お客様各位

カタログ等資料中の旧社名の扱いについて

2010年4月1日を以ってNECエレクトロニクス株式会社及び株式会社ルネサステクノロジが合併し、両社の全ての事業が当社に承継されております。従いまして、本資料中には旧社名での表記が残っておりますが、当社の資料として有効ですので、ご理解の程宜しくお願い申し上げます。

ルネサスエレクトロニクス ホームページ（http://www.renesas.com）

2010年4月1日
ルネサスエレクトロニクス株式会社

【発行】ルネサスエレクトロニクス株式会社（http://www.renesas.com）

【問い合わせ先】http://japan.renesas.com/inquiry

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。

User's Manual

RENESAS

M3T-MR308 V.1.20

リファレンスマニュアル

M32C/80,M16C/80,70 シリーズ用リアルタイムOS

ルネサス エレクトロニクス
www.renesas.com

Rev.2.00　2003.09

安全設計に関するお願い

- 弊社は品質、信頼性の向上に努めておりますが、半導体製品は故障が発生したり、誤動作する場合があります。弊社の半導体製品の故障又は誤動作によって結果として、人身事故火災事故、社会的損害などを生じさせないような安全性を考慮した冗長設計、延焼対策設計、誤動作防止設計などの安全設計に十分ご留意ください。

本資料ご利用に際しての留意事項

- 本資料は、お客様が用途に応じた適切なルネサス テクノロジ製品をご購入いただくための参考資料であり、本資料中に記載の技術情報について株式会社ルネサス テクノロジおよび株式会社ルネサス ソリューションズが所有する知的財産権その他の権利の実施、使用を許諾するものではありません。
- 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他応用回路例の使用に起因する損害、第三者所有の権利に対する侵害に関し、株式会社ルネサス テクノロジおよび株式会社ルネサス ソリューションズは責任を負いません。
- 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他全ての情報は本資料発行時点のものであり、株式会社ルネサス テクノロジおよび株式会社ルネサス ソリューションズは、予告なしに、本資料に記載した製品又は仕様を変更することがあります。ルネサス テクノロジ半導体製品のご購入に当たりましては、事前に株式会社ルネサス テクノロジ、株式会社ルネサス ソリューションズ、株式会社ルネサス販売又は特約店へ最新の情報をご確認頂きますとともに、ルネサス テクノロジホームページ（http://www.renesas.com）などを通じて公開される情報に常にご注意ください。
- 本資料に記載した情報は、正確を期すため、慎重に制作したものですが万一本資料の記述誤りに起因する損害がお客様に生じた場合には、株式会社ルネサス テクノロジおよび株式会社ルネサス ソリューションズはその責任を負いません。
- 本資料に記載の製品データ、図、表に示す技術的な内容、プログラム及びアルゴリズムを流用する場合は、技術内容、プログラム、アルゴリズム単位で評価するだけでなく、システム全体で十分に評価し、お客様の責任において適用可否を判断してください。株式会社ルネサス テクノロジおよび株式会社ルネサス ソリューションズは、適用可否に対する責任は負いません。
- 本資料に記載された製品は、人命にかかわるような状況の下で使用される機器あるいはシステムに用いられることを目的として設計、製造されたものではありません。本資料に記載の製品を運輸、移動体用、医療用、航空宇宙用、原子力制御用、海底中継用機器あるいはシステムなど、特殊用途へのご利用をご検討の際には、株式会社ルネサス テクノロジ、株式会社ルネサス ソリューションズ、株式会社ルネサス販売又は特約店へご照会ください。
- 本資料の転載、複製については、文書による株式会社ルネサス テクノロジおよび株式会社ルネサス ソリューションズの事前の承諾が必要です。
- 本資料に関し詳細についてのお問い合わせ、その他お気付きの点がございましたら株式会社ルネサス テクノロジ、株式会社ルネサス ソリューションズ、株式会社ルネサス販売又は特約店までご照会ください。

製品の内容及び本書についてのお問い合わせ先

インストーラが生成する以下のテキストファイルに必要事項を記入の上、ツール技術サポート窓口 support_tool@renesas.com **まで送信ください。**

¥SUPPORT¥**製品名**¥SUPPORT.TXT

株式会社ルネサス ソリューションズ マイコンツール部

ツール技術サポート窓口	support_tool@renesas.com
ユーザ登録窓口	regist_tool@renesas.com
ホームページ	http://www.renesas.com/jp/tools

はじめに

MR308 は M16C/80 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]です。MR308 はμITRON 仕様[2]に準拠しています。

本マニュアルは MR308 の機能と構成について説明します。

ソフトウエアの使用権

ソフトウエアの使用権はソフトウエア使用権許諾契約書に基づきます。

MR308 はお客様の製品開発の目的でのみ使用できます。その他の目的での使用はできませんのでご注意ください。

また、本マニュアルによってソフトウエ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。

ドキュメント一覧

MR308 に添付されているドキュメントは以下の３種類あります。

- リリースノート

  ソフトウェアの概要やユーザーズマニュアル、リファレンスマニュアルの訂正など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）

  MR308 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。

- リファレンスマニュアル（PDF ファイル）

  MR308 のシステムコールの使用方法や使用例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。

[1] 以降リアルタイム OS と略します。

[2] μITRON 仕様は、東京大学理学部坂村健博士とその研究室により考案されたものです。したがって、μITRON 仕様の著作権は同氏に属しています。MR308 は同氏に承認を得て、μITRON 仕様に基づき製作されたものです。

目次

第1章　システムコールリファレンスの見方

1.1. システムコールリファレンスの見方

システムコールリファレンスは、以下の形式で記述しています。

【システムコール名】

システムコールの名称 → システムコールの機能

【アセンブリ言語による呼び出し方法】

.include mr308.inc
アセンブリ言語から **MR308** 機能呼び出し方法

《引数》

システムコールのパラメータの説明

パラメータはマクロの引数として記述します。
引数名　サイズ　説明
サイズは、以下の記号で示しています。
[---*]　1 バイトデータ
[--]　2** バイトデータ
[-*]　3** バイトデータ
[****]　4 バイトデータ
1 バイトデータの引き数として使用可能なレジスタは、R0H/R0L,R1H/R1L のみです。

《レジスタ設定》

システムコールマクロ発行後のレジスタに設定される値を示します。

レジスタ名	システムコール発行後の内容
*1	*2

*1 レジスタ名。この欄には、R0、R1、R2、R3 を記述しています。
*2 各レジスタに何が設定されるかを示しています。'—'が記述してあれば、「そのレジスタを使用することを設定していれば、保存されるが、使用することを設定していなければ、不定となる」ことを示します。

FLG の IPL,U,I,B はシステムコール発行前の値が保存されますが、その他のフラグについては不定です。

また、この表に記載していないレジスタ(SB,FB)についても同様に、そのレジスタを使用することを設定していれば、保存されますが、使用することを設定していなければ、不定となります。

各(リターン)パラメータが使用するレジスタは、おおよそ以下のように決めています。

R0 レジスタ(16 ビット)	機能コード(システムコール用マクロ内で設定)、エラーコード
R1 レジスタ(16 ビット)	wfmode(イベントフラグの待ちモード)、 パケットアドレスの下位 16 ビット
R2 レジスタ(16 ビット)	パケットアドレス 16 ビット、その他のパラメータ
R3 レジスタ(16 ビット)	パケットアドレス上位 16 ビット、その他のパラメータ
A0 レジスタ(24 ビット)	オブジェクトの ID 番号
A1 レジスタ(24 ビット)	パケットアドレス 24 ビット

【 C 言語による呼び出し方法 】

C 言語からの **MR308** 機能呼び出し方法

《引数》

引き数の型の宣言

《戻り値》

呼び出しの結果の戻り値の説明

なおシステムコールリファレンス文中で使用されているデータ型はインクルードファイル "mr308.h" に定義されています。付録にその定義を示しています。

【エラーコード】

エラーコード名　エラーコード値: エラーコードの意味
エラーコードの文字列、例えば E_OK などは"mr308.h"に"#define"を使って、mr308.inc に ".EQU" を使って定義されている。エラー判定をプログラム中で記述する場合は、この定義された文字列を用いなければなりません。

【機能説明】

機能の詳細説明

【使用例】

使用例

1.2. 各システムコール処理時間と最大割り込み禁止時間

各システムコールの処理速度（引数渡し処理を除く）と最大割り込み禁止時間を以下に示します。
最大割り込み禁止時間は、OS 依存割り込み(システムコールを発行している割り込み)に対しての時間です。
なお、OS 独立割り込み(システムコールを発行しない割り込み)に対する最大割り込み禁止時間は、

1μs

です。
この処理速度は、動作周波数 20MHz、16 ビットバス、ノーウェイトで計測した結果です。
単位は、μs です。α は、スケジューラの処理時間です。

システムコール	処理時間	最大割り込み禁止時間	システムコール	処理時間	最大割り込み禁止時間
sta_tsk	11+α	13	isig_sem	14	14
ista_tsk	11	11	wai_sem	9+α	11
ext_tsk	4+α	4	preq_sem	6	6
ter_tsk	16+α	17	ref_sem	6	6
dis_dsp	3	3	snd_msg	15+α	16
ena_dsp	9+α	9	isnd_msg	15	15
chg_pri	11+α	12	rcv_msg	9+α	11
ichg_pri	8	8	prcv_msg	8	8
rot_rdq	8+α	9	ref_mbx	7	7
irot_rdq	7	7	loc_cpu	4	-
rel_wai	18+α	19	unl_cpu	7+α	-
irel_wai	18	18	set_tim	6	6
get_tid	6	6	get_tim	6	6
ref_tsk	10	10	pget_blf	15	14
sus_tsk	8+α	9	rel_blf	9	9
isus_tsk	7	7	ref_mpf	15	15
rsm_tsk	8+α	9	pget_blk	29+α	6
irsm_tsk	8	8	rel_blk	16+α以上	6
slp_tsk	7+α	8	ref_mpl	11	11
wup_tsk	11+α	13	act_cyc	6	6
iwup_tsk	11	11	ref_cyc	6	6
can_wup	7	6	ref_alm	9	9
wai_flg	10+α	12	get_ver	11	11
clr_flg	5	5	vrst_msg	6	6
pol_flg	8	6	vrst_blf	5	5
ref_flg	6	6	vrst_blk	35	4
sig_sem	14+α	16			

システムコール	β	処理時間	最大割り込み禁止時間
dly_tsk	0	7＋α	13
	1 以上	11+4(β-1)+α	
tslp_tsk	0	10＋α	14
	1 以上	12+4(β-1)+α	
twai_flg	0	12＋α	17
	1 以上	12+4(β-1)+α	
twai_sem	0	6＋α	16
	1 以上	11+4(β-1)+α	
trcv_msg	0	8＋α	16
	1 以上	10+4(β-1)+α	

β: タイムアウトキューを検索したタスク数

システムコール	β	処理時間	最大割り込み禁止時間
set_flg	0	6+4γ＋α	20
	1以上	18+15(β-1)+4γ+α	
iset_flg	0	7+4γ	18
	1以上	20＋15(β-1)+4γ	

β: 待ち条件をみたしたタスク数
γ: 待ち条件をみたさないタスク数

ret_int の処理時間と割り込み禁止時間は、6ページを参照してください。

1.3. スケジューラの処理時間

1.3.1. システムコールのスケジューラ処理時間

タスクからのシステムコールにより起動されるスケジューラの処理時間、および割り込み禁止時間を以下に示します。

- 処理時間

  5+0.7(x-1) μs
  x: 優先度

- 割り込み禁止時間

  5+0.7 (x-1) μs
  x: 優先度(32 以上の場合は、32 で計算)

1.3.2. 割り込みハンドラのスケジューラ処理時間

割り込み復帰時に起動されるスケジューラの処理時間、および割り込み禁止時間を以下に示します。

- 処理時間

  10+0.7(x-1) μs
  x: 優先度

- 割り込み禁止時間

  6+0.6(x-1) μs
  x: 優先度(32 以上の場合は、32 で計算)

1.3.3. システムタイマの処理時間

- 処理時間

  システムタイマの処理時間 = 5 + (システム時刻更新時間 +)
  (周期起動ハンドラ処理時間 +)
  (アラームハンドラ処理時間 +)
  タイムアウト起床処理時間 +
  ret_int 処理 μs

  システム時刻更新時間 = 1μs
  周期起動ハンドラ処理時間 = 2× n + Σ (1.1 + ハンドラ実行時間)
  アラームハンドラ処理時間 = 2+ Σ (4× n + ハンドラ実行時間)
  n: 登録ハンドラ数
  タイムアウト起床処理時間 = 3 (起床するタスクがないとき)
  9+12(n-1) （起床するタスクがあるとき）
  n: 起床するタスク数

- 最大割り込み禁止時間

  12μs

1.4. 各システムコールのスタック使用量

表 1.1は、タスクから発行可能なシステムコールのスタック使用量(ユーザースタック及び、システムスタック)を示しています。

表 1.1　タスクから発行するシステムコールのスタック使用量一覧(単位:バイト)

システムコール	スタックサイズ		システムコール	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
sta_tsk	0	4	twai_flg	0 (6*)	8
ext_tsk	0	4	sig_sem	0	8
ter_tsk	0	8	wai_sem	0	4
dis_dsp	0	0	twai_sem	0	8
ena_dsp	0	0	snd_msg	0	8
chg_pri	0	4	rcv_msg	0 (8*)	4
rot_rdq	0	0	trcv_msg	0 (8*)	8
rel_wai	0	8	loc_cpu	0	0
sus_tsk	0	4	unl_cpu	0	0
rsm_tsk	0	4	dly_tsk	0	8
slp_tsk	0	4	pget_blk	0 (8*)	34
tslp_tsk	0	8	rel_blk	0	62
wup_tsk	0	8	vrst_msg*	12	0
set_flg	0	8	vrst_blf*	12	0
wai_flg	0 (6*)	4	vrst_blk*	50	0

*: C 言語で使用時に必要となるスタック使用量。

表 1.2は、ハンドラから発行可能なシステム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 1.2　ハンドラから発行するシステムコールのスタック使用量一覧(単位:バイト)

システムコール	スタックサイズ	システムコール	スタックサイズ
ista_tsk	16	iwup_tsk	24
ichg_pri	16	iset_flg	28
irot_rdq	16	isig_sem	24
irel_wai	20	isnd_msg	28
isus_tsk	16	ret_int	12
irsm_tsk	16		

表 1.3は、タスクあるいはハンドラの両方から発行可能なシステム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、タスクからシステムコールを発行した場合は、ユーザースタックを使用し、ハンドラから発行した場合は、システムスタックを使用します。

表 1.3　タスク、ハンドラの両方から発行可能なシステムコールのスタック使用量一覧

システムコール	スタックサイズ	システムコール	スタックサイズ
get_tid	12 (20*)	act_cyc	12
can_wup	12 (18*)	get_ver	16
clr_flg	12	ref_tsk	20
pol_flg	12 (18*)	ref_flg	12
preq_sem	12	ref_sem	12
prcv_msg	16 (24*)	ref_mbx	12
pget_blf	12 (24*)	ref_mpf	16
rel_blf	20	ref_mpl	12
set_tim	12	ref_cyc	12
get_tim	12	ref_alm	12

*: C 言語で使用時に必要となるスタック使用量。

1.5. スタックサイズの算出方法

MR308 のスタックには、システムスタックとユーザースタックの 2 種類があります。スタックサイズの計算方法は、ユーザースタックとシステムスタックで異なります。

●ユーザースタック

タスクに存在するスタックです。従って、MR308 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、各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

MR308 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。MR308 では、システムコールをタスクが発行するとユーザー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。

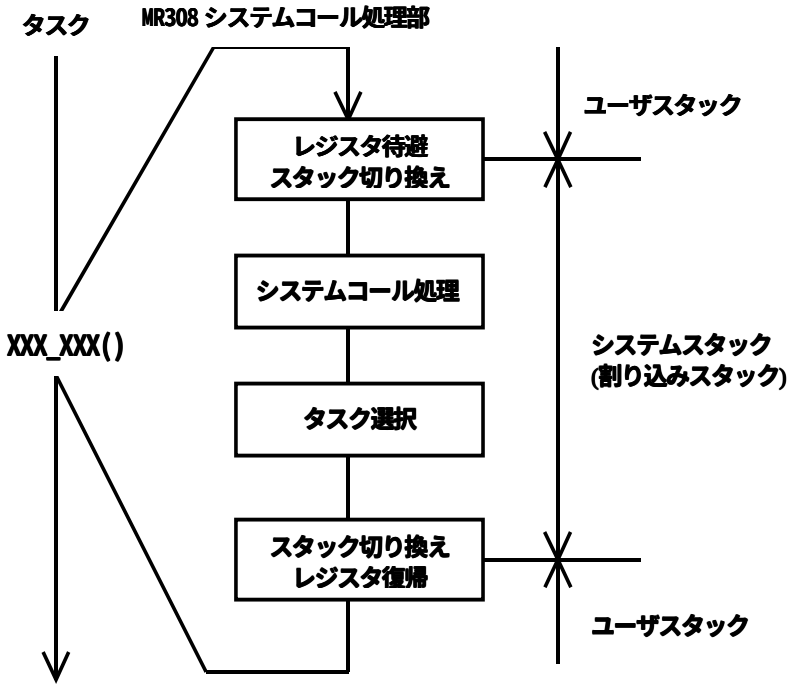


図 1.1:システムスタックとユーザースタック

システムスタックとユーザー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。

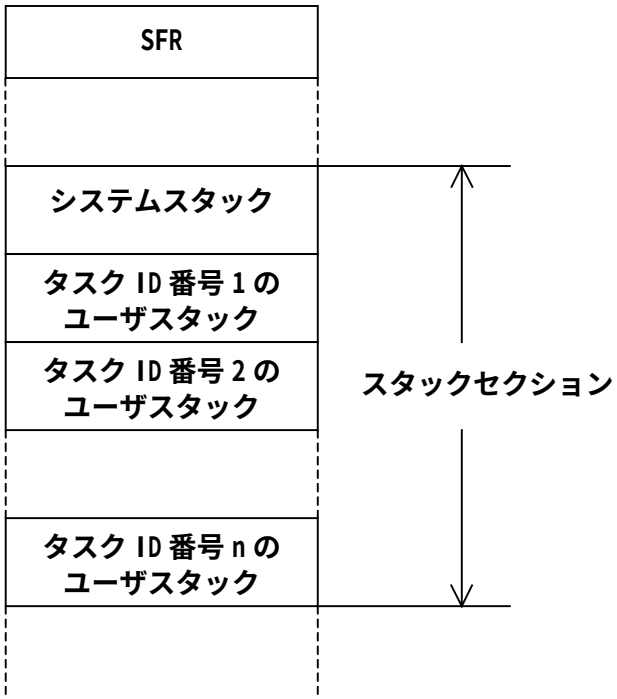


図 1.2:スタックの配置

1.5.1. ユーザースタックの算出方法

ユーザースタックは、各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

STK ビューワ[3]をご使用下さい。
STK ビューワ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 30 バイト[4]の合計が、タスクのスタックサイズとなります。
STK ビューワの詳細な使用方法については、STK ビューワ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

