

FR Family

μ T-Kernel Specification Compliant

SOFTUNE™ μ T-REALOS/FR

USER'S GUIDE

FR Family
μT-Kernel Specification Compliant
SOFTUNE™ μT-REALOS/FR
USER'S GUIDE

FUJITSU MICROELECTRONICS LIMITED

Preface

■ Purpose and Intended Reader of This Manual

This manual covers how to create application programs that use SOFTUNE μ T-REALOS/FR (referred to as " μ T-REALOS" in this manual), and describes the overall functionality of μ T-REALOS, how to create application programs, and the procedure for building a system.

Reading this manual requires basic knowledge of the FR processor and basic knowledge related to real-time OSs.

See the "SOFTUNE μ T-REALOS/FR API Reference" (referred to as the "API Reference" in this manual) for details on the system call interfaces, and the "SOFTUNE μ T-REALOS/FR Analyzer Guide" (referred to as the "Analyzer Guide" in this manual) for details on the analyzer.

■ About the μ T-Kernel

The μ T-Kernel specifications are specifications for an open real-time OS established by the T-Engine Forum. The μ T-Kernel specifications are available from the T-Engine Forum website (<http://www.t-engine.org/>). The original copyright for the μ T-Kernel belongs to Mr. Ken Sakamura. The copyright for the μ T-Kernel specifications belongs to the T-Engine Forum. This product uses the μ T-Kernel source code from the T-Engine Forum (www.t-engine.org) based on the μ T-License.

■ Trademarks

SOFTUNE is a trademark of Fujitsu Microelectronics Limited.

REALOS is a trademark of Fujitsu Microelectronics Limited.

TRON is an abbreviation of "The Real-time Operating system Nucleus".

ITRON is an abbreviation of "Industrial TRON".

μ ITRON is an abbreviation of "Micro Industrial TRON".

T-Kernel and μ T-Kernel are the name of computer specifications, and do not refer to a particular product or group of products.

The company names and brand names herein are the trademarks or registered trademarks of their respective owners.

■ Overall Overall Structure of This Manual

This manual consists of five chapters and an appendix as follows.

CHAPTER 1 OVERVIEW OF μ T-REALOS

This chapter explains an overview of μ T-REALOS.

CHAPTER 2 BASIC CONCEPTS OF THE μ T-REALOS KERNEL

This chapter describes the basic concepts that need to be understood in advance before using the μ T-REALOS kernel.

CHAPTER 3 μ T-REALOS FUNCTIONS

This chapter describes the functions supported by μ T-REALOS.

CHAPTER 4 WRITING A USER PROGRAM

This chapter describes the basic items in writing a user program on μ T-REALOS.

CHAPTER 5 HOW TO CONSTRUCT A SYSTEM

This chapter describes how to construct a user system.

APPENDIX

The appendix describes error messages of the configurator.

■ Reference Manuals

See the manuals listed below as required while using this system.

SOFTUNE μ T-REALOS/FR API Reference

SOFTUNE μ T-REALOS/FR Analyzer Guide

FR Family SOFTUNE C/C++ Compiler Manual V6

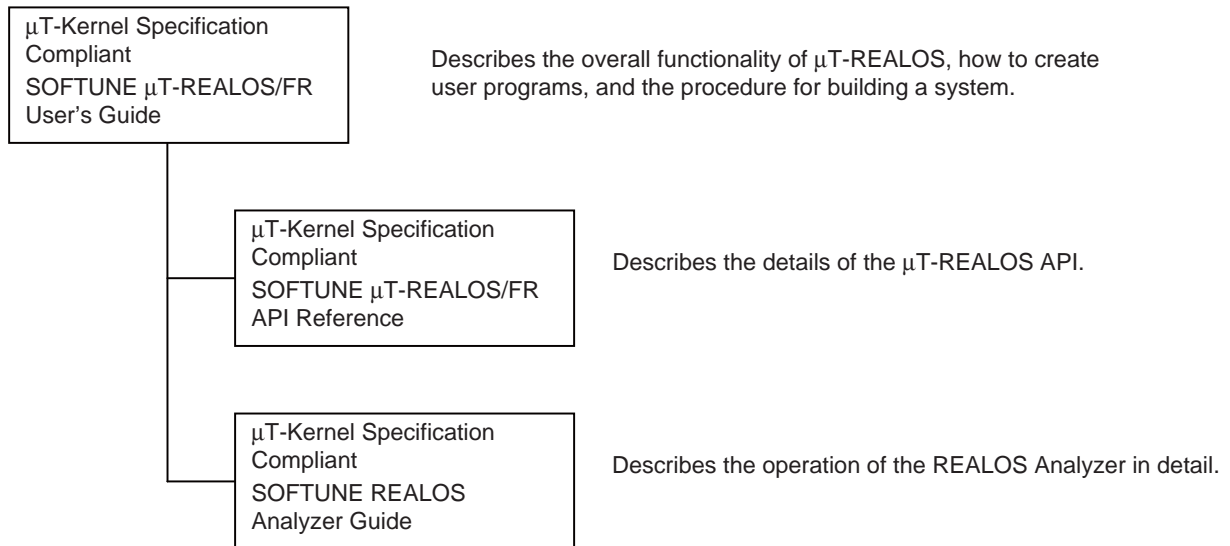
FR Family SOFTUNE Assembler Manual V6

FR Family SOFTUNE Linkage Kit Manual V6

■ Organization of the μ T-REALOS Manuals

The μ T-REALOS manuals are divided into the following three volumes.

First-time users of μ T-REALOS should read the "SOFTUNE μ T-REALOS/FR User's Guide" first.



■ How to read This Manual

● Explanation of terminology

The terminology used in this manual is described below.

Word	Overview
Kernel	The program that provides the OS functionality is called the kernel.
User program	Refers to application programs that use μ T-REALOS functions. In order to emphasize the point that these programs are created by the user, these are called user programs in this manual.
User system	Refers to an executable program formed by linking a user program with μ T-REALOS.
System call	The group of functions that implement OS functionality and that can be called directly from a user program are called system calls.
Object	The resources that are handled by the kernel are called objects. Specifically, this refers to semaphores, mailboxes, and other objects that implement functionality such as tasks, synchronization, and communications.
Configuration definition macros	The configuration definition macros are written in the system configuration file, and act as an interface for setting kernel configuration parameters.
Idle state	The state when there are no tasks ready to execute.

- The contents of this document are subject to change without notice.
Customers are advised to consult with sales representatives before ordering.
- The information, such as descriptions of function and application circuit examples, in this document are presented solely for the purpose of reference to show examples of operations and uses of FUJITSU MICROELECTRONICS device; FUJITSU MICROELECTRONICS does not warrant proper operation of the device with respect to use based on such information. When you develop equipment incorporating the device based on such information, you must assume any responsibility arising out of such use of the information. FUJITSU MICROELECTRONICS assumes no liability for any damages whatsoever arising out of the use of the information.
- Any information in this document, including descriptions of function and schematic diagrams, shall not be construed as license of the use or exercise of any intellectual property right, such as patent right or copyright, or any other right of FUJITSU MICROELECTRONICS or any third party or does FUJITSU MICROELECTRONICS warrant non-infringement of any third-party's intellectual property right or other right by using such information. FUJITSU MICROELECTRONICS assumes no liability for any infringement of the intellectual property rights or other rights of third parties which would result from the use of information contained herein.
- The products described in this document are designed, developed and manufactured as contemplated for general use, including without limitation, ordinary industrial use, general office use, personal use, and household use, but are not designed, developed and manufactured as contemplated (1) for use accompanying fatal risks or dangers that, unless extremely high safety is secured, could have a serious effect to the public, and could lead directly to death, personal injury, severe physical damage or other loss (i.e., nuclear reaction control in nuclear facility, aircraft flight control, air traffic control, mass transport control, medical life support system, missile launch control in weapon system), or (2) for use requiring extremely high reliability (i.e., submersible repeater and artificial satellite).
Please note that FUJITSU MICROELECTRONICS will not be liable against you and/or any third party for any claims or damages arising in connection with above-mentioned uses of the products.
- Any semiconductor devices have an inherent chance of failure. You must protect against injury, damage or loss from such failures by incorporating safety design measures into your facility and equipment such as redundancy, fire protection, and prevention of over-current levels and other abnormal operating conditions.
- Exportation/release of any products described in this document may require necessary procedures in accordance with the regulations of the Foreign Exchange and Foreign Trade Control Law of Japan and/or US export control laws.
- The company names and brand names herein are the trademarks or registered trademarks of their respective owners.

Copyright ©2008 FUJITSU MICROELECTRONICS LIMITED All rights reserved.

Copyright ©2006 T-Engine Forum. All rights reserved.

This manual is made based on the specification of μ -Kernel with the formal agreement by the T-Engine Forum.

CONTENTS

CHAPTER 1	OVERVIEW OF μT-REALOS	1
1.1	Supported Functions	2
1.2	Directory Structure of Provided Files	3
1.3	Tools Required for Development	4
1.4	Structure of Product	5
CHAPTER 2	BASIC CONCEPTS OF THE μT-REALOS KERNEL	7
2.1	System Calls	8
2.2	Execution Units of User Program	9
2.2.1	Tasks	10
2.2.2	Initial Routines	14
2.2.3	Interrupt Handlers	15
2.2.4	Time Event Handlers	16
2.2.5	Error Routines	18
2.2.6	Extended SVC Handlers	19
2.2.7	Device Processing Functions	20
2.3	Objects	21
2.4	System States	22
2.5	Enabling and Disabling Dispatching and Interrupts	24
2.6	Precedence of Execution of Tasks and Handlers	25
CHAPTER 3	μT-REALOS FUNCTIONS	27
3.1	Overview of μ T-REALOS Functions	28
3.2	Task Management Functions	29
3.3	Task Synchronization Functions	30
3.4	Synchronization and Communication Functions	31
3.4.1	Semaphore Functions	32
3.4.2	Event Flag Functions	34
3.4.3	Mailbox Functions	35
3.5	Extended Synchronization and Communication Functions	37
3.5.1	Mutex Functions	38
3.5.2	Message Buffer Functions	40
3.5.3	Rendezvous Port Functions	42
3.6	Memory Pool Management Functions	45
3.6.1	Fixed-size Memory Pool Functions	46
3.6.2	Variable-size Memory Pool Functions	47
3.7	Time Management Functions	48
3.7.1	System Time Management Functions	49
3.7.2	Cyclic Handler Functions	50
3.7.3	Alarm Handler Functions	52
3.8	Interrupt Management Functions	53
3.9	System State Management Functions	54
3.10	Subsystem Management Functions	55

3.11	Device Management Functions	56
3.12	Power Saving Functions	58
3.13	Configuration Functions	59
3.14	Debugging Assistance Functions	64
CHAPTER 4	WRITING A USER PROGRAM	73
4.1	Configuring a User Program	74
4.2	Start Flow	75
4.3	Reset Entry Routine	76
4.4	Initial Routine	79
4.5	Task	81
4.6	Period Handler	85
4.7	Alarm Handler	86
4.8	Interrupt Handler	87
4.9	Error Routine	89
4.10	Power Saving Routine	90
4.11	Extension SVC Handler	91
4.12	Device Driver	92
4.13	Notes when Writing a User Program	95
CHAPTER 5	HOW TO CONSTRUCT A SYSTEM	97
5.1	Steps of Constructing a System	98
5.2	Create the μ T-REALOS Project	99
5.3	Setting of Configuration	102
5.4	Setting of Linker Option	109
5.5	Build a User System	114
APPENDIX	115
	APPENDIX A Error Messages of the Configurator	116
INDEX.....	127

CHAPTER 1

OVERVIEW OF μ T-REALOS

This chapter explains an overview of μ T-REALOS. μ T-REALOS is a μ T-Kernel specification real-time OS that runs on the FR family of 32-bit RISC controllers.

μ T-REALOS is conforms to the μ T-Kernel specifications.

- 1.1 Supported Functions
- 1.2 Directory Structure of Provided Files
- 1.3 Tools Required for Development
- 1.4 Structure of Product

1.1 Supported Functions

This section provides an overview of the functions supported by μ T-REALOS.

■ Supported Functions

Libraries and header files are provided with μ T-REALOS. The libraries are used by linking them with the user program to create an executable program that can run on the target processor. The header files are included by user programs in order to use the μ T-Kernel API. The combination of libraries and header files is called the μ T-REALOS kernel (referred to as the "kernel" in this manual). The kernel is a program that implements the following functionality of μ T-Kernel.

- Task management functions
- Task synchronization functions
- Synchronization and communication functions (semaphores, event flags, mailboxes)
- Extended synchronization and communication functions (message buffers, mutexes, rendezvous ports)
- Memory pool management functions (fixed length memory pool, variable length memory pool)
- Time management functions
- Interrupt management functions
- System configuration management functions
- Subsystem management functions
- Device management functions
- Power saving functions

See "CHAPTER 3 μ T-REALOS FUNCTIONS" and "CHAPTER 3 SYSTEM CALL INTERFACE" of the "API Reference" for details on the above functions.

The following development tools are also provided with μ T-REALOS for use when building or debugging a system. These are Windows applications that run on a PC.

- SOFTUNE μ T-REALOS Configurator
- SOFTUNE μ T-REALOS Analyzer

SOFTUNE μ T-REALOS Configurator (referred to as the "Configurator" in this manual) is used when building a user system to configure the kernel based on a predefined structure. See "3.13 Configuration Functions" for details on the Configurator functions.

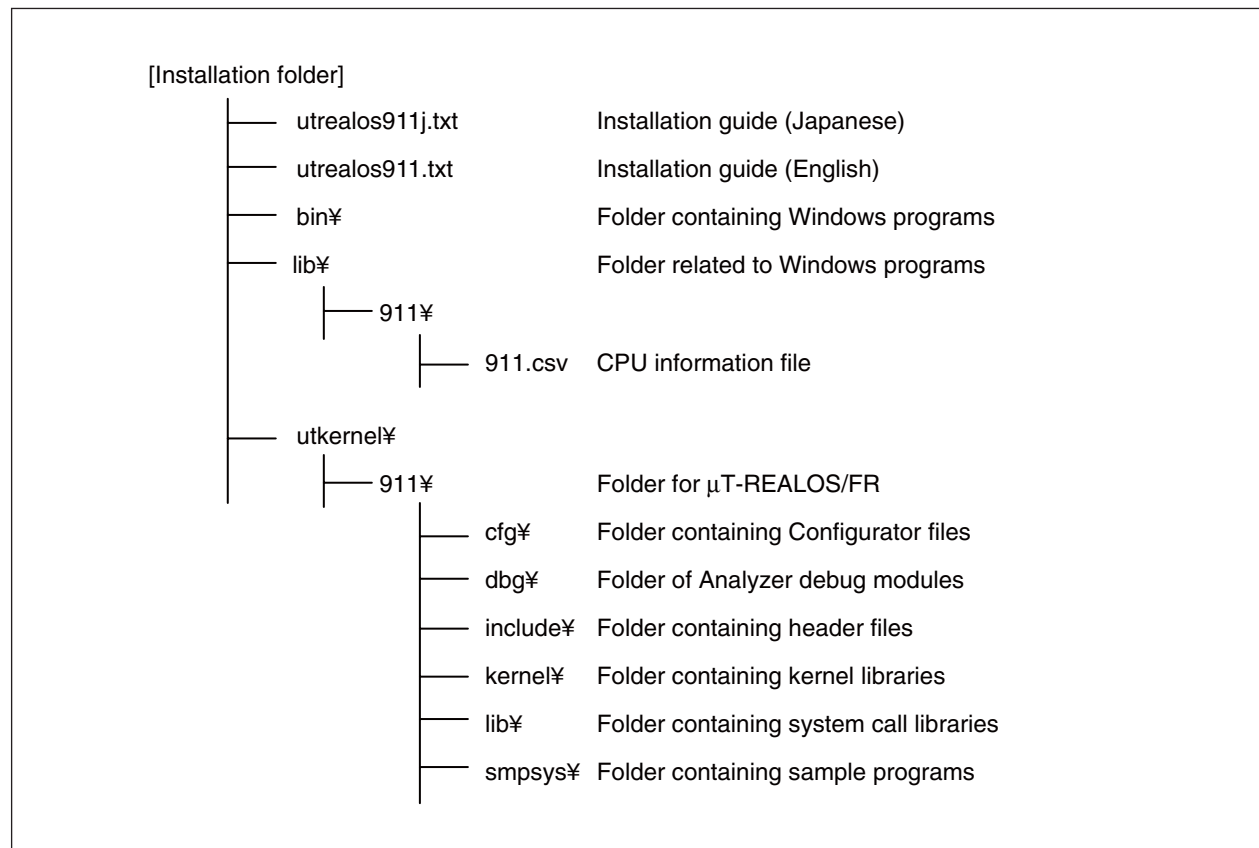
SOFTUNE μ T-REALOS Analyzer (referred to as the "Analyzer" in this manual) is used when debugging a user program and includes a variety of functions for improving debugging efficiency. See "3.14 Debugging Assistance Functions" and the "Analyzer Guide" for details.

1.2 Directory Structure of Provided Files

This section describes the directory structure of the files provided with the μ T-REALOS API.

■ Directory Structure of Provided Files

μ T-REALOS is installed using the following directory structure of the file.



See the "Installation Guide" for details of the directory structure. The Installation Guide can be found on the product CD-ROM and in the installation folder.

1.3 Tools Required for Development

This section describes the tools that are required to develop a user system.

■ Tools Required for Development

The following tools are required to develop a μ T-REALOS user system.

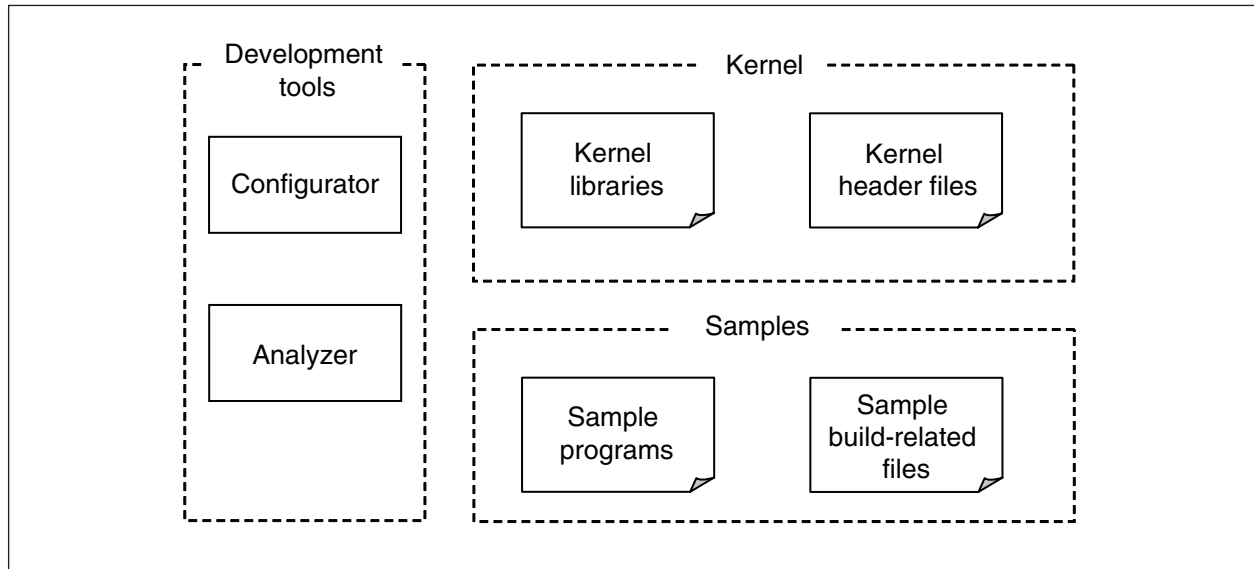
- Cross-development tool
FR Family SOFTUNE Professional Pack V6
- ICE
Fujitsu MB2198 series

1.4 Structure of Product

This section explains the structure of the product.

■ Structure of Product

The structure of μ T-REALOS is shown below.



- **Configurator**
The Configurator modules that run under Windows. This includes command format (.exe) executable files that are run from the command prompt window and DLL format files that are used as SOFTUNE Workbench add-ins.
- **Analyzer**
Consists of DLL format files that are used as add-ins in SOFTUNE Workbench and debugger object files that are linked with user programs to collect trace data.
- **Kernel libraries**
The μ T-REALOS kernel object files are included in SOFTUNE library format.
- **Kernel header files**
Header files that are included by user programs, and which define system calls and parameter types.
- **Sample programs**
Samples programs of reset entry routines, initialization processing, timer interrupt handlers, and tasks.
- **Sample build-related files**
SOFTUNE project files, configuration files, and other files for the sample programs.

CHAPTER 2

BASIC CONCEPTS OF THE μ T-REALOS KERNEL

This chapter describes the basic concepts that need to be understood in advance before using the μ T-REALOS kernel.

- 2.1 System Calls
- 2.2 Execution Units of User Program
- 2.3 Objects
- 2.4 System States
- 2.5 Enabling and Disabling Dispatching and Interrupts
- 2.6 Precedence of Execution of Tasks and Handlers

2.1 System Calls

This section describes the system calls which act as an interface for calling kernel functions from the user program.

■ System Calls

The interface for calling kernel functions using general-purpose data types and constant macros from a user program are called system calls. The system calls conform to the μ T-Kernel specifications.

See "CHAPTER 3 SYSTEM CALL INTERFACE" of the "API Reference" for details on the system calls.

2.2 Execution Units of User Program

This section describes the execution units of a user program.

■ Execution Units of User Program

The execution units of a user program can be broadly divided into tasks, initial routines, interrupt handlers, time event handlers, error routines, extended SVC handlers, and device driver processing functions.

- Tasks
- Initial routines
- Interrupt handlers
- Time event handlers
- Error routines
- Extended SVC handlers
- Device driver processing functions

2.2.1 Tasks

This section describes tasks.

■ Tasks

Tasks are the program execution unit that form the basis of user program processing.

In μ T-REALOS, if the execution of a task is interrupted, the state prior to the interrupt (register values) is saved on a per-task basis. This is called the task context. The information saved in the task context can be used to resume execution of the interrupted task.

Tasks have a variety of states, including the run state, ready state, WAITING, etc. See "■ Task Portions" for details on the task portion transitions.

■ Current Task and Other Tasks

When a system call is made from a task, the calling task is called the current task and all other tasks are called other tasks.

■ Priority Sequence and Task Priorities

The order of execution of program execution units is called the precedence. The value that determines the precedence of a task is called the task priority. The smaller the value of the task priority, the higher the priority. Tasks with a higher priority (small task priority value) have precedence when executing.

The task priority consists of a base priority, current priority, and startup priority. The term task priority by itself refers to the current priority. The current priority is used to determine the execution sequence of the task. The base priority is the base priority of the task, and normally has the same value as the current priority. When mutex functions are used, however, the current priority may be changed temporarily in some cases and can differ from the base priority. Even in these situations, however, the modified current priority is restored to the base priority when the mutex function has finished being used (see "3.5.1 Mutex Functions"). The startup priority is the priority specified when a task is created, and the base priority of the task is initialized to the value of the startup priority when the task starts.

■ Dispatching and Preemption

The process of switching between running tasks is called dispatching. The process of a task that is in the run state losing the execution right is called preemption. The functionality within the kernel that implements dispatching is called the dispatcher.

Dispatching occurs when a task that has a higher priority than the currently executing task enters the ready state. Preemption occurs when a dispatch occurs or an interrupt handler is activated while a task is executing.

■ Task Portions

Tasks have the following states.

● RUNNING

The state where the task is running.

Note that if a program other than a task is running, the task that was running prior to that program remains in the run state.

● READY

The state where the task is ready to execute, but is unable to run because a task that is higher in the precedence is currently running.

● WAITING

The state where execution has been suspended due to calling a system call with some kind of wait condition. This is categorized into the following states depending on the wait condition.

- Wakeup wait state (waiting due to tk_slp_tsk)
- Elapsed time wait state (waiting due to tk_dly_tsk)
- Semaphore resource acquisition wait state (waiting due to tk_wai_sem)
- Event flag wait state (waiting due to tk_wai_flg)
- Receive from mailbox wait state (waiting due to tk_rcv_mbx)
- Mutex lock wait state (waiting due to tk_loc_mtx)
- Send to message buffer wait state (waiting due to tk_snd_mbf)
- Receive from message buffer wait state (waiting due to tk_rcv_mbf)
- Fixed length memory block acquisition wait state (waiting due to tk_get_mpf)
- Variable length memory block acquisition wait state (waiting due to tk_get_mpl)
- Rendezvous call/termination wait state (waiting due to tk_cal_por)
- Rendezvous accept wait state (waiting due to tk_acp_por)

● SUSPENDED

The state where execution has been forcefully suspended by another task.

● WAITING-SUSPENDED

This state is both WAITING and SUSPENDED at the same time.

● DORMANT

The state where the task has not yet been started, or the task has ended.

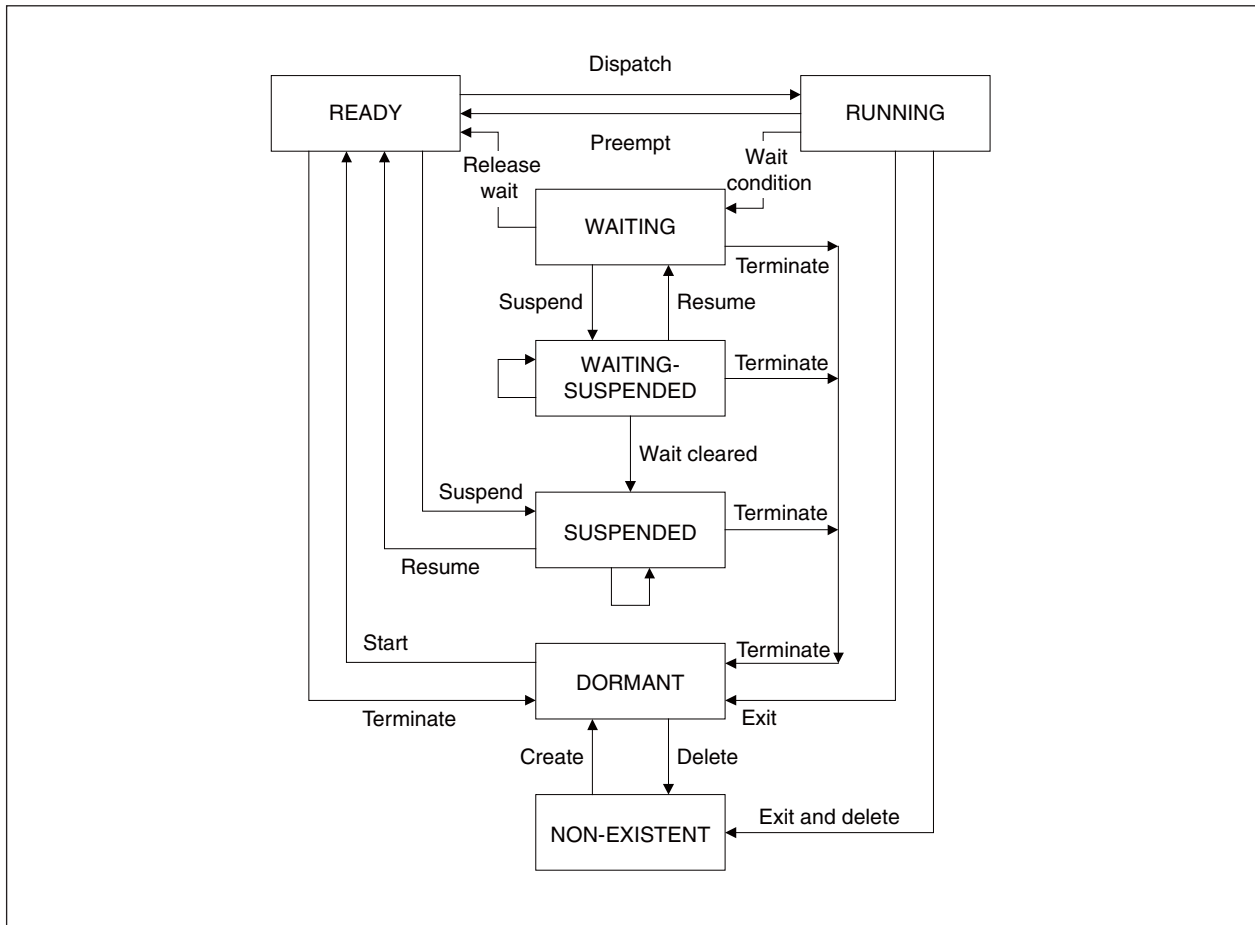
● NON-EXISTENT

The state where the task has not yet been created, or the task has been deleted.

