カーネル勉強会資料

冨樫 拓寛

平成 13 年 9 月 3 日

1 概要

μITRON4.0 で定められている割込み管理機能のサービスコール・静的 API のうち、今回使用している JSP カーネルでは、SH-3 用に DEF_INH が提供されている。DEF_INH では、割込み番号、割込みハンドラの起動番地に基づいて割込みハンドラを定義する。静的 API として DEF_INH を使用して割込みハンドラを登録すると、_kernel_inhinib_table に割込み番号、起動番地のテーブルが作成される。

更に、カーネルオブジェクト初期化ルーチン (object_initialize()) に割込みハンドラ管理機能初期化関数 (interrupt_initialize()) が追加され、カーネル起動時に呼び出される。interrupt_initialize() はターゲット非依存の関数であり、_kernel_inhinib_table を読み込み、ターゲット依存の割込みハンドラ登録関数 (define_inh()) を呼ぶ。

同様に、CPU 例外ハンドラの定義を行う静的 API(DEF_EXC) を使用して、例外ハンドラを登録すると、_kernel_excinib_table に CPU 例外番号、ハンドラの起動番地のテーブルが作成され、カーネル起動時に CPU 例外ハンドラ管理機能初期化関数 (exception_initialize()) が呼び出される。この関数では、ターゲット依存の CPU 例外ハンドラ登録関数 (define_exc()) を呼ぶ。

2 機種依存部のコード

2.1 define_inh(), define_exc()

define_inh() では割込みハンドラの登録、define_exc() では CPU 例外ハンドラの登録を行う。define_inh では、最初に割込みハンドラの擬似テーブルである int_table にハンドラの起動番地を格納する。STUB を併用する場合は、次に trapa 命令を実行して STUB を呼び出して登録する。STUB 側のテーブルには割込み番号と interrupt() のアドレスが格納される。(詳細は後で)

なお、割込み/例外処理時の実際に実行されるルーチンは STUB のルーチンであり、JSP の例外処理コード (cpu_support.s) で記述されているコードについては STUB から間接的に呼ばれる。JSP が使用する割込み/例外要因については、あらかじめ trapa 命令で STUB を呼び、STUB で用意されている擬似テーブルに割込み要因発生時の実行アドレスを登録する (図 1 参照)。JSP 側で宣言した要因以外の要因が入ると JSP では異常処理として扱われる (STUB 側で拾われる)。

また、ベクタベースレジスタ (VBR) には例外処理ベクタ領域のベースアドレスが格納されるが、STUB と併用する場合の VBR の値は STUB によって設定される。

```
Inline void
define_inh(INHNO inhno, FP inthdr)
{
        int_table[inhno >> 5] = inthdr;
#ifdef WITH_STUB
        Asm("mov #0x8,r0;  mov %0,r4; mov %1,r5; trapa #0x3f"
    : /* no output */
    : "r"(inhno),"r"(interrupt)
    : "r0", "r4", "r5");
#endif
}

Inline void
define_exc(EXCNO excno, FP exchdr)
{
        exc_table[excno >> 5] = exchdr;
#ifdef WITH_STUB
        Asm("mov #0x8,r0;  mov %0,r4; mov %1,r5;  trapa #0x3f"
    : /* no output */
    : "r"(excno),"r"(general_exception)
    : "r0", "r4", "r5");
#endif
}
```