- **ユーザー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。
- **MR308 で使用する部分**
  システムコールを発行することで消費するスタックサイズです。
  MR308 では、タスクから発行可能なシステムコールのみを発行した場合は、PC+FLG レジスタを格納する領域 6 バイトを確保してください。また、タスクまたはハンドラの両方から発行できるシステムコールを発行した場合は、表 1.3に記載されたスタックサイズを参考に確保して下さい。
  複数のシステムコールを発行している場合は、それらのシステム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、
ユーザースタックサイズ =
ユーザープログラムで使用する部分 + 使用するレジスタ分 + MR308 で使用する部分
になります。(使用するレジスタ分は、R0,R1,R2,R3 の場合は各 2byte、A0,A1,SB,FB の場合は各 4byte で使用分を加算する)
図 1.3にユーザー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。

[3] STK ビューワは、ルネサス製 C コンパイラ NC308WA に付属されているスタックを計算するためのツールです。
[4] C 言語で記述した場合、このサイズは固定となります。

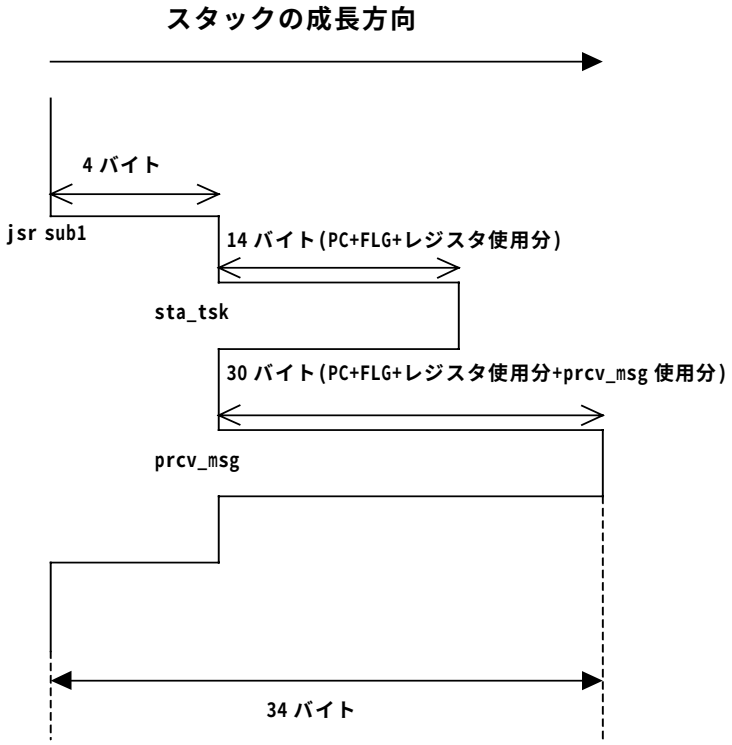


図 1.3:ユーザースタックサイズの算出例

1.5.2. 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システムコール処理中[5]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

  使用するシステムコールの中で最大のシステムスタックサイズ[6]。

  例えば、sta_tsk、ext_tsk、slp_tsk、dly_tsk を使用する場合、表 1.1で、それぞれのシステムスタックサイズを調べると、

システムコール名	システムスタックサイズ
sta_tsk	4 バイト
ext_tsk	4 バイト
slp_tsk	4 バイト
dly_tsk	8 バイト

  となるのでこの場合、使用するシステム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 8 バイトです。

- βi

  割り込みハンドラ[7]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。

[5] ユーザースタックからシステムスタックに切り替えた後
[6] それぞれのシステムコールに使用するスタックサイズは、表 1.1から表 1.3を参照してください。
[7] OS 依存割り込みハンドラ(ここでは、システムクロック割り込みハンドラを含まない)、OS 独立ハンドラ

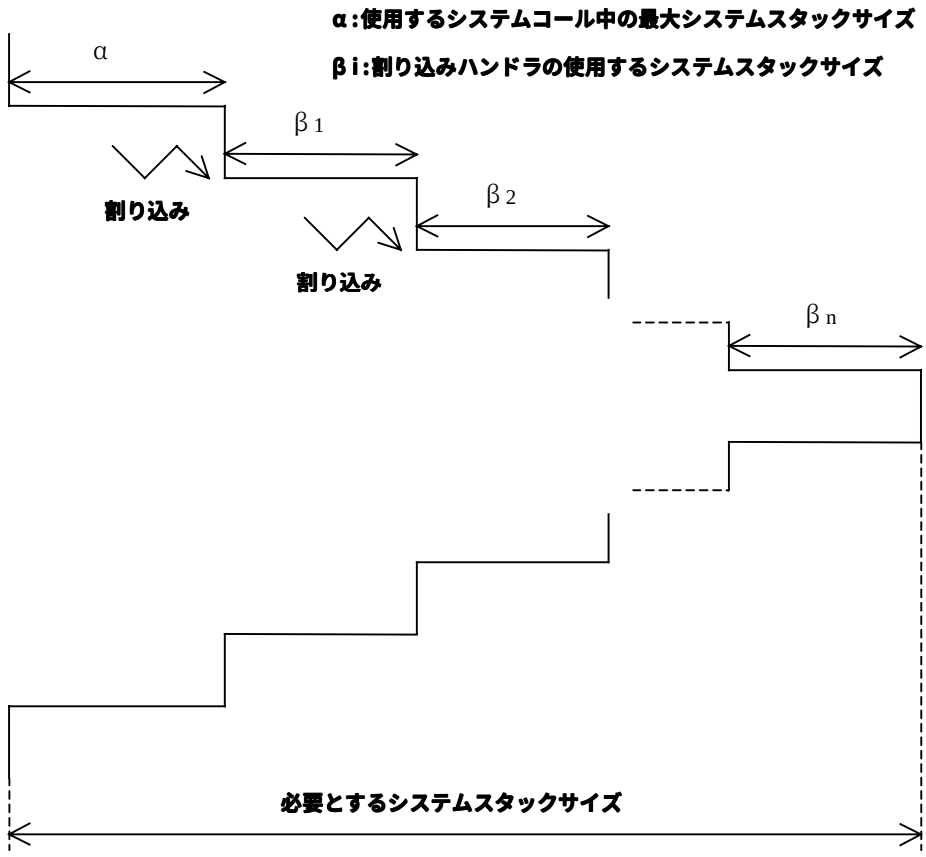


図 1.4:システムスタックサイズの算出方法

【割り込みハンドラの使用するスタックサイズβi】

システム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**

  STK ビューワ[8]をご使用下さい。

  STK ビューワ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。

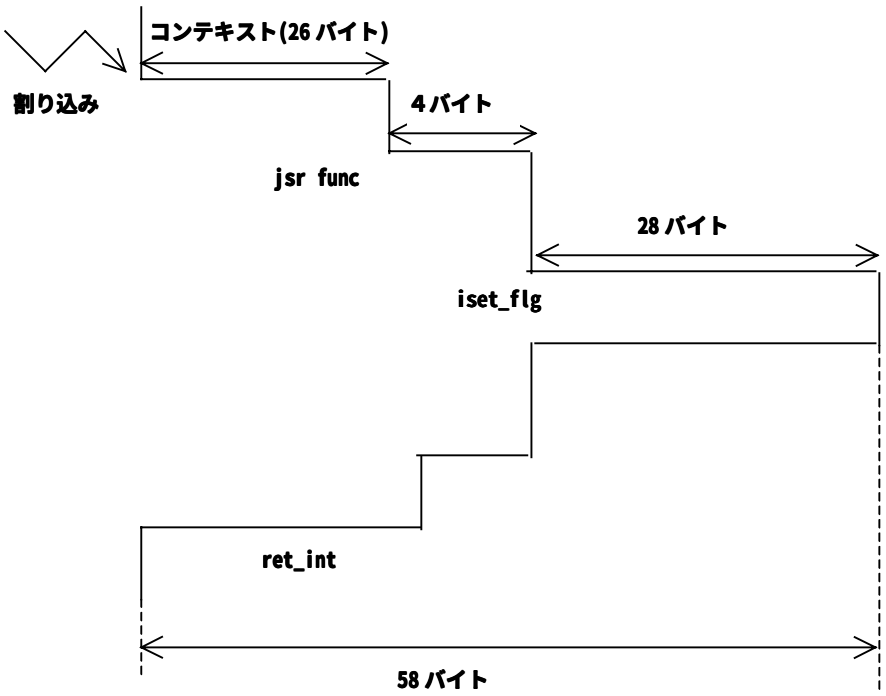
  STK ビューワの詳細な使用方法については、STK ビューワのマニュアルをご覧ください。

- **アセンブリ言語**

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 + ユーザー使用量 + システムコールの使用量

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 + ユーザー使用量

ユーザー使用量は、ユーザー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 26 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+6 (PC+FLG)バイト

図 1.5: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

[8] STK ビューワは、ルネサス製 C コンパイラ NC308WA に付属されているスタックを計算するためのツールです。

の大きいサイズです。

- **38 + 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**
- **36+ 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**
- **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**

  STK ビューワ[9]をご使用下さい。

  STK ビューワ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。

  STK ビューワの詳細な使用方法については、STK ビューワ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 + ユーザー使用量 + システムコールの使用量

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 = 26 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。

[9] STK ビューワは、ルネサス製 C コンパイラ NC308WA に付属されているスタックを計算するためのツールです。

第2章　システムコールリファレンス

2.1. タスク管理機能

2.1.1. sta_tsk(Start Task)

【システムコール名】

sta_tsk　　→　　タスクを起動します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER sta_tsk (tskid, stacd);
```

《引数》

```
ID          tskid;   起動するタスク ID 番号
INT         stacd;   タスク起動コード
```

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
sta_tsk  tskid, stacd
```

《引数》

```
tskid       [---*]   起動するタスクの ID 番号
stacd       [--**]   タスク起動コード
```

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	タスク起動コード
A0	起動するタスクの ID 番号

【エラーコード】

```
E_OK        00000H(-H'0000): 正常終了
E_OBJ       0FFC1H(-H'003f): オブジェクトの状態が不正
```

【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(RUN)状態へ移行します。

起動コード stacd は、16 ビットです。C 言語の場合、stacd は起動タスクに引数として渡されます。また、アセンブリ言語の場合、stacd は起動タスクの R0 レジスタに格納されます。

本システムコールは、指定したタスクが休止(DORMANT)状態であるときのみ有効です。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、システムコール発行タスクにエラーE_OBJ を返します。

本システムコールの発行により、コンフィグレーションファイルで定義した開始アドレスと優先度でタスクが起動されます。[10]

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[11]。

●コンフィグレーションファイルで設定した開始アドレスからスタートします。

●起床要求カウントは 0(ゼロ)クリアします。

●優先度はコンフィグレーションファイルで設定した初期優先度となります。

●PC、R0(stacd)、SB、FLG レジスタ以外のレジスタの初期値は不定です。

レジスタ	初期値
R0	タスク起動コード
R1	不定
R2	不定
R3	不定
A0	不定
A1	不定
SB	__SB__
FB	不定
FLG	C0H
PC	タスク開始アドレス

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラからは発行する場合は、ista_tsk システムコールを使用してください。

[10] ただし、指定したタスクがただちに動作するとは限りません。あくまで動作するかどうかはその時のレディキューの状態により決定されます。

[11] すなわち、タスクは完全にリセット状態からスタートします。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
    sta_tsk( ID_task2, stacd );
        :
}
void task2(int msg)
{
    if(msg == 0)
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task,task2
task:
    sta_tsk #ID_task2, msg
        :
task2:
    cmp.w       #0,R0
        :
```

2.1.2. ista_tsk(Start Task)

【システムコール名】

ista_tsk　　→　　タスクを起動します（ハンドラ専用）

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER ista_tsk (tskid, stacd);
```

《引数》

ID	tskid;	起動するタスク ID 番号
INT	stacd;	タスク起動コード

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
ista_tsk tskid, stacd
```

《引数》

tskid	[---*]	起動するタスクの ID 番号
stacd	[--**]	タスク起動コード

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	タスク起動コード
A0	起動するタスクの ID 番号

【エラーコード】

E_OK	00000H(-H'0000): 正常終了
E_OBJ	0FFC1H(-H'003f): オブジェクトの状態が不正

【機能説明】

sta_tsk システムコールと同じ機能を割込みハンドラ、周期起動ハンドラ、アラームハンドラから利用する場合に、このシステムコールを使用してください。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void inthand()
{
    ista_tsk( ID_task2, stacd );
        :
}
void task2(int msg)
{
    if(msg == 0)
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        intr
intr:
    ista_tsk #ID_task2, msg
        :
    ret_int
```

2.1.3. ext_tsk(Exit Task)

【システムコール名】

ext_tsk　　→　　自タスクを終了します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
void ext_tsk ();
```

《引数》

なし

《戻り値》

本システムコールを発行したタスクには戻りません。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
ext_tsk
```

《引数》

なし

《レジスタ設定》

本システムコールを発行したタスクには戻りません。

【エラーコード】

本システムコールを発行したタスクには戻りません。

【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(RUN)状態から休止(DORMANT)状態へ移行します。一度タスクが終了すると再度 sta_tsk、ista_tsk システムコールにより起動するまで動作することはありません。再度 sta_tsk、ista_tsk システムコールにより起動したタスクは、リセットされたように動作します。

本システム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。

loc_cpu または dis_dsp を発行した場合、必ず、unl_cpu または ena_dsp を発行してからタスクを終了(ext_tsk の発行)してください。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラから発行した場合、正常に動作しません。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
void task(void)
{
        :
    ext_tsk();
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    ext_tsk
```

2.1.4. ter_tsk(Terminate Task)

【システムコール名】

ter_tsk　　→　　他タスクを強制的に終了します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER ter_tsk (tskid);
```

《引数》

ID　　tskid;　強制終了するタスク ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
ter_tsk  tskid
```

《引数》

tskid　　[---*]　強制終了するタスクの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	強制終了するタスクの ID 番号

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了
E_OBJ　　0FFC1H(-H'003f): オブジェクトの状態が不正

【機能説明】

tskid で示されたタスクを、強制的に終了させます。

このシステムコールで自タスクを指定することはできません。したがって、自タスクを終了する場合は ext_tsk システムコールを使用してください。

指定したタスクが待ち状態に入り、何らかの待ち行列[12]につながれていた場合には、このシステム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、システムコールの戻り値としてエラー E_OBJ を返します。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよび、アラームハンドラから発行した場合は、正常に動作しません。

[12] タイムアウト待ち行列、イベントフラグ待ち行列など

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
        :
    ter_tsk( ID_main );
        :
    ext_tsk();
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    ter_tsk #ID_task2
        :
    .GLB        task2
task2:
        :
```

2.1.5. dis_dsp(Disable Dispatch)

【システムコール名】

dis_dsp　　→　　タスクのディスパッチを禁止します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER dis_dsp ();
```

《引数》

なし

《戻り値》

関数の戻り値としてエラーコードを返す。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
dis_dsp
```

《引数》

なし

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	--

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了

【機能説明】

タスクのディスパッチを禁止します。

本システムコール実行後、ena_dsp システムコールが実行されるまでの間はタスクのディスパッチは禁止状態となります。したがって、割り込みハンドラあるいは dis_dsp を実行したタスクから発行されたシステムコールによって、dis_dsp を実行したタスクより高い優先度を持つタスクが実行可能(READY)状態となっても、そのタスクにはディスパッチされません。すなわち、優先度の高いタスクへのディスパッチは、ディスパッチ禁止状態が終了するまで遅延されます。

ただし、外部割り込みは禁止しないので、ディスパッチ禁止状態であっても割り込みハンドラは起動されます。

既に、ディスパッチ禁止状態にあるタスクが dis_dsp を発行した場合は、ディスパッチ禁止状態がそのまま継続するだけで、エラーにはなりません。ただし、dis_dsp を複数回、発行しても、その後、ena_dsp を 1 回発行するだけでディスパッチ禁止状態が解除されます。

loc_cpu または dis_dsp を発行した場合、必ず、unl_cpu または ena_dsp を発行してからタスクを終了(ext_tsk の発行)してください。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラから発行した場合は、正常に動作しません。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
        :
    dis_dsp();
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    dis_dsp
        :
```

2.1.6. ena_dsp(Enable Dispatch)

【システムコール名】

ena_dsp　　→　　タスクのディスパッチを許可します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER ena_dsp();
```

《引数》

なし

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
ena_dsp
```

《引数》

なし

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	--

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了

【機能説明】

タスクのディスパッチを許可します。すなわち、dis_dsp によって設定されていたディスパッチ禁止状態を解除し、スケジューラを動作させます。ディスパッチ禁止状態でないタスクが ena_dsp を発行した場合は、ディスパッチを許可した状態がそのまま継続するだけで、エラーとはなりません。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラから発行した場合は、正常に動作しません。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
        :
    ena_dsp();
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    ena_dsp
        :
```

2.1.7. chg_pri(Change Task Priority)

【システムコール名】

chg_pri　　→　　タスクの優先度を変更します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER chg_pri (tskid,tskpri);
```

《引数》

ID	tskid;	優先度を変更するタスクの ID 番号
PRI	tskpri;	変更する優先度

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
chg_pri  tskid, tskpri
```

《引数》

tskid	[---*]	優先度を変更するタスクの ID 番号
tskpri	[--**]	変更する優先度

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	変更する優先度
A0	優先度を変更するタスクの ID 番号 （TSK_SELF を含む）

【エラーコード】

E_OK	00000H(-H'0000): 正常終了
E_OBJ	0FFC1H(-H'003f): オブジェクトの状態が不正

【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ます。また、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。レディキューにつながれているタスク(RUN 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システム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。[13]

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グ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、コンフィグレーションファイルで

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

の場合は指定できる優先度の範囲は 1 から 13 までです。[14]

tskid=TSK_SELF=0 を指定すると自タスクの指定になります。

本システムコールで休止(DORMANT)状態にあるタスクの優先度を変更することはできません。したがって、tskid で示された対象タスクが休止(DORMANT)状態にある場合は、システムコールの戻り値としてエラーE_OBJ を返します。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラから発行する場合は ichg_pri システムコールを使用してください。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
        :
    chg_pri (ID_task2, 2);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    chg_pri #ID_task2,#2
        :
```

[13] したがって、自タスクを対象として現在と同じ優先度で本システムコールを発行することにより、実行権の放棄を行なうことができます。

[14] 優先度の低いタスクへの切り替えは、処理時間、割り込み禁止時間が多く必要になります。したがって、優先度の範囲を必要最小現にすることを推奨します。

2.1.8. ichg_pri(Change Task Priority)

【システムコール名】

ichg_pri　　→　　タスクの優先度を変更します（ハンドラ専用）

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER ichg_pri (tskid,tskpri);
```

