μT-Kernel 実装仕様書

H8S/2212 版

Version 1.01.02

2013年4月

はじめに

本書では、μT-Kernel の特定のボードへの実装の仕様を記載する。

対象とするμT-Kernel ソースコードのバージョンは 1.01.01 版 である。
準拠するμT-Kernel 仕様書のバージョンは 1.01.01 版 である。

本書に記された仕様は、μT-Kernel 仕様のハードウェアに依存した実装依存部に相当する。
μT-Kernel の仕様については μT-Kernel 仕様書を参照。
また、ボードや CPU などハードウェアの仕様については該当する各仕様書を参照。

［目次］

1. CPU

1.1 ハードウェア仕様

CPU : H8S/2212 (HD64F2212)
Renesas Technology Corp.
ROM : 128 KB (内蔵 FlashROM)
RAM : 12 KB (内蔵 SRAM)

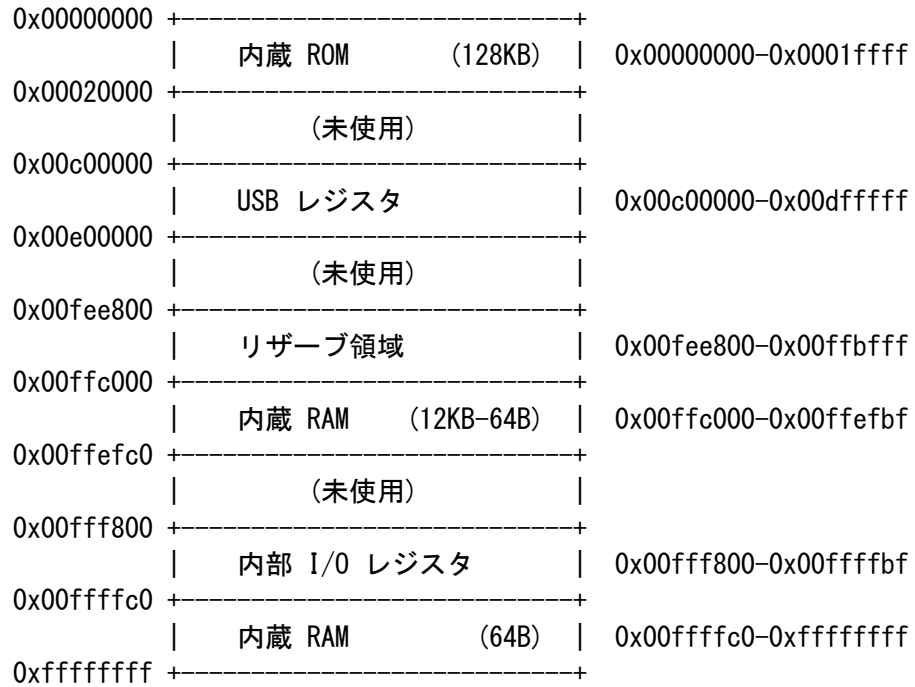
1.2 動作モードと保護レベル

本システムは単一の CPU 動作モードで動作する為、保護レベルの切替はない。
どの保護レベルが指定されても保護レベル 0 として扱う。

2. メモリマップ

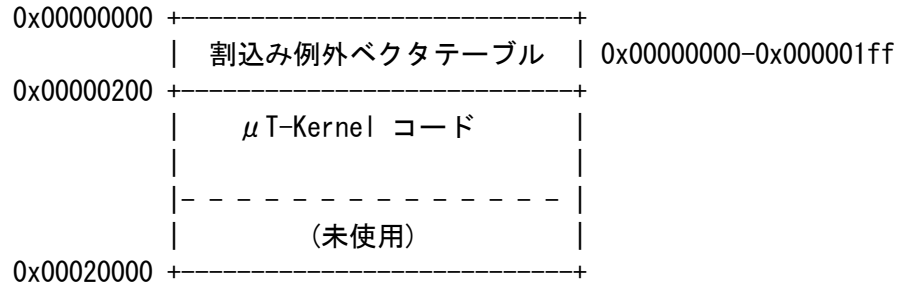
2.1 全体メモリマップ

システム全体のメモリマップを以下に示す。



2.2 ROM 領域メモリマップ

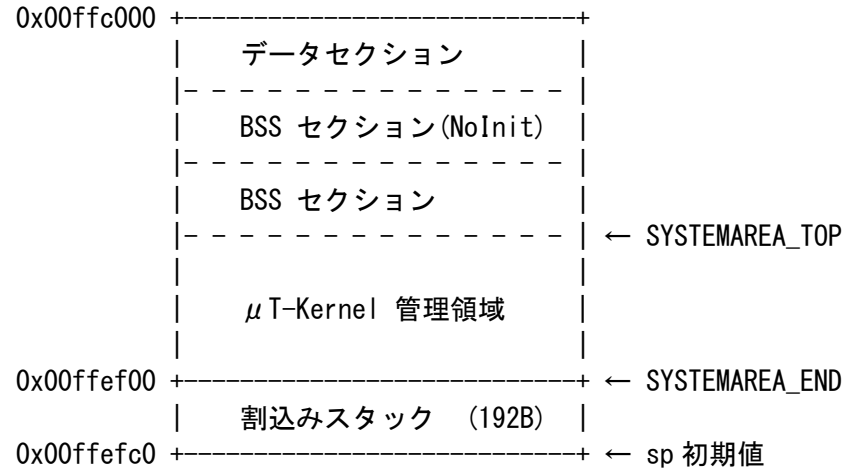
内蔵 ROM は 128 KB の領域が実装されている。
内蔵 ROM 領域のメモリマップを以下に示す。



内蔵 ROM には、割込み例外ベクタテーブルとμT-Kernel コードを配置する。

2.3 RAM 領域メモリマップ

内蔵 RAM は 12KB の領域が実装されている。
内蔵 RAM 領域のメモリマップを以下に示す。



NoInit： ゼロ初期化されない BSS セクション

内蔵 RAM の下位アドレスから、データセクションと BSS セクションを配置する。
μT-Kernel 管理領域は、μT-Kernel のメモリ管理機能で使用する領域である。
通常は、空いているメモリ領域を全て μT-Kernel 管理領域に割り当るが、設定により変更が可能である。μT-Kernel 管理領域は、設定ファイルの SYSTEMAREA_TOP と SYSTEMAREA_END で指定された間の領域となる。

