μT-Kernel 実装仕様書

FM3 (Cortex-M3)版

Version 1.01.03

2014年7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。

対象とするボードは T-Engine Appliance / FM3 (Cortex-M3) である。
対象とするμT-Kernel ソースコードのバージョンは 1.01.03 版 である。
準拠するμT-Kernel 仕様書のバージョンは 1.01.02 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 : MB9AF312K (ARM Cortex-M3 Core)
Spansion Inc.
ROM : 128 KB (Main Flash) + 32 KB (Work Flash)
RAM : 16 KB (SRAM)

1.2 保護レベルと動作モード

保護レベルは、CPU の動作モードに以下のように対応します。

保護レベル	動作モード
3	（保護レベル 0 として扱う）
2	（保護レベル 0 として扱う）
1	（保護レベル 0 として扱う）
0	SVC:スーパーバイザモード

・ユーザーモード(USR) と システムモード(SYS) の CPU 動作モードは使用しません。
・どの保護レベルが指定されても保護レベル 0 として扱われます。

1.3 Thumb-2 命令セットの使用

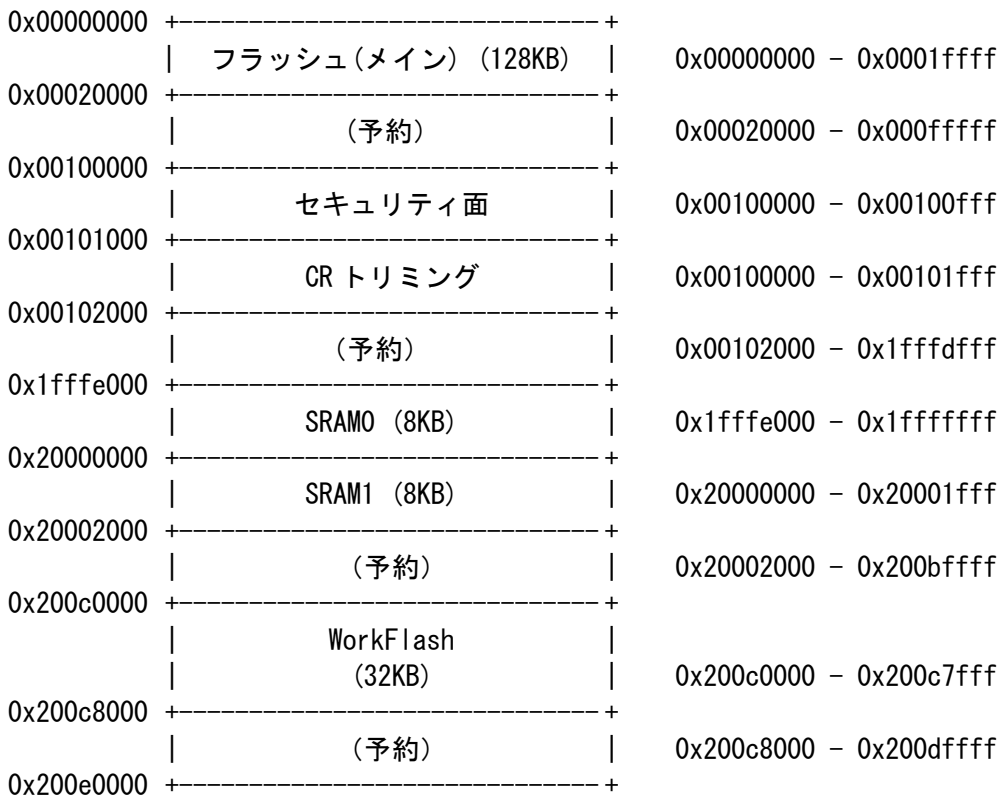
ターゲットボード SK-FM3-48PMC-USBSTICK の CPU である MB9AF312K(ARM Cortex-M3 Core)では Thumb-2 命令セットのみをサポートするため、「Thumb-2 Instruction Set」に従ってプログラミングする必要があります。

