優先度を返すアドレス

解説　tskid で指定されたタスクの現在優先度を tskpri に返します。tskid = TSK_SELF で自タスクを指定できます。

戻値　E_OK　正常終了
E_ID　タスク ID が範囲外*
E_NOEXS　タスクが生成されていない
E_OBJ　タスクが起動されていない

例
```
TASK task1(void)
{
    PRI tskpri;
        :
    get_pri(TSK_SELF, &tskpri);
        :
}
```

ref_tsk

機能　タスク状態参照

形式　ER ref_tsk(ID tskid, T_RTSK *pk_rtsk);
tskid　タスク ID
pk_rtsk　タスク状態パケットを格納する場所へのポインタ

解説　tskid で指定されたタスクの状態を、*pk_rtsk に返します。

tskid = TSK_SELF で自タスクを指定できます。

タスク状態パケットの構造は次の通りです。

```
typedef struct t_rtsk {
    STAT tskstat;    タスク状態
    PRI tskpri;      現在優先度
    STAT tskwait;    待ち要因
    ID wid;          待ちオブジェクト ID
    TMO lefttmo;     タイムアウトまでの時間
    UINT actcnt;     起動要求カウント
    UINT wupcnt;     起床要求カウント
    VP exinf;        拡張情報
    ATR tskatr;      タスク属性
    FP task;         タスク起動アドレス
    PRI itskpri;     タスク起動時優先度
    int stksz;       スタックサイズ(バイト数)
} T_RTSK;
```

exinf, tskatr, task, itskpri, stksz には、タスク生成で指定された値がそのまま返ります。

tskstat には、タスク状態を示す次の値が返ります。

TTS_RUN	0x0001	RUNNING 状態
TTS_RDY	0x0002	READY 状態
TTS_WAI	0x0004	WAITING 状態
TTS_SUS	0x0008	SUSPENDED 状態
TTS_WAS	0x000c	WAITING-SUSPENDED 状態
TTS_DMT	0x0010	DORMANT 状態

tskwait には、タスクが待ち状態の場合に、その要因を示す次の値が返ります。

TTW_SLP	0x0001	slp_tsk または tslp_tsk による待ち
TTW_DLY	0x0002	dly_tsk による待ち
TTW_SEM	0x0004	wai_sem または twai_sem による待ち
TTW_FLG	0x0008	wai_sem または twai_sem による待ち
TTW_SDTQ	0x0010	snd_dtq による待ち
TTW_RDTQ	0x0020	rcv_dtq による待ち
TTW_MBX	0x0040	rcv_mbx または trcv_mbx による待ち
TTW_MPF	0x2000	固定長メモリブロックの獲得待ち

戻値

E_OK	正常終了
E_ID	タスク ID が範囲外
E_NOEXS	タスクが生成されていない

例

```
#define ID_task2 2
TASK task1(void)
{
    T_RTSK rtsk;
        :
    ref_tsk(ID_task2, &rtsk);
    if (rtsk.tskstat == TTS_WAI)
        :
}
```

ref_tst

機能　タスク状態参照

形式　ER ref_tst(ID tskid, T_RTST *pk_rtst);
tskid　タスク ID
pk_rtst　タスク状態パケットを格納する場所へのポインタ

解説　tskid で指定されたタスクの状態を、*pk_rtst に返します。

tskid = TSK_SELF で自タスクを指定できます。

タスク状態パケットの構造は次の通りです。

```
typedef struct t_rtst {
    STAT tskstat;    タスク状態
    STAT tskwait;    待ち要因
} T_RTST;
```

tskstat, tskwait には ref_tsk と同様の内容が返ります。

戻値　E_OK　正常終了
E_ID　タスク ID が範囲外
E_NOEXS　タスクが生成されていない

例
```
#define ID_task2 2
TASK task1(void)
{
    T_RTSK rtst;
        :
    ref_tst(ID_task2, &rtst);
    if (rtst.tskstat == TTS_WAI)
        :
}
```

5.2 タスク付属同期機能

slp_tsk

機能 自タスクを起床待ち状態へ移行

形式 ER slp_tsk(void);

解説 タスク自ら WAITING 状態へ遷移します。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、または、rel_wai システムコールの発行により解除されます。

wup_tsk による待ち解除では、正常終了 E_OK としてリターンします。wup_tsk が先に発行されていて、起床要求がキューイングされている場合は、slp_tsk で待ち状態に入らずに、起床要求カウントを 1 つ減じて、即時に正常終了 E_OK としてリターンします。この時にタスクのレディキューは変化しません。

rel_wai による解除の場合は、エラーE_RLWAI としてリターンします。

戻値

E_OK	正常終了
E_CTX	非タスクコンテキストで、または、ディスパッチ禁止状態で待ち実行*
E_RLWAI	待ち状態を強制解除された(待ちの間に rel_wai を受け付け)

補足 tslp_tsk(TMO_FEVR)と同じです。

例

```
#define ID_task1 1
TASK task1(void)
{
        :
    slp_tsk();
        :
}
TASK task2(void)
{
        :
    wup_tsk(ID_task1);
        :
}
```

tslp_tsk

機能 自タスクを起床待ち状態へ移行(タイムアウト有)

形式 ER tslp_tsk(TMO tmout);
tmout タイムアウト値

解説 タスク自ら WAITING 状態へ遷移します。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や rel_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ます。

wup_tsk による待ち解除では、正常終了 E_OK としてリターンします。wup_tsk が先に発行されていて、起床要求がキューイングされている場合は、tslp_tsk で待ち状態に入らず、起床要求カウントを 1 つ減じて、即時に正常終了 E_OK としてリターンします。この時にタスクのレディキューは変化しません。

rel_wai による解除の場合は、エラーE_RLWAI としてリターンします。指定時間経過による解除の場合は、タイムアウトエラーE_TMOUT としてリターンします。tmout の時間の単位は、システムクロックの割り込み周期です。タイムアウトを検出するのは、tslp_tsk 発行から tmout 番目のシステムクロックです。

tmout = TMO_POL(= 0)とすると、起床要求がキューイングされている場合は、即時に正常終了 E_OK としてリターンし、起床要求がキューイングされていない場合は、即時にタイムアウトエラーE_TMOUT としてリターンします。tmout = TMO_FEVR(= -1)によりタイムアウトをおこなわない、すなわち slp_tsk と同じ動作になります。

戻値

E_OK	正常終了
E_CTX	非タスクコンテキストで、または、ディスパッチ禁止状態で待ち実行*
E_RLWAI	待ち状態を強制解除された(待ちの間に rel_wai を受け付け)
E_TMOUT	タイムアウト

補足 NORTi Oceans 独自の MSEC マクロを用いて tslp_tsk(100/MSEC);の様に記述することで待ち時間をミリ秒単位で指定できます。MSEC マクロは kernel.h に#define 10 と定義されていますが、システムクロックとして別の値を採用した場合は kernel.h を#include する前にその値に#define してください。

注意 タイムアウト付きのシステムコールを発行した後の、最初の周期タイマ割り込みが入るまでのタイミングはバラつきますから、タイムアウト時間には、-MSEC～0 の誤差があります。例えば MSEC = 10 の時に 100msec のタイムアウトを指定すると、実際には 90～100msec の範囲でタイムアウトします。

なお、μITRON4.0 仕様書では、タイムアウトは「指定された以上の時が経過した後」と規定されています。すなわち、上記の例では 100～110msec の範囲でタイムアウトさせねばなりませんが、NORTi Oceans の実装では 90～100msec となり、μITRON4.0 仕様書とは誤差の方向が逆になっています。

現実には時間待ちを行うタスクは周期タイマ割り込みに同期して動作しますので、次のような動作の違いとなります。

```
for (;;){
    led_on();               /* LED 点灯 */
    tslp_tsk(100/MSEC)      /* 100msec 待ち */
    led_off();              /* LED 消灯 */
    tslp_tsk(100/MSEC);     /* 100msec 待ち */
}
```

NORTi Oceans での動作 → 200msec 周期で点滅

μITRON4.0 仕様 → 220msec 周期で点滅

例

```
#define MSEC 2
#include "kernel.h"
TASK task1(void)
{
    ER ercd;
        :
    ercd = tslp_tsk(100/MSEC);
    if (ercd == E_TMOUT)
        :
}
```

wup_tsk, iwup_tsk

機能　他タスクの起床

形式　ER wup_tsk(ID tskid);
ER iwup_tsk(ID tskid);
tskid　タスク ID

解説　slp_tsk または tslp_tsk システムコールの実行により WAITING 状態になっているタスクを READY 状態へ遷移させます(現在の RUNNING タスクより高優先なら RUNNING 状態へ、WAITING-SUSPENDED 状態だったら SUSPENDED 状態へ遷移)。対象タスクは、tskid で指定されます。タスクコンテキストから自タスクを指定することができます。

対象タスクが slp_tsk または、tslp_tsk を実行しておらず待ち状態でない場合、この起床要求はキューイングされます。キューイングされた起床要求は、後に対象タスクが slp_tsk または tslp_tsk システムコールを実行した時に有効となります。すなわち、起床要求がキューイングされている場合、slp_tsk, tslp_tsk システムコールは、起床要求を１つ減じて即時にリターンします。

戻値

E_OK	正常終了
E_ID	タスク ID が範囲外*
E_ID	非タスクコンテキストで自タスク指定(tskid = TSK_SELF)*
E_NOEXS	タスクが生成されていない
E_OBJ	タスクが起動されていない
E_QOVR	起床要求数のオーバーフロー(TMAX_WUPCNT = 255 を超える)

例

```
#define ID_task1 1
TASK task1(void)
{
        :
    slp_tsk();
        :
}
TASK task2(void)
{
        :
    wup_tsk(ID_task1);
        :
}
```

can_wup

機能　タスクの起床要求を無効化

形式　ER_UINT can_wup(ID tskid);
tskid　タスク ID

解説　tskid で指定されたタスクにキューイングされていた起床要求回数(wupcnt)を返し、同時にその起床要求をすべて解除します。tskid = TSK_SELF によって自タスクの指定になります。

このシステムコールは、周期的にタスクを起床する処理をおこなう場合に、時間内に処理が終わっているかどうかを判定するために利用できます。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ことを示します。

戻値　正または 0 キューイングされていた起床要求回数
E_ID　タスク ID が範囲外*
E_NOEXS　タスクが生成されていない

例
```
TASK task1(void)
{
    ER_UINT wupcnt;
        :
    slp_tsk();
    wupcnt = can_wup(TSK_SELF);
        :
}
```

vcan_wup

機能　自タスクの起床要求を無効化

形式　void vcan_wup(void);

解説　キューイングされている起床要求があれば、それをクリアします。自タスク専用です。NORTi Oceans 独自のシステムコールで、起床要求クリアだけなら、can_wup より高速です。

戻値　なし

例
```
TASK task1(void)
{
        :
    vcan_wup();
    tslp_tsk(100/MSEC);
        :
}
```

rel_wai, irel_wai

機能 他タスクの待ち状態解除

形式
```
ER rel_wai(ID tskid);
ER irel_wai(ID tskid);
tskid        タスク ID
```

解説 tskid で指定されたタスクが何等かの待ち状態にある場合に、それを強制的に解除します。待ち解除されたタスクへは、E_RLWAI エラーが返ります。対象タスクが WAITING 状態だった場合、対象タスクは READY 状態へ遷移します。（現在の RUNNING タスクより高優先なら RUNNING 状態へ遷移）。

対象タスクがそれ以外の状態の時は、E_OBJ エラーとなります。この時、対象タスクの状態は変化しません。

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おこないません。

戻値

E_OK	正常終了
E_ID	タスク ID が範囲外*
E_OBJ	自タスク指定(tskid = TSK_SELF)*
E_NOEXS	タスクが生成されていない
E_OBJ	タスクが待ち状態でない

例
```
#define ID_task2 2
TASK task1(void)
{
        :
    rel_wai(ID_task2);
        :
}
```

dly_tsk

機能 自タスク遅延

形式 ER dly_tsk(RELTIM dlytim);
dlytim 遅延時間

解説 タスクの単純な時間待ちをおこないます。このシステムコールは、tslp_tsk(TMO tmout)とほぼ同じ機能ですが、wup_tsk システムコールの起床要求では待ち解除されません。単に時間待ちをおこなうだけの場合は、tslp_tsk ではなく、この dly_tsk を使用してください。

遅延時間 dlytim の RELTIM 型は、タイムアウトの TMO 型と同じ long です。遅延時間の単位も同じく、システムクロックの割り込み周期です。

戻値 E_OK 正常終了
E_CTX 非タスクコンテキストで、または、ディスパッチ禁止状態で発行*
E_RLWAI 待ち状態を強制解除された(待ちの間に rel_wai を受け付け)

5.3 同期・通信機能（セマフォ）

cre_sem

機能　セマフォ生成

形式　ER cre_sem(ID semid, const T_CSEM *pk_csem);
semid　セマフォ ID
pk_csem　セマフォ生成情報パケットへのポインタ

解説　semid で指定されたセマフォを生成します。すなわち、システムメモリから、セマフォ管理ブロックを動的に割り当てます。また、セマフォ生成情報の isemcnt で指定される初期値をセマフォカウントに設定します。

セマフォ生成情報パケットの構造は次の通りです。

```
typedef struct t_csem {
    ATR sematr;          セマフォ属性
    UINT isemcnt;        セマフォの初期値
    UINT maxsem;         セマフォの最大値
    const char *name;    セマフォ名へのポインタ（省略可）
} T_CSEM;
```

セマフォ属性 sematr には次の値を入れてください。

TA_TFIFO　待ちタスク行列は先着順(FIFO)

maxsem には使用可能とする資源数を設定してください。設定可能な上限値は TMAX_MAXSEM に定義されています。

name は対応デバッガ用ですので、名前を指定しない場合には""か NULL を入れてください。この構造体を初期値付きで定義する場合には、name を省略しても構いません。

戻値　E_OK　正常終了
E_PAR　セマフォ最大値が負または SEMCNT_MAX(255)を超える*
セマフォ初期値が負または最大値を超える*
E_ID　セマフォ ID が範囲外*
E_OBJ　セマフォが既に生成されている
E_CTX　割り込みハンドラから発行*
E_SYS　管理ブロック用のメモリが確保できない**

例
```
#define ID_sem1 1
const T_CSEM csem1 = {TA_TFIFO, 1, 1};
TASK task1(void)
{
    ER ercd;
        :
    ercd = cre_sem(ID_sem1, &csem1);
        :
}
```

acre_sem

機能　セマフォ生成(ID 自動割り当て)

形式　ER_ID acre_sem(const T_CSEM *pk_csem);
pk_csem　セマフォ生成情報パケットへのポインタ

解説　未生成セマフォの ID を、大きな方から検索して割り当てます。セマフォ ID が割り当てられない場合は、E_NOID エラーを返します。それ以外は、cre_sem と同じです。

戻値　正の値　割り当てられたセマフォ ID
E_PAR　セマフォ最大値が負または SEMCNT_MAX(65535)を超える*
セマフォ初期値が負または最大値を超える*
E_NOID　セマフォ ID が不足
E_CTX　割り込みハンドラから発行*
E_SYS　管理ブロック用のメモリが確保できない**