2.4 スタック

μT-Kernel では以下の 2 種類のスタックがある。

(1) システムスタック

割込みハンドラ以外で使用するスタックで、タスク毎に 1 本ずつ存在する。
μT-Kernel には保護レベルの概念が無いため、T-Kernel のようにユーザスタックとシステムスタックの使い分けはない。

(2) 割込みスタック

割込みハンドラで使用するスタックで、システムスタックとは独立したスタック領域が割り当てられる。
割込みスタックはシステムで共有されるため、使用中にタスクスイッチが起きてはいけない。

3. 割込みおよび例外

3.1 割込み定義番号

tk_def_int()で定義する割込み定義番号(dintno)は、ベクタ番号のイミディエート値 (0-127)を使用する。
割込み例外ベクタテーブルを、以下に示す。

ベクタ番号	ベクタアドレス	割込み要因
0	0x0000	パワーオンリセット
1	0x0004	マニュアルリセット
2	0x0008	システム予約
3	0x000C	システム予約
4	0x0010	システム予約
5	0x0014	トレース
6	0x0018	直接遷移
7	0x001C	外部割込み NMI
8	0x0020	トラップ命令 (#0)
9	0x0024	トラップ命令 (#1)
10	0x0028	トラップ命令 (#2)
11	0x002C	トラップ命令 (#3)
12	0x0030	システム予約
13	0x0034	システム予約
14	0x0038	システム予約
15	0x003C	システム予約
16	0x0040	外部割込み IRQ0
17	0x0044	外部割込み IRQ1
18	0x0048	外部割込み IRQ2
19	0x004C	外部割込み IRQ3
20	0x0050	外部割込み IRQ4
21	0x0054	RTC 割込み IRQ5
22	0x0058	USB 割込み IRQ6
23	0x005C	外部割込み IRQ7
24	0x0060	(未割当)
25	0x0064	ウォッチドッグタイマ WOVI
26	0x0068	(未割当)
27	0x006C	(未割当)
28	0x0070	A/D ADI
29	0x0074	(未割当)
30	0x0078	(未割当)
31	0x007C	(未割当)
32	0x0080	TPU チャネル 0 TGI0A
33	0x0084	TPU チャネル 0 TGI0B
34	0x0088	TPU チャネル 0 TGI0C
35	0x008C	TPU チャネル 0 TGI0D
36	0x0090	TPU チャネル 0 TCI0V

