# リアルタイムカーネル勉強会　資料

## プロセッサ依存部１
## ・コンテキスト
## ・ディスパッチャ

ヤマハ株式会社　安立直之
adachi-n@emi.yamaha.co.jp

**始めに**

本書はリアルタイムカーネルのプロセッサ依存部のうち、コンテキスト、ディスパッチャ、及びそれらに関連するプロセッサ依存部の定義について解説するものである。

## 1.　コンテキスト

### 1.1.　定義

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言うためには、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。但し、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。

### 1.2.　タスクコンテキストブロックのデータ型（CTXB）

ターゲット依存のタスクコンテキストを保存するために TCB 中に持つことが必要なデータ構造の型。SH3 用定義は cpu_config.h　にて以下のように定義されている。

```
/*
 * タスクコンテキストブロックの定義
 */
typedef struct task_context_block {
        VP    sp;             /* スタックポインタ */
        FP    pc;             /* プログラムカウンタ */
} CTXB;
```

### 1.3.　タスクコンテキスト設定処理

ターゲット依存のタスクコンテキストを設定するために、create_context と activate_context の二つの関数を用意する。二つの関数を呼び出すことで、タスクのコンテキスト（具体的にはタスクコンテキス

トブロックの内容とスタック領域）をタスクが起動できる状態に設定する。二つの関数は呼ばれるタイミングが異なるだけで明確な役割分担はなく、どのような処理がどちらの関数で行われなければなっらないという制約はない。
これらの関数の宣言及びマクロの定義は、cpu_context.h に含める。以下、SH3 用の定義である。

```
/*
 *  タスクコンテキストブロックの初期化
 *
 *  タスクが休止状態に移行する時（タスクの生成時，タスクの終了時）に呼
 *  ばれる．基本的には，タスクコンテキストブロックをタスクが起動できる
 *  状態に設定する処理を，create_context と activate_context で行えば
 *  よい．多くの処理はどちらの関数で行ってもよい．
 */
Inline void
create_context(TCB *tcb)
{
}

/*
 *  タスクの起動準備
 *
 *  タスクが休止状態から実行できる状態に移行する時に呼ばれる．
 *
 */
extern void activate_r(void);

Inline void
activate_context(TCB *tcb)
{
      VW    *sp;

      sp = (VW *)(((VB *) tcb->tinib->stk) + tcb->tinib->stksz);
      *--sp = (VW)(tcb->tinib->exinf);
      *--sp = (VW) ext_tsk;
      *--sp = (VW)(tcb->tinib->task);
      tcb->tskctxb.sp = sp;
      tcb->tskctxb.pc = activate_r;
}


/*
 *  ext_tsk がスタック上に確保するダミー領域のサイズ
 *  Not yet!
 */
#define     ACTIVATED_STACK_SIZE       (sizeof(VW) * 3)

#endif /* _CPU_CONTEXT_H_ */
```

### 1.3.1　void create_context( TCB *tcb)

タスクが休止状態に移行するときに呼ばれる。具体的にはタスクの生成時（JSP カーネルではCRE_TSKでタスクを生成するため、タスク管理モジュールの初期化(void task_initialize())）とタスクの終了時（ext_tsk,ter_tsk）に呼ばれる。

### 1.3.2　void activate_context(TCB *tcb)

タスクが実行できる状態に移行する時に呼ばれる。具体的には act_tsk でタスクを起動するとき、タスクの終了時（ext_tsk,ter_tsk）に起動要求のキューイングにより再起動するとき、TA_ACT 属性を指定してタスクを生成した時（タスク管理モジュールの初期化）に呼ばれる。

### 1.3.3 ACTIVATE_STACK_SIZE（オプション）

ext_tsk がスタック上に確保するダミー領域のサイズを定義するためのマクロ。ダミー領域が必要ない場合は定義する必要はない。

ext_tsk は、自タスクを終了させた後、自タスクに対して create_context を呼ぶ。また、タスクの起動要求がキューイングされていた場合には、自タスクに対して activate_context も呼ぶ。create_context と activate_context は、対象タスクのスタック領域を書き換える場合があるが、これが ext_tsk（およびそこから呼ばれる関数）が使用しているスタック領域と重なった場合、自分の使用しているスタック領域を自分で破壊する結果になる。

ACTIVATE_STACK_SIZE を、create_context と activate_context が書き換えるスタック領域のサイズ（厳密には、スタックの底から何バイトめまでを書き換えるか）にマクロ定義しておくと、ext_tsk 内でスタック上に定義したサイズのダミー領域を確保し、自分の使用しているスタック領域を破壊するのを防ぐ。

