

micro T-Kernel Implementation Specification M32C (M32C87)

Real-time Operating System for M32C Family



Rev.1.00
Mar. 28, 2008

Renesas Technology
www.renesas.com

Notes regarding these materials

1. This document is provided for reference purposes only so that Renesas customers may select the appropriate Renesas products for their use. Renesas neither makes warranties or representations with respect to the accuracy or completeness of the information contained in this document nor grants any license to any intellectual property rights or any other rights of Renesas or any third party with respect to the information in this document.
2. Renesas shall have no liability for damages or infringement of any intellectual property or other rights arising out of the use of any information in this document, including, but not limited to, product data, diagrams, charts, programs, algorithms, and application circuit examples.
3. You should not use the products or the technology described in this document for the purpose of military applications such as the development of weapons of mass destruction or for the purpose of any other military use. When exporting the products or technology described herein, you should follow the applicable export control laws and regulations, and procedures required by such laws and regulations.
4. All information included in this document such as product data, diagrams, charts, programs, algorithms, and application circuit examples, is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas products listed in this document, please confirm the latest product information with a Renesas sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas such as that disclosed through our website. (<http://www.renesas.com>)
5. Renesas has used reasonable care in compiling the information included in this document, but Renesas assumes no liability whatsoever for any damages incurred as a result of errors or omissions in the information included in this document.
6. When using or otherwise relying on the information in this document, you should evaluate the information in light of the total system before deciding about the applicability of such information to the intended application. Renesas makes no representations, warranties or guaranties regarding the suitability of its products for any particular application and specifically disclaims any liability arising out of the application and use of the information in this document or Renesas products.
7. With the exception of products specified by Renesas as suitable for automobile applications, Renesas products are not designed, manufactured or tested for applications or otherwise in systems the failure or malfunction of which may cause a direct threat to human life or create a risk of human injury or which require especially high quality and reliability such as safety systems, or equipment or systems for transportation and traffic, healthcare, combustion control, aerospace and aeronautics, nuclear power, or undersea communication transmission. If you are considering the use of our products for such purposes, please contact a Renesas sales office beforehand. Renesas shall have no liability for damages arising out of the uses set forth above.
8. Notwithstanding the preceding paragraph, you should not use Renesas products for the purposes listed below:
 - (1) artificial life support devices or systems
 - (2) surgical implantations
 - (3) healthcare intervention (e.g., excision, administration of medication, etc.)
 - (4) any other purposes that pose a direct threat to human lifeRenesas shall have no liability for damages arising out of the uses set forth in the above and purchasers who elect to use Renesas products in any of the foregoing applications shall indemnify and hold harmless Renesas Technology Corp., its affiliated companies and their officers, directors, and employees against any and all damages arising out of such applications.
9. You should use the products described herein within the range specified by Renesas, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas shall have no liability for malfunctions or damages arising out of the use of Renesas products beyond such specified ranges.
10. Although Renesas endeavors to improve the quality and reliability of its products, IC products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Please be sure to implement safety measures to guard against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other applicable measures. Among others, since the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
11. In case Renesas products listed in this document are detached from the products to which the Renesas products are attached or affixed, the risk of accident such as swallowing by infants and small children is very high. You should implement safety measures so that Renesas products may not be easily detached from your products. Renesas shall have no liability for damages arising out of such detachment.
12. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written approval from Renesas.
13. Please contact a Renesas sales office if you have any questions regarding the information contained in this document, Renesas semiconductor products, or if you have any other inquiries.

This Product uses the Source Code of μ T-Kernel under μ T-License granted by the T-Engine Forum (www.t-engine.org)

Preface

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

The applicable board is Renesas Starter Kit for M32C/87. Please refer to this website for more information on Renesas Starter Kit: <http://www.renesas.com/rsk>.

The applicable micro T-Kernel version is 1.01.00.

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

Please 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.

Table of Content

| | | |
|----------|---|---------------|
| 1 | CPU..... | - 1 - |
| 1.1 | HARDWARE SPECIFICATIONS..... | - 1 - |
| 1.2 | PROTECTION LEVEL AND OPERATION MODES..... | - 1 - |
| 1.3 | ENDIAN..... | - 1 - |
| 2 | MEMORY..... | - 2 - |
| 2.1 | OVERALL MEMORY MAP..... | - 2 - |
| 2.2 | INTERNAL ROM MEMORY MAP..... | - 2 - |
| 2.3 | INTERNAL RAM MEMORY MAP..... | - 4 - |
| 2.4 | STACK..... | - 5 - |
| 3 | INTERRUPTS AND EXCEPTIONS..... | - 6 - |
| 3.1 | INTERRUPT VECTOR TABLE IN FIXED VECTOR TABLE..... | - 6 - |
| 3.2 | INTERRUPTS DEFINITION NUMBER..... | - 6 - |
| 3.3 | INT EXCEPTION ASSIGNMENTS..... | - 8 - |
| 3.4 | INTERRUPT HANDLER..... | - 8 - |
| 3.5 | KERNEL INTERRUPT MASK LEVEL..... | - 9 - |
| 4 | INITIALIZATION AND STARTUP PROCESSING..... | - 12 - |
| 4.1 | MICRO T-KERNEL STARTUP PROCEDURE..... | - 12 - |
| 4.2 | USER INITIALIZATION PROGRAM..... | - 12 - |
| 5 | MICRO T-KERNEL IMPLEMENTATION DEFINITIONS..... | - 13 - |
| 5.1 | SYSTEM STATE DETECTION..... | - 13 - |
| 5.2 | EXCEPTION/INTERRUPT USED IN MICRO T-KERNEL..... | - 13 - |
| 5.3 | SYSTEM CALL/EXTENDED SVC INTERFACE..... | - 13 - |
| 5.4 | INTERRUPT HANDLERS..... | - 17 - |
| 5.5 | TIMER EVENT HANDLERS..... | - 18 - |
| 5.6 | STACK WHEN SYSTEM CALL IS INVOKED..... | - 18 - |
| 5.7 | STACK WHEN THE EXTENDED SVC IS INVOKED..... | - 20 - |
| 5.8 | STACK WHEN AN INTERRUPT IS RAISED..... | - 22 - |
| 5.9 | TASK IMPLEMENTATION-DEPENDENT DEFINITIONS..... | - 23 - |
| 5.10 | TASK REGISTERS..... | - 23 - |
| 5.11 | SYSTEM CALL/EXTENDED SVC HOOK ROUTINE..... | - 24 - |
| 5.12 | OTHER IMPLEMENTATION-DEPENDENT DEFINITIONS..... | - 25 - |
| 6 | SYSTEM CONFIGURATION DATA..... | - 26 - |
| 6.1 | SETTING VALUE OF UTK_CONFIG_DEPEND.H..... | - 26 - |
| 6.2 | BUILDING FLAG..... | - 29 - |
| 7 | RESOURCE USED BY MICRO T-KERNEL..... | - 30 - |
| 7.3 | KERNEL OBJECTS..... | - 30 - |

1 CPU

1.1 Hardware Specifications

CPU: M32C/87 (M30879FLGP)

Renesas Technology Corp.

ROM: 1024 KB (On-chip FlashROM) and additional 4 KB On-chip Data Flash

RAM: 48 KB (On-chip SRAM)

1.2 Protection level and operation modes

The processor runs at Single Chip Mode (PM0.PM00=PM0.PM01=0) on this platform.

Only the internal areas (SFR, internal RAM, and internal ROM) are accessible. The ports (P0-P10) can be used either as programmable I/O ports or as I/O ports for internal peripheral functions.

Since this system operates in Single Chip Mode, there is no switch-over of the protection level. Even if any of the protection level is specified, it shall be treated as protection level 0.

1.3 Endian

M32C family microcomputer uses little-endian format for both bit and byte.

2 Memory

2.1 Overall Memory Map

Memory map of overall system is shown below.