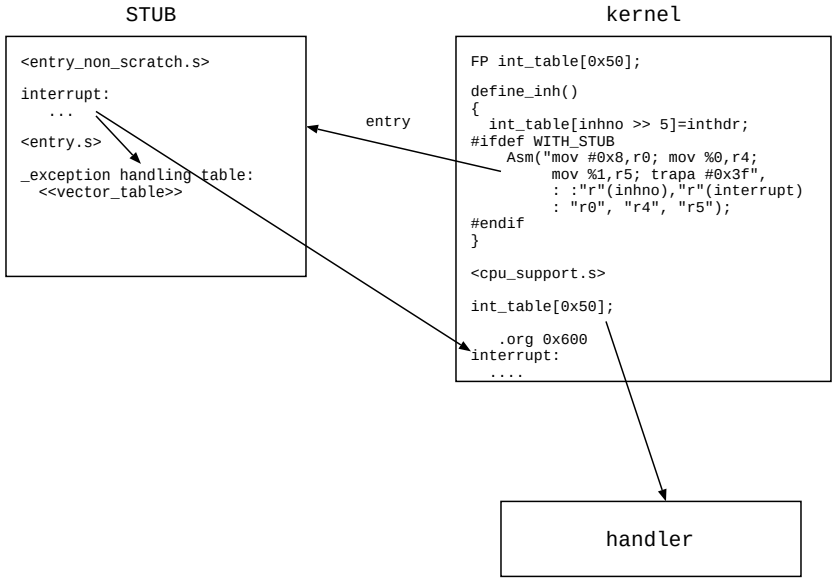

図 1: 割込みハンドラの登録

2.2 割込みレベルの設定

SH-3 の割り込みコントローラ (INTC) では、割り込み要因の優先順位を判定し、CPU への割り込み要求を制御する。INTC は、各割込みの優先順位を設定するためのレジスタがあり、ユーザがこのレジスタに設定した優先順位に従って、割込み要求が処理される。そのため、割り込みを行うモジュールごとに HW と SW の割込みレベルを行う必要がある。hw_timer.h の場合の初期化例を下に示す。define_int_plevel() の関数を実行すると、割込み優先度が割込み優先度テーブル (int_plevel_table) に書き込まれ、HW 側の設定としては割込み優先レベル設定レジスタ (IPRA：16 ビットレジスタに 4 種類の機器の優先レベルを書き込む) に優先度が書き込まれる。なお、割込み番号は 4 ビット

である。

```
＜ hw_timer.h ＞
Inline void
hw_timer_initialize()
{
    .
    .
/*
 *  割込み関連の設定
 */
    define_int_plevel(TMU0_INTEVT,TINTLVL0);  /* 割り込みレベル設定 (SF) */
    *IPRA=(*IPRA&0x0fff)|(TINTLVL0<<12);      /* 割り込みレベル設定 (HW) */
    TMU.TCR0 &= ~TCR_UNF;                     /* 割り込み要求をクリア   */
}

＜ cpu_config.h ＞
Inline void
define_int_plevel(UINT dintno, UW plevel)
{
    int_plevel_table[dintno >> 5] = (plevel << 4) | 0x40000000;
}
```

2.3 割込みマスク

2.3.1 起動時の割込みマスクの設定

```
＜ cpu_config.h ＞
#define SUPPORT_CHG_IPM

＜ cpu_config.c ＞
void
cpu_initialize()
{
/*
*  タスクコンテキストでの割込みマスクの初期化
*/
#ifdef SUPPORT_CHG_IPM
    task_intmask = 0x0000;
#endif /* SUPPORT_CHG_IPM */

#ifndef WITH_STUB
    /*
     * 割り込みコントローラの初期化
     */
    *ICR0 = 0x0000;
    *IPRA = 0x0000;
    *IPRB = 0x0000;

#ifndef SH7708
    *ICR1 = 0x0000;
    *ICR2 = 0x0000;
    *PINTER = 0x0000;
    *IPRC = 0x0000;
    *IPRD = 0x0000;
    *IPRE = 0x0000;
    *IRR0 = 0x0000;
    *IRR1 = 0x0000;
    *IRR2 = 0x0000;
#endif

    /*
     *  ベクタベースレジスターの初期化
     */
    set_vbr(BASE_VBR);
#endif
}
```

SH-3 では chg_ipm がサポートされているため、cpu_config.c でタスクコンテキストでの割込みマスクの初期化を行う。また、STUB を使用しない場合は割込みコントローラの初期化を行う。