《引数》

ID	tskid;	優先度を変更するタスクの ID 番号
PRI	tskpri;	変更する優先度

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
ichg_pri tskid, tskpri
```

《引数》

tskid	[---*]	優先度を変更するタスクの ID 番号
tskpri	[--**]	変更する優先度

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	変更する優先度
A0	優先度を変更するタスクの ID 番号

【エラーコード】

E_OK	00000H(-H'0000): 正常終了
E_OBJ	0FFC1H(-H'003f): オブジェクトの状態が不正

【機能説明】

chg_pri システムコールと同じ機能を割込みハンドラ、周期起動ハンドラ、アラームハンドラから利用する場合に、このシステムコールを使用してください。

本システムコールでは、tskid=TSK_SELF=0 による自タスク指定はできません。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void inthand()
{
        :
    ichg_pri(ID_main, 2);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        intr
intr:
        :
    ichg_pri #ID_task2, 2
        :
    ret_int
```

2.1.9. rot_rdq(Rotate Ready Queue)

【システムコール名】

rot_rdq　　→　　タスクのレディキューを回転します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER rot_rdq (tskpri);
```

《引数》

PRI　　tskpri; 回転するレディキューの優先度

《戻り値》

関数の戻り値は常に E_OK を返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
rot_rdq  tskpri
```

《引数》

tskpri　　[--**]　回転するレディキューの優先度

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	回転するレディキューの優先度 (TPRI_RUN 含む)
A0	--

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了

【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 2.1**に示します。

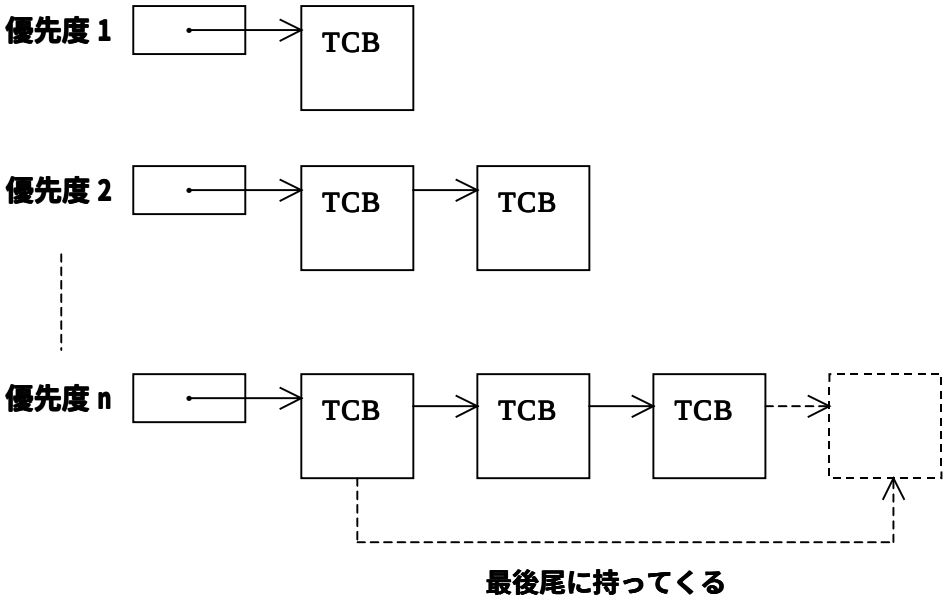


図 2.1:rot_rdq システムコールによるレディキューの操作

このシステムコールを一定時間間隔で発行することにより、ラウンドロビンスケジューリングをおこなうことができます。

tskpri=TPRI_RUN=0 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。

また、本システム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。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラから発行する場合は irot_rdq システムコールを使用してください。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
void task()
{
        :
    rot_rdq(2);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    rot_rdq #2
        :
```

2.1.10. irot_rdq(Rotate Ready Queue)

【システムコール名】

irot_rdq　　→　　タスクのレディキューを回転します（ハンドラ専用）

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER irot_rdq (tskpri);
```

《引数》

PRI　　tskpri;　優先度

《戻り値》

関数の戻り値は常に E_OK を返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
irot_rdq tskpri
```

《引数》

tskpri　　[--**]　回転するレディキューの優先度

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	回転するレディキューの優先度 (TPRI_RUN 含む)
A0	--

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了

【機能説明】

rot_rdq システムコールと同じ機能を割り込みハンドラ、周期起動ハンドラ、アラームハンドラから利用する場合に、このシステムコールを使用してください。

irot_rdq(tskpri=TPRI_RUN)を発行した場合、割り込みハンドラが発生した時に実行していたタスクが持つ優先度のレディキューが回転します。

周期起動ハンドラから本システムコールを発行することによりラウンドロビンスケジューリングができます。

【使用例】

この例では周期起動ハンドラにより周期的に優先度 2 のレディキューを回転することによりラウンドロビンスケジューリングを実現しています。

《 C 言語の使用例 》

```
#include <mr308.h>
void cyc()
{
        :
    irot_rdq(2);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        cyc
cyc:
        :
    irot_rdq #2
        :
```

2.1.11. rel_wai(Release Task Wait)

【システムコール名】

rel_wai　　→　　タスクの待ち状態を強制解除します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER rel_wai (tskid);
```

《引数》

ID　　tskid;　待ち状態を強制解除するタスクの ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
rel_wai  tskid
```

《引数》

tskid　　[---*]　待ち状態を強制解除するタスクの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	待ち状態を強制解除するタスクの ID 番号

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了
E_OBJ　　0FFC1H(-H'003f): オブジェクトの状態が不正

【機能説明】

tskid で示されたタスクの待ち状態(強制待ち(SUSPEND)状態を除く)を、強制的に解除します。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[15]につながれていた場合には、本システムコールの実行によってその待ち行列から削除されます。

対象タスクが待ち状態にない場合は、システムコール発行タスクにエラーE_OBJ を返します。

本システムコールは、自タスクを指定できません。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラから発行する場合は、irel_wai システムコールを使用してください。

[15] タイムアウト待ち行列、イベントフラグ待ち行列など。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "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    mr308.inc
    .GLB        task
task:
        :
    rel_wai #ID_main
        :
```

2.1.12. irel_wai(Release Task Wait)

【システムコール名】

irel_wai　　→　　タスクの待ち状態を強制解除します（ハンドラ専用）

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER irel_wai (tskid);
```

《引数》

ID　　tskid;　タスク ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
irel_wai tskid
```

《引数》

tskid　　[---*]　待ち状態を強制解除するタスクの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	待ち状態を強制解除するタスクの ID 番号

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了
E_OBJ　　0FFC1H(-H'003f): オブジェクトの状態が不正

【機能説明】

rel_wai システムコールの機能を割込みハンドラ、周期起動ハンドラ、アラームハンドラから利用する場合に、このシステムコールを使用してください。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void inthand()
{
        :
    if( irel_wai( ID_main ) != E_OK )
        error("Can't irel_wai task(2)¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        intr
intr:
        :
    irel_wai #ID_main
        :
    ret_int
```

2.1.13. get_tid(Get Self Task ID)

【システムコール名】

get_tid　　→　　自タスクの ID を得ます

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER get_tid (p_tskid);
```

《引数》

ID	*p_tskid;	タスク ID 番号を格納する領域の先頭アドレス

《戻り値》

関数の戻り値は常に E_OK を返します。
p_tskid の示す領域に、自タスクの ID 番号が設定されます。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
get_tid
```

《引数》

なし

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	自タスクの ID 番号

【エラーコード】

E_OK	00000H(-H'0000): 正常終了

【機能説明】

自タスクの ID 番号を獲得します。
割り込みハンドラ、周期起動ハンドラ、アラームハンドラから発行した場合は、FALSE=0(ゼロ)を返します。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
void task()
{
    ID tskid;
        :
    get_tid(&tskid);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    get_tid
        :
```

2.1.14. ref_tsk(Refer Task Status)

【システムコール名】

ref_tsk　　→　　タスクの状態を参照します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER ref_tsk (pk_rtsk,tskid);
```

《引数》

T_RTSK	*pk_rtsk;	タスク状態を返す構造体の先頭アドレス
ID	tskid;	状態を参照するタスクの ID 番号

《戻り値》

関数の戻り値にエラーコードを返します。
pk_rtsk の指す構造体には、以下の情報が返されます。

```
typedef struct t_rtsk {
    VP      exinf;   /* 拡張情報 */
    PRI     tskpri;  /*現在の優先度 */
    UINT    tskstat; /*タスク状態 */
    UINT    tskwait; /*タスクの待ち要因 */
    ID      wid;     /*タスクの待ちオブジェクト ID */
} T_RTSK;
```

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
ref_tsk  tskid, pk_rtsk
```

《引数》

taskid	[---*]	状態を参照するタスクの ID 番号
pk_rtsk	[-***]	タスクの状態を返すパケットアドレス

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	状態を参照するタスクの ID 番号(TSK_SELF 含む)
A1	タスクの状態を返すパケットの先頭アドレス

pk_rtsk の指す領域には、以下の情報が返されます。

オフセット

+0	exinf	拡張情報
+4	tskpri	現在の優先度
+6	tskstat	タスクの状態
+8	tskwait	タスクの待ち要因
+10	wid	タスクの待ちオブジェクト ID

【エラーコード】

E_OK	00000H(-H'0000): 正常終了

【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リターン値として返します。

●exinf

exinf には、指定したタスクの拡張情報を返します。
MR308 では、常に不定です。

●tskpri

tskpri には、指定したタスクの優先度値を返します。

●tskstat

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	(0001H)	実行(RUN)状態
TTS_RDY	(0002H)	実行可能(READY)状態
TTS_WAI	(0004H)	待ち(WAIT)状態
TTS_SUS	(0008H)	強制待ち(SUSPEND)状態
TTS_WAS	(000CH)	二重待ち(WAIT-SUSPEND)状態
TTS_DMT	(0010H)	休止(DORMANT)状態

●tskwait

tskwait には、対象タスクが待ち状態であるならば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。

TTW_SLP	(0001H)	slp_tsk、tslp_tsk による待ち
TTW_DLY	(0002H)	dly_tsk による待ち
TTW_FLG	(0010H)	wai_flg、twai_flg による待ち
TTW_SEM	(0020H)	wai_sem、twai_sem による待ち
TTW_MBX	(0040H)	rcv_msg、trcv_msg による待ち

●wid

wid には、対象タスクが待ち状態となった場合、そのオブジェクトの ID 番号を返します。

タスクから tskid=TSK_SELF=0 によって、自タスクの指定はできますが、割り込みハンドラから tskid=TSK_SELF の指定はできません。

割り込みハンドラから、割り込まれたタスクを対象とした ref_tsk を発行した場合は、tskstat には、RUN 状態(TTS_RUN)が返されます。

本システムコールはタスク、ハンドラのどちらからでも発行できます。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
    T_RTSK rtsk;
        :
    ref_tsk( &rtsk, ID_main );
        :
}
```

《 アセンブリ言語の使用例 》

```
rtsk:  .blkb   10

    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    ref_tsk #ID_task2, #rtsk
        :
```

2.2. タスク付属同期機能

2.2.1. sus_tsk(Suspend Task)

【システムコール名】

sus_tsk　　→　　タスクを強制待ち状態へ移行します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER sus_tsk (tskid);
```

《引数》

ID　　tskid;　強制待ちするタスクの ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
sus_tsk  tskid
```

《引数》

tskid　　[---*]　強制待ちにするタスクの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	強制待ちにするタスクの ID 番号

【エラーコード】

E_OK	00000H(-H'0000): 正常終了
E_QOVR	0FFB7H(-H'0049): キューイングまたはネストのオーバーフロー
E_OBJ	0FFC1H(-H'003f): オブジェクトの状態が不正

【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(SUSPEND)状態へ移行します。

強制待ち状態は、rsm_tsk システムコールの発行によって解除されます。

tskid で示された対象タスクが休止(DORMANT)状態にある場合は、システムコールの戻り値としてエラーE_OBJ を返します。

本システムコールによる強制待ち要求のネストは行いません。従って、tskid で示された対象タスクが強制待ち(SUSPEND)状態にある場合は、システムコールの戻り値としてエラーE_QOVR を返します。

本システムコールで自タスクを指定することはできません。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよび、アラームハンドラから発行する場合は isus_tsk システムコールを使用してください。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
        :
    if( sus_tsk( ID_main ) != E_OK )
        printf("Can't suspend task main()¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    sus_tsk #ID_task2
        :
```

2.2.2. isus_tsk(Suspend Task)

【システムコール名】

isus_tsk　　→　　タスクを強制待ち状態へ移行します（ハンドラ専用）

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER isus_tsk (tskid);
```

《引数》

ID　　tskid;　強制待ちするタスクの ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
isus_tsk tskid
```

《引数》

tskid　　[---*]　強制待ちにするタスクの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	強制待ちにするタスクの ID 番号

【エラーコード】

```
E_OK        00000H(-H'0000): 正常終了
E_QOVR      0FFB7H(-H'0049): キューイングまたはネストのオーバーフロー
E_OBJ       0FFC1H(-H'003f): オブジェクトの状態が不正
```

【機能説明】

sus_tsk システムコールと同じ機能を割込みハンドラ、周期起動ハンドラ、アラームハンドラから利用する場合に、このシステムコールを使用してください。

なお、ハンドラからシステムコールを発行するので全てのタスク ID を指定することができます。すなわち、被割り込みタスクもサスペンドすることができます。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void inthand()
{
        :
    if( isus_tsk( ID_main ) != E_OK )
        printf("Can't suspend main()¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        intr
intr:
        :
    isus_tsk #ID_main
        :
    ret_int
```

2.2.3. rsm_tsk(Resume Task)

【システムコール名】

rsm_tsk　　→　　強制待ち状態のタスクを再開します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER rsm_tsk (tskid);
```

《引数》

ID　　tskid;　タスク ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
rsm_tsk  tskid
```

《引数》

tskid　　[---*]　強制待ちを解除するタスクの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	強制待ちを解除するタスクの ID 番号

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了
E_OBJ　　0FFC1H(-H'003f): オブジェクトの状態が不正

【機能説明】

tskid で示されたタスクが sus_tsk システムコールによって中断されている場合、対象タスクの強制待ち状態を解除します。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。

対象タスクが強制待ち(SUSPEND)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システムコール発行タスクにエラーE_OBJ を返します。

本システムコールは強制待ち(SUSPEND)状態、または二重待ち(WAIT-SUSPEND)状態のタスクを対象としているので、自タスクを指定することはできません。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよび、アラームハンドラから発行する場合は irsm_tsk システムコールを使用してください。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
        :
    if( rsm_tsk( ID_main ) != E_OK )
        printf("Can't resume main()¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    rsm_tsk #ID_task2
        :
```

2.2.4. irsm_tsk(Resume Task)

【システムコール名】

irsm_tsk　　→　　強制待ち状態のタスクを再開します（ハンドラ専用）

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER irsm_tsk (tskid);
```

《引数》

ID　　tskid;　タスク ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
irsm_tsk tskid
```

《引数》

tskid　　[---*]　強制待ちを解除するタスクの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	強制待ちを解除するタスクの ID 番号

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了
E_OBJ　　0FFC1H(-H'003f): オブジェクトの状態が不正

【機能説明】

rsm_tsk システムコールと同じ機能を割込みハンドラ、周期起動ハンドラ、アラームハンドラから利用する場合に、このシステムコールを使用してください。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void inthand()
{
        :
    irsm_tsk( ID_main );
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        intr
intr:
    :
irsm_tsk #ID_main
    :
```

2.2.5. slp_tsk(Sleep Task)

【システムコール名】

slp_tsk　　→　　タスクを待ち状態へ移行します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER slp_tsk ();
```

《引数》

なし

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
slp_tsk
```

《引数》

なし

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	--

【エラーコード】

```
E_OK        00000H(-H'0000): 正常終了
E_RLWAI     0FFAAH(-H'0056): 待ち状態強制解除
```

【機能説明】

自タスクを実行(RUN)状態から起床待ち状態へ移行します。本システムコールによる待ち状態は、このタスクを対象として発行されたタスク起床のシステムコール[16]または待ち状態の強制解除のシステムコール[17]により解除されます。前者の場合エラーコードとして E_OK が、後者の場合エラーコードとして E_RLWAI が返されます。

本システムコールにより待ち(WAIT)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(WAIT-SUSPEND)状態になります。この場合はタスク起床のシステム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。

本システムコールはタスクからのみ発行してください。

[16] wup_tsk、iwup_tsk システムコール
[17] rel_wai、irel_wai システムコール

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
        :
    if( slp_tsk() != E_OK )
        error("Forced wakeup¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE mr308.inc
    .GLB   task
task:
        :
    slp_tsk
        :
```

2.2.6. tslp_tsk(Sleep Task with Timeout)

【システムコール名】

tslp_tsk　　→　　タスクを一定時間待ち状態へ移行します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER tslp_tsk (tmout);
```

《引数》

TMO　　tmout;　タイムアウト値

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
tslp_tsk tmout
```

《引数》

tmout　　[--**]　タイムアウト値

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
R3	タイムアウト値

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了
E_TMOUT　　0FFABH(-H'0055): ポーリング失敗またはタイムアウト
E_RLWAI　　0FFAAH(-H'0056): 待ち状態強制解除

【機能説明】

自タスクを、指定した一定時間だけ実行(RUN)状態から待ち(WAIT)状態へ移行します。
本システム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システムコール[18]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システムコール[19]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 で指定した時間が経過した場合、tmout=0 で起床カウントが 0 の場合

  この時のエラーコードは、E_TMOUT が返ります。

tmout で指定する時間の単位は、コンフィグレーションファイルで指定したシステムクロックの単位時間になります。コンフィグレーションファイルで、システムクロックの単位時間を 10ms に設定し、

```
tslp_tsk(10);
```

と記述すれば、100ms 間、自タスクが実行(RUN)状態から待ち(WAIT)状態へ移行します。
tmout には、-1 から 7FFFH まで指定できます。なお、tmout=TMO_FEVR(-1)にした場合は、永久待ちの指定で、slp_tsk システムコールと同じ動作になります。
本システムコールはタスクからのみ発行してください。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
        :
    if( tslp_tsk( 10 ) != E_TMOUT )
        printf("Forced wakeup¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE mr308.inc
    .GLB task
task:
        :
    tslp_tsk #200
        :
```

[18] wup_tsk、iwup_tsk システムコール
[19] rel_wai、irel_wai システムコール

2.2.7. wup_tsk(Wakeup Task)

【システムコール名】

wup_tsk　　→　　待ち状態のタスクを起床します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER wup_tsk (tskid);
```

《引数》

ID　　tskid;　タスク ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
wup_tsk  tskid
```

《引数》

tskid　　[---*]　起床するタスクの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	起床するタスクの ID 番号

【エラーコード】