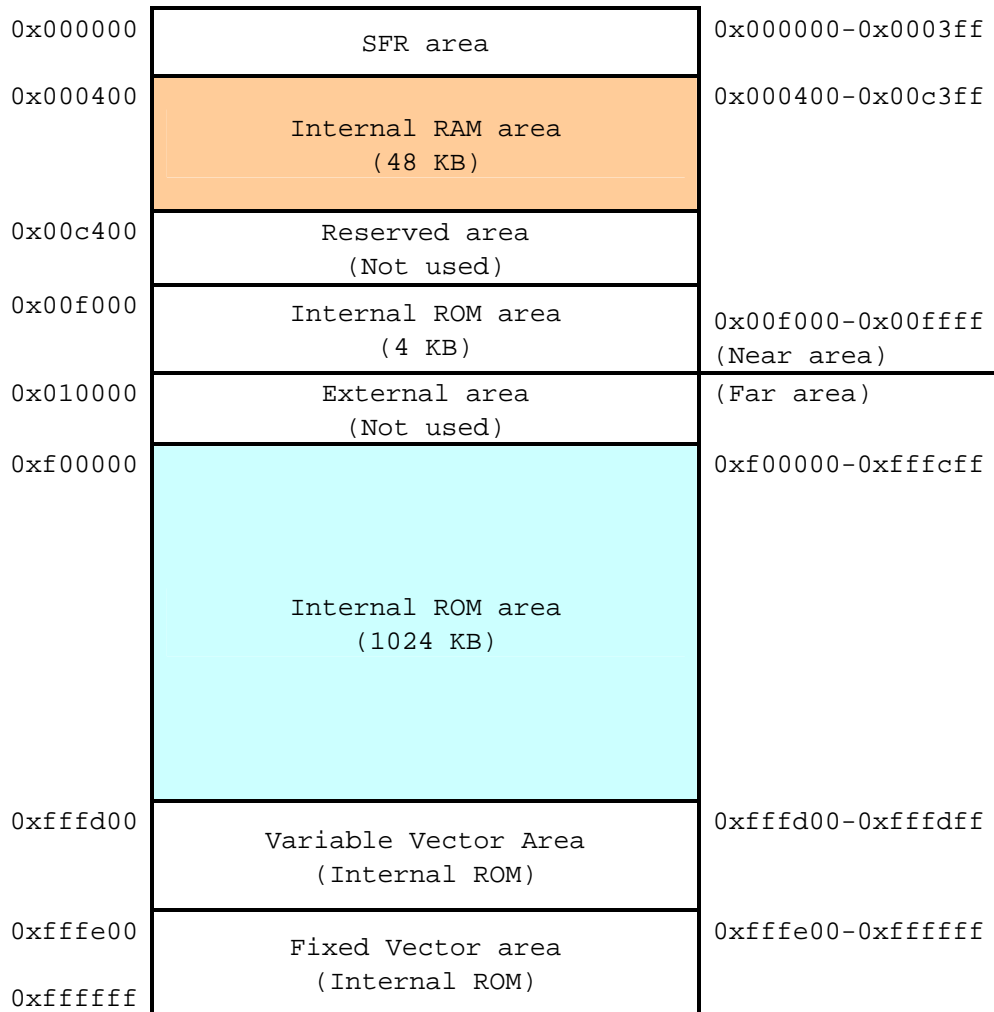


Figure 2-1 Overall Memory Map

The INTB register value is initialized to the start address of Variable Vector Area, namely, 0xffffd00.

2.2 Internal ROM Memory Map

There are two internal ROM areas in M32C/87: Near ROM area and Far ROM area.

Near ROM area is a 4 KB (0x00f000-0x00ffff) area.

Far ROM area is a 1024 KB (0xf00000-0xffffffff) area.

Interrupt vector table and micro T-Kernel code shall be located in the Far ROM area.

Memory map of Near ROM area is shown below.

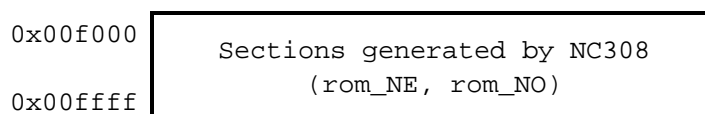


Figure 2-2 Memory Map of Near ROM area

Memory map of Far ROM area is shown below.

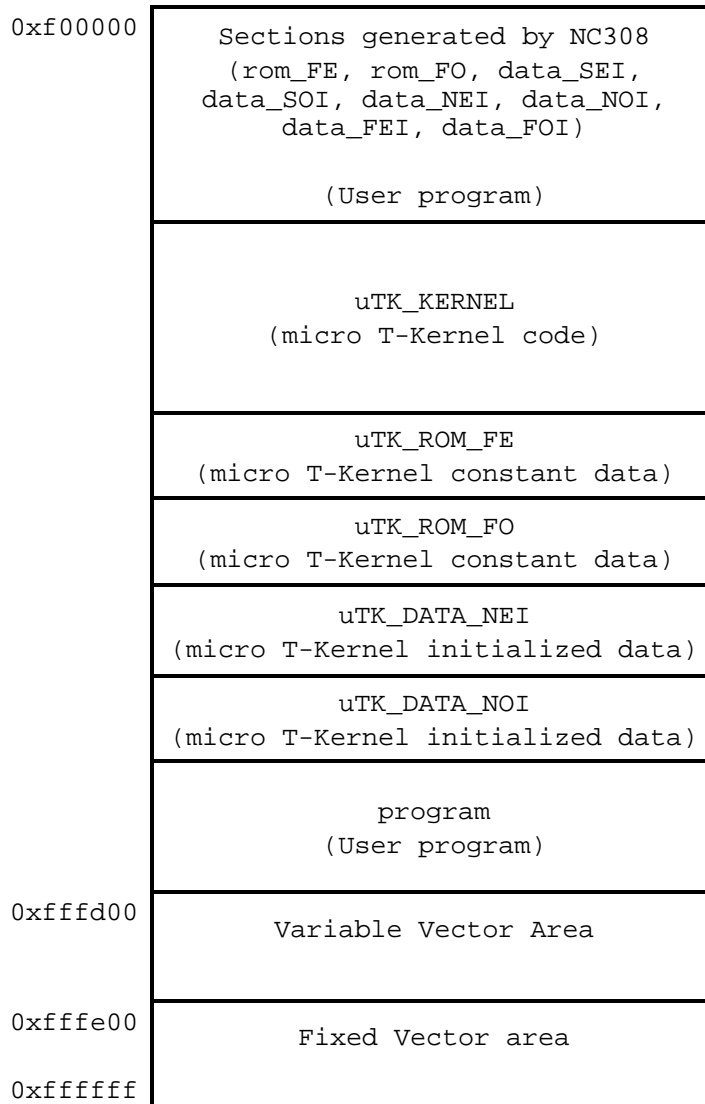


Figure 2-3 Memory Map of Far ROM area

The following explains the additional sections defined here:

- uTK_KERNEL
This section stores the micro T-Kernel kernel code.
- uTK_ROM_FE and uTK_ROM_FO
These two sections store the constant data used by the micro T-Kernel.
- uTK_DATA_NEI and uTK_DATA_NOI
These two sections store the initialized data used by the micro T-Kernel.
- program
This section stores the user programs (not used by the micro T-Kernel).

Memory map for fixed vector area in Internal ROM area is shown below.

