User's Manual

M3T-MR30/4 V.4.01
ユーザーズマニュアル

M16C シリーズ,R8C ファミリ用リアルタイム OS

誤記に関するお詫び:
本資料 P.254 【割り込みベクタ定義】の記載事項に誤記があり、訂正いたしました。

ルネサス エレクトロニクス
www.renesas.com

Rev.1.00　2011.06

ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、当社は、一切その責任を負いません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
3. 当社製品を改造、改変、複製等しないでください。
4. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器の設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
5. 輸出に際しては、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。本資料に記載されている当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途の目的で使用しないでください。また、当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器に使用することができません。
6. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質水準を「標準水準」、「高品質水準」および「特定水準」に分類しております。また、各品質水準は、以下に示す用途に製品が使われることを意図しておりますので、当社製品の品質水準をご確認ください。お客様は、当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途に当社製品を使用することができません。また、お客様は、当社の文書による事前の承諾を得ることなく、意図されていない用途に当社製品を使用することができません。当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途または意図されていない用途に当社製品を使用したことによりお客様または第三者に生じた損害等に関し、当社は、一切その責任を負いません。なお、当社製品のデータ・シート、データ・ブック等の資料で特に品質水準の表示がない場合は、標準水準製品であることを表します。

   標準水準： コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット

   高品質水準： 輸送機器（自動車、電車、船舶等）、交通用信号機器、防災・防犯装置、各種安全装置、生命維持を目的として設計されていない医療機器（厚生労働省定義の管理医療機器に相当）

   特定水準： 航空機器、航空宇宙機器、海底中継機器、原子力制御システム、生命維持のための医療機器（生命維持装置、人体に埋め込み使用するもの、治療行為（患部切り出し等）を行うもの、その他直接人命に影響を与えるもの）（厚生労働省定義の高度管理医療機器に相当）またはシステム等
8. 本資料に記載された当社製品のご使用につき、特に、最大定格、動作電源電圧範囲、放熱特性、実装条件その他諸条件につきましては、当社保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めておりますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害などを生じさせないようお客様の責任において冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、機器またはシステムとしての出荷保証をお願いいたします。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様が製造された最終の機器・システムとしての安全検証をお願いいたします。
10. 当社製品の環境適合性等、詳細につきましては製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを固くお断りいたします。
12. 本資料に関する詳細についてのお問い合わせその他お気付きの点等がございましたら当社営業窓口までご照会ください。

注 1. 本資料において使用されている「当社」とは、ルネサス エレクトロニクス株式会社およびルネサス エレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

はじめに

M3T-MR30/4(以下MR30 と略す)はM16C/10,20,30,M16C/Tiny,R8C/Tinyシリーズ用のリアルタイム・オペレーティングシステム [1]でμITRON4.0 仕様 [2]に準拠しています。

■ MR30 を使うために必要なこと

MR30 を使用したプログラムを作成するには弊社下記製品を別途御購入して頂く必要があります。

- M16C/10, M16C/20, M16C/30, M16C/60, M16C/Tiny,R8C/Tiny シリーズ用 C コンパイラパッケージ
  M3T-NC30WA(以下 NC30WA と略す)

■ ドキュメント一覧

MR30 には以下の 2 種類のドキュメントが添付されています。

- リリースノート (紙面または PDF ファイル)
  MR30 を使用したプログラムの作成手順や作成上の注意事項を記載したドキュメントです。

- ユーザーズマニュアル (PDF ファイル)
  MR30 のサービスコールの使用方法や使用例、注意事項を記述したドキュメントです。
  本マニュアルを読む前に必ずリリースノートをお読みください。

■ ソフトウェアの使用権

ソフトウェアの使用権はソフトウェア使用許諾契約書に基づきます。本マニュアルによってソフトウェアの使用権の実施に対する保証及び使用権の実施の許諾を行うものではありません。

[1] 以降リアルタイム OS と略します。

[2] ・μITRON4.0 仕様は、(社)トロン協会が策定したオープンなリアルタイムカーネル仕様です。
・μITRON4.0 仕様の仕様書は、(社)トロン協会ホームページ（http://www.assoc.tron.org/）から入手が可能です。
・μITRON 仕様の著作権は(社)トロン協会に属しています。

目次

図目次

表目次

1. 本マニュアルの構成

本マニュアルは、以下の章から構成されています。

■ **2 概要**
MR30 の目的や、概略の機能、位置づけなどを説明します。

■ **3 カーネル入門**
MR30 を使用する上で必要となる考え方や用語などを説明します。

■ **4 カーネルの機能**
MR30 のカーネルの機能について説明します。

■ **5 サービスコールリファレンス**
MR30 でサポートしているサービスコールの説明をします。

■ **6 アプリケーション作成手順概要**
MR30 を使用してアプリケーションプログラムを作成する場合の開発手順の概要を説明します。

■ **7 アプリケーション作成手順詳細**
MR30 を使用してアプリケーションプログラムを作成する場合の開発手順を詳細に説明します。

■ **8 コンフィギュレータの使用方法**
コンフィギュレーションファイルの記述方法、および、コンフィギュレータの使用方法を詳細に説明します。

■ **10 サンプルプログラム**
製品にソースファイル形式で含まれている MR30 サンプルアプリケーションプログラムについて説明します。

■ **11 スタックサイズの算出方法**
システムスタック、ユーザスタックのスタックサイズの算出方法について説明します。

■ **12 注意事項**
MR30 を使用上の注意事項について説明します。

■ **13 別ROM化**
別 ROM 化の方法について説明します。

■ **14 付録**
パケット形式、アセンブリ言語インタフェースなどについて記述しています。

2. 概要

2.1 MR30 のねらい

近年マイクロコンピュータの急激な進歩にともない、マイクロコンピュータ応用製品の機能が複雑化してきています。これにともない、マイクロコンピュータのプログラムサイズが大きくなってきています。また製品開発競争が激化しマイクロコンピュータ応用製品を短期間に開発しなければなりません。すなわち、マイクロコンピュータのソフトウェアを開発している技術者は今までより大きなプログラムを今までより短期間で開発することが要求されてきます。そこでこの困難な要求を解決するためには以下のことを考えていかなければなりません。

1. ソフトウェアの再利用性を高めて、 開発すべきソフトウェアの量を削減する

このためにはソフトウェアをできるだけ機能単位で独立したモジュールに分割して再利用できるようにする方法があります。すなわち、汎用サブルーチン集などを多く蓄積してそれをプログラム開発時に使用します。ただこの方法では、時間やタイミングに依存したプログラムは再利用するのは困難です。ところが実際の応用プログラムは時間やタイミングに依存したプログラムがかなりの部分を占めていてこのような手法で再利用できるプログラムはあまり多くありません。

2. チームプログラミングを推進し、1 つのソフトウェアを何名かの技術者でおこなうようにする

チームプログラミングをおこなうには色々な問題があります。1 つはデバッグ作業をおこなうにあたり、チームプログラミングをおこなっている技術者全員のソフトウェアがデバッグできる状態にないとデバッグに入れません。また、チーム内の意志統一を十分におこなう必要があります。

3. ソフトウェアの生産効率を向上させ、技術者 1 名あたりの開発可能量を増加させる

1 つは技術者の教育をおこない技術者のスキルアップをはかる方法があります。また、構造化記述アセンブラや C コンパイラなどを用いることにより、より簡単にプログラムを作成できるようにする方法があります。また、ソフトウェアのモジュール化を推進してデバッグの効率を向上させる方法等があります。

しかし、このような問題を解決するには従来の手法では限界があります。そこでリアルタイム OS という新しい手法の導入が必要になってきます。そこで、弊社はこの要求に答えるべく 16 ビットマイクロコンピュータ M16C/10, M16C/20, M16C/30, M16C/60, M16C/Tiny,R8C/Tiny シリーズ用にリアルタイム OS MR30 を開発しました。MR30 を導入することにより以下のような効果があります。

4. ソフトウェアの再利用が容易になる

リアルタイム OS を導入することにより、リアルタイム OS を介してタイミングをとり、タイミングに依存したプログラムが再利用できるようになります。また、プログラムをタスクというモジュールに分割しますので、自然と構造化プログラミングをおこなうようになります。すなわち再利用可能なプログラムを自然に作成するようになります。

5. チームプログラミングがおこないやすくなる

リアルタイム OS を導入することにより、プログラムがタスクという機能単位のモジュールに分割されますので、タスク単位で開発をおこなう技術者を振り分け開発からタスク単位でデバッグまでできるようになります。とくにリアルタイム OS を導入すると、プログラムが全てでき上がっていなくてもタスクさえ出来ていればその部分のデバッグを初めることが容易にできます。またタスク単位で技術者を割り振ることができますので、作業分担が容易におこなえます。

6. ソフトウェアの独立性が高くなり、 プログラムをデバックしやすくなる

リアルタイム OS を導入することにより、プログラムをタスクという独立した小さなモジュールに分割できますので、プログラムをデバッグする際ほとんどはその小さなモジュールに着目するだけでデバッグすることができます。

7. タイマ制御が簡単になる

従来例えば、10ms ごとにある処理を動作させるためには、マイクロコンピュータのタイマ機能を用いて定期的に割り込みを発生させて処理させていました。ところが、マイクロコンピュータのタイマの数には限りがありますのでタイマが足らなくなったら 1 本のタイマを複数の処理に使用するなどの手法を用いて解決していました。しかし、リアルタイム OS を導入することにより、リアルタイム OS の時間管理機能を使用して一定時間毎にある処理をさせるというプログラムを、マイクロコンピュータのタイマ機能を特に意識せずに作成することができます。また、同時にプログラマから見たとき疑似的にマイクロコンピュータに無限本数のタイマが搭載されたようにプログラムを作成することができます。

8. ソフトウェアの保守性が向上する

リアルタイム OS を導入することにより開発したソフトウェアが小さなタスクと呼ばれるプログラムの集合で構成されます。これにより開発完了後保守をおこなう場合、小さなタスクだけを保守すればよくなり保守性が向上します。

9. ソフトウェアの信頼性が向上する

リアルタイム OS を導入することにより、プログラムの評価、試験などがタスクという小さなモジュール単位でおこなえますので評価、試験が容易になりひいては信頼性が向上します。

10. マイクロコンピュータの性能を最大限生かすことができ、これにより応用製品の性能向上が望める

リアルタイム OS を導入することにより、入出力待ちなどのマイクロコンピュータのむだな動作を減少させることができます。これによりマイクロコンピュータの能力を最大限に引き出すことができます。ひいては応用製品の性能向上につながります。

2.2 TRON仕様とMR30

MR30 は、μITRON 4.0 仕様に準拠した 16 ビットマイクロコンピュータ M16C/10, M16C/20, M16C/30, M16C/60, M16C/Tin,R8C/Tinyy シリーズ用に開発された、リアルタイム・オペレーティングシステムです。μITRON 4.0 仕様では、ソフトウェアの移植性を確保するための試みとしてスタンダードプロファイルを規定していますが、MR30 は、スタンダードプロファイルのうち、静的 API およびタスク例外を除くすべてのサービスコールをインプリメントしています。

2.3 MR30 の特長

MR30 は以下に示す特長を持っています。

1. μITRON 仕様に準拠したリアルタイム・オペレーティングシステム

MR30 はμITRON 仕様に基づいて開発されました。μITRON 教科書として出版されている文献やμITRON セミナー等で得た知識をほとんどそのまま役立てることができます。また、他のμITRON 仕様に準拠したリアルタイム OS を用いて開発したアプリケーションプログラムを MR30 に移行するのは比較的容易に行えます。

2. 高速処理を実現

M16C マイクロコンピュータのアーキテクチャを活用し、高速処理を実現しています。

3. 必要モジュールのみを自動選択することにより常に最小サイズのシステムを構築

MR30 は M16C/10, M16C/20, M16C/30, M16C/60, M16C/Tiny,R8C/Tiny シリーズオブジェクトライブラリ形式で供給されています。したがって、リンケージエディタ LN30 のもつ機能により、数ある MR30 の機能モジュールのなかで使用しているモジュールのみを自動選択してシステムを生成します。このため、常に最小サイズのシステムが自動的に生成されます。

4. C コンパイラ NC30WA を用いて C 言語でアプリケーションプログラムが開発可能

C コンパイラ NC30WA を用いて、MR30 のアプリケーションプログラムを C 言語で開発できます。また C 言語から MR30 の機能を呼び出すためのインタフェースライブラリが製品に添付されています。

5. 統合環境を利用して効率のよい開発が可能

ルネサス統合環境 High-performance Embedded Workshop を使用して、他の High-performance Embedded Workshop 対応のルネサス開発ツールと共通した操作での開発が可能

6. 上流工程ツール "コンフィギュレータ" により、容易に開発が可能

ROM 書き込み形式ファイルまでの作成を簡単な定義のみでおこなえるコンフィギュレータを装備しています。これにより、どんなライブラリを結合する必要があるかなどを特に気にする必要はありません。また、M3T-MR30/4 V.4.00 Release 00 より、GUI 化されたコンフィギュレータを用意しました。これにより、コンフィギュレーションファイルの記述形式を習得しなくとも、容易にコンフィギュレーションが可能になりました。

3. カーネル入門

3.1 リアルタイムOSの考え方

本節では、リアルタイム OS の基本概念について説明します。

3.1.1 リアルタイムOSの必要性

近年半導体技術の進歩にともなってシングルチップマイクロコンピュータ (マイコン)のROM容量が増大してきています。このような大ROM容量のマイクロコンピュータの出現によりそのプログラム開発が従来の方法では困難になってきています。図 3.1にプログラムサイズと開発期間 (開発の困難さ) との関係を示します。この 図 3.1はあくまでイメージ図ですが、プログラムのサイズが大きくなるに従い開発期間が指数関数的に長くなってきます。例えば 32Kバイトのプログラムを 1 個開発するより、8Kバイトのプログラムを 4 個開発する方が簡単です。

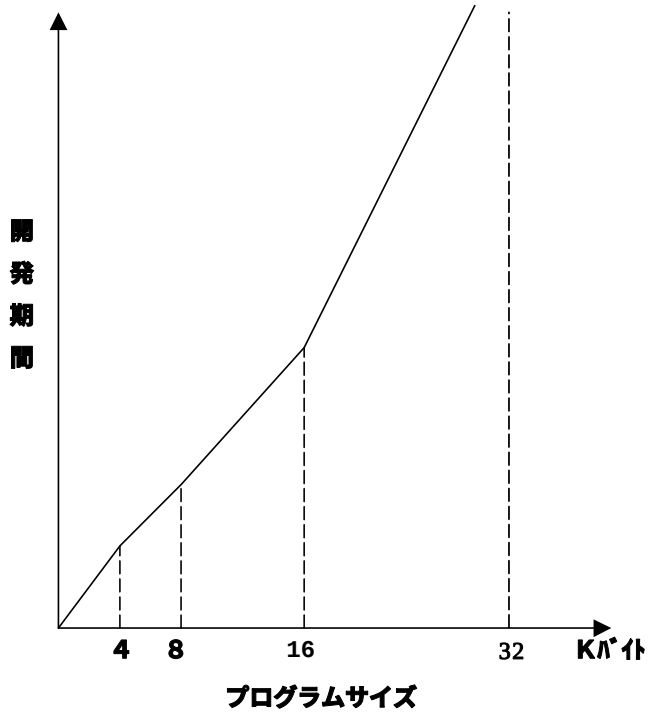


図 3.1 プログラムサイズと開発期間

そこで大きなプログラムを短期間に簡単に開発するための手法が必要になってきます。この方法として小さな ROM 容量のマイクロコンピュータを多く使う方法があります。たとえば、図 3.2 にオーディオ機器システムを複数のマイクロコンピュータで構成した例を示します。

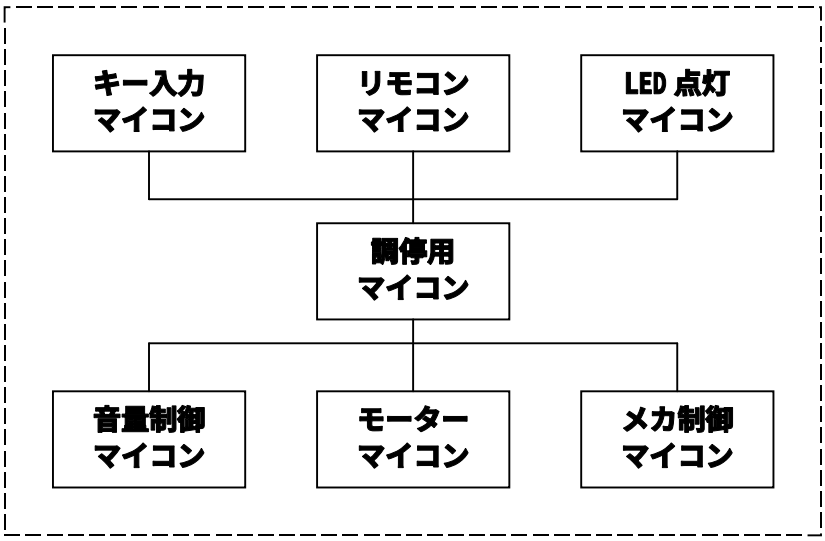


図 3.2 マイコンを多く使ったシステム例 (オーディオ機器)

このように機能単位で別々のマイクロコンピュータを用いることは以下の利点があります。

1. ひとつひとつのプログラムが小さくなり、プログラム開発が容易になる
2. 一度開発したソフトウェアを再利用することが非常に容易になる
3. 完全に機能ごとにプログラムが分離するので複数の技術者でプログラム開発が容易にできる

この反面以下のような欠点があります。

1. 部品点数が多くなり製品の原価を上昇させる
2. ハードウェア設計が複雑になる
3. 製品の物理的サイズが大きくなる

そこでそれぞれのマイクロコンピュータで動作しているプログラムを、1 つのマイクロコンピュータでソフトウェア的に、別々のマイクロコンピュータで動作しているように見せることのできるリアルタイムOSを採用すれば、上記の利点を残したままで欠点をすべて無くすことができます。図 3.3に、図 3.2に示したシステムにリアルタイムOSを導入した場合のシステム例を示します。

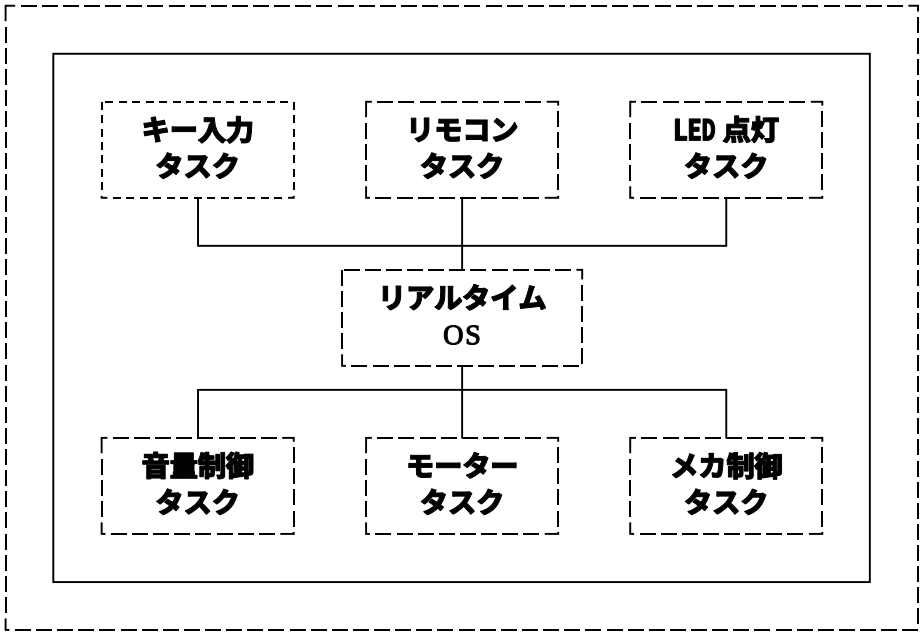


図 3.3 リアルタイム OS の導入システム例 (オーディオ機器)

すなわちリアルタイム OS とは 1 個のマイクロコンピュータを、あたかも複数のマイクロコンピュータが動作しているように見せるソフトウェアです。複数のマイクロコンピュータに相当するひとつひとつのプログラムをリアルタイム OS 用語でタスクと呼びます。

3.1.2 カーネルの動作原理

カーネルとは、リアルタイム OS の中核となるプログラムのことです。カーネルは、1 個のマイクロコンピュータを、あたかも複数のマイクロコンピュータが動作しているように見せることのできるソフトウェアです。では 1 個のマイクロコンピュータをどのようにして複数あるように見せかけるのでしょうか？

それは、図 3.4に示すようにそれぞれのタスクを時分割で動作させるからです。つまり実行するタスクを一定時間ごとに切り替えて、複数のタスクが同時に実行しているように見せるのです。

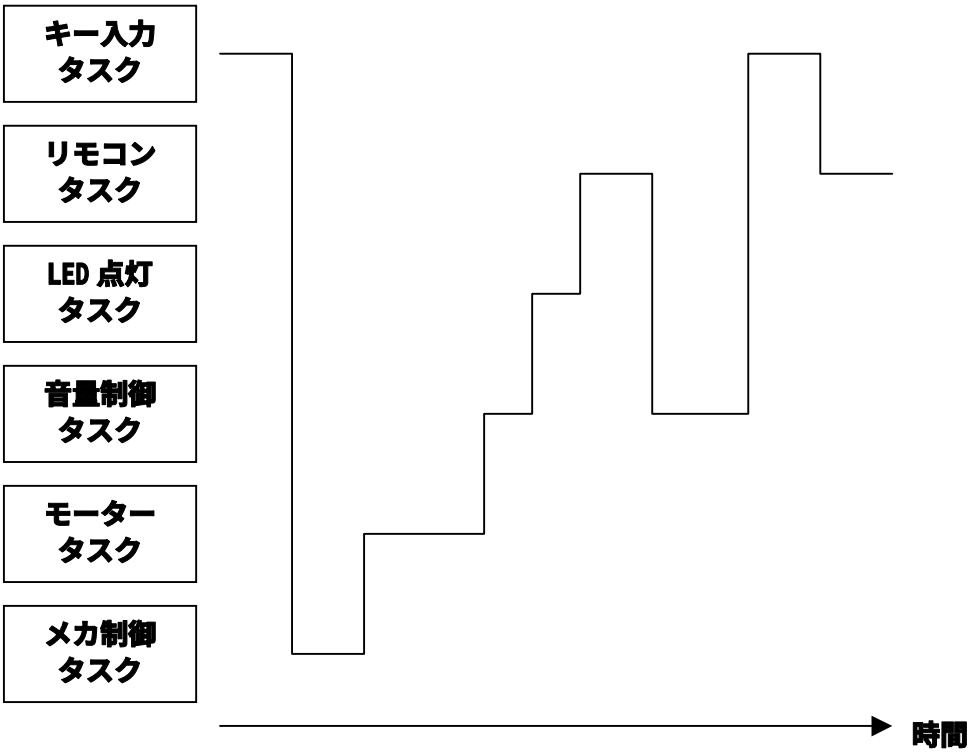


図 3.4 タスクの時分割動作

このようにタスクを一定時間ごとに切り替えて実行しています。このタスクを切り替えることをディスパッチと呼びます。タスク切り替え (ディスパッチ)が発生する要因として以下のものがあります。

- 自分自身で切り替えを要求する
- 割り込みなどの外的要因で切り替わる

タスク切り替えが発生し、再度、そのタスクを実行するときには、中断していたところから再開します。 (図 3.5参照)

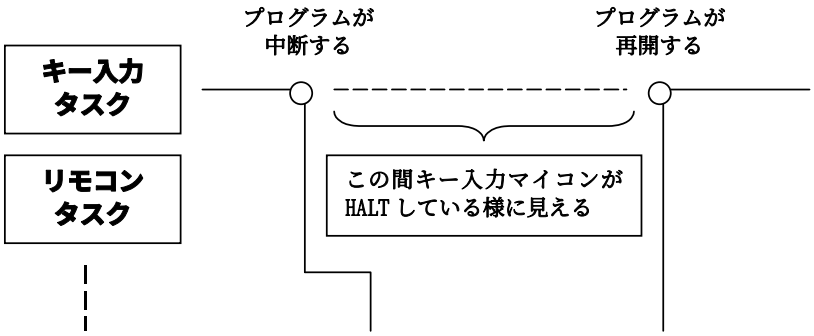


図 3.5 タスクの中断と再開

図 3.5においてキー入力タスクは、他のタスクに実行制御が移っている間、プログラマから見ればプログラムが中断しそのマイコンがHALT しているようにみえます。カーネルは、中断した時点のレジスタ内容を復帰することにより、タスクを中断した時点の状態から再開させます。すなわちタスクの切り替えとは、現在実行中のタスクのレジスタの内容をそのタス
クを管理するメモリ領域に退避し、切り替えるタスクのレジスタ内容を復帰することです(図 3.6参照)。

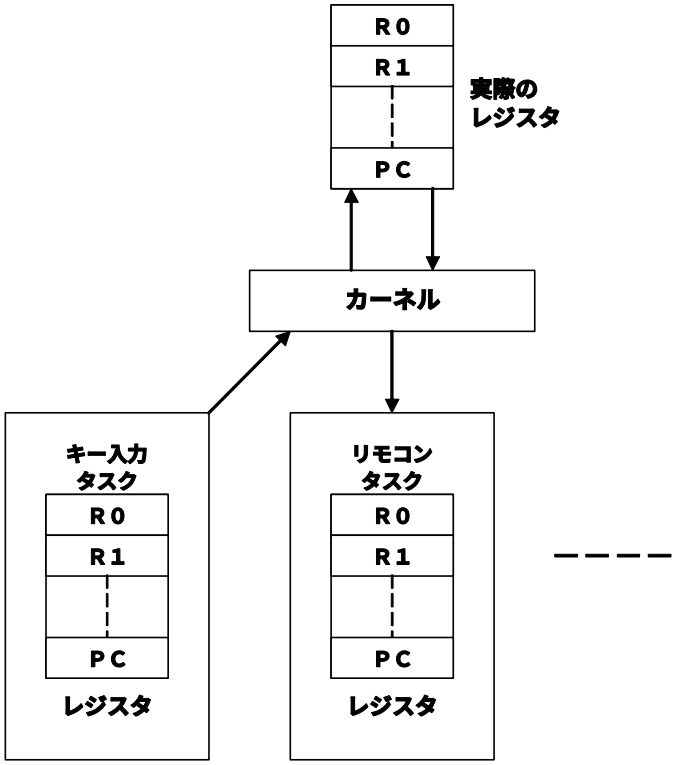


図 3.6 タスクの切り替え

図 3.7[3]は各タスクのレジスタをどのように管理しているか具体的に示したものです。 実際にはタスクごとに持つ必要のあるのはレジスタだけでなく、スタック領域もタスクごとに持つ必要があります。

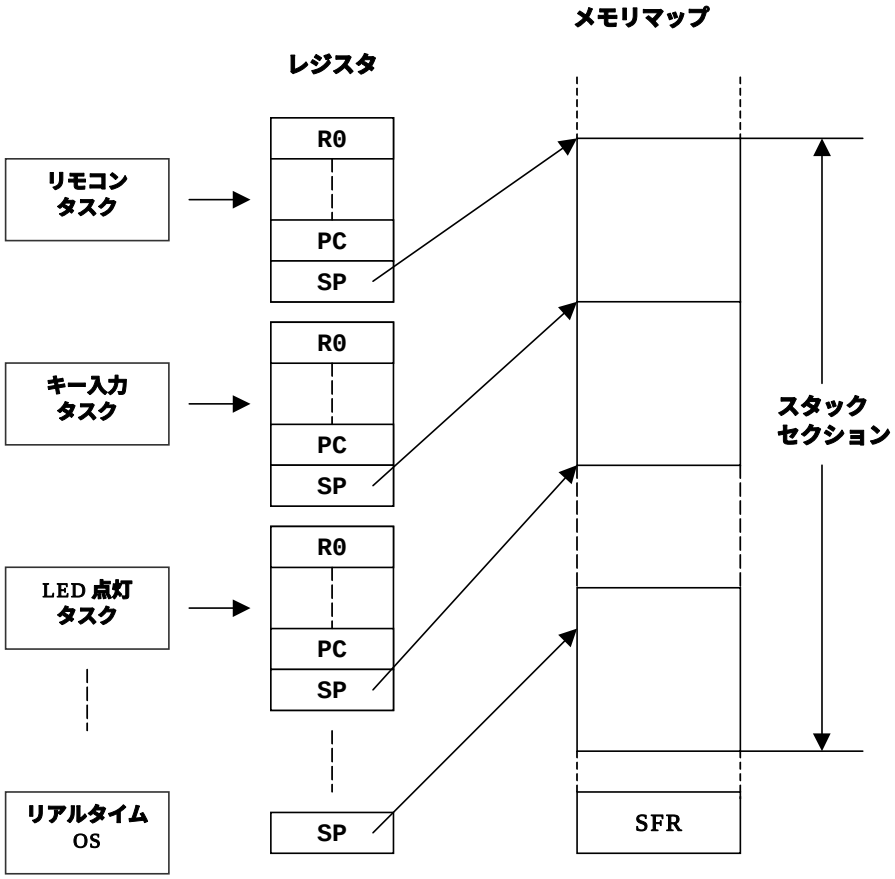


図 3.7 タスクのレジスタ領域

[3] 本バージョンより、タスクのスタック領域はセクション毎に分割することが可能となりましたが、この図は、タスクのスタック領域をすべて同じセクションに配置した場合の図です。

図 3.8 は各タスクのレジスタおよびスタック領域を詳細に説明したものです。MR30 では各タスクのレジスタは図 3.8 に示すようにスタック領域の中に格納され管理されています。図 3.8 は、レジスタ格納後の状態を示しています。

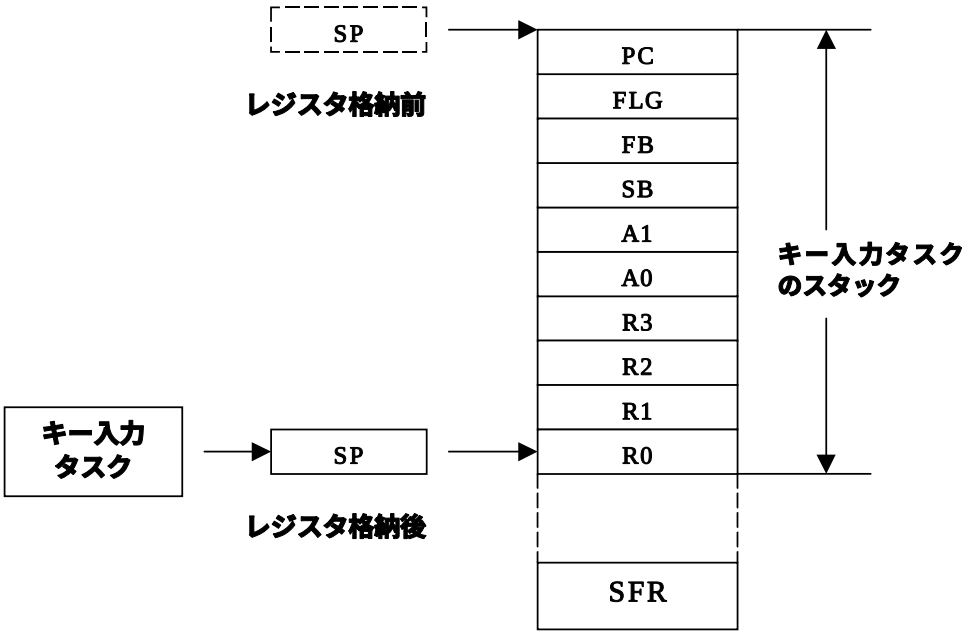


図 3.8 実際のレジスタとスタック領域の管理

3.2 サービスコール

プログラマはプログラム中でどのようにカーネルの機能を使用するのでしょうか? これにはカーネルの機能をプログラムから何らかの形で呼び出すことが必要です。このカーネルの機能を呼び出すことをサービスコールといいます。すなわちサービスコールにより、タスクの起動などの処理を行なうことができます (図 3.9 参照)。

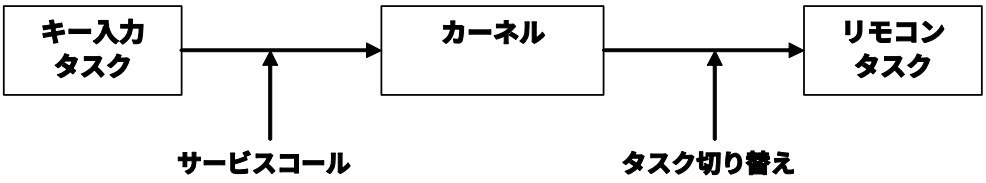


図 3.9 サービスコール

このサービスコールは、以下のように C 言語で応用プログラムを記述する場合は関数呼び出しで実現します。

```
act_tsk(ID_main);
```

またアセンブリ言語で応用プログラムを記述する場合は以下のようにアセンブラマクロ呼び出しにより実現します。

```
act_tsk #ID_main
```

3.2.1 サービスコール処理

サービスコールが発行されると以下の手順により処理がおこなわれます。

1. 現レジスタ内容を退避します
2. スタックポインタをタスクのものからリアルタイム OS(システム)のものへ切り替えます
3. サービスコール要求にしたがった処理を行います
4. 次に実行するタスクの選択をおこないます
5. スタックポインタをタスクのものに切り替えます
6. レジスタ内容を復帰してタスクの実行を再開します

サービスコールが発生してからタスク切り替えまでの処理の流れを図 3.10 に示します。

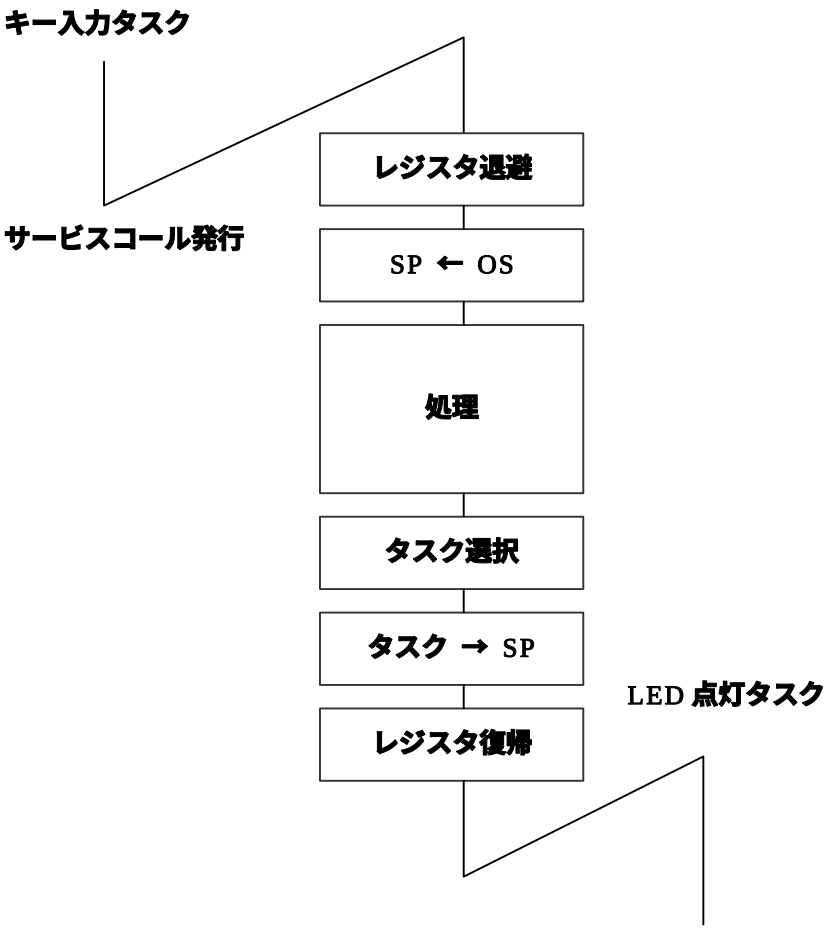


図 3.10 サービスコールの処理の流れ

3.2.2 ハンドラからのサービスコールの処理手順

ハンドラ[4]からのサービスコール発行はタスクからのサービスコールと異なり、サービスコール発行時にタスク切り替えは発生しません。タスク切り替えが発生するのはハンドラからの復帰時です。ハンドラからのサービスコール処理手順は大きく分けて以下の3通りがあります。

1. タスク実行中に割り込んだハンドラからのサービスコール
2. サービスコール処理中に割り込んだハンドラからのサービスコール
3. ハンドラ実行中に割り込んだ (多重割り込み) ハンドラからのサービスコール

タスク実行中に割り込んだハンドラからのサービスコール

スケジューリング (タスク切り替え)はret_intサービスコールによりおこなわれます。[5] (図 3.11参照)

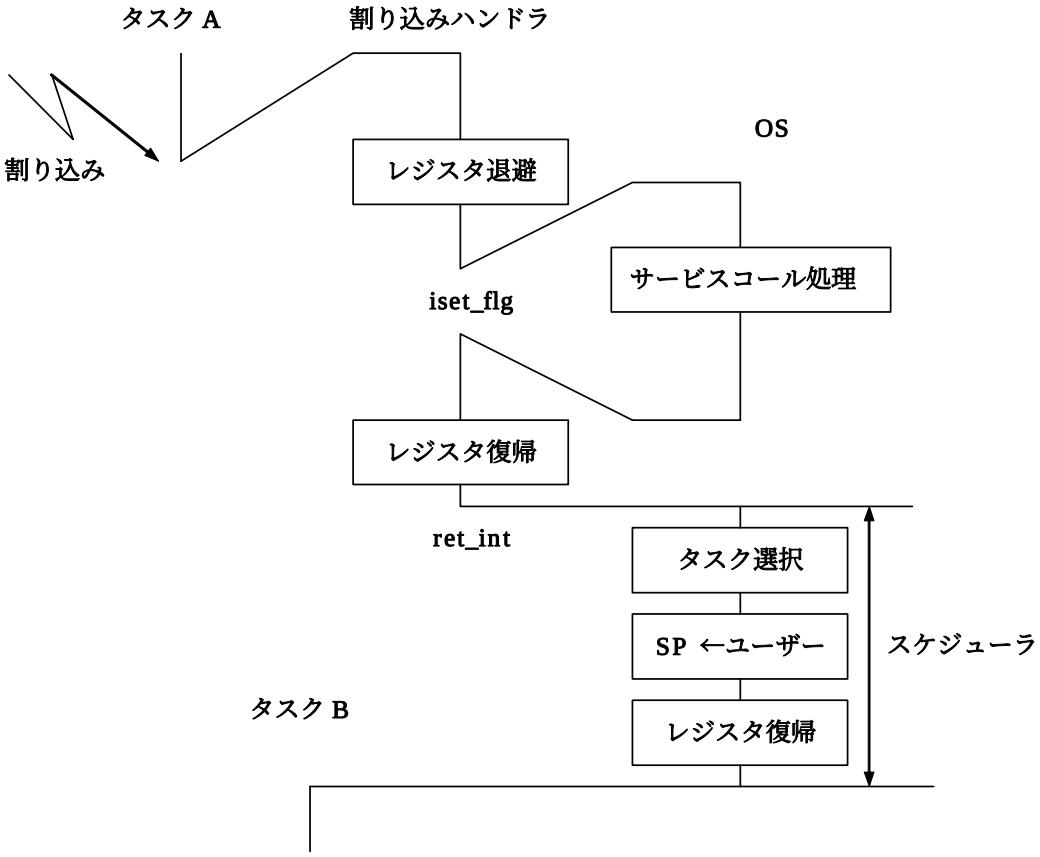


図 3.11 タスク実行中に割り込んだ割り込みハンドラからのサービスコール処理手順

[4] カーネル管理外(OS 独立)割り込みハンドラからはサービスコールは発行できませんので、ここで述べているハンドラはカーネル管理外(OS 独立)割り込みハンドラを含みません。

[5] C 言語でカーネル管理(OS 依存)割り込みハンドラを記述する場合（#pragma INTHANDLER 指定時)、ret_int サービスコールは自動的に発行されます。

サービスコール処理中に割り込んだハンドラからのサービスコール

スケジューリング (タスク切り替え)は割り込まれたサービスコール処理に戻った後におこなわれます。 (図 3.12参照)

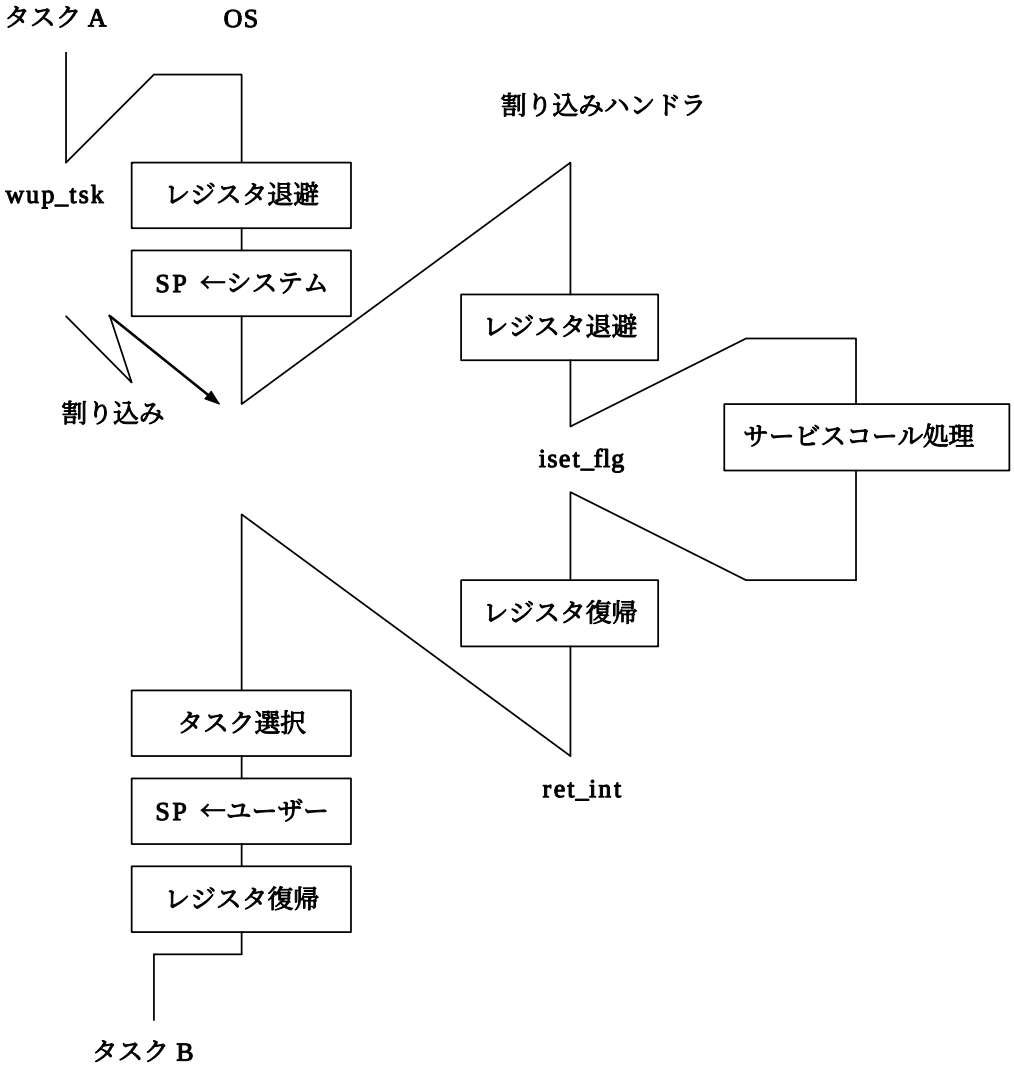


図 3.12 サービスコール処理中に割り込んだ割り込みハンドラからのサービスコール処理手順

ハンドラ実行中に割り込んだハンドラからのサービスコール

ハンドラ (以後ハンドラAと呼びます。)実行中に割り込みが発生した場合を考えます。ハンドラA実行中に割り込んだハンドラ (以後ハンドラBと呼びます。)が、発行したサービスコールによりタスク切り替えが必要になった場合は、ハンドラBから復帰するサービスコール (ret_intサービスコール)では、ハンドラAに戻るだけでタスク切り替えはおこないません。ハンドラAからのret_intサービスコールによりタスク切り替えが行われます。 (図 3.13参照)

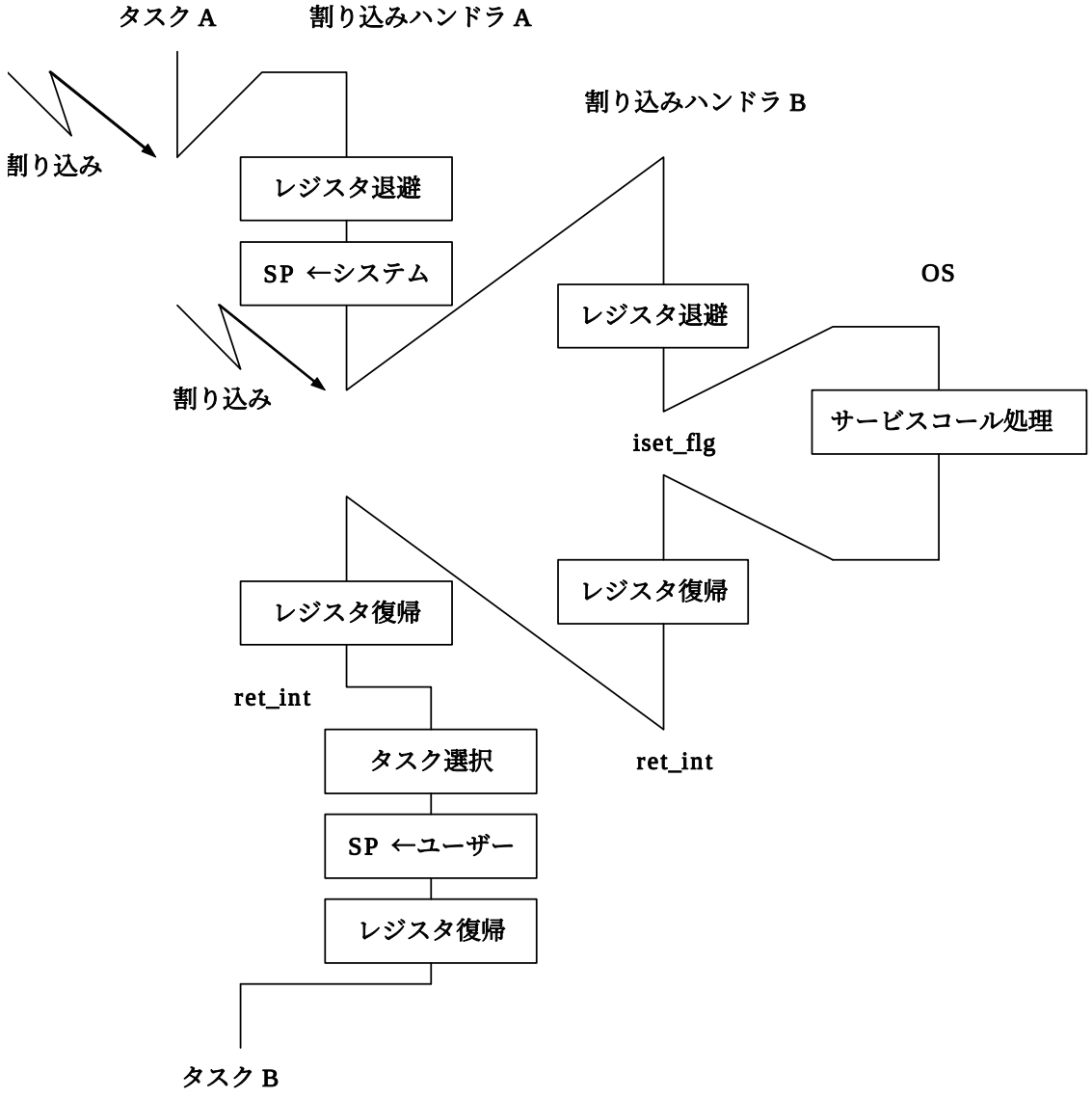


図 3.13 多重割り込みハンドラからのサービスコール処理手順

3.3 オブジェクト

タスクやセマフォなど、サービスコールによって操作する対象を「オブジェクト」と呼びます。オブジェクトは ID 番号によって識別されます。

3.3.1 サービスコールにおけるオブジェクトの指定方法

各オブジェクトの識別は、MR30 の内部では ID 番号でおこないます。 すなわち、"タスク ID 番号 1 のタスクを起動する"などというように管理されています。しかし、プログラム中にタスクの番号を直接書き込むと非常に可読性の低いプログラムになってしまいます。たとえば、

```
act_tsk(1);
```

とプログラム中に記述するとプログラマは絶えず ID 番号の 2 番のタスクは何かを知っている必要があります。また、他人がこのプログラムを見たときに ID 番号の 2 番のタスクが何かが一目では分かりません。そこで MR30 ではタスクの識別をそのタスクの名前 (関数もしくはシンボルの名前) で指定し、その名前からタスクの ID 番号への変換を MR30 に付属しているプログラム"コンフィギュレータ cfg30"が自動的におこないます。具体的には、コンフィギュレータは、各タスクと ID 番号が対応づけられるように下のように定義されたヘッダファイル(kernel_id.h)を出力します。

```
#define ID_TASK1 1
```

図 3.14は、タスクを識別する様子を示したものです。

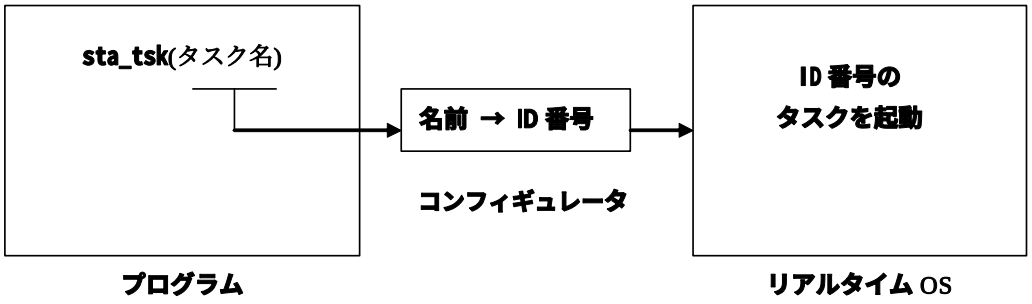


図 3.14 タスクの識別

この定義を用いると、先の例は以下のように記述できます。

```
act_tsk(ID_TASK1); /* タスクID が"ID_TASK1"のタスクを起動する */
```

この例では、"ID_ TASK1"に対応するタスクを起動するように指定しています。タスクの名前から ID 番号への変換は、プログラムを生成するときにコンパイラの機能を使用することによって行うため、この機能による処理速度の低下はありません。

3.4 タスク

本節ではタスクを MR30 がどのように管理しているかを説明します。

3.4.1 タスクの状態

リアルタイムOSではタスクを実行するべきか否かを、タスクの状態を管理することにより制御しています。例えば、図 3.15にキー入力タスクの実行制御と状態の関係を示します。キー入力が発生した場合はそのタスクを実行しなければなりません。すなわち、キー入力タスクが実行状態となります。またキー入力を待っているときはタスクを実行する必要はありません。すなわち、キー入力タスクは待ち状態になっています。

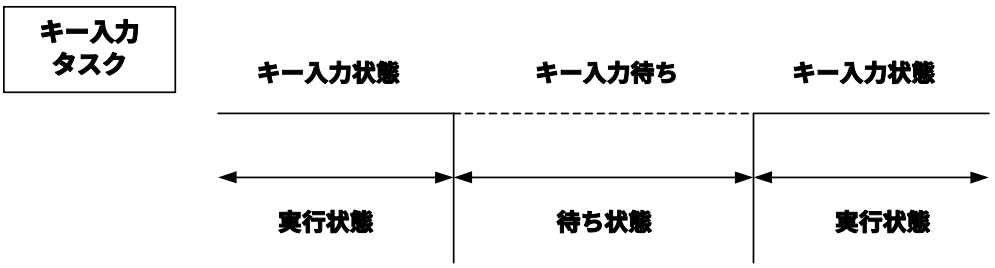


図 3.15 タスクの状態

MR30 では実行状態、待ち状態を含め以下の 6 つの状態を管理しています。

1. 実行状態 (RUNNING 状態)
2. 実行可能状態 (READY 状態)
3. 待ち状態 (WAITING 状態)
4. 強制待ち状態 (SUSPENDED 状態)
5. 二重待ち状態 (WAITING-SUSPENDED 状態)
6. 休止状態 (DORMANT 状態)

タスクは上記の 6 つの状態を遷移していきます。図 3.16に、タスクの状態遷移図を示します。

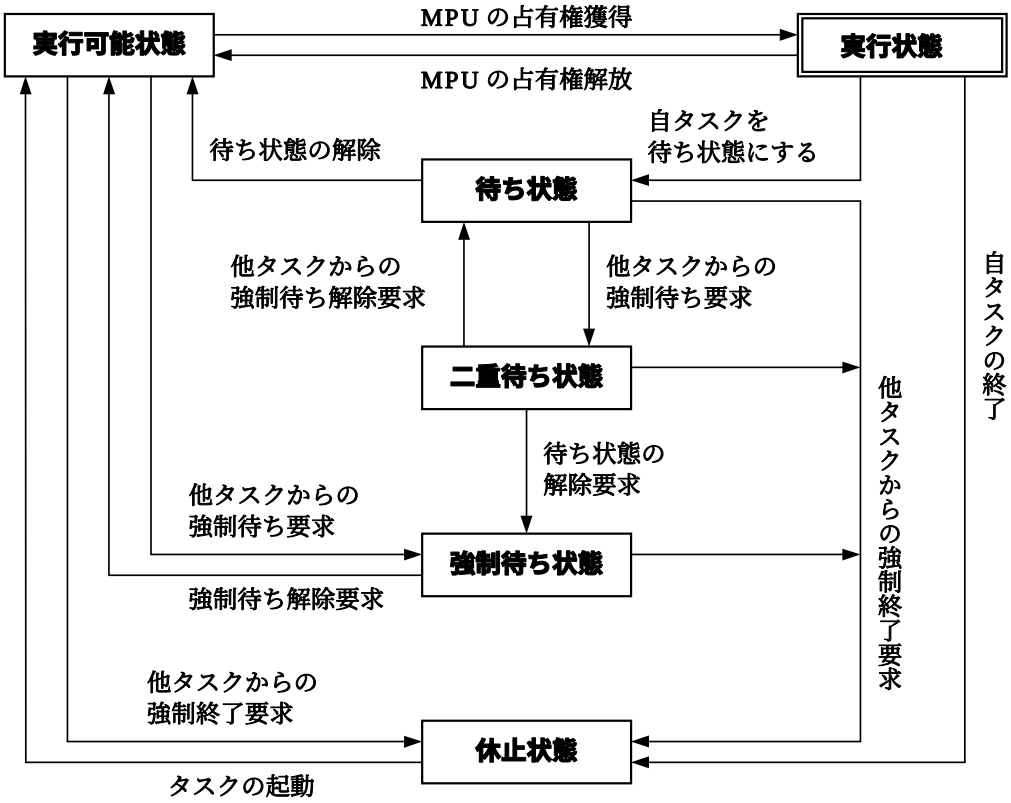


図 3.16 MR30 のタスク状態遷移図

1. 実行状態 (RUNNING 状態)

タスクが、まさに現在実行中の状態を実行状態といいます。マイクロコンピュータは 1 つしかないのですから当然実行状態にあるのは常に 1 つだけです。
現在実行状態のタスクが他の状態に移行するには、以下の事象のうちどれかが発生した場合です。

- ext_tsk サービスコールにより、自分で自タスクを正常終了させた場合
- 自分で待ち状態に入った場合 [6]
- 自タスクから発行したサービスコールにより、自タスクより優先度の高い他のタスクの待ち状態が解除された場合
- 割り込み等の事象の発生により、その割り込みハンドラから発行されたサービスコールによって自タスクより優先度の高いタスクが実行可能状態になった場合
- chg_pri, ichg_pri サービスコールによってタスクの優先度を変更し、他の実行可能状態のタスクが自タスクより優先度が高くなった場合
- rot_rdq, irot_rdq サービスコールにより、自タスク優先度のレディキューを回転し、実行権を放棄した場合

[6] dly_tsk, slp_tsk, tslp_tsk, wai_flg, twai_flg, wai_sem, twai_sem, rcv_mbx, trcv_mbx,snd_dtq,tsnd_dtq,rcv_dtq, trcv_dtq, vtsnd_dtq, vsnd_dtq,vtrcv_dtq,vrcv_dtq, get_mpf, tget_mpf サービスコールによる

上記の事象が発生すると再スケジュールされて実行状態と実行可能状態にあるタスクのなかで最も優先度の高いタスクが実行状態に移され、そのタスクのプログラムが実行されます。

2. 実行可能状態 (READY 状態)

タスクが実行される条件は整っているが、そのタスクより優先度の高いタスクもしくは同一優先度のタスクが実行されているために実行できずに実行待ち状態になっている状態を実行可能状態といいます。
実行可能状態であるタスクで、レディキュー では 2 番目に実行される可能性のあるタスクが実行状態になるのは、以下の事象の内いずれかが発生した場合です。

- 実行状態のタスクが ext_tsk サービスコールによって正常終了した場合
- 実行状態のタスクが自分で待ち状態に入った場合 [7]
- 実行状態のタスクが chg_pri サービスコールによりタスク優先度を変更し、実行可能状態のタスクが実行状態のタスクより優先度が高くなった場合
- 割り込み等の事象の発生により、その割り込みハンドラから発行されたサービスコールによって実行中のタスクより優先度の高いタスクが実行可能状態になった場合
- rot_rdq, irot_rdq サービスコールにより、レディキュー先頭タスクが、自タスク優先度のレディキューを回転し、実行権を放棄した場合

3. 待ち状態 (WAITING 状態)

実行状態のタスクが自分自身を待ち状態に移行させる要求を出すことにより、タスクは実行状態から待ち状態に移行することができます。待ち状態は通常入出力装置の入出力動作完了待ちや他のタスクの処理待ちなどの状態として使用されます。実行待ち状態に移行するには以下の方法があります。

- slp_tsk サービスコールにより単純に待ち状態に移行します。この場合、他のタスクから明示的に待ち状態から解除されないと実行可能状態に移行しません。
- dly_tsk サービスコールにより一定時間待ち状態に移行します。この場合、指定時間経過するかもしくは他のタスクから明示的に待ち状態を解除することにより実行可能状態に移行します。
- wai_flg、wai_sem、rcv_mbx、snd_dtq、rcv_dtq、vsnd_dtq、vrcv_dtq、get_mpf サービスコールにより要求待ちで待ち状態に移行します。この場合、要求事項が満たされるかもしくは他のタスクから明示的に待ち状態を解除することにより実行可能状態に移行します。
- tslp_tsk、twai_flg、twai_sem、trcv_mbx、tsnd_dtq、trcv_dtq、vtsnd_dtq、vtrcv_dtq、tget_mpf サービスコールは、slp_tsk、wai_flg、wai_sem、rcv_mbx、tsnd_dtq、trcv_dtq、vsnd_dtq、vrcv_dtq、get_mpf サービスコールにタイムアウトを指定したサービスコールです。各サービスコールの要求待ちで待ち状態に移行します。この場合、要求事項が満たされるかもしくは、指定時間が経過した場合、実行可能状態に移行します。
- タスクが wai_flg、wai_sem、rcv_mbx、snd_dtq、rcv_dtq、vsnd_dtq、vrcv_dtq、get_mpf、tslp_tsk、twai_flg、twai_sem、trcv_mbx、tsnd_dtq、trcv_dtq、vtsnd_dtq、vtrcv_dtq、tget_mpf サービスコールにより要求待ちで待ち状態になると、その要求事項により次の待ち行列のいずれかにつながります。