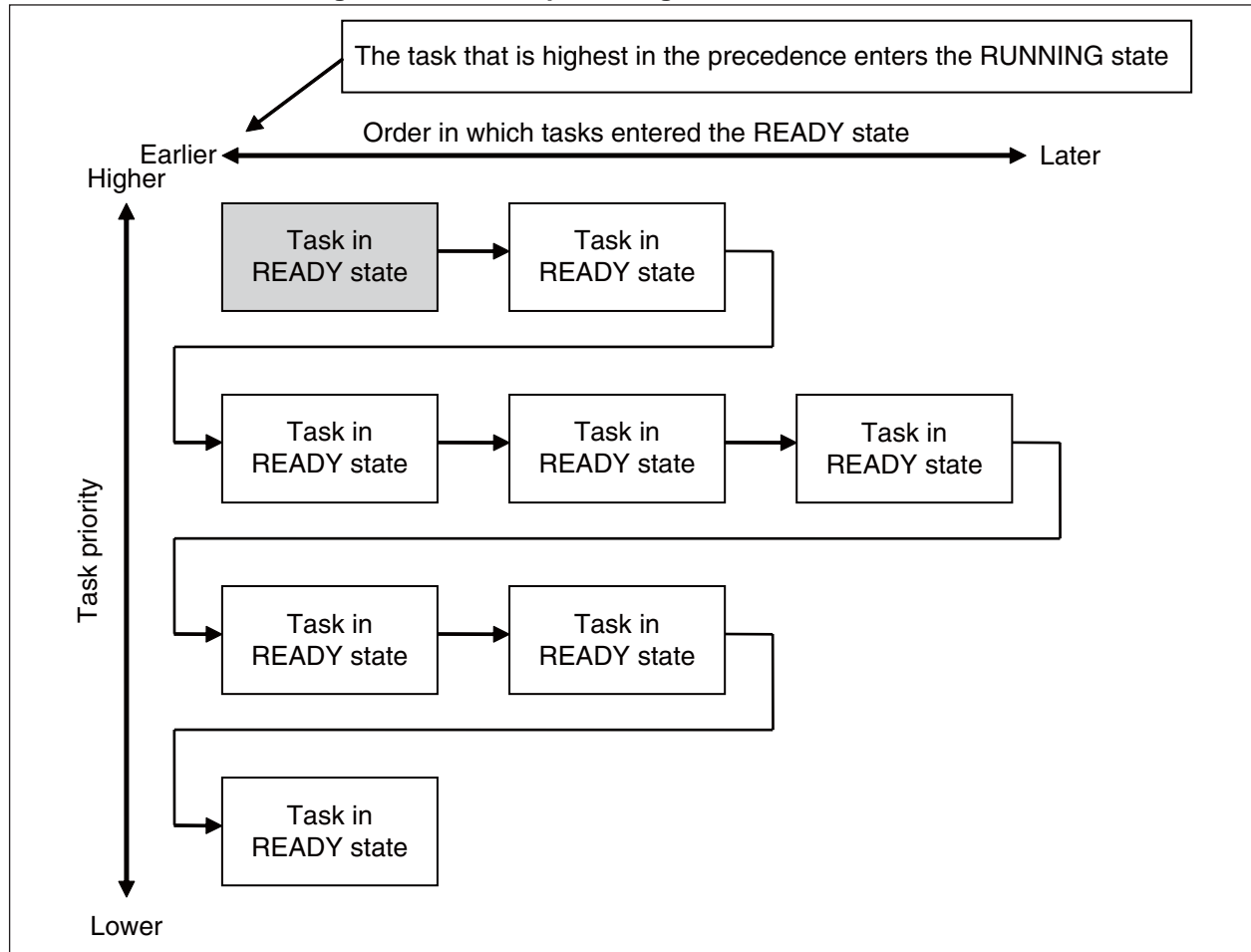
■ Task Portion Transitions

The state transitions for tasks are shown below.

Figure 2.2-1 Task Portion Transitions



When multiple tasks are in the **READY** state, the tasks are scheduled (controls the execution sequence) according to the task priorities. The task that has the highest position in the precedence from among the tasks that are in the **READY** state is placed in the **RUNNING** and the task is executed. The task precedence is ordered such that tasks that have higher task priorities are placed higher in the task precedence. For tasks with the same task priority, the task that entered the **READY** state first has the highest position in the precedence.

Figure 2.2-2 Conceptual Diagram of the Precedence

2.2.2 Initial Routines

This section describes initial routines.

■ Initial Routines

An initial routine is program to perform initialization processing that is specific to the user program, and generally prepares the environment in which the user program can run by creating tasks, semaphores and other objects and registering interrupt handlers and devices.

During kernel initialization, the initial task is automatically created to perform internal kernel initialization. This initial task calls the predefined initial routine from the user program. The initial routine therefore runs as a task.

2.2.3 Interrupt Handlers

This section describes the task-independent part of the interrupt handlers.

■ Interrupt Handlers

An interrupt handler is a program that is activated synchronously with peripheral hardware interrupt sources, CPU exceptions, and software interrupt instructions. Interrupt handlers can be defined for each interrupt source.

If an interrupt occurs while a task is running, the kernel temporarily interrupts task execution and runs the interrupt handler corresponding to the interrupt source that occurred. At this time, the stack switches to the stack that is provided for executing interrupt processing (the system stack). The interrupt handler therefore does not execute in the context of the task that had been running, but instead executes in an independent context.

Furthermore, all of the interrupt handlers run at a higher priority than the tasks, therefore tasks do not run until the interrupt handler has finished. If multiple interrupt handlers are activated, task execution does not continue until all of the interrupt handlers have finished processing. Therefore, even if an interrupt handler calls a system call that results in a dispatch (such as starting a task with a high priority), the actual task dispatch is not performed until after all of the interrupt handlers have finished processing. This behavior is called "delayed dispatch".

See "3.8 Interrupt Management Functions" for details on the interrupt handlers.

2.2.4 Time Event Handlers

This section describes time event handlers.

■ Time Event Handlers

Cyclic handlers and alarm handlers are collectively referred to as time event handlers.

■ Cyclic Handlers

A cyclic handler is a program that is activated at a specified interval regular. Programs that are executed periodically can be defined as cyclic handlers, and the execution and suspension of these handlers are able to be controlled.

If the specified interval elapses while a task is executing, the task execution is temporarily interrupted and the corresponding cyclic handler is activated. The cyclic handler does not execute in the context of the task that had been running, but instead executes in an independent context.

Furthermore, all of the cyclic handlers run at a higher priority than the tasks, therefore tasks do not run until the cyclic handler has finished. Even if the cyclic handler calls a system call that results in a dispatch (such as starting a task with a high priority), the actual task dispatch is not performed until after the cyclic handler has finished processing.

The cyclic handlers in μ T-REALOS are activated from within `isig_tim`, which is called from the timer interrupt handler for the system clock. The cyclic handlers therefore operate as part of the timer interrupt handler. Time-related handlers that are activated from the timer interrupt handler in this way are called "time event handlers". In μ T-REALOS, cyclic handlers and alarm handlers, which are described next, are collectively referred to as time event handlers. As described earlier, cyclic handlers execute as part of the timer interrupt handler, and a cyclic handler is therefore not interrupted to process other time event handlers while the cyclic handler is running.

The time when a cyclic handler is first activated is calculated based on the time tick following the time when the cyclic handler is created or activated. However, if a cyclic handler is created or activated from within a time event handler, the time is calculated based on the time when the time event handler was activated. The activation time after the first time is calculated based on the time when the cyclic handler was activated.

See "3.7.2 Cyclic Handler Functions" for details on cyclic handlers.

■ Alarm Handlers

An alarm handler is a program that is activated at a specified time. The program that is executed at the specified time is created as an alarm handler, and the execution and suspension of these handlers are able to be controlled.

If the specified time is reached while a task is executing, the task execution is temporarily interrupted and the corresponding alarm handler is executed. The alarm handler does not execute in the context of the task that had been running, but instead executes in an independent context.

Furthermore, all of the alarm handlers run at a higher priority than the tasks, therefore tasks do not run until the alarm handler has finished. Even if the alarm handler calls a system call that results in a dispatch (such as starting a task with a high priority), the actual task dispatch is not performed until after the alarm handler has finished processing.

The alarm handlers in μ T-REALOS operate as part of the interrupt handler for the system clock. Alarm handlers are therefore not interrupted to process other time event handlers while the alarm handler is running.

The time when the alarm handler is activated is calculated based on the time tick following the time when the alarm handler is activated. However, if an alarm handler is activated from within a time event handler, the time is calculated based on the time when the time event handler was activated.

See "3.7.3 Alarm Handler Functions" for details on alarm handlers.

2.2.5 Error Routines

This section describes error routines.

■ Error Routines

An error routine is a program that is run when the kernel detects some kind of error.

The error routine is activated under the following conditions.

- System Down
An internal kernel inconsistency is detected
- Initial Settings Error
An error occurs during kernel initialization
- Undefined Interrupt
An interrupt occurs that does not have a defined interrupt handler

The error routine is used for the purpose of debugging the user program. There is no way to recover from the error routine. Therefore, if the error routine has been called, clear the cause of the error and restart the system.

If the error routine is called due to an initial settings error, it runs as a task. Otherwise it runs in a task-independent portion.

2.2.6 Extended SVC Handlers

This section describes extended SVC handlers.

■ Extended SVC Handlers

An extended SVC handler is a handler that acts as a receiver for requests to subsystems. If called from a task, the handler runs as a quasi-task portion!. If called from a task-independent context, the handler runs in a task-independent portion. See "3.10 Subsystem Management Functions" for details on the subsystems.

2.2.7 Device Processing Functions

This section describes the device processing functions.

■ Device Processing Functions

The device processing functions are device driver functions that are called from device management functions. The device processing functions operate in the task context if they are called as a task extension. The functions run in a task-independent portion if they are called as a task-independent extension. See "3.11 Device Management Functions" for details on device processing functions.

2.3 Objects

This section describes objects.

■ Objects

μ T-REALOS supports a variety of functions, including synchronization/communication between tasks, exclusive control, and acquisition/release of memory regions. The resources that operate on system calls in order to use these functions from a user program are called objects.

The following objects are available in μ T-REALOS. See each of the descriptions in Table 2.3-1 for details on each of the objects.

Table 2.3-1 List of Objects

Object	Synopsis
Task	The task object is the most fundamental unit that makes up a user program.
Semaphore	Semaphores are objects for representing numerically the number and availability of unused resources, and for managing exclusive control and synchronization when using those resources.
Event flag	Event flags are objects that perform synchronization by representing the presence or absence of events as bit flags.
Mailboxes	Mailboxes are objects that perform synchronization and communication by receiving messages that are stored in memory.
Mutexes	Mutexes are objects that perform exclusive access control between tasks that use a shared resource.
Message buffers	Message buffers are objects that perform synchronization and communication by receiving variable-length messages.
Rendezvous ports	Rendezvous ports provide intertask synchronous communication functionality, and support a single sequence where one task requests processing of another task and the other task then returns the processing result to the first task. The object that synchronizes the waiting of both tasks is called a rendezvous port.
Fixed-size memory pool	Fixed-size memory pool are objects for dynamically managing fixed size memory blocks.
Variable-size memory pool	Variable-size memory pool are objects for dynamically managing arbitrary size memory blocks.
Cyclic handlers	Cyclic handlers are time event handlers that activate at a fixed period.
Alarm handlers	Alarm handlers are time event handlers that activate at a specified time.

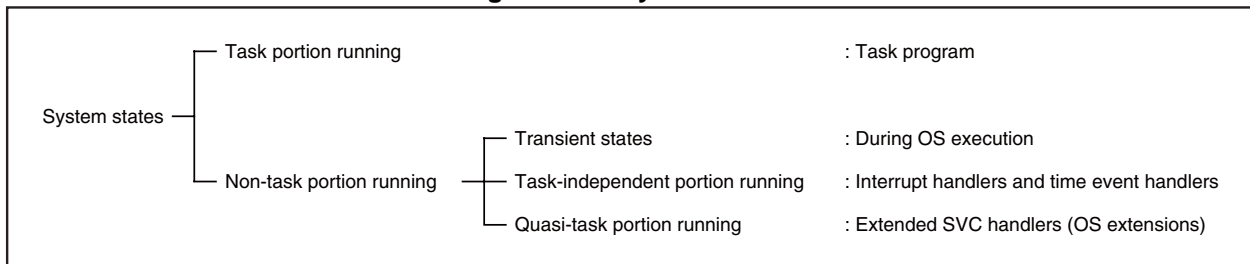
2.4 System States

This section describes the system states.

■ System States

The system states of μ T-REALOS are divided into the following categories.

Figure 2.4-1 System States



■ Task Portion Running

"Task portion running" are the states in which task programs run. This does not include states in which the OS (system calls) executes or states in which handlers execute, which are part of the "Non-task portion running" described below.

■ Non-task Portion Running

"Non-task portion running" are further subdivided into the three states of "transient states", "task-independent portion running", and "quasi-task portion running".

(1) "Transient States"

"Transient States" refer to the states in which μ T-REALOS system call processing is executed.

(2) "Task-independent Portion Running"

"Task-independent Portion Running" refer to the states in which interrupt handlers and time event handlers are executed.

(3) "Quasi-task Portion Running"

These are the states in which extended SVC handlers called from a task and device driver interface functions are executed.

■ System Calls that can be called

Except for system calls such as `tk_ret_int` and `isig_tim` that are required to be called from a "task-independent portion", all of the system calls can be called from the "task portions" and "quasi-task portions".

In contrast, "task-independent portions" execute in a context that is independent of any tasks, and do not have the concept of a task. The following system calls therefore cannot be called from the task-independent portions.

- System calls that explicitly specify the current task (calls where `tskid` is specified using the `"TSK_SELF"` macro)
- System calls that implicitly specify the current task (calls that enter a `WAITING`)

See Section "3.1 List of System Calls" of the "API Reference" for details on the system calls that can be called from each of the system states.

■ User Programs and System States

Table 2.4-1 shows the relationship between the parts of a user program and the system states.

Table 2.4-1 System States of Each Part of a User Program

User Program Component	System State
Tasks	Task portion
Extended SVC handlers	Non-task portion (quasi-task portion, task-independent portion)
Device drivers	Non-task portion (quasi-task portion)
Cyclic handlers	Non-task portion (task-independent portion)
Alarm handlers	Non-task portion (task-independent portion)
Interrupt handlers	Non-task portion (task-independent portion)
Error routines	Task portion, non-task portion (task-independent portion)

Note:

The μ T-Kernel specifications do not define `isig_tim` or error routines. These are extended functionality that is specific to μ T-REALOS.

2.5 Enabling and Disabling Dispatching and Interrupts

This section describes the dispatch enabled/disabled states and the interrupts enabled/disabled states.

■ Dispatch Enabled/disabled States

During execution of a user program, the dispatcher can either be in the dispatch disabled state or the dispatch enabled state. After system initialization, the dispatcher enters the dispatch enabled state when the initial task begins executing.

In the dispatch disabled state, the system does not switch the task that is in the RUNNING (dispatching does not occur). While in the dispatch disabled state, an error (E_CTX) occurs if a system call is made where there is a possibility of the currently running task entering the WAITING. However, interrupt handlers, cyclic handlers, and alarm handlers remain active.

The dispatch disabled/enabled states can be controlled from a user program by calling the following system calls.

- tk_dis_dsp: Enters the dispatch disabled state (disables dispatching)
- tk_ena_dsp: Enters the dispatch enabled state (enables dispatching)

■ Interrupts Enabled/disabled States

During execution of a user program, the system can either be in the interrupts disabled state or the interrupts enabled state. After system initialization, the system enters the interrupts enabled state when the initial task begins executing.

In the interrupts disabled state, the I flag in the PS register is set to 0 and all external interrupts are disabled such that control is not passed to an interrupt handler even if a hardware interrupt occurs. Furthermore, if dispatching is also disabled, then the system does not switch from the currently running task (dispatching does not occur). While in the interrupts disabled state, an error (E_CTX) occurs if a system call is made where there is a possibility of the currently running task entering the WAITING. An error (E_CTX) also occurs if a system call is made to enable or disable dispatching (tk_dis_dsp or tk_ena_dsp) while in the interrupts disabled state.

The interrupts enabled/disabled states can be controlled from a user program by calling the following macros.

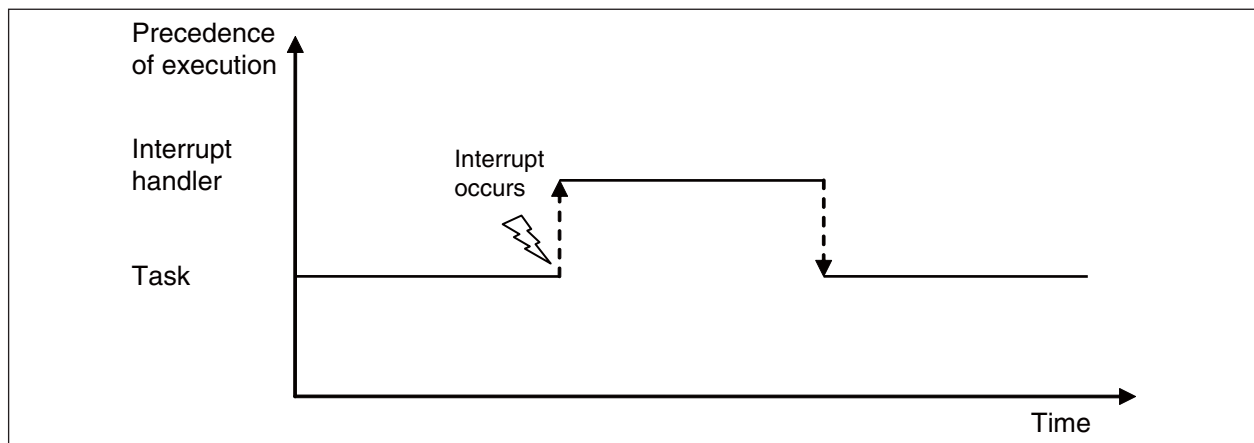
- DI: Enters the interrupts disabled state (disables interrupts)
- EI: Enters the interrupts enabled state (enables interrupts)

2.6 Precedence of Execution of Tasks and Handlers

This section describes the precedence of execution of tasks and handlers.

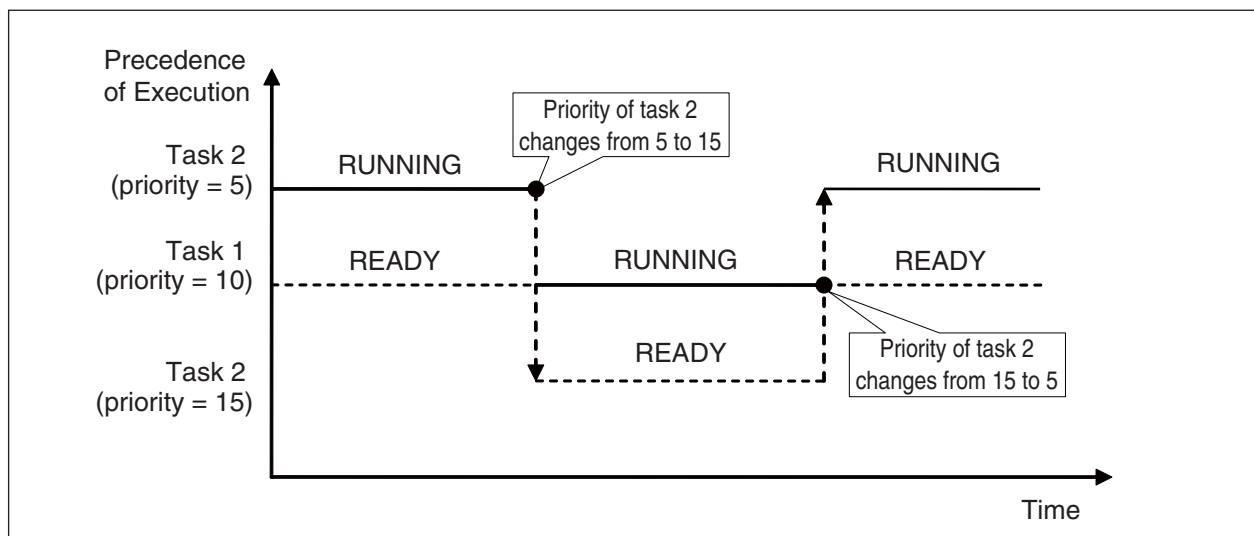
■ Precedence of Execution (Tasks vs. Interrupt Handlers and Time Event Handlers)

Handlers have a higher precedence of execution than tasks. For example, if a hardware interrupt occurs while a task is executing, the execution of the task is suspended and the interrupt handler corresponding to the interrupt is executed. When the interrupt handler finishes executing, the task resumes execution from the point where it was suspended.



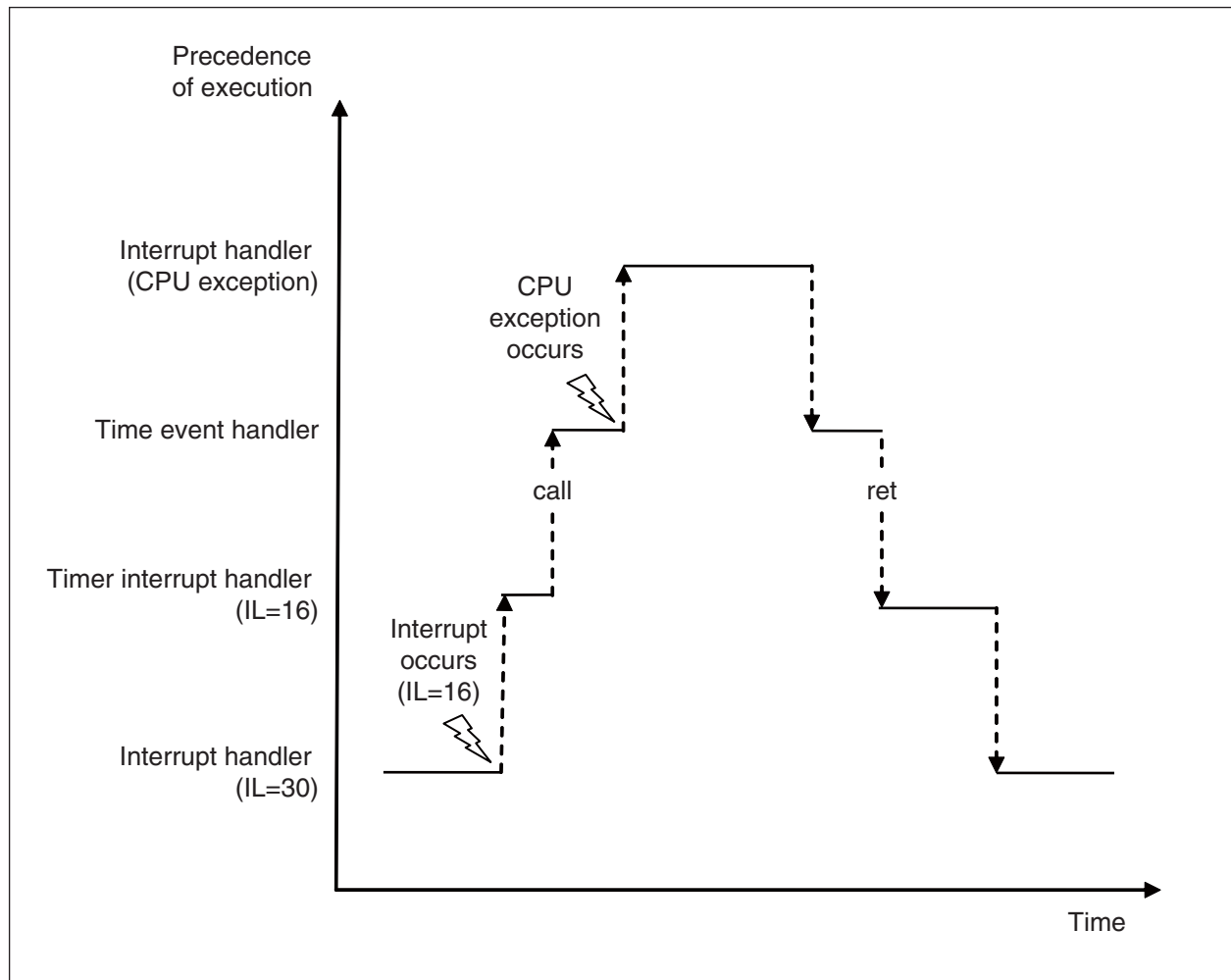
■ Precedence of Execution (Tasks vs. Tasks)

The precedence of execution of tasks executes tasks that have a higher priority first. If a task with a higher priority than the task that is currently executing enters the READY, the currently executing task is suspended and the higher priority task is executed.



■ Precedence of Execution (Handlers vs. Handlers)

In the precedence of execution of handlers, interrupt handlers for CPU exceptions execute with the highest precedence. The precedence of execution of other interrupt handlers depends on the hardware interrupt level (IL), with the interrupt handlers corresponding to interrupts that have a high interrupt level (the numerical value of the interrupt level is small) executing with precedence. Time event handlers execute as extensions of the timer interrupt handler. The precedence of execution of time event handlers therefore depends on the interrupt level of the timer interrupt.



CHAPTER 3

μ T-REALOS FUNCTIONS

This chapter describes the functions supported by μ T-REALOS.

