micro T-Kernel Implementation Specification

AT91(ARM7TDMI)

Version 1.01.02
April, 2013

Preface

This document gives the specifications for implementing micro T-Kernel on a specific target board.

The applicable board is T-Engine Appliance AT91(ARM7TDMI).

The applicable micro T-Kernel version is 1.01.01. The T-Kernel is compliant with the "micro T-Kernel Specification version 1.01.01".

The specifications given in this document correspond to the hardware-dependent implementation-specific parts of the micro T-Kernel specification.

Refer to the micro T-Kernel specification in regard to the specification of micro T-Kernel.

In addition, refer to the relevant specifications as for the specifications of hardware such as board and CPU.

Contents

1. CPU

1.1. Hardware Specifications

```
CPU  : AT91M55800A with ARM7TDMI core
       Atmel Corporation
ROM  : Flash ROM       4 MB
RAM  : On-chip SRAM    8 KB
       External SRAM   2 MB
```

1.2. Protection Levels and Operating Modes

Protection levels correspond to CPU operating modes as follows.

Protection Level	Operation Mode
3	(treated as protection level 0)
2	(treated as protection level 0)
1	(treated as protection level 0)
0	SVC : Supervisor mode

Even if any of the protection levels as above is specified, it shall be treated as protection level 0.

CPU Operation mode of user mode (USR) and system mode (SYS) is not used.

1.3. Use of the Thumb Instruction Set

When the Thumb instruction set is used, programming shall follow the "ARM/Thumb Interworking" directives. When C language (GNU C) is used, the "-mthumb-interwork-thumb" option is designated at compiling.

When a task or handler written using the Thumb instruction set is designated, 1 is set to the least significant bit (LSB) of the address. When the ARM instruction set is used, this bit is cleared to 0. Normally this is handled by the linker automatically and so programmers are not required to be aware of this.

Thumb instruction set is not used in kernel.

2. Memory

2.1. Overall Memory Map

Overall system memory maps are shown below.

[Before Remap]

```
0x00000000 +----------------------------+
           |    External FlashROM       | 0x00000000-0x000fffff
0x00100000 +----------------------------+
           |        (Reserved)          |
0x00300000 +----------------------------+
           |    On-chip SRAM     (8KB)  | 0x00300000-0x00301fff
0x00302000 +----------------------------+
           |        (Reserved)          |
0xffc00000 +----------------------------+
           |    On-chip Peripherals     | 0xffc00000-0xffffffff
0xffffffff +----------------------------+
```

Before executing the remap command, accessible only to on-chip SRAM, external FlashROM Selected by NCS0, and on-chip Peripherals.

[After Remap]

```
0x00000000 +----------------------------+
           |    On-chip SRAM     (8KB)  | 0x00000000-0x00001fff
0x00002000 +----------------------------+
           |        (Reserved)          |
0x10000000 +----------------------------+
           |    External FlashROM(4MB)  | 0x10000000-0x103fffff
0x10400000 +----------------------------+
           |        (Reserved)          |
0x20000000 +----------------------------+
           |    External SRAM    (2MB)  | 0x20000000-0x201fffff
0x20200000 +----------------------------+
           |        (Reserved)          |
0x40000000 +----------------------------+
           | Ethernet controller (512B) | 0x40000000-0x400001ff
0x40000200 +----------------------------+
           |        (Reserved)          |
0xffc00000 +----------------------------+
           |    On-chip Peripherals     | 0xffc00000-0xffffffff
0xffffffff +----------------------------+
```

After executing the remap command, accessible to the entire memory space.

2.2. ROM Memory Map

Space of 4 MB is implemented in external FlashROM. The external FlashROM memory map is shown below.

```
0x10000000 +---------------------------+
           |    micro T-Kernel code    |
           |- - - - - - - - - - - - - -|
           |         (unused)          |
0x10400000 +---------------------------+
```

micro T-Kernel code shall be located at the lower byte of external FlashROM.

2.3. On-Chip SRAM Memory Map

Space of 8MB is implemented in on-chip SRAM. The on-chip SRAM memory map is shown below.

```
0x00000000 +---------------------------+  --
           |   Reset                   |  |
0x00000004 +---------------------------+  |
           |   Undefined Instruction   |  |
0x00000008 +---------------------------+  |
           |   Software interrupt(SWI) |  |
0x0000000C +---------------------------+  |
           |   Prefetch Abort          |  | Exception vector table
0x00000010 +---------------------------+  |  (ram_vector_table)
           |   Data Abort              |  |
0x00000014 +---------------------------+  |
           |     (Reserved)            |  |
0x00000018 +---------------------------+  |
           |   Interrupt (IRQ)         |  |
0x0000001C +---------------------------+  |
           |   Fast Interrupt (FIQ)    |  |
0x00000020 +---------------------------+  --
           |   Jump table              |
           | (ram_vector_address_table) |
0x00000034 +---------------------------+
           |        (unused)           |
           |- - - - - - - - - - - - - -|        --
           |                           |        |
           |   Supervisor              |        |
           +---------------------------+ R13_svc |
           |   Interrupt (IRQ)         |        |
           +---------------------------+ R13_irq |Exception/interrupt
           |   Fast Interrupt (FIQ)    |        | stack area
           +---------------------------+ R13_fiq |
           |   Undefined Instruction   |        |
           +---------------------------+ R13_und |
           |   Abort                   |        |
0x00002000 +---------------------------+ R13_abt --
```

The vector table and the jump table are located at the lower byte of on-chip RAM.

