REJ10J1523-0100

Everywhere you imagine. RENESAS

M3T-MR100/4 V.1.00

User's Manual

Real-time OS for R32C/100 Series

User's Manual

Rev.1.00
September 16, 2007

Renesas Technology
www.renesas.com

Keep safety first in your circuit designs!

● Renesas Technology Corporation and Renesas Solutions Corporation put the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage. Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of nonflammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

● These materials are intended as a reference to assist our customers in the selection of the Renesas Technology product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corporation, Renesas Solutions Corporation or a third party.
● Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
● All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corporation and Renesas Solutions Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corporation, Renesas Solutions Corporation or an authorized Renesas Technology product distributor for the latest product information before purchasing a product listed herein. The information described here may contain technical inaccuracies or typographical errors. Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors. Please also pay attention to information published by Renesas Technology Corporation and Renesas Solutions Corporation by various means, including the Renesas home page (http://www.renesas.com).
● When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, liability or other loss resulting from the information contained herein.
● Renesas Technology semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corporation, Renesas Solutions Corporation or an authorized Renesas Technology product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
● The prior written approval of Renesas Technology Corporation and Renesas Solutions Corporation is necessary to reprint or reproduce in whole or in part these materials.
● If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination. Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
● Please contact Renesas Technology Corporation or Renesas Solutions Corporation for further details on these materials or the products contained therein.

For inquiries about the contents of this document or product, fill in the text file the installer generates in the following directory and email to your local distributor.

\SUPPORT\Product-name\SUPPORT.TXT

Renesas Tools Homepage http://www.renesas.com/en/tools

Preface

The M3T-MR100/4(abbreviated as MR100) is a real-time operating system[1] for the R32C/100 series microcomputers. The MR100 conforms to the μITRON Specification.[2]

This manual describes the procedures and precautions to observe when you use the MR100 for programming purposes. For the detailed information on individual service call procedures, refer to the MR100 Reference Manual.

Requirements for MR100 Use

When creating programs based on the MR100, it is necessary to purchase the following product of Renesas.

- C-compiler package for R32C/100 series microcomputers (abbreviated as NC100)

Document List

The following sets of documents are supplied with the MR100.

- Release Note
  Presents a software overview and describes the corrections to the Users Manual and Reference Manual.
- Users Manual (PDF file)
  Describes the procedures and precautions to observe when using the MR100 for programming purposes.

Right of Software Use

The right of software use conforms to the software license agreement. You can use the MR100 for your product development purposes only, and are not allowed to use it for the other purposes. You should also note that this manual does not guarantee or permit the exercise of the right of software use.

---

[1] Hereinafter abbreviated "real-time OS"

[2] μITRON4.0 Specification is the open real-time kernel specification upon which the TRON association decided
The specification document of μITRON4.0 specification can come to hand from a TRON association homepage (http://www.assoc.tron.org/).
The copyright of μITRON4.0 specification belongs to the TRON association.

Contents

List of Figures

List of Tables

1. User's Manual Organization

The MR100 User's Manual consists of nine chapters and thee appendix.

- 2 General Information
  Outlines the objective of MR100 development and the function and position of the MR100.
- 3 Introduction to Kernel
  Explains about the ideas involved in MR100 operations and defines some relevant terms.
- 4 Kernel
  Outlines the applications program development procedure for the MR100.
- 5 Service call reffernce
  Details MR100 service call API
- 6 Applications Development Procedure Overview
  Details the applications program development procedure for the MR100.
- 7 Detailed Applications
  Presents useful information and precautions concerning applications program development with MR100.
- 8 Using Configurator
  Describes the method for writing a configuration file and the method for using the configurator in detail.
- 9 Sample Program Description
  Describes the MR100 sample applications program which is included in the product in the form of a source file.
- 10 Stack Size Calculation Method
  Describes the calculation method of the task stack size and the system stack size.
- 11 Note
  Presents useful information and precautions concerning applications program development with MR100.
- 12 Appendix
  Data type and assembly language interface.

2. General Information

2.1 Objective of MR100 Development

In line with recent rapid technological advances in microcomputers, the functions of microcomputer-based products have become complicated. In addition, the microcomputer program size has increased. Further, as product development competition has been intensified, manufacturers are compelled to develop their microcomputer-based products within a short period of time.

In other words, engineers engaged in microcomputer software development are now required to develop larger-size programs within a shorter period of time. To meet such stringent requirements, it is necessary to take the following considerations into account.

1. **To enhance software recyclability to decrease the volume of software to be developed.**

   One way to provide for software recyclability is to divide software into a number of functional modules wherever possible. This may be accomplished by accumulating a number of general-purpose subroutines and other program segments and using them for program development. In this method, however, it is difficult to reuse programs that are dependent on time or timing. In reality, the greater part of application programs are dependent on time or timing. Therefore, the above recycling method is applicable to only a limited number of programs.

2. **To promote team programming so that a number of engineers are engaged in the development of one software package**

   There are various problems with team programming. One major problem is that debugging can be initiated only when all the software program segments created individually by team members are ready for debugging. It is essential that communication be properly maintained among the team members.

3. **To enhance software production efficiency so as to increase the volume of possible software development per engineer.**

   One way to achieve this target would be to educate engineers to raise their level of skill. Another way would be to make use of a structured descriptive assembler, C-compiler, or the like with a view toward facilitating programming. It is also possible to enhance debugging efficiency by promoting modular software development.

However, the conventional methods are not adequate for the purpose of solving the problems. Under these circumstances, it is necessary to introduce a new system named real-time OS [3]

To answer the above-mentioned demand, Renesas has developed a real-time operating system, tradenamed MR100, for use with the R32C/100 series of 32-bit microcomputers .

When the MR100 is introduced, the following advantages are offered.

1. **Software recycling is facilitated.**

   When the real-time OS is introduced, timing signals are furnished via the real-time OS so that programs dependent on timing can be reused. Further, as programs are divided into modules called tasks, structured programming will be spontaneously provided.
   That is, recyclable programs are automatically prepared.

2. **Ease of team programming is provided.**

   When the real-time OS is put to use, programs are divided into functional modules called tasks. Therefore, engineers can be allocated to individual tasks so that all steps from development to debugging can be conducted independently for each task.
   Further, the introduction of the real-time OS makes it easy to start debugging some already finished tasks even if the entire program is not completed yet. Since engineers can be allocated to individual tasks, work assignment is easy.

3. **Software independence is enhanced to provide ease of program debugging.**

   As the use of the real-time OS makes it possible to divide programs into small independent modules called tasks,

[3] OS:Operating System

the greater part of program debugging can be initiated simply by observing the small modules.

4. Timer control is made easier.

To perform processing at 10 ms intervals, the microcomputer timer function was formerly used to periodically initiate an interrupt. However, as the number of usable microcomputer timers was limited, timer insufficiency was compensated for by, for instance, using one timer for a number of different processing operations.
When the real-time OS is introduced, however, it is possible to create programs for performing processing at fixed time intervals making use of the real-time OS time management function without paying special attention to the microcomputer timer function. At the same time, programming can also be done in such a manner as to let the programmer take that numerous timers are provided for the microcomputer.

5. Software maintainability is enhanced

When the real-time OS is put to use, the developed software consists of small program modules called tasks. Therefore, increased software maintainability is provided because developed software maintenance can be carried out simply by maintaining small tasks.

6. Increased software reliability is assured.

The introduction of the real-time OS makes it possible to carry out program evaluation and testing in the unit of a small module called task. This feature facilitates evaluation and testing and increases software reliability.

7. The microcomputer performance can be optimized to improve the performance of microcomputer-based products.

With the real-time OS, it is possible to decrease the number of unnecessary microcomputer operations such as I/O waiting. It means that the optimum capabilities can be obtained from microcomputers, and this will lead to microcomputer-based product performance improvement.

2.2 Relationship between TRON Specification and MR100

MR100 is the real-time operating system developed for use with the R32C/10 series of 32-bit microcomputers compliant with μITRON 4.0 Specification. μITRON 4.0 Specification stipulates standard profiles as an attempt to ensure software portability. Of these standard profiles, MR100 has implemented in it all service calls except for static APIs and task exception APIs

2.3 MR100 Features

The MR100 offers the following features.

1. **Real-time operating system conforming to the μITORN Specification.**

   The MR100 is designed in compliance with the μITRON Specification which incorporates a minimum of the ITRON Specification functions so that such functions can be incorporated into a one-chip microcomputer. As the μITRON Specification is a subset of the ITRON Specification, most of the knowledge obtained from published ITRON textbooks and ITRON seminars can be used as is.
   Further, the application programs developed using the real-time operating systems conforming to the ITRON Specification can be transferred to the MR100 with comparative ease.

2. **High-speed processing is achieved.**

   MR100 enables high-speed processing by taking full advantage of the microcomputer architecture.

3. **Only necessary modules are automatically selected to constantly build up a system of the minimum size.**

   MR100 is supplied in the object library format of the R32C/100 series.
   Therefore, the Linkage Editor functions are activated so that only necessary modules are automatically selected from numerous MR100 functional modules to generate a system.
   Thanks to this feature, a system of the minimum size is automatically generated at all times.

4. **With the C-compiler NC100, it is possible to develop application programs in C language.**

   Application programs of MR100 can be developed in C language by using the C compiler NC100. Furthermore, the interface library necessary to call the MR100 functions from C language is included with the software package.

5. **An upstream process tool named "Configurator" is provided to simplify development procedures**

   A configurator is furnished so that various items including a ROM write form file can be created by giving simple definitions.
   Therefore, there is no particular need to care what libraries must be linked.
   In addition, a GUI version of the configurator is available. It helps the user to create a configuration file without the need to learn how to write it.

3. Introduction to Kernel

3.1 Concept of Real-time OS

This section explains the basic concept of real-time OS.

3.1.1 Why Real-time OS is Necessary

In line with the recent advances in semiconductor technologies, the single-chip microcomputer ROM capacity has increased. ROM capacity of 32K bytes.

As such large ROM capacity microcomputers are introduced, their program development is not easily carried out by conventional methods. Figure 3.1 shows the relationship between the program size and required development time (program development difficulty).

This figure is nothing more than a schematic diagram. However, it indicates that the development period increases exponentially with an increase in program size.

For example, the development of four 8K byte programs is easier than the development of one 32K byte program.[4]

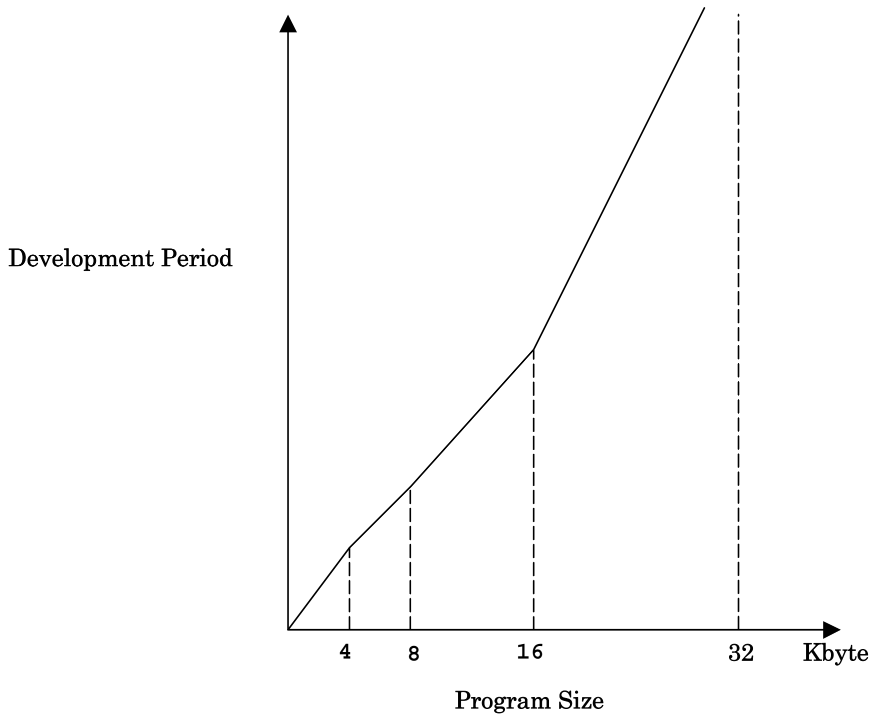

Figure 3.1 Relationship between Program Size and Development Period

Under these circumstances, it is necessary to adopt a method by which large-size programs can be developed within a short period of time. One way to achieve this purpose is to use a large number of microcomputers having a small ROM capacity. Figure 3.2 presents an example in which a number of microcomputers are used to build up an audio equipment system.

[4] On condition that the ROM program burning step need not be performed.

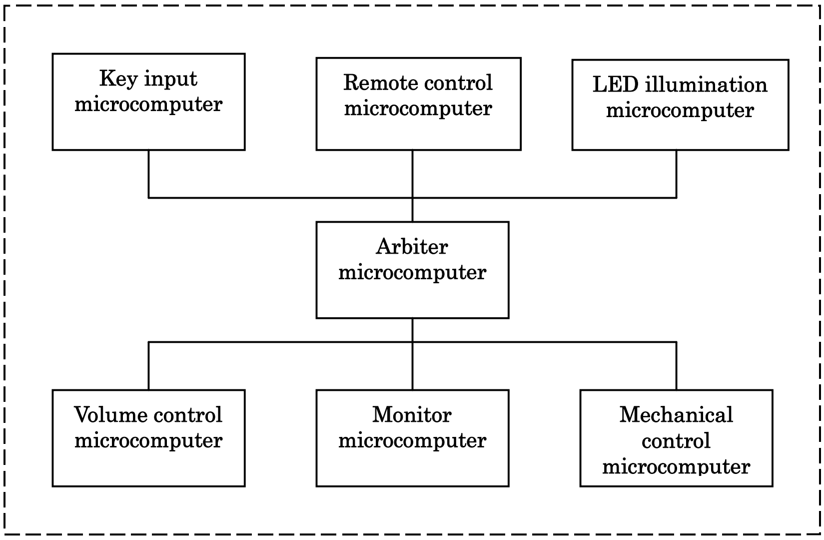

Figure 3.2 Microcomputer-based System Example(Audio Equipment)

Using independent microcomputers for various functions as indicated in the above example offers the following advantages.

1. **Individual programs are small so that program development is easy.**
2. **It is very easy to use previously developed software.**
3. **Completely independent programs are provided for various functions so that program development can easily be conducted by a number of engineers.**

On the other hand, there are the following disadvantages.

1. **The number of parts used increases, thereby raising the product cost.**
2. **Hardware design is complicated.**
3. **Product physical size is enlarged.**

Therefore, if you employ the real-time OS in which a number of programs to be operated by a number of microcomputers are placed under software control of one microcomputer, making it appear that the programs run on separate microcomputers, you can obviate all the above disadvantages while retaining the above-mentioned advantages.

Figure 3.3 shows an example system that will be obtained if the real-time OS is incorporated in the system indicated in Figure 3.2.

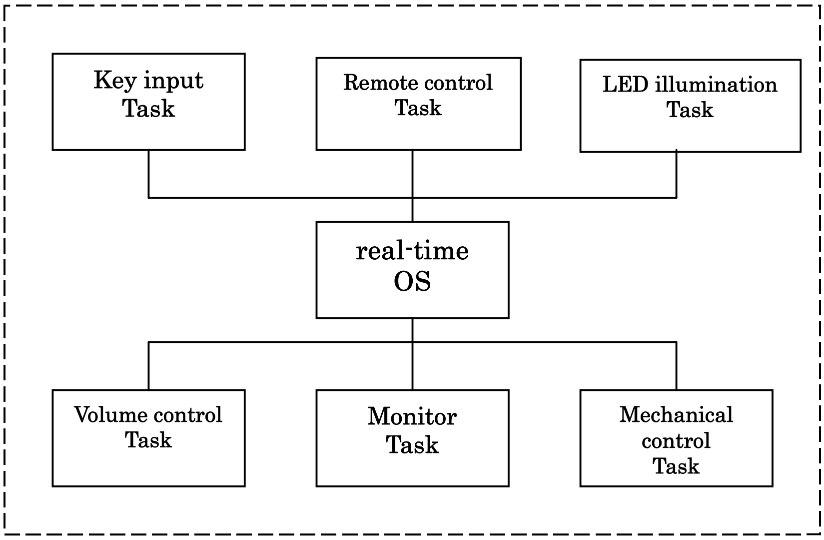

Figure 3.3 Example System Configuration with Real-time OS(Audio Equipment)

In other words, the real-time OS is the software that makes a one-microcomputer system look like operating a number of microcomputers.

In the real-time OS, the individual programs, which correspond to a number of microcomputers used in a conventional system, are called tasks.

3.1.2 Operating Principles of Kernel

A kernel is the core program of real-time OS. The kernel is the software that makes a one-microcomputer system look like operating a number of microcomputers. You should be wondering how the kernel makes a one-microcomputer system function like a number of microcomputers.

As shown in Figure 3.4 the kernel runs a number of tasks according to the time-division system. That is, it changes the task to execute at fixed time intervals so that a number of tasks appear to be executed simultaneously.

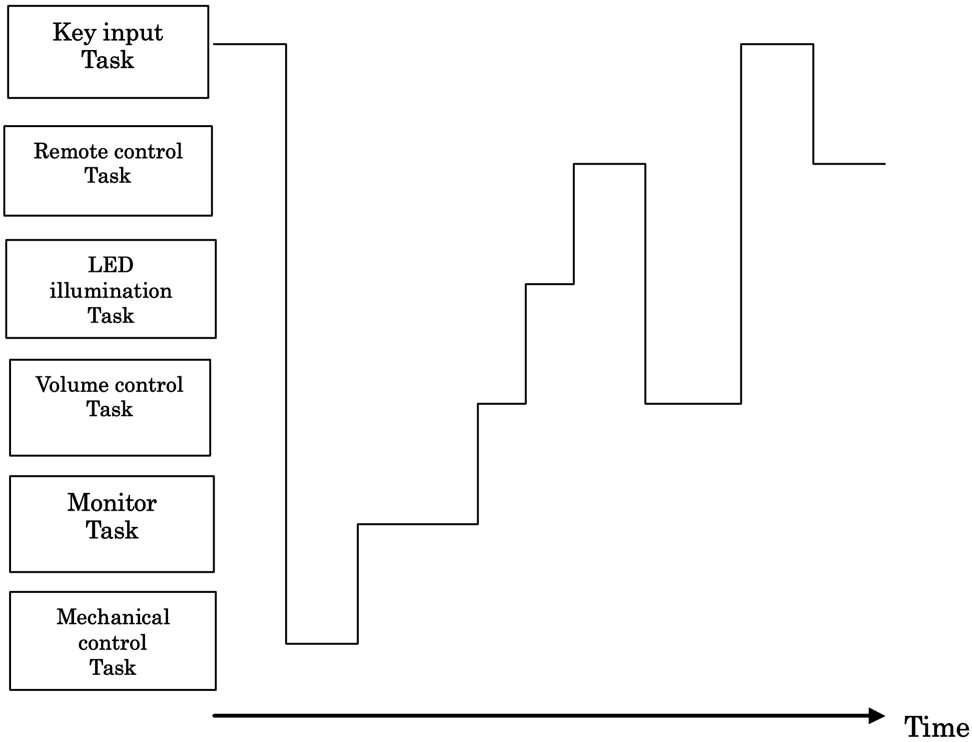

Figure 3.4 Time-division Task Operation

As indicated above, the kernel changes the task to execute at fixed time intervals. This task switching may also be referred to as dispatching. The factors causing task switching (dispatching) are as follows.

- Task switching occurs upon request from a task.
- Task switching occurs due to an external factor such as interrupt.

When a certain task is to be executed again upon task switching, the system resumes its execution at the point of last interruption (See Figure 3.5).

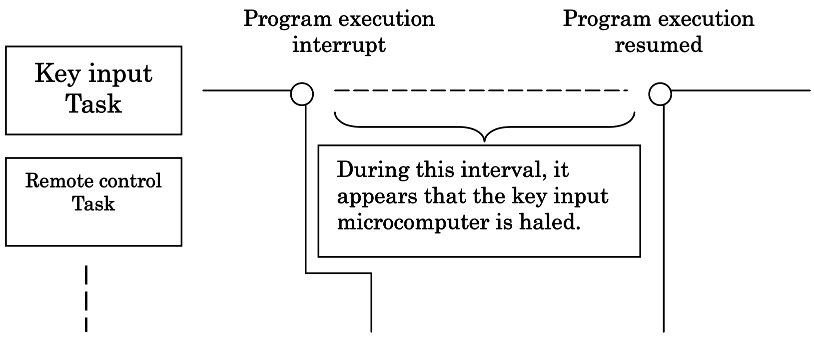

Figure 3.5 Task Execution Interruption and Resumption

In the state shown in Figure 3.5, it appears to the programmer that the key input task or its microcomputer is halted while another task assumes execution control.

Task execution restarts at the point of last interruption as the register contents prevailing at the time of the last interruption are recovered. In other words, task switching refers to the action performed to save the currently executed task register contents into the associated task management memory area and recover the register contents for the task to switch to.

To establish the kernel, therefore, it is only necessary to manage the register for each task and change the register contents upon each task switching so that it looks as if a number of microcomputers exist (See Figure 3.6).

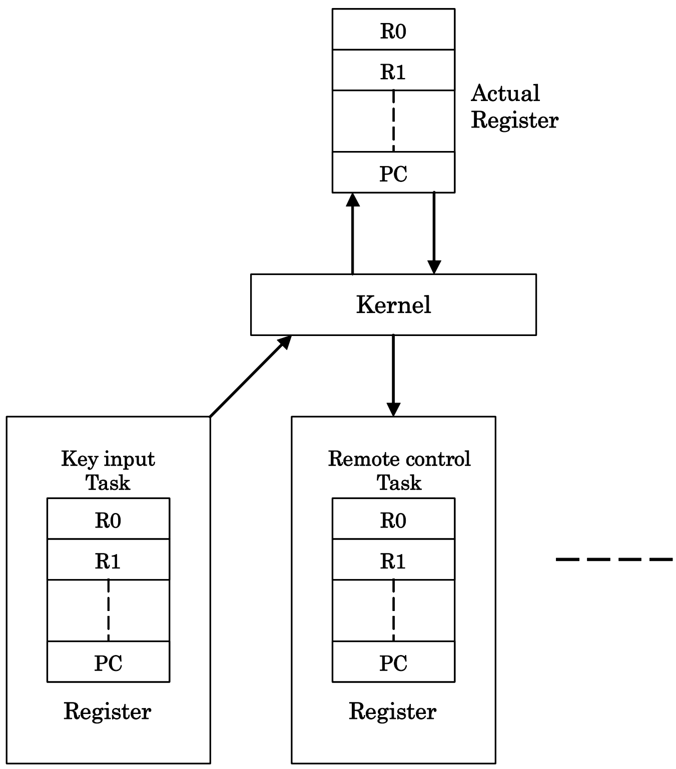

Figure 3.6 Task Switching

The example presented in Figure 3.7 [5] indicates how the individual task registers are managed. In reality, it is necessary to provide not only a register but also a stack area for each task.

[5] It is figure where all the stack areas of the task were arranged in the same section.

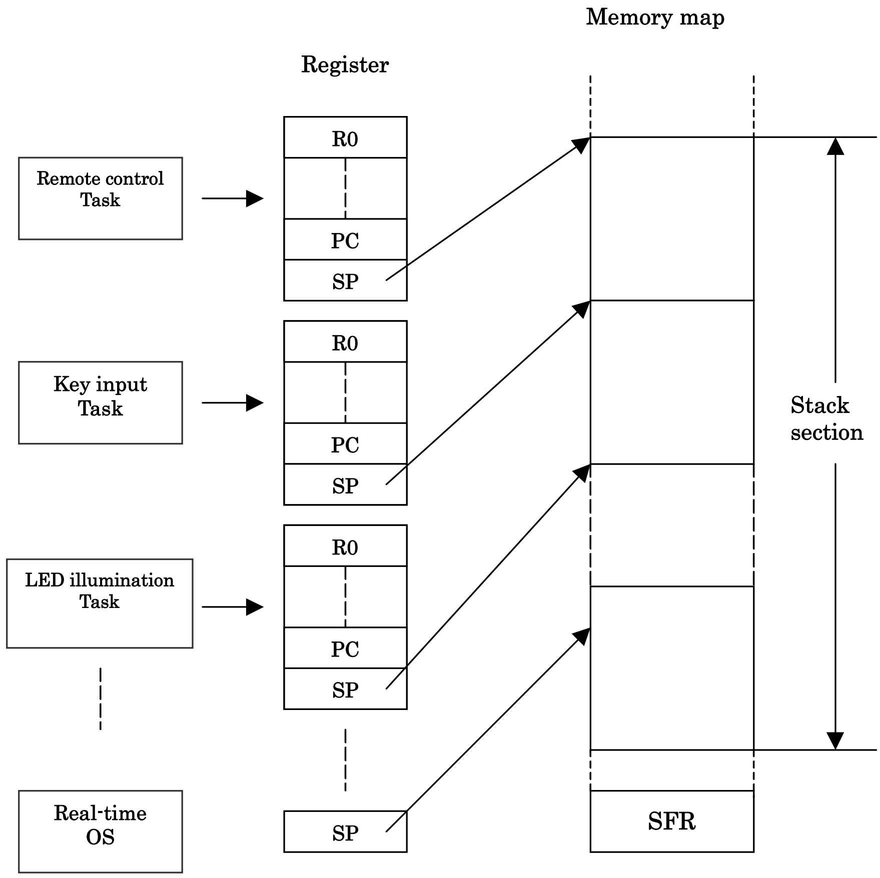

Figure 3.7 Task Register Area

Figure 3.8 shows the register and stack area of one task in detail. In the MR100, the register of each task is stored in a stack area as shown in Figure 3.8. This figure shows the state prevailing after register storage.

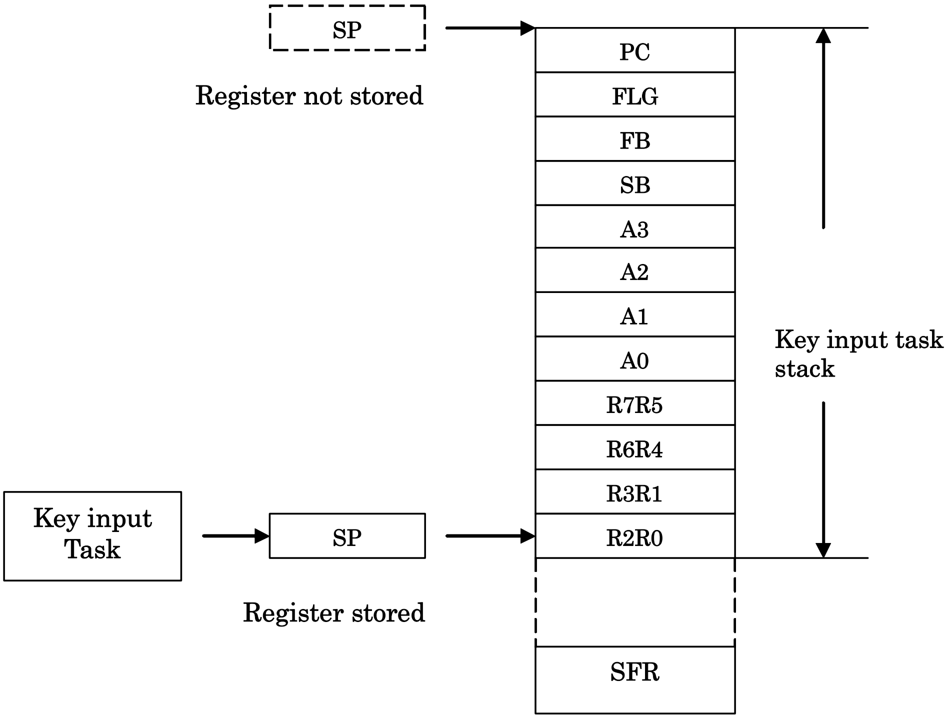

Figure 3.8 Actual Register and Stack Area Management

3.2 Service Call

How does the programmer use the kernel functions in a program?

First, it is necessary to call up kernel function from the program in some way or other. Calling a kernel function is referred to as a service call. Task activation and other processing operations can be initiated by such a service call (See Figure 3.9).

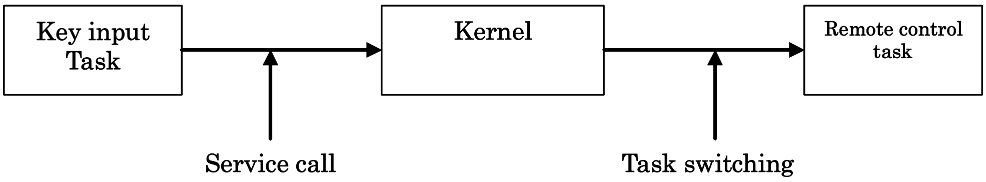

Figure 3.9 Service call

This service call is realized by a function call when the application program is written in C language, as shown below.

```
act_tsk(ID_main,3);
```

Furthermore, if the application program is written in assembly language, it is realized by an assembler macro call, as shown below.

```
act_tsk #ID_main
```

3.2.1 Service Call Processing

When a service call is issued, processing takes place in the following sequence.[6]

1. **The current register contents are saved.**
2. **The stack pointer is changed from the task type to the real-time OS (system) type.**
3. **Processing is performed in compliance with the request made by the service call.**
4. **The task to be executed next is selected.**
5. **The stack pointer is changed to the task type.**
6. **The register contents are recovered to resume task execution.**

The flowchart in Figure 3.10 shows the process between service call generation and task switching.

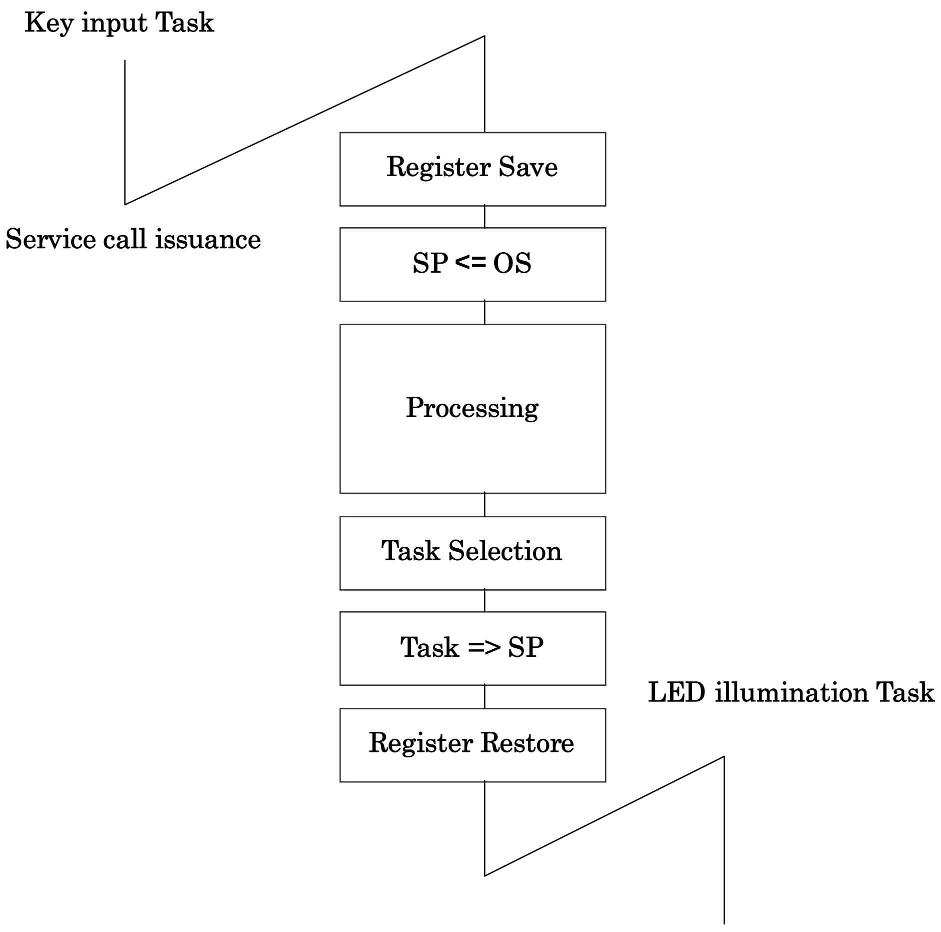

Figure 3.10 Service Call Processing Flowchart

[6] A different sequence is followed if the issued service call does not evoke task switching.

3.2.2 Processing Procedures for Service Calls from Handlers

When a service call is issued from a handler, task switching does not occur unlike in the case of a service call from a task. However, task switching occurs when a return from a handler [7] is made.

The processing procedures for service calls from handlers are roughly classified into the following three types.

1. **A service call from a handler that caused an interrupt during task execution**

2. **A service call from a handler that caused an interrupt during service call processing**

3. **A service call from a handler that caused an interrupt (multiplex interrupt) during handler execution**

[7] The service call can't be issued from non-kernel handler. Therefore, The handler described here does not include the non-kernel interrupt handler.

Service Calls from a Handler That Caused an Interrupt during Task Execution
Scheduling (task switching) is initiated by the ret_int service call [8](See Figure 3.11).

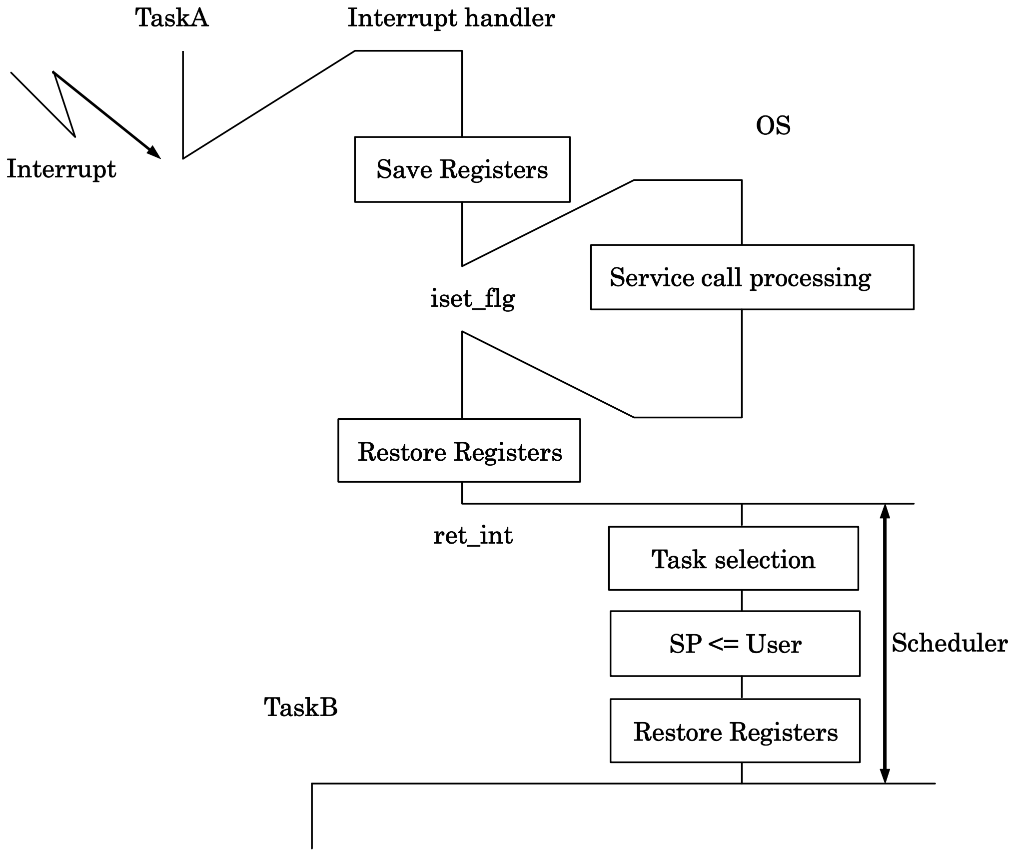

Figure 3.11 Processing Procedure for a Service Call a Handler that caused an interrupt during Task Execution

[8] The ret_int service call is issued automatically when kernel interrupt handler is written in C language (when #pragma INTHANDLER specified)

Service Calls from a Handler That Caused an Interrupt during Service Call Processing

Scheduling (task switching) is initiated after the system returns to the interrupted service call processing (See Figure 3.12).

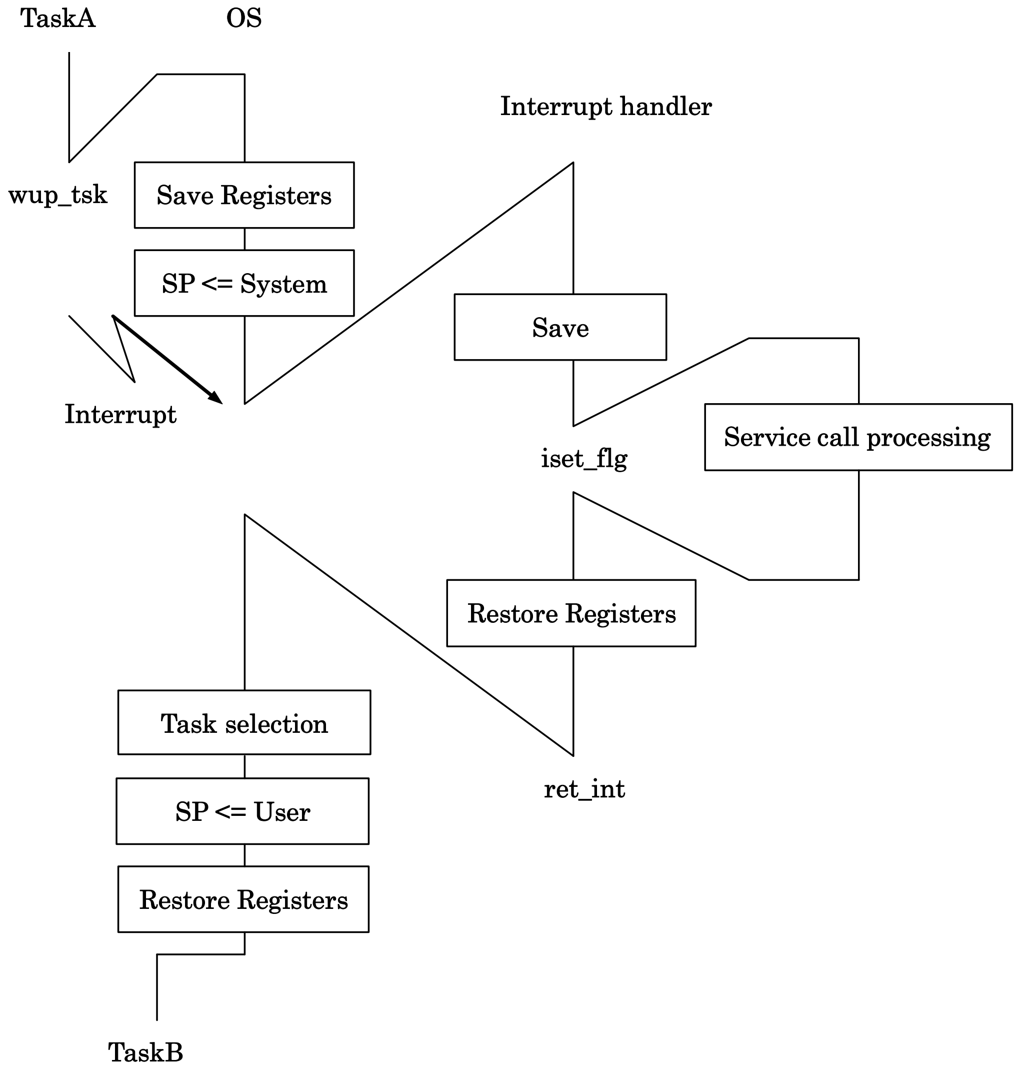

Figure 3.12 Processing Procedure for a Service Call from a Handler that caused an interrupt during Service Call Processing

Service Calls from a Handler That Caused an Interrupt during Handler Execution

Let us think of a situation in which an interrupt occurs during handler execution (this handler is hereinafter referred to as handler A for explanation purposes). When task switching is called for as a handler (hereinafter referred to as handler B) that caused an interrupt during handler A execution issued a service call, task switching does not take place during the execution of the service call (ret_int service call) returned from handler B, but is effected by the ret_int service call from handler A (See Figure 3.13).

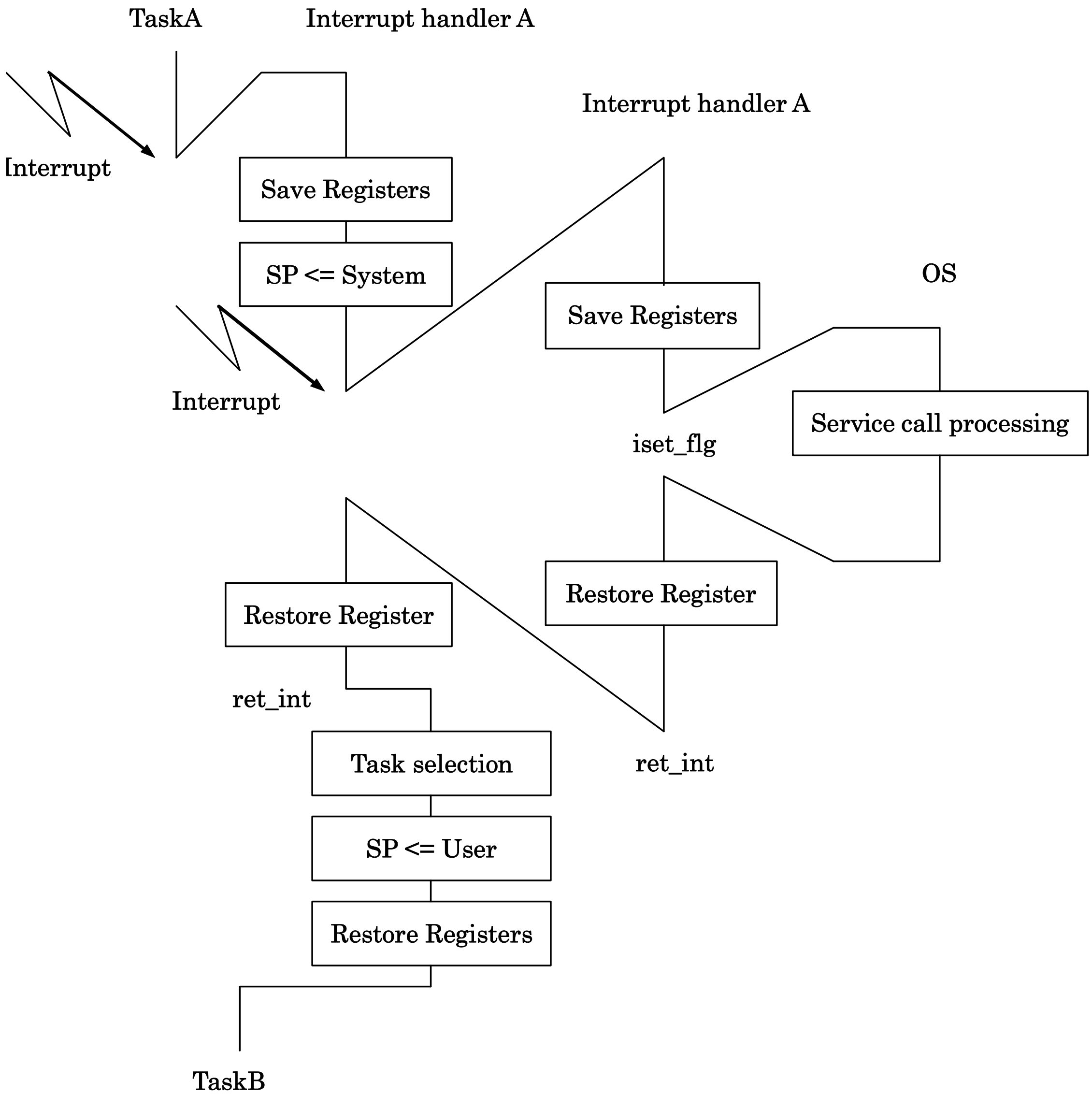

Figure 3.13 Processing Procedure for a service call from a Multiplex interrupt Handler

3.3 Object

The object operated by the service call of a semaphore, a task, etc. is called an "object." An object is identified by the ID number

3.3.1 The specification method of the object in a service call

Each task is identified by the ID number internally in MR100.

For example, the system says, "Start the task having the task ID number 1."

However, if a task number is directly written in a program, the resultant program would be very low in readability. If, for instance, the following is entered in a program, the programmer is constantly required to know what the No. 2 task is.

```
act_tsk(2);
```

Further, if this program is viewed by another person, he/she does not understand at a glance what the No. 2 task is. To avoid such inconvenience, the MR100 provides means of specifying the task by name (function or symbol name).

The program named "configurator cfg100 ,"which is supplied with the MR100, then automatically converts the task name to the task ID number. This task identification system is schematized in Figure 3.14.

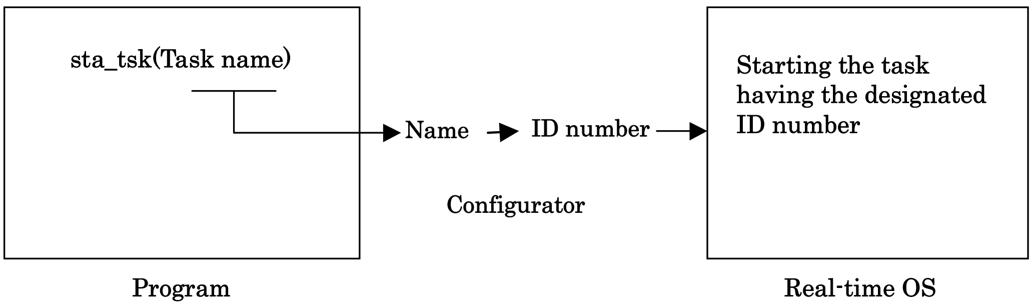

Figure 3.14 Task Identification

```
act_tsk(ID_task);
```

This example specifies that a task corresponding to "ID_task" be invoked.

It should also be noted that task name-to-ID number conversion is effected at the time of program generation. Therefore, the processing speed does not decrease due to this conversion feature.

3.4 Task

This section describes how tasks are managed by MR100.

3.4.1 Task Status

The real-time OS monitors the task status to determine whether or not to execute the tasks.

Figure 3.15 shows the relationship between key input task execution control and task status. When there is a key input, the key input task must be executed. That is, the key input task is placed in the execution (RUNNING) state. While the system waits for key input, task execution is not needed. In that situation, the key input task in the WAITING state.

