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

RENESAS

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

HI7200/MP　V.1.00

ユーザーズマニュアル

ルネサスマイクロコンピュータ開発環境システム

ルネサス エレクトロニクス
www.renesas.com

Rev.1.01　2007.09

本資料ご利用に際しての留意事項

１．本資料は、お客様に用途に応じた適切な弊社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弊社または第三者の知的財産権その他の権利の実施、使用を許諾または保証するものではありません。
２．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など全ての情報の使用に起因する損害、第三者の知的財産権その他の権利に対する侵害に関し、弊社は責任を負いません。
３．本資料に記載の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、あるいはその他軍事用途の目的で使用しないでください。また、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、それらの定めるところにより必要な手続を行ってください。
４．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などの全ての情報は本資料発行時点のものであり、弊社は本資料に記載した製品または仕様等を予告なしに変更することがあります。弊社の半導体製品のご購入およびご使用に当たりましては、事前に弊社営業窓口で最新の情報をご確認頂きますとともに、弊社ホームページ(http://www.renesas.com)などを通じて公開される情報に常にご注意下さい。
５．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、弊社はその責任を負いません。
６．本資料に記載の製品データ、図、表などに示す技術的な内容、プログラム、アルゴリズムその他応用回路例などの情報を流用する場合は、流用する情報を単独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下さい。弊社は、適用可否に対する責任は負いません。
７．本資料に記載された製品は、各種安全装置や運輸・交通用、医療用、燃焼制御用、航空宇宙用、原子力、海底中継用の機器・システムなど、その故障や誤動作が直接人命を脅かしあるいは人体に危害を及ぼすおそれのあるような機器・システムや特に高度な品質・信頼性が要求される機器・システムでの使用を意図して設計、製造されたものではありません（弊社が自動車用と指定する製品を自動車に使用する場合を除きます）。これらの用途に利用されることをご検討の際には、必ず事前に弊社営業窓口へご照会下さい。なお、上記用途に使用されたことにより発生した損害等について弊社はその責任を負いかねますのでご了承願います。
８．第７項にかかわらず、本資料に記載された製品は、下記の用途には使用しないで下さい。これらの用途に使用されたことにより発生した損害等につきましては、弊社は一切の責任を負いません。
　１）生命維持装置。
　２）人体に埋め込み使用するもの。
　３）治療行為（患部切り出し、薬剤投与等）を行なうもの。
　４）その他、直接人命に影響を与えるもの。
９．本資料に記載された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、実装条件およびその他諸条件につきましては、弊社保証範囲内でご使用ください。弊社保証値を越えて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、弊社はその責任を負いません。
１０．弊社は製品の品質および信頼性の向上に努めておりますが、特に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。弊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、誤動作防止設計などの安全設計（含むハードウエアおよびソフトウエア）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エアは、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致します。
１１．本資料に記載の製品は、これを搭載した製品から剥がれた場合、幼児が口に入れて誤飲する等の事故の危険性があります。お客様の製品への実装後に容易に本製品が剥がれることがなきよう、お客様の責任において十分な安全設計をお願いします。お客様の製品から剥がれた場合の事故につきましては、弊社はその責任を負いません。
１２．本資料の全部または一部を弊社の文書による事前の承諾なしに転載または複製することを固くお断り致します。
１３．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弊社営業窓口までご照会下さい。

はじめに

本マニュアルは、SH2A-DUAL マイコン用リアルタイム OS 製品「HI7200/MP」の使用方法を述べたものです。ご使用になる前に本マニュアルを良く読んで理解してください。

■表記上の注意事項

◆ 数値のプリフィックス

“H’”と"0x"は16進数、”D’”は10進数を意味するプリフィックスです。プリフィックスの無い場合は10進数です。

◆ ディレクトリ区切り記号は”¥”です。

◆ “cfg ファイル”は、カーネルコンフィギュレーションファイルを意味します。

◆ “system.stack_size”など、ピリオドで繋がれた表記は、以下のいずれかの意味です。

(1) cfgファイルの設定項目
(2) 構造体の要素
(3) レジスタ等の特定ビット

◆ [メニュー->メニューオプション]

“->”はメニューオプションを示します。(例[ファイル->保存])

◆ <RTOS_INST>

<RTOS_INST>は、ヘッダファイルやコンフィギュレータなどがインストールされているシステムディレクトリを示します。システムディレクトリの場所は、インストール時にユーザが指定したディレクトリ下の”x.yy.zzww”ディレクトリです。なお、x, yy, zzは製品バージョン、wwは内部識別番号で”00”～”99”の数値です。以下に例を示します。

・製品バージョンが”V.1.01 Release 02”の場合：”1.01.02ww”
・製品バージョンが”V.2.11 Release 13”の場合：”2.11.13ww”

◆ <SAMPLE_INST>

< SAMPLE_INST>は、本製品のサンプルが格納されているディレクトリを示します。本製品のセットアップ時に、ユーザがサンプルを格納するディレクトリを指定することができます。

◆ $(xxxx)

$(xxxx)は、High-performance Embedded Workshopのカスタムプレースホルダを示します。$(RTOS_INST)は<RTOS_INST>を示すカスタムプレースホルダ、$(SAMPLE_INST)は<SAMPLE_INST>を示すカスタムプレースホルダです。

■商標等

- TRON は、"The Real-time Operating system Nucleus" の略称です。ITRON は、"Industrial TRON" の略称です。μITRON は、"Micro Industrial TRON" の略称です。TRON、ITRON、およびμITRON は、コンピュータの仕様に対する名称であり、特定の商品ないし商品群を指すものではありません。μITRON4.0 仕様は、(社)トロン協会が策定したオープンなリアルタイムカーネル仕様です。μITRON4.0 仕様の仕様書は、(社)トロン協会ホームページ(http://www.assoc.tron.org/)から入手が可能です。μITRON 仕様の著作権は(社)トロン協会に属しています。
- Microsoft® Windows® 2000 operating system, Microsoft® Windows® XP operating system は、米国 Microsoft Corporation の米国およびその他の国における登録商標です。
- SuperH™ は、(株)ルネサステクノロジの商標です。
- その他、本書で登場するシステム名、製品名は各社の登録商標または商標です。

■ホームページ

弊社ホームページにて各種サポート情報をお知らせしておりますので、あわせてご利用ください。

http://japan.renesas.com/homepage.jsp

目次

1. 本マニュアルの構成 1

2. インストール 3

2.1 インストール方法 3

2.2 ディレクトリ構成 3

2.2.1 システムディレクトリ(<RTOS_INST>) 3

2.2.2 サンプルディレクトリ(<SAMPLE_INST>) 4

3. 概要 5

3.1 概要 5

3.2 特長 5

3.2.1 カーネル 5

3.2.2 RPC(リモートプロシージャコール)ライブラリ 5

3.2.3 OAL 5

3.2.4 スピンロックライブラリ 5

3.2.5 IPI機能 5

3.2.6 キャッシュサポートライブラリ 6

3.2.7 サンプルプログラム 6

3.2.8 コンフィギュレータ 6

3.2.9 デバッギングエクステンション(オプション製品) 6

3.3 マルチコア 7

3.4 動作環境 9

4. カーネル入門 11

4.1 カーネルの動作原理 11

4.2 サービスコール 13

4.3 CPUID 13

4.4 オブジェクト 14

4.4.1 概要 14

4.4.2 ID番号 14

4.4.3 ID名称によるオブジェクトの指定 15

4.5 タスク 16

4.5.1 タスクの状態 16

4.5.2 タスクのスケジューリング(優先度とレディキュー) 18

4.5.3 タスクの待ち行列 19

4.5.4 タスクのスタック 20

4.5.5 共有スタック機能 21

4.6 システムの状態 22

RENESAS

4.6.1 タスクコンテキストと非タスクコンテキスト 22
4.6.2 ディスパッチ禁止/許可状態 22
4.6.3 CPUロック/ロック解除状態 23
4.6.4 ディスパッチ保留状態 23
4.7 処理の単位と優先順位 24
4.8 割込み 26
4.8.1 割込みハンドラの種類 26
4.8.2 割込み制御方法(SRレジスタのIMASKビット) 28
4.8.3 サービスコールの制限 29
4.9 CPU 例外 30
4.9.1 CPU例外ハンドラの種類 30
4.9.2 予約例外 30
5. カーネルの機能 31
5.1 タスク管理機能 31
5.2 タスク付属同期機能 34
5.3 タスク付属イベントフラグ 36
5.4 タスク例外処理 37
5.5 セマフォ 39
5.5.1 優先度逆転問題 41
5.6 イベントフラグ 42
5.7 データキュー 44
5.8 メールボックス 46
5.9 ミューテックス 48
5.9.1 ベース優先度と現在優先度 49
5.10 メッセージバッファ 50
5.11 固定長メモリプール 52
5.12 可変長メモリプール 54
5.12.1 空き領域の断片化とその対策 55
5.12.2 可変長メモリプールの管理方法 57
5.13 時間管理機能 58
5.13.1 タスクのタイムアウト 58
5.13.2 タスクの遅延 59
5.13.3 タイマの停止と再開 59
5.13.4 周期ハンドラ 60
5.13.5 アラームハンドラ 62
5.13.6 オーバーランハンドラ 63
5.13.7 時間の精度 64
5.13.8 注意事項 66
5.14 システム状態管理機能 67
5.14.1 システム状態管理 67
5.14.2 初期化関連のサービスコール 68
5.14.3 システムダウン(vsys_dwn, ivsys_dwn) 68
5.14.4 サービスコールトレース機能 69

RENESAS

5.15 割込み管理機能 71
5.16 拡張サービスコール 71
5.17 システム構成管理機能 72
5.18 プロファイル管理機能 73
5.19 カーネルのアイドリング 74

6. カーネルサービスコール 75
6.1 呼び出し形式 75
6.2 ヘッダファイル 75
6.3 基本データ型 76
6.4 サービスコール呼び出し前後のレジスタ保証規則 77
6.5 サービスコールのリターン値とエラーコード 78
6.5.1 概要 78
6.5.2 パラメータチェック機能 78
6.5.3 スタックオーバフローの検出 78
6.5.4 メインエラーコードとサブエラーコード 78
6.6 システム状態とサービスコール 79
6.6.1 タスクコンテキストと非タスクコンテキスト 79
6.6.2 CPUロック状態 80
6.6.3 ディスパッチ禁止状態 80
6.6.4 ノーマルCPU例外ハンドラ 80
6.6.5 タスクコンテキストでSR.IMASKを0以外に変更している場合 80
6.7 ID 番号 81
6.7.1 概要 81
6.7.2 ID番号に関する関数マクロ 81
6.8 サービスコールの振る舞い 82
6.8.1 リモートサービスコールとローカルサービスコール 82
6.8.2 ローカルサービスコールの振る舞い 83
6.8.3 リモートサービスコールの振る舞い 83
6.8.4 リモートサービスコールの注意事項 84
6.9 μITRON4.0 仕様外の仕様 84
6.10 サービスコールの説明形式 85
6.11 タスク管理機能 87
6.11.1 タスクの生成 89
(cre_tsk, icre_tsk) 89
(acre_tsk, iacre_tsk：ID 番号自動割付け) 89
(vscr_tsk, ivscr_tsk)(スタティックスタック使用) 89
6.11.2 タスクの削除(del_tsk) 92
6.11.3 タスクの起動(act_tsk, iact_tsk) 93
6.11.4 タスクの起動要求のキャンセル(can_act, ican_act) 95
6.11.5 タスクの起動(起動コード指定)(sta_tsk, ista_tsk) 96
6.11.6 自タスクの終了(ext_tsk) 97
自タスクの終了と削除(exd_tsk) 97
6.11.7 タスクの強制終了(ter_tsk) 98
6.11.8 タスク優先度の変更(chg_pri, ichg_pri) 99

6.11.9 タスク優先度の参照(get_pri, iget_pri) ... 100
6.11.10 タスクの状態参照(ref_tsk, iref_tsk) ... 101
6.11.11 タスクの状態参照(簡易版)(ref_tst, iref_tst) ... 104
6.11.12 タスク実行モードの変更(vchg_tmd) ... 106

6.12 タスク付属同期機能 ... 107

6.12.1 起床待ち(slp_tsk, tslp_tsk) ... 109
6.12.2 タスクの起床(wup_tsk, iwup_tsk) ... 110
6.12.3 タスク起床要求のキャンセル(can_wup, ican_wup) ... 111
6.12.4 待ち状態の強制解除(rel_wai, irel_wai) ... 112
6.12.5 強制待ち状態への移行(sus_tsk, isus_tsk) ... 113
6.12.6 強制待ち状態からの再開(rsm_tsk, irsm_tsk) ... 115
強制待ち状態からの強制再開(frsm_tsk, ifrsm_tsk) ... 115
6.12.7 タスク遅延(dly_tsk) ... 116
6.12.8 タスク付属イベントフラグのセット(vset_tfl, ivset_tfl) ... 117
6.12.9 タスク付属イベントフラグのクリア(vclr_tfl, ivclr_tfl) ... 118
6.12.10 タスク付属イベントフラグ待ち(vwai_tfl, vpol_tfl, vtwai_tfl) ... 119

6.13 タスク例外処理機能 ... 120

6.13.1 タスク例外処理ルーチンの定義(def_tex, idef_tex) ... 121
6.13.2 タスク例外処理の要求(ras_tex, iras_tex) ... 123
6.13.3 タスク例外処理の禁止(dis_tex) ... 124
6.13.4 タスク例外処理の許可(ena_tex) ... 125
6.13.5 タスク例外禁止状態の参照(sns_tex) ... 126
6.13.6 タスク例外処理の状態参照(ref_tex, iref_tex) ... 127

6.14 同期・通信(セマフォ)機能 ... 128

6.14.1 セマフォの生成(cre_sem, icre_sem) ... 129
(acre_sem, iacre_sem：ID 番号自動割付け) ... 129
6.14.2 セマフォの削除(del_sem) ... 131
6.14.3 セマフォ資源の返却(sig_sem, isig_sem) ... 132
6.14.4 セマフォ資源の獲得(wai_sem, pol_sem, ipol_sem, twai_sem) ... 133
6.14.5 セマフォの状態参照(ref_sem, iref_sem) ... 134

6.15 同期・通信(イベントフラグ)機能 ... 135

6.15.1 イベントフラグの生成(cre_flg, icre_flg) ... 136
(acre_flg, iacre_flg：ID 番号自動割付け) ... 136
6.15.2 イベントフラグの削除(del_flg) ... 138
6.15.3 イベントフラグのセット(set_flg, iset_flg) ... 139
6.15.4 イベントフラグのクリア(clr_flg, iclr_flg) ... 140
6.15.5 イベントフラグ待ち(wai_flg, pol_flg, ipol_flg, twai_flg) ... 141
6.15.6 イベントフラグの状態参照(ref_flg, iref_flg) ... 143

6.16 同期・通信(データキュー)機能 ... 144

6.16.1 データキューの生成(cre_dtq, icre_dtq) ... 145
(acre_dtq, iacre_dtq：ID 番号自動割付け) ... 145
6.16.2 データキューの削除(del_dtq) ... 147
6.16.3 データキューへの送信(snd_dtq,psnd_dtq,ipsnd_dtq,tsnd_dtq, fsnd_dtq, ifsnd_dtq) ... 148
6.16.4 データキューからの受信(rcv_dtq, prcv_dtq, trcv_dtq) ... 150
6.16.5 データキューの状態参照(ref_dtq, iref_dtq) ... 152

6.17　同期･通信(メールボックス)機能 153
6.17.1　メールボックスの生成(cre_mbx, icre_mbx) 154
(acre_mbx, iacre_mbx：ID 番号自動割付け) 154
6.17.2　メールボックスの削除(del_mbx) 156
6.17.3　メールボックスへの送信(snd_mbx, isnd_mbx) 157
6.17.4　メールボックスからの受信(rcv_mbx, prcv_mbx, iprcv_mbx, trcv_mbx) 159
6.17.5　メールボックスの状態参照(ref_mbx, iref_mbx) 161
6.18　拡張同期・通信(ミューテックス)機能 163
6.18.1　ミューテックスの生成(cre_mtx) 164
(acre_mtx：ID 番号自動割付け) 164
6.18.2　ミューテックスの削除(del_mtx) 166
6.18.3　ミューテックスのロック(loc_mtx, ploc_mtx, tloc_mtx) 167
6.18.4　ミューテックスのロック解除(unl_mtx) 168
6.18.5　ミューテックスの状態参照(ref_mtx) 169
6.19　拡張同期・通信(メッセージバッファ)機能 170
6.19.1　メッセージバッファの生成(cre_mbf, icre_mbf) 171
(acre_mbf, iacre_mbf：ID 番号自動割付け) 171
6.19.2　メッセージバッファの削除(del_mbf) 173
6.19.3　メッセージバッファへの送信(snd_mbf, psnd_mbf, ipsnd_mbf, tsnd_mbf) 174
6.19.4　メッセージバッファからの受信(rcv_mbf, prcv_mbf, trcv_mbf) 176
6.19.5　メッセージバッファの状態参照(ref_mbf, iref_mbf) 178
6.20　メモリプール管理(固定長メモリプール)機能 179
6.20.1　固定長メモリプールの生成(cre_mpf, icre_mpf) 180
(acre_mpf, iacre_mpf：ID 番号自動割付け) 180
6.20.2　固定長メモリプールの削除(del_mpf) 183
6.20.3　固定長メモリブロックの獲得(get_mpf, pget_mpf, ipget_mpf, tget_mpf) 184
6.20.4　固定長メモリブロックの返却(rel_mpf, irel_mpf) 186
6.20.5　固定長メモリプールの状態参照(ref_mpf, iref_mpf) 187
6.21　メモリプール管理(可変長メモリプール)機能 188
6.21.1　可変長メモリプールの生成(cre_mpl, icre_mpl) 189
(acre_mpl, iacre_mpl：ID 番号自動割付け) 189
6.21.2　可変長メモリプールの削除(del_mpl) 193
6.21.3　可変長メモリブロックの獲得(get_mpl, pget_mpl, ipget_mpl, tget_mpl) 194
6.21.4　可変長メモリブロックの返却(rel_mpl, irel_mpl) 196
6.21.5　可変長メモリプールの状態参照(ref_mpl, iref_mpl) 197
6.22　時間管理機能(システム時刻管理) 198
6.22.1　システム時刻の設定(set_tim, iset_tim) 200
6.22.2　システム時刻の参照(get_tim, iget_tim) 201
6.22.3　タイムティックの供給(isig_tim) 202
6.22.4　タイマの停止(vstp_tmr) 203
6.22.5　タイマの再開(vrst_tmr, ivrst_tmr) 204
6.22.6　タイマ状態の参照（vsns_tmr) 205
6.23　時間管理機能(周期ハンドラ) 206
6.23.1　周期ハンドラの生成(cre_cyc, icre_cyc) 207
(acre_cyc, iacre_cyc：ID 番号自動割付け) 207
6.23.2　周期ハンドラの削除(del_cyc) 209

6.23.3 周期ハンドラの動作開始(sta_cyc, ista_cyc) 210
6.23.4 周期ハンドラの動作停止(stp_cyc, istp_cyc) 211
6.23.5 周期ハンドラの状態参照(ref_cyc, iref_cyc) 212
6.24 時間管理機能(アラームハンドラ) 213
6.24.1 アラームハンドラの生成(cre_alm, icre_alm) 214
(acre_alm, iacre_alm：ID 番号自動割付け) 214
6.24.2 アラームハンドラの削除(del_alm) 216
6.24.3 アラームハンドラの動作開始(sta_alm, ista_alm) 217
6.24.4 アラームハンドラの動作停止(stp_alm, istp_alm) 218
6.24.5 アラームハンドラの状態参照(ref_alm, iref_alm) 219
6.25 時間管理機能(オーバーランハンドラ) 220
6.25.1 オーバーランハンドラの定義(def_ovr) 221
6.25.2 オーバーランハンドラの動作開始(sta_ovr, ista_ovr) 222
6.25.3 オーバーランハンドラの動作停止(stp_ovr, istp_ovr) 223
6.25.4 オーバーランハンドラの状態参照(ref_ovr, iref_ovr) 224
6.26 システム状態管理機能 225
6.26.1 タスクの優先順位の回転(rot_rdq, irot_rdq) 226
6.26.2 実行状態のタスクIDの参照(get_tid, iget_tid) 227
6.26.3 CPUロック状態への移行(loc_cpu, iloc_cpu) 228
6.26.4 CPUロック状態の解除(unl_cpu, iunl_cpu) 229
6.26.5 ディスパッチの禁止(dis_dsp) 230
6.26.6 ディスパッチの許可(ena_dsp) 231
6.26.7 コンテキストの参照(sns_ctx) 232
6.26.8 CPUロック状態の参照(sns_loc) 233
6.26.9 ディスパッチ禁止状態の参照(sns_dsp) 234
6.26.10 ディスパッチ保留状態の参照(sns_dpn) 235
6.26.11 カーネルの起動(vsta_knl, ivsta_knl) 236
6.26.12 リモートサービスコール環境の初期化(vini_rmt) 237
6.26.13 システムダウン(vsys_dwn, ivsys_dwn) 239
6.26.14 トレースの取得(vget_trc, ivget_trc) 240
6.26.15 割込みハンドラの開始をトレースに取得(ivbgn_int) 241
6.26.16 割込みハンドラの終了をトレースに取得(ivend_int) 242
6.27 割込み管理機能 243
6.27.1 割込みハンドラの定義(def_inh, idef_inh) 244
6.27.2 割込みマスクの変更(chg_ims, ichg_ims) 247
6.27.3 割込みマスクの参照(get_ims, iget_ims) 248
6.28 サービスコール管理機能 249
6.28.1 拡張サービスコールの定義(def_svc, idef_svc) 250
6.28.2 サービスコールの呼び出し(cal_svc, ical_svc) 251
6.29 システム構成管理機能 252
6.29.1 CPU例外ハンドラの定義(def_exc, idef_exc) 253
6.29.2 CPU例外(TRAPA命令例外)ハンドラ定義(vdef_trp, ivdef_trp) 256
6.29.3 コンフィギュレーション情報の参照(ref_cfg, iref_cfg) 259
6.29.4 バージョン情報の参照(ref_ver, iref_ver) 261
6.30 プロファイル管理機能 263
6.30.1 プロファイルカウンタの参照(vref_prf, ivref_prf) 264

6.30.2 プロファイルカウンタのクリア(vclr_prf, ivclr_prf)....265
6.31 マクロ....266
6.31.1 定数マクロ....266
6.31.2 カーネル構成マクロ....270
6.31.3 itron.hで定義されている関数マクロ....273
6.31.4 kernel.hで定義されている関数マクロ....273
6.32 ディレクトリ・ファイル構成....275
6.33 ライブラリのビルド(ソースコード付き製品の場合のみ)....276
7. RPC ライブラリ....277
7.1 概要....277
7.2 RPC の動作概要....278
7.3 サーバ....279
7.3.1 サーバID....279
7.3.2 ファンクションID....279
7.3.3 サーバタスク....279
7.3.4 サーバスタブとサーバ関数....279
7.3.5 クライアントスタブ....279
7.3.6 サーバの競合....279
7.4 同期モードと非同期モード....280
7.5 パラメータの受け渡し....280
7.5.1 特長....280
7.5.2 IOVEC構造体....280
7.5.3 サーバパラメータ領域....281
7.5.4 RPCコールに必要なサーバパラメータ領域のサイズ....282
7.5.5 パラメータのコピー方法....282
7.5.6 応用例....283
7.6 RPC が使用する OS リソース....284
7.6.1 タスク....284
7.6.2 OAL_GetMemory()....284
7.6.3 IPI....284
7.6.4 スピンロックライブラリ....284
7.7 提供ファイル....285
7.8 ライブラリのビルド(ソースコード付き製品の場合のみ)....285
7.9 システムのビルド....286
7.9.1 カーネルのコンフィギュレーション....286
7.9.2 IPIのコンフィギュレーション....286
7.9.3 システムのビルド....286
7.10 API 関数....287
7.10.1 ヘッダファイル....287
7.10.2 基本データタイプ....287
7.10.3 RPCライブラリの初期化(rpc_init)....288
7.10.4 RPCライブラリの終了(rpc_shutdown)....290
7.10.5 ダイナミックサーバの開始(rpc_start_server)....291
7.10.6 スタティックサーバの開始(rpc_start_server_with_paramarea)....293

7.10.7 サーバの停止(rpc_stop_server)....295
7.10.8 サーバとの接続(rpc_connect)....296
7.10.9 サーバとの接続解除(rpc_disconnect)....297
7.10.10 サーバ関数呼び出し (rpc_call)....298
7.10.11 サーバ関数呼び出し(データ転送コールバック) (rpc_call_copycbk)....302
7.10.12 サーバプロパティの取得 (rpc_get_server_properties)....304
7.11 **スタブ**....305
7.11.1 サーバスタブ....305
7.11.2 クライアントスタブ....307
7.12 **サーバ停止コールバック関数**....308
7.13 CopyCbk1, CopyCbk2 **コールバック関数**....308
8. OAL....309
8.1 **概要**....309
8.2 **提供ファイル**....309
8.3 **コンフィギュレーションとビルド**....310
8.3.1 コンフィギュレーション....310
8.3.2 ビルド....310
8.4 API **関数**....311
8.4.1 ヘッダファイル....311
8.4.2 基本データタイプ....311
8.4.3 リターン値....312
8.4.4 OALの初期化(OAL_Init)....313
8.4.5 OALの終了(OAL_Shutdown)....313
8.4.6 タスクプリエンプションの禁止(OAL_DisablePreempt)....313
8.4.7 タスクプリエンプションの許可(OAL_EnablePreempt)....313
8.4.8 タスクプリエンプション状態の確認(OAL_IsDisablePreempt)....314
8.4.9 自タスク待機可否の確認(OAL_CanWait)....314
8.4.10 コンテキスト種別の確認(OAL_IsNotTaskLevel)....314
8.4.11 プロセッサの割込みマスクの確認(OAL_IsMaskInterrupt)....314
8.4.12 タスクの生成(OAL_CreateTask)....315
8.4.13 タスクの起動(OAL_ActivateTask)....315
8.4.14 自タスクの終了・削除(OAL_DestroyTask)....316
8.4.15 自タスク識別情報の取得(OAL_GetTaskID)....316
8.4.16 自タスクの待機(OAL_SleepTask)....316
8.4.17 タスク待機の解除(OAL_WakeupTask)....316
8.4.18 メモリの取得(OAL_GetMemory)....317
8.4.19 メモリの解放(OAL_ReleaseMemory)....317
9. スピンロックライブラリ....319
9.1 **概要**....319
9.2 **基本的な使用方法**....319
9.3 **スピンロックの振る舞いと注意事項**....320
9.3.1 同一CPU内での排他制御とデッドロック....320
9.3.2 ロック取得期間の問題....321

9.4 3 種類のスピンロック 322
9.5 ノーマルロックと RW ロックのロック変数 323
9.5.1 ロック変数の実体 323
9.5.2 ロック変数を置く RAM 323
9.6 提供ファイル 324
9.7 ライブラリのビルド 324
9.8 システムのビルド 324
9.9 API 関数 325
9.9.1 ヘッダファイル 325
9.9.2 基本データタイプ 325
9.9.3 注意 325
9.10 ノーマルロック 326
9.10.1 ノーマルロック変数の初期化(SPIN_InitLock) 326
9.10.2 ノーマルロック取得(SPIN_Lock) 326
9.10.3 ノーマルロック取得試行(SPIN_TryLock) 327
9.10.4 ノーマルロック解除(SPIN_Unlock) 327
9.10.5 ノーマルロック状態のチェック(SPIN_IsLocked) 328
9.11 RW ロック 329
9.11.1 RWロック変数の初期化(SPIN_InitRWLock) 329
9.11.2 リードロック獲得(SPIN_ReadLock) 329
9.11.3 リードロック獲得試行(SPIN_ReadTryLock) 330
9.11.4 リードロック解除(SPIN_ReadUnlock) 330
9.11.5 リードロック状態のチェック(SPIN_IsReadLocked) 331
9.11.6 ライトロック獲得(SPIN_WriteLock) 331
9.11.7 ライトロック獲得試行(SPIN_WriteTryLock) 332
9.11.8 ライトロック解除(SPIN_WriteUnlock) 332
9.11.9 ライトロック状態のチェック(SPIN_IsWriteLocked) 333
9.12 セマフォロック 334
9.12.1 セマフォレジスタの初期化(SPIN_InitSemLock) 334
9.12.2 セマフォロック獲得(SPIN_SemLock) 334
9.12.3 セマフォロック獲得試行(SPIN_SemTryLock) 335
9.12.4 セマフォロック解除(SPIN_SemUnlock) 335
10. IPI 337
10.1 概要 337
10.2 IPI の構成 337
10.3 ポート ID 337
10.4 動作概要 338
10.5 注意事項 338
10.6 提供ファイル 338
10.7 コンフィギュレーションとビルド 339
10.7.1 コンフィギュレーション 339
10.7.2 ビルド 341
10.8 API 関数 342

10.8.1 ヘッダファイル 342
10.8.2 基本データタイプ 342
10.8.3 IPIの初期化(IPI_init) 342
10.8.4 IPIポートの生成(IPI_create) 343
10.8.5 IPIポートの削除(IPI_delete) 344
10.8.6 IPIポートへの送信(IPI_send) 345
10.9 プロセッサ間割込みハンドラ 346
10.10 コールバック関数 346
11. SH2A-DUAL 用キャッシュサポートライブラリ 347
11.1 概要 347
11.2 留意事項 347
11.3 ディレクトリ・ファイル構成 348
11.4 ライブラリのビルド 348
11.5 システムのビルド 348
11.6 API 関数 349
11.6.1 ヘッダファイル 349
11.6.2 基本データタイプ 349
11.6.3 キャッシュの初期化(sh2adual_ini_cac) 350
11.6.4 キャッシュのクリア(sh2adual_clr_cac) 351
11.6.5 オペランドキャッシュのフラッシュ(sh2adual_fls_cac) 353
11.6.6 キャッシュの無効化(sh2adual_inv_cac) 354
12. アプリケーションプログラムの記述方法 355
12.1 FPU について 355
12.2 タスク 355
12.3 タスク例外処理ルーチン 357
12.4 拡張サービスコールルーチン 358
12.5 割込みハンドラ 359
12.5.1 割込みハンドラの種類 359
12.5.2 レジスタバンク 359
12.5.3 ノーマル割込みハンドラ 360
12.5.4 ダイレクト割込みハンドラ 362
12.6 CPU 例外ハンドラ(TRAPA 例外を含む) 364
12.6.1 CPU例外ハンドラの種類 364
12.6.2 ノーマルCPU例外ハンドラ 364
12.6.3 ダイレクトCPU例外ハンドラ 366
12.7 タイムイベントハンドラ 367
12.8 初期化ルーチン 369
12.9 タイマドライバ 370
12.9.1 tdr_ini_tmr()：タイマの初期化 371
12.9.2 tdr_int_tmr()：タイマ割込み処理 373
12.9.3 tdr_stp_tmr()：タイマの停止 374
12.9.4 tdr_rst_tmr()：タイマの再開 375

RENESAS

12.10 システムダウンルーチン....376

13. ロードモジュール作成手順概要....379

13.1 概要....379

14. コンフィギュレータ(cfg72mp)....381

14.1 cfg ファイル内の表現形式....381
14.1.1 コメント文....381
14.1.2 文の終わり....381
14.1.3 定義文....381
14.1.4 数値....382
14.1.5 シンボル....383
14.1.6 外部参照名....383
14.1.7 注意....383
14.2 デフォルト cfg ファイル....383
14.3 cfg ファイルの定義項目....384
14.3.1 説明形式....384
14.3.2 システム定義(system)....385
14.3.3 最大ID定義(maxdefine)....392
14.3.4 デフォルトタスクスタック用領域定義(memstk)....396
14.3.5 デフォルトデータキュー用領域定義(memdtq)....396
14.3.6 デフォルトメッセージバッファ用領域定義(memmbf)....397
14.3.7 デフォルト固定長メモリプール用領域定義(memmpf)....398
14.3.8 デフォルト可変長メモリプール用領域定義(memmpl)....399
14.3.9 システムクロック定義(clock)....400
14.3.10 リモートサービスコール環境定義(remote_svc)....402
14.3.11 タスク定義(task[])....404
14.3.12 スタティックスタック領域定義 (static_stack[])....408
14.3.13 セマフォ定義(semaphore[])....410
14.3.14 イベントフラグ定義(flag[])....412
14.3.15 データキュー定義(dataqueue[])....414
14.3.16 メールボックス定義(mailbox[])....416
14.3.17 ミューテックス定義(mutex[])....418
14.3.18 メッセージバッファ定義(message_buffer[])....419
14.3.19 固定長メモリプール定義(memorypool[])....422
14.3.20 可変長メモリプール定義(variable_memorypool[])....425
14.3.21 周期ハンドラ定義(cyclic_hand[])....428
14.3.22 アラームハンドラ定義(alarm_hand[])....430
14.3.23 オーバーランハンドラ定義(overrun_hand)....432
14.3.24 拡張サービスコールルーチン定義(extend_svc[])....433
14.3.25 割込みハンドラ、CPU例外ハンドラ定義(interrupt_vector[])....434
14.3.26 初期化ルーチン定義(init_routine[])....436
14.3.27 サービスコール定義(service_call)....437
14.4 コンフィギュレータの実行....439
14.4.1 コンフィギュレータ概要....439
14.4.2 環境設定....440

14.4.3 コンフィギュレータ実行に必要な入力ファイル 440
14.4.4 cfg72mpが出力するファイル 440
14.4.5 コンフィギュレータ起動方法 441
14.4.6 コマンドオプション 441
14.4.7 注意事項 441
14.5 エラーメッセージ 442
14.5.1 エラー形式とエラーレベル 442
14.5.2 メッセージ一覧 442
14.6 ID 名称ヘッダファイル 445
14.6.1 概要 445
14.6.2 ID名称ヘッダファイルの種類 445
14.7 kernel_macro.h 446
15. GUI コンフィギュレータ 449
16. サンプルプログラム 451
16.1 対象ハードウェア 451
16.2 ディレクトリ構成 453
16.3 スタートアップ 454
16.3.1 概要 454
16.3.2 リセットベクタ(cpuid1¥reset¥reset.src) 458
16.3.3 CPUID#1のリセット本体プログラム(cpuid1¥reset¥resetprg1.c) 465
16.3.4 共通ハードウェアおよびCPUID#1リソースの初期化関数
HardwareSetup_CPUID1() (cpuid1¥reset¥hwsetup1.c) 468
16.3.5 CPUID#2の仮想リセットベクタテーブル(cpuid2¥reset¥vreset.src) 469
16.3.6 CPUID#2のリセット本体プログラム(cpuid2¥reset¥resetprg2.c) 471
16.3.7 CPUID#1の初期起動タスクInitTask1() (cpuid1¥init¥init_task1.c) 474
16.3.8 CPUID#2の初期起動タスクInitTask2() (cpuid2¥init¥init_task2.c) 477
16.3.9 両CPUのスタートアップフェーズの同期メカニズム 480
16.4 RPC 使用例 483
16.4.1 概要 483
16.4.2 RPCサーバの登録(CPUID#2) 485
16.4.3 SampleAdd() 485
16.4.4 SampleStrlen() 487
16.4.5 SampleSort1(), SampleSort2() 489
16.4.6 SampleMemcopy() 489
16.4.7 SampleCreateTask(), SampleKillTask(), SampleRefTaskState() 489
16.4.8 RPC呼び出し例(CPUID#1) 491
16.4.9 サーバの初期化と終了(CPUID#2) 491
16.4.10 クライアントの初期化と終了(CPUID#1) 493
16.4.11 RPCライブラリの初期化(rpc_init()コール) 493
16.5 リモートサービスコール使用例 494
16.6 タイマドライバ 495
16.7 標準ライブラリ 496
16.7.1 概要 496
16.7.2 低水準インタフェースルーチン 497

16.7.3 標準ライブラリ環境の初期化(_INIT_LOWLEVEL(), _INIT_OTHERLIB()) 497
16.7.4 セクション初期化(_INITSCT()) 498
16.7.5 標準ライブラリのコンフィギュレーション(lowsrc_config.h) 499
16.7.6 ソースコード 500
16.8 ダミーオブジェクト 508
16.8.1 ダミープログラム 508
16.8.2 その他のダミーオブジェクト 509
16.9 I/O レジスタ定義、周辺クロック定義、kernel_intspec.h 510
16.10 カーネルオブジェクト一覧 511
16.10.1 タスク 511
16.10.2 その他のオブジェクト 512
16.11 cfg ファイル 514
16.11.1 CPUID#1 (cpuid1¥cfg_out¥sample.cfg) 514
16.11.2 CPUID#2 (cpuid2¥cfg_out¥sample.cfg) 522
16.12 IPI ポート 530
16.13 他ハードウェアへのポーティング 531

17. ビルド 533
17.1 カスタムプレースホルダ”$(RTOS_INST)”の設定 533
17.2 ワークスペースに cfg72mp をカスタムビルドフェーズとして登録する 534
17.2.1 ファイル拡張子の登録 534
17.2.2 cfg72mpカスタムビルドフェーズの作成 536
17.2.3 ビルドフェーズの設定 541
17.3 CPU 割込み仕様定義ファイル(kernel_intspec.h)の作成 542
17.3.1 IBNRレジスタアドレス(INTSPEC_IBNR_ADR1, INTSPEC_IBNR_ADR2) 544
17.3.2 レジスタバンク利用不可のベクタ番号(INTSPEC_NOBANK_VECxxx) 544
17.4 kernel_def.c と kernel_cfg.c 544
17.5 セクション 545
17.5.1 セクション名の付与ルール 545
17.5.2 セクション一覧 546
17.5.3 共有シンボル(CPUID#1からCPUID#2へのシンボルのexport) 550
17.5.4 CPUID#2の仮想リセットベクタテーブル 550
17.5.5 本サンプルのメモリマップ 550
17.6 カーネルライブラリ 554
17.7 各 CPU のビルド順序 555
17.7.1 基本形 555
17.7.2 ID名称をexportする場合 556
17.8 CPUID#1 側のビルド解説(cpuid1¥cpuid1.hws) 558
17.8.1 登録されているソース 558
17.8.2 コンパイラオプション 560
17.8.3 標準ライブラリ構築ツール 563
17.8.4 最適化リンカ 565
17.9 CPUID#2 側のビルド解説(cpuid2¥cpuid2.hws) 569
17.9.1 登録されているソース 569
17.9.2 コンパイラオプション 571

17.9.3　標準ライブラリ構築ツール....573
17.9.4　最適化リンカ....575
17.10 ターゲットへのダウンロード....578

18. スタックサイズの算出方法....579
18.1　スタックの種類....579
18.2　スタックサイズ計算の基本....580
18.2.1　関数ツリーによる消費サイズ....580
18.2.2　カーネルサービスコール....581
18.2.3　RPCライブラリの呼び出し....581
18.2.4　OAL、IPI、SH2A-DUALキャッシュサポートライブラリ、スピンロックライブラリ....581
18.2.5　拡張サービスコール....581
18.2.6　ノーマルCPU例外ハンドラ、ダイレクトCPU例外ハンドラ....581
18.3　Call Walker 使用時の注意事項....582
18.4　NMI 使用時の注意....583
18.5　スタックサイズの変化に関する注意....583
18.6　タスクのスタック....584
18.6.1　スタックサイズの算出....584
18.6.2　スタックサイズの指定箇所....585
18.6.3　デフォルトタスクスタック領域サイズの算出(memstk.all_memsize)....585
18.6.4　SVCサーバタスクのスタックサイズ(remote_svc.stack_size)....585
18.6.5　RPCサーバタスクとサーバスタブ....586
18.7　ノーマル割込みハンドラのスタック(system.stack_size)....587
18.7.1　各ハンドラのスタックサイズの算出....587
18.7.2　割込みスタック領域サイズの算出と指定箇所(system.stack_size)....587
18.8　ダイレクト割込みハンドラのスタック....588
18.8.1　スタックサイズの算出....588
18.8.2　スタックサイズの指定箇所....588
18.8.3　スタックの共有....588
18.9　タイマスタック(clock.stack_size)....589
18.10 カーネルスタック(system.kernel_stack_size)....590
18.11 HI7200/MP 提供機能による使用サイズ....591
18.11.1　カーネル....591
18.11.2　RPCライブラリ....591
18.11.3　OALのAPI関数....591
18.11.4　IPI....592
18.11.5　スピンロックライブラリのAPI関数....592
18.11.6　キャッシュサポートライブラリのAPI関数....592

19. types.h....593

20. FPU に関する注意....595
20.1　コンパイラのオプション....595
20.1.1　オプションの統一....595

RENESAS

20.1.2 cpuオプション 595
20.1.3 fpuオプションとfpscrオプション 595
20.2 タスク、タスク例外処理ルーチンで浮動小数点演算を行う場合 596
20.2.1 TA_COP1属性 596
20.2.2 FPSCRの初期化 596
20.3 各種ハンドラ等で浮動小数点演算を行う場合 597
20.3.1 概要 597
20.3.2 コーディング方法 598
20.4 拡張サービスコールルーチン浮動小数点演算を行う場合 599
20.4.1 タスクコンテキストから呼び出される場合 599
20.4.2 非タスクコンテキストから呼び出される場合 599
20.5 参考：コンパイラの扱い 600
改訂内容 601

RENESAS

図目次

図3.1 ソフトウェア構造 7
図3.2 ハードウェアリソースの分割(イメージ) 7
図3.3 メモリマップの分割(イメージ) 8
図3.4 RPCを用いた機能分割(イメージ) 8
図3.5 リモートサービスコールを用いた機能分割(イメージ) 9
図4.1 マルチタスク動作 11
図4.2 タスクの中断と再開 12
図4.3 タスクの切り替え 12
図4.4 サービスコール 13
図4.5 タスクの識別 15
図4.6 タスクの状態 16
図4.7 タスクの状態遷移図 17
図4.8 レディキュー(実行待ち状態) 18
図4.9 TA_TPRI属性の待ち行列 19
図4.10 TA_TFIFO属性の待ち行列 19
図4.11 共有スタック機能使用時のタスク状態遷移 21
図4.12 サービスコール内の割込み制御 28
図5.1 優先度の変更 32
図5.2 待ち行列のつなぎ換え 33
図5.3 起床要求の蓄積 34
図5.4 起床要求のキャンセル 34
図5.5 タスクの強制待ちと再開 35
図5.6 タスクの強制待ちと強制再開 35
図5.7 タスク付属イベントフラグの動作例 36
図5.8 タスク例外処理の動作例 37
図5.9 セマフォの動作例 39
図5.10 優先度逆転現象 41
図5.11 イベントフラグの動作例 42
図5.12 データキュー 44
図5.13 メールボックス 46
図5.14 ミューテックスの動作例 48
図5.15 メッセージバッファ 50
図5.16 固定長メモリプール 52
図5.17 空き領域の断片化 55
図5.18 可変長メモリプールの例 57
図5.19 タイムアウト 58
図5.20 タスクの遅延 59
図5.21 周期ハンドラの動作例 60
図5.22 アラームハンドラの動作例 62
図5.23 オーバーランハンドラの動作例 63
図5.24 時間の精度(tslp_tsk) 64
図5.25 時間の精度(sta_cyc) 65
図5.26 rot_rdqによるレディキューの操作 67
図5.27 プロファイル機能 73
図6.1 ID番号 81

RENESAS

図6.2　カーネルの構造 82
図6.3　サービスコールの説明形式 85
図6.4　メッセージの形式例 158
図6.5　優先度付きメッセージの形式例 158
図7.1　RPCの概念図 277
図7.2　RPCの構成と動作 278
図7.3　サーバパラメータ領域の例 307
図12.1　タスクの記述例 355
図12.2　無限ループするタスクの例 355
図12.3　タスク例外処理ルーチンの記述例 357
図12.4　拡張サービスコールルーチンの記述例 358
図12.5　ノーマル割込みハンドラの記述例 360
図12.6　ダイレクト割込みハンドラの記述例 362
図12.7　ノーマルCPU例外ハンドラの記述例 364
図12.8　ダイレクトCPU例外ハンドラの記述例 366
図12.9　周期ハンドラ、アラームハンドラのC言語記述例 367
図12.10　オーバーランハンドラのC言語記述例 367
図12.11　初期化ルーチンのC言語記述例 369
図13.1　ロードモジュール生成フロー 380
図14.1　コンフィギュレータ動作概要 439
図15.1　GUIコンフィギュレータとcfgファイルおよびcfg72mpの関係 449
図16.1　SH7265メモリマッピング 452
図16.2　スタートアップ概略フロー 454
図16.3　CPUID#1のスタートアップフロー 456
図16.4　CPUID#2のスタートアップフロー 457
図16.5　cpuid1¥reset¥reset.src 459
図16.6　cpuid1¥reset¥resetprg1.c 465
図16.7　cpuid2¥reset¥vreset.src 470
図16.8　cpuid2¥reset¥resetprg2.c 471
図16.9　cpuid1¥init¥init_task1.c 474
図16.10　cpuid2¥init¥init_task2.c 477
図16.11　スタートアップフェーズの同期 482
図16.12　SampleAdd() (クライアントスタブ) 485
図16.13　rpcsvr_SAMPLE_SampleAdd() (サーバスタブ) 486
図16.14　SampleStrlen()(クライアントスタブ) 487
図16.15　rpcsvr_SAMPLE_SampleStrlen() (サーバスタブ) 488
図16.16　SampleRefTaskState() (クライアントスタブ) 490
図16.17　rpcsvr_SAMPLE_SampleRefTaskState() (サーバスタブ) 491
図16.18　SampleInit(), SampleShutdwon()(サーバ側) 492
図16.19　SampleInit(), SampleShutdwon()(クライアント側) 493
図16.20　メッセージを送信するタスク(cpuid1¥remote_svc_sample¥sample_send.c) 494
図16.21　メッセージを受信するタスク(cpuid2¥remote_svc_sample¥sample_recv.c) 495
図16.22　lowsrc_config.h 499
図16.23　lowsrc.c 500
図16.24　initsct.c 506
図17.1　[ファイル拡張子]ダイアログボックス 534
図17.2　[ファイル拡張子の追加]ダイアログボックス 535

図17.3 [ビルドフェーズ]ダイアログボックス ... 536
図17.4 [新規ビルドフェーズ －1/4ステップ]ダイアログボックス ... 536
図17.5 [新規ビルドフェーズ －2/4ステップ]ダイアログボックス ... 537
図17.6 [新規ビルドフェーズ －3/4ステップ]ダイアログボックス ... 537
図17.7 [新規ビルドフェーズ －4/4ステップ]ダイアログボックス ... 538
図17.8 cfg72mp出力シンタックスの登録 ... 539
図17.9 [cfg72mp Options]ダイアログボックス ... 539
図17.10 [ビルドフェーズ]ダイアログボックス(ビルド順序) ... 541
図17.11 [ビルドフェーズ]ダイアログボックス(ファイルのビルド順序) ... 541
図17.12 プログラム、定数セクションの配置 ... 551
図17.13 変数セクションの配置（SDRAM） ... 552
図17.14 変数セクションの配置（内蔵RAM） ... 553
図17.15 各CPUのビルドフェーズの依存関係(基本形) ... 555
図17.16 CPUID#1がCPUID#2のID名称ヘッダファイルをインクルードする場合 ... 556
図17.17 CPUID#2がCPUID#1のID名称ヘッダファイルをインクルードする場合 ... 556
図17.18 両CPUが互いのID名称ヘッダファイルをインクルードする場合 ... 557
図17.19 prj_cpuid1プロジェクトに登録されているソース ... 558
図17.20 コンパイラ・インクルードディレクトリ(共通設定) ... 560
図17.21 コンパイラ・インクルードディレクトリ(remote_send.c) ... 561
図17.22 コンパイラ・インクルードディレクトリ(kernel_cfg.c, kernel_def.c) ... 561
図17.23 コンパイラ・マクロ定義 ... 562
図17.24 コンパイラ・出力ファイル形式 ... 562
図17.25 標準ライブラリ構築ツール・ライブラリ機能の選択 ... 563
図17.26 標準ライブラリ構築ツール・リエントラントライブラリ ... 563
図17.27 標準ライブラリ構築ツール・セクション名の設定 ... 564
図17.28 最適化リンカ・ライブラリの入力 ... 565
図17.29 最適化リンカ・CPUID#2の仮想リセットベクタテーブルのシンボル定義 ... 566
図17.30 最適化リンカ・セクション配置 ... 567
図17.31 最適化リンカ・ROMからRAMへのマップ ... 567
図17.32 最適化リンカ・シンボルアドレスファイルの出力 ... 568
図17.33 prj_cpuid2プロジェクトに登録されているソース ... 569
図17.34 コンパイラ・インクルードディレクトリ ... 571
図17.35 コンパイラ・インクルードディレクトリ(kernel_cfg.c, kernel_def.c) ... 571
図17.36 コンパイラ・マクロ定義 ... 572
図17.37 コンパイラ・出力ファイル形式 ... 572
図17.38 標準ライブラリ構築ツール・ライブラリ機能の選択 ... 573
図17.39 標準ライブラリ構築ツール・リエントラントライブラリ ... 573
図17.40 標準ライブラリ構築ツール・セクション名の設定 ... 574
図17.41 最適化リンカ・ライブラリの入力 ... 575
図17.42 最適化リンカ・セクション配置 ... 576
図17.43 最適化リンカ・ROMからRAMへのマップ ... 577
図18.1 関数ツリーによる消費サイズ ... 580
図18.2 関数テーブルを用いた呼び出しを行っている関数のCall Walker表示例 ... 582
図20.1 タスクにおけるFPSCR初期化例 ... 596
図20.2 ハンドラ等におけるFPSCR初期化・FPUレジスタの保証例 ... 598

表目次

表3.1　動作環境 9
表4.1　スタティックスタックと非スタティックスタックの相違 20
表4.2　タスクコンテキストと非タスクコンテキスト 22
表4.3　CPUロック状態で使用可能なサービスコール 23
表4.4　ノーマルCPU例外ハンドラで使用可能なサービスコール 30
表5.1　微小ブロックの扱い 56
表6.1　kernel.hからインクルードされるファイル 75
表6.2　基本データ型 76
表6.3　サービスコール呼び出し前後のレジスタ保証 77
表6.4　タスク管理サービスコール 87
表6.5　タスク管理の仕様 88
表6.6　タスク生成時に行われる処理 90
表6.7　タスク起動時に行われる処理 93
表6.8　タスク終了時に行われる処理 97
表6.9　タスク付属同期サービスコール 107
表6.10　タスク付属同期の仕様 108
表6.11　タスク例外処理サービスコール 120
表6.12　タスク例外の仕様 120
表6.13　同期･通信（セマフォ）サービスコール 128
表6.14　セマフォの仕様 128
表6.15　同期･通信（イベントフラグ）サービスコール 135
表6.16　イベントフラグの仕様 135
表6.17　同期･通信（データキュー）サービスコール 144
表6.18　データキューの仕様 144
表6.19　同期･通信（メールボックス）サービスコール 153
表6.20　メールボックスの仕様 153
表6.21　同期･通信（ミューテックス）サービスコール 163
表6.22　ミューテックスの仕様 163
表6.23　同期･通信（メッセージバッファ）サービスコール 170
表6.24　メッセージバッファの仕様 170
表6.25　メモリプール管理（固定長メモリプール）サービスコール 179
表6.26　固定長メモリプールの仕様 179
表6.27　メモリプール管理（可変長メモリプール）サービスコール 188
表6.28　可変長メモリプールの仕様 188
表6.29　システム時刻管理サービスコール 198
表6.30　システム時刻管理の仕様 198
表6.31　周期ハンドラサービスコール 206
表6.32　周期ハンドラの仕様 206
表6.33　アラームハンドラサービスコール 213
表6.34　アラームハンドラの仕様 213
表6.35　オーバーランハンドラサービスコール 220
表6.36　オーバーランハンドラの仕様 220
表6.37　システム状態管理サービスコール 225
表6.38　割込み管理サービスコール 243
表6.39　割込み管理の仕様 243

表6.40　サービスコール管理サービスコール........ 249
表6.41　サービスコール管理の仕様........ 249
表6.42　システム構成管理サービスコール........ 252
表6.43　システム構成管理の仕様........ 252
表6.44　プロファイル管理サービスコール........ 263
表7.1　RPCライブラリ概要........ 277
表7.2　API一覧........ 287
表8.1　OAL概要........ 309
表8.2　API関数一覧........ 311
表9.1　スピンロックライブラリ概要........ 319
表9.2　API関数一覧........ 325
表10.1　IPI概要........ 337
表10.2　ポートIDとプロセッサ間割込みの関係........ 337
表10.3　API関数一覧........ 342
表10.4　プロセッサ間割込みハンドラ........ 346
表11.1　SH2A-DUAL用キャッシュサポートライブラリ概要........ 347
表11.2　API関数一覧........ 349
表12.1　タスクのレジスタ使用規約........ 356
表12.2　タスク例外処理ルーチンのレジスタ使用規約........ 357
表12.3　レジスタバンク........ 359
表12.4　ノーマル割込みハンドラのレジスタ使用規約........ 361
表12.5　“tn=”指定と“resbank”指定........ 362
表12.6　ダイレクト割込みハンドラのレジスタ使用規約........ 363
表12.7　ノーマルCPU例外ハンドラのレジスタ使用規約........ 365
表12.8　ダイレクトCPU例外ハンドラのレジスタ使用規約........ 366
表12.9　タイムイベントハンドラのレジスタ使用規約........ 368
表12.10　初期化ルーチンのレジスタ使用規約........ 369
表12.11　tdr_ini_tmr()のレジスタ使用規約........ 372
表12.12　tdr_int_tmr()のレジスタ使用規約........ 373
表12.13　tdr_stp_tmr()のレジスタ使用規約........ 374
表12.14　tdr_rst_tmr()のレジスタ使用規約........ 375
表12.15　システムダウンルーチンに渡されるパラメータ........ 376
表12.16　初期登録失敗時のオブジェクト種別とオブジェクト番号........ 377
表14.1　数値表現例........ 382
表14.2　演算子........ 382
表14.3　外部参照名の指定例........ 383
表14.4　割込みベクタのタイプ........ 390
表14.5　regbankの定義方法........ 391
表14.6　タスクの種類........ 404
表14.7　データキュー領域の指定方法........ 415
表14.8　メッセージバッファ領域の指定方法........ 420
表14.9　固定長メモリプール領域の指定方法........ 423
表14.10　可変長メモリプール領域の指定方法........ 426
表14.11　オブジェクトの使用に必要なサービスコール........ 437
表14.12　補正されるサービスコール........ 438
表14.13　エラーレベル........ 442
表16.1　R0K572650D00BRの仕様概要........ 451

表16.2　スタートアップ関連のソースファイル.. 455
表16.3　リセットベクタテーブル.. 458
表16.4　内蔵RAMアクセス権の初期化.. 468
表16.5　仮想リセットベクタテーブル.. 469
表16.6　サーバ関数.. 483
表16.7　RPC使用例のソースファイル.. 484
表16.8　リモートサービスコール使用例のソースファイル.. 494
表16.9　タイマドライバのソースファイル.. 495
表16.10　標準ライブラリ関連のソースファイル.. 496
表16.11　低水準インタフェースルーチン.. 497
表16.12　標準ライブラリコンフィギュレーションファイルの設定項目.. 499
表16.13　ダミープログラムのソースファイル.. 508
表16.14　I/O定義ファイル.. 510
表16.15　タスク(CPUID#1).. 511
表16.16　タスク(CPUID#2).. 511
表16.17　その他のオブジェクト(CPUID#1).. 512
表16.18　その他のオブジェクト(CPUID#2).. 513
表16.19　IPIポート.. 530
表17.1　cfg72mp出力ファイル.. 540
表17.2　SH7265の内蔵RAMアドレス空間.. 545
表17.3　カーネルライブラリ、kernel_def.c、kernel_cfg.cのセクション.. 546
表17.4　RPCライブラリ、rpc_table.cのセクション.. 546
表17.5　スピンロックライブラリのセクション.. 546
表17.6　SH2A-DUALキャッシュサポートライブラリのセクション.. 547
表17.7　IPIのセクション.. 547
表17.8　OALのセクション.. 547
表17.9　サンプルのセクション(CPUID#1).. 548
表17.10　サンプルのセクション(CPUID#2).. 549
表17.11　カーネルライブラリの優先順位.. 554
表18.1　タスクのコンテキストサイズ.. 584
表19.1　types.hで定義されるデータ型.. 593
表20.1　コンパイラによるFPSCR.PRビットの扱い.. 600
表20.2　コンパイラによるFPSCR.RMビットの扱い.. 600

1. 本マニュアルの構成

本マニュアルは、以下の章から構成されています。

◆ 2. インストール
  インストール方法について解説しています。

◆ 3. 概要
  本製品の概要について解説しています。

◆ 4. カーネル入門
  本製品を使用する上で必要となる考え方や、本製品の基幹であるカーネルに関する基本的な項目を解説しています。

◆ 5. カーネルの機能
  カーネルの機能について解説しています。

◆ 6. カーネルサービスコール
  カーネルのサービスコール仕様を記載しています。

◆ 7.RPC ライブラリ
  RPCライブラリの仕様を記載しています。

◆ 8. OAL
  OALの仕様を記載しています。

◆ 9. スピンロックライブラリ
  スピンロックライブラリの仕様を記載しています。

◆ 10. IPI
  IPIの仕様を記載しています

◆ 11. SH2A-DUAL 用キャッシュサポートライブラリ
  SH2A-DUAL用キャッシュサポートライブラリの仕様を記載しています。

◆ 12. アプリケーションプログラムの記述方法
  タスクやハンドラの記述方法を説明しています。

◆ 13. ロードモジュール作成手順概要
  ロードモジュールの作成手順の概要を説明しています。

◆ 14. コンフィギュレータ(cfg72mp)
  cfg72mpの仕様を記載しています。

- 15. GUI コンフィギュレータ
  GUIコンフィギュレータについて紹介しています。GUIコンフィギュレータの使用方法は、オンラインヘルプを参照してください。

- 16. サンプルプログラム
  提供しているサンプルプログラムについて解説しています。

- 17. ビルド
  コンパイルしてロードモジュールを生成する方法を解説しています。

- 18. スタックサイズの算出方法
  タスクやハンドラなどに必要なスタックサイズの算出方法を説明しています。

- 19. types.h
  基本データ型を定義しているtypes.hについて解説しています。

- 20. FPU に関する注意
  FPUに関する注意事項を記載しています。FPUを搭載したCPUを使用する場合は、FPU機能を使用しない場合も、本章を一読してください。

2. インストール

2.1 インストール方法

インストール方法は、製品添付のリリースノートを参照してください。

2.2 ディレクトリ構成

HI7200/MP は、「システムディレクトリ」と「サンプルディレクトリ」の 2 箇所にインストールされます。それぞれは、異なるディレクトリにインストールすることができます。

2.2.1 システムディレクトリ(<RTOS_INST>)

システムディレクトリには、ヘッダファイルやコンフィギュレータなどがインストールされます。本マニュアルでは、システムディレクトリを<RTOS_INST>と表記します。

システムディレクトリの場所は、インストール時にユーザが指定したディレクトリ下の”x.yy.zzww”ディレクトリです。なお、x, yy, zz は製品バージョン、ww は内部識別番号で”00”～”99”の数値です。以下に例を示します。

- 製品バージョンが”V.1.01 Release 02”の場合：”1.01.02ww”
- 製品バージョンが”V.2.11 Release 13”の場合：”2.11.13ww”

以下に、システムディレクトリ下の構成を示します。

cfg72mp¥	cfg72mp(コマンドラインコンフィギュレータ)
gui_config¥	GUI コンフィギュレータ
manuals¥	マニュアル
os¥	
include¥	共通ヘッダ (types.h, itron.h など)
kernel¥	カーネルのソースコード (ソースコードライセンス付の HI7200/MP のみ)
lib¥	ライブラリ
rpc¥	RPC ライブラリのソースコード (ソースコードライセンス付の HI7200/MP のみ)
sh2adual_cache¥	SH2A-DUAL 用キャッシュサポートライブラリのソースコード
spinlock¥	スピンロックライブラリのソースコード
system¥	システム定義ファイル

2.2.2　サンプルディレクトリ(<SAMPLE_INST>)

　サンプルディレクトリは、文字通りサンプルプログラムがインストールされるディレクトリです。本マニュアルでは、サンプルディレクトリを<SAMPLE_INST>と表記します。
　サンプルディレクトリの場所は、インストール時にユーザが指定します。
　以下に、システムディレクトリ下の構成を示します。

R0K572650D000BR¥　SH7265 **を搭載した評価ボード**”R0K572650D000BR”**用サンプルプログラム**

　R0K572650D000BR ディレクトリのサンプルプログラムについては、「16. サンプルプログラム」で詳しく解説しています。

3. 概要

3.1 概要

本製品は、SH2A-DUAL マイコン用に開発されたリアルタイム OS で、各 CPU コア毎に機能分散したシステムに向けて設計されています。

本 OS 採用時は、各 CPU 毎に独立に OS を搭載したシステムが動作し、必要に応じて CPU 間で同期・通信を行うことができます。

3.2 特長

3.2.1 カーネル

本製品は、広く普及しているリアルタイム OS 仕様であるμITRON4.0 仕様に準拠したカーネルを装備しています。このため、市販されているμITRON 関連書籍やセミナ等で得た知識をほとんどそのまま役立てることができます。また、μITRON 仕様を前提とした各種ソフトウェア部品を容易に組み込むことができます。

また、従来のμITRON4.0 仕様の API のまま、他方の CPU コアのカーネルに対してサービスコールを呼び出すことができます。このため、従来のシングル CPU 用のアプリケーションプログラムを容易に２つの CPU に分散することができます。言い換えると、従来のシングル CPU でのμITRON 仕様 OS を用いたプログラミングモデルを、マルチコア環境に拡張することができます。

3.2.2 RPC(リモートプロシージャコール)ライブラリ

RPC を利用することで、他方の CPU にある関数を、通常の関数コールと同じ形式で呼び出すことができるようになります。これにより、関数単位に各 CPU への機能分散を容易に行うことができます。

3.2.3 OAL

OAL は、RPC の OS 依存部を抽出した機能モジュールです。OAL を書き換えれば、容易に他の OS に RPC をポーティングすることができます。

3.2.4 スピンロックライブラリ

CPU 間で共有リソースを扱う場合は、その排他制御が必要になります。CPU 間の排他制御のためのプリミティブとして、スピンロックライブラリを用意しています。

スピンロックライブラリは、カーネルや RPC でも利用しています。

3.2.5 IPI 機能

IPI 機能は、CPU 間での通信を担うプリミティブで、CPU 間割込みを使った受信ポートを提供します。

IPI 機能は、カーネルや RPC でも利用しています。

3.2.6 キャッシュサポートライブラリ

アプリケーションは、各 CPU が持つローカルキャッシュと実メモリのコヒーレンシの維持などの目的で、キャッシュサポートライブラリを利用することができます。

3.2.7 サンプルプログラム

OS 機能の理解を目的とした各種サンプルプログラム、およびそのビルド環境としての High-performance Embedded Workshop のワークスペースを提供しています。

3.2.8 コンフィギュレータ

カーネルのコンフィギュレータ(cfg72mp)を装備しているので、システムに最適なカーネルを容易に構築することができます。ユーザは、定められた書式でカーネルコンフィギュレーションファイル(cfg ファイル)を作成します。

さらに、初心者向けには cfg ファイルを生成するためのツールとして、GUI 画面で操作可能な GUI コンフィギュレータも用意しています。

3.2.9 デバッギングエクステンション(オプション製品)

High-performance Embedded Workshop V.4 以降にマルチタスクデバッグ機能を追加するデバッギングエクステンションを用意しています。デバッギングエクステンションでは、以下のような機能をサポートしています。

- タスクなどのオブジェクトの状態参照
- タスクの起動やイベントフラグのセットといった、オブジェクトに対する操作
- サービスコール履歴の表示

なお､デバッギングエクステンションは､当社ホームページより無償でダウンロードできます。（ただし、本マニュアル作成時点では、本 OS 対応のデバッギングエクステンションはまだ開発中です。）

3.3 マルチコア

本 OS では、両方の CPU で別々に OS が動作します。それぞれの CPU にどういった仕事をやらせるのか(機能分散)をアプリケーション設計者が静的に決定し、それぞれの OS 上に機能を実装します。

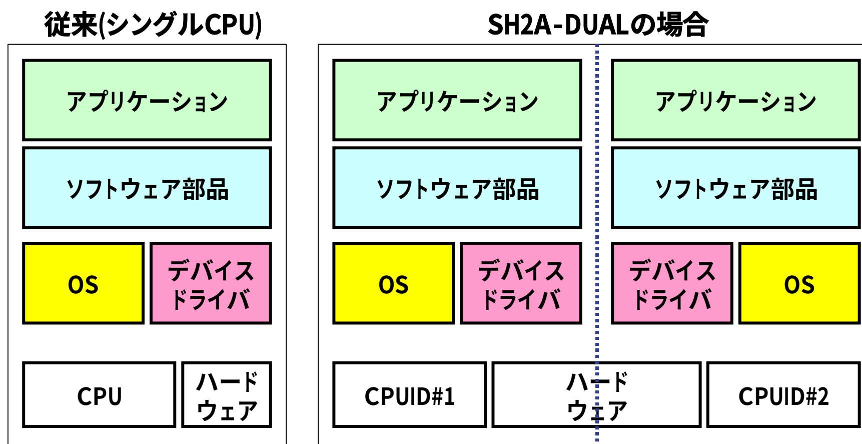
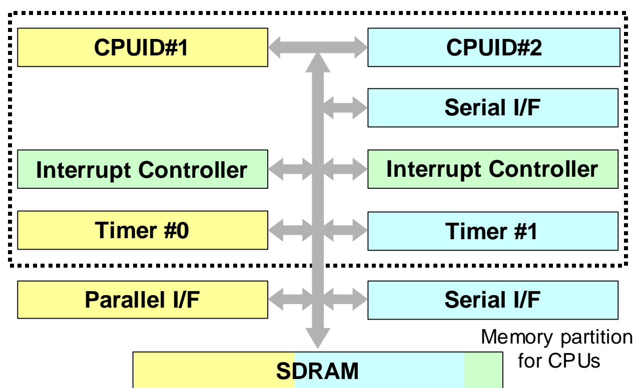


図3.1　ソフトウェア構造

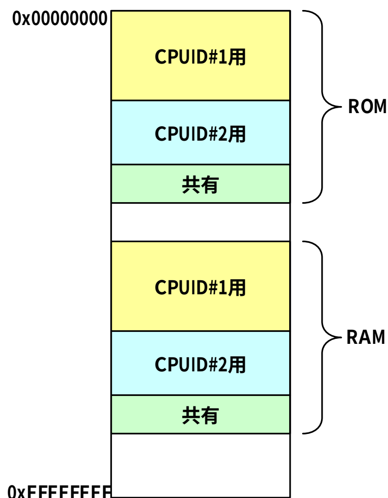
また、各種ハードウェアも、その機能分割に従ってどちらの CPU で制御するのかを静的に決定します。



【凡例】黄色：CPUID#1 が制御
水色：CPUID#2 が制御
緑：共有

図3.2　ハードウェアリソースの分割(イメージ)

メモリ空間は、SH2A-DUAL では基本的には両 CPU 共通に一面ですが、各 CPU が使用する範囲を静的に区切って使用します。もちろん、CPU 間通信のために両 CPU で共有するメモリを使用することもできます。



【凡例】黄色：CPUID#1 が制御
水色：CPUID#2 が制御
緑：共有

図3.3　メモリマップの分割(イメージ)

各 CPU への機能分割の基本的なやり方には、RPC ライブラリを使って関数単位に分割する方法(図 3.4)と、リモートサービスコールを用いてタスク単位に分割する方法(図 3.5)があります。
RPC ライブラリを用いると、他 CPU 上にある関数を呼び出すことができます。
リモートサービスコールを用いると、他 CPU のタスクやセマフォなどのカーネルオブジェクトに対して、従来と同じサービスコールを用いてアクセスすることができます。

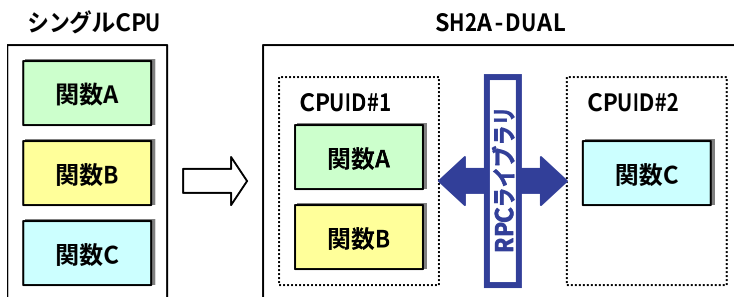


図3.4　RPC を用いた機能分割(イメージ)

RENESAS

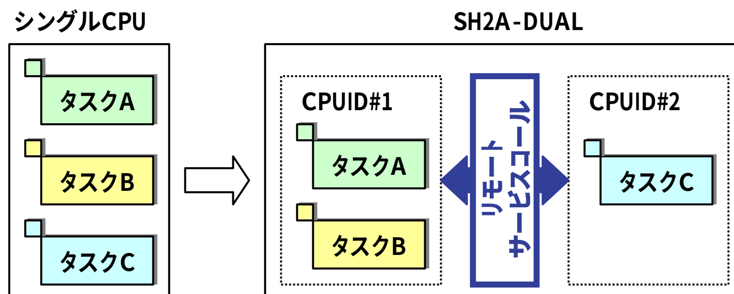


図3.5　リモートサービスコールを用いた機能分割(イメージ)

3.4　動作環境

表 3.1に、動作環境を示します。

表3.1　動作環境

項目	動作環境
対象 CPU コア	SH2A-DUAL
ホストマシン	下記 OS が稼動している IBM-PC/AT 互換機 Windows® 2000, Windows® XP
コンパイラ	ルネサス製「SuperH ファミリ用 C/C++コンパイラパッケージ」 V.9.00 Release 04A 以降
High-performance Embedded Workshop *	V.4.02.00 以降

【注】 提供する High-performance Embedded Workshop ワークスペースを使用する場合

4. カーネル入門

4.1 カーネルの動作原理

本製品では、SH2A-DUAL の各 CPU 上で、独立に別々のカーネルが動作します。

カーネルとは、リアルタイム OS の中核となるプログラムのことです。カーネルは、1 つの CPU を、あたかも複数の CPU が動作しているように見せることのできるソフトウェアです。では、1 つの CPU をどのように複数あるように見せかけているのでしょうか？それは、カーネルは図 4.1に示すようにそれぞれのタスクを必要に応じて切り替えながら動作させるからです。

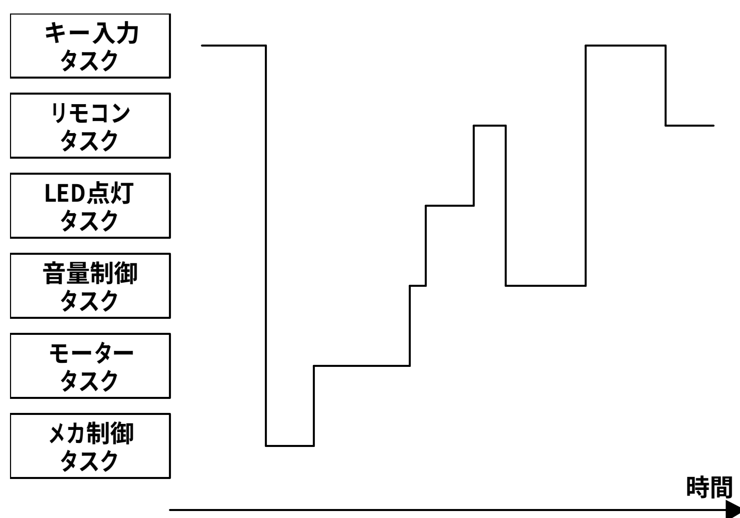


図4.1　マルチタスク動作

タスクを切り替えることをディスパッチと呼びます。ディスパッチが発生する要因として、以下のものがあります。

- タスクが自分自身で切り替えを要求する
- 割込みなど、タスクから見て外部の要因で切り替わる

言い換えると、一定時間毎にタスクが切り替わる(タイムシェアリング)わけではありません。このようなスケジューリングを一般に「イベントドリブン」または「イベント駆動型」と呼びます。

ディスパッチ発生後、再度そのタスクを実行するときには、中断していたところから実行を再開します(図 4.2参照)。

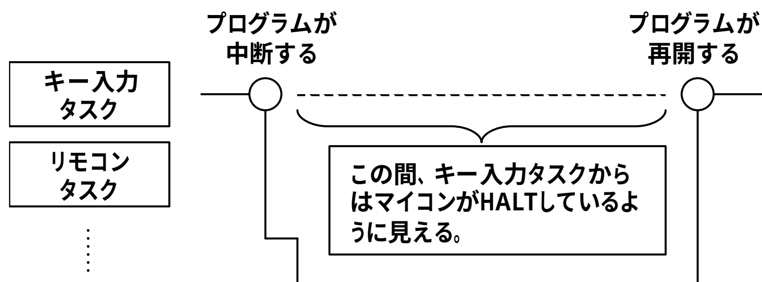


図4.2　タスクの中断と再開

図 4.2において、キー入力タスクは他のタスクに実行制御が移っている間、プログラマから見ればプログラムが中断し、そのマイコンが HALT しているように見えます。カーネルは、中断した時点の CPU レジスタ内容を復帰することにより、タスクを中断した時点の状態から再開させます。すなわち、ディスパッチとは、現在実行中のタスクの CPU レジスタの内容を、そのタスクを管理するメモリ領域に退避し、切り替えるタスクの CPU レジスタ内容を復帰することです(図 4.3参照)。

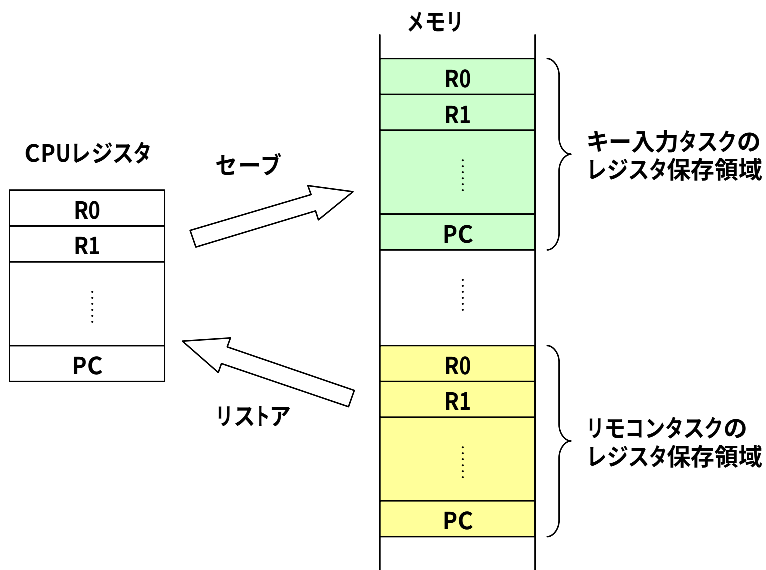


図4.3　タスクの切り替え

また、タスクが実行するには、CPU レジスタだけでなく、スタック領域も必要です。スタック領域も各タスク毎に別々の領域を使用します。

RENESAS

4.2 サービスコール

プログラムは、プログラム中でどのようにカーネルの機能を使用するのでしょうか？これには、カーネルの機能をプログラムから何らかの形で呼び出すことが必要です。このカーネルの機能を呼び出すことを、サービスコールといいます。すなわちサービスコールにより、タスクの起動などの処理を行うことができます(図 4.4参照)。

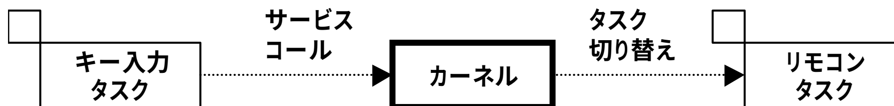


図4.4 サービスコール

サービスコールは、以下のように C 言語の関数呼び出しで実現します。

```
act_tsk(ID_MAINTASK);
```

また、サービスコールには、自 CPU のカーネルにだけ要求できるものと、他 CPU のカーネルにも要求できるものがあります。自 CPU のカーネルに要求するサービスコールを「ローカルサービスコール」、他 CPU のカーネルに要求するサービスコールを、「リモートサービスコール」と呼びます。

4.3 CPUID

CPUID は、各 CPU コアを識別するための番号です。

CPUID は、以下のような場面で使用します。

- サービスコールにおける操作対象オブジェクトが属する CPU の指定
- IPI での送信対象 CPU の指定

HI7200/MP では、SH2A-DUAL マイコン仕様での CPU#0 の CPUID は 1、CPU#1 の CPUID が 2 と決められています。SH2A-DUAL マイコン仕様の番号は 0 から始まるのに対し、CPUID は 1 から始まることに注意してください。

4.4 オブジェクト

4.4.1 概要

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

タスクを起動したり、イベントフラグをセットする、といったオブジェクトを操作するサービスコールでは、その操作対象のオブジェクト ID をパラメータに指定します。一部のサービスコールでは、指定するオブジェクト ID によって、他 CPU のカーネルのオブジェクトを操作することもできます。

4.4.2 ID 番号

ID 番号は符号付 16 ビット整数型で表現します。ID 番号の各ビットには、以下の意味があります。

- bit15(MSB)：ローカルオブジェクト ID の符号ビット
- bit14～12：CPUID
- bit11～0：ローカルオブジェクト ID

(1) CPUID

リモートサービスコールでは、操作対象のオブジェクト ID の bit14～12 に、対象の CPUID を指定します。CPUID に VCPU_SELF を指定すると、サービスコール呼び出し元と同じ CPU の指定ができます。VCPU_SELF は kernel.h で 0 に定義されているμITRON4.0 仕様外のマクロです。

(2) ローカルオブジェクト ID

オブジェクトには、正のローカルオブジェクト ID が付与されます。各オブジェクトのローカルオブジェクト ID の最大値は、cfg ファイルの maxdefine で定義します。

負のローカルオブジェクト ID を持つオブジェクトは存在しませんが、一部のサービスコールでは特殊な指定を行うために負のローカルオブジェクト ID 値を指定できるものがあります。

4.4.3 ID 名称によるオブジェクトの指定

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

```
act_tsk(1);  /* 自 CPU 側のローカルオブジェクト ID が 1 番のタスクを起動 */
```

とプログラム中に記述すると、プログラマは絶えずローカルオブジェクト ID 番号が 1 番のタスクは何かを知っている必要があります。また、他人がこのプログラムを見たときに ID 番号の 1 番のタスクが何かが一目では分かりません。

そこで本製品では、タスクの識別をそのタスクの名前(ID 名称)で指定し、その ID 名称からタスクの ID 番号への変換を、"コンフィギュレータ cfg72mp"が自動的に行います。具体的には、コンフィギュレータは、各タスクの ID 名称と ID 番号が対応づけられるように、以下のように定義されたヘッダファイル(kernel_id.h など)を出力します。

```
#define ID_MAINTASK MAKE_ID(1, 1)
```
[1]

図 4.5は、タスクを識別する様子を示したものです。

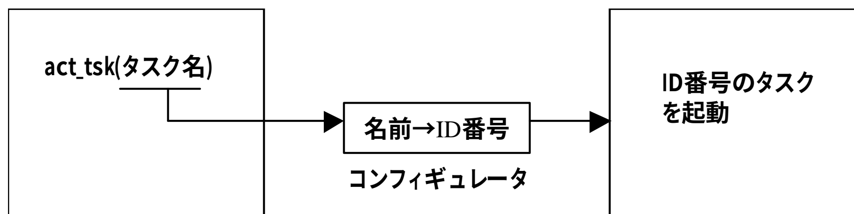


図4.5 タスクの識別

この定義を用いると、先の例は以下のように記述できます。

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

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

ここでは、タスクを例に説明しましたが、他の ID 番号で識別されるオブジェクトにも同様に ID 名称を付与することができます。

1 MAKE_ID()は、CPUIDとローカルオブジェクトIDからIDを生成するマクロです。詳細は、「6.31.4kernel.hで定義されている関数マクロ.」を参照してください。

4.5 タスク

4.5.1 タスクの状態

カーネルでは、タスクを実行するべきか否かをタスクの状態を管理することにより制御しています。例えば、図 4.6にキー入力タスクの実行制御と状態の関係を示します。キー入力が発生した場合は、そのタスクを実行しなければなりません。すなわち、キー入力タスクが実行状態となります。またキー入力を待っているときはタスクを実行する必要はありません。すなわち、キー入力タスクは待ち状態になっています。

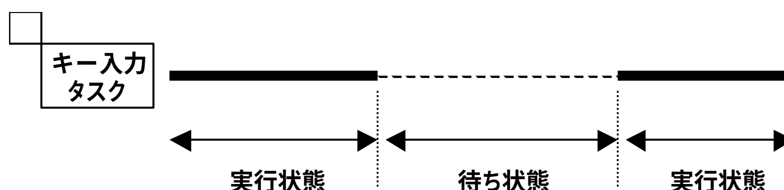


図4.6 タスクの状態

本カーネルでは、実行状態、待ち状態を含め、以下の 7 つの状態を管理しています。タスクは、この 7 つの状態を遷移していきます。図 4.7に、タスクの状態遷移図を示します。

(1) 未登録状態(NON-EXISTENT)

カーネルに登録されていない仮想的な状態です。

(2) 休止状態(DORMANT)

カーネルに登録された後、まだ起動されていない状態、または終了後の状態です。

(3) 実行可能状態(READY)

実行するための準備がすべて整った状態ですが、他の高い優先度のタスクが実行中のため実行はできない状態です。

(4) 実行状態(RUNNING)

現在、CPU 上で実行している状態です。
カーネルは実行可能状態のタスクの中で最も高い優先度のタスクを実行状態にします。

(5) 待ち状態(WAITING)

tslp_tsk サービスコールなどを呼び出した場合、条件が満たされない場合は待ち状態になります。待ち状態は、wup_tsk サービスコールなど、待ち状態になった要因に対応するサービスコールが呼び出されると解除され、実行可能状態に遷移します。

(6) 強制待ち状態(SUSPENDED)

タスクの実行が(sus_tsk サービスコール)により強制的に中断させられた状態です。

(7) 二重待ち状態(WAITING-SUSPENDED)

待ち状態と強制待ち状態が重なった状態です。

RENESAS

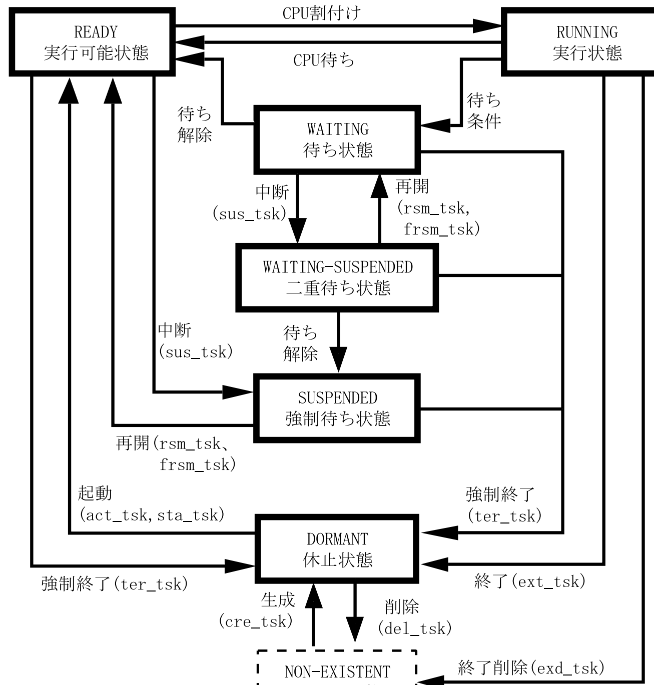


図4.7　タスクの状態遷移図

4.5.2 タスクのスケジューリング(優先度とレディキュー)

各タスクには、処理の優先順位を意味する「タスク優先度」が付与されます。タスク優先度は、値が小さいほど高い優先順位となり、1 が最高の優先順位です。利用可能な優先度の範囲は、1 から cfg ファイルに指定する system.priority となります。

カーネルは、実行可能状態にあるタスクの中で最も高い優先度のタスクを実行状態にします。

複数のタスクに同じ優先度を与えることもできます。

カーネルは、実行可能状態にあるタスクの中で最も高い優先度のタスクが複数存在する場合には、最も先に実行可能状態になったタスクを実行状態にします。カーネルはこれを実現するために、レディキューと呼ぶ実行可能状態のタスクの実行待ち行列を持っています。

図 4.8にレディキューの構造を示します。レディキューは優先度ごとに管理され、カーネルはタスクが接続されている最も優先度の高い待ち行列の先頭タスクを実行状態にします。

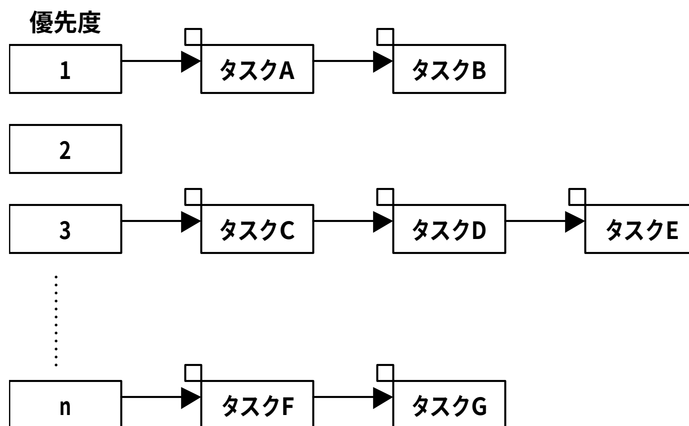


図4.8 レディキュー(実行待ち状態)

RENESAS

4.5.3 タスクの待ち行列

タスクは、セマフォやイベントフラグといったオブジェクトに対して、条件が満たされるまで待つ(待ち状態に遷移)ように要求(サービスコール)することができます。

オブジェクトによっては、複数のタスクが待つ状況もあります。この場合、どういう順序で待っているタスクを管理するかを、オブジェクトを生成するときに属性によって指定することができます。具体的には、FIFO 順で管理する(TA_TFIFO 属性)か、タスクの優先度順で管理する(TA_TPRI 属性)を選択することができます。

オブジェクトで待っているタスクの待ち解除は、この待ち行列の順序で行われます。

図 4.9および図 4.10に、あるオブジェクトに対して、タスク D(優先度 9)、タスク C(優先度 6)、タスク A(優先度 1)、タスク B(優先度 5)、という時間順で待ち行列に繋がれたときの様子を示します。

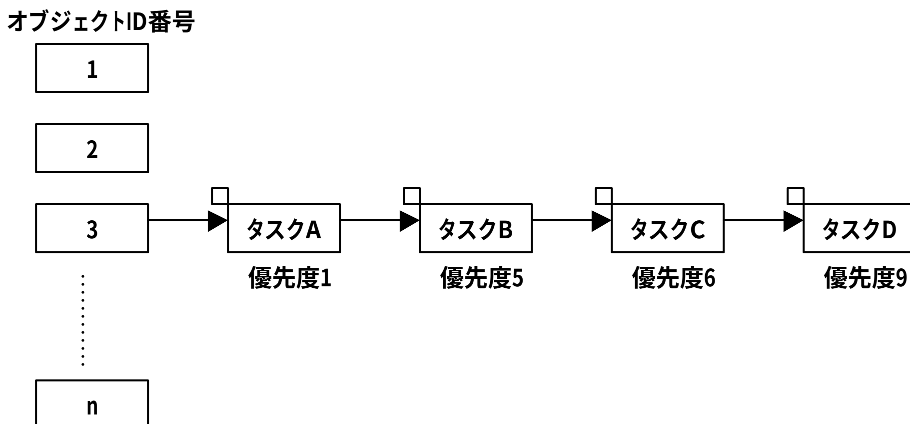


図4.9　TA_TPRI 属性の待ち行列

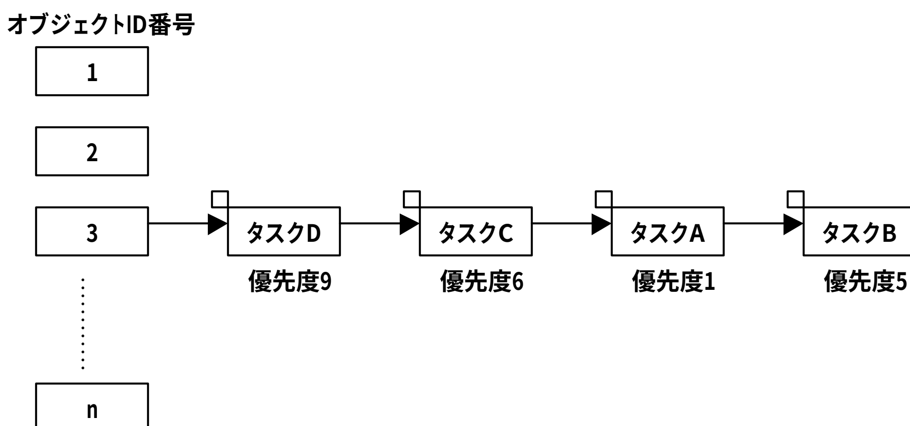


図4.10　TA_TFIFO 属性の待ち行列

4.5.4 タスクのスタック

各タスクには、それぞれ個別のスタック領域が必要です。

本カーネルでは、タスクが使用するスタックは、大きく「スタティックスタック」と「非スタティックスタック」に分類され、ローカルタスク ID が maxdefine.max_statictask 以下のタスクはスタティックスタック、それ以外のタスクは非スタティックスタックを使用することになっています。

スタティックスタックは、複数のタスクで共有することも可能です(共有スタック機能)。

(1) スタティックスタック

スタティックスタックは、cfg ファイルの static_stack[]により、複数定義することができます。static_stack[]では、スタックサイズ、スタックに付与するセクション名に加え、そのスタックを使用するローカルタスク ID を指定します。ローカルタスク ID を複数指定すると、それらのタスクでそのスタックを共有する意味になります。

(2) 非スタティックスタック

非スタティックスタックには、以下の種類があります。

(a) デフォルトタスクスタック用領域を使用する

cre_tsk, acre_tskサービスコールや、cfgファイルのtask[]などでタスクを生成する際に、スタックサイズだけを指定します。この場合、カーネルがデフォルトタスクスタック用領域からそのサイズのスタックを割り付けます。

デフォルトタスクスタック用領域はひとつだけ存在し、カーネル内部で可変長メモリプールのように管理されます。デフォルトタスクスタック用領域のセクション名は、”BC_hitskstk”です。

(b) アプリケーションで確保したスタック領域を使用する

アプリケーションでスタック領域を確保し、cre_tsk, acre_tskサービスコールや、cfgファイルのtask[]などでタスク生成する際に、そのアドレスとサイズを指定します。

(c) cfgファイルで、スタック領域を生成する

cfgファイルのtask[]で、スタックサイズとスタック領域に付与するセクション名を指定します。

表 4.1に、スタティックスタックと非スタティックスタックの相違を示します。

表4.1 スタティックスタックと非スタティックスタックの相違

<table>
<tr><th colspan="3" rowspan="2">項目</th><th colspan="2">使用するスタック</th></tr>
<tr><th>スタティックスタック</th><th>非スタティックスタック</th></tr>
<tr><td colspan="3">ローカルタスク ID 番号</td><td>1〜maxdefine.max_statictask</td><td>maxdefine.max_statictask+1 以上</td></tr>
<tr><td rowspan="4">タスク生成方法</td><td colspan="2">cfg ファイルによる生成</td><td>task[]</td><td>task[]</td></tr>
<tr><td rowspan="2"></td><td>ID 指定</td><td>省略不可(自動割当不可)</td><td>省略可能(自動割当可能)</td></tr>
<tr><td>その他</td><td>static_stack[]によるスタティックスタック領域の定義が必要</td><td>ー</td></tr>
<tr><td colspan="2">サービスコールによる生成</td><td>vscr_tsk, ivscr_tsk</td><td>cre_tsk, icre_tsk, acre_tsk, iacre_tsk</td></tr>
<tr><td colspan="3">タスク間のスタック共有</td><td>可能</td><td>不可</td></tr>
</table>

RENESAS

4.5.5 共有スタック機能

共有スタック機能は、複数のタスクでスタティックスタックを共有する機能です。共有スタック機能は、μITRON4.0 仕様の範囲外です。

スタティックスタックを複数のタスクで共有するには、cfg ファイルの static_stack[].tskid に、共有するタスク群のローカルタスク ID を羅列します。

スタティックスタックを共有するタスク群では、同時にはひとつのタスクのみがスタックを占有して実行する権利が与えられます。同一スタックを使用するように定義されているタスクが複数起動された場合は、先に起動されたタスクがスタックを占有し、他のタスクは共有スタック待ち状態となります。共有スタック待ちのタスクは、優先度に関係なく FIFO(First-In First-Out)で管理され、起動要求が行われた順番につながれます。

共有スタックは占有しているタスクが休止状態となったときに解放されます。この時、共有スタック待ちのタスクが存在すれば、待ちの先頭タスクがスタックを占有して実行可能状態となります。

図 4.11に共有スタック機能を使用するタスクの状態遷移を示します。

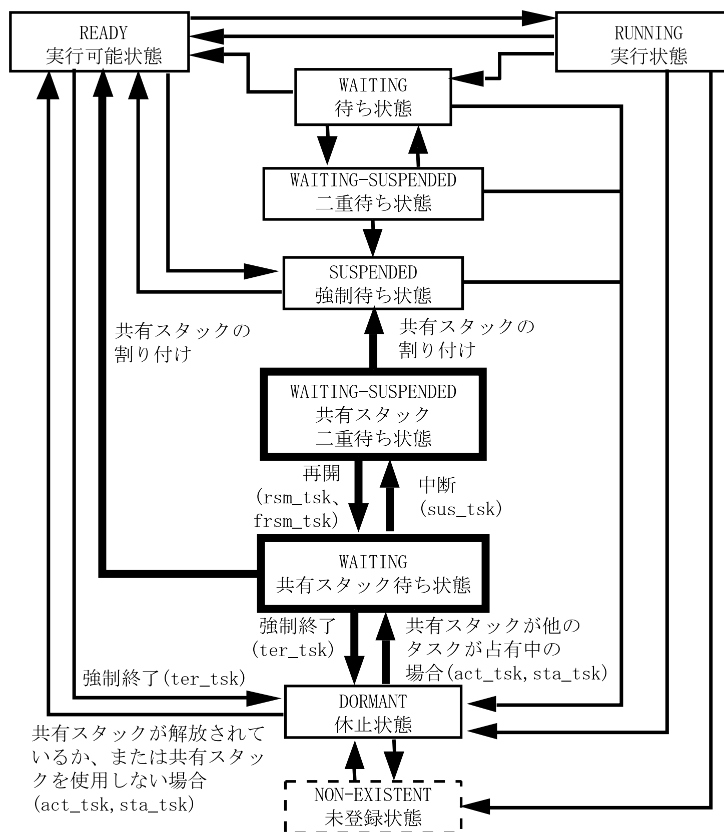


図4.11 共有スタック機能使用時のタスク状態遷移

4.6 システムの状態

システムの状態は、以下の３つの直行する状態によって区別されます。
(1)タスクコンテキスト/非タスクコンテキスト
(2)ディスパッチ禁止/許可状態
(3)CPU ロック/ロック解除状態
システムの挙動や呼び出し可能なサービスコールは、これらの状態によって決まります。

4.6.1 タスクコンテキストと非タスクコンテキスト

システムは、「タスクコンテキスト」か「非タスクコンテキスト」のいずれかのコンテキスト状態で実行します。タスクコンテキストと非タスクコンテキストの違いを、表 4.2に示します。

表4.2 タスクコンテキストと非タスクコンテキスト

	タスクコンテキスト	非タスクコンテキスト
呼び出し可能なサービスコール	タスクコンテキストから呼び出しできるもの	非タスクコンテキストから呼び出しできるもの
タスクスケジューリング	4.6.2, 4.6.3項参照	発生しない

非タスクコンテキストで実行される処理には、以下があります。
- 割込みハンドラ
- タイムイベントハンドラ（周期ハンドラ、アラームハンドラ、オーバーランハンドラ）
- chg_ims サービスコールで割込みマスクを 0 以外に変更して実行する部分

また、これらの状態から起動される拡張サービスコールルーチンも、非タスクコンテキストで実行されます。
なお、CPU 例外ハンドラは例外発生前と同じコンテキストで実行されます。

4.6.2 ディスパッチ禁止/許可状態

システムは、ディスパッチ禁止状態か許可状態かのいずれかの状態をとります。ディスパッチ禁止状態では、タスクスケジューリングは行われません。また、待ち状態に遷移するサービスコールは呼び出しできません。
ディスパッチ禁止状態へは dis_dsp、許可状態へは ena_dsp サービスコールによって遷移します。また、sns_dsp サービスコールでディスパッチ禁止状態かどうかを知ることができます。

RENESAS

4.6.3　CPU ロック/ロック解除状態

システムは、CPU ロック状態か CPU ロック解除状態かのいずれかの状態をとります。CPU ロック状態では、割込みの受け付けが禁止され、タスクスケジューリングも行われません。ただし、カーネル割込みマスクレベル(cfg ファイルの system.system_IPL)より高いレベルの割込みは受け付けられます。また、待ち状態に遷移するサービスコールは呼び出しできません。

CPU ロック状態へは loc_cpu, iloc_cpu、解除状態へは unl_cpu, iunl_cpu サービスコールによって遷移します。また、sns_loc サービスコールで CPU ロック状態かどうかを知ることができます。

また、CPU ロック状態から呼び出し可能なサービスコールは、表 4.3のように制限されます。

表4.3　CPU ロック状態で使用可能なサービスコール

loc_cpu, iloc_cpu	unl_cpu, iunl_cpu	sns_ctx	sns_loc	sns_dsp	sns_dpn
vsta_knl, ivsta_knl	vsys_dwn, ivsys_dwn	sns_tex	vsns_tmr	ext_tsk *	exd_tsk *

【注】 CPU ロック状態は解除されます。

4.6.4　ディスパッチ保留状態

タスクディスパッチが保留される状態のことを、ディスパッチ保留状態と呼びます。具体的には、以下のいずれかに該当する場合がディスパッチ保留状態です。

(1)　非タスクコンテキスト実行中
(2)　ディスパッチ禁止状態
(3)　CPUロック状態
(4)　CPUの割込みマスク(SRレジスタのIMASKビット)が0でない

sns_dpn サービスコールで、ディスパッチ保留状態かどうかを知ることができます。

4.7 処理の単位と優先順位

アプリケーションは、以下の処理単位によって実行制御されます。

(1) タスク
マルチタスクの制御対象となる単位です。
(2) タスク例外処理ルーチン
タスクに対してタスク例外処理が要求（ras_texサービスコール）されると、タスク例外処理ルーチンが実行されます。
(3) 割込みハンドラ
割込みが発生したときに実行されます。
(4) CPU例外ハンドラ
CPU例外が発生したときに実行されます。
(5) タイムイベントハンドラ（周期ハンドラ、アラームハンドラ、オーバーランハンドラ）
指定した周期や時刻になったときに実行されます。
(6) 拡張サービスコール
拡張サービスコールは、リンクしないモジュールを呼び出すための機能です。拡張サービスコールを発行すると、対応する拡張サービスコールルーチンが呼び出されます。

また、各処理単位は以下の優先順位で処理されます。
(1) 割込みハンドラ、タイムイベントハンドラ、CPU 例外ハンドラ
(2) ディスパッチャ (カーネルの処理の一部)
(3) タスク
なお、ディスパッチャとは、実行するタスクを切り換えるカーネルの処理のことです。割込みハンドラ、タイムイベントハンドラ、CPU 例外ハンドラはディスパッチャよりも優先順位が高いため、これらが実行している間は、タスクは実行されません。

割込みハンドラは、割込みレベルが高いほど優先順位が高くなります。
タイムイベントハンドラの優先順位は、タイマ割込みレベル(clock.IPL)と同じとなります。
CPU 例外ハンドラの優先順位は、CPU 例外が発生した処理の優先順位と、ディスパッチャの優先順位のいずれよりも高く、かつ CPU 例外が発生した処理よりも高い優先順位を持つ他のいずれの処理よりも低くなります。
タスク間の優先順位は、タスクに付与された優先度に従います。
拡張サービスコールルーチンの優先順位は、拡張サービスコールを呼び出した処理の優先順位よりも高く、かつ拡張サービスコールルーチンを呼び出した処理よりも高い優先順位を持つ他のいずれの処理よりも低くなります。
タスク例外処理ルーチンの優先順位は、当該タスクよりも高くかつ他の優先度の高いタスクよりも低くなります。

また以下のサービスコールを呼び出した場合は、一時的に上記に該当しない優先順位を作り出すことができます。

- ◆ dis_dsp を呼び出すと、その処理の優先順位は上記の(1)と(2)の間となります。この状態は、ena_dsp を呼び出すことによって元に戻ります。
- ◆ loc_cpu, iloc_cpu を呼び出すと、その処理の優先順位は割込みレベルがカーネル割込みマスクレベル(system.system_IPL)である割込みハンドラと同じになります。この状態は、unl_cpu, iunl_cpu を呼び出すことによって元に戻ります。
- ◆ SR レジスタの IMASK ビットを 0 以外に変更している間の優先順位は、そのレベルの割込みハンドラと同じとなります。

4.8 割込み

割込みが発生すると、あらかじめユーザが定義した割込みハンドラが起動されます。割込みハンドラを定義していない場合は、システムダウンとなります。

割込みハンドラは、非タスクコンテキストで実行します。

4.8.1 割込みハンドラの種類

割込みハンドラには、以下の 2 種類の分類があります。

- カーネル管理割込みハンドラとカーネル管理外割込みハンドラ
- ダイレクト割込みハンドラとノーマル割込みハンドラ

(1) カーネル管理割込みハンドラとカーネル管理外割込みハンドラ

◆ カーネル管理割込みハンドラ

カーネル割込みマスクレベル(system.system_IPL)以下の割込みレベルの割込みハンドラをカーネル管理割込みハンドラと呼びます。カーネル管理割込みハンドラ内では、非タスクコンテキストから呼び出し可能なサービスコールを呼び出すことができます。しかし、カーネル処理中に発生したカーネル管理割込みハンドラは、カーネル管理割込みを受け付け可能となるまで遅延される場合があります。

◆ カーネル管理外割込みハンドラ

カーネル割込マスクレベル(system.system_IPL)より高い割込みレベルの割込みハンドラをカーネル管理外割込みハンドラと呼びます。サービスコール処理中に発生したカーネル管理外割込みは、カーネル処理中かどうかに関わらず、直ちに受け付けられます。しかし、カーネル管理外割込ハンドラ内では、サービスコールを呼び出してはならないという制限があります。

(2) ダイレクト割込みハンドラとノーマル割込みハンドラ

ダイレクト割込みハンドラは、割込み発生時にカーネルを介さずに起動されます。一方、ノーマル割込みハンドラは、カーネルを介して起動されます。このため、ダイレクト割込みハンドラの方が低オーバヘッドです。

以下を参考に、どちらのタイプのハンドラとして記述するかを決定してください。

(a) 記述形式

ノーマル割込みハンドラは、C言語関数として記述します。

一方ダイレクト割込みハンドラは、割込み関数として記述します。具体的には、#pragma interruptディレクティブの指定が必要です。このため、ハンドラ関数のポータビリティはノーマル割込みハンドラよりも劣ります。また、#pragma interruptの指定方法は、以下の項目に依存します。詳細は、「12.5.4 ダイレクト割込みハンドラ」を参照してください。

- カーネル管理割込みハンドラとカーネル管理外割込みハンドラ(割込みレベルがカーネル割込みマスクレベル(system.system_IPL)より高いか否か)
- その割込みがレジスタバンクを使用するか否か

RENESAS

(b) 割込みレベルについて
カーネル管理外割込みハンドラは、必ずダイレクト割込みハンドラとして実装しなければなりません。また、「4.8.3 サービスコールの制限」に記載のように、そのハンドラではサービスコールを呼び出してはなりません。

(c) スタック
ノーマル割込みハンドラは、システムにひとつの「割込みスタック」を使用します。割込みハンドラ用スタックのサイズは、cfgファイルのsystem.stack_sizeで指定します。
一方、ダイレクト割込みハンドラのスタックは、アプリケーションで確保し、ダイレクト割込みハンドラの先頭でそのスタックに切り替え、ハンドラ終了時にスタックを元に戻さなければなりません。なお、この処理は#pragma interruptディレクティブの"sp="指定により実現されます。
また、同じ割込みレベルのハンドラは、同じスタックを共有することができます。

(d) 呼び出し可能なサービスコール
いずれのハンドラも、非タスクコンテキストで動作するため、(b)に記載のケースを除いて非タスクコンテキスト用のサービスコールを呼び出すことができます。

(e) 定義方法
ダイレクト割込みハンドラを定義する場合は、VTA_DIRECT属性の指定が必要です。

4.8.2 割込み制御方法(SR レジスタの IMASK ビット)

SH マイコンでは、SR レジスタの IMASK ビットによって、CPU が受理する割込みレベルを制御できるようになっています。

(1) サービスコール内の IMASK 制御

サービスコール内の割込み禁止・許可の制御は、SR レジスタの IMASK ビットの操作により行っています。サービスコール内では、処理の不可分性を保証するために必要に応じて IMASK をカーネル割込みマスクレベル(system.system_IPL)に変更して実行します。この区間をカーネルのクリティカルセクションと呼びます。それ以外の部分は、サービスコール呼び出し前の IMASK の状態で実行します。図 4.12に、サービスコール内での割込み制御の様子を示します。

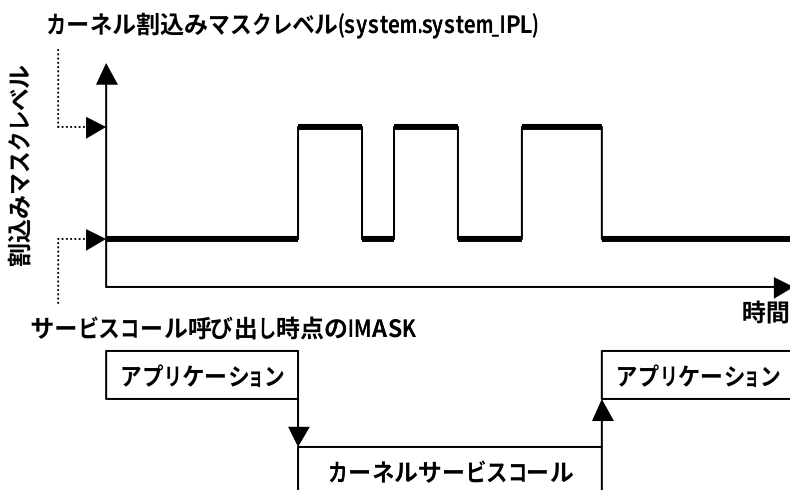


図4.12　サービスコール内の割込み制御

(2) アプリケーションでの IMASK 制御

図 4.12に示すように、IMASK はサービスコール内で変化します。アプリケーションで IMASK を変更するには、以下の 3 つの方法があります。

方法1：loc_cpu, iloc_cpuを使用する

IMASKはカーネル割込みマスクレベルに変更されます。CPUロック状態に遷移するため、タスクディスパッチは遅延されます。

また、CPUロック状態の間にこの方法でIMASKをカーネル割込みマスクレベルより小さい値に変更してはなりません。さらに、この方法でCPUロック状態に遷移させた後に、方法3でIMASKをカーネル割込みマスクレベルより大きい値に変更した場合は、CPUロック状態を解除する前にIMASKを元に戻さなければなりません。

方法2：chg_ims, ichg_imsを使用する

IMASKを任意の値に変更できます。ただし、CPUロック状態の時は使用してはなりません。

タスクコンテキストからIMASKを0以外に変更した場合、その間は非タスクコンテキストと扱われます。これにより、発行可能なサービスコールおよびサービスコールのスタック使用サイズがタスクコンテキストの場合と異なることに注意してください。また、IMASKを0に戻す場合も、非タスクコンテキスト用のichg_imsを使用してください。

また、非タスクコンテキストでは、タスクディスパッチは遅延されます。

方法3：直接IMASKを変更する(コンパイラが提供する組み込み関数set_imask()またはset_cr()を使用する)

CPUロック状態でも使用できる点を除いてchg_imsの場合と同じ振る舞いになりますが、以下の違いがあります。

- chg_imsよりも低オーバヘッドです。
- タスクコンテキストから、この方法でIMASKを変更する場合は制限があります。詳細は、以降の説明を参照してください。

以下、想定される各シーンについて説明します。

(a) タスクコンテキストでの IMASK 制御

- 特定のレベルでマスクする
  方法2または方法3を使用できますが、性能のためには方法3を推奨します。ただし、IMASKをカーネル割込みマスクレベル未満の値に変更する場合は、方法3を使用してはなりません。
- カーネル割込みマスクレベルでマスクする
  すべての方法を使用できます。ポータビリティのためには方法1、性能のためには方法3を推奨します。

(b) 非タスクコンテキストでの IMASK 制御

非タスクコンテキストでは、IMASK を起動時の値より下げてはなりません。

- 特定のレベルでマスクする
  方法 2 または方法 3 の方法を使用できますが、性能のためには方法 3 を推奨します。
- カーネル割込みマスクレベルでマスクする
  すべての方法を使用できます。ポータビリティのためには方法 1、性能のためには方法 3 を推奨します。

4.8.3 サービスコールの制限

割込みハンドラかどうかに関わらず、カーネルの処理の不可分性を保証するために、SR.IMASK がカーネル割込みマスクレベル(system.system_IPL)より高い時は、サービスコールを呼び出してはなりません。サービスコールを呼び出すと、サービスコール処理中に割込みマスクレベルが下がるため、意図しない割込みを受理してしまい、異常動作となります。

4.9 CPU 例外

アドレスエラーや TRAPA 命令などの CPU 例外が発生すると、あらかじめユーザが定義した CPU 例外ハンドラが起動されます。CPU 例外ハンドラを定義していない場合は、システムダウンとなります。

4.9.1 CPU 例外ハンドラの種類

CPU 例外ハンドラには、ダイレクト CPU 例外ハンドラとノーマル例外ハンドラがあります。
ダイレクト CPU 例外ハンドラは、CPU 例外発生時にカーネルを介さずに起動されます。一方、ノーマル CPU 例外ハンドラは、カーネルを介して起動されます。
以下を参考に、どちらのタイプのハンドラとして記述するかを決定してください。

(a) 記述形式
ノーマルCPU例外ハンドラは、C言語関数として記述します。ノーマルCPU例外ハンドラには、発生した例外番号(ベクタ番号)とCPU例外情報がパラメータとして渡されます。
一方、ダイレクトCPU例外ハンドラは、割込み関数として記述します。具体的には、#pragma interruptディレクティブの指定が必要です。このため、ハンドラ関数のポータビリティはノーマルCPU例外ハンドラよりも劣ります。
また、ダイレクトCPU例外ハンドラには、何もパラメータは渡されません。ダイレクトCPU例外ハンドラでCPU例外発生時の情報を取得するには、ハンドラをアセンブリ言語で記述する必要があります。

(b) 実行コンテキスト
いずれのハンドラも、CPU例外発生前と同じコンテキスト状態、同じスタックで実行します。このため、CPU例外発生元のスタックのオーバフローに注意してください。
ノーマルCPU例外ハンドラでは、CPU例外情報をスタックに生成するため、CPU例外発生元のスタックの消費量がダイレクトCPU例外ハンドラよりも大きくなります。
また、タスクコンテキストなど、ディスパッチが保留されていない状態で発生したCPU例外の場合、ノーマルCPU例外ハンドラ実行中はタスクディスパッチは保留されますが、ダイレクトCPU例外ハンドラ実行中はタスクディスパッチは保留されません。

(c) 呼び出し可能なサービスコール
ノーマルCPU例外ハンドラから呼び出し可能なサービスコールは、表4.4のように制限されます。ただし、「4.8.3 サービスコールの制限」に注意してください。

表4.4 ノーマル CPU 例外ハンドラで使用可能なサービスコール

get_tid, iget_tid	ras_tex, iras_tex	sns_tex	sns_ctx	sns_loc
sns_dsp	sns_dpn	vsta_knl, ivsta_knl	vsys_dwn, ivsys_dwn	vsns_tmr

一方、ダイレクトCPU例外ハンドラでは、カーネル内で発生したCPU例外の場合は、サービスコールを呼び出してはなりません。アプリケーション内で発生したCPU例外の場合は、CPU例外発生前に許可されていたのと同じサービスコールを呼び出すことができます。

(d) 定義方法
ダイレクトCPU例外ハンドラを定義する場合は、VTA_DIRECT属性の指定が必要です。

4.9.2 予約例外

TRAPA #60～TRAPA #63 はカーネル用に予約されているため、そのハンドラを定義することはできません。

RENESAS

5. カーネルの機能

本章では、主にカーネルサービスコールの機能やその使い方について解説します。

5.1 タスク管理機能

タスク管理機能とは、タスクの生成、削除、起動、終了、優先度の変更等のタスク操作を行う機能です。

なお、タスクのスタックについては「4.5.4 タスクのスタック」を参照してください。

タスク管理機能のサービスコールには、次のものがあります。

(1) タスクを生成する(cre_tsk, icre_tsk)

指定された ID で、タスクを生成します。

(2) タスクを生成する(acre_tsk, iacre_tsk)

任意の ID で、タスクを生成します。ID は、カーネルが自動的に割り当て、リターン値として返します。

(3) タスクを削除する(del_tsk)

指定された ID のタスクを削除します。

(4) タスクを起動する(act_tsk, iact_tsk)

指定された ID のタスクを起動します。本サービスコールは sta_tsk, ista_tsk とは異なり、起動要求は蓄積されますが、対象タスクに渡る起動コードを指定することはできません。対象タスクには、そのタスクを生成するときに指定した拡張情報が渡されます。

act_tsk は、他 CPU のタスクに対して要求することができます。

(5) 蓄積していたタスク起動要求を無効にする(can_act, ican_act)

指定された ID のタスクに蓄積されていた起動要求を無効にします。

can_act は、他 CPU のタスクに対して要求することができます。

(6) タスクを起動する(sta_tsk, ista_tsk)

指定された ID のタスクを起動します。本サービスコールは act_tsk, iact_tsk とは異なり、起動要求は蓄積されませんが、対象タスクに渡る起動コードを指定することができます。

sta_tsk は、他 CPU のタスクに対して要求することができます。

(7) 自タスクを終了する(ext_tsk)

自タスクを終了させ、休止状態に移行させます。起動要求が蓄積されている場合は、再度タスクの起動処理を行います。その際、自タスクはリセットされたように振る舞います。

タスクの開始関数からリターンした場合は、本サービスコールと同じ振る舞いになります。

(8) 自タスクを終了させ、削除する(exd_tsk)

自タスクを終了させ、さらに削除します。

(9) 他タスクを強制終了させる(ter_tsk)

休止状態以外の他タスクを終了させ、休止状態に移行させます。起動要求が蓄積されている場合は、再度タスクの起動処理を行います。

なお、vchg_tmd により、強制終了要求をマスクすることができます。

本サービスコールは、他 CPU のタスクに対して要求することができます。

(10)タスクの優先度を変更する(chg_pri, ichg_pri)

指定された ID のタスクの優先度を変更します。

タスクの優先度を変更すると、そのタスクが実行可能状態または実行状態であるときは、レディキューも更新されます(図 5.1参照)。

また、対象タスクが TA_TPRI 属性を持つオブジェクトの待ち行列につながれている場合は、待ち行列も更新されます(図 5.2参照)。

chg_pri は、他 CPU のタスクに対して要求することができます。

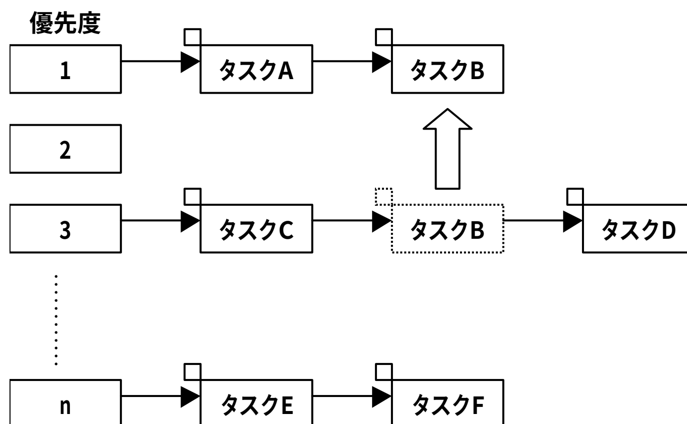
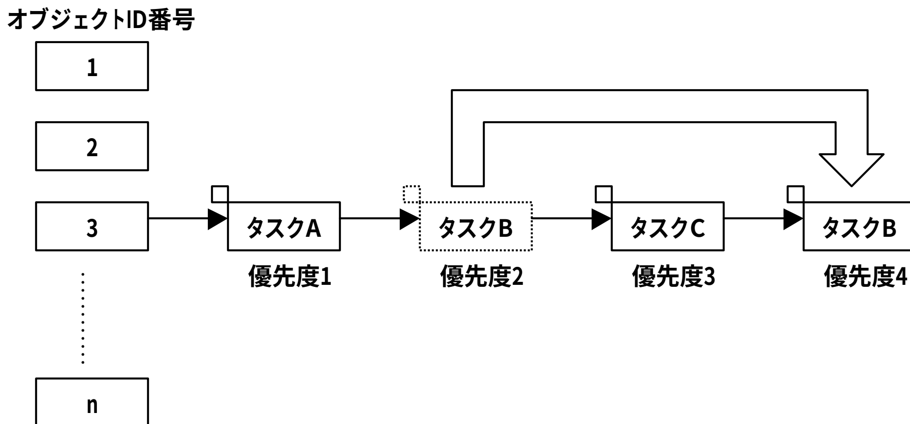


図5.1 優先度の変更

RENESAS



タスクBの優先度を2から4に変更した場合

図5.2　待ち行列のつなぎ換え

一般には、優先度の変更はシステム全体の振る舞いに影響を与えるため、本サービスコールを使用しないでシステム設計を行うことが推奨されます。

なお、タスクの優先度には「ベース優先度」と「現在優先度」の２つがあります。通常は、この２つは同じですが、ミューテックスをロックしている間だけ異なります。詳細は、「5.9 ミューテックス」を参照してください。

(11)タスクの優先度を取得する(get_pri, iget_pri)

指定された ID のタスクの優先度を取得します。

get_pri は、他 CPU のタスクに対して要求することができます。

(12)タスクの状態を参照する(ref_tsk, iref_tsk)

指定された ID のタスクの状態を参照します。

ref_tsk は、他 CPU のタスクに対して要求することができます。

(13)タスクの状態を参照する(簡易版)(ref_tst, iref_tst)

指定された ID のタスクの状態を参照します。ref_tsk, iref_tsk と比べ、参照する情報が少ないためにオーバヘッドが小さいのが特徴です。

ref_tst は、他 CPU のタスクに対して要求することができます。

(14)タスクの実行モードを変更する(vchg_tmd)

指定された ID のタスク実行モードを変更します。

タスク実行モードは、μITRON4.0 仕様外の機能です。

タスクは、確保した資源を解放する前に、他タスクからの強制終了要求(ter_tsk サービスコール)によって意図しないタイミングで休止状態になる可能性があります。また、タスクは sus_tsk, isus_tsk によって予期しないタイミングで実行が中断する可能性があります。

vchg_tmd サービスコールを用いることで、これらの要求をマスクすることができます。

5.2 タスク付属同期機能

タスク付属同期機能とは、タスク間の同期をとるためにタスクを待ち状態 (もしくは強制待ち状態・二重待ち状態)にしたり、待ち状態になったタスクを起床させたりする機能です。

タスク付属同期機能のサービスコールには、次のものがあります。

(1) 待ち状態への移行(slp_tsk, tslp_tsk)と、その起床(wup_tsk, iwup_tsk)

slp_tsk は、自タスクを待ち状態へ移行させます。

tslp_tsk は、slp_tsk に対してタイムアウトを指定できるようにしたサービスコールです。

wup_tsk および iwup_tsk は、slp_tsk または tslp_tsk による待ち状態のタスクを起床します。指定したタスクが slp_tsk または tslp_tsk による待ち状態でない場合は、起床要求が蓄積されます。起床要求が蓄積されているタスクが slp_tsk または tslp_tsk を呼び出すと、起床要求が減算(-1)されるだけで待ち状態には移行しません(図 5.3参照)。

wup_tsk は、他 CPU のタスクに対して要求することができます。

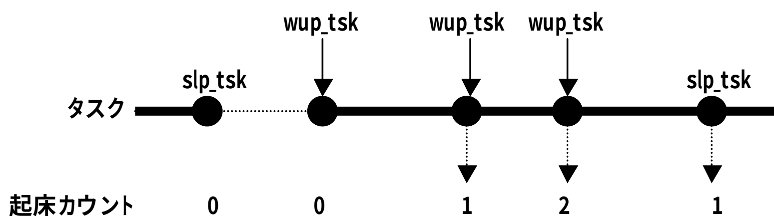


図5.3 起床要求の蓄積

(2) タスクの起床要求を無効にする(can_wup, ican_wup)

指定された ID のタスクに蓄積されていた起床要求をクリアします(図 5.4参照)。

can_wup は、他 CPU のタスクに対して要求することができます。

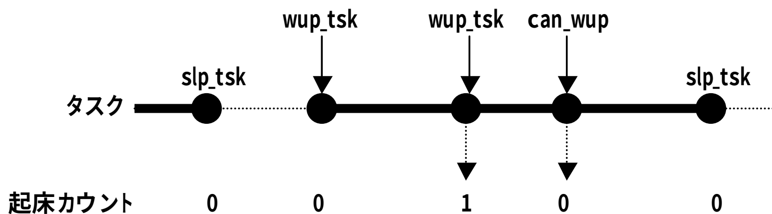


図5.4 起床要求のキャンセル

RENESAS

(3) 強制待ち要求(sus_tsk, isus_tsk)と、その再開(rsm_tsk, irsm_tsk, frsm_tsk, ifrsm_tsk)

sus_tsk, isus_tsk は、指定された ID のタスクの実行を強制的に中断させます。対象タスクは、実行状態または実行可能状態の場合は強制待ち状態に、待ち状態の場合は二重待ち状態に移行します。

sus_tsk, isus_tsk による強制待ち要求はネストされます。

rsm_tsk, irsm_tsk は、指定された ID のタスクの強制待ち要求ネストを減算(-1)します。その結果、ネスト数が 0 になれば、対象タスクは元の状態に戻ります(図 5.5参照)。

frsm_tsk, ifrsm_tsk は、指定された ID のタスクの強制待ち要求ネストをクリアし、対象タスクを強制的に元の状態に戻します(図 5.6参照)。

sus_tsk, rsm_tsk, frsm_tsk は、他タスクのカーネルに対して要求することができます。

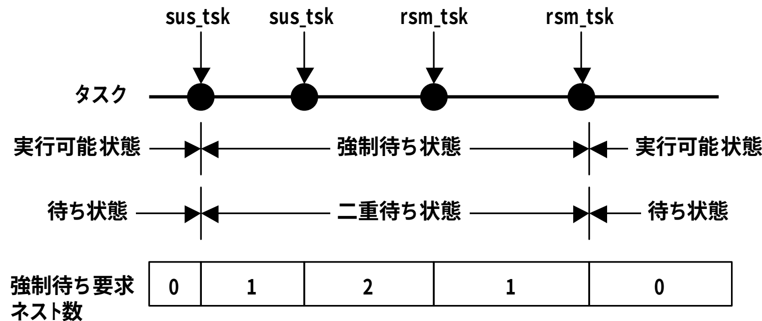


図5.5 タスクの強制待ちと再開

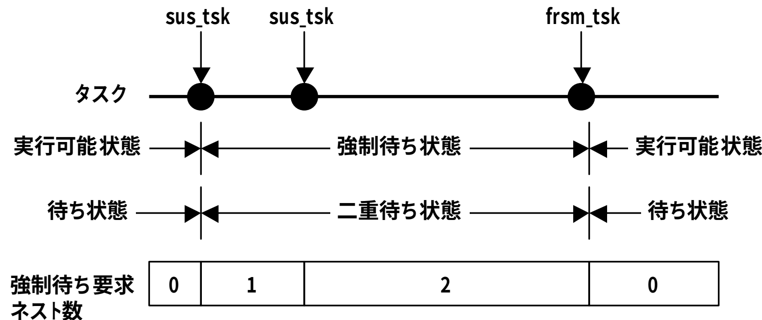


図5.6 タスクの強制待ちと強制再開

(4) タスクの待ち状態を強制解除する(rel_wai, irel_wai)

本サービスコールは、指定された ID のタスクの待ち状態を強制的に解除します。なお、本サービスコールで強制待ちを解除することはできません。

rel_wai は、他 CPU のタスクに対して要求することができます。

(5) 自タスクの一定時間待ち状態に移行する(dly_tsk)

自タスクを一定時間待たせます。自タスクは待ち状態に移行します。

5.3 タスク付属イベントフラグ

タスク付属イベントフラグとはタスクに付与されたビットパターンで､タスクはタスク付属イベントフラグの指定したいずれかのビットのセット（事象の発生）を待つことができます。

本機能は、μITRON4.0 仕様外の機能です。

図 5.7にタスク付属イベントフラグの動作例を示します。

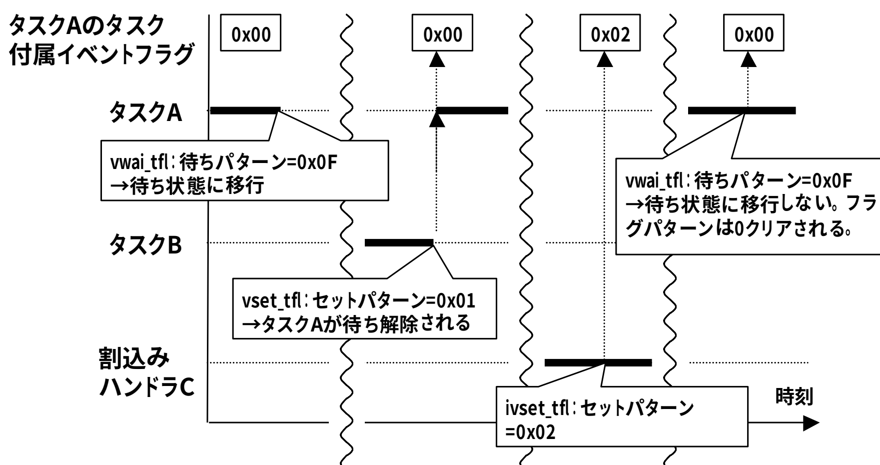


図5.7 タスク付属イベントフラグの動作例

タスク付属イベントフラグを操作するサービスコールには、次のものがあります。

(1) タスク付属イベントフラグを待つ(vwai_tfl, vtwai_tfl)

タスク付属イベントフラグに、指定したいずれかのビットがセットされるのを待ちます。いずれかのビットがセットされると、待ち状態が解除され、タスク付属イベントフラグは 0 クリアされます。また、そのクリア前のタスク付属イベントフラグのビットパターンが返ります。

既に指定したいずれかのビットがセットされている場合は、待ち状態に移行しません。

(2) タスク付属イベントフラグを得る(vpol_tfl)

タスク付属イベントフラグに、指定したいずれかのビットがセットされているか否かを調べます。vwai_tfl, vtwai_tfl と異なるのは、待ち条件を満たさない場合に、待ち状態にはならずに直ちにエラーリターンする点です。

(3) タスク付属イベントフラグをセットする(vset_tfl, ivset_tfl)

指定された ID のタスクのタスク付属イベントフラグに対し、指定されたビットをセットします。

vset_tfl は、他 CPU のタスクに対して要求することができます。

(4) タスク付属イベントフラグをクリアする(vclr_tfl, ivclr_tfl)

指定された ID のタスクのタスク付属イベントフラグに対し、指定されたビットをクリアします。

vclr_tfl は、他 CPU のタスクに対して要求することができます。

RENESAS

5.4　タスク例外処理

タスク例外処理機能は、タスクに発生した例外事象の処理を、タスク本体の処理とは非同期に行うための機能で、一般に「シグナル」と呼ばれている機能に類似しています。

図 5.8にタスク例外処理の動作例を示します。

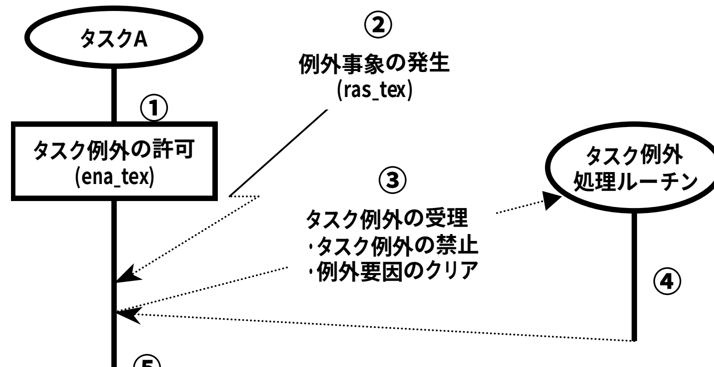


図5.8　タスク例外処理の動作例

図の解説

① タスクAが、タスク例外を許可します。
② タスクA実行中に、ras_texによってタスクAにタスク例外が要求されます。このとき、例外要因パターンを指定します。
③ タスクAにスケジュールされた時、タスクA本体の処理は実行されず、タスクAのタスク例外処理ルーチンが起動されます。このとき、タスク例外は禁止状態となり、保留例外要因パターンもクリアされます。
④ タスク例外処理ルーチンを実行します。タスク例外処理ルーチンには、保留していた例外要因パターンが渡されます。
⑤ タスク例外処理ルーチンからリターンすると、タスクA本体処理を実行を再開します。

タスク例外を扱うサービスコールには、次のものがあります。

(1) タスク例外処理ルーチンを定義する(def_tex, idef_tex)

指定された ID のタスクに対し、タスク例外処理ルーチンを定義します。

(2) タスク例外を要求する(ras_tex, iras_tex)

指定された ID のタスクに対し、タスク例外を要求します。このとき、例外要因パターンを指定します。

(3) タスク例外を許可する(ena_tex)

自タスクのタスク例外を許可します。

(4) タスク例外を禁止する(dis_tex)

自タスクのタスク例外を禁止します。

(5) タスク例外禁止状態を調べる(sns_tex)

自タスクがタスク例外禁止状態かどうかを調べます。

(6) タスク例外処理状態を参照する(ref_tex, iref_tex)

指定された ID のタスクのタスク例外処理状態を参照します。

RENESAS

5.5 セマフォ

セマフォは、複数のタスクで共有する装置や共有変数といった資源の競合を防ぐためのオブジェクトです。例えば、ある共有変数に対してタスク A が更新中にタスクスイッチが発生し、タスク B が共有変数を参照すると、タスク B は更新途中の不正な状態の共有変数を読み出してしまう可能性があります。セマフォを使用することで、このような競合を回避することができます。

セマフォは、このような資源の有無や数をカウンタで表現することで排他制御や同期機能を提供するオブジェクトです。

セマフォと実際に排他制御したい資源を対応付けるのは、アプリケーション側の責任です。

セマフォを使用した排他制御を行うために重要なことは、以下のルールを守ることです。

(1) 資源を使用する前にセマフォを獲得する
(2) 資源の使用を終えたらセマフォを解放する

図 5.9にセマフォの動作例を示します。

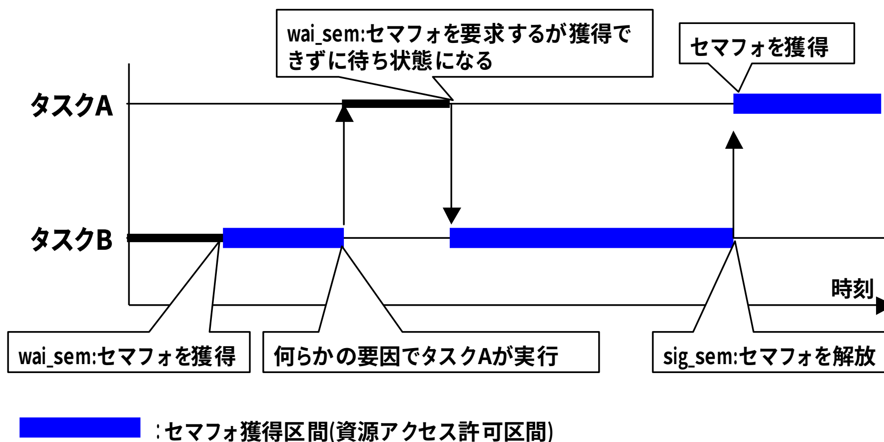


図5.9 セマフォの動作例

セマフォを操作するサービスコールには、次のものがあります。

(1) セマフォを生成する(cre_sem, icre_sem)

指定された ID で、セマフォを生成します。

(2) セマフォを生成する(acre_sem, iacre_sem)

任意の ID で、セマフォを生成します。ID は、カーネルが自動的に割り当て、リターン値として返します。

(3) セマフォを削除する(del_sem)

セマフォを削除します。

(4) セマフォを獲得する(wai_sem, twai_sem)

セマフォを獲得します。セマフォカウンタが正なら、セマフォカウンタを 1 減算します。セマフォカウンタが 0 の場合は、セマフォを得ることができないので、本サービスコールは呼び出しタスクを待ち状態に移行させます。

wai_sem, twai_sem は、他 CPU のセマフォに対して要求することができます。

(5) セマフォを獲得する(pol_sem, ipol_sem)

セマフォを獲得します。wai_sem, twai_sem と異なるのは、セマフォカウンタが 0 の場合に、待ち状態にはならずに直ちにエラーリターンする点です。

pol_sem は、他 CPU のセマフォに対して要求することができます。

(6) セマフォを返却する(sig_sem, isig_sem)

セマフォを返却します。本サービスコールは、セマフォを待っているタスクがあればそのタスクの待ち状態を解除し、なければセマフォカウンタを 1 加算します。

sig_sem は、他 CPU のセマフォに対して要求することができます。

(7) セマフォの状態を参照する(ref_sem, iref_sem)

セマフォのカウンタ値や待ちタスク ID を参照します。

ref_sem は、他 CPU のセマフォに対して要求することができます。

RENESAS

5.5.1 優先度逆転問題

セマフォを用いた排他制御では、優先度逆転という問題が発生するケースがあります。優先度逆転とは、資源を要求するタスクの実行が、資源を使用しない別のタスクによって遅延されてしまうという現象です。

この様子を図 5.10に示します。この図では、タスク A とタスク C は同じ資源を使用し、タスク B はその資源を使用しない例になっています。タスク A は資源を使用するためにセマフォを獲得しようとしますが、既にタスク C がセマフォを獲得しているので待ち状態になります。ところが、タスク C がセマフォを解放する前に、優先度がタスク C よりも高くてタスク A より低く、かつ資源とは関係のないタスク B が実行すると、タスク C によるセマフォの解放はタスク B の実行によって遅れ、その結果タスク A がセマフォを獲得するのも遅れます。タスク A の立場では、資源競合しておらず、かつ自分より優先度の低いタスク B が優先的に実行されてしまうことになります。

この問題を回避するには、セマフォではなく、ミューテックスを使用してください。

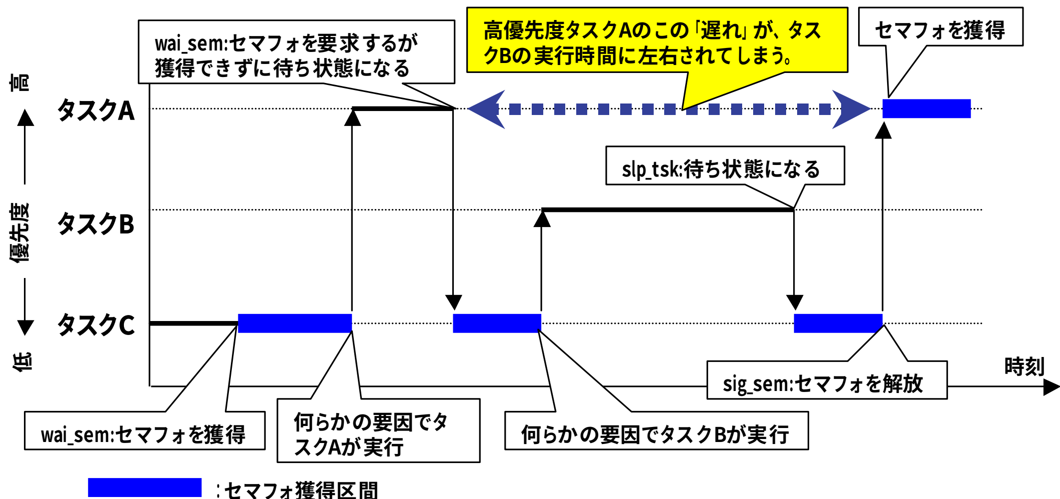


図5.10 優先度逆転現象

5.6 イベントフラグ

イベントフラグは、事象に対応したビットの集合で、1 つの事象が 1 ビットで表わされます。タスクは、イベントフラグの指定したビット郡のすべて(AND 条件)またはいずれか(OR 条件)がセットされるのを待つことができます。

また、複数のタスクが事象の発生を待つことができるかどうかを、生成時に属性として指定することができます。

- TA_WMUL 属性(複数タスクの待ち可能)
- TA_WSGL 属性(複数タスクの待ち禁止)

また、TA_CLR 属性を指定することにより、タスクがイベントフラグ待ち条件を満たしたときにイベントフラグを 0 クリアすることができます。

イベントフラグには、複数のタスクを同時に待ち解除できるという特徴があります。複数タスクの同時待ち解除を行うには、TA_WMUL 属性を指定し、かつ TA_CLR 属性を指定しないようにします。この場合の動作例を、図 5.11に示します。

図 5.11では、タスク A からタスク F までの 6 個のタスクが待ち行列につながれています。そして、set_flg によって、フラグパターンを 0x0F にすると、待ち条件を満たすタスクが待ち行列の前から順に外されていきます。この図では、タスク A、タスク C、タスク E が該当します。

もし、このイベントフラグに TA_CLR 属性が指定されていれば、タスク A の待ちが解除された時点でイベントフラグは 0 クリアされるため、タスク C、タスク E は待ち解除されません。

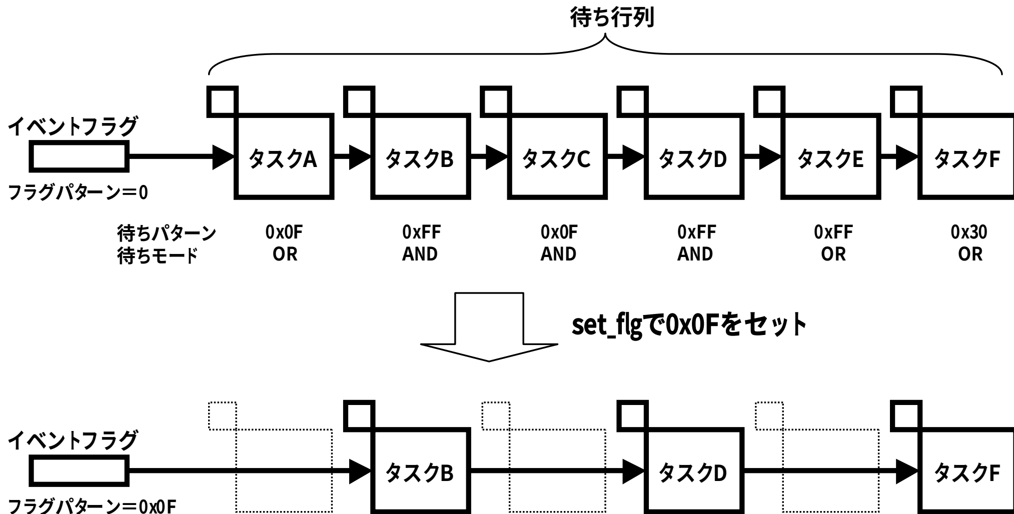


図5.11 イベントフラグの動作例

イベントフラグを操作するサービスコールには、次のものがあります。

(1) イベントフラグを生成する(cre_flg, icre_flg)

