micro T-Kernel
Implementation Specification

H8S/2212

Version 1.01.02
April, 2013

Preface

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

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.

See also the micro T-Kernel specifications, as well as the separate hardware specifications, for the target board and the CPU, etc.

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  : H8S/2212 (HD64F2212)
        Renesas Technology Corp.
ROM  : 128 KB (On-chip FlashROM)
RAM  :  12 KB (On-chip SRAM)
```

1.2. Protection Levels and Operating Modes

Since this system operates in single CPU operating mode, there is no switch-over of the protection level. Even if any of the protection levels is specified, it shall be treated as protection level 0.

2. Memory

2.1. Overall Memory Map

An overall system memory map is shown below.

```
0x00000000 +---------------------------+
           |   On-chip ROM     (128KB) | 0x00000000-0x0001ffff
0x00020000 +---------------------------+
           |         (unused)          |
0x00c00000 +---------------------------+
           |   USB registers           | 0x00c00000-0x00dfffff
0x00e00000 +---------------------------+
           |         (unused)          |
0x00fee800 +---------------------------+
           |   Reserved                | 0x00fee800-0x00ffbfff
0x00ffc000 +---------------------------+
           |   On-chip RAM  (12KB-64B) | 0x00ffc000-0x00ffefbf
0x00ffefc0 +---------------------------+
           |         (unused)          |
0x00fff800 +---------------------------+
           |   Internal I/O registers  | 0x00fff800-0x00ffffbf
0x00ffffc0 +---------------------------+
           |   On-chip RAM       (64B) | 0x00ffffc0-0xffffffff
0xffffffff +---------------------------+
```

2.2. ROM Memory Map

Space of 128KB is implemented in on-chip ROM. The ROM memory map is shown below.

```
0x00000000 +---------------------------+
           |Interrupt Exception Handling|
           |       Vector Table        | 0x00000000-0x000001ff
0x00000200 +---------------------------+
           |    micro T-Kernel code    |
           |                           |
           |- - - - - - - - - - - - - -|
           |         (unused)          |
0x00020000 +---------------------------+
```

Vector table and micro T-Kernel code shall be located in the on-chip ROM.

2.3. RAM Memory Map

Space of 12 KB is implemented in on-chip RAM. The RAM memory map is shown below.

```
0x00ffc000 +-----------------------------+
           |    Data section             |
           |- - - - - - - - - - - - - - -|
           |    BSS section (NoInit)     |
           |- - - - - - - - - - - - - - -|
           |    BSS section              |
           |- - - - - - - - - - - - - - -| ← SYSTEMAREA_TOP
           |                             |
           |    micro T-Kernel           |
           |    management area          |
           |                             |
0x00ffef00 +-----------------------------+ ← SYSTEMAREA_END
           |    Interrupt stack (192B)   |
0x00ffefc0 +-----------------------------+ ← sp initial value

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 on-chip RAM.

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.

2.4. 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) Interrupt stack

This is the stack used in an interrupt handler, and the stack area independent of system stack shall be allocated to it.

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

3. Interrupts and Exceptions

3.1. Interrupt Definition Numbers

The immediate values (0-127) of vector number shall be used by dintno, the interrupt definition numbers defined with tk_def_int().

Following is the Interrupt Exception Handling Vector Table.

```
+------------+---------------+-------------------------+
   Vector No    Vector Address     Interrupt Sources
+------------+---------------+-------------------------+

     0          0x0000            Power-on reset
     1          0x0004            Manual reset
     2          0x0008            Reserved for system use
     3          0x000C            Reserved for system use
     4          0x0010            Reserved for system use
     5          0x0014            Trace
     6          0x0018            Direct transitions
     7          0x001C            External interrupt NMI
     8          0x0020            Trap instruction (#0)
     9          0x0024            Trap instruction (#1)
     10         0x0028            Trap instruction (#2)
     11         0x002C            Trap instruction (#3)
     12         0x0030            Reserved for system use
     13         0x0034            Reserved for system use
     14         0x0038            Reserved for system use
     15         0x003C            Reserved for system use

     16         0x0040            External interrupt IRQ0
     17         0x0044            External interrupt IRQ1
     18         0x0048            External interrupt IRQ2
     19         0x004C            External interrupt IRQ3
     20         0x0050            External interrupt IRQ4
     21         0x0054            RTC interrupt IRQ5
     22         0x0058            USB interrupt IRQ6
     23         0x005C            External interrupt IRQ7
     24         0x0060            -
     25         0x0064            Watchdog Timer
     26         0x0068            -
     27         0x006C            -
     28         0x0070            A/D ADI
     29         0x0074            -
     30         0x0078            -
     31         0x007C            -

     32         0x0080            TPU channel 0 TGI0A
     33         0x0084            TPU channel 0 TGI0B
     34         0x0088            TPU channel 0 TGI0C
     35         0x008C            TPU channel 0 TGI0D
     36         0x0090            TPU channel 0 TCI0V
     37         0x0094            -
     38         0x0098            -
     39         0x009C            -
     40         0x00A0            TPU channel 1 TGI1A
     41         0x00A4            TPU channel 1 TGI1B
     42         0x00A8            TPU channel 1 TCI1V
```