2.3.2 割込みマスクの変更・参照

```
<cpu_config.c>
/*
 *  割込みマスクの変更
 */

SYSCALL ER
chg_ipm(IPM ipm)
{
    CHECK_TSKCTX_UNL();
    CHECK_PAR(0 <= ipm && ipm <= MAX_IPM - 1);

    t_lock_cpu();
    task_intmask = (ipm << 4);
    t_unlock_cpu();
    return(E_OK);
}

/*
 *  割込みマスクの参照
 */
SYSCALL ER
get_ipm(IPM *p_ipm)
{
    CHECK_TSKCTX_UNL();

    t_lock_cpu();
    *p_ipm = (task_intmask >> 4);
    t_unlock_cpu();
    return(E_OK);
}

<check.h(エラーチェック用マクロ)>
/*
 *  その他のパラメータエラーのチェック（E_PAR）
 */
#define CHECK_PAR(exp) { \
    if (!(exp)) { \
        return(E_PAR); \
    } \
}

/*
 *  呼出しコンテキストと CPU ロック状態のチェック（E_CTX）
 */
#define CHECK_TSKCTX_UNL() { \
    if (sense_context() || t_sense_lock()) { \
        return(E_CTX); \
} \
}
```

chg_ipm ではまず、呼び出し元が非タスクコンテキスト、あるいはCPU ロックでないかどうかの判定を行う (CHECK_TSKCTX_UNL();)。また、引数として与えられた設定値が割込みマスクとして設定できる値の範囲かどうかの判定を行った後、クリティカルセクション内で割込みマスクの書き込みを行う。

また、get_ipm では CHECK_TSKCTX_UNL() の判定を行った後、クリティカルセクション内でポインタに現在の割り込みマスク値を代入する。

なお、chg_ipm を使って割り込みマスクを MAX_IPM （NMI スタブリモートブレーク 以外のすべての割込みを禁止）以上に変更することはできない。NMI スタブリモートブレーク以外のすべての割込みを禁止したい場合には、

loc_cpu により CPU ロック状態にすればよい。割り込みマスクが 0 以外の時にも、タスクディスパッチは保留されない。割り込みマスクは、タスクディスパッチによって、新しく実行状態になったタスクへ引き継がれる。そのため、タスクが実行中に、別のタスクによって 割り込みマスクが変更される場合がある。JSP カーネルでは、割り込みマスクの変更はタスク例外処理ルーチンによっても起こるので、これによって扱いが難しくなる状況は少ないと思われる。割り込みマスクの値によってタスクディスパッチを禁止したい場合には、dis_dsp を併用すればよい。

3 割込み/例外処理

この章では割込み/例外処理のプログラム本体について説明する。表 1 に例外事象ベクタ、表 2 に各特定イベントを区別するため EXPEVT、INTEVT、INTEVT2 に書き込まれる例外コードを示す。

ベクタは、TLB ミス例外発生時は VBR+0x400、それ以外の例外処理は VBR+0x100、割込みは VBR+0x600 となっている。このアドレスに格納するのはジャンプ先となる処理ルーチンの先頭アドレスではない。そのため、割込み/例外処理ではプログラムで発生要因を一般例外なら EXPEVT、割込みなら INTEVT、INTEVT2 を読み込み判断する必要がある。

表 1: 例外事象ベクタ

例外タイプ	カレント命令	例外イベント	優先順位[*1]	例外順位	ベクタアドレス	ベクタオフセット
リセット	中断	パワーオン	1	-	0xA0000000	-
		マニュアルリセット	1	-	0xA0000000	-
		パワーオン	1	-	0xA0000000	-
		H-UDI リセット	2	-	0xA0000000	-
一般例外イベント	中断およびリトライ	CPU アドレスエラー (命令アクセス)	2	1	-	0x100
		TLB ミス	2	2	-	0x400
		TLB 無効 (命令アクセス)	2	3	-	0x100
		TLB 保護違反 (命令アクセス)	2	4	-	0x100
		予約命令コード例外	2	5	-	0x100
		スロット不当命令例外	2	5	-	0x100
		CPU アドレスエラー (データアクセス)	2	6	-	0x100
		TLB ミス (データアクセス)	2	7	-	0x400
		TLB 無効 (データアクセス)	2	8	-	0x100
		TLB 保護違反 (データアクセス)	2	9	-	0x100
		初期ページ書き込み	2	10	-	0x100
	完了	無条件トラップ (TRAPA 命令)	2	5	-	0x100
		ユーザブレークポイントトラップ	2	n[*2]	-	0x100
一般割り込み要求	完了	DMA アドレスエラー	2	12	-	0x100
		ノンマスカブル割り込み	3	-	-	0x600
		外部ハードウェア割り込み	4[*3]	-	-	0x600
		H-UDI 割り込み	4[*3]	-	-	0x600