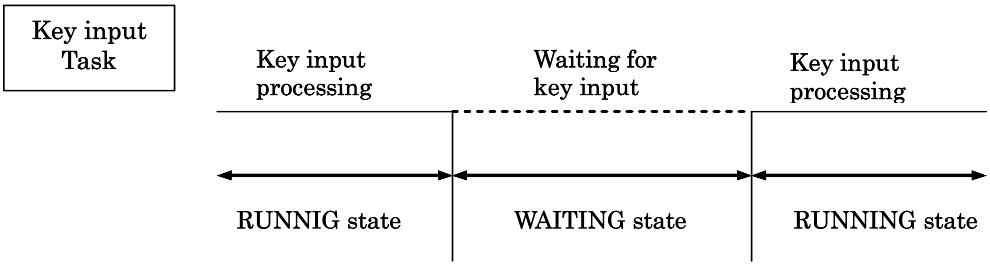

Figure 3.15 Task Status

The MR100 controls the following six different states including the RUNNING and WAITING states.

1. **RUNNING state**
2. **READY state**
3. **WAITING state**
4. **SUSPENDED state**
5. **WAITING-SUSPENDED state**
6. **DORMANT state**

Every task is in one of the above six different states. Figure 3.16 shows task status transition.

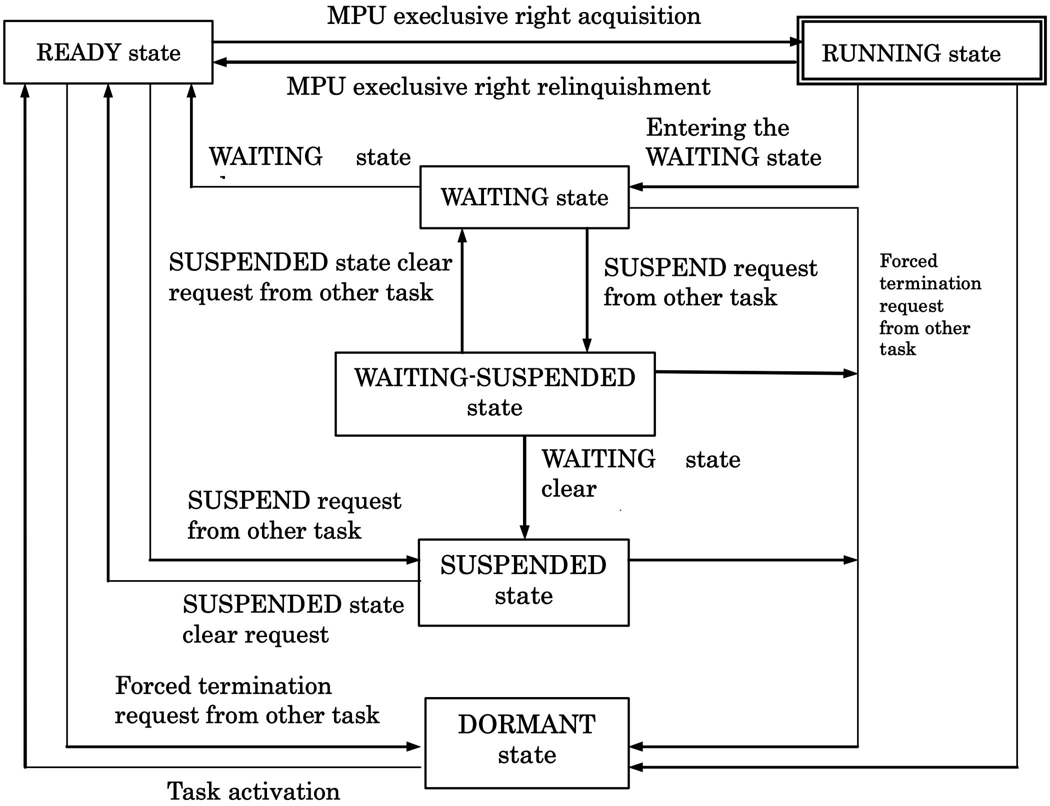

Figure 3.16 MR100 Task Status Transition

1. **RUNNING state**

In this state, the task is being executed. Since only one microcomputer is used, it is natural that only one task is being executed.
The currently executed task changes into a different state when any of the following conditions occurs.

- ♦ The task has normally terminated itself by ext_tsk service call.
- ♦ The task has placed itself in the WAITING. [9]
- ♦ Since the service call was issued from the RUNNING state task, the WAITING state of another task with a priority higher than the RUNNING state task is cleared.
- ♦ Due to interruption or other event occurrence, the interrupt handler has placed a different task having a higher priority in the READY state.
- ♦ The priority assigned to the task has been changed by chg_pri or ichg_pri service call so that the priority of another READY task is rendered higher.
- ♦ When the ready queue of the issuing task priority is rotated by the rot_rdq or irot_rdq service call and control of execution is thereby abandoned

When any of the above conditions occurs, rescheduling takes place so that the task having the highest priority among those in the RUNNING or READY state is placed in the RUNNING state, and the execution of that task starts.

2. **READY state**

The READY state refers to the situation in which the task that meets the task execution conditions is still waiting for execution because a different task having a higher priority is currently being executed.
When any of the following conditions occurs, the READY task that can be executed second according to the ready queue is placed in the RUNNING state.

- ♦ A currently executed task has normally terminated itself by ext_tsk service call.

[9] By issuing dly_tsk, slp_tsk, tslp_tsk, wai_flg, twai_flg, wai_sem, twai_sem, rcv_mbx, trcv_mbx,snd_dtq,tsnd_dtq,rcv_dtq, trcv_dtq, vtsnd_dtq, vsnd_dtq,vtrcv_dtq,vrcv_dtq, get_mpf, and tget_mpf service call.

- A currently executed task has placed itself in the WAITING state.[10]
- A currently executed task has changed its own priority by chg_pri or ichg_pri service call so that the priority of a different READY task is rendered higher.
- Due to interruption or other event occurrence, the priority of a currently executed task has been changed so that the priority of a different READY task is rendered higher.
- When the ready queue of the issuing task priority is rotated by the rot_rdq or irot_rdq service call and control of execution is thereby abandoned

3. WAITING state

When a task in the RUNNING state requests to be placed in the WAITING state, it exits the RUNNING state and enters the WAITING state. The WAITING state is usually used as the condition in which the completion of I/O device I/O operation or the processing of some other task is awaited.
The task goes into the WAITING state in one of the following ways.

- The task enters the WAITING state simply when the slp_tsk service call is issued. In this case, the task does not go into the READY state until its WAITING state is cleared explicitly by some other task.
- The task enters and remains in the WAITING state for a specified time period when the dly_tsk service call is issued. In this case, the task goes into the READY state when the specified time has elapsed or its WAITING state is cleared explicitly by some other task.
- The task is placed into WAITING state for a wait request by the wai_flg, wai_sem, rcv_mbx, snd_dtq, rcv_dtq, vsnd_dtq, vrcv_dtq, or get_mpf service call. In this case, the task goes from WAITING state to READY state when the request is met or WAITING state is explicitly canceled by another task.
- The tslp_tsk, twai_flg, twai_sem, trcv_mbx, tsnd_dtq, trcv_dtq, vtsnd_dtq, vtrcv_dtq and tget_mpf service calls are the timeout-specified versions of the slp_tsk, wai_flg, wai_sem, rcv_mbx, snd_dtq, rcv_dtq, vsnd_dtq, vrcv_dtq and get_mpf service calls. The task is placed into WAITING state for a wait request by one of these service calls. In this case, the task goes from WAITING state to READY state when the request is met or the specified time has elapsed.
- If the task is placed into WAITING state for a wait request by the wai_flg, wai_sem, rcv_mbx, snd_dtq, rcv_dtq, vsnd_dtq, vrcv_dtq, get_mpf, twai_flg, twai_sem, trcv_mbx, tsnd_dtq, trcv_dtq, vtsnd_dtq, vtrcv_dtq and tget_mpf service call, the task is queued to one of the following waiting queues depending on the request.

    - Event flag waiting queue
    - Semaphore waiting queue
    - Mailbox message reception waiting queue
    - Data queue data transmission waiting queue
    - Data queue data reception waiting queue
    - Short data queue data transmission waiting queue
    - Short data queue data reception waiting queue
    - Fixed-size memory pool acquisition waiting queue

4. SUSPENDED state

When the sus_tsk service call is issued from a task in the RUNNING state or the isus_tsk service call is issued from a handler, the READY task designated by the service call or the currently executed task enters the SUSPENDED state. If a task in the WAITING state is placed in this situation, it goes into the WAITING-SUSPENDED state.

The SUSPENDED state is the condition in which a READY task or currently executed task[11] is excluded from scheduling to halt processing due to I/O or other error occurrence. That is, when the suspend request is made to a READY task, that task is excluded from the execution queue.

Note that no queue is formed for the suspend request. Therefore, the suspend request can only be made to the

[10] Depends on the dly_tsk, slp_tsk, tslp_tsk, wai_flg, twai_flg, wai_sem, twai_sem, rcv_mbx, trcv_mbx,snd_dtq,tsnd_dtq,rcv_dtq, trcv_dtq, vtsnd_dtq, vsnd_dtq,vtrcv_dtq,tget_mpf, get_mpf and vrcv_dtq service call.
[11] If the task under execution is placed into a forcible wait state by the isus_tsk service call from the handler, the task goes from an executing state directly to a forcible wait state. Please note that in only this case exceptionally, it is possible that a task will go from an executing state directly to a forcible wait state.

tasks in the RUNNING, READY, or WAITING state.[12] If the suspend request is made to a task in the SUSPENDED state, an error code is returned.

5. WAITING-SUSPENDED

If a suspend request is issued to a task currently in a WAITING state, the task goes to a WAITING-SUSPENDED state. If a suspend request is issued to a task that has been placed into a WAITING state for a wait request by the slp_tsk, wai_flg, wai_sem, rcv_mbx, snd_dtq, rcv_dtq, vsnd_dtq, vrcv_dtq, get_mpf, tslp_tsk, twai_flg, twai_sem, trcv_mbx, tsnd_dtq, trcv_dtq, vtsnd_dtq, vtrcv_dtq or tget_mpf service call, the task goes to a WAITING-SUSPENDED state.

When the wait condition for a task in the WAITING-SUSPENDED state is cleared, that task goes into the SUSPENDED state. It is conceivable that the wait condition may be cleared, when any of the following conditions occurs.

- The task wakes up upon wup_tsk, or iwup_tsk service call issuance.
- The task placed in the WAITING state by the dly_tsk or tslp_tsk service call wakes up after the specified time elapse.
- The request of the task placed in the WAITING state by the wai_flg , wai_sem, rcv_mbx, snd_dtq, rcv_dtq, vsnd_dtq, vrcv_dtq, get_mpf, tslp_tsk, twai_flg, twai_sem, trcv_mbx, tsnd_dtq, trcv_dtq, vtsnd_dtq, vtrcv_dtq or tget_mpf service call is fulfilled.
- The WAITING state is forcibly cleared by the rel_wai or irel_wai service call

When the SUSPENDED state clear request by rsm_tsk or irsm_tsk is made to a task in the WAITING-SUSPENDED state, that task goes into the WAITING state. Since a task in the SUSPENDED state cannot request to be placed in the WAITING state, status change from SUSPENDED to WAITING-SUSPENDED does not possibly occur.

6. DORMANT

This state refers to the condition in which a task is registered in the MR100 system but not activated. This task state prevails when either of the following two conditions occurs.

- The task is waiting to be activated.
- The task is normally terminated by ext_tsk service call or forcibly terminated by ter_tsk service call.

[12] If a forcible wait request is issued to a task currently in a wait state, the task goes to a WAITING-SUSPENDED state.

3.4.2 Task Priority and Ready Queue

In the kernel, several tasks may simultaneously request to be executed. In such a case, it is necessary to determine which task the system should execute first. To properly handle this kind of situation, the system organizes the tasks into proper execution priority and starts execution with a task having the highest priority. To complete task execution quickly, tasks related to processing operations that need to be performed immediately should be given higher priorities.

The MR100 permits giving the same priority to several tasks. To provide proper control over the READY task execution order, the kernel generates a task execution queue called "ready queue." The ready queue structure is shown in Figure 3.17[13] The ready queue is provided and controlled for each priority level. The first task in the ready queue having the highest priority is placed in the RUNNING state.[14]

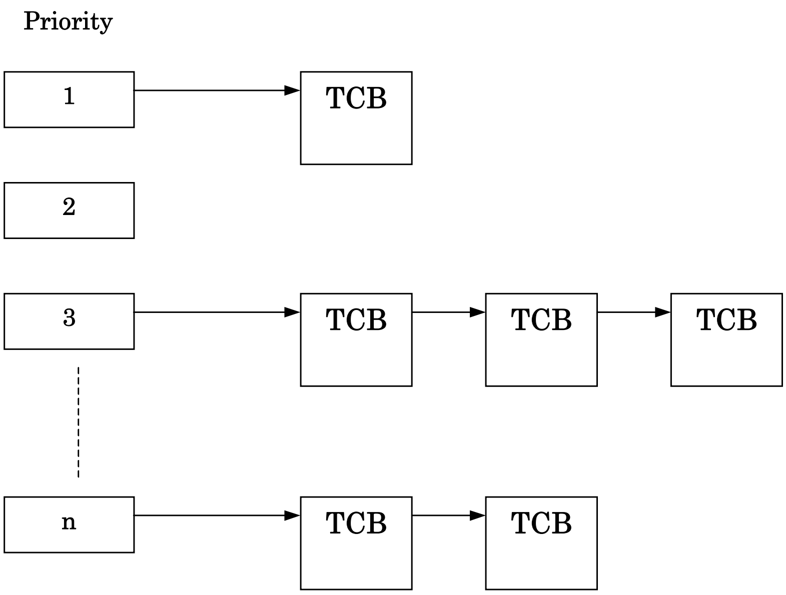

Figure 3.17 Ready Queue (Execution Queue)

[13] The TCB(task control block is described in the next chapter.)
[14] The task in the RUNNING state remains in the ready queue.

3.4.3 Task Priority and Waiting Queue

In The standard profiles in µITRON 4.0 Specification support two waiting methods for each object. In one method, tasks are placed in a waiting queue in order of priority (TA_TPRI attribute); in another, tasks are placed in a waiting queue in order of FIFO (TA_TFIFO).

Figure 3.18 and Figure 3.19 depict the manner in which tasks are placed in a waiting queue in order of "taskD," "taskC," "taskA," and "taskB."

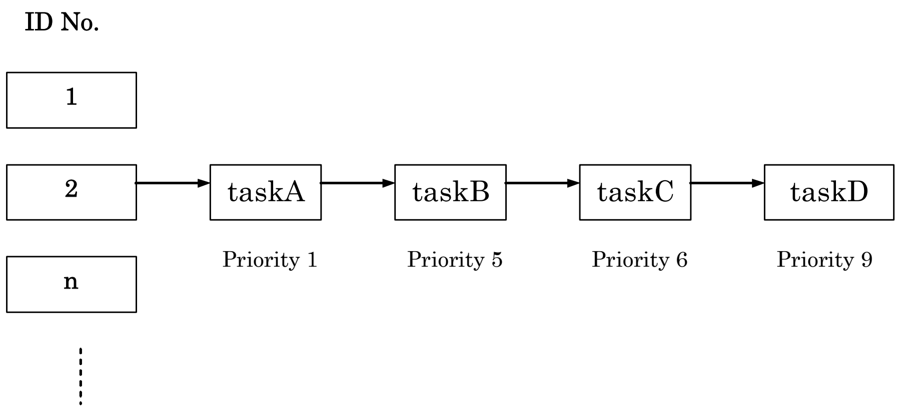

Figure 3.18 Waiting queue of the TA_TPRI attribute

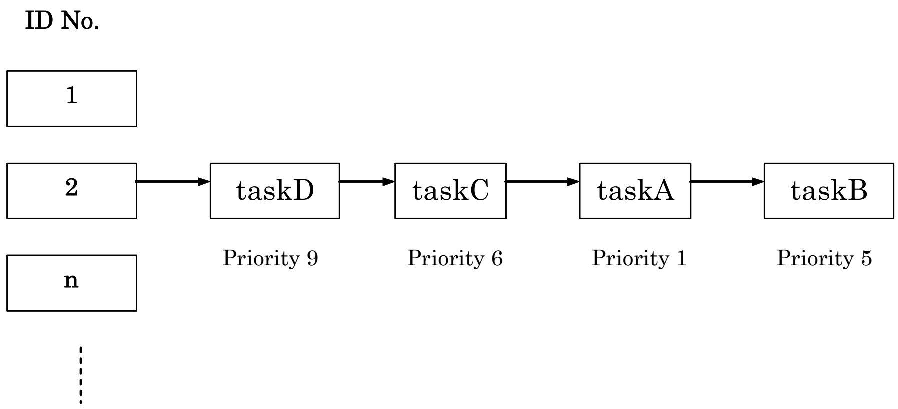

Figure 3.19 Waiting queue of the TA_TFIFO attribute

3.4.4 Task Control Block(TCB)

The task control block (TCB) refers to the data block that the real-time OS uses for individual task status, priority, and other control purposes.

The MR100 manages the following task information as the task control block

- Task connection pointer
  Task connection pointer used for ready queue formation or other purposes.
- Task status
- Task priority
- Task register information and other data[15] storage stack area pointer(current SP value)
- Wake-up counter
  Task wake-up request storage area.
- Flag wait mode
  This is a wait mode during eventflag wait.
- Flag wait pattern
  This area stores the flag wait pattern when using the eventflag wait service call (wai_flg, twai_flg). No flag wait pattern area is allocated when the eventflag is not used.
- Startup request counter
  This is the area in which task startup requests are accumulated.

The task control block is schematized in Figure 3.20.

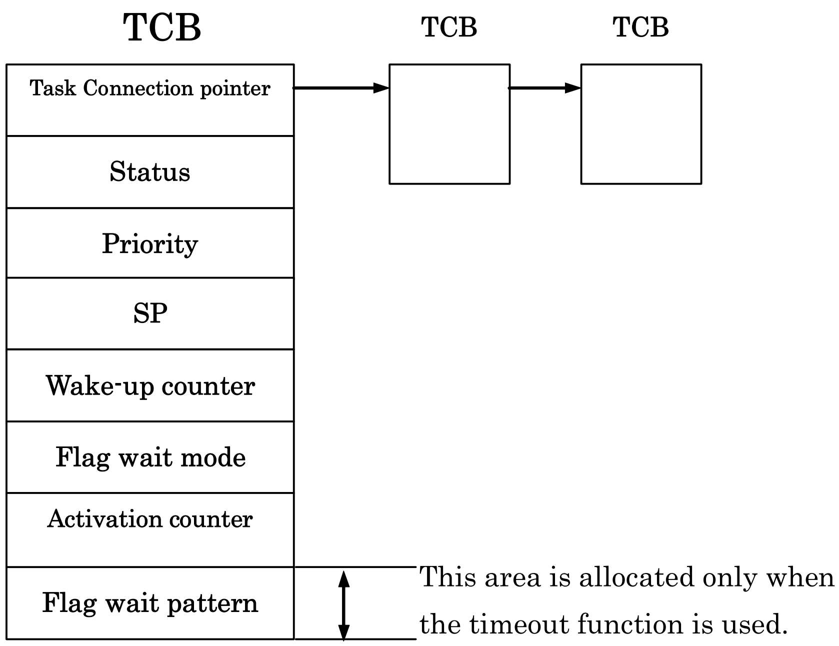

Figure 3.20 Task control block

[15] Called the task context

3.5 System States

3.5.1 Task Context and Non-task Context

The system runs in either context state, "task context" or "non-task context." The differences between the task content and non-task context are shown in Table 3-1. Task Context and Non-task Context.

Table 3.1 Task Context and Non-task Context

	Task context	Non-task context
Invocable service call	Those that can be invoked from task context	Those that can be invoked from non-task context
Task scheduling	Occurs when the queue state has changed to other than dispatch disabled and CPU locked states.	It does not occur.
Stack	User stack	System stack

The processes executed in non-task context include the following.

1. Interrupt Handler

A program that starts upon hardware interruption is called the interrupt handler. The MR100 is not concerned in interrupt handler activation. Therefore, the interrupt handler entry address is to be directly written into the interrupt vector table.

There are two interrupt handlers: Non-kernel interrupts (OS independent interrupts) and kernel interrupts (OS dependent interrupts). For details about each type of interrupt, refer to Section 3.6.

The system clock interrupt handler (isig_tim) is one of these interrupt handlers.

2. Cyclic Handler

The cyclic handler is a program that is started cyclically every preset time. The set cyclic handler may be started or stopped by the sta_cyc(ista_cyc) or stp_cyc(istp_cyc) service call.

The cyclic handler startup time of day is unaffected by a change in the time of day by set_tim(iset_tim).

3. Alarm Handler

The alarm handler is a handler that is started after the lapse of a specified relative time of day. The alarm handler startup time of day is determined by a time of day relative to the time of day set by sta_alm(ista_alm), and is unaffected by a change in the time of day by set_tim(iset_tim).

The cyclic and alarm handlers are invoked by a subroutine call from the system clock interrupt (timer interrupt) handler. Therefore, cyclic and alarm handlers operate as part of the system clock interrupt handler. Note that when the cyclic or alarm handler is invoked, it is executed in the interrupt priority level of the system clock interrupt.

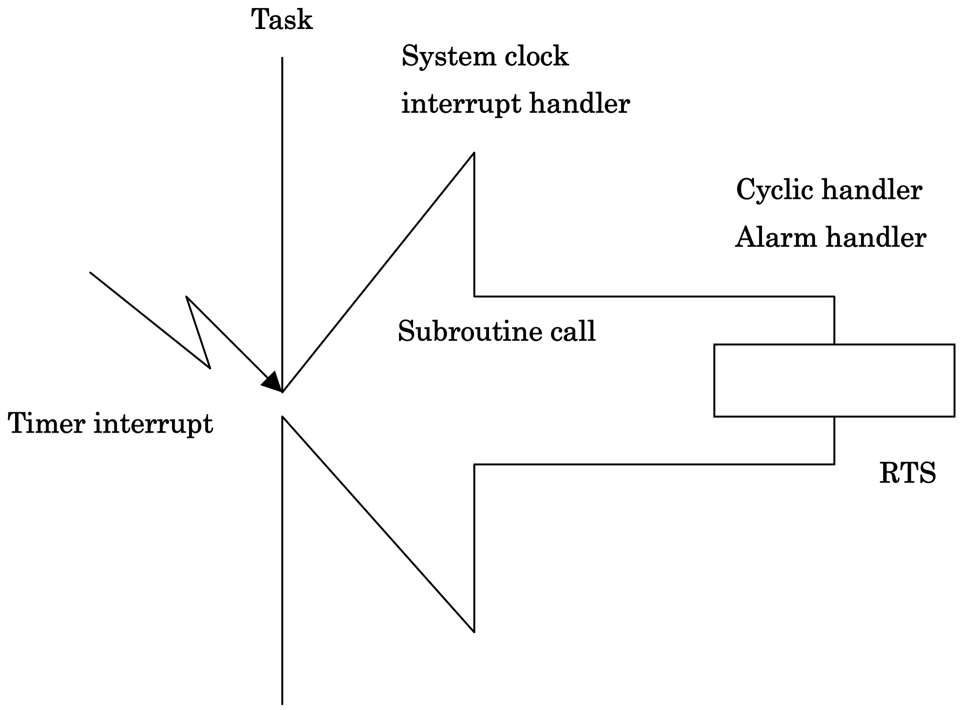

Figure 3.21 Cyclic Handler/Alarm Handler Activation

3.5.2 Dispatch Enabled/Disabled States

The system assumes either a dispatch enabled state or a dispatch disabled state. In a dispatch disabled state, no task scheduling is performed. Nor can service calls be invoked that may cause the service call issuing task to enter a wait state.[16]

The system can be placed into a dispatch disabled state or a dispatch enabled state by the dis_dsp or ena_dsp service call, respectively. Whether the system is in a dispatch disabled state can be known by the sns_dsp service call.

3.5.3 CPU Locked/Unlocked States

The system assumes either a CPU locked state or a CPU unlocked state. In a CPU locked state, all external interrupts are disabled against acceptance, and task scheduling is not performed either.

The system can be placed into a CPU locked state or a CPU unlocked state by the loc_cpu(iloc_cpu) or unl_cpu(iunl_cpu) service call, respectively. Whether the system is in a CPU locked state can be known by the sns_loc service call.

The service calls that can be issued from a CPU locked state are limited to those that are listed in Table 3-2.[17]

Table 3.2 Invocable Service Calls in a CPU Locked State

loc_cpu	iloc_cpu	unl_cpu	iunl_cpu
ext_tsk	exd_tsk	sns_tex	sns_ctx
sns_loc	sns_dsp	sns_dpn	

3.5.4 Dispatch Disabled and CPU Locked States

In µITRON 4.0 Specification, the dispatch disabled and the CPU locked states are clearly discriminated. Therefore, if the unl_cpu service call is issued in a dispatch disabled state, the dispatch disabled state remains intact and no task scheduling is performed. State transitions are summarized in Table 3.3.

Table 3.3 CPU Locked and Dispatch Disabled State Transitions Relating to dis_dsp and loc_cpu

State number	Content of state		dis_dsp executed	ena_dsp executed	loc_cpu executed	unl_cpu executed
	CPU locked state	Dispatch disabled state				
1	O	X	X	X	=> 1	=> 3
2	O	O	X	X	=> 2	=> 4
3	X	X	=> 4	=> 3	=> 1	=> 3
4	X	O	=> 4	=> 3	=> 2	=> 4

16 If a service call not issuable is issued when dispatch disabled, MR100 doesn't return the error and doesn't guarantee the operation.
17 MR100 does not return an error even when an uninvocable service call is issued from a CPU locked state, in which case, however, its operation cannot be guaranteed.

3.6 Regarding Interrupts

3.6.1 Types of Interrupt Handlers

MR100's interrupt handlers consist of kernel interrupt handlers and non-kernel interrupt handlers.

The following shows the definition of each type of interrupt handler.

- Kernel interrupt handler
  An interrupt handler whose interrupt priority level is lower than a kernel interruption mask level is called kernel interrupt handler. That is, interruption priority level is from 1 to system_IPL.
  A service call can be issued within a kernel interrupt handler. However, interrupt is delayed until it becomes receivable the kernel interrupt handler generated during service call processing.

- Non-kernel interrupt handler
  An interrupt handler whose interrupt priority level is higher than a kernel interrupt mask level is called non-kernel interrupt handler. That is, interruption priority level is from system_IPL+1 to 7.
  A service call cannot be issued within non-kernel interrupt handler. However, the non-kernel interrupt handler is able to be recieved during service call processing, even if it is the section where it is not able to receive a kernel interrupt handler:

Figure 3.22 shows the relationship between the non-kernel interrupt handlers and kernel interrupt handlers where the kernel mask level is set to 3.

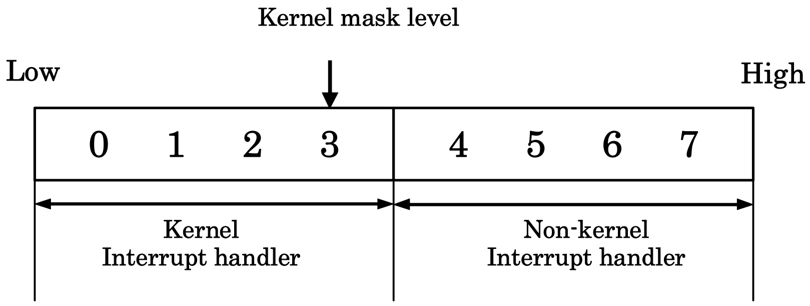

Figure 3.22 Interrupt handler IPLs

3.6.2 The Use of Non-maskable Interrupt

Non-maskable interrupt (ex. NMI interrupt ,Watchdog Timer interrupt) are treated as a non-kernel interrupt handler.

3.6.3 Controlling Interrupts

Interrupt enable/disable control in a service call is accomplished by IPL manipulation. The IPL value in a service call is set to the kernel mask level(OS interrupt disable level = system.IPL) in order to disable interrupts for the kernel interrupt handler. In sections where all interrupts can be enabled, it is returned to the initial IPL value when the service call was invoked.

- For service calls that can be issued from only task context.

When the I flag before issuing a service call is 1.

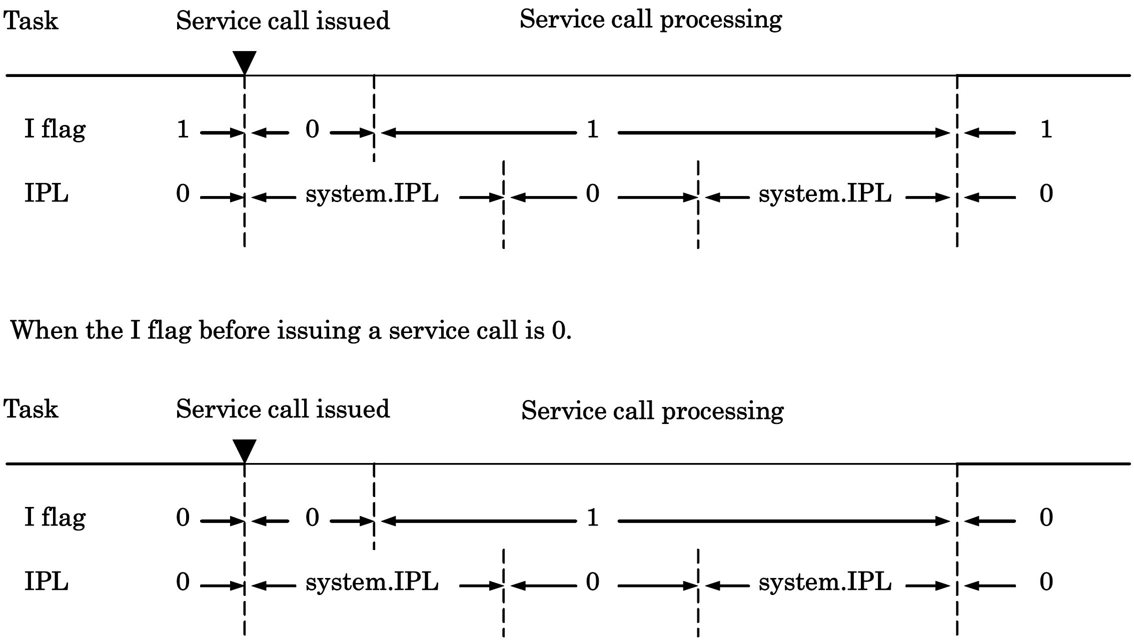

Figure 3.23 Interrupt control in a Service Call that can be Issued from only a Task

- For service calls that can be issued from only non-task context or from both task context and non-task context.

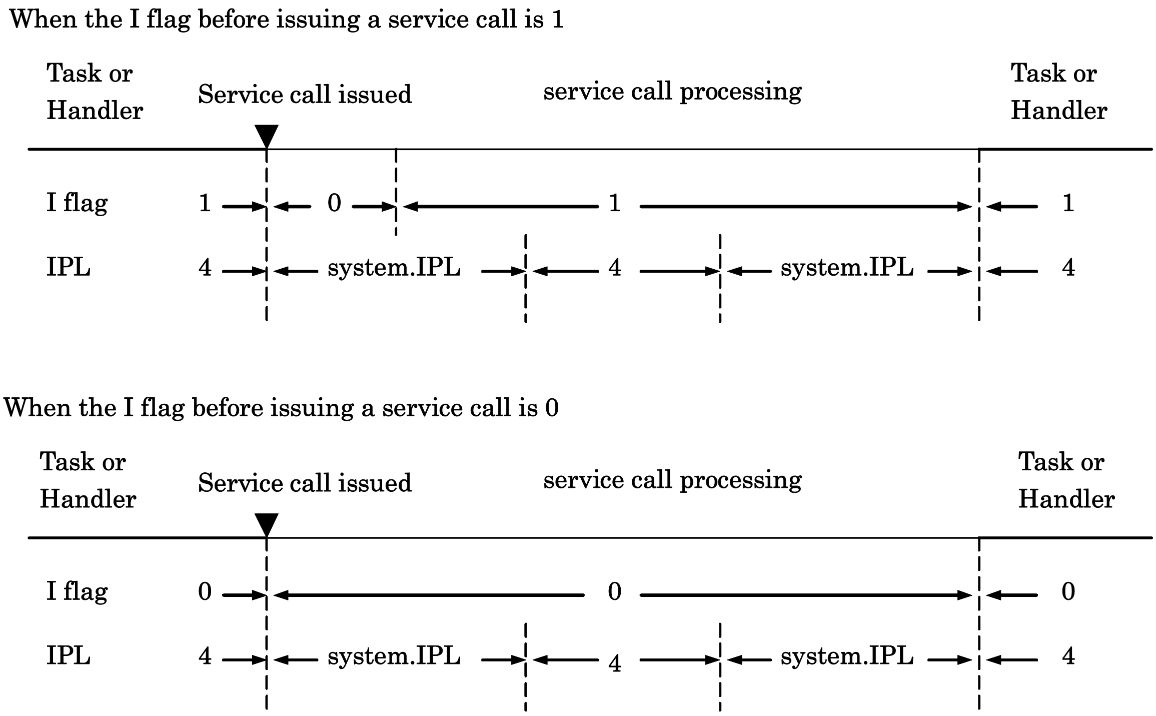

Figure 3.24 Interrupt control in a Service Call that can be Issued from a Task-independent

As shown in Figure 3.23 and Figure 3.24, the interrupt enable flag and IPL change in a service call. For this reason, if you want to disable interrupts in a user application, Renesas does not recommend using the method for manipulating the interrupt disable flag and IPL to disable the interrupts.

The following two methods for interrupt control are recommended:

1. **Modify the interrupt control register (SFR) for the interrupt you want to be disabled.**
2. **Use service calls loc_cpu(iloc_cpu) and unl_cpu(iunl_cpu).**

The interrupts that can be controlled by the loc_cpu service call are only the kernel interrupt. Use method 1 to control the non-kernel interrupts.

3.7 Stacks

3.7.1 System Stack and User Stack

The MR100 provides two types of stacks: system stack and user stack.

- User Stack
  One user stack is provided for each task. Therefore, when writing applications with the MR100, it is necessary to furnish the stack area for each task.

- System Stack
  This stack is used within the MR100 (during service call processing). When a service call is issued from a task, the MR100 switches the stack from the user stack to the system stack (See Figure 3.25).
  The system stack use the interrupt stack(ISP).

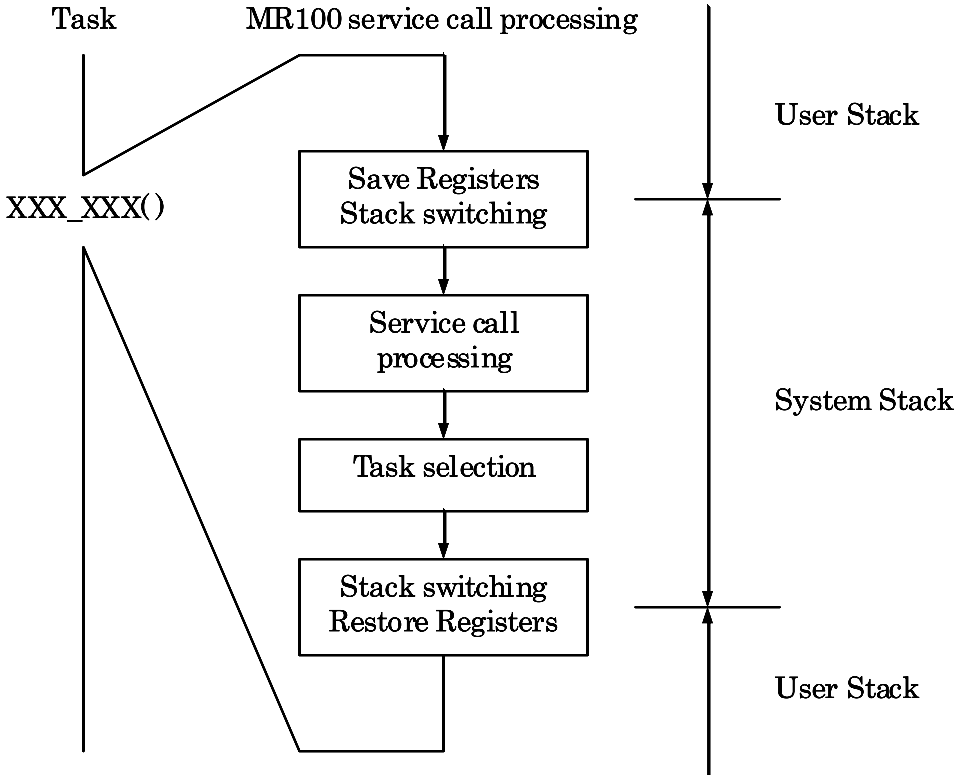

Figure 3.25 System Stack and User Stack

Switchover from user stack to system stack occurs when an interrupt of vector numbers 0 to 127 is generated. Consequently, all stacks used by the interrupt handler are the system stack.

4. Kernel

4.1.1 Module Structure

The MR100 kernel consists of the modules shown in Figure 4.1. Each of these modules is composed of functions that exercise individual module features.

The MR100 kernel is supplied in the form of a library, and only necessary features are linked at the time of system generation. More specifically, only the functions used are chosen from those which comprise these modules and linked by means of the Linkage Editor. However, the scheduler module, part of the task management module, and part of the time management module are linked at all times because they are essential feature functions.

The applications program is a program created by the user. It consists of tasks, interrupt handler, alarm handler, and cyclic handler.[18]

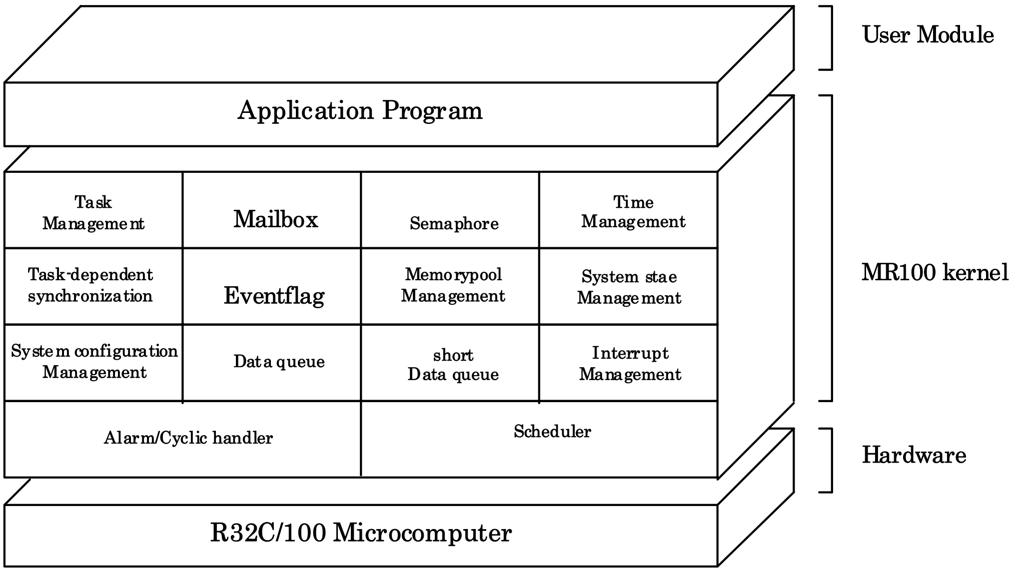

Figure 4.1 MR100 Structure

[18] For details, See 4.1.11.

4.1.2 Module Overview

The MR100 kernel modules are outlined below.

- **Scheduler**
  Forms a task processing queue based on task priority and controls operation so that the high-priority task at the beginning in that queue (task with small priority value) is executed.
- **Task Management Module**
  Exercises the management of various task states such as the RUNNING, READY, WAITING, and SUSPENDED state.
- **Task Synchronization Module**
  Accomplishes inter-task synchronization by changing the task status from a different task.
- **Interrupt Management Module**
  Makes a return from the interrupt handler.
- **Time Management Module**
  Sets up the system timer used by the MR100 kernel and starts the user-created alarm handler[19] and cyclic handler.[20].
- **System Status Management Module**
  Gets the system status of MR100.
- **System Configuration Management Module**
  Reports the MR100 kernel version number or other information.
- **Synchronization and Communication Module**
  This is the function for synchronization and communication among the tasks. The following four functional modules are offered.

  - Eventflag
    Checks whether the flag controlled within the MR100 is set up and then determines whether or not to initiate task execution. This results in accomplishing synchronization between tasks.
  - Semaphore
    Reads the semaphore counter value controlled within the MR100 and then determines whether or not to initiate task execution. This also results in accomplishing synchronization between tasks.
  - Mailbox
    Provides inter-task data communication by delivering the first data address.
  - Data queue
    Performs 32-bit data communication between tasks.

- **Memory pool Management Module**
  Provides dynamic allocation or release of a memory area used by a task or a handler.
- **Extended Function Module**
  Outside the scope of μITRON 4.0 Specification , this function performs reset processing on objects and short data queue function.

---

[19] This handler actuates once only at preselected times.
[20] This handler periodically actuates.

4.1.3 Task Management Function

The task management function is used to perform task operations such as task start/stop and task priority updating. The MR100 kernel offers the following task management function service calls.

- Activate Task (act_tsk, iact_tsk)
  Activates the task, changing its status from DORMANT to either READY or RUNNING. In this service call, unlike in sta_tsk(ista_tsk), startup requests are accumulated, but startup code cannot be specified.

- Activate Task (sta_tsk, ista_tsk)
  Activates the task, changing its status from DORMANT to either READY or RUNNING. In this service call, unlike in act_tsk(iact_tsk), startup requests are not accumulated, but startup code can be specified.

- Terminate Invoking Task (ext_tsk)
  When the issuing task is terminated, its state changes to DORMANT state. The task is therefore not executed until it is restarted. If startup requests are accumulated, task startup processing is performed again. In that case, the issuing task behaves as if it were reset.
  If written in C language, this service call is automatically invoked at return from the task regardless of whether it is explicitly written when terminated.

- Terminate Task (ter_tsk)
  Other tasks in other than DORMANT state are forcibly terminated and placed into DORMANT state. If startup requests are accumulated, task startup processing is performed again. In that case, the task behaves as if it was reset. (See Figure 4.2).

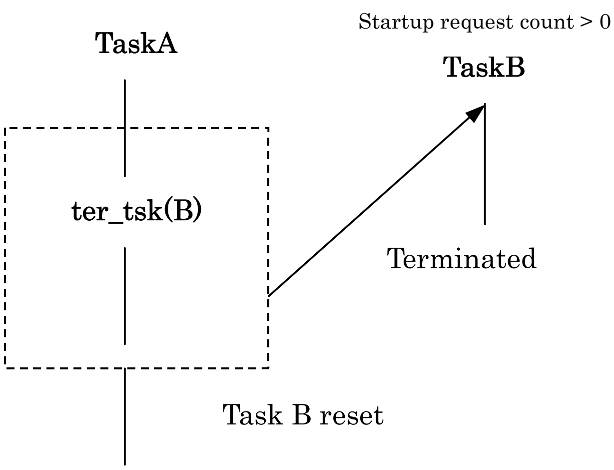

Figure 4.2 Task Resetting

- Change Task Priority (chg_pri, ichg_pri)
  If the priority of a task is changed while the task is in READY or RUNNING state, the ready queue also is updated. (See Figure 4.3).
  Furthermore, if the target task is placed in a waiting queue of objects with TA_TPRI attribute, the waiting queue also is updated. (See Figure 4.4).

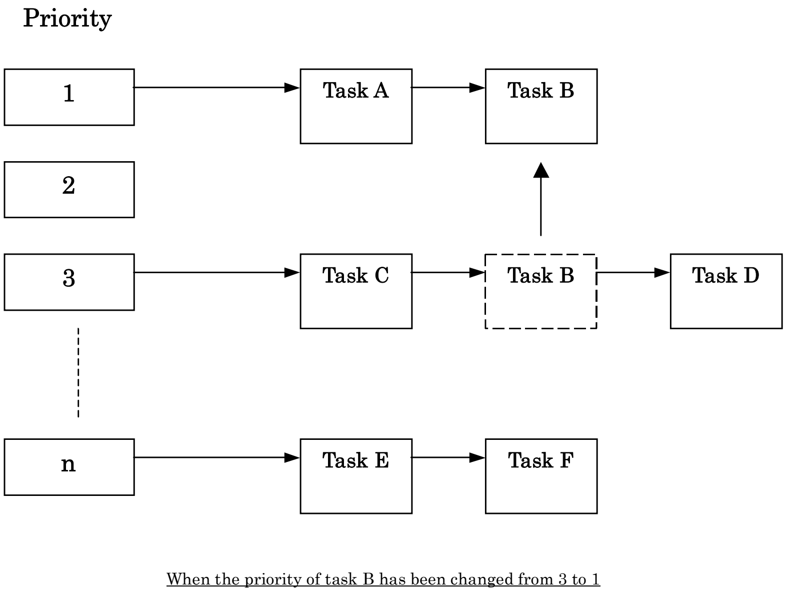

Figure 4.3 Alteration of task priority

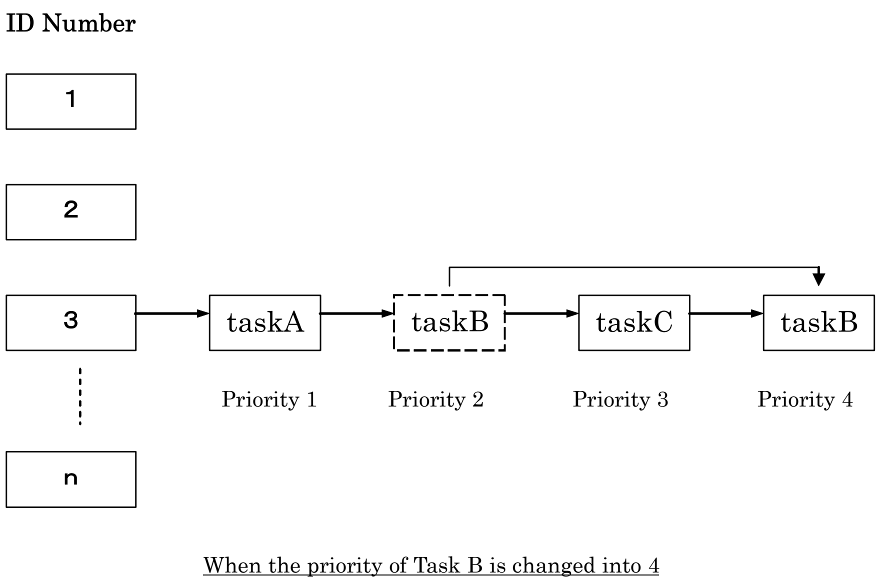

Figure 4.4 Task rearrangement in a waiting queue

- Reference task priority (get_pri, iget_pri)
  Gets the priority of a task.