例
```
ID ID_sem1;
const T_CSEM csem1 = {TA_TFIFO, 0, 1};
TASK task1(void)
{
    ER_ID ercd;
        :
    ercd = acre_sem(&csem1);
    if (ercd > 0)
    ID_sem1 = ercd;
        :
}
```

sig_sem, isig_sem

機能 セマフォ資源返却

形式
```
ER sig_sem(ID semid);
ER isig_sem(ID semid);
```
semid セマフォ ID

解説 semid で指定されたセマフォに対して待っているタスクがなければ、セマフォのカウント値を 1 だけ増やします(資源を返却)。セマフォのカウント値が、セマフォ生成時に指定した最大値を越えた場合には、エラーE_QOVR を返します。

このセマフォに対して待っているタスクがあれば、待ち行列の先頭タスクの待ちを解除します。すなわち、WAITING 状態から READY 状態へ遷移させます(現在の RUNNING タスクより高優先なら RUNNING 状態へ遷移)。

戻値

E_OK	正常終了
E_ID	セマフォ ID が範囲外*
E_NOEXS	セマフォが生成されていない
E_QOVR	セマフォカウントのオーバーフロー

wai_sem

機能　セマフォ資源獲得

形式　ER wai_sem(ID semid);
　　semid　　セマフォ ID

解説　semid で指定されたセマフォのカウント値が 1 以上の場合、このセマフォのカウント値を 1 だけ減じて(資源獲得して)、即リターンします。

セマフォのカウント値が 0 の場合、本システムコールの発行タスクはそのセマフォに対する待ち行列につながれます。この場合のセマフォのカウント値は 0 のままです。

戻値　E_OK　　正常終了
　　E_ID　　セマフォ ID が範囲外*
　　E_NOEXS　セマフォが生成されていない
　　E_CTX　　非タスクコンテキストで、または、ディスパッチ禁止状態で待ち実行*
　　E_RLWAI　待ち状態を強制解除された(待ちの間に rel_wai を受け付け)
　　E_DLT　　待ちの間にセマフォが削除された

補足　twai_sem(semid, TMO_FEVR)と同じです。

例
```
#define ID_sem1 1
TASK task1(void)
{
        :
    wai_sem(ID_sem1);
        :
    sig_sem(ID_sem1);
        :
}
```

pol_sem

機能 セマフォ資源獲得(ポーリング)

形式 ER pol_sem(ID semid);
semid セマフォ ID

解説 semid で指定されたセマフォのカウント値が 1 以上の場合、このセマフォのカウント値を 1 だけ減じて(資源獲得して)、即リターンします。

セマフォカウント値が 0 の場合は、待ち状態に入らずに、E_TMOUT エラーで即リターンします。

戻値

E_OK	正常終了
E_ID	セマフォ ID が範囲外*
E_NOEXS	セマフォが生成されていない
E_TMOUT	ポーリング失敗

補足 twai_sem(semid, TMO_POL)と同じです。

例

```
if (pol_sem(ID_sem1) == E_OK)
{
        :
    if (pol_sem(ID_sem1) != E_TMOUT)
        :
}
```

twai_sem

機能　セマフォ資源獲得(タイムアウト有)

形式　ER twai_sem(ID semid, TMO tmout);
semid　セマフォ ID
tmout　タイムアウト値

解説　semid で指定されたセマフォのカウント値が 1 以上の場合、このセマフォのカウント値を 1 だけ減じて(資源獲得して)、即リターンします。セマフォのカウント値が 0 の場合、本システムコールの発行タスクはそのセマフォに対する待ち行列につながれます。この場合のセマフォのカウント値は 0 のままです。tmout で指定した時間が経過すると、タイムアウトエラーE_TMOUT としてリターンします。tmout = TMO_POL(= 0)により待ちをおこなわない、すなわち pol_sem と同じ動作になります。tmout = TMO_FEVR(= -1)によりタイムアウトしない、すなわち wai_sem と同じ動作になります。

戻値
E_OK	正常終了
E_ID	セマフォ ID が範囲外*
E_NOEXS	セマフォが生成されていない
E_CTX	非タスクコンテキストで、または、ディスパッチ禁止状態で待ち実行*
E_RLWAI	待ち状態を強制解除された(待ちの間に rel_wai を受け付け)
E_DLT	待ちの間にセマフォが削除された
E_TMOUT	タイムアウト

例
```
#define ID_sem1 1
TASK task1(void)
{
    ER ercd;
        :
    ercd = twai_sem(ID_sem1, 100/MSEC);
    if (ercd == E_OK)
        :
}
```

ref_sem

機能　セマフォ状態参照

形式　ER ref_sem(ID semid, T_RSEM *pk_rsem);
semid　セマフォ ID
pk_rsem　セマフォ状態パケットを格納する場所へのポインタ

解説　semid で指定されたセマフォの状態を、*pk_rsem に返します。

セマフォ状態パケットの構造は次の通りです。

```
typedef struct t_rsem {
    ID wtskid;      待ちタスクのタスク ID 、無い場合は TSK_NONE
    UINT semcnt;    現在のセマフォカウント値
} T_RSEM;
```

wtskid には、待ちタスクがある場合、その先頭の待ちタスクの ID 番号が返ります。待ちタスクがない場合は、TSK_NONE が返ります。

戻値　E_OK　正常終了
E_ID　セマフォ ID が範囲外
E_NOEXS　セマフォが生成されていない

例
```
#define ID_sem1 1
TASK task1(void)
{
    T_RSEM rsem;
        :
    ref_sem(ID_sem1, &rsem);
    if (rsem.wtskid != TSK_NONE)
        :
}
```

5.4 同期・通信機能（イベントフラグ）

cre_flg

機能 イベントフラグ生成

形式 ER cre_flg(ID flgid, const T_CFLG *pk_cflg);
flgid イベントフラグ ID
pk_cflg イベントフラグ生成情報パケットへのポインタ

解説 flgid で指定されたイベントフラグを生成します。すなわち、システムメモリから、イベントフラグ管理ブロックを動的に割り当てます。また、イベントフラグ生成情報の iflgptn で指定される初期値をイベントフラグのビットパターンに設定します。

イベントフラグ生成情報パケットの構造は次の通りです。

```
typedef struct t_cflg {
    ATR flgatr;      イベントフラグ属性
    FLGPTN iflgptn;  イベントフラグの初期値
    B *name;         イベントフラグ名へのポインタ（省略可）
} T_CFLG;
```

使用可能なフラグビット数は TBIT_FLGPTN マクロにより参照できます。

イベントフラグ属性 flgatr には次の値を入れてください。

TA_WSGL 複数タスクの待ちを許さない
TA_CLR タスクの待ち解除時にフラグビットをすべてクリアする

NORTi カーネル Ver.4 との互換性のため、TA_WMUL(複数タスクの待ちを許す)、TA_TFIFO(待ちタスク行列は先着順)、TA_TPRI(待ちタスク行列はタスク優先度順)を指定することが出来ます。しかし、TA_WMUL を指定した場合には E_PAR エラーが返ります。TA_WSGL を指定した場合には、TA_TFIFO, TA_TPRI を指定しても意味がありません。

name は対応デバッガ用ですので､名前を指定しない場合には ""か NULL を入れてください。

この構造体を初期値付きで定義する場合には、name を省略しても構いません。

戻値

E_OK	正常終了
E_ID	イベントフラグ ID が範囲外*
E_PAR	flg_atr に TA_WMUL を指定した*
E_OBJ	イベントフラグが既に生成されている
E_CTX	割り込みハンドラから発行*
E_SYS	管理ブロック用のメモリが確保できない**

例

```
#define ID_flg1 1
const T_CFLG cflg1 = { TA_WSGL|TA_TFIFO|TA_CLR, 0 }
TASK task1(void)
{
    ER ercd;
        :
    ercd =cre_flg(ID_flg1, &cflg1);
        :
}
```

acre_flg

機能　イベントフラグ生成(ID 自動割り当て)

形式　ER_ID acre_flg(const T_CFLG *pk_cflg);
pk_cflg　イベントフラグ生成情報パケットへのポインタ

解説　未生成イベントフラグの ID を、大きな方から検索して割り当てます。イベントフラグ ID が割り当てられない場合は、E_NOID エラーを返します。それ以外は、cre_flg と同じです。

戻値　正の値　割り当てられたイベントフラグ ID
E_NOID　イベントフラグ ID が不足
E_PAR　flg_atr に TA_WMUL を指定した*
E_CTX　割り込みハンドラから発行*
E_SYS　管理ブロック用のメモリが確保できない**

例
```
ID ID_flg1;
const T_CFLG cflg1 = { TA_WSGL|TA_TFIFO|TA_CLR, 0 };
TASK task1(void)
{
    ER_ID ercd;
        :
    ercd = acre_flg(&cflg1);
    if (ercd > 0)
    ID_flg1 = ercd;
        :
}
```

set_flg, iset_flg

機能 イベントフラグのセット

形式
```
ER set_flg(ID flgid, FLGPTN setptn);
ER iset_flg(ID flgid, FLGPTN setptn);
```
flgid　　イベントフラグ ID
setptn　　セットするビットパターン

解説 flgid で指定されるイベントフラグの、setptn で示されるビットがセットされます。つまり、現在のイベントフラグの値に対して、setptn の値で論理和がとられます(flgptn |= setptn)。

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条件を満たすようになれば、そのタスクの待ちを解除します。すなわち、WAITING 状態から READY 状態へ遷移させます(現在の RUNNING タスクより高優先なら RUNNING 状態へ遷移) 。

イベントフラグ生成時に TA_CLR を指定した場合は、タスクを待ち解除した時点でイベントフラグをクリアします。

戻値
E_OK　　正常終了
E_ID　　イベントフラグ ID が範囲外*
E_NOEXS　　イベントフラグが生成されていない

例
```
#define ID_flg1 1
#define BIT0 0x0001
TASK task1(void)
{
        :
    set_flg(ID_flg1, BIT0);
        :
}
```

clr_flg

機能 イベントフラグのクリア

形式 ER clr_flg(ID flgid, FLGPTN clrptn);
flgid イベントフラグ ID
clrptn クリアするビットパターン

解説 flgid で指定されるイベントフラグの、clrptn で 0 となっているビットがクリアされます。つまり、現在のイベントフラグの値に対して、clrptn の値で論理積がとられます

（flgptn &= clrptn ）。

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ありません。

戻値 E_OK 正常終了
E_ID イベントフラグ ID が範囲外*
E_NOEXS イベントフラグが生成されていない

例
```
#define ID_flg1 1
#define BIT0 0x0001
TASK task1(void)
{
        :
    clr_flg(ID_flg1, ~BIT0);
        :
}
```

wai_flg

機能　イベントフラグ待ち

形式　ER wai_flg(ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn);
flgid　イベントフラグ ID
waiptn　待ちビットパターン
wfmode　待ちモード
p_flgptn　待ち解除時のビットパターンを格納する場所へのポインタ

解説　waiptn と wfmode で示される待ち条件にしたがって、flgid で指定されるイベントフラグがセットされるのを待ちます。

待ちモード wfmode には、次の様な値を入れてください。

TWF_ANDW　AND 待ち
TWF_ORW　OR 待ち
TWF_ANDW | TWF_CLR　クリア指定 AND 待ち
TWF_ORW | TWF_CLR　クリア指定 OR 待ち

TWF_ORW を指定した場合は、waiptn で指定したビットのいずれかがセットされるのを待ちます。TWF_ANDW を指定した場合は、waiptn で指定したビット全てがセットされるのを待ちます。waiptn で 1 のビットが 1 個だけなら、TWF_ANDW, TWF_ORW は同じ結果です。

TWF_CLR の指定がある場合は、条件が満足されてタスクが待ち解除となった時に、イベントフラグの全ビットをクリアします。ただし、生成情報でフラグ属性として TA_CLR を指定した場合は TWF_CLR を指定しなくとも常に全ビットクリアされます。

*p_flgptn には、待ち状態が解除される時のイベントフラグの値が返されます。クリア指定の場合は、クリアされる前の値が返されます。

すでにイベントフラグの条件が成立している場合には、待ち状態に入らず、上記の操作をおこないます。

戻値　E_OK　正常終了
E_PAR　待ちモード wfmode が正しくない*
待ちビットパターン waiptn が 0*
E_ID　イベントフラグ ID が範囲外*
E_NOEXS　イベントフラグが生成されていない
E_ILUSE　すでに待ちタスクあり(複数待ち許さない場合)
E_CTX　非タスクコンテキストで、または、ディスパッチ禁止状態で待ち実行*
E_RLWAI　待ち状態を強制解除された(待ちの間に rel_wai を受け付け)

補足 twai_flg(flgid, waiptn, wfmode, p_flgptn, TMO_FEVR)と同じです。

例

```
#define ID_flg1 1
#define BIT0 0x0001
TASK task1(void)
{
    FLGPTN ptn;
        :
    wai_flg(ID_flg1, BIT0, TWF_ANDW, &ptn);
        :
}
```

pol_flg

機能　イベントフラグ待ち(ポーリング)

形式　ER pol_flg(ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn);
flgid　イベントフラグ ID
waiptn　待ちビットパターン
wfmode　待ちモード
p_flgptn　待ち解除時のビットパターンを格納する場所へのポインタ

解説　waiptn と wfmode で示される待ち条件にしたがって、flgid で指定されるイベントフラグがセットされているかテストします。すでに待ち条件が満たされている場合には、正常終了します。待ち条件が満たされていない場合は、エラーE_TMOUT で即リターンします。

*p_flgptn には、待ち状態が解除される時のイベントフラグの値が返されます。クリア指定の場合は、クリアされる前の値が返されます。

wfmode の説明は、wai_flg を参照してください。

戻値　E_OK　正常終了
E_PAR　待ちモード wfmode が正しくない*
待ちビットパターン waiptn が 0*
E_ID　イベントフラグ ID が範囲外*
E_NOEXS　イベントフラグが生成されていない
E_ILUSE　すでに待ちタスクあり
E_TMOUT　ポーリング失敗

補足　twai_flg(flgid, waiptn, wfmode, p_flgptn, TMO_POL)と同じです。

例
```
#define ID_flg1 1
TASK task1(void)
{
    FLGPTN ptn;
        :
    if (pol_flg(ID_flg1, 0xffff, TWF_ORW|TWF_CLR, &ptn) == E_OK)
        :
}
```

twai_flg

機能　イベントフラグ待ち(タイムアウト有)

形式　ER twai_flg(ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn, TMO tmout);

flgid	イベントフラグ ID
waiptn	待ちビットパターン
wfmode	待ちモード
p_flgptn	待ち解除時のビットパターンを格納する場所へのポインタ
tmout	タイムアウト値

解説　waiptn と wfmode で示される待ち条件にしたがって、flgid で指定されるイベントフラグがセットされるのを待ちます。すでに待ち条件が満たされている場合には、待ち状態に入らず正常終了します。

tmout で指定した時間が経過すると、タイムアウトエラーE_TMOUT としてリターンします。tmout = TMO_POL(= 0)により待ちをおこなわない、すなわち pol_flg と同じ動作になります。tmout =TMO_FEVR(= -1)によりタイムアウトしない、すなわち wai_flg と同じ動作になります。

wfmode と p_flgptn の説明は、wai_flg を参照してください。

戻値

E_OK	正常終了
E_PAR	待ちモード wfmode が正しくない*
	待ちビットパターン waiptn が 0*
E_ID	イベントフラグ ID が範囲外*
E_NOEXS	イベントフラグが生成されていない
E_OBJ	すでに待ちタスクあり(複数待ちを許さない場合)
E_CTX	非タスクコンテキストで、または、ディスパッチ禁止状態で待ち実行*
E_RLWAI	待ち状態を強制解除された(待ちの間に rel_wai を受け付け)
E_TMOUT	タイムアウト