タスクやハンドラを Thumb 命令セット（Thumb-2 を含む）を使用して実装する場合、そのアドレスの最下位ビットに 1 を指定します（0 が指定された場合は ARM 命令セットとみなされます)。 これは通常リンカが自動的に行うため、プログラマは特に意識する必要はありません。

2. メモリマップ

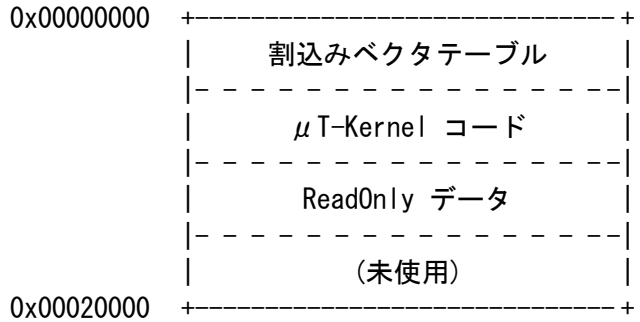
2.1 全体メモリマップ

システム全体のメモリマップを以下に示します



2.2 ROM 領域メモリマップ

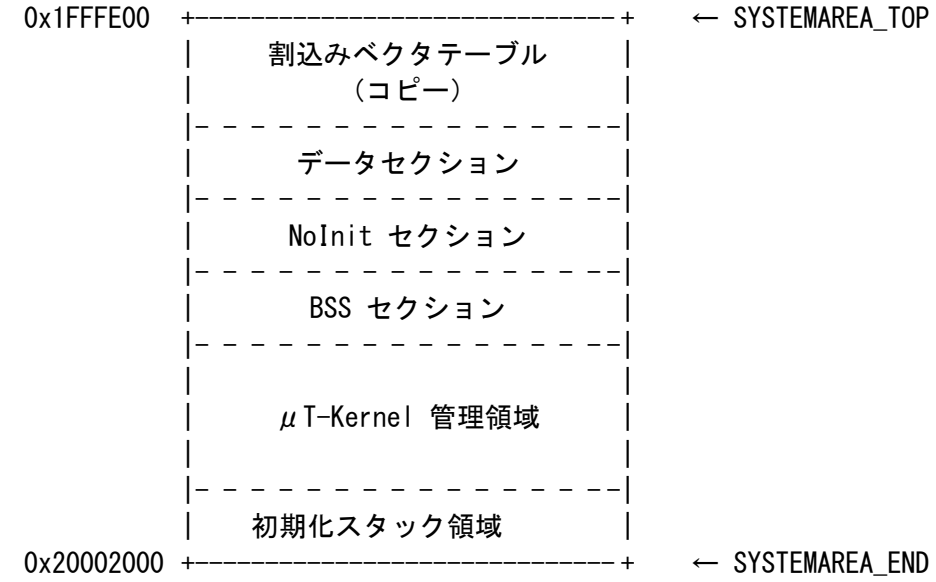
内蔵 FlashROM は 128KB の領域が実装されています。内蔵 FlashROM 領域のメモリマップを以下に示します。