[7] dly_tsk, slp_tsk, tslp_tsk, wai_flg, twai_flg, wai_sem, twai_sem, rcv_mbx, trcv_mbx,snd_dtq,tsnd_dtq,rcv_dtq, trcv_dtq, vtsnd_dtq, vsnd_dtq,vtrcv_dtq,vrcv_dtq, get_mpf, tget_mpf サービスコールによる

イベントフラグ待ち行列
セマフォ待ち行列
メールボックスメッセージ受信待ち行列
データキューデータ送信待ち行列
データキューデータ受信待ち行列
long データキューデータ送信待ち行列
long データキューデータ受信待ち行列
固定長メモリプールメモリ獲得待ち行列

4. 強制待ち状態 (SUSPENDED 状態)

実行状態のタスクから sus_tsk サービスコールが発行される、もしくはハンドラから isus_tsk サービスコールが発行されると、サービスコールにより指定された実行可能なタスクもしくは実行中のタスクは強制待ち状態になります。なお待ち状態のタスクが指定された場合は二重待ち状態になります。

強制待ち状態は入出力エラー等の発生により実行可能なタスクもしくは実行中のタスク[8] が処理を一時的に中断させるためにスケジューリングから外された状態です。すなわち実行可能状態のタスクに対して強制待ち要求が出された場合、そのタスクは実行待ち行列から外されます。

なお、強制待ち要求のキューイングは行いません。したがって強制待ち要求は実行状態、実行可能状態、待ち状態にあるタスク[9]にのみ行えます。すでに強制待ち状態にあるタスクに強制待ち要求した場合には、エラーコードが返されます。

5. 二重待ち状態 (WAITING-SUSPENDED 状態)

待ち状態にあるタスクに強制待ちの要求が出された場合、タスクは二重待ち状態になります。slp_tsk、dly_tsk、wai_flg、wai_sem、rcv_mbx、snd_dtq、rcv_dtq、vsnd_dtq、vrcv_dtq、get_mpf、tslp_tsk、twai_flg、twai_sem、trcv_mbx、tsnd_dtq、trcv_dtq、vtsnd_dtq、vtrcv_dtq、tget_mpf サービスコールによる要求待ちで待ち状態にあるタスクに対して強制待ち要求が出された場合、そのタスクは二重待ち状態に移行します。

また、二重待ち状態のタスクは待ち条件が解除されると強制待ち状態になります。待ち条件が解除されるには以下の場合が考えられます。

- wup_tsk、iwup_tsk サービスコールにより起床する場合
- dly_tsk、tslp_tsk サービスコールにより待ち状態になったタスクが時間経過により起床される場合
- wai_flg、wai_sem、rcv_mbx、snd_dtq、rcv_dtq、vsnd_dtq、vrcv_dtq、get_mpf、tslp_tsk、twai_flg、twai_sem、trcv_mbx、tsnd_dtq、trcv_dtq、vtsnd_dtq、vtrcv_dtq、tget_mpf サービスコールにより待ち状態になったタスクの要求が満たされた、もしくは、指定時間が経過した場合
- rel_wai、irel_wai サービスコールにより待ち状態が強制解除される場合

二重待ち状態のタスクに rsm_tsk, irsm_tsk サービスコールによる強制待ち解除要求がだされると待ち状態になります。なお、強制待ち状態にあるタスクが自分自身を待ち状態にする要求は出せないため、強制待ち状態から二重待ち状態への移行は発生しません。

[8] ハンドラから isus_tsk サービスコールにより実行タスクを強制待ち状態にする場合は、実行状態から直接強制待ち状態に移行されます。例外的にこの場合のみ実行状態から強制待ち状態に移行する場合はあることに注意してください。
[9] 待ち状態にあるタスクに対して強制待ち要求をおこなうと二重待ち状態になります。

6. 休止状態 (DORMANT 状態)

通常は、MR30 システムに登録されているが起動していない状態です。この状態になるには以下の 2 つの場合があります。

- タスクが起動をかけられるのを待っている場合
- ext_tsk サービスコールにより、タスクが正常終了 もしくは ter_tsk サービスコールにより、強制終了した場合

3.4.2 タスクの優先度とレディキュー

リアルタイム OS では実行したいタスクが同時にいくつも発生することがあります。 このときにどのタスクを実行するかを判断することが必要になります。そこでタスクに実行の優先度をつけ、優先度の高いタスクから実行するようにします。すなわち、処理を素早くおこなう必要のあるタスクの優先度を高くしておけば実行したいときに素早く実行することができるようになります。

MR30 では同一の優先度を複数のタスクに与えることができます。そこで、実行可能になったタスクの実行順を制御するためにタスクの待ち行列 (レディキュー)を生成します。図 3.17にレディキューの構造を示します。レディキューは優先度ごとに管理され、タスクが接続されている最も優先度の高い待ち行列の先頭タスクを実行状態にします。[10]

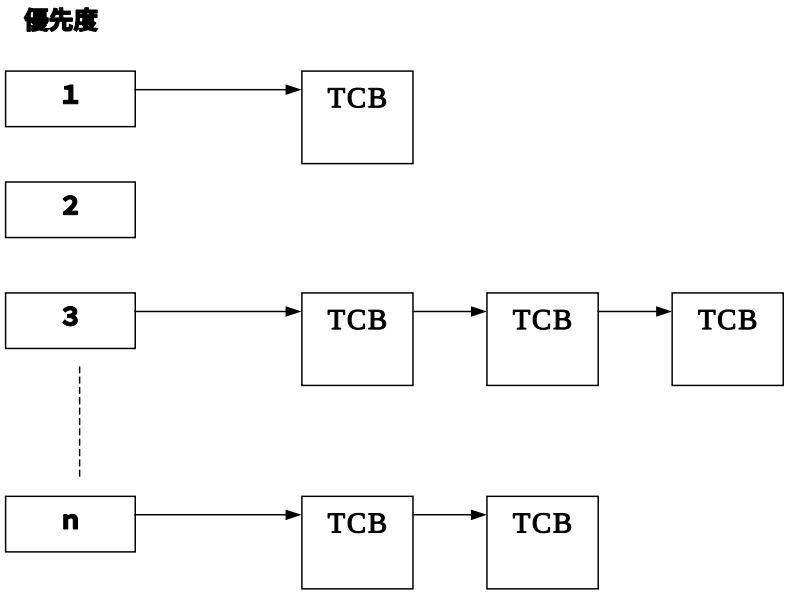


図 3.17 レディキュー (実行待ち状態)

[10] 実行状態のタスクはレディキューにつながれたままです。

3.4.3 タスクの優先度と待ち行列

μITRON 4.0 仕様のスタンダードプロファイルでは、各オブジェクトの待ち方にタスクの優先度順に待ち行列をつなぐ(TA_TPRI 属性)、FIFO 順に待ち行列につなぐ(TA_TFIFO)の 2 種類をサポートすることになっています。MR30 でもこの 2 種類の待ち方をサポートしています。
図 3.18、図 3.19にタスクが、"taskD"、"taskC"、"taskA"、"taskB"の順で待ち行列につながれたときの様子を示します。

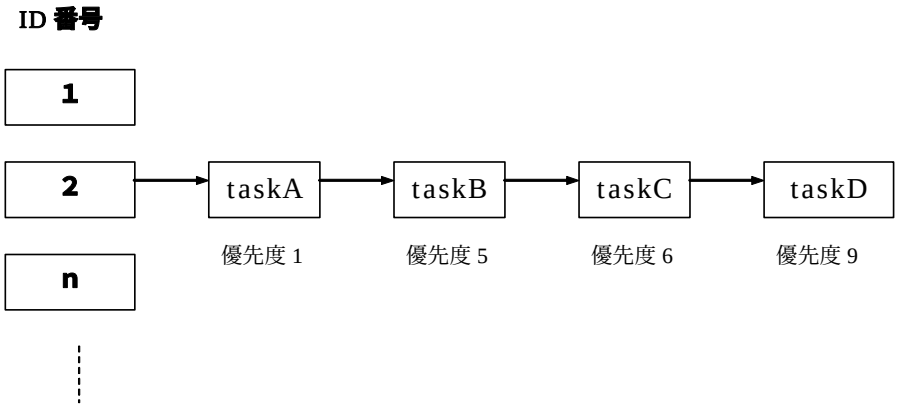


図 3.18 TA_TPRI 属性の待ち行列

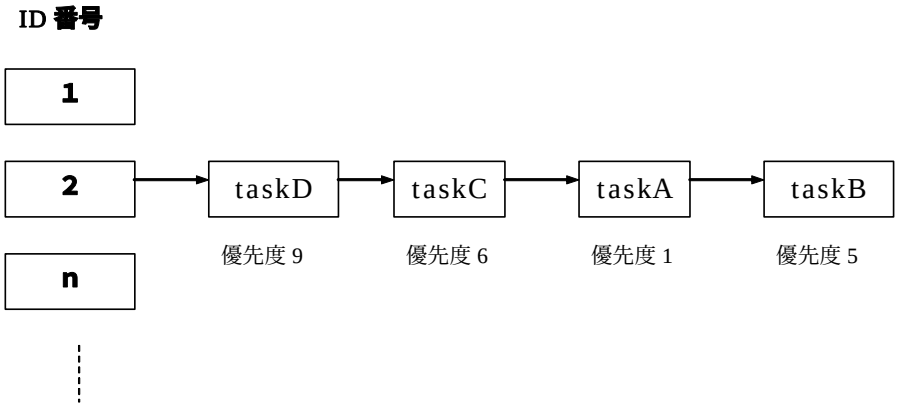


図 3.19 TA_TFIFO 属性の待ち行列

3.4.4 タスクコントロールブロック (TCB)

タスクコントロールブロック (TCB)とは、リアルタイム OS がそれぞれのタスクの状態や優先度などを管理するデータブロックのことを言います。MR30 ではタスクの以下の情報をタスクコントロールブロックとして管理しています。

- タスク接続ポインタ
  レディキューなどを構成するときに使用するタスク接続用ポインタ
- タスクの状態
- タスクの優先度
- タスクのレジスタ情報など を格納したスタック領域のポインタ (現在の SP レジスタの値)
- 起床要求カウンタ
  タスクの起床要求カウンタを蓄積する領域
- タイムアウトカウンタ、待ちフラグパターン
  タスクがタイムアウト待ち状態である時は、残りの待ち時間が格納され、フラグ待ち状態であれば、フラグの待ちパターンがこの領域に格納されます。
- フラグ待ちモード
  イベントフラグ待ちの時の待ちモード
- タイマキュー接続ポインタ
  タイムアウト機能を使用した場合に使用する領域です。タイマキューを構成する時に使用するタスクの接続用ポインタを格納する領域です。
- フラグ待ちパターン
  タイムアウト機能を使用した場合に使用する領域です。
  タイムアウト機能付きのイベントフラグ待ちのサービスコール (twai_flg)を使用した場合に、フラグ待ちパターンが格納されます。なお、この領域は、イベントフラグを使用しない場合は、確保されません。
- 起動要求カウンタ
  タスクの起動要求を蓄積する領域
- タスクの拡張情報
  タスク生成時に設定する、タスクの拡張情報がこの領域に格納されます。

タスクコントロールブロックを 図 3.20に示します。

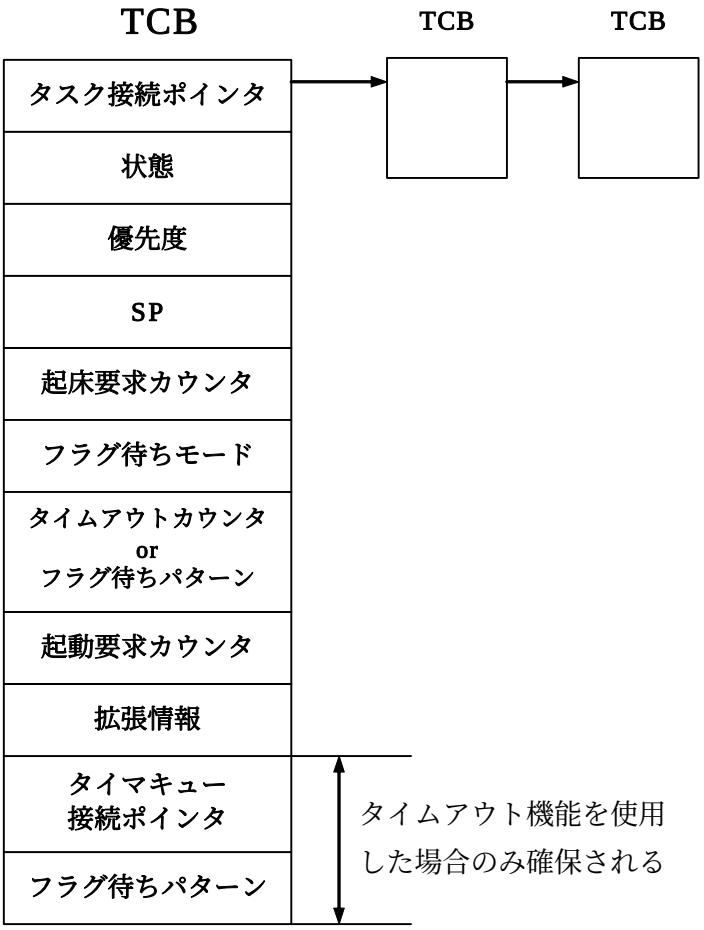


図 3.20 タスクコントロールブロック

3.5 システムの状態

3.5.1 タスクコンテキストと非タスクコンテキスト

システムは、「タスクコンテキスト」か「非タスクコンテキスト」のいずれかのコンテキスト状態で実行します。タスクコンテキストと非タスクコンテキストの違いを 表 3.1に示します。

表 3.1 タスクコンテキストと非タスクコンテキスト

	タスクコンテキスト	非タスクコンテキスト
呼び出し可能なサービスコール	タスクコンテキストから呼び出せるもの	非タスクコンテキストから呼び出せるもの
タスクスケジューリング	レディキューの状態が変化し、ディスパッチ禁止状態、CPU ロック状態のいずれでもない場合に発生	発生しない
スタック	ユーザスタック	システムスタック

非タスクコンテキストで実行される処理には以下のものがあります。

割り込みハンドラ

ハードウェア割り込みにより起動されるプログラムを割り込みハンドラと呼びます。割り込みハンドラの起動には MR30 は全く関与しません。したがって割り込みハンドラの入り口アドレスを割り込みベクタテーブルに直接書き込みます。
割り込みハンドラには、カーネル管理外(OS 独立)割り込み、カーネル管理(OS 依存)割り込みの 2 種類があります。各割り込みについては、5.5 節を参照して下さい。システムクロック割り込みハンドラ(isig_tim)も割り込みハンドラに含まれます。

周期ハンドラ

周期ハンドラはあらかじめ設定された時間毎に周期的に起動されるプログラムです。 設定された周期ハンドラを無効にするか有効にするかは sta_cyc(ista_cyc)や stp_cyc(istp_cyc)サービスコールによりおこないます。
また、周期ハンドラ起動時刻は、set_tim(iset_tim)による、時刻変化の影響を受けません。

アラームハンドラ

アラームハンドラは、 指定した相対時刻経過後に起動されるハンドラです。起動時刻は、sta_alm(ista_alm)設定時の時刻に対する相対時刻で決定され、set_tim(iset_tim)による、時刻変化の影響を受けません。

周期ハンドラとアラームハンドラはシステムクロック割り込み(タイマ割り込み)ハンドラからサブルーチンコールで呼び出されます(図 3.21参照)。したがって、周期ハンドラ、アラームハンドラはシステムクロック割り込みハンドラの一部として動作します。なお、周期ハンドラ、アラームハンドラが呼び出されるときは、システムクロック割り込みの割り込み優先レベルの状態で実行されます。

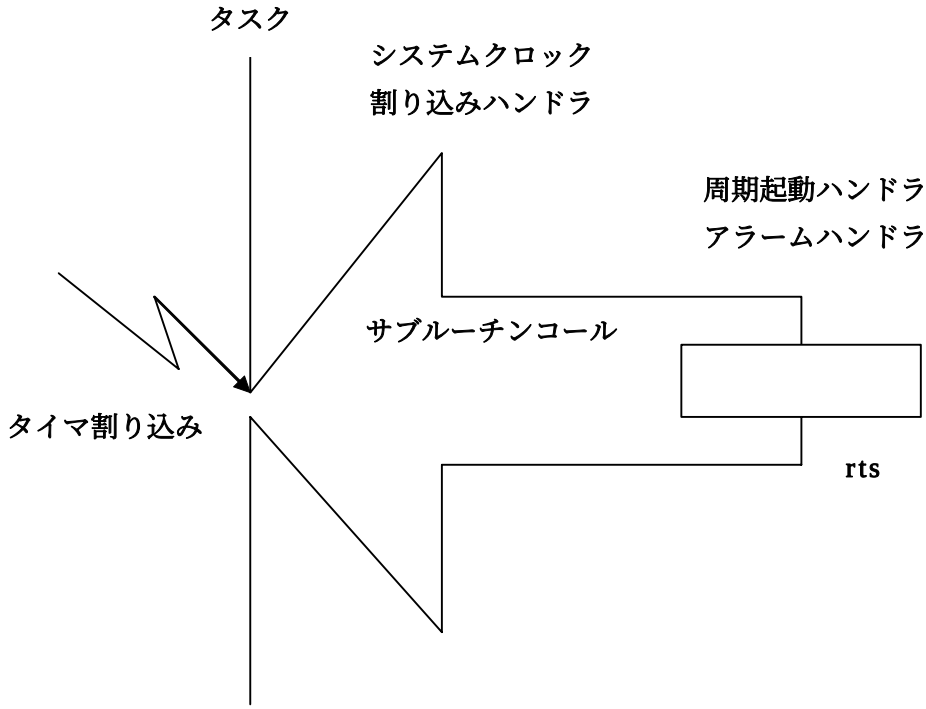


図 3.21 周期ハンドラ、アラームハンドラの起動

3.5.2 ディスパッチ禁止/許可状態

システムは、ディスパッチ許可状態、ディスパッチ禁止状態のいずれかの状態をとります。ディスパッチ禁止状態では、タスクスケジューリングが行われません。また、サービスコール発行タスクが待ち状態に移行するようなサービスコールも呼び出すことはできません。[11] ディスパッチ禁止状態へは、dis_dspサービスコール、ディスパッチ許可状態へはena_dspサービスコールの発行により遷移することができます。また、sns_dspサービスコールによりディスパッチ禁止状態がどうか知ることができます。

3.5.3 CPUロック/ロック解除状態

システムは、CPUロック状態かCPUロック解除状態のいずれかの状態をとります。CPUロック状態では、すべての外部割り込みの受付が禁止され、タスクスケジューリングも行われません。CPUロック状態へはloc_cpu(iloc_cpu)サービスコール、CPUロック解除状態へはunl_cpu(iunl_cpu)サービスコール発行により遷移します。また、sns_locサービスコールによってCPUロック状態かどうか調べることができます。CPUロック状態から発行できるサービスコールは 表 3.2のように制限されます。[12]

表 3.2 CPU ロック状態で使用可能なサービスコール

loc_cpu	iloc_cpu	unl_cpu	iunl_cpu
ext_tsk	sns_ctx	sns_loc	sns_dsp
sns_dpn			

3.5.4 ディスパッチ禁止状態とCPUロック状態

μITRON 4.0仕様では、ディスパッチ禁止状態とCPUロック状態が明確に区別されるようになりました。従って、ディスパッチ禁止状態でunl_cpuサービスコールを発行したとしても、ディスパッチ禁止状態のまま変化せず、タスクスケジューリングは行われません。状態遷移をまとめると 表 3.3のようになります。

表 3.3 dis_dsp,loc_cpu に関する CPU ロック、ディスパッチ禁止状態遷移

状態番号	状態の内容		dis_dsp を実行	ena_dsp を実行	loc_cpu を実行	unl_cpu を実行
	CPU ロック状態	ディスパッチ禁止状態				
1	○	×	×	×	→ 1	→ 3
2	○	○	×	×	→ 2	→ 4
3	×	×	→ 4	→ 3	→ 1	→ 3
4	×	○	→ 4	→ 3	→ 2	→ 4

11 MR30 は、ディスパッチ禁止状態から発行できないサービスコールを発行してもエラーは返しませんが、その場合の動作は保証しません。

12 MR30 は、CPU ロック状態から発行できないサービスコールを発行してもエラーは返しませんが、その場合の動作は保証しません。

3.6 割り込み

3.6.1 割り込みハンドラの種類

MR30 の割り込みハンドラには、カーネル管理(OS 依存)割り込みハンドラとカーネル管理外(OS 独立)割り込みハンドラを定義しています。それぞれの割り込みハンドラの定義を以下に示します。

- カーネル管理(OS 依存)割り込みハンドラ
  カーネル割込マスクレベル(OS 割込禁止レベル) (system.IPL)より割込優先レベルが低い(IPL=0 ～ system.IPL)割り込みハンドラをカーネル管理(OS 依存)割り込みハンドラといいます。
  カーネル管理(OS 依存)割込ハンドラ内では、サービスコールを発行することが出来ます。しかし、サービスコール処理中に発生したカーネル管理(OS 依存)割り込みハンドラはカーネル管理(OS 依存)割込を受け付け可能となるまで割込が遅延します。

- カーネル管理外(OS 独立)割り込みハンドラ
  カーネル割込マスクレベル(OS 割込禁止レベル) (system.IPL)より割込優先レベルが高い(system.IPL+1 ～ 7)割り込みハンドラをカーネル管理外(OS 独立)割り込みハンドラといいます。
  カーネル管理外(OS 独立)割込ハンドラ内では、サービスコールを発行することが出来ません。しかし、サービスコール処理中に発生したカーネル管理外(OS 独立)割込は、カーネル管理(OS 依存)割り込みハンドラを受付可能となっていない区間であっても、カーネル管理外(OS 独立)割込を受け付けることが可能です。

図 3.22 割り込みハンドラのIPに、カーネル割り込みマスクレベル(OS割り込み禁止レベル)を 3 にした場合の、カーネル管理(OS依存)割り込みハンドラとカーネル管理外(OS 独立)割り込みハンドラの関係を示します。

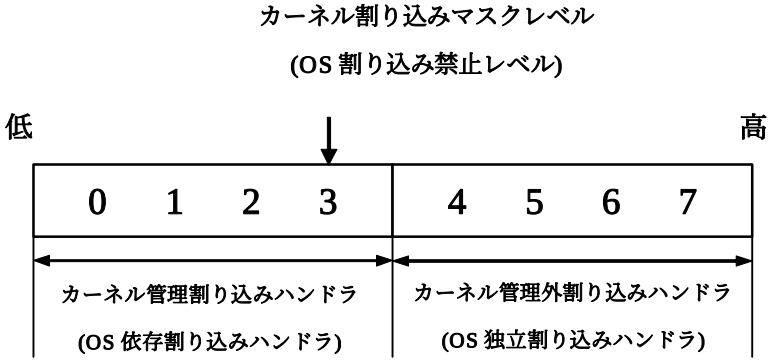


図 3.22 割り込みハンドラの IPL

3.6.2 ノンマスカブル割り込みについて

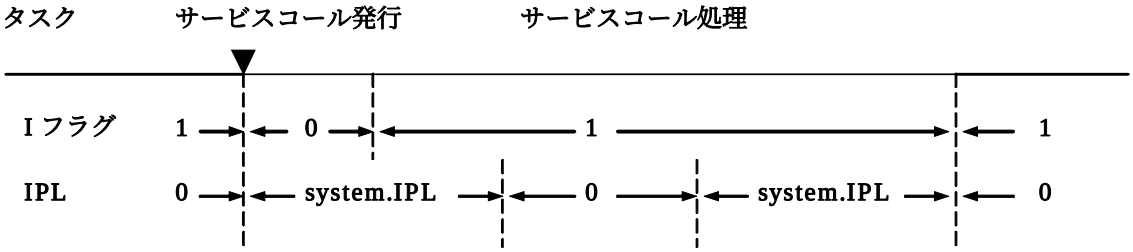
NMI 割り込みおよび監視タイマ割り込みは、必ず、カーネル管理外(OS 独立)割り込みにしてください。カーネル管理(OS 依存)割り込みにした場合、プログラム誤動作の原因となりますので、ご注意ください。

3.6.3 割り込み制御方法

サービスコール内の割り込み禁止/許可の制御は、IPLの操作により行っています。サービスコール内でのIPL値は、カーネル割り込みマスクレベル(OS割り込み禁止レベル) (system.IPL)にして、カーネル管理(OS依存)割り込みハンドラの割り込みを禁止しています。全ての割り込みを許可できる箇所では、サービスコール発行時の IPL 値に戻します。図 3.23に、サービスコール内での割り込み許可フラグとIPLの状態を示します。

- タスクコンテキストからのみ発行できるサービスコールの場合

・サービスコール発行前の I フラグが 1 の場合



・サービスコール発行前の I フラグが 0 の場合

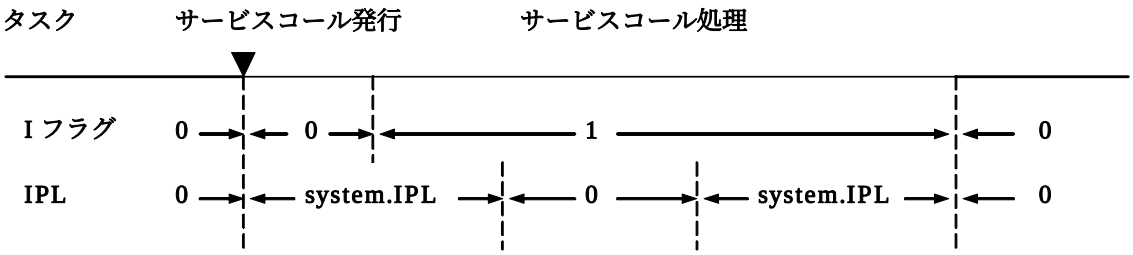
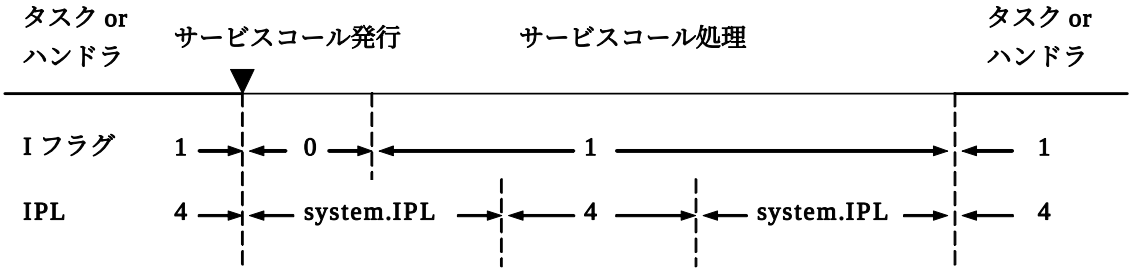


図 3.23 タスクコンテキストからのみ発行できるサービスコール内での割り込み制御

- 非タスクコンテキストからのみ発行できるサービスコール、もしくは、タスクコンテキストと非タスクコンテキストの両方から発行できるサービスコールの場合

・サービスコール発行前の I フラグが 1 の場合



・サービスコール発行前の I フラグが 0 の場合

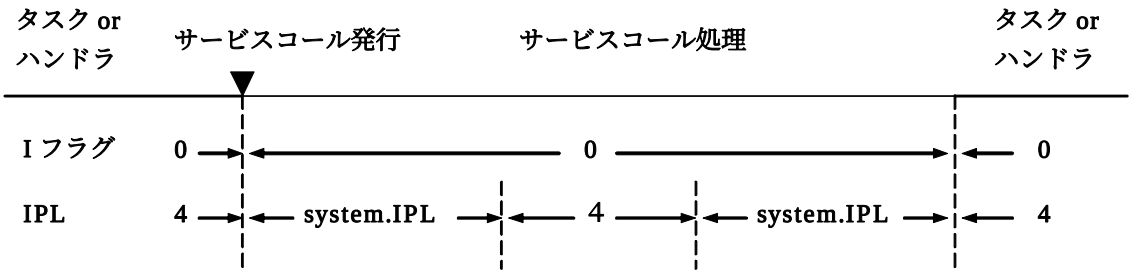


図 3.24 非タスクコンテキストから発行できるサービスコール内での割り込み制御

3.6.4 割り込みの許可、禁止

図 3.23、図 3.24に示すように割り込み許可フラグおよび IPLは、サービスコール内で変化します。従って、タスク、割り込みハンドラ内で割り込みの許可禁止を制御する場合、以下のように対応してください。

タスク内で割り込みを禁止する場合

1. 禁止にしたい割り込みの割り込み制御レジスタ (SFR)を変更する

2. loc_cpu ～ unl_cpu を使用する
   loc_cpu サービスコールにより、制御できる割り込みは、カーネル管理(OS 依存)割り込みのみです。カーネル管理外(OS 独立)割り込みを制御する場合には、1 または 3 による方法で行って下さい。

3. I フラグを操作する
   この方法を使用する場合、I フラグをクリアしてから I フラグをセットするまでの間、サービスコール呼び出しは出来ません。

割り込みハンドラで割り込みを許可する場合(多重割り込みを受け付ける場合)

1. 割り込みハンドラ定義に”E”スイッチを付加する
   割り込みハンドラ定義にて、”pragma_switch = E;”を設定することによって、多重割り込みを許可することが出来ます。

2. I フラグを操作する
   割り込みハンドラ内では、I フラグの操作に制限はありません。

3. 禁止にしたい割り込みの割り込み制御レジスタ (SFR)を変更する

3.7 M16C,R8Cのパワーコントロールとカーネルの動作について

カーネルは、M16C,R8C がサポートするパワーコントロールの機能に関与しません。従って、動作モードの遷移処理は、ユーザプログラムで処理する必要があります。ユーザプログラムで動作モードの遷移を行う場合、ご使用のマイコンのドキュメントに従って処理してください。

また、カーネルがパワーコントロール機能に関与しないため、ユーザプログラムでは特に以下の点に注意してください。

1. システムクロックの停止、動作開始について
   カーネルは、動作モードを移行するために必要なシステムクロックとして使用しているタイマ割り込みを停止、動作開始する処理は行いません。必要に応じてユーザプログラム内で停止、開始処理を記述してください。

2. タイムアウト、タイムイベントハンドラの起動処理について
   動作モードの遷移によってシステムクロックとして使用しているタイマの割り込みの停止、タイマに供給されるクロックの変更が必要になることがあります。これらの処理によって、カーネルは、次のように動作することに注意してください。

   - システム時刻が更新されない、または、時刻がずれる
     get_tim サービスコールのリターンパラメータ(p_systim)に影響があります。
   - 周期ハンドラ、アラームハンドラが起動しない、または起動が遅れる
   - タイムアウト、遅延待ち解除の処理がされない、または指定時間より待ち解除が遅れる
     以下のサービスコール呼び出しによってタイムアウト待ちもしくは、遅延待ちに移行したタスクに影響があります。

dly_tsk	tslp_tsk	twai_sem	twai_flg
trcv_mbx	tsnd_dtq	trcv_dtq	tget_mpf
vtsnd_dtq	vtrcv_dtq		

3.8 スタック

3.8.1 システムスタックとユーザスタック

MR30 のスタックにはシステムスタックとユーザスタックがあります。

- ユーザスタック
  タスクごとに 1 つずつ存在するスタックです。したがって MR30 を用いてアプリケーションを記述する場合はタスクごとのスタック領域を確保する必要があります。

- システムスタック
  MR30 内部 (サービスコール処理中) に使用されるスタックです。MR30 ではサービスコールをタスクが発行するとスタックをユーザスタックからシステムスタックに切り替えます。 (図 3.25を参照して下さい。)
  システムスタックは、割り込みスタックを使用します。

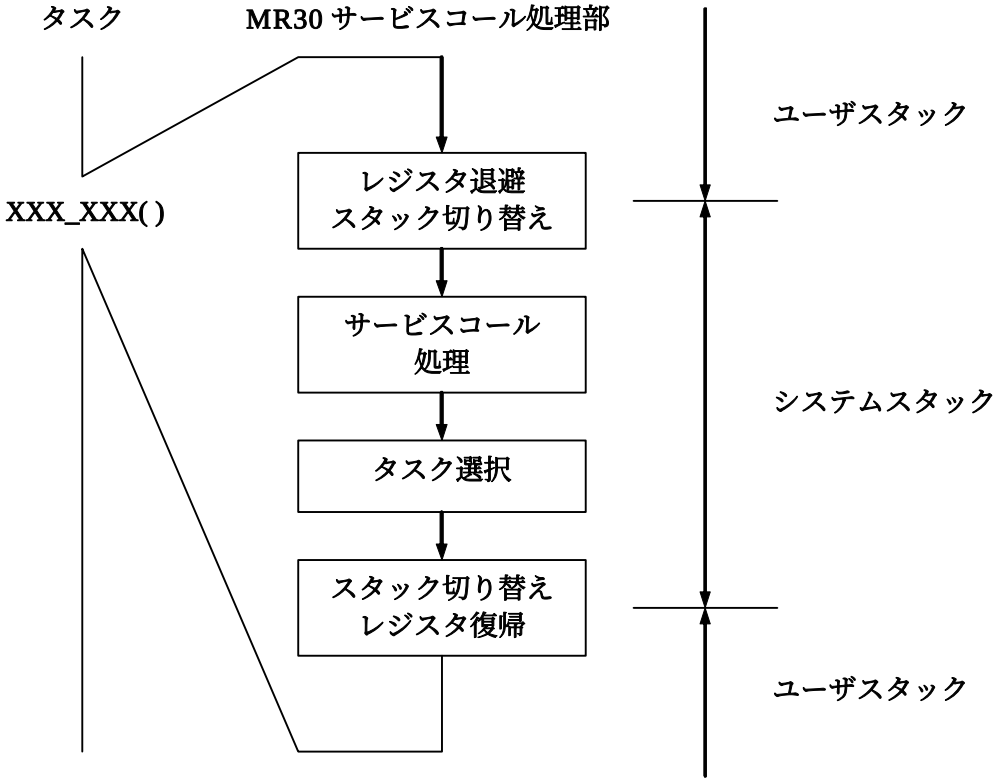


図 3.25 システムスタックとユーザスタック

また、ベクタ番号が 0～31、247～255 の割り込み発生時には、ユーザスタックからシステムスタックに切り替えます。したがって、割り込みハンドラで使用するスタックは全てシステムスタックを使用します。

4. カーネルの機能

4.1 MR30 のモジュール構成

MR30 カーネルは、図 4.1に示すモジュールから構成されています。これらの個々のモジュールはそれぞれのモジュールの機能を実現する関数群より構成されています。 MR30 カーネルはライブラリ形式で提供されシステム生成時に必要な機能のみがリンクされます。すなわちこれらのモジュールを構成する関数群の中で使用している関数のみをリンケージエディタLN30 の機能によりリンクします。ただし、スケジューラとタスク管理の一部および時間管理の一部は必須機能関数ですので常時リンクされます。

アプリケーションプログラムはユーザが作成するプログラムで、タスク・割り込みハンドラ・アラームハンドラおよび周期ハンドラから構成されます。

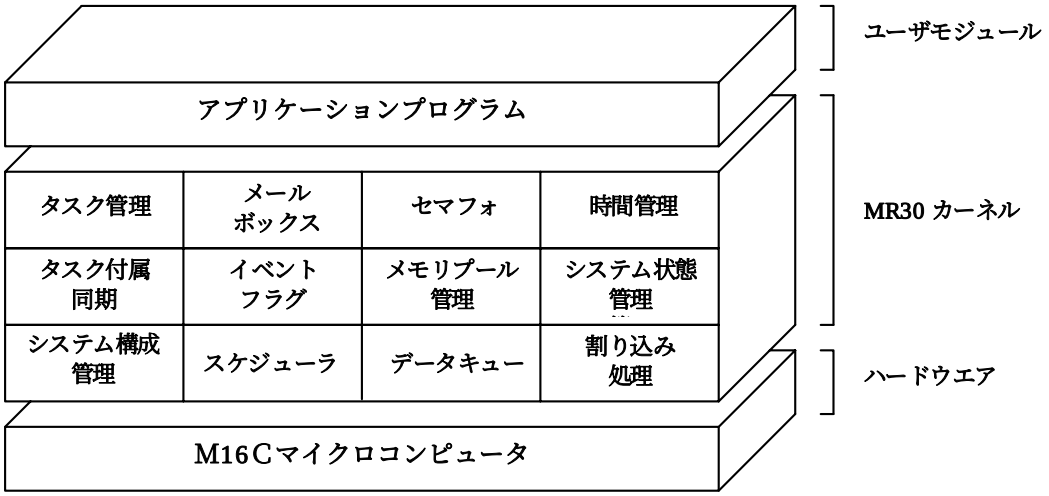


図 4.1 MR30 の構成

4.2 モジュール概要

MR30 カーネルを構成する各モジュールの概要を説明します。

- スケジューラ
  タスクの持つ優先度に基づいて、タスクの処理待ち行列を形成し、その待ち行列の先頭にある優先度の高い(優先度の値の小さい)タスクの処理を実行するよう制御をおこないます。
- タスク管理
  実行･実行可能･待ち･強制待ち等のタスク状態の管理をおこないます。
- タスク付属同期
  他タスクからタスクの状態を変化させることによりタスク間の同期をとります。
- 割り込み管理
  割り込みハンドラからの復帰処理をおこないます。
- 時間管理
  MR30 カーネルで使用するシステムタイマの設定、タイムアウトの処理、ユーザの作成したアラームハンドラ 、周期ハンドラ の起動をおこないます。
- システム状態管理
  MR30 のシステム状態を取得します。
- システム構成管理
  MR30 カーネルのバージョン番号等の情報を取得します。
- 同期･通信
  タスク間の同期をとったり、タスク間の通信をおこなうための機能です。以下の 4 つの機能モジュールが用意されています。

  イベントフラグ
  　MR30 内部で管理されているフラグが立っているか否かによりタスクを実行するかしないかを制御します。これによりタスク間の同期をとることができます。
  セマフォ
  　MR30 内部で管理されているセマフォカウンタ値によりタスクを実行するかしないかを制御します。これによりタスク間の同期をとることができます。
  メールボックス
  　タスク間のデータの通信をデータの先頭アドレスを渡すことによりおこないます。
  データキュー
  　タスク間の 16 ビットデータの通信をおこないます。
- メモリプール管理
  タスクまたはハンドラが使用するメモリ領域の動的な獲得および解放を行います。
- 拡張機能
  μITRON 4.0 仕様の仕様外の機能で long データキュー、オブジェクトのリセット処理を行います。

4.3 カーネルの機能

4.3.1 タスク管理機能

タスク管理機能とは、タスクの起動･終了･優先度の変更等のタスク操作をおこなう機能です。MR30 カーネルが提供するタスク管理機能のサービスコールには、次のものがあります。

- タスクを起動する (act_tsk, iact_tsk)
  あるタスクから、他タスクを起動することにより、起動対象となるタスクの状態を休止状態から実行可能状態もしくは実行状態に移行します。本サービスコールでは、sta_tsk(ista_tsk)と違い、起動要求は蓄積しますが、起動コードを指定することはできません。

- タスクを起動する (sta_tsk, ista_tsk)
  あるタスクから、他タスクを起動することにより、起動対象となるタスクの状態を休止状態から実行可能状態もしくは実行状態に移行します。
  本サービスコールは、act_tsk(iact_tsk)と違い、起動要求は蓄積しませんが、起動コードを指定することができます。

- 自タスクを終了する (ext_tsk)
  自タスクを終了するとタスクの状態が休止状態になります。これにより再起動されるまで、このタスクは実行しません。自タスクに対する起動要求が蓄積されている場合は、再度タスクの起動処理を行います。その際、自タスクは、リセットされたように振る舞います。
  C 言語で記述した場合、本サービスコールは、タスク終了時に明示的に記述されていなくても、タスクからリターンする際に自動的に呼び出されます。

- 他タスクを強制的に終了させる (ter_tsk)
  休止状態以外の他のタスクを強制的に終了させ休止状態にします。対象タスクに対する起動要求が蓄積されている場合は、再度タスクの起動処理を行います。その際、タスクは、リセットされたように振る舞います。 (図 4.2参照)

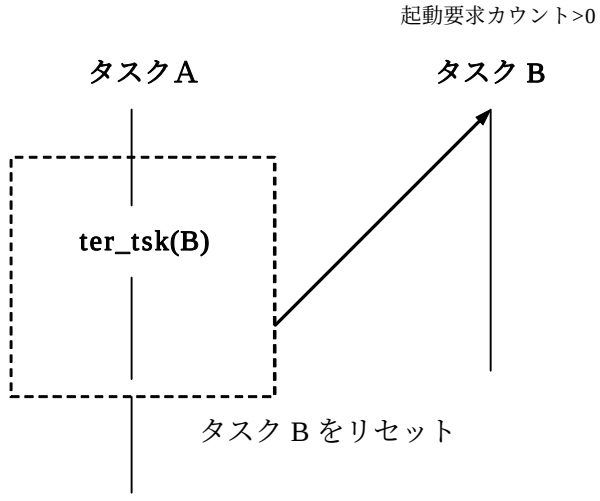
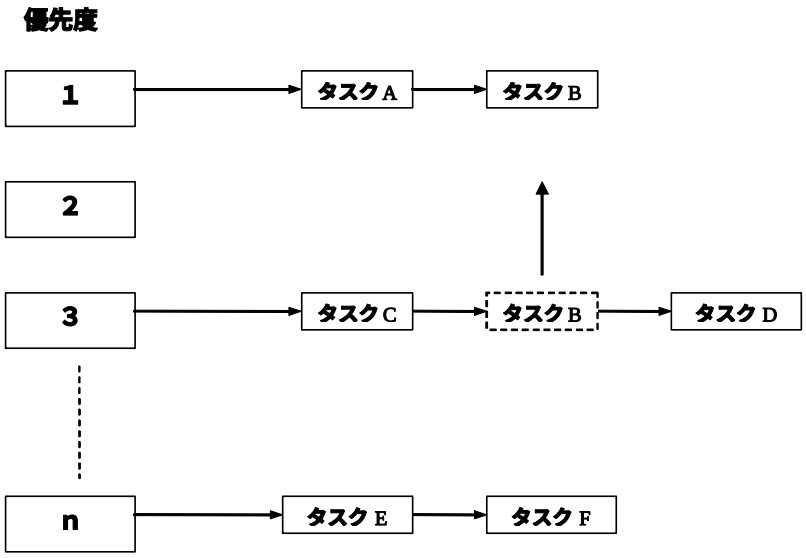


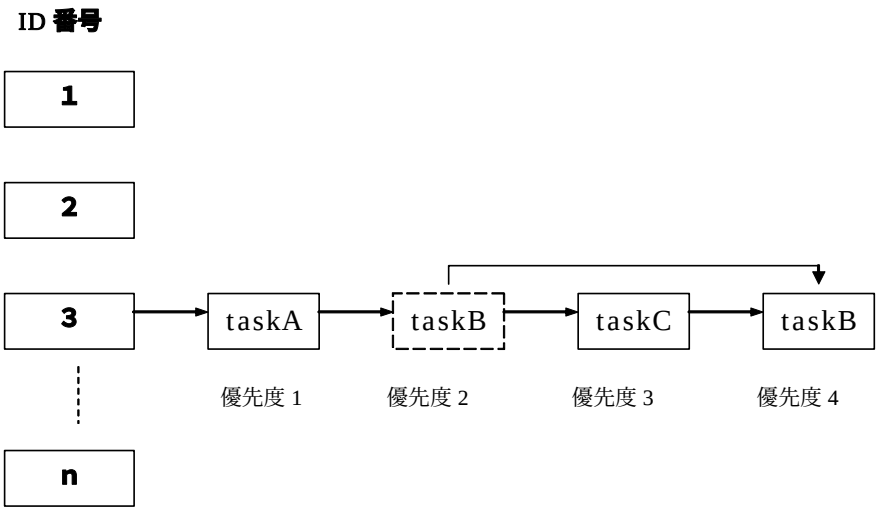
図 4.2 タスクのリセット

- タスクの優先度を変更する (chg_pri, ichg_pri)
  タスクの優先度を変更するとそのタスクが実行可能状態もしくは実行状態であるときは、レディキューも更新されます。 (図 4.3参照)
  また、対象タスクがTA_TPRI属性を持つオブジェクトの待ち行列につながれている場合は、待ち行列も更新されます。図 4.4参照)



タスク B の優先度を 3 から 1 に変更した場合

図 4.3 優先度の変更



タスク B の優先度を 2 から 4 に変更した場合

図 4.4 待ちキューのつなぎ換え

- タスクの優先度を取得する (get_pri, iget_pri)
  タスクの優先度を取得します。
- タスクの状態を参照する(簡易版) (ref_tst, iref_tst)
  対象タスクの状態を参照します。
- タスクの状態を参照する (ref_tsk, iref_tsk)
  対象タスクの状態およびその優先度等を参照します。

4.3.2 タスク付属同期機能

タスク付属同期機能とは、タスク間の同期をとるためにタスクを待ち状態 (もしくは強制待ち状態･二重待ち状態)にしたり、待ち状態になったタスクを起床させたりする機能です。MR30 カーネルが提供するタスク付属同期サービスコールには次のものがあります。

- タスクを待ち状態に移行する (slp_tsk, tslp_tsk)
- 待ち状態のタスクを起床する (wup_tsk, iwup_tsk)
  slp_tsk、tslp_tskサービスコールにより待ち状態に入ったタスクを起床させます。
  slp_tsk、tslp_tskサービスコール以外の条件で待ち状態にあるタスクは起床できません。
  slp_tsk、tslp_tskサービスコール以外の条件で待ちに入ったタスクや休止状態を除く他の状態 のタスクに対してwup_tsk、iwup_tskサービスコールにより起床要求をおこなうと、この起床要求だけが蓄積されます。
  したがって、例えば実行状態のタスクに対して起床要求をおこなうと、この起床要求が一時的に記憶されます。
  そして、その実行状態のタスクがslp_tsk、tslp_tskサービスコールにより待ち状態に入ろうとした時、蓄積された起床要求が有効になり、待ち状態にならずに再び実行を続けます。 (図 4.5参照)
- タスクの起床要求を無効にする (can_wup)
  蓄積された起床要求をクリアします。 (図 4.6参照)

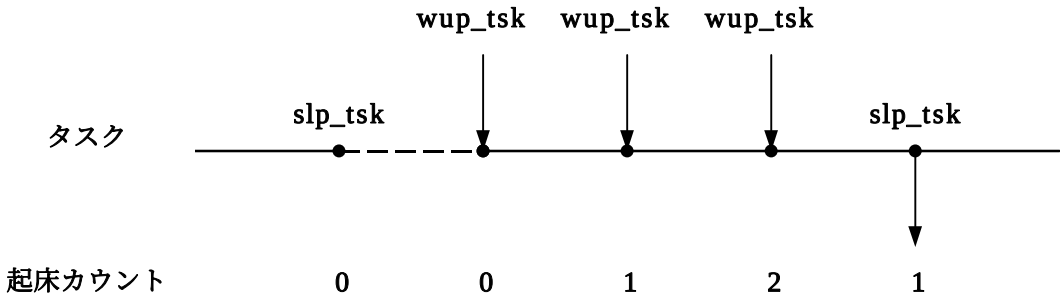


図 4.5 起床要求の蓄積

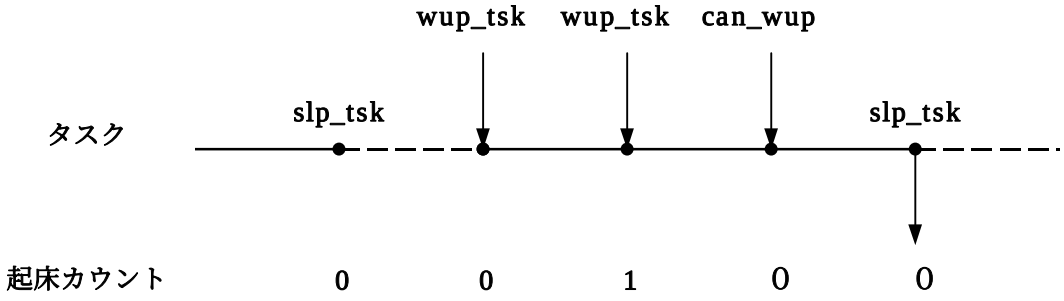


図 4.6 起床要求のキャンセル

- タスクを強制待ち状態に移行する (sus_tsk, isus_tsk)
- 強制待ち状態のタスクを再開する (rsm_tsk, irsm_tsk)
  タスクの実行を強制的に待たせたり、実行を再開したりします。実行可能状態のタスクを強制待ちすれば強制待ち状態になり、待ち状態のタスクを強制待ちすれば二重待ち状態になります。MR30 では最大強制待ち要求ネスト数は1であるため、強制待ち状態のタスクにsus_tskを発行するとエラーE_QOVRが返されます。(図 4.7参照)

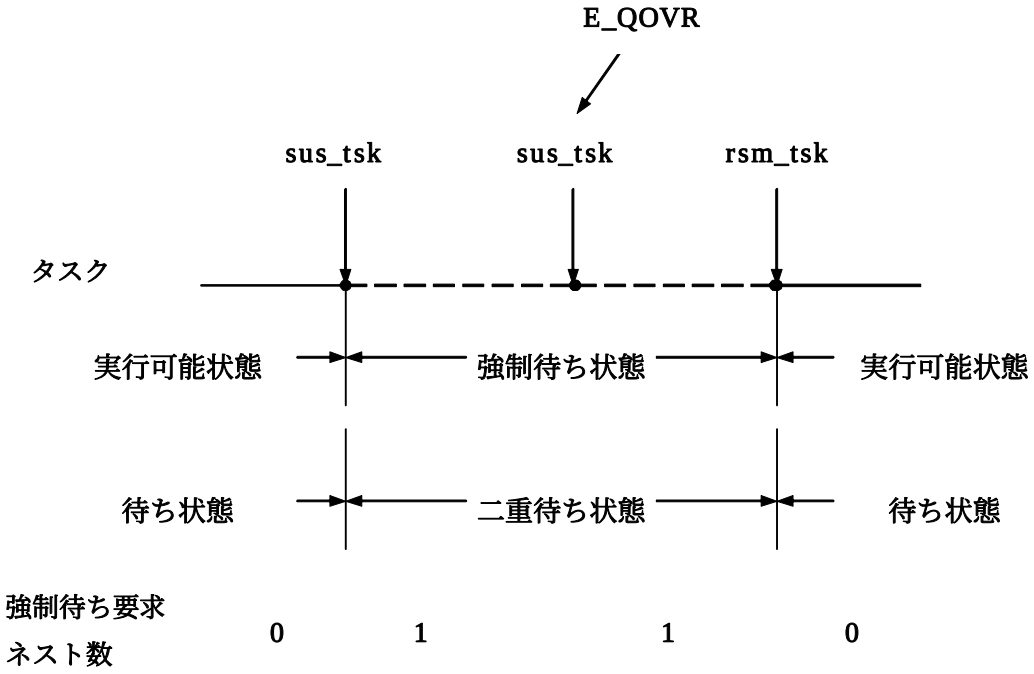


図 4.7 タスクの強制待ちと再開

- 強制待ち状態のタスクを強制再開する (frsm_tsk, ifrsm_tsk)
  強制待ち要求のネスト数をすべてクリアし、タスクの実行を強制的に再開します。MR30 では最大強制待ち要求ネスト数は1であるため、rsm_tsk, irsm_tskと同じ動作となります。(図 4.8参照)

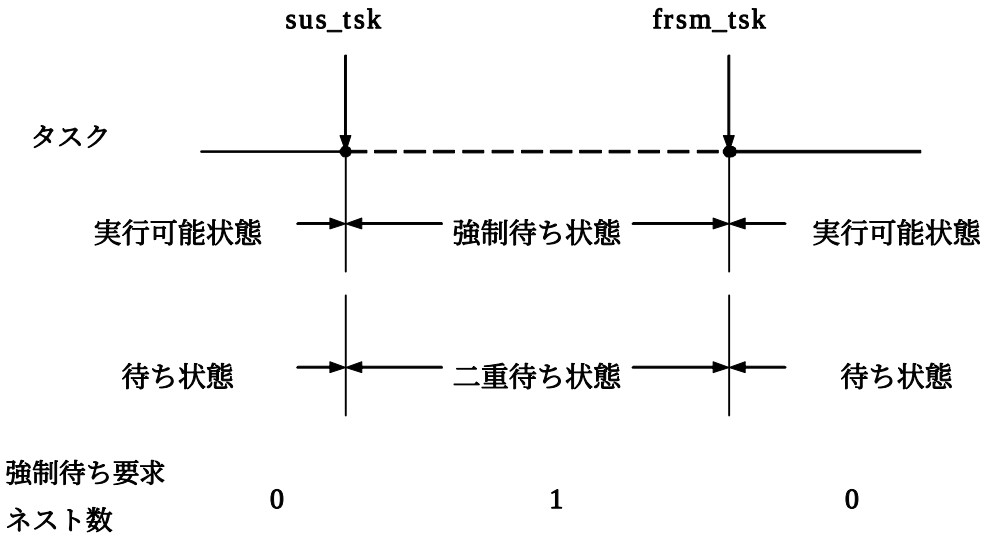


図 4.8 タスクの強制待ちと強制再開

- タスクの待ち状態を強制解除する (rel_wai, irel_wai)
  タスクの待ち状態を強制的に解除します。解除される待ち状態は以下の条件により待ちに入ったタスクです。

  タイムアウト待ち状態
  slp_tsk サービスコールによる (+タイムアウト有)待ち状態
  イベントフラグ (+タイムアウト有)待ち状態
  セマフォ (+タイムアウト有)待ち状態
  メッセージ (+タイムアウト有)待ち状態
  データ送信 (+タイムアウト有)待ち状態
  データ受信 (+タイムアウト有)待ち状態
  固定長メモリブロック (+タイムアウト有)獲得待ち状態
  long 送信 (+タイムアウト有)待ち状態
  long データ受信 (+タイムアウト有)待ち状態

- タスクを一定時間待ち状態に移行します (dly_tsk)
  タスクを一定時間待たせます。図 4.9にdly_tskサービスコールにより10msec間タスクの実行を待たせる例を示します。タイムアウト値として指定する単位は、タイムティック数ではなくms単位で指定します。

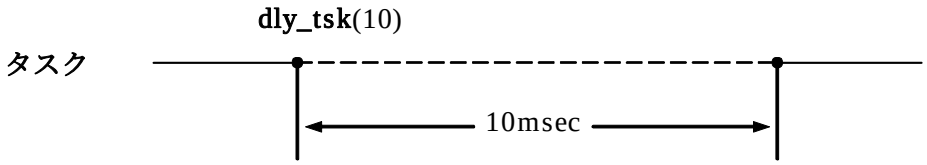


図 4.9 dly_tsk サービスコール

4.3.3 同期・通信機能 (セマフォ)

セマフォは複数のタスクで共有する装置などの資源の競合を防ぐための機能です。例えば図 4.10に示すような場合、すなわち通信回線が3本しかないシステムに4つのタスクが回線を獲得しようと競合した場合に、通信回線を競合することなくタスクに接続することがセマフォを用いるとできます。

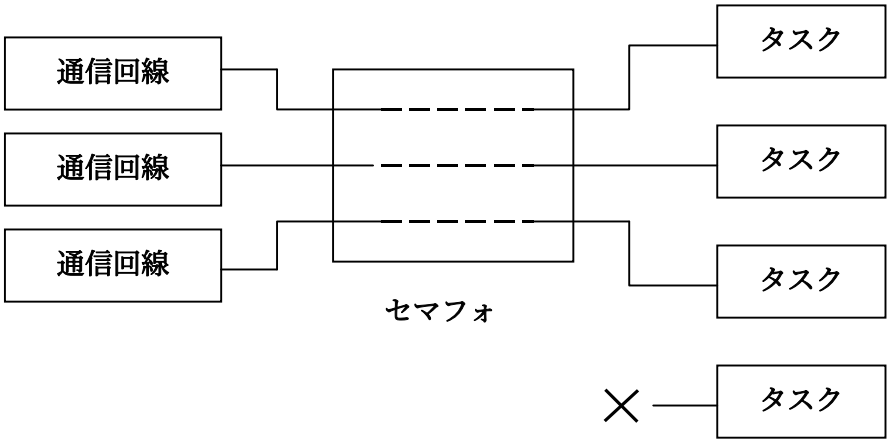


図 4.10 セマフォによる排他制御

セマフォは内部にセマフォカウンタと呼ばれる計数値を持っており、そのセマフォカウンタに基づきセマフォを獲得・解放をおこなうことによって資源の競合を防ぎます。 (図 4.11参照)

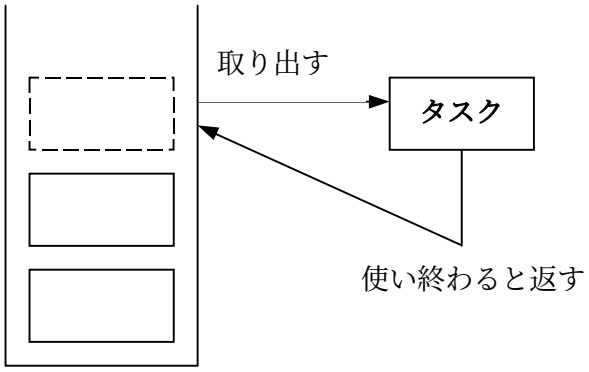


図 4.11 セマフォカウンタ

wai_sem、sig_semサービスコールを用いたタスクの実行制御の例を 図 4.12に示します。

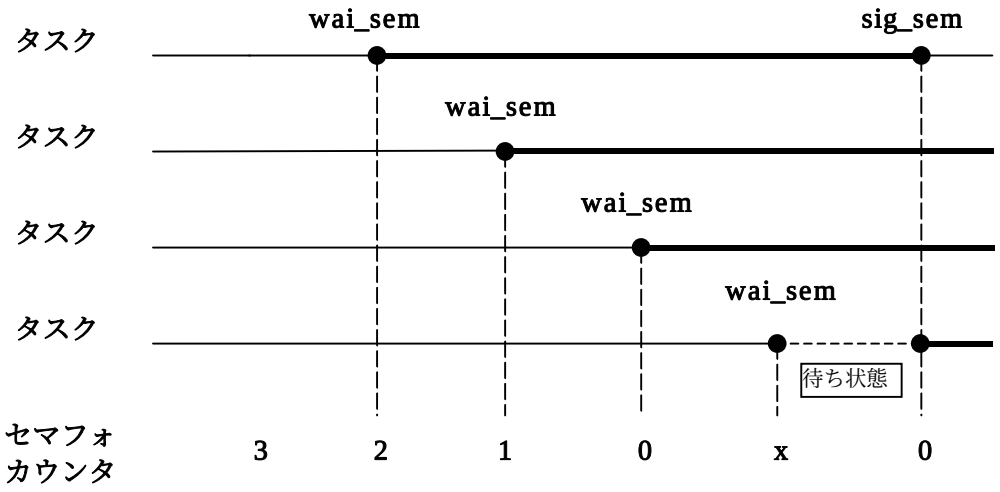


図 4.12 セマフォによるタスクの実行制御

MR30 カーネルが提供するセマフォ同期のサービスコールには次のものがあります。

- セマフォを解放する (sig_sem, isig_sem)
  セマフォを解放します。すなわち、セマフォを待っているタスクがあればそのタスクの待ち状態を解除し、なければセマフォカウンタを1増やします。

- セマフォを獲得する (wai_sem, twai_sem)
  セマフォを獲得します。セマフォカウンタが 1 以上の場合、セマフォカウンタの値を 1 減らします。セマフォカウンタが0であればセマフォを得ることができませんので待ち状態になります。

- セマフォを獲得する (pol_sem, ipol_sem)
  セマフォを得ます。セマフォカウンタが 1 以上の場合、セマフォカウンタの値を 1 減らします。セマフォカウンタが0であれば待ち状態に入らずにエラーコードをかえします。

- セマフォの状態を参照する (ref_sem, iref_sem)
  対象セマフォの状態を参照します。対象セマフォのカウント値や待ちタスクの有無を参照します。