指定された ID で、イベントフラグを生成します。

(2) イベントフラグを生成する(acre_flg, iacre_flg)

任意の ID で、イベントフラグを生成します。ID は、カーネルが自動的に割り当て、リターン値として返します。

(3) イベントフラグを削除する(del_flg)

イベントフラグを削除します。

(4) イベントフラグを待つ(wai_flg, twai_flg)

指定されたビットがイベントフラグにセットされるのを待ちます。以下のいずれかの待ちモードを指定します。

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

待ち解除時には、待ち解除直前のイベントフラグ値が呼び出し元タスクに返されます。対象イベントフラグに TA_CLR 属性が指定されていた場合は、この時イベントフラグが 0 クリアされます。この場合、返される値はクリア直前のイベントフラグ値です。

wai_flg, twai_flg は、他 CPU のイベントフラグに対して要求することができます。

(5) イベントフラグを得る(pol_flg, ipol_flg)

指定されたビットがイベントフラグにセットされているかどうかを調べます。wai_flg, twai_flg と異なるのは、待ち条件を満たさない場合に、待ち状態にはならずに直ちにエラーリターンする点です。

pol_flg は、他 CPU のイベントフラグに対して要求することができます。

(6) イベントフラグをセットする(set_flg, iset_flg)

イベントフラグに対し、指定されたビットをセットします。これにより、このイベントフラグで待っていたタスクの待ち状態が解除される場合があります。

set_flg は、他 CPU のイベントフラグに対して要求することができます。

(7) イベントフラグをクリアする(clr_flg, iclr_flg)

イベントフラグに対し、指定されたビットをクリアします。

clr_flg は、他 CPU のイベントフラグに対して要求することができます。

(8) イベントフラグの状態を参照する(ref_flg, iref_flg)

イベントフラグのビットパターンや待ちタスク ID を参照します。

ref_flg は、他 CPU のイベントフラグに対して要求することができます。

5.7 データキュー

データキューとは、1 ワード(32 ビット)のデータ通信を行うオブジェクトです。図 5.12に、データキューの概要を示します。

図5.12 データキュー

データキューに送信されたデータは蓄積されます。データキューからの受信では、最も古いデータから順に取り出されます(FIFO)。データキューに蓄積可能なデータ数は、データキューを生成する時に指定します。

データキュー領域は、カーネルが持つデフォルトデータキュー用領域から割り付ける方法と、アプリケーションが確保した領域をデータキュー領域として使用する方法があり、データキューを生成するときにどちらにするかを指定します。なお、デフォルトデータキュー用領域のサイズは、cfg ファイルの memdtq.all_memsize に指定します。

データキューを操作するサービスコールには、次のものがあります。

(1) データキューを生成する(cre_dtq, icre_dtq)

指定された ID で、データキューを生成します。

(2) データキューを生成する(acre_dtq, iacre_dtq)

任意の ID で、データキューを生成します。ID は、カーネルが自動的に割り当て、リターン値として返します。

(3) データキューを削除する(del_dtq)

データキューを削除します。

(4) データを送信する(snd_dtq, tsnd_dtq)

データキューにデータを送信します。データキューがデータでいっぱいの場合は、本サービスコールは呼び出し元タスクをデータ送信待ち状態に移行させます。

snd_dtq, tsnd_dtq は、他 CPU のデータキューに対して要求することができます。

(5) データを送信する(psnd_dtq, ipsnd_dtq)

データキューにデータを送信します。snd_dtq, tsnd_dtq と異なるのは、データキューがデータでいっぱいの場合に、データ送信待ち状態にはならずに直ちにエラーリターンする点です。

psnd_dtq は、他 CPU のデータキューに対して要求することができます。

RENESAS

(6) データを強制的に送信する(fsnd_dtq, ifsnd_dtq)

データキューにデータを送信します。データキューがデータでいっぱいの場合は、最も古いデータが破棄されます。

fsnd_dtq は、他 CPU のデータキューに対して要求することができます。

(7) データを受信する(rcv_dtq, trcv_dtq)

データキューからデータを受信します。データキューにデータがない場合は、本サービスコールは呼び出し元タスクをデータ受信待ち状態に移行させます。データキューがデータでいっぱいで送信待ちタスクが存在する場合は、データ送信待ち行列先頭のタスクの待ち状態が解除されます。

rcv_dtq, trcv_dtq は、他 CPU のデータキューに対して要求することができます。

(8) データを受信する(prcv_dtq, iprcv_dtq)

データキューからデータを受信します。データキューにデータがない場合は、本サービスコールは直ちにエラーリターンします。データキューがデータでいっぱいで送信待ちタスクが存在する場合は、データ送信待ち行列先頭のタスクの待ち状態が解除されます。

prcv_dtq は、他 CPU のデータキューに対して要求することができます。

(9) データキューの状態を参照する(ref_dtq, iref_dtq)

データキューに蓄積されているデータ数や送信・受信待ちタスク ID を参照します。

ref_dtq は、他 CPU のデータキューに対して要求することができます。

5.8 メールボックス

メールボックスとは、メッセージと呼ぶ任意のサイズのデータ通信を行うオブジェクトです。図5.13に、メールボックスの概要を示します。

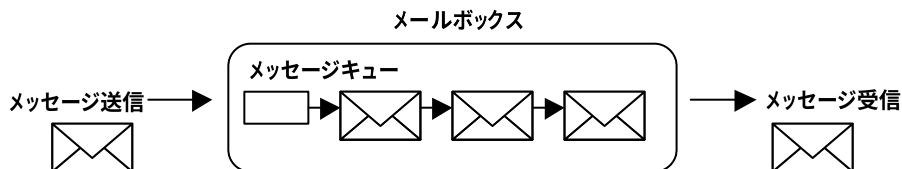


図5.13　メールボックス

メールボックスでは、メッセージアドレスの受け渡しによってデータ通信を行うため、メッセージサイズに依存しない高速な通信が行われます。

メッセージ領域は送信･受信側双方がアクセス可能なメモリにアプリケーションが作成しなければなりません（ローカル変数領域にメッセージを作成してはなりません）。また、メッセージ送信するタスクでは、メッセージ送信後にそのメッセージ領域をアクセスしてはなりません。

さらに、メールボックスを用いて CPU を跨った通信を行う場合は、以下のいずれかの処置が必要です。

(1)　メッセージを非キャッシャブル領域に作成する。
(2)　メッセージ送信前に、メッセージ内容を実メモリにライトバックさせる。

メールボックスに TA_MPRI 属性を指定すると、メッセージに優先度を設定することができます。この場合、優先度の高いメッセージから順に受信することができます。メッセージ優先度を使用したくない場合は、TA_MPRI 属性ではなく TA_MFIFO 属性を指定します。

メールボックスを操作するサービスコールには、次のものがあります。

(1) メールボックスを生成する(cre_mbx, icre_mbx)

指定された ID で、メールボックスを生成します。

(2) メールボックスを生成する(acre_mbx, iacre_mbx)

任意の ID で、メールボックスを生成します。ID は、カーネルが自動的に割り当て、リターン値として返します。

(3) メールボックスを削除する(del_mbx)

メールボックスを削除します。

(4) メッセージを送信する(snd_mbx, isnd_mbx)

メールボックスにメッセージを送信します。

snd_mbx は、他 CPU のメールボックスに対して要求することができます。

RENESAS

(5) メッセージを受信する(rcv_mbx, trcv_mbx)

メールボックスからメッセージを受信します。メールボックスにメッセージがない場合は、本サービスコールはメッセージが送信されるまで呼び出し元タスクを待ち状態に移行させます。

rcv_mbx, trcv_mbx は、他 CPU のメールボックスに対して要求することができます。

(6) メッセージを受信する(prcv_mbx, iprcv_mbx)

メールボックスからメッセージを受信します。rcv_mbx, trcv_mbx と異なるのは、メールボックスにメッセージがない場合に、待ち状態にはならずに直ちにエラーリターンする点です。

prcv_mbx は、他 CPU のメールボックスに対して要求することができます。

(7) メールボックスの状態を参照する(ref_mbx, iref_mbx)

メールボックスに入っている先頭のメッセージアドレスと待ちタスク ID を参照します。

ref_mbx は、他 CPU のメールボックスに対して要求することができます。

5.9　ミューテックス

ミューテックスは排他制御を行うためのオブジェクトです。セマフォとの主な相違は以下の通りです。

(1)　優先度逆転現象を回避するための優先度上限プロトコルをサポートしているため、「優先度逆転現象」が発生しません。

(2)　単一資源の排他制御にのみ使用できます。

(1)についてもう少し詳しく説明します。

本カーネルのミューテックスでは、優先度制御方式として優先度上限プロトコルのみをサポートしています。このプロトコルでは、タスクがミューテックスを獲得(ロック)すると、そのタスクの優先度は自動的にミューテックス生成時に指定された上限優先度まで引き上げられます。なお、本カーネルがサポートする優先度上限プロトコルは、より正確には「簡略化した優先度上限プロトコル」です。タスクがミューテクスを解放(アンロック)する際、カーネルはそのタスクが他にミューテックスをロックしていない場合のみ、そのタスクの優先度を元に戻します。

図 5.14に、ミューテックスの動作例を示します。

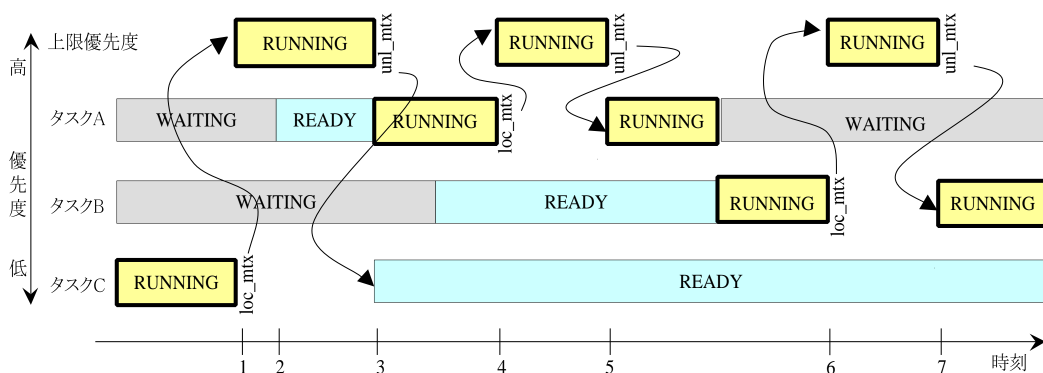


図5.14　ミューテックスの動作例

図の解説

時刻1：タスクCがloc_mtxでミューテックスをロックすると、優先度がミューテックスの上限優先度に引き上げられます。

時刻2：タスクCが上限優先度で実行中にタスクAがREADY状態になりました。タスクAの優先度は本来はタスクCよりも高いですが、タスクCはタスクAよりも高い上限優先度で実行しているため、タスクAはRUNNING状態にはなりません。すなわちタスクCは、ミューテックスをロックしている間は、本来の優先度がより高いタスクAが実行可能になっても、タスクAに邪魔されずに処理を継続することができます。

時刻3：タスクCがunl_mtxでミューテックスのロックを解除すると、タスクCは元の優先度に戻ります。この結果、優先度の高いタスクAがRUNNING状態になります。

時刻4：タスクAがloc_mtxを発行すると、タスクAの優先度は上限優先度に引き上げられます。

時刻5：タスクAがunl_mtxを発行すると、タスクAの優先度は元に戻ります。

時刻6：タスクBがloc_mtxを発行すると、タスクBの優先度は上限優先度に引き上げられます。

時刻7：タスクBがunl_mtxを発行すると、タスクBの優先度は元に戻ります。

ミューテックスを操作するサービスコールには、次のものがあります。

(1) ミューテックスを生成する(cre_mtx, icre_mtx)

指定された ID で、ミューテックスを生成します。このとき、上限優先度を指定します。

(2) ミューテックスを生成する(acre_mtx, iacre_mtx)

任意の ID で、ミューテックスを生成します。このとき、上限優先度を指定します。ID は、カーネルが自動的に割り当て、リターン値として返します。

(3) ミューテックスを削除する(del_mtx)

ミューテックスを削除します。

(4) ミューテックスをロックする(loc_mtx, tloc_mtx)

ミューテックスをロックし、優先度を上限優先度に引き上げます。他のタスクがロック中の場合は、ロックが解放されるまで本サービスコールは呼び出しタスクを待ち状態に移行させます。

(5) ミューテックスをロックする(ploc_mtx)

ミューテックスをロックし、優先度を上限優先度に引き上げます。loc_mtx, tloc_mtx と異なるのは、他のタスクがロック中の場合に、待ち状態にはならずに直ちにエラーリターンする点です。

(6) ミューテックスのロックを解除する(unl_mtx)

ミューテックスのロックを解除します。本ミューテックスのロックを待っているタスクがあればそのタスクの待ち状態を解除します。

(7) ミューテックスの状態を参照する(ref_mtx, iref_mtx)

ミューテックスをロックしているタスク ID や、待ちタスク ID を参照します。

5.9.1 ベース優先度と現在優先度

タスクの優先度には、ベース優先度と現在優先度があります。タスクのスケジューリングは、現在優先度に従って行われます。

ミューテックスをロックしていない時は、両者は常に同じです。

ミューテックスをロックすると、現在優先度のみがそのミューテックスの上限優先度に引き上げられます。

タスクの優先度を変更する chg_pri, ichg_pri では、ミューテックスをロックしていないタスクの場合は、ベース優先度・現在優先度とも変更されますが、ミューテックスをロックしているタスクの場合はベース優先度のみが変更されます。また、ミューテックスロック中またはミューテックスのロックを待っているタスクの場合は、ロック中またはロックを待っているミューテックスのいずれかの上限優先度よりも高い優先度を指定すると、E_ILUSE エラーになります。

なお、get_pri, iget_pri を用いると、現在優先度を参照することができます。

5.10 メッセージバッファ

メッセージバッファは、メールボックスと同様に任意のサイズのデータ通信を行うオブジェクトです。メールボックスと異なる点は、データ内容そのものがコピーされる点です。このため、メッセージ送信後は相手が受信したかどうかに関わらず、直ちに送信したメッセージ領域を再利用することができます。図 5.15に、メッセージバッファの概要を示します。

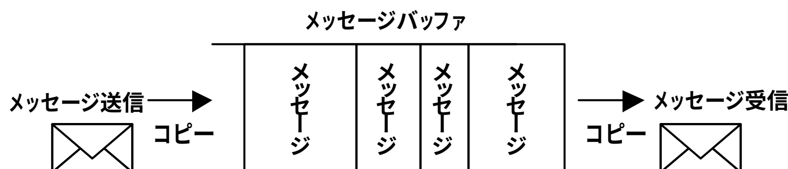


図5.15 メッセージバッファ

メッセージバッファには、メールボックスと同様にメッセージが蓄積されます。蓄積されたメッセージは、FIFO 順に取り出されます。

メッセージバッファ領域は、カーネルが持つデフォルトメッセージバッファ用領域から割り付ける方法と、アプリケーションが確保した領域をメッセージバッファ領域として使用する方法があり、メッセージバッファを生成するときにどちらにするかを指定します。なお、デフォルトメッセージバッファ用領域のサイズは、cfg ファイルの memmbf.all_memsize に指定します。

メッセージバッファを操作するサービスコールには、次のものがあります。

(1) メッセージバッファを生成する(cre_mbf, icre_mbf)

指定された ID で、メッセージバッファを生成します。

(2) メッセージバッファを生成する(acre_mbf, iacre_mbf)

任意の ID で、メッセージバッファを生成します。ID は、カーネルが自動的に割り当て、リターン値として返します。

(3) メッセージバッファを削除する(del_mbf)

メッセージバッファを削除します。

(4) メッセージを送信する(snd_mbf, tsnd_mbf)

メッセージバッファにメッセージを送信します。

メッセージバッファメッセージを送信するには、メッセージバッファに以下の空きサイズが必要です。

(送信メッセージサイズを 4 の倍数に切り上げたサイズ) + 4

メッセージバッファの空きサイズがこれに満たない場合は、空きサイズができるまで本サービスコールは呼び出し元タスクをメッセージ送信待ち状態に移行させます。

snd_mbf,tsnd_mbf は、他 CPU のメッセージバッファに対して要求することができます。

(5) メッセージを送信する(psnd_mbf, ipsnd_mbf)

メッセージバッファにメッセージを送信します。snd_mbf, tsnd_mbf と異なるのは、メッセージバッファの空きサイズが足りない場合に、メッセージ送信待ち状態にはならずに直ちにエラーリターンする点です。

psnd_mbf は、他 CPU のメッセージバッファに対して要求することができます。

(6) メッセージを受信する(rcv_mbf, trcv_mbf)

メッセージバッファからメッセージを受信します。メッセージバッファにメッセージがない場合は、メッセージバッファにメッセージが送信されるまで、本サービスコールは呼び出し元タスクをメッセージ受信待ち状態に移行させます。

メッセージバッファからメッセージを受信すると、メッセージバッファの空きサイズは以下のサイズだけ増加します。

(受信メッセージサイズを 4 の倍数に切り上げたサイズ) + 4

この結果、メッセージ送信待ちタスクが送信しようとしていたメッセージのサイズよりもメッセージバッファの空きサイズが大きくなると、そのメッセージがメッセージバッファに送信され、そのタスクの待ち状態が解除されます。

rcv_mbf,trcv_mbf は、他 CPU のメッセージバッファに対して要求することができます。

(7) メッセージを受信する(prcv_mbf)

メッセージバッファからメッセージを受信します。rcv_mbf, trcv_mbf と異なるのは、メッセージバッファにメッセージがない場合に、メッセージ受信待ち状態にはならずに直ちにエラーリターンする点です。

prcv_mbf は、他 CPU のメッセージバッファに対して要求することができます。

(8) メッセージバッファの状態を参照する(ref_mbf, iref_mbf)

メッセージバッファに蓄積されているメッセージ数やメッセージバッファの空きサイズ、送信・受信待ちタスク ID を参照します。

ref_mbf は、他 CPU のメッセージバッファに対して要求することができます。

5.11　固定長メモリプール

固定長メモリプールは、決められたサイズのメモリブロックを動的に獲得・解放するオブジェクトです。

固定長メモリプールは、可変長メモリプールに比べて、任意サイズのメモリブロックを獲得できない短所がある反面、獲得・返却のオーバヘッドが小さいというメリットがあります。

固定長メモリプールを生成するときには、メモリブロックのサイズと数を指定します。図 5.16に固定長メモリプールの概要を示します。

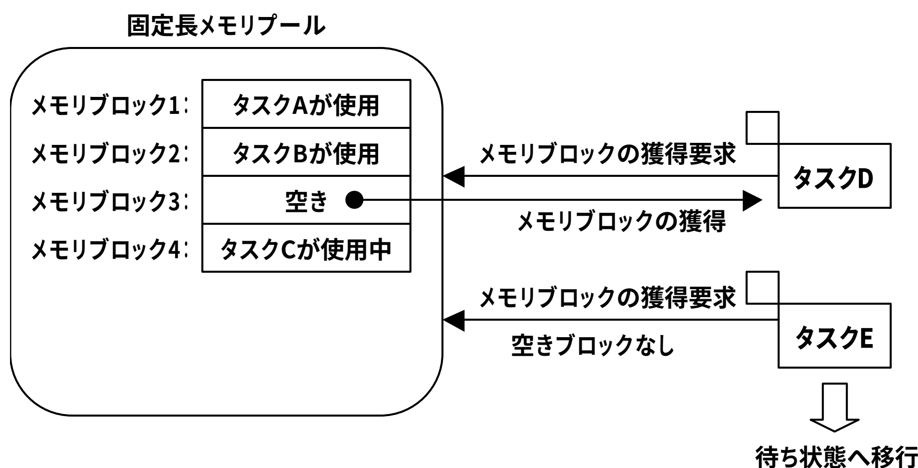


図5.16　固定長メモリプール

固定長メモリプール領域は、カーネルが持つデフォルト固定長メモリプール用領域から割り付ける方法と、アプリケーションが確保した領域を固定長メモリプール領域として使用する方法があり、固定長メモリプールを生成するときにどちらにするかを指定します。なお、デフォルト固定長メモリプール用領域のサイズは、cfg ファイルの memmpf.all_memsize に指定します。

また、固定長メモリプールの管理方式として、cfg ファイルの system.mpfmanage によって以下のいずれかを選択できます。

(1)　従来方式(system.mpfmanageにINを指定)
固定長メモリプール内のメモリブロックに隣接して、メモリブロックを管理するためのカーネル管理テーブルを配置する管理方式です。

(2)　拡張方式(system.mpfmanageにOUTを指定)
メモリブロックを管理するためのカーネル管理テーブルを固定長メモリプール領域とは別の場所に配置する方式です。この方式では、固定長メモリプール生成時に、その管理テーブル領域のアドレスを指定します。管理テーブル領域は、アプリケーションで確保する必要があります。この方式では、固定長メモリプールから獲得するメモリブロックのアドレスをアライメントしやすい、という特徴があります。

固定長メモリプールを操作するサービスコールには、次のものがあります。

(1) 固定長メモリプールを生成する(cre_mpf, icre_mpf)

指定された ID で、固定長メモリプールを生成します。

(2) 固定長メモリプールを生成する(acre_mpf, iacre_mpf)

任意の ID で、固定長メモリプールを生成します。ID は、カーネルが自動的に割り当て、リターン値として返します。

(3) 固定長メモリプールを削除する(del_mpf)

固定長メモリプールを削除します。

(4) メモリブロックを獲得する(get_mpf, tget_mpf)

固定長メモリプールからメモリブロックを獲得します。固定長メモリプールに空きメモリブロックがない場合は、メモリブロックが解放されるまで本サービスコールは呼び出し元タスクを待ち状態に移行させます。

get_mpf、tget_mpf は、他 CPU の固定長メモリプールに対して要求することができます。

(5) メモリブロックを獲得する(pget_mpf, ipget_mpf)

固定長メモリプールからメモリブロックを獲得します。get_mpf, tget_mpf と異なるのは、固定長メモリプールに空きメモリブロックがない場合に、待ち状態にはならずに直ちにエラーリターンする点です。

pget_mpf は、他 CPU の固定長メモリプールに対して要求することができます。

(6) メモリブロックを解放する(rel_mpf, irel_mpf)

メモリブロックを解放します。メモリブロックの獲得を待っているタスクがある場合は、そのタスクの待ち状態を解除します。

rel_mpf は、他 CPU の固定長メモリプールに対して要求することができます。

(7) 固定長メモリプールの状態を参照する(ref_mpf, iref_mpf)

固定長メモリプールの空きメモリブロック数や待ちタスク ID を参照します。

ref_mpf は、他 CPU の固定長メモリプールに対して要求することができます。

5.12 可変長メモリプール

可変長メモリプールは、任意のサイズのメモリブロックを動的に獲得・解放するオブジェクトです。
可変長メモリプールは、固定長メモリプールに比べて、任意サイズのメモリブロックを獲得できる長所がある反面、獲得・返却のオーバヘッドが大きいというデメリットがあります。また、後述する断片化問題にも注意する必要があります。

可変長メモリプールは、カーネルが持つデフォルト可変長メモリプール用領域から割り付ける方法と、アプリケーションが確保した領域を可変長メモリプール領域として使用する方法があり、可変長メモリプールを生成するときにどちらにするかを指定します。なお、デフォルト可変長メモリプール用領域のサイズは、cfg ファイルの memmpl.all_memsize に指定します。

可変長メモリプールを操作するサービスコールには、次のものがあります。

(1) 可変長メモリプールを生成する(cre_mpl, icre_mpl)

指定された ID で、可変長メモリプールを生成します。

(2) 可変長メモリプールを生成する(acre_mpl, iacre_mpl)

任意の ID で、可変長メモリプールを生成します。ID は、カーネルが自動的に割り当て、リターン値として返します。

(3) 可変長メモリプールを削除する(del_mpl)

可変長メモリプールを削除します。

(4) メモリブロックを獲得する(get_mpl, tget_mpl)

可変長メモリプールから指定したサイズのメモリブロックを獲得します。メモリブロックを獲得する際には、メモリブロックに加えて管理テーブルもメモリプール内に生成されます。このため、獲得後の空きサイズは、管理テーブルサイズも加えたサイズだけ減少します。
可変長メモリプールの空きサイズが不足している場合は、空きサイズができるまで本サービスコールは呼び出し元タスクを待ち状態に移行させます。
管理テーブルの詳細は「5.12.2 可変長メモリプールの管理方法」を参照してください。
get_mpl, tget_mpl は、他 CPU の可変長メモリプールに対して要求することができます。

(5) メモリブロックを獲得する(pget_mpl, ipget_mpl)

可変長メモリプールから指定したサイズのメモリブロックを獲得します。get_mpl tget_mpl と異なるのは、可変長メモリプールの空きサイズが足りない場合に、待ち状態にはならずに直ちにエラーリターンする点です。
pget_mpl は、他 CPU の可変長メモリプールに対して要求することができます。

(6) メモリブロックを解放する(rel_mpl, irel_mpl)

メモリブロックを解放します。

メモリブロックの解放により、可変長メモリプールの空きサイズは増加します。詳細は「5.12.2 可変長メモリプールの管理方法」を参照してください。

この結果、メモリブロックの獲得待ちタスクの要求メモリブロックサイズよりも可変長メモリプールの空きサイズが大きくなると、そのタスクはメモリブロックを獲得して待ち状態が解除されます。

rel_mpl は、他 CPU の可変長メモリプールに対して要求することができます。

(7) 可変長メモリプールの状態を参照する(ref_mpl, iref_mpl)

空きサイズの合計や最大連続空きメモリブロックサイズ、待ちタスク ID を参照します。

ref_mpl は、他 CPU の可変長メモリプールに対して要求することができます。

5.12.1 空き領域の断片化とその対策

可変長メモリプールからメモリブロックの獲得と解放(返却)を繰り返していると、空き領域の断片化が発生し、空き領域のトータルサイズは十分でも、連続した空き領域が存在しない、つまり大きなメモリブロックを獲得できない状況になることがあります(図 5.17)。

図5.17 空き領域の断片化

cfg ファイルで、system.newmpl に NEW を指定すると、この問題が軽減されるため、system.newmpl に NEW を指定することを強く推奨します。なお、system.newmpl に PAST を指定すると、カーネルは HI7000/4 シリーズ V1 と同じ方式で可変長メモリプールを管理します。

さらに、system.newmpl に NEW を指定した場合は、可変長メモリプールの属性に VTA_UNFRAGMENT を指定することができ、この場合より断片化を軽減できる場合があります。断片化の度合いは、一般には VTA_UNFRAGMENT を指定したほうが良くなる傾向がありますが、メモリプールの利用方法に依存します。

system.newmplにNEWを指定した場合、可変長メモリプール生成時に指定するT_CMPL構造体に、VTA_UNFRAGMENT属性用のパラメータ(最小ブロックサイズ、セクタ数、管理テーブルアドレス)が追加されます。

VTA_UNFRAGMENT属性を指定すると、可変長メモリプールがセクタ方式で管理されます。セクタ方式では、「最小ブロックサイズ」×8バイトまでのブロックを「微小なブロック」として扱います。また、獲得要求サイズは表5.1に示すように切り上げて扱います。
微小なブロックの獲得要求があると、カーネルはその切上げ後のサイズのブロックで構成されるセクタを生成します。セクタのサイズは常にminblksz×32です。つまり、要求サイズに応じて、セクタ内のブロック数が異なります。

表5.1　微小ブロックの扱い

獲得要求サイズ (blksz) *	切上げ後のサイズ *	セクタ内のブロック数
0＜blksz≦minblksz	minblksz	32
minblksz＜blksz≦minblksz×2	minblksz×2	16
minblksz×2＜blksz≦minblksz×4	minblksz×4	8
minblksz×4＜blksz≦minblksz×8	minblksz×8	4

【注】blksz：要求サイズ、minblksz：最小ブロックサイズ

そして、カーネルはセクタ内のメモリブロックを割り当てます。セクタ内の残りのブロックは、以降のそのブロックサイズ以下のメモリブロック獲得要求のために予約されることになります。
このように、微小なブロックを連続して使用されるようにすることで、比較的大きなサイズの連続空き領域が維持されやすくなっています.

図5.18に、「最小ブロックサイズ」が32の可変長メモリプールの例を示します。
最初に32バイトのメモリブロック獲得要求があると、32×32＝1024バイトのセクタ[A]を確保し、そのセクタ内の32バイトの領域[A-1]を割り付けます(図5.18 (1))。次に16バイトの獲得要求があると、Aの中の32バイトの領域(A-2)を割り付けます(図5.18 (2))。
さらに、今度は36バイトのメモリブロック獲得要求があったとします。この場合、セクタAの各ブロックは32バイトなので、この要求には対応できません。そこで、要求サイズの36を最小ブロックサイズで切上げた64バイトのブロックが16個で構成される新たなセクタB(64×16=1024バイト)を確保し、そのセクタ内の64バイトの領域[B-1]を割り付けます(図5.18 (3))。

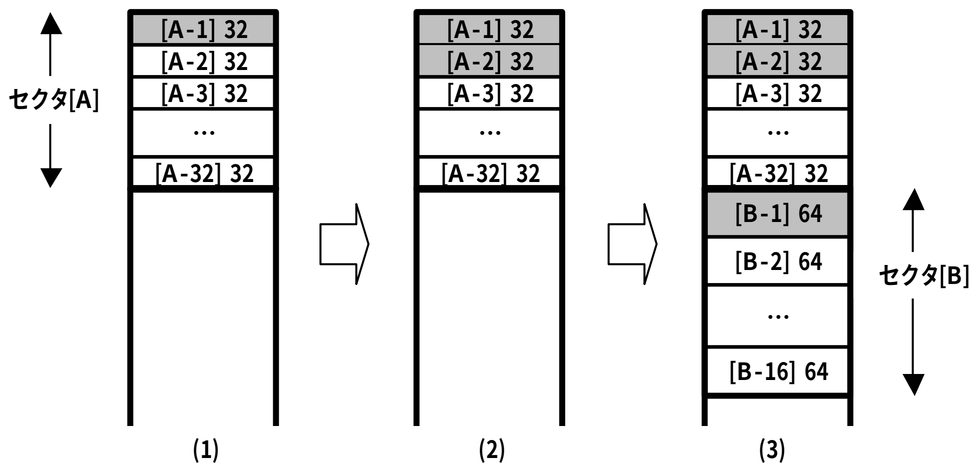


図5.18　可変長メモリプールの例

最大セクタ数を使いきってしまっている場合、または新たなセクタを生成するだけの連続空き領域がない場合には、セクタを生成せずに要求サイズのメモリブロックを獲得します。つまり、この場合には、断片化が発生しやすくなります。さらに要求サイズの連続空き領域も無い場合は、より大きなブロックサイズを持つセクタから獲得します。

セクタ内の全てのブロックが解放されると、セクタそのものも解放されます。

また、微小でないサイズの要求(minblksz×8 を超える要求サイズ)では、常にセクタを生成せずに要求サイズのメモリブロックを獲得します。

5.12.2　可変長メモリプールの管理方法

カーネルは、割り付けたメモリブロックを管理するために、可変長メモリプール内に管理テーブルを生成します。つまり、可変長メモリプール領域は、アプリケーションが獲得するメモリブロックの領域以外にカーネルの管理テーブルのためにも使用されます。このことを考慮して、可変長メモリプールサイズを決定する必要があります。

(1) system.newmpl に PAST を指定した場合

メモリブロックの獲得時に 16 バイトの管理テーブルを生成します。この管理テーブルは、メモリブロック返却時に解放されます。

(2) system.newmpl に NEW を指定した場合

VTA_UNFRAGMENT 属性の指定がある場合は、メモリブロック獲得時にセクタが生成されると、32 バイトの管理テーブルを生成します。この管理テーブルは、セクタ解放時に解放されます。

また、VTA_UNFRAGMENT 属性の指定がある場合でセクタ外のメモリブロックの獲得時、および VTA_UNFRAGMENT 属性の指定がない場合のメモリブロックの獲得時にも、32 バイトの管理テーブルを生成します。この管理テーブルは、メモリブロック返却時に解放されます。

5.13　時間管理機能

カーネルは、時間に関する以下のような機能をサポートしています。

- システム時刻の参照・設定
- タイムイベントハンドラ（周期ハンドラ、アラームハンドラ、オーバーランハンドラ）の実行制御
- タイムアウトなどの時間によるタスクの実行制御

カーネルは、システム時刻と呼ぶカウンタによってこれらの機能を実現しています。サービスコールで使用する時間パラメータの単位時間は 1ms です。一方、タイムティックを供給する周期は、cfg ファイルで指定する system.tic_nume, system.tic_deno によって決まります。

時間管理機能を使用するには、cfg ファイルで clock.timer に TIMER を指定し、さらにタイマドライバを作成する必要があります。タイマドライバの作成方法は「12.9 タイマドライバ」を参照してください。なお、本製品ではサンプルタイマドライバを提供しています。

5.13.1　タスクのタイムアウト

tslp_tsk や twai_sem のように、't'で始まるサービスコールではタイムアウトを指定できます。

指定したタイムアウト時間が経過しても待ち条件が満たされない場合、待ち状態が解除されます。そして、サービスコールのリターン値としてエラーE_TMOUT が返されます。

タイムアウトは、本来発生するはずのイベントが発生しなかったときの異常検出に利用できます。

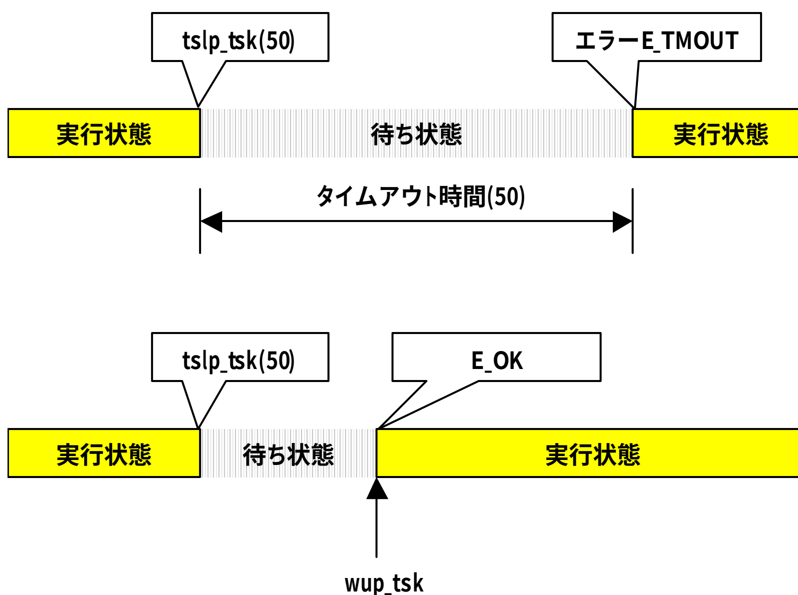


図5.19　タイムアウト

5.13.2 タスクの遅延

dly_tsk を用いて、タスクを指定した時間だけ待ち状態に移行させることができます。指定した時間が経過すると、待ち状態が解除され、リターン値として E_OK が返されます。

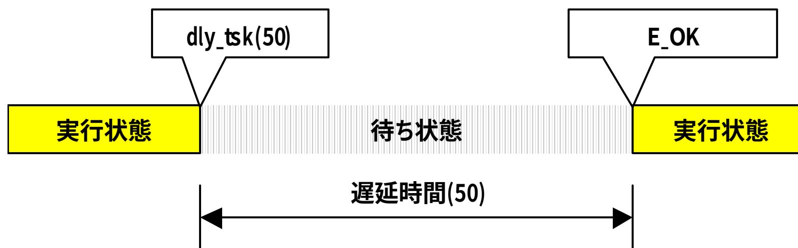


図5.20 タスクの遅延

5.13.3 タイマの停止と再開

vstp_tmr によってカーネルが使用するハードウェアタイマを停止し、vrst_tmr, ivrst_tmr によってタイマを再開することができます。

本機能は、CPU をスリープさせたい場合に、タイマ割込みも停止させたい時に利用できます。

vstp_tmr では、上限時間を指定します。上限時間未満のタイマイベント（タスクのタイムアウトやタイムイベントハンドラなど）がない場合のみ、タイマを停止します。

タイマを再開するには、vrst_tmr, ivrst_tmr を使います。タイマ停止中の時間を別の方法で計測できる場合には、その経過時間を vrst_tmr, ivrst_tmr に指定することで、カーネルが管理する時間を補正することができます。その結果、タイマイベントが発生することがあります。タイマ停止中の時間を計測できない場合は、経過時間として 0 を指定します。

vsns_tmr を使えば、タイマが動作中かどうかを確認することができます。

なお、これらの機能はμITRON4.0 仕様外の機能です。

5.13.4 周期ハンドラ

周期ハンドラは、指定した起動位相経過後、起動周期ごとに起動されるタイムイベントハンドラです。周期ハンドラの起動には、起動位相を保存する方法と起動位相を保存しない方法があります。起動位相を保存する場合は、周期ハンドラの生成時点を基準に周期ハンドラを起動します。起動位相を保存しない場合は、周期ハンドラの動作開始時点を基準に周期ハンドラを起動します。

図 5.21に、周期ハンドラの動作例を示します。

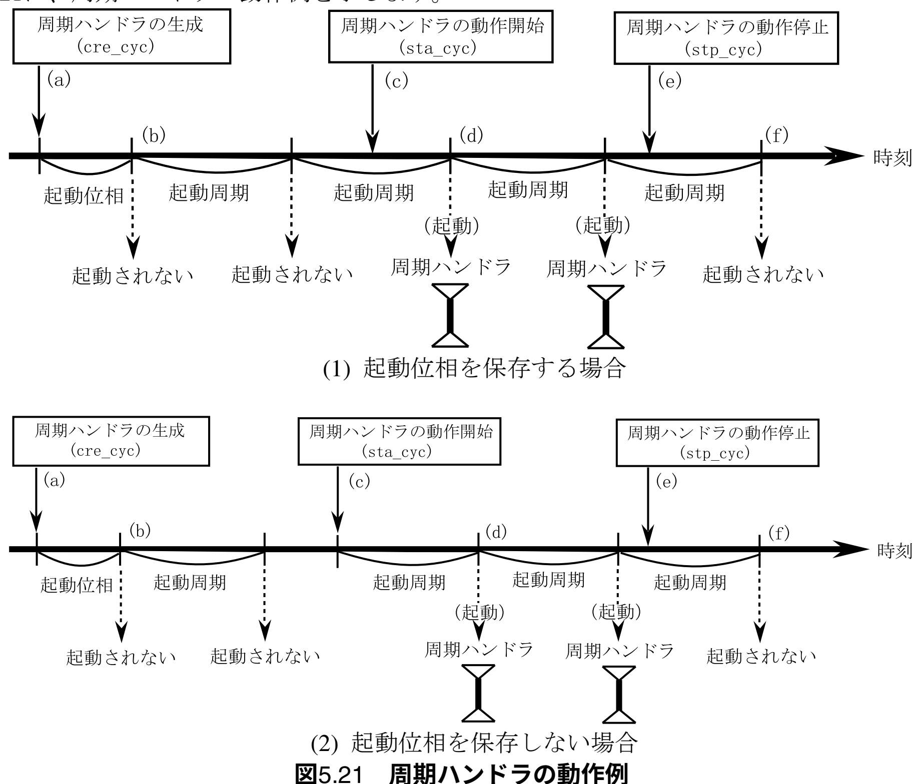


図5.21 周期ハンドラの動作例

周期ハンドラには、生成時に指定された拡張情報が渡されます。

周期ハンドラを操作するサービスコールには、次のものがあります。

(1) 周期ハンドラを生成する(cre_cyc, icre_cyc)

指定された ID で、周期ハンドラを生成します。

(2) 周期ハンドラを生成する(acre_cyc, iacre_cyc)

任意の ID で、周期ハンドラを生成します。ID は、カーネルが自動的に割り当て、リターン値として返します。

(3) 周期ハンドラを削除する(del_cyc)

周期ハンドラを削除します。

RENESAS

(4) 周期ハンドラの動作を開始する(sta_cyc, ista_cyc)

周期ハンドラの動作を開始します。
sta_cyc は、他 CPU の周期ハンドラに対して要求することができます。

(5) 周期ハンドラの動作を停止する(stp_cyc, istp_cyc)

周期ハンドラの動作を停止します。
stp_cyc は、他 CPU の周期ハンドラに対して要求することができます。

(6) 周期ハンドラの状態を参照する(ref_cyc, iref_cyc)

周期ハンドラの動作状態や、次の起動までの残り時間を参照します。
ref_cyc は、他 CPU の周期ハンドラに対して要求することができます。

5.13.5 アラームハンドラ

アラームハンドラは、指定した時刻になると 1 度だけ起動されるタイムイベントハンドラです。アラームハンドラを用いることにより、時刻に依存した処理を行うことができます。

図 5.22にアラームハンドラの動作例を示します。

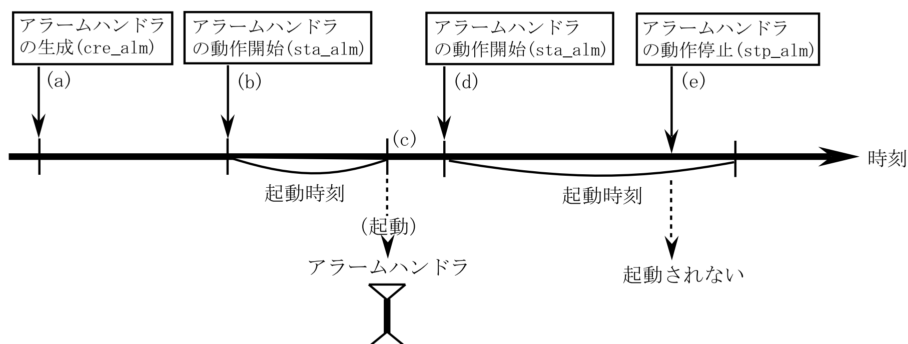


図5.22 アラームハンドラの動作例

アラームハンドラには、生成時に指定された拡張情報が渡されます。

アラームハンドラを操作するサービスコールには、次のものがあります。

(1) アラームハンドラを生成する(cre_alm, icre_alm)

指定された ID で、アラームハンドラを生成します。

(2) アラームハンドラを生成する(acre_alm, iacre_alm)

任意の ID で、アラームハンドラを生成します。ID は、カーネルが自動的に割り当て、リターン値として返します。

(3) アラームハンドラを削除する(del_alm)

アラームハンドラを削除します。

(4) アラームハンドラの動作を開始する(sta_alm, ista_alm)

指定された時間後に起動するように、アラームハンドラの動作を開始します。

sta_alm は、他 CPU のアラームハンドラに対して要求することができます。

(5) アラームハンドラの動作を停止する(stp_alm, istp_alm)

アラームハンドラの動作を停止します。

stp_alm は、他 CPU のアラームハンドラに対して要求することができます。

(6) アラームハンドラの状態を参照する(ref_alm, iref_alm)

アラームハンドラの動作状態や、起動までの残り時間を参照します。

ref_alm は、他 CPU のアラームハンドラに対して要求することができます。

RENESAS

5.13.6 オーバーランハンドラ

オーバーランハンドラは、各タスクに設定された時間を超えてプロセッサを使用した場合に起動されるタイムイベントハンドラで、ひとつだけ定義できます。

図 5.23にオーバーランハンドラの動作例を示します。

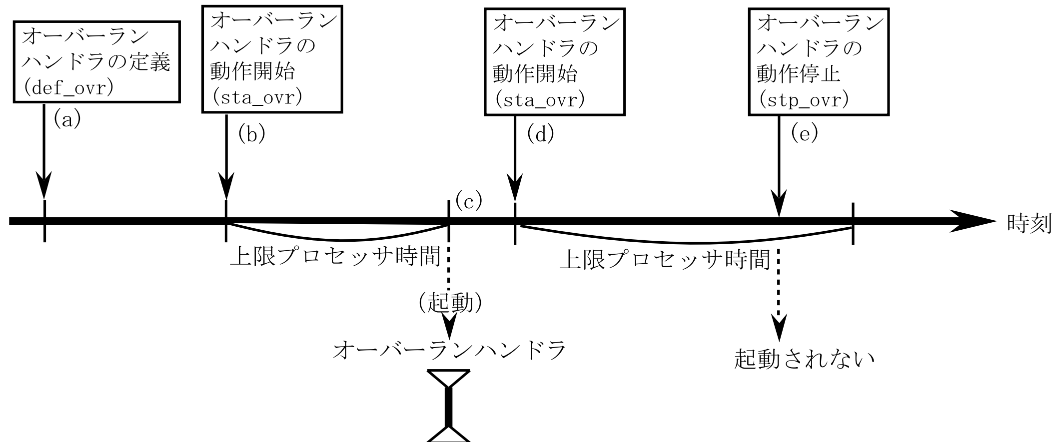


図5.23 オーバーランハンドラの動作例

オーバーランハンドラには、オーバーランしたタスク ID と、そのタスクの拡張情報が渡されます。
オーバーランハンドラを操作するサービスコールには、次のものがあります。

(1) オーバーランハンドラを定義する(def_ovr)

オーバーランハンドラを定義します。

(2) タスクのオーバーラン監視を開始する(sta_ovr, ista_ovr)

指定されたタスクについて、指定された上限時間を設定します。そのタスクが指定された上限時間を越えて実行すると、オーバーランハンドラが起動されます。
sta_ovr は、他 CPU のタスクに対して要求することができます。

(3) タスクのオーバーラン監視を停止する(stp_ovr, istp_ovr)

指定されたタスクのオーバーラン監視を停止します。
stp_ovr は、他 CPU のタスクに対して要求することができます。

(4) タスクのオーバーラン監視状態を参照する(ref_ovr, iref_ovr)

指定されたタスクのオーバーラン監視状態や、上限時間までの残り時間を参照します。
ref_ovr は、他 CPU のタスクに対して要求することができます。

5.13.7 時間の精度

タイムアウトなどの時間パラメータの単位はすべてミリ秒です。

その精度は TIC_NUME／TIC_DENO[ms]となります。この精度で、システム時刻の更新や時間管理が行われます。TIC_NUME および TIC_DENO は、それぞれ cfg ファイルの system.tic_nume および system.tic_deno に定義します。

時間イベント(タイムアウト発生や周期ハンドラ起動など)は、指定した時間以上が経過してから発生するようになっています。

図 5.24に、実時刻が 9.2[ms]の時点で tslp_tsk(5)を実行した場合の例を示します。

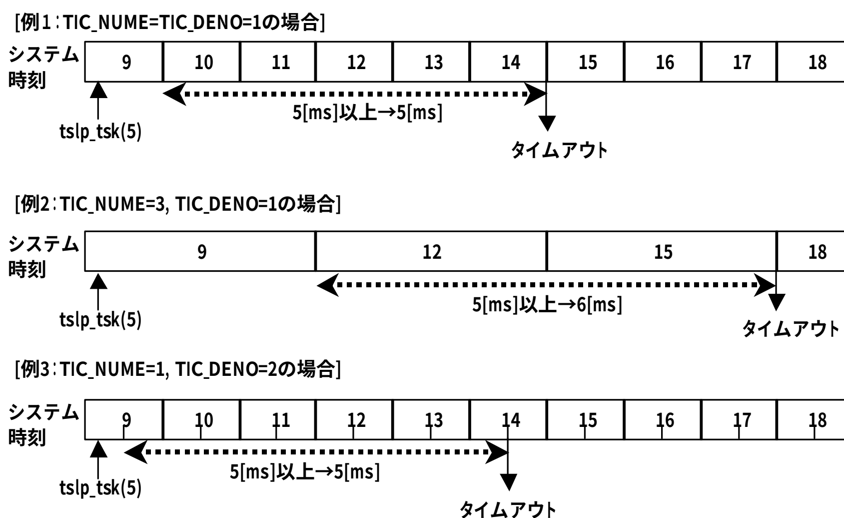


図5.24 時間の精度(tslp_tsk)

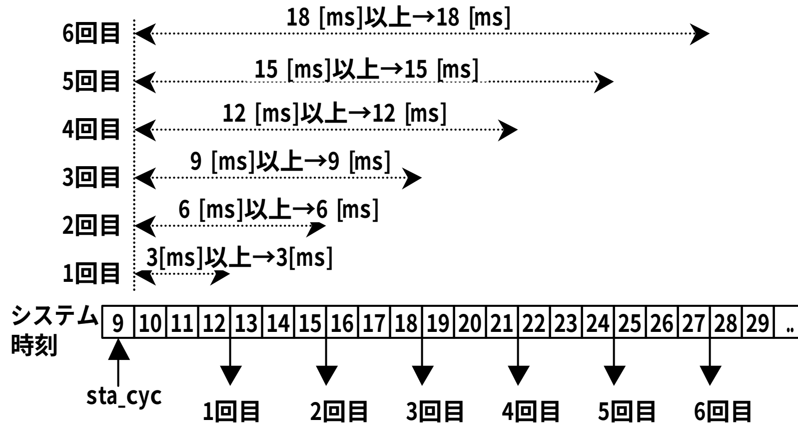
周期ハンドラの場合は、各起動タイミング毎に以下のように扱います。

(1) TA_PHS属性が指定されていない周期ハンドラ
  (a) sta_cyc, ista_cycで動作開始した場合
    sta_cyc, ista_cyc時点を基準として、n回目のハンドラ起動タイミングは次式の値として扱います。
      (起動周期) × n
  (b) 生成時にTA_STA属性を指定して動作開始した場合
    生成時点を基準として、n回目のハンドラ起動タイミングは次式の値として扱います。
      (起動位相) + (起動周期) × (n – 1)

(2) TA_PHS属性が指定された周期ハンドラ
  (1)の(b)と同じ扱いとなります。ただし、実際にハンドラが起動されるかどうかは、ハンドラの動作状態で決まります。

RENESAS

図 5.25に、実時刻が 9.5[ms]の時点 sta_cyc で起動周期が 3 の周期ハンドラの動作を開始した場合の例を示します。例 2 のように、起動周期が TIC_NUME/TIC_DENO 未満の場合は、同じタイミングで 2 回以上周期ハンドラが起動されるケースがあることに注意してください。



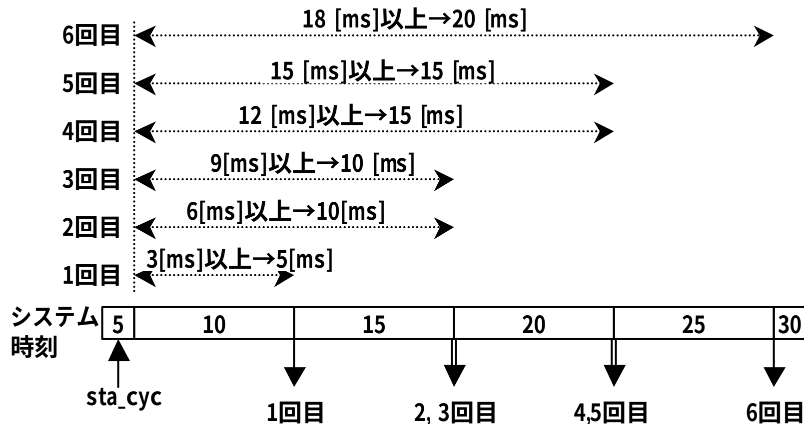


図5.25　時間の精度(sta_cyc)

5.13.8 注意事項

タイマ割込み発生時には、カーネルは以下の処理を行います。

(a) システム時刻の更新
(b) アラームハンドラの起動と実行
(c) 周期ハンドラの起動と実行
(d) オーバーランハンドラの起動と実行
(e) タイムアウト付きサービスコールおよびdly_tskによるタスクのタイムアウト処理

これらの処理は全て、タイマ割込みレベル以下の割込みをマスクした状態で行われます。

上記のうち、(b),(c),(e)は、複数のタスクやハンドラに対する処理が重複する可能性があるため、このような場合カーネルの処理時間が極端に長くなります。これは、以下のような弊害をもたらします。

- 割込みに対するレスポンスの悪化
- システム時刻の遅れ

これを避けるために、以下を遵守してください。

- タイムイベントハンドラの処理は、可能な限り短くしてください。
- タイムイベントハンドラの周期、タイムアウト付きサービスコールで指定するタイムアウト値は、なるべく大きな値にしてください。極端な例としては、ある周期ハンドラの周期時間が 1ms で、そのハンドラ処理時間が 1ms 以上かかるような場合、永久にその周期ハンドラだけが実行されることになり、事実上ハングアップします。

5.14　システム状態管理機能

5.14.1　システム状態管理

(1) タスクの実行待ち行列を回転する(rot_rdq, irot_rdq)

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

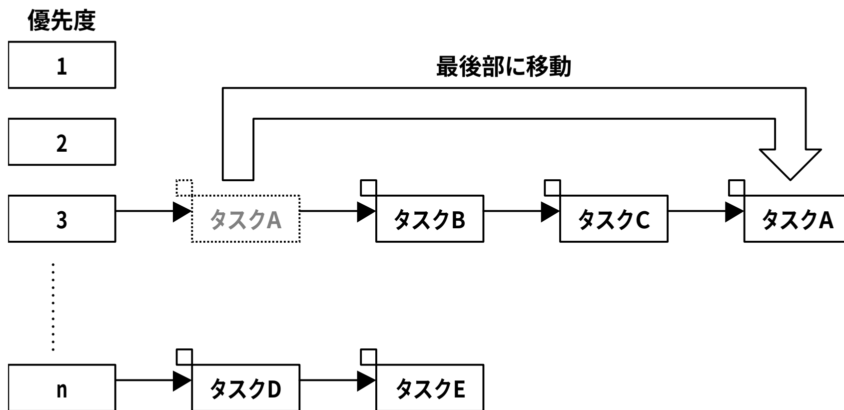


図5.26　rot_rdq によるレディキューの操作

(2) 実行状態のタスク ID を得る(get_tid, iget_tid)

get_tid は、自タスクのタスク ID を取得します。非タスクコンテキストから iget_tid を呼び出した場合は、その時点で実行していたタスク ID を取得します。

(3) CPU ロック状態への移行(loc_cpu, iloc_cpu)とその解除(unl_cpu, iunl_cpu)

loc_cpu, iloc_cpu により CPU ロック状態に移行することができます。CPU ロック状態を解除するには、unl_cpu, iunl_cpu を使います。

(4) ディスパッチ禁止状態への移行(dis_dsp)とその解除(ena_dsp)

dis_dsp によりディスパッチ禁止状態に移行することができます。ディスパッチ禁止状態を解除するには、ena_dsp を使います。

(5) コンテキスト種別を確認する(sns_ctx)

現在実行中のコンテキストが、タスクコンテキストか非タスクコンテキストかを確認します。

(6) CPU ロック状態かどうかを調べる(sns_loc)

現在が CPU ロック状態かどうかを調べます。

(7) ディスパッチ禁止状態かどうかを調べる(sns_dsp)

現在がディスパッチ禁止状態かどうかを調べます。

(8) ディスパッチ保留状態かどうかを調べる(sns_dpn)

現在がディスパッチ保留状態かどうかを調べます。

ディスパッチ保留状態とは、ディスパッチャより優先順位の高い処理を実行中であることを意味し、他のタスクは実行されません。具体的には、以下のいずれかのケースに該当する場合がディスパッチ保留状態です。

- CPU ロック状態
- ディスパッチ禁止状態
- 非タスクコンテキスト
- ノーマル CPU 例外ハンドラを実行中
- SR レジスタの IMASK が 0 でない

ディスパッチ保留状態でないときには、待ち状態に移行するサービスコールを利用可能です。ソフトウェア部品など、どういった状態で呼び出されるか分からないソフトウェアでは、本サービスコールを用いて待ち状態に移行するサービスコールを呼び出すかエラーとするか、といった判定を行うことができます。

5.14.2 初期化関連のサービスコール

(1) カーネルを起動する(vsta_knl, ivsta_knl)

コンフィギュレーション結果に従って、カーネルを初期化します。

(2) リモートサービスコール環境を初期化する(vini_rmt)

コンフィギュレーション結果に従って、リモートサービスコール環境を初期化します。

5.14.3 システムダウン(vsys_dwn, ivsys_dwn)

vsys_dwn は、システムダウンさせてシステムダウンルーチンを起動します。

5.14.4 サービスコールトレース機能

サービスコールトレース機能とは、サービスコールの呼び出し履歴を保存する機能です。トレース結果は、デバッギングエクステンションを用いて参照することができます。

トレース機能を組み込むかどうかは、cfg ファイルの system.trace で定義します。

トレース機能の詳細は、デバッギングエクステンションのヘルプを参照してください。

(1) 取得タイミングと取得される情報

トレースは、以下のようなタイミングで取得されます。

- サービスコール発行とリターン
- タスク、タスク例外処理ルーチンの実行開始と実行終了
- カーネルアイドリングへの遷移

取得される情報には、以下のようなものがあります。

- サービスコールの種類
- サービスコールのパラメータ
- サービスコールのエラーコード
- PC(プログラムカウンタ)
- トレースシリアル番号

トレースシリアル番号とは、全 CPU 共通のトレースのシリアル番号です。

(2) トレースのタイプ

履歴の保存場所として、system.trace でターゲット上の RAM に確保したバッファと、シミュレータやエミュレータのトレースメモリの、いずれかを選択できます。前者を「ターゲットトレース」、後者を「ツールトレース」と呼びます。前者の場合は、systerm.trace_buffer にバッファのサイズを指定します。

ツールトレースは、エミュレータやシミュレータなど、利用できる環境は限定されますが、ターゲット上にサービスコールトレースのための領域をほとんど必要としないという特徴があります。

(3) オブジェクト取得数

デバッギングエクステンションでは、ユーザが指定したオブジェクトの状態を、トレースを取得するタイミング毎に取得するように指定することができます。同時に取得可能なオブジェクト数は、system.trace_object で定義します。

(4) ユーザイベントの取得(vget_trc, ivget_trc)

vget_trc, ivget_trc を使用すれば、ユーザ任意のタイミングで任意の情報を取得することができます。

(5) 割込みハンドラ･CPU 例外ハンドラの開始･終了の取得(ivbgn_int, ivend_int)

割込みハンドラと CPU 例外ハンドラの開始と終了は、デフォルトではトレース取得されません。

必要に応じて、ハンドラの実行開始時に ivbgn_int、ハンドラの終了時に ivend_int を呼び出すことで、デバッギングエクステンションでそのハンドラの実行履歴が表示されるようになります。

ivbgn_int, ivend_int では、ハンドラを区別する情報として、ベクタ番号を指定します。

(6) 注意事項

(a) パフォーマンスの低下

サービスコールトレース機能を使用すると、カーネルのパフォーマンスが低下します。

(b) 取得されないサービスコール等

非タスクコンテキスト用の ixxx_yyy サービスコールとタスクコンテキスト用の xxx_yyy サービスコールが存在するものは、両方とも xxx_yyy サービスコールとして取得されます。

また、以下のサービスコールは取得されません。

cal_svc, ical_svc, vsta_knl, ivsta_knl, vsys_dwn, ivsys_dwn

(c) トレースシリアル番号

system.trace!=NO の場合、トレースシリアル番号の一貫性が崩れる場合があります。これを避けるには、CPUID#1 側の初期化ルーチンまたは最初のタスクの実行が開始した後に、CPUID#2 で vsta_knl を呼び出すようにしてください。

RENESAS

5.15 割込み管理機能

割込みが発生すると、割込みハンドラが起動されます。割込みハンドラは、cfg ファイルの interrupt_vector[]、または def_inh, idef_inh サービスコールで定義します。
また、「4.8 割込み」も参照してください。

(1) 割込みハンドラを定義する(def_inh, idef_inh)

指定されたベクタ番号に対する割込みハンドラを定義します。

(2) 割込みマスクを変更する(chg_ims, ichg_ims)

割込みマスク(SR レジスタの IMASK ビット)を指定された値に変更します。

(3) 割込みマスクを参照する(get_ims, iget_ims)

現在の割込みマスク(SR レジスタの IMASK ビット)を参照します。

5.16 拡張サービスコール

サービスコール処理ルーチンを作成し、拡張サービスコールルーチンとしてカーネルに登録することができます。アプリケーションは、拡張サービスコールルーチンとリンクすることなく、そのルーチンを呼び出すことができます。
拡張サービスコールは、正の機能コードによって識別されます。使用できる機能コードの最大値は、cfg ファイルの maxdefine.max_fncd に定義します。

拡張サービスコールを操作するサービスコールには、次のものがあります。

(1) 拡張サービスコールルーチンを定義する(def_svc, idef_svc)

指定された機能コードに対する拡張サービスコールルーチンを定義します。

(2) 拡張サービスコールを呼び出す(cal_svc, ical_svc)

指定された機能コードの拡張サービスコールを呼び出します。これにより、対応する拡張サービスコールルーチンが呼び出されます。
cal_svc, ical_svc では、拡張サービスコールルーチンに渡すパラメータとして、32bit 整数型データを最大 4 つまで指定することができます。

5.17　システム構成管理機能

(1) CPU 例外ハンドラを定義する(def_exc, idef_exc)

指定されたベクタ番号に対する CPU 例外ハンドラを定義します。

(2) CPU 例外(TRAPA 命令例外)ハンドラを定義する(vdef_trp, idef_trp)

指定されたトラップ番号に対する CPU 例外ハンドラを定義します。

(3) コンフィギュレーション情報を参照する(ref_cfg, iref_cfg)

各オブジェクトの最大ローカルオブジェクト ID などを参照します。

(4) バージョン情報を参照する(ref_ver, iref_ver)

本カーネルのμITRON 仕様バージョンや、カーネルのバージョンを参照します。なお、これらのサービスコールで取得される情報と同じ情報を、カーネル構成マクロ(「6.31.2 カーネル構成マクロ」参照)から取得することもできます。

5.18　プロファイル管理機能

プロファイル管理機能は、一定周期でその時点で実行していたタスクをサンプリングすることで、確率的な各タスクの実行割合を知るための機能です。本機能は、μITRON4.0 仕様外の機能です。

プロファイル機能では、「全体」、「各タスク」、および「カーネルアイドリング状態」について、それぞれ 32 ビットの「プロファイルカウンタ」を持っています。

TIC_NUME/TIC_DENO ミリ秒の周期で実行されるカーネルのタイマ割込み処理では、その時点で実行中のタスク、またはカーネルアイドリングのプロファイルカウンタと、全体のプロファイルカウンタを加算(+1)します(図 5.27)。

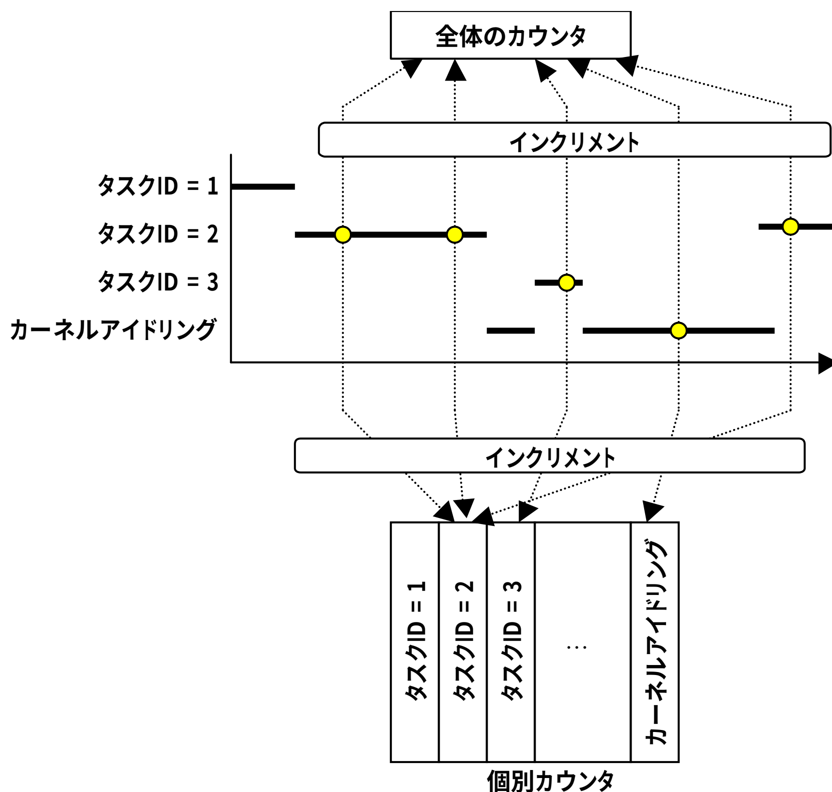


図5.27　プロファイル機能

正確ではありませんが、長時間の計測ではおおむね以下の式で各タスクの実行時間を求めることができます。

実行時間[ミリ秒]　＝　各タスクのプロファイルカウンタ×(TIC_NUME/TIC_DENO)

また、各タスクのプロファイルカウンタやカーネルアイドリングのプロファイルカウンタ値を全体のプロファイルカウンタで除することで、各タスクやカーネルアイドリングの CPU 占有率を算出することができます。

全プロファイルカウンタは、vsta_knl で 0 に初期化されます。また、タスクのプロファイルカウンタは、タスク削除時にも 0 に初期化されます。

RENESAS

なお、カーネルはプロファイルカウンタのオーバフロー検出を行わないので注意してください。例えば、TIC_NUME/TIC_DENO=1 ミリ秒の場合は 50 日弱、TIC_NUME/TIC_DENO=10 マイクロ秒の場合は 12 時間強でカウンタがオーバフローする可能性があります。

プロファイル管理機能のサービスコールには、次のものがあります。

(1) プロファイルカウンタを参照する(vref_prf, ivref_prf)

指定されたプロファイルカウンタを参照します。
vref_prf は、他 CPU のタスクに対して要求することができます。

(2) プロファイルカウンタをクリアする(vclr_prf, ivclr_prf)

指定されたプロファイルカウンタをクリアします。
vclr_prf は、他 CPU のタスクに対して要求することができます。

5.19 カーネルのアイドリング

実行可能状態のタスクが存在しなくなると、カーネル内部で無限ループとなり、割込みが発生するのを待ちます。
CPU の省電力モードを利用するには、通常は最低優先度のタスクで省電力モードに移行するようにします。

RENESAS

6. カーネルサービスコール

6.1 呼び出し形式

サービスコールは、以下のように C 言語の関数呼び出しの形式で記述します。

```
ercd = slp_tsk();
```

6.2 ヘッダファイル

サービスコールを呼び出すプログラムでは、< RTOS_INST>¥os¥include¥にある kernel.h をインクルードしてください。

kernel.h は、表 6.1に示すファイルをインクルードしています。

表6.1 kernel.h からインクルードされるファイル

ディレクトリ	ファイル名	説明
<RTOS_INST>¥os¥include¥	itron.h	ITRON 仕様共通規定の定義
	kernel_api.h	カーネルのサービスコール定義
ユーザ指定	kernel_intspec.h	割込みに関するハードウェア仕様の定義 詳細は「17.3 CPU 割込み仕様定義ファイル(kernel_intspec.h)の作成」を参照してください。
cfg72mp が出力	kernel_macro.h	コンフィギュレーションによって定まるカーネル構成定数などの定義
	mycpuid.h	MYCPUID(自 CPUID)の定義

6.3 基本データ型

基本データ型の詳細を、表 6.2に示します。これらは、types.h を元に定義されます。
types.h については、「19. types.h」を参照してください。

表6.2 基本データ型

No.	データ型	定義内容	No.	データ型	定義内容
1	B	符号付き 8 ビット整数	22	STAT	符号無し 32 ビット整数
2	H	符号付き 16 ビット整数	23	MODE	符号無し 32 ビット整数
3	W	符号付き 32 ビット整数	24	PRI	符号付き 16 ビット整数
4	D	符号付き 64 ビット整数	25	SIZE	符号無し 32 ビット整数
5	UB	符号無し 8 ビット整数	26	TMO	符号付き 32 ビット整数
6	UH	符号無し 16 ビット整数	27	RELTIM	符号無し 32 ビット整数
7	UW	符号無し 32 ビット整数	28	SYSTIM	以下のメンバから構成される構造体 上位：符号無し 16 ビット整数 下位：符号無し 32 ビット整数
8	UD	符号無し 64 ビット整数			
9	VB	符号付き 8 ビット整数 *			
10	VH	符号付き 16 ビット整数 *	29	VP_INT	符号付き 32 ビット整数 *
11	VW	符号付き 32 ビット整数 *	30	ER_BOOL	符号付き 32 ビット整数
12	VD	符号付き 64 ビット整数 *	31	ER_ID	符号付き 32 ビット整数
13	VP	void 型関数へのポインタ	32	ER_UINT	符号付き 32 ビット整数
14	FP	void 型関数へのポインタ	33	TEXPTN	符号無し 32 ビット整数
15	INT	符号付き整数(signed int)	34	FLGPTN	符号無し 32 ビット整数
16	UINT	符号無し整数(unsigned int)	35	RDVPTN	符号無し 32 ビット整数
17	BOOL	符号付き整数(signed int)	36	RDVNO	符号無し 32 ビット整数
18	FN	符号付き 32 ビット整数	37	OVRTIM	符号無し 32 ビット整数
19	ER	符号付き 32 ビット整数	38	INHNO	符号無し 32 ビット整数
20	ID	符号付き 16 ビット整数	39	EXCNO	符号無し 32 ビット整数
21	ATR	符号無し 32 ビット整数	40	IMASK	符号無し 32 ビット整数

【注】* これらのデータタイプの変数の値を参照する場合や代入する場合には、明示的に型変換(キャスト)を行う必要があります。

6.4 サービスコール呼び出し前後のレジスタ保証規則

サービスコール呼び出し前後において、レジスタの値が同一であることを保証するレジスタと保証しないレジスタがあります。この規則は、基本的には当社製 C コンパイラに準じていますが、その詳細を表 6.3に示します。

表6.3 サービスコール呼び出し前後のレジスタ保証

項番	レジスタ	レジスタ保証
1	SR, R8〜R15, PR, GBR, MACH, MACL	保証されます。ただし、chg_ims, ichg_ims, loc_cpu, iloc_cpu, unl_cpu, iunl_cpu の場合は、SR.IMASK は更新されます。
2	R0	正常終了（E_OK）、またはエラーコードが設定されます。
3	R1〜R7	リターンパラメータとして明示している場合以外は保証されません。
4	TBR	system.tbr を FOR_SVC(サービスコール呼び出し専用に使用)または TASK_CONTEXT(タスクコンテキストと扱う)と設定していた場合は保証されます。
5	[SH2A-FPU] FR0〜FR11	保証されません。[310]
6	[SH2A-FPU] FPSCR, FPUL, FR12〜FR15	以下に示す状態から発行した場合のみ保証されます。 ・TA_COP1 属性のタスクまたはタスク例外処理ルーチン ・ディスパッチ保留状態

6.5 サービスコールのリターン値とエラーコード

6.5.1 概要

リターン値を持つサービスコールでは、正の値または0(E_OK)が正常終了、負の値がエラーコードを意味します。正常終了時のリターン値の意味はサービスコール毎に異なりますが、多くのサービスコールの正常終了時はE_OKのみが返ります。

ただし、BOOL型のリターン値を持つサービスコールはこの限りではありません。

6.5.2 パラメータチェック機能

本カーネルでは、単純なパラメータエラーの検出を省略することができます。例えば、デバッグ完了後にパラメータチェック機能を取り外せば、オーバヘッドとコードサイズを削減することができます。

パラメータチェック機能を取り外すには、system.parameter_checkにNOを定義してください。

6.5.3 スタックオーバフローの検出

スタックオーバフローは、その発生箇所の特定が困難な不具合です。本機能は、これを補助するデバッグ用の機能です。

本機能では、タスクコンテキストから呼び出されたサービスコールについてのみ、その時点のスタックがオーバフローしているかどうかを検査します。オーバフローしていた場合は、システムダウンとなります。

ただし、この検査が行われないサービスコールもあります。どのサービスコールでこの検査が行われるかは、カーネルのバージョンに依存するため、公開していません。

また、あくまでもサービスコール発行時点のスタックがオーバフローしているかを検査する機能であることに注意してください。サービスコール呼び出し時よりもスタックのネストが深くなるケースがアプリケーション側にある場合など、アプリケーション側でのスタックオーバフローは検出されません。

なお、本機能は常に有効です(cfgファイルに、本機能に関する設定項目はありません)。

6.5.4 メインエラーコードとサブエラーコード

エラーコードは、下位8ビットのメインエラーコードとそれを除いた上位ビットのサブエラーコードから構成されています。本カーネルが返す全てのエラーコードのサブエラーコードは-1です。

なお、標準ヘッダitron.hには、以下のマクロが定義されています。

- ER MERCD (ER ercd);　エラーコードからメインエラーコードを取り出す
- ER SERCD (ER ercd);　エラーコードからサブエラーコードを取り出す
- ER ERCD (ER mercd, ER sercd);　メインエラーコードとサブエラーコードからエラーコードを生成する。

6.6 システム状態とサービスコール

サービスコールを呼び出せるかどうかは、システムの状態に依存します。

6.6.1 タスクコンテキストと非タスクコンテキスト

(1) 特殊なサービスコール

以下は、タスクコンテキストと非タスクコンテキストのどちらからも呼び出せます。

◆ vsta_knl, ivsta_knl

◆ vsys_dwn, ivsys_dwn

(2) sns, vsns で始まるサービスコール

sns, vsns で始まるサービスコールは、タスクコンテキストと非タスクコンテキストのどちらからも呼び出せます。

(3) (1), (2)以外のサービスコール

i で始まるサービスコールは非タスクコンテキスト専用、その他はタスクコンテキスト専用です。
これは、さらに以下の 2 種類に分類されます。

(a) del_tsk など、i を付加した"idel_tsk"が存在しないサービスコール

非タスクコンテキストから呼び出した場合は、E_CTX エラーが返ります。

(b) act_tsk と iact_tsk など、同機能で i で始まるものとそうでないものがあるサービスコール

i のあるものと無いもので、カーネルのサービスコール処理は共通です。このため、i で始まるサービスコールをタスクコンテキストから呼び出した場合、i の無いサービスコールを非タスクコンテキストから呼び出した場合、共に E_CTX エラーは検出されずに正常に処理されます。

ただし、これは本カーネルの実装仕様であり、今後のバージョンでは変更される可能性があります。

アプリケーションのポータビリティを高めるには、「i で始まるサービスコールは非タスクコンテキスト専用、その他はタスクコンテキスト専用」というルールを遵守したプログラミングが推奨されます。

6.6.2 CPU ロック状態

CPU ロック状態から呼び出し可能なサービスコールは以下のものに限定されています。これら以外のサービスコールを CPU ロック状態から呼び出した場合、E_CTX エラーは検出されず、その後の動作は保証されません。ただし、待ち状態に遷移するサービスコールでは、E_CTX エラーを検出します。

- ext_tsk(CPU ロック状態は解除されます)
- exd_tsk(CPU ロック状態は解除されます)
- sns_tex
- loc_cpu, iloc_cpu
- unl_cpu, iunl_cpu
- sns_ctx
- sns_loc
- sns_dsp
- sns_dpn
- vsta_knl, ivsta_knl
- vsys_dwn, ivsys_dwn
- vsns_tmr

6.6.3 ディスパッチ禁止状態

待ち状態に遷移するサービスコールを呼び出すと、E_CTX エラーを返します。

6.6.4 ノーマル CPU 例外ハンドラ

ノーマル CPU 例外ハンドラから呼び出し可能なサービスコールは以下のみに制限されています。

- iras_tex
- sns_tex
- sns_loc
- sns_dsp
- sns_dpn
- get_tid, iget_tid
- vsta_knl, ivsta_knl
- vsys_dwn, ivsys_dwn
- vsns_tmr

その他のサービスコールをノーマル CPU 例外ハンドラから呼び出した場合、E_CTX エラーは検出されず、その後の動作は保証されません。

6.6.5 タスクコンテキストで SR.IMASK を 0 以外に変更している場合

この間は、非タスクコンテキストと扱われます。

RENESAS

6.7 ID 番号

6.7.1 概要

タスクやセマフォなど、サービスコールによって操作する対象を「オブジェクト」と呼びます。オブジェクトは、ID 番号によって識別されます。ID 番号は符号付 16 ビット整数型で表現します。

ID 番号空間は、オブジェクト種別毎に独立しています。また、各種別の ID 番号空間は、全 CPU を通じて 1 面です。

ID 番号は、図 6.1に示すように CPUID とローカルオブジェクト ID から構成されます。

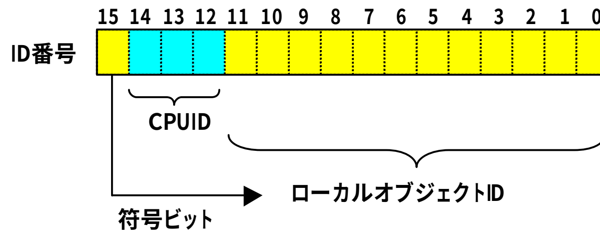


図6.1 ID 番号

(1) CPUID

各 CPU には、1 から順に CPUID が付与されます。

SH2A-DUAL では、CPU#0 の CPUID は 1、CPU#1 の CPUID は 2 と決められています。サービスコールで指定する操作対象オブジェクト ID に自 CPU 以外の CPUID を指定すると、それは「リモートサービスコール」と扱われます。

VCPU_SELF により、サービスコール呼び出し元と同じ CPU の指定ができます。VCPU_SELF は kernel.h で 0 に定義されているマクロで、μITRON4.0 仕様外です。

なお、CPUID は符号なしとして扱います。

(2) ローカルオブジェクト ID

ローカルオブジェクト ID は、1～__MAX_???の範囲を持ちます。_MAX_???は、cfg72mp が kernel_cfg.h に出力する各オブジェクトの「最大 ID」です。詳細は、「6.31.2(8) cfg72mp が kernel_cfg.h に出力するカーネル構成マクロ（μITRON4.0 仕様外）」を参照してください。

ID 番号の bit15 は、ローカルオブジェクト ID の符号ビットでもあります。負のローカルオブジェクト ID は、一部のサービスコールで特殊な指定を行うために用いられます。

6.7.2 ID 番号に関する関数マクロ

ID 番号と CPUID、ローカルオブジェクト ID を簡単に扱うための以下のマクロが用意されています。これらのマクロは kernel.h で定義されています。

なお、これらのマクロはμITRON4.0 仕様外の機能です。

- ID GET_CPUID(ID id)　　id の CPUID を返す
- ID GET_LOCALID(ID id)　id のローカルオブジェクト ID を返す
- ID MAKE_ID(ID cpuid, ID localid)　cpuid と localid からなる ID 番号を返す

6.8 サービスコールの振る舞い

6.8.1 リモートサービスコールとローカルサービスコール

操作対象のオブジェクト ID を引数に持つサービスコールでは、そのオブジェクト ID の CPUID によって操作対象オブジェクトが所属する CPU を指定します。ただし、他 CPU を指定できないサービスコールもあります。

他 CPU のオブジェクト ID を指定した場合は、他 CPU のカーネルに要求するサービスコールとなり、これを「リモートサービスコール」と呼びます。

一方、自 CPU のオブジェクト ID を指定した場合や、オブジェクト ID を引数に持たないサービスコールは、自 CPU のカーネルに要求するサービスコールとなり、これを「ローカルサービスコール」と呼びます。

図 6.2に、カーネルの構造を示します。

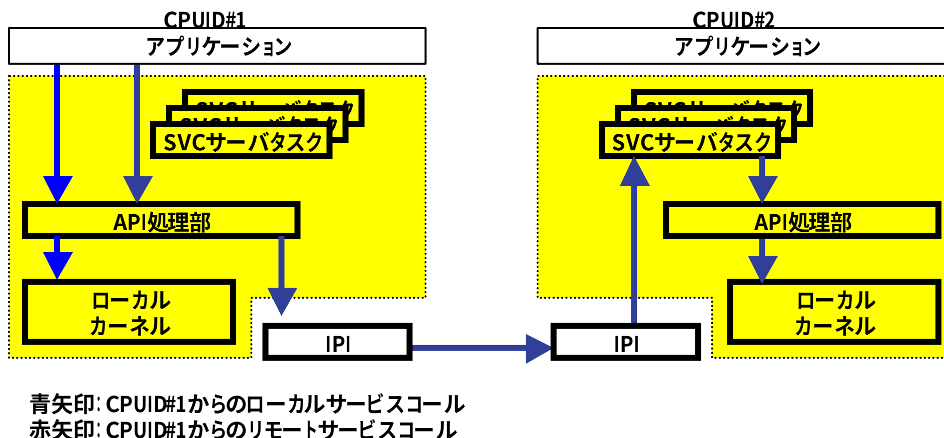


図6.2 カーネルの構造

図 6.2の黄色で示した部分が、それぞれの CPU におけるカーネルです。

ローカルカーネルは、従来のシングル CPU 用のμITRON4.0 仕様カーネルとほぼ等価のモジュールで、自 CPU 内のタスクや各種オブジェクトの管理と、タスクスケジューリング機能を持ちます。

API 処理部は、ローカルカーネルとアプリケーションの仲介を行う層です。サービスコールを呼び出すと、まず API 処理部が実行されます。API 処理部は、通常の関数と同じ扱いで実行します。すなわち、呼び出し元の状態(コンテキスト種別やスタック)を引き継いで動作し、処理中にタスクスイッチが発生することもあります。

SVC サーバタスクは、他 CPU からのリモートサービスコール要求を受け付け、要求元のタスクの代わりに自 CPU のカーネルにサービスコールを呼び出す役割を担います。

RENESAS

6.8.2 ローカルサービスコールの振る舞い

API 処理部は、自 CPU のローカルカーネルを呼び出します。ローカルカーネルからリターンすると、リターン値の設定などを行って呼び出し元にリターンします。

6.8.3 リモートサービスコールの振る舞い

各 CPU の cfg ファイルには、リモートサービスコールに関する以下の定義を行います。詳細は、「14.3.10 リモートサービスコール環境定義(remote_svc).」を参照してください。

- SVC サーバタスクの数(remote_svc.num_server)
- SVC サーバタスクの優先度(remote_svc.priority)
- SVC サーバタスクのスタック使用サイズ(remote_svc.stack_size)
- 使用する IPI ポート ID(remote_svc.ipi_portid)
- SVC サーバタスク空き待ちタスクの最大数(remote_svc.num_wait)

まず、リモートサービスコール環境を初期化する vini_rmt サービスコールでは、num_server 個の SVC サーバタスクを acre_tsk を用いて生成・起動します。SVC サーバタスクは、slp_tsk によって他 CPU からのリモートサービスコール要求を待ちます。

リモートサービスコールは、以下の手順で処理されます。

(1) リモートサービスコールが発行されると、API処理部は対象CPUの空きSVCサーバタスクをひとつ占有します。ただし、自CPU側または対象CPU側のvini_rmtが完了していない場合は、直ちにエラーEV_NOINITでリターンします。
対象CPUに空きSVCサーバタスクが存在しない場合は、空きができるまで呼び出し元タスクはAPI処理部内でslp_tskによる待ち状態になります。ただし、自CPU側のremote_svc.num_waitが0の場合は、直ちにエラーEV_NORESOURCEでリターンします。
また、この待ち状態に移行する前に、待ち状態を管理するためにpget_mpfを用いてメモリを取得します。この固定長メモリプールは、vini_rmt内でacre_mpfを用いて生成されます。メモリブロック数はremote_svc.num_wait個です。pget_mpfがメモリプールに空きブロックがないなどの理由で失敗した場合は、呼び出し元にエラーEV_NORESOURCEが返ります。

(2) SVCサーバタスクの占有に成功すると、次にAPI処理部はIPIを利用して占有した対象CPUのSVCサーバタスクを起床します。この時使用するIPIポートは、対象CPU側のremote_svc.ipi_portidです。そして、SVCサーバタスクの処理が完了するまで、呼び出し元タスクはAPI処理部内でslp_tskによる待ち状態になります。

(3) 対象CPU側で起床されたSVCサーバタスクは、自CPU(サービスコール発行元から見ると対象CPU)のカーネルに対して要求されたサービスコール(ローカルサービスコール)を呼び出します。このときの振る舞いは、「6.8.2 ローカルサービスコールの振る舞い」と同じです。
このサービスコールからリターンすると、SVCサーバタスクはリターン値の設定などを行い、処理完了をIPIを利用して要求元CPUに通知します。この時使用するIPIポートは、呼び出し元CPU側のremote_svc.ipi_portidです。

(4) API処理部内で待ち状態になっていた要求元タスクが起床されます。API処理部は、リターン値の設定などを行ってサービスコールからリターンします。

6.8.4 リモートサービスコールの注意事項

(1) リモートサービスコールを呼び出す時、呼び出しタスクは起床要求キューイング数が0でなければなりません。
(2) リモートサービスコールを呼び出し中のタスク、およびSVCサーバタスクに対して、ter_tsk, rel_wai, sus_tskなど、タスク状態を変更する操作を行ってはなりません。
(3) タイムアウト付きのリモートサービスコールの場合、タイムアウトはSVCサーバタスクが発行するサービスコールで指定されます。「6.8.3 リモートサービスコールの振る舞い」に示すように、クライアント側の処理としてSVCサーバタスクによって処理される前にslp_tsk(タイムアウト指定なし)による待ち状態になることがあります。

6.9 μITRON4.0 仕様外の仕様

vset_tfl サービスコールなどのように"v", "iv", "V"で始まる名称は、μITRON4.0 仕様外の本カーネル独自の仕様です。

また、以下の"ixxx_yyy"('i'で始まる名称)のサービスコールは、μITRON4.0 仕様でタスクコンテキスト専用として用意されている"xxx_yyy"のサービスコールを非タスクコンテキストから呼び出せるようにしたもので、μITRON4.0 仕様外です。

icre_tsk, iacre_tsk, ista_tsk, ichg_pri, iget_pri, iref_tsk, iref_tst, isus_tsk, irsm_tsk, frsm_tsk, idef_tex, iref_tex, icre_sem, iacre_sem, ipol_sem, iref_sem, icre_flg, iacre_flg, iclr_flg, ipol_flg, iref_flg, icre_dtq, iacre_dtq, iref_dtq, icre_mbx, iacre_mbx, isnd_mbx, iprcv_mbx, iref_mbx, icre_mbf, iacre_mbf, ipsnd_mbf, iref_mbf, icre_mpf, iacre_mpf, ipget_mpf, iref_mpf, icre_mpl, iacre_mpl, ipget_mpl, iref_mpl, iset_tim, iget_tim, icre_cyc, iacre_cyc, ista_cyc, istp_cyc, iref_cyc, iacre_alm, iacre_alm, ista_alm, istp_alm, iref_alm, ista_ovr, istp_ovr, iref_ovr, idef_inh, ichg_ims, iget_ims, idef_svc, ical_svc, idef_exc, iref_cfg, iref_ver

6.10　サービスコールの説明形式

以降の節では、サービスコールについての詳細を図 6.3の形式で説明します。

節番号　　　　　　　機能(サービスコール名)

■C 言語 API

　サービスコール呼び出し形式

■引数

パラメータ名	意味
・	・
・	・

■リターン値

パラメータ名	意味
・	・
・	・

■パケットの構造

型	メンバ名	意味
・	・	・
・	・	・

■エラーコード

ニモニック	種別	意味
・	・	・
・	・	・

■機能説明

　　・・・

図6.3　サービスコールの説明形式

(1) エラーコード

(a) E_CTX
本カーネルでは、E_CTXエラーの検出は限定的です。以降の各サービスコールのエラーコード欄にE_CTXの記載がないサービスコールでは、E_CTXエラーは検出されません。

(b) E_NOSPT
組み込まれていないサービスコール(service_callでNOに定義したサービスコール)を呼び出した場合は、E_NOSPTエラーが返ります。
また、SVCサーバタスク数(remote_svc.num_server)に０を指定してコンフィギュレーションされたカーネルに対してリモートサービスコールを発行した場合も、E_NOSPTが返ります。
以降の各サービスコールの「エラーコード」欄には、本エラーコードは記載していません。

(c) EV_NOINIT, EV_NORESOURCE
リモートサービスコールを呼び出した場合に、これらのエラーが返る可能性があります。詳細は、「6.8.3 リモートサービスコールの振る舞い」を参照してください。
以降の各サービスコールの「エラーコード」欄には、本エラーコードは記載していません。

(2) エラーコードの種別

[k] 常に検出されるエラーです。

[p] system.parameter_checkにYESを設定した場合のみ検出されるエラーです。

RENESAS

6.11 タスク管理機能

表 6.4にタスク管理でサポートしているサービスコール一覧を示します。

表6.4 タスク管理サービスコール

項番	サービスコール *1		機能	呼び出し可能なシステム状態 *2						
				T	N	E	D	U	L	C
1	cre_tsk	[s]	タスクの生成(非スタティックスタック使用)	○		○	○	○		
	icre_tsk				○	○	○	○		
2	vscr_tsk	[s]	タスクの生成(スタティックスタック使用)	○		○	○	○		
	ivscr_tsk				○	○	○	○		
3	acre_tsk		タスクの生成(ID 番号自動割付け)	○		○	○	○		
	iacre_tsk				○	○	○	○		
4	del_tsk		タスクの削除	○		○	○	○		
5	act_tsk	[S] [R]	タスクの起動	○		○	△	○		
	iact_tsk	[S]			○	○	○	○		
6	can_act	[S] [R]	タスク起動要求のキャンセル	○		○	△	○		
	ican_act				○	○	○	○		
7	sta_tsk	[B] [R]	タスクの起動(起動コード指定)	○		○	△	○		
	ista_tsk				○	○	○	○		
8	ext_tsk	[B] [S]	自タスクの終了	○		○	○	○	○	
9	exd_tsk	[S]	自タスクの終了と削除	○		○	○	○	○	
10	ter_tsk	[B] [S] [R]	タスクの強制終了	○		○	△	○		
11	chg_pri	[B] [S] [R]	タスク優先度の変更	○		○	△	○		
	ichg_pri				○	○	○	○		
12	get_pri	[S] [R]	タスク優先度の参照	○		○	△	○		
	iget_pri				○	○	○	○		
13	ref_tsk	[R]	タスクの状態参照	○		○	△	○		
	iref_tsk				○	○	○	○		
14	ref_tst	[R]	タスクの状態参照(簡易版)	○		○	△	○		
	iref_tst				○	○	○	○		
15	vchg_tmd		タスク実行モードの変更	○		○	○	○		

【注】 *1 "[S]"はスタンダードプロファイルのサービスコール、"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
"[B]"はベーシックプロファイルのサービスコールです。
"[R]"は、リモート呼び出し可能なサービスコールです。

*2 それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
"C"はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
"○"はその状態から呼び出し可能、"△"は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

表 6.5にタスク管理の仕様を示します。

表6.5　タスク管理の仕様

項番	項目	内容
1	ローカルタスク ID	1～_MAX_TSK (最大 1023)
2	タスク優先度	1～TMAX_TPRI (最大 255)
3	タスク起動要求キューイング数の最大値	15 回
4	タスク属性	TA_HLNG : 高級言語記述 TA_ASM : アセンブリ言語記述 TA_ACT : タスク生成後にタスクを実行可能状態へ TA_COP1 : FPU を使用

6.11.1 タスクの生成
(cre_tsk, icre_tsk)
(acre_tsk, iacre_tsk：ID 番号自動割付け)
(vscr_tsk, ivscr_tsk)(スタティックスタック使用)

■C 言語 API

```
ER ercd = cre_tsk(ID tskid, T_CTSK *pk_ctsk);
ER ercd = icre_tsk(ID tskid, T_CTSK *pk_ctsk);
ER_ID tskid = acre_tsk(T_CTSK *pk_ctsk);
ER_ID tskid = iacre_tsk(T_CTSK *pk_ctsk);
ER ercd = vscr_tsk(ID tskid, T_CTSK *pk_ctsk);
ER ercd = ivscr_tsk(ID tskid, T_CTSK *pk_ctsk);
```

■引数

pk_ctsk　　タスク生成情報を格納したパケットへのポインタ
《cre_tsk、icre_tsk、vscr_tsk、ivscr_tsk》
tskid　　タスク ID

■リターン値

《cre_tsk、icre_tsk、vscr_tsk、ivscr_tsk》
正常終了（E_OK）、またはエラーコード
《acre_tsk、iacre_tsk》
生成したタスク ID(正の値)またはエラーコード

■パケットの構造

```
typedef    struct  {
           ATR     tskatr;      タスク属性
           VP_INT  exinf;       拡張情報
           FP      task;        タスク起動アドレス
           PRI     itskpri;     初期タスク優先度
           SIZE    stksz;       タスクスタックサイズ
           VP      stk;         タスクスタック領域の先頭アドレス
} T_CTSK;
```

■エラーコード

E_NOMEM	[k]	メモリ不足(タスクスタック領域を確保できない)
E_RSATR	[p]	予約属性(tskatr が不正)
E_PAR	[p]	パラメータエラー (1)stksz が 4 の倍数以外、stksz=0、stksz≧0x80000000 (2)itskpri≦0、自 CPU の TMAX_TPRI<itskpri
E_ID	[p]	不正 ID 番号 (1)CPUID が不正(cre_tsk, icre_tsk, vscr_tsk, ivscr_tsk) (GET_CPUID(tskid)が自 CPU でない) (2)ローカル ID 範囲外

```
                    (a)GET_LOCALID(tskid)≦0,
                     (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid),
                     (cre_tsk, icre_tsk, vscr_tsk, ivscr_tsk)
                    (b)GET_LOCALID(tskid)≦(GET_CPUID(tskid)の_MAX_STTSK)
                     (cre_tsk, icre_tsk)
E_OBJ       [k]     オブジェクト状態不正(tskidのタスクが休止状態でない、または自タスク指定)
                    (cre_tsk, icre_tsk, vscr_tsk, ivscr_tsk)
E_NOID      [k]     空き IDなし(acre_tsk, iacre_tsk)
```

■機能説明

cre_tsk, icre_tsk, acre_tsk, iacre_tsk サービスコールはデフォルトスタック領域またはユーザが確保したスタック領域を使用するタスクを、vscr_tsk, ivscr_tsk サービスコールはスタティックスタックを使用するタスクを生成します。生成したタスクは、TA_ACT 属性の指定がない場合は休止状態へ、TA_ACT 属性の指定がある場合は実行可能状態に移行します。

これらのサービスコールで生成できるのは、自 CPU のカーネルに属するタスクです。本カーネルでは、他 CPU のカーネルに属するオブジェクトを生成するサービスコールは用意されていません。

タスク生成時に行われる処理は、表 6.6の通りです。

表6.6　タスク生成時に行われる処理

項番	処理内容
1	タスク起動要求キューイング数をクリアする。
2	タスク例外処理ルーチンを定義されていない状態にする。
3	上限プロセッサ時間が設定されていない状態にする。
4	スタックを割り付ける。（cre_tsk, acre_tsk の場合）

以下に各パラメータの意味を示します。

(1) tskid

vscr_tsk, ivscr_tsk サービスコールの場合、スタティックスタックを使用するタスクを生成します。tskid のローカル ID には 1 〜(自 CPU の_MAX_STTSK)の値を指定します。tskid の CPUID には、VCPU_SELF または自 CPUID を指定しなければなりません。

cre_tsk, icre_tsk サービスコールの場合、デフォルトスタック領域またはユーザが確保したスタック領域を使用するタスクを生成します。tskid のローカル ID には、(自 CPU の_MAX_STTSK+1)〜(自 CPU の_MAX_TSK)の値を指定します。tskid の CPUID には、VCPU_SELF または自 CPUID を指定しなければなりません。

acre_tsk, iacre_tsk サービスコールもデフォルトスタック領域またはユーザが確保したスタック領域を使用するタスクを生成しますが、未登録のタスク ID を検索してその ID でタスクを生成し、その ID を tskid に返します。検索するローカルタスク ID 範囲は、(自 CPU の_MAX_STTSK+1)〜(自 CPU の_MAX_TSK)となります。リターンされるタスク ID の CPUID は、自 CPUID となります。

(2) tskatr

tskatr には属性として、タスクを記述した言語およびコプロセッサの使用を指定します。

```
tskatr := ( (TA_HLNG‖TA_ASM) [ |TA_ACT] [ |TA_COP1 ] )
```

◆ TA_HLNG（0x00000000）高級言語記述
◆ TA_ASM（0x00000001）アセンブラ記述

◆ TA_ACT（0x00000002）タスク生成後にタスクを実行可能状態へ
◆ TA_COP1（0x00000200）FPU を使用

TA_ACT 属性を指定した場合、対象タスクには拡張情報(exinf)がパラメータとして渡ります。
FPU レジスタもタスクのコンテキストとして保証するには、TA_COP1 属性を指定してください。なお、TA_COP1 属性はμITRON4.0 仕様の範囲外の属性です。

(3) exinf

exinf は、ユーザが生成するタスクに関する情報を設定するなどの目的で自由に使用できます。

(4) task

task には、タスクの起動アドレスを指定します。

(5) itskpri

itskpri には、タスク起動時の優先度の初期値として 1～TMAX_TPRI の値を指定します。

(6) stksz

stksz は、cre_tsk, icre_tsk, acre_tsk, iacre_tsk サービスコールでのみ有効なパラメータで、生成するタスクのスタックサイズを指定します。スタックサイズには 4 の倍数の値を指定します。
スタティックスタックを使用するタスクを生成する vscr_tsk, ivscr_tsk では、stksz は意味を持ちません。カーネルの処理では無視されます。

(7) stk

stk は、cre_tsk, icre_tsk, acre_tsk, iacre_tsk サービスコールでのみ有効なパラメータです。
stk に NULL を指定した場合は、スタックはデフォルトタスクスタック用領域から割り付けられます。生成に成功すると、デフォルトタスクスタック用領域の空きは以下の式で計算されるサイズだけ減少します。
◆ 減少サイズ ＝ stksz＋16
生成するタスクのスタックアドレスを指定することもできます。この場合、stksz で指定したサイズのスタック領域を確保し、その先頭アドレスを指定してください。

なお、vscr_tsk, ivscr_tsk サービスコールはμITRON4.0 仕様外の機能です。

6.11.2 タスクの削除(del_tsk)

■C 言語 API

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

■引数

tskid　　タスク ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)≦0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)
E_OBJ	[k]	オブジェクト状態不正(tskid のタスクが休止状態でない、または自タスク指定)

■機能説明

tskid で示されたタスクを削除します。削除されたタスクは、未登録状態へ移行します。また、削除されたタスク ID のプロファイルカウンタは 0 クリアされます。

tskid には、自 CPU のカーネルに属するタスクのみ指定できます。

tskid で示されたタスクのスタックがデフォルトタスクスタック用領域から割り付けられていた場合は、そのタスクのスタックは解放され、デフォルトタスクスタック用領域の空きは以下の式で算出されるサイズだけ増加します。

◆ 増加サイズ ＝ (生成時に指定した stksz)＋16

6.11.3　タスクの起動(act_tsk, iact_tsk)

■C 言語 API

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

■引数

tskid　　　タスク ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID　　[p]　　不正 ID 番号
(1)CPUID が不正(GET_CPUID(tskid)が不正)
(2)ローカル ID 範囲外
(GET_LOCALID(tskid)<0,
(GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid))
(3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)

E_CTX　　[k]　　コンテキストエラー
(GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し)

E_NOEXS　　[k]　　未登録(tskid のタスクが存在しない)

E_QOVR　　[k]　　キューイングのオーバフロー(actcnt＞15)

■機能説明

tskid で示されたタスクを起動します。起動したタスクは休止状態から実行可能状態へ移行します。タスク起動時に行われる処理は、表 6.7の通りです。

表6.7　タスク起動時に行われる処理

項番	処理内容
1	タスクのベース優先度と現在優先度を初期化する。
2	起床要求キューイング数をクリアする。
3	強制待ち要求ネスト数をクリアする。
4	保留例外要因をクリアする。
5	タスク例外処理禁止状態にする。
6	タスク付属イベントフラグのフラグパターンをクリアする。

tskid=TSK_SELF（0）の指定により、自タスクの指定になります。
対象タスクには、タスク生成時に指定したタスクの拡張情報がパラメータとして渡ります。
tskid で示されたタスクがスタティックスタックを使用するタスクで、そのスタックが解放（どのタスクも使用していない）されている場合は、tskid で示されたタスクが共有スタックを占有し、実行可能状態に移行します。そのスタックを他のタスクが使用している場合は、スタック領域を使用できないために tskid で示されたタスクは共有スタック待ち状態になり、共有スタックの待ち行列につながれます。共有スタックの待ち行列は、FIFO で管理されます。
対象タスクが休止状態でない場合には、本サービスコールによるタスクの起動要求は、最大 15 回まで記憶されます。

act_tsk では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。一方、iact_tsk では、tskid に他 CPU のカーネルに属するタスクを指定することはできません。

6.11.4 タスクの起動要求のキャンセル(can_act, ican_act)

■C 言語 API

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

■引数

tskid　　タスク ID

■リターン値

キューイングされていた起動要求の回数(正の値または 0)、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)<0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid)) (3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_CTX	[k]	コンテキストエラー (GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(tskid のタスクが生成されていない)

■機能説明

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

tskid=TSK_SELF（0）の指定により、自タスクの指定になります。

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

can_act では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。一方、ican_act では、tskid に他 CPU のカーネルに属するタスクを指定することはできません。

6.11.5 タスクの起動(起動コード指定)(sta_tsk, ista_tsk)

■C 言語 API

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

■引数

tskid	タスク ID
stacd	タスク起動コード

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)≦0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)
E_OBJ	[k]	オブジェクト状態不正(tskid のタスクが休止状態でない、または自タスク指定)

■機能説明

tskid で示されたタスクを起動します。起動したタスクは休止状態から実行可能状態へ移行します。この時、タスク起動時に行うべき処理(表 6.7参照)を行います。

起動したタスクには、パラメータとして stacd で示されたタスク起動コードが渡されます。

tskid で示されたタスクがスタティックスタックを使用するタスクで、そのスタックが解放（どのタスクも使用していない）されている場合は、tskid で示されたタスクが共有スタックを占有し、実行可能状態に移行します。そのスタックを他のタスクが使用している場合は、スタック領域を使用できないために tskid で示されたタスクは共有スタック待ち状態になり、共有スタックの待ち行列につながれます。共有スタックの待ち行列は、FIFO で管理されます。

sta_tsk では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。一方、ista_tsk では、tskid に他 CPU のカーネルに属するタスクを指定することはできません。

6.11.6　自タスクの終了(ext_tsk)
自タスクの終了と削除(exd_tsk)

■C 言語 API

```
void ext_tsk(void);
void exd_tsk(void);
```

■リターン値

サービスコールの呼び出し元には戻りません。

ただし、本サービスコールをシステム構築時に組み込まずに呼び出した場合には、R0 に E_NOSPT を設定してリターンします。

また、以下のエラーが発生するとシステムダウンとなります。

E_CTX　　[k]　　コンテキストエラー(許可されていないシステム状態からの呼び出し)

■機能説明

ext_tsk サービスコールは、自タスクを正常終了します。タスクの状態は、実行状態から休止状態へ移行します。起動要求がキューイングされている場合は、自タスクをいったん終了させた後に再起動します。

タスク終了時に行われる処理は、表 6.8の通りです。

表6.8　タスク終了時に行われる処理

項番	処理内容
1	タスクがロックしていたミューテックスをロック解除する。
2	上限プロセッサ時間を解除する。

exd_tsk サービスコールは、自タスクを正常終了し、さらに削除します。タスクの状態は、実行状態から未登録状態へ移行します。また、削除されたタスク ID のプロファイルカウンタは 0 クリアされます。

ext_tsk, exd_tsk サービスコールは、タスクが占有していたミューテックス以外の資源（セマフォやメモリブロックなど）を自動的に解放する機能はありません。タスクは、必ず終了する前に資源の解放を行ってください。

ext_tsk, exd_tsk サービスコールを呼び出したタスクが他のタスクとスタックを共有しており、そのスタックの待ち行列に他のタスクがつながれている場合、先頭のタスクをスタックの待ち行列から外し、スタック待ち状態を解除します。その際、スタック待ち解除されたタスクに関して、タスクを起動する際に行うべき処理を行い(表 6.7参照)、そのタスクは実行可能状態に移行します。

デフォルトタスクスタック用領域から割り付けられたスタックを使用しているタスクが exd_tsk サービスコールを呼び出した場合、そのタスクのスタックは解放され、デフォルトタスクスタック用領域の空きは以下の式で算出されるサイズだけ増加します。

◆ 増加サイズ ＝ (生成時に指定した stksz) ＋16

ext_tsk, exd_tsk サービスコールは、ディスパッチ禁止状態および CPU ロック状態からも呼び出せます。この場合、ディスパッチ禁止状態および CPU ロック状態は解除されます。

なお、タスク開始関数からリターンした場合は、ext_tsk サービスコールと同じ動作となります。

6.11.7 タスクの強制終了(ter_tsk)

■C 言語 API

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

■引数

tskid	タスク ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)≦0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)
E_OBJ	[k]	オブジェクト状態不正(tskid のタスクが休止状態)
E_ILUSE	[k]	サービスコール不正使用(対象タスクが自タスク)

■機能説明

tskid で示された他タスクを強制的に終了させます。終了させた他タスクは休止状態へ移行します。この時、表 6.8に示す処理が行われます。

起動要求がキューイングされている場合には、タスクを起動する際に行うべき処理を行い、対象タスクを実行可能状態に移行します。

本サービスコールによる強制終了要求は、次の場合には遅延されます。

◆ tskid で示されたタスクが vchg_tmd サービスコールにより強制終了をマスクしている場合

本サービスコールは、タスクが占有していたミューテックス以外の資源（セマフォやメモリブロックなど）を自動的に解放する機能はありません。タスクは、必ず終了する前に資源の解放を行ってください。

tskid で示されたタスクが他のタスクとスタックを共有しており、そのスタックの待ち行列に他のタスクがつながれている場合、先頭のタスクをスタックの待ち行列から外し、スタック待ち状態を解除します。その際、スタック待ちが解除されたタスクに関して、タスクを起動する際に行うべき処理を行い(表 6.7参照)、そのタスクは実行可能状態に移行します。

ter_tsk では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。

RENESAS

6.11.8 タスク優先度の変更(chg_pri, ichg_pri)

■C 言語 API

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

■引数

tskid	タスク ID
tskpri	タスクのベース優先度

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR	[p]	パラメータエラー (1)tskpri＜0 (2)tskpri＞GET_CPUID(tskid)の TMAX_TPRI
E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)<0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid)) (3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_CTX	[k]	コンテキストエラー (GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)
E_ILUSE	[k]	サービスコール不正使用(上限優先度の違反)
E_OBJ	[k]	オブジェクト状態不正(タスクが休止状態である)

■機能説明

tskid で示されたタスクのベース優先度を、tskpri で示された値に変更します。それに伴い、タスクの現在優先度も変更します。

tskid=TSK_SELF（0）の指定により、自タスク指定となります。

tskpri=TPRI_INI（0）の指定により、タスク生成時に指定した初期タスク優先度に戻します。

変更したタスク優先度は、タスクが終了、または本サービスコールを呼び出すまで有効です。タスクが休止状態になると終了前のタスク優先度は無効になり、次に起動されたときにはタスク生成時に指定した初期タスク優先度になります。

tskid で示されたタスクが何らかの待ち状態で待ちのオブジェクトの属性が TA_TPRI の場合、本サービスコールによって待ち行列が変更され、その結果待ち行列につながれていたタスクの待ち状態が解除されることがあります。

対象タスクが TA_CEILING 属性のミューテックスをロックしているかロックを待っている場合で、tskpri に指定されたベース優先度が、それらのミューテックスのいずれかの上限優先度よりも高い場合には、E_ILUSE を返します。

chg_pri では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。一方、ichg_pri では、tskid に他 CPU のカーネルに属するタスクを指定することはできません。

6.11.9 タスク優先度の参照(get_pri, iget_pri)

■C 言語 API

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

■引数

tskid	タスク ID
p_tskpri	対象タスクの現在優先度を返す記憶域へのポインタ

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)<0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid)) (3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_CTX	[k]	コンテキストエラー (GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)
E_OBJ	[k]	オブジェクト状態不正(タスクが休止状態である)

■機能説明

tskid で示されたタスクの現在優先度を参照し、p_tskpri の指す領域に返します。

tskid=TSK_SELF（0）の指定により、自タスク指定となります。

get_pri では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。一方、iget_pri では、tskid に他 CPU のカーネルに属するタスクを指定することはできません。

RENESAS

6.11.10 タスクの状態参照(ref_tsk, iref_tsk)

■C 言語 API

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

■引数

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

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

```
typedef    struct {
           STAT    tskstat;     タスク状態
           PRI     tskpri;      タスクの現在優先度
           PRI     tskbpri;     タスクのベース優先度
           STAT    tskwait;     待ち要因
           ID      wobjid;      待ちオブジェクト ID
           TMO     lefttmo;     タイムアウトするまでの時間
           UINT    actcnt;      起動要求キューイング数
           UINT    wupcnt;      起床要求キューイング数
           UINT    suscnt;      強制待ち要求ネスト数
           UINT    tskmode;     タスクの実行モード
           UINT    tflptn;      現在のタスク付属イベントフラグの値
} T_RTSK;
```

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)<0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid)) (3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_CTX	[k]	コンテキストエラー (GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)

■機能説明

tskid で示されたタスクの状態を参照し、pk_rtsk が指す領域に返します。
tskid=TSK_SELF（0）の指定により自タスクの指定になります。
pk_rtsk の指す領域には、以下の値を返します。なお、*のデータはタスクが休止状態の場合は無効です。また、機能が組み込まれていないものに対する情報は不定となります。

- tskstat

現在のタスクの状態です。tskstat には、次の値を返します。
- ◆ TTS_RUN（0x00000001）実行状態
- ◆ TTS_RDY（0x00000002）実行可能状態
- ◆ TTS_WAI（0x00000004）待ち状態
- ◆ TTS_SUS（0x00000008）強制待ち状態
- ◆ TTS_WAS（0x0000000c）二重待ち状態
- ◆ TTS_DMT（0x00000010）休止状態
- ◆ TTS_STK（0x40000000）共有スタック解放待ち状態

- tskpri

タスクの現在優先度です。タスクが休止状態の場合は、タスクの初期優先度を返します。

- tskbpri

タスクのベース優先度です。タスクが休止状態の場合は、タスクの初期優先度を返します。

- tskwait *

tskstat が TTS_WAI、TTS_WAS のときに有効で、次の値を返します。
- ◆ TTW_SLP（0x00000001）slp_tsk、tslp_tsk サービスコールによる待ち
- ◆ TTW_DLY（0x00000002）dly_tsk サービスコールによる待ち
- ◆ TTW_SEM（0x00000004）wai_sem、twai_sem サービスコールによる待ち
- ◆ TTW_FLG（0x00000008）wai_flg、twai_flg サービスコールによる待ち
- ◆ TTW_SDTQ（0x00000010）snd_dtq、tsnd_dtq サービスコールによる待ち
- ◆ TTW_RDTQ（0x00000020）rcv_dtq、trcv_dtq サービスコールによる待ち
- ◆ TTW_MBX（0x00000040）rcv_mbx、trcv_mbx サービスコールによる待ち
- ◆ TTW_MTX（0x00000080）loc_mtx、tloc_mtx サービスコールによる待ち
- ◆ TTW_SMBF（0x00000100）snd_mbf、tsnd_mbf サービスコールによる待ち
- ◆ TTW_RMBF（0x00000200）rcv_mbf、trcv_mbf サービスコールによる待ち
- ◆ TTW_MPF（0x00002000）get_mpf、tget_mpf サービスコールによる待ち
- ◆ TTW_MPL（0x00004000）get_mpl、tget_mpl サービスコールによる待ち
- ◆ TTW_TFL（0x00008000）vwai_tfl、vtwai_tfl サービスコールによる待ち

- wobjid *

tskstat が TTS_WAI、TTS_WAS のときに有効で、待ち対象のオブジェクト ID を返します。
wobjid のビット 14～12 は、tskid のタスクが属する CPUID となります。

- lefttmo *

対象タスクがタイムアウトするまでの時間を返します。対象タスクが dly_tsk サービスコールによる待ち状態の場合は、この値は不定値となります。

- actcnt *

現在の起動要求キューイング数を返します。

♦ wupcnt *

現在の起床要求キューイング数を返します。

♦ suscnt *

現在の強制待ち要求ネスト数を返します。

♦ tskmode *

vchg_tmd サービスコールで設定したタスク実行モードと、それによって遅延されている要求があるかを返します。tskmode には次の値を返します。

- ◆ ECM_SUS（0x00000001）強制待ち要求がマスクされている
- ◆ ECM_TER（0x00000002）強制終了要求がマスクされている
- ◆ PND_SUS（0x00000004）強制待ち要求が遅延されている
- ◆ PND_TER（0x00000008）強制終了要求が遅延されている

♦ tflptn *

現在のタスク付属イベントフラグの値を返します。ただし、タスク付属イベントフラグ機能を組み込んでいない場合には、この値は不定値を返します。

tskmode および tflptn は、μITRON4.0 仕様の範囲外のメンバです。

ref_tsk では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。一方、iref_tsk では、tskid に他 CPU のカーネルに属するタスクを指定することはできません。

6.11.11　タスクの状態参照(簡易版)(ref_tst, iref_tst)

■C 言語 API

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

■引数

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

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

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

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)<0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid)) (3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_CTX	[k]	コンテキストエラー (GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)

■機能説明

tskid で示されたタスクについて、タスク状態と待ち要因を参照し、pk_rtst が指す領域に返します。tskid=TSK_SELF（0）の指定により自タスクの指定になります。

pk_rtst の指す領域には、以下の値を返します。なお、*のデータはタスクが休止状態の場合は無効です。また、機能が組み込まれていないものに対する情報参照値は不定となります。

- tskstat

現在のタスクの状態です。tskstat には、次の値を返します。

- ◆ TTS_RUN（0x00000001）実行状態
- ◆ TTS_RDY（0x00000002）実行可能状態
- ◆ TTS_WAI（0x00000004）待ち状態
- ◆ TTS_SUS（0x00000008）強制待ち状態
- ◆ TTS_WAS（0x0000000c）二重待ち状態
- ◆ TTS_DMT（0x00000010）休止状態
- ◆ TTS_STK（0x40000000）共有スタック解放待ち状態

♦ tskwait *

tskstat が TTS_WAI、TTS_WAS のときに有効で、次の値を返します。

- ◆ TTW_SLP（0x00000001）slp_tsk、tslp_tsk サービスコールによる待ち
- ◆ TTW_DLY（0x00000002）dly_tsk サービスコールによる待ち
- ◆ TTW_SEM（0x00000004）wai_sem、twai_sem サービスコールによる待ち
- ◆ TTW_FLG（0x00000008）wai_flg、twai_flg サービスコールによる待ち
- ◆ TTW_SDTQ（0x00000010）snd_dtq、tsnd_dtq サービスコールによる待ち
- ◆ TTW_RDTQ（0x00000020）rcv_dtq、trcv_dtq サービスコールによる待ち
- ◆ TTW_MBX（0x00000040）rcv_mbx、trcv_mbx サービスコールによる待ち
- ◆ TTW_MTX（0x00000080）loc_mtx、tloc_mtx サービスコールによる待ち
- ◆ TTW_SMBF（0x00000100）snd_mbf、tsnd_mbf サービスコールによる待ち
- ◆ TTW_RMBF（0x00000200）rcv_mbf、trcv_mbf サービスコールによる待ち
- ◆ TTW_MPF（0x00002000）get_mpf、tget_mpf サービスコールによる待ち
- ◆ TTW_MPL（0x00004000）get_mpl、tget_mpl サービスコールによる待ち
- ◆ TTW_TFL（0x00008000）vwai_tfl、vtwai_tfl サービスコールによる待ち

ref_tst では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。一方、iref_tst では、tskid に他 CPU のカーネルに属するタスクを指定することはできません。

6.11.12　タスク実行モードの変更(vchg_tmd)

■C 言語 API

```
ER ercd = vchg_tmd(UINT tmd);
```

■引数

tmd　　　変更するタスク実行モード

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR	[p]	パラメータエラー(tmd が不正)
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)

■機能説明

自タスクのタスク実行モードを変更します。tmd にはタスク実行モードとして他タスクからの要求に対するマスクを設定できます。

- ◆ ECM_SUS（0x00000001）強制待ち要求マスクのセット
- ◆ ECM_TER（0x00000002）強制終了要求マスクのセット

強制待ち要求マスクをセットすると、sus_tsk, isus_tsk サービスコールが呼び出されても、vchg_tmd サービスコールによってマスクを解除（tmd=0 を指定）するまでその要求は遅延されます。

強制終了要求マスクをセットすると、ter_tsk サービスコールが呼び出されても、vchg_tmd サービスコールによってマスクを解除（tmd=0 を指定）するまでその要求は遅延されます。

タスク実行モードは、タスクコンテキストとして動作する拡張サービスコールルーチン、タスク例外処理ルーチンでは、呼び出し元タスクの状態を引き継ぎます。

強制待ち要求、強制終了要求の遅延状況は、ref_tsk, iref_tsk サービスコールで参照できます。

なお、本サービスコールはμITRON4.0 仕様外の機能です。

6.12 タスク付属同期機能

表 6.9にタスク付属同期でサポートしているサービスコール一覧を示します。

表6.9 タスク付属同期サービスコール

項番	サービスコール *1		機 能	呼び出し可能な状態 *2						
				T	N	E	D	U	L	C
1	slp_tsk	[B] [S]	起床待ち	○		○		○		
2	tslp_tsk	[S]	同上(タイムアウト有)	○		○		○		
3	wup_tsk	[B] [S] [R]	タスクの起床	○		○	△	○		
	iwup_tsk	[B] [S]			○	○	○	○		
4	can_wup	[B] [S] [R]	タスク起床要求のキャンセル	○		○	△	○		
	ican_wup				○	○	○	○		
5	rel_wai	[B] [S] [R]	待ち状態の強制解除	○		○	△	○		
	irel_wai	[B] [S]			○	○	○	○		
6	sus_tsk	[B] [S] [R]	強制待ち状態への移行	○		○	△	○		
	isus_tsk				○	○	○	○		
7	rsm_tsk	[B] [S] [R]	強制待ち状態からの再開	○		○	△	○		
	irsm_tsk				○	○	○	○		
8	frsm_tsk	[S] [R]	強制待ち状態からの強制再開	○		○	△	○		
	ifrsm_tsk				○	○	○	○		
9	dly_tsk	[B] [S]	自タスクの遅延	○		○		○		
10	vset_tfl	[R]	タスク付属イベントフラグのセット	○		○	△	○		
	ivset_tfl				○	○	○	○		
11	vclr_tfl	[R]	タスク付属イベントフラグのクリア	○		○	△	○		
	ivclr_tfl				○	○	○	○		
12	vwai_tfl		タスク付属イベントフラグ待ち	○		○		○		
13	vpol_tfl		同上(ポーリング)	○		○	○	○		
14	vtwai_tfl		同上(タイムアウト有)	○		○		○		

【注】 *1 "[S]"はスタンダードプロファイルのサービスコール、"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
“[B]”はベーシックプロファイルのサービスコールです。
“[R]”は、リモート呼び出し可能なサービスコールです。

*2 それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
“C”はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
“○”はその状態から呼び出し可能、”△”は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

表 6.10にタスク付属同期の仕様を示します。

表6.10　タスク付属同期の仕様

項番	項目	内容
1	タスク起床要求キューイング数の最大値	15 回
2	タスク強制待ち要求ネスト数の最大値	15 回
3	タスク付属イベントフラグビット数	32 ビット(下位 16 ビットは将来拡張用)
4	タスク付属イベントフラグ初期値	タスク起動時に 0 に初期化される
5	タスク付属イベントフラグ待ち条件	OR 待ち

RENESAS

6.12.1　起床待ち(slp_tsk, tslp_tsk)

■C 言語 API

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

■引数

《tslp_tsk》

tmout　　　タイムアウト指定

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR	[p]	パラメータエラー(tmout≦-2)
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_TMOUT	[k]	タイムアウト
E_RLWAI	[k]	待ち状態強制解除(待ちの間に rel_wai サービスコールが呼び出された)

■機能説明

自タスクを起床待ち状態に移行させます。ただし、自タスクに対する起床要求がキューイングされている場合は、起床要求キューイング数を 1 減らしてそのまま実行を継続します。

起床待ち状態は、wup_tsk, iwup_tsk サービスコールによって解除されます。

tslp_tsk サービスコールでは、tmout には待ち時間を指定します。

tmout に正の値を指定した場合、待ち状態のまま tmout 時間が経過すると、待ち状態は解除され、エラーコードとして E_TMOUT が返ります。

tmout=TMO_POL（0）を指定した場合、起床要求キューイング数が正なら起床要求キューイング数を 1 減らして実行を継続し、0 ならエラーコードとして E_TMOUT を返します。

tmout=TMO_FEVR（-1）を指定した場合、タイムアウト監視を行いません。この場合、slp_tsk サービスコールと同じ動作となります。

tmout に指定可能な最大値は、(0x7FFFFFFF−TIC_NUME)/TIC_DENO に制限されます。これより大きな値を指定した場合の動作は保証されません。

時間の管理方法については、「5.13.7 時間の精度」を参照してください。

6.12.2　タスクの起床(wup_tsk, iwup_tsk)

■C 言語 API

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

■引数

tskid　　　タスク ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)<0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid)) (3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_CTX	[k]	コンテキストエラー (GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)
E_OBJ	[k]	オブジェクト状態不正(tskid のタスクが休止状態である)
E_QOVR	[k]	キューイングのオーバフロー(wupcnt＞15)

■機能説明

slp_tsk、または tslp_tsk サービスコールの呼び出しにより待ち状態になっているタスクの待ち状態を解除します。

対象タスクが slp_tsk、または tslp_tsk サービスコールによる待ち状態でない場合には、本サービスコールによる起床要求は、最大 15 回まで記憶されます。

tskid=TSK_SELF（0）の指定により自タスクの指定になります。

wup_tsk では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。一方、iwup_tsk では、tskid に他 CPU のカーネルに属するタスクを指定することはできません。

RENESAS

6.12.3 タスク起床要求のキャンセル(can_wup, ican_wup)

■C 言語 API

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

■引数

tskid　　　タスク ID

■リターン値

キューイングされていた起床要求の回数(正の値または 0)、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)<0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid)) (3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_CTX	[k]	コンテキストエラー (GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(tskid のタスクが生成されていない)
E_OBJ	[k]	オブジェクト状態不正(tskid のタスクが休止状態である)

■機能説明

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

tskid=TSK_SELF（0）の指定により、自タスクの指定になります。

can_wup では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。一方、ican_wup では、tskid に他 CPU のカーネルに属するタスクを指定することはできません。

6.12.4 待ち状態の強制解除(rel_wai, irel_wai)

■C 言語 API

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

■引数

tskid　　タスク ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)≦0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)
E_OBJ	[k]	オブジェクト状態不正（tskid のタスクが待ち状態でない）

■機能説明

tskid で示されるタスクが何らかの待ち状態（強制待ち状態および共有スタック解放待ち状態は含まれません）の場合、それを強制的に解除します。本サービスコールにより待ち状態を解除したタスクには、エラーコードとして E_RLWAI が返ります。

二重待ち状態のタスクに対して本サービスコールを呼び出すと、対象タスクは強制待ち状態へ移行します。その後 rsm_tsk, irsm_tsk、または frsm_tsk, ifrsm_tsk サービスコールが呼び出され、強制待ち状態が解除されると、対象タスクにはエラーコードとして E_RLWAI が返されます。

なお、強制待ち状態を解除するには、rsm_tsk, irsm_tsk, frsm_tsk, ifrsm_tsk を使用してください。また、共有スタック待ち状態を解除するサービスコールはありません。

rel_wai では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。一方、irel_wai では、tskid に他 CPU のカーネルに属するタスクを指定することはできません。

6.12.5 強制待ち状態への移行(sus_tsk, isus_tsk)

■C 言語 API

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

■引数

tskid　　タスク ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)<0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid)) (3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_CTX	[k]	コンテキストエラー (1)タスクコンテキストかつディスパッチ禁止状態で tskid=TSK_SELF(0)または自タスク ID を指定 (2)GET_CPUID(tskid)が他 CPUID で、許可されていない状態からの呼び出し
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)
E_OBJ	[k]	オブジェクト状態不正(tskid のタスクが休止状態である)
E_QOVR	[k]	キューイングのオーバフロー(suscnt＞15)

■機能説明

tskid で示されたタスクの実行を中断させ、強制待ち状態へ移行します。tskid で示されたタスクが待ち状態にある場合は、二重待ち状態へ移行します。

tskid=TSK_SELF（0）の指定により自タスクの指定になります。

強制待ち状態は、rsm_tsk, irsm_tsk、または frsm_tsk, ifrsm_tsk サービスコールの呼び出しにより解除されます。

本サービスコールによる強制待ちの要求はネストします。強制待ち要求のネスト数は最大 15 回まで記憶されます。

本サービスコールによる強制待ちの要求は、次の場合には遅延されます。

(1) tskidで示されたタスクがvchg_tmdサービスコールにより、強制待ち要求をマスクしている場合は、vchg_tmdにより強制待ちを解除（tmd=0を指定）した時点で強制待ち状態に移行します。

(2) tskidで示されたタスクがdis_dspサービスコールを呼び出してシステム状態がディスパッチ禁止状態になっていた場合は、タスク実行状態に戻った時点で強制待ち状態に移行します。

遅延されている強制待ち要求も、rsm_tsk, irsm_tsk、または frsm_tsk, ifrsm_tsk サービスコールの呼び出しによってその要求を解除できます。したがって強制待ち状態への移行は、遅延解除時の強制待ち要求ネスト数が 0 でない場合に行います。

sus_tsk では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。一方、isus_tsk では、tskid に他 CPU のカーネルに属するタスクを指定することはできません。

6.12.6 強制待ち状態からの再開(rsm_tsk, irsm_tsk)
強制待ち状態からの強制再開(frsm_tsk, ifrsm_tsk)

■C 言語 API

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

■引数

tskid　　タスク ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)≦0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)
E_OBJ	[k]	オブジェクト状態不正（tskid のタスクが強制待ち状態でない）

■機能説明

tskid で示されたタスクの強制待ち状態を解除します。

具体的には、rsm_tsk, irsm_tsk サービスコールは、tskid で示されたタスクの強制待ち要求ネスト数を 1 減算し、その結果が 0 になった場合に強制待ち状態を解除します。frsm_tsk, ifrsm_tsk サービスコールは、強制待ち要求ネスト数を 0 にし、強制待ち状態を解除します。二重待ち状態の場合は、待ち状態に移行させます。

rsm_tsk, frsm_tsk では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。一方、irsm_tsk, ifrsm_tsk では、tskid に他 CPU のカーネルに属するタスクを指定することはできません。

6.12.7 タスク遅延(dly_tsk)

■C 言語 API

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

■引数

dlytim　　遅延時間

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_RLWAI	[k]	待ち状態強制解除(待ちの間に rel_wai サービスコールが呼び出された)

■機能説明

自タスクの状態を実行状態から時間経過待ち状態へ移行し、dlytim で指定された時間が経過するのを待ちます。dlytim で指定された時間が経過した時点で、自タスクの状態を実行可能状態に移行します。dlytim＝0 を指定した場合にも、自タスクを待ち状態に移行させます。

dlytim に指定可能な最大値は、(0xFFFFFFFF-TIC_NUME)/TIC_DENO に制限されます。これより大きな値を指定した場合の動作は保証されません。

本サービスコールは、tslp_tsk サービスコールとは異なり、dlytim 時間だけ実行を遅延して終了した場合に正常終了します。また、遅延時間中に wup_tsk, iwup_tsk サービスコールが実行されても、待ち状態は解除されません。遅延時間が経過する前に待ち状態を解除するのは、rel_wai, irel_wai または ter_tsk サービスコールが呼び出された場合に限られます。

時間の管理方法については、「5.13.7 時間の精度」を参照してください。

RENESAS

6.12.8 タスク付属イベントフラグのセット(vset_tfl, ivset_tfl)

■C 言語 API

```
ER ercd = vset_tfl(ID tskid, UINT setptn);
ER ercd = ivset_tfl(ID tskid, UINT setptn);
```

■引数

tskid	タスク ID
setptn	セットするビットパターン

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)<0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid)) (3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_CTX	[k]	コンテキストエラー (GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(tskid のタスクが生成されていない)
E_OBJ	[k]	オブジェクト状態不正(tskid のタスクが休止状態である)

■機能説明

tskid で示されたタスクのタスク付属イベントフラグを、setptn で示された値との論理和（OR）に更新します。ただし、タスク付属イベントフラグの下位 16 ビットは将来拡張用ですので、setptn の下位 16 ビットは 0 としてください。

tskid=TSK_SELF（0）の指定により自タスクの指定になります。

タスク付属イベントフラグ値の更新の結果、対象タスクの待ちビットパターンとの論理和が 0 以外になると、そのタスクの待ち状態を解除します。このとき、タスク付属イベントフラグは 0 クリアされます。

vset_tfl では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。一方、ivset_tfl では、tskid に他 CPU のカーネルに属するタスクを指定することはできません。

なお、本サービスコールはμITRON4.0 仕様外の機能です。

6.12.9 タスク付属イベントフラグのクリア(vclr_tfl, ivclr_tfl)

■C 言語 API

```
ER ercd = vclr_tfl(ID tskid, UINT clrptn);
ER ercd = ivclr_tfl(ID tskid, UINT clrptn);
```

■引数

tskid	タスク ID
clrptn	クリアするビットパターン

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)<0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid)) (3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_CTX	[k]	コンテキストエラー (GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[p]	未登録(tskid のタスクが生成されていない)
E_OBJ	[k]	オブジェクト状態不正(tskid のタスクが休止状態である)

■機能説明

tskid で示されたタスクのタスク付属イベントフラグを、clrptn との論理積（AND）に更新します。ただし、タスク付属イベントフラグの下位 16 ビットは将来拡張用ですので、clrptn の下位 16 ビットは 0xffff としてください。
tskid=TSK_SELF（0）の指定により自タスクの指定になります。

vclr_tfl では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。一方、ivclr_tfl では、tskid に他 CPU のカーネルに属するタスクを指定することはできません。

なお、本サービスコールはμITRON4.0 仕様外の機能です。

6.12.10 タスク付属イベントフラグ待ち(vwai_tfl, vpol_tfl, vtwai_tfl)

■C 言語 API

```
ER ercd = vwai_tfl(UINT waiptn, UINT *p_tflptn);
ER ercd = vpol_tfl(UINT waiptn, UINT *p_tflptn);
ER ercd = vtwai_tfl(UINT waiptn, UINT *p_tflptn, TMO tmout);
```

■引数

waiptn	待ちビットパターン
p_tflptn	待ち解除時のビットパターンを返す記憶域へのポインタ
《vtwai_tfl》	
tmout	タイムアウト指定

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR	[p]	パラメータエラー(waiptn=0、tmout≦-2)
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_TMOUT	[k]	タイムアウト
E_RLWAI	[k]	待ち状態強制解除(待ちの間に rel_wai サービスコールが呼び出された)

■機能説明

タスク付属イベントフラグの waiptn で示されたビットのいずれかがセットされるのを待ちます。p_tflptn の指す領域には、待ち解除される時のタスク付属イベントフラグのビットパターンを返します。また、待ち解除されるときには、タスク付属イベントフラグは 0 にクリアされます。

本サービスコール呼び出し時にすでに waiptn で示されたビットのいずれかがセットされていた場合は、本サービスコールは直ちに終了します。いずれのビットもセットされていない場合は、vwai_tfl, vtwai_tfl サービスコールの場合は待ち状態に移行し、vpol_tfl サービスコールの場合は直ちにエラー E_TMOUT で終了します。待ち状態となった場合は、その後 vset_tfl, ivset_tfl サービスコールによって待っているビットがセットされたときに待ち状態が解除されます。

なお、タスク起動時のタスク付属イベントフラグの値は 0 です。

vtwai_tfl サービスコールの場合、tmout には待ち時間を指定します。

tmout に正の値を指定した場合、待ち条件が成立しないまま tmout 時間が経過すると、エラーコードとして E_TMOUT を返します。tmout=TMO_POL（0）を指定した場合、vpol_tfl サービスコールと同じ処理を行います。tmout=TMO_FEVR（-1）を指定した場合、タイムアウト監視を行いません。したがって、vwai_tfl サービスコールと同じ処理を行います。

tmout に指定可能な最大値は、(0x7FFFFFFF-TIC_NUME)/TIC_DENO に制限されます。これより大きな値を指定した場合の動作は保証されません。

時間の管理方法については、「5.13.7 時間の精度」を参照してください。

なお、本サービスコールはμITRON4.0 仕様外の機能です。

6.13　タスク例外処理機能

表 6.11にタスク例外処理でサポートしているサービスコール一覧を示します。

表6.11　タスク例外処理サービスコール

項番	サービスコール　*1	機　能	呼び出し可能なシステム状態　*2						
			T	N	E	D	U	L	C
1	def_tex　[s]	タスク例外処理ルーチンの定義	○		○	○	○		
	idef_tex			○	○	○	○		
2	ras_tex　[S]	タスク例外処理の要求	○		○	○	○		○
	iras_tex　[S]			○	○	○	○		○
3	dis_tex　[S]	タスク例外処理の禁止	○		○	○	○		
4	ena_tex　[S]	タスク例外処理の許可	○		○	○	○		
5	sns_tex　[S]	タスク例外処理禁止状態の参照	○	○	○	○	○	○	○
6	ref_tex	タスク例外処理の状態参照	○		○	○	○		
	iref_tex			○	○	○	○		

【注】 *1　"[S]"はスタンダードプロファイルのサービスコール、"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
“[B]”はベーシックプロファイルのサービスコールです。
“[R]”は、リモート呼び出し可能なサービスコールです。

*2　それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
“C”はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
“○”はその状態から呼び出し可能、”△”は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

表 6.12にタスク例外の仕様を示します。

表6.12　タスク例外の仕様

項番	項目	内容
1	例外要因	32 ビット
2	タスク起動時の状態	・タスク例外処理禁止状態 ・保留例外要因なし
3	サポート属性	TA_HLNG : 高級言語記述 TA_ASM : アセンブリ言語記述 TA_COP1 : FPU を使用

タスク例外処理ルーチンは、以下に示す条件が揃ったときに、タスクコンテキストとして起動されます。

- ◆ タスク例外処理許可状態
- ◆ 保留例外要因が 0 以外
- ◆ タスク実行状態
- ◆ 非タスクコンテキストまたはノーマル CPU 例外ハンドラが実行されていない

タスク例外処理ルーチンからリターンすると、タスク例外処理ルーチンを起動する前に実行していた処理を継続します。この時、このタスクはタスク例外許可状態に移行します。ここで、保留例外要因が 0 でない場合には、再びタスク例外処理ルーチンが起動されます。

6.13.1 タスク例外処理ルーチンの定義(def_tex, idef_tex)

■C 言語 API

```
ER ercd = def_tex(ID tskid, T_DTEX *pk_dtex);
ER ercd = idef_tex(ID tskid, T_DTEX *pk_dtex);
```

■引数

tskid	タスク ID
pk_dtex	タスク例外処理ルーチン定義情報を格納したパケットへのポインタ

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

```
typedef   struct {
          ATR      texatr;      タスク例外処理ルーチン属性
          FP       texrtn;      タスク例外処理ルーチン起動アドレス
} T_DTEX;
```

■エラーコード

E_RSATR	[p]	予約属性(texatr が不正)
E_ID	[p]	不正 ID 番号
		(1)CPUID が不正(GET_CPUID(tskid)が自 CPU でない)
		(2)ローカル ID 範囲外
		(GET_LOCALID(tskid)<0,
		(GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid))
		(3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)

■機能説明

tskid で示されたタスク例外処理ルーチンを pk_dtex で示された内容で定義します。
tskid には、自 CPU のカーネルに属するタスクのみ指定できます。

tskid=TSK_SELF（0）の指定により自タスクの指定になります。
texatr には属性として、タスク例外処理ルーチンを記述した言語およびコプロセッサの使用を指定します。

```
texatr := ( (TA_HLNG‖TA_ASM) [ |TA_COP1 ] )
```

◆ TA_HLNG（0x00000000）高級言語記述
◆ TA_ASM（0x00000001）アセンブラ記述
◆ TA_COP1（0x00000200）FPU を使用

FPU レジスタもタスク例外処理ルーチンのコンテキストとして保証するには、TA_COP1 属性を指定してください。なお、TA_COP1 属性はμITRON4.0 仕様の範囲外の属性です。
texrtn には、タスク例外処理ルーチンの先頭アドレスを指定します。
def_tex, idef_tex サービスコールでは、pk_dtex＝NULL（0）と指定した場合には tskid のタスク例外処理ルーチンの定義を解除します。この時、タスクの保留例外要因を 0 クリアし、タスクをタスク例外処理禁止状態に移行します。
すでにタスク例外処理ルーチンが定義されていた場合は、以前の定義を解除し新しい定義に置き換えます。この時、保留例外要因のクリアとタスク例外処理の禁止は行いません。

6.13.2 タスク例外処理の要求(ras_tex, iras_tex)

■C 言語 API

```
ER ercd = ras_tex(ID tskid, TEXPTN rasptn);
ER ercd = iras_tex(ID tskid, TEXPTN rasptn);
```

■引数

tskid	タスク ID
rasptn	要求するタスク例外処理のタスク例外要因

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR	[p]	パラメータエラー(rasptn=0)
E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)<0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid)) (3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)
E_OBJ	[k]	オブジェクト状態不正(tskid のタスクが休止状態である、またはタスク例外処理ルーチンが定義されていない)

■機能説明

tskid で示されたタスクに対して、rasptn で指定されるタスク例外要因によって、タスク例外処理を要求します。つまり、対象タスクの保留例外要因を、rasptn で示された値との論理和（OR）に更新します。

tskid には、自 CPU のカーネルに属するタスクのみ指定できます。

tskid=TSK_SELF（0）の指定により自タスクの指定になります。

このサービスコールにより、タスク例外処理ルーチンを起動する条件が揃った場合には、タスク例外処理ルーチンを起動する処理を行います。

本サービスコールは、ノーマル CPU 例外ハンドラからも呼び出せます。

6.13.3 タスク例外処理の禁止(dis_tex)

■C 言語 API

```
ER ercd = dis_tex(void);
```

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_OBJ	[k]	オブジェクト状態不正(自タスクにタスク例外処理ルーチンが定義されていない)
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)

■機能説明

自タスクをタスク例外処理禁止状態に移行させます。

6.13.4　タスク例外処理の許可(ena_tex)

■C 言語 API

```
ER ercd = ena_tex(void);
```

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_OBJ	[k]	オブジェクト状態不正(自タスクにタスク例外処理ルーチンが定義されていない)
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)

■機能説明

自タスクを、タスク例外許可状態に移行させます。

このサービスコールにより、タスク例外処理ルーチンを起動する条件が揃った場合には、タスク例外処理ルーチンを起動する処理を行います。

6.13.5　タスク例外禁止状態の参照(sns_tex)

■C 言語 API

```
BOOL state = sns_tex(void);
```

■リターン値

タスク例外禁止状態の場合に TRUE、タスク例外許可状態の場合に FALSE を返します。

■機能説明

実行状態のタスクが、タスク例外禁止状態かどうかを調べます。

実行状態のタスクとは、タスクコンテキストから呼び出した場合は自タスクであり、非タスクコンテキストから呼び出した場合は非タスクコンテキストに移行する直前に実行していたタスクです。非タスクコンテキストから呼び出された場合で、実行状態のタスクがないときには TRUE を返します。

タスク例外処理ルーチンが定義されていないタスクは、タスク例外処理禁止状態に保たれているため、実行状態のタスクにタスク例外処理ルーチンが定義されていない場合には、このサービスコールは TRUE を返します。

本サービスコールは、CPU ロック状態およびノーマル CPU 例外ハンドラからも呼び出せます。

6.13.6 タスク例外処理の状態参照(ref_tex, iref_tex)

■C 言語 API

```
ER ercd = ref_tex(ID tskid, T_RTEX *pk_rtex);
ER ercd = iref_tex(ID tskid, T_RTEX *pk_rtex);
```

■引数

tskid	タスク ID
pk_rtex	タスク例外処理状態を返すパケットへのポインタ

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

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

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)<0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid)) (3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)
E_OBJ	[k]	オブジェクト状態不正(tskid のタスクが休止状態である、またはタスク例外処理ルーチンが定義されていない)