内蔵 FlashROM の下位アドレスにμT-Kernel コードを配置します。

2.3 内蔵 SRAM 領域メモリマップ

内蔵 SRAM は 16KB の領域が実装されています。内蔵 SRAM 領域のメモリマップを以下に示します。



NoInit： ゼロ初期化されない BSS セクション

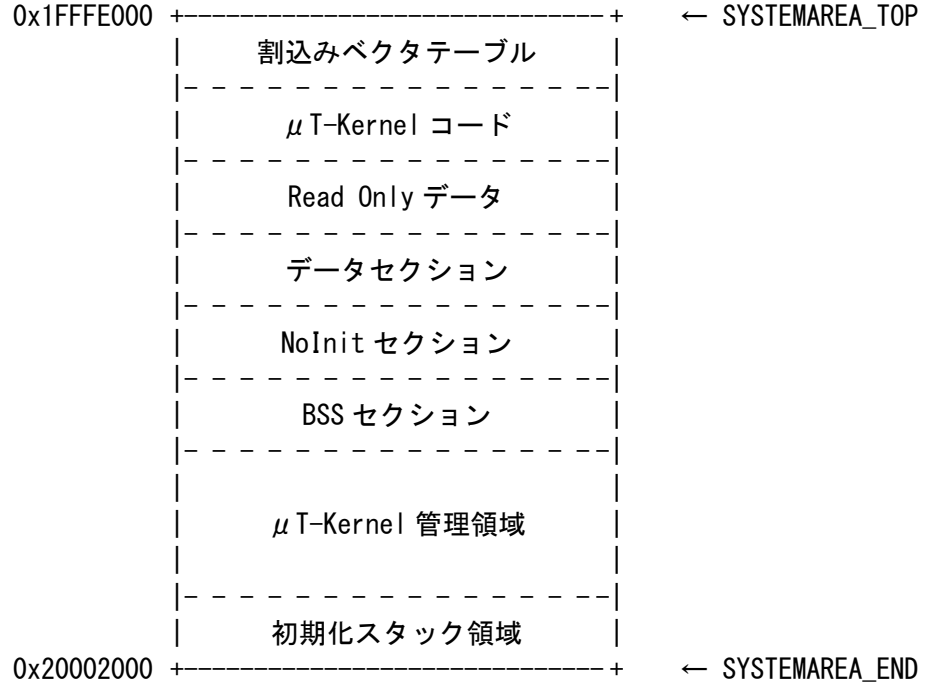
内蔵 SRAM の下位アドレスに割込みベクタテーブル用の領域が確保され、システム起動時に Flash ROM からベクタテーブルがコピーされます。

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り、原則として BSS セクションの終端から初期化スタック領域までの間の領域となります。

通常は、空いているメモリ領域が全て μT-Kernel 管理領域に割り当てられますが、設定により変更することが可能です。

2.4 内蔵 SRAM 領域メモリマップ（μT-Kernel コードを RAM に配置する場合）

RAM 版ロードモジュール(kernel-ram.sys)を使用した場合、μT-Kernel コードが RAM に配置され、Flash ROM は使用されません。この場合、以下のようなメモリマップとなります。



NoInit： ゼロ初期化されない BSS セクション

2.5 スタック

本システムには以下の 3 種類のスタックがあります。

(1) 初期化スタック

ハードウェアがリセットされてからμT-Kernel の初期タスクが起動するまでの間に使用されるスタックです。

初期化スタックのサイズは utk_config_depend.h(EXC_STACK_SIZE)で設定できます。

(2) 一時スタック

タスクディスパッチ処理の間に使用されるスタックです。

一時スタックのサイズは utk_config_depend.h(TMP_STACK_SIZE)で設定できます。

(3) システムスタック

μT-Kernel が管理するスタックで、タスク毎に 1 本ずつ存在します。

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ありません。

また、本システムでは割込みスタックが分離されていないため、割込みハンドラもタスクのスタックを使用します。

3. 割込みおよび例外

3.1 例外番号と IRQ 番号

MB9AF312K ではシステム例外として例外番号 1～15 が、IRQ(外部割込み)用に例外番号 16～63 があります。

本システムで使用できる IRQ 番号(tk_def_int の引数 dintno)は、0～47 です。(IRQ 番号に 16 を加算した値が例外番号となります)

IRQ 番号のうち、46、47 はμT-Kernel が使用します。その他の IRQ 番号は、ハードウェア(MB9AF312K)の割込みコントローラ(NVIC)が管理する割込み要因用に予約されています。詳細は「FM3 ファミリペリプェラルマニュアル」を参照してください。

3.2 ソフトウェア割込みの割当て

μT-Kernel のソフトウェア割込みが使用する IRQ 番号を以下に示します。

- IRQ 46　　ソフトウェアトリガ割込み
- IRQ 47　　強制ディスパッチ

3.3 例外・割込みハンドラ

本ハードウェアのコアである Cortex-M3 プロセッサは、割込みの入り口で自動的にスタックにレジスタ R0～R3、R12、LR、PSR、PC をプッシュし、割込みの出口でそれらをポップする仕組みとすることで、割込みハンドラを C 言語の関数で記述できる仕様になっています。