The stack area shall be allocated in descending order from the most significant byte of on-chip SRAM.

The address of stack area and the stack size of each mode other than supervisor mode can be specified in utk_config_depend.h. Although there are macro definitions for specifying the stack size of each mode in utk_config_depend.h, neither user mode (USR) nor system mode (SYS) are used in the μ T-Kernel reference implementation. Therefore USR_STACK_SIZE, the stack size corresponding to these modes, is set to 0.

The stack area of supervisor is allocated at the lower bytes in the interrupt (IRQ) stack area, and the unused RAM area is used.

2.4. External SRAM Memory Map

Space of 2MB is implemented in external SRAM. The external SRAM memory map is shown below.

```
0x20000000 +--------------------------+
           |    DATA section          |
           |- - - - - - - - - - - - - |
           |    BSS section (NoInit)  |
           |- - - - - - - - - - - - - |
           |    BSS section           |
           |- - - - - - - - - - - - - | ← SYSTEMAREA_TOP
           |    micro T-Kernel        |
           |    management area       |
           |                          |
0x20200000 +--------------------------+ ← SYSTEMAREA_END

  NoInit : BSS Section is not cleared to zero in initialization.
```

Data section and BSS section are located in ascending order from the lower byte of the external SRAM.

The micro T-Kernel management area is memory space dynamically managed by the micro T-Kernel memory management function.

Normally, all unused memory spaces are allocated to the micro T-Kernel management area, but this can be changed in system configuration. micro T-Kernel management area is allocated to the designated area between "SYSTEMAREA_TOP" and "SYSTEMAREA_END" in configuration file.

2.5. Stacks

micro T-Kernel has the following two kinds of stacks.

(1) System stack

This stack is used in non-interrupt handler, and one system stack exists per each task.

Since there is no concept of protection level in micro T-Kernel, unlike T-Kernel, user stack and system stack are not separated.

(2) Exception/Interrupt stacks

The stack is used in interrupt handlers. The stacks of all exception modes other than supervisor mode (svc) shall be exception/interrupt ones.

Since the interrupt stack is commonly used in system, task switching shall not happen during use.

In this implementation, stack areas for abort/undefined instruction/fast interrupt(FIQ) shall not be used.

3. Interrupts and Exceptions

3.1. Interrupt Definition Numbers

Regarding the interrupt definition number (dinto) defined with tk_def_int, while the numbers 0 to 31 support the interrupt source controlled by the interrupt controller(AIC), the numbers 32 to 111 support the software interrupts (SWI). Therefore, the numbers SWI 0 to SWI 31 can not be used.

Following is the list of the interrupt sources controlled by interrupt controllers.

```
/*
 * Interrupt Controller Sources
 */
#define FIQ       0    /* Fast interrupt */
#define SWIRQ     1    /* Software interrupt */
#define US0IRQ    2    /* USART Channel 0 interrupt */
#define US1IRQ    3    /* USART Channel 1 interrupt */
#define US2IRQ    4    /* USART Channel 2 interrupt */
#define SPIRQ     5    /* SPI interrupt */
#define TC0IRQ    6    /* Timer Channel 0 interrupt */
#define TC1IRQ    7    /* Timer Channel 1 interrupt */
#define TC2IRQ    8    /* Timer Channel 2 interrupt */
#define TC3IRQ    9    /* Timer Channel 3 interrupt */
#define TC4IRQ   10    /* Timer Channel 4 interrupt */
#define TC5IRQ   11    /* Timer Channel 5 interrupt */
#define WDIRQ    12    /* Watchdog interrupt */
#define PIOAIRQ  13    /* Parallel I/O Controller A interrupt */
#define PIOBIRQ  14    /* Parallel I/O Controller B interrupt */
#define AD0IRQ   15    /* Analog-to-digital Converter Channel 0 interrupt */
#define AD1IRQ   16    /* Analog-to-digital Converter Channel 1 interrupt */
#define DA0IRQ   17    /* Digital-to-analog Converter Channel 0 interrupt */
#define DA1IRQ   18    /* Digital-to-analog Converter Channel 1 interrupt */
#define RTCIRQ   19    /* Real-time Clock interrupt */
#define APMCIRQ  20    /* Advanced Power Management Controller interrupt */
                       /*  21  Reserved */
                       /*  22  Reserved */
#define SCLKIRQ  23    /* Slow Clock Interrupt */
#define IRQ5     24    /* External interrupt 5 */
#define IRQ4     25    /* External interrupt 4 */
#define IRQ3     26    /* External interrupt 3 */
#define IRQ2     27    /* External interrupt 2 */
#define IRQ1     28    /* External interrupt 1 */
#define IRQ0     29    /* External interrupt 0 */
#define COMMRX   30    /* RX Debug Communication Channel interrupt */
#define COMMTX   31    /* TX Debug Communication Channel interrupt */
```

3.2. Software Interrupt Assignments

The numbers 36 to 39 are used for software interrupts.