*1 優先順位は高い方から順番に示します。1 が最高で 4 が最低です

*2 ブレークポイントトラップはユーザーが定義できます。命令実行後のブレークポイントのとき 1、命令実行後のブレークポイントのとき 11、オペラントブレークポイントのときも 11 となります。

*3 ソフトウェアで外部ハードウェア割り込みと周辺モジュール割り込みの相対的優先順位を指定してください

表 2: 例外コード

例外タイプ	例外イベント	例外コード
リセット	パワーオン	0x000
	マニュアルリセット	0x020
	H-UDI リセット	0x000
一般例外イベント	TLB ミス/TLB 無効 (ロード)	0x040
	TLB ミス/TLB 無効 (ストア)	0x060
	初期ページ書き込み	0x080
	TLB 保護違反 (ロード)	0x0A0
	TLB 保護違反 (ストア)	0x0A0
	CPU アドレスエラー (ロード)	0x0E0
	CPU アドレスエラー (ストア)	0x100
	無条件トラップ (TRAPA 命令)	0x160
	予約命令コード例外	0x180
	スロット不当命令例外	0x1A0
	ユーザーブレークポイントトラップ	0x1E0
	DMA アドレスエラー	0x5C0
一般割り込み要求	ノンマスカブル割り込み (NMI)	0x1C0
	H-UDI 割り込み	0x5E0
	外部ハードウェア割り込み IRL3-IRL0=0000	0x200
	IRL3-IRL0=0001	0x220
	IRL3-IRL0=0010	0x240
	IRL3-IRL0=0011	0x260
	IRL3-IRL0=0100	0x280
	IRL3-IRL0=0101	0x2A0
	IRL3-IRL0=0110	0x2C0
	IRL3-IRL0=0111	0x2E0
	IRL3-IRL0=1000	0x300
	IRL3-IRL0=1001	0x320
	IRL3-IRL0=1010	0x340
	IRL3-IRL0=1011	0x360
	IRL3-IRL0=1100	0x380
	IRL3-IRL0=1101	0x3A0
	IRL3-IRL0=1110	0x3C0
TMU0	TMUI0(アンダーフロー)	0x400
TMU1	TMUI1(アンダーフロー)	0x420
TMU2	TMUI2(アンダーフロー)	0x440
	TMCPI2(インプットキャプチャ)	0x460
RTC	ATI(アラーム)	0x480
	PRI(周期)	0x4A0
	CUI(桁上げ)	0x4C0
SCI	ERI(受信エラー)	0x4E0
	RXI(受信データフロー)	0x500
	TXI(受信データエンプティ)	0x520
	TEI(送信完了)	0x540
WDT	ITI(インターバルタイマ)	0x560
REF	RCMI(コンペアマッチ)	0x580
	ROVI(リフレッシュカウントオーバフロー)	0x5A0

3.1 割込み処理 (VBR+0x600)

SH3 では割込みが発生すると VBR+0x600 番地からプログラムを実行するため、VBR+0x600 番地に配置するルーチンでは、スタックの切り替え、レジスタの保存、割込みマスクの設定, 割り込み要因の判定を行いその後 BL ビットを 0 にして割り込みハンドラを呼ぶ必要がある。

具体的には、まず割込み発生元のコンテキストである SPC、PR、SSR、GBR、MACH、MACL、R0〜R7 をスタックへ退避する。なお、割込み発生時は BANK1 に切り替わるため、保存すべき R0〜R7 は BANK0 に存在するので stc 命令を使用して退避を行う。スタックのイメージを図 2 に示す。