- 3.1 Overview of μ T-REALOS Functions
- 3.2 Task Management Functions
- 3.3 Task Synchronization Functions
- 3.4 Synchronization and Communication Functions
- 3.5 Extended Synchronization and Communication Functions
- 3.6 Memory Pool Management Functions
- 3.7 Time Management Functions
- 3.8 Interrupt Management Functions
- 3.9 System State Management Functions
- 3.10 Subsystem Management Functions
- 3.11 Device Management Functions
- 3.12 Power Saving Functions
- 3.13 Configuration Functions
- 3.14 Debugging Assistance Functions

3.1 Overview of μ T-REALOS Functions

This section explains an overview of μ T-REALOS functions.

■ Overview of μ T-REALOS Functions

μ T-REALOS supports the following functions.

- Kernel
 - Task management functions
 - Task-dependent synchronization functions
 - Synchronization and communication functions (semaphores, event flags, mailboxes)
 - Extended synchronization and communication functions (mutexes, message buffers, rendezvous ports)
 - Memory pool management functions (fixed length memory pool, variable length memory pool)
 - Time management functions (system time, cyclic handlers, alarm handlers)
 - Interrupt management functions
 - System state management functions
 - Subsystem management functions
 - Device management functions
 - Power saving functions
- Configurator
 - Configuration function
- Analyzer
 - Debugging assistance functions

See "CHAPTER 3 SYSTEM CALL INTERFACE" of the "API Reference" for details on the system calls described in this chapter.

3.2 Task Management Functions

This section describes the task management functions.

■ Task Management Functions

The task management functions are functions for directly operating and referring to the state of a task. This includes functions to create and delete tasks, functions to start and end tasks, functions to change the priority of tasks, and functions to refer to the states of tasks.

Tasks are identified by an ID number that is assigned uniquely to each task. The task ID number is called the task ID.

When a task exits, the kernel does not release resources acquired by the task (semaphore resources, memory blocks, etc). However, mutex locks are released (see "3.5.1 Mutex Functions"). When a task exits, ensure that the user program releases the resources that the task acquired.

The task management functions provide the following functions using the corresponding system calls.

- Creating and deleting tasks
 - Creating a task:tk_cre_tsk
 - Deleting a task:tk_del_tsk (Deletes a task in the DORMANT)
 - tk_ext_tsk (Ends and deletes a task)
- Starting and ending tasks
 - Starting a task:tk_sta_tsk
 - Ending a task:tk_ext_tsk (Ends the current task)
 - tk_ext_tsk (Ends and deletes the current task)
 - tk_ter_tsk (Forcibly terminate another task)
- Changing the task priority :tk_chg_pri
- Referring to the state of a task :tk_ref_tsk
- Setting and referring to task registers
 - Setting task registers:tk_set_reg
 - Retrieving to task registers:tk_get_reg

3.3 Task Synchronization Functions

This section describes the task synchronization functions.

■ Task Synchronization Functions

The task synchronization functions are functions for performing synchronization by directly operating task portions.

They include functions for task sleep and wakeup, for canceling wakeup requests, for forcibly releasing task WAITING state, for changing a task portion to SUSPENDED state, for resuming a task from SUSPENDED state and for delaying execution of the invoking task.

Wakeup requests for a task are queued. That is, when it is attempted to wake up a task that is not WAITING, the wakeup request is remembered, and the next time the task is to go to a WAITING state, it does not enter that state. To realize this, the kernel maintains the number of wakeup requests that have been queued for each task. This is called the "wakeup request queuing count". When the task is started, this count is cleared to 0.

Furthermore, suspend requests for a task are nested. That is, if it is attempted to suspend a task already in SUSPEND state (including WAIT-SUSPEND state), the request is remembered, and later when it is attempted to resume the task in SUSPEND state (including WAIT-SUSPEND state), it is not resumed. To realize this, the kernel maintains the number of nested suspension requests for each task. This is called the "suspend request nesting count". When the task is started, this count is cleared to 0.

The task synchronization functions provide the following functions using the corresponding system calls.

- Task sleep and wake up task
 Task sleep:tk_slp_tsk
 Task wake up:tk_wup_tsk
- Cancelling a wakeup request :tk_can_wup
- Forcibly releasing task WAITING state :tk_rel_wai
- Suspending and resuming tasks
 Suspend request:tk_sus_tsk
 Resume suspend:tk_rsm_tsk
 tk_frsm_tsk (forced resume)
- Delaying the execution of a task :tk_dly_tsk

If the tskstat is in a state other than TTS_WAI (or TTS_WAS), tskwait and wid are both "0". Furthermore, wupcnt and suscnt are both "0" for tasks in the DORMANT.

If packet address (pk_rtsk) for returning the task status is invalid, no error check is performed and the operation is not guaranteed.

3.4 Synchronization and Communication Functions

This section describes the synchronization and communication functions.

■ Synchronization and Communication Functions

The synchronization and communication functions are functions for performing synchronization and communication between tasks using task-independent objects.

The synchronization and communication functions support the following objects.

- Semaphores
- Event flags
- Mailboxes

3.4.1 Semaphore Functions

This section describes the semaphore functions.

■ Semaphore Functions

Semaphores are objects for representing numerically data of and availability of unused resources (called the semaphore count), and for managing exclusive control and synchronization when using those resources. The semaphore functions include functions for creating and deleting semaphores, functions for acquiring and returned semaphore resources, and functions for referring to the state of a semaphore.

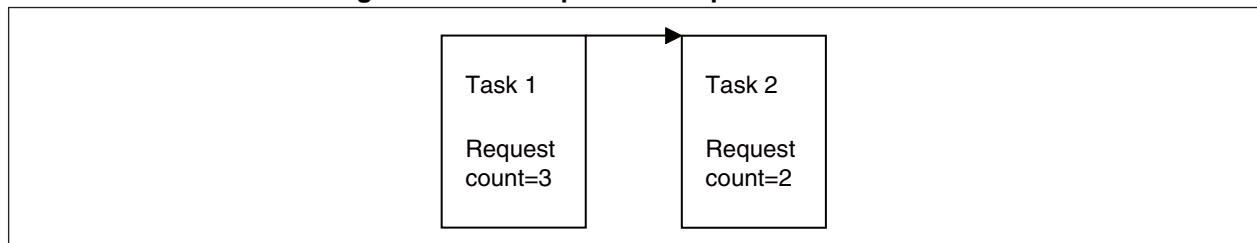
Semaphore objects are identified by an ID number. The semaphore ID number is called the semaphore ID.

Semaphores have a semaphore count and a wait queue of tasks waiting to acquire resources. When m resources are returning (the event notifier side), the semaphore count increases by m . When n resources are acquired (the event wait side), the semaphore count decreases by n . When a task attempts to acquire semaphore resources when the number of resources is insufficient (specifically, when the semaphore count reduces to a negative value), a task attempting to acquire resources goes into WAITING until the next time resources are returning. A task waiting for semaphore resources is linked to the wait queue of that semaphore. Furthermore, a maximum resource count can be configured on each semaphore to prevent too many resources from being returning. An error occurs when an attempt is made to return resources that exceed the maximum semaphore count to a semaphore (specifically, when the semaphore count increases and exceeds the maximum semaphore count).

The order of the wait queue can be selected from the two options of FIFO order (TA_TFIFO) and task priority order (TA_TPRI). Furthermore, the precedence of resource acquisition can be selected from the two options of task at head of wait queue first (TA_FIRST) or task with smallest request count first (TA_CNT). These are specified as semaphore attributes when the semaphore is created.

Figure 3.4-1 shows the situation when the semaphore count changes from 1 to 2 in a semaphore with the TA_CNT attribute. Task 1 is skipped and the resources are allocated to Task 2.

Figure 3.4-1 Example of Semaphore Wait Queue



The maximum value of the semaphore count is specified when the semaphore is created. The upper limit on the maximum value of the semaphore count is 0x7FFFFFFF. See Section "3.5.1.1 tk_cre_sem" in the "API Reference" for details.

The semaphore functions provide the following functions using the corresponding system calls.

- Creating and deleting semaphores
Creating a semaphore:tk_cre_sem
Deleting a semaphore:tk_del_sem
- Acquiring and returned semaphore resources
Returned semaphore resources:tk_sig_sem
Acquiring semaphore resources:tk_wai_sem
- Referring to the semaphore state :tk_ref_sem

3.4.2 Event Flag Functions

This section describes the event flag functions.

■ Event Flag Functions

Event flags are objects that perform synchronization by representing the presence or absence of events as bit flags. The event flag functions include functions for creating and deleting event flags, functions for setting and clearing event flags, functions for waiting for event flags, and functions for referring to the state of an event flag.

Event flag objects are identified by an ID number. The event flag ID number is called the event flag ID.

Event flags contain a bit pattern where each bit represents the presence or absence of the corresponding event, and a wait queue of tasks waiting for those event flags. Sometimes the bit pattern of an event flag is simply called the event flag. The event notifier side of the event flag is able to set or clear the specified bits in the bit pattern of the event flag. On the event waiting side of the event flag, a task is able to be WAITING state until all or some of the specified bits in the bit pattern of the event flag are set. A task waiting for event flag is linked to the wait queue of that event flag.

The conditions for release from WAITING state can specify either of the two attributes of AND wait or OR wait. These attributes specify how the release from WAITING state operates when waiting for multiple events. For the AND wait, the WAITING state is not released until all of the events are signaled, whereas for the OR wait, the WAITING state is released when even one of the events being waited for is signaled. Furthermore, it is possible to specify whether or not to clear the bits when the WAITING state is released, and there is a selection between clearing all of the bits or only clearing the bits that were matched.

In μ T-REALOS, event generation is managed using 32-bit bit patterns.

The event flag functions provide the following functions using the corresponding system calls.

- Creating and deleting event flags
 - Creating an event flag:tk_cre_flg
 - Deleting an event flag:tk_del_flg
- Setting and clearing event flag bits
 - Setting an event flag:tk_set_flg
 - Clearing an event flag:tk_clr_flg
- Waiting for an event flag :tk_wai_flg
- Referring to the state of an event flag :tk_ref_flg

3.4.3 Mailbox Functions

This section describes the mailbox functions.

■ Mailbox Functions

Mailboxes are objects that perform synchronization and communication by receiving messages that are stored in memory. The mailbox functions include functions for creating and deleting mailboxes, functions for sending and receiving messages to and from a mailbox, and functions for referring to the state of a mailbox.

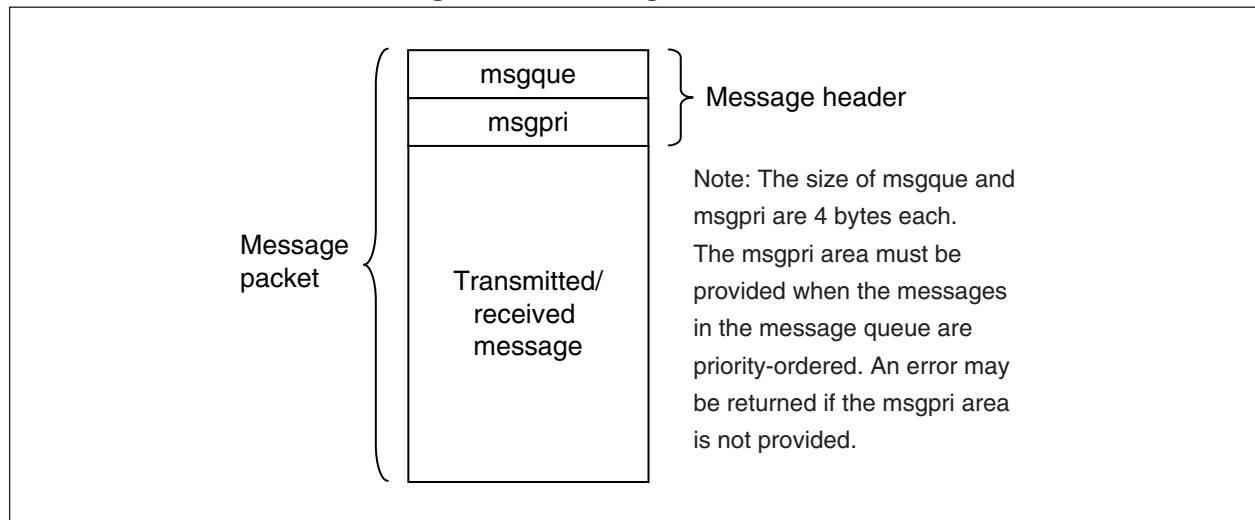
Mailbox objects are identified by an ID number. The mailbox ID number is called the mailbox ID.

Mailboxes have a message queue for storing messages that have been sent, and a wait queue for tasks that are waiting to receive a message. On the message-sending side (the event notifier side), the messages to be sent are placed in the message queue. On the message-receiving side (the event wait side), a single message is retrieved from the message queue.

If there are no messages in the message queue, the task enters a state of waiting for receipt from the mailbox until the next message is sent. Tasks that enter a state of waiting for receipt from the mailbox linked to the wait queue of that mailbox.

The information that is actually sent and received by the mailbox is only the starting address of a message in memory. This means that the contents of messages that are sent and received are not copied. The kernel manages the messages in the message queue using a linked list.

Figure 3.4-2 shows the message packet format of priority-ordered messages. The user program should allocate an area (msgque) at the top of a message being sent for the kernel to use for the linked list. This area is called the message header. Furthermore, if the message queue is ordered by message priority, an area for holding the message priority (msgpri) also needs to be reserved in the message header. The message header and the following area where the application stores the message are collectively called a message packet. System calls for sending messages to a mailbox take the starting address of the message packet (pk_msg) as a parameter. Furthermore, system calls for receiving messages from a mailbox return the starting address of the message packet as the return value.

Figure 3.4-2 Message Packet Format

The mailbox functions provide the following functions using the corresponding system calls.

- Creating and deleting mailboxes
 - Creating a mailbox:tk_cre_mbx
 - Deleting a mailbox:tk_del_mbx
- Sending to and receiving from a mailbox
 - Sending to a mailbox:tk_snd_mbx
 - Receiving from a mailbox:tk_rcv_mbx
- Referring to the state of a mailbox :tk_ref_mbx

■ Additional Notes

Because the area for the message header is allocated by the user program, the mailbox functions do not have an upper limit on the number of messages that can be placed in a message queue. Furthermore, the system calls for sending messages do not enter the WAITING state. Message packets are able to use memory blocks dynamically allocated from the fixed-size memory pool or variable-size memory pool, or statically allocated regions. Typical usage is for the sending task to allocate a memory block from the memory pool and send this as a message packet, and for the receiving task to directly release the memory block back to the memory pool after reading the contents of the message.

3.5 Extended Synchronization and Communication Functions

This section describes the extended synchronization and communication functions.

■ Extended Synchronization and Communication Functions

The extended synchronization and communication functions are functions for performing high-level synchronization and communication between tasks using task-independent objects. This includes functions for mutexes, message buffers, and rendezvous ports.

These functions support the following objects.

- Mutexes
- Message buffers
- Rendezvous ports

3.5.1 Mutex Functions

This section describes the mutex functions.

■ Mutex Functions

Mutexes are objects that perform exclusive control between tasks that use a shared resource.

The mutexes support the priority inheritance protocol and priority ceiling protocol as a mechanism to prevent priority inversion due to unlimited exclusive control. The mutex functions include functions for creating and deleting mutexes, functions for locking and unlocking mutexes, and functions for referring to the state of a mutex.

Mutex objects are identified by an ID number. The mutex ID number is called the mutex ID.

Mutexes have a state that can be locked or unlocked, and a wait queue of tasks waiting to lock the mutex. Furthermore, the kernel manages the following objects.

- The tasks that are locking each mutex
- The mutexes that are locked by each task

A task locks the mutex before using the resource. If the mutex is already locked by another task, the task waits for the mutex to become unlocked. Tasks in mutex lock waiting state are linked to the wait queue of that mutex. When the task finishes using the resource, the task releases the lock on the mutex.

The mutexes support the priority inheritance protocol by specifying `TA_INHERIT(=0x02)` for mutex attributes. And the mutexes support the priority ceiling protocol by specifying `TA_CEILING(=0x03)` for mutex attributes.

For mutexes that have the `TA_CEILING` attribute, the base priority of the task with the highest base priority from among the tasks that could lock the mutex is set as the ceiling priority when the mutex is created. An `E_ILUSE` error occurs if a task with a base priority higher than the ceiling priority of a mutex that has the `TA_CEILING` attribute attempts to lock that mutex. Furthermore, if an attempt is made to use `tk_chg_pri` to set the priority of a task that has a lock or is waiting for a lock on a mutex that has the `TA_CEILING` attribute to a higher priority than the ceiling priority of that mutex, `tk_chg_pri` returns the error `E_ILUSE`.

If these protocols are used, the current priority of a task is changed when the task operates a mutex in order to prevent unlimited priority inversion. The kernel changes the current priority of a task that has a lock on a mutex. Therefore it always equals the highest value from among the following priorities.

- The base priority of the task that is locking the mutex
- The current priority of the task that has the highest current priority from among the tasks that are waiting to lock that mutex if the task is locking a mutex that has the `TA_INHERIT` attribute
- The ceiling priority of the mutex that has the highest ceiling priority from among the mutexes being locked by that task if the task is locking a mutex that has the `TA_CEILING` attribute

If the current priority of a task that is waiting for a mutex that has the TA_INHERIT attribute is changed as a result of a mutex operation or the base priority being changed by tk_chg_pri, the current priority of the task that is locking that mutex may need to be changed. This is called transitive priority inheritance. Furthermore, if that task is waiting for another mutex that has the TA_INHERIT attribute, then transitive priority inheritance processing may be needed for the task that is locking that mutex.

The following processing is performed when the current priority of a task is changed as the result of operating on a mutex.

- If a task that has changed its priority is in a runnable state, the precedence of the task is changed based on the priority after the change (the task has the lowest precedence from among the tasks that have the same priority as the priority after the change).
- If the task that has changed its priority is linked to some kind of task priority ordered wait queue, the order within the wait queue is changed based on the priority after the change (the task has the lowest precedence from among the tasks that have the same priority as the priority after the change).
- If a task is still locking any mutexes when the task ends, the locks are released from all of those mutexes. If the task is locking multiple mutexes, those mutexes are released in order starting from the mutexes that were allocated last.

See Section "3.6.1.4 tk_unl_mtx" in the "API Reference" for specific details on the lock release process.

The mutex functions provide the following functions using the corresponding system calls.

- Creating and deleting mutexes
 Creating a mutex:tk_cre_mtx
 Deleting a mutex:tk_del_mtx
- Locking and unlocking a mutex
 Locking a mutex:tk_loc_mtx
 Unlocking a mutex:tk_unl_mtx
- Referring to the state of a mutex :tk_ref_mtx

■ Additional Notes

Mutexes that have the TA_TFIFO attribute or TA_TPRI attribute have the same functions as a semaphore with a maximum resource count of 1 (binary semaphore). However, mutexes differ in that the lock can only be released by the locking task, and the lock is automatically released when the task ends.

3.5.2 Message Buffer Functions

This section describes the message buffer functions.

■ Message Buffer Functions

Message buffers are objects that perform synchronization and communication by receiving variable-length messages. The message buffer functions include functions for creating and deleting message buffers, functions for sending and receiving messages to and from a message buffer, and functions for referring to the state of a message buffer.

Message buffer objects are identified by an ID number. The message buffer ID number is called the message buffer ID.

Message buffers have a wait queue of tasks waiting to send messages (send wait queue) and a wait queue of tasks waiting to receive messages (receive wait queue). The message buffer also has a message buffer area for storing sent messages.

On the message-sending side (the event notifier side), the messages to be sent are copied into the message buffer. If there is not enough free space in the message buffer area, the task waits for sending a message to message buffer until there is enough free space in the message buffer area. Tasks waiting to send a message to message buffer are linked to the send wait queue of that message buffer.

On the message-receiving side (the event wait side), a single message is retrieved from the message buffer. If there are no messages in the message buffer, the task waits for receiving a message from message buffer until the next message is sent. Tasks waiting for receiving a message from message buffer are linked to the receive wait queue of that message buffer.

Synchronous messaging functionality can be obtained by setting the size of the message buffer area to zero. This means that both the sending task and the receiving task wait for each-other to make the system call, and pass the message between them when both tasks have made the system call.

The message buffer functions provide the following functions using the corresponding system calls.

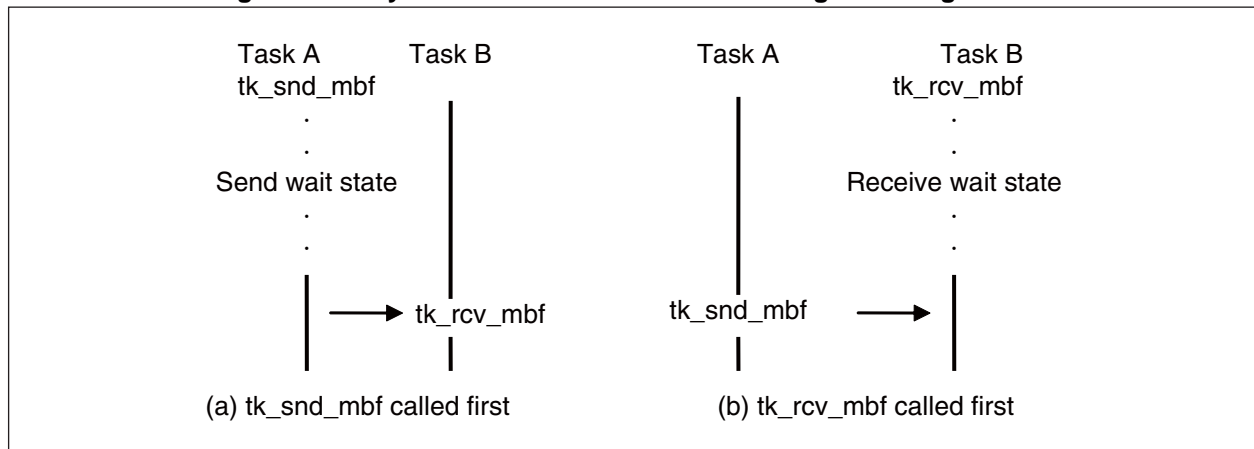
- Creating and deleting message buffers
 - Creating a message buffer: `tk_cre_mbf`
 - Deleting a message buffer: `tk_del_mbf`
- Sending and receiving messages to and from a message buffer
 - Sending to a message buffer: `tk_snd_mbf`
 - Receiving from a message buffer: `tk_rcv_mbf`
- Referring to the state of a message buffer : `tk_ref_mbf`

■ Additional Notes

Figure 3.5-1 shows the operation of a message buffer when the size of the message buffer area is set to 0. In this diagram, Task A and Task B are executing asynchronously.

- If Task A calls `tk_snd_mbf` first, Task A enters the wait state until Task B calls `tk_rcv_mbf`. In this case, Task A enters the send wait state to message buffer (Figure 3.5-1 (a)).
- If Task B calls `tk_rcv_mbf` first, Task B enters the wait state until Task A calls `tk_snd_mbf`. In this case, Task B enters the receive wait state from message buffer (Figure 3.5-1 (b)).
- The message is passed from Task A to Task B when Task A has called `tk_snd_mbf` and Task B has called `tk_rcv_mbf`. After this, both tasks enter a runnable state.

Figure 3.5-1 Synchronous Communication Using a Message Buffer



Tasks that are waiting to send to a message buffer send messages in the order that they are linked to the wait queue. For example, consider the situation where Task A which is attempting to send a 40-byte message to the message buffer and Task B which is attempting to send a 10-byte message, and these tasks are linked into the wait queue in this order. Now suppose that 20 bytes of free space are created by another task receiving a message. In this situation, Task B is unable to send its message until Task A sends its message.

Message buffers are different from mailboxes because they transfer variable-length messages by copying.

3.5.3 Rendezvous Port Functions

This section describes the rendezvous port functions.

■ Rendezvous Port Functions

Rendezvous ports provide intertask synchronous communication functionality, and support a single sequence where one task requests processing of another task and the other task then returns the processing result to the first task. The object that synchronizes the waiting of both tasks is called a rendezvous port. Although the rendezvous port functions can be used to implement a typical client/server model of intertask communication, they provide a synchronous communication model that is more flexible than the client/server model.

The rendezvous port functions include functions for creating and deleting rendezvous ports, functions for requesting processing from a rendezvous port (rendezvous call), functions for accepting processing requests from a rendezvous port (rendezvous accept), functions for returning processing results (rendezvous complete), functions for forwarding received processing requests to another rendezvous port (rendezvous forward), and functions for referring to rendezvous ports and rendezvous states.

Rendezvous port objects are identified by an ID number. The rendezvous port ID number is called the rendezvous port ID.

The task that makes the processing request to the rendezvous port (the client-side task) specifies the rendezvous port, rendezvous parameters, and a message containing information regarding the processing being requested (called the call message) and performs the rendezvous call. The task that accepts processing requests from the rendezvous port (the server-side task) specifies the rendezvous port and rendezvous parameters to accept the rendezvous.

The rendezvous parameters are specified as a bit pattern. For a given rendezvous port, a rendezvous is established if the result of logical bitwise ANDing of the rendezvous parameters bit pattern of the calling tasking and the rendezvous parameters bit pattern of the accepting task is non-zero. The task that calls the rendezvous enters the rendezvous call wait state until the rendezvous is established. Similarly, the task that receives a rendezvous enters the rendezvous accept wait state until the rendezvous is established.

Once the rendezvous is established, the call message is passed from the task that called the rendezvous to the task that accepted the rendezvous. The task that called the rendezvous then enters the rendezvous completion wait state and waits for the requested processing to finish. The task that accepted the rendezvous is released from the wait state and performs the requested processing. Once the task that accepted the rendezvous has finished the requested processing, the results of the processing are passed to the calling task in the form of a response message and the rendezvous finishes. At this time, the task that called the rendezvous is released from the rendezvous completion wait state.

Rendezvous ports have a call wait queue for linking tasks in the rendezvous call wait state, and a receive wait queue for linking tasks in the rendezvous accept wait state. Furthermore, after the rendezvous is established, both of the rendezvousing tasks are disconnected from the rendezvous port. This means that rendezvous ports do not have a wait queue for linking tasks in the rendezvous completion wait state. Furthermore, rendezvous ports do not have information about accepting rendezvous and tasks that are currently executing a processing request.

The kernel allocates object numbers for identifying rendezvous that are established at the same time. The rendezvous object number is called the rendezvous number. The upper 16 bits of the rendezvous number is the task ID of the task that called the rendezvous, and the lower 16 bits is a sequential number that is incremented by 1 for each rendezvous accepted. This means that even when rendezvous are called by the same task, different rendezvous numbers are allocated to the first rendezvous and the second rendezvous.

The rendezvous port functions provide the following functions using the corresponding system calls.