- Reference task status (simple version) (ref_tst, iref_tst)
  Refers to the state of the target task.

- Reference task status (ref_tsk, iref_tsk)
  Refers to the state of the target task and its priority, etc.

4.1.4 Synchronization functions attached to task

The task-dependent synchronization functions attached to task is used to accomplish synchronization between tasks by placing a task in the WAIT, SUSPENDED, or WAIT-SUSPENDED state or waking up a WAIT state task.

The MR100 offers the following task incorporated synchronization service calls.

- Put Task to sleep (slp_tsk,tslp_tsk)
- Wakeup task (wup_tsk, iwup_tsk)
  Wakeups a task that has been placed in a WAIT state by the slp_tsk or tslp_tsk service call.
  No task can be waked up unless they have been placed in a WAITING state by.[21]
  If a wakeup request is issued to a task that has been kept waiting for conditions other than the slp_tsk or tslp_tsk service call or a task in other than DORMANT state by the wup_tsk or iwup_tsk service call, that wakeup request only will be accumulated.
  Therefore, if a wakeup request is issued to a task RUNNING state, for example, this wakeup request is temporarily stored in memory. Then, when the task in RUNNING state is going to be placed into WAITING state by the slp_tsk or tslp_tsk service call, the accumulated wakeup request becomes effective, so that the task continues executing again without going to WAITING state. (See Figure 4.5).
- Cancel Task Wakeup Requests (can_wup)
  Clears the stored wakeup request.(See Figure 4.6).

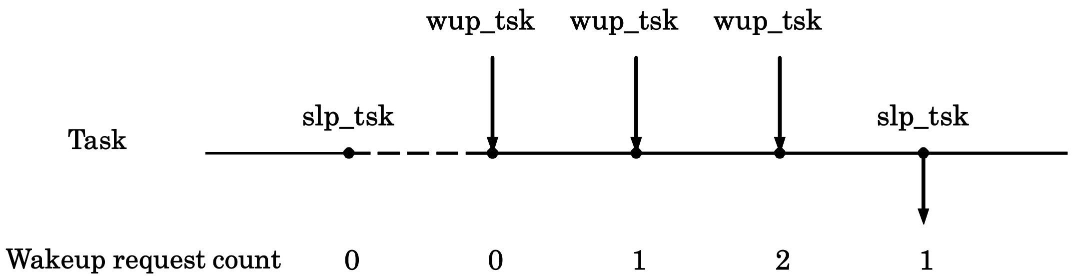

Figure 4.5 Wakeup Request Storage

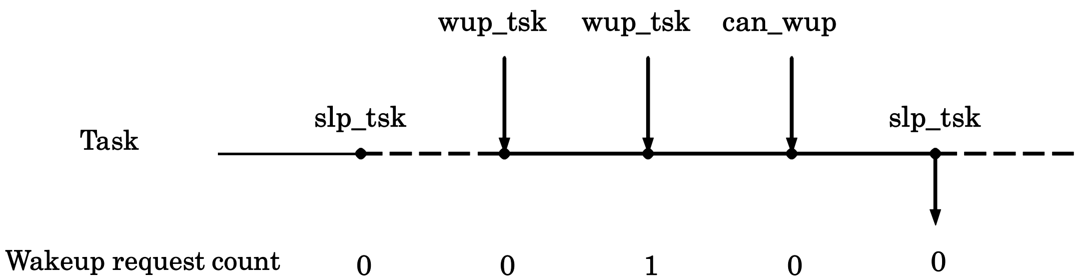

Figure 4.6 Wakeup Request Cancellation

[21] Note that tasks in WAITING state, but kept waiting for the following conditions are not awaken.
Eventflag wait state, semaphore wait state, data transmission wait state, data reception wait state, timeout wait state, fixed length memory pool acquisition wait, short data transmission wait, or short data reception wait

- Suspend task (sus_tsk, isus_tsk)
- Resume suspended task (rsm_tsk, irsm_tsk)

These service calls forcibly keep a task suspended for execution or resume execution of a task. If a suspend request is issued to a task in READY state, the task is placed into SUSPENDED state; if issued to a task in WAIT ING state, the task is placed into WAITING-SUSPENDED state. Since MR100 allows only one forcible wait request to be nested, if sus_tsk is issued to a task in a forcible wait state, the error E_QOVR is returned. (See Figure 4.7).

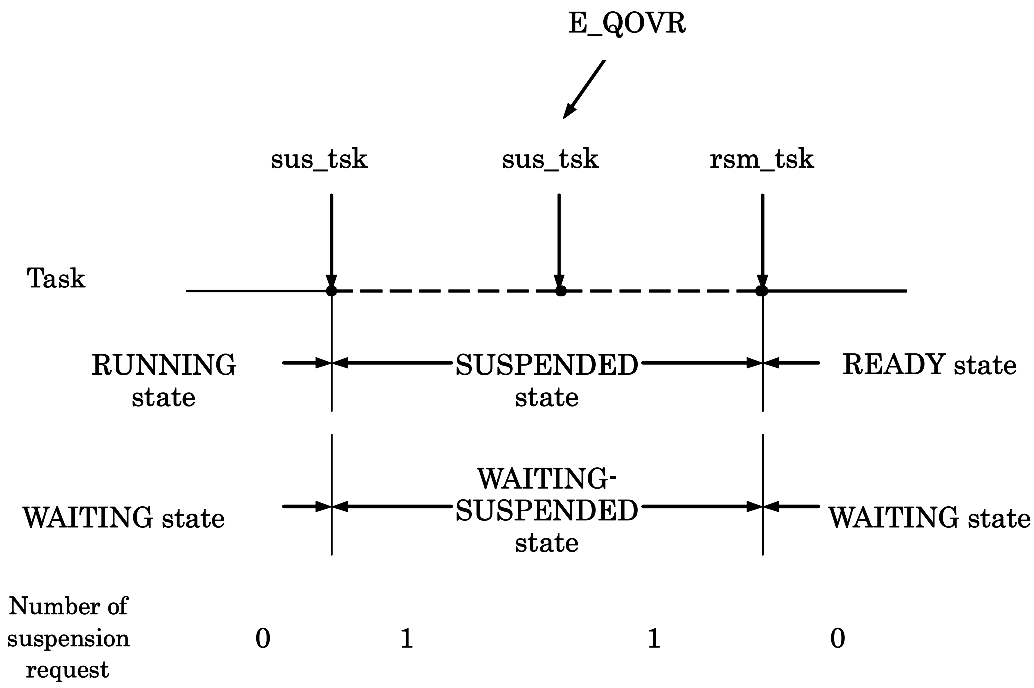

Figure 4.7 Forcible wait of a task and resume

- Forcibly resume suspended task (frsm_tsk, ifrsm_tsk)
  Clears the number of suspension requests nested to 0 and forcibly resumes execution of a task. Since MR100 allows only one suspension request to be nested, this service call behaves the same way as rsm_tsk and irsm_tsk..(See Figure 4.8).

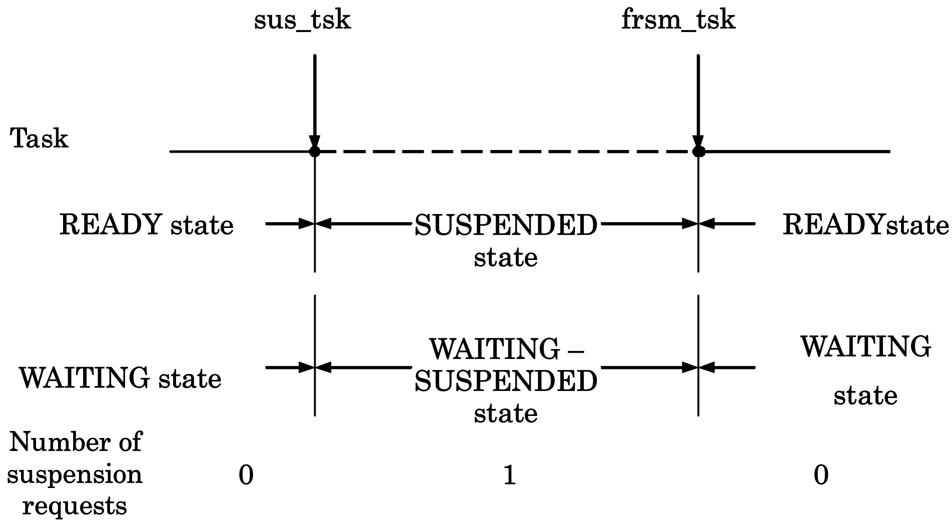

Figure 4.8 Forcible wait of a task and forcible resume

- Release task from waiting (rel_wai, irel_wai)
  Forcibly frees a task from WAITING state. A task is freed from WAITING state by this service call when it is in one of the following wait states.

  - Timeout wait state
  - Wait state entered by slp_tsk service call (+ timeout included)
  - Event flag (+ timeout included) wait state
  - Semaphore (+ timeout included) wait state
  - Message (+ timeout included) wait state
  - Data transmission (+ timeout included) wait state
  - Data reception (+ timeout included) wait state
  - Fixed–size memory block (+ timeout included) acquisition wait state
  - Short data transmission (+ timeout included) wait state
  - Short data reception (+ timeout included) wait state

- Delay task (dly_tsk)
  Keeps a task waiting for a finite length of time. Figure 4.9 shows an example in which execution of a task is kept waiting for 10 ms by the dly_tsk service call. The timeout value should be specified in ms units, and not in time tick units.

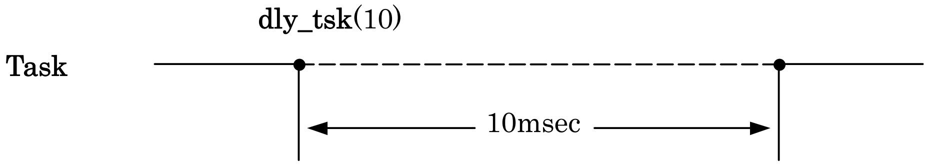

Figure 4.9 dly_tsk service call

4.1.5 Synchronization and Communication Function (Semaphore)

The semaphore is a function executed to coordinate the use of devices and other resources to be shared by several tasks in cases where the tasks simultaneously require the use of them. When, for instance, four tasks simultaneously try to acquire a total of only three communication lines as shown in Figure 4.10, communication line-to-task connections can be made without incurring contention.

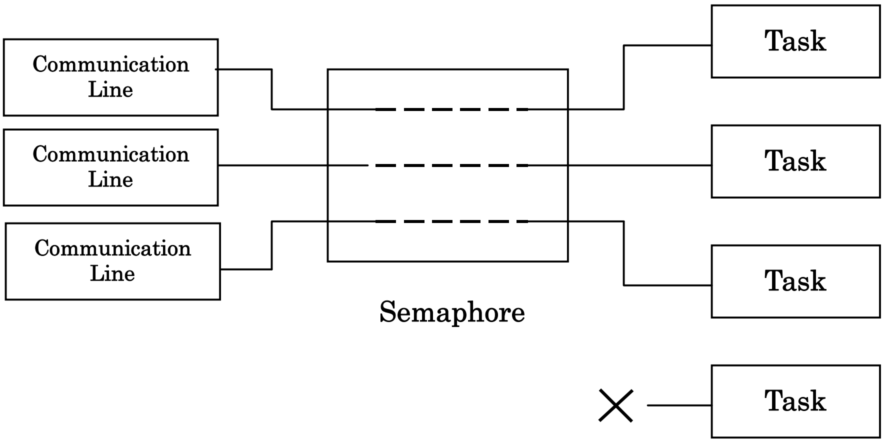

Figure 4.10 Exclusive Control by Semaphore

The semaphore has an internal semaphore counter. In accordance with this counter, the semaphore is acquired or released to prevent competition for use of the same resource.(See Figure 4.11).

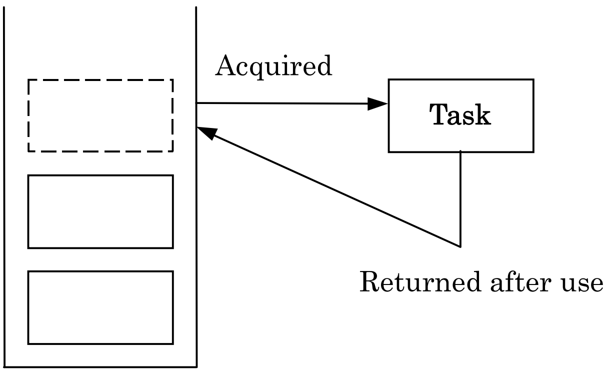

Figure 4.11 Semaphore Counter

The MR100 kernel offers the following semaphore synchronization service calls.

- Release Semaphore Resource(sig_sem, isig_sem)
  Releases one resource to the semaphore. This service call wakes up a task that is waiting for the semaphores service, or increments the semaphore counter by 1 if no task is waiting for the semaphores service.

- Acquire Semaphore Resource(wai_sem, twai_sem)
  Waits for the semaphores service. If the semaphore counter value is 0 (zero), the semaphore cannot be acquired. Therefore, the WAITING state prevails.

- Acquire Semaphore Resource(pol_sem, ipol_sem)
  Acquires the semaphore resource. If there is no semaphore resource to acquire, an error code is returned and the WAITING state does not prevail.

- Reference Semaphore Status (ref_sem, iref_sem)
  Refers the status of the target semaphore. Checks the count value and existence of the wait task for the target semaphore.
  Figure 4.12 shows example task execution control provided by the wai_sem and sig_sem service calls.

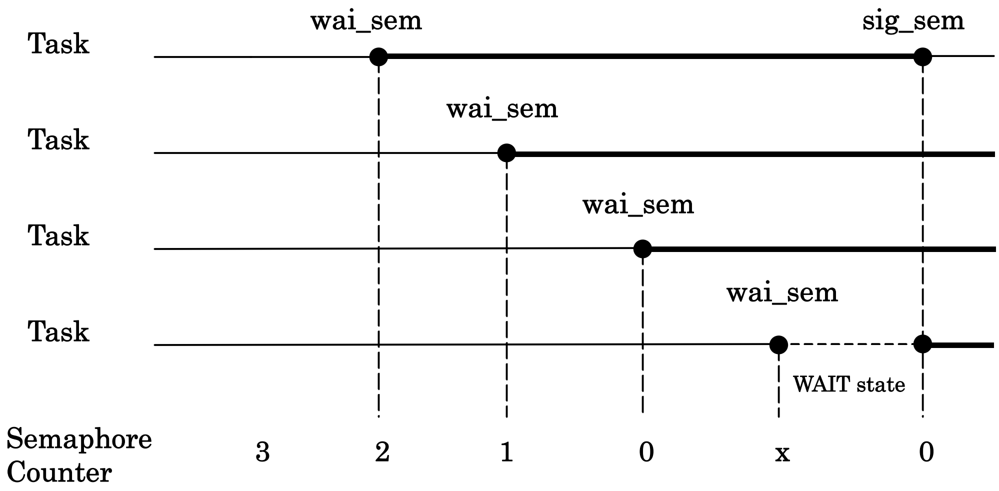

Figure 4.12 Task Execution Control by Semaphore

4.1.6 Synchronization and Communication Function (Eventflag)

The eventflag is an internal facility of MR100 that is used to synchronize the execution of multiple tasks. The eventflag uses a flag wait pattern and a 32-bit pattern to control task execution. A task is kept waiting until the flag wait conditions set are met.

It is possible to determine whether multiple waiting tasks can be enqueued in one eventflag waiting queue by specifying the eventflag attribute TA_WSGL or TA_WMUL.

Furthermore, it is possible to clear the eventflag bit pattern to 0 when the eventflag meets wait conditions by specifying TA_CLR for the eventflag attribute.

There are following eventflag service calls that are provided by the MR100 kernel.

- Set Eventflag (set_flg, iset_flg)
  Sets the eventflag so that a task waiting the eventflag is released from the WAITING state.

- Clear Eventflag (clr_flg, iclr_flg)
  Clears the Eventflag.

- Wait for Eventflag (wai_flg, twai_flg)
  Waits until the eventflag is set to a certain pattern. There are two modes as listed below in which the eventflag is waited for.

  - AND wait
    Waits until all specified bits are set.

  - OR wait
    Waits until any one of the specified bits is set

- Wait for Eventflag (polling)(pol_flg, ipol_flg)
  Examines whether the eventflag is in a certain pattern. In this service call, tasks are not placed in WAITING state.

- Reference Eventflag Status (ref_flg, iref_flg)
  Checks the existence of the bit pattern and wait task for the target eventflag.

Figure 4.13 shows an example of task execution control by the eventflag using the wai_flg and set_flg service calls.

The eventflag has a feature that it can wake up multiple tasks collectively at a time.

In Figure 4.13, there are six tasks linked one to another, task A to task F. When the flag pattern is set to 0xF by the set_flg service call, the tasks that meet the wait conditions are removed sequentially from the top of the queue. In this diagram, the tasks that meet the wait conditions are task A, task C, and task E. Out of these tasks, task A, task C, and task E are removed from the queue.

If this event flag has a TA_CLR attribute, when the waiting of Task A is canceled, the bit pattern of the event flag will be set to 0, and Task C and Task E will not be removed from queue.

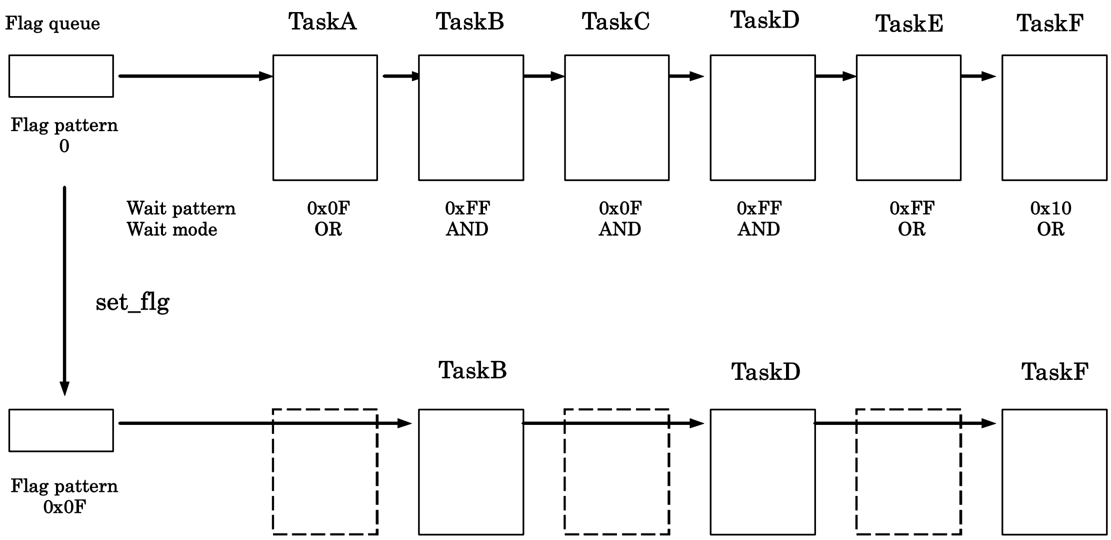

Figure 4.13 Task Execution Control by the Eventflag

4.1.7 Synchronization and Communication Function (Data Queue)

The data queue is a mechanism to perform data communication between tasks. In Figure 4.14, for example, task A can transmit data to the data queue and task B can receive the transmitted data from the data queue.

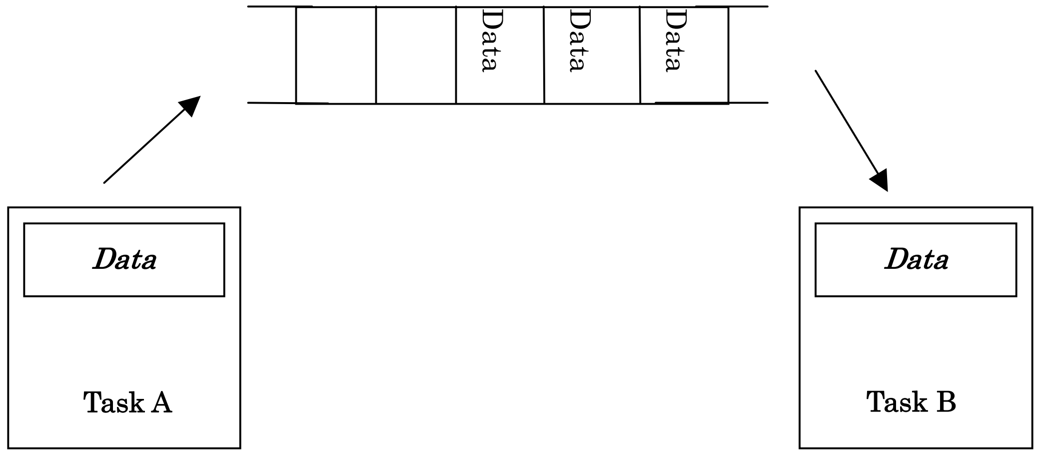

Figure 4.14 Data queue

Data in width of 32 bits can be transmitted to this data queue.

The data queue has the function to accumulate data. The accumulated data is retrieved in order of FIFO[22]. However, the number of data that can be accumulated in the data queue is limited. If data is transmitted to the data queue that is full of data, the service call issuing task goes to a data transmission wait state.

There are following data queue service calls that are provided by the MR100 kernel.

- Send to Data Queue(snd_dtq, tsnd_dtq)
  The data is transmitted to the data queue. If the data queue is full of data, the task goes to a data transmission wait state.
- Send to Data Queue (psnd_dtq, ipsnd_dtq)
  The data is transmitted to the data queue. If the data queue is full of data, the task returns error code without going to a data transmission wait state.
- Forced Send to Data Queue (fsnd_dtq, ifsnd_dtq)
  The data is transmitted to the data queue. If the data queue is full of data, the data at the top of the data queue or the oldest data is removed, and the transmitted data is stored at the tail of the data queue.
- Receive from Data Queue (rcv_dtq, trcv_dtq)
  The data is retrieved from the data queue. If the data queue has no data in it, the task is kept waiting until data is transmitted to the data queue.
- Receive from Data Queue (prcv_dtq,iprcv_dtq)
  The data is received from the data queue. If the data queue has no data in it, the task returns error code without going to a data reception wait state.
- Reference Data Queue Status (ref_dtq,iref_dtq)
  Checks to see if there are any tasks waiting for data to be entered in the target data queue and refers to the number of the data in the data queue.

[22] First In First Out

4.1.8 Synchronization and Communication Function (Mailbox)

The mailbox is a mechanism to perform data communication between tasks. In Figure 4.15, for example, task A can drop a message into the mailbox and task B can retrieve the message from the mailbox. Since mailbox-based communication is achieved by transferring the start address of a message from a task to another, this mode of communication is performed at high speed independently of the message size.

The kernel manages the message queue by means of a link list. The application should prepare a header area that is to be used for a link list. This is called the message header. The message header and the area actually used by the application to store a message are called the message packet. The kernel rewrites the content of the message header as it manages the message queue. The message header cannot be rewritten from the application. The structure of the message queue is shown in Figure 4.16. The message header has its data types defined as shown below.

T_MSG: **Mailbox message header**
T_MSG_PRI: **Mailbox message header with priority included**

Messages in any size can be enqueued in the message queue because the header area is reserved on the application side. In no event will tasks be kept waiting for transmission.

Messages can be assigned priority, so that messages will be received in order of priority beginning with the highest. In this case, TA_MPRI should be added to the mailbox attribute. If messages need to be received in order of FIFO, add TA_MFIFO to the mailbox attribute.[23] Furthermore, if tasks in a message wait state are to receive a message, the tasks can be prioritized in which order they can receive a message, beginning with one that has the highest priority. In this case, add TA_TPRI to the mailbox attribute. If tasks are to receive a message in order of FIFO, add TA_TFIFO to the mailbox attribute.[24]

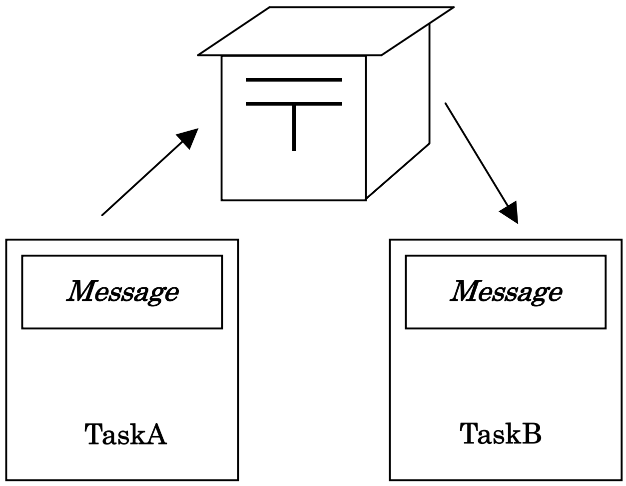

Figure 4.15 Mailbox

[23] It is in the mailbox definition "message_queue" of the configuration file that the TA_MPRI or TA_MFIFO attribute should be added.
[24] It is in the mailbox definition "wait_queue" of the configuration file that the TA_TPRI or TA_TFIFO attribute should be added.

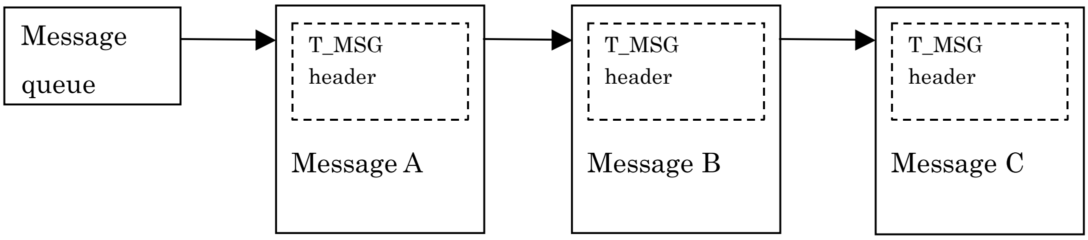

Figure 4.16 Message queue

There are following data queue service calls that are provided by the MR100 kernel.

- Send to Mailbox (snd_mbx, isnd_mbx)
  Transmits a message. Namely, a message is dropped into the mailbox.

- Receive from Mailbox (rcv_mbx, trcv_mbx)
  Receives a message. Namely, a message is retrieved from the mailbox. At this time, if the mailbox has no messages in it, the task is kept waiting until a message is sent to the mailbox.

- Receive from Mailbox (polling) (prcv_mbx, iprcv_mbx)
  Receives a message. The difference from the rcv_mbx service call is that if the mailbox has no messages in it, the task returns error code without going to a wait state.

- Reference Mailbox Status (ref_mbx, iref_mbx)
  Checks to see if there are any tasks waiting for a message to be put into the target mailbox and refers to the message present at the top of the mailbox.

4.1.9 Memory pool Management Function(Fixed-size Memory pool)

A fixed-size memory pool is the memory of a certain decided size. The memory block size is specified at the time of a configuration. Figure 4.17 is a figure about the example of a fixed-size memory pool of operation.

- Acquire Fixed-size Memory Block (get_mpf, tget_mpf)
  Acquires a memory block from the fixed-size memory pool that has the specified ID. If there are no blank memory blocks in the specified fixed-size memory pool, the task that issued this service call goes to WAITING state and is enqueued in a waiting queue.

- Acquire Fixed-size Memory Block (polling) (pget_mpf, ipget_mpf)
  Acquires a memory block from the fixed-size memory pool that has the specified ID. The difference from the get_mpf and tget_mpf service calls is that if there are no blank memory blocks in the memory pool, the task returns error code without going to WAITING state.

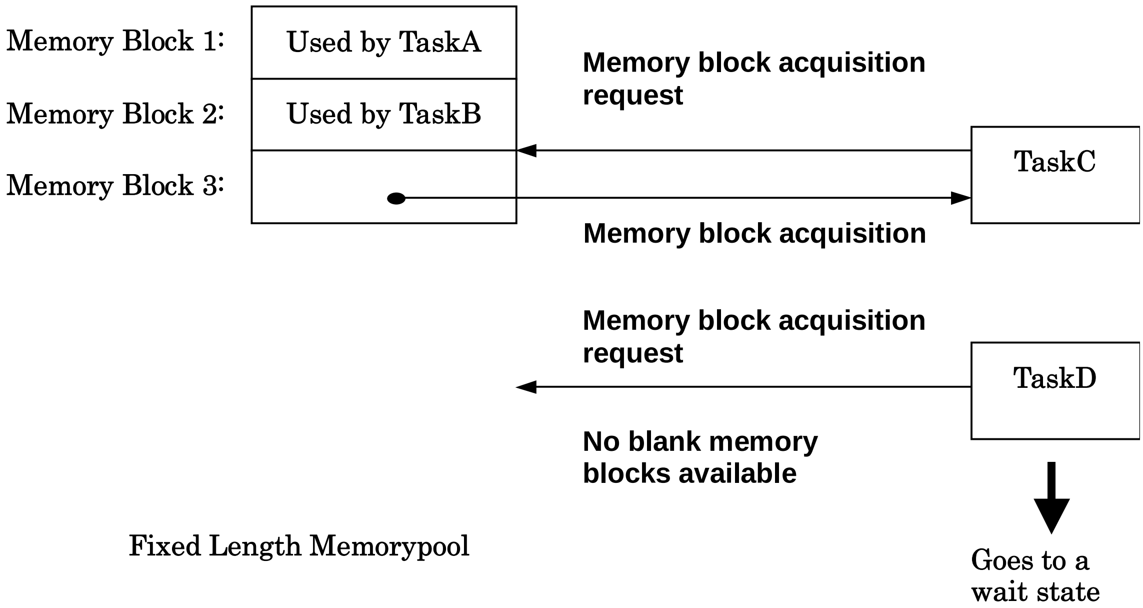

Figure 4.17 Memory Pool Management

- Release Fixed-size Memory Block (rel_mpf, irel_mpf)
  Frees the acquired memory block. If there are any tasks in a wait state for the specified fixed-size memory pool, the task enqueued at the top of the waiting queue is assigned the freed memory block. In this case, the task changes its state from WAITING state to READY state. If there are no tasks in a wait state, the memory block is returned to the memory pool.

- Reference Fixed-size Memory Pool Status (ref_mpf, iref_mpf)
  Checks the number and the size of blank blocks available in the target memory pool.

4.1.10 Variable-size Memory Pool Management Function

A variable-size memory pool refers to the one in which a memory block of any desired size can be acquired from the memory pool. The MR100 permits one of the following two memory pool management methods to be selected before the memory pool is used.

1. Normal block method
2. Small block method

Each of these methods are explained below.

[[Normal Block Method]]

The technique that allows you to arbitrary define the size of memory block acquirable from the memory pool is termed Variable-size scheme. The MR100 manages memory in terms of four fixed-size memory block sizes.

The MR100 calculates the size of individual blocks based on the maximum memory block size to be acquired. You specify the maximum memory block size using the configuration file.

- Equation for calculating four kinds of block sizes

```
a = (((max_memsize+(X-1))/ X × 8)+1) × 8
b = a × 2
c = a × 4
d = a × 8
```

max_memsize: the value specified in the configuration file
X: data size for block control (8 byte)

- Example of a configuration file

```
variable_memorypool[]{
     max_memsize     = 400; <---- Maximum size
     heap_size       = 5000;
};
```

If a variable-size memory pool is defined as shown above, the four kinds of fixed length block sizes are obtained from the define value of max_memsize as 56, 112, 224 and 448, respectively. Furthermore, the MR100 calculates the memory requested by the user based on a specified size to select the appropriate size from the four kinds of fixed length block sizes as it allocates the requested memory. In no event will a memory block other than these four kinds of size be allocated.

[[Small block method]]

Unlike the normal block method where memory is managed in four kinds of fixed length block sizes, the small block method manages memory in 12 kinds of fixed length block sizes. Since the block sizes in this method are prefixed as shown below, there is no need to specify a maximum size during configuration as in the normal block method.

The block sizes managed by the small block method are the following 12, beginning with the smallest:

24 bytes, 56 bytes, 120 bytes, 248 bytes, 504 bytes, 1,016 bytes, 2,040 bytes, 4,088 bytes, 8,184 byte, 16,376 bytes, 32,760 bytes and 65,528 bytes.

[[Comparison of Two Management Methods]]

- Processing speed
  Generally speaking, the normal block method is faster in memory allocation/deallocation processing than the small block method.

- Memory usage efficiency
  If the difference between the maximum and minimum sizes of memory to be acquired is 8 times or more, the small block method is higher in memory usage efficiency than the other method.

- Ease of configuration
  For the normal block method, it is necessary that the maximum memory size to be acquired be known to the MR100. However, this is unnecessary for the small block method..

The variable-length memory pool management service calls provided by the MR100 include the following.

- Get a memory block (pget_mpl)
  The block size specified by the user is acquired by first rounding it to the optimum block size among the four kinds of block sizes and then acquiring a memory block of the rounded size from the memory pool
  For example, if the user requests 200 bytes of memory, the requested size is rounded to 224 bytes, so that 224 bytes of memory is acquired. If a requested block of memory is successfully acquired, the start address of the acquired memory block and error code E_OK are returned. If memory acquisition fails, error code E_TMOUT is returned.

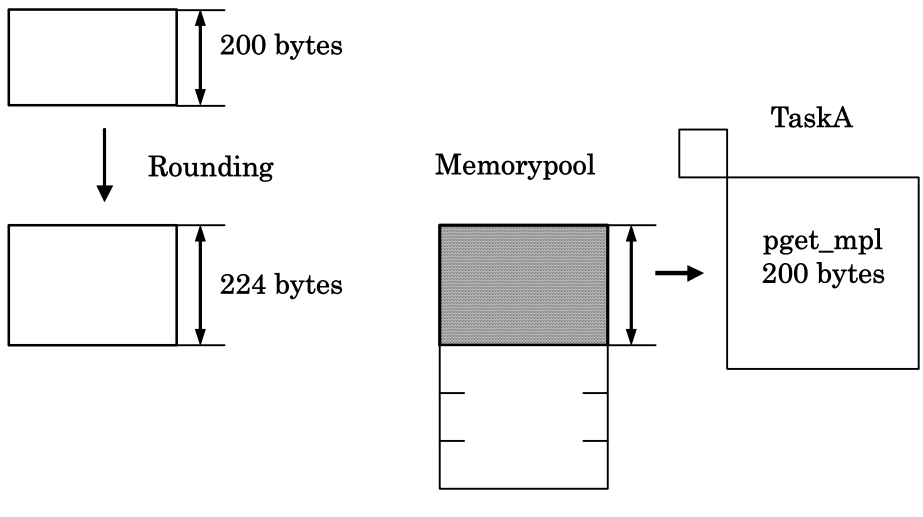

Figure 4.18 pget_mpl processing

- Release Acquire Variable-size Memory Block (rel_mpl)[25]
  Releases a acquired memory block by pget_mpl service call.

[25] The validity of the address of the memory block to which MR100 is passed as an argument and to release is not judged. Therefore, operation at the time of releasing the memory block which is already released or releasing the memory block which has not been gained is not guaranteed.

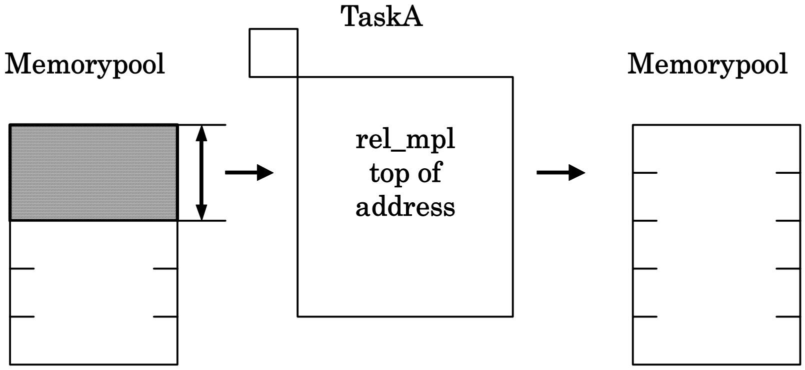

Figure 4.19 rel_mpl processing

- Reference Acquire Variable-size Memory Pool Status (ref_mpl, iref_mpl)
  Checks the total free area of the memory pool, and the size of the maximum free area that can immediately be acquired.

4.1.11 Time Management Function

The time management function provides system time management, time reading[26], time setup[27], and the functions of the alarm handler, which actuates at preselected times, and the cyclic handler, which actuates at preselected time intervals.

The MR100 kernel requires one timer for use as the system clock. There are following time management service calls that are provided by the MR100 kernel. Note, however, that the system clock is not an essential function of MR100. Therefore, if the service calls described below and the time management function of the MR100 are unused, a timer does not need to be occupied for use by MR100.

- Place a task in a finite time wait state by specifying a timeout value
  A timeout can be specified in a service call that places the issuing task into WAITING state.[28] This service call includes tslp_tsk, twai_flg, twai_sem, tsnd_dtq, trcv_dtq, trcv_mbx, tget_mpf, vtsnd_dtq, and vtrcv_dtq. If the wait cancel condition is not met before the specified timeout time elapses, the error code E_TMOUT is returned, and the task is freed from the waiting state. If the wait cancel condition is met, the error code E_OK is returned.
  The timeout time should be specified in ms units.

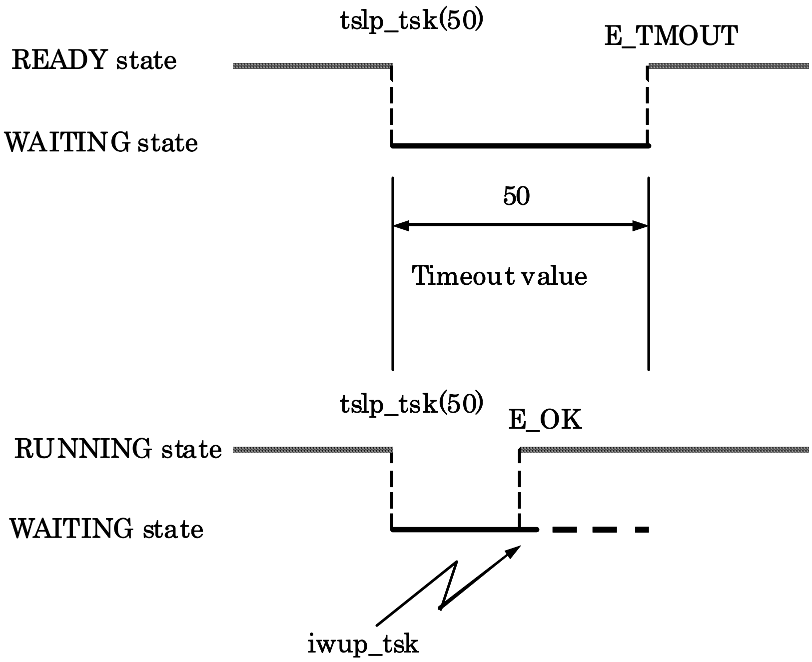

Figure 4.20 Timeout Processing

MR100 guarantees that as stipulated in µITRON specification, timeout processing is not performed until a time equal to or greater than the specified timeout value elapses. More specifically, timeout processing is performed with the following timing.

1. If the timeout value is 0 (for only dly_tsk)[29]
   The task times out at the first time tick after the service call is issued.[30]
2. If the timeout value is a multiple of time tick interval
   The timer times out at the (timeout value / time tick interval) + first time tick. For example, if the time tick interval is 10 ms and the specified timeout value is 40 ms, then the timer times out at the fifth occurrence of the time tick. Similarly, if the time tick interval is 5 ms and the specified timeout value is 15 ms, then the timer times out at the fourth occurrence of the time tick.

---

[26] get_tim service call
[27] set_tim service call
[28] SUSPENDED state is not included.
[29] Strictly, in a dly_tsk service call, the "timeout value" is not correct. "delay time" is correct.
[30] Strictly, in a dly_tsk service call, a timeout is not carried out, but the waiting for delay is canceled and the service call carries out the normal end.

3. If the timeout value is not a multiple of time tick interval
   The timer times out at the (timeout value / time tick interval) + second time tick. For example, if the time tick interval is 10 ms and the specified timeout value is 35 ms, then the timer times out at the fifth occurrence of the time tick.

- Set System Time (set_tim)

- Reference System Time (get_tim)

  The system time indicates an elapsed time from when the system was reset by using 48-bit data. The time is expressed in ms units.

4.1.12 Cyclic Handler Function

The cyclic handler is a time event handler that is started every startup cycle after a specified startup phase has elapsed.

The cyclic handler may be started with or without saving the startup phase. In the former case, the cyclic handler is started relative to the point in time at which it was generated. In the latter case, the cyclic handler is started relative to the point in time at which it started operating. Figure 4.21 and Figure 4.22 show typical operations of the cyclic handler.

If the startup cycle is shorter than the time tick interval, the cyclic handler is started only once every time tick supplied (processing equivalent to isig_tim). For example, if the time tick interval is 10 ms and the startup cycle is 3 ms and the cyclic handler has started operating when a time tick is supplied, then the cyclic handler is started every time tick.

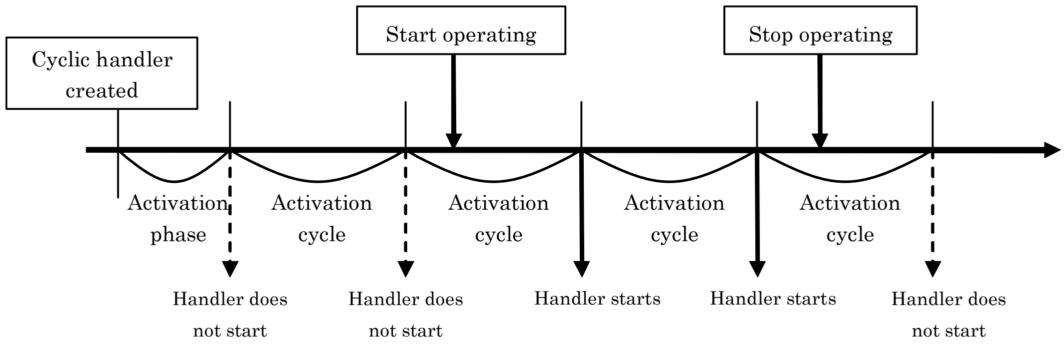

Figure 4.21 Cyclic handler operation in cases where the activation phase is saved

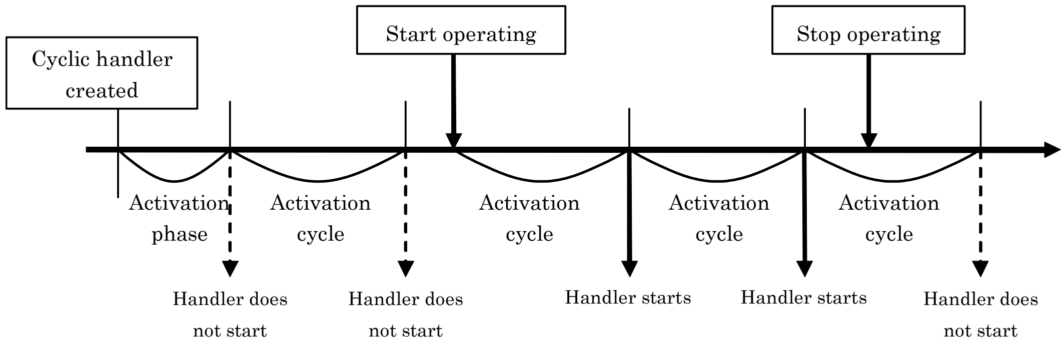

Figure 4.22 Cyclic handler operation in cases where the activation phase is not saved

- Start Cyclic Handler Operation (sta_cyc, ista_cyc)
  Causes the cyclic handler with the specified ID to operational state.

- Stop Cyclic Handler Operation (stp_cyc, istp_cyc)
  Causes the cyclic handler with the specified ID to non-operational state.

- Reference Cyclic Handler Status (ref_cyc, iref_cyc)
  Refers to the status of the cyclic handler. The operating status of the target cyclic handler and the remaining time before it starts next time are inspected.

4.1.13 Alarm Handler Function

The alarm handler is a time event handler that is started only once at a specified time.

Use of the alarm handler makes it possible to perform time-dependent processing. The time of day is specified by a relative time. Figure 4.23 shows a typical operation of the alarm handler.

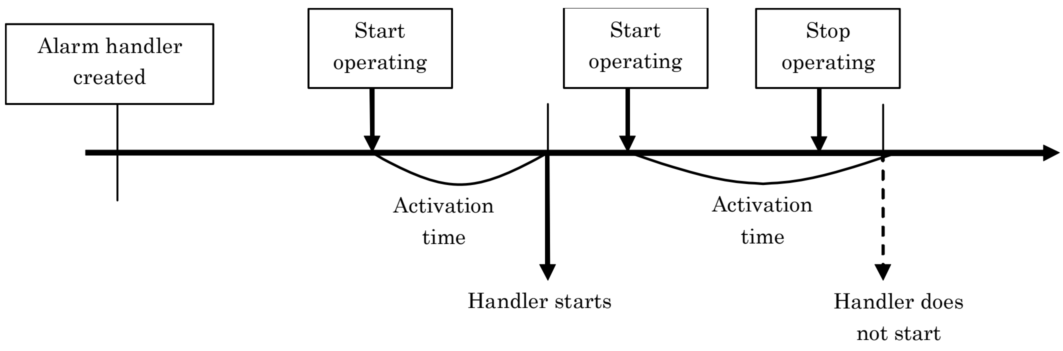

Figure 4.23 Typical operation of the alarm handler

- Start Alarm Handler Operation (sta_alm, ista_alm)
  Causes the alarm handler with the specified ID to operational state.

- Stop alarm Handler Operation (stp_alm, istp_alm)
  Causes the alarm handler with the specified ID to non-operational state.

- Reference Alarm Handler Status (ref_alm, iref_alm)
  Refers to the status of the alarm handler. The operating status of the target alarm handler and the remaining time before it starts are inspected.