```
E_OK        00000H(-H'0000): 正常終了
E_QOVR      0FFB7H(-H'0049): キューイングまたはネストのオーバーフロー
E_OBJ       0FFC1H(-H'003f): オブジェクトの状態が不正
```

【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(WAIT)状態であれば、待ちを解除します。

また、tskid で指定したタスクが二重待ち(WAIT-SUSPEND)状態である時は、待ちのみを解除して強制待ち(SUSPEND)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、システムコール発行タスクにエラーE_OBJ を返します。

本システムコールは自タスクを指定することはできません。

slp_tsk あるいは tslp_tsk システムコール実行による待ち(WAIT)状態もしくは、二重待ち(WAIT-SUSPEND)状態にないタスクに対して本システム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 TCB[20]内にある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。[21]

起床要求カウントの最大値は 0x7FFF です。起床要求カウントが 0x7FFF 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 0x7FFF のままで本システムコールの発行タスクには、エラーコード E_QOVR を返します。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラから発行する場合は iwup_tsk システムコールを使用してください。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "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    mr308.inc
    .GLB        task
task:
        :
    wup_tsk #ID_task2
```

[20] タスク制御ブロック

[21] この起床要求カウントには、wup_tsk、iwup_tsk システムコールにより起床しようとした時に対象タスクが待ち(WAIT)状態または二重待ち(WAIT-SUSPEND)状態でないために起床要求を実現できなかった回数が記憶されます。起床要求カウントが 1 以上である時に、そのタスクが slp_tsk あるいは tslp_tsk システムコールにより待ち状態に入ろうとした場合はその起床要求カウントを 1 つ減らします。この時、そのタスクは待ち状態にはなりません。slp_tsk あるいは tslp_tsk システムコールにより待ち (WAIT) 状態に入るのは起床要求カウントが 0 の時のみです。

2.2.8. iwup_tsk(Wakeup Task)

【システムコール名】

iwup_tsk　　→　　待ち状態のタスクを起床します（ハンドラ専用）

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER iwup_tsk (tskid);
```

《引数》

ID　　tskid;　タスク ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
iwup_tsk tskid
```

《引数》

tskid　　[---*]　起床するタスクの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	起床するタスクの ID 番号

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了
E_QOVR　　0FFB7H(-H'0049): キューイングまたはネストのオーバーフロー
E_OBJ　　0FFC1H(-H'003f): オブジェクトの状態が不正

【機能説明】

wup_tsk システムコールと同じ機能を割込みハンドラ、周期起動ハンドラ、アラームハンドラから利用する場合に、このシステムコールを使用してください。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void inthand()
{
    if( iwup_tsk( ID_main ) != E_OK )
        printf("Can't wakeup main()¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        intr
intr:
        :
    iwup_tsk #ID_main
        :
    ret_int
```

2.2.9. can_wup(Cancel Wakeup Task)

【システムコール名】

can_wup　　→　　タスクの起床要求を無効にします

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER can_wup (p_wupcnt,tskid);
```

《引数》

INT	*p_wupcnt;	無効になった起床要求回数を格納する領域の先頭アドレス
ID	tskid;	タスク ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。
変数 wupcnt に無効になった起床要求回数が設定されます。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
can_wup  tskid
```

《引数》

tskid	[---*]	起床要求を無効にするタスクの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	無効になった起床要求回数
A0	起床要求を無効にするタスクの ID 番号 (TSK_SELF 含む)

【エラーコード】

E_OK	00000H(-H'0000): 正常終了
E_OBJ	0FFC1H(-H'003f): オブジェクトの状態が不正

【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、本システムコール発行以前に wup_tsk、iwup_tsk システムコールにより起床しようとした時に、対象タスクが待ち(WAIT)状態もしくは二重待ち(WAIT-SUSPEND)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、本システム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、システムコール発行タスクにエラー E_OBJ を返します。

本システムコールは、タスクから発行する場合に限り、tskid=TSK_SELF=0 で自タスクの指定ができます。

エラーコードが E_OK 以外の場合は、リターンパラメータ(*p_wupcnt)は不定です。

本システムコールはタスク、ハンドラのどちらからでも発行できます。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
void task()
{
    INT wupcnt;
        :
    if( can_wup(&wupcnt,ID_main) != E_OK )
        printf("Can't cancle wakeup main() ¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    can_wup #ID_task2
        :
```

2.3. 同期・通信機能

2.3.1. set_flg(Set Eventflag)

【システムコール名】

set_flg　　→　　イベントフラグをセットします

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER set_flg (flgid, setptn);
```

《引数》

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

《戻り値》

関数の戻り値として常に E_OK を返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
set_flg  flgid, setptn
```

《引数》

```
flgid       [---*]  セットするイベントフラグの ID 番号
setptn      [--**]  セットするビットパターン
```

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	セットするビットパターン
A0	セットするイベントフラグの ID 番号

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了

【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 システムコールによってそのイベントフラグを待っていたタスクの待ち解除の条件を満たすようになれば、そのタスクの待ちを解除します。

イベントフラグは、同一フラグに対する複数タスクの待ちが可能ですので、一回の set_flg システムコール発行で、その複数タスクが同時に待ち解除となります。ただし、待ち行列のなかのタスクがクリア指定でイベントフラグがセットされるのを待っていた場合はそのタスクまでが待ち解除になります。

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。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラから発行する場合は iset_flg システムコールを使用してください。

【使用例】

システムコール発行前のイベントフラグのパターンが 0xf であった場合、システムコール発行後のパターンは 0xff となります。

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task(void)
{
        :
    set_flg( ID_flg,(UINT)0xf0 );
        :
    ext_tsk();
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    set_flg #ID_flg,#0f0H
        :
    ext_tsk
```

2.3.2. iset_flg(Set Eventflag)

【システムコール名】

iset_flg　　→　　イベントフラグをセットします (ハンドラ専用)

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER iset_flg (flgid, setptn);
```

《引数》

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

《戻り値》

関数の戻り値として常に E_OK を返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
iset_flg flgid, setptn
```

《引数》

flgid	[---*]	セットするイベントフラグの ID 番号
setptn	[--**]	セットするビットパターン

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	セットするビットパターン
A0	セットするイベントフラグの ID 番号

【エラーコード】

E_OK	00000H(-H'0000): 正常終了

【機能説明】

set_flg システムコールと同じ機能を割り込みハンドラ、周期起動ハンドラ、アラームハンドラから利用する場合に、このシステムコールを使用してください。

【使用例】

システムコール発行前のイベントフラグのパターンが 0xf であった場合、システムコール発行後のパターンは 0xff となります。

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void inthand(void)
{
        :
    iset_flg( ID_flg, (UINT)0xf0);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        intr
intr:
        :
    iset_flg  #ID_flg,#0f0H
        :
    ret_int
```

2.3.3. clr_flg(Clear Eventflag)

【システムコール名】

clr_flg　　→　　イベントフラグをクリアします

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER clr_flg (flgid, clrptn);
```

《引数》

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

《戻り値》

関数の戻り値として常に E_OK を返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
clr_flg  flgid, clrptn
```

《引数》

flgid	[---*]	クリアするイベントフラグの ID 番号
clrptn	[--**]	クリアするビットパターン

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	クリアするビットパターン
A0	クリアするイベントフラグの ID 番号

【エラーコード】

E_OK	00000H(-H'0000): 正常終了

【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値に対して、clrptn の値で論理積(AND)をとります。

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。

本システムコールはタスク、ハンドラのどちらからでも発行できます。

【使用例】

システムコール発行前のイベントフラグのパターンが 0xff であったとすると、システムコール発行後のパターンは 0xf0 となります。

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task(void)
{
       :
   clr_flg( ID_flg, (UINT)0xfff0 );
       :
}
```

《 アセンブリ言語の使用例 》

```
   .INCLUDE    mr308.inc
   .GLB        task
task:
       :
   clr_flg #ID_flg,#0fff0H
       :
```

2.3.4. wai_flg(Wait Eventflag)

【システムコール名】

wai_flg　　→　　イベントフラグを待ちます

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER wai_flg (p_flgptn, flgid, waiptn, wfmode);
```

《引数》

UINT	*p_flgptn;	待ち解除時のビットパターンを返す領域の先頭アドレス
ID	flgid;	イベントフラグ ID 番号
UINT	waiptn;	待ちビットパターン
UINT	wfmode;	待ちモード

《戻り値》

関数の戻り値としてエラーコードを返します。
p_flgptn の指す領域に、待ち解除時のビットパターンが設定されます。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
wai_flg  flgid, waiptn, wfmode
```

《引数》

flgid	[---*]	イベントフラグの ID 番号
waiptn	[--**]	待ちビットパターン
wfmode	[--**]	待ちモード

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	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
A0	待つイベントフラグの ID 番号

【エラーコード】

E_OK	00000H(-H'0000): 正常終了
E_RLWAI	0FFAAH(-H'0056): 待ち状態強制解除

【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。

waiptn には、待ちビットパターンを指定して下さい。なお、waiptn には、0(ゼロ)は指定できません。0 を指定した場合、本システムコールは、なにもせずに戻りますが、μITRON 仕様では、エラーE_PAR が返りますので、他のリアルタイムとの互換性が取れなくなります。

wfmode では、次のような指定を行います。

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

TWF_ANDW AND 待ち
TWF_ORW　OR 待ち
TWF_CLR　クリア指定

すなわち、以下のような意味を持ちます。

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_ANDW+TWF_CLR	waiptn で指定したビットの AND 待ち条件解除が満たされてタスクが待ち解除となった場合、イベントフラグの値を 0 にクリア（全てのビットを 0 クリア）する。
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)。
TWF_ORW+TWF_CLR	waiptn で指定したビットの OR 待ち解除条件が満たされてタスクが待ち解除となった場合、イベントフラグの値を 0 にクリア（全てのビットを 0 クリア）する。

flgptn には、本システムコールにより待ち状態が解除される時のイベントフラグの値 (クリア指定の場合は、イベントフラグがクリアされる前の値)が格納されます。flgptn に返る値は、待ち解除条件を満たす値になっています。

同一イベントフラグに対する複数タスクの待ちも可能です。この場合、一回の set_flg システムコール発行で複数のタスクが待ち解除となります。ただし、待ち行列中で待ち解除条件が満たされたタスクがクリア指定を行なっていた場合、そのタスクまでが待ち解除になります。

イベントフラグは以下に示す動作をするタスクの待ち行列を形成します。

●待ち行列の順番は、FIFO(ファーストインファーストアウト)です。

●待ち行列中にクリア指定のタスクがあれば、そのタスクが待ち解除時にフラグをクリアします。

●クリア指定をおこなっていたタスクよりも後ろの待ち行列にあったタスクは、既にクリアされた後のイベントフラグをもとに待ちを解除するか否かを決定するため、待ち解除とはなりません。

rel_wai、irel_wai システムコールによって待ち状態が強制解除された場合には、エラーコード E_RLWAI が返されます。

エラーコードが E_OK 以外の場合は、リターンパラメータ(*p_flgptn)は不定です。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラから発行した場合は、正常に動作しません。

【使用例】

この例ではフラグ名が flg2 のイベントフラグの指定したビットがセットされるのを待ちます。指定したビットがセットされたタスクは待ち状態が解除されます。

ここでは待ちモードとしてクリア指定をしているので、flg2 のイベントフラグはタスクの待ち状態が解除されると同時に 0 にクリアされます。

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
   UINT flgptn;
       :
   if(wai_flg(&flgptn,ID_flg2,(UINT)0x0ff0,TWF_ANDW+TWF_CLR)!=E_OK)
       error("Wait Released¥n”);
```

《 アセンブリ言語の使用例 》

```
   .INCLUDE    mr308.inc
   .GLB        task
task:
       :
   wai_flg #ID_flg2,#0ff0H,#(TWF_ANDW+TWF_CLR)
       :
```

2.3.5. twai_flg(Wait Eventflag with Timeout)

【システムコール名】

twai_flg　　→　　イベントフラグを待ちます（タイムアウトあり）

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER twai_flg (p_flgptn, flgid, waiptn, wfmode, tmout);
```

《引数》

UINT	*p_flptn;	待ち解除時のビットパターンを返す領域の先頭アドレス
ID	flgid;	イベントフラグ ID 番号
UINT	waiptn;	待ちビットパターン
UINT	wfmode;	待ちモード
TMO	tmout;	タイムアウト値

《戻り値》

関数の戻り値としてエラーコードを返します。
p_flgptn の指す領域に、待ち解除時のビットパターンが設定されます。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
twai_flg flgid, waiptn, wfmode, tmout
```

《引数》

flgid	[---*]	イベントフラグの ID 番号
waiptn	[--**]	待ちビットパターン
wfmode	[--**]	待ちモード
tmout	[--**]	タイムアウト値

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	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	タイムアウト値
A0	待つイベントフラグの ID 番号

【エラーコード】

E_OK	00000H(-H'0000): 正常終了
E_RLWAI	0FFAAH(-H'0056): 待ち状態強制解除
E_TMOUT	0FFABH(-H'0055): ポーリング失敗またはタイムアウト

【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。

本システムコールを発行したタスクは、イベントフラグ待ち行列とタイムアウト待ち行列の 2 つの待ち行列にタスクがつながれます。

本システム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。なお、待ち状態が解除されると本システムコールを発行したタスクはイベントフラグ待ち行列とタイムアウト待ち行列の 2 つの待ち行列からはずされ、レディキューに接続されます。

- 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 の時間が経過した場合

  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 システム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。

tmout には、-1 から 7FFFH まで指定できます。なお、tmout に TMO_POL(=0)を指定した場合は、pol_flg と同じ動作をします。また、TMO_FEVR(=-1)を指定した場合は、永久待ちの指定で、wai_flg システムコールと同じ動作になります。

wfmode の指定方法および各モードの意味については、wai_flg システムコールを参照してください。

エラーコードが E_OK 以外の場合は、リターンパラメータ(*p_flgptn)は不定です。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラから発行した場合は、正常に動作しません。

【使用例】

この例ではフラグ名が flg2 のイベントフラグの指定したビットがセットされるかあるいは、待ち時間 tmout が経過するのを待ちます。指定したビットがセットされるかあるいは待ち時間を経過した場合、待ち状態が解除されます。

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
    UINT flgptn;
        :
    if( twai_flg(&flgptn, ID_flg2,(UINT)0x0ff0, TWF_ANDW, 5) != E_OK )
        error("Wait Released¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    twai_flg #ID_flg2,#0ff0H,#(TWF_ANDW+TWF_CLR),#5
        :
```

2.3.6. pol_flg(Poll Eventflag)

【システムコール名】

pol_flg　　→　　イベントフラグを得ます（待ち無し）

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER pol_flg (p_flgptn, flgid, waiptn, wfmode);
```

《引数》

UINT	*p_flgptn;	待ち解除時のビットパターンを返す領域の先頭アドレス
ID	flgid;	調べるイベントフラグ ID 番号
UINT	waiptn;	待ちビットパターン
UINT	wfmode;	待ちモード

《戻り値》

p_flgptn の指す領域に、待ち解除時のビットパターンが設定されます。
戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
pol_flg  flgid, waiptn, wfmode
```

《引数》

flgid	[---*]	調べるイベントフラグの ID 番号
waiptn	[--**]	待ちビットパターン
wfmode	[--**]	待ちモード

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	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
A0	調べるイベントフラグの ID 番号

【エラーコード】

E_OK	00000H(-H'0000): 正常終了
E_TMOUT	0FFABH(-H'0055): ポーリング失敗またはタイムアウト

【機能説明】

flgid で示されるイベントフラグにおいて、waiptn で示される待ち解除ビットパターンが wfmode にしたがってセットされているかどうかを調べます。

対象イベントフラグが既に wfmode で示される待ち解除の条件を満たしている場合には、wai_flg と同様の処理をして(クリア指定がある場合はイベントフラグをクリアする)、正常終了します。

対象イベントフラグが wfmode で示される待ち解除の条件を満たしていない場合には、エラー E_TMOUT を返します。この場合、タスクは待ち状態にはなりません。また、この場合は、クリア指定があってもイベントフラグはクリアされません。

エラーコードが E_OK 以外の場合は、リターンパラメータ(*p_flgptn)は不定です。

本システムコールはタスク、ハンドラのどちらからでも発行できます。

【使用例】

この例ではフラグ名が flg2 のイベントフラグの指定したビットがセットされているかどうかを調べます。クリア指定が行なわれているので、イベントフラグが条件を満たしていれば、0 クリアされます。

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
   UINT flgptn;
       :
   if(pol_flg(&flgptn,ID_flg2,(UINT)0x0ff0,TWF_ORW+TWF_CLR)!=E_OK)
       printf("Not set EventFlag¥n”);
       :
}
```

《 アセンブリ言語の使用例 》

```
   .INCLUDE    mr308.inc
   .GLB        task
task:
   pol_flg #ID_flg2,#0ff0H,#(TWF_ORW+TWF_CLR)
       :
```

2.3.7. ref_flg(Refer Eventflag Status)

【システムコール名】

ref_flg　　→　　イベントフラグ状態を参照します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER ref_flg (pk_rflg, flgid);
```

《引数》

T_RFLG	*pk_rflg;	イベントフラグ状態を返す構造体の先頭アドレス
ID	flgid;	イベントフラグ ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。
pk_rflg が指す構造体には、以下のようなイベントフラグの状態が設定されます。

```
typedef struct t_rflg {
    VP      exinf;  /* 拡張情報 */
    BOOL_ID wtsk;   /* 待ちタスクの有無 */
    UINT    flgptn; /* イベントフラグのビットパターン */
} T_RFLG;
```

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
ref_flg  flgid, pk_rflg
```

《引数》

flgid	[---*]	状態を参照するイベントフラグ ID 番号
pk_rflg	[-***]	イベントフラグ状態を返すパケットアドレス

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	状態を参照するイベントフラグの ID 番号
A1	イベントフラグ状態を返すパケットの先頭アドレス

pk_rflg が指す領域には、以下の情報が返されます。

オフセット

+0	exinf	拡張情報
+4	wtsk	待ちタスクの有無
+6	flgptn	イベントフラグのビットパターン

【エラーコード】

E_OK	00000H(-H'0000): 正常終了

【機能説明】

flgid で示されたイベントフラグの以下の各種状態を返します。

●exinf

exinf には、拡張情報を返します。
MR308 では、常に不定です。

●wtskid

wtskid には、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 FALSE(0)を返します。

●flgptn

flgptn は現在のイベントフラグの値を返します。

本システムコールはタスク、ハンドラのどちらからでも発行できます。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
    T_RFLG rflg;
    ref_flg(&rflg, ID_flg );
        :
}
```

《 アセンブリ言語の使用例 》

```
rflg:  .blkb   8

    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    ref_flg     #ID_flg, #rflg
        :
```

2.4. 同期・通信機能(セマフォ)

2.4.1. sig_sem(Signal Semaphore)

【システムコール名】

sig_sem　　→　　セマフォ資源を返却します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER sig_sem (semid);
```