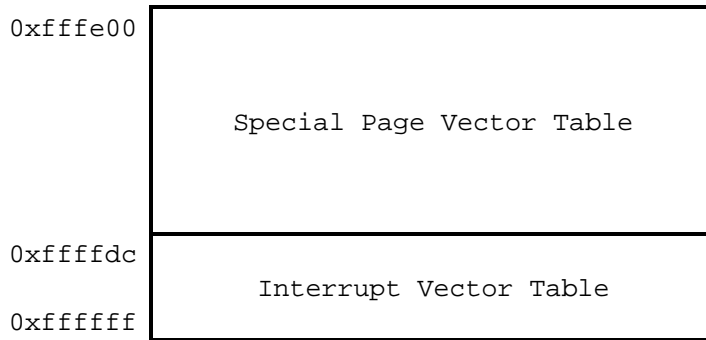
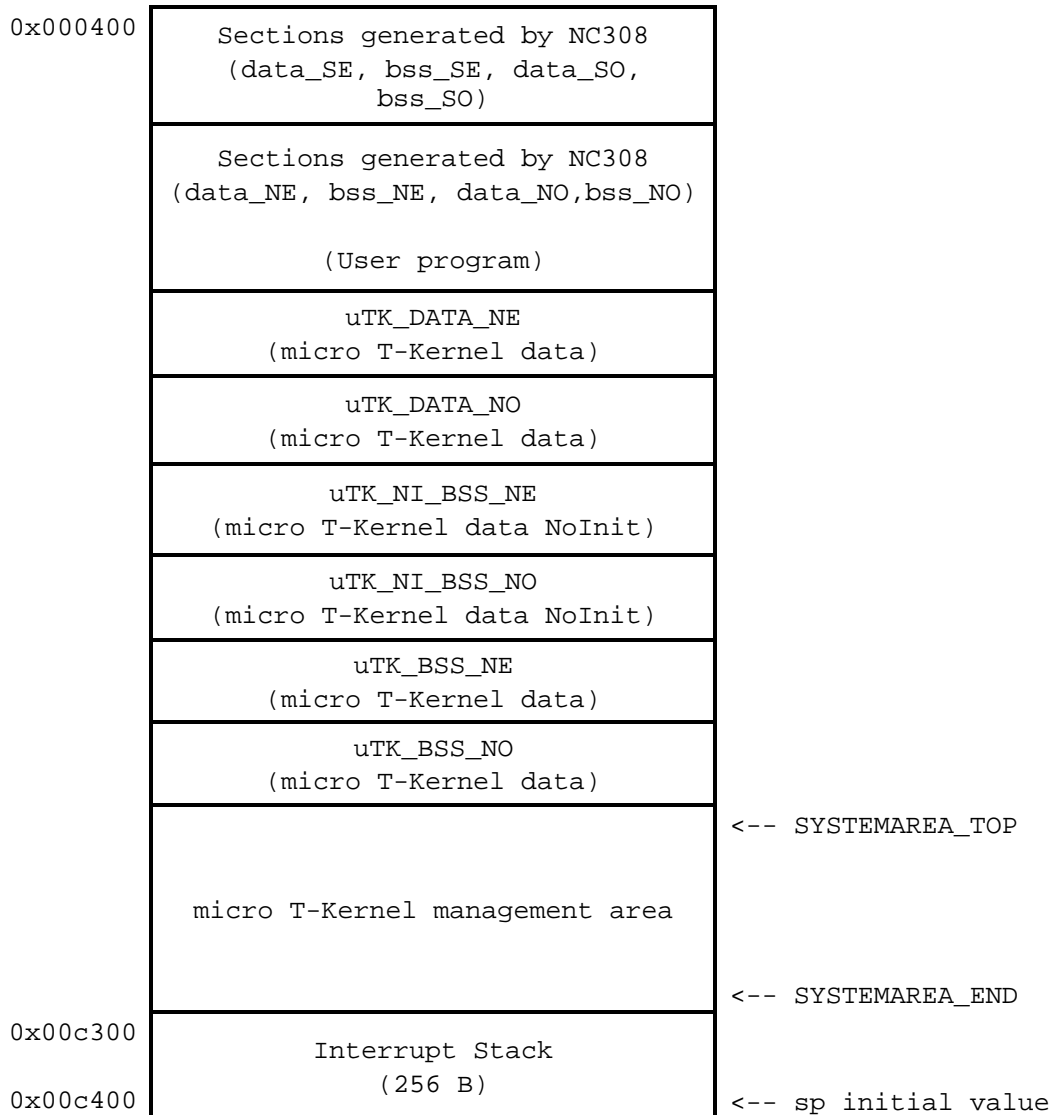


Figure 2-4 Memory Map of fixed vector area

2.3 Internal RAM Memory Map

In internal RAM, an area of 48 KB (0x000400-0x00c3ff) is implemented. The memory map of internal RAM area is shown below.



The data section and BSS section are separate for micro T-Kernel program and user program. They are located in ascending order from the lower byte of the on-chip RAM.

The following explains the additional sections defined for micro T-Kernel:

- `uTK_DATA_NE` and `uTK_DATA_NO`
These sections are where the micro T-Kernel initialized data is stored.
- `uTK_BSS_NE` and `uTK_BSS_NO`
These sections are where the micro T-Kernel uninitialized data is stored. The data in this section must be cleared during kernel initialization.
- `uTK_NI_BSS_NE` and `uTK_NI_BSS_NO`
These sections are where the micro T-Kernel uninitialized data is stored. The data in this section will not be cleared during kernel initialization when "USE_NOINIT" macro is set to 1.

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" (or the end of `uTK_BSS` section) and "SYSTEMAREA_END" in configuration file.

2.4 Stack

There are three kinds of stacks in micro T-Kernel as below.

(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.

The stack is usually used during kernel system call processing.

If `TA_USERBUF` is specified, the system stack for a task should be allocated in the memory area explicitly during task creation. Otherwise, the system stack is allocated in micro T-Kernel management area.

The stack area for the first kernel task "inittsk" is allocated in micro T-Kernel management area. Its default size is 1024 bytes.

(2) Interrupt stack

This stack is used in a 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.

When an interrupt occurred, micro T-Kernel switches system stack to the interrupt stack.

There is only one interrupt stack in the entire system.

(3) Temporary stack

This stack is for micro T-Kernel internal use only. It is used when 'dispatch_to_schedtsk()' is called. Its usage depends on the stack usage of `kn1_low_pow()` and task dispatch hook stop routine (if `USE_HOOK_TRACE` macro is enabled).

This temporary stack is allocated in the "uTK_NI_BSS" section.

3 Interrupts and Exceptions

The M32C/87 group provides Fixed Vector Table, which is allocated on a fixed address, and Variable Vector Table, which may be allocated on an arbitrarily address.

3.1 Interrupt Vector Table in Fixed Vector Table

| | |
|----------|-----------------------|
| 0xffffdc | Undefined instruction |
| 0xffffe0 | Overflow |
| 0xffffe4 | BRK instruction |
| 0xffffe8 | Address match |
| 0xffffec | Reserved space |
| 0xfffff0 | Watchdog timer |
| 0xfffff4 | Reserved space |
| 0xfffff8 | NMI |
| 0xfffffc | Reset handler |
| 0xffffff | |

Figure 3-1 Interrupt Vector Table in Fixed Vector Table

Reset vector address must be hard-coded in micro T-Kernel. After reset, the CPU starts executing the program from this address.

3.2 Interrupts definition number

The immediate values (0-63) of interrupt number shall be used by `dintno`, the interrupt definition numbers defined with `tk_def_int()`.

