T-Kernel 2.0 仕様書

2014 年 12 月

T-Engineフォーラム

http://www.t-engine.org/

Copyright © 2011-2014 T-Engine Forum

T-Kernel 2.0仕様書 (Ver.2.01.00)

本仕様書の著作権は、T-Engineフォーラムに属しています。
本仕様書の内容の転記、一部複製等には、T-Engineフォーラムの許諾が必要です。
本仕様書に記載されている内容は、今後改良等の理由でお断りなしに変更することがあります。
本仕様書に関しては、下記にお問い合わせください。

T-Engineフォーラム事務局
〒141-0031
東京都品川区西五反田2-20-1 第28興和ビル
YRPユビキタス・ネットワーキング研究所内
TEL: 03-5437-0572
FAX: 03-5437-2399
E-mail: office@t-engine.org

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
2.00.00	2011-04-01	初版	T-Engine Forum
2.00.01	2012-11-21	• 「アクセス権情報」または「アクセス権」の語を「呼出元アクセス権情報」と修正し、誤解が生じないように意味を明確化した。 • SetTaskSpace の仕様を正確となるように修正し、曖昧な記述を明確化した。	T-Engine Forum
2.00.02	2013-02-01	• 存在しない「タスクに対する起動要求をキャンセルする機能」に関する記述を削除した。 • tk_rel_waiに関する記述の一部を脚注から本文に移動し、記述が仕様として定義された動作であることを明確化した。	T-Engine Forum
2.00.03	2013-12-18	• Typoの修正: ac[0] -> av[0]	T-Engine Forum
2.01.00	2014-11-20	• T-Kernel/OS, T-Kernel/SM, T-Kernel/DSの表記を整理 • 実装依存となるエラーコードに関する説明を整理 • BOOL型を符号無し整数に変更 • 相対時間とシステム時刻に関する補足事項を追加 • tk_dly_tskのエラー一覧からE_NOMEMを削除 • tk_clr_flgの解説から不要な説明を削除 • tk_fwd_por, tk_rpl_rdvの引数msgにCONST修飾子を追加 • tk_rot_rdqの注釈の表示位置修正 • ChkSpaceR, ChkSpaceRW, ChkSpaceREの引数addrにCONST修飾子を追加 • ChkSpaceBstrR, ChkSpaceBstrRW, ChkSpaceTstrR, ChkSpaceTstrRのリターンパラメータのエラーコードに関する説明を追加	T-Engine Forum

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
2.01.00	2014-11-20	• パラメータの英文を修正: – wtsk: Wait Task Information -> Waiting Task ID – stsk: Send Task Information -> Send Waiting Task ID – atsk: Accept Task Information -> Accept Waiting Task ID – nblk: Number of Block -> Number of Blocks – nmemb: Number of Memory Block -> Number of Memory Blocks – nreq: Number of Request -> Number of Requests • Typoの修正: – モードを設定する設定 -> モードを設定する機能 – MAKER -> maker – attrに書込みができない -> addrに書込みができない – blockcount -> blockcont – pk_calm, almatr, almhdrが不正 -> pk_calm, almhdrが不正 – これらの待ちは -> この待ちは	T-Engine Forum

目次

APIの記述形式 1

T-Kernel/OS システムコール索引 3

T-Kernel/SM 拡張SVC・ライブラリ索引 7

T-Kernel/DS システムコール索引 10

1 T-Kernelの概要 12
- 1.1 全体的な位置付け 13
- 1.2 適応化 15
- 1.3 T-Kernel 2.0の概要 16
  - 1.3.1 T-Kernel 2.0の位置付けと基本方針 16
  - 1.3.2 T-Kernel 2.0の追加機能 16

2 T-Kernelの概念 18
- 2.1 基本的な用語の意味 19
- 2.2 タスク状態とスケジューリング規則 21
  - 2.2.1 タスク状態 21
  - 2.2.2 タスクのスケジューリング規則 23
- 2.3 割込み処理 26
- 2.4 タスク例外処理 27
- 2.5 システム状態 28
  - 2.5.1 非タスク部実行中のシステム状態 28
  - 2.5.2 タスク独立部と準タスク部 29
- 2.6 オブジェクト 31
- 2.7 メモリ 32
  - 2.7.1 アドレス空間 32
  - 2.7.2 非常駐メモリ 32
  - 2.7.3 保護レベル 33

3 T-Kernel共通規定 34
3.1 データ型 . . . 35
3.1.1 汎用的なデータ型 . . . 35
3.1.2 意味が定義されているデータ型 . . . 36
3.2 システムコール . . . 38
3.2.1 システムコールの形式 . . . 38
3.2.2 タスク独立部から発行できるシステムコール . . . 39
3.2.3 システムコールの呼出制限 . . . 39
3.2.4 パラメータパケット形式の変更 . . . 39
3.2.5 機能コード . . . 40
3.2.6 エラーコード . . . 40
3.2.7 タイムアウト . . . 40
3.2.8 相対時間とシステム時刻 . . . 41
3.2.9 タイマ割込み間隔 . . . 42
3.3 高級言語対応ルーチン . . . 43

4 T-Kernel/OSの機能 44
4.1 タスク管理機能 . . . 45
4.1.1 tk_cre_tsk - タスク生成 . . . 46
4.1.2 tk_del_tsk - タスク削除 . . . 50
4.1.3 tk_sta_tsk - タスク起動 . . . 51
4.1.4 tk_ext_tsk - 自タスク終了 . . . 52
4.1.5 tk_exd_tsk - 自タスクの終了と削除 . . . 53
4.1.6 tk_ter_tsk - 他タスク強制終了 . . . 54
4.1.7 tk_chg_pri - タスク優先度変更 . . . 56
4.1.8 tk_chg_slt - タスクスライスタイム変更 . . . 58
4.1.9 tk_chg_slt_u - タスクスライスタイム変更(マイクロ秒単位) . . . 60
4.1.10 tk_get_tsp - タスク固有空間の参照 . . . 61
4.1.11 tk_set_tsp - タスク固有空間の設定 . . . 62
4.1.12 tk_get_rid - タスクの所属リソースグループの参照 . . . 63
4.1.13 tk_set_rid - タスクの所属リソースグループの設定 . . . 64
4.1.14 tk_get_reg - タスクレジスタの取得 . . . 65
4.1.15 tk_set_reg - タスクレジスタの設定 . . . 66
4.1.16 tk_get_cpr - コプロセッサのレジスタの取得 . . . 67
4.1.17 tk_set_cpr - コプロセッサのレジスタの設定 . . . 69
4.1.18 tk_inf_tsk - タスク統計情報参照 . . . 71
4.1.19 tk_inf_tsk_u - タスク統計情報参照(マイクロ秒単位) . . . 73
4.1.20 tk_ref_tsk - タスク状態参照 . . . 74
4.1.21 tk_ref_tsk_u - タスク状態参照(マイクロ秒単位) . . . 77

4.2 タスク付属同期機能 79
4.2.1 tk_slp_tsk - 自タスクを起床待ち状態へ移行 80
4.2.2 tk_slp_tsk_u - 自タスクを起床待ち状態へ移行(マイクロ秒単位) 82
4.2.3 tk_wup_tsk - 他タスクの起床 83
4.2.4 tk_can_wup - タスクの起床要求を無効化 84
4.2.5 tk_rel_wai - 他タスクの待ち状態解除 85
4.2.6 tk_sus_tsk - 他タスクを強制待ち状態へ移行 87
4.2.7 tk_rsm_tsk - 強制待ち状態のタスクを再開 89
4.2.8 tk_frsm_tsk - 強制待ち状態のタスクを強制再開 91
4.2.9 tk_dly_tsk - タスク遅延 93
4.2.10 tk_dly_tsk_u - タスク遅延(マイクロ秒単位) 94
4.2.11 tk_sig_tev - タスクイベントの送信 95
4.2.12 tk_wai_tev - タスクイベント待ち 96
4.2.13 tk_wai_tev_u - タスクイベント待ち(マイクロ秒単位) 97
4.2.14 tk_dis_wai - タスク待ち状態の禁止 98
4.2.15 tk_ena_wai - タスク待ち禁止の解除 100
4.3 タスク例外処理機能 101
4.3.1 tk_def_tex - タスク例外ハンドラの定義 102
4.3.2 tk_ena_tex - タスク例外の許可 104
4.3.3 tk_dis_tex - タスク例外の禁止 105
4.3.4 tk_ras_tex - タスク例外を発生 106
4.3.5 tk_end_tex - タスク例外ハンドラの終了 108
4.3.6 tk_ref_tex - タスク例外の状態参照 110
4.4 同期･通信機能 111
4.4.1 セマフォ 112
4.4.1.1 tk_cre_sem - セマフォ生成 113
4.4.1.2 tk_del_sem - セマフォ削除 115
4.4.1.3 tk_sig_sem - セマフォ資源返却 116
4.4.1.4 tk_wai_sem - セマフォ資源獲得 117
4.4.1.5 tk_wai_sem_u - セマフォ資源獲得(マイクロ秒単位) 118
4.4.1.6 tk_ref_sem - セマフォ状態参照 119
4.4.2 イベントフラグ 120
4.4.2.1 tk_cre_flg - イベントフラグ生成 121
4.4.2.2 tk_del_flg - イベントフラグ削除 123
4.4.2.3 tk_set_flg - イベントフラグのセット 124
4.4.2.4 tk_clr_flg - イベントフラグのクリア 125
4.4.2.5 tk_wai_flg - イベントフラグ待ち 126
4.4.2.6 tk_wai_flg_u - イベントフラグ待ち(マイクロ秒単位) 129
4.4.2.7 tk_ref_flg - イベントフラグ状態参照 130

4.4.3 メールボックス 131
4.4.3.1 tk_cre_mbx - メールボックス生成 133
4.4.3.2 tk_del_mbx - メールボックス削除 135
4.4.3.3 tk_snd_mbx - メールボックスへ送信 136
4.4.3.4 tk_rcv_mbx - メールボックスから受信 138
4.4.3.5 tk_rcv_mbx_u - メールボックスから受信(マイクロ秒単位) 140
4.4.3.6 tk_ref_mbx - メールボックス状態参照 141
4.5 拡張同期・通信機能 142
4.5.1 ミューテックス 143
4.5.1.1 tk_cre_mtx - ミューテックス生成 145
4.5.1.2 tk_del_mtx - ミューテックス削除 147
4.5.1.3 tk_loc_mtx - ミューテックスのロック 148
4.5.1.4 tk_loc_mtx_u - ミューテックスのロック(マイクロ秒単位) 150
4.5.1.5 tk_unl_mtx - ミューテックスのアンロック 151
4.5.1.6 tk_ref_mtx - ミューテックス状態参照 152
4.5.2 メッセージバッファ 153
4.5.2.1 tk_cre_mbf - メッセージバッファ生成 155
4.5.2.2 tk_del_mbf - メッセージバッファ削除 158
4.5.2.3 tk_snd_mbf - メッセージバッファへ送信 159
4.5.2.4 tk_snd_mbf_u - メッセージバッファへ送信(マイクロ秒単位) 161
4.5.2.5 tk_rcv_mbf - メッセージバッファから受信 162
4.5.2.6 tk_rcv_mbf_u - メッセージバッファから受信(マイクロ秒単位) 163
4.5.2.7 tk_ref_mbf - メッセージバッファ状態参照 164
4.5.3 ランデブ 166
4.5.3.1 tk_cre_por - ランデブポート生成 169
4.5.3.2 tk_del_por - ランデブポート削除 171
4.5.3.3 tk_cal_por - ランデブポートに対するランデブの呼出 172
4.5.3.4 tk_cal_por_u - ランデブポートに対するランデブの呼出(マイクロ秒単位) 174
4.5.3.5 tk_acp_por - ランデブポートに対するランデブ受付 175
4.5.3.6 tk_acp_por_u - ランデブポートに対するランデブ受付(マイクロ秒単位) 178
4.5.3.7 tk_fwd_por - ランデブポートに対するランデブ回送 179
4.5.3.8 tk_rpl_rdv - ランデブ返答 183
4.5.3.9 tk_ref_por - ランデブポート状態参照 185
4.6 メモリプール管理機能 187
4.6.1 固定長メモリプール 188
4.6.1.1 tk_cre_mpf - 固定長メモリプール生成 189
4.6.1.2 tk_del_mpf - 固定長メモリプール削除 191
4.6.1.3 tk_get_mpf - 固定長メモリブロック獲得 192
4.6.1.4 tk_get_mpf_u - 固定長メモリブロック獲得(マイクロ秒単位) 194

4.6.1.5 tk_rel_mpf - 固定長メモリブロック返却 . . . 195
4.6.1.6 tk_ref_mpf - 固定長メモリプール状態参照 . . . 196
4.6.2 可変長メモリプール . . . 197
4.6.2.1 tk_cre_mpl - 可変長メモリプール生成 . . . 198
4.6.2.2 tk_del_mpl - 可変長メモリプール削除 . . . 200
4.6.2.3 tk_get_mpl - 可変長メモリブロック獲得 . . . 201
4.6.2.4 tk_get_mpl_u - 可変長メモリブロック獲得(マイクロ秒単位) . . . 203
4.6.2.5 tk_rel_mpl - 可変長メモリブロック返却 . . . 204
4.6.2.6 tk_ref_mpl - 可変長メモリプール状態参照 . . . 205
4.7 時間管理機能 . . . 206
4.7.1 システム時刻管理 . . . 207
4.7.1.1 tk_set_tim - システム時刻設定 . . . 208
4.7.1.2 tk_set_tim_u - システム時刻設定(マイクロ秒単位) . . . 209
4.7.1.3 tk_get_tim - システム時刻参照 . . . 210
4.7.1.4 tk_get_tim_u - システム時刻参照(マイクロ秒単位) . . . 211
4.7.1.5 tk_get_otm - システム稼働時間参照 . . . 212
4.7.1.6 tk_get_otm_u - システム稼働時間参照(マイクロ秒単位) . . . 213
4.7.2 周期ハンドラ . . . 214
4.7.2.1 tk_cre_cyc - 周期ハンドラの生成 . . . 215
4.7.2.2 tk_cre_cyc_u - 周期ハンドラの生成(マイクロ秒単位) . . . 218
4.7.2.3 tk_del_cyc - 周期ハンドラの削除 . . . 219
4.7.2.4 tk_sta_cyc - 周期ハンドラの動作開始 . . . 220
4.7.2.5 tk_stp_cyc - 周期ハンドラの動作停止 . . . 221
4.7.2.6 tk_ref_cyc - 周期ハンドラ状態参照 . . . 222
4.7.2.7 tk_ref_cyc_u - 周期ハンドラ状態参照(マイクロ秒単位) . . . 224
4.7.3 アラームハンドラ . . . 225
4.7.3.1 tk_cre_alm - アラームハンドラの生成 . . . 226
4.7.3.2 tk_del_alm - アラームハンドラの削除 . . . 228
4.7.3.3 tk_sta_alm - アラームハンドラの動作開始 . . . 229
4.7.3.4 tk_sta_alm_u - アラームハンドラの動作開始(マイクロ秒単位) . . . 230
4.7.3.5 tk_stp_alm - アラームハンドラの動作停止 . . . 231
4.7.3.6 tk_ref_alm - アラームハンドラ状態参照 . . . 232
4.7.3.7 tk_ref_alm_u - アラームハンドラ状態参照(マイクロ秒単位) . . . 234
4.8 割込み管理機能 . . . 235
4.8.1 tk_def_int - 割込みハンドラ定義 . . . 236
4.8.2 tk_ret_int - 割込みハンドラから復帰 . . . 239
4.9 システム状態管理機能 . . . 241
4.9.1 tk_rot_rdq - タスクの優先順位の回転 . . . 242
4.9.2 tk_get_tid - 実行状態タスクのタスクID参照 . . . 244

4.9.3 tk_dis_dsp - ディスパッチ禁止 245
4.9.4 tk_ena_dsp - ディスパッチ許可 247
4.9.5 tk_ref_sys - システム状態参照 248
4.9.6 tk_set_pow - 省電力モード設定 250
4.9.7 tk_ref_ver - バージョン参照 252
4.10 サブシステム管理機能 255
4.10.1 tk_def_ssy - サブシステム定義 257
4.10.2 tk_sta_ssy - スタートアップ関数呼出 262
4.10.3 tk_cln_ssy - クリーンアップ関数呼出 264
4.10.4 tk_evt_ssy - イベント処理関数呼出 265
4.10.5 tk_ref_ssy - サブシステム定義情報の参照 267
4.10.6 tk_cre_res - リソースグループの生成 268
4.10.7 tk_del_res - リソースグループの削除 270
4.10.8 tk_get_res - リソース管理ブロックの取得 271

5 T-Kernel/SMの機能 272
5.1 システムメモリ管理機能 273
5.1.1 システムメモリ割当て 274
5.1.1.1 tk_get_smb - システムメモリの割当て 275
5.1.1.2 tk_rel_smb - システムメモリの解放 277
5.1.1.3 tk_ref_smb - システムメモリ情報取得 278
5.1.2 メモリ割当てライブラリ 279
5.1.2.1 Vmalloc - 非常駐メモリの割当て 280
5.1.2.2 Vcalloc - 非常駐メモリの割当て 281
5.1.2.3 Vrealloc - 非常駐メモリの再割当て 282
5.1.2.4 Vfree - 非常駐メモリの解放 283
5.1.2.5 Kmalloc - 常駐メモリの割当て 284
5.1.2.6 Kcalloc - 常駐メモリの割当て 285
5.1.2.7 Krealloc - 常駐メモリの再割当て 286
5.1.2.8 Kfree - 常駐メモリの解放 287
5.2 アドレス空間管理機能 288
5.2.1 アドレス空間設定 289
5.2.1.1 SetTaskSpace - タスクのアドレス空間設定 290
5.2.2 アドレス空間チェック 292
5.2.2.1 ChkSpaceR - メモリ読込みアクセス権の検査 293
5.2.2.2 ChkSpaceRW - メモリ読込み書込みアクセス権の検査 294
5.2.2.3 ChkSpaceRE - メモリ読込みアクセス権および実行権の検査 295
5.2.2.4 ChkSpaceBstrR - 文字列読込みアクセス権の検査 296
5.2.2.5 ChkSpaceBstrRW - 文字列読込み書込みアクセス権の検査 297

5.2.2.6 ChkSpaceTstrR - TRONコード文字列読込みアクセス権の検査 298
5.2.2.7 ChkSpaceTstrRW - TRONコード文字列読込み書込みアクセス権の検査 299
5.2.3 論理アドレス空間管理 300
5.2.3.1 LockSpace - メモリ領域のロック 301
5.2.3.2 UnlockSpace - メモリ領域のアンロック 303
5.2.3.3 CnvPhysicalAddr - 物理アドレスの取得 305
5.2.3.4 MapMemory - メモリのマップ 307
5.2.3.5 UnmapMemory - メモリのアンマップ 309
5.2.3.6 GetSpaceInfo - アドレス空間の各種情報の取得 310
5.2.3.7 SetMemoryAccess - メモリアクセス権の設定 312
5.3 デバイス管理機能 314
5.3.1 デバイスドライバに関する共通事項 316
5.3.1.1 デバイスの基本概念 316
5.3.1.1.1 デバイス名 (UB*型) 316
5.3.1.1.2 デバイスID (ID型) 317
5.3.1.1.3 デバイス属性 (ATR型) 317
5.3.1.1.4 デバイスディスクリプタ (ID型) 318
5.3.1.1.5 リクエストID (ID型) 318
5.3.1.1.6 データ番号 (W型, D型) 318
5.3.1.2 属性データ 319
5.3.2 デバイスの入出力操作 321
5.3.2.1 tk_opn_dev - デバイスのオープン 322
5.3.2.2 tk_cls_dev - デバイスのクローズ 324
5.3.2.3 tk_rea_dev - デバイスの読込み開始 325
5.3.2.4 tk_rea_dev_du - デバイスの読込み開始(64ビットマイクロ秒単位) 327
5.3.2.5 tk_srea_dev - デバイスの同期読込み 329
5.3.2.6 tk_srea_dev_d - デバイスの同期読込み(64ビット) 331
5.3.2.7 tk_wri_dev - デバイスの書込み開始 333
5.3.2.8 tk_wri_dev_du - デバイスの書込み開始(64ビットマイクロ秒単位) 335
5.3.2.9 tk_swri_dev - デバイスの同期書込み 337
5.3.2.10 tk_swri_dev_d - デバイスの同期書込み(64ビット) 339
5.3.2.11 tk_wai_dev - デバイスの要求完了待ち 341
5.3.2.12 tk_wai_dev_u - デバイスの要求完了待ち(マイクロ秒単位) 343
5.3.2.13 tk_sus_dev - デバイスのサスペンド 345
5.3.2.14 tk_get_dev - デバイスのデバイス名取得 347
5.3.2.15 tk_ref_dev - デバイスのデバイス情報取得 348
5.3.2.16 tk_oref_dev - デバイスのデバイス情報取得 349
5.3.2.17 tk_lst_dev - 登録済みデバイス一覧の取得 350
5.3.2.18 tk_evt_dev - デバイスにドライバ要求イベントを送信 351

5.3.3 デバイスドライバの登録 . . . 352
5.3.3.1 デバイスドライバの登録方法 . . . 352
5.3.3.1.1 tk_def_dev - デバイスの登録 . . . 353
5.3.3.1.2 tk_ref_idv - デバイス初期情報の取得 . . . 355
5.3.3.2 デバイスドライバインタフェース . . . 356
5.3.3.2.1 openfn - オープン関数 . . . 359
5.3.3.2.2 closefn - クローズ関数 . . . 360
5.3.3.2.3 execfn - 処理開始関数 . . . 361
5.3.3.2.4 waitfn - 完了待ち関数 . . . 363
5.3.3.2.5 abortfn - 中止処理関数 . . . 365
5.3.3.2.6 eventfn - イベント関数 . . . 367
5.3.3.3 デバイス事象通知 . . . 369
5.3.3.4 各デバイスのサスペンド／リジューム処理 . . . 371
5.3.3.5 ディスクデバイスの特殊性 . . . 372
5.4 割込み管理機能 . . . 373
5.4.1 CPU割込み制御 . . . 374
5.4.1.1 DI - 外部割込み禁止 . . . 375
5.4.1.2 EI - 外部割込み許可 . . . 376
5.4.1.3 isDI - 外部割込み禁止状態の取得 . . . 377
5.4.2 割込みコントローラ制御 . . . 378
5.4.2.1 DINTNO - 割込みベクタから割込みハンドラ番号へ変換 . . . 379
5.4.2.2 EnableInt - 割込み許可 . . . 380
5.4.2.3 DisableInt - 割込み禁止 . . . 381
5.4.2.4 ClearInt - 割込み発生のクリア . . . 382
5.4.2.5 EndOfInt - 割込みコントローラにEOI発行 . . . 383
5.4.2.6 CheckInt - 割込み発生の検査 . . . 384
5.4.2.7 SetIntMode - 割込みモード設定 . . . 385
5.5 I/Oポートアクセスサポート機能 . . . 386
5.5.1 I/Oポートアクセス . . . 387
5.5.1.1 out_b - I/Oポート書込み(バイト) . . . 388
5.5.1.2 out_h - I/Oポート書込み(ハーフワード) . . . 389
5.5.1.3 out_w - I/Oポート書込み(ワード) . . . 390
5.5.1.4 out_d - I/Oポート書込み(ダブルワード) . . . 391
5.5.1.5 in_b - I/Oポート読込み(バイト) . . . 392
5.5.1.6 in_h - I/Oポート読込み(ハーフワード) . . . 393
5.5.1.7 in_w - I/Oポート読込み(ワード) . . . 394
5.5.1.8 in_d - I/Oポート読込み(ダブルワード) . . . 395
5.5.2 微小待ち . . . 396
5.5.2.1 WaitUsec - 微小待ち(マイクロ秒) . . . 396

5.5.2.2 WaitNsec - 微小待ち(ナノ秒) . . . 397
5.6 省電力機能 . . . 398
5.6.1 low_pow - システムを低消費電力モードに移行 . . . 399
5.6.2 off_pow - システムをサスペンド状態に移行 . . . 400
5.7 システム構成情報管理機能 . . . 401
5.7.1 システム構成情報の取得 . . . 402
5.7.1.1 tk_get_cfn - システム構成情報から数値列取得 . . . 403
5.7.1.2 tk_get_cfs - システム構成情報から文字列取得 . . . 404
5.7.2 標準システム構成情報 . . . 405
5.8 メモリキャッシュ制御機能 . . . 407
5.8.1 SetCacheMode - キャッシュモードの設定 . . . 408
5.8.2 ControlCache - キャッシュの制御 . . . 410
5.9 物理タイマ機能 . . . 412
5.9.1 物理タイマのユースケース . . . 413
5.9.2 StartPhysicalTimer - 物理タイマの動作開始 . . . 414
5.9.3 StopPhysicalTimer - 物理タイマの動作停止 . . . 415
5.9.4 GetPhysicalTimerCount - 物理タイマのカウント値取得 . . . 416
5.9.5 DefinePhysicalTimerHandler - 物理タイマハンドラ定義 . . . 417
5.9.6 GetPhysicalTimerConfig - 物理タイマのコンフィグレーション情報取得 . . . 419
5.10 ユーティリティ機能 . . . 421
5.10.1 オブジェクト名設定 . . . 422
5.10.1.1 SetOBJNAME - オブジェクト名設定 . . . 423
5.10.2 高速ロック・マルチロックライブラリ . . . 424
5.10.2.1 CreateLock - 高速ロックの生成 . . . 425
5.10.2.2 DeleteLock - 高速ロックの削除 . . . 426
5.10.2.3 Lock - 高速ロックのロック操作 . . . 427
5.10.2.4 Unlock - 高速ロックのロック解除操作 . . . 428
5.10.2.5 CreateMLock - 高速マルチロックの生成 . . . 429
5.10.2.6 DeleteMLock - 高速マルチロックの削除 . . . 430
5.10.2.7 MLock - 高速マルチロックのロック操作 . . . 431
5.10.2.8 MLockTmo - 高速マルチロックのロック操作(タイムアウト指定付き) . . . 432
5.10.2.9 MLockTmo_u - 高速マルチロックのロック操作(タイムアウト指定付き、マイクロ秒単位) . . . 433
5.10.2.10 MUnlock - 高速マルチロックのロック解除操作 . . . 434
5.11 サブシステムおよびデバイスドライバの起動 . . . 435
5.11.1 起動処理 . . . 435
5.11.2 終了処理 . . . 436

6 T-Kernel/DSの機能 437
6.1 カーネル内部状態取得機能 438
6.1.1 td_lst_tsk - タスクIDのリスト参照 439
6.1.2 td_lst_sem - セマフォIDのリスト参照 440
6.1.3 td_lst_flg - イベントフラグIDのリスト参照 441
6.1.4 td_lst_mbx - メールボックスIDのリスト参照 442
6.1.5 td_lst_mtx - ミューテックスIDのリスト参照 443
6.1.6 td_lst_mbf - メッセージバッファIDのリスト参照 444
6.1.7 td_lst_por - ランデブポートIDのリスト参照 445
6.1.8 td_lst_mpf - 固定長メモリプールIDのリスト参照 446
6.1.9 td_lst_mpl - 可変長メモリプールIDのリスト参照 447
6.1.10 td_lst_cyc - 周期ハンドラIDのリスト参照 448
6.1.11 td_lst_alm - アラームハンドラIDのリスト参照 449
6.1.12 td_lst_ssy - サブシステムIDのリスト参照 450
6.1.13 td_rdy_que - タスクの優先順位の参照 451
6.1.14 td_sem_que - セマフォの待ち行列の参照 452
6.1.15 td_flg_que - イベントフラグの待ち行列の参照 453
6.1.16 td_mbx_que - メールボックスの待ち行列の参照 454
6.1.17 td_mtx_que - ミューテックスの待ち行列の参照 455
6.1.18 td_smbf_que - メッセージバッファの送信待ち行列の参照 456
6.1.19 td_rmbf_que - メッセージバッファの受信待ち行列の参照 457
6.1.20 td_cal_que - ランデブ呼出待ち行列の参照 458
6.1.21 td_acp_que - ランデブ受付待ち行列の参照 459
6.1.22 td_mpf_que - 固定長メモリプールの待ち行列の参照 460
6.1.23 td_mpl_que - 可変長メモリプールの待ち行列の参照 461
6.1.24 td_ref_tsk - タスク状態参照 462
6.1.25 td_ref_tsk_u - タスク状態参照(マイクロ秒単位) 464
6.1.26 td_ref_tex - タスク例外の状態参照 466
6.1.27 td_ref_sem - セマフォ状態参照 467
6.1.28 td_ref_flg - イベントフラグ状態参照 468
6.1.29 td_ref_mbx - メールボックス状態参照 469
6.1.30 td_ref_mtx - ミューテックス状態参照 470
6.1.31 td_ref_mbf - メッセージバッファ状態参照 471
6.1.32 td_ref_por - ランデブポート状態参照 472
6.1.33 td_ref_mpf - 固定長メモリプール状態参照 473
6.1.34 td_ref_mpl - 可変長メモリプール状態参照 474
6.1.35 td_ref_cyc - 周期ハンドラ状態参照 475
6.1.36 td_ref_cyc_u - 周期ハンドラ状態参照(マイクロ秒単位) 476
6.1.37 td_ref_alm - アラームハンドラ状態参照 477

6.1.38 td_ref_alm_u - アラームハンドラ状態参照(マイクロ秒単位) 478
6.1.39 td_ref_sys - システム状態参照 479
6.1.40 td_ref_ssy - サブシステム定義情報の参照 480
6.1.41 td_inf_tsk - タスク統計情報参照 481
6.1.42 td_inf_tsk_u - タスク統計情報参照(マイクロ秒単位) 482
6.1.43 td_get_reg - タスクレジスタの参照 483
6.1.44 td_set_reg - タスクレジスタの設定 484
6.1.45 td_get_tim - システム時刻参照 485
6.1.46 td_get_tim_u - システム時刻参照(マイクロ秒単位) 486
6.1.47 td_get_otm - システム稼働時間参照 487
6.1.48 td_get_otm_u - システム稼働時間参照(マイクロ秒単位) 488
6.1.49 td_ref_dsname - DSオブジェクト名称の参照 489
6.1.50 td_set_dsname - DSオブジェクト名称の設定 491
6.2 実行トレース機能 492
6.2.1 td_hok_svc - システムコール・拡張SVCのフックルーチン定義 493
6.2.2 td_hok_dsp - タスクディスパッチのフックルーチン定義 495
6.2.3 td_hok_int - 割込みハンドラのフックルーチン定義 497

7 付録 499
7.1 参考とすべきデバイスドライバ関連の仕様 500
7.1.1 デバイス属性のディスク種別 500
7.1.2 デバイスの属性データ 500
7.1.3 デバイス事象通知用の事象タイプ 501

8 リファレンス 502
8.1 C言語インタフェース一覧 503
8.1.1 T-Kernel/OS 503
8.1.1.1 タスク管理機能 503
8.1.1.2 タスク付属同期機能 503
8.1.1.3 タスク例外処理機能 504
8.1.1.4 同期・通信機能 504
8.1.1.5 拡張同期・通信機能 505
8.1.1.6 メモリプール管理機能 505
8.1.1.7 時間管理機能 506
8.1.1.8 割込み管理機能 506
8.1.1.9 システム状態管理機能 506
8.1.1.10 サブシステム管理機能 507
8.1.2 T-Kernel/SM 507
8.1.2.1 システムメモリ管理機能 507

8.1.2.2 アドレス空間管理機能 507
8.1.2.3 デバイス管理機能 508
8.1.2.4 割込み管理機能 509
8.1.2.5 I/Oポートアクセスサポート機能 509
8.1.2.6 省電力機能 509
8.1.2.7 システム構成情報管理機能 510
8.1.2.8 メモリキャッシュ制御機能 510
8.1.2.9 物理タイマ機能 510
8.1.2.10 ユーティリティ機能 510
8.1.3 T-Kernel/DS 510
8.1.3.1 カーネル内部状態取得機能 510
8.1.3.2 実行トレース機能 512
8.2 エラーコード一覧 513
8.2.1 正常終了のエラークラス (0) 513
8.2.2 内部エラークラス (5〜8) 513
8.2.3 未サポートエラークラス (9〜16) 513
8.2.4 パラメータエラークラス (17〜24) 513
8.2.5 呼出コンテキストエラークラス (25〜32) 514
8.2.6 資源不足エラークラス (33〜40) 514
8.2.7 オブジェクト状態エラークラス (41〜48) 514
8.2.8 待ち解除エラークラス (49〜56) 515
8.2.9 デバイスエラークラス (57〜64) (T-Kernel/SM) 515
8.2.10 各種状態エラークラス (65〜72) (T-Kernel/SM) 515

図版目次

1.1 T-Kernelの位置付け 13

2.1 タスク状態遷移図 22
2.2 最初の状態の優先順位 24
2.3 タスクBが実行状態になった後の優先順位 25
2.4 タスクBが待ち状態になった後の優先順位 25
2.5 タスクBが待ち解除された後の優先順位 25
2.6 システム状態の分類 28
2.7 割込みのネストと遅延ディスパッチ 30
2.8 アドレス空間 32

3.1 高級言語対応ルーチンの動作 43

4.1 イベントフラグに対する複数タスク待ちの機能 128
4.2 メールボックスで使用されるメッセージの形式 131
4.3 メッセージバッファによる同期通信 154
4.4 bufsz＝0のメッセージバッファを使った同期式通信 157
4.5 クライアントタスクとサーバタスクの間のランデブの動作 166
4.6 ランデブの動作 167
4.7 select文を使ったAdaのプログラム例 177
4.8 ランデブによるAdaのselect機能の実現方法 177
4.9 tk_fwd_porを使ったサーバタスクの動作イメージ 181
4.10 tk_rot_rdq実行前の優先順位 243
4.11 tk_rot_rdq(tskpri=2)実行後の優先順位 243
4.12 maker のフォーマット 252
4.13 prid のフォーマット 253
4.14 spver のフォーマット 253
4.15 サブシステム概要 255
4.16 サブシステムの依存関係と優先度 263
4.17 サブシステムとリソースグループの関係 268

5.1 デバイス管理機能 315

表目次

2.1 自タスク、他タスクの区別と状態遷移図 23

4.1 tk_ter_tskの対象タスクの状態と実行結果 55
4.2 tskwait と wid の値 75
4.3 tk_rel_waiの対象タスクの状態と実行結果 86

5.1 同じデバイスを同時にオープンしようとしたときの可否 323

APIの記述形式

本仕様書のAPI説明の部分では、API(Application Program Interface)ごとに、以下のような形式で仕様の説明を行っている。なお、APIには、カーネルの機能を直接的に呼び出すシステムコールのほかに、拡張SVC(拡張システムコール)やマクロ、ライブラリとして実現されるものも含まれる。

API名称 - 説明

APIの名称および説明を示す。

C言語インタフェース

APIのC言語インタフェースおよびインクルードするヘッダファイルを示す。

パラメータ

APIのパラメータ、すなわちAPIを発行するときにT-Kernelに渡す情報に関する説明を行う。

リターンパラメータ

APIのリターンパラメータ、すなわちAPIの実行が終ったときにT-Kernelから返される情報に関する説明を行う。

なお、リターンパラメータのうち、APIの関数値として戻されるものを「戻値」と呼ぶことがある。リターンパラメータには、戻値のほかに、パラメータとして渡されたポインタの参照先に情報を返すものがある。

エラーコード

APIで発生する可能性のあるエラーに関して説明を行う。

以下のエラーコードについては、各APIのエラーコードの説明の項には含めていないが、各APIにおいて共通に発生する可能性がある。

E_SYS , E_NOSPT , E_RSFN , E_MACV , E_OACV

以下のエラーコードの検出は実装依存であり、エラーとして検出されない場合がある。

E_PAR , E_MACV , E_CTX

エラーコード E_CTX については、待ち状態に入るAPIのタスク独立部からの呼出など、APIの呼出コンテキストに意味的な誤りがある場合についてのみ、各APIのエラーコードの説明に含めている。呼出コンテキストに関する制約が実装依存であり、必ずしもエラーとする必要がない条件については、各APIのエラーコードの説明の項には含めていない。

また、実装上の制限事項により、エラーコードの説明に含まれない条件でエラーが発生する可能性もある。

利用可能なコンテキスト

APIを発行することができるコンテキスト(タスク部、準タスク部、タスク独立部)を示す。

解説

APIの機能の解説を行う。

いくつかの値を選択して設定するようなパラメータの場合には、以下のような記述方法によって仕様説明を行っている。

(x || y || z)
　　x, y, z のいずれか一つを選択して指定する。

x | y
　　x と y を同時に指定可能である。(同時に指定する場合は x と y の論理和をとる)

[x]
　　x は指定しても指定しなくても良い。

パラメータの記述例

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

の場合、wfmode の指定は次の4種のいずれかになる。

```
  TWF_ANDW
  TWF_ORW
 (TWF_ANDW | TWF_CLR)
 (TWF_ORW | TWF_CLR)
```

補足事項

特記事項や注意すべき点など、解説に対する補足事項を述べる。

仕様決定の理由

仕様決定の理由を述べる。

T-Kernel 1.0との差異

T-Kernel 1.0と異なる部分について説明する。

T-Kernel/OS システムコール索引

この索引は、本仕様書で説明されるT-Kernel/OSのシステムコールのアルファベット順索引である。

- tk_acp_por - ランデブポートに対するランデブ受付
- tk_acp_por_u - ランデブポートに対するランデブ受付(マイクロ秒単位)
- tk_cal_por - ランデブポートに対するランデブの呼出
- tk_cal_por_u - ランデブポートに対するランデブの呼出(マイクロ秒単位)
- tk_can_wup - タスクの起床要求を無効化
- tk_chg_pri - タスク優先度変更
- tk_chg_slt - タスクスライスタイム変更
- tk_chg_slt_u - タスクスライスタイム変更(マイクロ秒単位)
- tk_cln_ssy - クリーンアップ関数呼出
- tk_clr_flg - イベントフラグのクリア
- tk_cre_alm - アラームハンドラの生成
- tk_cre_cyc - 周期ハンドラの生成
- tk_cre_cyc_u - 周期ハンドラの生成(マイクロ秒単位)
- tk_cre_flg - イベントフラグ生成
- tk_cre_mbf - メッセージバッファ生成
- tk_cre_mbx - メールボックス生成
- tk_cre_mpf - 固定長メモリプール生成
- tk_cre_mpl - 可変長メモリプール生成
- tk_cre_mtx - ミューテックス生成
- tk_cre_por - ランデブポート生成
- tk_cre_res - リソースグループの生成
- tk_cre_sem - セマフォ生成
- tk_cre_tsk - タスク生成
- tk_def_int - 割込みハンドラ定義
- tk_def_ssy - サブシステム定義
- tk_def_tex - タスク例外ハンドラの定義

- tk_del_alm - アラームハンドラの削除
- tk_del_cyc - 周期ハンドラの削除
- tk_del_flg - イベントフラグ削除
- tk_del_mbf - メッセージバッファ削除
- tk_del_mbx - メールボックス削除
- tk_del_mpf - 固定長メモリプール削除
- tk_del_mpl - 可変長メモリプール削除
- tk_del_mtx - ミューテックス削除
- tk_del_por - ランデブポート削除
- tk_del_res - リソースグループの削除
- tk_del_sem - セマフォ削除
- tk_del_tsk - タスク削除
- tk_dis_dsp - ディスパッチ禁止
- tk_dis_tex - タスク例外の禁止
- tk_dis_wai - タスク待ち状態の禁止
- tk_dly_tsk - タスク遅延
- tk_dly_tsk_u - タスク遅延(マイクロ秒単位)
- tk_ena_dsp - ディスパッチ許可
- tk_ena_tex - タスク例外の許可
- tk_ena_wai - タスク待ち禁止の解除
- tk_end_tex - タスク例外ハンドラの終了
- tk_evt_ssy - イベント処理関数呼出
- tk_exd_tsk - 自タスクの終了と削除
- tk_ext_tsk - 自タスク終了
- tk_frsm_tsk - 強制待ち状態のタスクを強制再開
- tk_fwd_por - ランデブポートに対するランデブ回送
- tk_get_cpr - コプロセッサのレジスタの取得
- tk_get_mpf - 固定長メモリブロック獲得
- tk_get_mpf_u - 固定長メモリブロック獲得(マイクロ秒単位)
- tk_get_mpl - 可変長メモリブロック獲得
- tk_get_mpl_u - 可変長メモリブロック獲得(マイクロ秒単位)
- tk_get_otm - システム稼働時間参照
- tk_get_otm_u - システム稼働時間参照(マイクロ秒単位)
- tk_get_reg - タスクレジスタの取得
- tk_get_res - リソース管理ブロックの取得

- tk_get_rid - タスクの所属リソースグループの参照
- tk_get_tid - 実行状態タスクのタスクID参照
- tk_get_tim - システム時刻参照
- tk_get_tim_u - システム時刻参照(マイクロ秒単位)
- tk_get_tsp - タスク固有空間の参照
- tk_inf_tsk - タスク統計情報参照
- tk_inf_tsk_u - タスク統計情報参照(マイクロ秒単位)
- tk_loc_mtx - ミューテックスのロック
- tk_loc_mtx_u - ミューテックスのロック(マイクロ秒単位)
- tk_ras_tex - タスク例外を発生
- tk_rcv_mbf - メッセージバッファから受信
- tk_rcv_mbf_u - メッセージバッファから受信(マイクロ秒単位)
- tk_rcv_mbx - メールボックスから受信
- tk_rcv_mbx_u - メールボックスから受信(マイクロ秒単位)
- tk_ref_alm - アラームハンドラ状態参照
- tk_ref_alm_u - アラームハンドラ状態参照(マイクロ秒単位)
- tk_ref_cyc - 周期ハンドラ状態参照
- tk_ref_cyc_u - 周期ハンドラ状態参照(マイクロ秒単位)
- tk_ref_flg - イベントフラグ状態参照
- tk_ref_mbf - メッセージバッファ状態参照
- tk_ref_mbx - メールボックス状態参照
- tk_ref_mpf - 固定長メモリプール状態参照
- tk_ref_mpl - 可変長メモリプール状態参照
- tk_ref_mtx - ミューテックス状態参照
- tk_ref_por - ランデブポート状態参照
- tk_ref_sem - セマフォ状態参照
- tk_ref_ssy - サブシステム定義情報の参照
- tk_ref_sys - システム状態参照
- tk_ref_tex - タスク例外の状態参照
- tk_ref_tsk - タスク状態参照
- tk_ref_tsk_u - タスク状態参照(マイクロ秒単位)
- tk_ref_ver - バージョン参照
- tk_rel_mpf - 固定長メモリブロック返却
- tk_rel_mpl - 可変長メモリブロック返却
- tk_rel_wai - 他タスクの待ち状態解除

- tk_ret_int - 割込みハンドラから復帰
- tk_rot_rdq - タスクの優先順位の回転
- tk_rpl_rdv - ランデブ返答
- tk_rsm_tsk - 強制待ち状態のタスクを再開
- tk_set_cpr - コプロセッサのレジスタの設定
- tk_set_flg - イベントフラグのセット
- tk_set_pow - 省電力モード設定
- tk_set_reg - タスクレジスタの設定
- tk_set_rid - タスクの所属リソースグループの設定
- tk_set_tim - システム時刻設定
- tk_set_tim_u - システム時刻設定(マイクロ秒単位)
- tk_set_tsp - タスク固有空間の設定
- tk_sig_sem - セマフォ資源返却
- tk_sig_tev - タスクイベントの送信
- tk_slp_tsk - 自タスクを起床待ち状態へ移行
- tk_slp_tsk_u - 自タスクを起床待ち状態へ移行(マイクロ秒単位)
- tk_snd_mbf - メッセージバッファへ送信
- tk_snd_mbf_u - メッセージバッファへ送信(マイクロ秒単位)
- tk_snd_mbx - メールボックスへ送信
- tk_sta_alm - アラームハンドラの動作開始
- tk_sta_alm_u - アラームハンドラの動作開始(マイクロ秒単位)
- tk_sta_cyc - 周期ハンドラの動作開始
- tk_sta_ssy - スタートアップ関数呼出
- tk_sta_tsk - タスク起動
- tk_stp_alm - アラームハンドラの動作停止
- tk_stp_cyc - 周期ハンドラの動作停止
- tk_sus_tsk - 他タスクを強制待ち状態へ移行
- tk_ter_tsk - 他タスク強制終了
- tk_unl_mtx - ミューテックスのアンロック
- tk_wai_flg - イベントフラグ待ち
- tk_wai_flg_u - イベントフラグ待ち(マイクロ秒単位)
- tk_wai_sem - セマフォ資源獲得
- tk_wai_sem_u - セマフォ資源獲得(マイクロ秒単位)
- tk_wai_tev - タスクイベント待ち
- tk_wai_tev_u - タスクイベント待ち(マイクロ秒単位)
- tk_wup_tsk - 他タスクの起床

T-Kernel/SM 拡張SVC・ライブラリ索引

この索引は、本仕様書で説明されるT-Kernel/SMの拡張SVC・ライブラリのアルファベット順索引である。

- abortfn - 中止処理関数
- CheckInt - 割込み発生の検査
- ChkSpaceBstrR - 文字列読込みアクセス権の検査
- ChkSpaceBstrRW - 文字列読込み書込みアクセス権の検査
- ChkSpaceR - メモリ読込みアクセス権の検査
- ChkSpaceRE - メモリ読込みアクセス権および実行権の検査
- ChkSpaceRW - メモリ読込み書込みアクセス権の検査
- ChkSpaceTstrR - TRONコード文字列読込みアクセス権の検査
- ChkSpaceTstrRW - TRONコード文字列読込み書込みアクセス権の検査
- ClearInt - 割込み発生のクリア
- closefn - クローズ関数
- CnvPhysicalAddr - 物理アドレスの取得
- ControlCache - キャッシュの制御
- CreateLock - 高速ロックの生成
- CreateMLock - 高速マルチロックの生成
- DefinePhysicalTimerHandler - 物理タイマハンドラ定義
- DeleteLock - 高速ロックの削除
- DeleteMLock - 高速マルチロックの削除
- DI - 外部割込み禁止
- DINTNO - 割込みベクタから割込みハンドラ番号へ変換
- DisableInt - 割込み禁止
- EI - 外部割込み許可
- EnableInt - 割込み許可
- EndOfInt - 割込みコントローラにEOI発行
- eventfn - イベント関数
- execfn - 処理開始関数

- GetPhysicalTimerConfig - 物理タイマのコンフィグレーション情報取得
- GetPhysicalTimerCount - 物理タイマのカウント値取得
- GetSpaceInfo - アドレス空間の各種情報の取得
- in_b - I/Oポート読込み(バイト)
- in_d - I/Oポート読込み(ダブルワード)
- in_h - I/Oポート読込み(ハーフワード)
- in_w - I/Oポート読込み(ワード)
- isDI - 外部割込み禁止状態の取得
- Kcalloc - 常駐メモリの割当て
- Kfree - 常駐メモリの解放
- Kmalloc - 常駐メモリの割当て
- Krealloc - 常駐メモリの再割当て
- Lock - 高速ロックのロック操作
- LockSpace - メモリ領域のロック
- low_pow - システムを低消費電力モードに移行
- MapMemory - メモリのマップ
- MLock - 高速マルチロックのロック操作
- MLockTmo - 高速マルチロックのロック操作(タイムアウト指定付き)
- MLockTmo_u - 高速マルチロックのロック操作(タイムアウト指定付き、マイクロ秒単位)
- MUnlock - 高速マルチロックのロック解除操作
- off_pow - システムをサスペンド状態に移行
- openfn - オープン関数
- out_b - I/Oポート書込み(バイト)
- out_d - I/Oポート書込み(ダブルワード)
- out_h - I/Oポート書込み(ハーフワード)
- out_w - I/Oポート書込み(ワード)
- SetCacheMode - キャッシュモードの設定
- SetIntMode - 割込みモード設定
- SetMemoryAccess - メモリアクセス権の設定
- SetOBJNAME - オブジェクト名設定
- SetTaskSpace - タスクのアドレス空間設定
- StartPhysicalTimer - 物理タイマの動作開始
- StopPhysicalTimer - 物理タイマの動作停止
- tk_cls_dev - デバイスのクローズ
- tk_def_dev - デバイスの登録

- tk_evt_dev - デバイスにドライバ要求イベントを送信
- tk_get_cfn - システム構成情報から数値列取得
- tk_get_cfs - システム構成情報から文字列取得
- tk_get_dev - デバイスのデバイス名取得
- tk_get_smb - システムメモリの割当て
- tk_lst_dev - 登録済みデバイス一覧の取得
- tk_opn_dev - デバイスのオープン
- tk_oref_dev - デバイスのデバイス情報取得
- tk_rea_dev - デバイスの読込み開始
- tk_rea_dev_du - デバイスの読込み開始(64ビットマイクロ秒単位)
- tk_ref_dev - デバイスのデバイス情報取得
- tk_ref_idv - デバイス初期情報の取得
- tk_ref_smb - システムメモリ情報取得
- tk_rel_smb - システムメモリの解放
- tk_srea_dev - デバイスの同期読込み
- tk_srea_dev_d - デバイスの同期読込み(64ビット)
- tk_sus_dev - デバイスのサスペンド
- tk_swri_dev - デバイスの同期書込み
- tk_swri_dev_d - デバイスの同期書込み(64ビット)
- tk_wai_dev - デバイスの要求完了待ち
- tk_wai_dev_u - デバイスの要求完了待ち(マイクロ秒単位)
- tk_wri_dev - デバイスの書込み開始
- tk_wri_dev_du - デバイスの書込み開始(64ビットマイクロ秒単位)
- Unlock - 高速ロックのロック解除操作
- UnlockSpace - メモリ領域のアンロック
- UnmapMemory - メモリのアンマップ
- Vcalloc - 非常駐メモリの割当て
- Vfree - 非常駐メモリの解放
- Vmalloc - 非常駐メモリの割当て
- Vrealloc - 非常駐メモリの再割当て
- waitfn - 完了待ち関数
- WaitNsec - 微小待ち(ナノ秒)
- WaitUsec - 微小待ち(マイクロ秒)

T-Kernel/DS システムコール索引

この索引は、本仕様書で説明されるT-Kernel/DSのシステムコールのアルファベット順索引である。

- td_acp_que - ランデブ受付待ち行列の参照
- td_cal_que - ランデブ呼出待ち行列の参照
- td_flg_que - イベントフラグの待ち行列の参照
- td_get_otm - システム稼働時間参照
- td_get_otm_u - システム稼働時間参照(マイクロ秒単位)
- td_get_reg - タスクレジスタの参照
- td_get_tim - システム時刻参照
- td_get_tim_u - システム時刻参照(マイクロ秒単位)
- td_hok_dsp - タスクディスパッチのフックルーチン定義
- td_hok_int - 割込みハンドラのフックルーチン定義
- td_hok_svc - システムコール・拡張SVCのフックルーチン定義
- td_inf_tsk - タスク統計情報参照
- td_inf_tsk_u - タスク統計情報参照(マイクロ秒単位)
- td_lst_alm - アラームハンドラIDのリスト参照
- td_lst_cyc - 周期ハンドラIDのリスト参照
- td_lst_flg - イベントフラグIDのリスト参照
- td_lst_mbf - メッセージバッファIDのリスト参照
- td_lst_mbx - メールボックスIDのリスト参照
- td_lst_mpf - 固定長メモリプールIDのリスト参照
- td_lst_mpl - 可変長メモリプールIDのリスト参照
- td_lst_mtx - ミューテックスIDのリスト参照
- td_lst_por - ランデブポートIDのリスト参照
- td_lst_sem - セマフォIDのリスト参照
- td_lst_ssy - サブシステムIDのリスト参照
- td_lst_tsk - タスクIDのリスト参照
- td_mbx_que - メールボックスの待ち行列の参照

- td_mpf_que - 固定長メモリプールの待ち行列の参照
- td_mpl_que - 可変長メモリプールの待ち行列の参照
- td_mtx_que - ミューテックスの待ち行列の参照
- td_rdy_que - タスクの優先順位の参照
- td_ref_alm - アラームハンドラ状態参照
- td_ref_alm_u - アラームハンドラ状態参照(マイクロ秒単位)
- td_ref_cyc - 周期ハンドラ状態参照
- td_ref_cyc_u - 周期ハンドラ状態参照(マイクロ秒単位)
- td_ref_dsname - DSオブジェクト名称の参照
- td_ref_flg - イベントフラグ状態参照
- td_ref_mbf - メッセージバッファ状態参照
- td_ref_mbx - メールボックス状態参照
- td_ref_mpf - 固定長メモリプール状態参照
- td_ref_mpl - 可変長メモリプール状態参照
- td_ref_mtx - ミューテックス状態参照
- td_ref_por - ランデブポート状態参照
- td_ref_sem - セマフォ状態参照
- td_ref_ssy - サブシステム定義情報の参照
- td_ref_sys - システム状態参照
- td_ref_tex - タスク例外の状態参照
- td_ref_tsk - タスク状態参照
- td_ref_tsk_u - タスク状態参照(マイクロ秒単位)
- td_rmbf_que - メッセージバッファの受信待ち行列の参照
- td_sem_que - セマフォの待ち行列の参照
- td_set_dsname - DSオブジェクト名称の設定
- td_set_reg - タスクレジスタの設定
- td_smbf_que - メッセージバッファの送信待ち行列の参照

第 1 章

T-Kernelの概要

1.1 全体的な位置付け

T-KernelのT-Engineシステム全体における位置付けを[図 1.1.「T-Kernelの位置付け」]に示す。

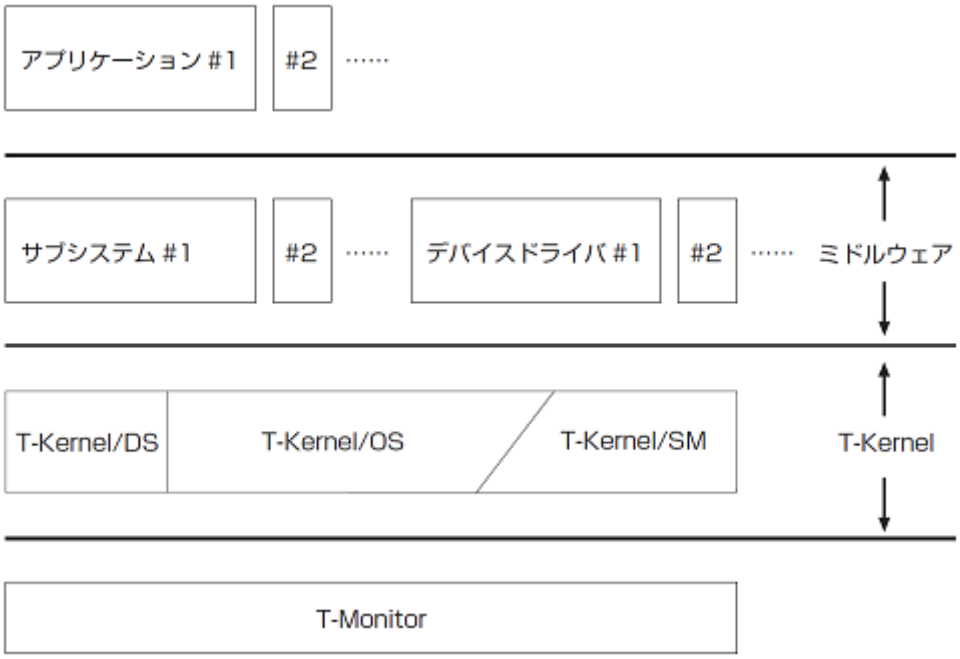

図 1.1: T-Kernelの位置付け

T-Kernelは、T-Kernel/OS(Operating System)、T-Kernel/SM(System Manager)、および T-Kernel/DS(Debugger Support)を含む全体を指すが、T-Kernel/OSのみを(狭義の)T-Kernelと呼ぶ場合もある。

T-Kernel/OSは、以下のような機能を提供する。

- タスク管理機能
- タスク付属同期機能
- タスク例外処理機能
- 同期・通信機能
- 拡張同期・通信機能
- メモリプール管理機能
- 時間管理機能
- 割込み管理機能
- システム状態管理機能
- サブシステム管理機能

T-Kernel/SMは、以下のような機能を提供する。

- システムメモリ管理機能
- アドレス空間管理機能

- デバイス管理機能
- 割込み管理機能
- I/Oポートアクセスサポート機能
- 省電力機能
- システム構成情報管理機能
- メモリキャッシュ制御機能
- 物理タイマ機能
- ユーティリティ機能

T-Kernel/DS(Debugger Support)は、デバッガ専用に以下のような機能を提供する。

- カーネル内部状態取得機能
- 実行トレース機能

T-Kernel 1.0との差異
メモリキャッシュ制御機能、物理タイマ機能、ユーティリティ機能は、T-Kernel 2.0で追加された機能である。

1.2 適応化

T-Kernelは、組込みシステム用のリアルタイムオペレーティングシステムとして、小規模から大規模のさまざまなシステムをターゲットとしており、デバイスドライバやミドルウェアの流通性の向上をねらっている。

T-Kernelは規模の大きなシステムにも対応できるように仕様が策定されている。そのため、小規模なシステム向けには必須とはされない機能も含まれているが、サブセット仕様を定めてしまうと、デバイスドライバやミドルウェア等の流通性や移植性を妨げることになる。また、ターゲットによって必要な機能はさまざまであり、一様にサブセット仕様を定めることも難しい。

T-Kernelでは、レベル分けなどのサブセット化のための仕様は定めない。原則として、すべてのT-Kernelはすべての仕様を実装しなければならない。ただし、ターゲットシステムで必要とされない機能は、簡易な実装として構わない。

「簡易な実装」とは、仕様通りの機能は持っていないが、その機能を呼び出すことで動作が異常になるようなことがない実装ということができる。つまり、大規模のシステムをターゲットとして作成されたミドルウェアが、そのまま実行できるような環境を提供することが重要となる。例えば、MMU(Memory Management Unit)を使用しないシステムでは、T-Kernel/SMの LockSpace() は次のように実装しても構わない。

```
#define LockSpace(addr, len)    ( E_OK )
```

しかし、MMUを使用しないシステムだからといって、LockSpace() を実装しなかったり、E_NOSPT を返すような実装としてはいけない。

逆に、ミドルウェアの作成においても、MMUを使用しないシステムだからといって、LockSpace() を使用しない実装をしてしまうと、MMUを使用したシステムには対応できなくなってしまう。

T-Kernelをターゲットシステムに実装する際に、必要とされない機能を省いたり、機能を変更することは構わない。ただし、そのT-Kernelは、改変されたT-Kernelとみなされる。

ミドルウェア提供者は、以下の点に注意しなくてはならない。

- T-Kernelが提供しているすべての要件を満たすように、ミドルウェアは作成されなければならない。つまり、特定のターゲットシステムに限定するのではなく、さまざまな規模のシステムにも対応できる必要がある。
- 利用者が不要な機能を省けるような仕組を提供してもよい。

1.3 T-Kernel 2.0の概要

1.3.1 T-Kernel 2.0の位置付けと基本方針

T-Engineフォーラム発足時に公開されたT-Kernel(T-Kernel 1.0)の実績や採用製品の着実な増加にともない、より高性能化、高機能化するハードウェアを活かせるような機能追加の要求が出てきた。このような要求に応える形で、新たな展開へのステップとして策定されたリアルタイムオペレーティングシステムの仕様がT-Kernel 2.0である。

T-Kernel 2.0の仕様は、これまでのT-Kernelの実績をそのまま活かしつつ、機能強化されたカーネルへの移行をスムーズに進められるように、T-Kernel 1.0の仕様に対して上位互換の関係になっている。ソース互換のみならずバイナリ互換も可能であり、たとえばT-Kernel 1.0をT-Kernel 2.0にバージョンアップした場合にも、T-Kernel 1.0上で動いていたデバイスドライバ、ミドルウェア、アプリケーション等は再コンパイル不要でそのまま動く。

またT-Kernel 2.0では、仕様書の記述をXMLベースとして閲覧性や検索性を高めたほか、表現や説明などについても、分かりにくい部分や古い部分の見直しを行っている。

1.3.2 T-Kernel 2.0の追加機能

1. マイクロ秒単位の時間管理機能

   周期ハンドラやアラームハンドラなどの時間管理機能、タイムアウトなどの時間関係の機能において、T-Kernel 1.0で扱える時間の単位はミリ秒であったが、T-Kernel 2.0ではマイクロ秒単位の時間を扱うAPIを追加する。

   マイクロ秒単位の時間を扱うにあたり、32ビット幅のデータでは扱える時間の長さが短くなってしまうため、時間関係のパラメータとして64ビット幅のデータを導入する(以下の「64ビット整数のデータタイプの導入」を参照)。

   64ビットでマイクロ秒単位のパラメータを持つAPIの名称は、対応するT-Kernel 1.0のAPI名称の後ろに"_u"を付けた名称とする。u は μ の意味である。また、64ビットでマイクロ秒単位となったパラメータには、パラメータ名称の後ろにも"_u"を付ける。

例 1.1 64ビットマイクロ秒単位のAPIの例

```
/* T-Kernel 1.0の32ビットミリ秒単位のAPI */
tk_sta_alm( ID almid, RELTIM almtim )

/* T-Kernel 2.0の64ビットマイクロ秒単位のAPI */
tk_sta_alm_u( ID almid, RELTIM_U almtim_u )
```

   なお、T-Kernel 2.0でも、時間を扱うAPIがマイクロ秒単位に一本化されるわけではない。上位互換の基本方針により、T-Kernel 1.0のミリ秒単位のAPIはT-Kernel 2.0でもそのまま使うことができ、両者が併存する形となる。

   T-Kernelの時間管理機能における実際の時間分解能は、「5.7.2節「標準システム構成情報」」の「タイマ割込み間隔」(TTimPeriod)によって指定される。したがって、時間管理機能のパラメータにおけるマイクロ秒単位の指定を正確に実現するには、「タイマ割込み間隔」(TTimPeriod)を十分短い時間に設定する必要がある。詳細は「5.7.2節「標準システム構成情報」」を参照のこと。

   ちなみに、32ビット符号付き整数で扱える最長時間は、ミリ秒単位とすると約24日間であるが、マイクロ秒単位とすると約35分間になる。これを64ビット幅のデータとすることにより、実用上の制限のない時間を扱うことができる。

2. 大容量デバイスへの対応

   ハードディスクなど大容量デバイスへの対応を考慮し、デバイス管理機能のパラメータの一部についても64ビット幅のデータを扱えるようにする。

   64ビットのパラメータを持つAPIの名称は、対応するT-Kernel 1.0のAPI名称の後ろに"_d"を付けた名称とする。dは double integer の意味である。また、64ビットとなったパラメータには、パラメータ名称の後ろにも"_d"を付ける。

例 1.2 64ビットパラメータを持つAPIの例

```
/* T-Kernel 1.0のAPI */
tk_swri_dev( ID dd, W start, VP buf, W size, W *asize )

/* T-Kernel 2.0の64ビットパラメータを持つAPI */
tk_swri_dev_d( ID dd, D start_d, void *buf, W size, W *asize )
```

たとえば、ブロックサイズ512バイトの一般的なハードディスクの場合、データ幅が32ビットのT-Kernel 1.0で扱える最大容量は約1TB(＝512バイト×(2 ^ 31))であった(' ^ 'はべき乗を表す)。T-Kernel 2.0では、データ幅を64ビットとしたAPIを追加することにより、この制限を解消する。

3. 64ビット整数のデータタイプの導入

   前2項を実現するため、データタイプやAPIのパラメータの一部に64ビットの整数を導入する。このため、C言語の規格として正式に仕様化(C99)されたlong long型をT-Kernelの仕様にも採用する。

   64ビット整数を表すデータタイプの名称は、符号付きをD、符号無しをUDとする。'D'は Double integer の意味である。

4. その他の機能追加

   キャッシュ関連の機能、物理タイマ機能、ユーティリティ機能の追加などを行う。

第 2 章

T-Kernelの概念

2.1 基本的な用語の意味

タスクと自タスク

プログラムの並行実行の単位を「タスク」と呼ぶ。すなわち、一つのタスク中のプログラムは逐次的に実行されるのに対して、異なるタスクのプログラムは並行して実行が行われる。ただし、並行して実行が行われるというのは、アプリケーションから見た概念的な動作であり、実装上はカーネルの制御のもとで、それぞれのタスクが時分割で実行される。

また、システムコールを呼び出したタスクを「自タスク」と呼ぶ。

ディスパッチとディスパッチャ

プロセッサが実行するタスクを切り替えることを「ディスパッチ」(または「タスクディスパッチ」)と呼ぶ。また、ディスパッチを実現するカーネル内の機構を「ディスパッチャ」(または「タスクディスパッチャ」)と呼ぶ。

スケジューリングとスケジューラ

次に実行すべきタスクを決定する処理を「スケジューリング」(または「タスクスケジューリング」)と呼ぶ。また、スケジューリングを実現するカーネル内の機構を「スケジューラ」(または「タスクスケジューラ」)と呼ぶ。スケジューラは、一般的な実装では、システムコール処理の中やディスパッチャの中で実現される。

コンテキスト

一般に、プログラムの実行される環境を「コンテキスト」と呼ぶ。コンテキストが同じというためには、少なくとも、プロセッサの動作モードが同一で、用いているスタック空間が同一(スタック領域が一連)でなければならない。ただし、コンテキストはアプリケーションから見た概念であり、独立したコンテキストで実行すべき処理であっても、実装上は同一のプロセッサ動作モードで同一のスタック空間で実行されることもある。

優先順位

処理が実行される順序を決める順序関係を「優先順位」と呼ぶ。優先順位の低い処理を実行中に、優先順位の高い処理が実行できる状態になった場合、優先順位の高い処理を先に実行するのが原則である。

補足事項
優先度は、タスクやメッセージの処理順序を制御するために、アプリケーションによって与えられるパラメータである。
それに対して優先順位は、仕様中で処理の実行順序を明確にするために用いる概念である。
タスク間の優先順位は、タスクの優先度に基づいて定められる。

APIとシステムコール

アプリケーションやミドルウェアからT-Kernelの提供する機能を呼び出すための標準インタフェースを総称して、API(Application Program Interface)と呼ぶ。APIには、カーネルの機能を直接的に呼び出すシステムコールのほか、拡張SVCやマクロ、ライブラリとして実現されるものも含まれる。

T-Kernel/OSやT-Kernel/DSの機能を呼び出すAPIはシステムコールであり、T-Kernel/SMの機能を呼び出すAPIは拡張SVC、マクロ、ライブラリのいずれかである。

カーネル

狭義の意味のカーネルは、T-Kernel/OSおよびT-Kernel/DSを合わせたものを指す。広義の意味ではT-Kernel全体を指す。

T-Kernel/SMは、T-Kernel/OSのサブシステムの機能を用いたT-Kernel/OSの拡張機能であるため、厳密な意味ではカーネルではない。

一方、T-KernelあるいはT-Kernel本体と呼んだ場合には、T-Kernel/OS、T-Kernel/SM、T-Kernel/DSを合わせたものを指す。

実装定義

仕様として標準化していない事項である。実装ごとに仕様を規定しなければならない。実装仕様書に具体的な実装内容について明記しなければならない。アプリケーションプログラムにおいて、実装定義の事項に依存している部分は移植性が確保されない。

実装依存

仕様において、ターゲットシステム、または、システムの動作条件によって振舞いが変わる事項であることを示す。実装ごとに振舞いを規定しなければならない。実装仕様書に具体的な実装内容について明記しなければならない。アプリケーションプログラムにおいて、実装依存の事項に依存している部分は基本的に移植する際に変更が必要となる。

2.2 タスク状態とスケジューリング規則

2.2.1 タスク状態

タスク状態は、大きく次の5つに分類される。この内、広義の待ち状態は、さらに3つの状態に分類される。また、実行状態と実行可能状態を総称して、実行できる状態と呼ぶ。

実行状態 (RUNNING)
: 現在そのタスクを実行中であるという状態。ただし、タスク独立部を実行している間は、別に規定されている場合を除いて、タスク独立部の実行を開始する前に実行していたタスクが実行状態であるものとする。

実行可能状態 (READY)
: そのタスクを実行する準備は整っているが、そのタスクよりも優先順位の高いタスクが実行中であるために、そのタスクを実行できない状態。言い換えると、実行できる状態のタスクの中で最高の優先順位になればいつでも実行できる状態。

広義の待ち状態
: そのタスクを実行できる条件が整わないために、実行ができない状態。言い換えると、何らかの条件が満たされるのを待っている状態。タスクが広義の待ち状態にある間、プログラムカウンタやレジスタなどのプログラムの実行状態を表現する情報は保存されている。タスクを広義の待ち状態から実行再開する時には、プログラムカウンタやレジスタなどを広義の待ち状態になる直前の値に戻す。広義の待ち状態は、さらに次の3つの状態に分類される。

  待ち状態 (WAITING)
  : 何らかの条件が整うまで自タスクの実行を中断するシステムコールを呼び出したことにより、実行が中断された状態。

  強制待ち状態 (SUSPENDED)
  : 他のタスクによって、強制的に実行を中断させられた状態。

  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。待ち状態にあるタスクに対して、強制待ち状態への移行が要求されると、二重待ち状態に移行させる。

    T-Kernelでは「待ち状態(WAITING)」と「強制待ち状態(SUSPENDED)」を明確に区別しており、タスクが自ら強制待ち状態(SUSPENDED)になることはできない。

休止状態 (DORMANT)
: タスクがまだ起動されていないか、実行を終了した後の状態。タスクが休止状態にある間は、実行状態を表現する情報は保存されていない。タスクを休止状態から起動する時には、タスクの起動番地から実行を開始する。また、別に規定されている場合を除いて、レジスタの内容は保証されない。

未登録状態 (NON-EXISTENT)
: タスクがまだ生成されていないか、削除された後の、システムに登録されていない仮想的な状態。

実装によっては、以上のいずれにも分類されない過渡的な状態が存在する場合がある(2.5節「システム状態」参照)。

実行可能状態に移行したタスクが、現在実行中のタスクよりも高い優先順位を持つ場合には、実行可能状態への移行と同時にディスパッチが起こり、即座に実行状態へ移行する場合がある。この場合、それまで実行状態であったタスクは、新たに実行状態へ移行したタスクにプリエンプトされたという。また、システムコールの機能説明などで、「実行可能状態に移行させる」と記述されている場合でも、タスクの優先順位によっては、即座に実行状態に移行させる場合もある。

タスクの起動とは、休止状態のタスクを実行可能状態に移行させることをいう。このことから、休止状態と未登録状態以外の状態を総称して、起動された状態と呼ぶことがある。タスクの終了とは、起動された状態のタスクを休止状態に移行させることをいう。

タスクの待ち解除とは、タスクが待ち状態の時は実行可能状態に、二重待ち状態の時は強制待ち状態に移行させることをいう。また、タスクの強制待ちからの再開とは、タスクが強制待ち状態の時は実行可能状態に、二重待ち状態の時は待ち状態に移行させることをいう。

一般的な実装におけるタスクの状態遷移を[図 2.1.「タスク状態遷移図」]に示す。実装によっては、この図にない状態遷移を行う場合がある。

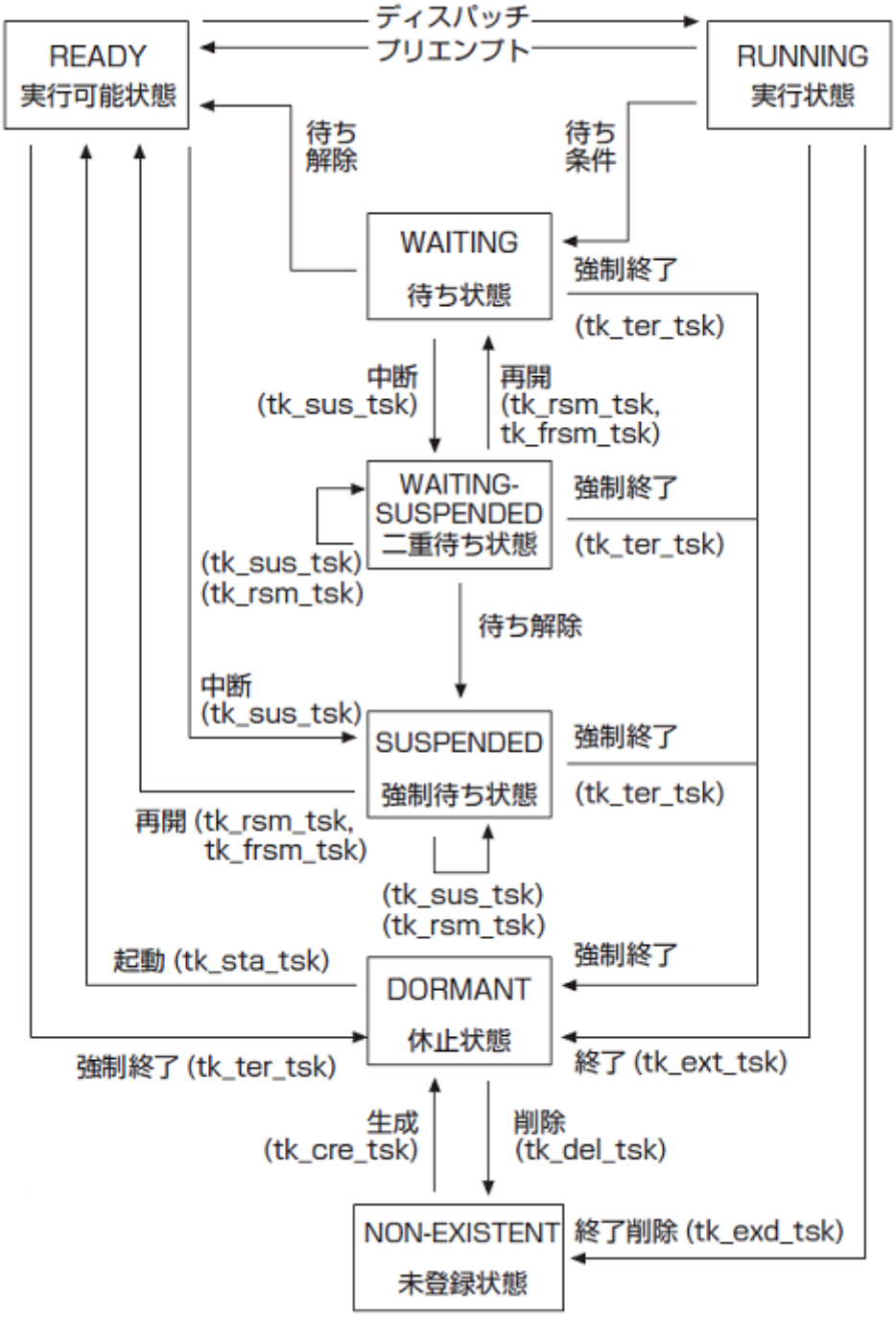

図 2.1: タスク状態遷移図

T-Kernelの特色として、自タスクの操作をするシステムコールと他タスクの操作をするシステムコールを明確に分離しているということがある[表 2.1.「自タスク、他タスクの区別と状態遷移図」]。これは、タスクの状態遷移を明確にし、システムコールの理解を容易にするためである。自タスク操作と他タスク操作のシステムコールを分離しているということは、言い換えると、実行状態からの状態遷移とそれ以外の状態からの状態遷移を明確に分離しているという意味にもなる。

	自タスクに対する操作 (実行状態からの遷移)	他タスクに対する操作 (実行状態以外からの遷移)
タスクの待ち状態(強制待ち状態を含む)への移行	tk_slp_tsk 実行状態 → 待ち状態	tk_sus_tsk 実行可能、待ち状態 → 強制待ち、二重待ち状態
タスクの終了	tk_ext_tsk 実行状態 → 休止状態	tk_ter_tsk 実行可能、待ち状態 → 休止状態
タスクの削除	tk_exd_tsk 実行状態 → 未登録状態	tk_del_tsk 休止状態 → 未登録状態

表 2.1: 自タスク、他タスクの区別と状態遷移図

補足事項
待ち状態と強制待ち状態は直交関係にあり、強制待ち状態への移行の要求は、タスクの待ち解除条件には影響を与えない。言い換えると、タスクが待ち状態にあるか二重待ち状態にあるかで、タスクの待ち解除条件は変化しない。そのため、資源獲得のための待ち状態(セマフォ資源の獲得待ち状態やメモリブロックの獲得待ち状態など)にあるタスクに強制待ち状態への移行が要求され、二重待ち状態になった場合にも、強制待ち状態への移行が要求されなかった場合と同じ条件で資源の割付け(セマフォ資源やメモリブロックの割付けなど)が行われる。

仕様決定の理由
T-Kernelにおいて待ち状態(自タスクによる待ち)と強制待ち状態(他のタスクによる待ち)を区別しているのは、それらが重なる場合があるためである。それらが重なった状態を二重待ち状態として区別することで、タスクの状態遷移が明確になり、システムコールの理解が容易になる。それに対して、待ち状態のタスクはシステムコールを呼び出せないため、複数の種類の待ち状態(例えば、起床待ち状態とセマフォ資源の獲得待ち状態)が重なることはない。T-Kernelでは、他のタスクによる待ちには一つの種類(強制待ち状態)しかないため、強制待ち状態が重なった状況を強制待ち要求のネストと扱うことで、タスクの状態遷移を明確にしている。

2.2.2 タスクのスケジューリング規則

T-Kernelにおいては、タスクに与えられた優先度に基づくプリエンプティブな優先度ベーススケジューリング方式を採用している。同じ優先度を持つタスク間では、FCFS(First Come First Served)方式によりスケジューリングを行う。具体的には、タスクのスケジューリング規則はタスク間の優先順位を用いて、タスク間の優先順位はタスクの優先度によって、それぞれ次のように規定される。実行できるタスクが複数ある場合には、その中で最も優先順位の高いタスクが実行状態となり、他は実行可能状態となる。タスク間の優先順位は、異なる優先度を持つタスク間では、高い優先度を持つタスクの方が高い優先順位を持つ。同じ優先度を持つタスク間では、先に実行できる状態(実行状態または実行可能状態)になったタスクの方が高い優先順位を持つ。ただし、システムコールの呼出により、同じ優先度を持つタスク間の優先順位が変更される場合がある。

最も高い優先順位を持つタスクが替わった場合には、ただちにディスパッチが起こり、実行状態のタスクが切り替わる。ただし、ディスパッチが起こらない状態(ハンドラ実行中やディスパッチ禁止状態など)になっている場合には、実行状態のタスクの切替えは、ディスパッチが起こる状態となるまで保留される。

補足事項
T-Kernelのスケジューリング規則では、優先順位の高いタスクが実行できる状態にある限り、それより優先順位の低いタスクは全く実行されない。すなわち、最も高い優先順位を持つタスクが待ち状態に入るなどの理由で実行できない状態とならない限り、他のタスクは全く実行されない。この点で、複数のタスクを公平に実行しようというTSS(Time Sharing System)のスケジューリング方式とは根本的に異なっている。
ただし、同じ優先度を持つタスク間の優先順位は、システムコールを用いて変更することが可能である。アプリケーションがそのようなシステムコールを用いて、TSSにおける代表的なスケジューリング方式であるラウンドロビン方式を実現することができる。
同じ優先度を持つタスク間では、先に実行できる状態(実行状態または実行可能状態)になったタスクの方が高い優先順位を持つことを、図の例を用いて説明する。[図 2.2.「最初の状態の優先順位」]は、優先度1のタスクA、優先度3のタスクE、優先度2のタスクB、タスクC、タスクDがこの順序で起動された後のタスク間の優先順位を示す。この状態では、最も優先順位の高いタスクAが実行状態となっている。
ここでタスクAが終了すると、次に優先順位の高いタスクBを実行状態に遷移させる[図 2.3.「タスクBが実行状態になった後の優先順位」]。その後タスク Aが再び起動されると、タスクBはプリエンプトされて実行可能状態に戻るが、この時タスクBは、タスクCとタスクDのいずれよりも先に実行できる状態になっていたことから、同じ優先度を持つタスクの中で最高の優先順位を持つことになる。すなわち、タスク間の優先順位は[図 2.2.「最初の状態の優先順位」]の状態に戻る。
次に、[図 2.3.「タスクBが実行状態になった後の優先順位」]の状態でタスクBが待ち状態になった場合を考える。タスクの優先順位は実行できるタスクの間で定義されるため、タスク間の優先順位は[図 2.4.「タスクBが待ち状態になった後の優先順位」]の状態となる。その後タスクBが待ち解除されると、タスクBはタスクCとタスクDのいずれよりも後に実行できる状態になったことから、同じ優先度を持つタスクの中で最低の優先順位となる[図 2.5.「タスクBが待ち解除された後の優先順位」]。
以上を整理すると、実行可能状態のタスクが実行状態になった後に実行可能状態に戻った直後には、同じ優先度を持つタスクの中で最高の優先順位を持っているのに対して、実行状態のタスクが待ち状態になった後に待ち解除されて実行できる状態になった直後には、同じ優先度を持つタスクの中で最低の優先順位となる。
なお、タスクが強制待ち状態(SUSPENDED)から実行できる状態になった直後にも、同じ優先度を持つタスクの中で最低の優先順位となる。仮想記憶システムにおいては、ページイン待ちを強制待ち(SUSPENDED)によって行うため、このようなシステムにおいては、ページイン待ちによってタスクの優先順位が変化する。

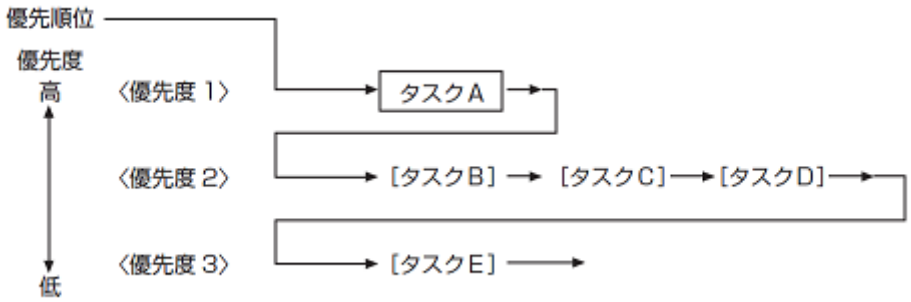

図 2.2: 最初の状態の優先順位

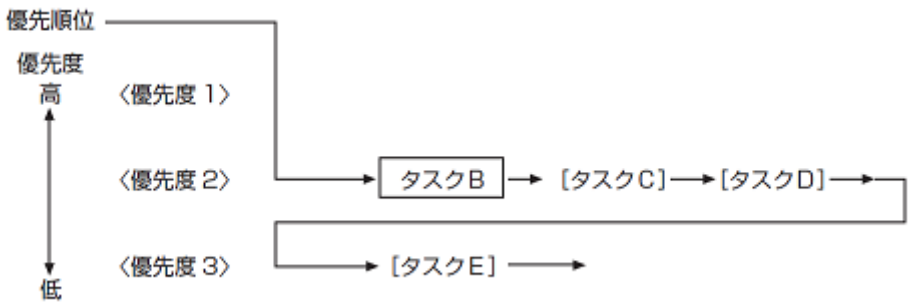

図 2.3: タスクBが実行状態になった後の優先順位

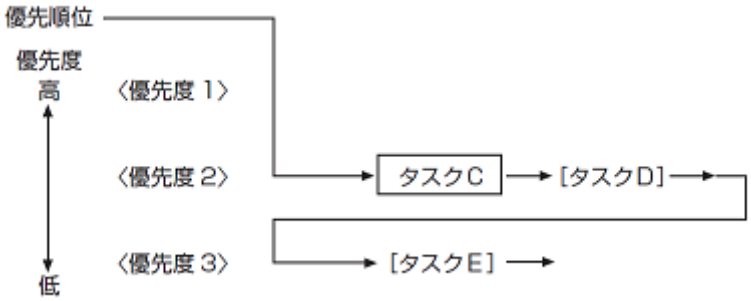

図 2.4: タスクBが待ち状態になった後の優先順位

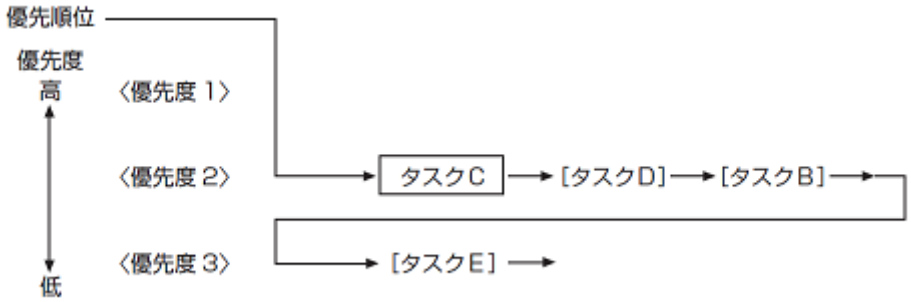

図 2.5: タスクBが待ち解除された後の優先順位

2.3 割込み処理

T-Kernelでの割込みは、デバイスからの外部割込みと、CPU例外による割込みの両方を含む。一つの割込みハンドラ番号に対して一つの割込みハンドラを定義できる。割込みハンドラの記述方法としては、基本的にカーネルが介入せずに直接起動する方法と、高級言語対応ルーチンを経由して起動する方法の2通りがある。

詳細は「4.8節「割込み管理機能」」を参照のこと。

2.4 タスク例外処理

T-Kernelでは、例外処理のための機能として、タスク例外処理の機能を規定する。なおCPU例外については、割込みとして扱うこととする。

タスク例外処理機能は、タスクを指定してタスク例外処理を要求するシステムコールを呼び出すことで、指定したタスクに実行中の処理を中断させ、タスク例外ハンドラを実行させるための機能である。タスク例外ハンドラの実行は、タスクと同じコンテキストで行われる。タスク例外ハンドラからリターンすると、中断された処理の実行が継続される。

タスク毎に一つのタスク例外ハンドラを、アプリケーションで登録することができる。

詳細は「4.3節 「タスク例外処理機能」」を参照のこと。

2.5 システム状態

2.5.1 非タスク部実行中のシステム状態

T-Kernelの上で動くタスクのプログラミングを行う場合には、タスク状態遷移図を見て、各タスクの状態の変化を追っていけばよい。しかし、割込みハンドラや拡張SVCハンドラなど、タスクよりカーネルに近いレベルのプログラミングもユーザが行う。この場合は、非タスク部、すなわちタスク以外の部分を実行している間のシステム状態についても考慮しておかないと、正しいプログラミングができない。ここでは、T-Kernelのシステム状態について説明を行う。

システム状態は、[図 2.6.「システム状態の分類」]のように分類される。

[図 2.6.「システム状態の分類」]で示されている状態のうち、「過渡的な状態」は、カーネル実行中(システムコール実行中)の状態に相当する。ユーザから見ると、ユーザのアプリケーションプログラムから発行したそれぞれのシステムコールが不可分に実行されるということが重要なのであり、システムコール実行中の内部状態はユーザからは見えない。カーネル実行中の状態を「過渡的な状態」と考え、その内部をブラックボックス的に扱うのは、こういった理由による。

しかし、次の場合、過渡的な状態が不可分に実行されない。

- メモリの獲得・解放を伴うシステムコールで、メモリの獲得・解放を行っている間。(T-Kernel/SMのシステムメモリ管理機能を呼び出している間)
- 仮想記憶システムにおいて、システムコールの処理中に非常駐メモリのアクセスを行った場合。

このような、過渡的な状態にあるタスクに対して、タスクの強制終了(tk_ter_tsk)を行った場合の動作は保証されない。また、タスクの強制待ち(tk_sus_tsk)も過渡的な状態のまま停止することになり、それによりデッドロック等を引き起こす可能性がある。

したがって、tk_ter_tsk, tk_sus_tsk は原則として使用できない。これらを使用するのは、仮想記憶システムやデバッガのような、OSの一部といえるようなサブシステム内のみとするべきである。

「非タスク部」ではあるが、特定のタスク(「要求タスク」と呼ぶ)から依頼された処理を実行していると見なされる部分を「準タスク部」と呼ぶ。たとえば、ユーザの定義したサブシステムの拡張SVCハンドラは、「準タスク部」として実行される。「準タスク部」の中では自タスクを特定でき、要求タスクが自タスクとなる。また、タスク部と同じようにタスクの状態遷移を定義することができ、準タスク部から待ち状態に入るシステムコールも発行可能である。これらの点において、準タスク部は、要求タスクから呼び出されたサブルーチンと同じような振る舞いをする。ただし、「準タスク部」はOS拡張部の位置付けであり、プロセッサの動作モードやスタック空間はタスク部と異なる。すなわち、タスク部から準タスク部に入る際には、プロセッサの動作モードやスタック空間の切り替えが起こる。この点は、タスク部の中で関数やサブルーチンを呼び出す場合とは異なっている。

一方、「非タスク部」のうち、タスク部や準タスク部の処理の進行とは全く別の要因で動作を始めるのが「タスク独立部」である。具体的には、外部割込みによって起動される割込みハンドラや、指定時間の経過によって起動されるタイムイベントハンドラ(周期ハンドラおよびアラームハンドラ)などが「タスク独立部」として実行される。外部割込みも、指定時間の経過も、その時点でたまたま実行中だったタスクとは無関係の要因である点に注意されたい。

結局、「非タスク部」は、「過渡的な状態」「準タスク部」「タスク独立部」の3つに分類できる。これ以外の状態が、タスクのプログラムを実行している状態、すなわち「タスク部実行中」の状態である。

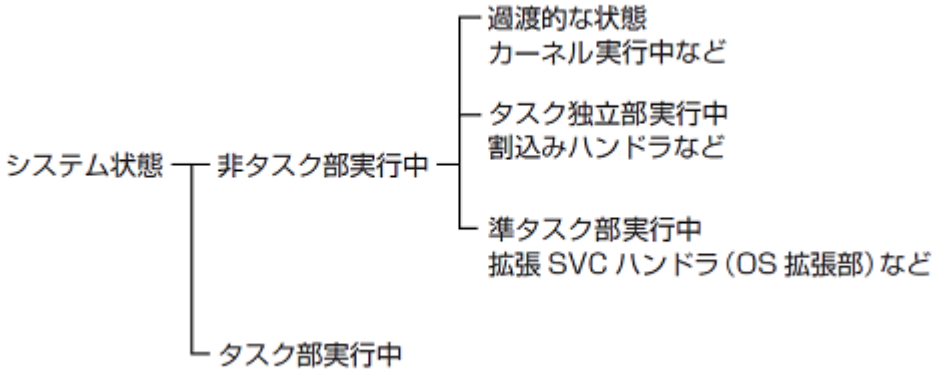

図 2.6: システム状態の分類

2.5.2 タスク独立部と準タスク部

タスク独立部(割込みハンドラやタイムイベントハンドラ)の特徴は、タスク独立部に入る直前に実行中だったタスクを特定することが無意味であり、「自タスク」の概念が存在しないことである。したがって、タスク独立部からは、待ち状態に入るシステムコールや、暗黙で自タスクを指定するシステムコールを発行することはできない。また、タスク独立部では現在実行中のタスクが特定できないので、タスクの切り換え(ディスパッチ)は起らない。ディスパッチが必要になっても、それはタスク独立部を抜けるまで遅らされる。これを遅延ディスパッチ(delayed dispatching)の原則と呼ぶ。

もし、タスク独立部である割込みハンドラの中でディスパッチを行うと、割込みハンドラの残りの部分の実行が、そこで起動されるタスクよりも後回しになるため、割込みがネストしたような場合に問題が起こる。この様子を[図 2.7.「割込みのネストと遅延ディスパッチ」]に示す。

[図 2.7.「割込みのネストと遅延ディスパッチ」]は、タスクAの実行中に割込みXが発生し、その割込みハンドラ中でさらに高優先度の割込みYが発生した状態を示している。この場合、(1)[1]の割込みYからのリターン時に即座にディスパッチを起こしてタスクBを起動すると、割込みXの(2)〜(3)の部分の実行がタスクBよりも後回しになり、タスクAが実行状態になった時にはじめて(2)〜(3)が実行されることになる。これでは、低優先度の割込みXのハンドラが、高優先度の割込みばかりではなく、それによって起動されたタスクBにもプリエンプトされる危険を持つことになる。したがって、割込みハンドラがタスクに優先して実行されるという保証がなくなり、割込みハンドラが書けなくなってしまう。遅延ディスパッチの原則を設けているのは、こういった理由による。

それに対して、準タスク部の特徴は、準タスク部に入る直前に実行中だったタスク(要求タスク)を特定することが可能であること、タスク部と同じようにタスクの状態が定義されており、準タスク部の中で待ち状態に入ることも可能なことである。したがって、準タスク部の中では、通常のタスク実行中の状態と同じようにディスパッチングが起きる。その結果、OS拡張部などの準タスク部は、非タスク部であるにもかかわらず、常にタスク部に優先して実行されるとは限らない。これは、割込みハンドラがすべてのタスクに優先して実行されるのとは対照的である。

次の二つの例は、タスク独立部と準タスク部の違いを示すものである。

- タスクA(優先度8＝低)の実行中に割込みがかかり、その割込みハンドラ(タスク独立部である)の中でタスクB(優先度2＝高)に対する tk_wup_tsk が実行された。しかし、遅延ディスパッチの原則により、ここではまだディスパッチが起きず、tk_wup_tsk 実行後はまず割込みハンドラの残りの部分が実行される。割込みハンドラの後の tk_ret_int によって、はじめてディスパッチングが起り、タスクBが実行される。
- タスクA(優先度8＝低)の中で拡張SVCが実行され、その拡張SVCハンドラ(準タスク部とする)の中のタスクB(優先度2＝高)に対する tk_wup_tsk が実行された。この場合は遅延ディスパッチの原則が適用されないので、tk_wup_tsk の中でディスパッチングが行われ、タスクAが準タスク部内での実行可能状態に、タスクBが実行状態になる。したがって、拡張SVCハンドラの残りの部分よりもタスクBの方が先に実行される。拡張SVCハンドラの残りの部分は、再びディスパッチングが起ってタスクAが実行状態となった後で実行される。

[1] (1) でディスパッチを行うと、割込みXのハンドラの残りの部分((2)〜(3))の実行が後回しになってしまう。

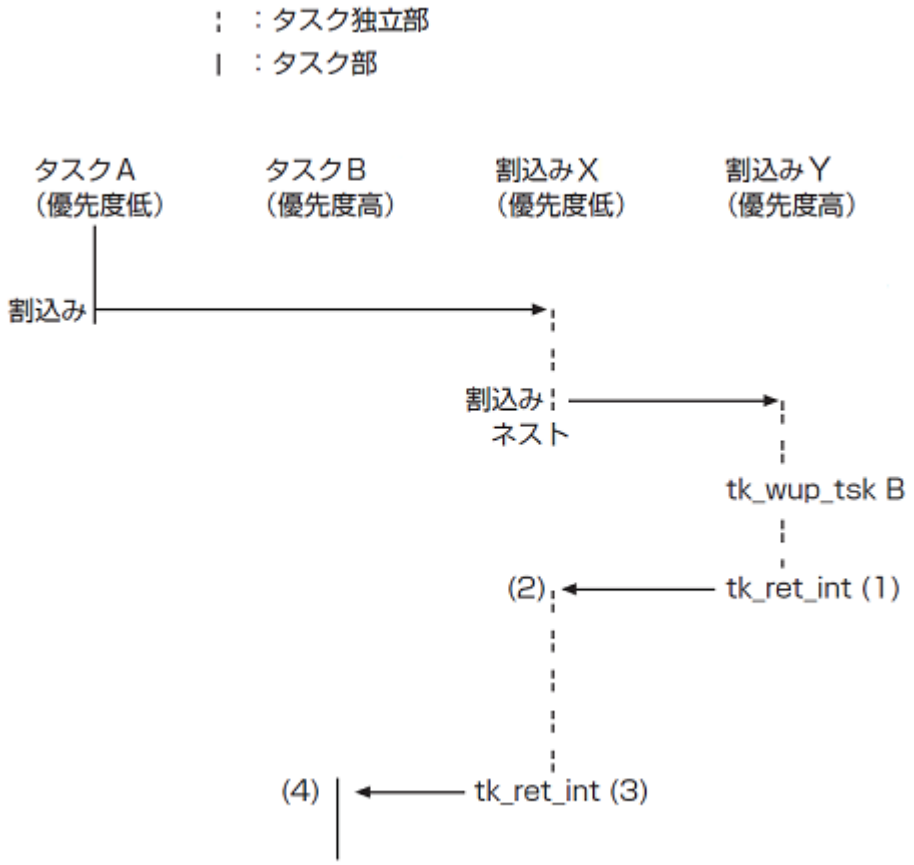

図 2.7: 割込みのネストと遅延ディスパッチ

2.6 オブジェクト

カーネルが操作対象とする資源をオブジェクトと総称する。オブジェクトには、タスクのほかに、メモリプールや、セマフォ、イベントフラグ、メールボックスなどの同期・通信機構、およびタイムイベントハンドラ(周期ハンドラおよびアラームハンドラ)が含まれている。

オブジェクト生成時には、原則として属性を指定することができる。属性はオブジェクトの細かな動作の違いやオブジェクトの初期状態を定める。属性にTA_XXXXXが指定されている場合、そのオブジェクトを「TA_XXXXX属性のオブジェクト」と呼ぶ。特に指定すべき属性がない場合には、TA_NULL(＝0)を指定する。オブジェクト登録後に属性を読み出すインタフェースは、一般には用意されない。