4.3.4 同期・通信機能 (イベントフラグ)

イベントフラグは複数のタスクの実行の同期をとるための MR30 内部に持つ機構です。イベントフラグは、フラグ待ちパターンと 16 ビットのビットパターンによりタスクの実行制御をおこないます。タスクは、設定したフラグ待ちの条件が満たされるまで待ちます。

ひとつのイベントフラグの待ち行列に複数の待ちタスクの接続を許可するかどうかをイベントフラグ属性 TA_WSGL、TA_WMUL を指定することにより決定することができます。

また、イベントフラグ属性に TA_CLR を指定することにより、イベントフラグが待ち条件を満たした場合イベントフラグのビットパターンをすべてクリアすることができます。

図 4.13にwai_flgとset_flgサービスコールを使用したイベントフラグによるタスクの実行制御の例を示します。
イベントフラグは複数のタスクを一度に起床できるという特徴があります。図 4.13では、タスクAからタスクFまでの 6 個のタスクがつながっています。 そして、set_flgサービスコールによって、フラグパターンを 0x0Fにすると、待ち条件にあっているタスクがキューの前から順にはずされていきます。この図で待ち条件を満たすタスクはタスクA、タスクC、タスクEです。このうち、タスクA、タスクC、タスクEがキューからはずされます。したがって、タスクFはキューからはずされません。

もし、本イベントフラグが TA_CLR 属性であれば、タスク A の待ちが解除された時点でイベントフラグのビットパターンは 0 となり、タスク C、タスク E はキューからはずされることはありません。

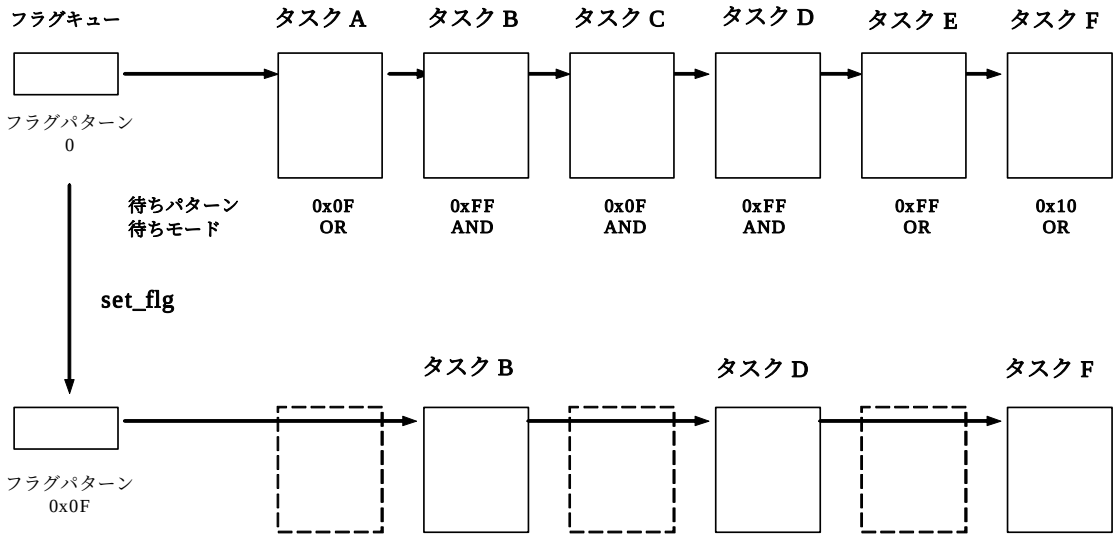


図 4.13 イベントフラグによるタスクの実行制御

MR30 カーネルが提供するイベントフラグのサービスコールには次のものがあります。

- イベントフラグをセットする (set_flg, iset_flg)
  イベントフラグをセットします。これにより、このイベントフラグの待ちパターンを待っていたタスクは待ち解除されます。

- イベントフラグをクリアする (clr_flg, iclr_flg)
  イベントフラグをクリアします。

- イベントフラグを待つ (wai_flg, twai_flg)
  イベントフラグがあるパターンにセットされるのを待ちます。イベントフラグを待つ時のモードは、 以下に示す 2 種類があります。

  AND 待ち
  　指定されたビットが全てセットされるのを待ちます。
  OR 待ち
  　指定されたビットの内いずれか 1 ビットがセットされるのを待ちます。

- イベントフラグを得る (pol_flg, ipol_flg)
  イベントフラグがあるパターンになっているか否かを調べます。このサービスコールではタスクは待ち状態に移行しません。

- イベントフラグの状態を得る (ref_flg, iref_flg)
  対象イベントフラグのビットパターンや待ちタスクの有無を参照します。

4.3.5 同期・通信機能 (データキュー)

データキューとはタスク間でデータの通信をおこなう機構です。例えば、図 3.32 においてタスクAがデータをデータキューに送信しタスクBがそのデータをデータキューから受信することができます。

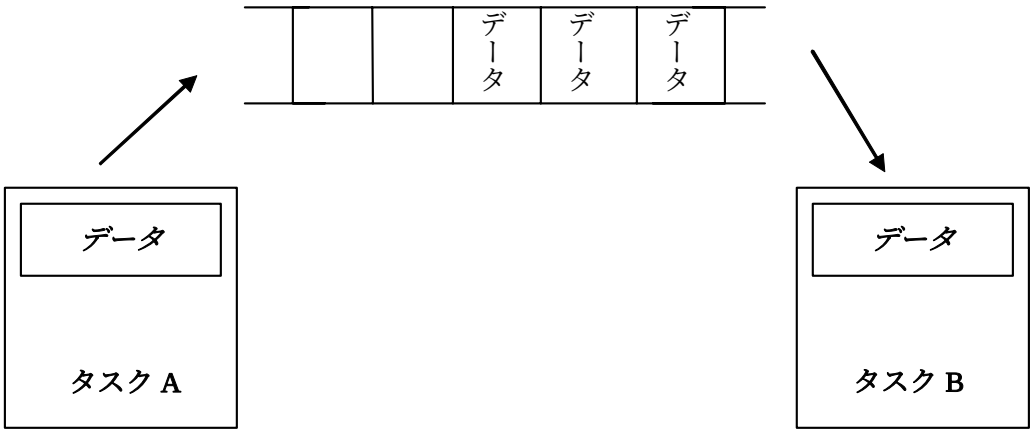


図 4.14 データキュー

このデータキューに送信できるデータ幅は 16 ビットのデータです。データキューにはデータを蓄積する機能があります。蓄積されたデータは FIFO でデータが取り出されます。ただし、データキューに蓄積できるデータの数には制限があります。データキューがデータで一杯になった状態で、データを送信した場合は、サービスコール発行タスクはデータ送信待ち状態に移行します。

MR30 カーネルが提供するデータキューのサービスコールには次のものがあります。

- データを送信します (snd_dtq, tsnd_dtq)
  データをデータキューに送信します。データキューがデータで一杯の場合は、データ送信待ち状態に移行します。
- データを送信します (psnd_dtq, ipsnd_dtq)
  データをデータキューに送信します。データキューがデータで一杯の場合は、データ送信待ち状態に移行せず、エラーコードを返します。
- データを受信します (rcv_dtq, trcv_dtq)
  データキューからデータを受信します。このときデータキューにデータがなければ、データ受信待ち状態になります。
- データを受信します (prcv_dtq, iprcv_dtq)
  データキューからデータを受信します。データキューにデータがない場合は、データ受信待ち状態に移行せず、エラーコードを返します。

- データキューの状態を参照します (ref_dtq, iref_dtq)
  対象データキューにデータが入るのを待っているタスクの有無やデータキューに入っているデータ数を参照します。

4.3.6 同期・通信機能 (メールボックス)

メールボックスとはタスク間でデータの通信をおこなう機構です。例えば、図 4.15においてタスクAがメッセージをメールボックスに投函しタスクBがそのメッセージをメールボックスから取り出すことができます。メールボックスを用いた通信は、メッセージの先頭アドレスの受け渡しによって実現されているため、メッセージサイズに依存せずに高速に行われます。

カーネルは、メッセージキューをリンクリストで管理します。アプリケーション側でリンクリストに用いるためのヘッダ領域を用意しなければいけません。これをメッセージヘッダと呼びます。メッセージヘッダと実際にアプリケーションが使用するメッセージを格納する領域をメッセージパケットと呼びます。カーネルはメッセージヘッダの内容を書き換えて管理しています。アプリケーションからはメッセージヘッダを書き換えることはできません。メッセージキューの状態を 図 4.16に示します。 メッセージヘッダのデータ型は以下の通り定義しています。

T_MSG:	**メールボックスのメッセージヘッダ**
T_MSG_PRI:	**メールボックスの優先度付きメッセージヘッダ**

メッセージキューに入れることのできるメッセージのサイズは、アプリケーション側でヘッダ領域を確保するため、制限はありません。また、送信するためにタスクが待ち状態になることはありません。

メッセージに優先度を設定し、優先度の高いメッセージから受信することができます。この場合メールボックス属性にTA_MPRIを付加します。メッセージをFIFO順に受信する場合はメールボックス属性にTA_MFIFOを付加します。 さらに、メッセージ待ち状態のタスクがメッセージを受信する際、優先度の高いタスクからメッセージを受信することもできます。この場合、この場合メールボックス属性にTA_TPRIを付加します [13]。タスクがFIFO順にメッセージを受信する場合はメールボックス属性にTA_TFIFOを付加します [14]。

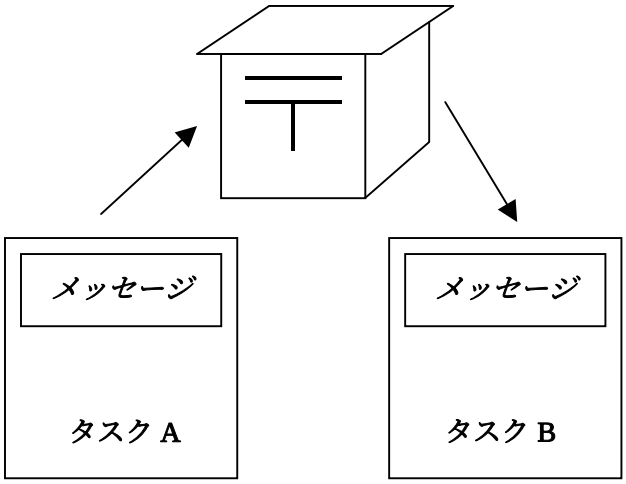


図 4.15 メールボックス

[13] コンフィギュレーションファイルのメールボックス定義”message_queue”項目において TA_MPRI,TA_MFIFO 属性を付加します。
[14] コンフィギュレーションファイルのメールボックス定義”wait_queue”項目において TA_TPRI,TA_TFIFO 属性を付加します。

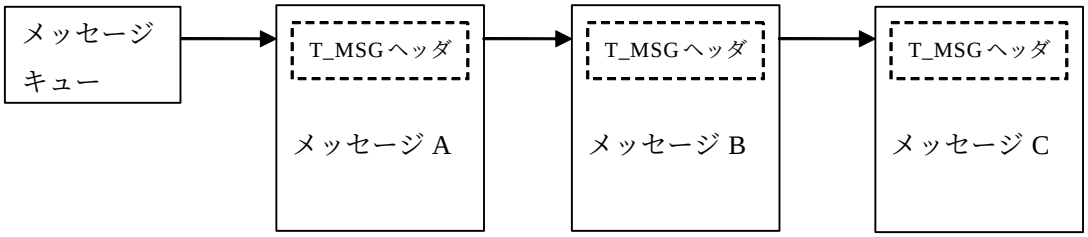


図 4.16 メッセージキュー

MR30 カーネルが提供するメールボックスのサービスコールには次のものがあります。

- メッセージを送信します (snd_mbx, isnd_mbx)
  メッセージをメールボックスに送信します。
- メッセージを受信します (rcv_mbx, trcv_mbx)
  メッセージをメールボックスから受信します。このときメッセージがメールボックスに蓄積されていなければ、送信されるまで待ち状態になります。
- メッセージを受信します (prcv_mbx, iprcv_mbx)
  メッセージをメールボックスから受信します。このときメッセージがメールボックスに蓄積されていなければ、待ち状態にならずにエラーコードを返します。
- メールボックスの状態を参照します (ref_mbx, iref_mbx)
  対象メールボックスにメッセージが入るのを待っているタスクの有無やメールボックスに入っている先頭のメッセージを参照します。

4.3.7 メモリプール管理機能(固定長メモリプール)

固定長メモリプールは、ある決められたサイズのメモリ を動的に管理するための機能です。そのメモリブロックサイズは、コンフィギュレーション時に指定します。固定長メモリプールの動作例を 図 4.17に示します。

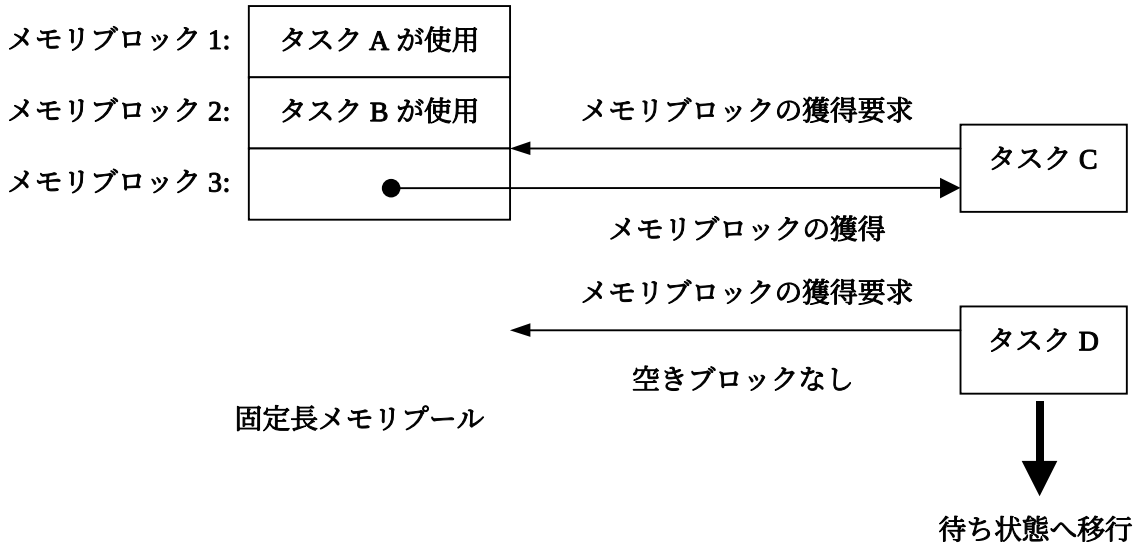


図 4.17 固定長メモリプールの獲得

MR30 カーネルが提供する固定長メモリプール管理サービスコールには次のものがあります。

- メモリブロックを獲得する (get_mpf、tget_mpf)
  指定された ID の固定長メモリプールからメモリブロックを獲得します。空きメモリブロックが、指定された固定長メモリプールにない場合、このサービスコールを発行したタスクは待ち状態に移行し、待ち行列につながれます。

- メモリブロックを獲得する (pget_mpf, ipget_mpf)
  指定された ID の固定長メモリプールからメモリブロックを獲得します。get_mpf、tget_mpf と異なるのは、空きメモリブロックがメモリプールにない場合は、待ち状態に移行せず、エラーコードを返します。

- メモリブロックを解放する (rel_mpf, irel_mpf)[15]
  獲得しているメモリブロックを解放します。指定された固定長メモリプールに対する待ち状態のタスクがある場合には、待ち行列の先頭につながれたタスクに解放したメモリブロックを割り当てます。この時のタスクの状態は、待ち状態から実行可能(READY)状態に移行します。また、待ち状態のタスクがない場合は、メモリプールにメモリブロックを返却します。

- メモリプールの状態を参照する (ref_mpf, iref_mpf)
  対象メモリプールの空きブロック数やブロックサイズを参照します。

[15] MR30 は、引数として渡される解放するメモリブロックのアドレスの妥当性は判定しません。従って、既に解放されたメモリブロックを再度解放しようとした場合や、獲得していないメモリブロックを解放しようとした場合の動作は保証しません。

4.3.8 メモリプール管理機能(可変長メモリプール)

メモリプールから獲得できるメモリブロックサイズが任意に指定可能なことを可変長といいます。MR30 では、メモリを 4 種類の固定長ブロックサイズで管理しています。4 種類の各々のサイズは、ユーザが獲得するメモリブロックの最大サイズから以下の計算式に基づき、MR30 が計算します。本計算に必要なメモリブロックの最大サイズは、コンフィギュレーション時に指定します。

■ **4 種類のブロックサイズ(下記 a,b,c,d) の計算式**

```
a = (((max_memsize+(X-1))/(X * 8))+1) * X
b = a * 2
c = a * 4
d = a * 8

max_memsize : コンフィギュレーションファイルで指定した値
X : ブロック管理用データサイズ（8バイト）
```

■ **コンフィギュレーションファイル例**

```
 variable_memorypool[]{
    max_memsize = 400; <---- 最大獲得サイズ
    heap_size = 5000;
};
```

上記のように、可変長メモリプール定義を行った場合、4 種類の固定長ブロックサイズは、 max_memsize 定義値から 56,112,224,448 となります。

また、ユーザが要求したメモリは、指定サイズをもとに MR30 が計算を行い 4 種類の固定長メモリブロックサイズの中から最適なサイズを選択し、メモリを割り当てます。この 4 種類以外のサイズのメモリブロックを割り当てることはありません。

MR30 カーネルが提供する可変長メモリプール管理サービスコールには次のものがあります。

- メモリブロックを獲得する (pget_mpl)
  ユーザが指定したブロックサイズは、 4 種類のブロックサイズのうちから最適なブロックサイズに丸めて、丸めたサイズ分のメモリをメモリプールから獲得します。
  例えば、ユーザが 200 バイトのメモリを要求した場合 224 バイトに丸めて、224 バイト分のメモリを獲得します。
  メモリが獲得できた場合、獲得したメモリの先頭アドレスとエラーコード E_OK を返します。 獲得できなかった場合には、エラーコード E_TMOUT を返します。

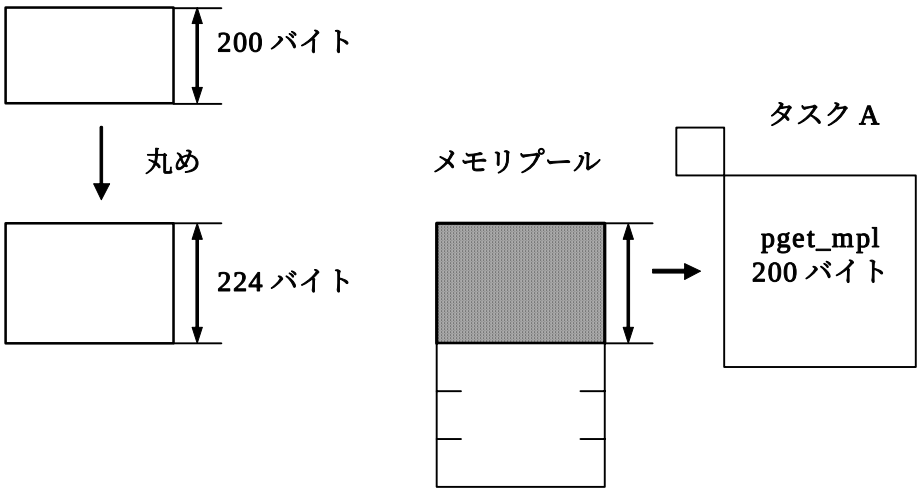


図 4.18 pget_mpl 処理

- メモリブロックを解放する (rel_mpl)[16]
  pget_mplで獲得したメモリブロックを解放します。

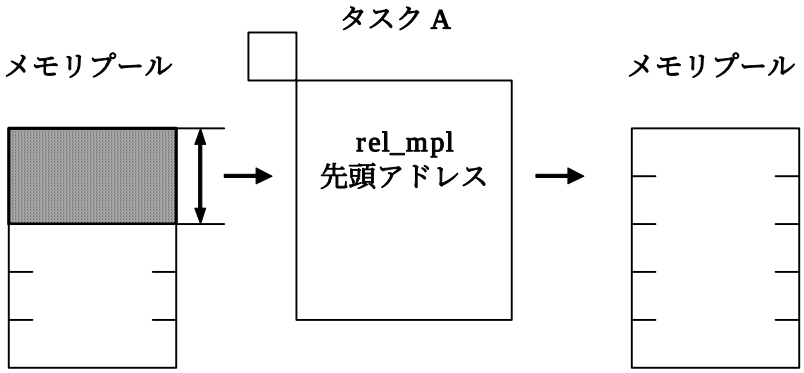


図 4.19 rel_mpl 処理

- メモリプールの状態を参照する (ref_mpl, iref_mpl)
  メモリプールの空き領域の合計サイズやすぐに獲得できる最大の空き領域のサイズを参照します。

[16] MR30 は、引数として渡される解放するメモリブロックのアドレスの妥当性は判定しません。従って、既に解放されたメモリブロックを再度解放しようとした場合や、獲得していないメモリブロックを解放しようとした場合の動作は保証しません。

4.3.9 時間管理機能

時間管理機能はシステムの時刻を管理し、時刻の読みだし 、時刻の設定機能、タイムアウトの処理や特定時刻に起動するアラームハンドラや定期的に起動する周期ハンドラの機能を提供します。

MR30 カーネルはシステムクロックとしてタイマを一つ必要とします。MR30 カーネルが提供する時間管理サービスコールには次のものがあります。なお、システムクロックは必須機能ではありません。したがって下記のサービスコールおよび時間管理機能を使用しなければ、タイマを MR30 用に占有する必要がありません。

- 待ち状態にタイムアウト値を指定すれば、一定時間待ち状態に移行します
  タスクを待ち状態に移行するサービスコール [17]にタイムアウトを指定することができます。サービスコール名は、tslp_tsk、twai_flg、twai_sem、tsnd_dtq、trcv_dtq、trcv_mbx、tget_mpf、vtsnd_dtq、vtrcv_dtqです。タイムアウトの指定時間が経過するまでに待ち解除条件が満たされない場合、エラーコードE_TMOUTを返し、待ち状態が解除されます。待ち解除条件が満たされた場合は、エラーコードはE_OKを返します。タイムアウトの時間単位は、msです。

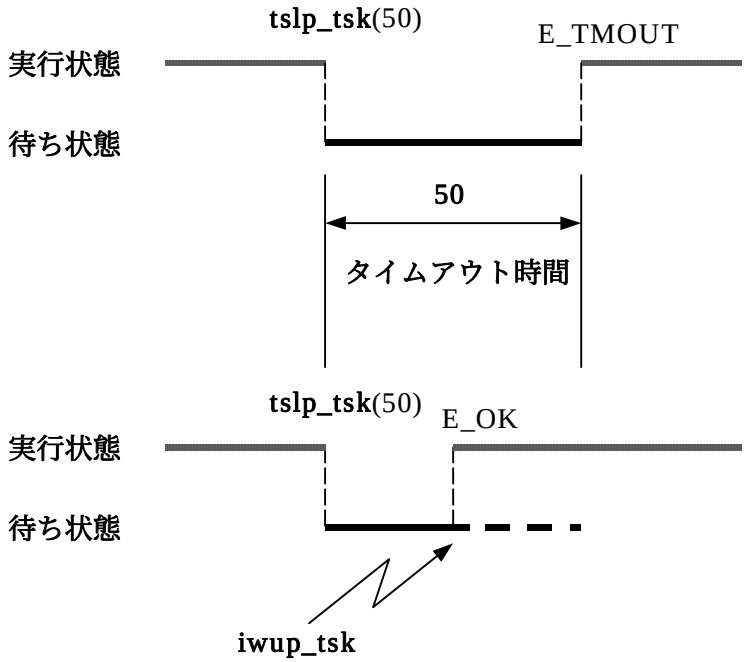


図 4.20 タイムアウト処理

[17] sus_tsk, isus_tsk サービスコールを除きます。

MR30 では、μITRON 仕様の規定通り、指定されたタイムアウト値分以上の時間が経過してからタイムアウト処理を行うことを保証します。具体的には以下のタイミングでタイムアウト処理を行います。

1. タイムアウト値[18]が 0 の場合(dly_taskの場合のみ)[19]

サービスコール発行後の最初のタイムティックでタイムアウトします。

2. タイムアウト値が、タイムティック間隔の倍数である場合

(タイムアウト値/タイムティック間隔)+1 回目のタイムティックでタイムアウトします。例えば、タイムティック間隔が 10ms でタイムアウト値に 40ms を指定した場合、5 回目のタイムティックでタイムアウトします。また、タイムティック間隔が 5ms でタイムアウト値に 15ms を指定した場合、4 回目のタイムティックでタイムアウトします。

3. タイムアウト値が、タイムティック間隔の倍数でない場合

(タイムアウト値/タイムティック間隔)+2 回目のタイムティックでタイムアウトします。例えば、タイムティック間隔が 10ms でタイムアウト値に 35ms を指定した場合、5 回目のタイムティックでタイムアウトします。

- システム時刻を設定する (set_tim, iset_tim)
- システム時刻の値を読みだす (get_tim, iget_tim)
  システム時刻はリセット時からの経過時間を 48 ビットのデータで表します。時間の単位は ms です。

[18] 厳密には、dly_tsk サービスコールの場合は、「タイムアウト値」ではなく「遅延時間」になります。
[19] 厳密には、dly_tsk サービスコールの場合は、「タイムアウト」するのではなく、「遅延待ちが解除されサービスコールが正常終了」します。

4.3.10 時間管理機能(周期ハンドラ)

周期ハンドラは、指定した起動位相経過後、起動周期ごとに起動されるタイムイベントハンドラです。
周期ハンドラの起動には、起動位相を保存する方法と起動位相を保存しない方法があります。起動位相を保存する場合は、周期ハンドラの生成時点を基準に周期ハンドラを起動します。起動位相を保存しない場合は、周期ハンドラの動作開始時点を基準に周期ハンドラを起動します。図 4.21、図 4.22に周期ハンドラの動作例を示します。
タイムティック間隔より、起動周期が短い場合、タイムティック供給(isig_tim 相当の処理)毎に、1 回だけ周期ハンドラを起動します。例えば、タイムティック間隔が 10ms、起動周期が 3ms で、タイムティック供給時に周期ハンドラ動作が開始された場合、タイムティックごとに1回だけ周期ハンドラが起動されることになります。

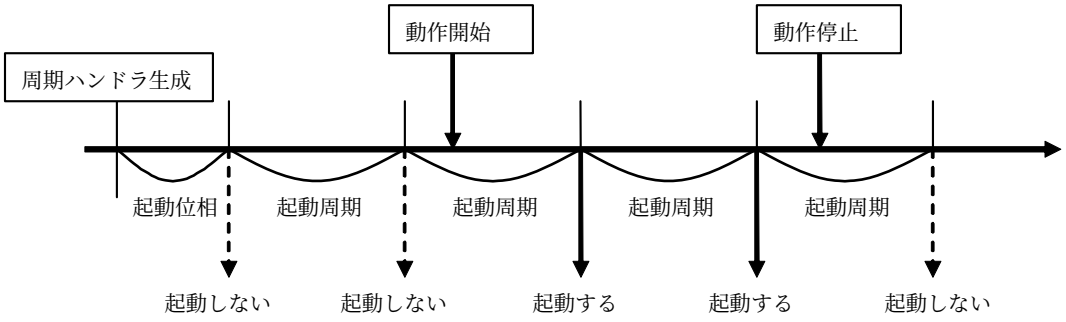


図 4.21 起動位相を保存する場合の動作

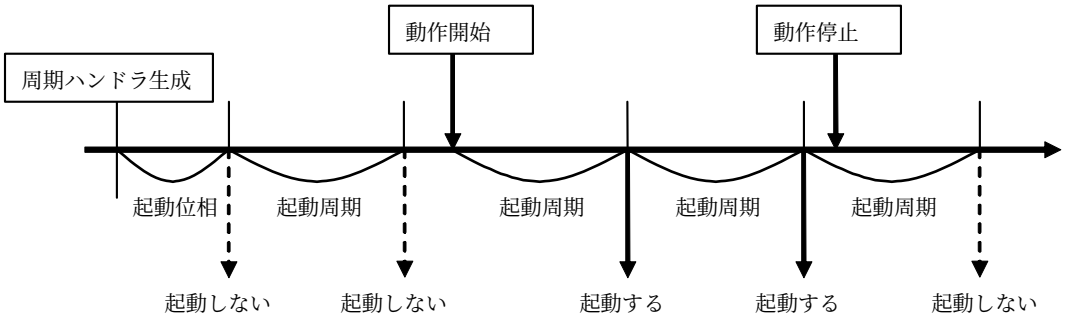


図 4.22 起動位相を保存しない場合の動作

- 周期ハンドラの動作を開始する(sta_cyc, ista_cyc)
  指定された ID の周期ハンドラの動作を開始します。
- 周期ハンドラの動作を停止する(stp_cyc, istp_cyc)
  指定された ID の周期ハンドラの動作を停止します。
- 周期ハンドラの状態を参照する (ref_cyc, iref_cyc)
  周期ハンドラの状態を参照します。対象周期ハンドラの動作状態と次の起動までの残り時間を調べます。

4.3.11 時間管理機能(アラームハンドラ)

アラームハンドラは、指定した時刻になると 1 度だけ起動されるタイムイベントハンドラです。
アラームハンドラを用いることにより、時刻に依存した処理を行うことができます。時刻の指定は、相対時間です。図 4.23に動作例を示します。

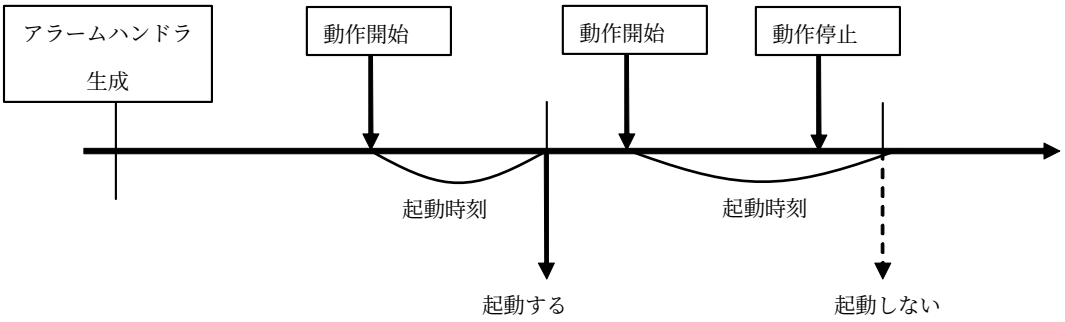


図 4.23 アラームハンドラの動作

- アラームハンドラの動作を開始する(sta_alm, ista_alm)
  指定された ID のアラームハンドラの動作を開始します。
- アラームハンドラの動作を停止する(stp_alm, istp_alm)
  指定された ID のアラームハンドラの動作を停止します。
- アラームハンドラの状態を参照する (ref_alm, iref_alm)
  アラームハンドラの状態を参照します。対象アラームハンドラの動作状態と起動までの残り時間を調べます。

4.3.12 システム状態管理機能

- タスクの実行待ち行列を回転する (rot_rdq , irot_rdq)
  本サービスコールによりTSS(タイムシェアリングシステム) を実現することができます。すなわち、一定周期でレディキューを回転すれば、TSSで必要なラウンドロビンスケジューリングを実現することができます。 (図 4.24参照)

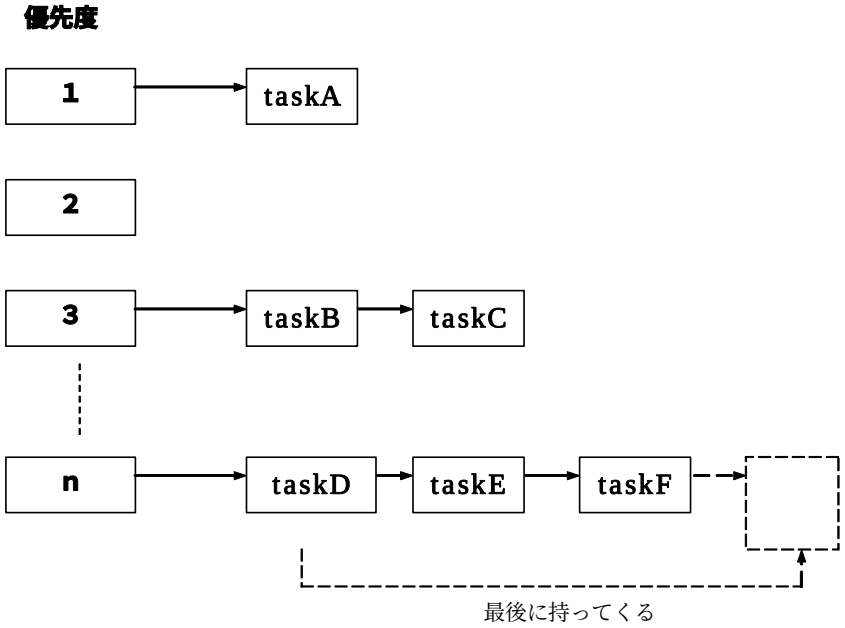


図 4.24 rot_rdq サービスコールによるレディキューの操作

- 自タスクの ID を得る (get_tid, iget_tid)
  自タスクの ID 番号を得ます。ハンドラから発行した場合は、ID 番号の替わりに TSK_NONE(=0)が得られます。
- CPU ロック状態に移行する (loc_cpu, iloc_cpu)
  CPU ロック状態に移行します。
- CPU ロック状態を解除する (unl_cpu, iunl_cpu)
  CPU ロック状態を解除します。
- ディスパッチ禁止状態に移行する (dis_dsp)
  ディスパッチ禁止状態に移行します。
- ディスパッチ禁止状態を解除する (ena_dsp)
  ディスパッチ禁止状態を解除します。
- コンテキスト状態を得ます (sns_ctx)
  コンテキスト状態を得ます。
- CPU ロック状態を得ます (sns_loc)
  CPU ロック状態を得ます。
- ディスパッチ禁止状態を得ます (sns_dsp)
  ディスパッチ禁止状態を得ます。

- ディスパッチ保留状態を得ます (sns_dpn)
  ディスパッチ保留状態を得ます。

4.3.13 割り込み管理機能

割り込み管理機能は外部割り込みの発生に対して、実時間で処理をおこなう機能を提供します。MR30 カーネルが提供する割り込み管理サービスコールには次のものがあります。

- 割り込みハンドラから復帰します (ret_int)
  ret_int サービスコールは、割り込みハンドラから復帰するとき、必要ならばスケジューラを起動し、タスク切り替えをおこないます。
  本機能は、C言語を用いた場合 、ハンドラ関数の終了時に自動で呼び出されます。従って、この場合、本サービスコールを呼び出す必要はありません。
  図 4.25に割り込み処理の流れをしめします。なお、タスク選択からレジスタ復帰までの処理をスケジューラと呼びます。

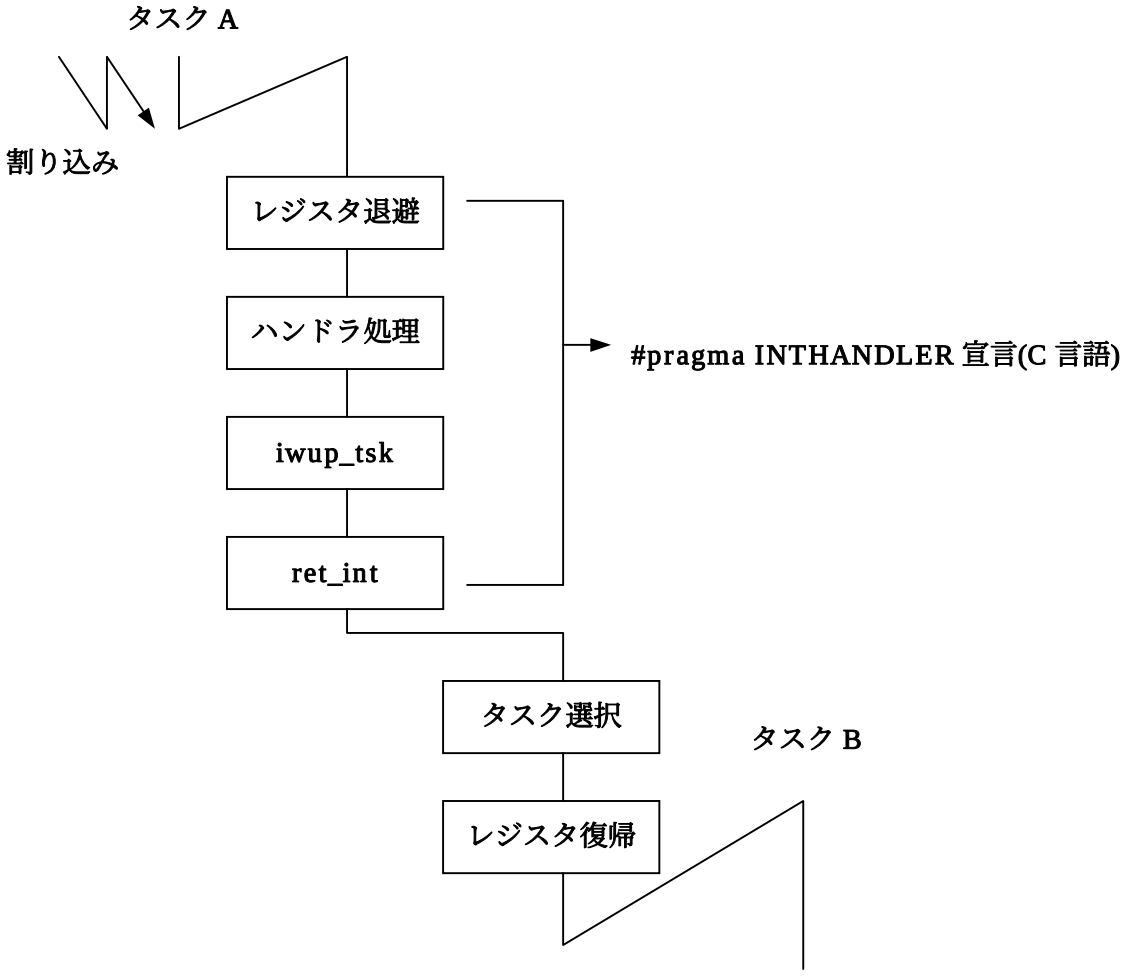


図 4.25 割り込み処理の流れ

4.3.14 システム構成管理機能

MR30 のバージョン情報を参照する機能です。

- MR30 のバージョンを得る(ref_ver, iref_ver)
  MR30 のバージョン情報を得ることができます。このバージョン情報はμITRON 仕様で標準化された形式で得ることができます。

4.3.15 拡張機能(longデータキュー)

long データキューは、μITRON 4.0 仕様の仕様外の機能です。データキュー機能は、データを 16bit として扱いますが、long データキューは、データを 32bit データとして扱います。両者の動作はデータサイズの差異以外は違いがありません。

- データを送信します (vsnd_dtq, vtsnd_dtq)
  データを long データキューに送信します。long データキューがデータで一杯の場合は、データ送信待ち状態に移行します。
- データを送信します (vpsnd_dtq, vipsnd_dtq)
  データを long データキューに送信します。long データキューがデータで一杯の場合は、データ送信待ち状態に移行せず、エラーコードを返します。
- データを受信します (vrcv_dtq, vtrcv_dtq)
  long データキューからデータを受信します。このとき long データキューにデータがなければ、データが送信されるまで待ち状態になります。
- データを受信します (vprcv_dtq, viprcv_dtq)
  long データキューからデータを受信します。long データキューにデータがない場合は、データ受信待ち状態に移行せず、エラーコードを返します。
- データキューの状態を参照します (vref_dtq, viref_dtq)
  対象 long データキューにデータが入るのを待っているタスクの有無や long データキューに入っているデータ数を参照します。

4.3.16 拡張機能(リセット機能)

リセット機能は、μITRON 4.0 仕様の仕様外の機能です。メールボックス、データキュー、メモリプールなどを初期状態にします。

- データキューを初期化する(vrst_dtq)
  データキューを初期化します。送信待ちのタスクがある場合は、待ち状態を解除し、エラーコード EV_RST を返します。
- メールボックスを初期化する(vrst_mbx)
  メールボックスを初期化します。
- 固定長メモリプールを初期化する(vrst_mpf)
  固定長メモリプールを初期化します。待ち状態のタスクがある場合は、待ち状態を解除し、エラーコード EV_RST を返します。
- 可変長メモリプールを初期化する(vrst_mpl)
  可変長メモリプールを初期化します。
- long データキューを初期化する(vrst_vdtq)
  long データキューを初期化します。送信待ちのタスクがある場合は、待ち状態を解除し、エラーコード EV_RST を返します。

5. サービスコールリファレンス

5.1 タスク管理機能

表 5.1にタスク管理機能の仕様を示します。項番4タスク属性の記述言語は、GUIコンフィギュレータでの指定内容です。コンフィギュレーションファイルには出力されず、カーネルも関知しません。タスクのスタックは、コンフィギュレーション時に、タスク毎にセクション名を指定し、異なる領域に配置することが出来ます。

表 5.1 タスク管理機能の仕様

項番	項目	内容
1	タスク ID	1-255
2	タスク優先度	1-255
3	タスク起動要求キューイング数の最大値	15 回
4	タスク属性	TA_HLNG : 高級言語記述 TA_ASM : アセンブリ言語記述 TA_ACT: 起動属性
5	タスクスタック	セクション指定可能

表 5.2 タスク管理機能サービスコール一覧

項番	サービスコール		機能	呼び出し可能なシステム状態					
				T	N	E	D	U	L
1	act_tsk	[S]	タスクの起動	○		○	○	○	
2	iact_tsk	[S]			○	○	○	○	
3	can_act	[S]	タスク起動要求のキャンセル	○		○	○	○	
4	ican_act				○	○	○	○	
5	sta_tsk		タスクの起動(起動コード指定)	○		○	○	○	
6	ista_tsk				○	○	○	○	
7	ext_tsk	[S]	自タスクの終了	○		○	○	○	○
8	ter_tsk	[S]	タスクの強制終了	○		○	○	○	
9	chg_pri	[S]	タスク優先度の変更	○		○	○	○	
10	ichg_pri				○	○	○	○	
11	get_pri	[S]	タスク優先度の参照	○		○	○	○	
12	iget_pri				○	○	○	○	
13	ref_tsk		タスクの状態参照	○		○	○	○	
14	iref_tsk				○	○	○	○	
15	ref_tst		タスクの状態参照(簡易版)	○		○	○	○	
16	iref_tst				○	○	○	○	

【注】

- “[S]”はスタンダードプロファイルのサービスコールです。
- “呼び出し可能なシステム状態”内のそれぞれの記号は、以下の意味です。
  "T"はタスクコンテキストから呼出し可能、"N"は非タスクコンテキストから呼出し可能
  "E"はディスパッチ許可状態から呼出し可能、"D"はディスパッチ禁止状態から呼出し可能
  "U"は CPU ロック解除状態から呼出し可能、"L"は CPU ロック状態から呼出し可能

act_tsk タスクの起動
iact_tsk タスクの起動 (ハンドラ専用)

■ C 言語 API

```
ER ercd = act_tsk( ID tskid );
ER ercd = iact_tsk( ID tskid );
```

● パラメータ

ID	tskid	対象タスク ID 番号

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード

■ アセンブリ言語 API

```
.include mr30.inc
act_tsk   TSKID
iact_tsk  TSKID
```

● パラメータ

TSKID	対象タスク ID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
A0	対象タスク ID 番号

■ エラーコード

E_QOVR	キューイングオーバーフロー

■ 機能説明

tskid で示されたタスクを起動します。起動したタスクは休止(DORMANT)状態から実行可能(READY)状態もしくは実行(RUNNING)状態へ移行します。
タスク起動時に行われる処理は、以下の通りです。

1 タスクの現在優先度を初期化する。
2 起床要求キューイング数をクリアする。
3 強制待ち要求ネスト数をクリアする。

tskid=TSK_SELF (0) の指定により、自タスクの指定になります。タスクには、タスク生成時に指定したタスクの拡張情報がパラメータとして渡ります。非タスクコンテキストにおいて、tskid に TSK_SELF は指定した場合の動作は保証されません。

対象タスクが休止状態でない場合には、本サービスコールによるタスクの起動要求は、キューイングされます。すなわち、起動要求カウントに1加算されます。起動要求カウントの最大値は、15 です。起動要求カウントの最大値を越える場合は、エラーコード E_QOVR を返します。

tskid に TSK_SELF が指定された場合は、自タスクを対象タスクとします。

本サービスコールは、タスクコンテキストからは、act_tsk、非タスクコンテキストからは、iact_tsk を使用してください。

■ 記述例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include “kernel_id.h”
void task1( VP_INT stacd )
{
   ER ercd;
        :
    ercd = act_tsk( ID_task2 );
        :
}
void task2( VP_INT stacd )
{
        :
    ext_tsk();
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr30.inc
    .GLB        task
task:
                :
    pushm               A0
    act_tsk             #ID_TASK3
                :
```

can_act 起動要求カウントのキャンセル
ican_act 起動要求カウントのキャンセル (ハンドラ専用)

■ C 言語 API

```
ER_UINT actcnt = can_act( ID tskid );
ER_UINT actcnt = ican_act( ID tskid );
```

● パラメータ

ID	tskid	対象タスク ID 番号

● リターンパラメータ

ER_UINT	actcnt > 0	キャンセルされた起動要求カウント
	actcnt = 0	

■ アセンブリ言語 API

```
.include mr30.inc
can_act   TSKID
ican_act  TSKID
```

● パラメータ

TSKID	対象タスク ID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	キャンセルされた起動要求カウント
A0	対象タスク ID 番号

■ エラーコード

なし

■ 機能説明

tskid で示されたタスクにキューイングされていた起動要求回数を求め、その結果をリターンパラメータとして返し、同時にその起動要求を全て無効にします。

tskid=TSK_SELF (0) の指定により、自タスクの指定になります。非タスクコンテキストにおいて、tskid に TSK_SELF は指定した場合の動作は保証されません。

休止状態のタスクを対象として呼び出すこともできます。その場合のリターンパラメータは 0 となります。

本サービスコールは、タスクコンテキストからは、can_act、非タスクコンテキストからは、ican_act を使用してください。

■ 記述例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task1()
 {
    ER_UINT actcnt;
        :
    actcnt = can_act( ID_task2 );
        :
 }
 void task2()
 {
        :
    ext_tsk();
 }
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr30.inc
    .GLB        task
task:
                :
    PUSHM       A0
    can_act     #ID_TASK2
                :
```

sta_tsk タスクの起動(起動コード指定)
ista_tsk タスクの起動 (起動コード指定、ハンドラ専用)

■ C 言語 API

```
ER ercd = sta_tsk( ID tskid,VP_INT stacd );
ER ercd = ista_tsk ( ID tskid,VP_INT stacd );
```

● パラメータ

ID	tskid	対象タスク ID 番号
VP_INT	stacd	タスク起動コード

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード

■ アセンブリ言語 API

```
.include mr30.inc
sta_tsk   TSKID, STACD
ista_tsk  TSKID, STACD
```

● パラメータ

TSKID	対象タスク ID 番号
STATCD	タスク起動コード

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R1	タスク起動コード
A0	対象タスク ID 番号

■ エラーコード

E_OBJ	オブジェクト状態が不正(tskid のタスクが休止状態ではない)

■ 機能説明

tskid で示されたタスクを起動します。なわち指定したタスクを休止(DORMANT)状態から実行可能(READY)状態もしくは、実行(RUNNING)状態へ移行します。本サービスコールは、起動要求をキューイングしません。したがって、対象タスクが休止(DORMANT)状態にない場合に発せられた要求に対しては、サービスコール発行タスクにエラーE_OBJ を返します。本サービスコールは、指定したタスクが休止(DORMANT)状態であるときのみ有効です。起動コード stacd は、16 ビットです。stacd は起動タスクにパラメータとして渡されます。

ter_tsk、ext_tsk などで終了したタスクを再起動した場合、タスクは以下の状態でスタートします 。

1 タスクの現在優先度を初期化する。
2 起床要求キューイング数をクリアする。
3 強制待ち要求ネスト数をクリアする。

本サービスコールは、タスクコンテキストからは、sta_tsk、非タスクコンテキストからは、ista_tsk を使用してください。

■ 記述例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
  ER ercd;
  VP_INT stacd = 0;
  ercd = sta_tsk( ID_task2, stacd );
        :
}
void task2(VP_INT msg)
{
  if(msg == 0)
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr30.inc
    .GLB        task
task:
                :
    PUSHM       A0,R1
    sta_tsk     #ID_TASK4,#01234H
                :
```

ext_tsk 自タスクの終了

■ C 言語 API

```
ER ercd = ext_tsk();
```

● パラメータ

なし

● リターンパラメータ

本サービスコールからﾘﾀｰﾝしない

■ アセンブリ言語 API

```
.include mr30.inc
ext_tsk
```

● パラメータ

なし

● サービスコール発行後のレジスタ内容

本サービスコールからﾘﾀｰﾝしない

■エラーコード

本サービスコールからﾘﾀｰﾝしない

■ 機能説明

自タスクを終了します。すなわち、自タスクを実行(RUNNING)状態から休止(DORMANT)状態へ移行します。ただし、自タスクに対する起動要求カウントが1の場合は、起動要求カウントを1減じ、再度 act_tsk,iact_tsk の処理と同様の処理を行い、タスクは、休止(DORMANT)状態から実行可能状態(READY)にします。タスクを起動するときにパラメータとして、タスク拡張情報を渡します。

C 言語で記述した場合、本サービスコールは、タスクからのﾘﾀｰﾝで自動的に発行されるようになっています。

本サービスコールの発行では自タスクが以前に獲得していた資源 (セマフォなど)は解放しません。

本サービスコールはタスクコンテキストでのみ使用可能です。また、本サービスコールは、CPU ロック状態、ディスパッチ禁止状態であっても使用可能です。この場合、CPU ロック状態、ディスパッチ禁止状態は解除されます。しかし、非タスクコンテキストでは使用できません。

■ 記述例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task(void)
 {
      :
   ext_tsk();
 }
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr30.inc
    .GLB        task
task:
        :
    ext_tsk
```

ter_tsk タスクの強制終了

■ C 言語 API

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

● パラメータ

ID	tskid	対象タスク ID 番号

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード

■ アセンブリ言語 API

```
.include mr30.inc
ter_tsk   TSKID
```

● パラメータ

TSKID	対象タスク ID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
A0	対象タスク ID 番号

■エラーコード

E_OBJ	オブジェクト状態が不正(tskid のタスクが休止状態)
E_ILUSE	サービスコール不正使用(tskid に自タスクを指定)

■ 機能説明

tskid で示されたタスクを、強制的に終了させます。対象タスクの起動要求カウントが1以上の場合、起動要求カウントを1減じ、再度 act_tsk,iact_tsk の処理と同様の処理を行い、タスクは、休止(DORMANT)状態から実行可能状態(READY)にします。タスクを起動するときにパラメータとして、タスク拡張情報を渡します。

このサービスコールで自タスクを指定した場合(TSK_SELF を指定した場合も)、自タスクは終了することはなく、エラーコード E_ILUSE を返します。自タスクを終了する場合は ext_tsk サービスコールを使用してください。

自タスクを指定したタスクが待ち状態に入り、何らかの待ち行列 につながれていた場合には、このサービスコールの実行によってその待ち行列から削除されます。しかし、指定したタスクがそれ以前に獲得したセマフォなどは解放されません。

tskid で示されたタスクが休止(DORMANT)状態にある場合は、サービスコールの戻り値としてエラーE_OBJ を返します。

本サービスコールはタスクコンテキストでのみ使用可能です。非タスクコンテキストでは使用できません。

■ 記述例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include “kernel_id.h”
 void task()
 {
         :
     ter_tsk( ID_main );
         :
 }
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr30.inc
    .GLB        task
task:
                :
    PUSHM       A0
    ter_tsk     #ID_TASK3
                :
```

chg_pri タスク優先度の変更
ichg_pri タスク優先度の変更 (ハンドラ専用)

■ C 言語 API

```
ER ercd = chg_pri( ID tskid, PRI tskpri );
ER ercd = ichg_pri( ID tskid, PRI tskpri );
```

● パラメータ

ID	tskid	対象タスク ID 番号
PRI	tskpri	対象タスク優先度

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード

■ アセンブリ言語 API

```
.include mr30.inc
chg_pri   TSKID, TSKPRI
ichg_pri  TSKID, TSKPRI
```

● パラメータ

TSKID	対象タスク ID 番号
TSKPRI	対象タスク優先度

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R3	タスク優先度
A0	対象タスク ID 番号

■エラーコード

E_OBJ	オブジェクト状態が不正(tskid のタスクが休止状態)

■ 機能説明

tskid で示されたタスクの優先度を、tskpri で示される値に変更し、その変更結果に基づいて再スケジューリングを行います。したがって、レディキューにつながれているタスク (実行状態のタスクを含む)、または優先度順の待ち行列の中のタスクに対して本サービスコールが実行された場合、対象タスクはキューの該当優先度の部分の最後尾に移動します。以前と同じ優先度を指定した場合も、同様に、そのキューの最後尾に移動します 。

タスクの優先度は、数の小さい方が高く 1 が最高優先度です。優先度として指定できる数値は最小値が 1 です。また、優先度の最大値はコンフィギュレーションファイルで指定した優先度の最大値であり、指定可能範囲は 1～255 です。例えば、コンフィギュレーションファイルで

```
system{       stack_size    = 0x100;
       priority     = 13;
};
```

と記述した場合は、指定できる優先度の範囲は 1 から 13 までです。

TSK_SELF が指定された場合は自タスクの優先度を変更します。非タスクコンテキストにおいて、tskid に TSK_SELF は指定した場合の動作は保証されません。TPRI_INI が指定された場合、タスク生成時に指定したタスクの起動時優先度に変更します。変更したタスク優先度は、タスクの終了もしくは、本サービスコールが再度実行されるまで有効です。

tskid で示されたタスクが休止(DORMANT)状態にある場合は、サービスコールの戻り値としてエラーE_OBJ を返します。MR30 では、ミューテックス機能をサポートしないため、エラーコード E_ILUSE を返すことはありません。

本サービスコールは、タスクコンテキストからは、chg_pri、非タスクコンテキストからは、ichg_pri を使用してください。

■ 記述例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include “kernel_id.h”
 void task()
 {
         :
     chg_pri( ID_task2, 2 );
         :
 }
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr30.inc
    .GLB        task
task:
                :
    pushm       A0,R3
    chg_pri     #ID_TASK3,#1
                :
```

get_pri タスク優先度の参照
iget_pri タスク優先度の参照 (ハンドラ専用)

■ C 言語 API

```
ER ercd = get_pri( ID tskid, PRI *p_tskpri );
ER ercd = iget_pri( ID tskid, PRI *p_tskpri );
```

● パラメータ

ID	tskid	対象タスク ID 番号
PRI	*p_tskpri	タスク優先度を返す領域へのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード

■ アセンブリ言語 API

```
.include mr30.inc
get_pri   TSKID
iget_pri  TSKID
```

● パラメータ

TSKID	対象タスク ID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
A0	獲得したタスク優先度

■エラーコード

E_OBJ	オブジェクト状態が不正(tskid のタスクが休止状態)

■ 機能説明

tskid で示されたタスクの優先度を、p_tskpri で示される領域に返します。TSK_SELF が指定された場合は自タスクの優先度を参照します。非タスクコンテキストにおいて、tskid に TSK_SELF は指定した場合の動作は保証されません。

tskid で示されたタスクが休止(DORMANT)状態にある場合は、サービスコールの戻り値としてエラーE_OBJ を返します。

本サービスコールは、タスクコンテキストからは、get_pri、非タスクコンテキストからは、iget_pri を使用してください。

■ 使用例

《 C 言語の使用例 》

```
  #include <itron.h>
  #include <kernel.h>
  #include “kernel_id.h”
  void task()
  {
     PRI p_tskpri;
     ER ercd;
           :
     ercd = get_pri( ID_task2, &p_tskpri );
           :
  }
```

《 アセンブリ言語の使用例 》

```
     .INCLUDE    mr30.inc
     .GLB        task
task:
                 :
     PUSHM       A0
     get_pri     #ID_TASK2
                 :
```

ref_tsk タスクの状態参照
iref_tsk タスクの状態参照 (ハンドラ専用)

■ C 言語 API

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

● パラメータ

ID	tskid	対象タスク ID 番号
T_RTSK	*pk_rtsk	タスク状態を返すパケットへのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)

pk_rtsk の内容

```
typedef   struct t_rtsk{
    STAT    tskstat     +0    2    タスク状態
    PRI     tskpri      +2    2    タスクの現在優先度
    PRI     tskbpri     +4    2    タスクのベース優先度
    STAT    tskWAITING  +6    2    待ち要因
    ID      wobjid      +8    2    待ちオブジェクト ID
    TMO     lefttmo     +10   4    タイムアウトするまでの時間
    UINT    actcnt      +14   2    起動要求キューイング数
    UINT    wupcnt      +16   2    起床要求キューイング数
    UINT    suscnt      +18   2    強制待ち要求ネスト数
} T_RTSK;
```

■ アセンブリ言語 API

```
.include mr30.inc
ref_tsk   TSKID, PK_RTSK
iref_tsk  TSKID, PK_RTSK
```

● パラメータ

TSKID	対象タスク ID 番号
PK_RTSK	タスク状態を返すパケットへのポインタ

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象タスク ID 番号
A1	タスク状態を返すパケットへのポインタ

■エラーコード

なし

■ 機能説明

tskid で示されたタスクの状態を参照し、そのタスクの現在の情報を pk_rtsk の指す領域にリターンパラメータとして返します。tskid として TSK_SELF が指定された場合は、自タスクの状態を参照します。非タスクコンテキストにおいて、tskid に TSK_SELF は指定した場合の動作は保証されません。

◆ tskstat(タスク状態)

tskstat には指定したタスクの状態によって次の値が返されます。

- TTS_RUN(0x0001) 実行(RUNNING)状態
- TTS_RDY(0x0002) 実行可能(READY)状態
- TTS_WAI(0x0004) 待ち(WAITING)状態
- TTS_SUS(0x0008) 強制待ち(SUSPENDED)状態
- TTS_WAS(0x000C) 二重待ち(WAITING-SUSPENDED)状態
- TTS_DMT(0x0010) 休止(DORMANT)状態

◆ tskpri(タスクの現在優先度)

tskpri には、指定したタスクの現在優先度を返します。タスクが休止状態の場合は、tskpri は、不定となります。

◆ tskbpri(タスクのベース優先度)

tskpri には、指定したタスクのベース優先度を返します。MR30 は、ミューテックス機能をサポートしていないため、tskpri と tskbpri は、同じ値となります。また、タスクが休止状態の場合は、tskpri は、不定となります。

◆ tskWAITING(タスクの待ち要因)

対象タスクが待ち状態であるならば次の待ち要因が返されます。各待ち要因の値を以下に示します。タスク状態が、待ち状態(TTS_WAI、または TTS_WAS)以外の場合は、tskWAITING は不定となります。

- TTW_SLP (0x0001) slp_tsk, tslp_tsk による待ち
- TTW_DLY (0x0002) dly_tsk による待ち
- TTW_SEM (0x0004) wai_sem, twai_sem による待ち
- TTW_FLG (0x0008) wai_flg, twai_flg による待ち
- TTW_SDTQ(0x0010) snd_dtq, tsnd_dtq による待ち
- TTW_RDTQ(0x0020) rcv_dtq, trcv_dtq による待ち
- TTW_MBX (0x0040) rcv_mbx, trcv_mbx による待ち
- TTW_MPF (0x2000) get_mpf, tget_mpf による待ち
- TTW_VSDTQ (0x4000) vsnd_dtq, vtsnd_dtqによる待ち [20]
- TTW_VRDTQ(0x8000) vrcv_dtq, vtrcv_dtq による待ち