例

```
#define ID_flg1 1
TASK task1(void)
{
    FLGPTN ptn;
    ER ercd;
        :
    ercd = twai_flg(ID_flg1, 0xffff, TWF_ANDW|TWF_CLR, &ptn, 1000/MSEC);
    if (ercd == E_TMOUT)
        :
}
```

ref_flg

機能　イベントフラグ状態参照

形式　ER ref_flg(ID flgid, T_RFLG *pk_rflg);
flgid　イベントフラグ ID
pk_rflg　イベントフラグ状態パケットを格納する場所へのポインタ

解説　flgid で指定されたイベントフラグの状態を、*pk_rflg に返します。

イベントフラグ状態パケットの構造は次の通りです。

```
typedef struct t_rflg {
    ID wtskid;      待ちタスク ID または TSK_NONE
    FLGPTN flgptn;  現在のビットパターン
} T_RFLG;
```

wtskid には、待ちタスクがある場合、その先頭の待ちタスクの ID 番号が返ります。待ちタスクがない場合は、TSK_NONE が返ります。

戻値　E_OK　正常終了
E_ID　イベントフラグ ID が範囲外
E_NOEXS　イベントフラグが生成されていない

例
```
#define ID_flg1 1
TASK task1(void)
{
    T_RFLG rflg;
        :
    ref_flg(ID_flg1, &rflg);
    if (rflg.flgptn != 0)
        :
}
```

5.5 同期・通信機能（データキュー）

cre_dtq

機能 データキュー生成

形式 ER cre_dtq(ID dtqid, const T_CDTQ *pk_cdtq);
dtqid データキューID
pk_cdtq データキュー生成情報パケットへのポインタ

解説 dtqid で指定されたデータキューを生成します。すなわち、システムメモリから、データキュー管理ブロックを動的に割り当てます。

データキュー生成情報パケットの構造は次の通りです。

```
typedef struct t_cdtq {
    ATR dtqatr;  データキュー属性（未使用）
    UINT dtqcnt; データキューサイズ(データ数)
    VP dtq;      データバッファアドレス
    B *name;     データキュー名へのポインタ(省略可)
} T_CDTQ;
```

データキュー属性 dtqatr は無視され、送信待ちタスク行列は先着順(FIFO)となりますが、NORTi Ver.4 との互換性のために、次の値を入れておくことを推奨します。

TA_TFIFO 送信待ちタスク行列は先着順(FIFO)

受信待ちタスク行列は常に先着順(FIFO)になります。また、データ順も送信順になります。ただし強制送信(fsnd_dtq, ifsnd_dtq)を使った場合は強制送信データが先に受信される場合があります。

dtqcnt にはキューイングするデータ数を、dtq にはデータバッファのアドレスを設定してください。TSZ_DTQ(n)マクロによりデータ数 n の場合の必要メモリ量を知ることができます。dtq に NULL を設定するとデータバッファはシステムメモリに取られます。dtqcnt に 0 を設定するとバッファを使用せずにタスク間のデータ直接渡しになり同期を取ることができます。

name は対応デバッガ用ですので､名前を指定しない場合には ""か NULL を入れてください。この構造体を初期値付きで定義する場合には、name を省略しても構いません。

戻値

E_OK	正常終了
E_ID	データキューIDが範囲外*
E_OBJ	データキューが既に生成されている
E_CTX	割り込みハンドラから発行*
E_SYS	管理ブロック用のメモリが確保できない**

例

```
#define ID_dtq1 1
const T_CDTQ cdtq1 = {TA_TFIFO, 30, NULL};
TASK task1(void)
{
    ER ercd;
        :
    ercd = cre_dtq(ID_dtq1, &cdtq1);
        :
}
```

acre_dtq

機能　データキュー生成(ID 自動割り当て)

形式　ER_ID acre_dtq(const T_CDTQ *pk_cdtq);
pk_cdtq　データキュー生成情報パケットへのポインタ

解説　未生成データキューの ID を、大きな方から検索して割り当てます。データキューID が割り当てられない場合は、E_NOID エラーを返します。それ以外は、cre_dtq と同じです。

戻値　正の値　割り当てられたデータキューID
E_NOID　データキューID が不足
E_CTX　割り込みハンドラから発行*
E_SYS　管理ブロック用のメモリが確保できない**

例
```
ID ID_dtq1;
const T_CDTQ cdtq1 = {TA_TFIFO 30, NULL};
TASK task1(void)
{
    ER_ID ercd;
        :
    ercd = acre_dtq(&cdtq1);
    if (rcd > 0)
    ID_dtq1 = ercd;
        :
}
```

snd_dtq

機能 データ送信

形式 ER snd_dtq(ID dtqid,VP_INT data);
dtqid データキューID
data送信するデータ

解説 dtqid で指定されるデータキューに、data が送信されます。

受信待ち行列にタスクがある場合は、先頭タスクの待ちを解除します。すなわち、WAITING 状態から READY 状態へ遷移させます(現在の RUNNING タスクより高優先なら RUNNING 状態へ遷移)。

受信待ちのタスクが無い場合は、データをデータバッファの末尾に入れます。データバッファに空きが無い場合は自タスクを送信待ち行列につなぎます。

戻値

E_OK	正常終了
E_ID	データキューID が範囲外*
E_NOEXS	データキューが生成されていない
E_RLWAI	待ち状態を強制解除された(待ちの間に rel_wai を受け付け)
E_CTX	非タスクコンテキスト部から、あるいはディスパッチ禁止中に実行

補足 tsnd_dtq(dtqid, data, TMO_FEVR)と同じです。

例

```
#define ID_dtq1 1
TASK task1(void)
{
    VP_INT data
        :
    data = (VP_INT) 1;
    snd_dtq(ID_dtq1, data);
        :
}
```

psnd_dtq, ipsnd_dtq

機能　データ送信(ポーリング)

形式　ER psnd_dtq(ID dtqid,VP_INT data);
ER ipsnd_dtq(ID dtqid,VP_INT data);
dtqid　データキューID
data　送信するデータ

解説　dtqid で指定されるデータキューに、data が送信されます。

受信待ち行列にタスクがある場合は、先頭タスクの待ちを解除します。すなわち、WAITING 状態から READY 状態へ遷移させます(現在の RUNNING タスクより高優先なら RUNNING 状態へ遷移)。

受信待ちのタスクが無い場合は、データをデータバッファの末尾に入れます。データバッファに空きが無い場合はエラーE_TMOUT で直ちにリターンします。データバッファサイズを 0 とした場合は、受信待ちタスクがない場合に E_TMOUT で返ります。

戻値　E_OK　正常終了
E_ID　データキューID が範囲外*
E_NOEXS　データキューが生成されていない
E_TMOUT　ポーリング失敗

補足　tsnd_dtq(dtqid, data, TMO_POL)と同じです。

例
```
#define ID_dtq1 1
TASK task1(void)
{
    VP_INT data;
    ER ercd;
        :
    data = (VP_INT) 1;
    ercd = psnd_dtq(ID_dtq1, data);
    if (ercd == E_OK)
        :
        :
}
```

tsnd_dtq

機能　データ送信

形式　ER tsnd_dtq(ID dtqid, VP_INT data, TMO tmout);
dtqid　データキューID
data　送信するデータ
tmout　タイムアウト値

解説　dtqid で指定されるデータキューに、data が送信されます。

受信待ち行列にタスクがある場合は、先頭タスクの待ちを解除します。すなわち、WAITING 状態から READY 状態へ遷移させます(現在の RUNNING タスクより高優先なら RUNNING 状態へ遷移)。

受信待ちのタスクが無い場合は、データをデータバッファの末尾に入れます。データバッファに空きが無い場合は自タスクを送信待ち行列につなぎます。

tmout で指定した時間が経過しても空きがない場合、タイムアウトエラーE_TMOUT としてリターンします。tmout = TMO_POL(= 0)により待ちをおこなわない、すなわち psnd_dtq と同じ動作になります。tmout = TMO_FEVR(= -1)によりタイムアウトしない、すなわち snd_dtq と同じになります。

戻値　E_OK　正常終了
E_ID　データキューID が範囲外*
E_NOEXS　データキューが生成されていない
E_RLWAI　待ち状態を強制解除された(待ちの間に rel_wai を受け付け)
E_CTX　非タスクコンテキスト部から、あるいはディスパッチ禁止中に実行
E_TMOUT　タイムアウト

例
```
#define ID_dtq1 1
TASK task1(void)
{
    VP_INT data;
    ER ercd;
        :
    data = (VP_INT) 1;
    ercd = tsnd_dtq(ID_dtq1, data, 1000/MSEC);
    if (ercd != E_TMOUT)
        :
        :
}
```

fsnd_dtq, ifsnd_dtq

機能 強制データ送信

形式
```
ER fsnd_dtq(ID dtqid, VP_INT data);
ER ifsnd_dtq(ID dtqid, VP_INT data);
```
dtqid　　データキューID
data　　送信するデータ

解説 dtqid で指定されるデータキューに、data を強制送信します。

受信待ち行列にタスクがある場合は、先頭タスクにデータを渡し、待ちを解除します。すなわち、WAITING 状態から READY 状態へ遷移させます(現在の RUNNING タスクより高優先なら RUNNING 状態へ遷移)。

受信待ちのタスクが無い場合は、データをデータバッファの末尾に入れます。データバッファに空きが無い場合はデータキューの先頭のデータを廃棄してそこに強制送信データを入れます。送信待ちタスクがある場合でもデータをバッファに入れます。

バッファサイズ 0 の場合は、受信待ちタスクがある場合でも E_ILUSE エラーを返します。

戻値
E_OK　　正常終了
E_ID　　データキューID が範囲外*
E_NOEXS　　データキューが生成されていない
E_ILUSE　　バッファサイズ 0

例
```
#define ID_dtq1 1
TASK task1(void)
{
    VP_INT data;
        :
    data = (VP_INT) 1;
    fsnd_dtq(ID_dtq1, data);
        :
}
```

rcv_dtq

機能　データキューからの受信

形式　ER rcv_dtq(ID dtqid, VP_INT *p_data);
dtqid　デ ータキューID
p_data　受信したデータを格納する場所へのポインタ

解説　dtqid で指定されるデータキューから先頭のデータを受信します。送信待ちのタスクがある場合には、送信しようとしているデータをデータキューに入れて送信待ちタスクの待ちを解除します。データキューサイズが 0 の場合は、送信待ち行列の先頭のタスクからデータを受け取りそのタスクの待ちを解除します。

データも送信待ちタスクも無い場合、発行タスクは受信待ち行列につながれます。

戻値　E_OK　正常終了
E_ID　データキューID が範囲外*
E_NOEXS　データキューが生成されていない
E_CTX　非タスクコンテキストで、または、ディスパッチ禁止状態で待ち実行*
E_RLWAI　待ち状態を強制解除された(待ちの間に rel_wai を受け付け)

補足　trcv_dtq(dtqid, p_data, TMO_FEVER)と同じです。

例
```
#define ID_dtq1 1
TASK task1(void)
{
    VP_INT data;
        :
    rcv_dtq(ID_dtq1, &data);
        :
}
```

prcv_dtq

機能　データキューからの受信(ポーリング)

形式　ER prcv_dtq(ID dtqid, VP_INT *p_data);
dtqid　データキューID
p_data　受信したデータを格納する場所へのポインタ

解説　dtqid で指定されるデータキューから先頭のデータを受信します。送信待ちのタスクがある場合には、送信しようとしているデータをデータキューに入れて送信待ちタスクの待ちを解除します。データキューサイズが 0 の場合は、送信待ち行列の先頭のタスクからデータを受け取りそのタスクの待ちを解除します。

データも送信待ちタスクも無い場合、E_TMOUT エラーで戻ります。

戻値　E_OK　正常終了
E_ID　データキューID が範囲外*
E_NOEXS　データキューが生成されていない
E_TMOUT　ポーリング失敗

補足　trcv_dtq(dtqid, p_data, TMO_POL)と同じです。

例
```
#define ID_dtq1 1
TASK task1(void)
{
    VP_INT data;
        :
    if (prcv_dtq(ID_dtq1, &data) == E_OK)
        :
}
```

trcv_dtq

機能　データキュー待ち(タイムアウト有)

形式　ER trcv_dtq(ID dtqid, VP_INT *p_data, TMO tmout);
dtqid　データキューID
p_data　受信したデータを格納する場所へのポインタ
tmout　タイムアウト値

解説　dtqid で指定されるデータキューから先頭のデータを受信します。送信待ちのタスクがある場合には、送信しようとしているデータをデータキューに入れて送信待ちタスクの待ちを解除します。データキューサイズが 0 の場合は、送信待ち行列の先頭のタスクからデータを受け取りそのタスクの待ちを解除します。

tmout で指定した時間が経過しても受信できない場合、タイムアウトエラーE_TMOUT としてリターンします。tmout = TMO_POL (= 0)により待ちをおこなわない、すなわち prcv_dtq と同じ動作になります。tmout = TMO_FEVR (= -1)によりタイムアウトしない、すなわち rcv_dtq と同じになります。

戻値　E_OK　正常終了
E_ID　データキューID が範囲外*
E_NOEXS　データキューが生成されていない
E_CTX　非タスクコンテキストで、または、ディスパッチ禁止状態で待ち実行*
E_RLWAI　待ち状態を強制解除された(待ちの間に rel_wai を受け付け)
E_TMOUT　タイムアウト

例
```
#define ID_dtq1 1
TASK task1(void)
{
    VP_INT data;
    ER ercd;
        :
    ercd = trcv_dtq(ID_dtq1, &data, 1000/MSEC);
    if (ercd == E_TMOUT)
        :
}
```

ref_dtq

機能　データキュー状態参照

形式　ER ref_dtq(ID dtqid, T_RDTQ *pk_rdtq);
dtqid　データキューID
pk_rdtq　データキュー状態パケットを格納する場所へのポインタ

解説　dtqid で指定されたデータキューの状態を、*pk_rdtq に返します。

データキュー状態パケットの構造は次の通りです。

```
typedef struct t_rdtq {
    ID stskid;      送信待ちタスク ID または TSK_NONE
    ID rtskid;      受信待ちタスク ID または TSK_NONE
    UINT sdtqcnt;   データキューに入っているデータ数
} T_RDTQ;
```

stskid, rtskid には、待ちタスクがある場合、その先頭の待ちタスク ID 番号が入ります。

待ちタスクがない場合は、TSK_NONE が返ります。

戻値　E_OK　正常終了
E_ID　データキューID が範囲外
E_NOEXS　データキューが生成されていない

例
```
#define ID_dtq1 1
TASK task1(void)
{
    T_RDTQ rdtq;
        :
    ref_dtq(ID_dtq1, &rdtq);
    if (rdtq.sdtqcnt != 0)
        :
}
```

5.6 同期・通信機能（メールボックス）

cre_mbx

機能　メールボックス生成

形式　ER cre_mbx(ID mbxid, const T_CMBX *pk_cmbx);
mbxid　メールボックス ID
pk_cmbx　メールボックス生成情報パケットへのポインタ