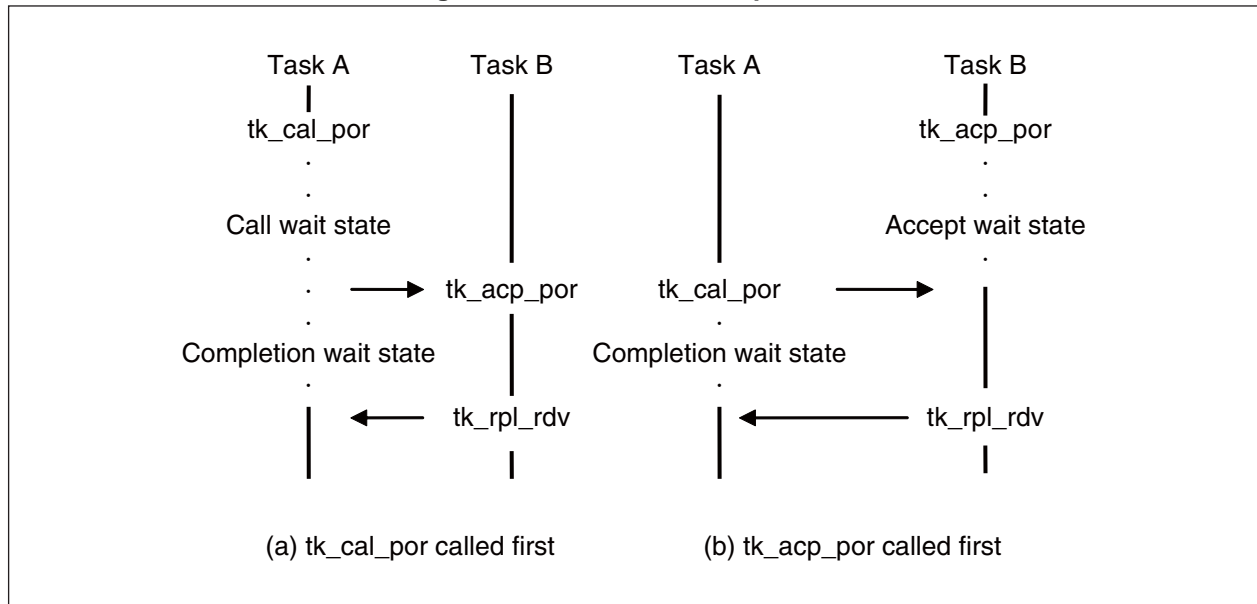
- Creating and deleting rendezvous ports
 - Creating a rendezvous port: `tk_cre_por`
 - Deleting a rendezvous port: `tk_del_por`
- Requesting processing, accepting, and replying to a rendezvous port
 - Requesting processing to a rendezvous port: `tk_cal_por`
 - Accepting processing from a rendezvous port: `tk_acp_por`
 - Replying to a rendezvous port: `tk_rpl_rdv`
- Forwarding a rendezvous port `:tk_fwd_por`

■ Additional Notes

Figure 3.5-2 shows the rendezvous operation. In this diagram, Task A and Task B are executing asynchronously.

- If Task A calls `tk_cal_por` first, Task A enters the wait state until Task B calls `tk_acp_por`. In this case, Task A enters the rendezvous call wait state (Figure 3.5-2 (a)).
- If Task B calls `tk_acp_por` first, Task B enters the wait state until Task A calls `tk_cal_por`. In this case, Task B enters the rendezvous accept wait state (Figure 3.5-2 (b)).
- When both Task A has called `tk_cal_por` and Task B has called `tk_acp_por`, a rendezvous is established. Task A is left in the wait state and Task B is released from the wait state. Task A now enters the rendezvous completion wait state.
- When Task B calls `tk_rpl_rdv`, Task A is released from the wait state. After this, both tasks enter a runnable state.

Figure 3.5-2 Rendezvous Operation



3.6 Memory Pool Management Functions

This section describes the memory pool management functions.

■ Memory Pool Management Functions

The "memory pool management functions" are functions for managing memory pools and allocating regions of memory (memory blocks) for use by user programs.

The available memory pools are the fixed-size memory pool and the variable-size memory pool. These two memory pools are separate objects and are accessed by different system calls. The size of memory blocks obtained from the fixed-size memory pool is fixed whereas arbitrary sizes can be specified for memory blocks obtained from the variable-size memory pool.

The memory pool management functions support the following types of memory pools.

- Fixed-size memory pool
- Variable-size memory pool

3.6.1 Fixed-size Memory Pool Functions

This section describes the fixed-size memory pool functions.

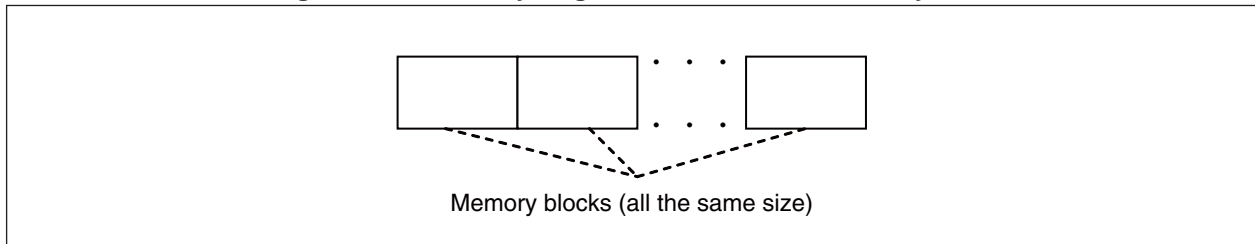
■ Fixed-size Memory Pool Functions

Fixed-size memory pool are objects that perform dynamic management of fixed size memory blocks. The fixed-size memory pool functions include functions for creating and deleting fixed length memory pools, functions for getting and returning memory blocks from a fixed-size memory pool, and functions for referring to the state of a fixed-size memory pool.

fixed-size memory pool objects are identified by an ID number. The fixed-size memory pool ID number is called the fixed-size memory pool ID.

Fixed-size memory pool have a region of memory that is used as the fixed-size memory pool (this is called the fixed-size memory pool region, or simply the memory pool region), and a wait queue for tasks that are waiting to get a memory block. If there is no free space in the memory pool region, a task that gets a memory block from a fixed-size memory pool enters the fixed length memory block acquisition wait state until the next memory block is returned. Tasks that enter the fixed length memory block acquisition wait state are linked to the wait queue of that fixed-size memory pool.

Figure 3.6-1 Memory Region of a Fixed-size Memory Pool



The fixed-size memory pool functions provide the following functions using the corresponding system calls.

- Creating and deleting fixed-size memory pool
 - Creating a fixed-size memory pool: `tk_cre_mpf`
 - Deleting a fixed-size memory pool: `tk_del_mpf`
- Getting and returning fixed-length memory blocks
 - Getting a fixed-length memory block: `tk_get_mpf`
 - Returning a fixed-length memory block: `tk_rel_mpf`
- Referring to the state of a fixed-length memory block : `tk_ref_mpf`

3.6.2 Variable-size Memory Pool Functions

This section describes the variable-size memory pool functions.

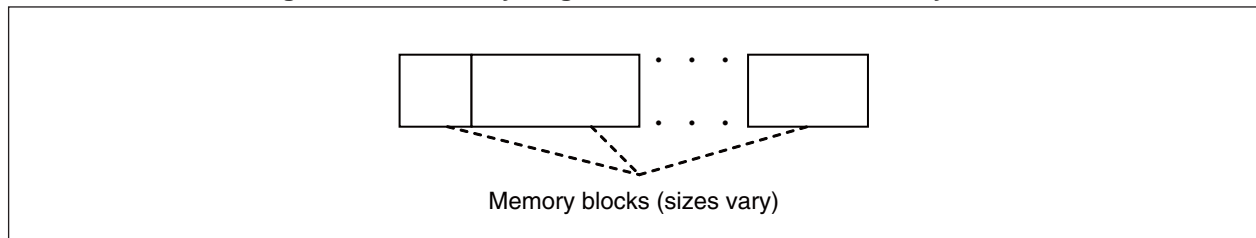
■ Variable-size Memory Pool Functions

Variable-size memory pool are objects for dynamically managing arbitrary size memory blocks. The variable-size memory pool functions include functions for creating and deleting variable-size memory pool, functions for getting and returning memory blocks from a variable-size memory pool, and functions for referring to the state of a variable-size memory pool.

variable-size memory pool objects are identified by an ID number. The variable-size memory pool ID number is called the variable-size memory pool ID.

Variable-size memory pool have a region of memory that is used as the variable-size memory pool (this is called the variable-size memory pool region, or simply the memory pool region), and a wait queue for tasks that are waiting to get a memory block. If there is insufficient free space in the memory pool region when a task gets a memory block from the variable-size memory pool, the task enters the variable length memory block acquisition WAITING until a memory block of sufficient size is returned. Tasks that enter the variable length memory block acquisition WAITING are linked to the wait queue of that variable-size memory pool.

Figure 3.6-2 Memory Region of a Variable-size Memory Pool



The variable-size memory pool functions provide the following functions using the corresponding system calls.

- Creating and deleting variable-size memory pool
 - Creating a variable-size memory pool: `tk_cre_mpl`
 - Deleting a variable-size memory pool: `tk_del_mpl`
- Getting and returning variable-length memory blocks
 - Getting a variable-length memory block: `tk_get_mpl`
 - Returning a variable-length memory block: `tk_rel_mpl`
- Referring to the state of a variable-length memory block :`tk_ref_mpl`

3.7 Time Management Functions

This section describes the time management functions.

■ Time Management Functions

The time management functions are functions for performing time-dependent processing. The functions include functions for system time management, cyclic handlers, and alarm handlers. Cyclic handlers and alarm handlers are collectively referred to as time event handlers.

The following functions are supported.

- System time management
- Cyclic handlers
- Alarm handlers

3.7.1 System Time Management Functions

This section describes the system time management functions.

■ System Time

The system time is represented by the accumulated number of milliseconds since the 1st January 1985 (GMT). For example, a system time value of 0 represents 12:00:00 AM on 1st January 1985 (GMT). A system time value of 1000 represents 12:00:01 AM on 1st January 1985 (GMT). Because μ T-REALOS does not have a function to automatically set the current time when the system starts, the current time needs to be set by the user program.

■ Updating the System Time

In μ T-REALOS, the user program is required to update the system time. μ T-REALOS therefore provides `isig_tim` for this purpose. The system time is increased by one by calling this system call. The `isig_tim` system call is specific to μ T-REALOS.

Because the resolution of the system time is 1 ms, an interval timer is typically made to generate an interrupt at an interval of 1 ms, and the system time is updated by calling `isig_tim` from that interrupt handler.

- Updating the system time `:isig_tim`

■ Setting and Getting the System Time

μ T-REALOS provides the following system calls for getting and setting the system time.

- Setting and getting the system time
 - Setting the system time: `tk_set_tim`
 - Getting the system time: `tk_get_tim`

■ Getting the System Uptime

The amount of time that has elapsed since the system was started is called the system uptime. The system uptime can be retrieved using the following system call.

- Getting the system uptime `:tk_get_otm`

The system uptime differs from the system time because it is not affected by setting the system time using `tk_set_tim`.

3.7.2 Cyclic Handler Functions

This section describes the cyclic handler functions.

■ Cyclic Handler Functions

Cyclic handlers are time event handlers that activate at a fixed period. The cyclic handler functions include functions for creating and deleting cyclic handlers, functions for starting and stopping the operation of cyclic handlers, and functions for referring to the state of a cyclic handler.

Cyclic handler objects are identified by an ID number. The cyclic handler ID number is called the cyclic handler ID.

Cyclic handlers can either be in the operating state or the non-operating state. When a cyclic handler is in the non-operating state, the cyclic handler is not activated even when the time when the cyclic handler is supposed to activate is reached, and only the time when the handler should next activate is set. When the system call to start the operation of the cyclic handler (`tk_sta_cyc`) is called, the cyclic handler is placed in the operating state and the time when the cyclic handler should next activate is reset if necessary. When the system call to stop the operation of the cyclic handler (`tk_stp_cyc`) is called, the cyclic handler changes to the non-operating state. After the cyclic handler is created, either the operating or non-operating state is determined by the cyclic handler attributes.

The activation phase of a cyclic handler is determined by the time when the cyclic handler first activates, which is specified as the relative time from the time when the system call to create the cyclic handler is called. The activation interval of a cyclic handler is determined by a relative time that specifies the time when the cyclic handler should next activate based on the time when the cyclic handler should have activated (not the time when the handler actually activated).

The activation interval and activation phase for each cyclic handler can be set when the cyclic handler is created. When a cyclic handler is operating, the kernel determines the time when the cyclic handler should next activate from the specified activation interval and activation phase. When the cyclic handler is created, the time when the handler should next activate is set to the cyclic handler creation time plus the activation phase. When the time when the cyclic handler should activate is reached, the cyclic handler is activated with the extended information (`exinf`) of that cyclic handler as a parameter. In this case, the time when the handler should next activate is set to the time when the cyclic handler should have activated plus the activation interval. When the operation of a cyclic handler is started, the time when the handler should next activate may need to be reset.

If the time that is longer than that of the activation interval is specified to an activation phase, the cyclic handler will not activate until the time specified by the activation phase has elapsed. For example, if the activation interval is 100 ms and the activation phase is 200 ms, the cyclic handler will first activate 200 ms later, and then after $200 + 100 \times (n-1)$ ms have elapsed.

The cyclic handler functions provide the following functions using the corresponding system calls.

- Creating and deleting cyclic handlers
Creating a cyclic handler: `tk_cre_cyc`

Deleting a cyclic handler:tk_del_cyc

- Starting and stopping the operation of a cyclic handler

Starting the operation of a cyclic handler:tk_sta_cyc

tk_cre_cyc

(create and start operation by specifying TA_STA)

Stopping the operation of a cyclic handler:tk_stp_cyc

- Referring to the state of a cyclic handler :tk_ref_cyc

See Section "4.6 Period Handler" for details on writing cyclic handlers.

3.7.3 Alarm Handler Functions

This section describes the alarm handler functions.

■ Alarm Handler Functions

Alarm handlers are time event handlers that activate at a specified time. The alarm handler functions include functions for creating and deleting alarm handlers, functions for starting and stopping the operation of alarm handlers, and functions for referring to the state of an alarm handler.

Alarm handler objects are identified by an ID number. The alarm handler ID number is called the alarm handler ID.

The time when an alarm handler activates (this is called the alarm handler activation time) can be set for each alarm handler. When the alarm handler activation time is reached, the alarm handler is activated with the extended information (exinf) of that alarm handler as a parameter.

Immediately after an alarm handler is created, the alarm handler activation time is not set and the alarm handler operation is stopped. When the system call to start the operation of an alarm handler (tk_sta_alm) is called, the alarm handler activates after the specified relative time. When the system call to stop the operation of an alarm handler (tk_stp_alm) is called, the alarm handler activation time setting is cleared. In addition, when an alarm handler activates, the alarm handler activation time setting is cleared and the alarm handler stops operating.

The alarm handler functions provide the following functions using the corresponding system calls.

- Creating and deleting alarm handlers
 - Creating an alarm handler:tk_cre_alm
 - Deleting an alarm handler:tk_del_alm
- Starting and stopping the operation of an alarm handler
 - Starting the operation of an alarm handler:tk_sta_alm
 - Stopping the operation of an alarm handler:tk_stp_alm
- Referring to the state of an alarm handler :tk_ref_alm

See Section "4.7 Alarm Handler" for details on writing alarm handlers.

3.8 Interrupt Management Functions

This section describes the interrupt management functions.

■ Interrupt Management Functions

The interrupt management functions are functions for performing operations such as defining handlers for external interrupts and CPU exceptions, and controlling interrupts.

The interrupt management functions provide the following functions using the corresponding system calls.

- Managing interrupt handlers

Defining an interrupt handler: `tk_def_int`

Returning from an interrupt handler: `tk_ret_int`

Interrupt handlers are handled in the task-independent portion. Although system calls can be called from the task-independent portion using the same format as the task portion, the following limitations are placed on system calls that are called from the task-independent portion.

- System calls that specify the current task and system calls that enter a WAITING state internally cannot be called and produce an error.

During execution in the task-independent portion, if a dispatch request is made during the processing of a system call, the dispatch is delayed until the system leaves the task-independent portion. This is called delayed dispatch.

See Section "4.8 Interrupt Handler" for details on writing interrupt handlers.

- CPU interrupt control

Disabling all external interrupts: `DI`

Enabling all external interrupts: `EI`

Retrieving the interrupts disabled state prior to calling `DI`: `isDI`

These functions manipulate the CPU registers to set the interrupts enabled/disabled. `DI`, `EI`, and `isDI` cannot be called from the task-independent portion or from a state where dispatch and interrupts are disabled.

3.9 System State Management Functions

This section describes the system state management functions.

■ System State Management Functions

The system state management functions are functions for changing and referring to the state of the system. This includes functions for rotating the precedence of tasks, functions for referring to the task ID of the executing state, functions for disabling and enabling task dispatch, functions for referring to context and system states, and functions for referring to the version of the kernel.

The system state management functions provide the following functions using the corresponding system calls.

- Rotating the task precedence :tk_rot_rdq
- Referring to the task ID of the running state:tk_get_tid
- Disabling and enabling dispatch
 - Disabling dispatch:tk_dis_dsp
 - Enabling dispatch:tk_ena_dsp
- Referring to the system state :tk_ref_sys
- Referring to the kernel version :tk_ref_ver

3.10 Subsystem Management Functions

This section describes the subsystem management functions.

■ Subsystem Management Functions

The subsystem management functions only consist of the extended SVC handlers for accepting requests. The subsystem management functions provide the following functions using the corresponding system calls.

- Defining a subsystem `:tk_def_ssy`
- Referring to information about the defined subsystems: `tk_ref_ssy`

3.11 Device Management Functions

This section describes the device management functions.

■ Device Management Functions

The device management functions provide a common API for handling different devices and include functions for performing device-related operations such as registering and deleting devices, accessing device data, and retrieving device information.

The device management functions provide the following functions using the corresponding system calls.

- System calls

Registering a device:tk_def_dev

Retrieving device initialization information:tk_ref_idv

Opening a device:tk_opn_dev

Closing a device:tk_cls_dev

Starting a device read:tk_rea_dev

Reading a device synchronously:tk_srea_dev

Starting a device write:tk_wri_dev

Writing a device synchronously:tk_swri_dev

Waiting for a device request to finish:tk_wai_dev

Suspending a device:tk_sus_dev

Retrieving the device name:tk_get_dev

Retrieving device information:tk_ref_dev

:tk_oref_dev

Retrieving a list of registered devices:tk_lst_dev

Sending driver request events to a device:tk_evt_dev

The system calls that can be called depending on the device registration and open state are as follows.

Device registered	Device opened	Callable system calls
No	-	tk_def_dev
Yes	No	tk_opn_dev, tk_ref_idv, tk_get_dev, tk_ref_dev, tk_lst_dev, tk_sus_dev, tk_def_dev
	Yes	All system calls of the device management function

The specifications for the interface between the device drivers and the kernel are defined by the device driver interface. The device driver interface defines the device processing functions that are called from the kernel, the format of data passed between the kernel and the device driver, etc.

Because the device driver interface is supported by all μ T-Kernel specification OSs, device drivers that were created in compliance with device driver interface have improved portability between μ T-Kernel specification OSs.

See "APPENDIX C: Device Driver Interface" of the "API Reference" for details on the device driver interface.

3.12 Power Saving Functions

This section describes the power saving functions.

■ Power Saving Functions

The μ T-REALOS has a power saving function where a user-defined power saving routine is called by the kernel when all of the tasks have stopped running and have switched to the idle state. This is called the power saving function.

The processing performed by the power saving routine can be written freely by the user to suit the target hardware. This power saving routine is useful because it is defined using the static API of the configurator. See "3.13 Configuration Functions" for details on how to define the power saving routine and to "4.10 Power Saving Routine" for details on writing the power saving routine.

3.13 Configuration Functions

This section describes the configuration functions.

■ Configuration Functions

The configuration functions provide the functionality for the user to define the configuration of the kernel, such as upper limits on the number of resources used by the kernel, and to configure the internal kernel management data based on this. The amount of memory used by the kernel can be reduced by optimizing the kernel configuration to suit the user program. This also provides functions for statically registering user program modules such as interrupt handlers and the initial routine.

The μ T-REALOS development tool for executing configurations is called the "Configurator". Furthermore, the macros that are provided for defining the kernel configuration are called "configuration macros" and the declaration statements for registering user program modules are called the "static API".

When a configuration is executed, the configuration definition macros and static API are written in a text format file called the "configuration file", and the Configurator is executed with this as the input file.

Configurator takes the configuration file as the input file and outputs the kernel configuration file in the SOFTUNE language tool library format. The kernel configuration file is linked with the user program when an object of the executable format is created.

Because configuration files are normally generated automatically by editing the configuration using the GUI screen of SOFTUNE Workbench, there is no need to be aware of the syntax of the configuration definition macros and static API. Furthermore, kernel configuration is automatically performed when SOFTUNE Workbench is used as the build environment. See Section "5.3 Setting of Configuration" for details on editing the configuration using the GUI screens.

■ Configuration Definition Macros

The configuration definition macros are macros provided for user-defined kernel configurations. A list of the configuration definition macros is given below.

Table 3.13-1 List of Configuration Definition Macros

Function type	Name	Meaning	Range of values (default values shown in bold)
Priority definitions	_KERNEL_MAX_TSKPRI	Maximum task priority	1 to 1024
	_KERNEL_INIT_TSKPRI	Initial task priority	1 to 1024
	_KERNEL_MAX_SSPRI	Maximum subsystem priority	1 to 16
Function selection	_KERNEL_USE_TKDEFINT	Use or not use tk_def_int (1 means use)	0 or 1
	_KERNEL_USE_IMALLOC	Use or not use the heap area (1 means use)	0 or 1
	_KERNEL_REALMEMSZ	Size of the heap area	Any value
Maximum number of each object	_KERNEL_MAX_TSK	Maximum number of tasks	1 to 32767
	_KERNEL_MAX_SEM	Maximum number of semaphores	0 to 32767
	_KERNEL_MAX_FLG	Maximum number of event flags	0 to 32767
	_KERNEL_MAX_MBX	Maximum number of mailboxes	0 to 32767
	_KERNEL_MAX_MTX	Maximum number of mutexes	0 to 32767
	_KERNEL_MAX_MBF	Maximum number of message buffers	0 to 32767
	_KERNEL_MAX_POR	Maximum number of rendezvous ports	0 to 32767
	_KERNEL_MAX_MPF	Maximum number of fixed-size memory pool	0 to 32767
	_KERNEL_MAX_MPL	Maximum number of variable-size memory pool	0 to 32767
	_KERNEL_MAX_CYC	Maximum number of cyclic handlers	0 to 32767
	_KERNEL_MAX_ALM	Maximum number of alarm handlers	0 to 32767
	_KERNEL_MAX_SSY	Maximum number of subsystems	0 to 255
	_KERNEL_MAX_REGDEV	Maximum number of registered devices	0 to 255
	_KERNEL_MAX_OPNDEV	Maximum number of open devices	0 to 255
	_KERNEL_MAX_REQDEV	Maximum number of device requests	0 to 255
Size specified	_KERNEL_SYS_STKSIZE	System stack size	128 to 4294967292
	_KERNEL_INIT_TSKSTKSZ	Initial task stack size	128 to 4294967292

The value of `"_KERNEL_MAX_TSKPRI"` and the value of `"_KERNEL_INIT_TSKPRI"` are required to satisfy the following condition.

value of `"_KERNEL_MAX_TSKPRI"` \geq value of `"_KERNEL_INIT_TSKPRI"`

`_KERNEL_REALMEMSZ` specifies the size of the heap area. If the `TA_USERBUF` is not specified upon creation of the task, the message buffer, or the memory pool, the task stack, the message buffer area, or the memory pool can be automatically got from the heap area.

If the definition of any of the configuration definition macros is omitted, the minimum value that can be taken is selected as the default value. For example, if the definition of `"_KERNEL_MAX_TSKPRI"` is omitted, the maximum value of the task priority is set to "1". Similarly, if the definition of `"_KERNEL_MAX_SEM"` is omitted, the maximum number of semaphores is set to "0".

If the maximum number of an object is "0", that object cannot be used. For example, if the maximum number of semaphores is set to "0" and the user program contains semaphore-related system calls, an undefined error occurs for the semaphore-related system calls when the user system is built.

The configuration definition macros are written in the configuration file using the following syntax. Normally, however, these values are edited from the "CFG" tab in the project window of SOFTUNE Workbench (see "5.3 Setting of Configuration").

[Configuration Definition Macro Syntax]

Configuration definition macro	Defined value
Example)	
<code>_KERNEL_MAX_TSK</code>	256
<code>_KERNEL_INIT_TSKSTKSZ</code>	0x1000

Note:

The maximum number of tasks defined by `"_KERNEL_MAX_TSK"` includes the initial task that is created within the kernel. Therefore, when the number of tasks that a user program creates is N, define the maximum number of tasks as (N+1) or more.

Furthermore, because the device management function uses the following objects, set the value for the maximum number of objects to "number used by user program + number used by device management".

- Semaphores :One used for each device opened
 - Message buffers :One used by all device management functions
 - Event flags :One used by all device management functions
-

■ Static API

The interface for the user program modules that are statically defined by the Configurator is called the static API.

The initial routine, interrupt handlers, error routine, and power saving routine can be registered in the static API. Although the initial routine, error routine, and power saving routine can only be registered in the static API, interrupt handlers can also be registered from a user program using `tk_def_int`.

A list of the static API is shown in Table 3.13-2.

Table 3.13-2 List of Static API

Name	Function
ATT_INI	Defines the initial routine
DEF_INH	Defines an interrupt handler
VATT_ERR	Defines the error routine
VDEF_PSR	Defines a power saving routine

The static API is written in the configuration file using the following syntax. Normally, however, these definitions are edited from the "CFG" tab in the project window of SOFTUNE Workbench (see "5.3 Setting of Configuration").

[Static API Syntax]

- ATT_INI
ATT_INI({Attributes, Extended Information, Entry Point});
- DEF_INH
DEF_INH(Interrupt Number, {Attributes, Entry Point});
- VATT_ERR
VATT_ERR({Attributes, Entry Point});
- VDEF_PSR
VDEF_PSR({Attributes, Entry Point});

Example)

```
ATT_INI({ TA_HLNG, 0, uint});
DEF_INH(35, { TA_HLNG, inthdr});
VATT_ERR({TA_HLNG, uerr});
VDEF_PSR({TA_HLNG, pow_down});
```

■ Running the Configurator

The configurator is automatically executed if the build or make menu item is selected from a SOFTUNE Workbench project.

Configurator is located in the "bin" folder under the μ T-REALOS installation folder and has the name "ftcfs.exe". Use the following syntax to start the configurator manually or from a batch procedure.

```
ftcfs -f file_name -cpu cpu_name -out path [ -V ] [ -g ] [ -cif cif_name ]
```

-f file_name	Specifies the configuration file name as the file_name. This parameter cannot be omitted.
-cpu cpu_name	Specifies the target CPU. This parameter cannot be omitted.
-out path	Specifies the output folder for the kernel configuration file that is finally output by the configurator as "path". This parameter cannot be omitted.
-V	Outputs the configurator startup message. This also applies to the tools that are called by configurator.
-g	Outputs debugging information.
-cif cif_name	Specifies the CPU information file name as "cif_name". If this parameter is omitted, the file "[SOFTUNE Installation Folder]\lib\911\911.csv" is used as the CPU information file.