◆ wobjid(待ちオブジェクト ID)

対象タスクが待ち状態(TTS_WAI または、TTS_WAS)であるならば、待ち対象オブジェクト ID を返します。それ以外の場合は、wobjid は、不定となります。

◆ lefttmo(タイムアウトまでの時間)

対象タスクが dly_tsk 以外による待ち状態(TTS_WAI,または TTS_WAS)の場合、タイムアウトするまでの時間を返します。永久待ちの場合は、TMO_FEVR を返します。それ以外の状態の場合は、lefttmo は不定となります。

◆ actcnt(起動要求カウント)

現在の起動要求キューイング数を返します。

◆ wupcnt(起床要求カウント)

現在の起床要求キューイング数を返します。タスクが休止状態の場合は、wupcnt は、不定となります。

◆ suscnt(強制待ち要求カウント)

現在の強制待ち要求ネスト数を返します。タスクが休止状態の場合は、suscnt は、不定となります。

本サービスコールは、タスクコンテキストからは、ref_tsk、非タスクコンテキストからは、iref_tsk を使用してください。

[20] TTW_VSDTQ,TTW_VRDTQ は μITRON 仕様 V.4.0 の仕様外の待ち要因です。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include “kernel_id.h”
 void task()
 {
     T_RTSK rtsk;
     ER ercd;
          :
     ercd = ref_tsk( ID_main, &rtsk );
          :
 }
```

《 アセンブリ言語の使用例 》

```
_refdata:       .blkb   20
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       A0,A1
    ref_tsk     #TSK_SELF,#_refdata
          :
```

ref_tst タスクの状態参照(簡易版)
iref_tst タスクの状態参照 (簡易版、ハンドラ専用)

■ C 言語 API

```
ER ercd = ref_tst( ID tskid, T_RTST *pk_rtst );
ER ercd = iref_tst( ID tskid, T_RTST *pk_rtst );
```

● パラメータ

ID	tskid	対象タスク ID 番号
T_RTST	*pk_rtst	タスク状態を返すパケットへのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)

pk_rtsk の内容

```
typedef   struct t_rtst{
    STAT    tskstat     +0    2     タスク状態
    STAT    tskWAITING  +2    2     待ち要因
} T_RTST;
```

■ アセンブリ言語 API

```
.include mr30.inc
ref_tst   TSKID, PK_RTST
iref_tst  TSKID, PK_RTST
```

● パラメータ

TSKID	対象タスク ID 番号
PK_RTST	タスク状態を返すパケットへのポインタ

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象タスク ID 番号
A1	タスク状態を返すパケットへのポインタ

■エラーコード

なし

■ 機能説明

tskid で示されたタスクの状態を参照し、そのタスクの現在の情報を pk_rtst が指す領域にリターン値として返します。tskid として TSK_SELF が指定された場合は、自タスクの状態を参照します。非タスクコンテキストにおいて、tskid に TSK_SELF は指定した場合の動作は保証されません。

◆ tskstat(タスク状態)

tskstat には指定したタスクの状態によって次の値が返されます。

- TTS_RUN(0x0001) 実行(RUNNING)状態
- TTS_RDY(0x0002) 実行可能(READY)状態
- TTS_WAI(0x0004) 待ち(WAITING)状態
- TTS_SUS(0x0008) 強制待ち(SUSPENDED)状態
- TTS_WAS(0x000C) 二重待ち(WAITING-SUSPENDED)状態
- TTS_DMT(0x0010) 休止(DORMANT)状態

◆ tskWAITING(タスクの待ち要因)

対象タスクが待ち状態であるならば次の待ち要因が返されます。各待ち要因の値を以下に示します。タスク状態が、待ち状態(TTS_WAI、または TTS_WAS)以外の場合は、tskWAITING は不定となります。

- TTW_SLP (0x0001) slp_tsk, tslp_tsk による待ち
- TTW_DLY (0x0002) dly_tsk による待ち
- TTW_SEM (0x0004) wai_sem, twai_sem による待ち
- TTW_FLG (0x0008) wai_flg, twai_flg による待ち
- TTW_SDTQ(0x0010) snd_dtq, tsnd_dtq による待ち
- TTW_RDTQ(0x0020) rcv_dtq, trcv_dtq による待ち
- TTW_MBX (0x0040) rcv_mbx, trcv_mbx による待ち
- TTW_MPF (0x2000) get_mpf, tget_mpf による待ち
- TTW_VSDTQ (0x4000) vsnd_dtq, vtsnd_dtqによる待ち [21]
- TTW_VRDTQ(0x8000) vrcv_dtq, vtrcv_dtq による待ち

本サービスコールは、タスクコンテキストからは、ref_tst、非タスクコンテキストからは、iref_tst を使用してください。

[21] TTW_VSDTQ,TTW_VRDTQ はμITRON 仕様 V.4.0 の仕様外の待ち要因です。

■ 使用例

《 C 言語の使用例 》

```
    #include <itron.h>
    #include <kernel.h>
    #include "kernel_id.h"
    void task()
    {
        T_RTST rtst;
        ER ercd;
                :
        ercd = ref_tst( ID_main, &rtst );
                :
    }
```

《 アセンブリ言語の使用例 》

```
_refdata:       .blkb   4
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       A0,A1
    ref_tst     #ID_TASK2,#_refdata
          :
```

5.2 タスク付属同期機能

表 5.3にタスク付属同期機能の仕様を示します。

表 5.3 タスク付属同期機能の仕様

項番	項目	内容
1	タスク起床要求カウントの最大値	15 回
2	タスク強制待ち要求ネスト数の最大値	1 回

表 5.4 タスク付属同期機能サービスコール一覧

項番	サービスコール		機能	呼び出し可能なシステム状態					
				T	N	E	D	U	L
1	slp_tsk	**[S]**	起床待ち	○		○		○	
2	tslp_tsk	**[S]**	同上(タイムアウト有)	○		○		○	
3	wup_tsk	**[S]**	タスクの起床	○		○	○	○	
4	iwup_tsk	**[S]**			○	○	○	○	
5	can_wup		タスク起床要求のキャンセル	○		○	○	○	
6	ican_wup				○	○	○	○	
7	rel_wai	**[S]**	待ち状態の強制解除	○		○	○	○	
8	irel_wai	**[S]**			○	○	○	○	
9	sus_tsk	**[S]**	強制待ち状態への移行	○		○	○	○	
10	isus_tsk				○	○	○	○	
11	rsm_tsk	**[S]**	強制待ち状態からの再開	○		○	○	○	
12	irsm_tsk				○	○	○	○	
13	frsm_tsk	**[S]**	強制待ち状態からの強制再開	○		○	○	○	
14	ifrsm_tsk				○	○	○	○	
15	dly_tsk	**[S]**	自タスクの遅延	○		○		○	

【注】

- “[S]”はスタンダードプロファイルのサービスコールです。
- “呼び出し可能なシステム状態”内のそれぞれの記号は、以下の意味です。
  "T"はタスクコンテキストから呼出し可能、"N"は非タスクコンテキストから呼出し可能
  "E"はディスパッチ許可状態から呼出し可能、"D"はディスパッチ禁止状態から呼出し可能
  "U"は CPU ロック解除状態から呼出し可能、"L"は CPU ロック状態から呼出し可能

slp_tsk 起床待ち
tslp_tsk 起床待ち (タイムアウト)

■ C 言語 API

```
ER ercd = slp_tsk();
ER ercd = tslp_tsk( TMO tmout );
```

● パラメータ

- slp_tsk の場合
  なし
- tslp_tsk の場合

TMO	tmout	タイムアウト値

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード

■ アセンブリ言語 API

```
.include mr30.inc
slp_tsk
tslp_tsk[22]
```

● パラメータ

なし

● サービスコール発行後のレジスタ内容

tslp_tsk の場合

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R1	タイムアウト値(下位 16bit)
R3	タイムアウト値(上位 16bit)

slp_tsk の場合

レジスタ名	サービスコール発行後の内容
R0	エラーコード

■エラーコード

E_TMOUT	タイムアウト
E_RLWAI	強制待ち解除

[22] R3(タイムアウト値上位 16bit)、R1(タイムアウト値下位 16bit)を格納して呼び出す必要があります。

■ 機能説明

自タスクを実行(RUNNING)状態から起床待ち状態へ移行します。本サービスコール実行による待ち状態は、以下に示す場合に解除されます。

◆ 他タスクおよび割り込みからタスク起床のサービスコール を発行した場合

この時のエラーコードは、E_OK が返ります。

◆ 他タスクおよび割り込みから待ち状態強制解除のサービスコール を発行した場合

この時のエラーコードは、E_RLWAI が返ります。

◆ tmout 経過後、最初のタイムティックが発生した場合(tslp_tsk の場合)

この時のエラーコードは、E_TMOUT が返ります。

本サービスコールにより待ち(WAITING)状態となっているときに他のタスクから sus_tsk されるとそのタスクの状態は二重待ち(WAITING-SUSPENDED)状態になります。この場合はタスク起床のサービスコールにより待ち状態が解除されても、まだ強制待ち状態であり、rsm_tsk の発行まで、タスクの実行は再開されません。

tslp_tsk では、引数に tmout を指定し一定時間自タスクを起床待ち状態に移行させることが出来ます。tmout の単位は、msです。すなわち、tslp_tsk(10);と記述すれば、10ms 間、自タスクが実行(RUNNING)状態から待ち(WAITING)状態へ移行します。tmout=TMO_FEVR(-1)にした場合は、永久待ちの指定となり、slp_tsk サービスコールと同じ動作を行います。

tmout に指定可能な値は、(0x7FFFFFFF-タイムティック)以内でなければいけません。これより大きな値を指定した場合の動作は、保証されません。

本サービスコールはタスクコンテキストからのみ発行してください。非タスクコンテキストから発行することはできません。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
                :
    if( slp_tsk() != E_OK )
        error("Forced wakeup¥n");
                :
    if( tslp_tsk( 10 ) == E_TMOUT )
        error("time out¥n");
                :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
    slp_tsk
          :
    PUSHM       R1,R3
    tslp_tsk    #TMO_FEVR
          :
    PUSHM       R1,R3
    MOV.W       #100,R1
    MOV.W       #0,R3
    tslp_tsk
          :
```

wup_tsk タスクの起床
iwup_tsk タスクの起床 (ハンドラ専用)

■ C 言語 API

```
ER ercd = wup_tsk( ID tskid );
ER ercd = iwup_tsk( ID tskid );
```

● パラメータ

ID	tskid	対象タスク ID 番号

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード

■ アセンブリ言語 API

```
.include mr30.inc
wup_tsk   TSKID
iwup_tsk  TSKID
```

● パラメータ

TSKID	対象タスク ID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
A0	対象タスク ID 番号

■エラーコード

E_OBJ	オブジェクト状態が不正(tskid のタスクが休止状態)
E_QOVR	キューイングのオーバーフロー

■ 機能説明

tskid で指定したタスクが slp_tsk あるいは tslp_tsk の実行による待ち(WAITING)状態であれば、待ちを解除して実行可能(READY)状態もしくは実行(RUNNING)状態に移行します。また、tskid で指定したタスクが二重待ち(WAITING-SUSPENDED)状態である時は、待ちのみを解除して強制待ち(SUSPENDED)状態に移行します。

対象タスクが休止(DORMANT)状態にある場合に発せられた要求に対しては、サービスコール発行タスクにエラーE_OBJ を返します。tskid に TSK_SELF が指定された場合は、自タスク指定となります。非タスクコンテキストにおいて、tskid に TSK_SELF は指定した場合の動作は保証されません。

slp_tsk あるいは tslp_tsk サービスコール実行による待ち(WAITING)状態もしくは二重待ち(WAITING-SUSPENDED)状態にないタスクに対して本サービスコールを行なった場合は、起床要求が蓄積されます。すなわち、起床要求対象タスクの起床要求カウントを 1 つ増やすことにより起床要求を蓄積します。
起床要求カウントの最大値は 15 です。起床要求カウントが 15 の時に、これを越えて起床要求を発生させると起床要求カウントは 15 のままで本サービスコールの発行タスクには、エラーコード E_QOVR を返します。
本サービスコールは、タスクコンテキストからは、wup_tsk、非タスクコンテキストからは、iwup_tsk を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include “kernel_id.h”
 void task()
 {
                :
   if( wup_tsk( ID_main ) != E_OK )
       printf(“Can’t wakeup main()¥n”);
                :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       A0
    wup_tsk     #ID_TASK1
          :
```

can_wup 起床要求のキャンセル
ican_wup 起床要求のキャンセル (ハンドラ専用)

■ C 言語 API

```
ER_UINT wupcnt = can_wup( ID tskid );
ER_UINT wupcnt = icam_wup( ID tskid );
```

● パラメータ

ID	tskid	対象タスク ID 番号

● リターンパラメータ

ER_UINT	wupcnt > 0	キャンセルされた起床要求カウント
	wupcnt = 0	
	wupcnt <0	エラーコード

■ アセンブリ言語 API

```
.include mr30.inc
can_wup   TSKID
ican_wup  TSKID
```

● パラメータ

TSKID	対象タスク ID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	エラーコードまたはキャンセルされた起床要求カウント
A0	対象タスク ID 番号

■エラーコード

E_OBJ	オブジェクト状態が不正(tskid のタスクが休止状態)

■ 機能説明

tskid で示された対象タスクの起床要求カウントを 0(ゼロ)クリアします。すなわち、本サービスコール発行以前に wup_tsk、iwup_tsk サービスコールにより起床しようとした時に対象タスクが待ち(WAITING)状態もしくは二重待ち(WAITING-SUSPENDED)状態でないために起床要求のみが蓄積されていたのをすべて無効にします。

また、本サービスコールの戻り値として 0(ゼロ)クリアする前の起床要求カウント、すなわち無効になった起床要求回数(wupcnt)が返されます。対象タスクが休止(DORMANT)状態に発せられた要求に対しては、サービスコール発行タスクにエラーE_OBJ を返します。tskid に TSK_SELF が指定された場合は、自タスク指定となります。非タスクコンテキストにおいて、tskid に TSK_SELF は指定した場合の動作は保証されません。

本サービスコールは、タスクコンテキストからは、can_wup、非タスクコンテキストからは、ican_wup を使用してください。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include “kernel_id.h”
void task()
{
    ER_UINT wupcnt;
              :
    wupcnt = can_wup(ID_main);
    if( wup_cnt  > 0 )
        printf(“wupcnt = %d¥n”,wupcnt);
              :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       A0
    can_wup     #ID_TASK3
          :
```

rel_wai 待ち状態の強制解除
irel_wai 待ち状態の強制解除 (ハンドラ専用)

■ C 言語 API

```
ER ercd = rel_wai( ID tskid );
ER ercd = irel_wai( ID tskid );
```

● パラメータ

ID	tskid	対象タスク ID 番号

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード

■ アセンブリ言語 API

```
.include mr30.inc
rel_wai   TSKID
irel_wai  TSKID
```

● パラメータ

TSKID	対象タスク ID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
A0	対象タスク ID 番号

■エラーコード

E_OBJ	オブジェクト状態が不正(tskid のタスクが待ち状態でない)

■ 機能説明

tskid で示されたタスクの待ち状態(強制待ち(SUSPENDED)状態を除く)を、強制的に解除し、実行可能(READY)状態あるいは実行(RUNNING)状態に移行します。強制的に解除されたタスクにはエラーコードE_RLWAI を返します。対象タスクが何らかの待ち行列 につながれていた場合には、本サービスコールの実行によってその待ち行列から削除されます。

二重待ち状態(WAITING-SUSPENDED)のタスクに対して、本サービスコールを発行した場合、対象タスクの待ち状態は解除され、強制待ち状態(SUSPENDED)に移行します。[23]

対象タスクが待ち状態にない場合は、サービスコール発行タスクにエラーE_OBJ を返します。また、本サービスコールは、自タスクを指定できません。

本サービスコールは、タスクコンテキストからは、rel_wai、非タスクコンテキストからは、irel_wai を使用してください。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
            :
    if( rel_wai( ID_main ) != E_OK )
        error("Can't rel_wai main()¥n");
         :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
         :
    PUSHM       A0
    rel_wai     #ID_TASK2
         :
```

[23] 強制待ち状態は、本サービスコールにより待ちは解除されません。強制待ち状態は、rsm_tsk,irsm_tsk,frsm_tsk,ifrsm_tsk サービスコールによって待ちが解除されます。

sus_tsk 強制待ち状態への移行
isus_tsk 強制待ち状態への移行 (ハンドラ専用)

■ C 言語 API

```
ER ercd = sus_tsk( ID tskid );
ER ercd = isus_tsk( ID tskid );
```

● パラメータ

ID	tskid	対象タスク ID 番号

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード

■ アセンブリ言語 API

```
.include mr30.inc
sus_tsk   TSKID
isus_tsk  TSKID
```

● パラメータ

TSKID	対象タスク ID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
A0	対象タスク ID 番号

■エラーコード

E_OBJ	オブジェクト状態が不正(tskid のタスクが休止状態)
E_QOVR	キューイングのオーバーフロー

■ 機能説明

tskid で示されたタスクの実行を中断させ、強制待ち(SUSPENDED)状態へ移行します。強制待ち状態は、rsm_tsk, irsm_tsk, frsm_tsk, ifrsm_tsk サービスコールの発行によって解除されます。tskid で示された対象タスクが休止(DORMANT)状態にある場合は、サービスコールの戻り値としてエラーE_OBJ を返します。

本サービスコールによる強制待ち要求のネスト数の最大値は1です。強制待ち状態のタスクに対して本サービスコールを発行した場合は、エラーE_QOVR を返します。

本サービスコールで自タスクを指定することはできません。

本サービスコールは、タスクコンテキストからは、sus_tsk、非タスクコンテキストからは、isus_tsk を使用してください。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
                :
    if( sus_tsk( ID_main ) != E_OK )
        printf("Can't SUSPENDED task main()¥n");
                :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       A0
    sus_tsk     #ID_TASK2
          :
```

rsm_tsk	強制待ち状態の解除
irsm_tsk	強制待ち状態の解除 (ハンドラ専用)
frsm_tsk	強制待ち状態の強制解除
ifrsm_tsk	強制待ち状態の強制解除 (ハンドラ専用)

■ C 言語 API

```
ER ercd = rsm_tsk( ID tskid );
ER ercd = irsm_tsk( ID tskid );
ER ercd = frsm_tsk( ID tskid );
ER ercd = ifrsm_tsk( ID tskid );
```

● パラメータ

ID	tskid	対象タスク ID 番号

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード

■ アセンブリ言語 API

```
.include mr30.inc
rsm_tsk   TSKID
irsm_tsk  TSKID
frsm_tsk  TSKID
ifrsm_tsk TSKID
```

● パラメータ

TSKID	対象タスク ID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
A0	対象タスク ID 番号

■エラーコード

E_OBJ	オブジェクト状態が不正(tskid のタスクが強制待ち状態でない)

■ 機能説明

tskid で示されたタスクが sus_tsk サービスコールによって中断されている場合、対象タスクの強制待ち状態を解除し、実行を再開します。このとき、対象タスクはレディキューの最後尾につながれます。対象タスクの強制待ち状態の場合は、強制待ち状態を解除します。

対象タスクが強制待ち(SUSPENDED)状態にない場合(休止(DORMANT)状態を含む)に発せられた要求に対してはサービスコール発行タスクにエラーE_OBJ を返します。

rsm_tsk, irsm_tsk, frsm_tsk, ifrsm_tsk サービスコールいずれにおいても、強制待ち要求の最大ネスト数が1であるため、同じ動作をします。

本サービスコールは、タスクコンテキストからは、rsm_tsk, frsm_tsk、非タスクコンテキストからは、irsm_tsk, ifrsm_tsk を使用してください。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task1()
{
                :
    if( rsm_tsk( ID_main ) != E_OK )
        printf("Can't resume main()¥n");
                :
                :
    if(frsm_tsk( ID_task2 ) != E_OK )
        printf("Can't forced resume task2()¥n");
                :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       A0
    rsm_tsk     #ID_TASK2
          :
    PUSHM       A0
    frsm_tsk    #ID_TASK1
          :
```

dly_tsk タスクの遅延

■ C 言語 API

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

● パラメータ

RELTIM	dlytim	遅延時間

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード

■ アセンブリ言語 API

```
.include mr30.inc
dly_tsk[24]
```

● パラメータ

なし

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R1	遅延時間(下位 16bit)
R3	遅延時間(上位 16bit)

■エラーコード

E_RLWAI	強制待ち解除

■ 機能説明

自タスクの実行を、dlytim で指定した時間だけ一時的に停止し、実行(RUNNING)状態から待ち状態へ移行します。具体的には、dlytim で指定した時間経過後の最初のタイムティックで待ち状態が解除されます。そのため、dlytim に 0 が指定された場合は、いったん待ち状態に移行し、最初のタイムティックで待ち状態が解除されます。

本サービスコール発行による待ち状態は、以下に示す場合に解除されます。なお、待ち状態が解除されると本サービスコールを発行したタスクは、タイムアウト待ち行列からはずされ、レディキューに接続されます。

◆ dlytim 経過後、最初のタイムティックが発生した場合場合

この時のエラーコードは、E_OK を返します。

◆ dlytim の時間が経過する前に前に rel_wai、irel_wai サービスコールを発行した場合

この時のエラーコードは、E_RLWAI を返します。

なお、遅延時間中に wup_tsk, iwup_tsk サービスコールが発行されても、待ち解除とはなりません。

dlytim の単位は、ms です。すなわち、dly_tsk(50);と記述すれば、50ms 間、自タスクが実行(RUNNING)状態から遅延待ち状態へ移行します。

dlytim に指定可能な値は、(0x7FFFFFFF-タイムティック)以内でなければいけません。これより大きな値を指定した場合は、正しく動作しません。

本サービスコールはタスクコンテキストからのみ発行してください。非タスクコンテキストから発行した場合は正常に動作しません。

[24] R3(遅延時間上位 16bit)、R1(遅延時間下位 16bit)を格納して呼び出す必要があります。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include “kernel_id.h”
void task()
{
                :
    if( dly_tsk() != E_OK )
        error(“Forced wakeup¥n”);
                :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       R1,R3
    MOV.W       #500,R1
    MOV.W       #0,R3
    dly_tsk
          :
```

5.3 同期･通信機能(セマフォ)

表 5.5にセマフォ機能の仕様を示します。

表 5.5 セマフォ機能の仕様

項番	項目	内容
1	セマフォ ID	1-255
2	最大資源数	1-65535
3	セマフォ属性	TA_FIFO ： タスクキューFIFO 順 TA_TPRI ： タスクキュー優先度順

表 5.6 セマフォ機能サービスコール一覧

項番	サービスコール		機能	呼び出し可能なシステム状態					
				T	N	E	D	U	L
1	sig_sem	**[S]**	セマフォ資源の返却	○		○	○	○	
2	isig_sem	**[S]**			○	○	○	○	
3	wai_sem	**[S]**	セマフォ資源の獲得	○		○		○	
4	pol_sem	**[S]**	同上(ポーリング)	○		○	○	○	
5	ipol_sem				○	○	○	○	
6	twai_sem	**[S]**	同上(タイムアウト有)	○		○		○	
7	ref_sem		セマフォの状態参照	○		○	○	○	
8	iref_sem				○	○	○	○	

【注】

- “[S]”はスタンダードプロファイルのサービスコールです。
- “呼び出し可能なシステム状態”内のそれぞれの記号は、以下の意味です。
  "T"はタスクコンテキストから呼出し可能、"N"は非タスクコンテキストから呼出し可能
  "E"はディスパッチ許可状態から呼出し可能、"D"はディスパッチ禁止状態から呼出し可能
  "U"は CPU ロック解除状態から呼出し可能、"L"は CPU ロック状態から呼出し可能

sig_sem セマフォ資源の返却
isig_sem セマフォ資源の返却 (ハンドラ専用)

■ C 言語 API

```
ER ercd = sig_sem( ID semid );
ER ercd = isig_sem( ID semid );
```

● パラメータ

ID	semid	対象セマフォ ID 番号

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード

■ アセンブリ言語 API

```
.include mr30.inc
sig_sem   SEMID
isig_sem  SEMID
```

● パラメータ

SEMID	対象セマフォ ID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
A0	対象セマフォ ID 番号

■ エラーコード

E_QOVR	キューイングオーバーフロー

■ 機能説明

semid で示されたセマフォに対して、資源を 1 つ返却します。

対象セマフォの待ち行列にタスクがつながれている場合には、行列の先頭タスクを実行可能(READY)状態へ移行します。一方、待ち行列にタスクがつながれていない場合には、そのセマフォの計数値を 1 だけ増やします 。セマフォの計数値がコンフィギュレーションファイルで指定した最大値 (maxsem)を越えて資源の返却(sig_sem、isig_sem サービスコール)をおこなうとセマフォの計数値はそのままで、サービスコール発行タスクにエラーE_QOVR を返します。

本サービスコールは、タスクコンテキストからは、sig_sem、非タスクコンテキストからは、isig_sem を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include “kernel_id.h”
 void task()
 {
                :
     if( sig_sem( ID_sem ) == E_QOVR )
       error(“Overflow¥n”);
                :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       A0
    sig_sem     #ID_SEM2
          :
```

wai_sem セマフォ資源の獲得
pol_sem セマフォ資源の獲得 (ポーリング)
ipol_sem セマフォ資源の獲得 (ポーリング、ハンドラ専用)
twai_sem セマフォ資源の獲得 (タイムアウト)

■ C 言語 API

```
ER ercd = wai_sem( ID semid );
ER ercd = pol_sem( ID semid );
ER ercd = ipol_sem( ID semid );
ER ercd = twai_sem( ID semid, TMO tmout );
```

● パラメータ

ID	semid	対象セマフォ ID 番号
TMO	tmout	タイムアウト値(twai_sem の場合)

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード

■ アセンブリ言語 API

```
.include mr30.inc
wai_sem   SEMID
pol_sem   SEMID
ipol_sem  SEMID
twai_sem  SEMID[25]
```

● パラメータ

SEMID	セマフォ ID 番号

● サービスコール発行後のレジスタ内容

wai_sem, pol_sem, ipol_sem の場合

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
A0	対象セマフォ ID 番号

twai_sem の場合

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R1	タイムアウト値(下位 16bit)
R3	タイムアウト値(上位 16bit)
A0	対象セマフォ ID 番号

■ エラーコード

E_RLWAI	待ち状態強制解除
E_TMOUT	ポーリング失敗、またはタイムアウト

[25] R3(タイムアウト値上位 16bit)、R1(タイムアウト値下位 16bit)を格納して呼び出す必要があります。

■ 機能説明

semid で示されたセマフォから、資源を一つ獲得する操作を行います。

そのセマフォの計数値が 1 以上の場合には、計数値を 1 だけ減じてサービスコール発行タスクは実行を継続します。一方、セマフォの計数値が 0 の場合には、wai_sem,twai_sem サービスコール呼び出しタスクは、そのセマフォ待ち行列につなぎます。semid のセマフォ属性が TA_TFIFO の場合は、待ち行列に FIFO 順でタスクをつなぎます。TA_TPRI の場合は、待ち行列に優先度順でタスクをつなぎます。pol_sem,ipol_sem サービスコールでは、直ちにﾘﾀｰﾝし、エラーE_TMOUT を返します。

twai_sem サービスコールの場合は、tmout には、待ち時間を ms 単位で指定します。tmout に指定可能な値は、(0x7FFFFFFF-タイムティック)以内でなければいけません。これより大きな値を指定した場合の動作は、保証されません。tmout に TMO_POL=0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、pol_sem と同じ動作をします。また、tmout=TMO_FEVR(-1)にした場合は、永久待ちの指定で、wai_sem サービスコールと同じ動作をします。

wai_sem, twai_sem サービスコール実行による待ち状態は、以下に示す場合に解除されます。

- **tmout の時間が経過する前に、sig_sem、isig_sem サービスコールが発行され、待ち解除条件が満足された場合**
  この場合、エラーコードは、E_OK を返します。
- **待ち解除条件が満足されないまま、tmout 経過し、最初のタイムティックが発生した場合**
  この場合、エラーコードは、E_TMOUT を返します。
- **他のタスクおよびハンドラから発行した rel_wai、irel_wai サービスコールによって待ち状態が強制解除された場合**
  この場合、エラーコードは、E_RLWAI を返します。

タスクコンテキストにおいては、wai_sem, twai_sem, pol_sem サービスコール、非タスクコンテキストにおいては、ipol_sem を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
                :
   if( wai_sem( ID_sem ) != E_OK )
       printf("Forced wakeup¥n");
                :
   if( pol_sem( ID_sem ) != E_OK )
       printf("Timeout¥n");
                :
     if( twai_sem( ID_sem, 10 ) != E_OK )
         printf("Forced wakeup or Timeout"n");
                :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       A0
    pol_sem     #ID_SEM1
          :
    PUSHM       A0
    wai_sem     #ID_SEM2
          :
    PUSHM       A0,R1,R3
    MOV.W       #300,R1
    MOV.W       #0,R3
    twai_sem    #ID_SEM3
          :
```

ref_sem セマフォの状態参照
iref_sem セマフォの状態参照 (ハンドラ専用)

■ C 言語 API

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

● パラメータ

ID	semid	対象セマフォ ID 番号
T_RSEM	*pk_rsem	セマフォ状態を返すパケットへのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)
T_RSEM	*pk_rsem	セマフォ状態を返すパケットへのポインタ

pk_rsem の内容

```
typedef   struct t_rsem{
    ID      wtskid    +0    2     待ちタスク ID
    UINT    semcnt    +2    2     現在のセマフォカウンタ値
} T_RSEM;
```

■ アセンブリ言語 API

```
.include mr30.inc
ref_sem   SEMID, PK_RSEM
iref_sem  SEMID, PK_RSEM
```

● パラメータ

SEMID	対象セマフォ ID 番号
PK_RSEM	セマフォ状態を返すパケットへのポインタ

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象セマフォ ID 番号
A1	セマフォ状態を返すパケットへのポインタ

■エラーコード

なし

■ 機能説明

semid で示されたセマフォの各種の状態を返します。

- **wtskid**
  wtskid には待ち行列の先頭タスク(次に待ち行列から削除されるタスク)の ID 番号を返します。待ちタスクの無い場合は TSK_NONE を返します。
- **semcnt**
  semcnt には、現在のセマフォカウント値を返します。

本サービスコールは、タスクコンテキストからは、ref_sem、非タスクコンテキストからは、iref_sem を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include “kernel_id.h”
 void task()
 {
     T_RSEM rsem;
     ER ercd;
         :
     ercd = ref_sem( ID_sem1, &rsem );
         :
 }
```

《 アセンブリ言語の使用例 》

```
_ refsem:       .blkb   4
    .include mr30.inc
    .GLB        task
task:
         :
        PUSHM   A0,A1
        ref_sem #ID_SEM1,#_refsem
         :
```

5.4 同期･通信機能(イベントフラグ)

表 5.7にｲﾍﾞﾝﾄﾌﾗｸﾞ機能の仕様を示します。

表 5.7 イベントフラグ機能の仕様

項番	項目	内容	
1	ｲﾍﾞﾝﾄﾌﾗｸﾞ ID	1-255	
2	ｲﾍﾞﾝﾄﾌﾗｸﾞのﾋﾞｯﾄ数	16 ﾋﾞｯﾄ	
3	ｲﾍﾞﾝﾄﾌﾗｸﾞ属性	TA_TFIFO： TA_TPRI： TA_WMUL： TA_WSGL： TA_CLR：	待ちタスクのキューイングは FIFO 順 待ちタスクのキューイングは優先度順 複数タスクの待ちを許す 複数タスクの待ちを許さない 待ちタスクを解除する時にビットパターンをクリアする

表 5.8 イベントフラグ機能サービスコール一覧

項番	サービスコール		機能	呼び出し可能なシステム状態					
				T	N	E	D	U	L
1	set_flg	**[S]**	ｲﾍﾞﾝﾄﾌﾗｸﾞのｾｯﾄ	○		○	○	○	
2	iset_flg	**[S]**			○	○	○	○	
3	clr_flg	**[S]**	ｲﾍﾞﾝﾄﾌﾗｸﾞのｸﾘｱ	○		○	○	○	
4	iclr_flg				○	○	○	○	
5	wai_flg	**[S]**	ｲﾍﾞﾝﾄﾌﾗｸﾞ待ち	○		○		○	
6	pol_flg	**[S]**	同上(ﾎﾟｰﾘﾝｸﾞ)	○		○	○	○	
7	ipol_flg	**[S]**			○	○	○	○	
8	twai_flg	**[S]**	同上(ﾀｲﾑｱｳﾄ有)	○		○		○	
9	ref_flg		ｲﾍﾞﾝﾄﾌﾗｸﾞの状態参照	○		○	○	○	
10	iref_flg				○	○	○	○	

【注】

- “[S]”はスタンダードプロファイルのサービスコールです。
- “呼び出し可能なシステム状態”内のそれぞれの記号は、以下の意味です。
  "T"はタスクコンテキストから呼出し可能、"N"は非タスクコンテキストから呼出し可能
  "E"はディスパッチ許可状態から呼出し可能、"D"はディスパッチ禁止状態から呼出し可能
  "U"は CPU ロック解除状態から呼出し可能、"L"は CPU ロック状態から呼出し可能

set_flg イベントフラグのセット
iset_flg イベントフラグのセット(ハンドラ専用)

■ C言語API

```
ER ercd = set_flg( ID flgid, FLGPTN setptn );
ER ercd = iset_flg( ID flgid, FLGPTN setptn );
```

● パラメータ

ID	flgid	対象イベントフラグID番号
FLGPTN	setptn	セットするビットパターン

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語API

```
.include mr30.inc
set_flg   FLGID, SETPTN
iset_flg  FLGID, SETPTN
```

● パラメータ

FLGID	対象イベントフラグID番号
SETPTN	セットするビットパターン

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
R3	セットするビットパターン
A0	対象イベントフラグID番号

■ エラーコード

なし

■ 機能説明

flgidで示される16ビットのイベントフラグのうち、setptnで示されているビットをセットします。つまり、flgidで示されるイベントフラグの値に対してsetptnの論理和(OR)をとります。イベントフラグ値の変更の結果、wai_flg, twai_flgサービスコールによってイベントフラグを待っていたタスクの待ち解除の条件を満たすようになれば、そのタスクの待ちを解除して実行可能(READY)状態もしくは実行(RUNNING)状態へ移行します 。

待ち解除条件は、待ち行列の先頭から順に評価されます。イベントフラグ属性として、TA_WMULが指定されている場合、イベントフラグは、一回のset_flg, iset_flgサービスコール発行によって、複数タスクの待ち状態を同時に解除することができます。また、対象のイベントフラグ属性にTA_CLR属性が指定されている場合は、イベントフラグのすべてのビットパターンをクリアし、サービスコールの処理を終了します。[26]

setptnの全ビットを0とした場合は、対象イベントフラグに対して何の操作も行いませんが、エラーとはなりません。

本サービスコールは、タスクコンテキストからは、set_flg、非タスクコンテキストからは、iset_flgを使用してください。

[26] 本サービスコールとiclr_flg,iref_flg,iref_tsk,iref_tstサービスコールの組み合わせにおいては、サービスコールの不可分性は保証されません。すなわち、本サービスコール実行中の状態に対して処理されることが発生し得ます。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include “kernel_id.h”
 void task(void)
 {
                :
   set_flg( ID_flg,(FLGPTN)0xff00 );
                :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       A0, R3
    set_flg     #ID_FLG3,#0ff00H
          :
```

clr_flg イベントフラグのクリア
iclr_flg イベントフラグのクリア (ハンドラ専用)

■ C 言語 API

```
ER ercd = clr_flg( ID flgid, FLGPTN clrptn );
ER ercd = iclr_flg( ID flgid, FLGPTN clrptn );
```

● パラメータ

ID	flgid	対象イベントフラグ ID 番号
FLGPTN	clrptn	クリアするビットパターン

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語 API

```
.include mr30.inc
clr_flg   FLGID, CLRPTN
iclr_flg  FLGID, CLRPTN
```

● パラメータ

FLGID	対象イベントフラグ ID 番号
CLRPTN	クリアするビットパターン

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象イベントフラグ ID 番号
R3	クリアするビットパターン

■ エラーコード

なし

■ 機能説明

flgid で示される 16 ビットイベントフラグのうち、対応する clrptn の 0 になっているビットをクリアします。つまり、flgid で示されるイベントフラグビットパターンを、clrptn 値との論理積(AND)に更新します。clrptn の全ビットを 1 とした場合、イベントフラグに対して何の操作も行なわないことになりますが、エラーにはなりません。

本サービスコールは、タスクコンテキストからは、clr_flg、非タスクコンテキストからは、iclr_flg を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task(void)
 {
                :
   clr_flg( ID_flg,(FLGPTN) 0xf0f0);
                :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       A0, R3
    clr_flg     #ID_FLG1,#0f0f0H
          :
```

wai_flg イベントフラグ待ち
pol_flg イベントフラグ待ち (ポーリング)
ipol_flg イベントフラグ待ち (ポーリング、ハンドラ専用)
twai_flg イベントフラグ待ち (タイムアウト)

■ C言語API

```
ER ercd = wai_flg( ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn );
ER ercd = pol_flg( ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn );
ER ercd = ipol_flg( ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn );
ER ercd = twai_flg( ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn,
          TMO tmout );
```

● パラメータ

ID	flgid	対象イベントフラグ ID 番号
FLGPTN	waiptn	待ちビットパターン
MODE	wfmode	待ちモード
FLGPTN	*p_flgptn	待ち解除時のビットパターンを返す領域へのポインタ
TMO	tmout	タイムアウト値(twai_flg の場合)

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード
FLGPTN	*p_flgptn	待ち解除時のビットパターンを返す領域へのポインタ

■ アセンブリ言語API

```
.include mr30.inc
wai_flg   FLGID, WAIPTN, WFMODE
pol_flg   FLGID, WAIPTN, WFMODE
ipol_flg  FLGID, WAIPTN, WFMODE
twai_flg  FLGID, WAIPTN, WFMODE[27]
```

● パラメータ

FLGID	対象イベントフラグ ID 番号
WAIPTN	待ちビットパターン
WFMODE	待ちモード

[27] R2(タイムアウト値上位 16bit)、R0(タイムアウト値下位 16bit)を格納して呼び出す必要があります。

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R1	待ちモード
R2	待ち解除時のビットパターン
R3	待ちビットパターン
A0	対象イベントフラグ ID 番号

■ エラーコード

E_RLWAI	待ち状態強制解除
E_TMOUT	ポーリング失敗、またはタイムアウト
E_ILUSE	サービスコール不正使用 (TA_WSGL 属性のイベントフラグに待ちタスクが存在)

■ 機能説明

flgid で示されるイベントフラグにおいて、waiptn で指定したビットが wfmode で示される待ち解除条件にしたがってセットされるのを待ちます。p_flgptn の指す領域には、待ち解除されるときのイベントフラグのビットパターンを返します。

対象イベントフラグが TA_WSGL 属性の場合、既に他のタスクが待っている場合は、エラーE_ILUSE を返します。

本サービスコール呼び出し時にすでに待ち解除条件を満たしている場合は、すぐにリターンし、E_OK を返します。待ち解除条件を満たしていない時、wai_flg,twai_flg サービスコールの場合は、イベントフラグ待ち行列につながれます。その際、flgid のイベントフラグ属性が TA_TFIFO の場合は、FIFO 順で待ち行列にタスクをつなぎ、TA_TPRI の場合は、優先度順でタスクをつなぎます。pol_flg,ipol_flg サービスコールの場合は、直ちにリターンし、エラーE_TMOUT を返します。

twai_flg サービスコールの場合は、tmout に待ち時間を ms 単位で指定します。tmout に指定可能な値は、(0x7FFFFFFF-タイムティック)以内でなければいけません。これより大きな値を指定した場合は、正しく動作しません。

tmout に TMO_POL(=0)を指定した場合は、タイムアウト値として 0 を指定したことを示し、pol_flg と同じ動作をします。また、tmout=TMO_FEVR(-1)にした場合は、永久待ちの指定で、wai_flg サービスコールと同じ動作になります。

wai_flg,twai_flg サービスコール実行による待ち状態は、以下に示す場合に解除されます。

- **tmout の時間が経過する前に、待ち解除条件が成立した場合**
  この時のエラーコードは、E_OK を返します。
- **待ち解除条件が満たされないまま、tmout 経過し、最初のタイムティックが発生した場合**
  この時のエラーコードは、E_TMOUT を返します。
- **他のタスクおよびハンドラから発行した rel_wai、irel_wai サービスコールによって待ち状態が強制解除された場合**
  この時のエラーコードは、E_RLWAI を返します。

wfmode の指定方法および各モードの意味は以下のとおりです。

wfmode(待ちモード)	意味
TWF_ANDW	waiptn で指定したビットが全てセットされるのを待つ(AND 待ち)。
TWF_ORW	waiptn で指定したビットのいずれかがセットされるの待つ(OR 待ち)。

タスクコンテキストにおいては、wai_flg, twai_flg, pol_flg サービスコール、非タスクコンテキストにおいては、ipol_flg を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include “kernel_id.h”
 void task()
 {
   UINT flgptn;
              :
     if(wai_flg(ID_flg2, (FLGPTN)0x0ff0, TWF_ANDW, &flgptn)!=E_OK)
       error(“WAITING Released¥n”);
              :
              :
     if(pol_flg(ID_flg2, (FLGPTN)0x0ff0, TWF_ORW, &flgptn)!=E_OK)
       printf(“Not set EventFlag¥n”);
              :
              :
     if( twai_flg(ID_flg2, (FLGPTN)0x0ff0, TWF_ANDW, &flgptn, 5) != E_OK )
       error(“WAITING Released¥n”);
              :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       A0,R1,R3
    wai_flg     #ID_FLG1,#0003H,#TWF_ANDW
          :
    PUSHM       A0,R1,R3
    pol_flg     #ID_FLG2,#0008H,#TWF_ORW
          :
    PUSHM       A0,R0,R1,R2,R3
    MOV.W       #20,R0
    MOV.W       #0,R2
    twai_flg    #ID_FLG3,#0003H,#TWF_ANDW
       :
```

ref_flg イベントフラグの状態参照
iref_flg イベントフラグの状態参照 (ハンドラ専用)

■ C 言語 API

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

● パラメータ

ID	flgid	対象イベントフラグ ID 番号
T_RFLG	*pk_rflg	イベントフラグ状態を返すパケットへのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)
T_RFLG	*pk_rflg	イベントフラグ状態を返すパケットへのポインタ

pk_rflg の内容

```
typedef   struct t_rflg{
    ID      wtskid    +0    2     待ちタスク ID
    FLGPTN  flgptn    +2    2     現在のイベントフラグビットパターン
} T_RFLG;
```

■ アセンブリ言語 API

```
.include mr30.inc
ref_flg   FLGID, PK_RFLG
iref_flg  FLGID, PK_RFLG
```

● パラメータ

FLGID	対象イベントフラグ ID 番号
PK_RFLG	イベントフラグ状態を返すパケットへのポインタ

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象イベントフラグ ID 番号
A1	イベントフラグ状態を返すパケットへのポインタ

■エラーコード

なし

■ 機能説明

flgid で示されたイベントフラグの各種の状態を返します。

- **wtskid**
  wtskid には待ち行列の先頭タスク(次に待ち行列から削除されるタスク)の ID 番号を返します。待ちタスクの無い場合は TSK_NONE を返します。
- **flgptn**
  flgptn には、現在のイベントフラグビットパターンを返します。

本サービスコールは、タスクコンテキストからは、ref_flg、非タスクコンテキストからは、iref_flg を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
     T_RFLG rflg;
     ER ercd;
        :
     ercd = ref_flg( ID_FLG1, &rflg );
        :
 }
```

《 アセンブリ言語の使用例 》

```
_ refflg:       .blkb   4
    .include mr30.inc
    .GLB        task
task:
        :
       PUSHM   A0,A1
       ref_flg #ID_FLG1,#_refflg
        :
```

5.5 同期･通信機能(データキュー)

表 5.9にデータキュー機能の仕様を示します。

表 5.9 データキュー機能の仕様

項番	項目	内容
1	データキューID	1～255
2	データキュー領域の容量 (データの個数)	0～16383
3	データサイズ	16 ビット
4	データキュー属性	TA_TFIFO : 待ちタスクのキューイングは FIFO 順 TA_TPRI : 待ちタスクのキューイングは優先度順

表 5.10 データキュー機能サービスコール一覧

項番	サービスコール		機能	呼び出し可能なシステム状態					
				T	N	E	D	U	L
1	snd_dtq	**[S]**	データキューへの送信	○		○		○	
2	psnd_dtq	**[S]**	同上(ポーリング)	○		○	○	○	
3	ipsnd_dtq	**[S]**			○	○	○	○	
4	tsnd_dtq	**[S]**	同上(タイムアウト有)	○		○		○	
5	fsnd_dtq	**[S]**	データキューへの強制送信	○		○	○	○	
6	ifsnd_dtq	**[S]**			○	○	○	○	
7	rcv_dtq	**[S]**	データキューからの受信	○		○		○	
8	prcv_dtq	**[S]**	同上(ポーリング)	○		○	○	○	
9	iprcv_dtq				○	○	○	○	
10	trcv_dtq	**[S]**	同上(タイムアウト有)	○		○		○	
11	ref_dtq		データキューの状態参照	○		○	○	○	
12	iref_dtq				○	○	○	○	

【注】

- “[S]”はスタンダードプロファイルのサービスコールです。
- “呼び出し可能なシステム状態”内のそれぞれの記号は、以下の意味です。
  "T"はタスクコンテキストから呼出し可能、"N"は非タスクコンテキストから呼出し可能
  "E"はディスパッチ許可状態から呼出し可能、"D"はディスパッチ禁止状態から呼出し可能
  "U"は CPU ロック解除状態から呼出し可能、"L"は CPU ロック状態から呼出し可能

snd_dtq	**データキューへのデータ送信**
psnd_dtq	**データキューへのデータ送信 (ポーリング)**
ipsnd_dtq	**データキューへのデータ送信 (ポーリング、ハンドラ専用)**
tsnd_dtq	**データキューへのデータ送信 (タイムアウト)**
fsnd_dtq	**データキューへのデータ強制送信**
ifsnd_dtq	**データキューへのデータ強制送信 (ハンドラ専用)**

■ C 言語 API

```
ER ercd = snd_dtq( ID dtqid, VP_INT data );
ER ercd = psnd_dtq( ID dtqid, VP_INT data );
ER ercd = ipsnd_dtq( ID dtqid, VP_INT data );
ER ercd = tsnd_dtq( ID dtqid, VP_INT data, TMO tmout );
ER ercd = fsnd_dtq( ID dtqid, VP_INT data );
ER ercd = ifsnd_dtq( ID dtqid, VP_INT data );
```

● パラメータ

ID	dtqid	対象データキューID 番号
TMO	tmout	タイムアウト値(tsnd_dtq の場合)
VP_INT	data	送信するデータ

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード

■ アセンブリ言語 API

```
.include mr30.inc
snd_dtq   DTQID, DTQDATA
isnd_dtq  DTQID, DTQDATA
psnd_dtq  DTQID, DTQDATA
ipsnd_dtq DTQID, DTQDATA
tsnd_dtq  DTQID, DTQDATA[28]
fsnd_dtq  DTQID, DTQDATA
ifsnd_dtq DTQID, DTQDATA
```

● パラメータ

DTQID	対象データキューID 番号
DTQDATA	送信するデータ

● サービスコール発行後のレジスタ内容

snd_dtq, psnd_dtq, ipsnd_dtq, fsnd_dtq, ifsnd_dtq の場合

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R1	データ
A0	対象データキューID 番号

[28] R2(タイムアウト値上位 16bit)、R0(タイムアウト値下位 16bit)を格納して呼び出す必要があります。

tsnd_dtq の場合

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R1	データ
R2	タイムアウト値(上位 16bit)
A0	対象データキューID 番号

■ エラーコード

E_RLWAI	待ち状態強制解除
E_TMOUT	ポーリング失敗、またはタイムアウト
E_ILUSE	サービスコール不正使用 (dtqcnt が 0 のデータキューに対して fsnd_dtq, ifsnd_dtq を発行)
EV_RST	データキュー領域クリアによって待ち状態が解除された

■ 機能説明

dtqid で示されたデータキューに対して、data で示された 16bit のデータを送信します。対象データキューに受信待ちタスクが存在する場合は、データキューにデータを格納せず、受信待ち行列の先頭タスクにデータを送信し、そのタスクの受信待ち状態を解除します。

一方、既にデータで一杯になったデータキューに対して、snd_dtq,tsnd_dtq を発行した場合、これらのサービスコールを発行したタスクは、実行状態からデータ送信待ち状態に移行し、データキューの空きを待つ送信待ち行列につながれます。その際、dtqid のデータキュー属性が TA_TFIFO の場合は、FIFO 順で待ち行列にタスクをつなぎ、TA_TPRI の場合は、優先度順でタスクをつなぎます。psnd_dtq,ipsnd_dtq の場合は、直ちにリターンし、エラーE_TMOUT を返します。

tsnd_dtq サービスコールの場合は、tmout には、待ち時間を ms 単位で指定します。tmout に指定可能な値は、(0x7FFFFFFF-タイムティック)以内でなければいけません。これより大きな値を指定した場合は、正しく動作しません。tmout に TMO_POL=0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、psnd_dtq と同じ動作をします。また、tmout=TMO_FEVR(-1)にした場合は、永久待ちの指定で、snd_dtq サービスコールと同じ動作をします。

受信待ちタスクがなく、データキュー領域も一杯でない場合、送信したデータはデータキューに格納されます。

snd_dtq,tsnd_dtq サービスコール実行による待ち状態は、以下に示す場合に解除されます。

- **tmout の時間が経過する前に、rcv_dtq,trcv_dtq,prcv_dtq,iprcv_dtq サービスコールが発行され、待ち解除条件が満足された場合**
  この場合、エラーコードは、E_OK を返します。
- **待ち解除条件が満足されないまま、tmout 経過し、最初のタイムティックが発生した場合**
  この場合、エラーコードは、E_TMOUT を返します。
- **他のタスクおよびハンドラから発行した rel_wai、irel_wai サービスコールによって待ち状態が強制解除された場合**
  この場合、エラーコードは、E_RLWAI を返します。
- **他のタスクから発行した vrst_dtq サービスコールによって待ち状態の対象となっているデータキューが削除された場合**
  この場合、エラーコードは、EV_RST を返します。

fsnd_dtq, ifsnd_dtq の場合は、データキューの先頭(最古)のデータを削除し、送信データをデータキュー末尾に格納します。データキュー領域がデータで一杯になっていない場合は、fsnd_dtq, ifsnd_dtq は、snd_dtq と同じ動作を行います。

タスクコンテキストにおいては、snd_dtq, tsnd_dtq, psnd_dtq, fsnd_dtq サービスコール、非タスクコンテキストにおいては、ipsnd_dtq, ifsnd_dtq を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include “kernel_id.h”
 VP_INT data[10];
 void task(void)
 {
                :
     if( snd_dtq( ID_dtq, data[0]) == E_RLWAI ){
       error(“Forced released¥n”);
   }
                :
     if( psnd_dtq( ID_dtq, data[1])== E_TMOUT ){
       error(“Timeout¥n”);
   }
                :
     if( tsnd_dtq( ID_dtq, data[2], 10 ) != E_ TMOUT ){
       error(“Timeout ¥n”);
   }
                :
     if( fsnd_dtq( ID_dtq, data[3]) != E_OK ){
       error(“error¥n”);
   }
                :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
_g_dtq: .WORD   1234H
task:
          :
    PUSHM       R0,R1,R2,A0
    MOV.W       #100,R0
    MOV.W       #0,R2
    tsnd_dtq    #ID_DTQ1,_g_dtq
          :
    PUSHM       R1,A0
    psnd_dtq    #ID_DTQ2,#0FFFFH
          :
    PUSHM       R1,A0
    fsnd_dtq    #ID_DTQ3,#0ABCDH
          :
```

rcv_dtq データキューからのデータ受信
prcv_dtq データキューからのデータ受信 (ポーリング)
iprcv_dtq データキューからのデータ受信 (ポーリング、ハンドラ専用)
trcv_dtq データキューからのデータ受信 (タイムアウト)

■ C 言語 API

```
ER ercd = rcv_dtq( ID dtqid, VP_INT *p_data );
ER ercd = prcv_dtq( ID dtqid, VP_INT *p_data );
ER ercd = iprcv_dtq( ID dtqid, VP_INT *p_data );
ER ercd = trcv_dtq( ID dtqid, VP_INT *p_data, TMO tmout );
```

● パラメータ

ID	dtqid	対象データキューID 番号
TMO	tmout	タイムアウト値(trcv_dtq の場合)
VP_INT	*p_data	受信データを格納する領域先頭へのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード
VP_INT	*p_data	受信データを格納する領域先頭へのポインタ

■ アセンブリ言語 API

```
.include mr30.inc
rcv_dtq   DTQID
prcv_dtq  DTQID
iprcv_dtq DTQID
trcv_dtq  DTQID[29]
```

● パラメータ

DTQID	対象データキューID 番号
TMO	タイムアウト値(trcv_dtq の場合)

● サービスコール発行後のレジスタ内容

rcv_dtq, prcv_dtq, iprcv_dtq の場合

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R1	受信データ
A0	対象データキューID 番号

trcv_dtq の場合

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R1	受信データ
R2	タイムアウト値(上位 16bit)
A0	対象データキューID 番号

[29] R3(タイムアウト値上位 16bit)、R1(タイムアウト値下位 16bit)を格納して呼び出す必要があります。

■ エラーコード

E_RLWAI	待ち状態強制解除
E_TMOUT	ポーリング失敗、またはタイムアウト

■ 機能説明

dtqid で示されたデータキューから、データを受信し、p_data の指す領域に格納します。対象データキューにデータが存在する場合は、その先頭の(最古の)データを受信します。この結果、データキュー領域に空きが発生するため、送信待ち行列につながれているタスクは、その送信待ち状態が解除され、データキュー領域へのデータを送信します。

データキューにデータが存在せず、データ送信待ちタスクが存在する場合（データキュー領域の容量が 0 の場合）、データ送信待ち行列先頭タスクのデータを受信します。この結果、そのデータ送信待ちタスクの待ち状態は解除されます。

一方、データキュー領域にデータが格納されていないデータキューに対して、rcv_dtq,trcv_dtq を発行した場合、これらのサービスコールを発行したタスクは、実行状態からデータ受信待ち状態に移行し、データ受信待ち行列につながれます。このとき、受信待ち行列へは、FIFO 順でつながれます。prcv_dtq,iprcv_dtq の場合は、直ちにリターンし、エラーE_TMOUT を返します。

trcv_dtq サービスコールの場合は、tmout には、待ち時間を ms 単位で指定します。tmout に指定可能な値は、(0x7FFFFFFF-タイムティック)以内でなければいけません。これより大きな値を指定した場合は、正しく動作しません。tmout に TMO_POL=0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、prcv_dtq と同じ動作をします。また、tmout=TMO_FEVR(-1)にした場合は、永久待ちの指定で、rcv_dtq サービスコールと同じ動作をします。

rcv_dtq,trcv_dtq サービスコール実行による待ち状態は、以下に示す場合に解除されます。

- **tmout の時間が経過する前に、snd_dtq,tsnd_dtq,psnd_dtq,ipsnd_dtq サービスコールが発行され、待ち解除条件が満足された場合**
  この場合、エラーコードは、E_OK を返します。
- **待ち解除条件が満足されないまま、tmout 経過し、最初のタイムティックが発生した場合**
  この場合、エラーコードは、E_TMOUT を返します。
- **他のタスクおよびハンドラから発行した rel_wai、irel_wai サービスコールによって待ち状態が強制解除された場合**
  この場合、エラーコードは、E_RLWAI を返します。

タスクコンテキストにおいては、rcv_dtq, trcv_dtq, prcv_dtq サービスコール、非タスクコンテキストにおいては、iprcv_dtq を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include “kernel_id.h”
 void task()
 {
   VP_INT data;
              :
     if( rcv_dtq( ID_dtq, &data ) != E_RLWAI )
         error(“forced wakeup¥n”);
              :
     if( prcv_dtq( ID_dtq, &data ) != E_TMOUT )
         error(“Timeout¥n”);
              :
     if( trcv_dtq( ID_dtq, &data, 10 ) != E_TMOUT )
         error(“Timeout ¥n”);
              :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
         :
    PUSHM       A0,R1,R3
    MOV.W       #0,R1
    MOV.W       #0,R3
    trcv_dtq    #ID_DTQ1
                  :
    PUSHM       A0
    prcv_dtq    #ID_DTQ2
         :
    PUSHM       A0
    rcv_dtq     #ID_DTQ2
         :
```

ref_dtq データキューの状態参照
iref_dtq データキューの状態参照 (ハンドラ専用)

■ C 言語 API

```
ER ercd = ref_dtq( ID dtqid, T_RDTQ *pk_rdtq );
ER ercd = iref_dtq( ID dtqid, T_RDTQ *pk_rdtq );
```

● パラメータ

ID	dtqid	対象データキューID 番号
T_RDTQ	*pk_rdtq	データキュー状態を返すパケットへのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)
T_RDTQ	*pk_rdtq	データキュー状態を返すパケットへのポインタ

pk_rdtq の内容

```
typedef   struct t_rdtq{
    ID      stskid   +0   2    送信待ちタスク ID
    ID      wtskid   +2   2    受信待ちタスク ID
    UINT    sdtqcnt  +4   2    データキューに入っているデータ数
} T_RDTQ;
```

■ アセンブリ言語 API

```
.include mr30.inc
ref_dtq   DTQID, PK_RDTQ
iref_dtq  DTQID, PK_RDTQ
```

● パラメータ

DTQID	対象データキューID 番号
PK_RDTQ	データキュー状態を返すパケットへのポインタ

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象データキューID 番号
A1	データキュー状態を返すパケットへのポインタ

■エラーコード

なし

■ 機能説明

dtqid で示されたデータキューの各種の状態を返します。

- **stskid**
  stskid には送信待ち行列の先頭タスク(次に待ち行列から削除されるタスク)の ID 番号を返します。待ちタスクの無い場合は TSK_NONE を返します。
- **wtskid**
  wtskid には受信待ち行列の先頭タスク(次に待ち行列から削除されるタスク)の ID 番号を返します。待ちタスクの無い場合は TSK_NONE を返します。
- **sdtqcnt**
  sdtqcnt には、データキュー領域に格納されているデータ個数を返します。

本サービスコールは、タスクコンテキストからは、ref_dtq、非タスクコンテキストからは、iref_dtq を使用してください。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
    T_RDTQ rdtq;
    ER ercd;
        :
    ercd = ref_dtq( ID_DTQ1, &rdtq );
        :
}
```

《 アセンブリ言語の使用例 》

```
_ refdtq:       .blkb   6
    .include mr30.inc
    .GLB        task
task:
        :
      PUSHM   A0,A1
      ref_dtq #ID_DTQ1,#_refdtq
        :
```

5.6 同期･通信機能(メールボックス)

表 5.11にメールボックス機能の仕様を示します。

表 5.11 メールボックス機能の仕様

項番	項目	内容	
1	メールボックス ID	1～255	
2	メッセージ優先度	1～255	
3	メールボックス属性	TA_TFIFO :	待ちタスクのキューイングは FIFO
		TA_TPRI :	待ちタスクのキューイングは優先度順
		TA_MFIFO :	メッセージのキューイングは FIFO
		TA_MPRI :	メッセージのキューイングは優先度順

表 5.12 メールボックス機能サービスコール一覧

項番	サービスコール		機能	呼び出し可能なシステム状態					
				T	N	E	D	U	L
1	snd_mbx	**[S]**	メールボックスへの送信	○		○	○	○	
2	isnd_mbx				○	○	○	○	
3	rcv_mbx	**[S]**	メールボックスからの受信	○		○		○	
4	prcv_mbx	**[S]**	同上(ポーリング)	○		○	○	○	
5	iprcv_mbx				○	○	○	○	
6	trcv_mbx	**[S]**	同上(タイムアウト有)	○		○		○	
7	ref_mbx		メールボックスの状態参照	○		○	○	○	
8	iref_mbx				○	○	○	○	