《引数》

ID　　semid;　セマフォ ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
sig_sem  semid
```

《引数》

semid　　[---*] 返却をおこなうセマフォの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	返却をおこなうセマフォの ID 番号

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了
E_QOVR　　0FFB7H(-H'0049): キューイングまたはネストのオーバーフロー

【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[22]。

セマフォの計数値の最大値は 0x7FFF (32767)です。セマフォの計数値が最大値 (0x7FFF)を越えて資源の返却(sig_sem、isig_sem システムコール)をおこなうとセマフォの計数値はそのままで、システムコール発行タスクにエラーE_QOVR を返します。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラから発行する場合は isig_sem システムコールを使用してください。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
        :
    if( sig_sem( ID_sem ) != E_OK )
        error("Overflow¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
    sig_sem     #ID_sem
        :
```

[22] このシステムコールの発行によって、計数値がコンフィグレーションファイルで定義したセマフォの初期値を越えてもエラーにはなりません。

2.4.2. isig_sem(Signal Semaphore)

【システムコール名】

isig_sem　　→　　セマフォ資源を返却します（ハンドラ専用）

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER isig_sem (semid);
```

《引数》

ID　　semid;　セマフォ ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
isig_sem semid
```

《引数》

semid　　[---*]　返却をおこなうセマフォの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	返却をおこなうセマフォの ID 番号

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了
E_QOVR　　0FFB7H(-H'0049): キューイングまたはネストのオーバーフロー

【機能説明】

sig_sem システムコールと同じ機能を割り込みハンドラ、周期起動ハンドラ、アラームハンドラから利用する場合に、このシステムコールを使用してください。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void inthand()
{
        :
    if( isig_sem( ID_sem ) != E_OK )
        error("Overflow¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        intr
intr:
    isig_sem    #ID_sem
        :
    ret_int
```

2.4.3. wai_sem(Wait on Semaphore)

【システムコール名】

wai_sem　　→　　セマフォ資源を獲得します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER wai_sem (semid);
```

《引数》

ID　　semid;　セマフォ ID 番号

《戻り値》

関数の戻り値としてエラーコードを返す。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
wai_sem  semid
```

《引数》

semid　　[---*]　獲得をおこなうセマフォの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	獲得をおこなうセマフォの ID 番号

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了
E_RLWAI　　0FFAAH(-H'0056): 待ち状態強制解除

【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おこないます。

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、システム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、計数値は変更せず、システムコール発行タスクをそのセマフォの待ち行列に FIFO[23]順でつながれます。

rel_wai、irel_wai システムコールによって待ち状態が強制解除された場合には、エラーE_RLWAI が返されます。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラから発行した場合は、正常に動作しません。

[23] ファーストインファーストアウト。すなわち wai_sem システムコールで待ちに入った順で sig_sem、isig_sem システムコールで待ちが解除されます。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
        :
    if( wai_sem( ID_sem ) != E_OK )
        printf("Forced wakeup¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
    wai_sem     #ID_sem
        :
```

2.4.4. twai_sem(Wait on Semaphore with Timeout)

【システムコール名】

twai_sem　　→　　セマフォ資源を獲得します（タイムアウトあり）

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER twai_sem (semid,tmout);
```

《引数》

ID	semid;	セマフォ ID 番号
TMO	tmout;	タイムアウト値

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
twai_sem semid, tmout
```

《引数》

semid	[---*]	獲得をおこなうセマフォ ID 番号
tmout	[--**]	タイムアウト値

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
R3	タイムアウト値
A0	獲得をおこなうセマフォの ID 番号

【エラーコード】

E_OK	00000H(-H'0000): 正常終了
E_TMOUT	0FFABH(-H'0055): ポーリング失敗またはタイムアウト
E_RLWAI	0FFAAH(-H'0056): 待ち状態強制解除

【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 だけ減じてシステム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、計数値は変更せず、システムコール発行タスクをそのセマフォ待ち行列とタイムアウト待ち行列につなぎます。
本システム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。なお、待ち状態が解除されると本システムコールを発行したタスクはセマフォ待ち行列とタイムアウト待ち行列の 2 つの待ち行列からはずされ、レディキューに接続されます。

- tmout の時間が経過する前に、sig_sem、isig_sem システムコールが発行され、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 の時間が経過した場合

  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 システム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。

tmout には、-1 から 0x7FFF まで指定できます。なお、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、preq_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 システムコールと同じ動作をします。
本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラから発行した場合は、正常に動作しません。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
       :
   if( twai_sem( ID_sem, 10 ) != E_OK )
       printf("Forced wakeup¥n”);
       :
}
```

《 アセンブリ言語の使用例 》

```
   .INCLUDE    mr308.inc
   .GLB        task
task:
       :
   twai_sem    #ID_sem,#10
       :
```

2.4.5. preq_sem(Poll and Request Semaphore)

【システムコール名】

preq_sem　　→　　セマフォ資源を獲得します（待ち無し）

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER preq_sem (semid);
```

《引数》

ID　　semid;　セマフォ ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
preq_sem semid
```

《引数》

semid　　[---*]　獲得をおこなうセマフォの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	獲得をおこなうセマフォの ID 番号

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了
E_TMOUT　　0FFABH(-H'0055): ポーリング失敗またはタイムアウト

【機能説明】

semid で示されたセマフォから資源を一つ獲得(待ち無し)します。
対象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、システム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 の場合には計数値は変更せず、システムコール発行タスクにエラー E_TMOUT を返し、本システムコールを終了します。
本システムコールはタスク、ハンドラのどちらからでも発行できます。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
        :
    if( preq_sem( ID_sem ) != E_OK )
        printf("No more resource¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    preq_sem    #ID_sem
        :
```

2.4.6. ref_sem(Refer Semaphore Status)

【システムコール名】

ref_sem　　→　　セマフォ状態を参照します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER ref_sem(pk_rsem,semid);
```

《引数》

T_RSEM	*pk_rsem;	セマフォ状態を返す構造体の先頭アドレス
ID	semid;	セマフォ ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。
pk_rsem が指す構造体には、以下の情報が返されます。

```
typedef struct t_rsem {
    VP      exinf;  /* 拡張情報 */
    BOOL_ID wtsk;   /* 待ちタスクの有無 */
    INT     semcnt; /* 現在のセマフォカウント値 */
} T_RSEM;
```

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
ref_sem  semid, pk_rsem
```

《引数》

semid	[---*]	セマフォ ID 番号
pk_rsem	[-***]	セマフォの状態を返すパケットの先頭アドレス

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	参照するセマフォの ID 番号
A1	セマフォの状態を返すパケットの先頭アドレス

pk_rsem が指す領域には、以下の情報が返されます。

オフセット

+0	exinf	拡張情報
+4	wtsk	待ちタスクの有無
+6	semcnt	現在のセマフォのカウント値

【エラーコード】

E_OK	00000H(-H'0000): 正常終了

【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。

●exinf

拡張情報を返します。
MR308 では、常に不定です。

●wtsk

wtsk には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 0(FALSE)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。

本システムコールはタスク、ハンドラのどちらからでも発行できます。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
    T_RSEM   rsem;
        :
    ref_sem( &rsem, ID_sem );
        :
}
```

《 アセンブリ言語の使用例 》

```
rsem:  .blkb   8
    .INCLUDE    mr308.inc
    . GLB       task
task:
        :
    ref_sem     #ID_sem1,#rsem
        :
```

2.5. 同期・通信機能(メイルボックス)

2.5.1. snd_msg(Send Message to Mailbox)

【システムコール名】

snd_msg　　→　　メッセージを送信します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER snd_msg (mbxid, pk_msg);
```

《引数》

ID	mbxid;	メイルボックス ID 番号
T_MSG	*pk_msg;	メッセージパケットの先頭アドレス

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
snd_msg  mbxid, pk_msg
```

《引数》

mbxid	[---*]	送信を行うメイルボックスの ID 番号
pk_msg	[--**]	メッセージパケットの先頭アドレス(16 ビット)
	[****]	メッセージパケットの先頭アドレス(32 ビット)

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	メッセージパケットの先頭下位アドレス(32 ビットのみ)
R2	メッセージパケットの先頭アドレス(16 ビットのみ)
R3	メッセージパケットの先頭上位アドレス(32 ビットのみ)
A0	送信を行うメイルボックスの ID 番号

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了
E_QOVR　0FFB7H(-H'0049): キューイングまたはネストのオーバーフロー

【機能説明】

mbxid で示されたメイルボックスにメッセージを送信します。

メッセージを待つタスクがないときには、メッセージはメッセージキューに FIFO[24]順で格納します。すなわち、本システムコール発行によりメイルボックスに送信された順にメッセージが取り出されます。メッセージを待つタスクがある場合は、メッセージをそのタスクに渡し、そのタスクの待ち状態を解除します。

メッセージキューのサイズは、コンフィグレーションファイルにより定義します。

メッセージキューが一杯になった状態のメイルボックスに対して本システムコールを発行した場合、システムコール発行タスクにエラーE_QOVR を返します。

メッセージは 16 または 32 ビット幅データです。[25] μITRON 仕様では、このデータをメッセージパケット(メッセージを含む構造体)の先頭アドレスとして扱うことを標準としています(アドレス渡し)が、**MR308** では、メッセージを以下の 2 通りの方法でデータを通信することができます。

1. メッセージを、メッセージパケットの先頭アドレス(16 または 32 ビット)とする場合

   MR308 では、メッセージパケットの型(T_MSG)は特に規定していないので、ユーザーが自由に定義できます。例えば、配列でもかまいません[26]。

   例えば、

   ```
   typedef char T_MSG;
   ```

   メッセージパケットの先頭アドレス pk_msg は以下のように定義しなければなりません。

   16 ビットの場合

   ```
   T_MSG near * pk_msg;
   ```

   32 ビットの場合

   ```
   T_MSG far * pk_msg;
   ```

2. メッセージを単なるデータとする場合

   この場合、snd_msg、isnd_msg システムコールの第二引数(送信するメッセージデータ pk_msg)を(PT_MSG)で、rcv_msg、prcv_msg の第一引数(メッセージデータを格納する領域のアドレス ppk_msg)を(PT_MSG *)でキャストしなければなりません。

   例えば、long 型の変数 i を通信する場合、

   ```
   long i, j;
   snd_msg( ID_mbx, (PT_MSG)i );
   rcv_msg( (PT_MSG *)&j, ID_mbx );
   ```

   このように記述すれば、直接 32 ビットデータを通信できます。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラから発行する場合は isnd_msg システムコールを使用してください。

[24] ファーストインファーストアウト

[25] 16、32 ビットのデータ幅の選択はコンフィグレーションファイルで行います。

[26] 本マニュアルの**【 C 言語による呼び出し方法】**ではメッセージパケットの先頭アドレスを送信することを標準として記述しています。

【使用例】

この例では、メッセージパケットの先頭アドレスを送る場合を示しています。

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
typedef char T_MSG;
T_MSG far *msg;
void task(void)
{
        :
    if( snd_msg( ID_msg, msg ) != E_OK ){
        error("overflow¥n”);
    }
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
msg:    .BLKL   1

task:
    snd_msg     #ID_msg, msg
        :
```

2.5.2. isnd_msg(Send Message to Mailbox)

【システムコール名】

isnd_msg　　→　　メッセージを送信します (ハンドラ専用)

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER isnd_msg (mbxid, pk_msg);
```

《引数》

ID	mbxid;　メイルボックス ID 番号
T_MSG	*pk_msg; メッセージパケットの先頭アドレス

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
isnd_msg mbxid, pk_msg
```

《引数》

mbxid	[---*]	送信をおこなうメイルボックス ID 番号
pk_msg	[--**]	メッセージパケットの先頭アドレス(16 ビット)
	[****]	メッセージパケットの先頭アドレス(32 ビット)

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	メッセージパケットの先頭下位アドレス(32 ビットのみ)
R2	メッセージパケットの先頭アドレス(16 ビットのみ)
R3	メッセージパケットの先頭上位アドレス(32 ビットのみ)
A0	送信を行うメイルボックスの ID 番号

【エラーコード】

E_OK	00000H(-H'0000): 正常終了
E_QOVR	0FFB7H(-H'0049): キューイングまたはネストのオーバーフロー

【機能説明】

snd_msg システムコールの機能を割り込みハンドラ、周期起動ハンドラ、アラームハンドラから利用する場合に、このシステムコールを使用してください。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
typedef char T_MSG;
T_MSG msg[10];
void inthand()
{
        :
    if( isnd_msg(ID_msg, msg) != E_OK ) {
        error("overflow¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        intr
intr:
        :
    isnd_msg    #ID_msg, #1234H
        :
    ret_int
```

2.5.3. rcv_msg(Receive Message from Mailbox)

【システムコール名】

rcv_msg　　→　　メッセージを受信します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER rcv_msg (ppk_msg, mbxid);
```

《引数》

ID	mbxid;	メイルボックス ID 番号
T_MSG	**ppk_msg;	メッセージパケットの先頭アドレスを格納する領域のアドレス

《戻り値》

関数の戻り値としてエラーコードを返します。
ppk_msg が指す領域に、受信したメッセージパケットの先頭アドレスが設定されます。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
rcv_msg  mbxid
```

《引数》

mbxid	[---*]	受信するメイルボックスの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	メッセージパケットの先頭上位アドレス(32 ビットのみ)
R2	メッセージパケットの先頭下位アドレス
A0	受信するメイルボックスの ID 番号

【エラーコード】

E_OK	00000H(-H'0000): 正常終了
E_RLWAI	0FFAAH(-H'0056): 待ち状態強制解除

【機能説明】

mbxid で示されたメイルボックスからメッセージを受信します。

対象メイルボックスにメッセージが到着している場合には、メッセージキューの先頭にあるメッセージを 1 つ取り出して、それをリターンパラメータ pk_msg として返します。一方、そのメイルボックスにまだメッセージが送信されていない場合、本システムコールを発行したタスクは待ち状態となり、待ち行列に FIFO 順でつながれます。

rel_wai、irel_wai システムコールによって待ち状態が解除された場合には、エラーE_RLWAI が返されます。

エラーコードが E_OK 以外の場合は、リターンパラメータ(*ppk_msg)は不定です。

メッセージを受信する場合、以下の点の注意が必要です。

1. メッセージをメッセージパケットの先頭アドレスとする場合

   メッセージパケットの先頭アドレスを格納する領域へのポインタ変数(ppk_msg)を以下のように宣言します。

   16 ビットのデータを受信する場合

   ```
   T_MSG near * ppk_msg;
   ```

   32 ビットのデータを受信する場合

   ```
   T_MSG far * ppk_msg;
   ```

   ```
   rcv_msg(&ppk_msg,ID_mbx);
   ```

2. メッセージを、単なるデータとする場合

   また、rcv_msg、prcv_msg の第一引数 (メッセージデータを格納する領域のアドレス ppk_msg)を(PT_MSG *)でキャストしてください。

   例えば、int 型の変数 i を通信する場合、

   ```
   int i, j;
   snd_msg( ID_mbx, (PT_MSG)i );
   rcv_msg( (PT_MSG *)&j, ID_mbx );
   ```

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラからは発行した場合は、正常に動作しません。

【使用例】

この例では、メッセージをデータの先頭アドレスとして送る場合を示している。

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
typedef T_MSG char;
void task()
{
    T_MSG *msg;
        :
    if( rcv_msg( &msg, ID_mbx ) != E_OK )
        error("forced wakeup¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    rcv_msg     #ID_mbx
        :
```

2.5.4. trcv_msg(Receive Message with Timeout)

【システムコール名】

trcv_msg　　→　　メッセージを受信します（タイムアウトあり）

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER trcv_msg (ppk_msg, mbxid, tmout);
```

《引数》

ID	mbxid;	メイルボックス ID 番号
T_MSG	**ppk_msg;	メッセージパケットの先頭アドレスを格納する領域のアドレス
TMO	tmout	タイムアウト値

《戻り値》

変数 pk_msg に、受信したメッセージパケットの先頭アドレスが設定されます。関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
trcv_msg mbxid,tmout
```

《引数》

mbxid	[---*]	受信するメイルボックスの ID 番号
tmout	[--**]	タイムアウト値

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	メッセージパケットの先頭上位アドレス(32 ビットのみ)
R2	メッセージパケットの先頭下位アドレス
R3	--
A0	受信するメイルボックスの ID 番号

【エラーコード】

E_OK	00000H(-H'0000): 正常終了
E_TMOUT	0FFABH(-H'0055): ポーリング失敗またはタイムアウト
E_RLWAI	0FFAAH(-H'0056): 待ち状態強制解除

【機能説明】

mbxid で示されたメイルボックスにメッセージがあれば受信します。そのメイルボックスにメッセージが入っている場合には、メッセージキューの先頭にあるメッセージを 1 つ取り出して、それをリターンパラメータ ppk_msg として得ます。

一方、そのメイルボックスに、まだメッセージが送信されていない場合には、本システムコールを発行したタスクは、待ち状態となり、メッセージ待ち行列とタイムアウト待ち行列の 2 つの待ち行列につながれます。

本システム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。なお、待ち状態が解除されると本システムコールを発行したタスクはメッセージ待ち行列とタイムアウト待ち行列の 2 つの待ち行列からはずされ、レディキューに接続されます。

- tmout の時間が経過する前にメッセージが到着した場合

  この時のエラーコードは、E_OK を返します。

- メッセージが到着しないまま、tmout の時間が経過した場合

  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 システム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。

tmout には、-1 から 7FFFH まで指定できます。なお、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_msg 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_msg システムコールと同じ動作になります。

メッセージ受信時の注意事項は、rcv_msg を参照してください。

エラーコードが E_OK 以外の場合は、リターンパラメータ(*ppk_msg)は不定です。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラからは発行した場合は、正常に動作しません。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
typedef char T_MSG;
void task()
{
    T_MSG   *msg;
        :
    if( trcv_msg( &msg, ID_mbx, 10 ) != E_OK ){
        error("Can't Get Message¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    trcv_msg    #ID_mbx,#10
        :
```

2.5.5. prcv_msg(Poll and Receive Message)

【システムコール名】

prcv_msg　　→　　メッセージを受信します (待ちなし)

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER prcv_msg (ppk_msg, mbxid);
```

《引数》

ID	mbxid;	メイルボックス ID 番号
T_MSG	**ppk_msg;	メッセージパケットの先頭アドレスを格納する領域のアドレス

《戻り値》

変数 pk_msg に、受信したメッセージパケットの先頭アドレスが設定されます。
関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
prcv_msg mbxid
```

《引数》

mbxid	[---*]	受信するメイルボックスの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	メッセージパケットの先頭上位アドレス(32 ビットのみ)
R2	メッセージパケットの先頭下位アドレス
A0	受信するメイルボックスの ID 番号

【エラーコード】

E_OK	00000H(-H'0000): 正常終了
E_TMOUT	0FFABH(-H'0055): ポーリング失敗またはタイムアウト

【機能説明】

mbxid で示されたメイルボックスからメッセージがあれば受信します(待ちなし)。そのメイルボックスにメッセージが入っている場合には、メッセージキューの先頭にあるメッセージを 1 つ取り出して、それをリターンパラメータ ppk_msg として得ます。

一方、そのメイルボックスに、まだメッセージが送信されていない場合には、システムコール発行タスクにエラーE_TMOUT を返し、本システムコールを終了します。rcv_msg、trcv_msg とは、異なり本システムコールを発行したタスクは、待ち状態には移行しません。

エラーコードが E_OK 以外の場合は、リターンパラメータ(*ppk_msg)は不定です。

本システムコールはタスク、ハンドラのどちらからでも発行できます。

メッセージ受信時の注意事項は、rcv_msg を参照してください。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
typedef char T_MSG;
void task()
{
    T_MSG * msg;
        :
    if( prcv_msg( &msg, ID_mbx ) != E_OK ){
        error("Can't Get Message¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
    prcv_msg    #ID_mbx1
        :
```