- Startup example

```
ftcfs -f C:\smpsys\system.tcf -cpu MB91403 -out C:\smpsys
```

Note:

Indicates that the elements inside the [] may be omitted.

3.14 Debugging Assistance Functions

This section describes the analyzer provided by μ T-REALOS for assisting in the debugging user programs.

■ Overview of the Debugging Assistance Functions

User programs are debugged by using the SOFTUNE Workbench debugger (referred to as the SOFTUNE debugger in this manual) operated from the Windows GUI screen. In μ T-REALOS, the analyzer is provided as a plug-in tool for the SOFTUNE debugger. The analyzer contains the following functions for assisting in the debugging of user programs.

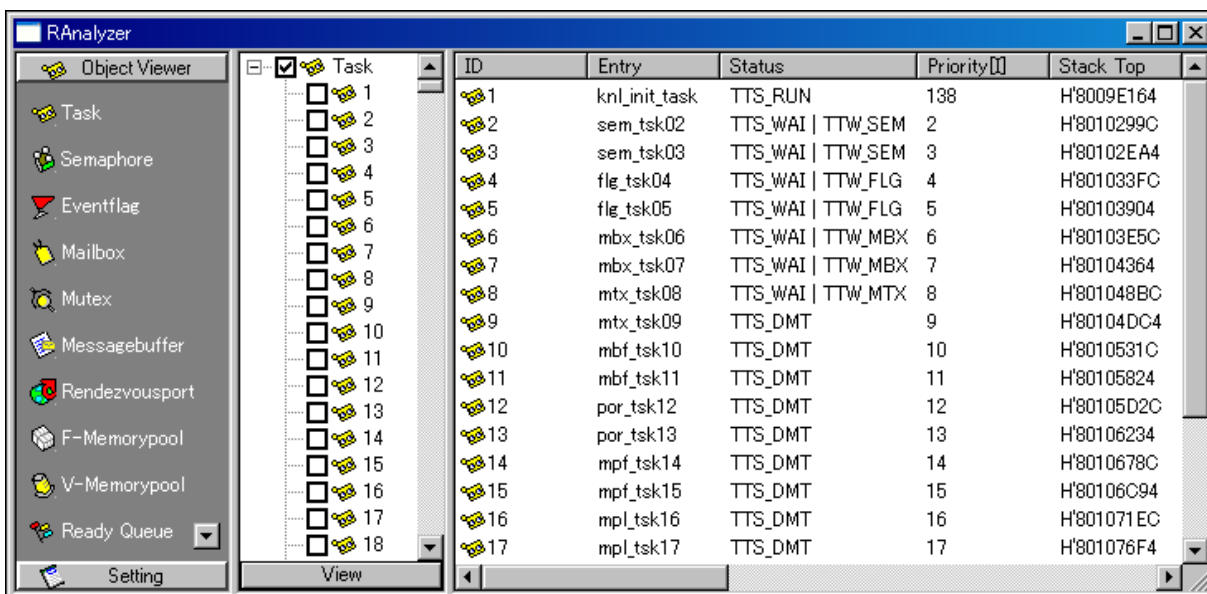
- Object list display
- OS breakpoints
- Logs
- Issuing system calls
- Stack information
- Task context display

The following sections describe these functions. See the "Analyzer Guide" for details on these functions and how to use them.

■ Object List Display

Displays a list of the ID numbers and states of the objects created by a user program categorized by object type. Figure 3.14-1 shows a screen example of the object list display.

Figure 3.14-1 Screen Example of Object List Display



■ OS Breakpoints

OS breakpoints is a function that can set breakpoints on separate tasks. When multiple tasks are sharing the same code (shared functions etc.), this enables breakpoints that are triggered when a particular task runs that code. For example, if Task 1, Task 2, and Task 3 all call the common function "comm_func()", a breakpoint can be set for when Task 2 calls "comm_func()".

Furthermore, breakpoints can be set using the following conditions on a per-task basis.

- When a task accesses specific data.
- When a task gains or loses the execution right.
- At the entry point or exit point of system calls called from a task.

■ Logs

During the execution of a user program, a log of the operation of the program can be acquired, and the contents of the log can be displayed in time-sequence in a variety of formats. This allows the operation of a user program to be analyzed easily.

The information that can be captured in the log is as follows. You can specify whether or not to capture this information by user definition.

- Interrupt handler start/stop
- Timer interrupt handler start/stop
- Dispatch start/stop
- System call start/stop
- User-specified events

The following formats are available for the display format of the logs.

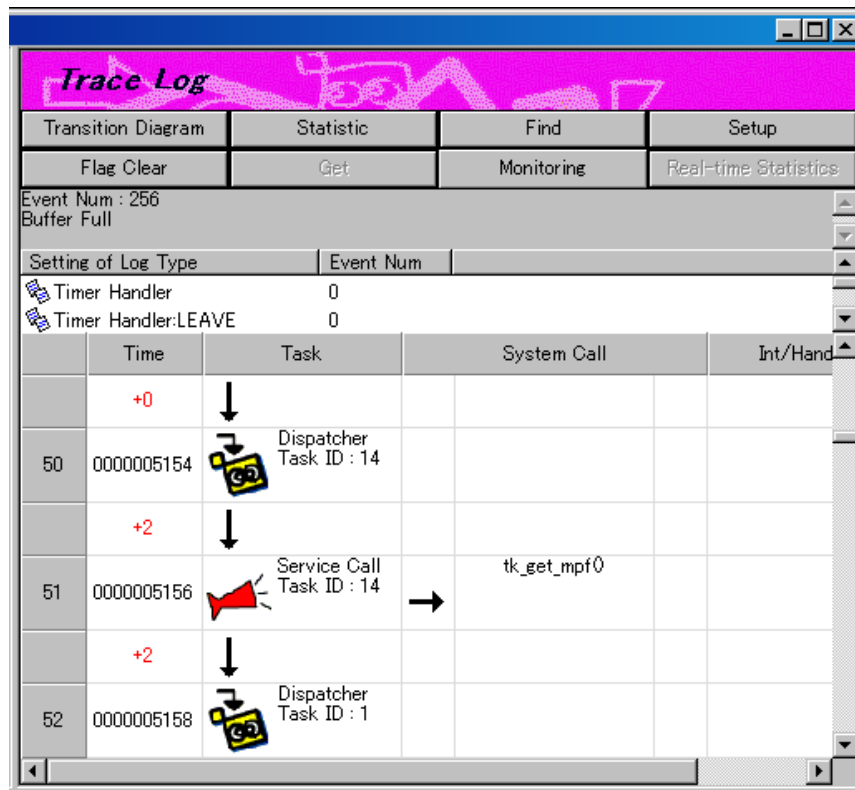
- List format
- State transition diagram
- Statistical format

There is also a "monitoring" function that displays the "state transition diagram" in real time.

● List format

Figure 3.14-2 shows a screen example of the list format log display.

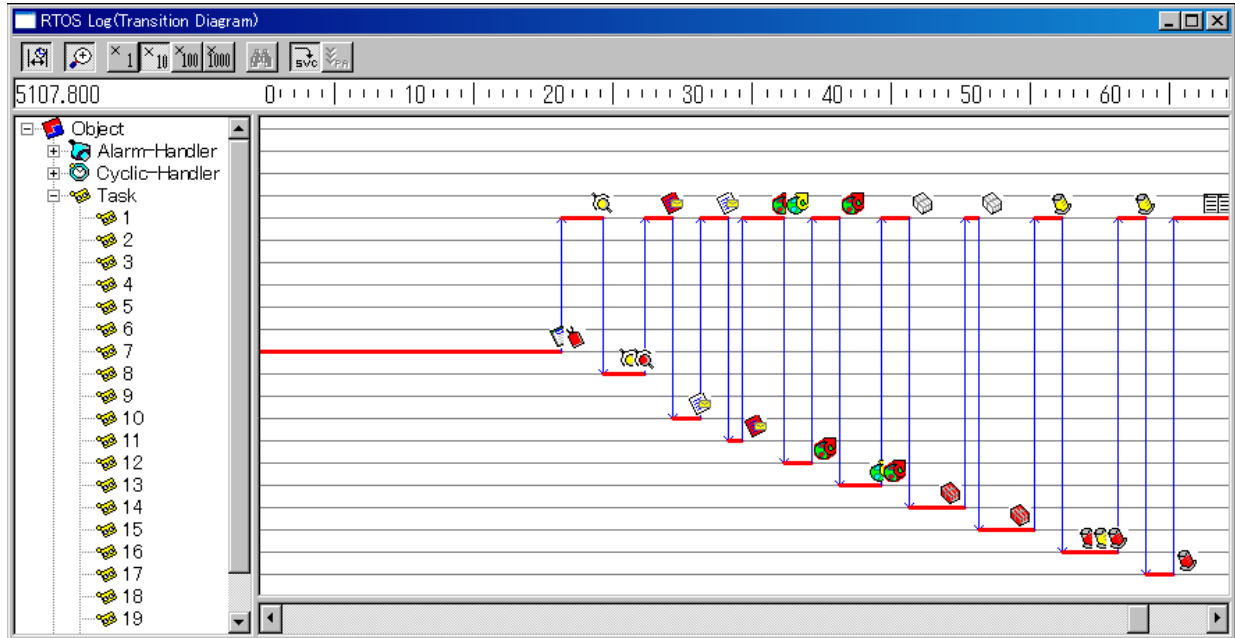
Figure 3.14-2 Screen Example of List Format Log Display



● State transition diagram

Figure 3.14-3 shows a screen example of the state transition diagram format log display. The state transition diagram allows the state of the task dispatcher to be understood at a glance.

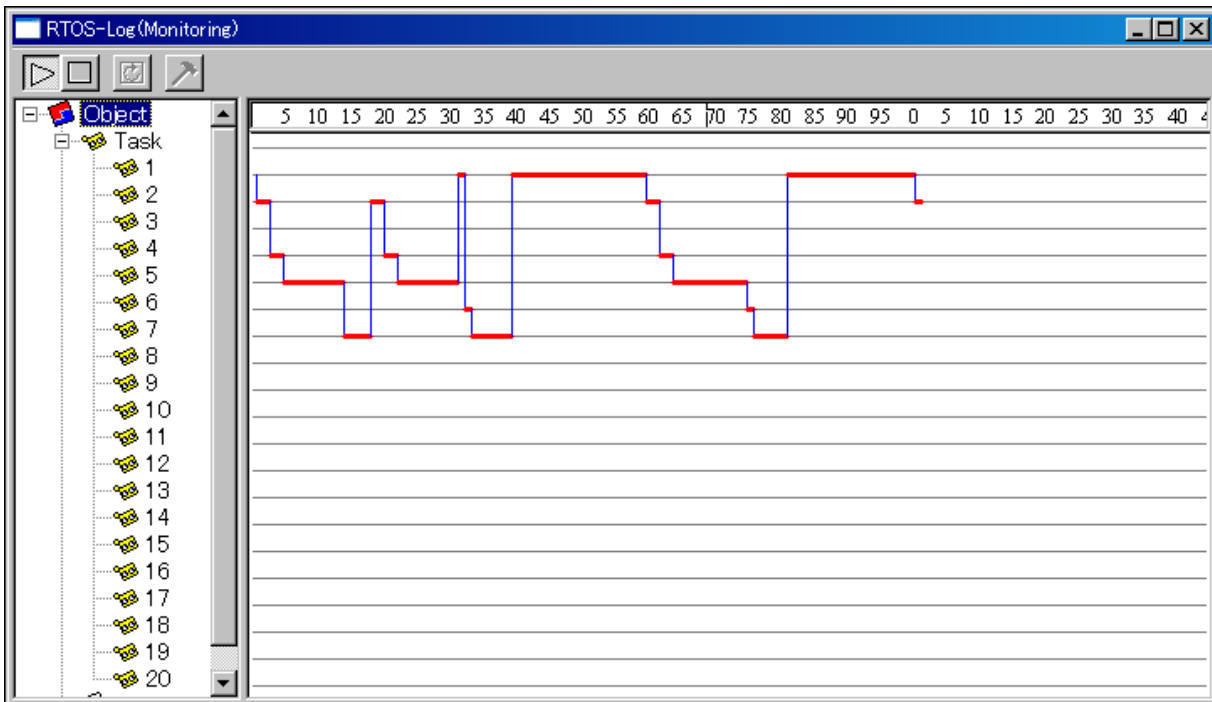
Figure 3.14-3 Screen Example of State Transition Diagram



● Monitoring

In Figure 3.14-3, the user program is stopped, and a state transition diagram is displayed based on the log information up to immediately prior to the stop. As an alternative, it is possible to have the state transition diagram displayed as the user program is running. This function is called "monitoring". Figure 3.14-4 shows a screen example of the monitoring.

Figure 3.14-4 Screen Example of Monitoring

**Note:**

The monitoring function cannot be used if the CPU does not have a built-in DSU4 debug unit. See the "Analyzer Guide" for details.

● Statistical Format

Figure 3.14-5 shows a screen example of the statistical format log display. In the statistical format, information such as the proportion of execution time and number of dispatches for each task can be obtained.

Figure 3.14-5 Screen Example of Statistical Format

ID	Percentage (%)	Total Run Time	Run Count	Max Run Time	Min Run Time
Idle	0 %	0	0	0	0
Task-1 [ipri=138]	1 %	28	15	3	1
Task-2 [ipri=2]	0 %	0	0	0	0
Task-3 [ipri=3]	0 %	3	1	3	3
Task-4 [ipri=4]	8 %	225	1	225	225
Task-5 [ipri=5]	52 %	1337	1	1337	1337
Task-6 [ipri=6]	4 %	113	1	113	113
Task-7 [ipri=7]	26 %	670	1	670	670
Task-8 [ipri=8]	0 %	3	1	3	3
Task-9 [ipri=9]	0 %	0	0	0	0
Task-10 [ipri=10]	0 %	2	1	2	2
Task-11 [ipri=11]	0 %	1	1	1	1
Task-12 [ipri=12]	0 %	2	1	2	2
Task-13 [ipri=13]	0 %	3	1	3	3
Task-14 [ipri=14]	0 %	4	1	4	4
Task-15 [ipri=15]	0 %	4	1	4	4

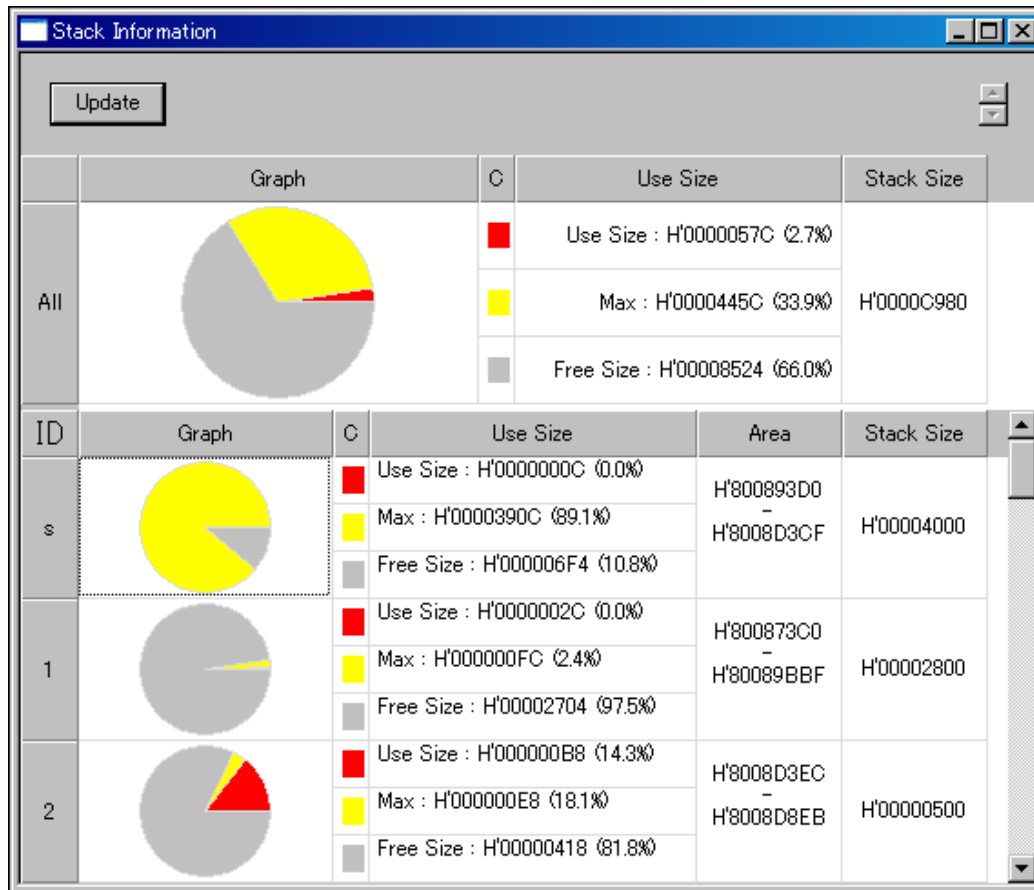
■ Issuing System Calls

While the user program is stopped, a selected system call function can be executed.

■ Stack Information

Display the current task stack usage and maximum usage using a graph. Figure 3.14-6 shows a screen example of the stack information.

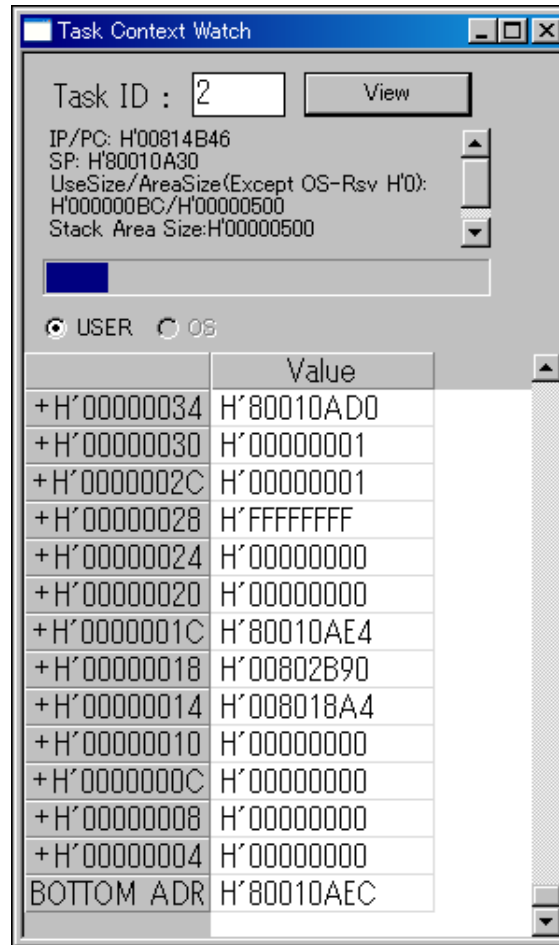
Figure 3.14-6 Screen Example of Stack Information



■ Task Context Display

Displays the contents of the context of a specified task. Figure 3.14-7 shows a screen example of the task context display.

Figure 3.14-7 Screen Example of Task Context Display



CHAPTER 4

WRITING A USER PROGRAM

This chapter describes the basic items in writing a user program on μ T-REALOS.

- 4.1 Configuring a User Program
- 4.2 Start Flow
- 4.3 Reset Entry Routine
- 4.4 Initial Routine
- 4.5 Task
- 4.6 Period Handler
- 4.7 Alarm Handler
- 4.8 Interrupt Handler
- 4.9 Error Routine
- 4.10 Power Saving Routine
- 4.11 Extension SVC Handler
- 4.12 Device Driver
- 4.13 Notes when Writing a User Program

4.1 Configuring a User Program

This section describes how to configure a user program.

■ Configuring a User Program

A user program consists of the modules in Table 4.1-1. Build the user system after creating modules necessary for the user system.

Table 4.1-1 User Program Configuration Elements

Module name	Processing overview	Necessity
Reset entry routine	This routine is first launched by the hardware reset. It performs hardware initialization, and starts the kernel.	Mandatory
Initial routine	This routine is called from the kernel initialization. It provides the operating environment for the user program.	Mandatory
Task	Performs the main process of a user program.	Mandatory
Period handler	Created when performing a process at regular time intervals.	Optional
Alarm handler	Created if there is a process to be performed after a certain time.	Optional
Interrupt handler	Created when handling hardware interrupts. When system time is used, it is necessary to create a timer interrupt handler.	Optional
Error routine	Created when handling kernel errors from a user program.	Optional
Power saving routine	Created when performing power saving process using a user program.	Optional
Extension SVC Handler	Created when defining user function using the extension SVC handler.	Optional
Device driver	Created when controlling the device driver using the device management API.	Optional

For more on these, see "4.3 Reset Entry Routine" to "4.12 Device Driver" in this document respectively.

4.2 Start Flow

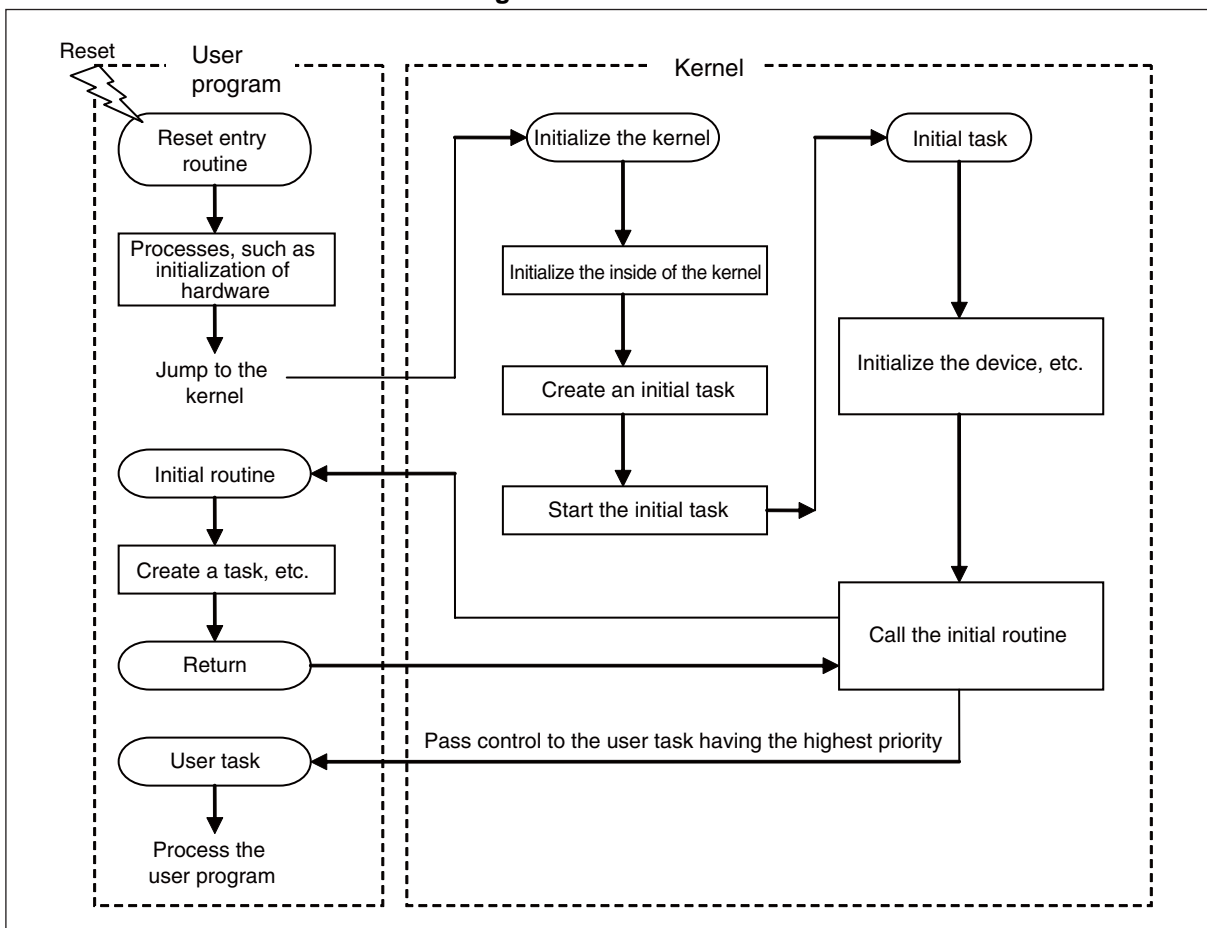
This section explains the process flow from the hardware reset occurrence until control is passed to the user program task.

■ Starting a User Program

Figure 4.2-1 shows the processing flow after hardware reset occurs.

When the initial routine of a user program is called by the initial task, control is passed to the user program having the highest priority task.

Figure 4.2-1 Start Flow



4.3 Reset Entry Routine

This section describes how to write the reset entry routine.

■ Reset Entry Routine

The reset entry routine, which is launched by reset, performs initialization of the processor and of the peripheral devices for which initial settings are necessary during reset. Control is then moved to μ T-REALOS.

■ Process of Reset Entry Routine

The reset entry routine generally performs the following processes:

- Setting of Stack pointer (SP) (mandatory)
Sets SP with the address of stack pointer used when the reset entry routine is running.
- Hardware initial settings (optional)
Initially sets those items of hardware for which settings are necessary before starting the kernel such as memory controller, CPU clock, interrupt controller, CPU cache
- Copy to RAM of INIT section inside ROM (optional)
When burning a user program into the ROM, the INIT section (the area inside the object program where the global variables with initial values are contained) is located inside the ROM. This is copied to the RAM area, where read/write is enabled.
- Initializing the DATA section of a user program (Mandatory)
Clears the DATA section (the area inside the object program where global variables without initial values are contained) inside a user system which has been loaded into the memory of the target hardware.
- Starting the kernel (mandatory)
Starting the kernel at the end of the reset entry routine. The kernel startup jumps to the label of "`__kernel_start`" by the JMP command of the assembler language after the system stack is set.

■ A specific Example of the Reset Entry Routine

Figure 4.3-1 to Figure 4.3-3 shows the description examples for each process of the reset entry routine. The source codes are attached to the product as a sample program (icrt0.asm) of μ T-REALOS.