37	0x0094	(未割当)
38	0x0098	(未割当)
39	0x009C	(未割当)
40	0x00A0	TPU チャネル 1 TGI1A
41	0x00A4	TPU チャネル 1 TGI1B
42	0x00A8	TPU チャネル 1 TCI1V
43	0x00AC	TPU チャネル 1 TCI1U
44	0x00B0	TPU チャネル 2 TGI2A
45	0x00B4	TPU チャネル 2 TGI2B
46	0x00B8	TPU チャネル 2 TCI2V
47	0x00BC	TPU チャネル 2 TCI2U
48	0x00C0	(未割当)
49	0x00C4	(未割当)
50	0x00C8	(未割当)
51	0x00CC	(未割当)
52	0x00D0	(未割当)
53	0x00D4	(未割当)
54	0x00D8	(未割当)
55	0x00DC	(未割当)
56	0x00E0	(未割当)
57	0x00E4	(未割当)
58	0x00E8	(未割当)
59	0x00EC	(未割当)
60	0x00F0	(未割当)
61	0x00F4	(未割当)
62	0x00F8	(未割当)
63	0x00FC	(未割当)
64	0x0100	(未割当)
65	0x0104	(未割当)
66	0x0108	(未割当)
67	0x010C	(未割当)
68	0x0110	(未割当)
69	0x0114	(未割当)
70	0x0118	(未割当)
71	0x011C	(未割当)
72	0x0120	DMAC DEND0A
73	0x0124	DMAC DEND0B
74	0x0128	DMAC DEND1A
75	0x012C	DMAC DEND1B
76	0x0130	(未割当)
77	0x0134	(未割当)
78	0x0138	(未割当)
79	0x013C	(未割当)
80	0x0140	SCI チャネル 0 ERI0
81	0x0144	SCI チャネル 0 RXI0
82	0x0148	SCI チャネル 0 TXI0
83	0x014C	SCI チャネル 0 TEI0
84	0x0150	(未割当)
85	0x0154	(未割当)
86	0x0158	(未割当)

87	0x015C	(未割当)
88	0x0160	SCI チャネル 2 ERI2
89	0x0164	SCI チャネル 2 RXI2
90	0x0168	SCI チャネル 2 TXI2
91	0x016C	SCI チャネル 2 TEI2
92	0x0170	(未割当)
93	0x0174	(未割当)
94	0x0178	(未割当)
95	0x017C	(未割当)
96	0x0180	(未割当)
97	0x0184	(未割当)
98	0x0188	(未割当)
99	0x018C	(未割当)
100	0x0190	(未割当)
101	0x0194	(未割当)
102	0x0198	(未割当)
103	0x019C	(未割当)
104	0x01A0	USB EXIRQ0
105	0x01A4	USB EXIRQ1
106	0x01A8	(未割当)
107	0x01AC	(未割当)
108	0x01B0	(未割当)
109	0x01B4	(未割当)
110	0x01B8	(未割当)
111	0x01BC	(未割当)
112	0x01C0	(未割当)
113	0x01C4	(未割当)
114	0x01C8	(未割当)
115	0x01CC	(未割当)
116	0x01D0	(未割当)
117	0x01D4	(未割当)
118	0x01D8	(未割当)
119	0x01DC	(未割当)
120	0x01E0	(未割当)
121	0x01E4	(未割当)
122	0x01E8	(未割当)
123	0x01EC	(未割当)
124	0x01F0	(未割当)
125	0x01F4	(未割当)
126	0x01F8	(未割当)
127	0x01FC	(未割当)

3.2 TRAPA 命令の割り当て

TRAPA 命令は 0～3 を使用し、割込み定義番号の 8～11 に割り当てられている。
各 TRAPA 命令は以下のように使用される。

trapa 0	μT-Kernel システムコール・拡張 SVC
trapa 1	tk_ret_int() システムコール
trapa 2	タスクディスパッチャ
trapa 3	デバッガサポート機能

