μITRON4.0 仕様準拠リアルタイム OS

カーネル編

ユ ー ザ ー ズ ガ イ ド

MiSPO
株式会社ミスポ

（余白）

μITRON4.0 仕様準拠リアルタイム ＯＳ
NORTi Version 4 ユーザーズガイド・カーネル編

は じめ に

株式会社ミスポが自信を持ってお届けする「NORTi（ノートアイ） Version 4」は、社団法人トロン協会が策定し公開している「μITRON4.0 仕様」に準拠したリアルタイム OS です。本製品には、CPU 例外ハンドラ定義を除くほぼ全ての μITRON4.0 仕様のシステムコールが実装されています。さらに、μITRON3.0 仕様準拠「NORTi3」と互換のシステムコールも含めることにより、従来のソフトウェア資源をそのまま活用できる配慮をしてあります。

NORTi は、手軽に組込むことのできるコンパクトな組込み専用の OS です。コンパイラの標準ライブラリと同様に、ユーザーの作成したプログラムと、NORTi のライブラリとをリンクすることで、OS の機能が利用できるようになります。

NORTi は、省メモリを特長とし組込み機器に適した「ITRON TCP/IP API 仕様」準拠の TCP/IP プロトコルスタックを搭載しています。不可欠な技術となったネットワーク接続の組込み機器開発に、NORTi を使用することで、いち早く対応できます。

豊富な機能と優れた性能と共に、全ソースコード標準添付、組込みロイヤリティ無料という特長を持つ NORTi を、皆様方のシステム開発に、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 Version 4 シリーズのリアルタイム・マルチタスク機能の共通マニュアルです。前半部で概要を説明し、後半部で各システムコールの解説をおこなっています。プロセッサ固有の記事については、ディスク内のドキュメントを参照してください。TCP/IP プロトコルスタック機能については、「ネットワーク編」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

第２版で訂正された項目

ページ	内容
20	データキュー管理ブロックサイズの① 32 → 28
132	pol_dtq を prcv_dtq に修正（PDF のみ、製本マニュアルの訂正は第 3 版にて ）

第３版で訂正された項目

ページ	内容
5	タスクの状態の NORTi3 との差異追加
8	静的 API と動的 API 追加
9	引数の名称の説明に属性、個数、ビットパターン追加
10	RDVPTN，MODE 追加、ER_BOOL 削除
14 ～ 15	１．７ データタイプ（8 ビット CPU の場合）を削除
19	DTQID_MAX，MTXID_MAX，ISRID_MAX，SVCFN_MAX 定義追加
20	管理ブロックのサイズ表修正
28	割込みハンドラと割込みサービスルーチンの違いを追加
56	RISC プロセッサでの割込み修正
56	カーネルより高優先度の割込みルーチン追加
70	can_act の戻値の型 ER → ER_UINT
76	chg_rqd → chg_pri
78，80	8 ビット CPU での tskwai の型の説明削除
89	can_wup の戻値の型 ER → ER_UINT
92	8 ビット CPU の TMO 型の説明削除
173	acre_por の戻値の型 ER → ER_ID
187	" 割込みエントリを共有する RISC 系プロセッサを除く " を削除
196	acre_isr の戻値の型 ER → ER_ID
199	8 ビット CPU のでのメモリプール使用時の OS が消費するメモリサイズ削除
217,218	8 ビット CPU の場合の SYSTIM 削除
226	acre_alm の戻値の型 ER → ER_ID
229	ref_sem の引数が不正 ID 時の戻値 E_PAR → E_ID
235	拡張サービスコールルーチンの戻値の型 ER → ER_UINT
237	cal_svc の戻値の型 ER → ER_UINT
249,282	T_RCFG 中のメモリサイズの型 int → SIZE
5 章	can_act，ref_tsk，ref_tst，def_tex，ref_tex，ras_tex，stp_ovr，ref_ovr に自タスクの指定 (tskid = TSK_SELF) の説明追加

5 章	can_wup, acre_sem, acre_flg, acre_dtq, acre_mbx, acre_mtx, acre_mbf, trcv_mbf, acre_por, cal_por, acp_por, acre_isr, acre_mpl, acre_mbf, acre_cyc, acre_alm の例のエラーコードを代入する変数の型を修正
252	標準の割込み周期 1msec → 10msec
252	intsta のコンフィグレータを使用する場合の説明修正
276	8 ビット CPU での tskwai の型の説明削除
283	TA_ACT 追加
付録	ライブラリ説明の削除（製本マニュアルのみ、PDF では削除済み）
全体	名称の修正 周期起動ハンドラ → 周期ハンドラ ハンドラ番号 , ハンドラ指定番号 → ハンドラ ID タイマハンドラ → タイムイベントハンドラ タスク独立部 → 非タスクコンテキスト

第４版 (本版) で訂正され た項目

ページ	内容
4	タスク切り替えの起きるタイミングの説明を追加
21	スタック用メモリのサイズの説明からタイムイベントハンドラのスタックを除外
25	サンプルのコンパイル例に、vecxxx.asm や init.c を含める
64	T_CTSK の name 初期値の説明に、NULL あるいは省略可を追記
65	cre_tsk の例の T_CTSK stk 初期値 "" → NULL
76	chg_pri 非タスクコンテキストで TSK_SELF 指定は E_ID エラー
100,110 122,135 149,158 171,199 208	各生成情報パケットの説明に、name の説明を追加
24,35 110,111 123,124 136,137 149,150 158,159 172,173 199,200 209,210	各生成情報パケットの定義例で、name の初期値を省略
101	T_CSEM isemcnt 初期値 0 → 1
139-141	標準 T_MSG 型から msgpri を削除
154	pol_mtx → ploc_mtx
217	iset_tim 削除

260	システムコール一覧表に ifsnd_dtq 追加
266	pget_mpl は、割込みハンドラから発行不可
275	静的 API 一覧を削除

目次

第１章 基本事項

第４章 機能概説

第５章 システムコール解説

第１章 基本事項

１．１ 特長

高速な応答性

NORTi 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元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います。システムコール内部でスキャンすることなく１発で操作対象を選択でき、また、割込み禁止時間も旧来より半減されていて、CPU の能力を最大限に引き出すことができます。さらに、OS より高優先の割込みルーチンを導入することができ、この場合の割込み禁止時間は、限りなくゼロです。

コンパクトなサイズ

TCB 等のカーネル内部の管理ブロック変数は、徹底的にサイズの最適化が行われています。貴重な RAM 領域を１バイトたりとも無駄にしません。

C で設計されたカーネル

NORTi の大部分は、理解しやすい C 言語で記述されています。C で設計された OS が、全てアセンブラで記述された OS より性能が劣ると考えるなら、それは誤解です。レジスタ割り付けをコンパイラに任せた方が、内部でのレジスタ待避 / 復元が最小で済み、返って高速となります。さらに、実績のあるソースコードを複数種の CPU で共有できますので、新規 CPU 対応版のリリース直後から信頼性を確保できます。

μITRON4.0 と μITRON3.0 の両仕様に準拠

トロン協会の μITRON4.0 仕様は、3.0 仕様との互換性を犠牲にしてしまいました。そこで、NORTi では、μITRON4.0 仕様だけでなく、μITRON3.0 仕様のインターフェースも実装することにより、前バージョン用に開発されたソフトウェアを、変更することなく共存して利用できるよう工夫してあります。

フルセットの μITRON

μITRON 仕様を謳いながら、面倒な部分の実装を省いたり、日本向けとして他アーキテクチャへ無理に μITRON API を被せたような OS がある中、NORTi では μITRON4.0/3.0 仕様でフルセットと位置づけられる機能を丁寧に実装し、豊富な同期通信手段を提供しています (CPU 例外ハンドラの定義を除く)。

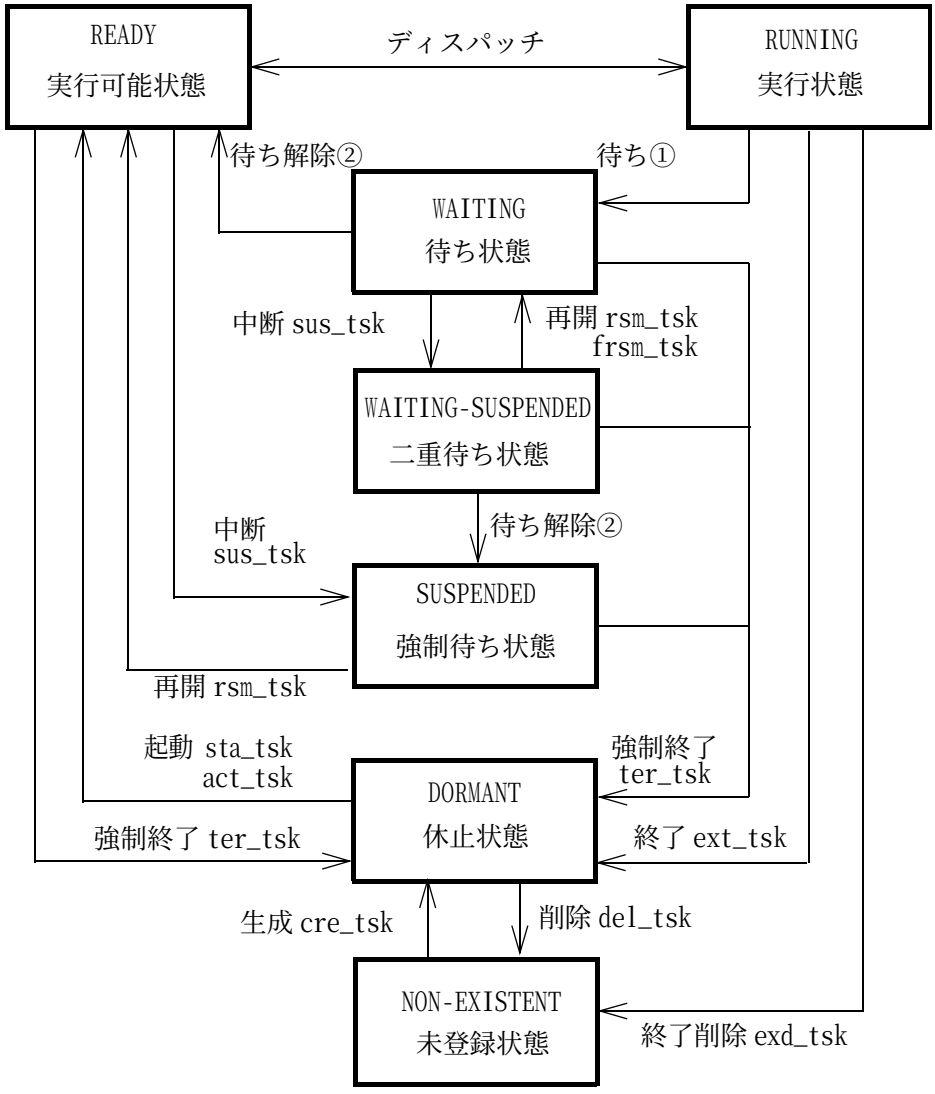
複数種のプロセッサ／コンパイラ／デバッガに対応

多くの 16 ～ 32 ビットのプロセッサに対応済みですので、ターゲットが変わっても、同じ作法で NORTi が使えます。また、開発環境の選択の幅が広がるよう、複数の開発ツールメーカーと協力して、積極的な対応を行っています。

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ます。タスクは、NON-EXISTENT, DORMANT, READY, RUNNING, WAITING, SUSPENDED, WAITING-SUSPENDED の７つの状態のいずれかをとります。

タスクの状態遷移を図に示します。



① slp_tsk, tslp_tsk, wai_sem, twai_sem, wai_flg, twai_flg, rcv_mbx, trcv_mbx, rcv_mbf, trcv_mbf, snd_mbf, tsnd_mbf, cal_por, tcal_por, acp_por, tacp_por, get_mpl, tget_mpl, get_mpf, tget_mpf, dly_tsk, snd_dtq, tsnd_dtq, rcv_dtq, trcv_dtq, loc_mtx, tloc_mtx

② rel_wai, wup_tsk, sig_sem, set_flg, del_sem, snd_mbx, snd_mbf, tsnd_mbf, psnd_mbf, rcv_mbf, prcv_mbf, trcv_mbf, del_mbf, cal_por, tcal_por, acp_por, tacp_por, del_por, rpl_rdv, rel_mpl, del_mpl, rel_mpf, del_mpf, snd_dtq, psnd_dtq, tsnd_dtq, del_dtq, unl_mtx, del_mtx, ter_tsk

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。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 状態で過ごすはずです。そうでないと、タスクの待ちの間を利用して、別のタスクを実行できません。

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）
ミューテックス獲得待ち（loc_mtx, tloc_mtx）
メールボックスでのメッセージ受信待ち（rcv_mbx, trcv_mbx）
メッセージバッファでのメッセージ送信待ち（snd_mbf, tsnd_mbf）
メッセージバッファでのメッセージ受信待ち（rcv_mbf, trcv_mbf）
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）
ランデブ呼出待ち（cal_por, tcal_por）
ランデブ受付待ち（acp_por, tacp_por）
ランデブ終了待ち（cal_por, tcal_por）
可変長メモリブロック獲得待ち（get_mpl, tget_mpl）
固定長メモリブロック獲得待ち（get_mpf, tget_mpf）

強制待ち状態（SUSPENDED）

他タスクによって、実行を止められた状態です。この SUSPENDED 状態を利用する機会は多くありません。デバッグのために、タスクの実行を一時的に停止させる用途が考えられます。

二重待ち状態（WAITING-SUSPENDED）

管理の都合上、分けてありますが、SUSPENDED 状態と同じと考えて差し支えありません。他タスクから実行を止めにいったら READY でなく WAITING 状態だったという違いだけです。待つことまで止められている訳ではありません。待ち条件が満たされれば、WAITING のみ外れて SUSPENDED 状態へ移ります。

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、あるいはタスクが削除された状態です。

タスク切り替えの起きるタイミング

NORTi 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（割込みハンドラ / 割込みサービスルーチン / 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。

優先度が同じ場合も低い場合と同様ですが、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。

NORTi3 との差異

RUN → RUNNING, WAIT → WAITING と、呼び名が変わりました。

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、カーネルやソフトウェア部品の内部に閉じていてユーザが直接指定できないものをオブジェクト番号と呼びます。

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、メッセージバッファ、ランデブ用ポート、可変長／固定長メモリプール、データキュー、ミューテックス、周期ハンドラ、アラームハンドラ、割込みサービスルーチンがあります。オブジェクト番号で識別されるオブジェクトには、割込みハンドラ、ランデブ、静的に生成された割込みサービスルーチンがあります。

コンテキスト

直訳は「文脈」となりますが、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。コンテキストとは、タスクが切り替えられる時に、保存／復元される物の総称ですが、具体的には CPU のレジスタと読み代えても構いません。

DSP や浮動小数点演算ユニットをマルチタスクで使用する場合、そのレジスタ類もコンテキスト切り替えしなければなりません。NORTi でそれをサポートしていない場合、浮動小数点演算等は、排他制御する必要があります。

非タスクコンテキスト

割込みハンドラと、タイムイベント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。タイムイベントハンドラには、周期起動ハンドラとアラームハンドラ、オーバーランハンドラの３種類があります (μITRON3.0 仕様では、非タスクコンテキストはタスク独立部、タイムイベントハンドラはタイマハンドラと呼称)。

非タスクコンテキストの各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 として区別してよいことになっています。NORTi 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ますので区別は無く、i 付きのシステムコールは、i 無しのシステムコールに同じと、kernel.h で定義してあります。

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。

セマフォ、イベントフラグ、メールボックス、メッセージバッファ、ランデブ用ポート、データキュー、ミューテックスという７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、イベントフラグ成立待ち、メールボックスのメッセージ受信待ち、メッセージバッファのメッセージ送信／受信待ち、ポートでのランデブ呼出／受付待ち、可変長／固定長メモリプールのメモリブロック獲得待ち、データキュー送信／受信待ち、ミューテックス獲得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）が基本ですが、セマフォ、メールボックス、メッセージバッファ受信側、可変長／固定長メモリプール、ミューテックスでは、タスクの優先度順あるいはメッセージ優先度順で並ぶことも可能です。

キューイング

相手のタスクが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／メッセージバッファ／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。メッセージバッファ／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。

イベントフラグおよびタスク例外では、キューイングではなく、OR 演算によるイベントおよび要因の保留が行われます。この場合は、事象の有無のみ記録され回数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、引き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、複数のタスクが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あるいはミューテックスが使われます。

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。ミューテックスには必要に応じて優先度を上げるオプションがあります。しかし、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。

静的 API と動的 API

μITRON 仕様では､大文字で記述される生成系のシステムコールを静的 API と呼びますが、これを OS で直接サポートする訳ではありません。静的 API の仕組みは、コンパイル時に TCB 等の管理ブロックが確保され、それがシステム起動時に初期化されていることを前提としています。つまり、コンパイル前に静的な API に合わせたコード生成が必要となり、そのために μITRON4.0 仕様から導入されたのがコンフィグレータです。

NORTi の基本は動的なオブジェクトの生成ですので、NORTi のコンフィグレータは、コンフィグレーションファイルに記述された静的 API を、初期化時に実行される通常の動的 API のコードに置き換えることで、静的 API 対応を実現しています。

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 でも使用していません。

１．５ データタイプ（３２ビット ＣＰＵ の場合）

ITRON では、このように再定義した型でシステムコールを規定しています。INT, UINT は 32 ビットです。

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 int 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 int RDVNO;              ランデブ番号
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 RDVPTN;    ランデブパターン
typedef unsigned int INHNO;     割込みハンドラ番号
typedef unsigned int INTNO;     割込み番号
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
```

※ NORTi カーネル 4.05.00 以前では、MODE が UINT と誤実装されています。
※ ER_BOOL は ITRON 仕様書では定義されていますが NORTi では使用していません。

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
typedef long OVRTIM;            オーバーラン時間
```

NORTi3 との差異

構造体だった CYCTIME, ALMTIME は、整数型の RELTIM に統合されました。

SYSTIME → SYSTIM, RNO → RDVNO, HNO → INHNO に改名しました。

BOOL_ID は廃止されました。

VP_INT, ER_ID, ER_UINT, SIZE, MODE, STAT, FLGPTN, RDVPTN, TEXPTN, OVRTIM 等が新設されました。

特に、SIZE や MODE という型は、ユーザープログラムではマクロとして使いがちですので、使わないように注意してください。

１．６ データタイプ（１６ビット ＣＰＵ の場合）

INT, UINT は 16 ビットです。int と short が同じなので、H と UH を short でなく int としています。

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 int H;                  符号付き 16 ビット整数
typedef unsigned int 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 int 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 ID;                 オブジェクトの ID 番号
typedef int RDVNO;              ランデブ番号
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 RDVPTN;    ランデブパターン
typedef unsigned int INHNO;     割込みハンドラ番号
typedef unsigned int INTNO;     割込み番号
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
```

※ NORTi カーネル 4.05.00 以前では、MODE が UINT と誤実装されています。
※ ER_BOOL は ITRON 仕様書では定義されていますが NORTi では使用していません。

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
typedef long OVRTIM;            オーバーラン時間
```

NORTi3 との差異

32 ビット CPU の場合と同じ。

（余白）

（余白）

第２章 導入

２．１ インストール

ｲﾝｽﾄｰﾙされた NORTiの 標準的なﾌｫﾙﾀﾞ構成は、次の様になっています。**XXX**は ﾌﾟﾛｾｯｻｼﾘｰｽﾞ名 例：**SH，H8S，H83，**etc.）、**BBB** は 評価ボード名（例：**MS7709A，**etc.)、**YYY**は 対応ｺﾝﾊﾟｲﾗ略称 例：**SHC，GHS，GCC，**etc.）です。

/NORTi/INC ····· インクルードファイル
/NORTi/SRC ····· カーネルのソースファイル
/NORTi/SMP/XXX/BBB ··· サンプル
/NORTi/LIB/XXX/YYY ··· ライブラリ
/NORTi/DOC ····· ドキュメント

ここで説明するファイル名の **xxx** の部分は、プロセッサ／デバイスに依存します。拡張子は代表例で、実際にはコンパイラに依存します。ディスク内容についての最新の情報は、ディスク内の補足説明書または README を参照してください。同じファイル名であっても、異なるバージョンや異なるプロセッサ、あるいは NORTi3 Standard/Extended/Network のファイルと混在させないでください。

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
nosys4.h ····· システム内部定義ヘッダ
nocfg4.h ····· コンフギュレーションヘッダ
n4rxxx.h ····· CPU 差異定義ヘッダ
no4hook.h ····· フックルーチン定義ヘッダ
norti3.h ····· NORTi3 互換用カーネル標準ヘッダ
nosys3.h ····· NORTi3 互換用システム内部定義ヘッダ
nocfg3.h ····· NORTi3 互換用コンフギュレーションヘッダ
n3rxxx.h ····· NORTi3 互換用 CPU 差異定義ヘッダ
no3hook.h ····· NORTi3 互換用フックルーチン定義ヘッダ
nosio.h ····· シリアル入出力関数ヘッダ
non????.h ····· ネットワーク用のヘッダ（ネットワーク編参照）

kernel.h は、NORTi 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 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。

nocfg4.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。コンフィグレータを使用しない場合、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。コンフィグレータを使

用する場合は、コンフィグレータが生成する kernel_cfg.c に #include されるので、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。

nosys4.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。nocfg4.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。n4rxxx.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。nosys4.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、カーネルのライブラリモジュールファイルと、それを生成するためのメイクファイルが収められています。

n4exxx.lib ・・・・・・・・・・・・・・・・ カーネルライブラリ
n4exxx.mak ・・・・・・・・・・・・・・・・ 同上を生成するメイクファイル
n4fxxx.lib ・・・・・・・・・・・・・・・・ パラメータチェック無しカーネルライブラリ
n4fxxx.mak ・・・・・・・・・・・・・・・・ 同上を生成するメイクファイル
n4nxxx.???, n4dxxx.??? ・・・・ ネットワーク用のライブラリ（ネットワーク編参照）

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。
コンパイラによっては、ライブラリアンのコマンドファイルも納められています。

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 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。

ソースファイル

SRC フォルダには、カーネルのソースファイルが収められています。

n4cxxx.asm ・・・・・・・・・・・・ CPU インターフェースモジュール
noknl4.c ・・・・・・・・・・・・・・ NORTi カーネルソース
non????.c ・・・・・・・・・・・・・ ネットワーク用のソース（ネットワーク編参照）

コンパイラによっては、アセンブラソースの拡張子が asm 以外の場合があります。

サンプル

周期タイマ割込みハンドラと、ハードウェア依存の割込み管理機能は、基本的にはユーザー側で作成すべきモジュールです。サンプルとして付属しているソースファイルを参考にして作成してください。

n4ixxxx.c ・・・・・・・・・・・・・ 割込み管理機能／周期タイマ割込みハンドラソース
nosxxxx.c ・・・・・・・・・・・・・ シリアル入出力ドライバソース（付属しない場合もあり）
nosxxxx.h ・・・・・・・・・・・・・ シリアル入出力ドライバヘッダ（付属しない場合もあり）