次に割込み発生元のコンテキストを判定して、BANK1 の R7(例外/割込みネスト回数) をインクリメントした後 (遅延スロットの使用)、割込み発生時のコンテキストが非タスクコンテキストの場合は _inrerrupt_from_int へジャンプする。この場合、スタックは割込みスタックを使用しているので、スタックの変更は行わない。また、非タスクコンテキストの場合は終了後に元の処理に戻る必要がある。割込み発生時のコンテキストがタスクコンテキストの場合はスタックをユーザスタックから割込みスタックへ切り替えて、元 (ユーザスタック) のスタックポインタを割込みスタックへ退避させる。

割込み発生時のコンテキストがタスクコンテキストの場合は次に、割込み要因を INTEVT レジスタから取得して、そこから割込み優先度レベルと割込みハンドラの開始番地をそれぞれ int_plevel_table[] と int_table[] から呼び出す。その際のオフセットの計算は、割り込み要因レジスタを右に 3 ビットシフトして行う。割り込み要因レジスタは SH7708 では INTEVT レジスタにセットされるが、SH7709 および SH7709A では INTEVT2 にセットされるため、ifdef により切り分けている。割込みハンドラの開始番地を取得後、割込みハンドラへジャンプするが、その前に割り込み許可レベルとレジスタバンクの変更を行うため、あらかじめ割込みハンドラのアドレスはレジスタバンク 0 へコピーしておく。

割込みハンドラの実行終了後、まず＜ ldc r0,sr ＞の命令でステータスレジスタの RB(レジスタバンクビット)、BL(ブロックビット) を 0 にセットして割込み禁止にする。次に、例外/割り込みのネスト回数をデクリメントした後、reqflg を参照して、ディスパッチ・タスク例外処理ルーチンの起動要求が入っているかどうかチェックを行う。そして、スタックを割込みスタックからユーザスタックへ戻す。reqflg をチェックする前に割込みを禁止しないと、reqflg をチェック後に起動された割込みハンドラ内でディスパッチが要求された場合に、ディスパッチされない。そこで、起動要求がない場合は ret_to_task_int へジャンプして R0〜R7,SSR,SPC,PR,GBR,MACH,MACL を復帰してタスクに戻る。起動要求がある場合は、reqflg をクリアした後、割込みハンドラ/CPU 例外ハンドラ出口処理 (ret_int()) へジャンプする。

3.1.1 interrupt_from_int()

interrupt_from_int は割込み発生時のコンテキストが非タスクコンテキストの場合の処理である。タスクコンテキストの場合との処理の違いは、割込みハンドラの処理の終了後にそのまま ret_to_task_ int のルーチンを実行するので必ず元の処理に戻ることである。

3.2 例外処理 (内部要因)(VBR+0x100)

まず、例外発生元のコンテキストである SPC、PR、SSR、GBR、MACH、MACL、R0〜R7 をスタックへ退避する。なお、例外発生時は BANK1 に切り替わるため、保存すべき R0〜R7 は BANK0 に存在するので stc 命令を使用して退避を行う。

次に例外発生元のコンテキストを判定して、BANK1 の R7(例外/割込みネスト回数) をインクリメントした後、CPU 例外発生時のコンテキストが非タスクコンテキストの場合は_exception_from_exc へジャンプする。CPU 例外発生時のコンテキストがタスクコンテキストの場合は、戻り先が割込みハンドラでないので、スタックをユーザスタックから割込みスタックへ切り替えて、元 (ユーザスタック) のスタックポインタを割込みスタックへ退避させる。