4.1.14 System Status Management Function

- Rotate Task Precedence (rot_rdq, irot_rdq)
  This service call establishes the TSS (time-sharing system). That is, if the ready queue is rotated at regular intervals, round robin scheduling required for the TSS is accomplished (See Figure 4.24)

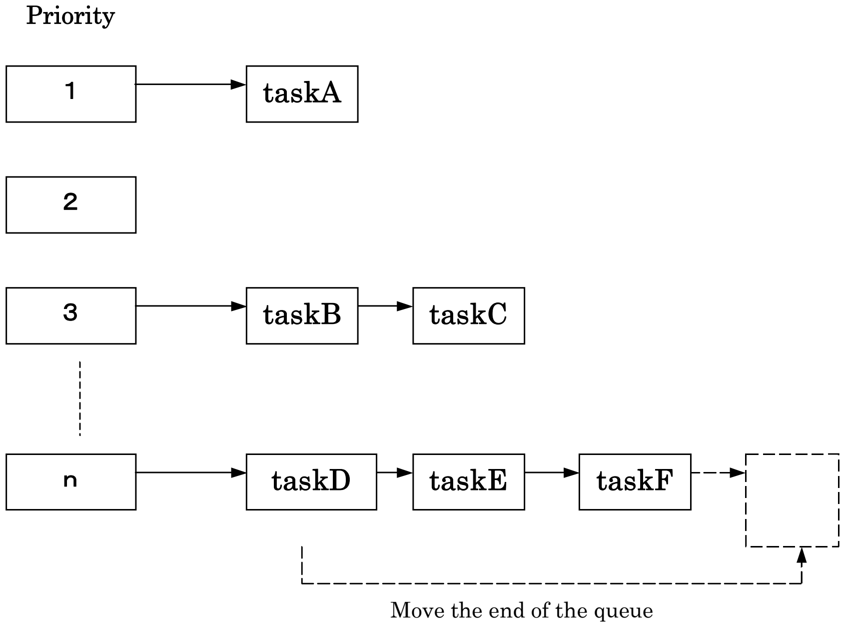

Figure 4.24 Ready Queue Management by rot_rdq Service Call

- Reference task ID in the RUNNING state(get_tid, iget_tid)
  References the ID number of the task in the RUNNING state. If issued from the handler, TSK_NONE(=0) is obtained instead of the ID number.
- Lock the CPU (loc_cpu, iloc_cpu)
  Places the system into a CPU locked state.
- Unlock the CPU (unl_cpu, iunl_cpu)
  Frees the system from a CPU locked state.
- Disable dispatching (dis_dsp)
  Places the system into a dispatching disabled state.
- Enable dispatching (ena_dsp)
  Frees the system from a dispatching disabled state.
- Reference context (sns_ctx)
  Gets the context status of the system.
- Reference CPU state (sns_loc)
  Gets the CPU lock status of the system.
- Reference dispatching state (sns_dsp)
  Gets the dispatching disable status of the system.
- Reference dispatching pending state (sns_dpn)
  Gets the dispatching pending status of the system.

4.1.15 Interrupt Management Function

The interrupt management function provides a function to process requested external interrupts in real time.

The interrupt management service calls provided by the MR100 kernel include the following:

- Returns from interrupt handler (ret_int)
  The ret_int service call activates the scheduler to switch over tasks as necessary when returning from the interrupt handler.
  When using the C language,[31], this function is automatically called at completion of the handler function. In this case, therefore, there is no need to invoke this service call.

Figure 4.25 shows an interrupt processing flow. Processing a series of operations from task selection to register restoration is called a "scheduler.".

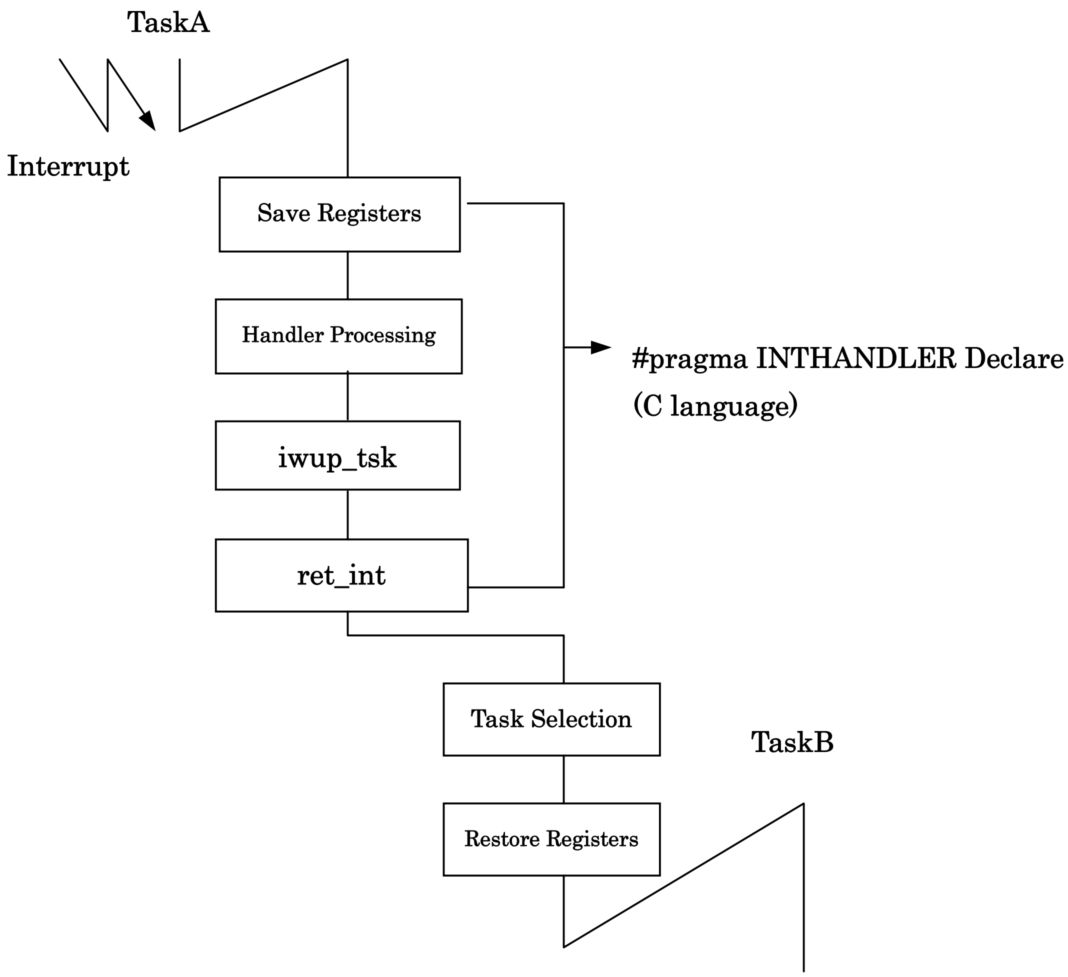

Figure 4.25 Interrupt process flow

[31] In the case that the interruput handler is specified by "#pragma INTHANDLER".

4.1.16 System Configuration Management Function

This function inspects the version information of MR100.

- References Version Information(ref_ver, iref_ver)
  The ref_ver service call permits the user to get the version information of MR100. This version information can be obtained in the standardized format of µITRON specification.

4.1.17 Extended Function (Short Data Queue)

The short data queue is a function outside the scope of µITRON 4.0 Specification. The data queue function handles data as consisting of 32 bits, whereas the short data queue handles data as consisting of 16 bits. Both behave the same way except only that the data sizes they handle are different.

- Send to Short Data Queue (vsnd_dtq, vtsnd_dtq)
  The data is transmitted to the short data queue. If the short data queue is full of data, the task goes to a data transmission wait state.

- Send to Short Data Queue (vpsnd_dtq, vipsnd_dtq)
  The data is transmitted to the short data queue. If the short data queue is full of data, the task returns error code without going to a data transmission wait state.

- Forced Send to Short Data Queue (vfsnd_dtq, vifsnd_dtq)
  The data is transmitted to the short data queue. If the short data queue is full of data, the data at the top of the short data queue or the oldest data is removed, and the transmitted data is stored at the tail of the short data queue.

- Receive from Short Data Queue(vrcv_dtq, vtrcv_dtq)
  The data is retrieved from the short data queue. If the short data queue has no data in it, the task is kept waiting until data is transmitted to the short data queue.

- Receive from Short Data Queue (vprcv_dtq, viprcv_dtq)
  The data is received from the short data queue. If the short data queue has no data in it, the task returns error code without going to a data reception wait state.

- Reference Short Data Queue Status (vref_dtq, viref_dtq)
  Checks to see if there are any tasks waiting for data to be entered in the target short data queue and refers to the number of the data in the short data queue.

4.1.18 Extended Function (Reset Function)

The reset function is a function outside the scope of μITRON 4.0 Specification. It initializes the mailbox, data queue, and memory pool, etc.

- Clear Data Queue Area (vrst_dtq)
  Initializes the data queue. If there are any tasks waiting for transmission, they are freed from WAITING state and the error code EV_RST is returned.

- Clear Mailbox Area (vrst_mbx)
  Initializes the mailbox.

- Clear Fixed-size Memory Pool Area (vrst_mpf)
  Initializes the fixed-size memory pool. If there are any tasks in WAITING state, they are freed from the WAITING state and the error code EV_RST is returned.

- Clear Variable-size Memory Pool Area (vrst_mpl)
  Initializes the variable length memory pool.

- Clear Short Data Queue Area (vrst_vdtq)
  Initializes the short data queue. If there are any tasks waiting for transmission, they are freed from WAITING state and the error code EV_RST is returned.

5. Service call reffernce

5.1 Task Management Function

Specifications of the task management function of MR100 are listed in Table 5.1 below. The task description languages in item No. 4 are those specified in the GUI configurator. They are not output to a configuration file, nor are the MR100 kernel concerned with them.

The task stack permits a section name to be specified for each task individually.

Table 5.1 Specifications of the Task Management Function

No.	Item	Content
1	Task ID	1-255
2	Task priority	1-255
3	Maximum number of activation request count	255
4	Task attribute	TA_HLNG : Tasks written in high-level language TA_ASM : Tasks written in assem-bly language TA_ACT : Startup attribute
5	Task stack	Section specifiable

Table 5.2 List of Task Management Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	act_tsk	**[S]**	Activates task	O		O	O	O	
2	iact_tsk	**[S]**			O	O	O	O	
3	can_act	**[S]**	Cancels task activation request	O		O	O	O	
4	ican_act				O	O	O	O	
5	sta_tsk	**[B]**	Starts task and specifies start code	O		O	O	O	
6	ista_tsk				O	O	O	O	
7	ext_tsk	**[S][B]**	Exits current task	O		O	O	O	O
8	ter_tsk	**[S][B]**	Forcibly terminates a task	O		O	O	O	
9	chg_pri	**[S][B]**	Changes task priority	O		O	O	O	
10	ichg_pri				O	O	O	O	
11	get_pri	**[S]**	Refers to task priority	O		O	O	O	
12	iget_pri				O	O	O	O	
13	ref_tsk		Refers to task state	O		O	O	O	
14	iref_tsk				O	O	O	O	
15	ref_tst		Refers to task state (simple version)	O		O	O	O	
16	iref_tst				O	O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls

- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

act_tsk	**Activate task**
iact_tsk	**Activate task (handler only)**

[[C Language API]]

```
ER ercd = act_tsk( ID tskid );
ER ercd = iact_tsk( ID tskid );
```

- **Parameters**

ID	tskid	ID number of the task to be started

- **Return parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr100.inc
act_tsk  TSKID
iact_tsk TSKID
```

- **Parameters**

TSKID	ID number of the task to be started

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	Task ID

[[Error Code]]

E_QOVR	Queuing overflow

[[Functional description]]

This service call starts the task indicated by tskid. The started task goes from DORMANT state to READY state or RUNNING state.

The following lists the processing performed on startup.

1. Initializes the current priority of the task.
2. Clears the number of queued wakeup requests.
3. Clears the number of suspension requests.

Specifying tskid=TSK_SELF(0) specifies the issuing task itself. The task has passed to it as parameter the extended information of it that was specified when the task was created. If TSK_SELF is specified for tskid in non-task context, operation of this service call cannot be guaranteed.

If the target task is not in DORMANT state, a task activation request by this service call is enqueued. In other words, the activation request count is incremented by 1. The maximum value of the task activation request is 255. If this limit is exceeded, the error code E_QOVR is returned.

If TSK_SELF is specified for tskid, the issuing task itself is made the target task.

If this service call is to be issued from task context, use act_tsk; if issued from non-task context, use iact_tsk.

[[Example program statement]]

<<Example statement in C language>>

```
#include <itron.h>
#include <kernel.h>
#include “kernel_id.h”
void task1( VP_INT stacd )
{
   ER ercd;
         :
     ercd = act_tsk( ID_task2 );
         :
}
void task2( VP_INT stacd )
{
        :
     ext_tsk();
}
```

<<Example statement in assembly language>>

```
    .INCLUDE    mr100.inc
    .GLB        task
task:
                :
    PUSH.W      R2
    act_tsk             #ID_TASK3
                :
```

can_act	Cancel task activation request
ican_act	Cancel task activation request (handler only)

[[C Language API]]

```
ER_UINT actcnt = can_act( ID tskid );
ER_UINT actcnt = ican_act( ID tskid );
```

● Parameters

ID	tskid	ID number of the task to cancel

● Return Parameters

ER_UINT	actcnt > 0	Canceled activation request count
	actcnt < 0	Error code

[[Assembly language API]]

```
.include mr100.inc
can_act  TSKID
ican_act TSKID
```

● Parameters

TSKID	ID number of the task to cancel

● Register contents after service call is issued

Register name	Content after service call is issued
R2R0	Canceled startup request count or error code

[[Error code]]

None

[[Functional description]]

This service call finds the number of task activation requests enqueued for the task indicated by tskid, returns the result as a return parameter, and at the same time invalidates all of the task's activation requests.

Specifying tskid=TSK_SELF(0) specifies the issuing task itself. If TSK_SELF is specified for tskid in non-task context, operation of this service call cannot be guaranteed.

This service call can be invoked for a task in DORMANT state as the target task. In that case, the return parameter is 0.

If this service call is to be issued from task context, use can_act; if issued from non-task context, use ican_act.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task1()
  {
    ER_UINT actcnt;
          :
     actcnt = can_act( ID_task2 );
          :
  }
  void task2()
  {
         :
     ext_tsk();
  }
```

<<Example statement in assembly language>>

```
    .INCLUDE    mr100.inc
    .GLB        task
task:
                :
    can_act     #ID_TASK2
                :
```

sta_tsk — Activate task with a start code
ista_tsk — Activate task with a start code (handler only)

[[C Language API]]

```
ER ercd = sta_tsk( ID tskid,VP_INT stacd );
ER ercd = ista_tsk ( ID tskid,VP_INT stacd );
```

- **Parameters**

ID	tskid	ID number of the target task
VP_INT	stacd	Task start code

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr100.inc
sta_tsk  TSKID,STACD
ista_tsk TSKID,STACD
```

- **Parameters**

TSKID	ID number of the target task
STATCD	Task start code

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R3R1	Task start code
R2	ID number of the target task

[[Error code]]

E_OBJ	Object status invalid (task indicated by tskid is not DOMANT state)

[[Functional description]]

This service call starts the task indicated by tskid. In other words, it places the specified task from DORMANT state into READY state or RUNNING state. This service call does not enqueue task activation requests. Therefore, if a task activation request is issued while the target task is not DORMANT state, the error code E_OBJ is returned to the service call issuing task. This service call is effective only when the specified task is in DORMANT state. The task start code stacd is 32 bits long. This task start code is passed as parameter to the activated task.

If a task is restarted that was once terminated by ter_tsk or ext_tsk, the task performs the following as it starts up.

1. Initializes the current priority of the task.

2. Clears the number of queued wakeup requests.

3. Clears the number of nested forcible wait requests.

If this service call is to be issued from task context, use sta_tsk; if issued from non-task context, use ista_tsk.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
     ER ercd;
     VP_INT stacd = 0;
     ercd = sta_tsk( ID_task2, stacd );
            :
  }
  void task2(VP_INT msg)
  {
     if(msg == 0)
            :
  }
```

<<Example statement in assembly language>>

```
     .INCLUDE    mr100.inc
     .GLB        task
task:
                 :
     PUSHM       R3R1
     PUSH.W      R2
     sta_tsk     #ID_TASK4,#100
                 :
```

ext_tsk Terminate invoking task

[[C Language API]]

```
ER ercd = ext_tsk();
```

- **Parameters**

  None

- **Return Parameters**

  Not return from this service call

[[Assembly language API]]

```
.include mr100.inc
ext_tsk
```

- **Parameters**

  None

- **Register contents after service call is issued**

  Not return from this service call

[[Error code]]

Not return from this service call

[[Functional description]]

This service call terminates the invoking task. In other words, it places the issuing task from RUNNING state into DORMANT state. However, if the activation request count for the issuing task is 1 or more, the activation request count is decremented by 1, and processing similar to that of act_tsk or iact_tsk is performed. In that case, the task is placed from DORMANT state into READY state. The task has its extended information passed to it as parameter when the task starts up.

This service call is designed to be issued automatically at return from a task.

In the invocation of this service call, the resources the issuing task had acquired previously (e.g., semaphore) are not released.

This service call can only be used in task context. This service call can be used even in a CPU locked state, but cannot be used in non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task(void)
  {
       :
    ext_tsk();
  }
```

<<Example statement in assembly language>>

```
    .INCLUDE    mr100.inc
    .GLB        task
task:
        :
    ext_tsk
```

ter_tsk Terminate task

[[C Language API]]

```
ER ercd = ter_tsk( ID tskid );
```

- **Parameters**

ID	tskid	ID number of the forcibly terminated task

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr100.inc
ter_tsk  TSKID
```

- **Parameters**

TSKID	ID number of the forcibly terminated task

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	ID number of the target task

[[Error code]]

E_OBJ	Object status invalid(task indicated by tskid is an inactive state)
E_ILUSE	Service call improperly used task indicated by tskid is the issuing task itself)

[[Functional description]]

This service call terminates the task indicated by tskid. If the activation request count of the target task is equal to or greater than 1, the activation request count is decremented by 1, and processing similar to that of act_tsk or iact_tsk is performed. In that case, the task is placed from DORMANT state into READY state. The task has its extended information passed to it as parameter when the task starts up.

If a task specifies its own task ID or TSK_SELF, an E_ILUSE error is returned.

If the specified task was placed into WAITING state and has been enqueued in some waiting queue, the task is dequeued from it by execution of this service call. However, the semaphore and other resources the specified task had acquired previously are not released.

If the task indicated by tskid is in DORMANT state, it returns the error code E_OBJ as a return value for the service call.

This service call can only be used in task context, and cannot be used in non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
        :
    ter_tsk( ID_main );
        :
}
```

<<Example statement in assembly language>>

```
    .INCLUDE    mr100.inc
    .GLB        task
task:
                :
    PUSH.W      R2
    ter_tsk     #ID_TASK3
                :
```

chg_pri — Change task priority
ichg_pri — Change task priority(handler only)

[[C Language API]]

```
ER ercd = chg_pri( ID tskid, PRI tskpri );
ER ercd = ichg_pri( ID tskid, PRI tskpri );
```

- **Parameters**

ID	tskid	ID number of the target task
PRI	tskpri	Priority of the target task

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr100.inc
chg_pri  TSKID,TSKPRI
ichg_pri  TSKID,TSKPRI
```

- **Parameters**

TSKID	ID number of the target task
TSKPRI	Priority of the target task

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R3	Priority of the target task
R2	ID number of the target task

[[Error code]]

E_OBJ	Object status invalid(task indicated by tskid is an inactive state)

[[Functional description]]

The priority (base priority) of the task specified by tskid is changed to the value indicated by tskpri, and tasks are rescheduled based on the result of change.

If this service call is executed on a task queued in a ready queue (including a task under execution) or a task in a wait queue in which tasks are queued in order of priority, the object task is moved to the tail end of the tasks of relevant priority in the queue. When the same priority as before is specified, the object task is moved to the tail end of that queue also.

The smaller the number, the higher the task priority, with numeral 1 assigned the highest priority. The minimum numeric value specifiable as priority is 1. Furthermore, the maximum value of priority is the one specified in a configuration file, and the specifiable range of priority is 1 to 255. For example, if the following statement is written in a configuration file,

```
system{
        stack_size    = 0x100;
        priority      = 13;
};
```

then the specifiable range of priority is 1 to 13.
If TSK_SELF is specified, the priority (base priority) of the issuing task is changed. If TSK_SELF is specified for tskid in a non-task context, the program operation cannot be guaranteed. If TPRI_INI is specified, the priority of a task is changed to its startup priority specified when it is generated. The changed task priority (base priority) remains effective until the task terminates or this service call is reexecuted.
If the task indicated by tskid is in an inactive (DORMANT) state, error code E_OBJ is returned as the service call's return value.
To use these service calls from task contexts, be sure to use chg_pri; to use them from non-task contexts, be sure to use ichg_pri.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include “kernel_id.h”
  void task()
  {
          :
      chg_pri( ID_task2, 2 );
          :
  }
```

<<Example statement in assembly language>>

```
    .INCLUDE    mr100.inc
    .GLB        task
task:
                :
    PUSH.W      R2
    PUSH.W      R3
    chg_pri     #ID_TASK3,#1
                :
```

get_pri — Reference task priority
iget_pri — Reference task priority(handler only)

[[C Language API]]

```
ER ercd = get_pri( ID tskid, PRI *p_tskpri );
ER ercd = iget_pri( ID tskid, PRI *p_tskpri );
```

- **Parameters**

ID	tskid	ID number of the target task
PRI	*p_tskpri	Pointer to the area to which task priority is returned

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr100.inc
get_pri  TSKID
iget_pri  TSKID
```

- **Parameters**

TSKID	ID number of the target task

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	Acquired task priority

[[Error code]]

E_OBJ	Object status invalid(task indicated by tskid is an inactive state)

[[Functional description]]

This service call returns the priority of the task indicated by tskid to the area indicated by p_tskpri. If TSK_SELF is specified, the priority of the issuing task itself is acquired. If TSK_SELF is specified for tskid in non-task context, operation of the service call cannot be guaranteed.

If the task indicated by tskid is in DORMANT state, it returns the error code E_OBJ as a return value for the service call.

If this service call is to be issued from task context, use get_pri; if issued from non-task context, use iget_pri.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
     PRI p_tskpri;
     ER ercd;
            :
     ercd = get_pri( ID_task2, &p_tskpri );
            :
  }
```

<<Example statement in assembly language>>

```
    .INCLUDE    mr100.inc
    .GLB        task
task:
                :
    get_pri     #ID_TASK2
                :
```

ref_tsk — Reference task status
iref_tsk — Reference task status (handler only)

[[C Language API]]

```
ER ercd = ref_tsk( ID tskid, T_RTSK *pk_rtsk );
ER ercd = iref_tsk( ID tskid, T_RTSK *pk_rtsk );
```

● **Parameters**

ID	tskid	ID number of the target task
T_RTSK	*pk_rtsk	Pointer to the packet to which task status is returned

● **Return Parameters**

ER	ercd	Terminated normally (E_OK)

Contents of pk_rtsk

typedef struct t_rtsk{

STAT	tskstat	+0	2	Task status
PRI	tskpri	+2	2	Current priority of task
PRI	tskbpri	+4	2	Base priority of task
STAT	tskwait	+6	2	Cause of wait
ID	wobjid	+8	2	Waiting object ID
TMO	lefttmo	+10	4	Left time before timeout
UINT	actcnt	+14	4	Number of queued activation request counts
UINT	wupcnt	+18	4	Number of queued wakeup request counts
UINT	suscnt	+22	4	Number of nested suspension request counts

} T_RTSK;

[[Assembly language API]]

```
.include mr100.inc
ref_tsk  TSKID, PK_RTSK
iref_tsk  TSKID, PK_RTSK
```

● **Parameters**

TSKID	ID number of the target task
PK_RTSK	Pointer to the packet to which task status is returned

● **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	ID number of the target task
A1	Pointer to the packet to which task status is returned

[[Error code]]

None

[[Functional description]]

This service call inspects the status of the task indicated by tskid and returns the current information on that task to the area pointed to by pk_rtsk as a return parameter. If TSK_SELF is specified, the status of the issuing task itself is inspected. If TSK_SELF is specified for tskid in non-task context, operation of the service call cannot be guaranteed.

◆ **tskstat (task status)**

tskstat has one of the following values returned to it depending on the status of the specified task.

- TTS_RUN(0x0001) RUNNING state
- TTS_RDY(0x0002) READY state
- TTS_WAI(0x0004) WAITING state
- TTS_SUS(0x0008) SUSPENDED state
- TTS_WAS(0x000C) WAITING-SUSPENDED state
- TTS_DMT(0x0010) DORMANT state

◆ **tskpri (current priority of task)**

tskpri has the current priority of the specified task returned to it. If the task is in DOMANT state, tskpri is indeterminate.

◆ **tskbpri (base priority of task)**

tskbpri has the base priority of the specified task returned to it. If the task is in DOMANT state, tskbpri is indeterminate.

◆ **tskwait (cause of wait)**

If the target task is in a wait state, one of the following causes of wait is returned. The values of the respective causes of wait are listed below. If the task status is other than a wait state (TTS_WAI or TTS_WAS), tskwait is indeterminate.

- TTW_SLP (0x0001) Kept waiting by slp_tsk or tslp_tsk
- TTW_DLY (0x0002) Kept waiting by dly_tsk
- TTW_SEM (0x0004) Kept waiting by wai_sem or twai_sem
- TTW_FLG (0x0008) Kept waiting by wai_flg or twai_flg
- TTW_SDTQ(0x0010) Kept waiting by snd_dtq or tsnd_dtq
- TTW_RDTQ(0x0020) Kept waiting by rcv_dtq or trcv_dtq
- TTW_MBX (0x0040) Kept waiting by rcv_mbx or trcv_mbx
- TTW_MPF (0x2000) Kept waiting by get_mpf or tget_mpf
- TTW_VSDTQ (0x4000) Kept waiting by vsnd_dtq or vtsnd_dtq[32]
- TTW_VRDTQ(0x8000) Kept waiting by vrcv_dtq or vtrcv_dtq

◆ **wobjid (waiting object ID)**

If the target task is in a wait state (TTS_WAI or TTS_WAS), the ID of the waiting target object is returned. Otherwise, wobjid is indeterminate.

◆ **lefttmo(left time before timeout)**

If the target task has been placed in WAITING state (TTS_WAI or TTS_WAS) by other than dly_tsk, the left time before it times out is returned. If the task is kept waiting perpetually, TMO_FEVR is returned. Otherwise, lefttmo is indeterminate.

◆ **actcnt(task activation request)**

The number of currently queued task activation request is returned.

◆ **wupcnt (wakeup request count)**

The number of currently queued wakeup requests is returned. If the task is in DORMANT state, wupcnt is indeterminate.

◆ **suscnt (suspension request count)**

The number of currently nested suspension requests is returned. If the task is in DORMANT state, suscnt is indeterminate.

If this service call is to be issued from task context, use ref_tsk; if issued from non-task context, use iref_tsk.

[32] TTW_VSDTQ and TTW_VRDTQ are the causes of wait outside the scope of µITRON 4.0 Specification.

[[Example program statement]]

<<Example statement in C language>>

```
#include <itron.h>
#include <kernel.h>
#include “kernel_id.h”
void task()
{
    T_RTSK rtsk;
    ER ercd;
        :
    ercd = ref_tsk( ID_main, &rtsk );
        :
}
```

<<Example statement in assembly language>>

```
_refdata:       .blkb   26
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    PUSH.L      A1
    ref_tsk     #TSK_SELF,#_refdata
          :
```

ref_tst Reference task status (simplified version)
iref_tst Reference task status (simplified version, handler only)

[[C Language API]]

```
ER ercd = ref_tst( ID tskid, T_RTST *pk_rtst );
ER ercd = iref_tst( ID tskid, T_RTST *pk_rtst );
```

- **Parameters**

ID	tskid	ID number of the target task
T_RTST	*pk_rtst	Pointer to the packet to which task status is returned

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

Contents of pk_rtsk

```
typedef    struct    t_rtst{
      STAT    tskstat     +0    2    Task status
      STAT    tskwait     +2    2    Cause of wait
} T_RTST;
```

[[Assembly language API]]

```
.include mr100.inc
ref_tst  TSKID, PK_RTST
iref_tst  TSKID, PK_RTST
```

- **Parameters**

TSKID	ID number of the target task
PK_RTST	Pointer to the packet to which task status is returned

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
A0	ID number of the target task
A1	Pointer to the packet to which task status is returned

[[Error code]]

None

[[Functional description]]

This service call inspects the status of the task indicated by tskid and returns the current information on that task to the area pointed to by pk_rtst as a return value. If TSK_SELF is specified, the status of the issuing task itself is inspected. If TSK_SELF is specified for tskid in non-task context, operation of the service call cannot be guaranteed.

◆ **tskstat (task status)**

tskstat has one of the following values returned to it depending on the status of the specified task.

- TTS_RUN(0x0001) RUNNING state
- TTS_RDY(0x0002) READY state
- TTS_WAI(0x0004) WAITING state
- TTS_SUS(0x0008) SUSPENDED state
- TTS_WAS(0x000C) WAITING-SUSPENDED state
- TTS_DMT(0x0010) DORMANT state

◆ **tskwait (cause of wait)**

If the target task is in a wait state, one of the following causes of wait is returned. The values of the respective causes of wait are listed below. If the task status is other than a wait state (TTS_WAI or TTS_WAS), tskwait is indeterminate.

- TTW_SLP (0x0001) Kept waiting by slp_tsk or tslp_tsk
- TTW_DLY (0x0002) Kept waiting by dly_tsk
- TTW_SEM (0x0004) Kept waiting by wai_sem or twai_sem
- TTW_FLG (0x0008) Kept waiting by wai_flg or twai_flg
- TTW_SDTQ(0x0010) Kept waiting by snd_dtq or tsnd_dtq
- TTW_RDTQ(0x0020) Kept waiting by rcv_dtq or trcv_dtq
- TTW_MBX (0x0040) Kept waiting by rcv_mbx or trcv_mbx
- TTW_MPF (0x2000) Kept waiting by get_mpf or tget_mpf
- TTW_VSDTQ (0x4000) Kept waiting by vsnd_dtq or vtsnd_dtq[33]
- TTW_VRDTQ(0x8000) Kept waiting by vrcv_dtq or vtrcv_dtq

If this service call is to be issued from task context, use ref_tst; if issued from non-task context, use iref_tst.

[[Example program statement]]

<<Example statement in C language>>

```
    #include <itron.h>
    #include <kernel.h>
    #include "kernel_id.h"
    void task()
    {
        T_RTST rtst;
        ER ercd;
                  :
        ercd = ref_tst( ID_main, &rtst );
                  :
    }
```

<<Example statement in assembly language>>

```
_refdata:       .blkb   4
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    PUSH.L      A1
    ref_tst     #ID_TASK2,#_refdata
          :
```

[33] TTW_VSDTQ and TTW_VRDTQ are the causes of wait outside the scope of μITRON 4.0 Specification.

5.2 Task Dependent Synchronization Function

Specifications of the task-dependent synchronization function are listed in below.

Table 5.3 Specifications of the Task Dependent Synchronization Function

No.	Item	Content
1	Maximum value of task wakeup request count	255
2	Maximum number of nested forcible task wait requests count	1

Table 5.4 List of Task Dependent Synchronization Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	slp_tsk	**[S][B]**	Puts task to sleep	O		O		O	
2	tslp_tsk	**[S]**	Puts task to sleep (with timeout)	O		O		O	
3	wup_tsk	**[S][B]**		O		O	O	O	
4	iwup_tsk	**[S][B]**			O	O	O	O	
5	can_wup	**[B]**		O		O	O	O	
6	ican_wup				O	O	O	O	
7	rel_wai	**[S][B]**	Releases task from waiting	O		O	O	O	
8	irel_wai	**[S][B]**			O	O	O	O	
9	sus_tsk	**[S][B]**	Suspends task	O		O	O	O	
10	isus_tsk				O	O	O	O	
11	rsm_tsk	**[S][B]**	Wakes up task	O		O	O	O	
12	irsm_tsk				O	O	O	O	
13	frsm_tsk	**[S]**	Cancels wakeup request	O		O	O	O	
14	ifrsm_tsk				O	O	O	O	
15	dly_tsk	**[S][B]**	Delays task	O		O		O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls
- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

slp_tsk Put task to sleep
tslp_tsk Put task to sleep (with timeout)

[[C Language API]]

```
ER ercd = slp_tsk();
ER ercd = tslp_tsk( TMO tmout );
```

- **Parameters**
  - `slp_tsk`
    None
  - `tslp_tsk`

TMO	tmout	Timeout value

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr100.inc
slp_tsk
tslp_tsk TMO
```

- **Parameters**

TMO	Timeout value

- **Register contents after service call is issued**

`tslp_tsk`

Register name	Content after service call is issued
R0	Error code
R6R4	Timeout value

`slp_tsk`

Register name	Content after service call is issued
R0	Error code

[[Error code]]

E_TMOUT	Timeout
E_RLWAI	Forced release from waiting

[[Functional description]]

This service call places the issuing task itself from RUNNING state into sleeping wait state. The task placed into WAITING state by execution of this service call is released from the wait state in the following cases:

- ◆ **When a task wakeup service call is issued from another task or an interrupt**
  The error code returned in this case is E_OK.
- ◆ **When a forcible awaking service call is issued from another task or an interrupt**
  The error code returned in this case is E_RLWAI.
- ◆ **When the first time tick occurred after tmout elapsed (for tslp_tsk)**
  The error code returned in this case is E_TMOUT.

If the task receives sus_tsk issued from another task while it has been placed into WAITING state by this service call, it goes to WAITING-SUSPENDED state. In this case, even when the task is released from WAITING state by a task wakeup service call, it still remains in SUSPENDED state, and its execution cannot be resumed until rsm_tsk is issued.

The service call tslp_tsk may be used to place the issuing task into sleeping state for a given length of time by specifying tmout in a parameter to it. The parameter tmout is expressed in ms units. For example, if this service call is written as tslp_tsk(10);, then the issuing task is placed from RUNNING state into WAITING state for a period of 10 ms. If specified as tmout =TMO_FEVR(–1), the task will be kept waiting perpetually, with the service call operating the same way as slp_tsk.

The values specified for tmout must be within (0x7FFFFFFF-time tick value). If any value exceeding this limit is specified, operation of the service call cannot be guaranteed.

This service call can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include “kernel_id.h”
  void task()
  {
                :
      if( slp_tsk() != E_OK )
          error(“Forced wakeup\n”);
                :
      if( tslp_tsk( 10 ) == E_TMOUT )
          error(“time out\n”);
                :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    slp_tsk
          :
    PUSHM       R6R4
    tslp_tsk    #TMO_FEVR
          :
    PUSHM       R6R4
    tslp_tsk    #100
          :
```

wup_tsk Wakeup task
iwup_tsk Wakeup task (handler only)

[[C Language API]]

```
ER ercd = wup_tsk( ID tskid );
ER ercd = iwup_tsk( ID tskid );
```

- **Parameters**

ID	tskid	ID number of the target task

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr100.inc
wup_tsk  TSKID
iwup_tsk  TSKID
```

- **Parameters**

TSKID	ID number of the target task

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	ID number of the target task

[[Error code]]

E_OBJ	Object status invalid(task indicated by tskid is an inactive state)
E_QOVR	Queuing overflow

[[Functional description]]

If the task specified by tskid has been placed into WAITING state by slp_tsk or tslp_tsk, this service call wakes up the task from WAITING state to place it into READY or RUNNING state. Or if the task specified by tskid is in WAITING-SUSPENDED state, this service call awakes the task from only the sleeping state so that the task goes to SUSPENDED state.

If a wakeup request is issued while the target task remains in DORMANT state, the error code E_OBJ is returned to the service call issuing task. If TSK_SELF is specified for tskid, it means specifying the issuing task itself. If TSK_SELF is specified for tskid in non-task context, operation of the service call cannot be guaranteed.

If this service call is issued to a task that has not been placed in WAITING state or in WAITING-SUSPENDED state by execution of slp_tsk or tslp_tsk, the wakeup request is accumulated. More specifically, the wakeup request count for the target task to be awakened is incremented by 1, in which way wakeup requests are accumulated.

The maximum value of the wakeup request count is 255. If while the wakeup request count = 255 a new wakeup request is generated exceeding this limit, the error code E_QOVR is returned to the task that issued the service call, with the wakeup request count left intact.

If this service call is to be issued from task context, use wup_tsk; if issued from non-task context, use iwup_tsk.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
                :
     if( wup_tsk( ID_main ) != E_OK )
         printf("Can't wakeup main()\n");
                :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB       task
task:
          :
    PUSH.W     R2
    wup_tsk    #ID_TASK1
          :
```

can_wup Cancel wakeup request
ican_wup Cancel wakeup request (handler only)

[[C Language API]]

```
ER_UINT wupcnt = can_wup( ID tskid );
ER_UINT wupcnt = ican_wup( ID tskid );
```

- **Parameters**

ID	tskid	ID number of the target task

- **Return Parameters**

ER_UINT	wupcnt > 0	Canceled wakeup request count
	wupcnt <0	Error code

[[Assembly language API]]

```
.include mr100.inc
can_wup  TSKID
ican_wup  TSKID
```

- **Parameters**

TSKID	ID number of the target task

- **Register contents after service call is issued**

Register name	Content after service call is issued
R2R0	Error code,Canceled wakeup request count

[[Error code]]

E_OBJ	Object status invalid(task indicated by tskid is an inactive state)

[[Functional description]]

This service call clears the wakeup request count of the target task indicated by tskid to 0. This means that because the target task was in either WAITING state nor WAITING-SUSPENDED state when an attempt was made to wake it up by wup_tsk or iwup_tsk before this service call was issued, the attempt resulted in only accumulating wakeup requests and this service call clears all of those accumulated wakeup requests.

Furthermore, the wakeup request count before being cleared to 0 by this service call, i.e., the number of wakeup requests that were issued in vain (wupcnt) is returned to the issuing task. If a wakeup request is issued while the target task is in DORMANT state, the error code E_OBJ is returned. If TSK_SELF is specified for tskid, it means specifying the issuing task itself. If TSK_SELF is specified for tskid in non-task context, operation of this service call cannot be guaranteed.

If this service call is to be issued from task context, use can_wup; if issued from non-task context, use ican_wup.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
      ER_UINT wupcnt;
                :
      wupcnt = can_wup(ID_main);
      if( wup_cnt  > 0 )
          printf("wupcnt = %d\n",wupcnt);
                :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB       task
task:
          :
    can_wup    #ID_TASK3
          :
```

rel_wai Release task from waiting
irel_wai Release task from waiting (handler only)

[[C Language API]]

```
ER ercd = rel_wai( ID tskid );
ER ercd = irel_wai( ID tskid );
```

- **Parameters**

ID	tskid	ID number of the target task

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr100.inc
rel_wai  TSKID
irel_wai  TSKID
```

- **Parameters**

TSKID	ID number of the target task

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	ID number of the target task

[[Error code]]

E_OBJ	Object status invalid(task indicated by tskid is not an wait state)

[[Functional description]]

This service call forcibly release the task indicated by tskid from waiting (except SUSPENDED state) to place it into READY or RUNNING state. The forcibly released task returns the error code E_RLWAI. If the target task has been enqueued in some waiting queue, the task is dequeued from it by execution of this service call.

If this service call is issued to a task in WAITING-SUSPENDED state, the target task is released from WAITING state and goes to SUSPENDED state.[34]

If the target task is not in WAITING state, the error code E_OBJ is returned. This service call forbids specifying the issuing task itself for tskid.

If this service call is to be issued from task context, use rel_wai; if issued from non-task context, use irel_wai.

[34] This means that tasks cannot be resumed from SUSPENDED state by this service call. Only the rsm_tsk, irsm_tsk, frsm_tsk, and ifrsm_tsk service calls can release them from SUSPENDED state.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
               :
      if( rel_wai( ID_main ) != E_OK )
          error("Can't rel_wai main()\n");
           :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    rel_wai     #ID_TASK2
          :
```

sus_tsk Suspend task
isus_tsk Suspend task (handler only)

[[C Language API]]

```
ER ercd = sus_tsk( ID tskid );
ER ercd = isus_tsk( ID tskid );
```

- **Parameters**

ID	tskid	ID number of the target task

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr100.inc
sus_tsk  TSKID
isus_tsk  TSKID
```

- **Parameters**

TSKID	ID number of the target task

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	ID number of the target task

[[Error code]]

E_OBJ	Object status invalid(task indicated by tskid is an inactive state)
E_QOVR	Queuing overflow

[[Functional description]]

This service call aborts execution of the task indicated by tskid and places it into SUSPENDED state. Tasks are resumed from this SUSPENDED state by the rsm_tsk, irsm_tsk, frsm_tsk, or ifrsm_tsk service call. If the task indicated by tskid is in DORMANT state, it returns the error code E_OBJ as a return value for the service call.

The maximum number of suspension requests by this service call that can be nested is 1. If this service call is issued to a task which is already in SUSPENDED state, the error code E_QOVR is returned.

This service call forbids specifying the issuing task itself for tskid.

If this service call is to be issued from task context, use sus_tsk; if issued from non-task context, use isus_tsk.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
                :
      if( sus_tsk( ID_main ) != E_OK )
          printf("Can't suspend task main()\n");
                :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    sus_tsk     #ID_TASK2
          :
```

rsm_tsk	**Resume suspended task**
irsm_tsk	**Resume suspended task(handler only)**
frsm_tsk	**Forcibly resume suspended task**
ifrsm_tsk	**Forcibly resume suspended task(handler only)**

[[C Language API]]

```
ER ercd = rsm_tsk( ID tskid );
ER ercd = irsm_tsk( ID tskid );
ER ercd = frsm_tsk( ID tskid );
ER ercd = ifrsm_tsk( ID tskid );
```

● **Parameters**

ID	tskid	ID number of the target task

● **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr100.inc
rsm_tsk  TSKID
irsm_tsk  TSKID
frsm_tsk TSKID
ifrsm_tsk TSKID
```

● **Parameters**

TSKID	ID number of the target task

● **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	ID number of the target task

[[Error code]]

E_OBJ	Object status invalid(task indicated by tskid is not a forcible wait state)

[[Functional description]]

If the task indicated by tskid has been aborted by sus_tsk, this service call resumes the target task from SUSPENDED state. In this case, the target task is linked to behind the tail of the ready queue. In the case of frsm_tsk and ifrsm_tsk, the task is forcibly resumed from SUSPENDED state.

If a request is issued while the target task is not in SUSPENDED state (including DORMANT state), the error code E_OBJ is returned to the service call issuing task.

The rsm_tsk, irsm_tsk, frsm_tsk, and ifrsm_tsk service calls each operate the same way, because the maximum number of forcible wait requests that can be nested is 1.

If this service call is to be issued from task context, use rsm_tsk/frsm_tsk; if issued from non-task context, use irsm_tsk/ifrsm_tsk.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task1()
  {
                :
      if( rsm_tsk( ID_main ) != E_OK )
          printf("Can't resume main()\n");
                :
                :
      if(frsm_tsk( ID_task2 ) != E_OK )
          printf("Can't forced resume task2()\n");
                :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    rsm_tsk     #ID_TASK2
          :
    PUSH.W      R2
    frsm_tsk    #ID_TASK1
          :
```

dly_tsk Delay task

[[C Language API]]

```
ER ercd = dly_tsk(RELTIM dlytim);
```

- **Parameters**

RELTIM	dlytim	Delay time

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr100.inc
dly_tsk  RELTIM
```

- **Parameters**

RELTIM	Delay time

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R6R4	Delay time

[[Error code]]

E_RLWAI	Forced release from waiting

[[Functional description]]

This service call temporarily stops execution of the issuing task itself for a duration of time specified by dlytim to place the task from RUNNING state into WAITING state. In this case, the task is released from the WAITING state at the first time tick after the time specified by dlytim has elapsed. Therefore, if specified dlytim = 0, the task is placed into WAITING state briefly and then released from the WAITING state at the first time tick.

The task placed into WAITING state by invocation of this service call is released from the WAITING state in the following cases. Note that when released from WAITING state, the task that issued the service call is removed from the timeout waiting queue and linked to a ready queue.

- ◆ **When the first time tick occurred after dlytim elapsed**
  The error code returned in this case is E_OK.
- ◆ **When the rel_wai or irel_wai service call is issued before dlytim elapses**
  The error code returned in this case is E_RLWAI.

Note that even when the wup_tsk or iwup_tsk service call is issued during the delay time, the task is not released from WAITING state.

The delay time dlytim is expressed in ms units. Therefore, if specified as dly_tsk(50);, the issuing task is placed from RUNNING state into a delayed wait state for a period of 50 ms.

The values specified for dlytim must be within (0x7FFFFFFF- time tick value). If any value exceeding this limit is specified, the service call may not operate correctly.

This service call can be issued only from task context. It cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
                :
      if( dly_tsk() != E_OK )
          error("Forced wakeup\n");
                :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    PUSHM       R6R4
    dly_tsk     #500
          :
```

5.3 Synchronization & Communication Function (Semaphore)

Specifications of the semaphore function of MR100 are listed in Table 5.5.