その他に、対応プロセッサの内蔵 I/O を定義したヘッダファイル、スタートアップルーチン例、サンプル main ソース、ネットワーク用のサンプルソース、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 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 nocfg4.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。このような場合、コンフィグレータを使用することで ID 番号の自動割付等をおこなうことができます。コンフィグレータに付いては、コンフィグレータ編を参照してください。ここではコンフィグレータを使用しない場合のカーネルコンフィグレーションについて説明します。

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 "nocfg4.h" を記述するだけです。

タスク ID ･･･････････････････････････ 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 ･･･････････････････ 8
タスク優先度上限 ･･･････････････････････ 8
割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、n4rxxx.h に定義されていて、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。
(*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 ～ 32

(*3) この ID は、1 バイトで管理されており、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。また、優先度順の待ち行列を管理する各内部データのサイズが、優先度１毎に１バイト増加します。

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 MBFID_MAX 2     メッセージバッファ ID 上限
#define PORID_MAX 2     ランデブ用ポート ID 上限
#define MPLID_MAX 3     可変長メモリプール 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 MTXID_MAX 1     ミューテックス ID 上限
#define ISRID_MAX 1     割込みサービスルーチン ID 上限
#define SVCFN_MAX 1     拡張サービスコールルーチン ID 上限
#define CYCNO_MAX 2     周期ハンドラ ID 上限
#define ALMNO_MAX 2     アラームハンドラ ID 上限
#define TPRI_MAX 4      タスク優先度上限
#include "nocfg4.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、3 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
#define ALMQSZ 64       アラームハンドラのタイマキューサイズ
    :
#include "nocfg4.h"
```

割込みハンドラのスタックサイズ

割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、全ての割込みハンドラで、このスタック領域を共有します。多重割込みがあるならば、割込みハンドラのスタックサイズに割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      割込みハンドラのスタックサイズ
    :
#include "nocfg4.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（周期起動ハンドラとアラームハンドラ）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 "nocfg4.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。この表は各管理ブロックの最小サイズを示しています。

①	②	
４０	４０	×タスク数
１２	１２	×セマフォ数
１６	１２	×ミューテックス数
１２	８	×イベントフラグ数
１２	１２	×メールボックス数
２４	２４	×メッセージバッファ数
２８	２８	×データキュー数
１２	１２	×ランデブ用ポート数
２０	１６	×可変長メモリプール数
２０	１８	×固定長メモリプール数
３２	２８	×周期ハンドラ数
１２	１２	×アラームハンドラ数
８	８	×拡張サービスコール数
２０	１８	×割込みサービスルーチン数
１６	１４	×タスク例外処理ルーチン数

①ポインタ 32 ビット ,int 型 32 ビットの場合（SH,68K,V800,PowerPC,ARM,MIPS 等）
②ポインタ 32 ビット ,int 型 16 ビットの場合（H8S,H8/300H,8086 等）

タスク優先度順の待ちを指定して生成したオブジェクトの管理ブロックには、（１バイト×タスク優先度上限 TPRI_MAX）のサイズが加算されます。加算した結果のサイズ合計が int サイズで割り切れない場合は、端数分が切り上げられます。また、オブジェクト生成情報が ROM ではなく RAM に存在する場合、オブジェクト生成情報がシステムメモリにコピーされます。

システムメモリ使用量は同時に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として 8 を指定しても同時に生成しなければ 8 個分確保する必要はありません。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 "nocfg4.h"
```

メモリプール用メモリのサイズ

固定長／可変長メモリプールのメモリブロックとメッセージバッファのリングバッファ領域は、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 "nocfg4.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 / 割込みサービスルーチンのスタックは、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 "nocfg4.h"
```

動的なメモリ管理について

システムメモリ、メモリプール用メモリ、および、スタック用メモリにおいて、生成と削除とを繰り返すと、メモリが断片化する可能性は避けられません。すなわち、合計サイズでは足りているのに、連続空き領域のサイズが小さくて、要求サイズが確保できなくなる可能性があります。また、動的なメモリ管理の処理時間は、その時のメモリ割当ての状態に依存します。処理時間の上限を押さえることはできません。

従って、システム起動時にまとめてオブジェクトを生成し、その後は削除 / 生成を繰り返さないプログラミングを推奨します。

カーネルの割込み禁止レベル

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

NORTi の割込みハンドラの割込みレベルは、カーネルの割込み禁止レベル以下でなければなりません。割込みハンドラの優先度を高いままにして、カーネルの割込み禁止レベルだけを下げると、暴走の原因となりますので注意してください。

```
    :
#define KNL_LEVEL 6     カーネルの割込み禁止レベル
    :
#include "nocfg4.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）－ kernel_id.h －

```
#define ID_MainTsk 1
#define ID_KeyTsk  2
#define ID_ConSem  1
#define ID_KeyFlg  1
#define ID_ErrMbf  1
        :
```

－各ソース－

```
#include "kernel.h"
#include "kernel_id.h"

        :
```

コンフィグレータを使用した場合、kernel_id.h は、コンフィグレーションファイルの静的 API を元に自動的に生成されます。

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）－ xxx_id.c －

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

－各ソース－

```
#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、（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。

空き 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 "nocfg4.h"

TASK task1(void)        /* タスク１ */
{
    FLGPTN ptn;

    for (;;)
    {     tslp_tsk(100/MSEC);
          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();               /* 周期タイマ割込み起動 */
    syssta();               /* システム起動 */
}
```

コンパイル例

前ページの sample.c をコンパイル / リンクする例を示します。vecxxx.asm と init.c は、割込みベクタの定義とスタートアップルーチンです。スタートアップルーチンは、コンパイラによってファイル名が異なったり、C の標準ライブラリに含まれていて不要だったりします。n4ixxx.c と n4exxx.lib は、NORTi の周期タイマ割込みハンドラとカーネルライブラリです。standard.lib は、C の標準ライブラリを示しており、コンパイラによってファイル名が異なります。

```
>asm vecxxx.asm
>cc init.c
>cc sample.c
>cc n4ixxx.c
>link vecxxx.obj init.obj sample.obj n4ixxx.obj n4exxx.lib standard.lib
```

以上の例題から、マルチタスクプログラムを作成するにあたって、なんら特別な手順が必要ないことが、理解できると思います。

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 部分とユーザプログラム部分に分けることができます。一般にタスクとタスク例外処理ハンドラはユーザ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

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。

ext_tsk() は省略可能ですが NORTi3 との互換用には記述を推奨します。

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

繰り返すタイプのタスク

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

割込みマスク状態

起動されたタスクは、割込み許可状態です。

タスク例外処理ルーチン

各タスクに一つタスク例外処理ルーチンを定義することができます。タスク例外処理ルーチンの記述は以下の様にします。

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

TEXPTN は、タスク例外要因型で itron.h で定義されています。

３．２ 割込みサービスルーチンと割込みハンドラの記述

概要

ITRON 仕様では、割込みが発生すると、割込みベクタから直接ユーザーの作成した割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込みサービスルーチンを呼び出す方式があります。

割込みハンドラでは、レジスタの待避と復元（NORTiでは、ent_int と ret_int）をユーザーが記述する必要があります。一方、割込みサービスルーチンでは、先に、OS 内部の割込みハンドラがその処理を行っているため、レジスタ待避 / 復元処理をユーザーが記述する必要がありません。つまり普通の C 関数とすることができます。この割込みサービスルーチンは、μITRON4.0 仕様から導入された仕組みです。

割込みハンドラおよび割込みサービスルーチンは、割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、後は割込みを待っているタスクを起床して、実質的な割込み処理を行わせるのが、一般的です。当然ですが、割込み処理の中では、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／削除や可変長メモリブール）も発行できません。

割込みサービスルーチンの記述法

割込みサービスルーチン(ISR)は以下の様な一般の C関数として記述することができます。普通の割込みルーチン同様の配慮をおこなう以外、auto 変数の使用制限等はありません。exinf は ISR 生成時に指定した拡張情報です。

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

割込みマスク状態

割込み禁止／許可の 2値しかない CPU の場合、起動された ISR は、割込み禁止状態です。レベル割込み機能を持った CPU の場合、起動された ISR のレベルは、ハードウェアの割込みのレベルと一致しています。より優先度の高い割込みが発生した場合は、多重割込みが受け付けられます。

割込みハンドラの記述方法

割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 割込み禁止 レベル より 高優先度の 割込みハンドラは 除く)

割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

割込みハンドラを全て C で記述できるようにするため、NORTi では独自の仕様として、割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、割込みハンドラのコンパイルには、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、割込みサービスルーチンを使用するか、割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 命令で打ち消す等々。

割込みマスク状態

割込み禁止／許可の２値しかない CPU の場合、起動された割込みハンドラは、割込み禁止状態です。多重割込みを許す場合は、割込みコントローラの操作により処理中の割込みをマスクした上で、直接 CPU の割込みマスクを変更して、割込み許可にできます。

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

タイムイベントハンドラは、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ます。タイムイベントハンドラを用いると、正確な時間による制御が可能です。また、タスクより管理ブロックやスタックに必要なメモリが少なくて済みます。

ただし、タイムイベントハンドラの中では、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。

タイムイベントハンドラの記述方法

周期ハンドラおよびアラームハンドラの記述に関しては普通の割込みルーチン同様の配慮をおこなってください。タイムイベント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 はタイムイベントハンドラ生成時に指定した拡張情報です。

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

オーバーランハンドラの記述に関しても普通の割込みルーチン同様の配慮をおこなってください。オーバーランハンドラは以下の様な C 関数として記述してください。tskid は、持ち時間を使い切ったタスクのタスク ID、exinf はそのタスク生成時に指定した拡張情報です。

```
void ovrhdr(ID tskid, VP_INT exinf)
{
    :
    :
}
```

割込みマスク状態

全てのタイムイベント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、割込みは許可状態です。タイムイベントハンドラ内で割込み禁止にした場合は、割込み許可状態に戻してからリターンしてください。

補足

タイムイベントハンドラは、割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 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 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 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、周期タイマ割込みの起動とシステム起動 syssta 等をおこないます。

コンフィグレータを使用した場合 main 関数は kernel_cfg.c ファイル内にコンフィグレータによって生成されます。

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 n4ixxx.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。

コンフィグレータを使用する場合は、ATT_INI 静的 API で登録したユーザー関数が呼び出されます。

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。

オブジェクト生成や削除には動的な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１度だけおこなって、その後の生成や削除は最小限にした方がよいでしょう。

コンフィグレータを使用する場合は、CRE_xxx 静的 API で登録したオブジェクト生成がおこなわれます。

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。起動されるタスクは既に生成されている必要があります。

コンフィグレータを使用する場合のタスク起動は、CRE_TSK静的APIのタスク属性にTA_ACTを指定します。

周期タイマ割込み起動

標準的には、intsta 関数で、周期タイマ割込みを起動します。

機種依存する周期タイマ割込み、および割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 n4ixxx.c をコンパイルしてリンクする必要があります。付属の n4ixxx.c では対応できない場合は、ユーザーご自身で n4ixxx.c を作成してください。

コンフィグレータを使用する場合、周期タイマはソフトウェア部品の一つとして扱われます。起動タイミング等はコンフィグレータマニュアルを参照してください。

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、NORTi のデフォルトのアイドルタスクになります）

ただし、syssta 関数を実行する前の、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

コンフィグレータを使用しない場合の記述例を以下に示します。

```
#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       /* カーネルの割込み禁止レベル */
#include "nocfg4.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）*/

void 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();                   /* 周期タイマ割込みを起動 */
    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。タスク削除は、自タスクを終了削除する exd_tsk、他タスクを削除する del_tsk とに分かれています。

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。

NORTi3 との差異

タスクの起動要求 act_tsk、起動要求をキャンセルする can_act、タスク状態を参照する ref_tst が追加されました。
スタック領域をユーザ領域に確保できる機能が追加されました。
タスク生成後実行可能状態にするオプションが追加されました。
現在優先度の概念が導入されました。
現在優先度を参照する get_pri が追加になりました。
タスク名を設定できるようになりました。
タスクのメインルーチンからのリターンでタスクを終了できるようになりました。
dis_dsp, ena_dsp, rot_rdq, get_tid, rel_wai の機能分類が変更されました。
vcre_tsk が acre_tsk に名称変更されました。
vsta_tsk は廃止されました。sta_tsk を使用してください。

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 では、nosys4.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。

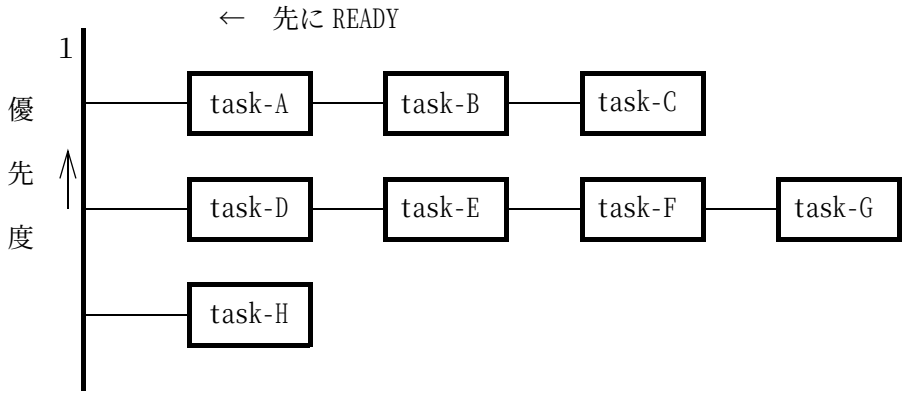
スケジューリングとレディキュー

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 SUSPENDED 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。



タスク優先度順オブジェクト待ち行列も、レディキューと同様の方法で実現しています。

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、sus_tsk，rsm_tsk，frsm_tsk，lp_tsk，tslp_tsk，wup_tsk，can_wup，rel_wai，dly_tsk システムコールです。

NORTi3 との差異

dly_tsk がタスク付属同期機能に分類されました。
can_wup のリターン値としてキューイングされていた起床要求数を返します。
ディスパッチ可能であれば sus_tsk で自タスクを指定できるようになりました。
wup_tsk で自タスクを指定できるようになりました。
rel_wai がタスク付属同期機能に分類されました。

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_msg 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。

中断と再開

他のタスクの実行を中断して、強制待ち状態 SUSPENDED へ移行させるシステムコールとして sus_tsk があります。

強制待ち状態にあるタスクは、他からの rsm_tsk または frsm_tsk システムコールにより再開されます。rsm_tsk と frsm_tsk の違いはキューイングの扱いにあります。frsm_tsk では、キューイングをすべてキャンセルしてタスクの実行を再開しますが、rsm_tsk ではキューイングされていた場合、キューイング数を -1 するだけです。

二重待ち状態

待ち状態 WAITING にあるタスクに対して sus_tsk システムコールを発行すると、二重待ち状態 WAITING-SUSPENDED へ移行します。

WAITING-SUSPENDED 状態でも、WAITING 状態と同様に、順番が来れば資源の割り当てがおこなわれます。そして資源が割り当てられると、WAITING-SUSPENDED 状態から SUSPENDED 状態に移行します。

すなわち、WAITING-SUSPENDED 状態のタスクに対する特別措置（資源割り当て遅延等）はおこなわれませんから注意してください。

４．３ タスク例外処理機能

概要

タスク例外処理機能は、指定したタスクに実行中の処理を中断させ、タスク例外処理ルーチンを実行させるための機能です。タスク例外処理ルーチンは中断されたタスクのコンテキストで実行されます。指定したタスクが、WAITING 状態などで実行中で無い場合には例外処理ルーチンの実行もおこなわれず、指定タスクが RUNNING 状態になるまで待たされます。RUNNING 状態になると、タスク本体ではなく例外処理ルーチンが先に実行され、例外処理ルーチンからリターンすることでタスク本体の処理が継続されます。タスク例外処理ルーチンは各タスクに一つ登録することができます。

タスクに対して例外処理ルーチンを定義するための def_tex、例外処理を要求する ras_tex、例外処理を禁止する dis_tex、許可する ena_tex、禁止状態か否かを参照する sns_tex、例外処理状態を参照する ref_tex システムコールがあります。

NORTi3 との差異

μITRON4.0 から導入された機能です。

例外処理ルーチンの起動と終了

タスク例外処理ルーチンを起動する場合には、要求する例外処理の種類を表す例外要因を指定して ras_tex を呼びます。実際に例外処理ルーチンが起動されるのは、ena_tex により例外処理が許可され、例外要因が 0 以外で、指定されたタスクが RUNNING 状態になったときです。例外処理ルーチンが起動されると、例外要因は 0 クリアされ、例外処理は禁止状態になります。例外処理ルーチンからリターンすると、例外処理ルーチンを起動する前に実行していた処理を継続します。この時、例外処理許可状態に戻ります。

タスク例外処理ルーチンからリターンせずに longjmp を用いて大域脱出した場合は、例外処理を継続している状態であり例外処理許可状態に戻りません。また、例外処理ルーチンを起動する前の情報は失われます。たとえば、rcv_mbf で WAITING していた場合、受信したメッセージの情報は失われます。longjmp した場合は、タスクを終了するようにしてください。

例外要因

例外要因が 0 以外の時を例外処理要求ありと定義します。例外処理禁止状態のとき例外要求があると、例外要求は例外処理許可になるまで保留されます。例外要因は TEXPTN 型の変数で実装されています。要求の保留は論理和を取ることでおこなわれます。したがって同一の要求が複数回上がっても OS 機構としてタスク例外処理ルーチンは要求回数を認識できません。

４．４ 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。

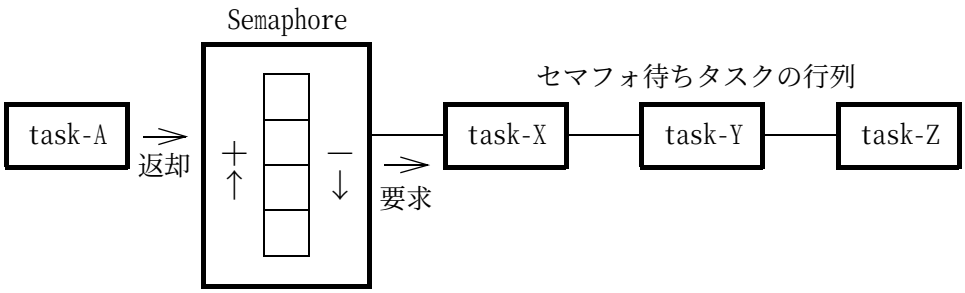
セマフォの生成と削除は cre_sem, acre_sem と del_sem でおこないます。資源の返却をおこなう sig_sem に対し、資源の獲得待ちをおこなう wai_sem、待たずにポーリングをおこなう pol_sem、タイムアウト付きで待つ twai_sem があります。その他に、セマフォの状態を参照する ref_sem システムコールがあります。

NORTi3 との差異

preq_sem が pol_sem に名称変更されました。
生成情報から拡張情報が削除されました。
ref_sem で参照できる情報から、拡張情報が削除されました。
ref_sem で参照できる情報で待ちタスクが無い場合、FALSE でなく TSK_NONE が返ります。
vcre_sem が acre_sem に名称変更されました。

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ます。セマフォ生成時に FIFO を指定した場合は、先に要求した順番で、待ち行列をつくります。セマフォ生成時に優先度順を指定した場合は、タスクの優先度順（同一優先度なら先に要求した順）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。

４．５ 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と削除は cre_flg，acre_flg と del_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。

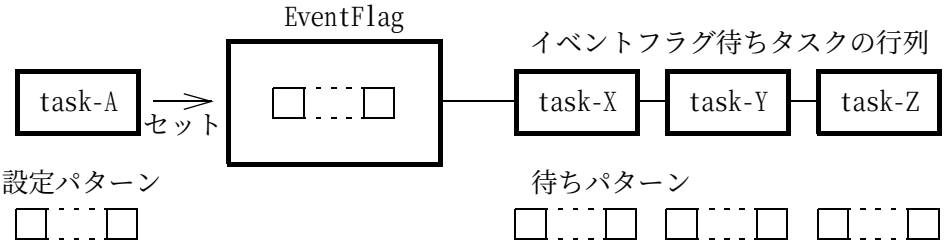
NORTi3 との差異

イベントフラグのクリア指定がwai_flgの待ちモードで指定する方法のほかに生成情報でも指定できるようになりました。
複数タスク待ちイベントフラグでタスク優先度順オプションが使えるようになりました。
生成情報から拡張情報が削除されました。
ref_flg で参照できる情報から拡張情報が削除されました。
ref_flg で参照できる情報で待ちタスクが無い場合、FALSE でなく TSK_NONE が返ります。
vcer_flg が acre_flg に名称変更されました。

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ができます。これらのタスクの待ち条件が同じなら、１回の set_flg で一斉に待ちを解除できます。ただし、クリア指定がされている時、それより後ろにつながれているタスクの待ちは解除されません。

ただし、同時に複数のタスクの待ちが解除される場合、システムコール処理時間の上限が押さえられませんので、複数タスク待ち機能は、なるべく使わないことを推奨します。複数タスク待ちを許すか否かは、イベントフラグ生成時に指定できます。



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。

４．６ 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と削除は cre_dtq，acre_dtq と del_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。

NORTi3 との差異

μITRON4.0 から導入された機能です。

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。

送信待ち行列は、到着順 (FIFO) かタスク優先度順を指定することができます。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。

４．７ 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と削除は cre_mbx，acre_mbx と del_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。

NORTi3 との差異

メールボックス生成情報から拡張情報が削除されました。

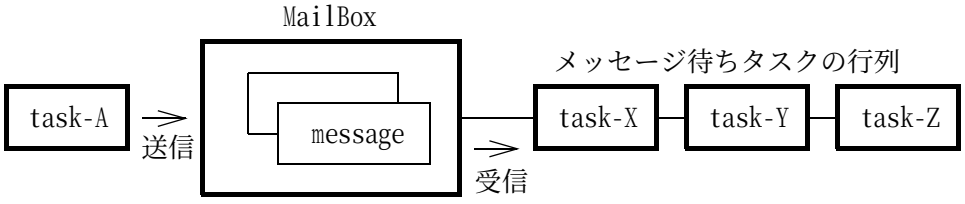
ref_mbx で参照できる情報から、拡張情報が削除されました。

vcre_mbx が acre_mbx に名称変更されました。

システムコール名称が XXX_msg から XXX_mbx に変わりました。

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ます。メールボックス生成時に FIFO を指定した場合は、先に要求した順番で待ち行列をつくります。メールボックス生成時に優先度順を指定した場合は、タスクの優先度順（同一優先度なら先に要求した順）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。

メールボックス生成時に、メッセージのキューイング方法を FIFO に指定した場合は、先に送信された順番で、メッセージが行列をつくります。

メールボックス生成時に、メッセージのキューイング方法を優先度順に指定した場合は、メッセージが優先度別の行列をつくります(同一優先度内では送信された順に並びます)。したがって、優先度数を多くすると必要メモリ量が増加します。必要メモリ量は、TSZ_MPRIHD マクロによって知ることができます。

