micro T-Kernel Implementation Specification

FM3 (Cortex-M3)

Version 2.00.00

December, 2014

Preface

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

The applicable board is T-Engine Appliance FM3 (Cortex-M3).

The applicable micro T-Kernel version is 2.00.00. The T-Kernel is compliant with the "micro T-Kernel Specification version 2.00.02".

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 4
  1.1. Hardware Specifications 4
  1.2. Protection Levels and Operating Modes 4
  1.3. Use of the Thumb-2 Instruction Set 4
2. Memory 5
  2.1. Overall Memory Map 5
  2.2. ROM Memory Map 6
  2.3. On-Chip SRAM Memory Map 6
  2.4. On-Chip SRAM Memory Map (When the μT-Kernel code is disposed to RAM) 7
  2.5. Stacks 7
3. Interrupts and Exceptions 8
  3.1. Interrupt Definition Numbers 8
  3.2. Software Interrupt Assignments 8
  3.3. Exception/Interrupt Handler 8
4. Initialization and Startup Processing 9
  4.1. micro T-Kernel Startup Procedure 9
  4.2. User initialization program 10
5. micro T-Kernel Implementation Definitions 11
  5.1. System State Detection 11
  5.2. Exceptions/Interrupts Used by micro T-Kernel 11
  5.3. System Call/Extended SVC Interface 12
  5.4. Interrupt Handlers 15
  5.5. Time Event Handlers 16
  5.6. Task implementation-dependent definitions 17
  5.7. Task registers 18
  5.8. System Call/Extended SVC Hook Routine 19
  5.9. Interrupt Controllers 20
6. System Configuration Data 21
  6.1. Setting value of utk_config_depend.h 21
  6.2. Setting value of sysinfo_depend.h 24
7. Limitation 25
  7.1. The limitation of specifying the “variable length size of memory block” 25
  7.2. The notes of the cycle handler startup 25
  7.3. Not support 25

1. CPU

1.1. Hardware Specifications

```
CPU :  MB9AF312K with Cortex-M3 core
       Spansion Inc.
ROM :  128 KB (Main Flash) + 32 KB (Work Flash)
RAM :  On-chip SRAM   16 KB
```

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

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

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

1.3. Use of the Thumb-2 Instruction Set

Since the target board's CPU is MB9AF312K (ARM Cortex-M3 Core) which only supports Thumb-2 instruction set, it is necessary to use the "Thumb-2 Instruction Set" when programming.

When a task or handler written using the Thumb instruction set (include Thumb-2) is designated, 1 is set to the least significant bit (LSB) of the address (When 0 is specified, it is considered as the ARM instruction set). Normally this is handled by the linker automatically and so programmers are not required to be aware of this.

2. Memory

2.1. Overall Memory Map

Overall system memory maps are shown below.

```
0x00000000 +--------------------------------+
           |      Flash (Main) (128KB)      |    0x00000000 - 0x0001ffff
0x00020000 +--------------------------------+
           |            Reserved            |    0x00020000 - 0x000fffff
0x00100000 +--------------------------------+
           |            Security            |    0x00100000 - 0x00100fff
0x00101000 +--------------------------------+
           |          CR trimming           |    0x00100000 - 0x00101fff
0x00102000 +--------------------------------+
           |            Reserved            |    0x00102000 - 0x1fffdfff
0x1fffe000 +--------------------------------+
           |          SRAM0 (8KB)           |    0x1fffe000 - 0x1fffffff
0x20000000 +--------------------------------+
           |          SRAM1 (8KB)           |    0x20000000 - 0x20001fff
0x20002000 +--------------------------------+
           |            Reserved            |    0x20002000 - 0x200bffff
0x200c0000 +--------------------------------+
           |       Work Flash (32KB)        |    0x200c0000 - 0x200c7fff
0x200c8000 +--------------------------------+
           |            Reserved            |    0x200c8000 - 0x200dffff
0x200e0000 +--------------------------------+
```

2.2. ROM Memory Map

Space of 128 KB is implemented in internal Flash ROM. The internal Flash ROM memory map is shown below.

```
0x00000000  +-----------------------------+
            |        vector table         |
            |- - - - - - - - - - - - - - -|
            |       µT-Kernel  code       |
            |- - - - - - - - - - - - - - -|
            |       Read Only data        |
            |- - - - - - - - - - - - - - -|
            |         (Not used)          |
0x00020000  +-----------------------------+
```

micro T-Kernel code shall be located at the lower byte of internal Flash ROM.