3.3 割込みハンドラ

割込みハンドラを定義する場合は、vector.S 内で、対応するベクタ番号にハンドラのアドレスを定義する必要がある。vector.S に定義されたベクタ番号は、tk_def_int で使用することができる。

割込みが発生すると設定したアドレスに直に飛ぶので、飛んだ先の割込みハンドラではコンテキストの保存などを行わなくてはならない。

エントリルーチンは、INT_ENTRY マクロで定義する。アセンブラファイル内で“INT_ENTRY vecno”と記述すると、knl_inthdr_entryN（N は割込みベクタ番号）という名前で割込みハンドラのエントリルーチンが生成される。かつ、vector.S に knl_inthdr_entryN を定義すると、そのベクタ番号を使用することができる。

このエントリルーチンは、以下の処理を行う。

- er0, er1 をスタックに保存
- er0 に割込みベクタ番号を設定
- tk_def_int 内で設定したハンドラにジャンプ

tk_ret_int はこの処理を前提としているので、INT_ENTRY マクロを使わずにハンドラを定義する場合は、スタックに er0, er1 を保存する必要がある。

また、tk_ret_int で遅延ディスパッチを実現するために、高級言語対応ルーチン内で割込みのネスト数（knl_int_nest）をインクリメントしている。高級言語対応ルーチンを使わない場合は、自分で knl_int_nest をインクリメントする必要がある。

utk_config_depend.h の USE_FULL_VECTOR を 1 に設定すると、自動的に全てのベクタに対して INT_ENTRY マクロが使用され、全ての割込みが tk_def_int で扱えるようになる。
デフォルト設定では、USE_FULL_VECTOR を 1 とする。

4. 初期化および起動処理

4.1 μT-Kernel の起動手順

システムがリセットされるとμT-Kernel が起動する。
μT-Kernel が起動してから、main 関数が呼ばれるまでのμT-Kernel の起動手順を以下に示す。

icrt0.S
(1) スタックポインタの設定 [start:]
(2) CCR の初期化 [flashrom_init:]
(3) EXR の初期化 [flashrom_init:]
(4) データセクションの初期値設定(ROM->RAM) [data_loop:]
(5) BSS セクションのゼロクリア [bss_loop:]
(6) μT-Kernel 管理領域の範囲計算 [bss_done:]
(7) main 関数(sysinit_main.c) の呼出し [kernel_start:]

4.2 ユーザー初期化プログラム

ユーザー初期化プログラムは、ユーザー定義のシステム起動処理/終了処理を実行するためのルーチンである。ユーザー初期化プログラムは、初期タスクから次の形式で呼び出される。

```
INT userinit( INT flag )

flag    = 0             起動時呼出し
        = -1            終了時呼出し

戻り値  1               usermain()を起動
        それ以外        システム終了
```

システム起動時に flag = 0 で呼出され、システム終了時に flag = -1 で呼出される。終了時の呼出では戻り値は無視される。処理の概略は次のようになる。

```
fin = userinit(0);
if ( fin > 0 ){
        usermain();
}
userinit(-1);
```

ユーザー初期化プログラムは初期タスクのコンテキストで実行される。タスク優先度は(CFN_MAX_PRI-2)である。

5. μT-Kernel 実装仕様

5.1 システム状態判定

(1) タスク独立部（割込みハンドラ、タイムイベントハンドラ）

μT-Kernel 内にソフトウェア的なフラグを設けて判定する。
knl_taskindp = 0　タスク部
knl_taskindp > 0　タスク独立部

(2) 準タスク部（拡張 SVC ハンドラ）

μT-Kernel 内にソフトウェア的なフラグを設けて判定する。
TCB の sysmode = 0　タスク部
TCB の sysmode > 0　準タスク部

5.2 μT-Kernel で使用する例外・割込み

trapa 0	μT-Kernel システムコール・拡張 SVC
trapa 1	tk_ret_int() システムコール
trapa 2	タスクディスパッチャ
trapa 3	デバッガサポート機能
dintno 32	プログラマブルタイマーA(TGIOA)