```
mprihdsz = TSZ_MPRIHD(8);
```

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と組み合わせる場合は、必然的にメッセージパケットのサイズが固定されます。

４．８ 拡張同期・通信機能（ミューテックス）

概要

ミューテックスは、セマフォと同様に資源の排他制御に使用します。セマフォとの違いは、優先度逆転を防ぐ機構をサポートしていることと、資源をロックしたままタスクが終了すると自動的にロック解除する点です。逆に、セマフォは資源が複数ある場合に使用するカウンタを持っていることと、ロックしたタスク以外のタスクでもロック解除できる特徴があります。

ミューテックスの生成と削除は cre_mtx, acre_mtx と del_mtx でおこないます。資源の返却を行う unl_mtx に対し、資源の獲得待ちをおこなう loc_mtx、待たずにポーリングをおこなう ploc_mtx、タイムアウト付きで待つ tloc_mtx があります。その他に、ミューテックスの状態を参照する ref_mtx システムコールがあります。

NORTi3 との差異

μITRON4.0 から導入された機能です。

優先度逆転

優先度の低いタスクが資源をロックしているときに、優先度の高いタスクが既にロックされている資源を使おうとして待ち状態になることがあります。この時、中間の優先度を持つタスクが RUNNING 状態になると、間接的に優先度の高いタスクの実行を中間の優先度のタスクがプリエンプトしてしまう結果になります。このことを優先度逆転と呼びます。優先度逆転が起こると優先度ベースのスケジューリングを前提に設計されたシステムの動作を保証できません。

ミューテックスでは、優先度逆転を防ぐために優先度継承プロトコルと優先度上限プロトコルをサポートしています。

優先度継承プロトコルは、ロックしているタスクの優先度をロック待ちしているタスクの中で最高の優先度を持っているタスクと一時的に同一にします。こうすることで中間の優先度を持ったタスクの介入を排除します。動的に優先度を変更するためシステムの負荷が大きくなります。特にロック中のタスクが別のミューテックスのロック待ちをしていた場合には優先度変更が遷移的に起こるので注意が必要です。

優先度上限プロトコルは、ロックしたタスクの優先度を、待ちタスクの有無とは無関係にあらかじめ決めた優先度に変更するものです。優先度継承ほど負荷は高くありませんが、待ちタスクが無くとも優先度変更が発生します。

一時的に変更された優先度はロック解除によってもとの優先度に戻ります。

４．９ 拡張同期・通信機能（メッセージバッファ）

概要

メッセージバッファはタスク間での比較的小さなデータの受け渡しに使用します。メールボックスと異なるのは、メッセージの内容が内部のリングバッファへコピーされて送受信されることです。したがって、メールボックスのように、メッセージパケット領域をメモリプールから獲得する必要はありません。なお、割込み禁止状態でコピーを実行しているので、大きなデータを渡すと割込み禁止時間が延びますので、注意してください。

メッセージバッファの生成と削除は cre_mbf, acre_mbf と del_mbf でおこないます。メッセージを送信する snd_mbf、バッファに空きが無い場合待たずに直ちにリターンする psnd_mbf、バッファに空きが無い場合タイムアウト付きで待つ tsnd_mbf、メッセージの受信待ちをおこなう rcv_mbf、メッセージが無い場合に待たずにポーリングをおこなう prcv_mbf、同様にタイムアウト付きで待つ trcv_mbf があります。その他に、メッセージバッファの状態を参照する ref_mbf システムコールがあります。

NORTi3 との差異

送信待ちタスクに対しても優先度待ちを指定できるようになりました。

vcer_tsk が acre_tsk に名称変更されました。

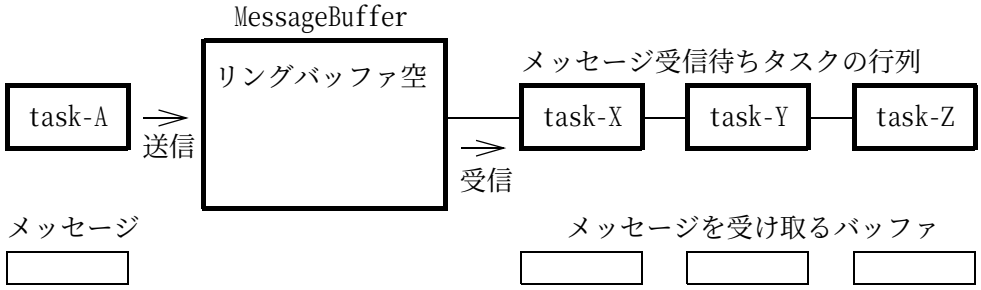
メッセージキュー

メッセージデータは、メッセージバッファ内部のリング状のバッファにコピーされます。メールボックスのように、メッセージパケット領域をメモリプールから獲得する必要はありません。また、OS が使用するメッセージヘッダ部分も必要ありません。

メッセージサイズは、メッセージバッファ生成時に指定した最大長を超えない限り、任意です。受信側では、最大長のメッセージを受け取れるバッファを用意する必要があります。キューイングされたメッセージの並びは、FIFO のみです。メッセージに優先度を付ける機能はありません。

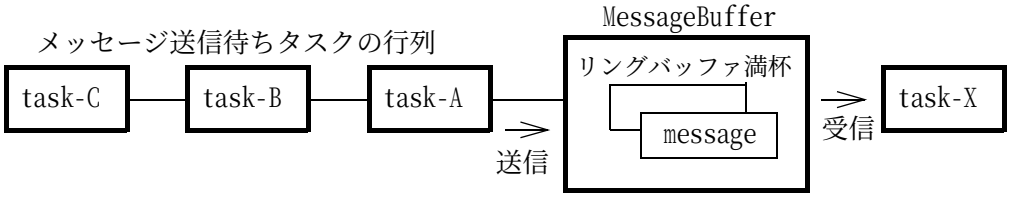
メッセージ受信待ち行列

複数のタスクが同じメッセージバッファで待つことができます。メッセージバッファ生成時に FIFO を指定した場合は、先に要求した順番で、待ち行列をつくります。メッセージバッファ生成時に優先度順を指定した場合は、タスクの優先度順（同一優先度なら先に要求した順）で、待ち行列をつくります。



メッセージ送信待ち行列

メールボックス機能との違いは、リングバッファに空きがない場合に、送信側のタスクも WAITING 状態となることです。複数の送信待ちタスクがある場合、メッセージバッファ生成時に FIFO を指定した場合は、先に送信要求した順番で、待ち行列を作ります。メッセージバッファ生成時に優先度順を指定した場合は、タスクの優先度順（同一優先度なら先に要求した順）で、待ち行列をつくります。



リングバッファ領域

リングバッファへは、メッセージサイズを示す２バイトのヘッダが付加されてメッセージデータがコピーされます。したがって、リングバッファ領域の全てを、データ用として使うことはできません。一つのメッセージサイズが msgsz バイトのメッセージを msgcnt 個格納できるリングバッファサイズは TSZ_MBF マクロによって得ることができます。ただし、msgsz が１以外の場合です。

TSZ_MBF(msgcnt, msgsz)

msgsz が１の場合、すなわちメッセージの最大長を１バイトとしてメッセージバッファを生成した場合に限って、メッセージサイズを示すヘッダの付加を省略します。この機能により、１バイトのメッセージ送受信では、リングバッファ領域の全てをデータ用として効率的に利用できます。

サイズ０のリングバッファ

リングバッファの総サイズをゼロとして、メッセージバッファを生成することもできます。この場合、送信メッセージは、受信側のタスクが用意したバッファへ直接コピーされます。そのため、受信側のタスクが現れるまで、送信側のタスクは待ち状態になります。この機能により、メッセージバッファでも、ランデブ機能に似た同期通信を実現することができます。

４．１０ 拡張同期・通信機能（ランデブ用ポート）

概要

ランデブ機能は、タスク間の密接な同期を行うために使用します。相互にデータの受け渡しを行うことも可能です。ランデブ（直訳：会合する）という言葉から分かるように、２つのタスクが互いに待ち合わせをおこないます。応答を待つ本機能に比べると、他の同期・通信機能は、一方的な待ちや通信と言えます。

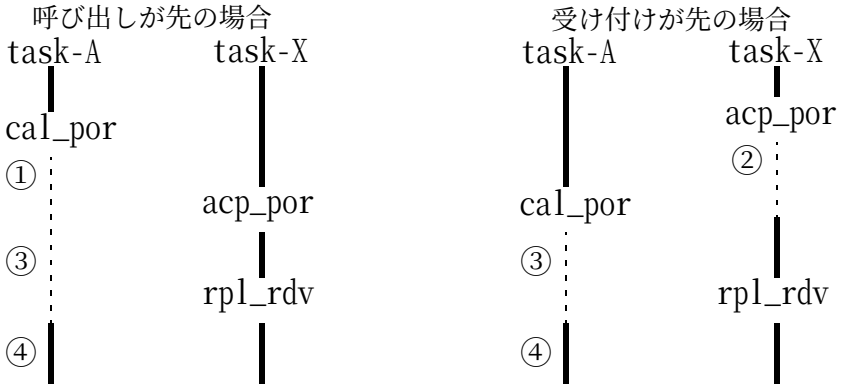
ポートの生成と削除は cre_por，acre_por と del_por でおこないます。ランデブ呼び出し cal_por に対し、ランデブ受け付け acp_por とランデブ返答 rpl_rdv があります。受け付けには待たないでポーリングする pacp_por、および、ランデブ呼び出し / ランデブ受け付けには、タイムアウト付きで待つ tcal_por/tacp_por もあります。その他に、受け付けたランデブを別のポートへ回送する fwd_por、ポートの状態を参照する ref_por、ランデブの状態を参照する ref_rdv システムコールがあります。

NORTi3 との差異

ランデブ呼出待ちにタスク優先度順が追加されました。
ランデブ生成情報から拡張情報が削除されました。
tcal_por のタイムアウト時間はランデブが「成立するまで」ではなく「終了するまで」の時間になりました。これに伴い、pcal_por は廃止されました。
呼出メッセージサイズは acp_por 関数の引数から、返値に変更されました。
ランデブ相手がランデブ待ちか調べる ref_rdv が追加されました。
ランデブ受付け条件として 0 を指定するとエラー (E_PAR) になります。
vcre_por が acre_por に名称変更されました。

ランデブの基本的な流れ

task-A と task-X が、下図のようにランデブを行う例で説明します。点線は、WAITING 状態であることを示します。



task-A がランデブ呼び出し cal_por をおこなった時に、まだ task-X がランデブ受け付け acp_por をおこなっていなければ、task-A はランデブ呼出待ち状態①となります。

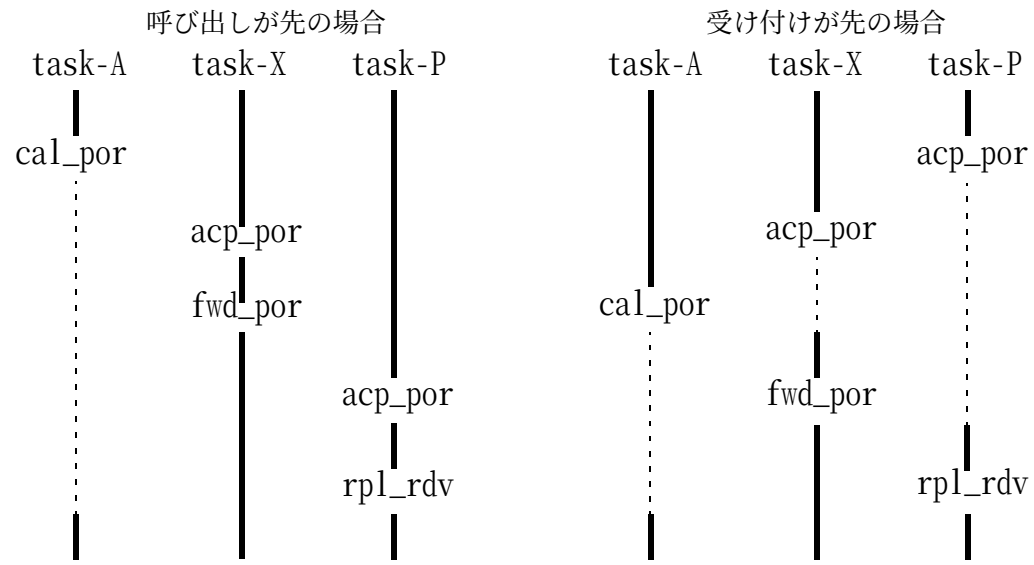
逆に、task-X がランデブ受け付け acp_por をおこなった時に、まだ task-A がランデブ呼び出し cal_por をおこなっていなければ、task-X はランデブ受付待ち状態②となります。

呼び出しと受け付けが揃うと、task-A はランデブ終了待ち状態③となります。task-X は実行を続け、ランデブ返答 rpl_rdv をおこなった時点で、task-A の待ちが解除④され、ランデブ終了となります。

ランデブ回送

受け付けたランデブを、fwd_por によって別のポートへ回送することができます。

下図は、task-X が別ポート回送したランデブを、task-P が受け付けて返答をおこなう例です。



ランデブ成立条件

イベントフラグの様なビットパターンで、呼出側選択条件と受付側選択条件を指定することができます。呼出側選択条件のビットパターンと受付側選択条件のビットパターンとの論理積（AND）が非ゼロの場合に、ランデブ成立となります。

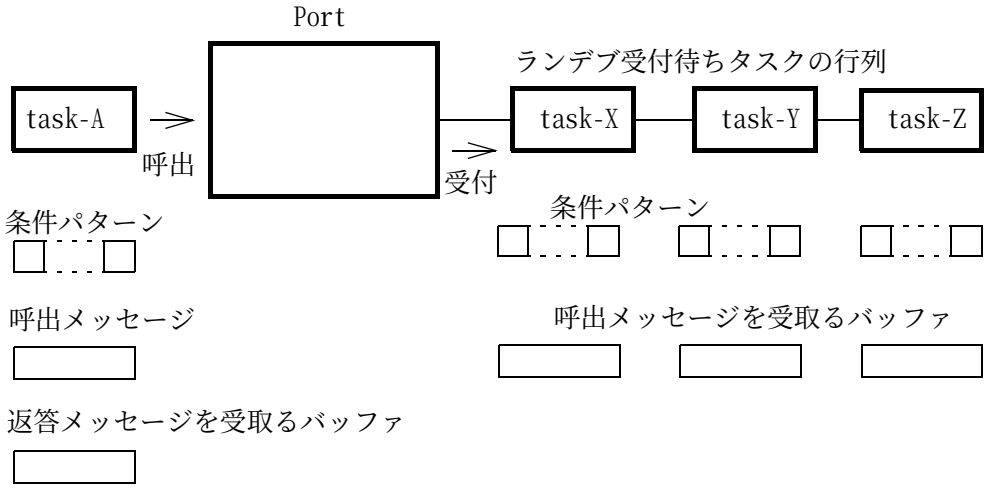
メッセージ

呼出メッセージは、ランデブ成立時に、呼出側タスクから受付側タスクへ渡されます。返答メッセージは、ランデブ終了時に、受付側（返答側）タスクから呼出側タスクへ渡されます。

メッセージは、それぞれのタスクが用意したバッファ間でコピーされます。メッセージバッファ機能と似ていますが、ランデブという同期方法の性質上、メッセージキューは存在しません。なお、割込み禁止状態でコピーを実行しているので、大きなデータを渡すと割込み禁止時間が延びますので、注意してください。

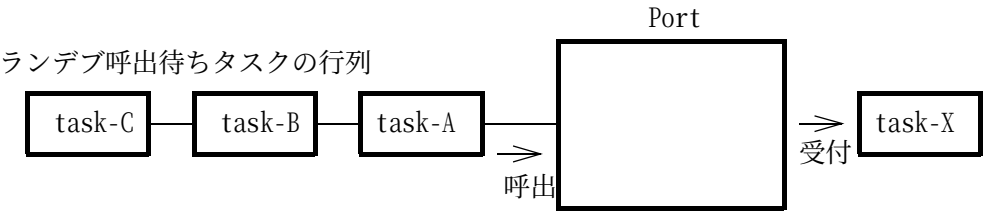
ランデブ受付待ち行列

複数のタスクが同じポートで受け付けを待つことができます。呼出側のタスクがない場合や、ランデブが成立しない場合、先に受け付けを発行した順番で、待ち行列をつくります。タスク優先度順にすることはできません。



ランデブ呼出待ち行列

複数のタスクが同じポートで呼び出しを待つことができます。受付側のタスクがない場合や、ランデブが成立しない場合、先に呼び出しを発行した順番あるいはタスク優先度順で、待ち行列をつくります。



４．１１ 割込み管理機能

概要

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

NORTi3 との差異

loc_cpu，unl_cpu はシステム状態管理機能に分類されました。
def_int は def_inh に名称変更されました。
ref_ims は get_ims に名称変更されました。
ret_wup は廃止されました。
cre_isr，acre_isr，del_isr，ref_isr は新設のシステムコールです。

割込みハンドラおよび割込みサービスルーチンの定義

割込みハンドラを定義する def_inh、割込みサービスルーチンに関連した cre_isr，acre__isr，del_isr システムコールでは、割込みベクタの設定をおこないます。しかし、割込みの設定方法はシステムにより異なりますので、カーネルにはこれらのシステムコールを含めていません。付属の n4ixxx.c に定義されているこれら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。

特定の割込みの禁止／許可

μITRON 仕様にある、特定の割込みを禁止／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 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。)

割込みハンドラの起動

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

そして NORTi では、割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込みハンドラ専用のスタック領域に切り替えています。

割込みサービスルーチンの起動

割込みサービスルーチンを登録した割込みが発生するとまずカーネル内の割込みハンドラにコントロールが移りそこからユーザの記述した割込みサービスルーチンを呼び出しています。

RISC プロセッサの割込み

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、割込みベクタテーブルの代わりに、n4ixxx.c に定義してある配列へ、割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、カーネルの割込み禁止レベルより高優先度の割込みルーチンは 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込みルーチン

カーネルの割込み禁止レベルより高いレベルの割込みルーチンを使うことができます。この割込みルーチンにとって、カーネル内部の割込み禁止区間は割込み許可と同じことになり、非常に高速な割込み応答が要求されるシステムでも NORTi 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込みルーチンでは、システムコールを発行できません。割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、高優先の割込みルーチンからカーネルのレベル以下の割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。

４．１２ メモリプール管理機能

概要

NORTi のメモリ管理機能は、可変長のメモリブロック、および、固定長のメモリブロックを扱う２種類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。

可変長メモリプールの生成と削除は cre_mpl，acre_mpl と del_mpl でおこないます。メモリブロックを返却する rel_mpl に対し、メモリブロックの獲得待ちをおこなう get_mpl、待たずにポーリングをおこなう pget_mpl、タイムアウト付きで待つ tget_mpl があります。その他に、可変長メモリプールの状態を参照する ref_mpl システムコールがあります。

固定長メモリプールの生成と削除は cre_mpf，acre_mpf と del_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。

NORTi3 との差異

rel_blk，get_blk，pget_blk，tget_blk はそれぞれ XXX_mpl に名称変更されました。

rel_blf，get_blf，pget_blf，tget_blf はそれぞれ XXX_mpf に名称変更されました。

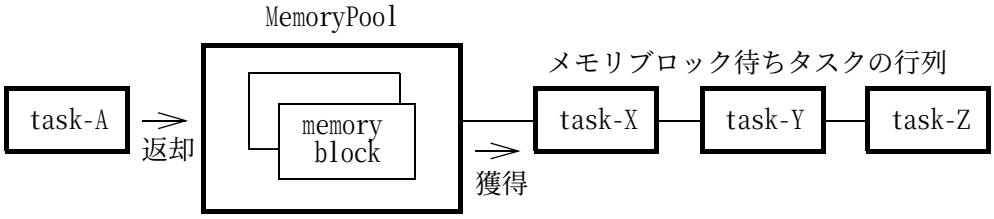
vcre_mpl，vcre_mpf はそれぞれ acre_XXX に名称変更されました。

生成情報から拡張情報が削除されました。

ref_mpl，rel_mpf で参照できる情報から、拡張情報が削除されました。

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ます。メモリプール生成時に FIFO を指定した場合は、先に要求した順番で、待ち行列をつくります。メモリプール生成時に優先度順を指定した場合は、タスクの優先度順（同一優先度なら先に要求した順）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。

可変長と固定長

可変長メモリプールの方が固定長メモリプールより便利ですが、動的なメモリ管理を伴うため、可変長メモリプールは重たい部類のシステムコールです。固定長で済む用途には、固定長メモリプールの方を使うことを推奨します。

可変長メモリプールでは、１つのメモリブロックを獲得する際に、そのサイズを記憶するために int サイズのメモリを余分に消費します。それに対し、固定長メモリブロックでは、無駄なメモリ消費がありません。

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。

４．１３ 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，del_cyc，sta_cyc，stp_cyc，ref_cyc，cre_alm，acre_alm，del_alm，sta_alm，stp_alm，ref_alm，def_ovr，sta_ovr，stp_ovr，ref_ovr，isig_tim システムコールです。

NORTi3 との差異

システムが使用するシステムクロックとは別にユーザ用にシステム時刻が導入されました。
set_tim，get_tim はシステム時刻を設定・参照するように仕様変更されました。
タスク実行時間を監視するオーバーランハンドラが導入されました。
周期ハンドラに起動位相を扱う機能が追加されました。
周期ハンドラ生成情報から cycact が削除されました。生成時の動作状態は停止状態です。
絶対時刻でアラームハンドラを起動する機能は廃止されました。
アラームハンドラの解除は自動的におこなわれません。
act_cyc が sta_cyc と stp_cyc に分割されました。
def_cyc が cre_cyc と del_cyc に分割されました。
acre_cyc が新設されました。
def_alm が cre_alm に変更されました。
del_alm が新設されました。
sta_alm，stp_alm が新設されました。
ret_tmr が廃止されました。

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、del_cyc により取り消しをおこないます。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。

アラームハンドラ

指定した時間後に１度だけ起動実行されるされるタイムイベントハンドラです。

アラームハンドラは、cre_alm，acre_alm システムコールにより登録し、del_alm により取り消しをおこないます。登録直後は起動時刻が設定されておらず sta_alm により設定をおこないます。設定を解除は stp_alm によりおこないます。アラームハンドラが起動されると自動的に設定が解除されますが、登録の取り消しはおこないません。他、起動までの時間を調べる ref_alm システムコールがあります。

オーバーランハンドラ

タスク毎に設定された実行時間を監視し、設定された時間を使い切った場合に起動されるタイムイベントハンドラです。実行時間の監視はシステムクロックを用いておこなっています。したがって監視対象タスクの連続実行時間がシステムクロックインタバル以下の場合実行時間を充分な精度で監視することができません。実行条件等により無限ループに陥る可能性のあるタスクの監視に使用してください。

オーバーランハンドラはシステムに一つだけdef_ovrによって登録および解除することができます。sta_ovr によって監視するタスクとそのタスクの持ち時間を設定します。複数のタスクに対して設定することが可能です。監視を解除する場合は stp_ovr を使用します。ref_ovr によってオーバーランハンドラの動作状態と指定したタスクの残り時間を参照することができます。

４．１４ 拡張サービスコール管理機能

概要

サービスコール管理機能によって拡張サービスコールの定義と呼出をおこなうことができます。拡張サービスコールは、ダイナミックにロードしたモジュールやファームウェアに置かれたモジュールなどのシステム全体を一つにリンクしない場合のモジュールを呼び出すための機能です。

def_svc で拡張サービスコールの登録／解除をおこない、cal_svc により登録したルーチンを呼び出します。

NORTi3 との差異

μITRON4.0 から導入された機能です。

拡張サービスコールルーチンの記述