オブジェクトの属性は、下位側がシステム属性を表し、上位側が実装独自属性を表す。どのビット位置を境に上位側／下位側と区別するかは特に定めない。基本的には、標準仕様で定義されていないビットは実装独自属性として使用できる。ただし、原則としてシステム属性はLSB(最下位ビット)から MSB(最上位ビット)に向かって順に割り当てる。実装独自属性ではMSBからLSBに向かって割り当てるものとする。属性の未定義のビットは0クリアされていなければならない。

オブジェクトは拡張情報を持つ場合がある。拡張情報はオブジェクト登録時に指定し、オブジェクトが実行を始める時にパラメータとして渡される情報で、カーネルの動作には影響を与えない。拡張情報はオブジェクトの状態参照システムコールで読み出すことができる。

オブジェクトはID番号により識別される。T-Kernelでは、ID番号はオブジェクトの生成時に自動的に割り当てられる。利用者がID番号を指定することはできない。このため、デバッグの際にオブジェクトを識別することが困難となる。そこで、各オブジェクトの生成の際に、デバッグ用のオブジェクト名称を指定することができる。この名称はあくまでデバッグ用であり、T-Kernel/DSの機能からのみ参照できる。また、名称に関するチェックは、T-Kernelでは一切行われない。

2.7 メモリ

2.7.1 アドレス空間

メモリのアドレス空間は、共有空間とタスク固有空間に区別される。共有空間は、すべてのタスクから同じようにアクセスできる空間で、タスク固有空間はそのタスク固有空間に属しているタスクからのみアクセスできる[図 2.8.「アドレス空間」]。1つのタスク固有空間に複数のタスクが属している場合もある。

タスク固有空間と共有空間の論理アドレス空間は、CPU(およびMMU)の制限に依存するため実装依存となるが、タスク固有空間を低位アドレス、共有空間を高位アドレスにおくことを原則とする。

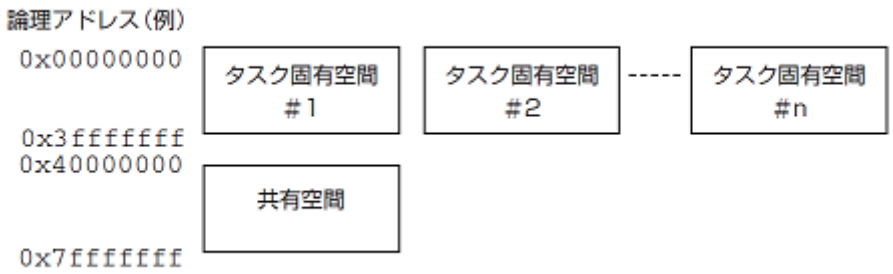

図 2.8: アドレス空間

割込みハンドラなどのタスク独立部は、タスクでないためタスク固有空間を所有しない。そのため、タスク独立部では、タスク独立部に入る直前に実行していたタスクのタスク固有空間に属するものとする。これは、tk_get_tid で返される現在実行状態のタスクのタスク固有空間と一致する。実行状態のタスクがない時は、タスク固有空間は不定である。

共有空間およびタスク固有空間に関しては、tk_cre_tsk、メモリプール、システムメモリ管理機能にも関連する説明がある。

MMUのない(またはMMUを使用しない)システムにおいては、タスク固有空間が存在しないものと考える。

2.7.2 非常駐メモリ

メモリには常駐メモリと非常駐メモリがある。

非常駐メモリは、そのメモリをアクセスすることにより、ディスク等からメモリへデータの転送などが行われる。そのため、デバイスドライバによるディスクアクセスなどの複雑な処理が必要になる。したがって、非常駐メモリをアクセスする時には、デバイスドライバなどが動作できる状況でなければならない。そのため、ディスパッチ禁止中や割込み禁止中、タスク独立部実行中などの状況でアクセスすることはできない。

同様に、OS内部の処理においても、クリティカルセクション内で非常駐メモリにアクセスしないようにする必要がある。特に、システムコールのパラメータとして渡されるメモリアドレスが、非常駐メモリを指している場合が考えられる。システムコールのパラメータが非常駐メモリにあることを許すか否かは、実装依存とする。

非常駐メモリのアクセスによるディスク等からの転送は、T-Kernelでは行わない。通常は、仮想記憶管理などを行うサブシステムがT-Kernelに組み合わされる。

仮想記憶を使用しないシステムにおいて、システムコール等で非常駐メモリが指定された場合は、単に無視してすべて常駐メモリとして扱えばよい。

2.7.3 保護レベル

T-Kernelでは、0～3の4段階の保護レベルを仮定する。

- 0が最も高い特権を持ち、3が最も低い。
- 現在実行中の保護レベルと同じか低い保護レベルのメモリにのみアクセスできる。
- 保護レベルの遷移は、システムコールまたは拡張SVCの呼出、および割込み・CPU例外により行われる。
- 現在実行中のタスク等の保護レベルが、アクセス対象となったメモリの保護レベルよりも低い場合には、一般にはMMUがメモリアクセス権の違反を検出し、CPU例外を発生する。

各保護レベルは次のような用途に使用する。

保護レベル	用途
0	カーネル、サブシステム、デバイスドライバなど
1	システムアプリケーションタスク
2	(予約)
3	ユーザアプリケーションタスク

非タスク部(タスク独立部、準タスク部など)は保護レベル0で実行する。保護レベル1～3で実行できるのはタスク部のみである。保護レベル0でもタスク部は実行可能である。

MMUによっては、特権とユーザの2段階の保護レベルしかない場合もある。そのような場合は、保護レベル0～2を特権、保護レベル3をユーザとして4段階に見立てて扱う。MMUのないシステム、またはMMUを使用しないシステムでは、保護レベル0～3をすべて同一に扱えばよい。

第 3 章

T-Kernel共通規定

3.1 データ型

3.1.1 汎用的なデータ型

```
typedef signed char          B;      /* 符号付き  8ビット整数 */
typedef signed short         H;      /* 符号付き 16ビット整数 */
typedef signed long          W;      /* 符号付き 32ビット整数 */
typedef signed long long     D;      /* 符号付き 64ビット整数 */
typedef unsigned char       UB;      /* 符号無し  8ビット整数 */
typedef unsigned short      UH;      /* 符号無し 16ビット整数 */
typedef unsigned long       UW;      /* 符号無し 32ビット整数 */
typedef unsigned long long  UD;      /* 符号無し 64ビット整数 */

typedef char                VB;      /* 型が一定しない  8ビットのデータ */
typedef short               VH;      /* 型が一定しない 16ビットのデータ */
typedef long                VW;      /* 型が一定しない 32ビットのデータ */
typedef long long           VD;      /* 型が一定しない 64ビットのデータ */
typedef void                *VP;     /* 型が一定しないデータへのポインタ */

typedef volatile B          _B;      /* volatile 宣言付 */
typedef volatile H          _H;
typedef volatile W          _W;
typedef volatile D          _D;
typedef volatile UB         _UB;
typedef volatile UH         _UH;
typedef volatile UW         _UW;
typedef volatile UD         _UD;

typedef signed int          INT;     /* プロセッサのビット幅の符号付き整数、32ビット以上 */
typedef unsigned int        UINT;    /* プロセッサのビット幅の符号無し整数、32ビット以上 */

typedef INT                 ID;      /* ID一般 */
typedef W                   MSEC;    /* 時間一般(ミリ秒) */

typedef void            (*FP)();     /* 関数アドレス一般 */
typedef INT             (*FUNCP)(); /* 関数アドレス一般 */

#define LOCAL           static       /* ローカルシンボル定義 */
#define EXPORT                       /* グローバルシンボル定義 */
#define IMPORT          extern       /* グローバルシンボル参照 */

/*
 * ブール値
 *      TRUE = 1 と定義するが、0 以外はすべて真(TRUE)である。
 *      したがって、bool == TRUE の様な判定をしてはいけない。
 *      bool != FALSE の様に判定すること。
 */
typedef UINT            BOOL;
#define TRUE            1            /* 真 */
#define FALSE           0            /* 偽 */

/*
 * TRON 文字コード
 */
typedef UH              TC;          /* TRON 文字コード */
#define TNULL           ((TC)0)      /* TRON コード文字列の終端 */
```

注意

- VB, VH, VW, VDと B, H, W, Dとの違いは、前者はビット数のみが分かっており、データ型の中身が分からないものを表すのに対して、後者は整数を表すことがはっきりしているという点である。
- プロセッサのビット幅は32ビット以上に限定する。したがって、INT, UINTは必ず32ビット以上の幅がある。
- BOOLは、TRUE を1と定義するが、0以外はすべて真である。したがって、ブール値 == TRUE の様な判定を行ってはいけない。ブール値 != FALSE の様に判定する。

補足事項

stksz, wupcnt, メッセージサイズなど、明らかに負の数にならないパラメータも、原則として符号付き整数(INT)のデータ型となっている。これは、整数はできるだけ符号付きの数として扱うというTRON全般のルールに基づいたものである。また、タイムアウト(TMO tmout)のパラメータでは、これが符号付きの整数であることを利用し、TMO_FEVR(= -1) を特殊な意味に使っている。符号無しのデータ型を持つパラメータは、ビットパターンとして扱われるもの(オブジェクト属性やイベントフラグなど)である。

T-Kernel 1.0との差異

- 64ビットのD, UDを追加した。'D'は Double integer の意味である。また、符号付き整数の宣言にsignedを追加した。W, UWが32ビットであることを明確にするため、intからlongに変更した。
- T-Kernel 1.0のMSECはINT(プロセッサのビット幅の整数)だったが、T-Kernel 2.0のMSECはW(32ビット固定)に変更した。これは、μT-Kernelの仕様書からフィードバックする形で、INTが16ビットになった場合に不都合な部分をW(32ビット固定)としたことによる。
- T-Kernel 1.0では exinf などのデータタイプをVPとしていたが、T-Kernel 2.0では、CONST修飾子との関係から、原則としてVPは使わず、VPの代わりに"void *"のデータタイプ定義を直接書くことにした。T-Kernel 2.0でも、互換性維持のためにVP自体の定義は残しているが、VPの新規の使用は推奨しない。

3.1.2 意味が定義されているデータ型

パラメータの意味を明確にするため、出現頻度の高いデータ型や特殊な意味を持つデータ型に対して、以下のような名称を使用する。

```
typedef INT             FN;             /* 機能コード */
typedef INT             RNO;            /* ランデブ番号 */
typedef UW              ATR;            /* オブジェクト/ハンドラ属性 */
typedef INT             ER;             /* エラーコード */
typedef INT             PRI;            /* 優先度 */
typedef W               TMO;            /* ミリ秒単位のタイムアウト指定 */
typedef D               TMO_U;          /* 64ビットでマイクロ秒単位のタイムアウト指定 */
typedef UW              RELTIM;         /* ミリ秒単位の相対時間 */
typedef UD              RELTIM_U;       /* 64ビットでマイクロ秒単位の相対時間 */

typedef struct systim {                 /* ミリ秒単位のシステム時刻 */
        W       hi;                     /* 上位32ビット */
        UW      lo;                     /* 下位32ビット */
} SYSTIM;

typedef D               SYSTIM_U;       /* 64ビットでマイクロ秒単位のシステム時刻 */

/*
 * 共通定数
```

```
 */
#define NULL            0               /* 無効ポインタ */
#define TA_NULL         0               /* 特別な属性を指定しない */
#define TMO_POL         0               /* ポーリング */
#define TMO_FEVR        (-1)            /* 永久待ち */
```

注意

- 複数のデータ型を複合したデータ型の場合は、その内の最も主となるデータ型で代表する。例えば、tk_cre_tsk の戻値はタスクIDかエラーコードであるが、主となるのはタスクIDなので、データ型はIDとなる。

T-Kernel 1.0との差異

64ビットでマイクロ秒単位のタイムアウト指定を示すTMO_U、64ビットでマイクロ秒単位の相対時間を示すRELTIM_U、64ビットでマイクロ秒単位のシステム時刻を示すSYSTIM_Uを追加した。RELTIM_UはRELTIMに合わせて符号無し、SYSTIM_UはSYSTIMに合わせて符号付きである。また、SYSTIMは32ビットのメンバを2つ合わせた構造体であるが、SYSTIM_Uは64ビットデータの利便性を直接活かす形で、構造体ではない通常の64ビット整数となっている。
ミリ秒単位のタイムアウト指定を示すTMOは、T-Kernel 1.0ではINTだったが、T-Kernel 2.0ではWに変更した。また、オブジェクト属性などを示すATRおよびミリ秒単位の相対時間を示すRELTIMは、T-Kernel 1.0ではUINTだったが、T-Kernel 2.0ではUWに変更した。

補足事項

マイクロ秒(μsec)を意味するパラメータやデータタイプには最後に"_u"(uはμの意味) または"_U"を付け、それ以外の64ビット整数を意味するパラメータやデータタイプには最後に"_d"(dは double integer の意味)または"_D"を付ける方針としている。TMO_U, RELTIM_U, SYSTIM_Uはこの方針によるデータタイプ名である。

3.2 システムコール

3.2.1 システムコールの形式

T-Kernelでは、C言語を標準的な高級言語として使用することにしており、C言語からシステムコールを実行する場合のインタフェース方法を標準化している。

一方、アセンブリ言語のインタフェース方法は実装定義とする。アセンブリ言語でプログラムを作成する場合でも、C言語のインタフェースを利用して呼び出す方法を推奨する。これにより、アセンブリ言語で作成したプログラムも同一CPUであればOSが変わっても移植性が確保できる。

システムコールインタフェースでは、次のような共通原則を設けている。

- システムコールはすべてC言語の関数として定義される。
- 関数としての戻値は、0または正の値が正常終了、負の値がエラーコードとなる。

システムコールインタフェースの処理部分(システムコールを表す関数の中から、実際にT-Kernelの機能を呼び出す部分)は、アセンブリ言語で書かれたライブラリとして実装される。これをインタフェースライブラリと呼ぶ。移植性への配慮から、インタフェースライブラリの実装には、C言語のマクロやインライン関数、インラインアセンブラなどを用いない。

システムコールのC言語インタフェースのうち、パケットやポインタを使ってパラメータを渡すものについては、T-Kernelがポインタ参照先のパラメータを書き換えないことを明示するために、CONSTという修飾子を付けている。

CONSTは、C言語の const 修飾子を意図したものであるが、const 修飾子に未対応のプログラムが混在した場合に、#defineのマクロ機能を使ってコンパイラのチェックを無効化できるように、const と類似した別名を使っている。

具体的なCONSTの使い方は以下のようになる。なお、詳細は開発環境に依存する。

1. 共通のインクルードファイルに次の記述を含める。

```
/* TKERNEL_CHECK_CONST の定義がある場合に const のチェックを有効化 */
#ifdef TKERNEL_CHECK_CONST
#define CONST const
#else
#define CONST
#endif
```

2. プログラム中の関数定義やシステムコールの定義は、CONSTを用いて記述する。

例 3.1 CONSTの記述例

```
tk_cre_tsk( CONST T_CTSK *pk_ctsk );
foo_bar( CONST void *buf );
```

3. Makefile 内の指定で const を有効化する。(推奨)

例 3.2 const 有効化の例

```
CFLAGS += -DTKERNEL_CHECK_CONST
```

※ 上記の指定がない場合は、const のチェックが無効になる。

T-Kernel 2.0以降では、プログラム中にCONSTを明示し、const のチェックを有効にして運用することを強く推奨する。

T-Kernel 1.0との差異
システムコールのC言語インタフェースにCONSTを追加し、const 修飾子によるチェックを推奨することとした。ただし、const 修飾子に未対応のプログラムに対する回避策も設けた。

3.2.2 タスク独立部から発行できるシステムコール

次のシステムコールは、タスク独立部およびディスパッチ禁止状態から発行できる。

システムコール名称	説明
tk_sta_tsk	タスク起動
tk_wup_tsk	他タスクの起床
tk_rel_wai	他タスクの待ち状態解除
tk_sus_tsk	他タスクを強制待ち状態へ移行
tk_sig_sem	セマフォ資源返却
tk_set_flg	イベントフラグのセット
tk_sig_tev	タスクイベントの送信
tk_rot_rdq	タスクの優先順位の回転
tk_get_tid	実行状態タスクのタスクID参照
tk_sta_cyc	周期ハンドラの動作開始
tk_stp_cyc	周期ハンドラの動作停止
tk_sta_alm	アラームハンドラの動作開始
tk_sta_alm_u	アラームハンドラの動作開始(マイクロ秒単位)
tk_stp_alm	アラームハンドラの動作停止
tk_ref_tsk	タスク状態参照
tk_ref_tsk_u	タスク状態参照(マイクロ秒単位)
tk_ref_cyc	周期ハンドラ状態参照
tk_ref_cyc_u	周期ハンドラ状態参照(マイクロ秒単位)
tk_ref_alm	アラームハンドラ状態参照
tk_ref_alm_u	アラームハンドラ状態参照(マイクロ秒単位)
tk_ref_sys	システム状態参照
tk_ret_int	割込みハンドラから復帰(アセンブリ言語で記述された割込みハンドラからのみ発行可能)

上記以外のシステムコールがタスク独立部およびディスパッチ禁止状態から発行できるか否かは実装依存である。

3.2.3 システムコールの呼出制限

システムコールを呼び出すことのできる保護レベルを制限することができる。この場合、指定した保護レベルより低い保護レベルで動作しているタスク(タスク部)からシステムコールを発行した場合に、E_OACV エラーとする。

拡張SVCの呼出は制限されない。

例えば、保護レベル1より低い保護レベルからのシステムコールの呼出を禁止した場合、保護レベル2,3で実行しているタスクからはシステムコールは発行できなくなる。つまり、保護レベル2,3で動作しているタスクは、拡張SVCしか発行できないということになり、サブシステムの機能のみを使ってプログラムすることになる。

これは、T-KernelをT-Kernel Extensionと組み合わせる場合、T-Kernel Extensionの仕様に基づくタスクからT-Kernelの機能を直接操作させないために使用する。T-Kernelをマイクロカーネルとして使用するための機能である。

システムコール呼出制限をする保護レベルは、システム構成情報管理機能により設定する。システム構成情報管理機能については、5.7節「システム構成情報管理機能」を参照のこと。

3.2.4 パラメータパケット形式の変更

システムコールへ渡すパラメータのいくつかは、パケット形式になっている。このパケット形式のパラメータには、システムコールへ情報を渡す入力パラメータとなるもの(T_CTSKなど)とシステムコールから情報を返される出力パラメータとなるもの(T_RTSKなど)がある。

これらパケット形式のパラメータには、実装独自の情報を追加することができる。ただし、標準仕様で定義されているデータの型および順序を変更してはならず、削除してもいけない。実装独自の情報を追加する場合は、標準仕様で定義されているものの後ろに追加しなければならない。

システムコールへの入力パラメータとなるパケット(T_CTSKなど)では、実装独自で追加された情報が未初期化(メモリの内容が不定)のまま、システムコールを呼び出した場合でも正常に動作するようにしなければならない。

通常は、追加した情報に有効な値が入っていることを示すフラグを標準仕様に含まれている属性フラグの実装独自領域に定義する。そのフラグがセットされている(1)場合にのみ追加した情報を使用し、セットされていない(0)場合にはその追加情報は未初期化(メモリの内容が不定)であるとして、デフォルト値を適用する。

これは、標準仕様の範囲内で作成されたプログラムを、再コンパイルのみで実装独自の機能拡張を行われたOS上で動作させるための規定である。

3.2.5 機能コード

機能コードは、システムコールを識別するために、各システムコールに割り付けられる番号である。

システムコールの機能コードは特に定めない。すべて実装定義とする。

拡張SVCの機能コードについては、tk_def_ssy を参照。

3.2.6 エラーコード

システムコールの返値は原則として符号付きの整数で、エラーが発生した場合には負の値のエラーコード、処理を正常に終了した場合は E_OK(＝0)または正の値とする。正常終了した場合の返値の意味はシステムコール毎に規定する。この原則の例外として、呼び出されるとリターンすることのないシステムコールがある。リターンすることのないシステムコールは、C言語インタフェースでは返値を持たないもの(すなわちvoid型の関数)として宣言する。

エラーコードは、メインエラーコードとサブエラーコードで構成される。エラーコードの下位16ビットがサブエラーコード、残りの上位ビットがメインエラーコードとなる。メインエラーコードは、検出の必要性や発生状況などにより、エラークラスに分類される。T-Kernel/OSではサブエラーコードは使用せず、常に0となる。

```
#define MERCD(er)        ( (ER)(er) >> 16 )     /* メインエラーコード */
#define SERCD(er)        ( (H)(er) )            /* サブエラーコード */
#define ERCD(mer, ser)   ( (ER)(mer) << 16 | (ER)(UH)(ser) )
```

3.2.7 タイムアウト

待ち状態に入る可能性のあるシステムコールには、タイムアウトの機能を持たせる。タイムアウトは、指定された時間が経過しても処理が完了しない場合に、処理をキャンセルしてシステムコールからリターンするものである(この時、システムコールは E_TMOUT エラーを返す)。

そのため、「システムコールがエラーコードを返した場合には、システムコールを呼び出したことによる副作用はない」という原則より、タイムアウトした場合には、システムコールを呼び出したことで、システムの状態は変化していないのが原則である。ただし、システムコールの機能上、処理のキャンセル時に元の状態に戻せない場合は例外とし、システムコールの説明でその旨を明示する。

タイムアウト時間を0に設定すると、システムコールの中で待ち状態に入るべき状況になっても、待ち状態には入らない。そのため、タイムアウト時間を0としたシステムコール呼出では、待ち状態に入る可能性がない。タイムアウト時間を0としたシステムコール呼出を、ポーリングと呼ぶ。すなわち、ポーリングを行うシステムコールでは、待ち状態に入る可能性がない。

各システムコールの説明では、タイムアウトがない(言い換えると、永久待ちの)場合の振舞いを説明するのを原則とする。システムコールの説明で「待ち状態に入る」ないしは「待ち状態に移行させる」と記述されている場合でも、タイムアウト時間を指定した場合には、指定時間経過後に待ち状態が解除され、メインエラーコードを E_TMOUT としてシステムコールからリターンする。また、ポーリングの場合には、待ち状態に入らずにメインエラーコードを E_TMOUT としてシステムコールからリターンする。

タイムアウト指定(TMO型およびTMO_U型)は、正の値でタイムアウト時間、TMO_POL(＝0)でポーリング、TMO_FEVR(＝-1)で永久待ちを指定する。タイムアウト時間が指定された場合、タイムアウトの処理は、システムコールが呼び出されてから、指定された以上の時間が経過した後に行うことを保証しなければならない。

補足事項
ポーリングを行うシステムコールでは待ち状態に入らないため、それを呼び出したタスクの優先順位は変化しない。
一般的な実装においては、タイムアウト時間に1が指定されると、システムコールが呼び出されてから2回めのタイマ割込み(「タイムティック」と呼ぶ場合がある)でタイムアウト処理を行う。タイムアウト時間に0を指定することはできないため(0はTMO_POL に割り付けられている)、このような実装では、システムコールが呼び出された後の最初のタイマ割込みでタイムアウトすることはない。

3.2.8 相対時間とシステム時刻

イベントの発生する時刻を、システムコールを呼び出した時刻などからの相対値で指定する場合には、相対時間(RELTIM型またはRELTIM_U型)を用いる。相対時間を用いてイベントの発生時刻が指定された場合、イベントの処理は、基準となる時刻から指定された以上の時間が経過した後に行うことを保証しなければならない。イベントの発生間隔など、イベントの発生する時刻以外を指定する場合にも、相対時間(RELTIM型またはRELTIM_U型)を用いる。その場合、指定された相対時間の解釈方法は、それぞれの場合毎に定める。時刻を絶対値で指定する場合には、システム時刻(SYSTIM型またはSYSTIM_U型)を用いる。T-Kernelには現在のシステム時刻を設定する機能が用意されているが、この機能を用いてシステム時刻を変更した場合にも、相対時間を用いて指定されたイベントが発生する実世界の時刻(これを実時刻と呼ぶ)は変化しない。言い換えると、相対時間を用いて指定されたイベントが発生するシステム時刻は変化することになる。

SYSTIM システム時刻
基準時間1ミリ秒、64ビットの符号付整数

```
typedef struct systim {
        W       hi;     /* 上位32ビット */
        UW      lo;     /* 下位32ビット */
} SYSTIM;
```

SYSTIM_U システム時刻
基準時間1マイクロ秒、64ビットの符号付整数

```
typedef D       SYSTIM_U;       /* 64ビット */
```

RELTIM 相対時間
基準時間1ミリ秒、32ビットの符号なし整数(UW)

```
typedef UW      RELTIM;
```

RELTIM_U 相対時間
基準時間1マイクロ秒、64ビットの符号なし整数(UD)

```
typedef UD      RELTIM_U;       /* 64ビットでマイクロ秒単位の相対時間 */
```

TMO タイムアウト時間
基準時間1ミリ秒、32ビットの符号付整数(W)

```
typedef W       TMO;
```

TMO_FEVR＝(-1)で永久待ちを指定できる。

TMO_U タイムアウト時間
基準時間1マイクロ秒、64ビットの符号付整数(D)

```
typedef D       TMO_U;          /* 64ビットでマイクロ秒単位のタイムアウト */
```

TMO_FEVR＝(-1)で永久待ちを指定できる。

補足事項
RELTIM, RELTIM_U, TMO, TMO_Uで指定された時間は、指定された時間以上経過した後にタイムアウト等が起こることを保証しなければならない。例えば、タイマ割込み間隔が1ミリ秒で、タイムアウト時間として1ミリ秒が指定された場合、システムコール呼出後の2回目のタイマ割込みでタイムアウトする。(1回目のタイマ割込みでは1ミリ秒未満である。)
システム時刻(SYSTIM_U)がカーネル内部でオーバーフローしてしまうような値を引数に指定した場合のシステムコールの動作は未定義である。

3.2.9 タイマ割込み間隔

T-Kernelの時間管理機能における実際の時間分解能は、「5.7.2節「標準システム構成情報」」の「タイマ割込み間隔」(TTimPeriod)によって指定される。「タイマ割込み間隔」(TTimPeriod)のデフォルトは10ミリ秒である。実際に設定可能な範囲や動作可能な範囲は実装依存である。

「タイマ割込み間隔」を短くすると、タイマ割込みによるシステムのオーバーヘッドが増大するほか、タイマに供給されるクロックやハードウェアの制限により、時計の誤差が大きくなる場合がある。

3.3 高級言語対応ルーチン

タスクやハンドラを高級言語で記述した場合にもカーネルに関する処理と言語環境に関する処理を分離できるように、高級言語対応ルーチンの機能を用意している。高級言語対応ルーチンの利用の有無は、オブジェクト属性やハンドラ属性の一つ(TA_HLNG)として指定される。

TA_HLNG の指定が無い場合は、tk_cre_tsk や tk_def_??? のパラメータで指定されたスタートアドレスから直接タスクやハンドラを起動するが、TA_HLNG の指定がある場合は、最初に高級言語のスタートアップ処理ルーチン(高級言語対応ルーチン)を起動し、その中から tk_cre_tsk や tk_def_??? のパラメータで指定されたタスク起動アドレスやハンドラアドレスに間接ジャンプする。この場合、カーネルから見ると、タスク起動アドレスやハンドラアドレスは高級言語対応ルーチンに対するパラメータとなる。こういった方法をとることにより、カーネルに関する処理と言語環境に関する処理が分離され、異なる言語環境にも容易に対応できる。

また、高級言語対応ルーチンを利用すれば、タスクやハンドラをC言語の関数として書いた場合に、単なる関数のリターン(returnや"}")を行うだけで、タスクを終了するシステムコールやハンドラから戻るシステムコールを自動的に実行することが可能となる。

しかし、MMUを持つシステムにおいては、カーネルと同じ保護レベルで動作する割込みハンドラなどでは比較的容易に高級言語対応ルーチンを実現できるのに対し、カーネルと異なる保護レベルで動作するタスクやタスク例外ハンドラなどは高級言語対応ルーチンを実現することが難しい。そのため、タスクの場合は高級言語対応ルーチンを使用したとしても関数からのリターンによるタスクの終了は保証しない。タスクの関数からreturnや"}"を用いてリターンした場合の動作は未定義とする。タスクの最後には、必ず自タスク終了(tk_ext_tsk)または自タスクの終了と削除(tk_exd_tsk)を発行する必要がある。

また、タスク例外ハンドラの高級言語対応ルーチンはソースコードで提供することとし、ユーザプログラムに組み込むものとしている。

高級言語対応ルーチンの内部動作は[図 3.1.「高級言語対応ルーチンの動作」]のようになる。

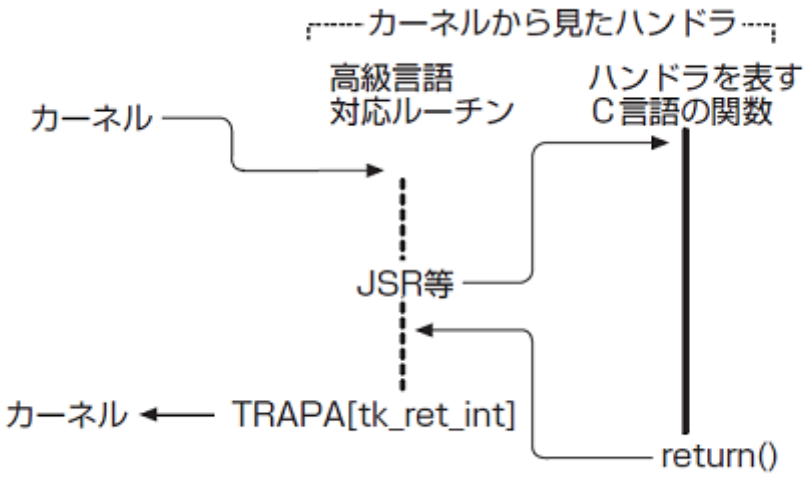

図 3.1: 高級言語対応ルーチンの動作

第 4 章

T-Kernel/OSの機能

この章では、T-Kernel/OS(Operating System)で提供しているシステムコールの詳細について説明を行う。

4.1 タスク管理機能

タスク管理機能は、タスクの状態を直接的に操作／参照するための機能である。タスクを生成／削除する機能、タスクを起動／終了する機能、タスクの優先度を変更する機能、タスクの状態を参照する機能が含まれる。タスクはID番号で識別されるオブジェクトである。タスクのID番号をタスクIDと呼ぶ。タスク状態とスケジューリング規則については、2.2節「タスク状態とスケジューリング規則」を参照すること。

タスクは、実行順序を制御するために、ベース優先度と現在優先度を持つ。単にタスクの優先度といった場合には、タスクの現在優先度を指す。タスクのベース優先度は、タスクの起動時にタスクの起動時優先度に初期化する。ミューテックス機能を使わない場合には、タスクの現在優先度は常にベース優先度に一致している。そのため、タスク起動直後の現在優先度は、タスクの起動時優先度になっている。ミューテックス機能を使う場合に現在優先度がどのように設定されるかについては、4.5.1節「ミューテックス」で述べる。

カーネルは、タスクの終了時に、ミューテックスのロック解除を除いては、タスクが獲得した資源(セマフォ資源、メモリブロックなど)を解放する処理を行わない。タスク終了時に資源を解放するのは、アプリケーションの責任である。

4.1.1 tk_cre_tsk - タスク生成

C言語インタフェース

```
#include <tk/tkernel.h>

ID tskid = tk_cre_tsk (CONST T_CTSK *pk_ctsk );
```

パラメータ

CONST T_CTSK*	pk_ctsk	Packet to Create Task	タスク生成情報

pk_ctsk の内容

void*	exinf	Extended Information	拡張情報
ATR	tskatr	Task Attribute	タスク属性
FP	task	Task Start Address	タスク起動アドレス
PRI	itskpri	Initial Task Priority	タスク起動時優先度
INT	stksz	Stack Size	スタックサイズ(バイト数)
INT	sstksz	System Stack Size	システムスタックサイズ(バイト数)
void*	stkptr	User Stack Pointer	ユーザスタックポインタ
void*	uatb	Address of Task Space Page Table	タスク固有空間ページテーブル
INT	lsid	Logical Space ID	論理空間ID
ID	resid	Resource ID	リソースID
UB	dsname[8]	DS Object name	DSオブジェクト名称

――(以下に実装独自に他の情報を追加してもよい)――

リターンパラメータ

ID	tskid	Task ID	タスクID
	または	Error Code	エラーコード

エラーコード

E_NOMEM	メモリ不足(管理ブロックやスタック用の領域が確保できない)
E_LIMIT	タスクの数がシステムの制限を超えた
E_RSATR	予約属性(tskatr が不正あるいは利用できない)、指定のコプロセッサは存在しない
E_NOSPT	未サポート機能(TA_USERSTACK, TA_TASKSPACE が未サポートの場合)
E_PAR	パラメータエラー
E_ID	リソースID(resid)が不正
E_NOCOP	指定のコプロセッサが使用できない(動作中のハードウェアには搭載されていない、または動作異常が検出された)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

タスクを生成しタスクID番号を割り当てる。具体的には、生成するタスクに対してTCB(Task Control Block)を割り付け、itskpri, task, stksz などの情報をもとにその初期設定を行う。

対象タスクは生成後、休止状態(DORMANT)となる。

itskpri によって、タスクが起動する時の優先度の初期値を指定する。タスク優先度としては、1～140の値を指定することができ、数の小さい方が高い優先度となる。

exinf は、対象タスクに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、タスクに起動パラメータとして渡される他、tk_ref_tsk で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。カーネルでは exinf の内容について関知しない。

tskatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。tskatr のシステム属性の部分では、次のような指定を行う。

```
tskatr := (TA_ASM || TA_HLNG)
   | [TA_SSTKSZ] | [TA_USERSTACK] | [TA_TASKSPACE] | [TA_RESID] | [TA_DSNAME]
   | (TA_RNG0 || TA_RNG1 || TA_RNG2 || TA_RNG3)
   | [TA_COP0] | [TA_COP1] | [TA_COP2] | [TA_COP3] | [TA_FPU]
```

TA_ASM	対象タスクがアセンブリ言語で書かれている
TA_HLNG	対象タスクが高級言語で書かれている
TA_SSTKSZ	システムスタックサイズを指定する
TA_USERSTACK	ユーザスタックポインタを指定する
TA_TASKSPACE	タスク固有空間を指定する
TA_RESID	所属リソースグループを指定する
TA_DSNAME	DSオブジェクト名称を指定する
TA_RNGn	対象タスクは保護レベルnで実行する
TA_COPn	対象タスクが第n番目のコプロセッサを使用する(浮動小数点演算用コプロセッサやDSPを含む)
TA_FPU	対象タスクが浮動小数点演算用コプロセッサを使用する(TA_COPn による指定の内、特に浮動小数点演算を使用するための、CPUに依存しない汎用的な指定である)

実装独自属性の機能は、例えば、被デバッグ対象のタスクであることを指定したりするために利用できる。また、システム属性の残りの部分は、将来マルチプロセッサ属性の指定などを行うために利用できる。

```
#define TA_ASM          0x00000000      /* アセンブリ言語によるプログラム */
#define TA_HLNG         0x00000001      /* 高級言語によるプログラム */
#define TA_SSTKSZ       0x00000002      /* システムスタックサイズを指定 */
#define TA_USERSTACK    0x00000004      /* ユーザスタックポインタを指定 */
#define TA_TASKSPACE    0x00000008      /* タスク固有空間を指定 */
#define TA_RESID        0x00000010      /* 所属リソースグループを指定 */
#define TA_DSNAME       0x00000040      /* DSオブジェクト名称を指定 */
#define TA_RNG0         0x00000000      /* 保護レベル0 で実行 */
#define TA_RNG1         0x00000100      /* 保護レベル1 で実行 */
#define TA_RNG2         0x00000200      /* 保護レベル2 で実行 */
#define TA_RNG3         0x00000300      /* 保護レベル3 で実行 */
#define TA_COP0         0x00001000      /* ID=0 のコプロセッサを使用 */
#define TA_COP1         0x00002000      /* ID=1 のコプロセッサを使用 */
#define TA_COP2         0x00004000      /* ID=2 のコプロセッサを使用 */
#define TA_COP3         0x00008000      /* ID=3 のコプロセッサを使用 */
```

TA_HLNG の指定を行った場合には、タスク起動時に直接 task のアドレスにジャンプするのではなく、高級言語の環境設定プログラム(高級言語対応ルーチン)を通してから task のアドレスにジャンプする。TA_HLNG 属性の場合のタスクは次の形式となる。

```
void task( INT stacd, void *exinf )
```

```
{
        /*
                処理
        */

        tk_ext_tsk(); または tk_exd_tsk(); /* タスクの終了 */
}
```

タスクの起動パラメータとして、tk_sta_tsk で指定するタスク起動コード stacd、および tk_cre_tsk で指定する拡張情報 exinf を渡す。

関数からの単純なリターン(return)でタスクを終了することはできない(してはいけない)。その場合の動作は不定(実装依存)である。

TA_ASM 属性の場合のタスクの形式は実装依存とする。ただし、起動パラメータとして stacd, exinf を渡さなければならない。

タスクは、TA_RNGn で指定された保護レベルで動作する。システムコールや拡張SVCを呼び出すことで保護レベル0に移行し、システムコールや拡張SVCから戻ると元の保護レベルに復帰する。

各タスクはシステムスタックとユーザスタックの2本のスタックを持つ。ユーザスタックは TA_RNGn で指定した保護レベルで使用される。システムスタックは保護レベル0で使用される。システムコールや拡張SVCを呼び出すことにより保護レベルが遷移したときに使用するスタックが切り替えられる。

なお、TA_RNG0 を指定したタスクでは、保護レベルの遷移が起きないためスタックの切替も起きない。TA_RNG0 の場合は、ユーザスタックサイズとシステムスタックサイズの合計を1本のスタックとし、ユーザスタック兼システムスタックとして使用する。

TA_SSTKSZ を指定した場合に sstksz が有効になる。TA_SSTKSZ を指定しなかった場合は、sstksz は無視されデフォルトサイズが適用される。

TA_USERSTACK を指定した場合に stkptr が有効になる。この場合、ユーザスタックはカーネルで用意しない。ユーザスタックは呼出側で用意する。stksz には0を設定しなければならない。TA_USERSTACK を指定しなかった場合は、stkptr は無視される。ただし、TA_RNG0 の場合は、TA_USERSTACK を指定することはできない。TA_RNG0 と TA_USERSTACK を同時に指定した場合は E_PAR が発生する。

TA_TASKSPACE を指定した場合に uatb, lsid が有効となり、タスク固有空間として設定される。TA_TASKSPACE を指定しなかった場合は、uatb, lsid は無視され、タスク固有空間は不定となる。タスク固有空間が不定の間は、共有空間のみアクセスが許される。固有空間にはアクセスしてはいけない。TA_TASKSPACE を指定した場合も指定しなかった場合も、タスク生成後にタスク固有空間を変更することができる。なお、タスク固有空間を変更した場合、タスクがDORMANT状態に戻ってもタスク生成時に指定したタスク固有空間に戻ることはなく、最後に設定されたタスク固有空間のままとなる。

TA_RESID を指定した場合に resid が有効となり、タスクの所属するリソースグループとして resid のリソースグループ(「4.10節「サブシステム管理機能」」参照)が設定される。TA_RESID を指定しなかった場合は resid は無視され、システムリソースグループに所属するよう設定される。なお、所属リソースグループを変更した場合、タスクがDORMANT状態に戻ってもタスク生成時に指定したリソースグループに戻ることはなく、最後に設定されたリソースグループのままとなる。(tk_cre_res 参照)

TA_DSNAME を指定した場合に dsname が有効となり、DSオブジェクト名称として設定される。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DSのシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname 、td_set_dsname を参照のこと。TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

補足事項

タスクは、TA_RNGn で指定した保護レベルと保護レベル0のいずれかでのみ動作する。例えば、TA_RNG3 を指定したタスクが保護レベル1および2で動作することはない。

割込みスタックが分離されていないシステムでは、割込みハンドラもシステムスタックを使用する。割込みハンドラは保護レベル0で動作する。

システムスタックのデフォルトサイズは、システムコールの実行により消費するサイズおよび割込みスタックが分離されていないシステムでは割込みハンドラにより消費されるサイズを考慮して決定する。

システムスタックは保護レベル0の共有空間の常駐メモリとなる。TA_USERSTACK が指定されなかった場合、ユーザスタックは、TA_RNGn で指定した保護レベルの共有空間の常駐メモリとなる。TA_USERSTACK が指定された場合、ユーザスタックのメモリの属性は呼出側で任意に決定する。タスク固有空間の非常駐メモリとしてもよい。

TA_COPn の定義は、CPUなどのハードウェアに依存して決められるため移植性はない。

TA_FPU は、TA_COPn の定義の内、浮動小数点演算の使用に関してのみ移植性のある指定方法として用意される。例えば、浮動小数点コプロセッサが TA_COP0 の場合は、TA_FPU＝TA_COP0 となる。浮動小数点演算を行うのに特にコプロセッサの使用を指定する必要がない場合は、TA_FPU＝0となる。

MMUのないシステムにおいても、移植性確保のために TA_RNGn などすべての属性を受け付けなければならない。例えば、TA_RNGn の指定はすべて TA_RNG0 相当として処理してもよいが、エラーとはしない。

ただし、TA_USERSTACK と TA_TASKSPACE については MMUなしでは対応不可能な場合が多いと思われるため、E_NOSPT としてよい。

4.1.2 tk_del_tsk - タスク削除

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_del_tsk (ID tskid);

パラメータ

ID	tskid	Task ID	タスクID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが休止状態(DORMANT)でない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid で示されたタスクを削除する。

具体的には、tskid で指定されたタスクを休止状態(DORMANT)から未登録状態(NON-EXISTENT)(システムに存在しない状態)へと移行させ、それに伴ってTCBおよびスタック領域を解放する。また、タスクID番号も解放される。休止状態(DORMANT)でないタスクに対してこのシステムコールを実行すると、E_OBJ のエラーとなる。

このシステムコールで自タスクの指定はできない。自タスクを指定した場合には、自タスクが休止状態(DORMANT)ではないため、E_OBJ のエラーとなる。自タスクを削除するには、本システムコールではなく、tk_exd_tsk システムコールを発行する。

4.1.3 tk_sta_tsk - タスク起動

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_sta_tsk (ID tskid , INT stacd);

パラメータ

ID	tskid	Task ID	タスクID
INT	stacd	Task Start Code	タスク起動コード

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが休止状態(DORMANT)でない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

tskid で示されたタスクを起動する。具体的には、休止状態(DORMANT)から実行可能状態(READY)へと移す。

stacd により、タスクの起動時にタスクに渡すパラメータを設定することができる。このパラメータは、対象タスクから参照することができ、簡単なメッセージ通信の目的で利用できる。

タスク起動時のタスク優先度は、対象タスクが生成された時に指定されたタスク起動時優先度(itskpri)となる。

このシステムコールによる起動要求のキューイングは行わない。すなわち、対象タスクが休止状態(DORMANT)でないのにこのシステムコールが発行された場合、このシステムコールは無視され、発行タスクに E_OBJ のエラーが返る。

4.1.4 tk_ext_tsk - 自タスク終了

C言語インタフェース

```
#include <tk/tkernel.h>
```

void tk_ext_tsk (void);

パラメータ

なし

リターンパラメータ

※ システムコールを発行したコンテキストには戻らない

エラーコード

※ 次のようなエラーを検出する可能性があるが、エラーを検出した場合でも、システムコールを発行したコンテキストには戻らない。したがって、システムコールのリターンパラメータとして直接エラーコードを返すことはできない。万一エラーを検出した場合の動作は、実装依存となる。

E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

自タスクを正常終了させ、休止状態(DORMANT)へと移行させる。

補足事項

tk_ext_tsk によるタスクの終了時に、終了するタスクがそれ以前に獲得した資源(メモリブロック、セマフォなど) を自動的に解放するということはない。タスク終了前に資源を解放しておくのは、ユーザの責任である。

tk_ext_tsk は発行元のコンテキストに戻らないシステムコールである。したがって、何らかのエラーを検出した場合にエラーコードを返しても、これらのシステムコールを呼んだ側ではエラーのチェックを行っていないのが普通であり、プログラムが暴走する可能性がある。そこで、これらのシステムコールでは、エラーを検出した場合にも、システムコール発行元へは戻らないものとする。

タスクが休止状態(DORMANT)に戻る時は、タスク優先度などTCBに含まれている情報もリセットされるというのが原則である。たとえば、tk_chg_pri によりタスク優先度を変更されているタスクが、tk_ext_tsk により終了した時、タスク優先度は tk_cre_tsk で指定したタスク起動時優先度(itskpri)に戻る。tk_ext_tsk 実行時のタスク優先度になるわけではない。

元のコンテキストに戻らないシステムコールは、すべてtk_ret_???またはtk_ext_???(tk_exd_???)の名称となっている。

4.1.5 tk_exd_tsk - 自タスクの終了と削除

C言語インタフェース

```
#include <tk/tkernel.h>
```

void tk_exd_tsk (void);

パラメータ

なし

リターンパラメータ

※ システムコールを発行したコンテキストには戻らない

エラーコード

※ 次のようなエラーを検出する可能性があるが、エラーを検出した場合でも、システムコールを発行したコンテキストには戻らない。したがって、システムコールのリターンパラメータとして直接エラーコードを返すことはできない。万一エラーを検出した場合の動作は、実装依存となる。

E_CTX　　コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

自タスクを正常終了させ、さらに自タスクを削除する。すなわち、自タスクを未登録状態(NON-EXISTENT)(システムに存在しない状態)へと移行させる。

補足事項

tk_exd_tsk によるタスクの終了時に、終了するタスクがそれ以前に獲得した資源(メモリブロック、セマフォなど) を自動的に解放するということはない。タスク終了前に資源を解放しておくのは、ユーザの責任である。

tk_exd_tsk は発行元のコンテキストに戻らないシステムコールである。したがって、何らかのエラーを検出した場合にエラーコードを返しても、これらのシステムコールを呼んだ側ではエラーのチェックを行っていないのが普通であり、プログラムが暴走する可能性がある。そこで、これらのシステムコールでは、エラーを検出した場合にも、システムコール発行元へは戻らないものとする。

4.1.6 tk_ter_tsk - 他タスク強制終了

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ter_tsk (ID tskid);

パラメータ

ID	tskid	Task ID	タスクID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが休止状態(DORMANT)または自タスク)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid で示されたタスクを強制的に終了させる。すなわち、tskid で示された対象タスクを休止状態(DORMANT)に移行させる。

対象タスクが待ち状態(強制待ち状態(SUSPENDED)を含む)にあった場合でも、対象タスクは待ち解除となって終了する。また、対象タスクが何らかの待ち行列(セマフォ待ちなど)につながれていた場合には、tk_ter_tsk の実行によってその待ち行列から削除される。

本システムコールでは、自タスクの指定はできない。自タスクを指定した場合には、E_OBJ のエラーとなる。

tk_ter_tsk の対象タスクの状態と実行結果との関係についてまとめたものを表 4.1.「tk_ter_tskの対象タスクの状態と実行結果」に示す。

補足事項

tk_ter_tsk によるタスクの終了時に、終了するタスクがそれ以前に獲得した資源(メモリブロック、セマフォなど)を自動的に解放するということはない。タスク終了前に資源を解放しておくのは、ユーザの責任である。

タスクが休止状態(DORMANT)に戻る時は、タスク優先度などTCBに含まれている情報もリセットされるというのが原則である。たとえば、tk_chg_pri によりタスク優先度を変更されているタスクが、tk_ter_tsk により終了した時、タスク優先度は tk_cre_tsk で指定したタスク起動時優先度(itskpri)に戻る。tk_sta_tsk によって再度タスクを起動した場合、tk_ter_tsk を実行して強制終了された時のタスク優先度になるわけではない。

対象タスク状態	tk_ter_tsk の ercd	処理
実行できる状態(RUNNING,READY) (自タスク以外)	E_OK	強制終了処理
実行状態(RUNNING)(自タスク)	E_OBJ	何もしない
待ち状態(WAITING)	E_OK	強制終了処理
強制待ち状態(SUSPENDED)	E_OK	強制終了処理
二重待ち状態(WAITING-SUSPENDED)	E_OK	強制終了処理
休止状態(DORMANT)	E_OBJ	何もしない
未登録状態(NON-EXISTENT)	E_NOEXS	何もしない

表 4.1: tk_ter_tskの対象タスクの状態と実行結果

他タスクの強制終了は、デバッガなどのOSに密接に関連したごく一部でのみ使用することを原則とする。一般のアプリケーションやミドルウェアでは、他タスクの強制終了は原則として使用してはいけない。これは次のような理由による。

強制終了は、対象タスクの実行状態に関係なく行われる。例えば、タスクがあるミドルウェアの機能を呼び出しているとき、そのタスクを強制終了するとミドルウェアの実行途中でタスクが終了してしまうことになる。そのような状況になれば、ミドルウェアの正常動作は保証できなくなる。

このように、タスクの状態(何を実行中か)が不明な状況で、そのタスクを強制終了させることはできない。したがって、一般にタスクの強制終了は使用してはならない。

4.1.7 tk_chg_pri - タスク優先度変更

C言語インタフェース

```
#include <tk/tkernel.h>

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

パラメータ

ID	tskid	Task ID	タスクID
PRI	tskpri	Task Priority	タスク優先度

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(tskpri が不正あるいは利用できない値)
E_ILUSE	不正使用(上限優先度違反)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid で指定されるタスクのベース優先度を、tskpri で指定される値に変更する。それに伴って、タスクの現在優先度も変更する。

タスク優先度としては、1～140の値を指定することができ、数の小さい方が高い優先度となる。

tskid に TSK_SELF(＝0)が指定されると、自タスクを対象タスクとする。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF を指定した場合には、E_ID のエラーとなる。また、tskpri に TPRI_INI(＝0)が指定されると、対象タスクのベース優先度を、タスクの起動時優先度(itskpri)に変更する。

このシステムコールで変更した優先度は、タスクが終了するまで有効である。タスクが休止状態(DORMANT)に戻る時、終了前のタスクの優先度は捨てられ、タスク生成時に指定されたタスク起動時優先度(itskpri)になる。ただし、休止状態(DORMANT)中に変更した優先度は有効である。次にタスクを起動したときは、その変更された優先度で起動される。

このシステムコールを実行した結果、対象タスクの現在優先度がベース優先度に一致している場合(ミューテックス機能を使わない場合には、この条件は常に成り立つ)には、次の処理を行う。

対象タスクが実行できる状態である場合、タスクの優先順位を、変更後の優先度にしたがって変化させる。変更後の優先度と同じ優先度を持つタスクの間では、対象タスクの優先順位を最低とする。

対象タスクが何らかのタスク優先度順の待ち行列につながれている場合にも、その待ち行列の中での順序を、変更後の優先度にしたがって変化させる。変更後の優先度と同じ優先度を持つタスクの間では、対象タスクを最後につなぐ。

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 で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 エラーを返す。

補足事項

このシステムコールを呼び出した結果、対象タスクのタスク優先度順の待ち行列の中での順序が変化した場合、対象タスクないしはその待ち行列で待っている他のタスクの待ち解除が必要になる場合がある(メッセージバッファの送信待ち行列、および可変長メモリプールの獲得待ち行列)。

対象タスクが、TA_INHERIT 属性のミューテックスのロック待ち状態である場合、このシステムコールでベース優先度を変更したことにより、推移的な優先度継承の処理が必要になる場合がある。

ミューテックス機能を使わない場合には、対象タスクに自タスク、変更後の優先度に自タスクのベース優先度を指定してこのシステムコールが呼び出されると、自タスクの実行順位は同じ優先度を持つタスクの中で最低となる。そのため、このシステムコールを用いて、実行権の放棄を行うことができる。

4.1.8 tk_chg_slt - タスクスライスタイム変更

C言語インタフェース

```
#include <tk/tkernel.h>

ER ercd = tk_chg_slt (ID tskid , RELTIM slicetime );
```

パラメータ

ID	tskid	Task ID	タスクID
RELTIM	slicetime	Slice Time	スライスタイム(ミリ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(slicetime が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid で示されたタスクのスライスタイムを、slicetime で示される値に変更する。

スライスタイムはタスクのラウンドロビンスケジューリングのための機能である。タスクが slicetime 以上の時間、連続して実行されると、同じ優先度を持つタスクの中で最低の優先順位となり、自動的に実行権を次のタスクに譲る。

slicetime＝0は無制限を示し、自動的に実行権を譲ることはない。タスク生成時には slicetime＝0に設定される。

tskid＝TSK_SELF＝0によって自タスクの指定を行うことができる。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF＝0を指定した場合には、E_ID のエラーとなる。

このシステムコールで変更したスライスタイムは、タスクが終了するまで有効である。タスクが休止状態(DORMANT)に戻る時、終了前のタスクのスライスタイムは捨てられ、タスク生成時の値(slicetime＝0)になる。ただし、休止状態(DORMANT)中に変更したスライスタイムは有効である。次にタスクを起動したときは、その変更されたスライスタイムが適用される。

補足事項

より優先度の高いタスクによって実行権が奪われている間は連続実行時間としてカウントされない。また、より高い優先度のタスクによって実行権が奪われても、不連続と扱わない。つまり、より高い優先度のタスクによって実行権が奪われている間は無視して、実行時間をカウントする。

同一優先度に一つしか実行できるタスクがなければ、スライスタイムは実質的に意味はなく、そのタスクが連続して実行される。

同一優先度のタスクの中に、slicetime＝0のタスクが含まれていると、そのタスクが実行権を得た時点でラウンドロビンスケジューリングは停止することになる。

実行時間のカウント方法は実装依存だが、それほど高い精度は要求されない。逆に、アプリケーションは高い精度を期待してはいけない。

4.1.9 tk_chg_slt_u - タスクスライスタイム変更(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_chg_slt_u (ID tskid , RELTIM_U slicetime_u);

パラメータ

ID	tskid	Task ID	タスクID
RELTIM_U	slicetime_u	Slice Time	スライスタイム(マイクロ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(slicetime_u が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_chg_slt のパラメータである slicetime を64ビットマイクロ秒単位の slicetime_u としたシステムコールである。

パラメータが slicetime_u となった点を除き、本システムコールの仕様は tk_chg_slt と同じである。詳細は tk_chg_slt の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.1.10 tk_get_tsp - タスク固有空間の参照

C言語インタフェース

```
#include <tk/tkernel.h>

ER ercd = tk_get_tsp (ID tskid , T_TSKSPC *pk_tskspc );
```

パラメータ

ID	tskid	Task ID	対象タスクのID
T_TSKSPC*	pk_tskspc	Packet of Task Space	タスク固有空間の情報を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_tskspc の内容

void*	uatb	Address of Task Space Page Table	タスク固有空間ページテーブルのアドレス
INT	lsid	Logical Space ID	タスク固有空間ID(論理空間ID)

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(pk_tskspc が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid のタスクの現在のタスク固有空間情報を取得する。

tskid＝TSK_SELF＝0で自タスクを指定することができる。ただし、タスク独立部から発行したシステムコールでtskid＝TSK_SELF＝0を指定した場合には、E_ID のエラーとなる。

補足事項

pk_tskspc の内容(uatb, lsid)の正確な定義は、実装定義である。ただし、できる限り上記の定義に沿うよう実装すること。

4.1.11 tk_set_tsp - タスク固有空間の設定

C言語インタフェース

```
#include <tk/tkernel.h>

ER ercd = tk_set_tsp (ID tskid , CONST T_TSKSPC *pk_tskspc );
```

パラメータ

ID	tskid	Task ID	対象タスクのID
CONST T_TSKSPC*	pk_tskspc	Packet of Task Space	タスク固有空間の情報

pk_tskspc の内容

void*	uatb	Address of Task Space Page Table	タスク固有空間ページテーブルのアドレス
INT	lsid	Logical Space ID	タスク固有空間ID(論理空間ID)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(pk_tskspc が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid のタスクのタスク固有空間を設定する。

tskid＝TSK_SELF＝0で自タスクを指定することができる。ただし、タスク独立部から発行したシステムコールでtskid＝TSK_SELF＝0を指定した場合には、E_ID のエラーとなる。

タスクの固有空間を変更したことによる影響は、カーネルは関知しない。例えば、タスク固有空間を実行中のタスクに対してタスク固有空間を変更すると、そのタスクが暴走するなどの可能性がある。このような問題が起きないようにするのは、呼出側の責任である。

補足事項

pk_tskspc の内容(uatb, lsid)の正確な定義は、実装定義である。ただし、できる限り上記の定義に沿うよう実装すること。

4.1.12 tk_get_rid - タスクの所属リソースグループの参照

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID resid = tk_get_rid (ID tskid);

パラメータ

ID	tskid	Task ID	対象タスクのID

リターンパラメータ

ID	resid	Resource ID	リソースID
	または	Error Code	エラーコード

エラーコード

E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	所属リソースグループが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid のタスクが現在所属しているリソースグループのリソースIDを返す。

tskid＝TSK_SELF＝0で自タスクを指定することができる。ただし、タスク独立部から発行したシステムコールでtskid＝TSK_SELF＝0を指定した場合には、E_ID のエラーとなる。

補足事項

リソースグループについては「4.10節「サブシステム管理機能」」を参照。

所属しているリソースグループが削除されていた場合にも、削除されたリソースIDを返す場合がある。エラー(E_OBJ)を返すか否かは実装に依存する。(tk_cre_res, tk_del_res 参照)

本システムコールは、サブシステムで利用する。サブシステムは、リソースIDによってプロセスを認識している。しかしながら、アプリケーションから拡張SVCを発行してサブシステムに処理を実行させる場合、リソースIDを指定することはできない。そこでサブシステムでは本システムコールを利用してリソースIDを取得する。

4.1.13 tk_set_rid - タスクの所属リソースグループの設定

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID oldid = tk_set_rid (ID tskid , ID resid);

パラメータ

ID	tskid	Task ID	対象タスクのID
ID	resid	Resource ID	新リソースID

リターンパラメータ

ID	oldid	Old Resource ID	旧リソースID
	または	Error Code	エラーコード

エラーコード

E_ID	不正ID番号(tskid, resid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid, resid のオブジェクトが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid のタスクが現在所属しているリソースグループを resid のリソースグループに変更する。戻値に変更前のリソースグループのリソースIDを返す。

tskid＝TSK_SELF＝0で自タスクを指定することができる。ただし、タスク独立部から発行したシステムコールでtskid＝TSK_SELF＝0を指定した場合には、E_ID のエラーとなる。

補足事項

リソースグループについては「4.10節「サブシステム管理機能」」を参照。

resid がすでに削除されたリソースグループであっても、エラーとならない場合がある。エラー(E_NOEXS)となるか否かは実装依存である。原則として、削除されたリソースグループを指定しないようにするのは、呼出側の責任となる。

4.1.14 tk_get_reg - タスクレジスタの取得

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_reg (ID tskid , T_REGS *pk_regs , T_EIT *pk_eit , T_CREGS *pk_cregs);

パラメータ

ID	tskid	Task ID	対象タスクのID
T_REGS*	pk_regs	Packet of Registers	汎用レジスタの値を返す領域へのポインタ
T_EIT*	pk_eit	Packet of EIT Registers	例外時に保存されるレジスタの値を返す領域へのポインタ
T_CREGS*	pk_cregs	Packet of Control Registers	制御レジスタの値を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

T_REGS, T_EIT, T_CREGSの内容は、CPUおよび実装ごとに定義する。

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが自タスク)
E_CTX	コンテキストエラー(タスク独立部からの発行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid のタスクの現在のレジスタの内容を参照する。

pk_regs, pk_eit, pk_cregs にそれぞれ NULL を指定すると、対応するレジスタは参照されない。

参照されたレジスタの値が、タスク部実行中のものであるとは限らない。

自タスクに対して本システムコールを発行することはできない。(E_OBJ)

補足事項

参照可能なレジスタは、タスクのコンテキストに含まれるすべてのレジスタを原則とする。また、CPUに物理的に存在するレジスタ以外に、カーネルが仮想的にレジスタとして扱っているものがあればそれも含まれる。

4.1.15 tk_set_reg - タスクレジスタの設定

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_set_reg (ID tskid , CONST T_REGS *pk_regs , CONST T_EIT *pk_eit , CONST T_CREGS *pk_cregs);

パラメータ

ID	tskid	Task ID	対象タスクのID
CONST T_REGS*	pk_regs	Packet of Registers	汎用レジスタ
CONST T_EIT*	pk_eit	Packet of EIT Registers	例外時に保存されるレジスタ
CONST T_CREGS*	pk_cregs	Packet of Control Registers	制御レジスタ

T_REGS, T_EIT, T_CREGSの内容は、CPUおよび実装ごとに定義する。

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが自タスク)
E_CTX	コンテキストエラー(タスク独立部からの発行)
E_PAR	設定するレジスタ値が不正(実装依存)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid のタスクのレジスタを指定の内容に設定する。

pk_regs, pk_eit, pk_cregs にそれぞれ NULL を指定すると、対応するレジスタは設定されない。

設定するレジスタの値が、タスク部実行中のものであるとは限らない。レジスタの値を設定したことによる影響には、カーネルは関知しない。

ただし、カーネルの動作上変更が許されないレジスタやレジスタ内の一部のビットが変更できないようになっている場合がある。(実装依存)

自タスクに対して本システムコールを発行することはできない。(E_OBJ)

4.1.16 tk_get_cpr - コプロセッサのレジスタの取得

C言語インタフェース

```
#include <tk/tkernel.h>

ER ercd = tk_get_cpr (ID tskid , INT copno , T_COPREGS *pk_copregs );
```

パラメータ

ID	tskid	Task ID	対象タスクのID
INT	copno	Coprocessor Number	コプロセッサ番号(0～3)
T_COPREGS*	pk_copregs	Packet of Coprocessor Registers	コプロセッサのレジスタの値を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_copregs の内容

T_COP0REG	cop0	Coprocessor Number 0 Register	コプロセッサ番号0のレジスタ
T_COP1REG	cop1	Coprocessor Number 1 Register	コプロセッサ番号1のレジスタ
T_COP2REG	cop2	Coprocessor Number 2 Register	コプロセッサ番号2のレジスタ
T_COP3REG	cop3	Coprocessor Number 3 Register	コプロセッサ番号3のレジスタ

T_COPnREGの内容は、CPUおよび実装ごとに定義する。

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが自タスク)
E_CTX	コンテキストエラー(タスク独立部からの発行)
E_PAR	パラメータエラー(copno が不正または指定のコプロセッサは存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid のタスクの copno で指定したコプロセッサの現在のレジスタ内容を参照する。

参照されたレジスタの値が、タスク部実行中のものであるとは限らない。

自タスクに対して本システムコールを発行することはできない。(E_OBJ)

補足事項

参照可能なレジスタは、タスクのコンテキストに含まれるすべてのレジスタを原則とする。また、CPUに物理的に存在するレジスタ以外に、カーネルが仮想的にレジスタとして扱っているものがあればそれも含まれる。

4.1.17 tk_set_cpr - コプロセッサのレジスタの設定

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_set_cpr (ID tskid , INT copno , CONST T_COPREGS *pk_copregs);

パラメータ

ID	tskid	Task ID	対象タスクのID
INT	copno	Coprocessor Number	コプロセッサ番号(0～3)
CONST T_COPREGS*	pk_copregs	Packet of Coprocessor Registers	コプロセッサのレジスタ

pk_copregs の内容

T_COP0REG	cop0	Coprocessor Number 0 Register	コプロセッサ番号0のレジスタ
T_COP1REG	cop1	Coprocessor Number 1 Register	コプロセッサ番号1のレジスタ
T_COP2REG	cop2	Coprocessor Number 2 Register	コプロセッサ番号2のレジスタ
T_COP3REG	cop3	Coprocessor Number 3 Register	コプロセッサ番号3のレジスタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが自タスク)
E_CTX	コンテキストエラー(タスク独立部からの発行)
E_PAR	パラメータエラー(copno が不正または指定のコプロセッサは存在しない)、設定するレジスタ値が不正(実装依存)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid のタスクの copno で指定したコプロセッサのレジスタに指定の内容を設定する。

設定するレジスタの値が、タスク部実行中のものであるとは限らない。レジスタの値を設定したことによる影響には、カーネル

は関知しない。

ただし、カーネルの動作上変更が許されないレジスタやレジスタ内の一部のビットが変更できないようになっている場合がある。(実装依存)

自タスクに対して本システムコールを発行することはできない。(E_OBJ)

4.1.18 tk_inf_tsk - タスク統計情報参照

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_inf_tsk (ID tskid , T_ITSK *pk_itsk , BOOL clr);

パラメータ

ID	tskid	Task ID	タスクID
T_ITSK*	pk_itsk	Packet to Refer Task Statistics	タスク統計情報を返す領域へのポインタ
BOOL	clr	Clear	タスク統計情報のクリアの有無

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_itsk の内容

RELTIM	stime	System Time	累積システムレベル実行時間(ミリ秒)
RELTIM	utime	User Time	累積ユーザレベル実行時間(ミリ秒)

——(以下に実装独自に他の情報を追加してもよい)——

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(pk_itsk が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid で示された対象タスクの統計情報を参照する。

clr＝TRUE≠0の場合は、統計情報を取り出した後、累積時間をリセット(0クリア)する。

tskid＝TSK_SELF＝0によって自タスクの指定を行うことができる。ただし、タスク独立部から発行したシステムコールでtskid＝TSK_SELF＝0を指定した場合には、E_ID のエラーとなる。

タスク統計情報(T_ITSK)の stime, utime は、ミリ秒単位に切り上げた値(単位ミリ秒)を返す。マイクロ秒単位の情報を知りたい場合には、tk_inf_tsk_u を使う。

補足事項

システムレベル実行時間は TA_RNG0 で実行していた時間、ユーザレベル実行時間は TA_RNG0 以外で実行していた時間である。したがって、TA_RNG0 で生成されたタスクは、すべてシステムレベル実行時間としてカウントされることになる。

実行時間のカウント方法は実装依存だが、それほど高い精度は要求されない。逆に、アプリケーションは高い精度を期待してはいけない。

4.1.19 tk_inf_tsk_u - タスク統計情報参照(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_inf_tsk_u (ID tskid , T_ITSK_U *pk_itsk_u , BOOL clr);

パラメータ

ID	tskid	Task ID	タスクID
T_ITSK_U*	pk_itsk_u	Packet to Refer Task Statistics	タスク統計情報を返す領域へのポインタ
BOOL	clr	Clear	タスク統計情報のクリアの有無

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_itsk_u の内容

RELTIM_U	stime_u	System Time	累積システムレベル実行時間(マイクロ秒)
RELTIM_U	utime_u	User Time	累積ユーザレベル実行時間(マイクロ秒)

——(以下に実装独自に他の情報を追加してもよい)——

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(pk_itsk_u が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_inf_tsk のリターンパラメータである stime と utime を64ビットマイクロ秒単位の stime_u, utime_u としたシステムコールである。

リターンパラメータが stime_u, utime_u となった点を除き、本システムコールの仕様は tk_inf_tsk と同じである。詳細は tk_inf_tsk の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.1.20 tk_ref_tsk - タスク状態参照

C言語インタフェース

```
#include <tk/tkernel.h>

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

パラメータ

ID	tskid	Task ID	タスクID
T_RTSK*	pk_rtsk	Packet to Refer Task Status	タスク状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rtsk の内容

void*	exinf	Extended Information	拡張情報
PRI	tskpri	Task Priority	現在の優先度
PRI	tskbpri	Task Base Priority	ベース優先度
UINT	tskstat	Task State	タスク状態
UINT	tskwait	Task Wait Factor	待ち要因
ID	wid	Waiting Object ID	待ちオブジェクトID
INT	wupcnt	Wakeup Count	起床要求キューイング数
INT	suscnt	Suspend Count	強制待ち要求ネスト数
RELTIM	slicetime	Slice Time	最大連続実行時間(ミリ秒)
UINT	waitmask	Wait Mask	待ちを禁止されている待ち要因
UINT	texmask	Task Exception Mask	許可されているタスク例外
UINT	tskevent	Task Event	発生しているタスクイベント
——(以下に実装独自に他の情報を追加してもよい)——			

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(pk_rtsk が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

tskid で示された対象タスクの各種の状態を参照する。

tskstat は次のような値をとる。

TTS_RUN	0x0001	実行状態(RUNNING)
TTS_RDY	0x0002	実行可能状態(READY)
TTS_WAI	0x0004	待ち状態(WAITING)
TTS_SUS	0x0008	強制待ち状態(SUSPENDED)
TTS_WAS	0x000c	二重待ち状態(WAITING-SUSPENDED)
TTS_DMT	0x0010	休止状態(DORMANT)
TTS_NODISWAI	0x0080	待ち禁止拒否状態

TTS_RUN, TTS_WAI などによるタスク状態の表現はビット対応になっているため、和集合の判定を行う(例えば、実行状態(RUNNING)または実行可能状態(READY)であることを判定する)場合に便利である。なお、上記の状態のうち、TTS_WAS は TTS_SUS と TTS_WAI が複合したものであるが、TTS_SUS がこれ以外の状態(TTS_RUN, TTS_RDY, TTS_DMT)と複合することはない。

TTS_WAI(TTS_WAS 含む)の場合、tk_dis_wai による待ち禁止を拒否している状態であれば、TTS_NODISWAI がセットされる。TTS_WAI 以外と TTS_NODISWAI が組み合わされることはない。

割込みハンドラの中から、割り込まれたタスクを対象とした tk_ref_tsk を実行した場合は、tskstat として実行状態(RUNNING)(TTS_RUN)を返す。

tskstat が TTS_WAI(TTS_WAS を含む)の場合、tskwait, wid は 表 4.2.「tskwait と wid の値」のような値をとる。

tskwait	値	意味	wid
TTW_SLP	0x00000001	tk_slp_tsk による待ち	0
TTW_DLY	0x00000002	tk_dly_tsk による待ち	0
TTW_SEM	0x00000004	tk_wai_sem による待ち	待ち対象の semid
TTW_FLG	0x00000008	tk_wai_flg による待ち	待ち対象の flgid
TTW_MBX	0x00000040	tk_rcv_mbx による待ち	待ち対象の mbxid
TTW_MTX	0x00000080	tk_loc_mtx による待ち	待ち対象の mtxid
TTW_SMBF	0x00000100	tk_snd_mbf による待ち	待ち対象の mbfid
TTW_RMBF	0x00000200	tk_rcv_mbf による待ち	待ち対象の mbfid
TTW_CAL	0x00000400	ランデブ呼出待ち	待ち対象の porid
TTW_ACP	0x00000800	ランデブ受付待ち	待ち対象の porid
TTW_RDV	0x00001000	ランデブ終了待ち	0
(TTW_CAL | TTW_RDV)	0x00001400	ランデブ呼出または終了待ち	0
TTW_MPF	0x00002000	tk_get_mpf による待ち	待ち対象の mpfid
TTW_MPL	0x00004000	tk_get_mpl による待ち	待ち対象の mplid
TTW_EV1	0x00010000	タスクイベント#1待ち	0
TTW_EV2	0x00020000	タスクイベント#2待ち	0
TTW_EV3	0x00040000	タスクイベント#3待ち	0
TTW_EV4	0x00080000	タスクイベント#4待ち	0
TTW_EV5	0x00100000	タスクイベント#5待ち	0
TTW_EV6	0x00200000	タスクイベント#6待ち	0
TTW_EV7	0x00400000	タスクイベント#7待ち	0
TTW_EV8	0x00800000	タスクイベント#8待ち	0

表 4.2: tskwait と wid の値

tskstat が TTS_WAI (TTS_WAS を含む) でない場合は、tskwait, wid はともに0となる。

waitmask は、tskwait と同じビット並びとなる。

休止状態(DORMANT)のタスクでは wupcnt=0, suscnt=0, tskevent=0 である。

tskid＝TSK_SELF＝0によって自タスクの指定を行うことができる。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF＝0を指定した場合には、E_ID のエラーとなる。

tk_ref_tsk で、対象タスクが存在しない場合には、エラー E_NOEXS となる。

タスク状態情報(T_RTSK)の slicetime は、ミリ秒単位に切り上げた値(単位ミリ秒)を返す。マイクロ秒単位の情報を知りたい場合には、tk_ref_tsk_u を使う。

補足事項

このシステムコールで tskid＝TSK_SELF を指定した場合でも、自タスクのIDは分からない。自タスクのIDを知りたい場合には、tk_get_tid を利用する。

4.1.21 tk_ref_tsk_u - タスク状態参照(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_tsk_u (ID tskid , T_RTSK_U *pk_rtsk_u);

パラメータ

ID	tskid	Task ID	タスクID
T_RTSK_U*	pk_rtsk_u	Packet to Refer Task Status	タスク状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rtsk_u の内容

void*	exinf	Extended Information	拡張情報
PRI	tskpri	Task Priority	現在の優先度
PRI	tskbpri	Task Base Priority	ベース優先度
UINT	tskstat	Task State	タスク状態
UINT	tskwait	Task Wait Factor	待ち要因
ID	wid	Waiting Object ID	待ちオブジェクトID
INT	wupcnt	Wakeup Count	起床要求キューイング数
INT	suscnt	Suspend Count	強制待ち要求ネスト数
RELTIM_U	slicetime_u	Slice Time	最大連続実行時間(マイクロ秒)
UINT	waitmask	Wait Mask	待ちを禁止されている待ち要因
UINT	texmask	Task Exception Mask	許可されているタスク例外
UINT	tskevent	Task Event	発生しているタスクイベント

——(以下に実装独自に他の情報を追加してもよい)——

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(pk_rtsk_u が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

tk_ref_tsk のリターンパラメータである slicetime を64ビットマイクロ秒単位の slicetime_u としたシステムコールである。リターンパラメータが slicetime_u となった点を除き、本システムコールの仕様は tk_ref_tsk と同じである。詳細は tk_ref_tsk の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.2 タスク付属同期機能

タスク付属同期機能は、タスクの状態を直接的に操作することによって同期を行うための機能である。タスクを起床待ちにする機能とそこから起床する機能、タスクの起床要求をキャンセルする機能、タスクの待ち状態を強制解除する機能、タスクを強制待ち状態へ移行する機能とそこから再開する機能、自タスクの実行を遅延する機能、タスクイベントに関する機能、タスクの待ち状態を禁止する機能が含まれる。

タスクに対する起床要求は、キューイングされる。すなわち、起床待ち状態でないタスクを起床しようとすると、そのタスクを起床しようとしたという記録が残り、後でそのタスクが起床待ちに移行しようとした時に、タスクを起床待ち状態にしない。タスクに対する起床要求のキューイングを実現するために、タスクは起床要求キューイング数を持つ。タスクの起床要求キューイング数は、タスクの起動時に0にクリアする。

タスクに対する強制待ち要求は、ネストされる。すなわち、すでに強制待ち状態(二重待ち状態を含む)になっているタスクを再度強制待ち状態に移行させようとすると、そのタスクを強制待ち状態に移行させようとしたという記録が残り、後でそのタスクを強制待ち状態(二重待ち状態を含む)から再開させようとした時に、強制待ちからの再開を行わない。タスクに対する強制待ち要求のネストを実現するために、タスクは強制待ち要求ネスト数を持つ。タスクの強制待ち要求ネスト数は、タスクの起動時に0にクリアする。

4.2.1 tk_slp_tsk - 自タスクを起床待ち状態へ移行

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_slp_tsk (TMO tmout);

パラメータ

TMO	tmout	Timeout	タイムアウト指定(ミリ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(tmout≦(-2))
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_slp_tsk システムコールでは、自タスクを実行状態(RUNNING)から起床待ち状態(tk_wup_tsk を待つ状態)に移す。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、具体的には自タスクの起床要求キューイング数が1以上の場合には、起床要求キューイング数から1を減じ、自タスクを待ち状態に移行させず、そのまま実行を継続する。

tmout で指定した時間が経過する前にこのタスクを対象とした tk_wup_tsk が発行された場合は、このシステムコールは正常終了する。一方、tmout で指定した時間が経過する間に tk_wup_tsk が発行されなかった場合は、タイムアウトエラー E_TMOUT となる。なお、tmout＝TMO_FEVR＝(-1)により、タイムアウトまでの時間が無限大であることを示す。この場合は、tk_wup_tsk が発行されるまで永久に待ち状態になる。

補足事項

tk_slp_tsk は自タスクを待ち状態に移すシステムコールであるため、tk_slp_tsk がネストすることはあり得ない。しかし、tk_slp_tsk によって待ち状態になっているタスクに対して、他のタスクから tk_sus_tsk が実行される可能性はある。この場合、このタスクは二重待ち状態(WAITING-SUSPENDED)となる。

タスクの単純な遅延(時間待ち)を行うのであれば、tk_slp_tsk ではなく、tk_dly_tsk を用いるべきである。

タスク起床待ちはアプリケーションでの利用を前提とし、ミドルウェアでは原則として使用してはいけない。これは、次の理由による。

同一タスクにおいて、2ヵ所以上でタスク起床待ちによる同期を行うと、起床要求が混同してしまい誤動作することになる。例えば、アプリケーションとミドルウェアの双方で起床待ちによる同期を利用していた場合、ミドルウェア内で起床待ちしている間に、アプリケーションが起床要求してしまうことがありえる。このような状況になれば、ミドルウェアもアプリケーションも正常な動作はできなくなる。

このように、どこで起床待ちしているかなどの状況がわからないと、正しいタスクの同期ができなくなる。タスク起床待ちによるタスクの同期は簡便な方法としてよく利用されるため、アプリケーションで自由に利用できることを確保するために、ミドルウェアでは使用しないことを原則とする。

4.2.2 tk_slp_tsk_u - 自タスクを起床待ち状態へ移行(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_slp_tsk_u (TMO_U tmout_u);

パラメータ

TMO_U	tmout_u	Timeout	タイムアウト指定(マイクロ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(tmout_u≦(-2))
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_slp_tsk のパラメータである tmout を64ビットマイクロ秒単位の tmout_u としたシステムコールである。

パラメータが tmout_u となった点を除き、本システムコールの仕様は tk_slp_tsk と同じである。詳細は tk_slp_tsk の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.2.3 tk_wup_tsk - 他タスクの起床

C言語インタフェース

```
#include <tk/tkernel.h>

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

パラメータ

ID	tskid	Task ID	タスクID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが自タスクまたは休止状態(DORMANT))
E_QOVR	キューイングまたはネストのオーバーフロー(キューイング数 wupcnt のオーバーフロー)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

tskid で示されるタスクが tk_slp_tsk の実行による待ち状態であった場合に、その待ち状態を解除する。

本システムコールでは、自タスクを指定することはできない。自タスクを指定した場合には、E_OBJ のエラーとなる。

対象タスクが tk_slp_tsk を実行しておらず、待ち状態でない場合には、tk_wup_tsk による起床要求はキューイングされる。すなわち、対象タスクに対して tk_wup_tsk が発行されたという記録が残り、この後で対象タスクが tk_slp_tsk を実行した場合にも、待ち状態にはならない。これを起床要求のキューイングと呼ぶ。

起床要求のキューイングの動作は、具体的には次のようになる。各タスクは、TCBの中に起床要求キューイング数(wupcnt)という状態を持っており、その初期値(tk_sta_tsk 実行時の値)は0である。起床待ち状態でないタスクに対して tk_wup_tsk を実行することにより、対象タスクの起床要求キューイング数がプラス1される。一方、タスクが tk_slp_tsk を実行することにより、そのタスクの起床要求キューイング数がマイナス1される。そうして、起床要求キューイング数＝0のタスクが tk_slp_tsk を実行した時に、起床要求キューイング数がマイナスになる代わりに、そのタスクが待ち状態になる。

tk_wup_tsk を1回キューイングすること(wupcnt＝1)は常に可能であるが、起床要求キューイング数(wupcnt)の最大値は実装依存であり、1以上の適当な値をとる。すなわち、待ち状態でないタスクに対して tk_wup_tsk を1回発行してもエラーとはならないが、2回目以降の tk_wup_tsk がエラーとなるかどうかは実装依存である。

起床要求キューイング数(wupcnt)の最大値の制限を越えて tk_wup_tsk を発行した場合には、E_QOVR のエラーとなる。

4.2.4 tk_can_wup - タスクの起床要求を無効化

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT wupcnt = tk_can_wup (ID tskid);

パラメータ

ID	tskid	Task ID	タスクID

リターンパラメータ

INT	wupcnt	Wakeup Count	キューイングされていた起床要求回数
	または	Error Code	エラーコード

エラーコード

E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが休止状態(DORMANT))

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid で示されたタスクの起床要求キューイング数(wupcnt)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すべてキャンセルする。すなわち、対象タスクの起床要求キューイング数(wupcnt)を0にする。

tskid＝TSK_SELF＝0によって自タスクの指定になる。ただし、タスク独立部から発行したシステムコールでtskid＝TSK_SELF＝0を指定した場合には、E_ID のエラーとなる。

補足事項

このシステムコールは、周期的にタスクを起床して動かすような処理を行う場合に、時間内に処理が終わっているかどうかを判定するために利用できる。すなわち、前の起床要求に対する処理が終了して tk_slp_tsk を発行する前に、それを監視するタスクが tk_can_wup を発行し、そのリターンパラメータである wupcnt が1以上の値であった場合、前の起床要求に対する処理が時間内に終了しなかったということを示す。したがって、処理の遅れに対して何らかの処置をとることができる。

4.2.5 tk_rel_wai - 他タスクの待ち状態解除

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_rel_wai (ID tskid);

パラメータ

ID	tskid	Task ID	タスクID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが待ち状態ではない(自タスクや休止状態(DORMANT)の場合を含む))

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

tskid で示されるタスクが何らかの待ち状態(強制待ち状態(SUSPENDED)を除く)にある場合に、それを強制的に解除する。

tk_rel_wai により待ち状態が解除されたタスクに対しては、エラー E_RLWAI が返る。このとき、対象タスクは資源を確保せずに(待ち解除の条件が満たされないまま)待ち解除となることが保証される。

tk_rel_wai では、待ち状態解除要求のキューイングは行わない。すなわち、tskid で示される対象タスクが既に待ち状態であればその待ち状態を解除するが、対象タスクが待ち状態でなければ、発行元にエラー E_OBJ が返る。本システムコールで自タスクを指定した場合にも、同様に E_OBJ のエラーとなる。

tk_rel_wai では、強制待ち状態(SUSPENDED)の解除は行わない。二重待ち状態(WAITING-SUSPENDED)のタスクを対象として tk_rel_wai を発行すると、対象タスクは強制待ち状態(SUSPENDED)となる。強制待ち状態(SUSPENDED)も解除する必要がある場合には、別に tk_rsm_tsk または tk_frsm_tsk を発行する。

tk_rel_wai の対象タスクの状態と実行結果との関係についてまとめたものを 表 4.3.「tk_rel_waiの対象タスクの状態と実行結果 」に示す。

補足事項

アラームハンドラ等を用いて、あるタスクが待ち状態に入ってから一定時間後にこのシステムコールを発行することにより、タイムアウトに類似した機能を実現することができる。

対象タスク状態	tk_rel_wai の ercd	処理
実行できる状態(RUNNING, READY)(自タスク以外)	E_OBJ	何もしない
実行状態(RUNNING)(自タスク)	E_OBJ	何もしない
待ち状態(WAITING)	E_OK	待ち解除
強制待ち状態(SUSPENDED)	E_OBJ	何もしない
二重待ち状態(WAITING-SUSPENDED)	E_OK	SUSPENDED状態に移行
休止状態(DORMANT)	E_OBJ	何もしない
未登録状態(NON-EXISTENT)	E_NOEXS	何もしない

表 4.3: tk_rel_waiの対象タスクの状態と実行結果

tk_rel_wai と tk_wup_tsk とは次のような違いがある。

- tk_wup_tsk は tk_slp_tsk による待ち状態のみを解除するが、tk_rel_wai ではそれ以外の要因(tk_wai_flg, tk_wai_sem, tk_rcv_mbx, tk_get_mpl, tk_dly_tsk 等) による待ち状態も解除する。
- 待ち状態に入っていたタスクから見ると、tk_wup_tsk による待ち状態の解除は正常終了(E_OK)であるのに対して、tk_rel_wai による待ち状態の解除はエラー(E_RLWAI)である。
- tk_wup_tsk の場合は、対象タスクがまだ tk_slp_tsk を実行していなくても、要求がキューイングされる。一方、tk_rel_wai の場合は、対象タスクが既に待ち状態に無い場合には、E_OBJ のエラーとなる。

4.2.6 tk_sus_tsk - 他タスクを強制待ち状態へ移行

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_sus_tsk (ID tskid);

パラメータ

ID	tskid	Task ID	タスクID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが自タスクまたは休止状態(DORMANT))
E_CTX	ディスパッチ禁止状態で実行状態のタスクを指定した
E_QOVR	キューイングまたはネストのオーバーフロー(ネスト数 suscnt のオーバーフロー)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

tskid で示されたタスクを強制待ち状態(SUSPENDED)に移し、タスクの実行を中断させる。

強制待ち状態(SUSPENDED)は、tk_rsm_tsk, tk_frsm_tsk システムコールの発行によって解除される。

tk_sus_tsk の対象タスクが既に待ち状態であった場合には、tk_sus_tsk の実行により、対象タスクは待ち状態と強制待ち状態が複合した二重待ち状態(WAITING-SUSPENDED)となる。その後、このタスクの待ち解除の条件が満たされると、対象タスクは強制待ち状態(SUSPENDED)に移行する。一方、このタスクに対して tk_rsm_tsk が発行されると、対象タスクは前と同じ待ち状態に戻る。([図 2.1.「タスク状態遷移図」]を参照)

強制待ち状態(SUSPENDED)は他タスクの発行したシステムコールによる中断状態を意味するものなので、本システムコールで自タスクを指定することはできない。自タスクを指定した場合には、E_OBJ のエラーとなる。

タスク独立部からの本システムコールの発行において、ディスパッチ禁止状態で実行状態(RUNNING)のタスクを指定した場合は E_CTX のエラーとなる。

あるタスクに対して複数回の tk_sus_tsk が発行された場合、そのタスクは多重の強制待ち状態(SUSPENDED)になる。これを強制待ち要求のネストと呼ぶ。この場合、tk_sus_tsk が発行された回数(suscnt)と同じ回数の tk_rsm_tsk を発行することにより、対象タスクが元の状態に戻る。したがって、tk_sus_tsk～tk_rsm_tsk の対をネストすることが可能である。

強制待ち要求のネストの機能(同一タスクに対して2回以上の tk_sus_tsk を発行する機能)の有無およびその回数の制限値は、実装依存である。

強制待ち要求をネストできないシステムで複数の tk_sus_tsk が発行された場合や、ネスト回数の制限値を越えた場合には、E_QOVR のエラーとなる。

補足事項

あるタスクが資源獲得のための待ち状態(セマフォ待ちなど)で、かつ強制待ち状態(SUSPENDED)の場合でも、強制待ち状態(SUSPENDED)でない時と同じ条件によって資源の割当て(セマフォの割当てなど)が行われる。強制待ち状態(SUSPENDED)であっても、資源割当ての遅延などが行われるわけではなく、資源割当てや待ち状態の解除に関する条件や優先度は全く変わらない。すなわち、強制待ち状態(SUSPENDED)は、他の処理やタスク状態と直交関係にある。

強制待ち状態(SUSPENDED)のタスクに対して資源割当ての遅延(一時的な優先度の低下) を行いたいのであれば、ユーザ側で、tk_sus_tsk, tk_rsm_tsk に tk_chg_pri を組み合せて発行すれば良い。

タスク強制待ちは、仮想記憶システムにおけるページフォルト処理やデバッガのブレークポイント処理等のOSに密接に関連したごく一部でのみ使用することを原則とする。一般のアプリケーションおよびミドルウェアでは原則として使用してはいけない。これは、次のような理由による。

タスク強制待ちは、対象のタスクの実行状態に関係なく行われる。例えば、タスクがあるミドルウェアの機能を呼び出しているとき、そのタスクを強制待ち状態にするとミドルウェアの内部でタスクが停止することになる。ミドルウェアでは、リソース管理などで排他制御等を行っている場合があり、あるタスクがミドルウェア内でリソースを獲得したまま停止してしまったことにより、他のタスクがそのミドルウェアを使用できなくなる状況が起きうる。このようにして連鎖的にタスクが停止してしまい、システム全体が停止(デッドロック)することもありうる。

このように、タスクの状態(何を実行中か)が不明な状況で、そのタスクを停止させることはできない。したがって、一般にタスク強制待ちは使用してはならない。

4.2.7 tk_rsm_tsk - 強制待ち状態のタスクを再開

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_rsm_tsk (ID tskid);

パラメータ

ID	tskid	Task ID	タスクID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが強制待ち状態(SUSPENDED)ではない(自タスクや休止状態(DORMANT)の場合を含む))

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid で示されたタスクの強制待ち状態(SUSPENDED)を解除する。すなわち、対象タスクが以前に発行された tk_sus_tsk によって強制待ち状態(SUSPENDED)に入り、その実行が中断している場合に、その状態を解除し、実行を再開させる。

対象タスクが待ち状態(WAITING)と強制待ち状態(SUSPENDED)の複合した二重待ち状態(WAITING-SUSPENDED)であった場合には、tk_rsm_tsk の実行により強制待ち状態(SUSPENDED)のみが解除され、対象タスクは待ち状態となる。([図 2.1.「タスク状態遷移図」]を参照)

本システムコールでは、自タスクを指定することはできない。自タスクを指定した場合には、E_OBJ のエラーとなる。

tk_rsm_tsk では、強制待ち要求のネスト(suscnt)を1回分だけ解除する。したがって、対象タスクに対して2回以上の tk_sus_tsk が発行されていた場合(suscnt≧2)には、tk_rsm_tsk の実行が終わった後も、対象タスクはまだ強制待ち状態(SUSPENDED)のままである。

補足事項

実行状態(RUNNING)もしくは実行可能状態(READY)のタスクが tk_sus_tsk により強制待ち状態(SUSPENDED)になった後、tk_rsm_tsk や tk_frsm_tsk によって実行を再開した場合、そのタスクは同じ優先度を持つタスクの中で最低の優先順位となる。

たとえば、同じ優先度のtask_Aとtask_Bに対して以下のシステムコールを実行した場合、次のような動作をする。

```
tk_sta_tsk (tskid＝task_A, stacd_A);
tk_sta_tsk (tskid＝task_B, stacd_B);
  /* この時はFCFSの原則により、優先順位はtask_A→task_Bとなっている */

tk_sus_tsk (tskid＝task_A);
tk_rsm_tsk (tskid＝task_A);
  /* この時に優先順位はtask_B→task_Aとなる */
```

4.2.8 tk_frsm_tsk - 強制待ち状態のタスクを強制再開

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_frsm_tsk (ID tskid);

パラメータ

ID	tskid	Task ID	タスクID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが強制待ち状態(SUSPENDED)ではない(自タスクや休止状態(DORMANT)の場合を含む))

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid で示されたタスクの強制待ち状態(SUSPENDED)を解除する。すなわち、対象タスクが以前に発行された tk_sus_tsk によって強制待ち状態(SUSPENDED)に入り、その実行が中断している場合に、その状態を解除し、実行を再開させる。

対象タスクが待ち状態(WAITING)と強制待ち状態(SUSPENDED)の複合した二重待ち状態(WAITING-SUSPENDED)であった場合には、tk_frsm_tsk の実行により強制待ち状態(SUSPENDED)のみが解除され、対象タスクは待ち状態となる。([図 2.1.「タスク状態遷移図」]を参照)

本システムコールでは、自タスクを指定することはできない。自タスクを指定した場合には、E_OBJ のエラーとなる。

tk_frsm_tsk では、強制待ち要求のネスト(suscnt)をすべて解除(suscnt＝0)する。したがって、対象タスクに対して2回以上の tk_sus_tsk が発行されていた場合(suscnt≧2)でも、それらの要求をすべて解除(suscnt＝0)する。すなわち、強制待ち状態(SUSPENDED)は必ず解除され、対象タスクが二重待ち状態(WAITING-SUSPENDED)でない限り実行を再開できる。

補足事項

実行状態(RUNNING)もしくは実行可能状態(READY)のタスクが tk_sus_tsk により強制待ち状態(SUSPENDED)になった後、tk_rsm_tsk や tk_frsm_tsk によって実行を再開した場合、そのタスクは同じ優先度を持つタスクの中で最低の優先順位となる。

たとえば、同じ優先度のtask_Aとtask_Bに対して以下のシステムコールを実行した場合、次のような動作をする。

```
tk_sta_tsk (tskid＝task_A, stacd_A);
tk_sta_tsk (tskid＝task_B, stacd_B);
  /* この時はFCFSの原則により、優先順位はtask_A→task_Bとなっている */

tk_sus_tsk (tskid＝task_A);
tk_frsm_tsk (tskid＝task_A);
  /* この時に優先順位はtask_B→task_Aとなる */
```

4.2.9 tk_dly_tsk - タスク遅延

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_dly_tsk (RELTIM dlytim);

パラメータ

RELTIM	dlytim	Delay Time	遅延時間(ミリ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(dlytim が不正)
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

自タスクの実行を一時的に停止し、時間経過待ちの状態に入る。タスクの実行を停止する時間は、dlytim により指定される。

時間経過待ちの状態は待ち状態の一つであり、tk_rel_wai により時間経過待ちを解除することができる。

このシステムコールを発行したタスクが強制待ち状態(SUSPENDED)または二重待ち状態(WAITING-SUSPENDED)になっている間も、時間経過のカウントは行われる。

dlytim の基準時間(時間の単位)はシステム時刻の基準時間(＝1ミリ秒)と同じである。

補足事項

このシステムコールは、tk_slp_tsk とは異なり、遅延時間として指定された時間が経過した場合に正常終了となる。また、遅延時間中に tk_wup_tsk が実行されても、待ち解除とはならない。遅延時間が経過する前に tk_dly_tsk が終了するのは、tk_ter_tsk や tk_rel_wai が発行された場合に限られる。

4.2.10 tk_dly_tsk_u - タスク遅延(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_dly_tsk_u (RELTIM_U dlytim_u);

パラメータ

RELTIM_U	dlytim_u	Delay Time	遅延時間(マイクロ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(dlytim_u が不正)
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_dly_tsk のパラメータである dlytim を64ビットマイクロ秒単位の dlytim_u としたシステムコールである。

パラメータが dlytim_u となった点を除き、本システムコールの仕様は tk_dly_tsk と同じである。詳細は tk_dly_tsk の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.2.11 tk_sig_tev - タスクイベントの送信

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_sig_tev (ID tskid , INT tskevt);

パラメータ

ID	tskid	Task ID	タスクID
INT	tskevt	Task Event	タスクイベント番号(1～8)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが休止状態(DORMANT))
E_PAR	パラメータエラー(tskevt が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

tskid のタスクへ tskevt で指定したタスクイベントを送信する。

タスクイベントは、タスクごとに保持される8種類のイベントであり、1～8の番号で指定する。

タスクイベントの発生回数は保持されず、発生の有無のみが保持される。

tskid＝TSK_SELF＝0によって自タスクの指定になる。ただし、タスク独立部から発行したシステムコールでtskid＝TSK_SELF＝0を指定した場合には、E_ID のエラーとなる。

補足事項

タスクイベントは、tk_slp_tsk～tk_wup_tsk と同様のタスク付属の同期機能であるが、次の点が異なる。

- 起床要求(タスクイベント)の回数は保持されない。
- 8種類のイベントタイプで起床要求を分類できる。

同一タスクにおいて、同じイベントタイプを使用して2ヵ所以上で同期を行うと混乱することになる。イベントタイプの割当ては明確に行うべきである。

タスクイベントは、ミドルウェアでの利用を前提とし、アプリケーションでは原則として使用しない。アプリケーションでは、tk_slp_tsk～tk_wup_tsk の利用を推奨する。

4.2.12 tk_wai_tev - タスクイベント待ち

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT tevptn = tk_wai_tev (INT waiptn , TMO tmout);

パラメータ

INT	waiptn	Wait Event Pattern	待ちタスクイベントのパターン
TMO	tmout	Timeout	タイムアウト指定(ミリ秒)

リターンパラメータ

INT	tevptn	Task Event Pattern	待ち解除時のタスクイベント発生状況
	または	Error Code	エラーコード

エラーコード

E_PAR	パラメータエラー(waiptn, tmout が不正)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

waiptn で指定したタスクイベントのいずれかが発生するまで待つ。タスクイベントの発生で待ちが解除された場合、waiptn で指定したタスクイベントはクリア(発生中のタスクイベント &= ~waiptn)される。また、戻値(tevptn)に待ちが解除された時に発生していたタスクイベントの状態(クリア前の状態)を返す。

waiptn および tevptn は、各タスクイベントを1<<(タスクイベント番号-1)のビット値として論理和(OR)した値である。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。待ち解除の条件が満足されない(tk_sig_tev が実行されない)まま tmout の時間が経過すると、タイムアウトエラー E_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1ミリ秒)と同じである。

tmout として TMO_POL＝0を指定した場合は、タイムアウト値として0を指定したことを示し、タスクイベントが発生していなくても待ちに入らず E_TMOUT を返す。また、tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずにタスクイベントが発生するまで待ち続ける。

4.2.13 tk_wai_tev_u - タスクイベント待ち(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT tevptn = tk_wai_tev_u (INT waiptn , TMO_U tmout_u);

パラメータ

INT	waiptn	Wait Event Pattern	待ちタスクイベントのパターン
TMO_U	tmout_u	Timeout	タイムアウト指定(マイクロ秒)

リターンパラメータ

INT	tevptn	Task Event Pattern	待ち解除時のタスクイベント発生状況
	または	Error Code	エラーコード

エラーコード

E_PAR	パラメータエラー(waiptn, tmout_u が不正)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_wai_tev のパラメータである tmout を64ビットマイクロ秒単位の tmout_u としたシステムコールである。

パラメータが tmout_u となった点を除き、本システムコールの仕様は tk_wai_tev と同じである。詳細は tk_wai_tev の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.2.14 tk_dis_wai - タスク待ち状態の禁止

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT tskwait = tk_dis_wai (ID tskid , UINT waitmask);

パラメータ

ID	tskid	Task ID	タスクID
UINT	waitmask	Wait Mask	タスク待ち禁止設定

リターンパラメータ

INT	tskwait	Task Wait	タスク待ち禁止後のタスク待ち状態
	または	Error Code	エラーコード

エラーコード

E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(waitmask が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid のタスクに対して、waitmask で指定した待ち要因による待ちに入ることを禁止する。タスクがすでに waitmask で指定した待ち要因の待ちに入っている場合は、その待ちを解除する。

waitmask には、以下の待ち要因を任意に組み合わせて論理和(OR)で指定する。