The following is the Interrupt Exception Handling Vector Table.

| Vector Address | Interrupt Number | Interrupt Sources |
|----------------|------------------|-------------------|
| 0xfffd00 | 0 | BRK Instruction |
| 0xfffd04 | 1 | (Reserved) |
| 0xfffd08 | 2 | (Reserved) |
| 0xfffd0c | 3 | (Reserved) |
| 0xfffd10 | 4 | (Reserved) |
| 0xfffd14 | 5 | (Reserved) |
| 0xfffd18 | 6 | (Reserved) |
| 0xfffd1c | 7 | (Reserved) |
| 0xfffd20 | 8 | DMA0 |
| 0xfffd24 | 9 | DMA1 |
| 0xfffd28 | 10 | DMA2 |
| 0xfffd2c | 11 | DMA3 |

| | | |
|-----------|----|--------------------------------------|
| 0xffffd30 | 12 | Timer A0 |
| 0xffffd34 | 13 | Timer A1 |
| 0xffffd38 | 14 | Timer A2 |
| 0xffffd3c | 15 | Timer A3 |
| 0xffffd40 | 16 | Timer A4 |
| 0xffffd44 | 17 | UART0 Transmission, NACK |
| 0xffffd48 | 18 | UART0 Reception, ACK |
| 0xffffd4c | 19 | UART1 Transmission, NACK |
| 0xffffd50 | 20 | UART1 Reception, ACK |
| 0xffffd54 | 21 | Timer B0 |
| 0xffffd58 | 22 | Timer B1 |
| 0xffffd5c | 23 | Timer B2 |
| 0xffffd60 | 24 | Timer B3 |
| 0xffffd64 | 25 | Timer B4 |
| 0xffffd68 | 26 | INT5 |
| 0xffffd6c | 27 | INT4 |
| 0xffffd70 | 28 | INT3 |
| 0xffffd74 | 29 | INT2 |
| 0xffffd78 | 30 | INT1 |
| 0xffffd7c | 31 | INT0 |
| 0xffffd80 | 32 | Timer B5 |
| 0xffffd84 | 33 | UART2 Transmission, NACK |
| 0xffffd88 | 34 | UART2 Reception, ACK |
| 0xffffd8c | 35 | UART3 Transmission, NACK |
| 0xffffd90 | 36 | UART3 Reception, ACK |
| 0xffffd94 | 37 | UART4 Transmission, NACK |
| 0xffffd98 | 38 | UART4 Reception, ACK |
| 0xffffd9c | 39 | Bus Conflict Detection (UART2) |
| 0xffffda0 | 40 | Bus Conflict Detection (UART3/UART0) |
| 0xffffda4 | 41 | Bus Conflict Detection (UART4/UART1) |
| 0xffffda8 | 42 | A/D0 |
| 0xffffdac | 43 | Key Input |
| 0xffffdb0 | 44 | Intelligent I/O Interrupt 0, CAN 3 |
| 0xffffdb4 | 45 | Intelligent I/O Interrupt 0, CAN 4 |
| 0xffffdb8 | 46 | Intelligent I/O Interrupt 2 |
| 0xffffdbc | 47 | Intelligent I/O Interrupt 3 |
| 0xffffdc0 | 48 | Intelligent I/O Interrupt 4 |
| 0xffffdc4 | 49 | Intelligent I/O Interrupt 5, CAN 5 |
| 0xffffdc8 | 50 | Intelligent I/O Interrupt 6 |
| 0xffffdcc | 51 | Intelligent I/O Interrupt 7 |
| 0xffffdd0 | 52 | Intelligent I/O Interrupt 8 |
| 0xffffdd4 | 53 | Intelligent I/O Interrupt 9, CAN 0 |
| 0xffffdd8 | 54 | Intelligent I/O Interrupt 10 |
| 0xffffddc | 55 | INT #55 |
| 0xffffde0 | 56 | INT #56 (TRAP_DEBUG) |
| 0xffffde4 | 57 | Intelligent I/O Interrupt 11 |
| 0xffffde8 | 58 | INT #58 |
| 0xffffdec | 59 | INT #59 |
| 0xffffdf0 | 60 | INT #60 (TRAP_DISPATCH) |
| 0xffffdf4 | 61 | INT #61 (TRAP_RETINT) |
| 0xffffdf8 | 62 | INT #62 |
| 0xffffdfc | 63 | INT #63 (TRAP_SVC) |

Table 3-1 Interrupt Vector Table in Variable Vector Table

3.3 INT exception assignments

The INT exceptions are assigned to interrupt definition numbers #55 (0x37) through #63 (0x3f). Each INT exception is used as shown below:

```
INT #63 micro T-Kernel system call/extended SVC
INT #62 (Reserved)
INT #61 "tk_ret_int()" system call
INT #60 Task dispatcher call
INT #59 (Reserved)
INT #58 (Reserved)
INT #56 Debugger support function
INT #55 (Reserved)
```

INT #55, #58, #59 and #62 are available for users.

3.4 Interrupt handler

If an interrupt handler is defined in the "vector.c" file, the address of the handler shall be defined to the corresponding interrupt number. The interrupt number defined in "vector.c" file can be used with tk_def_int().

If an interrupt occurs, it directly jumps to the set address of 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 assembly code, the entry routine of the interrupt handler named "knl_inthdr_entryN" (N is an interrupt number) is generated. And if "knl_inthdr_entryN" is defined in "vector.c" file, the interrupt number can be used.

The example of Exception/Interrupt handling routine code is shown as below.

```
-----
[include\tk\sysdepend\app_m32c87\asm_depend.h]

/*
 * Exception/Interrupt entry common processing
 * system IPL is set to KNL_INTMASK
 */
#pragma ASM
INT_ENTRY .macro vecno
    .glb knl_inthdr_entry@vecno
    .glb _knl_intvec
knl_inthdr_entry@vecno:
    pushm    a0,a1                ; save A0, A1

    mov.w    #vecno, a0
    mov.l    #_knl_intvec, a1
    add.w    #(vecno << 2), a1
    jmp.i.a  [a1]

.endm
#pragma ENDASM
-----
```

The entry routine executes the following processing

- Save A0 and A1 to the stack
- Save an interrupt number in A1
- Jump to the handler set in `tk_def_int()`

Since `tk_def_int()` is premised on this processing, A0 and A1 need to be saved in stack if handler is defined without using `INT_ENTRY` macro.

The PC and FLG are automatically saved to the stack by the CPU. The A0 and A1 are used as the working register for branch processing to each handler, and thus, saved by each handling routine.

Registers other than A0 and A1 are not saved at "INT_ENTRY" and therefore needs to be saved at each handler as needed. For high-level language support routine defined by `tk_def_int()`, those registers are saved by "knl_inthdr_startup" routine during interrupt startup. If a high-level language support routine is not used, those registers need to be saved explicitly.

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 explicitly.

If 1 is set to "USE_FULL_VECTOR" of "uk_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".

3.5 Kernel Interrupt Mask Level

Kernel interrupt mask level (`KNL_INTMASK`) is the system IPL (`FLG.IPL`) set during kernel internal processing (e.g. system call critical sections).

This parameter is configurable in the following kernel header files. One is used by C code; the other is used by assembly code.

```
-----
[include\tk\sysdepend\app_m32c87\sysdef_depend.h]

/*
 * Kernel interrupt mask level (System IPL)
 */
#define KNL_INTMASK      4          /* 0-7 */
#ifdef NC308
#pragma ASM
_KNL_INTMASK      .define      40h  /* 00h, 10h, 20h, ..., 70h */
#pragma ENDASM
#endif
-----

[include\tk\sysdepend\app_m32c87\sysdef_depend.inc]

;/*
; * Kernel interrupt mask level (System IPL)
; */
KNL_INTMASK .equ      40h  ;/* 00h, 10h, 20h, ..., 70h */
-----
```

User can change the kernel interrupt mask level according to the application requirement. Therefore, for the interrupt which requires high-speed processing, if its IPL is higher than `KNL_INTMASK`, its interrupt handler will be processed as soon as possible regardless of the internal state of micro T-Kernel.

For normal operation, the IPL of kernel dependent interrupt, such as timer interrupt, must be lower or equal to `KNL_INTMASK`. Timer IPL is defined the following kernel header file:

```
-----
[kernel\sysdepend\device\app_m32c87\tkdev_conf.h]
```

```
/*
 * Timer interrupt level
 */
#define TIMER_INTLEVEL          3
-----
```

The system call of micro T-Kernel cannot be executed from the interrupt handler, whose IPL is higher than the kernel interrupt mask level (`KNL_INTMASK`), because it may cause system crash when a system call is issued. The error code "E_CTX" returns if the system call is issued.

The system call of micro T-Kernel can be issued from the interrupt handler, whose IPL is lower or equal to the kernel interrupt mask level (`KNL_INTMASK`).