```
ER_UINT svcrtn(VP_INT par1, VP_INT par2, ..., VP_INT par6)
{
    :
    :
}
```

C 言語により上記の様な形式でサービスコールルーチンを記述してください。パラメータ数は 0 ～ 6 個まで指定できます。

４．１５ 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、レディー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。

NORTi3 との差異

新設の機能分類です。

get_tid を非タスクコンテキスト部から呼んだ場合 FALSE ではなく RUNNING 状態のタスク ID が返ります。

CPU ロック状態とディスパッチ禁止状態は独立に操作できるようになりました。

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込みを禁止することもできます。

４．１６ 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。

NORTi3 との差異

get_ver は、ref_ver に名称変更されました。

未サポート機能

NORTi では、CPU 例外ハンドラ定義 def_exc は未サポートです。

※ 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。

*** パラメータチェック無しライブラリでは検出しない静的なエラー。**
標準のライブラリではチェックされ SYSER 変数に記録される。

**** いずれの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 エラー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
    B *name;        タスク名へのポインタ
} 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るほか、オーバーランハンドラにも渡されます。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****

注　意　タスク生成情報パケットは、タスク管理ブロックへコピーされないので、本システムコール発行後も保持する必要があります。const 変数として定義し ROM に配置してください。ROM 以外に配置された場合には、実行中に変更または廃棄された場合の動作異常を防ぐために、システムメモリにタスク生成情報パケットのコピーを作成します。

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
```
正の値ならば、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, "" };

TASK task1(void)
{
    ER_ID ercd;
      :
    ercd = acre_tsk(&ctsk2);
    if (ercd > 0)
        ID_task2 = ercd;
      :
}
```

ｄｅｌ＿ｔｓｋ

機　能　タスク削除

形　式
```
ER del_tsk(ID tskid);
tskid タスク ID
```

解　説　tskid で指定されたタスクを削除します。すなわち、このタスクのスタック領域をスタック用メモリへ解放し、タスク管理ブロック（TCB）をシステムメモリへ解放します。削除した結果、対象タスクは DORMANT 状態から NON-EXISTENT 状態へ遷移します。このシステムコールでは、自タスクは指定できません、exd_tsk を使用してください。

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_CTX   割込みハンドラから発行 *
E_NOEXS タスクが生成されていない
E_OBJ   タスクが DORMANT 状態でない
```

注　意　対象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、タスク削除の前に資源を解放してください。

例
```
#define ID_task2 2

TASK task1(void)
{
      :
    del_tsk(ID_task2);
      :
}
```

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。

tskid = TSK_SELF で自タスクを指定できます。

戻　値
```
0 または正の値ならばキューイングされていた起動要求数 (actcnt)
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, "task2" };

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、タスクがミューテックスをロックしていた場合はアンロックし、オーバーランハンドラへの登録が解除され、タスク例外処理が禁止され、優先度・起床要求数・強制待ち要求数・保留例外要因・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

```
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。

ｅｘｄ＿ｔｓｋ

機　能　自タスクの終了と削除

形　式　**void exd_tsk(void);**

解　説　タスク自ら終了し、削除されます。すなわち、タスクのスタック領域をスタック用メモリへ解放し、タスク管理ブロック（TCB）をシステムメモリへ解放します。削除された結果、タスクは RUNNING 状態から、直接 NON-EXISTENT 状態へ遷移します。キューイングされていた起動要求はキャンセルされます。

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で、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

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

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 または WAITING-SUSPEND 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　タスクベース優先度変更

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ベース優先度を 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ベース優先度、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、タスク起動時にベース優先度にコピーされます。タスクは通常ベース優先度で走行しますが、ミューテックスをロックした場合に一時的に優先度が変更される場合があります。一時的に変更された優先度が現在優先度です。ミューテックスをアンロックした時点でタスク優先度はベース優先度に戻ります。chg_pri はこのベース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 で初期優先度とすることができます。

対象タスクが優先度順の待ち行列（レディーキューあるいはセマフォやメモリプール等の優先度順待ち行列）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。現在優先度が変更された場合にも待ち行列のつなぎ替えが起こります。ミューテックスを使用した場合には遷移的（芋づる式）に待ち行列のつなぎ替えが起こるため注意が必要です。

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, 1);           /* 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
    PRI tskbpri;           ベース優先度
    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
    UINT suscnt;           強制待ち要求カウント
    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_msg または trcv_msg による待ち
TTW_MTX	0x0080	loc_mtx による待ち
TTW_SMBF	0x0100	snd_mbf または tsnd_mbf による待ち
TTW_RMBF	0x0200	rcv_mbf または trcv_mbf による待ち
TTW_CAL	0x0400	ランデブ呼出待ち
TTW_ACP	0x0800	ランデブ受付待ち
TTW_RDV	0x1000	ランデブ終了待ち
TTW_MPF	0x2000	固定長メモリブロックの獲得待ち
TTW_MPL	0x4000	可変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

ｓｕｓ＿ｔｓｋ

機　能　タスクを強制待ち状態へ移行

形　式
```
ER sus_tsk(ID tskid);
tskid   タスク ID
```

解　説　tskid で指定されたタスクの実行を抑制します。すなわち、対象タスクが READY 状態ならば、SUSPENDED 状態へ遷移させます。対象タスクが WAITING 状態ならば、WAITING-SUSPENDED 状態へ遷移させます。tskid = TSK_SELF で自タスクを指定できます。

この強制待ち状態は、rsm_tsk, frsm_tsk システムコールにより解除されます。強制待ち要求はネストさせることができます。すなわち、sus_tsk の発行回数と同一回数の rsm_tsk の発行で、はじめて SUSPENDED 状態が解除されます。

戻　値
```
E_OK    正常終了
E_ID    タスク ID が範囲外 *
E_CTX   ディスパッチ禁止状態で自タスクを指定（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  強制待ち要求数のオーバフロー（TMAX_SUSCNT = 255 を超える）
```

例
```
#define ID_task2 2

TASK task1(void)
{
      :
    sus_tsk(ID_task2);
      :
}
```

ｒｓｍ＿ｔｓｋ

機　能　強制待ち状態のタスクを再開

形　式
```
ER rsm_tsk(ID tskid);
```
tskid タスク ID

解　説　tskid で指定されたタスクの実行抑制を解除します。すなわち、対象タスクが SUSPENDED 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 WAITING-SUSPENDED 状態だった場合、対象タスクは WAITING 状態へ遷移します。

rsm_tsk では、sus_tsk １回分の強制待ち要求を解除します。つまり、対象タスクに２回以上の sus_tsk が発行されていた場合は、rsm_tsk を１回実行した後も、対象タスクは強制待ち状態のままです。

本システムコールでは、自タスクを指定することはできません。

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 SUSPENDED 状態でない
```

例
```
#define ID_task2 2

TASK task1(void)
{
      :
    sus_tsk(ID_task2);
      :
    rsm_tsk(ID_task2);
      :
}
```

ｆｒｓｍ＿ｔｓｋ

機　能　強制待ち状態のタスクを強制再開

形　式
```
ER frsm_tsk(ID tskid);
tskid   タスク ID
```

解　説　tskid で指定されたタスクの実行抑制を解除します。すなわち、対象タスクが SUSPENDED 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 WAITING-SUSPENDED 状態だった場合、対象タスクは WAITING 状態へ遷移します。

frsm_tsk は、強制待ち要求を全て解除します。つまり、対象タスクに２回以上の sus_tsk が発行されていた場合でも、frsm_tsk の１回の実行で強制待ち状態を解除できます。

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 SUSPENDED 状態でない
```

例
```
#define ID_task2 2

TASK task1(void)
{
      :
    sus_tsk(ID_task2);
    sus_tsk(ID_task2);
      :
    frsm_tsk(ID_task2);
      :
}
```

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。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 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、0 ～ -MSEC の誤差があります。例えば MSEC = 10 の時に 100 msec のタイムアウトを指定すると、実際には 90 ～ 100 msec の範囲でタイムアウトします。

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 wupcnt = can_wup(ID tskid);
wupcnt   キューイングされていた起床要求回数
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
```
0 または正の値ならば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 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 rel_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 WAITING-SUSPENDED 状態だった場合、対象タスクは SUSPENDED 状態へ遷移します。

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 タスク例外処理機能

ｄｅｆ＿ｔｅｘ

機　能　タスク例外処理ルーチンの定義

形　式
```
ER def_tex(ID tskid, const T_DTEX *pk_dtex);
tskid   タスク ID
pk_dtex タスク例外処理ルーチン定義情報パケットへのポインタ
```

解　説　tskidで指定されたタスクに対してタスク例外処理ルーチンを定義します。pk_dtexにNULLを指定すると定義解除します。また、別の定義情報を指定すると再定義します。再定義の場合は、例外処理要求・例外処理許可 / 禁止状態を継承します。tskid = TSK_SELF で自タスクを対象タスクにします。

タスクが再起動された場合、例外処理要求はクリアされ、例外処理禁止状態になります。タスクが削除された場合、タスク例外処理ルーチン定義は解除されます。

タスク例外処理ルーチン定義情報は、次の通りです。

```
typedef struct t_dtex
{   ATR texatr;      タスク例外処理ルーチン属性
    FP texrtn;       タスク例外処理ルーチン起動番地
} T_DTEX;
```

texatr の内容に OS は感知しませんが、他の μITRON との互換性を維持するために TA_HLNG を指定してください。定義情報パケットを ROM 以外に置いた場合、すなわち const を付けなかった場合、定義情報パケットはシステムメモリにコピーされます。

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_PAR   パラメータエラー (texrtn == NULL)*
```

例

```
#define ID_task1 1

void  texrtn(TEXPTN texptn, VP_INT exinf)
{
      :
}

const T_DTEX dtex = { TA_HLNG, (FP)texrtn };

TASK task1(void)
{
      :
    def_tex(ID_task1, &dtex);
      :
}
```

ras_tex, iras_tex

機　能　タスク例外処理要求

形　式
```
ER ras_tex(ID tskid, TEXPTN rasptn);
ER iras_tex(ID tslid, TEXPTN rasptn);
tskid   タスク ID
rasptn  タスク例外要因
```

解　説　tskid で指定されたタスクに対して rasptn で指定される例外処理を要求します。対象タスクが広義の待ち状態のときは例外要因は保留され例外処理は実行されません。対象タスクが実行状態になるまで実行されません。tskid = TSK_SELF で自タスクを対象タスクにします。

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   タスク例外処理ルーチン未定義
E_PAR   rasptn が 0
```

例
```
#define ID_task1 1

TASK task1(void)
{
      :
    ras_tex(ID_task1, 1);
      :
    ras_tex(ID_task1, 2);
      :
}
```

ｄｉｓ＿ｔｅｘ

機　能　タスク例外処理禁止

形　式

```
ER dis_tex(void);
```

解　説　タスクコンテキストでは自タスク、割込みハンドラでは実行状態タスクに対してタスク例外処理を禁止します。タイムイベントハンドラでは、E_CTX エラーになります。

戻　値

E_OK	正常終了
E_CTX	コンテキストエラー
E_OBJ	タスク例外処理ルーチンが未定義

例

```
TASK task1(void)
{
      :
    dis_tex();
      :
}
```

ena__tex

機　能　タスク例外処理許可

形　式　**ER ena_tex(void);**

解　説　タスクコンテキストでは自タスク、割込みハンドラでは実行状態タスクに対してタスク例外処理を許可します。タイムイベントハンドラでは、E_CTX エラーになります。

保留例外要因があれば指定タスクがRUNNING状態になった時に例外処理ルーチンが実行されます。

戻　値　**E_OK　正常終了**
E_CTX　コンテキストエラー
E_OBJ　タスク例外処理ルーチンが未定義

例

```
TASK task1(void)
{
      :
    ena_tex();
      :
}
```

ｓｎｓ＿ｔｅｘ

機　能　自タスクのタスク例外処理禁止状態の参照

形　式　**BOOL sns_tex(void);**

解　説　実行状態のタスクがタスク例外処理禁止状態であれば TRUE、許可状態であれば FALSE を返します。実行状態のタスクが無い場合には TRUE を返します。

戻　値　**TRUE　禁止中**
FALSE　許可中

例
```
TASK task1(void)
{
      :
    if (sns_tex())
    {
        :
    }
      :
}
```

ref_tex

機　能　タスク例外処理状態参照

形　式
```
ER ref_tex(ID tskid, T_RTEX *pk_rtex);
tskid     タスク ID
pk_rtex   タスク例外処理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タスク例外処理状態を、*pk_rtex に返します。

tskid = TSK_SELF で自タスクを指定できます。

タスク例外処理状態パケットの構造は次の通りです。

```
typedef struct t_rtex
{   STAT texstat;           例外処理の状態
    TEXPTN pndptn;          保留例外要因
} T_RTEX;
```

texstat には次の値が返されます。

```
TTEX_ENA        0x00 タスク例外処理許可状態
TTEX_DIS        0x01 タスク例外処理禁止状態
```

例外処理要求が無いときには pndptn = 0 となります。

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
E_OBJ   タスク例外処理ルーチンが未定義
E_OBJ   指定タスクが休止状態
```

例
```
#define ID_task2 2

TASK task1(void)
{
    T_RTEX rtex;
      :
    ref_tex(ID_task2, &rtex);
    if (rtex.pndptn != 0)
         :
}
```

５．４ 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。

定義情報パケットを ROM 以外に置いた場合、すなわち const を付けなかった場合、定義情報パケットはシステムメモリにコピーされます。

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
    B *name;                セマフォ名へのポインタ
} 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）
TA_TPRI         待ちタスク行列はタスク優先度順
```

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 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
正の値ならば、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 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_NOID　セマフォ ID が不足
E_CTX　割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;
      :
}
```

ｄｅｌ＿ｓｅｍ

機　能　セマフォ削除

形　式
```
ER del_sem(ID semid);
semid   セマフォ ID
```

解　説　semid で指定されたセマフォを削除します。すなわち、セマフォ管理ブロックをシステムメモリへ解放します。

このセマフォに対して待っているタスクがあった場合、このタスクの待ちを解除します。待ち解除されたタスクへは、削除されたことを示す E_DLT エラーが返ります。

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
E_CTX   割込みハンドラから発行 *
```

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

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、WAITING-SUSPENDED 状態だったら SUSPENDED 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.wtsk != FALSE)
      :
}
```

５．５ 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。

定義情報パケットを ROM 以外に置いた場合、すなわち const を付けなかった場合、定義情報パケットはシステムメモリにコピーされます。

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
} 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_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（FIFO）
TA_TPRI         待ちタスク行列はタスク優先度順
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
```

待ち行列につながれたタスクは、待ち行列につながれた順に待ち解除されるとは限りません。待っているフラグビットパーターンに合致したタスクから待ち解除されます。また、TA_CLR を指定しない場合、複数のタスクが同時に待ち解除されることもあります。TA_CLR を指定した場合、最初にタスクを待ち解除した時点でフラグがクリアされるため複数のタスクが同時に待ち解除されることはありません。

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_OBJ	イベントフラグが既に生成されている
E_CTX	割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_WMUL, 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　**正の値ならば、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_CTX　割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_WMUL, 0 };

TASK task1(void)
{
    ER_ID ercd;
      :
    ercd = acre_flg(&cflg1);
    if (ercd > 0)
        ID_flg1 = ercd;
      :
}
```

ｄｅｌ＿ｆｌｇ

機　能　イベントフラグ削除

形　式
```
ER del_flg(ID flgid);
flgid   イベントフラグ ID
```

解　説　flgid で指定されたイベントフラグを削除します。すなわち、イベントフラグ管理ブロックをシステムメモリへ解放します。

このイベントフラグに対して待っているタスクがあった場合、このタスクの待ちを解除します。待ち解除されたタスクへは、削除されたことを示す E_DLT エラーが返ります。

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
E_CTX   割込みハンドラから発行 *
```

例
```
#define ID_flg1 1

TASK task1(void)
{
       :
     del_flg(ID_flg1);
       :
}
```

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、WAITING-SUSPENDED 状態だったら SUSPENDED 状態へ遷移）。

イベントフラグ生成時に TA_CLR を指定した場合で、待ち解除されたタスクがある場合は、最初の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。

イベントフラグでの複数タスクの待ちを許していて TA_CLR を指定しない場合、１回の set_flgで複数のタスクが一斉に待ち解除となることがあります。wai_flgにおけるwaiptnや wfmode、生成情報の TA_CLR の有無との関係により、必ずしも行列先頭のタスクから待ち解除になるとは限りません。また、待ち行列中にクリア指定のタスクがあってこれが待ち解除される場合、このタスクより後ろに並んでいるタスクは、クリアされたイベントフラグを見ることになるので、待ち解除されません。

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)
{
      :
    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, UINT *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）**
E_DLT　**待ちの間にイベントフラグが削除された**

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（複数待ち許さない場合）
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_DLT　待ちの間にイベントフラグが削除された
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_AND|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);
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
flgid   イベントフラグ ID
```

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(&rflg, ID_flg1);
    if (rflg.flgptn != 0)
      :
}
```

５．６ 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。

定義情報パケットを ROM 以外に置いた場合、すなわち const を付けなかった場合、定義情報パケットはシステムメモリにコピーされます。

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
    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
} T_CDTQ;
```

データキュー属性 dtqatr には次の値を入れてください。

TA_TFIFO	送信待ちタスク行列は先着順（FIFO）
TA_TPRI	送信待ちタスク行列はタスク優先度順

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
_CTX　割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_TPRI, 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　**正の値ならば、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_TPRI, 30, NULL };

TASK task1(void)
{
    ER_ID ercd;
      :
    ercd = acre_dtq(&cdtq1);
    if (ercd > 0)
        ID_dtq1 = ercd;
      :
}
```

del_dtq

機　能　データキュー削除

形　式
```
ER del_dtq(ID dtqid);
dtqid   データキュー ID
```

解　説　dtqid で指定されたデータキューを削除します。すなわち、データキュー管理ブロックをシステムメモリへ解放します。データバッファを OS が確保した場合はデータバッファも開放されます。バッファ内のデータは破棄されます。

このデータキューに対して待っているタスクがあった場合、このタスクの待ちを解除します。待ち解除されたタスクへは、削除されたことを示す E_DLT エラーが返ります。

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
E_CTX   割込みハンドラから発行 *
```

例
```
#define ID_dtq1 1

TASK task1(void)
{
      :
    del_dtq(ID_dtq1);
      :
}
```

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、WAITING-SUSPENDED 状態だったら SUSPENDED 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_DLT   待ちの間にデータキューが削除された
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、WAITING-SUSPENDED 状態だったら SUSPENDED 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、WAITING-SUSPENDED 状態だったら SUSPENDED 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_DLT　待ちの間にデータキューが削除された
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

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

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、WAITING-SUSPENDED 状態だったら SUSPENDED 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）
E_DLT   待ちの間にデータキューが削除された
```

補　足　trcv_dtq(dtqid, p_data, TMO_FEVR) 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_DLT   待ちの間にデータキューが削除された
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)
       :
}
```

５．７ 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
    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先頭アドレス
    B *name;                メールボックス名へのポインタ
} T_CMBX;
```

メールボックス属性 mbxatr には次の値を入れてください。

```
TA_TFIFO        受信待ちタスク行列は先着順（FIFO）
TA_TPRI         受信待ちタスク行列はタスク優先度順
TA_MFIFO        メッセージのキューイングは先着順（FIFO）
TA_MPRI         メッセージのキューイングはメッセージ優先度順
```

mbxatr に TA_MPRI が指定された場合にはメッセージ優先度別のメッセージ待ち行列を作ります。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。

maxmpri には、メッセージ優先度の最大値を設定してください。大きな値を指定するとメモリ消費量が多くなるので注意してください。メッセージ優先度はタスク優先度と同様に 1 が最優先で、値が大きくなるほど優先度が低くなります。

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
正の値ならば、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;
}
```

ｄｅｌ＿ｍｂｘ

機　能　メールボックス削除

形　式
```
ER del_mbx(ID mbxid);
```
mbxid メールボックス ID

解　説　mbxid で指定されたメールボックスを削除します。すなわち、メールボックス管理ブロック等の生成時に確保したメモリをシステムメモリへ解放します。

このメールボックスに対して、メッセージ受信を待っているタスクがあった場合、このタスクの待ちを解除します。待ち解除されたタスクへは、削除されたことを示す E_DLT エラーが返ります。

キューイングされたメッセージがあると、それは失われます。メモリプールからメッセージを動的に確保していた場合にはメールボックス削除の前に、prcv_mbx でメッセージを読み出して、適切なメモリプールへの返却をしてください。ユーザプログラムが確保した領域を自動的に OS が開放することはできないので、いわゆるメモリリークが発生します。

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_CTX   割込みハンドラから発行 *
```

例
```
#define ID_mbx1 1

TASK task1(void)
{
      :
    del_mbx(ID_mbx1);
      :
}
```

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、WAITING-SUSPENDED 状態だったら SUSPENDED 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。メッセージ優先度を使う場合は、next に続けて INT msgpri; を設けてください（次々ページ例 2 参照）。

既にキューイングされているメッセージを再度 snd_mbx した場合も 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)。メールボックス生成時に、メッセージ優先度順のキューイングを指定しない場合、msgpri メンバーは省略できます。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(&msg, ID_mpf);  /* 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）*/
    INT msgpri;               /* メッセージ優先度（使わない場合は定義不要）*/
    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->msgpri = 1;        /* メッセージ優先度（使わない場合は設定不要）*/
    msg->fncd = 2;
    msg->data = 3;
    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）
E_DLT　待ちの間にメールボックスが削除された

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);
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
mbxid メールボックス ID
```

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, 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_DLT　待ちの間にメールボックスが削除された
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)
         :
}
```

５．８ 拡張同期・通信機能（ミューテックス）

ｃｒｅ＿ｍｔｘ

機　能　ミューテックス生成

形　式
```
ER cre_mtx(ID mtxid, const T_CMTX *pk_cmtx);
mtxid    ミューテックス ID
pk_cmtx ミューテックス生成情報パケットへのポインタ
```

解　説　mtxid で指定されたミューテックスを生成します。すなわち、システムメモリから、ミューテックス管理ブロックを動的に割り当てます。

定義情報パケットを ROM 以外に置いた場合、すなわち const を付けなかった場合、定義情報パケットはシステムメモリにコピーされます。

ミューテックス生成情報パケットの構造は次の通りです。

```
typedef struct t_cmtx
{    ATR mtxatr;            ミューテックス属性
     PRI ceilpri;           優先度上限プロトコルで使用する上限優先度
     B *name;               ミューテックス名へのポインタ
} T_CMTX;
```