■機能説明

tskid で示されたタスクのタスク例外処理に関する状態を参照し、pk_rtex が指す領域に返します。
texstat には、対象タスクがタスク例外許可状態かタスク例外処理禁止状態かによって次のいずれかの値を返します。

◆ TTEX_ENA（0x00000000）タスク例外許可状態
◆ TTEX_DIS（0x00000001）タスク例外禁止状態

pndptn には、対象タスクの保留例外要因を返します。処理されていない例外処理要求がないときには、pndptn には 0 を返します。
tskid には、自 CPU のカーネルに属するタスクのみ指定できます。
tskid=TSK_SELF（0）の指定により自タスクの指定になります。

6.14 同期･通信(セマフォ)機能

表 6.13にセマフォでサポートしているサービスコール一覧を示します。

表6.13 同期･通信（セマフォ）サービスコール

項番	サービスコール *1		機 能	呼び出し可能な状態 *2						
				T	N	E	D	U	L	C
1	cre_sem	[s]	セマフォの生成	○		○	○	○		
	icre_sem				○	○	○	○		
2	acre_sem		セマフォの生成	○		○	○	○		
	iacre_sem		(ID 番号自動割付け)		○	○	○	○		
3	del_sem		セマフォの削除	○		○	○	○		
4	sig_sem	[B] [S] [R]	セマフォ資源の返却	○		○	△	○		
	isig_sem	[B] [S]			○	○	○	○		
5	wai_sem	[B] [S] [R]	セマフォ資源の獲得	○		○		○		
6	pol_sem	[B] [S] [R]	同上(ポーリング)	○		○	△	○		
	ipol_sem				○	○	○	○		
7	twai_sem	[S] [R]	同上(タイムアウト有)	○		○		○		
8	ref_sem	[R]	セマフォの状態参照	○		○	△	○		
	iref_sem				○	○	○	○		

【注】 *1 "[S]"はスタンダードプロファイルのサービスコール､"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
“[B]”はベーシックプロファイルのサービスコールです。
“[R]”は、リモート呼び出し可能なサービスコールです。

*2 それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
“C”はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
“○”はその状態から呼び出し可能、”△”は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

表 6.14にセマフォの仕様を示します。

表6.14 セマフォの仕様

項番	項目	内容
1	ローカルセマフォ ID	1～_MAX_SEM (最大 1023)
2	セマフォカウンタ最大値	65535
3	サポート属性	TA_TFIFO : 待ちタスクのキューイングは FIFO TA_TPRI : 待ちタスクのキューイングは優先度順

6.14.1 セマフォの生成(cre_sem, icre_sem)
(acre_sem, iacre_sem：ID 番号自動割付け)

■C 言語 API

```
ER ercd = cre_sem(ID semid, T_CSEM *pk_csem);
ER ercd = icre_sem(ID semid, T_CSEM *pk_csem);
ER_ID semid = acre_sem(T_CSEM *pk_csem);
ER_ID semid = iacre_sem(T_CSEM *pk_csem);
```

■引数

pk_csem　セマフォ生成情報を格納したパケットへのポインタ

《cre_sem、icre_sem》

semid　セマフォ ID

■リターン値

《cre_sem、icre_sem》

正常終了（E_OK）、またはエラーコード

《acre_sem、iacre_sem》

生成したセマフォの ID 番号（正の値）またはエラーコード

■パケットの構造

```
typedef    struct {
           ATR       sematr;      セマフォ属性
           UINT      isemcnt;     セマフォの資源数の初期値
           UINT      maxsem;      セマフォの最大資源数
} T_CSEM;
```

■エラーコード

E_RSATR	[p]	属性不正(sematr が不正)
E_PAR	[p]	パラメータエラー(maxsem=0、maxsem＞0xffff、isemcnt＞maxsem)
E_ID	[p]	不正 ID 番号(cre_sem, icre_sem) (1)CPUID が不正(GET_CPUID(semid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(semid)≦0, (GET_CPUID(semid)の_MAX_SEM<GET_LOCALID(semid))
E_OBJ	[k]	オブジェクト状態不正(semid のセマフォが存在) (cre_sem, icre_sem)
E_NOID	[k]	空き ID なし(acre_sem, iacre_sem)

■機能説明

セマフォを生成します。

これらのサービスコールで生成できるのは、自 CPU のカーネルに属するセマフォです。本カーネルでは､他 CPU のカーネルに属するオブジェクトを生成するサービスコールは用意されていません。

cre_sem, icre_sem サービスコールは、semid で示された ID を持つセマフォを生成します。semid のローカル ID には、1～(自 CPU の_MAX_SEM)の値を指定します。semid の CPUID には、VCPU_SELF または自 CPUID を指定しなければなりません。

acre_sem, iacre_sem サービスコールは、未登録のセマフォ ID を検索してその ID でセマフォを生成し、その ID を semid に返します。検索するローカルセマフォ ID 範囲は、1～(自 CPU の_MAX_SEM) となります。リターンされるセマフォ ID の CPUID は、自 CPUID となります。

sematr には属性として、セマフォ資源獲得待ちタスクが待ち行列に並ぶ際の並び方を指定します。

```
sematr := (TA_TFIFO || TA_TPRI)
```

◆ TA_TFIFO（0x00000000）待ちタスクのキューイングは FIFO

◆ TA_TPRI（0x00000001）待ちタスクのキューイングは優先度順

isemcnt には、生成するセマフォの初期値を指定します。指定できる値の範囲は、0 から maxsem です。

maxsem には、生成するセマフォの最大資源数を指定します。指定できる値の範囲は、1 から 65,535 です。

RENESAS

6.14.2 セマフォの削除(del_sem)

■C 言語 API

```
ER ercd = del_sem(ID semid);
```

■引数

semid　　セマフォ ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号
		(1)CPUID が不正(GET_CPUID(semid)が自 CPU でない)
		(2)ローカル ID 範囲外
		(GET_LOCALID(semid)≦0,
		(GET_CPUID(semid)の_MAX_SEM)<GET_LOCALID(semid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(semid のセマフォが存在しない)

■機能説明

semid で示されたセマフォを削除します。

semid には、自 CPU のカーネルに属するセマフォのみ指定できます。

semid で示されたセマフォで資源獲得を待っているタスクがあった場合でもエラーにはなりませんが、待ち状態だったタスクは待ち状態が解除され、エラーコードとして E_DLT が返されます。

6.14.3 セマフォ資源の返却(sig_sem, isig_sem)

■C 言語 API

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

■引数

semid　　セマフォ ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(semid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(semid)≦0, (GET_CPUID(semid)の_MAX_SEM)<GET_LOCALID(semid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(semid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(semid のセマフォが存在しない)
E_QOVR	[k]	キューイングオーバフロー(semcnt＞maxsem[*1])

*1　maxsem：セマフォ生成時に指定した最大セマフォ資源数

■機能説明

semid で示されたセマフォに資源をひとつ返却します。対象セマフォで待っているタスクがあれば、セマフォの待ち行列先頭タスクに資源を割り付けて待ち状態を解除します。セマフォに対して待っているタスクがなければ、そのセマフォのカウント値を 1 増やします。

セマフォのカウント値の最大値は、セマフォ生成時に指定した maxsem です。

sig_sem では、ディスパッチ保留状態以外の場合のみ、semid に他 CPU のカーネルに属するセマフォを指定することができます。一方、isig_sem では、semid に他 CPU のカーネルに属するセマフォを指定することはできません。

6.14.4 セマフォ資源の獲得(wai_sem, pol_sem, ipol_sem, twai_sem)

■C 言語 API

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

■引数

semid　　セマフォ ID

《twai_sem》

tmout　　タイムアウト指定

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR	[p]	パラメータエラー(tmout≦-2)
E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(semid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(semid)≦0, (GET_CPUID(semid)の_MAX_SEM)<GET_LOCALID(semid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(semid のセマフォが存在しない)
E_DLT	[k]	待ちオブジェクト削除(semid のセマフォが削除された)
E_RLWAI	[k]	待ち状態強制解除(待ちの間に rel_wai サービスコールが呼び出された)
E_TMOUT	[k]	ポーリング失敗、またはタイムアウト

■機能説明

semid で指定されるセマフォから、資源をひとつ獲得します。

対象セマフォの資源数が 1 以上の場合には、セマフォの資源数から 1 を減じ、実行を継続します。資源数が 0 の場合には、wai_sem, twai_sem サービスコールでは呼び出しタスクはそのセマフォの待ち行列につながれ、pol_sem, ipol_sem サービスコールでは直ちにエラーE_TMOUT で終了します。待ち行列は、生成時に指定した属性にしたがって管理されます。

twai_sem サービスコールの場合、tmout には待ち時間を指定します。

tmout に正の値を指定した場合、待ち解除の条件が満たされないまま tmout 時間が経過すると、エラーコードとして E_TMOUT を返します。tmout=TMO_POL（0）を指定した場合、pol_sem サービスコールと同じ処理を行います。tmout=TMO_FEVR（-1）を指定した場合、タイムアウト監視を行いません。この場合、wai_sem サービスコールと同じ動作となります。

tmout に指定可能な最大値は、(0x7FFFFFFF-TIC_NUME)/TIC_DENO に制限されます。これより大きな値を指定した場合の動作は保証されません。

時間の管理方法については、「5.13.7 時間の精度」を参照してください。

wai_sem, pol_sem, twai_sem サービスコールでは、ディスパッチ保留状態以外の場合のみ、semid に他 CPU のカーネルに属するセマフォを指定することができます。一方、ipol_sem では、semid に他 CPU のカーネルに属するセマフォを指定することはできません。

6.14.5 セマフォの状態参照(ref_sem, iref_sem)

■C 言語 API

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

■引数

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

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

```
typedef    struct {
           ID      wtskid;      待ちタスク ID
           UINT    semcnt;      現在のセマフォカウント値
} T_RSEM;
```

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(semid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(semid)≦0, (GET_CPUID(semid)の_MAX_SEM)<GET_LOCALID(semid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(semid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(semid のセマフォが存在しない)

■機能説明

semid で示されたセマフォの状態を参照します。

pk_rsem が指す領域に、待ち行列の先頭タスク ID(wtskid)、現在のセマフォカウント値(semcnt)を返します。

待ちタスク ID のビット 14～12 は、必ず対象セマフォが属する CPUID(1 または 2)となります。

他 CPU のタスクが対象セマフォで待つ場合は、実際にはそのタスクの代わりに対象セマフォと同じ CPU に属する SVC サーバタスクが対象セマフォで待つので、このような場合には待ちタスク ID にこの SVC サーバタスク ID が返ります。

対象セマフォの待ちタスクが無い場合は、待ちタスク ID として TSK_NONE（0）を返します。

ref_sem では、ディスパッチ保留状態以外の場合のみ、semid に他 CPU のカーネルに属するセマフォを指定することができます。一方、iref_sem では、semid に他 CPU のカーネルに属するセマフォを指定することはできません。

6.15 同期･通信(イベントフラグ)機能

表 6.15にイベントフラグでサポートしているサービスコール一覧を示します。

表6.15 同期･通信（イベントフラグ）サービスコール

項番	サービスコール *1		機 能	呼び出し可能なシステム状態 *2						
				T	N	E	D	U	L	C
1	cre_flg	[s]	イベントフラグの生成	○		○	○	○		
	icre_flg				○	○	○	○		
2	acre_flg		イベントフラグの生成 (ID 番号自動割付け)	○		○	○	○		
	iacre_flg				○	○	○	○		
3	del_flg		イベントフラグの削除	○		○	○	○		
4	set_flg	[B] [S] [R]	イベントフラグのセット	○		○	△	○		
	iset_flg	[B] [S]			○	○	○	○		
5	clr_flg	[B] [S] [R]	イベントフラグのクリア	○		○	△	○		
	iclr_flg				○	○	○	○		
6	wai_flg	[B] [S] [R]	イベントフラグ待ち	○		○		○		
7	pol_flg	[B] [S] [R]	同上(ポーリング)	○		○	△	○		
	ipol_flg	[S]			○	○	○	○		
8	twai_flg	[S] [R]	同上(タイムアウト有)	○		○		○		
9	ref_flg	[R]	イベントフラグの状態参照	○		○	△	○		
	iref_flg				○	○	○	○		

【注】 *1 "[S]"はスタンダードプロファイルのサービスコール、"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
“[B]”はベーシックプロファイルのサービスコールです。
“[R]”は、リモート呼び出し可能なサービスコールです。

*2 それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
“C”はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
“○”はその状態から呼び出し可能、”△”は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

表 6.16にイベントフラグの仕様を示します。

表6.16 イベントフラグの仕様

項番	項目	内容
1	ローカルイベントフラグ ID	1～_MAX_FLAG(最大 1023)
2	イベントフラグのビット数	32 ビット
3	サポート属性	TA_TFIFO : 待ちタスクのキューイングは FIFO TA_TPRI : 待ちタスクのキューイングは優先度順 TA_WSGL : 複数タスクの待ちを許さない TA_WMUL : 複数タスクの待ちを許す TA_CLR : 待ち解除時にイベントフラグを 0 クリア

6.15.1 イベントフラグの生成(cre_flg, icre_flg)
(acre_flg, iacre_flg：ID 番号自動割付け)

■C 言語 API

```
ER ercd = cre_flg(ID flgid, T_CFLG *pk_cflg);
ER ercd = icre_flg(ID flgid, T_CFLG *pk_cflg);
ER_ID flgid = acre_flg(T_CFLG *pk_cflg);
ER_ID flgid = iacre_flg(T_CFLG *pk_cflg);
```

■引数

pk_cflg　　イベントフラグ生成情報を格納したパケットへのポインタ

《cre_flg、icre_flg》

flgid　　イベントフラグ ID

■リターン値

《cre_flg、icre_flg》

正常終了（E_OK）、またはエラーコード

《acre_flg、iacre_flg》

生成したイベントフラグの ID 番号（正の値）、またはエラーコード

■パケットの構造

```
typedef    struct {
           ATR      flgatr;      イベントフラグ属性
           FLGPTN   iflgptn;     イベントフラグの初期値
} T_CFLG;
```

■エラーコード

E_RSATR	[p]	属性不正(flgatr が不正)
E_ID	[p]	不正 ID 番号(cre_flg, icre_flg) (1)CPUID が不正(GET_CPUID(flgid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(flgid)≦0, (GET_CPUID(flgid)の_MAX_FLAG<GET_LOCALID(flgid))
E_OBJ	[k]	オブジェクト状態不正(flgid のイベントフラグが存在)（cre_flg, icre_flg）
E_NOID	[k]	空き ID なし(acre_flg, iacre_flg)

RENESAS

■機能説明

イベントフラグを生成します。

これらのサービスコールで生成できるのは、自 CPU のカーネルに属するイベントフラグです。本カーネルでは、他 CPU のカーネルに属するオブジェクトを生成するサービスコールは用意されていません。

cre_flg, icre_flg サービスコールは、flgid で示された ID を持つイベントフラグを生成します。flgid のローカル ID には、1～(自 CPU の_MAX_FLAG)の値を指定します。flgid の CPUID には、VCPU_SELF または自 CPUID を指定しなければなりません。

acre_flg, iacre_flg サービスコールは、未登録のイベントフラグ ID を検索してその ID でイベントフラグを生成し、その ID を flgid に返します。検索するローカルイベントフラグ ID 範囲は、1～(自 CPU の_MAX_FLAG)となります。リターンされるイベントフラグ ID の CPUID は、自 CPUID となります。

flgatr には属性として、イベントフラグ待ちタスクが待ち行列に並ぶ際の並び方と、生成するイベントフラグに待つことのできるタスク数を指定します。

flgatr := ((TA_TFIFO ‖ TA_TPRI) | (TA_WSGL ‖ TA_WMUL) | [TA_CLR])

◆ TA_TFIFO（0x00000000）待ちタスクのキューイングは FIFO

◆ TA_TPRI（0x00000001）待ちタスクのキューイングは優先度順

◆ TA_WSGL（0x00000000）複数タスクの待ちを許さない

◆ TA_WMUL（0x00000002）複数タスクの待ちを許す

◆ TA_CLR（0x00000004）待ち解除時にイベントフラグを 0 クリア

TA_WSGL 属性のイベントフラグでは、待つことのできるタスクは 1 つになります。この場合は、TA_TFIFO と TA_TPRI のどちらの属性を指定してもイベントフラグの動作は同じになります。これに対して、TA_WMUL 属性のイベントフラグでは、複数のタスクが待ち状態に遷移することができます。TA_CLR 属性が指定された場合は、タスクをイベントフラグ待ち状態から解除する時に、イベントフラグのビットパターンのすべてのビットをクリアします。

iflgptn には、生成するイベントフラグの初期値を指定します。

6.15.2 イベントフラグの削除(del_flg)

■C 言語 API

```
ER ercd = del_flg(ID flgid);
```

■引数

flgid　　イベントフラグ ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(flgid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(flgid)≦0, (GET_CPUID(flgid)の_MAX_FLAG)<GET_LOCALID(flgid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(flgid のイベントフラグが存在しない)

■機能説明

flgid で示されたイベントフラグを削除します。
flgid には、自 CPU のカーネルに属するイベントフラグのみ指定できます。
flgid で示されたイベントフラグで条件成立を待っているタスクがあった場合でもエラーにはなりませんが、待ち状態だったタスクは待ち状態が解除され、エラーコードとして E_DLT が返されます。

6.15.3 イベントフラグのセット(set_flg, iset_flg)

■C 言語 API

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

■引数

flgid	イベントフラグ ID
setptn	セットするビットパターン

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(flgid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(flgid)≦0, (GET_CPUID(flgid)の_MAX_FLAG)<GET_LOCALID(flgid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(flgid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(flgid のイベントフラグが存在しない)

■機能説明

flgid で示されたイベントフラグを、setptn で示された値との論理和（OR）に更新します。

イベントフラグ値の更新の結果、そのイベントフラグで待っているタスクの待ち解除条件を満たせば、そのタスクの待ち状態を解除します。なお、待ち解除条件は待ち行列の順に評価されます。この時、対象のイベントフラグ属性に TA_CLR 属性が指定されている場合には、イベントフラグのビットパターンのすべてのビットをクリアし、サービスコールの処理を終了します。

イベントフラグに TA_WMUL 属性が指定されており、TA_CLR 属性が指定されていない場合、set_flg の 1 回の呼び出しで複数のタスクが待ち解除される可能性があります。待ち解除されるタスクが複数ある場合には、イベントフラグの待ち行列につながれていた順序で待ち解除されます。

set_flg では、ディスパッチ保留状態以外の場合のみ、flgid に他 CPU のカーネルに属するイベントフラグを指定することができます。一方、iset_flg では、flgid に他 CPU のカーネルに属するイベントフラグを指定することはできません。

6.15.4 イベントフラグのクリア(clr_flg, iclr_flg)

■C 言語 API

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

■引数

flgid	イベントフラグ ID
clrptn	クリアするビットパターン

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(flgid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(flgid)≦0, (GET_CPUID(flgid)の_MAX_FLAG)<GET_LOCALID(flgid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(flgid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(flgid のイベントフラグが存在しない)

■機能説明

flgid で示されたイベントフラグを、clrptn で示された値との論理積（AND）に更新します。

clr_flg では、ディスパッチ保留状態以外の場合のみ、flgid に他 CPU のカーネルに属するイベントフラグを指定することができます。一方、iclr_flg では、flgid に他 CPU のカーネルに属するイベントフラグを指定することはできません。

6.15.5 イベントフラグ待ち(wai_flg, pol_flg, ipol_flg, twai_flg)

■C 言語 API

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

■引数

flgid	イベントフラグ ID
waiptn	待ちビットパターン
wfmode	待ちモード
p_flgptn	待ち解除時のビットパターンを返す記憶域へのポインタ
《twai_flg》	
tmout	タイムアウト値

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR	[p]	パラメータエラー(waiptn=0、wfmode が不正、tmout≦-2)
E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(flgid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(flgid)≦0, (GET_CPUID(flgid)の_MAX_FLAG)<GET_LOCALID(flgid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(flgid のイベントフラグが存在しない)
E_ILUSE	[k]	サービスコール不正使用(TA_WSGL 属性のイベントフラグに待ちタスクが存在)
E_DLT	[k]	待ちオブジェクト削除(flgid のイベントフラグが削除された)
E_TMOUT	[k]	ポーリング失敗、またはタイムアウト
E_RLWAI	[k]	待ち状態強制解除(待ちの間に rel_wai サービスコールが呼び出された)

■機能説明

flgid で指定されるイベントフラグのビットパターンが、waiptn と wfmode で指定される待ち解除条件を満たすのを待ちます。p_flgptn の指す領域には、待ち解除される時のイベントフラグのビットパターンを返します。

対象イベントフラグが TA_WSGL 属性で、すでに他のタスクが待っている場合は、本サービスコールはエラーE_ILUSE となります。

本サービスコール呼び出し時にすでに待ち解除条件が成立している場合は、本サービスコールは直ちに終了します。待ち解除条件が成立していない場合は、wai_flg, twai_flg サービスコールの場合はイベント待ち行列につながれ、pol_flg, ipol_flg サービスコールの場合は直ちにエラーE_TMOUT で終了します。

wfmode には、次のような指定を行います。

```
wfmode := ( (TWF_ANDW || TWF_ORW) )
```

◆ TWF_ANDW（0x00000000）AND 待ち

◆ TWF_ORW（0x00000001）OR 待ち

TWF_ANDW では、waiptn で指定したビットの全てがセットされるのを待ちます。TWF_ORW では、flgid で示されたイベントフラグのうち waiptn で指定したビットのいずれかがセットされるのを待ちます。

twai_flg サービスコールの場合、tmout には待ち時間を指定します。

tmout に正の値を指定した場合、待ち条件が満たされないまま tmout 時間が経過すると、エラーコードとして E_TMOUT を返します。tmout=TMO_POL（0）を指定した場合、pol_flg サービスコールと同じ処理を行います。tmout=TMO_FEVR（-1）を指定した場合、タイムアウト監視を行いません。この場合、wai_flg サービスコールと同じ動作となります。

tmout に指定可能な最大値は、(0x7FFFFFFF-TIC_NUME)/TIC_DENO に制限されます。これより大きな値を指定した場合の動作は保証されません。

時間の管理方法については、「5.13.7 時間の精度」を参照してください。

wai_flg, pol_flg, twai_flg では、ディスパッチ保留状態以外の場合のみ、flgid に他 CPU のカーネルに属するイベントフラグを指定することができます。一方、ipol_flg では、flgid に他 CPU のカーネルに属するイベントフラグを指定することはできません。

6.15.6 イベントフラグの状態参照(ref_flg, iref_flg)

■C 言語 API

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

■引数

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

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

```
typedef    struct {
           ID      wtskid;      待ちタスク ID
           FLGPTN  flgptn;      イベントフラグのビットパターン
} T_RFLG;
```

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(flgid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(flgid)≦0, (GET_CPUID(flgid)の_MAX_FLAG)<GET_LOCALID(flgid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(flgid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(flgid のイベントフラグが存在しない)

■機能説明

flgid で示されたイベントフラグの状態を参照します。

pk_rflg の指す領域に、待ち行列の先頭タスク ID(wtskid)、現在のイベントフラグのビットパターン(flgptn)を返します。

待ちタスク ID のビット 14～12 は、必ず対象イベントフラグが属する CPUID(1 または 2)となります。

他 CPU のタスクが対象イベントフラグで待つ場合は、実際にはそのタスクの代わりに対象イベントフラグと同じ CPU に属する SVC サーバタスクが対象イベントフラグで待つので、このような場合には待ちタスク ID にこの SVC サーバタスク ID が返ります。

対象イベントフラグの待ちタスクが無い場合は、待ちタスク ID として TSK_NONE（0）を返します。

ref_flg では、ディスパッチ保留状態以外の場合のみ、flgid に他 CPU のカーネルに属するイベントフラグを指定することができます。一方、iref_flg では、flgid に他 CPU のカーネルに属するイベントフラグを指定することはできません。

6.16 同期･通信(データキュー)機能

表 6.17にデータキューでサポートしているサービスコール一覧を示します。

表6.17 同期･通信（データキュー）サービスコール

項番	サービスコール *1		機 能	呼び出し可能なシステム状態 *2						
				T	N	E	D	U	L	C
1	cre_dtq	[s]	データキューの生成	○		○	○	○		
	icre_dtq				○	○	○	○		
2	acre_dtq		データキューの生成	○		○	○	○		
	iacre_dtq		(ID 番号自動割付け)		○	○	○	○		
3	del_dtq		データキューの削除	○		○	○	○		
4	snd_dtq	[S] [R]	データキューへの送信	○		○		○		
5	psnd_dtq	[S] [R]	同上(ポーリング)	○		○	△	○		
	ipsnd_dtq	[S]			○	○	○	○		
6	tsnd_dtq	[S] [R]	同上(タイムアウト有)	○		○		○		
7	fsnd_dtq	[S] [R]	データキューへの強制送信	○		○	△	○		
	ifsnd_dtq	[S]			○	○	○	○		
8	rcv_dtq	[S] [R]	データキューからの受信	○		○		○		
9	prcv_dtq	[S] [R]	同上(ポーリング)	○		○	△	○		
10	trcv_dtq	[S] [R]	同上(タイムアウト有)	○		○		○		
11	ref_dtq	[R]	データキューの状態参照	○		○	△	○		
	iref_dtq				○	○	○	○		

【注】 *1 "[S]"はスタンダードプロファイルのサービスコール､"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
“[B]”はベーシックプロファイルのサービスコールです。
“[R]”は、リモート呼び出し可能なサービスコールです。

*2 それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
“C”はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
“○”はその状態から呼び出し可能、”△”は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

表 6.18にデータキューの仕様を示します。

表6.18 データキューの仕様

項番	項目	内容
1	ローカルデータキューID	1～_MAX_DTQ (最大 1023)
2	1 ワード	32 ビット
3	サポート属性	TA_TFIFO : 送信待ちタスクのキューイングは FIFO TA_TPRI : 送信待ちタスクのキューイングは優先度順

RENESAS

6.16.1 データキューの生成(cre_dtq, icre_dtq)
(acre_dtq, iacre_dtq：ID 番号自動割付け)

■C 言語 API

```
ER ercd = cre_dtq(ID dtqid, T_CDTQ *pk_cdtq);
ER ercd = icre_dtq(ID dtqid, T_CDTQ *pk_cdtq);
ER_ID dtqid = acre_dtq(T_CDTQ *pk_cdtq);
ER_ID dtqid = iacre_dtq(T_CDTQ *pk_cdtq);
```

■引数

pk_cdtq　　データキュー生成情報を格納したパケットへのポインタ

《cre_dtq、icre_dtq》

dtqid　　データキューID

■リターン値

《cre_dtq、icre_dtq》

正常終了（E_OK）、またはエラーコード

《acre_dtq、iacre_dtq》

生成したデータキューの ID 番号（正の値）、またはエラーコード

■パケットの構造

```
typedef  struct {
         ATR     dtqatr;     データキュー属性
         UINT    dtqcnt;     データキュー領域の容量（データの個数）
         VP      dtq;        データキュー領域の先頭アドレス
} T_CDTQ;
```

■エラーコード

E_NOMEM	[k]	メモリ不足(データキュー領域が確保できない)
E_RSATR	[p]	属性不正(dtqatr が不正)
E_PAR	[p]	パラメータエラー (1)TSZ_DTQ(dtqcnt)が 32 ビット範囲を超えている
E_ID	[p]	不正 ID 番号(cre_dtq, icre_dtq) (1)CPUID が不正(GET_CPUID(dtqid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(dtqid)≦0, (GET_CPUID(dtqid)の_MAX_DTQ<GET_LOCALID(dtqid))
E_OBJ	[k]	オブジェクト状態不正(dtqid のデータキューが存在)(cre_dtq, icre_dtq)
E_NOID	[k]	空き ID なし(acre_dtq, iacre_dtq)

■機能説明

データキューを生成します。

これらのサービスコールで生成できるのは、自 CPU のカーネルに属するデータキューです。本カーネルでは、他 CPU のカーネルに属するオブジェクトを生成するサービスコールは用意されていません。

cre_dtq, icre_dtq サービスコールは、dtqid で示された ID を持つデータキューを生成します。dtqid のローカル ID には、1～(自 CPU の_MAX_DTQ)の値を指定します。dtqid の CPUID には、VCPU_SELF または自 CPUID を指定しなければなりません。

acre_dtq, iacre_dtq サービスコールは、未登録のデータキューID を検索してその ID でデータキューを生成し、その ID を dtqid に返します。検索するローカルデータキューID 範囲は、1～(自 CPU の_MAX_DTQ)となります。リターンされるデータキューID の CPUID は、自 CPUID となります。

(1) dtqatr

dtqatr には属性として、送信待ちタスクが待ち行列に並ぶ際の並び方を指定します。

```
dtqatr := (TA_TFIFO || TA_TPRI)
```

◆ TA_TFIFO（0x00000000）送信待ちタスクのキューイングは FIFO

◆ TA_TPRI（0x00000001）送信待ちタスクのキューイングは優先度順

受信待ちタスクの待ち行列は、dtqatr に関わらず FIFO（First-In First-Out）となります。

(2) dtqcnt

dtqcnt には、データキュー領域に格納できるデータの個数を指定します。

また、dtqcnt=0 でデータキューを生成することもできます。dtqcnt =0 で生成したデータキューではデータを蓄えておくことができないため、送信側と受信側の先に実行した方が待ち状態になり、他方が行われた時点で待ちが解除される、つまり送信側と受信側が完全に同期した動作となります。

(3) dtq

dtq には、データキューとして使用する空き領域の先頭アドレスを指定します。dtq から TSZ_DTQ(dtqcnt)バイトをデータキューとして使用します。TSZ_DTQ()は、データキューのサイズを算出するマクロです。

dtqcnt=0 の場合、dtq は意味を持たず単に無視されます。

dtq に NULL を指定すると、カーネルはデフォルトデータキュー用領域から TSZ_DTQ(dtqcnt)バイトのデータキュー領域を割り付けます。これにより、デフォルトデータキュー用領域の空きは以下の式で計算されるサイズだけ減少します。

◆ 減少サイズ ＝ TSZ_DTQ(dtqcnt) ＋ 16

6.16.2　データキューの削除(del_dtq)

■C 言語 API

```
ER ercd = del_dtq(ID dtqid);
```

■引数

dtqid　　　データキューID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(dtqid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(dtqid)≦0, (GET_CPUID(dtqid)の_MAX_DTQ)<GET_LOCALID(dtqid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(dtqid のデータキューが存在しない)

■機能説明

dtqid で示されたデータキューを削除します。

dtqid には、自 CPU のカーネルに属するデータキューのみ指定できます。

dtqid で示されたデータキューで送信待ち、受信待ちのタスクがあった場合でもエラーにはなりませんが、待ち状態だったタスクは待ち状態が解除され、エラーコードとして E_DLT が返されます。

削除により、デフォルトデータキュー用領域の空きは以下の式で計算されるサイズだけ増加します。

◆ 増加サイズ ＝ TSZ_DTQ(生成時に指定した dtqcnt) ＋ 16

6.16.3 データキューへの送信(snd_dtq,psnd_dtq,ipsnd_dtq,tsnd_dtq, fsnd_dtq, ifsnd_dtq)

■C 言語 API

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

■引数

dtqid	データキューID
data	データキューへ送信するデータ
《tsnd_dtq》	
tmout	タイムアウト指定

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR	[p]	パラメータエラー(tmout≦-2)
E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(dtqid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(dtqid)≦0, (GET_CPUID(dtqid)の_MAX_DTQ)<GET_LOCALID(dtqid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_ILUSE	[k]	サービスコール不正使用(dtqcnt が 0 のデータキューに対する fsnd_dtq, ifsnd_dtq の発行)
E_NOEXS	[k]	未登録(dtqid のデータキューが存在しない)
E_DLT	[k]	待ちオブジェクト削除(dtqid のデータキューが削除された)
E_TMOUT	[k]	ポーリング失敗、またはタイムアウト
E_RLWAI	[k]	待ち状態強制解除(待ちの間に rel_wai サービスコールが呼び出された)

■機能説明

dtqid で示されたデータキューに対して、data で示されたデータ（4 バイト）を送信します。

なお、fsnd_dtq, ifsnd_dtq は、dtqcnt=0 で生成したデータキューを指定した場合は常に E_ILUSE エラーとなります。

(1) 対象データキューに受信待ちタスクが存在する場合

データキューには格納せずに受信待ち行列の先頭タスクにデータを渡し、そのタスクの待ち状態を解除します。

(2) 対象データキューに受信待ちタスクが存在しない場合

(a) データキューに空きがある場合

data をデータキューの末尾に格納します。データキューカウントは＋1 されます。

(b) データキューに空きがない場合

- **snd_dtq, tsnd_dtq の場合**

  呼び出しタスクはデータキューの空き領域を待つための待ち行列（送信待ち行列）につながれます。
  tsnd_dtqサービスコールの場合、tmoutには待ち時間を指定します。
  tmoutに正の値を指定した場合、待ち条件が満たされないままtmout時間が経過すると、エラーコードとしてE_TMOUTを返します。tmout=TMO_POL（0）を指定した場合、psnd_dtqサービスコールと同じ処理を行います。tmout=TMO_FEVR（-1）を指定した場合、タイムアウト監視を行いません。したがって、snd_dtqサービスコールと同じ処理を行います。
  tmoutに指定可能な最大値は、(0x7FFFFFFF-TIC_NUME)/TIC_DENOに制限されます。これより大きな値を指定した場合の動作は保証されません。
  時間の管理方法については、「5.13.7 時間の精度」を参照してください。

- **psnd_dtq, ipsnd_dtq の場合**

  直ちにエラーE_TMOUTで終了します。

- **fsnd_dtq, ifsnd_dtq の場合**

  送信待ちタスクが存在するかどうかに関わらず、データキューの先頭のデータ(最も古いデータ)を削除した後、dataをデータキューの末尾に格納します。

psnd_dtq, snd_dtq, tsnd_dtq, および fsnd_dtq では、ディスパッチ保留状態以外の場合のみ、dtqid に他 CPU のカーネルに属するデータキューを指定することができます。一方、ipsnd_dtq および ifsnd_dtq では、dtqid に他 CPU のカーネルに属するデータキューを指定することはできません。

6.16.4 データキューからの受信(rcv_dtq, prcv_dtq, trcv_dtq)

■C 言語 API

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

■引数

dtqid　　データキューID

p_data　　受信したデータを返す記憶域へのポインタ

《trcv_dtq》

tmout　　タイムアウト指定

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR	[p]	パラメータエラー(tmout≦-2)
E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(dtqid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(dtqid)≦0, (GET_CPUID(dtqid)の_MAX_DTQ)<GET_LOCALID(dtqid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(dtqid のデータキューが存在しない)
E_DLT	[k]	待ちオブジェクト削除(dtqid のデータキューが削除された)
E_TMOUT	[k]	ポーリング失敗、またはタイムアウト
E_RLWAI	[k]	待ち状態強制解除(待ちの間に rel_wai サービスコールが呼び出された)

RENESAS

■機能説明

dtqid で示されたデータキューからデータを受信し、p_data の指す領域に格納します。

データキューにデータがあれば、その先頭のデータ（最古のメッセージ）を受信します。データキュー内のデータを受信することで、データキューカウントは−1 されます。この結果、送信待ち行列のタスクに対してもデータの格納が可能であれば、待ち行列の順にデータの送信処理を行います。

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

データキューにデータがなく、データ送信待ちタスクも存在しない場合、rcv_dtq, trcv_dtq サービスコールでは、呼び出しタスクはメッセージ到着を待つ待ち行列（受信待ち行列）につながれ、prcv_dtq サービスコールでは直ちにエラーE_TMOUT で終了します。受信待ち行列は、FIFO で管理されます。

trcv_dtq サービスコールの場合、tmout には待ち時間を指定します。

tmout に正の値を指定した場合、待ち解除の条件が満たされないまま tmout 時間が経過すると、エラーコードとして E_TMOUT を返します。tmout=TMO_POL（0）を指定した場合、prcv_dtq サービスコールと同じ処理を行います。tmout=TMO_FEVR（-1）を指定した場合、タイムアウト監視を行いません。したがって、rcv_dtq サービスコールと同じ処理を行います。

tmout に指定可能な最大値は、(0x7FFFFFFF-TIC_NUME)/TIC_DENO に制限されます。これより大きな値を指定した場合の動作は保証されません。

時間の管理方法については、「5.13.7 時間の精度」を参照してください。

これらのサービスコールでは、ディスパッチ保留状態以外の場合のみ、dtqid に他 CPU のカーネルに属するデータキューを指定することができます。

6.16.5 データキューの状態参照(ref_dtq, iref_dtq)

■C 言語 API

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

■引数

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

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

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

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(dtqid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(dtqid)≦0, (GET_CPUID(dtqid)の_MAX_DTQ)<GET_LOCALID(dtqid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(dtqid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(dtqid のデータキューが存在しない)

■機能説明

dtqid で示されたデータキューの状態を参照し、pk_rdtq が指す領域に送信待ちタスク ID(stskid)、受信待ちタスク ID(rtskid)、データキューに入っているデータの数(sdtqcnt)を返します。

待ちタスク ID のビット 14～12 は､必ず対象データキューが属する CPUID(1 または 2)となります。

他 CPU のタスクが対象データキューで待つ場合は、実際にはそのタスクの代わりに対象データキューと同じ CPU に属する SVC サーバタスクが対象データキューで待つので、待ちタスク ID にこの SVC サーバタスク ID が返ることがあります。

送信待ちタスク、受信待ちタスクが無い場合は、待ちタスク ID として TSK_NONE（0）を返します。

ref_dtq では、ディスパッチ保留状態以外の場合のみ、dtqid に他 CPU のカーネルに属するデータキューを指定することができます。一方、iref_dtq では、dtqid に他 CPU のカーネルに属するデータキューを指定することはできません。

6.17 同期･通信(メールボックス)機能

表 6.19にメールボックスでサポートしているサービスコール一覧を示します。

表6.19 同期･通信（メールボックス）サービスコール

項番	サービスコール *1	機 能	呼び出し可能な状態 *2						
			T	N	E	D	U	L	C
1	cre_mbx [s]	メールボックスの生成	○		○	○	○		
	icre_mbx			○	○	○	○		
2	acre_mbx	メールボックスの生成 (ID 番号自動割付け)	○		○	○	○		
	iacre_mbx			○	○	○	○		
3	del_mbx	メールボックスの削除	○		○	○	○		
4	snd_mbx [B] [S] [R]	メールボックスへの送信	○		○	△	○		
	isnd_mbx			○	○	○	○		
5	rcv_mbx [B] [S] [R]	メールボックスからの受信	○		○		○		
6	prcv_mbx [B] [S] [R]	同上(ポーリング)	○		○	△	○		
	iprcv_mbx			○	○	○	○		
7	trcv_mbx [S] [R]	同上(タイムアウト有)	○		○		○		
8	ref_mbx [R]	メールボックスの状態参照	○		○	△	○		
	iref_mbx			○	○	○	○		

【注】 *1 "[S]"はスタンダードプロファイルのサービスコール､"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
“[B]”はベーシックプロファイルのサービスコールです。
“[R]”は、リモート呼び出し可能なサービスコールです。

*2 それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
“C”はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
“○”はその状態から呼び出し可能、”△”は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

表 6.20にメールボックスの仕様を示します。

表6.20 メールボックスの仕様

項番	項目	内容
1	ローカルメールボックス ID	1～_MAX_MBX (最大 1023)
2	メッセージ優先度	1～TMAX_MPRI (最大 255)
3	サポート属性	TA_TFIFO : 待ちタスクのキューイングは FIFO TA_TPRI : 待ちタスクのキューイングは優先度順 TA_MFIFO : メッセージのキューイングは FIFO TA_MPRI : メッセージのキューイングは優先度順

6.17.1 メールボックスの生成(cre_mbx, icre_mbx) (acre_mbx, iacre_mbx:ID 番号自動割付け)

■C 言語 API

```
ER ercd = cre_mbx(ID mbxid, T_CMBX *pk_cmbx);
ER ercd = icre_mbx(ID mbxid, T_CMBX *pk_cmbx);
ER_ID mbxid = acre_mbx(T_CMBX *pk_cmbx);
ER_ID mbxid = iacre_mbx(T_CMBX *pk_cmbx);
```

■引数

pk_cmbx　　メールボックス生成情報を格納したパケットへのポインタ

《cre_mbx、icre_mbx》

mbxid　　メールボックス ID

■リターン値

《cre_mbx、icre_mbx》

正常終了（E_OK）、またはエラーコード

《acre_mbx、iacre_mbx》

生成したメールボックスの ID 番号（正の値）、またはエラーコード

■パケットの構造

```
typedef    struct {
           ATR     mbxatr;     メールボックス属性
           PRI     maxmpri;    メッセージ優先度の最大値
           VP      mprihd;     優先度別メッセージキューヘッダの先頭アドレス
} T_CMBX;
```

■エラーコード

E_RSATR	[p]	属性不正(mbxatr が不正)
E_PAR	[p]	パラメータエラー(maxmpri≦0、maxmpri＞自 CPU の TMAX_MPRI)
E_ID	[p]	不正 ID 番号(cre_mbx, icre_mbx) (1)CPUID が不正(GET_CPUID(mbxid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(mbxid)≦0, (GET_CPUID(mbxid)の_MAX_MBX<GET_LOCALID(mbxid)))
E_OBJ	[k]	オブジェクト状態不正(mbxid のメールボックスが存在)(cre_mbx, icre_mbx)
E_NOID	[k]	空き ID なし(acre_mbx, iacre_mbx)

■機能説明

メールボックスを生成します。

これらのサービスコールで生成できるのは、自 CPU のカーネルに属するメールボックスです。本カーネルでは、他 CPU のカーネルに属するオブジェクトを生成するサービスコールは用意されていません。

cre_mbx, icre_mbx サービスコールは、mbxid で示された ID を持つメールボックスを生成します。mbxid のローカル ID には、1～(自 CPU の_MAX_MBX)の値を指定します。mbxid の CPUID には、VCPU_SELF または自 CPUID を指定しなければなりません。

acre_mbx, iacre_mbx サービスコールは、未登録のメールボックス ID を検索してその ID でメールボックスを生成し、その ID を mbxid に返します。検索するローカルメールボックス ID 範囲は、1～(自 CPU の_MAX_MBX)となります。リターンされるメールボックス ID の CPUID は、自 CPUID となります。

mbxatr には属性として、受信待ちタスクおよびメッセージが待ち行列に並ぶ際の並び方を指定します。

```
mbxatr := ( (TA_TFIFO‖TA_TPRI) | (TA_MFIFO‖TA_MPRI) )
```

◆ TA_TFIFO（0x00000000）受信待ちタスクのキューイングは FIFO
◆ TA_TPRI（0x00000001）受信待ちタスクのキューイングは優先度順
◆ TA_MFIFO（0x00000000）メッセージのキューイングは FIFO
◆ TA_MPRI（0x00000002）メッセージのキューイングは優先度順

mprihd は、mbxatr に TA_MPRI を指定した場合は必ず NULL を指定してください。μITRON4.0 仕様では、NULL 以外を指定することで、mprihd で指定された領域にメッセージキューヘッダ領域を生成する仕様になっていますが、本カーネルは NULL 以外を指定した場合の機能はサポートしていません。**NULL 以外の指定を行った場合の動作は保証されません。**なお、TA_MPRI の指定がない場合は、mprihd は意味を持たず、単に無視されます。

6.17.2　メールボックスの削除(del_mbx)

■C 言語 API

```
ER ercd = del_mbx(ID mbxid);
```

■引数

mbxid	メールボックス ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(mbxid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(mbxid)≦0, (GET_CPUID(mbxid)の_MAX_MBX)<GET_LOCALID(mbxid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(mbxid のメールボックスが存在しない)

■機能説明

mbxid で示されたメールボックスを削除します。

mbxid には、自 CPU のカーネルに属するメールボックスのみ指定できます。

mbxid で示されたメールボックスでメッセージを待っているタスクがあった場合でもエラーにはなりませんが、待ち状態だったタスクは待ち状態が解除され、エラーコードとして E_DLT が返されます。また、メールボックス内にメッセージが存在する場合でもエラーにはなりませんが、メッセージ領域については何ら処理を行いません。たとえば、メモリプールから獲得したメモリブロックをメッセージとして使用していた場合でも、カーネルが自動的にメッセージ領域をメモリプールに返却するわけではありません。

6.17.3 メールボックスへの送信(snd_mbx, isnd_mbx)

■C 言語 API

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

■引数

mbxid	メールボックス ID
pk_msg	送信メッセージの先頭アドレス

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

《メールボックスのメッセージヘッダ》

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

《メールボックスの優先度付きメッセージヘッダ》

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

■エラーコード

E_PAR	[p]	パラメータエラー(メッセージ先頭 4 バイトが 0 以外)
	[k]	(msgpri≦0、msgpri＞自 CPU の TMAX_MPRI)
E_ID	[p]	不正 ID 番号
		(1)CPUID が不正(GET_CPUID(mbxid)が不正)
		(2)ローカル ID 範囲外
		(GET_LOCALID(mbxid)≦0,
		(GET_CPUID(mbxid)の_MAX_MBX)<GET_LOCALID(mbxid))
E_CTX	[k]	コンテキストエラー
		(GET_CPUID(mbxid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(mbxid のメールボックスが存在しない)

■機能説明

mbxid で示されたメールボックスに pk_msg で示されたメッセージを送信します。

すでに対象メールボックスにメッセージの受信を待つタスクが存在していれば、待ち行列先頭のタスクに送信したメッセージが渡され、そのタスクの待ち状態が解除されます。

メッセージの受信を待つタスクが存在しない場合は、メッセージをメッセージ待ち行列につなぎます。待ち行列は、生成時に指定した属性にしたがって管理されます。

TA_MFIFO 属性のメールボックスにメッセージを送る場合は、図 6.4に示すように先頭に T_MSG 構造体を付加した形式で、メッセージを作成してください。

TA_MPRI 属性のメールボックスにメッセージを送る場合は、図 6.5に示すように先頭に T_MSG_PRI 構造体を付加した形式で、メッセージを作成してください。

TA_MFIFO, TA_MPRI いずれの属性の場合もメッセージは RAM 領域に作成し、送信前に T_MSG 領域を 0 にしてください。

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

```
typedef struct {
    T_MSG   t_msg;      /* T_MSG 構造体                              */
    B       data[8];    /* ユーザメッセージデータ構造の例(任意の構造) */
} USER_MSG;
```

図6.4　メッセージの形式例

```
typedef struct {
    T_MSG_PRI  t_msg;    /* T_MSG_PRI 構造体                          */
    B          data[8];  /* ユーザメッセージデータ構造の例(任意の構造) */
} USER_MSG;
```

図6.5　優先度付きメッセージの形式例

snd_mbx では、ディスパッチ保留状態以外の場合のみ、mbxid に他 CPU のカーネルに属するメールボックスを指定することができます。この場合、メッセージは非キャッシャブル領域に作成しなければなりません。

一方、isnd_mbx では、mbxid に他 CPU のカーネルに属するメールボックスを指定することはできません。

6.17.4 メールボックスからの受信(rcv_mbx, prcv_mbx, iprcv_mbx, trcv_mbx)

■C 言語 API

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

■引数

mbxid　　メールボックス ID

ppk_msg　　受信メッセージ先頭アドレスを返す領域へのポインタ

《trcv_mbx》

tmout　　タイムアウト指定

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

《メールボックスのメッセージヘッダ》

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

《メールボックスの優先度付きメッセージヘッダ》

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

■エラーコード

E_PAR	[p]	パラメータエラー(tmout≦-2)
E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(mbxid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(mbxid)≦0, (GET_CPUID(mbxid)の_MAX_MBX)<GET_LOCALID(mbxid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(mbxid のメールボックスが存在しない)
E_DLT	[k]	待ちオブジェクト削除(mbxid のメールボックスが削除された)
E_TMOUT	[k]	ポーリング失敗、またはタイムアウト
E_RLWAI	[k]	待ち状態強制解除(待ちの間に rel_wai サービスコールが呼び出された)

■機能説明

mbxid で示されたメールボックスからメッセージを受信し、受信したメッセージの先頭アドレスを pk_msg に返します。

メールボックスにメッセージが存在しない場合は、rcv_mbx, trcv_mbx サービスコールでは、呼び出しタスクはメッセージ到着を待つ待ち行列（受信待ち行列）につながれ、prcv_mbx, iprcv_mbx サービスコールでは直ちにエラーE_TMOUT で終了します。待ち行列は、生成時に指定した属性にしたがって管理されます。

trcv_mbx サービスコールの場合、tmout には待ち時間を指定します。

tmout に正の値を指定した場合、待ち解除の条件が満たされないまま tmout 時間が経過すると、エラーコードとして E_TMOUT を返します。tmout=TMO_POL（0）を指定した場合、prcv_mbx サービスコールと同じ処理を行います。tmout=TMO_FEVR（-1）を指定した場合、タイムアウト監視を行いません。したがって、rcv_mbx サービスコールと同じ処理を行います。

tmout に指定可能な最大値は、(0x7FFFFFFF-TIC_NUME)/TIC_DENO に制限されます。これより大きな値を指定した場合の動作は保証されません。

時間の管理方法については、「5.13.7 時間の精度」を参照してください。

prcv_mbx, rcv_mbx, および trcv_mbx では、ディスパッチ保留状態以外の場合のみ、mbxid に他 CPU のカーネルに属するメールボックスを指定することができます。一方、iprcv_mbx では、mbxid に他 CPU のカーネルに属するメールボックスを指定することはできません。

RENESAS

6.17.5 メールボックスの状態参照(ref_mbx, iref_mbx)

■C 言語 API

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

■引数

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

■リターン値

正常終了(E_OK)、またはエラーコード

■パケットの構造

(1)T_RMBX

```
typedef     struct {
            ID      wtskid;       待ちタスク ID
            T_MSG   *pk_msg;      次に受信されるメッセージの先頭アドレス
} T_RMBX;
```

(2)T_MSG

《メールボックスのメッセージヘッダ》

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

《メールボックスの優先度付きメッセージヘッダ》

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

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(mbxid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(mbxid)≦0, (GET_CPUID(mbxid)の_MAX_MBX)<GET_LOCALID(mbxid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(mbxid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(mbxid のメールボックスが存在しない)

■機能説明

　mbxidで示されたメールボックスの状態を参照します。pk_rmbxが示す領域に待ちタスクID(wtskid)、次に受信されるメッセージの先頭アドレス(pk_msg)を返します。
　待ちタスクIDのビット14〜12は、必ず対象メールボックスが属するCPUID(1または2)となります。
　他CPUのタスクが対象メールボックスで待つ場合は、実際にはそのタスクの代わりに対象メールボックスと同じCPUに属するSVCサーバタスクが対象メールボックスで待つので、待ちタスクIDにこのSVCサーバタスクIDが返ることがあります。
　対象メールボックスの待ちタスクが無い場合は、待ちタスクIDとしてTSK_NONE（0）を返します。
　次に受信されるメッセージが無い場合は、メッセージの先頭アドレスとしてNULL（0）を返します。

　ref_mbxでは、ディスパッチ保留状態以外の場合のみ、mbxidに他CPUのカーネルに属するメールボックスを指定することができます。一方、iref_mbxでは、mbxidに他CPUのカーネルに属するメールボックスを指定することはできません。

6.18　拡張同期・通信(ミューテックス)機能

表 6.21にミューテックスでサポートしているサービスコール一覧を示します。

表6.21　同期・通信（ミューテックス）サービスコール

項番	サービスコール *1	機　能	呼び出し可能な状態 *2						
			T	N	E	D	U	L	C
1	cre_mtx	ミューテックスの生成	○		○	○	○		
2	acre_mtx	ミューテックスの生成 (ID 番号自動割付け)	○		○	○	○		
3	del_mtx	ミューテックスの削除	○		○	○	○		
4	loc_mtx	ミューテックスのロック	○		○		○		
5	ploc_mtx	同上(ポーリング)	○		○	○	○		
6	tloc_mtx	同上(タイムアウト有)	○		○		○		
7	unl_mtx	ミューテックスのロック解除	○		○	○	○		
8	ref_mtx	ミューテックスの状態参照	○		○	○	○		

【注】 *1 "[S]"はスタンダードプロファイルのサービスコール、"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
"[B]"はベーシックプロファイルのサービスコールです。
"[R]"は、リモート呼び出し可能なサービスコールです。

*2 それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
"C"はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
"○"はその状態から呼び出し可能、"△"は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

表 6.22にミューテックス機能の仕様を示します。

表6.22　ミューテックスの仕様

項番	項目	内容
1	ローカルミューテックス ID	1～_MAX_MTX (最大 1023)
2	サポート属性	TA_CEILING（優先度上限プロトコル）

【注】* 本カーネルにおける TA_CEILING 属性(優先度上限プロトコル)では、簡略化した優先度制御規則を採用しています。簡略化した優先度制御規則では、タスクの優先度を高くする制御はすべて行われますが、タスクの優先度を低くする制御は、タスクがロックしていたミューテックスが無くなったとき(複数のミューテックスをロックしていた場合は、それら全てを解放したとき)にのみ行われます。

6.18.1　ミューテックスの生成(cre_mtx)
(acre_mtx：ID 番号自動割付け)

■C 言語 API

```
ER ercd = cre_mtx(ID mtxid, T_CMTX *pk_cmtx);
ER_ID mtxid = acre_mtx(T_CMTX *pk_cmtx);
```

■引数

pk_cmtx　　ミューテックス生成情報を格納したパケットへのポインタ

《cre_mtx》

mtxid　　ミューテックス ID

■リターン値

《cre_mtx》

正常終了（E_OK）、またはエラーコード

《acre_mtx》

生成したミューテックスの ID 番号（正の値）、またはエラーコード

■パケットの構造

```
typedef    struct {
           ATR     mtxatr;      ミューテックス属性
           PRI     ceilpri;     ミューテックスの上限優先度
} T_CMTX;
```

■エラーコード

E_RSATR	[p]	属性不正(mtxatr が不正)
E_PAR	[p]	パラメータエラー(ceilpri≦0、ceilpri＞自 CPU の TMAX_TPRI)
E_ID	[p]	不正 ID 番号(cre_mtx) (1)CPUID が不正(GET_CPUID(mtxid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(mtxid)≦0, (GET_CPUID(mtxid)の_MAX_MTX<GET_LOCALID(mtxid))
E_OBJ	[k]	オブジェクト状態不正(mtxid のミューテックスが存在)（cre_mtx)
E_NOID	[k]	空き ID なし(acre_mtx, iacre_mtx)

RENESAS

■機能説明

ミューテックスを生成します。

これらのサービスコールで生成できるのは、自 CPU のカーネルに属するミューテックスです。本カーネルでは、他 CPU のカーネルに属するオブジェクトを生成するサービスコールは用意されていません。

cre_mtx サービスコールは、mtxid で示された ID を持つミューテックスを生成します。mtxid のローカル ID には、1〜(自 CPU の_MAX_MTX)の値を指定します。mtxid の CPUID には、VCPU_SELF または自 CPUID を指定しなければなりません。

acre_mtx サービスコールは､未登録のミューテックス ID を検索してその ID でミューテックスを生成し、その ID を mtxid に返します。検索するローカルミューテックス ID 範囲は、1〜(自 CPU の_MAX_MTX)となります。リターンされるミューテックス ID の CPUID は、自 CPUID となります。

mtxatr には属性としては、優先度上限プロトコル（TA_CEILING）のみを指定できます。

```
mtxatr := (TA_CEILING)
```

◆ TA_CEILING（0x00000003）優先度上限プロトコル

ceilpri には、生成するミューテックスの上限優先度を指定します。指定できる値の範囲は、1〜TMAX_TPRI です。

6.18.2 ミューテックスの削除(del_mtx)

■C 言語 API

```
ER ercd = del_mtx(ID mtxid);
```

■引数

mtxid	ミューテックス ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(mtxid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(mtxid)≦0, (GET_CPUID(mtxid)の_MAX_MTX)<GET_LOCALID(mtxid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(mtxid のミューテックスが存在しない)

■機能説明

mtxid で示されたミューテックスを削除します。

mtxid には、自 CPU のカーネルに属するミューテックスのみ指定できます。

mtxid で示されたミューテックスにロック待ちタスクがあった場合でもエラーにはなりませんが、待ち状態だったタスクは待ち状態が解除され、エラーコードとして E_DLT が返されます。

対象ミューテックスがロックされていた場合には、それをロックしているタスクのロックを解除します。その結果、そのタスクがロックしているミューテックスがなくなった場合のみ、タスクの現在優先度をベース優先度に戻します。

削除されたミューテックスをロックしているタスクには、ミューテックスが削除されたことは通知されません。後でミューテックスをロック解除しようとした時点でエラーが返されます。

6.18.3 ミューテックスのロック(loc_mtx, ploc_mtx, tloc_mtx)

■C 言語 API

```
ER ercd = loc_mtx(ID mtxid);
ER ercd = ploc_mtx(ID mtxid);
ER ercd = tloc_mtx(ID mtxid, TMO tmout);
```

■引数

mtxid　　　ミューテックス ID

《tloc_mtx》

tmout　　　タイムアウト指定

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR	[p]	パラメータエラー(tmout≦-2)
E_ID	[p]	不正 ID 番号
		(1)CPUID が不正(GET_CPUID(mtxid)が自 CPU でない)
		(2)ローカル ID 範囲外
		(GET_LOCALID(mtxid)≦0,
		(GET_CPUID(mtxid)の_MAX_MTX)<GET_LOCALID(mtxid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_ILUSE	[k]	サービスコール不正使用
		(1)呼び出しタスクは既に mtxid のミューテックスをロック済み
		(2)呼び出しタスクのベース優先度 > 対象ミューテックスの上限優先度
E_NOEXS	[k]	未登録(mtxid のミューテックスが存在しない)
E_DLT	[k]	待ちオブジェクト削除(mtxid のミューテックスが削除された)
E_RLWAI	[k]	待ち状態強制解除(待ちの間に rel_wai サービスコールが呼び出された)
E_TMOUT	[k]	ポーリング失敗、またはタイムアウト

■機能説明

mtxid で指定されるミューテックスをロックします。

対象ミューテックスがロックされていない場合には、自タスクがミューテックスをロックした状態にして、サービスコールの処理を終了します。その際、自タスクの現在優先度はミューテックスの上限優先度まで引き上げられます。

対象ミューテックスがロックされている場合には、自タスクを待ち行列につなぎ、ミューテックスのロック待ち状態に移行させます。待ち行列は、優先度順に管理されます。

tloc_mtx サービスコールの場合、tmout には待ち時間を指定します。

tmout に正の値を指定した場合、待ち解除の条件が満たされないまま tmout 時間が経過すると、エラーコードとして E_TMOUT を返します。tmout=TMO_POL（0）を指定した場合、ploc_mtx サービスコールと同じ処理を行います。tmout=TMO_FEVR（-1）を指定した場合、タイムアウト監視を行いません。この場合、loc_mtx サービスコールと同じ動作となります。

tmout に指定可能な最大値は、(0x7FFFFFFF-TIC_NUME)/TIC_DENO に制限されます。これより大きな値を指定した場合の動作は保証されません。

時間の管理方法については、「5.13.7 時間の精度」を参照してください。

6.18.4 ミューテックスのロック解除(unl_mtx)

■C 言語 API

```
ER ercd = unl_mtx(ID mtxid);
```

■引数

mtxid	ミューテックス ID

■リターン値

正常終了 (E_OK) 、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(mtxid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(mtxid)≦0, (GET_CPUID(mtxid)の_MAX_MTX)<GET_LOCALID(mtxid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_ILUSE	[k]	サービスコール不正使用(対象ミューテックスをロックしていない)
E_NOEXS	[k]	未登録(mtxid のミューテックスが存在しない)

■機能説明

mtxid で示されたミューテックスのロックを解除します。対象ミューテックスに対してロックを待っているタスクがあれば、ミューテックスの待ち行列先頭タスクを待ち解除し、待ち解除されたタスクがミューテックスをロックした状態にします。その際、ロックするタスクの現在優先度はミューテックスの上限優先度まで引き上げられます。ミューテックスに対して待っているタスクがなければ、そのミューテックスをロックされていない状態にします。

本カーネルの TA_CEILING 属性は、簡略化した優先度上限プロトコルを採用しています。つまり、本サービスコールによって呼び出しタスクがロックしているミューテックスが全て無くなったときのみ、現在優先度をベース優先度に戻します。呼び出しタスクがまだ他のミューテックスをロックしている場合、本サービスコールでは現在優先度は変化しません。

6.18.5 ミューテックスの状態参照(ref_mtx)

■C 言語 API

```
ER ercd = ref_mtx(ID mtxid, T_RMTX *pk_rmtx);
```

■引数

mtxid	ミューテックス ID
pk_rmtx	ミューテックス状態を返すパケットへのポインタ

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

```
typedef    struct {
           ID      htskid;      ミューテックスをロックしているタスク ID
           ID      wtskid;      ミューテックスの待ち行列の先頭タスク ID
} T_RMTX;
```

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(mtxid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(mtxid)≦0, (GET_CPUID(mtxid)の_MAX_MTX)<GET_LOCALID(mtxid))
E_NOEXS	[k]	未登録(mtxid のミューテックスが存在しない)

■機能説明

mtxid で示されたミューテックスの状態を参照します。

pk_rmtx が指す領域に、ミューテックスをロックしているタスク ID(htskid)、ミューテックスの待ち行列の先頭タスク ID(wtskid)を返します。

htskid および wtskid のビット 14～12 は、必ず対象ミューテックスが属する CPUID(1 または 2)となります。

対象ミューテックスをロックしているタスクが存在しない場合は、htskid には TSK_NONE（0）が返ります。

対象ミューテックスに待ちタスクが無い場合は、wtskid には TSK_NONE（0）が返ります。

6.19 拡張同期･通信(メッセージバッファ)機能

表 6.23にメッセージバッファでサポートしているサービスコール一覧を示します。

表6.23 同期･通信（メッセージバッファ）サービスコール

項番	サービスコール *1	機 能	呼び出し可能な状態 *2						
			T	N	E	D	U	L	C
1	cre_mbf	メッセージバッファの生成	○		○	○	○		
	icre_mbf			○	○	○	○		
2	acre_mbf	メッセージバッファの生成 (ID 番号自動割付け)	○		○	○	○		
	iacre_mbf			○	○	○	○		
3	del_mbf	メッセージバッファの削除	○		○	○	○		
4	snd_mbf [R]	メッセージバッファへの送信	○		○		○		
5	psnd_mbf [R]	同上(ポーリング)	○		○	△	○		
	ipsnd_mbf			○	○	○	○		
6	tsnd_mbf [R]	同上(タイムアウト有)	○		○		○		
7	rcv_mbf [R]	メッセージバッファからの受信	○		○		○		
8	prcv_mbf [R]	同上(ポーリング)	○		○	△	○		
9	trcv_mbf [R]	同上(タイムアウト有)	○		○		○		
10	ref_mbf [R]	メッセージバッファの状態参照	○		○	△	○		
	iref_mbf			○	○	○	○		

【注】 *1 "[S]"はスタンダードプロファイルのサービスコール、"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
“[B]”はベーシックプロファイルのサービスコールです。
“[R]”は、リモート呼び出し可能なサービスコールです。

*2 それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
“C”はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
“○”はその状態から呼び出し可能、”△”は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

表 6.24にメッセージバッファ機能の仕様を示します。

表6.24 メッセージバッファの仕様

項番	項目	内容
1	ローカルメッセージバッファ ID	1～_MAX_MBF (最大 1023)
2	サポート属性	TA_TFIFO : 送信待ちタスクのキューイングは FIFO TA_TPRI : 送信待ちタスクのキューイングは優先度順

6.19.1 メッセージバッファの生成(cre_mbf, icre_mbf) (acre_mbf, iacre_mbf：ID 番号自動割付け)

■C 言語 API

```
ER ercd = cre_mbf(ID mbfid, T_CMBF *pk_cmbf);
ER ercd = icre_mbf(ID mbfid, T_CMBF *pk_cmbf);
ER_ID mbfid = acre_mbf(T_CMBF *pk_cmbf);
ER_ID mbfid = iacre_mbf(T_CMBF *pk_cmbf);
```

■引数

pk_cmbf　　メッセージバッファ生成情報を格納したパケットへのポインタ

《cre_mbf、icre_mbf》

mbfid　　メッセージバッファ ID

■リターン値

《cre_mbf、icre_mbf》

正常終了（E_OK）、またはエラーコード

《acre_mbf、iacre_mbf》

生成したメッセージバッファの ID 番号（正の値）、またはまたはエラーコード

■パケットの構造

```
typedef  struct {
         ATR     mbfatr;     メッセージバッファ属性
         UINT    maxmsz;     メッセージの最大サイズ（バイト数）
         SIZE    mbfsz;      メッセージバッファ領域のサイズ（バイト数）
         VP      mbf;        メッセージバッファ領域の先頭アドレス
} T_CMBF;
```

■エラーコード

E_NOMEM	[k]	メモリ不足(メッセージバッファ領域が確保できない)
E_RSATR	[p]	属性不正(mbfatr が不正)
E_PAR	[p]	パラメータエラー (1)mbfsz が 4 の倍数以外 (2)maxmsz=0 (3)mbfsz が 0 以外で maxmsz+4＞mbfsz
E_ID	[p]	不正 ID 番号(cre_mbf, icre_mbf) (1)CPUID が不正(GET_CPUID(mbfid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(mbfid)≦0, (GET_CPUID(mbfid)の_MAX_MBF<GET_LOCALID(mbfid))
E_OBJ	[k]	オブジェクト状態不正(mbfid のメッセージバッファが存在)(cre_mbf, icre_mbf)
E_NOID	[k]	空き ID なし(acre_mbf, iacre_mbf)

■機能説明

メッセージバッファを生成します。

これらのサービスコールで生成できるのは、自 CPU のカーネルに属するメッセージバッファです。本カーネルでは、他 CPU のカーネルに属するオブジェクトを生成するサービスコールは用意されていません。

cre_mbf, icre_mbf サービスコールは、mbfid で示された ID を持つメッセージバッファを生成します。mbfid のローカル ID には、1～(自 CPU の_MAX_MBF)の値を指定します。mbfid の CPUID には、VCPU_SELF または自 CPUID を指定しなければなりません。

acre_mbf, iacre_mbf サービスコールは、未登録のメッセージバッファ ID を検索してその ID でメッセージバッファを生成し、その ID を mbfid に返します。検索するローカルメッセージバッファ ID 範囲は、1～(自 CPU の_MAX_MBF)となります。リターンされるメッセージバッファ ID の CPUID は、自 CPUID となります。

(1) mbfatr

mbfatr には属性として、メッセージ送信待ちタスクが待ち行列に並ぶ際の並び方を指定します。

mbfatr := (TA_TFIFO || TA_TPRI)

- ◆ TA_TFIFO（0x00000000）送信待ちタスクのキューイングは FIFO
- ◆ TA_TPRI（0x00000001）送信待ちタスクのキューイングは優先度順

メッセージ受信待ちタスクの待ち行列、およびメッセージ行列は、mbfatr に関わらず FIFO（First-In First-Out）となります。

(2) mbfsz

mbfsz には、生成するメッセージバッファのサイズを指定します。

なお、mbfsz に指定すべきサイズの目安を知るために、以下のマクロが用意されています。

SIZE mbfsz = TSZ_MBF(UINT msgcnt, UINT msgsz)
　サイズが msgsz バイトのメッセージを msgcnt 数格納するのに必要なメッセージバッファ領域のサイズ(目安のバイト数)

また、mbfsz=0 でメッセージバッファを生成することもできます。mbfsz=0 で生成したメッセージバッファではバッファにメッセージを蓄えておくことができないため、メッセージ送信側と受信側の先に実行した方が待ち状態になり、他方が行われた時点で待ちが解除される、つまりメッセージ送信側と受信側が完全に同期した動作となります。また、mbfsz=0 のメッセージバッファでは、メッセージバッファを介したコピー動作を伴わないというメリットもあります。

(3) maxmsz

maxmsz には、生成するメッセージバッファで扱うことのできるメッセージの最大長を指定します。

(4) mbf

mbf には、メッセージバッファとして使用する空き領域の先頭アドレスを指定します。mbf から mbfsz バイトをメッセージバッファとして使用します。

mbfsz=0 の場合、mbf は意味を持たず単に無視されます。

mbf に NULL を指定すると、カーネルはデフォルトメッセージバッファ用領域から mbfsz バイトのメッセージバッファ領域を割り付けます。これにより、デフォルトメッセージバッファ用領域の空きは以下の式で計算されるサイズだけ減少します。

- ◆ 減少サイズ ＝ mbfsz ＋ 16

6.19.2 メッセージバッファの削除(del_mbf)

■C 言語 API

```
ER ercd = del_mbf(ID mbfid);
```

■引数

mbfid　　　メッセージバッファ ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(mbfid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(mbfid)≦0, (GET_CPUID(mbfid)の_MAX_MBF<GET_LOCALID(mbfid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(mbfid のメッセージバッファが存在しない)

■機能説明

mbfid で示されたメッセージバッファを削除します。

mbfid には、自 CPU のカーネルに属するメッセージバッファのみ指定できます。

mbfid で示されたメッセージバッファにおいて、メッセージ受信またはメッセージ送信を待っているタスクがあった場合でもエラーにはなりませんが、待ち状態だったタスクは待ち状態が解除され、エラーコードとして E_DLT が返されます。また、メッセージバッファ内にメッセージが格納されていた場合でもエラーにはなりませんが、格納されていたメッセージは全て破棄されます。

削除により、デフォルトメッセージバッファ用領域の空きは以下の式で計算されるサイズだけ増加します。

◆ 増加サイズ ＝ (生成時に指定した mbfsz) ＋ 16

6.19.3 メッセージバッファへの送信(snd_mbf, psnd_mbf, ipsnd_mbf, tsnd_mbf)

■C 言語 API

```
ER ercd = snd_mbf(ID mbfid, VP msg, UINT msgsz);
ER ercd = psnd_mbf(ID mbfid, VP msg, UINT msgsz);
ER ercd = ipsnd_mbf(ID mbfid, VP msg, UINT msgsz);
ER ercd = tsnd_mbf(ID mbfid, VP msg, UINT msgsz, TMO tmout);
```

■引数

mbfid	メッセージバッファ ID
msg	送信メッセージの先頭アドレス
msgsz	送信メッセージのサイズ(バイト数)
《tsnd_mbf》	
tmout	タイムアウト指定

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR	[p]	パラメータエラー(msgsz=0、tmout≦-2)
	[k]	(msgsz＞maxmsz[*1])
E_ID	[p]	不正 ID 番号
		(1)CPUID が不正(GET_CPUID(mbfid)が不正)
		(2)ローカル ID 範囲外
		(GET_LOCALID(mbfid)≦0,
		(GET_CPUID(mbfid)の_MAX_MBF)<GET_LOCALID(mbfid))
E_CTX	[k]	コンテキストエラー
		(GET_CPUID(mbfid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(mbfid のメッセージバッファが存在しない)
E_DLT	[k]	待ちオブジェクト削除(mbfid のメッセージバッファが削除された)
E_TMOUT	[k]	ポーリング失敗、またはタイムアウト
E_RLWAI	[k]	待ち状態強制解除(待ちの間に rel_wai サービスコールが呼び出された)

*1 maxmsz：メッセージバッファ生成時に指定したメッセージの最大長

RENESAS

■機能説明

mbfid で示されたメッセージバッファに対して、msg で示されたメッセージを送信します。送信するサイズは msgsz で示されたバイト数です。

対象メッセージバッファに受信待ちタスクが存在する場合には、メッセージバッファには格納せずに受信待ち行列の先頭タスクにメッセージを渡し、そのタスクの待ち状態を解除します。

対象メッセージバッファに既に送信待ちタスクが存在する場合、snd_mbf, tsnd_mbf サービスコールではメッセージバッファの空き領域を待つための待ち行列（送信待ち行列）につながれ、psnd_mbf, ipsnd_mbf サービスコールでは直ちにエラーE_TMOUT で終了します。送信待ち行列は、生成時に指定した属性にしたがって管理されます。

受信待ちタスクも送信待ちタスクも存在しない場合は、メッセージをメッセージバッファに格納します。この結果、メッセージバッファの空きサイズは、以下の式で算出されるサイズだけ減少します。

◆ 減少サイズ ＝ msgsz＋4

このサイズだけの空きがメッセージバッファに存在しない場合（バッファサイズが 0 の場合も含む）は、呼び出しタスクは送信待ち行列につながれます。

ipsnd_mbf は、非タスクコンテキストからも発行可能です。対象のメッセージバッファに TA_TPRI 属性が指定されている場合は、非タスクコンテキストはタスクよりも優先順位が高いため、バッファに空きがあれば、先に送信を待っているタスクが存在しても、バッファにメッセージがコピーされます。

tsnd_mbf サービスコールの場合、tmout には待ち時間を指定します。

tmout に正の値を指定した場合、待ち条件が満たされないまま tmout 時間が経過すると、エラーコードとして E_TMOUT を返します。tmout=TMO_POL（0）を指定した場合、psnd_mbf サービスコールと同じ処理を行います。tmout=TMO_FEVR（-1）を指定した場合、タイムアウト監視を行いません。したがって、snd_mbf サービスコールと同じ処理を行います。

tmout に指定可能な最大値は、(0x7FFFFFFF-TIC_NUME)/TIC_DENO に制限されます。これより大きな値を指定した場合の動作は保証されません。

時間の管理方法については、「5.13.7 時間の精度」を参照してください。

snd_mbf, psnd_mbf, および tsnd_mbf では、ディスパッチ保留状態以外の場合のみ、mbfid に他 CPU のカーネルに属するメッセージバッファを指定することができます。この場合、メッセージは非キャッシャブル領域に作成しなければなりません。

一方、ipsnd_mbf では、mbfid に他 CPU のカーネルに属するメッセージバッファを指定することはできません。

6.19.4 メッセージバッファからの受信(rcv_mbf, prcv_mbf, trcv_mbf)

■C 言語 API

```
ER_UINT msgsz = rcv_mbf(ID mbfid, VP msg);
ER_UINT msgsz = prcv_mbf(ID mbfid, VP msg);
ER_UINT msgsz = trcv_mbf(ID mbfid, VP msg, TMO tmout);
```

■引数

mbfid	メッセージバッファ ID
msg	受信メッセージを格納する記憶域へのポインタ
《trcv_mbf》	
tmout	タイムアウト指定

■リターン値

受信メッセージのサイズ（バイト数、正の値）、またはエラーコード

■エラーコード

E_PAR	[p]	パラメータエラー(tmout≦-2)
E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(mbfid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(mbfid)≦0, (GET_CPUID(mbfid)の_MAX_MBF)<GET_LOCALID(mbfid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(mbfid のメッセージバッファが存在しない)
E_DLT	[k]	待ちオブジェクト削除(mbfid のメッセージバッファが削除された)
E_TMOUT	[k]	ポーリング失敗、またはタイムアウト
E_RLWAI	[k]	待ち状態強制解除(待ちの間に rel_wai サービスコールが呼び出された)

RENESAS

■機能説明

mbfid で示されたメッセージバッファからメッセージを受信し、受信したメッセージを msg の指す領域に格納します。また、受信したメッセージサイズをリターンパラメータとして返します。

メッセージバッファにメッセージがあれば、メッセージ行列先頭のメッセージ(最古のメッセージ)を受信します。メッセージバッファ内のメッセージを受信することで、メッセージバッファの空きサイズは以下の式で算出されるサイズだけ増加します。

◆ 増加サイズ ＝ msgsz＋4

この結果、空きサイズがメッセージ送信待ち行列先頭のタスクが送信しようとしていたメッセージサイズよりも大きくなると、そのメッセージがメッセージバッファに格納され、そのタスクの待ち状態が解除されます。送信待ち行列の以降のタスクに対してもメッセージの格納が可能であれば、待ち行列の順に同様の処理を行います。

メッセージバッファにメッセージが存在せず、メッセージ送信待ちタスクが存在する場合、メッセージ送信待ち行列先頭タスクのメッセージを受信します。この結果、そのメッセージ送信待ちタスクの待ち状態は解除されます。

メッセージバッファにメッセージがなく、メッセージ送信待ちタスクも存在しない場合、rcv_mbf, trcv_mbf サービスコールでは、呼び出しタスクはメッセージ到着を待つ待ち行列（受信待ち行列）につながれ、prcv_mbf サービスコールでは直ちにエラーE_TMOUT で終了します。受信待ち行列は、FIFO で管理されます。

msg の指す領域として、cre_mbf, icre_mbf, acre_mbf, iacre_mbf サービスコールで指定した maxmsz 分の RAM 領域が必要です。

trcv_mbf サービスコールの場合、tmout には待ち時間を指定します。

tmout に正の値を指定した場合、待ち解除の条件が満たされないまま tmout 時間が経過すると、エラーコードとして E_TMOUT を返します。tmout=TMO_POL（0）を指定した場合、prcv_mbf サービスコールと同じ処理を行います。tmout=TMO_FEVR（-1）を指定した場合、タイムアウト監視を行いません。したがって、rcv_mbf サービスコールと同じ処理を行います。

tmout に指定可能な最大値は、(0x7FFFFFFF-TIC_NUME)/TIC_DENO に制限されます。これより大きな値を指定した場合の動作は保証されません。

時間の管理方法については、「5.13.7 時間の精度」を参照してください。

これらのサービスコールでは、ディスパッチ保留状態以外の場合のみ、mbfid に他 CPU のカーネルに属するメッセージバッファを指定することができます。この場合、msg の指す領域は非キャッシャブル領域でなければなりません。

6.19.5 メッセージバッファの状態参照(ref_mbf, iref_mbf)

■C 言語 API

```
ER ercd = ref_mbf(ID mbfid, T_RMBF *pk_rmbf);
ER ercd = iref_mbf(ID mbfid, T_RMBF *pk_rmbf);
```

■引数

mbfid	メッセージバッファ ID
pk_rmbf	メッセージバッファ状態を返すパケットへのポインタ

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

```
typedef  struct {
         ID      stskid;     送信待ち行列の先頭タスク ID
         ID      rtskid;     受信待ち行列の先頭タスク ID
         UINT    smsgcnt;    メッセージバッファに入っているメッセージの数
         SIZE    fmbfsz;     空きバッファのサイズ（バイト数）
} T_RMBF;
```

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(mbfid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(mbfid)≦0, (GET_CPUID(mbfid)の_MAX_MBF)<GET_LOCALID(mbfid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(mbfid)が他 CPU で、許可されていない状態からの呼び出し)

■機能説明

mbfid で示されたメッセージバッファの状態を参照し、pk_rmbf が指す領域に送信待ちタスク ID (stskid)、受信待ちタスク ID(rtskid)、メッセージバッファに入っているメッセージの数(smsgcnt)、および空きバッファサイズ(fmbfsz)を返します。

stskid および rtskid のビット 14～12 は、必ず対象メッセージバッファが属する CPUID(1 または 2) となります。

受信待ちタスク、送信待ちタスクが無い場合は、待ちタスク ID として TSK_NONE（0）を返します。

ref_mbf では、ディスパッチ保留状態以外の場合のみ、mbfid に他 CPU のカーネルに属するメッセージバッファを指定することができます。一方、iref_mbf では、mbfid に他 CPU のカーネルに属するメッセージバッファを指定することはできません。

RENESAS

6.20 メモリプール管理(固定長メモリプール)機能

表 6.25に固定長メモリプールでサポートしているサービスコール一覧を示します。

表6.25 メモリプール管理（固定長メモリプール）サービスコール

項番	サービスコール *1		機 能	呼び出し可能な状態 *2						
				T	N	E	D	U	L	C
1	cre_mpf	[s]	固定長メモリプールの生成	○		○	○	○		
	icre_mpf				○	○	○	○		
2	acre_mpf		固定長メモリプールの生成(ID 番号自動割付け)	○		○	○	○		
	iacre_mpf				○	○	○	○		
3	del_mpf		固定長メモリプールの削除	○		○	○	○		
4	get_mpf	[B] [S] [R]	固定長メモリブロックの獲得	○		○		○		
5	pget_mpf	[B] [S] [R]	同上(ポーリング)	○		○	△	○		
	ipget_mpf				○	○	○	○		
6	tget_mpf	[S] [R]	同上(タイムアウト有)	○		○		○		
7	rel_mpf	[B] [S] [R]	固定長メモリブロックの返却	○		○	△	○		
	irel_mpf				○	○	○	○		
8	ref_mpf	[R]	固定長メモリプールの状態参照	○		○	△	○		
	iref_mpf				○	○	○	○		

【注】 *1 "[S]"はスタンダードプロファイルのサービスコール､"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
“[B]”はベーシックプロファイルのサービスコールです。
“[R]”は、リモート呼び出し可能なサービスコールです。

*2 それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
“C”はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
“○”はその状態から呼び出し可能、”△”は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

表 6.26に固定長メモリプールの仕様を示します。

表6.26 固定長メモリプールの仕様

項番	項目	内容
1	ローカル固定長メモリプール ID	1～_MAX_MPF(最大 1023) *
2	サポート属性	TA_TFIFO : 待ちタスクのキューイングは FIFO TA_TPRI : 待ちタスクのキューイングは優先度順
3	管理方式	メモリプール領域内に管理情報を置くかどうかを、cfg ファイルの system.mpfmanage で選択できます。

【注】 _MAX_MPF の最大値は 1023 ですが､cfg ファイルの maxdefine.max_mpf に指定可能な最大値は 1022 です。

6.20.1　固定長メモリプールの生成(cre_mpf, icre_mpf)
(acre_mpf, iacre_mpf：ID 番号自動割付け)

■C 言語 API

```
ER ercd = cre_mpf(ID mpfid, T_CMPF *pk_cmpf);
ER ercd = icre_mpf(ID mpfid, T_CMPF *pk_cmpf);
ER_ID mpfid = acre_mpf(T_CMPF *pk_cmpf);
ER_ID mpfid = iacre_mpf(T_CMPF *pk_cmpf);
```

■引数

pk_cmpf　　固定長メモリプール生成情報を格納したパケットへのポインタ

《cre_mpf、icre_mpf》

mpfid　　固定長メモリプール ID

■リターン値

《cre_mpf、icre_mpf》

正常終了（E_OK）、またはエラーコード

《acre_mpf、iacre_mpf》

生成した固定長メモリプールの ID 番号（正の値）、またはエラーコード

■パケットの構造

（1） system.mpfmanage が IN の場合

```
typedef    struct {
           ATR     mpfatr;     固定長メモリプール属性
           UINT    blkcnt;     メモリプール全体のブロック数
           UINT    blksz;      固定長メモリブロックサイズ（バイト数）
           VP      mpf;        固定長メモリプール領域の先頭アドレス
} T_CMPF;
```

（2） system.mpfmanage が OUT の場合

```
typedef    struct {
           ATR     mpfatr;     固定長メモリプール属性
           UINT    blkcnt;     メモリプール全体のブロック数
           UINT    blksz;      固定長メモリブロックサイズ（バイト数）
           VP      mpf;        固定長メモリプール領域の先頭アドレス
           VP      mpfmb;      固定長メモリブロック管理領域の先頭アドレス
} T_CMPF;
```


RENESAS

■エラーコード

E_NOMEM	[k]	メモリ不足(メモリプール用の領域が確保できない)
E_RSATR	[p]	属性不正(mpfatr が不正)
E_PAR	[p]	パラメータエラー(blkcnt=0、blksz が 4 の倍数以外、blksz=0)
	[k]	TSZ_MPF(blkcnt, blksz)が 32 ビット範囲を超えている
E_ID	[p]	不正 ID 番号(cre_mpf, icre_mpf)
		(1)CPUID が不正(GET_CPUID(mpfid)が自 CPU でない)
		(2)ローカル ID 範囲外
		(GET_LOCALID(mpfid)≦0,
		(GET_CPUID(mpfid)の_MAX_MPF<GET_LOCALID(mpfid))
E_OBJ	[k]	オブジェクト状態不正(mpfid の固定長メモリプールが存在)(cre_mpf, icre_mpf)
E_NOID	[k]	空き ID なし(acre_mpf, iacre_mpf)

■機能説明

固定長メモリプールを生成します。

これらのサービスコールで生成できるのは、自 CPU のカーネルに属する固定長メモリプールです。本カーネルでは、他 CPU のカーネルに属するオブジェクトを生成するサービスコールは用意されていません。

cre_mpf, icre_mpf サービスコールは、mpfid で示された ID を持つ固定長メモリプールを生成します。mpfid のローカル ID には、1～(自 CPU の_MAX_MPF)の値を指定します。mpfid の CPUID には、VCPU_SELF または自 CPUID を指定しなければなりません。

acre_mpf, iacre_mpf サービスコールは、未登録の固定長メモリプール ID を検索してその ID で固定長メモリプールを生成し、その ID を mpfid に返します。検索するローカル固定長メモリプール ID 範囲は、1～(自 CPU の_MAX_MPF)となります。リターンされる固定長メモリプール ID の CPUID は、自 CPUID となります。

mpfatr には属性として、メモリブロック獲得を待つ待ち行列に並ぶ際の並び方を指定します。

mpfatr := (TA_TFIFO ‖ TA_TPRI)

◆ TA_TFIFO（0x00000000）メモリブロック獲得待ちタスクのキューイングはFIFO

◆ TA_TPRI（0x00000001）メモリブロック獲得待ちタスクのキューイングは優先度順

blkcnt には、生成するメモリプールの総ブロック数を指定します。

blksz には、メモリブロックのサイズを指定します。指定するサイズは 4 の倍数でなければなりません。

mpf に NULL を指定した場合は、カーネルが自動的に固定長メモリプールを確保します。その際、固定長メモリプールは、コンフィギュレータで指定したデフォルト固定長メモリプール用領域から割り付けられます。生成に成功すると、デフォルト固定長メモリプール用領域の空きは以下の式で計算されるサイズだけ減少します。

減少サイズ＝TSZ_MPF(blkcnt, blksz) + 16

TSZ_MPF()は、固定長メモリプールに必要なサイズを算出するためのマクロです。

SIZE　TSZ_MPF(UINT blkcnt, UINT blksz)
サイズが blksz バイトのメモリブロックを blkcnt 個獲得可能な固定長メモリプールのサイズ

なお、TSZ_MPF()マクロの定義内容は、system.mpfmanage の設定によって以下のように異なります。

(1) system.mpfmanageがINの場合
TSZ_MPF(blkcnt, blksz) ＝ (blksz+4)×blkcnt

(2) system.mpfmanageがOUTの場合
TSZ_MPF(blkcnt, blksz) ＝ blksz×blkcnt

mpf に生成する固定長メモリプールのアドレスを指定することもできます。この場合、固定長メモリプールとして TSZ_MPF(blkcnt, blksz)で算出されるサイズの領域を確保し、そのアドレスを mpf に指定してください。

system.mpfmanage が OUT の場合は、mpfmb に固定長メモリブロック管理領域の先頭アドレスを指定しなければなりません。具体的には、VTSZ_MPFMB(blkcnt, blksz)で算出されるサイズの領域を確保し、そのアドレスを mpfmb に指定しなければなりません。

mpfmb は、μITRON4.0 仕様の範囲外のメンバです。

固定長メモリプールから獲得したメモリブロックを他 CPU からアクセスする可能性がある場合、固定長メモリプール領域は非キャッシャブル領域でなければなりません。特に、デフォルト固定長メモリプール領域から割り付けられた固定長メモリプールから獲得するメモリブロックを他 CPU からアクセスする可能性がある場合は、リンク時にデフォルト固定長メモリプール用領域のセクション(BC_himpf)を非キャッシャブル領域に配置する必要があります。

6.20.2 固定長メモリプールの削除(del_mpf)

■C 言語 API

```
ER ercd = del_mpf(ID mpfid);
```

■引数

mpfid　　固定長メモリプール ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(mpfid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(mpfid)≦0, (GET_CPUID(mpfid)の_MAX_MPF<GET_LOCALID(mpfid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(mpfid の固定長メモリプールが存在しない)

■機能説明

mpfid で示された固定長メモリプールを削除します。

mpfid には、自 CPU のカーネルに属する固定長メモリプールのみ指定できます。

mpfid で示された固定長メモリプールにおいて、メモリ獲得を待っているタスクがあった場合でもエラーにはなりませんが、待ち状態だったタスクは待ち状態が解除され、エラーコードとして E_DLT が返されます。

デフォルト固定長メモリプール用領域から割り付けられた固定長メモリプール(生成時に mpf に NULL を指定)を削除すると、デフォルト固定長メモリプール用領域の空きは以下の式で計算されるサイズだけ増加します。

増加サイズ＝TSZ_MPF(生成時に指定した blkcnt，生成時に指定した blksz) + 16

なお、すでに獲得済みのブロックがあっても、カーネルはそれに関して何も処理しません。

6.20.3 固定長メモリブロックの獲得(get_mpf, pget_mpf, ipget_mpf, tget_mpf)

■C 言語 API

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

■引数

mpfid	固定長メモリプール ID
p_blk	メモリブロック先頭アドレスを返す記憶域へのポインタ
《tget_mpf》	
tmout	タイムアウト指定

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR	[p]	パラメータエラー(tmout≦-2)
E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(mpfid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(mpfid)≦0, (GET_CPUID(mpfid)の_MAX_MPF<GET_LOCALID(mpfid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(mpfid の固定長メモリプールが存在しない)
E_DLT	[k]	待ちオブジェクト削除(mpfid の固定長メモリプールが削除された)
E_TMOUT	[k]	ポーリング失敗、またはタイムアウト
E_RLWAI	[k]	待ち状態強制解除(待ちの間に rel_wai サービスコールが呼び出された)

■機能説明

mpfid で示される固定長メモリプールからひとつの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 p_blk の指す領域に返します。

既にメモリブロック獲得待ちタスクが存在する場合、または待ちタスクは存在しないが対象となる固定長メモリプールに空きブロックが存在しない場合は、get_mpf, tget_mpf サービスコールでは呼び出しタスクはそのメモリプールのメモリ獲得の待ち行列につながれ、pget_mpf, ipget_mpf サービスコールでは直ちにエラーE_TMOUT で終了します。待ち行列は、生成時に指定した属性にしたがって管理されます。

tget_mpf サービスコールの場合、tmout には待ち時間を指定します。

tmout に正の値を指定した場合、待ち解除の条件が満たされないまま tmout 時間が経過すると、エラーコードとして E_TMOUT を返します。tmout=TMO_POL（0）を指定した場合、pget_mpf サービスコールと同じ処理を行います。tmout=TMO_FEVR（-1）を指定した場合は、タイムアウト監視を行いません。したがって、get_mpf サービスコールと同じ処理を行います。

tmout に指定可能な最大値は、(0x7FFFFFFF-TIC_NUME)/TIC_DENO に制限されます。これより大きな値を指定した場合の動作は保証されません。

時間の管理方法については、「5.13.7 時間の精度」を参照してください。

pget_mpf, get_mpf, および tget_mpf では、ディスパッチ保留状態以外の場合のみ、mpfid に他 CPU のカーネルに属する固定長メモリプールを指定することができます。一方、ipget_mpf では、mpfid に他 CPU のカーネルに属する固定長メモリプールを指定することはできません。

6.20.4 固定長メモリブロックの返却(rel_mpf, irel_mpf)

■C 言語 API

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

■引数

mpfid	固定長メモリプール ID
blk	メモリブロックの先頭アドレス

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR	[p]	パラメータエラー(blk が 4 の倍数以外)
	[k]	(メモリブロックの先頭アドレス以外、またはすでに返却した blk を指定)
E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(mpfid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(mpfid)≦0, (GET_CPUID(mpfid)の_MAX_MPF<GET_LOCALID(mpfid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(mpfid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(mpfid の固定長メモリプールが存在しない)

■機能説明

mpfid で示された固定長メモリプールへ blk で示されたメモリブロックを返却します。

blk には、get_mpf、pget_mpf, ipget_mpf または tget_mpf サービスコールで獲得したメモリブロックの先頭アドレスを指定してください。

対象固定長メモリプールでメモリブロックの獲得を待っているタスクがある場合、本サービスコールで返却したブロックを待ち行列先頭のタスクに割り付け、待ち状態を解除します。

rel_mpf では、ディスパッチ保留状態以外の場合のみ、mpfid に他 CPU のカーネルに属する固定長メモリプールを指定することができます。一方、irel_mpf では、mpfid に他 CPU のカーネルに属する固定長メモリプールを指定することはできません。

RENESAS

6.20.5 固定長メモリプールの状態参照(ref_mpf, iref_mpf)

■C 言語 API

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

■引数

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

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

```
typedef    struct {
           ID       wtskid;     待ちタスク ID
           UINT     fblkcnt;    空き領域のブロック数
} T_RMPF;
```

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(mpfid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(mpfid)≦0, (GET_CPUID(mpfid)の_MAX_MPF<GET_LOCALID(mpfid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(mpfid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(mpfid の固定長メモリプールが存在しない)

■機能説明

mpfid で示された固定長メモリプールの状態を参照します。

pk_rmpf の指す領域に待ちタスク ID(wtskid)、空き領域のブロック数(fblkcnt)を返します。

待ちタスク ID のビット 14～12 は、必ず対象固定長メモリプールが属する CPUID(1 または 2)となります。

他 CPU のタスクが対象固定長メモリプールで待つ場合は、実際にはそのタスクの代わりに対象固定長メモリプールと同じ CPU に属する SVC サーバタスクが対象固定長メモリプールで待つので、待ちタスク ID にこの SVC サーバタスク ID が返ることがあります。

対象メモリプールの待ちタスクが無い場合は、待ちタスク ID として TSK_NONE (0) を返します。

ref_mpf では、ディスパッチ保留状態以外の場合のみ、mpfid に他 CPU のカーネルに属する固定長メモリプールを指定することができます。一方、iref_mpf では、mpfid に他 CPU のカーネルに属する固定長メモリプールを指定することはできません。

6.21 メモリプール管理(可変長メモリプール)機能

表 6.27に可変長メモリプールでサポートしているサービスコール一覧を示します。

表6.27 メモリプール管理（可変長メモリプール）サービスコール

項番	サービスコール *1	機 能	呼び出し可能なシステム状態 *2						
			T	N	E	D	U	L	C
1	cre_mpl	可変長メモリプールの生成	○		○	○	○		
	icre_mpl			○	○	○	○		
2	acre_mpl	可変長メモリプールの生成	○		○	○	○		
	iacre_mpl	(ID 番号自動割付け)		○	○	○	○		
3	del_mpl	可変長メモリプールの削除	○		○	○	○		
4	get_mpl [R]	可変長メモリブロックの獲得	○		○		○		
5	pget_mpl [R]	同上(ポーリング)	○		○	△	○		
	ipget_mpl			○	○	○	○		
6	tget_mpl [R]	同上(タイムアウト有)	○		○		○		
7	rel_mpl [R]	可変長メモリブロックの返却	○		○	△	○		
	irel_mpl			○	○	○	○		
8	ref_mpl [R]	可変長メモリプールの状態参照	○		○	△	○		
	iref_mpl			○	○	○	○		

【注】 *1 "[S]"はスタンダードプロファイルのサービスコール、"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
“[B]”はベーシックプロファイルのサービスコールです。
“[R]”は、リモート呼び出し可能なサービスコールです。

*2 それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
“C”はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
“○”はその状態から呼び出し可能、”△”は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

表 6.28に可変長メモリプールの仕様を示します。

表6.28 可変長メモリプールの仕様

項番	項目	内容
1	ローカル可変長メモリプール ID	1～_MAX_MPL(最大 1023)
2	管理方式	cfg ファイルで、system.newmpl に NEW を指定すると、以下が改善されます。 (1) 特に多くのメモリブロックを扱うメモリプールで、獲得/返却処理が高速化されます。 (2) 空き領域の断片化が軽減されます。 (3) さらに空き領域の断片化を軽減する VTA_UNFRAGMENT 属性を使用できます。
3	サポート属性	・TA_TFIFO : 待ちタスクのキューイングは FIFO ・VTA_UNFRAGMENT：セクタ管理方式

可変長メモリプールでは、空き領域が断片化する問題があります。「5.12.1 空き領域の断片化とその対策」も参照してください。

6.21.1 可変長メモリプールの生成(cre_mpl, icre_mpl)
(acre_mpl, iacre_mpl：ID 番号自動割付け)

■C 言語 API

```
ER ercd = cre_mpl(ID mplid, T_CMPL *pk_cmpl);
ER ercd = icre_mpl(ID mplid, T_CMPL *pk_cmpl);
ER_ID mplid = acre_mpl(T_CMPL *pk_cmpl);
ER_ID mplid = iacre_mpl(T_CMPL *pk_cmpl);
```

■引数

pk_cmpl　　可変長メモリプール生成情報を格納したパケットへのポインタ

《cre_mpl、icre_mpl》

mplid　　可変長メモリプール ID

■リターン値

《cre_mpl、icre_mpl》

正常終了（E_OK）、またはエラーコード

《acre_mpl、iacre_mpl》

生成した可変長メモリプールの ID 番号（正の値）、またはエラーコード

■パケットの構造

(1) system.newmpl が PAST の場合

```
typedef    struct {
           ATR     mplatr;     可変長メモリプール属性
           SIZE    mplsz;      メモリプール全体のサイズ（バイト数）
           VP      mpl;        可変長メモリプール領域の先頭アドレス
}T_CMPL;
```

(2) system.newmpl が NEW の場合

```
typedef    struct {
           ATR     mplatr;     可変長メモリプール属性
           SIZE    mplsz;      メモリプール全体のサイズ（バイト数）
           VP      mpl;        可変長メモリプール領域の先頭アドレス
           VP      mplmb;      可変長メモリプール管理領域の先頭アドレス
           UINT    minblksz;   最小ブロックサイズ
           UINT    sctnum;     最大セクタ数
}T_CMPL;
```

■エラーコード

E_NOMEM	[k]	メモリ不足(メモリプール用の領域が確保できない)
E_RSATR	[p]	属性不正(mplatr が不正)
E_PAR	[p]	パラメータエラー
		(1)mplsz が 4 の倍数以外、
		(2)mplsz＜TSZ_MPL(1,4)
		(3)mplsz≧0x80000000、
		(4)VTA_UNFRAGMENT 属性の場合で、minblksz が 0
		(5)VTA_UNFRAGMENT 属性の場合で、sctnum=0
		(6)VTA_UNFRAGMENT 属性の場合で、mplsz<minblksz*32
E_ID	[p]	不正 ID 番号(cre_mpl, icre_mpl)
		(1)CPUID が不正(GET_CPUID(mplid)が自 CPU でない)
		(2)ローカル ID 範囲外
		(GET_LOCALID(mplid)≦0,
		(GET_CPUID(mplid)の_MAX_MPL<GET_LOCALID(mplid))
E_OBJ	[k]	オブジェクト状態不正(mplid の可変長メモリプールが存在)(cre_mpl,icre_mpl)
E_NOID	[k]	空き ID なし(acre_mpl, iacre_mpl)

■機能説明

可変長メモリプールを生成します。

これらのサービスコールで生成できるのは、自 CPU のカーネルに属する可変長メモリプールです。本カーネルでは、他 CPU のカーネルに属するオブジェクトを生成するサービスコールは用意されていません。

cre_mpl, icre_mpl サービスコールは、mplid で示された ID を持つ可変長メモリプールを生成します。mplid のローカル ID には、1～(自 CPU の_MAX_MPL)の値を指定します。mplid の CPUID には、VCPU_SELF または自 CPUID を指定しなければなりません。

acre_mpl, iacre_mpl サービスコールは、未登録の可変長メモリプール ID を検索してその ID で可変長メモリプールを生成し、その ID を mplid に返します。検索するローカル可変長メモリプール ID 範囲は、1～(自 CPU の_MAX_MPL)となります。リターンされる可変長メモリプール ID の CPUID は、自 CPUID となります。

(1) mplatr

mplatr には、以下の論理和を指定してください。

(a) メモリブロック獲得を待つ待ち行列に並ぶ際の並び方

TA_TFIFO のみを指定できます。

◆ TA_TFIFO(0x00000000) : メモリ獲得待ちタスクのキューイングは FIFO 順

(b) 管理方式

cfg ファイルで、system.newmpl に NEW を指定していた場合は、VTA_UNFRAGMENT を指定できます。

◆ VTA_UNFRAGMENT(0x80000000)：セクタ管理方式(空き領域の断片化が発生しにくい方式)

VTA_UNFRAGMENT 属性は、微小なメモリブロックを大量に獲得するメモリプールに適した属性で、微小なブロックをできるだけ連続して配置することで、大きなサイズの連続空き領域が維持されやすくします。

VTA_UNFRAGMENT 属性を指定した場合のみ、mplmb, minblksz, sctnum が有効になります。sctnum に mplsz/(minblksz×32)よりも大きい値を指定した場合は、mplsz/(minblksz×32)として扱います。

詳細は、「5.12.1 空き領域の断片化とその対策」を参照してください。

(2) mplsz

mplsz には、生成する可変長メモリプール領域のサイズを指定します。「5.12.2 可変長メモリプールの管理方法」も参照してください。

なお、mplsz に指定すべきサイズの目安を知るために、以下のマクロが用意されています。

SIZE mplsz = TSZ_MPL(UINT blkcnt, UINT blksz)

サイズが blksz バイトのメモリブロックを blkcnt 個獲得するのに必要な可変長メモリプール領域のサイズ(目安のバイト数)

このマクロは、VTA_UNFRAGMENT 属性の指定が無い前提で計算します。また、system.newmpl の設定によって計算式が異なります。

(3) mpl

mpl には、可変長メモリプールとして使用する空き領域の先頭アドレスを指定します。mpl から mplsz バイトを可変長メモリプールとして使用します。

mpl に NULL を指定すると、カーネルはデフォルト可変長メモリプール用領域から mplsz バイトのメモリプール領域を割り付けます。これにより、デフォルト可変長メモリプール用領域の空きは以下の式で計算されるサイズだけ減少します。

◆ 減少サイズ ＝ mplsz ＋ 16

可変長メモリプールから獲得したメモリブロックを他 CPU からアクセスする可能性がある場合、可変長メモリプール領域は非キャッシャブル領域でなければなりません。特に、デフォルト可変長メモリプール領域から割り付けられた可変長メモリプールから獲得するメモリブロックを他 CPU からアクセスする可能性がある場合は、リンク時にデフォルト可変長メモリプール用領域のセクション(BC_himpl)を非キャッシャブル領域に配置する必要があります。

(4) mplmb

これはμITRON4.0 仕様外のメンバです。

mplmb は、VTA_UNFRAGMENT 属性を指定した場合のみ有効です。その他の場合は単に無視されます。

以下のマクロで算出されるサイズの領域を確保し、その先頭アドレスを mplmb に指定してください。

VTSZ_MPLMB(最大セクタ数)

(5) minblksz と sctnum

これらはμITRON4.0 仕様外のメンバです。

これらは VTA_UNFRAGMENT 属性を指定した場合のみ有効です。詳細は、「5.12.2 可変長メモリプールの管理方法」を参照してください。

■補足

通常、獲得するメモリブロックのアドレスのアライメントは 4 です。

メモリブロックのアドレスを、キャッシュラインサイズ(16 または 32)のアドレスにアライメントするには、以下のようにしてください。ここでは、アライメント数を N と表記します。

(1) system.newmpl が NEW、VTA_UNFRAGMENT 属性なし

- アプリケーション側で、N バイト境界アドレスにメモリプール領域を確保し、メモリプール生成時にそのアドレスを指定してください。
- 獲得するメモリブロックのサイズは、全て N の倍数としてください。

(2) system.newmpl が NEW、VTA_UNFRAGMENT 属性あり

- アプリケーション側で、N バイト境界アドレスにメモリプール領域を確保し、メモリプール生成時にそのアドレスを指定してください。
- 最小ブロックサイズは、N としてください。
- 獲得するメモリブロックのサイズは、全て N の倍数としてください。

(3) system.newmpl が PAST

(a) N=16の場合

- アプリケーション側で、16 バイト境界アドレスにメモリプール領域を確保し、メモリプール生成時にそのアドレスを指定してください。
- 獲得するメモリブロックのサイズは、全て 16 の倍数としてください。

(b) N=32の場合

- アプリケーション側で、「32 バイト境界アドレス-16」のアドレスにメモリプール領域を確保し、メモリプール生成時にそのアドレスを指定してください。
- 獲得するメモリブロックのサイズは、全て「(N の倍数)+16」としてください。

6.21.2 可変長メモリプールの削除(del_mpl)

■C 言語 API

```
ER ercd = del_mpl(ID mplid);
```

■引数

mplid　　　可変長メモリプール ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(mplid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(mplid)≦0, (GET_CPUID(mplid)の_MAX_MPL<GET_LOCALID(mplid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(mplid の可変長メモリプールが存在しない)

■機能説明

mplid で示された可変長メモリプールを削除します。

mplid には、自 CPU のカーネルに属する可変長メモリプールのみ指定できます。

mplid で示された可変長メモリプールにおいて、メモリ獲得を待っているタスクがあった場合でもエラーにはなりませんが、待ち状態だったタスクは待ち状態が解除され、エラーコードとして E_DLT が返されます。

デフォルト可変長メモリプール用領域から割り付けられた可変長メモリプール(生成時に mpl に NULL を指定)を削除すると、デフォルト可変長メモリプール用領域の空きは以下の式で計算されるサイズだけ増加します。

◆ 増加サイズ ＝ (生成時に指定した mplsz) ＋ 16

なお、すでに獲得済みのブロックがあっても、カーネルはそれに関して何も処理しません。

6.21.3 可変長メモリブロックの獲得(get_mpl, pget_mpl, ipget_mpl, tget_mpl)

■C 言語 API

```
ER ercd = get_mpl(ID mplid, UINT blksz, VP *p_blk);
ER ercd = pget_mpl(ID mplid, UINT blksz, VP *p_blk);
ER ercd = ipget_mpl(ID mplid, UINT blksz, VP *p_blk);
ER ercd = tget_mpl(ID mplid, UINT blksz, VP *p_blk, TMO tmout);
```

■引数

mplid　　可変長メモリプール ID
blksz　　メモリブロックサイズ（バイト数）
p_blk　　メモリブロックの先頭アドレスを返す記憶域へのポインタ
《tget_mpl》
tmout　　タイムアウト指定

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR	[p]	パラメータエラー(blksz が 4 の倍数以外または 0、tmout≦-2)
	[k]	(mplsz[*1]-16＜blksz)
E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(mplid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(mplid)≦0, (GET_CPUID(mplid)の_MAX_MPL<GET_LOCALID(mplid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(mplid の可変長メモリプールが存在しない)
E_DLT	[k]	待ちオブジェクト削除(mplid の可変長メモリプールが削除された)
E_TMOUT	[k]	ポーリング失敗、またはタイムアウト
E_RLWAI	[k]	待ち状態強制解除(待ちの間に rel_wai サービスコールが呼び出された)

*1　**可変長メモリプール生成時に指定したメモリプールサイズ**

RENESAS

■機能説明

mplid で示される可変長メモリプールから、blksz で示されるサイズ（バイト数）の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 p_blk の指す領域に返します。

メモリブロックの獲得により、可変長メモリプールの空きサイズが減少します。詳細は、「5.12.2 可変長メモリプールの管理方法」を参照してください。

既にメモリブロック獲得待ちタスクが存在する場合、または待ちタスクは存在しないが上記サイズの連続した空き領域が存在しない場合は、get_mpl, tget_mpl サービスコールでは呼び出しタスクはそのメモリプールのメモリ獲得の待ち行列につながれ、pget_mpl, ipget_mpl サービスコールでは直ちにエラーE_TMOUT で終了します。待ち行列は、FIFO で管理されます。

tget_mpl サービスコールの場合、tmout には待ち時間を指定します。

tmout に正の値を指定した場合、待ち解除の条件が満たされないまま tmout 時間が経過すると、エラーコードとして E_TMOUT を返します。tmout=TMO_POL（0）を指定した場合、pget_mpl サービスコールと同じ処理を行います。tmout=TMO_FEVR（-1）を指定した場合、タイムアウト監視を行いません。したがって、get_mpl サービスコールと同じ処理を行います。

tmout に指定可能な最大値は、(0x7FFFFFFF-TIC_NUME)/TIC_DENO に制限されます。これより大きな値を指定した場合の動作は保証されません。

時間の管理方法については、「5.13.7 時間の精度」を参照してください。

get_mpl, pget_mpl, tget_mpl では、ディスパッチ保留状態以外の場合のみ、mplid に他 CPU のカーネルに属する可変長メモリプールを指定することができます。一方、ipget_mpl では、mplid に他 CPU のカーネルに属する可変長メモリプールを指定することはできません。

6.21.4 可変長メモリブロックの返却(rel_mpl, irel_mpl)

■C 言語 API

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

■引数

mplid	可変長メモリプール ID
blk	メモリブロックの先頭アドレス

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR	[k]	(メモリブロックの先頭アドレス以外、またはすでに返却した blk を指定)
E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(mplid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(mplid)≦0, (GET_CPUID(mplid)の_MAX_MPL<GET_LOCALID(mplid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(mplid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(mplid の可変長メモリプールが存在しない)

■機能説明

mplid で示された可変長メモリプールへ blk で示されたメモリブロックを返却します。

blk には、get_mpl、pget_mpl, ipget_mpl または tget_mpl サービスコールで獲得したメモリブロックの先頭アドレスを指定してください。

メモリブロックの返却により、可変長メモリプールの空きサイズが増加します。詳細は、「5.12.2 可変長メモリプールの管理方法」を参照してください。

この結果、対象可変長メモリプールでメモリブロックの獲得待ち行列の先頭タスクが要求するだけの連続空き領域ができると、そのタスクにメモリブロックを割り付けて待ち状態を解除します。待ち行列の以降のタスクに対してもメモリブロックを割り付け可能であれば、待ち行列の順に同様の処理を行います。

rel_mpl では、ディスパッチ保留状態以外の場合のみ、mplid に他 CPU のカーネルに属する可変長メモリプールを指定することができます。一方、irel_mpl では、mplid に他 CPU のカーネルに属する可変長メモリプールを指定することはできません。

6.21.5　可変長メモリプールの状態参照(ref_mpl, iref_mpl)

■C 言語 API

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

■引数

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

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

```
typedef   struct {
          ID      wtskid;     待ちタスク ID
          SIZE    fmplsz;     空き領域の合計サイズ（バイト数）
          UINT    fblksz;     獲得可能な最大メモリブロックサイズ（バイト数）
} T_RMPL;
```

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(mplid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(mplid)≦0, (GET_CPUID(mplid)の_MAX_MPL<GET_LOCALID(mplid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(mplid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(mplid の可変長メモリプールが存在しない)

■機能説明

mplid で示された可変長メモリプールの状態を参照します。

pk_rmpl が指す領域に待ちタスク ID(wtskid)、現在の空き領域の合計サイズ(fmplsz)、獲得可能な最大メモリブロックのサイズ(fblksz)を返します。

待ちタスク ID のビット 14～12 は、必ず対象可変長メモリプールが属する CPUID(1 または 2)となります。

他 CPU のタスクが対象可変長メモリプールで待つ場合は、実際にはそのタスクの代わりに対象可変長メモリプールと同じ CPU に属する SVC サーバタスクが対象可変長メモリプールで待つので、待ちタスク ID にこの SVC サーバタスク ID が返ることがあります。

対象メモリプールの待ちタスクが無い場合は、待ちタスク ID として TSK_NONE (0) を返します。

通常空き領域は分断されており、fblksz には分断されている空き領域の中で最大の連続サイズが返ります。1 回の get_mpl、pget_mpl, ipget_mpl または tget_mpl サービスコールで、fblksz までのブロックを即座に獲得できます。

ref_mpl では、ディスパッチ保留状態以外の場合のみ、mplid に他 CPU のカーネルに属する可変長メモリプールを指定することができます。一方、iref_mpl では、mplid に他 CPU のカーネルに属する可変長メモリプールを指定することはできません。

6.22 時間管理機能(システム時刻管理)

表 6.29にシステム時刻管理でサポートしているサービスコール一覧を示します。

表6.29 システム時刻管理サービスコール

項番	サービスコール *1	機 能	呼び出し可能なシステム状態 *2						
			T	N	E	D	U	L	C
1	set_tim [S]	システム時刻の設定	○		○	○	○		
	iset_tim			○	○	○	○		
2	get_tim [S]	システム時刻の参照	○		○	○	○		
	iget_tim			○	○	○	○		
3	isig_tim [S]	タイムティックの供給	(clock.timer に TIMER を指定することで、自動的に実行されるようになります)						
4	vstp_tmr	タイマの停止	○		○	○	○		
5	vrst_tmr	タイマの再開	○		○	○	○		
	ivrst_tmr			○	○	○	○		
6	vsns_tmr	タイマ状態の参照	○	○	○	○	○	○	○

【注】 *1 "[S]"はスタンダードプロファイルのサービスコール、"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
“[B]”はベーシックプロファイルのサービスコールです。
“[R]”は、リモート呼び出し可能なサービスコールです。

*2 それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
“C”はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
“○”はその状態から呼び出し可能、”△”は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

表 6.30にシステム時刻管理の仕様を示します。

表6.30 システム時刻管理の仕様

項番	項目	内容
1	システム時刻値	符号なし 48 ビット
2	システム時刻の単位	1[ms]
3	システム時刻の更新周期	TIC_NUME/TIC_DENO[ms]
4	システム時刻初期値(初期起動時)	0x000000000000

RENESAS

システム時刻は SYSTIM 型の構造体によって符号無し 48bit 整数として表現されますが、その最大値は以下のようになります。

[TIC_NUME/TIC_DENO ≦ 1 の場合]

最大値＝0x7FFFFFFFFFFF/TIC_DENO

[TIC_NUME/TIC_DENO ＞ 1 の場合]

最大値＝0x7FFFFFFFFFFF

タイマ割込み(isig_tim)によってシステム時刻を更新する際に上記最大値を超える場合には、システム時刻は 0 に戻ります。

また、set_tim, iset_tim サービスコールで、上記最大値を超える値を指定した場合の動作は保証されません。

6.22.1 システム時刻の設定(set_tim, iset_tim)

■C 言語 API

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

■引数

p_systim　設定するシステム時刻を示すパケットへのポインタ

■リターン値

正常終了（E_OK）

■パケットの構造

```
typedef    struct {
           UH      utime;       システムの現在時刻（上位）
           UW      ltime;       システムの現在時刻（下位）
} SYSTIM;
```

■機能説明

システムが保持しているシステム時刻の現在の値を、p_systim で示される値に設定します。

TIC_DENO(タイムティック周期時間の分母)が 1 より大きな場合は、指定可能な最大値は 0x7FFFFFFFFFFF/TIC_DENO となります。これより大きな値を指定した場合の動作は保証されません。

時間の管理方法については、「5.13.7 時間の精度」を参照してください。

6.22.2 システム時刻の参照(get_tim, iget_tim)

■C 言語 API

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

■引数

p_systim　システム時刻を返すパケットへのポインタ

■リターン値

正常終了（E_OK）

■パケットの構造

```
typedef    struct {
           UH      utime;      システムの現在時刻（上位）
           UW      ltime;      システムの現在時刻（下位）
} SYSTIM;
```

■機能説明

システム時刻の現在値を読み出し、その結果を p_systim の指す領域に返します。

6.22.3 タイムティックの供給(isig_tim)

■機能説明

システム時刻を更新します。

cfg ファイルで clock.timer に TIMER を指定すると、TIC_NUME/TIC_DENO[ms]で計算される周期で、自動的に isig_tim サービスコール処理が実行されるようにコンフィギュレーションされます。つまり、本機能はサービスコールではありませんので、アプリケーションから呼び出すことはできません。

タイムティックの供給時には、カーネルは時間に関する次のような処理を行います。

(1) タイマドライバの割込み処理関数(tdr_int_tmr())の呼び出し
(2) システム時刻の更新（+1）
(3) プロファイルカウンタの更新
(4) タイムイベントハンドラの起動
(5) tslp_tskサービスコールなどのタイムアウト付きサービスコールで待ち状態になっているタスクのタイムアウト処理

カーネルの時間に関する機能を使用するには、タイマドライバの組み込みが必要です。詳細は、「12.9 タイマドライバ」を参照してください。

6.22.4 タイマの停止(vstp_tmr)

■C 言語 API

```
ER ercd = vstp_tmr(RELTIM limit);
```

■引数

limit	上限時間

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR	[p]	パラメータエラー (1) 0x80000000 ≦ limit ≦ 0xFFFFFFFE
E_OBJ	[k]	オブジェクト状態不正 (1)直近のタイマイベントまでの時間が limit 未満 (2)既にカーネルタイマは停止済み
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)

■機能説明

カーネルタイマを停止します。

ただし、直近のタイマイベント(txxx_yyy によるタイムアウト、dly_tsk による遅延、周期ハンドラ、アラームハンドラ)までの時間が limit 未満の場合は、エラーE_OBJ を返します。

limit には、1～0x7FFFFFFF を指定できます。

limit に 0xFFFFFFFF を指定するとタイマイベントが存在しない場合のみタイマを停止することができます。

limit に 0 を指定するとタイマイベントに関係なくタイマを停止することができます。

カーネルタイマの停止中は、以下の扱いとなります。

(1) システム時刻は更新されません。
(2) 周期ハンドラやタスクのタイムアウトなどのタイマイベントは発生しません。また、tslp_tskやsta_almなどの新たなタイマイベントを生成させるサービスコールを呼び出した場合は、エラーE_OBJが返ります。
(3) タスクのオーバーラン監視時間のカウントは停止します。
(4) プロファイルカウンタは更新されません。

本サービスコールでは、タイマハードウェアの動作を停止するために、タイマドライバの tdr_stp_tmr()がコールバックされます。

通常、本サービスコールは、CPU をスリープさせるときにタイマ割込みも停止したい場合に利用します。

なお、本サービスコールはμITRON4.0 仕様外の機能です。

6.22.5 タイマの再開(vrst_tmr, ivrst_tmr)

■C 言語 API

```
ER ercd = vrst_tmr(RELTIM eratim);
ER ercd = ivrst_tmr(RELTIM eratim);
```

■引数

eratim　　経過時間

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_OBJ	[k]	オブジェクト状態不正 （1）カーネルタイマは停止していない

■機能説明

カーネルタイマ停止時点から eratim だけ時間が経過した扱いで、カーネルタイマを再開します。
eratim に 0 以外を指定した場合は、タイマイベントが発生する場合があります。
タイマハードウェアの動作を再開するために、タイマドライバの tdr_rst_tmr()がコールバックされます。

なお、本サービスコールはμITRON4.0 仕様外の機能です。

RENESAS

6.22.6 タイマ状態の参照 (vsns_tmr)

■C 言語 API

```
BOOL  state = vsns_tmr(void);
```

■リターン値

カーネルタイマが停止中の場合に TRUE、停止中でない場合に FALSE を返します。

■機能説明

カーネルタイマが停止しているかどうかを調べます。
本サービスコールは、CPU ロック状態およびノーマル CPU 例外ハンドラからも呼び出せます。

なお、本サービスコールはμITRON4.0 仕様外の機能です。

6.23 時間管理機能(周期ハンドラ)

表 6.31に周期ハンドラでサポートしているサービスコール一覧を示します。

表6.31 周期ハンドラサービスコール

項番	サービスコール *1	機 能	呼び出し可能なシステム状態 *2						
			T	N	E	D	U	L	C
1	cre_cyc [s]	周期ハンドラの生成	○		○	○	○		
	icre_cyc			○	○	○	○		
2	acre_cyc	周期ハンドラの生成(ID 番号自動割付け)	○		○	○	○		
	iacre_cyc			○	○	○	○		
3	del_cyc	周期ハンドラの削除	○		○	○	○		
4	sta_cyc [B] [S] [R]	周期ハンドラの動作開始	○		○	△	○		
	ista_cyc			○	○	○	○		
5	stp_cyc [B] [S] [R]	周期ハンドラの動作停止	○		○	△	○		
	istp_cyc			○	○	○	○		
6	ref_cyc [R]	周期ハンドラの状態参照	○		○	△	○		
	iref_cyc			○	○	○	○		

【注】 *1 "[S]"はスタンダードプロファイルのサービスコール､"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
“[B]”はベーシックプロファイルのサービスコールです。
“[R]”は、リモート呼び出し可能なサービスコールです。

*2 それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
“C”はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
“○”はその状態から呼び出し可能、”△”は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

表 6.32に周期ハンドラの仕様を示します。

表6.32 周期ハンドラの仕様

項番	項目	内容
1	ローカル周期ハンドラ ID	1～_MAX_CYH (最大 15) *
2	サポート属性	TA_HLNG : 高級言語記述 TA_ASM : アセンブリ言語記述 TA_STA : 周期ハンドラの動作開始 TA_PHS : 起動位相の保存

【注】 _MAX_CYH の最大値は 15 ですが、cfg ファイルの maxdefine.max_cyh に指定可能な最大値は 14 です。

RENESAS

6.23.1 周期ハンドラの生成(cre_cyc, icre_cyc) (acre_cyc, iacre_cyc：ID 番号自動割付け)

■C 言語 API

```
ER ercd = cre_cyc(ID cycid, T_CCYC *pk_ccyc);
ER ercd = icre_cyc(ID cycid, T_CCYC *pk_ccyc);
ER_ID cycid = acre_cyc(T_CCYC *pk_ccyc);
ER_ID cycid = iacre_cyc(T_CCYC *pk_ccyc);
```

■引数

pk_ccyc　周期ハンドラ生成情報を格納したパケットへのポインタ

《cre_cyc、icre_cyc》

cycid　周期ハンドラ ID

■リターン値

《cre_cyc、icre_cyc》

正常終了（E_OK）、またはエラーコード

《acre_cyc、iacre_cyc》

生成した周期ハンドラの ID 番号（正の値）、またはエラーコード

■パケットの構造

```
typedef   struct {
          ATR      cycatr;       周期ハンドラ属性
          VP_INT   exinf;        拡張情報
          FP       cychdr;       周期ハンドラアドレス
          RELTIM   cyctim;       周期ハンドラの起動周期
          RELTIM   cycphs;       周期ハンドラの起動位相
} T_CCYC;
```

■エラーコード

E_RSATR	[p]	属性不正(cycatr が不正)
E_PAR	[p]	パラメータエラー(cyctim=0、cycphs＞cyctim)
E_ID	[p]	不正 ID 番号(cre_cyc, icre_cyc) (1)CPUID が不正(GET_CPUID(cycid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(cycid)≦0, (GET_CPUID(cycid)の_MAX_CYH<GET_LOCALID(cycid))
E_OBJ	[k]	オブジェクト状態不正(cycid の周期ハンドラが存在)（cre_cyc, icre_cyc)
E_NOID	[k]	空き ID なし(acre_cyc, iacre_cyc)

■機能説明

周期ハンドラを生成します。

これらのサービスコールで生成できるのは、自 CPU のカーネルに属する周期ハンドラです。本カーネルでは、他 CPU のカーネルに属するオブジェクトを生成するサービスコールは用意されていません。

cre_cyc, icre_cyc サービスコールは、cycid で示された ID を持つ周期ハンドラを生成します。cycid のローカル ID には、1～(自 CPU の_MAX_CYH)の値を指定します。cycid の CPUID には、VCPU_SELF または自 CPUID を指定しなければなりません。

acre_cyc, iacre_cyc サービスコールは、未登録の周期ハンドラ ID を検索してその ID で周期ハンドラを生成し、その ID を cycid に返します。検索するローカル周期ハンドラ ID 範囲は、1～(自 CPU の_MAX_CYH)となります。リターンされる周期ハンドラ ID の CPUID は、自 CPUID となります。

周期ハンドラは、一定周期で起動される非タスクコンテキストのタイムイベントハンドラです。
cycatr には属性として、ハンドラを記述した言語や起動属性を指定します。

```
cycatr := ((TA_HLNG || TA_ASM) | [TA_STA] | [TA_PHS])
```

- ◆ TA_HLNG（0x00000000）高級言語記述
- ◆ TA_ASM（0x00000001）アセンブラ記述
- ◆ TA_STA（0x00000002）周期ハンドラの動作開始
- ◆ TA_PHS（0x00000004）起動位相の保存

TA_STA が指定された場合には、周期ハンドラを生成した後に、周期ハンドラを動作している状態にします。指定されていない場合は、sta_cyc, ista_cyc サービスコールが呼び出されるまで周期ハンドラは動作しません。TA_PHS が指定された場合は、周期ハンドラの動作を開始する時に、周期ハンドラの起動位相を保存して、次に起動すべき時刻を決定します。指定されていない場合は、sta_cyc, ista_cyc サービスコールが呼び出された時刻を基準として、周期ハンドラを次に起動する時刻を決定します。

exinf は、周期ハンドラを起動するときに、パラメータとして渡す拡張情報を指定します。周期ハンドラに関する情報を設定するなどの目的でユーザが自由に使用できます。

cychdr には、周期ハンドラの先頭アドレスを指定します。

cyctim には、周期ハンドラの起動周期を指定します。

cycphs には、周期ハンドラの起動位相を指定します。

周期ハンドラを生成するサービスコールが呼び出されてから、指定した cycphs(起動位相)以上の時間が経過した時が、周期ハンドラを一回目に起動すべき時刻となります。その後は cyctim(起動周期)が経過する毎に、周期ハンドラが起動されます。

cyctim, cycphs に指定可能な最大値は、(0x7FFFFFFF-TIC_NUME)/TIC_DENO に制限されます。これより大きな値を指定した場合の動作は保証されません。

時間の管理方法については、「5.13.7 時間の精度」を参照してください。

6.23.2　周期ハンドラの削除(del_cyc)

■C 言語 API

```
ER ercd = del_cyc(ID cycid);
```

■引数

cycid	周期ハンドラ ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(cycid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(cycid)≦0, (GET_CPUID(cycid)の_MAX_CYH<GET_LOCALID(cycid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(cycid の周期ハンドラが存在しない)

■機能説明

cycid で示された周期ハンドラを削除します。
cycid には、自 CPU のカーネルに属する周期ハンドラのみ指定できます。

6.23.3 周期ハンドラの動作開始(sta_cyc, ista_cyc)

■C 言語 API

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

■引数

cycid	周期ハンドラ ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(cycid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(cycid)≦0, (GET_CPUID(cycid)の_MAX_CYH<GET_LOCALID(cycid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(cycid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(cycid の周期ハンドラが存在しない)

■機能説明

cycid で示された周期ハンドラを、動作している状態に移行させます。

周期ハンドラ属性に TA_PHS が指定されていない場合には、このサービスコールが呼び出された時刻を基準として、その時刻から起動周期が経過する毎に、周期ハンドラが起動されます。

TA_PHS が指定されていない動作している状態の周期ハンドラが指定された場合は、周期ハンドラを次に起動する時刻の再設定のみを行います。

TA_PHS が指定されている場合は、周期ハンドラ生成時点の時刻を基準に起動するため、時刻の設定は行いません。

sta_cyc では、ディスパッチ保留状態以外の場合のみ、cycid に他 CPU のカーネルに属する周期ハンドラを指定することができます。一方、ista_cyc では、cycid に他 CPU のカーネルに属する周期ハンドラを指定することはできません。

6.23.4 周期ハンドラの動作停止(stp_cyc, istp_cyc)

■C 言語 API

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

■引数

cycid　　周期ハンドラ ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(cycid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(cycid)≦0, (GET_CPUID(cycid)の_MAX_CYH<GET_LOCALID(cycid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(cycid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(cycid の周期ハンドラが存在しない)

■機能説明

cycid で示された周期ハンドラを、動作していない状態に移行させせます。

stp_cyc では、ディスパッチ保留状態以外の場合のみ、cycid に他 CPU のカーネルに属する周期ハンドラを指定することができます。一方、istp_cyc では、cycid に他 CPU のカーネルに属する周期ハンドラを指定することはできません。

6.23.5 周期ハンドラの状態参照(ref_cyc, iref_cyc)

■C 言語 API

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

■引数

cycid	周期ハンドラ ID
pk_rcyc	周期ハンドラの状態を返すパケットへのポインタ

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

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

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(cycid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(cycid)≦0, (GET_CPUID(cycid)の_MAX_CYH<GET_LOCALID(cycid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(cycid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(cycid の周期ハンドラが存在しない)

■機能説明

cycid で示された周期ハンドラの状態を参照し、pk_rcyc が指す領域に周期ハンドラの動作状態(cycstat)、周期ハンドラ起動までの残り時間(lefttim)を返します。

cycstat には、対象周期ハンドラの動作状態を返します。

◆ TCYC_STP（0x00000000）周期ハンドラが動作していない

◆ TCYC_STA（0x00000001）周期ハンドラが動作している

lefttim には、対象周期ハンドラを次に起動する時刻までの相対時間を返します。対象周期ハンドラが動作していない場合、lefttim は不定値となります。

ref_cyc では、ディスパッチ保留状態以外の場合のみ、cycid に他 CPU のカーネルに属する周期ハンドラを指定することができます。一方、iref_cyc では、cycid に他 CPU のカーネルに属する周期ハンドラを指定することはできません。

6.24 時間管理機能(アラームハンドラ)

表 6.33にアラームハンドラでサポートしているサービスコール一覧を示します。

表6.33 アラームハンドラサービスコール

項番	サービスコール *1	機 能	呼び出し可能なシステム状態 *2						
			T	N	E	D	U	L	C
1	cre_alm	アラームハンドラの生成	○		○	○	○		
	icre_alm			○	○	○	○		
2	acre_alm	アラームハンドラの生成 (ID 番号自動割付け)	○		○	○	○		
	iacre_alm			○	○	○	○		
3	del_alm	アラームハンドラの削除	○		○	○	○		
4	sta_alm [R]	アラームハンドラの動作開始	○		○	△	○		
	ista_alm			○	○	○	○		
5	stp_alm [R]	アラームハンドラの動作停止	○		○	△	○		
	istp_alm			○	○	○	○		
6	ref_alm [R]	アラームハンドラの状態参照	○		○	△	○		
	iref_alm			○	○	○	○		

【注】 *1 "[S]"はスタンダードプロファイルのサービスコール、"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
"[B]"はベーシックプロファイルのサービスコールです。
"[R]"は、リモート呼び出し可能なサービスコールです。
*2 それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
"C"はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
"○"はその状態から呼び出し可能、"△"は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

表 6.34にアラームハンドラの仕様を示します。

表6.34 アラームハンドラの仕様

項番	項目	内容
1	ローカルアラームハンドラ指定番号	1～_MAX_ALH (最大 15)
2	サポート属性	TA_HLNG : 高級言語記述 TA_ASM : アセンブリ言語記述

6.24.1 アラームハンドラの生成(cre_alm, icre_alm)
(acre_alm, iacre_alm：ID 番号自動割付け)

■C 言語 API

```
ER ercd = cre_alm(ID almid, T_CALM *pk_calm);
ER ercd = icre_alm(ID almid, T_CALM *pk_calm);
ER_ID almid = acre_alm(T_CALM *pk_calm);
ER_ID almid = iacre_alm(T_CALM *pk_calm);
```

■引数

pk_calm　　アラームハンドラ生成情報を格納したパケットへのポインタ

《cre_alm、icre_alm》

almid　　アラームハンドラ ID

■リターン値

《cre_alm、icre_alm》

正常終了（E_OK）、またはエラーコード

《acre_alm、iacre_alm》

生成したアラームハンドラの ID 番号（正の値）、またはエラーコード

■パケットの構造

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

■エラーコード

E_RSATR	[p]	属性不正(almatr が不正)
E_ID	[p]	不正 ID 番号(cre_alm, icre_alm) (1)CPUID が不正(GET_CPUID(almid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(almid)≦0, (GET_CPUID(almid)の_MAX_ALH<GET_LOCALID(almid))
E_OBJ	[k]	オブジェクト状態不正(almid のアラームハンドラが存在) (cre_alm, icre_alm)
E_NOID	[k]	空き ID なし(acre_alm, iacre_alm)

■機能説明

アラームハンドラを生成します。

これらのサービスコールで生成できるのは、自 CPU のカーネルに属するアラームハンドラです。本カーネルでは、他 CPU のカーネルに属するオブジェクトを生成するサービスコールは用意されていません。

cre_alm, icre_alm c サービスコールは、almid で示された ID を持つアラームハンドラを生成します。almid のローカル ID には、1～(自 CPU の_MAX_ALH)の値を指定します。almid の CPUID には、VCPU_SELF または自 CPUID を指定しなければなりません。

acre_alm, iacre_alm サービスコールは、未登録のアラームハンドラ ID を検索してその ID でアラームハンドラを生成し、その ID を almid に返します。検索するローカルアラームハンドラ ID 範囲は、1～(自 CPU の_MAX_ALH)となります。リターンされるアラームハンドラ ID の CPUID は、自 CPUID となります。

アラームハンドラは、指定した時刻に一度だけ起動される非タスクコンテキストのタイムイベントハンドラです。

almatr には属性として、ハンドラを記述した言語を指定します。

almatr := (TA_HLNG || TA_ASM)

◆ TA_HLNG（0x00000000）高級言語記述

◆ TA_ASM（0x00000001）アセンブラ記述

exinf は、アラームハンドラを起動するときに、パラメータとして渡す拡張情報を指定します。アラームハンドラに関する情報を設定するなどの目的でユーザが自由に使用できます。

almhdr には、アラームハンドラの先頭アドレスを指定します。

アラームハンドラの生成直後は、アラームハンドラの起動時刻は設定されていません。アラームハンドラは停止状態となっています。

6.24.2 アラームハンドラの削除(del_alm)

■C 言語 API

```
ER ercd = del_alm(ID almid);
```

■引数

almid　　アラームハンドラ ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(almid)が自 CPU でない) (2)ローカル ID 範囲外 (GET_LOCALID(almid)≦0, (GET_CPUID(almid)の_MAX_ALH<GET_LOCALID(almid))
E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_NOEXS	[k]	未登録(almid のアラームハンドラが存在しない)

■機能説明

almid で示されたアラームハンドラを削除します。
almid には、自 CPU のカーネルに属するアラームハンドラのみ指定できます。

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

■C 言語 API

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

■引数

almid	アラームハンドラ ID
almtim	アラームハンドラの起動時刻

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(almid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(almid)≦0, (GET_CPUID(almid)の_MAX_ALH<GET_LOCALID(almid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(almid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(almid のアラームハンドラが存在しない)

■機能説明

almid で示されたアラームハンドラの起動時刻を、サービスコールが呼び出された時刻から almtim で指定された相対時間後に設定し、アラームハンドラの動作を開始します。

すでに動作しているアラームハンドラが指定された場合は、以前の起動時刻の設定を解除し、新しい起動時刻を設定します。

almtim に 0 が指定された場合は、次のタイムティックでアラームハンドラが起動されます。

almtim に指定可能な最大値は、(0xFFFFFFFF-TIC_NUME)/TIC_DENO に制限されます。これより大きな値を指定した場合の動作は保証されません。

時間の管理方法については、「5.13.7 時間の精度」を参照してください。

sta_alm では、ディスパッチ保留状態以外の場合のみ、almid に他 CPU のカーネルに属するアラームハンドラを指定することができます。一方、ista_alm では、almid に他 CPU のカーネルに属するアラームハンドラを指定することはできません。

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

■C 言語 API

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

■引数

almid　　アラームハンドラ ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号
		(1)CPUID が不正(GET_CPUID(almid)が不正)
		(2)ローカル ID 範囲外
		(GET_LOCALID(almid)≦0,
		(GET_CPUID(almid)の_MAX_ALH<GET_LOCALID(almid))
E_CTX	[k]	コンテキストエラー
		(GET_CPUID(almid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(almid のアラームハンドラが存在しない)

■機能説明

almid で示されたアラームハンドラの起動時刻の設定を解除し、アラームハンドラの動作を停止します。

stp_alm では、ディスパッチ保留状態以外の場合のみ、almid に他 CPU のカーネルに属するアラームハンドラを指定することができます。一方、istp_alm では、almid に他 CPU のカーネルに属するアラームハンドラを指定することはできません。

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

■C 言語 API

```
ER ercd = ref_alm(ID almid, T_RALM *pk_ralm);
ER ercd = iref_alm(ID almid, T_RALM *pk_ralm);
```

■引数

almid　　アラームハンドラ ID
pk_ralm　アラームハンドラ状態を返すパケットへのポインタ

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

```
typedef    struct {
           STAT     almstat;     アラームハンドラの動作状態
           RELTIM   lefttim;     アラームハンドラ起動までの残り時間
} T_RALM;
```

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(almid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(almid)≦0, (GET_CPUID(almid)の_MAX_ALH<GET_LOCALID(almid))
E_CTX	[k]	コンテキストエラー (GET_CPUID(almid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(almid のアラームハンドラが存在しない)

■機能説明

almid で示されたアラームハンドラの状態を参照し、pk_ralm が指す領域にアラームハンドラの動作状態(almstat)、アラームハンドラ起動までの残り時間(lefttim)を返します。

almstat には、対象アラームハンドラの動作状態を返します。

◆ TALM_STP（0x00000000）アラームハンドラが動作していない
◆ TALM_STA（0x00000001）アラームハンドラが動作している

lefttim には、対象アラームハンドラ起動までの相対時間を返します。対象アラームハンドラが動作していない場合、lefttim は不定値となります。

ref_alm では、ディスパッチ保留状態以外の場合のみ、almid に他 CPU のカーネルに属するアラームハンドラを指定することができます。一方、iref_alm では、almid に他 CPU のカーネルに属するアラームハンドラを指定することはできません。

6.25 時間管理機能(オーバーランハンドラ)

表 6.35にオーバーランハンドラでサポートしているサービスコール一覧を示します。

表6.35 オーバーランハンドラサービスコール

項番	サービスコール *1	機 能	呼び出し可能なシステム状態 *2						
			T	N	E	D	U	L	C
1	def_ovr	オーバーランハンドラの定義	○		○	○	○		
2	sta_ovr [R]	オーバーランハンドラの動作開始	○		○	△	○		
	ista_ovr			○	○	○	○		
3	stp_ovr [R]	オーバーランハンドラの動作停止	○		○	△	○		
	istp_ovr			○	○	○	○		
4	ref_ovr [R]	オーバーランハンドラの状態参照	○		○	△	○		
	iref_ovr			○	○	○	○		

【注】 *1 "[S]"はスタンダードプロファイルのサービスコール、"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
“[B]”はベーシックプロファイルのサービスコールです。
“[R]”は、リモート呼び出し可能なサービスコールです。

*2 それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
“C”はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
“○”はその状態から呼び出し可能、”△”は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

オーバーランハンドラは、システムに 1 つだけ定義できるタイムイベントハンドラです。

タスクが使用したプロセッサ時間には、タスクとタスクが呼び出したサービスコール、そのタスクの実行中に起動された割込みハンドラの各実行時間が含まれます。

タスクが実行状態以外の間は、使用プロセッサ時間はカウントされません。

表 6.36にオーバーランハンドラの仕様を示します。

表6.36 オーバーランハンドラの仕様

項番	項目	内容
1	プロセッサ時間の単位(OVRTIM)	システム時刻と同じ(1[ms])
2	サポート属性	TA_HLNG : 高級言語記述 TA_ASM : アセンブリ言語記述

RENESAS

6.25.1　オーバーランハンドラの定義(def_ovr)

■C 言語 API

```
ER ercd = def_ovr(T_DOVR *pk_dovr);
```

■引数

pk_dovr　　オーバーランハンドラ定義情報を格納したパケットへのポインタ

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

```
typedef    struct {
           ATR     ovratr;      オーバーランハンドラ属性
           FP      ovrhdr;      オーバーランハンドラアドレス
} T_DOVR;
```

■エラーコード

E_RSATR　　[p]　　属性不正(ovratr が不正)

■機能説明

自 CPU のカーネルに対し、pk_dovr で示された内容でオーバーランハンドラを定義します。

オーバーランハンドラは、タスクが設定された時間を超えてプロセッサを使用した場合に起動される非タスクコンテキストのタイムイベントハンドラです。

ovratr には属性として、ハンドラを記述した言語を指定します。

ovratr := (TA_HLNG || TA_ASM)

◆ TA_HLNG（0x00000000）高級言語記述

◆ TA_ASM（0x00000001）アセンブラ記述

ovrhdr には、オーバーランハンドラの先頭アドレスを指定します。

def_ovr サービスコールでは、pk_dovr=NULL(0)が指定された場合は、オーバーランハンドラの定義を解除します。

また、すでにオーバーランハンドラが定義されている状態で、本サービスコールが呼び出された場合は、以前の定義を解除し、新しい定義に置き換えます。

6.25.2　オーバーランハンドラの動作開始(sta_ovr, ista_ovr)