具体的には、例外が発生するとプロセッサにより以下の処理が行われます。

(1) R0～R3、IP(R12)、LR、PC、xPSR の各レジスタの内容をスタックにプッシュ
(2) ベクタテーブルから例外・割込みハンドラの開始アドレスを取得
(3) SP、PSR、LR、PC の更新
  - PSR の下位 9 ビット（IPSR）に、発生した例外の番号が設定されます
  - LR は、割込み復帰操作のための特殊な値に更新されます

ハンドラを終了しリターンする時は、EXC_RETURN(asm_depend.h)を使用します。リターン時にプロセッサにより(1)に示した 8 種類のレジスタの内容がレジスタからポップされ、SP も例外・割込み発生時の値に戻されます。

なお、(1)に示した 8 種類以外のレジスタの値はスタックに退避されないため、各ハンドラで必要に応じて保存する必要があります。ただし、この仕様は C 言語の呼出規則に従っていますので、割込みハンドラを C 言語で記述する場合は特に意識する必要はありません。

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 が起動します。

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します。

[ROM 版]

startup_rom.S

(1) 初期化処理起動 [Reset_Handler]
(2) システムクロック設定 [init_clock_control]
(3) 割込みベクタを ROM から RAM にコピー [rom_init ～ vector_done]
(4) 割込みベクタのアドレスを(3)でコピーされた RAM 上のアドレスに設定
(5) data セクションの初期化

icrt0.S

(1) bss セクションの初期化
(2) μT-Kernel 管理領域の再計算
(3) シリアル I/O の初期化 [sio_init]
(4) 割込み優先度の設定
(5) 強制ディスパッチ割込みの許可
(6) ソフトウェアトリガ割込みの許可
(7) main 関数(sysinit_main.c) 呼出し

[RAM 版]

startup_ram.S

(1) 初期化処理起動 [Reset_Handler]
(2) システムクロック設定 [init_clock_control]
(3) 割込みベクタのアドレスを設定

icrt0.S

～～ 以下 ROM 版と共通 ～～

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す。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ます。

```
INT     userinit( INT ac, UB **av )

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

戻値    1 以上          usermain() を起動
        0 以下          システム終了
```

システム起動時に ac = 0 で呼出され、システム終了時に ac = -1 で呼出されます。終了時の呼出では戻値は無視されます。処理の概略は次のようになります。

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

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ます。タスク優先度は(CFN_MAX_PRI-2)です。

5. μT-Kernel 実装定義

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します。

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します。

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

例外番号 11	SVC 割込み
例外番号 14	PendSV 割込み
例外番号 15	System Tick Timer (SisTick) 割込み
例外番号 62 (IRQ 46)	ソフトウェアトリガ割込み
例外番号 63 (IRQ 47)	強制ディスパッチ

5.3 システムコールのインターフェース

システムコールの呼出し側は、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ます（カーネルの構築時にオプションとして選択）。

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しますが、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 SVC 命令で呼び出しても構いません。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ります。(但し、SVC 割込み発生時に自動的に保存されるものを除きます)

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ります。

- R12 レジスタに機能コードを設定して SVC 命令（SVC x）により呼び出します。
- 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 となります。
- x には以下のマクロ値を指定します。

SVC_SYSCALL	システムコール（tk_xxx_yyy）
SVC_EXTENDED_SVC	拡張 SVC
SVC_DEBUG_SUPPORT	デバッガサポート機能（td_xxx_yyy）

本ハードウェアのコアである Cortex-M3 プロセッサは、割込み（例外）の入り口で自動的にスタックにレジスタ R0～R3、R12、LR、PSR、PC をプッシュし、割込みの出口でそれらをポップする仕組みとすることで、割込みハンドラを C 言語の関数で記述できる仕様になっているため、SVC 割込み発生時に上記のレジスタがスタックに保存されます。