ミューテックス属性 mtxatr 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）
TA_TPRI         待ちタスク行列はタスク優先度順
TA_INHERIT      優先度継承プロトコルを使用
TA_CEILING      優先度上限プロトコルを使用
```

TA_INHERIT, TA_CEILING いずれかを指定しない場合ミューテックスは基本的にバイナリセマフォと同一の機能を提供します。ただし、ミューテックスの場合タスクがロックしたまま終了した場合自動的にアンロックされます。

TA_INHERIT を指定した場合、優先度継承プロトコルを使ってタスクの現在優先度を操作して優先度逆転を防ぎます。ミューテックスをロック中に、優先度の高いタスクがそのミューテックスをロックしようとして WAITING 状態になると、ロック中のタスクの優先度が待ち行列にあるタスクのうちもっとも優先度の高いタスクの優先度と同一になります。このようにすることで中間の優先度を持つタスクがミューテックスをロック中のタスクをプリエンプトして間接的にそのミューテックスをロック待ちしているより優先度の高いタスクをブロックすることを防ぎます。

TA_CEILING を指定した場合、優先度上限プロトコルを使ってタスクの現在優先度を操作します。優先度上限プロトコルでは、生成情報で指定された ceilpri を使用します。タスクが TA_CEILING 指定されたミューテックスをロックするとそのタスクの現在優先度が ceilpri で指定した値になります。ceilpri にそのミューテックスを共有するタスクの中で最高の優先度を持つタスクの優先度値を設定することで、優先度継承プロトコルと同様の効果を得ることができます。

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	**ミューテックス ID が範囲外 ***
E_OBJ	**ミューテックスが既に生成されている**
E_CTX	**割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_mtx1 1
const T_CMTX cmtx1 = { TA_INHERIT, 0 };

TASK task1(void)
{
    ER ercd;
       :
    ercd = cre_mtx(ID_mtx1, &cmtx1);
       :
}
```

ａｃｒｅ＿ｍｔｘ

機　能　ミューテックス生成（ID 自動割り当て）

形　式
```
ER_ID acre_mtx(const T_CMTX *pk_cmtx);
```
pk_cmtx ミューテックス生成情報パケットへのポインタ

解　説　未生成ミューテックスの ID を、大きな方から検索して割り当てます。ミューテックス 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_mtx と同じです。

戻　値
正の値ならば、割り当てられたミューテックス ID
E_NOID　ミューテックス ID が不足
E_CTX　割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_mtx1;
const T_CMTX cmtx1 = { TA_TFIFO, 0 };

TASK task1(void)
{
    ER_ID ercd;
      :
    ercd = acre_mtx(&cmtx1);
    if (ercd > 0)
        ID_mtx1 = ercd;
      :
}
```

ｄｅｌ＿ｍｔｘ

機　能　ミューテックス削除

形　式
```
ER del_mtx(ID mtxid);
mtxid   ミューテックス ID
```

解　説　mtxid で指定されたミューテックスを削除します。すなわち、ミューテックス管理ブロックをシステムメモリへ解放します。

このミューテックスに対して待っているタスクがあった場合、タスクの待ちを解除します。待ち解除されたタスクへは、削除されたことを示す E_DLT エラーが返ります。

戻　値
```
E_OK    正常終了
E_ID    ミューテックス ID が範囲外 *
E_NOEXS ミューテックスが生成されていない
E_CTX   割込みハンドラから発行 *
```

例
```
#define ID_mtx1 1

TASK task1(void)
{
      :
    del_mtx(ID_mtx1);
      :
}
```

unl_mtx

機　能　ミューテックスロック解除

形　式
```
ER unl_mtx(ID mtxid);
```
mtxid ミューテックス ID

解　説　mtxid で指定されたミューテックスをロック解除する。

このミューテ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、WAITING-SUSPENDED 状態だったら SUSPENDED 状態へ遷移)。そして、ミューテックスをロック状態にします。

ロック待ちしているタスクが無い場合ロックを解除します。

自タスクがロックしていないミューテックスをロック解除することはできません。

戻　値
```
E_OK     正常終了
E_ID     ミューテックス ID が範囲外 *
E_NOEXS ミューテックスが生成されていない
E_ILUSE 対象ミューテックスをロックしていない
```

loc_mtx

機　能　ミューテックス資源獲得

形　式
```
ER loc_mtx(ID mtxid);
mtxid   ミューテックス ID
```

解　説　mtxid で指定されたミューテックスがロックされていない場合はロック状態にします。対象ミューテックスが既にロックされている場合は自タスクを待ち行列につなぎロック待ち状態にします。

自タスクがすでに対象ミューテックスをロックしている場合は、すなわち多重ロックしようとすると E_ILUSE エラーを返します。また、TA_CEILING 指定されたミューテックスを上限優先度より高いベース優先度を持ったタスクがロックしようとした場合もE_ILUSEエラーを返します。

戻　値
```
E_OK    正常終了
E_ID    ミューテックス ID が範囲外 *
E_NOEXS ミューテ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_DLT   待ちの間にミューテックスが削除された
E_ILUSE 多重ロック、上限優先度違反
```

補　足　tloc_mtx(mtxid, TMO_FEVR) と同じです。

例
```
#define ID_mtx1 1

TASK task1(void)
{
      :
    loc_mtx(ID_mtx1);
      :
    unl_mtx(ID_mtx1);
      :
}
```

ploc_mtx

機　能　ミューテックス資源獲得（ポーリング）

形　式
```
ER ploc_mtx(ID mtxid);
mtxid ミューテックス ID
```

解　説　mtxid で指定されたミューテックスがロックされていない場合はロック状態にします。対象ミューテックスが既にロックされている場合は E_TMOUT エラーで返ります。その他は、loc_mtx と同様です。

戻　値
```
E_OK     正常終了
E_ID     ミューテックス ID が範囲外 *
E_NOEXS ミューテックスが生成されていない
E_ILUSE 多重ロック、上限優先度違反
E_TMOUT ポーリング失敗
```

補　足　tloc_mtx(mtxid, TMO_POL) と同じです。

例
```
if (ploc_mtx(ID_mtx1) == E_OK)
{
      :
    unl_mtx(ID_mtx1);
      :
}
```

ｔｌｏｃ＿ｍｔｘ

機　能　ミューテックス資源獲得（タイムアウト有）

形　式
```
ER tloc_mtx(ID mtxid, TMO tmout);
mtxid   ミューテックス ID
tmout   タイムアウト値
```

解　説　mtxid で指定されたミューテックスがロックされていない場合はロック状態にします。対象ミューテックスが既にロックされている場合は自タスクを待ち行列につなぎロック待ち状態にします。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。その他は、loc_mtx と同様です。

tmout = TMO_POL (= 0) により待ちをおこなわない、すなわち pol_mtx 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 loc_mtx と同じ動作になります。

戻　値
```
E_OK     正常終了
E_ID     ミューテックス ID が範囲外 *
E_NOEXS ミューテ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_loc を受け付け）
E_DLT    待ちの間にミューテックスが削除された
E_ILUSE 多重ロック、上限優先度違反
E_TMOUT タイムアウト
```

例
```
#define ID_mtx1 1

TASK task1(void)
{
    ER ercd;
      :
    ercd = tloc_mtx(ID_mtx1, 100/MSEC);
    if (ercd == E_OK)
      :
}
```

ｒｅｆ＿ｍｔｘ

機　能　ミューテックス状態参照

形　式
```
ER ref_mtx(ID mtxid, T_RMTX *pk_rmtx);
mtxid   ミューテックス ID
pk_rmtx ミューテックス状態パケットを格納する場所へのポインタ
```

解　説　mtxid で指定されたミューテックスの状態を、*pk_rmtx に返します。

ミューテックス状態パケットの構造は次の通りです。

```
typedef struct t_rmtx
{   ID htskid;          ロックしているタスクのタスク ID または TSK_NONE
    ID wtskid;          ロック待ちしているタスクのタスク ID または TSK_NONE
} T_RMTX;
```

htskid には、対象ミューテックスをロックしているタスクがあればそのタスク ID 番号が返ります。無い場合には、TSK_NONE が返ります。

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    ミューテックス ID が範囲外
E_NOEXS ミューテックスが生成されていない
```

例
```
#define ID_mtx1 1

TASK task1(void)
{
    T_RMTX rmtx;
      :
    ref_mtx(ID_mtx1, &rmtx);
      :
}
```

５．９ 拡張同期・通信機能（メッセージバッファ）

ｃｒｅ＿ｍｂｆ

機　能　メッセージバッファ生成

形　式
```
ER cre_mbf(ID mbfid, const T_CMBF *pk_cmbf);
mbfid    メッセージバッファ ID
pk_cmbf 生成情報パケットへのポインタ
```

解　説　mbfid で指定されたメッセージバッファを生成します。すなわち、システムメモリからメッセージバッファ管理ブロックを動的に割り当てます。

生成情報パケットを ROM 以外に置いた場合、すなわち const を付けなかった場合、生成情報パケットはシステムメモリにコピーされます。

メッセージバッファ生成情報パケットの構造は次の通りです。

```
typedef struct t_cmbf
{    ATR mbfatr;          メッセージバッファ属性
     UINT maxmsz;         メッセージの最大長（バイト数）
     SIZE mbfsz;          リングバッファの総サイズ（バイト数）
     VP mbf;              リングバッファのアドレス
     B *name;             メッセージバッファ名へのポインタ
} T_CMBF;
```

メッセージバッファ属性 mbfatr には次の値を入れてください。

```
TA_TFIFO      送信待ちタスク行列は先着順（FIFO）
TA_TPRI       送信待ちタスク行列はタスク優先度順
TA_TPRIR      受信待ちタスク行列はタスク優先度順
```

mbfatr に TA_TPRIR を指定しない場合受信待ちタスク行列は先着順（FIFO）になります。

リングバッファ領域をユーザプログラムで確保した場合には、その先頭アドレスを mbf に設定してください。この場合 OS がメッセージを管理するためバッファの一部を使用するので、全てをユーザープログラムで使用することは出来ません。msgsz バイト（msgsz > 1) のメッセージを msgcnt 個格納するために確保すべきサイズは

```
TSZ_MBF(msgcnt, msgsz)
```

マクロによって、取得できます。ただし、メッセージサイズを１バイト（msgsz = 1）とした場合は msgsz バイトの領域を必要とします。すなわち OS によるオーバーヘッドはありません。

mbf が NULL の場合、メモリプール用メモリから動的に mbufsz で指定されたサイズだけリングバッファ領域を確保します。

mbfsz に 0 を設定することも可能です。この場合リングバッファは必要ありません。この時は、タスク間で同期を取って直接データを渡すようになります。

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	**メッセージバッファ ID が範囲外 ***
E_OBJ	**メッセージバッファが既に生成されている**
E_PAR	**パラメータエラー (maxmsz = 0) ***
E_CTX	**割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	**リングバッファ用のメモリが確保できない ****

例

```
#define ID_mbf1 1
const T_CMBF cmbf1 = { TA_TFIFO, 32, 512, NULL };

TASK task1(void)
{
    ER ercd;
      :
    ercd = cre_mbf(ID_mbf1, &cmbf1);
      :
}
```

ａｃｒｅ＿ｍｂｆ

機　能　メッセージバッファ生成（ID 自動割り当て）

形　式
```
ER_ID acre_mbf(const T_CMBF *pk_cmbf);
```
pk_cmbf メッセージバッファ生成情報パケットへのポインタ

解　説　未生成メッセージバッファの ID を、大きな方から検索して割り当てます。メッセージバッファ 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_mbf と同じです。

戻　値
```
正の値ならば、割り当てられたメッセージバッファ ID
E_NOID   メッセージバッファ ID が不足
E_PAR    パラメータエラー (maxmsz = 0) *
E_CTX    割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 リングバッファ用のメモリが確保できない **
```

例
```
ID ID_mbf1;
const T_CMBF cmbf1 = { TA_TFIFO, 32, 512, NULL };

TASK task1(void)
{
    ER_ID ercd;
      :
    ercd = acre_mbf(&cmbf1);
    if (ercd > 0)
         ID_mbf1 = ercd;
}
```

ｄｅｌ＿ｍｂｆ

機　能　メッセージバッファ削除

形　式
```
ER del_mbf(ID mbfid);
mbfid   メッセージバッファ ID
```

解　説　mbfid で指定されたメッセージバッファを削除します。すなわち、メッセージバッファ管理ブロックをシステムメモリへ解放し、OS が確保した場合はリングバッファ領域をメモリプール用メモリへ解放します。

このメッセージバッファに対して、メッセージ送信やメッセージ受信を待っているタスクがあった場合、このタスクの待ちを解除します。待ち解除されたタスクへは、削除されたことを示す E_DLT エラーが返ります。

戻　値
```
E_OK    正常終了
E_ID    メッセージバッファ ID が範囲外 *
E_NOEXS メッセージバッファが生成されていない
E_CTX   割込みハンドラから発行 *
```

例
```
#define ID_mbf1 1

TASK task1(void)
{
      :
    del_mbf(ID_mbf1);
      :
}
```

ｓｎｄ＿ｍｂｆ

機　能　メッセージバッファへ送信

形　式　**ER snd_mbf(ID mbfid, VP msg, UINT msgsz);**
mbfid　メッセージバッファ ID
msg　　送信メッセージへのポインタ
msgsz　送信メッセージのサイズ（バイト数）

解　説　mbfid で指定されたメッセージバッファを使って、msg と msgsz で示されるメッセージを送信します。

このメッセージバッファで受信を待っているタスクがある場合、受信待ち行列の先頭タスクの受信バッファへメッセージをコピーし、そのタスクの待ちを解除します。このメッセージバッファで受信を待っているタスクがない場合、メッセージをメッセージバッファが使用するリングバッファへコピーします。ただし、リングバッファに空きがなかった場合は、このシステムコールを発行したタスクの方が、送信待ち状態となります。

snd_mbf, psnd_mbf, tsnd_mbf で、サイズが msgsz のメッセージをキューイングするためには、リングバッファに、msgsz + 2 バイト (メッセージサイズを示すヘッダの分) の空きが必要です。ただし、メッセージバッファ生成時に指定したメッセージの最大長 maxmsz が１バイトの場合に限って、+ 2 バイトのヘッダ領域が不要となります。

戻　値　**E_OK　　正常終了**
E_PAR　メッセージのサイズが範囲外（msgsz = 0, msgsz > 生成情報の maxmsz）*
E_ID　　メッセージバッファ ID が範囲外 *
E_NOEXS メッセージバッファ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 待ち状態強制解除された
E_DLT　待ちの間にメッセージバッファが削除された

補　足　tsnd_mbf(mbfid, msg, msgsz, TMO_FEVR) と同じです。

例

```
#define ID_mbf1 1

TASK task1(void)
{
    H cmd = 0x0012;
      :
    snd_mbf(ID_mbf1, (VP)&cmd, sizeof cmd);
      :
}
```

ｐｓｎｄ＿ｍｂｆ

機　能　メッセージバッファへ送信（ポーリング）

形　式
```
ER psnd_mbf(ID mbfid, VP msg, UINT msgsz);
mbfid   メッセージバッファ ID
msg     送信メッセージへのポインタ
msgsz   送信メッセージのサイズ（バイト数）
```

解　説　mbfid で指定されたメッセージバッファを使って、msg と msgsz で示されるメッセージを送信します。

このメッセージバッファで受信を待っているタスクがある場合、受信待ち行列の先頭タスクの受信バッファへメッセージをコピーし、そのタスクの待ちを解除します。このメッセージバッファで受信を待っているタスクがない場合、メッセージをメッセージバッファ内部のリングバッファへコピーします。リングバッファに空きがなかった場合は、待ち状態に入らずに、E_TMOUT を返します。

戻　値
```
E_OK    正常終了
E_PAR   メッセージのサイズが範囲外（msgsz = 0, msgsz > 生成情報の maxmsz）*
E_ID    メッセージバッファ ID が範囲外 *
E_NOEXS メッセージバッファが生成されていない
E_TMOUT ポーリング失敗
```

補　足　tsnd_mbf(mbfid, msg, msgsz, TMO_POL) と同じです。

例
```
#define ID_mbf2 2

TASK task1(void)
{
    B msg[16] ;
      :
    strcpy(msg, "Hello");
    if (psnd_mbf(ID_mbf2, (VP)msg, strlen(msg)) != E_OK)
      :
}
```

ｔｓｎｄ＿ｍｂｆ

機　能　メッセージバッファへ送信（タイムアウト有）

形　式
```
ER tsnd_mbf(ID mbfid, VP msg, UINT msgsz, TMO tmout);
```
mbfid　メッセージバッファ ID
msg　　送信メッセージへのポインタ
msgsz　送信メッセージのサイズ（バイト数）
tmout　タイムアウト値

解　説　mbfid で指定されたメッセージバッファを使って、msg と msgsz で示されるメッセージを送信します。

このメッセージバッファで受信を待っているタスクがある場合、受信待ち行列の先頭タスクの受信バッファへメッセージをコピーし、そのタスクの待ちを解除します。このメッセージバッファで受信を待っているタスクがない場合、メッセージをメッセージバッファ内部のリングバッファへコピーします。ただし、リングバッファに空きがなかった場合は、このシステムコールを発行したタスクの方が、送信待ち状態となります。

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_mbf 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_mbf と同じ動作になります。

戻　値
E_OK　　正常終了
E_PAR　　メッセージのサイズが範囲外（msgsz = 0, msgsz > 生成情報の maxmsz）*
E_ID　　メッセージバッファ ID が範囲外*
E_NOEXS メッセージバッファ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_DLT　　待ちの間にメッセージバッファが削除された
E_TMOUT タイムアウト

例

```
#define ID_mbf2 2

TASK task1(void)
{
    B *res = "Hello";
    ER ercd;
      :
    ercd = tsnd_mbf(ID_mbf2, (VP)res, 5, 1000/MSEC);
    if (ercd = E_TMOUT)
      :
}
```

ｒｃｖ＿ｍｂｆ

機　能　メッセージバッファから受信

形　式
```
ER_UINT = rcv_mbf(ID mbfid, VP msg);
msg 受信メッセージを格納する場所へのポインタ
mbfid メッセージバッファ ID
```

解　説　mbfid で指定されたメッセージバッファを使ってメッセージを受信します。受信したメッセージは、msg へコピーされます。受信したメッセージのサイズは、関数値として返されます。

msg で指し示される領域は、メッセージバッファ生成時に指定したメッセージの最大長 maxmsz 以上としてください。

メッセージバッファにまだメッセージが到着していない場合、本システムコールの発行タスクは、そのメッセージバッファの受信待ち行列につながれます。

戻　値
```
正の値   受信メッセージバイト数
E_ID     メッセージバッファ ID が範囲外 *
E_NOEXS メッセージバッファ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_DLT    待ちの間にメッセージバッファが削除された
```

補　足　trcv_mbf(mbfid, msg, TMO_FEVR) と同じです。

例
```
#define ID_mbf1 1

TASK task1(void)
{
    H cmd;
    ER dummy;
      :
    dummy = rcv_mbf(ID_mbf1, (VP)&cmd);
      :

}
```

ｐｒｃｖ＿ｍｂｆ

機　能　メッセージバッファから受信（ポーリング）

形　式
```
ER_UINT = prcv_mbf(ID mbfid, VP msg);
mbfid   メッセージバッファ ID
msg     受信メッセージを格納する場所へのポインタ
```

解　説　mbfid で指定されたメッセージバッファを使ってメッセージを受信します。受信したメッセージは、msg へコピーされます。受信したメッセージのサイズは、関数値として返されます。

msg で指し示される領域は、メッセージバッファ生成時に指定したメッセージの最大長 maxmsz 以上としてください。

メッセージバッファにまだメッセージが到着していない場合、待ち状態に入らずに、E_TMOUT エラーを返します。

戻　値
```
正の値   受信メッセージバイト数
E_ID     メッセージバッファ ID が範囲外 *
E_NOEXS メッセージバッファが生成されていない
E_TMOUT ポーリング失敗
```

補　足　trcv_mbf(msg, p_msgsz, mbfid, TMO_POL) と同じです。

例
```
#define ID_mbf2 2

TASK task1(void)
{
    B buf[16] ;
    ER size;
       :
    if (size = prcv_mbf(ID_mbf2, (VP)buf) > 0)
       :

}
```

ｔｒｃｖ＿ｍｂｆ

機　能　メッセージバッファから受信（タイムアウト有）

形　式
```
ER_UINT = trcv_mbf(ID mbfid, VP msg, TMO tmout);
```
mbfid　メッセージバッファ ID
msg　　受信メッセージを格納する場所へのポインタ
tmout　タイムアウト値

解　説　mbfid で指定されたメッセージバッファを使ってメッセージを受信します。

受信したメッセージは、msg へコピーされます。受信したメッセージのサイズは、関数値として返されます。msg で指し示される領域は、メッセージバッファ生成時に指定したメッセージの最大長 maxmsz 以上としてください。

メッセージバッファにまだメッセージが到着していない場合、本システムコールの発行タスクは、そのメッセージバッファの受信待ち行列につながれます。

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 prcv_mbf 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_mbf と同じ動作になります。

戻　値　**正の値　受信メッセージバイト数**
E_ID　　メッセージバッファ ID が範囲外 *
E_NOEXS メッセージバッファ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_DLT　待ちの間にメッセージバッファが削除された
E_TMOUT タイムアウト

例

```
#define ID_mbf2 2

TASK task1(void)
{
    B buf[16] ;
    ER_UINT size;
      :
    size = trcv_mbf(ID_mbf2, (VP)buf, 1000/MSEC)
    if (ercd == E_TMOUT)
      :
}
```

ｒｅｆ＿ｍｂｆ

機　能　メッセージバッファ状態参照

形　式
```
ER ref_mbf(ID mbfid, T_RMBF *pk_rmbf);
mbfid メッセージバッファ ID
pk_rmbf メッセージバッファ状態パケットを格納する場所へのポインタ
```

解　説　mbfid で指定されたメッセージバッファの状態を、*pk_rmbf に返します。

メッセージバッファ状態パケットの構造は次の通りです。

```
typedef struct t_rmbf
{   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 smsgcnt;           メッセージバッファに入っているメッセージ数
    SIZE fmbfsz;            リングバッファの空きサイズ（バイト数）
} T_RMBF;
```

stskid と rtskid 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    メッセージバッファ ID が範囲外
E_NOEXS メッセージバッファが生成されていない
```

例
```
#define ID_mbf1 1

TASK task1(void)
{
    T_RMBF rmbf;
      :
    ref_mbf(ID_mbf1, &rmbf);
    if (rmbf.fmbufsz >= 32 + sizeof(int))
      :
}
```

５．１０ 拡張同期・通信機能（ランデブ用ポート）

ｃｒｅ＿ｐｏｒ

機　能　ランデブ用ポート生成

形　式
```
ER cre_por(ID porid, const T_CPOR *pk_cpor);
porid    ランデブ用ポート ID
pk_cpor ランデブ用ポート生成情報パケットへのポインタ
```

解　説　porid で指定されたランデブ用ポートを生成します。すなわち、システムメモリから、ランデブ用ポート管理ブロックを動的に割り当てます。

定義情報パケットを ROM 以外に置いた場合、すなわち const を付けなかった場合、定義情報パケットはシステムメモリにコピーされます。

ランデブ用ポート生成情報パケットの構造は次の通りです。