2.3. On-Chip SRAM Memory Map

Space of 16KB is implemented in on-chip SRAM. The on-chip SRAM memory map is shown below.

```
0x1FFFE000  +-----------------------------+    ← SYSTEMAREA_TOP
            |        vector table         |
            |           (copy)            |
            |- - - - - - - - - - - - - - -|
            |        DATA section         |
            |- - - - - - - - - - - - - - -|
            |       NoInit section        |
            |- - - - - - - - - - - - - - -|
            |         BSS section         |
            |- - - - - - - - - - - - - - -|
            |          µT-Kernel          |
            |       management area       |
            |- - - - - - - - - - - - - - -|
            |    initialize stack area    |
0x20002000  +-----------------------------+    ← SYSTEMAREA_END
```

NoInit： BBS section that is not zero-initialized

The vector table is located at the lower byte of on-chip RAM, and the vector table is copied from Flash to RAM during system starting.

The µT-Kernel management area is an area used by the memory management function of µT-Kernel, it is an area between the BSS section and the initialization stack area in principle.

Normally, all unused memory spaces are allocated to the micro T-Kernel management area, but this can be changed during system configuration.

2.4. On-Chip SRAM Memory Map (When the µT-Kernel code is disposed to RAM)

The µT-Kernel code is disposed to RAM when the RAM version load module (kernel-ram.sys) is used, while the Flash is not used. In this case, the memory map will be as follow.

```
0x1FFFE000 +-------------------------------+    ← SYSTEMAREA_TOP
           |         vector table          |
           |- - - - - - - - - - - - - - - -|
           |        µT-Kernel code         |
           |- - - - - - - - - - - - - - - -|
           |        Read Only data         |
           |- - - - - - - - - - - - - - - -|
           |         DATA section          |
           |- - - - - - - - - - - - - - - -|
           |        NoInit section         |
           |- - - - - - - - - - - - - - - -|
           |          BSS section          |
           |- - - - - - - - - - - - - - - -|
           |           µT-Kernel           |
           |        management area        |
           |- - - - - - - - - - - - - - - -|
           |     initialize stack area     |
0x20002000 +-------------------------------+    ← SYSTEMAREA_END
```

NoInit： BBS section that is not zero-initialized

2.5. Stacks

Micro T-Kernel has three kinds of stacks as follows.

(1) Initialization stack

It is a stack used from the hardware reset to the start-up of the initial task of µT-Kernel.

The size of the initialization stack can be set in utk_config_depend.h (EXC_STACK_SIZE).

(2) Temporary stack

It is a stack used during the task dispatch procedure.

The size of the temporary stack can be set in utk_config_depend.h (TMP_STACK_SIZE).

(3) System stack

The system stack exists in each task of µT-Kernel.

Since there is no concept of protection level in µT-Kernel, user stack and system stack are not separated which is different from T-Kernel.

Moreover, because the interrupt stack is not reserved in this system, the interrupt handler also uses the stack of the current task.

3. Interrupts and Exceptions

3.1. Interrupt Definition Numbers

Number 1-15 are used as system exception number and number 16-63 are used as IRQ (external interruption) number in MB9AF312K.

The IRQ numbers (correspond to the argument dintno of tk_def_int) that can be used by this system are 0-47. (The value which added 16 becomes the real exception number)

μT-Kernel uses 46 and 47 among the IRQ numbers as software interrupt. The nest vector interrupt controller (NVIC) of hardware (MB9AF312K) reserved the other IRQ numbers for the interrupt manage factor. Please refer to "FM3 family Peripheral manual" for more detail.

3.2. Software Interrupt Assignments

The numbers of the software interrupt are used as follow.