```
#define TTW_SLP         0x00000001      /* 起床待ちによる待ち */
#define TTW_DLY         0x00000002      /* タスクの遅延による待ち */
#define TTW_SEM         0x00000004      /* セマフォ待ち */
#define TTW_FLG         0x00000008      /* イベントフラグ待ち */
#define TTW_MBX         0x00000040      /* メールボックス待ち */
#define TTW_MTX         0x00000080      /* ミューテックス待ち */
#define TTW_SMBF        0x00000100      /* メッセージバッファ送信待ち */
#define TTW_RMBF        0x00000200      /* メッセージバッファ受信待ち */
#define TTW_CAL         0x00000400      /* ランデブ呼出待ち */
#define TTW_ACP         0x00000800      /* ランデブ受付待ち */
#define TTW_RDV         0x00001000      /* ランデブ終了待ち */
#define TTW_MPF         0x00002000      /* 固定長メモリプール待ち */
#define TTW_MPL         0x00004000      /* 可変長メモリプール待ち */
#define TTW_EV1         0x00010000      /* タスクイベント#１待ち */
#define TTW_EV2         0x00020000      /* タスクイベント#２待ち */
#define TTW_EV3         0x00040000      /* タスクイベント#３待ち */
#define TTW_EV4         0x00080000      /* タスクイベント#４待ち */
```

```
#define TTW_EV5         0x00100000      /* タスクイベント#5待ち */
#define TTW_EV6         0x00200000      /* タスクイベント#6待ち */
#define TTW_EV7         0x00400000      /* タスクイベント#7待ち */
#define TTW_EV8         0x00800000      /* タスクイベント#8待ち */
#define TTX_SVC         0x80000000      /* 拡張SVC呼出禁止 */
```

TTX_SVC はタスクの待ちではないが、拡張SVCの呼出を禁止する特殊な指定となる。TTX_SVC が指定されている場合、そのタスクが拡張SVCを呼び出そうとすると、拡張SVCを呼び出さずに E_DISWAI を返す。すでに呼び出している拡張SVCを終了させるような効果はない。

戻値(tskwait)には、tk_dis_wai によって待ち禁止処理が行われた後のタスクの待ち状態を返す。この値は、tk_ref_tsk の tskwait と同じ値となる。なお、tskwait には TTX_SVC に関する情報は返されない。tskwait が0であれば、タスクは待ち状態に入っていない(または待ちが解除された)ことを示す。tskwait が0でなければ、waitmask に指定した以外の要因で待っていることになる。

タスクの待ちが tk_dis_wai により解除された場合、または待ち禁止状態で新たに待ちに入ろうとした場合は、E_DISWAI が返される。

待ち禁止状態でその待ちに入る可能性があるシステムコールを呼び出したとき、待ちに入らずに処理できる場合であっても E_DISWAI が返される。例えば、メッセージバッファに空きがあり、待ちに入ることなく送信できるような状況で、メッセージバッファへ送信(tk_snd_mbf)を行った場合も、メッセージの送信は行われずに E_DISWAI が返される。

拡張SVCの実行中に設定された待ち禁止は、拡張SVCから呼出元に戻る際に自動的に解除される。また、拡張SVCを呼び出した時も自動的に解除され、拡張SVCから戻ってきたときに元の設定に復帰する。

タスクが休止状態(DORMANT)に戻るときにも、自動的に解除される。ただし、休止状態(DORMANT)であるときの設定は有効であり、次にタスクが起動したときはその待ち禁止が適用される。

セマフォなど主なオブジェクトでは、オブジェクト生成時に TA_NODISWAI を指定することができる。TA_NODISWAI を指定して生成されたオブジェクトでは、tk_dis_wai による待ち禁止は拒否され、待ちは禁止されない。

tskid＝TSK_SELF＝0によって自タスクの指定になる。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF＝0を指定した場合には、E_ID のエラーとなる。

補足事項

待ち禁止機能は、拡張SVCハンドラの実行を中止するために用意された機能で、一般的にはブレーク関数で使用される。(その用途に限定されるものではない。)

ランデブの場合の待ち禁止は、他に比べて複雑である。基本的には、ランデブの待ち状態が変化するときに待ち禁止が検出されると待ちが解除される。

いくつか具体例を挙げる。

TTW_CAL の待ちが禁止されておらず、TTW_RDV の待ちが禁止されている場合、ランデブ呼出の待ちには入るが、ランデブが受け付けられてランデブ終了待ちになる時点で待ちが解除され E_DISWAI を返す。このとき、メッセージは受付タスクに送られ、受付タスクはメッセージを受け取りランデブ成立状態となる。受付タスクは返答(tk_rpl_rdv)を行った時点で相手タスクがないことがわかりエラー(E_OBJ)となる。

ランデブの回送の場合も待ち禁止は適用される。このとき、回送先のランデブポートの属性に従うことになる。つまり、回送先のランデブポートに TA_NODISWAI 属性の指定があれば、待ち禁止は拒否される。

ランデブ終了待ち状態となったあと TTW_CAL の待ちが禁止された場合、その状態で回送を行うと、回送によっていったんランデブ呼出待ちの状態になるため、TTW_CAL の指定により待ちが禁止される。このとき、ランデブ呼出側(tk_cal_por)と回送側(tk_fwd_por)の双方に E_DISWAI が返される。

4.2.15 tk_ena_wai - タスク待ち禁止の解除

C言語インタフェース

```
#include <tk/tkernel.h>

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

パラメータ

ID	tskid	Task ID	タスクID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid のタスクに tk_dis_wai によって設定された待ち禁止をすべて解除する。

tskid＝TSK_SELF＝0によって自タスクの指定になる。ただし、タスク独立部から発行したシステムコールでtskid＝TSK_SELF＝0を指定した場合には、E_ID のエラーとなる。

4.3 タスク例外処理機能

タスク例外処理機能は、タスクに発生した例外事象に対する処理を、タスクのコンテキストで行うための機能である。

タスク例外ハンドラが起動されるのは、次の条件がすべて揃った時である。

1. tk_def_tex によるタスク例外ハンドラの登録
2. tk_ena_tex によるタスク例外の許可
3. tk_ras_tex によるタスク例外の発生

タスク例外ハンドラは、タスク例外が発生したタスクのコンテキストで、かつタスク生成時に指定した保護レベルで、そのタスクの一部として実行される。また、タスク例外ハンドラ内では、タスク例外に関する状態を除き、通常のタスク部の実行中と同じ状態であり、使用できるシステムコールも同等である。

タスク例外ハンドラは、対象タスクがタスク部を実行しているときにのみ起動される。対象タスクがタスク部以外を実行中のときにタスク例外が発生した場合は、タスク部へ戻ってくるまで待ってタスク例外ハンドラが起動される。準タスク部(拡張SVC)を実行中にタスク例外が発生した場合は、その拡張SVCに対応するブレーク関数が呼び出される。ブレーク関数によって拡張SVCの処理が中止されタスク部へ戻ってくることになる。

発生したタスク例外要求は、タスク例外ハンドラが呼び出された(タスク例外ハンドラが実行を開始した)時点でクリアされる。

タスク例外は、0～31の32種類のタスク例外コードにより指定され、0が最も優先度が高く31が低い。また、タスク例外コード0は特殊な扱いとなる。

タスク例外コード1～31の動作:

- タスク例外ハンドラはネストして実行されない。タスク例外ハンドラの実行中に発生したタスク例外はペンディングされる。(タスク例外コード0の場合を除く)
- タスク例外ハンドラからリターンすることで、タスク例外によって割り込まれた位置に復帰する。
- タスク例外ハンドラからリターンせずに、`longjmp()` などによりタスク内の任意の位置にジャンプすることもできる。

タスク例外コード0の動作:

- タスク例外コード1～31のタスク例外ハンドラを実行中でもネストして実行される。タスク例外コード0のタスク例外ハンドラの実行中はネストされない。
- ユーザスタックポインタをタスク起動時の初期値に設定してから、タスク例外ハンドラが実行される。ただし、ユーザスタックとシステムスタックが分離されていないシステムでは、スタックポインタは初期値に戻されない。
- タスク例外ハンドラから復帰することは出来ない。必ず tk_ext_tsk または tk_exd_tsk によってタスクを終了しなければならない。

4.3.1 tk_def_tex - タスク例外ハンドラの定義

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_def_tex (ID tskid , CONST T_DTEX *pk_dtex);

パラメータ

ID	tskid	Task ID	タスクID
CONST T_DTEX*	pk_dtex	Packet to Define Task Exception	タスク例外ハンドラ定義情報

pk_dtex の内容

ATR	texatr	Task Exception Attribute	タスク例外ハンドラ属性
FP	texhdr	Task Exception Handler	タスク例外ハンドラアドレス

――(以下に実装独自に他の情報を追加してもよい)――

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(tskid のタスクは TA_RNG0 属性)
E_RSATR	予約属性(texatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_dtex が不正あるいは利用できない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid で指定したタスクに対するタスク例外ハンドラを定義する。タスク例外ハンドラはタスクに対して1つのみ定義可能で、すでに定義されていた場合は、後から定義した関数が有効となる。pk_dtex＝NULL の時は定義を解除する。

タスク例外ハンドラを定義または解除すると、ペンディングされているタスク例外要求はクリアされ、すべてのタスク例外は禁止状態となる。

texatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。texatr のシステム属性には、現在のバージョンでは割当てがなく、システム属性は使われていない。

タスク例外ハンドラは、下記のような形式になる。

```
void texhdr( INT texcd )
{
        /*
                タスク例外の処理
        */

        /* タスク例外ハンドラの終了 */
        if ( texcd == 0 ) {
                tk_ext_tsk() または tk_exd_tsk();
        } else {
                tk_end_tex();
                return または longjmp();
        }
}
```

タスク例外ハンドラは、TA_ASM 属性相当のみで高級言語対応ルーチンを経由しての呼出は行われない。したがって、タスク例外ハンドラのエントリー部分はアセンブリ言語で作成する必要がある。カーネル提供者は、上記のC言語のタスク例外ハンドラを呼び出すためのエントリールーチンのアセンブリ言語のソースコードを提供しなければならない。つまり、高級言語対応ルーチンに相当するソースコードを提供しなければならない。

タスク生成時の保護レベルが TA_RNG0 のタスクに対して、タスク例外を使用することはできない。

補足事項

タスク生成時にはタスク例外ハンドラは定義されておらず、タスク例外も禁止されている。

タスクが休止状態(DORMANT)に戻る時には自動的にタスク例外ハンドラは解除され、タスク例外は禁止される。また、ペンディングされていたタスク例外はクリアされる。ただし、休止状態(DORMANT)であるときに、タスク例外ハンドラを定義することはできる。

タスク例外は、tk_ras_tex によって発生するソフトウェア的なもので、CPUの例外と直接の関連はない。

4.3.2 tk_ena_tex - タスク例外の許可

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ena_tex (ID tskid , UINT texptn);

パラメータ

ID	tskid	Task ID	タスクID
UINT	texptn	Task Exception Pattern	タスク例外パターン

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない、タスク例外ハンドラが定義されていない)
E_PAR	パラメータエラー(texptn が不正あるいは利用できない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid のタスクへのタスク例外の発生を許可する。

texptn は、タスク例外コードを1<<タスク例外コードのビット値として、論理和(OR)した値である。

tk_ena_tex は、texptn で指定したタスク例外を許可する。現在のタスク例外の許可状態を texmask とすると次のようになる。

許可:texmask |= texptn

texptn の全ビットを0とした場合は、texmask に対して何の操作も行わないことになる。ただし、この場合でもエラーとはならない。

タスク例外ハンドラが定義されていないタスクのタスク例外を許可することはできない。

休止状態(DORMANT)のタスクに対しても適用される。

4.3.3 tk_dis_tex - タスク例外の禁止

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_dis_tex (ID tskid , UINT texptn);

パラメータ

ID	tskid	Task ID	タスクID
UINT	texptn	Task Exception Pattern	タスク例外パターン

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない、タスク例外ハンドラが定義されていない)
E_PAR	パラメータエラー(texptn が不正あるいは利用できない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid のタスクへのタスク例外の発生を禁止する。

texptn は、タスク例外コードを1<<タスク例外コードのビット値として、論理和(OR)した値である。

tk_dis_tex は、texptn で指定したタスク例外を禁止する。現在のタスク例外の許可状態を texmask とすると次のようになる。

禁止:texmask &= ~texptn

texptn の全ビットを0とした場合は、texmask に対して何の操作も行わないことになる。ただし、この場合でもエラーとはならない。

禁止されているタスク例外は無視され、ペンディングもされない。ペンディングされているタスク例外がある状態で、そのタスク例外が禁止された場合は、タスク例外要求が捨てられる(ペンディング状態がクリアされる)。

休止状態(DORMANT)のタスクに対しても適用される。

4.3.4 tk_ras_tex - タスク例外を発生

C言語インタフェース

```
#include <tk/tkernel.h>

ER ercd = tk_ras_tex (ID tskid , INT texcd );
```

パラメータ

ID	tskid	Task ID	タスクID
INT	texcd	Task Exception Code	タスク例外コード(0～31)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない、タスク例外ハンドラが定義されていない)
E_OBJ	オブジェクトの状態が不正(tskid のタスクは休止状態(DORMANT))
E_PAR	パラメータエラー(texcd が不正あるいは利用できない)
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid のタスクに対して texcd のタスク例外を発生させる。ただし、tskid のタスクにおいて texcd のタスク例外が許可されていない場合は、発生させたタスク例外は無視され、ペンディングもされない。この場合でも、本システムコールには E_OK が返される。

tskid のタスクがタスク例外ハンドラの実行中であれば、タスク例外はペンディングされる。ペンディングされる場合、対象タスクが拡張SVCの実行中であってもブレーク関数は実行されない。

ただし、texcd＝0の場合は、対象タスクが例外ハンドラを実行中であってもペンディングされない。対象タスクが、タスク例外コード1～31の例外に対するタスク例外ハンドラの実行中であればタスク例外は受け付けられ、拡張SVC実行中であればブレーク関数が呼び出される。対象タスクが、タスク例外コード0の例外に対するタスク例外ハンドラを実行中の場合は、タスク例外の発生は無視される。

tskid＝TSK_SELF＝0によって自タスクの指定を行うことができる。

タスク独立部から発行することはできない。(E_CTX)

補足事項

対象タスクが拡張SVCを実行中の場合には、その拡張SVCに対応するブレーク関数が、tk_ras_tex の発行タスクの準タスク部として実行される。すなわち、ブレーク関数の実行されるコンテキストは、tk_ras_tex の発行タスクを要求タスクとする準タスク部である。

したがって、このような場合、ブレーク関数の実行が終わるまで tk_ras_tex から戻ってこない。このため、タスク独立部から tk_ras_tex を発行することはできない仕様になっている。

また、ブレーク関数実行中に tk_ras_tex を呼び出したタスクに発生したタスク例外は、ブレーク関数の終了まで保留される。

4.3.5 tk_end_tex - タスク例外ハンドラの終了

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT texcd = tk_end_tex (BOOL enatex);

パラメータ

BOOL	enatex	Enable Task Exception	タスク例外ハンドラ呼出の許可

リターンパラメータ

INT	texcd	Task Exception Code または Error Code	発生している例外コード(0～31) エラーコード

エラーコード

E_CTX	コンテキストエラー(タスク例外ハンドラ以外またはタスク例外コード0(検出は実装依存))

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

タスク例外ハンドラを終了し、新たなタスク例外ハンドラの呼出を許可する。ペンディングされているタスク例外がある場合は、その内の最も優先度の高いタスク例外コードを戻値に返す。ペンディングされているタスク例外がなければ0を返す。

enatex＝FALSE の場合、ペンディングされているタスク例外があると新たなタスク例外ハンドラの呼出は許可されない。この場合、tk_end_tex から戻った時点で、戻値に返された texcd の例外ハンドラを実行している状態になっている。ペンディングされているタスク例外がない場合は新たなタスク例外ハンドラの呼出が許可される。

enatex＝TRUE の場合は、ペンディングされているタスク例外の有無に関係なく、新たなタスク例外ハンドラの呼出を許可する。ペンディングされているタスク例外があっても、タスク例外ハンドラを終了した状態になる。

タスク例外ハンドラの終了は tk_end_tex を呼び出すこと以外にはない。タスク例外ハンドラが起動されてから tk_end_tex を呼び出すまでがタスク例外ハンドラの実行中となる。tk_end_tex を呼び出さずにタスク例外ハンドラからリターンしたとしても、リターン先はまだタスク例外ハンドラの実行中となる。同様に、tk_end_tex を呼び出さずに longjmp によりタスク例外ハンドラから抜けたとしても、そのジャンプ先はタスク例外ハンドラの実行中である。

タスク例外がペンディングされている状態で tk_end_tex を呼び出すことにより、ペンディングされていたタスク例外が新たに受け付けられる。この時、tk_end_tex を拡張SVCハンドラ内から呼び出した場合でも、tk_end_tex を呼び出した拡張SVCハンドラに対するブレーク関数は呼び出されない。拡張SVCをネストして呼び出していた場合は、拡張SVCのネストが1段浅くなるときに戻先の拡張SVCに対応するブレーク関数が呼び出される。タスク例外ハンドラが呼び出されるのは、タスク部に戻ってからとなる。

タスク例外コード0の場合、タスク例外ハンドラを終了することはできないため、tk_end_tex を発行することはできない。必ず tk_ext_tsk または tk_exd_tsk でタスクを終了しなければならない。タスク例外コード0を処理中に tk_end_tex を発行した場合の動作は不定(実装依存)である。

タスク例外ハンドラ以外から発行することはできない。タスク例外ハンドラ以外から発行した場合の動作は不定(実装依存)である。

補足事項

tk_end_tex(TRUE) とすると、ペンディングされたタスク例外がある場合、tk_end_tex の直後にさらにタスク例外ハンドラが呼び出されることになる。そのため、スタックが戻されないままにタスク例外ハンドラが呼び出されることとなり、スタックオーバーフローの可能性がある。

通常は tk_end_tex(FALSE)を利用し、次のようにタスク例外が残っている間繰り返し処理するようにするとよい。

```
void texhdr( INT texcd )
{
  if ( texcd == 0 ){
    /*
      タスク例外0用の処理
    */
    tk_exd_tsk();
  }

  do {
    /*
      タスク例外1～31用の処理
    */
  } while ( (texcd = tk_end_tex(FALSE)) > 0 );
}
```

厳密には、tk_end_tex で0が返されループを終了してから texhdr を抜けるまでの間にタスク例外が発生した場合には、スタックが戻されずに texhdr に再入する可能性はある。しかし、タスク例外はソフトウェア的なものであり、タスクの実行と無関係に発生することは通常ないため、実用上は問題ないと思われる。

4.3.6 tk_ref_tex - タスク例外の状態参照

C言語インタフェース

```
#include <tk/tkernel.h>

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

パラメータ

ID	tskid	Task ID	タスクID
T_RTEX*	pk_rtex	Packet to Refer Task Exception Status	タスク例外状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rtex の内容

UINT	pendtex	Pending Task Exception	発生しているタスク例外
UINT	texmask	Task Exception Mask	許可されているタスク例外

――(以下に実装独自に他の情報を追加してもよい)――

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_PAR	パラメータエラー(pk_rtex が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tskid で示された対象タスクのタスク例外の状態を参照する。

pendtex は現在発生しているタスク例外を示す。タスク例外の発生からタスク例外ハンドラが呼び出されるまでの間、pendtex に示される。

texmask は許可されているタスク例外を示す。

pendtex, texmask とも、1<<タスク例外コードのビット値として表した値である。

tskid＝TSK_SELF＝0によって自タスクの指定を行うことができる。ただし、タスク独立部から発行したシステムコールで tskid＝TSK_SELF＝0を指定した場合には、E_ID のエラーとなる。

4.4 同期・通信機能

同期・通信機能は、タスクとは独立したオブジェクトにより、タスク間の同期・通信を行うための機能である。セマフォ、イベントフラグ、メールボックスの各機能が含まれる。

4.4.1 セマフォ

セマフォは、使用されていない資源の有無や数量を数値で表現することにより、その資源を使用する際の排他制御や同期を行うためのオブジェクトである。セマフォ機能には、セマフォを生成／削除する機能、セマフォの資源を獲得／返却する機能、セマフォの状態を参照する機能が含まれる。セマフォはID番号で識別されるオブジェクトである。セマフォのID番号をセマフォIDと呼ぶ。

セマフォは、対応する資源の有無や数量を表現する資源数と、資源の獲得を待つタスクの待ち行列を持つ。資源をm個返却する側(イベントを知らせる側)では、セマフォの資源数をm個増やす。一方、資源をn個獲得する側(イベントを待つ側)では、セマフォの資源数をn個減らす。セマフォの資源数が足りなくなった場合(具体的には、資源数を減らすと資源数が負になる場合)、資源を獲得しようとしたタスクは、次に資源が返却されるまでセマフォ資源の獲得待ち状態となる。セマフォ資源の獲得待ち状態になったタスクは、そのセマフォの待ち行列につながれる。

また、セマフォに対して資源が返却され過ぎるのを防ぐために、セマフォ毎に最大資源数を設定することができる。最大資源数を越える資源がセマフォに返却されようとした場合(具体的には、セマフォの資源数を増やすと最大資源数を越える場合)には、エラーを報告する。

4.4.1.1 tk_cre_sem - セマフォ生成

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID semid = tk_cre_sem (CONST T_CSEM *pk_csem);

パラメータ

CONST T_CSEM*	pk_csem	Packet to Create Semaphore	セマフォ生成情報

pk_csem の内容

void*	exinf	Extended Information	拡張情報
ATR	sematr	Semaphore Attribute	セマフォ属性
INT	isemcnt	Initial Semaphore Count	セマフォの初期値
INT	maxsem	Maximum Semaphore Count	セマフォの最大値
UB	dsname[8]	DS Object name	DSオブジェクト名称

――(以下に実装独自に他の情報を追加してもよい)――

リターンパラメータ

ID	semid	Semaphore ID	セマフォID
	または	Error Code	エラーコード

エラーコード

E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_LIMIT	セマフォの数がシステムの上限を超えた
E_RSATR	予約属性(sematr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_csem が不正, isemcnt, maxsem が負または不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

セマフォを生成しセマフォID番号を割り当てる。具体的には、生成するセマフォに対して管理ブロックを割り付け、その初期値を isemcnt、最大値(上限値)を maxsem とする。なお、maxsem には少なくとも65535が指定できなくてはならない。65536以上の値が指定できるかは実装に依存する。

exinf は、対象セマフォに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、tk_ref_sem で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。カーネルでは exinf の内容について関知しない。

sematr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。sematr のシステム属性の部分では、次のような指

定を行う。

sematr:= (TA_TFIFO || TA_TPRI) | (TA_FIRST || TA_CNT) | [TA_DSNAME] | [TA_NODISWAI]

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_FIRST	待ち行列先頭のタスクを優先
TA_CNT	要求数の少ないタスクを優先
TA_DSNAME	DSオブジェクト名称を指定する
TA_NODISWAI	tk_dis_wai による待ち禁止を拒否する

TA_TFIFO, TA_TPRI では、タスクがセマフォの待ち行列に並ぶ際の並び方を指定することができる。属性が TA_TFIFO であればタスクの待ち行列はFIFOとなり、属性が TA_TPRI であればタスクの待ち行列はタスクの優先度順となる。

TA_FIRST, TA_CNT では、資源獲得の優先順を指定する。TA_FIRST および TA_CNT の指定によって待ち行列の並び順が変わることはない。待ち行列の並び順は TA_TFIFO, TA_TPRI によってのみ決定される。

TA_FIRST では、要求カウントに関係なく待ち行列の先頭のタスクから順に資源を割り当てる。待ち行列の先頭のタスクが要求分の資源を獲得できない限り、待ち行列の後ろのタスクが資源を獲得することはない。

TA_CNT では、要求カウント分の資源が獲得できるタスクから順に割り当てる。具体的には、待ち行列の先頭のタスクから順に要求カウント数を検査し、要求カウント数分の資源が割り当てられるタスクに割り当てる。要求カウント数の少ない順に割り当てる訳ではない。

TA_DSNAME を指定した場合に dsname が有効となり、DSオブジェクト名称として設定される。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DSのシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname 、td_set_dsname を参照のこと。TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

```
#define TA_TFIFO        0x00000000      /* 待ちタスクをFIFOで管理 */
#define TA_TPRI         0x00000001      /* 待ちタスクを優先度順で管理 */
#define TA_FIRST        0x00000000      /* 待ち行列先頭のタスクを優先 */
#define TA_CNT          0x00000002      /* 要求数の少ないタスクを優先 */
#define TA_DSNAME       0x00000040      /* DSオブジェクト名称を指定 */
#define TA_NODISWAI     0x00000080      /* 待ち禁止拒否 */
```

4.4.1.2 tk_del_sem - セマフォ削除

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_del_sem (ID semid);

パラメータ

ID	semid	Semaphore ID	セマフォID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(semid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(semid のセマフォが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

semid で示されたセマフォを削除する。

本システムコールの発行により、対象セマフォのID番号および管理ブロック用の領域は解放される。

対象セマフォにおいて条件成立を待っているタスクがあった場合にも、本システムコールは正常終了するが、待ち状態にあったタスクにはエラー E_DLT が返される。

4.4.1.3 tk_sig_sem - セマフォ資源返却

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_sig_sem (ID semid , INT cnt);

パラメータ

ID	semid	Semaphore ID	セマフォID
INT	cnt	Count	資源返却数

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(semid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(semid のセマフォが存在しない)
E_QOVR	キューイングまたはネストのオーバーフロー(カウント値 semcnt のオーバーフロー)
E_PAR	パラメータエラー(cnt≦0)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

semid で示されたセマフォに対して、cnt 個の資源を返却する操作を行う。対象セマフォに対して既に待っているタスクがあれば、要求カウントを確認して可能であれば資源を割り当てる。資源を割り当てられたタスクを実行可能状態(READY)に移す。条件によっては、複数のタスクに資源が割り当てられ実行可能状態(READY)になる場合がある。

セマフォのカウント値の増加により、その値がセマフォの最大値(maxsem)を越えようとした場合は、E_QOVR のエラーとなる。この場合、資源の返却は一切行われずカウント値(semcnt)も変化しない。

補足事項

カウント値(semcnt)がセマフォ初期値(isemcnt)を越えた場合にも、エラーとはならない。排他制御ではなく、同期の目的(tk_wup_tsk～tk_slp_tsk と同様)でセマフォを使用する場合には、セマフォのカウント値(semcnt)が初期値(isemcnt)を越えることがある。一方、排他制御の目的でセマフォを使う場合は、セマフォ初期値(isemcnt)とセマフォ最大値(maxsem)を等しい値にしておくことにより、カウント値の増加によるエラーをチェックすることができる。

4.4.1.4 tk_wai_sem - セマフォ資源獲得

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_wai_sem (ID semid , INT cnt , TMO tmout);

パラメータ

ID	semid	Semaphore ID	セマフォID
INT	cnt	Count	資源要求数
TMO	tmout	Timeout	タイムアウト指定(ミリ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(semid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(semid のセマフォが存在しない)
E_PAR	パラメータエラー(tmout≦(-2), cnt≦0)
E_DLT	待ちオブジェクトが削除された(待ちの間に対象セマフォが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

semid で示されたセマフォから、cnt 個の資源を獲得する操作を行う。資源が獲得できれば、本システムコールの発行タスクは待ち状態に入らず、実行を継続する。この場合、そのセマフォのカウント値(semcnt)は cnt 分減算される。資源が獲得できなければ、本システムコールを発行したタスクは待ち状態に入る。すなわち、そのセマフォに対する待ち行列につながれる。この場合、そのセマフォのカウント値(semcnt)は不変である。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。待ち解除の条件が満足されない(tk_sig_sem が実行されない)まま tmout の時間が経過すると、タイムアウトエラー E_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1ミリ秒)と同じである。

tmout として TMO_POL＝0を指定した場合は、タイムアウト値として0を指定したことを示し、資源を獲得できなくても待ちに入らず E_TMOUT を返す。また、tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずに資源が獲得できるまで待ち続ける。

4.4.1.5 tk_wai_sem_u - セマフォ資源獲得(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_wai_sem_u (ID semid , INT cnt , TMO_U tmout_u);

パラメータ

ID	semid	Semaphore ID	セマフォID
INT	cnt	Count	資源要求数
TMO_U	tmout_u	Timeout	タイムアウト指定(マイクロ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(semid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(semid のセマフォが存在しない)
E_PAR	パラメータエラー(tmout_u≦(-2), cnt≦0)
E_DLT	待ちオブジェクトが削除された(待ちの間に対象セマフォが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_wai_sem のパラメータである tmout を64ビットマイクロ秒単位の tmout_u としたシステムコールである。

パラメータが tmout_u となった点を除き、本システムコールの仕様は tk_wai_sem と同じである。詳細は tk_wai_sem の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.4.1.6 tk_ref_sem - セマフォ状態参照

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_sem (ID semid , T_RSEM *pk_rsem);

パラメータ

ID	semid	Semaphore ID	セマフォID
T_RSEM*	pk_rsem	Packet to Refer Semaphore Status	セマフォ状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rsem の内容

void*	exinf	Extended Information	拡張情報
ID	wtsk	Waiting Task ID	待ちタスクのID
INT	semcnt	Semaphore Count	現在のセマフォカウント値

――(以下に実装独自に他の情報を追加してもよい)――

エラーコード

E_OK	正常終了
E_ID	不正ID番号(semid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(semid のセマフォが存在しない)
E_PAR	パラメータエラー(pk_rsem が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

semid で示された対象セマフォの各種の状態を参照し、リターンパラメータとして現在のセマフォカウント値(semcnt)、待ちタスクのID(wtsk)、拡張情報(exinf) を返す。

wtsk は、このセマフォで待っているタスクのIDを示す。複数のタスクが待っている場合には、待ち行列の先頭のタスクのIDを返す。待ちタスクが無い場合は wtsk＝0となる。

対象セマフォが存在しない場合には、エラー E_NOEXS となる。

4.4.2 イベントフラグ

イベントフラグは、イベントの有無をビット毎のフラグで表現することにより、同期を行うためのオブジェクトである。イベントフラグ機能には、イベントフラグを生成／削除する機能、イベントフラグをセット／クリアする機能、イベントフラグで待つ機能、イベントフラグの状態を参照する機能が含まれる。イベントフラグはID番号で識別されるオブジェクトである。イベントフラグのID番号をイベントフラグIDと呼ぶ。

イベントフラグは、対応するイベントの有無をビット毎に表現するビットパターンと、そのイベントフラグで待つタスクの待ち行列を持つ。イベントフラグのビットパターンを、単にイベントフラグと呼ぶ場合もある。イベントを知らせる側では、イベントフラグのビットパターンの指定したビットをセットないしはクリアすることが可能である。一方、イベントを待つ側では、イベントフラグのビットパターンの指定したビットのすべてまたはいずれかがセットされるまで、タスクをイベントフラグ待ち状態にすることができる。イベントフラグ待ち状態になったタスクは、そのイベントフラグの待ち行列につながれる。

4.4.2.1 tk_cre_flg - イベントフラグ生成

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID flgid = tk_cre_flg (CONST T_CFLG *pk_cflg);

パラメータ

CONST T_CFLG*	pk_cflg	Packet to Create EventFlag	イベントフラグ生成情報

pk_cflg の内容

void*	exinf	Extended Information	拡張情報
ATR	flgatr	EventFlag Attribute	イベントフラグ属性
UINT	iflgptn	Initial EventFlag Pattern	イベントフラグの初期値
UB	dsname[8]	DS Object name	DSオブジェクト名称
――(以下に実装独自に他の情報を追加してもよい)――			

リターンパラメータ

ID	flgid	EventFlag ID	イベントフラグID
	または	Error Code	エラーコード

エラーコード

E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_LIMIT	イベントフラグの数がシステムの上限を超えた
E_RSATR	予約属性(flgatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_cflg が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

イベントフラグを生成しイベントフラグID番号を割り当てる。具体的には、生成するイベントフラグに対して管理ブロックを割り付け、その初期値を iflgptn とする。一つのイベントフラグで、プロセッサの1ワード分のビットをグループ化して扱う。操作はすべて1ワード分を単位とする。

exinf は、対象イベントフラグに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、tk_ref_flg で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。カーネルでは exinf の内容について関知しない。

flgatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。flgatr のシステム属性の部分では、次のような指定を行う。

```
flgatr:= (TA_TFIFO || TA_TPRI) | (TA_WMUL || TA_WSGL) | [TA_DSNAME] | [TA_NODISWAI]
```

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(Wait Single Task)
TA_WMUL	複数タスクの待ちを許す(Wait Multiple Task)
TA_DSNAME	DSオブジェクト名称を指定する
TA_NODISWAI	tk_dis_wai による待ち禁止を拒否する

TA_WSGL を指定した場合は、複数のタスクが同時に待ち状態になることを禁止する。TA_WMUL を指定した場合は、同時に複数のタスクが待ち状態となることが許される。

TA_TFIFO, TA_TPRI では、タスクがイベントフラグの待ち行列に並ぶ際の並び方を指定することができる。属性が TA_TFIFO であればタスクの待ち行列はFIFOとなり、属性が TA_TPRI であればタスクの待ち行列はタスクの優先度順となる。ただし、TA_WSGL を指定した場合は待ち行列を作らないため、TA_TFIFO, TA_TPRI のどちらを指定しても動作に変わりはない。

複数のタスクが待っている場合、待ち行列の先頭から順に待ち条件が成立しているか検査し、待ち条件が成立しているタスクの待ちを解除する。したがって、待ち行列の先頭のタスクが必ずしも最初に待ちが解除される訳ではない。また、待ち条件が成立したタスクが複数あれば、複数のタスクの待ちが解除される。

TA_DSNAME を指定した場合に dsname が有効となり、DSオブジェクト名称として設定される。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DSのシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname 、td_set_dsname を参照のこと。TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

```
#define TA_TFIFO        0x00000000      /* 待ちタスクをFIFOで管理 */
#define TA_TPRI         0x00000001      /* 待ちタスクを優先度順で管理 */
#define TA_WSGL         0x00000000      /* 複数タスクの待ちを許さない */
#define TA_WMUL         0x00000008      /* 複数タスクの待ちを許す */
#define TA_DSNAME       0x00000040      /* DSオブジェクト名称を指定 */
#define TA_NODISWAI     0x00000080      /* 待ち禁止拒否 */
```

4.4.2.2 tk_del_flg - イベントフラグ削除

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_del_flg (ID flgid);

パラメータ

ID	flgid	EventFlag ID	イベントフラグID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(flgid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(flgid のイベントフラグが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

flgid で示されたイベントフラグを削除する。

本システムコールの発行により、対象イベントフラグのID番号および管理ブロック用の領域は解放される。

対象イベントフラグにおいて条件成立を待っているタスクがあった場合にも、本システムコールは正常終了するが、待ち状態にあったタスクにはエラー E_DLT が返される。

4.4.2.3 tk_set_flg - イベントフラグのセット

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_set_flg (ID flgid , UINT setptn);

パラメータ

ID	flgid	EventFlag ID	イベントフラグID
UINT	setptn	Set Bit Pattern	セットするビットパターン

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(flgid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(flgid のイベントフラグが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

tk_set_flg では、flgid で示される1ワードのイベントフラグのうち、setptn で示されているビットがセットされる。すなわち、flgid で示されるイベントフラグの値に対して、setptn の値で論理和がとられる。(イベントフラグ値 flgptn に対して flgptn |= setptn の処理を実行)

tk_set_flg によりイベントフラグの値が変更された結果、tk_wai_flg でそのイベントフラグを待っていたタスクの待ち解除の条件を満たすようになれば、そのタスクの待ち状態が解除され、実行状態(RUNNING)または実行可能状態(READY)(待っていたタスクが二重待ち状態(WAITING-SUSPENDED)であった場合には強制待ち状態(SUSPENDED))へと移行する。

tk_set_flg で setptn の全ビットを0とした場合には、対象イベントフラグに対して何の操作も行わないことになる。ただし、この場合でもエラーとはならない。

TA_WMUL の属性を持つイベントフラグに対しては、同一のイベントフラグに対して複数のタスクが同時に待つことができる。したがって、イベントフラグでもタスクが待ち行列を作ることになる。この場合、一回の tk_set_flg で複数のタスクが待ち解除となることがある。

4.4.2.4 tk_clr_flg - イベントフラグのクリア

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_clr_flg (ID flgid , UINT clrptn);

パラメータ

ID	flgid	EventFlag ID	イベントフラグID
UINT	clrptn	Clear Bit Pattern	クリアするビットパターン

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(flgid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(flgid のイベントフラグが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_clr_flg では、flgid で示される1ワードのイベントフラグのうち、対応する clrptn の0になっているビットがクリアされる。すなわち、flgid で示されるイベントフラグの値に対して、clrptn の値で論理積がとられる。(イベントフラグ値 flgptn に対して flgptn &= clrptn の処理を実行)

tk_clr_flg では、対象イベントフラグを待っているタスクが待ち解除となることはない。すなわち、ディスパッチは起らない。

tk_clr_flg で clrptn の全ビットを1とした場合には、対象イベントフラグに対して何の操作も行わないことになる。ただし、この場合でもエラーとはならない。

4.4.2.5 tk_wai_flg - イベントフラグ待ち

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_wai_flg (ID flgid , UINT waiptn , UINT wfmode , UINT *p_flgptn , TMO tmout);

パラメータ

ID	flgid	EventFlag ID	イベントフラグID
UINT	waiptn	Wait Bit Pattern	待ちビットパターン
UINT	wfmode	Wait EventFlag Mode	待ちモード
UINT*	p_flgptn	Pointer to EventFlag Bit Pattern	リターンパラメータ flgptn を返す領域へのポインタ
TMO	tmout	Timeout	タイムアウト指定(ミリ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード
UINT	flgptn	EventFlag Bit Pattern	待ち解除時のビットパターン

エラーコード

E_OK	正常終了
E_ID	不正ID番号(flgid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(flgid のイベントフラグが存在しない)
E_PAR	パラメータエラー(waiptn＝0, wfmode が不正, tmout≦(-2))
E_OBJ	オブジェクトの状態が不正(TA_WSGL 属性のイベントフラグに対する複数タスクの待ち)
E_DLT	待ちオブジェクトが削除された(待ちの間に対象イベントフラグが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_wai_flg では、wfmode で示される待ち解除の条件にしたがって、flgid で示されるイベントフラグがセットされるのを待つ。

flgid で示されるイベントフラグが、既に wfmode で示される待ち解除条件を満たしている場合には、発行タスクは待ち状態にならずに実行を続ける。

wfmode では、次のような指定を行う。

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

TWF_ANDW	0x00	AND待ち

TWF_ORW	0x01	OR待ち
TWF_CLR	0x10	全クリア指定
TWF_BITCLR	0x20	条件ビットのみクリア指定

TWF_ORW を指定した場合には、flgid で示される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つ(OR待ち)。また、TWF_ANDW を指定した場合には、flgid で示されるイベントフラグのうち、waiptn で指定したビットのすべてがセットされるのを待つ(AND待ち)。

TWF_CLR の指定が無い場合には、条件が満足されてこのタスクが待ち解除となった場合にも、イベントフラグの値はそのままである。TWF_CLR の指定がある場合には、条件が満足されてこのタスクが待ち解除となった場合、イベントフラグの値(全部のビット)が0にクリアされる。TWF_BITCLR の指定がある場合には、条件が満足されてこのタスクが待ち解除となった場合、イベントフラグの待ち解除条件に一致したビットのみが0クリアされる。(イベントフラグ値 &= ~待ち解除条件)

flgptn は、本システムコールによる待ち状態が解除される時のイベントフラグの値(TWF_CLR または TWF_BITCLR 指定の場合は、イベントフラグがクリアされる前の値) を示すリターンパラメータである。flgptn で返る値は、このシステムコールの待ち解除の条件を満たすものになっている。なお、タイムアウト等で待ちが解除された場合は、flgptn の内容は不定となる。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。待ち解除の条件が満足されないまま tmout の時間が経過すると、タイムアウトエラー E_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1ミリ秒)と同じである。

tmout として TMO_POL＝0を指定した場合は、タイムアウト値として0を指定したことを示し、条件を満たしていなくても待ちに入らず E_TMOUT を返す。また、tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずに条件が成立するまで待ち続ける。

タイムアウトした場合は、TWF_CLR または TWF_BITCLR の指定があってもイベントフラグのクリアは行われない。

waiptn を0とした場合は、パラメータエラー E_PAR になる。

既に待ちタスクの存在する TA_WSGL 属性のイベントフラグに対して、別のタスクが tk_wai_flg を実行することはできない。この場合は、後から tk_wai_flg を実行したタスクが待ち状態に入るかどうか(待ち解除条件が満たされているかどうか)にかかわらず、後から tk_wai_flg を実行したタスクは E_OBJ のエラーとなる。

一方、TA_WMUL の属性を持つイベントフラグに対しては、同一のイベントフラグに対して複数のタスクが同時に待つことができる。したがって、イベントフラグでもタスクが待ち行列を作ることになる。この場合、一回の tk_set_flg で複数のタスクが待ち解除となることがある。

TA_WMUL 属性を持つイベントフラグに対して複数のタスクが待ち行列を作った場合、次のような動作をする。

- 待ち行列の順番はFIFOまたはタスク優先度順である。(ただし、waiptn や wfmode との関係により、必ずしも行列先頭のタスクから待ち解除になるとは限らない。)
- 待ち行列中にクリア指定のタスクがあれば、そのタスクが待ち解除になる時に、フラグをクリアする。
- クリア指定を行っていたタスクよりも後ろの待ち行列にあったタスクは、既にクリアされた後のイベントフラグを見ることになる。

tk_set_flg によって同じ優先度を持つ複数のタスクが同時に待ち解除となる場合、待ち解除後のタスクの優先順位は、元のイベントフラグの待ち行列の順序を保存する。

補足事項

tk_wai_flg の待ち解除条件として全ビットの論理和を指定すれば、(waiptn＝0xfff...ff, wfmode＝TWF_ORW)、tk_set_flg と組み合わせることによって、1ワードのビットパターンによるメッセージ転送を行うことができる。ただし、この場合、全部のビットが0というメッセージを送ることはできない。また、前のメッセージが tk_wai_flg で読まれる前に tk_set_flg により次のメッセージが送られると、前のメッセージは消えてしまう。すなわち、メッセージのキューイングはできない。

waiptn＝0の指定は E_PAR のエラーになるので、イベントフラグで待つタスクの waiptn は0でないことが保証されている。したがって、tk_set_flg で全ビットをセットすれば、どのような条件でイベントフラグを待つタスクであっても、待ち行列の先頭にあるタスクは必ず待ち解除となる。

イベントフラグに対する複数タスク待ちの機能は、次のような場合に有効である。例えば、タスクAが(1)の tk_set_flg を実行するまで、タスクB、タスクCを(2), (3)の tk_wai_flg で待たせておく場合に、イベントフラグの複数待ちが可能であれば、(1)(2)(3)のどのシステムコールが先に実行されても結果は同じになる[図 4.1.「イベントフラグに対する複数タスク待ちの機能」]。一方、イベントフラグの複数待ちができなければ、(2)→(3)→(1)の順でシステムコールが実行された場合に、(3)の tk_wai_flg が E_OBJ のエラーになる。

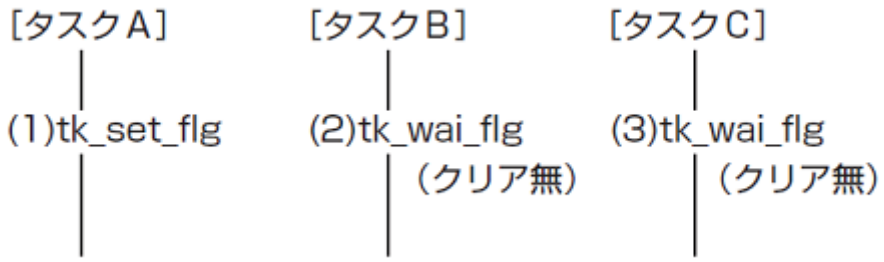

図 4.1: イベントフラグに対する複数タスク待ちの機能

仕様決定の理由

`waiptn`＝0の指定を E_PAR のエラーとしたのは、`waiptn`＝0の指定を許した場合に、それ以後イベントフラグがどのような値に変わっても待ち状態から抜けることができなくなるためである。

4.4.2.6 tk_wai_flg_u - イベントフラグ待ち(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_wai_flg_u (ID flgid , UINT waiptn , UINT wfmode , UINT *p_flgptn , TMO_U tmout_u);

パラメータ

ID	flgid	EventFlag ID	イベントフラグID
UINT	waiptn	Wait Bit Pattern	待ちビットパターン
UINT	wfmode	Wait EventFlag Mode	待ちモード
UINT*	p_flgptn	Pointer to EventFlag Bit Pattern	リターンパラメータ flgptn を返す領域へのポインタ
TMO_U	tmout_u	Timeout	タイムアウト指定(マイクロ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード
UINT	flgptn	EventFlag Bit Pattern	待ち解除時のビットパターン

エラーコード

E_OK	正常終了
E_ID	不正ID番号(flgid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(flgid のイベントフラグが存在しない)
E_PAR	パラメータエラー(waiptn＝0, wfmode が不正, tmout_u≦(-2))
E_OBJ	オブジェクトの状態が不正(TA_WSGL 属性のイベントフラグに対する複数タスクの待ち)
E_DLT	待ちオブジェクトが削除された(待ちの間に対象イベントフラグが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_wai_flg のパラメータである tmout を64ビットマイクロ秒単位の tmout_u としたシステムコールである。

パラメータが tmout_u となった点を除き、本システムコールの仕様は tk_wai_flg と同じである。詳細は tk_wai_flg の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.4.2.7 tk_ref_flg - イベントフラグ状態参照

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_flg (ID flgid , T_RFLG *pk_rflg);

パラメータ

ID	flgid	EventFlag ID	イベントフラグID
T_RFLG*	pk_rflg	Packet to Refer EventFlag Status	イベントフラグ状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rflg の内容

void*	exinf	Extended Information	拡張情報
ID	wtsk	Waiting Task ID	待ちタスクのID
UINT	flgptn	EventFlag Bit Pattern	現在のイベントフラグのビットパターン

――(以下に実装独自に他の情報を追加してもよい)――

エラーコード

E_OK	正常終了
E_ID	不正ID番号(flgid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(flgid のイベントフラグが存在しない)
E_PAR	パラメータエラー(pk_rflg が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

flgid で示された対象イベントフラグの各種の状態を参照し、リターンパラメータとして現在のフラグ値(flgptn)、待ちタスクのID(wtsk)、拡張情報(exinf) を返す。

wtsk は、このイベントフラグで待っているタスクのIDを示す。このイベントフラグで複数のタスクが待っている場合(TA_WMUL属性のときのみ)には、待ち行列の先頭のタスクのIDを返す。待ちタスクが無い場合は wtsk＝0となる。

対象イベントフラグが存在しない場合には、エラー E_NOEXS となる。

4.4.3 メールボックス

メールボックスは、共有メモリ上に置かれたメッセージを受渡しすることにより、同期と通信を行うためのオブジェクトである。メールボックス機能には、メールボックスを生成／削除する機能、メールボックスに対してメッセージを送信／受信する機能、メールボックスの状態を参照する機能が含まれる。メールボックスはID番号で識別されるオブジェクトである。メールボックスのID番号をメールボックスIDと呼ぶ。

メールボックスは、送信されたメッセージを入れるためのメッセージキューと、メッセージの受信を待つタスクの待ち行列を持つ。メッセージを送信する側(イベントを知らせる側)では、送信したいメッセージをメッセージキューに入れる。一方、メッセージを受信する側(イベントを待つ側)では、メッセージキューに入っているメッセージを一つ取り出す。メッセージキューにメッセージが入っていない場合は、次にメッセージが送られてくるまでメールボックスからの受信待ち状態になる。メールボックスからの受信待ち状態になったタスクは、そのメールボックスの待ち行列につながれる。

メールボックスによって実際に送受信されるのは、送信側と受信側で共有しているメモリ上に置かれたメッセージの先頭番地のみである。すなわち、送受信されるメッセージの内容のコピーは行わない。カーネルは、メッセージキューに入っているメッセージを、リンクリストにより管理する。アプリケーションプログラムは、送信するメッセージの先頭に、カーネルがリンクリストに用いるための領域を確保しなければならない。この領域をメッセージヘッダと呼ぶ。また、メッセージヘッダと、それに続くアプリケーションがメッセージを入れるための領域をあわせて、メッセージパケットと呼ぶ。メールボックスへメッセージを送信するシステムコールは、メッセージパケットの先頭番地(`pk_msg`)をパラメータとする。

また、メールボックスからメッセージを受信するシステムコールは、メッセージパケットの先頭番地をリターンパラメータとして返す。

メッセージキューをメッセージの優先度順にする場合には、メッセージの優先度を入れるための領域(`msgpri`)も、メッセージヘッダ中に持つ必要がある。[図 4.2.「メールボックスで使用されるメッセージの形式」]

ユーザが実際にメッセージを入れることができるのは、メッセージ先頭アドレスの直後からではなく、メッセージヘッダの後の部分から(図のメッセージの内容の部分) である。

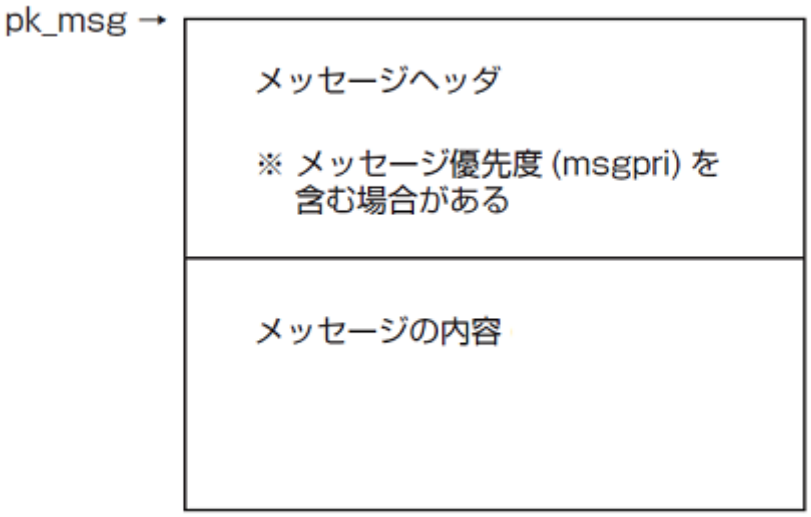

図 4.2: メールボックスで使用されるメッセージの形式

カーネルは、メッセージキューに入っている(ないしは、入れようとしている)メッセージのメッセージヘッダ(メッセージ優先度のための領域を除く)の内容を書き換える。一方、アプリケーションは、メッセージキューに入っているメッセージのメッセージヘッダ(メッセージ優先度のための領域を含む)の内容を書き換えてはならない。アプリケーションによってメッセージヘッダの内容が書き換えられた場合の振舞いは未定義である。この規定は、アプリケーションプログラムがメッセージヘッダの内容を直接書き換えた場合に加えて、メッセージヘッダの番地をカーネルに渡し、カーネルにメッセージヘッダの内容を書き換えさせた場合にも適用される。したがって、すでにメッセージキューに入っているメッセージを再度メールボックスに送信した場合の振舞いは未定義となる。

補足事項
メールボックス機能では、メッセージヘッダの領域をアプリケーションプログラムで確保することとしているため、メッセージキューに入れることができるメッセージの数には上限がない。また、メッセージを送信するシステムコールで待ち状態になることもない。
メッセージパケットとしては、固定長メモリプールまたは可変長メモリプールから動的に確保したメモリブロックを用いることも、静的に確保した領域を用いることも可能であるが、タスク固有空間上に置くことはできない。
一般的な使い方としては、送信側のタスクがメモリプールからメモリブロックを確保し、それをメッセージパケットとして送信し、受信側のタスクはメッセージの内容を取り出した後にそのメモリブロックを直接メモリプールに返却するという手順をとることが多い。
メモリプール管理機能によって管理されるメモリは共有空間のメモリである。
上記のような使い方をする場合のプログラム例を以下に示す。

```
/* メッセージの型定義 */
typedef struct  {
  T_MSG msgque;     /* メッセージヘッダ、T_MFIFO属性の場合 */
  UB  msgcont[MSG_SIZE];  /* メッセージ内容 */
} T_MSG_PACKET;
```

```
/* メモリブロックの獲得とメッセージの送信を行うタスクの処理 */

  T_MSG_PACKET  *pk_msg;
    ...

  /* 固定長メモリプールからメモリブロックを獲得 */
  /* 固定長メモリブロックサイズが sizeof(T_MSG_PACKET) 以上であること */
  tk_get_mpf( mpfid, (void**)&pk_msg, TMO_FEVR );

  /* pk_msg -> msgcont[] 以下にメッセージを作成 */
    ...

  /* メッセージを送信 */
  tk_snd_mbx( mbxid, (T_MSG*)pk_msg );
```

```
/* メッセージの受信とメモリブロックの解放を行うタスクの処理 */

  T_MSG_PACKET  *pk_msg;
    ...

  /* メッセージを受信 */
  tk_rcv_mbx( mbxid, (T_MSG**)&pk_msg, TMO_FEVR );

  /* pk_msg -> msgcont[] 以下のメッセージ内容の確認、それに応じた処理 */
    ...

  /* 固定長メモリプールにメモリブロックを返却 */
  tk_rel_mpf( mpfid, (void*)pk_msg );
```

4.4.3.1 tk_cre_mbx - メールボックス生成

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID mbxid = tk_cre_mbx (CONST T_CMBX *pk_cmbx);

パラメータ

CONST T_CMBX*	pk_cmbx	Packet to Create Mailbox	メールボックス生成情報

pk_cmbx の内容

void*	exinf	Extended Information	拡張情報
ATR	mbxatr	Mailbox Attribute	メールボックス属性
UB	dsname[8]	DS Object name	DSオブジェクト名称
——(以下に実装独自に他の情報を追加してもよい)——			

リターンパラメータ

ID	mbxid	Mailbox ID	メールボックスID
	または	Error Code	エラーコード

エラーコード

E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_LIMIT	メールボックスの数がシステムの上限を超えた
E_RSATR	予約属性(mbxatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_cmbx が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

メールボックスを生成しメールボックスID番号を割り当てる。具体的には、生成するメールボックスに対して管理ブロックなどを割り付ける。

exinf は、対象メールボックスに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、tk_ref_mbx で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。カーネルでは exinf の内容について関知しない。

mbxatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。mbxatr のシステム属性の部分では、次のような指定を行う。

```
mbxatr:= (TA_TFIFO || TA_TPRI) | (TA_MFIFO || TA_MPRI) | [TA_DSNAME] | [TA_NODISWAI]
```

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
TA_DSNAME	DSオブジェクト名称を指定する
TA_NODISWAI	tk_dis_wai による待ち禁止を拒否する

TA_TFIFO, TA_TPRI では、メッセージを受信するタスクがメールボックスの待ち行列に並ぶ際の並び方を指定することができる。属性が TA_TFIFO であればタスクの待ち行列はFIFOとなり、属性が TA_TPRI であればタスクの待ち行列はタスクの優先度順となる。

一方、TA_MFIFO, TA_MPRI では、メッセージがメッセージキュー(受信されるのを待つメッセージの待ち行列) に入る際の並び方を指定することができる。属性が TA_MFIFO であればメッセージキューはFIFOとなり、属性が TA_MPRI であればメッセージキューはメッセージの優先度順となる。メッセージの優先度は、メッセージパケットの中の特定領域で指定する。メッセージ優先度は正の値で、1が最も優先度が高く、数値が大きくなるほど優先度は低くなる。PRI型で表せる最大の正の値が最も低い優先度となる。同一優先度の場合はFIFOとなる。

TA_DSNAME を指定した場合に dsname が有効となり、DSオブジェクト名称として設定される。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DSのシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname 、td_set_dsname を参照のこと。TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

```
#define TA_TFIFO        0x00000000      /* 待ちタスクをFIFOで管理 */
#define TA_TPRI         0x00000001      /* 待ちタスクを優先度順で管理 */
#define TA_MFIFO        0x00000000      /* メッセージをFIFOで管理 */
#define TA_MPRI         0x00000002      /* メッセージを優先度順で管理 */
#define TA_DSNAME       0x00000040      /* DSオブジェクト名称を指定 */
#define TA_NODISWAI     0x00000080      /* 待ち禁止拒否 */
```

補足事項

メールボックスで送受信されるメッセージの本体は共有メモリ上に置かれており、実際に送受信されるのはその先頭アドレスのみである。そのため、タスク固有空間上にメッセージを置くことはできない。

4.4.3.2 tk_del_mbx - メールボックス削除

C言語インタフェース

```
#include <tk/tkernel.h>

ER ercd = tk_del_mbx (ID mbxid );
```

パラメータ

ID	mbxid	Mailbox ID	メールボックスID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mbxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbxid のメールボックスが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

mbxid で示されたメールボックスを削除する。

本システムコールの発行により、対象メールボックスのID番号および管理ブロック用の領域などが解放される。

対象メールボックスにおいてメッセージを待っているタスクがあった場合にも、本システムコールは正常終了するが、待ち状態にあったタスクにはエラー E_DLT が返される。また、対象メールボックスの中にメッセージが残っている場合でも、エラーとはならず、メールボックスの削除が行われる。

4.4.3.3 tk_snd_mbx - メールボックスへ送信

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_snd_mbx (ID mbxid , T_MSG *pk_msg);

パラメータ

ID	mbxid	Mailbox ID	メールボックスID
T_MSG*	pk_msg	Packet of Message	メッセージパケットの先頭アドレス

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mbxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbxid のメールボックスが存在しない)
E_PAR	パラメータエラー(pk_msg が不正, msgpri≦0)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

mbxid で示された対象メールボックスに、pk_msg を先頭アドレスとするメッセージパケットを送信する。

メッセージパケットの内容はコピーされず、受信時には先頭アドレス(pk_msg の値)のみが渡される。したがって、このメッセージを受信したタスクがメッセージパケットの内容を取り出すまで、メッセージパケットの内容を書き換えてはいけない。

対象メールボックスで既にメッセージを待っているタスクがあった場合には、待ち行列の先頭のタスクの待ち状態が解除され、tk_snd_mbx で指定した pk_msg がそのタスクに送信されて、tk_rcv_mbx のリターンパラメータとなる。一方、対象メールボックスでメッセージを待っているタスクが無ければ、送信されたメッセージは、メールボックスの中のメッセージキュー(メッセージの待ち行列)に入れられる。どちらの場合にも、tk_snd_mbx 発行タスクは待ち状態とはならない。

pk_msg は、メッセージヘッダを含めたメッセージパケットの先頭アドレスである。メッセージヘッダは次の形式となる。

```
typedef struct t_msg {
        ?       ?               /* 内容は実装依存(ただし、固定長) */
} T_MSG;

typedef struct t_msg_pri {
        T_MSG   msgque;         /* メッセージキューのためのエリア */
        PRI     msgpri;         /* メッセージ優先度 */
} T_MSG_PRI;
```

メッセージヘッダは、TA_MFIFO の場合は T_MSG、TA_MPRI の場合は T_MSG_PRI となる。いずれの場合も、メッセージヘッダのサイズは固定長で、sizeof(T_MSG)またはsizeof(T_MSG_PRI)で取得する。

実際にメッセージを格納できるのは、メッセージヘッダより後ろの領域となる。メッセージ本体部分のサイズには制限はなく、可変長でもよい。

補足事項

tk_snd_mbx によるメッセージ送信は、受信側のタスクの状態とは無関係に行われる。すなわち、非同期のメッセージ送信が行われる。待ち行列につながれるのは、そのタスクの発行したメッセージであって、タスクそのものではない。すなわち、メッセージの待ち行列(メッセージキュー)や受信タスクの待ち行列は存在するが、送信タスクの待ち行列は存在しない。

メールボックスで送受信されるメッセージの本体は共有メモリ上に置かれており、実際に送受信されるのはその先頭アドレスのみである。そのため、タスク固有空間上にメッセージを置くことはできない。

4.4.3.4 tk_rcv_mbx - メールボックスから受信

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_rcv_mbx (ID mbxid , T_MSG **ppk_msg , TMO tmout);

パラメータ

ID	mbxid	Mailbox ID	メールボックスID
T_MSG**	ppk_msg	Pointer to Packet of Message	リターンパラメータ pk_msg を返す領域へのポインタ
TMO	tmout	Timeout	タイムアウト指定(ミリ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード
T_MSG*	pk_msg	Packet of Message	メッセージパケットの先頭アドレス

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mbxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbxid のメールボックスが存在しない)
E_PAR	パラメータエラー(tmout≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象メールボックスが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_rcv_mbx では、mbxid で示されたメールボックスからメッセージを受信する。

対象メールボックスにまだメッセージが送信されていない場合(メッセージキューが空の場合) には、本システムコールを発行したタスクは待ち状態となり、メッセージの到着を待つ待ち行列につながれる。一方、対象メールボックスに既にメッセージが入っている場合には、メッセージキューの先頭にあるメッセージを1つ取り出して、それをリターンパラメータ pk_msg とする。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。タイムアウト指定が行われた場合、待ち解除の条件が満足されない(メッセージが到着しない)まま tmout の時間が経過すると、タイムアウトエラー E_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1ミリ秒)と同じである。

tmout として TMO_POL＝0を指定した場合は、タイムアウト値として0を指定したことを示し、メッセージがない場合も待ちに入らず E_TMOUT を返す。また、tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずにメッセージが到着するまで待ち続ける。

補足事項

pk_msg は、メッセージヘッダを含めたメッセージパケットの先頭アドレスである。メッセージヘッダは TA_MFIFO の場合は T_MSG、TA_MPRI の場合は T_MSG_PRI となる。

メールボックスで送受信されるメッセージの本体は共有メモリ上に置かれており、実際に送受信されるのはその先頭アドレスのみである。そのため、タスク固有空間上にメッセージを置くことはできない。

4.4.3.5 tk_rcv_mbx_u - メールボックスから受信(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_rcv_mbx_u (ID mbxid , T_MSG **ppk_msg , TMO_U tmout_u);

パラメータ

ID	mbxid	Mailbox ID	メールボックスID
T_MSG**	ppk_msg	Pointer to Packet of Message	リターンパラメータ pk_msg を返す領域へのポインタ
TMO_U	tmout_u	Timeout	タイムアウト指定(マイクロ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード
T_MSG*	pk_msg	Packet of Message	メッセージパケットの先頭アドレス

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mbxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbxid のメールボックスが存在しない)
E_PAR	パラメータエラー(tmout_u≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象メールボックスが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_rcv_mbx のパラメータである tmout を64ビットマイクロ秒単位の tmout_u としたシステムコールである。

パラメータが tmout_u となった点を除き、本システムコールの仕様は tk_rcv_mbx と同じである。詳細は tk_rcv_mbx の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.4.3.6 tk_ref_mbx - メールボックス状態参照

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_mbx (ID mbxid , T_RMBX *pk_rmbx);

パラメータ

ID	mbxid	Mailbox ID	メールボックスID
T_RMBX*	pk_rmbx	Packet to Refer Mailbox Status	メールボックス状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rmbx の内容

void*	exinf	Extended Information	拡張情報
ID	wtsk	Waiting Task ID	待ちタスクのID
T_MSG*	pk_msg	Packet of Message	次に受信されるメッセージ

――(以下に実装独自に他の情報を追加してもよい)――

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mbxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbxid のメールボックスが存在しない)
E_PAR	パラメータエラー(pk_rmbx が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

mbxid で示された対象メールボックスの各種の状態を参照し、リターンパラメータとして次に受信されるメッセージ(メッセージキューの先頭のメッセージ)、待ちタスクのID(wtsk)、拡張情報(exinf) を返す。

wtsk は、このメールボックスで待っているタスクのIDを示す。このメールボックスで複数のタスクが待っている場合には、待ち行列の先頭のタスクのIDを返す。待ちタスクが無い場合は wtsk＝0となる。

対象メールボックスが存在しない場合には、エラー E_NOEXS となる。

pk_msg は、次に tk_rcv_mbx を実行した場合に受信されるメッセージである。メッセージキューにメッセージが無い時は、pk_msg＝NULL となる。また、どんな場合でも、pk_msg＝NULL と wtsk＝0の少なくとも一方は成り立つ。

4.5 拡張同期・通信機能

拡張同期・通信機能は、タスクとは独立したオブジェクトにより、タスク間の高度な同期・通信を行うための機能である。ミューテックス、メッセージバッファ、ランデブの各機能が含まれる。

4.5.1 ミューテックス

ミューテックスは、共有資源を使用する際にタスク間で排他制御を行うためのオブジェクトである。ミューテックスは、排他制御に伴う上限のない優先度逆転を防ぐための機構として、優先度継承プロトコル(priority inheritance protocol)と優先度上限プロトコル(priority ceiling protocol)をサポートする。

ミューテックス機能には、ミューテックスを生成／削除する機能、ミューテックスをロック／ロック解除する機能、ミューテックスの状態を参照する機能が含まれる。ミューテックスはID番号で識別されるオブジェクトである。ミューテックスのID番号をミューテックスIDと呼ぶ。

ミューテックスは、ロックされているかどうかの状態と、ロックを待つタスクの待ち行列を持つ。また、カーネルは、各ミューテックスに対してそれをロックしているタスクを、各タスクに対してそれがロックしているミューテックスの集合を管理する。タスクは、資源を使用する前に、ミューテックスをロックする。ミューテックスが他のタスクにロックされていた場合には、ミューテックスがロック解除されるまで、ミューテックスのロック待ち状態となる。ミューテックスのロック待ち状態になったタスクは、そのミューテックスの待ち行列につながれる。タスクは、資源の使用を終えると、ミューテックスのロックを解除する。

ミューテックスは、ミューテックス属性に TA_INHERIT(＝0x02)を指定することにより優先度継承プロトコルを、TA_CEILING(＝0x03)を指定することにより優先度上限プロトコルをサポートする。TA_CEILING 属性のミューテックスに対しては、そのミューテックスをロックする可能性のあるタスクの中で最も高いベース優先度を持つタスクのベース優先度を、ミューテックス生成時に上限優先度として設定する。TA_CEILING 属性のミューテックスを、その上限優先度よりも高いベース優先度を持つタスクがロックしようとした場合、E_ILUSE エラーとなる。また、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タスクのベース優先度を、tk_chg_pri によってそのミューテックスの上限優先度よりも高く設定しようとした場合、tk_chg_pri が E_ILUSE エラーを返す。

これらのプロトコルを用いた場合、上限のない優先度逆転を防ぐために、ミューテックスの操作に伴ってタスクの現在優先度が自動的に変更される。優先度継承プロトコルと優先度上限プロトコルに厳密に従うなら、タスクの現在優先度を、次に挙げる優先度の最高値に常に一致するように変更する必要がある。これを、厳密な優先度制御規則と呼ぶ。

- タスクのベース優先度
- タスクが TA_INHERIT 属性のミューテックスをロックしている場合、それらのミューテックスのロックを待っているタスクの中で、最も高い現在優先度を持つタスクの現在優先度
- タスクが TA_CEILING 属性のミューテックスをロックしている場合、それらのミューテックス中で、最も高い上限優先度を持つミューテックスの上限優先度

ここで、TA_INHERIT 属性のミューテックスを待っているタスクの現在優先度が、ミューテックス操作か tk_chg_pri によるベース優先度の変更に伴って変更された場合、そのミューテックスをロックしているタスクの現在優先度の変更が必要になる場合がある。これを推移的な優先度継承と呼ぶ。さらにそのタスクが、別の TA_INHERIT 属性のミューテックスを待っていた場合には、そのミューテックスをロックしているタスクに対して推移的な優先度継承の処理が必要になる場合がある。

T-Kernelでは、上述の厳密な優先度制御規則に加えて、現在優先度を変更する状況を限定した優先度制御規則(これを簡略化した優先度制御規則と呼ぶ)を規定し、どちらを採用するかは実装定義とする。具体的には、簡略化した優先度制御規則においては、タスクの現在優先度を高くする方向の変更はすべて行うのに対して、現在優先度を低くする方向の変更は、タスクがロックしているミューテックスがなくなった時にのみ行う(この場合には、タスクの現在優先度をベース優先度に戻すことになる)。より具体的には、次の状況でのみ現在優先度を変更する処理を行えばよい。

- タスクがロックしている TA_INHERIT 属性のミューテックスを、そのタスクよりも高い現在優先度を持つタスクが待ち始めた時
- タスクAによってロックされている TA_INHERIT 属性のミューテックスを待っている別のタスクBが、タスクAよりも高い現在優先度に変更された時
- タスクが、そのタスクの現在優先度よりも高い上限優先度を持つ TA_CEILING 属性のミューテックスをロックした時
- タスクがロックしているミューテックスがなくなった時

ミューテックスの操作に伴ってタスクの現在優先度を変更した場合には、次の処理を行う。

優先度を変更されたタスクが実行できる状態である場合、タスクの優先順位を、変更後の優先度にしたがって変化させる。変更後の優先度と同じ優先度を持つタスクの間での優先順位は、実装依存である。優先度が変更されたタスクが何らかのタスク優先度順の待ち行列につながれている場合にも、その待ち行列の中での順序を、変更後の優先度にしたがって変化させる。

変更後の優先度と同じ優先度を持つタスクの間での順序は、実装依存である。タスクが終了する時に、そのタスクがロックしているミューテックスが残っている場合には、それらのミューテックスをすべてロック解除する。ロックしているミューテックスが複数ある場合には、それらをロック解除する順序は実装依存である。ロック解除の具体的な処理内容については、tk_unl_mtx の機能説明を参照すること。

補足事項

TA_TFIFO 属性または TA_TPRI 属性のミューテックスは、最大資源数が1のセマフォ(バイナリセマフォ)と同等の機能を持つ。ただし、ミューテックスは、ロックしたタスク以外はロック解除できない、タスク終了時に自動的にロック解除されるなどの違いがある。

ここでいう優先度上限プロトコルは、広い意味での優先度上限プロトコルで、最初に優先度上限プロトコルとして提案されたアルゴリズムではない。厳密には、highest locker protocolなどと呼ばれているアルゴリズムである。

ミューテックスの操作に伴ってタスクの現在優先度を変更した結果、優先度を変更されたタスクのタスク優先度順の待ち行列の中での順序が変化した場合、優先度を変更されたタスクないしはその待ち行列で待っている他のタスクの待ち解除が必要になる場合がある。

仕様決定の理由

ミューテックスの操作に伴ってタスクの現在優先度を変更した場合に、変更後の優先度と同じ優先度を持つタスクの間での優先順位を実装依存としたのは、次の理由による。アプリケーションによっては、ミューテックス機能による現在優先度の変更が頻繁に発生する可能性があり、それに伴ってタスク切替えが多発するのは望ましくない(同じ優先度を持つタスクの間での優先順位を最低とすると、不必要なタスク切替えが起こる)。理想的には、タスクの優先度ではなく優先順位を継承するのが望ましいが、このような仕様にすると実装上のオーバヘッドが大きくなるため、実装依存とすることにした。

4.5.1.1 tk_cre_mtx - ミューテックス生成

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID mtxid = tk_cre_mtx (CONST T_CMTX *pk_cmtx);

パラメータ

CONST T_CMTX*	pk_cmtx	Packet to Create Mutex	ミューテックス生成情報

pk_cmtx の内容

void*	exinf	Extended Information	拡張情報
ATR	mtxatr	Mutex Attribute	ミューテックス属性
PRI	ceilpri	Ceiling Priority of Mutex	ミューテックスの上限優先度
UB	dsname[8]	DS Object name	DSオブジェクト名称

――(以下に実装独自に他の情報を追加してもよい)――

リターンパラメータ

ID	mtxid	Mutex ID	ミューテックスID
	または	Error Code	エラーコード

エラーコード

E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_LIMIT	ミューテックスの数がシステムの上限を超えた
E_RSATR	予約属性(mtxatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_cmtx, ceilpri が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ミューテックスを生成しミューテックスID番号を割り当てる。具体的には、生成するミューテックスに対して管理ブロックなどを割り付ける。

exinf は、対象ミューテックスに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、tk_ref_mtx で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。カーネルでは exinf の内容について関知しない。

mtxatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。mtxatr のシステム属性の部分では、次のような指定を行う。

```
mtxatr:= (TA_TFIFO || TA_TPRI || TA_INHERIT || TA_CEILING) | [TA_DSNAME] | [TA_NODISWAI]
```

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_INHERIT	優先度継承プロトコル
TA_CEILING	優先度上限プロトコル
TA_DSNAME	DSオブジェクト名称を指定する
TA_NODISWAI	tk_dis_wai による待ち禁止を拒否する

TA_TFIFO の場合、ミューテックスのタスクの待ち行列はFIFOとなる。TA_TPRI, TA_INHERIT, TA_CEILING では、タスクの優先度順となる。TA_INHERIT では優先度継承プロトコル、TA_CEILING では優先度上限プロトコルが適用される。

TA_CEILING の場合のみ ceilpri が有効となり、ミューテックスの上限優先度を設定する。

TA_DSNAME を指定した場合に dsname が有効となり、DSオブジェクト名称として設定される。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DSのシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname 、td_set_dsname を参照のこと。TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

```
#define TA_TFIFO        0x00000000      /* 待ちタスクをFIFOで管理 */
#define TA_TPRI         0x00000001      /* 待ちタスクを優先度順で管理 */
#define TA_INHERIT      0x00000002      /* 優先度継承プロトコル */
#define TA_CEILING      0x00000003      /* 優先度上限プロトコル */
#define TA_DSNAME       0x00000040      /* DSオブジェクト名称を指定 */
#define TA_NODISWAI     0x00000080      /* 待ち禁止拒否 */
```

4.5.1.2 tk_del_mtx - ミューテックス削除

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_del_mtx (ID mtxid);

パラメータ

ID	mtxid	Mutex ID	ミューテックスID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mtxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mtxid のミューテックスが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

mtxid で示されたミューテックスを削除する。

本システムコールの発行により、対象ミューテックスのID番号および管理ブロック用の領域は解放される。

対象ミューテックスにおいてロック待ちしているタスクがあった場合にも、本システムコールは正常終了するが、待ち状態にあったタスクにはエラー E_DLT が返される。

ミューテックスが削除されると、そのミューテックスをロックしているタスクにとっては、ロックしているミューテックスが減ることになる。したがって、削除されるミューテックスが TA_INHERIT または TA_CEILING 属性の場合には、ロックしていたタスクの優先度が変更される場合がある。

4.5.1.3 tk_loc_mtx - ミューテックスのロック

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_loc_mtx (ID mtxid , TMO tmout);

パラメータ

ID	mtxid	Mutex ID	ミューテックスID
TMO	tmout	Timeout	タイムアウト指定(ミリ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mtxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mtxid のミューテックスが存在しない)
E_PAR	パラメータエラー(tmout≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象ミューテックスが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)
E_ILUSE	不正使用(多重ロック、上限優先度違反)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

mtxid のミューテックスをロックする。ミューテックスがロックできれば、本システムコールの発行タスクは待ち状態に入らず、実行を継続する。この場合、そのミューテックスはロック状態になる。ロックできなければ、本システムコールを発行したタスクは待ち状態に入る。すなわち、そのミューテックスに対する待ち行列につながれる。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。待ち解除の条件が満足されない(ロックが解除されない)まま tmout の時間が経過すると、タイムアウトエラー E_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1ミリ秒)と同じである。

tmout として TMO_POL＝0を指定した場合は、タイムアウト値として0を指定したことを示し、ロックできなくても待ちに入らず E_TMOUT を返す。また、tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずにロックできるまで待ち続ける。

自タスクがすでに対象ミューテックスをロックしていた場合には、E_ILUSE(多重ロック)を返す。

対象ミューテックスが TA_CEILING 属性の場合、自タスクのベース優先度[1]が対象ミューテックスの上限優先度より高い場合には E_ILUSE(上限優先度違反)を返す。

補足事項

- TA_INHERIT 属性のミューテックスの場合

  自タスクがロック待ち状態になる場合、そのミューテックスをロックしているタスクの現在優先度が自タスクより低ければ、ロックしているタスクの優先度を自タスクと同じ優先度まで引き上げる。ロックを待っているタスクがロックを獲得せずに待ちを終了した場合(タイムアウトなど)、そのミューテックスをロック中のタスクの優先度を、次の内の最も高い優先度まで引き下げる。ただし、この優先度の引き下げを行うか否かは実装依存である。

  a. そのミューテックスでロック待ちしているタスクの現在優先度の内の最も高い優先度。
  b. そのミューテックスをロック中のタスクがロックしている他のすべてのミューテックスの内の最も高い優先度。
  c. ロック中のタスクのベース優先度。

- TA_CEILING 属性のミューテックスの場合

  自タスクがロックを獲得した場合、自タスクの現在優先度がミューテックスの上限優先度より低ければ、自タスクの優先度をミューテックスの上限優先度まで引き上げる。

[1] ベース優先度:ミューテックスによって自動的に引き上げられる前のタスクの優先度を示す。最後(ミューテックスのロック中も含む)に tk_chg_pri によって設定された優先度、または tk_chg_pri を一度も発行していない場合はタスク生成時に指定したタスク優先度が、ベース優先度である。

4.5.1.4 tk_loc_mtx_u - ミューテックスのロック(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_loc_mtx_u (ID mtxid , TMO_U tmout_u);

パラメータ

ID	mtxid	Mutex ID	ミューテックスID
TMO_U	tmout_u	Timeout	タイムアウト指定(マイクロ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mtxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mtxid のミューテックスが存在しない)
E_PAR	パラメータエラー(tmout_u≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象ミューテックスが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)
E_ILUSE	不正使用(多重ロック、上限優先度違反)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_loc_mtx のパラメータである tmout を64ビットマイクロ秒単位の tmout_u としたシステムコールである。

パラメータが tmout_u となった点を除き、本システムコールの仕様は tk_loc_mtx と同じである。詳細は tk_loc_mtx の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.5.1.5 tk_unl_mtx - ミューテックスのアンロック

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_unl_mtx (ID mtxid);

パラメータ

ID	mtxid	Mutex ID	ミューテックスID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mtxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mtxid のミューテックスが存在しない)
E_ILUSE	不正使用(自タスクがロックしたミューテックスではない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

mtxid のミューテックスのロックを解除する。

ロック待ちしているタスクがあれば、待ち行列の先頭のタスクの待ちを解除し、そのタスクをロック獲得状態にする。

自タスクがロックしていないミューテックスを指定した場合、E_ILUSE を返す。

補足事項

ロック解除したミューテックスが TA_INHERIT または TA_CEILING 属性の場合、次のようにタスク優先度を引き下げる必要がある。

ロックを解除することにより、自タスクがロックしているミューテックスがすべてなくなった場合は、自タスクの優先度をベース優先度まで引き下げる。

自タスクがロック中のミューテックスが残っている場合、自タスクの優先度を次の内の最も高い優先度まで引き下げる。

a. 自タスクがロックしている TA_INHERIT 属性を持つミューテックスの待ち行列につながれているタスクの現在優先度の中で最も高い優先度
b. 自タスクがロックしている TA_CEILING 属性を持つミューテックスに設定されている上限優先度の中で最も高い優先度
c. 自タスクのベース優先度

ただし、ロック中のミューテックスが残っている場合の優先度の引き下げを行うか否かは実装依存である。

ミューテックスをロックした状態でタスクを終了した(休止状態(DORMANT)または未登録状態(NON-EXISTENT)になった)場合、当該タスクがロックしているすべてのミューテックスはT-Kernelによって自動的にロック解除される。

4.5.1.6 tk_ref_mtx - ミューテックス状態参照

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_mtx (ID mtxid , T_RMTX *pk_rmtx);

パラメータ

ID	mtxid	Mutex ID	ミューテックスID
T_RMTX*	pk_rmtx	Packet to Refer Mutex Status	ミューテックス状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rmtx の内容

void*	exinf	Extended Information	拡張情報
ID	htsk	Locking Task ID	ロックしているタスクのID
ID	wtsk	Lock Waiting Task ID	ロック待ちタスクのID

――(以下に実装独自に他の情報を追加してもよい)――

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mtxid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mtxid のミューテックスが存在しない)
E_PAR	パラメータエラー(pk_rmtx が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

mtxid で示された対象ミューテックスの各種の状態を参照し、リターンパラメータとしてロック中のタスク(htsk)、ロック待ちタスク(wtsk)、拡張情報(exinf) を返す。

htsk は、このミューテックスをロックしているタスクのIDを示す。ロックしているタスクがない場合は htsk＝0となる。

wtsk は、このミューテックスで待っているタスクのIDを示す。複数のタスクが待っている場合には、待ち行列の先頭のタスクのIDを返す。待ちタスクが無い場合は wtsk＝0となる。

対象ミューテックスが存在しない場合には、エラー E_NOEXS となる。

4.5.2 メッセージバッファ

メッセージバッファは、可変長のメッセージを受渡しすることにより、同期と通信を行うためのオブジェクトである。メッセージバッファ機能には、メッセージバッファを生成／削除する機能、メッセージバッファに対してメッセージを送信／受信する機能、メッセージバッファの状態を参照する機能が含まれる。メッセージバッファはID番号で識別されるオブジェクトである。メッセージバッファのID番号をメッセージバッファIDと呼ぶ。

メッセージバッファは、メッセージの送信を待つタスクの待ち行列(送信待ち行列)とメッセージの受信を待つタスクの待ち行列(受信待ち行列)を持つ。また、送信されたメッセージを格納するためのメッセージバッファ領域を持つ。メッセージを送信する側(イベントを知らせる側)では、送信したいメッセージをメッセージバッファにコピーする。メッセージバッファ領域の空き領域が足りなくなった場合、メッセージバッファ領域に十分な空きができるまでメッセージバッファへの送信待ち状態になる。

メッセージバッファへの送信待ち状態になったタスクは、そのメッセージバッファの送信待ち行列につながれる。一方、メッセージを受信する側(イベントを待つ側)では、メッセージバッファに入っているメッセージを一つ取り出す。メッセージバッファにメッセージが入っていない場合は、次にメッセージが送られてくるまでメッセージバッファからの受信待ち状態になる。メッセージバッファからの受信待ち状態になったタスクは、そのメッセージバッファの受信待ち行列につながれる。

メッセージバッファ領域のサイズを0にすることで、同期メッセージ機能を実現することができる。すなわち、送信側のタスクと受信側のタスクが、それぞれ相手のタスクがシステムコールを呼び出すのを待ち合わせ、両者がシステムコールを呼び出した時点で、メッセージの受渡しが行われる。

補足事項
メッセージバッファ領域のサイズを0にした場合の、メッセージバッファの動作を[図 4.3.「メッセージバッファによる同期通信」]の例を用いて説明する。この図で、タスクAとタスクBは非同期に実行しているものとする。

- もしタスクAが先に tk_snd_mbf を呼び出した場合には、タスクBが tk_rcv_mbf を呼び出すまでタスクAは待ち状態となる。この時タスクAは、メッセージバッファへの送信待ち状態になっている[図 4.3.「メッセージバッファによる同期通信」(a)]。
- 逆にタスクBが先に tk_rcv_mbf を呼び出した場合には、タスクAが tk_snd_mbf を呼び出すまでタスクBは待ち状態となる。この時タスクBは、メッセージバッファからの受信待ち状態になっている[図 4.3.「メッセージバッファによる同期通信」(b)]。
- タスクAが tk_snd_mbf を呼び出し、タスクBが tk_rcv_mbf を呼び出した時点で、タスクAからタスクBへメッセージの受渡しが行われる。その後は、両タスクとも実行できる状態となる。

メッセージバッファへの送信を待っているタスクは、待ち行列につながれている順序でメッセージを送信する。例えば、あるメッセージバッファに対して40バイトのメッセージを送信しようとしているタスクAと、10バイトのメッセージを送信しようとしているタスクBが、この順で待ち行列につながれている時に、別のタスクによるメッセージの受信により20バイトの空き領域ができたとする。このような場合でも、タスクAがメッセージを送信するまで、タスクBはメッセージを送信できない。
メッセージバッファは、可変長のメッセージをコピーして受渡しする。メールボックスとの違いは、メッセージをコピーすることである。
メッセージバッファは、リングバッファで実装することを想定している。

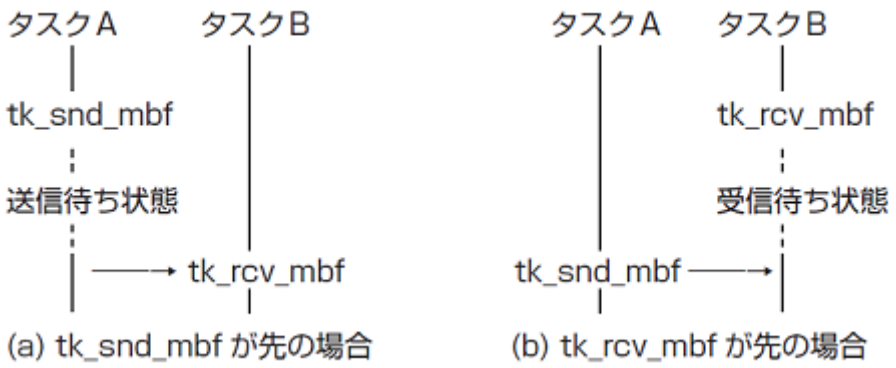

図 4.3: メッセージバッファによる同期通信

4.5.2.1 tk_cre_mbf - メッセージバッファ生成

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID mbfid = tk_cre_mbf (CONST T_CMBF *pk_cmbf);

パラメータ

CONST T_CMBF*	pk_cmbf	Packet to Create Message Buffer	メッセージバッファ生成情報

pk_cmbf の内容

void*	exinf	Extended Information	拡張情報
ATR	mbfatr	Message Buffer Attribute	メッセージバッファ属性
INT	bufsz	Buffer Size	メッセージバッファのサイズ(バイト数)
INT	maxmsz	Max Message Size	メッセージの最大長(バイト数)
UB	dsname[8]	DS Object name	DSオブジェクト名称

――(以下に実装独自に他の情報を追加してもよい)――

リターンパラメータ

ID	mbfid	Message Buffer ID	メッセージバッファID
	または	Error Code	エラーコード

エラーコード

E_NOMEM	メモリ不足(管理ブロックやリングバッファ用の領域が確保できない)
E_LIMIT	メッセージバッファの数がシステムの上限を超えた
E_RSATR	予約属性(mbfatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_cmbf が不正, bufsz, maxmsz が負または不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

メッセージバッファを生成しメッセージバッファID番号を割り当てる。具体的には、生成するメッセージバッファに対して管理ブロックを割り付ける。また、bufsz の情報を元に、メッセージキュー(受信されるのを待つメッセージの待ち行列) として利用するためのリングバッファの領域を確保する。

メッセージバッファは、可変長メッセージの送受信の管理を行うオブジェクトである。メールボックス(mbx)との違いは、送信時と受信時に可変長のメッセージ内容がコピーされるということである。また、バッファが一杯の場合に、メッセージ送信側も待ち状態に入る機能がある。

exinf は、対象メッセージバッファに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、tk_ref_mbf で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を

変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。カーネルでは exinf の内容について関知しない。

mbfatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。mbfatr のシステム属性の部分では、次のような指定を行う。

```
mbfatr:= (TA_TFIFO || TA_TPRI) | [TA_DSNAME] | [TA_NODISWAI]
```

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
TA_DSNAME	DSオブジェクト名称を指定する
TA_NODISWAI	tk_dis_wai による待ち禁止を拒否する

TA_TFIFO, TA_TPRI では、バッファが一杯の場合にメッセージを送信するタスクがメッセージバッファの待ち行列に並ぶ際の並び方を指定することができる。属性が TA_TFIFO であればタスクの待ち行列はFIFOとなり、属性が TA_TPRI であればタスクの待ち行列はタスクの優先度順となる。なお、メッセージキューの順序はFIFOのみである。

メッセージ受信待ちのタスクの待ち行列の順序はFIFOのみである。

TA_DSNAME を指定した場合に dsname が有効となり、DSオブジェクト名称として設定される。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DSのシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname 、td_set_dsname を参照のこと。TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

```
#define TA_TFIFO        0x00000000      /* 送信待ちタスクをFIFOで管理 */
#define TA_TPRI         0x00000001      /* 送信待ちタスクを優先度順で管理 */
#define TA_DSNAME       0x00000040      /* DSオブジェクト名称を指定 */
#define TA_NODISWAI     0x00000080      /* 待ち禁止拒否 */
```

補足事項

送信待ちのタスクが複数あった場合、バッファの空きができて送信待ちが解除されるのは常に待ち行列の順となる。

例えば、30バイトのメッセージを送信しようとしているタスクAと、10バイトのメッセージを送信しようとしているタスクBがA-Bの順で待っていた場合、メッセージバッファに20バイトの空きができてもAのタスクを追い越してBのタスクが先に送信することはない。

メッセージキューを入れるリングバッファの中には、一つ一つのメッセージを管理する情報も入るため、bufsz で指定されたリングバッファのサイズとキューに入るメッセージのサイズの合計とは、一般には一致しない。後者の方が小さい値をとるのが普通である。その意味で、bufsz の情報は厳密な意味をもつものではない。

bufsz＝0のメッセージバッファを生成することは可能である。この場合、このメッセージバッファでは送受信側が完全に同期した通信を行うことになる。すなわち、tk_snd_mbf と tk_rcv_mbf の一方のシステムコールが先に実行されると、それを実行したタスクは待ち状態となる。もう一方のシステムコールが実行された段階で、メッセージの受け渡し(コピー)が行われ、その後双方のタスクが実行を再開する。

bufsz＝0のメッセージバッファの場合、具体的な動作は次のようになる。

1. [図 4.4.「bufsz＝0のメッセージバッファを使った同期式通信」]で、タスクAとタスクBは非同期に動いている。もし、タスクAが先に(1)に到達し、tk_snd_mbf(mbfid) を実行した場合には、タスクBが(2)に到達するまでタスクAはメッセージ送信待ち状態になる。この状態のタスクAを対象として tk_ref_tsk を発行すると、tskwait＝TTW_SMBF となる。逆に、タスクBが先に(2)に到達し、tk_rcv_mbf(mbfid) を実行した場合には、タスクAが(1)に到達するまでタスクBはメッセージ受信待ち状態になる。この状態のタスクBを対象として tk_ref_tsk を発行すると、tskwait＝TTW_RMBF となる。
2. タスクAが tk_snd_mbf(mbfid) を実行し、かつタスクBが tk_rcv_mbf(mbfid) を実行した時点でタスクAからタスクBにメッセージが送信され、どちらのタスクも待ち解除となって実行を再開する。

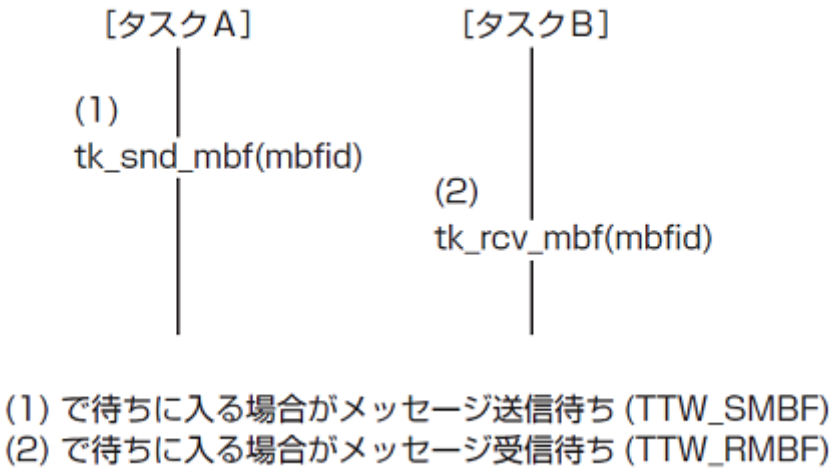

図 4.4: bufsz＝0のメッセージバッファを使った同期式通信

4.5.2.2 tk_del_mbf - メッセージバッファ削除

C言語インタフェース