```
typedef struct t_cpor
{   ATR poratr;          ランデブ用ポート属性
    UINT maxcmsz;        呼出メッセージの最大長（バイト数）
    UINT maxrmsz;        返答メッセージの最大長（バイト数）
    B *name;             ポート名へのポインタ
} T_CPOR;
```

ランデブ用ポート属性 poratr には以下の値を設定してください。

```
TA_TFIFO        呼出待ち行列を先着順 (FIFO) とする
TA_TPRI         呼出待ち行列をタスク優先度順とする
```

ランデブ受付の待ち行列は FIFO のみです。ランデブでは、呼出側と受付側がそろった時点で、メッセージのコピーがおこなわれますので、メッセージをキューイングするためのリングバッファ等は存在しません。

maxcmsz と maxrmsz には 0 を設定することもできます。

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    ランデブ用ポート ID が範囲外 *
E_OBJ   ランデブ用ポートが既に生成されている
E_CTX   割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_por1 1
const T_CPOR cpor1 = { TA_TFIFO, 64, 32 };

TASK task1(void)
{
    ER ercd;
      :
    ercd = cre_por(ID_por1, &cpor1);
      :
}
```

ａｃｒｅ＿ｐｏｒ

機　能　ランデブ用のポート生成（ID 自動割り当て）

形　式
```
ER_ID acre_por(const T_CPOR *pk_cpor);
```
pk_cpor ランデブ用ポート生成情報パケットへのポインタ

解　説　未生成ランデブ用ポートの ID を、大きな方から検索して割り当てます。ランデブ用ポート 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_por と同じです。

戻　値
正の値ならば、割り当てられたランデブ用ポート ID
E_NOID　ランデブ用ポート ID が不足
E_CTX　割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_por1;
const T_CPOR cpor1 = { TA_TFIFO, 64, 32 };

TASK task1(void)
{
    ER_ID ercd;
      :
    ercd = acre_por(&cpor1);
    if (ercd > 0)
         ID_por1 = ercd;
      :
}
```

ｄｅｌ＿ｐｏｒ

機　能　ランデブ用のポート削除

形　式
```
ER del_por(ID porid);
```
porid ランデブ用ポート ID

解　説　porid で指定されたランデブ用のポートを削除します。すなわち、ランデブ用ポート管理ブロックをシステムメモリへ解放します。

このランデブ用ポートに対して、ランデブ受付やランデブ呼出を待っているタスクがあった場合、このタスクの待ちを解除します。待ち解除されたタスクへは、削除されたことを示す E_DLT エラーが返ります。

ランデブポートを削除してもすでに成立したランデブには影響ありません。

戻　値
```
E_OK     正常終了
E_ID     ランデブ用ポート ID が範囲外 *
E_NOEXS ランデブ用ポートが生成されていない
E_CTX    割込みハンドラから発行 *
```

例
```
#define ID_por1 1
TASK task1(void)
{
      :
    del_por(ID_por1);
      :
}
```

cal_por

機　能　ポートに対するランデブの呼出

形　式
```
ER_UINT = cal_por(ID porid, RDVPTN calptn, VP msg, UINT cmsgsz);
```
porid　ランデブ用ポート ID
calptn　呼出側選択条件を表すビットパターン
msg　呼出メッセージへのポインタ、かつ、返答メッセージ格納場所へのポインタ
cmsgsz　呼出メッセージのサイズ（バイト数）

解　説　porid で指定されたランデブ用のポートを使い、受付側タスクと待ち合わせをおこなった上で、受付側タスクへ呼出メッセージを渡します。さらに待ちをおこなって、受付側タスクから、返答メッセージを受け取ります。

calptn のビットパターンで、呼出側－受付側の組合せを選択することができます。この呼出側 cal_por システムコールの calptn と、受付側 acp_por システムコールの acpptn との論理積（calptn & acpptn）が 0 でない場合に、ランデブ成立となります。

このポートでランデブ受付待ちのタスクがある場合、受付待ちタスクとランデブが成立するか調べます（受付待ちタスクが複数ある場合は、受付待ち行列の先頭タスクから順に、ランデブ成立まで）。ランデブ受付待ちタスクがない場合やどの受付側タスクともランデブが成立しない場合、このシステムコールを発行した呼出側タスクは、ランデブ呼出待ちとして待ち行列につながれます。

ランデブが成立したなら、呼出メッセージを受付側タスクのバッファへコピーし、そのタスクの受付待ちを解除します。そしてこのシステムコールを発行した呼出側タスクは、ランデブ終了待ち状態になります。ランデブ終了待ち中は、タスクがポートから切り離されますので、待ち行列は作りません。

さらに、受付側タスクが rpl_rdv システムコールにより返答を返すと、その返答メッセージを受け取ってランデブを終了します。返答メッセージは、msg へコピーされます。

返答メッセージのサイズは、関数値として返されます。

msg で指し示される領域は、ランデブ用ポート生成時に指定した返答メッセージの最大長 maxrmsz 以上としてください。

戻　値　0 または正の値は返答メッセージサイズ
E_PAR　呼出側選択条件を表すビットパターン calptn が 0*
　　メッセージサイズが範囲外（cmsgsz = 0, cmsgsz > maxcmsz）*
E_ID　ランデブ用ポート ID が範囲外 *
E_NOEXS ランデブ用ポート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）
E_DLT　待ちの間にランデブ用ポートが削除された

補　足　tcal_por(porid, calptn, msg, cmsgsz, TMO_FEVR) と同じです。

例

```
#define ID_por1 1

TASK task1(void)
{
    B msg[16] ;
    ER_UINT size;
      :
    strcpy(msg, "Hello");
    size = cal_por(ID_por1, 0x0001, (VP)msg, strlen(msg));
    if (size >= 0)
         :
}
```

tcal_por

機　能　ポートに対するランデブの呼出（タイムアウト有）

形　式
```
ER_UINT = tcal_por(ID porid, RDVPTN calptn, VP msg, UINT cmsgsz, TMO tmout);
```
porid　ランデブ用ポート ID
calptn　呼出側選択条件を表すビットパターン
msg　返答メッセージを格納する場所へのポインタ
cmsgsz　呼出メッセージのサイズ（バイト数）
tmout　タイムアウト値

解　説　cal_por との違いは次の通りです。

ランデブが終了しないまま、このシステムコール発行から tmout で指定した時間が経過すると、タイムアウトエラー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指定は E_PAR エラーで返ります。tmout = TMO_FEVR (= -1) によりタイムアウトしない指定は cal_por と同じ動作になります。

戻　値
０または正の値は返答メッセージサイズ
E_PAR　呼出側選択条件を表すビットパターン calptn が 0*
メッセージサイズが範囲外（cmsgsz = 0, cmsgsz > maxcmsz）*
ポーリング指定 *
E_ID　ランデブ用ポート ID が範囲外 *
E_NOEXS　ランデブ用ポート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）
E_DLT　待ちの間にランデブ用ポートが削除された
E_TMOUT　タイムアウト

ａｃｐ＿ｐｏｒ

機　能　ポートに対するランデブ受付

形　式
```
ER_UINT = acp_por(ID porid, RDVPTN acpptn, RDVNO *p_rdvno, VP msg);
porid    ランデブ用ポート ID
acpptn   受付側選択条件を表すビットパターン
p_rdvno ランデブ番号を格納する場所へのポインタ
msg      呼出メッセージを格納する場所へのポインタ
```

解　説　porid で指定されたランデブ用ポートを使い、呼出側タスクと待ち合わせをおこなった上で、呼出メッセージを受け取ります。

acpptn のビットパターンで、呼出側－受付側の組合せを選択することができます。呼出側 cal_por システムコールの calptn と、この受付側 acp_por システムコールの acpptn との論理積（calptn & acpptn）が 0 でない場合に、ランデブ成立となります。

このポートでランデブ呼出待ちのタスクがある場合、呼出待ちタスクとランデブが成立するか調べます（呼出待ちタスクが複数ある場合は、呼出待ち行列の先頭タスクから順に、ランデブ成立まで）。ランデブ呼出待ちタスクがない場合や、どの呼出側タスクともランデブが成立しない場合、このシステムコールを発行した受付側タスクは、ランデブ受付待ちとして待ち行列につながれます。

ランデブが成立したら、呼出メッセージを受け取り、呼出側タスクを、呼出待ち状態からランデブ終了待ち状態にします。呼出メッセージは、msg へコピーされます。呼出メッセージのサイズは、関数値として返されます。

msg で指し示される領域は、ランデブ用ポート生成時に指定した呼出メッセージの最大長以上としてください。

*p_rdvno には、後で fwd_por や rpl_rdv システムコールを発行する際に使用するランデブ番号が返されます。ランデブ終了待ち中の呼出側タスクはポートから切り離されており、ポート番号ではなく、タスク固有のランデブ番号による特定が必要なためです。

戻　値　**正の値は呼び出しメッセージサイズ（バイト）**
E_PAR　受付側選択条件を表すビットパターン acpptn が 0*
E_ID　ランデブ用ポート ID が範囲外 *
E_NOEXS ランデブ用ポート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_DLT　待ちの間にランデブ用ポートが削除された

補　足　tacp_por(porid, acpptn, p_rdvno, msg, TMO_FEVR) と同じです。

例

```
#define ID_por1 1
#define ID_por2 2

TASK task1(void)
{
    B msg[64] ;
    ER_UINT size;
    RDVNO rdvno;
      :
    strcpy(msg, "Welcome");
    size = acp_por(ID_por1, 0xffff, &rdvno, (VP)msg);
    if (memcmp(msg, "Hello", size) == 0)
    {   strcpy(msg, "World");
        rpl_rdv(rdvno, msg, strlen(msg));
    } else
        fwd_por(ID_por2, 0x0001, rdvno, msg, strlen(msg));
      :
      :
}
```

ｐａｃｐ＿ｐｏｒ

機　能　ポートに対するランデブ受付（ポーリング）

形　式
```
ER_UINT = pacp_por(ID porid, RDVPTN acpptn, RDVNO *p_rdvno, VP msg);
```
porid　ランデブ用ポート ID
acpptn　受付側選択条件を表すビットパターン
p_rdvno　ランデブ番号を格納する場所へのポインタ
msg　呼出メッセージを格納する場所へのポインタ

解　説　acp_por との違いは次の通りです。

ランデブ呼出待ちタスクがない場合や、どの呼出側タスクともランデブが成立しない場合、待ち状態に入らずに、E_TMOUT エラーを返します。

戻　値　**正の値は呼出メッセージのサイズ (バイト数)**
E_PAR　受付側選択条件を表すビットパターン acpptn が 0*
E_ID　ランデブ用ポート ID が範囲外 *
E_NOEXS　ランデブ用ポートが生成されていない
E_TMOUT　ポーリング失敗

補　足　tacp_por(porid, acpptn, p_rdvno, msg, TMO_POL) と同じです。

ｔａｃｐ＿ｐｏｒ

機　能　ポートに対するランデブ受付（タイムアウト有）

形　式
```
ER_UINT = tacp_por(ID porid, RDVPTN acpptn, RDVNO *p_rdvno, VP msg, TMO tmout);
```
cmsgsz　呼出メッセージのサイズ (バイト数)
porid　ランデブ用ポート ID
acpptn　受付側選択条件を表すビットパターン
p_rdvno ランデブ番号を格納する場所へのポインタ
msg　　呼出メッセージを格納する場所へのポインタ
tmout　タイムアウト値

解　説　acp_por 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ランデブが成立しない場合、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 pacp_por 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 acp_por と同じ動作になります。

戻　値　**正の値は呼出メッセージのサイズ (バイト数)**
E_PAR　受付側選択条件を表すビットパターン acpptn が 0*
E_ID　　ランデブ用ポート ID が範囲外 *
E_NOEXS ランデブ用ポート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_DLT　待ちの間にランデブ用ポートが削除された
E_TMOUT タイムアウト

fwd_por

機　能　ポートに対するランデブ回送

形　式
```
ER fwd_por(ID porid, RDVPTN calptn, RDVNO rdvno, VP msg, UINT cmsgsz);
```
porid　回送先のランデブ用ポート ID
calptn　呼出側選択条件を表すビットパターン
rdvno　回送前のランデブ番号
msg　呼出メッセージへのポインタ
cmsgsz　呼出メッセージのサイズ（バイト数）

解　説　受け付けたランデブを他のポート（同じポートでもよい）へ回し、別のタスクに改めてランデブ受付をおこなわせます。

ランデブ終了待ち状態だった呼出側タスクは、最初に呼び出したポートとは別のポートで、再度、ランデブ呼出処理をおこなうことになります。また、ランデブ成立判定に使われるビットパターンは、このシステムコールの calptn と置き換えられます。

回送後のポートで、ランデブ受付待ちのタスクがある場合、受付待ちタスクとランデブが成立するか調べます（受付待ちタスクが複数ある場合は、受付待ち行列の先頭タスクから順に、ランデブ成立まで）。ランデブ受付待ちタスクがない場合や、どの受付側タスクともランデブが成立しない場合、回送の対象となった呼出側タスクは、ランデブ呼出待ちとして待ち行列につながれます。

ランデブが成立したら、呼出メッセージを受付側タスクのバッファへコピーし、そのタスクの受付待ちを解除します。そして回送の対象となった呼出側タスクは、再び、ランデブ終了待ち状態になります。

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。このシステムコールを発行できるのは、ランデブ受付をおこなった後に限ります。回送されたランデブを、さらに回送することも可能です。

戻　値
E_OK　正常終了
E_PAR　呼出側選択条件を表すビットパターン calptn が 0*
　　メッセージサイズが範囲外（cmsgsz = 0，cmsgsz > maxcmsz）*
E_ID　ランデブ用ポート ID が範囲外 *
E_OBJ　対象タスクがランデブ終了待ちでない
　　回送後のポートの maxrmsz が、回送前の maxrmsz より大きい *
E_NOEXS　ランデブ用ポートが生成されていない

ｒｐｌ＿ｒｄｖ

機　能　ランデブ返答

形　式　**ER rpl_rdv(RDVNO rdvno, VP msg, UINT rmsgsz);**
rdvno　ランデブ番号
msg　　返答メッセージへのポインタ
rmsgsz　返答メッセージのサイズ（バイト数）

解　説　rdvno で特定されるランデブ呼出側タスクに返答メッセージを渡し、一連のランデブ処理を終了させます。返答メッセージは、ランデブ呼出側タスクのバッファへコピーされます。

ランデブ呼出側タスクは、ランデブ終了待ちの WAITING 状態から、READY 状態へと遷移します（現在の RUNNING タスクより高優先なら RUNNING 状態へ、WAITING-SUSPENDED 状態の場合は SUSPENDED 状態へ遷移）。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。

このシステムコールを発行できるのは、ランデブ受付をおこなった後に限ります。

戻　値　**E_OK　正常終了**
E_PAR　メッセージサイズが範囲外
E_OBJ　対象タスクがランデブ終了待ちでない *

ｒｅｆ＿ｐｏｒ

機　能　ポート状態参照

形　式
```
ER ref_por(ID porid, T_RPOR *pk_rpor);
porid    ランデブ用ポート ID
pk_rpor 状態パケットを格納する場所へのポインタ
```

解　説　porid で指定されたランデブ用ポートの状態を、*pk_rpor に返します。ランデブ用ポート状態パケットの構造は次の通りです。

```
typedef struct t_rpor
{    ID ctskid;              呼出待ちタスク ID または TSK_NONE
     ID atskid;              受付待ちタスク ID または TSK_NONE
} T_RPOR;
```

ctskid と atskid 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     ランデブ用ポート ID が範囲外
E_NOEXS ランデブ用ポートが生成されていない
```

例
```
#define ID_por1 1

TASK task1(void)
{
    T_RPOR rpor;
      :
    ref_por(ID_por1, &rpor);
    if (rpor.atskid != TSK_NONE)
         :
}
```

ｒｅｆ＿ｒｄｖ

機　能　ランデブ状態参照

形　式
```
ER ref_rdv(RDVNO rdvno, T_RRDV *pk_rrdv);
rdvno    ランデブ番号
pk_rrdv 状態パケットを格納する場所へのポインタ
```

解　説　rdvno で指定されたランデブの状態を、*pk_rrdv に返します。ランデブ状態パケットの構造は次の通りです。

```
typedef struct t_rrdv
{
    ID wtskid;      ランデブ終了待ちタスク ID または TSK_NONE
} T_RRDV;
```

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    ランデブ用ポート ID が範囲外
E_NOEXS ランデブ用ポートが生成されていない
```

例
```
TASK task1(void)
{
    T_RRDV rrdv;
    RDVNO rdvno;
      :
    ref_rdv(rdvno, &rrdv);
    if (rrdv.wtskid != TSK_NONE)
      :
}
```

５．１１ 割込み管理機能

def_inh

機　能　割込みハンドラ定義

形　式
```
ER def_inh(INHNO inhno, const T_DINH *pk_dinh);
inhno   割込みハンドラ ID
pk_dinh 割込みハンドラ定義情報パケットへのポインタ
```

解　説　inhno で指定される割込みベクタテーブルに、inthdr で示される割込みハンドラを設定します。割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(割込みベクタ番号が一般的)。

割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、割込みハンドラ開始時の割込みマスク imask が追加されている場合があります。

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

inhatr の値は NORTi 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 を入れてください。

プロセッサに依存しますので、カーネルとは別の n4ixxx.c にサンプルが記述されています。これが、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、組み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。割込みハンドラのアドレスを、直接、割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   割込み定義番号 inhno が範囲外 *
```

ｅｎｔ＿ｉｎｔ

機　能　割込みハンドラ開始

形　式　**void ent_int(void);**

解　説　割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込みハンドラ専用領域に切り替えます。必ず割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込みルーチン（マルチタスク動作に関与する他の割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　割込みハンドラを C で記述できるようにするための、NORTi 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　割込みハンドラから復帰

形　式　**void ret_int(void);**

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

割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　割込みマスク変更

形　式
```
ER chg_ims(UINT imask);
```
imask 割込みマスク値

解　説　プロセッサの割込みマスクを、imask で指定した値に変更します。

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

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

割込み禁止中に発行されたシステムコールで、タスク切り替が必要となった場合は、chg_ims(0) が発行されて割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　割込みマスク参照

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

解　説　プロセッサの割込みマスクを参照し、*p_imask に返します。

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

レベル割込み機能のあるプロセッサでは、*p_imask の値で割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込みマスクを、割込み禁止状態にセットします。レベル割込みの機能を持ったプロセッサでは、最高の割込みレベルに設定して、全割込みを禁止します。

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 独自のシステムコールです。vset_psw と組合せて、一時的な割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込みルーチンからも発行できます。

例

```
voidfunc(void)
{
    UINT psw;

    psw = vdis_psw();       割込み禁止
      :
    set_psw(psw);           割込み禁止 / 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);        割込み禁止 / 許可を調べる
    chg_ims(7);             割込み禁止（imask 値はプロセッサ依存）
      :
    chg_ims(imask);         割込み禁止 / 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 独自のシステムコールです。割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込みルーチンからも発行できます。

例

```
void func(void)
{
    UINT psw;

    psw = vdis_psw();
      :
    set_psw(psw |0x8000);
      :
}
```

cre_isr

機　能　割込みサービスルーチンの生成

形　式
```
ER cre_isr(ID isrid, const T_CISR *pk_cisr);
isrid    割込みサービスルーチン ID
pk_cisr 割込みサービスルーチン生成情報パケットへのポインタ
```

解　説　intno で指定される割込み番号に、isr で示される割込みサービスルーチンを設定します。割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込みハンドラテーブルへ、isr を設定します。intno の内容はプロセッサにより異なります。割込みベクタ番号または割込み要因番号が一般的です。

割込みサービスルーチン生成情報パケットの構造は次の通りです。

```
typedef struct t_cisr
{   ATR istatr;          割込みハンドラ属性
    VP_INT exinf;        拡張情報
    INTNO intno;         割込み番号
    FP isr;              割込みサービスルーチンアドレス
    UINT imask;          割込みマスク（一部のプロセッサのみ）
} T_CISR;
```

inhatr の値は NORTi 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 を入れてください。exinf は、割込みサービスルーチン起動時に第一引数として渡されます。

プロセッサに依存するので、カーネルとは別の n4ixxx.c に記述された関数を呼び出しています。これが、ユーザーのシステムに適合しない場合は、独自の関数を作成してください。

割込みサービスルーチンでは、割込み処理の入り口 / 出口処理を OS がおこなうため、ent_int, ret_int 等を呼ぶ必要がありません。また auto 変数の使用禁止などの割込みハンドラにある制限は無いので、一般の C 関数として記述することが出来ます。ただし、カーネルレベルより優先度の高い割込みに対して割込みサービスルーチンを使用することはできません。

同一の割込み番号に対して複数の割込みサービスルーチンを生成することができます。

割込み番号と割込みハンドラ ID が同一のプロセッサ（SH2, SH1）の場合でも、ひとつの割込み番号に対して割込みハンドラと割込みサービスルーチンを重複して定義することはできません。

戻　値　　**E_OK**　**正常終了**
E_PAR　**割込み番号 intno が範囲外 ***
E_ID　**ID が範囲外 ***
E_SYS　**管理ブロック用のメモリが確保できない ****

ａｃｒｅ＿ｉｓｒ

機　能　割込みサービスルーチンの生成（ID 自動割り当て）

形　式
```
ER_ID cre_isr(const T_CISR *pk_cisr);
pk_cisr 割込みサービスルーチン生成情報パケットへのポインタ
```

解　説　未生成割込みサービスルーチンの ID を、大きな方から検索して割り当てます。割込みサービスルーチン 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_isr と同じです。

戻　値
```
正の値ならば、割り当てられた割込みサービスルーチン ID
E_NOID   割込みサービスルーチン ID が不足
E_CTX    割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_isr1;
extern void sioist(VP_INT);
const T_CISR cisr1 = { TA_HLNG, NULL, INT_SIO1, sioisr, 0X07 };

TASK task1(void)
{
    ER_ID ercd;
      :
    ercd = acre_isr(&cisr1);
    if (ercd > 0)
        ID_isr1 = ercd;
}
```

del_isr

機　能　割込みサービスルーチンの削除

形　式
```
ER del_isr(ID isrid);
isrid   割込みサービスルーチン ID
```

解　説　isrid で指定される割込みサービスルーチンを削除します。

戻　値
```
E_OK    正常終了
E_ID    ID が範囲外 *
E_NOEXS オブジェクト未生成
E_CTX   割込みハンドラから実行 *
```

ref_isr

機　能　割込みサービスルーチンの状態参照

形　式
```
ER del_isr(ID isrid, T_RISR *pk_risr);
isrid   割込みサービスルーチン ID
pk_risr 割込みサービスルーチン状態パケットを格納する場所へのポインタ
```

解　説　isrid で指定される割込みサービスルーチンの状態を参照し、pk_risr に返します。

割込みサービスルーチン状態パケットの構造は次の通りです。