2.5.6. ref_mbx(Refer Mailbox Status)

【システムコール名】

ref_mbx　　→　　メイルボックス状態を参照します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER ref_mbx (pk_rmbx, mbxid);
```

《引数》

```
T_RMBX      *rmbx;   メイルボックス状態を返すパケットの先頭アドレス
ID          mbxid;   状態を参照するメイルボックス ID 番号
```

《戻り値》

関数の戻り値としてエラーコードを返します。
pk_rmbx の指すパケットには、以下の情報が返されます。

```
typedef struct t_rmbx {
    VP      exinf;   /*拡張情報*/
    BOOL_ID wtsk;    /*待ちタスクの有無*/
    T_MSG   *pk_msg; /*次に受信されるメッセージパケットの先頭アドレス*/
    INT     msgcnt;  /* メッセージ数 */
} T_RMBX;
```

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
ref_mbx  mbxid, pk_rmbx
```

《引数》

```
mbxid       [---*]   状態を参照するメイルボックス ID 番号
pk_rmbx     [-***]   メイルボックス状態を返すパケットの先頭アドレス
```

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	状態を参照するメイルボックスの ID 番号
A1	メイルボックスの状態を返すパケットの先頭アドレス

pk_rmbx の指す領域には、以下の情報が返されます。

オフセット

```
+0      exinf    拡張情報
+4      wtsk     待ちタスクの有無
+6      pk_msg   次に受信されるメッセージパケットの先頭アドレス
 *      msgcnt   メッセージ数
   *:   メッセージのサイズが 16 ビットの場合 +8
        メッセージのサイズが 32 ビットの場合 +10
```

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了

【機能説明】

本システムコールでは、mbxid で指定したメイルボックスの以下の状態を返す。

●exinf

拡張情報を返します。
MR308 では、常に不定です。

●wtsk

wtsk には､指定したメイルボックスでメッセージを待っている先頭タスク(最も早く待ちに入ったタスク)ID 番号を返します。メッセージを待っているタスクがない場合は FALSE(0)を返します。

●pk_msg

pk_msg には、次に rcv_msg または trcv_msg を実行した場合に受信されるメッセージ(キューイングされている先頭メッセージ)を返します。メッセージがない場合は、NADR(-1)=0xFFFF[27]を返します。

●msgcnt

対象メイルボックス中にある、現在のメッセージ数を返します。

本システムコールはタスク、ハンドラのどちらからでも発行できます。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
    T_RMBX rmbx;
        :
    ref_mbx(&rmbx, ID_mbx);
        :
}
```

《 アセンブリ言語の使用例 》

```
rmbx:  .blkb   10
    .INCLUDE   mr308.inc
    .GLB       task
task:
        :
    ref_mbx    #ID_mbx,#rmbx
        :
```

[27] メッセージサイズが 32 ビットの場合は 0FFFFFFFFH を返します。

2.6. 割り込み管理機能

2.6.1. ret_int(Return from Interrupt Handler)

【システムコール名】

ret_int　　→　　割り込みハンドラから復帰します

【 C 言語による呼び出し方法】

本システムコールは、C 言語では記述できません。[28]

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
ret_int
```

《引数》

なし

《レジスタ設定》

本システムコールを発行した割り込みハンドラには戻りません。

【エラーコード】

本システム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システム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 システム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、本システム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システム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    mr308.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    mr308.inc
   /* OK */
   .GLB        intr
intr:
   jsr.b       func
   ret_int
func:
   :
   rts
```

本システム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

[28] 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 システムコールを発行します。

ラ及びタスクから発行した場合は、正常に動作しません。

【使用例】

《 C 言語の使用例 》

本システムコールは、C 言語では記述できません。
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 システムコールを発行します。

《 アセンブリ言語の使用例 》

```
    .INCLUDE mr308.inc
    .GLB        intr
intr:
        :
    iwup_tsk #ID_main
        :
    ret_int
```

2.6.2. loc_cpu(Lock CPU)

【システムコール名】

loc_cpu　　→　　割り込みとディスパッチを禁止します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER loc_cpu ();
```

《引数》

なし

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
loc_cpu
```

《引数》

なし

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	--

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了

【機能説明】

OS 依存の外部割り込み(IPL=0～OS 割り込み禁止レベル)とタスクのディスパッチを禁止します。

本システムコール実行後、unl_cpu が実行されるまでの間は、OS 依存の外部割り込み禁止およびディスパッチ禁止状態となり、loc_cpu を実行したタスクが割り込みハンドラや他のタスクによってプリエンプト(CPU の実行権の横取り)される可能性はなくなります。つまり、loc_cpu 実行後の割り込み要求や、loc_cpu を実行したタスクが発行したシステムコールにより発生したディスパッチは、unl_cpu により割り込み禁止およびディスパッチ禁止状態が解除されるまで遅延されます。

loc_cpu または dis_dsp を発行した場合、必ず、unl_cpu または ena_dsp を発行してからタスクを終了(ext_tsk の発行)してください。

既に割り込みおよびディスパッチ禁止状態にあるタスクが loc_cpu を発行した場合は、同じ状態を継続するだけでエラーとはなりません。ただし、loc_cpu を複数回発行しても、その後、unl_cpu を 1 回発行すると割り込みおよびディスパッチ禁止状態が解除されます。

本システムコールは、タスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラ、アラームハンドラから発行した場合は、正常に動作しません。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
        :
    loc_cpu();
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    loc_cpu
        :
```

2.6.3. unl_cpu(Unlock CPU)

【システムコール名】

unl_cpu　　→　　割り込みディスパッチを許可します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER unl_cpu ();
```

《引数》

なし

《戻り値》

関数の戻り値として常に E_OK を返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
unl_cpu
```

《引数》

なし

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	--

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了

【機能説明】

OS 依存の外部割り込みとタスクのディスパッチを許可します。したがって、本システムコール発行後の IPL 値は、loc_cpu 発行時の IPL 値になります。

割り込みおよびディスパッチが禁止されていない状態で unl_cpu を発行した場合は、同じ状態が継続するだけでエラーとはなりません。

本システムコールは、タスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラ、アラームハンドラから発行した場合は、正常に動作しません。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
        :
    unl_cpu();
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
        :
    unl_cpu
        :
```

2.7. メモリプール管理機能

2.7.1. pget_blf(Poll and Get Fixed-size Memory Block)

【システムコール名】

pget_blf　　→　　固定長メモリブロックを獲得します(待ちなし)

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER pget_blf (p_blf,mpfid);
```

《引数》

ID	mpfid;	メモリプール ID 番号
VP	*p_blf;	獲得した固定長メモリブロックの先頭アドレスを格納する領域のアドレス

《戻り値》

変数 p_blf に獲得したメモリブロックの先頭アドレスが設定されます。関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
pget_blf mpfid
```

《引数》

mpfid	[---*]	獲得する固定長メモリプールの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	獲得したメモリブロックの先頭下位アドレス
R2	--
R3	獲得したメモリブロックの先頭上位アドレス
A0	獲得する固定長メモリプールの ID 番号

【エラーコード】

E_OK	00000H(-H'0000): 正常終了
E_TMOUT	0FFABH(-H'0055): ポーリング失敗またはタイムアウト

【機能説明】

mpfid で示される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_blf に格納します。

指定したメモリプールにメモリブロックがない場合は、エラーE_TMOUT を返します。

メモリブロックは、固定長です。1 つのメモリブロックのサイズ、および、メモリブロック数はコンフィグレーションファイルで設定します。1 つのメモリプールには、メモリブロック数を、最大 16 個まで指定することができます。

エラーコードが E_OK 以外の場合は、リターンパラメータ(*p_blf)は不定です。

本システムコールは、タスク、ハンドラのどちらからでも発行できます。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
VP      p_blf;
void task()
{
    if( pget_blf(&p_blf,ID_mpf) != E_OK ){
        error("Not enough memory¥n”);
    }
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB task
p_blf: .BLKL    1
task:
        :
    pget_blf    #ID_mpf
    mov.w       R1, p_blf
    mov.w       R3, p_blf+2
        :
    ext_tsk
```

2.7.2. rel_blf(Release Fixed-size Memory Block)

【システムコール名】

rel_blf　　→　　固定長メモリブロックを解放します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER rel_blf (mpfid, p_blf);
```

《引数》

ID	mpfid;	解放するメモリの固定長メモリプール ID 番号
VP	p_blf;	解放する固定長メモリブロックの先頭アドレス

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
rel_blf  mpfid, p_blf
```

《引数》

mpfid	[---*]	解放するメモリの固定長メモリプール ID 番号
p_blf	[-***]	解放する固定長メモリブロックの先頭アドレス

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	解放する固定長メモリブロックの先頭下位アドレス
R2	--
R3	解放する固定長メモリブロックの先頭上位アドレス
A0	解放するメモリの固定長メモリプール ID 番号

【エラーコード】

E_OK	00000H(-H'0000): 正常終了

【機能説明】

p_blf 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。
解放するメモリブロックの先頭アドレスは必ず、pget_blf で獲得した先頭アドレスを指定してください。
本システムコールは、p_blf の内容をチェックしません。
本システムコールは、タスク、ハンドラのどちらからでも発行できます。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
#define ID_mpf1 1
void task()
{
    VP   p_blf;
    if( pget_blf(&p_blf, ID_mpf1) != E_OK )
        error("Not enough memory ¥n”);
        :
    rel_blf(ID_mpf1,p_blf);
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        _task
p_blf:  .BLKL   1

_task:
        :
    pget_blf    #ID_mpf1
    mov.w       R1, p_blf
    mov.w       R3, p_blf+2
        :
        :
    rel_blf     #ID_mpf1,p_blf
        :
    ext_tsk
```

2.7.3. ref_mpf(Refer Fixed-size Memorypool Status)

【システムコール名】

ref_mpf　　→　　固定長メモリプール状態を参照します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER ref_mpf (pk_rmpf, mpfid);
```

《引数》

T_RMPF	*pk_rmpf;	固定長メモリプール状態を返すパケットの先頭アドレス
ID	mpfid;	参照する固定長メモリプール ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。
pk_rmpf の指す構造体には、以下の情報が返されます。

```
typedef  struct t_rmpf {
    VP      exinf;  /* 拡張情報 */
    BOOL_ID wtsk;   /* 待ちタスクの有無 */
    INT     frbcnt; /* 空き領域のブロック数 */
    INT     blksz;  /* ブロックサイズ */
} T_RMPF;
```

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
ref_mpf  mpfid, pk_rmpf
```

《引数》

mpfid	[---*]	参照する固定長メモリプール ID 番号
pk_rmpf	[-***]	固定長メモリプールの状態を返すパケットの先頭アドレス

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	参照する固定長メモリプール ID 番号
A1	固定長メモリプールの状態を返すパケットの先頭アドレス

pk_rmpf の指す領域には、以下の情報が返されます。

オフセット

+0	exinf	拡張情報
+4	wtsk	待ちタスクの有無
+6	frbcnt	空き領域のブロック数
+8	blksz	メモリブロックサイズ

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了

【機能説明】

本システムコールでは、mpfid で指定した固定長メモリプールの以下の状態を返します。

●exinf

拡張情報を返します。
MR308 では、常に不定です。

●wtsk

wtsk には、指定されたメモリプールを待っている先頭タスクの ID 番号を返します。**MR308** ではメモリプールに対して待ち状態になることがないので、常に FALSE(0)を返します。

●frbcnt

指定した固定長メモリプールの空きブロック数を返します。

●blksz

対象となる固定長メモリプールのブロックサイズを返します。

本システムコールは、タスク、ハンドラのどちらからでも発行できます。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
    T_RMPF rmpf;
        :
    ref_mpf(&rmpf, ID_mpf);
        :
    ext_tsk();
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
rmpf:   .BLKL   8

task:
        :
    ref_mpf     #ID_mpf,#rmpf
        :
```

2.7.4. pget_blk(Poll and Get Variable-size Memory Block)

【システムコール名】

pget_blk　　→　　可変長メモリブロックを獲得します（待ちなし）

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER pget_blk (p_blk,mplid,blksz);
```

《引数》

VP	*p_blk;	獲得したメモリブロックの先頭アドレスを格納する領域のアドレス
ID	mplid;	メモリプール ID 番号
INT	blksz;	獲得するメモリのサイズ

《戻り値》

変数 p_blk に獲得したメモリブロックの先頭アドレスが設定されます。関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
pget_blk blksz
```

《引数》

blksz	[--**]	獲得する可変長メモリブロックのサイズ

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	獲得したメモリブロックの先頭下位アドレス
R2	--
R3	獲得したメモリブロックの先頭上位アドレス
A0	--

【エラーコード】

E_OK	00000H(-H'0000): 正常終了
E_TMOUT	0FFABH(-H'0055): ポーリング失敗またはタイムアウト

【機能説明】

可変長のメモリブロックを獲得します。

変数 blksz に指定された大きさのメモリを可変長メモリプールから獲得し、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。

指定したサイズのメモリブロックが、メモリプールにない場合は、エラーE_TMOUT を返します。

MR308 では、可変長メモリプールは、1 つのみで mplid には 1 を指定してください。

この可変長メモリプールのサイズは、コンフィグレーションファイルで指定してください。

エラーコードが E_OK 以外の場合は、リターンパラメータ(*p_blk)は不定です。

本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよび、アラームハンドラから発行した場合は、正常に動作しません。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
#define ID_mpl1 1
VP      p_blk;
void task()
{
    /* 70 バイトのメモリブロックを獲得 */
    if( pget_blk(&p_blk,ID_mpl1,70) != E_OK )-
        error("Not enough memory¥n”);
    }
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE mr308.inc
    .GLB task

p_blk: .blkb 4
task:
        :
    pget_blk    #50             ; 50 バイトのメモリを獲得
    mov.w       R1, p_blk       ; 獲得したメモリのアドレスを転送
    mov.w       R3, p_blk+2
        :
    ext_tsk
```

2.7.5. rel_blk(Release Variable-size Memory Block)

【システムコール名】

rel_blk　　→　　可変長メモリブロックを解放します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER rel_blk (mplid,p_blk);
```

《引数》

ID	mplid;	メモリプール ID 番号
VP	p_blk;	解放メモリブロックの先頭アドレス

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
rel_blk  p_pblk
```

《引数》

p_blk	[-***]	解放するメモリブロックの先頭アドレス

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	解放するメモリブロックの先頭下位アドレス
R2	--
R3	解放するメモリブロックの先頭上位アドレス
A0	--

【エラーコード】

E_OK	00000H(-H'0000): 正常終了

【機能説明】

p_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。
解放するメモリブロックの先頭アドレスは必ずpget_blkで獲得した先頭アドレスを指定してください。
本システムコールは、p_blk の内容をチェックしていません。
本システムコールはタスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラおよびアラームハンドラから発行した場合は、正常に動作しません。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
#define ID_mpl1 1
void task()
{
    VP   p_blk;
    /* 60 バイトのメモリブロックを獲得 */
    if( pget_blk(&p_blk,ID_mpl1,60) != E_OK )
        error("Not enough memory ¥n”);
        :
        :
    rel_blk(ID_mpl1,p_blk);     /* メモリブロックを解放 */
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        _task

    p_blk:      .BLKL   1
_task:
        :
    pget_blk    #60             ; 60 バイトのメモリブロックを獲得
    mov.w       R1, p_blk
    mov.w       R3, p_blk+2
        :
    rel_blk     p_blk           ; メモリブロックを解放
```

2.7.6. ref_mpl(Refer Variable-size Memorypool Status)

【システムコール名】

ref_mpl　　→　　可変長メモリプール状態を参照します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER ref_mpl (pk_rmpl, mplid);
```

《引数》

T_RMPL	*pk_rmpl;	可変長メモリプール状態を返すパケットの先頭アドレス
ID	mplid;	状態を参照する可変長メモリプール ID 番号(1)

《戻り値》

関数の戻り値としてエラーコードを返します。
pk_rmpl の指す構造体には、以下の情報が返されます。

```
typedef struct t_rmpl {
    VP      exinf;  /* 拡張情報 */
    BOOL_ID wtsk;   /* 待ちタスクの有無 */
    W       frsz;   /* 空き領域の合計サイズ */
    INT     maxsz;  /* 空き領域の最大サイズ */
} T_RMPL;
```

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
ref_mpl  pk_rmpl
```

《引数》

pk_rmpl	[-***]	可変長メモリプール状態を返すパケットの先頭アドレス

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	--
A1	可変長メモリプールの状態を返すパケットの先頭アドレス

pk_rmpl の指す領域には、以下の情報が返されます。

オフセット

+0	exinf	拡張情報
+4	wtsk	待ちタスクの有無
+6	frsz	空き領域の合計サイズ
+10	maxsz	空き領域の最大サイズ

【エラーコード】

E_OK	00000H(-H'0000): 正常終了

【機能説明】

mplid で指定した可変長メモリプールの状態を返します。

●exinf

拡張情報を返します。
MR308 では、常に不定です。

●wtsk

wtsk には、指定されたメモリプールを待っている先頭タスク ID 番号を返します。MR308 では、可変長メモリプールに対して待ち状態になることがないので、常に FALSE(0)を返します。

●frsz

空き領域の合計サイズを返します。

●maxsz

すぐに獲得できる最大の空き領域サイズを返します。

本システムコールは、タスク、ハンドラのどちらからでも発行できます。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
    T_RMPL rmpl;
    ref_mpl(&rmpl,ID_mpl1);
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
rmpl:  .BLKB    10

task:
        :
    ref_mpl     #rmpl
        :
```

2.8. 時間管理機能

2.8.1. set_tim(Set Time)

【システムコール名】

set_tim　　→　　システムクロックを設定します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER set_tim (pk_tim);
```