43	0x00AC	TPU channel 1 TCI1U
44	0x00B0	TPU channel 2 TGI2A
45	0x00B4	TPU channel 2 TGI2B
46	0x00B8	TPU channel 2 TCI2V
47	0x00BC	TPU channel 2 TCI2U
48	0x00C0	-
49	0x00C4	-
50	0x00C8	-
51	0x00CC	-
52	0x00D0	-
53	0x00D4	-
54	0x00D8	-
55	0x00DC	-
56	0x00E0	-
57	0x00E4	-
58	0x00E8	-
59	0x00EC	-
60	0x00F0	-
61	0x00F4	-
62	0x00F8	-
63	0x00FC	-
64	0x0100	-
65	0x0104	-
66	0x0108	-
67	0x010C	-
68	0x0110	-
69	0x0114	-
70	0x0118	-
71	0x011C	-
72	0x0120	DMAC DEND0A
73	0x0124	DMAC DEND0B
74	0x0128	DMAC DEND1A
75	0x012C	DMAC DEND1B
76	0x0130	-
77	0x0134	-
78	0x0138	-
79	0x013C	-
80	0x0140	SCI channel 0 ERI0
81	0x0144	SCI channel 0 RXI0
82	0x0148	SCI channel 0 TXI0
83	0x014C	SCI channel 0 TEI0
84	0x0150	-
85	0x0154	-
86	0x0158	-
87	0x015C	-
88	0x0160	SCI channel 2 ERI2
89	0x0164	SCI channel 2 RXI2
90	0x0168	SCI channel 2 TXI2
91	0x016C	SCI channel 2 TEI2
92	0x0170	-
93	0x0174	-
94	0x0178	-
95	0x017C	-
96	0x0180	-
97	0x0184	-

```
      98          0x0188           -
      99          0x018C           -
      100         0x0190           -
      101         0x0194           -
      102         0x0198           -
      103         0x019C           -
      104         0x01A0           USB EXIRQ0
      105         0x01A4           USB EXIRQ1
      106         0x01A8           -
      107         0x01AC           -
      108         0x01B0           -
      109         0x01B4           -
      110         0x01B8           -
      111         0x01BC           -

      112         0x01C0           -
      113         0x01C4           -
      114         0x01C8           -
      115         0x01CC           -
      116         0x01D0           -
      117         0x01D4           -
      118         0x01D8           -
      119         0x01DC           -
      120         0x01E0           -
      121         0x01E4           -
      122         0x01E8           -
      123         0x01EC           -
      124         0x01F0           -
      125         0x01F4           -
      126         0x01F8           -
      127         0x01FC           -

+-------------+----------------+--------------------------+
```

3.2. TRAPA exception assignments

The TRAPA instruction uses trapa 0 to trapa 3, and is assigned to the definition numbers 8 to 11. Each TRAPA instruction is used as follows.

```
trapa 0    micro T-Kernel system call/extended SVC
trapa 1    "tk_ret_int()" system call
trapa 2    Task dispatcher call
trapa 3    Debugger support function
```

3.3. Interrupt handler

If an interrupt handler is defined, in the vector.S, the address of the handler shall be defined to the corresponding vector number. The vector number defined to vector.S can be used with tk_def_int.

If an interrupt occurs, it directly jumps to the set address of an interrupt handler. Therefore, the context shall be saved in the processing of the interrupt handler.

An entry routine is defined by using INT_ENTRY macro. When "INT_ENTRY vecno" is written in an assembler file, the entry routine of the interrupt handler named knl_inthdr_entryN(N is an interrupt vector number)is generated. And if knl_inthdr_entryN is defined to vector.S, the vector number can be used.

This entry routine executes the following processing.

・Save er0 and er1 to the stack.

・Save an interrupt vector number in er0.

・Jump to the handler set in tk_def_int.

Since tk_ret_int is premised on this processing, er0 and er1 need to be saved in stack if handler is defined without using INT_ENTRY macro.

In order to realize the delayed dispatching with tk_ret_int, interrupt nesting count (knl_int_nest) is incremented within a high-level language support routine. If a high-level language support routine is not used, knl_int_nest needs to be incremented.