Each software interrupt is used as follows.

```
SWI 36    micro T-Kernel system call/extended SVC
SWI 37    "tk_ret_int()" system call
SWI 38    Task dispatcher call
SWI 39    Debugger support function
```

3.3. Exception/Interrupt Handler

When an exception is raised, a branch routine that refers to the exception vector table and jumps to each handler is started, and then vector address is set in ip(R12) register. By this vector table address, the raised exception vector number can be found. The raised exception vector number can be found as follows based on the vector table address.

(ip - knl_intvec) / 4 = vector number

The processing jumps to handler with an interrupt disable state. Besides, since the bx instruction is used for jumping to a handler, it switches to Thumb mode if the LSB of the handler address set in the vector table is 1. Otherwise the jump is executed in continuous ARM mode.

Some registers are used for branch processing in branch routine. These registers are saved to stack as shown below. The registers other than the following shall be saved in each handler as needed, because these are not saved in the branch routine.

In addition, when returning from handlers, the registers, including those saved in branch routine, need to be returned.

```
Stack when handler is invoked
        +---------------+
   ssp->| R3            |
        | SPSR          |
        | R12 = ip      |
        | R14 = lr      |
        +---------------+

Register status when handler is called
   ip = Vector table address
   lr, R3 = Indeterminate
```

3.4. Exception/Interrupt Handling Routines

Examples of exception/interrupt handling routines are given below.

```
/*
 * Program status register (PSR)
 */
#define PSR_N     0x80000000      /* condition flag negative */
#define PSR_Z     0x40000000      /*                zero     */
#define PSR_C     0x20000000      /*                carry    */
#define PSR_V     0x10000000      /*                overflow */

#define PSR_I     0x00000080      /* interrupts (IRQ) disabled */
#define PSR_F     0x00000040      /* fast interrupts (FIQ) disabled */
#define PSR_T     0x00000020      /* Thumb mode */

#define PSR_M(n)  ( n )           /* processor mode 0 to 31 */
#define PSR_USR   PSR_M(16)       /* user mode */
#define PSR_FIQ   PSR_M(17)       /* fast interrupt (FIQ) mode */
#define PSR_IRQ   PSR_M(18)       /* interrupt (IRQ) mode */
#define PSR_SVC   PSR_M(19)       /* supervisor mode */
#define PSR_ABT   PSR_M(23)       /* abort mode */
#define PSR_UND   PSR_M(27)       /* undefined instruction mode */
#define PSR_SYS   PSR_M(31)       /* system mode */

#define N_INTVEC  112             /* vector table size */


/*
 *   Software interrupts numbers for micro T-Kernel
 */

#define SWI_SVC       36  /* micro T-Kernel system call/extended SVC */
#define SWI_RETINT    37  /* tk_ret_int() system call */
#define SWI_DISPATCH  38  /* task dispatcher */
#define SWI_DEBUG     39  /* debugger support function */

/* ------------------------------------------------------------------ */

/*
 * Exception branch routine
 */
      .section .vector,"ax"
      .code 32
      .align 0
      global __reset
__reset:
      B         start             /* 00:reset */
      .global   undef_vector
undef_vector:
      b         undef_vector      /* 04:undefined instruction exception */
      .global   swi_vector
swi_vector:
      b         swi_handler       /* 08:software interrupts (SWI) */
      .global   prefetch_vector
prefetch_vector:
      b         prefetch_vector   /* 0C:prefetch abort */
      .global   data_abort_vector
data_abort_vector:
      b         data_abort_vector /* 10:data abort */
      global    reserved_vector
reserved_vector:
      b         reserved_vector   /* 14:(reserved) */
      .global   irq_vector
irq_vector:
      ldr       pc, [pc, #-0xf20]  /* 18:interrupt (IRQ:AIC_IVR) */
      .global   fiq_vector
fiq_vector:
      ldr       pc, [pc, #-0xf20]  /* 1C:fast interrupt (FIQ:AIC_FVR) */

/*
 * Software interrupts (SWI)
 */

swi_handler:
      str    lr, [sp, #-4]!
      str    ip, [sp, #-4]!
      mrs    ip, spsr
      str    ip, [sp, #-4]!

      ldr    ip, [lr, #-4]          /* load SWI No. */
      bic    ip, ip, #(0xff << 24)

      ldr    lr, =Csym(knl_intvec)  /* exception vector table */
      add    ip, lr, ip, LSL #2     /* lr := lr + ip*4 = vecaddr */
      ldr    lr, [ip]
      bx     lr

/*
 * interrupts (IRQ)
 */
      .global knl_irq_handler
knl_irq_handler:
      sub    lr, lr, #4
      stmfd  sp!, {lr}      /* sp-> lr_xxx */

#if USE_PROTECT_MODE
      ldr    lr, =AIC_BASE
      str    lr, [lr, #AIC_IVR]
#else
      ldr    lr, =AIC_BASE
      ldr    lr, [lr, #AIC_IVR]
#endif

      stmfd  sp!, {ip}   /* sp-> ip, lr_xxx */
      mrs    ip, spsr
      stmfd  sp!, {ip}   /* sp-> spsr_xxx, ip, lr_xxx */
      stmfd  sp!, {r3}   /* sp-> r3, spsr_xxx, ip, lr_xxx */

      ldr    lr, =(AIC_BASE | AIC_ISR)
      ldr    lr, [lr]               /* lr := IRQ No. */
      ldr    ip, =Csym(knl_intvec)  /* exception vector table */
      add    ip, ip, lr, LSL #2     /* ip := &vector[IRQ No.] */
      ldr    r3, [ip]               /* r3 := vector[IRQ No.] */
      mov    lr, pc
      bx     r3
```

```
/* ------------------------------------------------------------------ */
/*
 * Exception return processing (asm_depend.h)
 */
.macro EXC_RETURN
      .arm
      ldmfd  sp!, {ip}
      msr    spsr_fsxc, ip
      ldmfd  sp!, {ip, pc}^
.endm

/* ------------------------------------------------------------------- */
/*
 * Return from handler by tk_ret_int()
 */
.macro TK_RET_INT_FIQ mode
      .arm
      mov  r3, lr                     // r3 = lr_svc
      msr  cpsr_c, #PSR_I|PSR_F|\mode // Return to original exception mode
      swp  r3, r3, [sp]               // Save lr_svc and restore r3
      swi  SWI_RETINT
.endm

/* ------------------------------------------------------------------- */
```