解説　mbxid で指定されたメールボックスを生成します。すなわち、システムメモリから、メールボックス管理ブロックを動的に割り当てます。

メールボックス生成情報パケットの構造は次の通りです。

```
typedef struct t_cmbx {
    ATR mbxatr;      メールボックス属性（未使用）
    PRI maxmpri;     メッセージ優先度の最大値（未使用）
    VP mprihd;       メッセージ待ち行列先頭アドレス
    B *name;メールボックス名へのポインタ（省略可）
} T_CMBX;
```

受信待ちタスク行列およびメッセージのキューイングは常に先着順(FIFO)となります。つまり、メールボックス属性 mbxatr とメッセージ優先度の最大値 maxpri は無視されますが、NORTi Ver.4 との互換性のため、mbxatr には次の値を、maxmpri には 1 を入れておくことを推奨します。

TA_TFIFO　受信待ちタスク行列は先着順(FIFO)
TA_MFIFO　メッセージのキューイングは先着順(FIFO)

メッセージ待ち行列ヘッダのサイズは、TSZ_MPRIHD マクロにより知ることができます。ユーザー領域に待ち行列ヘッダを用意する場合は TSZ_MPRIHD で得たバイト数のメモリ領域を確保して先頭アドレスを mprihd に設定してください。mprihd に NULL を設定した場合、行列ヘッダはシステムメモリに確保されます。

name は対応デバッガ用ですので、名前を指定しない場合には""か NULL を入れてください。この構造体を初期値付きで定義する場合には、name を省略しても構いません。

戻値 E_OK 正常終了
E_ID メールボックス ID が範囲外*
E_OBJ メールボックスが既に生成されている
E_CTX 割り込みハンドラから発行*
E_SYS 管理ブロック用のメモリが確保できない**

例
```
#define ID_mbx1 1
const T_CMBX cmbx1 = {TA_TFIFO|TA_MFIFO, 1, NULL};
TASK task1(void)
{
    ER ercd;
        :
    ercd = cre_mbx(ID_mbx1, &cmbx1);
        :
}
```

acre_mbx

機能　メールボックス生成(ID 自動割り当て)

形式　ER_ID acre_mbx(const T_CMBX *pk_cmbx);
pk_cmbx　メールボックス生成情報パケットへのポインタ

解説　未生成メールボックスの ID を、大きな方から検索して割り当てます。メールボックス ID が割り当てられない場合は、E_NOID エラーを返します。それ以外は、cre_mbx と同じです。

戻値　正の値　割り当てられたメールボックス ID
E_NOID　メールボックス ID が不足
E_CTX　割り込みハンドラから発行*
E_SYS　管理ブロック用のメモリが確保できない**

例
```
ID ID_mbx1;
const T_CMBX cmbx1 = {TA_TFIFO|TA_MFIFO, 1, NULL };
TASK task1(void)
{
    ER_ID ercd;
        :
    ercd = acre_mbx(&cmbx1);
    if (ercd > 0)
        ID_mbx1 = ercd;
}
```

snd_mbx

機能 メールボックスへ送信

形式 ER snd_mbx(ID mbxid, T_MSG *pk_msg);
mbxid メールボックス ID
pk_msg メッセージパケットへのポインタ

解説 mbxid で指定されるメールボックスを使って、pk_msg で指し示されるメッセージを送信します。メッセージの内容はコピーされずに、受信側にはポインタ(pk_msg の値)のみが渡されます。OS はメッセージのサイズを関知しません。

このメールボックスに対して待っているタスクがなければ、メッセージをメールボックスのメッセージキューにつないで、即リターンします。

このメールボックスに対して待っているタスクがあれば、待ち行列の先頭タスクへメッセージを渡して、その待ちを解除します。すなわち、WAITING 状態から READY 状態へ遷移させます(現在の RUNNING タスクより高優先なら RUNNING 状態へ遷移)。

標準のメッセージパケットとして定義されている T_MSG 型の構造を示します。

```
typedef struct t_msg {
    struct t_msg *next;  次のメッセージへのポインタ
    VB msgcont[MSGS] ;   メッセージの内容
} T_MSG;
```

メッセージをキューイングするために、メッセージヘッダ部 next を、OS がポインタとして使います。ユーザー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 msgcont からとなります。

T_MSG 型は、システムコール関数のプロトタイプ宣言のために定義されており、ユーザープログラムでは、通常、これを使用しません。用途に応じたメッセージの型を定義し、システムコールへ渡す際に、(T_MSG *)や(T_MSG**)でキャストしてください。

既にキューイングされているメッセージを再度 snd_mbx した場合は OS が使用する領域が破壊されますので多重送信はしないでください。

戻値 E_OK 正常終了
E_ID メールボックス ID が範囲外*
E_NOEXS メールボックスが生成されていない

補足 メッセージ長 MSGS は標準で 16 バイトですが、#include "kernel.h"の前で MSGS を別の値に#define できます(例 1)。

それよりも、用途に応じて msgcont の部分を変更したメッセージパケット構造体を、ユーザーが独自定義する方がよいでしょう(例 2)。メッセージはコピーされずにキューイングされるので、各メッセージはメモリプール等から取得した別々の領域へ格納してください。グローバルな 1 個の変数を使用する場合は、2 つ以上キューイングすると多重送信問題が発生します。

また、関数の中で自動変数として確保した領域は、その関数から抜けると開放されてしまうため、メッセージ領域としては使用禁止です。

例 1

```
#define MSGS 4
#include "kernel.h"
#define ID_mbx 1
#define ID_mpf 1

TASK task1(void)
{
    T_MSG *msg;
        :
    get_mpf(ID_mpf, &msg);    /* メッセージ領域を得る */
    msg->msgcont[0] = 2;
    msg->msgcont[1] = 0;
    msg->msgcont[2] = 3;
    msg->msgcont[3] = 0;
    snd_mbx(ID_mbx, msg);     /* メールボックスへ送信 */
        :
}
```

例２

```
typedef struct t_mymsg
{   struct t_mymsg *next;     /* 次のメッセージへのポインタ(注) */
    H fncd;
    H data;
}T_MYMSG;

#define ID_mbx 1
#define ID_mpf 1

TASK task1(void)
{
    T_MYMSG *msg;
        :
    get_mpf(ID_mpf, &msg);    /* メッセージ領域を得る */
    msg->fncd = 1;
    msg->data = 2;
    snd_mbx(ID_mbx, (T_MSG *)msg);    /* メールボックスへ送信 */
        :
}
```

(注)FAR ポインタのある処理系では、struct t_mymsg PFAR *next;の様に記述する必要があります。

rcv_mbx

機能 メールボックスから受信

形式 ER rcv_mbx(ID mbxid, T_MSG **ppk_msg);
mbxid メールボックス ID
ppk_msg メッセージパケットへのポインタを格納する場所へのポインタ

解説 mbxid で指定されたメールボックスからメッセージを受け取ります。メッセージ内容はコピーされずに、ポインタのみを*ppk_msg へ受け取ります。

すでにメッセージがキューイングされている場合、先頭のメッセージへのポインタを*ppk_msg に入れ、即リターンします。メールボックスにまだメッセージが到着していない場合、本システムコールの発行タスクは、そのメールボックスの待ち行列につながれます。

戻値
E_OK	正常終了
E_ID	メールボックス ID が範囲外*
E_NOEXS	メールボックスが生成されていない
E_CTX	非タスクコンテキストで、または、ディスパッチ禁止状態で待ち実行*
E_RLWAI	待ち状態を強制解除された(待ちの間に rel_wai を受け付け)

注意 ppk_msg は、ポインタへのポインタです。

補足 trcv_mbx(ppk_msg, mbxid, TMO_FEVR)と同じです

送信側タスクがメッセージ領域をメモリプールから獲得していた場合、受信側タスクでは、メッセージの参照が終わったら、その領域を同じメモリプールへ返却しなければなりません。

例
```
#define ID_mbx1 1
#define ID_mpf1 1
TASK task2(void)
{
    T_MYMSG *msg;
        :
    rcv_mbx(ID_mbx1, (T_MSG**)&msg);
        :
    rel_mpf(ID_mpf1, (VP)msg);    /* メッセージをメモリプールへ返却 */
}
```

prcv_mbx

機能　メールボックスから受信(ポーリング)

形式　ER prcv_mbx(ID mbxid, T_MSG **ppk_msg);
mbxid　メールボックス ID
ppk_msg　メッセージパケットへのポインタを格納する場所へのポインタ

解説　mbxid で指定されたメールボックスからメッセージを受け取ります。

メッセージ内容はコピーされずに、ポインタのみを*ppk_msg へ受け取ります。

すでにメッセージがキューイングされている場合、先頭のメッセージへのポインタを*ppk_msg に入れ、即リターンします。メールボックスにまだメッセージが到着していない場合は、待ち状態に入らずに、E_TMOUT エラーで即リターンします。

戻値　E_OK　正常終了
E_ID　メールボックス ID が範囲外*
E_NOEXS　メールボックスが生成されていない
E_TMOUT　ポーリング失敗

注意　ppk_msg は、ポインタへのポインタです。

補足　trcv_mbx(ppk_msg, mbxid, ppk_msg, TMO_POL)と同じです。

例
```
#define ID_mbx1 1
TASK task1(void)
{
    T_MYMSG *msg;
    ER ercd;
        :
    ercd = prcv_mbx(ID_mbx1, (T_MSG**)&msg);
    if (ercd == E_OK)
        :
}
```

trcv_mbx

機能 メールボックスから受信(タイムアウト有)

形式
```
ER trcv_mbx(ID mbxid, T_MSG **ppk_msg, TMO tmout);
```
mbxid　メールボックス ID
ppk_msg　メッセージパケットへのポインタを格納する場所へのポインタ
tmout　タイムアウト値

解説 mbxid で指定されたメールボックスからメッセージを受け取ります。

メッセージ内容はコピーされずに、ポインタのみを*ppk_msg へ受け取ります。

すでにメッセージがキューイングされている場合、先頭のメッセージへのポインタを*ppk_msg に入れ、即リターンします。メールボックスにまだメッセージが到着していない場合、本システムコールの発行タスクは、そのメールボックスの待ち行列につながれます。

tmout で指定した時間が経過してもメッセージが来ない場合、タイムアウトエラーE_TMOUTとしてリターンします。tmout = TMO_POL(= 0)により待ちをおこなわない、すなわちprcv_mbx と同じ動作になります。tmout = TMO_FEVR(= -1)によりタイムアウトしない、すなわち rcv_mbx と同じ動作になります。

戻値
E_OK　正常終了
E_ID　メールボックス ID が範囲外*
E_NOEXS　メールボックスが生成されていない
E_CTX　非タスクコンテキストで、または、ディスパッチ禁止状態で待ち実行*
E_RLWAI　待ち状態を強制解除された(待ちの間に rel_wai を受け付け)
E_TMOUT　タイムアウト

注意 ppk_msg は、ポインタへのポインタです。

例
```
#define ID_mbx1 1
TASK task1(void)
{
    T_MYMSG *msg;
    ER ercd;
        :
    ercd = trcv_mbx(ID_mbx1, (T_MSG **)&msg, 1000/MSEC);
    if (ercd == E_OK)
        :
}
```

ref_mbx

機能　メールボックス状態参照

形式　ER ref_mbx(ID mbxid, T_RMBX *pk_rmbx);
mbxid　メールボックス ID
pk_rmbx　メールボックス状態パケットを格納する場所へのポインタ

解説　mbxid で指定されたメールボックスの状態を、*pk_rmbx に返します。メールボックス状態パケットの構造は次の通りです。

```
typedef struct t_rmbx {
    ID wtskid;      待ちタスク ID または TSK_NONE
    T_MSG *pk_msg;  先頭の送信待ちメッセージアドレスまたは NULL
} T_RMBX;
```

wtskid には、待ちタスクがある場合、その先頭の待ちタスクの ID 番号が返ります。待ちタスクがない場合は、TSK_NONE が返ります。

戻値　E_OK　正常終了
E_ID　メールボックス ID が範囲外
E_NOEXS　メールボックスが生成されていない

例
```
#define ID_mbx1 1
TASK task1(void)
{
    T_RMBX rmbx;
        :
    ref_mbx(ID_mbx1, &rmbx);
    if (rmbx.pk_msg != NULL)
        :
}
```

5.7 割り込み管理機能

def_inh

機能 割り込みハンドラ定義

形式 ER def_inh(INHNO inhno, const T_DINH *pk_dinh);
inhno 割り込みハンドラ番号
pk_dinh 割り込みハンドラ定義情報パケットへのポインタ

解説 inhno で指定される割り込みベクタテーブルに、inthdr で示される割り込みハンドラを設定します。割り込みベクタテーブルが使えないプロセッサでは、配列変数として定義した割り込みハンドラテーブルへ、inthdr を設定します。inhno の内容はプロセッサにより異なります(割り込みベクタ番号が一般的)。

割り込みハンドラ定義情報パケットの構造は次の通りです。プロセッサによっては、割り込みハンドラ開始時の割り込みマスク imask が追加されている場合があります。

```
typedef struct t_dinh {
    ATR inhatr;  割り込みハンドラ属性(未使用)
    FP inthdr;   割り込みハンドラとする関数へのポインタ
    UINT imask;  割り込みマスク(一部のプロセッサのみ)
} T_DINH;
```

inhatr の値は NORTi Oceans では参照していませんが、他社μITRON との互換のためには、ハンドラが高級言語で記述されていることを示す TA_HLNG を入れておくことを推奨します。

プロセッサに依存しますので、カーネルとは別の nocixxx.c にサンプルが記述されています。ユーザーのシステムに適合しない場合は、独自の def_inh を作成してください。μITRON 仕様では、pk_dint = NULL で、割り込みハンドラの定義を解除する仕様となっていますが、組込みシステムでは意味を持たないため、この機能は実装しなくても構いません。

割り込みベクタテーブル領域を ROM に割り当てる場合、このシステムコールは機能しません。割り込みハンドラのアドレスを、直接、割り込みベクタテーブルに記述してください。

戻値 E_OK 正常終了
E_PAR 割り込みハンドラ番号 inhno が範囲外*

ent_int

機能 割り込みハンドラ開始

形式 void ent_int(void);

解説 割り込み発生時のレジスタ類を保存し、スタックポインタを割り込みハンドラ専用領域に切り替えます。必ず割り込みハンドラの先頭で呼び出してください。

スタックポインタがずれてしまいますので、割り込みハンドラ入り口で、auto 変数は定義できません。static 変数にするか、割り込みハンドラからさらに関数を呼んで、そこに auto 変数を定義してください。

また、アセンブラレベルで見ると、ent_int をコールする前の部分に、レジスタを破壊するようなコードが展開される場合があります。この場合には、最適化をかけてコンパイルするか、実際の処理を割り込みハンドラから呼ばれる関数へ移すことで、このコード展開を抑制してください。

マルチタスク動作に関与しない割り込みルーチン(マルチタスク動作に関与する他の割り込みハンドラの優先度以上であること)では、この ent_int と次の ret_int システムコールを使わなくても構いません。その場合、コンパイラの拡張機能である interrupt 関数の機能を使うか、ユーザーが独自に、アセンブラでレジスタを保存／復元してください。