■C 言語 API

```
ER ercd = sta_ovr(ID tskid, OVRTIM ovrtim);
ER ercd = ista_ovr(ID tskid, OVRTIM ovrtim);
```

■引数

tskid	タスク ID
ovrtim	上限プロセッサ時間

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)<0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid)) (3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_CTX	[k]	コンテキストエラー (GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)
E_OBJ	[k]	オブジェクト状態不正(オーバーランハンドラが定義されていない)

■機能説明

tskid で示されたタスクに対して、オーバーランハンドラの動作を開始します。

tskid=TSK_SELF（0）の指定により自タスクの指定になります。

対象タスクの上限プロセッサ時間を ovrtim で指定される時間に設定し、使用プロセッサ時間を 0 クリアします。すでに動作しているオーバーランハンドラが指定された場合は、以前の上限プロセッサ時間の設定を解除し、新しい上限プロセッサ時間を設定します。

使用プロセッサ時間が上限プロセッサ時間を超えたときに、オーバーランハンドラが起動されます。

ovrtim に 0 が指定された場合は、対象タスクがプロセッサを使用してから、1 回目のタイムティックでオーバーランハンドラが起動されます。

ovrtim に指定可能な最大値は、(0xFFFFFFFF-TIC_NUME)/TIC_DENO に制限されます。これより大きな値を指定した場合の動作は保証されません。

時間の管理方法については、「5.13.7 時間の精度」を参照してください。

sta_ovr では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。一方、ista_ovr では、tskid に他 CPU のカーネルに属するタスクを指定することはできません。

RENESAS

6.25.3 オーバーランハンドラの動作停止(stp_ovr, istp_ovr)

■C 言語 API

```
ER ercd = stp_ovr(ID tskid);
ER ercd = istp_ovr(ID tskid);
```

■引数

tskid　　タスク ID

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号
		(1)CPUID が不正(GET_CPUID(tskid)が不正)
		(2)ローカル ID 範囲外
		(GET_LOCALID(tskid)<0,
		(GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid))
		(3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_CTX	[k]	コンテキストエラー
		(GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)
E_OBJ	[k]	オブジェクト状態不正(オーバーランハンドラが定義されていない)

■機能説明

tskid で示されたタスクに対して、上限プロセッサ時間の設定を解除し、オーバーランハンドラの動作を停止します。

tskid=TSK_SELF（0）の指定により、自タスクの指定になります。

stp_ovr では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。一方、istp_ovr では、tskid に他 CPU のカーネルに属するタスクを指定することはできません。

6.25.4 オーバーランハンドラの状態参照(ref_ovr, iref_ovr)

■C 言語 API

```
ER ercd = ref_ovr(ID tskid, T_ROVR *pk_rovr);
ER ercd = iref_ovr(ID tskid, T_ROVR *pk_rovr);
```

■引数

tskid	タスク ID
pk_rovr	オーバーランハンドラの状態を返すパケットへのポインタ

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

```
typedef    struct {
           STAT     ovrstat;     オーバーランハンドラの動作状態
           OVRTIM   leftotm;     残りのプロセッサ時間
} T_ROVR;
```

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)<0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid)) (3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_CTX	[k]	コンテキストエラー (GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)
E_OBJ	[k]	オブジェクト状態不正(オーバーランハンドラが定義されていない)

■機能説明

tskid で示されたタスクのオーバーランハンドラに関する状態を参照し、pk_rovr が指す領域にオーバーランハンドラの動作状態(ovrstat)、残りのプロセッサ時間(leftotm)を返します。

tskid=TSK_SELF（0）の指定により、自タスクの指定になります。

ovrstat には、対象オーバーランハンドラの動作状態として、上限プロセッサ時間の設定状態を返します。

◆ TOVR_STP（0x00000000）上限プロセッサ時間が設定されていない

◆ TOVR_STA（0x00000001）上限プロセッサ時間が設定されている

leftotm には、対象タスクを原因としてオーバーランハンドラが起動されるまでの残りプロセッサ時間を返します。対象タスクに上限プロセッサ時間が設定されていない場合、leftotm は不定値となります。

ref_ovr では、ディスパッチ保留状態以外の場合のみ、tskid に他 CPU のカーネルに属するタスクを指定することができます。一方、iref_ovr では、tskid に他 CPU のカーネルに属するタスクを指定することはできません。

6.26 システム状態管理機能

表 6.37にシステム状態管理でサポートしているサービスコール一覧を示します。

表6.37 システム状態管理サービスコール

項番	サービスコール *1		機 能	呼び出し可能なシステム状態 *2						
				T	N	E	D	U	L	C
1	rot_rdq	[B] [S]	タスクの優先順位の回転	○		○	○	○		
	irot_rdq	[B] [S]			○	○	○	○		
2	get_tid	[B] [S]	実行状態のタスク ID の参照	○		○	○	○		○
	iget_tid	[S]			○	○	○	○		○
3	loc_cpu	[B] [S]	CPU ロック状態への移行	○		○	○	○	○	
	iloc_cpu	[S]			○	○	○	○	○	
4	unl_cpu	[B] [S]	CPU ロック状態の解除	○		○	○	○	○	
	iunl_cpu	[S]			○	○	○	○	○	
5	dis_dsp	[B] [S]	ディスパッチの禁止	○		○	○	○		
6	ena_dsp	[B] [S]	ディスパッチの許可	○		○	○	○		
7	sns_ctx	[S]	コンテキストの参照	○	○	○	○	○	○	○
8	sns_loc	[S]	CPU ロック状態の参照	○	○	○	○	○	○	○
9	sns_dsp	[S]	ディスパッチ禁止状態の参照	○	○	○	○	○	○	○
10	sns_dpn	[S]	ディスパッチ保留状態の参照	○	○	○	○	○	○	○
11	vsta_knl	[s]	カーネルの起動	○	○	○	○	○	○	○
	ivsta_knl	[s]		○	○	○	○	○	○	○
12	vini_rmt		リモートサービスコール環境の初期化	○		○		○		
13	vsys_dwn	[s]	システムダウン	○	○	○	○	○	○	○
	ivsys_dwn	[s]		○	○	○	○	○	○	○
14	vget_trc		トレースの取得	○		○	○	○		
	ivget_trc				○	○	○	○		
15	ivbgn_int		割込みハンドラの開始をトレースに取得		○	○	○	○		
16	ivend_int		割込みハンドラの終了をトレースに取得		○	○	○	○		

【注】 *1 "[S]"はスタンダードプロファイルのサービスコール、"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
“[B]”はベーシックプロファイルのサービスコールです。
“[R]”は、リモート呼び出し可能なサービスコールです。

*2 それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
“C”はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
“○”はその状態から呼び出し可能、”△”は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

6.26.1 タスクの優先順位の回転(rot_rdq, irot_rdq)

■C 言語 API

```
ER ercd = rot_rdq(PRI tskpri);
ER ercd = irot_rdq(PRI tskpri);
```

■引数

tskpri　　タスク優先度

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR	[p]	パラメータエラー(tskpri＜0、tskpri＞自 CPU の TMAX_TPRI、非タスクコンテキストで tskpri=TPRI_SELF(0)を指定)

■機能説明

tskpri で示された優先度の自 CPU のレディキューにつながれている先頭タスクをレディキューの最後尾につなぎかえ（レディキューを回転）、次につながれているタスクに実行を切り替えます。

tskpri=TPRI_SELF（0）を指定すると、自タスクのベース優先度のレディキューを回転します。ミューテックス機能を使用しない場合は、ベース優先度と現在優先度は同じですが、ミューテックスをロック中は一般には現在優先度とベース優先度は一致しないため、TPRI_SELF を指定しても現在の自タスクが属する優先度のレディキューを回転することはできません。

6.26.2 実行状態のタスク ID の参照(get_tid, iget_tid)

■C 言語 API

```
ER ercd = get_tid(ID *p_tskid);
ER ercd = iget_tid(ID *p_tskid);
```

■引数

p_tskid　タスク ID を返す記憶域へのポインタ

■リターン値

正常終了（E_OK）

■機能説明

自 CPU の実行状態のタスクの ID を求め、その結果を p_tskid の指す領域に返します。

具体的には、タスクコンテキストから呼び出された場合は自タスクの ID を返し、非タスクコンテキストから呼び出された場合はその時実行していたタスクの ID を返します。ビット 14～12 は、自 CPU の CPUID(1 または 2)となります。実行状態のタスクがない場合は、TSK_NONE（0）を返します。

本サービスコールは、ノーマル CPU 例外ハンドラからも呼び出せます。

6.26.3 CPU ロック状態への移行(loc_cpu, iloc_cpu)

■C 言語 API

```
ER ercd = loc_cpu(void);
ER ercd = iloc_cpu(void);
```

■リターン値

正常終了（E_OK）

■機能説明

自 CPU のカーネルのシステム状態を CPU ロック状態とし、割込みとタスクのディスパッチを禁止します。

CPU ロック状態の特長を以下に示します。

(1) CPUロック状態の間は、タスクのスケジューリングは行われません。
(2) cfgファイルで指定したsystem.system_IPL(カーネル割込みマスクレベル)以下のレベルの割込みが禁止されます。
(3) CPUロック状態から呼び出し可能なサービスコールは、以下のサービスコールのみです。その他のサービスコールが呼び出された場合の動作は保証されません。

- ext_tsk
- exd_tsk
- sns_tex
- loc_cpu, iloc_cpu
- unl_cpu, iunl_cpu
- sns_ctx
- sns_loc
- sns_dsp
- sns_dpn
- vsta_knl, ivsta_knl
- vsys_dwn, ivsys_dwn
- vsns_tmr

CPU ロック状態は、以下の操作で解除されます。

(a) unl_cpu, iunl_cpuサービスコールの呼び出し
(b) ext_tsk, exd_tskサービスコールの呼び出し

CPU ロック状態と CPU ロック解除状態の間の遷移は、loc_cpu, iloc_cpu, unl_cpu, iunl_cpu, ext_tsk, exd_tsk サービスコールによってのみ発生します。カーネル割込みマスクレベル以下の割込みハンドラ、タイムイベントハンドラ、初期化ルーチン、タスク例外処理ルーチンの終了時には、必ず CPU ロック解除状態でなければなりません。CPU ロック状態の場合、動作は保証されません。なお、これらのハンドラ開始時は、常に CPU ロック解除状態です。

CPU 例外ハンドラで CPU ロック/ロック解除状態を変更した場合、必ず終了前に元の状態に戻さなくてはなりません。戻さない場合、動作は保証されません。

すでに CPU ロック状態のときに、再度本サービスコールを呼び出してもエラーにはなりませんが、キューイングは行いません。

RENESAS

6.26.4　CPU ロック状態の解除(unl_cpu, iunl_cpu)

■C 言語 API

```
ER ercd = unl_cpu(void);
ER ercd = iunl_cpu(void);
```

■リターン値

正常終了（E_OK）

■機能説明

loc_cpu, iloc_cpu サービスコールによって設定されていた自 CPU のカーネルの CPU ロック状態を解除します。ディスパッチ許可状態から unl_cpu サービスコールを発行した場合、タスクのスケジューリングが行われます。

割込みハンドラ内で iloc_cpu を呼び出し、CPU ロック状態に移行した場合は、割込みハンドラからリターンする前に必ず iunl_cpu を呼び出し、CPU ロック状態を解除してください。

CPU ロック状態とディスパッチ禁止状態は、独立して管理されます。そのため、unl_cpu, iunl_cpu サービスコールでは、dis_dsp サービスコールによるディスパッチ禁止状態は解除されません。

CPU ロック解除状態から本サービスコールを呼び出してもエラーにはなりませんが、キューイングは行いません。

6.26.5　ディスパッチの禁止(dis_dsp)

■C 言語 API

```
ER ercd = dis_dsp(void);
```

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_CTX　　[k]　　コンテキストエラー(許可されていないシステム状態からの呼び出し)

■機能説明

自 CPU のカーネルのシステム状態をディスパッチ禁止状態にします。ディスパッチ禁止状態の特長を、以下に示します。

(1) タスクのスケジューリングが行われなくなるため、自タスク以外のタスクが実行状態に移行することはなくなります。
(2) 割込みは受け付けられます。
(3) 待ち状態になるサービスコールを呼び出せません。

ディスパッチ禁止状態の間に以下の操作を行うと、システム状態はタスク実行状態に戻ります。

(1) ena_dspサービスコールの呼び出し
(2) ext_tsk, exd_tskサービスコールの呼び出し

ディスパッチ禁止状態とディスパッチ許可状態の間の遷移は、dis_dsp, ena_dsp, ext_tsk, exd_tsk サービスコールによってのみ発生します。

CPU 例外ハンドラでディスパッチ禁止/許可状態を変更した場合、必ず終了前に元の状態に戻さなくてはなりません。戻さない場合、動作は保証されません。

本サービスコールによってディスパッチ禁止状態の間は、ref_tsk サービスコールで自タスクの状態を参照しても実行状態とは見えない場合があるので、注意してください。

すでにディスパッチ禁止状態のときに再度本サービスコールを呼び出してもエラーにはなりませんが、キューイングは行いません。

本サービスコールは、必ずタスクコンテキストかつ CPU ロック解除状態で呼び出すようにしてください。その他の状態から本サービスコールを呼び出した場合は、正常な動作は保証されません。

RENESAS

6.26.6 ディスパッチの許可(ena_dsp)

■C 言語 API

```
ER ercd = ena_dsp(void);
```

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_CTX [k] コンテキストエラー(許可されていないシステム状態からの呼び出し)

■機能説明

dis_dsp サービスコールによって設定されていた自 CPU のカーネルのディスパッチ禁止状態を解除します。それにより、システムがタスク実行状態になった場合は、タスクのスケジューリングが行われます。

タスク実行状態から本サービスコールを呼び出してもエラーにはなりませんが、キューイングは行いません。

本サービスコールは、必ずタスクコンテキストかつ CPU ロック解除状態で呼び出すようにしてください。その他の状態から本サービスコールを呼び出した場合は、正常な動作は保証されません。

6.26.7 コンテキストの参照(sns_ctx)

■C 言語 API

```
BOOL state = sns_ctx(void);
```

■リターン値

非タスクコンテキストから呼び出された場合に TRUE、タスクコンテキストから呼び出された場合に FALSE を返します。

■機能説明

現在のコンテキスト種別を調べます。
本サービスコールは、CPU ロック状態およびノーマル CPU 例外ハンドラからも呼び出せます。

6.26.8 CPU ロック状態の参照(sns_loc)

■C 言語 API

```
BOOL state = sns_loc(void);
```

■リターン値

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

■機能説明

CPU ロック状態かどうかを調べます。
本サービスコールは、CPU ロック状態およびノーマル CPU 例外ハンドラからも呼び出せます。

6.26.9 ディスパッチ禁止状態の参照(sns_dsp)

■C 言語 API

```
BOOL state = sns_dsp(void);
```

■リターン値

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

■機能説明

ディスパッチ禁止状態かどうかを調べます。
本サービスコールは、CPU ロック状態およびノーマル CPU 例外ハンドラからも呼び出せます。

RENESAS

6.26.10 ディスパッチ保留状態の参照(sns_dpn)

■C 言語 API

```
BOOL state = sns_dpn(void);
```

■リターン値

ディスパッチ保留状態の場合に TRUE、そうでない場合に FALSE を返します。

■機能説明

ディスパッチ保留状態かどうかを調べます。
以下のいずれかの条件を満たすときは、ディスパッチ保留状態です。

(1) ディスパッチ禁止状態である
(2) CPUロック状態である
(3) 非タスクコンテキスト実行中である
(4) ノーマルCPU例外ハンドラ実行中である
(5) SRレジスタのIMASKが0でない

本サービスコールは、CPU ロック状態およびノーマル CPU 例外ハンドラからも呼び出せます。

6.26.11 カーネルの起動(vsta_knl, ivsta_knl)

■C 言語 API

```
void vsta_knl(void);
void ivsta_knl(void);
```

■機能説明

カーネルを起動します。

すでにカーネルを起動済みの場合は、それまでのマルチタスク環境は全て破棄されます。

本サービスコールは、CPU ロック状態およびノーマル CPU 例外ハンドラからも呼び出せます。また、カーネル起動前であっても呼び出せます。

本サービスコールは、すべての割込みをマスクした状態(SR.IMASK=15)で発行してください。

本サービスコールを呼び出すアプリケーションは、カーネルとリンクする必要があります。

本サービスコールからリターンすることはありません。

本サービスコールの処理概要を以下に示します。

(1) 割込みベクタテーブルを生成し、VBRレジスタを初期化
(2) カーネル内部テーブルを初期化
(3) cfgファイルで指定された各種オブジェクトの生成
(4) SR.IMASKをカーネル割込みマスクレベル(system.system_IPL)に設定
(5) タイマ初期化関数tdr_ini_tmr()をコール
(6) cfgファイルで指定された初期化ルーチンをコール
(7) マルチタスク環境へ移行

system.trace!=NO の場合、トレースシリアル番号の一貫性が崩れる場合があります。これを避けるには、CPUID#1 側の初期化ルーチンまたは最初のタスクの実行が開始した後に、CPUID#2 で vsta_knl を呼び出すようにしてください。

なお、本サービスコールはμITRON4.0 仕様外の機能です。

6.26.12 リモートサービスコール環境の初期化(vini_rmt)

■C 言語 API

```
ER ercd = vini_rmt(void);
```

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_CTX	[k]	コンテキストエラー(許可されていないシステム状態からの呼び出し)
E_SYS	[k]	システムエラー
EV_NORESOURCE	[k]	リソース不足 (1)IPI ポートの生成に失敗 (2)SVC サーバタスクの生成に失敗 (3)固定長メモリプールの生成に失敗

■機能説明

本サービスコールは、他 CPU からのリモートサービスコールを受理するため、および他 CPU へのリモートサービスコールのための初期化を行います。

remote_svc.num_server が 0 でない場合は、他 CPU からのリモートサービスコールを受理するための初期化として、以下を行います。

(1) IPIポートの生成
remote_svc.ipi_portidで指定されたIPIポートをIPI_create()によって生成します。

(2) SVCサーバタスクの生成・起動
remote_svc.num_serverで指定された数のSVCサーバタスクを、acre_tskによって生成します。
acre_tskに与えるパラメータパケット(T_CTSK構造体)の主な内容は以下の通りです。

- tskatr(タスク属性)： TA_HLNG | TA_ACT
- task(タスク開始アドレス)：カーネル内の SVC サーバタスク関数のアドレス
- itskpri(初期優先度)：remote_svc.priority
- stksz(スタックサイズ)：remote_svc.stack_size
- stk(スタックアドレス)：コンフィギュレーションによって自動生成された領域 (BC_hirmtstk セクション内)

また、remote_svc.num_wait が 0 でない場合は、他 CPU へのリモートサービスコールのための初期化として、以下を行います。

(3) 固定長メモリプールの生成
acre_mpfによってremote_svc.num_wait個のメモリブロックを持つ固定長メモリプールをひとつ生成します。acre_mpfに与えるパラメータパケット(T_CMPF構造体)の主な内容は以下の通りです。

- mpfatr(固定長メモリプール属性)： TA_TFIFO
- blkcnt(メモリブロック数)：remote_svc.num_wait
- blksz(メモリブロックサイズ)：20
- mpf(固定長メモリプール先頭アドレス)：コンフィギュレーションによって自動生成された領域(BD_hirmtmpf セクション内)
- mpfmb(固定長メモリプール管理テーブルアドレス)(system.mpfmanage が OUT の場合のみ)
  ：コンフィギュレーションによって自動生成された領域(BC_hiwrkセクション内)

本 CPU に対するリモートサービスコール、および本 CPU からのリモートサービスコールは、本サービスコールの完了後に可能になります。

本サービスコールを呼び出す前に、IPI_init()が完了している必要があります。さらに、自 CPUID が 1 でない場合は、本サービスコールの前にマスタ CPUID#1 側の vini_rmt サービスコールが完了している必要があります。これらが満たされない場合の動作は未定義です。

本サービスコールは、カーネル起動直後に一度だけ呼び出すようにしてください。

本サービスコールは、API 処理部のみで実装されています。すなわち、本サービスコール処理はすべて呼び出し元と同じコンテキストで実行します。

なお、本サービスコールはμITRON4.0 仕様外の機能です。

RENESAS

6.26.13 システムダウン(vsys_dwn, ivsys_dwn)

■C 言語 API

```
void vsys_dwn(W type, VW inf1, VW inf2, VW inf3);
void ivsys_dwn(W type, VW inf1, VW inf2, VW inf3);
```

■引数

type	エラー種別
inf1	システム異常情報 1
inf2	システム異常情報 2
inf3	システム異常情報 3

■機能説明

システムダウンルーチンに制御を渡します。

type には、エラー種別として発生したエラーに対応した値(1～0x7FFFFFFF)を設定してください。なお、0 以下の値はシステム用に予約されています。

カーネル内で異常を検出した場合にも、システムダウンルーチンが呼び出されます。

本サービスコールは、CPU ロック状態およびノーマル CPU 例外ハンドラからも呼び出せます。

本サービスコールからリターンすることはありません。

なお、本サービスコールはμITRON4.0 仕様外の機能です。

6.26.14 トレースの取得(vget_trc, ivget_trc)

■C 言語 API

```
ER ercd = vget_trc(VW para1, VW para2, VW para3, VW para4);
ER ercd = ivget_trc(VW para1, VW para2, VW para3, VW para4);
```

■引数

para1	パラメータ 1
para2	パラメータ 2
para3	パラメータ 3
para4	パラメータ 4

■リターン値

正常終了(E_OK)

■機能説明

ユーザ任意の情報をトレース取得します。

para1～para4 は、取得される情報を識別するために、ユーザが自由に使用できます。

取得したトレース情報は、デバッギングエクステンション(DX)でトレース表示されます。

cfg ファイルで、system.trace に NO を指定していた場合は、本サービスコールは何も処理せずに終了します。

なお、本サービスコールはμITRON4.0 仕様外の機能です。

6.26.15 割込みハンドラの開始をトレースに取得(ivbgn_int)

■C 言語 API

```
ER ercd = ivbgn_int(UINT dintno);
```

■引数

dintno　　割込みハンドラ番号

■リターン値

正常終了（E_OK）

■機能説明

dintno で示された割込みハンドラ番号に対する割込みハンドラの処理開始を、トレース取得します。
割込みハンドラ番号には、CPU の割込みベクタ番号を指定します。
本サービスコールは、割込みハンドラの先頭で呼び出すようにしてください。また、必ず ivend_int サービスコールとセットで使用してください。
割込みハンドラ以外から呼び出してもエラーにはなりませんが、この場合デバッギングエクステンションによるトレース表示が不正になる可能性があります。
cfg ファイルで、system.trace に NO を指定していた場合は、本サービスコールは何も処理せずに終了します。

なお、本サービスコールはμITRON4.0 仕様外の機能です。

6.26.16　割込みハンドラの終了をトレースに取得(ivend_int)

■C 言語 API

```
ER ercd = ivend_int(UINT dintno);
```

■引数

dintno　　割込みハンドラ番号

■リターン値

正常終了（E_OK）

■機能説明

dintno で示された割込みハンドラ番号に対する割込みハンドラの処理終了をトレース情報に取得します。

割込みハンドラ番号には、CPU の割込みベクタ番号を指定します。

本サービスコールは、割込みハンドラの最後で呼び出すようにしてください。また、必ず ivbgn_int サービスコールとセットで使用してください。

割込みハンドラ以外から呼び出ししてもエラーにはなりませんが、この場合デバッギングエクステンションによるトレース表示が不正になる可能性があります。

cfg ファイルで、system.trace に NO を指定していた場合は、本サービスコールは何も処理せずに終了します。

なお、本サービスコールはμITRON4.0 仕様外の機能です。

RENESAS

6.27　割込み管理機能

表 6.38に割込み管理でサポートしているサービスコール一覧を示します。

表6.38　割込み管理サービスコール

項番	サービスコール　*1	機　能	呼び出し可能なシステム状態　*2						
			T	N	E	D	U	L	C
1	def_inh	割込みハンドラの定義	○		○	○	○		
	idef_inh			○	○	○	○		
2	chg_ims	割込みマスクの変更	○		○	○	○		
	ichg_ims			○	○	○	○		
3	get_ims	割込みマスクの参照	○		○	○	○		
	iget_ims			○	○	○	○		

【注】 *1　"[S]"はスタンダードプロファイルのサービスコール、"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
“[B]”はベーシックプロファイルのサービスコールです。
“[R]”は、リモート呼び出し可能なサービスコールです。

*2　それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
“C”はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
“○”はその状態から呼び出し可能、”△”は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

表 6.39に割込み管理の仕様を示します。

表6.39　割込み管理の仕様

項番	項目	内容
1	割込みハンドラ番号	4〜_MAX_INT (最大 511)
2	サポート属性	TA_HLNG : 高級言語記述 TA_ASM : アセンブリ言語記述

6.27.1 割込みハンドラの定義(def_inh, idef_inh)

■C 言語 API

```
ER ercd = def_inh(INHNO inhno, T_DINH *pk_dinh);
ER ercd = idef_inh(INHNO inhno, T_DINH *pk_dinh);
```

■引数

inhno	割込みハンドラ番号
pk_dinh	割込みハンドラ定義情報を格納したパケットへのポインタ

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

```
typedef   struct {
          ATR     inhatr;     ハンドラ属性
          FP      inthdr;     ハンドラアドレス
          UINT    inhsr;      (将来拡張用)
} T_DINH;
```

■エラーコード

E_RSATR	[p]	属性不正(inhatr が不正)
E_PAR	[p]	パラメータエラー
		不正番号指定(inhno が 0～3, 60～63, または inhno>自 CPU の_MAX_INT)

■機能説明

自 CPU のカーネルに対し、inhno で示された割込みハンドラ番号に対する割込みハンドラを、pk_dinh で示された内容で定義します。
本サービスコールは、cfg ファイルで system.vector_type に RAM または RAM_ONLY_DIRECT を指定した場合のみ使用できます。

割込みハンドラ番号には、CPU のベクタ番号を指定します。
割込みハンドラ番号 0～3（パワーオンリセット、マニュアルリセット）には定義できません。また、割込みハンドラ番号 60～63 はシステム予約なので指定できません。

inhatr には属性として、以下を指定します。

```
inhatr := (TA_HLNG‖TA_ASM) [ | (VTA_DIRECT || VTA_REGBANK) ]
```

◆ TA_HLNG（0x00000000）高級言語記述
◆ TA_ASM（0x00000001）アセンブラ記述
◆ VTA_DIRECT（0x80000000）ダイレクト属性
◆ VTA_REGBANK（0x40000000）レジスタバンクを使用するノーマル割込みハンドラ

VTA_DIRECT 属性を指定すると、割込み発生時にカーネルの介入無く定義したハンドラが起動されます。このハンドラを「ダイレクト割込みハンドラ」と呼びます。一方、VTA_DIRECT 属性を指定しない場合は、割込み発生時にカーネルを介してハンドラが起動されます。このハンドラを「ノーマル割込みハンドラ」と呼びます。
カーネル割込みマスクレベルより高い割込みレベルの割込みハンドラを定義する場合は、必ず VTA_DIRECT を指定しなければなりません。
なお、VTA_DIRECT の有無によって、ハンドラの記述方法が異なることに注意してください。詳細は、「12.5 割込みハンドラ」を参照してください。
また、cfg ファイルで system.vector_type に RAM_ONLY_DIRECT を指定した場合は、VTA_DIRECT 指定のないハンドラを定義することはできません。

VTA_REGBANK は、以下の全ての条件を満たす場合のみ有効です。その他の場合は、VTA_REGBANK は意味を持ちません。

(a) VTA_DIRECT属性を指定していない。
(b) system.regbankにBANKLEVELxxを指定している。
(c) 「CPU割込み仕様定義ファイル(kernel_intspec.h)」で、当該CPUに対するINTSPEC_IBNR_ADR1(CPUID#1用)またはINTSPEC_IBNR_ADR2(CPUID#2用)に、0以外を指定している（レジスタバンクをサポートしたCPUを使用する）
(d) ベクタ番号に、「CPU割込み仕様定義ファイル(kernel_intspec.h)」に定義したINTSPEC_NOBANK_VEC???に対応するベクタ番号以外を指定している。（CPU仕様上、レジスタバンクを利用可能な割込み要因のベクタ番号を指定している）

これらの条件を満たす場合、定義する割込みハンドラの割込みレベルによって、以下のように適切に VTA_REGBANK を指定しなければなりません。これらが守られない場合、割込みハンドラは正常に動作しません。

(i) system.regbankに、使用する割込みハンドラの割込みレベルに対応したBANKLEVELxxが指定されている場合
VTA_REGBANKの指定が必須です。

(ii) system.regbankに、使用する割込みハンドラの割込みレベルに対応したBANKLEVELxxが指定されていない場合
VTA_REGBANKを指定してはなりません。

inhsr は将来拡張用であり、単に無視されます。
pk_dinh＝NULL（0）と指定した場合には、inhno の定義を解除します。

inhsr は、μITRON4.0 仕様の範囲外のメンバです。

6.27.2 割込みマスクの変更(chg_ims, ichg_ims)

■C 言語 API

```
ER ercd = chg_ims(IMASK imask);
ER ercd = ichg_ims(IMASK imask);
```

■引数

imask　　割込みマスク値

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_PAR　　[p]　　パラメータエラー(imask に SR_IMS00～SR_IMS15 以外の値を指定)

■機能説明

CPU の割込みマスク(SR レジスタの IMASK ビット)を imask で指定した値に変更します。
imask には、以下の指定ができます。

◆ SR_IMSnn（0x0000000m）割込みマスクレベルを nn に変更
- nn : 0～15 を 10 進数 2 桁で表現した文字列（“00”, “01”, “02”, ... , “15”）
- m : nn を 16 進数に変換した文字

割込みマスクの制御についての詳細は、「4.8.2 割込み制御方法(SR レジスタの IMASK ビット)」を参照してください。

6.27.3 割込みマスクの参照(get_ims, iget_ims)

■C 言語 API

```
ER ercd = get_ims(IMASK *p_imask);
ER ercd = iget_ims(IMASK *p_imask);
```

■引数

p_imask　　割込みマスクレベルを返す記憶域へのポインタ

■リターン値

正常終了（E_OK）

■機能説明

現在の CPU ステータスレジスタ(SR)の割込みマスクビット(IMASK ビット)を参照し、割込みマスクレベルを p_imask の指す領域に返します。p_imask の指す領域に返る値は、chg_ims サービスコールで用いるパラメータ imask と同じフォーマットです。

RENESAS

6.28 サービスコール管理機能

表 6.40にサービスコール管理でサポートしているサービスコール一覧を示します。

表6.40 サービスコール管理サービスコール

項番	サービスコール *1	機 能	呼び出し可能なシステム状態 *2						
			T	N	E	D	U	L	C
1	def_svc	拡張サービスコールの定義	○		○	○	○		
	idef_svc			○	○	○	○		
2	cal_svc	サービスコールの呼び出し	○		○	○	○		
	ical_svc			○	○	○	○		

【注】 *1 "[S]"はスタンダードプロファイルのサービスコール、"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
“[B]”はベーシックプロファイルのサービスコールです。
“[R]”は、リモート呼び出し可能なサービスコールです。

*2 それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
“C”はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
“○”はその状態から呼び出し可能、”△”は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

表 6.41にサービスコール管理の仕様を示します。

表6.41 サービスコール管理の仕様

項番	項目	内容
1	拡張サービスコールの機能コード	1～_MAX_FNCD (最大 1023)
2	渡せるパラメータ	VP_INT 型で、0～4 個
3	サポート属性	TA_HLNG： 高級言語記述 TA_ASM： アセンブリ言語記述

6.28.1 拡張サービスコールの定義(def_svc, idef_svc)

■C 言語 API

```
ER ercd = def_svc(FN fncd, T_DSVC *pk_dsvc);
ER ercd = idef_svc(FN fncd, T_DSVC *pk_dsvc);
```

■引数

fncd	拡張サービスコールの機能コード
pk_dsvc	拡張サービスコール定義情報を格納したパケットへのポインタ

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

```
typedef    struct {
           ATR      svcatr;      拡張サービスコールルーチン属性
           FP       svcrtn;      拡張サービスコールルーチンドレス
} T_DSVC;
```

■エラーコード

E_RSATR	[p]	属性不正(svcatr が不正)
E_PAR	[p]	パラメータエラー(fncd≦0、fncd＞自 CPU の_MAX_FNCD)

■機能説明

自 CPU のカーネルに対し、fncd で示された拡張機能コードに対する拡張サービスコールルーチンを、pk_dsvc で示された内容で定義します。

svcatr には属性として、ルーチンを記述した言語を指定します。

svcatr := （（TA_HLNG ‖ TA_ASM））

◆ TA_HLNG（0x00000000）高級言語記述

◆ TA_ASM（0x00000001）アセンブラ記述

svcrtn には、拡張サービスコールルーチンの先頭アドレスを指定します。

pk_dsvc＝NULL（0）と指定した場合には、fncd の拡張サービスコールルーチンの定義を解除します。

拡張サービスコールルーチンは、呼び出し元の状態を引き継ぎます。

6.28.2　サービスコールの呼び出し(cal_svc, ical_svc)

■C 言語 API

```
ER_UINT ercd = cal_svc(FN fncd, ...);
ER_UINT ercd = ical_svc(FN fncd, ...);
```

■引数

fncd　　拡張サービスコールの機能コード

"..."で示した部分には、VP_INT 型の引数を 0 から 4 個記述できます。5 つ以上指定しても、拡張サービスコールルーチンには 4 つ目の引数までしか渡りません。

par1　　パラメータ 1

par2　　パラメータ 2

par3　　パラメータ 3

par4　　パラメータ 4

■リターン値

サービスコールからのリターン値

■エラーコード

E_RSFN　[p]　予約機能コード(fncd が不正あるいは使用できない)

■機能説明

fncd で指定された機能コードに対応する拡張サービスコールルーチンを実行します。

パラメータは、VP_INT 型で 0～4 個指定できます。呼び出される拡張サービスコールルーチンでは、par 1 ～par4 がそれぞれ R4～R7 に格納されて渡されます。

詳細は、「12.4 拡張サービスコールルーチン」を参照してください。

RENESAS

6.29　システム構成管理機能

表 6.42にシステム構成管理でサポートしているサービスコール一覧を示します。

表6.42　システム構成管理サービスコール

項番	サービスコール　*1	機　能	呼び出し可能なシステム状態　*2						
			T	N	E	D	U	L	C
1	def_exc	CPU 例外ハンドラの定義	○		○	○	○		
	idef_exc			○	○	○	○		
2	vdef_trp	CPU 例外ハンドラの定義 (TRAPA 命令例外)	○		○	○	○		
	ivdef_trp			○	○	○	○		
3	ref_cfg	コンフィギュレーション情報の参照	○		○	○	○		
	iref_cfg			○	○	○	○		
4	ref_ver	バージョン情報の参照	○		○	○	○		
	iref_ver			○	○	○	○		

【注】 *1　"[S]"はスタンダードプロファイルのサービスコール、"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
“[B]”はベーシックプロファイルのサービスコールです。
“[R]”は、リモート呼び出し可能なサービスコールです。

*2　それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
"C"はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
“○”はその状態から呼び出し可能、”△”は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

表 6.43にシステム構成管理の仕様を示します。

表6.43　システム構成管理の仕様

項番	項目	内容
1	CPU 例外ハンドラ番号	4～_MAX_INT (最大 511)
2	サポート属性	TA_HLNG：高級言語記述 TA_ASM：アセンブリ言語記述

RENESAS

6.29.1 CPU 例外ハンドラの定義(def_exc, idef_exc)

■C 言語 API

```
ER ercd = def_exc(EXCNO excno, T_DEXC *pk_dexc);
ER ercd = idef_exc(EXCNO excno, T_DEXC *pk_dexc);
```

■引数

excno	CPU 例外ハンドラ番号
pk_dexc	CPU 例外ハンドラ定義情報を格納したパケットへのポインタ

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

```
typedef   struct {
          ATR     excatr;      ハンドラ属性
          FP      exchdr;      ハンドラアドレス
          UINT    excsr;       (将来拡張用)
} T_DEXC;
```

■エラーコード

E_RSATR	[p]	属性不正(excatr が不正)
E_PAR	[p]	パラメータエラー
		不正番号指定(inhno が 0～3，60～63，または inhno>自 CPU の_MAX_INT)

■機能説明

自 CPU のカーネルに対し、excno で示された CPU 例外ハンドラ番号に対する CPU 例外ハンドラを、pk_dexc で示された内容で定義します。

本サービスコールは、cfg ファイルで system.vector_type に RAM または RAM_ONLY_DIRECT を指定した場合のみ使用できます。

CPU 例外ハンドラ番号には、CPU のベクタ番号を指定します。

CPU 例外ハンドラ番号 0～3（パワーオンリセット、マニュアルリセット）には定義できません。また、CPU 例外ハンドラ番号 60～63 はシステム予約なので指定できません。

excatr には属性として、以下を指定します。

excatr := （TA_HLNG || TA_ASM） [| VTA_DIRECT]

- ◆ TA_HLNG（0x00000000）高級言語記述
- ◆ TA_ASM（0x00000001）アセンブラ記述
- ◆ VTA_DIRECT（0x80000000）ダイレクト属性

VTA_DIRECT 属性を指定すると、CPU 例外発生時にカーネルの介入無く定義したハンドラが起動されます。このハンドラを「ダイレクト CPU 例外ハンドラ」と呼びます。一方、VTA_DIRECT 属性を指定しない場合は、CPU 例外発生時にカーネルを介してハンドラが起動されます。このハンドラを「ノーマル CPU 例外ハンドラ」と呼びます。

なお、VTA_DIRECT の有無によって、ハンドラの記述方法が異なることに注意してください。詳細は、「12.6 CPU 例外ハンドラ(TRAPA 例外を含む)」を参照してください。

また、cfg ファイルで system.vector_type に RAM_ONLY_DIRECT を指定した場合は、VTA_DIRECT 指定のないハンドラを定義することはできません。

excsr は将来拡張用であり、単に無視されます。実際のハンドラ起動時の SR は、CPU の例外処理によって決まります。

pk_dexc＝NULL（0）と指定した場合には、excno の定義を解除します。

ノーマル CPU 例外ハンドラは、CPU 例外発生元とは別の「CPU 例外ハンドラ実行状態」と呼ぶコンテキスト状態で実行されます。この状態から呼び出し可能なサービスコールは、以下のサービスコールのみです。その他のサービスコールが呼び出された場合の動作は保証されません。

- ◆ sns_tex
- ◆ sns_ctx
- ◆ sns_loc
- ◆ sns_dsp
- ◆ sns_dpn
- ◆ get_tid, iget_tid
- ◆ ras_tex, iras_tex
- ◆ vsta_knl, ivsta_knl
- ◆ vsys_dwn, ivsys_dwn
- ◆ vsns_tmr

一方、ダイレクト CPU 例外ハンドラは、CPU 例外発生前と同じコンテキスト状態で実行します。このため、呼び出し可能なサービスコールも CPU 例外発生前と同じになり、静的には決まりません。

なお、TRAPA 命令に対する CPU 例外ハンドラを定義するには、本サービスコールではなく vdef_trp, ivdef_trp サービスコールを使用してください。

excsr は、μITRON4.0 仕様の範囲外のメンバです。

6.29.2 CPU 例外(TRAPA 命令例外)ハンドラ定義(vdef_trp, ivdef_trp)

■C 言語 API

```
ER ercd = vdef_trp(UINT dtrpno, T_DTRP *pk_dtrp);
ER ercd = ivdef_trp(UINT dtrpno, T_DTRP *pk_dtrp);
```

■引数

dtrpno	トラップ番号
pk_dtrp	CPU 例外(TRAPA 命令例外)ハンドラ定義情報を格納したパケットへのポインタ

■リターン値

正常終了（E_OK）、またはエラーコード

■パケットの構造

```
typedef  struct {
         ATR     trpatr;     CPU 例外(TRAPA 命令例外)ハンドラ属性
         FP      trphdr;     CPU 例外(TRAPA 命令例外)ハンドラアドレス
         UINT    trpsr;      (将来拡張用)
} T_DTRP;
```

■エラーコード

E_RSATR	[p]	属性不正(trpatr が不正)
E_PAR	[p]	パラメータエラー 不正番号指定(inhno が 0～3, 60～63, または inhno>自 CPU の_MAX_INT)

RENESAS

■機能説明

自 CPU のカーネルに対し、dtrpno で示されたトラップ番号に対する CPU 例外ハンドラ(TRAPA 命令例外)ハンドラを、pk_dtrp で示された内容で定義します。

本サービスコールは、cfg ファイルで system.vector_type に RAM または RAM_ONLY_DIRECT を指定した場合のみ使用できます。

トラップ番号には、CPU のベクタ番号を指定します。

トラップ番号 0～3（パワーオンリセット、マニュアルリセット）には定義できません。また、トラップ番号 60～63 はシステム予約なので指定できません。

trpatr には属性として、以下を指定します。

```
trpatr := (TA_HLNG‖TA_ASM) [ | VTA_DIRECT ]
```

- ◆ TA_HLNG（0x00000000）高級言語記述
- ◆ TA_ASM（0x00000001）アセンブラ記述
- ◆ VTA_DIRECT（0x80000000）ダイレクト属性

VTA_DIRECT 属性を指定すると、TRAPA 命令例外発生時にカーネルの介入無く定義したハンドラが起動されます。このハンドラを「ダイレクト CPU 例外ハンドラ」と呼びます。一方、VTA_DIRECT 属性を指定しない場合は、TRAPA 命令例外発生時にカーネルを介してハンドラが起動されます。このハンドラを「ノーマル CPU 例外ハンドラ」と呼びます。

なお、VTA_DIRECT の有無によって、ハンドラの記述方法が異なることに注意してください。詳細は、「12.6 CPU 例外ハンドラ(TRAPA 例外を含む)」を参照してください。

また、cfg ファイルで system.vector_type に RAM_ONLY_DIRECT を指定した場合は、VTA_DIRECT 指定のないハンドラを定義することはできません。

trpsr は将来拡張用であり、単に無視されます。実際のハンドラ起動時の SR は、CPU の例外処理によって決まります。

pk_dtrp＝NULL（0）と指定した場合には、dtrpno の定義を解除します。

ノーマル CPU 例外ハンドラは、CPU 例外発生元とは別の「CPU 例外ハンドラ実行状態」と呼ぶコンテキスト状態で実行されます。この状態から呼び出し可能なサービスコールは、以下のサービスコールのみです。その他のサービスコールが呼び出された場合の動作は保証されません。

- ◆ sns_tex
- ◆ sns_ctx
- ◆ sns_loc
- ◆ sns_dsp
- ◆ sns_dpn
- ◆ get_tid, iget_tid
- ◆ ras_tex, iras_tex
- ◆ vsta_knl, ivsta_knl
- ◆ vsys_dwn, ivsys_dwn
- ◆ vsns_tmr

一方、ダイレクト CPU 例外ハンドラは、CPU 例外発生前と同じコンテキスト状態で実行します。このため、呼び出し可能なサービスコールも CPU 例外発生前と同じになり、静的には決まりません。

本サービスコールはμITRON4.0 仕様外の機能です。

6.29.3 コンフィギュレーション情報の参照(ref_cfg, iref_cfg)

■C 言語 API

```
ER ercd = ref_cfg(T_RCFG *pk_rcfg);
ER ercd = iref_cfg(T_RCFG *pk_rcfg);
```

■引数

pk_rcfg　　コンフィギュレーション情報を返すパケットへのポインタ

■リターン値

正常終了（E_OK）

■パケットの構造

```
typedef    struct {
           ID      maxtskid;    最大ローカルタスク ID
           ID      ststkid;     最大ローカルスタティックスタック使用タスク ID
           ID      maxsemid;    最大ローカルセマフォ ID
           ID      maxflgid;    最大ローカルイベントフラグ ID
           ID      maxdtqid;    最大ローカルデータキューID
           ID      maxmbxid;    最大ローカルメールボックス ID
           ID      maxmtxid;    最大ローカルミューテックス ID
           ID      maxmbfid;    最大ローカルメッセージバッファ ID
           ID      maxmplid;    最大ローカル可変長メモリプール ID
           ID      maxmpfid;    最大ローカル固定長メモリプール ID
           ID      maxcycid;    最大ローカル周期ハンドラ ID
           ID      maxalmid;    最大ローカルアラームハンドラ ID
           FN      maxs_fncd;   最大拡張サービスコール機能コード
} T_RCFG;
```

■機能説明

pk_rcfg で示された領域に、システムコンフィギュレーション情報を返します。
pk_rcfg の指すパケットには、次の情報を返します。

◆ maxtskid：最大ローカルタスク ID（_MAX_TSK）
◆ ststkid：スタティックスタックを使用する最大ローカルタスク ID（_MAX_STTSK）
◆ maxsemid：最大ローカルセマフォ ID（_MAX_SEM）
◆ maxflgid：最大ローカルイベントフラグ ID（_MAX_FLAG）
◆ maxdtqid：最大ローカルデータキューID（_MAX_DTQ）
◆ maxmbxid：最大ローカルメールボックス ID（_MAX_MBX）
◆ maxmtxid：最大ローカルミューテックス ID（_MAX_DTQ）
◆ maxmbfid：最大ローカルメッセージバッファ ID（_MAX_MBF）
◆ maxmplid：最大ローカル可変長メモリプール ID（_MAX_MPL）
◆ maxmpfid：最大ローカル固定長メモリプール ID（_MAX_MPF）
◆ maxcycid：最大ローカル周期ハンドラ ID（_MAX_CYH）
◆ maxalmid：最大ローカルアラームハンドラ ID（_MAX_ALH）
◆ maxs_fncd：最大拡張サービスコール機能コード（_MAX_FNCD）

T_RCFG 構造体のメンバはすべて、μITRON4.0 仕様の範囲外です。なお、μITRON4.0 仕様では、T_RCFG 構造体の内容は規定されていません。

RENESAS

6.29.4 バージョン情報の参照(ref_ver, iref_ver)

■C 言語 API

```
ER ercd = ref_ver(T_RVER *pk_rver);
ER ercd = iref_ver(T_RVER *pk_rver);
```

■引数

pk_rver　　バージョン情報を返すパケットへのポインタ

■リターン値

正常終了（E_OK）

■パケットの構造

```
typedef    struct {
           UH      maker;       メーカー
           UH      prid;        形式番号
           UH      spver;       仕様書バージョン
           UH      prver;       製品バージョン
           UH      prno[4];     製品管理情報
} T_RVER;
```

■機能説明

現在実行中のカーネルのバージョンに関する情報を読み出し、その結果を pk_rver の指す領域に返します。

pk_rver の指すパケットには、次の情報を返します。

(1) maker

maker は、このカーネルを作ったメーカを表します。本カーネルでは、ルネサスを意味する 0x0115 です。

(2) prid

prid は、カーネルや VLSI の種類を区別する番号を表します。本カーネルでは、0x0013 が返ります。

(3) spver

spver は、カーネルの準拠する仕様を表しており、ビット対応に意味を持っています。

- ◆ ビット 15～12：MAGIC（TRON 仕様のシリーズを区別する番号）
  本カーネルでは、0x5（μITRON仕様）です。
- ◆ ビット 11～0：SpecVer（製品の元となった TRON 仕様書のバージョン番号）
  本カーネルでは、0x403（μITRON4.0仕様 Ver.4.03.00）です。

(4) prver

prver は、カーネルのバージョン番号を表します。

prver の値は、製品バージョンによって異なります。製品添付のリリースノートを参照してください。例えば、V.1.00 Release 00 の prver は、0x0100 となります。

(5) prno

prno は、製品管理情報や製品番号などを表します。

本カーネルの prno[0]から prno[3]の値は 0x0000 です。

6.30 プロファイル管理機能

表 6.44にプロファイル管理でサポートしているサービスコール一覧を示します。

表6.44 プロファイル管理サービスコール

項番	サービスコール *1		機 能	呼び出し可能なシステム状態 *2						
				T	N	E	D	U	L	C
1	vref_prf	[R]	プロファイルカウンタの参照	○		○	△	○		
	ivref_prf				○	○	○	○		
2	vclr_prf	[R]	プロファイルカウンタのクリア	○		○	△	○		
	ivclr_prf				○	○	○	○		

【注】 *1 "[S]"はスタンダードプロファイルのサービスコール、"[s]"はスタンダードプロファイルではありませんが、スタンダードプロファイルの機能を使用するために必要となるサービスコールです。
“[B]”はベーシックプロファイルのサービスコールです。
“[R]”は、リモート呼び出し可能なサービスコールです。
*2 それぞれの記号は、以下の意味です。
"T"はタスクコンテキストから呼び出し可能、"N"は非タスクコンテキストから呼び出し可能
"E"はディスパッチ許可状態から呼び出し可能、"D"はディスパッチ禁止状態から呼び出し可能
"U"は CPU ロック解除状態から発行可能、"L"は CPU ロック状態から呼び出し可能
“C”はノーマル CPU 例外ハンドラ実行状態から呼び出し可能
“○”はその状態から呼び出し可能、”△”は対象がローカルオブジェクトの場合のみその状態から呼び出し可能

プロファイル機能では、「全体」、「各タスク」、および「カーネルアイドリング状態」について、それぞれ 32 ビットの「プロファイルカウンタ」を持っています。

TIC_NUME/TIC_DENO ミリ秒の周期で実行されるカーネルタイマ割込み処理では、その時点で実行中のタスク、またはカーネルアイドリングのプロファイルカウンタと、全体プロファイルカウンタを加算(+1)します。

つまり、正確ではありませんが、長時間の計測ではおおむね以下の式で各タスクの実行時間を求めることができます。

実行時間[ミリ秒] ＝ 各タスクのプロファイルカウンタ×(TIC_NUME/TIC_DENO)

また、各タスクのプロファイルカウンタやカーネルアイドリングのプロファイルカウンタ値を全体のプロファイルカウンタで除することで、各タスクやカーネルアイドリングの CPU 占有率を算出することができます。

vsta_knl では、全プロファイルカウンタを 0 に初期化します。タスクのプロファイルカウンタは、タスク削除時にも 0 に初期化されます。

なおカーネルはプロファイルカウンタのオーバフロー検出を行わないので注意してください。例えば、TIC_NUME/TIC_DENO=1 ミリ秒の場合は 50 日弱、TIC_NUME/TIC_DENO=10 マイクロ秒の場合は 12 時間強でカウンタがオーバフローする可能性があります。

また、コンフィギュレーションでこれらのサービスコールを選択しない場合でも、プロファイル取得は行われ、デバッギングエクステンションでプロファイル情報を表示することができます。

6.30.1 プロファイルカウンタの参照(vref_prf, ivref_prf)

■C 言語 API

```
ER ercd = vref_prf (ID tskid, UW *p_count, UW *p_allcount);
ER ercd = ivref_prf (ID tskid, UW *p_count, UW *p_allcount);
```

■引数

tskid	タスク ID
p_count	tskid に対するプロファイルカウンタを返す記憶域へのポインタ
p_allcount	全体のプロファイルカウンタを返す記憶域へのポインタ

■リターン値

正常終了（E_OK）、またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)<VKNL_IDLE, VKNL_IDLE<GET_LOCALID(tskid)<0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid)) (3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_CTX	[k]	コンテキストエラ (GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し)
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)(GET_LOCALID(tskid)>0 の場合のみ)

■機能説明

tskid で指定されたタスクのプロファイル情報を取得し、p_count の指す領域に返します。

tskid のローカル ID に VKNL_IDLE を指定すると、指定した CPUID のカーネルアイドリング指定となります。

タスクコンテキストからの呼び出しで tskid に TSK_SELF(0)を指定すると自 CPU の自タスク指定となります。

また、p_allcount の指す領域に、全体のプロファイルカウンタを返します。

p_count および p_allcount の内容がオーバフローしたのかを知る手段はありません。

vref_prf では、ディスパッチ保留状態以外の場合のみ、tskid の CPUID に他 CPUID を指定することができます。一方、ivref_prf では、tskid の CPUID に他 CPUID を指定することはできません。

6.30.2 プロファイルカウンタのクリア(vclr_prf, ivclr_prf)

■C 言語 API

```
ER ercd = vclr_prf (ID tskid);
ER ercd = ivclr_prf (ID tskid);
```

■引数

tskid　　タスク ID

■リターン値

正常終了（E_OK）またはエラーコード

■エラーコード

E_ID	[p]	不正 ID 番号 (1)CPUID が不正(GET_CPUID(tskid)が不正) (2)ローカル ID 範囲外 (GET_LOCALID(tskid)<VCTX_ALL, VKNL_IDLE<GET_LOCALID(tskid)<0, (GET_CPUID(tskid)の_MAX_TSK)<GET_LOCALID(tskid)) (3)非タスクコンテキストからの呼び出しで、tskid=TSK_SELF(0)
E_CTX	[k]	GET_CPUID(tskid)が他 CPU で、許可されていない状態からの呼び出し
E_NOEXS	[k]	未登録(tskid のタスクが存在しない)(GET_LOCALID(tskid)>0 の場合のみ)

■機能説明

tskid で指定されたタスクのプロファイルカウンタを 0 クリアします。

tskid のローカル ID に VKNL_IDLE を指定すると、指定した CPUID のカーネルアイドリングのプロファイルカウンタを 0 クリアします。

tskid のローカル ID に VTSK_ALL を指定すると、指定した CPUID の全タスクのプロファイルカウンタを 0 クリアします。

tskid のローカル ID に VCTX_ALL を指定すると、指定した CPUID のすべてのプロファイルカウンタ(全体、全タスク、カーネルアイドリング)を 0 クリアします。

タスクコンテキストからの呼び出しで tskid に TSK_SELF(0)を指定すると自 CPU の自タスク指定となります

vclr_prf では、ディスパッチ保留状態以外の場合のみ、tskid の CPUID に他 CPUID を指定することができます。一方、ivclr_prf では、tskid の CPUID に他 CPUID を指定することはできません。

6.31 マクロ

6.31.1 定数マクロ

(1) エラーコード

エラーコードは、itron.h で定義されています。以下に、その内容を示します。

```
#define E_OK            0L          /* normal end                    */

/*---- internal error class ----*/
#define E_SYS           (-5L)       /* system error                  */

/*---- no support error class ----*/
#define E_NOSPT         (-9L)       /* no support function           */
#define E_RSFN          (-10L)      /* reserved function code number    */
#define E_RSATR         (-11L)      /* reserved attribute code number   */

/*---- parameter error class ----*/
#define E_PAR           (-17L)      /* parameter error               */
#define E_ID            (-18L)      /* reserved id number            */

/*---- context error class ----*/
#define E_CTX           (-25L)      /* context error                 */
#define E_MACV          (-26L)      /* memory access violation       */
#define E_OACV          (-27L)      /* object access violation       */
#define E_ILUSE         (-28L)      /* service call illegal use      */

/*---- resource insufficiency error class ----*/
#define E_NOMEM         (-33L)      /* no memory                     */
#define E_NOID          (-34L)      /* no ID                         */

/*---- object status error class ----*/
#define E_OBJ           (-41L)      /* object status error           */
#define E_NOEXS         (-42L)      /* object non existent           */
#define E_QOVR          (-43L)      /* queuing over flow             */

/*---- wait release error class ----*/
```

```
#define E_RLWAI        (-49L)        /* wait status forced release     */
#define E_TMOUT        (-50L)        /* time out                       */
#define E_DLT          (-51L)        /* delete object                  */

/*---- other errors ---*/
#define EV_NOINIT      (-97L)        /* not initialized                */
#define EV_NORESOURCE  (-98L)        /* no resource                    */
#define EV_OBJ         (-99L)        /* kernel busy                    */
```

(2) 一般

定義名	定義場所	定義値	説明
NULL	itron.h	C 標準インクルードファイル stddef.h の定義と同じです。NULL が定義されていない場合のみ定義されます。	
TRUE		1	BOOL 型データの「真」
FALSE		0	BOOL 型データの「偽」

(3) オブジェクト属性

定義名	定義場所	定義値	説明
TA_NULL	itron.h	0UL	オブジェクト属性の指定なし
TA_HLNG	kernel.h	0x00000000UL	高級言語用インタフェースでプログラムを起動 *1
TA_ASM		0x00000001UL	アセンブリ言語用インタフェースでプログラムを起動 *1
TA_COP1		0x00000200UL	FPU を使用
TA_TFIFO		0x00000000UL	タスクの待ち行列は FIFO 順
TA_TPRI		0x00000001UL	タスクの待ち行列はタスク優先度順
TA_MFIFO		0x00000000UL	メッセージキューは FIFO 順
TA_MPRI		0x00000002UL	メッセージキューはメッセージ優先度順
TA_ACT		0x00000002UL	タスクを起動された状態で生成
TA_WSGL		0x00000000UL	複数のタスクが待つことを許さないイベントフラグ
TA_WMUL		0x00000002UL	複数のタスクが待つことを許すイベントフラグ
TA_CLR		0x00000004UL	待ち解除時にイベントフラグをクリア
TA_CEILING		0x00000003UL	ミューテックスの優先度上限プロトコル
TA_STA		0x00000002UL	周期ハンドラを動作している状態で生成
TA_PHS		0x00000004UL	周期ハンドラの位相を保存
VTA_REGBANK		0x40000000UL	レジスタバンクを使用するノーマル割込みハンドラ
VTA_DIRECT		0x80000000UL	ダイレクト割込みハンドラ、またはダイレクト CPU 例外ハンドラ
VTA_UNFRAGMENT		0x80000000UL	可変長メモリプールのセクタ管理指定

【注】 *1 本カーネルでは、TA_HLNG と TA_ASM のどちらを指定しても同じ動作となります。

(4) タイムアウト指定

定義名	定義場所	定義値	説明
TMO_POL	itron.h	0L	ポーリング
TMO_FEVR		-1L	永久待ち

(5) サービスコールの動作モード

定義名	定義場所	定義値	説明
TWF_ANDW	kernel.h	0x00000000UL	イベントフラグの AND 待ち
TWF_ORW		0x00000001UL	イベントフラグの OR 待ち

(6) タスクの状態

定義名	定義場所	定義値	説明
TTS_RUN	kernel.h	0x00000001UL	実行状態
TTS_RDY		0x00000002UL	実行可能状態
TTS_WAI		0x00000004UL	待ち状態
TTS_SUS		0x00000008UL	強制待ち状態
TTS_WAS		0x0000000cUL	二重待ち状態
TTS_DMT		0x00000010UL	休止状態
TTS_STK *1		0x40000000UL	スタック待ち状態

【注】 *1 μITRON4.0 仕様外

(7) タスクの待ち要因

定義名	定義場所	定義値	説明
TTW_SLP	kernel.h	0x00000001UL	起床待ち(slp_tsk, tslp_tsk)
TTW_DLY		0x00000002UL	時間経過待ち(dly_tsk)
TTW_SEM		0x00000004UL	セマフォ待ち(wai_sem, twai_sem)
TTW_FLG		0x00000008UL	イベントフラグ待ち(wai_flg, twai_flg)
TTW_SDTQ		0x00000010UL	データキューへの送信待ち(snd_dtq, tsnd_dtq)
TTW_RDTQ		0x00000020UL	データキューからの受信待ち(rcv_dtq, trcv_dtq)
TTW_MBX		0x00000040UL	メールボックスからの受信待ち(rcv_mbx, trcv_mbx)
TTW_MTX		0x00000080UL	ミューテックスロック待ち(loc_mtx, tloc_mtx)
TTW_SMBF		0x00000100UL	メッセージバッファへの送信待ち(snd_mbf, tsnd_mbf)
TTW_RMBF		0x00000200UL	メッセージアッファからの受信待ち(rcv_mbf, trcv_mbf)
TTW_MPF		0x00002000UL	固定長メモリブロック獲得待ち(get_mpf, tget_mpf)
TTW_MPL		0x00004000UL	可変長メモリブロック獲得待ち(get_mpl, tget_mpl)
TTW_TFL *1		0x00008000UL	タスク付属イベントフラグ待ち(vwai_tfl, vtwai_tfl)

【注】 *1 μITRON4.0 仕様外

(8) 各種状態

定義名	定義場所	定義値	説明
TTEX_ENA	kernel.h	0x00000000UL	タスク例外処理許可状態
TTEX_DIS		0x00000001UL	タスク例外処理禁止状態
TCYC_STP		0x00000000UL	周期ハンドラ停止状態
TCYC_STA		0x00000001UL	周期ハンドラ動作状態
TALM_STP		0x00000000UL	アラームハンドラ停止状態
TALM_STA		0x00000001UL	アラームハンドラ動作状態
TOVR_STP		0x00000000UL	上限プロセッサ時間が設定されていない
TOVR_STA		0x00000001UL	上限プロセッサ時間が設定されている

(9) その他

定義名	定義場所	定義値	説明
TSK_SELF	kernel.h	0	自タスク指定
TSK_NONE		0	該当するタスクがない
TPRI_SELF		0	自タスクのベース優先度の指定
TPRI_INI		0	タスクの初期優先度の指定
VCPU_SELF *1		0	自 CPU 指定
VCPU_MAX *1		2	最大 CPUID
VKNL_IDLE *1		-32768	カーネルアイドルング指定
VTSK_ALL *1		-32767	全タスク指定
VCTX_ALL *1		-32766	全コンテキスト指定
ECM_SUS *1		0x00000001UL	強制待ち要求
ECM_TER *1		0x00000002UL	強制終了要求
PND_SUS *1		0x00000004UL	強制待ち要求を遅延中
PND_TER *1		0x00000008UL	強制終了要求を遅延中
SR_IMS00 *1		0x00000000UL	割込みマスクレベル＝0
SR_IMS01 *1		0x00000001UL	割込みマスクレベル＝1
SR_IMS02 *1		0x00000002UL	割込みマスクレベル＝2
SR_IMS03 *1		0x00000003UL	割込みマスクレベル＝3
SR_IMS04 *1		0x00000004UL	割込みマスクレベル＝4
SR_IMS05 *1		0x00000005UL	割込みマスクレベル＝5
SR_IMS06 *1		0x00000006UL	割込みマスクレベル＝6
SR_IMS07 *1		0x00000007UL	割込みマスクレベル＝7
SR_IMS08 *1		0x00000008UL	割込みマスクレベル＝8
SR_IMS09 *1		0x00000009UL	割込みマスクレベル＝9
SR_IMS10 *1		0x0000000aUL	割込みマスクレベル＝10
SR_IMS11 *1		0x0000000bUL	割込みマスクレベル＝11
SR_IMS12 *1		0x0000000cUL	割込みマスクレベル＝12
SR_IMS13 *1		0x0000000dUL	割込みマスクレベル＝13
SR_IMS14 *1		0x0000000eUL	割込みマスクレベル＝14
SR_IMS15 *1		0x0000000fUL	割込みマスクレベル＝15

【注】 *1 μITRON4.0 仕様外

6.31.2　カーネル構成マクロ

カーネル構成マクロの一部は、cfg72mp によって kernel_macro.h、kernel_def.h、kernel_cfg.h に出力されます。

(1) 優先度の範囲

定義名	定義場所	定義値	説明
TMIN_TPRI	kernel.h	1	タスク優先度の最小値
TMAX_TPRI	kernel_macro.h	system.priority	タスク優先度の最大値
TMIN_MPRI	kernel.h	1	メッセージ優先度の最小値
TMAX_MPRI	kernel_macro.h	system.message_pri	メッセージ優先度の最大値

(2) バージョン情報

定義名	定義場所	定義値	説明
TKERNEL_MAKER	kernel.h	0x0115	カーネルのメーカコード
TKERNEL_PRID		0x0013	カーネルの識別番号
TKERNEL_SPVER		0x5403	ITRON 仕様のバージョン番号
TKERNEL_PRVER		*1	カーネルのバージョン番号

【注】 *1　製品添付のリリースノートを参照してください。例えば、V.1.00 Release 00 の prver は、0x0100 となります。

(3) キューイング･ネスト回数の最大値など

定義名	定義場所	定義値	説明
TMAX_ACTCNT	kernel.h	15U	タスク起動要求キューイング数の最大値
TMAX_WUPCNT		15U	タスク起床要求キューイング数の最大値
TMAX_SUSCNT		15U	タスク強制待ち要求ネスト数の最大値
TMAX_SEMCNT		65535U	セマフォの最大資源数の最大値

(4) ビットパターンのビット数

定義名	定義場所	定義値	説明
TBIT_TEXPTN	kernel.h	32U	タスク例外要因のビット数
TBIT_FLGPTN		32U	イベントフラグのビット数

(5) タイムティックの周期

定義名	定義場所	定義値	説明
TIC_NUME	kernel_macro.h	system.tic_nume	タイムティックの周期の分子
TIC_DENO		system.tic_deno	タイムティックの周期の分母

(6) cfg72mp が kernel_macro.h に出力するその他のカーネル構成マクロ（μITRON4.0 仕様外）

(a) カーネル構成マクロ

定義名	定義値	説明
VTCFG_TBR	system.tbr に定義したシンボルの先頭に'_'を付加したマクロに定義されます。	
VTCFG_MPFMANAGE	system.mpfmanage に定義したシンボルの先頭に'_'を付加したマクロに定義されます。	
VTCFG_NEWMPL	system.newmpl に定義したシンボルの先頭に'_'を付加したマクロに定義されます。	
VTCFG_VECTYPE	system.vector_type に定義したシンボルの先頭に'_'を付加したマクロに定義されます。	
VTCFG_REGBANK	system.regbank に定義したシンボルの先頭に'_'を付加したマクロに定義されます。	
TIM_LVL	clock.IPL	タイマ割込み優先レベル

(b) 各種定義で使用する定数マクロ

分類	定義名	定義場所	定義値
共通	_NOTUSE	kernel.h	0UL
VTCFG_TBR 用	_NOMANAGE		0UL
	_FOR_SVC		1UL
	_TASK_CONTEXT		2UL
VTCFG_MPFMANAGE 用	_IN		0UL
	_OUT		1UL
VTCFG_NEWMPL 用	_PAST		0UL
	_NEW		1UL
VTCFG_VECTYPE 用	_ROM_ONLY_DIRECT		0UL
	_RAM_ONLY_DIRECT		1UL
	_ROM		2UL
	_RAM		3UL
VTCFG_REGBANK 用	_ALL		0x40000000UL
	_BANKLEVEL01		0x00000002UL
	_BANKLEVEL02		0x00000004UL
	_BANKLEVEL03		0x00000008UL
	_BANKLEVEL04		0x00000010UL
	_BANKLEVEL05		0x00000020UL
	_BANKLEVEL06		0x00000040UL
	_BANKLEVEL07		0x00000080UL
	_BANKLEVEL08		0x00000100UL
	_BANKLEVEL09		0x00000200UL
	_BANKLEVEL10		0x00000400UL
	_BANKLEVEL11		0x00000800UL
	_BANKLEVEL12		0x00001000UL
	_BANKLEVEL13		0x00002000UL
	_BANKLEVEL14		0x00004000UL
	_BANKLEVEL15		0x00008000UL

(7) cfg72mp が kernel_def.h に出力するカーネル構成マクロ（μITRON4.0 仕様外）

ここでは、cfg72mp が kernel_def.h に出力するカーネル構成マクロのうち、仕様を公開するものについて記載します。ただし、これらのマクロの仕様については、将来との互換性は保証していません。

定義名	定義値	説明
_SYSTEM_IPL	system.system_IPL	カーネル割込みマスクレベル
_MAX_STTSK	maxdefine.max_statictask	スタティックスタックを使用する最大ローカルタスク ID
_MAX_INT	maxdefine.max_int	最大割込みベクタ番号

(8) cfg72mp が kernel_cfg.h に出力するカーネル構成マクロ（μITRON4.0 仕様外）

ここでは、cfg72mp が kernel_cfg.h に出力するカーネル構成マクロのうち、仕様を公開するものについて記載します。ただし、これらのマクロの仕様については、将来との互換性は保証していません。

定義名	定義値	説明
_SYSTEM_STACK_SIZE	system.stack_size	割込みスタックサイズ
_SYSTEM_KERNEL_STACK_SIZE	system.kernel_stack_size	カーネルスタックサイズ
_MAX_TSK	maxdefine.max_task	最大ローカルタスク ID
_MAX_SEM	maxdefine.max_sem	最大ローカルセマフォ ID
_MAX_FLG	maxdefine.max_flag	最大ローカルイベントフラグ ID
_MAX_DTQ	maxdefine.max_dtq	最大ローカルデータキューID
_MAX_MBX	maxdefine.max_mbx	最大ローカルメールボックス ID
_MAX_MTX	maxdefine.max_mtx	最大ローカルミューテックス ID
_MAX_MBF	maxdefine.max_mbf	最大ローカルメッセージバッファ ID
_MAX_MPF	maxdefine.max_mpf	最大ローカル固定長メモリプール ID
_MAX_MPL	maxdefine.max_mpl	最大ローカル可変長メモリプール ID
_MAX_CYH	maxdefine.max_cyh	最大ローカル周期ハンドラ ID
_MAX_ALH	maxdefine.max_alh	最大ローカルアラームハンドラ ID
_MAX_FNCD	maxdefine.max_fncd	最大ローカル拡張サービスコール機能コード
_MEMSTK_ALLMEMSIZE	memstk.all_memsize	デフォルトタスクスタック用領域のサイズ
_MEMDTQ_ALLMEMSIZE	memdtq.all_memsize	デフォルトデータキュー用領域のサイズ
_MEMMBF_ALLMEMSIZE	memmbf.all_memsize	デフォルトメッセージバッファ用領域のサイズ
_MEMMPF_ALLMEMSIZE	memmpf.all_memsize	デフォルト固定長メモリプール用領域のサイズ
_MEMMPL_ALLMEMSIZE	memmpl.all_memsize	デフォルト可変長メモリプール用領域のサイズ

6.31.3 itron.h で定義されている関数マクロ

(1) ER MERCD(ER ercd)

説明	ercd のメインエラーコードを返します。	
ヘッダ	itron.h	
引数	ercd	エラーコード
リターン値	ercd のメインエラーコード	

(2) ER SERCD(ER ercd)

説明	ercd のサブエラーコードを返します。	
ヘッダ	itron.h	
引数	ercd	エラーコード
リターン値	ercd のサブエラーコード	
備考	カーネルが返すエラーコードのサブエラーコードは全て-1 です。	

(3) ER ERCD(ER mercd, ER sercd)

説明	mercd のメインエラーコードと sercd のサブエラーコードからなるエラーコードを返します。	
ヘッダ	itron.h	
引数	mercd	メインエラーコード
	sercd	サブエラーコード
リターン値	エラーコード	
備考	カーネルが返すエラーコードのサブエラーコードは全て-1 です。	

6.31.4 kernel.h で定義されている関数マクロ

(1) UH GET_CPUID(ID id)

説明	id の CPUID を返します。	
ヘッダ	kernel.h	
引数	id	オブジェクト ID
リターン値	CPUID	
備考	本マクロはμITRON4.0 仕様外の機能です。	

(2) ID GET_LOCALID(ID id)

説明	id のローカルオブジェクト ID を返します。	
ヘッダ	kernel.h	
引数	id	オブジェクト ID
リターン値	ローカルオブジェクト ID	
備考	本マクロはμITRON4.0 仕様外の機能です。	

(3) ID MAKE_ID(UH cpuid, ID localid)

説明	cpuid の CPUID と localid のローカルオブジェクト ID からなるオブジェクト ID を返します。
ヘッダ	kernel.h
引数	cpuid　　CPUID
	localid　　ローカルオブジェクト ID
リターン値	オブジェクト ID
備考	本マクロはμITRON4.0 仕様外の機能です。

(4) SIZE TSZ_DTQ(UINT dtqcnt)

説明	データを dtqcnt 個格納可能なデータキューのサイズを返します。
ヘッダ	kernel.h
引数	dtqcnt　　データ数
リターン値	データキューのサイズ

(5) SIZE TSZ_MBF(UINT msgcnt, UINT msgsz)

説明	サイズが msgsz バイトのメッセージを msgcnt 個格納可能なメッセージバッファの目安のサイズを返します。
ヘッダ	kernel.h
引数	msgcnt　　メッセージ数
	msgsz　　メッセージのサイズ
リターン値	メッセージバッファの目安のサイズ

(6) SIZE TSZ_MPF(UINT blkcnt, UINT blksz)

説明	サイズが blksz バイトのメモリブロックを blkcnt 個獲得可能な固定長メモリプールのサイズを返します。
ヘッダ	kernel.h
引数	blkcnt　　メモリブロック数
	blksz　　メモリブロックのサイズ
リターン値	固定長メモリプールのサイズ
備考	本マクロが返す値は、system.mpfmanage によって変わります。

(7) SIZE VTSZ_MPFMB(UINT blkcnt, UINT blksz)

説明	サイズが blksz バイトのメモリブロックを blkcnt 個獲得可能な固定長メモリプールのために必要な管理領域のサイズを返します。
ヘッダ	kernel.h
引数	blkcnt　　メモリブロック数
	blksz　　メモリブロックのサイズ
リターン値	固定長メモリプール管理領域のサイズ
備考	本マクロは、system.mpfmanage==OUT の場合のみ定義されます。 また、本マクロはμITRON4.0 仕様外の機能です。

(8) SIZE TSZ_MPL(UINT blkcnt, UINT blksz)

説明	サイズが blksz バイトのメモリブロックを blkcnt 個獲得可能な可変長メモリプールの目安のサイズを返します。	
ヘッダ	kernel.h	
引数	blkcnt	メモリブロック数
	blksz	メモリブロックのサイズ
リターン値	可変長メモリプールの目安のサイズ	
備考	本マクロが返す値は、system.newmpl によって変わります。	

6.32 ディレクトリ・ファイル構成

<RTOS_INST>¥os¥include¥	
itron.h	ITRON 仕様定義
kernel.h	カーネル仕様定義
kernel_api.h	サービスコール API 定義
kernel_dbg.h	デバッグ機能 API 定義
sh2afpu.h	ハンドラでの FPU を使用するためのヘッダ
<RTOS_INST>¥os¥lib¥release¥	
hiknl.lib	ベースライブラリ（デバッグ情報なし）
fpu_knl.lib	FPU サポート差分ライブラリ（デバッグ情報なし）
hiexpand.lib	デバッグ用 FPU 未サポート差分ライブラリ（デバッグ情報なし）
fpu_expand.lib	デバッグ用 FPU サポート差分ライブラリ（デバッグ情報なし）
<RTOS_INST>¥os¥system¥	システム定義

以降は、ソースコード付き製品の場合のみ提供されます。

<RTOS_INST>¥os¥lib¥debug¥	
hiknl.lib	ベースライブラリ（デバッグ情報付き）
fpu_knl.lib	FPU サポート差分ライブラリ（デバッグ情報付き）
hiexpand.lib	デバッグ用 FPU 未サポート差分ライブラリ（デバッグ情報付き）
fpu_expand.lib	デバッグ用 FPU サポート差分ライブラリ（デバッグ情報付き）
<RTOS_INST>¥os¥kernel¥	ライブラリ生成用ワークスペース等
<RTOS_INST>¥os¥kernel¥knl_src¥	ソースコード
<RTOS_INST>os¥kernel¥fpu_expand¥	プロジェクトディレクトリ
<RTOS_INST>os¥kernel¥fpu_expand¥debug¥	コンフィギュレーションディレクトリ(デバッグ情報付き)
<RTOS_INST>os¥kernel¥fpu_expand¥release¥	コンフィギュレーションディレクトリ(デバッグ情報なし)
<RTOS_INST>os¥kernel¥fpu_knl¥	プロジェクトディレクトリ
<RTOS_INST>os¥kernel¥fpu_knl¥debug¥	コンフィギュレーションディレクトリ(デバッグ情報付き)
<RTOS_INST>os¥kernel¥fpu_knl¥release¥	コンフィギュレーションディレクトリ(デバッグ情報なし)
<RTOS_INST>os¥kernel¥hiexpand¥	プロジェクトディレクトリ
<RTOS_INST>os¥kernel¥hiexpand¥debug¥	コンフィギュレーションディレクトリ(デバッグ情報付き)
<RTOS_INST>os¥kernel¥hiexpand¥release¥	コンフィギュレーションディレクトリ(デバッグ情報なし)
<RTOS_INST>os¥kernel¥hiintfc¥	プロジェクトディレクトリ
<RTOS_INST>os¥kernel¥hiintfc¥debug¥	コンフィギュレーションディレクトリ(デバッグ情報付き)
<RTOS_INST>os¥kernel¥hiintfc¥release¥	コンフィギュレーションディレクトリ(デバッグ情報なし)
<RTOS_INST>os¥kernel¥hiknl¥	プロジェクトディレクトリ
<RTOS_INST>os¥kernel¥hiknl¥debug¥	コンフィギュレーションディレクトリ(デバッグ情報付き)

<RTOS_INST>os¥kernel¥hiknl¥release¥	コンフィギュレーションディレクトリ(デバッグ情報なし)
<RTOS_INST>os¥kernel¥intdwn¥	プロジェクトディレクトリ
<RTOS_INST>os¥kernel¥intdwn¥debug¥	コンフィギュレーションディレクトリ(デバッグ情報付き)
<RTOS_INST>os¥kernel¥intdwn¥release¥	コンフィギュレーションディレクトリ(デバッグ情報なし)
<RTOS_INST>os¥kernel¥svcapi¥	プロジェクトディレクトリ
<RTOS_INST>os¥kernel¥svcapi¥debug¥	コンフィギュレーションディレクトリ(デバッグ情報付き)
<RTOS_INST>os¥kernel¥svcapi¥release¥	コンフィギュレーションディレクトリ(デバッグ情報なし)
<RTOS_INST>os¥kernel¥svcrmt¥	プロジェクトディレクトリ
<RTOS_INST>os¥kernel¥svcrmt¥debug¥	コンフィギュレーションディレクトリ(デバッグ情報付き)
<RTOS_INST>os¥kernel¥svcrmt¥release¥	コンフィギュレーションディレクトリ(デバッグ情報なし)

6.33 ライブラリのビルド(ソースコード付き製品の場合のみ)

通常はライブラリをビルドする必要はありません。デバッグなどの目的でライブラリをビルドする場合は、提供する High-performance Embedded Workshop ワークスペース(kernel.hws)を用いてビルドしてください。

7. RPC ライブラリ

7.1 概要

RPC は、別 CPU にあらかじめ登録された関数を実行させる機能です。図 7.1に、RPC の概念図を示します。

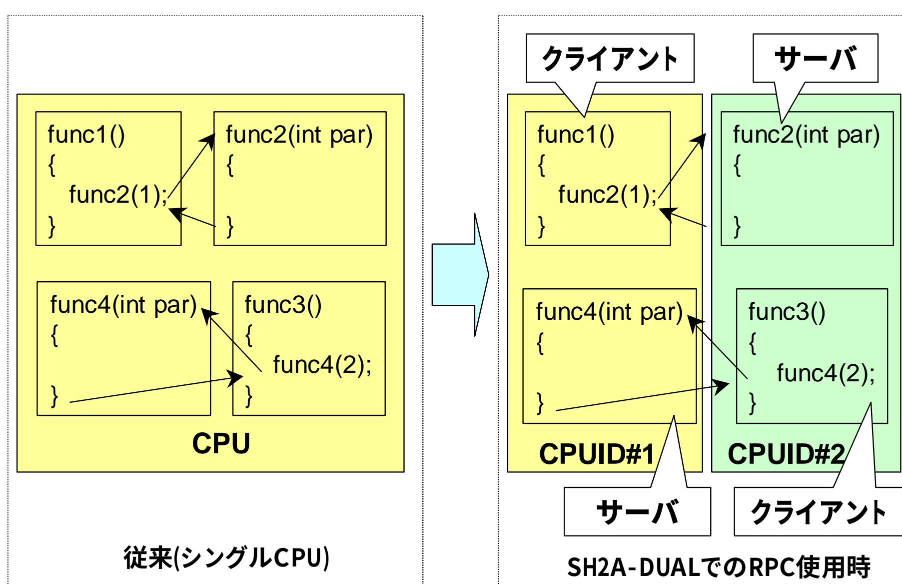


図7.1 RPC の概念図

RPC では、関数の呼び出し側をクライアント、関数を実行する側をサーバと呼びます。

また、RPC では自 CPU のサーバを呼び出すこともできます。これにより、サーバ単位での(静的な)CPU 移動が比較的容易に行えるようになっています。ただし、自 CPU サーバの呼び出しは、当然ながら通常の関数呼び出しよりも遅くなる点は、注意が必要です。

表7.1 RPC ライブラリ概要

項番	項目	内容
1	使用するハードウェアリソース	なし
2	本ソフトウェアが使用するソフトウェア部品	(1)OAL (2)IPI (3)スピンロックライブラリ
3	本ソフトウェアを使用している他のソフトウェア部品	なし

RENESAS

7.2 RPC の動作概要

図 7.2を用いて、RPC の構成と、クライアント側が CPUID#1、サーバ側が CPUID#2 の場合の動作を説明します。

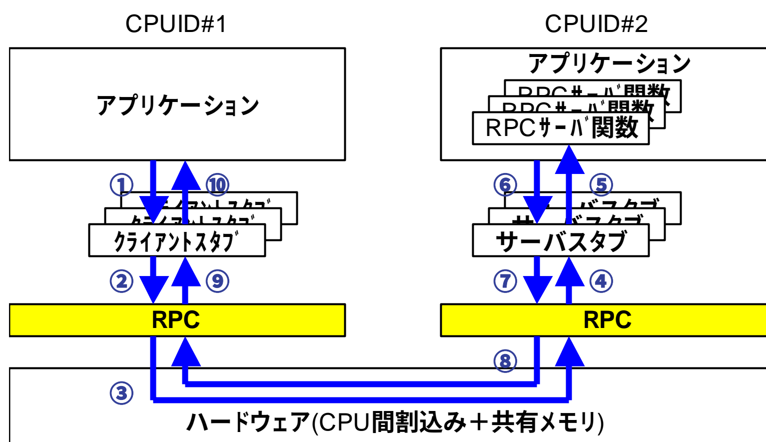


図7.2　RPC の構成と動作

① CPUID#1側のアプリケーションがCPUID#2側にRPCサーバ関数を実行させるため、RPCサーバ関数と同じAPIを使って関数コールを行います。

② CPUID#1側では､RPCサーバ関数と同名･同APIを持つクライアントスタブが呼び出されます。クライアントスタブでは、入出力パラメータをRPCが解釈できる形式に変換し、RPCのリモート関数コールAPI(rpc_call)を呼び出します。なお、クライアントスタブはサーバ作成者が実装する必要があります。

③ CPUID#1側のrpc_call()は、呼び出されたタスクと同じコンテキストで実行します。rpc_call()は、渡された情報をサーバの領域に転送し、IPIプリミティブを用いてCPUID#2側のサーバタスクを起床します。rpc_call()呼び出し元は、RPCサーバ関数の実行完了までrpc_call()内で待ち状態になります。
サーバの領域は非キャッシャブルな共有メモリに配置します。

④ CPUID#2側のサーバタスクは、要求に対応したサーバスタブを呼び出します。サーバスタブは、ユーザが実装する必要があり、あらかじめ登録しておく必要があります。また、サーバスタブは、サーバ毎に定義されるサーバタスクのコンテキストで実行します。
なお、サーバタスクは事前にサーバ開始のAPI(rpc_start_server())によって生成・起動されます。

⑤ サーバスタブには、③で転送された入力パラメータなどが渡されます。サーバスタブは、RPCから渡された情報をもとに、RPCサーバ関数のインタフェース仕様に入力パラメータを整形し、RPCサーバ関数を呼び出します。なお、サーバスタブはサーバ作成者が実装する必要があります。

⑥ RPCサーバ関数からリターンします。

⑦ サーバスタブは、出力情報を設定し、サーバタスクにリターンします。

⑧ サーバタスクは、IPIプリミティブを用いて、③で待機しているクライアントタスクを起床します。

⑨ 起床されたクライアントタスクは、rpc_call()の処理として⑦で設定された出力情報をクライアント領域に転送し、クライアントスタブにリターンします。

⑩ クライアントスタブは、リターン情報の設定などを行い、呼び出し元にリターンします。

7.3 サーバ

7.3.1 サーバ ID

サーバ ID は、サーバを識別するための情報で、各サーバの ID はシステム全体でユニークでなければなりません。

サーバ ID は、サーバを開始するときに指定します。通常、サーバ ID は、システム全体で重複しないように、システム設計者が静的に決定しておく必要があります。

サーバ ID は符号なし 32 ビット整数として表現します。値としての制限はありませんので、重複しない規則を設定しやすいようになっています。

ただし、0x80000000 以上のサーバ ID は将来 OS 側で使用する可能性があるため、使用しないでください。

一方、各 CPU に登録可能なサーバ数の上限は、RPC ライブラリを開始する rpc_init()で指定します。

7.3.2 ファンクション ID

サーバは、ひとつ以上のサーバ関数を持ちます。各サーバ関数は、0 から始まる連番のファンクション ID によって識別されます。サーバコール(rpc_call)時には、ファンクション ID を指定します。

サーバは、最大 32767 までのファンクション ID を持つことができます。

7.3.3 サーバタスク

サーバスタブ、およびサーバ関数は、サーバタスクのコンテキストで実行します。すなわち、サーバスタブはサーバタスクから呼び出されます。

サーバタスクのコード実体は RPC ライブラリ内にあり、サーバ開始時にサーバタスクが生成されます。サーバタスクの優先度とスタックサイズは、サーバ開始時に指定できます。

7.3.4 サーバスタブとサーバ関数

サーバスタブは、RPC ライブラリ内のサーバタスクから呼び出されます。サーバスタブには、RPC の仕様で決められた形式でサーバ関数へのパラメータ情報が渡されます。サーバスタブは、これを目的のサーバ関数の API 仕様に変換して呼び出します。

サーバスタブは、サーバ作成者が実装してください。

7.3.5 クライアントスタブ

クライアントスタブは、目的の関数と同一 API 仕様を持ち、パラメータを RPC の仕様で決められた形式に変換して RPC コールします。

クライアントスタブは、サーバ作成者が実装してください。

7.3.6 サーバの競合

本 RPC では、クライアントーサーバ間のパラメータ転送領域の管理を簡略化してオーバヘッドを小さくするために、各サーバについてサーバタスクをひとつだけ用意する仕様としています。

これは、サーバは同時にはひとつの RPC コール要求しか処理できないことを意味します。

同じサーバに対して、同時に複数の RPC コールが要求された場合、サーバはそれらを呼び出された順番で処理します。処理待ちのクライアントは、ブロックされることになります。

7.4 同期モードと非同期モード

本 RPC では、コール方法として、同期モード(RPC_ACK)と非同期モード(RPC_UNACK)をサポートしています。どちらのモードで呼び出すかは、RPC コール時に指定します。

同期モードでの RPC コールでは、サーバの実行が終了するまでクライアントタスクはブロックされます。クライアントは、サーバからの出力を取得することができます。

非同期モードでの RPC コールでは、サーバ実行を依頼すると直ちにリターンします。非同期モードでは、クライアントは、サーバからの出力を取得することはできません。また、クライアントがサーバの処理完了を認識する方法はありません。

ただし、いずれの場合も「7.3.6 サーバの競合」に記載のケースでは、クライアントはブロックされる可能性があることに注意してください。

7.5 パラメータの受け渡し

7.5.1 特長

- ◆ RPC 自身は、パラメータの受け渡しのためのバッファは持ちません。
- ◆ サーバは、クライアントからの入力パラメータ、およびクライアントへの出力パラメータを格納するための「サーバパラメータ領域」を持ちます。サーバパラメータ領域は、非キャッシャブル領域に配置しなければなりません。
- ◆ サーバパラメータ領域は、静的にサーバ作成者が確保する方法と、オンデマンドに RPC が自動的に確保する方法があります。
- ◆ クライアントが指定した入力パラメータは、直接サーバパラメータ領域にコピーされます。同様に、サーバの出力パラメータは、直接クライアントが指定した領域にコピーされます。つまり、各々コピーは 1 回ずつです。
- ◆ コピー方法として、クライアント側の RPC ライブラリが CPU 命令によってコピーする方法と、クライアント側の RPC ライブラリからコールバックされるユーザ作成関数でコピーする方法があります。後者の機能を用いると、パラメータを DMA 転送することもできます。

7.5.2 IOVEC 構造体

本 RPC では、入力パラメータ(クライアント→サーバ)、および出力パラメータ(サーバ→クライアント)それぞれについて、IOVEC と呼ぶ構造体の配列でパラメータを指定します。これにより、非連続なメモリに分散されたパラメータを効率的に扱えるようになっています。

```
typedef struct {
    void *pBaseAddress;     // データ領域の先頭アドレス
    UINT32 ulSize;          // そのサイズ(バイト数)
} IOVEC;
```

なお、ulSize=0 は領域なしを意味します。

RENESAS

7.5.3 サーバパラメータ領域

サーバは、クライアントからの入力パラメータを格納するため、およびクライアントへの出力パラメータを格納するためのパラメータ領域を持ちます。本 RPC では、サーバは同時にはひとつのクライアントの要求しか処理できないようになっているため、サーバパラメータ領域はサーバ毎にひとつだけ存在します。

サーバパラメータ領域の確保方法として、オンデマンドに領域を確保する方法と、スタティックに確保された領域を使用する方法があります。前者を「ダイナミックサーバ」と呼び、rpc_start_server()を用いて開始します。後者を「スタティックサーバ」と呼び、rpc_start_server_with_paramarea()を用いて開始します。

いずれの場合も、サーバ・クライアント間のコヒーレンシ維持のため、サーバ領域は非キャッシャブル領域でなければなりません。

(1) ダイナミックサーバ

ダイナミックサーバは、rpc_start_server()を用いて開始します。

RPC コールでダイナミックサーバを呼び出すと、サーバは OAL_GetMemory()を用いてその呼び出しに必要なサイズの領域を確保します。この領域は、その呼び出しが完了したときに解放されます。

OAL は、OAL_GetMemory()で取得されるメモリが非キャッシャブル領域になるようにコンフィギュレーションする必要があります。

(2) スタティックサーバ

スタティックサーバは、rpc_start_server_with_paramarea()を用いて開始します。

rpc_start_server_with_paramarea()では、サーバパラメータ領域のアドレスとサイズを指定します。サーバパラメータ領域はアプリケーション側で確保します。

ダイナミックサーバは、入出力パラメータサイズが一定でない場合にメモリを効率的に利用できるという特徴がありますが、動的にメモリの取得・解放を行うため、処理時間はスタティックサーバよりも劣ります。

7.5.4 RPC コールに必要なサーバパラメータ領域のサイズ

RPC コールを要求するには、rpc_call()または rpc_call_copycbk()を使用します。
RPC コールを受理するために必要なサーバパラメータ領域のサイズは、以下の計算式で算出されます。

必要サイズ＝sizeof(rpc_server_stub_info)
＋ΣALIGNUP4(pCallInfo->pInputIOVectorTable->ulSize)
＋ (pCallInfo->ulOutputIOVectorTableSize)×sizeof(IOVEC) … (a)
＋ΣALIGNUP4 (pCallInfo->pOutputIOVectorTable->ulSize) … (b)

ただし、非同期モード指定時は、(a),(b)は共に 0 として算出します。
なお、ALIGNUP4 (data)は、data を 4 の倍数に切り上げた値を意味し、types.h で関数マクロとして定義されています。

(1) ダイナミックサーバの場合

RPC コールが要求されたとき、サーバは上式で算出されるサイズのメモリを OAL_GetMemory()を用いて取得します。これに失敗した場合は、rpc_call()は RPC_E_NOMEM エラーでリターンします。
ただし、rpc_start_server()で ulMaxParamAreaSize に 0 以外を指定した場合で、rpc_call()での上記要求サイズが ulMaxParamAreaSize を超える場合は、サーバは OAL_GetMemory()を行わず、RPC コールは RPC_E_PAR エラーでリターンします。

(2) スタティックサーバの場合

上式で算出されるサイズが rpc_start_server_with_paramarea()で指定した ulMaxParamAreaSize を超える場合は、RPC コールは RPC_E_PAR エラーでリターンします。

7.5.5 パラメータのコピー方法

(1) rpc_call()を用いた場合

入力パラメータ、出力パラメータともに、クライアント側の rpc_call()によってコピーされます。
具体的には、クライアントが指定した入力パラメータは、クライアント側の rpc_call()によってサーバパラメータ領域にコピーされます。また、サーバスタブがサーバパラメータ領域に生成した出力パラメータも、クライアント側の rpc_call()によって、クライアントが指定した出力パラメータ領域にコピーされます。

(2) rpc_call_copycbk()を用いた場合

rpc_call_copycbk()では、コピー処理を行うためのコールバック関数を指定します。コールバック関数は２つ指定します。それぞれ、入力パラメータをサーバパラメータ領域にコピーする必要が生じたとき、および出力パラメータをサーバパラメータ領域からクライアントが指定した出力パラメータ領域にコピーする必要が生じたときに、クライアント側の rpc_call_copycbk()からコールバックされます。

7.5.6 応用例

(1) コピーオーバヘッドの削減

メモリ空間を共有するマルチプロセッサシステムでは、そのポインタだけを受け渡すようにすれば、コピーのオーバヘッドを小さくすることができます。

ただし、SH2A-DUAL のように、キャッシュスヌープコントローラを持たないマルチコアシステムでは、クライアント CPU とサーバ CPU のキャッシュ間のコヒーレンシを保証することはできません。このため、そのポインタの指す領域を非キャッシャブル領域としなければなりません。

(2) キャッシュの活用

サーバ側は、サーバパラメータ領域から入力パラメータを取得したり、出力パラメータを設定しますが、サーバパラメータ領域は非キャッシャブル領域であるため、サーバによるこれらパラメータへのアクセス頻度によっては、キャッシュされないことによる性能劣化が問題となる可能性があります。

このような場合は、サーバはいったんキャッシャブル領域に確保した領域にパラメータを転送してください。

7.6 RPC が使用する OS リソース

7.6.1 タスク

サーバ毎にサーバタスクがあります。

サーバタスクは、rpc_start_server(), rpc_start_server_with_paramarea()によって生成され、rpc_stop_server()によって削除されます。サーバタスクの優先度と使用するスタックサイズは、rpc_start_server(), rpc_start_server_with_paramarea()で指定します。

サーバタスクのエントリの関数は、RPC ライブラリ内に存在し、この関数がサーバスタブを呼び出します。

なお、RPC の API はすべて単なる関数として実行されます。すなわち、呼び出し元のコンテキストで実行されます。

7.6.2 OAL_GetMemory()

RPC では、以下の場合に OAL_GetMemory()を用いてメモリを取得します。OAL_GetMemory()で取得するメモリは非キャッシャブル領域となるように、OAL をコンフィギュレーションする必要があります。

(1) ダイナミックサーバによるパラメータ領域の取得

ダイナミックサーバに対して rpc_call(), rpc_call_copycbk()が要求されたとき、サーバタスクはそのパラメータ領域を OAL_GetMemory()を用いて取得します。

(2) サーバ呼び出し待ち

rpc_call(), rpc_call_copycbk()で対象サーバが空くのを待つ場合、その待ち状態を管理するためのメモリ領域を OAL_GetMemory()を用いて取得します。

7.6.3 IPI

RPC では、IPI ポートをひとつ使用します。どの IPI ポートを使用するかは、rpc_init()で指定します。

IPI のコンフィギュレーション時には、rpc_init()で指定した IPI ポートを利用できるようにコンフィギュレーションしておく必要があります。

7.6.4 スピンロックライブラリ

RPC では、CPU 間の排他制御のために、RW ロックを使用します。

7.7 提供ファイル

<RTOS_INST>¥os¥include¥

rpc_pubic.h	API 定義ヘッダ

<RTOS_INST>¥os¥lib¥debug¥ *

rpc.lib *	ライブラリ(デバッグ情報付き)

<RTOS_INST>¥os¥lib¥release¥

rpc.lib	ライブラリ(デバッグ情報なし)

<SAMPLE_INST>¥R0K572650D000BR¥cpuid1¥rpc_config¥

rpc_table.c	管理テーブル(「7.9 システムのビルド」参照)

<SAMPLE_INST>¥R0K572650D000BR¥cpuid2¥rpc_config¥

rpc_table.c	管理テーブル(「7.9 システムのビルド」参照)

<RTOS_INST>¥os¥rpc¥ *	ライブラリを生成するワークスペース等
<RTOS_INST>¥os¥rpc¥rpc¥ *	ライブラリを生成するプロジェクト等
<RTOS_INST>¥os¥rpc¥rpc¥include¥ *	内部定義
<RTOS_INST>¥os¥rpc¥rpc¥ source¥ *	ソースコード

*で示したディレクトリは、ソースコード付き製品の場合のみ提供されます。
また、各ディレクトリの rpc_table.c は同じ内容です。

7.8 ライブラリのビルド(ソースコード付き製品の場合のみ)

通常はライブラリをビルドする必要はありません。デバッグなどの目的でライブラリをビルドする場合は、提供する High-performance Embedded Workshop ワークスペース(rpc.hws)を用いてビルドしてください。

7.9 システムのビルド

7.9.1 カーネルのコンフィギュレーション

RPC を使用する場合は、それに応じたカーネルコンフィギュレーションが必要です。

(1) system.system_IPL

rpc_init()で指定する rpc_config.ulIPIPortID の割込みレベルは、system.system_IPL 以下でなければなりません。

(2) maxdefine.max_task, memstk.all_memsize

rpc_start_server()および rpc_start_server_with_paramarea()では、OAL_CreateTask()(acre_tsk)によってサーバタスクを生成します。同時に生成される可能性のあるサーバタスク数は、rpc_init()で指定する rpc_config.ulTableSize です。このことを考慮して maxdefine.max_task を設定する必要があります。

また、サーバタスクはデフォルトタスクスタック用領域を使用します。このことを考慮して memstk.allmemsize を設定する必要があります。

(3) service_call

RPC(正確には OAL)では、以下のサービスコールを使用するので、これらのサービスコールを組み込むように設定しなければなりません。

- acre_tsk
- act_tsk
- exd_tsk
- slp_tsk
- wup_tsk
- acre_mpl
- del_mpl
- pget_mpl
- rel_mpl
- get_tid
- dis_dsp
- ena_dsp
- sns_ctx
- sns_dsp
- sns_dpn

7.9.2 IPI のコンフィギュレーション

rpc_init()では、指定された rpc_config.ulIPIPortID で、IPI ポートを生成(IPI_Create())します。

IPI のコンフィギュレーションでは、この IPI ポートを利用可能なようにコンフィギュレーションする必要があります。

7.9.3 システムのビルド

API 関数を使用するプログラムは、RPC ライブラリとリンクが必要です。

また、各 CPU 毎に rpc_table.c をコンパイルし、RPC ライブラリとリンクしてください。rpc_table.c を編集してはなりません。

RPC ライブラリおよび rpc_table.c のセクションは、「17.5.2 セクション一覧」を参照してください。

7.10　API 関数

表7.2　API 一覧

項番	分類	API 名称	機能
1	初期化	rpc_init	RPC ライブラリの初期化
2	終了	rpc_shutdown	RPC ライブラリの終了
3	サーバ用	rpc_start_server	ダイナミックサーバの開始
4		rpc_start_server_with_paramarea	スタティックサーバの開始
5		rpc_stop_server	サーバの停止
6	クライアント用	rpc_connect	サーバへの接続
7		rpc_disconnect	サーバとの接続解除
8		rpc_call	サーバ関数コール
9		rpc_call_copycbk	サーバ関数コール(データ転送コールバック)
10	その他	rpc_get_server_properties	サーバプロパティの取得

7.10.1　ヘッダファイル

“rpc_public.h”をインクルードしてください。

7.10.2　基本データタイプ

RPC では、types.h で定義される基本データタイプを使用します。
types.h については、「19. types.h」を参照してください。
また、各 API で使用する構造体については、以降の各 API の節で説明しています。

RENESAS

7.10.3　RPC ライブラリの初期化(rpc_init)

■C 言語 API

```
INT32 rpc_init(rpc_config *pConfig);
```

■引数

pConfig　　RPC ライブラリ初期化情報パケットへのポインタ

■パケットの構造

```
typedef struct {
        rpc_info        *pRpcTable;
        UINT32          ulTableSize;
        UINT32          ulCmdRspRangeBaseValue;
        UINT32          RedirectionTaskStackSize;
        UINT32          ServerTaskStackSize;
        UINT32          MFIFramePriority;
        UINT32          RPCTaskPriority;
        UINT32          ulIPIPortID;
} rpc_config;
```

なお、rpc_info 構造体は本 RPC の内部仕様のため、説明しません。

■リターン値

RPC_E_OK	正常終了
RPC_E_SYS	OS 状態が不正
RPC_E_NOINIT	RPC が初期化されていない（MYCPUID が 1 以外の場合のみ）
RPC_E_NORESOURCE	IPI ポートの生成に失敗

■機能

自 CPU 側の RPC ライブラリ環境を、pConfig の内容に従って初期化します。また、自 CPU が CPUID#1 の場合は、CPU 間で共有する RPC ライブラリ環境を初期化します。

本関数呼び出しの前に、IPI プリミティブの IPI_init()と OAL_Init()が完了している必要があります。
また、CPUID#2 側で本関数を呼び出すには、それ以前に CPUID#1 側で rpc_init()が完了している必要があります。
また、本関数は割込みがマスクされておらず、かつタスクレベルのコンテキスト状態から呼び出さなければなりません。この状態であれば、プリエンプト禁止状態でも呼び出し可能です。

本関数は、各 CPU で最初に一度だけ呼び出すようにしてください。なお、自 CPU 側でサーバを作らない場合も、他 CPU に対して RPC コールする場合は、本関数による初期化が必要です。

RENESAS

(1) pRpcTable, ulTableSize

RPC サーバを管理するためのテーブル領域を指定します。ulTableSize には、自 CPU 側に同時に生成可能なサーバ数を指定します。自 CPU 側にサーバを作らない場合は、ulTableSize に 0 を指定してください。

次式で算出されるサイズの非キャッシャブル領域を確保し、その先頭アドレスを pRpcTable に指定してください。pRpcTable は 4 バイト境界でなければなりません。

サイズ＝sizeof(rpc_info)×ulTableSize

ulTableSize が 0 の場合は、rpc_config_info 構造体の ulIPIPortID 以外のメンバは無視されます。

(2) ServerTaskStackSize

サーバタスクのスタックサイズを指定します。

詳細は、「18.6.4 SVC サーバタスクのスタックサイズ(remote_svc.stack_size)」を参照してください。

(3) ulIPIPortID

他 CPU へ要求した RPC からのリターン通知を受理するため、および他 CPU からの RPC 要求を受理するために使用する IPI ポート ID を指定します。

本関数では、IPI_create()を用いて、ulIPIPortID で指定された IPI ポートを生成します。

なお、ulIPIPortID の割込みレベルは、カーネル割込みマスクレベル(system.system_IPL)以下でなければなりません。

(4) ulCmdRspRangeBaseValue, RedirectionTaskStackSize, MFIFramePriority, RPCTaskPriority

これらは将来拡張用であり、単に無視されます。

7.10.4 RPC ライブラリの終了(rpc_shutdown)

■C 言語 API

```
INT32 rpc_shutdown(void);
```

■リターン値

RPC_E_OK	正常終了
RPC_E_SYS	OS 状態が不正
RPC_E_NOINIT	RPC が初期化されていない
RPC_E_STATE	起動済みのサーバが存在

■機能

自 CPU 側の RPC ライブラリ環境を終了します。rpc_init()で生成された IPI ポートは、IPI_delete()を用いて削除されます。ただし、自 CPU 側に開始済みのサーバがある場合は、エラーとなります。

また、自 CPU が CPUID#1 の場合は、CPU 間で共有する RPC ライブラリ環境も終了します。ただし、他の CPU で開始済みのサーバが存在する場合はエラーとなります。

本関数が成功した場合、以降自 CPU からの RPC の API 関数は、すべて RPC_E_NOINIT エラーになります。また、自 CPU が CPUID#1 の場合は、以降全 CPU からの RPC の API 関数は、すべて RPC_E_NOINIT エラーになります。

本関数は割込みがマスクされておらず、かつタスクレベルのコンテキスト状態から呼び出さなければなりません。この状態であれば、プリエンプト禁止状態でも呼び出し可能です。

本関数を、rpc_init()による初期化前に呼び出した場合の動作は未定義です。

7.10.5 ダイナミックサーバの開始(rpc_start_server)

■C 言語 API

```
INT32 rpc_start_server( rpc_server_info *pServerInfo );
```

■引数

pServerInfo	サーバ登録情報パケットへのポインタ

■パケットの構造

```
typedef struct {
        UINT32          ulRPCServerID;
        UINT32          ulRPCServerVersion;
        UINT32          ServerStubTaskPriority;
        UINT32          (**ServerStubList)(rpc_server_stub_info *);
        UINT32          ulNumFunctions;
        UINT32          ulStubStackSize;
        UINT32          ulMaxParamAreaSize;
        void            *pUserDefinedData;
} rpc_server_info;
```

rpc_server_stub_info は、「7.11.1 サーバスタブ」を参照してください。

■リターン値

RPC_E_OK	正常終了
RPC_E_SYS	OS 状態が不正(OAL_CanWait()に失敗)
RPC_E_NOINIT	RPC が初期化されていない
RPC_E_PAR	パラメータエラー
	(1) 0 < ulMaxParamSize < sizeof(rpc_server_stub_info)
RPC_E_NORESOURCE	既に rpc_config.ulTableSize 個のサーバが開始されている
RPC_E_STATE	ulServerID のサーバは既に開始されている
RPC_E_CREATETASK	サーバタスクの生成に失敗

■機能

自 CPU 側に、pServerInfo で指定された情報に従ってサーバを開始します。

本 API によって開始されたサーバは、クライアントとの通信に使用するパラメータ領域を、クライアントからのコール要求を受理した時点で OAL_GetMemory()を用いて動的に確保します。

本 API では、登録するサーバ用にサーバタスクを生成し、起動します。サーバタスクは、クライアントからの呼び出しがあるまで、OAL_SleepTask()によって待機(待ち状態)します。

(1) ulRPCServerID
登録するサーバIDを指定します。
すでに登録済みのサーバIDを指定するとエラーが返ります。

(2) ulRPCServerVersion
登録するサーバのバージョンを指定します。

(3) ServerStubTaskPriority
サーバタスクの優先度を指定します。サーバが動作するCPU側の他のタスクとの関係を考慮して設定してください。OSがサポートしていない優先度を指定するとエラーになります。

(4) ServerStubList, ulNumFunctions
ServerStubListは、0～(ulNumFunctions-1)のファンクションIDのサーバスタブ関数のアドレスを保持する関数テーブルのアドレスです。
この関数テーブルは、サーバが終了するまでRPCライブラリで参照するため、スタティックな領域に作成する必要があります。

(5) ulStubStackSize
サーバスタブのスタックサイズです。
スタックサイズの算出方法は、「18.11.2 RPCライブラリ」を参照してください。

(6) ulMaxParamAreaSize
サーバが動的に獲得するパラメータ領域の最大許容サイズを指定します。これを超えるパラメータ領域が必要なRPCコール要求はエラーになります。
ulMaxParamAreaSizeに0を指定すると、許容サイズの制限なしと扱います。ただし、OAL_GetMemory()で取得可能なサイズには上限があるため、これを越えるケースではRPCコール要求はエラーになります。

(7) pUserDefinedData
ここで指定したデータは、そのままサーバスタブに渡されます。RPCは、この情報を一切利用しません。ポインタでなくてもかまいません。

本関数は割込みがマスクされておらず、かつタスクレベルのコンテキスト状態で、プリエンプト許可状態で呼び出さなければなりません。

7.10.6 スタティックサーバの開始(rpc_start_server_with_paramarea)

■C 言語 API

```
INT32 rpc_start_server_with_paramarea(
          rpc_server_info *pServerInfo,
          UINT8 *pParamArea );
```

■引数

pServerInfo	サーバ登録情報パケットへのポインタ
pParamArea	パラメータ領域の先頭アドレス

■パケットの構造

```
typedef struct {
          UINT32              ulRPCServerID;
          UINT32              ulRPCServerVersion;
          UINT32              ServerStubTaskPriority;
          UINT32              (**ServerStubList)(rpc_server_stub_info *);
          UINT32              ulNumFunctions;
          UINT32              ulStubStackSize;
          UINT32              ulMaxParamAreaSize;
          void                *pUserDefinedData;
} rpc_server_info;
```

rpc_server_stub_info は、「7.11.1 サーバスタブ」を参照してください。

■リターン値

RPC_E_OK	正常終了
RPC_E_SYS	OS 状態が不正(OAL_CanWait()に失敗)
RPC_E_NOINIT	RPC が初期化されていない
RPC_E_PAR	パラメータエラー
	(1) ulMaxParamSize < sizeof(rpc_server_stub_info)
RPC_E_NORESOURCE	既に rpc_config.ulTableSize 個のサーバが開始されている
RPC_E_STATE	ulServerID のサーバは既に開始されている
RPC_E_CREATETASK	サーバタスクの生成に失敗

■機能

自 CPU 側に、pServerInfo で指定された情報に従ってサーバを開始します。

本 API によって生成されたサーバは、クライアントとの通信に使用するパラメータ領域として、pParamArea および ulMaxParamAreaSize で指定された領域を使用します。

本 API では、登録するサーバ用にサーバタスクを生成し、起動します。サーバタスクは、クライアントからの呼び出し(rpc_call)があるまで、OAL_SleepTask()によって待機(待ち状態)します。

以下、各パラメータについて rpc_start_server()との相違点のみを示します。

(1) ulMaxParamAreaSize, pParamArea

ulMaxParamAreaSizeバイトの空き領域を確保し、その先頭アドレスをpParamAreaに指定してください。

このサーバパラメータ領域は、サーバが終了するまでRPCライブラリで参照するため、スタティックな領域に確保する必要があります。また、4バイト境界にアライメントされていなければなりません。pParamAreaは4バイト境界でなければなりません。

また、SH2A-DUALではパラメータ領域は非キャッシャブル領域に生成しなければなりません。

7.10.7 サーバの停止(rpc_stop_server)

■C 言語 API

```
INT32 rpc_stop_server(
          UINT32  ulServerID;
          UINT32  ulServerVersion,
          void    (*cbk)(UINT32),
          UINT32  ulParam);
```

■引数

ulServerID	サーバ ID
ulServerVersion	サーバのバージョン
cbk	サーバ停止コールバック関数
ulParam	サーバ停止コールバック関数に渡すデータ

■リターン値

RPC_E_OK	正常終了
RPC_E_SYS	OS 状態が不正(OAL_CanWait()に失敗)
RPC_E_NOINIT	RPC が初期化されていない
RPC_E_STATE	自 CPU で ulServerID のサーバは開始されていない
RPC_E_VER	バージョンが一致しない

■機能

ulServerID で指定されたサーバ ID のサーバを停止します。ulServerVersion には、停止したいサーバのバージョンを指定します。

本 API で、他 CPU のサーバを停止することはできません。

rpc_call()を呼び出して待機していたタスクは待ち解除され、エラーが返されます。

サーバがクライアント要求を処理中の場合も、本 API は正常終了しますが、サーバの停止は処理中のサーバ関数が終了したときに行われます。

サーバが完全に停止するとき、cbk で指定されたコールバック関数が呼び出されます。ただし、cbk に NULL を指定した場合は、コールバック関数はコールされません。

コールバック関数は、何かイベントを通知することだけを行い、速やかにリターンすべきです。コールバック関数は、ブロックしたり、イベント通知以外の他の処理を行うべきではありません。

7.10.8 サーバとの接続(rpc_connect)

■C 言語 API

```
INT32 rpc_connect (
          UINT32  ulServerID;
          UINT32  ulServerVersion);
```

■引数

ulServerID	サーバ ID
ulServerVersion	サーババージョン

■リターン値

RPC_E_OK	正常終了

■機能

ulServerID で指定されたサーバと接続します。rpc_call()する前に、本 API で対象サーバに接続するようにしてください。

ただし、本 API は将来拡張用であり、現在の実装では rpc_public.h で以下のように常に正常終了するように定義されています。上記の仕様は実装されていません。

```
#define rpc_connect(ulServerID, ulServerVersion)  RPC_E_OK
```

7.10.9 サーバとの接続解除(rpc_disconnect)

■C 言語 API

```
INT32 rpc_disconnect(
          UINT32 ulServerID;
          UINT32 ulServerVersion,
          void   (*cbk)(UINT32),
          UINT32 ulParam);
```

■引数

ulServerID	サーバ ID
ulServerVersion	サーババージョン
cbk	コールバック関数
ulParam	コールバック関数に渡すパラメータ

■リターン値

RPC_E_OK	正常終了

■機能

ulServerID で指定されたサーバとの接続を解除します。

接続元のクライアントタスクと同じ CPU/OS であれば､接続元と別のタスクでも本 API で接続を解除することができます。

接続解除時に、cbk で指定されたコールバック関数が呼び出されます。cbk に NULL を指定した場合は、コールバック関数はコールされません。

コールバック関数は、何かイベントを通知することだけを行い、速やかにリターンすべきです。コールバック関数は、ブロックしたり、イベント通知以外の他の処理を行うべきではありません。

ただし、本 API は将来拡張用であり、現在の実装では rpc_public.h で以下のように常に正常終了するように定義されています。上記の仕様は実装されていません。

```
#define rpc_disconnect(ulServerID, ulServerVersion, cbk, ulParam)  RPC_E_OK
```

7.10.10 サーバ関数呼び出し (rpc_call)

■C 言語 API

```
INT32 rpc_call(rpc_call_info *pCallInfo );
```

■引数

pCallInfo　サーバ関数呼び出し情報パケットへのポインタ

■パケットの構造

```
typedef struct {
          UINT32              ulMarshallingType;
          UINT32              ulServerID;
          UINT32              ulServerVersion;
          UINT32              ulServerProcedureID;
          IOVEC               *pInputIOVectorTable;
          UINT32              ulInputIOVectorTableSize;
          IOVEC               *pOutputIOVectorTable;
          UINT32              ulOutputIOVectorTableSize;
          UINT32              *pulLastOutputIOVectorSize;
          UINT32              *pulReturnValue;
          enum rpc_ack_mode   AckMode;
} rpc_call_info;
```

■リターン値

RPC_E_OK	正常終了
RPC_E_SYS	OS 状態が不正(OAL_CanWait()に失敗)
RPC_E_NOINIT	RPC が初期化されていない
RPC_E_GETTASKID	OAL_GetTaskID()に失敗
RPC_E_STATE	ulServerID のサーバは開始されていない
RPC_E_VER	バージョンが一致しない
RPC_E_PARM	(1)AckMode が RPC_ACK，RPC_UNACK 以外
	(2)呼び出しに必要なサイズが、サーバの許容サイズを超えている
	(3)ulProcedureID≧rpc_server_info.ulNumFunctions
RPC_E_STOP	呼び出し待ちの間にサーバが停止された
RPC_E_NOMEM	メモリ不足
	(1)呼び出し待ち管理用メモリの取得に失敗
	(2)サーバがパラメータ領域の取得に失敗

■機能

ulServerID で指定されたサーバの ulServerProcedureID で指定されたファンクション ID のサーバ関数を呼び出します。サーバ関数は、サーバ側の CPU で実行されます。

pInputIOVectorTable で指定される入力ベクタの内容は、すべてサーバのパラメータ領域に転送されます。

コールモード(AckMode)として、RPC_ACK(同期モード)または RPC_UNACK(非同期モード)を指定します。

同期モードでは、本 API 内でサーバ関数の完了を待機し、その後本 API からリターンします。

非同期モードでは、本 API はサーバへ実行依頼を行った後、サーバ関数の完了を待たずにリターンします。

同期モードでは、サーバからの出力を受け取ることができますが、非同期モードではできません。

本 API は、以下のフェーズに分けて処理されます。

(1) フェーズ 1:サーバ使用権の取得 (呼び出し待ち)

サーバ使用権を取得します。

対象サーバが他のクライアント要求を処理中の場合は、サーバ使用権を取得できないため、そのクライアント要求が完了するまで OAL_SleepTask()によって待機します。これを「呼び出し待ち」と呼びます。呼び出し待ちは AckMode に関係なく行われます。複数のクライアントタスクがこの待機状態になることがあります。これらは FIFO で管理されます。

なお、この OAL_SleepTask()に先立って、待ち状態を管理するためのメモリを OAL_GetMemory()で取得します。OAL_GetMemory()に失敗した場合は、直ちに RPC_E_NOMEM エラーでリターンします。

(2) フェーズ 2:サーバのパラメータ領域の確保(ダイナミックサーバの場合のみ)

対象サーバに対してパラメータ領域の確保を依頼し、その完了を OAL_SleepTask()によって待ちます。この待機は AckMode に関係なく行われます。

サーバがパラメータ領域の取得に失敗した場合は、直ちに RPC_E_NOMEM エラーでリターンします。

(3) フェーズ 3:入力パラメータの転送

指定された入力 IOVEC 配列(pInputIOVectorTable から ulInputIOVectorTableSize 個の配列)によって定義される入力パラメータ領域の内容を、サーバのパラメータ領域にコピーします。

(4) フェーズ 4:サーバへの実行依頼

ulServerProcedureID で指定されたサーバ関数の実行を依頼します。

同期モードの場合は、サーバ関数が終了するまで、本 API を呼び出したタスクは、本 API 内で OAL_SleepTask()によって待機します。

一方、非同期モードの場合は、待機は行わずに直ちに API からリターンし、以降のフェーズは実行しません。なお。フェーズ 6,7 に相当する処理は、RPC サーバタスクがサーバスタブ関数からリターンしたときに実施されます。

(5) フェーズ 5：出力パラメータの回収

同期モードの場合は、サーバのパラメータ領域内に設定された出力パラメータを、pOutputIOVectorTable および ulOutputIOVectorTableSize で定義される領域にコピーします。

(6) フェーズ 6：サーバのパラメータ領域の解放(ダイナミックサーバの場合のみ)

サーバに対してパラメータ領域の解放を依頼し、その完了を OAL_SleepTask()によって待ちます。

(7) フェーズ 7：サーバ使用権の解放

サーバの使用権を解放します。これにより、そのサーバは別のクライアント要求を処理可能になります。呼び出し待ちタスクが存在する場合は、その待ち行列先頭のタスクが起床されます。そのタスクは、フェーズ 2 から処理を再開することになります。

以下、各パラメータについて解説します。

(1) ulMarshallingType
マーシャリングタイプを指定します。マーシャリングタイプの扱いは、クライアントスタブとサーバスタブの間で決めてください。RPCライブラリ自身が、この情報を利用することはありません。ulMarshallingTypeは、そのままサーバスタブに渡されます。
以下の条件を満たす場合は、一般にはulMarshallingTypeを利用する必要はありません。
(a)クライアントとサーバのCPU(バイトオーダなど)が同じ場合 (SH2A-DUALの場合は、これに該当します)
(b)クライアントとサーバのコンパイラ環境が同じ場合

(2) ulServerID, ulServerVersion, ulServerProcedureID
ulServerIDで指定されたサーバのulServerProcedureIDで指定されたサーバ関数を呼び出します。
指定されたサーバのバージョンがulServerVersionと一致しない場合は、エラーが返ります。

(3) pInputIOVectorTable, ulInputIOVectorTableSize
サーバ関数に渡す入力パラメータ領域を指定します。入力パラメータは、キャッシャブル領域でも構いません。ulInputIOVectorTableSizeには、pInputIOVectorTableが指すIOVEC配列の要素数を指定します。
入力パラメータの数(ulInputIOVectorTableSize)は、常に固定でなければなりません。また、最後以外のIOVEC.ulSizeは、常に固定サイズとしなければなりません。これらは、サーバスタブで各入力パラメータの格納アドレスを求めるためです。最後のIOVEC.ulSizeは可変サイズを指定できます。
何も情報を渡さない場合は、ulInputIOVectorTableSizeに0を指定してください。
なお、本APIでは、このIOVEC配列の内容は更新しません。

(4) pOutputIOVectorTable, ulOutputIOVectorTableSize, pulLastOutputIOVectorSize
サーバ関数からの出力を受け取る領域を指定します。ulOutputIOVectorTableSizeには、pOutputIOVectorTableが指すIOVEC配列の要素数を指定します。
何も情報を受け取らない場合は、ulOutputIOVectorTableSizeに0を指定してください。
各IOVECが指す領域に、出力が格納されます。
最後以外のIOVECで示される領域には、そのIOVECのulSizeバイトが出力されます。一方、最後のIOVECで示される領域には、そのIOVECのulSizeバイトが出力されるわけではありません。実際に出力されたサイズは、pulLastOutputIOVectorSizeが指す領域に返されます。

なお、本APIでは、このIOVEC配列の内容は更新しません。

(5) pulReturnValue
*pulReturnValueには、サーバ関数のリターン値が返ります。

(6) AckMode
以下のいずれかを指定できます。

- RPC_ACK(同期モード)
  本APIを呼び出したタスクは、サーバ関数からリターンするまで、本API内で待機します。
- RPC_UNACK(非同期モード)
  サーバ関数の実行を依頼した後、サーバ関数の終了を待たずに、直ちにリターンします。非同期モード指定時は、pOutputIOVectorTable, ulOutputIOVectorTableSize, pulReturnValue は無視されます。

なお、本 API で自 CPU 内のサーバ関数を呼び出すこともできます。ただし、自 CPU であっても他 CPU へのコールと同様の手順となるため、普通に関数コールするよりも遅くなる点に注意してください。

7.10.11　サーバ関数呼び出し(データ転送コールバック) (rpc_call_copycbk)

■C 言語 API

```
INT32 rpc_call_copycbk(
          rpc_call_info *pCallInfo,
          void (*CopyCbk1)(void *, const void *, UINT32),
          void (*CopyCbk2)(void *, const void *, UINT32) );
```

■引数

pCallInfo	サーバ関数呼び出し情報パケットへのポインタ
CopyCbk1	クライアントの入力パラメータをサーバパラメータ領域にコピーする関数の開始アドレス
CopyCbk2	サーバが出力したパラメータをクライアントの出力パラメータ領域にコピーする関数の開始アドレス

■パケットの構造

```
typedef struct {
          UINT32                ulMarshallingType;
          UINT32                ulServerID;
          UINT32                ulServerVersion;
          UINT32                ulServerProcedureID;
          IOVEC                 *pInputIOVectorTable;
          UINT32                ulInputIOVectorTableSize;
          IOVEC                 *pOutputIOVectorTable;
          UINT32                ulOutputIOVectorTableSize;
          UINT32                *pulLastOutputIOVectorSize;
          UINT32                *pulReturnValue;
          enum rpc_ack_mode     AckMode;
} rpc_call_info;
```

■リターン値

RPC_E_OK	正常終了
RPC_E_SYS	OS 状態が不正(OAL_CanWait()に失敗)
RPC_E_NOINIT	RPC が初期化されていない
RPC_E_GETTASKID	OAL_GetTaskID()に失敗
RPC_E_STATE	ulServerID のサーバは開始されていない
RPC_E_VER	バージョンが一致しない
RPC_E_PARM	(1)AckMode が RPC_ACK, RPC_UNACK 以外 (2)呼び出しに必要なサイズが、サーバの許容サイズを超えている (3)ulProcedureID≧rpc_server_info.ulNumFunctions
RPC_E_STOP	呼び出し待ちの間にサーバが停止された
RPC_E_NOMEM	メモリ不足 (1)呼び出し待ち管理用メモリの取得に失敗 (2)サーバがパラメータ領域の取得に失敗

■機能

ulServerID で指定されたサーバの ulServerProcedureID で指定されたサーバ関数を呼び出します。サーバ関数は、サーバ側の CPU で実行します。

本 API は、rpc_call()と以下の点だけが異なります。

rpc_call()のフェーズ 3 で実施されるサーバパラメータ領域への入力パラメータの転送、およびフェーズ 5 で実施されるサーバパラメータ領域からの出力パラメータの回収を、rpc_call()ではなく、それぞれ CopyCbk1 および CopyCbk2 で指定したコールバック関数で実施します。ただし、CopyCbk1, CopyCbk2 に NULL を指定した場合は、該当するコールバック関数はコールされません。

7.10.12 サーバプロパティの取得 (rpc_get_server_properties)

■C 言語 API

```
INT32 rpc_get_server_properties(
          UINT32 ulServerID,
          rpc_server_properties *pProp );
```

■引数

ulServerID	サーバ ID
pProp	サーバプロパティ情報パケットへのポインタ

■パケットの構造

```
typedef struct {
          UINT32            ulServerVersion;
          UINT32            ulMaxParamArea;
} rpc_server_properties;
```

■リターン値

RPC_E_SYS	OS 状態が不正(OAL_CanWait()に失敗)
RPC_E_NOINIT	RPC が初期化されていない
RPC_E_STATE	ulServerID のサーバは開始されていない

■機能

ulServerID で指定されたサーバの情報を取得し、pProp の指す領域に返します。

pProp->ulServerVersion にはサーバのバージョンを、pProp->ulMaxParamArea にはサーバ開始時に指定した rpc_server_info.ulMaxParamAreaSize を返します。

7.11　スタブ

サーバスタブおよびクライアントスタブは、サーバ作成者が作成する必要があります。ともに、サーバ関数ごとに用意してください。

7.11.1　サーバスタブ

サーバスタブは、RPC サーバタスクからコールされます。以下の仕様にしたがって作成してください。関数名は任意です。

```
UINT32 stub_function(rpc_server_stub_info *pInfo );


typedef struct {
            UINT32              ulProcedureID;
            enum rpc_ack_mode AckMode;
            UINT32              ulMarshallingType;
            UINT8               *pucParamArea;
            UINT32              ulMaxParamArea;
            UINT32              ulInParamSize;
            IOVEC               *pOutputIOVectorTable;
            UINT32              ulOutputIOVectorTableSize;
            void                *pUserDefinedData;
} rpc_server_stub_info;
```

(1)　ulProcedureID
rpc_call(), rpc_call_copycbk()で指定されたrpc_call_info. ulServerProcedureIDが渡されます。

(2)　AckMode
rpc_call(), rpc_call_copycbk()で指定されたrpc_call_info.AckMode(RPC_ACKまたはRPC_UNACK)が渡されます。

(3)　ulMarshallingType
rpc_call(), rpc_call_copycbk()で指定されたrpc_call_info. ulMarshallingTypeが渡されます。

(4)　pucParamArea, ulInParamSize
クライアントからの入力パラメータを示す情報が渡されます。
pucParamAreaからulInParamSizeバイトの領域に、rpc_call(), rpc_call_copycbk()で指定した入力パラメータが格納されます。なお、pucParamAreaはサーバパラメータ領域内のアドレスであり、必ず4バイト境界にアライメントされます。
ulInParamSizeには、以下のサイズが格納されます。
Σ(ALIGNUP4(rpc_call_info.pInputIOVectorTable[i]->ulSize)) …(a)

0番目のパラメータの格納アドレスは、pucParamAreaとなります。
k番目(k = 1 … rpc_call_info.ulInputIOVectorTableSize-1)のパラメータ格納アドレスは、以下によって計算されます。

$$pucParamArea + \sum_{i=0}^{k-1}(ALIGNUP4(rpc_call_info.pInputIOVectorTable[i]\text{->}ulSize)) \quad \ldots(b)$$

サーバスタブが式(b)に従って各パラメータ格納アドレスを正しく求められるようにするために、「常にパラメータの数は固定、最後以外のパラメータのサイズも固定」という仕様にしたがってサーバスタブ・クライアントスタブを実装する必要があります。

(5) pOutputIOVectorTable, ulOutputIOVectorTableSize
クライアントに出力するパラメータを格納する領域を示す情報が渡されます。
pOutputIOVectorTableは出力IOVEC配列の先頭アドレスです。ulOutputIOVectorTableSizeはその要素数で、rpc_call(), rpc_call_copycbk()でクライアントが指定した rpc_call_info.ulInputIOVectorTableSizeが設定されます。
各出力IOVECは以下の設定になっています。つまり、デフォルトの出力領域があらかじめ設定されています。

- pBaseAddress：サーバパラメータ領域内のアドレス(入力パラメータ領域と重複しない)
- ulSize : rpc_call(), rpc_call_copycbk()でクライアントが指定したサイズ、すなわちクライアントが受け取ることのできるサイズ

なお、クライアントが出力を要求しない場合、および非同期モードでの呼び出しの場合は、ulOutputIOVectorTableSize とpOutputIOVectorTableに0が設定されます。

サーバスタブでは、各出力IOVECで指定された領域に出力データを設定し、リターンしてください。その際、出力IOVECは以下のようにしてください。

- pBaseAddress : 通常は変更しないでください。入力パラメータと出力パラメータの領域をオーバーラップさせたい場合などは、「IOVEC が指す領域がサーバパラメータ領域内」という条件を満たすように設定してください。
- ulSize:最後以外の IOVEC の ulSize は変更しないでください。最後の IOVEC の ulSize は、実際のサイズに書き換えてください。

RPCは、サーバスタブからリターンした時点のこのIOVEC配列が指すデータを、クライアントが指定した出力パラメータ領域(rpc_call_info.pOutputIOVectorTableのIOVEC配列が指す領域)にコピーします。クライアントには、rpc_call_info.pulLastOutputIOVectorSizeの指す領域に、最後のIOVECのulSizeが返されます。

(6) ulMaxParamArea
ulMaxParamAreaには、pucParamAreaからサーバスタブが利用可能なサイズが渡されます。この中には、pOutputIOVectorTableの各IOVECが指す出力パラメータ格納領域も含まれています。
なお、サーバスタブに渡されるulMaxParamAreaの値と、サーバ開始時に指定する rpc_server_info. ulMaxParamAreaSizeは異なります。

(7) pUserDefinedData
rpc_start_server(), rpc_start_server_with_paramarea()で指定したユーザ定義情報が渡されます。

図 7.3に、入力パラメータ数が 4、出力パラメータ数が 2 の場合のサーバパラメータ領域の例を示します。

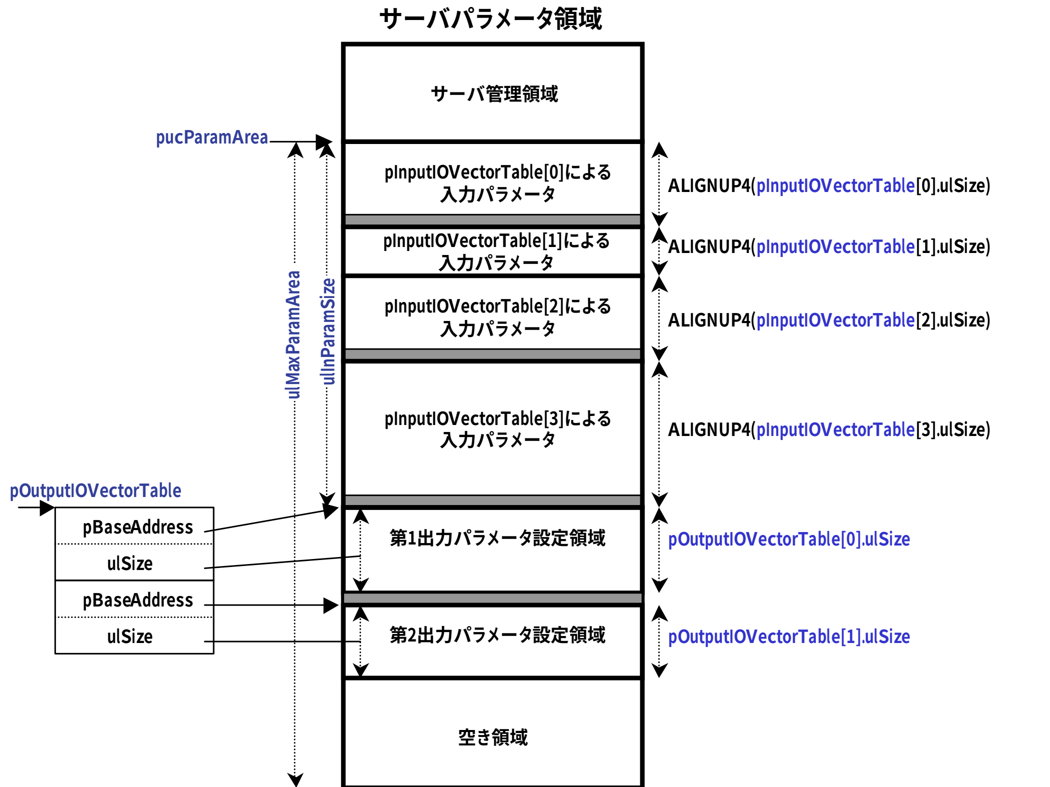


図7.3　サーバパラメータ領域の例

7.11.2　クライアントスタブ

クライアントスタブは、アプリケーションからオリジナルのサーバ関数と同じ関数名を持ち、rpc_call()または rpc_call_copycbk()を用いて RPC コールを行うように実装してください。

7.12 サーバ停止コールバック関数

本コールバック関数は、サーバタスクからコールされます。以下の仕様にしたがって作成してください。関数名は任意です。

```
void StopServer(UINT32 ulParam);
```

ulParam には、rpc_stop_server()で指定した ulParam が渡されます。

7.13 CopyCbk1, CopyCbk2 コールバック関数

これらのコールバック関数は、rpc_call_copycbk()からコールされます。いずれも以下の仕様にしたがって作成してください。関数名は任意です。

```
void CopyFunc(void *pDest, const void *pSource, UINT32 ulSize);
```

pSource で指定されるアドレスから ulSize バイトを pDest の指すアドレスにコピーします。

DMA など CPU 以外の機能を用いてコピーを行う場合で、オペランドキャッシュを有効な状態で使用する場合は、以下に留意してください。

(1) CopyCbk1

pDest はサーバパラメータ領域(非キャッシャブル領域)を指しています。
一方、pSource はクライアントが指定した入力パラメータ領域を指しています。オペランドキャッシュがライトバックモードの場合で、pSource の指す内容がオペランドキャッシュに登録されており、かつ実メモリに書き戻されていない場合、DMA 転送前に pSource が指す領域の内容をオペランドキャッシュから実メモリに書き戻す(ライトバック)必要があります。これを行わないと、DMAC はライトバック前のデータを pSource から読み出してしまうことになります。

(2) CopyCbk2

pSource はサーバが出力したパラメータ領域(非キャッシャブル領域)を指しています。
一方、pDest はクライアントが指定した出力パラメータ領域を指しています。pDest の指す領域がオペランドキャッシュに登録されている場合、DMA 転送前に pDest が指す領域のオペランドキャッシュ内容を無効化する必要があります。これを行わないと、DMA 転送後にクライアントが出力パラメータ領域を読み出したときにオペランドキャッシュにヒットしてしまい、クライアント側は DMA によって実メモリに転送されたデータではなく、オペランドキャッシュの内容を読み出してしまうことになります。また、オペランドキャッシュがライトバックモードの場合で、pDest の指す内容がオペランドキャッシュに登録されており、かつ実メモリに書き戻されていない場合、DMA 転送後にオペランドキャッシュのリプレースによるライトバックが発生すると、DMA によって実メモリに転送されたデータが破壊されてしまうことになります。
SH2A-DUAL では特定のアドレス範囲のオペランドキャッシュ内容を無効化することができないため、オペランドキャッシュ全体を無効化するか、もしくはクライアントが指定する出力パラメータ領域を非キャッシャブル領域に制限する必要があります。

8. OAL

8.1 概要

OAL は、RPC を容易に他の OS へ移植できるように、RPC の OS 依存部を局所化したものです。これにより、RPC の他の OS への移植性を高めています。

本 OAL では、RPC が必要とする最小限の OS 機能のみを実装しています。

表8.1 OAL 概要

No.	項目	内容
1	使用するハードウェアリソース	なし
2	本ソフトウェアが使用するソフトウェア部品	なし
3	本ソフトウェアを使用している他のソフトウェア部品	RPC

8.2 提供ファイル

<RTOS_INST>¥os¥include¥

oal.h API 定義ヘッダ

<SAMPLE_INST>¥R0K572650D000BR¥cpuid1¥ipi¥

oal_config.h コンフィギュレーションファイル(「8.3.1 コンフィギュレーション」参照)

oal.c ソースコード

<SAMPLE_INST>¥R0K572650D000BR¥cpuid2¥ipi¥

oal_config.h コンフィギュレーションファイル(「8.3.1 コンフィギュレーション」参照)

oal.c ソースコード

なお、各ディレクトリの oal.c は同じ内容です。

8.3 コンフィギュレーションとビルド

OAL は、CPU 毎にコンフィギュレーションしてください。

8.3.1 コンフィギュレーション

(1) OAL のコンフィギュレーション

oal_config.h で、以下の定義を記述してください。

```
#define OAL_MEMSIZE 0x1000
```

OAL_MEMSIZE には、OAL_GetMemory()で扱うメモリ領域のサイズを定義します。

(2) カーネルのコンフィギュレーション

「7.9.1(3) service_call」を参照してください。

8.3.2 ビルド

oal.c をコンパイルし、API 関数を使用するプログラムとリンクしてください。
OAL のセクションは、「17.5.2 セクション一覧」を参照してください。

8.4 API 関数

表 8.2に API 関数一覧を示します。

表8.2　API 関数一覧

項番	種別	API 名称	機能
1	関数	OAL_Init	OAL の初期化
2	関数	OAL_Shutdown	OAL の終了
3	関数	OAL_DisablePreempt	タスクプリエンプションの禁止
4	関数	OAL_EnablePreempt	タスクプリエンプションの許可
5	関数	OAL_IsDisablePreempt	タスクプリエンプション状態の確認
6	関数	OAL_CanWait	自タスク待機可否の確認
7	関数	OAL_IsNotTaskLevel	コンテキスト種別の確認
8	関数	OAL_IsMaskInterrupt	プロセッサの割込みマスクの確認
9	関数	OAL_CreateTask	タスクの生成
10	関数	OAL_ActivateTask	タスクの起動
11	関数	OAL_DestroyTask	自タスクの終了・削除
12	関数	OAL_GetTaskID	自タスクの識別情報の取得
13	関数	OAL_SleepTask	タスクの待機
14	関数	OAL_WakeupTask	タスク待機の解除
15	関数	OAL_GetMemory	メモリの取得
16	関数	OAL_ReleaseMemory	メモリの解放

8.4.1 ヘッダファイル

oal.h をインクルードしてください。

8.4.2 基本データタイプ

types.h で定義される基本データタイプを使用します。
types.h については、「19. types.h」を参照してください。

8.4.3 リターン値

基本的には、リターン値を返す API では、0 または正が正常終了、負がエラーという仕様になっています。

エラーリターン値としては、以下を規定しています。ただし、どういう条件でどういうエラーを返すかは、OAL の実装および OS に依存するため、OAL を利用するアプリケーションではデバッグ以外の目的でエラーリターン値を判定することは推奨されません。例えば、ある OS ではエラー OAL_E_PAR を返す条件で、別の OS では別のエラーが返ることがあります。

このような理由から、以降の各 API の節ではエラーリターン値に関する解説は行いません。

```
#define OAL_E_OK              0L       正常終了
#define OAL_E_PAR             (-1L)    パラメータエラー
#define OAL_E_SYS             (-4L)    システム状態に関するエラー
#define OAL_E_STATE           (-5L)    OS オブジェクト状態に関するエラー
#define OAL_E_NOMEM           (-7L)    メモリ不足
#define OAL_E_NORESOURCE      (-8L)    リソース不足
#define OAL_E_TIMEOUT         (-16L)   タイムアウト
#define OAL_E_RELEASED        (-17L)   強制待ち解除
```

8.4.4 OAL の初期化(OAL_Init)

■C 言語 API

```
INT32 OAL_Init( void );
```

■機能説明

OAL を初期化し、開始します。
タスク以外のコンテキスト状態で呼び出した場合の動作は未定義です。

8.4.5 OAL の終了(OAL_Shutdown)

■C 言語 API

```
void OAL_Shutdown( void );
```

■機能説明

OAL を終了します。
OAL_DisablePreempt()した状態で呼び出した場合の動作は未定義です。
タスクレベル以外のコンテキスト状態で呼び出した場合の動作は未定義です。

8.4.6 タスクプリエンプションの禁止(OAL_DisablePreempt)

■C 言語 API

```
void OAL_DisablePreempt( void );
```

■機能説明

タスクプリエンプションを禁止します。
OAL_DisablePreempt()した状態で呼び出してはなりません。
タスクレベル以外のコンテキスト状態で呼び出した場合の動作は未定義です。

8.4.7 タスクプリエンプションの許可(OAL_EnablePreempt)