5.3 システムコール/拡張 SVC のインターフェース

呼出し側は、C 言語の関数の呼出形式で、インターフェースライブラリを呼出す方法と直接呼出す方法を選択できる（設定ファイルで選択可能）。
アセンブラから呼び出す場合も、C 言語と同様に関数形式によりインターフェースライブラリを経由して呼び出すこととするが、下記のインターフェースライブラリ相当のことを行い、直接 TRAPA 命令で呼び出してもよい。その場合も、C 言語の規則にしたがってレジスタの保存を行う必要がある。

インターフェースライブラリの基本的な処理は以下のようになる。

・ R0 レジスタに機能コードを設定して TRAPA #1 により呼び出す。
機能コードが負の値ならシステムコール、0 または正の値なら拡張 SVC となる。
ただし、デバッガサポート機能のサービスコールは TRAPA #3 を使用する。

・ レジスタの保存規則は C 言語レジスタ保存規則に従い以下の通りとなる。

テンポラリレジスタ	R0～R2,
パーマネントレジスタ	R3～R6,
スタックポインタ	R7
未使用	EXR

引数	R0～R2
戻り値	R0

テンポラリレジスタは関数の呼び出しによって破壊される。それ以外のレジスタは保存される。

(1) システムコールのインターフェース

システムコールは、第 3 引数まではレジスタに設定し、第 4 引数以降はスタックに積んで、TRAPA #1 (trapa #TRAP_SVC)により呼び出す。システムコールのインターフェースの実装例を以下に示す。

```
ER tk_xxx_yyy(p1, p2, p3, p4, p5)

//                  stack state
//   High Address +---------------+
//                | p5            |
//                | p4            |
//                | SPC(24bit)    | saved by I/F call
//          SP => | SCCR(8bit)    |
//    Low Address +---------------+
//
//                     er0 = p1
//                     er1 = p2
//                     er2 = p3
Csym(tk_xxx_yyy):
        mov.w   r0, @-er7
        mov.w   機能コード, r0
#if USE_TRAP
        trapa   #TRAP_SVC
#else
        jsr     Csym(knl_call_entry)
#endif
        inc.l   #2, er7
        rts
#endif
```

(2) 拡張 SVC のインターフェースライブラリ

引数は全てパケット化し、パケットの先頭アドレスを er1 レジスタに設定して TRAPA #1(trapa #TRAP_SVC)により呼び出す。パケットは通常スタックに作成するが、他の場所でも構わない。引数はパケット化するため、数や型に制限はない。

拡張 SVC のインターフェースの実装例を以下に示す。

```
        W zxxx_yyy(p1, p2, p3, p4, p5, p6)

Csym(${func}):
        mov.l   er1, @-er7      // レジスタ上の引数をスタックに積む
        mov.l   er0, @-er7
        mov.l   er7, er1        // er1 = 引数パケットのアドレス
        mov.l   er2, @-er7
        mov.w   @zxxx_yyy, r0
        trapa   #TRAP_SVC
        add.l   #3*4, er7
        rts
```

(3) デバッガサポート機能 システムコールのインターフェース

デバッガサポート機能 システムコールは、基本的には他の μT-Kernel のサービスコールと同じだが、TRAPA #4(trapa #TRAP_DEBUG)を使用する。

5.4　システムコール呼び出し時のスタック

(1) C 言語 I/F (func(arg1, arg2, ...))

```
          High Address +---------------+
                       | ...           |
                       | 5th arg       |
                       | 4th arg       |
                       | SPC(24bit)    | saved by I/F call
                       | SCCR(8bit)    |
           Low Address +---------------+
```

er0 = 1st arg
er1 = 2nd arg
er2 = 3rd arg

(2) knl_call_entry 先頭 (trapa #TRAP_SVC 直後)

```
          High Address +---------------+
                       | ...           |
                       | 5th arg       |
                       | 4th arg       |
                       | SPC(24bit)    | saved by I/F call
                       | SCCR(8bit)    |
                       | r0(16bit)     |
                       | SPC(24bit)    | saved by trapa
                 SP => | SCCR(8bit)    |
           Low Address +---------------+
```

r0 = fncd
e0 = 1st arg の上位バイト
スタックに退避された r0 = 1st arg の下位バイト
er1 = 2nd arg
er2 = 3rd arg