(1) システムコールのインターフェースライブラリ

システムコールは、第 4 引数まではレジスタに設定し、第 5 引数以降はスタックに積んで SVC 命令により呼び出します。レジスタは以下のように使用されます。

R12=ip 機能コード（<0)
R0 第 1 引数
R1 第 2 引数
R2 第 3 引数
R3 第 4 引数
R4 第 5 引数以降が格納されたスタックへのポインタ

R0 戻値

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します。

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

引数は 0～5 個の整数またはポインタで、C 言語の関数の引数渡しと同じ形式にしてください。

```
/* r0 = p1
 * r1 = p2
 * r2 = p3
 * r3 = p4
 *         +---------------+
 *   sp -> | p5            |
 *         +---------------+
 */
Csym(tk_xxx_yyy):
        stmfd   sp!, {r4}       /* r4 保存 */
        add     r4, sp, #4      /* r4 = スタック上のパラメータの位置 */
        stmfd   sp!, {lr}       /* lr 保存 */
        ldr     ip, =機能コード
#if USE_TRAP
        swi     SVC_SYSCALL
#else
        bl      Csym(knl_call_entry)
#endif
        ldmfd   sp!, {lr}       /* lr 復帰 */
        ldmfd   sp!, {r4}       /* r4 復帰 */
        bx      lr
```

(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 R0 レジスタに設定して SVC 命令により呼び出します。パケットは通常スタックに作成しますが、他の場所でもかまいません。引数はパケット化するため、数や型に制限はありません。レジスタは以下のように使用します。

R12=ip 機能コード（≧0)
R0 引数パケット

R0 戻値

拡張 SVC のインターフェースの実装例を以下に示します。

```
INT zxxx_yyy( .... )
```

引数は呼出側でパケット化し、パケットの先頭アドレスを R0 レジスタに設定します。

```
callsvc
        stmfd   sp!, {r1-r3}    /* レジスタ上の引数をスタックに積みパケット化 */
        mov     ip, r0          /* ip = R0 = 機能コード */
        mov     r0, sp          /* R0 = 引数パケットのアドレス */
        stmfd   sp!, {lr}       /* lr 保存 */

#if USE_TRAP
        SVC     SVC_EXTENDED_SVC
#else
        bl      knl_call_entry
#endif

        ldmfd   sp!, {lr}       /* lr 復帰 */
        add     sp, sp, #3*4    /* スタックに積んだ引数を捨てる */
        bx      lr
```

5.4 割込みハンドラ

割込みハンドラのハードウェアに依存した実装定義を以下に示します。

・ 割込みハンドラ定義情報 : T_DINT

```
typedef struct t_dint {
        ATR     intatr;         /* 割込みハンドラ属性 */
        FP      inthdr;         /* 割込みハンドラアドレス */
} T_DINT;
```

・ 割込み定義番号 : dintno

dintno は 0～47 の範囲で指定可能（但し 46, 47 は使用しないでください）

本ハードウェアのコアである Cortex-M3 プロセッサは、割込みの入口で自動的にスタックにレジスタ R0～R3、R12、LR、PSR、PC をプッシュし、割込みの出口でそれらをポップする仕組みとすることで、割込みハンドラを C 言語の関数で記述できる仕様になっています。

本システムではこれに従い、以下の実装としています。

・割込みハンドラは C 言語の関数で記述
・ハンドラ属性は TA_HLNG を指定

割込みハンドラ属性は TA_HLNG であるため、例外/割込みベクタテーブルにはμT-Kernel 内の高級言語対応ルーチンのアドレスが設定され、高級言語対応ルーチンから設定された割込みハンドラが呼び出されます。

割込みハンドラの定義は以下のようになります。

```
void inthdr(UINT dintno)

dintno  発生した例外/割込みベクタ番号
        デフォルトハンドラの場合、デフォルトハンドラのベクタ番号ではなく、
        発生した例外/割込みのベクタ番号となる
```

割込みハンドラに入ったときの CPU の状態は以下のようになります。

```
PRIMASK = 0             割込許可
xPSR.T  = 1             Thumb モード
                        （ハンドラ起動アドレスの最下位ビットが 1）
```