■C 言語 API

```
void OAL_EnablePreempt( void );
```

■機能説明

タスクプリエンプションを許可します。
OAL_DisablePreempt()した状態でも呼び出せます。
タスクレベル以外のコンテキスト状態で呼び出した場合の動作は未定義です。

8.4.8 タスクプリエンプション状態の確認(OAL_IsDisablePreempt)

■C 言語 API

```
INT32 OAL_IsDisablePreempt( void );
```

■機能説明

タスクプリエンプション禁止状態なら 1、許可状態なら 0 を返します。
エラーリターンすることはありません。
タスクレベル以外のコンテキスト状態で呼び出した場合の動作は未定義です。

8.4.9 自タスク待機可否の確認(OAL_CanWait)

■C 言語 API

```
INT32 OAL_CanWait( void );
```

■機能説明

呼び出し元コンテキストが OS の待ち状態へ遷移可能なら 1、遷移不可なら 0 を返します。
エラーリターンすることはありません。

8.4.10 コンテキスト種別の確認(OAL_IsNotTaskLevel)

■C 言語 API

```
INT32 OAL_IsNotTaskLevel( void );
```

■機能説明

呼び出し元コンテキストがタスクレベルの場合は 0、そうでない場合は 1 を返します。
エラーリターンすることはありません。

8.4.11 プロセッサの割込みマスクの確認(OAL_IsMaskInterrupt)

■C 言語 API

```
INT32 OAL_IsMaskInterrupt ( void );
```

■機能説明

プロセッサの割込みマスクによってマスクされている割込みがある場合は 1、全ての割込みがマスクされていない場合は 0 を返します。
エラーリターンすることはありません。

8.4.12 タスクの生成(OAL_CreateTask)

■C 言語 API

```
INT32 OAL_CreateTask(
          void    **pTaskID
          void    *pTaskStartAddress,
          void    *pArg,
          UINT32  ulTaskPriority,
          UINT32  ulStackSize,
          enum OAL_TASK_START AutoStart);
```

■引数

pTaskID	生成したタスクの識別情報を返す記憶域へのポインタ
pTaskStartAddress	タスクの開始アドレス
pArg	タスクに渡すパラメータ
ulTaskPriority	タスクの優先度
ulStackSize	スタックサイズ
AutoStart	タスク起動指定

■機能説明

タスクを生成し、*pTaskID に生成したタスクの識別情報を返します。タスク識別情報は、他の OAL の API でタスクを指定するために利用します。

AutoStart には、OAL_AUTO_START または OAL_NO_START のいずれかを指定できます。

OAL_AUTO_START 指定時は、指定されたタスクがすぐに OS 上の実行可能状態になります。OAL_NO_START 指定時は、指定されたタスクは生成されるだけで、実行されることはありません。実行させるには、別途 OAL_ActivateTask()でタスクを起動する必要があります。

pArg には、タスクに渡すパラメータを指定します。

OAL_DisablePreempt()した状態でも呼び出せます。

タスクレベル以外のコンテキスト状態で呼び出した場合の動作は未定義です。

8.4.13 タスクの起動(OAL_ActivateTask)

■C 言語 API

```
INT32 OAL_ActivareTask(void *TaskID);
```

■引数

TaskID	タスク識別情報

■機能説明

タスクを起動します。

OAL_DisablePreempt()した状態でも呼び出せます。

タスクレベル以外のコンテキスト状態で呼び出した場合の動作は未定義です。

8.4.14 自タスクの終了･削除(OAL_DestroyTask)

■C 言語 API

```
INT32 OAL_DestroyTask(void);
```

■機能説明

自タスクを終了し、削除します。
OAL_DisablePreempt()した状態で呼び出した場合の動作は未定義です。
タスクレベル以外のコンテキスト状態で呼び出した場合の動作は未定義です。

8.4.15 自タスク識別情報の取得(OAL_GetTaskID)

■C 言語 API

```
INT32 OAL_GetTaskID(void **pTaskID);
```

■引数

pTaskID　　タスクの識別情報を返す記憶域へのポインタ

■機能説明

自タスクの識別情報を、*pTaskID に返します。
OAL_DisablePreempt()した状態で呼び出した場合の動作は未定義です。
タスクレベル以外のコンテキスト状態で呼び出した場合の動作は未定義です。

8.4.16 自タスクの待機(OAL_SleepTask)

■C 言語 API

```
INT32 OAL_SleepTask(void);
```

■機能説明

自タスクを待機します。OAL_WakeupTask()によって、待機が解除されます。
OS の待ち状態へ遷移できない状態からの呼び出しは、動作保証しません。
OAL_DisablePreempt()した状態で呼び出した場合の動作は未定義です。
タスクレベル以外のコンテキスト状態で呼び出した場合の動作は未定義です。

8.4.17 タスク待機の解除(OAL_WakeupTask)

■C 言語 API

```
INT32 OAL_WakeupTask(void *TaskID);
```

■引数

TaskID　　タスク識別情報

■機能説明

タスクの待機を解除します。
OAL_DisablePreempt()した状態でも呼び出せます。
タスクレベル以外のコンテキスト状態でも呼び出せます。

8.4.18 メモリの取得(OAL_GetMemory)

■C 言語 API

```
INT32 OAL_GetMemory(
          UINT32  ulSize,
          void    **ppAddress);
```

■引数

ulSize　　取得サイズ
ppAddress　取得メモリのアドレスを返す記憶域へのポインタ

■機能説明

ulSize で指定したサイズのメモリを取得し、その先頭アドレスを ppAddress の指す領域に返します。
メモリを取得できない場合は、待機せずに直ちにエラーリターンします。
返されるメモリの先頭アドレスは、4 バイト境界にアライメントされます。

OAL_DisablePreempt()した状態でも呼び出せます。
タスクレベル以外のコンテキスト状態で呼び出した場合の動作は未定義です。

8.4.19 メモリの解放(OAL_ReleaseMemory)

■C 言語 API

```
INT32 OAL_ReleaseMemory(void *pAddress);
```

■引数

pAddress　取得メモリのアドレス

■機能説明

先頭アドレスが pAddress のメモリを解放します。pAddress は OAL_GetMemory()で取得したメモリの先頭アドレスでなければなりません。
OAL_DisablePreempt()した状態でも呼び出せます。
タスクレベル以外のコンテキスト状態で呼び出した場合の動作は未定義です。

9. スピンロックライブラリ

9.1 概要

スピンロックを使うことで、同時にひとつの CPU しか CPU 間共有リソースにアクセスできないようにすることができます。

同じことは、カーネルのセマフォを使っても実現できますが、カーネルのセマフォを使う場合に対して以下の相違があります。

(1) カーネルのセマフォよりも低オーバヘッドです。

(2) 割込みハンドラなどのディスパッチ保留状態でも利用可能（カーネルのセマフォは、ディスパッチ保留状態では利用できません）

一方、「9.3 スピンロックの振る舞いと注意事項」に記載の注意点もありますので、必ず一読してください。

表9.1 スピンロックライブラリ概要

項番	項目	内容
1	使用するハードウェアリソース	なし(ただし、セマフォロックではパラメータで指定されたセマフォレジスタにアクセスします)
2	使用するソフトウェア部品	なし
3	本ソフトウェアを使用している他のソフトウェア部品	(1)カーネル(リモートサービスコールなど) (2)RPC ライブラリ (3)IPI

アプリケーションではスピンロックの使用を極力避けるべきです。スピンロックを多用することはアプリケーションの CPU 間結合度が高いことを意味し、各 CPU への機能分散に難があるとも言えます。また、「9.3 スピンロックの振る舞いと注意事項」に記載したような不具合を作り込む可能性も高くなります。

9.2 基本的な使用方法

スピンロックでは、メモリ上の「ロック変数」、または SH2A-DUAL に搭載されている「セマフォレジスタ」を使って、CPU 間共有リソースの排他制御を行います。ロック変数またはセマフォレジスタと実際の CPU 間共有リソースの関連付けは、アプリケーションで決めます。

CPU 間共有リソースへアクセスしたいアプリケーションでは、ロックを取得する API を呼び出してから共有リソースにアクセスするようにします。そして、共有リソースへのアクセスが終わったらロックを解除する API を呼び出します。

9.3 スピンロックの振る舞いと注意事項

スピンロック取得 API では、既にそのロックは取得されているかどうかをチェックします。ロック済みでない場合は、直ちにロック済みに状態を更新してリターンしますが、既にロック済みの場合はそのロックが外れるまでビジーウェイトします。

すなわち、ロック済みのロック変数を取得しようとした場合、そのロックを取得していたプログラムがロック解除 API を呼び出すまで、CPU 時間を無駄に消費し続けることになります。

このビジーウェイトは、以下のような問題を引き起こす可能性があります。本節を理解の上、このようなことが発生しないしないように注意してください。

9.3.1 同一 CPU 内での排他制御とデッドロック

スピンロック機能は、CPU を跨るプログラム間での排他制御をサポートする機能です。

CPU 間共有リソースに対して、同じ CPU 上の複数のプログラムからも同時にアクセスする可能性がある場合は、スピンロックを使う以前に、同一 CPU 内での排他制御が必要です。これを行わずにスピンロック機能だけを使用すると、デッドロックが発生する可能性があります。

■例1

ある CPU 間共有リソースに対して、ある同じ CPU 上のタスク A とタスク B が同時にアクセスする可能性がある場合は、カーネルのセマフォ機能やディスパッチ禁止などの方法で、タスク A とタスク B の間の排他制御を行う必要があります。

これを行わない場合、以下のようなケースでデッドロックが発生します。

タスク A がロックを取得中に、より優先度の高いタスク B にプリエンプトされたとします。

タスク B がロックを取得しようとすると、取得できずにビジーウェイトとなります。しかし、ロックを取得しているタスク A はタスク B よりも優先度が低いので、決して実行されません。その結果、タスク B は解放されないロックの取得待ちに陥り、デッドロックとなります。

■例2

ある CPU 間共有リソースに対して､ある同じ CPU 上のタスク A と割込みハンドラが同時にアクセスする可能性がある場合は、割込み禁止や CPU ロックなどの方法で、タスク A と割込みハンドラの間の排他制御を行う必要があります。

これを行わない場合、以下のようなケースでデッドロックが発生します。

タスク A がロックを取得中に、割込みハンドラが起動されたとします。

ハンドラがロックを取得しようとすると、取得できずビジーウェイトとなります。しかし、ロックを取得しているタスク A は、割込みハンドラが終了しないと決して実行されません。その結果、割込みハンドラは解放されないロックの取得待ちに陥り、デッドロックとなります。

9.3.2 ロック取得期間の問題

スピンロックによるビジーウェイトは、純粋に CPU 時間の浪費となります。

この無駄を短くするには、ロックを取得している期間をできるだけ短くしてください。これにより、他のプログラムがロックをしようとしたときに発生する可能性のあるビジーウェイトを短くすることができます。

しかし、プログラマが意図せずにロック期間が長引いてしまうことがあるので注意が必要です。以下にその例を示します。

■例1

あるタスクを、ロックを取得してから CPU 間共有リソースにアクセスし、その後直ちにロックを解放するようにコーディングした。しかし、実行時には、ロックを解放する前に他のタスクにプリエンプトされてしまった。この間に、別の CPU からロックを取得しようとし、長時間ビジーウェイトしてしまった。

このようなケースは、ロック取得前にディスパッチを禁止しておくことで改善することができます。

■例2

あるタスクを、ディスパッチ禁止状態にしてから、ロックを取得してから CPU 間共有リソースにアクセスし、その後直ちにロックを解放するようにコーディングした。しかし、実行時には、ロックを解放する前に割込みが発生し、割込みハンドラが終了するまでロックの解放が遅れた。この間に、別の CPU からロックを取得しようとし、長時間ビジーウェイトしてしまった。

このようなケースは、ロック取得前に割込みを禁止しておくことで改善することができます。

9.4 3 種類のスピンロック

本スピンロックライブラリでは、以下の 3 種類のスピンロック機能を提供しています。

(1) ノーマルロック

基本となるロック機能です。CPU 間共有メモリに配置されたロック変数への TST 命令により、CPU 間の排他制御を行います。

(2) RW ロック

ノーマルロックと同様に、CPU 間共有メモリに配置されたロック変数への TST 命令により、CPU 間の排他制御を行いますが、参照(リード)同士の排他制御は不要という観点でノーマルロックよりも効率化したものです。RW ロックは、ノーマルロックに比べて、参照(リード)のみを行うことの多いリソースの排他制御に向いています。

(3) セマフォロック

セマフォロックでは、SH2A-DUAL に搭載されたセマフォレジスタを使って、CPU 間の排他制御を行います。なお、このセマフォレジスタはカーネルのセマフォとは何ら関係ありません。

ノーマルロックおよび RW ロックの場合、ロック待ちのビジーウェイトではロック変数へのアクセスのためにロック変数が配置されたメモリへのバスを消費するため、他 CPU からの同じバスへのアクセス性能が悪化することがあります。

一方、セマフォレジスタへのアクセスはメモリと異なるバスを使うため、このような弊害が小さいことがメリットです。

なお、IPI ではセマフォロックを使用しています。

9.5 ノーマルロックと RW ロックのロック変数

9.5.1 ロック変数の実体

ロック変数の実体は、CPUID#1 側で定義してください。その際、他とは異なる専用のセクションとしてください。CPUID#1 側のリンク時に、そのセクションのシンボルアドレスファイル(拡張子 fsy)を出力し、CPUID#2 側でそのシンボルアドレスファイルをアセンブル・リンクすることで、CPUID#2 側のプログラムから CPUID#1 側のロック変数へのシンボル解決が可能になります。

9.5.2 ロック変数を置く RAM

ロック変数は、そのロック変数をアクセスする各 CPU から同じバスを介して接続されたメモリに置かなければなりません。また、非キャッシャブルアクセスでなけれなりません。

SH7205, SH7265 の場合の例を、以下に示します。

(1) ロック変数を内蔵 RAM に配置したい場合

内蔵 RAM へのアクセスはキャッシュされません。

ロック変数は、リンク時に内蔵 RAM のシャドー空間(0xFFD80000～0xFFDA7FFF)のアドレスに配置してください。このアドレスへのアクセスは、両 CPU から同じ「高速内蔵 RAM アクセスバス」が使用されます。

(2) ロック変数を外部 RAM に配置する場合

SDRAM0 空間(0x18000000～0x1BFFFFFF)に接続された SDRAM にロック変数を配置する場合は、リンク時にロック変数を SDRAM0 空間の非キャッシャブルシャドー空間(0x38000000～0x3BFFFFFF)のアドレスに配置してください。

9.6 提供ファイル

<RTOS_INST>¥os¥include¥	
spinlock.h	API 定義ヘッダ
<RTOS_INST>¥os¥lib¥debug¥	
spinlock.lib	ライブラリ(デバッグ情報付き)
<RTOS_INST>¥os¥lib¥release¥	
spinlock.lib	ライブラリ(デバッグ情報なし)
<RTOS_INST>¥os¥spinlock¥	ライブラリを生成するワークスペース等
<RTOS_INST>¥os¥spinlock¥spinlock¥	ライブラリを生成するプロジェクト等
<RTOS_INST>¥os¥spinlock¥spinlock¥include¥	内部定義(アセンブリ言語)
<RTOS_INST>¥os¥spinlock¥spinlock¥source¥	ソースコード
<RTOS_INST>¥os¥spinlock¥spinlock¥debug¥	コンフィギュレーションディレクトリ(デバッグ情報付き)
<RTOS_INST>¥os¥spinlock¥spinlock¥release¥	コンフィギュレーションディレクトリ(デバッグ情報なし)

9.7 ライブラリのビルド

通常はライブラリをビルドする必要はありません。デバッグなどの目的でライブラリをビルドする場合は、提供する High-performance Embedded Workshop ワークスペース(spinlock.hws)を用いてビルドしてください。

9.8 システムのビルド

スピンロックライブラリのセクションは、「17.5.2 セクション一覧」を参照してください。

RENESAS

9.9 API 関数

表 9.2に API 関数一覧を示します。各 API 関数は、C 言語関数マクロまたは C 言語関数として実装されています。

表9.2　API 関数一覧

項番	分類	関数名	機能
1	ノーマルロック	SPIN_InitLock	ノーマルロック変数の初期化
2		SPIN_Lock	ノーマルロック取得
3		SPIN_TryLock	ノーマルロック取得試行
4		SPIN_Unlock	ノーマルロック解除
5		SPIN_IsLocked	ノーマルロック状態のチェック
6	RW ロック	SPIN_InitRWLock	RW ロック変数の初期化
7		SPIN_ReadLock	リードロック取得
8		SPIN_ReadTryLock	リードロック取得試行
9		SPIN_ReadUnlock	リードロック解除
10		SPIN_IsReadLocked	リードロック状態のチェック
11		SPIN_WriteLock	ライトロック取得
12		SPIN_WriteTryLock	ライトロック取得試行
13		SPIN_WriteUnlock	ライトロック解除
14		SPIN_IsWriteLocked	ライトロック状態のチェック
15	セマフォロック	SPIN_InitSemLock	セマフォレジスタの初期化
16		SPIN_SemLock	セマフォロック取得
17		SPIN_SemTryLock	セマフォロック取得試行
18		SPIN_SemUnlock	セマフォロック解除

9.9.1 ヘッダファイル

spinlock.h をインクルードしてください。

9.9.2 基本データタイプ

types.h で定義される基本データタイプを使用します。
types.h については、「19. types.h」を参照してください。

9.9.3 注意

API 関数では、エラー検出を行いません。

9.10 ノーマルロック

9.10.1 ノーマルロック変数の初期化(SPIN_InitLock)

■C 言語 API

```
void SPIN_InitLock(LOCK *pLock);
```

■引数

```
pLock      ロック変数へのポインタ
```

■パケットの構造

```
typedef    struct  {
           UINT8   ucLock;       ロック変数
} LOCK;
```

■機能説明

pLock で示されるロック変数を初期化します。

ロック変数は、スピンロックライブラリによって管理されます。アプリケーションから直接書き換えてはなりません。

pLock は、非キャッシャブルアクセスとなるアドレスでなければなりません。

9.10.2 ノーマルロック取得(SPIN_Lock)

■C 言語 API

```
void SPIN_Lock(LOCK *pLock);
```

■引数

```
pLock      ロック変数へのポインタ
```

■パケットの構造

```
typedef    struct  {
           UINT8   ucLock;       ロック変数
} LOCK;
```

■機能説明

pLock に関連付けられたリソースへのアクセス権をロックします。

pLock は SPIN_InitLock()で初期化済のロック変数へのポインタで無ければなりません。

既にロックされている場合、ロックが解除されるまで本関数内でビジーループ実行により待ち、その後ロックします。

本関数でのロックは、SPIN_Unlock()によって解除します。

9.10.3 ノーマルロック取得試行(SPIN_TryLock)

■C 言語 API

```
INT32 SPIN_TryLock(LOCK *pLock);
```

■引数

pLock　　ロック変数へのポインタ

■リターン値

1:ロック成功
0:ロック失敗

■パケットの構造

```
typedef    struct  {
           UINT8   ucLock;      ロック変数
} LOCK;
```

■機能説明

pLock に関連付けられたリソースへのアクセス権をロックします。
pLock は SPIN_InitLock()で初期化済のロック変数へのポインタで無ければばなりません。
既にロックされている場合は、ロックは失敗し、リターン値として 0 が返ります。
一方、ロックされていない場合は、ロックは成功し、リターン値として 1 が返ります。
本関数でのロックは、SPIN_Unlock()によって解除します。

9.10.4 ノーマルロック解除(SPIN_Unlock)

■C 言語 API

```
void SPIN_Unlock(LOCK *pLock);
```

■引数

pLock　　ロック変数へのポインタ

■パケットの構造

```
typedef    struct  {
           UINT8   ucLock;      ロック変数
} LOCK;
```

■機能説明

pLock に関連付けられたリソースへのアクセス権のロックを解除します。
pLock は SPIN_Lock()でまたは SPIN_TryLock()で取得したロック変数へのポインタで無ければばなりません。

9.10.5 ノーマルロック状態のチェック(SPIN_IsLocked)

■C 言語 API

```
INT32 SPIN_IsLocked(LOCK *pLock);
```

■引数

pLock　　ロック変数へのポインタ

■パケットの構造

```
typedef    struct  {
           UINT8   ucLock;      ロック変数
} LOCK;
```

■リターン値

1:ロックされている

0:ロックされていない

■機能説明

pLock がロックされていれば 1、ロックされていなければ 0 を返します。

9.11 RW ロック

9.11.1 RW ロック変数の初期化(SPIN_InitRWLock)

■C 言語 API

```
void SPIN_InitRWLock(RWLOCK *pRWLock);
```

■引数

pRWLock　　ロック変数へのポインタ

■パケットの構造

```
typedef    struct  {
           UINT8   ucWriteLock;   ライトロック変数
           UINT8   ucReadLock;    リードロック変数
} RWLOCK;
```

■機能説明

pRWLock で示される RW ロック変数を初期化します。

RW ロック変数は、スピンロックライブラリによって管理されます。アプリケーションから直接書き換えてはなりません。

また、pRWLock は、非キャッシャブルアクセスとなるアドレスで無ければなりません。

9.11.2 リードロック獲得(SPIN_ReadLock)

■C 言語 API

```
void SPIN_ReadLock(RWLOCK *pRWLock);
```

■引数

pRWLock　　ロック変数へのポインタ

■パケットの構造

```
typedef    struct  {
           UINT8   ucWriteLock;   ライトロック変数
           UINT8   ucReadLock;    リードロック変数
} RWLOCK;
```

■機能説明

pRWLock に関連付けられたリソースへのリードアクセス権をロックします。

pRWLock は SPIN_InitRWLock()で初期化済のロック変数へのポインタで無ければなりません。

リードロックは、ネスト可能です。ネストの最大値は 255 回です。

既にライトロックされている場合、ライトロックが解除されるまで本関数内でビジーループ実行により待ち、その後リードロックします。

一方、ライトロックされていない場合は、既にリードロックされていても直ちに成功します。

本関数でのロックは、SPIN_ReadUnlock()によって解除します。

9.11.3 リードロック獲得試行(SPIN_ReadTryLock)

■C 言語 API

```
INT32 SPIN_ReadTryLock(RWLOCK *pRWLock);
```

■引数

pRWLock　ロック変数へのポインタ

■リターン値

1:リードロック成功
0:リードロック失敗

■パケットの構造

```
typedef   struct {
          UINT8   ucWriteLock;  ライトロック変数
          UINT8   ucReadLock;   リードロック変数
} RWLOCK;
```

■機能説明

pRWLock に関連付けられたリソースへのリードアクセス権をロックします。
pRWLock は SPIN_InitRWLock()で初期化済のロック変数へのポインタで無ければなりません。
リードロックは、ネスト可能です。ネストの最大値は 255 回です。
既にライトロックされている場合、リードロックは失敗し、リターン値として 0 が返ります。
一方、ライトロックされていない場合は、既にリードロックされていても直ちにリードロックに成功し、リターン値として 1 が返ります。
本関数でのロックは、SPIN_ReadUnlock()によって解除します。

9.11.4 リードロック解除(SPIN_ReadUnlock)

■C 言語 API

```
void SPIN_ReadUnlock(RWLOCK *pRWLock);
```

■引数

pRWLock　ロック変数へのポインタ

■パケットの構造

```
typedef   struct {
          UINT8   ucWriteLock;  ライトロック変数
          UINT8   ucReadLock;   リードロック変数
} RWLOCK;
```

■機能説明

pRWLock に関連付けられたリソースへのリードアクセス権のロックを解除します。
pRWLock は SPIN_ReadLock()でまたは SPIN_ReadTryLock()で取得したロック変数へのポインタで無ければなりません。

9.11.5 リードロック状態のチェック(SPIN_IsReadLocked)

■C 言語 API

```
INT32 SPIN_IsReadLocked(RWLOCK *pRWLock);
```

■引数

pRWLock　　ロック変数へのポインタ

■パケットの構造

```
typedef    struct  {
           UINT8   ucWriteLock;  ライトロック変数
           UINT8   ucReadLock;   リードロック変数
} RWLOCK;
```

■リターン値

1:ロックされている

0:ロックされていない

■機能説明

pRWLock がリードロックされていれば 1、リードロックされていなければ 0 を返します。

9.11.6 ライトロック獲得(SPIN_WriteLock)

■C 言語 API

```
void SPIN_WriteLock(RWLOCK *pRWLock);
```

■引数

pRWLock　　ロック変数へのポインタ

■パケットの構造

```
typedef    struct  {
           UINT8   ucWriteLock;  ライトロック変数
           UINT8   ucReadLock;   リードロック変数
} RWLOCK;
```

■機能説明

pRWLock に関連付けられたリソースへのライトアクセス権をロックします。

pRWLock は SPIN_InitRWLock()で初期化済のロック変数へのポインタで無ければなりません。

既にリードロックまたはライトロックされている場合、それらが解除されるまで本関数内でビジーループ実行により待ち、その後ライトロックします。

本関数でのロックは、SPIN_WriteUnlock()によって解除します。

9.11.7 ライトロック獲得試行(SPIN_WriteTryLock)

■C 言語 API

```
INT32 SPIN_WriteTryLock(RWLOCK *pRWLock);
```

■引数

pRWLock　　ロック変数へのポインタ

■リターン値

1:ライトロック成功
0:ライトロック失敗

■パケットの構造

```
typedef    struct {
           UINT8    ucWriteLock;  ライトロック変数
           UINT8    ucReadLock;   リードロック変数
} RWLOCK;
```

■機能説明

pRWLock に関連付けられたリソースへのライトアクセス権をロックします。
pRWLock は SPIN_InitRWLock()で初期化済のロック変数へのポインタで無ければなりません。
既にリードロックまたはライトロックされている場合、ライトロックは失敗し、リターン値として 0 が返ります。
一方、リードロックおよびライトロックされていない場合は、ライトロックは成功し、リターン値として 1 が返ります。
本関数でのロックは、SPIN_WriteUnlock()によって解除します。

9.11.8 ライトロック解除(SPIN_WriteUnlock)

■C 言語 API

```
void SPIN_WriteUnlock(RWLOCK *pRWLock);
```

■引数

pRWLock　　ロック変数へのポインタ

■パケットの構造

```
typedef    struct {
           UINT8    ucWriteLock;  ライトロック変数
           UINT8    ucReadLock;   リードロック変数
} RWLOCK;
```

■機能説明

pRWLock に関連付けられたリソースへのライトアクセス権のロックを解除します。
pRWLock は SPIN_WriteLock()でまたは SPIN_WriteTryLock()で取得したロック変数へのポインタで無ければなりません。

9.11.9 ライトロック状態のチェック(SPIN_IsWriteLocked)

■C 言語 API

```
INT32 SPIN_IsWriteLocked(RWLOCK *pRWLock);
```

■引数

pRWLock　　ロック変数へのポインタ

■パケットの構造

```
typedef    struct {
           UINT8   ucWriteLock;  ライトロック変数
           UINT8   ucReadLock;   リードロック変数
} RWLOCK;
```

■リターン値

1:ロックされている

0:ロックされていない

■機能説明

pRWLock がライトロックされていれば 1、ライトロックされていなければ 0 を返します。

9.12 セマフォロック

9.12.1 セマフォレジスタの初期化(SPIN_InitSemLock)

■C 言語 API

```
void SPIN_InitSemLock(UINT8 *pucSemRegister);
```

■引数

pucSemRegister　　セマフォレジスタのアドレス

■機能説明

pucSemRegister で示されるセマフォレジスタを初期化します。
pucSemRegister には、使用するマイコンが内蔵するセマフォレジスタのアドレスを指定してください。

9.12.2 セマフォロック獲得(SPIN_SemLock)

■C 言語 API

```
void SPIN_SemLock(UINT8 *pucSemRegister);
```

■引数

pucSemRegister　　セマフォレジスタのアドレス

■機能説明

pucSemRegister に関連付けられたリソースへのアクセス権をロックします。
pucSemRegister には、使用するマイコンが内蔵するセマフォレジスタのアドレスを指定してください。
既にロックされている場合、ロックが解除されるまで本関数内でビジーループ実行により待ち、その後ロックします。
本関数でのロックは、SPIN_SemUnlock()によって解除します。

RENESAS

9.12.3 セマフォロック獲得試行(SPIN_SemTryLock)

■C 言語 API

```
INT32 SPIN_SemTryLock(UINT8 *pucSemRegister);
```

■引数

pucSemRegister　　セマフォレジスタのアドレス

■リターン値

lSts　　　1:ロック成功

0:ロック失敗

■機能説明

pucSemRegister に関連付けられたリソースへのアクセス権をロックします。

pucSemRegister には、使用するマイコンが内蔵するセマフォレジスタのアドレスを指定してください。

既にロックされている場合は、ロックは失敗し、リターン値として 0 が返ります。

一方、ロックされていない場合は、ロックは成功し、リターン値として 1 が返ります。

本関数でのロックは、SPIN_SemUnlock()によって解除します。

9.12.4 セマフォロック解除(SPIN_SemUnlock)

■C 言語 API

```
void SPIN_SemUnlock(UINT8 *pucSemRegister);
```

■引数

pucSemRegister　　セマフォレジスタのアドレス

■機能説明

pucSemRegister に関連付けられたリソースへのアクセス権のロックを解除します。

pucSemRegister には、使用するマイコンが内蔵するセマフォレジスタのアドレスを指定してください。

RENESAS

10. IPI

10.1 概要

IPI は、プロセッサ間通信のプリミティブな機能を提供するソフトウェアです。

IPI では、他 CPU からのデータを受信するための「IPI ポート」を生成することができます。生成可能な IPI ポート数は 8 個までです。

表10.1 IPI 概要

項番	項目	内容
1	使用するハードウェアリソース	(1)プロセッサ間割込み機能 (2)セマフォレジスタ(スピンロックライブラリ内でアクセスします)
2	本ソフトウェアが使用するソフトウェア部品	スピンロックライブラリ
3	本ソフトウェアを使用している他のソフトウェア部品	(1) カーネル(リモートサービスコール) (2) RPC ライブラリ

10.2 IPI の構成

IPI は、API 関数とプロセッサ間割込みハンドラ関数から構成されます。

API 関数は、IPI の API を処理する関数です。

プロセッサ間割込みハンドラ関数は、プロセッサ間割込みが発生したときに実行する処理関数です。ユーザは、この関数を割込みハンドラとして、適切にカーネルに登録しなければなりません（IPI を初期化する API「IPI_init()」では、カーネルへの割込みハンドラ定義は行っていません）。

10.3 ポート ID

各 CPU には、0～7 の値を持つポート ID があります。ポート ID は、表 10.2に示すようにプロセッサ間割込みと 1 対 1 に対応しています。

表10.2 ポート ID とプロセッサ間割込みの関係

ポート ID	ベクタ番号	プロセッサ間割込みのレベル
0	21	15
1	22	14
2	23	13
3	24	12
4	25	11
5	26	10
6	27	9
7	28	8

10.4　動作概要

まず、IPI のコンフィギュレーションで IPI で扱うポート ID を定義します。つまり、IPI が占有するプロセッサ間割込みのベクタ番号を決めます。
IPI を用いた通信を受理するアプリケーションでは、最初に IPI_create()を用いて IPI ポートを生成します。このとき、生成するポート ID と、データ受信時に実行されるコールバック関数を登録します。
他 CPU の IPI ポートへデータを送信するアプリケーションでは、IPI_send()を使用します。このとき、対象 CPUID とポート ID、および送信するデータを指定します。送信データは、1 バイト+4 バイトです。
IPI_send()により、送信先 CPU にプロセッサ間割込みが要求されます。送信先 CPU では、この割込みによって IPI_create()で登録されたコールバック関数が呼び出されます。コールバック関数には、送信されたデータが渡されます。

10.5　注意事項

各 API 関数では、IPI ポートに対するセマフォロックを取得する場合があります。同じ CPU の別のプログラムから、同じ IPI ポートのセマフォロックを取得する API を呼び出すと、デッドロックする可能性があります。このような場合の同一 CPU 内での排他制御は、API 関数を呼び出す側で行ってください。
デッドロックについては、「9.3.1 同一 CPU 内での排他制御とデッドロック」を参照してください。

10.6　提供ファイル

<RTOS_INST>¥os¥include¥

ipi.h	API 定義ヘッダ

<SAMPLE_INST>¥R0K572650D000BR¥cpuid1¥ipi¥

ipi_config.h	コンフィギュレーションファイル（「10.7.1　コンフィギュレーション」参照）
ipi_defs.h	内部定義
ipi.c	ソースコード

<SAMPLE_INST>¥R0K572650D000BR¥cpuid2¥ipi¥

ipi_config.h	コンフィギュレーションファイル（「10.7.1　コンフィギュレーション」参照）
ipi_defs.h	内部定義
ipi.c	ソースコード

なお、各ディレクトリの ipi_defs.h と ipi.c は同じ内容です。

10.7 コンフィギュレーションとビルド

IPIは、CPU毎にコンフィギュレーションしてください。

10.7.1 コンフィギュレーション

ipi_config.hにて、以下を定義します。

(1) 使用するポートの定義

各IDのポートを使用可能とするかどうかを定義します。

プロセッサ間割込みをIPI以外の目的で使用する場合は、そのポートIDに対する定義を0にしてください。IPI_create()では、0に定義されたポートIDが指定されるとエラーを返します。

```
/*** Defines using ports ***/
#define PORT0   1   /* 1:use PORT0, 0:not use PORT0 */
#define PORT1   1   /* 1:use PORT1, 0:not use PORT1 */
#define PORT2   1   /* 1:use PORT2, 0:not use PORT2 */
#define PORT3   1   /* 1:use PORT3, 0:not use PORT3 */
#define PORT4   1   /* 1:use PORT4, 0:not use PORT4 */
#define PORT5   1   /* 1:use PORT5, 0:not use PORT5 */
#define PORT6   1   /* 1:use PORT6, 0:not use PORT6 */
#define PORT7   1   /* 1:use PORT7, 0:not use PORT7 */
```

本定義にあたっては、以下に留意してください。

(1) 他CPUからのリモートサービスコールを受理する場合
cfgファイルのremote_svc.num_server(SVCサーバ数)に0以外を指定した場合は、remote_svc.ipi_portidで指定したIPIポートIDが有効となるように定義してください。
なお、remote_svc.ipi_portidの割込みレベルは、system.system_IPL以下でなければなりません。そうでない場合は、cfg72mpがエラーを報告します。

(2) 他CPUからのRPCを受理する場合
rpc_init()でrpc_config.ulTableSize(登録可能なRPCサーバ数)に0以外を指定した場合は、rpc_config.ulIPIPortIDで指定したIPIポートIDが有効となるように定義してください。
なお、rpc_config.ulIPIPortIDの割込みレベルは、system.system_IPL以下でなければなりません。そうでない場合の動作は保証されません。

(2) 各ポートで使用するセマフォレジスタアドレス

各ポートでは、プロセッサ間の排他制御のためにセマフォレジスタを使用します。ここでは、各ポートで使用するセマフォレジスタのアドレスを定義します。なお、「(1) 使用するポートの定義」で 0 に定義したポートに対するセマフォレジスタアドレスの定義は無視されます。

両 CPU を通じて、異なるポートに同じセマフォレジスタアドレスを設定してはなりません。

```
/*** Defines using ports ***/
#define PORT0_SEMADR 0xFFFC1E00
#define PORT1_SEMADR 0xFFFC1E04
#define PORT2_SEMADR 0xFFFC1E08
#define PORT3_SEMADR 0xFFFC1E0C
#define PORT4_SEMADR 0xFFFC1E10
#define PORT5_SEMADR 0xFFFC1E14
#define PORT6_SEMADR 0xFFFC1E18
#define PORT7_SEMADR 0xFFFC1E1C
```

(3) プロセッサ間割込み制御レジスタのベースアドレス

CPUID に関わらず、割込みコントローラの C0IPCR15 レジスタのアドレスを、定数式で指定してください。

```
/*** Defines C0IPCR15 register address ***/
#define ADDR_C0IPCR15 0xFFFC1C00
```

(4) 割込みハンドラのスタックサイズ

各ポート用のプロセッサ間割込みハンドラで使用するスタックサイズを定義します。なお、「(1) 使用するポートの定義」で 0 に定義したポートに対するスタックサイズの定義は無視されます。

```
/*** Defines stack size for interupt handlers ***/
#define PORT0_STKSZ 0x400
#define PORT1_STKSZ 0x400
#define PORT2_STKSZ 0x400
#define PORT3_STKSZ 0x400
#define PORT4_STKSZ 0x400
#define PORT5_STKSZ 0x400
#define PORT6_STKSZ 0x400
#define PORT7_STKSZ 0x400
```

10.7.2 ビルド

ipi.c をコンパイルし、API 関数を使用するプログラム(例えば RPC)とリンクしてください。

なお、ipi.c は cfg72mp が出力する mycpuid.h をインクルードしており、この中で定義されているマクロ"MYCPUID"を使用しています。

また、割込みハンドラは cfg ファイルに登録するなどの方法で、カーネルに定義する必要があります。

IPI のセクションは、「17.5.2 セクション一覧」を参照してください。

10.8 API 関数

表 10.3に API 関数一覧を示します。

表10.3 API 関数一覧

項番	API 名称	機能
1	IPI_init	IPI の初期化
2	IPI_create	IPI ポートの生成
3	IPI_delete	IPI ポートの削除
4	IPI_send	IPI ポートへの送信

10.8.1 ヘッダファイル

ipi.h をインクルードしてください。

10.8.2 基本データタイプ

types.h で定義される基本データタイプを使用します。
types.h については、「19. types.h」を参照してください。

10.8.3 IPI の初期化(IPI_init)

■C 言語 API

```
INT32 IPI_init(void);
```

■リターン値

IPI_E_OK	正常終了
IPI_E_NOINIT	IPI が未初期化（MYCPUID=2 の場合のみ検出）

■機能説明

IPI を初期化します。
CPUID#2 から呼び出す場合は、事前に CPUID#1 側で IPI_init()が完了している必要があります。
本 API は、各 CPU で最初に一度だけ呼び出すようにしてください。

■API 関数実行中にロックする自 CPU 側の IPI ポート

ipi_config.h で「使用する」と定義された全ポート

■注意

後述の割込みハンドラは、本初期化関数内で動的にカーネルに定義する実装が一般的ですが、出荷時の形態ではこれを行っていません。これは、割込みベクタテーブルを ROM 化する場合には、動的な割込みハンドラの定義ができないためです。後述の割込みハンドラは、カーネルコンフィギュレーション時に適切なベクタ番号(プロセッサ間割込みのベクタ番号)に定義してください。

RENESAS

10.8.4 IPI ポートの生成(IPI_create)

■C 言語 API

```
INT32 IPI_create(
          UINT32  ulPortID,
          void    (*pCallBack)(UINT8, UINT32));
```

■引数

ulPortID　対象ポート ID
pCallBack　コールバック関数のアドレス

■リターン値

IPI_E_OK　正常終了
IPI_E_NOINIT　IPI が未初期化
IPI_E_PAR　ulPortID が 8 以上
IPI_E_STATE　生成済みのポート ID を指定
コンフィギュレーションファイルで未使用に定義しているポート ID を指定

■機能説明

自 CPU に、ulPortID で指定されたポート ID の IPI ポートを生成します。
pCallBack には、当該ポートへの送信があったときに呼び出されるコールバックルーチンのアドレスを指定します。コールバック関数については、「10.10 コールバック関数」を参照してください。

■API 関数実行中にロックする自 CPU 側の IPI ポート

ulPortID で指定されたポート

10.8.5 IPI ポートの削除(IPI_delete)

■C 言語 API

```
INT32 IPI_delete(
          UINT32  ulPortID);
```

■引数

ulPortID　対象ポート ID

■リターン値

IPI_E_OK　正常終了

IPI_E_NOINIT	IPI が未初期化
IPI_E_PAR	ulPortID が 8 以上
IPI_E_STATE	生成されていないポート ID を指定
	送信中のポート ID を指定
	コンフィギュレーションファイルで未使用に定義しているポート ID を指定

■機能説明

ulPortID で指定されたポート ID の IPI ポートを削除します。

■API 関数実行中にロックする自 CPU 側の IPI ポート

ulPortID で指定されたポート

10.8.6 IPI ポートへの送信(IPI_send)

■C 言語 API

```
INT32 IPI_send(
          UINT32 ulCpuID,
          UINT32  ulPortID,
          UINT8  ucCode,
          UINT32  ulData );
```

■引数

ulCpuID	対象 CPUID
ulPortID	ポート ID
ucCode	送信コード
ulData	送信データ

■リターン値

IPI_E_OK	正常終了
IPI_E_NOINIT	IPI が未初期化
IPI_E_PAR	ulPortID が 8 以上
IPI_E_STATE	生成されていないポート ID を指定 コンフィギュレーションファイルで未使用に定義しているポート ID を指定

■機能説明

ulCpuID で指定された CPU の ulPortID で指定されたポートに、ucCode と ucData を送信します。

ucCode と ulData は、送信先ポートのコールバック関数に渡されます。コールバック関数については、「10.10 コールバック関数」を参照してください。

本関数では、対象 CPU に対してプロセッサ間割込みを要求します。その後、対象 CPU で割込みが受理されるまで、本関数内でビジーウェイトします。

IPI が自 CPU への割込みをサポートしている場合は、自 CPU に対しても送信可能ですが、その場合は該当するプロセッサ間割込みが受理可能な状態でコールしなければなりません。そうでない場合、CPU がデッドロックします。

■API 関数実行中にロックする自 CPU 側の IPI ポート

ulCpuID が自 CPU の場合は、ulPortID で指定されたポートをロックします。

10.9　プロセッサ間割込みハンドラ

プロセッサ間割込みハンドラは、各 CPU の各ポート毎に存在します。
表 10.4に、プロセッサ間割込みハンドラを示します。

表10.4　プロセッサ間割込みハンドラ

ポート ID	ベクタ番号	プロセッサ間割込みのレベル	割込みハンドラ関数名
0	21	15	IPI_Port0Handler
1	22	14	IPI_Port1Handler
2	23	13	IPI_Port2Handler
3	24	12	IPI_Port3Handler
4	25	11	IPI_Port4Handler
5	26	10	IPI_Port5Handler
6	27	9	IPI_Port6Handler
7	28	8	IPI_Port7Handler

前述したように、出荷時の構成では、IPI_init()ではカーネルへの割込みハンドラ定義は行っていないため、ユーザ側でカーネルにこれらのハンドラを登録する必要があります。
出荷時は、以下のようになっています。

(1)　割込みハンドラは、HI7200/MPの「ダイレクト割込みハンドラ」として実装
(2)　割込みハンドラのカーネルへの定義は、サンプルのcfgファイルにて実施

10.10 コールバック関数

IPI_create()では、データ受信時に呼び出されるコールバック関数を指定します。
このコールバック関数は以下の形式で記述してください。関数名は任意です。

```
void callback(UINT8 ucCode, UINT32 ulData)
```

ucCode, ulData には、それぞれ IPI_send()で指定した値が渡されます。
コールバック関数は、IPI の各プロセッサ間割込みハンドラから呼び出されます。割込みハンドラの使用スタックサイズを算出する際は、このことを考慮してください。

11. SH2A-DUAL 用キャッシュサポートライブラリ

11.1 概要

本キャッシュサポートライブラリは、SH2A-DUAL の各 CPU が持つローカルキャッシュをメンテナンスする関数を提供します。キャッシュサポートライブラリ関数は、キャッシュと実メモリのコヒーレンシを維持するなどの目的で利用します。

なお、本ライブラリでは、各 CPU のキャッシュ間のコヒーレンシを維持するための機能はサポートしていません。

表11.1 SH2A-DUAL 用キャッシュサポートライブラリ概要

項番	項目	内容
1	使用するハードウェアリソース	SH2A-DUAL 内蔵キャッシュ
2	本ソフトウェアが使用するソフトウェア部品	なし
3	本ソフトウェアを使用している他のソフトウェア部品	なし

11.2 留意事項

(1) キャッシュサポートライブラリは、使い方を誤るとキャッシュと実メモリのコヒーレンシを確保できなくなるなど、その後のシステムの動作に影響を与える可能性があるため、十分注意して使用してください。使用するマイコンのキャッシュ仕様と、本ライブラリの振舞いを十分に理解した上で使用するようにしてください。

(2) キャッシュサポートライブラリは単なる関数であるため、処理中にタスクスイッチや割込み受理が発生する場合があります。必要なら、これらを抑止した状態でキャッシュサポートライブラリ関数を呼び出すようにしてください。

11.3 ディレクトリ・ファイル構成

<RTOS_INST>¥os¥include¥
 sh2adual_cache.h API 定義ヘッダ
<RTOS_INST>¥os¥lib¥debug¥
 sh2adual_cache.lib ライブラリ (デバッグ情報付き)
<RTOS_INST>¥os¥lib¥release¥
 sh2adual_cache.lib ライブラリ (デバッグ情報なし)
<RTOS_INST>¥os¥sh2adual_cache¥ ライブラリ生成用ワークスペース等
<RTOS_INST>¥os¥sh2adual_cache¥sh2adual_cache¥ ライブラリ生成用プロジェクト等
<RTOS_INST>¥os¥sh2adual_cache¥sh2adual_cache¥include¥ 内部定義
<RTOS_INST>¥os¥sh2adual_cache¥sh2adual_cache¥source¥ ソースコード
<RTOS_INST>¥os¥sh2adual_cache¥sh2adual_cache¥Debug¥ コンフィギュレーションディレクトリ (デバッグ情報付き)
<RTOIS_INST>¥os¥sh2adual_cache¥sh2adual_cache¥Release¥ コンフィギュレーションディレクトリ (デバッグ情報なし)

11.4 ライブラリのビルド

通常はライブラリをビルドする必要はありません。デバッグなどの目的でライブラリをビルドする場合は、提供する High-performance Embedded Workshop ワークスペース(sh2adual_cache.hws)を用いてビルドしてください。

11.5 システムのビルド

API 関数を使用するプログラムは、キャッシュサポートライブラリとリンクする必要があります。
本ライブラリのセクションは、「17.5.2 セクション一覧」を参照してください。

11.6 API 関数

表 11.2に API 関数一覧を示します。

表11.2 API 関数一覧

項番	分類	API 名称	機能
1	初期化	sh2adual_ini_cac	キャッシュの初期化
2	クリア	sh2adual_clr_cac	キャッシュのクリア
3	フラッシュ	sh2adual_fls_cac	キャッシュのフラッシュ
4	無効化	sh2adual_inv_cac	キャッシュの無効化

11.6.1 ヘッダファイル

include¥sh2adual_cache.h をインクルードしてください。

11.6.2 基本データタイプ

types.h で定義される基本データタイプを使用します。
types.h については、「19. types.h」を参照してください。

11.6.3　キャッシュの初期化(sh2adual_ini_cac)

■C 言語 API

```
INT32 sh2adual_ini_cac(UINT32 ulCacAtr);
```

■引数

ulCacAtr　キャッシュ初期化属性

■リターン値

CAC_E_OK　正常終了

■機能説明

キャッシュを初期化します。具体的には、指定された ulCacAtr に基づいて、以下に示すように CCR1 レジスタを更新します。CCR1 レジスタの更新は、SR.IMASK=15 の状態で行います。

ulCacAtr には、以下の各項目の論理和を指定できますが、ulCacAtr に指定された値のエラーチェックは一切行いません。

また、本処理では ulCacAtr の指定内容に関わらず、CCR1.ICF と OCF ビットに 1 を書き込みますので、本処理呼び出し以前のキャッシュ内容は全て廃棄されます。

- TCAC_IC_ENABLE(H'00000100)
  これを指定すると命令キャッシュを有効(CCR1.ICE=1)にします。指定の無い場合は無効(CCR1.ICE=0)にします。
- TCAC_OC_ENABLE(H'00000001)
  これを指定するとオペランドキャッシュを有効(CCR1.OCE=1)にします。指定の無い場合は無効(CCR1.OCE=0)にします。
- TCAC_OC_WT(0x00000002)
  これを指定するとキャッシュ対象領域への書込みモードをライトスルーモード(CCR1.WT=1)とします。指定の無い場合はライトバックモード(CCR1.WT=0)とします。

なお、本 API では、CCR2 レジスタは操作しません。

また、キャッシュがモジュールスタンバイの時には、本 API を呼び出してはなりません。

11.6.4 キャッシュのクリア(sh2adual_clr_cac)

■C 言語 API

```
INT32 sh2adual_clr_cac(void *pStart, void *pEnd, UINT32 ulMode);
```

■引数

pStart	クリア先頭アドレス
pEnd	クリア最終アドレス
ulMode	対象キャッシュ指定

■リターン値

CAC_E_OK	正常終了
CAC_E_PAR	パラメータエラー(pStart>pEnd、ulMode 不正)

■機能説明

キャッシュをクリアします。即ち、キャッシュ内容を無効化すると共に、オペランドキャッシュにメモリに書き戻していないデータがあれば、それをメモリに書き戻します。キャッシュロックモードを使用している場合でも、ロックされているキャッシュエントリはクリアされます。ただし、ロックは解除されません。

対象のキャッシュは、ulMode によって決まります。ulMode には、以下のいずれかを指定できます。

- TC_FULL(H'00000000) : 命令キャッシュ・オペランドキャッシュの両方を対象とする。
- TC_EXCLUDE_IC(H'00000001): 命令キャッシュを対象外とする(オペランドキャッシュのみ)
- TC_EXCLUDE_OC(H'00000002) : オペランドキャッシュを対象外とする(命令キャッシュのみ)

ただし、対象キャッシュが Disable の場合は、そのキャッシュに対しては何も処理しません。

クリアするアドレス範囲は、pStart と pEnd によって決まります。pStart は 16 の倍数に切り捨て、pEnd は 16 の倍数-1 に切り上げて扱います。

(1) アドレス範囲を指定

ulMode で決まるキャッシュに対し、アドレスが pStart～pEnd のエントリをクリアします。ただし、指定された範囲に非キャッシャブル領域が含まれる場合は、その範囲については何も処理を行いません。

オペランドキャッシュが対象に含まれる場合(ulMode が TC_FULL または TC_EXCLUDE_IC の場合)は、そのエントリがダーティ(メモリに書き出されていない)であれば、クリアする前にメモリへのコピーバックを行います。

これらの処理は、メモリ割り付けキャッシュを操作することにより行います。これらの処理を実行するときの SR.IMASK は呼び出し時と同じです。本関数の処理中に割込みを受け付けたくない場合は、割込みをマスクした状態で呼び出してください。

(2) 全エントリを対象とする

pStart=0, pEnd=H'ffffffff を指定すると、ulMode で決まる対象キャッシュの全エントリをクリアします。この場合、本関数は以下のように処理します。

(a) ulModeがTC_FULLまたはTC_EXCLUDE_OCの場合は、CCR1.ICF=1を設定することで、命令キャッシュの全エントリを無効化します。CCR1の更新は、SR.IMASK=15の状態で行います。

(b) (a)の後、ulModeがTC_FULLまたはTC_EXCLUDE_ICの場合は、オペランドキャッシュのメモリ割り付けキャッシュの全エントリについて、V=0, U=0を書き込みます。この時、ダーティ(U=1：メモリに書き出されていない)であったエントリの内容はメモリにコピーバックされます。この処理を実行するときのSR.IMASKは呼び出し時と同じです。本関数の処理中に割込みを受け付けたくない場合は、割込みをマスクした状態で呼び出してください。

本関数では、CCR1 レジスタを読み出します。本関数処理中に CCR1 レジスタが変更された場合の動作は未定義です。

また、キャッシュがモジュールスタンバイの時には、本 API を呼び出してはなりません。

RENESAS

11.6.5 オペランドキャッシュのフラッシュ(sh2adual_fls_cac)

■C 言語 API

```
INT32 sh2adual_fls_cac(void * pStart, void *pEnd);
```

■引数

pStart　　フラッシュ先頭アドレス
pEnd　　　フラッシュ最終アドレス

■リターン値

CAC_E_OK　　正常終了
CAC_E_PAR　パラメータエラー(pStart>pEnd)

■機能説明

オペランドキャッシュをフラッシュします。すなわち、メモリに書き出されていない内容をメモリに書き出します(コピーバック)。キャッシュロックモードを使用している場合でも、ロックされているキャッシュエントリはフラッシュされます。ただし、ロックは解除されません。

フラッシュするアドレス範囲は、pStart と pEnd によって決まります。pStart は 16 の倍数に切り捨て、pEnd は 16 の倍数-1 に切り上げて扱います。
なお、オペランドキャッシュが Disable、またはライトスルーモードの場合は、本関数は何もせずにリターンします。

(1) アドレス範囲を指定

アドレスが pStart～pEnd に対応するオペランドキャッシュエントリをフラッシュします。すなわち、該当するオペランドキャッシュエントリがメモリに書き出されていない場合、メモリに書き出します。ただし、指定された範囲に非キャッシャブル領域が含まれる場合は、その範囲については何も処理を行いません。
この処理は、メモリ割り付けキャッシュを操作することにより行います。この処理を実行するときの SR.IMASK は呼び出し時と同じです。本関数の処理中に割込みを受け付けたくない場合は、割込みをマスクした状態で呼び出してください。

(2) 全エントリを対象とする

pStart=0, pEnd=H'ffffffff を指定すると、オペランドキャッシュの全エントリをフラッシュします。具体的には、オペランドキャッシュのメモリ割り付けキャッシュの全エントリについてそれを読み出し、有効(V=1)なエントリについて V=1, U=0 を書き込みます。この処理を実行するときの SR.IMASK は呼び出し時と同じです。本関数の処理中に割込みを受け付けたくない場合は、割込みをマスクした状態で呼び出してください。

本関数では、CCR1 レジスタを読み出します。処理中に CCR1 レジスタが変更された場合の動作は未定義です。
また、キャッシュがモジュールスタンバイの時には、本 API を呼び出してはなりません。

11.6.6 キャッシュの無効化(sh2adual_inv_cac)

■C 言語 API

```
INT32 sh2adual_inv_cac(UINT32 ulMode);
```

■引数

ulMode　　対象キャッシュ指定

■リターン値

CAC_E_OK　正常終了

CAC_E_PAR　パラメータエラー(ulMode 不正)

■機能説明

キャッシュを無効化します。オペランドキャッシュ内のデータは、実メモリにライトバックされず、破棄されます。

対象のキャッシュは、ulMode によって決まります。ulMode には以下のいずれかを指定します。

- TC_FULL(H'00000000)：命令キャッシュ・オペランドキャッシュ両方を対象にする
- TC_EXCLUDE_IC(H'00000001)：命令キャッシュを対象外とする
- TC_EXCLUDE_OC(H'00000002)：オペランドキャッシュを対象外とする

対象キャッシュが Disable の場合は、そのキャッシュに対しては何も処理しません。

本関数は、ulMode に応じて、CCR1 レジスタを以下のように更新します。CCR1 レジスタの更新は、SR.IMASK=15 の状態で行います。

(1) ulMode = TC_FULLの場合
命令キャッシュがEnableの場合は、CCR1.ICFに1を設定します。
また、オペランドキャッシュがEnableの場合は、CCR1.OCFに1を設定します。

(2) ulMode = TC_EXCLUDE_ICの場合
オペランドキャッシュがEnableの場合は、CCR1.OCFに1を設定します。

(3) ulMode = TC_EXCLUDE_OCの場合
命令キャッシュがEnableの場合は、CCR1.ICFに1を設定します。

また、キャッシュがモジュールスタンバイの時には、本 API を呼び出してはなりません。

12. アプリケーションプログラムの記述方法

12.1 FPU について

SH2A-FPU を使用する場合は、浮動小数点演算を行わない場合も、「20. FPU に関する注意」を一読してください。

12.2 タスク

(1) 記述方法

タスクは、図 12.1に示すように通常の C 言語関数として記述します。タスクを終了する場合は、ext_tsk または exd_tsk サービスコールを用いて終了してください。ext_tsk, exd_tsk サービスコールを発行せずにタスク関数からリターンした場合は、ext_tsk サービスコールを発行した場合と同じ動作となります。

```
#include "kernel.h"
#pragma noregsave(Task)       ←タスクのエントリ関数はレジスタを保証する必要はないので、#pragma
                                noregsave を指定することができます。
void Task(VP_INT exinf)       ←タスク生成時に TA_ACT 属性が指定されて起動された場合、および
{                               act_tsk で起動された場合には、パラメータとしてタスク生成時に指
                                定された exinf が渡され、sta_tsk で起動された場合は sta_tsk で
    /* タスクの処理 */          指定された stacd が渡されます。
    if(…)
        ext_tsk();            ←タスクを終了する場合は、ext_tsk または exd_tsk
                                サービスコールを呼び出してください。
}                             ←関数の終了で、ext_tsk が自動的に呼び出されます。
```

図12.1 タスクの記述例

タスクは、終了せずに無限ループしても構いません。その例を、図 12.2に示します。

```
#include "kernel.h"
#pragma noregsave(Task)
void Task(VP_INT exinf)
{
    while(1) {
        /* タスクの処理 */
    }
}
```

図12.2 無限ループするタスクの例

(2) レジスタ使用規約

表 12.1に、タスクのレジスタ使用規約を示します。デバッグやアセンブリ言語でタスクを作成する場合の参考にしてください。

表12.1 タスクのレジスタ使用規約

No	レジスタ	保証 *1	初期値
1	PC	不要	タスクのアドレス
2	SR	*2	H'00000000
3	R0～R3	不要	不定
4	R4	不要	TA_ACT属性またはact_tskで起動された場合はタスク生成時に指定した exinf、sta_tsk で起動された場合は sta_tsk で指定された stacd
5	R5～R14, MACH, MACL, GBR	不要	不定
6	R15	必要	タスクのスタック領域最終アドレス
7	PR	必要	不定
8	TBR	*3	*3
9	[SH2A-FPU] FPSCR	不要 *4	H'00040001
10	[SH2A-FPU] FPUL, FR0～FR15	不要 *4	不定

【注】 *1 タスク開始関数からリターンする時点で、レジスタの値が起動時と同一であることを要求するかどうか。

*2 CPU ロック状態である場合を除き、IMASK=0 のみ保証必要です。

*3 system.tbr の設定に依存します。

(1) system.tbr=NOMANAGE の場合：カーネルは TBR を一切操作しません。

(2) system.tbr=FOR_SVC の場合：変更禁止です。

(3) system.tbr=TASK_CONTEXT の場合：保証不要です。初期値は不定です。

*4 TA_COP1 属性の場合のみ使用可能であり、保証不要です。

12.3　タスク例外処理ルーチン

(1) 記述方法

タスク例外処理ルーチンは、図 12.3に示すように通常の C 言語関数として記述します。

```
#include "kernel.h"
#pragma noregsave(Texrtn)                    ←タスク例外処理ルーチン関数はレジスタを保証す
                                               る必要はないので、#pragma noregsave を指定
                                               することができます。
void Texrtn(TEXPTN texptn,VP_INT exinf)     ←パラメータとして、例外要因と拡張情報が
{                                              渡されます。
    /* タスク例外処理ルーチンの処理 */
}
```

図12.3　タスク例外処理ルーチンの記述例

(2)　レジスタ使用規約

表 12.2に、タスク例外処理ルーチンのレジスタ使用規約を示します。デバッグやアセンブリ言語でタスク例外処理ルーチンを作成する場合の参考にしてください。

表12.2　タスク例外処理ルーチンのレジスタ使用規約

No	レジスタ	保証 *1	初期値
1	PC	不要	タスク例外処理ルーチンのアドレス
2	SR	*2	0
3	R0～R3	不要	不定
4	R4	不要	例外要因パターン
5	R5	不要	タスクの拡張情報
6	R6～R14, MACH, MACL, GBR	不要	不定
7	R15	必須	タスクのスタック領域を指しています。
8	PR	必須	不定
9	TBR	*3	*3
10	[SH2A-FPU] FPSCR,	不要 *4	H'00040001
11	[SH2A-FPU] FPUL, FR0～FR15	不要 *4	不定

【注】 *1　タスク例外処理ルーチン開始関数からリターンする時点で、レジスタの値が起動時と同一であることを要求するかどうか。

*2　CPU ロック状態である場合を除き、IMASK=0 のみ保証必要です。

*3　system.tbr の設定に依存します。

(1) system.tbr=NOMANAGE の場合：カーネルは TBR を一切操作しません。

(2) system.tbr=FOR_SVC の場合：変更禁止です。

(3) system.tbr=TASK_CONTEXT の場合：保証必須です。初期値は不定です。

*4　TA_COP1 属性の場合のみ

12.4 拡張サービスコールルーチン

(1) 記述方法

拡張サービスコールルーチンは、図 12.4に示すように通常の C 言語関数として記述します。

```
#include "kernel.h"
ER_UINT Svcrtn(VP_INT par1, VP_INT par2) ←拡張サービスコールルーチンには、cal_svc で指定
{                                          したパラメータが渡されます。
    /* 拡張ｻｰﾋﾞｽｺｰﾙﾙｰﾁﾝの処理 */
    return E_OK;                          ←発行元にリターン値を返します。
}
```

図12.4 拡張サービスコールルーチンの記述例

(2) レジスタ使用規約

拡張サービスコールルーチンは、cal_svc, ical_svc サービスコールによって、単なる関数呼び出しを行った場合と同様に呼び出されます。したがって、拡張サービスコールルーチンのレジスタ使用規約は通常の C 言語関数と同じです。詳細は、『SuperH™ RISC engine C/C++コンパイラ ユーザーズマニュアル』を参照してください。

R4～R7 には、cal_svc で指定した第 1～第 4 パラメータが設定されます。

拡張サービスコールルーチンで FPU のレジスタを使用できるかは、cal_svc, ical_svc 発行元に依存しますので、注意してください。

12.5 割込みハンドラ

12.5.1 割込みハンドラの種類

割込みハンドラには、ノーマル割込みハンドラとダイレクト割込みハンドラがあります。

ノーマル割込みハンドラは、割込み発生時にカーネルを介して起動され、通常の C 言語関数として記述することができます。

ダイレクト割込みハンドラは、CPU の割込み例外ベクタテーブルに登録されます。割込み発生時にカーネルを介さずに起動されるため、高速です。ダイレクト割込みハンドラは、割込み関数(#pragma interrupt ディレクティブを使用)として記述する必要があります。

いずれの割込みハンドラも、非タスクコンテキストとして動作します。

12.5.2 レジスタバンク

本カーネルでは、割込みがレジスタバンクを使うかどうかで、ダイレクト割込みハンドラの記述方法、およびノーマル割込みハンドラの定義方法が異なります。表 12.3に割込みがレジスタバンクを使うかどうかの区別方法を示します。

表12.3 レジスタバンク

kernel_intspec.h		cfg ファイル	割込みレベル	レジスタバンク
INTSPEC_IBNR_ADR	割込み番号に対する INTSPEC_NOBANK_VEC ???の定義	system.regbank		
0	−	−	−	使用しない
0 以外	あり	−	−	使用しない
	なし	NOTUSE	−	使用しない
		ALL	−	使用する
		BANKLEVELxx	system.regbank に指定のないレベル	使用しない ノーマル割込みハンドラを定義する場合は、VTA_REGBANK 属性を指定してはなりません。
			system.regbank に指定のあるレベル	使用する ノーマル割込みハンドラを定義する場合は、VTA_REGBANK 属性の指定が必須です。

12.5.3 ノーマル割込みハンドラ

ノーマル割込みハンドラの定義時には、その割込みがレジスタバンクを使用する場合はVTA_REGBANK属性を指定しなければなりません。逆に、その割込みがレジスタバンクを使用しない場合は、VTA_REGBANK属性を指定してはなりません(表12.3参照)。

なお、カーネル割込みマスクレベル(system.system_IPL)よりも高いレベルの割込み(NMIを含む)のハンドラは、ダイレクト割込みハンドラとして記述・定義する必要があります。**これらのハンドラをノーマル割込みハンドラとして記述・定義した場合、システムの正常な動作は保証されません。**

(1) 記述方法

ノーマル割込みハンドラは、図12.5のように通常のC言語関数として記述します。

```
#include "kernel.h"
#pragma noregsave(Inh)          ←レジスタバンクを使用する割込みの場合、レジスタを保証
                                  する必要はないので、#pragma noregsave を指定するこ
                                  とができます。
void Inh(void)                  ←ハンドラは、引数もリターン値も持たない関数として記述
{                                 します。
    /* ハンドラの処理 */
}
```

図12.5 ノーマル割込みハンドラの記述例

(2) レジスタ使用規約

表 12.4に、ノーマル割込みハンドラのレジスタ使用規約を示します。デバッグやアセンブリ言語でノーマル割込みハンドラを作成する場合の参考にしてください。

表12.4 ノーマル割込みハンドラのレジスタ使用規約

No	レジスタ	保証 *1	初期値
1	PC	不要	ノーマル割込みハンドラのアドレス
2	SR	*2	IMASK：割込みレベル。ハンドラ実行中は、IMASK を自割込みレベルよりも下げてはなりません。 その他のビット：不定
3	R0～R7	不要	不定
4	R8～R14, MACH, MACL, GBR	必須 *3	不定
5	R15	必須	割込みハンドラスタックを指しています。 割込み発生時には、カーネルの出入口処理でスタックを割込みハンドラ用のスタックに切り替えます。すべてのノーマル割込みハンドラは、同じ割込みハンドラスタックを使用します。 割込みハンドラスタックのサイズは、system.stack_size で指定します。割込みはネストする可能性があるので、割込みハンドラスタックのサイズは割込みのネストを考慮して算出する必要があります。詳しくは、「18.7 ノーマル割込みハンドラのスタック(system.stack_size)」を参照してください。
6	PR	必須	不定
7	TBR	*4	*4
8	[SH2A-FPU] FPSCR, FPUL, FR12～FR15	必須 *5	不定

【注】 *1 ノーマル割込みハンドラ開始関数からリターンする時点で、レジスタの値が起動時と同一であることを要求するかどうか。

*2 IMASK ビットのみ保証必須です。

*3 レジスタバンクを使用する割込み(表 12.3参照)の場合のみ、保証不要です。

*4 system.tbr の設定に依存します。

(1) system.tbr=NOMANAGE の場合：カーネルは TBR を一切操作しません。

(2) system.tbr=FOR_SVC の場合：変更禁止です。

(3) system.tbr=TASK_CONTEXT の場合：保証必須です。初期値は不定です。

*5 ハンドラで FPU を使用する場合は、「20.3 各種ハンドラ等で浮動小数点演算を行う場合」を参照してください。

12.5.4　ダイレクト割込みハンドラ

ダイレクト割込みハンドラを定義するときには、VTA_DIRECT 属性の指定が必要です。

なお、カーネル割込みマスクレベル(system.system_IPL)よりも高いレベルの割込み(NMI を含む)のハンドラは、ダイレクト割込みハンドラとして記述・定義する必要があります。**これらのハンドラをノーマル割込みハンドラとして記述・定義した場合、システムの正常な動作は保証されません。**

(1) 記述方法

ダイレクト割込みハンドラは、図 12.6に示すように割込み関数として記述します。指定すべき割込み関数仕様は、表 12.5に示すようにケースによって異なることに注意してください。

```
#include "kernel.h"
#define stksz 512                                                          (1)
VW  stk[stksz / sizeof(VW)];
static const VP  p_stk=(VP)&stk[stksz/sizeof(VW)];                         (2)
#pragma interrupt(DirectInh(sp=p_stk,tn=62,resbank))                       (3)
void DirectInh(void)                                                       (4)
{
   /* ハンドラの処理 */
}
```

図12.6　ダイレクト割込みハンドラの記述例

■図の解説

(1) ハンドラで使用するスタックを確保します。これは、割り込まれる側のスタックのオーバフローを避けるためです。同じ割込みレベルのハンドラは、スタックを共有できます。

(2) スタックポインタの初期値をconst型として定義します。

(3) #pragma interruptにより、ハンドラを割込み関数として宣言します。割込み関数仕様として、以下を指定してください。

　(a) "sp="指定(スタック切り替え)
　NMI以外の割込みは、必ずスタックを切り替えなければなりません。スタックを切り替える場合は、(2)の変数を指定してください。

　(b) "tn="指定(トラップリターン指定)
　詳細は、表12.5を参照してください。

　(c) "resbank"指定(バンク割込み)
　レジスタバンクを使用する割込みの場合は指定が必要です。詳細は、表12.5を参照してください。

(4) 割込み関数は、引数もリターン値も持たない関数として記述します。

表12.5　"tn="指定と"resbank"指定

レジスタバンク	割込みレベル	"tn="指定	"resbank"指定
使用しない *	カーネル割込みマスクレベル(system.system_IPL)より高い	指定禁止	指定禁止
	カーネル割込みマスクレベル(system.system_IPL)以下	"tn=63"	
使用する *	カーネル割込みマスクレベル(system.system_IPL)より高い	指定禁止	指定必須
	カーネル割込みマスクレベル(system.system_IPL)以下	"tn=62"	

【注】 表 12.3参照

(2) レジスタ使用規約

表 12.6に、ダイレクト割込みハンドラのレジスタ使用規約を示します。デバッグやアセンブリ言語でダイレクト割込みハンドラを作成する場合の参考にしてください。

表12.6 ダイレクト割込みハンドラのレジスタ使用規約

No	レジスタ	保証 *1	初期値
1	PC	不要	ダイレクト割込みハンドラのアドレス
2	SR	*2	IMASK：割込みレベル。ハンドラ実行中は、IMASK を自割込みレベルよりも下げてはなりません。 その他のビット：割込み発生前と同じ
3	R0～R14, MACH, MACL, GBR	必須 *3	不定
4	R15	必須	割込まれたプログラムのスタックを指しています。 [NMI 以外の場合] 割込み前に実行していたプログラムのスタックオーバフローを防ぐため、必ず割込み前のスタックからハンドラ専用のスタックに切り替えてください。これを怠ると、ハンドラは割込み前に実行していたタスクのスタックを使うことになるため、そのタスクのスタックがオーバフローする可能性があります。 同じ割込みレベルのハンドラは同時に実行することは無いため、スタックを共有できます。この場合、最も多くスタックを使用するハンドラのサイズを確保してください。なお、割込みハンドラでは割込み前のスタックを 4 バイトだけ使用することが許されます。 [NMI の場合] NMI がネストする可能性がある場合は、スタックを切り換えてはなりません。この場合、NMI 割込みハンドラは、NMI 発生時のスタックを使用することになるので、NMI 割込みハンドラが使用するサイズを、タスクや割込みハンドラなどのスタックに加算する必要があります。
5	PR	必須 *3	不定
6	TBR	*4	*4
7	[SH2A-FPU] FPSCR, FPUL, FR0～FR15	必須 *5	不定

【注】 *1 ダイレクト割込みハンドラ開始関数が終了(RTE または TRAPA 命令)する時点で、レジスタの値が起動時と同一であることを要求するかどうか。

*2 IMASK ビットのみ保証必須です。

*3 レジスタバンクを使用する割込み(表 12.3参照)の場合のみ、保証不要です。

*4 system.tbr の設定に依存します。

(1) system.tbr=NOMANAGE の場合：カーネルは TBR を一切操作しません。

(2) system.tbr=FOR_SVC の場合：変更禁止です。

(3) system.tbr=TASK_CONTEXT の場合：保証必須です。初期値は不定です。

*5 ハンドラで FPU を使用する場合は、「20.3 各種ハンドラ等で浮動小数点演算を行う場合」を参照してください。

12.6　CPU 例外ハンドラ(TRAPA 例外を含む)

12.6.1　CPU 例外ハンドラの種類

CPU 例外ハンドラには、ノーマル CPU 例外ハンドラとダイレクト CPU 例外ハンドラがあります。

ノーマル CPU 例外ハンドラは、CPU 例外時にカーネルを介して起動され、通常の C 言語関数として記述することができます。また、ハンドラには例外発生時の情報がパラメータとして渡されます。

ダイレクト CPU 例外ハンドラは、CPU の割込み例外ベクタテーブルに登録されます。ダイレクト CPU 例外ハンドラは、割込み関数(#pragma interupt ディレクティブを使用)として記述する必要があります。また、ダイレクト CPU 例外ハンドラには、何もパラメータは渡りません。

12.6.2　ノーマル CPU 例外ハンドラ

(1) 記述方法

ノーマル CPU 例外ハンドラは、図 12.7のように通常の C 言語関数として記述します。

```
#include "kernel.h"
void Exc(UW excno, VT_EXC *pk_exc)     ←ハンドラは、引数もリターン値も持たない関数として記
{                                        述します。excno には、CPU 例外番号、pk_exc には
    /*ハンドラの処理 */                  CPU 例外情報が渡されます。
}
```

図12.7　ノーマル CPU 例外ハンドラの記述例

VT_EXC 構造体の仕様は、以下の通りです。

```
typedef struct {
    UW  r0;    /* CPU 例外発生時の R0 */
    UW  r1;    /* CPU 例外発生時の R1 */
    UW  r2;    /* CPU 例外発生時の R2 */
    UW  r3;    /* CPU 例外発生時の R3 */
    UW  r4;    /* CPU 例外発生時の R4 */
    UW  r5;    /* CPU 例外発生時の R5 */
    UW  r6;    /* CPU 例外発生時の R6 */
    UW  r7;    /* CPU 例外発生時の R7 */
    UW  pr;    /* CPU 例外発生時の PR */
    UW  pc;    /* CPU 例外発生時の PC */
    UW  sr;    /* CPU 例外発生時の SR */
} VT_EXC;
```

なお、ノーマル CPU 例外ハンドラ起動時の R8～R14, GBR, MACH, MACL は、CPU 例外発生時と同じ値になっています。また、CPU 例外発生時の R15 は、pk_exc + sizeof(VT_EXC)で求めることができます。

(2) レジスタ使用規約

表 12.7に、ノーマル CPU 例外ハンドラのレジスタ使用規約を示します。デバッグやアセンブリ言語でノーマル CPU 例外ハンドラを作成する場合の参考にしてください。

表12.7 ノーマル CPU 例外ハンドラのレジスタ使用規約

No	レジスタ	保証 *1	初期値
1	PC	不要	ノーマル CPU 例外ハンドラのアドレス
2	SR	*2	CPU 例外発生前と同じ
3	R0～R3, R6～R7	不要	不定
4	R4	不要	excno(発生した CPU 例外のベクタ番号)
5	R5	不要	pk_exc
6	R8～R14, MACH, MACL, GBR	必須	CPU 例外発生時と同じ
7	R15	必須	例外発生元のプログラムのスタックを指しています。CPU 例外ハンドラは再入の可能性があるため、例外を発生させたプログラムのスタックを使用して動作します。CPU 例外ハンドラ専用のスタックを持つことはできません
8	PR	必須	不定
9	TBR	*3	*3
10	[SH2A-FPU] FPSCR, FPUL, FR12～FR15	必須 *4	CPU 例外発生時と同じ

【注】 *1 ノーマル CPU 例外ハンドラ開始関数からリターンする時点で、レジスタの値が起動時と同一であることを要求するかどうか。

*2 IMASK ビットのみ保証必須です。

*3 system.tbr の設定に依存します。

(1) system.tbr=NOMANAGE の場合：カーネルは TBR を一切操作しません。

(2) system.tbr=FOR_SVC の場合：変更禁止です。

(3) system.tbr=TASK_CONTEXT の場合：保証必須です。初期値は不定です。

*4 ハンドラで FPU を使用する場合は、「20.3 各種ハンドラ等で浮動小数点演算を行う場合」を参照してください。

12.6.3 ダイレクト CPU 例外ハンドラ

(1) 記述方法

ダイレクト CPU 例外は、図 12.8に示すように割込み関数として記述します。

```
#include "kernel.h"
#pragma interrupt(DirectExc)                                    (1)
void DirectExc(void)                                            (2)
{
    /*ハンドラの処理 */
}
```

図12.8 ダイレクト CPU 例外ハンドラの記述例

■図の解説

(1) #pragma interruptにより、ハンドラを割込み関数として宣言します。割込み関数仕様には何も指定しないでください。

(2) 割込み関数は、引数もリターン値も持たない関数として記述します。

(2) レジスタ使用規約

表 12.8に、ダイレクト CPU 例外ハンドラのレジスタ使用規約を示します。デバッグやアセンブリ言語でダイレクト CPU 例外ハンドラを作成する場合の参考にしてください。

表12.8 ダイレクト CPU 例外ハンドラのレジスタ使用規約

No	レジスタ	保証 *1	初期値
1	PC	不要	ダイレクト CPU 例外ハンドラのアドレス
2	SR	*2	CPU 例外発生前と同じ。ハンドラ実行中は、IMASK を起動時よりも下げてはなりません。
3	R0～R14, MACH, MACL, GBR	必須	CPU 例外発生前と同じ。
4	R15	必須	例外発生元のプログラムのスタックを指しています。CPU 例外ハンドラは再入の可能性があるため、例外を発生させたプログラムのスタックを使用して動作します。CPU 例外ハンドラ専用のスタックを持つことはできません。 ← 4 バイト → R15→ CPU 例外発生時点の PC CPU 例外発生時点の SR CPU 例外発生前の R15→
5	PR	必須	不定
7	TBR	*3	*3
8	[SH2A-FPU] FPSCR, FPUL, FR0～FR15	必須 *4	不定

【注】 *1 ダイレクト CPU 例外ハンドラ開始関数が終了(RTE または TRAPA 命令)する時点で、レジスタの値が起動時と同一であることを要求するかどうか。

*2 IMASK ビットのみ保証必須です。

*3 system.tbr の設定に依存します。

(1) system.tbr=NOMANAGE の場合：カーネルは TBR を一切操作しません。
(2) system.tbr=FOR_SVC の場合：変更禁止です。
(3) system.tbr=TASK_CONTEXT の場合：保証必須です。初期値は不定です。

*4 ハンドラで FPU を使用する場合は、「20.3 各種ハンドラ等で浮動小数点演算を行う場合」を参照してください。

12.7 タイムイベントハンドラ

(1) 記述方法

タイムイベントハンドラは、通常の C 言語関数として記述します。図 12.9に周期ハンドラ、アラームハンドラ、図 12.10にオーバーランハンドラの C 言語記述例を示します。これらのハンドラは非タスクコンテキストで動作します。

```
#include "kernel.h"
void Handler(VP_INT exinf)    ←パラメータとして生成時に指定した exinf が渡されます。
{
    /* ハンドラの処理 */
}
```

図12.9 周期ハンドラ、アラームハンドラの C 言語記述例

```
#include "kernel.h"
void Ovrhdr(ID tskid, VP_INT exinf)   ←パラメータとして起動の原因となった tskid と、そのタ
{                                      スクの exinf が渡されます。
    /* ハンドラの処理 */
}
```

図12.10 オーバーランハンドラの C 言語記述例

(2) レジスタ使用規約

表 12.9に、タイムイベントハンドラのレジスタ使用規約を示します。デバッグやアセンブリ言語でこれらのプログラムを作成する場合の参考にしてください。

表12.9 タイムイベントハンドラのレジスタ使用規約

No	レジスタ	保証 *1	初期値
1	PC	不要	ハンドラのアドレス
2	SR	*2	(1)IMASK タイマ割込みによって起動された場合： タイマ割込みレベル(clock.IPL) vrst_tmr によって起動された場合： タイマ割込みレベル(clock.IPL)と vrst_tmr 呼び出し時点の IMASK の大きい方の値 ハンドラ実行中は、IMASK を起動時よりも下げてはなりません。 (2)その他のビット：不定
3	R0～R3	不要	不定
4	R4	不要	[周期ハンドラ、アラームハンドラ] ハンドラの拡張情報 [オーバーランハンドラ] 対象のタスク ID
5	R5	不要	[周期ハンドラ、アラームハンドラ]不定 [オーバーランハンドラ] 対象タスクの拡張情報
6	R6～R7	不要	不定
7	R8～R14, MACH, MACL, GBR	必須	不定
8	R15	必須	タイマスタック内を指しています。
9	PR	必須	不定
11	TBR	*3	*3
12	[SH2A-FPU] FPSCR, FPUL, FR12～FR15	必須 *4	不定

【注】 *1 ハンドラ開始関数からリターンする時点で、レジスタの値が起動時と同一であることを要求するかどうか。

*2 IMASK ビットのみ保証必須です。

*3 system.tbr の設定に依存します。

(1) system.tbr=NOMANAGE の場合：カーネルは TBR を一切操作しません。

(2) system.tbr=FOR_SVC の場合：変更禁止です。

(3) system.tbr=TASK_CONTEXT の場合：保証必須です。初期値は不定です。

*4 ハンドラで FPU を使用する場合は、「20.3 各種ハンドラ等で浮動小数点演算を行う場合」を参照してください。

12.8　初期化ルーチン

(1) 記述方法

初期化ルーチンは、通常の C 言語関数として記述します。図 12.11に初期化ルーチンの C 言語記述例を示します。初期化ルーチンは非タスクコンテキストで動作します。

```
#include "kernel.h"
void InitRoutine(VP_INT exinf)   ←パラメータとして定義時に指定した exinf が渡されます。
{
    /* ハンドラの処理 */
}
```

図12.11　初期化ルーチンの C 言語記述例

(2) レジスタ使用規約

表 12.10に、初期化ルーチンのレジスタ使用規約を示します。デバッグやアセンブリ言語でこれらのプログラムを作成する場合の参考にしてください。

表12.10　初期化ルーチンのレジスタ使用規約

No	レジスタ	保証 *1	初期値
1	PC	不要	ルーチンのアドレス
2	SR	*2	(1)IMASK カーネル割込みマスクレベル (system.system_IPL) ルーチン実行中は、IMASK を起動時よりも下げてはなりません (2)その他のビット：不定
3	R0～R3	不要	不定
4	R4	不要	ルーチンの拡張情報
5	R5～R7	不要	不定
6	R8～R14, MACH, MACL, GBR	必須	不定
7	R15	必須	カーネルスタック内を指しています。
8	PR	必須	不定
9	TBR	*3	*3
10	[SH2A-FPU] FPSCR, FPUL, FR12～FR15	必須 *4	不定

【注】 *1　ハンドラ・ルーチン開始関数からリターンする時点で、レジスタの値が起動時と同一であることを要求するかどうか。

*2　IMASK ビットのみ保証必須です。

*3　system.tbr の設定に依存します。

(1) system.tbr=NOMANAGE の場合：カーネルは TBR を一切操作しません。

(2) system.tbr=FOR_SVC の場合：変更禁止です。

(3) system.tbr=TASK_CONTEXT の場合：保証必須です。初期値は不定です。

*4　ルーチンで FPU を使用する場合は、「20.3 各種ハンドラ等で浮動小数点演算を行う場合」を参照してください。

12.9　タイマドライバ

cfg ファイルで clock.timer に TIMER を指定した場合は、タイマドライバを作成し、カーネルとリンクする必要があります。
なお、両方の CPU について、それぞれタイマドライバが必要です。

タイマドライバは、以下の関数から構成されます。以降の解説およびサンプル提供のタイマドライバファイルを参考に、タイマドライバを実装してください。

- tdr_ini_tmr()：タイマの初期化
- tdr_int_tmr()：タイマ割込み処理
- tdr_stp_tmr()：タイマの停止
- tdr_rst_tmr()：タイマの再開

12.9.1 tdr_ini_tmr()：タイマの初期化

■関数仕様

```
void tdr_ini_tmr(void);
```

■引数

なし

■リターン値

なし

■機能説明

タイマを初期化します。

初期化にあたっては、cfg72mp が kernel_macro.h に出力する以下のマクロを利用してください。特に、タイマ割込みの周期時間は、TIC_NUME/TIC_DENO [ミリ秒]となるようにしてください。なお、kernel_macro.h は、kernel.h からインクルードされています。

- TIC_NUME：タイマ割込み周期(ミリ秒)の分子
- TIC_DENO：タイマ割込み周期(ミリ秒)の分母
- TIM_LVL：タイマ割込みレベル

clock.timer に TIMER を指定した場合は、cfg72mp は以下を自動的に行います。

(1) tdr_ini_tmr()を初期化ルーチンとして登録します。
(2) clock.numberで指定された割込み番号に対して、カーネルライブラリ内の処理モジュールをダイレクト割込みハンドラとして定義します。なお、tdr_int_tmr()は、このカーネルモジュールから呼び出されます。

表 12.11に、tdr_ini_tmr()のレジスタ使用規約を示します。デバッグやアセンブリ言語で tdr_ini_tmr()を作成する場合の参考にしてください。

表12.11　tdr_ini_tmr()のレジスタ使用規約

No	レジスタ	保証 *1	初期値
1	PC	不要	tdr_ini_tmr()
2	SR	*2	IMASK：カーネル割込みマスクレベル (system.system_IPL)。tdr_ini_tmr()実行中は、IMASK を自割込みレベルよりも下げてはなりません。 その他のビット：不定
3	R0～R7	不要	不定
4	R8～R14, MACH, MACL, GBR	必須	不定
5	R15	必須	カーネルスタック内を指しています。
6	PR	必須	不定
7	TBR	*3	*3
8	[SH2A-FPU] FPSCR, FPUL, FR12～FR15	必須 *4	不定

【注】 *1 tdr_ini_tmr()からリターンする時点で、レジスタの値が起動時と同一であることを要求するかどうか。

*2 IMASK ビットのみ保証必須です。

*3 system.tbr の設定に依存します。

(1) system.tbr=NOMANAGE の場合：カーネルは TBR を一切操作しません。

(2) system.tbr=FOR_SVC の場合：変更禁止です。

(3) system.tbr=TASK_CONTEXT の場合：保証必須です。初期値は不定です。

*4 tdr_ini_tmr()で FPU を使用する場合は、「20.3 各種ハンドラ等で浮動小数点演算を行う場合」を参照してください。

12.9.2 tdr_int_tmr()：タイマ割込み処理

■関数仕様

```
void tdr_int_tmr(void);
```

■引数

なし

■リターン値

なし

■機能説明

タイマ割込み要因をクリアします。

本関数は、カーネル内のタイマ割込み用ダイレクト割込みハンドラから呼び出されます。すなわち、本関数は非タスクコンテキストで実行されます。本関数では、非タスクコンテキスト用のサービスコールを呼び出すことができます。

表 12.12に、tdr_int_tmr()のレジスタ使用規約を示します。デバッグやアセンブリ言語で tdr_int_tmr()を作成する場合の参考にしてください。

表12.12 tdr_int_tmr()のレジスタ使用規約

No	レジスタ	保証 *1	初期値
1	PC	不要	tdr_int_tmr()
2	SR	*2	IMASK：タイマ割込みレベル(clock.IPL)。tdr_int_tmr()実行中は、IMASK を起動時よりも下げてはなりません。 その他のビット：不定
3	R0～R7	不要	不定
4	R8～R14, MACH, MACL, GBR	必須	不定
5	R15	必須	タイマスタック内を指しています。
6	PR	必須	不定
7	TBR	*3	*3
8	[SH2A-FPU] FPSCR, FPUL, FR12～FR15	必須 *4	不定

【注】 *1 tdr_int_tmr()からリターンする時点で、レジスタの値が起動時と同一であることを要求するかどうか。

*2 IMASK ビットのみ保証必須です。

*3 system.tbr の設定に依存します。

(1) system.tbr=NOMANAGE の場合：カーネルは TBR を一切操作しません。

(2) system.tbr=FOR_SVC の場合：変更禁止です。

(3) system.tbr=TASK_CONTEXT の場合：保証必須です。初期値は不定です。

*4 tdr_int_tmr()で FPU を使用する場合は、「20.3 各種ハンドラ等で浮動小数点演算を行う場合」を参照してください。

12.9.3 tdr_stp_tmr()：タイマの停止

■関数仕様

```
void tdr_stp_tmr(void);
```

■引数

なし

■リターン値

なし

■機能説明

タイマを停止します。すなわち、タイマ割込みが発生しないようにします。

本関数は、カーネルの vstp_tmr サービスコールからコールバックされ、vstp_tmr サービスコール処理の一部として実行されます。本関数では、非タスクコンテキスト用のサービスコールを呼び出すことができます。

cfg ファイルで vstp_tmr サービスコールを選択しない場合は、本関数を実装する必要はありません。

表 12.13に、tdr_stp_tmr()のレジスタ使用規約を示します。デバッグやアセンブリ言語で tdr_stp_tmr()を作成する場合の参考にしてください。

表12.13 tdr_stp_tmr()のレジスタ使用規約

No	レジスタ	保証 *1	初期値
1	PC	不要	tdr_stp_tmr()
2	SR	*2	IMASK：タイマ割込みレベル(clock.IPL)。tdr_stp_tmr()実行中は、IMASK を起動時よりも下げてはなりません。 その他のビット：不定
3	R0～R7	不要	不定
4	R8～R14, MACH, MACL, GBR	必須	不定
5	R15	必須	タイマスタック内を指しています。
6	PR	必須	不定
7	TBR	*3	*3
8	[SH2A-FPU] FPSCR, FPUL, FR12～FR15	必須 *4	不定

【注】 *1 tdr_stp_tmr()からリターンする時点で、レジスタの値が起動時と同一であることを要求するかどうか。

*2 IMASK ビットのみ保証必須です。

*3 system.tbr の設定に依存します。

(1) system.tbr=NOMANAGE の場合：カーネルは TBR を一切操作しません。

(2) system.tbr=FOR_SVC の場合：変更禁止です。

(3) system.tbr=TASK_CONTEXT の場合：保証必須です。初期値は不定です。

*4 tdr_stp_tmr()で FPU を使用する場合は、「20.3 各種ハンドラ等で浮動小数点演算を行う場合」を参照してください。

RENESAS

12.9.4 tdr_rst_tmr()：タイマの再開

■関数仕様

```
void tdr_rst_tmr(void);
```

■引数

なし

■リターン値

なし

■機能説明

タイマを再開します。すなわち、タイマ割込みが発生するようにします。

本関数は、カーネルの vrst_tmr または ivrst_tmr サービスコールからコールバックされ、vrst_tmr または ivrst_tmr サービスコール処理の一部として実行されます。本関数では、非タスクコンテキスト用のサービスコールを呼び出すことができます。

cfg ファイルで vrst_tmr, ivrst_tmr サービスコールを選択しない場合は、本関数を実装する必要はありません。

表 12.14に、tdr_rst_tmr()のレジスタ使用規約を示します。デバッグやアセンブリ言語で tdr_rst_tmr()を作成する場合の参考にしてください。

表12.14　tdr_rst_tmr()のレジスタ使用規約

No	レジスタ	保証 *1	初期値
1	PC	不要	tdr_rst_tmr()
2	SR	*2	IMASK：以下の中の最大値 (1)タイマ割込みレベル(clock.IPL) (2)vrst_tmr, ivrst_tmr 時点の IMASK tdr_rst_tmr()実行中は、IMASK を起動時よりも下げてはなりません。 その他のビット：不定
3	R0～R7	不要	不定
4	R8～R14, MACH, MACL, GBR	必須	不定
5	R15	必須	タイマスタック内を指しています。
6	PR	必須	不定
7	TBR	*3	*3
8	[SH2A-FPU] FPSCR, FPUL, FR12～FR15	必須	不定

【注】 *1 tdr_rst_tmr()からリターンする時点で、レジスタの値が起動時と同一であることを要求するかどうか。

*2 IMASK ビットのみ保証必須です。

*3 system.tbr の設定に依存します。

(1) system.tbr=NOMANAGE の場合：カーネルは TBR を一切操作しません。

(2) system.tbr=FOR_SVC の場合：変更禁止です。

(3) system.tbr=TASK_CONTEXT の場合：保証必須です。初期値は不定です。

*4 tdr_rst_tmr()で FPU を使用する場合は、「20.3 各種ハンドラ等で浮動小数点演算を行う場合」を参照してください。

12.10 システムダウンルーチン

システムダウンルーチンは、以下の C 言語関数として作成します。この名前は固定です。

```
void   _kernel_sysdwn(W type, VW inf1, VW inf2, VW inf3)
```

システムダウンルーチンは、必ず作成してカーネルとリンクしなければなりません。

表 12.15に、システムダウンルーチンに渡されるパラメータの仕様を示します。

システムダウンルーチンでは異常内容に応じた処理を行うことができますが、カーネル内部でシステムダウンとなった場合（エラー種別が負）は、サービスコールなどカーネルの機能は使用できません。

また、システムダウンルーチンからリターンしてはなりません。

デバッグ時には、システムダウンではその時の状態を保持して無限ループさせ、システムダウン要因の解析、対策を行ってください。

表12.15　システムダウンルーチンに渡されるパラメータ

項番	システムダウン要因	エラー種別 W type (R4)	システムダウン情報 1 VW inf1(R5)	システムダウン情報 2 VW inf2 (R6)	システムダウン情報 3 VW inf3 (R7)
1	vsys_dwn, ivsys_dwn サービスコール	1～H'7fffffff	vsys_dwn, ivsys_dwn サービスコールのパラメータ		
2	cfg ファイルが生成した初期登録情報パケットの内容が不正(データ破壊など)	0	0	不正データを検出したアドレス	不定
3	cfg ファイルでの初期登録情報に誤りがあった場合		エラーコード(負) *1	オブジェクト種別 *1	オブジェクト番号 *1
4	非タスクコンテキストからの ext_tsk サービスコールでコンテキストエラーが発生	H'ffffffff(-1)	E_CTX (H'ffffffe7)	ext_tsk を呼び出したアドレス	不定
5	非タスクコンテキストからの exd_tsk サービスコールでコンテキストエラーが発生	H'fffffffe(-2)	E_CTX (H'ffffffe7)	exd_tsk を呼び出したアドレス	不定
6	タスクコンテキストから呼び出されたサービスコールで、スタックオーバフローを検出	H'fffffff8(-8)	タスク ID	タスクのスタック領域の先頭アドレス	オーバフロー検出時点のスタックポインタ値
7	未定義割込みが発生	H'fffffff0(-16)	ベクタ番号	不定	不定
8	未定義 CPU 例外が発生			例外発生時の PC	VT_EXC *pk_exc *2

【注】 *1　エラーコード、オブジェクト種別、オブジェクト番号には、初期登録に失敗したオブジェクトについて、表 12.16に示す情報が設定されます。

*2　system.vector_type が ROM または RAM の場合のみ設定されます。

RENESAS

表12.16　初期登録失敗時のオブジェクト種別とオブジェクト番号

項番	オブジェクト種別	エラーコード (inf1) *1	オブジェクト 種別(inf2)	オブジェクト番号 (inf3)
1	割込み・CPU 例外ハンドラ	def_inh	0	ベクタ番号
2	タスク	cre_tsk	1	ローカルタスク ID
3	タスク例外処理ルーチン	def_tex	2	ローカルタスク ID
4	セマフォ	cre_sem	3	ローカルセマフォ ID
5	イベントフラグ	cre_flg	4	ローカルイベントフラグ ID
6	データキュー	cre_dtq	5	ローカルデータキューID
7	メールボックス	cre_mbx	6	ローカルメールボックス ID
8	ミューテックス	cre_mtx	7	ローカルミューテックス ID
9	メッセージバッファ	cre_mbf	8	ローカルメッセージバッファ ID
10	固定長メモリプール	cre_pf	9	ローカル固定長メモリプール ID
11	可変長メモリプール	cre_mpl	10	ローカル可変長メモリプール ID
12	周期ハンドラ	cre_cyc	11	ローカル周期ハンドラ ID
13	アラームハンドラ	cre_alm	12	ローカルアラームハンドラ ID
14	オーバーランハンドラ	def_ovr	13	不定
15	拡張サービスコールルーチン	def_svc	14	機能コード

【注】 *1　ここに記載したサービスコールで返るエラーコードの意味になります。

13. ロードモジュール作成手順概要

13.1 概要

HI7200/MP では、各 CPU 毎に独立にロードモジュールを生成します。
各ロードモジュールは、一般に以下に示す手順で開発します。

(1) High-performance Embedded Workshop ワークスペース･プロジェクトの作成

High-performance Embedded Workshop を使用する場合は、サンプル提供の High-performance Embedded Workshop ワークスペース・プロジェクトをコピーし、雛形として使用してください。

(2) アプリケーションプログラムのコーディング

アプリケーションプログラムをコーディングします。提供する各種サンプルプログラムも参考にしてください。

(3) cfg ファイルの作成

タスクのエントリアドレスやスタックサイズなどを定義した cfg ファイル(拡張子.cfg)をテキストエディタなどを用いて作成します。提供するサンプル cfg ファイルも参考にしてください。
なお、GUI コンフィギュレータを用いて cfg ファイルを生成することもできます。

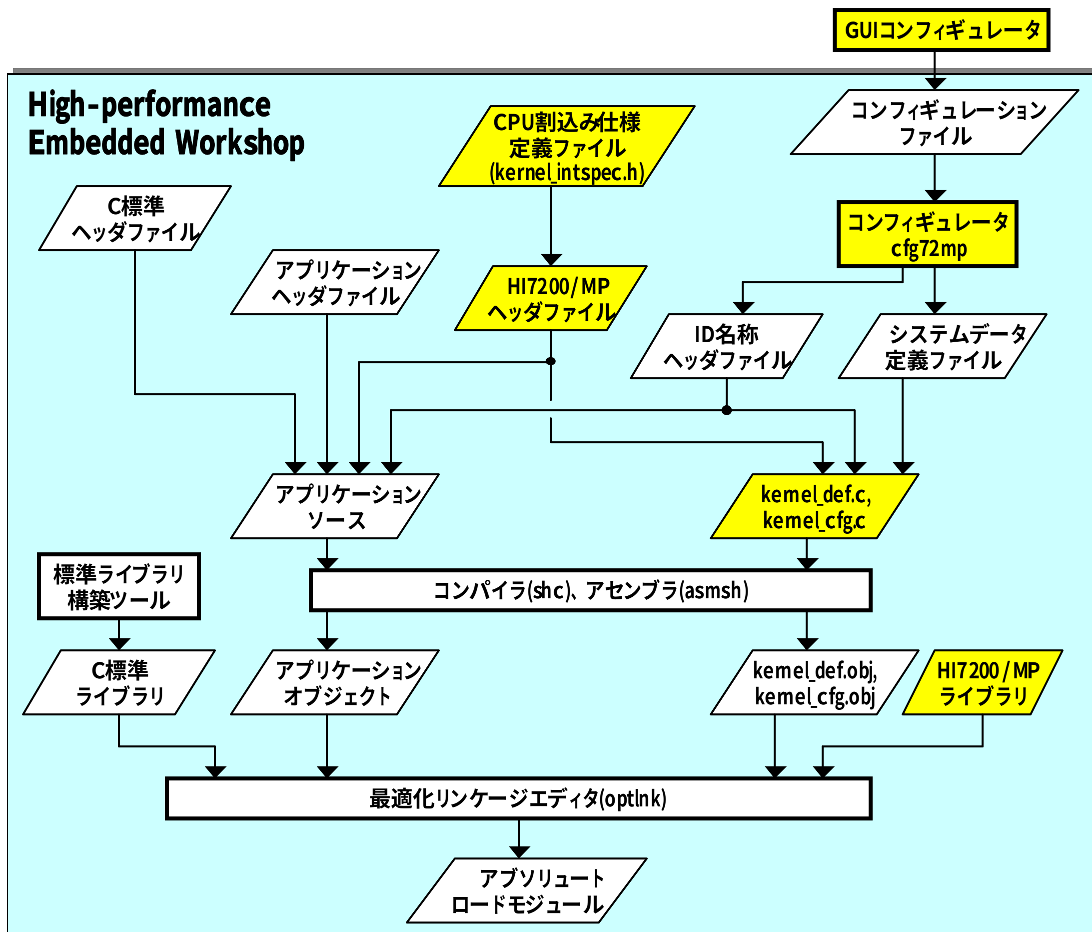
(4) コンフィギュレータ実行

コンフィギュレータ cfg72mp により、構築ファイルやヘッダファイルなどを生成します。
High-performance Embedded Workshop のビルドフェーズに、cfg72mp をカスタムビルドフェーズとして登録することで、コンフィギュレータ実行とビルドをワンアクションで行うこともできます。

(5) ビルドによるロードモジュールの生成

High-performance Embedded Workshop を用いてビルドを行い、ロードモジュールを生成します。

図 13.1にロードモジュール生成フローを示します。



黄色は、HI7200/MP で提供されるものです。

図13.1 ロードモジュール生成フロー

14. コンフィギュレータ(cfg72mp)

14.1 cfg ファイル内の表現形式

この節では cfg ファイル内における定義データの表現形式について説明します。

14.1.1 コメント文

'//'から行の終わりまではコメント文とみなし、処理の対象になりません。

14.1.2 文の終わり

';'で文を終ります。

14.1.3 定義文

cfg ファイルでは、以下のいずれかの書式で定義文を記述します。

```
書式１:    [定義名]  {
               [定義項目名] = [設定値] ;
               [定義項目名] = [設定値] ;
                    …
           };

書式２ :   [定義名][ [番号] ] {
               [定義項目名] = [設定値] ;
               [定義項目名] = [設定値] ;
                    …
           };
```

「定義名」および「定義項目名」は cfg ファイル仕様として定められた文字列であり、総称して「キーワード」と呼びます。キーワードの書式は、後述の「シンボル」と同じです。

「設定値」はユーザが設定する項目で、後述の「数値」「シンボル」「外部参照名」のいずれかの書式で設定します。どの書式を記述できるかは、キーワードによって異なります。また、設定値を複数指定できるキーワードもあります。

「番号」は、タスクなどの複数の定義を行う場合に、各定義の区別を行うための番号です。「定義名」によって、"[番号]"を持つものと持たないものがあります。

番号は、「数値」の書式で設定します。番号は、定義名によって、「タスク ID」、「セマフォ ID」、「割込みベクタ番号」などの意味になります。また、定義名によっては、番号を省略できる場合があります。

14.1.4 数値

数値は以下の形式で入力できます。

- 16 進数
  数値の先頭に'0x'か'0X'を付加します。または、数値の最後に'h'か'H'を付加します。後者の場合でかつ先頭が英文字（A～F)で始まる場合は先頭に必ず'0'を付加してください。なおここで使用する数値表現で英文字（A～F)は大文字・小文字を識別しません。[2]
- 10 進数
  23 のように整数のみで表します。ただし'0'で始めることはできません。
- 8 進数
  数値の先頭に'0'を付加するか数値の最後に'O'もしくは'o'を付加します。
- 2 進数
  数値の最後に'B'または'b'を付加します。ただし'0'で始めることはできません。

表14.1 数値表現例

16 進数	0xf12 0Xf12 0a12h 0a12H 12h 12H
10 進数	32
8 進数	017 17o 17O
2 進数	101110b 101010B

また数値内に演算子を記述できます。使用できる演算子を表 14.2に示します。

表14.2 演算子

演算子	優先度	演算方向
()	高	左から右
－(単項マイナス)		右から左
* / %		左から右
+ －(二項マイナス)	低	左から右

数値の例を以下に示します。

- 123
- 123 + 0x23
- (23/4 + 3) * 2
- 100B + 0aH

0xFFFFFFFF を超える数値を指定してはなりません。

[2] 数値表現内の’A’～’F’,’a’～’f’を除いて全ての文字は、大文字・小文字の区別を行います。

14.1.5 シンボル

シンボルは数字、英大文字、英小文字、'_'（アンダースコア）、'?'より構成される数字以外の文字で始まる文字列で表されます。

シンボルの例を以下に示します。

- _TASK1
- IDLE3

シンボルは、具体的にはオブジェクトの ID 名称およびセクション名の指定に使います。

14.1.6 外部参照名

外部参照名は、C 言語の外部参照シンボル名であり、数字、英大文字、英小文字、'_'(アンダースコア)、'$'(ドル記号)より構成される数字以外の文字で始まり、"()"で終わる文字列で表されます。

外部参照名は、cfg ファイルで外部の関数や変数のアドレスを参照したい場合に使います。

表 14.3に外部参照名の指定例を示します。

表14.3 外部参照名の指定例

参照したい外部定義シンボル	外部参照名の記述方法
C 言語の main()関数	main()
C 言語の int data	(アドレスでないので記述できません)
C 言語の int data のアドレス	data()
C 言語の int *pointer のアドレス	pointer()
C 言語の int array[]の配列のアドレス	array()
アセンブリ言語のラベル_LABEL1	LABEL1()
アセンブリ言語のラベル LABEL2	(参照できません)

14.1.7 注意

コンフィギュレータは、cfg ファイルで指定する ID 名称、セクション名、外部参照名に関する重複などのエラーを検出しません。多くの場合、コンフィギュレータの出力ファイルがコンパイルされるときにエラーが報告されることになります。

14.2 デフォルト cfg ファイル

多くの定義項目では、ユーザが記述を省略した場合にデフォルト cfg ファイルの内容が補われます。デフォルト cfg ファイルは、<RTOS_INST>¥cfg72mp¥default.cfg です。なお、このファイルを編集してはなりません。

14.3　cfg ファイルの定義項目

cfg ファイルでは以下の項目の定義をおこないます。

- システム定義(system)
- 最大 ID 定義(maxdefine)
- デフォルトタスクスタック用領域定義(memstk)
- デフォルトデータキュー用領域定義(memdtq)
- デフォルトメッセージバッファ用領域定義(memmbf)
- デフォルト固定長メモリプール用領域定義(memmpf)
- デフォルト可変長メモリプール用領域定義(memmpl)
- システムクロック定義(clock)
- リモートサービスコール環境定義(remote_svc)
- タスク定義(task[])
- スタティックスタック領域定義 (static_stack[])
- セマフォ定義(semaphore[])
- イベントフラグ定義(flag[])
- データキュー定義(dataqueue[])
- メールボックス定義(mailbox[])
- ミューテックス定義(mutex[])
- メッセージバッファ定義(message_buffer[])
- 固定長メモリプール定義(memorypool[])
- 可変長メモリプール定義(variable_memorypool[])
- 周期ハンドラ定義(cyclic_hand[])
- アラームハンドラ定義(alarm_hand[])
- オーバーランハンドラ定義(overrun_hand)
- 拡張サービスコールルーチン定義(extend_svc[])
- 割込みハンドラ、CPU 例外ハンドラ定義(interrupt_vector[])
- 初期化ルーチン定義(init_routine[])
- サービスコール定義(service_call)

14.3.1　説明形式

【説明】定義項目に関する説明です。
【定義形式】指定可能な定義形式を記述します。
【定義範囲】指定可能な範囲を記述します。
【省略時の扱い】定義項目を記述しなかった場合の扱いを記述します。
【GUIコンフィギュレータの該当設定項目】GUIコンフィギュレータの該当設定項目名を記述します。
【備考】特殊なケースの仕様などを記述します。

14.3.2 システム定義(system)

ここでは、カーネルシステム全般に関連する情報を定義します。system は省略できません。

■形式

```
system {
        cpuid               = <設定値>; // (1)CPUID
        stack_size          = <設定値>; // (2)割込みスタックサイズ
        kernel_stack_size   = <設定値>; // (3)カーネルスタックサイズ
        priority            = <設定値>; // (4)最大タスク優先度
        system_IPL          = <設定値>; // (5)カーネル割込みマスクレベル
        message_pri         = <設定値>; // (6)最大メッセージ優先度数
        tic_deno            = <設定値>; // (7)タイムティック分母
        tic_nume            = <設定値>; // (8)タイムティック分子
        tbr                 = <設定値>; // (9)TBR レジスタの利用形態
        parameter_check     = <設定値>; // (10)サービスコールのパラメータチェック
        mpfmanage           = <設定値>; // (11)固定長メモリプール管理方式
        newmpl              = <設定値>; // (12)可変長メモリプール管理方式
        trace               = <設定値>; // (13)サービスコールトレース機能
        trace_buffer        = <設定値>; // (14)サービスコールトレース機能のバッファサイズ
        trace_object        = <設定値>; // (15)サービスコールトレース機能のオブジェクト取得数
        action              = <設定値>; // (16)オブジェクト操作機能
        vector_type         = <設定値>; // (17)割込みベクタのタイプ
        regbank             = <設定値>; // (18)レジスタバンクの使用方法
};
```

■内容

(1) CPUID (cpuid)

【説明】このカーネルを実行させるCPUIDを定義します。
【定義形式】数値
【定義範囲】1または2
【省略時の扱い】省略不可(エラー)
【GUIコンフィギュレータの該当設定項目】CFG_MYCPUID

(2) 割込みスタックサイズ(stack_size)

【説明】ノーマル割込みハンドラで使用するスタックサイズを定義します。指定した値は4の倍数に切り上げて扱います。設定すべき値の算出方法については、「18.7 ノーマル割込みハンドラのスタック(system.stack_size)」を参照してください。
本項目の定義により、stack_sizeバイトのBC_hiirqstkセクションが生成されます。
【定義形式】数値
【定義範囲】128～0x20000000
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は0x1000)を適用(Warningあり)
【GUIコンフィギュレータの該当設定項目】CFG_IRQSTKSZ
【備考】system.vector_typeがROM_ONLY_DIRECTまたはRAM_ONLY_DIRECTの場合、本項目は意味を持ちません。また、BC_hiirqstkセクションも生成されません。

(3) カーネルスタックサイズ(kernel_stack_size)

【説明】カーネルが使用するスタックサイズを定義します。指定した値は4の倍数に切り上げて扱います。設定すべき値の算出方法については、「18.10 カーネルスタック(system.kernel_stack_size)」を参照してください。
本項目の定義により、kernel_stack_sizeバイトのBC_hiknlstkセクションが生成されます。
【定義形式】数値
【定義範囲】256～0x20000000
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は0x400)を適用(Warningあり)
【GUIコンフィギュレータの該当設定項目】CFG_KNLSTKSZ

(4) 最大タスク優先度(priority)

【説明】アプリケーションで使用するタスク優先度の最大値を定義します。
【定義形式】数値
【定義範囲】1～255
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は255)を適用(Warningあり)
【GUIコンフィギュレータの該当設定項目】CFG_MAXTSKPRI

(5) カーネル割込みマスクレベル(system_IPL)

【説明】カーネルのクリティカルセクション実行時の割込みマスクレベルを定義します。これよりも高いレベルの割込みは、「カーネル管理外割込み」の扱いとなります。
【定義形式】数値
【定義範囲】1～15
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は15)を適用(Warningあり)
【GUIコンフィギュレータの該当設定項目】CFG_KNLMSKLVL

(6) 最大メッセージ優先度(message_pri)

【説明】メールボックス機能で使用するメッセージの優先度の最大値を定義します。
なお、メールボックス機能を使用しない場合は、本項目は意味を持ちません。
【定義形式】数値
【定義範囲】1～255
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は255)を適用(Warningあり)
【GUIコンフィギュレータの該当設定項目】CFG_MAXMSGPRI

(7) タイムティックの分母(tic_deno)

【説明】タイムティックの分母を定義します。タイムティック分子と分母の少なくとも一方は1でなければなりません。

タイムティック時間(カーネルタイマの割込み周期)は、次の式で算出されます。

タイムティック時間(単位：ミリ秒) = tic_nume / tic_deno

例えば、システムクロックの周期間隔を10ミリ秒 とする場合は、tic_deno =1, tic_nume = 10 とします。

システムクロックの周期間隔を0.1ミリ秒とする場合は、tic_deno = 10, tic_nume = 1 とします。

tic_deno = 5, tic_nume = 4 などのように、分母、分子のいずれも1でない場合はエラーとなります。

tic_nume, tic_denoに関わらず、サービスコールで扱う時間の単位は常にミリ秒になります。

tic_nume, tic_denoによって、カーネルが管理する時間の精度を規定することになります。

なお、時間管理機能を使用しない場合(system.timerにNOTIMERを指定)は、tic_denoおよびtic_denoの設定は意味を持ちません。

【定義形式】数値

【定義範囲】1～100

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は1)を適用(Warningあり)

【GUIコンフィギュレータの該当設定項目】CFG_TICDENO

(8) タイムティックの分子(tic_nume)

【説明】タイムティックの分子を定義します。詳細は、前項を参照してください。

【定義形式】数値

【定義範囲】1～65535

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は1)を適用(Warningあり)

【GUIコンフィギュレータの該当設定項目】CFG_TICNUME

(9) TBR レジスタの利用形態(tbr)

【説明】CPUに内蔵されているTBRレジスタの利用方法を定義します。

【定義形式】シンボル

【定義範囲】以下のいずれかから選択してください。

- NOMANAGE：カーネルは TBR を使用も管理もしません。
- FOR_SVC：カーネルは、TBR をサービスコール呼び出し用に使います。
- TASK_CONTEXT：TBR をタスクのコンテキストレジスタとして扱います。

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はNOMANAGE)を適用(Warningあり)

【GUIコンフィギュレータの該当設定項目】CFG_TBR

(10)サービスコールのパラメータチェック(parameter_check)

【説明】サービスコールのパラメータエラーを検出するかどうかを定義します。検出しない設定にした場合は、「6. カーネルサービスコール」中の[p]で示したパラメータエラーが検出されなくなるため、このエラーケースが発生した場合の動作は未定義になります。一方、検出しない設定の方がサービスコール処理は高速になります。

【定義形式】シンボル

【定義範囲】以下のいずれかから選択してください。

- YES：サービスコールのパラメータエラーを検出する
- NO：サービスコールのパラメータエラーを検出しない

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はYES)を適用(Warningあり)

【GUIコンフィギュレータの該当設定項目】CFG_PARCHK

(11)固定長メモリプールの管理方式(mpfmanage)

【説明】固定長メモリプールの管理方式を定義します。具体的には、メモリプール領域内にカーネルの管理情報を置く(IN)か置かない(OUT)かを定義します。それぞれの相違については「5.11 固定長メモリプール」を参照してください。

なお、固定長メモリプール機能を使用しない場合は、本項目は意味を持ちません。

【定義形式】シンボル

【定義範囲】以下のいずれかから選択してください。

- IN：メモリプール領域内にカーネルの管理情報を置く
- OUT：メモリプール領域内にカーネルの管理情報を置かない

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はIN)を適用(Warningあり)

【GUIコンフィギュレータの該当設定項目】CFG_MPFMANAGE

(12)可変長メモリプールの管理方式(newmpl)

【説明】可変長メモリプールの管理方式として、従来方式(PAST)か新方式(NEW)かを定義します。新方式は、従来方式に比べて以下が改善されています。

・可変長メモリプールの空き領域の断片化度合いが軽い

・メモリブロックの獲得・解放のオーバヘッドが小さい

それぞれの相違については「5.12.1 空き領域の断片化とその対策」を参照してください。

なお、可変長メモリプール機能を使用しない場合は、本項目は意味を持ちません。

【定義形式】シンボル

【定義範囲】以下のいずれかから選択してください。

- PAST：従来方式
- NEW：新方式

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はPAST)を適用(Warningあり)

【GUIコンフィギュレータの該当設定項目】CFG_NEWMPL

(13)サービスコールトレース(trace)

【説明】サービスコールトレース機能を組み込むかどうかを定義します。蓄積されたサービスコールトレース情報は、デバッギングエクステンションを用いて表示することができます。本機能を組み込むと、サービスコール処理時間が悪化します。サービスコールトレースについては、「5.14.4 サービスコールトレース機能」を参照してください。

【定義形式】シンボル

【定義範囲】以下のいずれかから選択してください。

- NO：サービスコールトレース機能を組み込まない
- TARGET_TRACE : ターゲットシステム上のバッファにサービスコールトレース情報を蓄積します。バッファのサイズは、次項の trace_buffer で指定します。
- TOOL_TRACE : 弊社製エミュレータまたはシミュレータにサービスコールトレース情報を蓄積します。なお、エミュレータには、本機能に対応可能なエミュレータとそうでないエミュレータがあります。非対応のエミュレータ使用時は、TOOL_TRACE を指定してもデバッギングエクステンションでサービスコールトレース情報を表示できません。

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はNO)を適用(Warningあり)

【GUIコンフィギュレータの該当設定項目】CFG_TRACE

(14)サービスコールトレースのバッファサイズ(trace_buffer)

【説明】traceがTARGET_TRACEの場合に、そのバッファサイズを定義します。単位はバイトです。指定したサイズは4の倍数に切り上げられます。

【定義形式】数値

【定義範囲】512～0x20000000

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は0x10000)を適用(Warningあり)

【GUIコンフィギュレータの該当設定項目】CFG_TRCBUFSZ

【備考】system.trace != TARGET_TRACEの場合、本項目は意味を持ちません。

(15)サービスコールトレースのオブジェクト取得数(trace_object)

【説明】サービスコールトレース機能で取得可能なオブジェクト数を定義します。

【定義形式】数値

【定義範囲】0～32

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は4)を適用(Warningあり)

【GUIコンフィギュレータの該当設定項目】CFG_TRCOBJCNT

【備考】system.trace がNOの場合、本項目は意味を持ちません。

(16)オブジェクト操作機能(action)

【説明】デバッギングエクステンションなど、OSデバッグ機能に対応したデバッガから、サービスコールを発行する機能を組み込むかどうかを定義します。
オブジェクト操作機能では、周期ハンドラをひとつ使用するため、maxdefine.max_cyhおよびservice_call.cre_cycおよびicre_cycが補正される場合があります。
YESを指定した場合で、時間管理機能が選択されていない場合(clock.timer == NOTIMER)はWARNINGを表示し、本設定をNOに補正します。

【定義形式】シンボル

【定義範囲】以下のいずれかから選択してください。

- YES：オブジェクト操作機能を組み込む
- NO：オブジェクト操作機能を組み込まない

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はNO)を適用(Warningあり)

【GUIコンフィギュレータの該当設定項目】CFG_ACTION

(17)割込みベクタのタイプ(vector_type)

【説明】使用するハンドラの種類、および割込みベクタテーブルの配置方法を定義します。

【定義形式】シンボル

【定義範囲】以下のいずれかから選択してください。表14.4にそれぞれの相違を示します。

- ROM_ONLY_DIRECT
- RAM_ONLY_DIRECT
- ROM
- RAM

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はROM)を適用(Warningあり)

【GUIコンフィギュレータの該当設定項目】CFG_DIRECT, CFG_VCTRAM

表14.4　割込みベクタのタイプ

	ROM_ ONLY_DIRECT	RAM_ ONLY_DIRECT	ROM	RAM
使用できるハンドラ	・ダイレクト割込みハンドラ ・ダイレクト CPU 例外ハンドラ		・ダイレクト割込みハンドラ ・ダイレクト CPU 例外ハンドラ ・ノーマル割込みハンドラ ・ノーマル CPU 例外ハンドラ	
def_int, def_exc, vdef_trp サービスコール *1	組み込まれない	組み込まれる	組み込まれない	組み込まれる
GUI コンフィギュレータ設定	CFG_DIRECT オン CFG_VCTRAM オフ	CFG_DIRECT オン CFG_VCTRAM オン	CFG_DIRECT オフ CFG_VCTRAM オフ	CFG_DIRECT オフ CFG_VCTRAM オン

【注】 *1　service_call 定義では、これらのサービスコールの定義は無視されます。

(18)レジスタバンクの使用方法(regbank)

【説明】プロセッサに搭載されているレジスタバンクの使用方法を定義します。
ただし、kernel_intspec.hの設定によっては、本設定に関わらず全ての割込みでレジスタバンクを使用しない設定になります。詳細は、「17.3 CPU割込み仕様定義ファイル(kernel_intspec.h)の作成」を参照してください。
なお、ダイレクト割込みハンドラの記述形式は、レジスタバンクを使用するかどうかによって異なる点に注意してください。
【定義形式】シンボル
【定義範囲】表14.5に従って設定してください。

表14.5　regbank の定義方法

使用方法	定義値
レジスタバンク機能を使用しない。	NOTUSE
全割込みでレジスタバンクを使用する。 *	ALL
割込みレベルごとにレジスタバンクの使用の有無を指定する。 *	レジスタバンクを使用する割込みレベルに対応した以下のシンボルを指定してください。複数指定する場合はカンマで区切ってください。 BANKLEVEL01：割込みレベル 1 BANKLEVEL02：割込みレベル 2 BANKLEVEL03：割込みレベル 3 BANKLEVEL04：割込みレベル 4 BANKLEVEL05：割込みレベル 5 BANKLEVEL06：割込みレベル 6 BANKLEVEL07：割込みレベル 7 BANKLEVEL08：割込みレベル 8 BANKLEVEL09：割込みレベル 9 BANKLEVEL10：割込みレベル 10 BANKLEVEL11：割込みレベル 11 BANKLEVEL12：割込みレベル 12 BANKLEVEL13：割込みレベル 13 BANKLEVEL14：割込みレベル 14 BANKLEVEL15：割込みレベル 15

* kernel_intspec.hでレジスタバンクを使用しないと定義された要因は対象外です。

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はALL)を適用(Warningあり)
【GUIコンフィギュレータの該当設定項目】CFG_REGBANK
【備考】カーネルは、kernel_intspec.hのINTSPEC_IBNR_ADR1(CPUID#1の場合)またはINTSPEC_IBNR_ADR2(CPUID#2の場合)が0以外の場合、vsta_knl時に割込みコントローラの該当するレジスタを以下のように初期化します。
(1) system.regbankにALLを指定した場合
IBNRレジスタを0x4000に初期化
(2) system.regbankにBANKLEVELxxを指定した場合
IBNRレジスタを0xC000に初期化し、IBCRレジスタを指定されたレベルがレジスタバンクを使用する設定に初期化します。
(3) system.regbankにNOTUSEを指定した場合
IBNRレジスタを0に初期化します。

14.3.3 最大 ID 定義(maxdefine)

各カーネルオブジェクトの最大ローカル ID などを定義します。各カーネルオブジェクトは、ローカル ID=1 から指定する最大 ID までを使用できます。

static_task 以外の項目は、定義を省略できます。この場合、cfg ファイル中の対応するオブジェクトの定義(xxxx[])等に応じた最低限の値が自動的に設定されます。

また、省略しない場合でも、設定値が cfg ファイル中の対応するオブジェクトの定義(xxxx[ ])で指定した番号(ID 番号やベクタ番号)に満たない場合は、最大 ID が自動的に拡張されます。この場合、ワーニングメッセージが出力されます。

■形式

```
maxdefine{
        max_task            = <設定値>; // 最大ローカルタスク ID
        max_statictask      = <設定値>; // スタティックスタックを使用する最大ローカルタスク ID
        max_sem             = <設定値>; // 最大ローカルセマフォ ID
        max_flag            = <設定値>; // 最大ローカルイベントフラグ ID
        max_dtq             = <設定値>; // 最大ローカルデータキューID
        max_mbx             = <設定値>; // 最大ローカルメールボックス ID
        max_mtx             = <設定値>; // 最大ローカルミューテックス ID
        max_mbf             = <設定値>; // 最大ローカルメッセージバッファ ID
        max_mpf             = <設定値>; // 最大ローカル固定長メモリプール ID
        max_mpl             = <設定値>; // 最大ローカル可変長メモリプール ID
        max_cyh             = <設定値>; // 最大ローカル周期ハンドラ ID
        max_alh             = <設定値>; // 最大ローカルアラームハンドラ ID
        max_fncd            = <設定値>; // 最大拡張サービスコール機能コード
        max_int             = <設定値>; // 最大割込みベクタ番号
};
```

■内容

(1) 最大ローカルタスク ID(max_task)

【説明】ローカルタスクIDとして、1～max_taskを使用できます。
【定義形式】数値
【定義範囲】1～1023
【省略時の扱い】自動算出
【GUIコンフィギュレータの該当設定項目】CFG_MAXTSKID

(2) スタティックスタックを使用する最大ローカルタスク ID(max_statictask)

【説明】本OSでは、ローカルタスクIDが1～max_statictaskのタスクは「14.3.12 スタティックスタック領域定義 (static_stack[])」で定義されるスタティックスタックを使用します。0を指定すると、すべてのタスクがスタティクスタックを使用しないことになります。
【定義形式】数値
【定義範囲】0～1023
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は0)を適用(Warningあり)
【GUIコンフィギュレータの該当設定項目】CFG_STSTKID

(3) 最大ローカルセマフォ ID(max_sem)

【説明】ローカルセマフォIDとして、1～max_semを使用できます。max_semに0を指定した場合は、一切セマフォを使用できません。
【定義形式】数値
【定義範囲】0～1023
【省略時の扱い】自動算出
【GUIコンフィギュレータの該当設定項目】CFG_MAXSEMID
【備考】((service_call.cre_sem==NO)&&(service_call.icre_sem==NO)) の場合、max_semは0と扱われます。また、すべてのsemaphore[]は無視されます。

(4) 最大ローカルイベントフラグ ID(max_flag)

【説明】ローカルイベントフラグIDとして、1～max_flagを使用できます。max_flagに0を指定した場合は、一切イベントフラグを使用できません。
【定義形式】数値
【定義範囲】0～1023
【省略時の扱い】自動算出
【GUIコンフィギュレータの該当設定項目】CFG_MAXFLGID
【備考】((service_call.cre_flg==NO)&&(service_call.icre_flg==NO)) の場合、max_flagは0と扱われます。また、すべてのeventflag[]は無視されます。

(5) 最大ローカルデータキューID(max_dtq)

【説明】ローカルデータキューIDとして、1～max_dtqを使用できます。max_dtqに0を指定した場合は、一切データキューを使用できません。
【定義形式】数値
【定義範囲】0～1023
【省略時の扱い】自動算出
【GUIコンフィギュレータの該当設定項目】CFG_MAXDTQID
【備考】((service_call.cre_dtq==NO)&&(service_call.icre_dtq==NO))の場合、max_dtqは0と扱われます。また、すべてのdataqueue[]は無視されます。memdtqの定義も無視され、デフォルトデータキュー用領域は生成されません。

(6) 最大ローカルメールボックス ID(max_mbx)

【説明】ローカルメールボックスIDとして、1～max_mbxを使用できます。max_mbxに0を指定した場合は、一切メールボックスを使用できません。
【定義形式】数値
【定義範囲】0～1023
【省略時の扱い】自動算出
【GUIコンフィギュレータの該当設定項目】CFG_MAXMBXID
【備考】((service_call.cre_mbx==NO)&&(service_call.icre_mbx==NO))の場合、max_mbxは0と扱われます。また、すべてのmailbox[]は無視されます。

(7) 最大ローカルミューテックス ID(max_mtx)

【説明】ローカルミューテックスIDとして、1～max_mtxを使用できます。max_mtxに0を指定した場合は、一切ミューテックスを使用できません。
【定義形式】数値
【定義範囲】0～1023
【省略時の扱い】自動算出
【GUIコンフィギュレータの該当設定項目】CFG_MAXMTXID
【備考】(service_call.cre_mtx==NO) の場合、max_mtxは0と扱われます。また、すべてのmutex[]は無視されます。

(8) 最大ローカルメッセージバッファ ID(max_mbf)

【説明】ローカルメッセージバッファIDとして、1～max_mbfを使用できます。max_mbfに0を指定した場合は、一切メッセージバッファを使用できません。
【定義形式】数値
【定義範囲】0～1023
【省略時の扱い】自動算出
【GUIコンフィギュレータの該当設定項目】CFG_MAXMBFID
【備考】((service_call.cre_mbf==NO)&&(service_call.icre_mbf==NO)) の場合、max_mbfは0と扱われます。また、すべてのmessage_buffer[]は無視されます。memmbfの定義も無視され、デフォルトメッセージバッファ用領域は生成されません。

(9) 最大ローカル固定長メモリプール ID(max_mpf)

【説明】ローカル固定長メモリプールIDとして、1～max_mpfを使用できます。max_mpfに0を指定した場合は、一切固定長メモリプールを使用できません。
【定義形式】数値
【定義範囲】0～1022
【省略時の扱い】自動算出
【GUIコンフィギュレータの該当設定項目】CFG_MAXMPFID
【備考】((service_call.cre_mpf==NO)&&(service_call.icre_mpf==NO)) の場合、max_mpfは0と扱われます。また、すべてのmemorypool[]は無視されます。memmpfの定義も無視され、デフォルト固定長メモリプール用領域は生成されません。
なお、remote_svc.num_waitが正の場合は、cre_mpfとicre_mpfはともにYESに補正されます。

(10)最大ローカル可変長メモリプール ID(max_mpl)

【説明】ローカル可変長メモリプールIDとして、1～max_mplを使用できます。max_mplに0を指定した場合は、一切可変長メモリプールを使用できません。
【定義形式】数値
【定義範囲】0～1023
【省略時の扱い】自動算出
【GUIコンフィギュレータの該当設定項目】CFG_MAXMPLID
【備考】((service_call.cre_mpl==NO)&&(service_call.icre_mpl==NO)) の場合、max_mplは0と扱われます。また、すべてのvariable_memorypool[]は無視されます。memmplの定義も無視され、デフォルト可変長メモリプール用領域は生成されません。

(11)最大ローカル周期ハンドラ ID(max_cyh)

【説明】ローカル周期ハンドラIDとして、1～max_cyhを使用できます。max_cyhに0を指定した場合は、一切周期ハンドラを使用できません。
【定義形式】数値
【定義範囲】0～14
【省略時の扱い】自動算出
【GUIコンフィギュレータの該当設定項目】CFG_MAXCYCID
【備考】((service_call.cre_cyc==NO)&&(service_call.icre_cyc==NO)) の場合、max_cyhは0と扱われます。また、すべてのcyclic_hand[]は無視されます。
なお、system.actionがYESの場合は、cre_cycとicre_cycはともにYESに補正されます。

(12)最大ローカルアラームハンドラ ID(max_alh)

【説明】ローカルアラームハンドラIDとして、1～max_alhを使用できます。max_alhに0を指定した場合は、一切アラームハンドラを使用できません。
【定義形式】数値
【定義範囲】0～15
【省略時の扱い】自動算出
【GUIコンフィギュレータの該当設定項目】CFG_MAXALMID
【備考】((service_call.cre_alm==NO)&&(service_call.icre_alm==NO)) の場合、max_alhは0と扱われます。また、すべてのalarm_hand[]は無視されます。

(13)最大拡張サービスコール機能コード(max_fncd)

【説明】拡張サービスコールの機能コードとして、1～max_fncdを使用できます。max_fncdに0を指定した場合は、一切拡張サービスコールを使用できません。
【定義形式】数値
【定義範囲】0～1023
【省略時の扱い】自動算出
【GUIコンフィギュレータの該当設定項目】CFG_MAXSVCCD
【備考】((service_call.def_svc==NO)&& service_call.idef_svc==NO)) の場合、max_fncdは0と扱われます。また、すべてのextend_svc []は無視されます。

(14)最大割込みベクタ番号(max_int)

【説明】割込み・例外ベクタ番号として、0～max_intを使用できます。ただし、リセットベクタ(ベクタ番号0～3)は、本OSの管理外です。
システム稼動中にmax_intを超えるベクタ番号の割込みが発生した場合の振る舞いは未定義です。
【定義形式】数値
【定義範囲】64～511
【省略時の扱い】自動算出
【GUIコンフィギュレータの該当設定項目】CFG_MAXVCTNO

14.3.4 デフォルトタスクスタック用領域定義(memstk)

本項目の定義により、all_memsize バイトの BC_hitskstk セクションが生成されます。

■形式

```
memstk{
        all_memsize          = <設定値>; // デフォルトタスクスタック用領域のサイズ
};
```

■内容

(1) デフォルトタスクスタック用領域のサイズ(all_memsize)

【説明】デフォルトタスクスタック用領域のサイズを定義します。指定した値は4の倍数に切り上げて扱います。設定すべき値の算出方法については、「18.6.3 デフォルトタスクスタック領域サイズの算出(memstk.all_memsize)」を参照してください。

本項目の定義により、all_memsizeバイトのBC_hitskstkセクションが生成されます。

【定義形式】数値

【定義範囲】0～0x20000000

【省略時の扱い】全task[]を元に、「18.6.3 デフォルトタスクスタック領域サイズの算出(memstk.all_memsize)」に従って自動算出

【GUIコンフィギュレータの該当設定項目】CFG_TSKSTKSZ

14.3.5 デフォルトデータキュー用領域定義(memdtq)

本項目の定義により、all_memsize バイトの BC_hidtq セクションが生成されます。

■形式

```
memdtq {
        all_memsize          = <設定値>; // デフォルトデータキュー用領域のサイズ
};
```

■内容

(1) デフォルトデータキュー用領域のサイズ(all_memsize)

【説明】デフォルトデータキュー用領域のサイズを定義します。指定した値は4の倍数に切り上げて扱います。

なお、デフォルトデータキュー用領域に必要なサイズは、次の式で求めることができます。

Σ(TSZ_DTQ(データ数)+ 0x10) + 0x1C

Σの項は、以下の全ての条件を満たすデータキューについて計算します。

(1)データ数が0でない

(2)データキューアドレスがNULLである。

ただし、Σの項が0になる場合はΣの項を0x10として計算します。

本項目の定義により、all_memsizeバイトのBC_hidtqセクションが生成されます。

【定義形式】数値

【定義範囲】0～0x20000000

【省略時の扱い】全dataqueue[]を元に、前述の方法で自動算出

【GUIコンフィギュレータの該当設定項目】CFG_DTQSZ

14.3.6 デフォルトメッセージバッファ用領域定義(memmbf)

本項目の定義により、all_memsize バイトの BC_himbf セクションが生成されます。

■形式

```
memmbf{
        all_memsize          = <設定値>; // デフォルトメッセージバッファ用領域のサイズ
};
```

■内容

(1) デフォルトメッセージバッファ用領域のサイズ(all_memsize)

【説明】デフォルトメッセージバッファ用領域のサイズを定義します。指定した値は4の倍数に切り上げて扱います。

なお、デフォルトメッセージバッファ用領域に必要なサイズは、次の式で求めることができます。

Σ(メッセージバッファのサイズ ＋0x10)＋0x1C

Σの項は、以下の全ての条件を満たすメッセージバッファについて計算します。

(1)メッセージバッファのサイズが0でない。

(2)メッセージバッファのアドレスがNULLである。

ただし、Σの項が0になる場合はΣの項を0x10として計算します。

本項目の定義により、all_memsizeバイトのBC_himbfセクションが生成されます。

【定義形式】数値

【定義範囲】0～0x20000000

【省略時の扱い】全message_buffer[]を元に、前述の方法で自動算出

【GUIコンフィギュレータの該当設定項目】CFG_MBFSZ

14.3.7 デフォルト固定長メモリプール用領域定義(memmpf)

本項目の定義により、all_memsize バイトの BC_himpf セクションが生成されます。

■形式

```
memmpf{
        all_memsize         = <設定値>; // デフォルト固定長メモリプール用領域のサイズ
};
```

■内容

(1) デフォルト固定長メモリプール用領域のサイズ(all_memsize)

【説明】デフォルト固定長メモリプール用領域のサイズを定義します。指定した値は4の倍数に切り上げて扱います。

なお、デフォルト固定長メモリプール用領域に必要なサイズは、次の式で求めることができます。

Σ(TSZ_MPF(ブロック数, ブロックサイズ) + 0x10) + 0x1C

Σの項は、アドレスがNULLでない固定長メモリプールについて計算します。ただし、Σの項が0になる場合はΣの項を0x10として計算します。

なお、TSZ_MPF()マクロの定義内容は、system.mpfmanageによって異なります。

本項目の定義により、all_memsizeバイトのBC_himpfセクションが生成されます。

【定義形式】数値

【定義範囲】0～0x20000000

【省略時の扱い】説明欄に記載の仕様で自動算出

【GUIコンフィギュレータの該当設定項目】CFG_MPFSZ

RENESAS

14.3.8 デフォルト可変長メモリプール用領域定義(memmpl)

本項目の定義により、all_memsize バイトの BC_himpl セクションが生成されます。

■形式

```
memmpl{
        all_memsize          = <設定値>; // デフォルト可変長メモリプール用領域のサイズ
};
```

■内容

(1) デフォルト可変長メモリプール用領域のサイズ(all_memsize)

【説明】デフォルト可変長メモリプール用領域のサイズを定義します。指定した値は4の倍数に切り上げて扱います。

なお、デフォルト可変長メモリプール用領域に必要なサイズは、次の式で求めることができます。

Σ(メモリプールのサイズ ＋0x10)＋0x1C

Σの項は、アドレスがNULLでない可変長メモリプールについて計算します。ただし、Σの項が0になる場合はΣの項を0x10として計算します。

本項目の定義により、all_memsizeバイトのBC_himplセクションが生成されます。

【定義形式】数値

【定義範囲】0～0x20000000

【省略時の扱い】説明欄に記載の仕様で自動算出

【GUIコンフィギュレータの該当設定項目】CFG_MPLSZ

14.3.9 システムクロック定義(clock)

システムクロックに関する定義を行います。

■形式

```
clock {
        timer               = <設定値>; // (1)タイマモード
        IPL                 = <設定値>; // (2)タイマ割込みレベル
        number              = <設定値>; // (3)タイマ割込みのベクタ番号
        stack_size          = <設定値>; // (4)タイマスタックサイズ
};
```

■内容

(1) タイマモード(timer)

【説明】カーネルの時間管理機能を使用するかどうかを定義します。時間管理機能を使用する場合は、タイマドライバをカーネルにリンクする必要があります。

TIMERを指定した場合は、cfgファイルに以下の暗黙の記述があるものとして扱います。

```
// タイマドライバの初期化ルーチンの定義
init_routine[] {
    exinf = 0;
    entry_address = tdr_ini_tmr();
};

// カーネルのタイマ割込みハンドラの定義
interrupt_vector[<clock.number の設定値>] {
    direct = ON;
    regbank = ON;
    entry_address = _kernel_isig_tim();
};
```

【定義形式】シンボル

【定義範囲】以下のいずれかから選択してください。

- TIMER：カーネルの時間管理機能を使用する。
- NOTIMER：カーネルの時間管理機能を使用しない。

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はNOTIMER)を適用(Warningあり)

【GUIコンフィギュレータの該当設定項目】CFG_TIMUSE

(2) タイマ割込みレベル(IPL)

【説明】カーネルタイマの割込みレベルを定義します。タイマモードがNOTIMERの場合、本項目は意味を持ちません。
設定値はカーネル割込みマスクレベル(system.system_IPL)以下でなければなりません。そうでない場合はエラーになります。
【定義形式】数値
【定義範囲】1～15
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は15)を適用(Warningあり)
(system.system_IPLが15未満の場合は、結果的にエラーになります)
【GUIコンフィギュレータの該当設定項目】CFG_TIMINTLVL
【備考】clock.timerがNOTIMERの場合、本項目は意味を持ちません。

(3) タイマ割込みベクタ番号(number)

【説明】カーネルタイマで使用するベクタ番号を定義します。
ここで指定するベクタ番号に対して、interrupt_vector[]の記述がある場合は、コンフィギュレータはエラーを報告します。
【定義形式】数値
【定義範囲】64～511
【省略時の扱い】省略不可(エラー)
【GUIコンフィギュレータの該当設定項目】CFG_TIMINTNO
【備考】clock.timerがNOTIMERの場合、本項目は意味を持ちません。

(4) タイマスタックサイズ(stack_size)

【説明】タイマスタックサイズを定義します。指定した値は4の倍数に切り上げて扱います。設定すべき値の算出方法については、「18.9 タイマスタック(clock.stack_size)」を参照してください。
本項目の定義により、stack_sizeバイトのBC_hitmrstkセクションが生成されます。
【定義形式】数値
【定義範囲】0～0x20000000
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は0x100)を適用(Warningあり)
【GUIコンフィギュレータの該当設定項目】CFG_TMRSTKSZ
【備考】clock.timerがNOTIMERの場合、本項目は意味を持ちません。

14.3.10 リモートサービスコール環境定義(remote_svc)

■形式

```
remote_svc{
        num_server          = <設定値>; //  (1)SVC サーバタスクの数
        priority            = <設定値>; //  (2)SVC サーバタスクの優先度
        stack_size          = <設定値>; //  (3)SVC サーバタスクのスタック使用サイズ
        ipi_portid          = <設定値>; //  (4)使用する IPI ポート ID
        num_wait            = <設定値>; //  (5)SVC サーバタスク空き待ちタスクの最大数
};
```

■内容

(1) SVC サーバタスクの数(num_server)

【説明】SVCサーバタスクの数を定義します。サーバ数とは、他のCPUから同時に受理可能なサービスコール数です。

0を指定すると、他CPUからのリモートサービスコールを一切受理できなくなります。なお、他CPUへのリモートサービスコール要求は、本設定に関わらず常に可能です。

0以外を指定すると、service_callの以下がYESに補正されます。

- acre_tsk, iacre_tsk
- ter_tsk
- del_tsk
- slp_tsk
- wup_tsk, iwup_tsk

【定義形式】数値

【定義範囲】0～1023

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は1)を適用(Warningあり)

【GUIコンフィギュレータの該当設定項目】CFG_REMOTE_NUMSERVER

(2) SVC サーバタスクの優先度(priority)

【説明】リモートサービスコール要求を処理するSVCサーバタスクの優先度を定義します。

【定義形式】数値

【定義範囲】1～255

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は1)を適用(Warningあり)