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.

```
[kernel\sysdepend\device\app_m32c87\icrt.c]
```

- (1) Set interrupt stack pointer (ISP)
- (2) Initialize processor mode
- (3) Initialize system clock
- (4) Initialize FLG register
- (5) Initialize FB register
- (6) Initialize SB register
- (7) Set variable vector address in INTB register
- (8) Set the initial value of DATA section (copy from ROM to RAM)
- (9) Clear BSS section to 0
- (10) Calculate the range of micro T-Kernel management area
- (11) Reset interrupt stack pointer (ISP)
- (12) Set FLG register with system IPL = KNL_INTMASK
- (13) Set FB register to 0
- (14) Call "main" function (sysinit_main.c)

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 part (interrupt handler or timer event handler)

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

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

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

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

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

5.2 Exception/Interrupt used in micro T-Kernel

The following software interrupts are used in micro T-Kernel.

```
INT #63        micro T-Kernel system calls/extended SVC call
INT #61        "tk_ret_int()" system call
INT #60        Task dispatcher call
INT #56        Debugger support function

#12           Timer A0 interrupt
```

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 an INT instruction by the processing equivalent to that of an interface library. Even in this case, the register-saving rules must conform to the NC308 C language rules.

The basic processing of an interface library is executed as below:

- The function code is set in the R3 register and system call is invoked by "INT #63". A function code negative value indicates a system call while the one at 0 or a positive value indicates an extended SVC. However, note that "INT #56" is used for Debugger Support service calls.
- If the debugger support function is enabled for micro T-Kernel, the function code is also stored in R2 register for bookkeeping purpose.
- Registers are saved as follows in accordance with NC308 C language register-saving rule:

| | |
|------------------------|----|
| Frame pointer: | FP |
| Stack pointer: | SP |
| Arguments (<= 16-bit): | R0 |
| Return value: | R0 |

Other registers are saved.

(1) System call interface

The system call interfaces slightly differ depending on whether there is any 32-bit type argument in the first two arguments. There are two cases here:

Case (1) System call without 32-bit type arguments

The first parameter is set to register, and the rest are saved onto the stack. A system call is invoked by "INT #63" ("INT #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 (16-bit) |
//              | p4 (16-bit) |
//              | p3 (16-bit) |
//              | p2 (16-bit) |
//              +-----+
//              | (8bit)      | saved by I/F call
//              | PC (24bit)  |
// Low Address  +-----+
//
//                               R0 = p1
// Function code is set in R3

$tk_xxx_yyy:
    pushm    r2, r3
    mov.w   #TFN_XXX_YYY, r3
.if USE_DBGSP == 1
.if USE_HOOK_TRACE == 1
    mov.w   #TFN_XXX_YYY, r2
.endif
.endif
.if USE_TRAP == 1
    int     #TRAP_SVC
.else
    .glb    _knl_call_entry
    jsr    _knl_call_entry
.endif
    popm    r2, r3
    rts
```

Case (2) System call with 32-bit type arguments

If the first parameter is 16-bit data, it is set to registers, otherwise it is saved onto stack. Other parameter are saved onto the stack. Note, all the 32-bit parameters are saved onto the stack. A system call is invoked by "INT #63" ("INT #TRAP_SVC").

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

```

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

$tk_xxx_yyy:
    pushm    r2, r3
    mov.w    #(TFN_XXX_YYY + n), r3
    .if USE_DBGSP == 1
    .if USE_HOOK_TRACE == 1
        mov.w    #TFN_XXX_YYY, r2
    .endif
    .endif
    .if USE_TRAP == 1
        int     #TRAP_SVC
    .else
        .glb    _knl_call_entry
        jsr    _knl_call_entry
    .endif
    popm    r2, r3
    rts

```

In this case, the function code is increased by n , which depends on the number of 32-bit type arguments and whether the first or second argument is 32-bit type. Here is the rule for calculating the increment factor “ n ”:

If $p1$ is 16-bit, $n = (\text{number of 32-bit type arguments})$

If $p1$ is 32-bit, $n = (\text{number of 32-bit type arguments}) + 1$

If no argument, $n = 0$

Note: if there is no argument saved in register R0, the function symbol prefix would be “_” instead of “\$” (for example, “_tk_xxx_yyy” instead of “\$tk_xxx_yyy”).

(2) Extended SVC interface

Regarding an extended SVC, arguments are wrapped in a packet by the caller, and the start address of packet is set in R2R0 register. An extended SVC call is invoked by INT #63 (“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.

The following additional assumptions are imposed for micro T-Kernel on M32C:

- *All arguments are stored on stack by NC308 compiler.*
- *No function prototype is defined for the extended SVC.*

```

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

/*
 * The function prototype should be removed
 * W zxxx_yyy(p1, p2, p3, p4, p5, p6);
 */

_zxxx_yyy:
    ; All argument value has been stored into stack by compiler
    pushm    r2, r3
    mov.w    #TFN_XXX_YYY, r3 ; R3 = Function code
    stc     sp, r2r0          ; R2R0 = Argument packet top address
    add.l    #04, r2r0
.if USE_TRAP == 1
    int     #TRAP_SVC
.else
    .glb    _knl_call_entry
    jsr    _knl_call_entry
.endif
    popm    r2, r3
    rts

```

(3) micro T-Kernel Debugger Support system call interface

Debugger Support Functions system calls are essentially like other micro T-Kernel service calls, but are called using "INT #56" ("INT #TRAP_DEBUG").

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

```

ER td_xxx_yyy(p1, p2, p3, p4, p5)

$td_xxx_yyy:
    push.w  r3
    mov.w   #(TDFN_XXX_YYY+n), r3
.if USE_TRAP == 1
    int     #TRAP_DEBUG
.else
    .glb    _knl_call_dbgsp
    jsr    _knl_call_dbgsp
.endif
    pop.w   r3
    rts

```

Similar to micro T-Kernel service calls, the function code is increased by n, which depends on the number of 32-bit type arguments and whether the first or second argument is 32-bit type. Here is the rule for calculating the increment factor "n":

If p1 is 16-bit, n = (number of 32-bit type arguments)
 If p1 is 32-bit, n = (number of 32-bit type arguments) + 1
 If no argument, n = 0

Note: if there is no argument saved in register R0, the function symbol prefix would be "_" instead of "\$" (for example, "_tk_xxx_yyy" instead of "\$tk_xxx_yyy").

5.4 Interrupt Handlers

The interrupt handler hardware-dependent implementation definitions are indicated below.

- Interrupt handler definition information T_DINT

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

- Interrupt definition numbers: "dintno"
Definition numbers "dintno" can be designated in the range from 0 to 63.

The 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 the high level programming language support routine within micro T-Kernel is set to interrupt vector table. And, the interrupt handler specified by the high level programming language support routine is called.

The definitions of interrupt handlers are as below.

```
void inthdr ( UINT dintno );
```

- dintno: vector number of the raised interrupt.

At the high level language support routine in micro T-Kernel, no processing is executed to the interrupt controller. Operations such as clearing of the interrupt shall be processed by the interrupt handler.

The restoring from interrupt handler is executed by "return" ("rts" instruction) from function.

(2) "TA_ASM" attribute interrupt handlers

When the attribute of interrupt handler is "TA_ASM", the address of the interrupt handler is directly set to the 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 (FLG.I = 1 and FLG.IPL < KNL_INTMASK), a task dispatch may occur. Other than the exceptions caused by an INT 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 (FLG.I = 1 and FLG.IPL >= KNL_INTMASK). In addition, flag shall be certainly returned before exiting the interrupt handler.

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

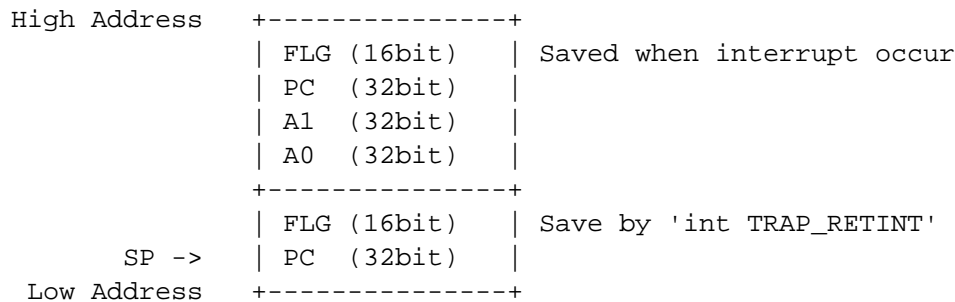
Because of nested interrupt and the like, "knl_taskindp" shall always be set by increment and decrement. A direct value such as "knl_taskindp = 0" shall never be set.

For returning from an interrupt handler, the "tk_ret_int()" system call is used.

Unlike other system calls, the "tk_ret_int()" system call is invoked using the dedicated software interrupt INT #61. It cannot be called in functional form, unlike other system calls.