If 1 is set to USE_FULL_VECTOR of utk_config_depend.h, INT_ENTRY macro is automatically used to all vectors and all interrupts can be treated with tk_def_int.

By default setting, 1 shall be set to USE_FULL_VECTOR.

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) Set the stack pointer [start:]

(2) Initialize CCR [flashrom_init:]

(3) Initialize EXR [flashrom_init:]

(4) Set the initial value of data section (ROM->RAM) [data_loop:]

(5) Clear BSS section to 0 [bss_loop:]

(6) Calculate the range of micro T-Kernel management area [bss_done:]

(7) 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 code 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

```
trapa 0         micro T-Kernel system calls/extended SVC
trapa 1         "tk_ret_int()" system call
trapa 2         Task dispatcher call
trapa 3         Debugger support functions

dintno 32 Programmable timer A(TGI0A)
```

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 TRAPA 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 R0 register and the system call is invoked by TRAPA #1. 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 TRAPA #3 is used for Debugger Support Functions service calls.

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

```
R0 to R2  Temporary registers
R3 to R6  Permanent registers
R7 = sp   Stack pointer
EXR       Unused

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

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

(1) System call interface

Parameters of up to third are set to registers, and the ones of fourth or more are saved onto the stack. A system call is invoked by TRAPA #1(TRAPA #TRAP_SVC).

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

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

//                    stack state
//   High Address  +---------------+
//                 | p5            |
//                 | p4            |
//                 | SPC(24bit)    | saved by I/F call
//          SP =>  | SCCR(8bit)    |
//    Low Address  +---------------+
//
//                     er0 = p1
//                     er1 = p2
//                     er2 = p3
Csym(tk_xxx_yyy):
      mov.w r0, @-er7
      mov.w function code, r0
#if USE_TRAP
      trapa #TRAP_SVC
#else
      jsr   Csym(knl_call_entry)
#endif
      inc.l #2, er7
      rts
#endif
```

(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 er1 register. An extended SVC call is invoked by TRAPA #1(TRAPA #TRAP_SVC).

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.

An example of extended SVC interface implementation is shown below.

```
        W zxxx_yyy(p1, p2, p3, p4, p5, p6)

Csym(${func}):
         mov.l   er1, @-er7       // Save register arguments on stack
         mov.l   er0, @-er7
         mov.l   er7, er1         // er1 = arguments packet address
         mov.l   er2, @-er7
         mov.w   @zxxx_yyy, r0
         trapa   #TRAP_SVC
         add.l   #3*4, er7
         rts
```

(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 TRAPA #4(TRAPA TRAP_DEBUG).

5.4. The stack when system call is invoked

(1) C language I/F(func(arg1, arg2, ...))

```
         High Address  +---------------+
                       | ...           |
                       | 5th arg       |
                       | 4th arg       |
                       | SPC(24bit)    | saved by I/F call
                       | SCCR(8bit)    |
         Low Address   +---------------+

     er0 = 1st arg
     er1 = 2nd arg
     er2 = 3rd arg
```

(2) knl_call_entry top(Immediately after trapa #TRAP_SVC is executed)

```
         High Address  +---------------+
                       | ...           |
                       | 5th arg       |
                       | 4th arg       |
                       | SPC(24bit)    | saved by I/F call
                       | SCCR(8bit)    |
                       | r0(16bit)     |
                       | SPC(24bit)    | saved by trapa
                SP =>  | SCCR(8bit)    |
          Low Address  +---------------+

           r0 = fncd
           e0 = High byte of 1st arg
          r0 which was saved to stack = Low byte of 1st arg
          er1 = 2nd arg
          er2 = 3rd arg
```

(3) Immediately after bpl l_esvc_function is executed

```
High Address  +---------------+
              | ...           |
              | 5th arg       |
              | 4th arg       |
              | SPC(24bit)    | saved by I/F call
              | SCCR(8bit)    |
              | r0(16bit)     |
              | SPC(24bit)    | saved by trapa
              | SCCR(8bit)    |
              | er4           |
              | er5           |
  er5, SP =>  | er6           |
Low Address   +---------------+

   r0 = fncd
   e0 = High byte of 1st arg
  r0 which was saved to stack = Low byte of 1st arg
  er1 = 2nd arg
  er2 = 3rd arg
   r4 = fncd
```

(4) Immediately before system call is invoked

```
High Address  +---------------+
              | ...           |
              | 5th arg       |
              | 4th arg       |
              | SPC(24bit)    | saved by I/F call
              | SCCR(8bit)    |
              | r0(16bit)     |
              | SPC(24bit)    | saved by trapa
              | SCCR(8bit)    |
              | er4           |
              | er5           |
      er5 =>  | er6           |
              | (5th arg)     |
              | (4th arg)     |
 Low Address  +---------------+

  er0 = 1st arg
  er1 = 2nd arg
  er2 = 3rd arg
```