・ IRQ 46	software interrupt
・ IRQ 47	force dispatch

3.3. Exception/Interrupt Handler

The Cortex-M3 core can push the registers R0-R3, R12, LR, PSR and PC to the stack automatically at the entrance of the interruption, and pop them at the exit of the interruption.

Concretely, when the exception is generated, the following works are done by the processor.

(1) The contents of R0-R3, IP (R12), LR, PC and xPSR are pushed to the stack.
(2) The exception handler's start address is loaded from the vector table.
(3) The registers SP, PSR, LR and PC are updated.
- The exception number generated is set to the corresponding 9 bits (IPSR) of PSR.
- LR is updated to a special value for the interruption return operation.

When the exception/interrupt handler finishes and exits, EXC_RETURN (asm_depend.h) is used. The contents of eight registers shown in (1) are popped from the stack by the processor, and the SP is updated to the original value when the exception/interrupt was generated by the processor at the beginning.

Because the registers except those mentioned in (1) will not be saved automatically, it is necessary to save the registers destroyed by each handler. However when the interrupt handler is implement by C language, it is not necessary to worry about it because the C language will do the proper preserve by the help of compiler.

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.

[ROM version]

startup_rom.S

(1) Start initialization [Reset_Handler]
(2) System clock setting [init_clock_control]
(3) Copy the interrupt vector to RAM from ROM [rom_init ～ vector_done]
(4) Set the RAM address specified by (3) as the vector table address
(5) Data section initialization

icrt0.S

(1) BSS section initialization
(2) μT-Kernel manage area recalculate
(3) UART I/O [sio_init] initialization
(4) Setting the interrupt priority
(5) Enable force dispatch
(6) Enable software interrupt
(7) Call main function (sysinit_main.c)

[RAM version]

startup_ram.S

(1) Start initialization [Reset_Handler]
(2) System clock setting [init_clock_control]
(3) Set the address of the interruption vector

icrt0.S
~~The below is same as the ROM version~~

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 ac, UB **av )

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

Return code:  >   0       Startup usermain()
              <=  0       terminate the system
```

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

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

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

Exception Number 11	SVC interrupt
Exception Number 14	PendSV interrupt
Exception Number 15	System Tick Timer (SisTick) interrupt
Exception Number 62（IRQ 46）	Software interrupt
Exception Number 63（IRQ 47）	Force dispatch

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)

When the caller calls the system call via interface library, the method of using assembler language is same with the C language's method. You can also implement the same action like interface library, which is calling the SVC instruction directly. If that so, it is necessary to preserve the register according to the rule of C language. (Except the one preserved automatically when the SVC exception generating)

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 supervisor call instruction (SVC x).
A function code in negative value indicates a system call while the one at 0 or in a positive value indicates an extended SVC.
The following macro values are specified for x.

SVC_SYSCALL	system call(tk_xxx_yyy）
SVC_EXTENDED_SVC	extended SVC
SVC_DEBUG_SUPPORT	Debugger support function(td_xxx_yyy）

When the SVC exception generates, the registers above-mentioned will be preserved into the stack. Because the Cortex-M3 core can automatically push the registers R0-R3, R12, LR, PSR and PC to the stack at the entrance of the exception, and pop them at the exit of the interrupt, the interrupt handler can be described as C language.

(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 SVC instruction.

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     SVC_SYSCALL
#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 SVC instruction.

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.

```
callsvc
       stmfd   sp!, {r1-r3}         /* save register arguments to stack */
                                    /*and wrap in the packet */
       mov     ip, r0               /* ip = R0 = Function code */
       mov     r0, sp               /* R0 = arguments packet address */
       stmfd   sp!, {lr}            /*  Save lr */

#if USE_TRAP
       SVC     SVC_EXTENDED_SVC
#else
       bl      knl_call_entry
#endif

       ldmfd   sp!, {lr}            /*  Restore lr */
       add     sp, sp, #3*4         /* Discard arguments from stack */
       bx      lr
```

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 47 (However, please don't use number 46 and number 47).

Because the Cortex-M3 core can push the registers R0-R3, R12, LR, PSR, and PC automatically to the stack at the entrance of the interrupt (exception), and pop them at the exit of the interrupt. The interrupt handler can be described as C language.

The following rules are preconditions before the interrupt handler is implemented in this kernel.

・The interrupt handler is implemented as C language.

・The handler attribute is TA_HLNG.

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 interrupt handler definitions are as follows.

void inthdr(UINT dintno)

- 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.

The state of CPU when entering the interruption handler is as follows.