4. Initialization and Startup Processing

4.1. micro T-Kernel Startup Procedure

When system is reset, micro T-Kernel starts up.

The procedures from the startup of micro T-Kernel to the call of main function are as follows.

icrt0.S

(1) Switch to supervisor mode [start:]

(2) Set FlashROM (WaitState:5) [flashrom_init:]

(3) Set Crystal (clock frequency:16MHz) [crystal _init:]

(4) Set an exception vector to the top of SRAM [setup_ram_vectors:]

(5) Set a FlashROM (WaitState:5) [setup_ram_vectors:]

(6) Set an external SRAM (WaitState:3) [setup_ram_vectors:]

(7) Set an external ethernet controller [setup_ram_vectors:]

(8) REMAP [setup_ram_vectors:]

(9) Set the Mode of an interrupt controller [after_remap_start:]

(10) Set the stack pointer (from the end of on-chip SRAM) [init_stacks:]

R13_abt : Abort mode

R13_und : Undefined instruction exception mode

R13_fiq : fast interrupt (FIQ) mode

R13_irq : Interrupt (IRQ) mode

R13_svc : supervisor mode

(11) Validate TC0, TC1, and TC2 [tm_init:]

(12) Set the serial (115200 bps, 8bit, non-parity, and 1 stop bit) [tm_init:]

(13) Set the initialization value of data section (ROM->RAM)

(14) Clear BSS Section to 0

(15) Recalculate the micro T-Kernel management area

(16) Call "main" function (sysinit_main.c) [kernel_start:]

4.2. User initialization program

A user initialization program is the routine to initialize/terminate the system defined by user. The user initialization program is called in the following format from the initial task.

```
INT userinit( INT flag )

flag          = 0        call at initialization
              = -1       call at termination

Return code:  1          Startup usermain()
              Others     terminate the system
```

This program is called with flag=0 at a system initialization, and called with flag=1 at a system termination. Return value is ignored in calling at a system termination. Following is a processing flow.

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

The user initialization program is executed in the context of initial task. The task priority is (CFN_MAX_PRI-2).