```
#include <tk/tkernel.h>

ER ercd = tk_del_mbf (ID mbfid );
```

パラメータ

ID	mbfid	Message Buffer ID	メッセージバッファID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mbfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbfid のメッセージバッファが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

mbfid で示されたメッセージバッファを削除する。

本システムコールの発行により、対象メッセージバッファのID番号および管理ブロック用の領域およびメッセージを入れるバッファ領域は解放される。

対象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にも、本システムコールは正常終了するが、待ち状態にあったタスクにはエラー E_DLT が返される。また、対象メッセージバッファの中にメッセージが残っている場合でも、エラーとはならず、メッセージバッファの削除が行われ、中にあったメッセージは消滅する。

4.5.2.3 tk_snd_mbf - メッセージバッファへ送信

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_snd_mbf (ID mbfid , CONST void *msg , INT msgsz , TMO tmout);

パラメータ

ID	mbfid	Message Buffer ID	メッセージバッファID
CONST void*	msg	Send Message	送信メッセージの先頭アドレス
INT	msgsz	Send Message Size	送信メッセージのサイズ(バイト数)
TMO	tmout	Timeout	タイムアウト指定(ミリ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mbfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbfid のメッセージバッファが存在しない)
E_PAR	パラメータエラー(msgsz≦0, msgsz>maxmsz, msg が不正, tmout≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象メッセージバッファが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×(※条件により可能)

解説

tk_snd_mbf では、mbfid で示されたメッセージバッファに対して、msg のアドレスに入っているメッセージを送信する。メッセージのサイズは msgsz で指定される。すなわち、msg 以下の msgsz バイトが、mbfid で指定されたメッセージバッファのメッセージキューにコピーされる。メッセージキューは、リングバッファによって実現されることを想定している。

msgsz が、tk_cre_mbf で指定した maxmsz よりも大きい場合は、エラー E_PAR となる。

バッファの空き領域が少なく、msg のメッセージをメッセージキューに入れられない場合、本システムコールを発行したタスクはメッセージ送信待ち状態となり、バッファの空きを待つための待ち行列(送信待ち行列)につながれる。待ち行列の順序は tk_cre_mbf 時の指定によりFIFOまたはタスク優先度順となる。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。タイムアウト指定が行われた場合、待ち解除の条件が満足されない(バッファに十分な空き領域ができない)まま tmout の時間が経過すると、タイムアウトエラー E_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1ミリ秒)と同じである。

tmout として TMO_POL＝0を指定した場合は、タイムアウト値として0を指定したことを示し、バッファに十分な空きがない場合は待ちに入らず E_TMOUT を返す。また、tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずにバッファに空きができるまで待ち続ける。

長さが0のメッセージは送信することができない。msgsz≦0の場合には、エラー E_PAR となる。

タスク独立部やディスパッチ禁止状態から実行した場合はエラー E_CTX となるが、tmout＝TMO_POL の場合は、実装によってはタスク独立部やディスパッチ禁止状態から実行することができる場合がある。

4.5.2.4 tk_snd_mbf_u - メッセージバッファへ送信(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_snd_mbf_u (ID mbfid , CONST void *msg , INT msgsz , TMO_U tmout_u);

パラメータ

ID	mbfid	Message Buffer ID	メッセージバッファID
CONST void*	msg	Send Message	送信メッセージの先頭アドレス
INT	msgsz	Send Message Size	送信メッセージのサイズ(バイト数)
TMO_U	tmout_u	Timeout	タイムアウト指定(マイクロ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mbfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbfid のメッセージバッファが存在しない)
E_PAR	パラメータエラー(msgsz≦0, msgsz>maxmsz, msg が不正, tmout_u≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象メッセージバッファが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×(※条件により可能)

解説

tk_snd_mbf のパラメータである tmout を64ビットマイクロ秒単位の tmout_u としたシステムコールである。

パラメータが tmout_u となった点を除き、本システムコールの仕様は tk_snd_mbf と同じである。詳細は tk_snd_mbf の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.5.2.5 tk_rcv_mbf - メッセージバッファから受信

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT msgsz = tk_rcv_mbf (ID mbfid , void *msg , TMO tmout);

パラメータ

ID	mbfid	Message Buffer ID	メッセージバッファID
void*	msg	Receive Message	受信メッセージを入れるアドレス
TMO	tmout	Timeout	タイムアウト指定(ミリ秒)

リターンパラメータ

INT	msgsz	Receive Message Size	受信したメッセージのサイズ(バイト数)
	または	Error Code	エラーコード

エラーコード

E_ID	不正ID番号(mbfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbfid のメッセージバッファが存在しない)
E_PAR	パラメータエラー(msg が不正, tmout≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象メッセージバッファが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_rcv_mbf では、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、msg で指定した領域に入れる。すなわち、mbfid で指定されたメッセージバッファのメッセージキューの先頭のメッセージの内容を、msg 以下の msgsz バイトにコピーする。

mbfid で示されたメッセージバッファにまだメッセージが送信されていない場合(メッセージキューが空の場合) には、本システムコールを発行したタスクは待ち状態となり、メッセージの到着を待つ待ち行列(受信待ち行列)につながれる。受信待ちタスクの待ち行列はFIFOのみである。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。タイムアウト指定が行われた場合、待ち解除の条件が満足されない(メッセージが到着しない)まま tmout の時間が経過すると、タイムアウトエラー E_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1ミリ秒)と同じである。

tmout として TMO_POL＝0を指定した場合は、タイムアウト値として0を指定したことを示し、メッセージがない場合にも待ちに入らず E_TMOUT を返す。また、tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずにメッセージが到着するまで待ち続ける。

4.5.2.6 tk_rcv_mbf_u - メッセージバッファから受信(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT msgsz = tk_rcv_mbf_u (ID mbfid , void *msg , TMO_U tmout_u);

パラメータ

ID	mbfid	Message Buffer ID	メッセージバッファID
void*	msg	Receive Message	受信メッセージを入れるアドレス
TMO_U	tmout_u	Timeout	タイムアウト指定(マイクロ秒)

リターンパラメータ

INT	msgsz	Receive Message Size	受信したメッセージのサイズ(バイト数)
	または	Error Code	エラーコード

エラーコード

E_ID	不正ID番号(mbfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbfid のメッセージバッファが存在しない)
E_PAR	パラメータエラー(msg が不正, tmout_u≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象メッセージバッファが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_rcv_mbf のパラメータである tmout を64ビットマイクロ秒単位の tmout_u としたシステムコールである。

パラメータが tmout_u となった点を除き、本システムコールの仕様は tk_rcv_mbf と同じである。詳細は tk_rcv_mbf の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.5.2.7 tk_ref_mbf - メッセージバッファ状態参照

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_mbf (ID mbfid , T_RMBF *pk_rmbf);

パラメータ

ID	mbfid	Message Buffer ID	メッセージバッファID
T_RMBF*	pk_rmbf	Packet to Refer Message Buffer Status	メッセージバッファ状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rmbf の内容

void*	exinf	Extended Information	拡張情報
ID	wtsk	Waiting Task ID	受信待ちタスクのID
ID	stsk	Send Waiting Task ID	送信待ちタスクのID
INT	msgsz	Message Size	次に受信されるメッセージのサイズ(バイト数)
INT	frbufsz	Free Buffer Size	空きバッファのサイズ(バイト数)
INT	maxmsz	Maximum Message Size	メッセージの最大長(バイト数)

――(以下に実装独自に他の情報を追加してもよい)――

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mbfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mbfid のメッセージバッファが存在しない)
E_PAR	パラメータエラー(pk_rmbf が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

mbfid で示された対象メッセージバッファの各種の状態を参照し、リターンパラメータとして送信待ちタスクのID(stsk)、次に受信されるメッセージのサイズ(msgsz)、空きバッファのサイズ(frbufsz)、メッセージの最大長(maxmsz)、受信待ちタスクのID(wtsk)、拡張情報(exinf) を返す。

wtsk は、このメッセージバッファで受信待ちしているタスクのIDを示す。また、stsk は送信待ちしているタスクのIDを示す。このメッセージバッファで複数のタスクが待っている場合には、待ち行列の先頭のタスクのIDを返す。待ちタスクが無い場合は0となる。

対象メッセージバッファが存在しない場合には、エラー E_NOEXS となる。

msgsz には、メッセージキューの先頭のメッセージ(次に受信されるメッセージ)のサイズが返る。メッセージキューにメッセージが無い場合には、msgsz＝0となる。なお、サイズが0のメッセージを送ることはできない。

どんな場合でも、msgsz＝0とwtsk＝0の少なくとも一方は成り立つ。

frbufsz は、メッセージキューを構成するリングバッファの空き領域のサイズを示すものである。この値は、あとどの程度の量のメッセージを送信できるかを知る手掛かりになる。

maxmsz には、tk_cre_mbf で指定したメッセージの最大長が返される。

4.5.3 ランデブ

ランデブ機能は、複数のタスクがサーバとクライアントの関係にある場合に、それらのタスク間での同期通信を行う機能である。具体的には、クライアント側タスクとサーバ側タスクの双方が処理の受付を待ち合わせる機能、クライアント側タスクからサーバ側タスクに処理を依頼するメッセージ(呼出メッセージ)を送る機能、サーバ側タスクの処理の完了をクライアント側タスクが待つ機能、サーバ側タスクからクライアント側タスクに処理結果のメッセージ(返答メッセージ)を返す機能が含まれる。ランデブのシステムコールを使うことにより、上記のような複数の一連の処理を簡単な手順で実現できる。ランデブ機能は、ランデブポートと呼ばれるオブジェクトの上で動作する。

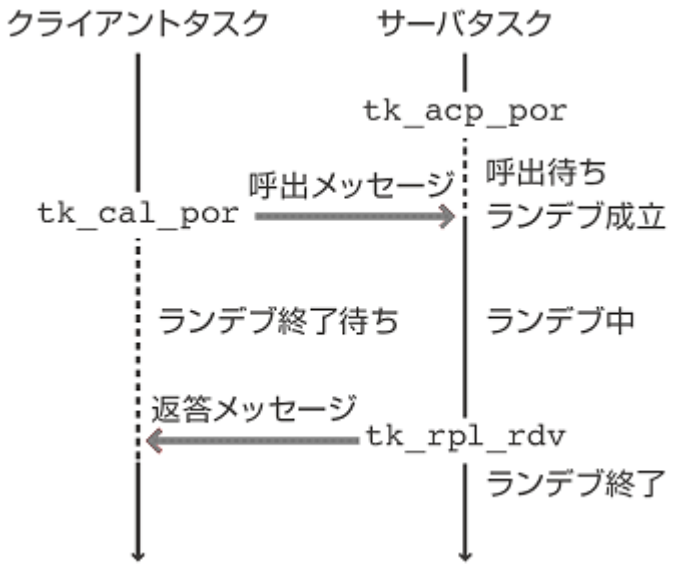

図 4.5: クライアントタスクとサーバタスクの間のランデブの動作

ランデブ機能には、ランデブポートを生成／削除する機能、ランデブポートに対して処理の依頼を行う機能(ランデブの呼出)、ランデブポートで処理依頼を受け付ける機能(ランデブの受付)、処理結果を返す機能(ランデブの終了)、受け付けた処理依頼を他のランデブポートに回送する機能(ランデブの回送)ランデブポートおよびランデブの状態を参照する機能が含まれる。ランデブポートはID番号で識別されるオブジェクトである。ランデブポートのID番号をランデブポートIDと呼ぶ。

ランデブポートに対して処理依頼を行う側のタスク(クライアント側のタスク)は、ランデブポートとランデブ条件、依頼する処理に関する情報を入れたメッセージ(これを呼出メッセージと呼ぶ)を指定して、ランデブの呼出を行う。一方、ランデブポートで処理依頼を受け付ける側のタスク(サーバ側のタスク)は、ランデブポートとランデブ条件を指定して、ランデブの受付を行う。

ランデブ条件は、ビットパターンで指定する。あるランデブポートに対して、呼び出したタスクのランデブ条件のビットパターンと、受け付けたタスクのランデブ条件のビットパターンをビット毎に論理積をとり、結果が0以外の場合にランデブが成立する。ランデブを呼び出したタスクは、ランデブが成立するまでランデブ呼出待ち状態となる。逆に、ランデブを受け付けるタスクは、ランデブが成立するまでランデブ受付待ち状態となる。

ランデブが成立すると、ランデブを呼び出したタスクから受け付けたタスクへ、呼出メッセージが渡される。ランデブを呼び出したタスクはランデブ終了待ち状態へ移行し、依頼した処理が完了するのを待つ。一方、ランデブを受け付けたタスクは待ち解除され、依頼された処理を行う。ランデブを受け付けたタスクが依頼された処理を完了すると、処理結果を返答メッセージの形で呼び出したタスクに渡し、ランデブを終了する。この時点で、ランデブを呼び出したタスクが、ランデブ終了待ち状態から待ち解除される。

上記の動作を[図 4.6.「ランデブの動作」]の例を用いて説明する。この図で、タスクAとタスクBは非同期に実行しているものとする。

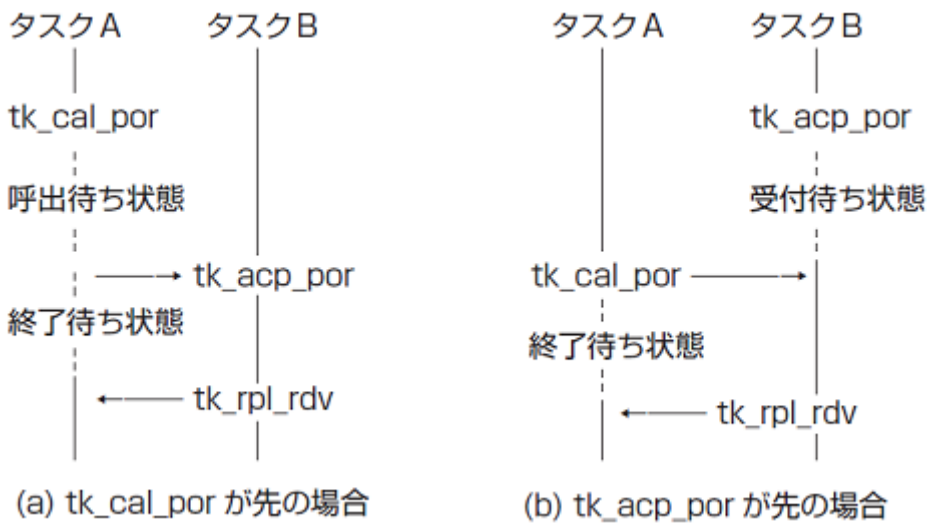

図 4.6: ランデブの動作

- もしタスクAが先に tk_cal_por を呼び出した場合には、タスクBが tk_acp_por を呼び出すまでタスクAは待ち状態となる。この時タスクAは、ランデブの呼出待ち状態になっている[図 4.6.「ランデブの動作」(a)]。
- 逆にタスクBが先に tk_acp_por を呼び出した場合には、タスクAが tk_cal_por を呼び出すまでタスクBは待ち状態となる。この時タスクBは、ランデブの受付待ち状態になっている[図 4.6.「ランデブの動作」(b)]。
- タスクAが tk_cal_por を呼び出し、タスクBが tk_acp_por を呼び出した時点でランデブが成立し、タスクAを待ち状態としたまま、タスクBが待ち解除される。この時タスクAは、ランデブの終了待ち状態になっている。
- タスクBが tk_rpl_rdv を呼び出した時点で、タスクAは待ち解除される。その後は、両タスクとも実行できる状態となる。

ランデブポートは、ランデブ呼出待ち状態のタスクをつなぐための呼出待ち行列と、ランデブ受付待ち状態のタスクをつなぐための受付待ち行列を持つ。それに対して、ランデブが成立した後は、ランデブした双方のタスクはランデブポートから切り離される。すなわち、ランデブポートは、ランデブ終了待ち状態のタスクをつなぐための待ち行列は持たない。また、ランデブを受け付け、依頼された処理を実行しているタスクに関する情報も持たない。

カーネルは、同時に成立しているランデブを識別するために、ランデブ番号と呼ばれるユニークな番号を付与する。ランデブ番号の付与方法は実装依存であるが、少なくとも、ランデブを呼び出したタスクを指定するための情報を含んでいなければならない。また、同じタスクが呼び出したランデブであっても、1回目のランデブと2回目のランデブで異なるランデブ番号が付与される。

補足事項
ランデブ番号の付与方法の一例として、ランデブを呼び出したタスクのID番号をランデブ番号の下位ビットとし、その上位ビットにシーケンシャルな番号を付加する方法がある。

仕様決定の理由
本機能の名称である「ランデブ(rendezvous)」は、クライアント側タスクとサーバ側タスクの両者が「待ち合わせ」を行うことに由来している。また、本機能の導入に際しては、Ada言語のランデブや、その元になったCSP(Communicating Sequential Processes)の影響を受けている。ただし、T-Kernelが提供するランデブ機能とAda言語のランデブの機能が同一というわけではない。
ランデブ機能は、他の同期・通信機能を組み合わせて実現することも可能であるが、返答を伴う通信を行う場合には、それ専用の機能を用意した方が、アプリケーションプログラムが書きやすくなる。
また、他の同期・通信機能を組み合わせるよりも効率を上げることができると考えられる。一例として、ランデブ機能は、メッセージの受渡しが終わるまで双方のタスクを待たせておくために、メッセージを格納するための領域が必要ないという利点がある。
同じタスクが呼び出したランデブであっても、ランデブ番号をできる限りユニークにしなければならないのは、次の理由による。ランデブが成立してランデブ終了待ち状態となっているタスクが、タイムアウトや待ち状態の強制解除などにより待ち解除された後、再度ランデブを呼び出してランデブが成立した場合を考える。この時、最初のランデブのランデブ番号と、後のランデブのランデブ番号が同一の値であると、最初のランデブを終了させようとした時に、ランデブ番号が同一であるために後のランデブが終了してしまう。2つのランデブに異なるランデブ番号を付与し、ランデブ終了待ち状態のタスクに待ち対象のランデブ番号を記憶しておけば、最初のランデブを終了させようとした時にエラーとすることができる。

4.5.3.1 tk_cre_por - ランデブポート生成

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID porid = tk_cre_por (CONST T_CPOR *pk_cpor);

パラメータ

CONST T_CPOR*	pk_cpor	Packet to Create Port	ランデブポートの生成情報

pk_cpor の内容

void*	exinf	Extended Information	拡張情報
ATR	poratr	Port Attribute	ランデブポート属性
INT	maxcmsz	Max Call Message Size	呼出時のメッセージの最大長(バイト数)
INT	maxrmsz	Max Reply Message Size	返答時のメッセージの最大長(バイト数)
UB	dsname[8]	DS Object name	DSオブジェクト名称

――(以下に実装独自に他の情報を追加してもよい)――

リターンパラメータ

ID	porid	Port ID	ランデブポートID
	または	Error Code	エラーコード

エラーコード

E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_LIMIT	ランデブポートの数がシステムの上限を超えた
E_RSATR	予約属性(poratr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_cpor が不正, maxcmsz, maxrmsz が負または不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ランデブポートを生成しランデブポートID番号を割り当てる。具体的には、生成されたランデブポートに対して管理ブロックを割り付ける。ランデブポートは、ランデブを実現するためのプリミティブとなるオブジェクトである。

exinf は、対象ランデブポートに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、tk_ref_por で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。カーネルでは exinf の内容について関知しない。

poratr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。poratr のシステム属性の部分では、次のような指定を行う。

```
poratr:= (TA_TFIFO || TA_TPRI) | [TA_DSNAME] | [TA_NODISWAI]
```

TA_TFIFO	呼出待ちタスクのキューイングはFIFO
TA_TPRI	呼出待ちタスクのキューイングは優先度順
TA_DSNAME	DSオブジェクト名称を指定する
TA_NODISWAI	tk_dis_wai による待ち禁止を拒否する

TA_TFIFO, TA_TPRI では、ランデブ呼出待ちのタスクの待ち行列の並び順を指定する。ランデブ受付待ちのタスクの待ち行列はFIFOのみである。

TA_DSNAME を指定した場合に dsname が有効となり、DSオブジェクト名称として設定される。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DSのシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname 、td_set_dsname を参照のこと。TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

```
#define TA_TFIFO        0x00000000      /* 待ちタスクをFIFOで管理 */
#define TA_TPRI         0x00000001      /* 待ちタスクを優先度順で管理 */
#define TA_DSNAME       0x00000040      /* DSオブジェクト名称を指定 */
#define TA_NODISWAI     0x00000080      /* 待ち禁止拒否 */
```

maxcmsz にはランデブの呼出時に渡すメッセージの最大サイズ(バイト数)を指定する。maxcmsz に0を指定することも可能である。ただし、maxcmsz に0を指定した場合は、ランデブの呼出時に渡すメッセージのサイズは0に限定され、メッセージ無しでの同期にのみ使用されることになる。

maxrmsz にはランデブの返答時に渡すメッセージの最大サイズ(バイト数)を指定する。maxrmsz に0を指定することも可能である。ただし、maxrmsz に0を指定した場合はランデブの返答時に渡すメッセージのサイズは0に限定されることになる。

4.5.3.2 tk_del_por - ランデブポート削除

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_del_por (ID porid);

パラメータ

ID	porid	Port ID	ランデブポートID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(porid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(porid のランデブポートが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

porid で示されたランデブポートを削除する。

本システムコールの発行により、対象ランデブポートのID番号および管理ブロック用の領域は解放される。

対象ランデブポートにおいてランデブ受付(tk_acp_por)や呼出(tk_cal_por)を待っているタスクがあった場合にも、本システムコールは正常終了するが、待ち状態にあったタスクにはエラー E_DLT が返される。

tk_del_por によりランデブポートが削除されても、既にランデブ成立済のタスクに対しては、影響を与えない。ランデブ受付側タスク(待ち状態ではない) には何も通知されないし、ランデブ呼出側タスク(ランデブ終了待ち状態) の状態もそのままである。ランデブ受付側タスクが tk_rpl_rdv を行う時に、ランデブ成立に使ったランデブポートが既に削除されていても、tk_rpl_rdv は正常に実行される。

4.5.3.3 tk_cal_por - ランデブポートに対するランデブの呼出

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT rmsgsz = tk_cal_por (ID porid , UINT calptn , void *msg , INT cmsgsz , TMO tmout);

パラメータ

ID	porid	Port ID	ランデブポートID
UINT	calptn	Call Bit Pattern	呼出側選択条件を表すビットパターン
void*	msg	Message	メッセージを入れるアドレス
INT	cmsgsz	Call Message Size	呼出メッセージのサイズ(バイト数)
TMO	tmout	Timeout	タイムアウト指定(ミリ秒)

リターンパラメータ

INT	rmsgsz	Reply Message Size	返答メッセージのサイズ(バイト数)
	または	Error Code	エラーコード

エラーコード

E_ID	不正ID番号(porid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(porid のランデブポートが存在しない)
E_PAR	パラメータエラー(cmsgsz<0, cmsgsz>maxcmsz, calptn=0, msg が不正, tmout≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象ランデブポートが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ランデブポートに対するランデブ呼出を行う。

tk_cal_por の具体的な動作は次のようになる。porid で指定したランデブポートにおいてランデブ受付待ち状態のタスクがあり、そのタスクと tk_cal_por 発行タスクとの間でランデブ成立条件が満たされた場合は、ランデブ成立となる。この場合、ランデブ受付待ちだったタスクは実行可能状態(READY)となり、tk_cal_por 発行タスクはランデブ終了待ちの状態になる。ランデブ終了待ちの状態になったタスクは、ランデブの相手のタスク(ランデブ受付タスク)が tk_rpl_rdv を実行することにより待ち状態が解除される。この時点で tk_cal_por のシステムコールが終了する。

porid で指定したランデブポートに受付待ちタスクが無かった場合や、受付待ちタスクがあってもランデブ成立条件が満たされなかった場合には、tk_cal_por 発行タスクはこのランデブポートの呼出側待ち行列に並びランデブ呼出待ちの状態となる。ランデブ呼出待ち行列の順序は、tk_cre_por の指定によりFIFOまたはタスク優先度順となる。

ランデブ成立条件は、受付側タスクの acpptn と呼出側タスクの calptn との論理積が0かどうかによって判定される。論理積が0でない場合にランデブ成立となる。calptn が0の場合は、決してランデブが成立しなくなるので、パラメータエラー E_PAR とする。

ランデブ成立時には、呼出側タスクから受付側タスクに対してメッセージ(呼出メッセージ)を送ることができる。呼出メッセージのサイズは cmsgsz で指定される。具体的には、呼出側タスクが tk_cal_por で指定した msg 以下の領域の cmsgsz バイトが、受付側タスクが tk_acp_por で指定した msg 以下の領域にコピーされる。

逆に、ランデブ終了時には、受付側タスクから呼出側タスクに対してメッセージ(返答メッセージ)を送ることができる。具体的には、受付側タスクが tk_rpl_rdv で指定した返答メッセージの内容が、呼出側タスクが tk_cal_por で指定した msg 以下の領域にコピーされる。また、返答メッセージのサイズ rmsgsz は、tk_cal_por のリターンパラメータとなる。結局、tk_cal_por の msg パラメータで指定されたメッセージ領域は、tk_rpl_rdv 実行の際に送られてくるメッセージによって破壊されることになる。

なお、ランデブが回送された場合は、tk_cal_por で指定した msg のアドレスから最大で maxrmsz の領域をバッファとして使用するため、その内容を破壊する可能性がある。したがって、tk_cal_por で要求したランデブが回送される可能性がある場合は、期待する返答メッセージのサイズにかかわらず、msg 以下に少なくとも maxrmsz のサイズの領域を確保しておかなければならない。(詳細は tk_fwd_por の項を参照)

cmsgsz が、tk_cre_por で指定した maxcmsz よりも大きい場合は、エラー E_PAR となる。このエラーはランデブ呼出待ち状態に入る前にチェックされ、エラーの場合、tk_cal_por を実行したタスクはランデブ呼出待ち状態には入らない。

tmout によりランデブが成立するまでの待ち時間の最大値(タイムアウト値)を指定することができる。タイムアウト指定が行なわれた場合、待ち解除の条件が満足されない(ランデブが成立しない)まま tmout の時間が経過すると、タイムアウトエラー E_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1ミリ秒)と同じである。

tmout として TMO_POL＝0を指定した場合は、タイムアウト値として0を指定したことを示し、対象ランデブポートにランデブ受付待ちタスクが無いか、ランデブ成立条件が満たされない場合には、待ちに入らず E_TMOUT を返す。

tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトすることなくランデブが成立するまで待ち続ける。

いずれにしても、tmout の指定はランデブ成立までの時間に関するタイムアウトを意味するものであり、ランデブ成立からランデブ終了までの時間には関係しない。

4.5.3.4 tk_cal_por_u - ランデブポートに対するランデブの呼出(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT rmsgsz = tk_cal_por_u (ID porid , UINT calptn , void *msg , INT cmsgsz , TMO_U tmout_u);

パラメータ

ID	porid	Port ID	ランデブポートID
UINT	calptn	Call Bit Pattern	呼出側選択条件を表すビットパターン
void*	msg	Message	メッセージを入れるアドレス
INT	cmsgsz	Call Message Size	呼出メッセージのサイズ(バイト数)
TMO_U	tmout_u	Timeout	タイムアウト指定(マイクロ秒)

リターンパラメータ

INT	rmsgsz	Reply Message Size	返答メッセージのサイズ(バイト数)
	または	Error Code	エラーコード

エラーコード

E_ID	不正ID番号(porid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(porid のランデブポートが存在しない)
E_PAR	パラメータエラー(cmsgsz<0, cmsgsz>maxcmsz, calptn=0, msg が不正, tmout_u≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象ランデブポートが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_cal_por のパラメータである tmout を64ビットマイクロ秒単位の tmout_u としたシステムコールである。

パラメータが tmout_u となった点を除き、本システムコールの仕様は tk_cal_por と同じである。詳細は tk_cal_por の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.5.3.5 tk_acp_por - ランデブポートに対するランデブ受付

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT cmsgsz = tk_acp_por (ID porid , UINT acpptn , RNO *p_rdvno , void *msg , TMO tmout);

パラメータ

ID	porid	Port ID	ランデブポートID
UINT	acpptn	Accept Bit Pattern	受付側選択条件を表すビットパターン
RNO*	p_rdvno	Pointer to Rendezvous Number	リターンパラメータ rdvno を返す領域へのポインタ
void*	msg	Packet of Call Message	呼出メッセージを入れるアドレス
TMO	tmout	Timeout	タイムアウト指定(ミリ秒)

リターンパラメータ

RNO	rdvno	Rendezvous Number	ランデブ番号
INT	cmsgsz	Call Message Size	呼出メッセージのサイズ(バイト数)
	または	Error Code	エラーコード

エラーコード

E_ID	不正ID番号(porid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(porid のランデブポートが存在しない)
E_PAR	パラメータエラー(acpptn＝0, msg が不正, tmout≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象ランデブポートが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ランデブポートに対するランデブ受付を行う。

tk_acp_por の具体的な動作は次のようになる。porid で指定したランデブポートの呼出側待ち行列に入っているタスクと、このタスクとの間で、ランデブ成立条件が満たされた場合は、ランデブ成立となる。この場合、呼出側待ち行列にあったタスクは行列から外れ、ランデブ呼出待ち(ランデブ成立待ち)の状態からランデブ終了待ちの状態に変わる。tk_acp_por の発行タスクは、実行を継続する。

porid で指定したランデブポートの呼出側待ち行列にタスクが無かった場合や、タスクがあってもランデブ成立条件が満たされなかった場合、tk_acp_por の発行タスクはそのランデブポートに対するランデブ受付待ち状態になる。この時、既に別のタスクがランデブ受付待ち状態であったとしても、エラーとはならず、tk_acp_por を発行したタスクはランデブ受付待ち行列につながれる。また、一つのランデブポートを使って、複数のタスクが同時にランデブを行うことが可能である。そのため、porid で

指定したランデブポートで別のタスクがランデブを行っている間に(前に成立したランデブに関する tk_rpl_rdv が実行される前に) に次のランデブを行っても、エラーとはならない。

ランデブ成立条件は、受付側タスクの acpptn と呼出側タスクの calptn との論理積が0かどうかによって判定される。論理積が0でない場合にランデブ成立となる。先頭のタスクが条件を満たさなければ、待ち行列の次のタスクについて順にチェックを行う。calptn と acpptn に0以外の同じ値を指定すれば、条件が無い(無条件)のと同じになる。また、acpptn が0の場合は、決してランデブが成立しなくなるので、パラメータエラー E_PAR とする。ランデブ成立までの処理に関しては、ランデブ呼出側とランデブ受付側で完全に対称である。

ランデブ成立時には、呼出側タスクから受付側タスクに対して呼出メッセージを送ることができる。呼出側タスクが指定した呼出メッセージの内容は、受付側タスクが tk_acp_por で指定した msg 以下の領域にコピーされる。また、呼出メッセージのサイズ cmsgsz は、tk_acp_por のリターンパラメータとなる。

ランデブ受付側のタスクが、同時に複数のランデブを行うことも可能である。具体的には、tk_acp_por によりあるランデブを受け付けたタスクが、tk_rpl_rdv を実行する前に、もう一度 tk_acp_por を実行しても構わない。また、この時の tk_acp_por は、前と異なるランデブポートを対象としたものであっても、前と同じランデブポートを対象としたものであっても構わない。特殊な例であるが、ランデブ中のタスクが同じランデブポートに対してもう一度 tk_acp_por を実行してランデブが成立した場合、同一のタスクが同一のランデブポートに対して複数の(多重の)ランデブを行っているという状態になる。もちろん、この場合にランデブの相手(呼出側タスク)は異なっている。

tk_acp_por のリターンパラメータとして返される rdvno は、同時に成立している複数のランデブを区別するための情報であり、ランデブ終了時に tk_rpl_rdv のパラメータとして使用する。また、ランデブ回送時には tk_fwd_por のパラメータとして使用する。rdvno の具体的な内容は実装依存であるが、ランデブ成立相手の呼出側タスクを指定するための情報を含んでいるはずである。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。タイムアウト指定が行われた場合、待ち解除の条件が満足されない(ランデブが成立しない)まま tmout の時間が経過すると、タイムアウトエラー E_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1ミリ秒)と同じである。

tmout として TMO_POL＝0を指定した場合は、タイムアウト値として0を指定したことを示し、対象ランデブポートにランデブ呼出待ちタスクが無いか、ランデブ成立条件が満たされない場合には、待ちに入らず E_TMOUT を返す。また、tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトすることなくランデブの成立まで待ち続ける。

補足事項

ランデブ受付側のタスクも待ち行列を作る機能は、同じ処理をするサーバを複数個並列に走らせるような時に有用である。また、ランデブポートをタスク独立にしたという特徴を活かすことができる。

ランデブを受け付けたタスクが、何らかの理由でランデブ終了前(tk_rpl_rdv 実行前) に異常終了したような場合は、tk_cal_por を実行したランデブ呼出側のタスクがランデブ終了待ち状態から解放されないまま残ることになる。このようなケースを避けるためには、ランデブ受付側タスクの異常終了時に tk_rpl_rdv または tk_rel_wai を実行し、ランデブがエラーで終了したことをランデブ呼出側のタスクにも通知しておく必要がある。

rdvno にはランデブ成立相手の呼出側タスクを指定する情報を含むが、その番号の割当て方法は、できるだけユニークになるように配慮しなければならない。同一タスク間のランデブであっても、1回目のランデブと2回目のランデブでは異なる rdvno を割り当てる必要がある。この配慮により、以下のような問題を回避できる。

tk_cal_por を実行してランデブ終了待ちのタスクが、tk_rel_wai や tk_ter_tsk+tk_sta_tsk 等によって強制的に待ち解除となり、再度 tk_cal_por を実行してランデブが成立した場合を考える。最初のランデブに対する rdvno と後のランデブに対する rdvno が同じ値であると、最初のランデブに対する tk_rpl_rdv によって後のランデブが終了してしまう。rdvno の割当てをユニークにし、ランデブ終了待ちタスクの側で期待する rdvno を覚えておけば、最初のランデブに対する tk_rpl_rdv のエラーを検出することができる。

rdvno の具体的な実現方法としては、たとえば、rdvno の下位バイトを呼出側タスクのID、上位バイトをシーケンシャルな番号とすればよい。

calptn, acpptn によるランデブ成立条件の機能を使えば、ランデブの選択受付(Adaのselectに相当)の機能が実現可能となる。Adaのselect文の例[図 4.7.「select文を使ったAdaのプログラム例」]に相当する具体的な処理方法を[図 4.8.「ランデブによるAdaのselect機能の実現方法」]に示す。

```
select
    when condition_A
        accept entry_A do ... end;
or
    when condition_B
        accept entry_B do ... end;
or
    when condition_C
        accept entry_C do ... end;
end select;
```

図 4.7: select文を使ったAdaのプログラム例

- entry_A, entry_B, entry_C がそれぞれ一つのランデブポートに対応するのではなく、select文全体が一つのランデブポートに対応する。
- entry_A, entry_B, entry_C を、calptn, acpptn の 2 ^ 0, 2 ^ 1, 2 ^ 2 のビットに対応させる。
- Adaのプログラム例の中のselect文は、次のようになる。

```
ptn := 0;
if conditon_A then ptn := ptn + 2^0 endif;
if conditon_B then ptn := ptn + 2^1 endif;
if conditon_C then ptn := ptn + 2^2 endif;
tk_acp_por(acpptn := ptn);
```

- もし、プログラム例中のselect文以外に、select無しの単なるentry_Aのacceptがあれば、

```
tk_acp_por(acpptn := 2^0);
```

を実行すれば良い。また、entry_A, entry_B, entry_Cを無条件にORで待ちたい時は、

```
tk_acp_por(acpptn := 2^2+2^1+2^0);
```

を実行すれば良い。

- 一方、これを呼び出す側は、entry_Aの呼出であれば

```
tk_cal_por(calptn := 2^0);
```

を実行し、entry_Cの呼出であれば

```
tk_cal_por(calptn := 2^2);
```

を実行すれば良い。

図 4.8: ランデブによるAdaのselect機能の実現方法

Adaの選択機能は受け付け側にしか用意されていないが、ランデブでは、calptn で複数のビットを指定することにより、呼出側に選択機能を持たせることも可能である。

仕様決定の理由

ランデブ成立条件に関して呼出側と受付側が全く対称であるにもかかわらず、tk_cal_por と tk_acp_por が別システムコールとなっているのは、ランデブ成立後の処理が異なるからである。すなわち、呼出側はランデブ成立後に待ち状態になるのに対して、受付側はランデブ成立後に実行可能状態(READY)となる。

4.5.3.6 tk_acp_por_u - ランデブポートに対するランデブ受付(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT cmsgsz = tk_acp_por_u (ID porid , UINT acpptn , RNO *p_rdvno , void *msg , TMO_U tmout_u);

パラメータ

ID	porid	Port ID	ランデブポートID
UINT	acpptn	Accept Bit Pattern	受付側選択条件を表すビットパターン
RNO*	p_rdvno	Pointer to Rendezvous Number	リターンパラメータ rdvno を返す領域へのポインタ
void*	msg	Packet of Call Message	呼出メッセージを入れるアドレス
TMO_U	tmout_u	Timeout	タイムアウト指定(マイクロ秒)

リターンパラメータ

RNO	rdvno	Rendezvous Number	ランデブ番号
INT	cmsgsz	Call Message Size	呼出メッセージのサイズ(バイト数)
	または	Error Code	エラーコード

エラーコード

E_ID	不正ID番号(porid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(porid のランデブポートが存在しない)
E_PAR	パラメータエラー(acpptn＝0, msg が不正, tmout_u≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象ランデブポートが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_acp_por のパラメータである tmout を64ビットマイクロ秒単位の tmout_u としたシステムコールである。

パラメータが tmout_u となった点を除き、本システムコールの仕様は tk_acp_por と同じである。詳細は tk_acp_por の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.5.3.7 tk_fwd_por - ランデブポートに対するランデブ回送

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_fwd_por (ID porid , UINT calptn , RNO rdvno , CONST void *msg , INT cmsgsz);

パラメータ

ID	porid	Port ID	回送先のランデブポートID
UINT	calptn	Call Bit Pattern	呼出側選択条件を表すビットパターン
RNO	rdvno	Rendezvous Number	回送前のランデブ番号
CONST void*	msg	Call Message	回送するメッセージを入れるアドレス
INT	cmsgsz	Call Message Size	回送するメッセージのサイズ(バイト数)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(porid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(porid のランデブポートが存在しない)
E_PAR	パラメータエラー(cmsgsz<0, cmsgsz>回送後の maxcmsz, cmsgsz>回送前の maxrmsz, calptn=0, msg が不正)
E_OBJ	オブジェクトの状態が不正(rdvno が不正, 回送後の maxrmsz>回送前の maxrmsz)
E_CTX	コンテキストエラー(タスク独立部から発行,実装依存)
E_DISWAI	待ち禁止による待ち解除

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

一旦受け付けたランデブを別のランデブポートに回送する。

このシステムコールを発行したタスク(タスクXとする)は、現在ランデブ中の状態(tk_acp_por を実行した後の状態)でなければならない。また、ランデブ相手の呼出側タスクはタスクY、tk_acp_por のリターンパラメータとして返されたランデブ番号は rdvno であるものとする。その状態で tk_fwd_por を実行すると、タスクXとタスクYとの間のランデブ状態が解除され、その後 porid で示される別のランデブポート(ランデブポートB)に対してタスクYがランデブ呼出を行ったのと同じ状況になる。

tk_fwd_por の具体的な動作は次のようになる。

1. rdvno で示されるランデブを解除する。
2. タスクYを、porid のランデブポートに対してランデブ呼出待ちの状態にする。この時、ランデブ成立のための呼出側選択条件を表すビットパターン calptn は、タスクYが tk_cal_por で指定したものではなく、タスクXが tk_fwd_por で指定したものが使用される。タスクYから見ると、ランデブ終了待ちの状態からランデブ呼出待ちの状態に戻ることになる。

3. その後、porid のランデブポートに対するランデブが受け付けられれば、それを受け付けたタスクとタスクYとの間でランデブ成立となる。もちろん、porid のランデブポートに既にランデブ受付待ちタスクが存在し、ランデブ成立条件が満たされていれば、tk_fwd_por の実行により即座にランデブ成立となることもある。ここで、ランデブ成立時に受付側に送られるメッセージは、タスクYが tk_cal_por で指定したものではなく、タスクXが tk_fwd_por で指定したものが使用される(calptnと同様)。
4. 新しいランデブが終了した際に tk_rpl_rdv で返されるメッセージは、タスクXが tk_fwd_por で指定した msg 以下の領域ではなく、タスクYが tk_cal_por で指定した msg 以下の領域にコピーされる。

基本的には、

「tk_cal_por (porid=portA, calptn=ptnA, msg=mesA) の後で tk_fwd_por (porid=portB, calptn=ptnB, msg=mesB) が実行された状態」

と、

「tk_cal_por (porid=portB, calptn=ptnB, msg=mesB) が実行された状態」

とは全く同じ状態になる。結果的に、カーネルとしてはランデブ回送の履歴を覚えておく必要はないことになる。

tk_fwd_por によってランデブ呼出待ち状態に戻ったタスクに対して tk_ref_tsk を実行した場合、tskwait は TTW_CAL となる。また、wid も回送先のランデブポートのID番号になる。

tk_fwd_por の実行は即座に終了する。このシステムコールで待ち状態になることはない。また、tk_fwd_por の実行が終わった後の tk_fwd_por 発行タスクは、回送前のランデブが成立したランデブポート、回送後のランデブポート(porid のランデブポート)、それらの上でランデブを行ったタスクのいずれとも無関係になる。

cmsgsz が、回送後のランデブポートの maxcmsz よりも大きい場合は、エラー E_PAR となる。このエラーはランデブの回送を行う前にチェックされる。エラーの場合、ランデブの回送は行われず、rdvno で示されるランデブも解除されない。

tk_fwd_por で指定した送信メッセージは、tk_fwd_por 実行時に他の領域(たとえば tk_cal_por で指定したメッセージ領域)にコピーされる。したがって、回送されたランデブが成立する前に tk_fwd_por の msg で示されるメッセージ領域の内容が変更されても、回送されたランデブはその影響を受けない。

tk_fwd_por でランデブを回送する場合、回送後のランデブポート(porid のランデブポート)の maxrmsz は、回送前のランデブの成立したランデブポートの maxrmsz と等しいか、それよりも小さくなければならない。回送後のランデブポートの maxrmsz が回送前のランデブポートの maxrmsz よりも大きかった場合は、回送先のランデブポートが不適当であるという意味で、E_OBJ のエラーとなる。ランデブ呼出側では、回送前のランデブポートの maxrmsz に合わせて返答メッセージ受信領域を用意しているため、ランデブの回送によって返答メッセージの最大サイズが大きくなると、呼出側に対して予期せぬ大きさの返答メッセージを返す可能性を生じ、問題を起こす。maxrmsz の大きなランデブポートにランデブを回送できないのは、このような理由による。

また、tk_fwd_por で送信するメッセージのサイズ cmsgsz についても、回送前のランデブの成立したランデブポートの maxrmsz と等しいか、それよりも小さくなければならない。これは、tk_fwd_por の実装方法として、tk_cal_por で指定したメッセージ領域を送信メッセージのバッファとして使うことを想定しているためである。cmsgsz が回送前のランデブポートの maxrmsz より大きかった場合には、E_PAR のエラーとなる。(詳細は【補足事項】を参照)

タスク独立部から tk_fwd_por, tk_rpl_rdv を発行する必要はないが、ディスパッチ禁止中あるいは割込み禁止中のタスクから tk_fwd_por, tk_rpl_rdv を発行することは可能である。この機能は、tk_fwd_por や tk_rpl_rdv と不可分に何らかの処理を行う場合に利用できる。なお、タスク独立部から tk_fwd_por, tk_rpl_rdv が発行された場合のエラーチェックは実装依存である。

tk_fwd_por により、ランデブ終了待ち状態であったタスクYがランデブ呼出待ちの状態に戻った場合、次にランデブが成立するまでのタイムアウトは、常に永久待ち(TMO_FEVR) として扱われる。

回送先のランデブポートは、前のランデブに使っていたランデブポート(rdvno のランデブが成立したランデブポート)と同じランデブポートであっても構わない。この場合は、tk_fwd_por によって、一旦受け付けたランデブの受付処理をとりやめることになる。ただし、この場合でも、呼出メッセージや calptn は、呼出側タスクが tk_cal_por で指定したものではなく、受付側タスクが tk_fwd_por で指定したものに変更される。

一旦回送されてきたランデブを、さらに回送することも可能である。

補足事項

tk_fwd_por を使ったサーバタスクの動作イメージを[図 4.9.「tk_fwd_porを使ったサーバタスクの動作イメージ」][2] に示す。

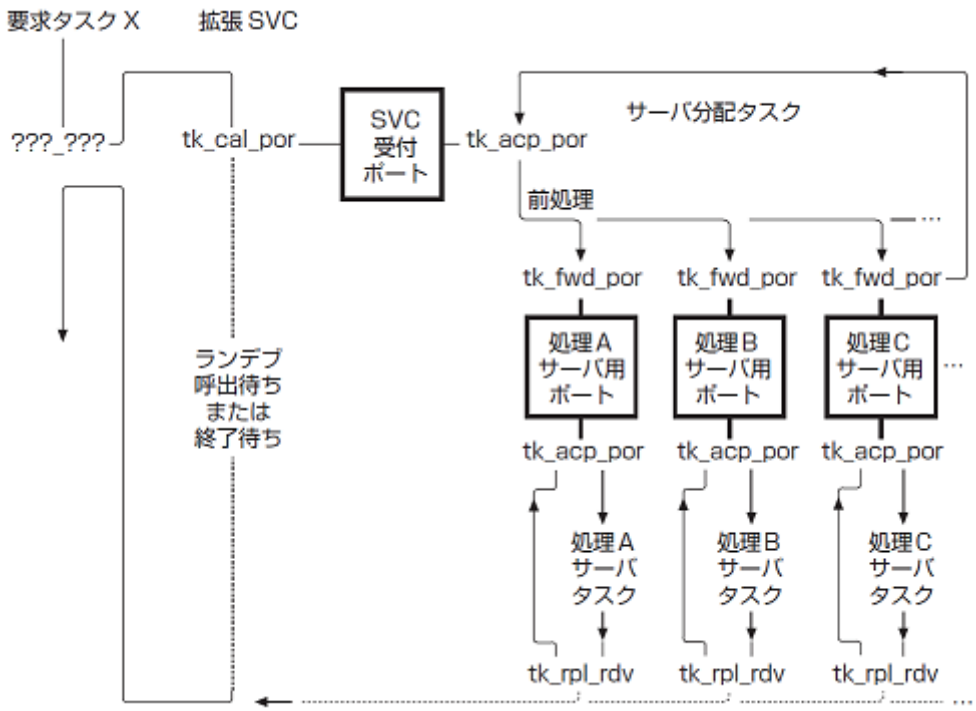

図 4.9: tk_fwd_porを使ったサーバタスクの動作イメージ

一般に、tk_fwd_por を実行するのは、[図 4.9.「tk_fwd_porを使ったサーバタスクの動作イメージ」]に示されるようなサーバ分配タスク(サーバの受け付けた処理を別のタスクに分配するためのタスク)である。したがって、tk_fwd_por を実行したサーバ分配タスクは、回送したランデブの成立の可否にかかわらず、次の要求を受け付ける処理に移らなければならない。その場合、tk_fwd_por のメッセージ領域は次の要求を処理するために利用されるので、メッセージ領域の内容を変更しても、前に処理したランデブの回送には影響しないようにしなければならない。このため、tk_fwd_por の実行後は、回送されたランデブの成立前であっても、tk_fwd_por の msg で示されるメッセージ領域の内容を変更することが可能でなければならない。

実装上は、この仕様を実現するために、tk_cal_por で指定したメッセージ領域をバッファとして使うことが許される。すなわち、tk_fwd_por の処理では、tk_fwd_por で指定した呼出メッセージを tk_cal_por の msg で指定したメッセージ領域にコピーし、tk_fwd_por 発行タスクがメッセージ領域の内容を変更しても構わないようにする。ランデブ成立時には、ランデブが回送されたものかどうかにかかわらず、tk_cal_por のメッセージ領域に置かれていたメッセージが受付側に渡される。

このような実装方法を適用できるように、以下のような仕様を設けている。

- tk_cal_por で要求したランデブが回送される可能性がある場合は、期待する返答メッセージのサイズにかかわらず、tk_cal_por の msg 以下に少なくとも maxrmsz のサイズの領域を確保しておかなければならない。
- tk_fwd_por で送信するメッセージのサイズ cmsgsz は、回送前のランデブポートの maxrmsz と等しいか、それよりも小さくなければならない。
- tk_fwd_por によりランデブが回送される場合に、回送後のランデブポートの maxrmsz が回送前のランデブポートの maxrmsz より大きくなることはない。等しい値をとるか、小さくなっていく。

2

- 太枠内はランデブポート(ランデブエントリ)を表す。
- tk_fwd_por の代わりに tk_cal_por を使うことも可能であるが、その場合はランデブがネストする。処理A～Cのサーバタスクの処理終了後にそのまま要求タスクXの実行を再開して構わないのであれば、tk_fwd_por を利用することによってランデブのネストが不要となり、効率良い動作をすることができる。

仕様決定の理由

システム全体で持つべき状態の数を減らすため、tk_fwd_por の仕様は、ランデブ回送の履歴を保存しないという前提で設計されている。ランデブ回送の履歴を覚える必要がある用途では、tk_fwd_por ではなく、tk_cal_por〜tk_acp_por の組をネストして使えばよい。

4.5.3.8 tk_rpl_rdv - ランデブ返答

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_rpl_rdv (RNO rdvno , CONST void *msg , INT rmsgsz);

パラメータ

RNO	rdvno	Rendezvous Number	ランデブ番号
CONST void*	msg	Reply Message	返答メッセージを入れるアドレス
INT	rmsgsz	Reply Message Size	返答メッセージのサイズ(バイト数)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(rmsgsz<0, rmsgsz>maxrmsz, msg が不正)
E_OBJ	オブジェクトの状態が不正(rdvno が不正)
E_CTX	コンテキストエラー(タスク独立部から発行,実装依存)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ランデブの相手のタスク(呼出側タスク)に対して返答を返し、ランデブを終了する。

このシステムコールを発行したタスク(タスクXとする)は、ランデブ中の状態(tk_acp_por を実行した後の状態)でなければならない。また、ランデブ相手の呼出側タスクはタスクY、tk_acp_por のリターンパラメータとして返されたランデブ番号は rdvno であるものとする。その状態で tk_rpl_rdv を実行すると、タスクXとタスクYとの間のランデブ状態が解除され、ランデブ終了待ち状態にあった呼出側タスクYは実行可能状態(READY)に戻る。

tk_rpl_rdv でランデブを終了する時には、受付側タスクXから呼出側タスクYに対して返答メッセージを送ることができる。受付側タスクが指定した返答メッセージの内容は、呼出側タスクが tk_cal_por で指定した msg 以下の領域にコピーされる。また、返答メッセージのサイズ rmsgsz は、tk_cal_por のリターンパラメータとなる。

rmsgsz が、tk_cre_por で指定した maxrmsz よりも大きい場合は、エラー E_PAR となる。このエラーが検出された場合、ランデブは終了せず、tk_cal_por を実行したタスクのランデブ終了待ち状態は解除されない。

タスク独立部から tk_fwd_por, tk_rpl_rdv を発行することはできないが、ディスパッチ禁止中あるいは割込み禁止中のタスクから tk_fwd_por, tk_rpl_rdv を発行することは可能である。この機能は、tk_fwd_por や tk_rpl_rdv と不可分に何らかの処理を行う場合に利用できる。なお、タスク独立部から tk_fwd_por, tk_rpl_rdv が発行された場合のエラーチェックは実装依存である。

補足事項

ランデブを呼び出したタスクが、何らかの理由でランデブ終了前(tk_rpl_rdv 実行前) に異常終了したような場合にも、ランデブ受付側のタスクは直接それを知ることができない。この場合は、ランデブ受付側のタスクが tk_rpl_rdv を実行する時に E_OBJ のエラーとなる。

ランデブ成立後は、原則としてタスクとランデブポートとが切り離されてしまう(相互に情報を参照する必要がない)が、tk_rpl_rdv のメッセージ長のチェックに使用する maxrmsz のみは、ランデブポートに依存した情報であるため、ランデブ中のタスクがどこかに覚えておく必要がある。実装上の工夫としては、待ち状態になっている呼出側タスクのTCB、あるいはTCBから参照可能な領域(スタック等)に入れておくことが考えられる。

仕様決定の理由

tk_rpl_rdv や tk_fwd_por のパラメータでは、成立中のランデブを区別する情報として rdvno を指定するが、ランデブ成立に利用したランデブポートのID(porid) は指定しない。これは、ランデブ成立後のタスクがランデブポートとは無関係になるという方針に基づいたものである。

rdvno が不正値の場合のエラーコードとして、E_PAR ではなく E_OBJ を使用している。これは、rdvno の実体が呼出側タスクを示しているという理由による。

4.5.3.9 tk_ref_por - ランデブポート状態参照

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_por (ID porid , T_RPOR *pk_rpor);

パラメータ

ID	porid	Port ID	ランデブポートID
T_RPOR*	pk_rpor	Packet to Refer Port Status	ランデブポート状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rpor の内容

void*	exinf	Extended Information	拡張情報
ID	wtsk	Waiting Task ID	呼出待ちタスクのID
ID	atsk	Accept Waiting Task ID	受付待ちタスクのID
INT	maxcmsz	Maximum Call Message Size	呼出時のメッセージの最大長(バイト数)
INT	maxrmsz	Maximum Reply Message Size	返答時のメッセージの最大長(バイト数)

——(以下に実装独自に他の情報を追加してもよい)——

エラーコード

E_OK	正常終了
E_ID	不正ID番号(porid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(porid のランデブポートが存在しない)
E_PAR	パラメータエラー(pk_rpor が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

porid で示された対象ランデブポートの各種の状態を参照し、リターンパラメータとして受付待ちタスクのID(atsk)、呼出待ちタスクのID(wtsk)、メッセージの最大長(maxcmsz,maxrmsz)、拡張情報(exinf) を返す。

wtsk は、このランデブポートでランデブ呼出待ちになっているタスクのIDを示す。ランデブ呼出待ちタスクが無い場合は wtsk＝0となる。一方、atsk は、このランデブポートでランデブ受付待ちになっているタスクのIDを示す。ランデブ受付待ちタスクが無い場合は atsk＝0となる。

このランデブポートで複数のタスクが呼出待ち状態または受付待ち状態になっている場合には、それぞれ呼出待ち行列または受付待ち行列の先頭のタスクのIDを返す。

対象ランデブポートが存在しない場合には、エラー E_NOEXS となる。

補足事項

このシステムコールでは、現在ランデブ中のタスクに関する情報を知ることはできない。

4.6 メモリプール管理機能

メモリプール管理機能は、ソフトウェアによってメモリプールの管理やメモリブロックの割当てを行うための機能である。

メモリプールには、固定長メモリプールと可変長メモリプールがある。両者は別のオブジェクトであり、操作のためのシステムコールが異なっている。固定長メモリプールから獲得されるメモリブロックはサイズが固定されているのに対して、可変長メモリプールから獲得されるメモリブロックでは、任意のブロックサイズを指定することができる。

メモリプール管理機能によって管理されるメモリは共有空間のメモリのみである。タスク固有空間のメモリを管理する機能はない。

4.6.1 固定長メモリプール

固定長メモリプールは、固定されたサイズのメモリブロックを動的に管理するためのオブジェクトである。固定長メモリプール機能には、固定長メモリプールを生成／削除する機能、固定長メモリプールに対してメモリブロックを獲得／返却する機能、固定長メモリプールの状態を参照する機能が含まれる。固定長メモリプールはID番号で識別されるオブジェクトである。固定長メモリプールのID番号を固定長メモリプールIDと呼ぶ。

固定長メモリプールは、固定長メモリプールとして利用するメモリ領域(これを固定長メモリプール領域、または単にメモリプール領域と呼ぶ)と、メモリブロックの獲得を待つタスクの待ち行列を持つ。固定長メモリプールからメモリブロックを獲得するタスクは、メモリプール領域に空きがなくなった場合、次にメモリブロックが返却されるまで固定長メモリブロックの獲得待ち状態となる。固定長メモリブロックの獲得待ち状態になったタスクは、その固定長メモリプールの待ち行列につながれる。

補足事項
固定長メモリプールの場合、何種類かのサイズのメモリブロックが必要となる場合には、サイズ毎に複数のメモリプールを用意する必要がある。

4.6.1.1 tk_cre_mpf - 固定長メモリプール生成

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID mpfid = tk_cre_mpf (CONST T_CMPF *pk_cmpf);

パラメータ

CONST T_CMPF*	pk_cmpf	Packet to Create Memory Pool	固定長メモリプール生成情報

pk_cmpf の内容

void*	exinf	Extended Information	拡張情報
ATR	mpfatr	Memory Pool Attribute	メモリプール属性
INT	mpfcnt	Memory Pool Block Count	メモリプール全体のブロック数
INT	blfsz	Memory Block Size	固定長メモリブロックサイズ(バイト数)
UB	dsname[8]	DS Object name	DSオブジェクト名称

——(以下に実装独自に他の情報を追加してもよい)——

リターンパラメータ

ID	mpfid	Memory Pool ID	固定長メモリプールID
	または	Error Code	エラーコード

エラーコード

E_NOMEM	メモリ不足(管理ブロックやメモリプール用の領域が確保できない)
E_LIMIT	固定長メモリプールの数がシステムの上限を超えた
E_RSATR	予約属性(mpfatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_cmpf が不正, mpfcnt, blfsz が負または不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

固定長メモリプールを生成し固定長メモリプールIDを割り当てる。具体的には、mpfcnt, blfsz の情報を元に、メモリプールとして利用するメモリ領域を確保する。また、生成したメモリプールに対して管理ブロックを割り付ける。ここで生成されたメモリプールに対して tk_get_mpf システムコールを発行することにより、blfsz のサイズ(バイト数)をもつメモリブロックを獲得することができる。

exinf は、対象メモリプールに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、tk_ref_mpf で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。カーネルでは exinf の内容について関知しない。

mpfatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。mpfatr のシステム属性の部分では、次のような指定を行う。

```
mbxatr:= (TA_TFIFO || TA_TPRI) | [TA_DSNAME] | [TA_NODISWAI]
       | (TA_RNG0 || TA_RNG1 || TA_RNG2 || TA_RNG3)
```

TA_TFIFO	メモリ獲得待ちタスクのキューイングはFIFO
TA_TPRI	メモリ獲得待ちタスクのキューイングは優先度順
TA_RNGn	メモリのアクセス制限を保護レベルnとする
TA_DSNAME	DSオブジェクト名称を指定する
TA_NODISWAI	tk_dis_wai による待ち禁止を拒否する

```
#define TA_TFIFO        0x00000000      /* 待ちタスクをFIFOで管理 */
#define TA_TPRI         0x00000001      /* 待ちタスクを優先度順で管理 */
#define TA_DSNAME       0x00000040      /* DSオブジェクト名称を指定 */
#define TA_NODISWAI     0x00000080      /* 待ち禁止拒否 */
#define TA_RNG0         0x00000000      /* 保護レベル0 */
#define TA_RNG1         0x00000100      /* 保護レベル1 */
#define TA_RNG2         0x00000200      /* 保護レベル2 */
#define TA_RNG3         0x00000300      /* 保護レベル3 */
```

TA_TFIFO, TA_TPRI では、タスクがメモリ獲得のためにメモリプールの待ち行列に並ぶ際の並び方を指定することができる。属性が TA_TFIFO であればタスクの待ち行列はFIFOとなり、属性が TA_TPRI であればタスクの待ち行列はタスクの優先度順となる。

TA_RNGn では、メモリのアクセスを制限する保護レベルを指定する。指定された保護レベルと同じかより高い保護レベルで実行しているタスクからのみアクセス可能である。低いレベルで実行しているタスクがアクセスするとCPUの保護違反の例外が発生する。例えば、TA_RNG1 を指定して作成したメモリプールから獲得したメモリは、TA_RNG0, 1 で動作しているタスクからはアクセス可能だが、TA_RNG2, 3 で動作しているタスクからはアクセスできない。

作成されるメモリプールは、共有空間上の常駐メモリとなる。タスク固有空間上にメモリプールを作成する機能はない。

TA_DSNAME を指定した場合に dsname が有効となり、DSオブジェクト名称として設定される。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DSのシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname 、td_set_dsname を参照のこと。TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

補足事項

固定長メモリプールの場合、ブロックサイズを変えるためには別のメモリプールを用意しなければならない。すなわち、何種類かのメモリブロックサイズが必要となる場合は、サイズごとに複数のメモリプールを設ける必要がある。

MMUのないシステムにおいても、移植性確保のために TA_RNGn 属性を受け付けなければならない。例えば、TA_RNGn の指定はすべて TA_RNG0 相当として処理してもよいが、エラーとはしない。

4.6.1.2 tk_del_mpf - 固定長メモリプール削除

C言語インタフェース

```
#include <tk/tkernel.h>

ER ercd = tk_del_mpf (ID mpfid );
```

パラメータ

ID	mpfid	Memory Pool ID	固定長メモリプールID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mpfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mpfid の固定長メモリプールが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

mpfid で示される固定長メモリプールを削除する。

このメモリプールからメモリを獲得しているタスクが存在しても、そのチェックや通知は行われない。すべてのメモリブロックが返却されていなくても、このシステムコールは正常終了する。

本システムコールの発行により、対象メモリプールのID番号および管理ブロック用の領域やメモリプール本体の領域は解放される。

対象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にも、本システムコールは正常終了するが、待ち状態にあったタスクにはエラー E_DLT が返される。

4.6.1.3 tk_get_mpf - 固定長メモリブロック獲得

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_mpf (ID mpfid , void **p_blf , TMO tmout);

パラメータ

ID	mpfid	Memory Pool ID	固定長メモリプールID
void**	p_blf	Pointer to Block Start Address	リターンパラメータ blf を返す領域へのポインタ
TMO	tmout	Timeout	タイムアウト指定(ミリ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード
void*	blf	Block Start Address	メモリブロックの先頭アドレス

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mpfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mpfid の固定長メモリプールが存在しない)
E_PAR	パラメータエラー(tmout≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象メモリプールが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

mpfid で示される固定長メモリプールからメモリブロックを獲得する。獲得したメモリブロックの先頭アドレスが blf に返される。獲得されるメモリブロックのサイズは、固定長メモリプール生成時に blfsz パラメータで指定された値となる。

獲得したメモリのゼロクリアは行われず、獲得されたメモリブロックの内容は不定となる。

指定したメモリプールからメモリブロックが獲得できなければ、tk_get_mpf 発行タスクがそのメモリプールのメモリ獲得待ち行列につながれ、メモリを獲得できるようになるまで待つ。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。タイムアウト指定が行われた場合、待ち解除の条件が満足されない(空きメモリができない)まま tmout の時間が経過すると、タイムアウトエラー E_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1ミリ秒)と同じである。

tmout として TMO_POL＝0を指定した場合は、タイムアウト値として0を指定したことを示し、メモリが獲得できなかった場合も待ちに入ることなく E_TMOUT を返す。

tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずにメモリが獲得できるまで待ち続ける。

メモリブロック獲得待ちを行う場合の待ち行列の順序は、メモリプールの属性によって、FIFOまたはタスク優先度順のいずれかとなる。

4.6.1.4 tk_get_mpf_u - 固定長メモリブロック獲得(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_mpf_u (ID mpfid , void **p_blf , TMO_U tmout_u);

パラメータ

ID	mpfid	Memory Pool ID	固定長メモリプールID
void**	p_blf	Pointer to Block Start Address	リターンパラメータ blf を返す領域へのポインタ
TMO_U	tmout_u	Timeout	タイムアウト指定(マイクロ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード
void*	blf	Block Start Address	メモリブロックの先頭アドレス

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mpfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mpfid の固定長メモリプールが存在しない)
E_PAR	パラメータエラー(tmout_u≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象メモリプールが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_get_mpf のパラメータである tmout を64ビットマイクロ秒単位の tmout_u としたシステムコールである。

パラメータが tmout_u となった点を除き、本システムコールの仕様は tk_get_mpf と同じである。詳細は tk_get_mpf の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.6.1.5 tk_rel_mpf - 固定長メモリブロック返却

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_rel_mpf (ID mpfid , void *blf);

パラメータ

ID	mpfid	Memory Pool ID	固定長メモリプールID
void*	blf	Block Start Address	メモリブロックの先頭アドレス

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mpfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mpfid の固定長メモリプールが存在しない)
E_PAR	パラメータエラー(blf が不正, 異なるメモリプールへの返却)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

blf で示されるメモリブロックを、mpfid で示される固定長メモリプールへ返却する。

tk_rel_mpf の実行により、mpfid のメモリプールでメモリを待っていた別のタスクがメモリを獲得し、そのタスクの待ち状態が解除される場合がある。

メモリブロックの返却を行う固定長メモリプールは、メモリブロックの獲得を行った固定長メモリプールと同じものでなければならない。メモリブロックの返却を行うメモリプールが、メモリブロックの獲得を行ったメモリプールと異なっていることが検出された場合には、E_PAR のエラーとなる。ただし、エラーを検出するか否かは実装依存である。

4.6.1.6 tk_ref_mpf - 固定長メモリプール状態参照

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_mpf (ID mpfid , T_RMPF *pk_rmpf);

パラメータ

ID	mpfid	Memory Pool ID	固定長メモリプールID
T_RMPF*	pk_rmpf	Packet to Refer Memory Pool Status	メモリプール状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rmpf の内容

void*	exinf	Extended Information	拡張情報
ID	wtsk	Waiting Task ID	待ちタスクのID
INT	frbcnt	Free Block Count	空き領域のブロック数

——(以下に実装独自に他の情報を追加してもよい)——

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mpfid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mpfid の固定長メモリプールが存在しない)
E_PAR	パラメータエラー(pk_rmpf が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

mpfid で示された対象固定長メモリプールの各種の状態を参照し、リターンパラメータとして現在の空きブロック数(frbcnt)、待ちタスクのID(wtsk)、拡張情報(exinf) を返す。

wtsk は、この固定長メモリプールで待っているタスクのIDを示す。この固定長メモリプールで複数のタスクが待っている場合には、待ち行列の先頭のタスクのIDを返す。待ちタスクが無い場合は wtsk＝0となる。

tk_ref_mpf で、対象固定長メモリプールが存在しない場合には、エラー E_NOEXS となる。

どんな場合でも、frbcnt＝0とwtsk＝0の少なくとも一方は成り立つ。

補足事項

tk_ref_mpl の frsz ではメモリの空き領域の合計サイズがバイト数で返るのに対して、tk_ref_mpf の frbcnt では空きブロックの数が返る。

4.6.2 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管理するためのオブジェクトである。可変長メモリプール機能には、可変長メモリプールを生成／削除する機能、可変長メモリプールに対してメモリブロックを獲得／返却する機能、可変長メモリプールの状態を参照する機能が含まれる。可変長メモリプールはID番号で識別されるオブジェクトである。可変長メモリプールのID番号を可変長メモリプールIDと呼ぶ。

可変長メモリプールは、可変長メモリプールとして利用するメモリ領域(これを可変長メモリプール領域、または単にメモリプール領域と呼ぶ)と、メモリブロックの獲得を待つタスクの待ち行列を持つ。可変長メモリプールからメモリブロックを獲得するタスクは、メモリプール領域の空き領域が足りなくなった場合、十分なサイズのメモリブロックが返却されるまで可変長メモリブロックの獲得待ち状態となる。可変長メモリブロックの獲得待ち状態になったタスクは、その可変長メモリプールの待ち行列につながれる。

補足事項
可変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は、待ち行列につながれている順序でメモリブロックを獲得する。例えば、ある可変長メモリプールに対して400バイトのメモリブロックを獲得しようとしているタスクAと、100バイトのメモリブロックを獲得しようとしているタスクBが、この順で待ち行列につながれている時に、別のタスクからのメモリブロックの返却により200バイトの連続空きメモリ領域ができたとする。このような場合でも、タスクAがメモリブロックを獲得するまで、タスクBはメモリブロックを獲得できない。

4.6.2.1 tk_cre_mpl - 可変長メモリプール生成

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID mplid = tk_cre_mpl (CONST T_CMPL *pk_cmpl);

パラメータ

CONST T_CMPL*	pk_cmpl	Packet to Create Memory Pool	可変長メモリプール生成情報

pk_cmpl の内容

void*	exinf	Extended Information	拡張情報
ATR	mplatr	Memory Pool Attribute	メモリプール属性
INT	mplsz	Memory Pool Size	メモリプール全体のサイズ(バイト数)
UB	dsname[8]	DS Object name	DSオブジェクト名称
——(以下に実装独自に他の情報を追加してもよい)——			

リターンパラメータ

ID	mplid	Memory Pool ID	可変長メモリプールID
	または	Error Code	エラーコード

エラーコード

E_NOMEM	メモリ不足(管理ブロックやメモリプール用の領域が確保できない)
E_LIMIT	可変長メモリプールの数がシステムの上限を超えた
E_RSATR	予約属性(mplatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_cmpl が不正, mplsz が負または不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

可変長メモリプールを生成し可変長メモリプールIDを割り当てる。具体的には、mplsz の情報を元に、メモリプールとして利用するメモリ領域を確保する。また、生成したメモリプールに対して管理ブロックを割り付ける。

exinf は、対象メモリプールに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、tk_ref_mpl で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。カーネルでは exinf の内容について関知しない。

mplatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。mplatr のシステム属性の部分では、次のような指定を行う。

```
mplatr:= (TA_TFIFO || TA_TPRI) | [TA_DSNAME] | [TA_NODISWAI]
```

```
        | (TA_RNG0 || TA_RNG1 || TA_RNG2 || TA_RNG3)
```

TA_TFIFO	メモリ獲得待ちタスクのキューイングはFIFO
TA_TPRI	メモリ獲得待ちタスクのキューイングは優先度順
TA_RNGn	メモリのアクセス制限を保護レベルnとする
TA_DSNAME	DSオブジェクト名称を指定する
TA_NODISWAI	tk_dis_wai による待ち禁止を拒否する

```
#define TA_TFIFO        0x00000000      /* 待ちタスクをFIFOで管理 */
#define TA_TPRI         0x00000001      /* 待ちタスクを優先度順で管理 */
#define TA_DSNAME       0x00000040      /* DSオブジェクト名称を指定 */
#define TA_NODISWAI     0x00000080      /* 待ち禁止拒否 */
#define TA_RNG0         0x00000000      /* 保護レベル0 */
#define TA_RNG1         0x00000100      /* 保護レベル1 */
#define TA_RNG2         0x00000200      /* 保護レベル2 */
#define TA_RNG3         0x00000300      /* 保護レベル3 */
```

TA_TFIFO, TA_TPRI では、タスクがメモリ獲得のためにメモリプールの待ち行列に並ぶ際の並び方を指定することができる。属性が TA_TFIFO であればタスクの待ち行列はFIFOとなり、属性が TA_TPRI であればタスクの待ち行列はタスクの優先度順となる。

タスクがメモリ獲得待ちの行列を作った場合は、待ち行列先頭のタスクに優先してメモリを割り当てる。待ち行列の2番目以降により少ないメモリサイズを要求しているタスクがあった場合も、そのタスクが先にメモリを獲得することはない。例えば、ある可変長メモリプールに対して 要求メモリサイズ＝400のタスクAと要求メモリサイズ＝100のタスクBがこの順で待っており、別のタスクの tk_rel_mpl により メモリサイズ＝200の連続空きメモリ領域ができたとする。このとき、行列の先頭ではないが要求サイズの少ないタスクBが先にメモリを獲得することはない。

TA_RNGn では、メモリのアクセスを制限する保護レベルを指定する。指定された保護レベルと同じかより高い保護レベルで実行しているタスクからのみアクセス可能である。低いレベルで実行しているタスクがアクセスするとCPUの保護違反の例外が発生する。例えば、TA_RNG1 を指定して作成したメモリプールから獲得したメモリは、TA_RNG0,1 で動作しているタスクからはアクセス可能だが、TA_RNG2,3 で動作しているタスクからはアクセスできない。

作成されるメモリプールは、共有空間上の常駐メモリとなる。タスク固有空間上にメモリプールを作成する機能はない。

TA_DSNAME を指定した場合に dsname が有効となり、DSオブジェクト名称として設定される。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DSのシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname 、td_set_dsname を参照のこと。TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

補足事項

メモリ獲得待ち行列の先頭のタスクの待ちが強制解除されたり、タスク優先度が変更されるなどで待ち行列の順序が変化した場合は、新たに待ち行列の先頭になったタスクに対してメモリ割当てが試みられる。メモリを割り当てることができれば、そのタスクの待ちは解除される。したがって、tk_rel_mpl によるメモリの解放がなくても、状況によってはメモリの獲得が行われ、待ちが解除される場合がある。

MMUのないシステムにおいても、移植性確保のために TA_RNGn 属性を受け付けなければならない。例えば、TA_RNGn の指定はすべて TA_RNG0 相当として処理してもよいが、エラーとはしない。

仕様決定の理由

複数個の可変長メモリプールを設ける機能は、エラー処理時や緊急時などにメモリを確保するためのメモリプールを分離しておくために利用できる。

4.6.2.2 tk_del_mpl - 可変長メモリプール削除

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_del_mpl (ID mplid);

パラメータ

ID	mplid	Memory Pool ID	可変長メモリプールID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mplid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mplid の可変長メモリプールが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

mplid で示される可変長メモリプールを削除する。

このメモリプールからメモリを獲得しているタスクが存在しても、そのチェックや通知は行われない。すべてのメモリブロックが返却されていなくても、このシステムコールは正常終了する。

本システムコールの発行により、対象メモリプールのID番号および管理ブロック用の領域やメモリプール本体の領域は解放される。

対象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にも、本システムコールは正常終了するが、待ち状態にあったタスクにはエラー E_DLT が返される。

4.6.2.3 tk_get_mpl - 可変長メモリブロック獲得

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_mpl (ID mplid , INT blksz , void **p_blk , TMO tmout);

パラメータ

ID	mplid	Memory Pool ID	可変長メモリプールID
INT	blksz	Memory Block Size	メモリブロックサイズ(バイト数)
void**	p_blk	Pointer to Block Start Address	リターンパラメータ blk を返す領域へのポインタ
TMO	tmout	Timeout	タイムアウト指定(ミリ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード
void*	blk	Block Start Address	メモリブロックの先頭アドレス

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mplid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mplid の可変長メモリプールが存在しない)
E_PAR	パラメータエラー(tmout≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象メモリプールが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

mplid で示される可変長メモリプールから、blksz で指定されるサイズ(バイト数)のメモリブロックを獲得する。獲得したメモリブロックの先頭アドレスが blk に返される。

獲得したメモリのゼロクリアは行われず、獲得されたメモリブロックの内容は不定となる。

メモリが獲得できなければ、本システムコールを発行したタスクは待ち状態に入る。

tmout により待ち時間の最大値(タイムアウト値)を指定することができる。タイムアウト指定が行われた場合、待ち解除の条件が満足されない(空きメモリができない)まま tmout の時間が経過すると、タイムアウトエラー E_TMOUT となってシステムコールが終了する。

tmout としては、正の値のみを指定することができる。tmout の基準時間(時間の単位)はシステム時刻の基準時間(＝1ミリ秒)と同じである。

tmout として TMO_POL＝0を指定した場合は、タイムアウト値として0を指定したことを示し、メモリが獲得できなかった場合も待ちに入ることなく E_TMOUT を返す。

tmout として TMO_FEVR＝(-1)を指定した場合は、タイムアウト値として無限大の時間を指定したことを示し、タイムアウトせずにメモリが獲得できるまで待ち続ける。

メモリブロック獲得待ちを行う場合の待ち行列の順序は、メモリプールの属性によって、FIFOまたはタスク優先度順のいずれかとなる。

4.6.2.4 tk_get_mpl_u - 可変長メモリブロック獲得(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_mpl_u (ID mplid , INT blksz , void **p_blk , TMO_U tmout_u);

パラメータ

ID	mplid	Memory Pool ID	可変長メモリプールID
INT	blksz	Memory Block Size	メモリブロックサイズ(バイト数)
void**	p_blk	Pointer to Block Start Address	リターンパラメータ blk を返す領域へのポインタ
TMO_U	tmout_u	Timeout	タイムアウト指定(マイクロ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード
void*	blk	Block Start Address	メモリブロックの先頭アドレス

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mplid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mplid の可変長メモリプールが存在しない)
E_PAR	パラメータエラー(tmout_u≦(-2))
E_DLT	待ちオブジェクトが削除された(待ちの間に対象メモリプールが削除)
E_RLWAI	待ち状態強制解除(待ちの間に tk_rel_wai を受け付け)
E_DISWAI	待ち禁止による待ち解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_get_mpl のパラメータである tmout を64ビットマイクロ秒単位の tmout_u としたシステムコールである。

パラメータが tmout_u となった点を除き、本システムコールの仕様は tk_get_mpl と同じである。詳細は tk_get_mpl の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.6.2.5 tk_rel_mpl - 可変長メモリブロック返却

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_rel_mpl (ID mplid , void *blk);

パラメータ

ID	mplid	Memory Pool ID	可変長メモリプールID
void*	blk	Block Start Address	メモリブロックの先頭アドレス

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mplid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mplid の可変長メモリプールが存在しない)
E_PAR	パラメータエラー(blk が不正, 異なるメモリプールへの返却)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

blk で示されるメモリブロックを、mplid で示される可変長メモリプールへ返却する。

tk_rel_mpl の実行により、mplid のメモリプールでメモリを待っていた別のタスクがメモリを獲得し、そのタスクの待ち状態が解除される場合がある。

メモリブロックの返却を行う可変長メモリプールは、メモリブロックの獲得を行った可変長メモリプールと同じものでなければならない。メモリブロックの返却を行うメモリプールが、メモリブロックの獲得を行ったメモリプールと異なっていることが検出された場合には、E_PAR のエラーとなる。ただし、エラーを検出するか否かは実装依存である。

補足事項

複数のタスクが待っている可変長メモリプールに対してメモリを返却する場合は、要求メモリ数との関係により、複数のタスクが同時に待ち解除となることがある。この場合の待ち解除後のタスクの優先順位は、同じ優先度を持つタスクの間では待ち行列に並んでいたときと同じ順序となる。

4.6.2.6 tk_ref_mpl - 可変長メモリプール状態参照

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_mpl (ID mplid , T_RMPL *pk_rmpl);

パラメータ

ID	mplid	Memory Pool ID	可変長メモリプールID
T_RMPL*	pk_rmpl	Packet to Refer Memory Pool Status	メモリプール状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rmpl の内容

void*	exinf	Extended Information	拡張情報
ID	wtsk	Waiting Task ID	待ちタスクのID
INT	frsz	Free Memory Size	空き領域の合計サイズ(バイト数)
INT	maxsz	Max Memory Size	最大の空き領域のサイズ(バイト数)

——(以下に実装独自に他の情報を追加してもよい)——

エラーコード

E_OK	正常終了
E_ID	不正ID番号(mplid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(mplid の可変長メモリプールが存在しない)
E_PAR	パラメータエラー(pk_rmpl が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

mplid で示された対象可変長メモリプールの各種の状態を参照し、リターンパラメータとして現在の空き領域の合計サイズ(frsz)、すぐに獲得可能な最大の空き領域のサイズ(maxsz)、待ちタスクのID(wtsk)、拡張情報(exinf) を返す。

wtsk は、この可変長メモリプールで待っているタスクのIDを示す。この可変長メモリプールで複数のタスクが待っている場合には、待ち行列の先頭のタスクのIDを返す。待ちタスクが無い場合は wtsk＝0となる。

tk_ref_mpl で、対象可変長メモリプールが存在しない場合には、エラー E_NOEXS となる。

4.7 時間管理機能

時間管理機能は、時間に依存した処理を行うための機能である。システム時刻管理、周期ハンドラ、アラームハンドラの各機能が含まれる。

周期ハンドラとアラームハンドラを総称して、タイムイベントハンドラと呼ぶ。

4.7.1 システム時刻管理

システム時刻管理機能は、システム時刻を操作するための機能である。システム時刻を設定／参照する機能、システム稼働時間を参照する機能が含まれる。

4.7.1.1 tk_set_tim - システム時刻設定

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_set_tim (CONST SYSTIM *pk_tim);

パラメータ

CONST SYSTIM*	pk_tim	Packet of Current Time	現在時刻(ミリ秒)を示すパケット

pk_tim の内容

W	hi	High 32bits	システム時刻設定用現在時刻の上位32ビット
UW	lo	Low 32bits	システム時刻設定用現在時刻の下位32ビット

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(pk_tim が不正, 設定時間が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

システム時刻の値を pk_tim で示される値に設定する。

システム時刻は、1985年1月1日0時(GMT)からの通算のミリ秒数とする。

補足事項

システムの動作中に tk_set_tim を使ってシステム時刻を変更した場合にも、RELTIMやTMOで指定された相対時間は変化しない。例えば、60秒後にタイムアウトする様に指定した場合、タイムアウト待ちの間に tk_set_tim で時間を60秒進めてもそこでタイムアウトすることはなく、60秒後にタイムアウトする。したがって、tk_set_tim によってタイムアウトするシステム時刻は変化することになる。

tk_set_tim() の pk_tim により設定される時刻が、タイマ割込み周期の時間によって制約を受けることはないが、その後の tk_get_tim() で読み出される時刻は、タイマ割込み周期の時間分解能で変化する。たとえば、タイマ割込み周期が10ミリ秒のシステムにおいて、tk_set_tim() で0005(ミリ秒)の時刻を設定した場合に、その後の tk_get_tim() で得られる時刻は、0005(ミリ秒)→0015(ミリ秒)→0025(ミリ秒)のように変化する。

4.7.1.2 tk_set_tim_u - システム時刻設定(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_set_tim_u (SYSTIM_U tim_u);

パラメータ

SYSTIM_U	tim_u	Current Time	現在時刻(マイクロ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(tim_u が不正, 設定時間が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_set_tim のパラメータである pk_tim を64ビットマイクロ秒単位の tim_u としたシステムコールである。

tk_set_tim のパラメータ pk_tim は構造体SYSTIMを使ったパケット渡しであるが、tk_set_tim_u のパラメータ tim_u はパケット渡しではなく、64ビット符号付き整数のSYSTIM_Uを使った値渡しである。

上記の点を除き、本システムコールの仕様は tk_set_tim と同じである。詳細は tk_set_tim の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.7.1.3 tk_get_tim - システム時刻参照

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_tim (SYSTIM *pk_tim);

パラメータ

SYSTIM*	pk_tim	Packet of Current Time	現在時刻(ミリ秒)を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_tim の内容

W	hi	High 32bits	システムの現在時刻の上位32ビット
UW	lo	Low 32bits	システムの現在時刻の下位32ビット

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(pk_tim が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

システム時刻の現在の値を読み出し、リターンパラメータ pk_tim に返す。

システム時刻は、1985年1月1日0時(GMT)からの通算のミリ秒数とする。

本システムコールで読み出されるシステムの現在時刻は、タイマ割込み間隔(周期)の分解能で変化する。

補足事項

tk_get_tim() では、タイマ割込み間隔(周期)より短い時間経過を知ることはできない。タイマ割込み間隔(周期)より短い時間経過を知る必要がある場合は、tk_get_tim_u() または td_get_tim() の ofs を利用する。

4.7.1.4 tk_get_tim_u - システム時刻参照(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_tim_u (SYSTIM_U *tim_u , UINT *ofs);

パラメータ

SYSTIM_U*	tim_u	Time	現在時刻(マイクロ秒)を返す領域へのポインタ
UINT*	ofs	Offset	リターンパラメータ ofs を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード
SYSTIM_U	tim_u	Time	現在時刻(マイクロ秒)
UINT	ofs	Offset	tim_u からの相対的な経過時間(ナノ秒)

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(tim_u, ofs が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_get_tim のリターンパラメータである pk_tim を64ビットマイクロ秒単位の tim_u としたシステムコールである。また、ナノ秒単位の相対時間を返す ofs のリターンパラメータを追加している。

tim_u はタイマ割込み間隔(周期)の分解能となるが、さらに細かい精度の時刻情報として、tim_u からの経過時間を ofs にナノ秒単位で取得する。ofs の分解能は実装依存であるが、一般にはハードウェアタイマの分解能となる。

ofs＝NULL とした場合には、ofs の情報は格納されない。

上記の点を除き、本システムコールの仕様は tk_get_tim と同じである。また、tim_u のデータタイプがSYSTIM_Uとなっている点を除き、本システムコールの仕様は td_get_tim と同じである。詳細は tk_get_tim および td_get_tim の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.7.1.5 tk_get_otm - システム稼働時間参照

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_otm (SYSTIM *pk_tim);

パラメータ

SYSTIM*	pk_tim	Packet of Operating Time	稼働時間(ミリ秒)を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_tim の内容

W	hi	High 32bits	システム稼働時間の上位32ビット
UW	lo	Low 32bits	システム稼働時間の下位32ビット

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(pk_tim が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

システム稼働時間を取得する。

システム稼働時間はシステム時刻(時刻)と異なり、システム起動時からの単純増加する稼働時間を表す。tk_set_tim による時刻設定に影響されない。

システム稼働時間はシステム時刻と同じ精度でなければならない。

4.7.1.6 tk_get_otm_u - システム稼働時間参照(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_otm_u (SYSTIM_U *tim_u , UINT *ofs);

パラメータ

SYSTIM_U*	tim_u	Time	稼働時間(マイクロ秒)を返す領域へのポインタ
UINT*	ofs	Offset	リターンパラメータ ofs を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード
SYSTIM_U	tim_u	Time	稼働時間(マイクロ秒)
UINT	ofs	Offset	tim_u からの相対的な経過時間(ナノ秒)

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(tim_u, ofs が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_get_otm のリターンパラメータである pk_tim を64ビットマイクロ秒単位の tim_u としたシステムコールである。また、ナノ秒単位の相対時間を返す ofs のリターンパラメータを追加している。

tim_u はタイマ割込み間隔(周期)の分解能となるが、さらに細かい精度の時刻情報として、tim_u からの経過時間を ofs にナノ秒単位で取得する。ofs の分解能は実装依存であるが、一般にはハードウェアタイマの分解能となる。

ofs＝NULL とした場合には、ofs の情報は格納されない。

上記の点を除き、本システムコールの仕様は tk_get_otm と同じである。また、tim_u のデータタイプがSYSTIM_Uとなっている点を除き、本システムコールの仕様は td_get_otm と同じである。詳細は tk_get_otm および td_get_otm の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.7.2 周期ハンドラ

周期ハンドラは、一定周期で起動されるタイムイベントハンドラである。周期ハンドラ機能には、周期ハンドラを生成／削除する機能、周期ハンドラの動作を開始／停止する機能、周期ハンドラの状態を参照する機能が含まれる。周期ハンドラはID番号で識別されるオブジェクトである。周期ハンドラのID番号を周期ハンドラIDと呼ぶ。

周期ハンドラの起動周期と起動位相は、周期ハンドラの生成時に、周期ハンドラ毎に設定することができる。カーネルは、周期ハンドラの操作時に、設定された起動周期と起動位相から、周期ハンドラを次に起動すべき時刻を決定する。周期ハンドラの生成時には、周期ハンドラを生成した時刻に起動位相を加えた時刻を、次に起動すべき時刻とする。周期ハンドラを起動すべき時刻になると、その周期ハンドラの拡張情報(exinf)をパラメータとして、周期ハンドラを起動する。またこの時、周期ハンドラの起動すべき時刻に起動周期を加えた時刻を、次に起動すべき時刻とする。また、周期ハンドラの動作を開始する時に、次に起動すべき時刻を決定しなおす場合がある。

周期ハンドラの起動位相は、起動周期以下であることを原則とする。起動位相に、起動周期よりも長い時間が指定された場合の振舞いは、実装依存とする。

周期ハンドラは、動作している状態か動作していない状態かのいずれかの状態をとる。周期ハンドラが動作していない状態の時には、周期ハンドラを起動すべき時刻となっても周期ハンドラを起動せず、次に起動すべき時刻の決定のみを行う。周期ハンドラの動作を開始するシステムコール(tk_sta_cyc)が呼び出されると、周期ハンドラを動作している状態に移行させ、必要なら周期ハンドラを次に起動すべき時刻を決定しなおす。周期ハンドラの動作を停止するシステムコール(tk_stp_cyc)が呼び出されると、周期ハンドラを動作していない状態に移行させる。周期ハンドラを生成した後にどちらの状態になるかは、周期ハンドラ属性によって決めることができる。

周期ハンドラの起動位相は、周期ハンドラを生成するシステムコールが呼び出された時刻を基準に、周期ハンドラを最初に起動する時刻を指定する相対時間と解釈する。周期ハンドラの起動周期は、周期ハンドラを(起動した時刻ではなく)起動すべきであった時刻を基準に、周期ハンドラを次に起動する時刻を指定する相対時間と解釈する。そのため、周期ハンドラが起動される時刻の間隔は、個々には起動周期よりも短くなる場合があるが、長い期間で平均すると起動周期に一致する。

補足事項
T-Kernelの時間管理機能の処理における実際の時間分解能は、「5.7.2節「標準システム構成情報」」の「タイマ割込み間隔」(TTimPeriod)によって指定されたものとなる。すなわち、周期ハンドラやアラームハンドラが実際に起動される時刻も、タイマ割込み間隔(TTimPeriod)の分解能にしたがったものとなる。このため、周期ハンドラが実際に起動されるのは、周期ハンドラを起動すべき時刻の直後のタイマ割込みの時点である。T-Kernelの一般的な実装においては、タイマ割込みの処理の中で起動すべき周期ハンドラやアラームハンドラの有無を検出し、必要に応じてそれらのハンドラを起動する。

4.7.2.1 tk_cre_cyc - 周期ハンドラの生成

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID cycid = tk_cre_cyc (CONST T_CCYC *pk_ccyc);

パラメータ

CONST T_CCYC*	pk_ccyc	Packet to Create Cyclic Handler	周期ハンドラ定義情報

pk_ccyc の内容

void*	exinf	Extended Information	拡張情報
ATR	cycatr	Cyclic Handler Attribute	周期ハンドラ属性
FP	cychdr	Cyclic Handler Address	周期ハンドラアドレス
RELTIM	cyctim	Cycle Time	周期起動時間間隔(ミリ秒)
RELTIM	cycphs	Cycle Phase	周期起動位相(ミリ秒)
UB	dsname[8]	DS Object name	DSオブジェクト名称

——(以下に実装独自に他の情報を追加してもよい)——

リターンパラメータ

ID	cycid	Cyclic Handler ID	周期ハンドラID
	または	Error Code	エラーコード

エラーコード

E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_LIMIT	周期ハンドラの数がシステムの制限を超えた
E_RSATR	予約属性(cycatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_ccyc, cychdr, cyctim, cycphs が不正あるいは利用できない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

周期ハンドラを生成し周期ハンドラIDを割り当てる。具体的には、生成された周期ハンドラに対して管理ブロックを割り付ける。

周期ハンドラは、指定した時間間隔で動くタスク独立部のハンドラである。

exinf は、対象となる周期ハンドラに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、周期ハンドラにパラメータとして渡される他、tk_ref_cyc で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。カーネルでは exinf の内容について関知しない。

cycatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。cycatr のシステム属性の部分では、次のような指定を行う。

cycatr := (TA_ASM || TA_HLNG) | [TA_STA] | [TA_PHS] | [TA_DSNAME]

TA_ASM	対象ハンドラがアセンブリ言語で書かれている
TA_HLNG	対象ハンドラが高級言語で書かれている
TA_STA	周期ハンドラ生成後直ちに起動する
TA_PHS	起動位相を保存する
TA_DSNAME	DSオブジェクト名称を指定する

```
#define TA_ASM          0x00000000      /* アセンブリ言語によるプログラム */
#define TA_HLNG         0x00000001      /* 高級言語によるプログラム */
#define TA_STA          0x00000002      /* 周期ハンドラ起動 */
#define TA_PHS          0x00000004      /* 周期ハンドラ起動位相を保存 */
#define TA_DSNAME       0x00000040      /* DSオブジェクト名称を指定 */
```

cychdr は周期ハンドラの先頭アドレス、cyctim は周期起動の時間間隔、cycphs は起動位相を表す。

TA_HLNG 属性の場合は、高級言語対応ルーチンを経由して周期ハンドラを起動する。高級言語対応ルーチンによって、レジスタの退避と復帰が行われる。周期ハンドラからは、単純な関数からのリターンによって終了する。TA_HLNG 属性の場合の周期ハンドラは次の形式となる。

```
void cychdr( void *exinf )
{
        /*
                処理
        */

        return; /* 周期ハンドラの終了 */
}
```

TA_ASM 属性の場合の周期ハンドラの形式は実装定義とする。ただし、起動パラメータとして exinf を渡さなければならない。

cycphs は tk_cre_cyc によって周期ハンドラを生成してから最初の周期ハンドラの起動までの時間を表す。その後は、cyctim 間隔で周期起動を繰り返す。cycphs に0を指定した場合は、周期ハンドラの生成直後に周期ハンドラが起動される。cyctim に0を指定することはできない。

周期ハンドラのn回目の起動は、周期ハンドラを生成してから cycphs + cyctim * (n - 1) 以上の時間が経過した後に行う。

TA_STA を指定した場合は、周期ハンドラの生成時から周期ハンドラは動作状態となり、前述の時間間隔で周期ハンドラが起動される。TA_STA を指定しなかった場合は、起動周期の計測は行われるが、周期ハンドラは起動されない。

TA_PHS が指定されている場合は、tk_sta_cyc によって周期ハンドラが活性化されても、起動周期はリセットされず前述のように周期ハンドラの生成時から計測している周期を維持する。TA_PHS が指定されていない場合は、tk_sta_cyc によって起動周期がリセットされ、tk_sta_cyc 呼出時から cyctim 間隔で周期ハンドラが起動される。tk_sta_cyc によるリセットでは、cycphs は適用されない。この場合、tk_sta_cyc からn回目の周期ハンドラの起動は、tk_sta_cyc 呼出時から cyctim * n 以上の時間が経過した後となる。

周期ハンドラの中でシステムコールを発行することにより、それまで実行状態(RUNNING)であったタスクがその他の状態に移行し、代わりに別のタスクが実行状態(RUNNING)となった場合でも、周期ハンドラ実行中はディスパッチ(実行タスクの切り替え)が起こらない。ディスパッチングが必要になっても、まず周期ハンドラを最後まで実行することが優先され、周期ハンドラを終了する時にはじめてディスパッチが行われる。すなわち、周期ハンドラ実行中に生じたディスパッチ要求はすぐに処理されず、周期ハンドラ終了までディスパッチが遅らされる。これを遅延ディスパッチの原則と呼ぶ。

周期ハンドラはタスク独立部として実行される。したがって、周期ハンドラの中では、待ち状態に入るシステムコールや、自タスクの指定を意味するシステムコールを実行することはできない。

TA_DSNAME を指定した場合に dsname が有効となり、DSオブジェクト名称として設定される。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DSのシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname 、td_set_dsname を参照のこと。TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

補足事項

tk_cre_cyc では回数の指定が無いので、一旦定義された周期ハンドラは、tk_stp_cyc によって停止するか、周期起動ハンドラを削除するまで周期起動が繰り返される。

複数のタイムイベントハンドラや割込みハンドラが同時に動く場合に、それらのハンドラがシリアルに起動される(1つのハンドラの実行を終了してから別のハンドラの実行を始める)か、ネストして起動される(1つのハンドラの実行を中断して別のハンドラを実行し、そのハンドラが終了した後で前のハンドラの続きを実行する)かということは実装依存である。いずれにしても、タイムイベントハンドラや割込みハンドラはタスク独立部なので、遅延ディスパッチの原則が適用される。

cycphs に0を指定した場合、1回目の周期ハンドラの起動は、本システムコールの実行直後になる。ただし、実装上の都合により、本システムコールの実行終了の直後に周期ハンドラを起動(実行)するのではなく、本システムコールの処理中に1回目の周期ハンドラを起動(実行)する場合がある。この場合、周期ハンドラ内での割込み禁止などの状態が、2回目以降の通常の周期ハンドラ起動時とは異なっていることがある。また、cycphs に0を指定した場合の1回目の周期ハンドラの起動は、タイマ割込みを待つことなく、すなわちタイマ割込み間隔とは無関係に起動される。この点についても、2回目以降の通常の周期ハンドラの起動や、cycphs が0でない場合の周期ハンドラの起動とは異なっている。

4.7.2.2 tk_cre_cyc_u - 周期ハンドラの生成(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID cycid = tk_cre_cyc_u (CONST T_CCYC_U *pk_ccyc_u);

パラメータ

CONST T_CCYC_U*	pk_ccyc_u	Packet to Create Cyclic Handler	周期ハンドラ定義情報

pk_ccyc_u の内容

void*	exinf	Extended Information	拡張情報
ATR	cycatr	Cyclic Handler Attribute	周期ハンドラ属性
FP	cychdr	Cyclic Handler Address	周期ハンドラアドレス
RELTIM_U	cyctim_u	Cycle Time	周期起動時間間隔(マイクロ秒)
RELTIM_U	cycphs_u	Cycle Phase	周期起動位相(マイクロ秒)
UB	dsname[8]	DS Object name	DSオブジェクト名称
——(以下に実装独自に他の情報を追加してもよい)——			

リターンパラメータ

ID	cycid	Cyclic Handler ID	周期ハンドラID
	または	Error Code	エラーコード

エラーコード

E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_LIMIT	周期ハンドラの数がシステムの制限を超えた
E_RSATR	予約属性(cycatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_ccyc_u, cychdr, cyctim_u, cycphs_u が不正あるいは利用できない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_cre_cyc のパラメータである cyctim と cycphs を64ビットマイクロ秒単位の cyctim_u, cycphs_u としたシステムコールである。

パラメータが cyctim_u, cycphs_u となった点を除き、本システムコールの仕様は tk_cre_cyc と同じである。詳細は tk_cre_cyc の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.7.2.3 tk_del_cyc - 周期ハンドラの削除

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_del_cyc (ID cycid);

パラメータ

ID	cycid	Cyclic Handler ID	周期ハンドラID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(cycid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(cycid の周期ハンドラが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

周期ハンドラを削除する。

4.7.2.4 tk_sta_cyc - 周期ハンドラの動作開始

C言語インタフェース

```
#include <tk/tkernel.h>

ER ercd = tk_sta_cyc (ID cycid );
```

パラメータ

ID	cycid	Cyclic Handler ID	周期ハンドラID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(cycid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(cycid の周期ハンドラが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

周期ハンドラを動作状態にする。

TA_PHS 属性を指定している場合は、周期ハンドラの起動周期はリセットされず、周期ハンドラを動作状態にする。すでに動作状態であれば何もせずに動作状態を維持する。

TA_PHS 属性を指定していない場合は、起動周期をリセットして、周期ハンドラを動作状態にする。すでに、動作状態であれば起動周期のリセットをした上で、動作状態を維持する。したがって、次に周期ハンドラが起動されるのは、cyctim 後となる。

4.7.2.5 tk_stp_cyc - 周期ハンドラの動作停止

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_stp_cyc (ID cycid);

パラメータ

ID	cycid	Cyclic Handler ID	周期ハンドラID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(cycid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(cycid の周期ハンドラが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

周期ハンドラを動作していない状態にする。すでに動作していない状態であれば何もしない。

4.7.2.6 tk_ref_cyc - 周期ハンドラ状態参照

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_cyc (ID cycid , T_RCYC *pk_rcyc);

パラメータ

ID	cycid	Cyclic Handler ID	周期ハンドラID
T_RCYC*	pk_rcyc	Packet to Refer Cyclic Handler Status	周期ハンドラの状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rcyc の内容

void*	exinf	Extended Information	拡張情報
RELTIM	lfttim	Left Time	次のハンドラ起動までの残り時間(ミリ秒)
UINT	cycstat	Cyclic Handler Status	周期ハンドラの状態

――(以下に実装独自に他の情報を追加してもよい)――

エラーコード

E_OK	正常終了
E_ID	不正ID番号(cycid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(cycid の周期ハンドラが存在していない)
E_PAR	パラメータエラー(pk_rcyc が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

cycid で示された周期ハンドラの状態を参照し、周期ハンドラの状態(cycstat)、次のハンドラ起動までの残り時間(lfttim)、拡張情報(exinf)をリターンパラメータとして返す。

cycstat には次の情報が返される。