```
PRIMASK = 0      Enable interrupt
xPSR.T  = 1      Thumb mode
                 (The LSB of the handler start address is 1)
```

The return from the interruption handler is done through the following two steps.

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

(2) A return from an interrupt handler is at the end of the high-level executing EXC_RETURN.

When EXC_RETURN is executed, an exception active bit of interruption controller (NVIC) register is cleared. (The interruption of NVIC need not be cleared among interruption 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_HLNG
    | [TA_USERBUF] | [TA_DSNAME]
    | (TA_RNG0 || TA_RNG1 || TA_RNG2 || TA_RNG3)
```

(3) Task format

The format of task is as follows.

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

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

```
PRIMASK   = 0   Enable interrupt
xPSR.T    = 1   Thumb mode
                (When LSB of the task start address is 1)
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 */
} 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.

```
      +--------------+
sp -> |taskmode      |
      |R0 - R7       |
      |function code |
      |lr(2)         | The return address from knl_call_entry (cpu_support.S)
      |lr(1)         | The return address from interface library
      |R4            |
      +--------------+
```

The return address from a system call or an extended SVC is set in lr (2). However, since normally a call is made using an interface library, lr (2) indicates the interface library address.

The return address from the interface library is set in lr (1).

5.9. Interrupt Controllers

Since the interrupt controller control is highly hardware-dependent, it is not specified in micro T-Kernel. However, the SK-FM3-48PMC-USBSTICK is implemented as follows.

(1) Interrupt Definition Numbers

The interrupt numbers among 0 to 47 are used as following.

- 46 Software interrupt
- 47 force dispatch

(2) Enable interrupt

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

Enable the interrupt designated in intvec.

(3) Disable interrupt

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

Disable the interrupt designated in intvec.

(4) 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 (FM3)
 *    System Configuration Definition
 */

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

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

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

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        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
```

Version information(tk_ref_ver)

```
#define CFN_REALMEMEND       ((VP) 0x20002000)
```

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

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

Initial task priority

```
/*
 * 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.

```
/*
 * Stack size for each mode
 */
#define EXC_STACK_SIZE        0x200

#define TMP_STACK_SIZE        0x80

#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  SYSTEMAREA_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.

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

Initialization position of temporary stack area.

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

Initialization position of application stack area.
This setting is not used.

```
/*
 * 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(It cann’t be changed)

6.2. Setting value of sysinfo_depend.h

```
#define N_INTVEC             48
```

Upper limit of the interrupt definition number.

7. Limitation

The following are limitations and notes of the micro T-Kernel version

7.1. The limitation of specifying the "variable length size of memory block"

When you call the tk_get_mpl (get variable length memory block), the parameter blksz (size of the memory block) should be set as the following values:

blksz = 0x7FFFFFF9 ～ 0x7FFFFFFF

7.2. The notes of the cycle handler startup

When the parameter of the tk_cre_cyc is specified as 0, some status such as the interrupt state at start of the handler that executes the first time are different from the status at the start of the handler that executes the second time or more.

The setting in μT-Kernel is the same as T-Kernel.

[reference] (T-Kernel 2.0 specification the abstract of tk_cre_cyc's supplementation)

When 0 is specified for cycphs, the handler that executes the first time starts immediately after the system call. But based on the implement, sometimes cycle handler doesn't start immediately after the system call, the cycle handler that executes the first time might execute while processing this system call. In this case, sometimes the status such as the interrupt state at start of the handler that executes the first time are different from the status at the start of the handler that executes the second time or more. In addition, when 0 is specified for cychps, the cycle handler that executes the first time does not wait for the timer interrupt. In other words, it executes without any relation to the timer interrupt. It is different from the usual cycle handler that executes the second time or more and the cycle handler when cycphs is not 0.

7.3. Not support

The following functions are not support now

・ TA_ASM attribute