(3) bpl l_esvc_function の直後

```
          High Address +---------------+
                       | ...           |
                       | 5th arg       |
                       | 4th arg       |
                       | SPC(24bit)    | saved by I/F call
                       | SCCR(8bit)    |
                       | r0(16bit)     |
                       | SPC(24bit)    | saved by trapa
                       | SCCR(8bit)    |
                       | er4           |
                       | er5           |
            er5, SP => | er6           |
           Low Address +---------------+
```

r0 = fncd
e0 = 1st arg の上位バイト
スタックに退避された r0 = 1st arg の下位バイト
er1 = 2nd arg
er2 = 3rd arg
r4 = fncd

(4) システムコール呼び出し直前

```
          High Address +---------------+
                       | ...           |
                       | 5th arg       |
                       | 4th arg       |
                       | SPC(24bit)    | saved by I/F call
                       | SCCR(8bit)    |
                       | r0(16bit)     |
                       | SPC(24bit)    | saved by trapa
                       | SCCR(8bit)    |
                       | er4           |
                       | er5           |
                 er5 => | er6           |
                       | (5th arg)     |
                       | (4th arg)     |
           Low Address +---------------+
```

er0 = 1st arg
er1 = 2nd arg
er2 = 3rd arg

5.5 拡張 SVC 呼び出し時のスタック

(1) C 言語 I/F (func(arg1, arg2, ...))

```
          High Address +---------------+
                       | ...           |
                       | 5th arg       |
                       | 4th arg       |
                       | SPC(24bit)    | saved by I/F call
                       | SCCR(8bit)    |
           Low Address +---------------+

        er0 = 1st arg
        er1 = 2nd arg
        er2 = 3rd arg
```

(2) knl_call_entry 先頭 (trapa #TRAP_SVC 直後)

```
          High Address +---------------+
                       | ...           |
                       | 5th arg       |
                       | 4th arg       |
                       | SPC(24bit)    | saved by I/F call
                       | SCCR(8bit)    |
                       | 2nd arg       |
                 er1 => | 1st arg       |
                       | 3rd arg       |
                       | SPC(24bit)    | saved by trapa
                  SP => | SCCR(8bit)    |
           Low Address +---------------+

         r0 = fncd
        er1 = pk_para
```

(3) l_esvc_function 先頭

```
          High Address +---------------+
                       | ...           |
                       | 5th arg       |
                       | 4th arg       |
                       | SPC(24bit)    | saved by I/F call
                       | SCCR(8bit)    |
                       | 2nd arg       |
                 er1 => | 1st arg       |
                       | 3rd arg       |
                       | SPC(24bit)    | saved by trapa
                       | SCCR(8bit)    |
                       | er4           |
                       | er5           |
            er5, SP => | er6           |
             Low Address +---------------+

           r0 = fncd
          er1 = pk_para
```

(4) knl_svc_ientry 先頭

```
          High Address +---------------+
                       | ...           |
                       | 5th arg       |
                       | 4th arg       |
                       | 3rd arg       |
                       | 2nd arg       |
                 er0 => | 1st arg       |
                       | 3rd arg       |
                       | SPC(24bit)    | saved by trapa
                       | SCCR(8bit)    |
                       | er4           |
                       | er5           |
            er5, SP => | er6           |
             Low Address +---------------+

          er0 = pk_para
           r1 = fncd

          er4 = ret - addr (saved by I/F call)
```

5.6　割込み発生時のスタック

・ハードウェア割込み発生時のスタック

```
          High Address +---------------+
                       | SPC(24bit)    | saved by Interrupt Controller
                 SP => | SCCR(8bit)    |
           Low Address +---------------+

       er0 = 1st arg
       er1 = 2nd arg
       er2 = 3rd arg
```

・tk_def_int 使用時の割込みハンドラ入り口のスタック
（knl_inthdr_entryN（Nは割込みベクタ番号）から呼出し）

```
          High Address +---------------+
                       | SPC(24bit)    | saved by Interrupt Controller
                       | SCCR(8bit)    |
                       | er0           |
                 SP => | er1           |
           Low Address +---------------+

       er0 = 割込みベクタ番号
```