戻値 なし

補足 割り込みハンドラを C で記述できるようにするための、NORTi Oceans 独自のシステムコールです。

例

```
void func(void)←(注)最適化で inthdr 内部にインライン展開されないこと
{
    int c;
        :
}

INTHDR inthdr(void)
{
    ent_int();
    func();
    ret_int();
}
```

ret_int

機能 割り込みハンドラから復帰

形式 void ret_int(void);

解説 割り込みハンドラを終了します。必ず割り込みハンドラの最後で呼び出してください。

割り込みハンドラ内で発行したシステムコールによるタスク切り替えは、この ret_int が発行されるまで遅延させられます(遅延ディスパッチ)。

戻値 なし(呼び出し元に戻りません)

例

```
INTHDR inthdr(void)
{
    ent_int();
        :
    ret_int();
}
```

chg_ims

機能 割り込みマスク変更

形式 ER chg_ims(UINT imask);
imask 割り込みマスク値

解説 プロセッサの割り込みマスクを、imask で指定した値に変更します。

割り込み禁止／許可の 2 状態しかないプロセッサでは、imask = 0 で割り込み許可、imask != 0 で割り込み禁止を指定します。

レベル割り込み機能のあるプロセッサでは、imask に割り込みマスクレベルを指定します(0 で割り込み許可、1～で割り込み禁止)。imask 値の範囲はチェックしていません。

割り込み禁止中に発行されたシステムコールで、タスク切り替えが必要となった場合は、chg_ims(0)が発行されて割り込み許可となる時に、タスクの切り替えが行われます(遅延ディスパッチ)。

戻値 E_OK 正常終了

get_ims

機能 割り込みマスク参照

形式 ER get_ims(UINT *p_imask);
p_imask 割り込みマスク値を格納する場所へのポインタ

解説 プロセッサの割り込みマスクを参照し、*p_imask に返します。

割り込み禁止／許可の 2 状態しかないプロセッサでは、*p_imask = 0 で割り込み許可状態、*p_imask = 1 で割り込み禁止状態を示します。

レベル割り込み機能のあるプロセッサでは、*p_imask の値で割り込みマスクレベルを示します。

戻値 E_OK 正常終了

vdis_psw

機能 ステータスレジスタの割り込みマスクセット

形式 UINT vdis_psw(void);

解説 プロセッサのステータスレジスタにある割り込みマスクを、割り込み禁止状態にセットします。レベル割り込みの機能を持ったプロセッサでは、最高の割り込みレベルに設定して、全割り込みを禁止します。

戻値として、この操作の前のプロセッサのステータスレジスタ値を返します。

戻値 割り込み禁止前のプロセッサのステータスレジスタ値

補足 NORTi Oceans 独自のシステムコールです。vset_psw と組合せて、一時的な割り込み禁止をおこなうのに便利です。カーネルより高優先の割り込みルーチンからも発行できます。

例

```
void func(void)
{
    UINT psw;
    psw = vdis_psw();     割り込み禁止
        :
    vset_psw(psw);        割り込み禁止/許可状態を元に戻す
        :
}
```

同じことを chg_ims で実現するためには...

```
void func(void)
{
    UINT imask;
    get_ims(&imask); 割り込み禁止/許可を調べる
    chg_ims(7);      割り込み禁止(imask 値はプロセッサ依存)
        :
    chg_ims(imask);  割り込み禁止/許可状態を元に戻す
}
```

vset_psw

機能 ステータスレジスタのセット

形式 void vset_psw(UINT psw);
psw プロセッサのステータスレジスタ値

解説 プロセッサのステータスレジスタを psw で指定した値に設定します。vdis_psw システムコールの戻値を psw とすれば、割り込みマスクの完全な復元がおこなえます。

chg_ims(0)との違いは、遅延されたディスパッチがあっても実行されないことです。したがって vdis_psw～vset_psw の区間では、タスク切り替えが発生するようなシステムコールを発行できません。

戻値 なし

補足 NORTi Oceans 独自のシステムコールです。割り込みマスクだけではなく、ステータスレジスタの全ビットが操作できます。カーネルより高優先の割り込みルーチンからも発行できます。

例

```
void func(void)
{
    UINT psw;
    psw = vdis_psw();
        :
    vset_psw(psw | 0x8000);
        :
}
```

5.8 メモリプール管理機能（固定長）

cre_mpf

機能 固定長メモリプール生成

形式 ER cre_mpf(ID mpfid, const T_CMPF *pk_cmpf);
mpfid 固定長メモリプール ID
pk_cmpf 固定長メモリプール生成情報パケットへのポインタ

解説 mpfid で指定された固定長メモリプールを生成します。すなわち、システムメモリから固定長メモリプール管理ブロックを動的に割り当て、また pk_cmpf->mpf が NULL の場合メモリプール用メモリから、blkcnt×blksz で指定されたサイズだけ、メモリプール領域を動的に割り当てます。ユーザープログラムでメモリプール領域を確保した場合はその先頭アドレスを pk_cmpf->mpf に設定してください。

固定長メモリプール生成情報パケットの構造は次の通りです。

```
typedef struct t_cmpf {
    ATR mpfatr;      固定長メモリプール属性(未使用)
    UINT blkcnt;     メモリプール全体のブロック数
    UINT blksz;      固定長メモリブロックサイズ(バイト数)
    VP mpf;          メモリプール先頭番地または NULL
    B *name;         メモリプール名へのポインタ（省略可）
} T_CMPF;
```

獲得待ちタスク行列は常に先着順（FIFO）となります。つまり、固定長メモリプール属性 mpfatr は無視されますが、NORTi Ver.4 との互換性のために次の値を入れておくことを推奨します。

TA_TFIFO 獲得待ちタスク行列は先着順(FIFO)

メモリブロックサイズ blksz の最小値は、処理系のポインタのサイズ以上です。また、ワードのアラインメントの必要なプロセッサでは、blksz を int のサイズの整数倍としてください。（整数倍とせず端数のあるサイズを指定した場合は、内部で切り上げられます。）

サイズが blksz のメモリブロック獲得によって消費されるメモリプールのサイズは blksz に等しく、無駄がありません。

name は対応デバッガ用ですので、名前を指定しない場合には""か NULL を入れてください。この構造体を初期値付きで定義する場合には、name を省略しても構いません。

戻値	E_OK	正常終了
	E_ID	固定長メモリプール ID が範囲外*
	E_PAR	固定長メモリプール生成情報が不正*
	E_OBJ	固定長メモリプールが既に生成されている
	E_CTX	割り込みハンドラから発行*
	E_SYS	管理ブロック用のメモリが確保できない**
	E_NOMEM	メモリプール用のメモリが確保できない**

例

```
#define ID_mpf1 1
const T_CMPF cmpf1 = {TA_TFIFO, 10, 256, NULL};

TASK task1(void)
{
    ER ercd;
            :
    ercd = cre_mpf(ID_mpf1, &cmpf1);
            :
}
```

acre_mpf

機能　固定長メモリプール生成(ID 自動割り当て)

形式　ER_ID acre_mpf(const T_CMPF *pk_cmpf);
pk_cmpf　固定長メモリプール生成情報パケットへのポインタ

解説　未生成固定長メモリプールの ID を、大きな方から検索して割り当てます。固定長メモリプール ID が割り当てられない場合は、E_NOID エラーを返します。それ以外は、cre_mpf と同じです。

戻値　正の値　割り当てられた固定長メモリプール ID
E_NOID　固定長メモリプール ID が不足
E_CTX　割り込みハンドラから発行*
E_SYS　管理ブロック用のメモリが確保できない**
E_NOMEM　メモリプール用のメモリが確保できない**

例
```
ID ID_mpf1;
const T_CMPF cmpf1 ={TA_TFIFO, 10, 256, NULL};

TASK task1(void)
{
    ER_ID ercd;
        :
    ercd = acre_mpf(&cmpf1);
    if (ercd > 0)
    ID_mpf1 = ercd;
        :
}
```

get_mpf

機能　固定長メモリブロック獲得

形式　ER get_mpf(ID mpfid, VP *p_blf);
mpfid　固定長メモリプール ID
p_blf　メモリブロックへのポインタを格納する場所へのポインタ

解説　mpfid で指定された固定長メモリプールから、1 個のメモリブロックを獲得して、そのメモリブロックへのポインタを*p_blf に返します。メモリブロックのサイズは、固定長メモリブロック生成で指定した blksz に固定です。獲得したメモリブロックのゼロクリアは行われません。内容は不定です。

固定長メモリプールに空きブロックがない場合、本システムコールの発行タスクは、その固定長メモリプールの待ち行列につながれます。

戻値　E_OK　正常終了
E_ID　固定長メモリプール ID が範囲外*
E_NOEXS　固定長メモリプールが生成されていない
E_CTX　非タスクコンテキストから発行、または、ディスパッチ禁止状態で待ち実行*
E_RLWAI　待ち状態を強制解除された(待ちの間に rel_wai を受け付け)

注意　p_blf は、ポインタへのポインタです。

補足　tget_mpf(mpfid, p_blf, TMO_FEVR)と同じです。

例
```
#define ID_mpf1 1

TASK task1(void)
{
    B *blf;
        :
    get_mpf(ID_mpf1, (VP *)&blf);
    blf[0] = 0;
    blf[1] = 1;
        :
}
```

pget_mpf

機能　固定長メモリブロック獲得(ポーリング)

形式　ER pget_mpf(ID mpfid, VP *p_blf);
mpfid　固定長メモリプール ID
p_blf　メモリブロックへのポインタを格納する場所へのポインタ

解説　get_mpf との違いは次の通りです。

固定長メモリプールの空きブロックがない場合、待ち状態に入らずに、E_TMOUT エラーを返します。

戻値　E_OK　正常終了
E_ID　固定長メモリプール ID が範囲外*
E_NOEXS　固定長メモリプールが生成されていない
E_TMOUT　ポーリング失敗

注意　p_blf は、ポインタへのポインタです。

補足　tget_mpf(mpfid, p_blf, TMO_POL)と同じです。

例
```
#define ID_mpf1 1

TASK task1(void)
{
    B *blf;
    ER ercd;
        :
    ercd = pget_mpf(ID_mpf1, (VP *)&blf);
    if (ercd == E_OK)
        :
}
```

tget_mpf

機能　固定長メモリブロック獲得(タイムアウト有)

形式　ER tget_mpf(ID mpfid, VP *p_blf, TMO tmout);
mpfid　固定長メモリプール ID
p_blf　メモリブロックへのポインタを格納する場所へのポインタ
tmout　タイムアウト値

解説　get_mpf との違いは次の通りです。

tmout で指定した時間が経過してもメモリブロックが獲得できない場合は、タイムアウトエラーE_TMOUT としてリターンします。

tmout = TMO_POL(= 0)により待ちをおこわない、すなわち pget_mpf と同じ動作になります。tmout = TMO_FEVR(= -1)によりタイムアウトしない、すなわち get_mpf と同じ動作になります。

戻値　E_OK　正常終了
E_ID　固定長メモリプール ID が範囲外*
E_NOEXS　固定長メモリプールが生成されていない
E_CTX　非タスクコンテキストから発行、または、ディスパッチ禁止状態で待ち実行*
E_RLWAI　待ち状態を強制解除された(待ちの間に rel_wai を受け付け)
E_TMOUT　タイムアウト

例
```
#define ID_mpf1 1

TASK task1(void)
{
    B *blf;
    ER ercd;
        :
    ercd = tget_mpf(ID_mpf1, (VP *)&blf, 100/MSEC);
    if (ercd == E_OK)
        :
}
```

rel_mpf

機能　固定長メモリブロック返却

形式　ER rel_mpf(ID mpfid, VP blf);
mpfid　固定長メモリプール ID
blf　メモリブロックへのポインタ

解説　mpfid で指定された固定長メモリプールへ、blf で指し示されるメモリブロックを返却します。この固定長メモリプールでメモリブロック獲得を待っているタスクがあれば、先頭の待ちタスクにメモリブロックを獲得させ、待ちを解除します。

1 回の返却で複数のタスクのメモリブロック獲得待ちが解除されることはありません。

このシステムコールを発行したタスクが、待ち状態となることはありません。メモリブロックは、必ず、獲得した元のメモリプールへ返却してください。

戻値　E_OK　正常終了
E_PAR　異なるメモリプールへの返却
E_ID　固定長メモリプール ID が範囲外*
E_NOEXS　固定長メモリプールが生成されていない

例
```
#define ID_mpf1 1
TASK task1(void)
{
    B *blf;
        :
    get_mpf(ID_mpf1, (VP *)&blf);
        :
    rel_mpf(ID_mpf, (VP)blf);
        :
}
```

ref_mpf

機能　固定長メモリプール状態参照

形式　ER ref_mpf(ID mpfid, T_RMPF *pk_rmpf);
mpfid　固定長メモリプール ID
pk_rmpf　固定長メモリプール状態パケットを格納する場所へのポインタ

解説　mpfid で指定された固定長メモリプールの状態を、*pk_rmpf に返します。

固定長メモリプール状態パケットの構造は次の通りです。

```
typedef struct t_rmpf
{   ID wtskid;      待ちタスクの ID または TSK_NONE
    UINT fblkcnt;   空きメモリブロック数
}T_RMPF;
```

wtskid には、待ちタスクがある場合、その先頭の待ちタスクの ID 番号が返ります。待ちタスクがない場合は、TSK_NONE が返ります。

戻値　E_OK　正常終了
E_ID　固定長メモリプール ID が範囲外
E_NOEXS　固定長メモリプールが生成されていない

例
```
#define ID_mpf1 1
TASK task1(void)
{
    T_RMPF rmpf;
        :
    ref_mpf(ID_mpf1, &rmpf);
    if (rmpf.fblkcnt > 0)
        :
}
```

5.9 メモリ確保機能

vget_mem

機能　システムメモリからのメモリ確保

形式　VP vget_mem(SIZE size);
size　確保するサイズ

解説　システムメモリからメモリを確保します。
システムメモリについては、「2.2 カーネルコンフィグレーション」の「システムメモリと管理ブロックのサイズ」を参照してください。

補足　NORTi Oceans 独自のシステムコールです。

戻値　NULL 以外　確保した領域へのポインタ
NULL　確保失敗

vget_mpl

機能 メモリプール用メモリからのメモリの確保

形式 ER vget_mpl(SIZE size);
size 確保するサイズ

解説 メモリプール用メモリからメモリを確保します。

メモリプール用メモリについては、「2.2 カーネルコンフィグレーション」の「メモリプール用メモリのサイズ」を参照してください。

補足 NORTi Oceans 独自のシステムコールです。

戻値 NULL 以外 確保した領域へのポインタ
NULL 確保失敗

5.10 時間管理機能

set_tim

機能　システム時刻設定

形式　ER set_tim(SYSTIM *p_systim);
p_systim　現在時刻パケットへのポインタ

解説　時間管理をおこなうシステムクロックを、*p_systim で示される値に変更します。

時刻パケットの構造は次の通りです。

```
typedef struct {
    H utime;     上位 16 ビット
    UW ltime;    下位 32 ビット
} SYSTIM;
```

set_tim されたシステム時刻を周期割り込みごとにカウントアップします。したがって、システム時刻は、周期割り込みの回数をカウントしたデータです。msec 等の時刻の単位との変換は、ユーザー側でおこなう必要があります。