【注】

- “[S]”はスタンダードプロファイルのサービスコールです。
- “呼び出し可能なシステム状態”内のそれぞれの記号は、以下の意味です。
  "T"はタスクコンテキストから呼出し可能、"N"は非タスクコンテキストから呼出し可能
  "E"はディスパッチ許可状態から呼出し可能、"D"はディスパッチ禁止状態から呼出し可能
  "U"は CPU ロック解除状態から呼出し可能、"L"は CPU ロック状態から呼出し可能

snd_mbx メールボックスへのメッセージ送信
isnd_mbx メールボックスへのメッセージ送信 (ハンドラ専用)

■ C 言語 API

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

● パラメータ

ID	mbxid	対象メールボックス ID 番号
T_MSG	*pk_msg	送信するメッセージ

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語 API

```
.include mr30.inc
snd_mbx   MBXID, PK_MBX
isnd_mbx  MBXID, PK_MBX
```

● パラメータ

MBXID	対象メールボックス ID 番号
PK_MBX	送信するメッセージ (アドレス)

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象メールボックス ID 番号
A1	送信するメッセージ (アドレス)

■ メッセージパケットの構造

<<メールボックスのメッセージヘッダ>>

```
typedef   struct t_msg{
    VP      msghead   +0    2     カーネル管理領域
} T_MSG;
```

<<メールボックスの優先度付きメッセージヘッダ>>

```
typedef   struct t_msg{
    T_MSG   msgque    +0    2     メッセージヘッダ
    PRI     msgpri    +2    2     メッセージ優先度
} T_MSG_PRI;
```

■ エラーコード

なし

■ 機能説明

mbxid で示されたメールボックスに pk_msg で示されたメッセージを送信します。T_MSG*は、near ポインタとして扱います。 すでに対象メールボックスにメッセージの受信を待つタスクが存在していれば、待ち行列先頭のタスクに送信したメッセージが渡され、そのタスクの待ち状態が解除されます。

TA_MFIFO 属性のメールボックスにメッセージを送る場合は、以下の例に示すように先頭に T_MSG 構造体を付加した形式で、メッセージを作成してください。

TA_MPRI 属性のメールボックスにメッセージを送る場合は、以下の例に示すように先頭に T_MSG_PRI 構造体を付加した形式で、メッセージを作成してください。

TA_MFIFO, TA_MPRI いずれの属性の場合もメッセージは RAM 領域に作成してください。

T_MSG の領域はカーネルが使用するため、送信後は書き換えてはなりません。メッセージ送信後、メッセージが受信される前にこの領域を書き換えた場合の動作は保証されません。

タスクコンテキストにおいては、snd_mbx サービスコール、非タスクコンテキストにおいては、isnd_mbx を使用してください。

<<メッセージの形式例>>

```
typedef struct user_msg{
          T_MSG  t_msg;                /* T_MSG 構造体 */
          B      data[16];             /* ユーザメッセージデータ */
} USER_MSG;
```

<<優先度付きメッセージの形式例>>

```
typedef struct user_msg{
          T_MSG_PRI     t_msg;         /* T_MSG_PRI 構造体 */
          B             data[16];      /* ユーザーメッセージデータ */
} USER_MSG;
```

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
typedef struct pri_message
{
       T_MSG_PRI      msgheader;
       char   body[12];
} PRI_MSG;

void task(void)
{
   PRI_MSG *   msg;
               :
   msg->msgheader.msgpri = 5;
   snd_mbx( ID_msg,(T_MSG *)&msg);
       :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
_g_userMsg:     .blkb   2       ; Header
                .blkb   12      ; Body
task:
          :
    PUSHM       A0,A1
    snd_mbx     #ID_MBX1,#_g_userMsg            :
          :
```

rcv_mbx メールボックスからのメッセージ受信
prcv_mbx メールボックスからのメッセージ受信 (ポーリング)
iprcv_mbx メールボックスからのメッセージ受信 (ポーリング、ハンドラ専用)
trcv_mbx メールボックスからのメッセージ受信 (タイムアウト)

■ C 言語 API

```
ER ercd = rcv_mbx( ID mbxid, T_MSG **ppk_msg );
ER ercd = prcv_mbx( ID mbxid, T_MSG **ppk_msg );
ER ercd = iprcv_mbx( ID mbxid, T_MSG **ppk_msg );
ER ercd = trcv_mbx( ID mbxid, T_MSG **ppk_msg, TMO tmout );
```

● パラメータ

ID	mbxid	対象メールボックス ID 番号
TMO	tmout	タイムアウト値(trcv_mbx の場合)
T_MSG	**ppk_msg	受信メッセージを格納する領域先頭へのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード
T_MSG	**ppk_msg	受信メッセージを格納する領域先頭へのポインタ

■ アセンブリ言語 API

```
.include mr30.inc
rcv_mbx   MBXID
prcv_mbx  MBXID
iprcv_mbx MBXID
trcv_mbx  MBXID[30]
```

● パラメータ

MBXID	対象メールボックス ID 番号

● サービスコール発行後のレジスタ内容

rcv_mbx, prcv_mbx, iprcv_mbx の場合

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R1	受信したメッセージ
A0	対象メールボックス ID 番号

[30] R3(タイムアウト値上位 16bit)、R1(タイムアウト値下位 16bit)を格納して呼び出す必要があります。

trcv_mbx の場合

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R1	受信したメッセージ
R3	タイムアウト値(上位 16bit)
A0	対象メールボックス ID 番号

■ エラーコード

E_RLWAI	待ち状態強制解除
E_TMOUT	ポーリング失敗、またはタイムアウト

■ 機能説明

mbxid で示されたメールボックスから、メッセージを受信し、ppk_msg の指す領域に受信したメッセージの先頭アドレスを格納します。T_MSG**は、near ポインタとして扱います。 対象メールボックスにデータが存在する場合は、その先頭のデータを受信します。

一方、メールボックスにメッセージがないメールボックスに対して、rcv_mbx, trcv_mbx を発行した場合、これらのサービスコールを発行したタスクは、実行状態からメッセージ受信待ち状態に移行し、メッセージ受信待ち行列につながれます。その際、mbxid のメールボックス属性が TA_TFIFO の場合は、FIFO 順で待ち行列にタスクをつなぎ、TA_TPRI の場合は、優先度順でタスクをつなぎます。prcv_mbx, iprcv_mbx の場合は、直ちにﾘﾀｰﾝし、エラーE_TMOUT を返します。

trcv_mbx サービスコールの場合は、tmout には、待ち時間を ms 単位で指定します。tmout に指定可能な値は、(0x7FFFFFFF-タイムティック)以内でなければいけません。これより大きな値を指定した場合は、正しく動作しません。tmout に TMO_POL=0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、prcv_mbx と同じ動作をします。また、tmout=TMO_FEVR(-1)にした場合は、永久待ちの指定で、rcv_mbx サービスコールと同じ動作をします。

rcv_mbx,trcv_mbx サービスコール実行による待ち状態は、以下に示す場合に解除されます。

- **tmout の時間が経過する前に、rcv_mbx,trcv_mbx,prcv_mbx,iprcv_mbx サービスコールが発行され、待ち解除条件が満足された場合**
  この場合、エラーコードは、E_OK を返します。
- **待ち解除条件が満足されないまま、tmout 経過し、最初のタイムティックが発生した場合**
  この場合、エラーコードは、E_TMOUT を返します。
- **他のタスクおよびハンドラから発行した rel_wai、irel_wai サービスコールによって待ち状態が強制解除された場合**
  この場合、エラーコードは、E_RLWAI を返します。

タスクコンテキストにおいては、rcv_mbx, trcv_mbx, prcv_mbx サービスコール、非タスクコンテキストにおいては、iprcv_mbx を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"

 typedef struct fifo_message
 {
       T_MSG   head;
       char    body[12];
 } FIFO_MSG;
 void task()
 {
   FIFO_MSG *msg;
              :
     if( rcv_mbx((T_MSG **)&msg, ID_mbx) == E_RLWAI )
         error("forced wakeup¥n");
              :
              :
     if( prcv_mbx((T_MSG **)&msg, ID_mbx) != E_TMOUT )
         error("Timeout¥n");
              :
              :
     if( trcv_mbx((T_MSG **)&msg, ID_mbx,10) != E_TMOUT )
         error("Timeout¥n");
              :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
         :
    PUSHM       R3,A0
    MOV.W       #100,R1
    MOV.W       #0,R3
    trcv_mbx    #ID_MBX1
         :
    PUSHM       R3,A0
    rcv_mbx     #ID_MBX1
         :
    PUSHM       R3,A0
    prcv_mbx    #ID_MBX1
         :
```

ref_mbx メールボックスの状態参照
iref_mbx メールボックスの状態参照 (ハンドラ専用)

■ C 言語 API

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

● パラメータ

ID	mbxid	対象メールボックス ID 番号
T_RMBX	*pk_rmbx	メールボックス状態を返すパケットへのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)
T_RMBX	*pk_rmbx	メールボックス状態を返すパケットへのポインタ

pk_rmbx の内容

```
typedef   struct t_rmbx{
    ID      wtskid   +0   2    受信待ちタスク ID
    T_MSG   *pk_msg  +2   2    次に受信されるメッセージパケット
} T_RMBX;
```

■ アセンブリ言語 API

```
.include mr30.inc
ref_mbx   MBXID, PK_RMBX
iref_mbx  MBXID, PK_RMBX
```

● パラメータ

MBXID	対象メールボックス ID 番号
PK_RMBX	メールボックス状態を返すパケットへのポインタ

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象メールボックス ID 番号
A1	メールボックス状態を返すパケットへのポインタ

■エラーコード

なし

■ 機能説明

mbxid で示されたメールボックスの各種の状態を返します。

- **wtskid**
  wtskid には受信待ち行列の先頭タスク(次に待ち行列から削除されるタスク)の ID 番号を返します。待ちタスクの無い場合は TSK_NONE を返します。
- ***pk_msg**
  次に受信されるメッセージの先頭アドレスを返します。次に受信されるメッセージがない場合は、NULL を返します。

本サービスコールは、タスクコンテキストからは、ref_mbx、非タスクコンテキストからは、iref_mbx を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
      T_RMBX rmbx;
      ER ercd;
           :
      ercd = ref_mbx( ID_MBX1, &rmbx );
           :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
_ refmbx:       .blkb   4
task:
           :
        PUSHM   A0,A1
        ref_mbx #ID_MBX1,#_refmbx
           :
```

5.7 メモリプール管理機能(固定長メモリプール)

表 5.13に固定長メモリプール機能の仕様を示します。メモリプール領域は、コンフィギュレーション時に、メモリプール毎にセクション名を指定することが出来ます。

表 5.13 固定長メモリプール機能の仕様

項番	項目	内容
1	固定長メモリプール ID	1～255
2	固定長メモリプール個数	1～65535
3	固定長メモリプールサイズ	2～65535
4	サポート属性	TA_TFIFO： 待ちタスクのキューイングは FIFO TA_TPRI： 待ちタスクのキューイングは優先度順
5	メモリプール領域の指定	メモリプール領域をセクション指定可能

表 5.14 固定長メモリプールサービスコール一覧

項番	サービスコール		機能	呼び出し可能なシステム状態					
				T	N	E	D	U	L
1	get_mpf	**[S]**	固定長メモリブロックの獲得	○		○		○	
2	pget_mpf	**[S]**	同上(ポーリング)	○		○	○	○	
3	ipget_mpf				○	○	○	○	
4	tget_mpf	**[S]**	同上(タイムアウト有)	○		○		○	
5	rel_mpf	**[S]**	固定長メモリブロックの返却	○		○	○	○	
6	irel_mpf				○	○	○	○	
7	ref_mpf		固定長メモリプールの状態参照	○		○	○	○	
8	iref_mpf				○	○	○	○	

【注】

- “[S]”はスタンダードプロファイルのサービスコールです。
- “呼び出し可能なシステム状態”内のそれぞれの記号は、以下の意味です。
  "T"はタスクコンテキストから呼出し可能、"N"は非タスクコンテキストから呼出し可能
  "E"はディスパッチ許可状態から呼出し可能、"D"はディスパッチ禁止状態から呼出し可能
  "U"は CPU ロック解除状態から呼出し可能、"L"は CPU ロック状態から呼出し可能

get_mpf	**固定長メモリブロックの獲得**
pget_mpf	**固定長メモリブロックの獲得 (ポーリング)**
ipget_mpf	**固定長メモリブロックの獲得 (ポーリング、ハンドラ専用)**
tget_mpf	**固定長メモリブロックの獲得 (タイムアウト)**

■ C 言語 API

```
ER ercd = get_mpf( ID mpfid, VP *p_blk );
ER ercd = pget_mpf( ID mpfid, VP *p_blk );
ER ercd = ipget_mpf( ID mpfid, VP *p_blk );
ER ercd = tget_mpf( ID mpfid, VP *p_blk,TMO tmout );
```

● パラメータ

ID	mpfid	対象固定長メモリプール ID 番号
VP	*p_blk	獲得したメモリブロック先頭アドレスへのポインタ
TMO	tmout	タイムアウト値(tget_mpf の場合)

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード
VP	*p_blk	獲得したメモリブロック先頭アドレスへのポインタ

■ アセンブリ言語 API

```
.include mr30.inc
get_mpf   MPFID
pget_mpf  MPFID
ipget_mpf MPFID
tget_mpf  MPFID[31]
```

● パラメータ

MPFID	対象固定長メモリプール ID 番号

● サービスコール発行後のレジスタ内容

get_mpf, pget_mpf, ipget_mpf の場合

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R1	獲得したブロックの先頭アドレス
A0	対象固定長メモリプール ID 番号

tget_mpf の場合

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R1	獲得したブロックの先頭アドレス
R3	タイムアウト値(下位 16bit)
A0	対象固定長メモリプール ID 番号

[31] R3(タイムアウト値上位 16bit)、R1(タイムアウト値下位 16bit)を格納して呼び出す必要があります。

■ エラーコード

E_RLWAI	待ち状態強制解除
E_TMOUT	ポーリング失敗、またはタイムアウト
EV_RST	メモリプール領域クリアによって待ち状態が解除された

■ 機能説明

mpfid で示される固定長メモリプールからメモリブロックを獲得し、獲得したメモリブロックの先頭アドレスを変数 p_blk に格納します。獲得したメモリブロックの内容は、不定です。

指定した固定長メモリプールにメモリブロックがない場合は、tget_mpf, get_mpf サービスコール使用時には、本サービスコールを発行したタスクは、メモリブロック待ち状態に移行し、メモリブロック待ち行列につながれます。その際、mpfid の固定長メモリプール属性が TA_TFIFO の場合は、FIFO 順で待ち行列にタスクをつなぎ、TA_TPRI の場合は、優先度順でタスクをつなぎます。pget_mpf, ipget_mpf サービスコール使用時は、直ちにリターンし、エラーE_TMOUT を返します。

tget_mpf サービスコールの場合は、tmout には、待ち時間を ms 単位で指定します。tmout に指定可能な値は、〔0x7FFFFFFF-タイムティック〕以内でなければいけません。これより大きな値を指定した場合は、正しく動作しません。tmout に TMO_POL=0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、pget_mpf と同じ動作をします。また、tmout=TMO_FEVR(-1)にした場合は、永久待ちの指定で、get_mpf サービスコールと同じ動作をします。

get_mpf, tget_mpf サービスコール実行による待ち状態は、以下に示す場合に解除されます。

- **tmout の時間が経過する前に、rel_mpf,irel_mpf サービスコールが発行され、待ち解除条件が満足された場合**
  この場合、エラーコードは、E_OK を返します。
- **待ち解除条件が満足されないまま、tmout 経過し、最初のタイムティックが発生した場合**
  この場合、エラーコードは、E_TMOUT を返します。
- **他のタスクおよびハンドラから発行した rel_wai、irel_wai サービスコールによって待ち状態が強制解除された場合**
  この場合、エラーコードは、E_RLWAI を返します。
- **他のタスクから発行した vrst_mpf サービスコールによって待ち状態の対象となっているメモリプールが削除された場合**
  この場合、エラーコードは、EV_RST を返します。

本サービスコールによって獲得されたメモリブロックの値は、初期化しないため不定となります。

本サービスコールは、タスクコンテキストからは、get_mpf, pget_mpf, tget_mpf、非タスクコンテキストからは、ipget_mpf を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include “kernel_id.h”
 VP      p_blk;
 void task()
 {
   if( get_mpf(ID_mpf ,&p_blk) != E_OK ){
         error(“Not enough memory¥n”);
   }
               :
   if( pget_mpf(ID_mpf ,&p_blk) != E_OK ){
         error(“Not enough memory¥n”);
   }
               :
   if( tget_mpf(ID_mpf ,&p_blk, 10) != E_OK ){
         error(“Not enough memory¥n”);
   }

 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       A0
    get_mpf     #ID_MPF1
          :
    PUSHM       A0
    pget_mpf    #ID_MPF1
          :
    PUSHM       A0
    MOV.W       R1,#200
    MOV.W       R3,#0
    tget_mpf    #ID_MPF1
          :
```

rel_mpf 固定長メモリプールブロックの解放
irel_mpf 固定長メモリプールブロックの解放 (ハンドラ専用)

■ C 言語 API

```
ER ercd = rel_mpf( ID mpfid, VP blk );
ER ercd = irel_mpf( ID mpfid, VP blk);
```

● パラメータ

ID	mpfid	対象固定長メモリプール ID 番号
VP	blk	返却するメモリブロックの先頭アドレス

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語 API

```
.include mr30.inc
rel_mpf   MPFID, BLK
irel_mpf  MPFID, BLK
```

● パラメータ

MPFID	対象固定長メモリプール ID 番号
BLK	返却するメモリブロックの先頭アドレス

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
R1	返却するメモリブロックの先頭アドレス
A0	対象固定長メモリプール ID 番号

■ エラーコード

なし

■ 機能説明

blk に示される先頭アドレスをもつメモリブロックを解放します。返却するメモリブロックの先頭アドレスは必ず、get_mpf, tget_mpf, pget_mpf, ipget_mpf で獲得した先頭アドレスを指定してください。

また、対象メモリプールの待ち行列にタスクがつながれている場合は、待ち行列の先頭タスクを待ち行列からはずし、レディキューにつなぎかえこのタスクにメモリブロックを割り当てます。この時のタスクの状態は、メモリブロック待ち状態から実行(RUNNING)状態あるいは実行可能(READY)状態へ移行します。本サービスコールは、blk の内容をチェックしません。従って、blk に正しいアドレスが格納されていない場合は正しく動作しません。

タスクコンテキストにおいては、rel_mpf サービスコール、非タスクコンテキストにおいては、irel_mpf を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include “kernel_id.h”
 void task()
 {
     VP   p_blf;
     if( get_mpf(ID_mpf1,&p_blf) != E_OK )
         error(“Not enough memory ¥n”);
               :
     rel_mpf(ID_mpf1,p_blf);
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
_g_blk: .blkb 2
task:
          :
    PUSHM       A0
    get_mpf     #ID_MPF1
          :
    MOV.W       R1,_g_blk
    PUSHM       A0
    rel_mpf     #ID_MPF1,_g_blk
          :
```

ref_mpf 固定長メモリプールの状態参照
iref_mpf 固定長メモリプールの状態参照 (ハンドラ専用)

■ C 言語 API

```
ER ercd = ref_mpf( ID mpfid, T_RMPF *pk_rmpf );
ER ercd = iref_mpf( ID mpfid, T_RMPF *pk_rmpf );
```

● パラメータ

ID	mpfid	対象固定長メモリプール ID 番号
T_RMPF	*pk_rmpf	固定長メモリプール状態を返すパケットへのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)
T_RMPF	*pk_rmpf	固定長メモリプール状態を返すパケットへのポインタ

pk_rmpf の内容

```
typedef   struct t_rmpf{
    ID      wtskid   +0   2    メモリブロック獲得待ちタスク ID
    UINT    fblkcnt  +2   2    空きメモリブロック数(個数)
} T_RMPF;
```

■ アセンブリ言語 API

```
.include mr30.inc
ref_mpf   MPFID, PK_RMPF
iref_mpf  MPFID, PK_RMPF
```

● パラメータ

MPFID	対象固定長メモリプール ID 番号
PK_RMPF	固定長メモリプール状態を返すパケットへのポインタ

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象固定長メモリプール ID 番号
A1	固定長メモリプール状態を返すパケットへのポインタ

■エラーコード

なし

■ 機能説明

mpfid で示されたメッセージバッファの各種の状態を返します。

- **wtskid**
  wtskid にはメモリブロック獲得待ち行列の先頭タスク(最も早く待ちに入ったタスク)の ID 番号を返します。待ちタスクの無い場合は TSK_NONE を返します。
- **fblkcnt**
  指定したメモリプールの空きブロック数を返します。

本サービスコールは、タスクコンテキストからは、rel_mpf、非タスクコンテキストからは、irel_mpf を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
     T_RMPF rmpf;
     ER ercd;
         :
     ercd = ref_mpf( ID_MPF1, &rmpf );
         :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
_ refmpf:       .blkb   4
task:
         :
        PUSHM   A0,A1
        ref_mpf #ID_MPF1,#_refmpf
         :
```

5.8 メモリプール管理機能(可変長メモリプール)

表 5.15に可変長メモリプール機能の仕様を示します。
メモリプール領域は、コンフィギュレーション時に、メモリプール毎にセクション名を指定することが出来ます。

表 5.15 可変長メモリプール機能の仕様

項番	項目	内容
1	可変長メモリプール ID	1～255
2	可変長メモリプールサイズ	16-65535
3	確保するメモリブロックの最大値	1-65520
4	サポート属性	メモリ不足時のタスク待ちの API は未サポート
5	メモリ領域の指定	可変長メモリプール領域をセクション指定可能

表 5.16 可変長メモリプールサービスコール一覧

項番	サービスコール		機能	呼び出し可能なシステム状態					
				T	N	E	D	U	L
1	pget_mpl		可変長メモリブロックの獲得(ポーリング)	○		○	○	○	
2	rel_mpl		可変長メモリブロックの返却	○		○	○	○	
3	ref_mpl		可変長メモリプールの状態参照	○		○	○	○	
4	iref_mpl				○	○	○	○	

【注】

- “[S]”はスタンダードプロファイルのサービスコールです。
- “呼び出し可能なシステム状態”内のそれぞれの記号は、以下の意味です。
  "T"はタスクコンテキストから呼出し可能、"N"は非タスクコンテキストから呼出し可能
  "E"はディスパッチ許可状態から呼出し可能、"D"はディスパッチ禁止状態から呼出し可能
  "U"は CPU ロック解除状態から呼出し可能、"L"は CPU ロック状態から呼出し可能

pget_mpl 可変長メモリブロックの獲得

■ C言語API

```
ER ercd = pget_mpl( ID mplid, UINT blksz, VP *p_blk );
```

● パラメータ

ID	mplid	対象可変長メモリプール ID 番号
UINT	blksz	獲得するメモリのサイズ(バイト数)
VP	*p_blk	獲得したメモリの先頭アドレスへのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード
VP	*p_blk	獲得したメモリブロック先頭アドレスへのポインタ

■ アセンブリ言語API

```
.include mr30.inc
pget_mpl  MPLID, BLKSZ
```

● パラメータ

MPLID	対象可変長メモリプール ID 番号
BLKSZ	獲得するメモリのサイズ(バイト数)

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R1	獲得したメモリの先頭アドレス
A0	対象可変長メモリプール ID 番号

■ エラーコード

E_TMOUT	メモリブロックなし

■ 機能説明

mplid で示される可変長メモリプールからメモリブロックを獲得し、獲得したメモリブロックの先頭アドレスを変数 p_blk に格納します。獲得したメモリブロックの内容は、不定です。

指定した可変長メモリプールにメモリブロックがない場合は、直ちにリターンし、エラーE_TMOUT を返します。本サービスコールによって獲得されたメモリブロックの値は、初期化しないため不定となります。

本サービスコールは、タスクコンテキストにおいてのみ使用可能です。非タスクコンテキストにおいて使用した場合は正常に動作しません。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include “kernel_id.h”
 VP      p_blk;
 void task()
 {
   if( pget_mpl(ID_mpl , 200, &p_blk) != E_OK ){
         error(“Not enough memory¥n”);
   }
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
         :
    PUSHM       A0
    pget_mpl    #ID_MPL1,#200
         :
```

rel_mpl 可変長メモリプールブロックの解放

■ C 言語 API

ER ercd = rel_mpl(ID mplid, VP blk);

● パラメータ

ID	mplid	対象可変長メモリプール ID 番号
VP	blk	返却するメモリブロックの先頭アドレス

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語 API

.include mr30.inc
rel_mpl MPLID,BLK

● パラメータ

MPLID	対象可変長メモリプール ID 番号
BLK	返却するメモリブロックの先頭アドレス

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
R1	返却するメモリブロックの先頭アドレス
A0	対象可変長メモリプール ID 番号

■ エラーコード

なし

■ 機能説明

blk に示される先頭アドレスをもつメモリブロックを解放します。返却するメモリブロックの先頭アドレスは必ず、pget_mpl で獲得した先頭アドレスを指定してください。
本サービスコールは、blk の内容をチェックしません。従って、blk に正しいアドレスが格納されていない場合は正しく動作しません。
本サービスコールは、タスクコンテキストにおいてのみ使用可能です。非タスクコンテキストにおいて使用した場合は正常に動作しません。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include “kernel_id.h”
 void task()
 {
     VP   p_blk;
     if( get_mpl(ID_mpl1, 200, &p_blk) != E_OK )
         error(“Not enough memory ¥n”);
             :
     rel_mpl(ID_mp1,p_blk);
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
_g_blk: .blkb 4
task:
          :
    PUSHM       A0
    get_mpl     #ID_MPL1,#200
          :
    MOV.L       R3R1,_g_blk
    PUSHM       A0
    rel_mpf     #ID_MPL1,_g_blk
          :
```

ref_mpl 可変長メモリプールの状態参照
iref_mpl 可変長メモリプールの状態参照(ハンドラ専用)

■ C言語API

```
ER ercd = ref_mpl( ID mplid, T_RMPL *pk_rmpl );
ER ercd = iref_mpl( ID mplid, T_RMPL *pk_rmpl );
```

● パラメータ

ID	mplid	対象可変長メモリプール ID 番号
T_RMPL	*pk_rmpl	可変長メモリプール状態を返すパケットへのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)
T_RMPL	*pk_rmpl	可変長メモリプール状態を返すパケットへのポインタ

pk_rmpl の内容

```
typedef   struct t_rmpl{
    ID      wtskid   +0   2    メモリ獲得待ちタスク ID (未使用)
    SIZE    fmplsz   +2   2    空きメモリサイズ(バイト数)
    UINT    fblksz   +4   2    すぐに獲得可能なメモリの最大サイズ(バイト数)
} T_RMPL;
```

■ アセンブリ言語API

```
.include mr30.inc
ref_mpl   MPLID, PK_RMPL
iref_mpl  MPLID, PK_RMPL
```

● パラメータ

MPLID	対象可変長メモリプール ID 番号
PK_RMPL	可変長メモリプール状態を返すパケットへのポインタ

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象可変長メモリプール ID 番号
A1	可変長メモリプール状態を返すパケットへのポインタ

■エラーコード

なし

■ 機能説明

mplid で示されたﾒｯｾｰｼﾞﾊﾞｯﾌｧの各種の状態を返します。

- **wtskid**
  未使用。
- **fmplsz**
  空きﾒﾓﾘｻｲｽﾞを返します。
- **fblksz**
  すぐに獲得できるﾒﾓﾘの最大ｻｲｽﾞを返します。

本サービスコールは、ﾀｽｸｺﾝﾃｷｽﾄからは、ref_mpl、非ﾀｽｸｺﾝﾃｷｽﾄからは、iref_mpl を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
      T_RMPL rmpl;
      ER ercd;
           :
      ercd = ref_mpl( ID_MPL1, &rmpl );
           :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
_ refmpl:       .blkb   6
task:
           :
        PUSHM   A0,A1
        ref_mpl #ID_MPL1,_refmpl
           :
```

5.9 時間管理機能(システム時刻管理)

表 5.17にシステム時刻管理の仕様を示します。

表 5.17 システム時刻管理の仕様

項番	項目	内容
1	システム時刻値	符号なし 48 ビット
2	システム時刻の単位	1[ms]
3	システム時刻の更新周期	ユーザ指定のタイムティック更新時間[ms]
4	システム時刻初期値(初期起動時)	000000000000H

表 5.18 時間管理機能(システム時刻管理)サービスコール一覧

項番	サービスコール		機能	呼び出し可能なシステム状態					
				T	N	E	D	U	L
1	get_tim	**[S]**	システム時刻の参照	○		○	○	○	
2	iget_tim				○	○	○	○	
3	set_tim	**[S]**	システム時刻の設定	○		○	○	○	
4	iset_tim				○	○	○	○	
5	isig_tim	**[S]**	システム時刻の供給		○	○	○	○	

【注】

- “[S]”はスタンダードプロファイルのサービスコールです。
- “呼び出し可能なシステム状態”内のそれぞれの記号は、以下の意味です。
  "T"はタスクコンテキストから呼出し可能、"N"は非タスクコンテキストから呼出し可能
  "E"はディスパッチ許可状態から呼出し可能、"D"はディスパッチ禁止状態から呼出し可能
  "U"は CPU ロック解除状態から呼出し可能、"L"は CPU ロック状態から呼出し可能

set_tim システム時刻の設定
iset_tim システム時刻の設定 (ハンドラ専用)

■ C 言語 API

```
ER ercd = set_tim( SYSTIM *p_systim );
ER ercd = iset_tim( SYSTIM *p_systim );
```

● パラメータ

SYSTIM	*p_systim	設定するシステム時刻を示すパケットへのポインタ

p_systim の内容

```
typedef   struct t_systim {
    UH      utime     0     2     上位 16bit
    UW      ltime     +4    4     下位 32bit
} SYSTIM;
```

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語 API

```
.include mr30.inc
set_tim   PK_TIM
iset_tim  PK_TIM
```

● パラメータ

PK_TIM	設定するシステム時刻を示すパケットへのポインタ

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	設定するシステム時刻を示すパケットへのポインタ

■ エラーコード

なし

■ 機能説明

システム時刻の現在値を p_systim で示されるパケットの値に更新します。パケットに指定する時刻の単位は、タイムティック数ではなく”ms”となります。

パケットに指定可能な値は、0x7FFF:FFFFFFFF 以内でなければいけません。これより大きな値を指定した場合は、正しく動作しません。

本サービスコールは、タスクコンテキストからは、set_tim、非タスクコンテキストからは、iset_tim を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
   SYSTIME time;          /* 時刻データ格納変数 */
   time.utime = 0;        /* 上位時刻データの設定 */
   time.ltime = 0;        /* 下位時刻データの設定 */
   set_tim( &time );      /* システム時刻の変更 */
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
_g_systim:
        .WORD   1111H
        .LWORD  22223333H
task:
          :
        PUSHM   A0
        set_tim #_g_systim
          :
```

get_tim システム時刻の参照
iget_tim システム時刻の参照 (ハンドラ専用)

■ C 言語 API

```
ER ercd = get_tim( SYSTIM *p_systim );
ER ercd = iget_tim( SYSTIM *p_systim );
```

● パラメータ

SYSTIM	*p_systim	現在のシステム時刻を返すパケットへのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)
SYSTIM	*p_systim	現在のシステム時刻を返すパケットへのポインタ

p_systim の内容

```
typedef   struct t_systim {
    UH       utime     0     2     上位 16bit
    UW       ltime     +4    4     下位 32bit
} SYSTIM;
```

■ アセンブリ言語 API

```
.include mr30.inc
get_tim      PK_TIM
iget_tim     PK_TIM
```

● パラメータ

PK_TIM	現在のシステム時刻を返すパケットへのポインタ

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	現在のシステム時刻を返すパケットへのポインタ

■ エラーコード

なし

■ 機能説明

システム時刻の現在値を p_systim に格納します。

本サービスコールは、タスクコンテキストからは、get_tim、非タスクコンテキストからは、iget_tim を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
   SYSTIME time;          /* 時刻データ格納変数 */
   get_tim( &time );      /* システム時刻の変更 */
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
_g_systim:      .blkb   6
task:
          :
        PUSHM   A0
        get_tim #_g_systim
          :
```

isig_tim タイムティックの供給

■ 機能説明

システム時刻を更新します。

コンフィギュレーションファイルにてシステムクロックを定義すると、tic_nume(ms)の間隔で isig_tim が自動的に起動されるようになっています。本機能は、サービスコールとして実装されていませんのでアプリケーションから呼び出すことは出来ません。

タイムティック供給時には、カーネルは、以下の処理を行います。

(1) システム時刻の更新
(2) アラームハンドラの起動
(3) 周期ハンドラの起動
(4) tslp_tsk などタイムアウト付きサービスコールで待ち状態になっているタスクのタイムアウト処理

5.10時間管理機能(周期ハンドラ)

表 5.19に周期ハンドラの仕様を示します。項番4周期ハンドラ属性の記述言語は、GUIコンフィギュレータでの指定内容です。コンフィギュレーションファイルには出力されず、カーネルも関知しません。

表 5.19 時間管理機能(周期ハンドラ)の仕様

項番	項目	内容
1	周期ハンドラ ID	1～255
2	起動周期	0～0x7FFFFFFF-タイムティック[ms]
3	起動位相	0～0x7FFFFFFF-タイムティック[ms]
4	拡張情報	16bit
4	周期ハンドラ属性	TA_HLNG : 高級言語記述 TA_ASM : アセンブリ言語記述 TA_STA ： 周期ハンドラの動作開始 TA_PHS ： 起動位相の保存

表 5.20 時間管理機能(周期ハンドラ)サービスコール一覧

項番	サービスコール		機能	呼び出し可能なシステム状態					
				T	N	E	D	U	L
1	sta_cyc	**[S]**	周期ハンドラの動作開始	○		○	○	○	
2	ista_cyc				○	○	○	○	
3	stp_cyc	**[S]**	周期ハンドラの動作停止	○		○	○	○	
4	istp_cyc				○	○	○	○	
5	ref_cyc		周期ハンドラの状態参照	○		○	○	○	
6	iref_cyc				○	○	○	○	

【注】

- “[S]”はスタンダードプロファイルのサービスコールです。
- “呼び出し可能なシステム状態”内のそれぞれの記号は、以下の意味です。
  "T"はタスクコンテキストから呼出し可能、"N"は非タスクコンテキストから呼出し可能
  "E"はディスパッチ許可状態から呼出し可能、"D"はディスパッチ禁止状態から呼出し可能
  "U"は CPU ロック解除状態から呼出し可能、"L"は CPU ロック状態から呼出し可能

sta_cyc 周期ハンドラの動作開始
ista_cyc 周期ハンドラの動作開始 (ハンドラ専用)

■ C 言語 API

```
ER ercd = sta_cyc( ID cycid );
ER ercd = ista_cyc( ID cycid );
```

● パラメータ

ID	cycid	対象周期ハンドラ ID 番号

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語 API

```
.include mr30.inc
sta_cyc   CYCNO
ista_cyc  CYCNO
```

● パラメータ

CYCNO	対象周期ハンドラ ID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象周期ハンドラ ID 番号

■ エラーコード

なし

■ 機能説明

cycid で示された周期ハンドラを動作している状態に移行させます。周期ハンドラ属性に TA_PHS が指定されていない場合は、このサービスコールを呼び出した時刻を基準として、その時刻から起動周期が経過する毎に周期ハンドラが起動されます。

TA_PHS が指定されておらず、既に動作状態の周期ハンドラに対して本サービスコールを発行した場合、周期ハンドラが次に起動する時刻を再設定します。

TA_PHS が指定されており、既に動作状態の周期ハンドラに対して本サービスコールを発行した場合、本サービスコールは起動時刻の再設定はしません。

本サービスコールは、タスクコンテキストからは、sta_cyc、非タスクコンテキストからは、ista_cyc を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
       :
   sta_cyc ( ID_cyc1 );
       :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
        PUSHM   A0
        sta_cyc #ID_CYC1
          :
```

stp_cyc 周期ハンドラの動作停止
istp_cyc 周期ハンドラの動作停止 (ハンドラ専用)

■ C 言語 API

```
ER ercd = stp_cyc( ID cycid );
ER ercd = istp_cyc( ID cycid );
```

● パラメータ

ID	cycid	対象周期ハンドラ ID 番号

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語 API

```
.include mr30.inc
stp_cyc   CYCNO
istp_cyc  CYCNO
```

● パラメータ

CYCNO	対象周期ハンドラ ID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象周期ハンドラ ID 番号

■ エラーコード

なし

■ 機能説明

cycid で示された周期ハンドラを動作していない状態に移行させます。

本サービスコールは、タスクコンテキストからは、stp_cyc、非タスクコンテキストからは、istp_cyc を使用してください。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
      :
   stp_cyc ( ID_cyc1 );
      :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
        :
        PUSHM   A0
        stp_cyc #ID_CYC1
          :
```

ref_cyc 周期ハンドラの状態参照
iref_cyc 周期ハンドラの状態参照 (ハンドラ専用)

■ C 言語 API

```
ER ercd = ref_cyc( ID cycid, T_RCYC *pk_rcyc );
ER ercd = iref_cyc( ID cycid, T_RCYC *pk_rcyc );
```

● パラメータ

ID	cycid	対象周期ハンドラ ID 番号
T_RCYC	*pk_rcyc	周期ハンドラ状態を返すパケットへのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)
T_RCYC	*pk_rcyc	周期ハンドラ状態を返すパケットへのポインタ

pk_rcyc の内容

```
typedef   struct t_rcyc{
    STAT    cycstat   +0    2     周期ハンドラの動作状態
    RELTIM  lefttim   +2    4     周期ハンドラ起動までの時間
} T_RCYC;
```

■ アセンブリ言語 API

```
.include mr30.inc
ref_cyc   ID, PK_RCYC
iref_cyc  ID, PK_RCYC
```

● パラメータ

CYCNO	対象周期ハンドラ ID 番号
PK_RCYC	周期ハンドラ状態を返すパケットへのポインタ

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象周期ハンドラ ID 番号
A1	周期ハンドラ状態を返すパケットへのポインタ

■エラーコード

なし

■ 機能説明

cycid で示された周期ハンドラの各種の状態を返します。

◆ **cycstat**

対象周期ハンドラの状態を返します。

・TCYC_STA 周期ハンドラは動作している

・TCYC_STP 周期ハンドラは動作していない

◆ **lefttim**

対象周期ハンドラの次に起動するまでの残り時間を返します。単位は、”ms”です。対象周期ハンドラが動作していない場合は、不定値となります。

本サービスコールは、タスクコンテキストからは、ref_cyc、非タスクコンテキストからは、iref_cyc を使用してください。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include “kernel_id.h”
void task()
{
    T_RCYC rcyc;
    ER ercd;
        :
    ercd = ref_cyc( ID_CYC1, &rcyc );
        :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
_ refcyc:       .blkb   6
task:
        :
      PUSHM   A0,A1
      ref_cyc #ID_CYC1,#_refcyc
        :
```

5.11時間管理機能(アラームハンドラ)

表 5.21に時間管理機能の仕様を示します。項番4アラームハンドラ属性の記述言語は、GUIコンフィギュレータでの指定内容です。コンフィギュレーションファイルには出力されず、カーネルも関知しません。

表 5.21 時間管理機能(アラームハンドラ)の仕様

項番	項目	内容	
1	アラームハンドラ ID	1-255	
2	起床時間	0～0x7FFFFFFF-タイムティック[ms]	
3	拡張情報	16bit	
4	アラームハンドラ属性	TA_HLNG : TA_ASM :	高級言語記述 アセンブリ言語記述

表 5.22 時間管理機能(アラームハンドラ)サービスコール一覧

項番	サービスコール		機能	呼び出し可能なシステム状態					
				T	N	E	D	U	L
1	sta_alm		アラームハンドラの動作開始	○		○	○	○	
2	ista_alm				○	○	○	○	
3	stp_alm		アラームハンドラの動作停止	○		○	○	○	
4	istp_alm				○	○	○	○	
5	ref_alm		アラームハンドラの状態参照	○		○	○	○	
6	iref_alm				○	○	○	○	

【注】

- “[S]”はスタンダードプロファイルのサービスコールです。
- “呼び出し可能なシステム状態”内のそれぞれの記号は、以下の意味です。
  "T"はタスクコンテキストから呼出し可能、"N"は非タスクコンテキストから呼出し可能
  "E"はディスパッチ許可状態から呼出し可能、"D"はディスパッチ禁止状態から呼出し可能
  "U"は CPU ロック解除状態から呼出し可能、"L"は CPU ロック状態から呼出し可能

sta_alm アラームハンドラの動作開始
ista_alm アラームハンドラの動作開始 (ハンドラ専用)

■ C 言語 API

```
ER ercd = sta_alm( ID almid, RELTIM almtim );
ER ercd = ista_alm( ID almid, RELTIM almtim );
```

● パラメータ

ID	almid	対象アラームハンドラ ID 番号
RELTIM	almtim	アラームハンドラの起動時刻 (相対時間)

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語 API

```
.include mr30.inc
sta_alm   ALMID[32]
ista_alm  ALMID[33]
```

● パラメータ

ALMID	対象アラームハンドラ ID 番号
ALMTIM	アラームハンドラの起動時刻(相対時間)

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
R1	アラームハンドラの起動時刻 (相対時間) (下位 16bit)
R3	アラームハンドラの起動時刻 (相対時間) (上位 16bit)
A0	対象アラームハンドラ ID 番号

■ エラーコード

なし

■ 機能説明

almid で示されたアラームハンドラの起動時刻を本サービスコールが呼び出された時刻から、almtim で指定された時間後の時刻と設定し、アラームハンドラを動作している状態に移行させます。

既に動作しているアラームハンドラが指定された場合は、以前の起動時刻の設定を解除し、新しい起動時刻に更新します。almtim に 0 が指定されて場合は、次のタイムティックでアラームハンドラが起動します。almtim に指定可能な値は、(0x7FFFFFFF-タイムティック) 以内でなければいけません。これより大きな値を指定した場合は、正しく動作しません。almtim に 0 を指定した場合は、次のタイムティックにてアラームハンドラを起動します。

本サービスコールは、タスクコンテキストからは、sta_alm、非タスクコンテキストからは、ista_alm を使用してください。

[32] R3(起動時刻上位 16bit)、R1(起動時刻下位 16bit)を格納して呼び出す必要があります。
[33] R3(起動時刻上位 16bit)、R1(起動時刻下位 16bit)を格納して呼び出す必要があります。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
       :
   sta_alm ( ID_alm1,100 );
       :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
        PUSHM   A0,R1,R3
        MOV.W #100,R1
        MOV.W #0,R3
        sta_alm #ID_ALM1
        POPM    A0,R1,R3
          :
```

stp_alm アラームハンドラの動作停止
istp_alm アラームハンドラの動作停止 (ハンドラ専用)

■ C 言語 API

```
ER ercd = stp_alm( ID almid );
ER ercd = istp_alm( ID almid );
```

● パラメータ

ID	almid	対象アラームハンドラ ID 番号

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語 API

```
.include mr30.inc
stp_alm   ALMID
istp_alm  ALMID
```

● パラメータ

ALMID	対象アラームハンドラ ID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象アラームハンドラ ID 番号

■ エラーコード

なし

■ 機能説明

almid で示されたアラームハンドラを動作していない状態に移行させます。

本サービスコールは、タスクコンテキストからは、stp_alm、非タスクコンテキストからは、istp_alm を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
       :
   stp_alm ( ID_alm1 );
       :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
        PUSHM   A0
        stp_alm #ID_ALM1
          :
```

ref_alm アラームハンドラの状態参照
iref_alm アラームハンドラの状態参照 (ハンドラ専用)

■ C 言語 API

```
ER ercd = ref_alm( ID almid, T_RALM *pk_ralm );
ER ercd = iref_alm( ID almid, T_RALM *pk_ralm );
```

● パラメータ

ID	almid	対象アラームハンドラ ID 番号
T_RALM	*pk_ralm	アラームハンドラ状態を返すパケットへのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)
T_RALM	*pk_ralm	アラームハンドラ状態を返すパケットへのポインタ

pk_ralm の内容

```
typedef   struct t_ralm{
    STAT    almstat   +0    2     アラームハンドラの動作状態
    RELTIM  lefttim   +2    4     アラームハンドラ起動までの時間
} T_RALM;
```

■ アセンブリ言語 API

```
.include mr30.inc
ref_alm   ALMID, PK_RALM
iref_alm  ALMID, PK_RALM
```

● パラメータ

ALMID	対象アラームハンドラ ID 番号
PK_RALM	アラームハンドラ状態を返すパケットへのポインタ

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象アラームハンドラ ID 番号
A1	アラームハンドラ状態を返すパケットへのポインタ

■エラーコード

なし

■ 機能説明

almid で示されたアラームハンドラの各種の状態を返します。

◆ **almstat**

対象アラームハンドラの状態を返します。

・TALM_STA　　アラームハンドラは動作している
・TALM_STP　　アラームハンドラは動作していない

◆ **lefttim**

対象アラームハンドラの次に起動するまでの残り時間を返します。単位は、”ms”です。対象アラームハンドラが動作していない場合は、不定値となります。

本サービスコールは、タスクコンテキストからは、ref_alm、非タスクコンテキストからは、iref_alm を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include “kernel_id.h”
 void task()
 {
      T_RALM ralm;
      ER ercd;
           :
      ercd = ref_alm( ID_ALM1, &ralm );
           :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
_ refalm:       .blkb   6
task:
           :
        PUSHM   A0,A1
        ref_alm #ID_ALM1,#_refalm
           :
```

5.12 システム状態管理機能

表 5.23 システム状態管理機能サービスコール一覧

項番	サービスコール		機能	呼び出し可能なシステム状態					
				T	N	E	D	U	L
1	rot_rdq	**[S]**	タスクの優先順位の回転	○		○	○	○	
2	irot_rdq	**[S]**			○	○	○	○	
3	get_tid	**[S]**	実行状態のタスク ID の参照	○		○	○	○	
4	iget_tid	**[S]**			○	○	○	○	
5	loc_cpu	**[S]**	CPU ロック状態への移行	○		○	○	○	○
6	iloc_cpu	**[S]**			○	○	○	○	○
7	unl_cpu	**[S]**	CPU ロック状態の解除	○		○	○	○	○
8	iunl_cpu	**[S]**			○	○	○	○	○
9	dis_dsp	**[S]**	ディスパッチの禁止	○		○	○	○	
10	ena_dsp	**[S]**	ディスパッチの許可	○		○	○	○	
11	sns_ctx	**[S]**	コンテキストの参照	○	○	○	○	○	○
12	sns_loc	**[S]**	CPU ロック状態の参照	○	○	○	○	○	○
13	sns_dsp	**[S]**	ディスパッチ禁止状態の参照	○	○	○	○	○	○
14	sns_dpn	**[S]**	ディスパッチ保留状態の参照	○	○	○	○	○	○

【注】

- “[S]”はスタンダードプロファイルのサービスコールです。
- “呼び出し可能なシステム状態”内のそれぞれの記号は、以下の意味です。
  "T"はタスクコンテキストから呼出し可能、"N"は非タスクコンテキストから呼出し可能
  "E"はディスパッチ許可状態から呼出し可能、"D"はディスパッチ禁止状態から呼出し可能
  "U"は CPU ロック解除状態から呼出し可能、"L"は CPU ロック状態から呼出し可能

rot_rdq タスク優先順位の回転
irot_rdq タスク優先順位の回転 (ハンドラ専用)

■ C 言語 API

```
ER ercd = rot_rdq( PRI tskpri );
ER ercd = irot_rdq( PRI tskpri );
```

● パラメータ

PRI	tskpri	回転するタスク優先度

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語 API

```
.include mr30.inc
rot_rdq   TSKPRI
irot_rdq  TSKPRI
```

● パラメータ

TSKPRI	回転するタスク優先度

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
R3	回転するタスク優先度

■ エラーコード

なし

■ 機能説明

tskpriで示された優先度のレディキューを回転します。すなわち、その優先度のレディキューの先頭につながれているタスクをレディキューの最後尾につなぎかえ、同一優先度のタスクの実行を切り替えます。この様子を図 5.1に示します。

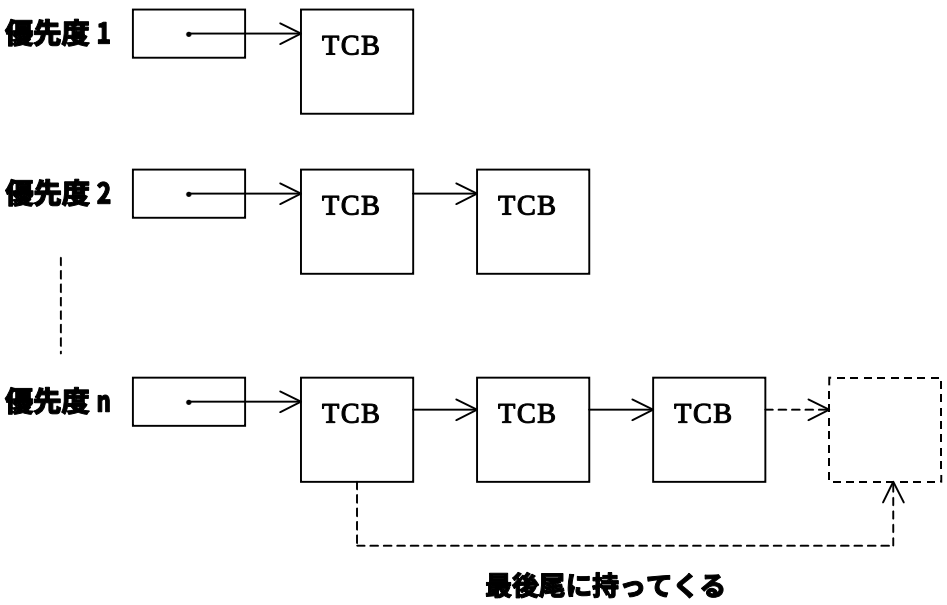


図 5.1 rot_rdq サービスコールによるレディキューの操作

このサービスコールを一定時間間隔で発行することにより、ラウンドロビンスケジューリングをおこなうことができます。rot_rdq サービスコール使用時は、tskpri=TPRI_SELF の指定により、自タスクの持つ優先度のレディキューを回転させます。irot_rdq サービスコールで TPRI_SELF を指定することは出来ません。指定してもエラーとなりません。

また、本サービスコールで自タスクの優先度を指定した場合には、自タスクがそのレディキューの最後尾にまわることになります。なお、指定した優先度のレディキューにタスクがない場合は何も行いません。

本サービスコールは、タスクコンテキストからは、rot_rdq、非タスクコンテキストからは、irot_rdq を使用してください。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
        :
    rot_rdq( 2 );
        :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
        :
      PUSHM   R3
      rot_rdq #2
        :
```

get_tid 実行中タスクIDの参照
iget_tid 実行中タスクIDの参照 (ハンドラ専用)

■ C 言語 API

```
ER ercd = get_tid( ID *p_tskid );
ER ercd = iget_tid( ID *p_tskid );
```

● パラメータ

ID	*p_tskid	タスク ID へのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)
ID	*p_tskid	タスク ID へのポインタ

■ アセンブリ言語 API

```
.include mr30.inc
get_tid
iget_tid
```

● パラメータ

なし

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	獲得したタスク ID

■ エラーコード

なし

■ 機能説明

実行状態のタスク ID を p_tskid の指す領域に返します。タスクから本サービスコールを発行した場合、自タスクの ID 番号を返します。また、非タスクコンテキストから本サービスコールを発行した場合は、そのとき実行していたタスク ID を返します。実行状態のタスクがない場合は、TSK_NONE を返します。

本サービスコールは、タスクコンテキストからは、get_tid、非タスクコンテキストからは、iget_tid を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
   ID tskid;
      :
    get_tid(&tskid);
      :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
        PUSHM   A0
        get_tid
          :
```

loc_cpu CPUロック状態への移行
iloc_cpu CPUロック状態への移行 (ハンドラ専用)

■ C 言語 API

```
ER ercd = loc_cpu();
ER ercd = iloc_cpu();
```

● パラメータ

なし

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語 API

```
.include mr30.inc
loc_cpu
iloc_cpu
```

● パラメータ

なし

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)

■ エラーコード

なし

■ 機能説明

システム状態を CPU ロック状態とし、割込みとタスクのディスパッチを禁止します。CPU ロック状態の特長を以下に示します。

(1) CPU ロック状態の間は、タスクのスケジューリングは行われません。
(2) コンフィギュレータで定義したカーネル割込みマスクレベルより高いレベルの割込み以外の外部割り込みは、受け付けられません。
(3) CPU ロック状態から呼び出し可能なサービスコールは、以下のサービスコールのみです。その他のサービスコールが呼び出された場合の動作は保証されません。

ext_tsk
loc_cpu, iloc_cpu
unl_cpu, iunl_cpu
sns_ctx
sns_loc
sns_dsp
sns_dpn

CPU ロック状態は、以下の操作で解除されます。

(a) unl_cpu, iunl_cpu サービスコールの呼び出し
(b) ext_tsk サービスコールの呼び出し

CPU ロック状態と CPU ロック解除状態の間の遷移は、loc_cpu, iloc_cpu, unl_cpu, iunl_cpu, ext_tsk サービスコールによってのみ発生します。割込みハンドラ、タイムイベントハンドラ終了時には、必ず CPU ロック解除状態でなければなりません。CPU ロック状態の場合、動作は保証されません。なお、これらのハンドラ開始時は、常に CPU ロック解除状態です。

すでに CPU ロック状態のときに、再度本サービスコールを呼び出してもエラーにはなりませんが、キューイングは行いません。

本サービスコールは、タスクコンテキストからは、loc_cpu、非タスクコンテキストからは、iloc_cpu を使用してくだ

さい。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
       :
     loc_cpu();
       :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
        loc_cpu
          :
```

unl_cpu CPUロック状態の解除
iunl_cpu CPUロック状態の解除 (ハンドラ専用)

■ C 言語 API

```
ER ercd = unl_cpu();
ER ercd = iunl_cpu();
```

● パラメータ

なし

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語 API

```
.include mr30.inc
unl_cpu
iunl_cpu
```

● パラメータ

なし

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)

■ エラーコード

なし

■ 機能説明

loc_cpu, iloc_cpu サービスコールによって設定されていた CPU ロック状態を解除します。ディスパッチ許可状態から unl_cpu サービスコールを発行した場合、タスクのスケジューリングが行われます。割込みハンドラ内で iloc_cpu を呼び出し、CPU ロック状態に移行した場合は、割込みハンドラからリターンする前に必ず iunl_cpu を呼び出し、CPU ロック状態を解除してください。

CPU ロック状態とディスパッチ禁止状態は、独立して管理されます。そのため、unl_cpu, iunl_cpu サービスコールでは、ena_dsp サービスコールによるディスパッチ禁止状態は解除されません。

本サービスコールは、タスクコンテキストからは、unl_cpu、非タスクコンテキストからは、iunl_cpu を使用してください。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
      :
    unl_cpu();
      :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
        unl_cpu
          :
```

dis_dsp ディスパッチの禁止

■ C言語API

```
ER ercd = dis_dsp();
```

● パラメータ

なし

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語API

```
.include mr30.inc
dis_dsp
```

● パラメータ

なし

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)

■ エラーコード

なし

■ 機能説明

システム状態をディスパッチ禁止状態にします。ディスパッチ禁止状態の特長を、以下に示します。

(1) タスクのスケジューリングが行われなくなるため、自タスク以外のタスクが実行状態に移行することはなくなります。
(2) 割込みは受け付けられます。
(3) 待ち状態になるサービスコールを呼び出せません。

ディスパッチ禁止状態の間に以下の操作を行うと、システム状態はタスク実行状態に戻ります。

(a) ena_dsp サービスコールの呼び出し
(b) ext_tsk サービスコールの呼び出し

ディスパッチ禁止状態とディスパッチ許可状態の間の遷移は、dis_dsp, ena_dsp, ext_tsk サービスコールによってのみ発生します。
すでにディスパッチ禁止状態のときに再度本サービスコールを呼び出してもエラーにはなりませんが、キューイングは行いません。
本サービスコールは、タスクコンテキストにおいてのみ使用可能です。非タスクコンテキストにおいて使用した場合は正常に動作しません。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
       :
     dis_dsp();
       :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
        dis_dsp
          :
```

ena_dsp ディスパッチの許可

■ C言語API

```
ER ercd = ena_dsp();
```

● パラメータ

なし

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語API

```
.include mr30.inc
ena_dsp
```

● パラメータ

なし

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)

■ エラーコード

なし

■ 機能説明

dis_dsp サービスコールによって設定されていたディスパッチ禁止状態を解除します。それにより、システムがタスク実行状態になった場合は、タスクのスケジューリングが行われます。

タスク実行状態から本サービスコールを呼び出してもエラーにはなりませんが、キューイングは行いません。

本サービスコールは、タスクコンテキストにおいてのみ使用可能です。非タスクコンテキストにおいて使用した場合は正常に動作しません。

■ 使用例

《 C言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
      :
    ena_dsp();
      :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
        ena_dsp
          :
```

sns_ctx コンテキストの参照

■ C 言語 API

```
BOOL state = sns_ctx();
```

● パラメータ

なし

● リターンパラメータ

BOOL	state	TRUE:非タスクコンテキスト FALSE:タスクコンテキスト

■ アセンブリ言語 API

```
.include mr30.inc
sns_ctx
```

● パラメータ

なし

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	TRUE:非タスクコンテキスト FALSE:タスクコンテキスト

■ エラーコード

なし

■ 機能説明

非タスクコンテキストから呼び出された場合に TRUE、タスクコンテキストから呼び出された場合に FALSE を返します。本サービスコールは、CPU ロック状態からも呼び出せます。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
  BOOL stat;
     :
   stat = sns_ctx();
     :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
        sns_ctx
          :
```

sns_loc CPUロック状態の参照

■ C言語API

```
BOOL state = sns_loc();
```

● パラメータ

なし

● リターンパラメータ

BOOL	state	TRUE:CPUロック状態 FALSE:CPUロック解除状態

■ アセンブリ言語API

```
.include mr30.inc
sns_loc
```

● パラメータ

なし

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	TRUE:CPUロック状態 FALSE:CPUロック解除状態

■ エラーコード

なし

■ 機能説明

システムが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。本サービスコールは、CPU ロック状態からも呼び出せます。

■ 使用例

《 C言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
  BOOL stat;
     :
   stat = sns_loc();
     :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
        sns_loc
          :
```

sns_dsp ディスパッチ禁止状態の参照

■ C 言語 API

BOOL state = sns_dsp();

● パラメータ

なし

● リターンパラメータ

BOOL	state	TRUE：ディスパッチ禁止状態 FALSE：ディスパッチ許可状態

■ アセンブリ言語 API

.include mr30.inc
sns_dsp

● パラメータ

なし

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	TRUE：ディスパッチ禁止状態 FALSE：ディスパッチ許可状態

■ エラーコード

なし

■ 機能説明

システムが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。
本サービスコールは、CPU ロック状態からも呼び出せます。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
   BOOL stat;
      :
    stat = sns_dsp();
      :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
         :
       sns_dsp
         :
```

sns_dpn ディスパッチ保留状態の参照

■ C 言語 API

```
BOOL state = sns_dpn();
```

● パラメータ

なし

● リターンパラメータ

BOOL	state	TRUE:ディスパッチ保留状態 FALSE:ディスパッチ保留状態ではない

■ アセンブリ言語 API

```
.include mr30.inc
sns_dpn
```

● パラメータ

なし

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	TRUE:ディスパッチ保留状態 FALSE:ディスパッチ保留状態ではない

■ エラーコード

なし

■ 機能説明

システムがディスパッチ保留状態の場合に TRUE、そうでない場合に FALSE を返します。具体的には、以下の全ての条件が満足される場合に FALSE を返し、その他の場合には TRUE を返します。