5. micro T-Kernel Implementation Definitions

5.1. System State Detection

(1) Task-independent portion (interrupt handler or time event handler)

Detection is made based on a software flag set in micro T-Kernel.

knl_taskindp = 0	Task portion
knl_taskindp > 0	Task-independent portion

(2) Quasi-task portion (extended SVC handler)

Detection is made based on a software flag set in micro T-Kernel.

sysmode of TCB = 0	Task portion
sysmode of TCB > 0	Quasi-task portion

5.2. Exceptions/Interrupts Used by micro T-Kernel

SWI 36	micro T-Kernel system calls /extended SVC
SWI 37	"tk_ret_int()" system call
SWI 38	Task dispatcher call
SWI 39	Debugger support functions
TC0IRQ	Programmable timer #0(TC0)

5.3. System Call/Extended SVC Interface

The caller side can select either the method of calling interface library in C language function call format or calling directly in C language function call format. (Selectable by configuration file when building Kernel)

Ordinarily the same functional format using interface libraries as above is applied to calling from an assembler. It is also possible to directly call using a SWI instruction by the processing equivalent to that of an interface library. Even in this case, the register-saving rules must conform to the C language rules.

The basic processing of interface library is as follows.

- The function code is set in the R12 register and the system call is invoked by SWI_SVC (SWI 36). A function code in negative value indicates a system call while the one at 0 or in a positive value indicates an extended SVC. However, note that SWI_DEBUG (SWI 39) is used for Debugger Support Functions service calls.

- System calls and extended SVC are called by SWI instructions as follows.

```
Stmfd sp!, {lr}
swi   36
ldmfd sp!, {lr}
```

In principle, SWI instructions can be used only in Supervisor mode(SVC).

If a system call is invoked in other mode, it is necessary to either switch to SVC mode or save R14_svc before calling (since R14_svc shall be restored to its original value on return from the system call).

Switching to USR mode and SYS mode is not allowed.

- Registers are saved as follows in accordance with the C language register-saving rules.

```
R0 to R3, R12=ip      Temporary registers
R4 to R10             Permanent registers
R11 = fp              Frame pointer
R13 = sp              Stack pointer
R14 = lr              Link register (function return address)
R15 = pc              Program counter

Arguments:  R0 to R3
Return code: R0
```

Temporary registers are destroyed when a function call is made. Other registers are saved.

(1) System call interface

Parameters of up to fourth are set to registers, and the ones of fifth or more are saved onto the stack. A system call is invoked by SWI_SVC (SWI 36).

Registers are used as follows.

```
R12=ip  Function code (<0)
R0      First parameter
R1      Second parameter
R2      Third parameter
R3      Fourth parameter
R4      Pointer to the stack in which the parameter of
        fifth or more are stored

R0        Return code
```

An example of system call interface implementation is shown as follows.

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

Argument is the integer number of 0 to 5 pieces, and has the same format as the delivery of C language function.

```
// r0 = p1
// r1 = p2
// r2 = p3
// r3 = p4
//        +---------------+
//   sp ->| p5            |
//        +---------------+
Csym(tk_xxx_yyy):
      stmfd sp!, {r4}   // Save r4
add   r4, sp, #4        // r4 = parameter position on stack
      stmfd sp!, {lr}   // Save lr
      ldr   ip, = function code
#if USE_TRAP
      swi   SWI_SVC
#else
      bl    Csym(knl_call_entry)
#endif
      ldmfd sp!, {lr}   // Restore lr
      ldmfd sp!, {r4}   // Restore r4
      bx    lr
```

(2) Extended SVC interface library

Regarding an extended SVC, arguments are wrapped in a packet by the caller, and the start address of packet is set in R0 register. An extended SVC call is invoked by SWI_SVC (SWI 36).

Normally, the packet is created in a stack area, but can be used in other areas as well. There are no restrictions on the number or types of arguments since argument is wrapped into packet.

Register is used as follows.

```
        R12=ip    Function code (=>0)
        R0        Argument packet

        R0        Return code
```

An example of extended SVC interface implementation is shown below.

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

Argument shall be wrapped into packet, and the start address of the packet shall be set in R0 register.

```
Csym(zxxx_yyy):
      stmfd sp!, {r0-r3}     // save register arguments to stack
                                  and wrap in the packet
mov   r0, sp                 //  R0 = arguments packet address
      stmfd sp!, {lr}        //  Save lr
      ldr   ip, = function code
      swi   SWI_SVC
      ldmfd sp!, {lr}        //  Restore lr
      add   sp, sp, #4*4     // Discard arguments from stack
      bx    lr
```

(3) Debugger Support Functions system call interface

Debugger Support Functions system calls are essentially like other micro T-Kernel service calls, but are called using SWI_DEBUG(SWI 39).

An example of system call interface implementation is shown below.

```
ER td_xxx_yyy(p1, p2, p3, p4)

//      r0 = p1
//      r1 = p2
//      r2 = p3
//      r3 = p4
Csym(td_xxx_yyy):
      stmfd sp!, {lr}        // Save lr
      ldr   ip, = function code
#if USE_TRAP
      swi   SWI_DEBUG
#else
      bl    Csym(knl_call_dbgspt)
#endif
      ldmfd sp!, {lr}        // Restore lr
      bx    lr
#endif
```

5.4. Interrupt Handlers

Interrupt handler hardware-dependent implementation definitions are indicated below.