システムクロックは、システム起動時に 0 クリアされその後カウントアップされる絶対時刻を表すのに対して、システム時刻は set_tim により初期化される相対時刻です。タイムイベントハンドラはシステムクロックを基準にしているため set_tim により影響を受けません。

戻値　E_OK　正常終了

例
```
SYSTIM tim;
    :
tim.utime = 0;
tim.ltime = 12345L;
set_tim(&tim);
    :
```

get_tim

機能　システム時刻参照

形式　ER get_tim(SYSTIM *p_systim);
p_systim　現在時刻パケットを格納する場所へのポインタ

解説　システム時刻の現在の値を、*p_systim に返します。

時刻パケットの構造は set_tim システムコールと同じです。

```
typedef struct {
    H utime;    上位 16 ビット
    UW ltime;   下位 32 ビット
} SYSTIM;
```

システム時刻は、周期割り込みの回数をカウントしたデータです。msec 等の時刻の単位との変換は、ユーザー側でおこなう必要があります。

戻値　E_OK　正常終了

例
```
SYSTIM tim;
    :
get_tim(&tim);
if (tim.ltime == 10000L)
    :
```

cre_cyc

機能　周期ハンドラ生成

形式　ER cre_cyc(ID cycid, const T_CCYC *pk_ccyc);
cycid　周期ハンドラ ID
pk_ccyc　周期ハンドラ生成情報パケットへのポインタ

解説　cycid で指定される周期ハンドラを生成します。すなわち、システムメモリから、周期ハンドラ管理ブロックを動的に割り当てます。

周期ハンドラ生成情報パケットの構造は次の通りです

```
typedef struct t_ccyc {
    ATR cycatr;      周期ハンドラ属性
    VP_INT exinf;    拡張情報
    FP cychdr;       周期ハンドラとする関数へのポインタ
    RELTIM cyctim;   周期起動時間間隔
    RELTIM cycphs;   周期ハンドラ起動位相
} T_CCYC;
```

cycatr には、次の指定をできます。TA_STA と TA_PHS が不要な場合は TA_HLNG 属性のみを指定してください。

TA_HLNG　他社μITRON との互換のためには高級言語を示すこれを定義してください。
TA_STA　ハンドラ生成と同時に動作状態とします。
TA_PHS　ハンドラの起動位相を保存します。

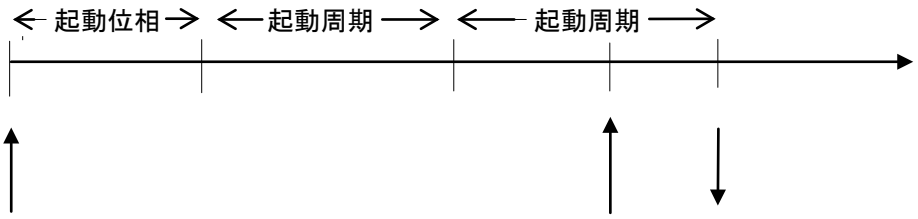
起動位相を保存しない場合、ハンドラ動作を開始した時点で周期を初期化します。したがって、最初のハンドラ起動はつねに動作開始から起動周期経過後になります。位相を保存した場合、生成した時点から動作の有無に関係なく計時をおこないます。

exinf に指定した値は、ハンドラ起動時に第一パラメータとして渡されます。

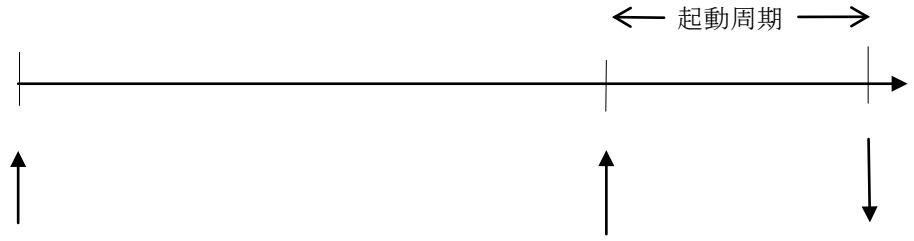
cychdr は、周期ハンドラとする関数へのポインタです。周期ハンドラは、void 型の関数として記述してください。

cyctim は、周期起動の時間間隔です。時間の単位は、システムクロックの割り込み周期です。

cycphs には、ハンドラの動作を開始してから最初に起動するまでの時間を設定してください。二回目以降の起動は cyctim 間隔になります。



cre_cyc(TA_PHS 指定あり) sta_cyc 周期ハンドラ起動



cre_cyc(TA_PHS 指定なし) sta_cyc 周期ハンドラ起動

戻値

E_OK	正常終了
E_ID	周期ハンドラ ID 番号が範囲外*
E_PAR	pk_ccyc が NULL*
E_CTX	割り込みハンドラから発行*
E_SYS	管理ブロック用のメモリが確保できない**

例

```
#define ID_cyc1 1
extern void cyc1(VP_INT);
const T_CCYC ccyc1 ={TA_HLNG|TA_STA, NULL, cyc1, 10, 5};

TASK task1(void)
{
    ER ercd;
        :
    ercd = cre_cyc(ID_cyc1, &ccyc1);
        :
}
```

acre_cyc

機能　周期ハンドラ生成(ID 自動割り当て)

形式　ER_ID acre_cyc(const T_CCYC *pk_ccyc);
pk_ccyc　周期ハンドラ生成情報パケットへのポインタ

解説　未生成の周期ハンドラ ID を、大きな方から検索して割り当てます。周期ハンドラ ID が割り当てられない場合は、E_NOID エラーを返します。それ以外は、cre_cyc と同じです。

戻値　正の値　割り当てられた周期ハンドラ ID
E_NOID　周期ハンドラ ID が不足
E_PAR　pk_ccyc が NULL *
E_CTX　割り込みハンドラから発行*
E_SYS　管理ブロック用のメモリが確保できない**

例
```
ID ID_cyc1;
extern void cyc1(VP_INT);
const T_CCYC ccyc1 = {TA_HLNG|TA_STA, NULL, cyc1, 10, 5};

TASK task1(void)
{
    ER_ID ercd;
        :
    ercd = acre_cyc(&ccyc1);
    if (ercd > 0)
    ID_cyc1 = ercd;
        :
}
```

sta_cyc

機能 周期ハンドラ動作開始

形式 ER sta_cyc(ID cycid);
cycid 周期ハンドラ ID 番号

解説 cycid で指定される周期ハンドラを動作状態にします。

TA_PHS 指定の無い場合は sta_cyc 呼出しから起動周期経過後にハンドラを起動します。TA_PHS を指定した場合で、すでに動作状態の場合は何もしません。TA_PHS を指定した場合で停止状態の場合は、起動時刻の変更はせずに起動可能状態とします。TA_PHS を指定した場合は、起動可能か否かにかかわらず起動時間の更新を継続しておこなっています。

戻値 E_OK 正常終了
E_ID 周期ハンドラ ID が範囲外*
E_NOEXS 周期ハンドラが生成されていない

stp_cyc

機能 周期ハンドラ動作停止

形式 ER stp_cyc(ID cycid);
cycid 周期ハンドラ ID 番号

解説 cycid で指定される周期ハンドラ動作していない状態にします。すでに停止しているハンドラが指定された場合は何もしません。

生成時に TA_PHS を指定した場合には、起動時刻の更新を継続します。

戻値 E_OK 正常終了
E_ID 周期ハンドラ ID が範囲外*
E_NOEXS 周期ハンドラが生成されていない

ref_cyc

機能　周期ハンドラ状態参照

形式　ER ref_cyc(ID cycid, T_RCYC *pk_rcyc);
cycid　周期ハンドラ ID
pk_rcyc　周期ハンドラ状態パケットを格納する場所へのポインタ

解説　cycid で指定される周期ハンドラの状態を、*pk_rcyc に返します。

周期ハンドラ状態パケットの構造は次の通りです。

```
typedef struct t_rcyc {
    STAT cycstat;    ハンドラの動作状態
    RELTIM lefttim;  次回起動までの残り時間
} T_RCYC;
```

cycstat には、動作状態に応じて次の値が入ります。

TCYC_STP　ハンドラは動作していない
TCYC_STA　ハンドラは動作している

lefttim の単位は、システムクロックの割り込み周期です。

戻値　E_OK　正常終了
E_ID　周期ハンドラ ID が範囲外
E_NOEXS　周期ハンドラが生成されていない

例

```
#define ID_cyc 1

TASK task1(void)
{
    T_RCYC rcyc;
        :
    ref_cyc(ID_cyc, &rcyc);
    if (rcyc.cycstat == TCYC_STA)
        :
}
```

isig_tim

機能 チックタイムの経過通知

形式 void isig_tim(void);

解説 OS に、周期タイマ割り込みが入ったことを知らせます。割り込みハンドラ専用です。

戻値 なし

補足 NORTi Oceans 独自のシステムコールです。

例
```
INTHDR inthdr(void)
{
    ent_int();
    isig_tim();
    ret_int();
}
```

5.11 システム状態管理機能

rot_rdq, irot_rdq

機能 タスクのレディキュー回転

形式
```
ER rot_rdq(PRI tskpri);
ER irot_rdq(PRI tskpri);
tskpri      優先度
```

解説 tskpri で指定された優先度のレディキューにおいて、先頭につながれているタスクを最後尾へつなぎ替えます。つまり、同一優先度のタスクの実行を切り替えます。

tskpri = TPRI_SELF で、自タスクのベース優先度を対象優先度とします。

このシステムコールを周期ハンドラから一定間隔で発行することにより、ラウンドロビン・スケジューリングが実現できます。

このシステムコールを発行したタスクのレディキューが回転する場合、このタスクは RUNNING 状態から READY 状態へ遷移し、次に実行順序を待っていたタスクが READY 状態から RUNNING 状態へ遷移します。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ます。

指定した優先度のレディキューにタスクがひとつ、あるいは無いい場合は何もしませんが、エラーとはなりません。

戻値
```
E_OK      正常終了
E_PAR     優先度が範囲外*
```

例
```
TASK task1(void)
{
        :
    rot_rdq(TPRI_SELF);
        :
}
```

get_tid, iget_tid

機能　実行タスクのタスク ID 参照

形式　ER get_tid(ID *p_tskid);
ER iget_tid(ID *p_tskid);
p_tskid　タスク ID を格納する場所へのポインタ

解説　このシステムコールを発行したタスクの ID 番号を、*p_tskid に返します。割り込みハンドラなどの非タスクコンテキスト部から呼ばれた場合には現在 RUNNING 状態にあるタスクの ID を返します。RUNNING 状態のタスクが無い場合は TSK_NONE を返します。

戻値　E_OK　正常終了

例
```
TASK task1(void)
{
    ID tskid;
        :
    get_tid(&tskid);
        :
}
```

vget_tid

機能 自タスクのタスク ID を得る

形式 ID vget_tid(void);

解説 このシステムコールを発行したタスクの ID 番号を、関数戻り値として返します。その他は get_tid と同一です。

戻値 タスク ID

補足 NORTi Oceans 独自のシステムコールです。

例
```
TASK task1(void)
{
    ID tskid;
        :
    tskid = vget_tid();
        :
}
```

loc_cpu, iloc_cpu

機能 CPU ロック状態への移行(割り込みとディスパッチの禁止)

形式
```
ER loc_cpu(void);
ER iloc_cpu(void);
```

解説 割り込みの受付とタスク切り替えを禁止します。この禁止状態は、unl_cpu システムコールで解除されます。

すでに、CPU ロック状態にあるとき、このシステムコールを発行してもエラーとはなりません。ただし、loc_cpu～unl_cpu の対はネスト管理されていませんので、複数回の loc_cpu に対して、1 回の unl_cpu で禁止状態が解除されてしまいます。

割り込みハンドラからは、このシステムコールを発行しないでください。割り込みハンドラ以外の非タスクコンテキストから CPU ロック状態に移行した場合は、復帰する前に CPU ロック解除状態にしてください。

戻値

E_OK	正常終了

補足 レベル割り込み機能のあるプロセッサの場合、NORTi Oceans では、カーネルの割り込み禁止レベルを標準で最高とはしていません。loc_cpu で設定される割り込みマスクは、カーネルの割り込み禁止レベルまでを禁止するものであり、カーネルより高優先の割り込みは受付られます。

unl_cpu, iunl_cpu

機能 CPU ロック状態の解除

形式
```
ER unl_cpu(void);
ER iunl_cpu(void);
```

解説 loc_cpu で設定された禁止状態を解除します。ただし、割り込みの受付とタスク切り替えが許可されるとは限りません。loc_cpu を発行した時点ですでに dis_dsp によりディスパッチ禁止であればディスパッチは禁止されたままになります。この場合、ディスパッチ可能とするためには ena_dsp によらなければなりません。

すでに、CPU ロック解除状態にあるとき、このシステムコールを発行してもエラーとはなりません。ただし、loc_cpu～unl_cpu の対はネスト管理されていませんので、複数回の loc_cpu に対して、1 回の unl_cpu で禁止状態が解除されてしまいます。

非タスクコンテキストのうちタイマイベントハンドラから iunl_cpu を呼ぶことは可能ですが、割り込みハンドラからはこのシステムコールを発行しないでください。全割り込みマスクが解除されてしまいます。レベル割り込みをサポートしているプロセッサの場合、割り込みハンドラでは ent_int から戻ってきた時点で、割り込みサービスルーチンでは割り込みサービスルーチンが呼ばれた時点でその割り込みレベルまで割り込みマスクは下がっています。

戻値

E_OK	正常終了

dis_dsp

機能 ディスパッチ禁止

形式 ER dis_dsp(void);

解説 タスクの切り替えを禁止します。割り込みは禁止されません。このシステムコールを発行した後の、他システムコール発行によるタスクの切り替えは保留されます。保留されたタスクの切り替えは、ena_dsp システムコールを発行した時に実行されます。

注意 ディスパッチ禁止の間は、待ちの生じるシステムコールを発行すると E_CTX エラーになります。

戻値

E_OK	正常終了
E_CTX	非タスクコンテキスト部からの発行*

例

```
TASK task1(void)
{
        :
    dis_dsp();
        :       /* ディスパッチ禁止区間 */
    ena_dsp();
        :
}
```

ena_dsp

機能 ディスパッチ許可

形式 ER ena_dsp(void);

解説 dis_dsp システムコールにより設定されていた、ディスパッチ禁止状態を解除します。先に dis_dsp を発行していなくてもエラーとはしません。ディスパッチ禁止状態で保留されていたタスクの切り替えがあれば、このシステムコールで実行されます。

戻値

E_OK	正常終了
E_CTX	非タスクコンテキストからの発行*

sns_ctx

機能 コンテキスト参照

形式 BOOL sns_ctx(void);

解説 非タスクコンテキスト部から呼ばれた場合に TRUE、タスクコンテキスト部から呼ばれた場合に FALSE を返します。

戻値

TRUE	非タスクコンテキスト
FALSE	タスクコンテキスト部

sns_loc

機能　CPU ロック状態参照

形式　BOOL sns_loc(void);

解説　CPU ロック状態の場合に TRUE、CPU ロック解除状態の場合に FALSE を返します。

戻値　TRUE　CPU ロック状態
　　FALSE　CPU ロック解除状態

例
```
BOOL cpu_lock = sns_loc();
        :
if (!cpu_lock)
    loc_cpu();
        :
/* CPU ロック状態でおこなわせたい処理 */
        :
if (!cpu_lock)   /* 不用意にロック解除しないため */
    unl_cpu();
        :
```

sns_dsp

機能　ディスパッチ禁止状態参照

形式　BOOL sns_dsp(void);