割込みハンドラからの復帰は以下の 2 つのステップを経て行われます。

(1) ハンドラ関数から高級言語対応ルーチンへ return
(2) 高級言語対応ルーチンの終端で EXC_RETURN を実行

EXC_RETURN を実行すると、割込みコントローラ(NVIC)レジスタの例外アクティブビットがクリアされます。(割込みハンドラ内で NVIC の割込みのクリアを行う必要はありません)

5.5 タイムイベントハンドラ

ハンドラ属性に TA_ASM 属性を指定した場合も、TA_HLNG 属性の場合と同様に高級言語対応ルーチンを経由して呼び出されます。したがって、TA_ASM 属性の場合もハンドラへ渡されるパラメータ (exinf) は C 言語の規則にしたがって R0 レジスタに渡されます。また、C 言語の呼出規則に従ってレジスタを保存する必要があります。

5.6 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します。

(1) タスク生成情報　T_CTSK

独自に追加した情報はありません。

```
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

実装独自属性はありません。
タスク属性 (tskatr) は TA_HLNG のみ指定可能です。(TA_ASM は対応していません)

```
tskatr := 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記述します。

```
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ります。

```
PRIMASK = 0     割込許可
xPSR.T  = 1     Thumb モード (タスク起動アドレスの最下位ビットが 1)

R0 = stacd      タスク起動パラメータ
R1 = exinf      タスク拡張情報
R13(sp)         スタックポインタ
```

その他のレジスタは不定です。
タスクの終了は、必ず tk_ext_tsk() または tk_exd_tsk() を用いてください。単に return してもタスクの終了とはなりません。return した場合の動作は保証されません。

5.7 タスクレジスタの取得/設定

```
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ます。

(1) 汎用レジスタ T_REGS

```
typedef struct t_regs {
        VW r[13];       /* 汎用レジスタ R0～R12 */
        VP lr;          /* リンクレジスタ R14 */
} 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、R0, R1 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ります。

(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 R15 */
        UW cpsr;        /* プログラムステータスレジスタ */
        UW taskmode;    /* タスクモードフラグ */
} T_EIT;
```

CPSR は、フラグフィールド(ビット 31～24)以外は変更できません。他のフィールド(ビット 23～0)への設定は無視されます。
taskmode は、システム共有情報にあるタスクモードフラグと同じです。メモリのアクセス権情報を保持するレジスタとして扱われます。

(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 R13 */
} T_CREGS;
```

5.8 システムコール・拡張 SVC 呼び出し元情報

システムコール・拡張 SVC フックルーチンの実装依存定義を以下に示します。

フックルーチン定義情報 TD_CALINF

```
typedef struct td_calinf {
        VP      ssp;    /* システムスタックポインタ */
        VP      r11;    /* 呼出時のフレームポインタ */
} TD_CALINF;
```

フックルーチンに入ったときのシステムスタックは以下の状態になっています。

```
        +---------------+
  sp -> |taskmode       |
        |R0 ～ R7       |
        |機能コード     |
        |lr(2)          | knl_call_entry(cpu_support.S)からの戻りアドレス
        |lr(1)          | インタフェースライブラリからの戻りアドレス
        |R4             |
        +---------------+
```

システムコール・拡張 SVC からの戻り番地は lr(2) となります。しかし、通常はインターフェースライブラリを利用して呼び出すため、lr(2) はインターフェースライブラリの番地を指しています。
インターフェースライブラリからの戻り番地は lr(1) となります。

5.9 割込みコントローラ制御

割込みコントローラ制御は、ハードウェアへの依存が強いため、μT-Kernel 仕様書では規定されていません。SK-FM3-48PMC-USBSTICK では以下の通り実装しています。

(1) 割込み定義番号（IRQ 番号）

割込み定義番号（0～47）は以下のものを使用します。

- 46　ソフトウェアトリガ割込み
- 47　強制ディスパッチ

(2) 割込許可

```
void    EnableInt( INTVEC intvec );
```

intvec で指定した割込み定義番号の割込みを許可します。

(3) 割込禁止