```
INT #61      ; Call "tk_ret_int()" (non-return)
```

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 R3, SB, FB) shall be restored.



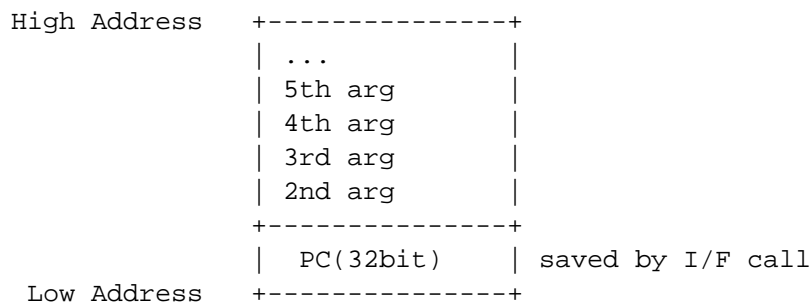
The operation is not guaranteed if "tk_ret_int()" is called by non-interrupt handlers.

5.5 Timer Event Handlers

Timer event handler is called via high-level language support routine even if "TA_ASM" attribute is specified to a handler attribute (just like "TA_HLNG" attribute). Thus, when "TA_ASM" attribute is designated, the parameter("exinf") to be passed to handler is passed via stack in accordance with NC308 C language rules. Besides, the register shall be saved in accordance with NC308 C language rules.

5.6 Stack when system call is invoked

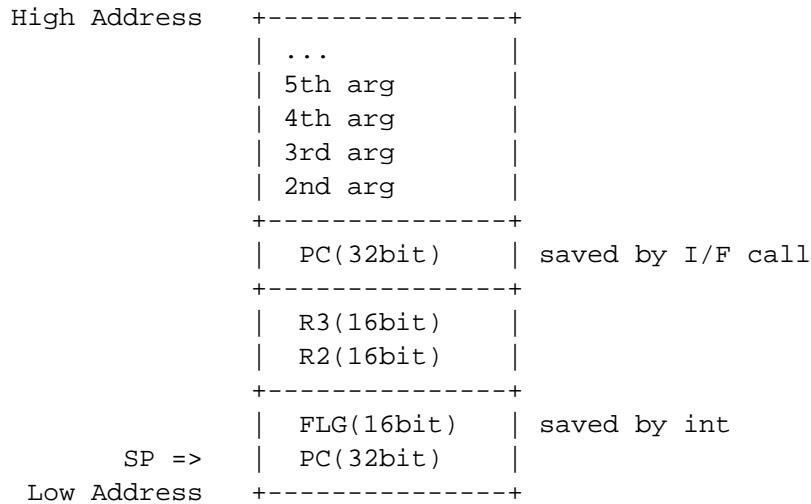
(1) C language I/F (func(arg1, and arg2, ...))



R0 = 1st arg (16-bit)

Note: the above example is the only for the case that the first two arguments are all 16-bit data. However, if the first argument is a 32-bit data, it will be pushed onto the stack before the 2nd argument in accordance with NC308 C language rules.

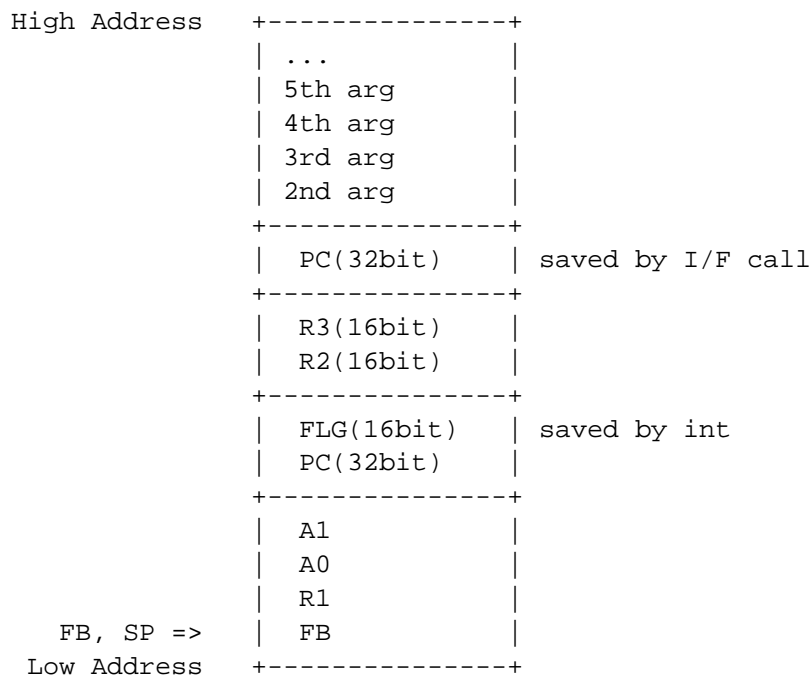
(2) "knl_call_entry" top (immediately after "int #TRAP_SVC" is executed)



R0 = 1st arg (16-bit)

Note: the above example is the only for the case that the first two arguments are all 16-bit data. However, if the first argument is a 32-bit data, it will be pushed onto the stack before the 2nd argument in accordance with NC308 C language rules.

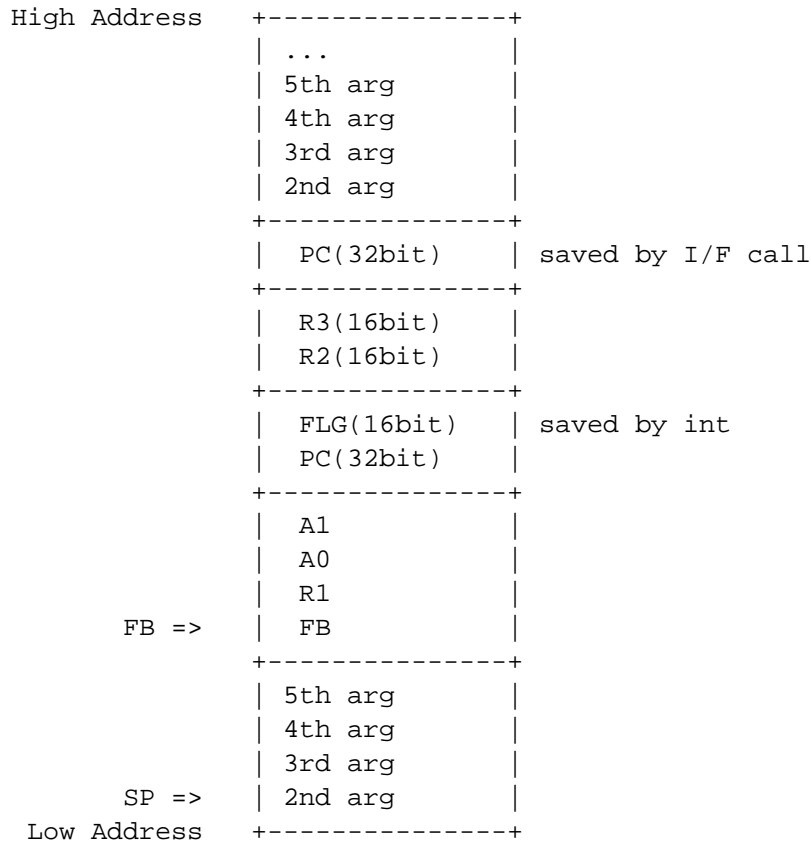
(3) Immediately after "jnz l_esvc_function" is executed



R0 = 1st arg (16-bit)
 R1 = fncd
 R3 = fncd