5.5. Stack when the extended SVC is invoked

(1) C language I/F(func(arg1, and arg2, ...))

```
High Address  +---------------+
              | ...           |
              | 5th arg       |
              | 4th arg       |
              | SPC(24bit)    | saved by I/F call
              | SCCR(8bit)    |
 Low Address  +---------------+

 er0 = 1st arg
 er1 = 2nd arg
 er2 = 3rd arg
```

(2) knl_call_entry top(immediately after trapa #TRAP_SVC is executed)

```
High Address  +---------------+
              | ...           |
              | 5th arg       |
              | 4th arg       |
              | SPC(24bit)    | saved by I/F call
              | SCCR(8bit)    |
              | 2nd arg       |
      er1 =>  | 1st arg       |
              | 3rd arg       |
              | SPC(24bit)    | saved by trapa
       SP =>  | SCCR(8bit)    |
 Low Address  +---------------+

  r0 = fncd
 er1 = pk_para
```

(3) l_esvc_function top

```
High Address  +---------------+
              | ...           |
              | 5th arg       |
              | 4th arg       |
              | SPC(24bit)    | saved by I/F call
              | SCCR(8bit)    |
              | 2nd arg       |
      er1 =>  | 1st arg       |
              | 3rd arg       |
              | SPC(24bit)    | saved by trapa
              | SCCR(8bit)    |
              | er4           |
              | er5           |
  er5, SP =>  | er6           |
 Low Address  +---------------+

 r0  = fncd
 er1 = pk_para
```

(4) knl_svc_ientry top

```
High Address  +---------------+
              | ...           |
              | 5th arg       |
              | 4th arg       |
              | 3rd arg       |
              | 2nd arg       |
      er0 =>  | 1st arg       |
              | 3rd arg       |
              | SPC(24bit)    | saved by trapa
              | SCCR(8bit)    |
              | er4           |
              | er5           |
  er5, SP =>  | er6           |
 Low Address  +---------------+

 er0 = pk_para
  r1 = fncd
 er4 = ret - addr (saved by I/F call)
```

5.6. Stack when an interrupt is raised

- Stack when hardware interrupt is raised

```
High Address  +---------------+
              | SPC(24bit)    | saved by Interrupt Controller
       SP =>  | SCCR(8bit)    |
 Low Address  +---------------+

 er0 = 1st arg
 er1 = 2nd arg
 er2 = 3rd arg
```

- Stack of the interrupt handler entry when tk_def_int is used.
  (called from knl_inthdr_entryN(N is an interrupt vector number))

```
High Address  +---------------+
              | SPC(24bit)    | saved by Interrupt Controller
              | SCCR(8bit)    |
              | er0           |
       SP =>  | er1           |
 Low Address  +---------------+
```

er0 = interrupt vector number

5.7. 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.

```
CCR.I = 0       Interrupts enabled
er0 = stacd     Task start parameters
er1 = exinf     Task extended information
er7(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.

5.8. 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 er[7];        /* General-purpose registers er0-er6 */
} T_REGS;
```

When setting registers to a task in DORMANT state, the task start parameters and the extended information are set in er0 and er1 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 PC */
    VB ccr;          /* condition code register CCR */
} T_EIT;
```

(3) Control registers

```
typedef struct t_cregs {
    VP ssp;          /* system stack pointer er7 */
} T_CREGS;
```

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 (h8s2212)
 *    System Configuration Definition
 */

/* ROMINFO */
#define SYSTEMAREA_TOP       0x00ffc000
#define SYSTEMAREA_END       0x00ffef00
```

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

```
/* User definition */
#define RI_USERAREA_TOP      0x00ffefc0
```

This setting is not used.

```
#define RI_USERINIT          NULL
```

User definition initialization/termination program

```
/* Stacks */
#define RI_INTSTACK          0x00ffefc0
```

Initial position of an interrupt stack

```
/* 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)0x00ffefc0)
```

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

```
/*
 * Use non-clear 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 full interrupt vector
 */
#define USE_FULL_VECTOR      (1)
```

1 : Prepare the interrupt initialization processing(save of partial register or setting of an interrupt number)for the all interrupt vectors.
0 : Prepare the initial processing only for the defined interrupts. the consumption of ROM decreases.

```
/*
 * 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.

```
/*
 * Use dynamic interrupt handler definition
 */
#define USE_DYNAMIC_INTHDR   (1)
```

1 : Change an interrupt handler with tk_def_int.
0 : Do not change an interrupt handler with tk_def_int.
Jump table is not used, and the consumption of ROM is decreased.

6.2. 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
```