## 2. システム状態参照

### 2.1. BOOL sense_context(void)

現在の実行コンテキストが、タスクコンテキストの場合は FALSE、非タスクコンテキストの場合は TRUE を返す関数。

```
Inline BOOL
sense_context()
{
	UW	nest;
	Asm("stc r7_bank,%0":"=r"(nest));

	return(nest > 0);
}
```

### 2.2. BOOL sense_lock(void)

現在のシステム状態が、CPU ロック状態の場合は TRUE、CPU ロック解除状態の時は FALSE を返す関数。

```
/*
 *	現在の割込みマスクの読出し
 */
Inline UW
current_intmask()
{
	return(current_sr() & 0x000000f0);
}

Inline BOOL
sense_lock()
{
	return(current_intmask() == MAX_IPM << 4);
}
```

### 2.3. BOOL t_sense_lock(void)

タスクコンテキストにおいて、現在のシステム状態が、CPU ロック状態の場合は TRUE、CPU ロック解除状態の時は FALSE を返す関数。この関数が、非タスクコンテキストから呼ばれることはない。

SH3 用コードでは、sense_lock でマクロが定義されている。

### 2.4. BOOL i_sense_lock(void)

非タスクコンテキストにおいて、現在のシステム状態が、CPU ロック状態の場合は TRUE、CPU ロック解除状態の時は FALSE を返す関数。この関数が、タスクコンテキストから呼ばれることはない。
SH3 用コードでは、sense_lock でマクロが定義されている。

## 3. CPU ロックと解除

### 3.1. BOOL t_lock_cpu(void)

タスクコンテキストにおいて、CPU ロック解除状態から、CPU ロック状態に遷移させる関数。この関数が、CPU ロック状態で呼ばれることはない。また、非タスクコンテキストから呼ばれることもない。

```
#cpu_insn.h
Inline void
disint(void)
{
      set_sr((current_sr() & ~0x000000f0) |    MAX_IPM << 4 );
}

#cpu_config.h
Inline void
t_lock_cpu()
{
      disint();
}
```

### 3.2. BOOL t_unlock_cpu(void)

タスクコンテキストにおいて、CPU ロック状態から、CPU ロック解除状態に遷移させる関数。この関数が、CPU ロック解除状態で呼ばれることはない。また、非タスクコンテキストから呼ばれることもない。

```
#cpu_insn.h
Inline void
enaint()
{
	set_sr(current_sr() & ~0x000000f0);
}

#cpu_config.h
/*
 *  割込みマスクの設定
 */
Inline void
set_intmask(UW intmask)
{
	set_sr((current_sr() & ~0x000000f0) | intmask);
}

Inline void
t_unlock_cpu()
{
#ifdef SUPPORT_CHG_IPM
	/*
	 *  t_unlock_cpu が呼び出されるのは CPU ロック状態のみであるた
	 *  め，処理の途中で task_intmask が書き換わることはない．
	 */
	set_intmask(task_intmask);
#else /* SUPPORT_CHG_IPM */
	enaint();
#endif /* SUPPORT_CHG_IPM */
}
```

### 3.3. BOOL i_lock_cpu(void)

非タスクコンテキストにおいて、CPU ロック解除状態から、CPU ロック状態に遷移させる関数。この関数が、CPU ロック状態で呼ばれることはない。また、タスクコンテキストから呼ばれることもない。

```
extern UW	int_intmask;	/* 非タスクコンテキストでの割込みマスク */

Inline void
i_lock_cpu()
{
	UW	intmask;

	/*
	 *  一時変数 intmask を使っているのは，current_intmask()を呼ん
	 *  だ直後に割込みが発生し，起動された割込みハンドラ内で
	 *  int_intmask が変更される可能性があるためである．
	 */
	intmask = current_intmask();
	disint();
	int_intmask = intmask;
}
```

### 3.4. BOOL i_unlock_cpu(void)

非タスクコンテキストにおいて、CPU ロック状態から、CPU ロック解除状態に遷移させる関数。この関数が、CPU ロック解除状態で呼ばれることはない。また、タスクコンテキストから呼ばれることもない。

```
Inline void
i_unlock_cpu()
{
      set_intmask(int_intmask);
}
```