Figure 4.3-1 Description Example of INIT Section Copy

- As constants RAM_INIT, ROM_INIT, and INIT are defined at the linker, it is not necessary to define them at the user program. RAM_INIT refers to the start address of the INIT section in the RAM, ROM_INIT refers to the start address of the INIT section in the ROM, and INIT refers to the size (number of bytes) of the INIT section.
- In the following example, copy_rom1 performs copy byte by byte, and copy_rom2 performs copy in units of 4 bytes.

```

/*
 *      Initialization of 'data' area (ROM startup)
 */
        ldi        #_RAM_INIT, r0
        ldi        #_ROM_INIT, r1
        ldi        #size of(INIT), r2
        cmp        #0, r2
        beq:d      copy_rom_end
        ldi        #3, r12
        and        r2, r12
        beq:d      copy_rom2
        mov        r2, r13
        mov        r2, r3
        sub        r12, r3
copy_rom1:
        add        #-1, r13
        ldub       @(r13, r1), r12
        cmp        r3, r13
        bhi:d      copy_rom1
        stb        r12, @(r13, r0)
        cmp        #0, r3
        beq:d      copy_rom_end
copy_rom2:
        add        #-4, r13
        ld         @(r13, r1), r12
        bgt:d      copy_rom2
        st         r12, @(r13, r0)
copy_rom_end:

```

Figure 4.3-2 Description example of DATA section 0 Clear

- As constants DATA and sizeof DATA are defined at the linker, it is not necessary to define them at the user program. DATA is the start address of the DATA section area. sizeof DATA is the size (number of bytes) of the DATA section.
- In the following example, clear_ram0 performs 0 clear in units of 4 bytes, and clear_ram2 performs 0 clear byte by byte in the final area that is less than 4 bytes.

```

/*
 *      Clear 'bss' area
 */
        ldi:8      #0, r0
        ldi        #sizeof DATA & ~0x3, r1
        ldi        #DATA, r13
        cmp        #0, r1
        beq        clear_ram1
clear_ram0:
        add2       #-4, r1
        bne:d      clear_ram0
        st         r0, @(r13, r1)
clear_ram1:
        ldi:8      #sizeof DATA & 0x3, r1
        ldi        #DATA + (sizeof DATA & ~0x3), r13
        cmp        #0, r1
        beq        clear_ramE
clear_ram2:
        add2       #-1, r1
        bne:d      clear_ram2
        stb        r0, @(r13, r1)
clear_ramE:

```

Figure 4.3-3 Description Example of Kernel Startup

- Set the kernel stack address to the SP register
- Jump to the label of "__kernel_start" to start the kernel

```

        ldi:32     #__kernel_stack_end, sp // Set SP(SSP)
        ldi:32     #__kernel_start, r0    // System startup
        jmp        @r0

```

4.4 Initial Routine

This section describes how to write the initial routine.

■ Process of the Initial Routine

The initial routine is called from the initial task created during initialization of the kernel. Although the initial routine can be described freely in accordance with the user program, it generally performs the following processes:

- Creating and starting objects necessary for operations of the user program, such as tasks, semaphores, time event handlers
 - Initializing and registering device drivers
 - Initializing hardware
 - Starting a timer interrupt
-

Note:

The initial routine is executed while the interrupt is enabled.

■ Description Format of the Initial Routine

The initial routine is described as follows:

Figure 4.4-1 Description Format of the Initial Routine

```
void sample_init(void)
{
    /*
        Process the initial routine
    */
    return;
}
```

■ Description Example of the Initial Routine

Figure 4.4-2 shows the description example of the initial routine. The source codes are attached to the product as a sample program (init_task.c) of μ T-REALOS.

Figure 4.4-2 Description Example of the Initial Routine

- After the timer for the system clock is started(`START_TIMER0()`), semaphore and three tasks (task ID=tsk1, tsk2, tsk3) will be created. Then, the created 3 tasks are started. The task whose task ID is task1 and the task whose task ID is tsk2 use the same function (task1).

```
static UB task1_stack[0x400], task2_stack[0x400], task3_stack[0x400];
void unit_task(void)
{
    ID tsk1, tsk2, tsk3;
    T_CTSK ctsk;
    T_CSEM csem;

    START_TIMER0();                /* Start Timer0 for System clock */

    csem.sematr = TA_TFIFO | TA_FIRST;
    csem.isemcnt = 0;
    csem.maxsem = 1;
    sem1 = tk_cre_sem(&csem);      /* Create semaphore */

    ctsk.exinf = (VP)1;
    ctsk.tskatr = TA_HLNG | TA_RNG0 | TA_USERBUF;
    ctsk.task = task1;
    ctsk.itskpri = 1;
    ctsk.stksz = sizeof(task1_stack);
    ctsk.bufptr = task1_stack;
    tsk1 = tk_cre_tsk(&ctsk);      /* Create task1 */

    ctsk.exinf = (VP)2;
    ctsk.tskatr = TA_HLNG | TA_RNG0 | TA_USERBUF;
    ctsk.task = task1;
    ctsk.itskpri = 2;
    ctsk.stksz = sizeof(task2_stack);
    ctsk = sizeof(task1_stack);
    tsk2 = tk_cre_tsk(&ctsk);      /* Create task2 */

    ctsk.exinf = (VP)3;
    ctsk.tskatr = TA_HLNG | TA_RNG0 | TA_USERBUF;
    ctsk.task = task2;
    ctsk.itskpri = 3;
    ctsk.stksz = sizeof(task3_stack);
    ctsk.bufptr = task3_stack;
    tsk3 = tk_cre_tsk(&ctsk);      /* Create task3 */

    tk_sta_tsk(tsk1, 1);           /* Start task1 */
    tk_sta_tsk(tsk2, 2);           /* Start task2 */
    tk_sta_tsk(tsk3, 3);           /* Start task3 */
}
```

4.5 Task

This section describes how to write a task.

■ Description Format of the Task

The task is described as follows:

Figure 4.5-1 Description Format of the Task

```
void task(INT stacd, VP exinf)
{
    /*
     * Process the body of the task program
     */
    tk_ext_tsk(); or tk_exd_tsk(); /* Task termination */
}
```

The task start code (stacd) specified during task startup (when tk_sta_tsk is called) is passed to stacd. The extension information (exinf) specified when the task is created is passed to exinf.

A function(task) cannot be terminated by a simple return. Using tk_ext_tsk or tk_exd_tsk to ensure termination.

■ Creating a Task

tk_cre_tsk is called to create a task. An example is shown below. In this example, function, "task1", is being created using task priority 1. If tk_cre_tsk is terminated normally, the task ID will be returned as the return value.

Figure 4.5-2 Description Example of Task Creation

```
ID tid1;           /* Task ID of task1 */
T_CTSK ctsk;       /* Input parameter of tk_cre_tsk */
INT task1_stack[256]; /* Stack area of the task */

ctsk.exinf = (VP)1; /* Extension information=1 */
ctsk.tskatr = TA_HLNG | TA_RNG0 | TA_USERBUF; /* Attribute */
ctsk.task = task1; /* Start address of the task */
ctsk.itskpri = 1; /* Task priority */
ctsk.stksz = sizeof(task1_stack); /* Stack size */
ctsk.bufptr = task1_stack; /* Start address of the stack */
tid1 = tk_cre_tsk(&ctsk); /* Create the task */
```

For details of tk_cre_tsk, see "3.3.1 tk_cre_tsk" of "API Reference" .

■ Starting a Task

A task created by `tk_cre_tsk` is initially in the stop status. Therefore, `tk_sta_tsk` is called to run this task. In the example below, the task whose task ID is `tid1` is being started.

Figure 4.5-3 Description Example of the Task Startup

```
tk_sta_tsk(tid1, 1);    /* Start the task whose task ID is tid1 */
```

The status of a task started by `tk_sta_tsk` is executable. If the priority of a task is higher than those of other tasks of execution status or executable status, the task attains execution status.

For details of `tk_sta_tsk`, see "3.3.3 `tk_sta_tsk`" of "API Reference".

■ A specific Example of the Task

Figure 4.5-4 shows the description example of a task, and Figure 4.5-5 shows the operation diagram of the program in the description example. The source codes are attached to the product as a sample program (`init_task.c`) of μ T-REALOS. In addition, the task in this specific example is created/started using the initial routine given in Figure 4.4-2.

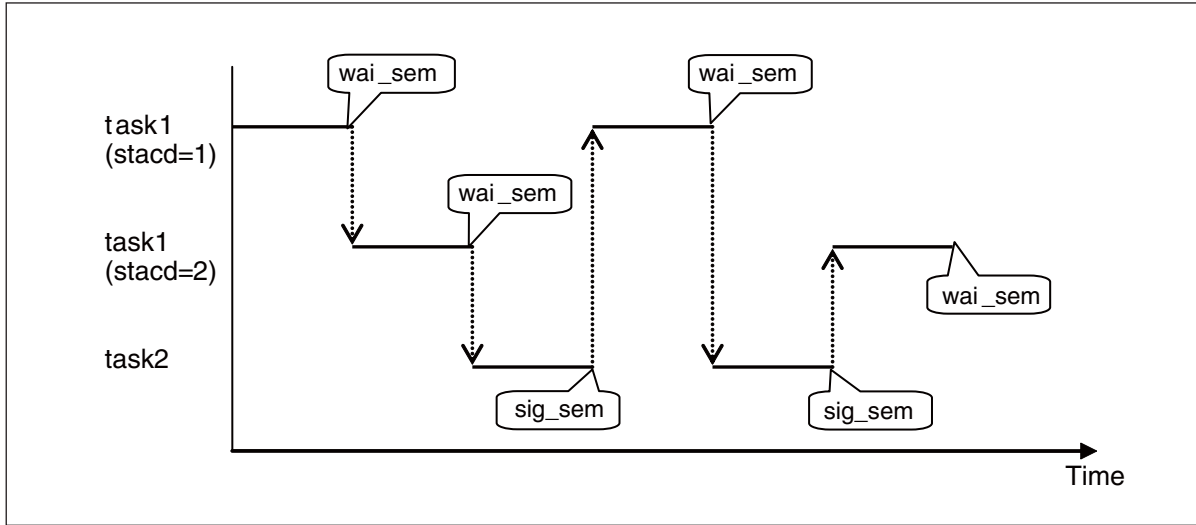
Figure 4.5-4 Description Example of a Task

- Function task1 runs at two task: task started with stacd=1 and task started with stacd=2. Each task moves to the status of waiting for the semaphore of sem1 by the system call of tk_wai_sem.
- task2 releases the semaphore resources of sem1 by the system call of tk_sig_sem. This releases task1 from the standby status.
- The task priorities are: task1(stacd=1) > task1(stacd=2) > task2

```
static void task1( INT stacd, VP exinf )
{
    if(stacd == 1){
        while (1) {
            tk_wai_sem(sem1, 1, TMO_FEVR);
        }
    }
    else if (stacd == 2){
        while (1) {
            tk_wai_sem(sem1, 1, TMO_FEVR);
        }
    }
    else{
        tk_ext_tsk();    /* Exit task */
    }
}

static void task2( INT stacd, VP exinf )
{
    if (stacd == 3){
        while (1) {
            tk_sig_sem(sem1, 1);
        }
    }
    else{
        tk_ext_tsk();    /* Exit task */
    }
}
```

Figure 4.5-5 Operation Diagram of the Description Example



4.6 Period Handler

This section describes how to write a period handler.

■ Description Format of a Period Handler

The period handler is described as follows:

Figure 4.6-1 Description Example of a Period Handler

```
void cychdr1(VP exinf)
{
    /* Process the period handler */

    return; /* Terminate the period handler */
}
```

■ Creation of a Period Handler

tk_cre_cyc is called to create a period handler. An example is shown below. In this example, function "cyhdr1" is being created as a period handler of the start period 1 second (1000ms). When creation of a period handler normally ends, the period handler ID is returned as the return value.

Figure 4.6-2 Description Example of the Period Handler Creation

```
ID cycid1; /* Cyclic handler ID */
T_CCYC ccyc; /* Input parameter of tk_cre_cyc*/
ccycexinf = (VP)1; /* Extension information=1 */
ccyc.cycatr = TA_HLNG | TA_RNG0 | TA_STA; /* Attribute */
ccyc.cyhdr = cychdr1; /* Start address of the period handler */
ccyc.cyctim = 1000; /* Start Cyclic */
ccyc.cycphs = 0; /* Start phase */
cycid1 = tk_cre_cyc(&ccyc); /* Create the Cyclic handler */
```

■ Launch of a Period Handler

To move a period handler from the stop status to the action status, tk_sta_cyc is called. In the following example, the period handler ID is started as the period handler of cycid1.

Figure 4.6-3 Description Example of the Period Handler Startup

```
tk_sta_cyc(cycid1); /* Start the Cyclic handler whose ID is cycid1 */
```

4.7 Alarm Handler

This section describes how to write an alarm handler.

■ Description Format of an Alarm Handler

An alarm handler is described as follows:

Figure 4.7-1 Description Example of an Alarm Handler

```
void almhdr1(VP exinf)
{
    /* Process the alarm handler */
    return; /* Terminate the alarm handler */
}
```

The extension information specified when an alarm handler is created (tk_cre_alm) is passed to exinf.

■ Creating an Alarm Handler

tk_cre_alm is called to create an alarm handler. An example is shown below. In this example, function "almhdr1" is created as an alarm handler. If creation of an alarm handler is terminated normally, the alarm handler ID will be returned as the return value.

An alarm handler moves to the stop status after it is created.

Figure 4.7-2 Description Example of the Alarm Handler Creation

```
ID almid1;          /* Alarm handler ID*/
T_CALM calm;        /* Input parameter of tk_cre_alm*/

calm.exinf = (VP)1; /*Extension information =1 */
calm.almatr = TA_HLNG | TA_RNG0; /* Attribute */
calm.almhdr = almhdr1; /* Start address of the alarm handler */
almid1 = tk_cre_alm(&calm); /* Create the alarm handler */
```

■ Starting an Alarm Handler

To move an alarm handler from the stop status to the action status, call tk_sta_alm. In the example below, the alarm handler with an alarm handler ID of almid1 is started using time-out time 100ms.

Figure 4.7-3 Description Example of the Alarm Handler Startup

```
tk_sta_alm(almid1, 100); /* Start the alarm handler whose ID is
                           almid1 using a time-out time of 100ms*/
```

4.8 Interrupt Handler

This section describes how to write an interrupt handler.

■ Description Format of an Interrupt Handler

An interrupt handler is described as follows:

Figure 4.8-1 Description Example of an Interrupt Handler

```
void sample_inthdr( void)
{
    /* Interrupt handler body */
}
```

The interrupt handler is executed at the task independent portion. In addition, it is started while interrupt is enabled. Therefore, while an interrupt handler is being executed, the interrupt handler may be started multiply. For details, see "CHAPTER 4 RESET AND EIT PROCESSING" of "FR Family Instruction Manual".

■ Registering an Interrupt Handler

There are two methods of registering an interrupt handler: static, and dynamic. For static registering method, see "5.3 Setting of Configuration". For dynamic registering method, `tk_def_int` is called from a user program.

- Example

When the timer interrupt handler of vector number 24 registers "timer" as an interrupt handler through a user program.

Figure 4.8-2 Example of Registering an Interrupt Handler Through a User Program

```
T_DINT dint;
ER err;

dint.intatr = TA_HLNG|TA_RNG0;      /* Attribute */
dint.inthdr = timer;               /* Start address of the interrupt handler */
err = tk_def_int(&dint);           /* Register the interrupt handler */
```

To register an interrupt handler through a user program, set "`_KERNEL_USE_TKDEFINT`" of the macro specified by the configurator to 1, and execute the configuration. In addition, to perform operations with the vector table placed in the ROM, register an interrupt handler using static API.

■ Timer Interrupt Handler

To use the functions of time event handler, timeout, and system time, it is necessary to let the timer interrupt occur at intervals of 1ms, and then update the system time using the interrupt handler. System time will be updated when `isig_tim` is called.

An example of the timer interrupt is shown below.

Figure 4.8-3 Description Example of an Timer Interrupt Handler

```
void timer(void)
{
    /*
     Clear the timer interrupt factors
    */
    isig_tim();
}
```

Note:

When an interrupt handler is described using assembler, note the following points:

- Calling an interrupt handler or returning from an interrupt handler is not via the OS.
 - As the OS does not back up and restore registers or perform stack settings, perform such processes on the interrupt handler side.
-

4.9 Error Routine

This section describes how to write an error routine.

■ Description Format of an Error Routine

The description format of an error routine is shown as follows:

Figure 4.9-1 Description Format of an Error Routine

```
void sample_errrtn (UINT errrtn, INT errinf1,INT errinf2)
{
    /* Error routine body */
}
```

The following information are passed to errrtn, errinf1, and errinf2.

- errrtn : Error factor
 - = _KERNEL_ERR_SYS_DOWN (0x01): System down
 - = _KERNEL_ERR_INI_ERR (0x02): Initial setting error
 - = _KERNEL_ERR_EIT_DOWN (0x04): Undefined interrupt
- errinf1 : Error information1
 - In the case of [_KERNEL_ERR_SYS_DOWN]
 - = 0x1 : tk_ext_tsk was called from the task independent portion.
 - = 0x2 : tk_exd_tsk was called from the task independent portion.
 - = 0x3 : tk_ext_tsk had been called while dispatch was disabled.
 - = 0x4 : tk_exd_tsk had been called while dispatch was disabled.
 - In the case of [_KERNEL_ERR_INI_ERR]
 - Initial setting error information
 - = 0x1 : Heap area assignment error
 - = 0x2 : System startup error
 - = 0x3 : Initial task startup error
 - = 0x4 : Module initialization error
 - = 0x5 : Power off processing error
 - [In the case of [_KERNEL_ERR_EIT_DOWN]
 - Uncertain value
- errinf2 : Error information2
 - Not used. Reserved for future extension.

■ Registering an Error Routine

For registering an error routine, see "5.3 Setting of Configuration".

4.10 Power Saving Routine

This section describes how to write a power saving routine.

■ Description Format of a Power Saving Routine

The power saving routine is a process called when the status has become idle inside the kernel. The processing of transition to the power saving mode is described inside a function.

Figure 4.10-1 Description Example of a Power Saving Routine

```
void usr_low_pow ( void )
{
    /* Describe the process of transition to the power saving mode*/
}
```

■ Registering a Power Saving Routine

For registering a power saving routine, see "5.3 Setting of Configuration".

Note:

When a system call is made inside the power saving routine, the operation is not guaranteed.

4.11 Extension SVC Handler

This section describes how to create and call an extension SVC handler.

■ Description Format of an Extension SVC Handler

The description format of an extension SVC handler is shown as follows:

Figure 4.11-1 Description Format of an Extension SVC Handler

```
INT svchdr(VP pk_para, FN fncd)
{
    /*
     Branch and proceed according to fncd
    */
    return retcode; /*Terminate the extension SVC handler */
}
```

pk_para turns the parameters passed from the caller into the packet format. The packet format can be determined by subsystems arbitrarily.

fncd, which is the function code, contains the subsystem ID in its low 8 bits. The remaining high bits are determined by subsystems arbitrarily.

■ Calling Format of an Expansion SVC Handler

An extension SVC handler is called from a user program using software interrupt of interrupt number 64 with the fncd value set to r0 register. Therefore, it is necessary to describe an extension SVC handler calling section as a user program in the assembler.

Figure 4.11-2 Calling Format of an Expansion SVC Handler

```
#define FUNC1_FNCD 0x10A /* ssid = 10 */
INT func1(int arg1, int arg2)
{
    __asm( "        ldi:32    #FUNC1_FNCD, r0" );
    __asm( "        int      #64" );
}
```

In the example of Figure 4.11-2, the SVC handler whose subsystem ID(ssaid) is 10 is called. In the called SVC handler, the beginning addresses of packets contained in arg1 and arg2 are passed to pk_para, and "0x10A" is passed to fncd.

4.12 Device Driver

This section describes how to write a device driver.

■ Device Driver Interface

In the μ T-Kernel specification, the device management function increases the portability of the device drivers by unifying their interfaces. The following describes how to create a driver based on the device driver interface. In addition, for details of a device driver, see "Appendix C Device Driver Interface" of "API Reference".

■ Determining A Device Name

Device name is the name granted to the type unit of a device using up to 8 bytes.

A device name using the following format:.

Type	Unit	Sub unit
------	------	----------

A device name consists of the following elements.

Type : A name indicating the type of a device. Characters that can be used are a-z, A-Z.

Unit : A number indicating the physical device. Characters that can be used are a-z. Specified using a single character. Assigned for each unit starting with a.

Sub unit : A number indicating the logical device. Characters that can be used are numbers between 0-254, not exceeding 3 digits. Assigned for each sub unit starting with 0.

■ Creating an Open Function (openfn)

An open function is called from the kernel when tk_opn_dev is called from a user program. An open processing function makes preparation to access the device data.

For details of an open processing function, see "Device processing function open processing function (Openfn)" of "Appendix C Device Driver Interface" in "API Reference".

```
ER ercd = openfn(ID devid, UINT omode, VP exinf)
{
    /*
     * Device open processing
     */
}
```


■ Creating a Close Function (closefn)

A close function is called from the kernel when tk_cls_dev is called from a user program.

Calling a close processing function means access to a device has been terminated. The driver then performs the device terminating process whenever necessary.

```
ERercd = closefn(ID devid, UINT option, VP exinf)
{
    /*
     Device close processing
    */
}
```

■ Creating a Process Start Function (execfn)

A process start function is called from the kernel when tk_rea_dev, tk_srea_dev, tk_wri_dev, or tk_swri_dev is called from a user program.

In the process start function, the data to be processed is first set to the parameter and then called. However, the function does not return upon completion of the data process, instead it returns when the process has been accepted. For example, when the data to be written to a device is passed via tk_wri_dev, it returns on completion of the write start instruction, and it is not necessary to wait for completion of the device write process.

For details of a process start function, see "Device processing function process start function (execfn)" of "Appendix C Device Driver Interface" in "API Reference".

```
ERercd = execfn(T_DEVREQ *devreq, TMO tmout, VP exinf)
{
    /*
     Device process start
    */
}
```

■ Creating a Waiting for Completion Function (waitfn)

A waiting for completion function is called from the kernel when tk_wai_dev, tk_srea_dev, or tk_swri_dev is called from a user program.

A waiting for completion function waits for the completion of I/O request accepted at the process start function. Therefore, system calls for entering the standby status (such as tk_slp_tsk) may be used.

For details of a waiting for completion function, see "Device processing function waiting for completion function (waitfn)" of "Appendix C Device Driver Interface" in "API Reference".

```
INT pktno = waitfn(T_DEVREQ *devreq, INT nreq, TMO tmout, VP exinf)
{
    /*
     Device wait for completion process
    */
}
```

■ Creating a Process Abort Function (abortfn)

The process abort function is called from the kernel if there are unfinished I/O requests for the device when the device close instruction is issued from a user program.

Process aborting requested by I/O specified using parameters is performed in a process aborting function. I/O for the device is aborted, and removes the waiting state if the task has entered a state waiting for I/O completion.

For details of a process abort function, see "Device processing function process abort function (abortfn)" of "Appendix C Device Driver Interface" in "API Reference".

```
INT pktno = waitfn(T_DEVREQ *devreq, INT nreq, TMO tmout, VP exinf)
{
    /*
     * Device process abort
     */
}
```

■ Event Function (eventfn)

An event function is called from the kernel when tk_sus_dev or tk_evt_dev is called from a user program. This function is called from a user program or the kernel when notifying some event to a device.

As the event type is passed to a parameter in an event function, process for that event is performed in the driver.

```
INT rtncd = eventfn(INT evtyp, INT evtinf, VP exinf)
{
    /*
     * Device event process
     */
}
```

4.13 Notes when Writing a User Program

This section describes notes when writing a user program of μ T-REALOS.

■ Notes on the Overall of a Program

- Internal identifiers starting with "_KERNEL", "_kernel", "tk_", "tm_" and "knl_"
The kernel of μ T-REALOS uses symbols and macros starting with the above mentioned. Do not use these symbols and macros in a user program. This may cause duplicate definition.
- Management register of μ T-REALOS
 μ T-REALOS uses the ILM field of the PS register. Do not change this field in a user program.
- Include file of kernel
Include "[SOFTUNE Install Directory]\utkernel\911\include\tk\tkernel.h in a user program using a system call

■ Notes on the Overall of a System Call

- A system call can be made while the task independent portion or dispatch is disable.
When calling is disabled, an E_CTX error or exception may occur. For availability of calling, see "3.1 System Call List" of "API Reference" in addition, operations cannot be guaranteed when isig_tim or tk_ret_int is called from the task section.
- Omitting the error check of a system call
Check for entry address and packet address will not be performed. Specifying an illegal address may cause an abnormal operation.

■ Notes on a Task

- Stack Definition
Secure the stack area to make its beginning address to become 4 byte boundaries.
- Status transition of tasks during execution while the dispatch is on hold
While dispatch is on hold, the state transition is delayed until dispatch occurs, when moving the tasks during execution forcibly to the forcible waiting state and the stop state by calling tk_ter_tsk and tk_sus_tsk from task independent portion. In such a case, tasks being executed will retain the execution status. However, when reference the task portion using tk_ref_tsk, it becomes the forcible waiting state or the stop state.

■ Notes on Interrupt

- Execution priority of an interrupt handler and a time event handler

The execution priority for each handler is determined according to the defined interrupt level. For details, see "CHAPTER 4 RESET AND EIT PROCESSING" of "FR Family Instruction Manual".

A time event handler is executed at the interrupt handler interrupt level for system clock calling `isig_tim`.

When designing the system stack size, taking these handler multiple startup into consideration, add 80 bytes for interrupt of 1 level.

CHAPTER 5

HOW TO CONSTRUCT A SYSTEM

This chapter describes how to construct a user system.

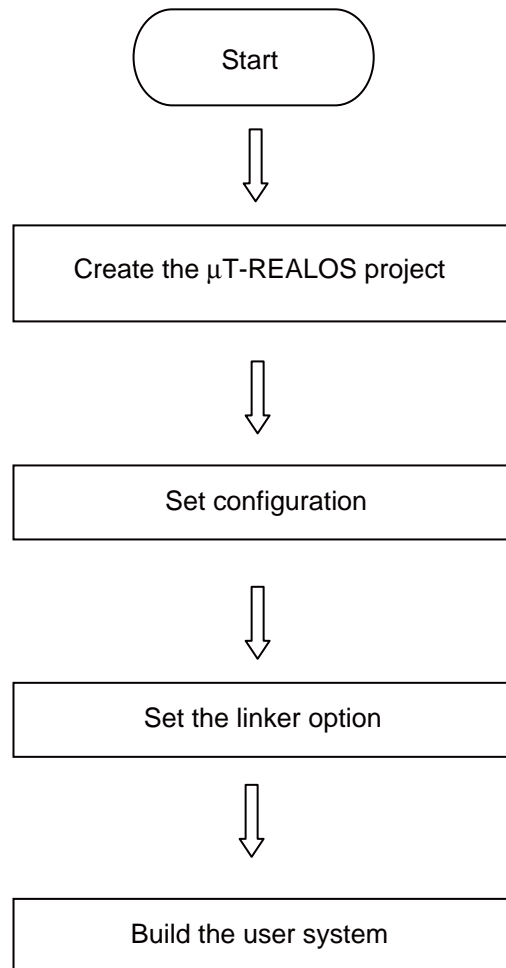
- 5.1 Steps of Constructing a System
- 5.2 Create the μ T-REALOS Project
- 5.3 Setting of Configuration
- 5.4 Setting of Linker Option
- 5.5 Build a User System

5.1 Steps of Constructing a System

This section describes steps of constructing a system including compiling, configuring, and linking a user program for μ T-REALOS.