Note: the above example is the only for the case that the first two arguments are all 16-bit data. However, if the first argument is a 32-bit data, it will be pushed onto the stack before the 2nd argument in accordance with NC308 C language rules.

(4) Immediately before system call is invoked

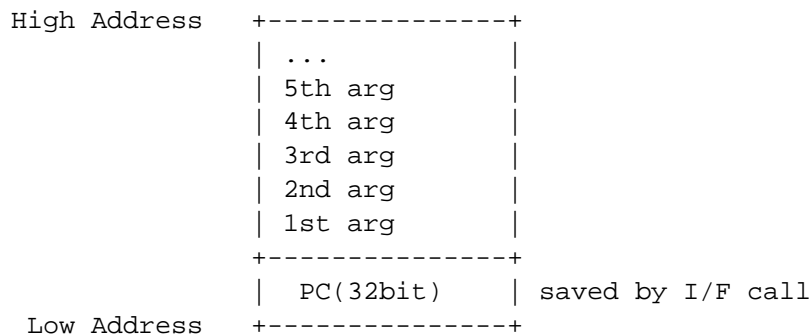


R0 = 1st arg (16-bit)

Note: the above example is the only for the case that the first argument is 16-bit data. However, if the first argument is a 32-bit data, it will be pushed onto the stack in accordance with NC308 C language rules.

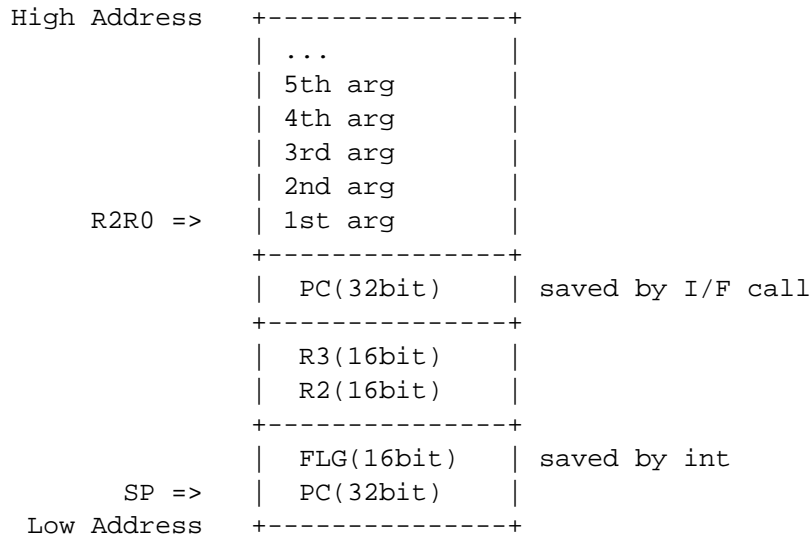
5.7 Stack when the extended SVC is invoked

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



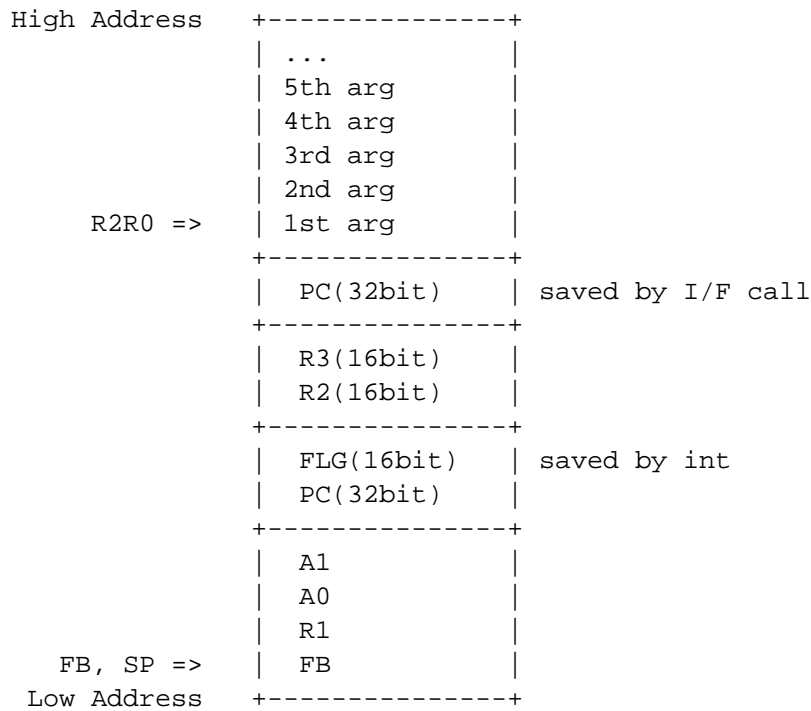
Note: for extended SVC call implementation on micro T-Kernel on M32C, we impose an restriction that no function prototype is defined for extended SVC. Thus, all the arguments are stored on stack in accordance with NC308 C language rules.

(2) "knl_call_entry" top (immediately after "int #TRAP_SVC" is executed)



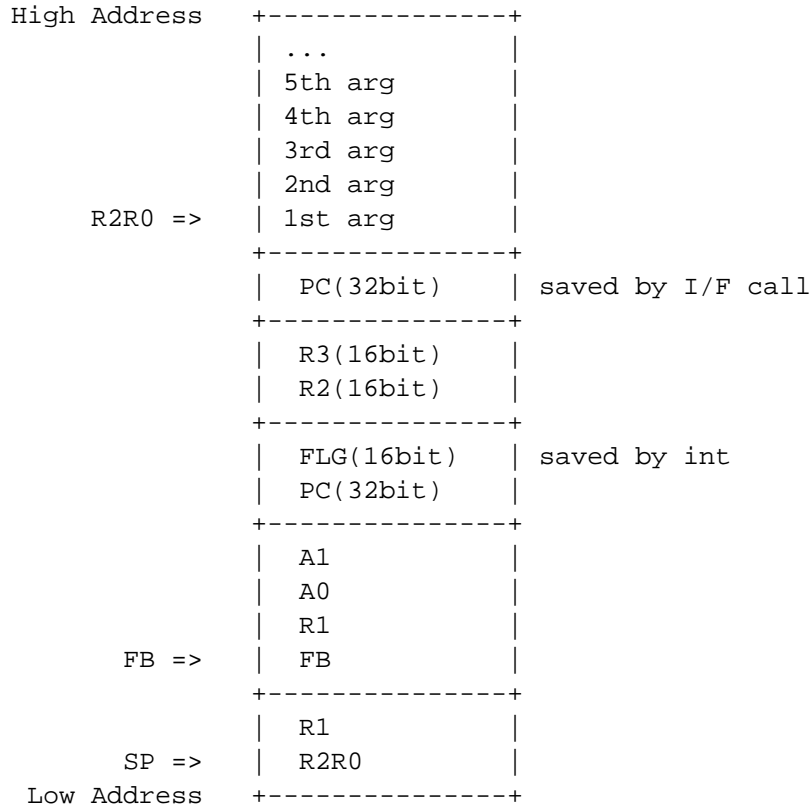
R2R0 = pk_para
R3 = fncd

(3) "l_esvc_function" top



R2R0 = pk_para
R1 = fncd
R3 = fncd

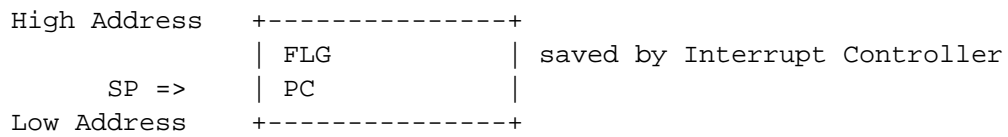
(4) "knl_svc_ientry" top



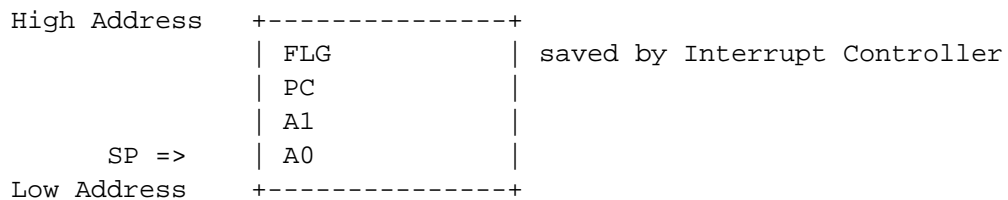
R0 = fncd
R2R0 = pk_para
R3 = fncd

5.8 Stack when an interrupt is raised

- Stack when hardware interrupt is raised



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



A0 = interrupt vector number

5.9 Task implementation-dependent definitions

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

(1) Task creation information T_CTSK