```
typedef struct t_risr
{   INTNO intno;     割込み番号
    UINT imask;      割込みマスク（一部のプロセッサのみ）
} T_RISR;
```

戻　値
```
E_OK    正常終了
E_ID    ID が範囲外 *
E_NOEXS オブジェクト未生成
```

５．１２ メモリプール管理機能（可変長）

ｃｒｅ＿ｍｐｌ

機　能　可変長メモリプール生成

形　式
```
ER cre_mpl(ID mplid, const T_CMPL *pk_cmpl);
mplid    可変長メモリプール ID
pk_cmpl 可変長メモリプール生成情報パケットへのポインタ
```

解　説　mplid で指定された可変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可変長メモリプール管理ブロックを動的に割り当て、また pk_mpl->mpl が NULL の場合メモリプール用メモリから pk_cmpl->mplsz 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。

定義情報パケットを ROM 以外に置いた場合、すなわち const を付けなかった場合、定義情報パケットはシステムメモリにコピーされます。

可変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_cmpl
{   ATR mplatr;             可変長メモリプール属性
    SIZE mplsz;             メモリプール全体のサイズ（バイト数）
    VP mpl;                 メモリプールの先頭アドレスまたは NULL
    B *name;                メモリプール名へのポインタ
} T_CMPL;
```

可変長メモリプール属性 mplatr には次の値を入れてください。

```
TA_TFIFO        獲得待ちタスク行列は先着順（FIFO）
TA_TPRI         獲得待ちタスク行列はタスク優先度順
```

ユーザプログラムでメモリプール領域を用意する場合は、確保した領域の先頭番地とバイトサイズを pk_cmpl->mpl と pk_cmpl->mplsz に設定してください。OS が使用するオーバヘッドがあるため確保した領域全てをユーザプログラムから使用できるわけではありません。

```
TSZ_MPL(cnt, size)
```

によって size バイトのデータを cnt 個確保するのに必要なトータルメモリサイズを得ることができます。

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	**可変長メモリプール ID が範囲外 ***
E_OBJ	**可変長メモリプールが既に生成されている**
E_CTX	**割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 ****

注意１　mplsz バイトで生成したメモリプールのうち、先頭の sizeof(int *) + sizeof(int) バイトだけ、すなわち、32 ビット CPU で８バイト、16 ビット CPU で６バイトを、OS が管理用に使用します。そして、１回のメモリブロック獲得毎に、sizeof(int) バイトだけ、すなわち、32 ビット CPU で４バイト、16 ビット CPU で２バイトを、OS が管理用に使用します。したがって、この OS 使用分を加味して、mplsz の値は決めてください。さらに、ワード境界をそろえるためにサイズによっては無駄な領域を取られる場合があります。

注意２　獲得と返却を繰り返すと、メモリが断片化する可能性があります。すなわち、連続空き領域のサイズが小さくなる可能性があります。（ガーベジコレクション機能はありません。）

例

```
#define ID_mpl1 1
const T_CMPL cmpl1 = { TA_TFIFO, 1024, NULL };

TASK task1(void)
{
    ER ercd;
      :
    ercd = cre_mpl(ID_mpl1, &cmpl1);
      :
}
```

ａｃｒｅ＿ｍｐｌ

機　能　可変長メモリプール生成（ID 自動割り当て）

形　式
```
ER_ID vcre_mpl(const T_CMPL *pk_cmpl);
pk_cmpl 可変長メモリプール生成情報パケットへのポインタ
```

解　説　未生成可変長メモリプールの ID を、大きな方から検索して割り当てます。可変長メモリプール 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_mpl と同じです。

戻　値
```
正の値ならば、割り当てられた可変長メモリプール ID
E_NOID   可変長メモリプール ID が不足
E_CTX    割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_mpl1;
const T_CMPL cmpl1 = { TA_TFIFO, 1024, NULL };

TASK task1(void)
{
    ER_ID ercd;
       :
    ercd = acre_mpl(&cmpl1);
    if (ercd > 0)
        ID_mpl1 = ercd;
       :
}
```

ｄｅｌ＿ｍｐｌ

機　能　可変長メモリプール削除

形　式
```
ER del_mpl(ID mplid);
mplid　可変長メモリプール ID
```

解　説　mplid で指定された可変長メモリプールを削除します。すなわち、OS が確保した場合はメモリプール領域をメモリプール用メモリへ解放し、可変長メモリプール管理ブロックをシステムメモリへ解放します。

この可変長メモリプールに対して、獲得を待っているタスクがあった場合、このタスクの待ちを解除します。待ち解除されたタスクへは、削除されたことを示す E_DLT エラーが返ります。

戻　値
```
E_OK    正常終了
E_ID    可変長メモリプール ID が範囲外 *
E_NOEXS 可変長メモリプールが生成されていない
E_CTX   割込みハンドラから発行 *
```

例
```
#define ID_mpl1 1

TASK task1(void)
{
      :
    del_mpl(ID_mpl1);
      :
}
```

get_mpl

機　能　可変長メモリブロック獲得

形　式
```
ER get_mpl(ID mplid, UINT blksz, VP *p_blk);
```
mplid　可変長メモリプール ID
blksz　メモリブロックサイズ（バイト数）
p_blk　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　mplid で指定された可変長メモリプールから、blksz で指定されるサイズのメモリブロックを切り出し、そのメモリブロックへのポインタを *p_blk に返します。獲得したメモリのゼロクリア等はおこなわれません。内容は不定です。

可変長メモリプールの空き領域が足りない場合、本システムコールの発行タスクは、その可変長メモリプールの待ち行列につながれます。

メモリブロックサイズ blksz の最小値は、１バイト以上です。ただし、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 のメモリブロックを得るためには、可変長メモリプールに blksz + sizeof(int) の空きが必要です。

要求するメモリブロックのサイズが小さい方を優先させる処理はおこなっていません。

戻　値
E_OK　正常終了
E_ID　可変長メモリプール ID が範囲外 *
E_NOEXS 可変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_DLT　待ちの間に可変長メモリプールが削除された

注　意　p_blk は、ポインタへのポインタです。

補　足　tget_mpl(mplid, blksz, p_blk, TMO_FEVR) と 同じです。

例

```
#define ID_mpl1 1

TASK task1(void)
{
    B *blk;
      :
    get_mpl(ID_mpl1, 256, (VP *)&blk);
    blk[0] = 0;
    blk[1] = 1;
      :
}
```

ｐｇｅｔ＿ｍｐｌ

機　能　可変長メモリブロック獲得（ポーリング）

形　式
```
ER pget_mpl(ID mplid, UINT blksz, VP *p_blk);
```
mplid 可変長メモリプール ID
blksz メモリブロックサイズ（バイト数）
p_blk 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_mpl との違いは次の通りです。

可変長メモリプールの空き領域が足りない場合、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　可変長メモリプール ID が範囲外 *
E_NOEXS 可変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
E_CTX　割込みハンドラから発行 *

注　意　p_blk は、ポインタへのポインタです。

補　足　tget_mpl(mplid, blksz, p_blk, TMO_POL) と同じです。

例
```
#define ID_mpl1 1

TASK task1(void)
{
    B *blk;
    ER ercd;
      :
    ercd = pget_mpl(ID_mpl1, 256, (VP *)&blk);
    if (ercd == E_OK)
        :
}
```

tget_mpl

機　能　可変長メモリブロック獲得（タイムアウト有）

形　式
```
ER tget_mpl(ID mplid, UINT blksz, VP *p_blk, TMO tmout);
mplid   可変長メモリプール ID
blksz   メモリブロックサイズ（バイト数）
p_blk   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_mpl 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プールの空き領域が不足したままの場合、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 pget_mpl 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_mpl と同じ動作になります。

戻　値
```
E_OK     正常終了
E_ID     可変長メモリプール ID が範囲外 *
E_NOEXS 可変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_DLT    待ちの間に可変長メモリプールが削除された
E_TMOUT タイムアウト
```

注　意　p_blk は、ポインタへのポインタです。

例
```
#define ID_mpl1 1

TASK task1(void)
{
    B *blk;
    ER ercd;
       :
    ercd = tget_mpl(ID_mpl1, 256, (VP *)&blk, 100/MSEC);
    if (ercd == E_OK)
       :
}
```

ｒｅｌ＿ｍｐｌ

機　能　可変長メモリブロック返却

形　式
```
ER rel_mpl(ID mplid, VP blk);
mplid   可変長メモリプール ID
blk     メモリブロックへのポインタ
```

解　説　mplid で指定された可変長メモリプールへ、blk で指し示されるメモリブロックを返却します。

この可変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り、返却の結果、メモリープールの空きサイズが、先頭の待ちタスクの要求サイズ以上になったならば、このタスクにメモリブロックを獲得させ、待ちを解除します。複数の待ちタスクの中で、要求サイズが小さい方を優先させるような処理はおこなっていません。

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場合があります。この場合は、待ち行列の先頭から順に、獲得待ちを解除していきます。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。返却せずにタスクを終了等した場合メモリリークが発生します。

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    可変長メモリプール ID が範囲外 *
E_NOEXS 可変長メモリプールが生成されていない
E_CTX   割込みハンドラから発行 *
```

例
```
#define ID_mpl1 1

TASK task1(void)
{
    B *blk;
      :
    get_mpl(ID_mpl1, 256, (VP *)&blk);
      :
    rel_mpl(ID_mpl1, (VP)blk);
      :
}
```

ｒｅｆ＿ｍｐｌ

機　能　可変長メモリプール状態参照

形　式
```
ER ref_mpl(ID mplid, T_RMPL *pk_rmpl);
mplid   可変長メモリプール ID
pk_rmpl 可変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　mplid で指定された可変長メモリプールの状態を、*pk_rmpl に返します。

可変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_rmpl
{   ID wtskid;          待ちタスクの ID または TSK_NONE
    SIZE fmplsz;        空き領域の合計サイズ（バイト数）
    UINT fblksz;        最大の連続空き領域のサイズ（バイト数）
} T_RMPL;
```

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    可変長メモリプール ID が範囲外
E_NOEXS 可変長メモリプールが生成されていない
```

例
```
#define ID_mpl1 1

TASK task1(void)
{
    T_RMPL rmpl;
      :
    ref_mpl(ID_mpl1, &rmpl);
    if (rmpl.fmplsz >= 256 +sizeof(int))
      :
}
```

５．１３ 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_mpf->mpf が NULL の場合メモリプール用メモリから、blkcnt × blfsz 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_mpf->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
    UINT blkcnt;         メモリプール全体のブロック数
    UINT blfsz;          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
} T_CMPF;
```

固定長メモリプール属性 mpfatr には次の値を入れてください。

```
TA_TFIFO        受信待ちタスク行列は先着順（FIFO）
TA_TPRI         受信待ちタスク行列はタスク優先度順
```

メモリブロックサイズ blfsz 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、blfsz 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 blfsz 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 blfsz 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_OBJ　固定長メモリプールが既に生成されている
E_CTX　割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
```
正の値ならば、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;
       :
}
```

ｄｅｌ＿ｍｐｆ

機　能　固定長メモリプール削除

形　式
```
ER del_mpf(ID mpfid);
mpfid　固定長メモリプール ID
```

解　説　mpfid で指定された固定長メモリプールを削除します。すなわち、OS が確保した場合はメモリプール領域をメモリプール用メモリへ解放し、固定長メモリプール管理ブロックをシステムメモリへ解放します。

この固定長メモリプールに対して、獲得を待っているタスクがあった場合、このタスクの待ちを解除します。待ち解除されたタスクへは、削除されたことを示す E_DLT エラーが返ります。

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
E_CTX   割込みハンドラから発行 *
```

例
```
#define ID_mpf1 1

TASK task1(void)
{
      :
    del_mpf(ID_mpf1);
      :
}
```

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);
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
mpfid   固定長メモリプール ID
```

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 blfsz 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）
E_DLT    待ちの間に固定長メモリプールが削除された
```

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、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_DLT    待ちの間に固定長メモリプールが削除された
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。

可変長のメモリブロックとは異なり、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)
        :
}
```

５．１４ 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 *pk_systim);**
pk_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、*pk_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;          周期ハンドラ起動フェーズ
} T_CCYC;
```

cycatr には、次の値を入れてください。

TA_STA　ハンドラ生成後動作状態とします。指定しない場合は動作していない状態とします。

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。

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　**周期ハンドラ活性状態が不正 ***
E_CTX　**割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 ****

※ NORTi カーネル 4.05.00 以前では、E_ID が E_PAR と誤実装されています。

例

```
#define ID_cyc1 1
extern void cyc1(VP_INT);
const T_DCYC dcyc1 = { TA_STA, NULL, cyc1, 10, 5 };

TASK task1(void)
{
    ER ercd;
      :
    ercd = cre_cyc(ID_cyc1, &dcyc1);
      :
}
```

ａｃｒｅ＿ｃｙｃ

機　能　周期ハンドラ生成（番号自動割り当て）

形　式
```
ER_ID acre_cyc(const T_DCYC *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
正の値ならば、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　周期ハンドラ活性状態が不正 *
E_CTX　割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_DCYC dcyc1 = { TA_STA, NULL, cyc1, 10, 5 };

TASK task1(void)
{
    ER_ID ercd;
      :
    ercd = acre_cyc(&dcyc1);
    if (ercd > 0)
        ID_cyc1 = ercd;
      :
}
```

ｄｅｌ＿ｃｙｃ

機　能　周期ハンドラの削除

形　式
```
ER del_cyc(ID cycid);
cycid   周期ハンドラ ID
```

解　説　cycid で指定された周期ハンドラを削除します。すなわち、周期ハンドラ管理ブロックをシステムメモリへ解放します。

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
E_CTX   割込みハンドラから発行 *
```

例
```
ID ID_cyc1;

TASK task1(void)
{
    ER ercd;
      :
    ercd = del_cyc(ID_cyc1);
      :
}
```

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

※ NORTi カーネル 4.05.00 以前では、E_ID が E_PAR と誤実装されています。

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)
         :
}
```

ｃｒｅ＿ａｌｍ

機　能　アラームハンドラ生成

形　式
```
ER cre_alm(ID almid, const T_DALM *pk_calm);
almid   アラームハンドラ ID
pk_calm アラームハンドラ生成情報パケットへのポインタ
```

解　説　almid で指定されるアラームハンドラを生成します。すなわち、システムメモリから、アラームハンドラ管理ブロックを動的に割り当てます。

アラーム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_calm
{   ATR almatr;             アラームハンドラ属性
    VP_INT exinf;           拡張情報
    FP almhdr;              アラームハンドラとする関数へのポインタ
} T_CALM;
```

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

almatr の値は NORTi 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 を入れてください。exinf の値はアラームハンドラに第一引数として渡されます。

戻　値
```
E_OK    正常終了
E_ID    アラームハンドラ ID 番号が範囲外 *
E_PAR   パラメータエラー *
E_OBJ   アラームハンドラが登録済み *
E_CTX   割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_alm1 1
extern void alm1(VP_INT);
const T_DALM dalm1 = { TA_HLNG, NULL, alm1 };

TASK task1(void)
{
    ER ercd;
      :
    ercd = cre_alm(ID_alm1, &dalm1);
      :
}
```

ａｃｒｅ＿ａｌｍ

機　能　アラームハンドラ生成（番号自動割り当て）

形　式

```
ER_ID acre_alm(const T_DALM *pk_calm);
```

pk_calm アラームハンドラ生成情報パケットへのポインタ

解　説　未生成のアラームハンドラ ID を、大きな方から検索して割り当てます。アラームハンドラ 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_alm と同じです。

戻　値

正の値ならば、割り当てられたアラームハンドラ ID
E_NOID　アラームハンドラ ID が不足
E_OBJ　アラームハンドラが登録済み *
E_CTX　割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_alm1;
extern void alm1(VP_INT);
const T_DALM dalm1 = { TA_HLNG, NULL, alm1 };

TASK task1(void)
{
    ER_ID ercd;
      :
    ercd = acre_alm(&dalm1);
    if (ercd > 0)
        ID_alm1 = ercd;
      :
}
```

ｄｅｌ＿ａｌｍ

機　能　アラームハンドラの削除

形　式
```
ER del_alm(ID almid);
almid アラームハンドラ ID
```

解　説　almid で指定されたアラームハンドラを削除します。すなわち、アラームハンドラ管理ブロックをシステムメモリへ解放します。

戻　値
```
E_OK     正常終了
E_ID     アラームハンドラ ID が範囲外 *
E_NOEXS アラームハンドラが生成されていない
E_CTX    割込みハンドラから発行 *
```

例
```
ID ID_alm1;

TASK task1(void)
{
    ER ercd;
      :
    ercd = del_alm(ID_alm1);
      :
}
```

sta__alm

機　能　アラームハンドラ動作開始

形　式
```
ER sta_alm(ID almid, RELTIM almtim);
almid   アラームハンドラ ID 番号
almtim  アラームハンドラ起動時刻 ( 相対時刻 )
```

解　説　almid で指定されるアラームハンドラの起動時刻を almtim に設定し動作を開始します。動作中のハンドラが指定された場合は起動時刻を新しい値に変更します。起動時刻は sta_alm が呼ばれた時刻を基準とした相対時刻で単位はタイマ割込み間隔です。

戻　値
```
E_OK    正常終了
E_ID    アラームハンドラ ID が範囲外 *
E_NOEXS アラームハンドラが生成されていない
```

stp__alm

機　能　アラームハンドラ動作停止

形　式
```
ER stp_alm(ID almid);
almid   アラームハンドラ ID 番号
```

解　説　almid で指定されるアラームハンドラの起動時刻を解除し動作していない状態にします。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。

戻　値
```
E_OK    正常終了
E_ID    アラームハンドラ ID が範囲外 *
E_NOEXS アラームハンドラが生成されていない
```

※ NORTi カーネル 4.05.00 以前では、E_ID が E_PAR と誤実装されています。

ｒｅｆ＿ａｌｍ

機　能　アラームハンドラ状態参照

形　式
```
ER ref_alm(ID almid, T_RALM *pk_ralm);
almid   アラームハンドラ ID
pk_ralm アラーム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　almid で指定されるアラームハンドラの状態を、*pk_ralm に返します。

アラームハンドラ状態パケットの構造は次の通りです。

```
typedef struct t_ralm
{   STAT almstat;           ハンドラの状態
    RELTIM lefttim;         起動までの時間
} T_RALM;
```

almstat には次の値が返ります。

```
TALM_STP        アラームハンドラが動作していない
TALM_STA        アラームハンドラが動作していない
```

lefttim には起動までの残り時間が返ります。

戻　値
```
E_OK    正常終了
E_ID    アラームハンドラ ID が範囲外 *
E_NOEXS アラームハンドラが生成されていない
```

※ NORTi カーネル 4.05.00 以前では、E_ID が E_PAR と誤実装されています。

例
```
#define ID_alm1 1

TASK task1(void)
{
    T_RALM ralm;
      :
    ref_alm(ID_alm1, &ralm);
    if (ralm.lefttim > 100/MSEC)
        :
}
```

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、周期タイマー割込みが入ったことを知らせます。割込みハンドラ専用です。

戻　値　**なし**

補　足　NORTi 独自のシステムコールです。

例

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

ｄｅｆ＿ｏｖｒ

機　能　オーバーランハンドラの定義

形　式　**ER def_ovr(const T_DOVR *pk_dovr);**
pk_dovr オーバーランハンドラ定義情報を入れたパケットへのポインタ

解　説　オーバーランハンドラを定義情報に基づいて定義します。

オーバーランハンドラ定義情報パケットの構造は次の通りです。

```
typedef struct t_dovr
{   ATR ovratr;             オーバーランハンドラ属性
    FP ovrhdr;              オーバーランハンドラ先頭アドレス
    INTNO intno;            使用する周期割込み番号
    FP ovrclr;              割込み要因クリア処理関数先頭アドレス
    UINT imask;             割込みマスク
} T_DOVR;
```

ovratr の値は NORTi 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 を入れてください。exinf の値はオーバーハンドラの第二引数として渡されます。

ovrhdr は、オーバーランハンドラとする関数へのポインタです。オーバーランハンドラは、void 型の関数として以下の様に記述してください。

```
void ovrhdr(ID tskid, VP_INT exinf)
{
      :
      :
}
```

intno にはオーバーランハンドラが使用する周期割込み番号を指定してください。一般的には、システムクロックと同じ周期割込み番号を使用します。ovrclr には、割込み要因をクリアするための関数を指定してください。割込み番号としてシステムクロックと同一の割込み番号を使用した場合、ovrclr には NULL を指定してください。

システムクロックとは異なる割込み番号を使用する場合は、割込みの初期化ルーチンと ovrclr を独自に作成する必要があります。ovrclr に登録した関数は割込みが入るたびにコールされます。

pk_dovr に NULL を指定するとオーバーランハンドラ定義を解除します。すでに定義してある状態で再度 pk_dovr に NULL 以外の値を指定するとオーバーランハンドラを再定義します。独自の割込みを使用する場合は、割込みを禁止してから定義解除／再定義してください。

オーバーランハンドラは指定された割込み番号に対して、内部で割込みサービスルーチンを生成／削除します。

戻　値

E_OK	正常終了
E_NOID	割込みサービスルーチン ID が不足
E_CTX	割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_PAR	割込み番号 intno が範囲外 *
その他	pk_dovr = NULL の時は acre_isr のエラー値
	pk_dovr ≠ NULL del_isr のエラー値

例

```
#define INT_CMT INT_CMI0
extern void ovrhdr(ID, VP_INT);
const T_DALM dovr = { TA_HLNG, ovrhdr, INT_CMT, NULL, 0x07 };

TASK task1(void)
{
    ER ercd;
      :
    ercd = def_ovr(&dovr);
      :
}
```

ｓｔａ＿ｏｖｒ

機　能　オーバーランハンドラの動作開始

形　式　**ER sta_ovr(ID tskid, OVRTIM ovrtim);**
tskid　持ち時間を設定するタスクの ID
ovrtim　持ち時間

解　説　tskid で指定されるタスクに対してプロセッサ時間を設定します。tskid に TSK_SELF を指定すると自タスクを対象とします。時間単位はdef_ovrで指定した割込み周期です。ovrtimで指定した時間を使い切るとオーバーランハンドラが起動されます。プロセッサ時間の計測は割込み時に実行していたタスクに対してプロセッサ時間を -1 します。したがって、連続実行時間が割込み周期に対して充分長いタスク以外では誤差が大きくなります。

すでにプロセッサ時間が設定してあるタスクに対して再度sta_ovrをおこなうとプロセッサ時間を更新します。

戻　値　**E_OK　正常終了**
E_ID　タスク ID が不正 *
E_NOEXS タスク未生成
E_PAR　持ち時間が不正
E_OBJ　オーバーランハンドラ未定義

ｓｔｐ＿ｏｖｒ

機　能　オーバーランハンドラの動作停止

形　式　**ER stp_ovr(ID tskid);**
tskid　計時を停止するタスクの ID

解　説　tskid で指定されるタスクに対してオーバーランハンドラの動作を停止します。プロセッサ時間の設定を解除します。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 が不正 *
E_OBJ　オーバーランハンドラ未定義

ｒｅｆ＿ｏｖｒ

機　能　オーバーランハンドラ状態参照

形　式
```
ER ref_ovr(ID tskid, T_ROVR *pk_rovr);
tskid    プロセッサ時間を参照するタスクの ID
pk_ralm オーバーラン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　tskid で指定されるタスクのオーバーランハンドラの状態を、*pk_rovr に返します。tskid = TSK_SELF で自タスクを指定できます。

オーバーランハンドラ状態パケットの構造は次の通りです。