## 4. タスクディスパッチャ

### 4.1 void dispatch(void)

タスクディスパッチャを明示的に呼ぶための関数。タスクコンテキストから呼ばれたサービスコール処理から、CPU ロック状態で呼ばれる。
この関数が呼ばれると，関数を呼んだタスクのコンテキストを保存し、実行できるタスクの中で最高優先順位のタスク（schedtsk）のコンテキストを復帰して実行状態とする。実行できるタスクがない場合（schedtsk が NULL の場合）には、割込みを許可して、実行できるタスクができるまで待つ。ここで、実行できるタスクができるのを待つ間に起動された割込みハンドラの出口で、ディスパッチャが呼ばれないように対策することが必要である。具体的には、実行できるタスクができるのを待つ間、一時的に非タスクコンテキストに切り換えるか、ディスパッチ禁止状態にする。

新たに実行状態になったタスクが、タスク例外処理ルーチンの起動条件を満たしていれば、タスク例外処理ルーチンを起動する。また、この関数を呼び出したタスクが次に実行状態になった時、タスク例外処理ルーチンの起動条件を満たしていれば、タスク例外処理ルーチンの起動を行う。タスク例外処理ルーチンの起動には、ターゲット独立部が提供する calltex 関数を用いることができる。

### 4.2 void exit_and_dispatch(void)

現在実行中のコンテキストを捨て、ディスパッチャを呼び出すための関数。タスクコンテキストから呼ばれたサービスコール（具体的には、ext_tsk）処理またはカーネルの初期化処理から、CPU ロック状態で呼ばれる。

この関数が呼ばれると、関数を呼んだタスクのコンテキストを保存せず、実行できるタスクの中で最高優先順位のタスク（schedtsk）のコンテキストを復帰して実行状態とする。実行できるタスクがない場合（schedtsk が NULL の場合）の処理は、dispatch と同様である。

新たに実行状態になったタスクが、タスク例外処理ルーチンの起動条件を満たしていれば、タスク例外処理ルーチンを起動する。

この関数は、カーネルの初期化処理からも呼ばれるために、非タスクコンテキストからも呼ばれても正しく処理できることが必要である。なお、この関数からはリターンしない。

### 4.3 SH3 用ディスパッチャ

以下、SH3 用のディスパッチャのソースコードである。
_dispatch においてはコンテキストの保存（具体的には pr、r8～r15 レジスタ及びタスクスタック、戻