■ Steps of System Construction

Construct the user system of μ T-REALOS using the following steps.



5.2 Create the μ T-REALOS Project

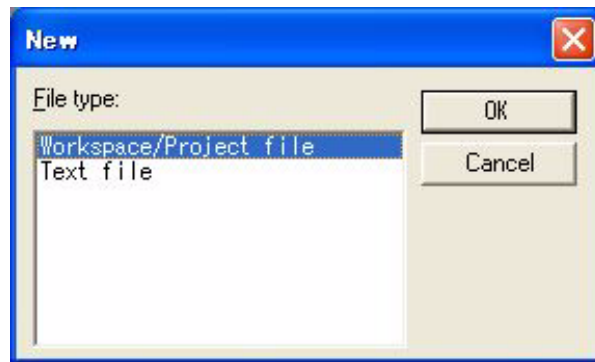
This section describes how to create the μ T-REALOS Project on SOFTUNE Workbench.

■ Create the μ T-REALOS Project

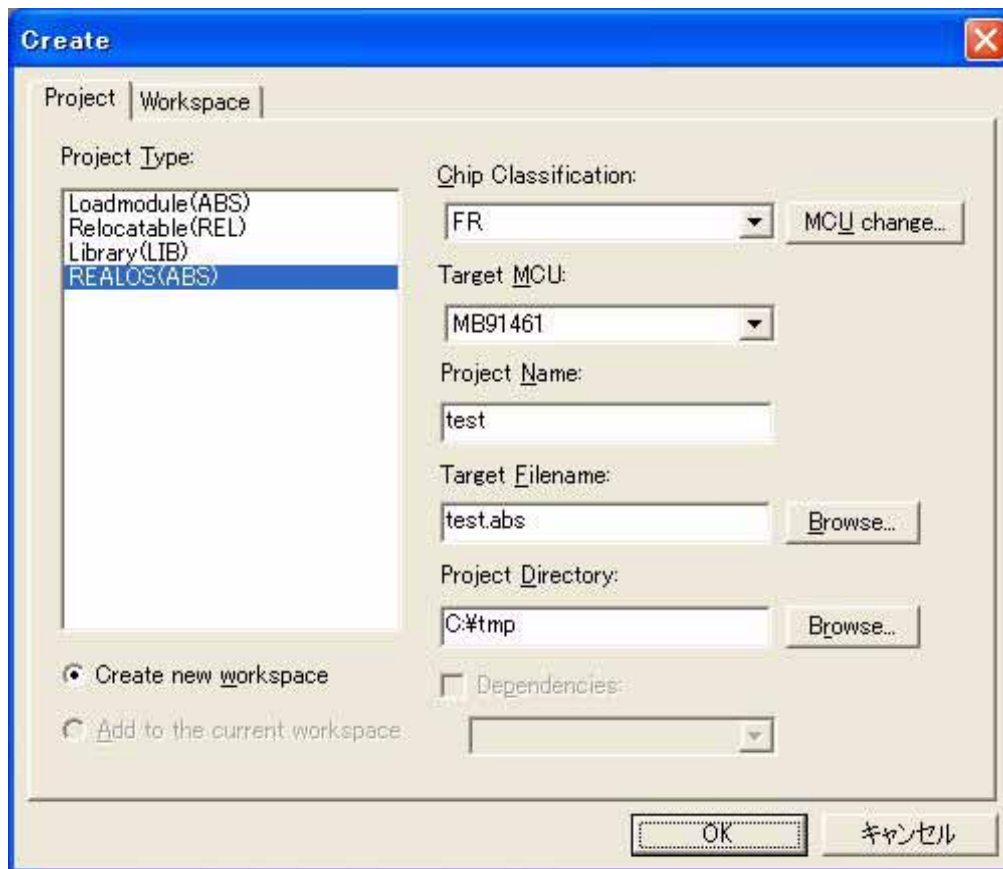
Create the μ T-REALOS using the following steps.

- Select [File]-[New Creation] menu on the SOFTUNE Workbench. Select "Workspace/Project File" as the file type in the opened new dialog, and then click the "OK" button.

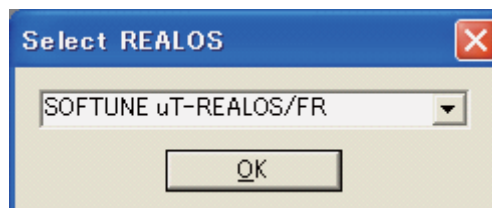
Figure 5.2-1 Selection of File Type



- Select a project tab in the "New Creation" dialog, and select "REALOS(ABS)" in the project type. Input "Chip Type", "Target MCU", and "Project Name", and then click the "OK" button.

Figure 5.2-2 Input the Project Information

- Dialog of "Select REALOS" is displayed. Select "SOFTUNE μ T-REALOS/FR", and then click the "OK" button.

Figure 5.2-3 Select the REALOS Type

- Select some of following items from the dialog for creating a configuration file.
 - New
 - Load an existing configuration file

Figure 5.2-4 Create a Configuration File

5.3 Setting of Configuration

This section describes setting of configuration.

■ Setting of Configuration

Define the following items at configuration.

- Definition of the maximum number of objects

Set the maximum value for the number of objects that the user program can use. The objects can be created up to the maximum value and used in the user program. Therefore, define the maximum value using a value bigger than the number of objects that are used in the user program.

For example, when 3 semaphores are used in the user program, define the configuration specification macro "_KERNEL_MAX_SEM" as 3. In such a case, the program can operate without problems even it is defined as 10. However, since 10 semaphore control areas are ensured in kernel, the 7 unused control areas will become useless. Therefore, it is necessary to define the maximum number of objects used in application in order to optimize usage efficiency of memory.

It is unnecessary to define objects not used in the user program.

Table 5.3-1 displays byte number of consumed memory in kernel for each object.

Table 5.3-1 Consumed Memory of Object Management Block

Object name	Configuration specification macro	Management area size (byte)
Task	_KERNEL_MAX_TSK	119
Semaphore	_KERNEL_MAX_SEM	28
Event Flag	_KERNEL_MAX_FLG	24
Mailbox	_KERNEL_MAX_MBX	28
Mutex	_KERNEL_MAX_MTX	32
Message Buffer	_KERNEL_MAX_MBF	52
Rendezvous Port	_KERNEL_MAX_POR	36
Fixed-size Memory Pool	_KERNEL_MAX_MPF	56
Variable-size Memory Pool	_KERNEL_MAX_MPL	56
Cyclic Handler	_KERNEL_MAX_CYC	44
Alarm Handler	_KERNEL_MAX_ALM	40
Device	_KERNEL_MAX_REGDEV	1328

For definition of maximum number of object, see "■ Setting Operation of Configuration" in this section.

- Definition of priority maximum value

Defines the maximum priority value of a task and subsystem. Same as the maximum object value, when the value defined for the task priority becomes smaller, consumed memory of kernel can be reduced. When Task Priority is P, its consumed memory can be calculated according to following formula.

Consumed Memory (byte) = $(8 * P) + 4 * (P / 32)$

- Definition of system stack size and stack size of the initial task

Specifies the size of system stack and stack size of initial task. For specification method of the stack size, see "■ Setting Operation of Configuration" in this section.

- Register the initial routine, error routine and power saving routine

When using the initial routine, error routine, and power saving routine, perform the registration through the static API. For registration method of these routines, see "4.4 Initial Routine", "4.9 Error Routine", and "4.10 Power Saving Routine" of "■ Setting Operation of Configuration" in this section.

- Register an Interrupt Handler

On using an interrupt handler, register it through a static API. After the system startup, dynamic registration of interrupt handler through tk_def_int is also available. In the case of dynamic registration, registration through a static API will be not necessary. In addition, in such a case, define "_KERNEL_USE_TKDEFINT" of configuration macro as "1".

On using an interrupt vector table located in the ROM area, setting "_KERNEL_USE_TKDEFINT" to "0" will allow the kernel to cancel copying the vector table from the ROM to the RAM. Therefore, memory used by the kernel can be reduced.

Whether to register an interrupt handler through a static API or tk_def_int is optional depending on processing of the user program. Registration through a static API has advantages in reducing the codes for registration through tk_def_int.

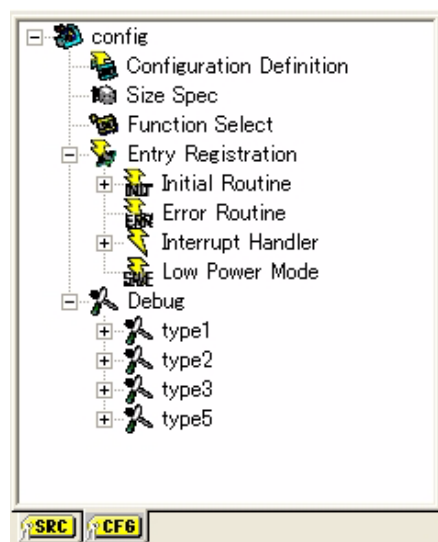
■ Setting Operation of Configuration

Set the configuration parameters using the following steps.

- Project window of the configurator

Clicking the "CFG" tab at lower left of SOFTUNE Workbench window will display Figure 5.3-1.

Figure 5.3-1 Project Window of the Configurator



- Configuration Definition

Double clicking "Configuration Definition" on the project window will display Figure 5.3-2. Set "Definition of Priority" and "Maximum number of each object" in Table 3.13-1.

Clicking the "Recalculate" button upon completion of entry in each configuration window will display the consumed RAM memory area corresponding to the number of inputted object and the total RAM memory area.

Figure 5.3-2 Configuration Definition Window

The Configuration Definition window displays a list of system parameters and their RAM area usage. The parameters are organized into two columns, with a 'RAM Area Use' label above each column. The parameters and their values are as follows:

Parameter	Value	RAM Area Use
Max Task Number	32	3808 bytes
Max Task Priority	140	1136 bytes
Max Semaphore Number	16	448 bytes
Max Eventflag Number	16	384 bytes
Max Mailbox Number	8	224 bytes
Max Mutex Number	8	256 bytes
Max Message Buffer Number	8	416 bytes
Max Fixed-Size Memory Pool Number	0	448 bytes
Max Variable-Size Memory Pool Number	8	448 bytes
Max Cyclic Handler Number	8	352 bytes
Max Alarm Handler Number	8	320 bytes
Max Rendezvous Port Number	8	288 bytes
Max Subsystem Number	8	32 bytes
Max Subsystem Priority	16	0 bytes
Max Device Registration Number	8	10624 bytes
Max Device Open Number	16	0 bytes
Max Device Request Number	16	0 bytes
Initial Task Priority	138	0 bytes

At the bottom left, a box labeled 'RAM Area Total Use' shows '19184 bytes'. At the bottom right, there are buttons for 'OK', 'Cancel', 'Re-calculation', and 'List Display'.

- Size Spec
- Double clicking "Size Spec" on the project window will display Figure 5.3-3. Set items of "Size Spec" in Table 3.13-1.

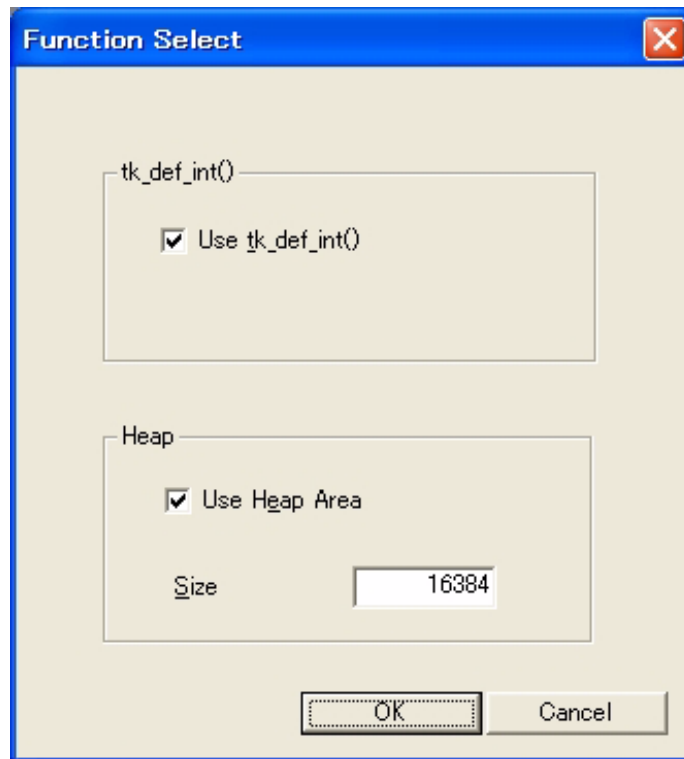
Figure 5.3-3 Size Spec Window

The Size Spec window displays two input fields for stack sizes. The parameters and their values are as follows:

Parameter	Value
Initial Task Stack Size	8*1024
System Stack Size	1024

At the bottom, there are buttons for 'OK' and 'Cancel'.

- Function Select
- Double clicking "Function Select" on the project window will display Figure 5.3-4. Set the items of "Function Select" in Table 3.13-1.

Figure 5.3-4 Function Select Window

- Entry registration

Performs registration of an initial routine, an error routine, an interrupt handler, and a power saving routine (power saving function).

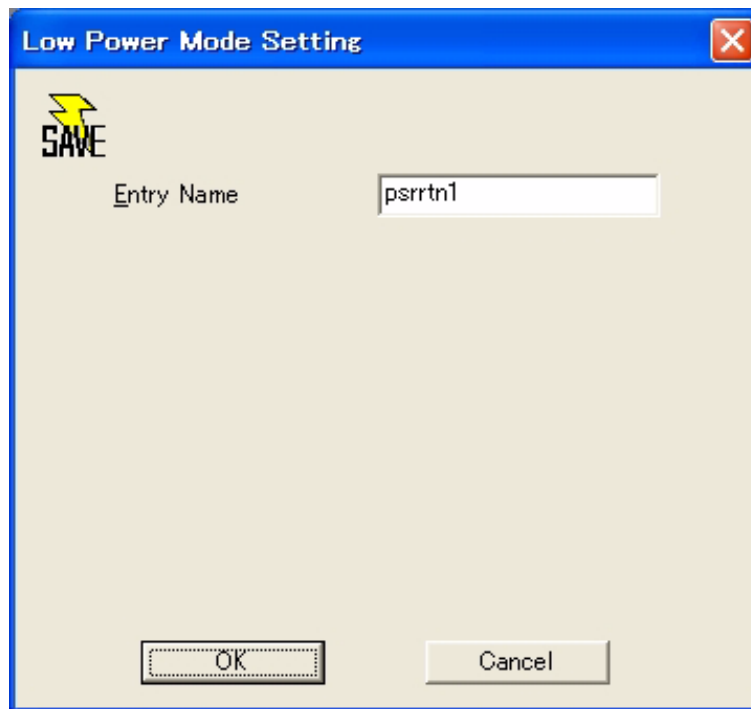
Double clicking "Initial routine", "Error Routine", or "Power saving function" on the project window will display Figure 5.3-5, Figure 5.3-6, and Figure 5.3-7 respectively. Set the entry name of functions used in the initial routine, error routine, and power saving routine.

Figure 5.3-5 Initialize Routine Setting Window

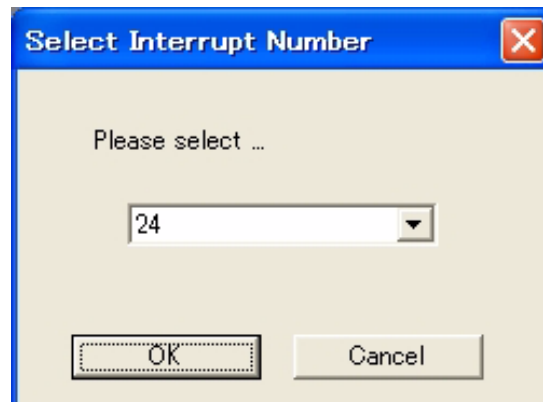


Figure 5.3-6 Error Routine Setting Window



Figure 5.3-7 Low Power Mode Window

New registration of an interrupt handler will display Figure 5.3-8 at the first. Select an interrupt number corresponding to the registered handler in this window.

Figure 5.3-8 Select Interrupt Number Window

Clicking the "OK" button at Figure 5.3-8 will display Figure 5.3-9. Set the entry name of functions used in the interrupt handler.

Figure 5.3-9 Interrupt Handler Setting Window

- **Debug**

Specifies the type of debug module used in the log function of μ T-REALOS analyzer (module log). Double clicking the names of "Type1", "Type2", "Type3", and "Type5" on the project window of the configurator first or selecting the [setting] menu by right clicking will set the selected debug modules.

For the module log, see "2.4 Log", and "CHAPTER 3 TASK ANALYSIS MODULE" of "Analyzer Guide".

■ Execution of Configuration

Configuration will be executed automatically when the project is built.

5.4 Setting of Linker Option

This section describes how to set the linker option

■ Memory map setting of ROM and RAM

Set the memory area used as ROM/RAM area according to the user system.

The sample program below allocates the ROM/RAM area to 1MB and 4MB respectively.

Table 5.4-1 Memory Map of ROM/RAM Area in the Sample Program

Area attribute	Area name	Start address	End address
ROM Image	_ROM_	0x00800000	0x008FFFFFFF
RAM Image	_RAM_	0x80000000	0x803FFFFFFF

■ Kernel specific section

Section names used in the kernel are described as follows:

Table 5.4-2 Kernel Specific Section List

Type	Section Name	Meaning
CONST	INTVECT	Vector Table
DATA	SYSINFO	Object management table
STACK	_KERNEL_STACK_SC	System stack area
DATA	HEAP	Kernel heap area

INTVECT is allocated in the ROM area, while SYSINFO and _KERNEL_STACK_SC are allocated in the RAM area.

For details of "Type" mentioned in the above table, see "5.3 Types of Section" in "SOFTUNE Linkage Kit Manual" (hereinafter, "Linkage Kit Manual").

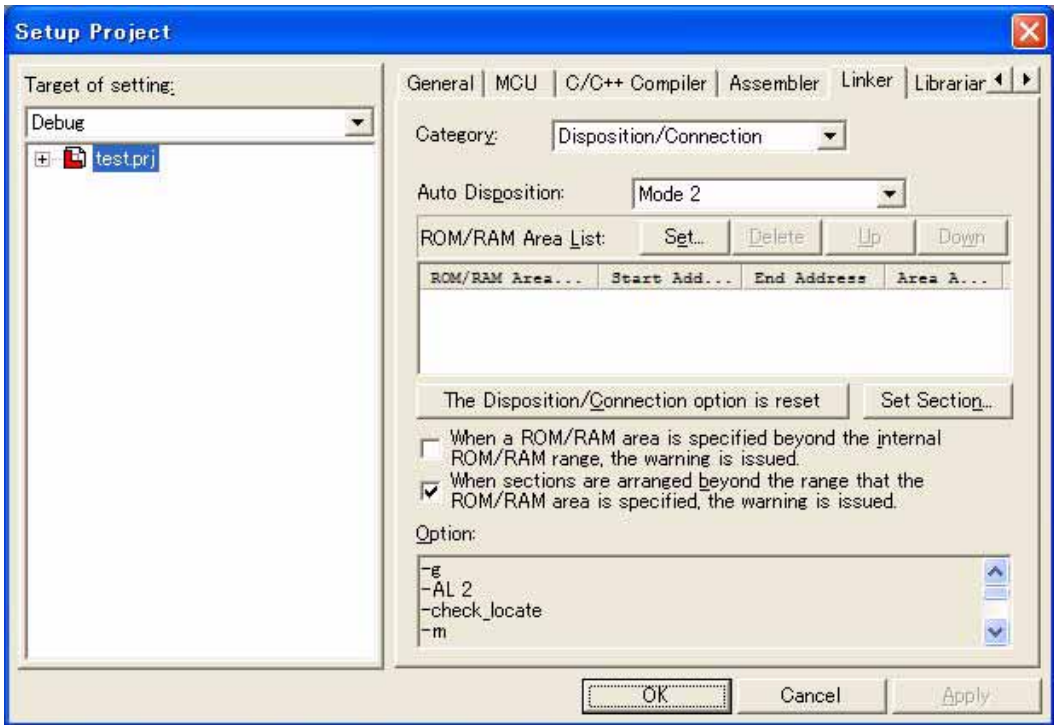
The start address of the vector table is set in table base register (TBR) of CPU. Therefore, INTVECT section will usually be specified when fixed addresses are linked.

■ How to specify the memory map

Memory map will be specified as the linker's option when the user system is linked. The operation method is described below. For details of the linker, see "PARTII LINKER" in "Linkage Kit Manual".

- Select [Project]-[Project settings] menu on the SOFTUNE Workbench. Click the "Linker" tab, then click the "add" button of the "ROM/RAM Area List" at the category of "Allocate/merge" in the project settings dialog.

Figure 5.4-1 Setup Project Dialog



- Input the attributes of RAM area and ROM area, start address, and end address, then click the "OK" button.

Figure 5.4-2 Setup RAM Area Name

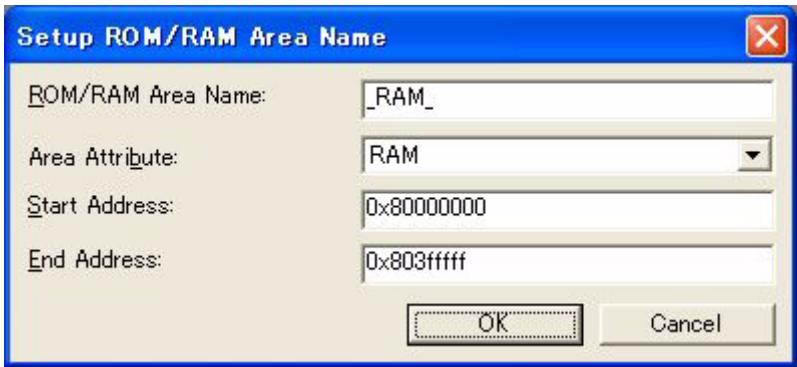
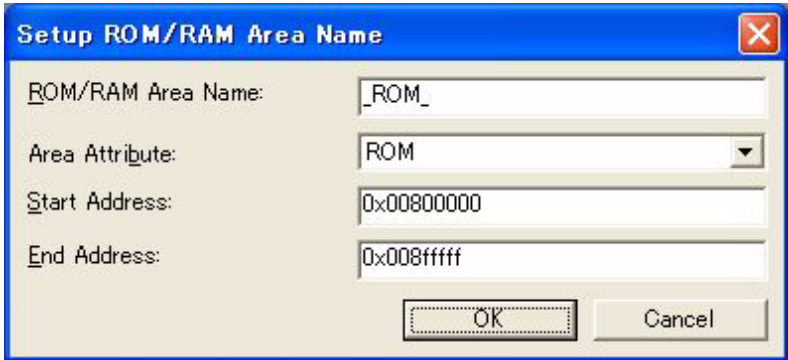
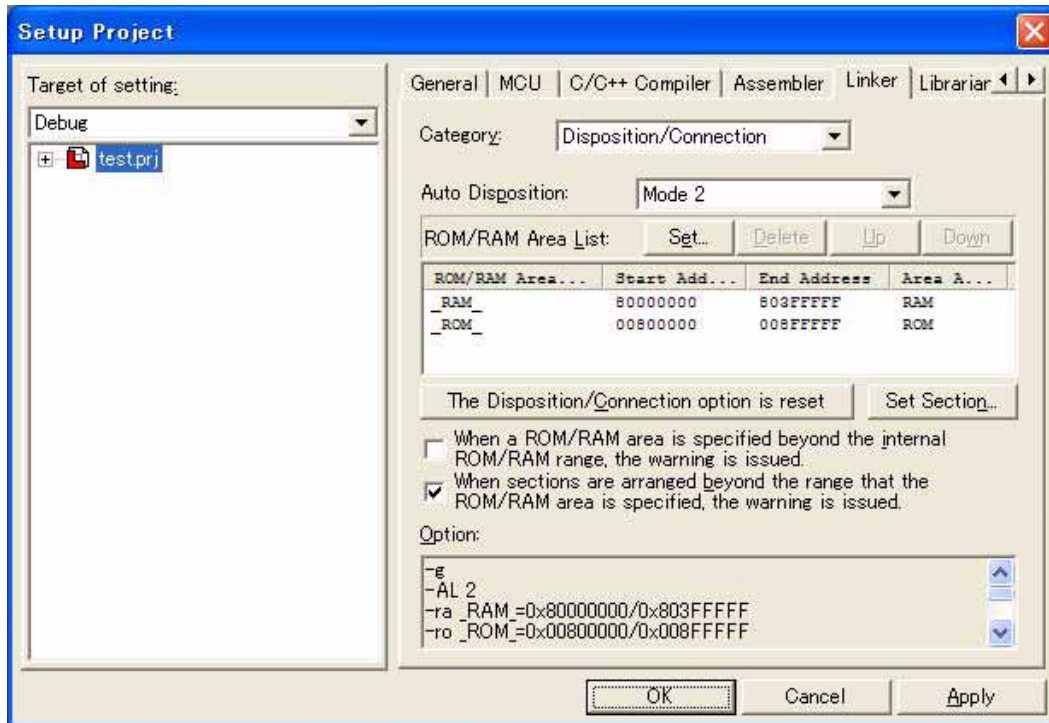


Figure 5.4-3 Setup ROM Area Name



- Set the address of the INTVECT section next. Click the "Setup Section" button on the "Setup Project" dialog.

Figure 5.4-4 Setup Project



- Select "Specify address" in the ROM/RAM area name in the "Setup Section" dialog. Input "INTVECT" in the section name, and input the start address of the INTVECT section in "Address", then click "OK" button.

Figure 5.4-5 Setup Section

Setup Section

ROM/RAM Area Name: Specify in Address

Section Name: INTVECT Set

Address: H'008FFC00

Contents Type: None

Section Name List:

INTVECT=H'008FFC00 Delete

Up

Down

OK Cancel

■ Link objects

Table 5.4-3 is a list of sample programs relevant to object files that are required for creating a user system. No user setting is required since these files will be automatically set as Link Option when REALOS Project is created, or when a user program is registered.