【GUIコンフィギュレータの該当設定項目】CFG_REMOTE_PRIORITY

【備考】remoete_svc.num_serverが0の場合、本項目は意味を持ちません。

RENESAS

(3) SVC サーバタスクのスタック使用サイズ(stack_size)

【説明】SVCサーバタスクが使用するスタックサイズを定義します。指定した値は4の倍数に切り上げて扱います。設定すべき値の算出方法については、「18.6.4 SVCサーバタスクのスタックサイズ(remote_svc.stack_size)」を参照してください。
本設定により、stack_size×num_serverバイトのBC_hirmtstkセクションが生成されます。
【定義形式】数値
【定義範囲】128～0x20000000
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は0x200)を適用(Warningあり)
【GUIコンフィギュレータの該当設定項目】CFG_REMOTE_STKSZ
【備考】remoete_svc.num_serverが0の場合、本項目は意味を持ちません。

(4) 使用する IPI ポート ID(ipi_portid)

【説明】リモートサービスコールを受理するため、および他CPUに対して呼び出すリモートサービスコールからのリターンを受理するために使用するIPIポートIDを定義します。
IPIの割込みレベルは(15 – ポートID)で算出されますが、この割込みレベルはsystem.system_IPL以下でなければなりません。
【定義形式】数値
【定義範囲】0～7
【省略時の扱い】省略不可(エラー)
【GUIコンフィギュレータの該当設定項目】CFG_REMOTE_IPI

(5) SVC サーバタスク空き待ちタスクの最大数(num_wait)

【説明】リモートサービスコールを呼び出した時、対象CPU側のSVCサーバタスクの空きがない場合は待ち状態になります。num_waitには、本カーネル側で同時に他CPU側のSVCサーバタスクの空きを待つタスクの最大数を定義します。
リモートサービスコール発行時点で以下の全ての条件が満たされる場合、そのリモートサービスコールには、直ちにEV_NORESOURCEが返ります。
(1) 対象CPU側に、SVCサーバタスクの空きがない。
(2) 呼び出し元のカーネル側で、既にnum_waitの数だけSVCサーバタスクの空きを待っているタスクが存在する。
常にSVCサーバタスクの空きを待たないようにするには、num_waitに0を定義してください。
なお、指定した値がmaxdefine.max_taskを超える場合は、max_taskと同じ値に補正されます。
0以外を指定すると、service_callの以下がYESに補正されます。

- slp_tsk
- wup_tsk, iwup_tsk
- cre_mpf, icre_mpf
- acre_mpf, iacre_mpf
- del_mpf
- pget_mpf
- rel_mpf

【定義形式】数値
【定義範囲】0～1023
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は0)を適用(Warningあり)。
【GUIコンフィギュレータの該当設定項目】CFG_REMOTE_NUMWAIT

14.3.11 タスク定義(task[])

task[]は、タスクを定義(生成)する定義項目です。

task[]は、GUI コンフィギュレータの[タスクの生成]ダイアログボックスと対応します。

タスクには、大きく static_stack[]で定義したスタティックスタックを使用するタスクと、スタティックスタック以外のスタックを使用するタスクに分類され、これらはローカルタスク ID によって分類されます。

スタティックスタックを使用するタスクを定義する場合は、ローカル ID 番号を省略できません。

表 14.6に、タスクの種類を示します。

表14.6 タスクの種類

スタック領域	task[]の定義項目			
	ローカル ID 番号	stack_size	stack_section	stack_address
デフォルトタスクスタック用領域から割り付ける	system.max_statictask +1 以上(省略時はこの範囲から割り当てられます)	記述する	記述しない	記述しない
コンフィギュレータがスタック領域を生成する		記述する	記述する	記述しない
ユーザ側でスタック領域を確保する		記述する	記述しない	記述する
スタティックスタックを使用する	1～ system.max_statictask (省略不可)	無視	無視	無視

■形式

```
task[<ローカル ID 番号>]{                    //  (1)ローカル ID 番号
        name               = <設定値>;  //  (2)ID 名称
        export             = <設定値>;  //  (3)ID 名称のエクスポート
        entry_address      = <設定値>;  //  (4)タスクの開始アドレス
        stack_size         = <設定値>;  //  (5)スタックサイズ
        stack_section      = <設定値>;  //  (6)スタック領域に付与するセクション名
        stack_address      = <設定値>;  //  (7)ユーザが確保したスタック領域の先頭アドレス
        priority           = <設定値>;  //  (8)タスクの起動時優先度
        initial_start      = <設定値>;  //  (9)初期起動状態
        exinf              = <設定値>;  //  (10)拡張情報
        fpu                = <設定値>;  //  (11)タスクでの FPU 使用の有無
        tex_address        = <設定値>;  //  (12)タスク例外処理ルーチンの開始アドレス
        tex_fpu            = <設定値>;  //  (13)タスク例外処理ルーチンでの FPU の使用の有無
};
```


RENESAS

■内容

(1) ローカル ID 番号

【説明】ローカルID番号は1～1023の範囲でなければなりません。
ローカルID番号は省略可能です。省略した場合、コンフィギュレータは maxdefine.max_statictask+1以上のローカルID番号を自動的に割り当てます。

なお、1～maxdefine.max_statictaskのローカルID番号のタスクは、「14.3.12 スタティックスタック領域定義 (static_stack[])」で定義するスタティックスタックを使用します。
【定義形式】数値
【定義範囲】1～1023
【省略時の扱い】maxdefine.max_statictask+1以上のローカルID番号を自動的に割り当てます。

(2) ID 名称(name)

【説明】タスクのID名称を定義します。指定されたID名称は、ID名称ヘッダファイルに出力されます。
【定義形式】シンボル
【定義範囲】なし
【省略時の扱い】省略した場合は、ID名称なしと扱い、ID名称ヘッダファイルにも出力しません。ただし、ローカルID番号を省略した場合はID名称は省略できません。この場合、エラーを表示します。

(3) ID 名称のエクスポート(export)

【説明】ID名称を他CPUにエクスポートするかどうかを定義します。
【定義形式】シンボル
【定義範囲】YESまたはNO
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はNO)を適用(Warningなし)
【備考】ID名称の指定がない場合、本項目は意味を持ちません。

(4) タスクの開始アドレス(entry_address)

【説明】タスクの実行開始アドレスを定義します。
【定義形式】外部参照名または数値
【定義範囲】数値の場合は、0～0xFFFFFFFFの範囲の偶数
【省略時の扱い】省略不可(エラー)

(5) スタックサイズ(stack_size)

【説明】タスクが使用するスタックサイズを定義します。指定した値は4の倍数に切り上げて扱います。タスクに必要なスタックサイズの算出方法は、「18.6.1 スタックサイズの算出」を参照してください。
【定義形式】数値
【定義範囲】128～0x20000000
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は0x100)を適用(Warningあり)
【備考】ID番号が1～maxdefine.max_statictaskの場合、タスクはスタティックスタックを使用するため、本項目は意味を持ちません。

(6) スタック領域に付与するセクション名(stack_section)

【説明】stack_sectionは、コンフィギュレータでスタック領域を生成させる場合に記述します。
表14.6も参照してください。
なお、stack_sectionとstack_addressの両方の記述がある場合はエラーとなります。
実際に生成されるセクション名は、stack_sectionで指定された文字列の先頭に'B'を付加した名称になります。リンク時には、ユーザがこのセクションを適切なアドレスに配置する必要があります。
【定義形式】シンボル
【定義範囲】なし
【省略時の扱い】表14.6参照
【備考】ID番号が1～maxdefine.max_statictaskの場合、タスクはスタティックスタックを使用するため、本項目は意味を持ちません。

(7) ユーザが確保したスタック領域の先頭アドレス(stack_address)

【説明】ユーザ側で確保したスタック領域を使用する場合は、stack_addressにそのスタック領域の先頭アドレスを定義してください。stack_addressからstack_sizeバイトの領域は、ユーザ側で確保する必要があります。
【定義形式】外部参照名または数値
【定義範囲】数値の場合は、0～0xFFFFFFFFの範囲の4の倍数
【省略時の扱い】表14.6参照
【備考】ID番号が1～maxdefine.max_statictaskの場合、タスクはスタティックスタックを使用するため、本項目は意味を持ちません。

(8) タスクの起動時優先度(priority)

【説明】タスクの起動時優先度を定義します。指定する値は1～255の範囲でなければなりません。
また、最大タスク優先度(system.priority)以下でなければなりません。
【定義形式】数値
【定義範囲】1～255
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は1)を適用(Warningあり)

(9) 初期起動状態(initial_start)

【説明】タスクの初期状態をREADY状態とするかDORMANT状態とするかを定義します。本項目は、タスクのTA_ACT属性に該当します。
【定義形式】シンボル
【定義範囲】以下のいずれかから選択してください。
- ON：カーネル起動に READY 状態にする。
- OFF：カーネル起動に DORMANT 状態にする。

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はOFF)を適用(Warningなし)

(10)拡張情報(exinf)

【説明】タスクの拡張情報を定義します。
【定義形式】外部参照名または数値
【定義範囲】数値の場合は0～0xFFFFFFFF
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は0)を適用(Warningなし)

(11)タスクでの FPU 使用(fpu)

【説明】タスクでFPUを使用するかどうか、すなわちFPUレジスタをタスクのコンテキストに含めるかどうかを定義します。本項目は、タスクのTA_COP1属性に該当します。
【定義形式】シンボル
【定義範囲】以下のいずれかから選択してください。

- ON：FPU を使用する。
- OFF：FPU を使用しない。

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はOFF)を適用(Warningあり)

(12)タスク例外処理ルーチンの開始アドレス(tex_address)

【説明】タスク例外処理ルーチンの実行開始アドレスを定義します。タスク例外処理ルーチンを定義しない場合は、本項目を記述しないでください。
【定義形式】外部参照名または数値
【定義範囲】数値の場合は、0～0xFFFFFFFFの範囲の偶数
【省略時の扱い】タスク例外処理ルーチンを定義しません。
【備考】(service_call.def_tex == NO) && (service_call.idef_tex == NO)の場合、本項目は意味を持ちません。

(13)タスク例外処理ルーチンでの FPU 使用(tex_fpu)

【説明】タスク例外処理ルーチンでFPUを使用するかどうか、すなわちFPUレジスタをタスク例外処理ルーチンのコンテキストに含めるかどうかを定義します。本項目は、タスク例外処理ルーチンのTA_COP1属性に該当します。
【定義形式】シンボル
【定義範囲】以下のいずれかから選択してください。

- ON：FPU を使用する。
- OFF：FPU を使用しない。

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はOFF)を適用(Warningあり)
【備考】(service_call.def_tex == NO) && (service_call.idef_tex == NO)の場合、本項目は意味を持ちません。

14.3.12 スタティックスタック領域定義 (static_stack[])

satic_stack[]は、スタティックスタック領域を定義し、そのスタックを使用するタスクを関連付けます。maxdefine.max_staticstack > 0 の場合は、static_stack の記述が必須です。

static_stack は複数記述できます。

1〜maxdefine.max_staticstack までのローカルタスク ID は、すべての static_stack[].tskid を通じて一度だけ出現するようにしなければなりません。

maxdefine.max_statictask が 0 の場合、static_stack[]は単に無視されます。この場合、ワーニングメッセージが出力されます。

■形式

```
static_stack[<スタック番号>]{                // (1)スタック番号
        tskid = <設定値>(,<設定値>,…);   // (2)当該スタックを使用するローカルタスク ID
        stack_size          = <設定値>; // (3)スタックサイズ
        stack_section       = <設定値>; // (4)スタック領域に付与するセクション名
};
```

■内容

(1) スタック番号

【説明】コンフィギュレータがstatic_stack[]を区別するための番号です。各static_stackのスタック番号は、ユニークでなければなりません。
スタック番号は、1から連続して採番する必要はありません。
【定義形式】数値
【定義範囲】1〜1023
【省略時の扱い】省略不可(エラー)

(2) 当該スタックを使用するローカルタスク ID

【説明】このスタック領域を使用するタスクを、ローカルID番号またはID名称で定義します。
ローカルID番号で指定する場合は、1〜maxdefine.max_statictaskでなければなりませんが、そのローカルID番号がtask[]で定義されていなくてもかまいません。
ID名称で指定する場合は、その名称のtask[]が存在し、かつそのtask[]ではID番号が省略されておらず、かつローカルID番号が1〜maxdefine.max_statictaskでなければなりません。
複数のタスクを指定する場合は、それぞれをカンマで区切ります。複数のタスクを指定すると、それらのタスク間でこのスタティックスタックを共有することになります(共有スタック機能)。
【定義形式】シンボルまたは数値
【定義範囲】シンボル(ID名称)の場合は、ローカルID番号が省略されておらず、かつmaxdefine.max_statictask以下であるtask[].nameのいずれか。
数値の場合は、1〜maxdefine.max_statictask
【省略時の扱い】省略不可(エラー)

RENESAS

(3) スタックサイズ(stack_size)

【説明】スタック領域のサイズを定義します。指定した値は4の倍数に切り上げて扱います。タスクに必要なスタックサイズの算出方法は、「18.6 タスクのスタック」を参照してください。

【定義形式】数値

【定義範囲】128～0x20000000

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は0x100)を適用(Warningあり)

(4) スタック領域に付与するセクション名(stack_section)

【説明】スタック領域に付与するセクション名を定義します。コンフィギュレータは、stack_sizeで指定された未初期化データセクションを、stack_sectionで指定された文字列の先頭に’B’を付加したセクション名で生成します。リンク時には、ユーザがこのセクションを適切なアドレスに配置する必要があります。

なお、GUIコンフィギュレータ使用時は、常にstack_section = C_histstkの扱いとなります(制限)。

【定義形式】シンボル

【定義範囲】なし

【省略可能条件】常に省略可能

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はC_histstk)を適用(Warningあり)

14.3.13 セマフォ定義(semaphore[])

semaphore[]は、セマフォを定義(生成)する定義項目です。
semaphore[]は、GUI コンフィギュレータの[セマフォの生成]ダイアログボックスと対応します。
service_call.cre_sem, icre_sem が共に NO の場合は、semaphore[]は無視されます。この場合、ワーニングメッセージが表示されます。

■形式

```
semaphore[<ローカル ID 番号>]{                  //  (1)ローカル ID 番号
        name                 = <設定値>; //  (2)ID 名称
        export               = <設定値>; //  (3)ID 名称のエクスポート
        max_count            = <設定値>; //  (4)セマフォカウンタ最大値
        initial_count        = <設定値>; //  (5)セマフォカウンタ初期値
        wait_queue           = <設定値>; //  (6)待ち行列属性
};
```

■内容

(1) ローカル ID 番号

【説明】ローカルID番号は1～1023の範囲でなければなりません。
ローカルID番号は省略可能です。省略した場合、コンフィギュレータはローカルID番号を自動的に割り当てます。
【定義形式】数値
【定義範囲】1～1023
【省略時の扱い】ローカルID番号を自動的に割り当てます。

(2) ID 名称(name)

【説明】セマフォのID名称を定義します。指定されたID名称は、ID名称ヘッダファイルに出力されます。
【定義形式】シンボル
【定義範囲】なし
【省略時の扱い】省略した場合は、ID名称なしと扱い、ID名称ヘッダファイルにも出力しません。
ただし、ローカルID番号を省略した場合はID名称は省略できません。この場合、エラーを表示します。

(3) ID 名称のエクスポート(export)

【説明】ID名称を他CPUにエクスポートするかどうかを定義します。
【定義形式】シンボル
【定義範囲】YESまたはNO
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はNO)を適用(Warningなし)
【備考】ID名称の指定がない場合、本項目は意味を持ちません。

RENESAS

(4) セマフォカウンタ最大値(max_count)

【説明】セマフォカウンタの最大値を定義します。
【定義形式】数値
【定義範囲】1～65535
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は1)を適用(Warningあり)

(5) セマフォカウンタ初期値(initial_count)

【説明】セマフォカウンタの初期値を定義します。セマフォカウンタ最大値以下でなければなりません。
【定義形式】数値
【定義範囲】0～65535
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は1)を適用(Warningあり)

(6) 待ち行列属性(wait_queue)

【説明】待ち行列属性を定義します。
【定義形式】シンボル
【定義範囲】以下のいずれかから選択してください。
- TA_TFIFO：待ち行列は FIFO 順
- TA_TPRI：待ち行列はタスク優先度順

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はTA_TFIFO)を適用(Warningあり)

14.3.14 イベントフラグ定義(flag[])

flag[]は、イベントフラグを定義(生成)する定義項目です。

flag[]は、GUI コンフィギュレータの[イベントフラグの生成]ダイアログボックスと対応します。

service_call.cre_flg, icre_flg が共に NO の場合は、flag[]は無視されます。この場合、ワーニングメッセージが表示されます。

■形式

```
flag[<ローカル ID 番号>]{                           //  (1)ローカル ID 番号
        name                = <設定値>; //  (2)ID 名称
        export              = <設定値>; //  (3)ID 名称のエクスポート
        initial_pattern     = <設定値>; //  (4 イベントフラグの初期ビットパターン
        wait_queue          = <設定値>; //  (5)待ち行列属性
        wait_multi          = <設定値>; //  (6)複数待ちの許可属性
        clear_attribute     = <設定値>; //  (7)クリア属性
};
```

■内容

(1) ローカル ID 番号

【説明】ローカルID番号は1～1023の範囲でなければなりません。
ローカルID番号は省略可能です。省略した場合、コンフィギュレータはローカルID番号を自動的に割り当てます。
【定義形式】数値
【定義範囲】1～1023
【省略時の扱い】ローカルID番号を自動的に割り当てます。

(2) ID 名称(name)

【説明】イベントフラグのID名称を定義します。指定されたID名称は、ID名称ヘッダファイルに出力されます。
【定義形式】シンボル
【定義範囲】なし
【省略時の扱い】省略した場合は、ID名称なしと扱い、ID名称ヘッダファイルにも出力しません。
ただし、ローカルID番号を省略した場合はID名称は省略できません。この場合、エラーを表示します。

(3) ID 名称のエクスポート(export)

【説明】ID名称を他CPUにエクスポートするかどうかを定義します。
【定義形式】シンボル
【定義範囲】YESまたはNO
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はNO)を適用(Warningなし)
【備考】ID名称の指定がない場合、本項目は意味を持ちません。

(4) イベントフラグの初期ビットパターン(initial_pattern)

【説明】イベントフラグの初期ビットパターンを定義します。
【定義形式】数値
【定義範囲】0～0xFFFFFFFF
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は0)を適用(Warningあり)

(5) 待ち行列属性(wait_queue)

【説明】待ち行列属性を定義します。
【定義形式】シンボル
【定義範囲】以下のいずれかから選択してください。

- TA_TFIFO：待ち行列は FIFO 順
- TA_TPRI：待ち行列はタスク優先度順

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はTA_TFIFO)を適用(Warningあり)

(6) 複数待ちの許可属性(wait_multi)

【説明】複数タスク待ちの許可に関する属性を定義します。
【定義形式】シンボル
【定義範囲】以下のいずれかから選択してください。

- TA_WMUL：複数タスク待ちを許可する
- TA_WSGL：複数タスク待ちを許可しない

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はTA_WMUL)を適用(Warningあり)

(7) クリア属性(clear_attribute)

【説明】イベントフラグのクリア属性を定義します。
【定義形式】シンボル
【定義範囲】以下のいずれかから選択してください。

- YES：クリア属性を設定する
- NO：クリア属性を設定しない

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はNO)を適用(Warningあり)

14.3.15 データキュー定義(dataqueue[])

dataqueue[]は、データキューを定義(生成)する定義項目です。

dataqueue[]は、GUI コンフィギュレータの[データキューの生成]ダイアログボックスと対応します。

service_call.cre_dtq, icre_dtq が共に NO の場合は、dataqueue[]は無視されます。この場合、ワーニングメッセージが表示されます。

■形式

```
dataqueue[<ローカル ID 番号>]{            // (1)ローカル ID 番号
        name               = <設定値>;  // (2)ID 名称
        export             = <設定値>;  // (3)ID 名称のエクスポート
        buffer_size        = <設定値>;  // (4)データキューの最大データ数
        section            = <設定値>;  // (5)データキュー領域に付与するセクション名
        address            = <設定値>;  // (6)データキュー領域の先頭アドレス
        wait_queue         = <設定値>;  // (7)待ち行列属性
};
```

■内容

(1) ローカル ID 番号

【説明】ローカルID番号は1～1023の範囲でなければなりません。
ローカルID番号は省略可能です。省略した場合、コンフィギュレータはローカルID番号を自動的に割り当てます。

【定義形式】数値

【定義範囲】1～1023

【省略時の扱い】ローカルID番号を自動的に割り当てます。

(2) ID 名称(name)

【説明】データキューのID名称を定義します。指定されたID名称は、ID名称ヘッダファイルに出力されます。

【定義形式】シンボル

【定義範囲】なし

【省略時の扱い】省略した場合は、ID名称なしと扱い、ID名称ヘッダファイルにも出力しません。
ただし、ローカルID番号を省略した場合はID名称は省略できません。この場合、エラーを表示します。

(3) ID 名称のエクスポート(export)

【説明】ID名称を他CPUにエクスポートするかどうかを定義します。

【定義形式】シンボル

【定義範囲】YESまたはNO

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はNO)を適用(Warningなし)

【備考】ID名称の指定がない場合、本項目は意味を持ちません。

(4) データキューの最大データ数(buffer_size)

【説明】データキューの最大データ数を定義します。データ数0のデータキューを生成することもできます。
【定義形式】数値
【定義範囲】0～0x08000000
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は1)を適用(Warningあり)

(5) データキュー領域に付与するセクション名(section)

【説明】sectionは、コンフィギュレータにデータキュー領域を生成させる場合に記述します。
データキュー領域の指定方法として、表14.7に示す方法があります。

表14.7　データキュー領域の指定方法

データキュー領域	section	address
デフォルトデータキュー用領域から割り付ける。	記述しない	記述しない
コンフィギュレータがデータキュー領域を生成する。	記述する	記述しない
ユーザ側でデータキュー領域を確保する。	記述しない	記述する

なお、sectionとaddressの両方の記述がある場合はエラーとなります。
section指定時、実際に生成されるセクション名は、sectionで指定された文字列の先頭に'B'を付加した名称になります。リンク時には、ユーザがこのセクションを適切なアドレスに配置する必要があります。
【定義形式】シンボル
【定義範囲】なし
【省略時の扱い】表14.7参照
【備考】buffer_sizeが0の場合、本項目は意味を持ちません。

(6) データキュー領域の先頭アドレス(address)

【説明】ユーザ側で確保したデータキュー領域を使用する場合は、addressにそのデータキュー領域の先頭アドレスを定義してください。addressからTSZ_DTQ(buffer_size)バイトの領域は、ユーザ側で確保する必要があります。
前項の「(5) データキュー領域に付与するセクション名(section)」も参照してください。
【定義形式】外部参照名または数値
【定義範囲】数値の場合は、0～0xFFFFFFFFの範囲の4の倍数
【省略時の扱い】表14.7参照
【備考】buffer_sizeが0の場合、本項目は意味を持ちません。

(7) 待ち行列属性(wait_queue)

【説明】送信待ち行列属性を定義します。なお、受信待ち行列は常にFIFO順で管理されます。
【定義形式】シンボル
【定義範囲】以下のいずれかから選択してください。

- TA_TFIFO：待ち行列は FIFO 順
- TA_TPRI：待ち行列はタスク優先度順

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はTA_TFIFO)を適用(Warningあり)

14.3.16 メールボックス定義(mailbox[])

mailbox[]は、メールボックスを定義(生成)する定義項目です。

mailbox[]は、GUI コンフィギュレータの[メールボックスの生成]ダイアログボックスと対応します。

service_call.cre_mbx, icre_mbx が共に NO の場合は、mailbox[]は無視されます。この場合、ワーニングメッセージが表示されます。

■形式

```
mailbox[<ローカル ID 番号>]{                    //  (1)ローカル ID 番号
        name                = <設定値>; //  (2)ID 名称
        export              = <設定値>; //  (3)ID 名称のエクスポート
        wait_queue          = <設定値>; //  (4)待ち行列属性
        message_queue       = <設定値>; //  (5)メッセージキュー属性
        max_pri             = <設定値>; //  (6)メッセージの最大優先度
};
```

■内容

(1) ローカル ID 番号

【説明】ローカルID番号は1～1023の範囲でなければなりません。
ローカルID番号は省略可能です。省略した場合、コンフィギュレータはローカルID番号を自動的に割り当てます。
【定義形式】数値
【定義範囲】1～1023
【省略時の扱い】ローカルID番号を自動的に割り当てます。

(2) ID 名称(name)

【説明】メールボックスのID名称を定義します。指定されたID名称は、ID名称ヘッダファイルに出力されます。
【定義形式】シンボル
【定義範囲】なし
【省略時の扱い】省略した場合は、ID名称なしと扱い、ID名称ヘッダファイルにも出力しません。
ただし、ローカルID番号を省略した場合はID名称は省略できません。この場合、エラーを表示します。

(3) ID 名称のエクスポート(export)

【説明】ID名称を他CPUにエクスポートするかどうかを定義します。
【定義形式】シンボル
【定義範囲】YESまたはNO
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はNO)を適用(Warningなし)
【備考】ID名称の指定がない場合、本項目は意味を持ちません。

RENESAS

(4) 待ち行列属性(wait_queue)

【説明】待ち行列属性を定義します。
【定義形式】シンボル
【定義範囲】以下のいずれかから選択してください。

- TA_TFIFO：待ち行列は FIFO 順
- TA_TPRI：待ち行列はタスク優先度順

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はTA_TFIFO)を適用(Warningあり)

(5) メッセージキュー属性(message_queue)

【説明】メッセージのメッセージキューへのつなぎ方を定義します。
【定義形式】シンボル
【定義範囲】以下のいずれかから選択してください。

- TA_MFIFO：メッセージキューは FIFO 順
- TA_MPRI：メッセージキューはメッセージ優先度順

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はTA_MFIFO)を適用(Warningあり)

(6) メッセージの最大優先度(max_pri)

【説明】message_queueにTA_MPRIを指定した場合は、ここにメッセージの最大優先度を定義します。
メッセージの最大優先度は1～255の範囲でなければなりません。また、最大メッセージ優先度(system.message_pri)以下でなければなりません。
【定義形式】数値
【定義範囲】1～255
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は1)を適用(Warningあり)
【備考】message_queueがTA_MFIFOの場合、本項目は意味を持ちません。

14.3.17 ミューテックス定義(mutex[])

mutex[]は、ミューテックスを定義(生成)する定義項目です。

mutex[]は、GUI コンフィギュレータの[ミューテックスの生成]ダイアログボックスと対応します。

service_call.cre_mtx が NO の場合は、mutex[]は無視されます。この場合、ワーニングメッセージが表示されます。

■形式

```
mutex[<ローカル ID 番号>]{                    //  (1)ローカル ID 番号
        name                = <設定値>;  //  (2)ID 名称
        protocol            = <設定値>;  //  (3)優先度制御プロトコル
        ceil_pri            = <設定値>;  //  (4)上限優先度
};
```

■内容

(1) ローカル ID 番号

【説明】ローカルID番号は1～1023の範囲でなければなりません。
ローカルID番号は省略可能です。省略した場合、コンフィギュレータはローカルID番号を自動的に割り当てます。

【定義形式】数値

【定義範囲】1～1023

【省略時の扱い】ローカルID番号を自動的に割り当てます。

(2) ID 名称(name)

【説明】ミューテックスのID名称を定義します。指定されたID名称は、ID名称ヘッダファイルに出力されます。

【定義形式】シンボル

【定義範囲】なし

【省略時の扱い】省略した場合は、ID名称なしと扱い、ID名称ヘッダファイルにも出力しません。
ただし、ローカルID番号を省略した場合はID名称は省略できません。この場合、エラーを表示します。

(3) 優先度制御プロトコル(protocol)

【説明】優先度制御プロトコルを定義します。現バージョンでは、TA_CEILING(優先度上限プロトコル)のみを指定できます。

【定義形式】シンボル

【定義範囲】以下のいずれかから選択してください。

- TA_CEILING : 優先度上限プロトコル

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はTA_CEILING)を適用(Warningなし)

(4) 上限優先度(ceil_pri)

【説明】優先度上限プロトコルにおける上限優先度を定義します。
上限優先度は1～255の範囲でなければなりません。また、最大タスク優先度(system.priority)以下でなければなりません。

【定義形式】数値

【定義範囲】1～255

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は1)を適用(Warningあり)

14.3.18 メッセージバッファ定義(message_buffer[])

message_buffer[]は、メッセージバッファを定義(生成)する定義項目です。
message_buffer[]は、GUI コンフィギュレータの[メッセージバッファの生成]ダイアログボックスと対応します。
service_call.cre_mbf, icre_mbf が共に NO の場合は、message_buffer[]は無視されます。この場合、ワーニングメッセージが表示されます。

■形式

```
message_buffer[<ローカル ID 番号>]{       //  (1)ローカル ID 番号
        name               = <設定値>; //  (2)ID 名称
        export             = <設定値>; //  (3)ID 名称のエクスポート
        buffer_size        = <設定値>; //  (4)メッセージバッファのサイズ
        section            = <設定値>; //  (5)メッセージバッファ領域に付与するセクション名
        address            = <設定値>; //  (6)メッセージバッファ領域の先頭アドレス
        max_msgsz          = <設定値>; //  (7)最大メッセージサイズ
        wait_queue         = <設定値>; //  (8)待ち行列属性
};
```

■内容

(1) ローカル ID 番号

【説明】ローカルID番号は1～1023の範囲でなければなりません。
ローカルID番号は省略可能です。省略した場合、コンフィギュレータはローカルID番号を自動的に割り当てます。
【定義形式】数値
【定義範囲】1～1023
【省略時の扱い】ローカルID番号を自動的に割り当てます。

(2) ID 名称(name)

【説明】メッセージバッファのID名称を定義します。指定されたID名称は、ID名称ヘッダファイルに出力されます。
【定義形式】シンボル
【定義範囲】なし
【省略時の扱い】省略した場合は、ID名称なしと扱い、ID名称ヘッダファイルにも出力しません。
ただし、ローカルID番号を省略した場合はID名称は省略できません。この場合、エラーを表示します。

(3) ID名称のエクスポート(export)

【説明】ID名称を他CPUにエクスポートするかどうかを定義します。
【定義形式】シンボル
【定義範囲】YESまたはNO
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はNO)を適用(Warningなし)
【備考】ID名称の指定がない場合、本項目は意味を持ちません。

(4) メッセージバッファのサイズ(buffer_size)

【説明】メッセージバッファのサイズをバイト単位で指定します。指定した値は4の倍数に切り上げて扱います。
サイズ0のメッセージバッファを生成することもできます。
【定義形式】数値
【定義範囲】0, または8～0x20000000
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は32)を適用(Warningあり)

(5) メッセージバッファ領域に付与するセクション名(section)

【説明】sectionは、コンフィギュレータにメッセージバッファ領域を生成させる場合に記述します。
メッセージバッファ領域の指定方法として、表14.8に示す方法があります。

表14.8　メッセージバッファ領域の指定方法

メッセージバッファ領域	section	address
デフォルトメッセージバッファ用領域から割り付ける。	記述しない	記述しない
コンフィギュレータがメッセージバッファ領域を生成する。	記述する	記述しない
ユーザ側でメッセージバッファ領域を確保する。	記述しない	記述する

なお、sectionとaddressの両方の記述がある場合はエラーとなります。
section指定時、実際に生成されるセクション名は、sectionで指定された文字列の先頭に’B’を付加した名称になります。リンク時には、ユーザがこのセクションを適切なアドレスに配置する必要があります。
【定義形式】シンボル
【定義範囲】なし
【省略時の扱い】表14.8参照
【備考】buffer_sizeが0の場合、本項目は意味を持ちません。

(6) バッファ領域の先頭アドレス(address)

【説明】ユーザ側で確保したバッファ領域を使用する場合は、addressにそのバッファ領域の先頭アドレスを定義してください。addressからbuffer_sizeバイトの領域は、ユーザ側で確保する必要があります。
前項の「(5) メッセージバッファ領域に付与するセクション名(section)」も参照してください。
【定義形式】外部参照名または数値
【定義範囲】数値の場合は、0～0xFFFFFFFFの範囲の4の倍数
【省略時の扱い】表14.8参照
【備考】buffer_sizeが0の場合、本項目は意味を持ちません。

(7) 最大メッセージサイズ(max_msgsz)

【説明】最大メッセージサイズをバイト単位で指定します。指定した値は4の倍数に切り上げて扱います。buffer_size>0の場合は、max_msgszは(buffer_size-4)以下でなければなりません。
【定義形式】数値
【定義範囲】4～0x20000000
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は4)を適用(Warningあり)

(8) 待ち行列属性(wait_queue)

【説明】送信待ち行列属性を定義します。なお、受信待ち行列は常にFIFO順で管理されます。
【定義形式】シンボル
【定義範囲】以下のいずれかから選択してください。

- TA_TFIFO：待ち行列は FIFO 順
- TA_TPRI：待ち行列はタスク優先度順

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はTA_TFIFO)を適用(Warningあり)

14.3.19 固定長メモリプール定義(memorypool[])

memorypool[]は、固定長メモリプールを定義(生成)する定義項目です。

memorypool[]は、GUI コンフィギュレータの[固定長メモリプールの生成]ダイアログボックスと対応します。

なお、system.mpfmanage が OUT の場合、「固定長メモリブロック管理領域(T_CMPF 構造体の mpfmb)」が必要ですが、この領域は自動的に生成されます。この領域のセクション名は BC_hicfg となります。

service_call.cre_mpf, icre_mpf が共に NO の場合は、memorypool[]は無視されます。この場合、ワーニングメッセージが表示されます。なお、service_call.cre_mpf, icre_mpf は、remote_svc.num_server>0 の場合に、共に YES に補正されます。

■形式

```
memorypool[<ローカル ID 番号>]{           //  (1)ローカル ID 番号
        name                = <設定値>;  //  (2)ID 名称
        export              = <設定値>;  //  (3)ID 名称のエクスポート
        section             = <設定値>;  //  (4)プール領域に付与するセクション名
        address             = <設定値>;  //  (5)プール領域の先頭アドレス
        num_block           = <設定値>;  //  (6)メモリプールのブロック数
        siz_block           = <設定値>;  //  (7)メモリプールのブロックサイズ
        wait_queue          = <設定値>;  //  (8)待ち行列属性
};
```

■内容

(1) ローカル ID 番号

【説明】ローカルID番号は1～1022の範囲でなければなりません。
ローカルID番号は省略可能です。省略した場合、コンフィギュレータはローカルID番号を自動的に割り当てます。
【定義形式】数値
【定義範囲】1～1022（カーネル仕様の最大値は1023ですが、cfgファイルでの最大値は1022です）
【省略時の扱い】ローカルID番号を自動的に割り当てます。

(2) ID 名称(name)

【説明】固定長メモリプールのID名称を定義します。指定されたID名称は、ID名称ヘッダファイルに出力されます。
【定義形式】シンボル
【定義範囲】なし
【省略時の扱い】省略した場合は、ID名称なしと扱い、ID名称ヘッダファイルにも出力しません。
ただし、ローカルID番号を省略した場合はID名称は省略できません。この場合、エラーを表示します。

RENESAS

(3) ID 名称のエクスポート(export)

【説明】ID名称を他CPUにエクスポートするかどうかを定義します。
【定義形式】シンボル
【定義範囲】YESまたはNO
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はNO)を適用(Warningなし)
【備考】ID名称の指定がない場合、本項目は意味を持ちません。

(4) プール領域に付与するセクション名(section)

【説明】sectionは、コンフィギュレータでプール領域を生成させる場合に記述します。
　プール領域の指定方法として、表14.9に示す方法があります。

表14.9　固定長メモリプール領域の指定方法

固定長メモリプール領域	section	address
デフォルト固定長メモリプール用領域から割り付ける。	記述しない	記述しない
コンフィギュレータがプール領域を生成する。	記述する	記述しない
ユーザ側でプール領域を確保する。	記述しない	記述する

　なお、sectionとaddressの両方の記述がある場合はエラーとなります。
　section指定時、実際に生成されるセクション名は、sectionで指定された文字列の先頭に’B’を付加した名称になります。リンク時には、ユーザがこのセクションを適切なアドレスに配置する必要があります。
【定義形式】シンボル
【定義範囲】なし
【省略時の扱い】表14.9参照

(5) プール領域の先頭アドレス(address)

【説明】ユーザ側で確保したプール領域を使用する場合は、addressにそのプール領域の先頭アドレスを定義してください。addressからTSZ_MPF(num_block, siz_block)バイトの領域は、ユーザ側で確保する必要があります。
　前項の「(4) プール領域に付与するセクション名(section)」も参照してください。
【定義形式】外部参照名または数値
【定義範囲】数値の場合は、0～0xFFFFFFFFの範囲の4の倍数
【省略時の扱い】表14.9参照

(6) メモリプールのブロック数(num_block)

【説明】メモリプールのブロック数を定義します。
【定義形式】数値
【定義範囲】1～0x08000000
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は4)を適用(Warningあり)

(7) メモリプールのブロックサイズ(siz_block)

【説明】メモリプールのブロックサイズをバイト単位で指定します。指定した値は4の倍数に切り上げて扱います。
【定義形式】数値
【定義範囲】4～0x20000000
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は4)を適用(Warningあり)

(8) 待ち行列属性(wait_queue)

【説明】待ち行列属性を定義します。
【定義形式】シンボル
【定義範囲】以下のいずれかから選択してください。

- TA_TFIFO：待ち行列は FIFO 順
- TA_TPRI：待ち行列はタスク優先度順

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はTA_TFIFO)を適用(Warningあり)

RENESAS

14.3.20 可変長メモリプール定義(variable_memorypool[])

variable_memorypool[]は、可変長メモリプールを定義(生成)する定義項目です。

variable_memorypool[]は、GUI コンフィギュレータの[可変長メモリプールの生成]ダイアログボックスと対応します。

service_call.cre_mpl, icre_mpl が共に NO の場合は、variable_memorypool[]は無視されます。この場合、ワーニングメッセージが表示されます。

■形式

```
variable_memorypool[<ローカル ID 番号>]{ //  (1) ローカル ID 番号
        name                = <設定値>; //  (2)ID 名称
        export              = <設定値>; //  (3)ID 名称のエクスポート
        heap_size           = <設定値>; //  (4)メモリプールのサイズ
        wait_queue          = <設定値>; //  (5)待ち行列属性
        mpl_section         = <設定値>; //  (6)プール領域に付与するセクション名
        mpl_address         = <設定値>; //  (7)プール領域の先頭アドレス
        unfragment          = <設定値>; //  (8)断片化軽減属性
        min_blksz           = <設定値>; //  (9)最小ブロックサイズ
        num_sector          = <設定値>; //  (10)セクタ数
};
```

■内容

(1) ローカル ID 番号

【説明】ローカルID番号は1～1023の範囲でなければなりません。
ローカルID番号は省略可能です。省略した場合、コンフィギュレータはローカルID番号を自動的に割り当てます。
【定義形式】数値
【定義範囲】1～1023
【省略時の扱い】ローカルID番号を自動的に割り当てます。

(2) ID 名称(name)

【説明】可変長メモリプールのID名称を定義します。指定されたID名称は、ID名称ヘッダファイルに出力されます。
【定義形式】シンボル
【定義範囲】なし
【省略時の扱い】省略した場合は、ID名称なしと扱い、ID名称ヘッダファイルにも出力しません。
ただし、ローカルID番号を省略した場合はID名称は省略できません。この場合、エラーを表示します。

(3) ID 名称のエクスポート(export)

【説明】ID名称を他CPUにエクスポートするかどうかを定義します。
【定義形式】シンボル
【定義範囲】YESまたはNO
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はNO)を適用(Warningなし)
【備考】ID名称の指定がない場合、本項目は意味を持ちません。

(4) メモリプールのサイズ (heap_size)

【説明】メモリプールのサイズをバイト単位で定義します。指定した値は4の倍数に切り上げられます。
【定義形式】数値
【定義範囲】36～0x20000000
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は0x200)を適用(Warningあり)

(5) 待ち行列属性(wait_queue)

【説明】待ち行列属性を定義します。現バージョンでは、TA_TFIFOのみを指定できます。
【定義形式】シンボル
【定義範囲】以下のいずれかから選択してください。

- TA_TFIFO：待ち行列は FIFO 順

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はTA_TFIFO)を適用(Warningなし)

(6) プール領域に付与するセクション名(mpl_section)

【説明】mpl_sectionは、コンフィギュレータでプール領域を生成させる場合に記述します。
プール領域の指定方法として、表14.10に示す方法があります。

表14.10　可変長メモリプール領域の指定方法

可変長メモリプール領域	mpl_section	mpl_address
デフォルト可変長メモリプール用領域から割り付ける。	記述しない	記述しない
コンフィギュレータがプール領域を生成する。	記述する	記述しない
ユーザ側でプール領域を確保する。	記述しない	記述する

なお、mpl_sectionとmpl_addressの両方の記述がある場合はエラーとなります。
mpl_section指定時、実際に生成されるセクション名は、mpl_sectionで指定された文字列の先頭に’B’を付加した名称になります。リンク時には、ユーザがこのセクションを適切なアドレスに配置する必要があります。
【定義形式】シンボル
【定義範囲】なし
【省略時の扱い】表14.10参照

(7) プール領域の先頭アドレス(mpl_address)

【説明】ユーザ側で確保したプール領域を使用する場合は、mpl_addressにそのプール領域の先頭アドレスを定義してください。mpl_addressからheap_sizeバイトの領域は、ユーザ側で確保する必要があります。
前項の「(6)プール領域に付与するセクション名(mpl_section)」も参照してください。
【定義形式】外部参照名または数値
【定義範囲】数値の場合は、0～0xFFFFFFFFの範囲の4の倍数
【省略時の扱い】表14.10参照

(8) 断片化軽減属性(unfragment)

【説明】system.newmplがNEWの場合は、断片化を軽減するVTA_UNFRAGMENT属性を使用することができます。
【定義形式】シンボル
【定義範囲】以下のいずれかから選択してください。

- ON：VTA_UNFRAGMENT 属性を付与する
- OFF：VTA_UNFRAGMENT 属性を付与しない

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はOFF)を適用(Warningあり)
【備考】system.newmplがPASTの場合、本項目は意味を持ちません。

(9) 最小ブロックサイズ(min_blksz)

【説明】VTA_UNFRAGMENT属性指定時は、最小ブロックサイズの指定が必要です。指定した値は4の倍数に切り上げられます。
最小ブロックサイズは、heap_size/32以下でなければなりません。
【定義形式】数値
【定義範囲】4～0x20000000
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は4)を適用(Warningあり)
【備考】system.newmplがPASTの場合、またはunfragmentがOFFの場合、本項目は意味を持ちません。

(10)セクタ数(num_sector)

【説明】VTA_UNFRAGMENT属性指定時は、セクタ数の指定が必要です。指定値が(heap_size/(min_blksz×32))より大きい場合は、この計算結果に補正されます。
【定義形式】数値
【定義範囲】1～0x 400000
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は1)を適用(Warningあり)
【備考】system.newmplがPASTの場合、またはunfragmentがOFFの場合、本項目は意味を持ちません。

14.3.21　周期ハンドラ定義(cyclic_hand[])

cyclic_hand[]は、周期ハンドラを定義(生成)する定義項目です。

cyclic_hand[]は、GUI コンフィギュレータの[周期ハンドラの生成]ダイアログボックスと対応します。

service_call.cre_cyc, icre_cyc が共に NO の場合は、cyclic_hand []は無視されます。この場合、ワーニングメッセージが表示されます。なお、service_call.cre_mpf, icre_mpf は、system.action が YES の場合に、共に YES に補正されます。

■形式

```
cyclic_hand[<ローカル ID 番号>]{          //  (1)ローカル ID 番号
        name                = <設定値>;  //  (2)ID 名称
        export              = <設定値>;  //  (3)ID 名称のエクスポート
        interval_counter    = <設定値>;  //  (4)起動周期
        start               = <設定値>;  //  (5)周期ハンドラの動作状態
        phsatr              = <設定値>;  //  (6)起動位相の保存
        phs_counter         = <設定値>;  //  (7)起動位相
        entry_address       = <設定値>;  //  (8)ハンドラ開始アドレス
        exinf               = <設定値>;  //  (9)拡張情報
};
```

■内容

(1) ローカル ID 番号

【説明】ローカルID番号は1～14の範囲でなければなりません。

ローカルID番号は省略可能です。省略した場合、コンフィギュレータはローカルID番号を自動的に割り当てます。

【定義形式】数値

【定義範囲】1～14

【省略時の扱い】ローカルID番号を自動的に割り当てます。

(2) ID 名称(name)

【説明】周期ハンドラのID名称を定義します。指定されたID名称は、ID名称ヘッダファイルに出力されます。

【定義形式】シンボル

【定義範囲】なし

【省略時の扱い】省略した場合は、ID名称なしと扱い、ID名称ヘッダファイルにも出力しません。

ただし、ローカルID番号を省略した場合はID名称は省略できません。この場合、エラーを表示します。

(3) ID 名称のエクスポート(export)

【説明】ID名称を他CPUにエクスポートするかどうかを定義します。

【定義形式】シンボル

【定義範囲】YESまたはNO

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はNO)を適用(Warningなし)

【備考】ID名称の指定がない場合、本項目は意味を持ちません。

(4) 起動周期(interval_counter)

【説明】起動周期をミリ秒単位で定義します。
【定義範囲】1～0x7FFFFFFF
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は1)を適用(Warningあり)

(5) 周期ハンドラの動作状態(start)

【説明】周期ハンドラの動作状態に関する属性を定義します。
【定義形式】シンボル
【定義範囲】以下のいずれかから選択してください。

- ON：周期ハンドラを動作状態にする(TA_STA 属性指定あり)
- OFF：周期ハンドラを動作状態にしない(TA_STA 属性指定なし)

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はOFF)を適用(Warningあり)

(6) 起動位相の保存(phsatr)

【説明】起動位相の保存に関する属性を定義します。
【定義形式】シンボル
【定義範囲】以下のいずれかから選択してください。

- ON：起動位相を保存する(TA_PHS 属性指定あり)
- OFF：起動位相を保存しない(TA_PHS 属性指定なし)

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はOFF)を適用(Warningあり)

(7) 起動位相(phs_counter)

【説明】起動位相をミリ秒単位で定義します。起動位相は起動周期以下でなければなりません。
【定義形式】数値
【定義範囲】0～0x7FFFFFFF
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は0)を適用(Warningあり)

(8) 周期ハンドラの開始アドレス(entry_address)

【説明】周期ハンドラの実行開始アドレスを定義します。
【定義形式】外部参照名または数値
【定義範囲】数値の場合は、0～0xFFFFFFFFの範囲の偶数
【省略時の扱い】省略不可(エラー)

(9) 拡張情報(exinf)

【説明】周期ハンドラの拡張情報を定義します。
【定義形式】外部参照名または数値
【定義範囲】数値の場合は0～0xFFFFFFFF、外部参照名の場合はなし
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は0)を適用(Warningなし)

14.3.22 アラームハンドラ定義(alarm_hand[])

alarm_hand[]は、アラームハンドラを定義(生成)する定義項目です。

alarm_hand[]は、GUI コンフィギュレータの[アラームハンドラの生成]ダイアログボックスと対応します。

なお、service_call.cre_cyc, icre_alm が共に NO の場合は、alarm_hand[]は無視されます。この場合、ワーニングメッセージが表示されます。

■形式

```
alarm_hand[<ローカル ID 番号>]{              //  (1)ローカル ID 番号
        name                = <設定値>;  //  (2)ID 名称
        export              = <設定値>;  //  (3)ID 名称のエクスポート
        entry_address       = <設定値>;  //  (4)ハンドラ開始アドレス
        exinf               = <設定値>;  //  (5)拡張情報
};
```

■内容

(1) ローカル ID 番号

【説明】ローカルID番号は1～15の範囲でなければなりません。
ローカルID番号は省略可能です。省略した場合、コンフィギュレータはローカルID番号を自動的に割り当てます。
【定義形式】数値
【定義範囲】1～15
【省略時の扱い】ローカルID番号を自動的に割り当てます。

(2) ID 名称(name)

【説明】アラームハンドラのID名称を定義します。指定されたID名称は、ID名称ヘッダファイルに出力されます。
【定義形式】シンボル
【定義範囲】なし
【省略時の扱い】省略した場合は、ID名称なしと扱い、ID名称ヘッダファイルにも出力しません。ただし、ローカルID番号を省略した場合はID名称は省略できません。この場合、エラーを表示します。

(3) ID 名称のエクスポート(export)

【説明】ID名称を他CPUにエクスポートするかどうかを定義します。
【定義形式】シンボル
【定義範囲】YESまたはNO
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はNO)を適用(Warningなし)
【備考】ID名称の指定がない場合、本項目は意味を持ちません。

RENESAS

(4) アラームハンドラの開始アドレス(entry_address)

【説明】アラームハンドラの実行開始アドレスを定義します。
【定義形式】外部参照名または数値
【定義範囲】数値の場合は、0～0xFFFFFFFFの範囲の偶数
【省略時の扱い】省略不可(エラー)

(5) 拡張情報(exinf)

【説明】アラームハンドラの拡張情報を定義します。
【定義形式】外部参照名または数値
【定義範囲】数値の場合は0～0xFFFFFFFF、外部参照名の場合はなし
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は0)を適用(Warningなし)

14.3.23 オーバーランハンドラ定義(overrun_hand)

overrun_hand は、オーバーランハンドラを定義する定義項目です。

overrun_hand は、GUI コンフィギュレータの[オーバーランハンドラ]ページと対応します。

オーバーランハンドラはシステムでひとつだけ定義できます。このため、他のオブジェクト定義とは異なり、overrun_hand はひとつだけ記述できます。

service_call.def_ovr が NO の場合は、overrun_hand は無視されます。この場合、ワーニングメッセージが表示されます。

■形式

```
overrun_hand{
        entry_address      = <設定値>; //オーバーランハンドラの開始アドレス
};
```

■内容

(1) オーバーランハンドラの開始アドレス(entry_address)

【説明】オーバーランハンドラの実行開始アドレスを定義します。
【定義形式】外部参照名または数値
【定義範囲】数値の場合は、0～0xFFFFFFFFの範囲の偶数
【省略時の扱い】省略不可(エラー)

RENESAS

14.3.24 拡張サービスコールルーチン定義(extend_svc[])

extend_svc[]は、拡張サービスコールルーチンを定義する定義項目です。

extend_svc[]は、GUI コンフィギュレータの[拡張サービスコールの定義]ダイアログボックスと対応します。

service_call.def_svc, idef_svc が共に NO の場合は、extend_svc[]は無視されます。この場合、ワーニングメッセージが表示されます。

■形式

```
extend_svc[<機能コード>]{                  // (1)機能コード
        entry_address      = <設定値>; // (2)拡張サービスコールルーチンの開始アドレス
};
```

■内容

(1) 機能コード

【説明】機能コードは1～1023の範囲でなければなりません。
【定義形式】数値
【定義範囲】1～1023
【省略時の扱い】省略不可(エラー)

(2) 拡張サービスコールルーチンの開始アドレス(entry_address)

【説明】拡張サービスコールルーチンの実行開始アドレスを定義します。
【定義形式】外部参照名または数値
【定義範囲】数値の場合は、0～0xFFFFFFFFの範囲の偶数
【省略時の扱い】省略不可(エラー)

14.3.25 割込みハンドラ、CPU 例外ハンドラ定義(interrupt_vector[])

interrupt_vector[]は、割込みハンドラおよび CPU 例外ハンドラを定義する定義項目です。

ベクタ番号は省略不可で、重複してはなりません。

ベクタ番号 0～3 はリセットベクタであり、コンフィギュレータでは定義できません。また、ベクタ番号 60～63 は OS 用に予約されているため、定義できません。

また、タイマ割込み番号(clock.number)と同じベクタ番号には、ハンドラは定義できません。定義しようとした場合は、エラーが報告されます。

interrupt_vector[]は、GUI コンフィギュレータの[割込み, CPU 例外ハンドラの定義]ダイアログボックスと対応します。

■形式

```
interrupt_vector[<ベクタ番号>]{          // (1)ベクタ番号
        entry_address       = <設定値>; // (2)ハンドラの開始アドレス
        direct              = <設定値>; // (3)ダイレクト属性
        regbank             = <設定値>; // (4)レジスタバンク属性
};
```

■内容

(1) ベクタ番号

【説明】ベクタ番号を定義します。ベクタ番号は4～59または64～511の範囲でなければなりません。

【定義形式】数値

【定義範囲】4～59, 64～511

【省略時の扱い】省略不可(エラー)

(2) ハンドラの開始アドレス(entry_address)

【説明】ハンドラの実行開始アドレスを定義します。

【定義形式】外部参照名または数値

【定義範囲】数値の場合は、0～0xFFFFFFFFの範囲の偶数

【省略時の扱い】省略不可(エラー)

(3) ダイレクト属性(direct)

【説明】VTA_DIRECT属性を付与するかどうかを定義します。

system.system_IPLより高いレベルの割込みハンドラ(NMIを含む)には、必ずVTA_DIRECT属性を付与しなければなりません。

【定義形式】シンボル

【定義範囲】以下のいずれかから選択してください。

- YES：VTA_DIRECT 属性を付与する。
- NO：VTA_DIRECT 属性を付与しない。

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はYES)を適用(Warningあり)

【備考】system.vector_typeがROM_ONLY_DIRECTまたはRAM_ONLY_DIRECTの場合で、direct=NOは、ハンドラは定義されません。この場合、ワーニングメッセージが表示されます。

(4) レジスタバンク属性(regbank)

【説明】ダイレクト属性(VTA_DIRECT)の指定がない割込みハンドラに、レジスタバンク属性(VTA_REGBANK)を付与するかどうかを定義します。
レジスタバンク属性は、以下の全ての条件を満たす場合にのみ意味を持ちます。その他の場合は、意味を持たず、ワーニングも表示されません。

(a) directにOFFを指定している。

(b) system.regbankにBANKLEVELxxを指定している。

(c) 「CPU割込み仕様定義ファイル(kernel_intspec.h)」で、当該CPUに対するINTSPEC_IBNR_ADR1(CPUID#1用)またはINTSPEC_IBNR_ADR2(CPUID#2用)に、0以外を指定している（レジスタバンクをサポートしたCPUを使用する）

(d) ベクタ番号に、「CPU割込み仕様定義ファイル(kernel_intspec.h)」に定義したINTSPEC_NOBANK_VEC???に対応するベクタ番号以外を指定している。（CPU仕様上、レジスタバンクを利用可能な割込み要因のベクタ番号を指定している）

これらの条件を満たす場合、定義する割込みハンドラの割込みレベルによって、以下のように適切にVTA_REGBANKを指定しなければなりません。これらが守られない場合、割込みハンドラは正常に動作しません。

(i) system.regbankに、使用する割込みハンドラの割込みレベルに対応したBANKLEVELxxが指定されている場合
必ずregbank=YESとしてください。

(ii) system.regbankに、使用する割込みハンドラの割込みレベルに対応したBANKLEVELxxが指定されていない場合
必ずregbank=NOとしてください。

【定義形式】シンボル

【定義範囲】以下のいずれかから選択してください。

- YES：VTA_REGBANK 属性を付与する。
- NO：VTA_REGBANK 属性を付与しない。

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時はNO)を適用(Warningあり)

14.3.26 初期化ルーチン定義(init_routine[])

init_routine[]は、初期化ルーチンを登録(定義)する定義項目です。
初期化ルーチンは複数定義できます。
init_routine[]は、GUI コンフィギュレータの[初期化ルーチンの登録]ダイアログボックスと対応します。
初期化ルーチンは、カーネル起動時に cfg ファイルでの記述順に実行されます。

■形式

```
init_routine[] {
        entry_address       = <設定値>; // (1)初期化ルーチンの開始アドレス
        exinf               = <設定値>; //  (2)拡張情報
};
```

■内容

(1) 初期化ルーチンの開始アドレス(entry_address)

【説明】初期化ルーチンの実行開始アドレスを定義します。
【定義形式】外部参照名または数値
【定義範囲】数値の場合は、0～0xFFFFFFFFの範囲の偶数
【省略時の扱い】省略不可(エラー)

(2) 拡張情報(exinf)

【説明】初期化ルーチンの拡張情報を定義します。初期化ルーチンは、パラメータとしてここで指定した拡張情報を受け取ります。
【定義形式】外部参照名または数値
【定義範囲】数値の場合は0～0xFFFFFFFF、外部参照名の場合はなし
【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は0)を適用(Warningなし)

14.3.27 サービスコール定義(service_call)

service_call は、ターゲットに組み込むサービスコールを定義する定義項目です。
service_call は、GUI コンフィギュレータの[サービスコール選択]ページと対応します。

■形式

```
service_call {
    <サービスコール名>= <設定値>;
        …
};
```

なお、基本的に各カーネルオブジェクトを使用するには、対応する cre_xxx や def_xxx サービスコールを組み込む必要があります。この詳細を、表 14.11に示します。

表14.11 オブジェクトの使用に必要なサービスコール

オブジェクト	必要なサービスコール	左記サービスコールが組み込まれない場合の扱い
タスク	−	常に使用可能です。
タスク例外処理ルーチン	def_tex または idef_tex	全ての task[].tex_address および task[].tex_fpu は無視されます。
セマフォ	cre_sem または icre_sem	全ての semaphore[]定義は無視され、maxdefine.max_sem は 0 の扱いとなります。
イベントフラグ	cre_flg または icre_flg	全ての flag[]定義は無視され、maxdefine.max_flag は 0 の扱いとなります。
データキュー	cre_dtq または icre_dtq	全ての dataqueue[]定義は無視され、maxdefine.max_dtq は 0 の扱いとなります。また、デフォルトデータキュー用領域も生成されません。
メールボックス	cre_mbx または icre_mbx	全ての mailbox[]定義は無視され、maxdefine.max_mbx は 0 の扱いとなります。
ミューテックス	cre_mtx	全ての mutex[]定義は無視され、maxdefine.max_mtx は 0 の扱いとなります。
メッセージバッファ	cre_mbf または icre_mbf	全ての message_buffer[]定義は無視され、maxdefine.max_mbf は 0 の扱いとなります。また、デフォルトメッセージバッファ用領域も生成されません。
固定長メモリプール	cre_mpf または icre_mpf	全ての memorypool[]定義は無視され、maxdefine.max_mpf は 0 の扱いとなります。また、デフォルト固定長メモリプール用領域も生成されません。
可変長メモリプール	cre_mpl または icre_mpl	全ての variable_memorypool[]定義は無視され、maxdefine.max_mpl は 0 の扱いとなります。また、デフォルト可変長メモリプール用領域も生成されません。
周期ハンドラ	cre_cyc または icre_cyc	全ての cyclic_hand[]定義は無視され、maxdefine.max_cyh は 0 の扱いとなります。
アラームハンドラ	cre_alm または icre_alm	全ての alarm_hand[]定義は無視され、maxdefine.max_alh は 0 の扱いとなります。
オーバーランハンドラ	def_ovr	overrun_hand 定義は無視されます。
拡張サービスコール	def_svc または idef_svc	全ての extend_svc[]定義は無視され、maxdefine.max_fncd は 0 の扱いとなります。
割込みハンドラ、CPU 例外ハンドラ	−	常に使用可能です。

また、一部のサービスコールについては補正が行われます。この詳細を表 14.12に示します。補正が行われた場合は、ワーニングメッセージが表示されます。

表14.12　補正されるサービスコール

サービスコール	補正条件
cre_tsk, icre_tsk, ext_tsk, slp_tsk, wup_tsk, iwup_tsk, dis_dsp, ena_dsp, sns_dpn, vsta_knl, ivsta_knl, vini_rmt, vsys_dwn,ivsys_dwn	常に YES に補正されます。 *1
vscr_tsk, ivscr_tsk	maxdefine.max_statictask>0 の場合は YES に､そうでない場合は NO に補正されます。 *1
def_inh, idef_inh, def_exc, idef_exc, vdef_trp, ivdef_trp	system.vector_type が ROM または ROM_ONLY_DIRECT の場合は NO に、そうでない場合は YES に補正されます。 *1
cre_cyc, icre_cyc	system.action が YES の場合は YES に補正されます。
acre_tsk、iacre_tsk, del_tsk, ter_tsk	remote_svc.num_server>0 の場合は、YES に補正されます。
cre_mpf, icre_mpf, acre_mpf,iacre_mpf, del_mpf, pget_mpf, rel_mpf,	remote_svc.num_wait>0 の場合は、YES に補正されます。

【注】 *1　結果的に、ユーザ指定の cfg ファイルおよびデフォルト cfg ファイル中の指定は無視されます。

また、clock.timer が NOTIMER の場合は、dly_tsk や cre_alm など、タイマを必要とするサービスコールは自動的に NO に補正されます。ただし、この補正に関するワーニングメッセージは表示されません。

■内容

(1) 設定値

【説明】当該サービスコールを組み込むかどうかを定義します。
なお、組み込まなかったサービスコールが呼び出された場合は、E_NOSPTエラーが返されます。

【定義形式】シンボル

【定義範囲】以下のいずれかから選択してください。

- YES：サービスコールを組み込む
- NO：サービスコールを組み込まない

【省略時の扱い】デフォルトcfgファイルの設定値(出荷時は全てNO)を適用(Warningなし)

14.4 コンフィギュレータの実行

14.4.1 コンフィギュレータ概要

コンフィギュレータの動作概要を図 14.1に示します。

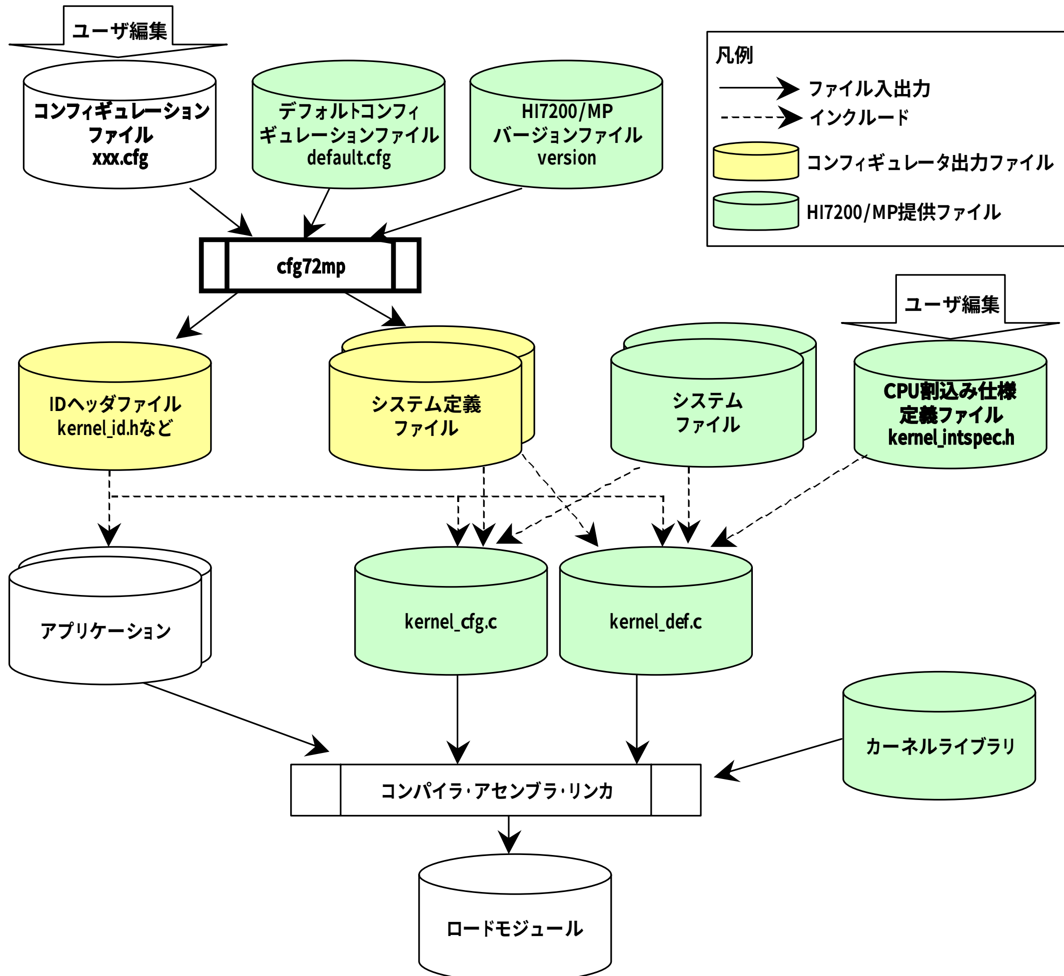


図14.1 コンフィギュレータ動作概要

CPU 割込み仕様定義ファイルについては、「17.3 CPU 割込み仕様定義ファイル(kernel_intspec.h)の作成」を参照してください。

RENESAS

14.4.2 環境設定

以下の環境変数の設定が必要です。

提供するサンプル High-performance Embedded Workshop ワークスペースでは、cfg72mp をカスタムビルドフェーズとして登録しており、この中で以下の設定も行っています。

- LIB72MP
  default.cfg, versionファイルの格納パス

14.4.3 コンフィギュレータ実行に必要な入力ファイル

◆ cfg ファイル(XXXX.cfg)
　ユーザが作成するシステムの初期設定項目を記述したファイルです。

◆ デフォルト cfg ファイル(default.cfg)
　多くの定義項目では、cfgファイルで省略された場合に、このファイルに記述されている設定値を適用します。デフォルトcfgファイルは、環境変数"LIB72MP"で示されるディレクトリに置きます。

◆ バージョンファイル(version)
　HI7200/MPのバージョンを記述したファイルです。バージョンファイルは、環境変数"LIB72MP"で示されるディレクトリに置きます。cfg72mpはこのファイルを読み込み、起動メッセージにHI7200/MPのバージョン情報を出力します。

14.4.4 cfg72mp が出力するファイル

cfg72mp は、カレントディレクトリに以下のファイルを出力します。なお、出力ディレクトリを指定することはできません。

(1) 自 CPUID 定義ファイル(mycpuid.h)

コンフィギュレーションしたカーネルを実行させる CPUID(system.cpuid)を定義したファイルです。本ファイルは、kernel.h からインクルードされます。以下に、system.cpuid が 1 の場合の出力例を示します。

```
#define MYCPUID 1U
```

(2) ID 名称ヘッダファイル

オブジェクトの ID 番号を定義したファイルです。必要に応じて、アプリケーションからインクルードしてください。詳細については「14.6 ID 名称ヘッダファイル」を参照してください。

ID 名称ヘッダファイルでは、kernel.h をインクルードしています。

(3) kernel_macro.h

本ファイルは、kernel.h からインクルードされます。出力内容については「14.7 kernel_macro.h」を参照してください。

(4) システム定義ファイル

システム定義ファイルは、kernel_def.c および kernel_cfg.c からインクルードされ、コンフィギュレーション結果に応じたシステムを生成するために使用されます。本ファイルの内容は、実装依存です。今後のバージョンでも互換性は保証されません。アプリケーションでは、これらのファイルをインクルードしてはなりません。

以下に、cfg72mp が出力するシステム定義ファイルを示します。

(1) kernel_cfg.h
(2) kernel_cfg_area.h
(3) kernel_cfg_extern.h
(4) kernel_cfg_inireg.h
(5) kernel_cfg_inirtn.h
(6) kernel_cfg_ststk.h
(7) kernel_def.h
(8) kernel_def_area.h
(9) kernel_def_extern.h
(10) kernel_def_inireg.h
(11) kernel_def_inirtn.h

14.4.5 コンフィギュレータ起動方法

コンフィギュレータは、以下の形式で起動します。

```
C> cfg72mp[-vV] cfg ファイル名
```

cfg ファイル名の拡張子を省略した場合は、拡張子".cfg"を補って解釈します。

14.4.6 コマンドオプション

(1) -v オプション

コマンドのオプションの説明と詳細なバージョンを表示します。

(2) -V オプション

コマンドが生成するファイルの作成状況を表示します。

14.4.7 注意事項

cfg72mp を実行した場合は、kernel.h をインクルードしているファイルの再コンパイルが必要です。これは、cfg72mp が kernel.h からインクルードされる kernel_macro.h を出力するためです。

14.5 エラーメッセージ

14.5.1 エラー形式とエラーレベル

本節では、以下の形式で出力するエラーメッセージとエラー内容を説明します。

エラー番号 (エラーレベル) エラーメッセージ

エラーレベルは、表 14.13に示すように分類されます。

表14.13 エラーレベル

エラーレベル		動作
(W)	ワーニング	処理を継続します。
(E)	エラー	処理を中断します。

14.5.2 メッセージ一覧

(1) エラーメッセージ

0001 (E) Illegal option --> <*文字*>
コマンドオプションに誤りがあります。

0002 (E) Ilegal argument --> <*文字列*>
起動形式に誤りがあります。

0003 (E) Invalid option
コマンドオプション・起動形式に誤りがあります。

0100 (E) syntax error
文法エラー

1000 (E) Not enough memory
メモリが足りません。

2000 (E) Can't write open <*ファイル名*>
ファイルを作成できません。ディレクトリの属性やディスクの残り容量を確認してください。

2002 (E) Can't open version file
カレントディレクトリまたは環境変数"LIB72MP"の示すディレクトリの下にバージョンファイル"version"がありません。

2003 (E) can't open default configuration file
カレントディレクトリまたは環境変数"LIB72MP"の示すディレクトリの下にデフォルトcfgファイル(default.cfg)がありません。

2004 (E) Can't open configuration file <*ファイル名*>
指定されたcfgファイルにアクセスできません。

3000 (E) Zero divide error
cfgファイルにゼロ除算が記述されています。

3001 (E) Illegal *XXXX* --> <*設定値*>
定義名XXXXに対する設定値が不正です。

3002 (E) Illegal number expression --> <*文字列*>
指定された文字列は数値に変換できません。

3003 (E) Unknown token --> <*文字列*>
指定された文字列は定義名として認識できません。

3003 (E) Unknown *XXXX* --> <*文字列*>
定義名XXXXの設定値として、規定外の文字列が指定されました。

3004 (E) Number of tasks exceeds upper limit(1023)
タスクの数が上限値(1023)を越えます。

3005 (E) Illegal number of *XXXX* --> <*値*>
XXXX[]の定義数が多すぎます。

4000 (E) *XXXX* not defined
その定義名XXXXが定義されていません。

4002 (E) *XXXX*[].*YYYY* not defined
XXXX[].YYYYが定義されていません。

4003 (E) When "name" is omitted, "ID" cannot be omitted.
"name"を省略した場合は、ローカルID番号の指定が必須です。

4200 (E) Double definition <*XXXX*>
定義名XXXXが多重定義されています。

4201 (E) Double definition *xxxx*[*番号*]
そのオブジェクト定義項目は多重定義されています。

4202 (E) System timer's vector <*ベクタ番号*> conflict
interrupt_vector[]で指定されたベクタ番号が、clock.numberと重複しています。

4203 (E) Double definition taskid=*ID 番号* in static_stack[*ｽﾀｯｸ番号*] and [*ｽﾀｯｸ番号*]
複数のstatic_stack[].tskidに、同じタスクが指定されています。

4300 (E) The ID of task[] with name=*ID 名称* does not use static stack
static_stack[].tskidに、maxdefine.max_statictaskより大きいローカルタスクID番号を持つtask[]、またはローカルID番号を省略したtask[]のnameが指定されています。スタティックスタックを使用するタスクをID名称で指定する場合は、task[]のローカルタスクID番号が省略されておらず、かつmaxdefine.max_statictask以下でなければなりません。

4301 (E) The task ID=*ID 番号* does not use static stack
static_stack[].tskidに、maxdefine.max_statictaskより大きいローカルタスクID番号が指定されています。スタティックスタックを使用するタスクをローカルタスクID番号で指定する場合は、maxdefine.max_statictask以下でなければなりません。

4302 (E) The task[] with name=*ID 名称* is not defined
static_stack[].tskidに指定されたID名称を持つtask[]定義が見つかりません。

4303 (E) Static stack for tskid=*ID 番号* is not assigned
そのID番号のタスクが使用するスタティックスタックが割り当てられていません。いずれかのstatic_stack[].tskidに、そのローカルタスクIDを追加してください。

4400 (E) *YYYY* must set *ZZZZ* or less in *XXXX* definition
XXXX.YYYYには、ZZZZ以下を設定しなければなりません。

4401 (E) *YYYY* must set *ZZZZ* or more in *XXXX* definition
XXXX.YYYYには、ZZZZ以上を設定しなければなりません。

4402 (E) Can't define both *XXXX* keyword and *YYYY* keyword in *ZZZZ* definition
定義項目ZZZZにおいて、XXXXとYYYYを同時に指定することはできません。

4403 (E) *XXXX* exceeds 512MB
XXXXが512MBを超えます。

4404 (E) Total of required default *XXXX* size exceeds 512MB
必要なデフォルトXXXX用領域サイズが512MBを超えます。

4406 (E) Too big task[*ID 番号*]'s priority --> <*優先度*>
指定されたtask[ID番号].priorityが、system.priorityを超えています。

4407 (E) Too big IPL --> <*clock.IPL 設定値* >
clock.IPLが、system.system_IPLを超えています。

4408 (E) Either system.tic_deno or system.tic_nume must be 1.
system.tic_numeとsystem.tic_denoの少なくとも一方は1でなければなりません。

(2) ワーニングーメッセージ

8000 (W) *XXXX* is not defined.
定義名XXXXが省略されたため、デフォルトcfgファイルの設定値を適用します。

8100 (W) Already definition *XXXX*
定義名XXXXは既に定義済みです。最初の定義を有効とします。

8101 (W) *XXXX[番号]* definition is ignored
対応するサービスコールが選択されていないため、そのオブジェクト定義は無視されました。

8200 (W) *XXXX* is corrected to *YYYY*
定義名XXXXの設定が、YYYYに補正されました。

8201 (W) *XXXX* is not multiple of 4 --> <*YYYY*>
定義名XXXXの設定値を、YYYYに切り上げました。

RENESAS

14.6 ID 名称ヘッダファイル

14.6.1 概要

ID 名称ヘッダファイルには、各オブジェクトの ID 名称が以下の形式で出力されます。

```
#define <ID 名称>  MAKE_ID(<CPUID>, <ID 番号>)
```

<CPUID>は、system.cpuid に指定した CPUID です。
<ID 名称>は、ユーザが指定した ID 名称です。

MAKE_ID()は、CPUID とローカル ID から ID 番号を生成するマクロで、kernel.h で定義されています。ID 名称ヘッダファイルは kernel.h をインクルードしています。

14.6.2 ID 名称ヘッダファイルの種類

(1) kernel_id.h

cfg ファイルで指定した ID 名称を使用するには、kernel_id.h をインクルードしてください。
kernel_id.h 内では、指定された ID 名称のうち、他 CPU にエクスポートしない ID 名称(xxx[].export が NO)の定義がなされており、さらに他 CPU にエクスポートする ID 名称を定義している下記のファイル(kernel_id_cpu1.h または kernel_id_cpu2.h)をインクルードしています。

(2) kernel_id_cpu1.h および kernel_id_cpu2.h

これらのファイルには、他 CPU にエクスポートする ID 名称(xxx[].export が YES)の定義が出力されます。
cfg72mp は、system.cpuid が 1 の場合は kernel_id_cpu1.h を出力し、syste.cpuid が 2 の場合は kernel_id_cpu2.h を出力します。

(3) kernel_id_sys.h, kernel_id_sys_cpu1.h, kernel_id_sys_cpu2.h

これらのファイルは将来拡張用です。通常は、これらのファイルには何も定義文は出力されません。

14.7 kernel_macro.h

本ファイルは、kernel.h からインクルードされます。

```
#define TMAX_TPRI  10
#define TMAX_MPRI 10
#define TIC_NUME 1UL
#define TIC_DENO 1UL
#define TIM_LVL 13UL
#define TIM_INHNO 64UL
#define VTCFG_TBR _FOR_SVC
#define VTCFG_MPFMANAGE _OUT
#define VTCFG_NEWMPL _NEW
#define VTCFG_VECTYPE _ROM
#define VTCFG_REGBANK (_BANKLEVEL01|_BANKLEVEL14|_BANKLEVEL15)
```

(1) TMAX_TPRI

最大タスク優先度(system.priority)です。

(2) TMAX_MPRI

最大メッセージ優先度(system.message_pri)です。

(3) TIC_NUME, TIC_DENO

それぞれ、タイムティックの分子(clock_tic_nume)と分母(clock.tic_deno)です。clock.timer が NOTIMER の場合は、意味を持ちません。

(4) TIM_LVL

タイマ割込みレベル(clock.IPL)です。clock.timer が NOTIMER の場合は 0 となります。

タイマドライバを作成する場合は、TIC_NUME, TIC_DENO および TIM_LVL を元にタイマの初期化を実装してください。

(5) VTCFG_TBR

TBR レジスタの利用形態(system.tbr)です。system.tbr で指定したシンボルの先頭に'_'を付加したシンボルに定義されます。これらのシンボルの定義値は、以下の通りです。

```
#define _NOMANAGE        0UL
#define _FOR_SVC         1UL
#define _TASK_CONTEXT    2UL
```

(6) VTCFG_MPFMANAGE

固定長メモリプールの管理方式(system.mpfmanage)です。system.mpfmanage で指定したシンボルの先頭に'_'を付加したシンボルに定義されます。これらのシンボルの定義値は、以下の通りです。

```
#define _IN             0UL
#define _OUT            1UL
```

(7) VTCFG_NEWMPL

可変長メモリプールの管理方式(system.newmpl)です。system.nemwpl で指定したシンボルの先頭に'_'を付加したシンボルに定義されます。これらのシンボルの定義値は、以下の通りです。

```
#define _PAST           0UL
#define _NEW            1UL
```

(8) VTCFG_VECTYPE

割込みベクタのタイプ(system.vector_type)です。system.vector_type で指定したシンボルの先頭に'_'を付加したシンボルに定義されます。これらのシンボルの定義値は、以下の通りです。

```
#define _ROM_ONLY_DIRECT        0UL
#define _RAM_ONLY_DIRECT        1UL
#define _ROM                    2UL
#define _RAM                    3UL
```

(9) VTCFG_REGBANK

レジスタバンクの使用方法(system.regbank)です。system.regbank で指定したシンボルの先頭に'_'を付加したシンボルに定義されます。これらのシンボルの定義値は、以下の通りです。

```
#define _NOTUSE         0UL
#define _ALL            0x40000000UL
#define _BANKLEVEL01    0x00000002UL
#define _BANKLEVEL02    0x00000004UL
#define _BANKLEVEL03    0x00000008UL
#define _BANKLEVEL04    0x00000010UL
#define _BANKLEVEL05    0x00000020UL
#define _BANKLEVEL06    0x00000040UL
#define _BANKLEVEL07    0x00000080UL
#define _BANKLEVEL08    0x00000100UL
#define _BANKLEVEL09    0x00000200UL
#define _BANKLEVEL10    0x00000400UL
#define _BANKLEVEL11    0x00000800UL
#define _BANKLEVEL12    0x00001000UL
#define _BANKLEVEL13    0x00002000UL
#define _BANKLEVEL14    0x00004000UL
#define _BANKLEVEL15    0x00008000UL
```

(10)VTCFG_TRACE

サービスコールトレース(system.trace)です。systemtrace で指定したシンボルの先頭に'_'を付加したシンボルに定義されます。これらのシンボルの定義値は、以下の通りです。

```
#define _NO                 0UL
#define _TARGET_TRACE       1UL
#define _TOOL_TRACE         2UL
```

(11)VTCFG_TRACE_OBJECT

サービスコールトレースのオブジェクト取得数(system.trace_object)が出力されます。system.trace が NO の場合は、0 が出力されます。

RENESAS

15.　GUI コンフィギュレータ

GUI コンフィギュレータは、GUI 画面上で各種カーネルコンフィギュレーション情報を入力することで、cfg ファイルを生成するツールです。cfg ファイルは、cfg72mp に入力します。

GUI コンフィギュレータを使用すれば、cfg ファイルの記法を習得しなくてもカーネルを構築することができます。

図 15.1に、GUI コンフィギュレータと cfg ファイルおよび cfg72mp の関係を示します。

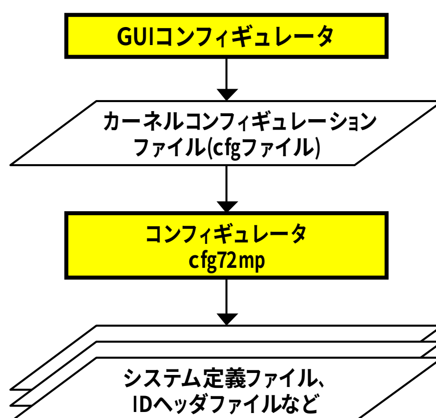


図15.1　GUI コンフィギュレータと cfg ファイルおよび cfg72mp の関係

GUI コンフィギュレータの使用方法については、オンラインヘルプを参照してください。

16. サンプルプログラム

本章では、<SAMPLE_INST>¥R0K572650D000BR ディレクトリのサンプルプログラムについて解説します。

本サンプルプログラムは、主に OS 機能の振る舞いを理解することを目的に作られています。また、E10A-USB などのエミュレータを用いて動作確認することを前提に作られています。外部との入出力は何も行っていません。

16.1 対象ハードウェア

本サンプルプログラムは、SH7265 を搭載した弊社製評価ボード「R0K572650D00BR」用に作られています。

表 16.1に、このボードの仕様概要を、図 16.1に R0K572650D00BR での SH7265 メモリマッピングを示します。

表16.1 R0K572650D00BR の仕様概要

項目	
搭載マイコン	• SH7265 (R5S72653P200BG) - 入力クロック(XIN)：16.67MHz - バスクロック：最大 66.67MHz - CPU クロック：最大 200MHz
外部メモリ	• NOR フラッシュメモリ(CS0 空間, 16 ビットバス幅)：16M バイト - SPANSION 製 S29GL128M90TFIR2 × 1 個 • SDRAM(SDRAM0 空間, 16 ビットバス幅)：32M バイト - エルピーダ製 EDS2516APTA-75 × 1 個

アドレス	SH7265 論理空間	アドレス	R0K572650D00BR メモリマッピング
0x00000000	CS0 空間 (64MB)	0x00000000 0x00FFFFFF	フラッシュメモリ(16MB) 16 ビットバス
			ユーザ領域
0x04000000	CS1 空間 (64MB)	0x04000000	ユーザ領域
0x08000000	CS2 空間 (64MB)	0x08000000	ユーザ領域
0x0C000000	CS3 空間 (64MB)	0x0C000000	ユーザ領域
0x10000000	CS4 空間 (64MB)	0x10000000	ユーザ領域
0x14000000	CS5 空間 (64MB)	0x14000000	ユーザ領域
0x18000000	SDRAM0 空間 (64MB)	0x18000000 0x19FFFFFF	SDRAM(32MB)
0c1C000000	SDRAM1 空間 (64MB)	0x1C000000 0x1DFFFFFF	SDRAM(32MB) 未実装(パターンのみ)
0x20000000	CS0～CS5, SDRAM0,1 空間 (キャッシュ無効空間)	0x20000000	CS0～CS5, SDRAM0,1 空間 (キャッシュ無効空間)
0x40000000	予約領域(使用禁止)	0x40000000	予約領域(使用禁止)
0xE8000000	内蔵周辺モジュール	0xE8000000	内蔵周辺モジュール
0xEC000000	予約領域(使用禁止)	0xEC000000	予約領域(使用禁止)
0xFF400000	内蔵周辺モジュール	0xFF400000	内蔵周辺モジュール
0xFFC00000	予約領域(使用禁止)	0xFFC00000	予約領域(使用禁止)
0xFFD80000	高速内蔵 RAM0(シャドー) (64KB)	0xFFD80000	高速内蔵 RAM0(シャドー) (64KB)
0xFFD90000	予約領域(使用禁止)	0xFFD90000	予約領域(使用禁止)
0xFFDA0000	高速内蔵 RAM1(シャドー) (32kB)	0xFFDA0000	高速内蔵 RAM1(シャドー) (32kB)
0xFFDA8000	予約領域(使用禁止)	0xFFDA8000	予約領域(使用禁止)
0xFFF80000	高速内蔵 RAM0 (64KB)	0xFFF80000	高速内蔵 RAM0 (64KB)
0xFFF90000	予約領域(使用禁止)	0xFFF90000	予約領域(使用禁止)
0xFFFA0000	高速内蔵 RAM1 (32kB)	0xFFFA0000	高速内蔵 RAM1 (32kB)
0xFFFA8000	予約領域(使用禁止)	0xFFFA8000	予約領域(使用禁止)
0xFFFC0000 0xFFFFFFFF	内蔵周辺モジュール	0xFFFC0000 0xFFFFFFFF	内蔵周辺モジュール

図16.1　SH7265 メモリマッピング

RENESAS

16.2 ディレクトリ構成

<SAMPLE_INST>¥R0K572650D000BR 以下のディレクトリ構成を以下に示します。

include¥	両 CPU に共通のヘッダファイル
iodefine¥	ハードウェア定義ヘッダファイル，周辺クロック周波数定義, kernel_intspec.h
cpuid1¥	CPUID#1 用のワークスペースディレクトリ
include¥	CPUID#1 のサンプル共通のヘッダファイル
cfg_out¥	cfg ファイル、コンフィギュレータ出力ファイル、kernel_def.c、kernel_cfg.c
ipi¥	IPI
oal¥	OAL
rpc_config¥	RPC データファイル(rpc_table.c)
reset¥	リセット処理
os_timer¥	SH7205, SH7265 内蔵 CMT 用タイマドライバ
init_task¥	初期起動タスク
stdlib¥	標準ライブラリの初期化関数、低水準関数
sysdwn¥	システムダウンルーチン
dummy_prog¥	各種ダミープログラム
rpc_sample_clnt¥	RPC クライアントスタブの例
rpc_caller¥	RPC コールの例
remote_svc_sample¥	リモートサービスコールの使用例
prj_cpuid1¥	プロジェクトディレクトリ
debug¥	“debug”コンフィギュレーションディレクトリ
cpuid2¥	CPUID#2 用のワークスペースディレクトリ
include¥	CPUID#2 のサンプル共通のヘッダファイル
cfg_out¥	cfg ファイル、コンフィギュレータ出力ファイル、kernel_def.c、kernel_cfg.c
ipi¥	IPI
oal¥	OAL
rpc_config¥	RPC データファイル(rpc_table.c)
reset¥	リセット処理
os_timer¥	SH7205, SH7265 内蔵 CMT 用タイマドライバ
init_task¥	初期起動タスク
stdlib¥	標準ライブラリの初期化関数、低水準関数
sysdwn¥	システムダウンルーチン
dummy_prog¥	各種ダミープログラム
rpc_sample_svr¥	RPC サーバスタブおよび RPC サーバ関数の例
remote_svc_sample¥	リモートサービスコールの使用例
prj_cpuid2¥	プロジェクトディレクトリ
debug¥	“debug”コンフィギュレーションディレクトリ

なお、以下のファイルはサンプルプログラムではありませんが、便宜上このディレクトリに格納しています。

(1) kernel_def.c, kernel_cfg.c
(2) IPI
(3) OAL
(4) RPCデータファイル(rpc_table.c)

16.3 スタートアップ

本節では、本サンプルにおける、リセットから各 CPU がマルチタスク環境へ移行するまでの処理手順を解説します。アプリケーションシステム開発の初期段階では、スタートアップの確認に時間を浪費してしまうケースも多いので、本節およびサンプルコードをよく理解することを推奨します。

16.3.1 概要

図 16.2に、リセットから各 CPU がマルチタスク環境へ移行するまでの概略フローを示します。

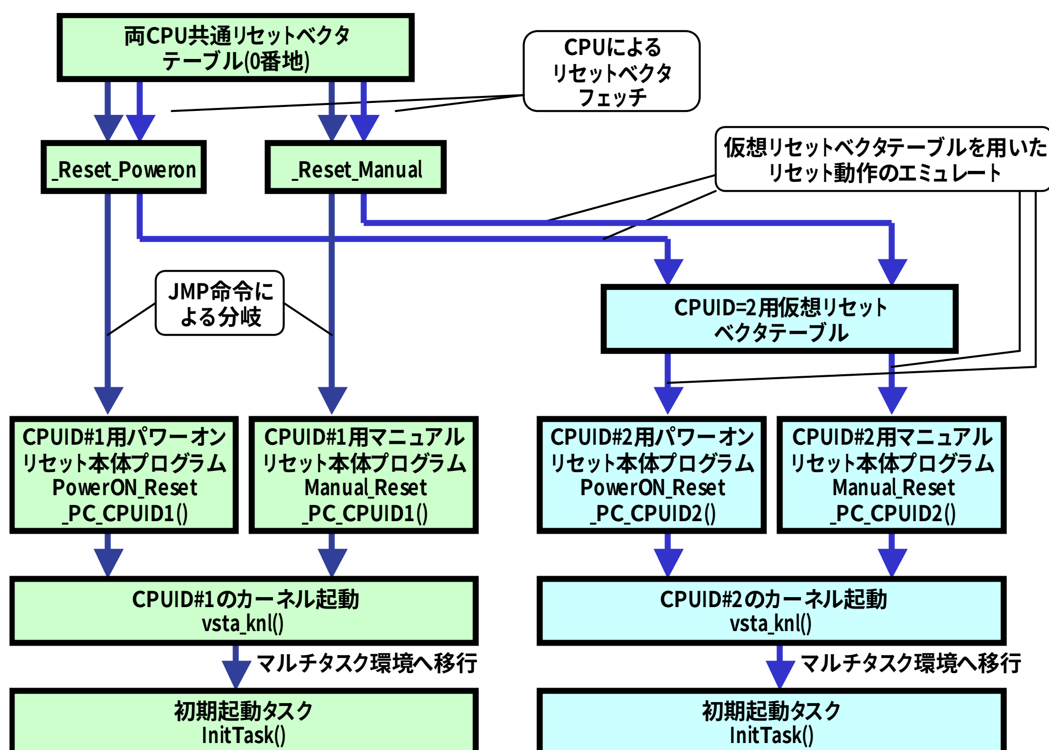


図16.2 スタートアップ概略フロー

図 16.2の凡例を、以下に示します。

- 赤矢印：CPUID#1 の動作
- 青矢印：CPUID#2 の動作
- 緑背景：CPUID#1 側のリンク単位
- 水色背景：CPUID#2 側のリンク単位

RENESAS

なお、各 CPU の初期化処理には順序性が必要なため、いくつかの初期化フラグを用いてこれを制御しています。詳細は「16.3.9 両 CPU のスタートアップフェーズの同期メカニズム」を参照してください。

表 16.2に、本節で解説するソースファイルを示します。

表16.2 スタートアップ関連のソースファイル

ディレクトリ	ファイル名	関数等
cpuid1¥reset	reset.src	(1)リセットベクタテーブル(0 番地) (2)_Reset_Poweron (3)_Reset_Manual
	resetprg1.c	CPUID#1 用リセット本体処理 (1) PowerON_Reset_PC_CPUID1() (2) Manual_Reset_PC_CPUID1()
	hwsetup1.c	共通ハードウェアと CPUID#1 固有ハードウェアの初期化 HardwareSetup_CPUID1()
	cpg1.c	CPG(FRQCR0)の初期化　io_set_cpg_couid1()
	uram.c	内蔵 RAM の初期設定　io_set_uram()
	bsc_cs0.c	CS0 の初期化　io_init_bsc_cs0()
	bscsdram.c	SDRAM 空間の初期化　io_init_sdram()
cpuid1¥init_task¥	init_task1.c	CPUID#1 用の初期起動タスク InitTask1()
cpuid2¥reset¥	reset¥vreset.src	CPUID#2 用の仮想リセットベクタテーブル
	reset¥resetprg2.c	CPUID#2 用リセット本体処理 (1) PowerON_Reset_PC_CPUID2() (2) Manual_Reset_PC_CPUID2()
	reset¥hwsetup2.c	CPUID#2 固有ハードウェアの初期化 HardwareSetup_CPUID2()
	reset¥cpg2.c	CPG(FRQCR2)の初期化　io_set_cpg_cpuid2()
cpuid2¥init_task¥	init_task¥init_task2.c	CPUID#2 用の初期起動タスク InitTask2()

図 16.3に CPUID#1 のスタートアップフローを、図 16.4に CPUID#2 のスタートアップフローを示します。図中、黄色で示した部分は、HI7200/MP が提供する API コールです。

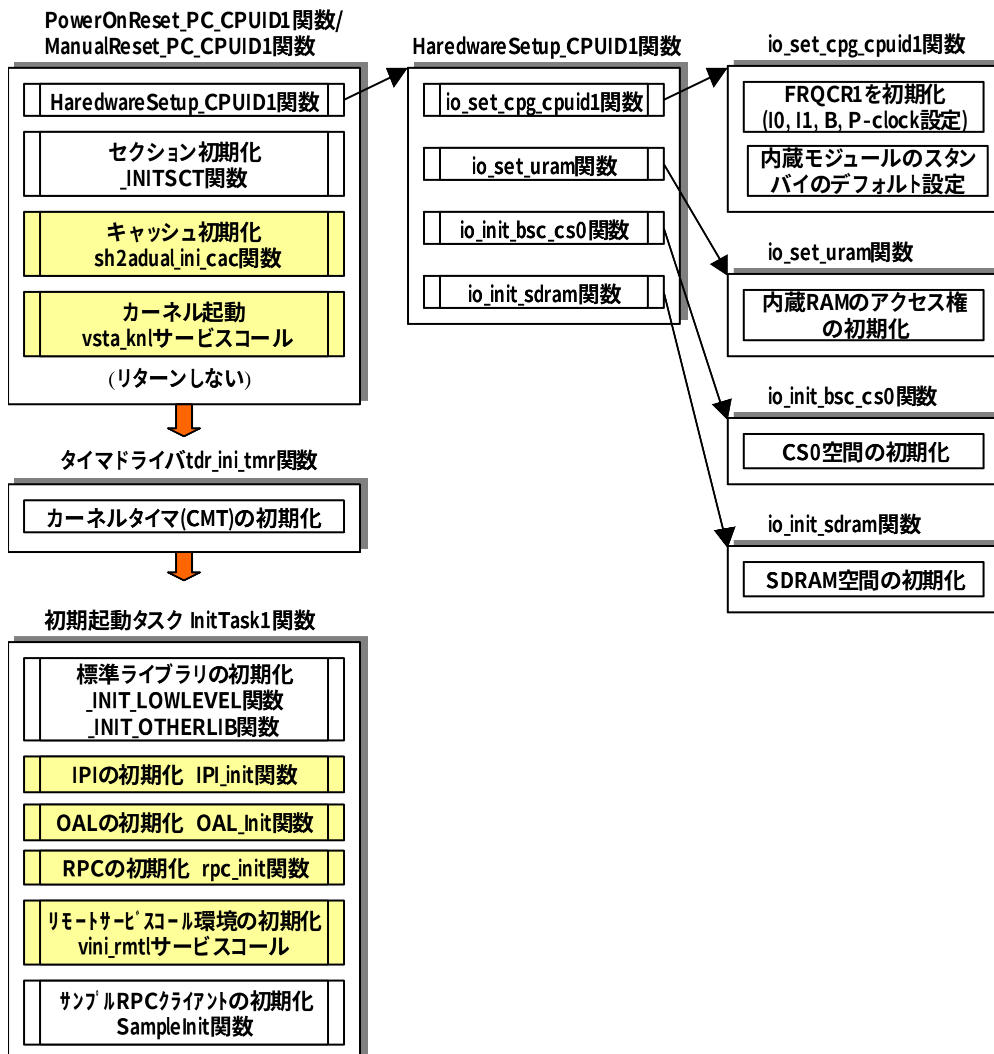


図16.3 CPUID#1 のスタートアップフロー

RENESAS

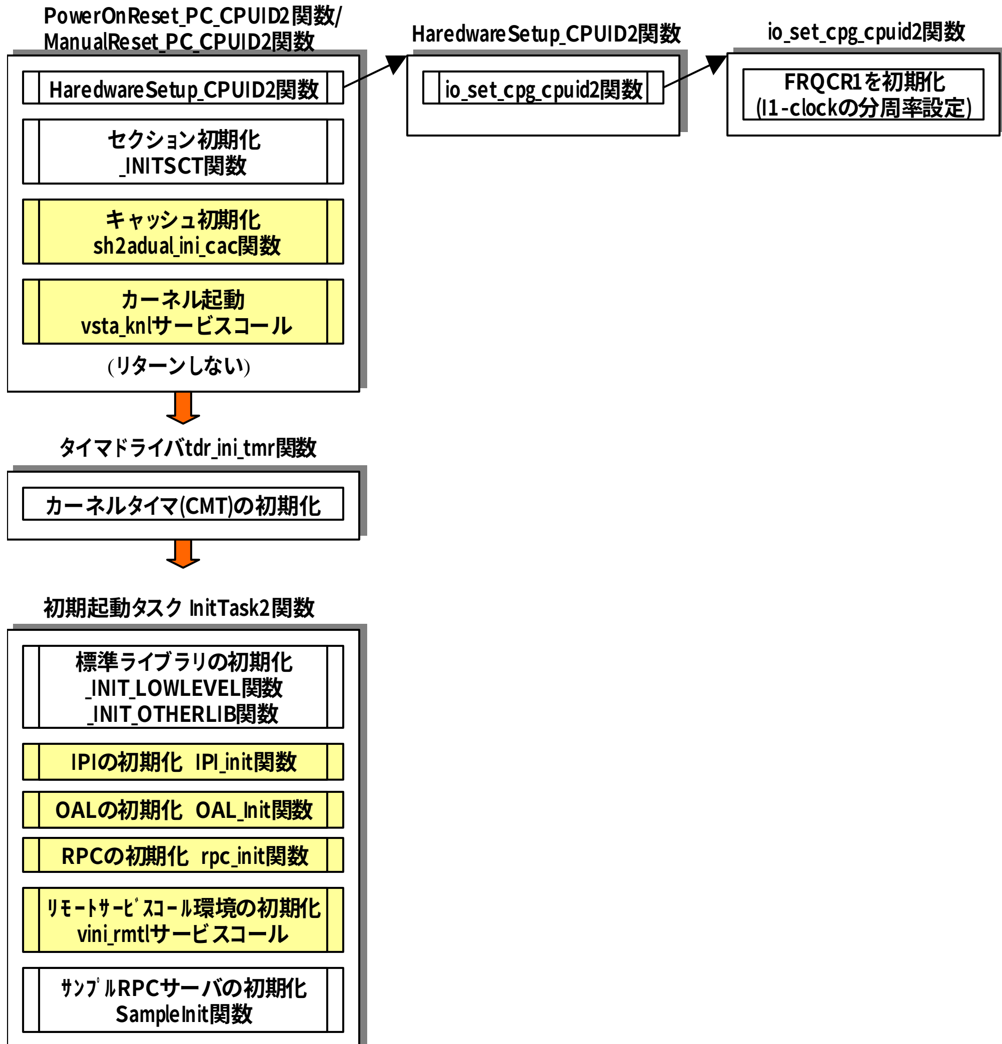


図16.4 CPUID#2 のスタートアップフロー

16.3.2 リセットベクタ(cpuid1¥reset¥reset.src)

SH7265 では、両方の CPU が 0 番地のリセットベクタテーブルからベクタフェッチして実行を開始します。

本カーネルでは、本ファイルのようにユーザがリセットベクタテーブルを作成する必要があります。

また、本カーネルでは CPU 毎にロードモジュールを分けて生成しますが、リセットベクタテーブルは CPUID#1 側のリンク単位で生成します。

表 16.3に、リセットベクタテーブルの登録内容を示します。リセットベクタテーブルと表 16.3に示す 2 つのプログラムは、共に CPUID#1 側の reset.src にあります。このファイルは、アセンブリ言語で記述されています。

表16.3 リセットベクタテーブル

ベクタ番号	意味	登録内容
0	パワーオンリセット PC	_Reset_Poweron：パワーオンリセット用プログラム
1	パワーオンリセット SP	内蔵 RAM0 の最終アドレス
2	マニュアルリセット PC	_Reset_Manual：マニュアルリセット用プログラム
3	マニュアルリセット SP	内蔵 RAM0 の最終アドレス

_Reset_Poweron および_Reset_Manual は、両方の CPU で同時に実行されることになります。

以下、これらのプログラムの処理概要について説明します。

リセットが発生すると、CPU のリセット処理によって、表 16.3に示すようにスタックポインタ(R15)が初期化され、両 CPU でこれらのプログラムの実行が開始されます。なお、初期化されたスタックポインタは CPUID#1 用なので、CPUID#2 として実行する場合は、CPUID#2 用にスタックポインタを初期化するまでは一切スタックを使用してはなりません。これらのプログラムがアセンブリ言語で実装されているのは、スタックを確実に使用しないようにするためです。

次に、実行中の CPU を判定します。

実行中の CPU が CPUID#1 なら、C 言語記述の CPUID#1 用の以下のリセット本体プログラムに JMP 命令により分岐します。分岐後、リターンすることはありません。

- パワーオンリセット用：PowerON_Reset_PC_CPUID1()
- マニュアルリセット用：Manual_Reset_PC_CPUID1()

一方、実行中の CPU が CPUID#2 なら、CPUID#1 による共通ハードウェアの初期化(「16.3.3 CPUID#1 のリセット本体プログラム」参照)が完了するまでビジーウェイトし、その後 CPUID#2 用の仮想リセットベクタテーブル(「16.3.5 CPUID#2 の仮想リセットベクタテーブル(cpuid2¥reset¥vreset.src)」参照)を参照して、CPUID#2 のみのリセット相当の動作をエミュレートします。すなわち、スタックポインタを仮想リセットベクタテーブルに従って初期化し、仮想リセットベクタテーブルに登録されたアドレスに JMP 命令により分岐します。分岐後、リターンすることはありません。

RENESAS

図 16.5に、reset.src のソースコードとその解説を示します。

1	;**	
2	;* Import virtual reset vector table symbol of CPUID#2	
3	;*	
4	;* When CPUID#1 is linked, you must define the absolute address of	
5	;* the symbol "_ResetVectorTable_CPUID2" manually.	
6	;* When CPUID#2 is linked, you must locate "CC_resetvct" section to the above address.	
7	;* If you locate this section to other address, you must re-link CPUID#1 side with	
8	;* new address definition for "_ResetVectorTable_CPUID2".	
9	;**	
10	.import _ResetVectorTable_CPUID2 ; In "CC_resetvct" section of CPUID#2	CPUID#2 側の仮想リセットベクタ
11		テーブルの外部参照宣言
12		(リンク時にアドレスを強制定義)
13	;**	
14	;* Please define copied program size for CPUID#2	
15	;**	
16	;*** for POWERON RESET	
17	COPYSIZE_POWERON .assign 20	CPUID#2 が、パワーオンリセット時に
18	; [Caution!]	内蔵 RAM でビジーウェイトするプロ
19	; The size must be equal to	グラムサイズ
20	; POWERON_EXEC_RAM1_END - POWERON_EXEC_RAM1_START)	
21		
22	;*** for MANUAL RESET	CPUID#2 が、マニュアルリセット時に
23	COPYSIZE_MANUAL .assign 20	内蔵 RAM でビジーウェイトするプロ
24	; [Caution!]	グラムサイズ
25	; The size must be equal to	
26	; (MANUAL_EXEC_RAM1_END - MANUAL_EXEC_RAM1_START)	
27		
28		
29	;**	
30	;* Definition	
31	;**	
32	URAMEND_CPUID1 .assign H'FFF90000 ;* End of URAM0 address (reset stack for CPUID#1)	内蔵RAM0最終アドレス(CPUID#1のスタックポインタ初期値)
33		
34	CPUIDR .assign H'FFFC1404 ;* CPUIDR register address	CPUIDR レジスタアドレス
35	CPUIDR_CPU2 .assign H'40000000	
36		
37	DELAY_CPUID2 .assign H'200 ;* delay count of CPUID#2 to wait for CPUID#1	
38	;* to initialize s_ucIsInitHW	
39		
40	;**	両CPU共通のリセットベクタテーブル
41	;* Reset vector table	
42	;**	セクション名=CC_resetvct
43	.section CC_resetvct, data, align=4	
44	.export _ResetVectorTable	
45		
46	_ResetVectorTable:	
47	;* 0 : Power-on Reset (PC)	
48	.data.l _Reset_Poweron ; in this file	

図16.5 cpuid1¥reset¥reset.src

49	;* 1 : Power-on Reset (SP)	
50	.data.l URAMEND_CPUID1	
51	;* 2 : Manual Reset (PC)	
52	.data.l _Reset_Manual ; in this file	
53	;* 3 : Manual Reset (SP)	
54	.data.l URAMEND_CPUID1	
55		
56	;**	
57	;* _Reset_Poweron	Reset_Poweron プログラム
58	;* Power-on reset program for both CPU (reset vector entry)	
59	;* This program should not use stack.	
60	;**	
61	.section PC_reset	
62	.export _Reset_Poweron	
63	.import _PowerON_Reset_PC_CPUID1	
64		
65	_Reset_Poweron:	
66	;*** if I'm CPUID#1 then POWERON_CPUID1, else POWERON_CPUID2	実行中 CPU の判定
67	mov.l #CPUIDR,r0	
68	mov.l @r0,r0	
69	mov.l #CPUIDR_CPU2,r1	
70	tst r0,r1	
71	bf POWERON_CPUID2	
72		
73	;************* For CPUID#1 *************	CPUID#1 の場合に実行するパス
74	;*** initialize s_ucIsInitHW	各初期化フラグの初期化(図 16.11の A1)
75	mov.l #_s_ucIsInitHW,r1	
76	mov #0,r0	
77	mov.b r0,@r1	
78		
79	;*** initialize s_ucIsInitKnlCPUID1	
80	mov.l #_s_ucIsInitKnlCPUID1,r1	
81	mov #0,r0	
82	mov.b r0,@r1	
83		
84	;*** initialize s_ucIsInitEnvCPUID1	
85	mov.l #_s_ucIsInitEnvCPUID1,r1	
86	mov #0,r0	
87	mov.b r0,@r1	
88		
89	;*** initialize s_ucIsInitEnvCPUID2	
90	mov.l #_s_ucIsInitEnvCPUID2,r1	
91	mov #0,r0	
92	mov.b r0,@r1	
93		
94	;*** wait for the completion that CPUID#2 set s_ucIsInitHW = 1	CPUID#2 が s_ucIsInitHW に 1 をセットするまで待つ(図 16.11の B11)
95	POWERON1_HW_WAIT:	
96	mov.b @r1,r0	
97	cmp/eq #1,r0	
98	bf POWERON1_HW_WAIT	
99		

図 16.5 cpuid1¥reset¥reset.src (続き)

100	;*** jump to PowerON_Reset_PC_CPUID1()	PowerON_Reset_PC_CPUID1()に分岐
101	mov.l #_PowerON_Reset_PC_CPUID1,r0	
102	jmp @r0	
103	nop	
104		
105	;************* For CPUID#2 *************	CPUID#2 の場合に実行するパス
106	POWERON_CPUID2:	
107	;*** get ResetVectorTable_CPUID2 address to R7	
108	mov.l #_ResetVectorTable_CPUID2,r7	
109		
110	;*** initialize R15	CPUID#2 側の仮想リセットベクタを元にスタック
111	mov.l @(4,r7),r15 ; load SP(R15)	ポインタを初期化
112		
113	;*** delay to wait for CPUID#1 to initialize s_ucIsInitHW	CPUID#1 が s_ucIsInitHW を初期化するのに十分な
114	mov.l #DELAY_CPUID2,r0	時間だけ遅延(図 16.11の B21)
115		
116	POWERON2_DELAY:	
117	dt r0	
118	bf POWERON2_DELAY	
119		
120	;*** jump to RAM1	内蔵 RAM1 にプロフラムをコピーし、コール
121	mov.l #POWERON_EXEC_RAM1_START,r1	(図 16.11の B22)
122	mov.l #R_POWERON_EXEC_RAM1_START,r2	
123	mov.l #COPYSIZE_POWERON,r3	
124	add r1,r3	
125		
126	POWERON_LOOP_COPY:	
127	mov.w @r1+,r0	
128	mov.w r0,@r2+	
129	cmp/hi r1,r3	
130	bt POWERON_LOOP_COPY	
131		
132	mov.l #R_POWERON_EXEC_RAM1_START,r0	
133	jsr/n @r0	
134		
135	;*** jump to PowerON_Reset_PC_CPUID2()	CPUID#2 側の仮想リセットベクタに登録されている
136	mov.l @(0,r7),r1 ; load PC	アドレス (PowerON_Reset_PC_CPUID2()) に分岐
137	jmp @r1	
138	nop	
139		
140	.pool	
141		

図 16.5 cpuid1¥reset¥reset.src (続き)

142	;*** This code is copied to RAM1 *****************	内蔵RAM1にコピーされるプログラム
143	POWERON_EXEC_RAM1_START:	
144	;*** set s_ucIsInitHW = 1	s_ucIsInitHWに1をセット(図16.11のB23)
145	mov.l #_s_ucIsInitHW,r1	
146	mov #1,r0	
147	mov.b r0,@r1	
148		
149	;*** wait for the completion that CPUID#1 set s_ucIsInitHW=2	CPUID#2がs_ucIsInitHWに2をセットするまで待つ(図16.11のB24)
150	POWERON2_HW_WAIT:	
151	mov.b @r1,r0	
152	cmp/eq #2,r0	
153	bf POWERON2_HW_WAIT	
154		
155	rts ; do not "rts/n"	リターン
156	nop	
157	.pool	
158	POWERON_EXEC_RAM1_END:	
159	;*** end of copied program	
160		
161		
162	;**	
163	;* _Reset_Manual	Reset_Manualプログラム
164	;* Manual reset program for both CPU (reset vector entry)	
165	;* This program should not use stack.	
166	;**	
167	.export _Reset_Manual	
168	.import _Manual_Reset_PC_CPUID1	
169		
170	_Reset_Manual	
171	;*** if I'm CPUID#1 then MANUAL_CPUID1, else MANUAL_CPUID2	実行中CPUの判定
172	mov.l #CPUIDR,r0	
173	mov.l @r0,r0	
174	mov.l #CPUIDR_CPU2,r1	
175	tst r0,r1	
176	bf MANUAL_CPUID2	
177		
178	;************* For CPUID#1 *************	CPUID#1の場合に実行するパス
179	;*** initialize s_ucIsInitHW	各初期化フラグの初期化(図16.11のA1)
180	mov.l #_s_ucIsInitHW,r1	
181	mov #0,r0	
182	mov.b r0,@r1	
183		
184	;*** initialize s_ucIsInitKnlCPUID1	
185	mov.l #_s_ucIsInitKnlCPUID1,r1	
186	mov #0,r0	
187	mov.b r0,@r1	
188		

図16.5 cpuid1¥reset¥reset.src (続き)

RENESAS

189	;*** initialize s_ucIsInitEnvCPUID1	
190	mov.l #_s_ucIsInitEnvCPUID1,r1	
191	mov #0,r0	
192	mov.b r0,@r1	
193		
194	;*** initialize s_ucIsInitEnvCPUID2	
195	mov.l #_s_ucIsInitEnvCPUID2,r1	
196	mov #0,r0	
197	mov.b r0,@r1	
198		
199	;*** wait for the completion that CPUID#2 set s_ucIsInitHW = 1	CPUID#2 が s_ucIsInitHW に 1 をセット
200	MANUAL1_HW_WAIT:	するまで待つ(図 16.11の B11)
201	mov.b @r1,r0	
202	cmp/eq #1,r0	
203	bf MANUAL1_HW_WAIT	
204		
205	;*** jump to Manual_Reset_PC_CPUID1()	ManualReset_PC_CPUID1()に分岐
206	mov.l #_Manual_Reset_PC_CPUID1,r0	
207	jmp @r0	
208	nop	
209		
210	;************* For CPUID#2 *************	CPUID#2 の場合に実行するパス
211	MANUAL_CPUID2:	
212	;*** get ResetVectorTable_CPUID2 address to R7	
213	mov.l #_ResetVectorTable_CPUID2,r7	
214		
215	;*** initialize R15	CPUID#2 側の仮想リセットベクタを元にスタックポインタを
216	mov.l @(12,r7),r15 ; load SP(R15)	初期化
217		
218	;*** delay to wait for CPUID#1 to initialize s_ucIsInitHW	CPUID#1 が s_ucIsInitHW を初期化するのに十分な
219	mov.l #DELAY_CPUID2,r0	時間だけ遅延(図 16.11の B21)
220		
221	MANUAL2_DELAY:	
222	dt r0	
223	bf MANUAL2_DELAY	
224		
225	;*** jump to RAM1	内蔵 RAM1 にプロフラムをコピーし、コール
226	mov.l #MANUAL_EXEC_RAM1_START,r1	(図 16.11の B22)
227	mov.l #R_MANUAL_EXEC_RAM1_START,r2	
228	mov.l #COPYSIZE_MANUAL,r3	
229	add r1,r3	
230		
231	MANUAL_LOOP_COPY:	
232	mov.w @r1+,r0	
233	mov.w r0,@r2+	
234	cmp/hi r1,r3	
235	bt MANUAL_LOOP_COPY	
236		
237	mov.l #R_MANUAL_EXEC_RAM1_START,r0	
238	jsr/n @r0	
239		

図 16.5 cpuid1¥reset¥reset.src (続き)

240	;*** jump to Manual_Reset_PC_CPUID2()	
241	mov.l @(8,r7),r1 ; load PC	
242	jmp @r1	
243	nop	
244		
245	.pool	
246		
247	;*** This code is copied to RAM1 *****************	内蔵RAM1にコピーされるプログラム
248	MANUAL_EXEC_RAM1_START:	
249	;*** set s_ucIsInitHW = 1	s_ucIsInitHW に 1 をセット
250	mov.l #_s_ucIsInitHW,r1	(図 16.11の B23)
251	mov #1,r0	
252	mov.b r0,@r1	
253		
254	;*** wait for the completion that CPUID#1 set s_ucIsInitHW=2	CPUID#2 が s_ucIsInitHW に 2 をセットするまで待つ(図 16.11の B24)
255	MANUAL2_HW_WAIT:	
256	mov.b @r1,r0	
257	cmp/eq #2,r0	
258	bf MANUAL2_HW_WAIT	
259		
260	rts ; do not "rts/n"	リターン
261	nop	
262	.pool	
263	MANUAL_EXEC_RAM1_END:	
264	;*** end of copied program	
265		
266	;***	
267	;* Program Section in RAM1 for CPUID#2	
268	;***	
269	.section BD_URAM1, data, align=4	内蔵 RAM1 に配置するセクション
270		
271	R_POWERON_EXEC_RAM1_START:	プログラムのコピー先の内蔵 RAM 領域(パワーオンリセット用)
272	.res.b COPYSIZE_POWERON	
273	R_MANUAL_EXEC_RAM1_START:	プログラムのコピー先の内蔵 RAM 領域(マニュアルリセット用)
274	.res.b COPYSIZE_MANUAL	
275		
276	;***	
277	;* Flags in RAM0	初期化フラグ
278	;***	
279	.section BL_S_URAM0, data, align=4	
280	.export _s_ucIsInitHW, _s_ucIsInitKnlCPUID1, _s_ucIsInitEnvCPUID1, _s_ucIsInitEnvCPUID2	
281	_s_ucIsInitHW:	
282	.res.b 1	
283	_s_ucIsInitKnlCPUID1:	
284	.res.b 1	
285	_s_ucIsInitEnvCPUID1:	
286	.res.b 1	
287	_s_ucIsInitEnvCPUID2:	
288	.res.b 1	
289		
290	.end	

図 16.5 cpuid1¥reset¥reset.src (続き)

16.3.3 CPUID#1 のリセット本体プログラム(cpuid1¥reset¥resetprg1.c)

CPUID#1 用のリセット本体プログラムには以下の２つがあります。

- パワーオンリセット用：PowerON_Reset_PC_CPUID1()
- マニュアルリセット用：Manual_Reset_PC_CPUID1()

(1) PowerON_Reset_PC_CPUID1()

本関数は、パワーオンリセットが発生し、実行 CPU が CPUID#1 の場合に、reset.src の_Reset_Poweron からの分岐によって起動されます。本関数からリターンしてはなりません。

本関数では、以下のような処理を行います。

(1) 共通およびCPUID#1のハードウェアの初期化(HardwareSetup_CPUID1()をコール)
(2) セクションの初期化(_INITSCT()をコール)
(3) キャッシュの初期化(sh2adual_ini_cac()をコール) [3]
(4) カーネルを起動(vsta_knlをコール)

なお、標準ライブラリの初期化は、初期起動タスクで行います。これは、リエントラントライブラリ使用時など、標準ライブラリを初期化するためにカーネルの機能が必要になる場合があるためです。

(2) Manual_Reset_PC_CPUID1()

本関数は、マニュアルリセットが発生し、実行 CPU が CPUID#1 の場合に、reset.src の_Reset_Manual から分岐によって起動されます。本関数からリターンしてはなりません。

本関数の処理内容は、PowerON_Reset_PC_CPUID1()とほぼ同じです。

図 16.6に、resetprg1.c のソースコードとその解説を示します。

```
1   /*********************************************************************************
2    Includes
3   *********************************************************************************/
4   #include <machine.h>
5
6   #include "types.h"
7   #include "kernel.h"
8   #include "sh2adual_cache.h"
9
10  #include "io_sys.h"
11  #include "io_multicore.h"
12
13  #include "initsct.h"
14
```

図16.6 cpuid1¥reset¥resetprg1.c

[3] キャッシュの初期化はこれより前に行っても構いませんが、以下のようなケースに注意してください。
プログラムセクションをROMからRAMに転送するセクション初期化を行う場合で、セクション初期化前にキャッシュをコピーバックモードで初期化する場合は、セクション初期化後に転送先のプログラムコード領域に対するオペランドキャッシュ内容を、実メモリにコピーバックさせる必要があります。これを行わないと、セクション初期化時にオペランドキャッシュに書き込まれたプログラムコードが、キャッシュエントリのリプレースによって実メモリにコピーバックされる前に、そのプログラムコードのアドレスから命令フェッチしたときに、不正な命令コードを実行することになるためです。

15	/**	
16	Prototypes	
17	**/	
18	void PowerON_Reset_PC_CPUID1(void);	
19	void Manual_Reset_PC_CPUID1(void);	
20		
21	/**	
22	External reference	
23	**/	
24	extern void HardwareSetup_CPUID1(void);	
25		
26	/**	
27	Section definition	
28	**/	
29	#pragma section C_reset	セクション名を設定
30		
31	/**	
32	Initialize section information	セクション初期化情報テーブル
33	**/	
34	/*** D section information table ***/	
35	static const ST_DTBL dtbl[]= {	
36	MACRO_ENTRY_DTBL("DC_stdlib", "RC_stdlib")	
37	};	
38	/*** B section information table ***/	
39	static const ST_BTBL btbl[]= {	
40	MACRO_ENTRY_BTBL("BC_stdlib"),	
41	MACRO_ENTRY_BTBL("BC_sample"),	
42	MACRO_ENTRY_BTBL("BC_heap")	
43	};	
44		
45	/**/	
46	/** Power-on Reset function	PowerON_Reset_PC_CPUID1()
47	* @retval None	
48	* @note This routine is called from "_Reset_Poweron" in "reset.src",	
49	* and must not return.	
50	**/	
51	#pragma noregsave(PowerON_Reset_PC_CPUID1)	リターンしないためレジスタ保証は不要
52	void PowerON_Reset_PC_CPUID1(void)	
53	{	
54	extern UINT8 s_ucIsInitHW; /* defined in reset.src */	
55		
56	set_imask(15); /* set SR.IMASK = 15 */	
57		
58	/*** check CPUID ***/	
59	if(IO_CPUIDR.BIT._ID != (MYCPUID)-1U) {	実行 CPU が想定以外の場合は停止
60	while(1){ /* error */	
61	};	
62	}	
63		

図 16.6 cpuid1¥reset¥resetprg1.c(続き)

RENESAS

64	`/*** initialize shared hardware and CPUID#1 hardware ***/`	
65	`HardwareSetup_CPUID1();`	共通ハード初期化後に
66	`s_ucIsInitHW = 2;`	s_ucIsInitHW に 2 をセット
67		(図 16.11の B12)
68	`/*** initialize section ***/`	セクション初期化
69	`_INITSCT(dtbl, sizeof dtbl, btbl, sizeof btbl);`	
70		
71	`/*** initialize cache ***/`	キャッシュ初期化
72	`sh2adual_ini_cac(TCAC_IC_ENABLE|TCAC_OC_ENABLE);`	
73		
74	`/*** start kernel (never return) ***/`	カーネル起動(ここにはリターン
75	`vsta_knl();`	しない)
76		
77	`/*** (NEVER return from vsta_knl()) ***/`	
78	`while(1) {`	
79	`}`	
80	`}`	
81		
82	`/***************************************************************************/`	
83	`/** Manual Reset function`	ManualReset_PC_CPUID1()
84	`* @retval None`	
85	`* @note This routine is called from "_Reset_Manual" in "reset.src",`	
86	`* and must not return.`	
87	`***************************************************************************/`	
88	`#pragma noregsave(Manual_Reset_PC_CPUID1)`	
89	`void Manual_Reset_PC_CPUID1(void)`	
90	`{`	
91	`extern UINT8 s_ucIsInitHW; /* defined in reset.src */`	
92		
93	`set_imask(15); /* set SR.IMASK = 15 */`	
94		
95	`/*** clear DSFR.MRES ***/`	
96	`IO_SYS.DSFR.BIT._MRES = 0;`	
97		
98	`/*** initialize shared hardware and CPUID#1 hardware ***/`	
99	`HardwareSetup_CPUID1();`	共通ハード初期化後に
100	`s_ucIsInitHW = 2;`	s_ucIsInitHW に 2 をセット
101		(図 16.11の B12)
102	`/*** initialize section ***/`	セクション初期化
103	`_INITSCT(dtbl, sizeof dtbl, btbl, sizeof btbl);`	
104		
105	`/*** initialize cache ***/`	キャッシュ初期化
106	`sh2adual_ini_cac(TCAC_IC_ENABLE|TCAC_OC_ENABLE);`	
107		
108	`/*** start kernel (never return) ***/`	カーネル起動(ここにはリターン
109	`vsta_knl();`	しない)
110		
111	`/*** (NEVER return from vsta_knl()) ***/`	
112	`while(1)`	
113	`{`	
114	`}`	
115	`}`	

図 16.6 cpuid1¥reset¥resetprg1.c(続き)

16.3.4 共通ハードウェアおよび CPUID#1 リソースの初期化関数 HardwareSetup_CPUID1() (cpuid1¥reset¥hwsetup1.c)

両 CPU で共有するハードウェアリソースと CPUID#1 のハードウェアリソースの初期化は、CPUID#1 側で実行する HardwareSetup_CPUID1()によって行っています。HardwareSetup_CPUID1()は、PowerON_Reset_PC_CPUID1()および Manual_Reset_PC_CPUID1()から呼び出されます。

HardwareSetup_CPUID1()は、以下の関数を呼び出します。

(1) io_set_cpg_cpuid1() (cpg1.c)

FRQCR0 レジスタの設定を行い、各クロックを以下のように設定します。

- CPUID#1 内部クロック(I0φ)：200MHz
- CPUID#2 内部クロック(I1φ)：200MHz
- バスクロック(Bφ)：66.67MHz
- 周辺クロック(Pφ)：33.33MHz

また、各内蔵モジュールをモジュールスタンバイにするかどうかのデフォルト設定を行っています。

なお、FRQCR1 レジスタによる I1φの分周率の設定は、CPUID#2 側の io_set_cpg_cpuid2() (cpg2.c)で行います。これは、FRQCR1 の変更は CPUID#2 からのみ可能な LSI 仕様であるためです。

また、本関数とは別に、周辺クロック周波数を iodefine¥pclock.h で定義しています。周辺クロックを変更した場合は、pclock.h の定義も変更してください。

(2) io_set_uram() (uram.c)

内蔵 RAM へのアクセス権を初期化します。本サンプルでは、表 16.4のように初期化しています。

表16.4 内蔵 RAM アクセス権の初期化

	CPUID#1 からのアクセス	CPUID#2 からのアクセス	DMAC からのアクセス
RAM0 ページ 0	リード・ライト可能	リード・ライト可能	リード・ライト可能
RAM0 ページ 1	リード・ライト可能	リードのみ可能	リード・ライト可能
RAM0 ページ 2	リード・ライト可能	リードのみ可能	リード・ライト可能
RAM0 ページ 2	リード・ライト可能	リードのみ可能	リード・ライト可能
RAM1 ページ 0	リードのみ可能	リード・ライト可能	リード・ライト可能
RAM1 ページ 1	リードのみ可能	リード・ライト可能	リード・ライト可能

(3) io_init_bsc_cs0() (bsc_cs0.c)

ピンファンクションコントローラ（PFC）およびバスステートコントローラ(BSC)の設定を行い、CS0 空間のフラッシュメモリに対するアクセスタイミングを設定します。

(4) io_init_sdram() (bscsdram.c)

ピンファンクションコントローラ（PFC）およびバスステートコントローラ(BSC)の設定を行い、SDRAM0 空間の SDRAM 空間を有効にします。

16.3.5 CPUID#2 の仮想リセットベクタテーブル(cpuid2¥reset¥vreset.src)

仮想リセットベクタテーブルは、CPUID#2 専用の仮想的なリセットベクタテーブルであり、CPUID#1 側の reset.src の_Reset_Poweron および_Reset_Manual から参照されます。

テーブルのシンボル名はアセンブリ言語で" _ResetVectorTable_CPUID2"、セクション名は" CC_vresetvct"です。

このテーブルは CPUID#2 側にリンクしますが、CPUID#1 側にリンクされる reset.src から参照されます。このため、以下が必要です。

(1) 仮想リセットベクタテーブルを配置するアドレスをあらかじめ決定しておく。

(2) CPUID#1側のリンク時には、シンボル" _ResetVectorTable_CPUID2"のアドレスを(1)で決定したアドレスに強制定義する。

(3) CPUID#2側のリンク時には、セクション" CC_vresetvct"を(1)で決定したアドレスに配置する。

表 16.5に、仮想リセットベクタテーブルの登録内容を示します。

表16.5 仮想リセットベクタテーブル

ベクタ番号	意味	登録内容
0	パワーオンリセット PC	PowerON_Reset_PC_CPUID2()：パワーオンリセット本体プログラム
1	パワーオンリセット SP	内蔵 RAM1 の最終アドレス
2	マニュアルリセット PC	Manual_Reset_PC_CPUID2()：マニュアルリセット本体プログラム
3	マニュアルリセット SP	内蔵 RAM1 の最終アドレス

PowerON_Reset_PC_CPUID2()および Manual_Reset_PC_CPUID2()は、CPUID#2 側の resetprg2.c にあるリセット本体プログラムで、それぞれ reset.src の_Reset_Poweron および_Reset_Manual から JMP 命令による分岐によって起動されます。起動時点では、それぞれベクタ番号 1, 3 にしたがってスタックポインタが初期化されています。

図 16.7に、vreset.src のソースコードとその解説を示します。

1	;**	
2	;* Definition	
3	;**	
4	URAMEND_CPUID2 .assign H'FFFA8000 ;* End of URAM1 address (reset stack for CPUID#2)	内蔵 RAM1 最終アドレス(CPUID#2 のスタックポインタ初期値)
5		
6	;**	
7	;* Virtual reset vector table	仮想リセットベクタテーブル
8	;*	
9	;* This table is referred by "reset.src".	
10	;* This is virtual reset vector table for CPUID#2.	
11	;* The "reset.src" emulates "Reset" by referring this table.	
12	;* When linking of CPUID#2, do not change the location address of the	
13	;* "CC_resetvct" section. If changed, CPUID#1 must be re-linked with	
14	;* changed this symbol address.	
15	;**	
16	.section CC_vresetvct, data, align=4	セクション名＝”CC_vresetvct”
17	.export _ResetVectorTable_CPUID2	仮想リセットベクタテーブルシンボルの外部定義宣言
18	.import _PowerON_Reset_PC_CPUID2	PowerON_Reset_PC_CPUID2()の外部参照定義
19	.import _Manual_Reset_PC_CPUID2	Manual_Reset_PC_CPUID2()の外部参照定義
20		
21	_ResetVectorTable_CPUID2:	
22	;* 0 : Power-on Reset (PC)	パワーオンリセット PC = PowerON_Reset_PC_CPUID2()
23	.data.l _PowerON_Reset_PC_CPUID2 ; in resetprg.c	
24	;* 1 : Power-on Reset (SP)	パワーオンリセット SP = 内蔵 RAM1 最終アドレス
25	.data.l URAMEND_CPUID2	
26	;* 2 : Manual Reset (PC)	マニュアルリセット PC
27	.data.l _Manual_Reset_PC_CPUID2 ; in resetprg.c	= Manual_Reset_PC_CPUID2()
28	;* 3 : Manual Reset (PC)	マニュアルリセット SP = 内蔵 1 最終アドレス
29	.data.l URAMEND_CPUID2	
30		
31	.end	

図16.7 cpuid2¥reset¥vreset.src

RENESAS

16.3.6 CPUID#2 のリセット本体プログラム(cpuid2¥reset¥resetprg2.c)

CPUID#2 用のリセット本体プログラムには以下の２つがあります。

- パワーオンリセット用：PowerON_Reset_PC_CPUID2()
- マニュアルリセット用：Manual_Reset_PC_CPUID2()

なお、共有ハードウェアリソースは、CPUID#1 によって初期化され、CPUID#2 側はその初期化が完了してから本関数が呼び出されます。

(1) PowerON_Reset_PC_CPUID2()

本関数は、パワーオンリセットが発生し、実行 CPU が CPUID#2 の場合に、reset.src の_Reset_Poweron から分岐によって起動されます。本関数からリターンしてはなりません。

本関数では、以下のような処理を行います。

(1) CPUID#2のハードウェアの初期化(HardwareSetup_CPUID2()をコール)
(2) セクションの初期化(_INITSCT()をコール)
(3) キャッシュの初期化(sh2adual_ini_cac()をコール) [4]
(4) カーネルを起動(vsta_knlをコール)

なお、標準ライブラリの初期化は、初期起動タスクで行います。これは、リエントラントライブラリ使用時など、標準ライブラリを初期化するためにカーネルの機能が必要になる場合があるためです。

(2) Manual_Reset_PC_CPUID2()

本関数は、マニュアルリセットが発生し、実行 CPU が CPUID#2 の場合に、reset.src の_Reset_Manual から分岐によって起動されます。本関数からリターンしてはなりません。

本関数の処理内容は、PowerON_Reset_PC_CPUID2()とほぼ同じです。

図 16.8に、resetprg2.c のソースコードとその解説を示します。

```
1   /**********************************************************************************
2    Includes
3   **********************************************************************************/
4
5
6   #include "types.h"
7   #include "kernel.h"
8   #include "sh2adual_cache.h"
9
10  #include "io_multicore.h"
11
12  #include "initsct.h"
13
```

図16.8 cpuid2¥reset¥resetprg2.c

[4] キャッシュの初期化はこれより前に行っても構いませんが、以下のようなケースに注意してください。
プログラムセクションをROMからRAMに転送するセクション初期化を行う場合で、セクション初期化前にキャッシュをコピーバックモードで初期化する場合は、セクション初期化後に転送先のプログラムコード領域に対するオペランドキャッシュ内容を、実メモリにコピーバックさせる必要があります。これを行わないと、セクション初期化時にオペランドキャッシュに書き込まれたプログラムコードが、キャッシュエントリのリプレースによって実メモリにコピーバックされる前に、そのプログラムコードのアドレスから命令フェッチしたときに、不正な命令コードを実行することになるためです。

14	/***	
15	Prototypes	
16	***/	
17	void PowerON_Reset_PC_CPUID2(void);	
18	void Manual_Reset_PC_CPUID2(void);	
19		
20	/***	
21	External reference	
22	***/	
23	extern void HardwareSetup_CPUID2(void);	
24		
25	/***	
26	Section definition	
27	***/	
28	#pragma section C_reset	セクション名を設定
29		
30	/***	
31	Initialize section information	セクション初期化情報テーブル
32	***/	
33	/*** D section information table ***/	
34	static const ST_DTBL dtbl[]= {	
35	MACRO_ENTRY_DTBL("DC_stdlib", "RC_stdlib")	
36	};	
37	/*** B section information table ***/	
38	static const ST_BTBL btbl[]= {	
39	MACRO_ENTRY_BTBL("BC_stdlib"),	
40	MACRO_ENTRY_BTBL("BC_sample"),	
41	MACRO_ENTRY_BTBL("BC_heap")	
42	};	
43		
44	/***/	
45	/** Power-on Reset function	PowerON_Reset_PC_CPUID2()
46	* @retval None	
47	* @note This routine is called from "_Reset_Poweron" in "reset.src",	
48	* and must not return.	
49	***/	
50	#pragma noregsave(PowerON_Reset_PC_CPUID2)	リターンしないためレジスタ保証は不要
51	void PowerON_Reset_PC_CPUID2(void)	
52	{	
53	extern UINT8 s_ucIsInitKnlCPUID1; /* defined in reset.src */	
54		
55	set_imask(15); /* set SR.IMASK = 15 */	
56		
57	/*** check CPUID ***/	
58	if(IO_CPUIDR.BIT._ID != (MYCPUID)-1U) {	実行 CPU が想定以外の場合は停止
59	while(1) { /* error */	
60	};	
61	}	
62		
63	/*** initialize CPUID#2 hardware ***/	
64	HardwareSetup_CPUID2();	CPUID#2 のハードウェア初期化
65		

図 16.8 cpuid2¥reset¥resetprg2.c(続き)

RENESAS

66	/*** initialize section ***/	セクション初期化
67	_INITSCT(dtbl, sizeof dtbl, btbl, sizeof btbl);	
68		
69	/*** initialize cache ***/	キャッシュ初期化
70	sh2adual_ini_cac(TCAC_IC_ENABLE|TCAC_OC_ENABLE);	
71		
72	/*** wait for s_ucIsInitKnlCPUID1=1 ***/	CPUID#1 が s_ucIsInitKnlCPUID1
73	while(s_ucIsInitKnlCPUID1 == 0) {	1 をセットするまで待つ
74	}	(図16.11の C21)
75		
76	/*** start kernel (never return) ***/	カーネル起動(ここにはリターン
77	vsta_knl();	しない)
78		
79	/*** (NEVER return from vsta_knl()) ***/	
80	while(1) {	
81	}	
82	}	
83		
84	/***/	
85	/** Manual Reset function	ManualReset_PC_CPUID2()
86	* @retval None	
87	* @note This routine is called from "_Reset_Manual" in "reset.src",	
88	* and must not return.	
89	***/	
90	#pragma noregsave(Manual_Reset_PC_CPUID2)	
91	void Manual_Reset_PC_CPUID2(void)	
92	{	
93	extern UINT8 s_ucIsInitKnlCPUID1; /* defined in reset.src */	
94		
95	set_imask(15); /* set SR.IMASK = 15 */	
96		
97	/*** initialize CPUID#2 hardware ***/	CPUID#2 のハードウェア初期化
98	HardwareSetup_CPUID2();	
99		
100	/*** initialize section ***/	セクション初期化
101	_INITSCT(dtbl, sizeof dtbl, btbl, sizeof btbl);	
102		
103	/*** initialize cache ***/	キャッシュ初期化
104	sh2adual_ini_cac(TCAC_IC_ENABLE|TCAC_OC_ENABLE);	
105		
106	/*** wait for s_ucIsInitKnlCPUID1=1 ***/	CPUID#1 が s_ucIsInitKnlCPUID1
107	while(s_ucIsInitKnlCPUID1 == 0) {	1 をセットするまで待つ
108	}	(図16.11の C21)
109		
110	/*** start kernel (never return) ***/	カーネル起動(ここにはリターン
111	vsta_knl();	しない)
112		
113	/*** (NEVER return from vsta_knl()) ***/	
114	while(1) {	
115	}	
116	}	

図 16.8 cpuid2¥reset¥resetprg2.c(続き)

16.3.7 CPUID#1 の初期起動タスク InitTask1() (cpuid1¥init¥init_task1.c)

InitTask1()は、CPUID#1 側で最初に実行されるタスクとして、CPUID#1 側の cfg ファイルに登録されています。

本タスクでは、以下のような初期化を行います。

(1) 標準ライブラリの初期化(_INIT_LOWLEVEL()および_INIT_OTHERLIB()をコール)
(2) IPIの初期化(IPI_init()をコール)
(3) OALの初期化(OAL_Init()をコール)
(4) RPCの初期化(rpc_init()をコール)
なお、rpc_init()コール前に、IPIおよびOALの初期化が完了している必要があります。
(5) リモートサービスコール環境の初期化(vini_rmtをコール)
なお、vini_rmtコール前に、IPIの初期化が完了している必要があります。
(6) サンプルのRPCクライアントの初期化(SampleInit()をコール)

図 16.9に、InitTask1()のソースコードとその解説を示します。

1	/**	
2	Includes	
3	**/	
4	#include "types.h"	
5	#include "kernel.h"	
6	#include "kernel_id.h"	
7		
8	#include "rpc_public.h"	
9	#include "oal.h"	
10	#include "ipi.h"	
11		
12	#include "lowsrc.h"	
13	#include "otherlib.h"	
14		
15	#include "rpc_sample.h"	
16		
17	/**	
18	Prototypes	
19	**/	
20	void InitTask1(VP_INT exinf);	
21		
22	/**	
23	Defines	
24	**/	
25	#define NUM_SERVER 10UL	rpc_init()で指定するサーバ数(実際には、サーバは一切登録していません)
26	#define RPCSERVER_STKSZ 0x200UL	rpc_init()で指定するサーバタスクのスタックサイズ
27	#define RPCSERVER_IPIPORT 2UL /* interrupt level = 13 */	rpc_init()で指定する IPI ポート ID
28		
29	/**	
30	Data	
31	**/	
32	#pragma section L_sample	rpc_init()に渡す rpc_info[]を非キャッシャブル領域に生成
33	static rpc_info RpcInfo[NUM_SERVER];	

図16.9 cpuid1¥init¥init_task1.c

RENESAS

34		
35	/**	
36	Section definition	
37	**/	
38	#pragma section C_sample	セクション名を設定
39		
40	/**/	
41	/** Initial task	初期起動タスク(InitTask1())
42	* This task calls various API to initialize OS,	
43	* and then notifies CPUID#2 to have completed initialization phase of CPUID#1	
44	* by setting s_ucIsInitEnvCPUID1.	
45	* After that, this task waits to complete the initialization phase of CPUID#2.	
46	* Afterwards, this task exits and be deleted.	
47	* This task is created and activated by "task[]" definition in .cfg file.	
48	* @param exinf Undefined	
49	* @retval None	
50	**/	
51	void InitTask1(VP_INT exinf)	
52	{	
53	extern UINT8 s_ucIsInitKnlCPUID1; /* defined in reset.src */	s_ucIsInitKnlCPUID1 の外部参照
54	extern UINT8 s_ucIsInitEnvCPUID1; /* defined in reset.src */	s_ucIsInitEnvCPUID1 の外部参照
55	extern UINT8 s_ucIsInitEnvCPUID2; /* defined in reset.src */	s_ucIsInitEnvCPUID2 の外部参照
56		
57	static const rpc_config ConfigInfo = {	rpc_init()に渡す rpc_config 構造体
58	RpcInfo, /* rpc_info *pRpcTable; */	rpc_info 構造体の配列へのポインタ
59	NUM_SERVER, /* UINT32 ulTableSize; */	登録可能なサーバ数
60	0UL, /* UINT32 ulCmdRspRangeBaseValue; */	予約メンバ
61	0UL, /* UINT32 RedirectionTaskStackSize; */	予約メンバ
62	RPCSERVER_STKSZ, /* UINT32 ServerTaskStackSize; */	サーバタスクのスタックサイズ
63	0UL, /* UINT32 MFIFramePriority; */	予約メンバ
64	1UL, /* UINT32 RPCTaskPriority; */	予約メンバ
65	RPCSERVER_IPIPORT /* UINT32 ulIPIPortID; */	RPC が使用する IPI ポート ID (割込みレベルは、カーネル割込みマスクレベル (system.system_IPL)以下でなければなりません)
66	};	
67		
68	/*** disable dispatch ***/	ディスパッチを禁止
69	dis_dsp();	
70		
71	/*** set s_ucIsInitKnlCPUID1 ***/	s_ucIsInitKnlCPUID1 に 1 をセット (図 16.11の C11)
72	s_ucIsInitKnlCPUID1 = 1;	
73		
74	/*** initialize standard library ***/	標準ライブラリの低水準インタフェースルーチンの初期化
75	if(INIT_LOWLEVEL() != 1) {	
76	while(1){ /* error */	
77	}	
78	}	
79		