```
cycstat:= (TCYC_STP | TCYC_STA)
```

```
#define TCYC_STP        0x00    /* 周期ハンドラが動作していない */
#define TCYC_STA        0x01    /* 周期ハンドラが動作している */
```

lfttim には、次に周期ハンドラが起動される予定の時刻までの残り時間(ミリ秒)が返される。周期ハンドラが現在起動中か停止中かは関係しない。

exinf には、周期ハンドラを生成する際のパラメータとして指定された拡張情報が返される。exinf は周期ハンドラに引数として渡される。

tk_ref_cyc で、cycid の周期ハンドラが存在していない場合には、エラー E_NOEXS となる。

周期ハンドラ状態情報(T_RCYC)における残り時間 lfttim は、ミリ秒単位に切り上げた値(単位ミリ秒)を返す。マイクロ秒単位の情報を知りたい場合には、tk_ref_cyc_u を使う。

4.7.2.7 tk_ref_cyc_u - 周期ハンドラ状態参照(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_cyc_u (ID cycid , T_RCYC_U *pk_rcyc_u);

パラメータ

ID	cycid	Cyclic Handler ID	周期ハンドラID
T_RCYC_U*	pk_rcyc_u	Packet to Refer Cyclic Handler Status	周期ハンドラの状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rcyc_u の内容

void*	exinf	Extended Information	拡張情報
RELTIM_U	lfttim_u	Left Time	次のハンドラ起動までの残り時間(マイクロ秒)
UINT	cycstat	Cyclic Handler Status	周期ハンドラの状態

――(以下に実装独自に他の情報を追加してもよい)――

エラーコード

E_OK	正常終了
E_ID	不正ID番号(cycid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(cycid の周期ハンドラが存在していない)
E_PAR	パラメータエラー(pk_rcyc_u が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

tk_ref_cyc のリターンパラメータである lfttim を64ビットマイクロ秒単位の lfttim_u としたシステムコールである。

リターンパラメータが lfttim_u となった点を除き、本システムコールの仕様は tk_ref_cyc と同じである。詳細は tk_ref_cyc の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.7.3 アラームハンドラ

アラームハンドラは、指定した時刻に起動されるタイムイベントハンドラである。アラームハンドラ機能には、アラームハンドラを生成／削除する機能、アラームハンドラの動作を開始／停止する機能、アラームハンドラの状態を参照する機能が含まれる。アラームハンドラはID番号で識別されるオブジェクトである。アラームハンドラのID番号をアラームハンドラIDと呼ぶ。

アラームハンドラを起動する時刻(これをアラームハンドラの起動時刻と呼ぶ)は、アラームハンドラ毎に設定することができる。アラームハンドラの起動時刻になると、そのアラームハンドラの拡張情報(exinf)をパラメータとして、アラームハンドラを起動する。

アラームハンドラの生成直後には、アラームハンドラの起動時刻は設定されておらず、アラームハンドラの動作は停止している。アラームハンドラの動作を開始するシステムコール(tk_sta_alm)が呼び出されると、アラームハンドラの起動時刻を、システムコールが呼び出された時刻から指定された相対時間後に設定する。アラームハンドラの動作を停止するシステムコール(tk_stp_alm)が呼び出されると、アラームハンドラの起動時刻の設定を解除する。また、アラームハンドラを起動する時にも、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する。

補足事項
アラームハンドラが実際に起動される時刻は、タイマ割込み間隔(TTimPeriod)の分解能にしたがったものとなる。詳細は「4.7.2節「周期ハンドラ」」の補足事項を参照。

4.7.3.1 tk_cre_alm - アラームハンドラの生成

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID almid = tk_cre_alm (CONST T_CALM *pk_calm);

パラメータ

CONST T_CALM*	pk_calm	Packet to Create Alarm Handler	アラームハンドラ定義情報

pk_calm の内容

void*	exinf	Extended Information	拡張情報
ATR	almatr	Alarm Handler Attribute	アラームハンドラ属性
FP	almhdr	Alarm Handler Address	アラームハンドラアドレス
UB	dsname[8]	DS Object name	DSオブジェクト名称

――(以下に実装独自に他の情報を追加してもよい)――

リターンパラメータ

ID	almid	Alarm Handler ID	アラームハンドラID
	または	Error Code	エラーコード

エラーコード

E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_LIMIT	アラームハンドラの数がシステムの制限を超えた
E_RSATR	予約属性(almatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_calm, almhdr が不正あるいは利用できない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

アラームハンドラを生成しアラームハンドラIDを割り当てる。具体的には、生成されたアラームハンドラに対して管理ブロックを割り付ける。

アラームハンドラ(指定時刻起動ハンドラ)は、指定した時刻に起動されるタスク独立部のハンドラである。

exinf は、対象となるアラームハンドラに関する情報を入れておくためにユーザが自由に利用できる。ここで設定した情報は、アラームハンドラにパラメータとして渡される他、tk_ref_alm で取り出すことができる。なお、ユーザの情報を入れるためにもっと大きな領域がほしい場合や、途中で内容を変更したい場合には、自分でそのためのメモリを確保し、そのメモリパケットのアドレスを exinf に入れる。カーネルでは exinf の内容について関知しない。

almatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。almatr のシステム属性の部分では、次のような指定を行う。

almatr := (TA_ASM || TA_HLNG) | [TA_DSNAME]

TA_ASM	対象ハンドラがアセンブリ言語で書かれている
TA_HLNG	対象ハンドラが高級言語で書かれている
TA_DSNAME	DSオブジェクト名称を指定する

```
#define TA_ASM          0x00000000      /* アセンブリ言語によるプログラム */
#define TA_HLNG         0x00000001      /* 高級言語によるプログラム */
#define TA_DSNAME       0x00000040      /* DSオブジェクト名称を指定 */
```

almhdr は起動されるアラームハンドラの先頭アドレスを表す。

TA_HLNG 属性の場合は、高級言語対応ルーチンを経由してアラームハンドラを起動する。高級言語対応ルーチンによって、レジスタの退避と復帰が行われる。アラームハンドラからは、単純な関数からのリターンによって終了する。TA_HLNG 属性の場合のアラームハンドラは次の形式となる。

```
void almhdr( void *exinf )
{
        /*
                処理
        */

        return; /* アラームハンドラの終了 */
}
```

TA_ASM 属性の場合のアラームハンドラの形式は実装定義とする。ただし、起動パラメータとして exinf を渡さなければならない。

アラームハンドラの中でシステムコールを発行することにより、それまで実行状態(RUNNING)であったタスクがその他の状態に移行し、代わりに別のタスクが実行状態(RUNNING)となった場合でも、アラームハンドラ実行中はディスパッチ(実行タスクの切り替え)が起こらない。ディスパッチングが必要になっても、まずアラームハンドラを最後まで実行することが優先され、アラームハンドラを終了する時にはじめてディスパッチが行われる。すなわち、アラームハンドラ実行中に生じたディスパッチ要求はすぐに処理されず、アラームハンドラ終了までディスパッチが遅らされる。これを遅延ディスパッチの原則と呼ぶ。

アラームハンドラはタスク独立部として実行される。したがって、アラームハンドラの中では、待ち状態に入るシステムコールや、自タスクの指定を意味するシステムコールを実行することはできない。

TA_DSNAME を指定した場合に dsname が有効となり、DSオブジェクト名称として設定される。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、T-Kernel/DSのシステムコール td_ref_dsname と td_set_dsname からのみ操作可能である。詳細は td_ref_dsname 、td_set_dsname を参照のこと。TA_DSNAME を指定しなかった場合は、dsname が無視され、td_ref_dsname や td_set_dsname が、E_OBJ エラーとなる。

補足事項

複数のタイムイベントハンドラや割込みハンドラが同時に動く場合に、それらのハンドラがシリアルに起動される(1つのハンドラの実行を終了してから別のハンドラの実行を始める)か、ネストして起動される(1つのハンドラの実行を中断して別のハンドラを実行し、そのハンドラが終了した後で前のハンドラの続きを実行する)かということは実装依存である。いずれにしても、タイムイベントハンドラや割込みハンドラはタスク独立部なので、遅延ディスパッチの原則が適用される。

4.7.3.2 tk_del_alm - アラームハンドラの削除

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_del_alm (ID almid);

パラメータ

ID	almid	Alarm Handler ID	アラームハンドラID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(almid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(almid のアラームハンドラが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

アラームハンドラを削除する。

4.7.3.3 tk_sta_alm - アラームハンドラの動作開始

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_sta_alm (ID almid , RELTIM almtim);

パラメータ

ID	almid	Alarm Handler ID	アラームハンドラID
RELTIM	almtim	Alarm Time	アラームハンドラ起動時刻(ミリ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(almid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(almid のアラームハンドラが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

アラームハンドラの起動時刻を設定して、動作状態にする。almtim は相対時刻で、tk_sta_alm が呼び出された時刻から almtim で指定した時間が経過した後にアラームハンドラが起動される。すでにアラームハンドラの起動時刻が設定されて動作状態であった場合には、その設定を解除した後、新たに起動時刻を設定し動作状態とする。

almtim＝0の場合には、起動時刻設定直後にアラームハンドラが起動される。

4.7.3.4 tk_sta_alm_u - アラームハンドラの動作開始(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_sta_alm_u (ID almid , RELTIM_U almtim_u);

パラメータ

ID	almid	Alarm Handler ID	アラームハンドラID
RELTIM_U	almtim_u	Alarm Time	アラームハンドラ起動時刻(マイクロ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(almid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(almid のアラームハンドラが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

tk_sta_alm のパラメータである almtim を64ビットマイクロ秒単位の almtim_u としたシステムコールである。

パラメータが almtim_u となった点を除き、本システムコールの仕様は tk_sta_alm と同じである。詳細は tk_sta_alm の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.7.3.5 tk_stp_alm - アラームハンドラの動作停止

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_stp_alm (ID almid);

パラメータ

ID	almid	Alarm Handler ID	アラームハンドラID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(almid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(almid のアラームハンドラが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

アラームハンドラの起動時刻を解除し、動作していない状態にする。すでに動作していない状態であれば何もしない。

4.7.3.6 tk_ref_alm - アラームハンドラ状態参照

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_alm (ID almid , T_RALM *pk_ralm);

パラメータ

ID	almid	Alarm Handler ID	アラームハンドラID
T_RALM*	pk_ralm	Packet to Refer Alarm Handler Status	アラームハンドラの状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_ralm の内容

void*	exinf	Extended Information	拡張情報
RELTIM	lfttim	Left Time	ハンドラ起動までの残り時間(ミリ秒)
UINT	almstat	Alarm Handler Status	アラームハンドラの状態

――(以下に実装独自に他の情報を追加してもよい)――

エラーコード

E_OK	正常終了
E_ID	不正ID番号(almid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(almid のアラームハンドラが存在しない)
E_PAR	パラメータエラー(pk_ralm が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

almid で示されたアラームハンドラの状態を参照し、ハンドラ起動までの残り時間(lfttim)、拡張情報(exinf)をリターンパラメータとして返す。

almstat には次の情報が返される。

almstat:= (TALM_STP | TALM_STA)

```
#define TALM_STP        0x00    /* アラームハンドラが動作していない */
#define TALM_STA        0x01    /* アラームハンドラが動作している */
```

アラームハンドラが動作している(TALM_STA)場合、lfttim には次にアラームハンドラが起動されるまでの相対時間が返される。tk_sta_alm で指定した almtim ≧ lfttim ≧ 0 の範囲の値となる。lfttim はタイマ割込みごとに減算されるため、次のタイマ割込みでアラームハンドラが起動される場合に lfttim＝0 となる。

exinf には、アラームハンドラを生成する際のパラメータとして指定された拡張情報が返される。exinf はアラームハンドラに引数として渡される。

アラームハンドラが動作していない(TALM_STP)場合は、lfttim は不定である。

tk_ref_alm で、almid のアラームハンドラが存在していない場合には、エラー E_NOEXS となる。

アラームハンドラ状態情報(T_RALM)における残り時間 lfttim は、ミリ秒単位に切り上げた値(単位ミリ秒)を返す。マイクロ秒単位の情報を知りたい場合には、tk_ref_alm_u を使う。

4.7.3.7 tk_ref_alm_u - アラームハンドラ状態参照(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_alm_u (ID almid , T_RALM_U *pk_ralm_u);

パラメータ

ID	almid	Alarm Handler ID	アラームハンドラID
T_RALM_U*	pk_ralm_u	Packet to Refer Alarm Handler Status	アラームハンドラの状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_ralm_u の内容

void*	exinf	Extended Information	拡張情報
RELTIM_U	lfttim_u	Left Time	ハンドラ起動までの残り時間(マイクロ秒)
UINT	almstat	Alarm Handler Status	アラームハンドラの状態

——(以下に実装独自に他の情報を追加してもよい)——

エラーコード

E_OK	正常終了
E_ID	不正ID番号(almid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(almid のアラームハンドラが存在しない)
E_PAR	パラメータエラー(pk_ralm_u が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

tk_ref_alm のリターンパラメータである lfttim を64ビットマイクロ秒単位の lfttim_u としたシステムコールである。

リターンパラメータが lfttim_u となった点を除き、本システムコールの仕様は tk_ref_alm と同じである。詳細は tk_ref_alm の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

4.8 割込み管理機能

割込み管理機能は、外部割込みおよびCPU例外に対するハンドラの定義などの操作を行うための機能である。

割込みハンドラは、タスク独立部として扱われる。タスク独立部でも、タスク部と同じ形式でシステムコールを発行することが可能であるが、タスク独立部で発行できるシステムコールには以下のような制限がつく。

- 暗黙に自タスクを指定するシステムコールや自タスクを待ち状態にするシステムコールは発行することができない。E_CTXのエラーとなる。

タスク独立部の実行中はタスクの切り換え(ディスパッチ)は起らず、システムコールの処理の結果ディスパッチの要求が出されても、タスク独立部を抜けるまでディスパッチが遅らされる。これを遅延ディスパッチ(delayed dispatching)の原則と呼ぶ。

4.8.1 tk_def_int - 割込みハンドラ定義

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_def_int (UINT dintno , CONST T_DINT *pk_dint);

パラメータ

UINT	dintno	Interrupt Handler Number	割込みハンドラ番号
CONST T_DINT*	pk_dint	Packet to Define Interrupt Handler	割込みハンドラ定義情報

pk_dint の内容

ATR	intatr	Interrupt Handler Attribute	割込みハンドラ属性
FP	inthdr	Interrupt Handler Address	割込みハンドラアドレス

――(以下に実装独自に他の情報を追加してもよい)――

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_RSATR	予約属性(intatr が不正あるいは利用できない)
E_PAR	パラメータエラー(dintno, pk_dint, inthdr が不正あるいは利用できない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

割込みには、デバイスからの外部割込みと、CPU例外による割込みの両方を含む。

dintno で示される割込みハンドラ番号に対して割込みハンドラを定義し、割込みハンドラを使用可能にする。すなわち、dintno で示される割込みハンドラ番号と割込みハンドラのアドレスや属性との対応付けを行う。

dintno は、複数の割込みハンドラを区別するための番号である。その具体的な意味は実装ごとに定義されるが、一般には、CPUハードウェアの割込み処理で定義される割込みベクタをそのまま使うか、割込みベクタとの対応付けが可能な何らかの番号を使う。割込みベクタから割込みハンドラ番号 dintno を得るには、T-Kernel/SMの DINTNO() を利用する。

intatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。intatr のシステム属性の部分では、次のような指定を行う。

```
intatr := (TA_ASM || TA_HLNG)
```

TA_ASM	対象ハンドラがアセンブリ言語で書かれている
TA_HLNG	対象ハンドラが高級言語で書かれている

```
#define TA_ASM          0x00000000      /* アセンブリ言語によるプログラム */
#define TA_HLNG         0x00000001      /* 高級言語によるプログラム */
```

TA_ASM 属性の場合、原則として、割込みハンドラの起動時にはカーネルが介入しない。割込み発生時には、CPUハードウェアの割込み処理機能により、このシステムコールで定義した割込みハンドラが直接起動される(実装によってはプログラムによる処理が含まれる場合もある)。したがって、割込みハンドラの先頭と最後では、割込みハンドラで使用するレジスタの退避と復帰を行う必要がある。割込みハンドラは、tk_ret_int システムコールまたはCPUの割込みリターン命令(またはそれに相当する手段)によって終了する。

tk_ret_int を使用せずにカーネルを介すことなく割込みハンドラから復帰する手段は必須である。ただし、tk_ret_int を使用しない場合は、遅延ディスパッチが行われなくてもよい。

tk_ret_int による割込みハンドラからの復帰も必須であり、この場合は遅延ディスパッチが行われなければならない。

TA_HLNG 属性の場合は、高級言語対応ルーチンを経由して割込みハンドラを起動する。高級言語対応ルーチンによって、レジスタの退避と復帰が行われる。割込みハンドラは、C言語関数からのリターンによって終了する。TA_HLNG 属性の場合の割込みハンドラは次の形式となる。

```
void inthdr( UINT dintno )
{
        /*
                割込み処理
        */

        return; /* 割込みハンドラの終了 */
}
```

割込みハンドラに渡される dintno は、発生した割込みの割込みハンドラ番号で、tk_def_int で指定するものと同一である。実装によっては、dintno 以外にも発生した割込みに関する情報が渡される場合がある。このような情報がある場合は、割込みハンドラの2番目以降のパラメータとして実装ごとに定義する。

TA_HLNG 属性の場合、割込み発生から割込みハンドラが呼び出されるまで、CPUの割込みフラグは割込み禁止状態であるものとする。つまり、割込み発生直後から多重割込みが禁止された状態であり、多重割込みが禁止された状態のまま割込みハンドラが呼び出される。多重割込みを許可する場合は、割込みハンドラ内でCPUの割込みフラグを操作して許可する必要がある。

また、TA_HLNG 属性の場合は、割込みハンドラに入った時点で、システムコールの呼出が可能な状態となっていなければならない。ただし、上述の機能を標準とした上で、多重割込みを許可した状態で割込みハンドラに入る機能を追加するなどの拡張は許される。

TA_ASM 属性の場合、割込みハンドラに入った時点の状態は実装ごとに定義される。割込みハンドラに入った時点のスタックやレジスタの状態、システムコールの呼出の可否、システムコールを呼び出せるようにする方法、および割込みハンドラからカーネルを介さずに復帰する方法などが明確に定義されていなければならない。

TA_ASM 属性の場合、実装によっては、割込みハンドラ実行中もタスク独立部として判定されない場合がある。タスク独立部として判定されない場合は、次の点に注意が必要である。

- 割込みを許可すると、タスクディスパッチが発生する可能性がある。
- システムコールを呼び出した場合、タスク部または準タスク部から呼び出したものとして処理される。

なお、割込みハンドラ内で何らかの操作を行ってタスク独立部として判定させる方法がある場合は、実装ごとにその方法を示すものとする。

TA_HLNG, TA_ASM 属性のいずれの場合も、割込みハンドラに入った時点では、割込みが発生した時点の論理空間を維持している。また、割込みハンドラからの復帰時に論理空間を割込み発生時点の状態に戻す処理は行われない。割込みハンドラ内での論理空間の切替は禁止されないが、論理空間を切り替えた場合の影響もカーネルでは関知しない。

割込みハンドラの中でシステムコールを発行することにより、それまで実行状態(RUNNING)であったタスクがその他の状態に移行し、代わりに別のタスクが実行状態(RUNNING)となった場合でも、割込みハンドラ実行中はディスパッチ(実行タスクの切り替え)が起こらない。ディスパッチングが必要になっても、まず割込みハンドラを最後まで実行することが優先され、割込みハンドラを終了する時にはじめてディスパッチが行われる。すなわち、割込みハンドラ実行中に生じたディスパッチ要求はすぐに処理されず、割込みハンドラ終了までディスパッチが遅らされる。これを遅延ディスパッチの原則と呼ぶ。

割込みハンドラはタスク独立部として実行される。したがって、割込みハンドラの中では、待ち状態に入るシステムコールや、自タスクの指定を意味するシステムコールを実行することはできない。

pk_dint＝NULL とした場合には、前に定義した割込みハンドラの定義解除を行う。割込みハンドラの定義解除状態では、システムの定義するデフォルトハンドラが設定される。

既に定義済みの割込みハンドラ番号に対して、割込みハンドラを再定義することも可能である。再定義のときにも、あらかじめその番号のハンドラの定義解除を行っておく必要はない。既に割込みハンドラが定義された dintno に対して新しいハンドラを再定義しても、エラーにはならない。

補足事項

TA_ASM 属性に関する種々の規定は、主に割込みのフックを行うためのものである。例えば、不正アドレスのアクセスによる例外の場合、通常は上位のプログラムで定義した割込みハンドラによって例外を検出してエラー処理を行うが、デバッグ中の場合は上位のプログラムでエラー処理を行う代りに、システムの定義するデフォルトの割込みハンドラに処理させてデバッガを起動するというようなことを行う。この場合、上位のプログラムで定義した割込みハンドラは、システムの定義するデフォルトの割込みハンドラをフックする形になる。そして、状況に応じてデバッガ等のシステムプログラムに割込み処理を引き渡すか、そのまま自身で処理することになる。

4.8.2 tk_ret_int - 割込みハンドラから復帰

C言語インタフェース

```
#include <tk/tkernel.h>
```

void tk_ret_int (void);

C言語インタフェースは形式として定義されるが、高級言語対応ルーチンを使用した場合には呼び出されることがない。

パラメータ

なし

リターンパラメータ

※ システムコールを発行したコンテキストには戻らない

エラーコード

※ 次のようなエラーを検出する可能性があるが、エラーを検出した場合でも、システムコールを発行したコンテキストには戻らない。したがって、システムコールのリターンパラメータとして直接エラーコードを返すことはできない。万一エラーを検出した場合の動作は、実装依存となる。

E_CTX　　コンテキストエラー(割込みハンドラ以外から発行,実装依存)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
×	×	○

解説

割込みハンドラを終了する。

割込みハンドラの中でシステムコールを実行してもディスパッチは起らず、tk_ret_int によって割込みハンドラを終了するまでディスパッチが遅延させられる(遅延ディスパッチの原則)。そのため、tk_ret_int では、割込みハンドラの中から発行されたシステムコールによるディスパッチ要求がまとめて処理される。

tk_ret_int は TA_ASM 属性の割込みハンドラの場合にのみ呼び出す。TA_HLNG 属性の割込みハンドラの場合は、高級言語対応ルーチン内で暗黙に tk_ret_int 相当の機能が実行されるので、tk_ret_int を明示的には呼び出さない(呼び出してはいけない)。

TA_ASM 属性の場合、原則として、割込みハンドラの起動時にはカーネルが介入しない。割込み発生時には、CPUハードウェアの割込み処理機能により、割込みハンドラが直接起動される。そのため、割込みハンドラで使用するレジスタの退避や復帰は割込みハンドラの中で行わなければならない。

また、この理由により、tk_ret_int を発行した時のスタックやレジスタの状態は、割込みハンドラへ入った時と同じでなければならず、この関係で tk_ret_int では機能コードを使用できないことがある。この場合は、他のシステムコールとは別ベクトルのトラップ命令を用いて tk_ret_int を実現する。

補足事項

tk_ret_int は発行元のコンテキストに戻らないシステムコールである。したがって、何らかのエラーを検出した場合にエラーコードを返しても、これらのシステムコールを呼んだ側ではエラーのチェックを行っていないのが普通であり、プログラムが暴走する可能性がある。そこで、これらのシステムコールでは、エラーを検出した場合にも、システムコール発行元へは戻らないものとする。

割込みハンドラから戻っても、ディスパッチが起こらない(必ず同じタスクが実行を継続する)ことがはっきりしている、またはディスパッチしなくても構わない場合には、tk_ret_int ではなく、アセンブリ言語の割込みリターン命令によって割込みハンドラを終了しても構わない。

また、CPUのアーキテクチャやカーネルの構成法によっては、アセンブリ言語の割込みリターン命令によって割込みハンドラを終了しても、遅延ディスパッチが可能となる場合がある。このような場合は、アセンブリ言語の割込みリターン命令がそのまま tk_ret_int のシステムコールであると解釈しても構わない。

タイムイベントハンドラから tk_ret_int を呼んだ場合の E_CTX のエラーチェックは実装依存である。実装によっては、種類の違うハンドラからそのままリターンしてしまう場合がある。

4.9 システム状態管理機能

システム状態管理機能は、システムの状態を変更／参照するための機能である。タスクの優先順位を回転する機能、実行状態のタスクIDを参照する機能、タスクディスパッチを禁止／解除する機能、コンテキストやシステム状態を参照する機能、省電力モードを設定する機能、カーネルのバージョンを参照する機能が含まれる。

4.9.1 tk_rot_rdq - タスクの優先順位の回転

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_rot_rdq (PRI tskpri);

パラメータ

PRI	tskpri	Task Priority	タスク優先度

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(tskpri が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

tskpri で指定される優先度のタスクの優先順位を回転する。すなわち、対象優先度を持った実行できる状態のタスクの中で、最も高い優先順位を持つタスクを、同じ優先度を持つタスクの中で最低の優先順位とする。

tskpri＝TPRI_RUN＝0により、その時実行状態(RUNNING)にあるタスクの優先度のタスクの優先順位を回転する。一般のタスクから発行される tk_rot_rdq では、これは自タスクの持つ優先度のタスクの優先順位を回転するのと同じ意味であるが、周期ハンドラなどのタスク独立部から tk_rot_rdq(tskpri=TPRI_RUN) を発行することも可能である。

補足事項

対象優先度を持った実行できる状態のタスクがない場合や、一つしかない場合には、何もしない(エラーとはしない)。

ディスパッチ許可状態で、対象優先度に TPRI_RUN または自タスクの現在優先度を指定してこのシステムコールが呼び出されると、自タスクの実行順位は同じ優先度を持つタスクの中で最低となる。そのため、このシステムコールを用いて、実行権の放棄を行うことができる。

ディスパッチ禁止状態では、同じ優先度を持つタスクの中で最高の優先順位を持ったタスクが実行されているとは限らないため、この方法で自タスクの実行順位が同じ優先度を持つタスクの中で最低となるとは限らない。

tk_rot_rdq の実行例を[図 4.10.「tk_rot_rdq実行前の優先順位」], [図 4.11.「tk_rot_rdq(tskpri=2)実行後の優先順位」]に示す。[図 4.10.「tk_rot_rdq実行前の優先順位」]の状態で、tskpri＝2をパラメータとしてこのシステムコールが呼ばれると、新しい優先順位は[図 4.11.「tk_rot_rdq(tskpri=2)実行後の優先順位」]のようになり、次に実行されるのはタスクCとなる。

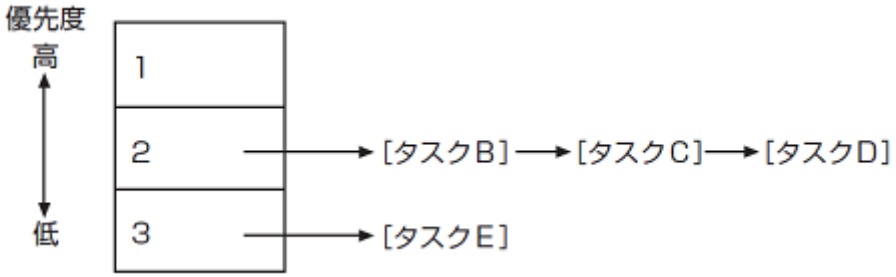

図 4.10: tk_rot_rdq実行前の優先順位

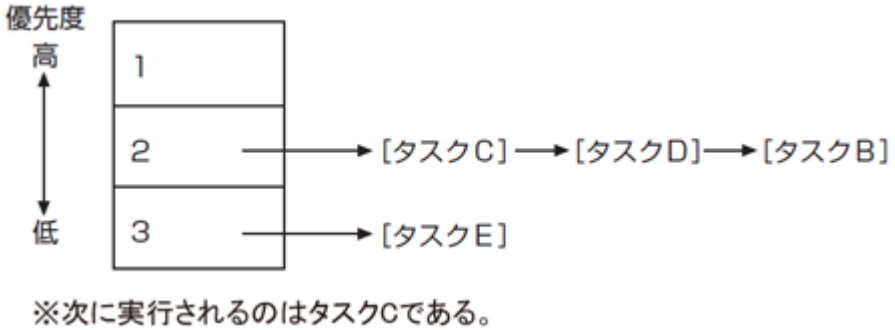

図 4.11: tk_rot_rdq(`tskpri`=2)実行後の優先順位

4.9.2 tk_get_tid - 実行状態タスクのタスクID参照

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID tskid = tk_get_tid (void);

パラメータ

なし

リターンパラメータ

ID	tskid	Task ID	実行状態タスクのID

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

現在実行状態にあるタスクのID番号を得る。タスク独立部の実行中を除けば、現在実行状態にあるタスクは自タスクである。現在実行状態のタスクがない場合は、0が返される。

補足事項

tk_get_tid で返されるタスクIDは、tk_ref_sys で返される runtskid と同一である。

4.9.3 tk_dis_dsp - ディスパッチ禁止

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_dis_dsp (void);

パラメータ

なし

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_CTX	コンテキストエラー(タスク独立部から発行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

タスクのディスパッチを禁止する。これ以後 tk_ena_dsp が実行されるまでの間はディスパッチ禁止状態となり、自タスクが実行状態(RUNNING)から実行可能状態(READY)に移ることはなくなる。また、待ち状態に移ることもできなくなる。ただし、外部割込みは禁止しないので、ディスパッチ禁止状態であっても割込みハンドラは起動される。ディスパッチ禁止状態においては、実行中のタスクが割込みハンドラによってプリエンプト(CPUの実行権の横取り)される可能性はあるが、他のタスクによってプリエンプトされる可能性はない。

ディスパッチ禁止状態の間は、具体的には次のような動作をする。

- 割込みハンドラあるいは tk_dis_dsp を実行したタスクから発行されたシステムコールによって、tk_dis_dsp を実行したタスクより高い優先度を持つタスクが実行可能状態(READY)となっても、そのタスクにはディスパッチされない。優先度の高いタスクへのディスパッチは、ディスパッチ禁止状態が終了するまで遅延される。
- tk_dis_dsp を実行したタスクが、自タスクを待ち状態に移す可能性のあるシステムコール(tk_slp_tsk、tk_wai_sem など)を発行した場合には、E_CTX のエラーとなる。
- tk_ref_sys によってシステム状態を参照した場合、sysstat として TSS_DDSP が返る。

既にディスパッチ禁止状態にあるタスクが tk_dis_dsp を発行した場合は、ディスパッチ禁止状態がそのまま継続するだけで、エラーとはならない。ただし、tk_dis_dsp を何回か発行しても、その後 tk_ena_dsp を1回発行するだけでディスパッチ禁止状態が解除される。したがって、tk_dis_dsp～tk_ena_dsp の対がネストした場合の動作は、必要に応じてユーザ側で管理しなければならない。

補足事項

ディスパッチ禁止状態では、実行状態のタスクが休止状態(DORMANT)や未登録状態(NON-EXISTENT)に移行することはできない。割込みおよびディスパッチ禁止状態で、実行状態のタスクが tk_ext_tsk または tk_exd_tsk を発行した場合には、E_CTX のエラーを検出する。ただし、tk_ext_tsk や tk_exd_tsk は元のコンテキストに戻らないシステムコールなので、これらのシステムコールのリターンパラメータとしてエラーを通知することはできない。

4.9.4 tk_ena_dsp - ディスパッチ許可

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ena_dsp (void);

パラメータ

なし

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_CTX	コンテキストエラー(タスク独立部から発行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

タスクのディスパッチを許可する。すなわち、tk_dis_dsp によって設定されていたディスパッチ禁止状態を解除する。

ディスパッチ禁止状態ではないタスクが tk_ena_dsp を発行した場合は、ディスパッチを許可した状態がそのまま継続するだけで、エラーとはならない。

4.9.5 tk_ref_sys - システム状態参照

C言語インタフェース

```
#include <tk/tkernel.h>

ER ercd = tk_ref_sys (T_RSYS *pk_rsys );
```

パラメータ

T_RSYS*	pk_rsys	Packet to Refer System Status	システム状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rsys の内容

INT	sysstat	System State	システム状態
ID	runtskid	Running Task ID	現在実行状態にあるタスクのID
ID	schedtskid	Scheduled Task ID	実行状態にすべきタスクのID

——(以下に実装独自に他の情報を追加してもよい)——

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(pk_rsys が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

実行状態を参照し、ディスパッチ禁止中、タスク独立部実行中などといった情報をリターンパラメータとして返す。

sysstat は次のような値をとる。

```
sysstat := ( TSS_TSK  | [TSS_DDSP] | [TSS_DINT] )
           || ( TSS_QTSK | [TSS_DDSP] | [TSS_DINT] )
           || ( TSS_INDP )
```

TSS_TSK	0	タスク部実行中
TSS_DDSP	1	ディスパッチ禁止中
TSS_DINT	2	割込み禁止中
TSS_INDP	4	タスク独立部実行中
TSS_QTSK	8	準タスク部実行中

runtskid には現在実行中のタスクのID、schedtskid には実行状態にすべきタスクのIDが返される。通常は runtskid＝schedtskid となるが、ディスパッチ禁止中により優先度の高いタスクが起床された場合などに runtskid≠schedtskid となる場合がある。なお、該当タスクがない場合は0を返す。

割込みハンドラやタイムイベントハンドラからも発行できなければならない。

補足事項

tk_ref_sys で返される情報は、カーネルの実装によっては、必ずしも常に正しい情報を返すとは限らない。

4.9.6 tk_set_pow - 省電力モード設定

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_set_pow (UINT powmode);

パラメータ

UINT	powmode	Power Mode	省電力モード

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(powmode が使用できない値)
E_QOVR	低消費電力モード切替禁止カウントのオーバーフロー
E_OBJ	低消費電力モード切替禁止カウントが0の状態で更に TPW_ENALOWPOW を要求した

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

省電力機能は、次の2つの機能からなる。

● アイドル(無負荷)中の消費電力低減
実行すべきタスクがない状態のとき、ハードウェアに用意されている低消費電力モードに切り替える。

低消費電力モードは、タイマ割込みから次のタイマ割込みの間のようにごく短い時間の電力消費を抑えるための機能で、CPUのクロック周波数を下げることなどにより行われる。したがって、ソフトウェアによる複雑なモード切替を必要とせず、主にハードウェアの機能によって実現される。

● 自動電源オフ
オペレータが何も操作しない状態が一定時間以上続いた場合、自動的に電源を切りサスペンド状態に移行する。周辺機器からの起動要求(割込みなど)またはオペレータが電源を入れることによりリジュームされ、電源が切れたときの状態に復帰する。

また、バッテリー切れなどの電源異常によっても、自動的に電源を切りサスペンド状態に移行する。

サスペンド中は、周辺機器や周辺回路およびCPUの電源が切れる。しかし、メインメモリの内容は保持される。

tk_set_pow は、省電力モードの設定を行う。

```
powmode:= ( TPW_DOSUSPEND || TPW_DISLOWPOW || TPW_ENALOWPOW )
```

```
#define TPW_DOSUSPEND   1       サスペンド状態へ移行
#define TPW_DISLOWPOW   2       低消費電力モード切替禁止
#define TPW_ENALOWPOW   3       低消費電力モード切替許可(デフォルト)
```

- TPW_DOSUSPEND

  すべてのタスク及びハンドラの実行を停止し、周辺回路(タイマや割込みコントローラ)を停止し、電源を切る(サスペンドする)。(off_pow を呼び出す)

  電源オンされたら、周辺回路の再起動をし、すべてのタスク及びハンドラの実行を再開して、電源を切る前の状態に復帰(リジューム)し、リターンする。

  何らかの理由によりリジュームに失敗したときには、通常(リセット時)のスタートアップ処理を行い、新たにシステムを立ち上げなおす。

- TPW_DISLOWPOW

  ディスパッチャ内で行われる低消費電力モードへの切替を禁止する。(low_pow を呼び出さない)

- TPW_ENALOWPOW

  ディスパッチャ内で行われる低消費電力モードへの切替を許可する。(low_pow を呼び出す)

起動時のデフォルトは切替許可(TPW_ENALOWPOW)となる。

TPW_DISLOWPOW が指定された場合、その要求回数がカウントされる。TPW_DISLOWPOW が要求された回数と同じだけ TPW_ENALOWPOW が要求されないと、低消費電力モードは許可されない。要求カウント数の最大値は実装定義だが、少なくとも255回以上カウントできなければならない。

補足事項

off_pow, low_pow は T-Kernel/SMの機能である。詳細は 5.6節「省電力機能」を参照のこと。

T-Kernelでは、電源異常などのサスペンド移行要因の検出は行わない。また、実際にサスペンドするためには、各周辺機器(デバイスドライバ)のサスペンド処理も必要である。サスペンドするには tk_set_pow を直接呼び出すのではなく、T-Kernel/SMのサスペンド機能を利用する。

4.9.7 tk_ref_ver - バージョン参照

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_ver (T_RVER *pk_rver);

パラメータ

T_RVER*	pk_rver	Packet to Refer Version Information	バージョン情報を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rver の内容

UH	maker	Maker Code	カーネルのメーカコード
UH	prid	Product ID	カーネルの識別番号
UH	spver	Specification Version	仕様書バージョン番号
UH	prver	Product Version	カーネルのバージョン番号
UH	prno[4]	Product Number	カーネル製品の管理情報

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(pk_rver が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

使用しているカーネルのバージョン情報を参照し、pk_rverで指定されるパケットに返す。具体的には、次の情報を参照することができる。

maker は、このカーネルを実装したT-Kernel提供者のコードである。maker のフォーマットを[図 4.12.「maker のフォーマット」]に示す。

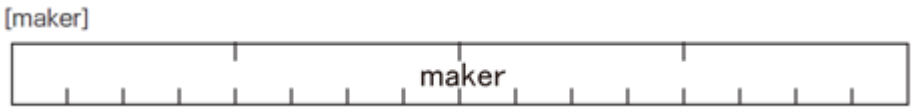

図 4.12: maker のフォーマット

prid は、カーネルの種類を区別するための番号である。prid のフォーマットを[図 4.13.「prid のフォーマット」]に示す。

pridの具体的な値の割付けは、カーネルを実装したT-Kernel提供者に任される。ただし、製品の区別はあくまでもこの番号のみで行うので、各T-Kernel提供者において番号の付け方を十分に検討した上、体系づけて使用するようにしなければならない。したがって、maker と prid の組でカーネルの種類を一意に識別することができる。

T-Kernelのオリジナルは、T-Engineフォーラムから提供され、その maker と prid は次のようになる。

```
maker   = 0x0000
prid    = 0x0000
```

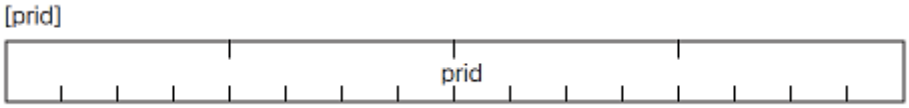

図 4.13: prid のフォーマット

spver では、上位4ビットでOS仕様の種類と、下位12ビットでカーネルが準拠する仕様のバージョン番号を表す。spver のフォーマットを[図 4.14.「spver のフォーマット」]に示す。

たとえばT-KernelのVer 2.01.xxの仕様書に対応する spver は次のようになる。

```
MAGIC   = 0x7           (T-Kernel)
SpecVer = 0x201         (Ver 2.01)
spver   = 0x7201
```

また、T-Kernel仕様書のドラフト版であるVer 2.B0.xxの仕様書に対応する spver は次のようになる。

```
MAGIC   = 0x7           (T-Kernel)
SpecVer = 0x2B0         (Ver 2.B0)
spver   = 0x72B0
```

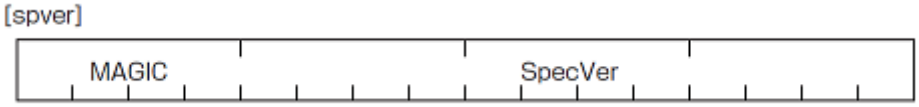

図 4.14: spver のフォーマット

MAGIC:

OS仕様の種類

0x0	TRON共通 (TAD等)
0x1	reserved
0x2	reserved
0x3	reserved
0x4	reserved
0x5	reserved
0x6	reserved
0x7	T-Kernel

SpecVer:
カーネルが準拠する仕様のバージョン番号。3桁のパックト形式BCDコードで入れる。ドラフトの仕様書の場合は、上から2桁目がA, B, Cとなる場合もある。この場合は、対応する16進数のA, B, Cを入れる。

prverは、カーネル実装上のバージョン番号を表す。prver の具体的な値の割付けは、カーネルを実装したT-Kernel提供者に任される。

prno は、カーネル製品の管理情報や製品番号などを入れるために使用するためのリターンパラメータである。prno の具体的な値の意味は、カーネルを実装したT-Kernel提供者に任される。

補足事項

バージョン情報を得るためのパケットの形式や構造体の各メンバのフォーマットは、各種T-Kernelの仕様の間でほぼ共通になっている。

tk_ref_ver の SpecVer として得られるのは仕様書のバージョン番号の上位3桁であるが、これより下位の桁はミスプリントの修正などといった表記上の変更を表す桁なので、tk_ref_ver で得られる情報には含めていない。仕様書の内容との対応という意味では、仕様書のバージョン番号の上位3桁を知ることで必要十分である。

ドラフト版の仕様書を対象として実装されたカーネルでは、SpecVer の2桁目がA, B, Cとなることがある。この場合、仕様書のリリース順序と SpecVer の大小関係が必ずしも一致しないので、注意が必要である。仕様書のリリース順は、たとえばVer 2.A1→Ver 2.A2→Ver 2.B1→Ver 2.C1→Ver 2.00→Ver 2.01→... となるが、SpecVer の大小関係はドラフト版から正式版に移る部分(Ver 2.Cx→Ver 2.00の部分)で逆転する。

4.10 サブシステム管理機能

サブシステム管理機能は、T-Kernel上で動作するミドルウェア等を実装するために、「サブシステム」と呼ばれるユーザ定義の機能をカーネルに追加し、T-Kernel本体の機能を拡張するための機能である。T-Kernel/SMの一部の機能についても、サブシステム管理機能を利用して実装されている。

サブシステムは、ユーザ定義のシステムコール(「拡張SVC」と呼ぶ)を実行するための拡張SVCハンドラのほか、例外発生時の処理を行うブレーク関数、デバイス等からのイベント発生時の処理を行うイベント処理関数、タスクのリソースグループ毎に起動時や終了時の処理を行うためのスタートアップ関数とクリーンアップ関数、およびリソース管理ブロックから構成される[図 4.15.「サブシステム概要」]。

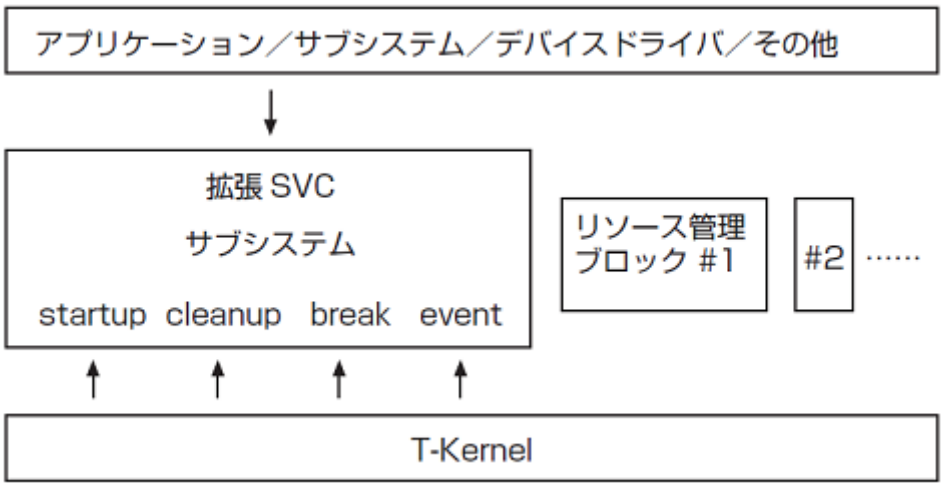

図 4.15: サブシステム概要

このうち、拡張SVCハンドラはアプリケーションなどからの要求を直接受け付けて処理する。一方、ブレーク関数、イベント処理関数、スタートアップ関数、クリーンアップ関数は、いわゆるコールバック型の関数であり、カーネルからの要求を受け付けて処理する。

補足事項
プロセス管理機能、ファイル管理機能などを含むT-Kernel Extension(T-Kernel Standard Extension)の機能も、サブシステム管理機能を利用して実装される。このほか、サブシステム管理機能を利用して実装されたT-Kernel用のミドルウェアの例として、TCP/IPマネージャ、USBマネージャ、PCカードマネージャなどがある。
サブシステム管理機能は、ITRONの拡張SVCハンドラや拡張サービスコールに相当する機能であるが、単なるユーザ定義システムコールの追加だけではなく、追加したシステムコールの処理に付随して必要となるリソース管理の機能や例外発生時の処理機能も提供することによって、高度で複雑な機能を持つミドルウェアを実現可能としたものである。
サブシステム管理機能では、タスクの所属するリソースグループを単位として、リソースを管理する。一方、T-Kernelの上位ミドルウェアであるT-Kernel Extension(T-Kernel Standard Extension)では、T-Kernelのリソースグループの機能を使ってプロセスの機能を実現する。このような関係により、サブシステムのリソース管理をプロセス毎に行うことができ、プロセスの生成時(起動時)や終了時には、その時点で定義されている各サブシステムに含まれるスタートアップ関数とクリーンアップ関数が自動的に実行される。たとえば、プロセスの終了時にクローズされていないファイルを自動的にクローズしたい場合には、ファイル管理のサブシステムに含まれるクリーンアップ関数の中で、このような処理を実行すればよい。
なお、T-Kernel本体の機能を拡張するための機能としては、サブシステム管理機能のほかに、デバイスドライバの機能がある。サブシステムもデバイスドライバも、T-Kernel本体とは独立した機能モジュールであり、対応するバイナリプログラムを共有空間にロードすることによって、T-Kernel上のタスクからこれらの機能を呼び出して利用できるようになる。また、両者とも保護レベル0で動作する。一方、両者の相違点としては、デバイスドライバを呼び出す際のAPIがオープン/クローズ、リード/ライト型に固定されているのに対して、サブシステムを呼び出す際のAPIは自由に定義できる。また、サブシステムの場合はリソースグループ(プロセス)の生成時(起動時)や終了時に自動的にリソースの管理を行う機能があるが、デバイスドライバの場合はこのような機能がない。
サブシステムは、サブシステムID(ssid)によって区別され、複数のサブシステムを同時に定義して利用することが可能である。また、あるサブシステムの中から別のサブシステムを呼び出して利用することも可能である。

4.10.1 tk_def_ssy - サブシステム定義

C言語インタフェース

```
#include <tk/tkernel.h>

ER ercd = tk_def_ssy (ID ssid , CONST T_DSSY *pk_dssy );
```

パラメータ

ID	ssid	Subsystem ID	サブシステムID
CONST T_DSSY*	pk_dssy	Packet to Define Subsystem	サブシステム定義情報

pk_dssy の内容

ATR	ssyatr	Subsystem Attributes	サブシステム属性
PRI	ssypri	Subsystem Priority	サブシステム優先度
FP	svchdr	Extended SVC Handler Address	拡張SVCハンドラアドレス
FP	breakfn	Break Function Address	ブレーク関数アドレス
FP	startupfn	Startup Function Address	スタートアップ関数アドレス
FP	cleanupfn	Cleanup Function Address	クリーンアップ関数アドレス
FP	eventfn	Event Handling Function Address	イベント処理関数アドレス
INT	resblksz	Resource Control Block Size	リソース管理ブロックサイズ(バイト数)

――(以下に実装独自に他の情報を追加してもよい)――

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(ssid が不正あるいは利用できない)
E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_RSATR	予約属性(ssyatr が不正あるいは利用できない)
E_PAR	パラメータエラー(pk_dssy が不正あるいは利用できない)
E_OBJ	ssid はすでに定義済みである。(pk_dssy≠NULL の時)
E_NOEXS	ssid は定義されていない。(pk_dssy＝NULL の時)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ssid のサブシステムを定義する。

1つのサブシステムに1つのサブシステムIDを、他のサブシステムと重複しないように割り当てなければならない。カーネルに自動割当ての機能はない。

サブシステムIDは1～9がT-Kernel用に予約されている。10～255がミドルウェア等で使用できる番号となる。ただし、使用可能なサブシステムIDの最大値は実装定義であり、255より小さい場合がある。

ssyatr は、下位側がシステム属性を表し、上位側が実装独自属性を表す。ssyatr のシステム属性には、現在のバージョンでは割当てがなく、システム属性は使われていない。

ssypri は、サブシステムの優先度で、スタートアップ関数、クリーンアップ関数、イベント処理関数が優先度順に呼び出される。同一優先度の場合の呼出順は不定である。サブシステム優先度は、1が最も優先度が高く、数値が大きくなるにしたがって優先度が下がる。指定できる優先度の範囲は実装定義だが、少なくとも1～16が指定できなければならない。

breakfn, startupfn, cleanupfn, eventfn はそれぞれ NULL を指定することができる。NULL を指定した場合、対応する関数は呼び出されない。

pk_dssy＝NULL とした場合は、サブシステムの定義を抹消する。このとき、ssid のサブシステムのリソース管理ブロックもすべて削除される。

- リソース管理ブロック

  リソース(資源)をグループ分けし、その所属などを管理するためのメモリブロックである。resblksz で指定したサイズのメモリが、リソースグループごとに1つずつ用意される。resblksz＝0を指定すれば、リソース管理ブロックは割り当てられない。しかし、その場合もリソースID(tk_cre_res 参照)は割り当てられる。

  タスクは、いずれかのリソースグループに所属する。あるタスクからサブシステムに対して要求があり、サブシステム内のリソースがそのタスクに割り当てられた場合に、その割当て情報をリソース管理ブロックに登録する。どのようなリソースをリソース管理ブロックに登録するか、またどのようにして登録するかは、サブシステム側で決める。

  リソース管理ブロックの内容についてはカーネルは関知しないので、サブシステム側で自由に利用できる。ただし、resblksz はできる限り小さなサイズとする。より大きなメモリブロックが必要であれば、メモリブロックは別途サブシステム側で用意し、そのアドレスをリソース管理ブロックに登録するようにする。

  リソース管理ブロックは、システム空間の常駐メモリである。

- 拡張SVCハンドラ

  アプリケーションなどからの要求の受け付け口となる。サブシステムのAPIとなる部分である。システムコールと同様の方法で呼び出すことが可能で、一般的にはトラップ命令などで呼び出す。

  拡張SVCハンドラは次の形式となる。

```
INT svchdr( void *pk_para, FN fncd )
{
        /*
                fncd により分岐して処理
        */

        return retcode;  /* 拡張SVCハンドラの終了 */
}
```

  fncd は機能コードである。機能コードの下位8ビットにはサブシステムIDが入る。残りの上位ビットは、サブシステム側で任意に使用できる。通常は、サブシステム内の機能コードとして使用する。ただし、機能コードは正の値でなければならないため、最上位ビットは必ず0となる。

  pk_para は呼出側から渡されたパラメータをパケット形式にしたものである。パケットの形式は、サブシステム側で任意に決めることができる。一般的には、C言語で関数に引数を渡すときのスタックの形式と同じになる。これは、多くの場合C言語の構造体の形式と同じである。

  拡張SVCハンドラからの戻値は、そのまま呼出元へ関数の戻値として戻される。原則として、負の値をエラー値、0または正の値を正常時の戻値とする。なお、何らかの理由で拡張SVCの呼出に失敗した場合は、拡張SVCハンドラは呼び出されずに、呼出元にはカーネルの返すエラーコード(負の値)が返されるため、それと混同しないようにしておくことが望ましい。

拡張SVCの呼出側の形式はカーネルの実装に依存する。ただし、サブシステムのAPIとしてはC言語の関数の形式で、カーネルの実装に依存しないように規定しなければならない。サブシステムは、C言語の関数の形式からカーネル依存の拡張SVCの呼出形式に変換するための、インタフェースライブラリを用意しなければならない。

拡張SVCハンドラは、準タスク部として実行される。

タスク独立部からも呼出が可能で、タスク独立部から呼び出された場合は、拡張SVCハンドラもタスク独立部として実行される。

- ブレーク関数

ブレーク関数は、拡張SVCハンドラの実行中のタスクに、タスク例外が発生した場合に呼び出される関数である。

ブレーク関数が呼び出されたら、タスク例外が発生した現在実行中の拡張SVCハンドラの処理を速やかに中止し、拡張SVCハンドラから呼出元に戻らなければならない。現在実行中の拡張SVCハンドラの処理を中止するための処理をブレーク関数で行う。

ブレーク関数は次の形式となる。

```
void breakfn( ID tskid )
{
        /*
                拡張SVCハンドラの実行中止処理
        */
}
```

tskid はタスク例外が発生したタスクのIDである。

ブレーク関数は、tk_ras_tex によりタスク例外が発生されたときに呼び出される。また、拡張SVCハンドラがネストして呼び出されていた場合は、拡張SVCハンドラから戻ってネストが1段浅くなったときに、戻った先の拡張SVCハンドラに対応するブレーク関数が呼び出される。

ブレーク関数は、1回のタスク例外で、1つの拡張SVCハンドラに対して1回のみ実行される。

タスク例外が発生している状態で、さらにネストして拡張SVCを呼び出した場合、呼出先の拡張SVCハンドラに対してはブレーク関数は呼び出されない。

ブレーク関数は準タスク部として実行されるが、その要求タスクは次のようになる。tk_ras_tex の発行によるブレーク関数実行の場合は、tk_ras_tex を発行したタスクの準タスク部としてブレーク関数が実行される。一方、拡張SVCハンドラのネストが1段浅くなったときのブレーク関数実行の場合は、タスク例外の発生したタスク(拡張SVCハンドラを実行しているタスク)の準タスク部としてブレーク関数が実行される。したがって、ブレーク関数の実行タスクと拡張SVCハンドラの実行タスクが異なっている場合がある。このような場合、ブレーク関数と拡張SVCハンドラがタスクスケジューリングにしたがって同時に実行されることになる。

そのため、ブレーク関数の実行が終了する前に拡張SVCハンドラから呼出元に戻るケースが考えられるが、そのような場合には拡張SVCハンドラから呼出元に戻る直前の状態でブレーク関数が終了するまで待たされる。この待ち状態がタスク状態遷移のどの状態になるかは実装依存とするが、READY状態のまま(RUNNING状態になれないREADY状態)とすることが望ましい。また、ブレーク関数の終了を待つ間、タスクの優先順位が変化することがあるが、タスクの優先順位がどのようになるかは実装依存とする。

同様に、ブレーク関数の実行が終了するまで拡張SVCハンドラから拡張SVCを呼び出すことはできず、ブレーク関数の終了まで待たされる。

つまり、タスク例外が発生してからブレーク関数が終了するまでの間、タスク例外が発生したときの拡張SVCハンドラから抜けないようにしなくてはならない。

ブレーク関数と拡張SVCハンドラの要求タスクが異なるケース、すなわちブレーク関数と拡張SVCハンドラが異なるタスクとして実行されるケースにおいて、ブレーク関数のタスク優先度が拡張SVCハンドラのタスク優先度より低い場合は、ブレーク関数の実行期間だけ、拡張SVCハンドラのタスク優先度と同じ優先度までブレーク関数のタスク優先度が引き上げられる。逆に、ブレーク関数のタスク優先度が同じかより高い場合には、優先度は変更されない。変更される優先度は現在優先度で、ベース優先度は変更されない。

優先度変更はブレーク関数へ入る直前のみで、その後に拡張SVCハンドラ側の優先度が変更されても追従してブレーク関数側の優先度が変更されることはない。ブレーク関数実行中にブレーク関数側の優先度を変更した場合も、拡張SVCハンドラ側の優先度が変更されることはない。また、このときブレーク関数を実行中であることにより優先度の変更が制限されることはない。

ブレーク関数終了時には現在優先度をベース優先度に戻す。ただし、ミューテックスをロックしていた場合には、ミューテックスにより調整された優先度に戻す。(つまり、ブレーク関数の入口と出口で現在優先度を調整する機能があるのみで、それ以外については通常のタスク実行中と同じである。)

- スタートアップ関数

  tk_sta_ssy の呼出によって呼び出される。

  リソース管理ブロックの初期化処理を行う。

  スタートアップ関数は次の形式となる。

```
void startupfn( ID resid, INT info )
{
        /*
                リソース管理ブロックの初期化処理
        */
}
```

  resid は初期化対象のリソースグループID、info は任意のパラメータで、いずれも tk_sta_ssy で指定されたものである。

  何らかの理由でリソース管理ブロックの初期化に失敗した場合も、スタートアップ関数は正常に終了させなければならない。リソース管理ブロックの初期化ができなかった場合、それによって正常に実行できないAPI(拡張SVC)が呼び出された時点で、そのAPIの戻値をエラーとする。

  スタートアップ関数は tk_sta_ssy の発行タスクの準タスク部として実行される。

- クリーンアップ関数

  tk_cln_ssy の呼出によって呼び出される。

  リソースの解放処理を行う。

  クリーンアップ関数は次の形式となる。

```
void cleanupfn( ID resid, INT info )
{
        /*
                リソース解放処理
        */
}
```

  resid はリソース解放の対象となるリソースグループID、info は任意のパラメータで、いずれも tk_cln_ssy で指定されたものである。

  何らかの理由でリソースの解放に失敗した場合でも、クリーンアップ関数は正常に終了させなければならない。エラーログを残すなどのエラー処理は、サブシステムで独自に決める。

  クリーンアップ関数が終了した後、自動的にリソース管理ブロックは0クリアされる。また、クリーンアップ関数が定義されなかった場合(cleanupfn＝NULL)にも、tk_cln_ssy によりリソース管理ブロックは0クリアされる。

  クリーンアップ関数は、tk_cln_ssy の発行タスクの準タスク部として実行される。

- イベント処理関数

  tk_evt_ssy の呼出によって呼び出される。

  サブシステムに対する各種要求を処理する。

  なお、すべての要求がすべてのサブシステムで処理されなければならないという性質のものではない。処理する必要がなければ何もせず単に E_OK を返せばよい。

  イベント処理関数は次の形式となる。

```
ER eventfn( INT evttyp, ID resid, INT info )
{
        /*
                各イベントに対応する処理
        */

        return ercd;
}
```

evttyp は要求の種別、resid は対象となるリソースグループのID、info は任意のパラメータで、いずれも tk_evt_ssy で指定されたものである。特定のリソースグループを対象としない場合は、resid＝0となる。

正常に処理した場合は戻値に E_OK を返す。異常があった場合はエラーコード(負の値)を返す。

evttyp には次のものがある。詳細は 5.3節「デバイス管理機能」を参照のこと。

```
#define TSEVT_SUSPEND_BEGIN     1       /* デバイスサスペンド開始前 */
#define TSEVT_SUSPEND_DONE      2       /* デバイスサスペンド完了後 */
#define TSEVT_RESUME_BEGIN      3       /* デバイスリジューム開始前 */
#define TSEVT_RESUME_DONE       4       /* デバイスリジューム完了後 */
#define TSEVT_DEVICE_REGIST     5       /* デバイス登録通知 */
#define TSEVT_DEVICE_DELETE     6       /* デバイス抹消通知 */
```

イベント処理関数は、tk_evt_ssy の発行タスクの準タスク部として実行される。

補足事項

拡張SVCハンドラおよびブレーク関数／スタートアップ関数／クリーンアップ関数／イベント関数は TA_HLNG 属性相当のみで、高級言語対応ルーチンを経由して呼び出される。TA_ASM 属性を指定する機能はない。

スタートアップ関数によりリソース管理ブロックが初期化される以前、およびクリーンアップ関数によりリソースが解放されたあとに、そのリソースグループに属するタスクから拡張SVCが呼び出された場合の動作は、サブシステムの実装に依存する。カーネルではこのような呼出を特に禁止しない。基本的には、スタートアップ関数の呼出前およびクリーンアップ関数の呼出後に、拡張SVCを呼び出さないようにする必要がある。

何らかの理由で、スタートアップ関数が呼び出されないまま、ブレーク関数／クリーンアップ関数／イベント処理関数が呼び出される場合がある。このような場合も、誤動作せずに実行されなければならない。リソース管理ブロックは、リソース管理ブロックが最初に生成された時および tk_cln_ssy によるクリーンアップ処理時に0クリアされる。したがって、スタートアップ関数により正常に初期化されていなければ、リソース管理ブロックは0クリアされているはずである。

拡張SVCハンドラ内での固有空間は、呼出側の固有空間と同じである。したがって、呼出側の渡したバッファにアクセスする際にも、固有空間を切り替える必要はない。ただし、拡張SVCハンドラは保護レベル0(特権モード)で実行されるので、呼出側のタスク等からは本来アクセスできないメモリ等にもアクセスが可能である。このため、拡張SVCハンドラの中では、必要に応じて、ChkSpaceR(), ChkSpaceRW() などを使ったアクセス権チェックを行わなければならない。

拡張SVCハンドラの中で待ち状態に入るシステムコールを発行しても構わないが、ブレーク関数による中止を考慮したプログラムにしておく必要がある。この場合の具体的な処理は次のようになる。拡張SVCハンドラの実行中に、呼出側のタスクを対象とした tk_ras_tex が発行された場合、できるだけ速やかに拡張SVCハンドラの実行中止処理を行い、呼出側に対して中止のエラーを返す必要がある。このためにブレーク関数が実行される。ブレーク関数の中では、速やかに実行中止処理を行うため、拡張SVCハンドラの処理中に待ち状態になっていた場合でも、その待ち状態を強制的に解除する必要がある。このためのシステムコールとして、通常は tk_dis_wai を使う。tk_dis_wai の機能により、その後も拡張SVCハンドラから呼出側に戻るまでの間は待ち状態に入らないようにできるが、拡張SVCハンドラ側のプログラムも、ブレーク関数による実行の中止を考慮しておくべきである。たとえば、待ち状態から E_DISWAI のエラーで抜けた場合には、ブレーク関数による実行の中止を意味すると考えられるので、その後に予定していた処理を行わず、速やかに拡張SVCハンドラを終了し、呼出側に対して中止のエラーを返すようにする。

拡張SVCハンドラは、複数のタスクから同時に並行して呼び出される可能性がある。そのため、共通に利用する資源等があった場合には、拡張SVCハンドラの中で、必要に応じて排他制御を行う。

4.10.2 tk_sta_ssy - スタートアップ関数呼出

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_sta_ssy (ID ssid , ID resid , INT info);

パラメータ

ID	ssid	Subsystem ID	サブシステムID
ID	resid	Resource ID	リソースID
INT	info	Information	任意パラメータ

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(ssid, resid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(ssid のサブシステムが定義されていない)
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ssid のサブシステムのスタートアップ関数を呼び出す。

ssid＝0を指定したときは、現在定義されているすべてのサブシステムのスタートアップ関数を呼び出す。この場合、サブシステム優先度の高いものから順に各サブシステムのスタートアップ関数を呼び出す。

同一優先度の場合の呼出順は不定である。

したがって、複数のサブシステム間に依存関係がある場合、サブシステム優先度をその依存関係にしたがって設定する必要がある。例えば、サブシステムBがサブシステムAの機能を利用しているような場合、サブシステムAの優先度をサブシステムBより高くしなければならない。

スタートアップ関数が定義されていないサブシステムに対して実行しても、単にスタートアップ関数を呼び出さないだけでエラーとはならない。

スタートアップ関数を実行中に、tk_sta_ssy を呼び出したタスクにタスク例外が発生した場合、タスク例外はスタートアップ関数が終了するまで保留される。

補足事項

T-Kernelの上位ミドルウェアであるT-Kernel Extension(T-Kernel Standard Extension)では、プロセス生成時(起動時)のスタートアップ処理とプロセス終了時のクリーンアップ処理を行うために、tk_sta_ssy および tk_cln_ssy を利用する。具体

的には、T-Kernel Extensionの中のプロセスの生成時(起動時)の処理において、ssid＝0を指定した tk_sta_ssy を発行し、新しく起動されたプロセスに対するスタートアップ処理を行う。また、T-Kernel Extensionの中のプロセス終了時の処理において、ssid＝0を指定した tk_cln_ssy を発行し、終了するプロセスに対するクリーンアップ処理を行う。たとえば、ファイル管理のサブシステムがこの機能を利用することにより、プロセス終了時に実行されるファイル管理サブシステムのクリーンアップ処理の中で、そのプロセスがオープンしていたファイルを自動的にクローズすることができる。

複数のサブシステムが定義されていた場合は、各サブシステムに対するスタートアップ関数やクリーンアップ関数が順に実行されるが、その実行順序はサブシステム優先度によって決められ、スタートアップ処理とクリーンアップ処理では逆の実行順序になる。

たとえば、サブシステムBの実装のために別のサブシステムAを使っているケースでは、サブシステムAの優先度をサブシステムBの優先度よりも高くしておく。そうすると、新しく起動するプロセスに対して、サブシステムAのスタートアップ処理がサブシステムBのスタートアップ処理よりも先に実行される。したがって、サブシステムBのスタートアップ処理の中で、サブシステムAの機能(拡張SVCハンドラ)を呼び出すことができる。また、終了するプロセスに対しては、サブシステムBのクリーンアップ処理がサブシステムAのクリーンアップ処理よりも先に実行される。したがって、サブシステムBのクリーンアップ処理の中で、サブシステムAの機能(拡張SVCハンドラ)を呼び出すことができる[図 4.16.「サブシステムの依存関係と優先度」]。

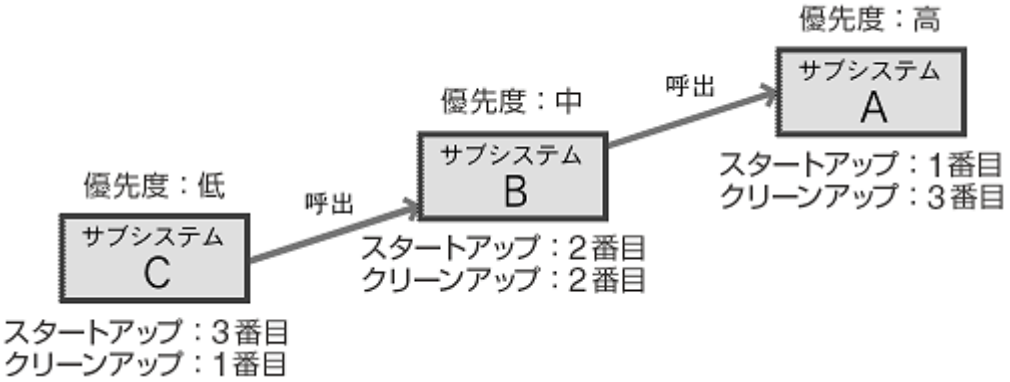

図 4.16: サブシステムの依存関係と優先度

すべてのサブシステムのスタートアップ関数は、新しいプロセスが生成(起動)される度に必ず実行される。しかし、起動されたプロセスがそのサブシステムの機能を利用するとは限らず、一度もそのサブシステムを呼び出さないかもしれない。サブシステムとは無関係のプロセスも含めて、プロセスの生成時(起動時)に各サブシステムのスタートアップ関数がすべて実行されることを考えると、スタートアップ関数によるオーバーヘッドは最小限としておくべきである。すなわち、スタートアップ関数の中では必要最小限の処理のみを行うようにし、複雑な処理を行う必要があれば、スタートアップ関数の中ではなく、実際にサブシステムが利用された時、たとえば最初にそのプロセスから拡張SVCハンドラが呼ばれた時までその処理を遅らせる方がよい。

4.10.3 tk_cln_ssy - クリーンアップ関数呼出

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_cln_ssy (ID ssid , ID resid , INT info);

パラメータ

ID	ssid	Subsystem ID	サブシステムID
ID	resid	Resource ID	リソースID
INT	info	Information	任意パラメータ

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(ssid, resid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(ssid のサブシステムが定義されていない)
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ssid のサブシステムのクリーンアップ関数を呼び出す。

ssid＝0を指定したときは、現在定義されているすべてのサブシステムのクリーンアップ関数を呼び出す。この場合、サブシステム優先度の低いものから順に各サブシステムのクリーンアップ関数を呼び出す。

同一優先度の場合の呼出順は不定である。

したがって、複数のサブシステム間に依存関係がある場合、サブシステム優先度をその依存関係にしたがって設定する必要がある。例えば、サブシステムBがサブシステムAの機能を利用しているような場合、サブシステムAの優先度をサブシステムBより高くしなければならない。

クリーンアップ関数が定義されていないサブシステムに対して実行しても、単にクリーンアップ関数を呼び出さないだけでエラーとはならない。

クリーンアップ関数を実行中に、tk_cln_ssy を呼び出したタスクにタスク例外が発生した場合、タスク例外はクリーンアップ関数が終了するまで保留される。

4.10.4 tk_evt_ssy - イベント処理関数呼出

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_evt_ssy (ID ssid , INT evttyp , ID resid , INT info);

パラメータ

ID	ssid	Subsystem ID	サブシステムID
INT	evttyp	Event Type	イベント要求種別
ID	resid	Resource ID	リソースID
INT	info	Information	任意パラメータ

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(ssid, resid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(ssid のサブシステムが定義されていない)
E_CTX	コンテキストエラー(タスク独立部またはディスパッチ禁止状態で実行)
その他	イベント処理関数の返したエラー値

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ssid のサブシステムのイベント処理関数を呼び出す。

ssid＝0を指定したときは、現在定義されているすべてのサブシステムのイベント処理関数を呼び出す。この場合、次の順序で各サブシステムのイベント処理関数を呼び出す。

evttyp が奇数のとき:
　サブシステム優先度の高い方から順に呼び出す。

evttyp が偶数のとき:
　サブシステム優先度の低い方から順に呼び出す。

同一優先度の場合の呼出順は不定である。

イベント処理関数が定義されていないサブシステムに対して実行しても、単にイベント処理関数を呼び出さないだけでエラーとはならない。

特定のリソースに対する処理要求でない場合は、resid＝0とする。

イベント処理関数がエラーを返した場合、システムコールの戻値としてそのエラー値をそのまま返す。ssid＝0の場合、イベント処理関数がエラーを返した場合も、すべてのサブシステムのイベント処理関数が呼び出される。システムコールの戻値には、エラーを返したイベント処理関数の内の1つのエラー値のみ返す。どのサブシステムのイベント処理関数が返したエラーかはわからない。

イベント処理関数を実行中に、tk_evt_ssy を呼び出したタスクにタスク例外が発生した場合、タスク例外はイベント処理関数が終了するまで保留される。

補足事項

イベント処理関数の利用例の1つは、省電力機能に関連したサスペンドやリジュームの処理である。具体的には、電源異常などの要因により電源を切った状態(デバイスのサスペンド状態)に移行する際、サスペンド状態への移行を各サブシステムに対して通知し、サブシステム毎に適切な処理を行うために、各サブシステムのイベント処理関数が呼び出される。T-Kernel/SMの tk_sus_dev の処理の中では、この目的で tk_evt_ssy を実行している。各サブシステムのイベント処理関数では、必要に応じて、サスペンド状態への移行時に処理すべきデータの保存などを行う。一方、電源の再投入などによってサスペンド状態から復帰(リジューム)する際には、サスペンド状態からの復帰を各サブシステムに対して通知し、サブシステム毎に適切な処理を行うために、各サブシステムのイベント処理関数が再度呼び出される。詳細は tk_sus_dev を参照。

このほか、tk_def_dev によって新しいデバイスが登録された際にも、デバイスの登録を各サブシステムに対して通知し、サブシステム毎に適切な処理を行うために、各サブシステムのイベント処理関数が呼び出される。T-Kernel/SMの tk_def_dev の処理の中では、この目的で tk_evt_ssy を実行している。

4.10.5 tk_ref_ssy - サブシステム定義情報の参照

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_ssy (ID ssid , T_RSSY *pk_rssy);

パラメータ

ID	ssid	Subsystem ID	サブシステムID
T_RSSY*	pk_rssy	Packet to Refer Subsystem	サブシステム定義情報を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rssy の内容

PRI	ssypri	Subsystem Priority	サブシステム優先度
INT	resblksz	Resource Control Block Size	リソース管理ブロックサイズ(バイト数)

――(以下に実装独自に他の情報を追加してもよい)――

エラーコード

E_OK	正常終了
E_ID	不正ID番号(ssid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(ssid のサブシステムが定義されていない)
E_PAR	パラメータエラー(pk_rssy が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ssid で示された対象サブシステムの各種の情報を参照する。

ssypri には、tk_def_ssy で指定したサブシステムの優先度が返される。

resblksz には、tk_def_ssy で指定したリソース管理ブロックサイズが返される。

ssid のサブシステムが定義されていない場合は、E_NOEXS となる。

4.10.6 tk_cre_res - リソースグループの生成

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_cre_res (void);

パラメータ

なし

リターンパラメータ

ID	resid		Resource ID	リソースID
		または	Error Code	エラーコード

エラーコード

E_LIMIT	リソースグループの数がシステムの制限を超えた
E_NOMEM	メモリ不足(管理ブロックの領域が確保できない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

新規のリソースグループを生成し、リソース管理ブロックとリソースIDを割り当てる。

リソースIDは全サブシステムに対して共通で割り当てられ、リソース管理ブロックが各サブシステムごとに生成される[図 4.17.「サブシステムとリソースグループの関係」]。

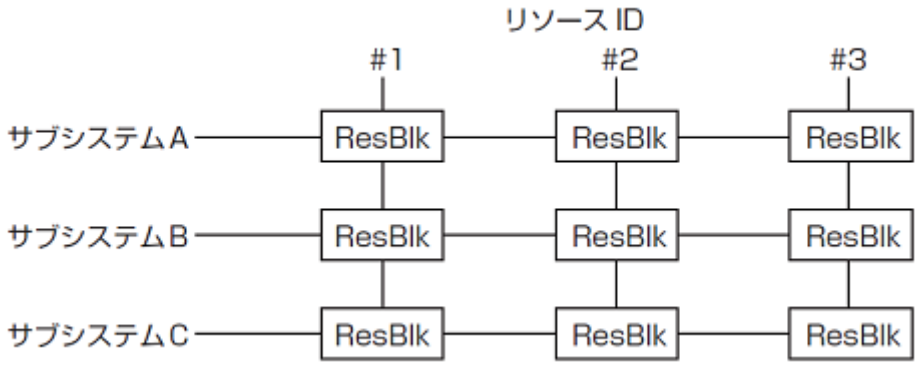

図 4.17: サブシステムとリソースグループの関係

すでにリソースグループが生成されている状態で、新たにサブシステムが定義される場合がある。この場合も、新たに登録されたサブシステムに対して、すでに存在するリソースグループのリソース管理ブロックが生成されなければならない。つまり、リソース管理ブロックの生成は tk_def_ssy でも行わなければならないケースがありうる。

例えば、[図 4.17.「サブシステムとリソースグループの関係」]のような状態のとき新たにサブシステムDが定義された場合、自動的にサブシステムDのためのリソースID #1,#2,#3のリソース管理ブロックが生成されなければならない。

補足事項

リソースIDは論理空間ID(`lsid`)としても使用される場合がある。そのため、リソースIDは論理空間IDとして直接利用可能な値、もしくは簡単な変換で論理空間IDとして利用可能な値とすることが望ましい。

特殊なリソースグループとして、システムリソースグループが必ず存在する。システムリソースグループは、tk_cre_res で生成するまでもなくシステム起動時から必ず1つ存在しているリソースグループである。システムリソースグループは削除することはできない。必ず存在していることを除けば、システムリソースグループが他のリソースグループと異なる部分はない。

リソース管理ブロックの生成は、次の2種類の実装が考えられる。

- (A) サブシステム定義時(tk_def_ssy)に最大リソースグループ数分のリソース管理ブロックを生成しておき、tk_cre_res では単にそれを割り当てる。
- (B) tk_cre_res で全サブシステム分のリソース管理ブロックを生成して、それを割り当てる。

最初にリソース管理ブロックを生成した時に、リソース管理ブロックを0クリアするという仕様としているため、(A)と(B)ではリソース管理ブロックの0クリアのタイミングが異なる。これによる影響は少ないと考えられるが、(A)の方が0クリアされるケースが少なくなるため、サブシステムは(A)を前提として実装しなければならない。なお、カーネルの実装としても(A)を推奨する。

T-Kernelの上位ミドルウェアであるT-Kernel Extension(T-Kernel Standard Extension)では、T-Kernelのリソースグループの機能を使ってプロセスの機能を実現しており、1つのプロセスが1つのリソースグループに対応する。このため、プロセスの生成時(起動時)には、tk_cre_res を実行して、プロセス毎のリソース管理ブロックを確保する。

リソース管理の機能により、プロセス単位(リソースグループ単位)で独立したリソースを各サブシステムに持たせたり、プロセス終了時に自動的にリソースを解放するなどの機能が実現できる。たとえば、ファイルを管理するサブシステムでは、プロセスがファイルをオープンする度に、ファイルディスクリプタと呼ばれるファイル操作用の識別子が割り当てられ、以後のファイル操作には、ファイルディスクリプタを用いる場合が多い。この場合、ファイルディスクリプタによって識別される、ファイル操作用の各種の管理情報がリソースとなる。このリソースを、ファイル管理サブシステムに対するリソース管理ブロック内に置くことにより、ファイル操作用の各種の情報をプロセス単位(リソースグループ単位)で独立に管理することができる。

一般に、プロセス毎に独立して管理すべき機能を実現するサブシステムでは、サブシステムに対する各プロセスの情報を独立して管理するために、リソース管理ブロックの利用が有効である。また、新しく生成(起動)されたプロセスに対するサブシステム側の準備やリソース管理ブロックの初期化のために、サブシステムのスタートアップ関数を利用したり、プロセス終了時の自動的なリソース解放のために、サブシステムのクリーンアップ関数を利用できる。一方、プロセスとの直接的な関連を持たない機能(どのプロセスからも区別されずに共有される機能、システム全体にかかわる機能など)を実現するサブシステムの場合は、リソース管理ブロックやリソース、リソースグループに関する機能を使う機会は少ない。

新しいプロセスを生成(起動)した際には、そのプロセスが各サブシステムの機能を実際に使うかどうかに関わらず、システム共有空間の常駐メモリ領域の中に、各サブシステムに対するリソース管理ブロックが確保される。すなわち、システム共通の資源を消費することになる。したがって、システム全体に対するオーバーヘッドを減らすには、リソース管理ブロックのサイズを最小限としておくのが望ましい。

たとえば、あるサブシステムにおいて、プロセス毎に独立した作業用メモリが1MBずつ必要であるとする。プロセス毎に必要であるという点から、リソース管理ブロックの一部を作業用メモリとして利用する方法が考えられるが、この容量はリソース管理ブロックとしては大きすぎる。リソース管理ブロックのサイズを1MBとした場合、新しいプロセスを生成(起動)する度に、これだけの容量のメモリが無条件に確保されることになり、システム共有空間の常駐メモリを消費し過ぎるからである。特に、新しいプロセスがこのサブシステムの機能を全く使わない場合には、メモリの無駄が大きい。

このようなケースでは、サブシステムの使う作業用メモリの確保を、実際の必要時まで遅らせる方がよい。たとえば、作業用メモリが確保できているかどうかのフラグと、作業用メモリのアドレスのみをリソース管理ブロックに入れておき、プロセスがサブシステムの機能を使った場合(拡張SVCハンドラを呼び出した場合)には、そのフラグをチェックして、作業用メモリがまだ確保されていない場合にのみメモリの確保を行うようにする。このような処理を行えば、サブシステムとは関係のないプロセスにも大きなリソース管理ブロックを割り当てるといった無駄がなくなる。

4.10.7 tk_del_res - リソースグループの削除

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_del_res (ID resid);

パラメータ

ID	resid	Resource ID	リソースID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(resid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(resid のリソースが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

resid のリソースグループのリソース管理ブロックを削除し、リソースIDを解放する。

全サブシステムのリソース管理ブロックが削除される。

補足事項

削除対象のリソースに属しているタスクが残っていても、リソースは削除される。原則として、削除対象のリソースに属しているすべてのタスクを終了・削除した後に、リソースを削除しなければならない。削除対象リソースに属しているタスクが残っている状態で、かつそのタスクがサブシステム(拡張SVC)を呼び出している状態で、リソースを削除した場合の動作は保証されない。また、削除されたリソースに属するタスクが、サブシステム(拡張SVC)を呼び出した場合の動作も保証されない。

リソース管理ブロックが実際に削除されるタイミングは実装に依存する。(tk_cre_res 参照)

システムリソースグループの削除はできない。(E_ID)

4.10.8 tk_get_res - リソース管理ブロックの取得

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_res (ID resid , ID ssid , void **p_resblk);

パラメータ

ID	resid	Resource ID	リソースID
ID	ssid	Subsystem ID	サブシステムID
void**	p_resblk	Resource Control Block	リターンパラメータ resblk を返す領域へのポインタ

リターンパラメータ

void*	resblk	Resource Control Block	リソース管理ブロック
ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(resid, ssid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(resid, ssid のリソースが存在しない)
E_PAR	パラメータエラー(p_resblk が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ssid のサブシステムの resid のリソース管理ブロックのアドレスを取得する。

補足事項

削除されたリソースIDに対しても E_OK となる場合がある。エラー(E_NOEXS)が返されるか否かは実装依存である。

第 5 章

T-Kernel/SMの機能

この章では、T-Kernel/SM(System Manager)で提供している機能の詳細について説明を行う。

全般的な注意・補足事項

- T-Kernel/SMの仕様で定義されているAPIには、tk_～の名称を持つものと、それ以外の名称を持つものがある。原則として、tk_～の名称を持つAPIは拡張SVCで実装され、それ以外の名称を持つAPIはライブラリ関数(インライン関数を含む)またはC言語のマクロで実装されている。なお、T-Kernel/SMで定義されるAPIに対しては、「システムコール」の名称を用いておらず、「システムコール」と呼んだ場合には、T-Kernel/OSやT-Kernel/DSで定義されているAPIを指す。
- ライブラリまたはマクロであっても、間接的に拡張SVCやシステムコールを呼び出している場合がある。
- 常に発生する可能性のあるエラー E_PAR, E_MACV, E_NOMEM などは、特に説明を必要とする場合以外は省略している。
- T-Kernel/SMの拡張SVCおよびライブラリは、特に明記されているものを除き、タスク独立部およびディスパッチ禁止中・割込み禁止中状態から呼び出すことはできない(E_CTX)。
- T-Kernel/SMの拡張SVCおよびライブラリは、特に明記されているものを除き、T-Kernel/OSのシステムコールの呼出可能な保護レベルより低い保護レベル(TSVCLimit より低い保護レベル)から呼び出すことはできない(E_OACV)。
- T-Kernel/SMの拡張SVCおよびライブラリは、特に明記されているものを除き、再入可能(reentrant)である。ただし、内部で排他制御を行っている場合がある。

5.1 システムメモリ管理機能

システムメモリ管理機能は、T-Kernelが動的に割り当てるすべてのメモリ(システムメモリ)を管理するための機能である。T-Kernel内部で使用しているメモリやタスクのスタック、メッセージバッファ、メモリプールなどもここから割り当てる。

システムメモリは、ブロック単位で管理される。ブロックサイズは、通常MMUで定義されるページサイズであり、現在の実装では4KB程度を想定している。MMUを使用しないシステムでは任意のサイズでよいが、MMUのページサイズと同程度を推奨する。ブロックサイズは、tk_ref_smb により取得できる。

システムメモリは、共有空間のメモリである。T-Kernelではタスク固有空間のメモリの管理は行わない。

システムメモリ管理機能は、システムメモリからのメモリの割当てや解放をブロック単位で行うシステムメモリ操作用の拡張SVCと、ブロック単位で確保されたシステムメモリをさらに細分化して管理するメモリ割当てライブラリからなる。

システムメモリ管理機能は、T-Kernel内部で利用する他、アプリケーションやサブシステム、デバイスドライバなどからも利用可能である。なお、T-Kernel内部での利用は拡張SVCを経由しない方法でもよく、実装定義とする。

5.1.1 システムメモリ割当て

システムメモリ割当ての機能では、システムメモリからのメモリの割当て、解放と、システムメモリ情報を参照する機能を拡張SVCで提供する。

5.1.1.1 tk_get_smb - システムメモリの割当て

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_get_smb (void **addr , INT nblk , UINT attr);

パラメータ

void**	addr	Pointer to Memory Start Address	割り当てたメモリの先頭アドレスを返す領域へのポインタ
INT	nblk	Number of Blocks	割り当てるメモリブロック数
UINT	attr	Attribute	割り当てるメモリの属性

リターンパラメータ

ER	ercd	Error Code	エラーコード
void*	addr	Memory Start Address	割り当てたメモリの先頭アドレス

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー((nblk≦0)または attr が不正)
E_NOMEM	メモリ不足(システムメモリが不足している)
E_MACV	メモリアクセス権違反(addr に書込みができない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

nblk で指定したブロック数分の連続したメモリ領域を attr で指定した属性で割り当てる。割り当てたメモリの先頭アドレスを addr に返す。

attr では、以下の属性を指定する。

```
attr := (TA_RNG0 || TA_RNG1 || TA_RNG2 || TA_RNG3) | [TA_NORESIDENT]
```

TA_RNG0	保護レベル0のメモリを指定する
TA_RNG1	保護レベル1のメモリを指定する
TA_RNG2	保護レベル2のメモリを指定する
TA_RNG3	保護レベル3のメモリを指定する
TA_NORESIDENT	非常駐メモリを指定する

TA_RNGn では、メモリのアクセスを制限する保護レベルを指定する。割り当てられたメモリは、指定された保護レベルと同じかより高い保護レベルで実行しているタスク等からのみアクセス可能である。

TA_NORESIDENT が指定された場合、割り当てられたメモリは非常駐メモリとなる。MMUのないシステムでは、非常駐メモリの

属性を指定した場合でも、実際には常駐メモリと同様の動作をすることになるが、エラーとはしない。

nblk に負の値が指定された場合や、attr に指定できない属性が指定された場合には、E_PAR のエラーを返す。また、addr の指すメモリ(割り当てたメモリの先頭アドレスを返す領域)への書込みアクセスが禁止されている場合には、E_MACV のエラーを返す。

nblk で指定したブロック数分の連続したメモリ領域の割当てができない場合、E_NOMEM のエラーを返す。この場合、addr の指すメモリには NULL が返される。

補足事項

MMUを使用しないシステムでは、実装上、メモリの保護レベルに違反したアクセスがあった場合にも、アクセス権違反の例外を検出することができず、正常なメモリアクセスができてしまう。しかし、プログラムの移植性や拡張性を考慮し、割り当てるメモリに対しては、アクセスするタスク等の保護レベルからアクセス可能な適切な保護レベルを指定することが推奨される。

5.1.1.2 tk_rel_smb - システムメモリの解放

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_rel_smb (void *addr);

パラメータ

void*	addr	Memory Start Address	解放するメモリの先頭アドレス

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(addr が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

addr で指定したメモリを解放する。addr は、tk_get_smb() で得たアドレスでなければならない。

addr に指定されたアドレスが不正であることが検出された場合は、E_PAR のエラーが返る。具体的には、addr がT-Kernel が管理しているメモリの範囲外を指している場合や、既に tk_rel_smb() で解放したメモリを再度解放しようとした場合に、E_PAR エラーが返る。ただし、実装上の制約により、addr が不正であってもエラーの検出ができない場合があり、その場合の動作は保証されない。そのため、addr の正当性は呼出側で保証しなければならない。

5.1.1.3 tk_ref_smb - システムメモリ情報取得

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_smb (T_RSMB *pk_rsmb);

パラメータ

T_RSMB*	pk_rsmb	Packet to Refer System Memory Block	システムメモリ情報を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rsmb の内容

INT	blksz	Block Size	ブロックサイズ(バイト数)
INT	total	Total Block Count	全ブロック数
INT	free	Free Block Count	残りブロック数

――(以下に実装独自に他の情報を追加してもよい)――

エラーコード

E_OK	正常終了
E_MACV	メモリアクセス権違反(pk_rsmb に書込みができない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

システムメモリに関する情報を取得する。

仮想記憶を行っているシステムでは、非常駐のメモリを割り当てることにより、物理的なメモリよりも大きなサイズのメモリを使うことができる。このため、全ブロック数や残りブロック数が一意に決定できないことがある。そのような場合の total, free の内容は実装依存とするが、free÷total が残りメモリ容量の割合の参考値となるような値とすることが望ましい。

5.1.2 メモリ割当てライブラリ

tk_get_smb() によるシステムメモリの割当てはブロック単位であるため、より細分化して効率的にメモリを使用するために、メモリ割当てライブラリを使用する。

メモリ割当てライブラリの内部では、tk_get_smb() によって確保されたシステムメモリを管理しており、その中からアプリケーションの要求したサイズのメモリを割り当てる。メモリ割当てライブラリが管理するメモリに、アプリケーションが要求するサイズの空きメモリが無い場合には、再度 tk_get_smb() を呼び出して追加のシステムメモリを割り当てる。

一方、アプリケーションからメモリが返却された際に、そのメモリが含まれるメモリブロックのすべてが未割当てとなった場合には、tk_rel_smb() によってそのメモリブロックを解放する。ただし、メモリブロックの確保や解放の厳密なタイミングなどは実装依存である。

メモリ割当てライブラリは、C言語標準ライブラリの `malloc`／`calloc`／`realloc`／`free` と同等の機能が提供される。対象とするメモリが非常駐メモリの場合は V～ のAPI名、常駐メモリの場合は K～ のAPI名である。

これらのメモリは、すべて `TSVCLimit` で指定された保護レベルのメモリとして割り当てられる。

5.1.2.1 Vmalloc - 非常駐メモリの割当て

C言語インタフェース

```
#include <tk/tkernel.h>
```

void* Vmalloc (size_t size);

パラメータ

size_t	size	Size	割り当てるメモリサイズ(バイト数)

リターンパラメータ

void*	addr	Memory Start Address	割り当てたメモリの先頭アドレス

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

size で指定したバイト数の非常駐メモリを割り当て、その先頭アドレスを addr に返す。

指定したサイズのメモリの割当てができなかった場合や、size に0が指定された場合は、addr に NULL が返る。

Vmalloc を含むメモリ割当てライブラリのAPIは、タスク独立部およびディスパッチ禁止中、割込み禁止中に呼び出すことはできない。呼び出した場合の動作は、システムダウンの可能性も含めて不定であり、呼出時の状態を保証するのは、呼出側の責任である。

補足事項

size には任意の値を指定できるが、管理領域の確保や、割り当てるメモリアドレスのアラインメントの調整といった理由により、内部的には、size で指定したバイト数よりも大きなメモリが割り当てられる場合がある。たとえば、割当て可能なメモリサイズの最低が16バイトで、アラインメントが8バイト単位という実装の場合には、size に16バイト未満の値を指定した場合でも、内部的には16バイトのメモリが割り当てられる。また、size に20バイトの値を指定した場合でも、内部的には24バイトのメモリが割り当てられる。

したがって、メモリ割当てライブラリ全体で使用するシステムメモリのサイズと、メモリ割当てライブラリの各APIで割り当てられたメモリのサイズの合計を比較すると、前者の方が大きな値をとる場合がある。

5.1.2.2 Vcalloc - 非常駐メモリの割当て

C言語インタフェース

```
#include <tk/tkernel.h>
```

void* Vcalloc (size_t nmemb , size_t size);

パラメータ

size_t	nmemb	Number of Memory Blocks	割り当てるメモリブロックの個数
size_t	size	Size	割り当てるメモリブロックのサイズ(バイト数)

リターンパラメータ

void*	addr	Memory Start Address	割り当てたメモリの先頭アドレス

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

size で指定したバイト数のメモリブロックを、nmemb で指定された個数だけ連続して割り当て、0クリアしてから、その先頭アドレスを addr に返す。メモリ割当ての動作は、size と nmemb を乗じたバイト数のメモリブロック1個を割り当てるのと同じである。割り当てられるメモリは、非常駐メモリである。

指定した個数のメモリブロックの割当てができなかった場合や、nmemb または size に0が指定された場合は、addr に NULL が返る。

Vcalloc を含むメモリ割当てライブラリのAPIは、タスク独立部およびディスパッチ禁止中、割込み禁止中に呼び出すことはできない。呼び出した場合の動作は、システムダウンの可能性も含めて不定であり、呼出時の状態を保証するのは、呼出側の責任である。

補足事項

内部的には、size と nmemb を乗じたバイト数よりも大きなメモリが割り当てられる場合がある。詳細は Vmalloc() の補足事項を参照のこと。

5.1.2.3 Vrealloc - 非常駐メモリの再割当て

C言語インタフェース

```
#include <tk/tkernel.h>
```

void* Vrealloc (void *ptr , size_t size);

パラメータ

void*	ptr	Pointer to Memory	再割当て対象のメモリアドレス
size_t	size	Size	再割当て後のメモリサイズ(バイト数)

リターンパラメータ

void*	addr	Memory Start Address	再割当てされたメモリの先頭アドレス

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ptr で指定した割当て済の非常駐メモリのサイズを、size で指定されたサイズに変更する。その際にメモリの再割当てを行い、再割当て後のメモリの先頭アドレスを addr に返す。

サイズ変更をともなうメモリの再割当てにより、一般にはメモリの先頭アドレスが移動し、addr は ptr と異なった値になる。ただし、その場合でも、再割当ての対象となったメモリの内容は保存される。このため、Vrealloc の処理の中でメモリ内容のコピーを行う。また、再割当てにより不要になったメモリは解放される。

ptr には、Vmalloc、Vcalloc、Vrealloc で割り当てられたメモリの先頭アドレスを指定する必要がある。ptr の正当性は呼出側で保証しなければならない。

ptr に NULL を指定した場合は、新しいメモリの割当てのみを行う。この場合の動作は Vmalloc() と同一である。

指定したサイズのメモリの再割当てができなかった場合や、size に0が指定された場合は、addr に NULL が返る。このとき、ptr に NULL 以外が指定されていれば、ptr のメモリの解放だけを行う。この場合の動作は Vfree() と同一である。

Vrealloc を含むメモリ割当てライブラリのAPIは、タスク独立部およびディスパッチ禁止中、割込み禁止中に呼び出すことはできない。呼び出した場合の動作は、システムダウンの可能性も含めて不定であり、呼出時の状態を保証するのは、呼出側の責任である。

補足事項

再割当てによりメモリサイズが小さくなる場合や、ptr で指定されるメモリの周辺に未割当てのメモリが残っている場合など、状況によっては、addr に返るメモリアドレスが ptr と同一となることがある。

内部的には、sizeで指定したバイト数よりも大きなメモリが割り当てられる場合がある。詳細は Vmalloc() の補足事項を参照のこと。

5.1.2.4 Vfree - 非常駐メモリの解放

C言語インタフェース

```
#include <tk/tkernel.h>
```

void Vfree (void *ptr);

パラメータ

void*	ptr	Pointer to Memory	解放するメモリの先頭アドレス

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ptr で指定した非常駐メモリを解放する。

ptr には、Vmalloc、Vcalloc、Vrealloc で割り当てられたメモリの先頭アドレスを指定する必要がある。ptr の正当性は呼出側で保証しなければならない。

Vfree を含むメモリ割当てライブラリのAPIは、タスク独立部およびディスパッチ禁止中、割込み禁止中に呼び出すことはできない。呼び出した場合の動作は、システムダウンの可能性も含めて不定であり、呼出時の状態を保証するのは、呼出側の責任である。

5.1.2.5 Kmalloc - 常駐メモリの割当て

C言語インタフェース

```
#include <tk/tkernel.h>
```

void* Kmalloc (size_t size);

パラメータ

size_t	size	Size	割り当てるメモリサイズ(バイト数)

リターンパラメータ

void*	addr	Memory Start Address	割り当てたメモリの先頭アドレス

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

size で指定したバイト数の常駐メモリを割り当て、その先頭アドレスを addr に返す。

指定したサイズのメモリの割当てができなかった場合や、size に0が指定された場合は、addr に NULL が返る。

Kmalloc を含むメモリ割当てライブラリのAPIは、タスク独立部およびディスパッチ禁止中、割込み禁止中に呼び出すことはできない。呼び出した場合の動作は、システムダウンの可能性も含めて不定であり、呼出時の状態を保証するのは、呼出側の責任である。

補足事項

内部的には、size で指定したバイト数よりも大きなメモリが割り当てられる場合がある。詳細は Vmalloc() の補足事項を参照のこと。

5.1.2.6 Kcalloc - 常駐メモリの割当て

C言語インタフェース

```
#include <tk/tkernel.h>
```

void* Kcalloc (size_t nmemb , size_t size);

パラメータ

size_t	nmemb	Number of Memory Blocks	割り当てるメモリブロックの個数
size_t	size	Size	割り当てるメモリブロックのサイズ(バイト数)

リターンパラメータ

void*	addr	Memory Start Address	割り当てたメモリの先頭アドレス

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

size で指定したバイト数のメモリブロックを、nmemb で指定された個数だけ連続して割り当て、0クリアしてから、その先頭アドレスを addr に返す。メモリ割当ての動作は、size と nmemb を乗じたバイト数のメモリブロック1個を割り当てるのと同じである。割り当てられるメモリは、常駐メモリである。

指定した個数のメモリブロックの割当てができなかった場合や、nmemb または size に0が指定された場合は、addr に NULL が返る。

Kcalloc を含むメモリ割当てライブラリのAPIは、タスク独立部およびディスパッチ禁止中、割込み禁止中に呼び出すことはできない。呼び出した場合の動作は、システムダウンの可能性も含めて不定であり、呼出時の状態を保証するのは、呼出側の責任である。

補足事項

内部的には、size と nmemb を乗じたバイト数よりも大きなメモリが割り当てられる場合がある。詳細は Vmalloc() の補足事項を参照のこと。

5.1.2.7 Krealloc - 常駐メモリの再割当て

C言語インタフェース

```
#include <tk/tkernel.h>
```

void* Krealloc (void *ptr , size_t size);

パラメータ

void*	ptr	Pointer to Memory	再割当て対象のメモリアドレス
size_t	size	Size	再割当て後のメモリサイズ(バイト数)

リターンパラメータ

void*	addr	Memory Start Address	再割当てされたメモリの先頭アドレス

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ptr で指定した割当て済の常駐メモリのサイズを、size で指定されたサイズに変更する。その際にメモリの再割当てを行い、再割当て後のメモリの先頭アドレスを addr に返す。

サイズ変更をともなうメモリの再割当てにより、一般にはメモリの先頭アドレスが移動し、addr は ptr と異なった値になる。ただし、その場合でも、再割当ての対象となったメモリの内容は保存される。このため、Krealloc の処理の中でメモリ内容のコピーを行う。また、再割当てにより不要になったメモリは解放される。

ptr には、Kmalloc、Kcalloc、Krealloc で割り当てられたメモリの先頭アドレスを指定する必要がある。ptr の正当性は呼出側で保証しなければならない。

ptr に NULL を指定した場合は、新しいメモリの割当てのみを行う。この場合の動作は Kmalloc() と同一である。

指定したサイズのメモリの再割当てができなかった場合や、size に0が指定された場合は、addr に NULL が返る。このとき、ptr に NULL 以外が指定されていれば、ptr のメモリの解放だけを行う。この場合の動作は Kfree() と同一である。

Krealloc を含むメモリ割当てライブラリのAPIは、タスク独立部およびディスパッチ禁止中、割込み禁止中に呼び出すことはできない。呼び出した場合の動作は、システムダウンの可能性も含めて不定であり、呼出時の状態を保証するのは、呼出側の責任である。

補足事項

再割当てによりメモリサイズが小さくなる場合や、ptr で指定されるメモリの周辺に未割当てのメモリが残っている場合など、状況によっては、addr に返るメモリアドレスが ptr と同一となることがある。

内部的には、sizeで指定したバイト数よりも大きなメモリが割り当てられる場合がある。詳細は Vmalloc() の補足事項を参照のこと。

5.1.2.8 Kfree - 常駐メモリの解放

C言語インタフェース