(1) ディスパッチ禁止状態でない
(2) CPU ロック状態でない
(3) タスクである

本サービスコールは、CPU ロック状態からも呼び出せます。システムが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。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
  BOOL stat;
     :
   stat = sns_dpn();
     :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
         :
       sns_dpn
         :
```

5.13割込管理機能

表 5.24 割り込み管理機能サービスコール一覧

項番	サービスコール		機能	呼び出し可能なシステム状態					
				T	N	E	D	U	L
1	ret_int		割り込みハンドラからの復帰		○	○	○	○	

【注】

- “[S]”はスタンダードプロファイルのサービスコールです。
- “呼び出し可能なシステム状態”内のそれぞれの記号は、以下の意味です。
  "T"はタスクコンテキストから呼出し可能、"N"は非タスクコンテキストから呼出し可能
  "E"はディスパッチ許可状態から呼出し可能、"D"はディスパッチ禁止状態から呼出し可能
  "U"は CPU ロック解除状態から呼出し可能、"L"は CPU ロック状態から呼出し可能

ret_int 割り込みハンドラからの復帰(アセンブリ言語記述時)

■ C 言語 API

本サービスコールは、C 言語では記述できません。[34]

■ アセンブリ言語による呼び出し方法

```
.include mr30.inc
ret_int
```

■ パラメータ

なし

■ エラーコード

本サービスコールを発行した割り込みハンドラには戻りません。

■ 機能説明

割り込みハンドラからの復帰処理を行います。復帰処理に応じてスケジューラを動作させ、タスクの切り替えを行います。

割り込みハンドラの中でサービスコールを実行してもタスク切り替えは起こらず、割り込みハンドラを終了するまでタスク切り替えが遅延されます。

ただし、多重割り込み発生により起動された割り込みハンドラからの ret_int サービスコールの発行の場合はスケジューラを動作させません。タスクからの割り込みの場合のみスケジューラを動作させます。

なお、アセンブリ言語で記述する場合、本サービスコールは割り込みハンドラ入り口ルーチンから呼ばれたサブルーチンからは発行できません。必ず、割り込みハンドラの入り口ルーチンまたは入り口関数内で本サービスコールを実行してください。すなわち、以下のようなプログラムは正常に動作しません。

```
.include mr30.inc
/* NG */
.GLB intr
intr:
    jsr.b func
        :
func:
    ret_int
```

すなわち、以下のように記述してください。

```
.include mr30.inc
/* OK */
.GLB intr
intr:
    jsr.b func
    ret_int
func:
        :
   rts
```

本サービスコールは割り込みハンドラからのみ発行してください。周期起動ハンドラ、アラームハンドラ及びタスクから発行した場合は、正常に動作しません。

[34] 割り込みハンドラの開始関数を#pragma INTHANDLER で宣言すると、関数の出口で自動的に ret_int サービスコールを発行します。

5.14 システム構成理機能

表 5.25 システム構成管理機能サービスコール一覧

項番	サービスコール		機能	呼び出し可能なシステム状態					
				T	N	E	D	U	L
1	ref_ver	**[S]**	バージョン情報の参照	○		○	○	○	
2	iref_ver				○	○	○	○	

【注】

- “[S]”はスタンダードプロファイルのサービスコールです。
- “呼び出し可能なシステム状態”内のそれぞれの記号は、以下の意味です。
  "T"はタスクコンテキストから呼出し可能、"N"は非タスクコンテキストから呼出し可能
  "E"はディスパッチ許可状態から呼出し可能、"D"はディスパッチ禁止状態から呼出し可能
  "U"は CPU ロック解除状態から呼出し可能、"L"は CPU ロック状態から呼出し可能

ref_ver バージョン情報の参照
iref_ver バージョン情報の参照 (ハンドラ専用)

■ C 言語 API

```
ER ercd = ref_ver( T_RVER *pk_rver );
ER ercd = iref_ver( T_RVER *pk_rver );
```

● パラメータ

T_RVER	*pk_rver	バージョン情報を返すパケットへのポインタ

pk_rver の内容

```
typedef   struct t_rver {
    UH      maker     0     2     カーネルのメーカーコード
    UH      prid      +2    2     カーネルの識別番号
    UH      spver     +4    2     ITRON 仕様のバージョン番号
    UH      prver     +6    2     カーネルのバージョン番号
    UH      prno[4]   +8    2     カーネル製品の管理情報
} T_RVER;
```

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語 API

```
.include mr30.inc
ref_ver   PK_VER
iref_ver  PK_VER
```

● パラメータ

PK_VER	バージョン情報を返すパケットへのポインタ

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	バージョン情報を返すパケットへのポインタ

■ エラーコード

なし

■ 機能説明

現在実行中のカーネルのバージョンに関する情報を読み出し、その結果を pk_rver の指す領域に返します。pk_rver の指すパケットには、次の情報を返します。

- **maker**
  株式会社ルネサスエレクトロニクスを示す H’11B が返されます。
- **prid**
  M3T-MR30 の内部識別 IDH’130 が返されます。
- **spver**
  μITRON 仕様書 Ver4.02.00 に準拠していることを示す H'5402 が返されます。
- **prver**
  M3T-MR30 カーネルのバージョンを示す H'0410 が返されます。
- **prno**
  - prno[0]
    拡張のための予約。
  - prno[1]
    拡張のための予約。
  - prno[2]
    拡張のための予約。
  - prno[3]
    拡張のための予約。

本サービスコールは、タスクコンテキストからは、ref_ver、非タスクコンテキストからは、iref_ver を使用してください。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include “kernel_id.h”
void task()
{
    T_RVER   pk_rver;
    ref_ver( &pk_rver );
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
_ refver:       .blkb   6
task:
          :
        PUSHM   A0
        ref_ver #_refver
          :
```

5.15拡張機能(longデータキュー)

表 5.26にデータキュー機能の仕様を示します。本機能は、データキューのデータを 32bitで扱います。本機能は、μITRON 4.0 仕様の仕様外の機能です。

表 5.26 long データキュー機能の仕様

項番	項目	内容
1	long データキューID	1～255
2	long データキュー領域の容量 (データの個数)	0～8191
3	データサイズ	32 ビット
4	long データキュー属性	TA_TFIFO : 待ちタスクのキューイングは FIFO 順 TA_TPRI : 待ちタスクのキューイングは優先度順

表 5.27 long データキュー機能サービスコール一覧

項番	サービスコール		機能	呼び出し可能なシステム状態					
				T	N	E	D	U	L
1	vsnd_dtq		long データキューへの送信	○		○		○	
2	vpsnd_dtq		同上(ポーリング)	○		○	○	○	
3	vipsnd_dtq				○	○	○	○	
4	vtsnd_dtq		同上(タイムアウト有)	○		○		○	
5	vfsnd_dtq		long データキューへの強制送信	○		○	○	○	
6	vifsnd_dtq				○	○	○	○	
7	vrcv_dtq		long データキューからの受信	○		○		○	
8	vprcv_dtq		同上(ポーリング)	○		○	○	○	
9	viprcv_dtq				○	○	○	○	
10	vtrcv_dtq		同上(タイムアウト有)	○		○		○	
11	vref_dtq		long データキューの状態参照	○		○	○	○	
12	viref_dtq				○	○	○	○	

【注】

- “[S]”はスタンダードプロファイルのサービスコールです。
- “呼び出し可能なシステム状態”内のそれぞれの記号は、以下の意味です。
  "T"はタスクコンテキストから呼出し可能、"N"は非タスクコンテキストから呼出し可能
  "E"はディスパッチ許可状態から呼出し可能、"D"はディスパッチ禁止状態から呼出し可能
  "U"は CPU ロック解除状態から呼出し可能、"L"は CPU ロック状態から呼出し可能

vsnd_dtq	longデータキューへのデータ送信
vpsnd_dtq	longデータキューへのデータ送信 (ポーリング)
vipsnd_dtq	longデータキューへのデータ送信 (ポーリング、ハンドラ専用)
vtsnd_dtq	longデータキューへのデータ送信 (タイムアウト)
vfsnd_dtq	longデータキューへのデータ強制送信
vifsnd_dtq	longデータキューへのデータ強制送信 (ハンドラ専用)

■ C言語API

```
ER ercd = vsnd_dtq( ID vdtqid, W data );
ER ercd = vpsnd_dtq( ID vdtqid, W data );
ER ercd = vipsnd_dtq( ID vdtqid, W data );
ER ercd = vtsnd_dtq( ID vdtqid, W data, TMO tmout );
ER ercd = vfsnd_dtq( ID vdtqid, W data );
ER ercd = vifsnd_dtq( ID vdtqid, W data );
```

● パラメータ

ID	vdtqid	対象 long データキューID 番号
TMO	tmout	タイムアウト値(vtsnd_dtq の場合)
W	data	送信するデータ

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード

■ アセンブリ言語API

```
.include mr30.inc
vsnd_dtq        VDTQID[35]
visnd_dtq       VDTQID[36]
vpsnd_dtq       VDTQID[37]
vipsnd_dtq      VDTQID[38]
vtsnd_dtq       VDTQID[39][40]
vfsnd_dtq       VDTQID[41]
vifsnd_dtq      VDTQID[42]
```

● パラメータ

VDTQID	対象 long データキューID 番号
DTQDATA	送信するデータ

[35] R3(データ上位 16bit)、R1(データ下位 16bit)を格納して呼び出す必要があります。
[36] R3(データ上位 16bit)、R1(データ下位 16bit)を格納して呼び出す必要があります。
[37] R3(データ上位 16bit)、R1(データ下位 16bit)を格納して呼び出す必要があります。
[38] R3(データ上位 16bit)、R1(データ下位 16bit)を格納して呼び出す必要があります。
[39] R3(データ上位 16bit)、R1(データ下位 16bit)を格納して呼び出す必要があります。
[40] R2(タイムアウト値上位 16bit)、R0(タイムアウト値下位 16bit)を格納して呼び出す必要があります。
[41] R3(データ上位 16bit)、R1(データ下位 16bit)を格納して呼び出す必要があります。
[42] R3(データ上位 16bit)、R1(データ下位 16bit)を格納して呼び出す必要があります。

● サービスコール発行後のレジスタ内容

vsnd_dtq, vpsnd_dtq, vipsnd_dtq, vfsnd_dtq, vifsnd_dtq の場合

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R1	データ(上位 16bit)
R3	データ(下位 16bit)
A0	対象 long データキューID 番号

vtsnd_dtq の場合

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R1	データ(上位 16bit)
R2	タイムアウト値(上位 16bit)
R3	データ(下位 16bit)
A0	対象 long データキューID 番号

■ エラーコード

E_RLWAI	待ち状態強制解除
E_TMOUT	ポーリング失敗、またはタイムアウト
E_ILUSE	サービスコール不正使用(dtqcnt が 0 の long データキューに対して fsnd_dtq, ifsnd_dtq を発行)
EV_RST	long データキュー領域クリアによって待ち状態が解除された

■ 機能説明

vdtqid で示された long データキューに対して、data で示された符号付き 32bit のデータを送信します。対象 long データキューに受信待ちタスクが存在する場合は、long データキューにデータを格納せず、受信待ち行列の先頭タスクにデータを送信し、そのタスクの受信待ち状態を解除します。

一方、既にデータで一杯になった long データキューに対して、vsnd_dtq, vtsnd_dtq を発行した場合、これらのサービスコールを発行したタスクは、実行状態からデータ送信待ち状態に移行し、long データキューの空きを待つ送信待ち行列につながれます。その際、vdtqid の long データキュー属性が TA_TFIFO の場合は、FIFO 順で待ち行列にタスクをつなぎ、TA_TPRI の場合は、優先度順でタスクをつなぎます。vpsnd_dtq, vipsnd_dtq の場合は、直ちにリターンし、エラーE_TMOUT を返します。

vtsnd_dtq サービスコールの場合は、tmout には、待ち時間を ms 単位で指定します。tmout に指定可能な値は、(0x7FFFFFFF-タイムティック)以内でなければいけません。これより大きな値を指定した場合は、正しく動作しません。tmout に TMO_POL=0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、vpsnd_dtq と同じ動作をします。また、tmout=TMO_FEVR(-1)にした場合は、永久待ちの指定で、vsnd_dtq サービスコールと同じ動作をします。

受信待ちタスクがなく、long データキュー領域も一杯でない場合、送信したデータは long データキューに格納されます。

vsnd_dtq, vtsnd_dtq サービスコール実行による待ち状態は、以下に示す場合に解除されます。

- **tmout の時間が経過する前に、vrcv_dtq,vtrcv_dtq,vprcv_dtq,viprcv_dtq サービスコールが発行され、待ち解除条件が満足された場合**
  この場合、エラーコードは、E_OK を返します。
- **待ち解除条件が満足されないまま、tmout 経過し、最初のタイムティックが発生した場合**
  この場合、エラーコードは、E_TMOUT を返します。

- **他のタスクおよびハンドラから発行した rel_wai、irel_wai サービスコールによって待ち状態が強制解除された場合**
  この場合、エラーコードは、E_RLWAI を返します。
- **他のタスクから発行した vrst_vdtq サービスコールによって待ち状態の対象となっている long データキューが削除された場合**
  この場合、エラーコードは、EV_RST を返します。

vfsnd_dtq, vifsnd_dtq の場合は、long データキューの先頭(最古)のデータを削除し、送信データを long データキュー末尾に格納します。long データキュー領域がデータで一杯になっていない場合は、vfsnd_dtq, vifsnd_dtq は、vsnd_dtq と同じ動作を行います。

タスクコンテキストにおいては、vsnd_dtq, vtsnd_dtq, vpsnd_dtq, vfsnd_dtq サービスコール、非タスクコンテキストにおいては、vipsnd_dtq, vifsnd_dtq を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include “kernel_id.h”
 W data[10];
 void task(void)
 {
                :
     if( vsnd_dtq( ID_dtq, data[0]) == E_RLWAI ){
       error(“Forced released¥n”);
   }
                :
     if( vpsnd_dtq( ID_dtq, data[1])== E_TMOUT ){
       error(“Timeout¥n”);
   }
                :
     if( vtsnd_dtq( ID_dtq, data[2], 10 ) != E_ TMOUT ){
       error(“Timeout ¥n”);
   }
                :
     if( vfsnd_dtq( ID_dtq, data[3]) != E_OK ){
       error(“error¥n”);
   }
                :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
_g_dtq: .WORD   1234H
        .WORD   5678H
task:
          :
    PUSHM       R0,R1,R2,R3,A0
    MOV.W       _g_dtq,R1
    MOV.W       _g_dtq+2,R3
    MOV.W       #100,R0
    MOV.W       #0,R2
    vtsnd_dtq   #ID_DTQ1
          :
    PUSHM       R1,R3,A0
    MOV.W       #1234H,R1
    MOV.W       #5678H,R3
    vpsnd_dtq   #ID_DTQ2
          :
    PUSHM       R1,R3,A0
    MOV.W       #1234H,R1
    MOV.W       #5678H,R3
    vfsnd_dtq   #ID_DTQ3
          :
```

vrcv_dtq longデータキューからのデータ受信
vprcv_dtq longデータキューからのデータ受信 (ポーリング)
viprcv_dtq longデータキューからのデータ受信 (ポーリング、ハンドラ専用)
vtrcv_dtq longデータキューからのデータ受信 (タイムアウト)

■ C言語API

```
ER ercd = vrcv_dtq( ID dtqid, W *p_data );
ER ercd = vprcv_dtq( ID dtqid, W *p_data );
ER ercd = viprcv_dtq( ID dtqid, W *p_data );
ER ercd = vtrcv_dtq( ID dtqid, W *p_data, TMO tmout );
```

● パラメータ

ID	vdtqid	対象 long データキューID 番号
TMO	tmout	タイムアウト値(vtrcv_dtq の場合)
W	*p_data	受信データを格納する領域先頭へのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)またはエラーコード
W	*p_data	受信データを格納する領域先頭へのポインタ

■ アセンブリ言語API

```
.include mr30.inc
vrcv_dtq        VDTQID
vprcv_dtq       VDTQID
viprcv_dtq      VDTQID
vtrcv_dtq       VDTQID[43]
```

● パラメータ

VDTQID	対象 long データキューID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)またはエラーコード
R1	受信データ(上位 16bit)
R3	受信データ(下位 16bit)
A0	対象 long データキューID 番号

[43] R3(タイムアウト値上位 16bit)、R1(タイムアウト値下位 16bit)を格納して呼び出す必要があります。

■ エラーコード

E_RLWAI	待ち状態強制解除
E_TMOUT	ポーリング失敗、またはタイムアウト

■ 機能説明

vdtqid で示された long データキューから、データを受信し、p_data の指す領域に格納します。対象 long データキューにデータが存在する場合は、その先頭の(最古の)データを受信します。この結果、long データキュー領域に空きが発生するため、送信待ち行列につながれているタスクは、その送信待ち状態が解除され、long データキュー領域へのデータを送信します。

long データキューにデータが存在せず、データ送信待ちタスクが存在する場合（long データキュー領域の容量が 0 の場合）、データ送信待ち行列先頭タスクのデータを受信します。この結果、そのデータ送信待ちタスクの待ち状態は解除されます。

一方、long データキュー領域にデータが格納されていない long データキューに対して、vrcv_dtq, vtrcv_dtq を発行した場合、これらのサービスコールを発行したタスクは、実行状態からデータ受信待ち状態に移行し、データ受信待ち行列につながれます。このとき、受信待ち行列へは、FIFO 順でつながれます。vprcv_dtq, viprcv_dtq の場合は、直ちにリターンし、エラーE_TMOUT を返します。

vtrcv_dtq サービスコールの場合は、tmout には、待ち時間を ms 単位で指定します。tmout に指定可能な値は、(0x7FFFFFFF-タイムティック)以内でなければいけません。これより大きな値を指定した場合は、正しく動作しません。tmout に TMO_POL=0 を指定した場合は、タイムアウト値として 0 を指定したことを示し、vprcv_dtq と同じ動作をします。また、tmout=TMO_FEVR(-1)にした場合は、永久待ちの指定で、vrcv_dtq サービスコールと同じ動作をします。

vrcv_dtq, vtrcv_dtq サービスコール実行による待ち状態は、以下に示す場合に解除されます。

- **tmout の時間が経過する前に、vsnd_dtq, vtsnd_dtq, vpsnd_dtq, vipsnd_dtq サービスコールが発行され、待ち解除条件が満足された場合**
  この場合、エラーコードは、E_OK を返します。
- **待ち解除条件が満足されないまま、tmout 経過し、最初のタイムティックが発生した場合**
  この場合、エラーコードは、E_TMOUT を返します。
- **他のタスクおよびハンドラから発行した rel_wai、irel_wai サービスコールによって待ち状態が強制解除された場合**
  この場合、エラーコードは、E_RLWAI を返します。

タスクコンテキストにおいては、vrcv_dtq, vtrcv_dtq, vprcv_dtq サービスコール、非タスクコンテキストにおいては、viprcv_dtq を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include “kernel_id.h”
 void task()
 {
   W data;
                :
     if( vrcv_dtq( ID_dtq, &data ) != E_RLWAI )
         error(“forced wakeup¥n”);
                :
     if( vprcv_dtq( ID_dtq, &data ) != E_TMOUT )
         error(“Timeout¥n”);
                :
     if( vtrcv_dtq( ID_dtq, &data, 10 ) != E_TMOUT )
         error(“Timeout¥n”);
                :
 }
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
         :
    PUSHM       A0,R3
    MOV.W       #0,R1
    MOV.W       #0,R3
    vtrcv_dtq   #ID_DTQ1
                  :
    PUSHM       A0
    vprcv_dtq   #ID_DTQ2
         :
    PUSHM       A0
    vrcv_dtq    #ID_DTQ2
         :
```

vref_dtq longデータキューの状態参照
viref_dtq longデータキューの状態参照 (ハンドラ専用)

■ C言語API

```
ER ercd = vref_dtq( ID vdtqid, T_RDTQ *pk_rdtq );
ER ercd = viref_dtq( ID vdtqid, T_RDTQ *pk_rdtq );
```

● パラメータ

ID	vdtqid	対象 long データキューID 番号
T_RDTQ	*pk_rdtq	long データキュー状態を返すパケットへのポインタ

● リターンパラメータ

ER	ercd	正常終了(E_OK)
T_RDTQ	*pk_rdtq	long データキュー状態を返すパケットへのポインタ

pk_rdtq の内容

```
typedef   struct t_rdtq{
    ID      stskid   +0   2    送信待ちタスク ID
    ID      wtskid   +2   2    受信待ちタスク ID
    UINT    sdtqcnt  +4   2    long データキューに入っているデータ数
} T_RDTQ;
```

■ アセンブリ言語API

```
.include mr30.inc
vref_dtq  VDTQID, PK_RDTQ
viref_dtq VDTQID, PK_RDTQ
```

● パラメータ

VDTQID	対象 long データキューID 番号
PK_RDTQ	long データキュー状態を返すパケットへのポインタ

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象 long データキューID 番号
A1	long データキュー状態を返すパケットへのポインタ

■エラーコード

なし

■ 機能説明

vdtqid で示された long データキューの各種の状態を返します。

- **stskid**
  stskid には送信待ち行列の先頭タスク(次に待ち行列から削除されるタスク)の ID 番号を返します。待ちタスクの無い場合は TSK_NONE を返します。
- **wtskid**
  wtskid には受信待ち行列の先頭タスク(次に待ち行列から削除されるタスク)の ID 番号を返します。待ちタスクの無い場合は TSK_NONE を返します。
- **sdtqcnt**
  sdtqcnt には、long データキュー領域に格納されているデータ個数を返します。

本サービスコールは、タスクコンテキストからは、ref_dtq、非タスクコンテキストからは、iref_dtq を使用してください。

■ 使用例

《 C 言語の使用例 》

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
     T_RDTQ rdtq;
     ER ercd;
         :
     ercd = vref_dtq( ID_DTQ1, &rdtq );
         :
 }
```

《 アセンブリ言語の使用例 》

```
_ refdtq:       .blkb   6
    .include mr30.inc
    .GLB        task
task:
         :
        PUSHM   A0,A1
        vref_dtq        #ID_DTQ1,#_refdtq
         :
```

5.16 拡張機能(リセット機能)

本機能は、オブジェクトの内容を初期化する機能です。本機能は、μITRON 4.0 仕様の仕様外の機能です。

表 5.28 拡張機能機能(リセット機能)サービスコール一覧

項番	サービスコール		機能	呼び出し可能なシステム状態					
				T	N	E	D	U	L
1	vrst_dtq		データキューのリセット	○		○	○	○	
2	vrst_vdtq		long データキューのリセット	○		○	○	○	
3	vrst_mbx		メールボックスのリセット	○		○	○	○	
4	vrst_mpf		固定長メモリプールのリセット	○		○	○	○	
5	vrst_mpl		可変長メモリプールのリセット	○		○	○	○	

【注】

- “[S]”はスタンダードプロファイルのサービスコールです。
- “呼び出し可能なシステム状態”内のそれぞれの記号は、以下の意味です。
  "T"はタスクコンテキストから呼出し可能、"N"は非タスクコンテキストから呼出し可能
  "E"はディスパッチ許可状態から呼出し可能、"D"はディスパッチ禁止状態から呼出し可能
  "U"は CPU ロック解除状態から呼出し可能、"L"は CPU ロック状態から呼出し可能

vrst_dtq データキュー領域のクリア

■ C 言語 API

```
ER ercd = vrst_dtq( ID dtqid );
```

● パラメータ

ID	dtqid	対象データキューID 番号

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語 API

```
.include mr30.inc
vrst_dtq  DTQID
```

● パラメータ

DTQID	対象データキューID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象データキューID 番号

■ エラーコード

なし

■ 機能説明

dtqid で示されたデータキューに格納されているデータをクリアします。データキュー領域にさらに追加するエリアがなく、データ送信待ち行列にタスクがつながれている場合、データ送信待ち行列につながれているすべてのタスクの待ち状態を解除します。さらに解除されたタスクに対してエラーEV_RST が返されます。

また、データキューの定義個数がゼロ個の場合においても、データ送信待ち行列につながれているすべてのタスクの待ち状態を解除します。

本サービスコールは、タスクコンテキストにおいてのみ使用可能です。非タスクコンテキストにおいて使用した場合は正常に動作しません。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task1(void)
{
      :
    vrst_dtq( ID_dtq1 );
      :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       A0
    vrst_dtq    #ID_DTQ1
          :
```

vrst_vdtq longデータキュー領域のクリア

■ C言語API

```
ER ercd = vrst_vdtq( ID vdtqid );
```

● パラメータ

ID	vdtqid	対象 short データキューID

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語API

```
.include mr30.inc
vrst_vdtq VDTQID
```

● パラメータ

VDTQID	対象 long データキューID

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象 long データキューID

■ エラーコード

なし

■ 機能説明

vdtqid で示された long データキューに格納されているデータをクリアします。long データキュー領域にさらに追加するエリアがなく、データ送信待ち行列にタスクがつながれている場合、データ送信待ち行列につながれているすべてのタスクの待ち状態を解除します。さらに解除されたタスクに対してエラーEV_RST が返されます。また、long データキューの定義個数がゼロ個の場合においても、データ送信待ち行列につながれているすべてのタスクの待ち状態を解除します。

本サービスコールは、タスクコンテキストにおいてのみ使用可能です。非タスクコンテキストにおいて使用した場合は正常に動作しません。

■ 使用例

《 C言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task1(void)
{
      :
    vrst_vdtq( ID_vdtq1 );
      :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       A0
    vrst_vdtq   #ID_VDTQ1
          :
```

vrst_mbx メールボックス領域のクリア

■ C 言語 API

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

● パラメータ

ID	mbxid	対象メールボックス ID 番号

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語 API

```
.include mr30.inc
vrst_mbx  MBXID
```

● パラメータ

MBXID	対象メールボックス ID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象メールボックス ID 番号

■ エラーコード

なし

■ 機能説明

mbxid で示されたメールボックスに格納されているメッセージをクリアします。

本サービスコールは、タスクコンテキストにおいてのみ使用可能です。非タスクコンテキストにおいて使用した場合は正常に動作しません。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task1(void)
{
      :
    vrst_mbx( ID_mbx1 );
      :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       A0
    vrst_mbx    #ID_MBX1
          :
```

vrst_mpf 固定長メモリプール領域のクリア

■ C 言語 API

```
ER ercd = vrst_mpf( ID mpfid );
```

● パラメータ

ID	mpfid	対象固定長メモリプール ID 番号

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語 API

```
.include mr30.inc
vrst_mpf  MPFID
```

● パラメータ

MPFID	対象固定長メモリプール ID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象固定長メモリプール ID 番号

■ エラーコード

なし

■ 機能説明

mpfid で示された固定長メモリプール初期状態にします。メモリブロック獲得待ち行列にタスクがつながれている場合、メモリブロック獲得待ち行列につながれているすべてのタスクの待ち状態を解除します。さらに解除されたタスクに対してエラーEV_RST が返されます。

本サービスコールは、タスクコンテキストにおいてのみ使用可能です。非タスクコンテキストにおいて使用した場合は正常に動作しません。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task1(void)
{
      :
    vrst_mpf( ID_mpf1 );
      :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       A0
    vrst_mpf    #ID_MPF1
          :
```

vrst_mpl 可変長メモリプール領域のクリア

■ C 言語 API

```
ER ercd = vrst_mpl( ID mplid );
```

● パラメータ

ID	mplid	対象可変長メモリプール ID 番号

● リターンパラメータ

ER	ercd	正常終了(E_OK)

■ アセンブリ言語 API

```
.include mr30.inc
vrst_mpl  MPLID
```

● パラメータ

MPLID	対象可変長メモリプール ID 番号

● サービスコール発行後のレジスタ内容

レジスタ名	サービスコール発行後の内容
R0	正常終了(E_OK)
A0	対象可変長メモリプール ID 番号

■ エラーコード

なし

■ 機能説明

mplid で示された可変長メモリプールを初期状態にします。

本サービスコールは、タスクコンテキストにおいてのみ使用可能です。非タスクコンテキストにおいて使用した場合は正常に動作しません。

■ 使用例

《 C 言語の使用例 》

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task1(void)
{
      :
    vrst_mpl( ID_mpl1 );
      :
}
```

《 アセンブリ言語の使用例 》

```
    .include mr30.inc
    .GLB        task
task:
          :
    PUSHM       A0
    vrst_mpl    #ID_MPL1
          :
```

6. アプリケーション作成手順概要

6.1 概要

MR30 のアプリケーションプログラムは一般的に以下に示す手順で開発します。

1. プロジェクトの生成

High-performance Embedded Workshop を使用する場合は、High-performance Embedded Workshop 上で MR30 を使用したプロジェクトを新規に作成します。

2. アプリケーションプログラムのコーディング

C 言語もしくはアセンブリ言語を用いてアプリケーションプログラムをコーディングします。必要があればサンプルのスタートアッププログラム(crt0mr.a30)、セクション定義ファイル(c_sec.inc もしくは asm_sec.inc)を修正してください。

3. コンフィギュレーションファイル作成

タスクのエントリーアドレスやスタックサイズなどを定義したコンフィギュレーションファイルをエディタなどで作成します。GUI コンフィギュレータを用いてコンフィギュレーションファイルを作成することも出来ます。

4. システム生成

High-performance Embedded Workshop 上でビルドを実行してシステムを生成します。

5. ROM 書き込み

作成された ROM 書き込み形式ファイルにより、ROM に書き込みます。もしくはデバッガに読み込ませてデバッグを行います。

図 6.1にシステム生成の詳細フローを示します。

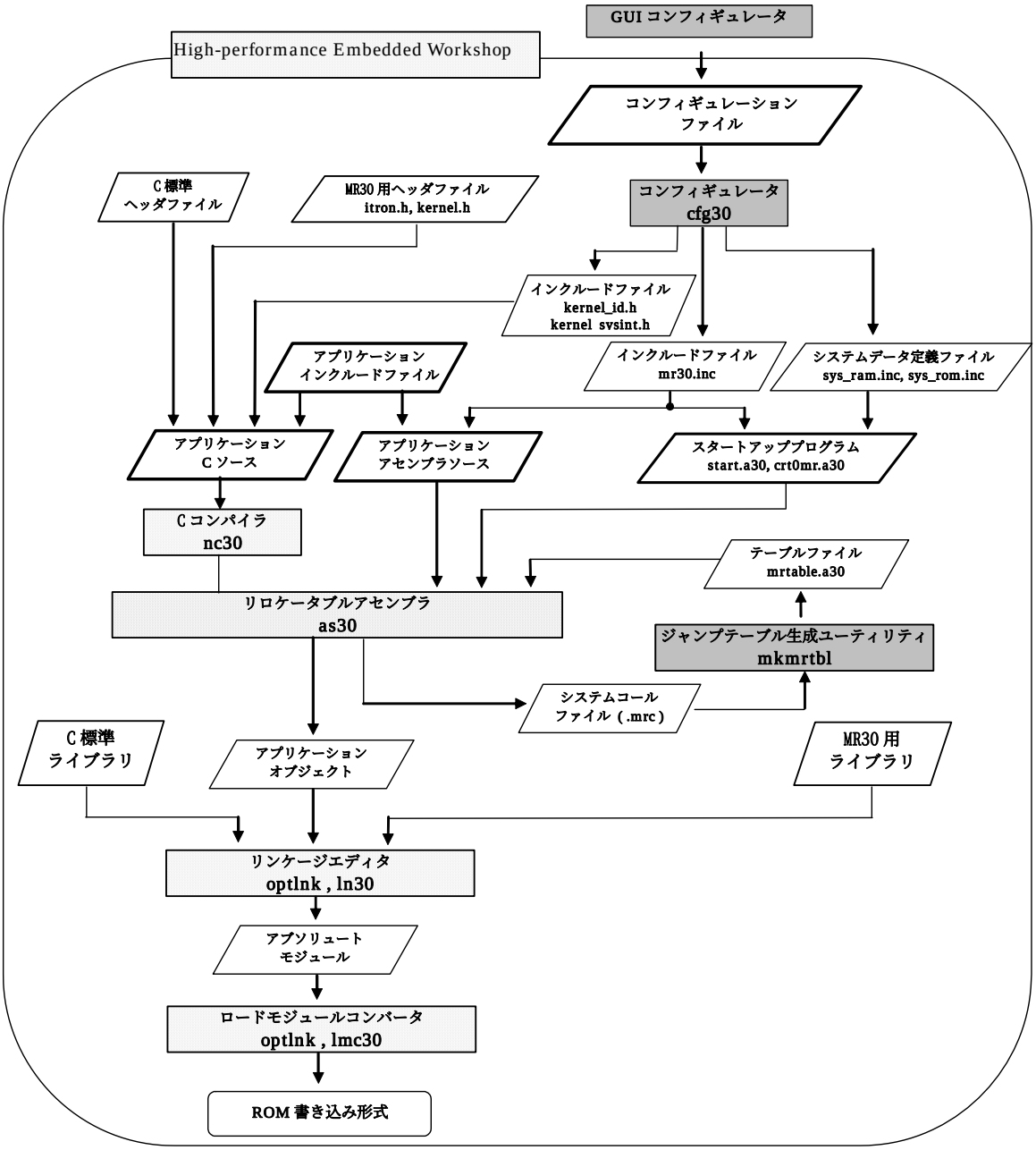


図 6.1 MR30 システム生成フロー

7. アプリケーション作成手順詳細

7.1 C言語によるコーディング方法

本節では、C 言語を用いてアプリケーションプログラムを記述する方法について述べます。

7.1.1 タスクの記述方法

C 言語を用いてタスクを記述する場合、以下の項目に注意してください。

1. タスクは関数として記述します。

タスクを MR30 に登録するにはコンフィギュレーションファイルに関数名を記述します。例えば関数名 "task()" をタスク ID 番号3で登録するには以下のようにおこないます。

```
task[3]{
        name            = ID_task;
        entry_address   = task();
        stack_size      = 100;
        priority        = 3;
};
```

2. ファイル先頭で必ずシステムディレクトリのなかの "itron.h","kernel.h" とカレントディレクトリ内の "kernel_id.h" をインクルードしてください。すなわちファイルの先頭で以下の3行を必ず記述してください。

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
```

3. タスク開始関数の戻り値はありません。したがって、void 型で宣言してください。

4. スタティック宣言をおこなった関数はタスクとして登録できません。

5. タスク開始関数の出口で ext_tsk()を記述する必要はありません。[44] タスク開始関数から呼び出した関数でタスクを終了する場合は、ext_tsk()を記述してください。

6. タスクの開始関数を無限ループで記述することも可能です。

[44] MR30 では、#pragma TASK 宣言を行うことで、自動的に ext_tsk()で終了します。関数の途中で return 文により戻る場合も同様に ext_tsk()で終了処理をおこないます。

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void task(VP_INT stacd)
{

        /* 処理 */

}
```

図 7.1 C 言語で記述したタスクの例

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void task(VP_INT stacd)
{
        for(;;){
                /* 処理 */

        }
}
```

図 7.2 C 言語で記述した無限ループタスクの例

7. タスクを指定する場合はコンフィギュレーションファイルのタスク定義の項目"name"に記述した文字列で指定してください 。[45]

```
wup_tsk(ID_main);
```

8. イベントフラグ、セマフォ、メールボックスを指定する場合は、コンフィギュレーションファイルで定義したそれぞれの名前の文字列で指定してください。

例えば、コンフィギュレーションファイルで以下のようにイベントフラグを定義した場合は、

```
flag[1]{
        name    = ID_abc;
};
```

このイベントフラグを指定するには以下のようにおこなってください。

```
set_flg(ID_abc,(FLGPTN)setptn);
```

9. 周期ハンドラ、アラームハンドラを指定する場合は、コンフィギュレーションファイルのアラームハンドラもしくは周期ハンドラ定義の項目"name"に記述した文字列で指定してください。

```
sta_cyc(ID_cyc);
```

[45] コンフィギュレータがタスクの ID 番号をタスクを指定するための文字列に変換するためのファイル”kernel_id.h”を生成します。すなわち、タスク定義の項目”name”に指定した文字列をそのタスクの ID 番号に変換するための#define 宣言を”kernel_id.h”で行います。周期ハンドラ,アラームハンドラも同様です。

10. タスクを ter_tsk() サービスコールなどで終了した後で sta_tsk() サービスコールなどで再起動した場合は、タスク自身は初期状態[46]から開始しますが 外部変数、スタティック変数はタスクの開始にともなっての初期化はされません。外部変数、スタティック変数の初期化はMR30が立ち上がる前に起動されるスタートアッププログラム (crt0mr.a30) でのみおこないます。

11. MR30 システム起動時に起動されるタスクは、コンフィギュレーションファイルで設定します。

12. 変数の記憶クラスについて

C言語の変数はMR30 から見て 表 7.1に示す扱いになります。

表 7.1 C 言語における変数の扱い

変数の記憶クラス	扱い
グローバル変数	すべてのタスクの共有変数
関数外のスタティック変数	同一ファイル内のタスクの共有変数
オート変数 レジスタ変数 関数内のスタティック変数	タスク固有の変数

[46] タスクの開始関数から初期優先度でなおかつ起床カウントがクリアされた状態で開始します。

7.1.2 カーネル管理(OS依存)割り込みハンドラの記述方法

C 言語を用いてカーネル管理(OS 依存)割り込みハンドラを記述する場合、以下の点に注意してください。

1. カーネル管理(OS 依存)割り込みハンドラは関数として記述します。
2. 割り込みハンドラ開始関数の戻り値および引き数は、必ず void 型で宣言してください。
3. ファイル先頭で必ずシステムディレクトリのなかの "itron.h","kernel.h" とカレントディレクトリ内の "kernel_id.h" をインクルードしてください。すなわちファイルの先頭で以下の3行を必ず記述してください。

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
```

4. 関数の最後に ret_int サービスコールは記述しないで下さい。 また、割り込みハンドラ関数から呼び出した関数のなかで割り込みハンドラを終了することはできません。
5. スタティック宣言をおこなった関数は割り込みハンドラとしては登録できません。

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void inthand(void)

{
        /* 処理 */

        iwup_tsk(ID_main);
}
```

図 7.3 カーネル管理(OS 依存)割り込みハンドラの例

7.1.3 カーネル管理外(OS独立)割り込みハンドラの記述方法

C 言語を用いてカーネル管理外(OS 独立)割り込みハンドラを記述する場合、以下の点に注意してください。

1. 割り込みハンドラ開始関数の戻り値および引き数は、必ず **void** 型で宣言してください。
2. カーネル管理外(OS 独立)割り込みハンドラからは、サービスコールは発行できません。

(注)サービスコールを発行した場合は不正動作をするので十分注意して下さい。

3. スタティック宣言をおこなった関数は割り込みハンドラとしては登録できません。
4. カーネル管理外(OS 独立)割り込みハンドラの中で多重割り込みを許可する場合は、必ず、カーネル管理外(OS独立)割り込みハンドラの割り込み優先レベルを、他のカーネル管理(OS依存)割り込みハンドラの割り込み優先レベルより高くしてください。[47]

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void inthand(void)
{
        /* 処理 */
}
```

図 7.4 カーネル管理外(OS 独立)割り込みハンドラの例

[47] カーネル管理外(OS 独立)割り込みハンドラの割り込みレベル優先レベルを、カーネル管理(OS 依存)割り込みハンドラの割り込み優先レベルより低くしたい場合は、カーネル管理外(OS 独立)割り込みハンドラをカーネル管理(OS 依存)割り込みハンドラの記述に変更してください。

7.1.4 周期ハンドラ、アラームハンドラの記述方法

C 言語を用いて周期ハンドラおよびアラームハンドラを記述する場合、以下の点に注意してください。

1. 周期ハンドラおよびアラームハンドラは関数として記述します。[48]
2. 関数の引数を VP_INT 型、戻り値は void 型で宣言してください。
3. ファイル先頭で必ずシステムディレクトリのなかの "itron.h","kernel.h" とカレントディレクトリ内の "kernel_id.h" をインクルードしてください。すなわちファイルの先頭で以下の3行を必ず記述してください。スタティック宣言をおこなった関数は周期ハンドラおよびアラームハンドラとしては登録できません。

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
```

4. 周期ハンドラおよびアラームハンドラはシステムクロックの割り込みハンドラからサブルーチン呼び出しにより起動されます。

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void cychand(VP_INT inf)
{
        /* 処理 */
}
```

図 7.5 C 言語で記述した周期ハンドラの例

[48] ハンドラと関数名との対応は、コンフィギュレーションファイルにより行います。

7.2 アセンブリ言語によるコーディング方法

本節では、アセンブリ言語を用いてアプリケーションを記述する方法について述べます。

7.2.1 タスクの記述方法

アセンブリ言語を用いてタスクを記述する場合、以下の項目に注意してください。

1. ファイルの先頭で必ず "mr30.inc"をインクルードしてください。
2. タスクの開始アドレスを示すシンボルは外部シンボル宣言[49]をおこなってください。
3. タスクは無限ループか ext_tsk サービスコールで終了してください。

```
        .INCLUDE mr30.inc       ----- (1)
        .GLB     task           ----- (2)

task:
                ; 処理
        jmp   task              ----- (3)
```

図 7.6 アセンブリ言語で記述した無限ループタスクの例

```
        .INCLUDE mr30.inc
        .GLB     task

task:
                ; 処理
        ext_tsk
```

図 7.7 アセンブリ言語で記述した ext_tsk で終了するタスクの例

4. タスク起動時のレジスタの初期値は、R0、PC、SB、FLG レジスタ以外は不定です。
5. タスクを指定する場合はコンフィギュレーションファイルのタスク定義の項目"name"に記述した文字列で指定してください

```
        wup_tsk  #ID_task
```

6. イベントフラグ、セマフォ、メールボックスを指定する場合は、コンフィギュレーションファイルで定義したそれぞれの名前の文字列で指定してください。

例えば、 コンフィギュレーションファイルで以下のようにセマフォを定義した場合、

```
        semaphore[1]{
                name            = ID_abc;
        };
```

このセマフォを指定するには以下のようにおこなってください。

```
        sig_sem  #ID_abc
```

7. 周期ハンドラ、アラームハンドラを指定する場合は、コンフィギュレーションファイルのアラームハンドラもしくは周期ハンドラ定義の項目"name"に記述した文字列で指定してください。

```
        sta_cyc  #ID_cyc
```

[49] .GLB 指示命令を使用してください。

8. MR30 システム起動時に起動されるタスクは、コンフィギュレーションファイルで設定します。

7.2.2 カーネル管理(OS依存)割り込みハンドラの記述方法

アセンブリ言語を用いてカーネル管理(OS 依存)割り込みハンドラを記述する場合、以下の項目に注意してください。

1. ファイルの先頭で必ず"mr30.inc"をインクルードしてください。
2. 割り込みハンドラの開始アドレスを示すシンボルは外部宣言 (グローバル宣言) をおこなってください。
3. ハンドラ内で使用するレジスタは、 ハンドラの入口でセーブし、使用後復帰して下さい。
4. ret_int サービスコールにて復帰してください。また、割り込みハンドラ関数から呼び出した関数のなかで割り込みハンドラを終了することはできません。

```
.INCLUDE mr30.inc            ------(1)
.GLB inth                    ------(2)

 inth:
;  使用レジスタ退避                  ------(3)
iwup_tsk  #ID_task1
:
処      理
:

;  使用レジスタ復帰                  ------(3)

ret_int                      ------(4)
```

図 7.8 カーネル管理(OS 依存)割り込みハンドラの例

7.2.3 カーネル管理外(OS独立)割り込みハンドラの記述方法

アセンブリ言語を用いてカーネル管理外(OS 独立)割り込みハンドラを記述する場合、以下の項目に注意してください。

1. 割り込みハンドラの開始アドレスを示すシンボルは外部宣言 (グローバル宣言) して下さい。
2. ハンドラ内で使用するレジスタは入り口でセーブし、使用後復帰して下さい。
3. REIT 命令で終了してください。
4. カーネル管理外(OS 独立)割り込みハンドラからは、サービスコールは発行できません。

(注)サービスコールを発行した場合は不正動作をするので十分注意して下さい。

5. カーネル管理外(OS 独立)割り込みハンドラ内で多重割り込みを許可する場合は、カーネル管理外(OS 独立)割り込みハンドラの割り込み優先レベルは、他のカーネル管理(OS 依存)割り込みハンドラの割り込み優先レベルより必ず高くしてください。

```
    .GLB    inthand         ----- (1)

inthand:
; 使用レジスタ退避              ----- (2)
; 割り込み処理
; 使用レジスタ復帰              ----- (2)
    REIT                    ----- (3)
```

図 7.9 カーネル管理外(OS 独立)割り込みハンドラの例

7.2.4 周期ハンドラ、アラームハンドラの記述方法

アセンブリ言語を用いて周期ハンドラおよびアラームハンドラを記述する場合、以下の点に注意してください。

1. ファイルの先頭で必ず"mr30.inc"をインクルードしてください。
2. ハンドラの開始アドレスを示すシンボルは外部宣言 (グローバル宣言) をおこなってください。
3. 周期ハンドラ、アラームハンドラは全て RTS 命令 (サブルーチンリターン命令) にて復帰してください。

```
    .INCLUDE        mr30.inc        ----- (1)
    .GLB            cychand         ----- (2)

cychand:
    :
; ハンドラ処理
    :

    rts                             ----- (3)
```

図 7.10 アセンブリ言語で記述したハンドラの例

7.3 MR30 スタートアッププログラムの修正方法

MR30 には、以下に示す 2 種類のスタートアッププログラムが用意されています。

- start.a30
  アセンブリ言語を使って、プログラムを作成した時に使用するスタートアッププログラムです。
- crt0mr.a30
  C 言語を使って、プログラムを作成した時に使用するスタートアッププログラムです。
  "start.a30"に C 言語の初期化ルーチンを追加したものです。

スタートアッププログラムは以下のようなことを行っています。

- リセット後のプロセッサの初期化
- C 言語の変数の初期化 (crt0mr.a30 のみ)
- システムタイマの設定
- MR30 のデータ領域の初期化

このスタートアッププログラムは、環境変数 "LIB30"の示すディレクトリからカレントディレクトリへコピーして下さい。なお、必要があれば以下の示す箇所を修正、あるいは追加して下さい。

- プロセッサモードレジスタの設定
  プロセッサモードレジスタに、システムに合わせたプロセッサモードを設定して下さい。 (crt0mr.a30 の 75 行目)
- ユーザで必要な初期化プログラムの追加
  ユーザに必要な初期化プログラムを追加する場合は、C 言語用スタートアッププログラム (crt0mr.a30)の 175 行目に追加して下さい。
- 標準入出力関数を使用する場合は crt0mr.a30 の 134～135 行目のコメントをはずしてください。

7.3.1 C言語用スタートアッププログラム (crt0mr.a30)

```
 1 ; ***************************************************************
 2 ;
 3 ;    MR30 start up program for C language
 4 ;    Copyright(C) 1996(1997-2011) Renesas Electronics Corporation
 5 ;    and Renesas Solutions Corp. All Rights Reserved.
 6 ;
 7 ; ***************************************************************
 8 ; $Id: crt0mr.a30 519 2006-04-24 13:36:30Z inui $
 9 ;
10      .list   OFF
11      .include        c_sec.inc
12      .include        mr30.inc
13      .include        sys_rom.inc
14      .include        sys_ram.inc
15      .list   ON
16
17 ;-----------------------------------------------------------------------
18 ; SBDATA area definition
19 ;-----------------------------------------------------------------------
20      .glb    __SB__
21      .SB     __SB__
22
23 ;=======================================================================
24 ; Initialize Macro declaration
25 ;-----------------------------------------------------------------------
26 N_BZERO      .macro  TOP_,SECT_
27      mov.b   #00H, R0L
28      mov.w   #(TOP_ & 0FFFFH), A1
29      mov.w   #sizeof SECT_, R3
30      sstr.b
31      .endm
32
33 N_BCOPY .macro       FROM_,TO_,SECT_
34      mov.w   #(FROM_ & 0FFFFH),A0
35      mov.b   #(FROM_>>16),R1H
36      mov.w   #TO_,A1
37      mov.w   #sizeof SECT_, R3
38      smovf.b
39      .endm
40
41 BZERO        .macro  TOP_,SECT_
42      push.w  #sizeof SECT_ >> 16
43      push.w  #sizeof SECT_ & 0ffffh
44      pusha   TOP_>>16
45      pusha   TOP_ & 0ffffh
46
47      .glb    _bzero
48      jsr.a   _bzero
49      .endm
50 ;
51
52 BCOPY        .macro  FROM_,TO_,SECT_
53      push.w  #sizeof SECT_ >> 16
54      push.w  #sizeof SECT_ & 0ffffh
55      pusha   TO_>>16
56      pusha   TO_ & 0ffffh
57      pusha   FROM_>>16
58      pusha   FROM_ & 0ffffh
59
60      .glb    _bcopy
61      jsr.a   _bcopy
62      .endm
63
64 ;=======================================================================
65 ; Interrupt section start
66 ;-----------------------------------------------------------------------
67      .glb    __SYS_INITIAL
68      .section        MR_KERNEL,CODE,ALIGN
69 __SYS_INITIAL:
70 ;-----------------------------------------------------------------------
71 ; after reset,this program will start
72 ;-----------------------------------------------------------------------
73      ldc     #(__Sys_Sp&0FFFFH),ISP  ; set initial ISP
74
```

```
 75      mov.b   #2H,0AH
 76      mov.b   #00,PMOD                ; Set Processor Mode Regsiter
 77      mov.b   #0H,0AH
 78
 79      ldc     #00H,FLG
 80      ldc     #(__Sys_Sp&0FFFFH),fb
 81      ldc     #__SB__,sb
 82
 83 ; +-------------------------------------------------------+
 84 ; |    ISSUE SYSTEM CALL DATA INITIALIZE                  |
 85 ; +-------------------------------------------------------+
 86      ; For PD30
 87      __INIT_ISSUE_SYSCALL
 88
 89 ; +-------------------------------------------------------+
 90 ; |    MR RAM DATA 0(zero) clear                          |
 91 ; +-------------------------------------------------------+
 92      N_BZERO MR_RAM_top,MR_RAM
 93
 94
 95 ;=====================================================================
 96 ; NEAR area initialize.
 97 ;---------------------------------------------------------------------
 98 ; bss zero clear
 99 ;---------------------------------------------------------------------
100      N_BZERO (TOPOF bss_SE),bss_SE
101      N_BZERO (TOPOF bss_SO),bss_SO
102
103      N_BZERO (TOPOF bss_NE),bss_NE
104      N_BZERO (TOPOF bss_NO),bss_NO
105
106 ;---------------------------------------------------------------------
107 ; initialize data section
108 ;---------------------------------------------------------------------
109      N_BCOPY (TOPOF data_SEI),data_SE_top,data_SE
110      N_BCOPY (TOPOF data_SOI),data_SO_top,data_SO
111      N_BCOPY (TOPOF data_NEI),data_NE_top,data_NE
112      N_BCOPY (TOPOF data_NOI),data_NO_top,data_NO
113
114 ;=====================================================================
115 ; FAR area initialize.
116 ;---------------------------------------------------------------------
117 ; bss zero clear
118 ;---------------------------------------------------------------------
119      BZERO   (TOPOF bss_FE),bss_FE
120      BZERO   (TOPOF bss_FO),bss_FO
121
122 ;---------------------------------------------------------------------
123 ; Copy edata_E(O) section from edata_EI(OI) section
124 ;---------------------------------------------------------------------
125      BCOPY   (TOPOF data_FEI),data_FE_top,data_FE
126      BCOPY   (TOPOF data_FOI),data_FO_top,data_FO
127
128      ldc     #(__Sys_Sp&0FFFFH),     sp
129      ldc     #(__Sys_Sp&0FFFFH),     fb
130
131 ;=====================================================================
132 ; Initialize standard I/O
133 ;---------------------------------------------------------------------
134 ;    .glb    __init
135 ;    jsr.a   __init
136
137 ;---------------------------------------------------------------------
138 ; Set System IPL
139 ; and
140 ; Set Interrupt Vector
141 ;---------------------------------------------------------------------
142      mov.b   #0,R0L
143      mov.b   #__SYS_IPL,R0H
144      ldc     R0,FLG                  ; set system IPL
145      ldc     #((__INT_VECTOR>>16)&0FFFFH),INTBH
146      ldc     #(__INT_VECTOR&0FFFFH),INTBL
147
148 .IF USE_TIMER
149 ; +-------------------------------------------------------+
150 ; |    System timer interrupt setting                     |
151 ; +-------------------------------------------------------+
```

```
152     tmroffset       .equ    -60h            ; Timer register offset for M16C/64
153     ;for M16C/64
154
155     mov.b   #stmr_mod_val,stmr_mod_reg+tmroffset    ;set timer mode for M16C/64
156     mov.b   #stmr_int_IPL,stmr_int_reg              ;set timer IPL
157     mov.w   #stmr_cnt,stmr_ctr_reg+tmroffset        ;set interval count for M16C/64
158     or.b    #stmr_bit+1,stmr_start+tmroffset        ;system timer start for M16C/64
159 .ENDIF
160
161 ; +-----------------------------------------------------+
162 ; |    System timer initialize                        |
163 ; +-----------------------------------------------------+
164 .IF     USE_SYSTEM_TIME
165       MOV.W   #__D_Sys_TIME_L, __Sys_time+4
166       MOV.W   #__D_Sys_TIME_M, __Sys_time+2
167       MOV.W   #__D_Sys_TIME_H, __Sys_time
168 .ENDIF
169
170 ; +-----------------------------------------------------+
171 ; |    User Initial Routine ( if there are )          |
172 ; +-----------------------------------------------------+
173 ;
174
175
176 ;    jmp     __MR_INIT       ; for Separate ROM
177
178 ; +-----------------------------------------------------+
179 ; |    Initialization of System Data Area             |
180 ; +-----------------------------------------------------+
181     .GLB    __init_sys,__init_tsk,__END_INIT
182     JSR.W   __init_sys
183     JSR.W   __init_tsk
184
185    .IF      __MR_TIMEOUT
186     .GLB    __init_tout
187     JSR.W   __init_tout
188    .ENDIF
189
190    .IF      __NUM_FLG
191     .GLB    __init_flg
192     JSR.W   __init_flg
193    .ENDIF
194
195    .IF      __NUM_SEM
196     .GLB    __init_sem
197     JSR.W   __init_sem
198    .ENDIF
199
200    .IF      __NUM_DTQ
201     .GLB    __init_dtq
202     JSR.W   __init_dtq
203    .ENDIF
204
205    .IF      __NUM_VDTQ
206     .GLB    __init_vdtq
207     JSR.W   __init_vdtq
208    .ENDIF
209
210    .IF      __NUM_MBX
211     .GLB    __init_mbx
212     JSR.W   __init_mbx
213    .ENDIF
214
215    .IF      ALARM_HANDLER
216     .GLB    __init_alh
217     JSR.W   __init_alh
218    .ENDIF
219
220    .IF      CYCLIC_HANDLER
221     .GLB    __init_cyh
222     JSR.W   __init_cyh
223    .ENDIF
224
225    .IF      __NUM_MPF
226     ; Fixed Memory Pool
227     .GLB    __init_mpf
228     JSR.W   __init_mpf
```

```
229     .ENDIF
230
231     .IF       __NUM_MPL
232      ; Variable Memory Pool
233      .GLB     __init_mpl
234      JSR.W    __init_mpl
235     .ENDIF
236
237
238      ; For PD30
239      __LAST_INITIAL
240
241 __END_INIT:
242 ; +------------------------------------------------------+
243 ; |    Start initial active task                         |
244 ; +------------------------------------------------------+
245      __START_TASK
246
247      .glb     __rdyq_search
248      jmp.W    __rdyq_search
249
250 ; +---------------------------------------------+
251 ; |           Define Dummy                    |
252 ; +---------------------------------------------+
253      .glb     __SYS_DMY_INH
254 __SYS_DMY_INH:
255      reit
256
257 .IF  CUSTOM_SYS_END
258 ; +---------------------------------------------+
259 ; | Syscall exit routine to customize
260 ; +---------------------------------------------+
261      .GLB     __sys_end
262 __sys_end:
263      ; Customize here.
264      REIT
265 .ENDIF
266
267 ; +---------------------------------------------+
268 ; |           exit() function                   |
269 ; +---------------------------------------------+
270      .glb     _exit,$exit
271 _exit:
272 $exit:
273      jmp      _exit
274
275 .if USE_TIMER
276 ; +---------------------------------------------+
277 ; |      System clock interrupt handler         |
278 ; +---------------------------------------------+
279      .SECTION         MR_KERNEL,CODE,ALIGN
280      .glb             __SYS_STMR_INH, __SYS_TIMEOUT
281      .glb             __DBG_MODE,__SYS_ISS
282 __SYS_STMR_INH:
283      ; process issue system call
284      ; For PD30
285      __ISSUE_SYSCALL
286
287
288
289 ; System timer interrupt handler
290      _STMR_hdr
291      ret_int
292 .endif
293
294      .end
```

図 7.11 M16C/63,64,65 用 C 言語スタートアッププログラム (crt0mr.a30)

1. セクション定義ファイルを組み込みます。 [図 7.11の 11 行目]

2. MR30 用インクルードファイルを組み込みます。 [図 7.11の 12 行目]

3. システム ROM領域定義ファイルを組み込みます。 [図 7.11の 13 行目]

4. システム RAM領域定義ファイルを組み込みます。 [図 7.11の 14 行目]

5. リセット直後に起動される初期化プログラム__SYS_INITIALです。 [図 7.11の 69 行目-249 行目]

- システムスタックポインタの設定 [図 7.11の 73 行目]
- プロセッサモードレジスタの設定 [図 7.11の 75 行目-77 行目]
- FLG、SB、FBレジスタの設定 [図 7.11の 79 行目-81 行目]
- C言語の初期設定をおこないます。 [図 7.11の 100 行目-126 行目]
- OS割り込み禁止レベルの設定 [図 7.11の 142 行目-144 行目]
- 割り込みベクタテーブルのアドレス設定 [図 7.11の 145 行目～146 行目]
- MR30 のシステムクロック割り込みの設定をおこないます。 [図 7.11の 152 行目-158 行目]
- 標準入出力関数の初期化[図 7.11の 134 行目-135 行目]
  標準入出力関数を使用する場合は、この行のコメントをはずしてください。
- MR30 のシステム時刻の初期設定をおこないます。 [図 7.11の 165 行目-167 行目]

6. 必要があればアプリケーション固有の初期設定をおこないます。 [図 7.11の 175 行目]

7. MR30 が使用するRAMデータの初期化をおこないます。 [図 7.11の 182 行目-235 行目]

8. スタートアップの終了を示すビットをセットします。 [図 7.11の 239 行目]

9. 初期起動タスクを起動します。 [図 7.11の 245 行目-248 行目]

10. システムクロックの割り込みハンドラです。 [図 7.11の 282 行目-291 行目]

7.4 メモリ配置方法

アプリケーションプログラムのデータのメモリ配置方法について説明します。MR30 で使用するセクションは、c_sec.inc または asm_sec.inc で定義しています。メモリ配置を設定するためには、High-performance Embedded Workshop 上で変更します。

- asm_sec.inc
  アセンブリ言語で、アプリケーション開発を行った場合に使用します。

- c_sec.inc
  C 言語で、アプリケーション開発を行った場合に使用します。c_sec.inc は、"asm_sec.inc"に C コンパイラ NC30 が生成するセクションを追加したものです。

ユーザシステムに合わせて、セクション配置、開始アドレスの設定を変更して下さい。

7.4.1 カーネルが使用するセクション

アセンブリ言語用サンプルスタートアッププログラム"start.a30"のセクション配置は、"asm_sec.inc"で定義しています。
C 言語用サンプルスタートアッププログラム"crt0mr.a30"のセクション配置は、"c_sec.inc"で定義しています。
以下に、MR30 が使用する各セクションについて説明します。

- MR_RAM_DBG セクション
  MR30 のデバッグ機能に必要となる RAM データが入ったセクションです。
  このセクションは、必ず、内蔵 RAM 領域に配置してください。