5.7　タスクの実装依存定義

タスクのハードウェアに依存した実装定義を以下に示す。

(1) タスク生成情報　T_CTSK

独自に追加した情報はなし。

```
typedef struct t_ctsk {
        VP      exinf;          /* 拡張情報 */
        ATR     tskatr;         /* タスク属性 */
        FP      task;           /* タスク起動アドレス */
        PRI     itskpri;        /* タスク起動時優先度 */
        W       stksz;          /* スタックサイズ(バイト) */
        UB      dsname[8];      /* DS オブジェクト名称 */
        VP      bufptr;         /* ユーザーバッファポインタ */
} T_CTSK;
```

(2) タスク属性

実装独自属性はなし。

```
tskatr := (TA_ASM // TA_HLNG)
        | [TA_USERBUF] | [TA_DSNAME]
        | (TA_RNG0 // TA_RNG1 // TA_RNG2 // TA_RNG3)
```

(3) タスクの形式

タスクは次の形式で、TA_HLNG, TA_ASM のどちらを指定しても違いはない。

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

タスク起動時のレジスタの状態は下記のようになる。

```
CCR.I = 0     割込許可
er0 = stacd   タスク起動パラメータ
er1 = exinf   タスク拡張情報
er7(sp)       スタックポインタ
```

その他のレジスタは不定である。
タスクの終了は、tk_ext_tsk() または tk_exd_tsk() を用いなければならない。単に return してもタスクの終了とはならない。return した場合の動作は保証されない。

5.8　タスクレジスタの設定/参照

```
ER tk_set_reg( ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs )
ER tk_get_reg( ID tskid, T_REGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs )
```

タスクレジスタの取得/設定（tk_get_reg/tk_set_reg）の対象となるレジスタは以下のように定義される。

(1) 汎用レジスタ T_REGS

```
typedef struct t_regs {
        VW er[7];       /* 汎用レジスタ er0-er6 */
} T_REGS;
```

DORMANT 状態のタスクに対してレジスタの設定を行ったとき、er0, er1 は tk_sta_tsk() によってタスク起動パラメータ／拡張情報が設定されるため、tk_set_reg() で設定した値は捨てられることになる。

(2) 例外時に保存されるレジスタ T_EIT

```
typedef struct t_eit {
        VP pc;          /* プログラムカウンタ PC */
        VB ccr;         /* コンディションコードレジスタ CCR */
} T_EIT;
```

(3) 制御レジスタ

```
typedef struct t_cregs {
        VP ssp;         /* システムスタックポインタ er7 */
} T_CREGS;
```

6. システムコンフィグレーションデータ

utk_config_depend.h では、μT-Kernel のシステム構成情報や各種資源数、各種制限値などの設定を記述する。なお、各項目の指定可能範囲の最大値は、論理的な最大値であり、実際にはメモリの使用量により制限を受ける。

6.1 utk_config_depend.h の設定値

```
/*
 *      utk_config_depend.h (h8s2212)
 *      System Configuration Definition
 */

/* ROMINFO */
#define SYSTEMAREA_TOP          0x00ffc000
#define SYSTEMAREA_END          0x00ffef00
```

※ μT-Kernel のメモリ管理機能により動的に管理される領域
RAM の最下位アドレスと最上位アドレスを指定

```
/* User definition */
#define RI_USERAREA_TOP         0x00ffefc0
```

※ 本設定は使用しない。

```
#define RI_USERINIT             NULL
```

※ ユーザー初期化/完了プログラム

```
/* Stacks */
#define RI_INTSTACK             0x00ffefc0
```

※ 割込みスタックの初期位置

```
/* SYSCONF */
#define CFN_TIMER_PERIOD        10
```

※ システムタイマの割込み周期(ミリ秒)。各種の時間指定の最小分解能(精度)となる。

```
#define CFN_MAX_TSKID           32
#define CFN_MAX_SEMID           16
#define CFN_MAX_FLGID           16
#define CFN_MAX_MBXID           8
#define CFN_MAX_MTXID           2
#define CFN_MAX_MBFID           8
#define CFN_MAX_PORID           4
#define CFN_MAX_MPLID           2
#define CFN_MAX_MPFID           8
#define CFN_MAX_CYCID           4
#define CFN_MAX_ALMID           8
#define CFN_MAX_SSYID           4
```