Table 5.5 Specifications of the Semaphore Function

No.	Item	Content	
1	Semaphore ID	1-255	
2	Maximum number of resources	1-65535	
3	Semaphore attribute	TA_FIFO:	Tasks enqueued in order of FIFO
		TA_TPRI:	Tasks enqueued in order of priority

Table 5.6 List of Semaphore Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	sig_sem	**[S][B]**	Releases semaphore resource	O		O	O	O	
2	isig_sem	**[S][B]**			O	O	O	O	
3	wai_sem	**[S][B]**	Acquires semaphore resource	O		O		O	
4	pol_sem	**[S][B]**	Acquires semaphore resource(polling)	O		O	O	O	
5	ipol_sem				O	O	O	O	
6	twai_sem	**[S]**	Acquires semaphore resource(with timeout)	O		O		O	
7	ref_sem		References semaphore status	O		O	O	O	
8	iref_sem				O	O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls
- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

sig_sem Release semaphore resource
isig_sem Release semaphore resource (handler only)

[[C Language API]]

```
ER ercd = sig_sem( ID semid );
ER ercd = isig_sem( ID semid );
```

- **Parameters**

ID	semid	Semaphore ID number to which returned

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr100.inc
sig_sem SEMID
isig_sem SEMID
```

- **Parameters**

SEMID	Semaphore ID number to which returned

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	Semaphore ID number to which returned

[[Error code]]

E_QOVR	Queuing overflow

[[Functional description]]

This service call releases one resource to the semaphore indicated by semid.

If tasks are enqueued in a waiting queue for the target semaphore, the task at the top of the queue is placed into READY state. Conversely, if no tasks are enqueued in that waiting queue, the semaphore resource count is incremented by 1. If an attempt is made to return resources (sig_sem or isig_sem service call) causing the semaphore resource count value to exceed the maximum value specified in a configuration file (maxsem), the error code E_QOVR is returned to the service call issuing task, with the semaphore count value left intact.

If this service call is to be issued from task context, use sig_sem; if issued from non-task context, use isig_sem.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
                 :
      if( sig_sem( ID_sem ) == E_QOVR )
        error("Overflow\n");
                 :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    sig_sem     #ID_SEM2
          :
```

wai_sem	**Acquire semaphore resource**
pol_sem	**Acquire semaphore resource (polling)**
ipol_sem	**Acquire semaphore resource (polling, handler only)**
twai_sem	**Acquire semaphore resource(with timeout)**

[[C Language API]]

```
ER ercd = wai_sem( ID semid );
ER ercd = pol_sem( ID semid );
ER ercd = ipol_sem( ID semid );
ER ercd = twai_sem( ID semid, TMO tmout );
```

● Parameters

ID	semid	Semaphore ID number to be acquired
TMO	tmout	Timeout value (for twai_sem)

● Return Parameters

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr100.inc
wai_sem SEMID
pol_sem SEMID
ipol_sem SEMID
twai_sem SEMID,TMO
```

● Parameters

SEMID	Semaphore ID number to be acquired
TMO	Timeout value(twai_sem)

● Register contents after service call is issued

wai_sem,pol_sem,ipol_sem

Register name	Content after service call is issued
R0	Error code
R2	Semaphore ID number to be acquired

twai_sem

Register name	Content after service call is issued
R0	Error code
R2	Semaphore ID number to be acquired
R6R4	Timeout value

[[Error code]]

E_RLWAI	Forced release from waiting
E_TMOUT	Polling failure or timeout

[[Functional description]]

This service call acquires one semaphore resource from the semaphore indicated by semid.

If the semaphore resource count is equal to or greater than 1, the semaphore resource count is decremented by 1, and the service call issuing task continues execution. On the other hand, if the semaphore count value is 0, the wai_sem or twai_sem service call invoking task is enqueued in a waiting queue for that semaphore. If the attribute of the semaphore semid is TA_TFIFO, the task is enqueued in order of FIFO; if TA_TPRI, the task is enqueued in order of priority. For the pol_sem and ipol_sem service calls, the task returns immediately and responds to the call with the error code E_TMOUT.

For the twai_sem service call, specify a wait time for tmout in ms units. The values specified for tmout must be within (0x7FFFFFFF-time tick value). If any value exceeding this limit is specified, operation of the service call cannot be guaranteed. If TMO_POL=0 is specified for tmout, it means specifying 0 as a timeout value, in which case the service call operates the same way as pol_sem. Furthermore, if specified as tmout=TMO_FEVR(–1), it means specifying an infinite wait, in which case the service call operates the same way as wai_sem.

The task placed into WAITING state by execution of the wai_sem or twai_sem service call is released from the WAITING state in the following cases:

- ◆ **When the sig_sem or isig_sem service call is issued before the tmout time elapses, with task-awaking conditions thereby satisfied**
  The error code returned in this case is E_OK.
- ◆ **When the first time tick occurred after tmout elapsed while task-awaking conditions remain unsatisfied**
  The error code returned in this case is E_TMOUT.
- ◆ **When the task is forcibly released from WAITING state by the rel_wai or irel_wai service call issued from another task or a handler**
  The error code returned in this case is E_RLWAI.

If this service call is to be issued from task context, use wai_sem, twai_sem, or pol_sem; ; if issued from non-task context, use ipol_sem.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
                :
     if( wai_sem( ID_sem ) != E_OK )
         printf("Forced wakeup\n");
                :
     if( pol_sem( ID_sem ) != E_OK )
         printf("Timeout\n");
                :
       if( twai_sem( ID_sem, 10 ) != E_OK )
           printf("Forced wakeup or Timeout"n");
                :
  }
```

<<Example statement in assembly language>>

```
     .include mr100.inc
     .GLB        task
task:
           :
     PUSH.W      R2
     pol_sem     #ID_SEM1
           :
     PUSH.W      R2
     wai_sem     #ID_SEM2
           :
     PUSH.W      R2
     PUSH.L      R6R4
     twai_sem    #ID_SEM3,300
           :
```

ref_sem Reference semaphore status
iref_sem Reference semaphore status (handler only)

[[C Language API]]

```
ER ercd = ref_sem( ID semid, T_RSEM *pk_rsem );
ER ercd = iref_sem( ID semid, T_RSEM *pk_rsem );
```

- **Parameters**

ID	semid	ID number of the target semaphore
T_RSEM	*pk_rsem	Pointer to the packet to which semaphore status is returned

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)
T_RSEM	*pk_rsem	Pointer to the packet to which semaphore status is returned

Contents of pk_rsem

```
typedef    struct    t_rsem{
    ID      wtskid    +0    2    ID number of the task at the head of the semaphore's wait queue
    UINT    semcnt    +2    4    Current semaphore resource count
} T_RSEM;
```

[[Assembly language API]]

```
.include mr100.inc
ref_sem SEMID, PK_RSEM
iref_sem SEMID, PK_RSEM
```

- **Parameters**

SEMID	ID number of the target semaphore
PK_RSEM	Pointer to the packet to which semaphore status is returned

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	ID number of the target semaphore
A1	Pointer to the packet to which semaphore status is returned

[[Error code]]

None

[[Functional description]]

This service call returns various statuses of the semaphore indicated by semid.

◆ **wtskid**

Returned to wtskid is the ID number of the task at the head of the semaphore's wait queue (the next task to be dequeued). If no tasks are kept waiting, TSK_NONE is returned.

◆ **semcnt**

Returned to semcnt is the current semaphore resource count.

If this service call is to be issued from task context, use ref_sem; if issued from non-task context, use iref_sem.

[[Example program statement]]

<<Example statement in C language>>

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
    T_RSEM rsem;
    ER ercd;
        :
    ercd = ref_sem( ID_sem1, &rsem );
        :
}
```

<<Example statement in assembly language>>

```
_ refsem:      .blkb   6
    .include mr100.inc
    .GLB       task
task:
          :
        PUSH.W  R2
        PUSH.L  A1
        ref_sem #ID_SEM1,#_refsem
          :
```

5.4 Synchronization & Communication Function (Eventflag)

Specifications of the eventflag function of MR100 are listed in Table 5.7.

Table 5.7 Specifications of the Eventflag Function

No.	Item	Content	
1	Event0flag ID	1-255	
2	Number of bits comprising eventflag	32 bits	
3	Eventflag attribute	TA_TFIFO:	Waiting tasks enqueued in order of FIFO
		TA_TPRI:	Waiting tasks enqueued in order of priority
		TA_WSGL:	Multiple tasks cannot be kept waiting
		TA_WMUL:	Multiple tasks can be kept waiting
		TA_CLR:	Bit pattern cleared when waiting task is released

Table 5.8 List of Eventflag Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	set_flg	**[S][B]**	Sets eventflag	O		O	O	O	
2	iset_flg	**[S][B]**			O	O	O	O	
3	clr_flg	**[S][B]**	Clears eventflag	O		O	O	O	
4	iclr_flg				O	O	O	O	
5	wai_flg	**[S][B]**	Waits for eventflag	O		O		O	
6	pol_flg	**[S][B]**	Waits for eventflag (polling)	O		O	O	O	
7	ipol_flg	**[S]**			O	O	O	O	
8	twai_flg	**[S]**	Waits for eventflag (with timeout)	O		O		O	
9	ref_flg		References eventflag status	O		O	O	O	
10	iref_flg				O	O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls
- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

set_flg	Set eventflag
iset_flg	Set eventflag (handler only)

[[C Language API]]

```
ER ercd = set_flg( ID flgid, FLGPTN setptn );
ER ercd = iset_flg( ID flgid, FLGPTN setptn );
```

- **Parameters**

ID	flgid	ID number of the eventflag to be set
FLGPTN	setptn	Bit pattern to be set

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
set_flg FLGID,SETPTN
iset_flg FLGID,SETPTN
```

- **Parameters**

FLGID	ID number of the eventflag to be set
SETPTN	Bit pattern to be set

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	Eventflag ID number
A1	Bit pattern to be set

[[Error code]]

None

[[Functional description]]

Of the 32-bit eventflag indicated by flgid, this service call sets the bits indicated by setptn. In other words, the value of the eventflag indicated by flgid is OR'd with setptn. If the alteration of the eventflag value results in task-awaking conditions for a task that has been kept waiting for the eventflag by the wai_flg or twai_flg service call becoming satisfied, the task is released from WAITING state and placed into READY or RUNNING state.

Task-awaking conditions are evaluated sequentially beginning with the top of the waiting queue. If TA_WMUL is specified as an eventflag attribute, multiple tasks kept waiting for the eventflag can be released from WAITING state at the same time by one set_flg or iset_flg service call issued. Furthermore, if TA_CLR is specified for the attribute of the target eventflag, all bit patterns of the eventflag are cleared, with which processing of the service call is terminated.

If all bits specified in setptn are 0, no operation will be performed for the target eventflag, in which case no errors are assumed, however.

If this service call is to be issued from task context, use set_flg; if issued from non-task context, use iset_flg.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include “kernel_id.h”
  void task(void)
  {
                :
    set_flg( ID_flg,(FLGPTN)0xff000000 );
                :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    PUSH.L      A1
    set_flg     #ID_FLG3,#0ff000000H
          :
```

clr_flg　　Clear eventflag
iclr_flg　　Clear eventflag (handler only)

[[C Language API]]

```
ER ercd = clr_flg( ID flgid, FLGPTN clrptn );
ER ercd = iclr_flg( ID flgid, FLGPTN clrptn );
```

- **Parameters**

ID	flgid	ID number of the eventflag to be cleared
FLGPTN	clrptn	Bit pattern to be cleared

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
clr_flg FLGID,CLRPTN
iclr_flg FLGID,CLRPTN
```

- **Parameters**

FLGID	ID number of the eventflag to be cleared
CLRPTN	Bit pattern to be cleared

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	ID number of the eventflag to be cleared
A1	Bit pattern to be cleared

[[Error code]]

None

[[Functional description]]

Of the 32-bit eventflag indicated by flgid, this service call clears the bits whose corresponding values in clrptn are 0. In other words, the eventflag bit pattern indicated by flgid is updated by AND'ing it with clrptn. If all bits specified in clrptn are 1, no operation will be performed for the target eventflag, in which case no errors are assumed, however.

If this service call is to be issued from task context, use clr_flg; if issued from non-task context, use iclr_flg.[35]

[35] When iclr_flg is issued from interruption generated during set_flg or iset_flg service call execution, the indivisibility of a service call is not guaranteed. That is, if there are two or more tasks which are waiting by the same bit pattern in the waiting queue, some tasks are released and some tasks are not released by the timing of interruption generating.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task(void)
  {
                :
    clr_flg( ID_flg,(FLGPTN) 0xf0f0f0f0);
                :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    PUSH.L      A1
    clr_flg     #ID_FLG1,#0f0f0f0f0H
          :
```

wai_flg Wait for eventflag
pol_flg Wait for eventflag(polling)
ipol_flg Wait for eventflag(polling, handler only)
twai_flg Wait for eventflag(with timeout)

[[C Language API]]

```
ER ercd = wai_flg( ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn );
ER ercd = pol_flg( ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn );
ER ercd = ipol_flg( ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn );
ER ercd = twai_flg( ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn,
                    TMO tmout );
```

● **Parameters**

ID	flgid	ID number of the eventflag waited for
FLGPTN	waiptn	Wait bit pattern
MODE	wfmode	Wait mode
FLGPTN	*p_flgptn	Pointer to the area to which bit pattern is returned when released from wait
TMO	tmout	Timeout value (for twai_flg)

● **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code
FLGPTN	*p_flgptn	Pointer to the area to which bit pattern is returned when released from wait

[[Assembly language API]]

```
.include mr100.inc
wai_flg  FLGID, WAIPTN, WFMODE
pol_flg  FLGID, WAIPTN, WFMODE
ipol_flg FLGID, WAIPTN, WFMODE
twai_flg FLGID, WAIPTN, WFMODE, TMO
```

● **Parameters**

FLGID	ID number of the eventflag waited for
WAIPTN	Wait bit pattern
WFMODE	Wait mode
TMO	Timeout value (for twai_flg)

● **Register contents after service call is issued**

wai_sem,pol_sem,ipol_sem

Register name	Content after service call is issued
R0	Error code
R3R1	bit pattern is returned when released from wait
R2	ID number of the eventflag waited for
A1	Wait bit pattern

twai_sem

Register name	Content after service call is issued
R0	Error code
R3R1	bit pattern is returned when released from wait
R2	ID number of the eventflag waited for
R6R4	Timeout value
A1	Wait bit pattern

[[Error code]]

E_RLWAI	Forced release from waiting
E_TMOUT	Polling failure or timeout or timed out
E_ILUSE	Service call improperly used (Tasks present waiting for TA_WSGL attribute eventflag)

[[Functional description]]

This service call waits until the eventflag indicated by flgid has its bits specified by waiptn set according to task-awaking conditions indicated by wfmode. Returned to the area pointed to by p_flgptn is the eventflag bit pattern at the time the task is released from WAITING state.
If the target eventflag has the TA_WSGL attribute and there are already other tasks waiting for the eventflag, the error code E_ILUSE is returned.
If task-awaking conditions have already been met when this service call is invoked, the task returns immediately and responds to the call with E_OK. If task-awaking conditions are not met and the invoked service call is wai_flg or twai_flg, the task is enqueued in an eventflag waiting queue. In that case, if the attribute of the specified eventflag is TA_TFIFO, the task is enqueued in order of FIFO; if TA_TPRI, the task is enqueued in order of priority. For the pol_flg and ipol_flg service calls, the task returns immediately and responds to the call with the error code E_TMOUT.
For the twai_flg service call, specify a wait time for tmout in ms units. The values specified for tmout must be within (0x7FFFFFFF-time tick value). If any value exceeding this limit is specified, the service call may not operate correctly. If TMO_POL=0 is specified for tmout, it means specifying 0 as a timeout value, in which case the service call operates the same way as pol_flg. Furthermore, if specified as tmout=TMO_FEVR(–1), it means specifying an infinite wait, in which case the service call operates the same way as wai_flg.

The task placed into a wait state by execution of the wai_flg or twai_flg service call is released from WAITING state in the following cases:

- ◆ **When task-awaking conditions are met before the tmout time elapses**
  The error code returned in this case is E_OK.
- ◆ **When the first time tick occurred after tmout elapsed while task-awaking conditions remain unsatisfied**
  The error code returned in this case is E_TMOUT.
- ◆ **When the task is forcibly released from WAITING state by the rel_wai or irel_wai service call issued from another task or a handler**
  The error code returned in this case is E_RLWAI.

The following shows how wfmode is specified and the meaning of each mode.

wfmdoe (wait mode)	Meaning
TWF_ANDW	Wait until all bits specified by waiptn are set (wait for the bits AND'ed)
TWF_ORW	Wait until one of the bits specified by waiptn is set (wait for the bits OR'ed)

If this service call is to be issued from task context, use wai_flg,twai_flg,pol_flg; if issued from non-task context, use ipol_flg.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include “kernel_id.h”
  void task()
  {
    UINT flgptn;
                :
      if(wai_flg(ID_flg2, (FLGPTN)0x00000ff0, TWF_ANDW, &flgptn)!=E_OK)
        error(“Wait Released\n”);
                :
                :
      if(pol_flg(ID_flg2, (FLGPTN)0x00000ff0, TWF_ORW, &flgptn)!=E_OK)
        printf(“Not set EventFlag\n”);
                :
                :
      if( twai_flg(ID_flg2, (FLGPTN)0x00000ff0, TWF_ANDW, &flgptn, 5) != E_OK )
        error(“Wait Released\n”);
                :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    PUSH.L      A1
    wai_flg     #ID_FLG1,#00000003H,#TWF_ANDW
          :
    PUSH.W      R2
    PUSH.L      A1
    pol_flg     #ID_FLG2,#00000008H,#TWF_ORW
          :
    PUSH.W      R2
    PUSH.L      A1
    PUSHM       R6R4
    wai_flg     #ID_FLG3,#00000003H,#TWF_ANDW,20
          :
```

ref_flg	Reference eventflag status
iref_flg	Reference eventflag status (handler only)

[[C Language API]]

```
ER ercd = ref_flg( ID flgid, T_RFLG *pk_rflg );
ER ercd = iref_flg( ID flgid, T_RFLG *pk_rflg );
```

● Parameters

ID	flgid	ID number of the target eventflag
T_RFLG	*pk_rflg	Pointer to the packet to which eventflag status is returned

● Return Parameters

ER	ercd	Terminated normally (E_OK)
T_RFLG	*pk_rflg	Pointer to the packet to which eventflag status is returned

Contents of pk_rflg

```
typedef     struct    t_rflg{
     ID          wtskid      +0    2     Reception waiting task ID
     FLGPTN      flgptn      +2    4     Current eventflag bit pattern
} T_RFLG;
```

[[Assembly language API]]

```
.include mr100.inc
ref_flg  FLGID, PK_RFLG
iref_flg FLGID, PK_RFLG
```

● Parameters

FLGID	ID number of the target eventflag
PK_RFLG	Pointer to the packet to which eventflag status is returned

● Register contents after service call is issued

Register name	Content after service call is issued
R0	Error code
R2	ID number of the target eventflag
A1	Pointer to the packet to which eventflag status is returned

[[Error code]]

None

[[Functional description]]

This service call returns various statuses of the eventflag indicated by flgid.

◆ **wtskid**

Returned to wtskid is the ID number of the task at the top of a waiting queue (the next task to be dequeued). If no tasks are kept waiting, TSK_NONE is returned.

◆ **flgptn**

Returned to flgptn is the current eventflag bit pattern.

If this service call is to be issued from task context, use ref_flg; if issued from non-task context, use iref_flg.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
       T_RFLG rflg;
       ER ercd;
           :
       ercd = ref_flg( ID_FLG1, &rflg );
           :
  }
```

<<Example statement in assembly language>>

```
_ refflg:       .blkb   6
    .include mr100.inc
    .GLB        task
task:
           :
        PUSH.W  R2
        PUSH.L  A1
        ref_flg #ID_FLG1,#_refflg
           :
```

5.5 Synchronization & Communication Function (Data Queue)

Specifications of the data queue function of MR100 are listed in Table 5.9.

Table 5.9 Specifications of the Data Queue Function

No.	Item	Content	
1	Data queue ID	1-255	
2	Capacity (data bytes) in data queue area	0-8191	
3	Data size	32 bits	
4	Data queue attribute	TA_TFIFO:	Waiting tasks enqueued in order of FIFO
		TA_TPRI:	Waiting tasks enqueued in order of priority

Table 5.10 List of Dataqueue Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	snd_dtq	**[S]**	Sends to data queue	O		O		O	
2	psnd_dtq	**[S]**	Sends to data queue (polling)		O	O	O	O	
3	ipsnd_dtq	**[S]**		O		O	O	O	
4	tsnd_dtq	**[S]**	Sends to data queue (with timeout)		O	O		O	
5	fsnd_dtq	**[S]**	Forced sends to data queue	O		O	O	O	
6	ifsnd_dtq	**[S]**		O		O	O	O	
7	rcv_dtq	**[S]**	Receives from data queue		O	O		O	
8	prcv_dtq	**[S]**	Receives from data queue (polling)	O		O	O	O	
9	iprcv_dtq			O		O	O	O	
10	trcv_dtq	**[S]**	Receives from data queue (with timeout)		O	O		O	
11	ref_dtq		References data queue status	O		O	O	O	
12	iref_dtq				O	O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls
- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

snd_dtq	**Send to data queue**
psnd_dtq	**Send to data queue (polling)**
ipsnd_dtq	**Send to data queue (polling, handler only)**
tsnd_dtq	**Send to data queue (with timeout)**
fsnd_dtq	**Forced send to data queue**
ifsnd_dtq	**Forced send to data queue (handler only)**

[[C Language API]]

```
ER ercd = snd_dtq( ID dtqid, VP_INT data );
ER ercd = psnd_dtq( ID dtqid, VP_INT data );
ER ercd = ipsnd_dtq( ID dtqid, VP_INT data );
ER ercd = tsnd_dtq( ID dtqid, VP_INT data, TMO tmout );
ER ercd = fsnd_dtq( ID dtqid, VP_INT data );
ER ercd = ifsnd_dtq( ID dtqid, VP_INT data );
```

● Parameters

ID	dtqid	ID number of the data queue to which transmitted
TMO	tmout	Timeout value(tsnd_dtq)
VP_INT	data	Data to be transmitted

● Return Parameters

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr100.inc
snd_dtq   DTQID, DTQDATA
isnd_dtq  DTQID, DTQDATA
psnd_dtq  DTQID, DTQDATA
ipsnd_dtq DTQID, DTQDATA
tsnd_dtq  DTQID, DTQDATA,TMO
fsnd_dtq  DTQID, DTQDATA
ifsnd_dtq DTQID, DTQDATA
```

● Parameters

DTQID	ID number of the data queue to which transmitted
DTQDATA	Data to be transmitted
TMO	Timeout value (tsnd_dtq)

● Register contents after service call is issued

snd_dtq,psnd_dtq,ipsnd_dtq,fsnd_dtq,ifsnd_dtq

Register name	Content after service call is issued
R0	Error code
R3R1	Data to be transmitted
R2	ID number of the data queue to which transmitted

tsnd_dtq

Register name	Content after service call is issued
R0	Error code
R3R1	Data to be transmitted
R2	ID number of the data queue to which transmitted
R6R4	Timeout value

[[Error code]]

E_RLWAI	Forced release from waiting
E_TMOUT	Polling failure or timeout or timed out
E_ILUSE	Service call improperly used (fsnd_dtq or ifsnd_dtq is issued for a data queue whose dtqcnt = 0)
EV_RST	Released from WAITING state by clearing of the data queue area

[[Functional description]]

This service call sends the 4-byte data indicated by data to the data queue indicated by dtqid. If any task is kept waiting for reception in the target data queue, the data is not stored in the data queue and instead sent to the task at the top of the reception waiting queue, with which the task is released from the reception wait state.

On the other hand, if snd_dtq or tsnd_dtq is issued for a data queue that is full of data, the task that issued the service call goes from RUNNING state to a data transmission wait state, and is enqueued in transmission waiting queue, kept waiting for the data queue to become available. In that case, if the attribute of the specified data queue is TA_TFIFO, the task is enqueued in order of FIFO; if TA_TPRI, the task is enqueued in order of priority. For psnd_dtq and ipsnd_dtq, the task returns immediately and responds to the call with the error code E_TMOUT.

For the tsnd_dtq service call, specify a wait time for tmout in ms units. The values specified for tmout must be within (0x7FFFFFFF-time tick value). If any value exceeding this limit is specified, the service call may not operate correctly. If TMO_POL=0 is specified for tmout, it means specifying 0 as a timeout value, in which case the service call operates the same way as psnd_dtq. Furthermore, if specified as tmout=TMO_FEVR(–1), it means specifying an infinite wait, in which case the service call operates the same way as snd_dtq.

If there are no tasks waiting for reception, nor is the data queue area filled, the transmitted data is stored in the data queue.

The task placed into WAITING state by execution of the snd_dtq or tsnd_dtq service call is released from WAITING state in the following cases:

- **When the rcv_dtq, trcv_dtq, prcv_dtq, or iprcv_dtq service call is issued before the tmout time elapses, with task-awaking conditions thereby satisfied**
  The error code returned in this case is E_OK.
- **When the first time tick occurred after tmout elapsed while task-awaking conditions remain unsatisfied**
  The error code returned in this case is E_TMOUT.
- **When the task is forcibly released from WAITING state by the rel_wai or irel_wai service call issued from another task or a handler**
  The error code returned in this case is E_RLWAI.
- **When the target data queue being waited for is initialized by the vrst_dtq service call issued from another task**
  The error code returned in this case is EV_RST.

For fsnd_dtq and ifsnd_dtq, the data at the top of the data queue or the oldest data is removed, and the transmitted data is stored at the tail of the data queue. If the data queue area is not filled with data, fsnd_dtq and ifsnd_dtq operate the same way as snd_dtq. If dtqcnt = 0 ,there is no task in the wait queue and fsnd_dtq or ifsnd_dtq service call is issued, error code E_ILUSE will be returned.

If this service call is to be issued from task context, use snd_dtq,tsnd_dtq,psnd_dtq,fsnd_dtq; if issued from non-task context, use ipsnd_dtq,ifsnd_dtq.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  VP_INT data[10];
  void task(void)
  {
              :
      if( snd_dtq( ID_dtq, data[0]) == E_RLWAI ){
        error("Forced released\n");
    }
              :
      if( psnd_dtq( ID_dtq, data[1])== E_TMOUT ){
        error("Timeout\n");
    }
              :
      if( tsnd_dtq( ID_dtq, data[2], 10 ) != E_ TMOUT ){
        error("Timeout \n");
    }
              :
      if( fsnd_dtq( ID_dtq, data[3]) != E_OK ){
        error("error\n");
    }
              :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
_g_dtq: .LWORD  12345678H
task:
          :
    PUSH.W      R2
    PUSHM       R6R4,R3R1
    tsnd_dtq    #ID_DTQ1,_g_dtq,#100
          :
    PUSH.W      R2
    PUSHM       R3R1
    psnd_dtq    #ID_DTQ2,#0FFFFFFFFH
          :
    PUSH.W      R2
    PUSHM       R3R1
    fsnd_dtq    #ID_DTQ3,#0ABCDH
          :
```

rcv_dtq	**Receive from data queue**
prcv_dtq	**Receive from data queue (polling)**
iprcv_dtq	**Receive from data queue (polling, handler only)**
trcv_dtq	**Receive from data queue (with timeout)**

[[C Language API]]

```
ER ercd = rcv_dtq( ID dtqid, VP_INT *p_data );
ER ercd = prcv_dtq( ID dtqid, VP_INT *p_data );
ER ercd = iprcv_dtq( ID dtqid, VP_INT *p_data );
ER ercd = trcv_dtq( ID dtqid, VP_INT *p_data, TMO tmout );
```

● Parameters

ID	dtqid	ID number of the data queue from which to receive
TMO	tmout	Timeout value (trcv_dtq)
VP_INT	*p_data	Pointer to the start of the area in which received data is stored

● Return Parameters

ER	ercd	Terminated normally (E_OK) or error code
VP_INT	*p_data	Pointer to the start of the area in which received data is stored

[[Assembly language API]]

```
.include mr100.inc
rcv_dtq   DTQID
prcv_dtq  DTQID
iprcv_dtq DTQID
trcv_dtq  DTQID,TMO
```

● Parameters

DTQID	ID number of the data queue from which to receive
TMO	Timeout value (trcv_dtq)

● Register contents after service call is issued

rcv_dtq,prcv_dtq,iprcv_dtq

Register name	Content after service call is issued
R0	Error code
R3R1	Received data
R2	Data queue ID number

trcv_dtq

Register name	Content after service call is issued
R0	Error code
R3R1	Received data
R2	ID number of the data queue from which to receive
R6R4	Timeout value

[[Error code]]

E_RLWAI	Forced release from waiting
E_TMOUT	Polling failure or timeout or timed out

[[Functional description]]

This service call receives data from the data queue indicated by dtqid and stores the received data in the area pointed to by p_data. If data is present in the target data queue, the data at the top of the queue or the oldest data is received. This results in creating a free space in the data queue area, so that a task enqueued in a transmission waiting queue is released from WAITING state, and starts sending data to the data queue area.

If no data exist in the data queue and there is any task waiting to send data (i.e., data bytes in the data queue area = 0), data for the task at the top of the data transmission waiting queue is received. As a result, the task kept waiting to send that data is released from WAITING state.

On the other hand, if rcv_dtq or trcv_dtq is issued for the data queue which has no data stored in it, the task that issued the service call goes from RUNNING state to a data reception wait state, and is enqueued in a data reception waiting queue. At this time, the task is enqueued in order of FIFO. For the prcv_dtq and iprcv_dtq service calls, the task returns immediately and responds to the call with the error code E_TMOUT.

For the trcv_dtq service call, specify a wait time for tmout in ms units. The values specified for tmout must be within (0x7FFFFFFF-time tick value). If any value exceeding this limit is specified, the service call may not operate correctly. If TMO_POL=0 is specified for tmout, it means specifying 0 as a timeout value, in which case the service call operates the same way as prcv_dtq. Furthermore, if specified as tmout=TMO_FEVR(–1), it means specifying an infinite wait, in which case the service call operates the same way as rcv_dtq.

The task placed into a wait state by execution of the rcv_dtq or trcv_dtq service call is released from the wait state in the following cases:

- ◆ **When the rcv_dtq, trcv_dtq, prcv_dtq, or iprcv_dtq service call is issued before the tmout time elapses, with task-awaking conditions thereby satisfied**
  The error code returned in this case is E_OK.
- ◆ **When the first time tick occurred after tmout elapsed while task-awaking conditions remain unsatisfied**
  The error code returned in this case is E_TMOUT.
- ◆ **When the task is forcibly released from WAITING state by the rel_wai or irel_wai service call issued from another task or a handler**
  The error code returned in this case is E_RLWAI.

If this service call is to be issued from task context, use rcv_dtq,trcv_dtq,prcv_dtq; if issued from non-task context, use iprcv_dtq.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
                                              void task()
  {
    VP_INT data;
                :
      if( rcv_dtq( ID_dtq, &data ) != E_RLWAI )
          error("forced wakeup\n");
                :
      if( prcv_dtq( ID_dtq, &data ) != E_TMOUT )
          error("Timeout\n");
                :
      if( trcv_dtq( ID_dtq, &data, 10 ) != E_TMOUT )
          error("Timeout \n");
                :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    PUSHM       R6R4
    trcv_dtq    #ID_DTQ1,#TMO_POL
                   :
    PUSH.W      R2
    prcv_dtq    #ID_DTQ2
          :
    PUSH.W      R2
    rcv_dtq     #ID_DTQ2
          :
```

ref_dtq Reference data queue status
iref_dtq Reference data queue status (handler only)

[[C Language API]]

```
ER ercd = ref_dtq( ID dtqid, T_RDTQ *pk_rdtq );
ER ercd = iref_dtq( ID dtqid, T_RDTQ *pk_rdtq );
```

● Parameters

ID	dtqid	ID number of the target data queue
T_RDTQ	*pk_rdtq	Pointer to the packet to which data queue status is returned

● Return Parameters

ER	ercd	Terminated normally (E_OK)
T_RDTQ	*pk_rdtq	Pointer to the packet to which data queue status is returned

Contents of pk_rdtq

```
typedef     struct     t_rdtq{
        ID          stskid      +0    2     Transmission waiting task ID
        ID          wtskid      +2    2     Reception waiting task ID
        UINT        sdtqcnt     +4    4     Data bytes contained in data queue
} T_RDTQ;
```

[[Assembly language API]]

```
.include mr100.inc
ref_dtq   DTQID, PK_RDTQ
iref_dtq  DTQID, PK_RDTQ
```

● Parameters

DTQID	ID number of the target data queue
PK_RDTQ	Pointer to the packet to which data queue status is returned

● Register contents after service call is issued

Register name	Content after service call is issued
R0	Error code
R2	ID number of the target data queue
A1	Pointer to the packet to which data queue status is returned

[[Error code]]

None

[[Functional description]]

This service call returns various statuses of the data queue indicated by dtqid.

◆ **stskid**
Returned to stskid is the ID number of the task at the top of a transmission waiting queue (the next task to be de-queued). If no tasks are kept waiting, TSK_NONE is returned.

◆ **wtskid**
Returned to wtskid is the ID number of the task at the top of a reception waiting queue (the next task to be de-queued). If no tasks are kept waiting, TSK_NONE is returned.

◆ **sdtqcnt**
Returned to sdtqcnt is the number of data bytes stored in the data queue area.

If this service call is to be issued from task context, use ref_dtq; if issued from non-task context, use iref_dtq.

[[Example program statement]]

<<Example statement in C language>>

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
     T_RDTQ rdtq;
     ER ercd;
          :
     ercd = ref_dtq( ID_DTQ1, &rdtq );
          :
 }
```

<<Example statement in assembly language>>

```
_ refdtq:       .blkb   8
    .include mr100.inc
    .GLB       task
task:
          :
        PUSH.W  R2
        PUSH.L  A1
        ref_dtq #ID_DTQ1,#_refdtq
          :
```

5.6 Synchronization & Communication Function (Mailbox)

Specifications of the mailbox function of MR100 are listed in Table 5.11.

Table 5.11 Specifications of the Mailbox Function

No.	Item	Content	
1	Mailbox ID	1-255	
2	Mailbox priority	1-255	
3	Mailbox attribute	TA_TFIFO:	Waiting tasks enqueued in order of FIFO
		TA_TPRI:	Waiting tasks enqueued in order of priority
		TA_MFIFO:	Messages enqueued in order of FIFO
		TA_MPRI:	Messages enqueued in order of priority

Table 5.12 List of Mailbox Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	snd_mbx	**[S][B]**	Send to mailbox	O		O	O	O	
2	isnd_mbx				O	O	O	O	
3	rcv_mbx	**[S][B]**	Receive from mailbox	O		O		O	
4	prcv_mbx	**[S][B]**	Receive from mailbox(polling)	O		O	O	O	
5	iprcv_mbx				O	O	O	O	
6	trcv_mbx	**[S]**	Receive from mailbox(with timeout)	O		O		O	
7	ref_mbx		Reference mailbox status	O		O	O	O	
8	iref_mbx				O	O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls

- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

snd_mbx	**Send to mailbox**
isnd_mbx	**Send to mailbox (handler only)**

[[C Language API]]

```
ER ercd = snd_mbx( ID mbxid, T_MSG *pk_msg );
ER ercd = isnd_mbx( ID mbxid, T_MSG *pk_msg );
```

● Parameters

ID	mbxid	ID number of the mailbox to which transmitted
T_MSG	*pk_msg	Message to be transmitted

● Return Parameters

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
snd_mbx   MBXID,PK_MBX
isnd_mbx  MBXID,PK_MBX
```

● Parameters

MBXID	ID number of the mailbox to which transmitted
PK_MBX	Message to be transmitted (address)

● Register contents after service call is issued

Register name	Content after service call is issued
R0	Error code
R2	ID number of the mailbox to which transmitted
A1	Message to be transmitted (address)

[[Structure of the message packet]]

<<Mailbox message header>>

```
typedef     struct     t_msg{
        VP        msghead     +0     4     Kernel managed area
} T_MSG;
```

<<Mailbox message header with priority included>>

```
typedef     struct     t_msg{
        T_MSG     msgque      +0     4     Message header
        PRI       msgpri      +2     2     Message priority
} T_MSG;
```

[[Error code]]

None

[[Functional description]]

This service call sends the message indicated by pk_msg to the mailbox indicated by mbxid. T_MSG* should be specified with a 32-bit address. If there is any task waiting to receive a message in the target mailbox, the transmitted message is passed to the task at the top of the waiting queue, and the task is released from WAITING state.

To send a message to a mailbox whose attribute is TA_MFIFO, add a T_MSG structure at the beginning of the message when creating it, as shown in the example below.

To send a message to a mailbox whose attribute is TA_MPRI, add a T_MSG_PRI structure at the beginning of the message when creating it, as shown in the example below.

Messages should always be created in a RAM area regardless of whether its attribute is TA_MFIFO or TA_MPRI.

The T_MSG area is used by the kernel, so that it cannot be rewritten after a message has been sent. If this area is rewritten before the message is received after it was sent, operation of the service call cannot be guaranteed.

If this service call is to be issued from task context, use snd_mbx; if issued from non-task context, use isnd_mbx.

<<Example format of a message>>

```
typedef struct user_msg{
        T_MSG  t_msg;        /* T_MSG structure */
        B      data[16];     /* User message data */
} USER_MSG;
```

<<Example format of a message with priority included>>

```
typedef struct user_msg{
        T_MSG_PRI     t_msg;        /* T_MSG_PRI structure */
        B             data[16];     /* User message data */
} USER_MSG;
```

[[Example program statement]]

<<Example statement in C language>>

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
typedef struct pri_message
{
      T_MSG_PRI      msgheader;
      char    body[12];
} PRI_MSG;

void task(void)
{
   PRI_MSG     msg;
               :
   msg.msgpri = 5;
   snd_mbx( ID_msg,(T_MSG *)&msg);
       :
}
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
_g_userMsg:     .blkb   6       ; Header
                .blkb   12      ; Body
task:
          :
    PUSH.W      R2
    PUSH.L      A1
    snd_mbx     #ID_MBX1,#_g_userMsg
          :
```

rcv_mbx Receive from mailbox
prcv_mbx Receive from mailbox (polling)
iprcv_mbx Receive from mailbox (polling, handler only)
trcv_mbx Receive from mailbox (with timeout)

[[C Language API]]

```
ER ercd = rcv_mbx( ID mbxid, T_MSG **ppk_msg );
ER ercd = prcv_mbx( ID mbxid, T_MSG **ppk_msg );
ER ercd = iprcv_mbx( ID mbxid, T_MSG **ppk_msg );
ER ercd = trcv_mbx( ID mbxid, T_MSG **ppk_msg, TMO tmout );
```

● **Parameters**

ID	mbxid	ID number of the mailbox from which to receive
TMO	tmout	Timeout value (for trcv_mbx)
T_MSG	**ppk_msg	Pointer to the start of the area in which received message is stored

● **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code
T_MSG	**ppk_msg	Pointer to the start of the area in which received message is stored

[[Assembly language API]]

```
.include mr100.inc
rcv_mbx   MBXID
prcv_mbx  MBXID
iprcv_mbx MBXID
trcv_mbx  MBXID, TMO
```

● **Parameters**

MBXID	ID number of the mailbox from which to receive
TMO	Timeout value (for trcv_mbx)

● **Register contents after service call is issued**

rcv_mbx,prcv_mbx,iprcv_mbx

Register name	Content after service call is issued
R0	Error code
R2	ID number of the mailbox from which to receive
A1	Received message

trcv_mbx

Register name	Content after service call is issued
R0	Error code
R2	ID number of the mailbox from which to receive
R6R4	Timeout value
A1	Received message

[[Error code]]

E_RLWAI	Forced release from waiting
E_TMOUT	Polling failure or timeout or timed out

[[Functional description]]

This service call receives a message from the mailbox indicated by mbxid and stores the start address of the received message in the area pointed to by ppk_msg. T_MSG** should be specified with a 32-bit address. If data is present in the target mailbox, the data at the top of the mailbox is received.

On the other hand, if rcv_mbx or trcv_mbx is issued for a mailbox that has no messages in it, the task that issued the service call goes from RUNNING state to a message reception wait state, and is enqueued in a message reception waiting queue. In that case, if the attribute of the specified mailbox is TA_TFIFO, the task is enqueued in order of FIFO; if TA_TPRI, the task is enqueued in order of priority. For prcv_mbx and iprcv_mbx, the task returns immediately and responds to the call with the error code E_TMOUT.

For the trcv_mbx service call, specify a wait time for tmout in ms units. The values specified for tmout must be within 0x7FFFFFFF. If any value exceeding this limit is specified, the service call may not operate correctly. If TMO_POL=0 is specified for tmout, it means specifying 0 as a timeout value, in which case the service call operates the same way as prcv_mbx. Furthermore, if specified as tmout=TMO_FEVR(–1), it means specifying an infinite wait, in which case the service call operates the same way as rcv_mbx.

The task placed into WAITING state by execution of the rcv_mbx or trcv_mbx service call is released from WAITING state in the following cases:

- **When the rcv_mbx, trcv_mbx, prcv_mbx, or iprcv_mbx service call is issued before the tmout time elapses, with task-awaking conditions thereby satisfied**
  The error code returned in this case is E_OK.
- **When the first time tick occurred after tmout elapsed while task-awaking conditions remain unsatisfied**
  The error code returned in this case is E_TMOUT.
- **When the task is forcibly released from WAITING state by the rel_wai or irel_wai service call issued from another task or a handler**
  The error code returned in this case is E_RLWAI.

If this service call is to be issued from task context, use rcv_mbx,trcv_mbx,prcv_mbx; if issued from non-task context, use iprcv_mbx.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"

  typedef struct fifo_message
  {
        T_MSG   head;
        char    body[12];
  } FIFO_MSG;
  void task()
  {
    FIFO_MSG *msg;
                :
      if( rcv_mbx((T_MSG **)&msg, ID_mbx) == E_RLWAI )
          error("forced wakeup\n");
                :
                :
      if( prcv_mbx((T_MSG **)&msg, ID_mbx) != E_TMOUT )
          error("Timeout\n");
                :
                :
      if( trcv_mbx((T_MSG **)&msg, ID_mbx,10) != E_TMOUT )
          error("Timeout\n");
                :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    PUSHM       R6R4
    trcv_mbx    #ID_MBX1,#100
          :
    PUSH.W      R2
    rcv_mbx     #ID_MBX1
          :
    PUSH.W      R2
    prcv_mbx    #ID_MBX1
          :
```

ref_mbx Reference mailbox status
iref_mbx Reference mailbox status (handler only)

[[C Language API]]

```
ER ercd = ref_mbx( ID mbxid, T_RMBX *pk_rmbx );
ER ercd = iref_mbx( ID mbxid, T_RMBX *pk_rmbx );
```

- **Parameters**

ID	mbxid	ID number of the target mailbox
T_RMBX	*pk_rmbx	Pointer to the packet to which mailbox status is returned

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)
T_RMBX	*pk_rmbx	Pointer to the packet to which mailbox status is returned

Contents of pk_rmbx

typedef struct t_rmbx{

ID	wtskid	+0	2	Reception waiting task ID
T_MSG	*pk_msg	+4	4	Next message packet to be received

} T_RMBX;

[[Assembly language API]]

```
.include mr100.inc
ref_mbx  MBXID, PK_RMBX
iref_mbx MBXID, PK_RMBX
```

- **Parameters**

MBXID	ID number of the target mailbox
PK_RMBX	Pointer to the packet to which mailbox status is returned

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	ID number of the target mailbox
A1	Pointer to the packet to which mailbox status is returned

[[Error code]]

None

[[Functional description]]

This service call returns various statuses of the mailbox indicated by mbxid.

◆ **wtskid**

Returned to wtskid is the ID number of the task at the top of a reception waiting queue (the next task to be dequeued). If no tasks are kept waiting, TSK_NONE is returned.

◆ ***pk_msg**

Returned to *pk_msg is the start address of the next message to be received. If there are no messages to be received next, NULL is returned. T_MSG* should be specified with a 32-bit address.

If this service call is to be issued from task context, use ref_mbx; if issued from non-task context, use iref_mbx.