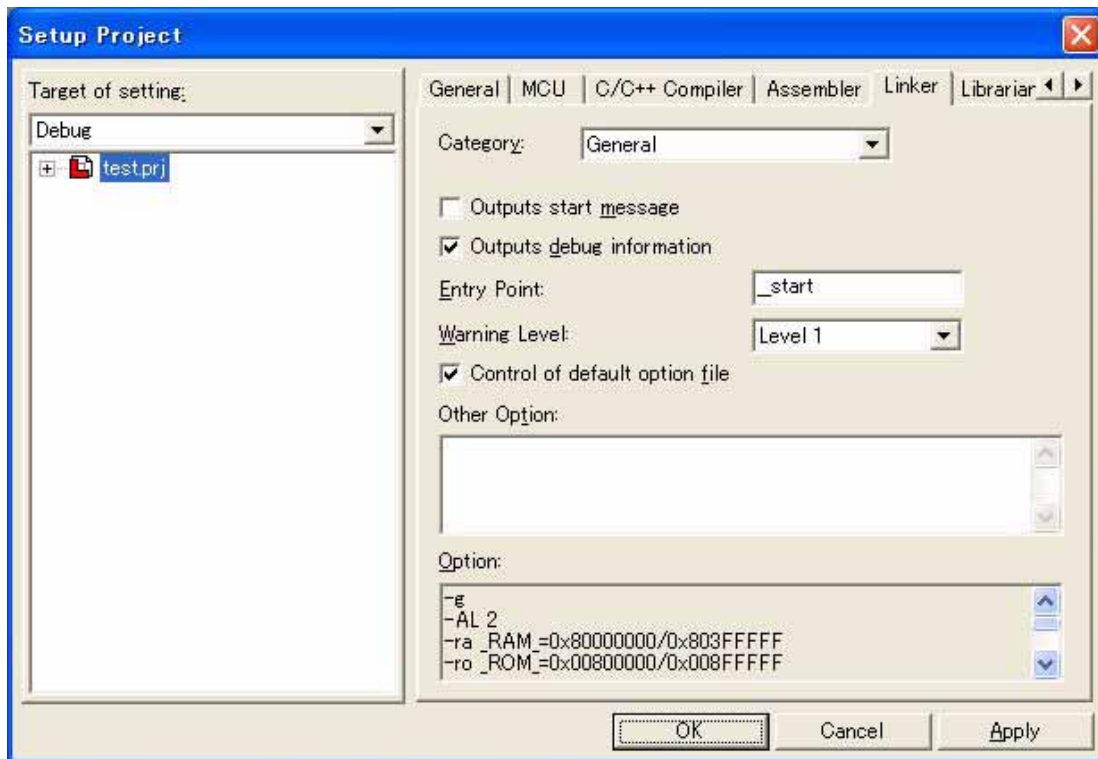
Table 5.4-3 Link Object List

Classification	File Name	Remarks
User Program (Sample Program)	icrt0.obj	Reset entry routine
	init_task.obj	Task, timer interrupt handler
Kernel configuration file	config.lib	File name is specified at the configuration startup
Kernel Library	libtm.lib	File name is fixed
	libtstdlib.lib	
	libstr.lib	
	libtk.lib	
	libtkernel.lib	

■ Reset entry settings

Set the address of the reset entry in the "Entry point" column at the "General" category of the project settings dialog. The example below sets the address of the "__start" symbol to the reset entry.

Figure 5.4-6 Reset Entry Settings



5.5 Build a User System

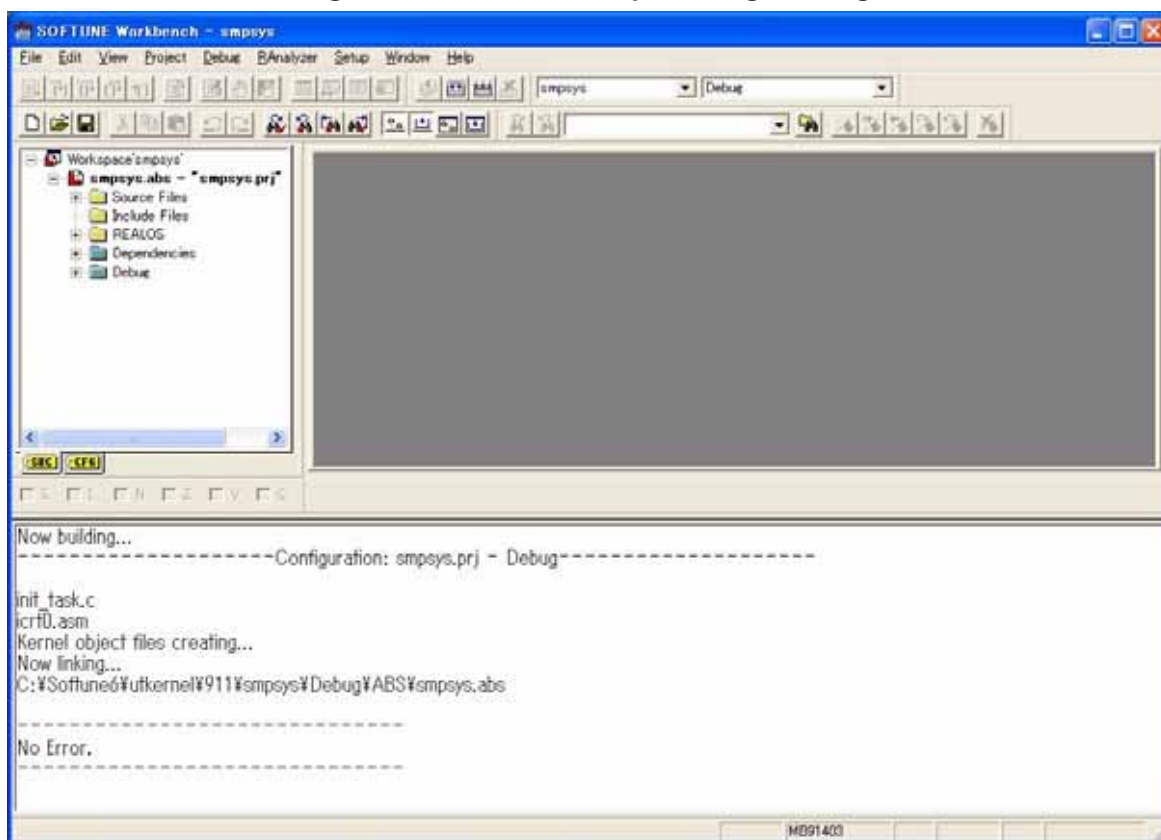
This section describes how to build a user system

■ Build a user system

Build the system to create an object in the executable format, which contains μ T-REALOS. Build automatically performs compilation and configuration of the user program and link with μ T-REALOS.

- Select [Project]-[Make] and [Build] on SOFTUNE Workbench. Build results will be displayed in the output window at the bottom of the SOFTUNE Workbench Window.

Figure 5.5-1 Screen example during building



APPENDIX

The appendix describes error messages of the configurator.

APPENDIX A Error Messages of the Configurator

APPENDIX A Error Messages of the Configurator

Appendix A describes the categorization, display format, and meaning of error messages output during configuration.

■ Error Message Categorization of the Configurator

Error messages output by the configurator on execution of configuration are categorized by importance into the following three levels.

- Warning message

Warning messages are less serious than the error messages described next, and the output results can be used almost without problems.

Occasionally, a process different from what the user intended may be performed.

Determine whether the output results are usable after checking the message contents.

- Error message

The process is continued. However, the configuration is not performed. It is necessary to eliminate the factors causing the error and perform the execution again.

This error mainly occurs while reading a file.

- Fatal error message

An error indicating that the process cannot be continued. Such an error may occur due to problems in the execution environment as well as wrong specification by the user.

In addition, there are messages that are output by the compiler, assembler, or linker executed inside the configurator. For messages output by the compiler, assembler, or linker, see the corresponding manual.

■ Display Format of Configurator Error Messages

Error messages are output in the following formats:

*** <u>File name(line number)</u> XnnnnT:Message text(<u>Assist message</u>)
--

Section	Description
File name (Line number)	Configuration file name and line number where the error occurred. Output when error occurred while reading the configuration file.
X	Error level is expressed using one of the following three characters. W Warning message E Error message F Fatal error message
nnnn	Error number The error number and error level are associated as follows: 1000 to 1999 W 4000 to 4999 E 9000 to 9999 F
T	Tool identification is expressed using the following character. M Configurator C Compiler A Assembler L Linker
Message text	Error message text
Assist message	More detailed information regarding the error. The symbol name indicating the error occurrence is displayed. It may be output to the error message body.

Note:

An error may occur at the compiler, assembler, or linker launched by the configurator ("C", "A", or "L" are respectively output to the tool identification.)

For details of an error in such a case, see the corresponding manual.

■ Description of Message Notation

Warning messages, error messages, and fatal error messages from the configurator are hereafter described in the following format:

Error code	English message
------------	-----------------

The variable character string in the message is described with underlines.

■ Warning Messages

W1130M	Multiple definition (<u>Definition name</u>)
--------	--

The definition displayed in the definition name was duplicated.

This definition was overwritten by the definition specified later.

When a number is displayed in the definition name, the ID indicated by the number was overwritten by the definition specified later.

W1401M	Not found maximum area definition (<u>Definition name</u>)
--------	--

Maximum area definition name displayed by the definition name does not exist.

When this error message is output, the maximum size is assigned automatically.

W1405M	EIT vector No." <u>num</u> " is system reserve
--------	--

As the interrupt number specified by "num" is system reserved, it cannot be used.

This error message is output when the interrupt number of an interrupt handler defined in "ATT_INI" is system reserved.

■ Error Messages

E4024M	Illegal character (<u>Parameter</u>)
--------	--

Characters that cannot be used in parameter displayed by parameter are contained

This error occurs when characters are specified in Parameter requiring numbers, or when numbers are specified in Parameter requiring labels.

E4026M	Specified value is out of range (<u>Parameter</u>)
--------	--

The value displayed in Parameter is out of range that can be specified.

This error occurs when a value exceeding 32767 was specified in object ID.

E4110M	Unknown API name (<u>Character string</u>)
--------	--

Cannot use the definition displayed in the character string.

This error occurs when unsupported API name is described.

E4111M	Too long line (MAX <u>value</u>)
--------	-----------------------------------

Description is not available when the line length exceeds the length displayed in MAX value.

Limit the description to the length displayed using MAX value.

E4112M	Illegal parameter expression
--------	------------------------------

Expression shown in line number is illegal.

This error message is output when the description syntax or the definition method of API is illegal.

E4120M	<u>Parameter</u> is too long
--------	------------------------------

The parameter length shown in Parameter is too long.

This message is output when a symbol is described with a length exceeding its specification.

E4121M	Too many <u>Definition name</u> (MAX <u>value</u>)
--------	---

Definition displayed in Definition name exceeds the number displayed in the value.
This error message is output if an attempt is made to define more than standard API.

E4123M	Too many parameters (<u>Parameter</u>)
--------	--

Parameter beyond those shown at Parameter are unnecessary.

E4125M	Short of parameter
--------	--------------------

The parameters of the definition name are inadequate.

E4130M	Multiple definition (<u>Definition name</u>)
--------	--

Definition that cannot be duplicated was duplicated and defined.
This error message is output when the API ID is duplicated.

E4131M	Parameter not defined (<u>Parameter name</u>)
--------	---

Parameters that cannot be omitted were omitted.
The definition displayed in the Parameter name cannot be omitted.

E4132M	Illegal parameter (<u>Parameter</u>)
--------	--

Parameter that cannot be specified was specified.
This error is displayed mainly when a string that cannot be selected in the selection item is specified.

E4133M	Symbol is already defined (<u>Symbol name</u>)
--------	--

A symbol that has already been defined was redefined.
This error occurs mainly when the task or event flag names, specified by API are duplicated.

E4136M	Illegal size or address (<u>value</u>)
--------	--

The specified size or address is not correct.

E4142M	Device open count is bigger than semaphore count (MAX <u>value</u>)
--------	--

The device open number was specified with a value larger than the semaphore number.
Lower the device open number, or increase the semaphore number.

E4402M	API ID exceed maximum area definition (<u>Parameter</u>)
--------	--

More APIs displayed by Parameter were defined than the value defined by the maximum area.
This error message is output even when the API is defined with no maximum area defined.

■ Fatal Error Messages

F9000M	Environment variable not found (<u>Environment variable name</u>)
--------	---

The environment variable displayed in Environment variable name is not defined.

F9001M	Insufficient memory
--------	---------------------

Insufficient memory for program execution.

F9002M	Not configured
--------	----------------

Configuration was not performed.

This error message is output when execution is interrupted due to an error during configuration.

F9011M	Input file is not found (<u>File name</u>)
--------	--

The specified input file is not found.

F9016M	Error read error (<u>File name</u>)
--------	---------------------------------------

Reasons such as file without read privilege, hardware problems can be considered.

F9017M	File write error (<u>File name</u>)
--------	---------------------------------------

Reasons such as file without write privilege, presence of the same directory or no free disk space can be considered.

F9022M	Unknown option name (<u>Option</u>)
--------	---------------------------------------

A parameter that cannot be specified was specified.

F9023M	Illegal option parameter (<u>Parameter</u>)
--------	---

The parameter displayed in Parameter is illegally specified.

F9024M	Option parameter not specified (<u>Option</u>)
--------	--

Parameter is not specified in Option specified by option.

F9030M	Missing input file name
--------	-------------------------

The configuration file was not specified.

F9033M	Illegal file format (<u>Parameter</u>)
--------	--

This error occurs when format of files such as CPU information file is illegal.

F9405M	Initial task priority is higher than maximum task priority (MAX <u>value</u>)
--------	--

The initial task priority is higher than the maximum priority.

Lower the initial task priority, or increase the maximum task priority.

F9501M	Not found CPU information file
--------	--------------------------------

The CPU information file is not found at the specified location.

F9502M	Not found CPU information
--------	---------------------------

This error occurs when the MB number specified by -cpu option has not been registered to the CPU information file.

Confirm the MB number specified by -cpu option.

F9801M	<u>Definition name</u> is not defined
--------	---------------------------------------

The definition content displayed using the Definition name is not defined.

This error message is output when a definition or an option that cannot be omitted has not been specified.

F9805M	EIT vector No. <u>Number</u> is system reserve
--------	--

Cannot use the vector Number displayed using number is system reserved.

F9895M	Error in Compiler (<u>File name</u>)
--------	--

Error occurred when compiling the displayed file.

F9897M	Error in Assembler (<u>File name</u>)
--------	---

Error occurred when assembling the displayed file.

F9898M	Error in Linker
--------	-----------------

Error occurred in the linker.

F9899M	<u>Tool name</u> is not found
--------	-------------------------------

The compiler, assembler, or linker could not be found from environment variable "PATH".
Define a path where the tool is contained in environment variable "PATH".

F9990M	File I/O error (<u>File name</u> , <u>Information</u>)
--------	--

Some error occurred during input/output of a file.

F9993M	Cannot create directory (<u>Directory name</u>)
--------	---

Failed to create a directory displayed using Directory name.

The reasons such as no directory writing privilege, presence of the same name directory name, no free disk space are considered.

F9994M	Cannot create file (<u>File name</u>)
--------	---

Failed to create a file displayed using File name.

The reasons such as no file writing privilege, presence of the same name directory name, no free disk space are considered.

APPENDIX A Error Messages of the Configurator

F9995M	Cannot close file (<u>File name</u>)
--------	--

Failed to close a file displayed using File name.

The reasons such as no file writing privilege, presence of the same name directory name, no free disk space are considered.

F9996M	Cannot open file (<u>File name</u>)
--------	---------------------------------------

Failed to open a file displayed using File name.

The reasons such as no file writing privilege, presence of the same name directory name, no free disk space are considered.

F9999M	Internal error (<u>Identification information</u>)
--------	--

When this error occurs, please contact sales representative immediately.

INDEX

The index follows on the next page.
This is listed in alphabetic order.

Index

A	
abortfn	
Creating a Process Abort Function (abortfn)	94
Additional	
Additional Notes	41
Additional Notes	
Additional Notes	36, 39, 43
Alarm Handler	
Alarm Handler Functions	52
Alarm Handlers	17
Creating an Alarm Handler	86
Description Format of an Alarm Handler	86
Starting an Alarm Handler	86
API	
Static API	62
Assistance	
Overview of the Debugging Assistance Functions	64
B	
Breakpoints	
OS Breakpoints	65
C	
Calling Format	
Calling Format of an Expansion SVC Handler	91
Close Function	
Creating a Close Function (closefn)	93
closefn	
Creating a Close Function (closefn)	93
Communication	
Extended Synchronization and Communication Functions	37
Synchronization and Communication Functions	31
Configuration	
Configuration Functions	59
Execution of Configuration	108
Setting of Configuration	102
Setting Operation of Configuration	103
Configuration Definition	
Configuration Definition Macros	60
Configurator	
Display Format of Configurator Error Messages	117
Error Message Categorization of the Configurator	116
Running the Configurator	63
Creating	
Creating a Close Function (closefn)	93
Creating a Process Abort Function (abortfn)	94
Creating a Process Start Function (execfn)	93
Creating a Task	81
Creating a Waiting for Completion Function (waitfn)	93
Creating an Alarm Handler	86
Creating an Open Function (openfn)	92
Creation	
Creation of a Period Handler	85
Current Task	
Current Task and Other Tasks	10
Cyclic Handler	
Cyclic Handler Functions	50
Cyclic Handlers	16
D	
Debugging	
Overview of the Debugging Assistance Functions	64
Debugging Assistance	
Overview of the Debugging Assistance Functions	64
Description Example	
Description Example of the Initial Routine	79
Description Format	
Description Format of a Period Handler	85
Description Format of a Power Saving Routine	90
Description Format of an Alarm Handler	86
Description Format of an Error Routine	89
Description Format of an Extension SVC Handler	91
Description Format of an Interrupt Handler	87
Description Format of the Initial Routine	79
Description Format of the Task	81
Development	
Tools Required for Development	4
Device	
Determining A Device Name	92
Device Driver Interface	92
Device Management Functions	56
Device Processing Functions	20
Device Driver Interface	
Device Driver Interface	92
Device Management	
Device Management Functions	56

Device Name		
Determining A Device Name	92	
Directory Structure		
Directory Structure of Provided Files	3	
Dispatch		
Dispatch Enabled/disabled States.....	24	
Dispatching		
Dispatching and Preemption.....	10	
Display Format		
Display Format of Configurator Error Messages	117	
Driver		
Device Driver Interface	92	
E		
Error Message		
Display Format of Configurator Error Messages	117	
Error Message Categorization of the Configurator	116	
Error Messages	120	
Fatal Error Messages	123	
Error Routine		
Description Format of an Error Routine	89	
Error Routines	18	
Registering an Error Routine	89	
Event Flag		
Event Flag Functions.....	34	
Event Function		
Event Function (eventfn)	94	
eventfn		
Event Function (eventfn)	94	
Example		
A specific Example of the Task	82	
execfn		
Creating a Process Start Function (execfn)	93	
Expansion SVC Handler		
Calling Format of an Expansion SVC Handler	91	
Extended SVC Handler		
Extended SVC Handlers	19	
Extended Synchronization		
Extended Synchronization and Communication	37	
Extension SVC Handler		
Description Format of an Extension SVC Handler	91	
F		
Fatal Error Message		
Fatal Error Messages	123	
Fixed-size Memory Pool		
Fixed-size Memory Pool Functions.....	46	
Format		
Calling Format of an Expansion SVC Handler.....	91	
Description Format of a Period Handler	85	
Description Format of a Power Saving Routine	90	
Description Format of an Alarm Handler	86	
Description Format of an Error Routine	89	
Description Format of an Extension SVC Handler	91	
Description Format of an Interrupt Handler	87	
Description Format of the Initial Routine	79	
Description Format of the Task	81	
Function		
Creating a Close Function (closefn).....	93	
Creating a Process Abort Function (abortfn).....	94	
Creating a Process Start Function (execfn)	93	
Creating a Waiting for Completion Function (waitfn)	93	
Creating an Open Function (openfn).....	92	
Event Function (eventfn)	94	
H		
Handler		
Alarm Handler Functions	52	
Alarm Handlers	17	
Calling Format of an Expansion SVC Handler.....	91	
Creating an Alarm Handler	86	
Creation of a Period Handler.....	85	
Cyclic Handler Functions	50	
Cyclic Handlers	16	
Description Format of a Period Handler	85	
Description Format of an Alarm Handler	86	
Description Format of an Extension SVC Handler	91	
Description Format of an Interrupt Handler	87	
Extended SVC Handlers.....	19	
Interrupt Handlers.....	15	
Launch of a Period Handler	85	
Precedence of Execution (Handlers vs. Handlers)	26	
Precedence of Execution (Tasks vs. Interrupt Handlers and Time Event Handlers)	25	
Registering an Interrupt Handler	87	
Starting an Alarm Handler.....	86	
Time Event Handlers	16	
Timer Interrupt Handler	88	
I		
Initial Routine		
Description Example of the Initial Routine.....	79	
Description Format of the Initial Routine	79	
Initial Routines	14	
Process of the Initial Routine	79	
Interface		
Device Driver Interface	92	

INDEX

Interrupt	
Interrupt Management Functions	53
Notes on Interrupt	96
Timer Interrupt Handler	88
Interrupt Handler	
Description Format of an Interrupt Handler.....	87
Interrupt Handlers	15
Precedence of Execution (Tasks vs. Interrupt Handlers and Time Event Handlers).....	25
Registering an Interrupt Handler	87
Interrupt Management	
Interrupt Management Functions	53
Interrupts	
Interrupts Enabled/disabled States.....	24
K	
Kernel	
Kernel specific section.....	109
Kernel specific section	
Kernel specific section.....	109
L	
Launch	
Launch of a Period Handler	85
Link objects	
Link objects	112
Logs	
Logs.....	65
M	
Macros	
Configuration Definition Macros	60
Mailbox	
Mailbox Functions.....	35
Management	
Device Management Functions	56
Interrupt Management Functions	53
Subsystem Management Functions	55
System State Management Functions	54
Memory map	
How to specify the memory map.....	109
Memory map setting of ROM and RAM.....	109
Memory Pool	
Fixed-size Memory Pool Functions	46
Memory Pool Management Functions	45
Variable-size Memory Pool Functions.....	47
Memory Pool Management	
Memory Pool Management Functions	45
Message	
Description of Message Notation	118
Display Format of Configurator Error Messages	117
Error Message Categorization of the Configurator	116
Error Messages	120
Fatal Error Messages.....	123
Warning Messages	119
Message Buffer	
Message Buffer Functions	40
μT-REALOS	
Create the μT-REALOS Project	99
Overview of μT-REALOS Functions	28
Mutex	
Mutex Functions.....	38
N	
Non-task	
Non-task Portion Running	22
Notation	
Description of Message Notation.....	118
O	
Object List	
Object List Display	64
Objects	
Link objects.....	112
Objects	21
Open Function	
Creating an Open Function (openfn)	92
openfn	
Creating an Open Function (openfn)	92
OS Breakpoints	
OS Breakpoints.....	65
P	
Period Handler	
Creation of a Period Handler	85
Description Format of a Period Handler	85
Launch of a Period Handler	85
Power Saving	
Power Saving Functions	58
Power Saving Routine	
Description Format of a Power Saving Routine	90
Registering a Power Saving Routine	90
Preemption	
Dispatching and Preemption	10
Priority	
Priority Sequence and Task Priorities.....	10
Priority Sequence	
Priority Sequence and Task Priorities.....	10
Process	
Process of Reset Entry Routine	76
Process of the Initial Routine	79

Process Abort Function	
Creating a Process Abort Function (abortfn)	94
Process Start Function	
Creating a Process Start Function (execfn)	93
Product	
Structure of Product	5
Program	
Notes on the Overall of a Program	95
Project	
Create the μ T-REALOS Project	99
R	
RAM	
Memory map setting of ROM and RAM	109
Registering	
Registering a Power Saving Routine	90
Registering an Error Routine	89
Rendezvous Port	
Rendezvous Port Functions	42
Reset	
A specific Example of the Reset Entry Routine	
.....	77
Process of Reset Entry Routine.....	76
Reset Entry Routine	76
Reset Entry Settings	113
Reset Entry	
Reset Entry Settings	113
Reset Entry Routine	
A specific Example of the Reset Entry Routine	
.....	77
Process of Reset Entry Routine.....	76
Reset Entry Routine	76
ROM	
Memory map setting of ROM and RAM	109
Routine	
A specific Example of the Reset Entry Routine	
.....	77
Description Example of the Initial Routine	79
Description Format of a Power Saving Routine	
.....	90
Description Format of an Error Routine	89
Description Format of the Initial Routine	79
Process of Reset Entry Routine.....	76
Process of the Initial Routine.....	79
Registering a Power Saving Routine	90
Registering an Error Routine	89
Reset Entry Routine	76
Running	
Non-task Portion Running.....	22
Task Portion Running.....	22
S	
section	
Kernel specific section	109
Semaphore	
Semaphore Functions	32
specific Example	
A specific Example of the Reset Entry Routine.....	77
A specific Example of the Task.....	82
Stack	
Stack Information	70
Starting	
Starting a Task	82
Starting an Alarm Handler	86
Static API	
Static API	62
Subsystem	
Subsystem Management Functions	55
Subsystem Management	
Subsystem Management Functions	55
SVC	
Calling Format of an Expansion SVC Handler.....	91
Description Format of an Extension SVC Handler	
.....	91
Extended SVC Handlers	19
Synchronization	
Extended Synchronization and Communication	
Functions	37
Synchronization and Communication Functions	
.....	31
System	
Notes on the Overall of a System Call	95
System Calls	
Issuing System Calls	69
System Calls	8
System Calls that can be called	23
System Construction	
Steps of System Construction	98
System State	
System State Management Functions	54
System States	22
User Programs and System States.....	23
System State Management	
System State Management Functions	54
System Time	
Setting and Getting the System Time	49
System Time	49
Updating the System Time	49
System Uptime	
Getting the System Uptime	49
T	
Task	
A specific Example of the Task.....	82

INDEX

Creating a Task	81
Current Task and Other Tasks	10
Description Format of the Task.....	81
Non-task Portion Running	22
Notes on a Task.....	95
Precedence of Execution (Tasks vs. Interrupt Handlers and Time Event Handlers)	25
Precedence of Execution (Tasks vs. Tasks)	25
Priority Sequence and Task Priorities	10
Starting a Task	82
Task Context Display	71
Task Management Functions	29
Task Portion Running	22
Task Portion Transitions	12
Task Portions	11
Task Synchronization Functions	30
Tasks	10
Task Context	
Task Context Display	71
Task Management	
Task Management Functions	29
Task Synchronization	
Task Synchronization Functions	30
Time Event Handler	
Precedence of Execution (Tasks vs. Interrupt Handlers and Time Event Handlers)	25
Time Event Handlers	16
Time Management	
Time Management Functions	48
Timer Interrupt Handler	
Timer Interrupt Handler.....	88
Tools	
Tools Required for Development.....	4
Transitions	
Task Portion Transitions	12
U	
User Program	
Configuring a User Program	74
Execution Units of User Program	9
Starting a User Program.....	75
User Programs and System States	23
user system	
Build a user system	114
V	
Variable-size Memory Pool	
Variable-size Memory Pool Functions	47
W	
waitfn	
Creating a Waiting for Completion Function (waitfn)	93
Waiting for Completion Function	
Creating a Waiting for Completion Function (waitfn)	93
Warning Message	
Warning Messages	119

CM81-00322-1E

FUJITSU MICROELECTRONICS • CONTROLLER MANUAL

FR Family

μ T-Kernel Specification Compliant

SOFTUNE™ μ T-REALOS

USER'S GUIDE

June 2008 the first edition

Published **FUJITSU MICROELECTRONICS LIMITED**

Edited Strategic Business Development Dept.