- Interrupt handler definition information : T_DINT

```
typedef struct t_dint {
    ATR  intatr;     /* interrupt handler attributes */
    FP   inthdr;     /* interrupt handler address */
} T_DINT;
```

- Interrupt definition numbers : dintno
  Definition numbers dintno can be designated in the range from 0 to 111.

An interrupt handler differs in accordance with attribute (TA_HLNG, or TA_ASM)

(1) TA_HLNG attribute interrupt handlers

When the interrupt handler attribute is TA_HLNG, the address of a high-level language support routine in micro T-Kernel is set to the exception/interrupt vector table, and the designated interrupt handler is called from the high-level language support routine.

The processor mode when exception/interrupt processing starts is any of the followings, depending on the type of interrupt/exception that was raised.

SVC	: supervisor mode
ABT	: abort mode
UND	: undefined instruction exception mode
IRQ	: interrupt mode
FIQ	: fast interrupt mode

When TA_HLNG is designated, however, the mode is automatically switched to SVC mode by the high-level language support routine. Accordingly, the mode is already SVC mode at the time of entry into the user interrupt handler regardless of the exception/interrupt type.

The interrupt handler definitions are as follows.

void inthdr(UINT dintno, VP sp)

- dintno

Vector number of the raised exception/interrupt

If the default handler is used, this is the vector number of the raised interrupt/exception, not that of the default handler.

- sp

A pointer to the following information saved in the stack:

```
        +---------------+
sp ->   | SPSR          |
        | R12=ip        |
        | R14=lr        | <- Return address
        +---------------+
Return address
FIQ:  Return address from interrupt
IRQ:  Return address from interrupt
SVC:  Return address to the instruction next to SWI instruction
ABT:  Return address to the aborted instruction
UND:  Return address to the instruction next to undefined instruction
```

The CPU state upon entry into the interrupt handler is as follows.

```
FIQ:
CPSR.F = 1   Fast interrupts disabled
CPSR.I = 1   Interrupts disabled
CPSR.M = 19  SVC: Supervisor mode

Non-FIQ:
CPSR.F = ?    Same state as when interrupt/exception was raised
CPSR.I = 1    Interrupts disabled
CPSR.M = 19   SVC: Supervisor mode
```

Nested interrupts are disabled. Interrupt nesting is enabled by clearing CPSR.I or F to 0. However, in that case, the interrupt controller settings must be properly changed. Any changes to the interrupt controller settings must be restored to their former state before returning from the interrupt handler.

The high-level language support routine in micro T-Kernel executes no processing to the interrupt controller. Operations such as clearing of the interrupt shall be executed at the interrupt handler.

A return from an interrupt handler is executed by executing "return" from function.

(2) TA_ASM attribute interrupt handlers

When the interrupt handler attribute is "TA_ASM", the interrupt handler address is directly set to the exception/interrupt vector table.

Since this handler does not go through a high-level language support routine, the flag used to detect a task-independent portion is not updated. Therefore, it is not detected as a task-dependent portion, and the following shall be noted.

If interrupts are enabled (CPSR.I=0 and F=0), a task dispatch may occur. Other than the exceptions caused by an SWI instruction, the processing after the occurrence of task dispatch will be abnormal. It is therefore necessary to set the task-independent portion detection flag as needed when enabling interrupts in a handler.

The task-independent portion flag is set by operating the knl_taskindp flag in system common information. This operation shall be executed in an interrupt disable state (CPSR.I=1 and F=1). In addition, flag shall be certainly returned before exiting the interrupt handler.

```
knl_taskindp ++;      /* enter task-independent portion */
knl_taskindp --;      /* exit task-independent portion */
```

Because of nested interrupt and the like, "knl_taskindp" shall always be set by increment and decrement. A direct value such as "knl_taskindp = 0" shall never be set.

For returning from an interrupt handler, either the "tk_ret_int()" system call or an "EXC _RETURN" macro is used. When the macro is used, delayed dispatching does not occur. Note that the original processing equivalent to that of the macro is possible, instead of using the macro.

Unlike other system calls, the "tk_ret_int()" system call is invoked using the dedicated software interrupt SWI 37. It cannot be called in functional form, unlike other system calls.

```
SWI 37    // Call "tk_ret_int()" (non-return)
```

The macro “TK_RET_INT_FIQ" are available for calling "tk_ret_int()".

Before calling "tk_ret_int()", the stack must be put in the state as shown below, and all registers other than the ones saved in the stack (R0 to R11) shall be restored.

```
             +---------------+
      sp ->  | R14_svc       |
             | SPSR          |
             | R12=ip        |
             | R14=lr        | <- Return address
             +---------------+
```

"sp" indicates the stack pointer (R13) to the raised interrupt/exception mode. The exception mode shall be returned to the mode of the raised interrupt/exception (the mode upon entering the handler).

"R14_svc" in the stack saves the value of the R14 register in SVC mode at the time of entering the handler.