[[Example program statement]]

<<Example statement in C language>>

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
     T_RMBX rmbx;
     ER ercd;
         :
     ercd = ref_mbx( ID_MBX1, &rmbx );
         :
 }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
_ refmbx:       .blkb   6
task:
          :
        PUSH.W  R2
        PUSH.L  A1
        ref_mbx #ID_MBX1,#_refmbx
          :
```

5.7 Memory Pool Management Function (Fixed-size Memory Pool)

Specifications of the fixed-size memory pool function of MR100 are listed in Table 5.13.

The memory pool area to be acquired can be specified by a section name for each memory pool during configuration.

Table 5.13 Specifications of the Fixed-size memory pool Function

No.	Item	Content	
1	Fixed-size memory pool ID	1-255	
2	Number of fixed-size memory block	1-65535	
3	Size of fixed-size memory block	4-65535	
4	Supported attributes	TA_TFIFO:	Waiting tasks enqueued in order of FIFO
		TA_TPRI:	Waiting tasks enqueued in order of priority
5	Specification of memory pool area	Area to be acquired specifiable by a section	

Table 5.14 List of Fixed-size memory pool Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	get_mpf	**[S][B]**	Aquires fixed-size memory block	O		O		O	
2	pget_mpf	**[S][B]**	Aquires fixed-size memory block (polling)	O		O	O	O	
3	ipget_mpf				O	O	O	O	
4	tget_mpf	**[S]**	Aquires fixed-size memory block (with timeout)	O		O		O	
5	rel_mpf	**[S][B]**	Releases fixed-size memory block	O		O	O	O	
6	irel_mpf				O	O	O	O	
7	ref_mpf		References fixed-size memory pool status	O		O	O	O	
8	iref_mpf				O	O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls
- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

get_mpf — Aquire fixed-size memory block
pget_mpf — Aquire fixed-size memory block (polling)
ipget_mpf — Aquire fixed-size memory block (polling, handler only)
tget_mpf — Aquire fixed-size memory block (with timeout)

[[C Language API]]

```
ER ercd = get_mpf( ID mpfid, VP *p_blk );
ER ercd = pget_mpf( ID mpfid, VP *p_blk );
ER ercd = ipget_mpf( ID mpfid, VP *p_blk );
ER ercd = tget_mpf( ID mpfid, VP *p_blk,TMO tmout );
```

● **Parameters**

ID	mpfid	ID number of the target fixed-size memory pool to be acquired
VP	*p_blk	Pointer to the start address of the acquired memory block
TMO	tmout	Timeout value(tget_mpf)

● **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code
VP	*p_blk	Pointer to the start address of the acquired memory block

[[Assembly language API]]

```
.include mr100.inc
get_mpf   MPFID
pget_mpf  MPFID
ipget_mpf MPFID
tget_mpf  MPFID,TMO
```

● **Parameters**

MPFID	ID number of the target fixed-size memory pool to be acquired
TMO	Timeout value(tget_mpf)

● Register contents after service call is issued

get_mpf,pget_mpf,ipget_mpf

Register name	Content after service call is issued
R0	Error code
R3R1	Start address of the acquired memory block
R2	ID number of the target fixed-size memory pool to be acquired

tget_mpf

Register name	Content after service call is issued
R0	Error code
R3R1	Start address of the acquired memory block
R2	ID number of the target fixed-size memory pool to be acquired
R6R4	Timeout value

[[Error code]]

E_RLWAI	Forced release from waiting
E_TMOUT	Polling failure or timeout or timed out
EV_RST	Released from WAITING state by clearing of the memory pool area

[[Functional description]]

This service call acquires a memory block from the fixed-size memory pool indicated by mpfid and stores the start address of the acquired memory block in the variable p_blk. The content of the acquired memory block is indeterminate.

If the fixed-size memory pool indicated by mpfid has no memory blocks in it and the used service call is tget_mpf or get_mpf, the task that issued it goes to a memory block wait state and is enqueued in a memory block waiting queue. In that case, if the attribute of the specified fixed-size memory pool is TA_TFIFO, the task is enqueued in order of FIFO; if TA_TPRI, the task is enqueued in order of priority. If the issued service call was pget_mpf or ipget_mpf, the task returns immediately and responds to the call with the error code E_TMOUT.

For the tget_mpf service call, specify a wait time for tmout in ms units. The values specified for tmout must be within (0x7FFFFFFF – time tick value). If any value exceeding this limit is specified, the service call may not operate correctly. If TMO_POL=0 is specified for tmout, it means specifying 0 as a timeout value, in which case the service call operates the same way as pget_mpf. Furthermore, if specified as tmout=TMO_FEVR(–1), it means specifying an infinite wait, in which case the service call operates the same way as get_mpf.

The task placed into WAITING state by execution of the get_mpf or tget_mpf service call is released from WAITING state in the following cases:

- ◆ **When the rel_mpf or irel_mpf service call is issued before the tmout time elapses, with task-awaking conditions thereby satisfied**
  The error code returned in this case is E_OK.
- ◆ **When the first time tick occurred after tmout elapsed while task-awaking conditions remain unsatisfied**
  The error code returned in this case is E_TMOUT.
- ◆ **When the task is forcibly released from WAITING state by the rel_wai or irel_wai service call issued from another task or a handler**
  The error code returned in this case is E_RLWAI.
- ◆ **When the target memory pool being waited for is initialized by the vrst_mpf service call issued from another task**
  The error code returned in this case is EV_RST.

If this service call is to be issued from task context, use get_mpf,pget_mpf,tget_mpf; if issued from non-task context, use ipget_mpf.

[[Example program statement]]

<<Example statement in C language>>

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
VP      p_blk;
void task()
{
    if( get_mpf(ID_mpf ,&p_blk) != E_OK ){
          error("Not enough memory\n");
    }
                :
    if( pget_mpf(ID_mpf ,&p_blk) != E_OK ){
          error("Not enough memory\n");
    }
                :
    if( tget_mpf(ID_mpf ,&p_blk, 10) != E_OK ){
          error("Not enough memory\n");
    }

}
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    get_mpf     #ID_MPF1
          :
    PUSH.W      R2
    pget_mpf    #ID_MPF1
          :
    PUSH.W      R2
    PUSHM       R6R4
    tget_mpf    #ID_MPF1,#200
          :
```

rel_mpf Release fixed-size memory block
irel_mpf Release fixed-size memory block (handler only)

[[C Language API]]

```
ER ercd = rel_mpf( ID mpfid, VP blk );
ER ercd = irel_mpf( ID mpfid, VP blk);
```

- **Parameters**

ID	mpfid	ID number of the fixed-size memory pool to be released
VP	blk	Start address of the memory block to be returned

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
rel_mpf   MPFID,BLK
irel_mpf  MPFID,BLK
```

- **Parameters**

MPFID	ID number of the fixed-size memory pool to be released
BLK	Start address of the memory block to be returned

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R3R1	Start address of the memory block to be returned
R2	ID number of the fixed-size memory pool to be released

[[Error code]]

None

[[Functional description]]

This service call releases a memory block whose start address is indicated by blk. The start address of the memory block to be released that is specified here should always be that of the memory block acquired by get_mpf, tget_mpf, pget_mpf, or ipget_mpf.

If tasks are enqueued in a waiting queue for the target memory pool, the task at the top of the waiting queue is dequeued and linked to a ready queue, and is assigned a memory block. At this time, the task changes state from a memory block wait state to RUNNING or READY state. This service call does not check the content of blk, so that if the address stored in blk is incorrect, the service call may not operate correctly.

If this service call is to be issued from task context, use rel_mpf; if issued from non-task context, use irel_mpf.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
      VP   p_blf;
      if( get_mpf(ID_mpf1,&p_blf) != E_OK )
          error("Not enough memory \n");
                :
      rel_mpf(ID_mpf1,p_blf);
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
_g_blk: .blkb 4
task:
          :
    PUSH.W      R2
    get_mpf     #ID_MPF1
          :
    MOV.L       R3R1,_g_blk
    PUSH.W      R2
    rel_mpf     #ID_MPF1,_g_blk
          :
```

ref_mpf Reference fixed-size memory pool status
iref_mpf Reference fixed-size memory pool status (handler only)

[[C Language API]]

```
ER ercd = ref_mpf( ID mpfid, T_RMPF *pk_rmpf );
ER ercd = iref_mpf( ID mpfid, T_RMPF *pk_rmpf );
```

● Parameters

ID	mpfid	Task ID waiting for memory block to be acquired
T_RMPF	*pk_rmpf	Pointer to the packet to which fixed-size memory pool status is returned

● Return Parameters

ER	ercd	Terminated normally (E_OK)
T_RMPF	*pk_rmpf	Pointer to the packet to which fixed-size memory pool status is returned

Contents of pk_rmpf

```
typedef     struct     t_rmpf{
        ID       wtskid      +0    2     Task ID waiting for memory block to be acquired
        UINT     fblkcnt     +2    4     Number of free memory blocks
} T_RMPF;
```

[[Assembly language API]]

```
.include mr100.inc
ref_mpf   MPFID,PK_RMPF
iref_mpf  MPFID,PK_RMPF
```

● Parameters

MPFID	Task ID waiting for memory block to be acquired
PK_RMPF	Pointer to the packet to which fixed-size memory pool status is returned

● Register contents after service call is issued

Register name	Content after service call is issued
R0	Error code
R2	Task ID waiting for memory block to be acquired
A1	Pointer to the packet to which fixed-size memory pool status is returned

[[Error code]]

None

[[Functional description]]

This service call returns various statuses of the message buffer indicated by mpfid.

◆ **wtskid**

Returned to wtskid is the ID number of the task at the top of a memory block waiting queue (the first queued task). If no tasks are kept waiting, TSK_NONE is returned.

◆ **fblkcnt**

The number of free memory blocks in the specified memory pool is returned.

If this service call is to be issued from task context, use rel_mpf; if issued from non-task context, use irel_mpf.

[[Example program statement]]

<<Example statement in C language>>

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
     T_RMPF rmpf;
     ER ercd;
          :
     ercd = ref_mpf( ID_MPF1, &rmpf );
          :
 }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB       task
_ refmpf:      .blkb   6
task:
          :
        PUSH.W  R2
        PUSH.L  A1
        ref_mpf #ID_MPF1,#_refmpf
          :
```

5.8 Memory Pool Management Function (Variable-size Memory Pool)

Specifications of the Variable-size Memory pool function of MR100 are listed in Table 5.15.

The memory pool area to be acquired can be specified by a section name for each memory pool during configuration.

Table 5.15 Specifications of the Variable-size memory Pool Function

No.	Item	Content
1	Variable-size memory pool ID	1-255
2	Size of Variable-size Memory pool	32-67108864
3	Maximum number of memory blocks to be acquired	4-65504
4	Supported attributes	When memory is insufficient, task-waiting APIs are not supported.
5	Specification of memory pool area	Area to be acquired specifiable by a section

Table 5.16 List of Variable -size memory pool Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	pget_mpl		Aquires variable-size memory block (polling)	O		O	O	O	
2	rel_mpl		Releases variable-size memory block	O		O	O	O	
3	ref_mpl		References variable-size memory pool status	O		O	O	O	
4	iref_mpl				O	O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls
- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

pget_mpl   Aquire variable-size memory block (polling)

[[C Language API]]

```
ER ercd = pget_mpl( ID mplid, UINT blksz, VP *p_blk );
```

- **Parameters**

ID	mplid	ID number of the target Variable-size Memory pool to be acquired
UINT	blksz	Memory size to be acquired (in bytes)
VP	*p_blk	Pointer to the start address of the acquired variable memory

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code
VP	*p_blk	Pointer to the start address of the acquired variable memory

[[Assembly language API]]

```
.include mr100.inc
pget_mpl  MPLID,BLKSZ
```

- **Parameters**

MPLID	ID number of the target Variable-size Memory pool to be acquired
BLKSZ	Memory size to be acquired (in bytes)

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R3R1	Memory size to be acquired
R2	ID number of the target Variable-size Memory pool to be acquired

[[Error code]]

E_TMOUT	No memory block

[[Functional description]]

This service call acquires a memory block from the variable-size memory pool indicated by mplid and stores the start address of the acquired memory block in the variable p_blk. The content of the acquired memory block is indeterminate.

If the specified variable-size memory pool has no memory blocks in it, the task returns immediately and responds to the call with the error code E_TMOUT.

This service call can be issued only from task context. It cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  VP      p_blk;
  void task()
  {
    if( pget_mpl(ID_mpl , 200, &p_blk) != E_OK ){
          error("Not enough memory\n");
    }
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB       task
task:
          :
    PUSH.W     R2
    pget_mpl   #ID_MPL1,#200
          :
```

rel_mpl Release variable-size memory block

[[C Language API]]

```
ER ercd = rel_mpl( ID mplid, VP blk );
```

- **Parameters**

ID	mplid	ID number of Variable-size Memory pool of the memory block to be released
VP	Blk	Start address of the memory block to be returned

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr100.inc
rel_mpl  MPLID,BLK
```

- **Parameters**

MPLID	ID number of Variable-size Memory pool of the memory block to be released
BLK	Start address of the memory block to be returned

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R3R1	Start address of the memory block to be returned
R2	ID number of Variable-size Memory pool of the memory block to be released

[[Error code]]

None

[[Functional description]]

This service call releases a memory block whose start address is indicated by blk. The start address of the memory block to be released that is specified here should always be that of the memory block acquired by pget_mpl.
This service call does not check the content of blk, so that if the address stored in blk is incorrect, the service call may not operate correctly.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
      VP   p_blk;
      if( get_mpl(ID_mpl1, 200, &p_blk) != E_OK )
          error("Not enough memory \n");
                :
      rel_mpl(ID_mp1,p_blk);
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
_g_blk: .blkb 4
task:
          :
    PUSH.W      R2
    pget_mpl    #ID_MPL1,#200
          :
    MOV.L       R3R1,_g_blk
    PUSH.W      R2
    rel_mpl     #ID_MPL1,_g_blk
          :
```

ref_mpl Reference variable-size memory pool status
iref_mpl Reference variable-size memory pool status (handler only)

[[C Language API]]

```
ER ercd = ref_mpl( ID mplid, T_RMPL *pk_rmpl );
ER ercd = iref_mpl( ID mplid, T_RMPL *pk_rmpl );
```

● Parameters

ID	mplid	ID number of the target variable-size memory pool
T_RMPL	*pk_rmpl	Pointer to the packet to which variable-size memory pool status is returned

● Return Parameters

ER	ercd	Terminated normally (E_OK)
T_RMPL	*pk_rmpl	Pointer to the packet to which variable-size memory pool status is returned

Contents of pk_rmpl

typedef struct t_rmpl{

ID	wtskid	+0	2	Task ID waiting for memory block to be acquired (unused)
SIZE	fmplsz	+4	4	Free memory size (in bytes)
UINT	fblksz	+8	4	Maximum size of memory that can be acquired immediately (in bytes)

} T_RMPL;

[[Assembly language API]]

```
.include mr100.inc
ref_mpl  MPLID,PK_RMPL
iref_mpl MPLID,PK_RMPL
```

● Parameters

MPLID	ID number of the target variable-size memory pool
PK_RMPL	Pointer to the packet to which variable-size memory pool status is returned

● Register contents after service call is issued

Register name	Content after service call is issued
R0	Error code
R2	ID number of the target variable-size memory pool
A1	Pointer to the packet to which variable-size memory pool status is returned

[[Error code]]

None

[[Functional description]]

This service call returns various statuses of the message buffer indicated by mplid.

- ◆ **wtskid**
  Unused.
- ◆ **fmplsz**
  A free memory size is returned.
- ◆ **fblksz**
  The maximum size of memory that can be acquired immediately is returned.

If this service call is to be issued from task context, use ref_mpl; if issued from non-task context, use iref_mpl.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
      T_RMPL rmpl;
      ER ercd;
          :
      ercd = ref_mpl( ID_MPL1, &rmpl );
          :
  }
```

<<Example statement in assembly language>>

```
     .include mr100.inc
     .GLB        task
_ refmpl:        .blkb   10
task:
            :
        PUSH.W  R2
        PUSH.L  A1
        ref_mpl #ID_MPL1,_refmpl
            :
```

5.9 Time Management Function

Specifications of the time management function of MR100 are listed in Table 5.17.

Table 5.17 Specifications of the Time Management Function

No.	Item	Content
1	System time value	Unsigned 48 bits
2	Unit of system time value	1[ms]
3	System time updating cycle	User-specified time tick updating time [ms]
4	Initial value of system time (at initial startup)	000000000000H

Table 5.18 List of Time Management Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	get_tim	**[S]**	Reference system time	○		○	○	○	
2	iget_tim				○	○	○	○	
3	set_tim	**[S]**	Set system time	○		○	○	○	
4	iset_tim				○	○	○	○	
5	isig_tim	**[S]**	Supply a time tick		○	○	○	○	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls
- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

set_tim	Set system time
iset_tim	Set system time (handler only)

[[C Language API]]

```
ER ercd = set_tim( SYSTIM *p_systim );
ER ercd = iset_tim( SYSTIM *p_systim );
```

● Parameters

SYSTIM *p_systim Pointer to the packet that indicates the system time to be set

Contents of p_systim

```
typedef     struct t_systim {
        UH          utime       0       2       (16 high-order bits)
        UW          ltime       +4      4       (32 low-order bits)
} SYSTIM;
```

● Return Parameters

ER ercd Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
set_tim   PK_TIM
iset_tim  PK_TIM
```

● Parameters

PK_TIM Pointer to the packet that indicates the system time to be set

● Register contents after service call is issued

Register name	Content after service call is issued
R0	Error code
A1	Pointer to the packet that indicates the system time to be set

[[Error code]]

None

[[Functional description]]

This service call updates the current value of the system time to the value indicated by p_systim. The time specified in p_systim is expressed in ms units, and not by the number of time ticks.

The values specified for p_systim must be within 0x7FFF: FFFFFFFF. If any value exceeding this limit is specified, the service call may not operate correctly.

If this service call is to be issued from task context, use set_tim; if issued from non-task context, use iset_tim.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
     SYSTIME time;         /* Time data storing variable */
     time.utime = 0;       /* Sets upper time data */
     time.ltime = 0;       /* Sets lower time data */
     set_tim( &time );      /* Sets the system time */
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB       task
_g_systim:
        .WORD   1111H
        .LWORD  22223333H
task:
           :
        PUSHM   A1
        set_tim #_g_systim
           :
```

get_tim — Reference system time
iget_tim — Reference system time (handler only)

[[C Language API]]

```
ER ercd = get_tim( SYSTIM *p_systim );
ER ercd = iget_tim( SYSTIM *p_systim );
```

- **Parameters**

SYSTIM	*p_systim	Pointer to the packet to which current system time is returned

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)
SYSTIM	*p_systim	Pointer to the packet to which current system time is returned

Contents of p_systim

```
typedef    struct t_systim {
    UH       utime      0     2     (16 high-order bits)
    UW       ltime      +4    4     (32 low-order bits)
} SYSTIM;
```

[[Assembly language API]]

```
.include mr100.inc
get_tim  PK_TIM
iget_tim PK_TIM
```

- **Parameters**

PK_TIM Pointer to the packet to which current system time is returned

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
A1	Pointer to the packet to which current system time is returned

[[Error code]]

None

[[Functional description]]

This service call stores the current value of the system time in p_systim.

If this service call is to be issued from task context, use get_tim; if issued from non-task context, use iget_tim.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
     SYSTIME time;          /* Time data storing variable */
     get_tim( &time );      /* Refers to the system time */
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
_g_systim:      .blkb   6
task:
            :
        PUSHM   A1
        get_tim #_g_systim
            :
```

isig_tim Supply a time tick

[[Functional description]]

This service call updates the system time.

The isig_tim is automatically started every tick_time interval(ms) if the system clock is defined by the configuration file. The application cannot call this function because it is not implementing as service call.

When a time tick is supplied, the kernel is processed as follows:

(1) Updates the system time
(2) Starts an alarm handler
(3) Starts a cyclic handler
(4) Processes the timeout processing of the task put on WAITING state by service call with timeout such as tslp_tsk.

5.10 Time Management Function (Cyclic Handler)

Specifications of the cyclic handler function of MR100 are listed in Table 5.19. The cyclic handler description languages in item No. 4 are those specified in the GUI configurator. They are not output to a configuration file, nor are the MR100 kernel concerned with them.

Table 5.19 Specifications of the Cyclic Handler Function

No.	Item	Content	
1	Cyclic handler ID	1-255	
2	Activation cycle	0-7fffffff[ms]	
3	Activation phase	0-7fffffff[ms]	
4	Extended information	32 bits	
5	Cyclic handler attribute	TA_HLNG:	Handlers written in high-level language
		TA_ASM:	Handlers written in assembly language
		TA_STA:	Starts operation of cyclic handler
		TA_PHS:	Saves activation phase

Table 5.20 List of Cyclic Handler Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	sta_cyc	**[S][B]**	Starts cyclic handler operation	O		O	O	O	
2	ista_cyc				O	O	O	O	
3	stp_cyc	**[S][B]**	Stops cyclic handler operation	O		O	O	O	
4	istp_cyc				O	O	O	O	
5	ref_cyc		Reference cyclic handler status	O		O	O	O	
6	iref_cyc				O	O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls
- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

sta_cyc Start cyclic handler operation
ista_cyc Start cyclic handler operation (handler only)

[[C Language API]]

```
ER ercd = sta_cyc( ID cycid );
ER ercd = ista_cyc( ID cycid );
```

- **Parameters**

ID	cycid	ID number of the cyclic handler to be operated

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
sta_cyc   CYCNO
ista_cyc  CYCNO
```

- **Parameters**

CYCNO	ID number of the cyclic handler to be operated

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	ID number of the cyclic handler to be operated

[[Error code]]

None

[[Functional description]]

This service call places the cyclic handler indicated by cycid into an operational state. If the cyclic handler attribute of TA_PHS is not specified, the cyclic handler is started every time the activate cycle elapses, start with the time at which this service call was invoked.

If while TA_PHS is not specified this service call is issued to a cyclic handler already in an operational state, it sets the time at which the cyclic handler is to start next.

If while TA_PHS is specified this service call is issued to a cyclic handler already in an operational state, it does not set the startup time.

If this service call is to be issued from task context, use sta_cyc; if issued from non-task context, use ista_cyc.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
        :
    sta_cyc ( ID_cyc1 );
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB       task
task:
           :
        PUSH.W  R2
        sta_cyc #ID_CYC1
           :
```

stp_cyc — Stops cyclic handler operation
istp_cyc — Stops cyclic handler operation (handler only)

[[C Language API]]

```
ER ercd = stp_cyc( ID cycid );
ER ercd = istp_cyc( ID cycid );
```

- **Parameters**

ID	cycid	ID number of the cyclic handler to be stopped

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
stp_cyc   CYCNO
istp_cyc  CYCNO
```

- **Parameters**

CYCNO	ID number of the cyclic handler to be stopped

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	ID number of the cyclic handler to be stopped

[[Error code]]

None

[[Functional description]]

This service call places the cyclic handler indicated by cycid into a non-operational state.

If this service call is to be issued from task context, use stp_cyc; if issued from non-task context, use istp_cyc.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
        :
     stp_cyc ( ID_cyc1 );
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
           :
        PUSH.W  R2
        stp_cyc #ID_CYC1
           :
```

ref_cyc	**Reference cyclic handler status**
iref_cyc	**Reference cyclic handler status (handler only)**

[[C Language API]]

```
ER ercd = ref_cyc( ID cycid, T_RCYC *pk_rcyc );
ER ercd = iref_cyc( ID cycid, T_RCYC *pk_rcyc );
```

● Parameters

ID	cycid	ID number of the target cyclic handler
T_RCYC	*pk_rcyc	Pointer to the packet to which cyclic handler status is returned

● Return Parameters

ER	ercd	Terminated normally (E_OK)
T_RCYC	*pk_rcyc	Pointer to the packet to which cyclic handler status is returned

Contents of pk_rcyc

```
typedef     struct     t_rcyc{
        STAT      cycstat     +0    2     Operating status of cyclic handler
        RELTIM    lefttim     +2    4     Left time before cyclic handler starts up
} T_RCYC;
```

[[Assembly language API]]

```
.include mr100.inc
ref_cyc   ID,PK_RCYC
iref_cyc  ID,PK_RCYC
```

● Parameters

CYCNO	ID number of the target cyclic handler
PK_RCYC	Pointer to the packet to which cyclic handler status is returned

● Register contents after service call is issued

Register name	Content after service call is issued
R0	Error code
R2	ID number of the target cyclic handler
A1	Pointer to the packet to which cyclic handler status is returned

[[Error code]]

None

[[Functional description]]

This service call returns various statuses of the cyclic handler indicated by cycid.

◆ **cycstat**

The status of the target cyclic handler is returned.

*TCYC_STA	Cyclic handler is an operational state.
*TCYC_STP	Cyclic handler is a non-operational state.

◆ **lefttim**

The remaining time before the target cyclic handler will start next is returned. This time is expressed in ms units. If the target cyclic handler is non-operational state, the returned value is indeterminate.

If this service call is to be issued from task context, use ref_cyc; if issued from non-task context, use iref_cyc.

[[Example program statement]]

<<Example statement in C language>>

```
 #include <itron.h>
 #include <kernel.h>
 #include "kernel_id.h"
 void task()
 {
     T_RCYC rcyc;
     ER ercd;
         :
     ercd = ref_cyc( ID_CYC1, &rcyc );
         :
 }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
_ refcyc:       .blkb   6
task:
          :
        PUSH.W  R2
        PUSH.L  A1
        ref_cyc #ID_CYC1,#_refcyc
          :
```

5.11 Time Management Function (Alarm Handler)

Specifications of the alarm handler function of MR100 are listed in Table 5.21. The alarm handler description languages in item No. 4 are those specified in the GUI configurator. They are not output to a configuration file, nor are the MR100 kernel concerned with them.

Table 5.21 Specifications of the Alarm Handler Function

No.	Item	Content	
1	Alarm handler ID	1-255	
2	Activation time	0-7fffffff [ms]	
3	Extended information	16 bits	
4	Alarm handler attribute	TA_HLNG:	Handlers written in high-level language
		TA_ASM:	Handlers written in assembly language

Table 5.22 List of Alarm Handler Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	sta_alm		Starts alarm handler operation	○		○	○	○	
2	ista_alm				○	○	○	○	
3	stp_alm		Stops alarm handler operation	○		○	○	○	
4	istp_alm				○	○	○	○	
5	ref_alm		References alarm handler status	○		○	○	○	
6	iref_alm				○	○	○	○	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls
- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

sta_alm Start alarm handler operation
ista_alm Start alarm handler operation (handler only)

[[C Language API]]

```
ER ercd = sta_alm( ID almid, RELTIM almtim );
ER ercd = ista_alm( ID almid, RELTIM almtim );
```

- **Parameters**

ID	almid	ID number of the alarm handler to be operated
RELTIM	almtim	Alarm handler startup time (relative time)

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
sta_alm   ALMID,ALMTIM
ista_alm  ALMID,ALMTIM
```

- **Parameters**

ALMID	ID number of the alarm handler to be operated
ALMTIM	Alarm handler startup time (relative time)

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	ID number of the alarm handler to be operated
R6R4	Alarm handler startup time (relative time)

[[Error code]]

None

[[Functional description]]

This service call sets the activation time of the alarm handler indicated by almid as a relative time of day after the lapse of the time specified by almtim from the time at which it is invoked, and places the alarm handler into an operational state.

If an already operating alarm handler is specified, the previously set activation time is cleared and updated to a new activation time. If almtim = 0 is specified, the alarm handler starts at the next time tick. The values specified for almtim must be within (0x7FFFFFFF – time tick value). If any value exceeding this limit is specified, the service call may not operate correctly. If 0 is specified for almtim , the alarm handler is started at the next time tick.

If this service call is to be issued from task context, use sta_alm; if issued from non-task context, use ista_alm.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
        :
    sta_alm ( ID_alm1,100 );
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB       task
task:
           :
        PUSH.W  R2
        PUSHM   R6R4
        sta_alm #ID_ALM1,#100
           :
```

stp_alm Stop alarm handler operation
istp_alm Stop alarm handler operation (handler only)

[[C Language API]]

```
ER ercd = stp_alm( ID almid );
ER ercd = istp_alm( ID almid );
```

- **Parameters**

ID	almid	ID number of the alarm handler to be stopped

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
stp_alm   ALMID
istp_alm  ALMID
```

- **Parameters**

ALMID	ID number of the alarm handler to be stopped

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	ID number of the alarm handler to be stopped

[[Error code]]

None

[[Functional description]]

This service call places the alarm handler indicated by almid into a non-operational state.

If this service call is to be issued from task context, use stp_alm; if issued from non-task context, use istp_alm.

[[Example program statement]]

<<Example statement in C language>>

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
      :
   stp_alm ( ID_alm1 );
      :
}
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB       task
task:
         :
        PUSH.W  R2
        stp_alm #ID_ALM1
         :
```

ref_alm Reference alarm handler status
iref_alm Reference alarm handler status (handler only)

[[C Language API]]

```
ER ercd = ref_alm( ID almid, T_RALM *pk_ralm );
ER ercd = iref_alm( ID almid, T_RALM *pk_ralm );
```

● **Parameters**

ID	almid	ID number of the target alarm handler
T_RALM	*pk_ralm	Pointer to the packet to which alarm handler status is returned

● **Return Parameters**

ER	ercd	Terminated normally (E_OK)
T_RALM	*pk_ralm	Pointer to the packet to which alarm handler status is returned

Contents of pk_ralm

typedef struct t_ralm{

STAT	almstat	+0	2	Operating status of alarm handler
RELTIM	lefttim	+2	4	This service call returns various statuses of the alarm handler indicat

} T_RALM;

[[Assembly language API]]

```
.include mr100.inc
ref_alm  ALMID,PK_RALM
iref_alm ALMID,PK_RALM
```

● **Parameters**

ALMID	ID number of the target alarm handler
PK_RALM	Pointer to the packet to which alarm handler status is returned

● **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	ID number of the target alarm handler
A1	Pointer to the packet to which alarm handler status is returned

[[Error code]]

None

[[Functional description]]

This service call returns various statuses of the alarm handler indicated by almid.

◆ **almstat**

The status of the target alarm handler is returned.

*TALM_STA	Alarm handler is an operational state.
*TALM_STP	Alarm handler is a non-operational state.

◆ **lefttim**

The remaining time before the target alarm handler will start next is returned. This time is expressed in ms units. If the target alarm handler is a non-operational state, the returned value is indeterminate.

If this service call is to be issued from task context, use ref_alm; if issued from non-task context, use iref_alm.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
      T_RALM ralm;
      ER ercd;
          :
      ercd = ref_alm( ID_ALM1, &ralm );
          :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
_ refalm:       .blkb   6
task:
          :
        PUSH.W  R2
        PUSH.L  A1
        ref_alm #ID_ALM1,#_refalm
          :
```

5.12 System Status Management Function

Table 5.23 List of System Status Management Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	rot_rdq	**[S][B]**	Rotates task precedence	O		O	O	O	
2	irot_rdq	**[S][B]**			O	O	O	O	
3	get_tid	**[S][B]**	References task ID in the RUNNING state	O		O	O	O	
4	iget_tid	**[S]**			O	O	O	O	
5	loc_cpu	**[S][B]**	Locks the CPU	O		O	O	O	O
6	iloc_cpu	**[S]**			O	O	O	O	O
7	unl_cpu	**[S][B]**	Unlocks the CPU	O		O	O	O	O
8	iunl_cpu	**[S]**			O	O	O	O	O
9	dis_dsp	**[S][B]**	Disables dispatching	O		O	O	O	
10	ena_dsp	**[S][B]**	Enables dispatching	O		O	O	O	
11	sns_ctx	**[S]**	References context	O		O	O	O	O
12	sns_loc	**[S]**	References CPU state	O		O	O	O	O
13	sns_dsp	**[S]**	References dispatching state	O		O	O	O	O
14	sns_dpn	**[S]**	References dispatching pending state	O		O	O	O	O

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls
- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

rot_rdq Rotate task precedence
irot_rdq Rotate task precedence (handler only)

[[C Language API]]

```
ER ercd = rot_rdq( PRI tskpri );
ER ercd = irot_rdq( PRI tskpri );
```

- **Parameters**

PRI	tskpri	Task priority to be rotated

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
rot_rdq   TSKPRI
irot_rdq  TSKPRI
```

- **Parameters**

TSKPRI	Task priority to be rotated

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R3	Task priority to be rotated

[[Error code]]

None

[[Functional description]]

This service call rotates the ready queue whose priority is indicated by tskpri. In other words, it relocates the task enqueued at the top of the ready queue of the specified priority by linking it to behind the tail of the ready queue, thereby switching over the executed tasks that have the same priority. Figure 5-1 depicts the manner of how this is performed.

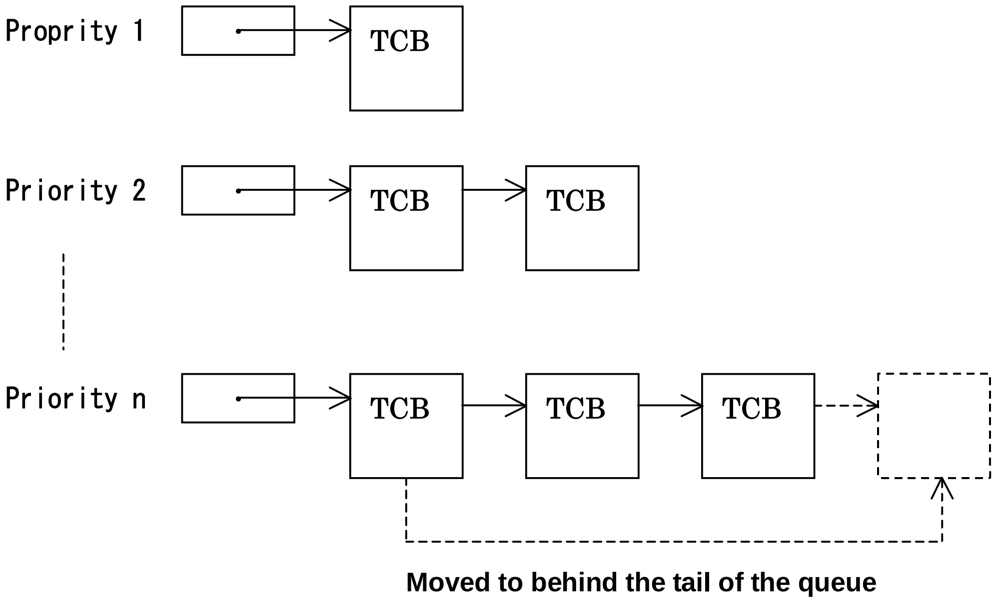

Figure 5-1. Manipulation of the ready queue by the rot_rdq service call

By issuing this service call at given intervals, it is possible to perform round robin scheduling. If tskpri=TPRI_SELF is specified when using the rot_rdq service call, the ready queue whose priority is that of the issuing task is rotated. TPRI_SELF cannot be specified in the irot_rdq service call. TPRI_SELF cannot be specified by irot_rdq service call. However, an error is not returned even if it is specified.

If the priority of the issuing task itself is specified in this service call, the issuing task is relocated to behind the tail of the ready queue in which it is enqueued. Note that if the ready queue of the specified priority has no tasks in it, no operation is performed.

If this service call is to be issued from task context, use rot_rdq; if issued from non-task context, use irot_rdq.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include “kernel_id.h”
  void task()
  {
           :
      rot_rdq( 2 );
           :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
           :
        PUSH.W  R3
        rot_rdq #2
           :
```

get_tid	**Reference task ID in the RUNNING state**
iget_tid	**Reference task ID in the RUNNING state (handler only)**

[[C Language API]]

```
ER ercd = get_tid( ID *p_tskid );
ER ercd = iget_tid( ID *p_tskid );
```

● Parameters

ID	*p_tskid	Pointer to task ID

● Return Parameters

ER	ercd	Terminated normally (E_OK)
ID	*p_tskid	Pointer to task ID

[[Assembly language API]]

```
.include mr100.inc
get_tid
iget_tid
```

● Parameters

None

● Register contents after service call is issued

Register name	Content after service call is issued
R0	Error code
R2	Acquired task ID

[[Error code]]

None

[[Functional description]]

This service call returns the task ID currently in RUNNING state to the area pointed to by p_tskid. If this service call is issued from a task, the ID number of the issuing task is returned. If this service call is issued from non-task context, the task ID being executed at that point in time is returned. If there are no tasks currently in an executing state, TSK_NONE is returned.

If this service call is to be issued from task context, use get_tid; if issued from non-task context, use iget_tid.

[[Example program statement]]

<<Example statement in C language>>

```
#include <itron.h>
#include <kernel.h>
#include “kernel_id.h”
void task()
{
  ID tskid;
     :
   get_tid(&tskid);
     :
}
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
       get_tid
          :
```

loc_cpu — Lock the CPU
iloc_cpu — Lock the CPU (handler only)

[[C Language API]]

```
ER ercd = loc_cpu();
ER ercd = iloc_cpu();
```

- **Parameters**

None

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
loc_cpu
iloc_cpu
```

- **Parameters**

None

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code

[[Error code]]

None

[[Functional description]]

This service call places the system into a CPU locked state, thereby disabling interrupts and task dispatches. The features of a CPU locked state are outlined below.

(1) No task scheduling is performed during a CPU locked state.
(2) No external interrupts are accepted unless their priority levels are higher than the kernel interrupt mask level defined in the configurator.
(3) Only the following service calls can be invoked from a CPU locked state. If any other service calls are invoked, operation of the service call cannot be guaranteed.
 * ext_tsk
 * loc_cpu, iloc_cpu
 * unl_cpu, iunl_cpu
 * sns_ctx
 * sns_loc
 * sns_dsp
 * sns_dpn

The system is freed from a CPU locked state by one of the following operations.

(a) Invocation of the unl_cpu or iunl_cpu service call
(b) Invocation of the ext_tsk service call

Transitions between CPU locked and CPU unlocked states occur only when the loc_cpu, iloc_cpu, unl_cpu, iunl_cpu, or ext_tsk service call is invoked. The system must always be in a CPU unlocked state when the interrupt handler or the time event handler is terminated. If either handler terminates while the system is in a CPU locked state, handler operation cannot be guaranteed. Note that the system is always in a CPU unlocked state when these handlers start.

Invoking this service call again while the system is already in a CPU locked state does not cause an error, in which case task queuing is not performed, however.

If this service call is to be issued from task context, use loc_cpu; if issued from non-task context, use iloc_cpu.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include “kernel_id.h”
  void task()
  {
        :
      loc_cpu();
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB       task
task:
          :
        loc_cpu
          :
```

unl_cpu	Unlock the CPU
iunl_cpu	Unlock the CPU (handler only)

[[C Language API]]

```
ER ercd = unl_cpu();
ER ercd = iunl_cpu();
```

- **Parameters**

None

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
unl_cpu
iunl_cpu
```

- **Parameters**

None

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code

[[Error code]]

None

[[Functional description]]

This service call frees the system from a CPU locked state that was set by the loc_cpu or iloc_cpu service call. If the unl_cpu service call is issued from a dispatching enabled state, task scheduling is performed. If the system was put into a CPU locked state by invoking iloc_cpu within an interrupt handler, the system must always be placed out of a CPU locked state by invoking iunl_cpu before it returns from the interrupt handler.

The CPU locked state and the dispatching disabled state are managed independently of each other. Therefore, the system cannot be freed from a dispatching disabled state by the unl_cpu or iunl_cpu service call unless the ena_dsp service call is used.

If this service call is to be issued from task context, use unl_cpu; if issued from non-task context, use iunl_cpu.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
        :
      unl_cpu();
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
           :
        unl_cpu
           :
```

dis_dsp Disable dispatching

[[C Language API]]

```
ER ercd = dis_dsp();
```

- **Parameters**

None

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
dis_dsp
```

- **Parameters**

None

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code

[[Error code]]

None

[[Functional description]]

This service call places the system into a dispatching disabled state. The features of a dispatching disabled state are outlined below.

(1) Since task scheduling is not performed anymore, no tasks other than the issuing task itself will be placed into RUNNING state.
(2) Interrupts are accepted.
(3) No service calls can be invoked that will place tasks into WAITING state.

If one of the following operations is performed during a dispatching disabled state, the system status returns to a task execution state.

(a) Invocation of the ena_dsp service call
(b) Invocation of the ext_tsk service call

Transitions between dispatching disabled and dispatching enabled states occur only when the dis_dsp, ena_dsp, or ext_tsk service call is invoked.

Invoking this service call again while the system is already in a dispatching disabled state does not cause an error, in which case task queuing is not performed, however.

This service call can be issued only from task context. It cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
        :
      dis_dsp();
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB       task
task:
           :
        dis_dsp
           :
```

ena_dsp Enables dispatching

[[C Language API]]

```
ER ercd = ena_dsp();
```

- **Parameters**

None

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
ena_dsp
```

- **Parameters**

None

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code

[[Error code]]

None

[[Functional description]]

This service call frees the system from a dispatching disabled state that was set by the dis_dsp service call. As a result, task scheduling is resumed when the system has entered a task execution state.

Invoking this service call from a task execution state does not cause an error, in which case task queuing is not performed, however.

This service call can be issued only from task context. It cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
#include <itron.h>
#include <kernel.h>
#include “kernel_id.h”
void task()
{
      :
    ena_dsp();
      :
}
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
            :
        ena_dsp
            :
```

sns_ctx Reference context

[[C Language API]]

```
BOOL state = sns_ctx();
```

- **Parameters**

None

- **Return Parameters**

BOOL	state	TRUE: Non-task context FALSE: Task context

[[Assembly language API]]

```
.include mr100.inc
sns_ctx
```

- **Parameters**

None

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	TRUE:Non-Task context FALSE: Task context

[[Error code]]

None

[[Functional description]]

This service call returns TRUE when it is invoked from non-task context, or returns FALSE when invoked from task context. This service call can also be invoked from a CPU locked state.

[[Example program statement]]

<<Example statement in C language>>

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
  BOOL stat;
     :
   stat = sns_ctx();
     :
}
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
        sns_ctx
          :
```

sns_loc Reference CPU state

[[C Language API]]

```
BOOL state = sns_loc();
```

- **Parameters**

None

- **Return Parameters**

BOOL	state	TRUE: CPU locked state FALSE: CPU unlocked state

[[Assembly language API]]

```
.include mr100.inc
sns_loc
```

- **Parameters**

None

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	TRUE: CPU locked state FALSE:CPUCPU unlocked state

[[Error code]]

None

[[Functional description]]

This service call returns TRUE when the system is in a CPU locked state, or returns FALSE when the system is in a CPU unlocked state. This service call can also be invoked from a CPU locked state.

[[Example program statement]]

<<Example statement in C language>>

```
#include <itron.h>
#include <kernel.h>
#include “kernel_id.h”
void task()
{
    BOOL stat;
        :
      stat = sns_loc();
        :
}
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
        sns_loc
          :
```

sns_dsp Reference dispatching state

[[C Language API]]

```
BOOL state = sns_dsp();
```

- **Parameters**

None

- **Return Parameters**

BOOL	state	TRUE: Dispatching disabled state FALSE: Dispatching enabled state

[[Assembly language API]]

```
.include mr100.inc
sns_dsp
```

- **Parameters**

None

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	TRUE: Dispatching disabled state FALSE: Dispatching enabled state

[[Error code]]

None

[[Functional description]]

This service call returns TRUE when the system is in a dispatching disabled state, or returns FALSE when the system is in a dispatching enabled state. This service call can also be invoked from a CPU locked state.

[[Example program statement]]

<<Example statement in C language>>

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
    BOOL stat;
        :
      stat = sns_dsp();
        :
}
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
            :
        sns_dsp
            :
```

sns_dpn Reference dispatching pending state

[[C Language API]]

```
BOOL state = sns_dpn();
```

- **Parameters**

None

- **Return Parameters**

BOOL	state	TRUE: Dispatching pending state FALSE: Not dispatching pending state

[[Assembly language API]]

```
.include mr100.inc
sns_dpn
```

- **Parameters**

None

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	TRUE: Dispatching pending state FALSE: Not dispatching pending state

[[Error code]]

None

[[Functional description]]

This service call returns TRUE when the system is in a dispatching pending state, or returns FALSE when the system is not in a dispatching pending state. More specifically, FALSE is returned when all of the following conditions are met; otherwise, TRUE is returned.

(1) The system is not in a dispatching pending state.
(2) The system is not in a CPU locked state.
(3) The object made pending is a task.

This service call can also be invoked from a CPU locked state. It returns TRUE when the system is in a dispatching disabled state, or returns FALSE when the system is in a dispatching enabled state.

[[Example program statement]]

<<Example statement in C language>>

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task()
{
    BOOL stat;
        :
       stat = sns_dpn();
        :
}
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
            :
        sns_dpn
            :
```

5.13 Interrupt Management Function

Table 5.24 List of Interrupt Management Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	ret_int		Returns from an interrupt handler		O	O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls
- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

ret_int — Returns from an interrupt handler (when written in assembly language)

[[C Language API]]

This service call cannot be written in C language.[36]

[[Assembly language API]]

```
.include mr100.inc
ret_int
```

- **Parameters**

None

[[Error code]]

Not return to the interrupt handler that issued this service call.

[[Functional description]]

This service call performs the processing necessary to return from an interrupt handler. Depending on return processing, it activates the scheduler to switch tasks from one to another.

If this service call is executed in an interrupt handler, task switching does not occur, and task switching is postponed until the interrupt handler terminates.

However, if the ret_int service call is issued from an interrupt handler that was invoked from an interrupt that occurred within another interrupt, the scheduler is not activated. The scheduler is activated for interrupts from a task only.

When writing this service call in assembly language, be aware that the service call cannot be issued from a subroutine that is invoked from an interrupt handler entry routine. Always make sure this service call is executed in the entry routine or entry function of an interrupt handler. For example, a program like the one shown below may not operate normally.

```
.include mr100.inc
/* NG */
.GLB intr
intr:
          jsr.b func
:
func:
          ret_int
```

Therefore, write the program as shown below.

```
.include mr100.inc
/* OK */
.GLB intr
intr:
          jsr.b func
          ret_int
func:
:
          rts
```

Make sure this service call is issued from only an interrupt handler. If issued from a cyclic handler, alarm handler, or a task, this service call may not operate normally.

[36] If the starting function of an interrupt handler is declared by #pragma INTHANDLER, the ret_int service call is automatically issued at the exit of the function.

5.14 System Configuration Management Function

Table 5.25 List of System Configuration Management Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	ref_ver	**[S]**	References version information	O		O	O	O	
2	iref_ver				O	O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls

- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

ref_ver Reference version information
iref_ver Reference version information (handler only)

[[C Language API]]

```
ER ercd = ref_ver( T_RVER *pk_rver );
ER ercd = iref_ver( T_RVER *pk_rver );
```

● **Parameters**

T_RVER *pk_rver Pointer to the packet to which version information is returned

Contents of pk_rver

typedef struct t_rver {				
UH	maker	0	2	Kernel manufacturer code
UH	prid	+2	2	Kernel identification number
UH	spver	+4	2	ITRON specification version number
UH	prver	+6	2	Kernel version number
UH	prno[4]	+8	2	Kernel product management information
} T_RVER;				

● **Return Parameters**

ER ercd Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
ref_ver   PK_VER
iref_ver  PK_VER
```

● **Parameters**

PK_VER Pointer to the packet to which version information is returned

● **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
A1	Pointer to the packet to which version information is returned

[[Error code]]

None

[[Functional description]]

This service call reads out information about the version of the currently executing kernel and returns the result to the area pointed to by pk_rver.

The following information is returned to the packet pointed to by pk_rver.

- ◆ **maker**
  The code H'0115 denoting Renesas Technology Corporation is returned.
- ◆ **prid**
  The internal identification code IDH'0014 of the M3T-MR100 is returned.
- ◆ **spver**
  The code H'5403 denoting that the kernel is compliant with μITRON Specification Ver 4.03.00 is returned.
- ◆ **prver**
  The code H'0100 denoting the version of the M3T-MR100/4 is returned.
- ◆ **prno**
  - prno[0]
    Reserved for future extension.
  - prno[1]
    Reserved for future extension.
  - prno[2]
    Reserved for future extension.
  - prno[3]
    Reserved for future extension.

If this service call is to be issued from task context, use ref_ver; if issued from non-task context, use iref_ver.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
      T_RVER    pk_rver;
      ref_ver( &pk_rver );
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
_ refver:       .blkb   16
task:
          :
        PUSHM   A1
        ref_ver #_refver
          :
```

5.15 Extended Function (Short Data Queue)

Specifications of the Short data queue function of MR100 are listed in Table 5.26. This function is outside the scope of μITRON 4.0 Specification.