```
typedef struct t_ctsk {
    VP          exinf;          /* Extended information */
    ATR         tskatr;        /* Task attribute */
    FP         task;          /* Task startup address */
    PRI         itskpri;       /* Priority at task startup */
    W          stksz;         /* User stack size (byte) */
    UB         dsname[8];     /* Object name */
    VP         bufptr;        /* User buffer */
} 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.

```
FLG.I = 1      Interrupts enabled
FLG.IPL = 0    System IPL at normal task execution
R0 = stacd    Task startup parameter
SP           stack pointer
```

Note: because exinf is a far pointer (32-bit), it is stored on the task stack frame.

The other register values are indeterminate.

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

5.10 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 target registers for getting/setting of task register ("tk_get/reg()" / "tk_set_reg()") are defined as below.

(1) General-purpose registers T_REGS

```

typedef struct t_regs {
    VW    r[4]; /* Data register R0-R3 */
    VW    a[2]; /* Address register A0-A1 */
    VW    fb;   /* frame base register */
    VW    sb;   /* static base register */
} T_REGS;

```

When setting register to a task in DORMANT state, the task start parameters and the extended information are set in R1 and R2 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 */
    VW    flg;  /* Flag register */
} T_EIT;

```

(3) Control registers T_CREGS

```

typedef struct t_cregs {
    VP    ssp; /* System stack pointer */
} T_CREGS;

```

5.11 System Call/Extended SVC hook routine

The implementation-dependent definitions of system call/extended SVC hook routine are shown below.

(1) Hook routine definition information TD_CALINF

```

typedef struct td_calinf {
    VP    isp; /* System stack pointer */
} TD_CALINF;

```

System stack at entering into hook routine is as follows.

For USE_TRAP == 1

| | | |
|--------------|---------|---------------------------|
| High Address | +-----+ | |
| | | FLG(16bit) saved by int |
| | | PC(32bit) |
| | +-----+ | |
| | | A1 |
| | +-----+ | |
| | | A0 |
| | +-----+ | |
| | | R1 |
| | +-----+ | |
| ISP/SSP => | | FB |
| Low Address | +-----+ | |

For USE_TRAP == 0

```

High Address +-----+
              | PC(32bit) | saved by jsr
              +-----+
              |   A1   |
              +-----+
              |   A0   |
              +-----+
              |   R1   |
              +-----+
ISP/SSP =>   |   FB   |
Low Address +-----+

```

(2) Function prototype for "enter()"

There is a restriction for declaring function prototype for "enter()".

The function format for "enter()" has to be defined with variable-length arguments ("...") as follows.

```
VP enter( FN fncd, TD_CALINF *calinf, ... )
```

If "enter()" does not follow the above format, the arguments may not be correctly passed.

5.12 Other implementation-dependent definitions

(1) Task Control Block (TCB) size and allocation

TCBs for all tasks are allocated in uTK_NI_BSS section.

The actual size of each TCB depends on the kernel configuration.

The maximum size of TCB is 118 bytes.

(2) Changes to Application Interface for device management functions

In the application interfaces of the following functions, the data type "w" is changed to "INT" in order to adapt to 16-bit microcomputer architecture.

```

ID tk_rea_dev_impl( ID dd, INT start, VP buf, INT size, TMO tmout )
ER tk_srea_dev_impl( ID dd, INT start, VP buf, INT size, INT *asize )
ID tk_wri_dev_impl( ID dd, INT start, VP buf, INT size, TMO tmout )
ER tk_swri_dev_impl( ID dd, INT start, VP buf, INT size, INT *asize )
ID tk_wai_dev_impl( ID dd, ID reqid, INT *asize, ER *ioer, TMO tmout )

```

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 (m32c87)
 *   System Configuration Definition
 */
```

```
/* ROMINFO */
#define SYSTEMAREA_TOP 0x0400      /* RAM system area top */
#define SYSTEMAREA_END 0xc300     /* RAM system area end */
```

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

```
/* User definition */
#define RI_USERINIT      NULL      /* User initialization program */
```

User definition initialization/termination program.

```
/* Stacks */
#define RI_INTSTACK      0xc400     /* Interrupt stack top(internal RAM end)
 */
#pragma ASM
__RI_INTSTACK__ .define      0c400H
#pragma ENDASM
```

Initial position of an interrupt stack.
(The additional macro "__RI_INTSTACK__" is defined for assembly 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 for 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.

The setting range is from 1 to the number of the maximum device registrations.

```
#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      0x6100
#define CFN_VER_PRVER      0x0000
#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)0x0000c3ff)
```

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_DBGSP 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 change
 * USE_FULL_VECTOR must be set to 0 if this macro is set to 0.
 */
#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 Building flag

The following modes are configured in the "Renesas M32C Standard Toolchain" configuration window.

- USE_TRAP (trap mode to be configured for "Defines" in C and Assembly Tabs)
 - 1 : use software interrupt (`int`) for system calls
 - 0 : not use software interrupt (`int`)

- USE_NC30LIB (to be configured for "Defines" in C Tab Only)
 - 1 : use built-in standard C library from NC308 tool chain
 - 0 : use string library ("`libstr`") inside micro T-Kernel
(in this case, the "`libstr`" needs to be manually inserted during linking phase)

7 Resource Used by micro T-Kernel

7.3 Kernel Objects

The following kernel objects are used in micro T-Kernel system. Meanwhile, ID number is dynamically allocated, and the ID number at micro T-Kernel normal start-up is written.

| Type | ID | Name | Description |
|----------------|----|---------|---|
| Task | 1 | inittsk | Initial task |
| Message Buffer | - | DEvt | Event notification (Unused by default) Change available in the CFN_DEVT_MBFSZ of utk_config_depend.h |
| Semaphore | - | (NULL) | Device management synchronous control (Use one semaphore every time device is opened) |
| Event flag | 1 | DMLk | Overall device management lock control |

Table 7-1 Kernel objects used in micro T-Kernel

micro T-Kernel

Copyright (C) 2007-2008 by Ken Sakamura. All rights reserved.
micro T-Kernel is distributed under the micro T-License.

Version: 1.01.00

Released by T-Engine Forum(<http://www.t-engine.org>) at 2008/03/05.

Modified Source Code:

- Renesas Starter Kit for M16C/62P Version m16c62p.1B
- Renesas Starter Kit for M32C/87

Changes:

- Adapted to the Renesas Starter Kit for M32C/87
- Corresponded to the RENESAS Compiler (M3T-NC308WA).

Changed by RENESAS TECHNOLOGY CORP.
and RENESAS SOLUTIONS CORP.
and RENESAS TECHNOLOGY SINGAPORE PTE. LTD. at 2008/03/10.

Real-time Operating System for M32C Family
micro T-Kernel Implementation Specification M32C (M32C87)

Publication Date: March 28, 2008 Rev.1.00

Published by: Renesas Technology Corp.

Edited by: System Engineering Department
Renesas Technology Singapore Pte. Ltd.

micro T-Kernel
Implementation Specification
M32C(M32C87)