《引数》

SYSTIME　　*pk_tim; 設定するシステム時刻データの先頭アドレス

《戻り値》

関数の戻り値として常に E_OK を返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
set_tim  pk_tim
```

《引数》

pk_tim　　[-***]　設定するシステム時刻を示すパケットの先頭アドレス

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	設定するシステム時刻パケットの先頭アドレス

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了

【機能説明】

システムクロックの値を pk_tim で示される値に設定します[29]。

48 ビットのシステムクロックは ltime、mtime、utime に分けて扱います。

なお、本システムコールにより、すでに実行されたアラームハンドラの起動時刻よりも前の時刻にシステムクロックの値を戻しても、そのアラームハンドラが再び起動されることはありません。また、まだ起動されていないアラームハンドラの起動時刻よりも後の時刻を設定した場合は、すべてのアラームハンドラが起動されなくなります。

システムクロックの単位は、**MR308** のシステムクロックの時間を 1 クロックとして扱います。

本システムコールはタスク、ハンドラのどちらからでも発行できます。

[29] システム時刻は、リセット時を 0 とし、システムクロックの割り込みの発生回数を 48 ビットのデータで表します。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
    SYSTITIME time;     /* 時刻データ格納変数 */
    time.utime = 2;     /* 上位時刻データの設定 */
    time.mtime = 1;     /* 中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    mr308.inc
    .GLB        task
time:
    .WORD       2
    .WORD       1
    .WORD       0

task:
    set_tim     #time
        :
```

2.8.2. get_tim(Get Time)

【システムコール名】

get_tim　　→　　システム時刻の値を読み出します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER get_tim (pk_tim);
```

《引数》

SYSTIME　　*pk_tim; 現在の時刻データを返す構造体の先頭アドレス

《戻り値》

関数の戻り値として常に E_OK を返します。
pk_tim の指す構造体には、現在の時刻データが返されます。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
get_tim
```

《引数》

pk_tim　　[-***]　読みだしたシステム時刻を格納する
パケットの先頭アドレス

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	読み出したシステム時刻を格納するパケットの先頭アドレス

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了

【機能説明】

システムクロックの現在の値を読みだし、リターンパラメータ pk_tim に返します[30]。
48 ビットのシステム時刻は、ltime、mtime、utime に分けて扱います。
本システムコールはタスク、ハンドラのどちらからでも発行できます。

[30] システム時刻は、リセット時を 0 とし、システムクロックの割り込みの発生回数を 48 ビットのデータで表します。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
 #include "id.h"
 void task()
 {
    SYSTIME     time;           /* 時刻データ格納変数 */
    get_tim( &time );           /* 時刻データの読み出し */
    printf("system_clock.utime = %X¥n”,time.utime);
    printf("system_clock.mtime = %X¥n”,time.mtime);
    printf("system_clock.ltime = %X¥n”,time.ltime);
 }
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
time:
    .BLKW       3

 task:
    get_tim     #time
        :
```

2.8.3. dly_tsk(Delay Task)

【システムコール名】

dly_tsk　　→　　タスクの実行を遅延します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER dly_tsk (dlytim);
```

《引数》

DLYTIME　　dlytim; 遅延時間

《戻り値》

関数の戻り値として常にエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
dly_tsk  dlytim
```

《引数》

dlytim　　[--**]　遅延時間

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	遅延時間
A0	--

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了
E_RLWAI　　0FFAAH(-H'0056): 待ち状態強制解除

【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(RUN)状態から待ち状態へ移行します。

本システム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システム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 の時間が経過した場合

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 rel_wai、irel_wai システム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。

また、遅延時間中に wup_tsk、iwup_tsk システムコールが発行されても、待ち解除とはなりません。

dlytim で指定する時間の単位はコンフィグレーションファイルで指定したシステムクロックの単位時間です[31]。

dlytim の最大値は 7FFF(32767)です。

システムクロックの単位時間が 10ms の場合プログラムで、

```
dly_tsk(5);
```

と記述すれば 50ms 間、自タスクが実行(RUN)状態からタイムアウト待ち状態へ移行します。

本システムコールは、タスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラ、アラームハンドラから発行した場合は、正常に動作しません。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
    if( dly_tsk( 10 ) != E_OK );
        printf("Forced wakeup¥n”);
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
 task:
    dly_tsk     #200
        :
```

[31] システムクロックの設定方法については、ユーザーズマニュアルを参照してください。

2.8.4. act_cyc (Activate Cyclic Handler)

【システムコール名】

act_cyc　　→　　周期起動ハンドラの活性制御します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER act_cyc (cycno, cycact);
```

《引数》

HNO	cycno;	周期起動ハンドラ指定番号
UINT	cycact;	周期起動ハンドラ活性状態

《戻り値》

関数の戻り値として常に E_OK を返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
act_cyc  cycno, cycact
```

《引数》

cycno	[---*]	周期起動ハンドラ指定番号
cycact	[--**]	周期起動ハンドラ活性状態

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	周期起動ハンドラ活性状態
A0	周期起動ハンドラ指定番号

【エラーコード】

E_OK	00000H(-H'0000): 正常終了

【機能説明】

cycno で示された周期起動ハンドラの活性状態を変更します。すなわち周期起動ハンドラをイネーブル(有効)にしたりディスイネーブル(無効)にしたりします。

cyhact で指定できるのは以下の 3 通りです。

周期起動ハンドラの活性状態指定

C 言語	アセンブリ言語	意味
TCY_OFF	TCY_OFF	周期起動ハンドラを無効にします。
TCY_ON	TCY_ON	周期起動ハンドラを有効にします。
TCY_ON|TCY_INI	TCY_INI_ON	周期起動ハンドラを有効にすると同時に周期カウンタをクリアします。

周期起動ハンドラは、システムクロック割り込みハンドラの一部として実行されます[32]。

本システムコールはタスク、ハンドラのどちらからでも発行できます。

[32] すなわち、システムクロック割り込みハンドラからサブルーチンコールにより呼び出されます。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
        :
    act_cyc ( ID_cyc, TCY_ON );
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task
task:
    act_cyc     #ID_cyc, #TCY_INI_ON
        :
```

2.8.5. ref_cyc(Refer Cyclic Handler Status)

【システムコール名】

ref_cyc　　→　　周期起動ハンドラ状態を参照します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER ref_cyc (pk_rcyc, cycno);
```

《引数》

HNO	cycno;	周期起動ハンドラ指定番号
T_RCYC	*pk_rcyc;	周期起動ハンドラの状態を返す構造体の先頭アドレス

《戻り値》

関数の戻り値としてエラーコードを返します。
pk_rcyc の指す構造体には、以下のような周期起動ハンドラの状態が設定されます。

```
typedef struct t_rcyc {
    VP      exinf;  /* 拡張情報 */
    CYCTIME lfttim; /* 次にハンドラが起動されるまでの残り時間 */
    UINT    cycact; /* 周期起動ハンドラ活性状態 */
}T_RCYC;
```

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
ref_cyc  cycno, pk_rcyc
```

《引数》

cycno	[---*]	周期起動ハンドラ指定番号
pk_rcyc	[-***]	周期起動ハンドラの状態を返すパケットの先頭アドレス

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	周期起動ハンドラの指定番号
A1	周期起動ハンドラの状態を返すパケットの先頭アドレス

pk_rcyc の指す領域には、以下の情報が返されます。

オフセット

+0	exinf	拡張情報
+4	lftim	次のハンドラ起動までの残り時間
+6	cycact	周期起動ハンドラ活性状態

【エラーコード】

E_OK	00000H(-H'0000): 正常終了

【機能説明】

cycno で指定した周期起動ハンドラの状態を参照し、その結果をリターンパラメータに返します。

●exinf

拡張情報を返します。
MR308 では、常に不定です。

●lfttime

lftime には、次の周期起動ハンドラ起動までの残り時間を返します。周期起動ハンドラ起動までの残り時間はシステムクロックのカウント数で表します。

●cycact

cycact には、周期起動ハンドラ活性状態すなわち周期起動ハンドラが有効か(TCY_ON(=1))無効か(TCY_OFF(=0))を返します。

本システムコールはタスク、ハンドラのどちらからでも発行できます。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
   T_RCYC rcyc;
   ref_cyc( &rcyc, ID_cyc );
}
```

《 アセンブリ言語の使用例 》

```
   .INCLUDE    mr308.inc
   .GLB        task
task:
       :
   ref_cyc     #ID_cyc,#rcyc
       :
```

2.8.6. ref_alm(Refer Alarm Handler Status)

【システムコール名】

ref_alm　　→　　アラームハンドラ状態を参照します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER ref_alm (pk_ralm, almno);
```

《引数》

HNO	almno;	アラームハンドラ指定番号
T_RALM	*pk_ralm;	アラームハンドラ状態を返す構造体の先頭アドレス

《戻り値》

関数の戻り値としてエラーコードを返します。
構造体 pk_ralm に以下のようなアラームハンドラの状態が設定されます。

```
typedef struct t_ralm {
    VP      exinf;  /* 拡張情報 */
    ALMTIME lfttim; /* ハンドラ起動までの残り時間 */
}T_RALM;
```

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
ref_alm  almno, pk_ralm
```

《引数》

almno	[---*]	アラームハンドラ指定番号
pk_ralm	[-***]	アラームハンドラ状態を返すパケットの先頭アドレス

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	アラームハンドラの状態を返すパケットの先頭アドレス
A1	アラームハンドラの指定番号

pk_ralm の示す領域に、以下の情報が返されます。

オフセット

+0	exinf	拡張情報
+4	lftim	ハンドラ起動までの残り時間

【エラーコード】

E_OK	00000H(-H'0000): 正常終了

【機能説明】

almno で指定したアラームハンドラの状態を参照し、その結果をリターンパラメータに返します。

●exinf

拡張情報を返します。
MR308 では、常に不定です。

●lfttim

lfttim には対象となるアラームハンドラが起動するまでの残り時間を返します。アラームハンドラ起動までの残り時間は残りのシステムクロックの割り込みの発生回数を 48 ビットのデータで表します。

48 ビットのシステム時刻は、ltime、mtime、utime に分けて扱います。

本システムコールはタスク、ハンドラのどちらからでも発行できます。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void func()
{
   T_RALM  ralm;
   ref_alm( &ralm, ID_alarm );
       :
}
```

《 アセンブリ言語の使用例 》

```
   .INCLUDE    mr308.inc
   .GLB task
task:
   ref_alm     #ID_alm,#ralm
       :
```

2.9. システム管理機能

2.9.1. get_ver(Get Version Information)

【システムコール名】

get_ver　　→　　**MR308** のバージョン番号を得ます

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER get_ver (pk_ver);
```

《引数》

T_VER　　*pk_ver; バージョン管理情報を返す構造体の先頭アドレス

《戻り値》

関数の戻り値として常に E_OK を返します。
pk_rtsk の指す構造体には、バージョン情報が設定されます。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
get_ver  pk_ver
```

《引数》

pk_ver　　[-***]　バージョン情報を返すパケットの先頭アドレス

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	バージョン情報を返すパケットの先頭アドレス

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了

【機能説明】

MR308 のバージョン番号等の情報を得ます。バージョン番号は TRON 仕様で標準化された形式で得ます。したがって、異なるマイクロコンピュータ間でも、あるいは他の TRON 仕様のオペレーティングシステム間でも共通の形式で得ることができます。

得られるバージョン情報を以下に示します。

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

バージョン番号のフォーマットは次のようになります。

1. メーカー
   メーカーを示すコードが返されます。
2. 形式番号
   MR308 の内部識別 ID 150H が返されます。

3. 仕様書バージョン
   μITRON 仕様書 Ver3.02 に準拠していることを示す 5302H が返されます。
4. 製品バージョン
   MR308 のバージョンを示す 110H が返されます。
5. 製品管理情報

   ●prno[0]

   製品のリリース番号が得られます。
   prno[0] ← 0001H

   ●prno[1]

   製品のリリース年と月が得られます。
   prno[1] ← 0007H

   ●prno[2]

   拡張のための予約
   prno[2] ← ????H

   ●prno[3]

   拡張のための予約
   prno[3] ← ????H
6. CPU 情報
   M16C/80 マイクロコンピュータを示す 0C25H が返されます。
7. バリエーション記述子
   MR308 のバリエーションを示す 8000H が返されます。

本システムコールは、タスク、ハンドラのどちらからでも発行できます。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task()
{
    T_VER    pk_ver;
    get_ver( &pk_ver );
}
```

《 アセンブリ言語の使用例 》

```
ver:
    .BLKW       10
    .INCLUDE    mr308.inc
    .GLB        task
task:
    get_ver     #ver
        :
```

2.10. 拡張機能

2.10.1. vrst_msg(Reset Message)

【システムコール名】

vrst_msg　　→　　メールボックスにあるメッセージをクリアします

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER vrst_msg ( mbxid );
```

《引数》

ID　　mbxid;　メッセージをクリアするメールボックスの ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
vrst_msg mbxid
```

《引数》

mbxid　　[---*]　メッセージをクリアするメイルボックスの ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
R3	--
A0	メッセージをクリアするメイルボックスの ID 番号

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了

【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。メッセージがないときには、何も行いません。

本システムコールを発行すると、エラーコード E_OK を返します。

本システムコールは、タスクからのみ発行してください。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task1(void)
{
    ER ercd;
        :
    ercd = vrst_msg( ID_mbx );
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task1
task1:
        :
    vrst_msg #ID_mbx
        :
    ext_tsk
```

2.10.2. vrst_blf(Reset Fixed-Memory Block)

【システムコール名】

vrst_blf　　→　　指定された固定長メモリブロックを全て解放します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER vrst_blf ( mbfid );
```

《引数》

ID　　mpfid;　解放するメモリプール ID 番号

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
vrst_blf mpfid
```

《引数》

mpfid　　[---*]　解放するメモリプール ID 番号

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	解放するメモリプール ID 番号

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了

【機能説明】

指定された固定長メモリプール ID 番号のメモリブロックを全て解放します。
指定メモリプールが、どのタスクにも使用されていない場合、このシステムコールを発行しても何も起こりません。
本システムコールは、タスクからのみ発行してください。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task1(void)
{
    ER ercd;
        :
    ercd = vrst_blf( ID_mpf );
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task1
task1:
        :
    vrst_blf #ID_mpf
        :
    ext_tsk
```

2.10.3. vrst_blk(Reset Variable-Memory Block)

【システムコール名】

vrst_blk　　→　　可変長メモリブロックを全て解放します

【 C 言語による呼び出し方法】

```
#include <mr308.h>
ER vrst_blk ( void );
```

《引数》

なし

《戻り値》

関数の戻り値としてエラーコードを返します。

【アセンブリ言語による呼び出し方法】

```
.include mr308.inc
vrst_blk
```

《引数》

なし

《レジスタ設定》

レジスタ名	システムコール発行後の内容
R0	エラーコード
R1	--
R2	--
A0	--

【エラーコード】

E_OK　　00000H(-H'0000): 正常終了

【機能説明】

可変長メモリプールのメモリブロックを全て解放します。

可変長メモリプールが、どのタスクにも使用されていない場合、このシステムコールを発行しても何も起こりません。

本システムコールは、タスクからのみ発行してください。割り込みハンドラ、周期起動ハンドラ、アラームハンドラから発行した場合は、正常に動作しません。

【使用例】

《 C 言語の使用例 》

```
#include <mr308.h>
#include "id.h"
void task1(void)
{
    ER ercd;
        :
    ercd = vrst_blk();
        :
}
```

《 アセンブリ言語の使用例 》

```
    .INCLUDE    mr308.inc
    .GLB        task1
task1:
        :
    vrst_blk
        :
    ext_tsk
```

第3章　付録

3.1. システムコール一覧

タスク管理機能システムコール

システムコール名		機能	スケジューラ
sta_tsk	[S]	タスクを起動します	起動
ista_tsk	[S]	タスクを起動します(ハンドラ専用)	―
ext_tsk	[S]	自タスクを正常終了します	起動
ter_tsk	[S]	他タスクを強制的に異常終了します	起動
chg_pri	[S]	タスクの優先度を変更します	起動
ichg_pri	[S]	タスクの優先度を変更します(ハンドラ専用)	―
dis_dsp	[S]	タスクのディスパッチを禁止します	―
ena_dsp	[S]	タスクのディスパッチを許可します	起動
rot_rdq	[S]	レディキューを回転します	起動
irot_rdq	[S]	レディキューを回転します(ハンドラ専用)	―
rel_wai	[S]	待ち状態を強制解除します	起動
irel_wai	[S]	待ち状態を強制解除します(ハンドラ専用)	―
get_tid	[S]	自タスクの ID を得ます	―
ref_tsk	[E]	タスクの状態を参照します	―

タスク付属同期機能システムコール

システムコール名		機能	スケジューラ
sus_tsk	[S]	タスクを強制待ち状態を移行します	起動
isus_tsk	[S]	タスクを強制待ち状態を移行します(ハンドラ専用)	―
rsm_tsk	[S]	強制待ち状態のタスクを再開する	起動
irsm_tsk	[S]	強制待ち状態のタスクを再開する(ハンドラ専用)	―
slp_tsk	[R]	タスクを待ち状態へ移行する	起動
tslp_tsk	[E]	タスクを一定時間待ち状態へ移行する	起動
wup_tsk	[R]	待ち状態のタスクを起床します	起動
iwup_tsk	[R]	待ち状態のタスクを起床します(ハンドラ専用)	―
can_wup	[S]	タスクの起床要求を無効にします	―

同期・通信機能システムコール

システムコール名		機能	スケジューラ
set_flg	[S]	イベントフラグをセットします	起動
iset_flg	[S]	イベントフラグをセットします (ハンドラ専用)	―
clr_flg	[S]	イベントフラグをクリアします	―
wai_flg	[S]	イベントフラグを待ちます	起動
twai_flg	[E]	イベントフラグを待ちます(タイムアウトあり)	起動
pol_flg	[S]	イベントフラグを得ます(待ちなし)	―
ref_flg	[E]	イベントフラグの状態を参照します	―
sig_sem	[R]	セマフォ資源返却	起動
isig_sem	[R]	セマフォ資源返却(ハンドラ専用)	―
wai_sem	[R]	セマフォ資源獲得	起動
twai_sem	[E]	セマフォ資源獲得(タイムアウトあり)	起動
preq_sem	[R]	セマフォ資源獲得(ポーリング)	―
ref_sem	[E]	セマフォの状態を参照します	―
snd_msg	[S]	メッセージを送信します	起動
isnd_msg	[S]	メッセージを送信します(ハンドラ専用)	―
rcv_msg	[S]	メッセージを受信します	起動
trcv_msg	[E]	メッセージを受信します(タイムアウトあり)	起動
prcv_msg	[S]	メッセージを受信します(待ちなし)	―
ref_mbx	[E]	メイルボックスの状態を参照します	―

割り込み管理機能

システムコール名		機能	スケジューラ
ret_int	[R]	割り込みハンドラから復帰します	起動
loc_cpu	[R]	割り込みとディスパッチを禁止します	―
unl_cpu	[R]	割り込みとディスパッチを許可します	起動

メモリプール管理機能

システムコール名		機能	スケジューラ
pget_blf	[E]	メモリブロックを獲得する(待ちなし)	−
rel_blf	[E]	メモリブロックを解放する	−
ref_mpf	[E]	固定長メモリプールの状態を参照する	−
pget_blk	[E]	メモリブロックを獲得する(待ちなし)	起動
rel_blk	[E]	メモリブロックを解放する	起動
ref_mpl	[E]	可変長メモリプールの状態を参照する	−