Table 5.26 Specifications of the Short Data Queue Function

No.	Item	Content	
1	Data queue ID	1-255	
2	Capacity (data bytes) in data queue area	0-16383	
3	Data size	16 bits	
4	Data queue attribute	TA_TFIFO:	Waiting tasks enqueued in order of FIFO
		TA_TPRI:	Waiting tasks enqueued in order of priority

Table 5.27 List of Long Dataqueue Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	vsnd_dtq		Sends to short data queue	O		O		O	
2	vpsnd_dtq		Sends to short data queue (polling)		O	O	O	O	
3	vipsnd_dtq			O		O	O	O	
4	vtsnd_dtq		Sends to short data queue (with timeout)		O	O		O	
5	vfsnd_dtq		Forced sends to short data queue	O		O	O	O	
6	vifsnd_dtq			O		O	O	O	
7	vrcv_dtq		Receives from short data queue		O	O		O	
8	vprcv_dtq		Receives from short data queue (polling)	O		O	O	O	
9	viprcv_dtq			O		O	O	O	
10	vtrcv_dtq		Receives from short data queue (with timeout)		O	O		O	
11	vref_dtq		References short data queue status	O		O	O	O	
12	viref_dtq				O	O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls
- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

vsnd_dtq Send to Short data queue
vpsnd_dtq Send to Short data queue (polling)
vipsnd_dtq Send to Short data queue (polling, handler only)
vtsnd_dtq Send to Short data queue (with timeout)
vfsnd_dtq Forced send to Short data queue
vifsnd_dtq Forced send to Short data queue (handler only)

[[C Language API]]

```
ER ercd = vsnd_dtq( ID vdtqid, H data );
ER ercd = vpsnd_dtq( ID vdtqid, H data );
ER ercd = vipsnd_dtq( ID vdtqid, H data );
ER ercd = vtsnd_dtq( ID vdtqid, H data, TMO tmout );
ER ercd = vfsnd_dtq( ID vdtqid, H data );
ER ercd = vifsnd_dtq( ID vdtqid, H data );
```

● **Parameters**

ID	vdtqid	ID number of the Short data queue to which transmitted
TMO	tmout	Timeout value(tsnd_dtq)
H	data	Data to be transmitted

● **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr100.inc
vsnd_dtq        VDTQID, DTQDATA
visnd_dtq       VDTQID, DTQDATA
vpsnd_dtq       VDTQID, DTQDATA
vipsnd_dtq      VDTQID, DTQDATA
vtsnd_dtq       VDTQID, DTQDATA,TMO
vfsnd_dtq       VDTQID, DTQDATA
vifsnd_dtq      VDTQID, DTQDATA
```

● **Parameters**

VDTQID	ID number of the Short data queue to which transmitted
DTQDATA	Data to be transmitted
TMO	Timeout value(tsnd_dtq)

● **Register contents after service call is issued**

vsnd_dtq,vpsnd_dtq,vipsnd_dtq,vfsnd_dtq,vifsnd_dtq

Register name	Content after service call is issued
R0	Error code
R1	Data to be transmitted
R2	ID number of the Short data queue to which transmitted

vtsnd_dtq

Register name	Content after service call is issued
R0	Error code
R1	Data to be transmitted
R2	ID number of the Short data queue to which transmitted
R6R4	Timeout value

[[Error code]]

E_RLWAI	Forced release from waiting
E_TMOUT	Polling failure or timeout or timed out
E_ILUSE	Service call improperly used (vfsnd_dtq or vifsnd_dtq is issued for a Short data queue whose dtqcnt = 0)
EV_RST	Released from a wait state by clearing of the Short data queue area

[[Functional description]]

This service call sends the signed 2-byte data indicated by data to the Short data queue indicated by vdtqid. If any task is kept waiting for reception in the target Short data queue, the data is not stored in the Short data queue and instead sent to the task at the top of the reception waiting queue, with which the task is released from the reception wait state.

On the other hand, if vsnd_dtq or vtsnd_dtq is issued for a Short data queue that is full of data, the task that issued the service call goes from RUNNING state to a data transmission wait state, and is enqueued in a transmission waiting queue, kept waiting for the Short data queue to become available. In that case, if the attribute of the specified Short data queue is TA_TFIFO, the task is enqueued in order of FIFO; if TA_TPRI, the task is enqueued in order of priority. For vpsnd_dtq and vipsnd_dtq, the task returns immediately and responds to the call with the error code E_TMOUT.

For the vtsnd_dtq service call, specify a wait time for tmout in ms units. The values specified for tmout must be within (0x7FFFFFFF-time tick value). If any value exceeding this limit is specified, the service call may not operate correctly. If TMO_POL=0 is specified for tmout, it means specifying 0 as a timeout value, in which case the service call operates the same way as vpsnd_dtq. Furthermore, if specified as tmout=TMO_FEVR(–1), it means specifying an infinite wait, in which case the service call operates the same way as vsnd_dtq.

If there are no tasks waiting for reception, nor is the Short data queue area filled, the transmitted data is stored in the Short data queue.

The task placed into a wait state by execution of the vsnd_dtq or vtsnd_dtq service call is released from WAITING state in the following cases:

- ◆ **When the vrcv_dtq, vtrcv_dtq, vprcv_dtq, or viprcv_dtq service call is issued before the tmout time elapses, with task-awaking conditions thereby satisfied**
  The error code returned in this case is E_OK.
- ◆ **When the first time tick occurred after tmout elapsed while task-awaking conditions remain unsatisfied**
  The error code returned in this case is E_TMOUT.
- ◆ **When the task is forcibly released from WAITING state by the rel_wai or irel_wai service call issued from another task or a handler**
  The error code returned in this case is E_RLWAI.
- ◆ **When the target Short data queue being waited for is initialized by the vrst_vdtq service call issued from another task**
  The error code returned in this case is EV_RST.

For vfsnd_dtq and vifsnd_dtq, the data at the top of the Short data queue or the oldest data is removed, and the transmitted data is stored at the tail of the Short data queue. If the Short data queue area is not filled with data, vfsnd_dtq and vifsnd_dtq operate the same way as vsnd_dtq. If dtqcnt = 0 ,there is no task in the wait queue and vfsnd_dtq or vifsnd_dtq service call is issued, error code E_ILUSE will be returned.

If this service call is to be issued from task context, use vsnd_dtq,vtsnd_dtq,vpsnd_dtq,vfsnd_dtq; if issued from non-task context, use vipsnd_dtq,vifsnd_dtq.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  H data[10];
  void task(void)
  {
                :
      if( vsnd_dtq( ID_dtq, data[0]) == E_RLWAI ){
        error("Forced released\n");
    }
                :
      if( vpsnd_dtq( ID_dtq, data[1])== E_TMOUT ){
        error("Timeout\n");
    }
                :
      if( vtsnd_dtq( ID_dtq, data[2], 10 ) != E_ TMOUT ){
        error("Timeout \n");
    }
                :
      if( vfsnd_dtq( ID_dtq, data[3]) != E_OK ){
        error("error\n");
    }
                :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
_g_dtq: .WORD   1234H
task:
          :
    PUSH.W      R1
    PUSH.W      R2
    PUSHM       R6R4
    vtsnd_dtq   #ID_DTQ1,_g_dtq,#100
          :
    PUSH.W      R1
    PUSH.W      R2
    vpsnd_dtq   #ID_DTQ2,#0FFFFH
          :
    PUSH.W      R1
    PUSH.W      R2
    vfsnd_dtq   #ID_DTQ3,#0ABCDH
          :
```

vrcv_dtq	**Receive from Short data queue**
vprcv_dtq	**Receive from Short data queue (polling)**
viprcv_dtq	**Receive from Short data queue (polling,handler only)**
vtrcv_dtq	**Receive from Short data queue (with timeout)**

[[C Language API]]

```
ER ercd = vrcv_dtq( ID dtqid, H *p_data );
ER ercd = vprcv_dtq( ID dtqid, H *p_data );
ER ercd = viprcv_dtq( ID dtqid, H *p_data );
ER ercd = vtrcv_dtq( ID dtqid, H *p_data, TMO tmout );
```

● Parameters

ID	vdtqid	ID number of the Short data queue from which to receive
TMO	tmout	Timeout value(vtrcv_dtq)
H	*p_data	Pointer to the start of the area in which received data is stored

● Return Parameters

ER	ercd	Terminated normally (E_OK) or error code
H	*p_data	Pointer to the start of the area in which received data is stored

[[Assembly language API]]

```
.include mr100.inc
vrcv_dtq        VDTQID
vprcv_dtq       VDTQID
viprcv_dtq      VDTQID
vtrcv_dtq       VDTQID,TMO
```

● Parameters

VDTQID	ID number of the Short data queue from which to receive
TMO	Timeout value(trcv_dtq)

● Register contents after service call is issued

vrcv_dtq,vprcv_dtq,viprcv_dtq

Register name	Content after service call is issued
R0	Error code
R1	Received data
R2	ID number of the Short data queue from which to receive

vtrcv_dtq

Register name	Content after service call is issued
R0	Error code
R1	Received data
R2	ID number of the Short data queue from which to receive
R6R4	Timeout value

[[Error code]]

E_RLWAI	Forced release from waiting
E_TMOUT	Polling failure or timeout or timed out

[[Functional description]]

This service call receives data from the Short data queue indicated by vdtqid and stores the received data in the area pointed to by p_data. If data is present in the target Short data queue, the data at the top of the queue or the oldest data is received. This results in creating a free space in the Short data queue area, so that a task enqueued in a transmission waiting queue is released from WAITING state, and starts sending data to the Short data queue area.

If no data exist in the Short data queue and there is any task waiting to send data (i.e., data bytes in the Short data queue area = 0), data for the task at the top of the data transmission waiting queue is received. As a result, the task kept waiting to send that data is released from WAITING state.

On the other hand, if vrcv_dtq or vtrcv_dtq is issued for the Short data queue which has no data stored in it, the task that issued the service call goes from RUNNING state to a data reception wait state, and is enqueued in a data reception waiting queue. At this time, the task is enqueued in order of FIFO. For the vprcv_dtq and viprcv_dtq service calls, the task returns immediately and responds to the call with the error code E_TMOUT.

For the vtrcv_dtq service call, specify a wait time for tmout in ms units. The values specified for tmout must be within 0x7FFFFFFF. If any value exceeding this limit is specified, the service call may not operate correctly. If TMO_POL=0 is specified for tmout, it means specifying 0 as a timeout value, in which case the service call operates the same way as vprcv_dtq. Furthermore, if specified as tmout=TMO_FEVR(–1), it means specifying an infinite wait, in which case the service call operates the same way as vrcv_dtq.

The task placed into a wait state by execution of the vrcv_dtq or vtrcv_dtq service call is released from the wait state in the following cases:

- **When the vrcv_dtq, vtrcv_dtq, vprcv_dtq, or viprcv_dtq service call is issued before the tmout time elapses, with task-awaking conditions thereby satisfied**
  The error code returned in this case is E_OK.
- **When the first time tick occurred after tmout elapsed while task-awaking conditions remain unsatisfied**
  The error code returned in this case is E_TMOUT.
- **When the task is forcibly released from WAITING state by the rel_wai or irel_wai service call issued from another task or a handler**
  The error code returned in this case is E_RLWAI.

If this service call is to be issued from task context, use vrcv_dtq,vtrcv_dtq,vprcv_dtq; if issued from non-task context, use viprcv_dtq.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
     H data;
                :
       if( vrcv_dtq( ID_dtq, &data ) != E_RLWAI )
           error("forced wakeup\n");
                :
       if( vprcv_dtq( ID_dtq, &data ) != E_TMOUT )
           error("Timeout\n");
                :
       if( vtrcv_dtq( ID_dtq, &data, 10 ) != E_TMOUT )
           error("Timeout\n");
                :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    PUSHM       R6R4
    vtrcv_dtq   #ID_DTQ1,#TMO_POL
                   :
    PUSH.W      R2
    vprcv_dtq   #ID_DTQ2
          :
    PUSH.W      R2
    vrcv_dtq    #ID_DTQ2
          :
```

vref_dtq — Reference Short data queue status
viref_dtq — Reference Short data queue status (handler only)

[[C Language API]]

```
ER ercd = vref_dtq( ID vdtqid, T_RDTQ *pk_rdtq );
ER ercd = viref_dtq( ID vdtqid, T_RDTQ *pk_rdtq );
```

● **Parameters**

ID	vdtqid	ID number of the target Short data queue
T_RDTQ	*pk_rdtq	Pointer to the packet to which Short data queue status is returned

● **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code
T_RDTQ	*pk_rdtq	Pointer to the packet to which Short data queue status is returned

Contents of pk_rdtq

```
typedef    struct    t_rdtq{
      ID       stskid     +0    2    Transmission waiting task ID
      ID       wtskid     +2    2    Reception waiting task ID
      UINT     sdtqcnt    +4    4    Data bytes contained in Short data queue
} T_RDTQ;
```

[[Assembly language API]]

```
.include mr100.inc
vref_dtq  VDTQID, PK_RDTQ
viref_dtq VDTQID, PK_RDTQ
```

● **Parameters**

VDTQID	ID number of the target Short data queue
PK_RDTQ	Pointer to the packet to which Short data queue status is returned

● **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	ID number of the target Short data queue
A1	Pointer to the packet to which Short data queue status is returned

[[Error code]]

None

[[Functional description]]

This service call returns various statuses of the Short data queue indicated by vdtqid.

- ◆ **stskid**
  Returned to stskid is the ID number of the task at the top of a transmission waiting queue (the next task to be de-queued). If no tasks are kept waiting, TSK_NONE is returned.
- ◆ **wtskid**
  Returned to wtskid is the ID number of the task at the top of a reception waiting queue (the next task to be de-queued). If no tasks are kept waiting, TSK_NONE is returned.
- ◆ **sdtqcnt**
  Returned to sdtqcnt is the number of data bytes stored in the Short data queue area.

If this service call is to be issued from task context, use ref_dtq; if issued from non-task context, use iref_dtq.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include “kernel_id.h”
  void task()
  {
       T_RDTQ rdtq;
       ER ercd;
            :
       ercd = vref_dtq( ID_DTQ1, &rdtq );
            :
  }
```

<<Example statement in assembly language>>

```
     .include mr100.inc
_ refdtq:       .blkb   8
     .GLB       task
task:
            :
         PUSH.W  R2
         PUSH.L  A1
         vref_dtq        #ID_DTQ1,#_refdtq
            :
```

5.16 Extended Function (Reset Function)

This function initializes the content of an object. This function is outside the scope of μITRON 4.0 Specification.

Table 5.28 List of Reset Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	vrst_dtq		Clear data queue area	O		O	O	O	
2	vrst_vdtq		Clear Short data queue area	O		O	O	O	
3	vrst_mbx		Clear mailbox area	O		O	O	O	
4	vrst_mpf		Clear fixed-size memory pool area	O		O	O	O	
5	vrst_mpl		Clear variable-size memory pool area	O		O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls
- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

vrst_dtq Clear data queue area

[[C Language API]]

```
ER ercd = vrst_dtq( ID dtqid );
```

- **Parameters**

ID	dtqid	Data queue ID to be cleared

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
vrst_dtq  DTQID
```

- **Parameters**

DTQID	Data queue ID to be cleared

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	Data queue ID to be cleared

[[Error code]]

None

[[Functional description]]

This service call clears the data stored in the data queue indicated by dtqid. If the data queue area has no more areas to be added and tasks are enqueued in a data transmission waiting queue, all of the tasks enqueued in the data transmission waiting queue are released from WAITING state. Furthermore, the error code EV_RST is returned to the tasks that have been released from WAITING state.

Even when the number of data queues defined is 0, all of the tasks enqueued in a data transmission waiting queue are released from WAITING state.

This service call can be issued only from task context. It cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task1(void)
  {
        :
      vrst_dtq( ID_dtq1 );
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    vrst_dtq    #ID_DTQ1
          :
```

vrst_vdtq Clear Short data queue area

[[C Language API]]

```
ER ercd = vrst_vdtq( ID vdtqid );
```

- **Parameters**

ID	vdtqid	Short data queue ID to be cleared

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
vrst_vdtq VDTQID
```

- **Parameters**

VDTQID	Short data queue ID to be cleared

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	Short data queue ID to be cleared

[[Error code]]

None

[[Functional description]]

This service call clears the data stored in the Short data queue indicated by vdtqid. If the Short data queue area has no more areas to be added and tasks are enqueued in a data transmission waiting queue, all of the tasks enqueued in the data transmission waiting queue are released from WAITING state. Furthermore, the error code EV_RST is returned to the tasks that have been released from WAITING state.

Even when the number of Short data queues defined is 0, all of the tasks enqueued in a data transmission waiting queue are released from WAITING state.

This service call can be issued only from task context. It cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include “kernel_id.h”
  void task1(void)
  {
        :
      vrst_vdtq( ID_vdtq1 );
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    vrst_vdtq   #ID_VDTQ1
          :
```

vrst_mbx Clear mailbox area

[[C Language API]]

```
ER ercd = vrst_mbx( ID mbxid );
```

- **Parameters**

ID	mbxid	Mailbox ID to be cleared

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
vrst_mbx  MBXID
```

- **Parameters**

MBXID	Mailbox ID to be cleared

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	Mailbox ID to be cleared

[[Error code]]

None

[[Functional description]]

This service call clears the messages stored in the mailbox indicated by mbxid.
This service call can be issued only from task context. It cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task1(void)
  {
        :
      vrst_mbx( ID_mbx1 );
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    vrst_mbx    #ID_MBX1
          :
```

vrst_mpf Clear fixed-size memory pool area

[[C Language API]]

```
ER ercd = vrst_mpf( ID mpfid );
```

- **Parameters**

ID	mpfid	Fixed-size memory pool ID to be cleared

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
vrst_mpf  MPFID
```

- **Parameters**

MPFID	Fixed-size memory pool ID to be cleared

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	Fixed-size memory pool ID to be cleared

[[Error code]]

None

[[Functional description]]

This service call initializes the fixed-size memory pool indicated by mpfid. If tasks are enqueued in a memory block waiting queue, all of the tasks enqueued in the memory block waiting queue are released from WAITING state. Furthermore, the error code EV_RST is returned to the tasks that have been released from WAITING state.

This service call can be issued only from task context. It cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task1(void)
{
        :
    vrst_mpf( ID_mpf1 );
        :
}
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    vrst_mpf    #ID_MPF1
          :
```

vrst_mpl Clear variable-size memory pool area

[[C Language API]]

```
ER ercd = vrst_mpl( ID mplid );
```

- **Parameters**

ID	mplid	Variable-size memory pool ID to be cleared

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr100.inc
vrst_mpl  MPLID
```

- **Parameters**

MPLID Variable-size memory pool ID to be cleared

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R2	Variable-size memory pool ID to be cleared

[[Error code]]

None

[[Functional description]]

This service call initializes the variable-size memory pool indicated by mplid.

This service call can be issued only from task context. It cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
void task1(void)
{
        :
     vrst_mpl( ID_mpl1 );
        :
}
```

<<Example statement in assembly language>>

```
    .include mr100.inc
    .GLB        task
task:
          :
    PUSH.W      R2
    vrst_mpl    #ID_MPL1
          :
```

6. Applications Development Procedure Overview

6.1 Overview

Application programs for MR100 should generally be developed following the procedure described below.

1. **Generating a project**

   When using HEW[37], create a new project using MR100 on HEW.

2. **Coding the application program**

   Write the application program in code form using C or assembly language. If necessary, correct the sample startup program (crt0mr.a30) and section definition file (c_sec.inc or asm_sec.inc).

3. **Creating a configuration file**

   Create a configuration file which has defined in it the task entry address, stack size, etc. by using an editor.

   The GUI configurator available for MR100 may be used to create a configuration file.

4. **Executing the configurator**

   From the configuration file, create system data definition files (sys_rom.inc, sys_ram.inc), include files (mr100.inc, kernel_id.h).

5. **System generation**

   Execute the make[38] command or execute build on HEW to generate a system.

6. **Writing to ROM**

   Using the ROM programming format file created, write the finished program file into the ROM. Or load it into the debugger to debug.

Figure 6.1 shows a detailed flow of system generation.

[37] It is abbreviation of High-performance Embedded Workshop.

[38] The make command comes the UNIX standard and UNIX compatible.

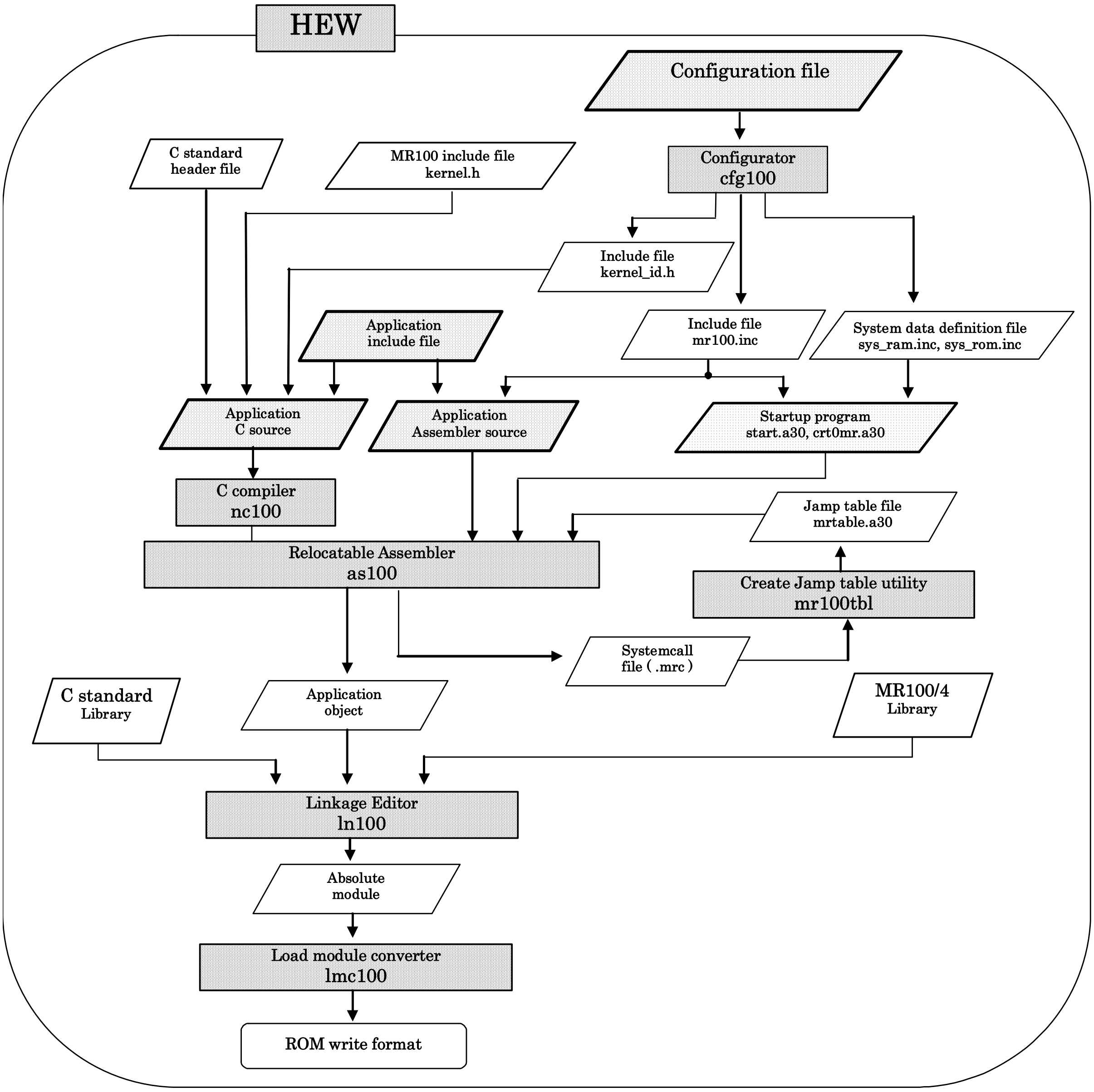

Figure 6.1 MR100 System Generation Detail Flowchart

6.2 Development Procedure Example

This chapter outlines the development procedures on the basis of a typical MR100 application example.

6.2.1 Applications Program Coding

Figure 6.2 shows a program that simulates laser beam printer operations. Let us assume that the file describing the laser beam printer simulation program is named lbp.c. This program consists of the following three tasks and one interrupt handler.

- Main Task
- Image expansion task
- Printer engine task
- Centronics interface interrupt handler

This program uses the following MR100 library functions.

- sta_tsk()
  Starts a task. Give the appropriate ID number as the argument to select the task to be activated. When the kernel_id.h file, which is generated by the configurator, is included, it is possible to specify the task by name (character string).[39]

- wai_flg()
  Waits until the eventflag is set up. In the example, this function is used to wait until one page of data is entered into the buffer via the Centronics interface.

- wup_tsk()
  Wakes up a specified task from the WAITING state. This function is used to start the printer engine task.

- slp_tsk()
  Causes a task in the RUNNING state to enter the WAITING state. In the example, this function is used to make the printer engine task wait for image expansion.

- iset_flg()
  Sets the eventflag. In the example, this function is used to notify the image expansion task of the completion of one-page data input.

[39] The configurator converts the ID number to the associated name(character string) in accordance with the information entered int the configuration file.

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void main() /* main task */
{
    printf("LBP start simulation \n");
    sta_tsk(ID_idle,1);     /* activate idle task */
    sta_tsk(ID_image,1);    /* activate image expansion task */
    sta_tsk(ID_printer,1);  /* activate printer engine task */
}
void image() /* activate image expansion task */
{
    while(1){
            wai_flg(ID_pagein,waiptn,TWF_ANDW, &flgptn);/* wait for 1-page input */

            printf(" bit map expansion processing \n");
            wup_tsk(ID_printer);    /* wake up printer engine task */
    }
}
void printer() /* printer engine task */
{
    while(1){
            slp_tsk();
            printf(" printer engine operation \n");
    }
}
void sent_in() /* Centronics interface handler */
{


    /* Process input from Centronics interface */
    if ( /* 1-page input completed */ )
            iset_flg(ID_pagein,setptn);
}
```

Figure 6.2 Program Example

6.2.2 Configuration File Preparation

Create a configuration file which has defined in it the task entry address, stack size, etc. Use of the GUI configurator available for MR100 helps to create a configuration file easily without having to learn how to write it.

Figure 6.3 Configuration File Example

shows an example configuration file for a laser beam printer simulation program (filename "lbp.cfg").

```
// System Definition
 system{
    stack_size              = 1024;
    priority                = 5;
    system_IPL              = 4;
    tick_nume               = 10;
 };
 //System Clock Definition
 clock{
    mpu_clock               = 20MHz;
    timer                   = A0;
    IPL                     = 4;
};
 //Task Definition
 task[1]{
    name                    = ID_main;
    entry_address           = main();
    stack_size              = 512;
    priority                = 1;
    initial_start           = ON;
 };
 task[2]{
    name                    = ID_image;
    entry_address           = image();
    stack_size              = 512;
    priority                = 2;
 };
 task[3]{
    name                    = ID_printer;
    entry_address           = printer();
    stack_size              = 512;
    priority                = 4;
 };
 task[4]{
    name                    = ID_idle;
    entry_address           = idle();
    stack_size              = 256;
    priority                = 5;
 };
//Eventflag Definition
 flag[1]{
     name                   = pagein;
 };
 //Interrupt Vector Definition
 interrupt_vector[0x23]{
    os_int                  = YES;
    entry_address           = sent_in();
 };
```

Figure 6.3 Configuration File Example

6.2.3 Configurator Execution

When using HEW, select "Build all," which enables the user to execute the procedures described in 6.2.3, "Executing the Configurator," and 6.2.4, "System Generation."

Execute the configurator cfg100 to generate system data definition files (sys_rom.inc, sys_ram.inc), include files (mr100.inc, kernel_id.h), and a system generation procedure description file (makefile) from the configuration file.

```
A> cfg100  -v  lbp.cfg

MR100 system configurator V.1.00.18
Copyright 2003,2005 RENESAS TECHNOLOGY CORPORATION
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED.
MR100 version ==> V.1.01 Release 01

A>
```

Figure 6.4 Configurator Execution

6.2.4 System generation

Execute the make command to generate the system.

```
A> make -f  makefile
as100 -F -Dtest=1 crt0mr.a30
nc100 -c task.c
ln100 @ln100.sub

A>
```

Figure 6.5 System Generation

6.2.5 Writing ROM

Using the lmc30 load module converter, convert the absolute module file into a ROM writable format and then write it into ROM. Or read the file into the debugger and debug it.

7. Detailed Applications

7.1 Program Coding Procedure in C Language

7.1.1 Task Description Procedure

1. **Describe the task as a function.**

   To register the task for the MR100, enter its function name in the configuration file. When, for instance, the function name "task()" is to be registered as the task ID number 3, proceed as follows.

```
task[3]{
        name            = ID_task;

        entry_address   = task();

        stack_size      = 100;

        priority        = 3;

};
```

2. **At the beginning of file, be sure to include "itron.h",”kernel.h” which is in system directory as well as "kernel_id.h" which is in the current directory. That is, be sure to enter the following two lines at the beginning of file.**

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
```

3. **No return value is provided for the task start function. Therefore, declare the task start function as a void function.**
4. **A function that is declared to be static cannot be registered as a task.**
5. **It isn't necessary to describe ext_tsk() at the exit of task start function.[40]If you exit the task from the subroutine in task start function, please describe ext_tsk() in the subroutine.**
6. **It is also possible to describe the task startup function, using the infinite loop.**

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void task(void)
{

        /* process */

}
```

Figure 7.1 Example Infinite Loop Task Described in C Language

[40] The task is ended by ext_tsk() automatically if #pramga TASK is declared in the MR100. Similarly, it is ended by ext_tsk when returned halfway of the function by return sentence.

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void task(void)
{
        for(;;){
                /* process */

        }
}
```

Figure 7.2 Example Task Terminating with ext_tsk() Described in C Language

7. **To specify a task, use the string written in the task definition item “name” of the configuration file.[41]**

```
wup_tsk(ID_main);
```

8. **To specify an event flag, semaphore, or mailbox, use the respective strings defined in the configuration file.**

   For example, if an event flag is defined in the configuration file as shown below,

```
flag[1]{
        name    = ID_abc;
};
```

   To designate this eventflag, proceed as follows.

```
set_flg(ID_abc,&setptn);
```

9. **To specify a cyclic or alarm handler, use the string written in the cyclic or alarm handler definition item “name” of the configuration file.**

```
sta_cyc(ID_cyc);
```

10. **When a task is reactivated by the sta_tsk() service call after it has been terminated by the ter_tsk() service call, the task itself starts from its initial state.[42] However, the external variable and static variable are not automatically initialized when the task is started. The external and static variables are initialized only by the startup program (crt0mr.a30), which actuates before MR100 startup.**

11. **The task executed when the MR100 system starts up is setup.**

12. **The variable storage classification is described below.**

    The MR100 treats the C language variables as indicated in Table 7.1 C Language Variable Treatment.

Table 7.1 C Language Variable Treatment

Variable storage class	Treatment
Global Variable	Variable shared by all tasks
Non-function static variable	Variable shared by the tasks in the same file
Auto Variable Register Variable Static variable in function	Variable for specific task

7.1.2 Writing a Kernel (OS Dependent) Interrupt Handler

When describing the kernel interrupt handler in C language, observe the following precautions.

[41] The configurator generates the file “kernel_id.h” that is used to convert the ID number of a task into the string to be specified. This means that the #define declaration necessary to convert the string specified in the task definition item “name” into the ID number of the task is made in “kernel_id.h.” The same applies to the cyclic and alarm handlers.

[42] The task starts from its start function with the initial priority in a wakeup counter cleared state.

1. **Describe the kernel interrupt handler as a function [43]**
2. **Be sure to use the void type to declare the interrupt handler start function return value and argument.**
3. **At the beginning of file, be sure to include "itron.h",”kernel.h” which is in the system directory as well as "kernel_id.h" which is in the current directory.**
4. **<u>Do not use</u> the ret_int service call in the interrupt handler.[44]**
5. **The static declared functions can not be registered as an interrupt handler.**

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void inthand(void)

{
        /* process */

        iwup_tsk(ID_main);
}
```

Figure 7.3 Example of Kernel Interrupt Handler

7.1.3 Writing Non-kernel Interrupt Handler

When describing the non-kernel interrupt handler in C language, observe the following precautions.

1. **Be sure to declare the return value and argument of the interrupt handler start function as a void type.**
2. **No service call can be issued from a non-kernel interrupt handler.**
   <u>NOTE: If this restriction is not observed, the software may malfunction.</u>
3. **A function that is declared to be static cannot be registered as an interrupt handler.**
4. **If you want multiple interrupts to be enabled in a non-kernel interrupt handler, always make sure that the non-kernel interrupt handler is assigned a priority level higher than other kernel interrupt handlers.[45]**

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void inthand(void)
{
        /* process */
}
```

Figure 7.4 Example of Non-kernel Interrupt Handler

7.1.4 Writing Cyclic Handler/Alarm Handler

When describing the cyclic or alarm handler in C language, observe the following precautions.

[43] A configuration file is used to define the relationship between handlers and functions.
[44] When an kernel interrupt handler is declared with #pragma INTHANDLER ,code for the ret_int service call is automatically generated.
[45] If you want the non-kernel interrupt handler to be assigned a priority level lower than kernel interrupt handlers, change the description of the non-kernel interrupt handler to that of the kernel interrupt handler.

1. **Describe the cyclic or alarm handler as a function.**[46]
2. **Be sure to declare the return value and argument of the interrupt handler start function as a void type.**
3. **At the beginning of file, be sure to include "itron.h",”kernel.h” which is in the system directory as well as "kernel_id.h" which is in the current directory.**
4. **The static declared functions cannot be registered as a cyclic handler or alarm handler.**
5. **The cyclic handler and alarm handler are invoked by a subroutine call from a system clock interrupt handler.**

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void cychand(void)
{
        /*process */
}
```

Figure 7.5 Example Cyclic Handler Written in C Language

[46] The handler-to-function name correlation is determined by the configuration file.

7.2 Program Coding Procedure in Assembly Language

This section describes how to write an application using the assembly language.

7.2.1 Writing Task

This section describes how to write an application using the assembly language.

1. **Be sure to include "mr100.inc" at the beginning of file.**
2. **For the symbol indicating the task start address, make the external declaration.**[47]
3. **Be sure that an infinite loop is formed for the task or the task is terminated by the ext_tsk service call.**

```
        .INCLUDE mr100.inc ----- (1)
        .GLB    task    ----- (2)

task:
                ; process
        jmp   task       ----- (3)
```

Figure 7.6 Example Infinite Loop Task Described in Assembly Language

```
        .INCLUDE mr100.inc
        .GLB    task

task:
                ; process
        ext_tsk
```

Figure 7.7 Example Task Terminating with ext_tsk Described in Assembly Language

4. **The initial register values at task startup are indeterminate except the PC, SB, R0 and FLG registers.**
5. **To specify a task, use the string written in the task definition item “name” of the configuration file.**

```
        wup_tsk  #ID_task
```

6. **To specify an event flag, semaphore, or mailbox, use the respective strings defined in the configuration file.**

   For example, if a semaphore is defined in the configuration file as shown below,:

```
        semaphore[1]{
                name            = abc;
        };
```

   To specify this semaphore, write your specification as follows:

```
        sig_sem  #ID_abc
```

7. **To specify a cyclic or alarm handler, use the string written in the cyclic or alarm handler definition item “name” of the configuration file**

   For example, if you want to specify a cyclic handler "cyc," write your specification as follows:

```
        sta_cyc  #ID_cyc
```

[47] Use the .GLB pseudo-directive

8. Set a task that is activated at MR100 system startup in the configuration file [48]

7.2.2 Writing Kernel Interrupt Handler

When describing the kernel interrupt handler in assembly language, observe the following precautions

1. **At the beginning of file, be sure to include "mr100.inc" which is in the system directory.**
2. **For the symbol indicating the interrupt handler start address, make the external declaration(Global declaration).[49]**
3. **Make sure that the registers used in a handler are saved at the entry and are restored after use.**
4. **Return to the task by ret_int service call.**

```
        .INCLUDE mr100.inc                 ------(1)
        .GLB    inth                       ------(2)

inth:
        ;  Registers used are saved to a stack   ------(3)
        iwup_tsk  #ID_task1
                :
                process
                :

        ;  Registers used are restored     ------(3)

        ret_int                            ------(4)
```

Figure 7.8 Example of kernel(OS-depend) interrupt handler

7.2.3 Writing Non-kernel Interrupt Handler

1. **For the symbol indicating the interrupt handler start address, make the external declaration (public declaration).**
2. **Make sure that the registers used in a handler are saved at the entry and are restored after use.**
3. **Be sure to end the handler by REIT instruction.**
4. **No service calls can be issued from a non-kernel interrupt handler.**
   NOTE: If this restriction is not observed, the software may malfunction.

5. **If you want multiple interrupts to be enabled in a non-kernel interrupt handler, always make sure that the non-kernel interrupt handler is assigned a priority level higher than other non-kernel interrupt handlers.[50]**

```
        .GLB    inthand                    ----- (1)

inthand:
        ; Registers used are saved to a stack    ----- (2)
        ; interrupt process
        ; Registers used are restored      ----- (2)
        REIT                               ----- (3)
```

Figure 7.9 Example of Non-kernel Interrupt Handler of Specific Level

7.2.4 Writing Cyclic Handler/Alarm Handler

When describing the cyclic or alarm handler in Assembly Language, observe the following precautions.

[48] The relationship between task ID numbers and tasks(program) is defined in the configuration file.
[49] Use the .GLB peudo-directive.
[50] If you want the non-kernel interrupt handler to be assigned a priority level lower than kernel interrupt handlers, change the description of the non-kernel interrupt handler to that of the kernel interrupt handler.

1. **At the beginning of file, be sure to include "mr100.inc" which is in the system directory.**
2. **For the symbol indicating the handler start address, make the external declaration.**[51]
3. **Always use the RTS instruction (subroutine return instruction) to return from cyclic handlers and alarm handlers.**

For examples:

```
.INCLUDE        mr100.inc       ----- (1)
.GLB            cychand         ----- (2)

cychand:
                :
        ; handler process
                :

        rts                     ----- (3)
```

Figure 7.10 Example Handler Written in Assembly Language

[51] Use the .GLB pseudo-directive.

7.3 Modifying MR100 Startup Program

MR100 comes with two types of startup programs as described below.

- start.a30
  This startup program is used when you created a program using the assembly language.

- crt0mr.a30
  This startup program is used when you created a program using the C language.

  This program is derived from "start.a30" by adding an initialization routine in C language.

The startup programs perform the following:

- Initialize the processor after a reset.
- Initialize C language variables (crt0mr.a30 only).
- Set the system timer.
- Initialize MR100's data area.

Copy these startup programs from the directory indicated by environment variable "LIB100" to the current directory.

If necessary, correct or add the sections below:

- Setting processor mode register
  Set a processor mode matched to your system to the processor mode register. (58-60th line in crt0mr.a30)

- Adding user-required initialization program
  When there is an initialization program that is required for your application, add it to the 140th line in the C language startup program (crt0mr.a30).

- Enable the 138th – 139th line in the C language startup program (crt0mr.a30) if standard I/O function is used.

7.3.1 C Language Startup Program (crt0mr.a30)

Figure 7.11 shows the C language startup program(crt0mr.a30).

```
; ****************************************************************
;
;     MR100 start up program for C language
;     COPYRIGHT(C) 2003,2006,2007 RENESAS TECHNOLOGY CORPORATION
;     AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
;     MR100
;
; ****************************************************************
;     "$Id: crt0mr.a30 512 2007-07-09 10:11:36Z inui $"
;*A1* 2005-02-28 for ES
;*G0* 2006-06-15 for MR100/4
;
      .LIST   OFF
      .INCLUDE        c_sec.inc
      .INCLUDE        mr100.inc
      .INCLUDE        sys_rom.inc
      .INCLUDE        sys_ram.inc
      .LIST   ON

      .GLB    __SYS_INITIAL
      .GLB    __END_INIT
      .GLB    __init_sys,__init_tsk

regoffset    .EQU    0

;-----------------------------------------------------------------
; SBDATA area definition
;-----------------------------------------------------------------
      .GLB    __SB__
      .SB     __SB__

;=================================================================
; Initialize Macro declaration
;-----------------------------------------------------------------
BZERO        .macro          TOP_,SECT_
      XOR.B           R0L,R0L
      mov.l           #TOP_,A1
      mov.l           #sizeof SECT_,R7R5
      sstr.b
      .endm
BCOPY        .macro          FROM_,TO_,SECT_
      mov.l           #FROM_,A0
      mov.l           #TO_,A1
      mov.l           #sizeof SECT_,R7R5
      smovf.b
      .endm
;=================================================================
; Interrupt section start
;-----------------------------------------------------------------
      .SECTION        MR_KERNEL,CODE,ALIGN

;-----------------------------------------------------------------
; after reset,this program will start
;-----------------------------------------------------------------
__SYS_INITIAL:
      LDC     #__Sys_Sp,ISP   ; set initial ISP

;     MOV.B   #2,0AH
;     MOV.B   #00,PMOD                ; Set Processor Mode Register
;     MOV.B   #0,0AH                  ;
      LDC     #00000010H,FLG
      LDC     #__SB__,SB
      LDC     #00000000H,FLG
      LDC     #__Sys_Sp,FB
      LDC     #__SB__,SB

;=================================================================
; MR_RAM zero clear
;---------------------------------------------------------
      BZERO   MR_RAM_top,MR_RAM

;=================================================================
; NEAR area initialize.
; FAR area initialize.
```

```
;--------------------------------------------------------
; bss zero clear
;--------------------------------------------------------
    ;---------------------------------------------------;
    ; zero clear BSS                                   ;
    ;---------------------------------------------------;
    BZERO           bss_SB8_top,  bss_SB8
;   BZERO           bss_SB16_top, bss_SB16
    BZERO           bss_NEAR_top, bss_NEAR
    BZERO           bss_FAR_top,  bss_FAR
    BZERO           bss_EXT_top,  bss_EXT
    BZERO           bss_MON1_top, bss_MON1
    BZERO           bss_MON2_top, bss_MON2
    BZERO           bss_MON3_top, bss_MON3
    BZERO           bss_MON4_top, bss_MON4

;--------------------------------------------------------
; initialize data section
;--------------------------------------------------------
    ;---------------------------------------------------;
    ; initialize DATA                                   ;
    ;---------------------------------------------------;
    BCOPY           data_SB8_INIT_top,  data_SB8_top,  data_SB8
;   BCOPY           data_SB16_INIT_top, data_SB16_top, data_SB16
    BCOPY           data_NEAR_INIT_top, data_NEAR_top, data_NEAR
    BCOPY           data_FAR_INIT_top,  data_FAR_top,  data_FAR
    BCOPY           data_EXT_INIT_top,  data_EXT_top,  data_EXT
    BCOPY           data_MON1_INIT_top, data_MON1_top, data_MON1
    BCOPY           data_MON2_INIT_top, data_MON2_top, data_MON2
    BCOPY           data_MON3_INIT_top, data_MON3_top, data_MON3
    BCOPY           data_MON4_INIT_top, data_MON4_top, data_MON4


;-----------------------------------------------------------------
; Set System IPL and Set Interrupt Vector
;-----------------------------------------------------------------
    __INI_IPL                                       ;*G0*
    LDC     #__INT_VECTOR,INTB

; +-----------------------------------------------------+
; |    System timer interrupt setting                 |
; +-----------------------------------------------------+
   .IF      USE_TIMER
    MOV.B   #stmr_mod_val,stmr_mod_reg+regoffset    ; set timer mode
    MOV.W   #stmr_cnt,stmr_ctr_reg+regoffset        ; set interval count
    MOV.B   #stmr_int_IPL,stmr_int_reg              ; set timer IPL
    OR.B    #stmr_bit+1,stmr_start+regoffset        ; system timer start
   .ENDIF

; +-----------------------------------------------------+
; |    System timer initialize                        |
; +-----------------------------------------------------+
   .IF      USE_SYSTEM_TIME
    MOV.W   #__D_Sys_TIME_L,__Sys_time+4
    MOV.W   #__D_Sys_TIME_M,__Sys_time+2
    MOV.W   #__D_Sys_TIME_H,__Sys_time
   .ENDIF
    MOV.L   #0,__HEAP_TMR

; +-----------------------------------------------------+
; |    User Initial Routine ( if there are )            |
; +-----------------------------------------------------+
; Initialize standard I/O
;    .GLB    __init
;    JSR.A   __init

; +-----------------------------------------------------+
; |    Initalization of System Data Area                |
; +-----------------------------------------------------+
    .GLB    __init_heap
    JSR.W   __init_sys
    JSR.W   __init_tsk
    JSR.W   __init_heap
   .IF      __NUM_FLG
    .GLB    __init_flg
    JSR.W   __init_flg
   .ENDIF

   .IF      __NUM_SEM
    .GLB    __init_sem
```

```
     JSR.W    __init_sem
    .ENDIF

    .IF       __NUM_DTQ
     .GLB     __init_dtq
     JSR.W    __init_dtq
    .ENDIF

    .IF       __NUM_VDTQ                     ;*A1*
     .GLB     __init_vdtq
     JSR.W    __init_vdtq
    .ENDIF

    .IF       __NUM_MBX
     .GLB     __init_mbx
     JSR.W    __init_mbx
    .ENDIF

    .IF       ALARM_HANDLER
     .GLB     __init_alh
     JSR.W    __init_alh
    .ENDIF

    .IF       CYCLIC_HANDLER
     .GLB     __init_cyh
     JSR.W    __init_cyh
    .ENDIF

    .IF       __NUM_MPF                      ;*A1*
     ; Fixed Memory Pool
     .GLB     __init_mpf
     JSR.W    __init_mpf
    .ENDIF

    .IF       __NUM_MPL                      ;*A1*
     ; Variable Memory Pool
     .GLB     __init_mpl
     JSR.W    __init_mpl
    .ENDIF


     ; For PD100
     __LAST_INITIAL

__END_INIT:

; +-------------------------------------------------------+
; |       Start initial active task                 |
; +-------------------------------------------------------+
     __START_TASK

     .GLB     __rdyq_search
     JMP.W    __rdyq_search

; +-----------------------------------------------+
; |    Define Dummy                         |
; +-----------------------------------------------+
     .GLB     __SYS_DMY_INH
__SYS_DMY_INH:
     REIT

.IF  CUSTOM_SYS_END
; +-----------------------------------------------+
; | Syscall exit rouitne to customize
; +-----------------------------------------------+
     .GLB     __sys_end
__sys_end:
     ; Customize here.
     REIT
.ENDIF

; +-----------------------------------------------+
; |          exit() function                |
; +-----------------------------------------------+
     .GLB     _exit,$exit
_exit:
$exit:
     JMP      _exit

.IF USE_TIMER
```

```
; +---------------------------------------------+
; |       System clock interrupt handler        |
; +---------------------------------------------+
    .GLB            __SYS_STMR_INH
    .ALIGN
__SYS_STMR_INH:
    ; process issue system call
    ; For PD100
    __ISSUE_SYSCALL

    ; System timer interrupt handler
    _STMR_hdr

    ret_int
.ENDIF

    .END

; ****************************************************************
;    COPYRIGHT(C) 2003,2007 RENESAS TECHNOLOGY CORPORATION
;    AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
;*************************************************************
```

Figure 7.11 C Language Startup Program (crt0mr.a30)

The following explains the content of the C language startup program (crt0mr.a30).

4. **Incorporate a section definition file [14 in Figure 7.11]**
5. **Incorporate an include file for MR100 [15 in Figure 7.11]**
6. **Incorporate a system ROM area definition file [16 in Figure 7.11]**
7. **Incorporate a system RAM area definition file [17 in Figure 7.11]**
8. **This is the initialization program __SYS_INITIAL that is activated immediately after a reset. [55 - 207 in Figure 7.11]**
    - Setting the System Stack pointer [56 in Figure 7.11]
    - Setting the SB,FB register [61 - 65 in Figure 7.11]
    - Initial set the C language. [72 - 105 in Figure 7.11]
    - Setting kernel interrupt mask level [111 in Figure 7.11]
    - Setting the address of interrupt vector table [112 in Figure 7.11]
    - Set MR100's system clock interrupt [114-122 in Figure 7.11]
    - Initial set MR100's system timer [124-132 in Figure 7.11]
9. **Initial set parameters inherent in the application [140 in Figure 7.11]**
10. **Initialize the RAM data used by MR100 [141- 197 in Figure 7.11]**
11. **Activate the initial startup task. [201-207 in Figure 7.11]**
12. **This is a system clock interrupt handler [235-248 in Figure 7.11]**

7.4 Memory Allocation

This section describes how memory is allocated for the application program data.

Use the section file provided by MR100 to set memory allocation.

MR100 comes with the following two types of section files:

- asm_sec.inc
  This file is used when you developed your applications with the assembly language.

- c_sec.inc
  This file is used when you developed your applications with the C language.

  c_sec.inc is derived from "asm_sec.inc" by adding sections generated by C compiler NC100.

Modify the section allocation and start address settings in this file to suit your system.

The following shows how to modify the section file.

e.g.