```
#include <tk/tkernel.h>
```

void Kfree (void *ptr);

パラメータ

void*	ptr	Pointer to Memory	解放するメモリの先頭アドレス

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ptr で指定した常駐メモリを解放する。

ptr には、Kmalloc、Kcalloc、Krealloc で割り当てられたメモリの先頭アドレスを指定する必要がある。ptr の正当性は呼出側で保証しなければならない。

Kfree を含むメモリ割当てライブラリのAPIは、タスク独立部およびディスパッチ禁止中、割込み禁止中に呼び出すことはできない。呼び出した場合の動作は、システムダウンの可能性も含めて不定であり、呼出時の状態を保証するのは、呼出側の責任である。

5.2 アドレス空間管理機能

アドレス空間管理機能は、論理アドレス空間に対して各種の操作や管理を行うための機能である。本機能は、主としてMMUやページテーブルを操作することによって実現され、タスク固有空間の設定を行うアドレス空間設定、アクセス権のチェックを行うアドレス空間チェック、メモリ領域のロック(常駐化)、論理アドレスと物理アドレスの変換やマッピングなどの機能を提供する。

本機能は、デバイスドライバやサブシステム等のシステムプログラムを実装するために利用されるほか、デマンドページングに関する処理を行うサブシステムを併用することにより、仮想記憶システムを実現するためにも利用される。

なお、MMUを使用しないシステムであっても、アドレス空間管理機能のAPIは提供される。移植性や拡張性への配慮から、MMUを使用しないシステム上でも、アプリケーションがこれらのAPIを適切に使用することが望ましい。

T-Kernelでは、実行時の保護レベルとして0～3の4段階(特権モード／ユーザモードなどに相当)、アクセス対象となるメモリの保護レベルとしても0～3の4段階がそれぞれ定義されており、現在実行中の保護レベルと同じか、それより低い保護レベルのメモリにのみアクセスができる。実行時のメモリアクセス権のチェックはMMUが行う。この機能は、プログラムの不正なアクセスから、OSなどのシステムを保護するために役立つ。メモリアクセス権のチェック機能を実現するために、T-Kernelは、MMU等の設定を適切に行う。

メモリの呼出元アクセス権情報は、基本的に拡張SVCを呼び出す直前の保護レベルでアクセスできる権利を示す情報であり、タスクごとに保持される。拡張SVCを呼び出す直前の保護レベルを示す情報であるため、一般には現在の保護レベルと一致しない。例えば、保護レベル3で実行中のタスクが拡張SVCを呼び出した場合、拡張SVCの実行中の保護レベルは0となるが、タスクがアクセスできる権利としては保護レベル3である。拡張SVC(a)からさらにネストして拡張SVC(b)を呼び出した場合、ネストして呼び出した拡張SVC(b)での呼出元アクセス権情報は、拡張SVC(b)を呼び出す直前の保護レベル、すなわち拡張SVC(a)を実行中の保護レベル0である。

メモリの呼出元アクセス権情報は次のように設定される。

- タスク起動直後は、そのタスクの生成時に指定された実行時の保護レベルが、呼出元アクセス権情報として設定される。
- 拡張SVCを呼び出すと、呼び出す直前に実行していた実行時の保護レベルが、呼出元アクセス権情報として設定される。
- 拡張SVCから戻ると、呼び出す直前の呼出元アクセス権情報に戻る。
- SetTaskSpace() を発行すると、対象タスクがその時点で拡張SVCを呼び出す直前に実行していた実行時の保護レベルが、自タスクの呼出元アクセス権情報として設定される。ネストして拡張SVCが呼び出されている場合には、最後に呼ばれた拡張SVCを呼び出す直前の保護レベルが設定される。なお、対象となるタスクがタスク部を実行中の場合には、対象タスクの生成時に指定された実行時の保護レベルが、自タスクの呼出元アクセス権情報として設定される。

メモリの呼出元アクセス権情報は、拡張SVCから呼び出し元の保護レベルに基づく処理を行うことを想定して保持される。一例としてアドレス空間チェック機能(ChkSpaceXXX)では、現在の実行時の保護レベルの代わりに呼出元アクセス権情報を利用することで、拡張SVCの呼び出し元の権限でのメモリアクセス権の検査が実現されている。

5.2.1 アドレス空間設定

T-Kernelのアドレス空間の扱いに関しては、「2.7.1節「アドレス空間」」で説明している。アドレス空間設定の機能では、タスクのアドレス空間と呼出元アクセス権情報を設定するAPIを提供する。

5.2.1.1 SetTaskSpace - タスクのアドレス空間設定

C言語インタフェース

```
#include <tk/tkernel.h>

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

パラメータ

ID	tskid	Task ID	設定元のアドレス空間を持つタスクのタスクID

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	tskid が不正
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	TSK_SELF 以外で自タスクを指定した

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

自タスクのタスク固有空間と呼出元アクセス権情報を、tskid で指定したタスクの情報に基づいて設定する。これにより、このAPIを実行したタスクは、tskid のタスクと同じアドレス空間と、指定されたタスクがその時点で拡張SVCを呼び出す直前に実行していた実行時の保護レベルを示す呼出元アクセス権情報を持つことになる。ネストして拡張SVCが呼び出されている場合には、最後に呼ばれた拡張SVCを呼び出す直前の保護レベルが自タスクの呼出元アクセス権情報として設定される。なお、tskid で指定したタスクがタスク部を実行中の場合には、指定したタスクの生成時に指定された実行時の保護レベルが、自タスクの呼出元アクセス権情報として設定される。

なお、本APIの実行後に、tskid のタスク(対象タスク)のアドレス空間や呼出元アクセス権情報が変わった場合でも、自タスクのアドレス空間や呼出元アクセス権情報は影響を受けない。すなわち、本APIを実行する時点における対象タスクの状態が自タスクに反映されるだけで、その後の対象タスクの状態に自タスクが追従するわけではない。

拡張SVCの実行中に本APIを実行した場合、拡張SVCから戻ると、呼出元アクセス権情報については拡張SVCの呼出前の状態に戻る。一方、タスク固有空間は拡張SVCの呼出前の状態には戻らない。本APIで設定されたタスク固有空間が、拡張SVCから戻った後もそのまま有効である。

tskid に自タスクのタスクIDを指定することはできない。ただし、TSK_SELF により自タスクを指定した場合は、現在実行中の保護レベルの呼出元アクセス権情報が設定される。このとき、タスク固有空間は切り替わらない。

なお、呼出元アクセス権情報の設定により、実行時の保護レベルが変更されるわけではないことに注意が必要である。

補足事項

SetTaskSpace() は、デバイスドライバやサブシステムの処理を行うために、処理の依頼元のタスクA(デバイス管理やサブシステムの拡張SVCを呼んだタスク)とは別の処理用タスクBが動作しているような状況において、処理用タスクBのタスク固有空間および呼出元アクセス権情報を、依頼元のタスクAと同じ設定にするために使用する。

たとえば、デバイスドライバの処理用タスクBが、デバイスから入力データを読み込み、依頼元のタスクAが指定したバッファXにそのデータを格納するケースを考える。ここで、バッファXのアドレスがタスクAのタスク固有空間の中にあり、かつ処理用タスクBと依頼元タスクAのタスク固有空間が異なっていた場合には、処理用タスクBからバッファXにアクセスできず、入力データを格納することもできない。

このようなケースでは、処理用タスクBがあらかじめ SetTaskSpace() を実行し、自タスクのタスク固有空間を依頼元タスクAと同じ設定にしておくことにより、バッファXへのアクセスが可能となる。また、処理用タスクBの呼出元アクセス権情報が依頼元タスクAと同じになるので、バッファXに入力データを格納する際のアクセス権のチェックも適切に行われる。

呼出元アクセス権情報の設定を行わず、タスク固有空間の設定のみを行う場合には、tk_set_tsp を利用する。

5.2.2 アドレス空間チェック

現在の呼出元アクセス権情報にしたがって、指定されたメモリ領域へのアクセスが許可されているか検査する。

検査のために、ChkSpaceXXX()のAPIを提供する。APIの名称の最後の文字の意味は次の通りである。

- ～R 読込みアクセス権があるか検査する。
- ～RW 読込みおよび書込みアクセス権があるか検査する。
- ～RE 読込みアクセス権および実行権があるか検査する。

現在の呼出元アクセス権情報にて対象となるメモリ領域へのアクセスが許可されていない場合や、対象となるメモリ領域にメモリが存在しない場合には、E_MACV のエラーを返す。対象となるメモリ領域の一部のアクセスが許可されていない場合や、一部のメモリが存在しない場合も、同様に E_MACV のエラーを返す。

なお、検査対象のメモリ領域がタスク固有空間の場合には、現在設定されているタスク固有空間がそのまま使用される。

補足事項
低い保護レベルで動作している一般のアプリケーションタスクAが、高い保護レベルで動作するデバイスドライバやサブシステムに処理を依頼するようなケースにおいて、処理のパラメータやリターンパラメータをメモリ領域Xに置いた場合には、依頼元のタスクAがメモリ領域Xに対してアクセス可能かどうかをデバイスドライバやサブシステムの側でチェックする必要がある。このチェックを行わないと、たとえば、タスクAからアクセス権のないメモリ領域を、デバイスドライバやサブシステム経由で不正にアクセスできてしまうからである。アドレス空間チェックのAPIは、このような場面でのチェックを行うために利用することを想定した機能である。

5.2.2.1 ChkSpaceR - メモリ読込みアクセス権の検査

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = ChkSpaceR (CONST void *addr , INT len);

パラメータ

CONST void*	addr	Memory Start Address	検査対象メモリの開始アドレス
INT	len	Length	検査対象メモリのサイズ(バイト数)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_MACV	メモリアクセス不能

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

addr が示すアドレスから len バイトの領域に対して、現在の呼出元アクセス権情報により読込みアクセスが可能かどうかを検査する。アクセス可能であれば E_OK を返し、アクセス不能であれば E_MACV を返す。

5.2.2.2 ChkSpaceRW - メモリ読込み書込みアクセス権の検査

C言語インタフェース

```
#include <tk/tkernel.h>

ER ercd = ChkSpaceRW (CONST void *addr , INT len );
```

パラメータ

CONST void*	addr	Memory Start Address	検査対象メモリの開始アドレス
INT	len	Length	検査対象メモリのサイズ(バイト数)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_MACV	メモリアクセス不能

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

addr が示すアドレスから len バイトの領域に対して、現在の呼出元アクセス権情報により読込みアクセスと書込みアクセスの両方が可能かどうかを検査する。両方のアクセス可能であれば E_OK を返し、少なくとも一方のアクセスが不能であれば E_MACV を返す。

5.2.2.3 ChkSpaceRE - メモリ読込みアクセス権および実行権の検査

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = ChkSpaceRE (CONST void *addr , INT len);

パラメータ

CONST void*	addr	Memory Start Address	検査対象メモリの開始アドレス
INT	len	Length	検査対象メモリのサイズ(バイト数)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_MACV	メモリアクセス不能

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

addr が示すアドレスから len バイトの領域に対して、現在の呼出元アクセス権情報により読込みアクセスが可能で、かつプログラムとして実行が可能かどうかを検査する。両方とも可能であれば E_OK を返し、少なくとも一方が不能であれば E_MACV を返す。

5.2.2.4 ChkSpaceBstrR - 文字列読込みアクセス権の検査

C言語インタフェース

```
#include <tk/tkernel.h>

INT rlen = ChkSpaceBstrR (CONST UB *str , INT max );
```

パラメータ

CONST UB*	str	String	検査対象の文字列の開始アドレス
INT	max	Max Length	検査対象の文字列の長さの最大値

リターンパラメータ

INT	rlen	Result Length	アクセス可能な文字列の長さ
	または	Error Code	エラーコード

エラーコード

E_MACV	メモリアクセス不能

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

str から文字列の終端('¥0')に達するか max 文字目(バイト数)に達するまでのメモリ領域に対して、現在の呼出元アクセス権情報により読込みアクセスが可能かどうかを検査する。max＝0の場合は、文字列終端まで検査する。

アクセス可能であれば、その文字列の長さ(バイト数)を返す。max 文字目までに文字列の終端があった場合は'¥0'の直前までの長さ、文字列の終端より先に max 文字に達した場合は max を返す。

アクセス不能であれば、E_MACV のエラーを返す。

5.2.2.5 ChkSpaceBstrRW - 文字列読込み書込みアクセス権の検査

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT rlen = ChkSpaceBstrRW (CONST UB *str , INT max);

パラメータ

CONST UB*	str	String	検査対象の文字列の開始アドレス
INT	max	Max Length	検査対象の文字列の長さの最大値

リターンパラメータ

INT	rlen	Result Length	アクセス可能な文字列の長さ
	または	Error Code	エラーコード

エラーコード

E_MACV	メモリアクセス不能

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

str から文字列の終端('¥0')に達するか max 文字目(バイト数)に達するまでのメモリ領域に対して、現在の呼出元アクセス権情報により読込みアクセスおよび書込みアクセスが可能かどうかを検査する。max＝0の場合は、文字列終端まで検査する。

読込み、書込みともアクセス可能であれば、その文字列の長さ(バイト数)を返す。max 文字目までに文字列の終端があった場合は'¥0'の直前までの長さ、文字列の終端より先に max 文字に達した場合は max を返す。

読込み、書込みの少なくとも一方がアクセス不能であれば、E_MACV のエラーを返す。

5.2.2.6 ChkSpaceTstrR - TRONコード文字列読込みアクセス権の検査

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT rlen = ChkSpaceTstrR (CONST TC *str , INT max);

パラメータ

CONST TC*	str	String	検査対象の文字列の開始アドレス
INT	max	Max Length	検査対象の文字列の長さの最大値

リターンパラメータ

INT	rlen	Result Length	アクセス可能な文字列の長さ
	または	Error Code	エラーコード

エラーコード

E_MACV	メモリアクセス不能

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

str からTRONコード文字列の終端(TNULL＝0x0000)に達するか max 文字目(TRONコード文字数)に達するまでのメモリ領域に対して、現在の呼出元アクセス権情報により読込みアクセスが可能かどうかを検査する。max＝0の場合は、文字列終端まで検査する。

アクセス可能であれば、その文字列の長さ(TRONコード文字数)を返す。max 文字目までに文字列の終端があった場合は TNULL の直前までの長さ、文字列の終端より先に max 文字に達した場合は max を返す。

アクセス不能であれば、E_MACV のエラーを返す。

str は、偶数アドレスでなければならない。

5.2.2.7 ChkSpaceTstrRW - TRONコード文字列読込み書込みアクセス権の検査

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT rlen = ChkSpaceTstrRW (CONST TC *str , INT max);

パラメータ

CONST TC*	str	String	検査対象の文字列の開始アドレス
INT	max	Max Length	検査対象の文字列の長さの最大値

リターンパラメータ

INT	rlen	Result Length	アクセス可能な文字列の長さ
	または	Error Code	エラーコード

エラーコード

E_MACV	メモリアクセス不能

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

str からTRONコード文字列の終端(TNULL＝0x0000)に達するか max 文字目(TRONコード文字数)に達するまでのメモリ領域に対して、現在の呼出元アクセス権情報により読込みアクセスおよび書込みアクセスが可能かどうかを検査する。max＝0の場合は、文字列終端まで検査する。

読込み、書込みともアクセス可能であれば、その文字列の長さ(TRONコード文字数)を返す。max 文字目までに文字列の終端があった場合は TNULL の直前までの長さ、文字列の終端より先に max 文字に達した場合は max を返す。

読込み、書込みの少なくとも一方がアクセス不能であれば、E_MACV のエラーを返す。

str は、偶数アドレスでなければならない。

5.2.3 論理アドレス空間管理

論理アドレス空間管理の機能では、アドレス変換(論理アドレスから物理アドレスへの変換)、メモリの常駐化、メモリアクセス権の設定に関連したAPIを提供する。

T-Kernelでは、メモリに対するアクセス権の管理やタスク固有空間の実現、メモリの効率的な利用などを行うために、MMUを利用したアドレス変換(論理アドレスから物理アドレスへの変換)を行っている。通常のプログラムは論理アドレス空間の中で動作しており、物理アドレスを扱う必要はないが、DMA転送を行うデバイスドライバなど、ハードウェアを直接操作する一部のシステムプログラムでは物理アドレスを扱う場合がある。この時、論理アドレスと物理アドレスとの対応関係の取得や設定を行う必要があるので、そのためのAPIとして CnvPhysicalAddr()、MapMemory()、UnmapMemory() が提供される。

また、T-Kernel上に仮想記憶システムを実現した場合には、プログラムAからアクセスされるメモリが物理的に存在しない状態(ページアウトされた状態)が発生する。ページアウトされたメモリへのアクセスがあった場合、CPUによっては、MMUがそれを検出してページフォルトのCPU例外を発生し、その例外を処理する仮想記憶システムがページアウトされていたメモリの内容をディスク(二次記憶装置)からメモリに戻す(ページイン)。このような処理を行うことによって、プログラムAから見ると、アクセス対象メモリがページアウトされているかどうかを気にすることなく、処理を進めることができる。これは、仮想記憶システムの一般的な実装方法である。

しかし、タスク独立部実行中、ディスパッチ禁止中、割込み禁止中のプログラムでは、ページフォルト発生時にも、上記のようなページインの処理を行うことができない。このため、プログラムの実行中にページフォルトが発生しないように、あらかじめアクセス対象となるすべてのメモリをページインして常駐化しておく必要がある。また、DMA転送を行う場合や、時間制約の強いプログラムを実行する場合にも、同様の処置が必要になる。このような場合に使用するAPIとして、メモリ領域をロック(常駐化)する LockSpace() や、それを解除する UnlockSpace() が提供される。

このほか、アドレス空間の各種情報を取得する GetSpaceInfo()、メモリアクセス権の設定を行う SetMemoryAccess() といったAPIが提供される。

上記のうち、DMA転送に関連した処理を行うAPIでは、DMA転送に合わせたメモリキャッシュ制御も行う、具体的には、CnvPhysicalAddr() によって論理アドレスから物理アドレスへの変換を行う際、対象となった領域は、DMA転送ができるようにメモリキャッシュがオフになる。DMA転送の終了後、UnlockSpace() を実行してメモリの常駐解除を行うと、キャッシュオン状態に戻る。

補足事項
T-Kernel/SMでは、論理アドレス空間と物理アドレス空間の対応関係(マッピング)、メモリのアクセス権、ページ不在や常駐化などの管理を行うために、MMUやページテーブルの設定や操作を行う。しかし、T-Kernel本体だけで仮想記憶システムを実現するわけではない。実際に仮想記憶を実現するためには、物理メモリとディスク(二次記憶装置)との間でページインやページアウトの処理を行うなど、他にもいろいろな処理が必要である。これらの処理は、T-Kernel本体ではなく、仮想記憶を実現するためのサブシステム(T-Kernel Extensionの一部など)によって行われる。

5.2.3.1 LockSpace - メモリ領域のロック

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = LockSpace (CONST void *addr , INT len);

パラメータ

CONST void*	addr	Memory Start Address	ロックするメモリの開始アドレス
INT	len	Length	ロックするメモリのサイズ(バイト数)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(len≦0)
E_MACV	メモリ領域ではない部分を指定した
E_NOMEM	メモリ不足(常駐化に伴うページインのためのメモリが確保できない)
E_LIMIT	ロック回数の上限を超えてロックした

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

論理アドレス addr から len バイトのメモリ領域(ロック対象領域)をロック(常駐化)する。本APIによる常駐化の後、ロック対象領域はページアウトされなくなり、常に物理アドレス空間にマッピングされるとともに、実メモリ(物理的なメモリ)も割り当てられた状態となる。

ロック対象領域の一部がページアウトされていた場合には、その領域に対してページインの処理を行ってから常駐化する。ページインの際に実メモリの確保ができなかった場合には、E_NOMEM のエラーを返す。

同じメモリ領域に対して複数回の LockSpace() を実行することができる。このとき、LockSpace() の回数はカウントされ、同じ回数の UnlockSpace() の実行によって常駐化が解除される。すなわち、LockSpace() により常駐化した状態をネストすることができる。ただし、この場合のネストの多重度(LockSpace() と UnlockSpace() の実行回数の差)には実装依存の上限値があり、その上限値を越えて LockSpace() を実行した場合には、E_LIMIT のエラーを返す。

len に0以下の値を指定した場合には、E_PAR のエラーを返す。また、メモリ領域ではない部分(メモリの割当てを想定していない論理アドレス)がロック対象領域に含まれていた場合には、E_MACV のエラーを返す。

本APIによるロック(常駐化)の処理は、MMUの機能を使ってページ単位で行われる。このため、addr がページの先頭アドレスでない場合や、len がページサイズの整数倍でない場合は、addr と len で指定した範囲を含むページ全体がロック対象領域となる。たとえば、len に1を指定しても、1ページ分の領域がロックされることになる。

MMUを使用しないシステムにおいては、すべてのメモリを常駐メモリとみなすことができる。そのため、LockSpace() において具体的な処理をする必要はないが、MMUを使用するシステムとの互換性を考慮し、エラーとはせずに E_OK を返す。MMUを使用しないシステムにおいて、E_PAR などエラーをチェックするかどうかは実装依存である。

補足事項

LockSpace() によるメモリ常駐化の処理のうち、ページインなどの処理は、仮想記憶を実現するサブシステムを呼び出すことによって行われる。呼出のインタフェースは実装依存である。

MapMemory() によって割り当てられた論理アドレス空間を、LockSpace() のロック対象領域に含めてはいけない。この場合の動作は保証されない。

5.2.3.2 UnlockSpace - メモリ領域のアンロック

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = UnlockSpace (CONST void *addr , INT len);

パラメータ

CONST void*	addr	Memory Start Address	アンロックするメモリの開始アドレス
INT	len	Length	アンロックするメモリのサイズ(バイト数)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(len≦0)
E_MACV	メモリ領域ではない部分を指定した
E_LIMIT	ロックされていない領域を指定した

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

論理アドレス addr から len バイトの領域(アンロック対象領域)をアンロック(常駐解除)する。本APIによる常駐解除の後、アンロック対象領域はページアウトの対象になる。

また、アンロック対象領域のメモリキャッシュモードがオフになっていた場合は、オンの設定に変更する。

アンロック対象領域としては、必ず LockSpace() の発行時に指定したロック対象領域と同じ領域を指定する必要がある。ロックした領域の一部のみをアンロックすることはできない。ただし、このような使い方をした場合でも、T-Kernelではエラーを検出できない。呼出側の責任で、必ず同じ領域を指定しなければならない。

同じメモリ領域に対して複数回の LockSpace() が実行されていた場合、同じ回数の UnlockSpace() の実行によって常駐解除される。UnlockSpace() の回数が LockSpace() の回数に満たず、常駐解除されない場合でも、UnlockSpace() はエラーとはならずに E_OK を返す。一方、まだロックされていない領域をアンロック対象領域として指定した場合には、ロックカウントのエラーとして E_LIMIT を返す。

len に0以下の値を指定した場合には、E_PAR のエラーを返す。また、メモリ領域ではない部分(メモリの割当てを想定していない論理アドレス)がアンロック対象領域に含まれていた場合には、E_MACV のエラーを返す。

本APIによるアンロック(常駐解除)の処理は、MMUの機能を使ってページ単位で行われる。このため、addr がページの先頭アドレスでない場合や、len がページサイズの整数倍でない場合は、addr と len で指定した範囲を含むページ全体がアンロック対象領域となる。たとえば、len に1を指定しても、1ページ分の領域がアンロックされることになる。

MMUを使用しないシステムにおいては、すべてのメモリを常駐メモリとみなすことができる。そのため、UnlockSpace() においても、LockSpace() と同様に具体的な処理をする必要はないが、MMUを使用するシステムとの互換性を考慮し、エラーとはせずに E_OK を返す。MMUを使用しないシステムにおいて、E_PAR などエラーをチェックするかどうかは実装依存である。

補足事項

MapMemory() によって割り当てられた論理アドレス空間を、UnlockSpace() のアンロック対象領域に含めてはいけない。この場合の動作は保証されない。

DMA転送を行う際には、バッファとなるメモリ領域を常駐化し、メモリキャッシュモード設定をオフにした上で、バッファの物理アドレスをDMAコントローラに設定する必要がある。この場合の通常の手順は次のようになる。

1. LockSpace() によってバッファの常駐化を行う。
2. CnvPhysicalAddr() によってバッファの物理アドレスを取得するとともに、バッファのメモリキャッシュモード設定をオフにする。
3. バッファと入出力デバイスとの間でDMA転送を行う。
4. UnlockSpace() によって、バッファの常駐解除を行うとともに、バッファのメモリキャッシュモード設定をオンに戻す。

ただし、UnlockSpace() では、上記のようなAPIの発行履歴とは無関係にメモリキャッシュモードをオンに設定する。UnlockSpace() の実行によりメモリキャッシュモード設定が変更される可能性について、注意が必要である。

5.2.3.3 CnvPhysicalAddr - 物理アドレスの取得

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT rlen = CnvPhysicalAddr (CONST void *vaddr , INT len , void **paddr);

パラメータ

CONST void*	vaddr	Vitrual Address	変換元の論理アドレス
INT	len	Length	領域サイズ(バイト数)
void**	paddr	Pointer to Physical Address	論理アドレスに対応する物理アドレスを返す領域へのポインタ

リターンパラメータ

INT	rlen	Result Length	連続物理アドレス空間のサイズ(バイト数)
	または	Error Code	エラーコード
void*	paddr	Physical Address	論理アドレスに対応する物理アドレス

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(対象領域のキャッシュ制御ができない)
E_MACV	メモリ領域ではない部分を指定した

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

論理アドレス vaddr のメモリ領域に対応する物理アドレスを求め、paddr へ返す。また、vaddr から len バイトのメモリ領域のうち、対応する物理アドレスが連続しているサイズ(バイト数)を戻値 rlen に返す。すなわち、論理アドレスと物理アドレスの対応関係が連続するのは、rlen のサイズまでであり、rlen≦len である。論理アドレス空間の vaddr から rlen のサイズの連続領域が、物理アドレス空間の paddr から rlen のサイズの連続領域に対応するということになる。

また、物理アドレス空間の paddr から rlen のサイズの領域(対象領域)に対して、メモリキャッシュモードをオフに設定する。これは、CnvPhysicalAddr() の実行後にDMA転送を行うことを想定しているためである。ただし、ハードウェアの制限から、部分的なメモリのキャッシュのオフができない場合には、キャッシュのフラッシュ(ライトバックして無効化)を行うものとする。

CnvPhysicalAddr() では、対象領域の常駐化の処理は行わない。DMA転送を行う場合には、バッファとして使用する領域に対して別途 LockSpace() を発行し、バッファ領域を常駐化(ロック)しておかなければならない。

len に0以下の値を指定した場合には、E_PAR のエラーを返す。また、vaddr から len バイトの領域の中に、メモリ領域ではない部分(メモリの割当てを想定していない論理アドレス)が含まれていた場合には、E_MACV のエラーを返す。

補足事項

CnvPhysicalAddr() は、DMA転送の準備のために使用することを想定したAPIである。DMA転送の際の具体的な使い方については、UnlockSpace() の補足事項を参照のこと。

CnvPhysicalAddr() の対象領域に対しては、キャッシュモードをオフに設定するだけでなく、メモリアクセスの完了を保証するメモリ属性に設定することが望ましい。

5.2.3.4 MapMemory - メモリのマップ

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = MapMemory (CONST void *paddr , INT len , UINT attr , void **laddr);

パラメータ

CONST void*	paddr	Physical Address	マップする物理アドレス
INT	len	Length	マップするメモリのサイズ(バイト数)
UINT	attr	Attribute	マップする際のメモリ属性
void**	laddr	Pointer to Logical Address	マップ先の論理アドレスを返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード
void*	laddr	Logical Address	マップ先の論理アドレス

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(len≦0)
E_LIMIT	マップ先の論理アドレス空間が不足
E_NOMEM	割り当てる実メモリが不足、論理アドレス空間の管理用メモリが不足

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

物理アドレス paddr から len バイトの連続領域を論理アドレス空間にマップし、マップ先の先頭の論理アドレスを laddr に返す。マップされたメモリ領域は、常駐化(ロック)された状態となる。また、マップされたメモリ領域に、attr で指定した属性を設定する。

attr では、次の属性を指定する。

```
attr := (MM_USER || MM_SYSTEM) | [MM_READ] | [MM_WRITE] | [MM_EXECUTE] | [MM_CDIS]
```

MM_USER	ユーザレベルアクセス可
MM_SYSTEM	システムレベルアクセス可
MM_READ	読込みアクセス可
MM_WRITE	書込みアクセス可
MM_EXECUTE	プログラム実行可
MM_CDIS	キャッシュ禁止

ハードウェアや実装によっては、上記の以外の属性を指定できる場合もある。

paddr に NULL を指定した場合には、物理アドレスの連続した len バイトの実メモリを確保して割当てを行い、その実メモリの物理アドレス空間を論理アドレス空間にマップする。

len に0以下の値を指定した場合には、E_PAR のエラーを返す。また、マップ先の論理アドレス空間が不足していて割当てに失敗した場合には、E_LIMIT のエラーを返す。論理アドレス空間を管理するために必要なメモリが確保できなかった場合や、paddr に NULL を指定したケースで実メモリの確保ができなかった場合には、E_NOMEM のエラーを返す。

補足事項

MapMemory() は、物理アドレス空間に置かれた入出力デバイス(Video RAMなど)の領域を、デバイスドライバなどのプログラムから直接アクセス可能な論理アドレス空間にマッピングするための機能である。

マップ先の論理アドレス laddr は、本APIの実行時に自動的に割り当てられる。マップ先の論理アドレスを指定することはできない。

T-Kernelが管理するシステムメモリに含まれるアドレスを paddr に指定してはならない。MapMemory() によってシステムメモリの確保を行いたい場合には、paddr に NULL を指定し、T-Kernelによって自動的に割り当てられたシステムメモリを使用する。

attr で指定する属性のシンボル(ニーモニック)に対応する値は、実装によって異なっている場合がある。そのため、互換性への配慮から、attr の指定には必ず上記のシンボルを使用すべきである。

既に MapMemory() の対象となっている物理アドレス領域に対して、重複して MapMemory() を実行してはいけない。また、MapMemory() によって割り当てたメモリは常駐メモリであるが、この常駐は解除できず、常駐解除のためにUnlockSpace()を呼び出してはならない。このような使い方をしないことは、呼出側で保証する必要がある。

MapMemory() を実行した後、何らかの方法にて、laddr として割り当てられた論理アドレスを経由せず、paddr 以降の物理アドレスに直接アクセスした場合、キャッシュ等の不整合が生じる可能性がある。このような使い方をする場合は、呼出側の責任において、データの整合性などに十分注意する必要がある。

attr に MM_CDIS を指定した場合のメモリ属性は、キャッシュを使用しない属性であるだけでなく、メモリアクセスの完了を保証する属性であることが望ましい。

5.2.3.5 UnmapMemory - メモリのアンマップ

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = UnmapMemory (CONST void *laddr);

パラメータ

CONST void*	laddr	Logical Address	アンマップする論理アドレス

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(laddr が不正)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

MapMemory() で割り当てられた論理アドレス空間をアンマップ(解放)する。laddr にはアンマップの対象となる領域の論理アドレスを指定するが、この値は、MapMemory() の戻値 laddr で得た値でなければならない。

MapMemory() の実行時に paddr＝NULL を指定し、実メモリの割当ても行っていた場合には、UnmapMemory() の実行によりそのメモリも解放される。

5.2.3.6 GetSpaceInfo - アドレス空間の各種情報の取得

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = GetSpaceInfo (CONST void *addr , INT len , T_SPINFO *pk_spinfo);

パラメータ

CONST void*	addr	Start Address	情報を取得する先頭論理アドレス
INT	len	Length	情報を取得する領域サイズ(バイト数)
T_SPINFO*	pk_spinfo	Packet to Refer Space Info	アドレス空間情報を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_spinfo の内容

void*	paddr	Physical Address	addr に対応する物理アドレス
void*	page	Page Start Address	addr が属するページの先頭物理アドレス
INT	pagesz	Page Size	ページサイズ(バイト数)
INT	cachesz	Cache Line Size	キャッシュラインサイズ(バイト数)
INT	cont	Continuous Length	物理アドレスが連続している領域のサイズ(バイト数)

——(以下に実装独自に他の情報を追加してもよい)——

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(addr, len, pk_spinfo が不正あるいは利用できない)
E_MACV	メモリアクセス不能、メモリアクセス権違反

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

論理アドレス addr から len の領域のアドレス空間情報を取得して、リターンパラメータ pk_spinfo に返す。paddr には addr に対応する物理アドレスを返す。page には addr が属するページの先頭の物理アドレスを返す。

pagesz にはページサイズを返す。ページサイズは、MMUで定義されるページサイズであり、SetMemoryAccess() でメモリアクセス権を設定したり、SetCacheMode() でキャッシュモードを設定したりする際の単位となるページサイズと同じ値である。

cachesz にはキャッシュラインサイズを返す。キャッシュラインサイズは、ControlCache() でキャッシュを制御する際の単位となるキャッシュラインサイズと同じ値である。

cont には addr から len バイトの領域のうち、対応する物理アドレスが連続しているサイズ(バイト数)を返す。すなわち、論理アドレスと物理アドレスの対応関係が連続するのは、cont のサイズまでであり、cont≦len である。論理アドレス空間の addr

から cont のサイズの連続領域が、物理アドレス空間の paddr から cont のサイズの連続領域に対応するということになる。

途中にページアウトされている領域が存在した場合は、その領域の直前までの物理アドレスが連続しているものとみなす。特に、addr の属するページがページアウトされていた場合は、cont＝0を返す。このとき、戻値 ercd は E_OK を返すが、pk_spinfo に返されるリターンパラメータのうち、cont 以外の内容は不定となる。

len に0以下の値を指定した場合には、E_PAR のエラーを返す。エラーが発生した場合に pk_spinfo に設定される内容は不定である。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.2.3.7 SetMemoryAccess - メモリアクセス権の設定

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT rlen = SetMemoryAccess (CONST void *addr , INT len , UINT mode);

パラメータ

CONST void*	addr	Start Address	アクセス権を設定するメモリ領域の先頭アドレス
INT	len	Length	アクセス権を設定するメモリ領域のサイズ(バイト数)
UINT	mode	Memory Access Mode	設定するメモリアクセス権を示すモード

リターンパラメータ

INT	rlen	Result Length	メモリアクセス権を設定できた領域のサイズ(バイト数)
	または	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(addr, len, mode が不正あるいは利用できない)
E_NOSPT	未サポート機能(mode で指定した機能が未サポート)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

論理アドレス addr から len バイトのメモリ領域に対して、mode で指定したメモリアクセス権を設定する。また、実際にメモリアクセス権を設定できた領域のサイズ(バイト数)を戻値 rlen に返す。

mode では、次のメモリアクセス権を指定する。

```
mode := ( MM_EXECUTE | MM_READ | MM_WRITE )
                MM_EXECUTE      プログラム実行アクセス可能
                MM_READ         読み出しアクセス可能
                MM_WRITE        書き込みアクセス可能
                    ...
                /* 実装独自のモードを追加してもよい */
```

本APIによるメモリアクセス権の設定は、MMUの機能を使ってページ単位で行われる。このため、addr がページの先頭アドレスでない場合や、len がページサイズの整数倍でない場合は、addr と len で指定した範囲を含むページ全体がメモリアクセス権設定の対象領域となる。たとえば、len に1を指定しても、1ページ分のメモリアクセス権が設定されることになる。

ハードウェアや実装によっては、上記の以外のメモリアクセス権を指定できる場合もある。また、ハードウェアや実装によっては、上記のメモリアクセス権の一部または全部の設定ができない場合がある。設定できないメモリアクセス権を mode で指定した場合には、E_NOSPT のエラーを返す。

補足事項

通常のアプリケーションなどで使用するメモリ領域に対しては、T-Kernelがあらかじめ適切なメモリアクセス権の設定を行っている。このため、通常のアプリケーションが SetMemoryAccess() を使う必要はない。SetMemoryAccess() を使用するのは、システムのメモリ割当てやセキュリティを動的に管理するプログラム、デバッグ用のプログラムなど、通常のアプリケーションとは異なる特別な目的を持ったプログラムである。

`mode` で指定するメモリアクセス権は、MapMemory の `attr` で指定する属性の一部と共通である。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.3 デバイス管理機能

デバイス管理機能は、T-Kernel上で動作するデバイスドライバを管理するための機能である。

デバイスドライバとは、主としてハードウェアのデバイスの操作や入出力を行うために、T-Kernel本体とは独立して実装されるプログラムである。アプリケーションやミドルウェアによるデバイスの操作や入出力を、デバイスドライバ経由で行うことにより、個々のデバイスの仕様の差異をデバイスドライバ側で吸収し、アプリケーションやミドルウェアのハードウェア非依存性や互換性を高めることができる。

デバイス管理機能には、デバイスドライバを定義するための機能、すなわちT-Kernelにデバイスドライバを登録するための機能と、登録済みのデバイスドライバをアプリケーションやミドルウェアから利用するための機能が含まれる。

デバイスドライバの登録は、システム起動時の初期化処理の中で行う場合が多いが、システムの通常の動作中に動的に行うことも可能である。デバイスドライバの登録は tk_def_dev() によって行われ、この拡張SVCのパラメータの一つであるデバイス登録情報(ddev)の中で、デバイスドライバの実際の処理を行うプログラムの関数(ドライバ処理関数)群を指定する。この中には、デバイスのオープン時に呼び出されるオープン関数(openfn)、読込み時や書込みの処理開始時に呼び出される処理開始関数(execfn)、読込み時や書込みの処理の完了を待つ完了待ち関数(waitfn)などがあり、これらのドライバ処理関数の中で、実際のデバイスの操作やデバイスとの入出力の処理を行う。

これらのドライバ処理関数は、準タスク部として保護レベル0で実行されるため、ハードウェアに直接アクセスすることも可能である。デバイスとの入出力などの処理は、これらのドライバ処理関数の中で直接行う場合もあるし、これらのドライバ処理関数の中から出された要求に応じて動作する別タスク等の中で行う場合もある。これらのドライバ処理関数が呼び出される際のパラメータ等の仕様は、デバイスドライバインタフェースとして規定される。デバイスドライバインタフェースは、デバイスドライバとT-Kernelのデバイス管理機能との間のインタフェースである。

デバイスドライバのプログラムは、その保守性や移植性を高めるために、インタフェース層、論理層、物理層の3段階の階層分けを意識して実装することが推奨される。インタフェース層はT-Kernelのデバイス管理機能とデバイスドライバとの間のインタフェースを処理する部分、論理層はデバイスの種類に応じて共通の処理を行う部分、物理層は実際のハードウェアや制御チップに依存した処理を行う部分である。ただし、インタフェース層、論理層、物理層の間のインタフェースについて、T-Kernelで仕様を規定しているわけではなく、実際の階層分けについては、個々のデバイスドライバにおいて最適な実装を行うことができる。インタフェース層については、個々のデバイスに依存しない共通の処理が多いため、インタフェース層の処理を行うプログラムがライブラリとして提供される場合がある。

一方、登録済みのデバイスドライバをアプリケーションやミドルウェアから利用するために、オープン(tk_opn_dev())、クローズ(tk_cls_dev())、読込み(tk_rea_dev())、書込み(tk_wri_dev())などが拡張SVCとして提供されている。これらの拡張SVCの仕様をアプリケーションインタフェースと呼ぶ。たとえば、アプリケーションがデバイスをオープンするために tk_opn_dev() を実行した場合、T-Kernelは、対応するデバイスドライバのオープン関数(openfn)を呼び出して、デバイスのオープンの処理を依頼する。

T-Kernelのデバイス管理機能の位置付けと構成を[図 5.1.「デバイス管理機能」]に示す。

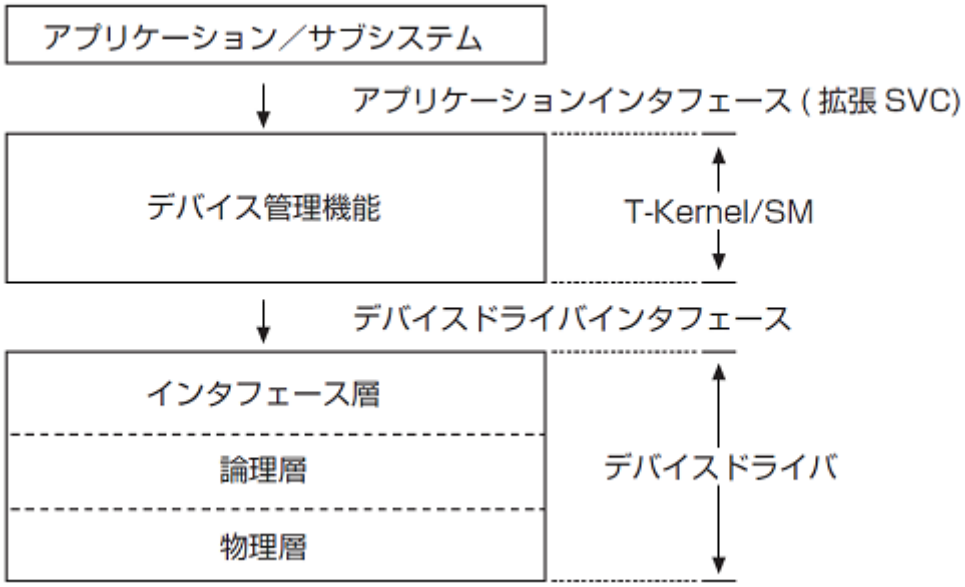

図 5.1: デバイス管理機能

補足事項
デバイスドライバは、T-Kernel本体とは独立して実装され、T-Kernelの機能に対して拡張や追加を行うシステムプログラムであるという点で、サブシステムと共通の特徴を持つ。また、プログラムを共有空間にロードして実行する点や、保護レベル0で動作し、ハードウェアへのアクセスが可能である点についても、両者とも同じである。一方、両者の相違点としては、デバイスドライバを呼び出す際のAPIがオープン／クローズ、リード／ライト型に固定されているのに対して、サブシステムを呼び出す際のAPIは自由に定義できる。また、サブシステムの場合はリソースの管理を行う機能があるが、デバイスドライバの場合はそのような機能がない。
デバイス管理機能によって管理されるT-Kernelのデバイスドライバは、物理的な意味でのデバイスやハードウェアに対するドライバを想定した機能であるが、物理的な意味でのデバイスやハードウェアを扱うことが必須ではない。一方、デバイスを操作するためのシステムプログラムであっても、オープン／クローズ、リード／ライト型のAPIが馴染まない場合など、デバイスドライバではなくサブシステムとして実装されることがある。

5.3.1 デバイスドライバに関する共通事項

5.3.1.1 デバイスの基本概念

デバイスには、物理的なハードウェアとしてのデバイスを表す物理デバイスのほかに、ソフトウェアから見たデバイスの単位である論理デバイスがある。

多くのデバイスにおいて両者は一致するが、ハードディスクやストレージ系のデバイス(SDカード, USBストレージなど)の中に区画(パーティション)を作った場合には、デバイス全体が物理デバイス、1つの区画が論理デバイスとなる。

同種類の物理デバイスは「ユニット」により区別され、1つの物理デバイス内の論理デバイスは「サブユニット」により区別される。たとえば、1台目のハードディスクと2台目のハードディスクを区別する情報が「ユニット」であり、1台目のハードディスク内の1つ目の区画と2つ目の区画を区別する情報が「サブユニット」である。

デバイス管理機能で使用するデータの定義は、以下の通りである。

5.3.1.1.1 デバイス名 (UB*型)

デバイス名は、デバイス毎につけられた最大8文字の文字列であり、次の要素により構成される。

```
#define L_DEVNM         8       /* デバイス名の長さ */
```

種別
: デバイスの種別を示す名前
: 使用可能な文字は a～z A～Z

ユニット
: 物理的なデバイスを示す英字1文字
: a～zでユニットごとにaから順に割り当てる

サブユニット
: 論理的なデバイスを示す数字最大3文字
: 0～254でサブユニットごとに0から順に割り当てる

デバイス名は、種別+ユニット+サブユニットの形式で表すが、ユニット、サブユニットはデバイスによっては存在しない場合もあり、その場合はそれぞれのフィールドは存在しない。

サブユニットは、ハードディスクなどの区画(パーティション)を区別するために使用するが、それ以外のデバイスにおいても、1つの物理デバイスの中に複数の論理的なデバイスを設けたい場合に使用できる。

種別+ユニットの形式を物理デバイス名と呼ぶ。また、種別+ユニット+サブユニットを論理デバイス名と呼ぶ。サブユニットがない場合は、物理デバイス名と論理デバイス名は同じになる。単にデバイス名と言ったときは、論理デバイス名を指す。

例 5.1 デバイス名の例

デバイス名	対象デバイス
hda	ハードディスク(ディスク全体)
hda0	ハードディスク(先頭の区画)
fda	フロッピーディスク
rsa	シリアルポート
kbpd	キーボード／ポインティングデバイス

5.3.1.1.2 デバイスID (ID型)

デバイス(デバイスドライバ)を T-Kernel/SMに登録することにより、デバイス(物理デバイス名)に対してデバイスID(>0)が割り当てられる。デバイスIDは物理デバイスごとに割り当てられ、論理デバイスのデバイスIDは物理デバイスに割り当てられたデバイスIDにサブユニット番号+1(1～255)を加えたものとなる。

devid: 登録時に割り当てられたデバイスID

devid	物理デバイス
devid + n+1	n 番目のサブユニット(論理デバイス)

例 5.2 デバイスIDの例

デバイス名	デバイスID	説明
hda	devid	ハードディスク(ディスク全体)
hda0	devid + 1	ハードディスクの先頭の区画
hda1	devid + 2	ハードディスクの2番目の区画

5.3.1.1.3 デバイス属性 (ATR型)

各デバイスの特徴を表すとともに、デバイスの種類分けを行うため、デバイス属性を定義する。デバイス属性は、デバイスドライバを登録する際に指定する。

デバイス属性の指定方法は、次の通りである。

```
IIII IIII IIII IIII PRxx xxxx KKKK KKKK
```

上位16ビットは、デバイス依存属性で、デバイスごとに定義される。下位16ビットは、標準属性で、下記のように定義される。

```
#define TD_PROTECT      0x8000  /* P: 書込み禁止 */
#define TD_REMOVABLE    0x4000  /* R: メディアの取り外し可能 */

#define TD_DEVKIND      0x00ff  /* K: デバイス/メディア種別 */
#define TD_DEVTYPE      0x00f0  /*    デバイスタイプ */

                                /* デバイスタイプ */
#define TDK_UNDEF       0x0000  /* 未定義/不明 */
#define TDK_DISK        0x0010  /* ディスクデバイス */
```

上記のデバイスタイプのうち、ディスクタイプ(TDK_DISK)であるかどうかは、サスペンド時の処理手順に影響する。詳細は tk_sus_dev および「5.3.3.5節「ディスクデバイスの特殊性」」を参照のこと。

T-Kernelの範囲では、ディスクタイプ以外のデバイスタイプは定義されておらず、ディスクタイプ以外のデバイスタイプを定義しても、T-Kernelの動作には影響しない。未定義のデバイスは、未定義 TDK_UNDEF とする。

ディスクデバイスの場合には、さらに、ディスク種別が定義される。代表的なディスク種別は以下の通りである。これ以外のディスク種別については、デバイスドライバ関連の仕様書、あるいは「7.1節「参考とすべきデバイスドライバ関連の仕様」」の「7.1.1節「デバイス属性のディスク種別」」を参照のこと。

```
/* ディスク種別 */
#define TDK_DISK_UNDEF  0x0010  /* その他のディスク */
#define TDK_DISK_HD     0x0015  /* ハードディスク */
#define TDK_DISK_CDROM  0x0016  /* CD-ROM */
```

ディスク種別の定義も、T-Kernelの動作には影響しない。これらの定義は、デバイスドライバやアプリケーションにおいて、必要な場合にのみ利用する。たとえば、アプリケーションがデバイスやメディアの種類によって処理内容を変える必要がある場合に、ディスク種別の情報を利用する。特に明確な区別をする必要がないデバイスやメディアに対しては、必ずしもディスク種別を割り当てる必要はない。

5.3.1.1.4 デバイスディスクリプタ (ID型)

デバイスディスクリプタは、デバイスをアクセスするための識別子である。

デバイスディスクリプタは、デバイスをオープンしたときに、T-Kernel/SMによって正の値(>0)が割り当てられる。

デバイスディスクリプタは、そのデバイスをオープンしたタスクと同じリソースグループに所属する。デバイスディスクリプタを使用した操作は、そのデバイスディスクリプタと同じリソースグループに所属するタスクからしかできない。異なるリソースグループに所属するタスクからの要求はエラーとなる(E_OACV)。

5.3.1.1.5 リクエストID (ID型)

デバイスに対する入出力を要求したときには、その要求の識別子として、リクエストID(>0)が割り当てられる。このリクエストIDにより入出力の完了を待つことができる。

5.3.1.1.6 データ番号 (W型, D型)

デバイスから入出力するデータはデータ番号により指定する。データは固有データと属性データに大別される。

固有データ:データ番号≧0
　　デバイス固有のデータで、データ番号はデバイスごとに定義される。

例 5.3 固有データの例

デバイス	データ番号
ディスク	データ番号＝物理ブロック番号
シリアル回線	データ番号は0のみ使用

属性データ:データ番号＜0
　　ドライバやデバイスの状態の取得や設定、特殊機能などを指定する。

　　データ番号のいくつかは共通で定義されているが、デバイス独自にも定義できる。詳細は「5.3.1.2節「属性データ」」を参照。

5.3.1.2 属性データ

属性データは大きく次の3つに分類される。

共通属性
　すべてのデバイス(デバイスドライバ)に共通に定義する属性。

デバイス種別属性
　同じ種類に分類されるデバイス(デバイスドライバ)に共通に定義する属性。

デバイス個別属性
　各デバイス(デバイスドライバ)ごとに独自に定義される属性。

デバイス種別属性およびデバイス個別属性については、デバイスドライバ関連の仕様書を参照のこと。ここでは、共通属性のみ定義する。

共通属性の属性データ番号は -1～-99 の範囲となる。共通属性のデータ番号はすべてのデバイスで共通となるが、すべてのデバイスが必ずしもすべての共通属性に対応しているとは限らない。対応していないデータ番号を指定されたときは、エラー E_PAR とする。

共通属性の定義は、以下の通りである。

```
#define TDN_EVENT        (-1)    /* RW:事象通知用メッセージバッファID */
#define TDN_DISKINFO     (-2)    /* R-:ディスク情報 */
#define TDN_DISPSPEC     (-3)    /* R-:表示デバイス仕様 */
#define TDN_PCMCIAINFO   (-4)    /* R-:PCカード情報 */
#define TDN_DISKINFO_D   (-5)    /* R-:ディスク情報(64ビットデバイス) */
```

RW: 読込み(tk_rea_dev)/書込み(tk_wri_dev)可能
R-: 読込み(tk_rea_dev)のみ可能

TDN_EVENT
　事象通知用メッセージバッファID

　データ形式　　ID

　デバイス事象通知用のメッセージバッファのIDである。

　デバイスドライバの起動時には、tk_def_dev によってデバイスの登録を行うが、このAPIのリターンパラメータとしてシステムデフォルトの事象通知用メッセージバッファID(evtmbfid)が返されるので、その値をデバイスドライバ内で保持し、本属性データの初期値とする。

　0が設定されている場合は、デバイス事象通知を行わない。デバイス事象通知については、「5.3.3.3節「デバイス事象通知」」を参照。

TDN_DISKINFO
　32ビットデバイス・ディスク情報

　データ形式　　DiskInfo

```
typedef enum {
        DiskFmt_STD     = 0,            /* 標準(HDなど) */
        DiskFmt_2HD     = 2,            /* 2HD 1.44MB */
        DiskFmt_CDROM   = 4             /* CD-ROM 640MB */
} DiskFormat;
```

```
typedef struct {
        DiskFormat format;              /* フォーマット形式 */
        UW      protect:1;              /* プロテクト状態 */
        UW      removable:1;            /* 取り外し可否 */
```

```
        UW      rsv:30;                 /* 予約（常に0）*/
        W       blocksize;              /* ブロックバイト数 */
        W       blockcont;              /* 総ブロック数 */
} DiskInfo;
```

上記の記載以外のDiskFormatの定義については、デバイスドライバ関連の仕様書、あるいは「7.1節「参考とすべきデバイスドライバ関連の仕様」」の「7.1.2節「デバイスの属性データ」」を参照のこと。

TDN_DISPSPEC
表示デバイス仕様

データ形式　　DEV_SPEC

DEV_SPECの定義については、デバイスドライバ関連の仕様書、あるいは「7.1節「参考とすべきデバイスドライバ関連の仕様」」の「7.1.2節「デバイスの属性データ」」を参照のこと。

TDN_DISKINFO_D
64ビットデバイス・ディスク情報

データ形式　　DiskInfo_D

```
typedef struct diskinfo_d {
        DiskFormat format;      /* フォーマット形式 */
        BOOL    protect:1;      /* プロテクト状態 */
        BOOL    removable:1;    /* 取り外し可否 */
        UW      rsv:30;         /* 予約（0）*/
        W       blocksize;      /* ブロックバイト数 */
        D       blockcont_d;    /* 64ビットの総ブロック数 */
} DiskInfo_D;
```

DiskInfo_DとDiskInfoとの差異は、blockcont あるいは blockcont_d の部分の名称およびデータタイプのみである。

T-Kernel/SMは、DiskInfoとDiskInfo_Dの間の変換は行わない。TDN_DISKINFO も TDN_DISKINFO_D も、デバイスドライバへの要求をそのまま渡すのみである。

ディスクドライバは、TDN_DISKINFO または TDN_DISKINFO_D のいずれか一方、または両方に対応する必要がある。T-Kernel 1.0との互換性のため、TDN_DISKINFO は可能な限り対応することを推奨する。

ディスク全体の総ブロック数がWに収まらない場合でも、個々の区画のブロック数はWに収まることがある。そのような場合、Wに収まる区画は TDN_DISKINFO に対応し、Wに収まらないもののみ TDN_DISKINFO はエラー(E_PAR)とするような実装が望ましい。また、ブロック数がWに収まる場合であっても、TDN_DISKINFO_D に対応することは望ましいことである。

TDN_DISKINFO_D への対応とデバイスドライバ属性の TDA_DEV_D には直接の依存関係がない。TDN_DISKINFO_D に対応しているからといって TDA_DEV_D 属性のデバイスドライバであるとは限らないし、TDA_DEV_D 属性のデバイスドライバであっても TDN_DISKINFO_D に対応しているとは限らない。

なお、上記の共通属性の定義は、T-Kernelではなくデバイスドライバの仕様の一部を定めるものであり、T-Kernelの動作には直接影響しない。また、各デバイスドライバは、共通属性に定義されたすべての機能を実装する必要はない。しかし、共通属性の定義はすべてのデバイスドライバに対して有効であり、各デバイスドライバの仕様は、これらの定義と矛盾のないように定める必要がある。

T-Kernel 1.0との差異
64ビットデバイスへの対応のため、TDN_DISKINFO_D の属性データを追加した。

5.3.2 デバイスの入出力操作

登録済みのデバイスドライバをアプリケーションやミドルウェアから利用するためには、アプリケーションインタフェースを使用する。アプリケーションインタフェースには下記の関数があり、拡張SVCにより呼び出す。これらの関数は、タスク独立部およびディスパッチ禁止中、割込み禁止中に呼び出すことはできない(E_CTX)。

```
ID  tk_opn_dev( CONST UB *devnm, UINT omode )
ER  tk_cls_dev( ID dd, UINT option )
ID  tk_rea_dev( ID dd, W start, void *buf, W size, TMO tmout )
ID  tk_rea_dev_du( ID dd, D start_d, void *buf, W size, TMO_U tmout_u )
ER  tk_srea_dev( ID dd, W start, void *buf, W size, W *asize )
ER  tk_srea_dev_d( ID dd, D start_d, void *buf, W size, W *asize )
ID  tk_wri_dev( ID dd, W start, CONST void *buf, W size, TMO tmout )
ID  tk_wri_dev_du( ID dd, D start_d, CONST void *buf, W size, TMO_U tmout_u )
ER  tk_swri_dev( ID dd, W start, CONST void *buf, W size, W *asize )
ER  tk_swri_dev_d( ID dd, D start_d, CONST void *buf, W size, W *asize )
ID  tk_wai_dev( ID dd, ID reqid, W *asize, ER *ioer, TMO tmout )
ID  tk_wai_dev_u( ID dd, ID reqid, W *asize, ER *ioer, TMO_U tmout_u )
INT tk_sus_dev( UINT mode )
ID  tk_get_dev( ID devid, UB *devnm )
ID  tk_ref_dev( CONST UB *devnm, T_RDEV *rdev )
ID  tk_oref_dev( ID dd, T_RDEV *rdev )
INT tk_lst_dev( T_LDEV *ldev, INT start, INT ndev )
INT tk_evt_dev( ID devid, INT evttyp, void *evtinf )
```

5.3.2.1 tk_opn_dev - デバイスのオープン

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID dd = tk_opn_dev (CONST UB *devnm , UINT omode);

パラメータ

CONST UB*	devnm	Device Name	デバイス名
UINT	omode	Open Mode	オープンモード

リターンパラメータ

ID	dd	Device Descriptor	デバイスディスクリプタ
	または	Error Code	エラーコード

エラーコード

E_BUSY	デバイスは使用中(排他オープン中)
E_NOEXS	デバイスは存在しない
E_LIMIT	オープン可能な最大数を超えた
その他	デバイスドライバから返されたエラーコード

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

devnm で指定したデバイスを omode で指定したモードでオープンし、デバイスへのアクセスを準備する。戻値に、デバイスディスクリプタを返す。

```
omode := (TD_READ || TD_WRITE || TD_UPDATE) | [TD_EXCL || TD_WEXCL || TD_REXCL]
         | [TD_NOLOCK]
```

```
#define TD_READ         0x0001          /* 読込み専用 */
#define TD_WRITE        0x0002          /* 書込み専用 */
#define TD_UPDATE       0x0003          /* 読込みおよび書込み */
#define TD_EXCL         0x0100          /* 排他 */
#define TD_WEXCL        0x0200          /* 排他書込み */
#define TD_REXCL        0x0400          /* 排他読込み */
#define TD_NOLOCK       0x1000          /* ロック(常駐化)不要 */
```

TD_READ
　読込み専用

TD_WRITE
　書込み専用

TD_UPDATE
: 読込みおよび書込み

アクセスモードを指定する。

TD_READ の場合は、tk_wri_dev() は使用できない。

TD_WRITE の場合は、tk_rea_dev() は使用できない。

TD_EXCL
: 排他

TD_WEXCL
: 排他書込み

TD_REXCL
: 排他読込み

排他モードを指定する。

TD_EXCL は、一切の同時オープンを禁止する。

TD_WEXCL は、書込みモード(TD_WRITE または TD_UPDATE)による同時オープンを禁止する。

TD_REXCL は、読込みモード(TD_READ または TD_UPDATE)による同時オープンを禁止する。

現在オープンモード		同時オープンモード											
		排他指定なし			TD_WEXCL			TD_REXCL			TD_EXCL		
		R	U	W	R	U	W	R	U	W	R	U	W
排他指定なし	R	○	○	○	○	○	○	×	×	×	×	×	×
	U	○	○	○	×	×	×	×	×	×	×	×	×
	W	○	○	○	×	×	×	○	○	○	×	×	×
TD_WEXCL	R	○	×	×	○	×	×	×	×	×	×	×	×
	U	○	×	×	×	×	×	×	×	×	×	×	×
	W	○	×	×	×	×	×	○	×	×	×	×	×
TD_REXCL	R	×	×	○	×	×	○	×	×	×	×	×	×
	U	×	×	○	×	×	×	×	×	×	×	×	×
	W	×	×	○	×	×	×	×	×	○	×	×	×
TD_EXCL	R	×	×	×	×	×	×	×	×	×	×	×	×
	U	×	×	×	×	×	×	×	×	×	×	×	×
	W	×	×	×	×	×	×	×	×	×	×	×	×

表 5.1: 同じデバイスを同時にオープンしようとしたときの可否

R = TD_READ
W = TD_WRITE
U = TD_UPDATE
○ = オープン可
× = オープン不可(E_BUSY)

TD_NOLOCK
: ロック(常駐化)不要

入出力時(tk_rea_dev/tk_wri_dev)に指定したメモリ(buf)の領域は、呼出側でロック(常駐化)済みで、デバイスドライバ側ではロック不要であることを指示する。この場合、デバイスドライバではロックを行わない(行ってはいけない)。この指定は、仮想記憶システムにおいてページイン／ページアウトのためのディスクアクセスを行う場合などに使用する。一般的には指定する必要はない。

デバイスディスクリプタは、オープンしたタスクのリソースグループに所属する。

なお、物理デバイスをオープンした場合、その物理デバイスに属する論理デバイスをすべて同じモードでオープンしたのと同様に扱い、排他オープンの処理が行われる。

5.3.2.2 tk_cls_dev - デバイスのクローズ

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_cls_dev (ID dd , UINT option);

パラメータ

ID	dd	Device Descriptor	デバイスディスクリプタ
UINT	option	Close Option	クローズオプション

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_ID	dd が不正またはオープンされていない
その他	デバイスドライバから返されたエラーコード

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

dd のデバイスディスクリプタをクローズする。処理中の要求があった場合は、その処理を中止させてクローズする。

option := [TD_EJECT]

```
#define TD_EJECT        0x0001          /* メディア排出 */
```

TD_EJECT
: メディア排出

同一デバイスが他からオープンされていなければ、メディアを排出する。ただし、メディアの排出ができないデバイスでは無視される。

サブシステムのクリーンアップ処理(tk_cln_ssy)により、対象リソースグループに属するデバイスディスクリプタはすべてクローズされる。

5.3.2.3 tk_rea_dev - デバイスの読込み開始

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID reqid = tk_rea_dev (ID dd , W start , void *buf , W size , TMO tmout);

パラメータ

ID	dd	Device Descriptor	デバイスディスクリプタ
W	start	Start Location	読込み開始位置(≧0:固有データ, <0:属性データ)
void*	buf	Buffer	読み込んだデータを格納するバッファ
W	size	Read Size	読み込むサイズ
TMO	tmout	Timeout	要求受付待ちタイムアウト時間(ミリ秒)

リターンパラメータ

ID	reqid	Request ID	リクエストID
	または	Error Code	エラーコード

エラーコード

E_ID	dd が不正またはオープンされていない
E_OACV	オープンモードが不正(読込みが許可されていない)
E_LIMIT	最大リクエスト数を超えた
E_TMOUT	他の要求を処理中で受け付けられない
E_ABORT	処理中止
その他	デバイスドライバから返されたエラーコード

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

デバイスから固有データまたは属性データの読込みを開始する。読込みを開始するのみで、読込み完了を待たずに呼出元へ戻る。読込みが完了するまで、buf を保持しなければならない。読込み完了は tk_wai_dev() により待つ。読込み開始のための処理にかかる時間はデバイスドライバにより異なる。必ずしも即座に戻るとは限らない。

固有データの場合、start および size の単位はデバイスごとに決められる。属性データの場合、start は属性データ番号、size はバイト数となり、start のデータ番号の属性データを読み込む。通常、size は読み込む属性データのサイズ以上でなければならない。複数の属性データを一度に読み込むことはできない。size＝0を指定した場合、実際の読込みは行わず、現時点で読込み可能なサイズを調べる。

読込みまたは書込みの動作中である場合、新たな要求を受け付けられるか否かはデバイスドライバによる。新たな要求を受け付けられない状態の場合、要求受付待ちとなる。要求受付待ちのタイムアウト時間を tmout に指定する。tmout には TMO_POL または TMO_FEVR を指定することもできる。なお、タイムアウトするのは要求受付までである。要求が受け付けられた後にはタイムアウトしない。

TDA_DEV_D や TDA_TMO_U 属性のデバイスドライバに対して、本拡張SVCを使用しても構わない。その場合、T-Kernel/SMの中でパラメータを適切に変換する。たとえば、デバイスドライバの属性が TDA_TMO_U の場合、本拡張SVCの tmout で指定さ

れたミリ秒単位のタイムアウト時間が、マイクロ秒単位の時間に換算された上で、TDA_TMO_U 属性のデバイスドライバに渡される。

T-Kernel 1.0との差異

start, size のデータタイプをINTからWに変更した。これは、T-Kernel 2.0で64ビットを導入した機能(時間管理、デバイス管理)と関連の深いパラメータについて、ビット数を固定した方が分かりやすく誤解がないという理由によるものである。MSECやTMOをINTからWに、RELTIMをUINTからUWに変更したのも、μT-Kernelとの関係に加えて、同様の趣旨がある。

5.3.2.4 tk_rea_dev_du - デバイスの読込み開始(64ビットマイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID reqid = tk_rea_dev_du (ID dd , D start_d , void *buf , W size , TMO_U tmout_u);

パラメータ

ID	dd	Device Descriptor	デバイスディスクリプタ
D	start_d	Start Location	読込み開始位置(64ビット, ≧0:固有データ, <0:属性データ)
void*	buf	Buffer	読み込んだデータを格納するバッファ
W	size	Read Size	読み込むサイズ
TMO_U	tmout_u	Timeout	要求受付待ちタイムアウト時間(マイクロ秒)

リターンパラメータ

ID	reqid	Request ID	リクエストID
	または	Error Code	エラーコード

エラーコード

E_ID	dd が不正またはオープンされていない
E_OACV	オープンモードが不正(読込みが許可されていない)
E_LIMIT	最大リクエスト数を超えた
E_TMOUT	他の要求を処理中で受け付けられない
E_ABORT	処理中止
その他	デバイスドライバから返されたエラーコード

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_rea_dev のパラメータである start および tmout を、64ビットの start_d および64ビットマイクロ秒単位の tmout_u とした拡張SVCである。

パラメータが start_d および tmout_u となった点を除き、本拡張SVCの仕様は tk_rea_dev と同じである。詳細は tk_rea_dev の説明を参照のこと。

補足事項

対応するデバイスドライバが TDA_DEV_D 属性のデバイスドライバでない場合に、開始位置 start_d としてWに入りきらない値を指定すると、E_PAR のエラーになる。

また、対応するデバイスドライバが TDA_TMO_U 属性のデバイスドライバでない(マイクロ秒単位に対応していない)場合には、デバイスドライバがマイクロ秒単位のタイムアウトを扱うことができない。この場合は、本拡張SVCの tmout_u で指定されたマイクロ秒単位のタイムアウト時間が、ミリ秒単位の時間に切り上げられた上で、デバイスドライバに渡される。

このように、T-Kernel/SMの中でパラメータの適切な変換を行うので、アプリケーション側では、デバイスドライバの属性が `TDA_DEV_D` かどうか、すなわちデバイスドライバが64ビット対応かどうかに関して意識する必要はない。

T-Kernel 1.0との差異

T-Kernel 2.0で追加された拡張SVCである。

なお、tk_rea_dev_du と tk_wri_dev_du は、開始位置が64ビット、タイムアウトが64ビットマイクロ秒のため、_uと_dの両方の意味を含む。

5.3.2.5 tk_srea_dev - デバイスの同期読込み

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_srea_dev (ID dd , W start , void *buf , W size , W *asize);

パラメータ

ID	dd	Device Descriptor	デバイスディスクリプタ
W	start	Start Location	読込み開始位置(≧0:固有データ, <0:属性データ)
void*	buf	Buffer	読み込んだデータを格納するバッファ
W	size	Read Size	読み込むサイズ
W*	asize	Actual Size	読込みサイズを返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード
W	asize	Actual Size	実際の読込みサイズ

エラーコード

E_ID	dd が不正またはオープンされていない
E_OACV	オープンモードが不正(読込みが許可されていない)
E_LIMIT	最大リクエスト数を超えた
E_ABORT	処理中止
その他	デバイスドライバから返されたエラーコード

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

同期読込み。以下と等価である。

```
ER tk_srea_dev( ID dd, W start, void *buf, W size, W *asize )
{
        ER      er, ioer;

        er = tk_rea_dev(dd, start, buf, size, TMO_FEVR);
        if ( er > 0 ) {
                er = tk_wai_dev(dd, er, asize, &ioer, TMO_FEVR);
                if ( er > 0 ) er = ioer;
        }

        return er;
}
```

TDA_DEV_D 属性のデバイスドライバに対して、本拡張SVCを使用しても構わない。その場合、T-Kernel/SMの中でパラメータを適切に変換する。

T-Kernel 1.0との差異

start, size のデータタイプをINTからWに、asize のデータタイプをINT *からW *に変更した。

5.3.2.6 tk_srea_dev_d - デバイスの同期読込み(64ビット)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_srea_dev_d (ID dd , D start_d , void *buf , W size , W *asize);

パラメータ

ID	dd	Device Descriptor	デバイスディスクリプタ
D	start_d	Start Location	読込み開始位置(64ビット, ≧0:固有データ, <0:属性データ)
void*	buf	Buffer	読み込んだデータを格納するバッファ
W	size	Read Size	読み込むサイズ
W*	asize	Actual Size	読込みサイズを返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード
W	asize	Actual Size	実際の読込みサイズ

エラーコード

E_ID	dd が不正またはオープンされていない
E_OACV	オープンモードが不正(読込みが許可されていない)
E_LIMIT	最大リクエスト数を超えた
E_ABORT	処理中止
その他	デバイスドライバから返されたエラーコード

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_srea_dev のパラメータである start を、64ビットの start_d とした拡張SVCである。

パラメータが start_d となった点を除き、本拡張SVCの仕様は tk_srea_dev と同じである。詳細は tk_srea_dev の説明を参照のこと。

補足事項

対応するデバイスドライバが TDA_DEV_D 属性のデバイスドライバでない場合に、開始位置 start_d としてWに入りきらない値を指定すると、E_PAR のエラーになる。

このように、T-Kernel/SMの中でパラメータの適切な変換を行うので、アプリケーション側では、デバイスドライバの属性が TDA_DEV_D かどうか、すなわちデバイスドライバが64ビット対応かどうかに関して意識する必要はない。

T-Kernel 1.0との差異

T-Kernel 2.0で追加された拡張SVCである。

5.3.2.7 tk_wri_dev - デバイスの書込み開始

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID reqid = tk_wri_dev (ID dd , W start , CONST void *buf , W size , TMO tmout);

パラメータ

ID	dd	Device Descriptor	デバイスディスクリプタ
W	start	Start Location	書込み開始位置(≧0:固有データ, <0:属性データ)
CONST void*	buf	Buffer	書き込むデータを格納したバッファ
W	size	Write Size	書き込むサイズ
TMO	tmout	Timeout	要求受付待ちタイムアウト時間(ミリ秒)

リターンパラメータ

ID	reqid	Request ID	リクエストID
	または	Error Code	エラーコード

エラーコード

E_ID	dd が不正またはオープンされていない
E_OACV	オープンモードが不正(書込みが許可されていない)
E_RONLY	書き込めないデバイス
E_LIMIT	最大リクエスト数を超えた
E_TMOUT	他の要求を処理中で受け付けられない
E_ABORT	処理中止
その他	デバイスドライバから返されたエラーコード

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

デバイスへ固有データまたは属性データの書込みを開始する。書込みを開始するのみで、書込み完了を待たずに呼出元へ戻る。書込みが完了するまで、buf を保持しなければならない。書込み完了は tk_wai_dev() により待つ。書込み開始のための処理にかかる時間はデバイスドライバにより異なる。必ずしも即座に戻るとは限らない。

固有データの場合、start および size の単位はデバイスごとに決められる。属性データの場合、start は属性データ番号、size はバイト数となり、start のデータ番号の属性データに書き込む。通常、size は書き込む属性データのサイズと同じでなければならない。複数の属性データを一度に書き込むことはできない。size＝0を指定した場合、実際の書込みは行わず、現時点で書込み可能なサイズを調べる。

読込みまたは書込みの動作中である場合、新たな要求を受け付けられるか否かはデバイスドライバによる。新たな要求を受け付けられない状態の場合、要求受付待ちとなる。要求受付待ちのタイムアウト時間を tmout に指定する。tmout には TMO_POL または TMO_FEVR を指定することもできる。なお、タイムアウトするのは要求受付までである。要求が受け付けられた後にはタイムアウトしない。

TDA_DEV_D や TDA_TMO_U 属性のデバイスドライバに対して、本拡張SVCを使用しても構わない。その場合、T-Kernel/SMの中でパラメータを適切に変換する。たとえば、デバイスドライバの属性が TDA_TMO_U の場合、本拡張SVCの tmout で指定されたミリ秒単位のタイムアウト時間が、マイクロ秒単位の時間に換算された上で、TDA_TMO_U 属性のデバイスドライバに渡される。

T-Kernel 1.0との差異

start, size のデータタイプをINTからWに変更した。

5.3.2.8 tk_wri_dev_du - デバイスの書込み開始(64ビットマイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID reqid = tk_wri_dev_du (ID dd , D start_d , CONST void *buf , W size , TMO_U tmout_u);

パラメータ

ID	dd	Device Descriptor	デバイスディスクリプタ
D	start_d	Start Location	書込み開始位置(64ビット, ≧0:固有データ, <0:属性データ)
CONST void*	buf	Buffer	書き込むデータを格納したバッファ
W	size	Write Size	書き込むサイズ
TMO_U	tmout_u	Timeout	要求受付待ちタイムアウト時間(マイクロ秒)

リターンパラメータ

ID	reqid	Request ID	リクエストID
	または	Error Code	エラーコード

エラーコード

E_ID	dd が不正またはオープンされていない
E_OACV	オープンモードが不正(書込みが許可されていない)
E_RONLY	書き込めないデバイス
E_LIMIT	最大リクエスト数を超えた
E_TMOUT	他の要求を処理中で受け付けられない
E_ABORT	処理中止
その他	デバイスドライバから返されたエラーコード

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_wri_dev のパラメータである start および tmout を、64ビットの start_d および64ビットマイクロ秒単位の tmout_u とした拡張SVCである。

パラメータが start_d および tmout_u となった点を除き、本拡張SVCの仕様は tk_wri_dev と同じである。詳細は tk_wri_dev の説明を参照のこと。

補足事項

対応するデバイスドライバが TDA_DEV_D 属性のデバイスドライバでない場合に、開始位置 start_d としてWに入りきらない値を指定すると、E_PAR のエラーになる。

また、対応するデバイスドライバが TDA_TMO_U 属性のデバイスドライバでない(マイクロ秒単位に対応していない)場合には、デバイスドライバがマイクロ秒単位のタイムアウトを扱うことができない。この場合は、本拡張SVCの tmout_u で指定されたマイクロ秒単位のタイムアウト時間が、ミリ秒単位の時間に切り上げられた上で、デバイスドライバに渡される。

このように、T-Kernel/SMの中でパラメータの適切な変換を行うので、アプリケーション側では、デバイスドライバの属性が TDA_DEV_D かどうか、すなわちデバイスドライバが64ビット対応かどうかに関して意識する必要はない。

T-Kernel 1.0との差異

T-Kernel 2.0で追加された拡張SVCである。

なお、tk_rea_dev_du と tk_wri_dev_du は、開始位置が64ビット、タイムアウトが64ビットマイクロ秒のため、_uと_dの両方の意味を含む。

5.3.2.9 tk_swri_dev - デバイスの同期書込み

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_swri_dev (ID dd , W start , CONST void *buf , W size , W *asize);

パラメータ

ID	dd	Device Descriptor	デバイスディスクリプタ
W	start	Start Location	書込み開始位置(≧0:固有データ,＜0:属性データ)
CONST void*	buf	Buffer	書き込むデータを格納したバッファ
W	size	Write Size	書き込むサイズ
W*	asize	Actual Size	書込みサイズを返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード
W	asize	Actual Size	実際の書込みサイズ

エラーコード

E_ID	dd が不正またはオープンされていない
E_OACV	オープンモードが不正(書込みが許可されていない)
E_RONLY	書き込めないデバイス
E_LIMIT	最大リクエスト数を超えた
E_ABORT	処理中止
その他	デバイスドライバから返されたエラーコード

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

同期書込み。以下と等価である。

```
ER tk_swri_dev( ID dd, W start, void *buf, W size, W *asize )
{
        ER      er, ioer;

        er = tk_wri_dev(dd, start, buf, size, TMO_FEVR);
        if ( er > 0 ) {
                er = tk_wai_dev(dd, er, asize, &ioer, TMO_FEVR);
                if ( er > 0 ) er = ioer;
        }

        return er;
}
```

TDA_DEV_D 属性のデバイスドライバに対して、本拡張SVCを使用しても構わない。その場合、T-Kernel/SMの中でパラメータを適切に変換する。

T-Kernel 1.0との差異

start, size のデータタイプをINTからWに、asize のデータタイプをINT *からW *に変更した。

5.3.2.10 tk_swri_dev_d - デバイスの同期書込み(64ビット)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_swri_dev_d (ID dd , D start_d , CONST void *buf , W size , W *asize);

パラメータ

ID	dd	Device Descriptor	デバイスディスクリプタ
D	start_d	Start Location	書込み開始位置(64ビット, ≧0:固有データ, <0:属性データ)
CONST void*	buf	Buffer	書き込むデータを格納したバッファ
W	size	Write Size	書き込むサイズ
W*	asize	Actual Size	書込みサイズを返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード
W	asize	Actual Size	実際の書込みサイズ

エラーコード

E_ID	dd が不正またはオープンされていない
E_OACV	オープンモードが不正(書込みが許可されていない)
E_RONLY	書き込めないデバイス
E_LIMIT	最大リクエスト数を超えた
E_ABORT	処理中止
その他	デバイスドライバから返されたエラーコード

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_swri_dev のパラメータである start を、64ビットの start_d とした拡張SVCである。

パラメータが start_d となった点を除き、本拡張SVCの仕様は tk_swri_dev と同じである。詳細は tk_swri_dev の説明を参照のこと。

補足事項

対応するデバイスドライバが TDA_DEV_D 属性のデバイスドライバでない場合に、開始位置 start_d としてWに入りきらない値を指定すると、E_PAR のエラーになる。

このように、T-Kernel/SMの中でパラメータの適切な変換を行うので、アプリケーション側では、デバイスドライバの属性が TDA_DEV_D かどうか、すなわちデバイスドライバが64ビット対応かどうかに関して意識する必要はない。

T-Kernel 1.0との差異

T-Kernel 2.0で追加された拡張SVCである。

5.3.2.11 tk_wai_dev - デバイスの要求完了待ち

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID creqid = tk_wai_dev (ID dd , ID reqid , W *asize , ER *ioer , TMO tmout);

パラメータ

ID	dd	Device Descriptor	デバイスディスクリプタ
ID	reqid	Request ID	リクエストID
W*	asize	Read/Write Actual Size	読込み/書込みサイズを返す領域へのポインタ
ER*	ioer	I/O Error	入出力エラーを返す領域へのポインタ
TMO	tmout	Timeout	タイムアウト時間(ミリ秒)

リターンパラメータ

ID	creqid	Completed Request ID	完了したリクエストID
	または	Error Code	エラーコード
W	asize	Read/Write Actual Size	実際の読込み/書込みサイズ
ER	ioer	I/O Error	入出力エラー

エラーコード

E_ID	dd が不正またはオープンされていない、reqid が不正または dd に対する要求ではない
E_OBJ	reqid の要求は他のタスクで完了待ちしている
E_NOEXS	処理中の要求はない(reqid＝0の場合のみ)
E_TMOUT	タイムアウト(処理継続中)
E_ABORT	処理中止
その他	デバイスドライバから返されたエラーコード

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

dd に対する reqid の要求の完了を待つ。reqid＝0 の場合は、dd に対する要求の内のいずれかが完了するのを待つ。なお、現時点で処理中の要求のみが完了待ちの対象となる。tk_wai_dev() の呼出後に要求された処理は完了待ちの対象とならない。

複数の要求を同時に処理している場合、その要求の完了の順序は必ずしも要求した順序ではなく、デバイスドライバに依存する。ただし、要求した順序で処理した場合と結果が矛盾しないような順序で処理されることは保証される。例えば、ディスクからの読込みの場合、次のような処理順序の変更が考えられる。

要求順ブロック番号
　　１４３２５

処理順ブロック番号
　　１２３４５

このように順序を入れ替えて処理することにより、シークや回転待ちを減らすことができ、より効率的にディスクアクセスができる。

tmout に完了待ちのタイムアウト時間を指定する。TMO_POL または TMO_FEVR を指定することもできる。タイムアウト(E_TMOUT)した場合は要求された処理を継続中なので、再度 tk_wai_dev() により完了を待つ必要がある。reqid > 0かつ tmout＝TMO_FEVR の場合はタイムアウトすることはなく、必ず処理が完了する。

要求された処理に対して、デバイスドライバから処理結果のエラー(入出力エラーなど)が返った場合、そのエラーコードは、戻値ではなく ioer に格納される。具体的には、tk_wai_dev の処理のために呼び出された完了待ち関数 waitfn の中で、要求パケットT_DEVREQの error に格納されたエラーコードが、処理結果のエラーとして ioer に返される。

一方、戻値には、要求の完了待ちが正しくできなかった場合にエラーを返す。戻値にエラーが返された場合、ioer の内容は無意味である。また、戻値にエラーが返された場合は処理を継続中なので、再度 tk_wai_dev() により完了を待つ必要がある。詳細は「5.3.3.2.4節「waitfn - 完了待ち関数」」を参照のこと。

tk_wai_dev() で完了待ち中にタスク例外が発生すると、reqid の要求を中止して処理を完了させる。中止した処理の結果がどのようになるかは、デバイスドライバに依存する。ただし、reqid＝0の場合は、要求を中止することなくタイムアウトと同様に扱われる。この場合は、E_TMOUT ではなく E_ABORT を返す。

同じリクエストIDに対して、複数のタスクから同時に完了待ちすることはできない。reqid＝0で待っているタスクがあれば、同じ dd に対して他のタスクは完了待ちできない。同様に、reqid > 0で待っているタスクがあれば、他のタスクで reqid＝0の完了待ちはできない。

TDA_TMO_U 属性のデバイスドライバに対して、本拡張SVCを使用しても構わない。その場合、T-Kernel/SMの中でパラメータを適切に変換する。たとえば、デバイスドライバの属性が TDA_TMO_U の場合、本拡張SVCの tmout で指定されたミリ秒単位のタイムアウト時間が、マイクロ秒単位の時間に換算された上で、TDA_TMO_U 属性のデバイスドライバに渡される。

T-Kernel 1.0との差異

asize のデータタイプをINT *からW *に変更した。

5.3.2.12 tk_wai_dev_u - デバイスの要求完了待ち(マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID creqid = tk_wai_dev_u (ID dd , ID reqid , W *asize , ER *ioer , TMO_U tmout_u);

パラメータ

ID	dd	Device Descriptor	デバイスディスクリプタ
ID	reqid	Request ID	リクエストID
W*	asize	Read/Write Actual Size	読込み/書込みサイズを返す領域へのポインタ
ER*	ioer	I/O Error	入出力エラーを返す領域へのポインタ
TMO_U	tmout_u	Timeout	タイムアウト時間(マイクロ秒)

リターンパラメータ

ID	creqid	Completed Request ID	完了したリクエストID
	または	Error Code	エラーコード
W	asize	Read/Write Actual Size	実際の読込み/書込みサイズ
ER	ioer	I/O Error	入出力エラー

エラーコード

E_ID	dd が不正またはオープンされていない、reqid が不正または dd に対する要求ではない
E_OBJ	reqid の要求は他のタスクで完了待ちしている
E_NOEXS	処理中の要求はない(reqid＝0の場合のみ)
E_TMOUT	タイムアウト(処理継続中)
E_ABORT	処理中止
その他	デバイスドライバから返されたエラーコード

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

tk_wai_dev のパラメータである tmout を、64ビットマイクロ秒単位の tmout_u とした拡張SVCである。

パラメータが tmout_u となった点を除き、本拡張SVCの仕様は tk_wai_dev と同じである。詳細は tk_wai_dev の説明を参照のこと。

補足事項

対応するデバイスドライバが TDA_TMO_U 属性のデバイスドライバでない(マイクロ秒単位に対応していない)場合には、デバイスドライバがマイクロ秒単位のタイムアウトを扱うことができない。この場合は、本拡張SVCの tmout_u で指定されたマイクロ秒単位のタイムアウト時間が、ミリ秒単位の時間に切り上げられた上で、デバイスドライバに渡される。

このように、T-Kernel/SMの中でパラメータの適切な変換を行うので、アプリケーション側では、デバイスドライバの属性が TDA_TMO_U かどうか、すなわちデバイスドライバがマイクロ秒単位に対応かどうかに関して意識する必要はない。

T-Kernel 1.0との差異

T-Kernel 2.0で追加された拡張SVCである。

tk_wai_dev_u は、デバイス管理機能の拡張SVCであるが、_dではなく_uが付くことに注意。

5.3.2.13 tk_sus_dev - デバイスのサスペンド

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT dissus = tk_sus_dev (UINT mode);

パラメータ

UINT	mode	Mode	モード

リターンパラメータ

INT	dissus	Suspend Disable Request Count	サスペンド禁止要求カウント数
	または	Error Code	エラーコード

エラーコード

E_BUSY	サスペンド禁止中
E_QOVR	サスペンド禁止要求カウントオーバー

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

mode にしたがって処理を行い、その処理を行ったあとのサスペンド禁止要求カウント数を戻値に返す。

mode := ((TD_SUSPEND | [TD_FORCE]) || TD_DISSUS || TD_ENASUS || TD_CHECK)

```
#define TD_SUSPEND      0x0001      /* サスペンド */
#define TD_DISSUS       0x0002      /* サスペンドを禁止 */
#define TD_ENASUS       0x0003      /* サスペンドを許可 */
#define TD_CHECK        0x0004      /* サスペンド禁止要求カウント取得 */
#define TD_FORCE        0x8000      /* 強制サスペンド指定 */
```

TD_SUSPEND
: サスペンド
サスペンド許可状態であればサスペンドする。
サスペンド禁止状態であれば E_BUSY を返す。

TD_SUSPEND|TD_FORCE
: 強制サスペンド
サスペンド禁止状態であってもサスペンドする。

TD_DISSUS
: サスペンド禁止
サスペンドを禁止する。

TD_ENASUS
: サスペンド許可
: サスペンドを許可する。
: 自リソースグループでサスペンド禁止した回数以上に許可した場合は何もしない。

TD_CHECK
: サスペンド禁止カウント取得
: サスペンド禁止要求を行っている回数の取得のみ行う。

サスペンドは、次の手順によって行われる。

1. 各サブシステムのサスペンド開始前の処理
   tk_evt_ssy(0, TSEVT_SUSPEND_BEGIN, 0, 0)
2. 各ディスクデバイス以外のサスペンド処理
3. 各ディスクデバイスのサスペンド処理
4. 各サブシステムのサスペンド完了後の処理
   tk_evt_ssy(0, TSEVT_SUSPEND_DONE, 0, 0)
5. サスペンド状態へ移行
   tk_set_pow(TPW_DOSUSPEND)

リジューム(サスペンドからの復帰)は、次の手順によって行われる。

1. サスペンド状態から復帰
   tk_set_pow(TPW_DOSUSPEND) から戻る
2. 各サブシステムのリジューム開始前の処理
   tk_evt_ssy(0, TSEVT_RESUME_BEGIN, 0, 0)
3. 各ディスクデバイスのリジューム処理
4. 各ディスクデバイス以外のリジューム処理
5. 各サブシステムのリジューム完了後の処理
   tk_evt_ssy(0, TSEVT_RESUME_DONE, 0, 0)

上記の処理において、各デバイスがディスクデバイスかそれ以外かの判断は、デバイス属性がディスクタイプ(TDK_DISK)かどうかによって行われる。

サスペンド禁止は、その要求回数がカウントされる。同じ回数だけサスペンド許可を要求しないとサスペンドは許可されない。システム起動時はサスペンド許可(サスペンド禁止要求カウント＝0)である。サスペンド禁止要求カウントはシステム全体で1つだが、その要求がどのリソースグループから行われたかを管理する。他のリソースグループで行ったサスペンド禁止要求を解除することはできない。なお、リソースグループのクリーンアップ処理が行われることによって、そのリソースグループで行ったサスペンド要求はすべて解除され、サスペンド禁止要求カウントは減算される。サスペンド禁止要求カウントの上限は実装依存だが、最低255回まではカウントできるものとする。上限を超えた場合は E_QOVR を返す。

5.3.2.14 tk_get_dev - デバイスのデバイス名取得

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID pdevid = tk_get_dev (ID devid , UB *devnm);

パラメータ

ID	devid	Device ID	デバイスID
UB*	devnm	Device Name	デバイス名の格納領域へのポインタ

リターンパラメータ

ID	pdevid	Device ID of Physical Device	物理デバイスのデバイスID
	または	Error Code	エラーコード
UB	devnm	Device Name	デバイス名

エラーコード

E_NOEXS　devid のデバイスは存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

devid で示すデバイスのデバイス名を取得し devnm に格納する。

devid は物理デバイスのデバイスIDまたは論理デバイスのデバイスIDである。

devid が物理デバイスであれば、devnm には物理デバイス名が格納される。

devid が論理デバイスであれば、devnm には論理デバイス名が格納される。

devnm は L_DEVNM+1 バイト以上の領域が必要である。

戻値には、devid のデバイスが属する物理デバイスのデバイスIDを返す。

5.3.2.15 tk_ref_dev - デバイスのデバイス情報取得

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID devid = tk_ref_dev (CONST UB *devnm , T_RDEV *rdev);

パラメータ

CONST UB*	devnm	Device Name	デバイス名
T_RDEV*	rdev	Packet to Refer Device Information	デバイス情報を返す領域へのポインタ

リターンパラメータ

ID	devid	Device ID	デバイスID
	または	Error Code	エラーコード

rdev の内容

ATR	devatr	Device Attribute	デバイス属性
INT	blksz	Block Size of Device-specific Data	固有データのブロックサイズ(-1:不明)
INT	nsub	Subunit Count	サブユニット数
INT	subno	Subunit Number	0:物理デバイス 1～nsub:サブユニット番号+1

――(以下に実装独自に他の情報を追加してもよい)――

エラーコード

E_NOEXS	devnm のデバイスは存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

devnm で示すデバイスのデバイス情報を取得し、rdev に格納する。rdev＝NULL とした場合には、デバイス情報は格納されない。

nsub は、devnm で示すデバイスが属する物理デバイスのサブユニット数である。

戻値に devnm のデバイスのデバイスIDを返す。

5.3.2.16 tk_oref_dev - デバイスのデバイス情報取得

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID devid = tk_oref_dev (ID dd , T_RDEV *rdev);

パラメータ

ID	dd	Device Descriptor	デバイスディスクリプタ
T_RDEV*	rdev	Packet to Refer Device Information	デバイス情報を返す領域へのポインタ

リターンパラメータ

ID	devid	Device ID	デバイスID
	または	Error Code	エラーコード

rdev の内容

ATR	devatr	Device Attribute	デバイス属性
INT	blksz	Block Size of Device-specific Data	固有データのブロックサイズ(-1:不明)
INT	nsub	Subunit Count	サブユニット数
INT	subno	Subunit Number	0:物理デバイス 1～nsub:サブユニット番号+1

――(以下に実装独自に他の情報を追加してもよい)――

エラーコード

E_ID	dd が不正またはオープンされていない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

dd で示すデバイスのデバイス情報を取得し、rdev に格納する。rdev＝NULL とした場合には、デバイス情報は格納されない。

nsub は、dd で示すデバイスが属する物理デバイスのサブユニット数である。

戻値に dd のデバイスのデバイスIDを返す。

5.3.2.17 tk_lst_dev - 登録済みデバイス一覧の取得

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT remcnt = tk_lst_dev (T_LDEV *ldev , INT start , INT ndev);

パラメータ

T_LDEV*	ldev	List of Devices	登録デバイス情報の格納領域(配列)
INT	start	Starting Number	開始番号
INT	ndev	Number of Devices	取得数

リターンパラメータ

INT	remcnt	Remaining Device Count	残りの登録数
	または	Error Code	エラーコード

ldev の内容

ATR	devatr	Device Attribute	デバイス属性
INT	blksz	Block Size of Device-specific Data	固有データのブロックサイズ(-1:不明)
INT	nsub	Subunit Count	サブユニット数
UB	devnm[L_DEVNM]	Physical Device Name	物理デバイス名

――(以下に実装独自に他の情報を追加してもよい)――

エラーコード

E_NOEXS	start が登録数を超えている

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

登録済みのデバイスの情報を取得する。登録デバイスは物理デバイス単位で管理される。したがって、登録デバイス情報も物理デバイス単位で取得される。

登録デバイスの数がNの時、登録デバイスに0～N-1の連番を振る。この連番にしたがって、start 番目から ndev 個の登録情報を取得し、ldev に格納する。ldev は ndev 個の情報を格納するのに十分な大きさの領域でなければならない。戻値には、start 以降の残りの登録数(N-start)を返す。

start 以降の残りが ndev 個に満たない場合は、残りのすべてを格納する。戻値≦ndev であれば、すべての登録情報が取得できたことを示す。なお、この連番はデバイスの登録・抹消があると変化する。したがって、複数回に分けて取得すると、正確な情報が得られない場合がある。

5.3.2.18 tk_evt_dev - デバイスにドライバ要求イベントを送信

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT retcode = tk_evt_dev (ID devid , INT evttyp , void *evtinf);

パラメータ

ID	`devid`	Device ID	イベント送信先デバイスID
INT	`evttyp`	Event Type	ドライバ要求イベントタイプ
void*	`evtinf`	Event Information	イベントタイプ別の情報

リターンパラメータ

INT	`retcode`	Return Code from eventfn	eventfn からの戻値
	または	Error Code	エラーコード

エラーコード

E_NOEXS	`devid` のデバイスは存在しない
E_PAR	デバイス管理内部イベント(`evttyp` < 0)は指定できない
その他	デバイスドライバから返されたエラーコード

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

`devid` のデバイス(デバイスドライバ)に、ドライバ要求イベントを送信する。

ドライバ要求イベントの機能(処理内容)および `evtinf` の内容はイベントタイプごとに定義される。ドライバ要求イベントについては、「5.3.3.2.6節「eventfn - イベント関数」」を参照のこと。

5.3.3 デバイスドライバの登録

5.3.3.1 デバイスドライバの登録方法

デバイスドライバは、物理デバイスごとに登録する。

5.3.3.1.1 tk_def_dev - デバイスの登録

C言語インタフェース

```
#include <tk/tkernel.h>
```

ID devid = tk_def_dev (CONST UB *devnm , CONST T_DDEV *ddev , T_IDEV *idev);

パラメータ

CONST UB*	devnm	Physical Device Name	物理デバイス名
CONST T_DDEV*	ddev	Define Device	デバイス登録情報
T_IDEV*	idev	Initial Device Information	デバイス初期情報

リターンパラメータ

ID	devid	Device ID	デバイスID
	または	Error Code	エラーコード

idev の内容

ID	evtmbfid	Event Notification Message Buffer ID	事象通知用メッセージバッファID

――(以下に実装独自に他の情報を追加してもよい)――

エラーコード

E_LIMIT	登録可能な最大数を超えた
E_NOEXS	devnm のデバイスは存在しない(ddev＝NULL の場合)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

devnm のデバイス名でデバイス(デバイスドライバ)を登録し、登録したデバイスのデバイスIDを戻値に返す。devnm のデバイスがすでに登録されているときは、新しい登録情報で更新する。更新の場合は、デバイスIDは変更されない。

ddev でデバイス登録情報を指定する。ddev＝NULL の場合は、devnm のデバイス登録を抹消する。

ddev は次の形式の構造体である。

```
typedef struct t_ddev {
        void    *exinf;   /* 拡張情報 */
        ATR     drvatr;   /* ドライバ属性 */
        ATR     devatr;   /* デバイス属性 */
        INT     nsub;     /* サブユニット数 */
        INT     blksz;    /* 固有データのブロックサイズ (-1:不明) */
        FP      openfn;   /* オープン関数 */
        FP      closefn;  /* クローズ関数 */
        FP      execfn;   /* 処理開始関数 */
```

```
        FP      waitfn;   /* 完了待ち関数 */
        FP      abortfn;  /* 中止処理関数 */
        FP      eventfn;  /* イベント関数 */
        /* 以下に実装独自の情報が追加される場合がある */
} T_DDEV;
```

exinf は、任意の情報の格納に使用できる。この値は、各処理関数に渡される。内容に関してデバイス管理では関知しない。

drvatr は、デバイスドライバの属性に関する情報を設定する。下位側がシステム属性を表し、上位側が実装独自属性を表す。実装独自属性は、T_DDEVに実装独自データを追加する場合に有効フラグを定義するためなどに使用する。

drvatr := [TDA_OPENREQ] | [TDA_TMO_U] | [TDA_DEV_D]

```
#define TDA_OPENREQ     0x0001  /* 毎回オープン/クローズ */
#define TDA_TMO_U       0x0002  /* マイクロ秒単位タイムアウト時間使用 */
#define TDA_DEV_D       0x0004  /* 64ビットデバイス */
```

drvatr では、以下のドライバ属性を組み合わせて指定することができる。

TDA_OPENREQ

デバイスが多重オープンされた場合、通常は最初のオープン時に openfn が呼び出され、最後のクローズ時に closefn が呼び出される。TDA_OPENREQ を指定した場合、多重オープンの場合でもすべてのオープン／クローズ時に openfn/closefn が呼び出される。