※ μT-Kernel の各オブジェクトの最大数。
カーネルが使用するオブジェクトの数も考慮して指定する必要がある。

```
#define CFN_MAX_REGDEV          8
```

※ tk_def_dev() で登録可能な最大デバイス数。物理デバイスの最大数となる。

```
#define CFN_MAX_OPNDEV          16
```

※ tk_opn_dev() でオープン可能な最大数。
デバイスディスクリプタの最大数となる。

```
#define CFN_MAX_REQDEV          16
```

※ tk_rea_dev()、tk_wri_dev()、tk_srea_dev()、tk_swri_dev() で要求可能な最大数。
リクエスト ID の最大数となる。

```
#define CFN_VER_MAKER           0
#define CFN_VER_PRID            0
#define CFN_VER_SPVER           0x6101
#define CFN_VER_PRVER           0x0101
#define CFN_VER_PRNO1           0
#define CFN_VER_PRNO2           0
#define CFN_VER_PRNO3           0
#define CFN_VER_PRNO4           0
```

※バージョン情報 (tk_ref_ver)

```
#define CFN_REALMEMEND          ((VP)0x00ffefc0)
```

※ μT-Kernel 管理領域で利用する RAM の最上位アドレス

```
/*
 * Use non-clear section
 */
#define USE_NOINIT              (1)
```

※ 1 : 初期値を持たない静的変数(BSS 配置)のうち、初期化が必要のない変数は
カーネル初期化処理内で ゼロクリアしない。
ゼロクリアの処理が削減される為、カーネル起動時間が短縮される。
0 : 全ての初期値を持たない静的変数(BSS 配置)をゼロクリアする。

```
/*
 * Use dynamic memory allocation
 */
#define USE_IMALLOC             (1)
```

※ 1 : カーネル内部の 動的メモリ割当て機能を使用する。
0 : カーネル内部の 動的メモリ割当て機能を使用しない。
タスク、メッセージバッファ、固定長／可変長メモリプールのオブジェクト生成時、
TA_USERBUF を指定して、アプリケーションがバッファを指定しなければならない。

```
/*
 * Use program trace function (in debugger support)
 */
#define USE_HOOK_TRACE          (0)
```

※ 1 : デバッガサポート機能のフック機構を使用する。
但し、USE_DBGSPT が 0 になっている場合は、フック機構は使えない。
0 : デバッガサポート機能のフック機構を使用しない。

```
/*
 * Use clean-up sequence
 */
#define USE_CLEANUP             (1)
```

※ 1 : アプリケーション終了後に、カーネルのクリーンアップ処理を行う。
0 : アプリケーション終了後に、カーネルのクリーンアップ処理を行わない。
usermain 関数から戻らないシステムは、本フラグをオフにすることで ROM 消費量が減る。

```
/*
 * Use full interrupt vector
 */
#define USE_FULL_VECTOR         (1)
```

※ 1 : 割込みの初期処理（一部のレジスタの退避や割込み番号の設定など）を
全ての割込みベクタに対して用意する。
0 : 定義した割込みに対してのみ初期処理を用意する。ROM を節約できる。

```
/*
 * Use high level programming language support routine
 */
#define USE_HLL_INTHDR          (1)
```

※ 1 : 割込みで高級言語対応ルーチンを使用する。
0 : 割込みで高級言語対応ルーチンを使用しない。
ジャンプテーブルなどが外れ、ROM/RAM 消費量が減る。

```
/*
 * Use dynamic interrupt handler definition
 */
#define USE_DYNAMIC_INTHDR      (1)
```

※ 1 : 割込みハンドラを tk_def_int で動的に変更する。
0 : 割込みハンドラを tk_def_int で動的に変更しない。
ジャンプテーブルが外れ、RAM 消費量が減る。

6.2 makerules

以下の引数を指定して make を行うことによってモード選択を行う。

・ mode（ コンパイルモード ）

指定なし ： リリースモード
debug ： デバックモード
例: $ make mode=debug

・trap（トラップモード ）

指定なし ： トラップ未使用
on ： システムコール、ディスパッチ時にトラップ実行
例: $ make mode=debug trap=on