図 16.9 cpuid1¥init¥init_task1.c(続き)

RENESAS

80	_INIT_OTHERLIB();	_s1ptr, rand()の初期化
81		
82	/*** initialize IPI ***/	IPI の初期化
83	IPI_init();	
84		
85	/*** initialize OAL ***/	OAL の初期化
86	OAL_Init();	
87		
88	/*** initialize RPC ***/	RPC ライブラリの初期化
89	rpc_init(&ConfigInfo);	
90		
91	/*** initialize remote-SVC ***/	リモートサービスコール環境の初期化
92	vini_rmt();	
93		
94	/*** set ucIsInitializedCPUID1 ***/	s_ucIsInitEnvCPUID1 に 1 をセット(図 16.11の D11)
95	s_ucIsInitEnvCPUID1 = 1;	
96		
97	/*** wait for s_ucIsInitEnvCPUID2=1 ***/	CPUID#2 が s_ucIsInitEnvCPUID2 に 1 をセットするまで待つ
98	while(s_ucIsInitEnvCPUID2 == 0) {	(図 16.11の E11)
99	}	
100		
101	/*** enable dispatch ***/	ディスパッチを許可
102	ena_dsp();	
103		
104	/*** connect to sample RPC server ***/	サンプルの RPC クライアントの初期化
105	SampleInit();	
106		
107	/*** exit and delete this task ***/	初期起動タスクを終了・削除
108	exd_tsk();	
109	}	

図 16.9　cpuid1¥init¥init_task1.c(続き)

16.3.8 CPUID#2 の初期起動タスク InitTask2() (cpuid2¥init¥init_task2.c)

InitTask2()は、CPUID#2 側で最初に実行されるタスクとして、CPUID#2 側の cfg ファイルに登録されています。

本タスクでは、以下のような初期化を行います。

(1) 標準ライブラリの初期化(_INIT_LOWLEVEL()および_INIT_OTHERLIB()をコール)
(2) IPIの初期化(IPI_init()をコール)
(3) OALの初期化(OAL_Init()をコール)
(4) RPCの初期化(rpc_init()をコール)
なお、rpc_init()コール前に、IPIおよびOALの初期化が完了している必要があります。
(5) リモートサービスコール環境の初期化(vini_rmtをコール)
なお、vini_rmtコール前に、IPIの初期化が完了している必要があります。
(6) サンプルのRPCサーバの初期化(SampleInit()をコール)

図 16.10に、InitTask2()のソースコードとその解説を示します。

行	コード	解説
1	/**	
2	Includes	
3	**/	
4	#include "types.h"	
5	#include "kernel.h"	
6	#include "kernel_id.h"	
7		
8	#include "rpc_public.h"	
9	#include "oal.h"	
10	#include "ipi.h"	
11		
12	#include "lowsrc.h"	
13	#include "otherlib.h"	
14		
15	#include "rpc_sample.h"	
16		
17	/**	
18	Prototypes	
19	**/	
20	void InitTask2(VP_INT exinf);	
21		
22	/**	
23	Defines	
24	**/	
25	#define NUM_SERVER 10UL	rpc_init()で指定するサーバ数
26	#define RPCSERVER_STKSZ 0x200UL	rpc_init()で指定するサーバタスクのスタックサイズ
27	#define RPCSERVER_IPIPORT 2UL /* interrupt level = 13 */	rpc_init()で指定する IPI ポート ID
28		
29	/**	
30	Data	
31	**/	
32	#pragma section L_sample	rpc_init()に渡す rpc_info[]を 非キャッシャブル領域に生成
33	static rpc_info RpcInfo[NUM_SERVER];	
34		

図16.10 cpuid2¥init¥init_task2.c

35	/**	
36	Section definition	
37	**/	
38	#pragma section C sample	セクション名を設定
39		
40	/**/	
41	/** Initial task	初期起動タスク(InitTask2())
42	* At first, this task waits to complete the initialization phase of CPUID#1.	
43	* and then calls various API to initialize OS.	
44	* After that, this task notifies CPUID#1 to have completed initialization	
45	* phase of CPUID#2 by setting s_ucIsInitializedCPUID2.	
46	* Afterwards, this task exits and be deleted.	
47	* This task is created and activated by "task[]" definition in .cfg file.	
48	* @param exinf Undefined	
49	* @retval None	
50	**/	
51	void InitTask2(VP_INT exinf)	
52	{	
53	extern UINT8 s_ucIsInitEnvCPUID1; /* defined in reset.src */	s_ucIsInitEnvCPUID1 外部参照
54	extern UINT8 s_ucIsInitEnvCPUID2; /* defined in reset.src */	s_ucIsInitEnvCPUID2 外部参照
55		
56	static const rpc_config ConfigInfo = {	rpc_init()に渡す rpc_config 構造体
57	RpcInfo, /* rpc_info *pRpcTable; */	rpc_info 構造体の配列へのポインタ
58	NUM_SERVER, /* UINT32 ulTableSize; */	登録可能なサーバ数
59	0UL, /* UINT32 ulCmdRspRangeBaseValue; */	予約メンバ
60	0UL, /* UINT32 RedirectionTaskStackSize; */	予約メンバ
61	RPCSERVER_STKSZ, /* UINT32 ServerTaskStackSize; */	サーバタスクのスタックサイズ
62	0UL, /* UINT32 MFIFramePriority; */	予約メンバ
63	1UL, /* UINT32 RPCTaskPriority; */	予約メンバ
64	RPCSERVER_IPIPORT /* UINT32 ulIPIPortID; */	RPC が使用する IPI ポート ID (割込みレベルは、カーネル割込みマスクレベル(system.system_IPL)以下でなければなりません)
65	};	
66		
67	/*** disable dispatch ***/	ディスパッチを禁止
68	dis_dsp();	
69		
70	/*** wait for s_ucIsInitEnvCPUID1=1 ***/	CPUID#1 が s_ucIsInitEnvCPUID に 1 をセットするまで待つ (図 16.11の D21)
71	while(s_ucIsInitEnvCPUID1 == 0) {	
72	}	
73		
74	/*** initialize standard library ***/	標準ライブラリの低水準インタフェースルーチンの初期化
75	if(_INIT_LOWLEVEL() != 1) {	
76	while(1){ /* error */	
77	}	
78	}	

図 16.10 cpuid2¥init¥init_task2.c (続き)

RENESAS

79		
80	_INIT_OTHERLIB();	_s1ptr, rand()の初期化
81		
82	/*** initialize IPI ***/	IPI の初期化
83	IPI_init();	
84		
85	/*** initialize OAL ***/	OAL の初期化
86	OAL_Init();	
87		
88	/*** initialize RPC ***/	RPC ライブラリの初期化
89	rpc_init(&ConfigInfo);	
90		
91	/*** initialize remote-SVC ***/	リモートサービスコール環境の初期化
92	vini_rmt();	
93		
94	/*** enable dispatch ***/	ディスパッチを許可
95	ena_dsp();	
96		
97	/*** start sample RPC server ***/	サンプルの RPC サーバを初期化
98	SampleInit();	
99		
100	/*** set s_ucIsInitEnvCPUID2 ***/	s_ucIsInitEnvCPUID2 フラグをセット(図 16.11の E21)
101	s_ucIsInitEnvCPUID2 = 1;	
102		
103	/*** exit and delete this task ***/	初期起動タスクを終了・削除
104	exd_tsk();	
105	}	

図 16.10 cpuid2¥init¥init_task2.c (続き)

16.3.9 両 CPU のスタートアップフェーズの同期メカニズム

両 CPU が互いの初期化処理状況を確認しながら初期化処理を進めるために、本サンプルでは以下の 4 つのフラグを使っています。これらのフラグは、CPUID#1 側にリンクする reset.src で定義されています。なお、これらのフラグは外部 RAM アクセスのための初期化の前にアクセスする必要があるため、初期化が不要で両 CPU からアクセスできる内蔵 RAM0 に配置しています。

それぞれのフラグの役割は、以下の通りです。また、図 16.11にこれら 4 つのフラグに着目したスタートアップフェーズの同期を示します。

(1) s_ucIsInitHW：共通ハードウェアリソース初期化変数

本サンプルでは、CPUID#1 が共通ハードウェアリソースを初期化します。CPUID#2 は、これが完了するまで SDRAM アクセスなどを行うことができないので、それが完了するまでは何もしないようにしています。

また、CPUID#1 が CS0 の初期化を行うためにバスステートコントローラを設定するときには、バスステートコントローラの仕様上、CPUID#2 は CS0 をアクセスしてはなりません。このために、CPUID#2 は CPUID#1 が CS0 の初期化を完了するまで、内蔵 RAM1 でビジーウェイトさせています。

s_ucIsInitHW は、上記の順序性を実現するための変数です。

s_ucIsInitHW の仕様は、以下の通りです。

- 0：初期状態
- 1：CPUID#1 が共通ハードウェアを初期化可能な状態(CPUID#2 が内蔵 RAM でビジーウェイトしている状態)
- 2：CPUID#1 が共通ハードウェアを初期化済

(2) s_ucIsInitKnlCPUID1：CPUID#1 のカーネル初期化フラグ

s_ucIsInitKnlCPUID1 は、CPUID#1 のカーネルの初期化が完了したことを示すフラグです。

本カーネルでは、CPUID#1 のカーネルの初期化が完了する前に CPUID#2 でサービスコールが行われると、トレースシリアル番号の一貫性が崩れる制限があります。本サンプルでは、s_ucIsInitKnlCPUID1 を用いて、CPUID#2 は CPUID#1 のカーネル初期化が完了するまで(初期起動タスクに制御が移るまで)は vsta_knl を呼び出さないようにしています。

s_ucIsInitKnlCPUID1 の仕様は、以下の通りです。

- 0：初期状態
- 1：CPUID#1 のカーネル初期化処理が完了

(3) s_ucIsInitEnvCPUID1：CPUID#1 のソフトウェア環境初期化フラグ

HI7200/MP では、初期化に関する API に関して、以下のような順序性の制限があります。

- CPUID#2 の IPI_init()は、CPUID#1 の IPI_init()完了後でなければならない
- CPUID#2 の rpc_init()は、CPUID#1 の vini_rmt 完了後でなければならない
- CPUID#2 の vini_rmt は、CPUID#1 の vini_rmt 完了後でなければならない

また、本サンプルでは該当しませんが、アプリケーション側で同様の順序性が必要になる場合もあります。

s_ucIsInitEnvCPUID1 は、この順序性を制御するためのフラグです。具体的には、CPUID#1 の初期起動タスクが上記の初期化 API を完了した後に、CPUID#2 が上記の初期化 API を呼び出すようにしています。

s_ucIsInitEnvCPUID1 の仕様は、以下の通りです。

- 0：初期状態
- 1：CPUID#1 の各種ソフトウェア環境の初期化処理が完了

(4) s_ucIsInitEnvCPUID2：CPUID#2 のソフトウェア環境初期化フラグ

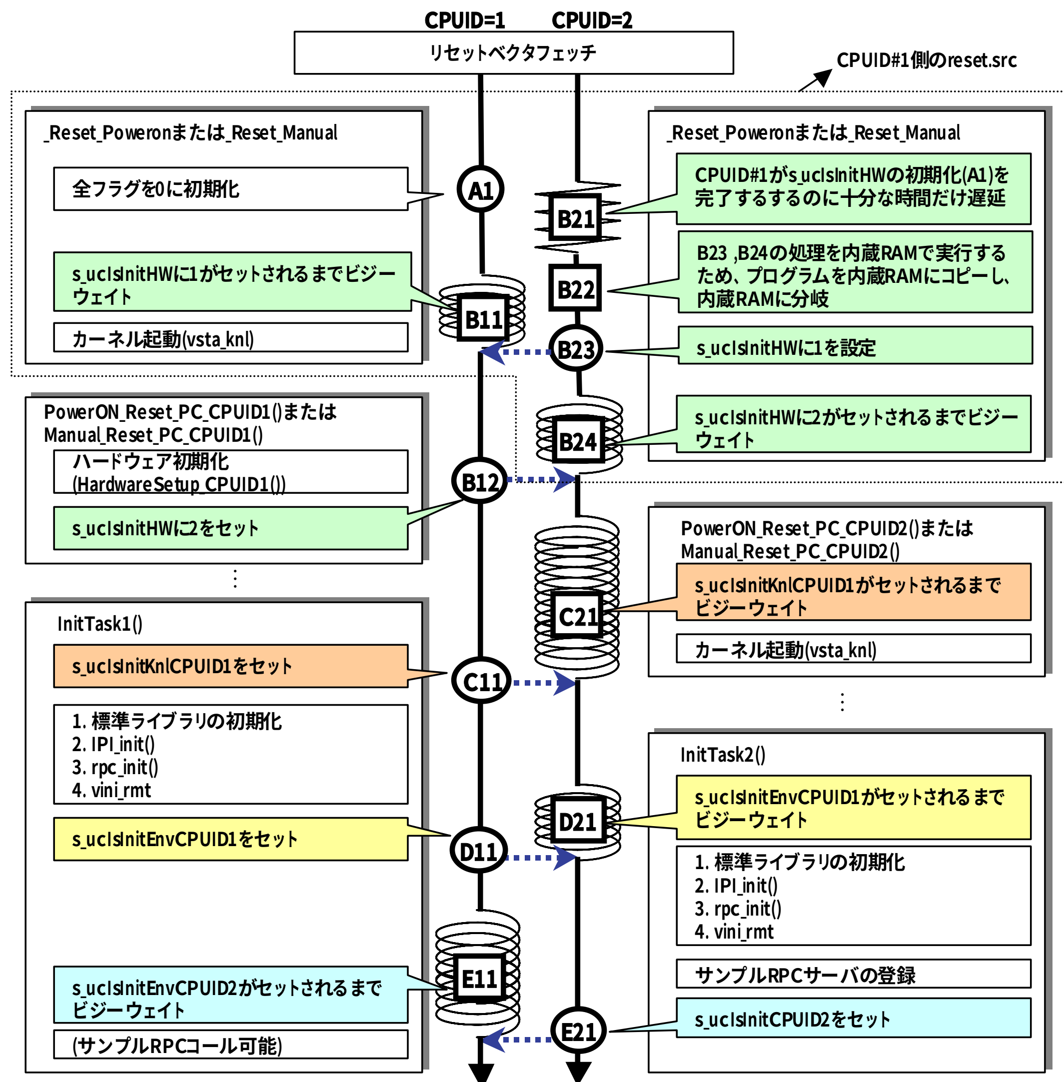
RPC を使用する場合、先に RPC コールを行う前に RPC サーバが登録される必要があります。

また、本サンプルでは該当しませんが、アプリケーション側で同様の順序性が必要になる場合もあります。

s_ucIsInitEnvCPUID2 は、この順序性を制御するためのフラグです。具体的には、CPUID#2 の初期起動タスクがサンプルの RPC サーバを登録するまで、CPUID#1 が RPC コールを行うことがないようにしています。

s_ucIsInitEnvCPUID2 の仕様は、以下の通りです。

- 0：初期状態
- 1：CPUID#2 の各種ソフトウェア環境の初期化処理が完了



【凡例】緑：s_ucIsInitHW に対するアクセス
橙色：s_ucIsInitKnlCPUID1 に対するアクセス
黄色：s_ucIsInitEnvCPUID1 に対するアクセス
水色：s_ucIsInitEnvCPUID2 に対するアクセス
A1, B11, B21, B22, B23, B24 ：「図 16.5　cpuid1¥reset¥reset.src」参照
B12：「図 16.6　cpuid1¥reset¥resetprg1.c」参照
C21：「図 16.8　cpuid2¥reset¥resetprg2.c」参照
C11, D11, E11：「図 16.9　cpuid1¥init¥init_task1.c」参照
D21, E21：「図 16.10　cpuid2¥init¥init_task2.c」参照

図16.11　スタートアップフェーズの同期

RENESAS

16.4 RPC 使用例

16.4.1 概要

RPC の例として、CPUID#1 側にクライアントの例、CPUID#2 側にサーバの例を提供しています。本節では、特にパラメータの受け渡し方法について解説します。

表 16.6にサーバ関数を示します。

表16.6 サーバ関数

項番	サーバ関数名	関数仕様
1	INT32 SampleAdd (INT32 lPar1, INT32 lPar2)	lPar1 と lPar2 の加算結果を返す。
2	UINT32 SampleStrlen (INT8 * pString)	pString が指す文字列の長さを返す。
3	void SampleSort1(INT32 * pData)	pData が指す要素数 10 の配列をソートする。
4	void SampleSort2(INT32 * pData)	pData が指す要素数 10 の配列をソートする。
5	void SampleMemcopy(void *pDest, const void *pSrc, UINT32 ulSize)	pSrc から ulSize バイトの内容を pDest にコピーする。
6	INT32 SampleCreateTask(void *entry , INT32 lPriority, UINT32 ulStackSize, void *UserData)	サーバ CPU 側にタスクを生成･起動する(TA_COP1 属性なし)
7	INT32 SampleKillTask(INT32 lTaskID)	サーバ CPU 側のタスクを強制終了し、削除する。
8	INT32 SampleRefTaskState (INT32 lTaskID, UINT32 *pulState)	サーバ CPU 側のタスクの状態を参照する。

表 16.7に、本節で解説するソースファイルを示します。

表16.7　RPC 使用例のソースファイル

ディレクトリ	ファイル	関数等
include¥	rpc_sample.h	RPC サーバ、クライアントの初期化 API"SampleInit()"と終了 API"SampleShutdown()"の定義
	sample_add.h	サーバ関数"SampleAdd()"の API 定義
	sample_strlen.h	サーバ関数"SampleStrlen()"の API 定義
	sample_sort.h	サーバ関数"SampleSort1()", "SampleSort2()"の API 定義
	sample_memcopy.h	サーバ関数"SampleMemcopy()"の API 定義
	sample_svc.h	サーバ関数" SampleCreateTask ()", SampleKillTask()", "SampleRefTaskState()"の API 定義
cpuid1¥ include¥	rpc_sample_clnt.h	クライアント用 API"SampleGetLastRPCErr()"の定義
cpuid1¥ rpc_sample_clnt¥	rpc_sample_clnt.c	RPC クライアントスタブ ・SampleAdd() ・SampleStrlen() ・SampleSort1() ・SampleSort2() ・SampleMemcopy() ・SampleCreateTask() ・SampleKillTask() ・SAMPLE_SampleRefTaskState() クライアント初期化 API 関数"SampleInit()" クライアント終了 API 関数"SampleShutdown()"
	rpc_sample_private.h *1	RPC クライアント・サーバスタブのプライベートヘッダ
cpuid1¥rpc_caller¥	rpc_caller.c	RPC コールを行う関数(タスク)
cpuid2¥ rpc_sample_svr¥	rpc_sample_svr.c	RPC サーバスタブ ・rpcsvr_SAMPLE_SampleAdd() ・rpcsvr_SAMPLE_SampleStrlen() ・rpcsvr_SAMPLE_SampleSort1() ・rpcsvr_SAMPLE_SampleSort2() ・rpcsvr_SAMPLE_SampleMemcopy() ・rpcsvr_SAMPLE_SampleCreateTask() ・rpcsvr_SAMPLE_SampleKillTask() ・rpcsvr_SAMPLE_SampleRefTaskState() サーバ初期化 API 関数"SampleInit()" サーバ終了 API 関数"SampleShutdown()"
	rpc_sample_private.h *1	RPC クライアント・サーバスタブのプライベートヘッダ
	sample_add.c	サーバ関数"SampleAdd()"
	sample_strlen.c	サーバ関数"SampleStrlen()"
	sample_sort.c	サーバ関数"SampleSort1()", "SampleSort2()", 内部関数"SampleSortMain()"
	sample_memcopy.c	サーバ関数"SampleMemcopy()"の API 定義
	sample_svc.c	サーバ関数" SampleCreateTask ()", SampleKillTask()", "SampleRefTaskState()"

【注】 *1 　ファイル内容は同じです。

RENESAS

16.4.2 RPC サーバの登録(CPUID#2)

本例では、前述したサーバ関数(のサーバスタブ)をひとつのサーバとして登録しています。

サーバの登録は、サーバ側の初期化 API"SampleInit()"で rpc_start_server()をコールすることで行っています。SampleInit()のコールは、初期起動タスクで行っています。

16.4.3 SampleAdd()

SampleAdd()は、INT32 型の入力パラメータを 2 つ(lPar1, lPar2)持っています。

クライアントスタブでは、IOVEC を 2 つ用意し、それぞれの IOVEC に入力パラメータのアドレスとサイズを設定しています。なお、入力 IOVEC を 2 つ使うのは、単に分かりやすさのためだけです。効率を考えると、IOVEC をひとつにしたほうがベターです。

サーバスタブでは、pInfo->pucParamArea の指す領域に、サーバ関数 API の順に入力パラメータが 4 バイトアラインされた状態で格納されているので、これを取り出してサーバ関数に渡します。

図 16.12にクライアントスタブ、図 16.13にサーバスタブのソースコードを示します。

1	/***/	
2	/** Client stub of "SampleAdd()"	
3	* @param lPar1 data1 to be added	
4	* @param lPar2 data2 to be added	
5	* @retval Result	
6	***/	
7	INT32 SampleAdd (INT32 lPar1, INT32 lPar2)	
8	{	
9	UINT32 ulLastOutputIOVectorSize;	
10	rpc_call_info info;	
11	IOVEC input[2];	入力 IOVEC を 2 つ用意
12	UINT32 ulReturn;	
13	INT32 lRPCRet;	
14		
15	info.ulMarshallingType = 0UL;	
16	info.ulServerID = SV_ID_SAMPLE;	
17	info.ulServerVersion = SV_VER_SAMPLE;	
18	info.ulServerProcedureID = RPC_SAMPLE_SAMPLEADD;	
19	info.AckMode = RPC_ACK;	
20	info.pInputIOVectorTable = input;	
21	info.ulInputIOVectorTableSize = sizeof(input) / sizeof (IOVEC);	
22	info.pOutputIOVectorTable = NULL;	出力なし
23	info.ulOutputIOVectorTableSize = 0UL;	
24	info.pulLastOutputIOVectorSize = &ulLastOutputIOVectorSize;	
25	info.pulReturnValue = &ulReturn;	リターン値設定領域
26		
27	input[0].pBaseAddress = &lPar1;	lPar1 情報の設定
28	input[0].ulSize = sizeof(INT32);	
29		
30	input[1].pBaseAddress = &lPar2;	lPar2 情報の設定
31	input[1].ulSize = sizeof(INT32);	
32		

図16.12 SampleAdd() (クライアントスタブ)

33
34
 	
35
36
37
38
39
40
	lRPCRet = rpc_call(&info);
	if(lRPCRet != RPC_E_OK) {
	
	    lLastErr = lRPCRet;
	    ulReturn = 0UL;
	}
	
	return ((INT32)ulReturn);
	}
		RPC コール
		RPC コールがエラーの場合、エラーコードを lLastErr に設定し、リターン値 0 でリターンする。

図 16.12　SampleAdd() (クライアントスタブ)(続き)

1	/***/	
2	/** Server stub of "SampleAdd()"	
3	* @param pInfo pointer to rpc_server_stub info structure	
4	* @retval Return value of server function	
5	***/	
6	static UINT32 rpcsvr_SAMPLE_SampleAdd (rpc_server_stub_info *pInfo)	
7	{	
8	INT32 lPar1;	
9	INT32 lPar2;	
10	INT32 lReturn;	
11		
12	lPar1 = *(INT32 *)(pInfo->pucParamArea);	サーバパラメータ領域から lPar1 を取得
13	lPar2 = *(INT32 *)(pInfo->pucParamArea + sizeof(INT32));	サーバパラメータ領域から lPar2 を取得
14		
15	lReturn = SampleAdd (lPar1, lPar2);	サーバ関数をコール
16		
17	pInfo->ulOutputIOVectorTableSize = 0UL;	出力なし
18		
19	return ((UINT32)lReturn);	
20	}	

図16.13　rpcsvr_SAMPLE_SampleAdd() (サーバスタブ)

RENESAS

16.4.4 SampleStrlen()

SampleStrlen ()は、ポインタの入力を持つ例です。
SapleStrlen()は、INT8*型の入力パラメータを 1 つ(pStrig)持っています。
クライアントスタブでは、IOVEC を 1 つ用意し、IOVEC に pString とその文字列のサイズを設定しています。
サーバスタブでは、pInfo->pucParamArea の指す領域に文字列が格納格納されているので、これを取り出してサーバ関数に渡します。
図 16.14にクライアントスタブ、図 16.15にサーバスタブのソースコードを示します。

1	/***/	
2	/** Client stub of "SampleStrlen()"	
3	* @param pString pointer to the string to be counted	
4	* @retval length of the string	
5	**/	
6	UINT32 SampleStrlen (INT8 * pString)	
7	{	
8	UINT32 ulLastOutputIOVectorSize;	
9	rpc_call_info info;	
10	IOVEC input[1];	
11	UINT32 ulReturn;	入力 IOVEC を 1 つ用意
12	INT32 lRPCRet;	
13		
14	info.ulMarshallingType = 0UL;	
15	info.ulServerID = SV_ID_SAMPLE;	
16	info.ulServerVersion = SV_VER_SAMPLE;	
17	info.ulServerProcedureID = RPC_SAMPLE_SAMPLESTRLEN;	
18	info.AckMode = RPC_ACK;	
19	info.pInputIOVectorTable = input;	
20	info.ulInputIOVectorTableSize = sizeof(input) / sizeof (IOVEC);	
21	info.pOutputIOVectorTable = NULL;	出力なし
22	info.ulOutputIOVectorTableSize = 0UL;	
23	info.pulLastOutputIOVectorSize = &ulLastOutputIOVectorSize;	
24	info.pulReturnValue = &ulReturn;	リターン値設定領域
25		
26	input[0].pBaseAddress = pString;	pString を設定
27	input[0].ulSize = strlen(pString) + 1UL;	この strlen()は標準ライブラリコールです。SampleStrlen()とは関係ありません。
28		
29	lRPCRet = rpc_call(&info);	RPC コール
30	if(lRPCRet != RPC_E_OK) {	RPC コールがエラーの場合、エラーコードを lLastErr に設定し、リターン値 0 でリターンする。
31	lLastErr = lRPCRet;	
32	ulReturn = 0UL;	
33	}	
34		
35	return ulReturn;	
36	}	

図16.14 SampleStrlen()(クライアントスタブ)

1	/**/	
2	/** Server stub of "SampleStrlen()"	
3	* @param pInfo pointer to rpc_server_stub info structure	
4	* @retval Return value of server function	
5	**/	
6	static UINT32 rpcsvr_SAMPLE_SampleStrlen (rpc_server_stub_info *pInfo)	
7	{	
8	INT8 * pString;	
9	UINT32 ulReturn;	
10		サーバパラメータ領域から pString を取得
11	pString = (INT8 *)(pInfo->pucParamArea);	
12		
13	ulReturn = SampleStrlen (pString);	サーバ関数をコール
14		
15	pInfo->ulOutputIOVectorTableSize = 0UL;	出力なし
16		
17	return (ulReturn);	
18	}	

図16.15　rpcsvr_SAMPLE_SampleStrlen() (サーバスタブ)

RENESAS

16.4.5 SampleSort1(), SampleSort2()

SampleSort1()と SampleSort2()はポインタの入力を持つ例で、まったく同じ機能を持ちます。
違うのは、SampleSort1()はポインタの指す内容をコピーしてサーバに渡す例であるのに対し、SampleSort2()はポインタそのものをサーバに渡す例になっている点です。SampleSort2()はコピー処理がないために高速ですが、ポインタは必ず非キャッシャブル領域を指さなければならないという制約があります。
前節までの解説を参考に、クライアントスタブ、サーバスタブを読んでみてください。

16.4.6 SampleMemcopy()

SampleMemcopy()も、入力ポインタそのものをサーバに渡す例です。

16.4.7 SampleCreateTask(), SampleKillTask(), SampleRefTaskState()

これらは、RPC を利用してカーネルのサービスコールをサーバ側のカーネルに対して要求できるように拡張する例にもなっています。
RPC という意味では、SampleRefTaskState()は出力パラメータを持つ例になっているので、これについて解説します。

SampleRefTaskState()は、UINT32*型の出力パラメータ(pulState)を持っています。
クライアントスタブでは、出力 IOVEC をひとつ用意し、IOVEC に pulState とそのサイズ(sizeof(UINT32))を設定しています。クライアント側の RPC が出力 IOVEC が指す領域にライトするので、pulState は非キャッシャブル領域を指している必要はありません。

サーバ側では、RPC は pInfo->pOutputIOVectorTable[0]を適切に設定(サーバパラメータ領域内を指す)してサーバスタブを呼び出します。サーバスタブはここから pulState を取り出し、サーバ関数を呼び出します。

図 16.16にクライアントスタブ、図 16.17にサーバスタブのソースコードを示します。

1	/***/	
2	/** Client stub of "SampleRefTaskState()"	
3	* @param lTaskID task ID to be referred	
4	* @param pulState pointer to be stored task state.	
5	* 0x00000001 .. RUNNING	
6	* 0x00000002 .. READY	
7	* 0x00000004 .. WAITING	
8	* 0x00000008 .. SUSPENDED	
9	* 0x0000000C .. WAITING and SUSPENDED	
10	* 0x00000010 .. DORMANT	
11	* 0x40000000 .. STACK-WAIT	
12	* @retval 0(success), Negative value(fail)	
13	***/	
14	INT32 SampleRefTaskState (INT32 lTaskID, UINT32 *pulState)	
15	{	
16	UINT32 ulLastOutputIOVectorSize;	
17	rpc_call_info info;	
18	IOVEC input[1];	入力 IOVEC を 1 つ用意
19	IOVEC output[1];	出力 IOVEC を 1 つ用意
20	UINT32 ulReturn;	
21	INT32 lRPCRet;	
22		
23	info.ulMarshallingType = 0UL;	
24	info.ulServerID = SV_ID_SAMPLE;	
25	info.ulServerVersion = SV_VER_SAMPLE;	
26	info.ulServerProcedureID = RPC_SAMPLE_SAMPLEREFTASKSTATE;	
27	info.AckMode = RPC_ACK;	
28	info.pInputIOVectorTable = input;	
29	info.ulInputIOVectorTableSize = sizeof(input) / sizeof (IOVEC);	
30	info.pOutputIOVectorTable = output;	
31	info.ulOutputIOVectorTableSize = sizeof(output) / sizeof (IOVEC);	
32	info.pulLastOutputIOVectorSize = &ulLastOutputIOVectorSize;	
33	info.pulReturnValue = &ulReturn;	
34		
35	input[0].pBaseAddress = &lTaskID;	入力として lTaskID を設定
36	input[0].ulSize = sizeof(INT32);	
37		
38	output[0].pBaseAddress = pulState;	出力として pulState を設定
39	output[0].ulSize = sizeof(UINT32);	
40		
41	lRPCRet = rpc_call(&info);	RPC コール
42	if(lRPCRet != RPC_E_OK) {	RPC コールがエラーの場合、エラーコードを lLastErr に設定し リターン値-1 でリターンする。
43	lLastErr = lRPCRet;	
44	ulReturn = 0xFFFFFFFFUL; /* return value = -1 */	
45	}	
46		
47	return ((INT32)ulReturn);	
48	}	

図16.16 SampleRefTaskState() (クライアントスタブ)

RENESAS

1	/***/	
2	/** Server stub of "SampleRefTaskState()"	
3	* @param pInfo pointer to rpc_server_stub info structure	
4	* @retval Size actually read	
5	**/	
6	static UINT32 rpcsvr_SAMPLE_SampleRefTaskState (rpc_server_stub_info *pInfo)	
7	{	
8	INT32 lTaskID;	
9	UINT32 *pulState;	
10	INT32 lRet;	
11		
12	lTaskID = *(INT32 *)(pInfo->pucParamArea);	サーバパラメータ領域から lTaskID を取得
13		
14	pulState = pInfo->pOutputIOVectorTable[0].pBaseAddress;	pulState を取得
15		
16	lRet = SampleRefTaskState (lTaskID, pulState);	サーバ関数をコール
17		
18	pInfo->ulOutputIOVectorTableSize = 1UL;	
19		
20	return ((UINT32)lRet);	
21	}	

図16.17 rpcsvr_SAMPLE_SampleRefTaskState() (サーバスタブ)

16.4.8 RPC 呼び出し例(CPUID#1)

cpuid1¥rpc_caller¥rpc_caller.c の TaskRpcCaller()は、本サンプルの RPC コールを順次行う例になっています。TaskRpcCaller()は、cfg ファイルにてタスクとして登録されています。

16.4.9 サーバの初期化と終了(CPUID#2)

本 RPC 例のサーバ側の環境を初期化する API 関数として SampleInit()、終了する API 関数として SampleShutdown()を用意しています。関数名はクライアント用と同じにしています。

出荷時の構成では、SampleInit()は初期起動タスクから呼び出しています。SampleShutdwon()は使用していません。

SampleInit()では、rpc_start_server()を用いてサーバを登録します。

SampleShutdwon()では、rpc_stop_server()を用いてサーバを削除します。

図 16.18に SampleInit()と SampleShutdwon()のソースコードを示します。

1	/**/	
2	/** Initialize RPC-Sample for server-side	
3	* @retval Return code of rpc_start_server()	
4	**/	
5	INT32 SampleInit (void)	
6	{	
7	/*** Server stub list ***/	
8	static UINT32 (* const rpcsvr_SAMPLE_StubTable[])(rpc_server_stub_info*) =	サーバスタブテーブル
9	{	
10	rpcsvr_SAMPLE_SampleAdd,	
11	rpcsvr_SAMPLE_SampleStrlen,	
12	rpcsvr_SAMPLE_SampleSort1,	
13	rpcsvr_SAMPLE_SampleSort2,	
14	rpcsvr_SAMPLE_SampleMemcopy,	
15	rpcsvr_SAMPLE_SampleCreateTask,	
16	rpcsvr_SAMPLE_SampleKillTask,	
17	rpcsvr_SAMPLE_SampleRefTaskState	
18	};	
19		
20	/*** Server information ***/	
21	static const rpc_server_info rpcsvr_SAMPLE_ServerInfo =	サーバ情報
22	{	
23	SV_ID_SAMPLE, /* ulRPCServerID */	
24	SV_VER_SAMPLE, /* ulRPCServerVersion */	
25	1UL, /* ServerStubTaskPriority */	
26	rpcsvr_SAMPLE_StubTable, /* ServerStubList */	
27	sizeof(rpcsvr_SAMPLE_StubTable) / sizeof(rpcsvr_SAMPLE_StubTable[0]),	
28	/* ulNumFunctions */	
29	0x400UL, /* ulStubStackSize */	
30	0UL, /* ulMaxParamAreaSize */	
31	NULL /* user_data */	
32	};	
33		
34	return rpc_start_server (&rpcsvr_SAMPLE_ServerInfo);	サーバの登録
35	}	
36		
37	/**/	
38	/** Shutdown RPC-Sample for server-side	
39	* @retval Return code of rpc_stop_server()	
40	**/	
41	INT32 SampleShutdown (void)	
42	{	
43	return rpc_stop_server(SV_ID_SAMPLE, SV_VER_SAMPLE, NULL, 0UL);	サーバの削除
44	}	

図16.18 SampleInit(), SampleShutdwon()(サーバ側)

RENESAS

16.4.10 クライアントの初期化と終了(CPUID#1)

本 RPC 例のクライアント側の環境を初期化する API 関数として SampleInit()、終了する API 関数として SampleShutdown()を用意しています。出荷時の構成では、SampleInit()は初期起動タスクから呼び出しています。SampleShutdwon()は使用していません。

SampleInit()では、rpc_connect()を用いてサーバと接続します。

SampleShutdwon()では、rpc_disconnect()を用いてサーバとの接続を切断します。

図 16.19に SampleInit()と SampleShutdwon()のソースコードを示します。

1	/**/	
2	/** Initialize RPC-Sample for client-side	
3	* @retval Return code of rpc_connect()	
4	**/	
5	INT32 SampleInit (void)	
6	{	
7	return rpc_connect(SV_ID_SAMPLE, SV_VER_SAMPLE);	サーバと接続
8		
9	}	
10		
11	/**/	
12	/** Shutdown RPC-Sample for client-side	
13	* @retval Return code of rpc_disconnect()	
14	**/	
15	INT32 SampleShutdown (void)	
16	{	
17	return rpc_disconnect(SV_ID_SAMPLE, SV_VER_SAMPLE, NULL, 0);	サーバとの接続を切断
18	}	
19	/**/	

図16.19　SampleInit(), SampleShutdwon()(**クライアント側**)

16.4.11 RPC ライブラリの初期化(rpc_init()コール)

クライアント側、サーバ側、ともに初期起動タスクから rpc_init()をコールしています。

rpc_init()に渡す rpc_config 構造体の内容は、クライアント側は図 16.9、サーバ側は図 16.10を参照してください。

16.5 リモートサービスコール使用例

リモートサービスコールの例として、メールボックスと固定長メモリプールを使用した簡単なメッセージ通信の例を提供しています。

もう少し具体的には、CPUID#1 側のタスクが CPUID#2 側の固定長メモリプールからメッセージ領域を取得し、CPUID#2 側のメールボックスにメッセージを送信します。CPUID#2 側のタスクはメールボックスからメッセージを受信し、メッセージ領域を固定長メモリプールに返却します。

メールボックスと固定長メモリプールは、CPUID#2 側の cfg ファイルによって生成しています。このとき、"export=ON"の指定により、その ID 名称を kernel_id_cpu2.h に出力するようにしています。CPUID#1 側のファイルでは、この kernel_id_cpu2.h をインクルードしています。

また、メッセージは両 CPU からアクセスするため、固定長メモリプール領域は非キャッシャブル領域に割り当てています。

表 16.8に、リモートサービスコール使用例のソースファイルを示します。

表16.8 リモートサービスコール使用例のソースファイル

ディレクトリ	ファイル	関数等
include¥	user_msg.h	**メッセージ形式の定義**
cpuid1¥ remote_svc_sample¥	remote_send.c	**メッセージを送信するタスク**(TaskSend())
cpuid2¥ remote_svc_sample¥	remote_recv.c	**メッセージを受信するタスク**(TaskRecv())

図 16.20に CPUID#1 側のメッセージ送信タスク、図 16.21に CPUID#2 側のメッセージ受信タスクのソースコードを示します。

```
1   /***********************************************************************************
2   /** Sample task that send message by using remote service call
3    ***********************************************************************************/
4   void TaskSend(VP_INT exinf)
5   {
6       USER_MSG     *message;
7       UINT32       ulIndex;
8
9       for(ulIndex = 0UL ; ; ulIndex++) {
10          /* get non-cached message area */
11          if(pget_mpf(EXID2_MPF_NONCACHED, (VP *)&message) != E_OK) {   メッセージ領域を固定長メモリプールから取得
12              ext_tsk();
13          }
14
15          /* create message */                                       メッセージを設定
16          message->osarea.msghead = NULL;
17          message->ulData = ulIndex;
18
19          /* send message to CPU2 */
20          snd_mbx(EXID2_MBX_COMM, (T_MSG *)&message);                メッセージを送信
21      }
22  }
```

図16.20 メッセージを送信するタスク(cpuid1¥remote_svc_sample¥sample_send.c)

1	/**	
2	/** Sample task that receives message.	
3	**/	
4	void TaskRecv(VP_INT exinf)	
5	{	
6	USER_MSG *p_message;	
7	UINT32 ulIndex;	
8		
9	while(TRUE) {	
10	/* receive message from CPU1 */	
11	if(rcv_mbx(EXID2_MBX_COMM, (T_MSG **)&p_message) != E_OK) {	メッセージを受信
12	ext_tsk();	
13	}	
14		
15	/* do operation according to message */	メッセージに応じた処理
16		
17	/* release memory */	
18	rel_mpf(EXID2_MPF_NONCACHED, (VP)p_message);	メッセージ領域を解放
19	}	
20	}	

図16.21 **メッセージを受信するタスク**(cpuid2¥remote_svc_sample¥sample_recv.c)

16.6 タイマドライバ

提供するタイマドライバは、SH7205, SH7265 に搭載されている CMT 用です。
CPUID#1 はチャネル 0 を、CPUID#2 はチャネル 1 を使用します。

表 16.9に、タイマドライバのソースファイルを示します。

表16.9 タイマドライバのソースファイル

ディレクトリ	ファイル	関数等
cpuid1¥ os_timer¥	tmrdrv.c *1	SH7205, SH7265 内蔵 CMT 用タイマドライバ
	tmrdrv.h *2	内部定義
cpuid2¥ os_timer¥	tmrdrv.c *1	SH7205, SH7265 内蔵 CMT 用タイマドライバ
	tmrdrv.h *2	内部定義

【注】 *1 ファイル内容は同じです。
*2 ファイル内容は同じです。

各ファイル内容は両 CPU で同じです。MYCPUID によって条件コンパイルするように実装しています。

16.7 標準ライブラリ

16.7.1 概要

本サンプルでは、両 CPU とも以下の仕様で標準ライブラリを組み込んでいます。

- 組み込む機能：stdlib.h, string.h
- リエントラントライブラリとして構築

なお、コンパイラマニュアルにも記載があるように、リエントラントライブラリを使用する場合は、標準ライブラリを使用するアプリケーションでは、標準インクルードファイルをインクルードする前に、"_REENTRANT"というマクロ名を#define 文で定義する(#define _REENTRANT)するか、コンパイル時に define オプションで_REENTRANT を定義する必要があることに注意してください。提供する High-performance Embedded Workshop プロジェクトでは、後者の方法を採用しています。

また、stdio.h を組み込んでいないことにも注意してください。stdio.h を使用するには、それに対応した低水準インタフェースルーチンの追加が必要です。

表 16.10に、標準ライブラリ関連のソースファイルを示します。

表16.10 標準ライブラリ関連のソースファイル

ディレクトリ	ファイル *	関数等
cpuid1¥stdlib¥	lowsrc.c	低水準インタフェースルーチン、_INIT_LOWLEVEL()
	lowsrc_config.h	低水準インタフェースルーチンのコンフィギュレーションファイル
	otherlib.c	_INIT_OTHERLIB()
	initsct.c	_INITSCT()
cpuid1¥include¥	lowsrc.h	lowsrc.c 外部ヘッダ
	initsct.h	initsct.c 外部ヘッダ
cpuid2¥stdlib¥	lowsrc.c	低水準インタフェースルーチン、_INIT_LOWLEVEL()
	lowsrc_config.h	低水準インタフェースルーチンのコンフィギュレーションファイル
	otherlib.c	_INIT_OTHERLIB()
	initsct.c	_INITSCT()
cpuid2¥include¥	lowsrc.h	lowsrc.c 外部ヘッダ
	initsct.h	initsct.c 外部ヘッダ

【注】 同名ファイルの内容は同じです。

16.7.2 低水準インタフェースルーチン

標準入出力、メモリ管理ライブラリを使用する場合、およびリエントラントライブラリを使用する場合は、各種の低水準インタフェースルーチンを作成する必要があります。

表 16.11に、コンパイラが規定している低水準インタフェースルーチンと、本サンプルで実装している低水準インタフェースルーチンを示します。

表16.11 低水準インタフェースルーチン

コンパイラ規定の 低水準インタフェースルーチン	本サンプルでの実装	機能
open()	×	ファイルのオープン
close()	×	ファイルのクローズ
read()	×	ファイルからの読み込み
write()	×	ファイルへの書き出し
lseek()	×	ファイルの読み込み/書き込み位置の設定
sbrk()	○	メモリ領域の確保
sbrk__X()	×	X メモリ領域の確保(DSP 内蔵マイコン用)
sbrk__Y()	×	Y メモリ領域の確保(DSP 内蔵マイコン用)
errno_adr() *	○	errno アドレスの取得
wait_sem() *	○	セマフォの確保
signal_sem() *	○	セマフォの解放

【注】 リエントラントライブラリを使用する場合に必要です。

wait_sem(), signal_sem()は、排他制御を行うための低水準インタフェースルーチンです。なお、ここでの「セマフォ」は標準ライブラリとしての用語であり、HI7200/MP のセマフォとは関係ありません。

wait_sem(), signal_sem()では、カーネルのミューテックスを使用して排他制御を行っています。

16.7.3 標準ライブラリ環境の初期化(_INIT_LOWLEVEL(), _INIT_OTHERLIB())

本サンプルでは、低水準インタフェースルーチンを初期化する_INIT_LOWLEVEL()と、strtok(), rand()の初期化を行う_INIT_OTHERLIB()を用意しています。

これらの初期化関数は、両 CPU とも初期起動タスクから呼び出すようにしています。

16.7.4 セクション初期化(_INITSCT())

標準的な_INITSCT()は、コンパイラの標準ライブラリによって提供されますが、本サンプルではこれを使わず、独自に_INITSCT()を実装しています。これは、標準ライブラリの_INITSCT()を使用すると、標準ライブラリ自身のコードを ROM から RAM に転送する手順が複雑になるためです。

また、サンプルの_INITSCT()は、標準ライブラリの_INITSCT()に対して、以下のように引数を追加することで、任意のタイミングで任意のセクションを初期化できるようにしています。

- 標準ライブラリ：void _INITSCT(void);
- 本サンプル：void _INITSCT(

```
const ST_DTBL  *dtbl_top,   // ST_DTBL配列の先頭アドレス
UINT32  dtbl_sz,            // ST_DTBL配列のサイズ(バイト数)
const ST_BTBL  *btbl_top,   // ST_BTBL配列の先頭アドレス
UINT32  btbl_sz );          // ST_BTBL配列のサイズ(バイト数)
```

各構造体の構造は以下の通りです。

```
typedef struct {  // 転送を行うセクションの初期化情報
          UINT8   *SecD_Start;  // 転送元セクションの先頭アドレス
          UINT32  SecD_Size;    // 転送元セクションのサイズ
          UINT8   *SecR_Start;  // 転送先セクションの先頭アドレス
} ST_DTBL;

typedef struct {  // 0クリアを行うセクションの初期化情報
          UINT8   *SecB_Start;  // セクションの先頭アドレス
          UINT32  SecB_Size;    // セクションのサイズ
} ST_BTBL;
```

また、これらの引数を容易に指定するための以下のマクロがあります。これらのマクロは、initsct.h で定義されています。

- MACRO_ENTRY_DTBL(dname, rname)
  ”dname”セクションを”rname”セクションに転送するためのST_DTBLを生成
- MACRO_ENTRY_BTBL(bname)
  ”bname”セクションを0クリアするためのST_BTBLを生成

16.7.5 標準ライブラリのコンフィギュレーション(lowsrc_config.h)

以下を参考に、各 define 文を設定してください。

```
1   /***************************************************************************
2    * Defines
3    ***************************************************************************/
4   /* size of area managed by sbrk */
5   #define HEAPSIZE 0x400UL
6
7   /* Mutex ceiling priority for wait_sem (only for reentrant library) */
8   #ifdef _REENTRANT
9   #define PRI_SBRK   1  /* for sbrk() */
10  #define PRI_S1PTR  1  /* for _s1ptr(strtok()) */
11  #define PRI_IOB    1  /* for iob */
12  #endif
13
14  /* Mutex timeout for wait_sem (only for reentrant library) */
15  #ifdef _REENTRANT
16  #define SEM_TMOUT  30000L /* 30000 msec */
17  #endif
18
19  /* Number of tasks for errno (only for reentrant library) */
20  #ifdef _REENTRANT
21  #define NUM_TASK   _MAX_TSK
22  #endif
```

図16.22 lowsrc_config.h

表16.12 標準ライブラリコンフィギュレーションファイルの設定項目

項目	解説
HEAPSIZE	sbrk()が管理するヒープ領域のサイズ
PRI_SBRK *	sbrk()の排他制御用に使用するミューテックスの上限優先度
PRI_S1PTR *	_s1ptr の排他制御用に使用するミューテックスの上限優先度
PRI_IOB *	iob の排他制御用に使用するミューテックスの上限優先度
SEM_TMOUT *	ミューテックスのロック取得時のタイムアウト
NUM_TASK *	最大ローカルタスク ID

【注】 リエントラントライブラリを使用する場合に必要です。

16.7.6 ソースコード

(1) lowsrc.c (低水準インタフェースルーチン、_INIT_LOWLEVEL())

1	/**	
2	* Include	
3	**/	
4	#include <stddef.h>	
5		
6	#include "kernel.h"	
7	#include "types.h"	
8		
9	#include "lowsrc.h"	
10	#include "lowsrc_config.h"	
11		
12	/**	
13	* Heap area	
14	**/	
15	#pragma section C_heap	ヒープ領域固有セクション名を付与
16	static union { /* memory-pool area */	
17	INT32 dummy ; /* Dummy for 4-byte boundary */	
18	INT8 heap[ALIGNUP4(HEAPSIZE)]; /* Declaration of the area managed */	
19	/* by sbrk */	
20	}heap_area ;	
21	#pragma section	
22		
23		
24	/**	
25	* Prototypes	
26	**/	
27	INT8 *sbrk(size_t size);	
28	#ifdef _REENTRANT	
29	INT wait_sem(INT semnum);	
30	INT signal_sem(INT semnum);	
31	INT *errno_addr(void);	
32	#endif	
33		
34	static INT sbrk_init(void);	
35	static INT sem_init(void);	
36	static void errno_init(void);	
37		
38		
39		
40	/**	
41	* Section	
42	**/	
43	#pragma section C_stdlib	セクション名定義
44		
45		
46	/**	
47	* Data	
48	**/	

図16.23 lowsrc.c

RENESAS

49	/*** for sbrk() ***/	
50	static ID sbrk_mplid; /* memory-pool ID */	sbrk()で使用する可変長メモリプールIDを保持する変数
51		
52	/*** for semaphore ***/	
53	#ifdef _REENTRANT	
54	#define NUM_SEM 3	
55		
56	#define SEM_SBRK 1 /* semnum for sbrk() */	標準ﾗｲﾌﾞﾗﾘ側で規定されているsbrk()用セマフォ番号
57	#define SEM_S1PTR 2 /* semnum for _s1ptr(strtok()) */	標準ﾗｲﾌﾞﾗﾘ側で規定されている_s1ptr用セマフォ番号
58	#define SEM_IOB 3 /* semnum for iob */	標準ﾗｲﾌﾞﾗﾘ側で規定されているiob)用セマフォ番号
59		
60	static ID mtx_id[NUM_SEM]; /* mutex ID for each semnum */	wait_sem()で使うミューテックスIDを保持する変数
61	#endif /* end of _REENTRANT */	
62		
63	/*** for errno ***/	
64	#ifdef _REENTRANT	
65	static INT errno_context[NUM_TASK+1];	タスク毎のerrno領域(タスクIDがindex)。Index=0は非タスクコンテキスト用。
66	#endif /* end of _REENTRANT */	
67		
68	/**	
69	/** Allocate memory	
70	* @param size Required memory size	
71	* @retval Pointer to allocated memory(Pass), -1(Failure)	
72	**/	
73	INT8 *sbrk(size_t size)	sbrk()関数
74	{	
75	ER ercd;	
76	INT8 *p;	
77		
78	if(sbrk_mplid != 0) {	
79	ercd = pget_mpl(sbrk_mplid, ALIGNUP4(size), (VP *)&p);	可変長メモリプールからメモリを取得
80	if(ercd != E_OK) {	
81	p = (INT8 *)(-1);	
82	}	
83	}	
84	else {	
85	p = (INT8 *)(-1);	
86	}	
87		
88	return p;	
89	}	
90		
91		

図 16.23 lowsrc.c(続き)

92	/**	
93	/** Initialize sbrk environment	
94	* @param None	
95	* @retval 1(Pass), 0(Failure)	
96	**/	
97	static INT sbrk_init(void)	sbrk_init()関数
98	{	
99	ER_ID mplid;	
100	INT rtn;	
101		
102	static const T_CMPL cmpl = {	可変長メモリプール生成情報
103	TA_TFIFO,	
104	sizeof(heap_area.heap),	
105	(VP)(heap_area.heap),	
106	#if ((VTCFG_NEWMPL) == _NEW)	
107	NULL,	
108	0U,	
109	0U	
110	#endif	
111	};	
112		
113	mplid = acre_mpl(&cmpl);	ヒープ領域を可変長メモリプールとして生成
114	if(mplid > 0L) {	
115	sbrk_mplid = mplid;	
116	rtn = 1;	
117	}	
118	else {	
119	sbrk_mplid = 0;	エラー時は、sbrk_mplid を 0
120	rtn = 0;	にする。
121	}	
122		
123	return rtn;	
124	}	
125		
126		
127	/**	
128	/** Get semaphore	
129	* @param semnum Semaphore number 1(malloc), 2(strtok), 3(iob)	
130	* @retval 1(Pass), 0(Failure)	
131	* @Note When calling from non-task context, this function returns error.	
132	**/	
133	#ifdef _REENTRANT	
134	INT wait_sem(INT semnum)	
135	{	wait_sem()関数
136	INT rtn;	
137	ID mtxid;	
138		
139	mtxid = mtx_id[semnum-1];	セマフォ番号に対する
140		ミューテックス ID を求める
141	rtn = 0;	

図 16.23 lowsrc.c(続き)

142		
143	`    if(mtxid != 0) {`	
144	`        if(tloc_mtx( mtxid, SEM_TMOUT) == E_OK) {`	ミューテックスをロック
145	`            rtn = 1;`	
146	`        }`	
147	`    }`	
148	`    return rtn;`	
149	`}`	
150	`#endif   /* end of _REENTRANT */`	
151		
152		
153	`/*****************************************************************************`	
154	`/**   Release semaphore`	
155	` *     @param semnum Semaphore number 1(malloc), 2(strtok), 3(iob)`	
156	` *     @retval 1(Pass), 0(Failure)`	
157	` *     @Note When calling from non-task context, this function returns error.`	
158	` *****************************************************************************/`	
159	`#ifdef _REENTRANT`	
160	`INT signal_sem(INT semnum)`	signal_sem()関数
161	`{`	
162	`    INT rtn;`	
163	`    ID   mtxid;`	
164		
165	`    mtxid = mtx_id[semnum-1];`	セマフォ番号に対するミ
166		ューテックス ID を求める
167	`    if(mtxid != 0) {`	
168	`        if(unl_mtx( mtxid ) == E_OK) {`	ミューテックスをアンロック
169	`            rtn = 1;`	
170	`        }`	
171	`    }`	
172	`    return rtn;`	
173	`}`	
174	`#endif   /* end of _REENTRANT */`	
175		
176		
177	`/*****************************************************************************`	
178	`/**   Initialize wait_sem/signal_sem environment`	
179	` *     @param None`	
180	` *     @retval 1(Pass), 0(Failure)`	
181	` *****************************************************************************/`	
182	`#ifdef _REENTRANT`	
183	`static INT sem_init(void)`	sem_init()関数
184	`{`	
185	`    ER_ID    mtxid;`	
186	`    INT      rtn, i;`	
187	`    static const T_CMTX cmtx[NUM_SEM] = {`	ミューテックス生成情報
188	`        {TA_CEILING, PRI_SBRK},      /* for sbrk() */`	
189	`        {TA_CEILING, PRI_S1PTR},     /* for _s1ptr(strtok()) */`	
190	`        {TA_CEILING, PRI_IOB}        /* for iob */`	
191	`    };`	
192		

図 16.23　lowsrc.c(続き)

RENESAS

193	for(i = 0 ; i < NUM_SEM ; i++) {	
194	mtx_id[i] = 0;	mtx_id[]を 0 クリア
195	}	
196		
197	rtn = 1;	
198	for(i = 0 ; i < NUM_SEM ; i++) {	各ミューテックスを生成
199	mtxid = acre_mtx(&cmtx[i]);	
200	if(mtxid > 0L) {	
201	mtx_id[i] = mtxid;	生成に成功したら、mtx_id[]に
202	}	その ID を設定
203	else {	
204	rtn = 0;	
205	break;	
206	}	
207	}	
208		
209	return rtn;	
210	}	
211	#endif /* end of _REENTRANT */	
212		
213		
214		
215	/**	
216	/** Return "errno" address for cuurent context	
217	* @param None	
218	* @retval Pointer to "errno"	
219	**/	
220	#ifdef _REENTRANT	
221	INT *errno_addr(void)	errno_addr()関数
222	{	
223	INT *rtn;	
224	ER ercd;	
225	ID id;	
226		
227	if(sns_ctx() == FALSE) {	
228	/* Case task context */	
229	get_tid(&id);	
230	id = GET_LOCALID(id);	
231	}	
232	else {	
233	/* Case non-task context */	
234	id = 0;	
235	}	
236	return (&errno_context[id]);	
237	}	
238	#endif /* end of _REENTRANT */	
239		
240		
241	/**	
242	/** Initialize errno environment	
243	* @param None	
244	* @retval None	

図 16.23 lowsrc.c(続き)

RENESAS

245	**/	
246	#ifdef _REENTRANT	
247	static void errno_init(void)	errno_init()関数
248	{	
249	INT i;	
250		
251	for(i= 0 ; i < sizeof(errno_context)/sizeof(INT) ; i++) {	
252	errno_context[i] = 0;	
253	}	
254	}	
255	#endif /* end of _REENTRANT */	
256		
257		
258		
259	/**	
260	/** Initialize low-level environment	
261	* @param None	
262	* @retval 1(Pass), 0(Failure)	
263	**/	
264	INT _INIT_LOWLEVEL(void)	_INIT_LOWLEVEL()関数
265	{	
266	INT rtn;	
267		
268	/* initialize sbrk */	
269	rtn = sbrk_init();	
270	if(rtn == 0) {	
271	return rtn;	
272	}	
273		
274	/* initialize semaphore */	
275	#ifdef _REENTRANT	
276	rtn = sem_init();	
277	if(rtn == 0) {	
278	return rtn;	
279	}	
280	#endif /* end of _REENTRANT */	
281		
282	/* initialize errno */	
283	#ifdef _REENTRANT	
284	errno_init();	
285	#endif /* end of _REENTRANT */	
286		
287	return rtn;	
288	}	

図 16.23　lowsrc.c(続き)

(2) otherlibc (_INIT_OTHERLIB())

_INIT_OTHERLIB()では、strtok()が使用する大域変数"_s1ptr"の初期化と、srand()のコールを行います。なお、ソースコードの掲載は省略します。

(3) initsct.c (_INITSCT())

```
1   /*******************************************************************************
2    * Include
3    *******************************************************************************/
4   #include "types.h"
5
6   #include "initsct.h"
7
8
9   /*******************************************************************************
10   * Section
11   *******************************************************************************/
12  #pragma section C_stdlib                                                          セクション名を設定
13
14
15  /*******************************************************************************
16  /**   Initialize sections                                                         _INITSCT()
17   *     @param dtbl_top information table address of D section
18   *     @param dtbl_sz size of section of information table
19   *     @param btbl_top information table address of B section
20   *     @param btbl_sz size of section of information table
21   *     @retval None
22   *******************************************************************************/
23  void _INITSCT(
24      const ST_DTBL    *dtbl_top,
25      UINT32           dtbl_sz,
26      const ST_BTBL    *btbl_top,
27      UINT32           btbl_sz
28      )
29  {
30      const ST_DTBL    *dtbl;
31      const ST_BTBL    *btbl;
32      UINT32   tblcnt;
33      UINT32   sz;
34      UINT8    *rp, *wp;
35
36      /*** Copy D-section to R-section ***/
37      dtbl = dtbl_top;
38      tblcnt = dtbl_sz/sizeof(ST_DTBL);
39
40      while(tblcnt > 0UL) {
41          rp = dtbl->SecD_Start;
42          wp = dtbl->SecR_Start;
43          sz = dtbl->SecD_Size;
44
45          while(sz > 0UL) {
46              *wp = *rp;       /* Copy D --> R */
47              rp++;
48              wp++;
49              sz--;
```

図16.24 initsct.c

```
50          }
51          dtbl++;
52          tblcnt--;
53      }
54
55      /*** 0-clear B-section ***/
56      btbl = btbl_top;
57      tblcnt = btbl_sz/sizeof(ST_BTBL);
58
59      while(tblcnt > 0UL) {
60          wp = btbl->SecB_Start;
61          sz = btbl->SecB_Size;
62
63          while(sz > 0UL) {
64              *wp = 0U;       /* 0-clear */
65              wp++;
66              sz--;
67          }
68          btbl++;
69          tblcnt--;
70      }
71  }
```

図 16.24　initsct.c（続き）

16.8 ダミーオブジェクト

16.8.1 ダミープログラム

本サンプルでは、ユーザのコーディング時の雛形、および cfg ファイルの記述例の意味で、いくつかのダミープログラムを用意しています。

表 16.13に、ダミープログラムのソースファイルを示します。両 CPU のダミープログラムの内容は同じです。また、これらのダミープログラムは、cfg ファイルによってカーネルに登録されています。

表16.13 ダミープログラムのソースファイル

ディレクトリ	ファイル	関数等
cpuid1¥dummy_prog¥	dummy_prog.c	CPUID#1 用ダミープログラム • DummyTask() (タスク) • DummyCyclicHandler() (周期ハンドラ) • DummyAlarmHandler() (アラームハンドラ) • DummyExtendedSVC() (拡張サービスコールルーチン) • DummyInitRoutine() (初期化ルーチン) • DummyNormalIntHandler508() (ノーマル割込みハンドラ) • DummyDirectIntHandler509() (ダイレクト割込みハンドラ)
cpuid2¥dummy_prog¥	dummy_prog.c	CPUID#2 用ダミープログラム • DummyTask() (タスク) • DummyCyclicHandler() (周期ハンドラ) • DummyAlarmHandler() (アラームハンドラ) • DummyExtendedSVC() (拡張サービスコールルーチン) • DummyInitRoutine() (初期化ルーチン) • DummyNormalIntHandler510() (ノーマル割込みハンドラ) • DummyDirectIntHandler511() (ダイレクト割込みハンドラ)

(1) ダミータスク(DummyTask())とダミー拡張サービスコール(DummyExtendedSVC())

ダミータスクでは、ダミー拡張サービスコールを呼び出します。ダミー拡張サービスコールルーチンは、何もせずにリターンします。ダミー拡張サービスコールの機能コードは 1 です。

(2) ダミー周期ハンドラ(DummyCyclicHandler())

一定周期で起動されますが、何もせずにリターンします。

(3) ダミーアラームハンドラ(DummyAlarmHandler())

一度だけ起動され、何もせずにリターンします。

(4) ダミー初期化ルーチン(DummyInitRoutine())

何もせずにリターンします。

(5) ダミーノーマル割込みハンドラ (DummyNormalIntHandler508(),DummyNormalIntHandler510())

CPUID#1 ではベクタ番号 508、CPUID#2 ではベクタ番号 510 に対して、それぞれダミーノーマル割込みハンドラを定義しています。

これらのハンドラは、何もせずにリターンするように作られています。ただし、これらの割込みが発生することはありません。

(6) ダミーダイレクト割込みハンドラ (DummyDirectIntHandler509(),DummyDirectIntHandler511())

CPUID#1 ではベクタ番号 509、CPUID#2 ではベクタ番号 511 に対して、それぞれダミーダイレクト割込みハンドラを定義しています。

これらのハンドラは、何もせずにリターンするように作られています。ただし、これらの割込みが発生することはありません。

16.8.2 その他のダミーオブジェクト

ダミープログラムと同様に、cfg ファイルの記述例の意味で、いくつかのカーネルオブジェクトを登録しています。詳細は、サンプルの cfg ファイルの内容を確認してください。

16.9 I/O レジスタ定義、周辺クロック定義、kernel_intspec.h

本サンプルでは、表 16.14に示す I/O 定義ファイルを提供しています。

表16.14 I/O 定義ファイル

ディレクトリ	ファイル	関数等	
iodefine¥	kernel_intspec.h	CPU の割込みハードウェア仕様定義	
	pclock.h	周辺クロック周波数の定義	
	io_bsc.h	SH7265 内蔵周辺レジスタの定義	BSC(バスステートコントローラ)
	io_cmt.h		CMT(コンペアマッチタイマ)
	io_cpg.h		CPG(クロクパルス発振器)
	io_intc.h		INTC(割込みコントローラ)
	io_multicore.h		マルチコア関連
	io_port.h		I/O ポート
	io_stb.h		低消費電力関連
	io_sys.h		システムコントロール関連
	io_wdt.h		WDT(ウォッチドッグタイマ)

kernel_intspec.h は、カーネルに割込みハードウェア仕様を教えるための重要なファイルです。詳細は、「17.3 CPU 割込み仕様定義ファイル(kernel_intspec.h)の作成」を参照してください。

また、pclock.h では周辺クロック周波数を定義しています。タイマドライバでは、この定義を使用しています。

RENESAS

16.10 カーネルオブジェクト一覧

本サンプルでは、cfg ファイルおよびサンプルプログラム中でのサービスコールによって、各オブジェクトを生成しています。

16.10.1 タスク

表16.15 タスク(CPUID#1)

分類	生成・起動方法	ID 名称	優先度
初期起動タスク	cfg ファイルの task[]	ID1_TASK_INIT	1
SVC サーバタスク	cfg ファイルの remote_svc.num_server の数だけ、vini_rmt によって生成 *1	(なし)	1
リモートサービスコール使用例 *2	cfg ファイルの task[]	ID1_TASK_SEND	3
RPC コールするタスク	cfg ファイルの task[]	ID1_TASK_RPCCALLER	5
ダミータスク	cfg ファイルの task[]	ID1_TASK_DUMMY	10

【注】 *1 出荷時の remote_svc.num_server 設定値は 3 です。

*2 このタスクのファイル (<SAMPLE_INST>¥R0K572650D000BR¥cpuid1¥remote_svc_sample¥remote_send.c)では、CPUID#2 側のオブジェクト ID を参照するために、CPUID#2 側で cfg72mp によって生成された kernel_id_cpu2.h をインクルードしています。

表16.16 タスク(CPUID#2)

分類	生成・起動方法	ID 名称	優先度
初期起動タスク	cfg ファイルの task[]	ID2_TASK_INIT	1
SVC サーバタスク	cfg ファイルの remote_svc.num_server の数だけ、vini_rmt によって生成 *1	(なし)	1
リモートサービスコール使用例	cfg ファイルの task[]	ID2_TASK_RECV	3
RPC 例のサーバタスク	rpc_start_server()によって生成	(なし)	5
ダミータスク	cfg ファイルの task[]	ID2_TASK_DUMMY	10

【注】 *1 出荷時の remote_svc.num_server 設定値は 3 です。

16.10.2 その他のオブジェクト

表16.17 その他のオブジェクト(CPUID#1)

オブジェクト種別	分類	生成方法	ID 名称
セマフォ	ダミー	cfg ファイルの semaphore[]	ID1_SEM_DUMMY
イベントフラグ	ダミー	cfg ファイルの flag[]	ID1_FLG_DUMMY
データキュー	ダミー	cfg ファイルの dataqueue[]	ID1_DTQ_DUMMY
メールボックス	ダミー	cfg ファイルの mailbox[]	ID1_MBX_DUMMY
ミューテックス	sbrk()排他制御用	_INIT_LOWLEVEL()からの acre_mtx コール	(なし)
	_s1ptr 排他制御用	_INIT_LOWLEVEL()からの acre_mtx コール	(なし)
	iob 排他制御用	_INIT_LOWLEVEL()からの acre_mtx コール	(なし)
メッセージバッファ	ダミー	cfg ファイルの message_buffer[]	ID1_MBF_DUMMY
固定長メモリプール	ダミー	cfg ファイルの memorypool[]	ID1_MPF_DUMMY
可変長メモリプール	ダミー	cfg ファイルの variale_memorypool[]	ID1_MPL_DUMMY
	OAL 用	OAL_Init()からの acre_mpl コール	―
周期ハンドラ	ダミー	cfg ファイルの cyclic_hand[]	ID1_CYC_DUMMY
アラームハンドラ	ダミー	cfg ファイルの alarm_hand[]	ID1_ALM_DUMMY
オーバーランハンドラ	(未使用)	―	―
ノーマル割込みハンドラ	ダミー	cfg ファイルの interrupt_vector[]	(ベクタ番号 508)
ダイレクト割込みハンドラ	ダミー	cfg ファイルの interrupt_vector[]	(ベクタ番号 509)
CPU 例外ハンドラ	(未使用)	―	―
拡張サービスコール	ダミー	cfg ファイルの extend_svc[]	―
初期化ルーチン	ダミー	cfg ファイルの init_routine[]	―

表16.18　その他のオブジェクト(CPUID#2)

オブジェクト種別	分類	生成方法	ID 名称
セマフォ	ダミー	cfg ファイルの semaphore[]	ID2_SEM_DUMMY
イベントフラグ	ダミー	cfg ファイルの flag[]	ID2_FLG_DUMMY
データキュー	ダミー	cfg ファイルの dataqueue[]	ID2_DTQ_DUMMY
メールボックス	リモートサービスコール例	cfg ファイルの mailbox[]	EXID2_MBX_REMOTE (他 CPU への export)
ミューテックス	sbrk()排他制御用	_INIT_LOWLEVEL()からの acre_mtx コール	(なし)
	_s1ptr 排他制御用	_INIT_LOWLEVEL()からの acre_mtx コール	(なし)
	iob 排他制御用	_INIT_LOWLEVEL()からの acre_mtx コール	(なし)
メッセージバッファ	ダミー	cfg ファイルの message_buffer[]	ID2_MBF_DUMMY
固定長メモリプール	リモートサービスコール例	cfg ファイルの memorypool[]	EXID2_MPF_REMOTE (他 CPU への export)
可変長メモリプール	ダミー	cfg ファイルの variable_memorypool[]	ID2_MPL_DUMMY
	OAL 用	OAL_Init()からの acre_mpl コール	−
周期ハンドラ	ダミー	cfg ファイルの cyclic_hand[]	ID2_CYC_DUMMY
アラームハンドラ	ダミー	cfg ファイルの alarm_hand[]	ID2_ALM_DUMMY
オーバーランハンドラ	(未使用)	−	−
ノーマル割込みハンドラ	ダミー	cfg ファイルの interrupt_vector[]	(ベクタ番号 510)
ダイレクト割込みハンドラ	ダミー	cfg ファイルの interrupt_vector[]	(ベクタ番号 511)
CPU 例外ハンドラ	(未使用)	−	−
拡張サービスコール	ダミー	cfg ファイルの extend_svc[]	−
初期化ルーチン	ダミー	cfg ファイルの init_routine[]	−

16.11 cfg ファイル

16.11.1 CPUID#1 (cpuid1¥cfg_out¥sample.cfg)

(1) system 定義

1	system {	
2	cpuid = 1;	CPUID
3	stack_size = 0x1000;	割込みスタックサイズ
4	kernel_stack_size = 0x400;	カーネルスタックサイズ
5	priority = 255;	最大タスク優先度
6	system_IPL = 14;	カーネル割込みマスクレベル
7	message_pri = 255;	最大メッセージ優先度
8	tic_deno = 1;	タイムティック周期＝TIC_NUME/TIC_DENO＝1msec
9	tic_nume = 1;	
10	tbr = FOR_SVC;	TBR レジスタをサービスコール専用に使用
11	parameter_check = YES;	カーネルサービスコールのパラメータエラーを検出する
12	mpfmanage = IN;	固定長メモリプールの管理テーブルをプール内部に配置する
13	newmpl = NEW;	可変長メモリプールは、新方式で管理する
14	trace = TARGET_TRACE;	サービスコールトレース機能としてターゲットトレース機能を使用する
15	trace_buffer = 0x10000;	トレースバッファのサイズ
16	trace_object = 5;	サービスコールトレースのオブジェクト取得数
17	action = YES;	オブジェクト操作機能を使用する
18	vector_type = ROM;	割込みベクタのタイプ
19	regbank = ALL;	レジスタバンクを利用可能な全割込み要因でレジスタバンクを使用する
20	};	

(2) maxdefine 定義

1	maxdefine {	
2	max_task = 20;	最大ローカルタスク ID
3	max_statictask = 0;	スタティックスタックを使用する最大ローカルタスク ID
4	max_sem = 10;	最大ローカルセマフォ ID
5	max_flag = 10;	最大ローカルイベントフラグ ID
6	max_dtq = 10;	最大ローカルデータキューID
7	max_mbx = 10;	最大ローカルメールボックス ID
8	max_mtx = 10;	最大ローカルミューテックス ID
9	max_mbf = 10;	最大ローカルメッセージバッファ ID
10	max_mpf = 10;	最大ローカル固定長メモリプール ID
11	max_mpl = 10;	最大ローカル可変長メモリプール ID
12	max_cyh = 10;	最大ローカル周期ハンドラ ID
13	max_alh = 10;	最大ローカルアラームハンドラ ID
14	max_fncd = 10;	最大拡張サービスコール機能コード
15	max_int = 511;	最大ベクタ番号
16	};	

(3) memstk 定義

1	memstk {	
2	all_memsize = 0x4000;	デフォルトタスクスタック用領域のサイズ
3	};	

(4) memdtq 定義

1	memdtq {	
2	all_memsize = 0x4000;	デフォルトデータキュー用領域のサイズ
3	};	

(5) memmbf 定義

1	memmbf {	
2	all_memsize = 0x4000;	デフォルトメッセージバッファ用領域のサイズ
3	};	

(6) memmpf 定義

1	memmpf {	
2	all_memsize = 0x4000;	デフォルト固定長メモリプール用領域のサイズ
3	};	

(7) memmplf 定義

1	memmpl {	
2	all_memsize = 0x4000;	デフォルト可変長メモリプール用領域のサイズ
3	};	

(8) clock 定義

1	clock {	
2	timer = TIMER;	時間管理機能を使用する
3	IPL = 13;	タイマ割込みレベル
4	number = 118; // CMT0-ch0	タイマ割込み番号(SH7265 の CMT0 の CH0)
5	stack_size = 0x800;	タイマスタックサイズ
6	};	

(9) remote_svc 定義

1	remote_svc {	
2	num_server = 3;	SVC サーバタスクの数
3	priority = 1;	SVC サーバタスクの優先度
4	stack_size = 0x400;	SVC サーバタスクのスタックサイズ
5	ipi_portid = 1; // Interrput priority = 14	リモートサービスコールで使用する IPI ポート ID(割込みレベル=14)
6	num_wait = 20;	SVC サーバタスク空き待ちタスクの最大数
7	};	

(10)task[]定義

1	// InitTask1() ****************	InitTask1()(初期起動タスク)
2	task[] { // the ID is assigned by configurator.	ID 番号を自動割当
3	name = ID1_TASK_INIT;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	entry_address = InitTask1();	タスクの開始アドレス
6	stack_size = 0x400; // the stack is allocated from default area.	スタックサイズ(スタック領域をデフォルトタスクスタック用領域から割り当てる)
7	// stack_section = <input section name>;	(スタック領域に付与するセクション名)
8	// stack_address = <input start address of stack area>;	(スタック領域の先頭アドレス)
9	priority = 1;	タスクの起動時優先度
10	initial_start = ON;	初期起動状態
11	exinf = 0;	拡張情報
12	fpu = OFF;	FPU を使用しない
13	};	
14		
15	// TaskSend() ****************	TaskSend()(リモートサービスコール使用例)
16	task[] { // the ID is assigned by configurator.	ID 番号を自動割当
17	name = ID1_TASK_SEND;	ID 名称
18	// export = <YES or NO>;	ID 名称を export しない
19	entry_address = TaskSend();	タスクの開始アドレス
20	stack_size = 0x400;	スタックサイズ(指定したセクション名でスタック領域を生成)
21	stack_section = C_TSKSTK; // the section mane is "BC_TSKSTK".	スタック領域に付与するセクション名＝BC_TSKSTK
22	// stack_address = <input start address of stack area>;	(スタック領域の先頭アドレス)
23	priority = 3;	タスクの起動時優先度
24	initial_start = ON;	初期起動状態
25	exinf = 0;	拡張情報
26	fpu = OFF;	FPU を使用しない
27	};	
28		
29	// TaskRpcCaller() ****************	TaskRpcCaller()(RPC コールタスク)
30	task[] { // the ID is assigned by configurator.	ID 番号を自動割当
31	name = ID1_TASK_RPCCALLER;	ID 名称
32	// export = <YES or NO>;	ID 名称を export しない
33	entry_address = TaskRpcCaller();	タスクの開始アドレス
34	stack_size = 0x400; // the stack is allocated from default area.	スタックサイズ(スタック領域をデフォルトタスクスタック用領域から割り当てる)
35	// stack_section = <input section name>;	(スタック領域に付与するセクション名)
36	// stack_address = <input start address of stack area>;	(スタック領域の先頭アドレス)
37	priority = 5;	タスクの起動時優先度
38	initial_start = ON;	初期起動状態
39	exinf = 0;	拡張情報
40	fpu = OFF;	FPU を使用しない
41	};	
42		
43	// DummyTask() ****************	DummyTask()(ダミータスク)
44	task[] { // the ID is assigned by configurator.	ID 番号を自動割当
45	name = ID1_TASK_DUMMY;	ID 名称
46	// export = <YES or NO>;	ID 名称を export しない
47	entry_address = DummyTask();	タスクの開始アドレス
48	stack_size = 0x200; // the stack is allocated from default area.	スタックサイズ(スタック領域をデフォルトタスク

		スタック用領域から割り当てる)
49	// stack_section = <input section name>;	(スタック領域に付与するセクション名)
50	// stack_address = <input start address of stack area>;	(スタック領域の先頭アドレス)
51	priority = 10;	タスクの起動時優先度
52	initial_start = ON;	初期起動状態
53	exinf = 0;	拡張情報
54	fpu = OFF;	FPU を使用しない
55	};	

(11)semaphore[]定義

1	// Dummy semaphore ***********	ダミーセマフォ
2	semaphore[1] { // the ID is 1.	ローカル ID 番号は 1
3	name = ID1_SEM_DUMMY;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	wait_queue = TA_TFIFO;	待ち行列属性
6	max_count = 1;	セマフォカウンタ最大値
7	initial_count = 1;	セマフォカウンタ初期値
8	};	

(12)eventflag[]定義

1	// Dummy eventflag ***********	ダミーイベントフラグ
2	flag[] { // the ID is assigned by configurator.	ID 番号を自動割当
3	name = ID1_FLG_DUMMY;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	wait_queue = TA_TFIFO;	待ち行列属性
6	initial_pattern = 0;	初期ビットパターン
7	wait_multi = TA_WSGL;	複数待ちを許可しない
8	clear_attribute = YES;	クリア属性
9	};	

(13)dataqueue[]定義

1	// Dummy dataqueue ***********	ダミーデータキュー
2	dataqueue[2] { // the ID is 2.	ローカル ID 番号は 2
3	name = ID1_DTQ_DUMMY;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	buffer_size = 256;	最大データ数
6	section = C_DTQ; // the section mane is "BC_DTQ".	データキュー領域に付与するセクション名
7	// address = <input start address of dataqueue area>;	(データキュー領域の先頭アドレス)
8	wait_queue = TA_TFIFO;	待ち行列属性
9	};	

(14)mailbox[]定義

1	// Dummy mailbox ***********	ダミーメールボックス
2	mailbox[] { // the ID is assigned by configurator.	ID 番号を自動割当
3	name = ID1_MBX_DUMMY;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	wait_queue = TA_TFIFO;	待ち行列属性
6	message_queue = TA_MFIFO;	メッセージキューへのつなぎ方
7	max_pri = 255;	メッセージの最大優先度(TA_MFIFO の場合は意味を持ちません)
8	};	

(15)mutex[]定義

1	// Dummy mutex ***********	ダミーミューテックス
2	mutex[1] { // the ID is 1.	ローカル ID 番号は 1
3	name = ID1_MTX_DUMMY;	ID 名称
4	protocol = TA_CEILING;	優先度上限プロトコル
5	ceil_pri = 1;	上限優先度
6	};	

(16)message_buffer[]定義

1	// Dummy message buffer ***********	ダミーメッセージバッファ
2	message_buffer[] { // the ID is assigned by configurator.	ID 番号を自動割当
3	name = ID1_MBF_DUMMY;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	buffer_size = 0x400;	バッファサイズ
6	section = C_MBF; // the section mane is "BC_MBF".	バッファ領域に付与するセクション名
7	// address = <input start address of buffer area>;	(バッファ領域の先頭アドレス)
8	max_msgsz = 0x100;	最大メッセージサイズ
9	wait_queue = TA_TFIFO;	待ち行列属性
10	};	

(17)memorypool[]定義

1	// Dummy fixed-size memory pool ***********	ダミー固定長メモリプール
2	memorypool[] { // the ID is assigned by configurator.	ローカル ID 番号は 2
3	name = ID1_MPF_DUMMY;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	section = C_MPF; // the section name is "BC_MPF".	プール領域に付与するセクション名
6	// address = <input start address of pool area>;	(プール領域の先頭アドレス)
7	num_block = 32;	ブロック数
8	siz_block = 16	ブロックサイズ
9	wait_queue = TA_TFIFO;	待ち行列属性
10	};	

(18)variable_memorypool[]定義

1	// Dummy variable size memory pool ***********	ダミー可変長メモリプール
2	variable_memorypool[] { // the ID is assigned by configurator.	ID 番号を自動割当
3	name = ID1_MPL_DUMMY;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	heap_size = 0x400;	プールサイズ
6	mpl_section = C_MPL;	プール領域に付与するセクション名
7	// mpl_address = <input start address of pool area>;	(プール領域の先頭アドレス)
8	unfragment = OFF;	断片化軽減機能を使用しない
9	wait_queue = TA_TFIFO;	待ち行列属性
10	};	

(19)cyclic_hand[]定義

1	// Dummy cyclic handler ***********	ダミー周期ハンドラ
2	cyclic_hand[] { // the ID is assigned by configurator.	ID 番号を自動割当
3	name = ID1_CYC_DUMMY;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	interval_counter = 100;	起動周期
6	start = ON;	動作状態にする
7	phsatr = OFF;	起動位相を保存しない
8	phs_counter = 30;	起動位相
9	exinf = 0;	拡張情報
10	entry_address = DummyCyclicHandler();	ハンドラの開始アドレス
11	};	

(20)alarm_hand[]定義

1	// Dummy alarm handler ***********	ダミーアラームハンドラ
2	alarm_hand[] { // the ID is assigned by configurator.	ID 番号を自動割当
3	name = ID1_ALM_DUMMY;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	exinf = 0;	拡張情報
6	entry_address = DummyAlarmHandler();	ハンドラの開始アドレス
7	};	

(21)overrun_hand[]定義

1	// Dummy overrun handler ***********	ダミーオーバーランハンドラ
2	overrun_hand {	
3	entry_address = DummyOverrunHandler();	ハンドラの開始アドレス
4	};	

(22)extend_svc[]定義

1	// Dummy extended SVC ***********	ダミー拡張サービスコール
2	extend_svc[1] { // the function code is 1.	機能コードは 1
3	entry_address = DummyExtendSVCRoutine();	拡張サービスコールルーチンの開始アドレス
4	};	

(23)interrupt_vector[]定義

1	// Direcr interrupt handler for IPI port ID#0 ***********	IPI ポート ID#0 のダイレクト割込みハンドラ(ipi.c)
2	interrupt_vector[21] {	ベクタ番号は 21
3	direct = YES;	ダイレクト属性あり
4	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ちません)
5	entry_address = IPI_Port0Handler();	ハンドラの開始アドレス
6	};	
7		
8	// Direcr interrupt handler for IPI port ID#1 ***********	IPI ポート ID#1 のダイレクト割込みハンドラ(ipi.c)
9	interrupt_vector[22] {	ベクタ番号は 22
10	direct = YES;	ダイレクト属性あり
11	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ちません)
12	entry_address = IPI_Port1Handler();	ハンドラの開始アドレス
13	};	
14		
15	// Direcr interrupt handler for IPI port ID#2 ***********	IPI ポート ID#2 のダイレクト割込みハンドラ(ipi.c)
16	interrupt_vector[23] {	ベクタ番号は 23
17	direct = YES;	ダイレクト属性あり
18	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ちません)
19	entry_address = IPI_Port2Handler();	ハンドラの開始アドレス
20	};	
21		
22	// Direcr interrupt handler for IPI port ID#3 ***********	IPI ポート ID#3 のダイレクト割込みハンドラ(ipi.c)
23	interrupt_vector[24] {	ベクタ番号は 24
24	direct = YES;	ダイレクト属性あり
25	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ちません)
26	entry_address = IPI_Port3Handler();	ハンドラの開始アドレス
27	};	
28		
29	// Direcr interrupt handler for IPI port ID#4 ***********	IPI ポート ID#4 のダイレクト割込みハンドラ(ipi.c)
30	interrupt_vector[25] {	ベクタ番号は 25
31	direct = YES;	ダイレクト属性あり
32	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ちません)
33	entry_address = IPI_Port4Handler();	ハンドラの開始アドレス
34	};	
35		
36	// Direcr interrupt handler for IPI port ID#5 ***********	IPI ポート ID#5 のダイレクト割込みハンドラ(ipi.c)
37	interrupt_vector[26] {	ベクタ番号は 26
38	direct = YES;	ダイレクト属性あり
39	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ちません)
40	entry_address = IPI_Port5Handler();	ハンドラの開始アドレス
41	};	
42		
43	// Direcr interrupt handler for IPI port ID#6 ***********	IPI ポート ID#6 のダイレクト割込みハンドラ(ipi.c)
44	interrupt_vector[27] {	ベクタ番号は 27
45	direct = YES;	ダイレクト属性あり
46	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ち

		ません)
47	entry_address = IPI_Port6Handler();	ハンドラの開始アドレス
48	};	
49		
50	// Direcr interrupt handler for IPI port ID#7 ***********	IPI ポート ID#7 のダイレクト割込みハンドラ(ipi.c)
51	interrupt_vector[28] {	ベクタ番号は 28
52	direct = YES;	ダイレクト属性あり
53	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ちません)
54	entry_address = IPI_Port7Handler();	ハンドラの開始アドレス
55	};	
56		
57	// Dummy normal interrupt handler ***********	ダミーノーマル割込みハンドラ
58	interrupt_vector[508] { // Normal interrupt handler	ベクタ番号は 508
59	direct = NO;	ダイレクト属性なし
60	regbank = YES;	レジスタバンクを使用(system.regnbank が ALL のため、意味を持ちません)
61	entry_address = DummyNormalIntHandler508();	ハンドラの開始アドレス
62	};	
63		
64	// Dummy direct interrupt handler ***********	ダミーダイレクト割込みハンドラ
65	interrupt_vector[509] { // Direct interrupt handler	ベクタ番号は 509
66	direct = YES;	ダイレクト属性あり
67	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ちません)
68	entry_address = DummyDirectIntHandler509();	ハンドラの開始アドレス
69	};	

(24)init_routine[]定義

1	// Dummy initialization routine ***********	ダミー初期化ルーチン
2	init_routine[] {	
3	exinf = 0;	拡張情報
4	entry_address = DummyInitRoutine();	初期化ルーチンの開始アドレス
5	};	

(25)service_call 定義

以下のサービスコールのみを NO に定義しています。

- vscr_tsk, ivscr_tsk（スタティックスタックを使用するタスクの生成）
- タスク例外処理機能の全サービスコール
- def_inh, idef_inh（割込みハンドラの定義）
- def_exc, idef_exc（CPU 例外ハンドラの定義）
- vdef_trp, ivdef_trp（TRAPA 例外ハンドラの定義）

16.11.2 CPUID#2 (cpuid2¥cfg_out¥sample.cfg)

(1) system 定義

1	system {	
2	cpuid = 2;	CPUID
3	stack_size = 0x1000;	割込みスタックサイズ
4	kernel_stack_size = 0x400;	カーネルスタックサイズ
5	priority = 255;	最大タスク優先度
6	system_IPL = 14;	カーネル割込みマスクレベル
7	message_pri = 255;	最大メッセージ優先度
8	tic_deno = 1;	タイムティック周期＝TIC_NUME/TIC_DENO＝1msec
9	tic_nume = 1;	
10	tbr = FOR_SVC;	TBR レジスタをサービスコール専用に使用
11	parameter_check = YES;	カーネルサービスコールのパラメータエラーを検出する
12	mpfmanage = IN;	固定長メモリプールの管理テーブルをプール内部に配置する
13	newmpl = NEW;	可変長メモリプールは、新方式で管理する。
14	trace = TARGET_TRACE;	サービスコールトレース機能としてターゲットトレース機能を使用する
15	trace_buffer = 0x10000;	トレースバッファのサイズ
16	trace_object = 5;	サービスコールトレースのオブジェクト取得数
17	action = YES;	オブジェクト操作機能を使用する
18	vector_type = ROM;	割込みベクタのタイプ
19	regbank = ALL;	レジスタバンクを利用可能な全割込み要因でレジスタバンクを使用する
20	};	

(2) maxdefine 定義

1	maxdefine {	
2	max_task = 20;	最大ローカルタスク ID
3	max_statictask = 0;	スタティックスタックを使用する最大ローカルタスク ID
4	max_sem = 10;	最大ローカルセマフォ ID
5	max_flag = 10;	最大ローカルイベントフラグ ID
6	max_dtq = 10;	最大ローカルデータキューID
7	max_mbx = 10;	最大ローカルメールボックス ID
8	max_mtx = 10;	最大ローカルミューテックス ID
9	max_mbf = 10;	最大ローカルメッセージバッファ ID
10	max_mpf = 10;	最大ローカル固定長メモリプール ID
11	max_mpl = 10;	最大ローカル可変長メモリプール ID
12	max_cyh = 10;	最大ローカル周期ハンドラ ID
13	max_alh = 10;	最大ローカルアラームハンドラ ID
14	max_fncd = 10;	最大拡張サービスコール機能コード
15	max_int = 511;	最大ベクタ番号
16	};	

(3) memstk 定義

1	memstk {	
2	all_memsize = 0x4000;	デフォルトタスクスタック用領域のサイズ
3	};	

(4) memdtq 定義

1	memdtq {	
2	all_memsize = 0x4000;	デフォルトデータキュー用領域のサイズ
3	};	

RENESAS

(5) memmbf 定義

1	memmbf {	
2	all_memsize = 0x4000;	デフォルトメッセージバッファ用領域のサイズ
3	};	

(6) memmpf 定義

1	memmpf {	
2	all_memsize = 0x4000;	デフォルト固定長メモリプール用領域のサイズ
3	};	

(7) memmplf 定義

1	memmpl {	
2	all_memsize = 0x4000;	デフォルト可変長メモリプール用領域のサイズ
3	};	

(8) clock 定義

1	clock {	
2	timer = TIMER;	時間管理機能を使用する
3	IPL = 13;	タイマ割込みレベル
4	number = 119; // CMT0-ch1	タイマ割込み番号(SH7265 の CMT0 の CH1)
5	stack_size = 0x800;	タイマスタックサイズ
6	};	

(9) remote_svc 定義

1	remote_svc {	
2	num_server = 3;	SVC サーバタスクの数
3	priority = 1;	SVC サーバタスクの優先度
4	stack_size = 0x400;	SVC サーバタスクのスタックサイズ
5	ipi_portid = 1; // Interrput priority = 14	リモートサービスコールで使用する IPI ポート ID(割込みレベル=14)
6	num_wait = 20;	SVC サーバタスク空き待ちタスクの最大数
7	};	

(10)task[]定義

1	// InitTask2() ****************	InitTask2()(初期起動タスク)
2	task[] { // the ID is assigned by configurator.	ID 番号を自動割当
3	name = ID2_TASK_INIT;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	entry_address = InitTask2();	タスクの開始アドレス
6	stack_size = 0x400; // the stack is allocated from default area.	スタックサイズ(スタック領域をデフォルトタスクスタック用領域から割り当てる)
7	// stack_section = <input section name>;	(スタック領域に付与するセクション名)
8	// stack_address = <input start address of stack area>;	(スタック領域の先頭アドレス)
9	priority = 1;	タスクの起動時優先度
10	initial_start = ON;	初期起動状態
11	exinf = 0;	拡張情報
12	fpu = OFF;	FPU を使用しない
13	};	
14		
15	// TaskRecv() ****************	TaskRecv()(リモートサービスコール使用例)
16	task[] { // the ID is assigned by configurator.	ID 番号を自動割当
17	name = ID2_TASK_RECV;	ID 名称
18	// export = <YES or NO>;	ID 名称を export しない
19	entry_address = TaskRecv();	タスクの開始アドレス
20	stack_size = 0x400;	スタックサイズ(指定したセクション名でスタック領域を生成)
21	stack_section = C_TSKSTK; // the section mane is "BC_TSKSTK".	スタック領域に付与するセクション名＝BC_TSKSTK
22	// stack_address = <input start address of stack area>;	(スタック領域の先頭アドレス)
23	priority = 3;	タスクの起動時優先度
24	initial_start = ON;	初期起動状態
25	exinf = 0;	拡張情報
26	fpu = OFF;	FPU を使用しない
27	};	
28		
29	// DummyTask() ****************	DummyTask()(ダミータスク)
30	task[] { // the ID is assigned by configurator.	ID 番号を自動割当
31	name = ID2_TASK_DUMMY;	ID 名称
32	// export = <YES or NO>;	ID 名称を export しない
33	entry_address = DummyTask();	タスクの開始アドレス
34	stack_size = 0x200; // the stack is allocated from default area.	スタックサイズ(スタック領域をデフォルトタスクスタック用領域から割り当てる)
35	// stack_section = <input section name>;	(スタック領域に付与するセクション名)
36	// stack_address = <input start address of stack area>;	(スタック領域の先頭アドレス)
37	priority = 10;	タスクの起動時優先度
38	initial_start = ON;	初期起動状態
39	exinf = 0;	拡張情報
40	fpu = OFF;	FPU を使用しない
41	};	

(11)semaphore[]定義

1	// Dummy semaphore ***********	ダミーセマフォ
2	semaphore[1] { // the ID is 1.	ローカル ID 番号は 1
3	name = ID2_SEM_DUMMY;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	wait_queue = TA_TFIFO;	待ち行列属性
6	max_count = 1;	セマフォカウンタ最大値
7	initial_count = 1;	セマフォカウンタ初期値
8	};	

(12)eventflag[]定義

1	// Dummy eventflag ***********	ダミーイベントフラグ
2	flag[] { // the ID is assigned by configurator.	ID 番号を自動割当
3	name = ID2_FLG_DUMMY;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	wait_queue = TA_TFIFO;	待ち行列属性
6	initial_pattern = 0;	初期ビットパターン
7	wait_multi = TA_WSGL;	複数待ちを許可しない
8	clear_attribute = YES;	クリア属性
9	};	

(13)dataqueue[]定義

1	// Dummy dataqueue ***********	ダミーデータキュー
2	dataqueue[2] { // the ID is 2.	ローカル ID 番号は 2
3	name = ID2_DTQ_DUMMY;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	buffer_size = 256;	最大データ数
6	section = C_DTQ; // the section mane is "BC_DTQ".	データキュー領域に付与するセクション名
7	// address = <input start address of dataqueue area>;	(データキュー領域の先頭アドレス)
8	wait_queue = TA_TFIFO;	待ち行列属性
9	};	

(14)mailbox[]定義

1	// Mailbox for remote-SVC sample ***********	リモートサービスコール使用例で使用するメールボックス
2	mailbox[1] { // the ID is 1.	ローカル ID 番号は 1
3	name = EXID2_MBX_COMM;	ID 名称
4	export = YES;	ID 名称を export する
5	wait_queue = TA_TFIFO;	待ち行列属性
6	message_queue = TA_MFIFO;	メッセージキューへのつなぎ方
7	max_pri = 255;	メッセージの最大優先度(TA_MFIFO の場合は意味を持ちません)
8	};	
9		
10	// Dummy mailbox ***********	ダミーメールボックス
11	mailbox[] { // the ID is assigned by configurator.	ID 番号を自動割当
12	name = ID2_MBX_DUMMY;	ID 名称
13	// export = <YES or NO>;	ID 名称を export しない
14	wait_queue = TA_TFIFO;	待ち行列属性
15	message_queue = TA_MFIFO;	メッセージキューへのつなぎ方
16	max_pri = 255;	メッセージの最大優先度(TA_MFIFO の場合は意味を持ちません)
17	};	

(15)mutex[]定義

1	// Dummy mutex ***********	ダミーミューテックス
2	mutex[1] { // the ID is 1.	ローカル ID 番号は 1
3	name = ID2_MTX_DUMMY;	ID 名称
4	protocol = TA_CEILING;	優先度上限プロトコル
5	ceil_pri = 1;	上限優先度
6	};	

(16)message_buffer[]定義

1	// Dummy message buffer ***********	ダミーメッセージバッファ
2	message_buffer[] { // the ID is assigned by configurator.	ID 番号を自動割当
3	name = ID2_MBF_DUMMY;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	buffer_size = 0x400;	バッファサイズ
6	section = C_MBF; // the section mane is "BC_MBF".	バッファ領域に付与するセクション名
7	// address = <input start address of buffer area>;	(バッファ領域の先頭アドレス)
8	max_msgsz = 0x100;	最大メッセージサイズ
9	wait_queue = TA_TFIFO;	待ち行列属性
10	};	

(17)memorypool[]定義

1	// Fixed-size memory pool for remote-SVC sample ***********	リモートサービスコール使用例で使用する固定長メモリプール
2	memorypool[1] { // the ID is 1.	ローカル ID 番号は 1
3	name = EXID2_MPF_NONCACHED;	ID 名称
4	export = YES;	ID 名称を export する
5	section = D_MPF; // the section name is "BD_MPF".	プール領域に付与するセクション名
6	// address = <input start address of pool area>;	(プール領域の先頭アドレス)
7	num_block = 16;	ブロック数
8	siz_block = 8; // sizeof(USER_MSG)	ブロックサイズ
9	wait_queue = TA_TFIFO;	待ち行列属性
10	};	
11		
12	// Dummy fixed-size memory pool ***********	ダミー固定長メモリプール
13	memorypool[] { // the ID is assigned by configurator.	ローカル ID 番号は 2
14	name = ID2_MPF_DUMMY;	ID 名称
15	// export = <YES or NO>;	ID 名称を export しない
16	section = C_MPF; // the section name is "BC_MPF".	プール領域に付与するセクション名
17	// address = <input start address of pool area>;	(プール領域の先頭アドレス)
18	num_block = 32;	ブロック数
19	siz_block = 16;	ブロックサイズ
20	wait_queue = TA_TFIFO;	待ち行列属性
21	};	

(18)variable_memorypool[]定義

1	// Dummy variable size memory pool ***********	ダミー可変長メモリプール
2	variable_memorypool[] { // the ID is assigned by configurator.	ID 番号を自動割当
3	name = ID2_MPL_DUMMY;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	heap_size = 0x400;	プールサイズ
6	mpl_section = C_MPL;	プール領域に付与するセクション名
7	// mpl_address = <input start address of pool area>;	(プール領域の先頭アドレス)
8	unfragment = OFF;	断片化軽減機能を使用しない
9	wait_queue = TA_TFIFO;	待ち行列属性
10	};	

(19)cyclic_hand[]定義

1	// Dummy cyclic handler ***********	ダミー周期ハンドラ
2	cyclic_hand[] { // the ID is assigned by configurator.	ID 番号を自動割当
3	name = ID2_CYC_DUMMY;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	interval_counter = 100;	起動周期
6	start = ON;	動作状態にする
7	phsatr = OFF;	起動位相を保存しない
8	phs_counter = 30;	起動位相
9	exinf = 0;	拡張情報
10	entry_address = DummyCyclicHandler();	ハンドラの開始アドレス
11	};	

(20)alarm_hand[]定義

1	// Dummy alarm handler ***********	ダミーアラームハンドラ
2	alarm_hand[] { // the ID is assigned by configurator.	ID 番号を自動割当
3	name = ID2_ALM_DUMMY;	ID 名称
4	// export = <YES or NO>;	ID 名称を export しない
5	exinf = 0;	拡張情報
6	entry_address = DummyAlarmHandler();	ハンドラの開始アドレス
7	};	

(21)overrun_hand[]定義

1	// Dummy overrun handler ***********	ダミーオーバーランハンドラ
2	overrun_hand {	
3	entry_address = DummyOverrunHandler();	ハンドラの開始アドレス
4	};	

(22)extend_svc[]定義

1	// Dummy extended SVC ***********	ダミー拡張サービスコール
2	extend_svc[1] { // the function code is 1.	機能コードは 1
3	entry_address = DummyExtendSVCRoutine();	拡張サービスコールルーチンの開始アドレス
4	};	

(23)interrupt_vector[]定義

1	// Direcr interrupt handler for IPI port ID#0 ***********	IPI ポート ID#0 のダイレクト割込みハンドラ(ipi.c)
2	interrupt_vector[21] {	ベクタ番号は 21
3	direct = YES;	ダイレクト属性あり
4	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ちません)
5	entry_address = IPI_Port0Handler();	ハンドラの開始アドレス
6	};	
7		
8	// Direcr interrupt handler for IPI port ID#1 ***********	IPI ポート ID#1 のダイレクト割込みハンドラ(ipi.c)
9	interrupt_vector[22] {	ベクタ番号は 22
10	direct = YES;	ダイレクト属性あり
11	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ちません)
12	entry_address = IPI_Port1Handler();	ハンドラの開始アドレス
13	};	
14		
15	// Direcr interrupt handler for IPI port ID#2 ***********	IPI ポート ID#2 のダイレクト割込みハンドラ(ipi.c)
16	interrupt_vector[23] {	ベクタ番号は 23
17	direct = YES;	ダイレクト属性あり
18	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ちません)
19	entry_address = IPI_Port2Handler();	ハンドラの開始アドレス
20	};	
21		
22	// Direcr interrupt handler for IPI port ID#3 ***********	IPI ポート ID#3 のダイレクト割込みハンドラ(ipi.c)
23	interrupt_vector[24] {	ベクタ番号は 24
24	direct = YES;	ダイレクト属性あり
25	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ちません)
26	entry_address = IPI_Port3Handler();	ハンドラの開始アドレス
27	};	
28		
29	// Direcr interrupt handler for IPI port ID#4 ***********	IPI ポート ID#4 のダイレクト割込みハンドラ(ipi.c)
30	interrupt_vector[25] {	ベクタ番号は 25
31	direct = YES;	ダイレクト属性あり
32	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ちません)
33	entry_address = IPI_Port4Handler();	ハンドラの開始アドレス
34	};	
35		
36	// Direcr interrupt handler for IPI port ID#5 ***********	IPI ポート ID#5 のダイレクト割込みハンドラ(ipi.c)
37	interrupt_vector[26] {	ベクタ番号は 26
38	direct = YES;	ダイレクト属性あり
39	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ちません)
40	entry_address = IPI_Port5Handler();	ハンドラの開始アドレス
41	};	
42		
43	// Direcr interrupt handler for IPI port ID#6 ***********	IPI ポート ID#6 のダイレクト割込みハンドラ(ipi.c)
44	interrupt_vector[27] {	ベクタ番号は 27
45	direct = YES;	ダイレクト属性あり
46	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ち

		ません)
47	entry_address = IPI_Port6Handler();	ハンドラの開始アドレス
48	};	
49		
50	// Direcr interrupt handler for IPI port ID#7 ***********	IPI ポート ID#7 のダイレクト割込みハンドラ(ipi.c)
51	interrupt_vector[28] {	ベクタ番号は 28
52	direct = YES;	ダイレクト属性あり
53	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ちません)
54	entry_address = IPI_Port7Handler();	ハンドラの開始アドレス
55	};	
56		
57	// Dummy normal interrupt handler ***********	ダミーノーマル割込みハンドラ
58	interrupt_vector[510] { // Normal interrupt handler	ベクタ番号は 510
59	direct = NO;	ダイレクト属性なし
60	regbank = YES;	レジスタバンクを使用(system.regnbank が ALL のため、意味を持ちません)
61	entry_address = DummyNormalIntHandler510();	ハンドラの開始アドレス
62	};	
63		
64	// Dummy direct interrupt handler ***********	ダミーダイレクト割込みハンドラ
65	interrupt_vector[511] { // Direct interrupt handler	ベクタ番号は 511
66	direct = YES;	ダイレクト属性あり
67	regbank = YES;	レジスタバンクを使用(ダイレクト属性ありのため、意味を持ちません)
68	entry_address = DummyDirectIntHandler511();	ハンドラの開始アドレス
69	};	

(24)init_routine[]定義

1	// Dummy initialization routine ***********	ダミー初期化ルーチン
2	init_routine[] {	
3	exinf = 0;	拡張情報
4	entry_address = DummyInitRoutine();	初期化ルーチンの開始アドレス
5	};	

(25)service_call 定義

以下のサービスコールのみを NO に定義しています。

- vscr_tsk, ivscr_tsk（スタティックスタックを使用するタスクの生成）
- タスク例外処理機能の全サービスコール
- def_inh, idef_inh（割込みハンドラの定義）
- def_exc, idef_exc（CPU 例外ハンドラの定義）
- vdef_trp, ivdef_trp（TRAPA 例外ハンドラの定義）

16.12 IPI ポート

本サンプルでは、CPUID#1, CPUID#2 ともに、リモートサービスコールおよび RPC で IPI ポートを使用しています。

IPI ポートの設定については、cfg ファイル、IPI コンフィギュレーションファイル、RPC ライブラリの rpc_init()など、複数箇所での設定が必要であり、そのために誤った設定をしてしまう可能性が高いため、ここでその方法を解説します。

IPI コンフィギュレーションファイルでは、各ポート ID を利用可能か不可かを定義します。

リモートサービスコールで使用するポート ID は､利用可能なポート ID の中でカーネル割込みマスクレベル以下の割込みレベルのポートを選び、cfg ファイルの remote_svc.ipi_portid に定義します。これにより、vini_rmt はこのポート ID で IPI_create()を行います。

RPC で使用するポート ID は、利用可能なポート ID の中でカーネル割込みマスクレベル以下の割込みレベルのポートを選び、rpc_init()に渡す rpc_config 構造体の ulIPIPortID に指定します。これにより、rpc_init()はこのポート ID で IPI_create()を行います。

IPI のコンフィギュレーションファイルは、CPUID#1 側は cpuid1¥ipi¥ipi_config.h、CPUID#2 側は cpuid2¥ipi¥ipi_config.h です。

rpc_init()を呼び出すのは、CPUID#1 側は cpuid1¥init_task¥init_task1.c の初期起動タスク、CPUID#2 側は cpuid2¥init_task¥init_task2.c の初期起動タスクです。

また、カーネル割込みマスクレベル(system.system_IPL)は、両 CPU とも 14 としています。

表 16.19に、本サンプルの IPI ポートの状況を示します。

表16.19　IPI ポート

ポート ID	ベクタ番号	プロセッサ間割込みレベル	CPUID#1		CPUID#2	
			IPI コンフィギュレーション	IPI_create()状況	IPI コンフィギュレーション	IPI_create()状況
0	21	15	利用可能	(未使用)	利用可能	(未使用)
1	22	14	利用可能	リモートサービスコール用 (remote_svc.ipi_portid)	利用可能	リモートサービスコール用 (remote_svc.ipi_portid)
2	23	13	利用可能	RPC 用 (rpc_config.ulIPIPortID)	利用可能	RPC 用 (rpc_config.ulIPIPortID)
3	24	12	利用可能	(未使用)	利用可能	(未使用)
4	25	11	利用可能	(未使用)	利用可能	(未使用)
5	26	10	利用可能	(未使用)	利用可能	(未使用)
6	27	9	利用可能	(未使用)	利用可能	(未使用)
7	28	8	利用可能	(未使用)	利用可能	(未使用)

16.13 他ハードウェアへのポーティング

本サンプルを他のハードウェアにポーティングする場合は、以下のファイルを使用するハードウェアに応じて変更してください。

(1) iodefine¥ディレクトリのファイル（I/Oレジスタ定義、周辺クロック定義、kernel_intspec.h）
(2) cpuid1¥reset¥ディレクトリ, cpuid2¥reset¥ディレクトリのファイル（リセット関係）
(3) cpuid1¥os_timer¥ディレクトリ, cpuid2¥os_timer¥ディレクトリのファイル（タイマドライバ）

17. ビルド

本章では、主に<SAMPLE_INST>R0K572650D000BR ディレクトリのサンプル High-performance Embedded Workshop ワークスペースを用いたビルド方法について解説します。

なお、本章を理解するには、以下のツールについてある程度の知識が必要です。

- High-performance Embedded Workshop
- ツールチェイン
- cfg72mp

サンプルの High-performance Embedded Workshop ワークスペースファイルは、以下です。

- CPUID#1 用：<SAMPLE_INST>¥R0K572650D000BR¥cpuid1¥cpuid1.hws
- CPUID#2 用：<SAMPLE_INST>¥R0K572650D000BR¥cpuid2¥cpuid2.hws

17.1 カスタムプレースホルダ”$(RTOS_INST)”の設定

提供するワークスペースでは、カスタムプレースホルダ”$(RTOS_INST)”を使用しています。これは、HI7200/MP のシステムディレクトリを意味します。

HI7200/MP をリビジョンアップした場合や別のマシンにワークスペースを移動した場合など、システムディレクトリの変更が必要になった場合は、ユーザのワークスペースの$(RTOS_INST)を変更する必要があります。

カスタムプレースホルダの追加・変更方法は、High-performance Embedded Workshop のマニュアルまたはオンラインヘルプを参照してください。

17.2 ワークスペースに cfg72mp をカスタムビルドフェーズとして登録する

ワークスペースに、cfg72mp をカスタムビルドフェーズとして登録します。これにより、High-performance Embedded Workshop のビルド操作で cfg72mp が実行されるようになります。本節では、カスタムビルドフェーズの登録方法を説明します。

なお、提供する High-performance Embedded Workshop ワークスペースファイルでは、cfg72mp をカスタムビルドフェーズとして登録済みのため、本節に記載の作業は不要です。

17.2.1 ファイル拡張子の登録

カスタムビルドフェーズを機能させるためには、カスタムビルドフェーズで扱うファイル拡張子として、".cfg"を登録する必要があります。

High-performance Embedded Workshop メニューバーより[プロジェクト->ファイルの拡張子]を選択すると、図 17.1のダイアログボックスが開きます。

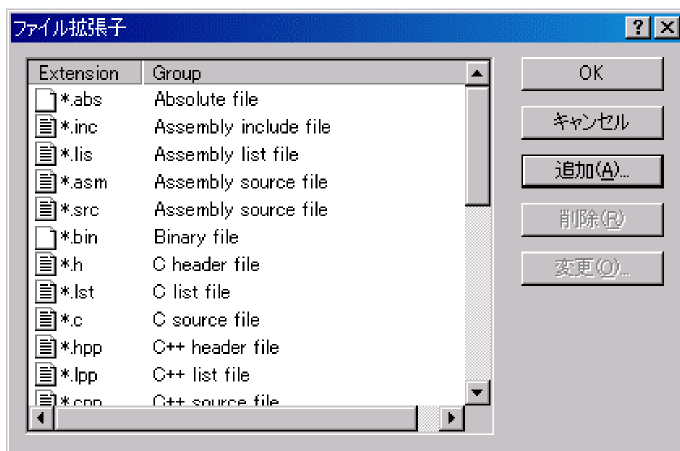


図17.1 [ファイル拡張子]ダイアログボックス

ここで、[追加]ボタンを押すと、図 17.2の[ファイル拡張子の追加]ダイアログボックスが開きますので、".cfg"を登録してください。

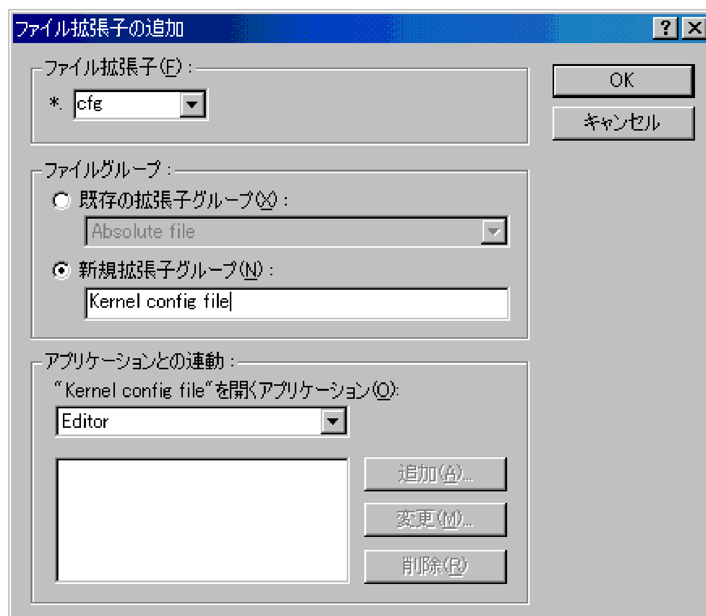


図17.2 [ファイル拡張子の追加]ダイアログボックス

17.2.2 cfg72mp カスタムビルドフェーズの作成

(1) High-performance Embedded Workshopメニューバーより[ビルド->ビルドフェーズ]を選択すると図17.3のダイアログボックスが開きます。

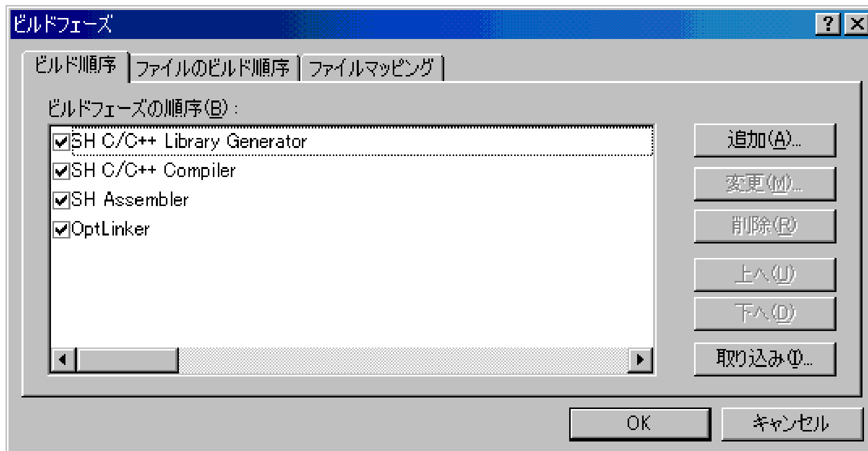


図17.3 [ビルドフェーズ]ダイアログボックス

ここで、[追加]を押してください。

(2) 下記ダイアログボックスが開きます。以降のダイアログで、cfg72mpカスタムビルドフェーズの設定を行っていきます。

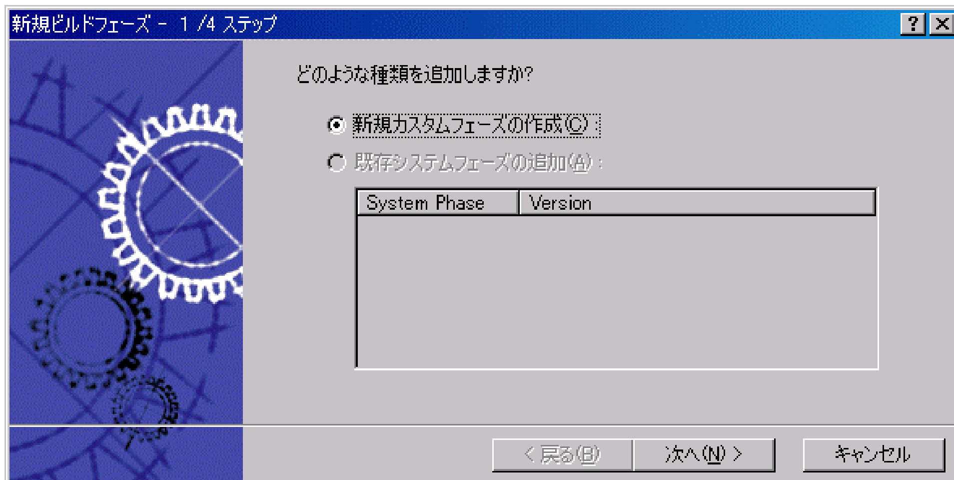


図17.4 [新規ビルドフェーズ – 1/4 ステップ]ダイアログボックス

[次へ]を押してください。

(3) [複数フェーズ]を選択してください。そして、[入力ファイル]で"Kernel config file"を選択し、[次へ]を押してください。

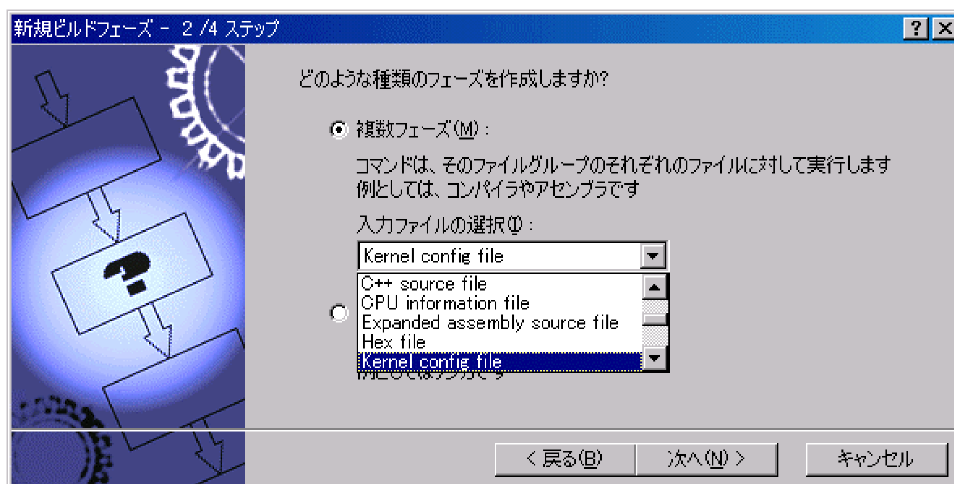


図17.5 [新規ビルドフェーズ – 2/4 ステップ]ダイアログボックス

(4) カスタムビルドフェーズ情報を設定します。
フェーズ名は、何でも良いですが本例では"cfg72mp"としています。
コマンドには、"$(RTOS_INST)¥cfg72mp¥cfg72mp.exe"を指定してください。
デフォルトオプションには、"$(FULLFILE)"を指定してください。
初期ディレクトリには、"$(WORKSPDIR)¥cfg_out"を指定してください。

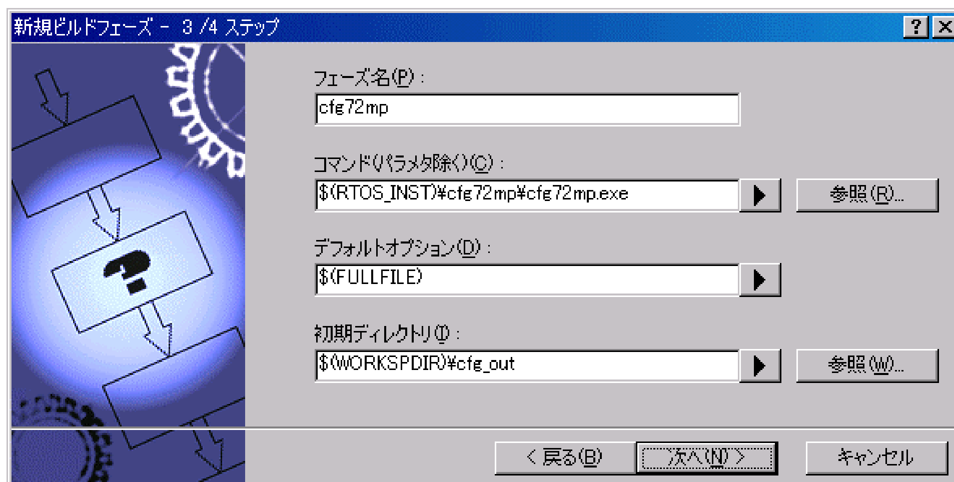


図17.6 [新規ビルドフェーズ – 3/4 ステップ]ダイアログボックス

[次へ]を押してください。

(5) 環境変数を設定します。[追加]ボタンを押して開く[環境変数]ダイアログボックスで、以下のように設定してください。

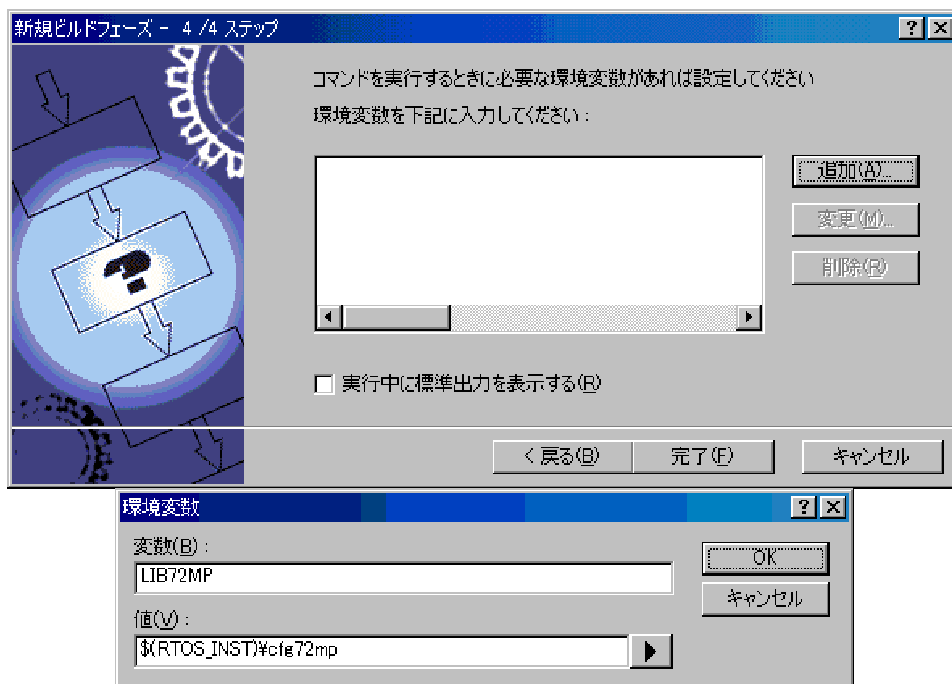


図17.7 [新規ビルドフェーズ – 4/4 ステップ]ダイアログボックス

以上で、cfg72mpカスタムビルドフェーズの作成は完了です。[完了]を押してください。

RENESAS

(6) 次に、cfg72mpのメッセージシンタックスを設定します。これにより、High-performance Embedded Workshopの[Build]ウィンドウに出力されるcfg72mpのエラー・ワーニングメッセージをダブルクリックすることで、cfgファイルの該当箇所にジャンプできるようになります。
[ビルドフェーズ]ダイアログボックスでcfg72mpを選択して[変更]ボタンを押してください。[cfg72mpの変更]ダイアログボックスが開きます。
[出力シンタックス]タブを選び、以下のようにエラーとワーニングのシンタックスを設定してください。

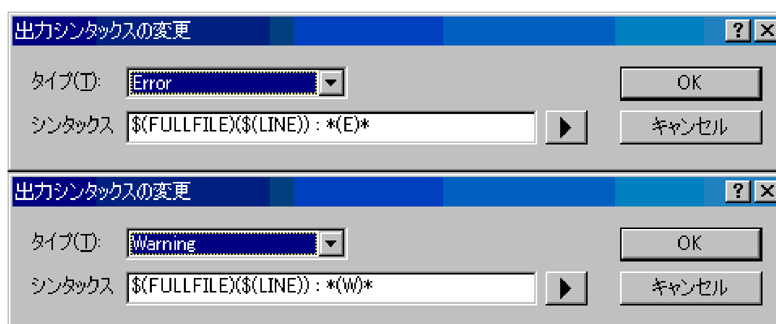


図17.8 cfg72mp 出力シンタックスの登録

(7) 次に、High-performance Embedded Workshopの「クリーン アクティブオブジェクト」、「クリーン 全プロジェクト」によって、cfg72mp出力ファイルも削除されるように設定します。[ビルド->cfg72mp…]メニューを選択し、[cfg72mp Options]ダイアログボックスを開いてください。[出力ファイル]タブを選び、"Kernel config file"のフォルダアイコンを選択している状態で[追加]ボタンを押し、表17.1に示す情報を登録してください。

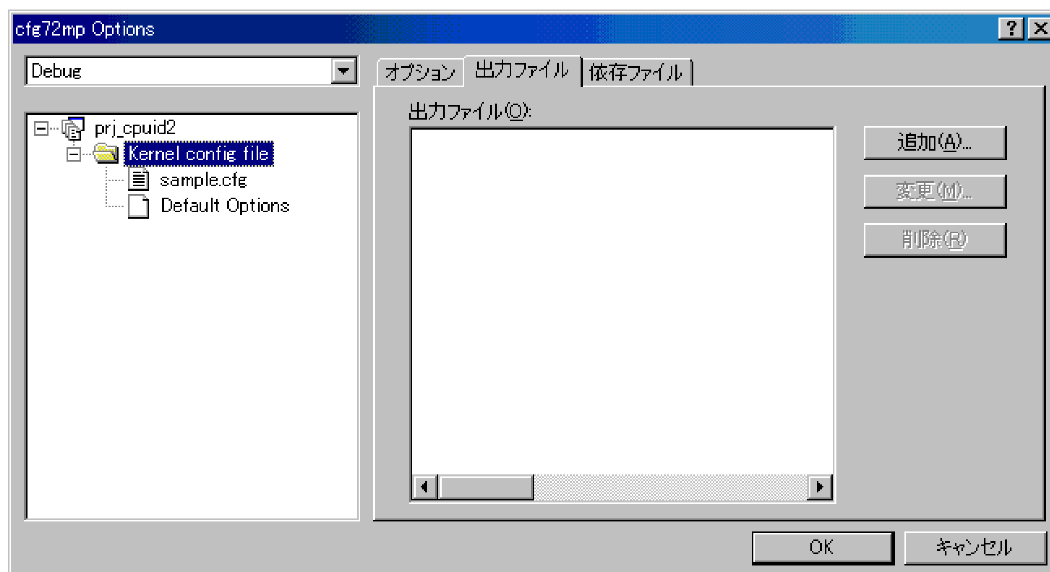


図17.9 [cfg72mp Options]ダイアログボックス

表17.1　cfg72mp 出力ファイル

設定	備考
$(FILEDIR)¥kernel_cfg.h	
$(FILEDIR)¥kernel_cfg_area.h	
$(FILEDIR)¥kernel_cfg_extern.h	
$(FILEDIR)¥kernel_cfg_inireg.h	
$(FILEDIR)¥kernel_cfg_inirtn.h	
$(FILEDIR)¥kernel_cfg_ststk.h	
$(FILEDIR)¥kernel_def.h	
$(FILEDIR)¥kernel_def_area.h	
$(FILEDIR)¥kernel_def_extern.h	
$(FILEDIR)¥kernel_def_inireg.h	
$(FILEDIR)¥kernel_def_inirtn.h	
$(FILEDIR)¥kernel_id.h	
$(FILEDIR)¥kernel_id_cpu1.h	CPUID#1 の場合のみ
$(FILEDIR)¥kernel_id_cpu2.h	CPUID#2 の場合のみ
$(FILEDIR)¥kernel_id_sys.h	
$(FILEDIR)¥kernel_id_sys_cpu1.h	CPUID#1 の場合のみ
$(FILEDIR)¥kernel_id_sys_cpu2.h	CPUID#2 の場合のみ
$(FILEDIR)¥kernel_macro.h	
$(FILEDIR)¥mycpuid.h	

17.2.3 ビルドフェーズの設定

(1) ビルド順序の設定

作成した cfg72mp カスタムビルドフェーズは、最下位に表示されます。これを以下のように”cfg72mp”が”SH C/C++ Compiler”より上位になるように、[上へ][下へ]ボタンを使って移動させてください。

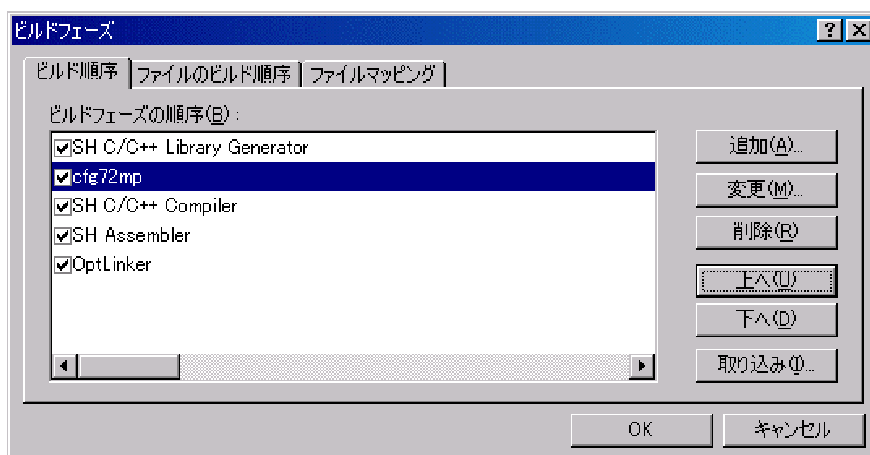


図17.10 [ビルドフェーズ]ダイアログボックス(ビルド順序)

(2) ファイルのビルド順序の設定

[ファイルのビルド順序]タブにおいて、[ファイルグループ]から”Kernel config file”を選択し、右側の[フェーズの順序]の[cfg72mp]をチェックしてください。

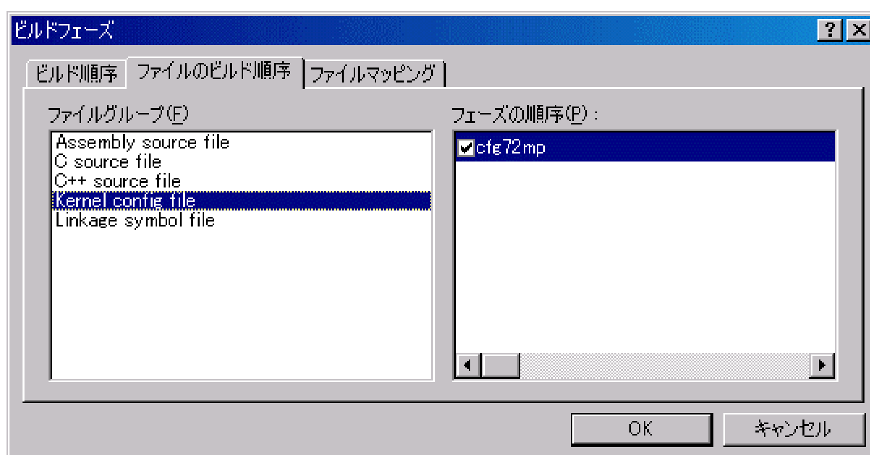


図17.11 [ビルドフェーズ]ダイアログボックス(ファイルのビルド順序)

以上で、ビルドフェーズの設定は完了です。

17.3 CPU 割込み仕様定義ファイル(kernel_intspec.h)の作成

kernel_intspec.h は、NMI や例外などのレジスタバンクを利用できないベクタ番号など、CPU の割込みハードウェア仕様をカーネルに教えるための重要なファイルです。本ファイルは CPUID #1、CPUID#2 共通にひとつ作成します。出荷時は、<SAMPLE_INST>¥R0K572650D000BR¥iodefine¥ディレクトリに格納されています。

kernel_intspec.h は、kernel.h からインクルードされます。

以下に、kernel_intspec.h の例を示します。

```
/***************************************************************************
1. Define IBNR register address (INTSPEC_IBNR_ADR)
   Specify 0 when the CPU used does not support register-bank.
****************************************************************************/
#define INTSPEC_IBNR_ADR1      0xFFFD940E     /**< IBNR register address for CPUID#1 */
#define INTSPEC_IBNR_ADR2      0xFFFD950E     /**< IBNR register address for CPUID#2 */


/***************************************************************************
2. Define the vector number that can not use register-bank(INTSPEC_NOBANK_VECxxx)
   "xxx" is an expression of the vector number by three decimal digits.
   These definitions are ignored when INTSPEC_IBNR_ADR is 0.
   Note, you don't have to define for vector number 0...3.
****************************************************************************/
#define INTSPEC_NOBANK_VEC004   /**< exception */
#define INTSPEC_NOBANK_VEC005   /**< exception */
#define INTSPEC_NOBANK_VEC006   /**< exception */
#define INTSPEC_NOBANK_VEC007   /**< exception */
#define INTSPEC_NOBANK_VEC008   /**< exception */
#define INTSPEC_NOBANK_VEC009   /**< exception */
#define INTSPEC_NOBANK_VEC010   /**< exception */
#define INTSPEC_NOBANK_VEC011   /**< NMI (cannot use register-bank) */
#define INTSPEC_NOBANK_VEC012   /**< user break interrupt (cannot use register-bank)  */
#define INTSPEC_NOBANK_VEC013   /**< exception */
/* #define INTSPEC_NOBANK_VEC014 *   H-UDI */
#define INTSPEC_NOBANK_VEC015   /**< exception */
#define INTSPEC_NOBANK_VEC016   /**< exception */
#define INTSPEC_NOBANK_VEC017   /**< exception */
#define INTSPEC_NOBANK_VEC018   /**< exception */
#define INTSPEC_NOBANK_VEC019   /**< exception */
#define INTSPEC_NOBANK_VEC020   /**< SCO interrupt (cannot use register-bank) */
/* #define INTSPEC_NOBANK_VEC021      * inter-processor interrupt */
/* #define INTSPEC_NOBANK_VEC022      * inter-processor interrupt */
/* #define INTSPEC_NOBANK_VEC023      * inter-processor interrupt */
```

```
/* #define INTSPEC_NOBANK_VEC024      * inter-processor interrupt */
/* #define INTSPEC_NOBANK_VEC025      * inter-processor interrupt */
/* #define INTSPEC_NOBANK_VEC026      * inter-processor interrupt */
/* #define INTSPEC_NOBANK_VEC027      * inter-processor interrupt */
/* #define INTSPEC_NOBANK_VEC028      * inter-processor interrupt */
#define INTSPEC_NOBANK_VEC029         /**< exception */
#define INTSPEC_NOBANK_VEC030         /**< exception */
#define INTSPEC_NOBANK_VEC031         /**< exception */
#define INTSPEC_NOBANK_VEC032         /**< TRAPA */
#define INTSPEC_NOBANK_VEC033         /**< TRAPA */
#define INTSPEC_NOBANK_VEC034         /**< TRAPA */
#define INTSPEC_NOBANK_VEC035         /**< TRAPA */
#define INTSPEC_NOBANK_VEC036         /**< TRAPA */
#define INTSPEC_NOBANK_VEC037         /**< TRAPA */
#define INTSPEC_NOBANK_VEC038         /**< TRAPA */
#define INTSPEC_NOBANK_VEC039         /**< TRAPA */
#define INTSPEC_NOBANK_VEC040         /**< TRAPA */
#define INTSPEC_NOBANK_VEC041         /**< TRAPA */
#define INTSPEC_NOBANK_VEC042         /**< TRAPA */
#define INTSPEC_NOBANK_VEC043         /**< TRAPA */
#define INTSPEC_NOBANK_VEC044         /**< TRAPA */
#define INTSPEC_NOBANK_VEC045         /**< TRAPA */
#define INTSPEC_NOBANK_VEC046         /**< TRAPA */
#define INTSPEC_NOBANK_VEC047         /**< TRAPA */
#define INTSPEC_NOBANK_VEC048         /**< TRAPA */
#define INTSPEC_NOBANK_VEC049         /**< TRAPA */
#define INTSPEC_NOBANK_VEC050         /**< TRAPA */
#define INTSPEC_NOBANK_VEC051         /**< TRAPA */
#define INTSPEC_NOBANK_VEC052         /**< TRAPA */
#define INTSPEC_NOBANK_VEC053         /**< TRAPA */
#define INTSPEC_NOBANK_VEC054         /**< TRAPA */
#define INTSPEC_NOBANK_VEC055         /**< TRAPA */
#define INTSPEC_NOBANK_VEC056         /**< TRAPA */
#define INTSPEC_NOBANK_VEC057         /**< TRAPA */
#define INTSPEC_NOBANK_VEC058         /**< TRAPA */
#define INTSPEC_NOBANK_VEC059         /**< TRAPA */
#define INTSPEC_NOBANK_VEC060         /**< TRAPA */
#define INTSPEC_NOBANK_VEC061         /**< TRAPA */
#define INTSPEC_NOBANK_VEC062         /**< TRAPA */
#define INTSPEC_NOBANK_VEC063         /**< TRAPA */
```

17.3.1 IBNR レジスタアドレス(INTSPEC_IBNR_ADR1, INTSPEC_IBNR_ADR2)

それぞれ、割込みコントローラの CPUID#1 用の IBNR レジスタ、および CPUID#2 用の IBNR レジスタのアドレスを定数式で定義します。

IBNR レジスタが存在しない場合、すなわちレジスタバンクをサポートしない CPU を使用する場合は、0 を指定してください。この場合、cfg ファイルの system.regbank の設定は意味を持たなくなり、全ての割込みがレジスタバンクを使用しないことになります。

参考として、SH7205, SH7265 の IBNR レジスタのアドレスを以下に示します。

- CPU#0(CPUID#1)：0xFFFD940E
- CPU#1(CPUID#2)：0xFFFD950E

17.3.2 レジスタバンク利用不可のベクタ番号(INTSPEC_NOBANK_VECxxx)

例外、NMI、ユーザブレークなど、CPU 仕様としてレジスタバンクを利用できない要因のベクタ番号を定義します。具体的には、それらのベクタ番号について、マクロ" INTSPEC_NOBANK_VECxxx"を定義してください。"xxx"の部分は、ベクタ番号を 10 進 3 桁表記したものとしてください。

17.4 kernel_def.c と kernel_cfg.c

この 2 つのファイルは、cfg72mp によって生成されたシステム定義ファイルを取り込むためのファイルで、<SAMPLE_INST>¥R0K572650D000BR¥cpuid1¥cfg_out¥ディレクトリ(CPUID#1 用)、および <SAMPLE_INST>¥R0K572650D000BR¥cpuid2¥cfg_out¥ディレクトリ(CPUID#2 用)にあります。ユーザは、これらのファイルを編集してはなりません。

kernel_def.c および kernel_cfg.c のコンパイル時には、インクルードパスとして <RTOS_INST>¥os¥system¥の指定が必要です。

また、kernel_def.c のコンパイル時には、"code=asmcode"オプションの指定が必要です。

17.5 セクション

各セクションは、ユーザがリンク時に適切なアドレスに配置しなければなりません。

17.5.1 セクション名の付与ルール

一部のセクションには、非キャッシャブル領域への配置が必須、といった制約を持つものがあります。

こういった制約の基にユーザが容易にセクション配置作業を行えるように、HI7200/MP が提供するすべてのセクションは、以下のルールに従って命名されています。アプリケーションでも、本ルールに従うことを推奨します。

ＰＣ＿ｈｉｋｎｌ

(1) 1 文字目

P：プログラムセクション
C：定数セクション
B：未初期化データセクション
D：初期化データセクションの ROM 部
R：初期化データセクションの RAM 部 (リンカの[ROM から RAM にマップするセクション]で生成されるセクション)

(2) 2 文字目

C：キャッシャブルアクセス可能
D：キャッシャブルアクセス禁止
L：スピンロック変数領域(キャッシャブルアクセス禁止、かつ全 CPU から同じバス経由でアクセス必須)

表 17.2に SH7265 の内蔵 RAM アドレス空間を示します。SH7265 の場合、内蔵 RAM へのアクセスは常に非キャッシャブルアクセスとなりますが、スピンロック変数を含むセクションはアドレス B に配置する必要があります。

表17.2 SH7265 の内蔵 RAM アドレス空間

ページ	アドレス A (非キャッシャブルアクセス)	アドレス B (非キャッシャブルアクセス、かつ全 CPU から同じバス経由でアクセス)
RAM0 ページ 0	0xFFF80000～0xFFF83FFF	0xFFD80000～0xFFD83FFF
RAM0 ページ 1	0xFFF84000～0xFFF87FFF	0xFFD84000～0xFFD87FFF
RAM0 ページ 2	0xFFF88000～0xFFF8BFFF	0xFFD88000～0xFFD8BFFF
RAM0 ページ 3	0xFFF8C000～0xFFF8FFFF	0xFFD8C000～0xFFD8FFFF
RAM1 ページ 0	0xFFFA0000～0xFFFA3FFF	0xFFDA0000～0xFFDA3FFF
RAM1 ページ 1	0xFFFA4000～0xFFFA7FFF	0xFFDA4000～0xFFDA7FFF