TDA_TMO_U

マイクロ秒単位タイムアウト時間を使用することを示す。

この場合、ドライバ処理関数のタイムアウト指定 tmout がTMO_U型(マイクロ秒単位)となる。

TDA_DEV_D

64ビットデバイスを使用することを示す。この場合、ドライバ処理関数の要求パケット devreq の型がT_DEVREQ_Dとなる。

TDA_TMO_U および TDA_DEV_D を指定した場合、ドライバ処理関数の一部の引数の型が変化する。このような引数の型を変化させるドライバ属性を複数組み合わせて指定した場合は、指定されたすべての型変化を行った引数を持つドライバ処理関数となる。

devatr は、デバイス属性を設定する。デバイス属性の詳細については前述。

nsub は、サブユニット数を設定する。サブユニットがない場合は0とする。

blksz は、固有データのブロックサイズをバイト数で設定する。ディスクデバイスの場合は、物理ブロックサイズとなる。シリアル回線などは1バイトとなる。固有データのないデバイスでは0とする。未フォーマットのディスクなど、ブロックサイズが不明の場合は-1とする。blksz≦0の場合は、固有データにアクセスできない。tk_rea_dev, tk_wri_dev で固有データをアクセスする場合に、size * blksz がアクセスする領域サイズ、つまり buf のサイズとならなければならない。

openfn, closefn, execfn, waitfn, abortfn, eventfn は、ドライバ処理関数のエントリーアドレスを設定する。ドライバ処理関数の詳細については、「5.3.3.2節「デバイスドライバインタフェース」」を参照。

idev にデバイス初期情報が返される。ここには、デバイスドライバ起動時のデフォルトとして設定するための情報などが返されるので、必要に応じて利用する。idev＝NULL とした場合には、デバイス初期情報は格納されない。

evtmbfid は、システムデフォルトの事象通知用メッセージバッファIDである。システムデフォルトの事象通知用メッセージバッファがない場合は0が設定される。

デバイス登録および抹消が行われたとき、各サブシステムに対して次のように通知が行われる。devid は登録・抹消された、物理デバイスのデバイスIDである。

デバイス登録・更新時
　　tk_evt_ssy(0, TSEVT_DEVICE_REGIST, 0, devid)

デバイス抹消時
　　tk_evt_ssy(0, TSEVT_DEVICE_DELETE, 0, devid)

T-Kernel 1.0との差異

64ビットデバイスへの対応のため、drvatr の属性として TDA_TMO_U および TDA_DEV_D を追加した。

5.3.3.1.2 tk_ref_idv - デバイス初期情報の取得

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = tk_ref_idv (T_IDEV *idev);

パラメータ

T_IDEV*	idev	Packet to Refer Initial Device Information	デバイス初期情報を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

idev の内容

ID	evtmbfid	Event Notification Message Buffer ID	事象通知用メッセージバッファID

――(以下に実装独自に他の情報を追加してもよい)――

エラーコード

E_MACV	メモリアクセス権違反

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

デバイス初期情報を取得する。tk_def_dev() で得られるものと同じ内容である。

補足事項

E_MACV のエラーは、多くのシステムコールで共通に発生する可能性があり、通常はシステムコール別のエラーコードには記載していない。しかし、本拡張SVCの場合は代表的なエラーが E_MACV のみであるため、エラーコード欄にこのエラーを記載している。

5.3.3.2 デバイスドライバインタフェース

デバイスドライバインタフェースは、デバイス登録時に指定した処理関数(ドライバ処理関数)群によって構成される。

オープン関数
ER openfn(ID devid, UINT omode, void *exinf);

クローズ関数
ER closefn(ID devid, UINT option, void *exinf);

処理開始関数
ER execfn(T_DEVREQ *devreq, TMO tmout, void *exinf);

完了待ち関数
INT waitfn(T_DEVREQ *devreq, INT nreq, TMO tmout, void *exinf);

中止処理関数
ER abortfn(ID tskid, T_DEVREQ *devreq, INT nreq, void *exinf);

イベント関数
INT eventfn(INT evttyp, void *evtinf, void *exinf);

ドライバ属性に TDA_TMO_U を指定した場合、次のドライバ処理関数のタイムアウト指定 tmout がTMO_U型(マイクロ秒単位)となる。

処理開始関数
ER execfn(T_DEVREQ *devreq, TMO_U tmout_u, void *exinf);

完了待ち関数
INT waitfn(T_DEVREQ *devreq, INT nreq, TMO_U tmout_u, void *exinf);

ドライバ属性に TDA_DEV_D を指定した場合、次のドライバ処理関数の要求パケット devreq の型がT_DEVREQ_Dとなる。

処理開始関数
ER execfn(T_DEVREQ_D *devreq_d, TMO tmout, void *exinf);

完了待ち関数
INT waitfn(T_DEVREQ_D *devreq_d, INT nreq, TMO tmout, void *exinf);

中止処理関数
ER abortfn(ID tskid, T_DEVREQ_D *devreq_d, INT nreq, void *exinf);

ドライバ属性に TDA_TMO_U と TDA_DEV_D を指定した場合、指定されたすべての型変化を行った引数を持つドライバ処理関数となる。

処理開始関数
ER execfn(T_DEVREQ_D *devreq_d, TMO_U tmout_u, void *exinf);

完了待ち関数
INT waitfn(T_DEVREQ_D *devreq_d, INT nreq, TMO_U tmout_u, void *exinf);

ドライバ処理関数は、デバイス管理によって呼び出され、準タスク部として実行される。これらのドライバ処理関数は、再入可能(reentrant)でなければならない。また、ドライバ処理関数が排他的に呼び出されることは保証されない。例えば、複数のタスクから同じデバイスに対して同時に要求があった場合、それぞれのタスクが同時にドライバ処理関数を呼び出すことがある。デバイスドライバ側は、必要に応じて排他制御などを行う必要がある。

デバイスドライバへの入出力要求は、リクエストIDに対応した下記の要求パケットにより行う。

```
/*
 * デバイス要求パケット: 32ビット用
 * In:  ドライバ処理関数への入力パラメータ(T-Kernel/SMのデバイス管理で設定)
 * Out: ドライバ処理関数からの出力パラメータ(ドライバ処理関数で設定)
 */
typedef struct t_devreq {
        struct t_devreq *next;    /* In:要求パケットのリンク (NULL:終端) */
        void     *exinf;          /* X:拡張情報 */
        ID       devid;           /* In:対象デバイスID */
        INT      cmd:4;           /* In:要求コマンド */
        BOOL     abort:1;         /* In:中止要求を行った時 TRUE */
        BOOL     nolock:1;        /* In:ロック(常駐化)不要のとき TRUE */
        INT      rsv:26;          /* In:予約(常に0) */
        T_TSKSPC tskspc;          /* In:要求タスクのタスク固有空間 */
        W        start;           /* In:開始データ番号 */
        W        size;            /* In:要求サイズ */
        void     *buf;            /* In:入出力バッファアドレス */
        W        asize;           /* Out:結果サイズ */
        ER       error;           /* Out:結果エラー */
        /* 以下に実装独自の情報が追加される場合がある */
} T_DEVREQ;
```

```
/*
 * デバイス要求パケット: 64ビット用
 * In:  ドライバ処理関数への入力パラメータ(T-Kernel/SMのデバイス管理で設定)
 * Out: ドライバ処理関数からの出力パラメータ(ドライバ処理関数で設定)
 */
typedef struct t_devreq_d {
        struct t_devreq_d *next;  /* In:要求パケットのリンク (NULL:終端) */
        void     *exinf;          /* X:拡張情報 */
        ID       devid;           /* In:対象デバイスID */
        INT      cmd:4;           /* In:要求コマンド */
        BOOL     abort:1;         /* In:中止要求を行った時 TRUE */
        BOOL     nolock:1;        /* In:ロック(常駐化)不要のとき TRUE */
        INT      rsv:26;          /* In:予約(常に0) */
        T_TSKSPC tskspc;          /* In:要求タスクのタスク固有空間 */
        D        start_d;         /* In:開始データ番号, 64ビット */
        W        size;            /* In:要求サイズ */
        void     *buf;            /* In:入出力バッファアドレス */
        W        asize;           /* Out:結果サイズ */
        ER       error;           /* Out:結果エラー */
        /* 以下に実装独自の情報が追加される場合がある */
} T_DEVREQ_D;
```

In: はドライバ処理関数への入力パラメータであり、T-Kernel/SMのデバイス管理で設定され、デバイスドライバ側で変更してはならない。入力パラメータ(In:)以外は、デバイス管理により最初に0クリアされる。その後はデバイス管理は変更しない。Out: はドライバ処理関数から戻る際の出力パラメータであり、ドライバ処理関数の中で設定する。

next は、要求パケットをリンクするために使用する。デバイス管理内の要求パケットの管理に使用される他、完了待ち関数(waitfn),中止処理関数(abortfn)でも使用する。

exinf は、デバイスドライバ側で任意に使用できる。デバイス管理では内容には関知しない。

devid は、要求対象のデバイスIDが設定される。

cmd は、要求コマンドが設定される。

cmd := (TDC_READ || TDC_WRITE)

```
#define TDC_READ        1       /* 読込み要求 */
#define TDC_WRITE       2       /* 書込み要求 */
```

abort は中止処理を行う場合、中止処理関数(abortfn)を呼び出す直前に TRUE を設定する。abort は中止処理を要求したことを示すフラグであり、処理が中止されたことを示すものではない。また、中止処理関数(abortfn)を呼び出さない場合でも abort が TRUE に設定される場合がある。abort が TRUE に設定された要求がデバイスドライバに渡された場合は、中止処理を行う。

nolock は、buf の領域が呼出側ですでにロック(常駐化)されており、デバイスドライバ側ではロック不要であることを示す。この場合、デバイスドライバはロックを行ってはいけない。(デバイスドライバでロックを行うと正しく動作しなくなる可能性がある状況で nolock が指定される。したがって、nolock＝TRUE の場合はロックを行ってはいけない。)

tskspc は、デバイス入出力操作のAPIを発行したタスク(API発行タスク)のタスク固有空間が設定される。処理関数は、API発行タスクを要求タスクとする準タスク部のコンテキストで実行されるため、tskspc は処理関数のタスク固有空間と同じである。しかし、実際の入出力処理(buf の領域の読込み／書込み)をデバイスドライバ内の別タスクで行う場合は、実際の入出力処理を行うタスクのタスク固有空間を、API発行タスクのタスク固有空間に切り替える必要がある。

start, start_d, size は、tk_rea_dev(), tk_rea_dev_du(), tk_wri_dev(), tk_wri_dev_du() で指定された start, start_d, size がそのまま設定される。

buf は、tk_rea_dev(), tk_rea_dev_du(), tk_wri_dev(), tk_wri_dev_du() で指定された buf がそのまま設定される。buf の指す領域は、非常駐である場合やタスク固有空間である場合がある。そのため、次の点に注意する必要がある。

- 非常駐のメモリは、タスク独立部やディスパッチ禁止・割込み禁止の状態でアクセスすることはできない。
- タスク固有空間のメモリは、異なるタスク固有空間からはアクセスできない。

そのため、必要に応じてタスク固有空間の切り替えや常駐化を行わなければならない。特に割込みハンドラから直接アクセスする場合には注意が必要である。一般的には、buf の領域に割込みハンドラから直接アクセスしない方がよい。また、buf の領域をアクセスする前に、buf の領域の有効性をアドレス空間チェック機能(ChkSpace～)により検査する必要がある。(ChkSpace～ については前述)

asize は、tk_wai_dev() の asize に返す値をデバイスドライバが設定する。

error は、tk_wai_dev() の戻値として返すエラーコードをデバイスドライバが設定する。正常であれば E_OK を設定する。

T_DEVREQとT_DEVREQ_Dとの差異は、start あるいは start_d の部分の名称およびデータタイプのみである。

デバイス要求パケットの種類(T_DEVREQまたはT_DEVREQ_D)は、デバイス登録時のドライバ属性 TDA_DEV_D によって選択される。したがって、1つのドライバに対する要求パケットにT_DEVREQとT_DEVREQ_Dが混在することはない。

T-Kernel 1.0との差異
T_DEVREQの start, size, asize のデータタイプをINTからWに変更した。また、64ビットデバイスへの対応のため、T_DEVREQ_Dのデバイス要求パケットを追加した。

5.3.3.2.1 openfn - オープン関数

C言語インタフェース

ER ercd = openfn (ID devid , UINT omode , void *exinf);

パラメータ

ID	`devid`	Device ID	オープンするデバイスのデバイスID
UINT	`omode`	Open Mode	オープンモード(tk_opn_dev と同じ)
void*	`exinf`	Extended Information	デバイス登録時に指定した拡張情報

リターンパラメータ

ER	`ercd`	Error Code	エラーコード

エラーコード

その他	デバイスドライバの返すエラーコード

解説

tk_opn_dev() が呼び出されたときに、オープン関数 openfn が呼び出される。

openfn では、デバイスの使用開始のための処理を行う。処理内容はデバイスに依存し、何もする必要がなければ何もしなくてもよい。また、オープンされているか否かをデバイスドライバで記憶する必要もなく、オープンされていない(openfn が呼び出されていない)という理由だけで、他の処理関数が呼び出されたときエラーにする必要もない。オープンされていない状態で他の処理関数が呼び出された場合でも、デバイスドライバの動作に問題がなければ要求を処理してしまって構わない。

openfn でデバイスの初期化等を行う場合でも、待ちを伴うような処理は原則として行わない。できる限り速やかに処理を行い openfn から戻らなければならない。例えば、シリアル回線のように通信モードを設定する必要があるようなデバイスでは、tk_wri_dev() により通信モードが設定されたときにデバイスの初期化を行えばよい。openfn ではデバイスの初期化を行う必要はない。

同じデバイスが多重オープンされた場合、通常は最初のオープン時のみ呼び出されるが、デバイス登録時にドライバ属性として `TDA_OPENREQ` が指定された場合は、すべてのオープン時に呼び出される。

多重オープンやオープンモードに関する処理はデバイス管理で行われるため、openfn ではそれらに関する処理は特に必要ない。`omode` も参考情報として渡されるだけで、`omode` に関する処理を行う必要はない。

openfn は、tk_opn_dev の発行タスクの準タスク部として実行される。すなわち、tk_opn_dev の発行タスクを要求タスクとする準タスク部のコンテキストで実行される。

5.3.3.2.2 closefn - クローズ関数

C言語インタフェース

ER ercd = closefn (ID devid , UINT option , void *exinf);

パラメータ

ID	`devid`	Device ID	クローズするデバイスのデバイスID
UINT	`option`	Close Option	クローズオプション(tk_cls_dev と同じ)
void*	`exinf`	Extended Information	デバイス登録時に指定した拡張情報

リターンパラメータ

ER	`ercd`	Error Code	エラーコード

エラーコード

その他	デバイスドライバの返すエラーコード

解説

tk_cls_dev() が呼び出されたときに、クローズ関数 closefn が呼び出される。

closefn では、デバイスの使用終了のための処理を行う。処理内容はデバイスに依存し、何もする必要がなければ何もしなくてもよい。

メディアの取り外しが可能なデバイスの場合、`option` に `TD_EJECT` が指定されていたならメディアの排出を行う。

closefn でデバイスの終了処理やメディアの排出を行う場合でも、待ちを伴うような処理は原則として行わない。できる限り速やかに処理を行い closefn から戻らなければならない。メディアの排出に時間がかかる場合、排出の完了を待たずに closefn から戻っても構わない。

同じデバイスが多重オープンされていた場合、通常は最後のクローズ時のみ呼び出されるが、デバイス登録時にドライバ属性として `TDA_OPENREQ` が指定された場合は、すべてのクローズ時に呼び出される。ただし、この場合も最後のクローズにしか `option` に `TD_EJECT` が指定されることはない。

多重オープンやオープンモードに関する処理はデバイス管理で行われるため、closefn ではそれらに関する処理は特に必要ない。

closefn は、tk_cls_dev の発行タスクの準タスク部として実行される。クリーンアップ処理によるデバイスクローズの場合は、クリーンアップ関数のコンテキスト、すなわち tk_cln_ssy の発行タスクの準タスク部として実行される。

5.3.3.2.3 execfn - 処理開始関数

C言語インタフェース

```
/* 処理開始関数(32ビット要求パケット、ミリ秒タイムアウト) */

ER ercd = execfn (T_DEVREQ *devreq , TMO tmout , void *exinf );

/* 処理開始関数(64ビット要求パケット、ミリ秒タイムアウト) */

ER ercd = execfn (T_DEVREQ_D *devreq_d , TMO tmout , void *exinf );

/* 処理開始関数(32ビット要求パケット、マイクロ秒タイムアウト) */

ER ercd = execfn (T_DEVREQ *devreq , TMO_U tmout_u , void *exinf );

/* 処理開始関数(64ビット要求パケット、マイクロ秒タイムアウト) */

ER ercd = execfn (T_DEVREQ_D *devreq_d , TMO_U tmout_u , void *exinf );
```

パラメータ

T_DEVREQ*	devreq	Device Request Packet	要求パケット(32ビット)
T_DEVREQ_D*	devreq_d	Device Request Packet	要求パケット(64ビット)
TMO	tmout	Timeout	要求受付待ちタイムアウト時間(ミリ秒)
TMO_U	tmout_u	Timeout	要求受付待ちタイムアウト時間(マイクロ秒)
void*	exinf	Extended Information	デバイス登録時に指定した拡張情報

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

その他	デバイスドライバの返すエラーコード

解説

tk_rea_dev() または tk_wri_dev() が呼び出されたときに、処理開始関数 execfn が呼び出される。

devreq の要求の処理を開始する。処理を開始するのみで、完了を待たずに呼出元へ戻る。処理開始のためにかかる時間はデバイスドライバに依存する。必ずしも即座に完了するとは限らない。

新たな要求を受け付けられない状態のときは、要求受付待ちとなる。tmout に指定した時間以内に新たな要求を受け付けられなければ、タイムアウトする。tmout には、TMO_POL または TMO_FEVR を指定することもできる。タイムアウトした場合、execfn の戻値に E_TMOUT を返す。要求パケットの error は変更しない。タイムアウトするのは要求を受け付けるまでで、要求を受け付けた後はタイムアウトしない。

execfn の戻値にエラーを返した場合は、要求が受け付けられなかったものとして、要求パケットは消滅する。

処理を中止する場合、その要求の受け付け前(処理開始前)であれば execfn の戻値に E_ABORT を返す。この場合、要求パケットは消滅する。要求受け付け後(処理開始後)の場合は、E_OK を返す。この場合、要求パケットは waitfn を実行し処理完了が確認されるまで消滅しない。

中止要求があった場合、execfn からできる限り速やかに戻らなければならない。処理を中止しなくてもすぐに終るのであれば中止しなくてもよい。

execfn は、tk_rea_dev, tk_wri_dev, tk_srea_dev, tk_swri_dev の発行タスクの準タスク部として実行される。

デバイス登録時のドライバ属性として TDA_DEV_D が指定されたデバイスドライバにおいて、tk_rea_dev() または tk_wri_dev() が呼び出されたときに、処理開始関数(64ビット要求パケット、ミリ秒タイムアウト) execfn が呼び出される。この場合、パラメータの要求パケットが64ビットのT_DEVREQ_D* devreq_d となった点を除き、関数の仕様は32ビット要求パケット、ミリ秒タイムアウトの execfn と同じである。

デバイス登録時のドライバ属性として TDA_TMO_U が指定されたデバイスドライバにおいて、tk_rea_dev() または tk_wri_dev() が呼び出されたときに、処理開始関数(32ビット要求パケット、マイクロ秒タイムアウト) execfn が呼び出される。この場合、パラメータのタイムアウト指定がマイクロ秒単位のTMO_U tmout_u となった点を除き、関数の仕様は32ビット要求パケット、ミリ秒タイムアウトの execfn と同じである。

デバイス登録時のドライバ属性として TDA_DEV_D および TDA_TMO_U が指定されたデバイスドライバにおいて、tk_rea_dev() または tk_wri_dev() が呼び出されたときに、処理開始関数(64ビット要求パケット、マイクロ秒タイムアウト) execfn が呼び出される。この場合、パラメータの要求パケットが64ビットのT_DEVREQ_D* devreq_d、パラメータのタイムアウト指定がマイクロ秒単位のTMO_U tmout_u となった点を除き、関数の仕様は32ビット要求パケット、ミリ秒タイムアウトの execfn と同じである。

T-Kernel 1.0との差異

処理開始関数(64ビット要求パケット、ミリ秒タイムアウト)、処理開始関数(32ビット要求パケット、マイクロ秒タイムアウト)、処理開始関数(64ビット要求パケット、マイクロ秒タイムアウト)は、T-Kernel 2.0で追加された関数である。

5.3.3.2.4 waitfn - 完了待ち関数

C言語インタフェース

```
/* 完了待ち関数(32ビット要求パケット、ミリ秒タイムアウト) */

INT creqno = waitfn (T_DEVREQ *devreq , INT nreq , TMO tmout , void *exinf );

/* 完了待ち関数(64ビット要求パケット、ミリ秒タイムアウト) */

INT creqno = waitfn (T_DEVREQ_D *devreq_d , INT nreq , TMO tmout , void *exinf );

/* 完了待ち関数(32ビット要求パケット、マイクロ秒タイムアウト) */

INT creqno = waitfn (T_DEVREQ *devreq , INT nreq , TMO_U tmout_u , void *exinf );

/* 完了待ち関数(64ビット要求パケット、マイクロ秒タイムアウト) */

INT creqno = waitfn (T_DEVREQ_D *devreq_d , INT nreq , TMO_U tmout_u , void *exinf );
```

パラメータ

T_DEVREQ*	devreq	Device Request Packet	要求パケットのリスト(32ビット)
T_DEVREQ_D*	devreq_d	Device Request Packet	要求パケットのリスト(64ビット)
INT	nreq	Number of Requests	要求パケットの数
TMO	tmout	Timeout	タイムアウト時間(ミリ秒)
TMO_U	tmout_u	Timeout	タイムアウト時間(マイクロ秒)
void*	exinf	Extended Information	デバイス登録時に指定した拡張情報

リターンパラメータ

INT	creqno	Completed Request Packet Number	完了した要求パケットの番号
	または	Error Code	エラーコード

エラーコード

その他	デバイスドライバの返すエラーコード

解説

tk_wai_dev() が呼び出されたときに、完了待ち関数 waitfn が呼び出される。

devreq は devreq->next で接続された要求パケットのリストで、devreq から nreq 個分の要求パケットについて、その内のいずれかが完了するのを待つ。リストの最後の next は必ずしも NULL になっているとは限らないため、必ず nreq の指定に従う。戻値に完了した要求パケットの番号(devreq から何番目か)を返す。最初が0番目で最後が nreq-1番目となる。なお、完了とは、正常終了／異常(エラー)終了／中止のいずれかである。

tmout に完了待ちのタイムアウト時間を指定する。TMO_POL または TMO_FEVR を指定することもできる。タイムアウトした場合は、要求された処理は継続中である。タイムアウトの場合、waitfn の戻値に E_TMOUT を返す。要求パケットの error は変更しない。なお、要求された処理を継続中で waitfn から戻るときは、waitfn の戻値に必ずエラーを返す。戻値にエラーを返したにもかかわらず処理が完了していてはいけないし、処理継続中であればエラー以外を返してもいけない。waitfn の戻値に

エラーが返されている限り、その要求は処理中として要求パケットは消滅しない。waitfn の戻値に処理を完了した要求パケットの番号が返されたとき、その要求の処理は完了したものとして要求パケットは消滅する。

入出力エラーなどデバイスに関するエラーを、要求パケットの error に格納する。waitfn の戻値には、完了待ちが正しくできなかった場合のエラーを返す。waitfn の戻値が tk_wai_dev の戻値となり、要求パケットの error が ioer に戻される。

waitfn による完了待ちの間に中止処理関数 abortfn が実行された時の中止の処理は、単一要求の完了待ちだった場合(waitfn の nreq＝1)と、複数要求の完了待ちだった場合(waitfn の nreq > 1)で異なっている。単一要求の完了待ちだった場合は、処理中の要求を中止する。一方、複数要求の完了待ちだった場合は、特別な扱いとして、waitfn による待ちの解除のみを行い、要求に対する処理自体は中止しない。すなわち、中止処理関数 abortfn が実行されても、要求パケットの abort は FALSE のままであり、要求に対する処理は継続する。待ち解除となった waitfn からは、戻値に E_ABORT を返す。

完了待ち要求の中には、中止要求が要求パケットの abort にセットされている場合がある。このような場合、単一要求の完了待ちではその要求の中止処理を行わなければならない。複数要求の完了待ちでも中止処理を行うのが望ましいが、abort フラグを無視しても構わない。

なお、中止処理では waitfn からできる限り速やかに戻ることが重要であり、処理を中止しなくてもすぐに処理が終るのであれば中止しなくてもよい。

処理が中止された場合は、要求パケットの error に E_ABORT を返すことを原則とするが、そのデバイスの特性に合わせて E_ABORT 以外のエラーを返してもよい。また、中止される直前までの処理を有効として E_OK としてもよい。なお、中止要求があっても正常に最後まで処理したのなら E_OK を返す。

waitfn は、tk_wai_dev, tk_srea_dev, tk_swri_dev の発行タスクの準タスク部として実行される。

デバイス登録時のドライバ属性として TDA_DEV_D が指定されたデバイスドライバにおいて、tk_wai_dev() が呼び出されたときに、完了待ち関数(64ビット要求パケット、ミリ秒タイムアウト) waitfn が呼び出される。この場合、パラメータの要求パケットが64ビットのT_DEVREQ_D* devreq_d となった点を除き、関数の仕様は32ビット要求パケット、ミリ秒タイムアウトの waitfn と同じである。

デバイス登録時のドライバ属性として TDA_TMO_U が指定されたデバイスドライバにおいて、tk_wai_dev() が呼び出されたときに、完了待ち関数(32ビット要求パケット、マイクロ秒タイムアウト) waitfn が呼び出される。この場合、パラメータのタイムアウト指定がマイクロ秒単位のTMO_U tmout_u となった点を除き、関数の仕様は32ビット要求パケット、ミリ秒タイムアウトの waitfn と同じである。

デバイス登録時のドライバ属性として TDA_DEV_D および TDA_TMO_U が指定されたデバイスドライバにおいて、tk_wai_dev() が呼び出されたときに、完了待ち関数(64ビット要求パケット、マイクロ秒タイムアウト) waitfn が呼び出される。この場合、パラメータの要求パケットが64ビットのT_DEVREQ_D* devreq_d、パラメータのタイムアウト指定がマイクロ秒単位のTMO_U tmout_u となった点を除き、関数の仕様は32ビット要求パケット、ミリ秒タイムアウトの waitfn と同じである。

T-Kernel 1.0との差異

完了待ち関数(64ビット要求パケット、ミリ秒タイムアウト)、完了待ち関数(32ビット要求パケット、マイクロ秒タイムアウト)、完了待ち関数(64ビット要求パケット、マイクロ秒タイムアウト)は、T-Kernel 2.0で追加された関数である。

5.3.3.2.5 abortfn - 中止処理関数

C言語インタフェース

/* 中止処理関数(32ビット要求パケット) */

ER ercd = abortfn (ID tskid , T_DEVREQ *devreq , INT nreq , void *exinf);

/* 中止処理関数(64ビット要求パケット) */

ER ercd = abortfn (ID tskid , T_DEVREQ_D *devreq_d , INT nreq , void *exinf);

パラメータ

ID	tskid	Task ID	execfn, waitfn を実行しているタスクのタスクID
T_DEVREQ*	devreq	Device Request Packet	要求パケットのリスト(32ビット)
T_DEVREQ_D*	devreq_d	Device Request Packet	要求パケットのリスト(64ビット)
INT	nreq	Number of Requests	要求パケットの数
void*	exinf	Extended Information	デバイス登録時に指定した拡張情報

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

その他	デバイスドライバの返すエラーコード

解説

現在実行中の処理開始関数 execfn や完了待ち関数 waitfn から速やかに戻らせたいときに、中止処理関数 abortfn が呼び出される。通常は、処理中の要求を中止して戻らせる。ただし、中止しなくてもすぐに処理が終るのであれば、必ずしも中止しなくてもよい。重要なのは、できる限り速やかに execfn, waitfn から戻ることである。

abortfn は、次のような場合に呼び出される。

- タスク例外の発生によるブレーク関数の実行時に、タスク例外の発生したタスクによる処理中の要求があった場合、そのタスクによる処理中の要求を中止する。
- tk_cls_dev() およびサブシステムのクリーンアップ処理によるデバイスクローズ時に、クローズするデバイスディスクリプタによる処理中の要求があった場合、そのデバイスディスクリプタによる処理中の要求を中止する。

tskid は、devreq で指定した要求を実行中のタスクである。つまり、execfn, waitfn を実行しているタスクである。devreq, nreq は、execfn, waitfn の引数として指定したものと同じである。ただし、execfn の場合は常に nreq＝1である。

abortfn は、execfn, waitfn を実行しているタスクとは別のタスクから呼び出される。両者は並行して実行されるため、必要に応じて排他制御等を行う必要がある。また、execfn, waitfn の呼出直前や execfn, waitfn から戻る途中に abortfn が呼び出される可能性もある。このような場合においても正しく動作するように配慮する必要がある。abortfn を呼び出す前に、中止処理対象の要求パケットの abort フラグに TRUE が設定されるので、execfn, waitfn はこれにより中止要求の有無を知ることもできる。また、abortfn では、任意のオブジェクトに対する tk_dis_wai() を使用することができる。

複数要求待ち(nreq > 1)の waitfn 実行中の場合は、特別な扱いとなり他の場合とは次の点で異なる。

- 要求の処理を中止することはせず、完了待ちのみ中止(待ちを解除)する。

・ 要求パケットの abort フラグはセットされない(abort＝FALSE のまま)。

なお、execfn, waitfn の実行中でないときに要求を中止させる場合には、abortfn を呼び出すことなく、要求パケットの abort フラグがセットされる。abort フラグがセットされた状態で execfn が呼び出されたときは、要求を受け付けない。waitfn が呼び出されたときは、abortfn が呼び出された場合と同様の中止処理を行う。

execfn により処理を開始した要求が、waitfn による完了待ちでない状態で中止された場合は、後で waitfn が呼び出されたときに中止されて処理が完了したことを知らせる。処理が中止されても、waitfn により完了が確認されるまでは要求自体は消滅しない。

abortfn は中止処理を開始するのみで、中止が完了するまで待たずに速やかに戻る。

タスク例外の際に実行される abortfn は、タスク例外を発生させた tk_ras_tex の発行タスクの準タスク部として実行される。また、デバイスクローズ時に実行される abortfn は、tk_cls_dev の発行タスクの準タスク部として実行される。クリーンアップ処理によるデバイスクローズの場合は、クリーンアップ関数のコンテキスト、すなわち tk_cln_ssy の発行タスクの準タスク部として実行される。

デバイス登録時のドライバ属性として TDA_DEV_D が指定されたデバイスドライバにおいて、現在実行中の処理開始関数 execfn や完了待ち関数 waitfn から速やかに戻らせたいときに、中止処理関数(64ビット要求パケット) abortfn が呼び出される。この場合、パラメータの要求パケットが64ビットのT_DEVREQ_D* devreq_d となった点を除き、関数の仕様は32ビット要求パケットの abortfn と同じである。

T-Kernel 1.0との差異

中止処理関数(64ビット要求パケット)は、T-Kernel 2.0で追加された関数である。

5.3.3.2.6 eventfn - イベント関数

C言語インタフェース

INT retcode = eventfn (INT evttyp , void *evtinf , void *exinf);

パラメータ

INT	evttyp	Event Type	ドライバ要求イベントタイプ
void*	evtinf	Event Information	イベントタイプ別の情報
void*	exinf	Extended Information	デバイス登録時に指定した拡張情報

リターンパラメータ

INT	retcode	Return Code	イベントタイプ別に定義された戻値
	または	Error Code	エラーコード

エラーコード

その他	デバイスドライバの返すエラーコード

解説

アプリケーションインタフェースによる通常のデバイス入出力処理とは別の要因によって、デバイスやシステムの状態変化が発生し、それに対してデバイスドライバ側で何らかの処理を要する場合に、ドライバ要求イベントが発生し、イベント関数 eventfn が呼び出される。

ドライバ要求イベントは、電源管理のためのサスペンド／リジューム時(tk_sus_dev 参照)や、USBやPCカードのような活性挿抜可能な機器の接続時に発生する。

たとえば、tk_sus_dev によってシステムがサスペンドする時には、T-Kernelの内部で(tk_sus_dev の処理の中で)、サスペンドのドライバ要求イベント(TDV_SUSPEND)を発生し、evttyp＝TDV_SUSPEND として各デバイスのイベント関数を呼び出す。各デバイスのイベント関数の中では、この呼出に応じて、サスペンド時に必要な状態保存などの処理を行う。

ドライバ要求イベントタイプには下記のものがある。

```
#define TDV_SUSPEND   (-1)   /* サスペンド */
#define TDV_RESUME    (-2)   /* リジューム */
#define TDV_CARDEVT     1    /* PCカードイベント */
#define TDV_USBEVT      2    /* USBイベント */
```

上記の値が負のドライバ要求イベントは、サスペンド／リジューム時の処理など、T-Kernel/SMのデバイス管理内部からの呼出によるものである。

一方、上記の値が正のドライバ要求イベント(TDV_CARDEVT, TDV_USBEVT)は、tk_evt_dev() の呼出によって発生するものであり、T-Kernelの動作とは直接関係しない参考仕様である。これらのドライバ要求イベントは、USBやPCカードなどのバスドライバを実装するために、必要に応じて利用する。

イベント関数で行う処理は、イベントタイプごとに定義される。サスペンド／リジュームについては「5.3.3.4節「各デバイスのサスペンド／リジューム処理」」を参照。

tk_evt_dev() による呼出の場合、eventfn の戻値はそのまま tk_evt_dev() の戻値となる。

イベント関数への要求は、他の要求の処理中であっても受け付け、できる限り速やかに処理しなければならない。

eventfn は、イベント発生の原因となった tk_evt_dev や tk_sus_dev の発行タスクの準タスク部として実行される。

補足事項

PCカードイベントやUSBイベントで想定している動作は以下のとおりである。

ただし、以下の説明はPCカードやUSBの機器を扱うデバイスドライバの実装例の説明であり、T-Kernelの仕様の範囲には含まれない。

USB機器の接続時には、そのUSB機器に対して、実際の入出力処理を行うクラスドライバを動的に対応付ける必要がある。

たとえば、USBメモリなどのストレージを接続した場合には、マスストレージクラス用のデバイスドライバがその機器の入出力処理を行うが、USBカメラを接続した場合には、ビデオクラス用のデバイスドライバがその機器の入出力処理を行う。どのデバイスドライバを使うべきかは、USB機器が接続されるまで分からない。

この時、USB機器とクラスドライバとの対応付けの処理を行うための手段として、USB接続のドライバ要求イベントと、各デバイスドライバのイベント関数を利用する。具体的には、USBポートを監視していたUSBバスドライバ(USBマネージャ)が、新しいUSB機器の接続を検出した場合に、クラスドライバの候補となる各デバイスドライバに対して、USB接続のドライバ要求イベント(`TDV_USBEVT`)を送り、各デバイスのイベント関数を呼び出す。

各デバイスのイベント関数では、この `TDV_USBEVT` に応答する形で、新しく接続されたUSB機器への対応の可否を返す。USBバスドライバでは、その戻値を見て、実際のクラスドライバとの対応付けを決定する。

PCカードの接続時の処理も、同様の手順になる。

5.3.3.3 デバイス事象通知

デバイスドライバは、各デバイスで発生した事象を、デバイス事象通知のメッセージとして特定のメッセージバッファ(事象通知用メッセージバッファ)へ送信する。事象通知用メッセージバッファのIDは、各デバイスに対する TDN_EVENT の属性データとして参照あるいは設定される。

デバイス登録の直後には、システムデフォルトの事象通知用メッセージバッファを使用する。デバイスドライバの起動時には、tk_def_dev によってデバイスの登録を行うが、このAPIのリターンパラメータとしてシステムデフォルトの事象通知用メッセージバッファのID値が返されるので、デバイスドライバではその値を保持し、属性データ TDN_EVENT の初期値とする。

なお、システムデフォルトの事象通知用メッセージバッファは、システム起動時に生成され、そのサイズとメッセージ最大長は、システム構成情報の TDEvtMbfSz により定義される。

デバイス事象通知で使用するメッセージの形式は以下のようになる。事象通知のメッセージの内容やサイズは、事象タイプごとに異なる。

◇デバイス事象通知の基本形式

```
typedef struct t_devevt {
        TDEvtTyp        evttyp;         /* 事象タイプ */
        /* 以下に事象タイプ別の情報が付加される */
} T_DEVEVT;
```

◇デバイスID付きのデバイス事象通知の形式

```
typedef struct t_devevt_id {
        TDEvtTyp        evttyp;         /* 事象タイプ */
        ID              devid;          /* デバイスID */
        /* 以下に事象タイプ別の情報が付加される */
} T_DEVEVT_ID;
```

◇拡張情報付きのデバイス事象通知の形式

```
typedef struct t_devevt_ex {
        TDEvtTyp        evttyp;         /* 事象タイプ */
        ID              devid;          /* デバイスID */
        UB              exdat[16];      /* 拡張情報 */
        /* 以下に事象タイプ別の情報が付加される */
} T_DEVEVT_EX;
```

デバイス事象通知の事象タイプは、以下のように大分類される。

a. 基本事象通知(事象タイプ 0x0001～0x002F)
   デバイスからの基本的な事象の通知

b. システム事象通知(事象タイプ 0x0030～0x007F)
   電源制御などシステム全体に関係する事象の通知

c. 拡張情報付き事象通知(事象タイプ 0x0080～0x00FF)
   拡張情報を持つデバイスからの事象の通知

d. ユーザ定義事象通知(事象タイプ 0x0100～0xFFFF)
   ユーザが内容を任意に定義可能な事象の通知

事象タイプのうち、代表的なものは以下の通りである。各事象の詳細や、これ以外の事象タイプについては、デバイスドライバ関連の仕様書、あるいは「7.1節「参考とすべきデバイスドライバ関連の仕様」」の「7.1.3節「デバイス事象通知用の事象タイプ」」を参照のこと。

```
typedef enum tdevttyp {
        TDE_unknown     = 0,            /* 未定義 */
        TDE_MOUNT       = 0x01,         /* メディア挿入 */
        TDE_EJECT       = 0x02,         /* メディア排出 */
        TDE_POWEROFF    = 0x31,         /* 電源スイッチオフ */
        TDE_POWERLOW    = 0x32,         /* 電源残量警告 */
        TDE_POWERFAIL   = 0x33,         /* 電源異常 */
        TDE_POWERSUS    = 0x34          /* 自動サスペンド */
} TDEvtTyp;
```

事象通知用メッセージバッファが一杯で事象通知を送信できない場合は、その事象が通知されないことで事象通知の受信側の動作に悪影響が出ないようにしなければならない。メッセージバッファが空くまで待ってから事象通知を行ってもよいが、その場合も原則として事象通知以外のデバイスドライバの処理が滞ってはならない。なお、事象通知の受信側は、できる限りメッセージバッファが溢れることがないように処理しなければならない。

T-Kernel 1.0との差異
デバイス事象通知で使用するメッセージの形式および事象タイプの説明を整理した。

5.3.3.4 各デバイスのサスペンド／リジューム処理

イベント関数(eventfn)へのサスペンド／リジューム(TDV_SUSPEND/TDV_RESUME)イベントの発行により、各デバイスドライバはデバイスのサスペンド／リジューム処理を行う。サスペンド／リジュームイベントは、物理デバイスに対してのみ発行される。

TDV_SUSPEND
: サスペンド

```
evttyp = TDV_SUSPEND
evtinf = NULL (なし)
```

次のような手順でサスペンド処理を行う。

1. 現在処理中の要求があれば、完了するまで待つか、中断または中止する。どの方法を選択するかはデバイスドライバの実装に依存する。ただし、できるだけ速やかにサスペンドする必要があるため、完了までに時間がかかる場合は中断または中止としなければならない。
   サスペンドイベントは物理デバイスに対してのみ発行されるが、そのデバイスに含まれるすべての論理デバイスに対しても、同様に処理する。
   中断：処理を一時的に中断し、リジューム後に続きを行う。
   中止：中止処理関数(abortfn)による中止と同様に、処理を中止する。リジューム後も再開されない。
2. リジュームイベント以外の新たな要求を受け付けないようにする。
3. デバイスの電源を切るなどのサスペンド処理を行う。

中止はアプリケーションへの影響が大きいと考えられるため極力避けたい。シリアル回線からの長期の入力待ちなどで、かつ中断とするのが難しい場合以外は中止としない。通常は、終了まで待つか、可能であれば中断とする。

サスペンド期間中にデバイスドライバへ来た要求は、リジュームまで待たせてリジューム後に受け付け処理する。ただし、デバイスへのアクセスを伴わない処理など、サスペンド中でも可能な処理は受け付けてもよい。

TDV_RESUME
: リジューム

```
evttyp = TDV_RESUME
evtinf = NULL (なし)
```

次のような手順でリジューム処理を行う。

1. デバイスの電源を入れデバイスの状態を復帰させるなどのリジューム処理を行う。
2. 中断していた処理があれば再開する。
3. 要求受け付けを再開する。

5.3.3.5 ディスクデバイスの特殊性

仮想記憶システムにおいて、ディスクデバイスは特別なものとなる。仮想記憶を行うにあたり、メモリとディスク間でのデータ転送を行うため、OS(具体的には、T-Kernel Extensionなどの中で仮想記憶の処理を行う部分)がディスクドライバを呼び出す必要がある。

OSがディスクとのデータ転送を行う必要があるのは、非常駐メモリにアクセスされたことにより、そのメモリの内容をディスクから読み出す(ページイン)ような場合である。このような場合に、OSはディスクドライバを呼び出す。

もし、ディスクドライバ内で非常駐メモリにアクセスした場合、OSはやはりディスクドライバを呼び出さなければならなくなる。このようなケースでは、非常駐メモリにアクセスしたことでページイン待ちとされているディスクドライバに対して、OSが再びディスクアクセスの要求を出す可能性がある。このような場合でも、ディスクドライバは後から来たOSからの要求を実行できなければならない。

同じ様なことが、サスペンド処理時にも起きる可能性がある。サスペンド処理中に非常駐メモリがアクセスされ、ディスクドライバが呼び出されたとき、そのディスクドライバがすでにサスペンドされているとページインができないことになる。このようなケースを避けるため、サスペンド時にはディスクデバイス以外を先にサスペンドした後、ディスクデバイスをサスペンドする。しかし、複数のディスクデバイスがあれば、そのディスクデバイスのサスペンド順序は不定である。したがって、ディスクドライバでは、サスペンド処理中に非常駐メモリにアクセスしてはならない。

このような制限から、ディスクドライバでは非常駐メモリを使用しない(アクセスしない)ようにしなければならない。ただし、tk_rea_dev(), tk_wri_dev() で指定される入出力バッファ(`buf`)に関しては、呼出側で指定したメモリ領域であるため非常駐メモリである可能性がある。そのため、入出力バッファに関しては、入出力を行う間は常駐化(LockSpace 参照)しておくなどの対応が必要である。

5.4 割込み管理機能

T-Kernel/SMの割込み管理機能は、外部割込みの禁止や許可、割込み禁止状態の取得、割込みコントローラの制御などを行うための機能である。

割込み関係はハードウェア依存度が高く、システムごとに異なっているため共通化することが難しい。下記を標準仕様として定めるが、システムによってはこの通りに実装することが難しい場合がある。できる限り標準仕様に合わせた実装を求めるが、実装不可能なものは実装しなくてもよい。標準仕様とは別の機能を追加することも許されるが、その場合は関数名などは標準仕様と異なるものでなければならない。ただし、DI(), EI(), isDI() は標準仕様にしたがって、必ず実装しなければならない。

割込み管理機能は、ライブラリ関数またはC言語のマクロで提供され、これらはタスク独立部およびディスパッチ禁止・割込み禁止の状態から呼び出すことができる。

5.4.1 CPU割込み制御

CPUの外部割込みフラグを制御する。一般的には、割込みコントローラに対しては何もしない。

DI(), EI(), isDI() は、C言語のマクロである。

5.4.1.1 DI - 外部割込み禁止

C言語インタフェース

```
#include <tk/tkernel.h>
```

DI (UINT intsts);

パラメータ

UINT	intsts	Interrupt Status	CPUの外部割込みフラグを保存する変数

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

CPU内の外部割込みフラグを制御し、すべての外部割込みを禁止する。また、割込みを禁止する前のフラグの状態を intsts に保存する。

本APIはC言語のマクロで定義されており、intsts はポインタではない。変数を直接記載する。

5.4.1.2 EI - 外部割込み許可

C言語インタフェース

```
#include <tk/tkernel.h>
```

EI (UINT intsts);

パラメータ

UINT	intsts	Interrupt Status	CPUの外部割込みフラグを保存した変数

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

CPU内の外部割込みフラグを制御し、intsts の状態に戻す。すなわち、これ以前に実行された DI(intsts) で割込みを禁止する前の状態に戻す。

DI(intsts) の実行前の状態が外部割込み許可であった場合には、その後の EI(intsts) の実行により、再度外部割込みを許可した状態となる。一方、DI(intsts) で割込みを禁止する以前から割込み禁止状態であった場合には、EI(intsts) を実行しても割込みは許可されない。ただし、intsts として0を指定した場合は、CPU内の外部割込みフラグが割込み許可状態となる。

intsts は、DI() で保存した値または0のいずれかでなければならない。それ以外の値を指定した場合の動作は保証されない。

5.4.1.3 isDI - 外部割込み禁止状態の取得

C言語インタフェース

```
#include <tk/tkernel.h>
```

BOOL disint = isDI (UINT intsts);

パラメータ

UINT	intsts	Interrupt Status	CPUの外部割込みフラグを保存した変数

リターンパラメータ

BOOL	disint	Interrupts Disabled Status	外部割込み禁止状態

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

以前に実行された DI() によって intsts に保存されていたCPU内の外部割込みフラグの状態を調べ、T-Kernel/OSで割込み禁止と判断される状態であった場合にはTRUE(0以外の値)を、割込み許可と判断される状態であった場合にはFALSEを返す。

intsts は、DI() で保存した値でなければならない。それ以外の値を指定した場合の動作は保証されない。

例 5.4 isDIの使用例

```
void foo()
{
        UINT    intsts;

        DI(intsts);

        if ( isDI(intsts) ) {
                /* 上記の DI() が呼び出された時点で既に割込み禁止であった */
        } else {
                /* 上記の DI() が呼び出された時点では割込み許可であった */
        }

        EI(intsts);
}
```

5.4.2 割込みコントローラ制御

割込みコントローラを制御する。一般的には、CPUの割込みフラグに対しては何もしない。

```
typedef UINT    INTVEC;         /* 割込みベクタ */
```

割込みベクタ(INTVEC)の具体的な内容は、実装定義である。ただし、tk_def_int() で指定する割込みハンドラ番号と同じか、簡単な方法で割込みハンドラ番号と相互に変換できる様なものであることが望ましい。

5.4.2.1 DINTNO - 割込みベクタから割込みハンドラ番号へ変換

C言語インタフェース

```
#include <tk/tkernel.h>
```

UINT dintno = DINTNO (INTVEC intvec);

パラメータ

INTVEC	intvec	Interrupt Vector	割込みベクタ

リターンパラメータ

UINT	dintno	Interrupt Handler Number	割込みハンドラ番号

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

割込みベクタから割込みハンドラ番号へ変換する。

5.4.2.2 EnableInt - 割込み許可

C言語インタフェース

```
#include <tk/tkernel.h>
```

void EnableInt (INTVEC intvec);
void EnableInt (INTVEC intvec , INT level);

パラメータ

INTVEC	intvec	Interrupt Vector	割込みベクタ
INT	level	Interrupt Priority Level	割込み優先度レベル

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

intvec の割込みを許可する。割込み優先度レベルを指定可能なシステムでは、level により割込み優先度レベルを指定する。level の具体的な意味は実装定義である。

level あり、または level なしの、いずれか一方を提供する。

5.4.2.3 DisableInt - 割込み禁止

C言語インタフェース

```
#include <tk/tkernel.h>
```

void DisableInt (INTVEC intvec);

パラメータ

INTVEC	intvec	Interrupt Vector	割込みベクタ

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

intvec の割込みを禁止する。一般的には、割込み禁止中の割込みはペンディングされ、EnableInt() により許可した時に割込みが発生する。割込み禁止中に発生した割込みを無効にしたい場合は、ClearInt() を行う必要がある。

5.4.2.4 ClearInt - 割込み発生のクリア

C言語インタフェース

```
#include <tk/tkernel.h>
```

void ClearInt (INTVEC intvec);

パラメータ

INTVEC	intvec	Interrupt Vector	割込みベクタ

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

intvec の割込みが発生していればクリアする。

5.4.2.5 EndOfInt - 割込みコントローラにEOI発行

C言語インタフェース

```
#include <tk/tkernel.h>
```

void EndOfInt (INTVEC intvec);

パラメータ

INTVEC	intvec	Interrupt Vector	割込みベクタ

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

割込みコントローラにEOI(End Of Interrupt)を発行する。intvec は EOI 発行対象の割込みでなければならない。一般的には、割込みハンドラの最後で実行する必要がある。

5.4.2.6 CheckInt - 割込み発生の検査

C言語インタフェース

```
#include <tk/tkernel.h>
```

BOOL rasint = CheckInt (INTVEC intvec);

パラメータ

INTVEC	intvec	Interrupt Vector	割込みベクタ

リターンパラメータ

BOOL	rasint	Interrupts Raised Status	外部割込み発生状態

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

intvec の割込みが発生しているか調べる。intvec の割込みが発生していれば TRUE(0以外の値)、発生していなければ FALSE を返す。

5.4.2.7 SetIntMode - 割込みモード設定

C言語インタフェース

```
#include <tk/tkernel.h>
```

void SetIntMode (INTVEC intvec , UINT mode);

パラメータ

INTVEC	intvec	Interrupt Vector	割込みベクタ
UINT	mode	Mode	割込みモード

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

intvec で指定した割込みを mode で指定したモードに設定する。

設定可能な機能や mode の指定方法は実装依存である。以下は、設定可能な機能の一例である。

```
mode := (IM_LEVEL || IM_EDGE) | (IM_HI || IM_LOW)
```

```
#define IM_LEVEL        0x0002          /* レベルトリガ */
#define IM_EDGE         0x0000          /* エッジトリガ */
#define IM_HI           0x0000          /* Hレベル/立ち上がりエッジで割込み */
#define IM_LOW          0x0001          /* Lレベル/立ち下がりエッジで割込み */
```

不正な mode を指定した場合の動作は保証されない。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.5 I/Oポートアクセスサポート機能

I/Oポートアクセスサポート機能は、入出力デバイスへのアクセスや操作をサポートするための機能である。アドレスを指定したI/Oポートに対して、バイト単位やワード単位で読込みや書込みを行う機能、入出力デバイスの操作時などに使う短い時間の待ち(微小待ち)を実現する機能が含まれる。

I/Oポートアクセスサポート機能は、ライブラリ関数またはC言語のマクロで提供され、これらはタスク独立部およびディスパッチ禁止・割込み禁止の状態から呼び出すことができる。

5.5.1 I/Oポートアクセス

I/O空間とメモリ空間が独立しているシステムでは、I/Oポートアクセス関数はI/O空間のアクセスとなる。メモリマップドI/Oのみのシステムでは、I/Oポートアクセス関数はメモリ空間のアクセスとなる。メモリマップドI/Oのシステムにおいても、これらの関数を利用することにより、ソフトウェアの移植性や可読性が向上する。

5.5.1.1 out_b - I/Oポート書込み(バイト)

C言語インタフェース

```
#include <tk/tkernel.h>
```

void out_b (INT port , UB data);

パラメータ

INT	port	I/O Port Address	I/Oポートアドレス
UB	data	Write Data	書き込むデータ(バイト)

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

port のアドレスで示されるI/Oポートに、バイト(8ビット)単位で data を書き込む。

5.5.1.2 out_h - I/Oポート書込み(ハーフワード)

C言語インタフェース

```
#include <tk/tkernel.h>
```

void out_h (INT port , UH data);

パラメータ

INT	port	I/O Port Address	I/Oポートアドレス
UH	data	Write Data	書き込むデータ(ハーフワード)

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

port のアドレスで示されるI/Oポートに、ハーフワード(16ビット)単位で data を書き込む。

5.5.1.3 out_w - I/Oポート書込み(ワード)

C言語インタフェース

```
#include <tk/tkernel.h>
```

```
void out_w (INT port , UW data );
```

パラメータ

INT	port	I/O Port Address	I/Oポートアドレス
UW	data	Write Data	書き込むデータ(ワード)

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

port のアドレスで示されるI/Oポートに、ワード(32ビット)単位で data を書き込む。

5.5.1.4 out_d - I/Oポート書込み(ダブルワード)

C言語インタフェース

```
#include <tk/tkernel.h>
```

void out_d (INT port , UD data);

パラメータ

INT	port	I/O Port Address	I/Oポートアドレス
UD	data	Write Data	書き込むデータ(ダブルワード)

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

port のアドレスで示されるI/Oポートに、ダブルワード(64ビット)の data を書き込む。

なお、ハードウェアの制約により、ダブルワード(64ビット)単位でI/0ポートを一括アクセスできないシステムにおいては、ダブルワード(64ビット)よりも短い単位に分割されて処理される。

仕様決定の理由

I/O用のデータバスが32ビット以下の場合など、ハードウェア構成の制約により、ダブルワード(64ビット)単位でのI/0ポートアクセスができないシステムも多く存在する。このようなシステム上には、指定のビット幅のデータを一括して処理するという意味での厳密な out_d() や in_d() の仕様は実装できないことになり、このAPIの本来の目的から言えば、out_d() や in_d() を未実装とするか、実行時にエラーを返すのが望ましい。しかしながら、実行時にバス構成等を判断してエラーを検出するのは現実的ではないし、64ビットのデータの書込みを32ビット以下の単位に分割して処理しても、問題にならないケースも多い。

このような状況から、out_d() や in_d() の仕様においては、64ビットのデータを一括して処理できないケースについても許容している。したがって、out_d() や in_d() で64ビットI/Oポートの一括アクセスが保証できるかどうかは実装依存である。64ビットI/Oポートの一括アクセスが必要であれば、システムのハードウェア構成および out_d() や in_d() の処理方法の確認が必要である。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.5.1.5 in_b - I/Oポート読込み(バイト)

C言語インタフェース

```
#include <tk/tkernel.h>
```

UB data = in_b (INT port);

パラメータ

INT	`port`	I/O Port Address	I/Oポートアドレス

リターンパラメータ

UB	`data`	Read Data	読み込んだデータ(バイト)

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

`port` のアドレスで示されるI/Oポートから、バイト(8ビット)単位で読み込んだデータを、リターンパラメータ `data` として返す。

5.5.1.6 in_h - I/Oポート読込み(ハーフワード)

C言語インタフェース

```
#include <tk/tkernel.h>

UH data = in_h (INT port );
```

パラメータ

INT	`port`	I/O Port Address	I/Oポートアドレス

リターンパラメータ

UH	`data`	Read Data	読み込んだデータ(ハーフワード)

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

`port` のアドレスで示されるI/Oポートから、ハーフワード(16ビット)単位で読み込んだデータを、リターンパラメータ `data` として返す。

5.5.1.7 in_w - I/Oポート読込み(ワード)

C言語インタフェース

```
#include <tk/tkernel.h>
```

UW data = in_w (INT port);

パラメータ

INT	port	I/O Port Address	I/Oポートアドレス

リターンパラメータ

UW	data	Read Data	読み込んだデータ(ワード)

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

port のアドレスで示されるI/Oポートから、ワード(32ビット)単位で読み込んだデータを、リターンパラメータ data として返す。

5.5.1.8 in_d - I/Oポート読込み(ダブルワード)

C言語インタフェース

```
#include <tk/tkernel.h>
```

UD data = in_d (INT port);

パラメータ

INT	`port`	I/O Port Address	I/Oポートアドレス

リターンパラメータ

UD	`data`	Read Data	読み込んだデータ(ダブルワード)

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

`port` のアドレスで示されるI/Oポートから、ダブルワード(64ビット)のデータを読み込み、リターンパラメータ `data` として返す。

なお、ハードウェアの制約により、ダブルワード(64ビット)単位でI/0ポートを一括アクセスできないシステムにおいては、ダブルワード(64ビット)よりも短い単位に分割されて処理される。

仕様決定の理由

「5.5.1.4節「out_d - I/Oポート書込み(ダブルワード)」」を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.5.2 微小待ち

5.5.2.1 WaitUsec - 微小待ち(マイクロ秒)

C言語インタフェース

```
#include <tk/tkernel.h>
```

void WaitUsec (UINT usec);

パラメータ

UINT	usec	Micro Seconds	待ち時間(マイクロ秒)

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

指定された時間分(マイクロ秒)の微小待ちを行う。

この待ちは通常ビジーループで実装される。すなわちカーネルの待ち状態ではなく、実行状態のまま微小待ちは行われる。

RAM上で実行する場合、ROM上で実行する場合、メモリキャッシュがオンの場合、オフの場合など、実行環境の影響を受けやすい。したがって、これらの待ち時間はあまり正確ではない。

5.5.2.2 WaitNsec - 微小待ち(ナノ秒)

C言語インタフェース

```
#include <tk/tkernel.h>
```

void WaitNsec (UINT nsec);

パラメータ

UINT	nsec	Nano Seconds	待ち時間(ナノ秒)

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

指定された時間分(ナノ秒)の微小待ちを行う。

この待ちは通常ビジーループで実装される。すなわちカーネルの待ち状態ではなく、実行状態のまま微小待ちは行われる。

RAM上で実行する場合、ROM上で実行する場合、メモリキャッシュがオンの場合、オフの場合など、実行環境の影響を受けやすい。したがって、これらの待ち時間はあまり正確ではない。

5.6 省電力機能

省電力機能は、システムの省電力を実現するための機能である。T-Kernel/OSの中からコールバック型の関数として呼び出される。

省電力機能で定義されるAPIには、low_pow(), off_pow() があるが、これらは参考仕様であり、T-Kernel内部でのみ使用する。デバイスドライバ、ミドルウェア、アプリケーションなどが本機能を直接呼び出して使うことはないので、本機能やそのAPIを独自の仕様とすることにより、より高度な省電力機能を実現しても構わない。ただし、本機能の参考仕様として定義されるAPIと同等程度の機能を実装するのであれば、プログラムの再利用性を高める意味で、参考仕様に合わせる方が望ましい。

本機能のAPIの呼出方法についても、実装定義である。単純な関数呼出でもよいし、トラップを使用してもよい。また、T-Kernel以外のプログラムの中にこれらの機能を用意してもよい。ただし、拡張SVCなどのT-Kernelの機能を利用する呼出方法は使用できない。

5.6.1 low_pow - システムを低消費電力モードに移行

C言語インタフェース

void low_pow (void);

パラメータ

なし

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
×	×	×

解説

T-Kernelのタスクディスパッチャの中から呼び出され、CPUのハードウェアを低消費電力モードに移行する処理を行う。

低消費電力モードに移行した後は、low_pow() の処理の中で外部割込みの発生を待ち、外部割込みが発生したら、CPU周辺のハードウェアを通常モード(低消費電力ではないモード)に戻す処理を行ってから、low_pow() の呼出元に復帰する。

low_pow() の具体的な処理手順は以下の通りである。

1. CPUを低消費電力モードへ移行する。たとえば、クロック周波数を下げる。
2. 外部割込みの発生を待ってCPUを停止する。たとえば、そのような機能を持つCPU命令を実行する。
3. 外部割込みの発生によりCPUが実行を再開する。(ハードウェアによる処理)
4. CPUを通常モードに戻す。たとえば、通常のクロック周波数に戻す。
5. 呼出元に復帰する。呼出元は、実際にはT-Kernel内部のディスパッチャである。

low_pow() の実装にあたっては、以下の点に注意する必要がある。

- 割込み禁止状態で呼び出される。
- 割込みを許可してはいけない。
- 処理速度が割込み応答速度に影響するので、できる限り高速であることが要求される。

補足事項

タスクディスパッチャは、実行すべきタスクが無くなった場合に、消費電力を下げる目的で low_pow() を呼び出す。

5.6.2 off_pow - システムをサスペンド状態に移行

C言語インタフェース

void off_pow (void);

パラメータ

なし

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
×	×	×

解説

T-Kernelの中から、powmode＝TPW_DOSUSPEND を指定した tk_set_pow() の処理中に呼び出され、CPU周辺のハードウェアをサスペンド状態(電源の切れた状態)に移行する処理を行う。

サスペンド状態に移行した後は、off_pow() の処理の中でリジューム要因(電源再投入など)の発生を待ち、リジューム要因が発生したら、サスペンド状態を解除して off_pow() の呼出元に復帰する。

off_pow() の具体的な処理手順は以下の通りである。

1. CPUをサスペンド状態へ移行し、リジューム要因の発生を待つ。たとえば、クロックを停止する
2. リジューム要因の発生によりCPUが実行を再開する。(ハードウェアによる処理)
3. 必要に応じて、CPUやハードウェアの状態を通常の状態に戻す。サスペンド状態を解除する。(前項と一緒にハードウェアで処理される場合もある)
4. 呼出元に復帰する。呼出元は、実際にはT-Kernel内部の tk_set_pow() の処理部分である。

off_pow() の実装にあたっては、以下の点に注意する必要がある。

- 割込み禁止状態で呼び出される。
- 割込みを許可してはいけない。

なお、周辺機器など各デバイスは、デバイスドライバによってサスペンド状態への移行および復帰を行う。詳細は tk_sus_dev() を参照。

5.7 システム構成情報管理機能

システム構成情報管理機能は、システム構成に関する各種の情報を保持・管理するための機能である。

システム構成情報の一部は標準定義として定めてあり、その中で最大タスク数やタイマ割込み間隔などの情報を定義するが、それ以外にもアプリケーションやサブシステム、デバイスドライバで任意に定義した情報を追加して利用できる。

システム構成情報の形式は、名称と定義データの組である。

名称
: 最大16文字の文字列で表される。
: 使用可能な文字(UB) a～z A～Z 0～9 _

定義データ
: 数値(整数)の並びまたは文字列により定義される。
: 使用可能な文字(UB) 0x00～0x1F,0x7F,0xFF(文字コード)を除く文字

例 5.5 システム構成情報の形式の例

名称	定義データ
SysVer	1 0
SysName	T-Kernel Version 1.00

システム構成情報をどの様に保持するかについては規定しないが、一般的にはメモリ(ROM/RAM)に保持する。したがって、あまり大量の情報を保持する目的には使用しない。

システム構成情報は、tk_get_cfn および tk_get_cfs によって取得できる。

なお、システムの実行中にシステム構成情報を追加・変更する機能はない。

5.7.1 システム構成情報の取得

システム構成情報を取得するための拡張SVCとして、tk_get_cfn と tk_get_cfs がある。これらの機能は、アプリケーションやサブシステム、デバイスドライバなどから利用可能であるほか、T-Kernelの内部でも利用している。T-Kernel内部での利用は拡張SVCを経由しない方法でもよく、実装定義とする。

5.7.1.1 tk_get_cfn - システム構成情報から数値列取得

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT ct = tk_get_cfn (CONST UB *name , INT *val , INT max);

パラメータ

CONST UB*	name	Name	名称
INT*	val	Value	数値列を格納する配列
INT	max	Maximum Count	val の配列の要素数

リターンパラメータ

INT	ct	Defined Numeric Information Count	定義されている数値情報の個数
	または	Error Code	エラーコード

エラーコード

E_NOEXS	name の名称で情報が定義されていない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

システム構成情報から数値列の情報を取得する。name の名称で定義されている数値列の情報を、最大 max 個まで取得し val へ格納する。戻値に定義されている数値列情報の数を返す。戻値 > max であれば、すべての情報を格納しきれていないことを示す。max＝0を指定することで、val に格納せずに数値列の数を知ることができる。

name の名称の情報が定義されていないときは E_NOEXS を返す。name の名称で定義されている情報が文字列であった場合の動作は不定である。

この機能は、T-Kernel/OSのシステムコールの呼出可能な保護レベルの制限を受けずに任意の保護レベルから呼び出すことができる。

5.7.1.2 tk_get_cfs - システム構成情報から文字列取得

C言語インタフェース

```
#include <tk/tkernel.h>
```

INT rlen = tk_get_cfs (CONST UB *name , UB *buf , INT max);

パラメータ

CONST UB*	name	Name	名称
UB*	buf	Buffer	文字列を格納する配列
INT	max	Maximum Length	buf の最大長(バイト数)

リターンパラメータ

INT	rlen	Size of Defined Character String Information	定義されている文字列情報の長さ(バイト数)
	または	Error Code	エラーコード

エラーコード

E_NOEXS	name の名称で情報が定義されていない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

システム構成情報から文字列の情報を取得する。name の名称で定義されている文字列の情報を、最大 max 文字まで取得し、buf へ格納する。格納した文字列が max 文字未満であれば、最後に'¥0'を格納する。戻値に定義されている文字列情報の長さ('¥0'は含まない)を返す。戻値 > max であれば、すべての情報を格納しきれていないことを示す。max＝0を指定することで、buf に格納せずに文字列の長さを知ることができる。

name の名称で情報が定義されていないときは E_NOEXS を返す。name の名称で定義されている情報が数値列であった場合の動作は不定である。

この機能は、T-Kernel/OSのシステムコールの呼出可能な保護レベルの制限を受けずに任意の保護レベルから呼び出すことができる。

5.7.2 標準システム構成情報

下記の定義情報を標準として定める。標準定義の名称はTを前置する。

文字列	説明
N	数値列情報
S	文字列情報

・ 製品情報

文字列	標準定義の名称	説明
S	TSysName	システム名称(製品名称)

・ 各オブジェクトの最大数

文字列	標準定義の名称	説明
N	TMaxTskId	最大タスク数
N	TMaxSemId	最大セマフォ数
N	TMaxFlgId	最大イベントフラグ数
N	TMaxMbxId	最大メールボックス数
N	TMaxMtxId	最大ミューテックス数
N	TMaxMbfId	最大メッセージバッファ数
N	TMaxPorId	最大ランデブポート数
N	TMaxMpfId	最大固定長メモリプール数
N	TMaxMplId	最大可変長メモリプール数
N	TMaxCycId	最大周期起動ハンドラ数
N	TMaxAlmId	最大アラームハンドラ数
N	TMaxResId	最大リソースグループ数
N	TMaxSsyId	最大サブシステム数
N	TMaxSsyPri	最大サブシステム優先度数

・ その他

文字列	標準定義の名称	説明
N	TSysStkSz	デフォルトシステムスタックサイズ(バイト数)
N	TSVCLimit	システムコールの呼出可能な最も低い保護レベル
N	TTimPeriod	タイマ割込み間隔(ミリ秒単位)タイマ割込み間隔(マイクロ秒単位)

ミリ秒単位の時間とマイクロ秒単位の時間を合わせた時間が、実際のタイマ割込み間隔の時間となる。マイクロ秒単位の時間を省略した場合は0とみなされる。

たとえば、タイマ割込み間隔を5ミリ秒とする場合は、"TTimPeriod 5" あるいは "TTimPeriod 0 5000" と記述する。タイマ割込み間隔を1.5ミリ秒(1500マイクロ秒)とする場合は、"TTimPeriod 1 500" あるいは "TTimPeriod 0 1500" と記述する。

・ デバイス管理

文字列	標準定義の名称	説明
N	TMaxRegDev	最大デバイス登録数
N	TMaxOpnDev	最大デバイスオープン数
N	TMaxReqDev	最大デバイスリクエスト数
N	TDEvtMbfSz	事象通知用メッセージバッファのサイズ(バイト数)事象通知のメッセージ最大長(バイト数)

TDEvtMbfSz が定義されなかった場合、またはメッセージバッファのサイズに負数が設定された場合は、事象通知用メッセージバッファを使用しない。

上記の数値列情報で複数の数値が定義されるものは、説明の順序と同じ順序で配列に格納される。

例 5.6 複数の数値の格納順の例

```
tk_get_cfn("TDEvtMbfSz", val, 2)
```

val[0] = 事象通知用メッセージバッファのサイズ
val[1] = 事象通知のメッセージ最大長

T-Kernel 1.0との差異
TTimPeriod の2個目の要素として、マイクロ秒単位の設定情報を追加した。

5.8 メモリキャッシュ制御機能

メモリキャッシュ制御機能は、キャッシュの制御やモード設定を行うための機能である。

T-Kernelにおけるキャッシュについての考え方は、以下の通りである。

基本的には、アプリケーションもデバイスドライバも、キャッシュの存在を意識することなくプログラムしていれば、自動的に適切なキャッシュ制御が行われるべきである。特に、プログラムの移植性を考慮すると、キャッシュのようにシステムへの依存性が強いものは、できるだけアプリケーションプログラムと切り離して扱えるのがよい。そのため、T-Kernelを使った実際の個々のシステムでは、T-Kernel自身が自動的にキャッシュの制御を行う方針となっている。

具体的には、通常のプログラムやデータを入れるメモリ等の領域はキャッシュON、I/Oなどの領域はキャッシュOFFとなるようにT-Kernelがキャッシュの設定を行う。したがって、通常のアプリケーションプログラムが明示的にキャッシュ制御の関数を呼ぶ必要はない。プログラムから意識的にキャッシュ制御を行わなくても、適切なキャッシュ制御が自動的に行われる。

ただし、T-Kernelによるキャッシュ制御(いわば、デフォルト設定によるキャッシュ制御)だけでは、適切な対応ができない場合もある。たとえば、DMA転送の絡んだ入出力処理を行う場合や、カーネル管理外のメモリ領域を使う場合には、明示的なキャッシュ制御が必要となることがある。また、プログラムを動的にロードあるいは生成(コンパイル)しながら実行するような場合には、データキャッシュと命令キャッシュを適切に同期させるためのキャッシュ制御が必要となることがある。このようなケースで利用することを想定した機能が、メモリキャッシュ制御機能である。

T-Kernel 1.0との差異
T-Kernel 2.0で追加された機能である。

5.8.1 SetCacheMode - キャッシュモードの設定

C言語インタフェース

```
#include <tk/tkernel.h>

INT rlen = SetCacheMode (void *addr , INT len , UINT mode );
```

パラメータ

void*	addr	Start Address	先頭アドレス
INT	len	Length	領域サイズ(バイト数)
UINT	mode	Mode	キャッシュモード

リターンパラメータ

INT	rlen	Result Length	キャッシュモードを設定できた領域のサイズ(バイト数)
	または	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(addr, len, mode が不正あるいは利用できない)
E_NOSPT	未サポート機能(mode で指定した機能が未サポート)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

メモリ領域のキャッシュモードを設定する。具体的には、addr から len バイトのメモリ領域のキャッシュに対して、mode で指定した設定を行う。メモリのキャッシュモードの設定はページ単位で行われる。

```
mode := ( CM_OFF || CM_WB || CM_WT ) | [CM_CONT]
                CM_OFF   キャッシュオフ
                CM_WB    キャッシュオン(ライトバック)
                CM_WT    キャッシュオン(ライトスルー)
                CM_CONT 物理アドレスが連続した領域のみキャッシュ設定を行う
                   ...
                /* 実装独自のモードを追加してもよい */
```

mode に CM_OFF を指定した場合は、キャッシュをフラッシュ(書き戻し)した後に無効化して、キャッシュモードをオフに設定する。

mode に CM_WT を指定した場合は、キャッシュをフラッシュした後にキャッシュモードをライトスルーに設定する。

mode に CM_WB を指定した場合は、キャッシュモードをライトバックに設定する。この時にキャッシュフラッシュを行うかどうかは実装依存とする。

mode に CM_CONT を指定した場合は、addr から物理アドレスが連続した領域のみキャッシュモード設定を行う。指定した領域内の物理アドレスが不連続である場合、およびページアウトしている領域が存在する場合は不連続となるアドレスの直前で処理を中止し、処理が完了した領域のサイズを返す。CM_CONT を指定しなかった場合は、指定した全ての領域のキャッシュを処理し、処理が完了した領域のサイズを返す。

CPUや実装によっては、キャッシュモードの一部、あるいは全部の設定が不可能な場合がある。設定不可能なモードを指定した場合は、何も処理せず E_NOSPT を返す。

len は1以上とする。0以下の数を指定した場合はエラーコード E_PAR を返す。

補足事項

キャッシュモードの設定はページ単位で行われるため、addr がページ境界ではない場合には、addr が含まれるページの先頭アドレス以降が設定対象となる。また、隣接した領域に対しても、意図しないキャッシュアクセスが発生する可能性があるので、使用には注意が必要である。ページのサイズは実装依存であり、GetSpaceInfo により取得することができる。

ハードウェア構成やCPUの持つキャッシュの機能に依存して、さらに詳細なキャッシュモードの設定を行いたい場合は、実装独自の mode を追加して利用する。たとえば、NORMAL CACHE OFF (Weakly Order), DEVICE CACHE OFF (Weakly Order), STRONG ORDER などのキャッシュモードを指定可能な場合がある。

利用できない mode を指定した場合のエラーが E_NOSPT となるか E_PAR となるかは実装依存である。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

T-Kernel 1.0では、物理アドレスによるDMA転送を行う際に利用するAPIとして CnvPhysicalAddr があり、(a)論理アドレスから物理アドレスへの変換、(b)DMA転送の前処理としてのキャッシュ書き戻し、(c)DMA転送バッファ領域のキャッシュ無効化、の3つの処理をこの1つのAPIで行っていた。しかし、状況によっては、これら3つの機能の一部が不要な場合があり、必要な機能のみを個別に呼び出せる方が効率の良いことがある。また、他OS用のデバイスドライバ等において、(a)(b)(c)の機能が独立して用意されていることを前提としている場合があり、それをT-Kernel上に移植するには、(a)(b)(c)の機能を個別に呼び出せる方が都合がよい。したがってT-Kernel 2.0では、CnvPhysicalAddr の3つの機能を分離独立させる形で、アドレス空間情報の取得(GetSpaceInfo)、キャッシュモードの設定(SetCacheMode)、キャッシュの制御(ControlCache)の3つのAPIを追加した。

5.8.2 ControlCache - キャッシュの制御

C言語インタフェース

```
#include <tk/tkernel.h>

INT rlen = ControlCache (void *addr , INT len , UINT mode );
```

パラメータ

void*	addr	Start Address	先頭アドレス
INT	len	Length	領域サイズ(バイト数)
UINT	mode	Mode	制御モード

リターンパラメータ

INT	rlen	Result Length	キャッシュモードを設定できた領域のサイズ(バイト数)
	または	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(addr, len, mode が不正)
E_NOSPT	未サポート機能(mode で指定した機能が未サポート)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

メモリ領域のキャッシュ制御(フラッシュあるいは無効化)を行う。具体的には、論理アドレス addr から len バイトのメモリ領域のキャッシュに対して、mode で指定した制御を行う。

```
mode := (CC_FLUSH | CC_INVALIDATE)
            CC_FLUSH        キャッシュのフラッシュ(書き戻し)
            CC_INVALIDATE   キャッシュの無効化
                ...
            /* 実装独自のモード値を追加してもよい */
```

CC_FLUSH と CC_INVALIDATE は同時に指定可能である。その場合はキャッシュをフラッシュした後に無効化する。

処理が成功すると、処理を行った領域のサイズを返す。指定した領域内にページアウトしている領域が存在する場合はその領域の直前で処理を中止し、処理が完了した領域のサイズを返す。

キャッシュモードや属性の異なる領域を跨ぐ範囲を指定してはいけない。例えば、キャッシュオンの領域とキャッシュオフの領域、タスク固有空間と共有空間、保護レベルの異なる領域のような異なる属性の領域の境界を跨ぐ範囲を指定してはいけない。このような範囲を指定した場合の動作は保証されない。

キャッシュ制御はハードウェアに依存する部分が多いため、CPUやハードウェア、実装によってその機能の詳細は異なる。指定された領域を指定されたモードで制御するのが基本だが、指定された領域を含めて、より多くの領域に影響を与える場合がある。例えば、以下のような場合がある。

- 正確な指定範囲のみを制御(フラッシュあるいは無効化)するとは限らない。指定範囲を含む領域が制御されるが、CPUやハードウェア、実装によっては、それ以外の領域(例えば全メモリ)に対しても、フラッシュあるいは無効化が行われる場合がある。
- キャッシュオフの領域を指定した場合は、通常は何も行わない。しかしこのケースにおいても、指定範囲以外の領域のキャッシュのフラッシュあるいは無効化が行われる可能性がある。(常に全領域をフラッシュする処理を行うなど)
- キャッシュのないシステムでは、何も行わない。

一般に、キャッシュの制御はキャッシュラインサイズ単位で行われる。このため隣接した領域に意図しないキャッシュアクセスが発生する可能性があるので、使用には注意が必要である。キャッシュラインサイズは実装依存であり、GetSpaceInfo で取得することができる。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.9 物理タイマ機能

物理タイマ機能は、複数のハードウェアタイマが使えるシステムにおいて、タイマ割込み間隔(TTimPeriod)よりも細かい単位の時間経過を条件とした処理を行う場合に有効な機能である。

物理タイマとは、一定の時間間隔で、0から1つずつカウント値が単調増加していくハードウェアのカウンタである。カウント値が、物理タイマごとに指定された特定の値(上限値)に達すると、物理タイマ毎に指定されたハンドラ(物理タイマハンドラ)を起動するとともに、カウント値が0に戻る。

システムで利用可能なハードウェアタイマの数に応じて、複数個の物理タイマが利用できる。利用可能な物理タイマの個数は実装依存である。通常のT-Kernelの実装では、時間管理機能を実現するためにハードウェアタイマを1つ使用するので、残りのハードウェアタイマを物理タイマ機能で使用することを想定している。

物理タイマ番号としては、1, 2, ...のように正の整数を小さい方から使う。たとえば、4個のハードウェアタイマがある場合、このうちの1個はT-Kernelの時間管理機能で使うため、残り3個の物理タイマが利用でき、物理タイマ番号は1, 2, 3となる。

T-Kernel/SMの物理タイマ機能において、個々の物理タイマと、それを使用するタスク等との対応を管理することはない。複数のタスク等が1つの物理タイマを共用したい場合は、アプリケーション側で排他制御などの調整を行う必要がある。

補足事項
T-Kernelの時間管理機能では、「5.7.2節「標準システム構成情報」」の「タイマ割込み間隔」(TTimPeriod)で指定された時間間隔で起動されるハンドラの中で、カーネルがアラームハンドラや周期ハンドラの起動、タイムアウトの処理など、複数の要求に対する処理を行っている。これに対して、物理タイマ機能は、ハードウェアタイマの設定、カウント値の読み出し、割込み発生といったプリミティブな機能を標準化して提供するだけのものであり、時間管理機能のように複数の要求を処理するわけではない。こういった点から、従来の時間管理機能よりも抽象度が低く、ハードウェアの階層に近い機能であるという意味で、「物理タイマ(Physical Timer)」の名称とした。
上記のような位置付けから、物理タイマ機能はできるだけシンプルで最小限の仕様とし、オーバーヘッドの少ないライブラリ関数で実現することを想定している。動的なID番号ではなく静的に固定された物理タイマ番号を使う仕様や、要求元タスクとの対応の管理や複数タスクからの要求の調整を一切行わないといった仕様も、この方針を反映したものである。
物理タイマの関数は、タイマ(カウンタ)というデバイスを操作するためのAPIを標準化したものである。ただし、タイマというデバイスでは、細かい時間経過に応じて割込みハンドラを呼び出すなど、時間関連の動作に直接関わる部分が多く、この点で他のデバイス(ストレージや通信など)よりもカーネルとの結び付きが強い。こういった理由から、タイマについては、デバイスドライバの仕様として標準化するのではなく、T-Kernel/SMの一部として仕様を標準化し、より汎用性の高い機能として提供する。
物理タイマ機能はT-Kernel/SMの機能なので、T-Kernel/SMの【全般的な注意・補足事項】が適用される。
物理タイマとして利用するハードウェアタイマは、32ビット以下を想定している。そのため、カウント値や上限値を表すデータタイプは32ビットのUWを使っている。将来、64ビットの関数を追加することは可能である。

仕様決定の理由
T-Kernel 2.0において時間管理の機能を強化するとともに、最近の組込み向けマイクロコンピュータやSoC(System on a Chip)に複数個の実装がみられるハードウェアタイマを有効活用し、これらのタイマを操作するプログラムの移植性を高める目的で、T-Kernel/SMに物理タイマの機能を導入した。

T-Kernel 1.0との差異
T-Kernel 2.0で追加された機能である。

5.9.1 物理タイマのユースケース

物理タイマ機能の有効性が高い例を以下に示す。

(a)実現すべき処理の例

2500マイクロ秒毎に行うべき周期的な処理Xと、1800マイクロ秒毎に行うべき周期的な処理Yがあるとする。これを、物理タイマで効率よく実現する。

(b)物理タイマ機能を使った実現方法

2つの物理タイマを使い、その一方は、ちょうど2500マイクロ秒毎に物理タイマハンドラが起動されるように設定する。

たとえば、物理タイマのクロック周波数が10MHzであれば、1クロックが0.1マイクロ秒(＝100ナノ秒)なので、物理タイマの上限値(`limit`)として24999(＝25000-1)を設定し、カウント値が24999から0になる時に物理タイマハンドラが起動されるようにする。

周期的に繰り返す処理なので、StartPhysicalTimer の `mode` では `TA_CYC_PTMR` を指定する。

この物理タイマハンドラの中で処理Xを行う。

これと同様に、もう一方の物理タイマを使って、ちょうど1800マイクロ秒毎に起動される物理タイマハンドラを設定し、この中で処理Yを行う。

T-Kernelの時間管理機能が使うタイマ割込み間隔(`TTimPeriod`)は、物理タイマ機能とは無関係なので、デフォルト値(10ミリ秒)のままでよい。

(c)物理タイマ機能を使わない実現方法

物理タイマハンドラの代わりに、マイクロ秒が指定可能なT-Kernel 2.0のシステムコール(tk_cre_cyc_u)を使って、2500マイクロ秒毎に起動される周期ハンドラを定義し、その中で処理Xを行う。また、1800マイクロ秒毎に起動される周期ハンドラを定義し、その中で処理Yを行う。

しかし、この場合には、2500マイクロ秒毎および1800マイクロ秒毎という時間がいずれも正確に処理されるように、T-Kernelの時間管理機能が使うタイマ割込み間隔を十分に短く設定する必要がある。具体的には、2500マイクロ秒と1800マイクロ秒の公約数である100マイクロ秒のタイマ割込み間隔とすることにより、2500マイクロ秒毎の処理も、1800マイクロ秒毎の処理も、ほぼ正確な時間間隔で実現できる。

物理タイマ機能を使った(b)の方法であれば、T-Kernelの時間管理機能は使わないので、タイマ割込み間隔はデフォルト(10ミリ秒毎)のままでよい。このほかに、物理タイマによる割込みが2500マイクロ秒毎および1500マイクロ秒毎に入り、その中から呼ばれる物理タイマハンドラの中で処理Xや処理Yを行うことになるが、これら以外に時間に関係した無駄な割込みが入ることはない。

一方、物理タイマ機能を使わない(c)の方法では、タイマ割込み間隔を短くする必要があるため、タイマ割込みの回数が増え、その分のオーバーヘッドが増加する。たとえば、10ミリ秒の間に入ってくる時間関係の割込みの回数で比較すると、(b)では時間管理機能のためのタイマ割込みが1回(＝10ミリ秒÷10ミリ秒)、処理Xのための物理タイマの割込みが4回(＝10ミリ秒÷2500マイクロ秒)、処理Yのための物理タイマの割込みが6回(＝10ミリ秒÷1500マイクロ秒)、合計11回であるのに対し、(c)では時間管理機能のためのタイマ割込みが100回(＝10ミリ秒÷100マイクロ秒)となる。時間の正確さとのトレードオフになるが、処理Xと処理Yの周期や位相の差によっては、タイマ割込み間隔をさらに短くする必要があり、もっと大きなオーバーヘッドを生じる可能性もある。このようなケースにおいて、物理タイマ機能の有効性が高い。

ただし、物理タイマ機能が有効なのは、時間に依存した処理の数が少なくて静的に決まっており、それに対して十分な数のハードウェアタイマが存在する場合である。物理タイマ機能は、文字通り、物理的なハードウェア資源の制約を受ける機能なので、ハードウェアタイマが少なければ、物理タイマ機能が十分に使えない。また、時間に依存した処理が動的に増えてくるようなケースには対応しにくい。そのようなケースでは、周期ハンドラやアラームハンドラなど従来の時間管理機能を使う方が、柔軟な対応が可能である。

マイクロ秒単位の時間管理機能と、物理タイマの機能は、用途の重複する面もあるが、上記のように特性の異なる面もあるため、ハードウェア構成やアプリケーションに応じて適切な方を利用できるとよい。物理タイマの機能は、このような理由で追加されたものである。

5.9.2 StartPhysicalTimer - 物理タイマの動作開始

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = StartPhysicalTimer (UINT ptmrno , UW limit , UINT mode);

パラメータ

UINT	ptmrno	Physical Timer Number	物理タイマ番号
UW	limit	Limit	上限値
UINT	mode	Mode	動作モード

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(ptmrno, limit, mode が不正あるいは利用できない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ptmrno で指定された物理タイマのカウント値を0とした後、カウントを開始する。本関数の実行後は、タイマのクロック周波数の逆数の時間間隔ごとに、カウント値が1ずつ増加する。

limit ではカウント値の上限値を指定する。カウント値が上限値に達した後、さらにクロック周波数の逆数の時間が経つと、カウント値は0に戻る。この時、この物理タイマに対する物理タイマハンドラが定義されていた場合には、そのハンドラが起動される。StartPhysicalTimer の発行により物理タイマがカウントを始めてから、次にカウント値が0になるまでの時間は、(クロック周波数の逆数の時間)×(上限値+1)である。

limit が0の場合は E_PAR のエラーになる。

mode では次の指定を行う。

TA_ALM_PTMR	0	カウント値が上限値から0に戻った後は、カウントを停止する。その後のカウント値は0のままとなる。
TA_CYC_PTMR	1	カウント値が上限値から0に戻った後も、再度カウント値の増加を続ける。したがって、カウント値は周期的な増加を繰り返す。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.9.3 StopPhysicalTimer - 物理タイマの動作停止

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = StopPhysicalTimer (UINT ptmrno);

パラメータ

UINT	ptmrno	Physical Timer Number	物理タイマ番号

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(ptmrno が不正あるいは利用できない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ptmrno で指定された物理タイマのカウントを停止する。

本関数の実行後、物理タイマのカウント値はそのまま保持される。すなわち、本関数の実行後に GetPhysicalTimerCount を実行した場合には、本関数実行直前の物理タイマのカウント値が返る。

既にカウントを停止している物理タイマに対して、本関数を実行しても、何も起こらない。エラーにもならない。

補足事項

使用の終わった物理タイマを動かしたままにしておいた場合、プログラムの動作上の不都合はなくても、クロックを無駄に使い、省電力などの面で望ましくない可能性がある。そのため、使用していない物理タイマに対しては、本関数を実行し、カウントを停止しておく方が望ましい。

本関数の発行が有効なのは、StartPhysicalTimer の mode として TA_CYC_PTMR を指定した物理タイマの使用が終了した場合である。mode として TA_ALM_PTMR を指定した場合は、カウント値が上限値から0に戻った後で自動的にカウントを停止し、本関数を実行した後と同じ状態になる。この場合は、本関数を別途発行する必要はない。発行しても問題はないが、何も変化しない。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.9.4 GetPhysicalTimerCount - 物理タイマのカウント値取得

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = GetPhysicalTimerCount (UINT ptmrno , UW *p_count);

パラメータ

UINT	ptmrno	Physical Timer Number	物理タイマ番号
UW*	p_count	Pointer to Physical Timer Count	物理タイマの現在カウント値を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード
UW	count	Physical Timer Count	現在カウント値

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(ptmrno が不正あるいは利用できない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ptmrno で指定された物理タイマの現在のカウント値を取得し、リターンパラメータ count として返す。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.9.5 DefinePhysicalTimerHandler - 物理タイマハンドラ定義

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = DefinePhysicalTimerHandler (UINT ptmrno , CONST T_DPTMR *pk_dptmr);

パラメータ

UINT	ptmrno	Physical Timer Number	物理タイマ番号
CONST T_DPTMR*	pk_dptmr	Packet to Define Physical Timer Handler	物理タイマハンドラ定義情報

pk_dptmr の内容

void*	exinf	Extended Information	拡張情報
ATR	ptmratr	Physical Timer Attribute	物理タイマハンドラ属性(TA_ASM || TA_HLNG)
FP	ptmrhdr	Physical Timer Handler Address	物理タイマハンドラアドレス

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_RSATR	予約属性(ptmratr が不正あるいは利用できない)
E_PAR	パラメータエラー(ptmrno, pk_dptmr, ptmrhdr が不正あるいは利用できない、ptmrno に対する物理タイマハンドラを定義できない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

pk_dptmr が NULL でない場合は、ptmrno で指定された物理タイマに対する物理タイマハンドラを定義する。物理タイマハンドラは、タスク独立部として実行されるハンドラであり、物理タイマのカウント値が StartPhysicalTimer の limit で指定された上限値から0に戻る時に起動される。

物理タイマハンドラのプログラム形式は、周期ハンドラやアラームハンドラと同様である。すなわち、TA_HLNG 属性が指定された場合は、高級言語対応ルーチンを経由して物理タイマハンドラが起動され、関数からのリターンによって終了する。TA_ASM 属性が指定された場合の物理タイマハンドラの形式は実装依存である。いずれの属性を指定した場合も、物理タイマハンドラの起動時のパラメータとして exinf を渡す。

pk_dptmr が NULL の場合は、ptmrno で指定された物理タイマに対する物理タイマハンドラの定義を解除する。システム起動直後は、すべての物理タイマに対する物理タイマハンドラの定義が解除された状態となっている。

ptmrno で指定された物理タイマに対する物理タイマハンドラを定義できない場合(GetPhysicalTimerConfig の pk_rptmr->defhdr で FALSE を返す場合)は、E_PAR のエラーになる。ptmrno で指定された番号を持つ物理タイマが存在しない場合や、利用できない場合にも、E_PAR のエラーになる。

補足事項

実装上は、物理タイマの機能を実現するための割込みハンドラを T-Kernel/SMの内部で定義し、物理タイマのカウント値が上限値から0に戻る時に、この割込みハンドラが起動されるように設定しておく。この割込みハンドラの中では、本関数で定義された物理タイマハンドラを呼び出すとともに、物理タイマの実装に関わる処理(たとえば、TA_ALM_PTMR と TA_CYC_PTMR に関する処理など)を行う。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.9.6 GetPhysicalTimerConfig - 物理タイマのコンフィグレーション情報取得

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = GetPhysicalTimerConfig (UINT ptmrno , T_RPTMR *pk_rptmr);

パラメータ

UINT	ptmrno	Physical Timer Number	物理タイマ番号
T_RPTMR*	pk_rptmr	Packet to Refer Physical Timer	物理タイマのコンフィグレーション情報を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rptmr の内容

UW	ptmrclk	Physical Timer Clock Frequency	物理タイマのクロック周波数
UW	maxcount	Maximum Count	最大カウント値
BOOL	defhdr	Handler Support	物理タイマハンドラサポートの有無

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー(ptmrno, pk_rptmr が不正あるいは利用できない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

ptmrno で指定された物理タイマに関するコンフィグレーション情報を取得する。

取得可能なコンフィグレーション情報には、物理タイマのクロック周波数 ptmrclk、最大カウント値 maxcount、物理タイマハンドラサポートの有無 defhdr がある。

ptmrclk は、対象物理タイマのカウントアップに使用されるクロックの周波数である。ptmrclk が1の場合はクロックが1Hz、ptmrclk が (2 ^ 32-1) の場合は (2 ^ 32-1)Hz、すなわち約4GHzである(' ^ 'はべき乗を表す)。1Hz未満の長いクロックの場合は、ptmrclk が0となる。ptmrclk が0以外の場合は、ptmrclk の逆数の時間間隔ごとに、物理タイマのカウント値が0から上限値 limit の値まで単調増加していく。

maxcount は、対象物理タイマでカウント可能な最大値であり、上限値として設定可能な最大値でもある。一般的には、16ビットのタイマカウンタの場合は maxcount が(2 ^ 16-1)、32ビットのタイマカウンタの場合は maxcount が(2 ^ 32-1)となるが、ハードウェアやシステム構成によっては、それ以外の値をとる場合もある。

defhdr が TRUE の場合は、対象物理タイマのカウント値が上限値に達した時に起動される物理タイマハンドラを定義できる。defhdr が FALSE の場合は、この物理タイマに対する物理タイマハンドラを定義することはできない。

ptmrno で指定された番号を持つ物理タイマが存在せず、利用できない場合には、E_PAR のエラーになる。物理タイマ番号は正の整数を小さい方から使っていくため、N個の物理タイマが使えるシステムでは、ptmrno が0または(N+1)以上の場合に E_PAR のエラーになる。

補足事項

本関数の「コンフィグレーション」の名称にも見られるとおり、本関数で取得する ptmrclk, maxcount, defhdr の情報は、ハードウェアあるいはシステム起動時の設定により静的に固定されたものであり、システム動作中の変更がないことを想定している。しかしながら、将来あるいは実装依存の追加機能として、物理タイマのコンフィグレーション、たとえばクロック周波数を積極的に設定あるいは変更する機能を導入する可能性はあり、その場合は、本関数で取得する情報についても、システム動作中に変化する動的な情報となる。このような使い方の差異は、運用や用途に依存する部分が大きく、T-Kernelの仕様として決めるよりは、物理タイマを使った上位のライブラリ等の中で吸収するのがよいと考えられる。したがって、T-Kernelの仕様としては、本関数で取得するコンフィグレーション情報がシステムの動作中に変化する可能性について、特に定めないことにする。すなわち、本関数で取得した情報が途中で変化する可能性があるかどうかは、実装依存である。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.10 ユーティリティ機能

ユーティリティ機能は、T-Kernel上のアプリケーション、ミドルウェア、デバイスドライバなどプログラム全般から利用される共通性の高い機能である。

ユーティリティ機能は、ライブラリ関数またはC言語のマクロで提供される。

T-Kernel 1.0との差異
T-Kernel 2.0で追加された機能である。

5.10.1 オブジェクト名設定

オブジェクト名設定のAPIは、C言語のマクロで提供され、タスク独立部およびディスパッチ禁止・割込み禁止の状態から呼び出すことができる。

5.10.1.1 SetOBJNAME - オブジェクト名設定

C言語インタフェース

```
#include <tk/tkernel.h>
```

void SetOBJNAME (void *exinf , CONST UB *name);

パラメータ

void*	exinf	Extended Information	拡張情報を設定する変数
CONST UB*	name	Object Name	設定するオブジェクト名

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

name で指定される4文字以下のASCII文字列を、1つの32ビットデータと解釈して、exinf に格納する。

本APIはC言語のマクロで定義されており、exinf はポインタではない。変数を直接記載する。

補足事項

本APIを使うことにより、T-Kernelの各オブジェクトの拡張情報 exinf の中に、個々のオブジェクトに対するASCII文字列の名称(タスク名など)を設定することができる。デバッガ等でオブジェクトの状態を表示する際、exinf の値をASCII文字列として表示することにより、本APIによって設定されたオブジェクトの名称を表示できる。

例 5.7 SetOBJNAMEの使用例

```
T_CTSK  ctsk;
...
/* ctskのタスクに "TEST" のオブジェクト名を設定 */
SetOBJNAME(ctsk.exinf, "TEST");
task_id = tk_cre_tsk ( &ctsk );
```

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.10.2 高速ロック・マルチロックライブラリ

高速ロック・マルチロックライブラリは、デバイスドライバやサブシステムの中において、複数タスク間の排他制御をより高速に行うためのライブラリである。排他制御を行うには、セマフォやミューテックスを使うこともできるが、高速ロックはT-Kernel/SMのライブラリ関数として実装されており、待ちに入らない場合のロック獲得の操作を特に高速に処理する。

高速ロック・マルチロックライブラリのうち、高速ロックは、セマフォやミューテックスよりも高速な排他制御用バイナリセマフォである。一方の高速マルチロックは、独立した排他制御用バイナリセマフォを32個あわせて一つのオブジェクトにしたものであり、0番～31番のロック番号で区別する。

たとえば10箇所で排他制御を行う場合、10個の高速ロックを使う方法でもよいが、1個の高速マルチロックを生成し、その中の0番～9番を使って排他制御を行う方法も可能である。前者の方がより高速になるが、必要リソースの合計は後者の方が少なくて済む。