り番地=dispatch_r の TCB への保存）を行い、分岐先の dispatcher_1 にてレディキュー中の最高優先順位のタスクへの切替を行っている。このとき schedtsk が存在していなければ、disptcher_2 にて割込待ち状態に移行する。
復帰後は dispatch_r にてコンテキストの復帰（具体的には pr、r8～r15 レジスタ）タスク例外ルーチンの呼出を行っている。

```
/*
 *   タスクディスパッチャ
 *
 *   _dispatch は、r7_bank0 = 0,割込み禁止状態で呼び出さなければならな
 *   い．_exit_and_dispatch も，r7_bank0 = 0・割込み禁止状態で呼び出す
 *   のが原則であるが，カーネル起動時に対応するため，r7_bank = 1で呼び
 *   出した場合にも対応している．
 */

      .text
      .align 2
      .global _dispatch
_dispatch:
      sts.l   pr,@-r15              /* pr,r8～r15 をスタックに保存                */
      mov.l   r14,@-r15             /* r0～r7は呼び出し元で保存しているため   */
      mov.l   r13,@-r15             /*  保存する必要が無い                        */
      mov.l   r12,@-r15
      mov.l   r11,@-r15
      mov.l   r10,@-r15
      mov.l   r9, @-r15
      mov.l   r8, @-r15
      mov.l   _runtsk_dis,r2     /* r0 <- runtsk                                  */
      mov.l   @r2,r0
      mov.l   r15,@(TCB_sp,r0) /*  タスクスタックをTCBに保存  60以下ならOK*/
      mov.l   dispatch_r_k,r1    /*  実行再開番地を保存                          */
      mov.l   r1,@(TCB_pc,r0)    /*  実行再開番地をTCBに保存    60以下ならOK */
      bra     dispatcher_1
      nop

dispatch_r:
      mov.l   @r15+,r8              /*  レジスタを復帰                           */
      mov.l   @r15+,r9
      mov.l   @r15+,r10
      mov.l   @r15+,r11
      mov.l   @r15+,r12
      mov.l   @r15+,r13
      mov.l   @r15+,r14
      lds.l   @r15+,pr
      mov.l   _calltex_dis,r1 /*  タスク例外ルーチンの呼び出し               */
      jmp     @r1
      nop
```

```
	.global _exit_and_dispatch
_exit_and_dispatch:
	mov.l	_mask_md_ipm_dis,r9	/* 割り込み禁止 */
	ldc	r9,sr
	xor	r1,r1			/* r7_bank0を0クリア */
	ldc	r1,r7_bank
dispatcher_1:
	/*
	 * ここには割り込み禁止で来ること
	 */
	mov.l	_schedtsk_dis,r12	/* r0 <- schedtsk */
	mov.l	@r12,r0
	cmp/eq #0,r0			/* schedtsk があるか？ */
	bt	dispatcher_2		/* 無ければジャンプ */
	mov.l	_runtsk_dis,r2
	mov.l	r0,@r2			/* schedtskをruntskに */
	mov.l	@(TCB_sp,r0),r15	/* TCBからタスクスタックを復帰 */
	mov.l	@(TCB_pc,r0),r1		/* TCBから実行再開番地を復帰 */
	jmp	@r1
	nop
dispatcher_2:
	/*
	 *	ここで割込みモードに切り換えるのは，ここで発生する割込み処理
	 *	にどのスタックを使うかという問題の解決と，割込みハンドラ内で
	 *	のタスクディスパッチの防止という二つの意味がある．
	 */
	mov.l	_stacktop_dis,r15	/* スタックを割り込みスタックに */
	mov	#0x01,r10
	ldc	r10,r7_bank		/* r7_bank0 を1にして割り込み状態に */
	mov.l	_mask_md_dis,r9		/* 割り込み許可 */
	ldc	r9,sr
	sleep				/* 割込み待ち */
	mov.l	_mask_md_ipm_dis,r8	/* 割り込み禁止 */
	ldc	r8,sr
	dt	r10			/* r7_bank0 をクリア */
	ldc	r10,r7_bank
	bra	dispatcher_1
	nop
	.align 4
_runtsk_dis:
	.long _runtsk
_schedtsk_dis:
	.long _schedtsk
_calltex_dis:
	.long _calltex
_mask_md_ipm_dis:
	.long 0x40000000 + MAX_IPM << 4
_mask_md_dis:
	.long 0x40000000
dispatch_r_k:
	.long dispatch_r
_stacktop_dis:
	.long STACKTOP		/* タスク独立部のスタックの初期値 */
```

disptcher_2 にて割込待ち状態に移行後、割込処理から飛んでくる割込ハンドラの出口処理の動作については、以下の通りである。
割込処理で reqflg が ON ならこの処理に入るが、ここで runtsk と schedtsk 比較し、異なる場合はディスパッチ処理を呼び出している。

```
/*
 *   割り込みハンドラ/CPU例外ハンドラ出口処理
 *
 * 戻り先がタスクでreqflgがセットされている場合のみここにくる。
 * r7_bank = 0,割り込み禁止状態,スクラッチレジスタを保存した
 * 状態で呼び出すこと。
 *
 */
      .text
      .align 2
      .globl ret_int
      .globl ret_exc
ret_exc:
ret_int:
      mov.l   _runtsk_ret,r1     /* r0 <- runtsk                          */
      mov.l   @r1,r0
      mov.l   _enadsp_ret,r2     /* enadspのチェック                      */
      mov.l   @r2,r3
      tst       r3,r3
      bt        ret_int_1
      mov.l   _schedtsk_ret,r4 /* r5 <- schedtsk                          */
      mov.l   @r4,r5
      cmp/eq r0,r5                  /* runtsk と schedtsk を比較          */
      bt        ret_int_1
      mov.l   r14,@-r15       /* 残りのレジスタを保存                     */
      mov.l   r13,@-r15
      mov.l   r12,@-r15
      mov.l   r11,@-r15
      mov.l   r10,@-r15
      mov.l   r9,@-r15
      mov.l   r8,@-r15
      sts.l   mach,@-r15
      sts.l   macl,@-r15
      stc.l   gbr,@-r15
      mov       #TCB_sp,r1          /* タスクスタックを保存               */
      mov.l   r15,@(r0,r1)
      mov.l   ret_int_r_k,r1    /* 実行再開番地を保存                     */
      mov       #TCB_pc,r2
      mov.l   r1,@(r0,r2)
      bra       dispatcher_1
      nop
ret_int_r:
      ldc.l   @r15+,gbr           /* レジスタを復帰                       */
      lds.l   @r15+,macl
      lds.l   @r15+,mach
      mov.l   @r15+,r8
      mov.l   @r15+,r9
      mov.l   @r15+,r10
      mov.l   @r15+,r11
      mov.l   @r15+,r12
      mov.l   @r15+,r13
      mov.l   @r15+,r14
ret_int_1:
      mov.l   _calltex_ret,r2       /* タスク例外処理ルーチン起動         */
      jsr       @r2
      nop
```

```
#ifdef SUPPORT_CHG_IPM
        mov       #32,r0
        mov.l   @(r0,r15),r1
        mov.l   _unmask_ipm,r2
        and       r2,r1
        mov.l   _task_intmask_k,r2
        mov.l   @r2,r3
        or        r3,r1
        mov.l   r1,@(r0,r15)
#endif /* SUPPORT_CHG_IMP */
        mov.l   @r15+,r7            /* spc,pr,ssr,スクラッチレジスタを復帰  */
        mov.l   @r15+,r6
        mov.l   @r15+,r5
        mov.l   @r15+,r4
        mov.l   @r15+,r3
        mov.l   @r15+,r2
        mov.l   @r15+,r1
        mov.l   @r15+,r0
        ldc.l   @r15+,ssr
        lds.l   @r15+,pr
        ldc.l   @r15+,spc
        rte
        nop
        .align 4
_calltex_ret:
        .long _calltex
_runtsk_ret:
        .long _runtsk
_schedtsk_ret:
        .long _schedtsk
_enadsp_ret:
        .long _enadsp
ret_int_r_k:
        .long ret_int_r
```