(3) 3 文字目

'_'固定

(4) 4 文字目以降

任意

17.5.2 セクション一覧

表17.3 カーネルライブラリ、kernel_def.c、kernel_cfg.c のセクション

No.	セクション	備考	No.	セクション	備考
1	PC_hiknl	カーネルプログラム	12	BC_hirmtstk	SVC サーバタスクスタック
2	CC_hicfg	カーネル内部データ	13	BD_hirmtmpf	リモート SVC 用固定長メモリプール領域
3	CC_hijmptbl	カーネル内部データ	14	BD_hiwrk	カーネル内部データ
4	CC_hivct	カーネル内部データ *2	15	BD_hitooltrc	ツールトレース用領域 *3
5	CC_hiinttbl	カーネル内部データ	16	BC_hitrcbuf	ターゲットトレース用バッファ
6	BC_hivct	カーネル内部データ *2	17	BL_S_hiwrk	カーネル内部データ(MYCPUID=1 の場合のみ) *1
7	BC_hiinttbl	カーネル内部データ	18	BC_hitskstk	デフォルトタスクスタック用領域
8	BC_hiknlstk	カーネルスタック	19	BC_hidtq	デフォルトデータキュー用領域
9	BC_hiirqstk	割込みスタック	20	BC_himbf	デフォルトメッセージバッファ用領域
10	BC_hitmrstk	タイマスタック	21	BC_himpf	デフォルト固定長メモリプール用領域
11	BC_hiwrk	カーネル内部データ	22	BC_himpl	デフォルト可変長メモリプール用領域

【注】 *1 CPUID#1 側のリンク時にこのセクションに対するシンボルアドレスファイルを出力し、CPUID#2 側でそのシンボルアドレスファイルをリンクする必要があります。

*2 vsta_knl では、system.vector_type に応じて VBR レジスタを以下のように初期化します。

(1) system.vector が ROM または ROM_ONLY_DIRECT の場合
CC_hivct セクション先頭アドレス-16

(2) system.vector が RAM または RAM_ONLY_DIRECT の場合
BC_hivct セクション先頭アドレス-16

*3 このセクションのサイズは 4 バイトです。カーネルは、このセクションにトレースデータをライトします。

表17.4 RPC ライブラリ、rpc_table.c のセクション

No.	セクション	備考	No.	セクション	備考
1	PC_rpc		3	BC_rpc	
2	CC_rpc		4	BL_S_rpc	MYCPUID=1 の場合のみ *1

【注】 *1 CPUID#1 側のリンク時にこのセクションに対するシンボルアドレスファイルを出力し、CPUID#2 側でそのシンボルアドレスファイルをリンクする必要があります。

表17.5 スピンロックライブラリのセクション

No.	セクション	備考
1	PC_spin	

表17.6　SH2A-DUAL キャッシュサポートライブラリのセクション

No.	セクション	備考
1	PC_cache	
2	PD_cache	
3	CC_cache	

表17.7　IPI のセクション

No.	セクション	備考
1	PC_ipi	
2	CC_ipi	
3	BC_ipi	
4	BL_S_ipi	MYCPUID=1 の場合のみ *1

【注】 *1　CPUID#1 側のリンク時にこのセクションに対するシンボルアドレスファイルを出力し、CPUID#2 側でそのシンボルアドレスファイルをリンクする必要があります。

表17.8　OAL のセクション

No.	セクション	備考
1	PC_oal	
2	CC_oal	
3	BC_oal	
4	BD_oalpool	OAL 用可変長メモリプール領域

表17.9　サンプルのセクション(CPUID#1)

No	分類	セクション	備考
1	リセット関係	CC_resetvct	リセットベクタテーブル
2		PC_reset	
3		CC_reset	
4		BL_S_URAM0	スタートアップ時の両CPUの同期に使用するフラグ(内蔵 RAM0 に配置) *1
5		BD_URAM1	CPUID#2 が CPUID#1 の共通ハードウェア初期化待ちを行うプログラムを配置する領域(内蔵 RAM1 に配置)
6	cfg ファイルで生成されるセクション	BC_TSKSTK	TaskSend()タスクのスタック
7		BC_DTQ	ダミーデータキュー領域
8		BC_MBF	ダミーメッセージバッファ領域
9		B５２	ダミー固定長メモリプール領域
10		BC_MPL	ダミー可変長メモリプール領域
11	標準ライブラリ，lowsrc.c, otherlib.c	PC_stdlib	
12		CC_stdlib	
13		DC_stdlib	
14		BC_stdlib	
15		RC_stdlib	リンカの ROM オプションによって生成
16		BC_heap	ヒープ領域(lowsrc.c)
17	タイマドライバ	PC_tmrdrv	
18		CC_tmrdrv	
19	システムダウン	PC_sysdwn	
20		BC_sysdwn	
21	サンプル	PC_sample	
22		CC_sample	
23		BC_sample	
24		BL_sample	rpc_init()に渡す rpc_info 構造体(init_task1.c)
25		BD_memcopy	RPC サンプルで使用する非キャッシャブル領域

【注】 *1　CPUID#1 側のリンク時にこのセクションに対するシンボルアドレスファイルを出力し、CPUID#2 側でそのシンボルアドレスファイルをリンクする必要があります。

表17.10　サンプルのセクション(CPUID#2)

No	分類	セクション	備考
1	リセット関係	CC_vresetvct	仮想リセットベクタテーブル
2		PC_reset	
3		CC_reset	
4	cfg ファイルで生成されるセクション	BC_TSKSTK	TaskSend()タスクのスタック
5		BC_DTQ	ダミーデータキュー領域
6		BC_MBF	ダミーメッセージバッファ領域
7		BD_MPF	リモートサービスコール例で使用する固定長メモリプール領域
8		BC_MPF	ダミー固定長メモリプール領域
9		BC_MPL	ダミー可変長メモリプール領域
10	標準ライブラリ，lowsrc.c, otherlib.c	PC_stdlib	
11		CC_stdlib	
12		DC_stdlib	
13		BC_stdlib	
14		RC_stdlib	リンカの ROM オプションによって生成
15		BC_heap	ヒープ領域(lowsrc.c)
16	タイマドライバ	PC_tmrdrv	
17		CC_tmrdrv	
18	システムダウン	PC_sysdwn	
19		BC_sysdwn	
20	サンプル	PC_sample	
21		CC_sample	
22		BC_sample	
23		BL_sample	rpc_init()に渡す rpc_info 構造体(init_task2.c)
24	CPUID#1 のシンボルアドレスファイル(prj_cpuid1.fsy)	P	サイズ 0 の P セクション

17.5.3 共有シンボル(CPUID#1 から CPUID#2 へのシンボルの export)

両 CPU で共有したいシンボルがある場合は､その実体を CPUID#1 側で定義してください。その際、そのシンボル実体に個別のセクション名を付与してください。そして、CPUID#1 側のリンク時に、そのセクション内のシンボルをシンボルアドレスファイル(拡張子.”fsy”)に出力するように指定してください。

CPUID#2 側では、そのシンボルアドレスファイルをアセンブル対象としてプロジェクトに登録してください。これにより、CPUID#2 側のプログラムから CPUID#1 側のシンボルを参照することが可能になります。

なお、CPUID#1、CPUID#2 の間で、互いが互いのシンボルを参照するようにしてはなりません。これは単に、依存関係が循環してしまうためです。

17.5.4 CPUID#2 の仮想リセットベクタテーブル

仮想リセットベクタテーブルは、CPUID#2 側に実体を生成しますが、CPUID#1 側のプログラム(cpuid1¥reset¥reset.src)から参照されます。前述したように、CPU 間のビルド依存関係の循環を避けるために、CPUID#1 から CPUID#2 のシンボルを参照することはできないため、本サンプルでは以下のようにしています。

(1) あらかじめ、仮想リセットベクタを配置するアドレスを決定しておく。
(2) CPUID#1のリンク時には、仮想リセットベクタテーブルのシンボルを、(1)で決定したアドレスに強制定義する。
(3) CPUID#2のリンク時には、(1)で決定したアドレスに仮想リセットベクタテーブルのセクションを配置する。

17.5.5 本サンプルのメモリマップ

本サンプルでは、容易にボードでの初期評価を行えるよう、フラッシュメモリを使用せずにSDRAM にダウンロードして実行することを想定したセクション配置にしています。これにより、以下の注意事項があります。

(1) ダウンロード前に、SDRAMをアクセス可能なように初期化する必要があります。
(2) リセットベクタテーブル(CC_resetvctセクション)を0番地に配置することができないため､リセットからの実行はできません｡デバッガでPC, SR, R15をリセット時と同様となるように手動で初期化してから実行させる必要があります。

本製品では、これらを容易に行うためのサンプルバッチファイルを提供しています。詳細は、「17.10 ターゲットへのダウンロード」を参照してください。

(1) プログラム、定数セクションの配置

これらは、最終的には ROM に配置することができるセクションです。

物理ｱﾄﾞﾚｽ: 0x18000000 / 0x18040000 / 0x18080000 / ～～～～～～

CPUID#1

論理ｱﾄﾞﾚｽ		
0x18000000	CC_resetvct	CC_hivct
(ｷｬｯｼｬﾌﾞﾙ)	CC_hijmptbl	CC_hiinttbl
	PC_hiknl	PC_cache *
	CC_cache *	PC_spin
	PC_ipi	CC_ipi
	PC_rpc	CC_rpc
	PC_oal	CC_oal
	CC_hicfg	PC_tmrdrv
	CC_tmrdrv	PC_stdlib
	CC_stdlib	PC_sysdwn
	PC_reset	CC_reset
	C PC_sample	CC_sample
0x1803EFF	DC_stdlib	(空き)
0x3803F000	PD_cache	
(非ｷｬｯｼｬﾌﾞﾙ) 0x3803FFFF	(空き)	

CPUID#2

論理ｱﾄﾞﾚｽ		
0x18040000	CC_vresetvct	CC_hivct
(ｷｬｯｼｬﾌﾞﾙ)	CC_hijmptbl	CC_hiinttbl
	PC_hiknl	PC_cache
	CC_cache	PC_spin
	PC_ipi	CC_ipi
	PC_rpc	CC_rpc
	PC_oal	CC_oal
	CC_hicfg	PC_tmrdrv
	CC_tmrdrv	PC_stdlib
	CC_stdlib	PC_sysdwn
	PC_reset	CC_reset
	PC_sample	CC_sample
	DC_stdlib	P
0x1807EFFF	(空き)	
0x3807F000	PD_cache	
(非ｷｬｯｼｬﾌﾞﾙ) 0x3807FFFF	(空き)	

* 出荷時は、このセクション実体は存在しません。

図17.12 プログラム、定数セクションの配置

RENESAS

(2) 変数セクションの配置 (SDRAM)

物理ｱﾄﾞﾚｽ

物理ｱﾄﾞﾚｽ
～～～～～～～
0x18100000
0x18200000
0x18300000
0x1AFFFFFF

CPUID#1

論理ｱﾄﾞﾚｽ		
0x18100000 (ｷｬｯｼｬﾌﾞﾙ)	BC_hivct *	BC_hiinttbl *
	BC_hiwrk	BC_hidtq
	BC_himbf	BC_himpf
	BC_himpl	BC_hitrcbuf
	BC_hiknlstk	BC_hiirqstk
	BC_hitmrstk	BC_hitskstk
	BC_hirmtstk	BC_rpc
	BC_oal	BC_ipi
	BC_TSKSTK	BC_DTQ
	BC_MBF	BC_MPF
	BC_MPL	BC_stdlib
	RC_stdlib	BC_heap
	BC_sysdwn	BC_sample
0x181EFFFF	(空き)	
0x381F0000 (非ｷｬｯｼｬﾌﾞﾙ)	BD_hirmtmpf	BD_hitooltrc *
	BD_oalpool	BD_memcopy
0x381FFFFF	(空き)	

CPUID#2

論理ｱﾄﾞﾚｽ		
0x18200000 (ｷｬｯｼｬﾌﾞﾙ)	BC_hivct *	BC_hiinttbl *
	BC_hiwrk	BC_hidtq
	BC_himbf	BC_himpf
	BC_himpl	BC_hitrcbuf
	BC_hiknlstk	BC_hiirqstk
	BC_hitmrstk	BC_hitskstk
	BC_hirmtstk	BC_rpc
	BC_oal	BC_ipi
	BC_TSKSTK	BC_DTQ
	BC_MBF	BC_MPF
	BC_MPL	BC_stdlib
	RC_stdlib	BC_heap
	BC_sysdwn	BC_sample
0x182EFFFF	(空き)	
0x382F0000 (非ｷｬｯｼｬﾌﾞﾙ)	BD_hirmtmpf	BD_hitooltrc *
	BD_oalpool	BD_MPF
0x382FFFFF	(空き)	

* 出荷時は、このセクション実体は存在しません。

図17.13　変数セクションの配置（SDRAM）

(3) 変数セクションの配置 (内蔵 RAM)

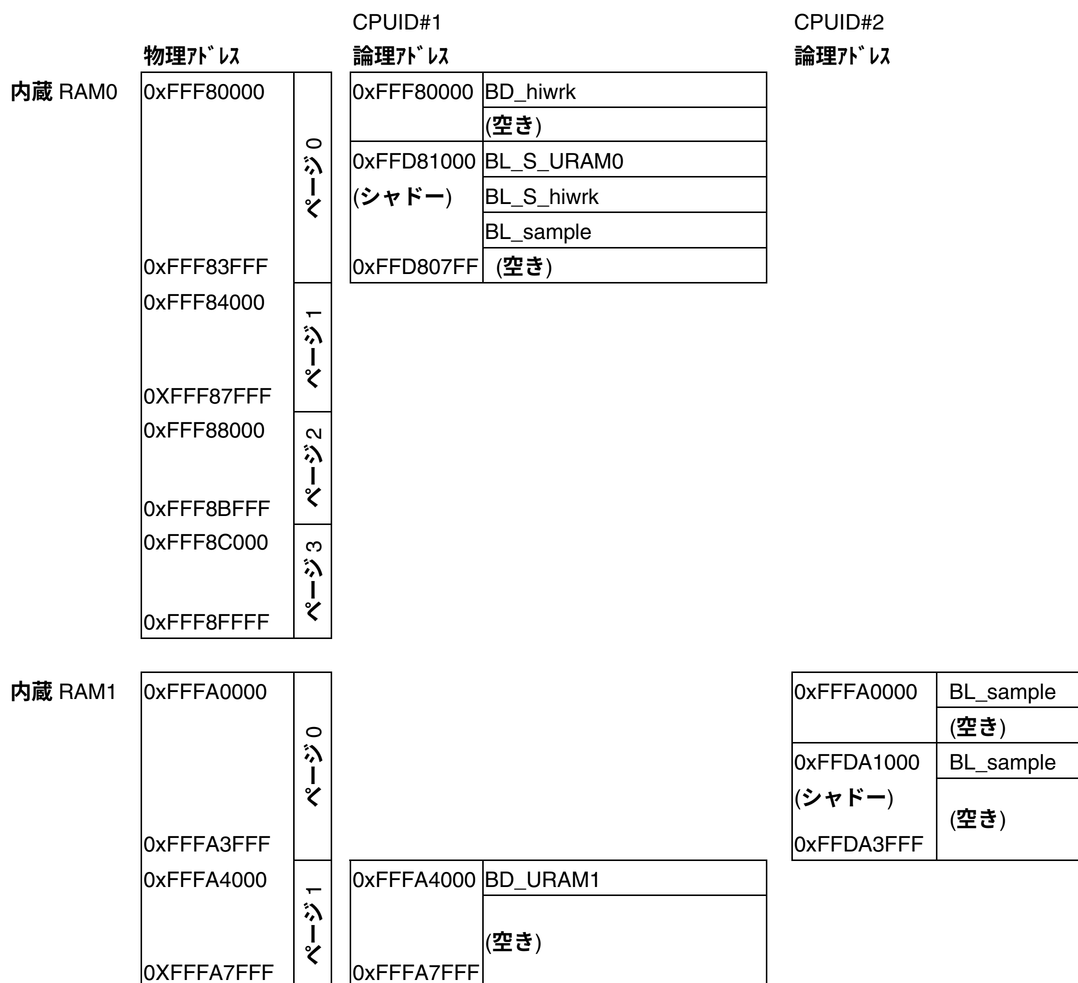


図17.14　変数セクションの配置（内蔵 RAM）

17.6　カーネルライブラリ

カーネルライブラリはいくつかのファイルに分割されており、これらを適切な優先順位で入力する必要があります。High-performance Embedded Workshop では、ライブラリ指定画面で上に表示されるほど優先扱いになります。この優先順位設定を誤ると、リンクでエラーが発生しなくても正常動作しないので、注意してください。

表17.11　カーネルライブラリの優先順位

使用する CPU コア	リンク優先順位
SH-2A	hiknl.lib
SH2A-FPU	(1) fpu_knl.lib (2) hiknl.lib

17.7　各 CPU のビルド順序

17.7.1　基本形

先に CPUID#1 側のビルドを行い、次に CPUID#2 側のビルドを行うのが基本形です。

「17.5.3 共有シンボル(CPUID#1 から CPUID#2 へのシンボルの export)」に記載のように、CPUID#1 側のシンボルを CPUID#2 側に export するために CPUID#1 側の最適化リンケージエディタが出力するシンボルアドレスファイルは、CPUID#2 側のアセンブラフェーズへの入力となります(図 17.15)。すなわち、CPUID#2 側のビルドは、CPUID#1 側のビルドが完了した後に行う必要があります。

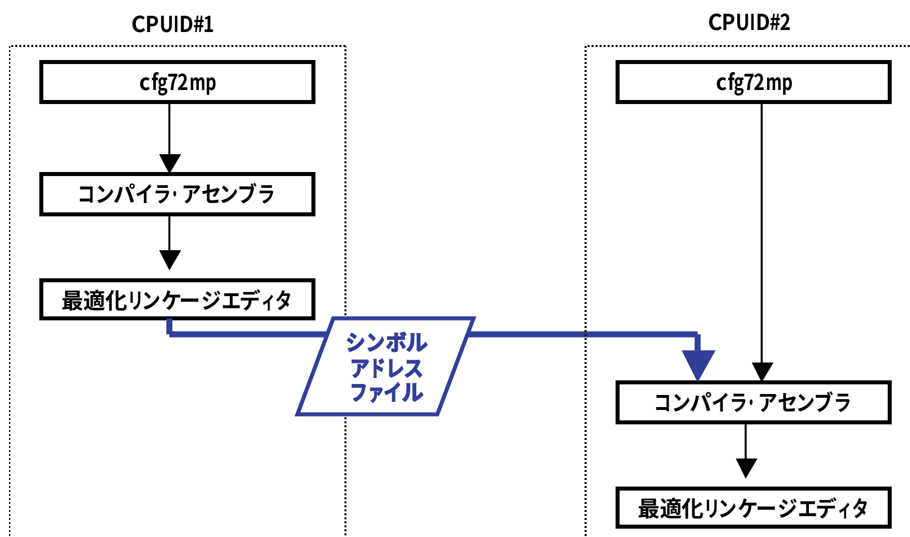


図17.15　各 CPU のビルドフェーズの依存関係(基本形)

17.7.2　ID 名称を export する場合

(1) CPUID#1 が CPUID#2 の ID 名称ヘッダファイルをインクルードする場合(基本形から逸脱)

図 17.16に、各 CPU でのビルドフェーズの依存関係を示します。CPUID#1 側のビルド時は、事前に CPUID#2 側の cfg72mp を実行して CPUID#2 の ID 名称ヘッダファイルを更新する必要があります。

なお、サンプルはこの形になっています。

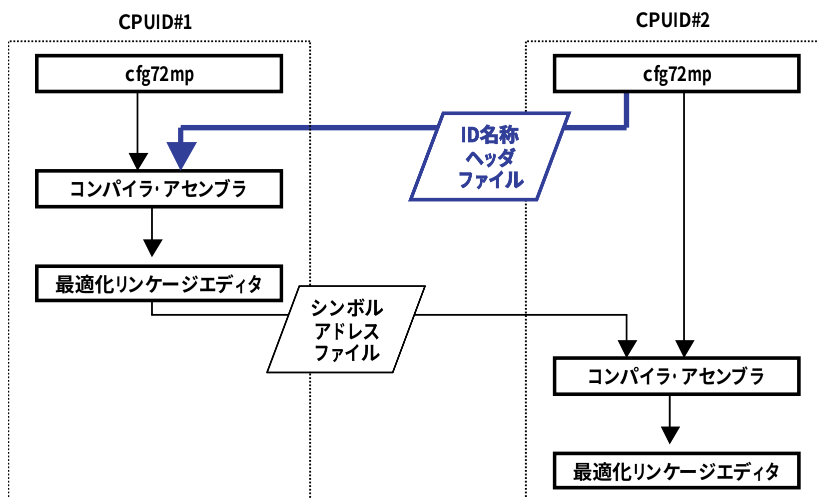


図17.16　CPUID#1 が CPUID#2 の ID 名称ヘッダファイルをインクルードする場合

(2) CPUID#2 が CPUID#1 の ID 名称ヘッダファイルをインクルードする場合(基本形と同じ)

図 17.17に、各 CPU でのビルドフェーズの依存関係を示します。CPUID#2 側のビルド時は、事前に CPUID#1 側の cfg72mp を実行して CPUID#1 の ID 名称ヘッダファイルを更新する必要がありますが、これは基本形と同じです。

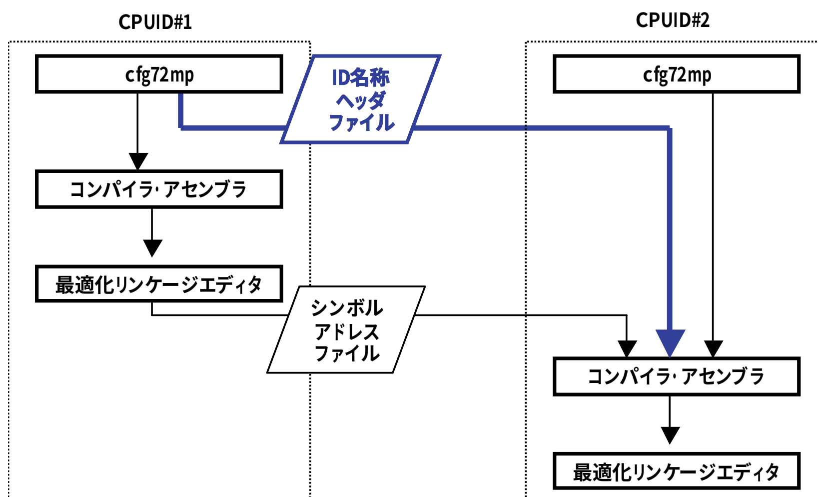


図17.17　CPUID#2 が CPUID#1 の ID 名称ヘッダファイルをインクルードする場合

(3) 両 CPU が互いの ID 名称ヘッダファイルをインクルードする場合(基本形から逸脱)

図 17.18に、各 CPU でのビルドフェーズの依存関係を示します。まず、両 CPU で cfg72mp を実行して各 CPU の ID 名称ヘッダファイルを更新し、その後各 CPU で基本系の通りにビルドする必要があります。

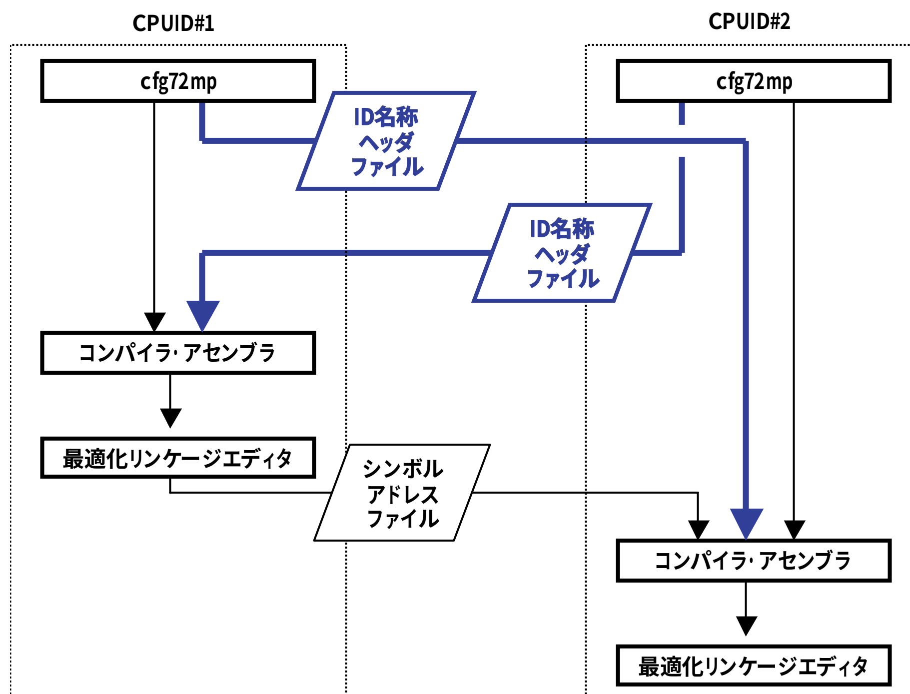


図17.18 両 CPU が互いの ID 名称ヘッダファイルをインクルードする場合

17.8　CPUID#1 側のビルド解説(cpuid1¥cpuid1.hws)

CPUID#1 側のワークスペースファイルは、cpuid1¥cpuid1.hws です。この cpuid1.hws を開いてください。cpuid1.hws には、”prj_cpuid1”というプロジェクトが含まれています。このプロジェクトを用いて、CPUID#1 側のロードモジュールファイルを生成します。
本節では、提供するプロジェクトの主要な設定内容について解説します。

17.8.1　登録されているソース

図 17.19に、prj_cpuid1 プロジェクトに登録されているソースを示します。cpuid1¥以下のディレクトリにある全ソース(C 言語ソース、アセンブリ言語ソース、cfg ファイル)が登録されています。各ソースについては「16. サンプルプログラム」を参照してください。

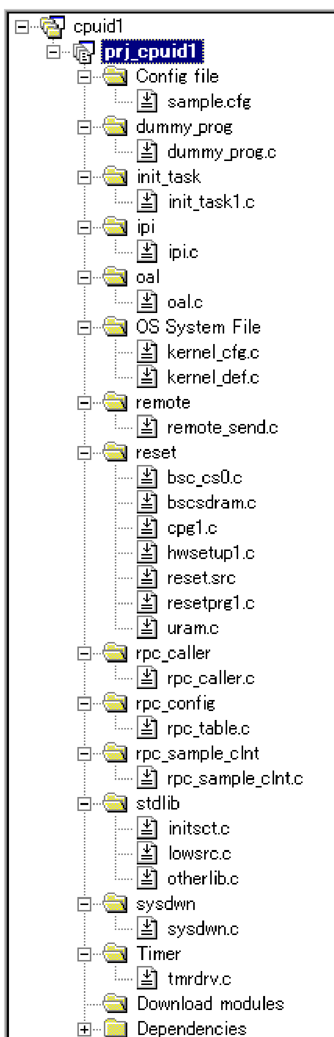


図17.19　prj_cpuid1 プロジェクトに登録されているソース

RENESAS

以下、特記すべき事項について説明します。

(1) Config file (sample.cfg)

cfg ファイルです。cpuid1¥cfg_out¥ディレクトリに格納されています。

(2) OS system file (kernel_def.c, kernel_cfg.c)

これらは cfg72mp 出力ファイルを取り込むためのファイルで、cpuid1¥cfg_out¥ディレクトリに格納されています。ユーザは、これらのファイルを変更してはなりません。

(3) cpuid1¥remote_svc_sample¥remote_send.c

このファイルは、CPUID#2 側の cfg72mp 実行によって生成される kernel_id_cpu2.h をインクルードしています。CPUID#2 側で cfg72mp を実行した場合は、本ファイルの再コンパイルが必要です。

17.8.2　コンパイラオプション

(1) インクルードディレクトリ

図 17.20に、全ソースの共通設定を示します。

図17.20　コンパイラ・インクルードディレクトリ(共通設定)

remote_send.c については、CPUID#2 側の kernel_id_cpu2.h をインクルードしているため、図 17.21 のようにその格納パスを追加しています。なお、「$(WORKSPDIR)¥cfg_out」(cpuid1¥cfg_out¥)と「$(WORKSPDIR)¥..¥cpuid2¥cfg_out」(cpuid2¥cfg_out¥)には、それぞれ CPUID#1 用、CPUID#2 用の kernel_id.h などの同名ファイルが格納されているため、必ず CPUID#1 用の「$(WORKSPDIR)¥cfg_out」が優先(画面上で上方)となるようにしなければなりません。

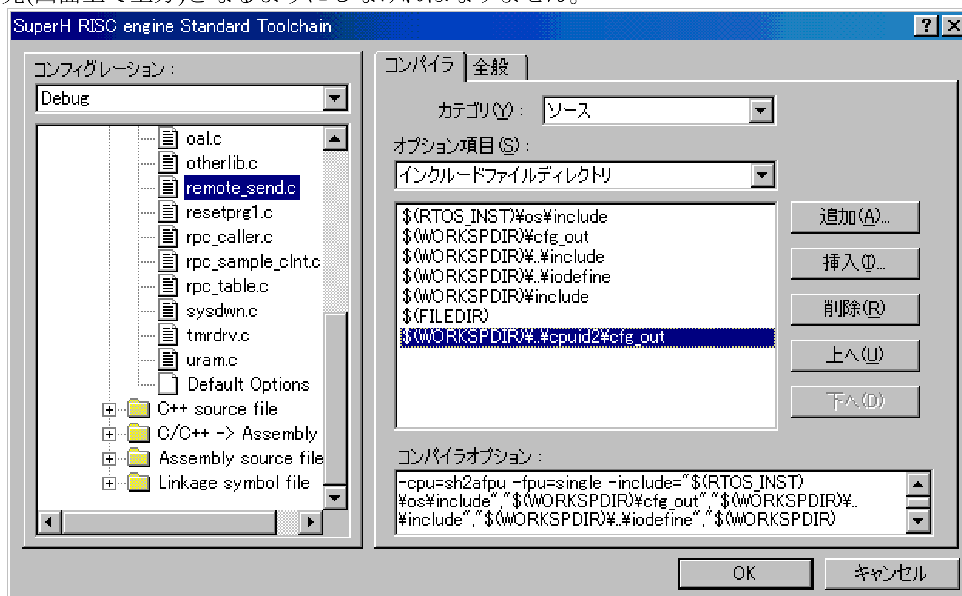


図17.21　コンパイラ・インクルードディレクトリ(remote_send.c)

また、kernel_cfg.c および kernel_def.c については、「$(RTOS_INST)¥os¥system」を定義しています。

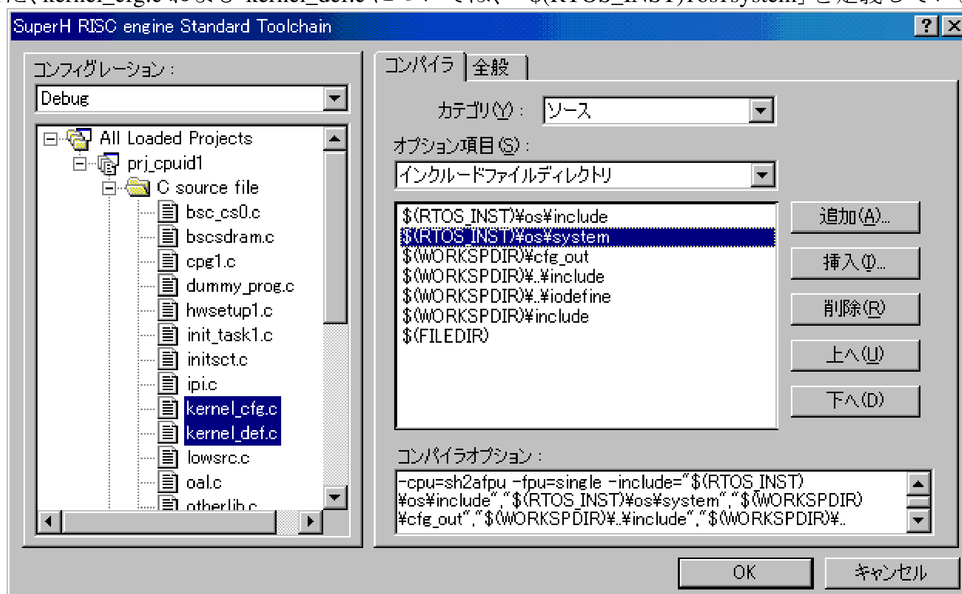


図17.22　コンパイラ・インクルードディレクトリ(kernel_cfg.c, kernel_def.c)

(2) マクロ定義

本サンプルでは、標準ライブラリをリエントラントライブラリとして使用しているため、図 17.23 のようにマクロ"_REENTRANT"を定義しています。

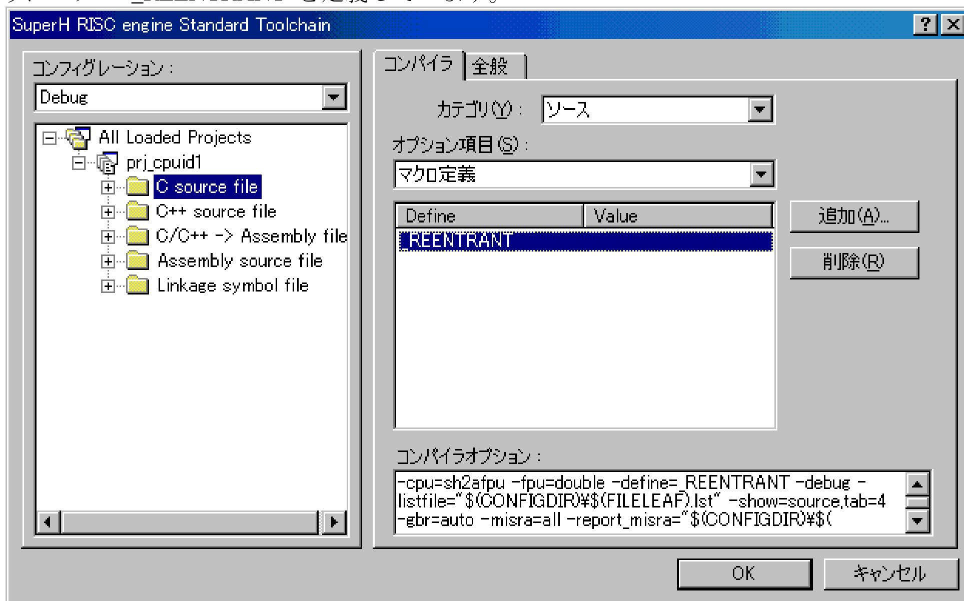


図17.23　コンパイラ・マクロ定義

(3) 出力ファイル形式

kernel_def.c のみ、インラインアセンブルを使用しているため、図 17.24のように出力ファイル形式を「アセンブリプログラム」としています。

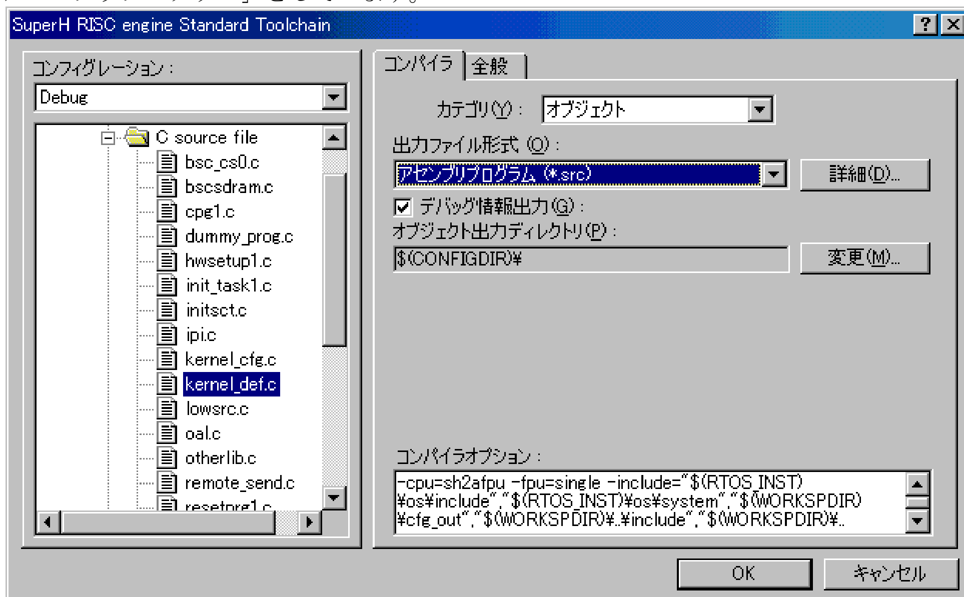


図17.24　コンパイラ・出力ファイル形式

RENESAS

17.8.3　標準ライブラリ構築ツール

(1) 組み込む標準ライブラリ機能

「16.7 標準ライブラリ」で説明しているように、本サンプルでは図 17.25のように stdlib.h と string.h のみを選択しています。

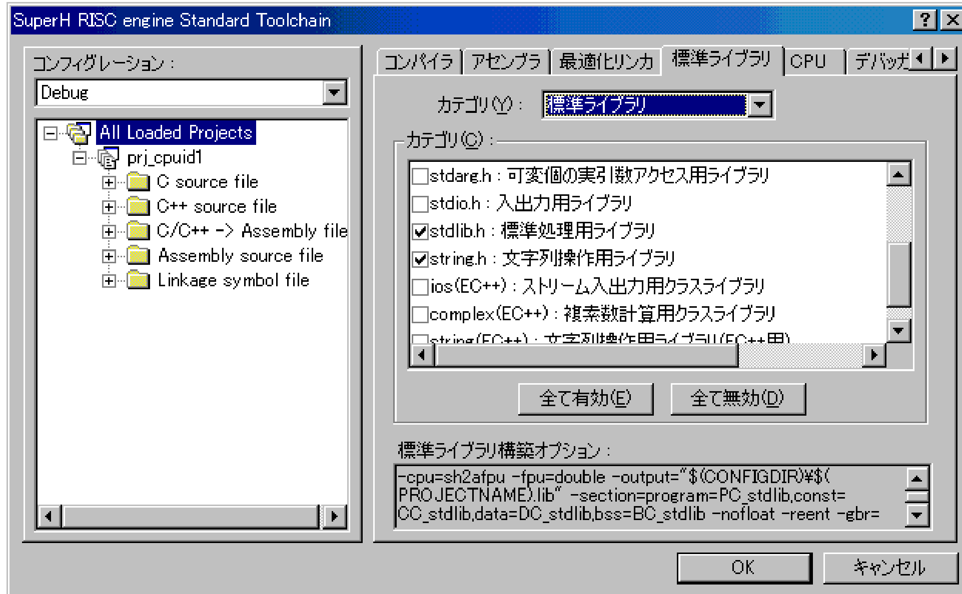


図17.25　標準ライブラリ構築ツール・ライブラリ機能の選択

(2) オブジェクト

図 17.26のように、リエントラントライブラリとして生成しています。

図17.26　標準ライブラリ構築ツール・リエントラントライブラリ

また、[詳細]ボタンで開く[Object details]ダイアログボックスでは、図 17.27のようにセクション名を設定しています。なお、lowsrc.c, otherlib.c でも同じセクション名を使用しています。

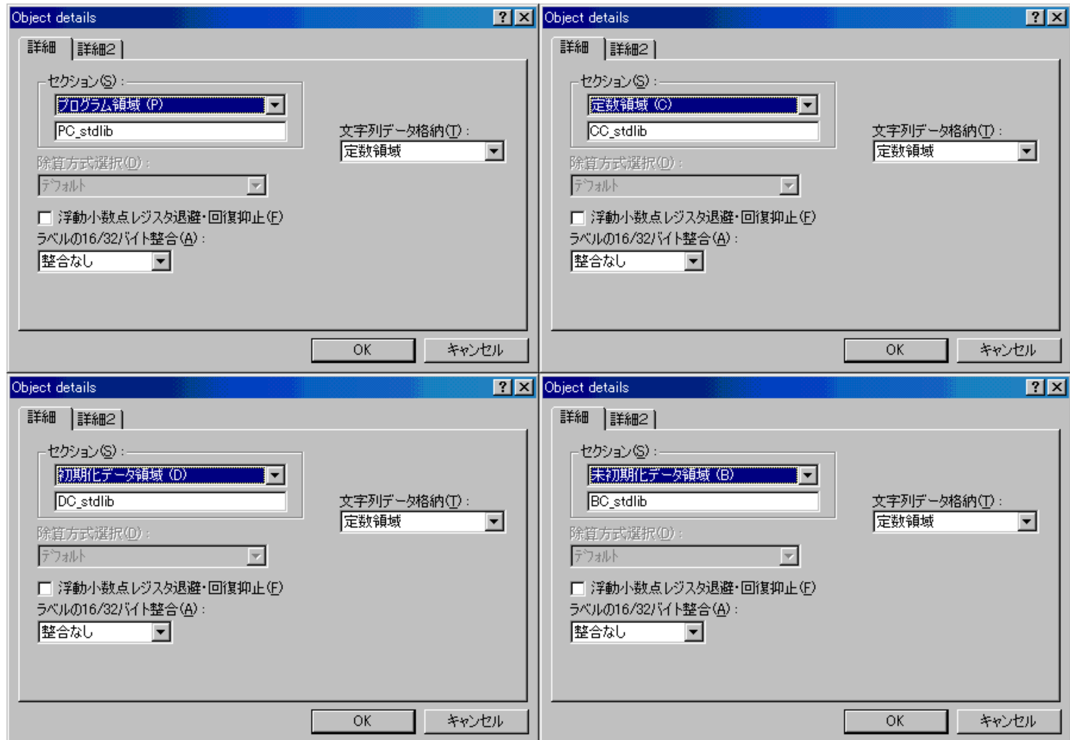


図17.27　標準ライブラリ構築ツール・セクション名の設定

17.8.4　最適化リンカ

(1) ライブラリの入力

図 17.28のように、HI7200/MP が提供する以下のライブラリを入力しています。カーネルライブラリの入力については、「17.6 カーネルライブラリ」を参照してください。

- fpu_knl.lib (カーネル)
- hiknl.lib (カーネル)
- sh2adual_cache.lib (SH2A-DUAL 用キャッシュサポートライブラリ)
- rpc.lib (RPC ライブラリ)
- spinlock.lib (スピンロックライブラリ)

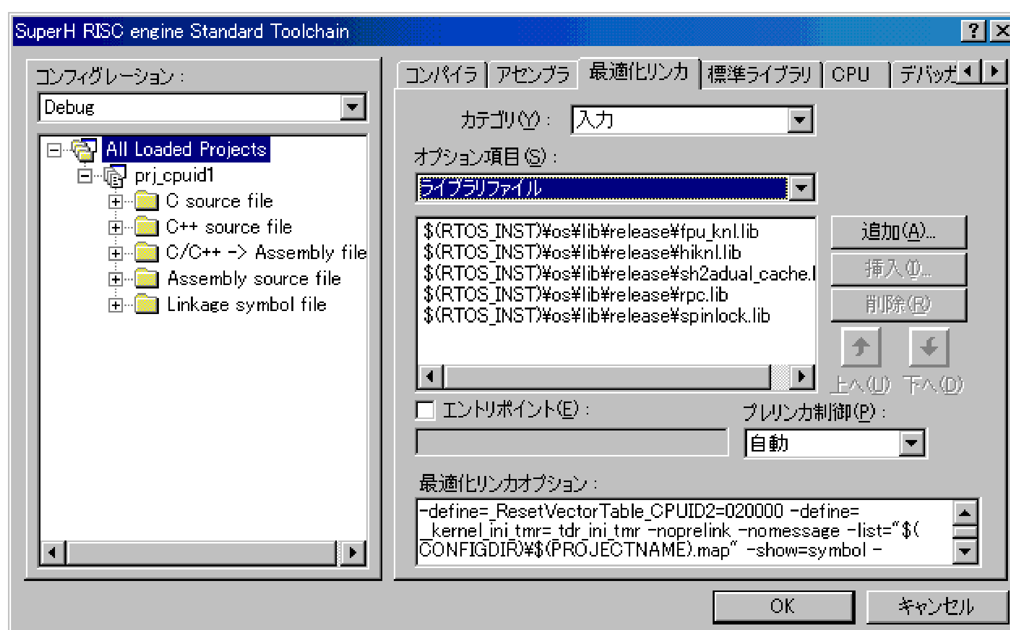


図17.28　最適化リンカ・ライブラリの入力

(2) CPUID#2 の仮想リセットベクタテーブルのシンボル定義

「17.5.4 CPUID#2 の仮想リセットベクタテーブル」で説明したように、図 17.29のように CPUID#2 の仮想リセットベクタテーブルのシンボル「_ResetVectorTable_CPUID2」のアドレスを 0x18040000 に強制定義しています。

図17.29　最適化リンカ・CPUID#2 の仮想リセットベクタテーブルのシンボル定義

(3) セクション配置

図 17.30では全セクションを確認できませんが、「17.5.5 本サンプルのメモリマップ」に記載の通りに各セクションを配置しています。

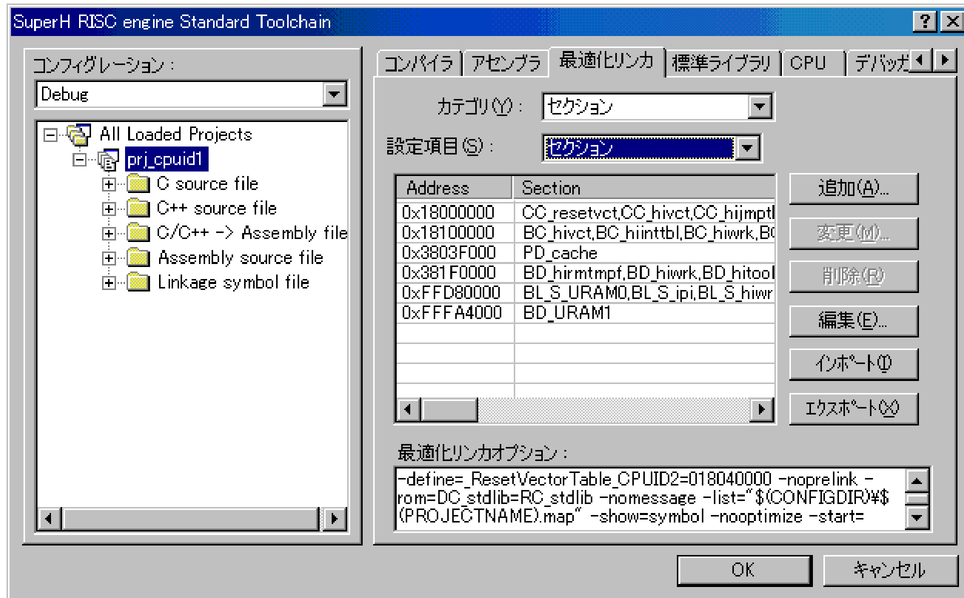


図17.30　最適化リンカ・セクション配置

(4) ROM から RAM へのマップ

初期化データセクションなど､ROM から RAM にマップする必要があるセクションを設定します。本サンプルでは、図 17.31のように設定しています。

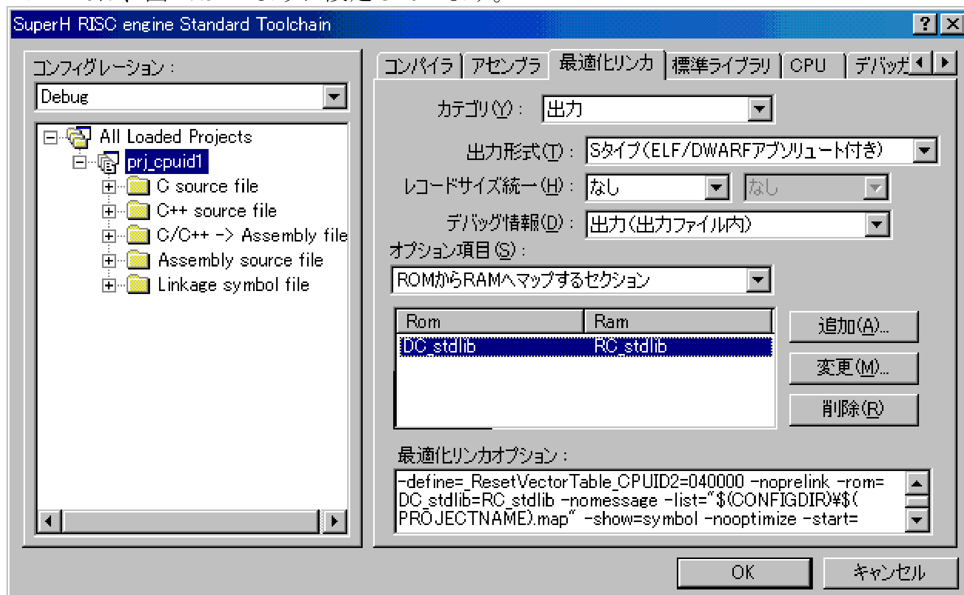


図17.31　最適化リンカ・ROM から RAM へのマップ

(5) シンボルアドレスファイルの出力

「17.5.3 共有シンボル(CPUID#1 から CPUID#2 へのシンボルの export)」で説明しているように、CPUID#1 側のシンボルを CPUID#2 側に公開するために、図 17.32のように設定しています。なお、シンボルファイルは”prj_cpuid1.fsy”というファイル名で、High-performance Embedded Workshop コンフィギュレーションディレクトリである cpuid1¥prj_cpuid1¥debug¥ディレクトリに生成されます。prj_cpuid1.fsy は、CPUID#2 側のソースとして使用します。

図17.32 最適化リンカ・シンボルアドレスファイルの出力

(6) 注意事項

1. L1100ワーニング
   リンク時に、指定されたセクションが見つからないというL1100ワーニング(下記)が出力される場合があります。
   L1100 (W) Cannot find "PC_cache" specified in option "start"
   見つからないセクションが「17.5.2 セクション一覧」に記載されたセクションの場合は、コンフィギュレーションによって存在しない場合があるため、問題ありません。
2. L1320ワーニング
   複数のカーネルライブラリを指定した場合、リンク時にL1320ワーニング(下記)が大量に出力されることがあります。これは、本カーネルが複数のライブラリファイルに同一シンボル・別プログラムを格納する実装方式を採用しているためです。生成されるロードモジュールに問題はありません。
   L1320 (W) Duplicate symbol "__kernel_act_tsk" in "C:¥…fpu_knl.lib(fpu_acttsk)"

RENESAS

17.9 CPUID#2 側のビルド解説(cpuid2¥cpuid2.hws)

CPUID#2 側のワークスペースファイルは、cpuid2¥cpuid2.hws です。この cpuid2.hws を開いてください。cpuid2.hws には、"prj_cpuid2"というプロジェクトが含まれています。このプロジェクトを用いて、CPUID#2 側のロードモジュールファイルを生成します。

本節では、提供するプロジェクトの主要な設定内容について解説します。

17.9.1 登録されているソース

図 17.33に、prj_cpuid2 プロジェクトに登録されているソースを示します。cpuid2¥以下のディレクトリにある全ソース(C 言語ソース、アセンブリ言語ソース、cfg ファイル)が登録されています。各ソースについては「16. サンプルプログラム」を参照してください。

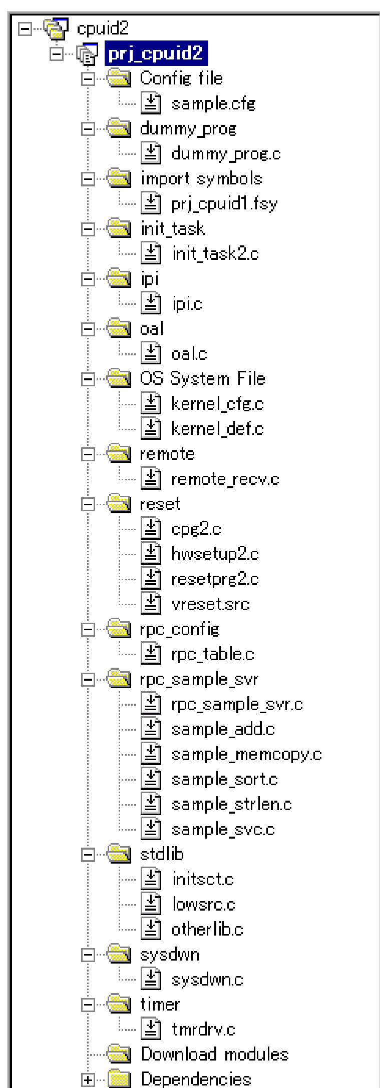


図17.33 prj_cpuid2 プロジェクトに登録されているソース

以下、特記すべき事項について説明します。

(1) Config file (sample.cfg)

cfg ファイルです。cpuid2¥cfg_out¥ディレクトリに格納されています。

(2) OS system file (kernel_def.c, kernel_cfg.c)

これらは cfg72mp 出力ファイルを取り込むためのファイルで、cpuid2¥cfg_out¥ディレクトリに格納されています。ユーザは、これらのファイルを変更してはなりません。

(3) import symbols (prj_cpuid1.fsy)

これは CPUID#1 側が export したシンボルアドレスファイルで、cpuid1¥prj_cpuid1¥debug¥ディレクトリに格納されています。なお、このディレクトリは CPUID#1 側の High-performance Embedded Workshop コンフィギュレーションディレクトリです。CPUID#1 側の High-performance Embedded Workshop コンフィギュレーション名を変更した場合は、手動でこのファイルの登録をやり直す必要があることに注意してください。

また、このファイルは CPUID#1 側のリンクによって生成されます。このため、CPUID#1 側のリンクを実行した場合は、本ファイルの再アセンブルが必要です。

17.9.2 コンパイラオプション

(1) インクルードディレクトリ

図 17.34に、全ソースの共通設定を示します。

図17.34 コンパイラ・インクルードディレクトリ

また、kernel_cfg.c および kernel_def.c については、「$(RTOS_INST)¥os¥system」を追加しています。

図17.35 コンパイラ・インクルードディレクトリ(kernel_cfg.c, kernel_def.c)

(2) マクロ定義

本サンプルでは、標準ライブラリをリエントラントライブラリとして使用しているため、図 17.36 のようにマクロ"_REENTRANT"を定義しています。

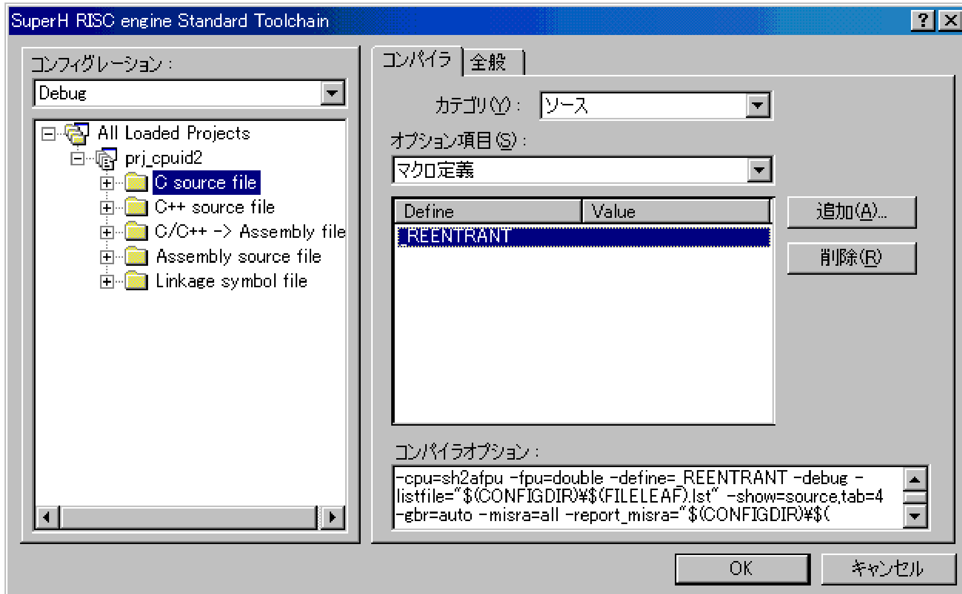


図17.36　コンパイラ・マクロ定義

(3) 出力ファイル形式

kernel_def.c はインラインアセンブルを使用しているため、図 17.37のように出力ファイル形式を「アセンブリプログラム」としています。

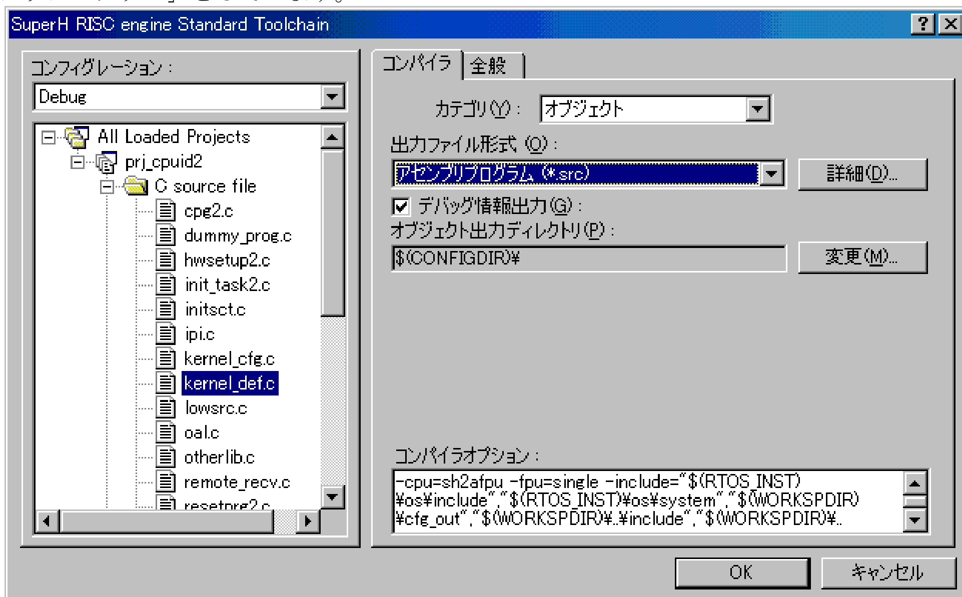


図17.37　コンパイラ・出力ファイル形式

RENESAS

17.9.3 標準ライブラリ構築ツール

(1) 組み込む標準ライブラリ機能

「16.7 標準ライブラリ」で説明しているように、本サンプルでは図 17.38のように stdlib.h と string.h のみを選択しています。

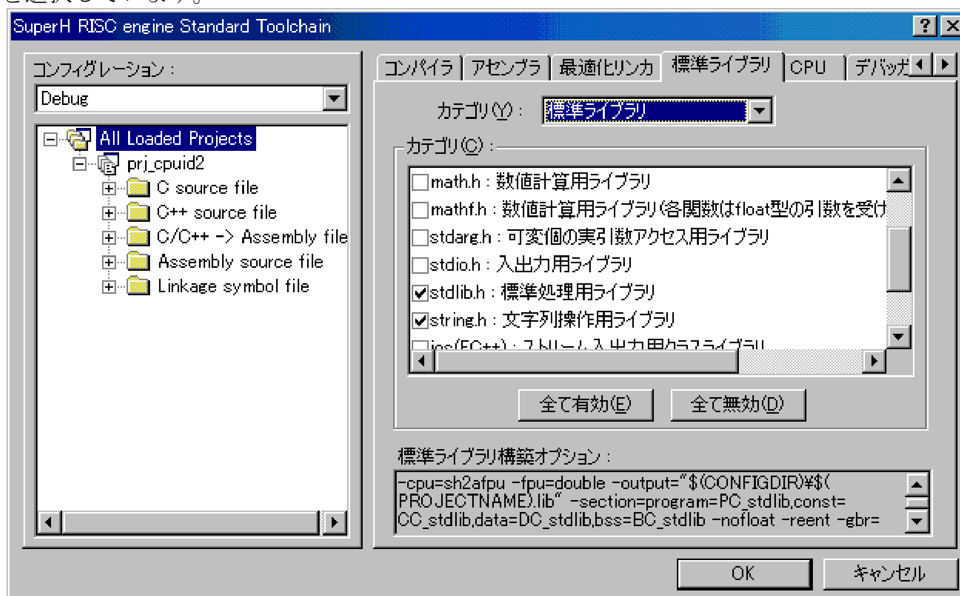


図17.38 標準ライブラリ構築ツール・ライブラリ機能の選択

(2) オブジェクト

図 17.39のように、リエントラントライブラリとして生成しています。

図17.39 標準ライブラリ構築ツール・リエントラントライブラリ

また、[詳細]ボタンで開くの[Object details]ダイアログボックスでは、図 17.40のようにセクション名を設定しています。なお、lowsrc.c, otherlib.c でも同じセクション名を使用しています。

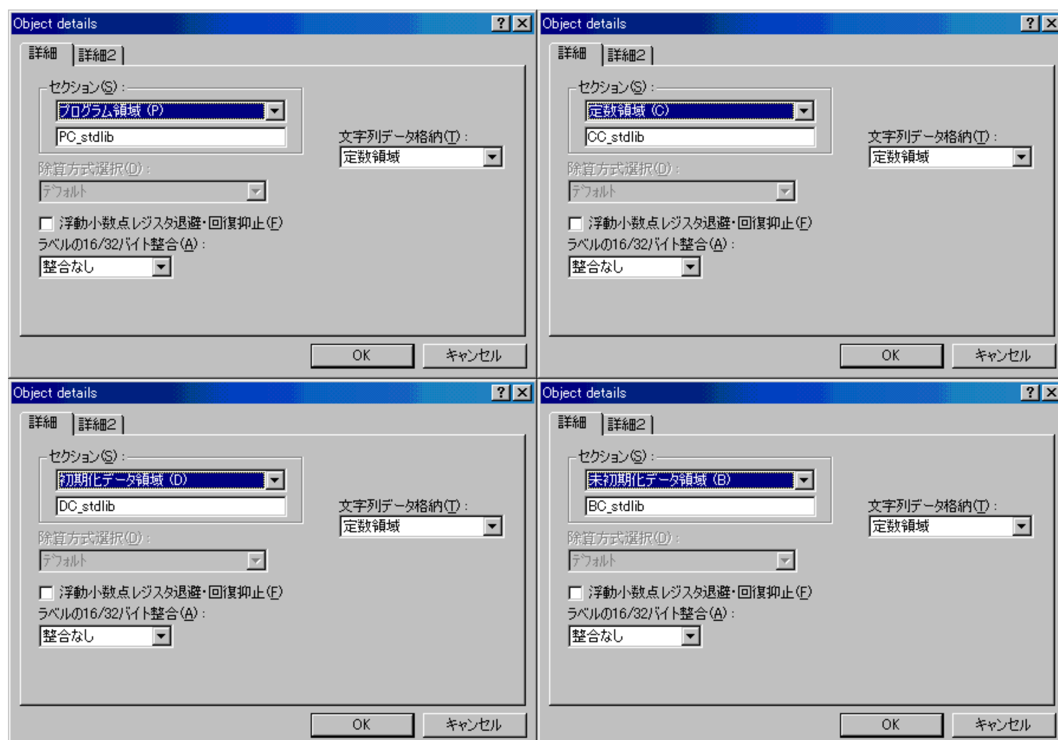


図17.40 標準ライブラリ構築ツール・セクション名の設定

RENESAS

17.9.4　最適化リンカ

(1) ライブラリの入力

図 17.41のように、HI7200/MP が提供する以下のライブラリを入力しています。カーネルライブラリの入力については、「17.6 カーネルライブラリ」を参照してください。

- fpu_knl.lib (カーネル)
- hiknl.lib (カーネル)
- sh2adual_cache.lib (SH2A-DUAL 用キャッシュサポートライブラリ)
- rpc.lib (RPC ライブラリ)
- spinlock.lib (スピンロックライブラリ)

図17.41　最適化リンカ・ライブラリの入力

(2) セクション配置

図 17.42では全セクションを確認できませんが、「17.5.5 本サンプルのメモリマップ」に記載の通りに各セクションを配置しています。特に注意すべき点は、仮想リセットベクタテーブルのセクション”CC_vresetvct”を「17.8.4(2) CPUID#2 の仮想リセットベクタテーブルのシンボル定義」で”_ResetVectorTable_CPUID2”のシンボルを定義したアドレス(0x18040000)に配置することです。

図17.42　最適化リンカ・セクション配置

RENESAS

(3) ROM から RAM へのマップ

初期化データセクションなど､ROM から RAM にマップする必要があるセクションを設定します。本サンプルでは、図 17.43のように設定しています。

図17.43　最適化リンカ・ROM から RAM へのマップ

(4) 注意事項

1. L1100ワーニング
   リンク時に、指定されたセクションが見つからないというL1100ワーニング(下記)が出力される場合があります。
   L1100 (W) Cannot find "PC_cache" specified in option "start"
   見つからないセクションが「17.5.2 セクション一覧」に記載されたセクションの場合は、コンフィギュレーションによって存在しない場合があるため、問題ありません。
2. L1320ワーニング
   複数のカーネルライブラリを指定した場合、リンク時にL1320ワーニング(下記)が大量に出力されることがあります。これは、本カーネルが複数のライブラリファイルに同一シンボル・別プログラムを格納する実装方式を採用しているためです。生成されるロードモジュールに問題はありません。
   L1320 (W) Duplicate symbol "__kernel_act_tsk" in "C:¥…fpu_knl.lib(fpu_acttsk)"

RENESAS

17.10 ターゲットへのダウンロード

本節では、生成したサンプルのロードモジュールを R0K57265D000BR の SDRAM にダウンロードして実行する手順について、簡単に解説します。ユーザ作成実機を使用する場合でも基本手順は同じですが、フラッシュメモリにダウンロードする場合などの詳細は、E10A-USB などのマニュアルを参照してください。

1. CPUID#1のワークスペースを開きます。
2. CPUID#2のワークスペースを開きます。
3. CPUID#1のHigh-performance Embedded Workshopからターゲットに接続します。
4. CPUID#2のHigh-performance Embedded Workshopからターゲットに接続します。
5. 両方のHigh-performance Embedded Workshopからリセットコマンドを入力します。
6. CPUID#1のHigh-performance Embedded Workshopで、コマンドラインを用いてSDRAM等を初期化（これにより、SDRAMへのダウンロードが可能になります）
7. それぞれのHigh-performance Embedded Workshopから、SDRAMにロードモジュールをダウンロードします。
8. CPUID#1側で生成したリセットベクタテーブル(_ResetVectorTable)はSDRAMにダウンロードされているので、このテーブルを元にそれぞれのHigh-performance Embedded WorkshopでPC, R15を初期化します。具体的な初期値は、以下の通りです。なお、フラッシュメモリにダウンロードする場合は、本項番の処理は不要です。
   ・PC：CPUID#1の_ResetVectorTable番地の内容(=CPUID#1のreset.srcの_Reser_Poweron)
   ・R15：CPUID#1の(_ResetVectorTable+4)番地の内容(=内蔵RAM0の最終アドレス)

本製品では、項番 6, 8 を簡略化するための以下の High-performance Embedded Workshop バッチファイルを提供しています。

(1) cpuid1¥prj_cpuid1¥hwsetup.hdc
    SDRAM初期化など、HardwareSetup_CPUID1()相当の初期化を行います。
(2) cpuid1¥prj_cpuid1¥reset_cpu1.hdc
    CPUID#1に関して、上記項番8に示した初期化を行います。
(3) cpuid2¥prj_cpuid2¥reset_cpu2.hdc
    CPUID#2に関して、上記項番8に示した初期化を行います。PCは0x180000000番地、R15は0x18000004番地のメモリ内容を元に初期化します。0x18000000番地はCPUID#1側での_ResetVectorTable (CC_resetvctセクション)の配置アドレスです。CPUID#1側でCC_resetvctセクションの配置アドレスを変更した場合は、バッチファイルを変更後のアドレスに修正してください。

RENESAS

18. スタックサイズの算出方法

18.1 スタックの種類

ユーザは、タスクやハンドラの実行にどれだけのスタック使用量が必要かを見極め、個々のタスクやハンドラに割当てる必要があります。スタックのオーバフローはシステムの異常動作を引き起こすので、本節を参考にして十分なスタック領域を確保してください。

スタックには、以下の種類があります。

- タスクのスタック
- 割込みスタック(ノーマル割込みハンドラ)
- ダイレクト割込みハンドラのスタック
- タイマスタック
- カーネルスタック

■カーネル起動前のスタック

リセット直後など、カーネル起動前に実行されるプログラムのスタックは、カーネルの管理外です。このため、任意の領域をスタックとして使用することができます。

内蔵 RAM を持つマイコンでは、通常はリセット時のスタックを内蔵 RAM にしてください。内蔵 RAM を持たないマイコンでは、リセット直後の BSC(バスステートコントローラ)の状態ではリセット時のスタック（実装されている外部 RAM）にアクセスできない場合があります。このような場合は、RAM が使用できるように BSC の設定が完了するまで、プログラムでスタックを使用しないのはもちろん、割込みや例外も起こさないようにする必要があります。これは、割込みや例外が発生するとスタックへのレジスタ保存が行われるためです。

18.2 スタックサイズ計算の基本

次節以降で、タスクやハンドラなどに必要なスタックサイズの算出方法を説明しますが、その前にサイズを算出するための基本的事項について説明します。

18.2.1 関数ツリーによる消費サイズ

タスクやハンドラのエントリ関数など、アプリケーションプログラムの実行が開始される関数を起点とした関数ツリーによる消費サイズを求めてください。

図 18.1の場合、関数ツリーによる消費サイズは、16 + 20 + 32 = 68 バイトとなります。

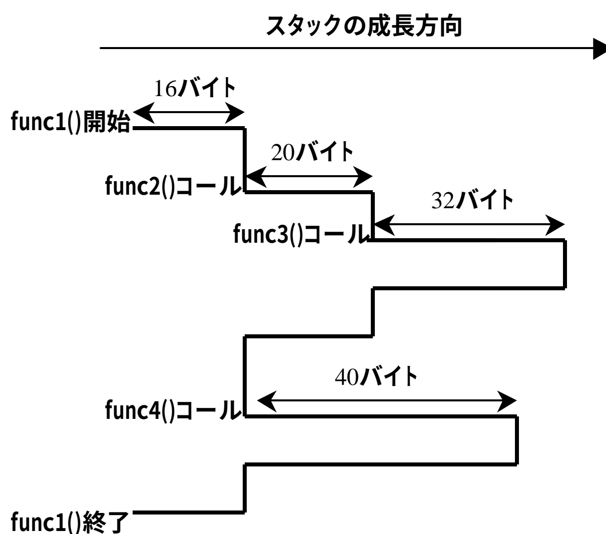


図18.1 関数ツリーによる消費サイズ

なお、各関数が使用するスタックサイズは、コンパイルリストファイル中の”frame size”から知ることができます。

RENESAS

18.2.2　カーネルサービスコール

関数呼び出しとして算出してください。具体的な値は、リリースノートを参照してください。

18.2.3　RPC ライブラリの呼び出し

関数呼び出しとして算出してください。具体的な値は、リリースノートを参照してください。

18.2.4　OAL、IPI、SH2A-DUAL キャッシュサポートライブラリ、スピンロックライブラリ

ユーザ作成関数と同様に扱ってください。

18.2.5　拡張サービスコール

拡張サービスコールも、拡張サービスコールルーチンという関数を呼び出すと考えて算出してください。ただし、拡張サービスコール毎に 8 バイトを加算してください。

18.2.6　ノーマル CPU 例外ハンドラ、ダイレクト CPU 例外ハンドラ

これらの CPU 例外ハンドラは、例外発生時点のスタックを引き継いで使用します。つまり、スタック消費という意味では、一種の関数ツリーと考えることができます。このため、関数コールと同様に CPU 例外ハンドラが消費するサイズを加算してください。

加算に当たっては、CPU 例外発生毎に以下の値も加算してください。

- ノーマル CPU 例外ハンドラ：44 バイト
- ダイレクト CPU 例外ハンドラ：8 バイト

18.3 Call Walker 使用時の注意事項

コンパイラパッケージに付属している Call Walker は、シンボル参照による関数呼び出し関係を解析し、その関数ツリーで使用するスタックサイズを表示するユーティリティです。ここでは、Call Walker 使用時の注意事項を説明します。

(1) カーネルのサービスコール

Call Walker はその実現方式上、関数テーブルを用いた呼び出しは、どこでどの関数を呼び出したのかを認識できません。カーネルのサービスコールは関数テーブルを使っているため、このケースに該当します。

図 18.2に関数テーブルを用いた呼び出しを複数回行っている関数 TaskSend()の例を示します。Call Walker では関数テーブル経由の呼び出し回数に関わらず、ひとつだけ”×”アイコンが表示され、使用サイズが 0 バイトとして扱われます。

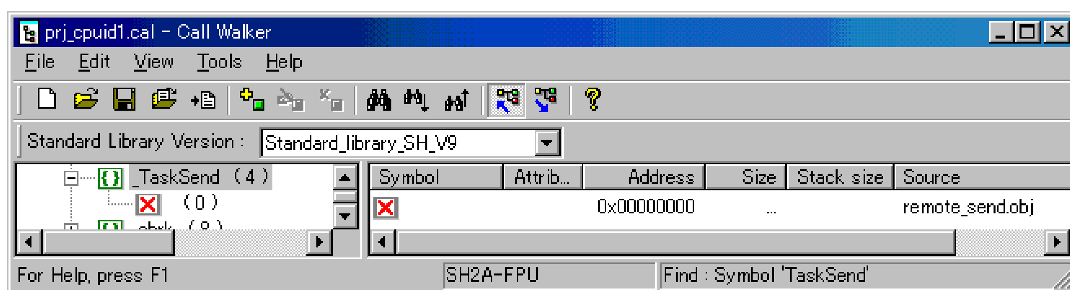


図18.2 関数テーブルを用いた呼び出しを行っている関数の Call Walker 表示例

(2) RPC ライブラリ、および OAL の API 関数

これらの API 関数は内部でカーネルサービスコールを呼び出しているため、Call Walker では実際の値より小さい値が表示されます。

(3) [Realtime OS Option]

Call Walker には、各種リアルタイム OS が使用するサイズのデータベースファイルを取り込み、サービスコール呼び出し箇所にそのファイルで定義されたサイズを加算して表示する機能があります。

しかし、HI7200/MP では「(1) カーネルのサービスコール」で示した理由から、そもそも Call Walker がサービスコール呼び出し箇所およびサービスコール種別を認識できないため、この機能は利用できません。

(4) Call Walker の有効活用法

「(1) カーネルのサービスコール」で示した理由から、サービスコールで使用するサイズを正確に反映するには、各関数が呼び出しているサービスコールを調べて手動で加算する必要があります。しかし、これはかなり煩雑な作業になります。

Call Walker で算出される最大ツリーのサイズに、一律にサービスコール使用サイズの最大値を加算するようにすれば、この作業がきわめて簡単になります。ただし、この方法では、実際に必要なサイズよりも大きなサイズが算出される可能性が高くなります。

RENESAS

18.4　NMI 使用時の注意

本章では、NMI を使用しない前提でサイズ算出方法を提示しています。NMI を使用する場合は、これによる影響をユーザ側で考慮してください。

18.5　スタックサイズの変化に関する注意

必要なスタックサイズは、以下のような要因で変化します。

- 使用するコンパイラのバージョン
- コンパイラの最適化などのオプション
- 使用する HI7200/MP のバージョン

これらを変更したことでスタックが足りなくなる場合には、確保するスタックサイズ、または OS に対して指定するスタックサイズを増やす必要があります。このような手間を避けるため、およびサイズ算出ミスによるオーバフロー予防のために、ある程度マージンを持ったサイズを確保・指定することを推奨します。

18.6 タスクのスタック

基本的に、タスクはタスク ID 毎に異なるスタックを使用します。カーネルは、ディスパッチ時にタスクのスタックを切り替えます。

また、本節に加え、「4.5.4 タスクのスタック」も参照してください。

18.6.1 スタックサイズの算出

必要なサイズは、以下の式で算出されます。

必要なサイズ＝
　　タスクのエントリ関数を起点とした関数ツリーによる消費サイズ　… (a-1)
　＋タスクのコンテキストサイズ … (a-2)
　＋タスク例外処理ルーチンのエントリ関数を起点とした関数ツリーによる消費サイズ　… (b-1)
　＋タスク例外処理ルーチンのコンテキストサイズ … (b-2)
　＋多重割込み加算値　… (c)

(b-1),(b-2)は、タスク例外処理ルーチンを使用しない場合は 0 です。

(a-1)および(b-1)は、「18.2 スタックサイズ計算の基本」に従って求めたサイズです。

(a-2), (b-2)のコンテキストサイズは、表 18.1を参照してください。

タスク例外処理ルーチンは、ネストする使い方をしないのが通常ですが、カーネルの仕様上はタスク例外処理ルーチンはネスト可能です。タスク例外処理ルーチンがネストする場合は、(b-1)はネストを加味したサイズ、(b-2)はネスト数を乗じたサイズとしてください。

表18.1　タスクのコンテキストサイズ

TA_COP1 属性	タスク	タスク例外処理ルーチン
なし	84 バイト	88 バイト
あり	84+72(FPU 分)=156 バイト	88+72(FPU 分)=160 バイト

(c)の多重割込み加算値は、cfg ファイルの設定によって異なります。

なお、ここでは以下の記号を使用します。

UPPINTNST：カーネル割込みマスクレベル(system.system_IPL)より高い割込みネスト数
LOWINTNST：カーネル割込みマスクレベル以下の割込みネスト数

(1) system.vector_typeがROM_ONLY_DIRECTまたはRAM_ONLY_DIRECTの場合
多重割込み加算値＝8×UPPINTNST＋16×LOWINTNST

(2) system.vector_typeがROM_またはRAMの場合
(a) system.regbankがALLの場合
多重割込み加算値＝8×UPPINTNST＋16×LOWINTNST＋α
ただし、LOWINTNSTに、kernel_intspec.hでレジスタバンクを使用できないと定義した割込みが含まれる場合で、その割込みをノーマル割込みとして使用する場合は、αとして36バイトを加算してください。
(b) system.regbankがALL以外の場合
多重割込み加算値＝8×UPPINTNST＋24×LOWINTNST＋20
ただし、LOWINTNSTが0の場合は、下線部を0として計算してください。

18.6.2 スタックサイズの指定箇所

(1) 非スタティックスタックを使用するタスクの場合

(a) cre_tsk, icre_tsk, acre_tsk, iacre_tskによる生成
T_CTSK構造体のstkszに、スタックサイズを指定してください。

(b) cfgファイルでの生成
task[].stack_sizeに、スタックサイズを指定してください。

(2) スタティックスタックを使用するタスクの場合

cfg ファイルで、static_stack[].stack_size にスタックサイズを指定してください。また、static_stack[].tskid に、そのスタックを使用するタスク ID を指定してください。

18.6.3 デフォルトタスクスタック領域サイズの算出(memstk.all_memsize)

デフォルトタスクスタック領域のサイズは、cfg ファイルの memstk.all_memsize に指定します。ここに指定すべき値は、以下のように算出してください。

デフォルトタスクスタック領域サイズ＝
Σ((デフォルトタスクスタックを使用するタスク生成時に指定するスタックサイズ) + 0x10)＋0x1C

なお、デフォルトタスクスタック領域には、専用の"BC_hitskstk"というセクション名が付与されます。

18.6.4 SVC サーバタスクのスタックサイズ(remote_svc.stack_size)

ここには、下記を「18.6.1 スタックサイズの算出」に適用して算出される値を指定してください。

(a-1)：max([A], [B])
[A]：依頼されたリモートサービスコールに対応したローカルサービスコールの使用サイズ
[B]：IPI_send()の使用サイズ
[A]の値は実装に依存するため、リリースノートに掲載しています。
(a-2)：84 バイト(TA_COP1 属性なし)
(b-1), (b-2)：0 バイト(タスク例外処理ルーチン未使用)

18.6.5 RPC サーバタスクとサーバスタブ

サーバスタブは、RPC ライブラリ内の RPC サーバタスクから呼び出されます。
RPC サーバタスクに必要なスタックサイズは、以下で表現されます。

必要なサイズ＝
RPC ライブラリ内の RPC サーバタスク関数による消費サイズ … (a)
＋サーバスタブをタスクエントリ関数として求めた必要サイズ …(b)

(a) 純粋にRPCライブラリ内の関数ツリーによって消費されるサイズで、rpc_init()でrpc_config.ServerTaskStackSizeに指定します。ここに指定すべき値は、リリースノートを参照してください。
(b) サーバスタブをタスクのエントリ関数として「18.6.1 スタックサイズの算出」に従って算出したサイズです。rpc_start_server()およびrpc_start_server_with_paramarea()で指定するrpc_server_info.ulStubStackSizeには、(b)の値を指定してください。

なお、実際にサーバタスクのスタックとして確保されるサイズは、(a)+(b)となります。

また、RPC サーバタスクのスタックは OAL_CreateTask()によってデフォルトタスクスタック用領域から割り当てられます。「18.6.3 デフォルトタスクスタック領域サイズの算出(memstk.all_memsize)」では、このことを考慮して算出してください。

18.7 ノーマル割込みハンドラのスタック(system.stack_size)

割込みスタックは、ノーマル割込みハンドラが使用するスタックで、各 CPU 毎にひとつ存在します。ノーマル割込みが発生すると、カーネルはスタックを割込みスタックに切り替えます。ただし、ノーマル割込みがネストした場合は、カーネルはスタックを切り替えません。

ノーマル割込みハンドラから呼び出されたサービスコール、およびそこからコールバックされる関数も、同じスタックを引き続いて使用します。

割込みスタックは各 CPU 毎にひとつだけ存在します。そのサイズを cfg ファイルの system.stack_size に指定することで、割込みスタック領域が生成されます。割込みスタック領域のセクション名は、"BC_hiirqstk"です。

18.7.1 各ハンドラのスタックサイズの算出

必要なサイズは、以下の式で算出されます。

必要なサイズ＝

ハンドラのエントリ関数を起点とした関数ツリーによる消費サイズ … (a)

(a)は、「18.2 スタックサイズ計算の基本」に従って求めたサイズです。

18.7.2 割込みスタック領域サイズの算出と指定箇所(system.stack_size)

割込みスタック領域のサイズは、cfg ファイルの system.stack_size に指定します。ここに指定すべき値は、以下のように算出してください。

なお、ここでは以下の記号を使用します。

UPPINTNST : カーネル割込みマスクレベル(system.system_IPL)より高い割込みネスト数
LOWINTNST : カーネル割込みマスクレベル以下の割込みネスト数

(1) system.vector_typeがROM_ONLY_DIRECTまたはRAM_ONLY_DIRECTの場合
この場合は、ノーマル割込みハンドラは存在しないため、system.stack_sizeの指定に関わらず、割込みスタック領域は生成されません。

(2) system.vector_typeがROMまたはRAMの場合

(a) system.regbankがALLの場合
割込みスタック領域サイズ＝
Σ(各割込みレベルで最も多く使用するハンドラの使用サイズ)＋4
＋8×UPPINTNST＋<u>16×(LOWINTNST-1)</u>＋α
ただし、LOWINTNSTが0または1の場合は、下線部を0として計算してください。
また、LOWINTNSTに、kernel_intspec.hでレジスタバンクを使用できないと定義した割込みが含まれる場合で、その割込みをノーマル割込みとして使用する場合は、αとして36バイトを加算してください。

(b) system.regbankがALL以外の場合
割込みスタック領域サイズ＝
Σ(各割込みレベルで最も多く使用するハンドラの使用サイズ)＋4
＋8×UPPINTNST＋<u>24×(LOWINTNST-1)＋20</u>
ただし、LOWINTNSTが0または1の場合は、下線部を0として計算してください。

18.8 ダイレクト割込みハンドラのスタック

ダイレクト割込みハンドラでは、アプリケーション側でハンドラ毎に個別のスタックを確保し、ハンドラ起動時にそのスタックに切り替え、ハンドラ終了時に元のスタックに戻さなければなりません。

なお、カーネル割込みマスクレベル(system.system_IPL)より高い割込みレベルのハンドラは、ダイレクト割込みハンドラとして実装しなければなりません。

18.8.1 スタックサイズの算出

各ハンドラで必要なサイズは、以下の式で算出されます。

必要なサイズ＝
　　ハンドラのエントリ関数を起点とした関数ツリーによる消費サイズ　… (a)
　＋多重割込み加算値　… (b)

(a)は、「18.2 スタックサイズ計算の基本」に従って求めたサイズです。
(b)の多重割込み加算値は、cfg ファイルの設定によって異なります。
なお、ここでは以下の記号を使用します。

UPPINTNST：カーネル割込みマスクレベル(system.system_IPL)および自割込みレベルのいずれよりもより高い割込みネスト数
LOWINTNST：カーネル割込みマスクレベル以下で、かつ自割込みレベルよりも高い割込みネスト数

(1) system.vector_typeがROM_ONLY_DIRECTまたはRAM_ONLY_DIRECT、または当該ダイレクト割込みよりも高いレベルのノーマル割込みが存在しない場合
多重割込み加算値＝8×UPPINTNST＋16×LOWINTNST

(2) (1)以外の場合
「18.7 ノーマル割込みハンドラのスタック(system.stack_size)」において、対象割込みを自割込みレベルより高いノーマル割込みに限定して算出される「割込みスタック領域サイズ」を、多重割込み加算値としてください。なお、この算出結果はあくまでもここでの多重割込み加算値です。system.stack_sizeに設定すべき値ではありません。

18.8.2 スタックサイズの指定箇所

ダイレクト割込みハンドラのスタック領域は、ユーザ側で確保してください。詳細は、「12.5.4 ダイレクト割込みハンドラ」を参照してください。

18.8.3 スタックの共有

同じ割込みレベルのダイレクト割込みハンドラ同士は同時に実行されることはないため、スタックを共有することができます。

18.9　タイマスタック(clock.stack_size)

タイマスタックは、カーネル内のタイマ割込み用割込みハンドラが使用するスタックです。以下のプログラムはカーネル内のタイマ割込み用割込みハンドラから呼び出されるため、タイマスタックを使用します。

- タイムイベントハンドラ
- tdr_int_tmr()

また、vstp_tmr からコールバックされる tdr_stp_tmr()と、vrst_tmr, ivrst_tmr からコールバックされる tdr_rst_tmr()もタイマスタックを使用します。

タイマスタックは各 CPU 毎にひとつだけ存在します。そのサイズを cfg ファイルの clock.stack_size に指定することで、タイマスタック領域が生成されます。タイマスタック領域のセクション名は、"BC_hitmrstk"です。

必要なサイズは、以下の式で算出されます。

必要なサイズ＝max([A], [B], [C], [D], [E], [F], [G])＋多重割込み加算値
[A]：TMR_A＋(tdr_int_tmr()の使用サイズ)
[B]：TMR_B＋(タイムイベントハンドラの最大使用サイズ)
[C]：TMR_C＋(タイムイベントハンドラの最大使用サイズ) …vrst_tmr, ivrst_tmr 使用時のみ
[D]：TMR_D
[E]：TMR_E＋ (tdr_stp_tmr()の使用サイズ) … vstp_tmr 使用時のみ
[F]：TMR_F＋(tdr_rst_tmr()の使用サイズ) …vrst_tmr, ivrst_tmr 使用時のみ
[G]：TMR_G

TMR_A などの値は実装に依存するため、リリースノートに掲載しています。

なお、「タイムイベントハンドラの最大使用サイズ」には、system.action が YES の場合はオブジェクト操作機能用の「デバッグデーモン」が自動的に周期ハンドラとして生成されるため、このサイズを考慮する必要があります。デバッグデーモンの使用サイズも実装に依存するため、リリースノートに掲載しています。

多重割込み加算値は、cfg ファイルの設定によって異なります。
なお、ここでは以下の記号を使用します。
UPPINTNST：カーネル割込みマスクレベル(system.system_IPL)より高い割込みネスト数
LOWINTNST：カーネル割込みマスクレベル以下で、かつタイマ割込みレベルよりも高い割込みネスト数

(1) system.vector_typeがROM_ONLY_DIRECTまたはRAM_ONLY_DIRECT、またはタイマ割込みよりも高いレベルのノーマル割込みが存在しない場合
多重割込み加算値＝8×UPPINTNST＋16×LOWINTNST

(2) (1)以外の場合
「18.7 ノーマル割込みハンドラのスタック(system.stack_size)」において、対象割込みを自割込みレベルより高いノーマル割込みに限定して算出される「割込みスタック領域サイズ」を、多重割込み加算値としてください。なお、この算出結果はあくまでもここでの多重割込み加算値です。system.stack_sizeに設定すべき値ではありません。

このようにして求めたタイマスタックのサイズを、cfg ファイルの clock.stack_size に指定してください。

18.10 カーネルスタック(system.kernel_stack_size)

カーネルスタックは、タスクコンテキストから呼び出されたサービスコール、および初期化ルーチンが使用します。

カーネルスタックは各 CPU 毎にひとつだけ存在します。そのサイズを cfg ファイルの system.kernel_stack_size に指定することで、カーネルスタック領域が生成されます。カーネルスタック領域のセクション名は、"BC_hiknlstk"です。

必要なサイズは、以下の式で算出されます。

必要なサイズ＝max([A], [B], [C], [D], [E])

[A]：KNL_A＋多重割込み加算値 1 … system.trace!=NO の場合のみ
[B]：KNL_B＋多重割込み加算値 1
[C]：KNL_C＋多重割込み加算値 1 … system.action==YES の場合のみ
[D]：KNL_D＋(初期化ルーチンの最大使用サイズ)＋多重割込み加算値 1
[E]：KNL_E＋多重割込み加算値 2

KNL_A などの値は実装に依存するため、リリースノートに掲載しています。
なお、「初期化ルーチンの最大使用サイズ」には、タイマドライバの tdr_ini_tmr()を考慮する必要があります。

多重割込み加算値 1 および多重割込み加算値 2 は、cfg ファイルの設定によって異なります。
なお、ここでは以下の記号を使用します。
UPPINTNST：カーネル割込みマスクレベル(system.system_IPL)より高い割込みネスト数
LOWINTNST：カーネル割込みマスクレベル以下の割込みネスト数

(1) system.vector_typeがROM_ONLY_DIRECTまたはRAM_ONLY_DIRECTの場合
多重割込み加算値1＝8×UPPINTNST
多重割込み加算値2＝8×UPPINTNST＋16×LOWINTNST

(2) system.vector_typeがROM_またはRAMの場合
(i) system.regbankがALLの場合
多重割込み加算値1＝8×UPPINTNST
多重割込み加算値2＝8×UPPINTNST＋16×LOWINTNST＋α
ただし、LOWINTNSTに、kernel_intspec.hでレジスタバンクを使用できないと定義した割込みが含まれる場合で、その割込みをノーマル割込みとして使用する場合は、αとして36バイトを加算してください。
(ii) system.regbankがALL以外の場合
多重割込み加算値1＝8×UPPINTNST
多重割込み加算値2＝8×UPPINTNST＋24×LOWINTNST＋20
ただし、LOWINTNSTが0の場合は、下線部を0として計算してください。

このようにして求めたカーネルスタックのサイズを、cfg ファイルの system.kernel_stack_size に指定してください。

RENESAS

18.11 HI7200/MP 提供機能による使用サイズ

ライブラリとして提供される関数が使用するスタックサイズは製品バージョンに依存するため、製品添付のリリースノートに掲載しています。

18.11.1 カーネル

リリースノートに以下の情報を掲載しています。

- 各サービスコールが使用するスタックサイズ
- タイマスタックに関する定数値(TMR_A など)、およびデバッグデーモンの使用サイズ
- カーネルスタックに関する定数値(KNL_A など)
- remote_svc.stack_size に指定すべきサイズ(SVC サーバタスクのスタックサイズ)
- IPI に登録するコールバック関数が使用するスタックサイズ
  vini_rmtではIPI_create()を用いてIPIポートを生成します。IPIのプロセッサ間割込みハンドラのスタックサイズを算出する際には、このサイズを加味してください。

18.11.2 RPC ライブラリ

(1) サーバスタブのスタックサイズ(rpc_server_info.ulStubStackSize)

rpc_start_server()および rpc_start_server_with_paramarea()では、サーバスタブのスタックサイズを rpc_server_info.ulStubStackSize に指定します。

ここに指定すべき値は、サーバスタブまたは rpc_stop_server()で指定するコールバック関数をタスクエントリ関数と扱った場合に「18.6.1 スタックサイズの算出」に従って算出される値としてください。

(2) rpc_disconnect()で指定するコールバック関数が使用するスタックサイズ

現在の実装では、このコールバック関数が実行されることはありません。

(3) リリースノートに掲載している情報

以下の情報は、リリースノートに掲載しています。

- API 関数が使用するスタックサイズ
- rpc_init()で、rpc_config.ServerTaskStackSize に指定すべきスタックサイズ
- IPI に登録するコールバック関数が使用するスタックサイズ
  RPC ライブラリは、IPI_create()を用いて IPI ポートを生成します。この時に指定するコールバック関数が使用するスタックサイズはリリースノートに掲載しています。IPI のプロセッサ間割込みハンドラのスタックサイズを算出する際には、このサイズを加味してください。

18.11.3 OAL の API 関数

アプリケーション関数と同様の方法でスタックサイズを求めてください。

18.11.4 IPI

(1) API 関数が使用するスタックサイズ

アプリケーション関数と同様の方法でスタックサイズを求めてください。

(2) IPI_create()で指定するコールバック関数が使用するスタックサイズ

このコールバック関数は、IPI のプロセッサ間割込みハンドラから呼び出されます。プロセッサ間割込みハンドラのスタックサイズを算出する際には、このことを考慮してください。

18.11.5 スピンロックライブラリの API 関数

アプリケーション関数と同様の方法でスタックサイズを求めてください。

ただし、現在の実装では、スピンロックライブラリ関数はすべてアセンブリ言語で記述されており、スタックを一切使用しないように実装されています。

18.11.6 キャッシュサポートライブラリの API 関数

アプリケーション関数と同様の方法でスタックサイズを求めてください。

なお、現在の実装では、一部の内部関数はアセンブリ言語で記述されており、その関数ではスタックを一切使用しないように実装されています。

19. types.h

カーネルや RPC ライブラリでは、types.h で定義されるデータ型を元に各種派生データ型を定義しています。

types.h は、<RTOS_INST>¥os¥include¥ディレクトリに格納されています。

表 19.1に、types.h で定義されるデータタイプを示します。

表19.1　types.h で定義されるデータ型

No.	データ型	定義内容
1	VOID	void
2	INT	signed int
3	INT8	signed char
4	INT16	signed short
5	INT32	signed long
6	INT64	signed long long
7	UINT	unsigned int
8	UINT8	unsigned char
9	UINT16	unsigned short
10	UINT32	unsigned long
11	UINT64	unsigned long long

20. FPU に関する注意

本章では、SH2A-FPU が内蔵する FPU に関する注意事項を解説します。特に、「20.1.3 fpu オプションと fpscr オプション」は浮動小数点演算を行わない場合も一読してください。

20.1 コンパイラのオプション

20.1.1 オプションの統一

コンパイラマニュアルにも記載があるように、FPU に関する以下のコンパイラオプションはリンク単位で統一しなければなりません。

- fpu オプション
- fpscr オプション
- round オプション

20.1.2 cpu オプション

必ず"cpu=sh2afpu"を指定してください。

20.1.3 fpu オプションと fpscr オプション

基本的に、fpu オプションには必ず"single"または"double"を指定するようにしてください。浮動小数点演算を行わない場合は、fpu=single を指定してください。

fpu オプションは省略することもできますが、推奨しません。fpu オプションを省略する場合は、必ず fpscr=safe オプションを指定してください。fpu オプションを省略し、かつ fpscr オプションに"safe"以外を指定した場合は、正常に動作しない場合があります。

20.2　タスク、タスク例外処理ルーチンで浮動小数点演算を行う場合

20.2.1　TA_COP1 属性

タスクおよびタスク例外処理ルーチンで浮動小数点演算を行う場合は、必ず TA_COP1 属性を指定してください。

20.2.2　FPSCR の初期化

本カーネルでは、タスクおよびタスク例外処理ルーチン起動時の FPSCR の値は、H'00040001(SZ=0, PR=0, DN=1, RM=B'01)となっています。この初期値は、コンパイラの各種オプションのデフォルトに対応しています。

浮動小数点演算を行う場合で、以下のいずれかのコンパイルオプションを指定している場合(デフォルトとは異なるオプション設定)は、タスクおよびタスク例外処理ルーチンのエントリ関数の先頭で FPSCR を初期化する必要があります。

具体的には、「20.5 参考：コンパイラの扱い」に記載の「関数開始時点でコンパイラが想定している値」に FPSCR を初期化する必要があります。

- fpu=double
- round=nearest

図 20.1に、下記の条件におけるタスクの FPSCR 初期化例を以下に示します。

[コンパイラオプション]

- cpu = sh2afpu
- fpu = double
- round = nearest

```
#include <machine.h>   /* 組み込み関数 set_fpscr()を使用するためにインクルード */
#define INI_FPSCR 0x000C0000 /* FPSCR 初期値(SZ=0, PR=1, DN=1, RM=B'00) */
#pragma noregsave(Task)
void Task(VP_INT exinf)
{
    set_fpscr(INI_FPSCR);    /* タスクの先頭で FPSCR を初期化 */
    /* タスクの処理 */
    ext_tsk();
}
```

図20.1　タスクにおける FPSCR 初期化例

20.3 各種ハンドラ等で浮動小数点演算を行う場合

本節では、以下のハンドラ等で浮動小数点演算を行う場合の注意事項を説明します。

- ノーマル割込みハンドラ
- ダイレクト割込みハンドラ
- ノーマル CPU 例外ハンドラ
- ダイレクト CPU 例外ハンドラ
- タイムイベントハンドラ
- 初期化ルーチン
- タイマドライバ(tdr_ini_tmr(), tdr_int_tmr(), tdr_stp_tmr(), tdr_rst_tmr())

20.3.1 概要

(1) FPU レジスタの保証

浮動小数点演算を行う場合は、これらのハンドラで全 FPU レジスタを明示的に保証しなければなりません。

(2) FPSCR の初期化

各ハンドラ起動時の FPSCR は、ノーマル・ダイレクト CPU 例外ハンドラは CPU 例外発生前と同じ、ノーマル・ダイレクト割込みハンドラは割込み発生前と同じ、その他のハンドラは不定です。

これらのハンドラで浮動小数点演算を行う場合は、ハンドラのエントリ関数の先頭で、「20.5 参考：コンパイラの扱い」に記載の「関数開始時点でコンパイラが想定している値」に FPSCR を初期化する必要があります。

20.3.2 コーディング方法

本製品では、これらを容易に行うための以下のマクロを提供しています。これらのマクロは、<RTOS_INST>¥os¥include¥sh2afpu.h で定義されています。これらのマクロでは#pragma inline_asm を使用しているので、コンパイル時にはオブジェクト形式指定オプション code =asmcode を用いてコンパイルする必要があります。

(1) void IniFPU (VT_FPU *pk_save, UW ini_fpscr)

ハンドラ関数の先頭で使用します。

pk_save に、FPSCR を含む現在の FPU レジスタを退避し、FPSCR を ini_fpscr に初期化します。

(2) void EndFPU(VT_FPU *pk_save)

ハンドラ関数の最後で使用します。

pk_save から FPSCR を含む FPU レジスタを復帰します。

図 20.2に、ハンドラ等における FPSCR 初期化・FPU レジスタ保証例を示します。

```
#include "sh2afpu.h"            /* ヘッダファイル"sh2afpu.h"をインクルード */
#define INI_FPSCR 0x00040001   /* FPSCR 初期値(SZ=0, PR=0, DN=1, RM=B'01) */
void HandlerMain(void)          /* ハンドラの本体処理を行う関数 */
{
    /* ハンドラの処理 */
}

void Handler(void)              /* ハンドラのエントリ関数 */
{
    VT_FPU area;                /* FPU レジスタの保存領域 */
    IniFPU(&area, INI_FPSCR);  /* FPU レジスタの保存、FPSCR の初期化 */
    HandlerMain();             /* 本体処理を行う HandlerMain()をコール */
    EndFPU(&area);             /* FPU レジスタの復帰 */
}
```

図20.2 ハンドラ等における FPSCR 初期化・FPU レジスタの保証例

20.4　拡張サービスコールルーチン浮動小数点演算を行う場合

拡張サービスコールの発行は、コンパイラからは「浮動小数点データを受け渡さない関数呼び出し」と扱われます。

20.4.1　タスクコンテキストから呼び出される場合

呼び出し元のタスクまたはタスク例外処理ルーチンには、TA_COP1 属性が指定されている必要があります。

20.4.2　非タスクコンテキストから呼び出される場合

呼び出し元の割込みハンドラなどで全 FPU レジスタを保証する必要があります。「20.3 各種ハンドラ等」を参照してください。

20.5 参考：コンパイラの扱い

本節では、コンパイラによる FPU の扱いについて解説します。

なお、コンパイラが FPSCR レジスタを変更するオブジェクトを生成するのは、fpu オプションを省略した場合のみです。

(1) FPSCR.PR(精度モード)

表20.1 コンパイラによる FPSCR.PR ビットの扱い

コンパイラオプション		コンパイラが想定している精度モード (FPSCR.PR ビット) *1	関数終了時の精度モード *2	備考
fpu オプション	fpscr オプション			
single	(指定不可)	単精度(0)	単精度(0)	コンパイラは PR ビットを変更するコード生成を行いません。
double	(指定不可)	倍精度(1)	倍精度(1)	
省略 (Mix)	safe	単精度(0)	単精度(0)	
	aggressive	単精度(0)	不定	本カーネルでは指定禁止

【注】 *1 コンパイラは、関数先頭でこの精度モードになっている前提でコード生成を行います。
*2 コンパイラは、関数終了時にこの精度モードとなるようにコード生成を行います。

(2) FPSCR.RM(丸めモード)

表20.2 コンパイラによる FPSCR.RM ビットの扱い

コンパイラオプション	コンパイラが想定している丸めモード (FPSCR.RM ビット) *1	備考
Round オプション		
Zero	0 で丸める(B'01)	コンパイラは RM ビットを変更するコード生成を行いません
Nearest	近傍に丸める(B'00)	

【注】 *1 コンパイラは、関数先頭でこの丸めモードになっている前提でコード生成を行います。

(3) FPSCR.SZ(転送サイズモード)

コンパイラは、常にSZ=0(FMOV命令の転送サイズは32ビット)を仮定しています。また、SZビットを変更するコード生成は行いません。

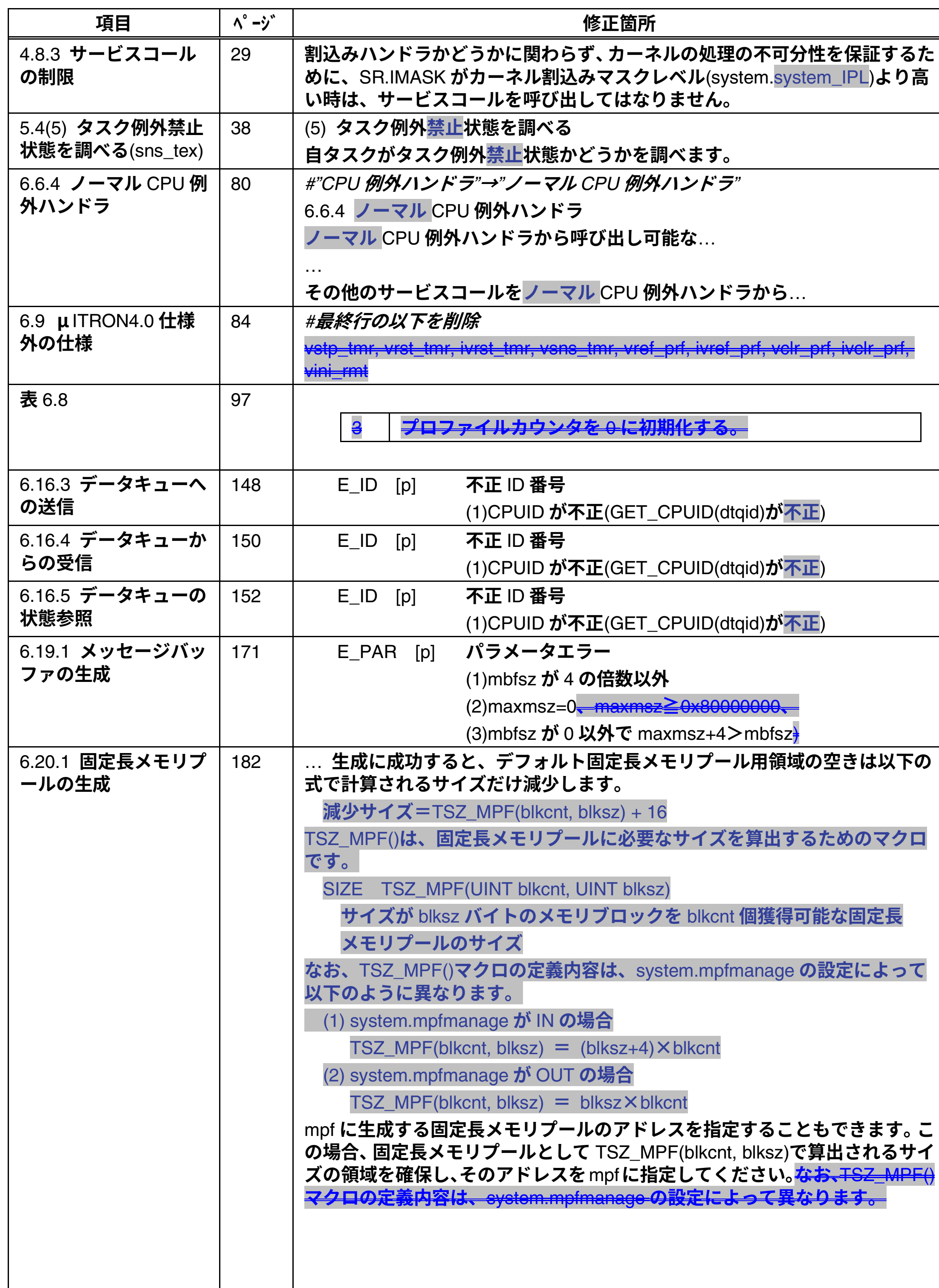
改訂内容

本章では、前版(RJJ10J1038-0100)からの主な改訂内容を記載します。改訂内容のすべてを記載したものではありません。軽微な誤記修正や表現変更は、本章にはリストアップしていません。改訂内容の詳細は、このマニュアルの本文で確認してください。

■凡例

これは例です　赤字は、修正または追加を示します。
~~これは例です~~　取り消し線を付与した青字は、削除を示します。
#これは例です　'#'で始まるイタリック体は、補足説明です。

項目	ﾍﾟｰｼﾞ	修正箇所
4.5.3 タスクの~~優先度と~~待ち行列	19	*#タイトルを修正*
4.5.4 タスクのスタック	20	*#"タスク ID"の前にローカル"を追記* 本カーネルでは、タスクが使用するスタックは、大きく「スタティックスタック」と「非スタティックスタック」に分類され、ローカルタスク ID が maxdefine.max_statictask 以下のタスクはスタティックスタック、それ以外のタスクは非スタティックスタックを使用することになっています。
4.5.4 (1)スタティックスタック 4.5.4(1) スタティックスタック	20	*#"タスク ID"の前にローカル"を追記* スタティックスタックは、cfg ファイルの static_stack[]により、複数定義することができます。static_stack[]では、スタックサイズ、スタックに付与するセクション名に加え、そのスタックを使用するローカルタスク ID を指定します。ローカルタスク ID を複数指定すると、それらのタスクでそのスタックを共有する意味になります。
4.5.4(2) 非スタティックスタック	20	*#冗長なため、以下の文を削除* ~~cfg72mp は、指定されたサイズのスタック領域を指定されたセクション名で生成し、そのアドレスを指定してカーネルにタスク生成を要求するファイルを生成します。つまり、(b)と(c)は、カーネルから見ると等価です。~~
表 4.1	20	*#"ID 番号"の前にローカルタスク"を追記* 項目 | … 　　 | … ローカルタスク ID 番号 | … … | …
4.5.5 共有スタック機能	21	*#"タスク ID"の前にローカル"を追記* スタティックスタックを複数のタスクで共有するには、cfg ファイルの static_stack[].tskid に、共有するタスク群のローカルタスク ID を羅列します。
4.8.1 割込みハンドラの種類	26	割込みハンドラには、以下の 2 種類の分類があり~~、その組み合わせで計 4 種類の割込みハンドラがあり~~ます。
4.8.1(1)(b) 割込みレベルについて	27	カーネル管理外割込みハンドラは、必ずダイレクト割込みハンドラとして実装しなければなりません。



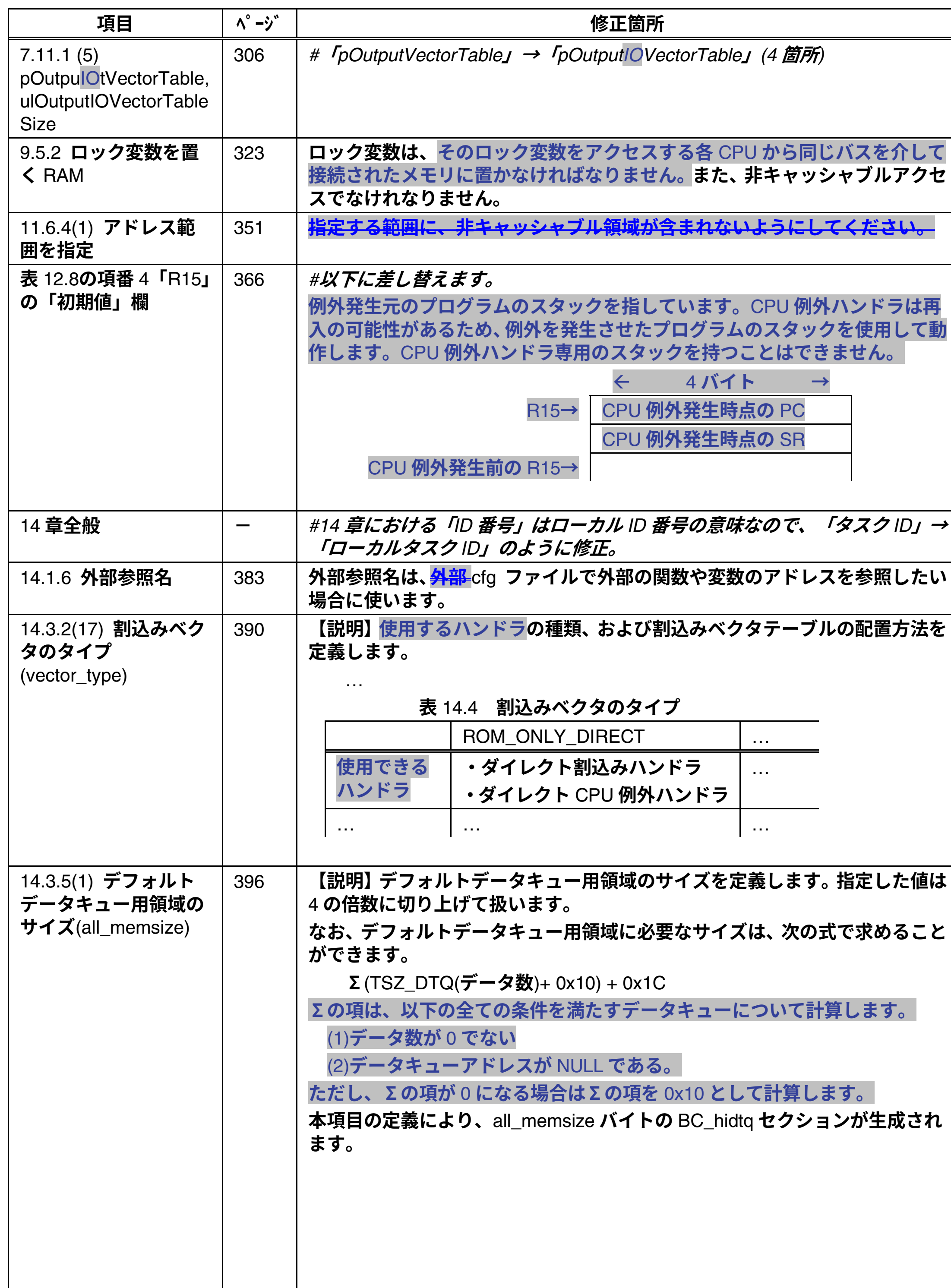
項目	ﾍﾟｰｼﾞ	修正箇所
4.8.3 サービスコールの制限	29	割込みハンドラかどうかに関わらず、カーネルの処理の不可分性を保証するために、SR.IMASK がカーネル割込みマスクレベル(system.system_IPL)より高い時は、サービスコールを呼び出してはなりません。
5.4(5) タスク例外禁止状態を調べる(sns_tex)	38	(5) タスク例外禁止状態を調べる 自タスクがタスク例外禁止状態かどうかを調べます。
6.6.4 ノーマル CPU 例外ハンドラ	80	*#"CPU 例外ハンドラ"→"ノーマル CPU 例外ハンドラ"* 6.6.4 ノーマル CPU 例外ハンドラ ノーマル CPU 例外ハンドラから呼び出し可能な… … その他のサービスコールをノーマル CPU 例外ハンドラから…
6.9 μITRON4.0 仕様外の仕様	84	*#最終行の以下を削除* ~~vstp_tmr, vrst_tmr, ivrst_tmr, vsns_tmr, vref_prf, ivref_prf, vclr_prf, ivclr_prf, vini_rmt~~
表 6.8	97	~~3~~ | ~~プロファイルカウンタを 0 に初期化する。~~
6.16.3 データキューへの送信	148	E_ID [p] 不正 ID 番号 (1)CPUID が不正(GET_CPUID(dtqid)が不正)
6.16.4 データキューからの受信	150	E_ID [p] 不正 ID 番号 (1)CPUID が不正(GET_CPUID(dtqid)が不正)
6.16.5 データキューの状態参照	152	E_ID [p] 不正 ID 番号 (1)CPUID が不正(GET_CPUID(dtqid)が不正)
6.19.1 メッセージバッファの生成	171	E_PAR [p] パラメータエラー (1)mbfsz が 4 の倍数以外 (2)maxmsz=0~~、maxmsz≧0x80000000、~~ (3)mbfsz が 0 以外で maxmsz+4＞mbfsz~~)~~
6.20.1 固定長メモリプールの生成	182	… 生成に成功すると、デフォルト固定長メモリプール用領域の空きは以下の式で計算されるサイズだけ減少します。 減少サイズ＝TSZ_MPF(blkcnt, blksz) + 16 TSZ_MPF()は、固定長メモリプールに必要なサイズを算出するためのマクロです。 SIZE TSZ_MPF(UINT blkcnt, UINT blksz) サイズが blksz バイトのメモリブロックを blkcnt 個獲得可能な固定長メモリプールのサイズ なお、TSZ_MPF()マクロの定義内容は、system.mpfmanage の設定によって以下のように異なります。 (1) system.mpfmanage が IN の場合 TSZ_MPF(blkcnt, blksz) ＝ (blksz+4)×blkcnt (2) system.mpfmanage が OUT の場合 TSZ_MPF(blkcnt, blksz) ＝ blksz×blkcnt mpf に生成する固定長メモリプールのアドレスを指定することもできます。この場合、固定長メモリプールとして TSZ_MPF(blkcnt, blksz)で算出されるサイズの領域を確保し、そのアドレスを mpf に指定してください。~~なお、TSZ_MPF()マクロの定義内容は、system.mpfmanage の設定によって異なります。~~

RENESAS

<table>
<tr><th>項目</th><th>ﾍﾟｰｼﾞ</th><th>修正箇所</th></tr>
<tr><td>6.20.2 固定長メモリプールの削除</td><td>183</td><td>…デフォルト固定長メモリプール用領域の空きは以下の式で計算されるサイズだけ増加します。
増加サイズ＝TSZ_MPF(生成時に指定した blkcnt，生成時に指定した blksz)
＋16</td></tr>
<tr><td>表 6.28およびその直後の本文</td><td>188</td><td><table><tr><th>項番</th><th>項目</th><th>内容</th></tr><tr><td>1</td><td>ローカル可変長メモリプール ID</td><td>1～_MAX_MPL(最大 1023)</td></tr><tr><td>2</td><td>管理方式</td><td>cfg ファイルで、system.newmpl に NEW を指定すると、以下が改善されます。
(1) 特に多くのメモリブロックを扱うメモリプールで、獲得/返却処理が高速化されます。
(2) 空き領域の断片化が軽減されます。
(3) さらに空き領域の断片化を軽減する VTA_UNFRAGMENT 属性を使用できます。</td></tr><tr><td>3</td><td>サポート属性</td><td>・TA_TFIFO：待ちタスクのキューイングは FIFO
・VTA_UNFRAGMENT：セクタ管理方式</td></tr></table>可変長メモリプールでは、空き領域が断片化する問題があります。~~VTA_UNFRAGMENT 属性は、これを軽減する機能です。~~「5.12.1 空き領域の断片化とその対策」も参照してください。</td></tr>
<tr><td>6.21.1(5) minblksz と sctnum192</td><td>192</td><td>…詳細は、「5.12.2 可変長メモリプールの管理方法」を参照してください。</td></tr>
<tr><td>6.26.3 CPU ロック状態への移行</td><td>228</td><td>(2) cfg ファイルで指定した system.system_IPL(カーネル割込みマスクレベル)
以下のレベルの割込みが禁止されます</td></tr>
<tr><td>6.30.1 プロファイルカウンタの参照
■引数</td><td>264</td><td>p_count　全体のプロファイルカウンタを返す記憶域へのポインタ</td></tr>
<tr><td>6.30.1 プロファイルカウンタの参照
■機能説明</td><td>264</td><td>また、p_allcount の指す領域に、全体のプロファイルカウンタ~~の総和~~を返します。
p_count および p_allcount の内容がオーバフローしたのかを知る手段はありません。~~特に p_allcount の内容については、個々にタスクのプロファイルカウンタがオーバフローしていなくても、総和を算出する際にオーバフローする可能性があります。~~</td></tr>
<tr><td>6.32 ディレクトリ・ファイル構成</td><td>275</td><td>#全面修正</td></tr>
<tr><td>7.5.6(1) コピーオーバヘッドの削減</td><td>283</td><td>ただし、SH2A-DUAL のように、キャッシュスヌープコントローラを持たないマルチコアシステムでは、クライアント CPU とサーバ CPU のキャッシュ間のコヒーレンシを保証することはできません。このため、そのポインタの指す領域を非キャッシャブル領域としなければなりません。</td></tr>
</table>

項目	ﾍﾟｰｼﾞ	修正箇所
7.10.3(1) pRpcTable, ulTableSize	289	RPC サーバを管理するためのテーブル領域を指定します。ulTableSize には、自 CPU 側に同時に生成可能なサーバ数を指定します。自 CPU 側にサーバを作らない場合は、ulTableSize に 0 を指定してください。~~この場合、pRpcTable は無視されます。~~ 次式で算出されるサイズの非キャッシャブル領域を確保し、その先頭アドレスを pRpcTable に指定してください。pRpcTable は 4 バイト境界でなければなりません。 サイズ＝sizeof(rpc_info)×ulTableSize ulTableSize が 0 の場合は、rpc_config_info 構造体内の ulIPIPortID 以外のメンバは無視されます。
7.10.3(3) ulIPIPortID	289	他 CPU へ要求した RPC からのリターン通知を受理するため、および他 CPU からの RPC 要求を受理するために使用する IPI ポート ID を指定します。
7.10.5ダイナミックサーバの開始 (rpc_start_server) ■パケットの構造	291	typedef struct { … UINT32 (**ServerStubList)(rpc_server_stub_info *); … ~~UINT32 ulParamAreaAllocation;~~ ~~void *pParamArea;~~ … } rpc_server_info; rpc_server_stub_info は、「7.11.1 サーバスタブ」を参照してください。
7.10.5ダイナミックサーバの開始 (rpc_start_server) ■機能	292	(6) ulMaxParamAreaSize サーバが動的に獲得するパラメータ領域の最大許容サイズを指定します。これを超えるパラメータ領域が必要な RPC コール要求はエラーになります。 ulMaxParamAreaSize に 0 を指定すると、許容サイズの制限なしと扱います。ただし、OAL_GetMemory()で取得可能なサイズには上限があるため、これを越えるケースでは RPC コール要求はエラーになります。
7.10.6 スタティックサーバの開始 (rpc_start_server_with _paramarea) ■パケットの構造	293	typedef struct { … UINT32 (**ServerStubList)(rpc_server_stub_info *); … } rpc_server_info; rpc_server_stub_info は、「7.11.1 サーバスタブ」を参照してください。
7.10.11 サーバ関数呼び出し(データ転送コールバック) (rpc_call_copycbk) ■C 言語 API	302	INT32 rpc_call_copycbk(rpc_call_info *pCallInfo, void (*CopyCbk1)(void *, const void *, UINT32), void (*CopyCbk2)(void *, const void *, UINT32));
7.10.12 サーバプロパティの取得 (rpc_get_server_prope rties)	304	■引数 ulServerID サーバ ID pProp サーバプロパティ情報パケットへのポインタ
7.11.1(4) pucParamArea, ulInParamSize	306	サーバスタブが式(b)に従って各パラメータ格納アドレスを正しく求められるようにするために、「常にパラメータの数は固定、最後以外のパラメータのサイズも固定」という仕様にしたがってサーバスタブ・クライアントスタブを実装する必要があります。

RENESAS



項目	ﾍﾟｰｼﾞ	修正箇所
7.11.1 (5) pOutpuIOtVectorTable, ulOutputIOVectorTable Size	306	*#「pOutputVectorTable」→「pOutputIOVectorTable」(4 箇所)*
9.5.2 ロック変数を置く RAM	323	ロック変数は、そのロック変数をアクセスする各 CPU から同じバスを介して接続されたメモリに置かなければなりません。また、非キャッシャブルアクセスでなければなりません。
11.6.4(1) アドレス範囲を指定	351	~~指定する範囲に、非キャッシャブル領域が含まれないようにしてください。~~
表 12.8の項番 4「R15」の「初期値」欄	366	*#以下に差し替えます。* 例外発生元のプログラムのスタックを指しています。CPU 例外ハンドラは再入の可能性があるため、例外を発生させたプログラムのスタックを使用して動作します。CPU 例外ハンドラ専用のスタックを持つことはできません。 ← 4 バイト → R15→ | CPU 例外発生時点の PC | CPU 例外発生時点の SR CPU 例外発生前の R15→ |
14 章全般	－	*#14 章における「ID 番号」はローカル ID 番号の意味なので、「タスク ID」→「ローカルタスク ID」のように修正。*
14.1.6 外部参照名	383	外部参照名は、~~外部~~cfg ファイルで外部の関数や変数のアドレスを参照したい場合に使います。
14.3.2(17) 割込みベクタのタイプ (vector_type)	390	【説明】使用するハンドラの種類、および割込みベクタテーブルの配置方法を定義します。 … 表 14.4 割込みベクタのタイプ | | ROM_ONLY_DIRECT | … | | 使用できるハンドラ | ・ダイレクト割込みハンドラ ・ダイレクト CPU 例外ハンドラ | … | | … | … | … |
14.3.5(1) デフォルトデータキュー用領域のサイズ(all_memsize)	396	【説明】デフォルトデータキュー用領域のサイズを定義します。指定した値は 4 の倍数に切り上げて扱います。 なお、デフォルトデータキュー用領域に必要なサイズは、次の式で求めることができます。 Σ(TSZ_DTQ(データ数)+ 0x10) + 0x1C Σの項は、以下の全ての条件を満たすデータキューについて計算します。 (1)データ数が 0 でない (2)データキューアドレスが NULL である。 ただし、Σの項が 0 になる場合はΣの項を 0x10 として計算します。 本項目の定義により、all_memsize バイトの BC_hidtq セクションが生成されます。

RENESAS

項目	ﾍﾟｰｼﾞ	修正箇所
14.3.6(1) **デフォルトメッセージバッファ用領域のサイズ** (all_memsize)	397	**【説明】デフォルトメッセージバッファ用領域のサイズを定義します。指定した値は** 4 **の倍数に切り上げて扱います。** **なお、デフォルトメッセージバッファ用領域に必要なサイズは、次の式で求めることができます。** Σ(**メッセージバッファのサイズ** + 0x10) + 0x1C **Σの項は、以下の全ての条件を満たすメッセージバッファについて計算します。** (1)**メッセージバッファのサイズが** 0 **でない** (2)**メッセージバッファのアドレスが** NULL **である。** **ただし、Σの項が** 0 **になる場合はΣの項を** 0x10 **として計算します。** **本項目の定義により、**all_memsize **バイトの** BC_himbf **セクションが生成されます。**
14.3.7(1) **デフォルト固定長メモリプール用領域のサイズ** (all_memsize)	398	**【説明】デフォルト固定長メモリプール用領域のサイズを定義します。指定した値は** 4 **の倍数に切り上げて扱います。** **なお、デフォルト固定長メモリプール用領域に必要なサイズは、次の式で求めることができます。** Σ(TSZ_MPF(**ブロック数，ブロックサイズ**) + 0x10) + 0x1C **Σの項は、アドレスが** NULL **でない固定長メモリプールについて計算します。** **ただし、Σの項が** 0 **になる場合はΣの項を** 0x10 **として計算します。** **本項目の定義により、**all_memsize **バイトの** BC_himpf **セクションが生成されます。**
14.3.8(1) **デフォルト可変長メモリプール用領域のサイズ** (all_memsize)	399	**【説明】デフォルト可変長メモリプール用領域のサイズを定義します。指定した値は** 4 **の倍数に切り上げて扱います。** **なお、デフォルト可変長メモリプール用領域に必要なサイズは、次の式で求めることができます。** Σ(**メモリプールのサイズ** + 0x10) + 0x1C **Σの項は、アドレスが** NULL **でない可変長メモリプールについて計算します。** **ただし、Σの項が** 0 **になる場合はΣの項を** 0x10 **として計算します。** **本項目の定義により、**all_memsize **バイトの** BC_himpf **セクションが生成されます。**

14.3.9 **システムクロック定義**(clock)	400	`clock {`
		`timer = <設定値>; // (1)タイマモード`
		`IPL = <設定値>; // (2)タイマ割込みレベル`
		`number = <設定値>; // (3)タイマ割込みのベクタ番号`
		`stack_size = <設定値>; // (4)タイマスタックサイズ`
		`};`

14.3.9(2) **タイマ割込みレベル**(IPL)	401	**【省略時の扱い】デフォルト** cfg **ファイルの設定値**(**出荷時は** 15)**を適用** (Warning **あり**) (system.system_IPL **が** 15 **未満の場合は、結果的にエラーになります**)

RENESAS

項目	ﾍﾟｰｼﾞ	修正箇所
14.3.10(5) SVC サーバタスク空き待ちタスクの最大数(num_wait)	403	【説明】リモートサービスコールを呼び出した時、対象 CPU 側の SVC サーバタスクの空きがない場合は待ち状態になります。num_wait には、本カーネル側で同時に他 CPU 側の SVC サーバタスクの空きを待つタスクの最大数を定義します。 リモートサービスコール発行時点で以下の全ての条件が満たされる場合、そのリモートサービスコールには、直ちに EV_NORESOURCE が返ります。 (1) 対象 CPU 側に、SVC サーバタスクの空きがない。 (2) 呼び出し元のカーネル側で、既に num_wait の数だけ SVC サーバタスクの空きを待っているタスクが存在する。 常に SVC サーバタスクの空きを待たないように…
14.3.10(5) SVC サーバタスク空き待ちタスクの最大数(num_wait)	403	【省略時の扱い】~~デフォルト cfg ファイルの設定値を適用(Warning あり)。出荷時のデフォルト cfg~~ ~~【省略時の扱い】~~デフォルト cfg ファイルの設定値(出荷時は 0)を適用(Warning あり)
14.3.21(4) 起動周期(interval_counter)	429	*#タイトルを修正*
14.3.21(7) 起動位相(phs_counter)	429	*#タイトルを修正*
14.3.25(3) ダイレクト属性(direct)	434	【説明】VTA_DIRECT 属性を付与するかどうかを定義します。 system.system_IPL より高いレベルの割込みハンドラ(NMI を含む)には、必ず VTA_DIRECT 属性を付与しなければなりません。
14.5.2 306メッセージ一覧	443	`4301 (E) The task ID=`*ID 番号*` does not use static stack` static_stack[].tskid に、maxdefine.max_statictask より大きいローカルタスク ID 番号が指定されています。スタティックスタックを使用するタスクをローカルタスク ID 番号で指定する場合は、maxdefine.max_statictask 以下でなければなりません。
	444	`4407 (E) Too big IPL --> <`*clock.IPL 設定値*`>` clock.IPL が、system.system_IPL を超えています。
図 16.3, 図 16.4	456, 457	*#図中の「_kernel_ini_tmr 関数」→「tdr_ini_tmr 関数」*
16.3.4(1) io_set_cpg_cpuid1() (cpg1.c)	468	なお、FRQCR1 レジスタによる I1φの分周率の設定は、CPUID#2 側の io_set_cpg_cpuid2() (cpg2.c)で行います。これは、FRQCR1 の変更は CPUID#2 からのみ可能な LSI 仕様であるためです。
表 16.12	499	(下表参照)

項目	解説
HEAPSIZE	sbrk()が管理するヒープ領域のサイズ
PRI_SBRK *	sbrk()の排他制御用に使用するミューテックスの上限優先度
PRI_S1PTR *	_s1ptr の排他制御用に使用するミューテックスの上限優先度
PRI_IOB *	iob の排他制御用に使用するミューテックスの上限優先度
SEM_TMOUT *	ミューテックスのロック取得時のタイムアウト
NUM_TASK *	最大ローカルタスク ID

<table>
<tr><th>項目</th><th>ﾍﾟｰｼﾞ</th><th>修正箇所</th></tr>
<tr><td>表 16.17</td><td>512</td><td>
<table>
<tr><th>ｵﾌﾞｼﾞｪｸﾄ種別</th><th>分類</th><th>生成方法</th><th>ID 名称</th></tr>
<tr><td>…</td><td>…</td><td>…</td><td>…</td></tr>
<tr><td>ﾒｰﾙﾎﾞｯｸｽ</td><td>ﾀﾞﾐｰ</td><td>cfg ﾌｧｲﾙの mailbox[]</td><td>ID1_MBX_DUMMY</td></tr>
<tr><td>…</td><td>…</td><td>…</td><td>…</td></tr>
<tr><td>固定長ﾒﾓﾘﾌﾟｰﾙ</td><td>ﾀﾞﾐｰ</td><td>cfg ﾌｧｲﾙの memorypool[]</td><td>ID1_MPF_DUMMY</td></tr>
<tr><td>…</td><td>…</td><td>…</td><td>…</td></tr>
</table>
</td></tr>
<tr><td>16.11.1(9) remote_svc 定義</td><td>515</td><td>num_wait = 20;</td></tr>
<tr><td>16.11.1(12) eventflag[] 定義</td><td>517</td><td>name = ID1_FLG_DUMMY;</td></tr>
<tr><td>16.11.1(17) memorypool[]定義</td><td>518</td><td>// Dummy fixed-size memory pool ***********
memorypool[ ] { // the ID is assigned by configurator.
name = ID1_MPF_DUMMY;
// export = <YES or NO>;
section = C_MPF; // the section name is "BC_MPF".
// address = <input start address of pool area>;
num_block = 32;
siz_block = 16;
wait_queue = TA_TFIFO;
};</td></tr>
<tr><td>16.11.1(18) variable_memorypool[] 定義</td><td>518</td><td>heap_size = 0x400;
mpl_section = C_MPL;</td></tr>
<tr><td>16.11.1(21) overrun_hand[]定義</td><td>519</td><td>entry_address = DummyOverrunHandler();</td></tr>
<tr><td>16.11.1(23) interrupt_vector[]定義</td><td>521</td><td>#行番号 60 の解説欄を修正
レジスタバンクを使用(system.regnbank が ALL のため、意味を持ちません)</td></tr>
<tr><td>16.11.2(9) remote_svc 定義</td><td>523</td><td>num_wait = 20;</td></tr>
<tr><td>16.11.2(10) task[]定義</td><td>524</td><td>// TaskRecv() ****************
task[] { // the ID is assigned by configurator.
name = ID2_TASK_RECV;
// export = <YES or NO>;
entry_address = TaskRecv();
…</td></tr>
<tr><td>16.11.2(18) variable_memorypool[] 定義</td><td>527</td><td>heap_size = 0x400;
mpl_section = C_MPL;</td></tr>
<tr><td>16.11.2(21) overrun_hand[]定義</td><td>527</td><td>entry_address = DummyOverrunHandler();</td></tr>
<tr><td>16.11.2(23) interrupt_vector[]定義</td><td>529</td><td>#行番号 60 の解説欄を修正
レジスタバンクを使用(system.regnbank が ALL のため、意味を持ちません)</td></tr>
<tr><td>16.11.2(9) remote_svc 定義</td><td>523</td><td>num_wait = 20;</td></tr>
</table>

<table>
<tr><th>項目</th><th>ﾍﾟｰｼﾞ</th><th>修正箇所</th></tr>
<tr><td>図 17.7</td><td>538</td><td>#環境変数ダイアログボックスの[値]ボックス内の表示を以下に修正
$(RTOS_INST)¥cfg72mp</td></tr>
<tr><td>図 17.8</td><td>539</td><td>#それぞれのダイアログボックスの[シンタックス]ボックス内の表示を、
#以下に修正
(1) [タイプ]が"Error"の[シンタックス]ボックス
$(FULLFILE)($(LINE)) : *(E)*
(2) [タイプ]が"Warning"の[シンタックス]ボックス
$(FULLFILE)($(LINE)) : *(W)*</td></tr>
<tr><td>17.2.3(2) ファイルのビルド順序の設定</td><td>541</td><td>[ファイルのビルド順序]タブにおいて、[ファイルグループ]から"Kernel config file"を選択し、右側の[フェーズの順序]の[cfg72mp]をチェックしてください。</td></tr>
<tr><td>表 17.3</td><td>546</td><td>
<table>
<tr><th>No.</th><th>セクション</th><th>備考</th></tr>
<tr><td>…</td><td>…</td><td>…</td></tr>
<tr><td>5</td><td>CC_hiinttbl</td><td>カーネル内部データ~~ *2~~</td></tr>
<tr><td>6</td><td>BC_hivct</td><td>カーネル内部データ *2</td></tr>
<tr><td>7</td><td>BC_hiinttbl</td><td>カーネル内部データ~~ *2~~</td></tr>
<tr><td>…</td><td>…</td><td>…</td></tr>
</table>
… …
*2 vsta_knl では、system.vector_type に応じて VBR レジスタを以下のように初期化します。
(1) system.vector が ROM または ROM_ONLY_DIRECT の場合
CC_hivct セクション先頭アドレス-16
(2) system.vector が RAM または RAM_ONLY_DIRECT の場合
BC_hivct セクション先頭アドレス-16
~~(3) system.vector が ROM の場合：CC_hiinttbl セクション先頭アドレス-16~~
~~(4) system.vector が RAM の場合：BC_hiinttbl セクション先頭アドレス-16~~</td></tr>
<tr><td>17.8.2(1) インクルードディレクトリ</td><td>561</td><td>#ディレクトリ区切り記号'¥'を追記
remote_send.c については、CPUID#2 側の kernel_id_cpu2.h をインクルードしているため、図 17.21のようにその格納パスを追加しています。なお、「$(WORKSPDIR)¥cfg_out」(cpuid1¥cfg_out¥)と「$(WORKSPDIR)¥..¥cpuid2¥cfg_out」(cpuid2¥cfg_out¥)には、それぞれ CPUID#1 用、CPUID#2 用の kernel_id.h などの同名ファイルが格納されているため、必ず CPUID#1 用の「$(WORKSPDIR)¥cfg_out」が優先(画面上で上方)となるようにしなければなりません。</td></tr>
<tr><td>17.10 ターゲットへのダウンロード</td><td>578</td><td>#「7. 両方の High-performance Embedded Workshop からリセットコマンドを入力します。」を No.5 とし、変更前の No.5,6 をそれぞれ No.6,7 に変更します。</td></tr>
<tr><td>18.8 ダイレクト割込みハンドラのスタック</td><td>588</td><td>なお、カーネル割込みマスクレベル(system.system_IPL)より高い割込みレベルのハンドラは、ダイレクト割込みハンドラとして実装しなければなりません。</td></tr>
<tr><td>18.11.1 カーネル</td><td>591</td><td>• IPI に登録するコールバック関数が使用するスタックサイズ
~~remote_svc.num_server>0 の場合、~~vini_rmt では IPI_create()を用いて IPI ポートを生成します。IPI のプロセッサ間割込みハンドラのスタックサイズを算出する際には、このサイズを加味してください。</td></tr>
</table>

項目	ﾍﾟｰｼﾞ	修正箇所
18.11.2(3) リリースノートに掲載している情報	591	以下の情報は、リリースノートに掲載しています。 • API 関数が使用するスタックサイズ • rpc_init()で、rpc_config.ServerTaskStackSize に指定すべきスタックサイズ • IPI に登録するコールバック関数が使用するスタックサイズ RPC ライブラリは、IPI_create()を用いて IPI ポートを生成します。IPI のプロセッサ間割込みハンドラのスタックサイズを算出する際には、このサイズを加味してください。
図 20.1	596	#define INI_FPSCR 0x000C0000 /* FPSCR 初期値(SZ=0, PR=1, DN=1, RM=B'00) */
図 20.2	598	/* FPSCR 初期値(SZ=0, PR=0, DN=1, RM=B'01) */

RENESAS

ルネサスマイクロコンピュータ開発環境システム
ユーザーズマニュアル
HI7200/MP V1.00

発行年月日　2007 年 9 月 27 日　Rev.1.01
発　　　行　株式会社ルネサス テクノロジ 営業統括部
　　　　　　〒100-0004 東京都千代田区大手町 2-6-2
編　　　集　株式会社ルネサスソリューションズ
　　　　　　グローバルストラテジックコミュニケーション本部
　　　　　　カスタマサポート部

© 2007. Renesas Technology Corp., All rights reserved. Printed in Japan.

株式会社ルネサス テクノロジ　営業統括部　〒100-0004 東京都千代田区大手町2-6-2 日本ビル

営業お問合せ窓口
株式会社ルネサス販売

http://www.renesas.com

本社	〒100-0004	千代田区大手町2-6-2 (日本ビル)	(03) 5201-5350
京浜支社	〒212-0058	川崎市幸区鹿島田890-12 (新川崎三井ビル)	(044) 549-1662
西東京支社	〒190-0023	立川市柴崎町2-2-23 (第二高島ビル2F)	(042) 524-8701
東北支社	〒980-0013	仙台市青葉区花京院1-1-20 (花京院スクエア13F)	(022) 221-1351
いわき支店	〒970-8026	いわき市平小太郎町4-9 (平小太郎ビル)	(0246) 22-3222
茨城支店	〒312-0034	ひたちなか市堀口832-2 (日立システムプラザ勝田1F)	(029) 271-9411
新潟支店	〒950-0087	新潟市東大通1-4-2 (新潟三井物産ビル3F)	(025) 241-4361
松本支社	〒390-0815	松本市深志1-2-11 (昭和ビル7F)	(0263) 33-6622
中部支社	〒460-0008	名古屋市中区栄4-2-29 (名古屋広小路プレイス)	(052) 249-3330
関西支社	〒541-0044	大阪市中央区伏見町4-1-1 (明治安田生命大阪御堂筋ビル)	(06) 6233-9500
北陸支社	〒920-0031	金沢市広岡3-1-1 (金沢パークビル8F)	(076) 233-5980
広島支店	〒730-0036	広島市中区袋町5-25 (広島袋町ビルディング8F)	(082) 244-2570
鳥取支店	〒680-0822	鳥取市今町2-251 (日本生命鳥取駅前ビル)	(0857) 21-1915
九州支社	〒812-0011	福岡市博多区博多駅前2-17-1 (博多プレステージ5F)	(092) 481-7695

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：コンタクトセンタ　E-Mail: csc@renesas.com

HI7200/MP　V.1.00
ユーザーズマニュアル

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

RJJ10J1938-0101