補足事項
高速ロックの機能は、ロックの状態を示すカウンタとセマフォにより実装される。また、高速マルチロックの機能は、ロックの状態を示すカウンタとイベントフラグにより実装される。ロック獲得時に待ちに入らない場合は、カウンタ操作のみを行うため、通常のセマフォやイベントフラグよりも高速に処理される。一方、ロック獲得時に待ちに入る場合は、通常のセマフォやイベントフラグの機能を使って待ち状態への移行や待ち行列の管理を行うため、セマフォやイベントフラグより高速というわけではない。高速ロック・マルチロックの機能が有効なのは、排他制御の際に待ちに入る可能性が低い場合である。

T-Kernel 1.0との差異
T-Kernel 2.0で追加された機能である。

5.10.2.1 CreateLock - 高速ロックの生成

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = CreateLock (FastLock *lock , CONST UB *name);

パラメータ

FastLock*	lock	Control Block of FastLock	高速ロックの管理ブロック
CONST UB*	name	Name of FastLock	高速ロックの名前

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_LIMIT	高速ロックの数がシステムの上限を超えた

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

高速ロックを生成する。

lock は高速ロックの管理のための構造体である。name は高速ロックに付ける名前であるが、NULL でもよい。

高速ロックは排他制御のためのバイナリセマフォであり、なるべく高速に操作できるように実装されている。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.10.2.2 DeleteLock - 高速ロックの削除

C言語インタフェース

```
#include <tk/tkernel.h>
```

void DeleteLock (FastLock *lock);

パラメータ

FastLock*	lock	Control Block of FastLock	高速ロックの管理ブロック

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

高速ロックを削除する。
高速化のため、エラーの検出は行わない。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.10.2.3 Lock - 高速ロックのロック操作

C言語インタフェース

```
#include <tk/tkernel.h>
```

void Lock (FastLock *lock);

パラメータ

FastLock*	lock	Control Block of FastLock	高速ロックの管理ブロック

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

高速ロックに対してロックの操作を行う。

既にロックされていれば、ロック解除されるまで自タスクは待ち状態となり、待ち行列につながれる。待ち行列はタスク優先度順である。

高速化のため、エラーの検出は行わない。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.10.2.4 Unlock - 高速ロックのロック解除操作

C言語インタフェース

```
#include <tk/tkernel.h>
```

```
void Unlock (FastLock *lock );
```

パラメータ

FastLock*	lock	Control Block of FastLock	高速ロックの管理ブロック

リターンパラメータ

なし

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

高速ロックに対してロック解除の操作を行う。
対象となる高速ロックを待っているタスクがあれば、待ち行列の先頭のタスクが新たにロックを獲得する。
高速化のため、エラーの検出は行わない。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.10.2.5 CreateMLock - 高速マルチロックの生成

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = CreateMLock (FastMLock *lock , CONST UB *name);

パラメータ

FastMLock*	lock	Control Block of FastMLock	高速マルチロックの管理ブロック
CONST UB*	name	Name of FastMLock	高速マルチロックの名前

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_NOMEM	メモリ不足(管理ブロック用の領域が確保できない)
E_LIMIT	高速マルチロックの数がシステムの上限を超えた

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

高速マルチロックを生成する。

lock は高速マルチロックの管理のための構造体である。name は高速マルチロックに付ける名前であるが、NULL でもよい。

高速マルチロックは、排他制御のための独立したバイナリセマフォを32個並べたものであり、なるべく高速に操作できるように実装されている。32個のバイナリセマフォは、0番～31番のロック番号で指定する。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.10.2.6 DeleteMLock - 高速マルチロックの削除

C言語インタフェース

```
#include <tk/tkernel.h>

ER ercd = DeleteMLock (FastMLock *lock );
```

パラメータ

FastMLock*	lock	Control Block of FastMLock	高速マルチロックの管理ブロック

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

高速マルチロックを削除する。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.10.2.7 MLock - 高速マルチロックのロック操作

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = MLock (FastMLock *lock , INT no);

パラメータ

FastMLock*	lock	Control Block of FastMLock	高速マルチロックの管理ブロック
INT	no	Lock Number	ロック番号

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー
E_DLT	待ちオブジェクトが削除された
E_RLWAI	待ち状態強制解除
E_CTX	コンテキストエラー

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

高速マルチロックに対してロックの操作を行う。

no はロック番号で、0～31を使用できる。

既に同じロック番号でロックされていれば、同じロック番号でロック解除されるまで自タスクは待ち状態となり、待ち行列につながれる。待ち行列はタスク優先度順である。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.10.2.8 MLockTmo - 高速マルチロックのロック操作(タイムアウト指定付き)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = MLockTmo (FastMLock *lock , INT no , TMO tmout);

パラメータ

FastMLock*	lock	Control Block of FastMLock	高速マルチロックの管理ブロック
INT	no	Lock Number	ロック番号
TMO	tmout	Timeout	タイムアウト指定(ミリ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー
E_DLT	待ちオブジェクトが削除された
E_RLWAI	待ち状態強制解除
E_TMOUT	タイムアウト
E_CTX	コンテキストエラー

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

高速マルチロックに対して、タイムアウト指定付きのロックの操作を行う。

tmout でタイムアウト時間の指定ができる点以外は、MLock() と同じである。tmout で指定した時間が経過してもロックの獲得ができない場合は、E_TMOUT を返す。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.10.2.9 MLockTmo_u - 高速マルチロックのロック操作(タイムアウト指定付き、マイクロ秒単位)

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = MLockTmo_u (FastMLock *lock , INT no , TMO_U tmout_u);

パラメータ

FastMLock*	lock	Control Block of FastMLock	高速マルチロックの管理ブロック
INT	no	Lock Number	ロック番号
TMO_U	tmout_u	Timeout	タイムアウト指定(マイクロ秒)

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	パラメータエラー
E_DLT	待ちオブジェクトが削除された
E_RLWAI	待ち状態強制解除
E_TMOUT	タイムアウト
E_CTX	コンテキストエラー

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

高速マルチロックに対して、マイクロ秒単位のタイムアウト指定付きのロックの操作を行う。

タイムアウト時間の指定が64ビットのマイクロ秒単位になる点以外は、MLockTmo() と同じである。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.10.2.10 MUnlock - 高速マルチロックのロック解除操作

C言語インタフェース

```
#include <tk/tkernel.h>
```

ER ercd = MUnlock (FastMLock *lock , INT no);

パラメータ

FastMLock*	lock	Control Block of FastMLock	高速マルチロックの管理ブロック
INT	no	Lock Number	ロック番号

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	×

解説

高速マルチロックに対してロック解除操作を行う。

no はロック番号で、0～31を使用できる。

同じロック番号に対して待ち状態のタスクがあれば、待ち行列の先頭のタスクが新たにロックを獲得する。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたAPIである。

5.11 サブシステムおよびデバイスドライバの起動

サブシステムおよびデバイスドライバは次のようなエントリーを持つ。

```
ER main( INT ac, UB *av[] )
{
        if ( ac >= 0 ) {
                /* サブシステム/デバイスドライバ起動処理 */
        } else {
                /* サブシステム/デバイスドライバ終了処理 */
        }

        return ercd;
}
```

このエントリールーチンは、サブシステムおよびデバイスドライバの起動処理または終了処理を行うだけで、実際のサービスの提供は行わない。起動処理または終了処理が済んだら直ちに呼出元に戻らなければならない。エントリールーチンは、できるだけ速やかに実行し、呼出元に戻らなければならない。

エントリールーチンは、通常はシステムの起動時および終了時にシステムリソースグループに属するタスクから呼び出され、システム全体の起動処理タスクまたは終了処理タスクのコンテキスト(保護レベル0)で実行される。なお、実装によっては、準タスク部として実行される場合がある。また、サブシステムやデバイスドライバの動的なロードをサポートしているシステムにおいては、システムの起動時および終了時以外に呼び出される場合もある。

複数のサブシステムおよびデバイスドライバがあるとき、システムの起動時および終了時のエントリールーチンの呼出は、一つずつ順番に行われる。複数のエントリールーチンが異なるタスクから同時に呼び出されることはない。したがって、サブシステムまたはデバイスドライバ間で初期化順序に依存関係があるような場合、エントリールーチンから戻る前に必要な処理をすべて終わらせておくことで、初期化の順序関係を維持することができる。

エントリールーチンの関数名は通常は main であるが、OSとリンクする場合など main が使用できない場合は任意の名称としてよい。

エントリールーチンをT-Kernelに指示する(登録する)方法、パラメータの指定方法、およびエントリールーチンの呼出順の指定方法は、T-Kernelの実装ごとに定義される。

5.11.1 起動処理

パラメータ

INT	ac	パラメータの数(≧0)
UB*	av	パラメータ(文字列)

リターンパラメータ

戻値	エラーコード

解説

ac ≧0の場合、起動処理となる。サブシステムおよびデバイスドライバの初期化処理を行った後、サブシステムおよびデバイスドライバを登録する。

戻値として負の値(エラー)を返した場合は起動処理の失敗とみなされる。T-Kernelの実装によっては、このサブシステムまたはデバイスドライバをメモリから削除してしまう場合があるので、サブシステムおよびデバイスドライバの登録を行った状態でエラーを戻してはいけない。エラーを戻すときは、必ず登録を抹消しておかなければならない。また、獲得したリソースも解放しておく必要がある。自動的には解放されない。

ac, av は C言語の標準的な main() の引数と同じである。ac にパラメータ数、av は ac+1個分のポインタの配列で、文字列のパラメータを指している。配列の終端(av[ac])は NULL である。

av[0] は、サブシステムまたはデバイスドライバの名称である。一般的には、サブシステムまたはデバイスドライバが格納されたファイル名などになるが、どのような名称が格納されるかは実装定義である。名称なし(空文字列 "")であってもよい。

av[1] 以降のパラメータは、各サブシステムおよびデバイスドライバごとに定義される。

av の指す文字列の領域は、エントリールーチンから抜けると消滅するため、必要に応じてパラメータは別の場所に保存しなければならない。

5.11.2 終了処理

パラメータ

INT	ac	-1
UB*	av	NULL

リターンパラメータ

戻値	エラーコード

解説

ac＜0の場合、終了処理となる。サブシステムおよびデバイスドライバの登録を抹消した後、獲得したリソースなどを解放する。終了処理中にエラーが発生した場合も、終了処理を中止してはいけない。可能な限り処理する。一部が正常に終了処理できなかった場合は、戻値にエラーを返す。

サブシステムおよびデバイスドライバに対する要求を処理中の状態で、終了処理が呼び出された場合の動作は、サブシステムおよびデバイスドライバの実装に依存する。一般的には、終了処理はシステム終了時に呼び出されることを前提とするため、処理中の要求がないのが普通である。そのため、処理中の要求があった場合の動作は保証されないのが普通である。

第 6 章

T-Kernel/DSの機能

この章では、T-Kernel/DS(Debugger Support)で提供している機能の詳細について説明を行う。

T-Kernel/DSは、デバッガがT-Kernelの内部状態の参照や実行のトレースを行うための機能を提供するものである。T-Kernel/DSの提供する機能は、デバッガ専用であり、一般のアプリケーション等からは使用しない。

全般的な注意・補足事項

- T-Kernel/DSのシステムコール(td_〜)は、特に明記されているものを除き、タスク独立部およびディスパッチ禁止中・割込み禁止中からも呼び出すことができる。

  ただし、実装によっては機能が制限される場合がある。

- T-Kernel/DSのシステムコール(td_〜)を割込み禁止状態で呼び出した場合、割込み許可されることなく処理される。同様に、カーネルのその他の状態も変化させない。割込み許可状態やディスパッチ許可状態で呼び出された場合は、カーネルの動作も継続されるため、カーネルの状態は変化する場合がある。
- T-Kernel/DSのシステムコール(td_〜)は、T-Kernel/OSのシステムコールの呼出可能な保護レベルより低い保護レベル(TSVCLimit より低い保護レベル)から呼び出すことはできない(E_OACV)。
- 常に発生する可能性のあるエラー E_PAR, E_MACV, E_CTX などは、特に説明を必要とする場合以外は省略している。

6.1 カーネル内部状態取得機能

カーネル内部状態取得機能は、デバッガがカーネルの内部状態を取得するための機能である。オブジェクトの一覧を取得する機能、タスクの優先順位を取得する機能、待ち行列に並んだタスクの並び順を取得する機能、オブジェクトやシステムやタスクレジスタの状態を取得する機能、および時刻を取得する機能が含まれる。

6.1.1 td_lst_tsk - タスクIDのリスト参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_tsk (ID list[], INT nent);

パラメータ

ID	list[]	List	タスクIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	使用されているタスクの個数
	または	Error Code	エラーコード

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

現在使用されているタスクIDのリストを取得し、list へ最大 nent 個分のタスクIDを格納する。戻値に、使用されているタスクの個数を返す。戻値 > nent であれば、すべてのタスクIDは取得できていないことを示す。

6.1.2 td_lst_sem - セマフォIDのリスト参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_sem (ID list[], INT nent);

パラメータ

ID	list[]	List	セマフォIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	使用されているセマフォの個数
	または	Error Code	エラーコード

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

現在使用されているセマフォIDのリストを取得し、list へ最大 nent 個分のセマフォIDを格納する。戻値に、使用されているセマフォの個数を返す。戻値 > nent であれば、すべてのセマフォIDは取得できていないことを示す。

6.1.3 td_lst_flg - イベントフラグIDのリスト参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_flg (ID list[], INT nent);

パラメータ

ID	list[]	List	イベントフラグIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	使用されているイベントフラグの個数
	または	Error Code	エラーコード

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

現在使用されているイベントフラグIDのリストを取得し、list へ最大 nent 個分のイベントフラグIDを格納する。戻値に、使用されているイベントフラグの個数を返す。戻値 > nent であれば、すべてのイベントフラグIDは取得できていないことを示す。

6.1.4 td_lst_mbx - メールボックスIDのリスト参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_mbx (ID list[], INT nent);

パラメータ

ID	list[]	List	メールボックスIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	使用されているメールボックスの個数
	または	Error Code	エラーコード

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

現在使用されているメールボックスIDのリストを取得し、list へ最大 nent 個分のメールボックスIDを格納する。戻値に、使用されているメールボックスの個数を返す。戻値 > nent であれば、すべてのメールボックスIDは取得できていないことを示す。

6.1.5 td_lst_mtx - ミューテックスIDのリスト参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_mtx (ID list[], INT nent);

パラメータ

ID	list[]	List	ミューテックスIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	使用されているミューテックスの個数
	または	Error Code	エラーコード

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

現在使用されているミューテックスIDのリストを取得し、list へ最大 nent 個分のミューテックスIDを格納する。戻値に、使用されているミューテックスの個数を返す。戻値 > nent であれば、すべてのミューテックスIDは取得できていないことを示す。

6.1.6 td_lst_mbf - メッセージバッファIDのリスト参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_mbf (ID list[], INT nent);

パラメータ

ID	list[]	List	メッセージバッファIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	使用されているメッセージバッファの個数
	または	Error Code	エラーコード

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

現在使用されているメッセージバッファIDのリストを取得し、list へ最大 nent 個分のメッセージバッファIDを格納する。戻値に、使用されているメッセージバッファの個数を返す。戻値 > nent であれば、すべてのメッセージバッファIDは取得できていないことを示す。

6.1.7 td_lst_por - ランデブポートIDのリスト参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_por (ID list[], INT nent);

パラメータ

ID	list[]	List	ランデブポートIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	使用されているランデブポートの個数
	または	Error Code	エラーコード

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

現在使用されているランデブポートIDのリストを取得し、list へ最大 nent 個分のランデブポートIDを格納する。戻値に、使用されているランデブポートの個数を返す。戻値 > nent であれば、すべてのランデブポートIDは取得できていないことを示す。

6.1.8 td_lst_mpf - 固定長メモリプールIDのリスト参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_mpf (ID list[], INT nent);

パラメータ

ID	list[]	List	固定長メモリプールIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	使用されている固定長メモリプールの個数
	または	Error Code	エラーコード

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

現在使用されている固定長メモリプールIDのリストを取得し、list へ最大 nent 個分の固定長メモリプールIDを格納する。戻値に、使用されている固定長メモリプールの個数を返す。戻値 > nent であれば、すべての固定長メモリプールIDは取得できていないことを示す。

6.1.9 td_lst_mpl - 可変長メモリプールIDのリスト参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_mpl (ID list[], INT nent);

パラメータ

ID	list[]	List	可変長メモリプールIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	使用されている可変長メモリプールの個数
	または	Error Code	エラーコード

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

現在使用されている可変長メモリプールIDのリストを取得し、list へ最大 nent 個分の可変長メモリプールIDを格納する。戻値に、使用されている可変長メモリプールの個数を返す。戻値 > nent であれば、すべての可変長メモリプールIDは取得できていないことを示す。

6.1.10 td_lst_cyc - 周期ハンドラIDのリスト参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_cyc (ID list[], INT nent);

パラメータ

ID	list[]	List	周期ハンドラIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	使用されている周期ハンドラの個数
	または	Error Code	エラーコード

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

現在使用されている周期ハンドラIDのリストを取得し、list へ最大 nent 個分の周期ハンドラIDを格納する。戻値に、使用されている周期ハンドラの個数を返す。戻値 > nent であれば、すべての周期ハンドラIDは取得できていないことを示す。

6.1.11 td_lst_alm - アラームハンドラIDのリスト参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_alm (ID list[], INT nent);

パラメータ

ID	list[]	List	アラームハンドラIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	使用されているアラームハンドラの個数
	または	Error Code	エラーコード

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

現在使用されているアラームハンドラIDのリストを取得し、list へ最大 nent 個分のアラームハンドラIDを格納する。戻値に、使用されているアラームハンドラの個数を返す。戻値 > nent であれば、すべてのアラームハンドラIDは取得できていないことを示す。

6.1.12 td_lst_ssy - サブシステムIDのリスト参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_lst_ssy (ID list[], INT nent);

パラメータ

ID	list[]	List	サブシステムIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	使用されているサブシステムの個数
	または	Error Code	エラーコード

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

現在使用されているサブシステムIDのリストを取得し、list へ最大 nent 個分のサブシステムIDを格納する。戻値に、使用されているサブシステムの個数を返す。戻値 > nent であれば、すべてのサブシステムIDは取得できていないことを示す。

6.1.13 td_rdy_que - タスクの優先順位の参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_rdy_que (PRI pri , ID list[], INT nent);

パラメータ

PRI	pri	Task Priority	対象タスク優先度
ID	list[]	Task ID List	タスクIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	優先度 pri の実行できる状態のタスクの個数
	または	Error Code	エラーコード

エラーコード

E_PAR	パラメータエラー(pri が不正あるいは利用できない値)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

実行できる状態(実行可能状態および実行状態)のタスクのうち、優先度が pri であるものを、優先順位の高い順に一列に並べ、その並び順にタスクIDのリストを取得する。

list には、優先順位が最高のものを先頭としてその並び順に、タスクIDが最大 nent 個まで格納される。

戻値には、pri の優先度を持つ実行できる状態のタスクの個数を返す。戻値 > nent であれば、すべてのタスクIDは取得できていないことを示す。

6.1.14 td_sem_que - セマフォの待ち行列の参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_sem_que (ID semid , ID list[], INT nent);

パラメータ

ID	semid	Semaphore ID	対象セマフォID
ID	list[]	Task ID List	待ちタスクのタスクIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	待ちタスクの個数
	または	Error Code	エラーコード

エラーコード

E_ID	ID番号が不正(semid が不正あるいは利用できない)
E_NOEXS	対象オブジェクトが存在しない(semid のセマフォが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

semid で指定したセマフォの待ち行列に並んでいるタスクのIDのリストを取得する。list には、セマフォの待ち行列の先頭からその並び順にタスクIDが最大 nent 個まで格納される。戻値には、セマフォの待ち行列に並んでいるタスクの個数を返す。戻値 > nent であれば、すべてのタスクIDは取得できないことを示す。

6.1.15 td_flg_que - イベントフラグの待ち行列の参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_flg_que (ID flgid , ID list[], INT nent);

パラメータ

ID	flgid	EventFlag ID	対象イベントフラグID
ID	list[]	Task ID List	待ちタスクのタスクIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	待ちタスクの個数
	または	Error Code	エラーコード

エラーコード

E_ID	ID番号が不正(flgid が不正あるいは利用できない)
E_NOEXS	対象オブジェクトが存在しない(flgid のイベントフラグが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

flgid で指定したイベントフラグの待ち行列に並んでいるタスクのIDのリストを取得する。list には、イベントフラグの待ち行列の先頭からその並び順にタスクIDが最大 nent 個まで格納される。戻値には、イベントフラグの待ち行列に並んでいるタスクの個数を返す。戻値 > nent であれば、すべてのタスクIDは取得できないことを示す。

6.1.16 td_mbx_que - メールボックスの待ち行列の参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_mbx_que (ID mbxid , ID list[], INT nent);

パラメータ

ID	mbxid	Mailbox ID	対象メールボックスID
ID	list[]	Task ID List	待ちタスクのタスクIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	待ちタスクの個数
	または	Error Code	エラーコード

エラーコード

E_ID	ID番号が不正(mbxid が不正あるいは利用できない)
E_NOEXS	対象オブジェクトが存在しない(mbxid のメールボックスが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

mbxid で指定したメールボックスの待ち行列に並んでいるタスクのIDのリストを取得する。list には、メールボックスの待ち行列の先頭からその並び順にタスクIDが最大 nent 個まで格納される。戻値には、メールボックスの待ち行列に並んでいるタスクの個数を返す。戻値 > nent であれば、すべてのタスクIDは取得できないことを示す。

6.1.17 td_mtx_que - ミューテックスの待ち行列の参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_mtx_que (ID mtxid , ID list[], INT nent);

パラメータ

ID	mtxid	Mutex ID	対象ミューテックスID
ID	list[]	Task ID List	待ちタスクのタスクIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	待ちタスクの個数
	または	Error Code	エラーコード

エラーコード

E_ID	ID番号が不正(mtxid が不正あるいは利用できない)
E_NOEXS	対象オブジェクトが存在しない(mtxid のミューテックスが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

mtxid で指定したミューテックスの待ち行列に並んでいるタスクのIDのリストを取得する。list には、ミューテックスの待ち行列の先頭からその並び順にタスクIDが最大 nent 個まで格納される。戻値には、ミューテックスの待ち行列に並んでいるタスクの個数を返す。戻値 > nent であれば、すべてのタスクIDは取得できないことを示す。

6.1.18 td_smbf_que - メッセージバッファの送信待ち行列の参照

C言語インタフェース

```
#include <tk/dbgspt.h>

INT ct = td_smbf_que (ID mbfid , ID list[], INT nent );
```

パラメータ

ID	mbfid	Message Buffer ID	対象メッセージバッファID
ID	list[]	Task ID List	待ちタスクのタスクIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	待ちタスクの個数
	または	Error Code	エラーコード

エラーコード

E_ID	ID番号が不正(mbfid が不正あるいは利用できない)
E_NOEXS	対象オブジェクトが存在しない(mbfid のメッセージバッファが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

mbfid で指定したメッセージバッファの送信待ち行列に並んでいるタスクのIDのリストを取得する。list には、メッセージバッファの送信待ち行列の先頭からその並び順にタスクIDが最大 nent 個まで格納される。戻値には、メッセージバッファの送信待ち行列に並んでいるタスクの個数を返す。戻値 > nent であれば、すべてのタスクIDは取得できないことを示す。

6.1.19 td_rmbf_que - メッセージバッファの受信待ち行列の参照

C言語インタフェース

```
#include <tk/dbgspt.h>

INT ct = td_rmbf_que (ID mbfid , ID list[], INT nent );
```

パラメータ

ID	mbfid	Message Buffer ID	対象メッセージバッファID
ID	list[]	Task ID List	待ちタスクのタスクIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	待ちタスクの個数
	または	Error Code	エラーコード

エラーコード

E_ID	ID番号が不正(mbfid が不正あるいは利用できない)
E_NOEXS	対象オブジェクトが存在しない(mbfid のメッセージバッファが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

mbfid で指定したメッセージバッファの受信待ち行列に並んでいるタスクのIDのリストを取得する。list には、メッセージバッファの受信待ち行列の先頭からその並び順にタスクIDが最大 nent 個まで格納される。戻値には、メッセージバッファの受信待ち行列に並んでいるタスクの個数を返す。戻値 > nent であれば、すべてのタスクIDは取得できないことを示す。

6.1.20 td_cal_que - ランデブ呼出待ち行列の参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_cal_que (ID porid , ID list[], INT nent);

パラメータ

ID	porid	Port ID	対象ランデブポートID
ID	list[]	Task ID List	待ちタスクのタスクIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	待ちタスクの個数
	または	Error Code	エラーコード

エラーコード

E_ID	ID番号が不正(porid が不正あるいは利用できない)
E_NOEXS	対象オブジェクトが存在しない(porid のランデブポートが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

porid で指定したランデブ呼出待ち行列に並んでいるタスクのIDのリストを取得する。list には、ランデブ呼出待ち行列の先頭からその並び順にタスクIDが最大 nent 個まで格納される。戻値には、ランデブ呼出待ち行列に並んでいるタスクの個数を返す。戻値 > nent であれば、すべてのタスクIDは取得できないことを示す。

6.1.21 td_acp_que - ランデブ受付待ち行列の参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_acp_que (ID porid , ID list[], INT nent);

パラメータ

ID	porid	Port ID	対象ランデブポートID
ID	list[]	Task ID List	待ちタスクのタスクIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	待ちタスクの個数
	または	Error Code	エラーコード

エラーコード

E_ID	ID番号が不正(porid が不正あるいは利用できない)
E_NOEXS	対象オブジェクトが存在しない(porid のランデブポートが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

porid で指定したランデブ受付待ち行列に並んでいるタスクのIDのリストを取得する。list には、ランデブ受付待ち行列の先頭からその並び順にタスクIDが最大 nent 個まで格納される。戻値には、ランデブ受付待ち行列に並んでいるタスクの個数を返す。戻値 > nent であれば、すべてのタスクIDは取得できないことを示す。

6.1.22 td_mpf_que - 固定長メモリプールの待ち行列の参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_mpf_que (ID mpfid , ID list[], INT nent);

パラメータ

ID	mpfid	Memory Pool ID	対象固定長メモリプールID
ID	list[]	Task ID List	待ちタスクのタスクIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	待ちタスクの個数
	または	Error Code	エラーコード

エラーコード

E_ID	ID番号が不正(mpfid が不正あるいは利用できない)
E_NOEXS	対象オブジェクトが存在しない(mpfid の固定長メモリプールが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

mpfid で指定した固定長メモリプールの待ち行列に並んでいるタスクのIDのリストを取得する。list には、固定長メモリプールの待ち行列の先頭からその並び順にタスクIDが最大 nent 個まで格納される。戻値には、固定長メモリプールの待ち行列に並んでいるタスクの個数を返す。戻値 > nent であれば、すべてのタスクIDは取得できないことを示す。

6.1.23 td_mpl_que - 可変長メモリプールの待ち行列の参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

INT ct = td_mpl_que (ID mplid , ID list[], INT nent);

パラメータ

ID	mplid	Memory Pool ID	対象可変長メモリプールID
ID	list[]	Task ID List	待ちタスクのタスクIDを格納する領域
INT	nent	Number of List Entries	list に格納可能な最大数

リターンパラメータ

INT	ct	Count	待ちタスクの個数
	または	Error Code	エラーコード

エラーコード

E_ID	ID番号が不正(mplid が不正あるいは利用できない)
E_NOEXS	対象オブジェクトが存在しない(mplid の可変長メモリプールが存在しない)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

mplid で指定した可変長メモリプールの待ち行列に並んでいるタスクのIDのリストを取得する。list には、可変長メモリプールの待ち行列の先頭からその並び順にタスクIDが最大 nent 個まで格納される。戻値には、可変長メモリプールの待ち行列に並んでいるタスクの個数を返す。戻値 > nent であれば、すべてのタスクIDは取得できないことを示す。

6.1.24 td_ref_tsk - タスク状態参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_tsk (ID tskid , TD_RTSK *rtsk);

パラメータ

ID	tskid	Task ID	対象タスクID(TSK_SELF 可)
TD_RTSK*	rtsk	Packet to Refer Task Status	タスク状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

rtsk の内容

void*	exinf	Extended Information	拡張情報
PRI	tskpri	Task Priority	現在の優先度
PRI	tskbpri	Task Base Priority	ベース優先度
UINT	tskstat	Task State	タスク状態
UINT	tskwait	Task Wait Factor	待ち要因
ID	wid	Waiting Object ID	待ちオブジェクトID
INT	wupcnt	Wakeup Count	起床要求キューイング数
INT	suscnt	Suspend Count	強制待ち要求ネスト数
RELTIM	slicetime	Slice Time	最大連続実行時間(ミリ秒)
UINT	waitmask	Wait Mask	待ちを禁止されている待ち要因
UINT	texmask	Task Exception Mask	許可されているタスク例外
UINT	tskevent	Task Event	発生しているタスクイベント
FP	task	Task Start Address	タスク起動アドレス
INT	stksz	User Stack Size	ユーザスタックサイズ(バイト数)
INT	sstksz	System Stack Size	システムスタックサイズ(バイト数)
void*	istack	Initial User Stack Pointer	ユーザスタックポインタ初期値
void*	isstack	Initial System Stack Pointer	システムスタックポインタ初期値

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

タスクの状態を参照する。tk_ref_tsk() と同等だが、タスク起動アドレスおよびスタックに関する情報が追加されている。

スタック領域は、スタックポインタ初期値の位置から低位アドレス(値の小さい方)へ向かって、スタックサイズ分となる。

- istack−stksz ≦ ユーザスタック領域 ＜ istack
- isstack−sstksz ≦ システムスタック領域 ＜ isstack

なお、スタックポインタ初期値(istack, isstack)は、スタックポインタの現在位置ではない。タスク起動前の状態であっても、スタック領域は使用されている場合がある。スタックポインタの現在位置を得るには、td_get_reg() を用いる。

タスク状態情報(TD_RTSK)の slicetime は、ミリ秒単位に切り上げた値(単位ミリ秒)を返す。マイクロ秒単位の情報を知りたい場合には、td_ref_tsk_u を使う。

6.1.25 td_ref_tsk_u - タスク状態参照(マイクロ秒単位)

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_tsk_u (ID tskid , TD_RTSK_U *rtsk_u);

パラメータ

ID	tskid	Task ID	対象タスクID(TSK_SELF 可)
TD_RTSK_U*	rtsk_u	Packet to Refer Task Status	タスク状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

rtsk_u の内容

void*	exinf	Extended Information	拡張情報
PRI	tskpri	Task Priority	現在の優先度
PRI	tskbpri	Task Base Priority	ベース優先度
UINT	tskstat	Task State	タスク状態
UINT	tskwait	Task Wait Factor	待ち要因
ID	wid	Waiting Object ID	待ちオブジェクトID
INT	wupcnt	Wakeup Count	起床要求キューイング数
INT	suscnt	Suspend Count	強制待ち要求ネスト数
RELTIM_U	slicetime_u	Slice Time	最大連続実行時間(マイクロ秒)
UINT	waitmask	Wait Mask	待ちを禁止されている待ち要因
UINT	texmask	Task Exception Mask	許可されているタスク例外
UINT	tskevent	Task Event	発生しているタスクイベント
FP	task	Task Start Address	タスク起動アドレス
INT	stksz	User Stack Size	ユーザスタックサイズ(バイト数)
INT	sstksz	System Stack Size	システムスタックサイズ(バイト数)
void*	istack	Initial User Stack Pointer	ユーザスタックポインタ初期値
void*	isstack	Initial System Stack Pointer	システムスタックポインタ初期値

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

td_ref_tsk のリターンパラメータである slicetime を64ビットマイクロ秒単位の slicetime_u としたシステムコールである。

リターンパラメータが slicetime_u となった点を除き、本システムコールの仕様は td_ref_tsk と同じである。詳細は td_ref_tsk の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

6.1.26 td_ref_tex - タスク例外の状態参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_tex (ID tskid , TD_RTEX *pk_rtex);

パラメータ

ID	tskid	Task ID	対象タスクID(TSK_SELF 可)
TD_RTEX*	pk_rtex	Packet to Refer Task Exception Status	タスク例外状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rtex の内容

UINT	pendtex	Pending Task Exception	発生しているタスク例外
UINT	texmask	Task Exception Mask	許可されているタスク例外

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

タスク例外の状態を参照する。tk_ref_tex() と同等。

6.1.27 td_ref_sem - セマフォ状態参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_sem (ID semid , TD_RSEM *rsem);

パラメータ

ID	semid	Semaphore ID	対象セマフォID
TD_RSEM*	rsem	Packet to Refer Semaphore Status	セマフォ状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

rsem の内容

void*	exinf	Extended Information	拡張情報
ID	wtsk	Waiting Task ID	待ちタスクのID
INT	semcnt	Semaphore Count	現在のセマフォカウント値

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

セマフォの状態を参照する。tk_ref_sem() と同等。

6.1.28 td_ref_flg - イベントフラグ状態参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_flg (ID flgid , TD_RFLG *rflg);

パラメータ

ID	flgid	EventFlag ID	対象イベントフラグID
TD_RFLG*	rflg	Packet to Refer EventFlag Status	イベントフラグ状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

rflg の内容

void*	exinf	Extended Information	拡張情報
ID	wtsk	Waiting Task ID	待ちタスクのID
UINT	flgptn	EventFlag Bit Pattern	現在のイベントフラグのビットパターン

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

イベントフラグの状態を参照する。tk_ref_flg() と同等。

6.1.29 td_ref_mbx - メールボックス状態参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_mbx (ID mbxid , TD_RMBX *rmbx);

パラメータ

ID	mbxid	Mailbox ID	対象メールボックスID
TD_RMBX*	rmbx	Packet to Refer Mailbox Status	メールボックス状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

rmbx の内容

void*	exinf	Extended Information	拡張情報
ID	wtsk	Waiting Task ID	待ちタスクのID
T_MSG*	pk_msg	Packet of Message	次に受信されるメッセージ

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

メールボックスの状態を参照する。tk_ref_mbx() と同等。

6.1.30 td_ref_mtx - ミューテックス状態参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_mtx (ID mtxid , TD_RMTX *rmtx);

パラメータ

ID	mtxid	Mutex ID	対象ミューテックスID
TD_RMTX*	rmtx	Packet to Refer Mutex Status	ミューテックス状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

rmtx の内容

void*	exinf	Extended Information	拡張情報
ID	htsk	Locking Task ID	ロックしているタスクのID
ID	wtsk	Lock Waiting Task ID	ロック待ちタスクのID

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

ミューテックスの状態を参照する。tk_ref_mtx() と同等。

6.1.31 td_ref_mbf - メッセージバッファ状態参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_mbf (ID mbfid , TD_RMBF *rmbf);

パラメータ

ID	mbfid	Message Buffer ID	対象メッセージバッファID
TD_RMBF*	rmbf	Packet to Refer Message Buffer Status	メッセージバッファ状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

rmbf の内容

void*	exinf	Extended Information	拡張情報
ID	wtsk	Waiting Task ID	受信待ちタスクのID
ID	stsk	Send Waiting Task ID	送信待ちタスクのID
INT	msgsz	Message Size	次に受信されるメッセージのサイズ(バイト数)
INT	frbufsz	Free Buffer Size	空きバッファのサイズ(バイト数)
INT	maxmsz	Maximum Message Size	メッセージの最大長(バイト数)

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

メッセージバッファの状態を参照する。tk_ref_mbf() と同等。

6.1.32 td_ref_por - ランデブポート状態参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_por (ID porid , TD_RPOR *rpor);

パラメータ

ID	porid	Port ID	対象ランデブポートID
TD_RPOR*	rpor	Packet to Refer Port Status	ランデブポート状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

rpor の内容

void*	exinf	Extended Information	拡張情報
ID	wtsk	Waiting Task ID	呼出待ちタスクのID
ID	atsk	Accept Waiting Task ID	受付待ちタスクのID
INT	maxcmsz	Maximum Call Message Size	呼出時のメッセージの最大長(バイト数)
INT	maxrmsz	Maximum Reply Message Size	返答時のメッセージの最大長(バイト数)

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

ランデブポートの状態を参照する。tk_ref_por() と同等。

6.1.33 td_ref_mpf - 固定長メモリプール状態参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_mpf (ID mpfid , TD_RMPF *rmpf);

パラメータ

ID	mpfid	Memory Pool ID	対象固定長メモリプールID
TD_RMPF*	rmpf	Packet to Refer Memory Pool Status	メモリプール状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

rmpf の内容

void*	exinf	Extended Information	拡張情報
ID	wtsk	Waiting Task ID	待ちタスクのID
INT	frbcnt	Free Block Count	空き領域のブロック数

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

固定長メモリプールの状態を参照する。tk_ref_mpf() と同等。

6.1.34 td_ref_mpl - 可変長メモリプール状態参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_mpl (ID mplid , TD_RMPL *rmpl);

パラメータ

ID	mplid	Memory Pool ID	対象可変長メモリプールID
TD_RMPL*	rmpl	Packet to Refer Memory Pool Status	メモリプール状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

rmpl の内容

void*	exinf	Extended Information	拡張情報
ID	wtsk	Waiting Task ID	待ちタスクのID
INT	frsz	Free Memory Size	空き領域の合計サイズ(バイト数)
INT	maxsz	Max Memory Size	最大の空き領域のサイズ(バイト数)

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

可変長メモリプールの状態を参照する。tk_ref_mpl() と同等。

6.1.35 td_ref_cyc - 周期ハンドラ状態参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_cyc (ID cycid , TD_RCYC *rcyc);

パラメータ

ID	cycid	Cyclic Handler ID	対象周期ハンドラID
TD_RCYC*	rcyc	Packet to Refer Cyclic Handler Status	周期ハンドラの状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

rcyc の内容

void*	exinf	Extended Information	拡張情報
RELTIM	lfttim	Left Time	次のハンドラ起動までの残り時間(ミリ秒)
UINT	cycstat	Cyclic Handler Status	周期ハンドラの状態

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

周期ハンドラの状態を参照する。tk_ref_cyc() と同等。

td_ref_cyc で取得する周期ハンドラ状態情報(TD_RCYC)における残り時間 lfttim は、ミリ秒単位に切り上げた値(単位ミリ秒)を返す。マイクロ秒単位の情報を知りたい場合には、td_ref_cyc_u を使う。

6.1.36 td_ref_cyc_u - 周期ハンドラ状態参照(マイクロ秒単位)

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_cyc_u (ID cycid , TD_RCYC_U *rcyc_u);

パラメータ

ID	cycid	Cyclic Handler ID	対象周期ハンドラID
TD_RCYC_U*	rcyc_u	Packet to Refer Cyclic Handler Status	周期ハンドラの状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

rcyc_u の内容

void*	exinf	Extended Information	拡張情報
RELTIM_U	lfttim_u	Left Time	次のハンドラ起動までの残り時間(マイクロ秒)
UINT	cycstat	Cyclic Handler Status	周期ハンドラの状態

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

td_ref_cyc のリターンパラメータである lfttim を64ビットマイクロ秒単位の lfttim_u としたシステムコールである。

リターンパラメータが lfttim_u となった点を除き、本システムコールの仕様は td_ref_cyc と同じである。詳細は td_ref_cyc の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

6.1.37 td_ref_alm - アラームハンドラ状態参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_alm (ID almid , TD_RALM *ralm);

パラメータ

ID	almid	Alarm Handler ID	対象アラームハンドラID
TD_RALM*	ralm	Packet to Refer Alarm Handler Status	アラームハンドラの状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

ralm の内容

void*	exinf	Extended Information	拡張情報
RELTIM	lfttim	Left Time	ハンドラ起動までの残り時間(ミリ秒)
UINT	almstat	Alarm Handler Status	アラームハンドラの状態

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

アラームハンドラの状態を参照する。tk_ref_alm() と同等。

td_ref_alm で取得するアラームハンドラ状態情報(TD_RALM)における残り時間 lfttim は、ミリ秒単位に切り上げた値(単位ミリ秒)を返す。マイクロ秒単位の情報を知りたい場合には、td_ref_alm_u を使う。

6.1.38 td_ref_alm_u - アラームハンドラ状態参照(マイクロ秒単位)

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_alm_u (ID almid , TD_RALM_U *ralm_u);

パラメータ

ID	almid	Alarm Handler ID	対象アラームハンドラID
TD_RALM_U*	ralm_u	Packet to Refer Alarm Handler Status	アラームハンドラの状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

ralm_u の内容

void*	exinf	Extended Information	拡張情報
RELTIM_U	lfttim_u	Left Time	ハンドラ起動までの残り時間(マイクロ秒)
UINT	almstat	Alarm Handler Status	アラームハンドラの状態

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

td_ref_alm のリターンパラメータである lfttim を64ビットマイクロ秒単位の lfttim_u としたシステムコールである。

リターンパラメータが lfttim_u となった点を除き、本システムコールの仕様は td_ref_alm と同じである。詳細は td_ref_alm の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

6.1.39 td_ref_sys - システム状態参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_sys (TD_RSYS *pk_rsys);

パラメータ

TD_RSYS*	pk_rsys	Packet to Refer System Status	システム状態を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_rsys の内容

INT	sysstat	System State	システム状態
ID	runtskid	Running Task ID	現在実行状態にあるタスクのID
ID	schedtskid	Scheduled Task ID	実行状態にすべきタスクのID

エラーコード

E_OK	正常終了

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

システムの状態を参照する。tk_ref_sys() と同等。

6.1.40 td_ref_ssy - サブシステム定義情報の参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_ssy (ID ssid , TD_RSSY *rssy);

パラメータ

ID	ssid	Subsystem ID	対象サブシステムID
TD_RSSY*	rssy	Packet to Refer Subsystem	サブシステム定義情報を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

rssy の内容

PRI	ssypri	Subsystem Priority	サブシステム優先度
INT	resblksz	Resource Control Block Size	リソース管理ブロックサイズ(バイト数)

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

サブシステムの状態を参照する。tk_ref_ssy() と同等。

6.1.41 td_inf_tsk - タスク統計情報参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_inf_tsk (ID tskid , TD_ITSK *pk_itsk , BOOL clr);

パラメータ

ID	tskid	Task ID	対象タスクID(TSK_SELF 可)
TD_ITSK*	pk_itsk	Packet to Refer Task Statistics	タスク統計情報を返す領域へのポインタ
BOOL	clr	Clear	タスク統計情報のクリアの有無

リターンパラメータ

ER	ercd	Error Code	エラーコード

pk_itsk の内容

RELTIM	stime	System Time	累積システムレベル実行時間(ミリ秒)
RELTIM	utime	User Time	累積ユーザレベル実行時間(ミリ秒)

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

タスク統計情報を参照する。tk_inf_tsk() と同等。clr＝TRUE(≠0)の場合は、統計情報を取り出した後、累積時間をリセット(0クリア)する。

タスク統計情報(TD_ITSK)の stime, utime は、ミリ秒単位に切り上げた値(単位ミリ秒)を返す。マイクロ秒単位の情報を知りたい場合には、td_inf_tsk_u を使う。

6.1.42 td_inf_tsk_u - タスク統計情報参照(マイクロ秒単位)

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_inf_tsk_u (ID tskid , TD_ITSK_U *itsk_u , BOOL clr);

パラメータ

ID	tskid	Task ID	対象タスクID(TSK_SELF 可)
TD_ITSK_U*	itsk_u	Packet to Refer Task Statistics	タスク統計情報を返す領域へのポインタ
BOOL	clr	Clear	タスク統計情報のクリアの有無

リターンパラメータ

ER	ercd	Error Code	エラーコード

itsk_u の内容

RELTIM_U	stime_u	System Time	累積システムレベル実行時間(マイクロ秒)
RELTIM_U	utime_u	User Time	累積ユーザレベル実行時間(マイクロ秒)

エラーコード

E_OK	正常終了
E_ID	ID番号が不正
E_NOEXS	対象オブジェクトが存在しない

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

td_inf_tsk のリターンパラメータである stime と utime を64ビットマイクロ秒単位の stime_u, utime_u としたシステムコールである。

リターンパラメータが stime_u, utime_u となった点を除き、本システムコールの仕様は td_inf_tsk と同じである。詳細は td_inf_tsk の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

6.1.43 td_get_reg - タスクレジスタの参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_get_reg (ID tskid , T_REGS *pk_regs , T_EIT *pk_eit , T_CREGS *pk_cregs);

パラメータ

ID	tskid	Task ID	対象タスクのID(TSK_SELF 不可)
T_REGS*	pk_regs	Packet of Registers	汎用レジスタの値を返す領域へのポインタ
T_EIT*	pk_eit	Packet of EIT Registers	例外時に保存されるレジスタの値を返す領域へのポインタ
T_CREGS*	pk_cregs	Packet of Control Registers	制御レジスタの値を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

T_REGS, T_EIT, T_CREGSの内容は、CPUおよび実装ごとに定義する。

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが現在実行状態のタスク)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

タスクのレジスタを参照する。tk_get_reg と同等。

現在実行状態にあるタスクは参照することはできない。タスク独立部の実行中を除けば、現在実行状態のタスクは自タスクである。

pk_regs, pk_eit, pk_cregs は、それぞれ NULL を指定すると、対応するレジスタは参照されない。

T_REGS, T_EIT, T_CREGSの内容は実装定義である。

6.1.44 td_set_reg - タスクレジスタの設定

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_set_reg (ID tskid , CONST T_REGS *pk_regs , CONST T_EIT *pk_eit , CONST T_CREGS *pk_cregs);

パラメータ

ID	tskid	Task ID	対象タスクのID(TSK_SELF 不可)
CONST T_REGS*	pk_regs	Packet of Registers	汎用レジスタ
CONST T_EIT*	pk_eit	Packet of EIT Registers	例外時に保存されるレジスタ
CONST T_CREGS*	pk_cregs	Packet of Control Registers	制御レジスタ

T_REGS, T_EIT, T_CREGSの内容は、CPUおよび実装ごとに定義する。

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_ID	不正ID番号(tskid が不正あるいは利用できない)
E_NOEXS	オブジェクトが存在していない(tskid のタスクが存在しない)
E_OBJ	オブジェクトの状態が不正(対象タスクが現在実行状態のタスク)

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

タスクのレジスタを設定する。tk_set_reg と同等。

現在実行状態にあるタスクに設定することはできない。タスク独立部の実行中を除けば、現在実行状態のタスクは自タスクである。

pk_regs, pk_eit, pk_cregs は、それぞれ NULL を指定すると、対応するレジスタは設定されない。

T_REGS, T_EIT, T_CREGS の内容は実装定義である。

6.1.45 td_get_tim - システム時刻参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_get_tim (SYSTIM *tim , UINT *ofs);

パラメータ

SYSTIM*	tim	Time	現在時刻(ミリ秒)を返す領域へのポインタ
UINT*	ofs	Offset	リターンパラメータ ofs を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード
SYSTIM	tim	Time	現在時刻(ミリ秒)
UINT	ofs	Offset	tim からの相対的な経過時間(ナノ秒)

tim の内容

W	hi	High 32bits	システムの現在時刻の上位32ビット
UW	lo	Low 32bits	システムの現在時刻の下位32ビット

エラーコード

E_OK	正常終了

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

現在時刻(1985年1月1日0時(GMT)からの通算のミリ秒数)を取得する。tim に返される値は tk_get_tim() と同じである。tim は、タイマ割込み間隔(周期)の分解能となるが、さらに細かい精度の時刻情報として tim からの経過時間を ofs にナノ秒単位で取得する。ofs の分解能は実装依存であるが、一般にはハードウェアタイマの分解能となる。

tim は、タイマ割込みによってカウントされる時刻であるため、割込み禁止期間中にタイマ割込み周期が来た場合、タイマ割込みハンドラが起動されず(起動が遅らされ)時刻が更新されないことがある。このような場合、tim には前回のタイマ割込みによって更新された時刻が返され、ofs には前回のタイマ割込みからの経過時間を返す。したがって、ofs はタイマ割込み間隔より長い時間となる場合がある。ofs がどの程度まで長い経過時間を計測できるかはハードウェアなどに依存するが、少なくともタイマ割込み間隔の2倍未満(0≦ofs<タイマ割込み間隔の2倍)の範囲まで計測できることが望ましい。

なお、tim および ofs に返される時刻は、td_get_tim() を呼び出してから戻るまでにかかった時間範囲の中のどこかの時点の時刻となる。td_get_tim() を呼び出した時点の時刻でも、td_get_tim() から戻った時点の時刻でもない。したがって、より正確な情報を得たい場合は、割込み禁止状態で呼び出すべきである。

6.1.46 td_get_tim_u - システム時刻参照(マイクロ秒単位)

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_get_tim_u (SYSTIM_U *tim_u , UINT *ofs);

パラメータ

SYSTIM_U*	tim_u	Time	現在時刻(マイクロ秒)を返す領域へのポインタ
UINT*	ofs	Offset	リターンパラメータ ofs を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード
SYSTIM_U	tim_u	Time	現在時刻(マイクロ秒)
UINT	ofs	Offset	tim_u からの相対的な経過時間(ナノ秒)

エラーコード

E_OK	正常終了

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

td_get_tim のリターンパラメータである tim を64ビットマイクロ秒単位の tim_u としたシステムコールである。

リターンパラメータが tim_u となった点を除き、本システムコールの仕様は td_get_tim と同じである。詳細は td_get_tim の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

6.1.47 td_get_otm - システム稼働時間参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_get_otm (SYSTIM *tim , UINT *ofs);

パラメータ

SYSTIM*	tim	Time	稼働時間(ミリ秒)を返す領域へのポインタ
UINT*	ofs	Offset	リターンパラメータ ofs を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード
SYSTIM	tim	Time	稼働時間(ミリ秒)
UINT	ofs	Offset	tim からの相対的な経過時間(ナノ秒)

tim の内容

W	hi	High 32bits	システム稼働時間の上位32ビット
UW	lo	Low 32bits	システム稼働時間の下位32ビット

エラーコード

E_OK	正常終了

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

システム稼働時間(システム起動時からの積算ミリ秒数)を取得する。tim に返される値は tk_get_otm() と同じである。tim は、タイマ割込み間隔(周期)の分解能となるが、さらに細かい精度の時刻情報として tim からの経過時間を ofs にナノ秒単位で取得する。ofs の分解能は実装依存であるが、一般にはハードウェアタイマの分解能となる。

tim は、タイマ割込みによってカウントされる時刻であるため、割込み禁止期間中にタイマ割込み周期が来た場合、タイマ割込みハンドラが起動されず(起動が遅らされ)時刻が更新されないことがある。このような場合、tim には前回のタイマ割込みによって更新された時刻が返され、ofs には前回のタイマ割込みからの経過時間を返す。したがって、ofs はタイマ割込み間隔より長い時間となる場合がある。ofs がどの程度まで長い経過時間を計測できるかはハードウェアなどに依存するが、少なくともタイマ割込み間隔の2倍未満(0≦ofs<タイマ割込み間隔の2倍)の範囲まで計測できることが望ましい。

なお、tim および ofs に返される時刻は、td_get_otm() を呼び出してから戻るまでにかかった時間範囲の中のどこかの時点の時刻となる。td_get_otm() を呼び出した時点の時刻でも、td_get_otm() から戻った時点の時刻でもない。したがって、より正確な情報を得たい場合は、割込み禁止状態で呼び出すべきである。

6.1.48 td_get_otm_u - システム稼働時間参照(マイクロ秒単位)

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_get_otm_u (SYSTIM_U *tim_u , UINT *ofs);

パラメータ

SYSTIM_U*	tim_u	Time	稼働時間(マイクロ秒)を返す領域へのポインタ
UINT*	ofs	Offset	リターンパラメータ ofs を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード
SYSTIM_U	tim_u	Time	稼働時間(マイクロ秒)
UINT	ofs	Offset	tim_u からの相対的な経過時間(ナノ秒)

エラーコード

E_OK	正常終了

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

td_get_otm のリターンパラメータである tim を64ビットマイクロ秒単位の tim_u としたシステムコールである。

リターンパラメータが tim_u となった点を除き、本システムコールの仕様は td_get_otm と同じである。詳細は td_get_otm の説明を参照のこと。

T-Kernel 1.0との差異

T-Kernel 2.0で追加されたシステムコールである。

6.1.49 td_ref_dsname - DSオブジェクト名称の参照

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_ref_dsname (UINT type , ID id , UB *dsname);

パラメータ

UINT	type	Object Type	対象オブジェクトタイプ
ID	id	Object ID	対象オブジェクトID
UB*	dsname	DS Object Name	DSオブジェクト名称を返す領域へのポインタ

リターンパラメータ

ER	ercd	Error Code	エラーコード

dsname の内容
オブジェクト生成時、または td_set_dsname() で設定されたDSオブジェクト名称

エラーコード

E_OK	正常終了
E_PAR	オブジェクトタイプ不正
E_NOEXS	対象オブジェクトが存在しない
E_OBJ	DSオブジェクト名称未使用

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

オブジェクト生成時に設定したDSオブジェクト名称(dsname)を参照する。対象となるオブジェクトは、オブジェクトタイプ(type)とオブジェクトID(id)で指定する。

指定可能なオブジェクトタイプ(type)は、以下の通りである。

TN_TSK	0x01	タスク
TN_SEM	0x02	セマフォ
TN_FLG	0x03	イベントフラグ
TN_MBX	0x04	メールボックス
TN_MBF	0x05	メッセージバッファ
TN_POR	0x06	ランデブポート
TN_MTX	0x07	ミューテックス
TN_MPL	0x08	可変長メモリプール
TN_MPF	0x09	固定長メモリプール

TN_CYC	0x0a	周期ハンドラ
TN_ALM	0x0b	アラームハンドラ

DSオブジェクト名称は、オブジェクトの属性に、TA_DSNAME が指定された場合に有効となる。オブジェクト生成後に、td_set_dsname() でDSオブジェクト名称を再設定した場合は、この名称が参照される。

DSオブジェクト名称は、

使用可能文字(UB)
　　a～z, A～Z, 0～9

名称長
　　8バイト(満たない場合は NULL で埋める)

とするが、文字コードのチェックは、T-Kernelでは行わない。

6.1.50 td_set_dsname - DSオブジェクト名称の設定

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_set_dsname (UINT type , ID id , CONST UB *dsname);

パラメータ

UINT	type	Object Type	対象オブジェクトタイプ
ID	id	Object ID	対象オブジェクトID
CONST UB*	dsname	DS Object Name	設定するDSオブジェクト名称

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

E_OK	正常終了
E_PAR	オブジェクトタイプ不正
E_NOEXS	対象オブジェクトが存在しない
E_OBJ	DSオブジェクト名称未使用

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

オブジェクト生成時に設定したDSオブジェクト名称(dsname)を再設定する。対象となるオブジェクトは、オブジェクトタイプ(type)とオブジェクトID(id)で指定する。

指定可能なオブジェクトタイプ(type)は、td_ref_dsname() と同様である。

なお、設定可能なDSオブジェクト名称は、

使用可能文字(UB)
　　a～z, A～Z, 0～9

名称長
　　8バイト(満たない場合は NULL で埋める)

とするが、文字コードのチェックは、T-Kernelでは行わない。

DSオブジェクト名称は、オブジェクトの属性に、TA_DSNAME が指定されている場合に有効である。TA_DSNAME 属性が指定されていないオブジェクトを対象とした場合は、エラー E_OBJ となる。

6.2 実行トレース機能

実行トレース機能は、デバッガがプログラムの実行をトレースするための機能である。実行トレースはフックルーチンを設定することによって行う。

- フックルーチンでは、各種の状態をフックルーチンが呼び出された時点の状態に戻してから、フックルーチンから戻らなければならない。ただし、レジスタに関してはC言語の関数の保存規則にしたがって復帰すればよい。
- フックルーチン内では、各種の状態を制限の緩い方へ変更してはいけない。例えば、割込み禁止状態で呼び出された場合は、割込みを許可してはいけない。
- フックルーチンは、保護レベル0で呼び出される。
- フックルーチンは、フックした時点のスタックをそのまま継承している。したがって、あまり多くスタックを消費するとスタックオーバーフローを引き起こす可能性がある。どの程度のスタックが使用可能であるかは、フックされた時点の状況により異なるため不確定である。フックルーチン内で独自スタックに切り替えれば、より安全である。

6.2.1 td_hok_svc - システムコール・拡張SVCのフックルーチン定義

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_hok_svc (CONST TD_HSVC *hsvc);

パラメータ

CONST TD_HSVC*	hsvc	SVC Hook Routine	フックルーチン定義情報

hsvc の内容

FP	enter	Hook Routine before Calling	呼出前のフックルーチン
FP	leave	Hook Routine after Calling	呼出後のフックルーチン

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

システムコールおよび拡張SVCの呼出前後に、フックルーチンを設定する。hsvc に NULL を指定することによりフックルーチンを解除する。

トレースの対象となるのは、T-Kernel/OSのシステムコール(tk_～)、および拡張SVCである。ただし、実装によるが、一般に tk_ret_int はトレースの対象とならない。

T-Kernel/DSのシステムコール(td_～)は、トレースの対象とならない。

フックルーチンは、フック対象となるシステムコールや拡張SVCを呼び出したタスクの準タスク部として実行される。そのため、例えばフックルーチン内での自タスクは、システムコールや拡張SVCを呼び出したタスクと同じである。

システムコール内でタスクディスパッチや割込みが起こる場合があるため、enter() と leave() が常にペアで連続して呼び出されるとは限らない。また、システムコールから戻らない場合は、leave() は呼び出されない。

void *enter (FN fncd , TD_CALINF *calinf , ...);

FN	fncd	機能コード<0 システムコール ≧0 拡張SVC
TD_CALINF*	calinf	呼出元情報
	...	パラメータ(可変個数)

戻値	leave() に引き渡す任意の値

```
typedef struct td_calinf {
        システムコール・拡張SVCの呼出元(アドレス)を特定するための情報で、
        スタックのバックトレースを行うための情報が含まれることが望ましい。
        内容は実装定義となる。
        一般的には、スタックポインタやプログラムカウンタなどのレジスタの値である。
} TD_CALINF;
```

システムコールまたは拡張SVCを呼び出す直前に呼び出される。

戻値に返された値は、そのまま対応する leave() に渡される。これにより、enter() と leave() のペアの確認や任意の情報の受け渡しを行うことができる。

```
exinf = enter(fncd, &calinf, ... )
ret = システムコール・拡張SVCの実行
leave(fncd , ret, exinf)
```

・システムコールの場合

パラメータは、システムコールのパラメータと同じとなる。

例 6.1 tk_wai_sem(ID semid, INT cnt, TMO tmout)の場合

```
enter(TFN_WAI_SEM, &calinf, semid, cnt, tmout)
```

・拡張SVCの場合

パラメータは、拡張SVCハンドラに渡されるパケットの状態となる。

fncd も拡張SVCハンドラに渡されるものと同一である。

enter (FN fncd , TD_CALINF *calinf , void *pk_para);
void leave (FN fncd , INT ret , void *exinf);

FN	fncd	機能コード
INT	ret	システムコールまたは拡張SVCの戻値
void*	exinf	enter() で戻された任意の値

システムコールまたは拡張SVCから戻った直後に呼び出される。

システムコールまたは拡張SVCが呼び出された後(システムコールまたは拡張SVCの実行中)にフックルーチンが設定された場合、enter() が呼び出されずに leave() のみ呼び出される場合がある。このような場合、exinf には NULL が渡される。

逆に、システムコールまたは拡張SVCが呼び出された後フックルーチンが解除された場合、enter() が呼び出されて、leave() が呼び出されない場合がある。

6.2.2 td_hok_dsp - タスクディスパッチのフックルーチン定義

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_hok_dsp (CONST TD_HDSP *hdsp);

パラメータ

CONST TD_HDSP*	hdsp	Dispatcher Hook Routine	フックルーチン定義情報

hdsp の内容

FP	exec	Hook Routine when Execution Starts	実行開始時のフックルーチン
FP	stop	Hook Routine when Execution Stops	実行停止時のフックルーチン

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

タスクディスパッチャに、フックルーチンを設定する。hdsp に NULLを指定することによりフックルーチンを解除する。

フックルーチンは、ディスパッチ禁止状態で呼び出される。フックルーチンでは、T-Kernel/OSのシステムコール(tk_～)および拡張SVCを呼び出してはいけない。T-Kernel/DSのシステムコール(td_～)は呼び出すことができる。

void exec (ID tskid , INT lsid);

ID	tskid	実行を開始・再開するタスクのタスクID
INT	lsid	tskid のタスクの論理空間ID

タスクの実行が開始・再開されるときに呼び出される。exec() が呼び出された時点で、すでに tskid のタスクはRUNNING状態となっており、論理空間も切り替えられている。ただし、tskid のタスクのプログラムコードが実行されるのは、exec() から戻った後である。

void stop (ID tskid , INT lsid , UINT tskstat);

ID	tskid	実行を停止したタスクのタスクID

INT	lsid	tskid のタスクの論理空間ID
UINT	tskstat	tskid のタスクの状態

タスクが実行を停止した時に呼び出される。tskstat には、停止後のタスクの状態が示され、以下のいずれかとなる。

TTS_RDY	READY状態 (実行可能状態)
TTS_WAI	WAITING状態 (待ち状態)
TTS_SUS	SUSPENDED状態 (強制待ち状態)
TTS_WAS	WAITING-SUSPENDED状態 (二重待ち状態)
TTS_DMT	DORMANT状態 (休止状態)
0	NON-EXISTENT状態 (未登録状態)

stop() が呼び出された時点で、すでに tskid のタスクは tskstat で示した状態となっている。論理空間は不定である。

6.2.3 td_hok_int - 割込みハンドラのフックルーチン定義

C言語インタフェース

```
#include <tk/dbgspt.h>
```

ER ercd = td_hok_int (CONST TD_HINT *hint);

パラメータ

CONST TD_HINT*	hint	Interruption Handler Hook Routine	フックルーチン定義情報

hint の内容

FP	enter	Hook Routine before Calling Handler	ハンドラ呼出前のフックルーチン
FP	leave	Hook Routine after Calling Handler	ハンドラ呼出後のフックルーチン

リターンパラメータ

ER	ercd	Error Code	エラーコード

エラーコード

なし

利用可能なコンテキスト

タスク部	準タスク部	タスク独立部
○	○	○

解説

割込みハンドラの呼出前後に、フックルーチンを設定する。フックルーチンの設定は、例外・割込み要因ごとに独立に行うことはできない。すべての例外・割込み要因で共通のフックルーチンを1つのみ設定できる。

hint に NULL を指定することによりフックルーチンを解除する。

フックルーチンはタスク独立部(割込みハンドラの一部)として呼び出される。したがって、フックルーチンからはタスク独立部から発行可能なシステムコールのみ呼び出すことができる。

なお、フックルーチンを設定できるのは、tk_def_int で TA_HLNG 属性を指定して定義された割込みハンドラのみである。TA_ASM 属性の割込みハンドラのフックはできない。TA_ASM 属性の割込みハンドラをフックしたい場合は、例外・割込みベクタテーブルを直接操作してフックするなどの方法があるが、それらの方法は実装によって異なる。

void enter (UINT dintno);
void leave (UINT dintno);

UINT	dintno	割込みハンドラ番号

enter() および leave() に渡される引数は、例外・割込みハンドラに渡される引数と同じものである。実装によっては、dintno 以外の情報も渡される場合がある。

フックルーチンは、高級言語対応ルーチンから次のようにして呼び出される。

```
enter(dintno);
inthdr(dintno); /* 例外・割込みハンドラ */
leave(dintno);
```

enter() は割込み禁止状態で呼び出されることになる。また、割込みを許可してはいけない。leave() は、inthdr() から戻ったときの状態となるため、割込み禁止状態は不確定である。

enter() では、inthdr() で得ることのできる情報と同じだけの情報を得ることができる。逆に、inthdr() で得ることのできない情報は enter() でも得ることはできない。enter() および inthdr() で得ることのできる情報としては、dintno が仕様としては保証されているが、それ以外の情報については実装定義である。なお、leave() では割込み禁止状態など各種の状態が変化している場合があるため、enter() や inthdr() と同じだけの情報を得ることができるとは限らない。

第 7 章

付録

7.1 参考とすべきデバイスドライバ関連の仕様

T-Kernel/SMのデバイス管理機能やデバイスドライバに関連した仕様のうち、T-Kernel本体の最新仕様には含まれないが、T-Kernel仕様書 Ver.1.00.xxのデバイス管理機能の中に記載があり、既存のシステム等でも実装例のあるものについて、本項で説明する。

既存のT-Kernel応用システムのデバイス関連機能や、既存のT-Kernel用デバイスドライバとの相互運用性に配慮を要する場合は、本項で説明する仕様を参考にしていただくのが望ましい。

なお、本項で説明する仕様の詳細、最新情報、その仕様の運用方法などについては、別途確認されたい。

7.1.1 デバイス属性のディスク種別

T-Kernel仕様書 Ver.1.00.xxのデバイス属性の定義においては、以下のディスク種別が定義されている。

```
/* ディスク種別*/
#define TDK_DISK_UNDEF  0x0010  /* その他のディスク */
#define TDK_DISK_RAM    0x0011  /* RAMディスク(主メモリ使用) */
#define TDK_DISK_ROM    0x0012  /* ROMディスク(主メモリ使用) */
#define TDK_DISK_FLA    0x0013  /* Flash ROM、その他のシリコンディスク */
#define TDK_DISK_FD     0x0014  /* フロッピーディスク */
#define TDK_DISK_HD     0x0015  /* ハードディスク */
#define TDK_DISK_CDROM  0x0016  /* CD-ROM */
```

7.1.2 デバイスの属性データ

TDN_DISKINFO
: ディスク情報

T-Kernel仕様書 Ver.1.00.xxのデバイス共通属性の属性データの中で使用するDiskFormatの定義は、以下の通りである。DiskFormatは、属性データのディスク情報(TDN_DISKINFO)で使用するデータ形式DiskInfoの中に含まれている。

```
typedef enum {
        DiskFmt_STD     = 0,            /* 標準(HDなど) */
        DiskFmt_2DD     = 1,            /* 2DD 720KB */
        DiskFmt_2HD     = 2,            /* 2HD 1.44MB */
        DiskFmt_CDROM   = 4             /* CD-ROM 640MB */
} DiskFormat ;
```

TDN_DISPSPEC
: 表示デバイス仕様

T-Kernel仕様書 Ver.1.00.xxのデバイス共通属性の属性データの中で使用するDEV_SPECの定義は、以下の通りである。DEV_SPECは、属性データの表示デバイス仕様(TDN_DISPSPEC)で使用するデータ形式である。

```
typedef struct {
        H       attr;           /* デバイス属性 */
        H       planes;         /* プレーン数 */
        H       pixbits;        /* ピクセルビット数(境界/有効) */
        H       hpixels;        /* 横のピクセル数 */
        H       vpixels;        /* 縦のピクセル数 */
        H       hres;           /* 横の解像度 */
        H       vres;           /* 縦の解像度 */
        H       color[4];       /* カラー情報 */
        H       resv[6];        /* 予約 */
} DEV_SPEC;
```

7.1.3 デバイス事象通知用の事象タイプ

T-Kernel仕様書 Ver.1.00.xxのデバイス事象通知では、以下の事象タイプが定義されている。

```
typedef enum tdevttyp {
        TDE_unknown     = 0,            /* 未定義 */
        TDE_MOUNT       = 0x01,         /* メディア挿入 */
        TDE_EJECT       = 0x02,         /* メディア排出 */
        TDE_ILLMOUNT    = 0x03,         /* メディア不正挿入 */
        TDE_ILLEJECT    = 0x04,         /* メディア不正排出 */
        TDE_REMOUNT     = 0x05,         /* メディア再挿入 */
        TDE_CARDBATLOW  = 0x06,         /* カードバッテリ残量警告 */
        TDE_CARDBATFAIL = 0x07,         /* カードバッテリ異常 */
        TDE_REQEJECT    = 0x08,         /* メディア排出要求 */
        TDE_PDBUT       = 0x11,         /* PD ボタン状態の変化 */
        TDE_PDMOVE      = 0x12,         /* PD 位置移動 */
        TDE_PDSTATE     = 0x13,         /* PDの状態変化 */
        TDE_PDEXT       = 0x14,         /* PD 拡張事象 */
        TDE_KEYDOWN     = 0x21,         /* キーダウン */
        TDE_KEYUP       = 0x22,         /* キーアップ */
        TDE_KEYMETA     = 0x23,         /* メタキー状態の変化 */
        TDE_POWEROFF    = 0x31,         /* 電源スイッチオフ */
        TDE_POWERLOW    = 0x32,         /* 電源残量警告 */
        TDE_POWERFAIL   = 0x33,         /* 電源異常 */
        TDE_POWERSUS    = 0x34,         /* 自動サスペンド */
        TDE_POWERUPTM   = 0x35,         /* 時計更新 */
        TDE_CKPWON      = 0x41          /* 自動電源オン通知 */
} TDEvtTyp;
```

第 8 章

リファレンス

8.1 C言語インタフェース一覧

8.1.1 T-Kernel/OS

8.1.1.1 タスク管理機能

- ID tskid = tk_cre_tsk (CONST T_CTSK *pk_ctsk);
- ER ercd = tk_del_tsk (ID tskid);
- ER ercd = tk_sta_tsk (ID tskid, INT stacd);
- void tk_ext_tsk (void);
- void tk_exd_tsk (void);
- ER ercd = tk_ter_tsk (ID tskid);
- ER ercd = tk_chg_pri (ID tskid, PRI tskpri);
- ER ercd = tk_chg_slt (ID tskid, RELTIM slicetime);
- ER ercd = tk_chg_slt_u (ID tskid, RELTIM_U slicetime_u);
- ER ercd = tk_get_tsp (ID tskid, T_TSKSPC *pk_tskspc);
- ER ercd = tk_set_tsp (ID tskid, CONST T_TSKSPC *pk_tskspc);
- ID resid = tk_get_rid (ID tskid);
- ID oldid = tk_set_rid (ID tskid, ID resid);
- ER ercd = tk_get_reg (ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs);
- ER ercd = tk_set_reg (ID tskid, CONST T_REGS *pk_regs, CONST T_EIT *pk_eit, CONST T_CREGS *pk_cregs);
- ER ercd = tk_get_cpr (ID tskid, INT copno, T_COPREGS *pk_copregs);
- ER ercd = tk_set_cpr (ID tskid, INT copno, CONST T_COPREGS *pk_copregs);
- ER ercd = tk_inf_tsk (ID tskid, T_ITSK *pk_itsk, BOOL clr);
- ER ercd = tk_inf_tsk_u (ID tskid, T_ITSK_U *pk_itsk_u, BOOL clr);
- ER ercd = tk_ref_tsk (ID tskid, T_RTSK *pk_rtsk);
- ER ercd = tk_ref_tsk_u (ID tskid, T_RTSK_U *pk_rtsk_u);

8.1.1.2 タスク付属同期機能

- ER ercd = tk_slp_tsk (TMO tmout);
- ER ercd = tk_slp_tsk_u (TMO_U tmout_u);
- ER ercd = tk_wup_tsk (ID tskid);
- INT wupcnt = tk_can_wup (ID tskid);
- ER ercd = tk_rel_wai (ID tskid);
- ER ercd = tk_sus_tsk (ID tskid);
- ER ercd = tk_rsm_tsk (ID tskid);

- ER ercd = tk_frsm_tsk (ID tskid);
- ER ercd = tk_dly_tsk (RELTIM dlytim);
- ER ercd = tk_dly_tsk_u (RELTIM_U dlytim_u);
- ER ercd = tk_sig_tev (ID tskid, INT tskevt);
- INT tevptn = tk_wai_tev (INT waiptn, TMO tmout);
- INT tevptn = tk_wai_tev_u (INT waiptn, TMO_U tmout_u);
- INT tskwait = tk_dis_wai (ID tskid, UINT waitmask);
- ER ercd = tk_ena_wai (ID tskid);

8.1.1.3 タスク例外処理機能

- ER ercd = tk_def_tex (ID tskid, CONST T_DTEX *pk_dtex);
- ER ercd = tk_ena_tex (ID tskid, UINT texptn);
- ER ercd = tk_dis_tex (ID tskid, UINT texptn);
- ER ercd = tk_ras_tex (ID tskid, INT texcd);
- INT texcd = tk_end_tex (BOOL enatex);
- ER ercd = tk_ref_tex (ID tskid, T_RTEX *pk_rtex);

8.1.1.4 同期・通信機能

- ID semid = tk_cre_sem (CONST T_CSEM *pk_csem);
- ER ercd = tk_del_sem (ID semid);
- ER ercd = tk_sig_sem (ID semid, INT cnt);
- ER ercd = tk_wai_sem (ID semid, INT cnt, TMO tmout);
- ER ercd = tk_wai_sem_u (ID semid, INT cnt, TMO_U tmout_u);
- ER ercd = tk_ref_sem (ID semid, T_RSEM *pk_rsem);
- ID flgid = tk_cre_flg (CONST T_CFLG *pk_cflg);
- ER ercd = tk_del_flg (ID flgid);
- ER ercd = tk_set_flg (ID flgid, UINT setptn);
- ER ercd = tk_clr_flg (ID flgid, UINT clrptn);
- ER ercd = tk_wai_flg (ID flgid, UINT waiptn, UINT wfmode, UINT *p_flgptn, TMO tmout);
- ER ercd = tk_wai_flg_u (ID flgid, UINT waiptn, UINT wfmode, UINT *p_flgptn, TMO_U tmout_u);
- ER ercd = tk_ref_flg (ID flgid, T_RFLG *pk_rflg);
- ID mbxid = tk_cre_mbx (CONST T_CMBX* pk_cmbx);
- ER ercd = tk_del_mbx (ID mbxid);
- ER ercd = tk_snd_mbx (ID mbxid, T_MSG *pk_msg);
- ER ercd = tk_rcv_mbx (ID mbxid, T_MSG **ppk_msg, TMO tmout);
- ER ercd = tk_rcv_mbx_u (ID mbxid, T_MSG **ppk_msg, TMO_U tmout_u);
- ER ercd = tk_ref_mbx (ID mbxid, T_RMBX *pk_rmbx);

8.1.1.5 拡張同期・通信機能

- ID mtxid = tk_cre_mtx (CONST T_CMTX *pk_cmtx);
- ER ercd = tk_del_mtx (ID mtxid);
- ER ercd = tk_loc_mtx (ID mtxid, TMO tmout);
- ER ercd = tk_loc_mtx_u (ID mtxid, TMO_U tmout_u);
- ER ercd = tk_unl_mtx (ID mtxid);
- ER ercd = tk_ref_mtx (ID mtxid, T_RMTX *pk_rmtx);
- ID mbfid = tk_cre_mbf (CONST T_CMBF *pk_cmbf);
- ER ercd = tk_del_mbf (ID mbfid);
- ER ercd = tk_snd_mbf (ID mbfid, CONST void *msg, INT msgsz, TMO tmout);
- ER ercd = tk_snd_mbf_u (ID mbfid, CONST void *msg, INT msgsz, TMO_U tmout_u);
- INT msgsz = tk_rcv_mbf (ID mbfid, void *msg, TMO tmout);
- INT msgsz = tk_rcv_mbf_u (ID mbfid, void *msg, TMO_U tmout_u);
- ER ercd = tk_ref_mbf (ID mbfid, T_RMBF *pk_rmbf);
- ID porid = tk_cre_por (CONST T_CPOR *pk_cpor);
- ER ercd = tk_del_por (ID porid);
- INT rmsgsz = tk_cal_por (ID porid, UINT calptn, void *msg, INT cmsgsz, TMO tmout);
- INT rmsgsz = tk_cal_por_u (ID porid, UINT calptn, void *msg, INT cmsgsz, TMO_U tmout_u);
- INT cmsgsz = tk_acp_por (ID porid, UINT acpptn, RNO *p_rdvno, void *msg, TMO tmout);
- INT cmsgsz = tk_acp_por_u (ID porid, UINT acpptn, RNO *p_rdvno, void *msg, TMO_U tmout_u);
- ER ercd = tk_fwd_por (ID porid, UINT calptn, RNO rdvno, CONST void *msg, INT cmsgsz);
- ER ercd = tk_rpl_rdv (RNO rdvno, CONST void *msg, INT rmsgsz);
- ER ercd = tk_ref_por (ID porid, T_RPOR *pk_rpor);

8.1.1.6 メモリプール管理機能

- ID mpfid = tk_cre_mpf (CONST T_CMPF *pk_cmpf);
- ER ercd = tk_del_mpf (ID mpfid);
- ER ercd = tk_get_mpf (ID mpfid, void **p_blf, TMO tmout);
- ER ercd = tk_get_mpf_u (ID mpfid, void **p_blf, TMO_U tmout_u);
- ER ercd = tk_rel_mpf (ID mpfid, void *blf);
- ER ercd = tk_ref_mpf (ID mpfid, T_RMPF *pk_rmpf);
- ID mplid = tk_cre_mpl (CONST T_CMPL *pk_cmpl);
- ER ercd = tk_del_mpl (ID mplid);
- ER ercd = tk_get_mpl (ID mplid, INT blksz, void **p_blk, TMO tmout);
- ER ercd = tk_get_mpl_u (ID mplid, INT blksz, void **p_blk, TMO_U tmout_u);
- ER ercd = tk_rel_mpl (ID mplid, void *blk);
- ER ercd = tk_ref_mpl (ID mplid, T_RMPL *pk_rmpl);

8.1.1.7 時間管理機能

- ER ercd = tk_set_tim (CONST SYSTIM *pk_tim);
- ER ercd = tk_set_tim_u (SYSTIM_U tim_u);
- ER ercd = tk_get_tim (SYSTIM *pk_tim);
- ER ercd = tk_get_tim_u (SYSTIM_U *tim_u, UINT *ofs);
- ER ercd = tk_get_otm (SYSTIM *pk_tim);
- ER ercd = tk_get_otm_u (SYSTIM_U *tim_u, UINT *ofs);
- ID cycid = tk_cre_cyc (CONST T_CCYC *pk_ccyc);
- ID cycid = tk_cre_cyc_u (CONST T_CCYC_U *pk_ccyc_u);
- ER ercd = tk_del_cyc (ID cycid);
- ER ercd = tk_sta_cyc (ID cycid);
- ER ercd = tk_stp_cyc (ID cycid);
- ER ercd = tk_ref_cyc (ID cycid, T_RCYC *pk_rcyc);
- ER ercd = tk_ref_cyc_u (ID cycid, T_RCYC_U *pk_rcyc_u);
- ID almid = tk_cre_alm (CONST T_CALM *pk_calm);
- ER ercd = tk_del_alm (ID almid);
- ER ercd = tk_sta_alm (ID almid, RELTIM almtim);
- ER ercd = tk_sta_alm_u (ID almid, RELTIM_U almtim_u);
- ER ercd = tk_stp_alm (ID almid);
- ER ercd = tk_ref_alm (ID almid, T_RALM *pk_ralm);
- ER ercd = tk_ref_alm_u (ID almid, T_RALM_U *pk_ralm_u);

8.1.1.8 割込み管理機能

- ER ercd = tk_def_int (UINT dintno, CONST T_DINT *pk_dint);
- void tk_ret_int (void);

8.1.1.9 システム状態管理機能

- ER ercd = tk_rot_rdq (PRI tskpri);
- ID tskid = tk_get_tid (void);
- ER ercd = tk_dis_dsp (void);
- ER ercd = tk_ena_dsp (void);
- ER ercd = tk_ref_sys (T_RSYS *pk_rsys);
- ER ercd = tk_set_pow (UINT powmode);
- ER ercd = tk_ref_ver (T_RVER *pk_rver);

8.1.1.10 サブシステム管理機能

- ER ercd = tk_def_ssy (ID ssid, CONST T_DSSY *pk_dssy);
- ER ercd = tk_sta_ssy (ID ssid, ID resid, INT info);
- ER ercd = tk_cln_ssy (ID ssid, ID resid, INT info);
- ER ercd = tk_evt_ssy (ID ssid, INT evttyp, ID resid, INT info);
- ER ercd = tk_ref_ssy (ID ssid, T_RSSY *pk_rssy);
- ER ercd = tk_cre_res (void);
- ER ercd = tk_del_res (ID resid);
- ER ercd = tk_get_res (ID resid, ID ssid, void **p_resblk);

8.1.2 T-Kernel/SM

8.1.2.1 システムメモリ管理機能

- ER ercd = tk_get_smb (void **addr, INT nblk, UINT attr);
- ER ercd = tk_rel_smb (void *addr);
- ER ercd = tk_ref_smb (T_RSMB *pk_rsmb);
- void* Vmalloc (size_t size);
- void* Vcalloc (size_t nmemb, size_t size);
- void* Vrealloc (void *ptr, size_t size);
- void Vfree (void *ptr);
- void* Kmalloc (size_t size);
- void* Kcalloc (size_t nmemb, size_t size);
- void* Krealloc (void *ptr, size_t size);
- void Kfree (void *ptr);

8.1.2.2 アドレス空間管理機能

- ER ercd = SetTaskSpace (ID tskid);
- ER ercd = ChkSpaceR (CONST void *addr, INT len);
- ER ercd = ChkSpaceRW (CONST void *addr, INT len);
- ER ercd = ChkSpaceRE (CONST void *addr, INT len);
- INT rlen = ChkSpaceBstrR (CONST UB *str, INT max);
- INT rlen = ChkSpaceBstrRW (CONST UB *str, INT max);
- INT rlen = ChkSpaceTstrR (CONST TC *str, INT max);
- INT rlen = ChkSpaceTstrRW (CONST TC *str, INT max);
- ER ercd = LockSpace (CONST void *addr, INT len);

- ER ercd = UnlockSpace (CONST void *addr, INT len);
- INT rlen = CnvPhysicalAddr (CONST void *vaddr, INT len, void **paddr);
- ER ercd = MapMemory (CONST void *paddr, INT len, UINT attr, void **laddr);
- ER ercd = UnmapMemory (CONST void *laddr);
- ER ercd = GetSpaceInfo (CONST void *addr, INT len, T_SPINFO *pk_spinfo);
- INT rlen = SetMemoryAccess (CONST void *addr, INT len, UINT mode);

8.1.2.3 デバイス管理機能

- ID dd = tk_opn_dev (CONST UB *devnm, UINT omode);
- ER ercd = tk_cls_dev (ID dd, UINT option);
- ID reqid = tk_rea_dev (ID dd, W start, void *buf, W size, TMO tmout);
- ID reqid = tk_rea_dev_du (ID dd, D start_d, void *buf, W size, TMO_U tmout_u);
- ER ercd = tk_srea_dev (ID dd, W start, void *buf, W size, W *asize);
- ER ercd = tk_srea_dev_d (ID dd, D start_d, void *buf, W size, W *asize);
- ID reqid = tk_wri_dev (ID dd, W start, CONST void *buf, W size, TMO tmout);
- ID reqid = tk_wri_dev_du (ID dd, D start_d, CONST void *buf, W size, TMO_U tmout_u);
- ER ercd = tk_swri_dev (ID dd, W start, CONST void *buf, W size, W *asize);
- ER ercd = tk_swri_dev_d (ID dd, D start_d, CONST void *buf, W size, W *asize);
- ID creqid = tk_wai_dev (ID dd, ID reqid, W *asize, ER *ioer, TMO tmout);
- ID creqid = tk_wai_dev_u (ID dd, ID reqid, W *asize, ER *ioer, TMO_U tmout_u);
- INT dissus = tk_sus_dev (UINT mode);
- ID pdevid = tk_get_dev (ID devid, UB *devnm);
- ID devid = tk_ref_dev (CONST UB *devnm, T_RDEV *rdev);
- ID devid = tk_oref_dev (ID dd, T_RDEV *rdev);
- INT remcnt = tk_lst_dev (T_LDEV *ldev, INT start, INT ndev);
- INT retcode = tk_evt_dev (ID devid, INT evttyp, void *evtinf);
- ID devid = tk_def_dev (CONST UB *devnm, CONST T_DDEV *ddev, T_IDEV *idev);
- ER ercd = tk_ref_idv (T_IDEV *idev);
- ER ercd = openfn (ID devid, UINT omode, void *exinf);
- ER ercd = closefn (ID devid, UINT option, void *exinf);
- ER ercd = execfn (T_DEVREQ *devreq, TMO tmout, void *exinf);
- ER ercd = execfn (T_DEVREQ_D *devreq_d, TMO tmout, void *exinf);
- ER ercd = execfn (T_DEVREQ *devreq, TMO_U tmout_u, void *exinf);
- ER ercd = execfn (T_DEVREQ_D *devreq_d, TMO_U tmout_u, void *exinf);
- INT creqno = waitfn (T_DEVREQ *devreq, INT nreq, TMO tmout, void *exinf);

- INT creqno = waitfn (T_DEVREQ_D *devreq_d, INT nreq, TMO tmout, void *exinf);
- INT creqno = waitfn (T_DEVREQ *devreq, INT nreq, TMO_U tmout_u, void *exinf);
- INT creqno = waitfn (T_DEVREQ_D *devreq_d, INT nreq, TMO_U tmout_u, void *exinf);
- ER ercd = abortfn (ID tskid, T_DEVREQ *devreq, INT nreq, void *exinf);
- ER ercd = abortfn (ID tskid, T_DEVREQ_D *devreq_d, INT nreq, void *exinf);
- INT retcode = eventfn (INT evttyp, void *evtinf, void *exinf);

8.1.2.4 割込み管理機能

- DI (UINT intsts);
- EI (UINT intsts);
- BOOL disint = isDI (UINT intsts);
- UINT dintno = DINTNO (INTVEC intvec);
- void EnableInt (INTVEC intvec);
- void EnableInt (INTVEC intvec, INT level);
- void DisableInt (INTVEC intvec);
- void ClearInt (INTVEC intvec);
- void EndOfInt (INTVEC intvec);
- BOOL rasint = CheckInt (INTVEC intvec);
- void SetIntMode (INTVEC intvec, UINT mode);

8.1.2.5 I/Oポートアクセスサポート機能

- void out_b (INT port, UB data);
- void out_h (INT port, UH data);
- void out_w (INT port, UW data);
- void out_d (INT port, UD data);
- UB data = in_b (INT port);
- UH data = in_h (INT port);
- UW data = in_w (INT port);
- UD data = in_d (INT port);
- void WaitUsec (UINT usec);
- void WaitNsec (UINT nsec);

8.1.2.6 省電力機能

- void low_pow (void);
- void off_pow (void);

8.1.2.7 システム構成情報管理機能

- INT ct = tk_get_cfn (CONST UB *name, INT *val, INT max);
- INT rlen = tk_get_cfs (CONST UB *name, UB *buf, INT max);

8.1.2.8 メモリキャッシュ制御機能

- INT rlen = SetCacheMode (void *addr, INT len, UINT mode);
- INT rlen = ControlCache (void *addr, INT len, UINT mode);

8.1.2.9 物理タイマ機能

- ER ercd = StartPhysicalTimer (UINT ptmrno, UW limit, UINT mode);
- ER ercd = StopPhysicalTimer (UINT ptmrno);
- ER ercd = GetPhysicalTimerCount (UINT ptmrno, UW *p_count);
- ER ercd = DefinePhysicalTimerHandler (UINT ptmrno, CONST T_DPTMR *pk_dptmr);
- ER ercd = GetPhysicalTimerConfig (UINT ptmrno, T_RPTMR *pk_rptmr);

8.1.2.10 ユーティリティ機能

- void SetOBJNAME (void *exinf, CONST UB *name);
- ER ercd = CreateLock (FastLock *lock, CONST UB *name);
- void DeleteLock (FastLock *lock);
- void Lock (FastLock *lock);
- void Unlock (FastLock *lock);
- ER ercd = CreateMLock (FastMLock *lock, CONST UB *name);
- ER ercd = DeleteMLock (FastMLock *lock);
- ER ercd = MLock (FastMLock *lock, INT no);
- ER ercd = MLockTmo (FastMLock *lock, INT no, TMO tmout);
- ER ercd = MLockTmo_u (FastMLock *lock, INT no, TMO_U tmout_u);
- ER ercd = MUnlock (FastMLock *lock, INT no);

8.1.3 T-Kernel/DS

8.1.3.1 カーネル内部状態取得機能

- INT ct = td_lst_tsk (ID list[], INT nent);
- INT ct = td_lst_sem (ID list[], INT nent);
- INT ct = td_lst_flg (ID list[], INT nent);
- INT ct = td_lst_mbx (ID list[], INT nent);

- INT ct = td_lst_mtx (ID list[], INT nent);
- INT ct = td_lst_mbf (ID list[], INT nent);
- INT ct = td_lst_por (ID list[], INT nent);
- INT ct = td_lst_mpf (ID list[], INT nent);
- INT ct = td_lst_mpl (ID list[], INT nent);
- INT ct = td_lst_cyc (ID list[], INT nent);
- INT ct = td_lst_alm (ID list[], INT nent);
- INT ct = td_lst_ssy (ID list[], INT nent);
- INT ct = td_rdy_que (PRI pri, ID list[], INT nent);
- INT ct = td_sem_que (ID semid, ID list[], INT nent);
- INT ct = td_flg_que (ID flgid, ID list[], INT nent);
- INT ct = td_mbx_que (ID mbxid, ID list[], INT nent);
- INT ct = td_mtx_que (ID mtxid, ID list[], INT nent);
- INT ct = td_smbf_que (ID mbfid, ID list[], INT nent);
- INT ct = td_rmbf_que (ID mbfid, ID list[], INT nent);
- INT ct = td_cal_que (ID porid, ID list[], INT nent);
- INT ct = td_acp_que (ID porid, ID list[], INT nent);
- INT ct = td_mpf_que (ID mpfid, ID list[], INT nent);
- INT ct = td_mpl_que (ID mplid, ID list[], INT nent);
- ER ercd = td_ref_tsk (ID tskid, TD_RTSK *rtsk);
- ER ercd = td_ref_tsk_u (ID tskid, TD_RTSK_U *rtsk_u);
- ER ercd = td_ref_tex (ID tskid, TD_RTEX *pk_rtex);
- ER ercd = td_ref_sem (ID semid, TD_RSEM *rsem);
- ER ercd = td_ref_flg (ID flgid, TD_RFLG *rflg);
- ER ercd = td_ref_mbx (ID mbxid, TD_RMBX *rmbx);
- ER ercd = td_ref_mtx (ID mtxid, TD_RMTX *rmtx);
- ER ercd = td_ref_mbf (ID mbfid, TD_RMBF *rmbf);
- ER ercd = td_ref_por (ID porid, TD_RPOR *rpor);
- ER ercd = td_ref_mpf (ID mpfid, TD_RMPF *rmpf);
- ER ercd = td_ref_mpl (ID mplid, TD_RMPL *rmpl);
- ER ercd = td_ref_cyc (ID cycid, TD_RCYC *rcyc);
- ER ercd = td_ref_cyc_u (ID cycid, TD_RCYC_U *rcyc_u);
- ER ercd = td_ref_alm (ID almid, TD_RALM *ralm);
- ER ercd = td_ref_alm_u (ID almid, TD_RALM_U *ralm_u);
- ER ercd = td_ref_sys (TD_RSYS *pk_rsys);

- ER ercd = td_ref_ssy (ID ssid, TD_RSSY *rssy);
- ER ercd = td_inf_tsk (ID tskid, TD_ITSK *pk_itsk, BOOL clr);
- ER ercd = td_inf_tsk_u (ID tskid, TD_ITSK_U *itsk_u, BOOL clr);
- ER ercd = td_get_reg (ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs);
- ER ercd = td_set_reg (ID tskid, CONST T_REGS *pk_regs, CONST T_EIT *pk_eit, CONST T_CREGS *pk_cregs);
- ER ercd = td_get_tim (SYSTIM *tim, UINT *ofs);
- ER ercd = td_get_tim_u (SYSTIM_U *tim_u, UINT *ofs);
- ER ercd = td_get_otm (SYSTIM *tim, UINT *ofs);
- ER ercd = td_get_otm_u (SYSTIM_U *tim_u, UINT *ofs);
- ER ercd = td_ref_dsname (UINT type, ID id, UB *dsname);
- ER ercd = td_set_dsname (UINT type, ID id, CONST UB *dsname);

8.1.3.2 実行トレース機能

- ER ercd = td_hok_svc (CONST TD_HSVC *hsvc);
- ER ercd = td_hok_dsp (CONST TD_HDSP *hdsp);
- ER ercd = td_hok_int (CONST TD_HINT *hint);

8.2 エラーコード一覧

8.2.1 正常終了のエラークラス (0)

エラーコードの名称	エラーコード	説明
E_OK	0	正常終了

8.2.2 内部エラークラス (5～8)

エラーコードの名称	エラーコード	説明
E_SYS	ERCD(-5, 0)	システムエラー

原因不明のエラーであり、システム全体に影響するエラーである。

エラーコードの名称	エラーコード	説明
E_NOCOP	ERCD(-6, 0)	コプロセッサ使用不可

現在動作中のハードウェアに指定のコプロセッサが搭載されていない。または、コプロセッサの動作異常を検出した。

8.2.3 未サポートエラークラス (9～16)

エラーコードの名称	エラーコード	説明
E_NOSPT	ERCD(-9, 0)	未サポート機能

システムコールの一部の機能がサポートされていない場合に、その機能を指定すると、E_RSATR または E_NOSPT のエラーを発生する。E_RSATR に該当しない場合には、E_NOSPT のエラーとなる。

エラーコードの名称	エラーコード	説明
E_RSFN	ERCD(-10, 0)	予約機能コード番号

予約機能コード(未定義の機能コード)を指定してシステムコールを実行しようとした場合に、このエラーが発生する。未定義の拡張SVCハンドラを実行しようとした場合(機能コードが正の場合)にも、このエラーが発生する。

エラーコードの名称	エラーコード	説明
E_RSATR	ERCD(-11, 0)	予約属性

未定義やサポートしていないオブジェクト属性を指定した場合に発生する。

システム依存の適応化を行う場合、このエラーのチェックは省略されることがある。

8.2.4 パラメータエラークラス (17～24)

エラーコードの名称	エラーコード	説明
E_PAR	ERCD(-17, 0)	パラメータエラー

システム依存の適応化を行う場合、このエラーのチェックは省略されることがある。

エラーコードの名称	エラーコード	説明
E_ID	ERCD(-18, 0)	不正ID番号

E_ID はID番号を持つオブジェクトに対してのみ発生するエラーである。

割込みハンドラ番号などの範囲外や予約番号といった静的なエラーが検出された場合には、E_PAR のエラーが発生する。

8.2.5 呼出コンテキストエラークラス (25～32)

エラーコードの名称	エラーコード	説明
E_CTX	ERCD(-25, 0)	コンテキストエラー

このシステムコールを発行できるコンテキスト(タスク部／タスク独立部の区別やハンドラ実行状態) にはないということを示すエラーである。

自タスクを待ち状態にするシステムコールをタスク独立部から発行した場合のように、システムコールの発行コンテキストに関して意味的な間違いのある場合には、必ずこのエラーが発生する。また、それ以外のシステムコールであっても、実装の制約のため、あるコンテキスト(割込みハンドラなど)からそのシステムコールを発行できない場合に、このエラーが発生する。

エラーコードの名称	エラーコード	説明
E_MACV	ERCD(-26, 0)	メモリアクセス不能,メモリアクセス権違反

エラーの検出は実装依存である。

エラーコードの名称	エラーコード	説明
E_OACV	ERCD(-27, 0)	オブジェクトアクセス権違反

ユーザタスクがシステムオブジェクトを操作した場合に発生する。

システムオブジェクトの定義およびエラーの検出は実装依存である。

エラーコードの名称	エラーコード	説明
E_ILUSE	ERCD(-28, 0)	システムコール不正使用

8.2.6 資源不足エラークラス (33～40)

エラーコードの名称	エラーコード	説明
E_NOMEM	ERCD(-33, 0)	メモリ不足

オブジェクト管理ブロック領域、ユーザスタック領域、メモリプール領域、メッセージバッファ領域などを獲得する時のメモリ不足(no memory)

エラーコードの名称	エラーコード	説明
E_LIMIT	ERCD(-34, 0)	システムの制限を超過

オブジェクト数の上限を超えてオブジェクトを生成しようとした場合など。

8.2.7 オブジェクト状態エラークラス (41～48)

エラーコードの名称	エラーコード	説明
E_OBJ	ERCD(-41, 0)	オブジェクトの状態が不正
E_NOEXS	ERCD(-42, 0)	オブジェクトが存在していない
E_QOVR	ERCD(-43, 0)	キューイングまたはネストのオーバーフロー

8.2.8　待ち解除エラークラス (49～56)

エラーコードの名称	エラーコード	説明
E_RLWAI	ERCD(-49, 0)	待ち状態強制解除
E_TMOUT	ERCD(-50, 0)	ポーリング失敗またはタイムアウト
E_DLT	ERCD(-51, 0)	待ちオブジェクトが削除された
E_DISWAI	ERCD(-52, 0)	待ち禁止による待ち解除

8.2.9　デバイスエラークラス (57～64) (T-Kernel/SM)

エラーコードの名称	エラーコード	説明
E_IO	ERCD(-57, 0)	入出力エラー

※ E_IO のサブエラーコードには、デバイスごとにエラー状態等を示す値が定義される場合がある。

エラーコードの名称	エラーコード	説明
E_NOMDA	ERCD(-58, 0)	メディアがない

8.2.10　各種状態エラークラス (65～72) (T-Kernel/SM)

エラーコードの名称	エラーコード	説明
E_BUSY	ERCD(-65, 0)	ビジー状態
E_ABORT	ERCD(-66, 0)	中止した
E_RONLY	ERCD(-67, 0)	書込み禁止