時間管理機能

システムコール名		機能	スケジューラ
set_tim	[S]	システムクロックを設定します	−
get_tim	[S]	システムクロックを参照します	−
dly_tsk	[S]	タスクの実行を一定時間遅延します	起動
act_cyc	[E]	周期起動ハンドラの活性制御を行います	−
ref_cyc	[E]	周期起動ハンドラの状態を参照します	−
ref_alm	[E]	アラームハンドラの状態を参照します	−

システム管理機能

システムコール名		機能	スケジューラ
get_ver	[R]	OS のバージョンを参照します	−

拡張機能

システムコール名		機能	スケジューラ
vrst_msg	[--]	メッセージをクリアする	−
vrst_blf	[--]	固定長メモリプールを解放する	−
vrst_blk	[--]	可変長メモリプールを解放する	−

3.2. エラーコード一覧

エラーコード	値	説明
E_OK	00000H(-H'0000)	正常終了
E_OBJ	0FFC1H(-H'003F)	オブジェクトの状態が不正
E_QOVR	0FFB7H(-H'0049)	キューイングまたはネストのオーバーフロー
E_TMOUT	0FFABH(-H'0055)	ポーリング失敗またはタイムアウト
E_RLWAI	0FFAAH(-H'0056)	待ち状態強制解除

3.3. アセンブリ言語インタフェース

アセンブリ言語でシステムコールを発行する場合、システムコールの呼び出し用マクロを使用します。

システム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システムコールのルーチンの実行を開始します。また、システムコールの呼び出し用マクロを使用せず直接システム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

Systemcall	INT No.	Parameter				Return Parameter		
		R0 (Function code)	R1	R2	A0	R0	R2	A0
sta_tsk	#63	H'00	-	stacd	tskid	ercd	-	-
ista_tsk	#62	H'24	-	stacd	tskid	ercd	-	-
ext_tsk	#58	-	-	-	-	-	-	-
ter_tsk	#63	H'02	-	-	tskid	ercd	-	-
dis_dsp	#60	-	-	-	-	ercd	-	-
ena_dsp	#63	H'0a	-	-	-	ercd	-	-
chg_pri	#63	H'04	-	tskpri	tskid	ercd	-	-
ichg_pri	#62	H'26	-	tskpri	tskid	ercd	-	-
rot_rdq	#63	H'06	-	tskpri	-	ercd	-	-
irot_rdq	#62	H'28	-	tskpri	-	ercd	-	-
rel_wai	#63	H'08	-	-	tskid	ercd	-	-
irel_wai	#62	H'2a	-	-	tskid	ercd	-	-
get_tid	#62	H'2c	-	-	-	ercd	-	tskid

Systemcall	INT No.	Parameter			Return Parameter		
		R0 (Function code)	A0	A1	R0	R2	A0
ref_tsk	#62	H'54	tskid	pk_rtsk	ercd	-	-

タスク付属同期機能

Systemcall	INT No.	Parameter				Return Parameter		
		R0 (Function code)	R1	R2	A0	R0	R2	A0
sus_tsk	#63	H'0c	-	-	tskid	ercd	-	-
isus_tsk	#62	H'2e	-	-	tskid	ercd	-	-
rsm_tsk	#63	H'0e	-	-	tskid	ercd	-	-
irsm_tsk	#62	H'30	-	-	tskid	ercd	-	-
slp_tsk	#63	H'10	-	-	-	ercd	-	-
wup_tsk	#63	H'12	-	-	tskid	ercd	-	-
iwup_tsk	#62	H'32	-	-	tskid	ercd	-	-
can_wup	#62	H'34	-	-	tskid	ercd	wupcnt	-

Systemcall	INT No.	Parameter					Return Parameter		
		R0 (Function code)	R1	R2	R3	A0	R0	R1	R2
tslp_tsk	#63	H'64	-	-	tmout	-	ercd	-	-

同期・通信機能

Systemcall	INT No.	Parameter				Return Parameter		
		R0 (Function code)	R1	R2	A0	R0	R2	A0
set_flg	#63	H'14	-	setptn	flgid	ercd	-	-
iset_flg	#62	H'36	-	setptn	flgid	ercd	-	-
clr_flg	#63	H'38	-	clrptn	flgid	ercd	-	-
wai_flg	#63	H'16	wfmode	waiptn	flgid	ercd	flgptn	-
pol_flg	#62	H'3a	wfmode	waiptn	flgid	ercd	flgptn	-
sig_sem	#63	H'18	-	-	semid	ercd	-	-
isig_sem	#62	H'3c	-	-	semid	ercd	-	-
wai_sem	#63	H'1a	-	-	semid	ercd	-	-
preq_sem	#62	H'3e	-	-	semid	ercd	-	-
snd_msg	#63	H'1c	-	pk_msg	mbxid	ercd	-	-
isnd_msg	#62	H'40	-	pk_msg	mbxid	ercd	-	-
rcv_msg	#63	H'1e	-	-	mbxid	ercd	pk_msg	-
prcv_msg	#62	H'42	-	-	mbxid	ercd	pk_msg	-

Systemcall	INT No.	Parameter					Return Parameter		
		R0 (Function code)	R1	R2	R3	A0	R0	R1	R2
twai_flg	#63	H'66	wfmode	waiptn	tmout	flgid	ercd	-	flgptn
twai_sem	#63	H'68	-	-	tmout	semid	ercd	-	-
trcv_msg	#63	H'6a	tmout	-	-	mbxid	ercd	-	pk_msg
trcv_msg*	#63	H'6a	tmout	-	-	mbxid	ercd	pk_msg	pk_msg

*: 32bit message size

Systemcall	INT No.	Parameter			Return Parameter		
		R0 (Function code)	A0	A1	R0	R2	A0
ref_flg	#62	H'56	flgid	pk_rflg	ercd	-	-
ref_sem	#62	H'58	semid	pk_rsem	ercd	-	-
ref_mbx	#62	H'5a	mbxid	pk_rmbx	ercd	-	-

Systemcall	INT No.	Parameter				Return Parameter			
		R0 (Function code)	R1	R3	A0	R0	R1	R2	R3
snd_msg*	#63	H'1c	pk_msg	pk_msg	mbxid	ercd	-	-	-
isnd_msg*	#62	H'40	pk_msg	pk_msg	mbxid	ercd	-	-	-
rcv_msg*	#63	H'1e	-	-	mbxid	ercd	pk_msg	pk_msg	-
prcv_msg*	#62	H'42	-	-	mbxid	ercd	pk_msg	pk_msg	-

*: 32bit message size

割り込み管理機能

Systemcall	INT No.	Parameter				Return Parameter		
		R0 (Function code)	R1	R2	A0	R0	R2	A0
ret_int	#61	-	-	-	-	-	-	-
loc_cpu	#59	-	-	-	-	ercd	-	-
unl_cpu	#63	H'20	-	-	-	ercd	-	-

メモリプール管理機能

Systemcall	INT No.	Parameter				Return Parameter			
		R0 (Function code)	R1	R3	A0	R0	R1	R2	R3
pget_blf	#62	H'4c	-	-	mpfid	ercd	p_blf	-	p_blf
rel_blf	#62	H'4e	p_blf	p_blf	mpfid	ercd	-	-	-
pget_blk	#63	H'50	-	-	-	ercd	p_blk	-	p_blk
rel_blk	#63	H'52	p_blk	p_blk	-	ercd	-	-	-

Systemcall	INT No.	Parameter			Return Parameter		
		R0 (Function code)	A0	A1	R0	R2	A0
ref_mpf	#62	H'5c	mpfid	pk_rmpf	ercd	-	-
ref_mpl	#62	H'5e	-	pk_rmpl	ercd	-	-

時間管理機能

Systemcall	INT No.	Parameter				Return Parameter		
		R0 (Function code)	R1	R2	A0	R0	R2	A0
set_tim	#62	H'44	-	-	pk_tim	ercd	-	-
get_tim	#62	H'46	-	-	pk_tim	ercd	-	-
dly_tsk	#63	H'22	-	dlytim	-	ercd	-	-
act_cyc	#62	H'48	-	cycact	cycno	ercd	-	-

Systemcall	INT No.	Parameter			Return Parameter		
		R0 (Function code)	A0	A1	R0	R2	A0
ref_cyc	#62	H'60	cycno	pk_rcyc	ercd	-	-
ref_alm	#62	H'62	almno	pk_ralm	ercd	-	-

システム管理機能

Systemcall	INT No.	Parameter				Return Parameter		
		R0 (Function code)	R1	R2	A0	R0	R2	A0
get_ver	#62	H'4a	-	-	pk_ver	ercd	-	-

拡張機能

Systemcall	INT No.	Parameter			Return Parameter		
		R0 (Function code)	A0	A1	R0	R2	A0
vrst_msg	#55	H'0	mbxid	-	ercd	-	-
vrst_blf	#55	H'2	mpfid	-	ercd	-	-
vrst_blk	#55	H'4	-	-	ercd	-	-

3.4. C 言語インタフェース

タスク管理機能

```
ER      ercd =  sta_tsk     (ID tskid, INT stacd);
ER      ercd =  ista_tsk    (ID tskid, INT stacd);
        void    ext_tsk     ();
ER      ercd =  ter_tsk     (ID tskid);
ER      ercd =  dis_dsp     ();
ER      ercd =  ena_dsp     ();
ER      ercd =  chg_pri     (ID tskid, PRI tskpri);
ER      ercd =  ichg_pri    (ID tskid, PRI tskpri);
ER      ercd =  rot_rdq     (PRI tskpri);
ER      ercd =  irot_rdq    (PRI tskpri);
ER      ercd =  rel_wai     (ID tskid);
ER      ercd =  irel_wai    (ID tskid);
ER      ercd =  get_tid     (ID *p_tskid);
ER      ercd =  ref_tsk     (T_RTSK *pk_rtsk, ID tskid);
```

タスク付属同期機能

```
ER      ercd =  sus_tsk     (ID tskid);
ER      ercd =  isus_tsk    (ID tskid);
ER      ercd =  rsm_tsk     (ID tskid);
ER      ercd =  irsm_tsk    (ID tskid);
ER      ercd =  slp_tsk     ();
ER      ercd =  tslp_tsk    (TMO tmout);
ER      ercd =  wup_tsk     (ID tskid);
ER      ercd =  iwup_tsk    (ID tskid);
ER      ercd =  can_wup     (INT *p_wupcnt, ID tskid)
```

同期・通信機能

```
ER  ercd =  set_flg   (ID flgid, UINT setptn);
ER  ercd =  iset_flg  (ID flgid, UINT setptn);
ER  ercd =  clr_flg   (ID flgid, UINT clrptn);
ER  ercd =  wai_flg   (UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode);
ER  ercd =  twai_flg  (UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode, TMO tmout);
ER  ercd =  pol_flg   (UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode);
ER  ercd =  ref_flg   (T_RFLG *pk_rflg, ID flgid);
ER  ercd =  sig_sem   (ID semid);
ER  ercd =  isig_sem  (ID semid);
ER  ercd =  wai_sem   (ID semid);
ER  ercd =  twai_sem  (ID semid, TMO tmout);
ER  ercd =  preq_sem  (ID semid);
ER  ercd =  ref_sem   (T_RSEM *pk_rsem, ID semid);
ER  ercd =  snd_msg   (ID mbxid, T_MSG *pk_msg);
ER  ercd =  isnd_msg  (ID mbxid, T_MSG *pk_msg);
ER  ercd =  rcv_msg   (T_MSG **ppk_msg, ID mbxid);
ER  ercd =  trcv_msg  (T_MSG **ppk_msg, ID mbxid, TMO tmout);
ER  ercd =  prcv_msg  (T_MSG **ppk_msg, ID mbxid);
ER  ercd =  ref_mbx   (T_RMBX *pk_rmbx, ID mbxid);
```

割り込み管理機能

```
    void    ret_int   ();
ER  ercd =  loc_cpu   ();
ER  ercd =  unl_cpu   ();
```

メモリプール管理機能

```
ER   ercd =  pget_blf   (VP *p_blf, ID mpfid);
ER   ercd =  rel_blf    (ID mpfid, VP blf);
ER   ercd =  ref_mpf    (T_RMPF *pk_rmpf, ID mpfid);
ER   ercd =  pget_blk   (VP *p_blk, ID mplid);
ER   ercd =  rel_blk    (ID mplid, VP blk);
ER   ercd =  ref_mpl    (T_RMPL *pk_rmpl, ID mplid);
```

時間管理機能

```
ER   ercd =  set_tim    (SYSTIME *pk_tim);
ER   ercd =  get_tim    (SYSTIME *pk_tim);
ER   ercd =  dly_tsk    (DLYTIME dlytim);
ER   ercd =  act_cyc    (HNO cycno, UINT cycact);
ER   ercd =  ref_cyc    (T_RCYC *pk_rcyc, HNO cycno);
ER   ercd =  ref_alm    (T_RALM *pk_ralm, HNO almno);
```

システム管理機能

```
ER   ercd =  get_ver    (T_VER *pk_ver);
```

拡張機能

```
ER   ercd =  vrst_msg   (ID mbxid);
ER   ercd =  vrst_blf   (ID mpfid);
ER   ercd =  vrst_blk   ();
```

3.5. データタイプ

```
typedef    signed char       B;          /* 符号付き 8 ビット整数 */
typedef    signed short      H;          /* 符号付き 16 ビット整数 */
typedef    signed long       W;          /* 符号付き 32 ビット整数 */
typedef    unsigned char     UB;         /* 符号なし 8 ビット整数 */
typedef    unsigned short    UH;         /* 符号なし 16 ビット整数 */
typedef    unsigned long     UW;         /* 符号なし 32 ビット整数 */
typedef    signed char       VB          /* データタイプが一致しないもの符号付き
                                            (8 ビットサイズ) */
typedef    signed short      VH;         /* データタイプが一致しないもの符号付き
                                            (16 ビットサイズ) */
typedef    signed long       VW;         /* データタイプが一致しないもの符号付き
                                            (32 ビットサイズ) */
typedef    void far          *VP;        /* データタイプが一致しないものへのポインタ */
typedef    void              (*FP)();    /* プログラムのスタートアドレス一般 */
typedef    H                 INT         /* 符号付き 16 ビット整数 */
typedef    UH                UINT;       /* 符号なし 16 ビット整数 */
typedef    H                 ID;         /* オブジェクト ID 番号 */
typedef    H                 PRI;        /* タスク優先度 */
typedef    H                 TMO;        /* タイムアウト */
typedef    H                 HNO;        /* ハンドラ番号 */
typedef    H                 ER;         /* エラーコード(符号付き整数) */
typedef    H                 DLYTIME;    /* 遅延時間 */
typedef    H                 CYCTIME;    /* 周期起動ハンドラの起動間隔*/
typedef    H                 BOOL_ID;    /* ブール値または ID 番号 */
typedef    void near * far   PT_MSG;      /* 16bit メッセージデータ */
typedef    void far * far    PT_MSG;     /* 32bit メッセージデータ */
```

3.6. 共通定数と構造体のパケット形式

```
----共通----
NADR        -1          /* アドレスやポインタ値が無効 */
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 */
taskstat:
    TTS_RUN             H'01    /* RUN */
    TTS_RDY             H'02    /* READY */
    TTS_WAI             H'04    /* WAIT */
    TTS_SUS             H'08    /* SUSPEND */
    TTS_WAS             H'0C    /* WAIT-SUSPEND */
    TTS_DMT             H'10    /* DORMANT */
tskwait:
    TTW_SLP             H'0001  /* slp_tsk,tslp_tsk による待ち */
    TTW_DLY             H'0002  /* dly_tsk による待ち */
    TTW_FLG             H'0010  /* wai_flg,twai_flg による待ち */
    TTW_SEM             H'0020  /* wai_sem,twai_sem による待ち */
    TTW_MBX             H'0040  /* rcv_msg,trcv_msg による待ち */
typedef struct t_rtsk {
    VP      exinf;      /* 拡張情報 */
    PRI     tskpri;     /* 現在の優先度 */
    UINT    tskstat;    /* タスクの状態 */
    UINT    tskwait;    /* タスクの待ち要因 */
    ID      wid;        /* タスクの待ちオブジェクト ID */
} T_RTSK;

----イベントフラグ関係----
wfmod:
    TWF_ANDW        H'0000  /* AND 待ち */
    TWF_ORW         H'0002  /* OR 待ち */
    TWF_CLR         H'0001  /* クリア指定 */
typedef struct t_rflg {
    VP              exinf;  /* 拡張情報 */
    BOOL_ID         wtsk;   /* 待ちタスクの有無 */
    UINT            flgptn; /* イベントフラグの初期値 */
} T_RFLG;

----セマフォ関係----
typedef struct t_rsem {
    VP              exinf;  /* 拡張情報 */
    BOOL_ID         wtsk;   /* 待ちタスクの有無 */
    INT             semcnt; /* 現在のセマフォカウントの値 */
} T_RSEM;

----メイルボックス関係----
typedef struct t_rmbx {
    VP              exinf;  /* 拡張情報 */
    BOOL_ID         wtsk;   /* 待ちタスクの有無 */
    PT_MSG          pk_msg; /* 次に受信されるメッセージ */
    INT             msgcnt; /* メイルボックスに格納されているメッセージ数 */
} T_RMBX;
```

```
----固定長メモリプール関係----
typedef struct t_rmpf {
    VP              exinf;    /* 拡張情報 */
    BOOL_ID         wtsk;     /* 待ちタスクの有無 */
    INT             frbcnt;   /* メモリブロック数 */
    INT             blksz;    /* メモリブロックサイズ */
} T_RMPF;

----可変長メモリプール関係----
typedef struct t_rmpl {
    VP              exinf;    /* 拡張情報 */
    BOOL_ID         wtsk;     /* 待ちタスクの有無 */
    INT             frsz;     /* 空き領域の合計サイズ */
    INT             maxsz;    /* 空き領域の最大サイズ */
} T_RMPL;

---- 時間管理関係 ----
typedef struct t_systime{
    H       utime;    /* 上位 16 ビット */
    UH      mtime;    /* 中位 16 ビット */
    UH      ltime;    /* 下位 16 ビット */
} SYSTIME, ALMTIME;
cycact:
    TCY_OFF         H'0000   /* 周期起動ハンドラが起動されない */
    TCY_ON          H'0001   /* 周期起動ハンドラが起動される */
    TCY_INI         H'0002   /* 周期カウンタが初期化される */

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

索引

T

V

W

あ

い

か

き

く

け

こ

し

す

せ

た

ち

て

に

は

ふ

ま

め

ゆ

ら

れ

わ

M3T-MR308 V.1.20 リファレンスマニュアル

Rev. 2.00
03.09.16
RJJ10J0130-0200Z

M3T-MR308 V.1.20
リファレンスマニュアル

ルネサス エレクトロニクス株式会社
神奈川県川崎市中原区下沼部1753 〒211-8668

RJJ10J0130-0200Z