- MR_RAM セクション
  MR30 のシステム管理データで、アブソリュートアドレシングで参照する RAM データが入るセクションです。
  このセクションは、必ず 0～0FFFFH(near 領域)以内に配置して下さい。

- stack セクション
  各タスクのユーザスタック、およびシステムスタックのセクションです。
  このセクションは、必ず 0～0FFFFH(near 領域)以内に配置して下さい。

- MR_HEAP セクション
  可変長メモリプールが格納されるセクションです。このセクションは、
  必ず 0～0FFFFH(near 領域)以内に配置して下さい。

- MR_KERNEL セクション
  MR30 カーネルプログラムを格納するセクションです。

- MR_CIF セクション
  MR30 用 C 言語インタフェースライブラリを格納するセクションです。

- MR_ROM セクション
  MR30 カーネルが参照するタスクの開始番地などのデータを格納するセクションです。

- INTERRUPT_VECTOR セクション

- FIX_INTERRUPT_VECTOR セクション
  割り込みベクタを格納するセクションです。このセクションの開始番地は使用する M16C ファミリ機種により異なります。サンプルスタートアッププログラムの番地は M16C/60 シリーズ用のものです。他のグループを使用する場合は変更してください。

8. コンフィギュレータの使用方法

8.1 コンフィギュレーションファイルの作成方法

アプリケーションプログラムのコーディング、スタートアッププログラムの修正が終わると、そのアプリケーションプログラムを MR30 システムに登録する必要があります。これを行うのがコンフィギュレーションファイルです。

8.1.1 コンフィギュレーションファイル内の表現形式

この節ではコンフィギュレーションファイル内における定義データの表現形式について説明します。

1. コメント文

'//'から行の終わりまではコメント文とみなし、処理の対象になりません。

2. 文の終わり

';'で文を終ります。

3. 数値

数値は以下の形式で入力できます。

- 16 進数
  数値の先頭に'0x'か'0X'を付加します。または、数値の最後に'h'か'H'を付加します。後者の場合でかつ先頭が英文字 (A～F)で始まる場合は先頭に必ず'0'を付加してください。なおここで使用する数値表現で英文字(A～F)は大文字･小文字を識別しません。[50]
- 10 進数
  23 のように整数のみで表します。ただし'0'で始めることはできません。
- 8 進数
  数値の先頭に'0'を付加するか数値の最後に'O'もしくは'o'を付加します。
- 2 進数
  数値の最後に'B'または'b'を付加します。ただし'0'で始めることはできません。

表 8.1 数値表現例

[50] 数値表現内の'A'～'F','a'～'f'を除いて全ての文字は、大文字・小文字の区別を行います。

16 進数	0xf12 0Xf12 0a12h 0a12H 12h 12H
10 進数	32
8 進数	017 17o 170
2 進数	101110b 101010B

また数値内に演算子を記述できます。使用できる演算子を 表 8.2に示します。

表 8.2 演算子

演算子	優先度	演算方向
()	高	左から右
- (単項マイナス)		右から左
* / %		左から右
+ - (二項マイナス)	低	左から右

数値の例を以下に示します。

- 123
- 123 + 0x23
- (23/4 + 3) * 2
- 100B + 0aH

4. シンボル

シンボルは数字、英大文字、英小文字、'_'(アンダースコア)、'?'より構成される数字以外の文字で始まる文字列で表されます。
シンボルの例を以下に示します。

- _TASK1
- IDLE3

5. 関数名

関数名は数字、英大文字、英小文字、'_' (アンダースコア)、'$'(ドル記号)より構成される数字以外の文字で始まり、'()'で終わる文字列で表されます。
C 言語で記述した関数名の例を以下に示します。

- main()
- func()

アセンブリ言語で記述した場合はモジュールの先頭ラベルを関数名とします。

6. 周波数

周波数は数字と'.'(ピリオド) から構成され'MHz'で終わる文字列で表されます。小数点以下は 6 桁が有効です。なお周波数は 10 進数のみで記述可能です。
周波数の例を以下に示します。

- 16MHz
- 8.1234MHz

なお、周波数は'.'で始まってはいけません。

7. 時間

時間は数字と'. ' (ピリオド) から構成され'ms'で終わる文字列で表されます。有効桁数は'ms'の場合小数点以下 3 桁です。なお時間は 10 進数のみで記述可能です。
時間の例を以下に示します。

- 10ms
- 10.5ms

なお時間は'.'(ピリオド)で始まってはいけません。

8.1.2 コンフィギュレーションファイルの定義項目

コンフィギュレーションファイルでは以下の項目[51]の定義をおこないます。

- システム定義
- システムクロック定義
- 最大項目定義
- タスク定義
- イベントフラグ定義
- セマフォ定義
- メールボックス定義
- データキュー定義
- long データキュー定義
- 固定長メモリプール定義
- 可変長メモリプール定義
- 周期ハンドラ定義
- アラームハンドラ定義
- 割り込みベクタ定義

[51] タスク定義以外の項目は、省略することができます。省略した場合にはデフォルトコンフィギュレーションファイルの定義が参照されます。

【システム定義】

<< 形式 >>

```
// System Definition
system{
    stack_size      = システムスタックサイズ ;
    priority        = 優先度の最大値 ;
    system_IPL      = カーネルマスクレベル(OS割り込み禁止レベル) ;
    timeout         = タイムアウト ;
    task_pause      = タスクポーズ ;
    tic_deno        = タイムティック分母 ;
    tic_nume        = タイムティック分子 ;
    message_pri     = 最大メッセージ優先度値;
};
```

<< 内容 >>

1. システムスタックサイズ (バイト)

   【 定義形式 】数値

   【 定義範囲 】4～ 0xFFFF

   【 デフォルト値 】400H

   サービスコール処理および割り込み処理で使用するスタックサイズの合計を定義します。

2. 優先度の最大値 (最低優先度の値)

   【 定義形式 】数値

   【 定義範囲 】1 ～ 255

   【 デフォルト値 】63

   MR30 のアプリケーションプログラムの使用する優先度の最大値を定義します。すなわち使用している優先度の最も大きい値を設定してください。[52]

3. カーネルマスクレベル(OS 割り込み禁止レベル)

   【 定義形式 】数値

   【 定義範囲 】1 ～ 7

   【 デフォルト値 】7

   サービスコール内での IPL の値、すなわちカーネルマスクレベル(OS 割り込み禁止レベル)を設定します。

4. タイムアウト

   【 定義形式 】シンボル

   【 定義範囲 】YES or NO

【 デフォルト値 】NO

tslp_tsk, twai_flg, twai_sem, trcv_mbx, tsnd_dtq, trcv_dtq, tget_mpf, vtsnd_dtq, vtrcv_dtq を使用している場合は、YES を設定し、使用していない場合は、NO を設定してください。

5. タスクポーズ

【 定義形式 】シンボル

【 定義範囲 】YES or NO

【 デフォルト値 】NO

デバッガの OS デバッグ機能であるタスクポーズ機能をご使用になる場合は、YES を設定し、使用していない場合は、NO を設定してください。

6. タイムティック分母

【 定義形式 】数値

【 定義範囲 】1 固定

【 デフォルト値 】1

タイムティックの分母を設定します。

7. タイムティック分子

【 定義形式 】数値

【 定義範囲 】1～65535

【 デフォルト値 】1

タイムティックの分子を設定します。タイムティック分母、分子の設定によってシステムクロックの割り込み間隔が決定されます。間隔は、(タイムティック分子/タイムティック分母)ms となります。すなわち、タイムティック分子 ms となります。

8. 最大メッセージ優先度値

【 定義形式 】数値

【 定義範囲 】1 ～ 255

【 デフォルト値 】なし

メッセージ優先度の最大値を定義します。

[52] MR30 の優先度は、値が大きいほど優先度は低くなります

【システムクロック定義】

<< 形式 >>

```
// System Clock Definition
clock{
    timer_clock     = タイマに供給されるクロック;
    timer           = システムクロックに使用するタイマ;
    IPL             = システムクロック割り込み優先レベル;
    current_reg_map = システムクロックアドレス補正有無;

};
```

<< 内容 >>

1. タイマに供給されるクロック(MHz)

【 定義形式 】周波数

【 定義範囲 】なし

【 デフォルト値 】20MHz

タイマに供給されるクロック(f1 の値)の周波数を MHz 単位で定義します。

2. システムクロックに使用するタイマ

【 定義形式 】シンボル

【 定義範囲 】

- M16C/60 シリーズ
  A0～A4, B0～B5, OTHER, NOTIMER
- M16C/30 シリーズ
  A0～A2, B1～B2, OTHER, NOTIMER
- M16C/20 シリーズ
  A0～A7, B0～B5, X0～X2, OTHER, NOTIMER
- M16C/10 シリーズ
  OTHER, NOTIMER
- R8Cファミリ
  RA,RB,OTHER,NOTIMER[53]

【 デフォルト値 】NOTIMER

システムクロックに使用するハードウェアタイマを定義します。
上記のシリーズごとのタイマの設定範囲はコンフィグレータではチェックしていませんので、マイコンにないタイマを指定しないように注意する必要があります。
M16C/10 シリーズでタイマを指定する場合は OTHER を指定して、使用するタイマの設定をスタートアップで行ってください。
システムクロックを使用しない場合は、"NOTIMER"を定義します。

[53] RA,RB を指定する場合は、current_reg_map=NO;でなければいけません。

3. システムクロック割り込み優先レベル

【 定義形式 】数値

【 定義範囲 】1 ～ (システム定義のカーネルマスクレベル(OS 割り込み禁止レベル))

【 デフォルト値 】4

システムクロック用タイマ割り込みの優先レベルを定義します。1 ～ カーネルマスクレベル(OS 割り込み禁止レベル)までの値を設定して下さい。
システムクロックの割り込みハンドラ処理中は、ここで定義した割り込みレベルより低いレベルの割り込みは受け付けられません。

4. システムクロックアドレス補正有無

【 定義形式 】シンボル

【 定義範囲 】YES,NO

【 デフォルト値 】NO

システムクロックに指定したタイマの SFR アドレスが M16C/64 と同じ場合に YES そうでない場合に NO を指定します。具体的には、M16C/63,64,64A,64C,65,65C,6B,6C グループ、M16C/50 シリーズを使用している場合は YES を指定します。そうでない場合は、NO を指定します。

【最大項目数定義】

この定義は、別ROM化[54]を行う場合のみ設定する項目です。
ここでの設定は、複数のアプリケーションの中で、各定義の最大数を定義します。

<< 形式 >>

```
// Max Definition
maxdefine{
    max_task        = 最大タスク定義数;
    max_flag        = 最大イベントフラグ定義数;
    max_dtq         = 最大データキュー定義数;
    max_mbx         = 最大メールボックス定義数;
    max_sem         = 最大セマフォ定義数;
    max_mpf         = 最大固定長メモリプール定義数;
    max_mpl         = 最大可変長メモリプール定義数;
    max_cyh         = 最大周期ハンドラ定義数;
    max_alh         = 最大アラームハンドラ定義数;
    max_vdtq        = 最大longデータキュー定義数;
};
```

<< 内容 >>

1. 最大タスク定義数

【 定義形式 】数値

【 定義範囲 】1 ～ 255

【 デフォルト値 】なし

タスク定義の最大数を定義します。

2. 最大イベントフラグ定義数

【 定義形式 】数値

【 定義範囲 】1 ～ 255

【 デフォルト値 】なし

イベントフラグ定義の最大数を定義します。

3. 最大データキュー定義数

【 定義形式 】数値

【 定義範囲 】1 ～ 255

【 デフォルト値 】なし

データキュー定義の最大数を定義します。

4. 最大メールボックス定義数

【 定義形式 】数値

【 定義範囲 】1 ～ 255

【 デフォルト値 】なし

メールボックス定義の最大数を定義します。

5. 最大セマフォ定義数

【 定義形式 】数値

【 定義範囲 】1 ～ 255

【 デフォルト値 】なし

セマフォ定義の最大数を定義します。

6. 最大固定長メモリプール定義数

【 定義形式 】数値

【 定義範囲 】1 ～ 255

【 デフォルト値 】なし

固定長メモリプール定義の最大数を定義します。

[54] 詳細は、13 別 ROM 化を参照ください。

7. 最大可変長メモリプール定義数

【 定義形式 】数値

【 定義範囲 】1 ～ 255

【 デフォルト値 】なし

可変長メモリプール定義の最大数を定義します。

8. 最大周期ハンドラ定義数

【 定義形式 】数値

【 定義範囲 】1 ～ 255

【 デフォルト値 】なし

周期ハンドラ定義の最大数を定義します。

9. 最大アラームハンドラ定義数

【 定義形式 】数値

【 定義範囲 】1 ～ 255

【 デフォルト値 】なし

アラームハンドラ定義の最大数を定義します。

10. 最大 long データキュー定義数

【 定義形式 】数値

【 定義範囲 】1 ～ 255

【 デフォルト値 】なし

long データキュー定義の最大数を定義します。

【タスク定義】

<< 形式 >>

```
 // Tasks Definition
task[ID番号]{
    name                    = ID名称;
    entry_address           = タスクの開始アドレス;
    stack_size              = タスクのユーザスタックサイズ;
    priority                = タスクの初期優先度;
    context                 = 使用するレジスタ;
    stack_section           = スタックを配置するセクション名;
    initial_start           = TA_ACT属性(初期起動状態);
    exinf                   = 拡張情報;
};
            :
            :
```

ID 番号は 1～255 の範囲でなければなりません。ID 番号は省略可能です。
省略した場合は番号を小さいほうから順に自動的に割り当てます。

<< 内容 >>

タスク ID 番号ごとに以下の定義をおこないます。

1. タスク ID 名称

   【 定義形式 】シンボル

   【 定義範囲 】なし

   【 デフォルト値 】なし

   タスクの ID 名称を定義します。なお、ここで定義した関数名は、以下のように kernel_id.h ファイルに出力されます。

   ```
   #define タスクID名称 タスクID
   ```

2. タスク開始アドレス

   【 定義形式 】シンボル、または、関数名

   【 定義範囲 】なし

   【 デフォルト値 】なし

   タスクの入り口アドレスを定義します。C 言語で記述したときはその関数名の最後に ()をつけるか、先頭に _をつけます。
   なお、ここで定義した関数名は、kernel_id.h ファイルに以下の宣言文が出力されます。

   ```
   #pragma TASK /V4 関数名
   ```

3. ユーザスタックサイズ (バイト)

   【 定義形式 】数値

【 定義範囲 】6 以上

【 デフォルト値 】256

タスクごとのユーザスタックサイズを定義します。ユーザスタックとは、各々のタスクが使用するスタック領域です。MR30 ではユーザ用のスタック領域をタスクごとに割り当てる必要があり最低で 6 バイトが必要です。

4. タスクの初期優先度

【 定義形式 】数値

【 定義範囲 】1 ～ (システム定義の優先度の最大値)

【 デフォルト値 】1

タスクの起動時の優先度を定義します。
MR30 の優先度は、値が小さいほど、優先度としては高くなります。

5. 使用するレジスタ

【 定義形式 】シンボル [,シンボル,.....]

【 定義範囲 】R0,R1,R2,R3,A0,A1,SB,FB から選択

【 デフォルト値 】全レジスタ

タスクで使用するレジスタを定義します。MR30 では、ここで定義されたレジスタをコンテキストとして扱います。 タスク起動時に、タスク起動コードが R1 レジスタに設定されますので、R1 レジスタは、必ず指定して下さい。ただし、タスクをアセンブリ言語で記述する場合のみ使用レジスタが選択可能です。C 言語で記述する場合、全レジスタを選択してください。
なお、レジスタを選択する場合、各タスクで使用しているサービスコールのパラメータを格納するレジスタについては、全て選択してください。
MR30 カーネル内では、レジスタバンク切り替えは行いません。本定義を省略した場合、全レジスタが選択されたものとします。

6. スタックを配置するセクション名

【 定義形式 】シンボル

【 定義範囲 】なし

【 デフォルト値 】stack

スタックを配置するセクション名を定義します。ここで定義したセクションは、必ず、セクションファイル (asm_sec.inc あるいは c_sec.inc)にて配置を行って下さい。
定義しない場合は、stack セクションに配置します。

7. TA_ACT 属性(初期起動状態)

【 定義形式 】シンボル

【 定義範囲 】ON or OFF

【 デフォルト値 】OFF

タスクの初期起動状態を定義します。ON を指定すると、システムの初期起動時に READY 状態になります。少なくとも一つのタスクについては ON を指定しなければなりません。

8. 拡張情報

【 定義形式 】数値

【 定義範囲 】0～0xFFFF

【 デフォルト値 】0

タスクの拡張情報を定義します。起動要求のキューイングによってタスクが再起動する際などに引数として渡されます。

【イベントフラグ定義】
この定義は、イベントフラグ機能を使用する場合に必ず設定する項目です。

<< 形式 >>

```
// Eventflag Definition
flag[ID番号]{
    name            = ID名称;
    wait_queue      = イベントフラグ待ちキュー選択;
    initial_pattern = イベントフラグ初期値;
    wait_multi      = 複数待ち属性;
    clear_attribute = クリア属性;
};
            :
            :
```

ID 番号は 1～255 の範囲でなければなりません。ID 番号は省略可能です。
省略した場合は番号を小さいほうから順に自動的に割り当てます。

<< 内容 >>

イベントフラグ ID 番号ごとに以下の定義をおこないます。

1. ID 名称

   【 定義形式 】シンボル

   【 定義範囲 】なし

   【 デフォルト値 】なし

   プログラム中でイベントフラグを指定する時の名前を定義します。

2. イベントフラグ待ちキュー選択

   【 定義形式 】シンボル

   【 定義範囲 】TA_TFIFO, TA_TPRI

   【 デフォルト値 】TA_TFIFO

   イベントフラグ待ちの方法を選択します。TA_TFIFO は、FIFO 順で待ちキューにつながれます。TA_TPRI は、タスクの優先度の高い順に待ちキューにつながれます。

3. イベントフラグ初期値

   【 定義形式 】数値

   【 定義範囲 】0～0xFFFF

   【 デフォルト値 】0

   イベントフラグの初期ビットパターンを指定します。

4. 複数待ち属性

【 定義形式 】シンボル

【 定義範囲 】TA_WMUL,TA_WSGL

【 デフォルト値 】TA_WSGL

イベントフラグ待ちキューに複数のタスクがつなぐことを許すかどうか指定します。TA_WMUL の場合、TA_WMUL 属性が付加され、複数タスクの待ちを許します。TA_WSGL の場合、TA_WSGL 属性が付加され、複数タスクの待ちを許しません。

5. クリア属性

【 定義形式 】シンボル

【 定義範囲 】YES,NO

【 デフォルト値 】NO

イベントフラグ属性として、TA_CLR 属性を付加するかどうかを指定します。YES の場合、TA_CLR 属性が付加されます。NO の場合、TA_CLR 属性は付加されません。

【セマフォ定義】

この定義は、セマフォ機能を使用する場合に必ず設定する項目です。

<< 形式 >>

```
// Semaphore Definition
semaphore[ID番号] {
    name                    = ID名称;
    wait_queue              = セマフォ待ちキューの選択;
    initial_count           = セマフォのカウンタ初期値;
    max_count               = セマフォのカウンタの最大値;
};
            :
            :
```

ID 番号は 1～255 の範囲でなければなりません。ID 番号は省略可能です。
省略した場合は番号を小さいほうから順に自動的に割り当てます。

<< 内容 >>

セマフォ ID 番号ごとに以下の定義をおこないます。

1. ID 名称

【 定義形式 】シンボル

【 定義範囲 】なし

【 デフォルト値 】なし

プログラム中でセマフォを指定する時の名前を定義します。

2. セマフォ待ちキューの選択

【 定義形式 】シンボル

【 定義範囲 】TA_TFIFO,TA_TPRI

【 デフォルト値 】TA_TFIFO

セマフォ待ちの方法を選択します。TA_TFIFO は、FIFO 順で待ちキューにつながれます。TA_TPRI は、タスクの優先度の高い順に待ちキューにつながれます。

3. セマフォカウンタ初期値

【 定義形式 】数値

【 定義範囲 】0 ～ 65535

【 デフォルト値 】1

セマフォカウンタの初期値を定義します。

4. セマフォカウンタ最大値

【 定義形式 】数値

【 定義範囲 】1 ～ 65535

【 デフォルト値 】1

セマフォカウンタの最大値を定義します。

【データキュー定義】
この定義は、データキュー機能を使用する場合に必ず設定する項目です。

<< 形式 >>

```
// Dataqueue Definition
dataqueue[ID番号]{
    name            = ID名称;
    buffer_size     = データキュー個数;
    wait_queue      = データキュー待ちキュー選択;
};
            :
            :
```

ID 番号は 1～255 の範囲でなければなりません。ID 番号は省略可能です。
省略した場合は番号を小さいほうから順に自動的に割り当てます。

<< 内容 >>
データキューID 番号ごとに以下の項目の定義をおこないます。

1. ID 名称

【 定義形式 】シンボル

【 定義範囲 】なし

【 デフォルト値 】なし

プログラム中でデータキューを指定する時の名前を定義します。

2. データ個数

【 定義形式 】数値

【 定義範囲 】0～0x3FFF

【 デフォルト値 】0

送信可能なデータの個数を指定します。指定するのはサイズではなく個数です。

3. データキュー待ちキューの選択

【 定義形式 】シンボル

【 定義範囲 】TA_TFIFO,TA_TPRI

【 デフォルト値 】TA_TFIFO

データキュー送信待ちの方法を選択します。TA_TFIFO は、FIFO 順で待ちキューにつながれます。
TA_TPRI は、タスクの優先度の高い順に待ちキューにつながれます。

【longデータキュー定義】
この定義は、short データキュー機能を使用する場合に必ず設定する項目です。

<< 形式 >>

```
// Vdataqueue Definition
vdataqueue[ID番号]{
    name            = ID名称;
    buffer_size     = データキュー個数;
    wait_queue      = データキュー待ちキュー選択;
};
            :
            :
```

ID 番号は 1～255 の範囲でなければなりません。ID 番号は省略可能です。
省略した場合は番号を小さいほうから順に自動的に割り当てます。

<< 内容 >>
long データキューID 番号ごとに以下の項目の定義をおこないます。

1. ID 名称

【 定義形式 】シンボル

【 定義範囲 】なし

【 デフォルト値 】なし

プログラム中で long データキューを指定する時の名前を定義します。

2. データ個数

【 定義形式 】数値

【 定義範囲 】0～0x1FFF

【 デフォルト値 】0

送信可能なデータの個数を指定します。指定するのはサイズではなく個数です。

3. long データキュー待ちキューの選択

【 定義形式 】シンボル

【 定義範囲 】TA_TFIFO,TA_TPRI

【 デフォルト値 】TA_TFIFO

long データキュー送信待ちの方法を選択します。TA_TFIFO は、FIFO 順で待ちキューにつながれます。TA_TPRI は、タスクの優先度の高い順に待ちキューにつながれます。

【メールボックス定義】
この定義は、メールボックス機能を使用する場合に必ず設定する項目です。

<< 形式 >>

```
// Mailbox Definition
mailbox[ID番号]{
    name            = ID名称;
    wait_queue      = メールボックス待ちキュー選択;
    message_queue   = メッセージキュー選択;
    max_pri         = メッセージ最大優先度;
};
            :
            :
```

ID 番号は 1～255 の範囲でなければなりません。ID 番号は省略可能です。
省略した場合は番号を小さいほうから順に自動的に割り当てます。

<< 内容 >>
メールボックス ID 番号ごとに以下の項目の定義をおこないます。

1. ID 名称

   【 定義形式 】シンボル

   【 定義範囲 】なし

   【 デフォルト値 】なし

   プログラム中でメールボックスを指定する時の名前を定義します。

2. メールボックス待ちキューの選択

   【 定義形式 】シンボル

   【 定義範囲 】TA_TFIFO,TA_TPRI

   【 デフォルト値 】TA_TFIFO

   メールボックス待ちの方法を選択します。TA_TFIFO は、FIFO 順で待ちキューにつながれます。TA_TPRI は、タスクの優先度の高い順に待ちキューにつながれます。

3. メッセージキュー選択

   【 定義形式 】シンボル

   【 定義範囲 】TA_MFIFO,TA_MPRI

   【 デフォルト値 】TA_MFIFO

   メールボックスのメッセージキュー選択の方法を選択します。TA_MFIFO は、FIFO 順でキューにつながれます。TA_MPRI は、メッセージの優先度の高い順に待ちキューにつながれます。

4. メッセージキュー最大優先度

【 定義形式 】数値

【 定義範囲 】1～「最大項目数定義」の「メッセージ優先度最大値」で指定した値。

【 デフォルト値 】1

メールボックスのメッセージの最大優先度を指定します。

【固定長メモリプール定義】
この定義は、固定長メモリプール機能を使用する場合に必ず設定する項目です。

<< 形式 >>

```
// Fixed Memorypool Definition
memorypool[ID番号]{
    name            = ID名称;
    section         = セクション名;
    num_block       = メモリプールのブロック数;
    siz_block       = メモリプールのブロックサイズ;
    wait_queue      = メモリプール待ちキュー選択;
};
```

ID 番号は、1～255 の範囲でなければなりません。ID 番号は、省略可能です。
省略した場合は番号を小さい方から順に自動的に割り当てます。

<< 内容 >>
メモリプール ID 番号ごとに以下の定義をおこないます。

1. ID 名称

【 定義形式 】シンボル

【 定義範囲 】なし

【 デフォルト値 】なし

プログラム中でメモリプールを指定する時の名前を指定します。

2. セクション名

【 定義形式 】シンボル

【 定義範囲 】なし

【 デフォルト値 】MR_HEAP

メモリプールを配置するセクションの名前を定義します。ここで定義したセクションは、必ず、セクションファイル (asm_sec.inc あるいは c_sec.inc)にて配置を行って下さい。
定義しない場合は、MR_HEAP セクションに配置します。

3. ブロック数

【 定義形式 】数値

【 定義範囲 】1～65535

【 デフォルト値 】1

メモリプールのブロック総数を定義します。

4. サイズ（バイト）

【 定義形式 】数値

【 定義範囲 】2～65535

【 デフォルト値 】256

メモリプールの 1 ブロック当たりのサイズを定義します。この定義によりメモリプールとして使用する RAM 容量は、 (ブロック数)×(サイズ)バイトです。

5. メモリプール待ちキューの選択

【 定義形式 】シンボル

【 定義範囲 】TA_TFIFO,TA_TPRI

【 デフォルト値 】TA_TFIFO

固定長メモリプール獲得待ちの方法を選択します。TA_TFIFO は、FIFO 順で待ちキューにつながれます。TA_TPRI は、タスクの優先度の高い順に待ちキューにつながれます。

【可変長メモリプール定義】
この定義は、可変長メモリプール機能を使用する場合に必ず設定する項目です。

<< 形式 >>

```
// Variable-Size Memorypool Definition
variable_memorypool[ID番号]{
    name            = ID名称;
    max_memsize     = 確保するメモリブロックサイズの最大値;
    mpl_section     = セクション名;
    heap_size       = メモリプールのサイズ;
};
```

ID 番号は、1～255 の範囲でなければなりません。ID 番号は、省略可能です。
省略した場合は番号を小さい方から順に自動的に割り当てます。

<< 内容 >>

1. ID 名称

   【 定義形式 】シンボル

   【 定義範囲 】なし

   【 デフォルト値 】なし

   プログラム中でメモリプールを指定する時の名前を指定します。

2. 確保するメモリブロックサイズの最大値 (バイト)

   【 定義形式 】数値

   【 定義範囲 】1 ～ 65520

   【 デフォルト値 】なし

   アプリケーションプログラム中で、確保しているメモリブロックサイズの最大値を指定します。

3. セクション名

   【 定義形式 】シンボル

   【 定義範囲 】なし

   【 デフォルト値 】MR_HEAP

   メモリプールを配置するセクションの名前を定義します。ここで定義したセクションは、必ず、セクションファイル (asm_sec.inc あるいは c_sec.inc)にて配置を行って下さい。
   定義しない場合は、MR_HEAP セクションに配置します。

4. メモリプールのサイズ (バイト)

   【 定義形式 】数値

   【 定義範囲 】16 ～ 0xFFFF

【 デフォルト値 】なし

メモリプールのサイズを指定します。
MR30 では、メモリを 4 種類の固定長ブロックサイズで管理しています。この 4 種類のブロックサイズ(下記 a,b,c,d)は、下記の計算式から算出されます。

a = (((max_memsize+(X-1)/(X×8))+1) ×X
b = a × 2
c = a × 4
d = a × 8
X: ブロック管理用データサイズ(1 ブロックあたり 8 バイト)

可変長メモリプールは、メモリブロックを管理する領域として 1 ブロックあたり 8 バイトの領域が必要となります。このため、max_memsize で指定したサイズのメモリブロックを獲得するためには、**max_memsize+8** の結果が収まる上記 a,b,c,d いずれかのブロックサイズ以上の値をメモリプールのサイズに指定しなければなりません。

【周期ハンドラ定義】
この定義は、周期ハンドラ機能を使用する場合に必ず設定する項目です。

<< 形式 >>

```
// Cyclic Handlar Definition
 cyclic_hand[ID番号]{
    name                    = ID名称;
    interval_counter        = 周期ハンドラの周期間隔;
    start                   = TA_STA属性;
    phsatr                  = TA_PHS属性;
    phs_counter             = 起動位相;
    entry_address           = 周期ハンドラの開始アドレス;
    exinf                   = 拡張情報;
 };
            :
            :
```

ID 番号は 1 ～ 255 の範囲でなければなりません。ID 番号は省略可能です。
省略した場合は番号を小さいほうから順に自動的に割り当てます。

<< 内容 >>
周期ハンドラ ID 番号ごとに以下の項目の定義をおこないます。

1. ID 名称

【 定義形式 】シンボル

【 定義範囲 】なし

【 デフォルト値 】なし

プログラム中で周期ハンドラを指定する時の名前を指定します。

2. 周期間隔

【 定義形式 】数値

【 定義範囲 】1 ～ 0x7FFFFFFF

【 デフォルト値 】なし

周期ハンドラの周期間隔を定義します。ここで定義する時間の単位は ms です。例えば、1 秒間隔で周期起動しようとすると、この値を 1000 に設定します。

3. TA_STA 属性

【 定義形式 】シンボル

【 定義範囲 】ON,OFF

【 デフォルト値 】OFF

周期ハンドラの TA_STA 属性を指定します。ON の場合は、TA_STA 属性が付加され、OFF の場合は、TA_STA 属性は付加されません。

4. TA_PHS 属性

【 定義形式 】シンボル

【 定義範囲 】ON,OFF

【 デフォルト値 】OFF

周期ハンドラの TA_PHS 属性を指定します。ON の場合は、TA_PHS 属性が付加され、OFF の場合は、TA_PHS 属性は付加されません。

5. 起動位相

【 定義形式 】数値

【 定義範囲 】0 ～ 0x7FFFFFFF

【 デフォルト値 】なし

周期ハンドラの起動位相を定義します。ここで定義する時間の単位は ms です。

6. 開始アドレス

【 定義形式 】シンボル、または、関数名

【 定義範囲 】なし

【 デフォルト値 】なし

周期ハンドラの開始アドレスを定義します。
なお、ここで定義した関数名は、kernel_id.h ファイルに以下の宣言文が出力されます。

```
#pragma CYCHANDLER 関数名
```

7. 拡張情報

【 定義形式 】数値

【 定義範囲 】0～0xFFFF

【 デフォルト値 】0

周期ハンドラの拡張情報を定義します。周期ハンドラを起動する際に引数として渡されます。

【アラームハンドラ定義】
この定義は、アラームハンドラ機能を使用する場合に必ず設定する項目です。

<< 形式 >>

```
// Alarm Handlar Definition
alarm_hand[ID番号]{
    name                    = ID名称;
    entry_address           = アラームハンドラの開始アドレス;
    exinf                   = 拡張情報;
};
            :
            :
```

ID 番号は 1～255 の範囲でなければなりません。ID 番号は省略可能です。
省略した場合は番号を小さいほうから順に自動的に割り当てます。

<< 内容 >>
アラームハンドラ ID 番号ごとに以下の項目の定義をおこないます。

1. ID 名称

【 定義形式 】シンボル

【 定義範囲 】なし

【 デフォルト値 】なし

プログラム中でアラームハンドラを指定する時の名前を指定します。

2. 開始アドレス

【 定義形式 】シンボル、または、関数名

【 定義範囲 】なし

アラームハンドラの開始アドレスを定義します。なお、ここで定義した関数名は、kernel_id.h ファイルに以下の宣言文が出力されます。

```
#pragma ALMHANDLER 関数名
```

3. 拡張情報

【 定義形式 】数値

【 定義範囲 】0～0xFFFF

【 デフォルト値 】0

アラームハンドラの拡張情報を定義します。アラームハンドラを起動する際に引数として渡されます。

【割り込みベクタ定義】
この定義は、割り込みハンドラを使用する場合に設定する項目です。

<< 形式 >>

```
    // Interrupt Vector Definition
    interrupt_vector[ベクタ番号]{
        os_int                  = カーネル管理(OS依存)割り込みハンドラ;
        entry_address           = 割り込みハンドラの開始アドレス;
pragma_switch  pragma_swicth    = PRAGMA拡張機能に渡すスイッチ;
    };
                :
                :
```

割り込みベクタ番号は 0～63、247～255 の範囲まで記述できます。
ただし、そのベクタ番号が有効か否かは使用しているマイクロコンピュータに依存します。
また、コンフィギュレータは、ここで指定した割り込みの割り込み制御レジスタ（IPL 等）や、割り込み要因等の初期設定のコードは生成しません。初期設定はスタートアップファイル中もしくは、開発されるアプリケーションにあわせて作成して頂く必要があります。

<< 内容 >>

1. カーネル管理(OS 依存)割り込みハンドラ

   【 定義形式 】シンボル

   【 定義範囲 】YES または NO

   【 デフォルト値 】なし

   ハンドラが カーネル管理(OS 依存)割り込みハンドラかどうかを定義します。カーネル管理(OS 依存)割り込みハンドラであれば YES を、カーネル管理外(OS 独立)割り込みハンドラであれば NO を定義して下さい。
   YES を定義した場合、kernel_id.h ファイルに以下の宣言文を出力します。

   ```
   #pragma INTHANDLER /V4 関数名
   ```

   また、NO を定義した場合、kernel_id.h ファイルに以下の宣言文を出力します。

   ```
   #pragma INTERRUPT /V4 関数名
   ```

2. 開始アドレス

   【 定義形式 】シンボルまたは関数名

   【 定義範囲 】なし

   【 デフォルト値 】__SYS_DMY_INH

   割り込みハンドラの入口アドレスを定義します。C 言語で記述した時はその関数名の最後に () をつけるか先頭に_をつけます。

3. PRAGMA 拡張機能に渡すスイッチ

【 定義形式 】シンボル

【 定義範囲 】E,B

【 デフォルト値 】なし

#pragma INTHANDLER や、#pragma INTERRUPT に渡すスイッチを指定します。”E”を指定した場合、”/E”スイッチが指定され、多重割り込みが許可されます。”B”を指定した場合は、”/B”スイッチが指定され、レジスタバンク1が指定されます。

複数のスイッチを同時に指定することも出来ます。ただし、カーネル管理(OS 依存)割り込みハンドラの場合は、”E”スイッチのみ指定することが出来ます。カーネル管理外(OS 独立)割り込みハンドラの場合は、”E,B”スイッチを指定することが出来ますが、”E”と”B”を同時に指定することは出来ません。

注意事項

1. レジスタバンク指定方法について

C 言語でレジスタバンク 1 のレジスタを使ったカーネル管理(OS 依存)割り込みハンドラの記述はできません。アセンブリ言語のみ記述することができます。 アセンブリ言語で記述する場合、割り込みハンドラの入口と出口を以下に示すように記述してください。

(ret_int サービスコールを発行する前に必ず B フラグをクリアしてください。)

例）
```
interrupt:
        fset    B
                :
        fclr    B
        ret_int
```

MR30 カーネル内では、レジスタバンク切り替えは行いません。

2. NMI 割り込み、監視タイマ割り込みは、カーネル管理(OS 依存)割り込みで使用しないでください。

以下に固定ベクタの割り込み要因とベクタ番号を以下に示します。可変ベクタについてはご使用になっているマイコンのハードウェアマニュアルを参照してください。

表 8.3 M16C/60 での固定ベクタ割り込み要因とベクタ番号との対応

割り込み要因	割り込みベクタ番号	セクション名
未定義命令	247	FIX_INTERRUPT_VECTOR
オーバーフロー	248	FIX_INTERRUPT_VECTOR
BRK 命令	249	FIX_INTERRUPT_VECTOR
アドレス一致	250	FIX_INTERRUPT_VECTOR
シングルステップ	251	FIX_INTERRUPT_VECTOR
監視タイマ	252	FIX_INTERRUPT_VECTOR
DBC(使用禁止)	253	FIX_INTERRUPT_VECTOR
NMI	254	FIX_INTERRUPT_VECTOR
リセット	255	FIX_INTERRUPT_VECTOR

8.1.3 コンフィギュレーションファイル例

```
 1 ////////////////////////////////////////////////////////////////////////////////
 2 //
 3 //    kernel.cfg : building file for MR30 Ver.4.00
 4 //
 5 //    Generated by M3T-MR30 GUI Configurator at 2005/02/28 19:01:20
 6 //
 7 ////////////////////////////////////////////////////////////////////////////////
 8
 9 // system definition
10 system{
11      stack_size      = 256;
12      system_IPL      = 4;
13      message_pri     = 64;
14      timeout  = NO;
15      task_pause      = NO;
16      tick_nume       = 10;
17      tick_deno       = 1;
18 };
19
20 // max definition
21 maxdefine{
22      max_task        = 3;
23      max_flag        = 4;
24      max_sem = 3;
25      max_dtq = 3;
26      max_mbx = 4;
27      max_mpf = 3;
28      max_mpl = 3;
29      max_cyh = 4;
30      max_alh = 2;
31 };
32
33 // system clock definition
34 clock{
35      timer_clock     = 20.000000MHz;
36      timer   = A0;
37      IPL     = 3;
38 };
39
40 task[]{
41      entry_address   = task1();
42      name    = ID_task1;
43      stack_size      = 256;
44      priority        = 1;
45      initial_start   = OFF;
46      exinf   = 0x0;
47 };
48 task[]{
49      entry_address   = task2();
50      name    = ID_task2;
51      stack_size      = 256;
52      priority        = 5;
53      initial_start   = ON;
54      exinf   = 0xFFFF;
55 };
56 task[3]{
57      entry_address   = task3();
58      name    = ID_task3;
59      stack_size      = 256;
60      priority        = 7;
61      initial_start   = OFF;
62      exinf   = 0x0;
63 };
64
65 flag[]{
66      name    = ID_flg1;
67      initial_pattern = 0x00000000;
68      wait_queue      = TA_TFIFO;
69      clear_attribute = NO;
70      wait_multi      = TA_WSGL;
71 };
72 flag[1]{
73      name    = ID_flg2;
74      initial_pattern = 0x00000001;
```

```
 75      wait_queue      = TA_TFIFO;
 76      clear_attribute = NO;
 77      wait_multi      = TA_WMUL;
 78 };
 79 flag[2]{
 80      name    = ID_flg3;
 81      initial_pattern = 0x0000ffff;
 82      wait_queue      = TA_TPRI;
 83      clear_attribute = YES;
 84      wait_multi      = TA_WMUL;
 85 };
 86 flag[]{
 87      name    = ID_flg4;
 88      initial_pattern = 0x00000008;
 89      wait_queue      = TA_TPRI;
 90      clear_attribute = YES;
 91      wait_multi      = TA_WSGL;
 92 };
 93
 94 semaphore[]{
 95      name    = ID_sem1;
 96      wait_queue      = TA_TFIFO;
 97      initial_count   = 0;
 98      max_count       = 10;
 99 };
100 semaphore[2]{
101      name    = ID_sem2;
102      wait_queue      = TA_TFIFO;
103      initial_count   = 5;
104      max_count       = 10;
105 };
106 semaphore[]{
107      name    = ID_sem3;
108      wait_queue      = TA_TPRI;
109      initial_count   = 255;
110      max_count       = 255;
111 };
112
113 dataqueue[]{
114      name    = ID_dtq1;
115      wait_queue      = TA_TFIFO;
116      buffer_size     = 10;
117 };
118 dataqueue[2]{
119      name    = ID_dtq2;
120      wait_queue      = TA_TPRI;
121      buffer_size     = 5;
122 };
123 dataqueue[3]{
124      name    = ID_dtq3;
125      wait_queue      = TA_TFIFO;
126      buffer_size     = 256;
127 };
128
129 mailbox[]{
130      name    = ID_mbx1;
131      wait_queue      = TA_TFIFO;
132      message_queue   = TA_MFIFO;
133      max_pri = 4;
134 };
135 mailbox[]{
136      name    = ID_mbx2;
137      wait_queue      = TA_TPRI;
138      message_queue   = TA_MPRI;
139      max_pri = 64;
140 };
141 mailbox[]{
142      name    = ID_mbx3;
143      wait_queue      = TA_TFIFO;
144      message_queue   = TA_MPRI;
145      max_pri = 5;
146 };
147 mailbox[4]{
148      name    = ID_mbx4;
149      wait_queue      = TA_TPRI;
150      message_queue   = TA_MFIFO;
151      max_pri = 6;
```

```
152 };
153
154 memorypool[]{
155      name    = ID_mpf1;
156      wait_queue      = TA_TFIFO;
157      section = MR_RAM;
158      siz_block       = 16;
159      num_block       = 5;
160 };
161 memorypool[2]{
162      name    = ID_mpf2;
163      wait_queue      = TA_TPRI;
164      section = MR_RAM;
165      siz_block       = 32;
166      num_block       = 4;
167 };
168 memorypool[3]{
169      name    = ID_mpf3;
170      wait_queue      = TA_TFIFO;
171      section = MPF3;
172      siz_block       = 64;
173      num_block       = 256;
174 };
175
176 variable_memorypool[]{
177      name    = ID_mpl1;
178      max_memsize     = 8;
179      heap_size       = 16;
180 };
181 variable_memorypool[]{
182      name    = ID_mpl2;
183      max_memsize     = 64;
184      heap_size       = 256;
185 };
186 variable_memorypool[3]{
187      name    = ID_mpl3;
188      max_memsize     = 256;
189      heap_size       = 1024;
190 };
191
192 cyclic_hand[]{
193      entry_address   = cyh1();
194      name    = ID_cyh1;
195      exinf   = 0x0;
196      start   = ON;
197      phsatr  = OFF;
198      interval_counter        = 0x1;
199      phs_counter     = 0x0;
200 };
201 cyclic_hand[]{
202      entry_address   = cyh2();
203      name    = ID_cyh2;
204      exinf   = 0x1234;
205      start   = OFF;
206      phsatr  = ON;
207      interval_counter        = 0x20;
208      phs_counter     = 0x10;
209 };
210 cyclic_hand[]{
211      entry_address   = cyh3;
212      name    = ID_cyh3;
213      exinf   = 0xFFFF;
214      start   = ON;
215      phsatr  = OFF;
216      interval_counter        = 0x20;
217      phs_counter     = 0x0;
218 };
219 cyclic_hand[4]{
220      entry_address   = cyh4();
221      name    = ID_cyh4;
222      exinf   = 0x0;
223      start   = ON;
224      phsatr  = ON;
225      interval_counter        = 0x100;
226      phs_counter     = 0x80;
227 };
228
```

```
229 alarm_hand[]{
230      entry_address   = alm1();
231      name    = ID_alm1;
232      exinf   = 0xFFFF;
233 };
234 alarm_hand[2]{
235      entry_address   = alm2;
236      name    = ID_alm2;
237      exinf   = 0x12345678;
238 };
239
240
241 //
242 // End of Configuration
243 //
```

8.2 コンフィギュレータの実行

8.2.1 コンフィギュレータ概要

コンフィギュレータはコンフィギュレーションファイルで定義した内容をアセンブリ言語のインクルードファイル等に変換するツールです。コンフィギュレータの動作概要を 図 8.1に示します。
HEW 上でビルドする際は、自動的にコンフィギュレータが起動し、アプリケーションプログラムがビルドされるようになっています。

1. コンフィギュレータの実行には以下の入力ファイルが必要です

- コンフィギュレーションファイル (XXXX.cfg)
  システムの初期設定項目を記述したファイルです。カレントディレクトリに作成します。
- デフォルトコンフィギュレーションファイル (default.cfg)
  コンフィギュレーションファイルで値の設定を省略した場合にこのファイルに書き込まれている値を設定します 環境変数 "LIB30"で示されるディレクトリ、もしくは、カレントディレクトリに置きます。両方のディレクトリに存在する場合は、カレントディレクトリのファイルが優先されます。
- インクルードテンプレートファイル (mr30.inc,sys_ram.inc)
  インクルードファイル mr30.inc,sys_ram.inc のテンプレートとなるファイルです。
  環境変数 "LIB30"で示されるディレクトリに存在します。
- MR30 バージョンファイル (version)
  MR30 のバージョンを記述したファイルです。環境変数 "LIB30" で示されるディレクトリに存在します。コンフィギュレータはこのファイルを読み込み、起動メッセージに MR30 のバージョン情報を出力します。

2. コンフィギュレータの実行によって以下のファイルが出力されます

コンフィギュレータが出力したファイルには、ユーザのデータ定義を行わないで下さい。データ定義を行った後で、コンフィギュレータを起動するとユーザの定義したデータは失われます。

- システムデータ定義ファイル (sys_rom.inc,sys_ram.inc)
  システムの設定を定義しているファイルです。
- インクルードファイル (mr30.inc)
  mr30.inc はアセンブリ言語用のインクルードファイルです。
- ID 番号定義ファイル (kernel_id.h)
  ID 番号を定義したファイルです。

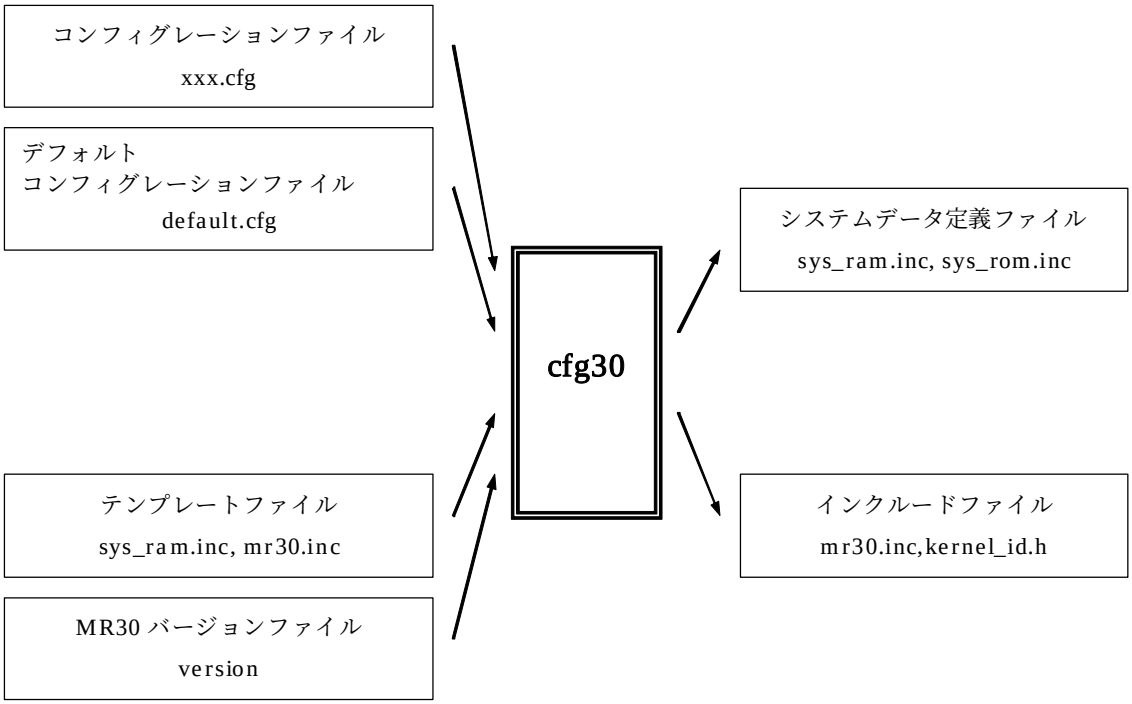


図 8.1 コンフィギュレータ動作概要

8.2.2 コンフィギュレータの環境設定

コンフィギュレータを実行するにあたって環境変数 "LIB30"が正しく設定されているかを確認してください。環境変数 "LIB30" で示すディレクトリ下には以下のファイルがないと正常に実行できません。

- デフォルトコンフィギュレーションファイル (default.cfg)
  カレントディレクトリにコピーして使用することもできます。その場合はカレントディレクトリのファイルを優先して使用します。
- システム RAM 領域定義データベースファイル (sys_ram.inc)
- mr30.inc のテンプレートファイル (mr30.inc)
- セクション定義ファイル (c_sec.inc または asm_sec.inc)
- スタートアップファイル (crt0mr.a30 または start.a30)
- MR30 バージョンファイル (version)

8.2.3 コンフィギュレータ起動方法

コンフィギュレータは以下の形式で起動します。

```
C> cfg30 [-vV] コンフィギュレーションファイル名
```

コンフィギュレーションファイル名は、 通常拡張子 (.cfg) かまたは拡張子 (.cfg) を除いたファイル名を指定します。

コマンドオプション

- -v オプション
  コマンドのオプションの説明と詳細なバージョンを表示します。
- -V オプション
  コマンドが生成するファイルの作成状況を表示します。

8.2.4 コンフィギュレータ実行上の注意

以下にコンフィギュレータ実行上の注意点を示します。

- スタートアッププログラム名、およびセクション定義ファイル名は変更しないでください。変更した場合、コンフィギュレータ実行時にエラーが発生します。

8.2.5 コンフィギュレータのエラーと対処方法

以下のメッセージが表示された場合はコンフィギュレータが正常に終了していませんのでコンフィギュレーションファイルを修正の上、再度コンフィギュレータを実行してください。

エラーメッセージ

1. cfg30 Error : syntax error near line xxx (test.cfg)

コンフィギュレーションファイルに文法エラーがあります。

2. cfg30 Error : not enough memory

メモリが足りません。

3. cfg30 Error : illegal option --> <x>

コンフィギュレータのコマンドオプションに誤りがあります。

4. cfg30 Error : illegal argument --> <xx>

コンフィギュレータの起動形式に誤りがあります。

5. cfg30 Error : can't write open <XXXX>

XXXX ファイルが作成できません。ディレクトリの属性やディスクの残り容量を確認してください。

6. cfg30 Error : can't open <XXXX>

XXXX ファイルにアクセスできません。XXXX ファイルの属性や、存在を確認してください。

7. cfg30 Error : can't open version file

環境変数"LIB30"の示すディレクトリの下に MR30 バージョンファイル"version"がありません。

8. cfg30 Error : can't open default configuration file

デフォルトコンフィギュレーションファイルがアクセスできません。環境変数 "LIB30"の示すディレクトリ、またはカレントディレクトリに"default.cfg"が必要です。

9. cfg30 Error : can't open configuration file <xxxxcfg>

コンフィギュレーションファイルがアクセスできません。コンフィグレータの起動形式を確認の上、正しいファイル名を指定してください。

10. cfg30 Error : illegal XXXX --> <xx> near line xxx (xxxx.cfg)

定義項目 XXXX の数値または ID 番号が間違っています。定義範囲を確認してください。

11. cfg30 Error : Unknown XXXX --> <xx> near line xxx (xxxx.cfg)

定義項目 XXXX のシンボル定義が間違っています。定義範囲を確認してください。

12. cfg30 Error : too big XXXX's ID number --> <xxx> (xxxx.cfg)

XXXX 定義の ID 番号に、定義したオブジェクトの総数を超える値が設定されています。ID 番号がオブジェクトの総数を超えることはありません。

13. cfg30 Error : too big task[x]'s priority --> <xxx> near line xxx (xxxx.cfg)

ID 番号 x のタスク定義項目の初期優先度が、システム定義項目の優先度値を越えています。

14. cfg30 Error : too big IPL --> <xxx> near line xxx (xxxx.cfg)

システムクロック定義項目のシステムクロック割り込み優先レベルがシステム定義項目の system IPL 値を越えています。

15. cfg30 Error : system timer's vector <x>conflict near line xxx

システムクロックの割り込みベクタに、 別の割り込みが定義されています。 割り込みベクタ番号を確認して下さい。

16. cfg30 Error : XXXX not defined (xxxx.cfg)

コンフィギュレーションファイルで XXXX の項目の定義が必要です。

17. cfg30 Error : system's default is not defined

デフォルトコンフィギュレーションファイルで定義が必要な項目です。

18. cfg30 Error : double definition <XXXX> near line xxx (xxxx.cfg)

項目 XXXX は既に定義されています。確認の上、重複定義を削除してください。

19. cfg30 Error : double definition XXXX[x] near line xxx (default.cfg)

20. cfg30 Error : double definition XXXX[x] near line xxx (xxxx.cfg)

項目 XXXX において ID 番号 x は既に登録されています。ID 番号を変更するか重複定義を削除してください。

21. cfg30 Error : you must define XXXX near line xxx (xxxx.cfg)

XXXX は、省略できない項目です。

22. cfg30 Error : you must define SYMBOL near line xxx (xxxxcfg)

省略できないシンボルです。

23. cfg30 Error : start-up-file (XXXX) not found

カレントディレクトリにスタートアップファイル XXXX が見つかりません。スタートアップファイル"start.a30"または"crt0mr.a30"が、カレントディレクトリに必要です。

24. cfg30 Error : bad start-up-file(XXXX)

カレントディレクトリに不要なスタートアップファイルがあります。

25. cfg30 Error : no source file

カレントディレクトリにソースファイルがありません。

26. cfg30 Error : zero divide error near line xxx (xxxx.cfg)

演算式で 0(ゼロ) 除算が発生しました。

27. cfg30 Error : task[X].stack_size must set XX or more near line xxx (xxxx.cfg)

タスクのスタックサイズを XX バイト以上のサイズをセットしてください。

28. cfg30 Error : "R0" must exist in task[x].context near line xxxx (xxxx.cfg)

タスクのコンテキスト選択項目では、必ず、R0 レジスタを選択してください。

29. cfg30 Error : can't define address match interrupt definition for Task Pause Function near line xxxx (xxxx.cfg)

タスクポーズ機能に必要な割り込みベクタに別の割り込みがコンフィギュレーションファイルに定義されています。

30. cfg30 Error : Set system.timer [system.timeout = YES] near line xxx (xxxx.cfg)

system.timeout = YES の設定にも関わらず、clock 定義において timer 項目が NOTIMER になっています。timer 項目でタイマを設定していください。

31. cfg30 Error : interrupt_vector[line xxx]:Can't specify B or F switch when os_int=YES.

“os_int = YES;”の場合、”B”または”F”スイッチは指定できません。

32. cfg30 Error : interrupt_vector[line 388]:Can't specify B and E switch at a time when os_int=NO.