次に例外事象レジスタ (EXPEVT) の内容を取得して 3 ビット右シフトを行い、STUB で用意されている擬似テー

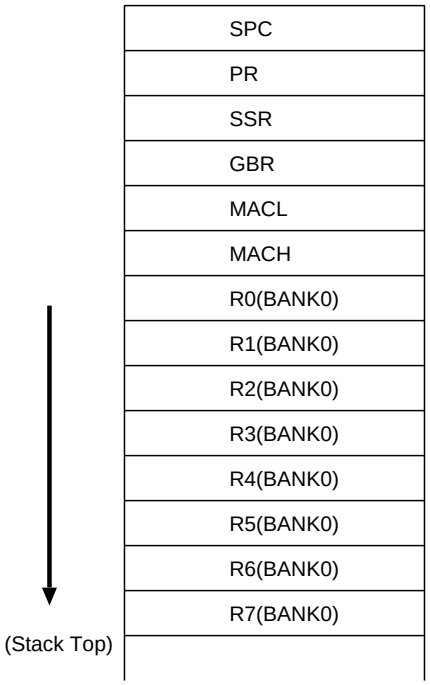

図 2: レジスタ退避時のスタックイメージ

ブル上の対応するエントリからハンドラの開始アドレスを取得する。その際にハンドラが登録されていない例外の場合は no_reg_exception(→ cpu_config.c 中の cpu_expevt()) へジャンプする (no_reg_exception では例外が発生した時点における EXPEVT、SPC、SSR、PR の内容を出力してカーネルを停止する)。

ハンドラが登録されている場合は、そのアドレスへジャンプする。ハンドラの処理の終了後、まず< ldc r0,sr >の命令でステータスレジスタの RB(レジスタバンクビット)、BL(ブロックビット) を 0 にセットして割込み禁止にする。次に、例外/割込みのネスト回数のデクリメントを行った後 reqflg のチェックを行い、ディスパッチ・タスク例外処理ルーチンの起動要求が入っているかどうかチェックを行った後、スタックを割込みスタックからユーザスタックへ戻す。起動要求がなければ ret_to_task_exc へジャンプして R0〜R7、SSR、SPC、PR、GBR、MACH、MACL を復帰してタスクに戻る。起動要求がある場合は、reqflg をクリアして割込みハンドラ/CPU 例外ハンドラ出口処理 (ret_exc()) へジャンプする。

3.2.1 exception_from_exc()

exception_from_exc は CPU 例外発生時のコンテキストが非タスクコンテキストの場合の処理である。タスクコンテキストの場合との処理の違いは、CPU 例外ハンドラの処理の終了後にそのまま ret_to_task_ exc のルーチンを実行するので必ず元の処理に戻ることである。

A　インライン・アセンブラ

C のソースの中にアセンブラを記述する。gcc では、この部分も最適化の対象になる。インライン・アセンブラの一般形は、

```
__asm( "mnemonic1 %0, %1; mnemonic2...."
       : output operands
       : input operands
       : clobberred hard registers);
```

() 内はコロンで 4 つの部分からなる。最初の項目にはコードを生成するニーモニックを記述する。複数に渡る場合にはセミコロンもしくは
n で区切って記述する。**output operands** には、結果をセットするレジスタ (C 言語の左辺値)、**input operands** には入力を渡す式を記述する。**clobbered hard registers** はこのアセンブラによって破壊されるレジスタをメモリが書き変る場合は”memory” と記述する。**output operands** と **input operands** は、“制御文字”(C の式) という形式で記述する。制御文字には、

”g” 任意のアドレッシング・モードを使用する

”r” レジスタを使用する

”d” データ・レジスタを使用する (68K 版)

”a” アドレス・レジスタを使用する (68K 版)

”f” 浮動小数点レジスタを使用する (68K 版)

などがある。だたし **output operands** では **“=”** をつけて **“=g”,”=r”** のように記述する。
インライン・アセンブラの使用例を次に示す。

```
void
define_inh(INHNO inhno, FP inthdr)
{
        int_table[inhno >> 4] = inthdr;
        __asm("mov %0,r0; mov %1,r1; trapa #0x50"
              : /* no output */
              : "r"(inhno),"r"(interrupt)
              : "r0", "r1");
}

unsigned int
set_sr(unsigned int new_sr){
unsigned int old_sr;
        __asm("stc sr,%0; ldc %1, sr"
              : "=r"(old_sr)
              : "r"(new_sr)
              :);
return(old_sr);
}
```

2 番目のコードを”-S” オプションでコンパイルすると次のアセンブラコードが生成される。

```
_set_sr:
        mov.l   r14,@-r15
        mov     r15,r14
        stc     sr,r0
        ldc     r4, sr
        mov     r14,r15
        rts
        mov.l   @r15+,r14
```