The processing is not guaranteed if "tk_ret_int()" is called by non-interrupt handlers.

5.5. Time Event Handlers

As with TA_HLNG attribute, time event handler is called via high-level language support routine if TA_ASM attribute is also specified to a handler attribute. Thus, even when the TA_ASM attribute is designated, the parameter (exinf) passed to the handler is passed to the R0 register in accordance with C language rules. Register saving must also comply with C language rules.

5.6. Task implementation-dependent definitions

The definitions of task hardware-dependent implementation are given below.

(1) Task creation information T_CTSK

There is no independently added information

```
       typedef struct t_ctsk {
           VP    exinf;     /* extended information */
           ATR   tskatr;    /* task attribute */
           FP    task;      /* task start address */
           PRI   itskpri;   /* task start priority */
           W     stksz;     /* stack size (in bytes) */
           UB    dsname[8]; /* DS object name*/
           VP    bufptr;    /* user buffer pointer */
       } T_CTSK;
```

(2) Task attributes

There is no implementation-dependent information

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

(3) Task format

The format of task is as follows, and makes no difference even if either TA_HLNG or TA_ASM is specified.

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

The register states when a task is started are as follows.

```
       CPSR.F = 0  Fast interrupts enabled
       CPSR.I = 0  Interrupts enabled
       CPSR.T = 0  ARM mode  When LSB of the task start address is 0
                1  Thumb mode  When LSB of the task start address is 1
       CPSR.M =19  SVC: Supervisor mode


       R0 = stacd      Task start parameters
       R1 = exinf      Task extended information
       R13(sp)         Stack pointer
```

The other register values are indeterminate.

tk_ext_tsk() or tk_exd_tsk() shall be used to exit task. Task doesn't exit by a simple return. The behavior if return is executed is not guaranteed.

5.7. Task registers

```
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 )
```

The registers targeted for getting and setting task registers (tk_get_reg() /tk_set_reg()) are defined as follows.

(1) General-purpose registers T_REGS

```
typedef struct t_regs {
     VW  r[13];  /* general-purpose registers R0 to R12 */
     VP  lr;     /* link register R14 */
} T_REGS;
```

When setting registers to a task in DORMANT state, the task start parameters and the extended information are set in R0 and R1 by "tk_sta_tsk()", and values set by "tk_set_reg()" are therefore discarded.

(2) Registers saved when an exception is raised T_EIT

```
typedef struct t_eit {
     VP  pc;          /* program counter R15 */
     UW cpsr;         /* program status register */
     UW taskmode;     /* task mode flag */
} T_EIT;
```

No changes can be made to CPSR other than to the flag fields (bits 31 to 24). Settings made for other fields (bits 23 to 0) are ignored. taskmode is a task mode flag in system-common information. It is treated as a register holding memory access privilege information.

(3) Control registers

```
typedef struct t_cregs {
     VP  ssp;         /* system stack pointer R13_svc */
     VP  usp;         /* user stack pointer R13_usr */
} T_CREGS;
```

5.8. System Call/Extended SVC Hook Routine

The system call/extended SVC hook routine implementation-dependent definitions are as follows.

```
Hook routine definition information TD_CALINF
typedef struct td_calinf {
    VP  ssp;    /* system stack pointer */
    VP  r11;    /* frame pointer at the time of calling */
} TD_CALINF;
```

The system stack state upon entry into the hook routine is as follows.

```
                 +-------------+
          ssp -> |SPSR         | Processor mode in calling context
                 |R12=ip       |
                 |R14_svc=lr   | The return address from SWI
                 +-------------+
                 +-------------+
      R13_xxx -> |R14_xxx=lr   | The return address from interface library
                 +-------------+
```

The return address from a system call or an extended SVC is set in R14_svc. However, since normally a call is made using an interface library, R14_svc indicates the interface library address.

The return address from the interface library is set in R14_xxx. R13_xxx is the stack pointer based on the processor mode of the calling context, which can be read from SPSR. When the caller's processor mode is SVC, ssp + 12 is equivalent to R13_xxx. Note that the above assumes the use of the standard interface library and may not apply otherwise.

5.9. Interrupt Controllers

Since the interrupt controller control is highly hardware-dependent, it is not specified in micro T-Kernel. However, it is implemented as follows.

(1) Interrupt Definition Numbers

Numbers 0 to 31 among interrupt definition numbers use same number as the interrupt source of the interrupt controller (AIC).

(2) Interrupt mode setting

```
void SetIntMode( INTVEC intvec, UINT mode )
```

The mode specified by "mode" shall be set to "AIC_SMRn" register in an interrupt controller which is compatible with "intvec".

```
mode := (IM_LEVEL ∥ IM_EDGE) | (IM_HI ∥ IM_LOW)

#define IM_LEVEL  0     /* level trigger */
#define IM_EDGE   1     /* edge  trigger */
#define IM_HI     2     /* high level trigger/positive edge trigger */
#define IM_LOW    0     /* low  level trigger/negative edge trigger */
```

(3) Enable interrupt

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

Enable the interrupt designated in intvec.