解説　ディスパッチ禁止状態の場合に TRUE、ディスパッチ許可状態の場合に FALSE を返します。

戻値　TRUE　ディスパッチ禁止状態
　　FALSE　ディスパッチ許可状態

例
```
BOOL task_lock = sns_dsp();
        :
if (!task_lock)
    dis_dsp();
        :
/* ディスパッチ禁止状態でおこなわせたい処理 */
        :
if (!task_lock)  /* 不用意にディスパッチ許可しないため */
    ena_dsp();
        :
```

sns_dpn

機能 ディスパッチ保留状態参照

形式 BOOL sns_dpn(void);

解説 CPU ロック状態またはディスパッチ禁止の場合に TRUE、そうでない場合に FALSE を返します。

戻値

TRUE	ディスパッチ保留状態
FALSE	ディスパッチ可能状態

ref_sys

機能　システム状態参照

形式　ER ref_sys(T_RSYS *pk_rsys);
pk_rsys　システム状態パケットを格納する場所へのポインタ

解説　OS の実行状態を、*pk_rsys に返します。

システム状態パケットの構造は次の通りです。

```
typedef struct t_rsys {
    INT sysstat;        システム状態
} T_RSYS;
```

sysstat には、次の値が返ります。

TSS_TSK　タスクコンテキスト部を実行中で、ディスパッチを許可した状態
TSS_DDSP　タスクコンテキスト部を実行中で、ディスパッチを禁止した状態
TSS_LOC　タスクコンテキスト部を実行中で、割り込みとディスパッチを禁止した状態
TSS_INDP　非タスクコンテキスト部を実行中

戻値　E_OK　正常終了

例
```
TASK task1(void)
{
    T_RSYS rsys;
        :
    ref_sys(&rsys);
    if (rsys.sysstat == TSS_LOC)
        :
}
```

5.12 システム構成管理機能

ref_ver

機能　　バージョン参照

形式　　ER ref_ver(T_RVER *pk_rver);
　　pk_rver　　バージョン情報パケットを格納する場所へのポインタ

解説　　NORTi Oceans のバージョンを、*pk_ver に返します。

　　バージョン情報パケットの構造は次の通りです。

```
typedef struct t_rver {
    UH maker;    メーカー          (0108H:株式会社ミスポ)
    UH prid;     形式番号          (4000H:NORTi Oceans カーネル)
    UH spver;    仕様書バージョン  (5400H:μITRON Ver.4.00)
    UH prver;    製品バージョン    (NORTi Oceans カーネル Version に依存)
    UH prno[4];  製品管理情報      (未使用)
} T_RVER;
```

戻値　　E_OK　　正常終了

ref_cfg

機能　コンフィグレーション情報参照

形式　ER ref_cfg(T_RCFG *pk_rcfg);
pk_rcfg　コンフィグレーション情報パケットを格納する場所へのポインタ

解説　コンフィグレーション情報を、*pk_rcfg に返します。

このコンフィグレーション情報パケットの構造は、NORTi Oceans 独自です。

```
typedef struct t_rcfg {
    ID tskid_max;    タスク ID 上限
    ID semid_max;    セマフォ ID 上限
    ID flgid_max;    イベントフラグ ID 上限
    ID mbxid_max;    メールボックス ID 上限
    ID mpfid_max;    固定長メモリプール ID 上限
    ID cycno_max;    周期ハンドラ ID 上限
    PRI tpri_max;    タスク優先度上限
    int tmrqsz;      タスクのタイマキューサイズ(バイト数)
    int cycqsz;      周期ハンドラのタイマキューサイズ(バイト数)
    int istksz;      割り込みハンドラのスタックサイズ(バイト数)
    int tstksz;      タイムイベントハンドラのスタックサイズ(バイト数)
    SIZE sysmsz;     システムメモリのサイズ(バイト数)
    SIZE mplmsz;     メモリプール用メモリのサイズ(バイト数)
    SIZE stkmsz;     スタック用メモリのサイズ(バイト数)
    ID dtqid_max;    データキューID 上限
        :(今後、追加される可能性があります)
} T_RCFG;
```

戻値　E_OK　正常終了

第６章 独自システム関数

sysini

機能 システム初期化

形式 ER sysini(void);

解説 カーネルを初期化します。他の全てのシステムコールに先だって、実行する必要があります。通常は、main 関数の先頭で呼び出してください。

ここで行われる初期化作業は、カーネルの内部変数の初期設定と、後述の intini 関数の呼び出しです。sysini 実行後は、割り込み禁止状態となります。

スタック用メモリとして、コンパイラが提供する標準的なスタック領域を使う場合、すなわち、コンフィグレーションで#define STKMSZ 0 とした場合、確保されるスタックの底は、この sysini を呼び出した時点のスタックポインタが基準となります。

戻値 E_OK 正常終了
E_SYS 管理ブロック用のメモリ不足**
E_NOMEM スタック用のメモリ不足**
その他、intini 関数の戻値

syssta

機能 システム起動

形式 ER syssta(void);

解説 初期化ハンドラを終了して、マルチタスク状態へと移行します。このシステムコールを発行する前に、少なくとも 1 個以上のタスクの生成と起動がおこなわれていなければなりません。通常は、main 関数の最後で呼び出してください。

起動されたタスクの中で、最も優先度の高いタスク(同優先度ならば、先に起動されたタスク)に制御が移ります。つまり、最初のディスパッチがおこなわれます。それに伴い、sysini で禁止されていた割り込みが、ここで許可されます。

syssta 実行前に、タスク生成等でエラーが発生していた場合は、システムを起動せずにエラーを返します。正常起動時は、syssta からリターンしません。

コンフィグレータを使用する場合、コンフィグレータが生成する main 関数(kernel_cfg.c 内)から自動的に呼ばれるようになります。

戻値

E_PAR	優先度等が範囲外*
E_ID	ID が範囲外*
E_OBJ	既に生成されている
E_SYS	管理ブロック用のメモリ不足**
E_NOMEM	スタック用やメモリプール用のメモリ不足**

intsta

機能 周期タイマ割り込み起動

形式 ER intsta(void);

解説 タスクの時間待ちを管理するための、周期タイマ割り込みを起動します。この関数は、syssta システムコールの直前で、呼び出してください。タイムアウト付きのシステムコールやタイムイベントハンドラを使用しない場合は、intsta を実行する必要はありません。

機種依存しますので、カーネルとは別の n4ixxx.c に定義されています。割り込み周期は、標準で 10msec です。サンプルとして付属の n4ixxx.c が適合しない場合は、ユーザーで作成してください。ユーザーが作成する場合、この関数名称にこだわる必要はありません。

戻値

E_OK	正常終了
E_PAR	割り込み周期等が範囲外(機種依存)

intext

機能 **周期タイマ割り込み終了**

形式 void intext();

解説 intsta で起動したタイマを停止させます。

機種依存しますので、カーネルとは別の n4ixxx.c に定義されています。サンプルとして付属の n4ixxx.c が適合しない場合は、ユーザーで作成してください。ユーザーが作成する場合、この関数名称にこだわる必要はありません。タイマ割り込みを止める必要性がなければ、作成しなくても構いません(サンプルの多くでも省略しています)。

戻値 **なし**

intini

機能 割り込み初期化

形式 ER intini(void);

解説 sysini から割り込み禁止状態で呼び出されます。ハードウェアの初期化等をおこないます。

機種依存しますので、カーネルとは別のサンプル n4ixxx.c に定義されています。この関数をユーザーが作成する場合、特に初期化するものがなければ、何もせずリターンしてください。

戻値

E_OK	正常終了
E_PAR	割り込みベクタサイズ等が範囲外(機種依存)

第７章 一覧

7.1 エラーコード一覧

E_OK	0	正常終了
E_SYS	0xf..ffb (-5)	システムエラー
E_NOSPT	0xf..ff7 (-9)	未サポート機能
E_RSFN	0xf..ff6 (-10)	予約機能コード
E_RSATR	0xf..ff5 (-11)	予約属性
E_PAR	0xf..fef (-17)	パラメータエラー
E_ID	0xf..fee (-18)	不正 ID 番号
E_CTX	0xf..fe7 (-25)	コンテキストエラー
E_ILUSE	0xf..fe4 (-28)	システムコール不正使用
E_NOMEM	0xf..fdf (-33)	メモリ不足
E_NOID	0xf..fde (-34)	ID 番号不足
E_OBJ	0xf..fd7 (-41)	オブジェクト状態エラー
E_NOEXS	0xf..fd6 (-42)	オブジェクト未生成
E_QOVR	0xf..fd5 (-43)	キューイングオーバーフロー
E_RLWAI	0xf..fcf (-49)	待ち状態の強制解除
E_TMOUT	0xf..fce (-50)	ポーリング失敗またはタイムアウト
E_DLT	0xf..fcd (-51)	待ちオブジェクトの削除
E_CLS	0xf..fcc (-52)	待ちオブジェクトの状態変化
E_WBLK	0xf..fc7 (-57)	ノンブロッキング受付け
E_BOVR	0xf..fc6 (-58)	バッファオーバフロー

7.2 システムコール一覧

タスク管理機能

	①②③
タスク生成 cre_tsk(tskid, pk_ctsk);	○○×
タスク生成(ID 自動割付け) acre_tsk(pk_ctsk);	○○×
タスク起動 act_tsk(tskid);	○○○
タスク起動 iact_tsk(tskid);	×○○
タスク起動要求のキャンセル can_act(tskid);	○○○
タスク起動(起動コード指定) sta_tsk(tskid, stacd);	○○○
自タスク終了 ext_tsk();	○××
他タスク強制終了 ter_tsk(tskid);	○○○
タスク優先度変更 chg_pri(tskid, tskpri);	○○○
タスク優先度参照 get_pri(tskid, p_tskpri);	○○○
タスク状態参照 ref_tsk(tskid, pk_rtsk);	○○○
タスク状態参照(簡易版) ref_tst(tskid, pk_rtst);	○○○

凡例

★ NORTi Oceans 独自システムコール

①タスクから発行可能

②タイムイベントハンドラから発行可能

③割り込みハンドラから発行可能

タスク付属同期

	①②③
起床待ち slp_tsk();	○××
起床待ち（タイムアウト有） tslp_tsk(tmout);	○××
タスク起床 wup_tsk(tskid);	○○○
タスク起床 iwup_tsk(tskid);	×○○
タスク起床要求のキャンセル can_wup(tskid);	○○○
自タスク起床要求キャンセル★ vcan_wup();	○○○
待ち状態の強制解除 rel_wai(tskid);	○○○
待ち状態の強制解除 irel_wai(tskid);	×○○
自タスクの遅延 dly_tsk(dlytim);	○××

凡例

★ NORTi Oceans 独自システムコール

①タスクから発行可能

②タイムイベントハンドラから発行可能

③割り込みハンドラから発行可能

同期・通信 セマフォ

	①②③
セマフォ生成 cre_sem(semid, pk_csem);	○○×
セマフォ生成（ID 自動割付け） acre_sem(pk_csem);	○○×
セマフォ資源返却 sig_sem(semid);	○○○
セマフォ資源返却 isig_sem(semid);	×○○
セマフォ資源獲得 wai_sem(semid);	○××
セマフォ資源獲得(ポーリング) pol_sem(semid);	○○○
セマフォ資源獲得(タイムアウト有) twai_sem(semid, tmout);	○××
セマフォ状態参照 ref_sem(semid, pk_rsem);	○○○

凡例

★ NORTi Oceans 独自システムコール

①タスクから発行可能

②タイムイベントハンドラから発行可能

③割り込みハンドラから発行可能

同期・通信 イベントフラグ

	①②③
イベントフラグ生成 cre_flg(flgid, pk_cflg);	○○×
イベントフラグ生成（ID 自動割付け） acre_flg(pk_cflg);	○○×
イベントフラグのセット set_flg(flgid, setptn);	○○○
イベントフラグのセット iset_flg(flgid, setptn);	×○○
イベントフラグのクリア clr_flg(flgid, clrptn);	○○○
イベントフラグ待ち wai_flg(flgid, waiptn, wfmode, p_flgptn);	○××
イベントフラグ待ち（ポーリング） pol_flg(flgid, waiptn, wfmode, p_flgptn);	○○○
イベントフラグ待ち（タイムアウト有） twai_flg(flgid, waiptn, wfmode,p_flgptn, tmout);	○××
イベントフラグ状態参照 ref_flg(flgid, pk_rflg);	○○○

凡例

★ NORTi Oceans 独自システムコール

①タスクから発行可能

②タイムイベントハンドラから発行可能

③割り込みハンドラから発行可能

同期・通信 データキュー

	①②③
データキュー生成 cre_dtq(dtqid, pk_cdtq);	○○×
データキュー生成(ID 自動割付け) acre_dtq(pk_cdtq);	○○×
データキューへ送信 snd_dtq(dtqid, data);	○××
データキューへ送信(ポーリング) psnd_dtq(dtqid, data);	○○○
データキューへ送信(ポーリング) ipsnd_dtq(dtqid, data);	×○○
データキューへ送信(タイムアウト有) tsnd_dtq(dtqid, data, tmout);	○××
データキューへ強制送信 fsnd_dtq(dtqid, data);	○○○
データキューへ強制送信 ifsnd_dtq(dtqid, data);	×○○
データキューからの受信 rcv_dtq(dtqid, p_data);	○××
データキューからの受信(ポーリング) prcv_dtq(dtqid, p_data);	○○○
データキューからの受信(タイムアウト有) trcv_dtq(dtqid, p_data, tmout);	○××
データキューの状態参照 ref_dtq(dtqid, pk_rdtq);	○○○

凡例

★ NORTi Oceans 独自システムコール

①タスクから発行可能

②タイムイベントハンドラから発行可能

③割り込みハンドラから発行可能

同期・通信機能(メールボックス)

	①②③
メールボックス生成 cre_mbx(mbxid, pk_cmbx);	○○×
メールボックス生成(ID 自動割付け) acre_mbx(pk_cmbx);	○○×
メールボックスへ送信 snd_mbx(mbxid, pk_msg);	○○○
メールボックスから受信 rcv_mbx(mbxid, ppk_msg);	○××
メールボックスから受信(ポーリング) prcv_mbx(mbxid, ppk_msg);	○○○
メールボックスから受信(タイムアウト有) trcv_mbx(mbxid, ppk_msg, tmout);	○××
メールボックスの状態参照 ref_mbx(mbxid, pk_rmbx);	○○○

凡例

★ NORTi Oceans 独自システムコール

①タスクから発行可能

②タイムイベントハンドラから発行可能

③割り込みハンドラから発行可能

メモリプール管理 固定長

	①②③
固定長メモリプール生成 cre_mpf(mpfid, pk_cmpf);	○○×
固定長メモリプール生成(ID 自動割付け) acre_mpf(pk_cmpf);	○○×
固定長メモリブロック獲得 get_mpf(mpfid, p_blk);	○××
固定長メモリブロック獲得(ポーリング) pget_mpf(mpfid, p_blk);	○○○
固定長メモリブロック獲得(タイムアウト有) tget_mpf(mpfid, p_blk, tmout);	○××
固定長メモリブロック返却 rel_mpf(mpfid, blk);	○○○
固定長メモリプールの状態参照 ref_mpf(mpfid, pk_rmpf);	○○○