### 4.4 makeoffset.c

makeoffset.c は offset.h 生成サポートプログラムである。

TCB 構造体の各メンバーへのオフセットを定義したものである。

```
#include "jsp_kernel.h"
#include "task.h"

#define offsetof(structure, field) ¥
                                ((INT) &(((structure *) 0)->field))

#define OFFSET_DEF(TYPE, FIELD)                                                    ¥
        Asm("! BEGIN¥n" #TYPE "_" #FIELD " = %0¥n¥t! END"                          ¥
          : /* no output */                                                        ¥
          : "g"(offsetof(TYPE, FIELD)))

#define OFFSET_DEF2(TYPE, FIELD, FIELDNAME)                                        ¥
        Asm("! BEGIN¥n" #TYPE "_" #FIELDNAME " = %0¥n¥t! END"                      ¥
          : /* no output */                                                        ¥
          : "g"(offsetof(TYPE, FIELD)))

void
makeoffset()
{
        OFFSET_DEF2(TCB, tskctxb.sp, sp);
       OFFSET_DEF2(TCB, tskctxb.pc, pc);
}
```

TCB の先頭から、TCB→tskctxb→sp までのオフセット値を求め、その値を「TCB_sp」という名前に定義しているのがこのプログラムである。

これは cpu_support.s 内で、TCB 構造体のメンバーへアクセスする際に利用されている。

具体例としては以下の通りである。

```
        .text
        .align 2
        .global _dispatch
_dispatch:
        sts.l   pr,@-r15            /* pr,r8〜r15 をスタックに保存                */
        mov.l   r14,@-r15             /* r0〜r7 は呼び出し元で保存しているため   */
        mov.l   r13,@-r15             /*  保存する必要が無い                      */
        mov.l   r12,@-r15
        mov.l   r11,@-r15
        mov.l   r10,@-r15
        mov.l   r9, @-r15
        mov.l   r8, @-r15
        mov.l   _runtsk_dis,r2    /* r0 <- runtsk                              */
        mov.l   @r2,r0
        mov.l   r15,@(TCB_sp,r0) /*  タスクスタックを TCB に保存  60 以下なら OK*/

        mov.l   dispatch_r_k,r1   /*  実行再開番地を保存                        */
        mov.l   r1,@(TCB_pc,r0)   /*  実行再開番地を TCB に保存   60 以下なら OK */
        bra     dispatcher_1
    nop
```

ここではレジスタをスタックに保存後、r15 のスタックポインタを TCB 構造体に含まれる CTXB の内の sp に保存する必要がある。アセンブラでは直接構造体のメンバーを参照することが出来ないので、構造体の先頭からのオフセット値＝TCB_sp を使っている。r0 に runtsk のアドレスを入れた後、r15 の値を r0+TCB_sp のアドレスに入れることが可能になっている。