(4) Disable interrupt

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

Disable the interrupt designated in intvec.

(5) Clear the interrupt request

```
void ClearInt( INTVEC intvec )
```

Clear the interrupt request designated in intvec.

This function is valid only in edge trigger mode. In edge trigger mode, interrupts need to be cleared by an interrupt handler.

(6) Check whether interrupt is requested or not.

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

Check whether there is an interrupt request for the interrupt designated in intvec or not, and return TRUE (non-zero value) if there is an interrupt request.

6. System Configuration Data

utk_config_depend.h defines the setting such as micro T-Kernel System Configuration, the number of resources in micro T-Kernel, and the number of limitation values in micro T-Kernel.

Note that the maximum value of the setting range for each item is a logical maximum value, and that in reality there are limits imposed by the memory usage.

6.1. Setting value of utk_config_depend.h

```
/*
 *    utk_config_depend.h (at91)
 *    System Configuration Definition
 */

/* RAMINFO */
#define SYSTEMAREA_TOP       0x20000000
#define SYSTEMAREA_END       0x20200000
```

Area dynamically managed by micro T-Kernel memory management function
Specify the top address and end address in an external RAM.

```
/* User definition */
#define RI_USERAREA_TOP      0x20100000
```

This setting is not used.

```
#define RI_USERINIT          NULL
```

User definition initialization/termination program

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

Designate the system timer interrupt cycle (in millisecond).
This is the smallest resolution (accuracy)

```
#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
```

Designate the maximum number for each micro T-Kernel object.
Also in the designation of an upper limit, the number of objects actually used by the system must be taken into account.

```
#define CFN_MAX_REGDEV 8
```

Designate the number of the maximum devices that can be registered with "tk_def_dev()".
This sets the limit for the maximum number of physical devices.

```
#define CFN_MAX_OPNDEV 16
```

Designate the maximum number of times "tk_opn_dev()" can be called to open a device.
This sets the limit for the maximum number of device opens.

```
#define CFN_MAX_REQDEV 16
```

Designate the maximum number of requests by "tk_rea_dev()","tk_wri_dev()", "tk_srea_dev()", and "tk_swri_dev()".
This sets the maximum number of request IDs.

```
#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
```

Version information(tk_ref_ver)

```
#define CFN_REALMEMEND      ((VP)0x20200000)
```

Most significant address of RAM used in micro T-Kernel management area

```
/*
 * Stack size for each mode
 */
#define IRQ_STACK_SIZE      2048
#define FIQ_STACK_SIZE      16    /* 4 words */
#define ABT_STACK_SIZE      4     /* 1 word */
#define UND_STACK_SIZE      4     /* 1 word */
#define USR_STACK_SIZE      0     /* not used */
```

Designate the stack size (in bytes) for each exception mode.
USR_STACK_SIZE specifies the size of the stack used in both user mode and system mode.

```
#define EXCEPTION_STACK_TOP  INTERNAL_RAM_END
```

Initialization position of stack area.
Since these stacks for exception modes are commonly used only at entry and exit of handler, only small memory space is required for these.

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

1 : Among the static variables (BSS alignment), the variables that require no initialization are not cleared to zero in Kernel initialization processing. Since the processing for zero-clear execution is reduced, Kernel start-up time is shortened.
0 : All static variables without initialization value (BSS alignment) shall be cleared to zero.

```
/*
```

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

1 : The dynamic memory allocation function in Kernel is used.
0 : The dynamic memory allocation function in Kernel is not used. When creating the objects for Task, Message buffer, Fixed-size Memory Pool, and Variable-size Memory Pool, buffer shall be specified by application with TA_USERBUF attribute.

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

1 : The hook function of debugger support function is used.
However, the hook function can not be used if USE_DBGSPT is 0.
0 : The hook function of debugger support function is not used.

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

1 : Clean up processing of Kernel shall be executed after the termination of application.
0 : Clean up processing of Kernel shall not be executed after the termination of application. As for the system that doesn't return from usermain function, the consumption of ROM decreases by turning off this flag.

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

1 : High level language support routine is used at interruption
0 : High level language support routine is not used at interruption.
Jump table, and so on is not used, and the consumption of ROM/RAM decreases.

6.2. Setting value of sysinfo_depend.h

```
#define N_INTVEC       112
```

Upper limit of the interrupt definition number.

6.3. makerules

The following modes are selected by executing “make“ command with these arguments.

- mode (compile mode)

```
(empty)   : release version
Debug     : debug version

ex.  $ make mode=debug
```

- trap (trap mode)

```
(empty)   : not use trap
on        : use trap for system calls, dispatch, etc.

ex. $ make mode=debug trap=on
```