```
typedef struct t_rovr
{   STAT ovrstat;          オーバーランハンドラの状態
    OVRTIM leftotm;        プロセッサ残り時間
} T_ROVR;
```

ovrstat には次の値が返ります。

```
TOVR_STP        プロセッサ時間が設定されていない
TOVR_STA        プロセッサ時間が設定されている
```

leftotm には起動までの残り時間が返ります。

戻　値
```
E_OK    正常終了
E_ID    タスク ID が不正 *
E_OBJ   オーバーランハンドラ未定義
```

例
```
TASK task1(void)
{
    T_ROVR rovr;
      :
    ref_ovr(TSK_SELF, &rovr);
    if (rovr.leftotm > 100/MSEC)
      :
}
```

５．１５ サービスコール管理機能

ｄｅｆ＿ｓｖｃ

機　能　拡張サービスコールの定義

形　式
```
ER def_svc(FN fncd, const T_DSVC *pk_dsvc);
fncd    定義対象の機能コード
pk_dsvc 拡張サービスコール定義情報を入れたパケットへのポインタ
```

解　説　fncd で指定される拡張サービスコールを pk_dsvc によって定義します。

拡張サービスコール定義情報パケットの構造は次の通りです。

```
typedef struct t_dsvc
{   ATR svcatr;             拡張サービスコール属性
    FP svcrtn;              拡張サービスコールルーチンアドレス
    INT parn;               拡張サービスコールルーチンのパラメータ数
} T_DSVC;
```

fncd には正の値を設定してください。svcatr の値は NORTi 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を入れてください。

拡張サービスコールルーチンは以下の様な形式の C 関数として記述してください。

```
ER_UINT svcrtn(VP_INT par1, VP_INT par2, ..., VP_INT par6)
{
      :
      :
}
```

parn にはパラメータの数を設定してください。パラメータ数は最小 0 個から最大は 6 個までです。拡張サービスコールルーチンは呼び出したコンテキストで実行されるサブルーチンです。標準システムコールを拡張サービスコールとして登録することも可能です。

戻　値
```
E_OK    正常終了
E_CTX   非タスクコンテキストから発行 *
E_PAR   パラメータエラー
```

例

```
#define svc_ter_tsk     2
const T_DSVC dsvc2      = { TA_HLNG, (FP)v4_ter_tsk, 1 };

Task task1(void)
{
      :
    ercd = def_svc(svc_ter_tsk, &dsvc2);
      :
}
```

ｃａｌ＿ｓｖｃ

機　能　サービスコールの呼出

形　式
```
ER_UINT cal_svc(FN fncd, VP_INT par1, VP_INT par2, ...);
fncd    呼び出すサービスコール機能コード
par1    サービスコールルーチンに渡す第一パラメータ
par2    サービスコールルーチンに渡す第二パラメータ
        ...
par6    サービスコールルーチンに渡す第六パラメータ
```

解　説　fncd で指定されるサービスコールルーチンを、par1 ～ par6 をパラメータとして呼び出します。パラメータは必要な数のみ記述してください。

戻　値
```
サービスコールからの戻り値
E_RSFN  サービスコールルーチン未定義
E_PAR   fncd 不正 *
```

例
```
#define svc_ter_tsk     2
#define Task2           2
const T_DSVC dsvc2      = { TA_HLNG, (FP)v4_ter_tsk, 1 };

Task task1(void)
{
      :
    ercd = def_svc(svc_ter_tsk, &dsvc2);
      :
    ercd = cal_svc(svc_ter_tsk, Task2);
      :
}
```

５．１６ 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一つあるいはな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 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　レベル割込み機能のあるプロセッサの場合、NORTi では、カーネルの割込み禁止レベルを標準で最高とはしていません。loc_cpu で設定される割込みマスクは、カーネルの割込み禁止レベルまでを禁止するのもであり、カーネルより高優先の割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。全割込みマスクが解除されてしまいます。レベル割込みをサポートしているプロセッサの場合、割込みハンドラでは ent_int から戻ってきた時点で、割込みサービスルーチンでは割込みサービスルーチンが呼ばれた時点でその割込みレベルまで割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)
        :
}
```

５．１７ 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_VER *pk_ver);
pk_ver  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 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
    UH spver;           仕様書バージョン
    UH prver;           製品バージョン
    UH prno[4];         製品管理情報
} T_RVER;
```

構造体のメンバーの詳しい意味については、μITRON 仕様書を、実際に返される値については、カーネルのソースファイル n4cxxx.asm を参照してください。

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 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 mbfid_max;       メッセージバッファ ID 上限
    ID porid_max;       ランデブ用ポート ID 上限
    ID mplid_max;       可変長メモリプール 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 限
    ID almno_max;       アラームハンドラ ID 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 almqsz;         アラームハンドラのタイマキューサイズ（バイト数）
    int istksz;         割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
    ID mtxid_max;       ミューテックス ID 上限
    ID isrid_max;       割込みサービスルーチン ID 上限
    ID svcfn_max;       拡張サービスコール機能番号上限
    :（今後、追加される可能性があります）
} T_RCFG;
```

戻　値　**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。

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_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。

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。

オーバーランハンドラで周期タイマ割込みを使用する場合、def_ovr は周期タイマ割込み初期化後呼び出してください。

戻　値　**E_OK　正常終了**
E_PAR　割込み周期等が範囲外（機種依存）

intext

機　能　周期タイマ割込み終了

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　割込み初期化

形　式　**ER intini(void);**

解　説　sysini から割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_TMOUT	0xf..fce (-50)	ポーリング失敗またはタイムアウト
E_RLWAI	0xf..fcf (-49)	待ち状態の強制解除
E_DLT	0xf..fcd (-51)	待ちオブジェクトの削除

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);**	○○×
タスク削除 **del_tsk(tskid);**	○○×
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();**	○××
自タスク終了と削除 **exd_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 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(tskid);**	○○○
待ち状態の強制解除 **rel_wai(tskid);**	○○○
待ち状態の強制解除 **irel_wai(tskid);**	×○○
強制待ち状態へ移行 **sus_tsk(tskid);**	○○○
強制待ち状態からの再開 **rsm_tsk(tskid);**	○○○
強制待ち状態からの強制再開 **frsm_tsk(tskid);**	○○○
自タスクの遅延 **dly_tsk(dlytim);**	○××

凡例
★ NORTi 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割込みハンドラから発行可能

タスク例外処理

	①②③
タスク例外処理ルーチンの定義 **def_tex(tskid, pk_dtex);**	○○×
タスク例外処理要求 **ras_tex(tskid, rasptn);**	○○○
タスク例外処理要求 **iras_tex(tskid, rasptn);**	×○○
タスク例外処理禁止 **dis_tex();**	○○○
タスク例外処理許可 **ena_tex();**	○○○
タスク例外処理禁止状態の参照 **sns_tex();**	○○○
タスク例外処理の状態参照 **ref_tex(tskid, pk_rtex);**	○○○

凡例
★ NORTi 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);**	○○×
セマフォ削除 **del_sem(semid);**	○○×
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 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);**	○○×
イベントフラグ削除 **del_flg(flgid);**	○○×
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 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);**	○○×
データキュー削除 **del_dtq(dtqid);**	○○×
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 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);**	○○×
メールボックス削除 **del_mbx(mbxid);**	○○×
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 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割込みハンドラから発行可能

拡張同期・通信 ミューテックス

	①②③
ミューテックス生成 **cre_mtx(mtxid, pk_cmtx);**	○○×
ミューテックス生成（ID 自動割付け） **acre_mtx(pk_cmtx);**	○○×
ミューテックス削除 **del_mtx(mtxid);**	○○×
ミューテックスのロック **loc_mtx(mtxid);**	○××
ミューテックスのロック（ポーリング） **ploc_mtx(mtxid);**	○○○
ミューテックスのロック（タイムアウト有） **tloc_mtx(mtxid, tmout);**	○××
ミューテックスのロック解除 **unl_mtx(mtxid);**	○○○
ミューテックスの状態参照 **ref_mtx(mtxid, pk_rmtx);**	○○○

凡例
★ NORTi 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割込みハンドラから発行可能

拡張同期・通信機能（メッセージバッファ）

	①②③
メッセージバッファ生成 **cre_mbf(mbfid, pk_cmbf);**	○○×
メッセージバッファ生成（ID 自動割付け） **acre_mbf(pk_cmbf);**	○○×
メッセージバッファ削除 **del_mbf(mbfid);**	○○×
メッセージバッファへ送信 **snd_mbf(mbfid, msg, msgsz);**	○××
メッセージバッファへ送信（ポーリング） **psnd_mbf(mbfid, msg, msgsz);**	○○○
メッセージバッファへ送信（タイムアウト有） **tsnd_mbf(mbfid, msg, msgsz, tmout);**	○××
メッセージバッファから受信 **rcv_mbf(mbfid, msg);**	○××
メッセージバッファから受信（ポーリング） **prcv_mbf(mbfid, msg);**	○○○
メッセージバッファから受信（タイムアウト有） **trcv_mbf(mbfid, msg, tmout);**	○××
メッセージバッファの状態参照 **ref_mbf(mbfid, pk_rmbf);**	○○○

凡例
★ NORTi 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割込みハンドラから発行可能

拡張同期・通信 ランデブ

	①②③
ランデブポート生成 **cre_por(porid, pk_cpor);**	○○×
ランデブポート生成（ID 自動割付け） **acre_por(pk_cpor);**	○○×
ランデブポート削除 **del_por(porid);**	○○×
ランデブポート呼出し **cal_por(porid, calptn, msg, cmsgsz);**	○××
ランデブポート呼出し（タイムアウト有） **tcal_por(porid, calptn, msg, cmsgsz, tmout);**	○××
ランデブポート待受け **acp_por(porid, acpptn, p_rdvno, msg);**	○××
ランデブポート待受け（ポーリング） **pacp_por(porid, acpptn, p_rdvno, msg);**	○○○
ランデブポート待受け（タイムアウト有） **tacp_por(porid, acpptn, p_rdvno, msg, tmout);**	○××
ランデブの回送 **fwd_por(porid, calptn, rdvno, msg, cmsgsz);**	○○○
ランデブの終了 **rpl_rdv(rdvno, msg, rmsgsz);**	○○○
ランデブポートの状態参照 **ref_por(porid, pk_rpor);**	○○○
ランデブの状態参照 **ref_rdv(rdvno, pk_rrdv);**	○○○

凡例
★ NORTi 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);**	○○×
固定長メモリプール削除 **del_mpf(mpfid);**	○○×
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 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割込みハンドラから発行可能

メモリプール管理 可変長

	①②③
可変長メモリプール生成 **cre_mpl(mplid, pk_cmpl);**	○○×
可変長メモリプール生成（ID 自動割付け） **acre_mpl(pk_cmpl);**	○○×
可変長メモリプール削除 **del_mpl(mplid);**	○○×
可変長メモリブロック獲得 **get_mpl(mplid, blksz, p_blk);**	○××
可変長メモリブロック獲得（ポーリング） **pget_mpl(mplid, blksz, p_blk);**	○○×
可変長メモリブロック獲得（タイムアウト有） **tget_mpl(mplid, blksz, p_blk, tmout);**	○××
可変長メモリブロック返却 **rel_mpl(mplid, blk);**	○○×
可変長メモリプールの状態参照 **ref_mpl(mplid, pk_rmpl);**	○○×

凡例
★ NORTi 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 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);**	○○×
周期ハンドラの削除 **del_cyc(cycid);**	○○×
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 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割込みハンドラから発行可能

時間管理 アラームハンドラ

	①②③
アラームハンドラの生成 **cre_alm(almid, pk_calm);**	○○×
アラームハンドラの生成（ID 自動割付け） **acre_alm(pk_calm);**	○○×
アラームハンドラの削除 **del_alm(almid);**	○○×
アラームハンドラの開始 **sta_alm(almid, almtim);**	○○○
アラームハンドラの停止 **stp_alm(almid);**	○○○
アラームハンドラの状態参照 **ref_alm(almid, pk_ralm);**	○○○

凡例
★ NORTi 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割込みハンドラから発行可能

時間管理 オーバランハンドラ

	①②③
オーバーランハンドラの定義 **def_ovr(pk_dovr);**	○○×
オーバーランハンドラの開始 **sta_ovr(tskid, ovrtim);**	○○○
オーバーランハンドラの停止 **stp_ovr(tskid);**	○○○
オーバーランハンドラの状態参照 **ref_ovr(tskid, pk_rovr);**	○○○

凡例
★ NORTi 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 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割込みハンドラから発行可能

割込み管理

	①②③
割込みハンドラの定義 **def_inh(inhno, pk_dinh);**	○○○
割込みサービスルーチン生成 **cre_isr(isrid, pk_cisr);**	○○×
割込みサービスルーチン生成（ID 自動割付け） **acre_isr(pk_cisr);**	○○×
割込みサービスルーチン削除 **del_isr(isrid);**	○○×
割込みサービスルーチン状態参照 **ref_isr(isrid, pk_risr);**	○○○
割込みの禁止 **dis_int(intno);**	○○×
割込みの許可 **ena_int(intno);**	○○×
割込みマスクの変更 **chg_ims(imask);**	○○○
割込みマスクの参照 **get_ims(p_imask);**	○○○
★割込みハンドラ開始 **ient_int();**	××○
★割込みハンドラ終了 **iret_int();**	××○
★ステータスレジスタセット **vset_psw();**	○○○
★ステータスレジスタの割込みマスクセット **vdis_psw();**	○○○

凡例
★ NORTi 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割込みハンドラから発行可能

サービスコール管理機能

①②③

拡張サービスコール定義 **def_svc(fncd, pk_dsvc);**	○○×
サービスコール呼出し **cal_svc(fncd, par1, par2, ...);**	**○？？ : サービスコールに依存**

凡例
★ NORTi 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
コンフィギュ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 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割込みハンドラから発行可能

７．３ 静的API一覧

（本ページの内容は、NORTiコンフィグレータのマニュアルへ移動しました）

７．４ 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
     PRI tskbpri;           ベース優先度
     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
     UINT suscnt;           強制待ち要求カウント
     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;
```

タスク例外処理生成情報パケット

```
typedef struct t_dtex
{    ATR texatr;            タスク例外処理属性
     FP texrtn;             タスク例外処理関数へのポインタ
} T_DTEX;
```

タスク例外処理状態パケット

```
typedef struct t_rtex
{    STAT texstat;          タスク例外処理状態
     TEXPTN pndptn;         例外処理保留起動要因
} T_RTEX;
```

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数
     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;
```

ミューテックス生成情報パケット

```
typedef struct t_cmtx
{    ATR mtxatr;             ミューテックス属性
     PRI ceilpri;            シーリングプロトコルにおける上限優先度
     B *name;                ミューテックス名へのポインタ
} T_CMTX;
```

ミューテックス状態パケット

```
typedef struct t_rmtx
{    ID htskid;              ロックしているタスクの ID
     ID wtskid;              解除待ちしているタスクの ID
} T_RMTX;
```

メッセージバッファ生成情報パケット

```
typedef struct t_cmbf
{    ATR mbfatr;             メッセージバッファ属性
     UINT maxmsz;            メッセージ最大長
     SIZE mbfsz;             メッセージバッファサイズ
     VP mbf;                 メッセージバッファアドレス
     B *name;                メッセージバッファ名へのポインタ
} T_CMBF;
```

メッセージバッファ状態パケット

```
typedef struct t_rmbf
{    ID stskid;             送信待ちタスクの ID
     ID rtskid;             受信待ちタスクの ID
     UINT smsgcnt;          メッセージバッファに入っているメッセージ数
     SIZE fmbfsz;           バッファの空きサイズ（バイト数）
} T_RMBF;
```

ランデブ用ポート生成情報パケット

```
typedef struct t_cpor
{    ATR poratr;            ランデブ用ポート属性
     UINT maxcmsz;          呼出メッセージ最大長
     UINT maxrmsz;          応答メッセージ最大長
     B *name;               ランデブ用ポート名へのポインタ
} T_CPOR;
```

ランデブ用ポート状態パケット

```
typedef struct t_rpor
{    ID ctskid;             呼出待ちタスクの ID
     ID atskid;             応答待ちタスクの ID
} T_RPOR;
```

ランデブ状態パケット

```
typedef struct t_rrdv
{    ID wtskid;             ランデブ終了待ちタスクの ID
} T_RRDV;
```

割込みハンドラ定義情報パケット

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

割込みサービスルーチン生成情報パケット

```
typedef struct t_cisr
{    ATR istatr;            割込みサービスルーチン属性
     VP_INT exinf;          拡張情報
     INTNO intno;           割込み番号
     FP isr;                割込みサービスルーチンのアドレス
     UINT imask;            割込みマスク
} T_CISR;
```

割込みサービスルーチン状態パケット

```
typedef struct t_risr
{    INTNO intno;          割込み番号
     UINT imask;           割込みマスク
} T_RISR;
```

可変長メモリプール生成情報パケット

```
typedef struct t_cmpl
{    ATR mplatr;           可変長メモリプール属性
     SIZE mplsz;           可変長メモリプールサイズ ( バイト )
     VP mpl;               可変長メモリプールアドレス
     B *name;              可変長メモリプール名へのポインタ
} T_CMPL;
```

可変長メモリプール状態パケット

```
typedef struct t_rmpl
{    ID wtskid;            獲得待ちタスクの ID
     SIZE fmplsz;          空き領域の合計サイズ ( バイト )
     UINT fblksz;          最大連続空き領域サイズ ( バイト )
} T_RMPL;
```

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;
```

アラームハンドラ生成情報パケット

```
typedef struct t_calm
{    ATR almatr;            アラームハンドラ属性
     VP_INT exinf;          拡張情報
     FP almhdr;             アラームハンドラ関数へのアドレス
} T_CALM;
```

アラームハンドラ状態パケット

```
typedef struct t_ralm
{    STAT almstat;          アラームハンドラ状態
     RELTIM lefttim;        起動すべき時刻までの時間
} T_RALM;
```

オーバーランハンドラ生成情報パケット

```
typedef struct t_dovr
{    ATR ovratr;            オーバーランハンドラ属性
     FP ovrhdr;             オーバーランハンドラ関数へのアドレス
     INTNO intno;           使用する割込み番号
     FP ovrclr;             割込み要因クリア関数へのポインタ
     UINT imask;            割込みマスク
} T_DOVR;
```

オーバーランハンドラ状態パケット

```
typedef struct t_rovr
{    STAT ovrstat;          オーバーランハンドラ状態
     OVRTIM leftotm;        タスク残り実行時間
} T_ROVR;
```

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 mbfid_max;             メッセージバッファ ID 上限
     ID porid_max;             ランデブ用ポート ID 上限
     ID mplid_max;             可変長メモリプール 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
     ID almno_max;             アラームハンドラ 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 almqsz;               アラームハンドラのタイマキューサイズ（バイト数）
     int istksz;               割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
     ID mtxid_max;             ミューテックス ID 上限
     ID isrid_max;             割込みサービスルーチン ID 上限
     ID svcfn_max;             拡張サービスコール機能番号上限
} T_RCFG;
```

拡張サービスコール定義情報

```
typedef struct t_dsvc
{    ATR svcatr;               拡張サービスコール属性
     FP svcrtn;                拡張サービスコールルーチンアドレス
     INT parn;                 拡張サービスコールルーチンのパラメータ数
} T_DSVC;
```

７．５ 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**
TA_TPRI	**0x0001**	**タスク優先度順**
TA_TPRIR	**0x0004**	**受信タスク優先度順（メッセージバッファ）**

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**	**最高優先度**

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**
TTS_WAS	**0x000c**	**WAITING-SUSPENDED 状態**
TTS_DMT	**0x0010**	**DORMANT 状態**

タスク例外処理状態

TTEX_ENA	**0x00**	**タスク例外処理許可**
TTEX_DIS	**0x01**	**タスク例外処理禁止**

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**
TTW_SMBF	**0x0100**	**メッセージバッファでメッセージ送信待ち**
TTW_MBF	**0x0200**	**メッセージバッファでメッセージ受信待ち**
TTW_CAL	**0x0400**	**ランデブ呼出待ち**
TTW_ACP	**0x0800**	**ランデブ受付け待ち**
TTW_RDV	**0x1000**	**ランデブ終了待ち**
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**

イベントフラグ属性

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

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**
TA_MPRI	**0x0002**	**メッセージ優先度順**

メッセージ優先度

TMIN_MPRI	**1**	**メッセージ最高優先度**

ミューテックス属性

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

ランデブ用ポート属性

TA_NULL	**0**	**属性特になし**

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**
TALM_STA	**0x0001**	**動作状態**

オーバーランハンドラ状態

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

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**

その他の定数

TRUE	**1**	**真**
FALSE	**0**	**偽**

７．６ NORTi3 互換モード

V3 マクロを定義することで NORTi3 互換モードで NORTi Version 4 を使用することが出来ます。norti3.h をインクルードすれば V3 マクロが定義されるので、NORTi3 形式のシステムコールが使用可能になります。ソースファイルに対する最小限の修正で NORTi3 から NORTi Version 4 に移行できます。

ただし、μITRON4.0 仕様との関係で以下の点が変更になっています。

・自タスクに対する強制終了 (ter_tsk) エラーコードは E_OBJ ではなく E_ILUSE です。

・自タスクに対する起床要求 (wup_tsk) はエラーにはなりません。キューイングされます。

・同時に複数タスク待ちを許さないイベントフラグに対して wai_flg で複数タスク待ちした場合のエラーコードは E_OBJ ではなく E_ILUSE です。

・自タスクに対する強制待ち要求 (sus_tsk) はディスパッチ禁止状態でなければエラーになりません。

・メールボックスで FIFO ではなく優先度つきメッセージキューを指定した場合の最大優先度はタスク優先度最大値と同一になります。

・オブジェクト生成情報の中で μITRON4.0 で削除された情報、たとえば拡張情報は指定しても無視されます。オブジェクト状態を参照するシステムコール (ref_xxx) ではこれらに対して NULL を返します。

・tcal_por のタイムアウトの考え方が変更になったため pcal_por は使用できません。また fwd_por におけるタイムアウトの意味も変更になっています。

・アラームハンドラの実行によってアラームハンドラは自動定義解除されません。

また、NORTi Version 4 実装上の理由から以下の点にご注意ください。

・自動 ID 割付は、使用可能 ID 番号の高い番号から順に割り当てられます。

・ID 番号 0 は、cre_yyy において自動 ID 割付と解釈されて処理されエラーにはなりません。

・NORTi3 型のオブジェクト生成情報 (T_Cxxx 型) は、NORTi4 型に変換されてシステムメモリにコピーされるのでシステムメモリ消費量が多くなります。

ＮＯＲＴｉ Version 4　ユーザーズガイド

カーネル編

2000 年　4 月　第 1 版
2000 年　5 月　第 2 版
2000 年 11 月　第 3 版
2002 年　4 月　第 4 版

株式会社ミスポ　http://www.mispo.co.jp/
〒 213-0012　川崎市高津区坂戸 3-2-1
TEL 044-829-3381　FAX 044-829-3382
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