```
void    DisableInt( INTVEC intvec );
```

intvec で指定した割込み定義番号の割込みを禁止します。

(4) 割込要求の有無の確認

```
BOOL    CheckInt( INTVEC intvec );
```

intvec の割込要求があるか調べ、割込要求があれば TRUE (0 以外) を返します。

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します。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ます。

6.1 utk_config_depend.h の設定値

```
/*
 *      utk_config_depend.h (FM3)
 *      System Configuration Definition
 */

/* RAMINFO */
#define SYSTEMAREA_TOP          0x1FFFE000
#define SYSTEMAREA_END          0x20002000
```

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

```
/* User definition */
#define RI_USERAREA_TOP         0x1FFFE000
```

※ 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

```
/* 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)となります。

```
#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ります。

```
#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ります。

```
#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ります。

```
#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 の最大数となります。

```
#define CFN_VER_MAKER           0x011C
#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)0x20002000)
```

※ μT-Kernel 管理領域で利用する RAM の最上位アドレス

```
/*
 * Initial task priority
 */
#define INIT_TASK_PRI           (MAX_PRI-2)
```

※ 初期タスクの優先度

```
/*
 * Use zero-clear bss section
 */
#define USE_NOINIT              (0)
```

※ 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ません。
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ます。
0 :全ての 初期値を持たない静的変数(BSS 配置) をゼロクリアします。

```
/*
 * Stack size for each mode
 */
#define EXC_STACK_SIZE          0x200
#define TMP_STACK_SIZE          0x80
#define USR_STACK_SIZE          0               /* not used */
```

※ 各例外モード用のスタックサイズ(バイト数)

※ 本システムでは初期化スタックのサイズとして使用します。

```
#define EXCEPTION_STACK_TOP     SYSTEMAREA_END
```

※ スタック領域の初期位置
これらの例外モード用スタックは、通常はハンドラの入り口と出口においてのみ使用するため、大きな容量は必要としません

```
#define TMP_STACK_TOP           (EXCEPTION_STACK_TOP - EXC_STACK_SIZE)
```

※ テンポラリスタックの位置

```
#define APPLICATION_STACK_TOP   (TMP_STACK_TOP - TMP_STACK_SIZE)
```

※ アプリケーション用スタック領域の初期位置。
本システムでは利用しません。

```
/*
 * 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します。
0 : カーネル内部の 動的メモリ割当て機能を使用しません。
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する必要があります。

```
/*
 * 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します。
但し、USE_DBGSPT が 0 になっている場合は、フック機構は使えません。
0 : デバッガサポート機能のフック機構を使用しません。

```
/*
 * 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います。
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いません。
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 消費量を低減できます。

```
/*
 * 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します。(変更不可)

6.2 sysinfo_depend.h の設定値

```
#define N_INTVEC                48
```

※割込み定義番号の個数を指定します。

7. 制限事項

本バージョンの UTK の制限事項、注意事項を以下に示します。

7.1 可変長メモリブロック獲得のサイズ指定に関する制限事項

tk_get_mpl（可変長メモリブロック獲得）のパラメータ blksz（メモリブロックサイズ）に以下の値を指定して呼び出した場合、tk_get_mpl は正常終了しますが、メモリは正常に獲得されていません。

blksz = 0x7FFFFFF9 ～ 0x7FFFFFFF

上記の値を blksz に指定して tk_get_mpl を呼び出さないでください。

7.2 周期ハンドラの起動に関する注意事項

tk_cre_cyc（周期ハンドラの生成）のパラメータ cycphs（周期起動位相）に 0 を指定した場合、本システムコール処理中に 1 回目の周期ハンドラを起動(実行)します。この場合、周期ハンドラ内での割込み禁止などの状態は、2 回目以降の(通常の)周期ハンドラ起動時とは異なります。

※本設定によるμT-Kernel の挙動は、T-Kernel の挙動と同じです。

【 参考 】 (T-Kernel 2.0 仕様書 tk_cre_cyc の補足事項から抜粋)

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。

7.3 未サポートの機能

下記の機能はサポートしていません。

・TA_ASM 属性