“os_int = NO;”の場合、”B”,”F”のスイッチは同時に指定できません。

33. cfg30 Error: Initial Start Task not defined

コンフィギュレーションファイルで、初期起動タスクの定義がありません。

警告メッセージ

以下のメッセージは警告ですので、内容が理解できていれば無視してもかまいません。

1. **cfg30 Warning : system is not defined (xxxx.cfg)**
2. **cfg30 Warning : system.XXXX is not defined (xxxx.cfg)**

コンフィギュレーションファイルでシステム定義またはシステム定義項目 XXXX が省略されています。

3. **cfg30 Warning : task[x].XXXX is not defined near line xxx (xxxx.cfg)**

ID 番号 x のタスク定義項目 XXXX が省略されています。

4. **cfg30 Warning : Already definition XXXX near line xxx (xxxx.cfg)**

XXXX は既に定義されています。定義内容は無視されます。確認の上、重複定義を削除してください。

5. **cfg30 Warning : interrupt_vector[x]'s default is not defined (default.cfg)**

デフォルトコンフィギュレーションファイルでベクタ番号 x の割り込みベクタ定義が抜けています。

6. **cfg30 Warning : interrupt_vector[x]'s default is not defined near line xxx (test.cfg)**

コンフィギュレーションファイルのベクタ番号 x の割り込みベクタは、デフォルトコンフィギュレーションファイルに定義されていません。

7. **cfg30 Warning : system.stack_size is an uneven number near line xxx**
8. **cfg30 Warning : task[x].stack_size is an uneven number near line xxx**

スタックサイズは、偶数サイズを指定してください。

その他のメッセージ

以下のメッセージは makefile を生成する場合にのみ出力される警告メッセージです。コンフィギュレータは要因となった部分を読み飛ばして makefile を生成します。

1. **cfg30 Error : xxxx (line xxx): include format error.**

ファイル読み込みの書式が間違っています。正しい書式に書き直してください。

2. **cfg30 Warning : xxxx (line xxx): can't find <XXXX>**
3. **cfg30 Warning : xxxx (line xxx): can't find "XXXX"**

インクルードファイル XXXX が見つかりません。ファイル名および存在を確認して下さい。

4. **cfg30 Warning : over character number of including path-name**

インクルードファイルのパス名が 255 文字を超えています。

9. テーブル生成ユーティリティの使用方法

9.1 概要

mkmrtbl は、アプリケーションで使用しているサービスコール情報を収集して、サービスコールテーブルと割込みベクタテーブルを生成するコマンドラインツールです。

kernel.h からインクルードされる kernel_sysint.h では、サービスコール関数使用時に.assert 制御命令によって mrc ファイルにサービスコール情報を出力するように定義されています。mkmrtbl は、これらのサービスコール情報ファイルを入力として、システムで使用するサービスコールだけがリンクされるようにサービスコールテーブルを生成します。
また、mkmrtbl は cfg30 が出力したベクタテーブルテンプレートファイルと mrc ファイルを元に、割込みベクタテーブルを生成します。

9.2 環境設定

以下の環境変数の設定が必要です。

- LIB30
  “インストールディレクトリ¥lib30”

9.3 テーブル生成ユーティリティ起動方法

テーブル生成ユーティリティは、以下の形式で起動します。

```
C:¥> mkmrtbl  <ディレクトリ名またはファイル名>
```

通常は、アプリケーションのコンパイル時に生成される”mrc”ファイルが格納されたディレクトリを引数に指定します。複数のディレクトリ、ファイルを指定することができます。
なお、カレントディレクトリにある”mrc”ファイルは無条件に入力となります。

また、カレントディレクトリに、cfg30 が出力した vector.tpl が存在する必要があります。

9.4 注意事項

アプリケーションのコンパイルによって生成された mrc ファイルを漏れなく指定してください。漏れがある場合、サービスコールモジュールがリンクされない場合があります。

10. サンプルプログラム

10.1 サンプルプログラム概要

MR30 の応用例として、タスク間で交互に標準出力に文字列を出力するプログラムを示します。

表 10.1 サンプルプログラムの関数一覧

関数名	種類	ID 番号	優先度	機能
main()	タスク	1	1	task1、task2 を起動させます。
task1()	タスク	2	2	“task1 running”を出力します。
task2()	タスク	3	3	“task2 running”を出力します。
cyh1()	ハンドラ	1		task1()を起床します。

以下に、処理内容を説明します。

- main タスクは、task1、task2、cyh1 を起動し、自タスクを終了させます。
- task1 は、次の順で動作します。
  1. セマフォを獲得します。
  2. 起床待ちに移行します。
  3. "task1 running"を出力します。
  4. セマフォを解放します。
- task2 は、次の順で動作します。
  1. セマフォを獲得します。
  2. "task2 running"を出力します。
  3. セマフォを解放します。
- cyh1 は、100ms 毎に起動し、task1 を起床します。

10.2サンプルプログラム

```
 1 /************************************************************************
 2 *                      MR30/4  sample program
 3 *
 4 * Copyright (C) 1996(1997-2011) Renesas Electronics Corporation
 5 * and Renesas Solutions Corp. All rights reserved.*
 6 *
 7 *     $Id: demo.c 496 2006-04-05 06:28:56Z inui $
 8 ************************************************************************/
 9
10 #include <itron.h>
11 #include <kernel.h>
12 #include "kernel_id.h"
13 #include <stdio.h>
14
15
16 void main( VP_INT stacd )
17 {
18      sta_tsk(ID_task1,0);
19      sta_tsk(ID_task2,0);
20      sta_cyc(ID_cyh1);
21 }
22 void task1( VP_INT stacd )
23 {
24      while(1){
25              wai_sem(ID_sem1);
26              slp_tsk();
27              printf("task1 running¥n");
28              sig_sem(ID_sem1);
29      }
30 }
31
32 void task2( VP_INT stacd )
33 {
34      while(1){
35              wai_sem(ID_sem1);
36              printf("task2 running¥n");
37              sig_sem(ID_sem1);
38      }
39 }
40
41 void cyh1( VP_INT exinf )
42 {
43      iwup_tsk(ID_task1);
44 }
45
```

10.3 サンプルコンフィギュレーションファイル

```
 1 //************************************************************************
 2 //
 3 // Copyright (C) 1996(1997-2011) Renesas Electronics Corporation
 4 // and Renesas Solutions Corp. All rights reserved.
 5 //
 6 // MR30/4 System Configuration File.
 7 // "$Id: smp.cfg 496 2006-04-05 06:28:56Z inui $"
 8 //
 9 //************************************************************************
10
11 // System Definition
12 system{
13       stack_size     = 1024;
14     priority        = 10;
15       system_IPL     = 4;
16     task_pause      = NO;
17     timeout         = YES;
18     tic_nume        = 1;
19     tic_deno        = 1;
20     message_pri     = 255;
21 };
22 //System Clock Definition
23 clock{
24     timer_clock     = 20MHz;
25       timer          = A0;
26       IPL            = 4;
27 };
28 //Task Definition
29 //
30 task[]{
31     entry_address   = main();
32     name            = ID_main;
33       stack_size     = 100;
34     priority        = 1;
35     initial_start   = ON;
36     exinf           = 0;
37 };
38 task[]{
39     entry_address   = task1();
40     name            = ID_task1;
41       stack_size     = 500;
42     priority        = 2;
43     exinf           = 0;
44 };
45 task[]{
46     entry_address   = task2();
47     name            = ID_task2;
48     stack_size      = 500;
49     priority        = 3;
50     exinf           = 0;
51 };
52
53 semaphore[]{
54     name            = ID_sem1;
55     max_count       = 1;
56     initial_count   = 1;
57     wait_queue      = TA_TPRI;
58 };
59
60
61
62 cyclic_hand [1] {
63     name                    = ID_cyh1;
64     interval_counter        = 100;
65     start                   = OFF;
66     phsatr                  = OFF;
67     phs_counter             = 0;
68     entry_address           = cyh1();
69     exinf                   = 1;
70 };
71
```

11. スタックサイズの算出方法

11.1 スタックサイズの算出方法

MR30 のスタックには、システムスタックとユーザスタックの 2 種類があります。スタックサイズの計算方法は、ユーザスタックとシステムスタックで異なります。

●ユーザスタック

タスクに存在するスタックです。従って、MR30 を使ってアプリケーションプログラムを記述する場合には、タスクごとにスタック領域を確保する必要があります。

●システムスタック

MR30 内部もしくはハンドラ実行中に使用するスタックサイズです。MR30 では、サービスコールをタスクが発行するとユーザスタックからシステムスタックに切り替えます。システムスタックは、マイコンの割り込みスタックを使用します。

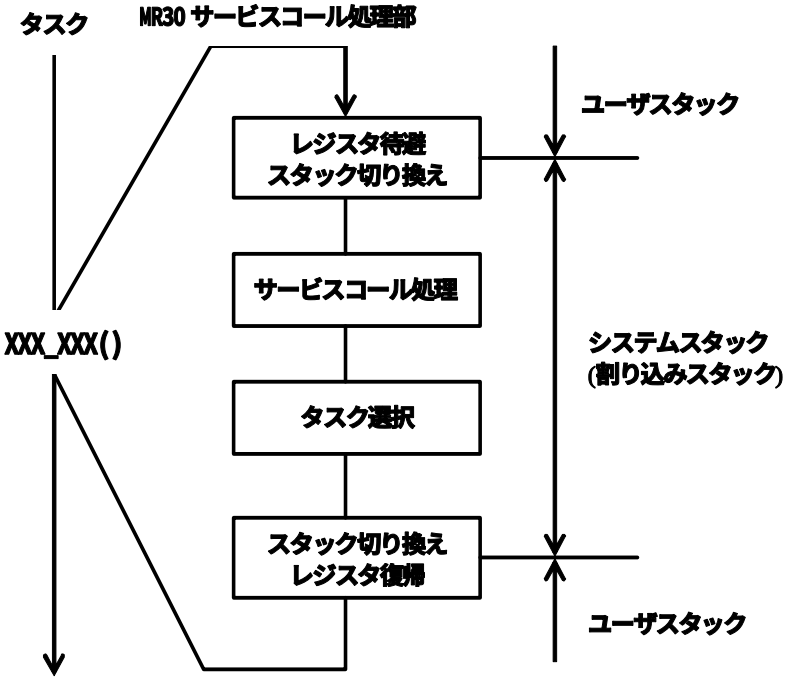


図 11.1 システムスタックとユーザスタック

システムスタックとユーザスタックの各セクションの配置は以下のようになります。ただし、以下の図は、コンフィギュレーション時にすべてのタスクのスタック領域を stack セクションに配置した場合です。

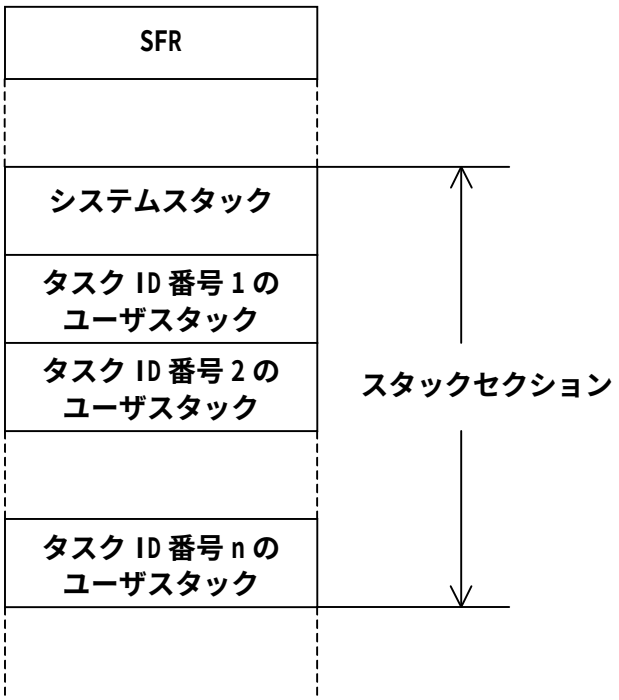


図 11.2 スタックの配置

11.1.1 ユーザスタックの算出方法

ユーザスタックは、タスクごとに算出する必要があります。以下にアプリケーションをC言語で記述した場合とアセンブリ言語で記述した場合のスタックの算出方法を以下に示します。

- C言語でアプリケーションを記述した場合

  NC30WA付属のスタック算出ユーティリティをご使用下さい。スタック算出ユーティリティは各タスクが使用するスタックサイズを表示します。その表示された各タスクのスタックサイズとコンテキスト格納領域20バイト[55]の合計が、タスクのスタックサイズとなります。スタック算出ユーティリティの詳細な使用方法については、スタック算出ユーティリティのマニュアルをご覧ください。

- アセンブリ言語でアプリケーションを記述した場合

  ユーザプログラムで使用する部分
  そのタスクがサブルーチン呼び出しで使用するスタック量、および、そのタスクでレジスタをスタックに保存する場合に使用する量などの合計。

  MR30で使用する部分
  サービスコールを発行することで消費するスタックサイズです。
  MR30では、タスクから発行可能なサービスコールのみを発行した場合は、6バイト確保してください。また、タスクまたはハンドラの両方から発行できるサービスコールを発行した場合は、表 11.3に記載されたスタックサイズを参考に確保して下さい。
  複数のサービスコールを発行している場合は、それらのサービスコールが消費するスタックサイズの最大値を確保して下さい。

よって、

ユーザスタックサイズ =
ユーザプログラムで使用する部分 + 使用するレジスタ分 + MR30で使用する部分

になります。(使用するレジスタ分は、レジスタ毎に2byteを加算する)
図 11.3にユーザスタックの算出例を示します。以下の例では、対象とするタスクが、R0,R1,A0レジスタを使用している場合です。

[55] C言語で記述した場合、このサイズは固定となります。

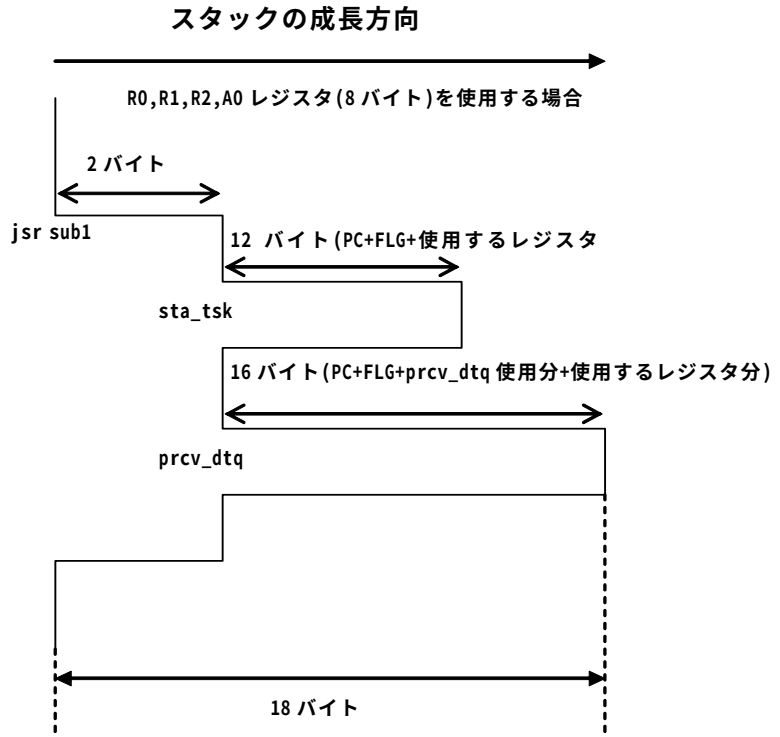


図 11.3:ユーザスタックサイズの算出例

11.1.2 システムスタックの算出方法

システムスタックを最も多く消費するのはサービスコール処理中[56]に割り込みが発生し、その上に多重割り込みが発生した場合です。すなわち、システムスタックの必要量 (最大サイズ)は以下の計算式で算出することができます。

$$\text{システムスタックの必要量} = \alpha + \sum \beta i (+\gamma)$$

- α

  使用するサービスコールの中で最大のシステムスタックサイズ[57]。
  例えば、sta_tsk、ext_tsk、slp_tsk、dly_tskを使用する場合、表 11.1で、それぞれのシステムスタックサイズを調べると、

サービスコール名	システムスタックサイズ
sta_tsk	2 バイト
ext_tsk	0 バイト
slp_tsk	2 バイト
dly_tsk	4 バイト

  となるのでこの場合、使用するサービスコールの中で最大のシステムスタックサイズは dly_tsk の場合で 4 バイトです。

- βi

  割り込みハンドラ[58]の使用するスタックサイズ。詳細は後述します。

- γ

  システムクロック割り込みハンドラの使用するスタックサイズ。詳細は後述します。

[56] ユーザスタックからシステムスタックに切り替えた後
[57] それぞれのサービスコールに使用するスタックサイズは、表 11.1から表 11.3を参照してください。
[58] システムクロック割り込みハンドラを含まないカーネル管理割り込みハンドラ(OS 依存割り込みハンドラ)

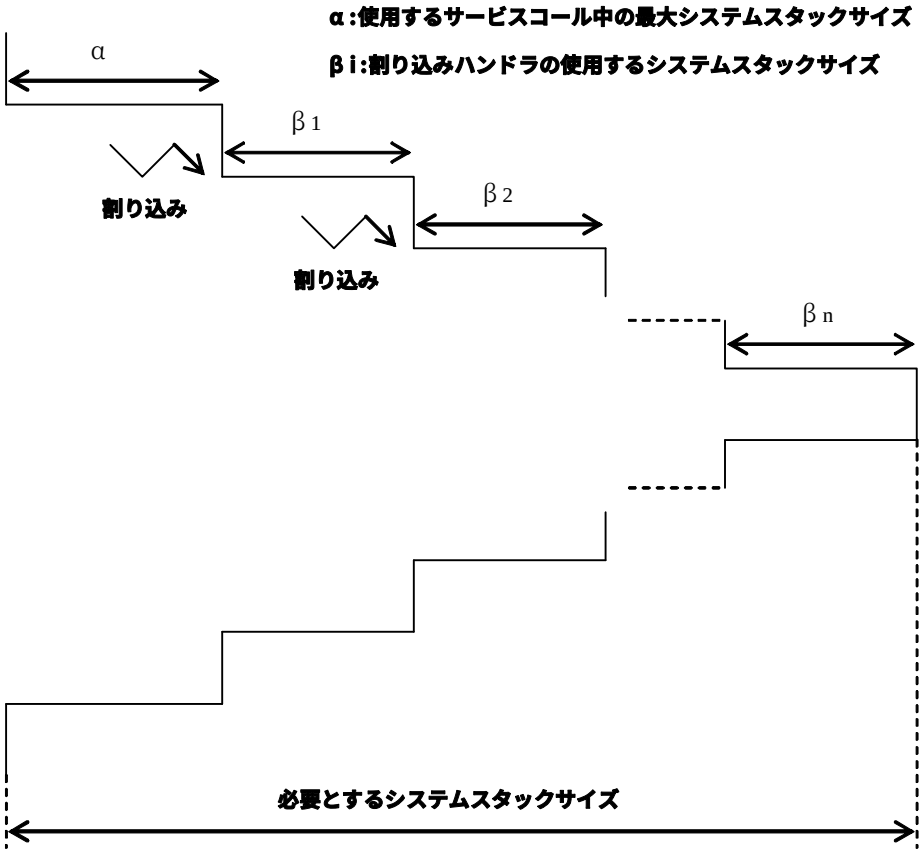


図 11.4:システムスタックサイズの算出方法

【割り込みハンドラの使用するスタックサイズβi】

サービスコール中に発生した割り込みハンドラの使用するスタックサイズは以下の計算式で算出できます。
割り込みハンドラの使用するスタックサイズβiを、以下に示します。

●C言語

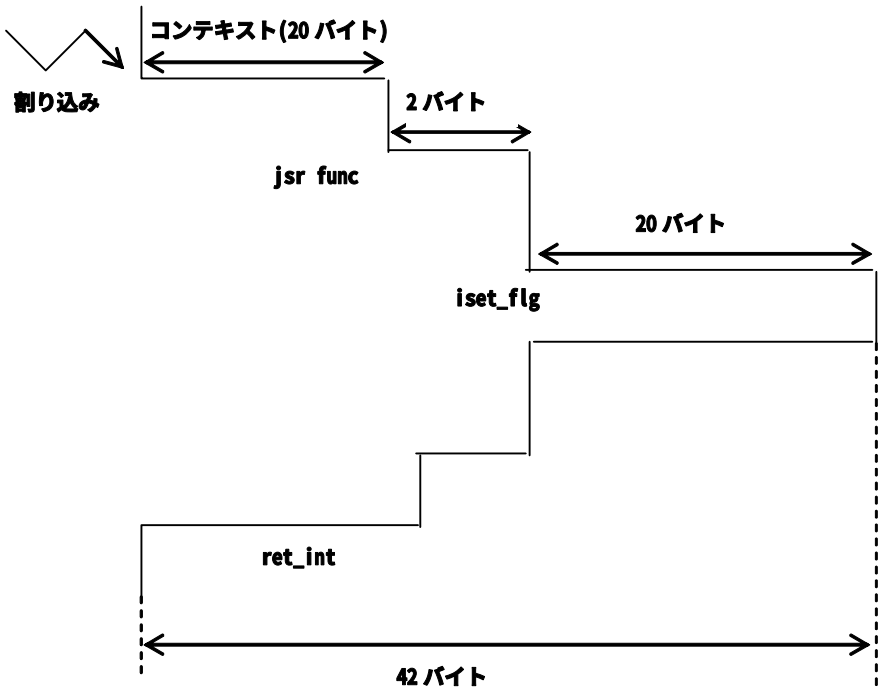
NC30WA付属のスタック算出ユーティリティをご使用下さい。スタック算出ユーティリティは各割り込みハンドラが使用するスタックサイズを表示します。その表示された値が各割り込みハンドラの使用するスタックサイズになります。スタック算出ユーティリティの詳細な使用方法については、スタック算出ユーティリティのマニュアルをご覧ください。

●アセンブリ言語

カーネル管理(OS依存)割り込みハンドラの使用するスタックサイズ =
使用するレジスタ分 + ユーザ使用量 + サービスコールの使用量

カーネル管理外(OS独立)割り込みハンドラの使用するスタックサイズ =
使用するレジスタ分 + ユーザ使用量

ユーザ使用量は、ユーザの記述する部分で使用するスタック使用量です。



コンテキスト： C 言語で記述した場合は 20 バイト
アセンブリ言語で記述した場合は、使用レジスタ分+4（PC+FLG)バイト

図 11.5:割り込みハンドラの使用するスタック量

【システムクロック割り込みハンドラが使用するシステムスタックサイズ γ】

システムタイマを使用しないときは、システムクロック割り込みハンドラが使用するシステムスタックを加算する必要はありません。

システムクロック割り込みハンドラが使用するシステムスタック量 γ は以下に示す 2 つの場合のうちの大きいサイズです。

24 + 周期起動ハンドラのスタック使用量の最も大きいサイズ
24+ アラームハンドラのスタック使用量の最も大きいサイズ

周期起動ハンドラおよびアラームハンドラが使用するスタックサイズの算出方法を以下に示します。

●C 言語

NC30WA 付属のスタック算出ユーティリティをご使用下さい。
スタック算出ユーティリティは各ハンドラが使用するスタックサイズを表示します。その表示された値が各ハンドラの使用するスタックサイズになります。
スタック算出ユーティリティの詳細な使用方法については、スタック算出ユーティリティのマニュアルをご覧ください。

●アセンブリ言語

周期起動ハンドラあるいはアラームハンドラの使用するスタックサイズ =
使用するレジスタ分 + ユーザ使用量 + サービスコールの使用量

周期起動、アラームハンドラのどちらも使用しない場合は、

γ = 14 バイト

になります。

割り込みハンドラとシステムクロック割り込みハンドラを併用して使用する場合は、双方の使用するスタックサイズを加算してください。

11.2 各サービスコールのスタック使用量

表 11.1は、タスクコンテキストから発行可能なサービスコールのスタック使用量(ユーザスタック及び、システムスタック)を示しています。

表 11.1 タスクコンテキストから発行するサービスコールのスタック使用量一覧(単位:バイト)

サービスコール	スタックサイズ		サービスコール	スタックサイズ	
	ユーザスタック	システムスタック		ユーザスタック	システムスタック
act_tsk	0	2	rcv_mbx	(5)	20
can_act	10	0	prcv_mbx	14(5)	0
sta_tsk	0	2	trcv_mbx	(5)	20
ext_tsk	0	0	ref_mbx	10	0
ter_tsk	0	4	get_mpf	(5)	24
chg_pri	0	22	pget_mpf	16(5)	0
get_pri	10(5)	0	tget_mpf	(5)	24
ref_tsk	22	0	rel_mpf	0	4
ref_tst	10	0	ref_mpf	10	0
slp_tsk	0	2	pget_mpl	(5)	32
tslp_tsk	0	4	rel_mpl	0	50
wup_tsk	0	4	ref_mpl	12	0
can_wup	10	0	set_tim	10	0
rel_wai	0	4	get_tim	10	0
sus_tsk	0	2	sta_cyc	10	0
rsm_tsk	0	2	stp_cyc	10	0
frsm_tsk	0	2	ref_cyc	10	0
dly_tsk	0	4	sta_alm	10	0
sig_sem	0	4	stp_alm	10	0
wai_sem	0	20	ref_alm	10	0
pol_sem	10	0	rot_rdq	0	0
twai_sem	0	22	get_tid	10(5)	0
ref_sem	10	0	loc_cpu	4	0
set_flg	0	8	unl_cpu	0	0
clr_flg	10	0	ref_ver	12	0
wai_flg	(5)	20	vsnd_dtq	0	20
pol_flg	10(5)	0	vpsnd_dtq	0	4
twai_flg	(7)	20	vtsnd_dtq	(5)	22
ref_flg	10	0	vfsnd_dtq	0	4
snd_dtq	0	20	vrcv_dtq	(7)	4
psnd_dtq	0	4	vprcv_dtq	(7)	4
tsnd_dtq	(5)	22	vtrcv_dtq	(7)	4
fsnd_dtq	0	4	vref_dtq	10	0
rcv_dtq	(5)	4	vrst_dtq	0	18
prcv_dtq	(5)	4	vrst_vdtq	0	18
trcv_dtq	(5)	4	vrst_mbx	10	0
ref_dtq	10	0	vrst_mpf	0	18
snd_mbx	0	18	vrst_mpl	60	0
dis_dsp	4	0	ena_dsp	0	0

()内: C 言語で使用時に必要となるスタック使用量。

表 11.2は、非タスクコンテキストから発行可能なサービスコールのスタック使用量(システムスタック)を示しています。

表 11.2 非タスクコンテキストから発行するサービスコールのスタック使用量一覧(単位:バイト)

サービスコール	スタックサイズ	サービスコール	スタックサイズ
iact_tsk	14	iprcv_mbx	14(5)
ican_act	10	iref_mbx	10
ista_tsk	14	ipget_mpf	16(5)
ichg_pri	32	irel_mpf	18
iget_pri	10(5)	iref_mpf	10
iref_tsk	22	iset_tim	10
iref_tst	10	iget_tim	10
iwup_tsk	16	ista_cyc	10
ican_wup	10	istp_cyc	10
irel_wai	14	iref_cyc	10
isus_tsk	12	ista_alm	10
irsm_tsk	12	istp_alm	10
ifrsm_tsk	12	iref_alm	10
isig_sem	16	irot_rdq	12
ipol_sem	10	iget_tid	10(5)
iref_sem	10	iloc_cpu	4
iset_flg	24	iunl_cpu	10
iclr_flg	10	ret_int	10
ipol_flg	10(5)	iref_ver	12
iref_flg	10	vipsnd_dtq	18
ipsnd_dtq	18	vifsnd_dtq	18
ifsnd_dtq	18	viprcv_dtq	20(7)
iprcv_dtq	18(5)	viref_dtq	10
iref_dtq	10	isnd_mbx	30
iref_mpl	12		

()内: C 言語で使用時に必要となるスタック使用量。

表 11.3は、タスクコンテキストあるいは非タスクコンテキストの両方から発行可能なサービスコールのスタック使用量を示しています。ここで示すスタックの使用量は、タスクからサービスコールを発行した場合は、ユーザスタックを使用し、非タスクコンテキストから発行した場合は、システムスタックを使用します。

表 11.3 両方から発行可能なサービスコールのスタック使用量一覧

サービスコール	スタックサイズ	サービスコール	スタックサイズ
sns_ctx	10	sns_loc	10
sns_dsp	10	sns_dpn	10

*: C 言語で使用時に必要となるスタック使用量。

12. 注意事項

12.1 INT命令の使用について

MR30 では、INT 命令の割り込み番号を表 5.2 に示すようにサービスコール発行のため予約しています。そのため、ユーザーアプリケーションでソフトウェア割り込みを使用する場合は、32 から 40 以外の割り込みを使用して下さい。

表 12.1　割り込み番号の割り当て

割り込み番号	使用するサービスコール
32	タスクからのみ発行可能なサービスコール
33	非タスクコンテキストからのみ発行可能なサービスコール タスクコンテキスト、非タスクコンテキストの両方から発行可能なサービスコール
34	ret_int サービスコール
35	dis_dsp サービスコール
36	loc_cpu, iloc_cpu サービスコール
37	ext_tsk サービスコール
38	tsnd_dtq,twai_flg,vtsnd_dtq サービスコール
39	拡張のため予約
40	拡張のため予約

12.2 レジスタバンクについて

MR30 では、タスク起動時のコンテキストは、レジスタバンク0を使用しています。カーネル処理中にレジスタバンク切り替えは行いません。プログラム誤動作の原因となりますので、以下の点にご注意ください。

- タスク内では、レジスタバンク切り替えは行わないで下さい。
- レジスタバンク切り替えを指定している割り込み同士が、多重に割り込まないようにして下さい。

12.3 ディスパッチ遅延について

MR30 では、ディスパッチ遅延に関するサービスコールが 4 つあります。

- dis_dsp
- ena_dsp
- loc_cpu, iloc_cpu
- unl_cpu, iunl_cpu

これらのサービスコールを使用し、一時的にディスパッチを遅延した場合のタスクの扱いについて以下に記述します。

1. ディスパッチ遅延中の実行タスクがプリエンプトされるべき状態になった場合

ディスパッチが禁止されている間は、実行中のタスクがプリエンプトされるべき状況となっても、新たに実行すべき状態となったタスクにはディスパッチされません。実行すべきタスクへのディスパッチは、ディスパッチ遅延状態が解除されるまで遅延されます。ディスパッチ遅延中は、システムは以下の状態となります。

- 実行中のタスクは RUNNING 状態であり、レディキューにつながれている。
- ディスパッチ禁止解除後に実行するタスクは、READY 状態であり、 (タスクがつながれている中で)最高優先度のレディキューにつながれている。

2. ディスパッチ遅延中に isus_tsk, irsm_tsk サービスコールが発行された場合

ディスパッチ禁止状態で起動された割り込みハンドラから、実行中のタスク (dis_dsp を発行したタスク) に対して isus_tsk を発行し SUSPENDED 状態へ移行させようとした場合、タスク状態の遷移はディスパッチ禁止状態が解除されるまで遅延されます。ディスパッチ遅延中は、システムは以下の状態となります。

- 実行中のタスクの状態の扱いは、OS 内部では、ディスパッチ遅延解除後の状態として扱います。そのため、実行中のタスクに対して発行された isus_tsk では、実行中のタスクをレディキューからはずし、SUSPENDED 状態に移行します。エラーコードは E_OK を返します。この後、実行中のタスクに対して irsm_tsk が発行されると、実行中のタスクをレディキューにつなぎ、エラーコードは E_OK を返します。ただし、ディスパッチ遅延が解除されるまでタスクの切り替えは起こりません。
- ディスパッチ禁止解除後に実行するタスクは、レディキューにつながれています。

3. ディスパッチ遅延中に rot_rdq, irot_rdq サービスコールが発行された場合

ディスパッチ遅延中に、rot_rdq(TPRI_RUN=0)を発行した場合、自タスクの持つ優先度のレディキューを回転させます。また、irot_rdq(TPRI_RUN=0)を発行した場合、実行中のタスクが持つ優先度のレディキューが回転します。この場合、実行中のタスクは該当レディキューにはつながれていない場合があります。 (ディスパッチ遅延中に、実行タスクに対し isus_tsk が発行された場合など。)

4. 注意事項

- dis_dsp, loc_cpu, iloc_cpu により、ディスパッチが禁止されている状態で、自タスクを待ち状態に移す可能性のあるサービスコール (slp_tsk, wai_sem など)は発行しないで下さい。
- loc_cpu, iloc_cpu により CPU ロック状態で ena_dsp, dis_dsp は発行できません。
- dis_dsp を何回か発行して、その後、ena_dsp を 1 回発行しただけでディスパッチ禁止状態は解除されます。

12.4初期起動タスクについて

MR30 では、システム起動時に READY 状態からスタートするタスクを指定できます。つまり、タスク属性として TA_STA を付加します。この指定はコンフィギュレーションファイルで設定を行います。
設定方法の詳細については、239ページを参照して下さい。

12.5機種ごとの注意事項

12.5.1 M16C/62 グループをご使用の場合

- メモリ拡張機能をメモリ空間拡張モード 1(メモリ空間=1.2M)でご使用の場合
  MR30 のカーネル(MR_KERNEL セクション)を 30000H から FFFFFH 番地以内に配置して下さい。
- メモリ拡張機能をメモリ空間拡張モード 2(メモリ空間=4M)でご使用の場合
  MR30 のカーネル(MR_KERNEL セクション)を 3C0000H から 3FFFFFH 番地以内に配置して下さい。

12.5.2 M16C/6Nグループをご使用の場合

スタートアッププログラム中の下記に示す処理を MR30 のシステムタイマ設定箇所に追加して下さい。(MR30 のシステムタイマ設定箇所は、"インストール先ディレクトリ¥LIB30"ディレクトリ内のスタートアッププログラム crt0mr.a30 では 160 行目、start.a30 では、73 行目になります。)
なお、周辺機能クロック選択レジスタ(25E 番地---0 ビット目)で 1 分周モードを設定する場合は下記の処理の追加は必要ありません。

```
;+-------------------------------------------------------------+
;|  System timer interrupt setting                             |
;+-------------------------------------------------------------+
mov.b       #stmr_mod_val,stmr_mod_reg      ; set timer mode
;    mov.b   #1H,0AH
;    bset    6,07H
     mov.b   #stmr_int_IPL,stmr_int_reg      ; set timer IPL
;    bclr    6,07H
;    mov.b   #0,0AH
     mov.w   #stmr_cnt_stmr_ctr_reg          ; set interval count

mov.b   stmr_mod_reg,R0L            <---- 追加
and.b   #0C0H,R0L                   <---- 追加
jnz     __MR_SYSTIME_END            <---- 追加
mov.w   #stmr_cnt/2,stmr_ctr_reg    <---- 追加
__MR_SYSTIME_END:                         <---- 追加

     or.b    #stmr_bit+1,stmr_start
```

13. 別ROM化

13.1 別ROM化の方法

本章では、MR30 のカーネルとアプリケーションプログラムを別の ROM に配置する方法を示します。
図 13.1は、2 つの異なるアプリケーション間の共通部分とカーネル部分はカーネルROMに、アプリケーション部分はそれぞれ別のROMに配置した場合の例を示しています。 この例をもとに、ROM分割の方法を示します。

1. システム構成

アプリケーションプログラムのシステム構成を行います。
ここでは、2 つのアプリケーションプログラムのシステム構成が以下に示すものとして説明を行います。

	アプリケーション 1	アプリケーション 2
タスク数	4	5
イベントフラグ数	1	3
セマフォ数	4	2
メールボックス数	3	5
固定長メモリプール数	3	1
周期起動ハンドラ数	3	3

2. コンフィグレーションファイル作成

システム構成を行なった結果をもとに、コンフィグレーションファイルを作成します。この時、以下に示す定義部で注意が必要です。

- maxdefine 定義

  この maxdefine 定義部で設定する値は、2 つのアプリケーション間の各定義について定義数の大きい方を指定します。よって、この 2 つのアプリケーションにおいて各項目の値は、一致していなければなりません。

例

```
maxdefine{
        max_task        = 5;
        max_flag        = 3;
        max_sem         = 4;
        max_mbx         = 5;
        max_mpl         = 3;
        max_cyh         = 3;
};
```

●system 定義

この system 定義の以下に示す項目を 2 つのアプリケーションにおいて共通にする必要があります。

timeout
task_pause
priority

●clock 定義

この項目に設定する値は、2 つのアプリケーション間で異なっても問題ありませんが、1 つのアプリケーションではこの項目を定義し、もう 1 つのアプリケーションでは定義を省略するといったことはしないでください。必ず、この項目の設定および省略は 2 つのアプリケーション間で統一してください。

●task 定義

initial_start

この項目には、システム起動後最初に起動されるタスクのみを”ON”に指定し、その他のすべてのタスクには”OFF”を指定してください。

その他の定義については、2 つのコンフィグレーションファイル間で異なっても問題ありません。

3. プロセッサモードレジスタの変更

スタートアッププログラムのプロセッサモードレジスタをシステムに応じて変更します。

4. アプリケーションプログラム作成

2 つのアプリケーションプログラムを作成します。

5. スタートアッププログラムのセクション名変更

サンプルスタートアップファイル(start.a30、crt0mr.a30)では、スタートアッププログラムは MR_KERNEL セクションとして定義していますが、これを MR_KERNEL とは別のセクション名に変更して下さい。

例

```
.section MR_KERNEL,CODE,ALIGN
        ↓（変更）
.section MR_STARTUP,CODE,ALIGN
```

6. 各セクション配置設定

カーネル ROM とアプリケーション ROM に配置するプログラムを以下に示します。

●カーネル ROM に配置するプログラム

- ◆ MR30 のカーネル (MR_KERNEL セクション)
- ◆ 2 つのアプリケーション間の共通プログラム (program セクション)

この例では ID=1 のタスクが共通プログラムとします。共通プログラムはアプリケーション ROM に配置しても問題ありません。なお、共通プログラムをカーネル ROM に配置する場合は、以下に示す C 言語インタフェースルーチンを介して呼び出すサービスコールは発行できませんので、ご注意ください。

get_mpf, get_pri, get_tid, iprcv_dtq, pget_mpf, pget_mpl, pol_flg, prcv_dtq, prcv_mbx, rcv_dtq, rcv_mbx, tget_mpf, trcv_dtq, trcv_mbx, tsnd_dtq, twai_flg, viprcv_dtq, vprcv_dtq, vrcv_dtq, vtrcv_dtq, vtsnd_dtq, wai_flg

共通プログラムから上記のサービスコールを発行したい場合は、アプリケーション ROM に配置してください。

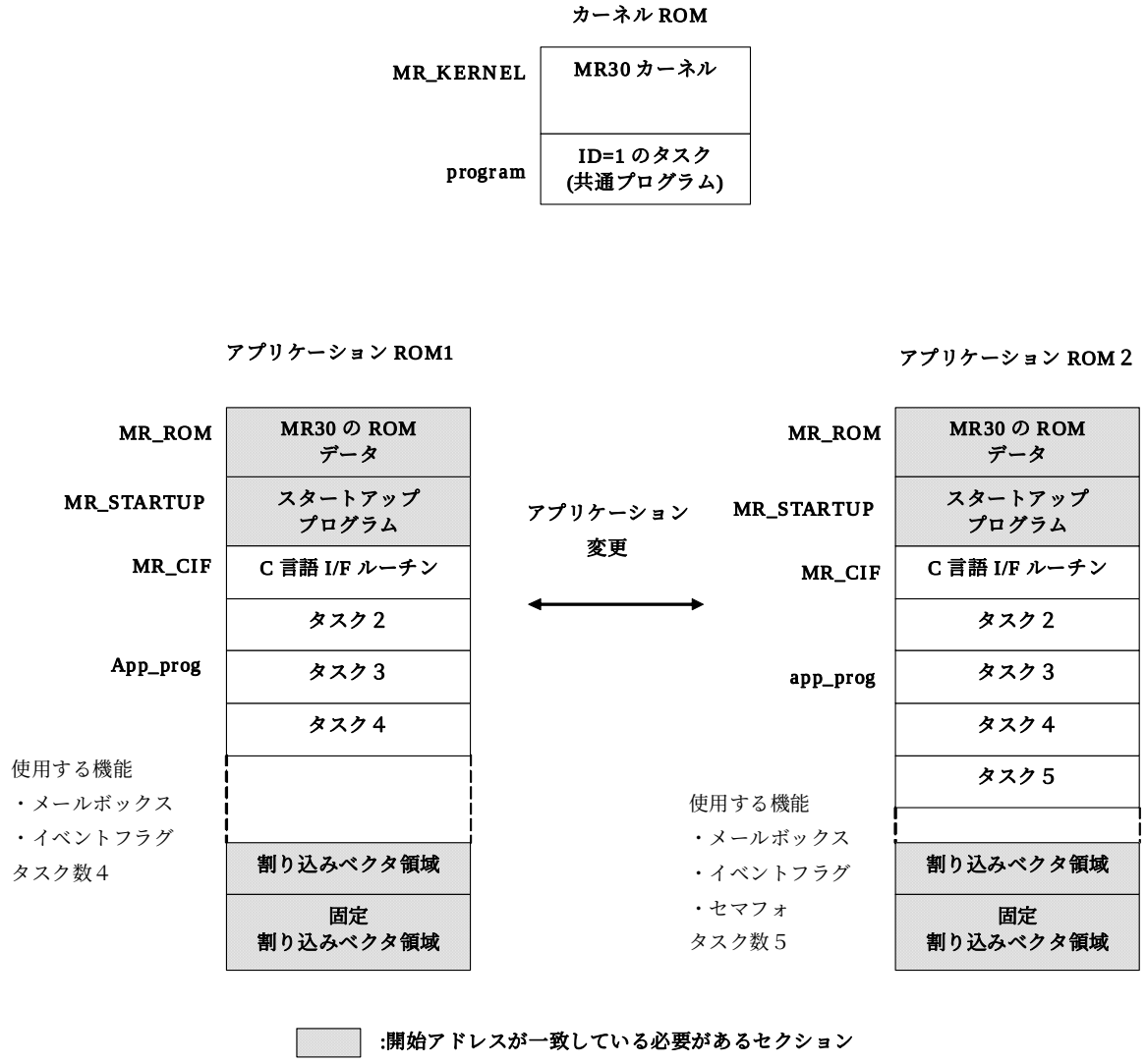


図 13.1 ROM 分割

●アプリケーション ROM に配置するプログラム

- ◆ スタートアッププログラム
- ◆ MR30 の ROM データ (MR_ROM セクション)
- ◆ C 言語 I/F ルーチン (MR_CIF セクション)
- ◆ アプリケーションプログラム (app_prog セクション)
- ◆ 割り込みベクタ領域 (INTERRUPT_VECTOR セクション)
- ◆ 固定割り込みベクタ領域(FIX_INTERRUPT_VECTOR セクション)

●各プログラムの配置方法を以下に示します。

ユーザプログラムのセクション名変更

C言語でアプリケーションプログラムを記述している場合は、以下に示すように #pragma SECTIONを使ってアプリケーションROMに配置するプログラムのセクション名を変更します。
NC30 では、ユーザプログラムのセクション名は、設定がなければprogramセクションになります。

そのため、アプリケーションROMに配置するタスクを別のセクション名に変更しなければなりません。[59]

```
#pragma SECTION program app_prog/* プログラムのセクションを変更 */
/* task2,task3 のセクション名は app_progに変わります。 */
void task2(void){
                :
}

void task3(void){
                :
}
```

●セクションの配置設定

セクションファイル (c_sec.inc、asm_sec.inc)を変更して、各セクションのアドレス設定を行ないます。この時、以下に示すセクションの開始アドレスは、2 つのアプリケーション間で一致していなければなりません。

- スタートアッププログラム
- MR30 の RAM データ(MR_RAM、MR_RAM_DBG セクション)
- MR_HEAP セクション
- MR30 カーネル(MR_KERNEL セクション)
- MR30 の ROM データ (MR_ROM セクション)
- 割り込みベクタ領域 (INTERRUPT_VECTOR セクション)

[59] カーネル ROM に配置するタスクのセクションは、変更する必要はありません。

以下に、セクションファイルの設定例を示します。

```
.section MR_RAM_DBG,DATA          ; MR30のRAMデータ
.org 500H                         ; 2つのアプリケーション間で共通アドレス
.section MR_RAM,DATA              ; MR30のRAMデータ
.org 600H                         ; 2つのアプリケーション間で共通アドレス
        :
.section MR_HEAP,DATA             ; MR_HEAPセクション
.org  10000H                      ; 2つのアプリケーション間で共通アドレス
        :
.section MR_ROM,ROMDATA           ; MR30のROMデータ
.org    0e0000H                   ; 2つのアプリケーション間で共通アドレス
        :
.section MR_STARTUP,CODE          ; スタートアッププログラム
.org    0e1000H                   ; 2つのアプリケーション間で共通アドレス

.section MR_CIF,CODE              ; C言語 I/F ルーチン
        :
.section app_prog,CODE            ;ユーザプログラム
        :
.section INTERRUPT_VECTOR         ; 可変割り込みベクタ
.org    0efd00H                   ; 2つのアプリケーション間で共通アドレス

.section MR_KERNEL,CODE           ; MR30のカーネル
.org    0f0000H                   ; 2つのアプリケーション間で共通アドレス
        :
.section FIX_INTERRUPT_VECTOR     ; 固定割り込みベクタ
.org    0fffdcH                   ; 2つのアプリケーション間で共通アドレス
```

図 13.2に示すメモリマップになります。

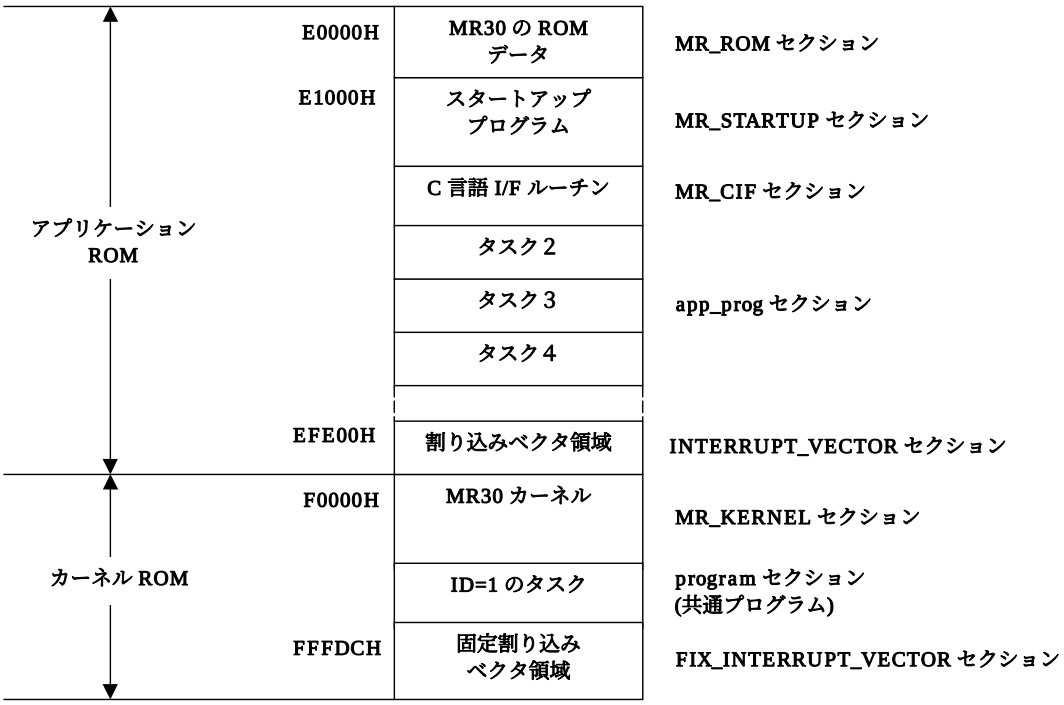


図 13.2 メモリマップ

7. コンフィグレータ **cfg30** を実行します。

8. 全サービスコール名を記述した **mrc** ファイルを作成してください。**(**コンパイルを行うとワークディレクトリに拡張子 **mrc** というファイルが生成されます。これを参考にして作成ください。**)**

9. システム生成

ビルドします。

10. **4**から **9**の処理をアプリケーション **2** についても行うことで、アプリケーション **2** のシステムが生成できます。

以上に述べた方法で、別 ROM 化が行なえます。

14. 付録

14.1 共通定数と構造体のパケット形式

```
----共通----
TRUE        1               /* 真 */
FALSE       0               /* 偽 */
----タスク管理関係----
TSK_SELF    0               /* 自タスク指定 */
TPRI_RUN    0           /* その時実行中の優先度を指定 */
typedef struct t_rtsk {
    STAT    tskstat;        /* タスクの状態 */
    PRI     tskpri;         /* タスクの現在優先度 */
    PRI     tskbpri;        /* タスクのベース優先度 */
    STAT    tskWAITING;     /* タスクの待ち要因 */
    ID      wid;            /* タスクの待ちオブジェクトID */
    TMO     tskatr;         /* タイムアウトするまでの時間 */
    UINT    actcnt;         /* 起動要求キューイング数 */
    UINT    wupcnt;         /* 起床要求キューイング数 */
    UINT    suscnt;         /* 強制待ち要求ネスト数 */
} T_RTSK;

typedef struct t_rtst {
    STAT    tskstat;        /* タスクの状態 */
    STAT    tskWAITING;     /* タスクの待ち要因 */
} T_RTST;

----セマフォ関係----
typedef struct t_rsem {
    ID      wtskid;         /* 待ち行列先頭タスクのID番号 */
    INT     semcnt;         /* 現在のセマフォカウントの値 */
} T_RSEM;

----イベントフラグ関係----
wfmod:
    TWF_ANDW   H'0000       /* AND待ち */
    TWF_ORW    H'0001       /* OR待ち */
typedef struct t_rflg {
    ID      wtskid;         /* 待ち行列先頭タスクのID番号 */
    UINT    flgptn;         /* イベントフラグの現在のビットパターン */
} T_RFLG;

----データキュー、shortデータキュー関係----
typedef struct t_rdtq {
    ID      stskid;         /* 送信待ち行列先頭タスクのID番号 */
    ID      rtskid;         /* 受信待ち行列先頭タスクのID番号 */
    UINT    sdtqcnt;        /* データキューに入っているデータの個数 */
} T_RDTQ;
```

```
----メールボックス関係----
typedef struct      t_msg  {
    VP      msghead;        /* メッセージヘッダ */
} T_MSG;
typedef struct t_msg_pri {
    T_MSG           msgque; /* メッセージヘッダ */
    PRI             msgpri; /* メッセージ優先度 */
} T_MSG_PRI;

typedef struct t_mbx {
    ID      wtskid;         /* 待ち行列先頭タスクのID番号*/
    T_MSG   *pk_msg;        /* 次に受信されるメッセージ */
} T_RMBX;

----固定長メモリプール関係----
typedef struct t_rmpf {
    ID      wtskid;         /* メモリ獲得待ち行列先頭タスクのID番号*/
    UINT    frbcnt;         /* メモリブロック数 */
} T_RMPF;

----可変長メモリプール関係----
typedef struct t_rmpl {
    ID      wtskid;         /* メモリ獲得待ち行列先頭タスクのID番号*/
    SIZE    fmplsz;         /* 空き領域の合計サイズ */
    UINT    fblksz;         /* すぐに獲得可能な最大メモリブロックサイズ */
} T_RMPL;

----周期ハンドラ関係----
typedef struct  t_rcyc {
    STAT    cycstat;        /* 周期ハンドラ動作状態 */
    RELTIM  lefttim;        /* 周期ハンドラの起動までの残り時間 */
} T_RCYC;

----アラームハンドラ関係----
typedef struct t_ralm {
    STAT    almstat;        /* アラームハンドラ動作状態 */
    RELTIM  lefttim;        /* アラームハンドラの起動までの残り時間 */
} T_RALM;

/* システム管理関係 */
typedef struct t_rver {
    UH      maker;          /* メーカー */
    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;
```

14.2 アセンブリ言語インタフェース

アセンブリ言語でサービスコールを発行する場合、サービスコールの呼び出し用マクロを使用します。

サービスコールの呼び出し用マクロ内の処理は、各パラメータをレジスタに設定してから、ソフトウエア割り込みによりサービスコールのルーチンの実行を開始します。また、サービスコールの呼び出し用マクロを使用せず直接サービスコールを呼び出した場合、将来のバージョンにおいて互換性が保証できなくなります。

以下にアセンブリ言語インタフェースの一覧表を記載します。機能コードについては、μITRON 仕様で規定された値は使用しておりません。

タスク管理機能

ServiceCall	INTNo.	Parameter					ReturnParameter	
		FuncCode R0	R1	R3	A0	A1	R0	A0
act_tsk	32	0	-	-	tskid	-	ercd	-
iact_tsk	33	2	-	-	tskid	-	ercd	-
can_act	33	4	-	-	tskid	-	actcnt	-
ican_act	33	4	-	-	tskid	-	actcnt	-
sta_tsk	32	6	stacd	-	tskid	-	ercd	-
ista_tsk	33	8	stacd	-	tskid	-	ercd	-
ext_tsk	37	-	-	-	-	-	-	-
ter_tsk	32	10	-	-	tskid	-	ercd	-
chg_pri	32	12	-	tskpri	tskid	-	ercd	-
ichg_pri	33	14	-	tskpri	tskid	-	ercd	-
get_pri	33	16	-	-	tskid	-	ercd	tskpri
iget_pri	33	16	-	-	tskid	-	ercd	tskpri
ref_tsk	33	18	-	-	tskid	pk_rtsk	ercd	-
iref_tsk	33	18	-	-	tskid	pk_rtsk	ercd	-
ref_tst	33	20	-	-	tskid	pk_rtst	ercd	-
iref_tst	33	20	-	-	tskid	pk_rtst	ercd	-

タスク付属同期

ServiceCall	INTNo.	Parameter					ReturnParameter
		FuncCode R0	R1	R3	A0	A1	R0
slp_tsk	32	22	-	-	-	-	ercd
tslp_tsk	32	24	tmout	tmout	-	-	ercd
wup_tsk	32	26	-	-	tskid	-	ercd
iwup_tsk	33	28	-	-	tskid	-	ercd
can_wup	33	30	-	-	tskid	-	wupcnt
ican_wup	33	30	-	-	tskid	-	wupcnt
rel_wai	32	32	-	-	tskid	-	ercd
irel_wai	33	34	-	-	tskid	-	ercd
sus_tsk	32	36	-	-	tskid	-	ercd
isus_tsk	33	38	-	-	tskid	-	ercd
rsm_tsk	32	40	-	-	tskid	-	ercd
irsm_tsk	33	42	-	-	tskid	-	ercd
frsm_tsk	32	40	-	-	tskid	-	ercd
ifrsm_tsk	33	42	-	-	tskid	-	ercd
dly_tsk	32	44	tmout	tmout	-	-	ercd

同期・通信機能

ServiceCall	INTNo.	Parameter						ReturnParameter			
		FuncCode R0	R1	R2	R3	A0	A1 FuncCode	R0	R1	R2	R3
sig_sem	32	46	-	-	-	semid	-	ercd	-	-	-
isig_sem	33	48	-	-	-	semid	-	ercd	-	-	-
wai_sem	32	50	-	-	-	semid	-	ercd	-	-	-
pol_sem	33	52	-	-	-	semid	-	ercd	-	-	-
ipol_sem	33	52	-	-	-	semid	-	ercd	-	-	-
twai_sem	32	54	tmout	-	tmout	semid	-	ercd	-	-	-
ref_sem	33	56	-	-	-	semid	pk_rsem	ercd	-	-	-
iref_sem	33	56	-	-	-	semid	pk_rsem	ercd	-	-	-
set_flg	32	58	-	-	setptn	flgid	-	ercd	-	-	-
iset_flg	33	60	-	-	setptn	flgid	-	ercd	-	-	-
clr_flg	33	62	-	-	clrptn	flgid	-	ercd	-	-	-
iclr_flg	33	62	-	-	clrptn	flgid	-	ercd	-	-	-
wai_flg	32	64	wfmode	-	waiptn	flgid	-	ercd	-	flgptn	-
twai_flg	38	tmout	wfmode	tmout	waiptn	flgid	68	ercd	-	flgptn	-
pol_flg	33	66	wfmode	-	waiptn	flgid	-	ercd	-	flgptn	-
ipol_flg	33	66	wfmode	-	waiptn	flgid	-	ercd	-	flgptn	-
ref_flg	33	70	-	-	-	flgid	pk_rflg	ercd	-	-	-
iref_flg	33	70	-	-	-	flgid	pk_rflg	ercd	-	-	-
snd_dtq	32	72	data	-	-	dtqid	-	ercd	-	-	-
psnd_dtq	32	74	data	-	-	dtqid	-	ercd	-	-	-
ipsnd_dtq	33	76	data	-	-	dtqid	-	ercd	-	-	-
fsnd_dtq	32	80	data	-	-	dtqid	-	ercd	-	-	-
ifsnd_dtq	33	82	data	-	-	dtqid	-	ercd	-	-	-
tsnd_dtq	38	tmout	data	tmout	-	dtqid	78	ercd	-	-	-
rcv_dtq	32	84	-	-	-	dtqid	-	ercd	data	-	-
prcv_dtq	32	86	-	-	-	dtqid	-	ercd	data	-	-
iprcv_dtq	33	88	-	-	-	dtqid	-	ercd	data	-	-
trcv_dtq	32	90	tmout	-	tmout	dtqid	-	ercd	data	-	-
ref_dtq	33	92	-	-	-	dtqid	pk_rdtq	ercd	-	-	-
iref_dtq	33	92	-	-	-	dtqid	pk_rdtq	ercd	-	-	-

同期・通信機能（つづき）

ServiceCall	INTNo.	Parameter						ReturnParameter			
		FuncCode R0	R1	R2	R3	A0	A1	R0	R1	R2	R3
snd_mbx	32	94	-	-	-	mbxid	pk_msg	ercd	-	-	-
isnd_mbx	33	96	-	-	-	mbxid	pk_msg	ercd	-	-	-
rcv_mbx	32	98	-	-	-	mbxid	-	ercd	pk_msg	-	-
prcv_mbx	33	100	-	-	-	mbxid	-	ercd	pk_msg	-	-
iprcv_mbx	33	100	-	-	-	mbxid	-	ercd	pk_msg	-	-
trcv_mbx	32	102	tmout	-	tmout	mbxid	-	ercd	pk_msg	-	-
ref_mbx	33	104	-	-	-	mbxid	pk_rmbx	ercd	-	-	-
iref_mbx	33	104	-	-	-	mbxid	pk_rmbx	ercd	-	-	-

メモリプール管理

ServiceCall	INTNo.	Parameter						ReturnParameter			
		FuncCode R0	R1	R2	R3	A0	A1	R0	R1	R2	R3
get_mpf	32	108	-	-	-	mpfid	-	ercd	p_blk	-	-
pget_mpf	33	106	-	-	-	mpfid	-	ercd	p_blk	-	-
ipget_mpf	33	106	-	-	-	mpfid	-	ercd	p_blk	-	-
tget_mpf	32	110	tmout	-	tmout	mpfid	-	ercd	p_blk	-	-
rel_mpf	32	112	blk	-	-	mpfid	-	ercd	-	-	-
irel_mpf	33	114	blk	-	-	mpfid	-	ercd	-	-	-
ref_mpf	33	116	-	-	-	mpfid	pk_rmpf	ercd	-	-	-
iref_mpf	33	116	-	-	-	mpfid	pk_rmpf	ercd	-	-	-
pget_mpl	32	118	-	-	-	mplid	-	ercd	p_blk	-	-
rel_mpl	32	120	blk	-	-	mplid	-	ercd	-	-	-
ref_mpl	33	122	-	-	-	mplid	pk_rmpl	ercd	-	-	-
iref_mpl	33	122	-	-	-	mplid	pk_rmpl	ercd	-	-	-

時間管理機能

ServiceCall	INTNo.	Parameter					ReturnParameter
		FuncCode R0	R1	R3	A0	A1	R0
set_tim	33	124	-	-	p_systim	-	ercd
iset_tim	33	124	-	-	p_systim	-	ercd
get_tim	33	126	-	-	p_systim	-	ercd
iget_tim	33	126	-	-	p_systim	-	ercd
sta_cyc	33	128	-	-	cycid	-	ercd
ista_cyc	33	128	-	-	cycid	-	ercd
stp_cyc	33	130	-	-	cycid	-	ercd
istp_cyc	33	130	-	-	cycid	-	ercd
ref_cyc	33	132	-	-	cycid	pk_rcyc	ercd
iref_cyc	33	132	-	-	cycid	pk_rcyc	ercd
sta_alm	33	134	almtim	almtim	almid	-	ercd
ista_alm	33	134	almtim	almtim	almid	-	ercd
stp_alm	33	136	-	-	almid	-	ercd
istp_alm	33	136	-	-	almid	-	ercd
ref_alm	33	138	-	-	almid	pk_ralm	ercd
iref_alm	33	138	-	-	almid	pk_ralm	ercd

システム状態管理機能
割り込み管理機能

ServiceCall	INTNo.	Parameter		ReturnParameter	
		FuncCode R0	R3	R0	A0
rot_rdq	32	140	pri	ercd	-
irot_rdq	33	142	pri	ercd	-
get_tid	33	144		ercd	tskid
iget_tid	33	144		ercd	tskid
loc_cpu	36	-	-	ercd	-
iloc_cpu	36	-	-	ercd	-
unl_cpu	32	146	-	ercd	-
iunl_cpu	33	148	-	ercd	-
dis_dsp	35	-	-	ercd	-
ena_dsp	32	150	-	ercd	-
sns_ctx	33	152	-	state	-
sns_loc	33	154	-	state	-
sns_dsp	33	156	-	state	-
sns_dpn	33	158	-	state	-
ret_int	34	--	--	--	--

システム構成管理機能

ServiceCall	INTNo.	Parameter		ReturnParameter
		FuncCode R0	A0	R0
ref_ver	33	160	pk_rver	ercd
iref_ver	33	160	pk_rver	ercd

拡張機能(リセット機能)

ServiceCall	INTNo.	Parameter		ReturnParameter
		FuncCode R0	A0	R0
vrst_vdtq	32	192	vdtqid	ercd
vrst_dtq	32	184	dtqid	ercd
vrst_mbx	33	186	mbxid	ercd
vrst_mpf	32	188	mpfid	ercd
vrst_mpl	33	190	mplid	ercd

拡張機能(long データキュー)

ServiceCall	INTNo.	Parameter						ReturnParameter			
		FuncCode R0	R1	R2	R3	A0	A1 FuncCode	R0	R1	R2	R3
vsnd_dtq	32	162	data	-	data	vdtqid	-	ercd	-	-	-
vpsnd_dtq	32	164	data	-	data	vdtqid	-	ercd	-	-	-
vipsnd_dtq	33	166	data	-	data	vdtqid	-	ercd	-	-	-
vfsnd_dtq	32	170	data	-	data	vdtqid	-	ercd	-	-	-
vifsnd_dtq	33	172	data	-	data	vdtqid	-	ercd	-	-	-
vtsnd_dtq	38	tmout	data	tmout	data	vdtqid	168	ercd	-	-	-
vrcv_dtq	32	174	-	-	-	vdtqid	-	ercd	data	-	data
vprcv_dtq	32	176	-	-	-	vdtqid	-	ercd	data	-	data
viprcv_dtq	33	178	-	-	-	vdtqid	-	ercd	data	-	data
vtrcv_dtq	32	180	tmout	-	tmout	vdtqid	-	ercd	data	-	data
vref_dtq	33	182	-	-	-	vdtqid	pk_rdtq	ercd	-	-	-
viref_dtq	33	182	-	-	-	vdtqid	pk_rdtq	ercd	-	-	-

M16C シリーズ,R8C ファミリ用リアルタイム OS
M3T-MR30/4 V.4.01
ユーザーズマニュアル

発行年月日　2011 年 6 月 1 日　Rev.1.00

発行　ルネサス エレクトロニクス株式会社
〒211-8668 神奈川県川崎市中原区下沼部 1753

編集　株式会社ルネサス ソリューションズ

■営業お問合せ窓口

ルネサス エレクトロニクス株式会社

http://www.renesas.com

※営業お問合せ窓口の住所・電話番号は変更になることがあります。最新情報につきましては、弊社ホームページをご覧ください。

ルネサス エレクトロニクス販売株式会社　〒100-0004　千代田区大手町2-6-2（日本ビル）　(03)5201-5307

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：http://japan.renesas.com/inquiry

M16C シリーズ,R8C ファミリ用リアルタイム OS
M3T-MR30/4 V.4.01
ユーザーズマニュアル

ルネサス エレクトロニクス株式会社

R20UT0655JJ0100