凡例

★ NORTi Oceans 独自システムコール

①タスクから発行可能

②タイムイベントハンドラから発行可能

③割り込みハンドラから発行可能

時間管理 システム時刻管理

	①②③
システム時刻の設定 set_tim(p_tim);	○○○
システム時刻の参照 get_tim(p_tim);	○○○
タイムティックの供給 isig_tim();	××○
タイムティックの供給 sig_tim();	××○

凡例

★ NORTi Oceans 独自システムコール

①タスクから発行可能

②タイムイベントハンドラから発行可能

③割り込みハンドラから発行可能

時間管理 周期ハンドラ

	①②③
周期ハンドラの生成 cre_cyc(cycid, pk_ccyc);	○○×
周期ハンドラの生成(ID 自動割付け) acre_cyc(pk_ccyc);	○○×
周期ハンドラの開始 sta_cyc(cycid);	○○○
周期ハンドラの停止 stp_cyc(cycid);	○○○
周期ハンドラの状態参照 ref_cyc(cycid, pk_rcyc);	○○○

凡例

★ NORTi Oceans 独自システムコール

①タスクから発行可能

②タイムイベントハンドラから発行可能

③割り込みハンドラから発行可能

システム状態管理

	①②③
タスクの実行順位の回転 rot_rdq(tskpri);	○○○
タスクの実行順位の回転 irot_rdq(tskpri);	×○○
実行状態のタスク ID 参照 get_tid(p_tskid);	○○○
実行状態のタスク ID 参照 iget_tid(p_tskid);	×○○
自タスク ID 参照★ vget_tid();	○○○
CPU ロック状態への移行 loc_cpu();	○○×
CPU ロック状態への移行 iloc_cpu();	×○×
CPU ロック状態の解除 unl_cpu();	○○×
CPU ロック状態の解除 iunl_cpu();	×○×
ディスパッチ禁止 dis_dsp();	○××
ディスパッチ許可 ena_dsp();	○××
システムの状態参照 ref_sys(pk_rsys);	○××
コンテキストの参照 sns_ctx();	○○○
CPU ロック状態の参照 sns_loc();	○○○
ディスパッチ禁止状態の参照 sns_dsp();	○○○
ディスパッチ保留状態の参照 sns_dpn();	○○○

凡例

★ NORTi Oceans 独自システムコール

①タスクから発行可能

②タイムイベントハンドラから発行可能

③割り込みハンドラから発行可能

割り込み管理

	①②③
割り込みハンドラの定義 def_inh(inhno, pk_dinh);	○○○
割り込みの禁止 dis_int(intno);	○○×
割り込みの許可 ena_int(intno);	○○×
割り込みマスクの変更 chg_ims(imask);	○○○
割り込みマスクの参照 get_ims(p_imask);	○○○
割り込みハンドラ開始★ ent_int();	××○
割り込みハンドラ終了★ ret_int();	××○
ステータスレジスタセット★ vset_psw(psw);	○○○
ステータスレジスタの割り込みマスクセット★ vdis_psw();	○○○

凡例

★ NORTi Oceans 独自システムコール

①タスクから発行可能

②タイムイベントハンドラから発行可能

③割り込みハンドラから発行可能

システム構成管理

	①②③
コンフィグレーション情報参照 ref_cfg(pk_rcfg);	○○○
バージョン情報参照 ref_ver(pk_rver);	○○○

凡例

★ NORTi Oceans 独自システムコール

①タスクから発行可能

②タイムイベントハンドラから発行可能

③割り込みハンドラから発行可能

7.3 パケット構造体一覧

タスク生成情報パケット

```
typedef struct t_ctsk {
    ATR tskatr;·····  タスク属性
    VP_INT exinf;····  タスク拡張情報
    FP task;·····  タスクとする関数へのポインタ
    PRI itskpri;·····  起動時タスク優先度
    SIZE stksz;·····  スタックサイズ(バイト数)
    VP stk; ·····  スタック領域先頭番地
    B *name;·····  タスク名へのポインタ
} T_CTSK;
```

タスク状態パケット

```
typedef struct t_rtsk {
    STAT tskstat;·····  タスク状態
    PRI tskpri;·····  タスク現在優先度
    STAT tskwait;·····  待ち要因
    ID wid; ·····  待ち対象オブジェクト ID
    TMO lefttmo;·····  タイムアウトまでの時間
    UINT actcnt;·····  起動要求カウント
    UINT wupcnt;·····  起床要求カウント
    VP exinf;·····  拡張情報
    ATR tskatr;·····  タスク属性
    FP task;·····  タスクとする関数へのポインタ
    PRI itskpri;·····  起動時タスク優先度
    SIZE stksz;·····  スタックサイズ(バイト数)
} T_RTSK;
```

タスク状態簡易パケット

```
typedef struct t_rtst {
    STAT tskstat;·····  タスク状態
    STAT tskwait;·····  待ち要因
} T_RTST;
```

セマフォ生成情報パケット

```
typedef struct t_csem {
    ATR sematr;·····  セマフォ属性
    UINT isemcnt;·····  セマフォ初期値
    UINT maxsem;·····  セマフォ最大値
    B *name;·····  セマフォ名へのポインタ
} T_CSEM;
```

セマフォ状態パケット

```
typedef struct t_rsem {
    ID wtskid;·····  待ちタスクの ID
    UINT semcnt;·····  セマフォ値
} T_RSEM;
```

イベントフラグ生成情報パケット

```
typedef struct t_cflg {
    ATR flgatr;·····  イベントフラグ属性
    FLGPTN iflgptn;·····  イベントフラグ初期値
    B *name;·····  イベントフラグ名へのポインタ
} T_CFLG;
```

イベントフラグ状態パケット

```
typedef struct t_rflg {
    ID wtskid;·····  待ちタスクの ID
    FLGPTN flgptn;·····  イベントフラグ値
} T_RFLG;
```

データキュー生成情報パケット

```
typedef struct t_cdtq {
    ATR dtqatr;·····  データキュー属性
    UINT dtqcnt;·····  データキューサイズ(データ数)
    VP dtq; ·····  リングバッファアドレス
    B *name;·····  データキュー名へのポインタ
} T_CDTQ;
```

データキュー状態パケット

```
typedef struct t_rdtq {
    ID stskid;·····  送信待ちタスクの ID
    ID rtskid;·····  受信待ちタスクの ID
    UINT sdtqcnt;·····  データキューに入っているデータ数
} T_RDTQ;
```

メールボックス生成情報パケット

```
typedef struct t_cmbx {
    ATR mbxatr;·····  メールボックス属性
    PRI maxmpri;·····  メッセージ優先度の最大値
    VP mprihd;·····  メッセージ待ち行列ヘッダへのポインタ
    B *name;·····  メールボックス名へのポインタ
} T_CMBX;
```

メールボックス状態パケット

```
typedef struct t_rmbx {
    ID wtskid;·····  受信待ちタスク ID
    T_MSG *pk_msg;·····  次に送信されるメッセージへのポインタ
} T_RMBX;
```

割り込みハンドラ定義情報パケット

```
typedef struct t_dinh {
    ATR inhatr;·····  割り込みハンドラ属性
    FP inthdr;·····  割り込みハンドラ関数のアドレス
    UINT imask;·····  割り込みマスク
} T_DINH;
```

固定長メモリプール生成情報パケット

```
typedef struct t_cmpf {
      ATR mpfatr;·····  固定長メモリプール属性
      UINT blkcnt;·····  総メモリブロック数
      UINT blfsz;·····  メモリブロックのサイズ(バイト)
      VP mpf; ·····  メモリプールアドレス
      B *name;·····  固定長メモリプール名へのポインタ
} T_CMPF;
```

固定長メモリプール状態パケット

```
typedef struct t_rmpf {
      ID wtskid;·····  獲得待ちタスクの ID
      UINT frbcnt;·····  空きブロック数
} T_RMPF;
```

周期ハンドラ生成情報パケット

```
typedef struct t_ccyc {
      ATR cycatr;·····  周期ハンドラ属性
      VP_INT exinf;·····  拡張情報
      FP cychdr;·····  周期ハンドラ関数のアドレス
      RELTIM cyctim;·····  起動周期
      RELTIM cycphs;·····  起動位相
} T_CCYC;
```

周期ハンドラ状態パケット

```
typedef struct t_rcyc {
      STAT cycstat;·····  周期ハンドラ動作状態
      RELTIM lefttim;·····  起動すべき時刻までの時間
} T_RCYC;
```

バージョン情報パケット

```
typedef struct t_rver {
      UH maker;·····  メーカーコード
      UH prid;·····  カーネル識別番号
      UH spver;·····  ITRON 仕様書バージョン
      UH prver;·····  カーネルバージョン番号
      UH prno[4];·····  管理情報
} T_RVER;
```

システム状態パケット

```
typedef struct t_rsys {
      INT sysstat;·····  システム状態
} T_RSYS;
```

コンフィグレーション情報パケット

```
typedef struct t_rcfg {
      ID tskid_max;·····タスク ID 上限
      ID semid_max;·····セマフォ ID 上限
      ID flgid_max;·····イベントフラグ ID 上限
      ID mbxid_max;·····メールボックス ID 上限
      ID mpfid_max;·····固定長メモリプール ID 上限
      ID cycno_max;·····周期ハンドラ ID 上限
      PRI tpri_max;·····タスク優先度上限
      int tmrqsz;·····タスクのタイマキューサイズ(バイト数)
      int cycqsz;·····周期ハンドラのタイマキューサイズ(バイト数)
      int istksz;·····割り込みハンドラのスタックサイズ(バイト数)
      int tstksz;·····タイムイベントハンドラのスタックサイズ(バイト数)
      SIZE sysmsz;·····システムメモリのサイズ(バイト数)
      SIZE mplmsz;·····メモリプール用メモリのサイズ(バイト数)
      SIZE stkmsz;·····スタック用メモリのサイズ(バイト数)
      ID dtqid_max;·····データキューID 上限
} T_RCFG;
```

7.4 定数一覧

タスク/ハンドラ属性		
TA_HLNG	0x0000	高級言語で記述されている
TA_ACT	0x0002	タスク実行可能状態でタスク生成
タスク待ち行列属性		
TA_TFIFO	0x0000	先着順 *1
TA_TPRI	0x0001	タスク優先度順 *1
TA_TPRIR	0x0004	受信タスク優先度順(メッセージバッファ) *1
タイムアウト		
TMO_POL	0	ポーリング(待ちなし)
TMO_FEVR	-1	無限待ち(タイムアウトなし)
タスク ID		
TSK_SELF	0	自タスク指定
TSK_NONE	0	タスクなし
タスク優先度		
TPRI_INI	0	起動時優先度
TPRI_SELF	0	自タスクのベース優先度
TMIN_TPRI	1	優先度の最小値
TMAX_TPRI		優先度の最大値（値はコンフィグレーションに依存）
タスク状態		
TTS_RUN	0x0001	実行状態
TTS_RDY	0x0002	実行可能状態
TTS_WAI	0x0004	WAITING 状態
TTS_SUS	0x0008	SUSPENDED 状態 *2
TTS_WAS	0x000c	WAITING-SUSPENDED 状態 *2
TTS_DMT	0x0010	DORMANT 状態
タスク待ち要因		
TTW_SLP	0x0001	起床待ち
TTW_DLY	0x0002	時間待ち
TTW_SEM	0x0004	セマフォ獲得待ち
TTW_FLG	0x0008	イベントフラグ待ち
TTW_SDTQ	0x0010	データキュー送信待ち
TTW_RDTQ	0x0020	データキュー受信待ち
TTW_MBX	0x0040	メールボックスでメッセージ待ち
TTW_MTX	0x0080	ミューテックス獲得待ち *2
TTW_SMBF	0x0100	メッセージバッファでメッセージ送信待ち *2

TTW_MBF	0x0200	メッセージバッファでメッセージ受信待ち *2
TTW_CAL	0x0400	ランデブ呼出待ち *2
TTW_ACP	0x0800	ランデブ受付け待ち *2
TTW_RDV	0x1000	ランデブ終了待ち *2
TTW_MPF	0x2000	可変長メモリブロック獲得待ち
TTW_MPL	0x4000	固定長メモリブロック獲得待ち *2

イベントフラグ属性

TA_WSGL	0x0000	複数タスク待ち禁止 *1
TA_CLR	0x0004	クリア指定
TA_WMUL	0x0002	複数タスク待ち許可 *3

イベントフラグ待ちモード

TWF_ANDW	0x0000	AND 待ち
TWF_ORW	0x0001	OR 待ち
TWF_CLR	0x0004	クリア指定

メッセージ待ち行列

TA_MFIFO	0x0000	先着順 *1
TA_MPRI	0x0002	メッセージ優先度順 *1

メッセージ優先度

TMIN_MPRI	1	メッセージ最高優先度 *1

ミューテックス属性

TA_INHERIT	0x0002	優先度継承プロトコル *2
TA_CEILING	0x0003	優先度上限プロトコル *2

ランデブ用ポート属性

TA_NULL	0	属性特になし *2

周期ハンドラ属性

TA_STA	0x0002	周期ハンドラ起動
TA_PHS	0x0004	位相保存

周期ハンドラ状態

TCYC_STP	0x0000	停止状態
TCYC_STA	0x0001	動作状態

アラームハンドラ状態

TALM_STP	0x0000	停止状態 *2
TALM_STA	0x0001	動作状態 *2

オーバーランハンドラ状態

TOVR_STP	0x0000	停止状態 *2
TOVR_STA	0x0001	動作状態 *2

システム状態

TSS_TSK	0	タスクコンテキスト部
TSS_DDSP	1	タスクコンテキスト部(ディスパッチ禁止状態)
TSS_LOC	3	タスクコンテキスト部(CPU ロック状態)
TSS_INDP	4	非タスクコンテキスト部

キューイング数等の最大値

TMAX_WUPCNT	255	wup_tsk による起床要求数の最大値
TMAX_SUSCNT	255	sus_tsk による強制待ち要求数の最大値 *2
TMAX_ACTCNT	255	act_tsk による起動要求数の最大値
TMAX_MAXSEM	255	セマフォカウントの最大値

その他の定数

TRUE	1	真
FALSE	0	偽

(*1) これらの値は NORTi カーネル Ver.4 用に作成されたアプリケーションとの互換性のために用意されています。使用しても無視され、エラーも含めてシステムコールの動作にはなにも影響しません。

(*2) これらの値は NORTi カーネル Ver.4 用に作成されたアプリケーションとの互換性のために用意されています。システムコールがこれらの値を返すことはありません。

(*3) これらの値は NORTi カーネル Ver.4 用に作成されたアプリケーションとの互換性のために用意されています。使用するとパラメータエラーになります。

索引

L

M

P

R

S

T

す

せ

そ

た

ち

て

は

ほ

NORTi Oceans ユーザーズガイド

カーネル編

2012 年　3 月　初版

株式会社ミスポ http://www.mispo.co.jp/
〒213-0002　川崎市高津区二子 5-1-1
TEL 044-829-3381　FAX 044-829-3382
一般的なお問い合わせ sales@mispo.co.jp
技術サポートご依頼　norti@mispo.co.jp