```
 If you want to change the rom_FAR section start address from FFE00000H to FFF00000H
;---------------------------------------------------------------------------;
; FAR ROM SECTIONS                                                   ;
;---------------------------------------------------------------------------;
    .section        rom_FAR, romdata
    .org            0FFE00000H
rom_FAR_top:

                     ↓

    .section        rom_FAR, romdata
    .org            0FFF00000H
rom_FAR_top:
```

7.4.1 Section used by the MR100

The sample section file for the C language is "asm_sec.inc". The sample section file for the assembly language is "asm_sec.inc". Edit these files if section reallocation is required.

The following explains each section that is used by the MR100.

- MR_RAM section
  This section is where the RAM data, MR100's system management data, is stored that is referenced in absolute addressing.

- stack section
  This section is provided for each task's user stack and system stack.

- MR_HEAP section
  This section stores the variable-size memorypool.

- MR_KERNEL section
  This section is where the MR100 kernel program is stored.

- MR_CIF section
  This section stores the MR100 C language interface library.

- MR_ROM section
  This section stores data such as task start addresses that area referenced by the MR100 kernel.

- program section
  This section stores user programs.

  This section is not used by the MR100 kernel at all. Therefore, you can use this section as desired.

- INTERRUPT_VECTOR section

- FIX_INTERRUPT_VECTOR section
  This section stores interrupt vectors. The start address of this section varies with the type of microcomputer used.

  .

8. Using Configurator

8.1 Configuration File Creation Procedure

When applications program coding and startup program modification are completed, it is then necessary to register the applications program in the MR100 system.

This registration is accomplished by the configuration file.

8.1.1 Configuration File Data Entry Format

This chapter describes how the definition data are entered in the configuration file.

Comment Statement

A statement from '//' to the end of a line is assumed to be a comment and not operated on.

End of statement

Statements are terminated by ';'.

Numerical Value

Numerical values can be entered in the following format.

- Hexadecimal Number
  Add "0x" or "0X" to the beginning of a numerical value, or "h" or "H" to the end. If the value begins with an alphabetical letter between A and F with "h" or "H" attached to the end, be sure to add "0" to the beginning. Note that the system does not distinguish between the upper- and lower-case alphabetical characters (A-F) used as numerical values.[52]

- Decimal Number
  Use an integer only as in '23'. However, it must not begin with '0'.

- Octal Numbers
  Add '0' to the beginning of a numerical value of 'O' or 'o' to end.

- Binary Numbers
  Add 'B' or 'b' to the end of a numerical value. It must not begin with '0'.

Table 8.1 Numerical Value Entry Examples

Hexadecimal	0xf12 0Xf12 0a12h 0a12H 12h 12H
Decimal	32
Octal	017 17o 17O
Binary	101110b 101010B

[52] The system distinguishes between the upper- and lower-case letters except for the numbers A-F and a-f.

It is also possible to enter operators in numerical values. Table 8.2 Operators lists the operators available.

Table 8.2 Operators

Operator	Priority	Direction of computation
()	High	From left to right
- (Unary_minus)		From right to left
∗ / %		From left to right
+ - (Binary_minus)	Low	From loft to right

13.

Numerical value examples are presented below.

- 123
- 123 + 0x23
- (23/4 + 3) ∗ 2
- 100B + 0aH

Symbol

The symbols are indicated by a character string that consists of numerals, upper- and lower-case alphabetical letters, _(underscore), and ?, and begins with a non-numeric character.

Example symbols are presented below.

- _TASK1
- IDLE3

Function Name

The function names are indicated by a character string that consists of numerals, upper and lower-case alphabetical letters,'$'(dollar) and '_'(underscore), begins with a non-numeric character, and ends with '()'.

The following shows an example of a function name written in the C language.

- main()
- func()

When written in the assembly language, the start label of a module is assumed to be a function name.

Frequency

The frequency is indicated by a character string that consist of numerals and . (period), and ends with MHz. The numerical values are significant up to six decimal places. Also note that the frequency can be entered using decimal numbers only.

Frequency entry examples are presented below.

- 16MHz
- 8.1234MHz

It is also well to remember that the frequency must not begin with . (period).

Time

The time is indicated by a character string that consists of numerals and . (period), and ends with ms. The time values are effective up to three decimal places when the character string is terminated with ms. Also note that the time can be entered using decimal numbers only.

- 10ms
- 10.5ms

It is also well to remember that the time must not begin with . (period).

8.1.2 Configuration File Definition Items

The following definitions [53] are to be formulated in the configuration file

- System definition
- System clock definition
- Respective maximum number of items
- Task definition
- Eventflag definition
- Semaphore definition
- Mailbox definition
- Data queue definition
- Short data queue definition
- Fixed-size Memory Pool definition
- Variable-size Memory Pool definition
- Cyclic handler definition
- Alarm handler definition
- Interrupt vector definition

[(System Definition Procedure)]

<< Format >>

```
// System Definition
system{
    stack_size     = System stack size ;
    priority       = Maximum value of priority ;
    system_IPL     = Kernel mask level;
    tic_deno       = Time tick denominator ;
    tic_nume       = Time tick numerator ;
    message_pri    = Maximum message priority value ;
};
```

[53] All items except task definition can omitted. If omitted, definitions in the default configuration file are referenced.

<< **Content** >>

1. **System stack size**

   [(Definition format)] Numeric value

   [(Definition range)] 6 or more

   [(Default value)] 400H

   Define the total stack size used in service call and interrupt processing.

2. **Maximum value of priority (value of lowest priority)**

   [(Definition format)] Numeric value

   [(Definition range)] 1 to 255

   [(Default value)] 32

   Define the maximum value of priority used in MR100's application programs. This must be the value of the highest priority used.

3. **Kernel mask level**

   [(Definition format)] Numeric value

   [(Definition range)] 1 to 7

   [(Default value)] 7

   Set the IPL value in service calls, that is, the OS interrupt disable level.

4. **Time tick denominator**

   [(Definition format)] Numeric value

   [(Definition range)] Fixed to 1

   [(Default value)] 1

   Set the denominator of the time tick.

5. **Time tick numerator**

   [(Definition format)] Numeric value

   [(Definition range)] 1 to 65,535

   [(Default value)] 1

   Set the numerator of the time tick. The system clock interrupt interval is determined by the time tick denominator and numerator that are set here. The interval is the time tick numerator divided by time tick denominator [ms]. That is, the time tick numerator [ms].

6. **Maximum message priority value**

   [(Definition format)] Numeric value

   [(Definition range)] 1 to 255

   [(Default value)] None

   Define the maximum value of message priority.

[(System Clock Definition Procedure)]

<< Format >>

```
// System Clock Definition
clock{
     timer_clock     = MPU clock ;
     timer           = Timers used for system clock ;
     IPL             = System clock interrupt priority level ;
};
```

<< Content >>

1. **MPU clock**

   [(Definition format)] Frequency(in MHz)

   [(Definition range)] None

   [(Default value)] 15MHz

   Define the MPU operating clock frequency of the microcomputer in MHz units.

2. **Timers used for system clock**

   [(Definition format)] Symbol

   [(Definition range)] A0, A1, A2, A3, A4, A5,A6,A7,B0, B1, B2, B3, B4, B5, OTHER, NOTIMER

   [(Default value)] NOTIMER

   The frequency of the circumference functional clock supplied to a system timer is defined per MHz. With this product, f1 or f8 is chosen as count sauce, and a value is set as a timer Ai register and a timer Bi register. Therefore, overflow may occur depending on the value of timer_clock, and the value of tick_nume of a system definition. In this case, OTHER must be set as the timer used for a system clock, and a system timer must be initialized by the user side.

   If you do not use a system clock, define "NOTIMER."

3. **System clock interrupt priority level**

   [(Definition format)] Numeric value

   [(Definition range)] 1 to Kernel mask level in system definition

   [(Default value)] 4

   Define the priority level of the system clock timer interrupt. The value set here must be smaller than the kernel interrupt mask level.

   Interrupts whose priority levels are below the interrupt level defined here are not accepted during system clock interrupt handler processing.

[(Definition respective maximum numbers of items)]

Here, define respective maximum numbers of items to be used in two or more applications.

<< Format >>

```
// Max Definition
maxdefine{
    max_task   = the maximum number of tasks defined ;
    max_flag   = the maximum number of eventflags defined ;
    max_dtq    = the maximum number of data queues defined ;
    max_mbx    = the maximum number of mailboxes defined ;
    max_sem    = the maximum number of semaphores defined ;
    max_mpf    = the maximum number of fixed-size
                 memory pools defined ;
    max_mpl    = the maximum number of variable-size
                 memory pools defined ;
    max_cyh    = the maximum number of cyclic handlers
                 defined ;
    max_alh    = the maximum number of alarm handlers
                 defined ;
    max_vdtq   = the maximum number of short data queues defined ;
};
```

<< Contents >>

1. **The maximum number of tasks defined**

   [(Definition format)] Numeric value

   [(Definition range)] 1 to 255

   [(Default value)] None

   Define the maximum number of tasks defined.

2. **The maximum number of eventflags defined**

   [(Definition format)] Numeric value

   [(Definition range)] 1 to 255

   [(Default value)] None

3. **The maximum number of data queues defined.**

   [(Definition format)] Numeric value

   [(Definition range)] 1 to 255

   [(Default value)] None

   Define the maximum number of data queues defined.

4. **The maximum number of mailboxes defined**

   [(Definition format)] Numeric value

   [(Definition range)] 1 to 255

   [(Default value)] None

   Define the maximum number of mailboxes defined.

5. **The maximum number of semaphores defined**

[(Definition format)] Numeric value

[(Definition range)] 1 to 255

[(Default value)] None

Define the maximum number of semaphores defined.

6. **The maximum number of fixed-size memory pools defined**

[(Definition format)] Numeric value

[(Definition range)] 1 to 255

[(Default value)] None

7. **The maximum number of variable length memory blocks defined.**

[(Definition format)] Numeric value

[(Definition range)] 1 to 255

[(Default value)] None

Define the maximum number of variable length memory blocks defined.

8. **The maximum number of cyclic activation handlers defined**

[(Definition format)] Numeric value

[(Definition range)] 1 to 255

[(Default value)] None

The maximum number of cyclic handler defined

9. **The maximum number of alarm handler defined**

[(Definition format)] Numeric value

[(Definition range)] 1 to 255

[(Default value)] None

Define the maximum number of alarm handlers defined.

10. **The maximum number of short data queues defined.**

[(Definition format)] Numeric value

[(Definition range)] 1 to 255

[(Default value)] None

Define the maximum number of short data queues defined.

[(Task definition)]

<< Format >>

```
 // Tasks Definition
task[ ID No. ]{
    name              = ID name ;
    entry_address     = Start task of address ;
    stack_size        = User stack size of task ;
    priority          = Initial priority of task ;
    context           = Registers used ;
    stack_section     = Section name in which the stack is located ;
    initial_start     = TA ACT attribute (initial startup state) ;
    exinf             = Extended information ;
 };
            :
            :
```

The ID number must be in the range of 1 to 255. The ID number can be omitted.

If omitted, numbers are automatically assigned sequentially beginning with the smallest.

<< Content >>

Define the following for each task ID number.

1. **Task ID name**

   [(Definition format)] Symbol

   [(Definition range)] None

   [(Default value)] None

   Define the ID name of a task. Note that the function name defined here is output to the kernel_id.h file, as shown below.

   #define <u>Task ID Name</u> <u>task ID</u>

2. **Start address of task**

   [(Definition format)] Symbol or function name

   [(Definition range)] None

   [(Default value)] None

   Define the entry address of a task. When written in the C language, add () at the end or _at the beginning of the function name you have defined.

   The function name defined here causes the following declaration statement to be output in the kernel_id.h file:

```
#pragma TASK /V4 Function Name
```

3. User stack size of task

[(Definition format)] Numeric value

[(Definition range)] 12 or more

[(Default value)] 256

Define the user stack size for each task. The user stack means a stack area used by each individual task. MR100 requires that a user stack area be allocated for each task, which amount to at least 12 bytes.

4. Initial priority of task

[(Definition format)] Numeric value

[(Definition range)] 1 to (maximum value of priority in system definition)

[(Default value)] 1

Define the priority of a task at startup time.

As for MR100's priority, the lower the value, the higher the priority.

5. Regisers Used

[(Definition format)] Symbol[,Symbol,....]

[(Definition range)] Selected from R2R0,R3R1,R6R4,R7R5,A0,A1,A2,A3,SB,FB

[(Default value)] All registers

Define the registers used in a task. MR100 handles the register defined here as a context. Specify the R2R0 register because task startup code is set in it when the task starts.

However, the registers used can only be selected when the task is written in the assembly language. Select all registers when the task is written in the C language. When selecting a register here, be sure to select all registers that store service call parameters used in each task.

MR100 kernel does not change the registers of bank.

If this definition is omitted, it is assumed that all registers are selected.

6. Section name in which the stack is located

[(Definition format)] Symbol

[(Definition range)] None

[(Default value)] stack

Define the section name in which the stack is located. The section defined here must always have an area allocated for it in the section file (asm_sec.inc or c_sec.inc).

If no section names are defined, the stack is located in the stack section.

7. TA_ACT attribute (initial startup state)

[(Definition format)] Symbol

[(Definition range)] ON or OFF

[(Default value)] OFF

Define the initial startup state of a task.

If this attribute is specified ON, the task goes to a READY state at the initial system startup time.

The task startup code of the initial startup task is the extended information.

8. **Extended information**

[(Definition format)] Numeric value

[(Definition range)] 0 to 0xFFFFFFFF

[(Default value)] 0

Define the extended information of a task. This information is passed to the task as argument when it is restarted by a queued startup request, for example.

[(Eventflag definition)]

This definition is necessary to use Eventflag function.

<< Format >>

```
// Eventflag Definition
flag[ ID No. ]{
    name            = Name ;
    wait_queue      = Selecting an event flag waiting queue ;
    initial_pattern = Initial value of the event flag ;
    wait_multi      = Multi-wait attribute ;
    clear_attribute = Clear attribute ;
};
            :
            :
```

The ID number must be in the range of 1 to 255. The ID number can be omitted.

If omitted, numbers are automatically assigned sequentially beginning with the smallest.

<< Content >>

Define the following for each eventflag ID number.

1. **ID Name**

[(Definition format)] Symbol

[(Definition range)] None

[(Default value)] None

Define the name with which an eventflag is specified in a program.

2. **Selecting an event flag waiting queue**

[(Definition format)] Symbol

[(Definition range)] TA_TFIFO or TA_TPRI

[(Default value)] TA_TFIFO

Select a method in which tasks wait for the event flag. If TA_TFIFO is selected, tasks are enqueued in order of FIFO. If TA_TPRI is selected, tasks are enqueued in order of priority beginning with the one that has the highest priority.

3. **Initial value of the event flag**

[(Definition format)] Numeric value

[(Definition range)] 0 to 0xFFFFFFFF

[(Default value)] 0

Specify the initial bit pattern of the event flag.

4. Multi-wait attribute

[(Definition format)] Symbol

[(Definition range)] TA_WMUL or TA_WSGL

[(Default value)] TA_WSGL

Specify whether multiple tasks can be enqueued in the eventflag waiting queue. If TA_WMUL is selected, the TA_WMUL attribute is added, permitting multiple tasks to be enqueued. If TA_WSGL is selected, the TA_WSGL attribute is added, prohibiting multiple tasks from being enqueued.

5. Clear attribute

[(Definition format)] Symbol

[(Definition range)] YES or NO

[(Default value)] NO

Specify whether the TA_CLR attribute should be added as an eventflag attribute. If YES is selected, the TA_CLR attribute is added. If NO is selected, the TA_CLR attribute is not added.

[(Semaphore definition)]

This definition is necessary to use Semaphore function.

<< Format >>

```
// Semaphore Definition
semaphore[ ID No. ]{
    name            = ID name ;
    wait_queue      = Selecting a semaphore waiting queue ;
    initial_count   = Initial value of semaphore counter ;
    max_count       = Maximum value of the semaphore counter ;
};
            :
            :
```

The ID number must be in the range of 1 to 255. The ID number can be omitted.

If omitted, numbers are automatically assigned sequentially beginning with the smallest.

<< Content >>

Define the following for each semaphore ID number.

1. ID Name

[(Definition format)] Symbol

[(Definition range)] None

[(Default value)] None

Define the name with which a semaphore is specified in a program.

2. **Selecting a semaphore waiting queue**

   [(Definition format)] Symbol

   [(Definition range)] TA_TFIFO or TA_TPRI

   [(Default value)] TA_TFIFO

   Select a method in which tasks wait for the semaphore. If TA_TFIFO is selected, tasks are enqueued in order of FIFO. If TA_TPRI is selected, tasks are enqueued in order of priority beginning with the one that has the highest priority.

3. **Initial value of semaphore counter**

   [(Definition format)] Numeric value

   [(Definition range)] 0 to 65535

   [(Default value)] 1

   Define the initial value of the semaphore counter. This value must be less than the maximum value of the semaphore counter.

4. **Maximum value of the semaphore counter**

   [(Definition format)] Numeric value

   [(Definition range)] 1 to 65535

   [(Default value)] 1

   Define the maximum value of the semaphore counter.

[(Data queue definition)]

This definition must always be set when the data queue function is to be used.

<< Format >>

```
// Dataqueue Definition
dataqueue[ ID No. ]{
    name            = ID name;
    buffer_size     = Number of data queues;
    wait_queue      = Select data queue waiting queue;
};
            :
            :
```

The ID number must be in the range 1 to 255. The ID number can be omitted. If omitted, ID numbers are automatically assigned in order of numbers beginning with the smallest.

<< Content >>

For each data queue ID number, define the items described below.

1. **ID name**

   [(Definition format)] Symbol

   [(Definition range)] None

   [(Default value)] None

   Define the name by which the data queue is specified in a program.

2. **Number of data**

   [(Definition format)] Numeric Value

   [(Definition range)] 0 to 0x1FFF

   [(Default value)] 0

   Specify the number of data that can be transmitted. What should be specified here is the number of data, and not a data size.

3. **Selecting a data queue waiting queue**

   [(Definition format)] Symbol

   [(Definition range)] TA_TFIFO or TA_TRPI

   [(Default value)] TA_TFIFO

   Select a method in which tasks wait for data queue transmission. If TA_TFIFO is selected, tasks are enqueued in order of FIFO. If TA_TPRI is selected, tasks are enqueued in order of priority beginning with the one that has the highest priority.

[(Short data queue definition)]

This definition must always be set when the short data queue function is to be used.

<< Format >>

```
// Vdataqueue Definition
vdataqueue [ ID No. ]{
    name            = ID name ;
    buffer_size     = Number of data queues ;
    wait_queue      = Select data queue waiting queue ;
};
            :
            :
```

The ID number must be in the range 1 to 255. The ID number can be omitted. If omitted, ID numbers are automatically assigned in order of numbers beginning with the smallest.

<< Content >>

For each short data queue ID number, define the items described below.

1. **ID name**

   [(Definition format)] Symbol

   [(Definition range)] None

   [(Default value)] None

   Define the name by which the short data queue is specified in a program.

2. **Number of data**

   [(Definition format)] Numeric Value

   [(Definition range)] 0 to 0x3FFF

   [(Default value)] 0

   Specify the number of data that can be transmitted. What should be specified here is the number of data, and not a data size.

3. Selecting a data queue waiting queue

[(Definition format)] Symbol

[(Definition range)] TA_TFIFO or TA_TRPI

[(Default value)] TA_TFIFO

Select a method in which tasks wait for short data queue transmission. If TA_TFIFO is selected, tasks are enqueued in order of FIFO. If TA_TPRI is selected, tasks are enqueued in order of priority beginning with the one that has the highest priority.

[(Mailbox definition)]

This definition must always be set when the mailbox function is to be used.

<< Format >>

```
// Mailbox Definition
mailbox[ ID No. ]{
    name            = ID name ;
    wait_queue      = Select mailbox waiting queue ;
    message_queue   = Select message queue ;
    max_pri         = Maximum message priority ;
};
            :
            :
```

The ID number must be in the range 1 to 255. The ID number can be omitted. If omitted, ID numbers are automatically assigned in order of numbers beginning with the smallest.

<< Content >>

For each mailbox ID number, define the items described below.

1. ID name

[(Definition format)] Symbol

[(Definition range)] None

[(Default value)] None

Define the name by which the mailbox is specified in a program.

2. Select mailbox waiting queue

[(Definition format)] Symbol

[(Definition range)] TA_TFIFO or TA_TPRI

[(Default value)] TA_TFIFO

Select a method in which tasks wait for the mailbox. If TA_TFIFO is selected, tasks are enqueued in order of FIFO. If TA_TPRI is selected, tasks are enqueued in order of priority beginning with the one that has the highest priority.

3. Select message queue

[(Definition format)] Symbol

[(Definition range)] TA_MFIFO or TA_MRPI

[(Default value)] TA_MFIFO

Select a method by which a message queue of the mailbox is selected. If TA_MFIFO is selected, messages are enqueued in order of FIFO. If TA_MPRI is selected, messages are enqueued in order of priority beginning with the one that has the highest priority.

4. **Maximum message priority**

[(Definition format)] Numeric Value

[(Definition range)] 1 to "maximum value of message priority" that was specified in "definition of maximum number of items"

[(Default value)] 1

Specify the maximum priority of message in the mailbox.

[(Fixed-size memory pool definition)]

This definition must always be set when the fixed-size memory pool function is to be used.

<< **Format** >>

```
// Fixed Memory pool Definition
memorypool[ ID No. ]{
    name            = ID name ;
    section         = Section Name ;
    num_block       = Number of blocks in memory pool ;
    siz_block       = Block size of Memory pool ;
    wait_queue      = Select memory pool waiting queue ;
};
```

The ID number must be in the range 1 to 255. The ID number can be omitted. If omitted, ID numbers are automatically assigned in order of numbers beginning with the smallest.

<< **Content** >>

For each memory pool ID number, define the items described below.

1. **ID name**

[(Definition format)] Symbol

[(Definition range)] None

[(Default value)] None

Define the name by which the memory pool is specified in a program.

2. **Section name**

[(Definition format)] Symbol

[(Definition range)] None

[(Default value)] MR_HEAP

Define the name of the section in which the memory pool is located. The section defined here must always have an area allocated for it in the section file (asm_sec.inc or c_sec.inc).

If no section names are defined, the memory pool is located in the MR_HEAP section.

3. **Number of block**

[(Definition format)] Numeric value

[(Definition range)] 1 to 65,535

[(Default value)] 1

Define the total number of blocks that comprise the memory pool.

4. **Size (in bytes)**

[(Definition format)] Numeric value

[(Definition range)] 4 to 65,535

[(Default value)] 256

Define the size of the memory pool per block. The RAM size to be used as a memory pool is determined by this definition: (number of blocks) x (size) in bytes.

5. **Selecting a memory pool waiting queue**

[(Definition format)] Symbol

[(Definition range)] TA_TFIFO or TA_TPRI

[(Default value)] TA_TFIFO

Select a method in which tasks wait for acquisition of the fixed-size memory pool. If TA_TFIFO is selected, tasks are enqueued in order of FIFO. If TA_TPRI is selected, tasks are enqueued in order of priority beginning with the one that has the highest priority.

[(Variable-size memory pool definition)]

This definition is necessary to use Variable-size memory pool function.

<< **Format** >>

```
// Message buffer Definition
message_buffer[ ID No. ]{
    name            = ID Name ;
    mbf_section     = Section name ;
    mbf_size        = Message buffer size ;
    max_msgsz       = Maximum message size ;
    wait_queue      = Message buffer transmit wait queue selection ;
};
```

The ID number must be in the range from 1 to 255. The ID number can be omitted. If omitted, ID numbers are automatically assigned in order of magnitude beginning with the smallest.

<< **Content** >>

6. **ID name**

[(Definition format)] Symbol

[(Definition range)] None

[(Default value)] None

Define the name by which the memory pool is specified in a program.

7. **The maximum memory block size to be allocated**

[(Definition format)] Numeric value

[(Definition range)] 1 to 65520

[(Default value)] None

Specify, within an application program, the maximum memory block size to be allocated.

8. Section name

[(Definition format)] Symbol

[(Definition range)] None

[(Default value)] MR_HEAP

Define the name of the section in which the memory pool is located. The section defined here must always have an area allocated for it in the section file (asm_sec.inc or c_sec.inc).

If no section names are defined, the memory pool is located in the MR_HEAP section.

9. Memory pool size

[(Definition format)] Numeric value

[(Definition range)] 16 to 0xFFFFFFFC

[(Default value)] None

Specify a memory pool size.

Round off a block size you specify to the optimal block size among the four block sizes, and acquires memory having the rounded-off size from the memory pool.

The following equations define the block sizes:

$$a = (((max\_memsize+(X-1))/ (X \times 8))+1) \times 8$$
$$b = a \times 2$$
$$c = a \times 4$$
$$d = a \times 8$$

max_memsize: the value specified in the configuration file
X: data size for block control (8 byte per a block control)

Variable-size memory pool function needs 8 byte RAM area per a block control. Memory pool size needs a size more than a, b, c or d that can be stored max_memsize + 8.

10. Select block usage

[(Definition format)] Symbol

[(Definition range)] ON,OFF

[(Default value)] OFF

This is an option to increase memory efficiency for even small-sized memory pools by means of small blocks. Memory is managed in 12 fixed-length memory pools ranging in size from 24 bytes to 65,528 bytes. When this option is turned on, the value of max_memsize has no effect.

[(Cyclic handler definition)]

This definition is necessary to use Cyclic handler function.

<< Format >>

```
// Cyclic Handlar Definition
cyclic_hand[ ID No. ]{
    name                    = ID name ;
    interval_counter        = Activation cycle ;
    start                   = TA STA attribute ;
    phsatr                  = TA PHS attribute ;
    phs_counter             = Activation phase ;
    entry_address           = Start address ;
    exitf                   = Extended information ;
};
            :
            :
```

The ID number must be in the range of 1 to 255. The ID number can be omitted.

If omitted, numbers are automatically assigned sequentially beginning with the smallest.

<< Content >>

Define the following for each cyclic handler ID number.

1. **ID name**

   [(Definition format)] Symbol

   [(Definition range)] None

   [(Default value)] None

   Define the name by which the memory pool is specified in a program.

2. **Activation cycle**

   [(Definition format)] Numeric value

   [(Definition range)] 1 to 0x7FFFFFFF

   [(Default value)] None

   Define the activation cycle at which time the cyclic handler is activated periodically. The activation cycle here must be defined in the same unit of time as the system clock's unit time that is defined in system clock definition item. If you want the cyclic handler to be activated at 1-second intervals, for example, the activation cycle here must be set to 1000.

3. **TA_STA attribute**

   [(Definition format)] Symbol

   [(Definition range)] ON or OFF

   [(Default value)] OFF

   Specify the TA_STA attribute of the cyclic handler. If ON is selected, the TA_STA attribute is added; if OFF is selected, the TA_STA attribute is not added.

4. **TA_PHS attribute**

   [(Definition format)] Symbol

   [(Definition range)] ON or OFF

   [(Default value)] OFF

   Specify the TA_PHS attribute of the cyclic handler. If ON is selected, the TA_PHS attribute is added; if OFF is selected, the TA_PHS attribute is not added.

5. **Activation phase**

[(Definition format)] Numeric value

[(Definition range)] 0 to 0x7FFFFFFF

[(Default value)] None

Define the activation phase of the cyclic handler. The time representing this startup phase must be defined in ms units.

6. **Start Address**

[(Definition format)] Symbol or Function Name

[(Definition range)] None

[(Default value)] None

Define the start address of the cyclic handler.

Note that the function name defined here will have the declaration statement shown below output to the kernel_id.h file.

#pragma CYCHANDLER /V4 function name

7. **Extended information**

[(Definition format)] Numeric value

[(Definition range)] 0 to 0xFFFFFFFF

[(Default value)] 0

Define the extended information of the cyclic handler. This information is passed as argument to the cyclic handler when it starts.

[(Alarm handler definition)]

This definition is necessary to use Alarm handler function.

<< Format >>

```
// Alarm Handlar Definition
alarm_hand[ ID No. ]{
    name              = ID name ;
    entry_address     = Start address ;
    exitf             = Extended information ;
};
              :
              :
```

The ID number must be in the range of 1 to 255. The ID number can be omitted.

If omitted, numbers are automatically assigned sequentially beginning with the smallest.

<< Content >>

Define the following for each alarm handler ID number.

1. **ID name**

[(Definition format)] Symbol

[(Definition range)] None

[(Default value)] None

Define the name by which the alarm handler is specified in a program.

2. **Start address**

[(Definition format)] Symbol or Function Name

[(Definition range)] None

Define the start address of the alarm handler. The function name defined here causes the following declaration statement to be output in the kernel_id.h file.

3. **Extended information**

[(Definition format)] Numeric value

[(Definition range)] 0 to 0xFFFFFFFF

[(Default value)] 0

Define the extended information of the alarm handler. This information is passed as argument to the alarm handler when it starts.

[(Interrupt vector definition)]
This definition is necessary to use Interrupt function.

<< Format >>

```
// Interrupt Vector Definition
interrupt_vector[ Vector No. ]{
    os_int          = Kernel-managed (OS dependent) interrupt handler ;
    entry_address   = Start address ;
    pragma_switch   = Switch passed to PRAGMA extended function ;
};
              :
              :
```

The vector number can be written in the range of 0 to 255. However, whether or not the defined vector number is valid depends on the microcomputer used

Configurator can't create an Initialize routine (interrupt control register, interrupt causes etc.) for this defined interrupt. You need to create that.

<< Content >>

4. **Kernel (OS dependent) interrupt handler**

[(Definition format)] Symbol

[(Definition range)] YES or NO

Define whether the handler is a kernel(OS dependent) interrupt handler. If it is a kernel(OS dependent) interrupt handler, specify YES; if it is a non-kernel(OS independent) interrupt handler, specify No.

If this item is defined as YES, the declaration statement shown below is output to the kernel_id.h file.

```
#pragma INTHANDLER /V4 function name
```

If this item is defined as NO, the declaration statement shown below is output to the kernel_id.h file.

```
#pragma INTERRUPT /V4 function name
```

5. **Start address**

[(Definition format)] Symbol or function name

[(Definition range)] None

[(Default value)] __SYS_DMY_INH

Define the entry address of the interrupt handler. When written in the C language, add () at the end or at the beginning of the function name you have defined.

6. Switch passed to PRAGMA extended function

[(Definition format)] Symbol

[(Definition range)] E, F, B or R

[(Default value)] None

Specify the switch to be passed to #pragma INTHANDLER or #pragma INTERRUPT. If "E" is specified, a "/E" switch is selected, in which case multiple interrupts are enabled. If "F" is specified, a "/F" switch is selected, in which case a "FREIT" instruction is output at return from the interrupt handler. If "B" is specified, a "/B" switch is selected, in which case register bank 1 is selected. If "R" is specified, a "/R" switch is selected, in which case no codes are output that change the floating-number rounding mode of the FLG register to the "nearest value."

Multiple switches can be specified at the same time. However, if a kernel managed interrupt handler is concerned, only the "E" or "R" switch can be specified. For non-kernel managed interrupt handlers, the "E", "F" and "B" switches can be specified, providing that "E" and "B" are not specified at the same time.

[(Fixed interrupt vector definition)]

This definition needs to be set when interrupt handlers based on fixed vector table are used.

<< Format >>

```
// Fixed Interrupt Vector Definition
interrupt_fvector[ Vector No. ]{
    entry_address  = Start address ;
    pragma_switch  = Switch passed to PRAGMA extended function ;
};
            :
            :
```

The interrupt vector number can be set in the range from 0 to 11. The relationship between the vector numbers and the interrupts and vector addresses is shown below. All these interrupts are handled as non-kernel managed interrupt handlers.

Table 8.3 List of vector number and vector address

Vector number	Vector address	Interrupt
0	FFFFFFD0H	Kernel reserved area
1	FFFFFFD4H	Kernel reserved area
2	FFFFFFD8H	Kernel reserved area
3	FFFFFFDCH	Undefined instruction
4	FFFFFFE0H	Overflow
5	FFFFFFE4H	BRK instruction
6	FFFFFFE8H	Reserved area
7	FFFFFFECH	Reserved area
8	FFFFFFF0H	Watchdog timer, voltage down detection, oscillation stop detection
9	FFFFFFF4H	Reserved area
10	FFFFFFF8H	NMI
11	FFFFFFFCH	Reset

<< Content >>

1. **Start address**

   [(Definition format)] Symbol or function name

   [(Definition range)] None

   [(Default value)] __SYS_DMY_INH

   Define the entry address to the interrupt handler. When written in C language, add () at the end of the function name or __ at the beginning of it.

2. **Switch passed to PRAGMA extended function**

   [(Definition format)] Symbol

   [(Definition range)] B or R

   [(Default value)] None

   Specify the switch to be passed to #pragma INTERRUPT. If "B" is specified, a "/B" switch is selected, in which case register bank 1 is selected. If "R" is specified, a "/R" switch is selected, in which case no codes are output that change the floating-number rounding mode of the FLG register to the "nearest value."

   Both switches can be specified at the same time.

[Precautions]

1. Regarding the method for specifying a register bank

No kernel interrupt handlers that use the registers in register bank 1 can be written in C language. These handlers can only be written in assembly language. When writing in assembly language, write the entry and exit to and from the interrupt handler as shown below.

(Always be sure to clear the B flag before issuing ret_int service call.)

Example: interrupt;

```
        fset    B
        fclr    B
        ret_int
```

Internally in the MR100 kernel, register banks are not switched over.

2. Regarding the method for specifying a high-speed interrupt

To ensure the effective use of a high-speed interrupt, be sure that the registers in register bank 1 are used in the high-speed interrupt. Also be aware that the high-speed interrupts used cannot be a kernel interrupt handler..

8.1.3 Configuration File Example

The following is the configuration file example.

```
////////////////////////////////////////////////////////////////////////////////
//
//    kernel.cfg : building file for MR100 Ver.1.00
//
//    Generated by M3T-MR100 GUI Configurator at 2007/02/28 19:01:20
//
////////////////////////////////////////////////////////////////////////////////

// system definition
system{
     stack_size      = 256;
     system_IPL       = 4;
     message_pri      = 64;
     timeout  = NO;
     task_pause       = NO;
     tick_nume        = 10;
     tick_deno        = 1;
};

// max definition
maxdefine{
     max_task        = 3;
     max_flag        = 4;
     max_sem = 3;
     max_dtq = 3;
     max_mbx = 4;
     max_mpf = 3;
     max_mpl = 3;
     max_cyh = 4;
     max_alh = 2;
};

// system clock definition
clock{
     timer_clock     = 20.000000MHz;
     timer   = A0;
     IPL     = 3;
};

task[]{
     entry_address   = task1();
     name    = ID_task1;
     stack_size      = 256;
     priority        = 1;
     initial_start   = OFF;
     exinf   = 0x0;
};
task[]{
     entry_address   = task2();
     name    = ID_task2;
     stack_size      = 256;
     priority        = 5;
     initial_start   = ON;
     exinf   = 0xFFFF;
};
task[3]{
     entry_address   = task3();
     name    = ID_task3;
     stack_size      = 256;
     priority        = 7;
     initial_start   = OFF;
     exinf   = 0x0;
};

flag[]{
     name    = ID_flg1;
     initial_pattern = 0x00000000;
     wait_queue      = TA_TFIFO;
     clear_attribute = NO;
     wait_multi      = TA_WSGL;
};
flag[1]{
     name    = ID_flg2;
     initial_pattern = 0x00000001;
```

```
    wait_queue      = TA_TFIFO;
    clear_attribute = NO;
    wait_multi      = TA_WMUL;
};
flag[2]{
    name    = ID_flg3;
    initial_pattern = 0x0000ffff;
    wait_queue      = TA_TPRI;
    clear_attribute = YES;
    wait_multi      = TA_WMUL;
};
flag[]{
    name    = ID_flg4;
    initial_pattern = 0x00000008;
    wait_queue      = TA_TPRI;
    clear_attribute = YES;
    wait_multi      = TA_WSGL;
};

semaphore[]{
    name    = ID_sem1;
    wait_queue      = TA_TFIFO;
    initial_count   = 0;
    max_count       = 10;
};
semaphore[2]{
    name    = ID_sem2;
    wait_queue      = TA_TFIFO;
    initial_count   = 5;
    max_count       = 10;
};
semaphore[]{
    name    = ID_sem3;
    wait_queue      = TA_TPRI;
    initial_count   = 255;
    max_count       = 255;
};

dataqueue[]{
    name    = ID_dtq1;
    wait_queue      = TA_TFIFO;
    buffer_size     = 10;
};
dataqueue[2]{
    name    = ID_dtq2;
    wait_queue      = TA_TPRI;
    buffer_size     = 5;
};
dataqueue[3]{
    name    = ID_dtq3;
    wait_queue      = TA_TFIFO;
    buffer_size     = 256;
};

mailbox[]{
    name    = ID_mbx1;
    wait_queue      = TA_TFIFO;
    message_queue   = TA_MFIFO;
    max_pri = 4;
};
mailbox[]{
    name    = ID_mbx2;
    wait_queue      = TA_TPRI;
    message_queue   = TA_MPRI;
    max_pri = 64;
};
mailbox[]{
    name    = ID_mbx3;
    wait_queue      = TA_TFIFO;
    message_queue   = TA_MPRI;
    max_pri = 5;
};
mailbox[4]{
    name    = ID_mbx4;
    wait_queue      = TA_TPRI;
    message_queue   = TA_MFIFO;
    max_pri = 6;
};

memorypool[]{
```

```
    name    = ID_mpf1;
    wait_queue      = TA_TFIFO;
    section = MR_RAM;
    siz_block       = 16;
    num_block       = 5;
};
memorypool[2]{
    name    = ID_mpf2;
    wait_queue      = TA_TPRI;
    section = MR_RAM;
    siz_block       = 32;
    num_block       = 4;
};
memorypool[3]{
    name    = ID_mpf3;
    wait_queue      = TA_TFIFO;
    section = MPF3;
    siz_block       = 64;
    num_block       = 256;
};

variable_memorypool[]{
    name    = ID_mpl1;
    max_memsize     = 8;
    heap_size       = 16;
};
variable_memorypool[]{
    name    = ID_mpl2;
    max_memsize     = 64;
    heap_size       = 256;
};
variable_memorypool[3]{
    name    = ID_mpl3;
    max_memsize     = 256;
    heap_size       = 1024;
};

cyclic_hand[]{
    entry_address   = cyh1();
    name    = ID_cyh1;
    exinf   = 0x0;
    start   = ON;
    phsatr  = OFF;
    interval_counter        = 0x1;
    phs_counter     = 0x0;
};
cyclic_hand[]{
    entry_address   = cyh2();
    name    = ID_cyh2;
    exinf   = 0x1234;
    start   = OFF;
    phsatr  = ON;
    interval_counter        = 0x20;
    phs_counter     = 0x10;
};
cyclic_hand[]{
    entry_address   = cyh3;
    name    = ID_cyh3;
    exinf   = 0xFFFF;
    start   = ON;
    phsatr  = OFF;
    interval_counter        = 0x20;
    phs_counter     = 0x0;
};
cyclic_hand[4]{
    entry_address   = cyh4();
    name    = ID_cyh4;
    exinf   = 0x0;
    start   = ON;
    phsatr  = ON;
    interval_counter        = 0x100;
    phs_counter     = 0x80;
};

alarm_hand[]{
    entry_address   = alm1();
    name    = ID_alm1;
    exinf   = 0xFFFF;
};
alarm_hand[2]{
```

```
    entry_address   = alm2;
    name     = ID_alm2;
    exinf    = 0x12345678;
};


//
// End of Configuration
//
```

8.2 Configurator Execution Procedures

8.2.1 Configurator Overview

The configurator is a tool that converts the contents defined in the configuration file into the assembly language include file, etc.Figure 8.1 outlines the operation of the configurator.

When used on HEW, the configurator is automatically started, and an application program is built.

Executing the configurator requires the following input files:

- Configuration file (XXXX.cfg)
  This file contains description of the system's initial setup items. It is created in the current directory.

- Default configuration file (default.cfg)
  This file contains default values that are referenced when settings in the configuration file are omitted. This file is placed in the directory indicated by environment variable "LIB30" or the current directory. If this file exists in both directories, the file in the current directory is prioritized over the other.

- include template file(mr100.inc, sys_ram.inc)
  This file serves as the template file of include file "mr100.inc" and “sys_ram.inc”. It resides in the directory indicated by environment variable "LIB100."

- MR100 version file (version)
  This file contains description of MR100's version. It resides in the directory indicated by environment variable "LIB100." The configurator reads in this file and outputs MR100's version information to the startup message.

- Service call definition file(kernel_sysint.h)
  This file contains description of MR100 service call definition. It resides in the directory indicated by environment variable "LIB100." The configurator reads in this file and outputs to thecurrent directory.

When the configurator is executed, the files listed below are output.

Do not define user data in the files output by the configurator. Starting up the configurator after entering data definitions may result in the user defined data being lost.

- System data definition file (sys_rom.inc, sys_ram.inc)
  This file contains definition of system settings.

- Include file (mr100.inc)
  This is an include file for the assembly language.

- Service call definition file(kernel_sysint.h)
  This file contains description of MR100 service call definition

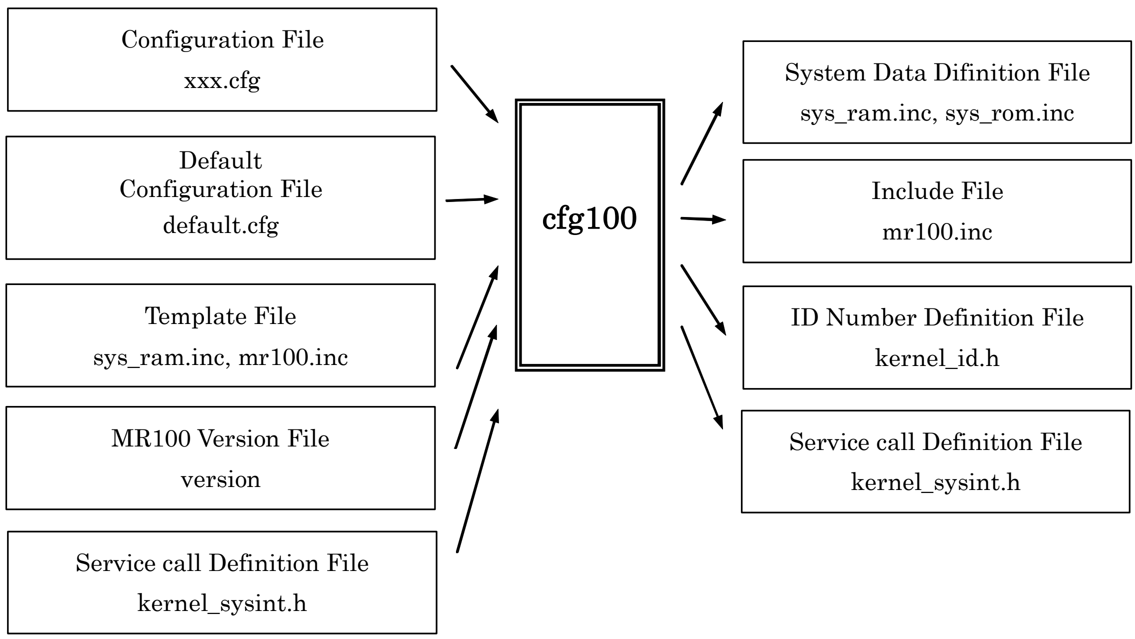

Figure 8.1 The operation of the Configurator

8.2.2 Setting Configurator Environment

Before executing the configurator, check to see if the environment variable "LIB100" is set correctly.

The configurator cannot be executed normally unless the following files are present in the directory indicated by the environment variable "LIB100":

- Default configuration file (default.cfg)

This file can be copied to the current directory for use. In this case, the file in the current directory is given priority.

- System RAM area definition database file (sys_ram.inc)
- mr100.inc template file (mr100.inc)
- Section definition file(c_sec.inc or asm_sec.inc)
- Startup file(crt0mr.a30 or start.a30)
- MR100 version file(version)
- Service call definition file(kernel_sysint.h)

8.2.3 Configurator Start Procedure

Start the configurator as indicated below.

```
C:\> cfg100  [-vV]  [-Eipl] [-Wipl]   Configuration file name
```

Normally, use the extension .cfg for the configuration file name. The file name can includes space character with “”.

Command Options

-v Option

Displays the command option descriptions and detailed information on the version.

-V Option

Displays the information on the files generated by the command.

-Eipl Option

Enable the check function of an IPL value. When System_IPL! = 7 in the con-figuration file, the error message " system_IPL should be 7" is displayed and the execution of cfg100 is stpped.

-Wipl Option

Enable the check function of a IPL value. When System_IPL! = 7 in the con-figuration file, the error message " system_IPL should be 7" is displayed..

8.2.4 Precautions on Executing Configurator

The following lists the precautions to be observed when executing the configurator:

- Do not modify the startup program name and the section definition file name. Otherwise, an error may be encountered when executing the configurator.

8.2.5 Configurator Error Indications and Remedies

If any of the following messages is displayed, the configurator is not normally functioning. Therefore, correct the configuration file as appropriate and the execute the configurator again.

Error messages

cfg100 Error : Syntax error near line xxx (xxxx.cfg)
There is an syntax error in the configuration file.

cfg100 Error : Not enough memory
Memory is insufficient.

cfg100 Error : Illegal option --> <x>
The configurator's command option is erroneous.

cfg100 Error : Illegal argument --> <xx>
The configurator's startup format is erroneous.

cfg100 Error : Can't write open <XXXX>
The XXXX file cannot be created. Check the directory attribute and the remaining disk capacity available.

cfg100 Error : Can't open <XXXX>
The XXXX file cannot be accessed. Check the attributes of the XXXX file and whether it actually exists.

cfg100 Error : Can't open version file
The MR100 version file "version" cannot be found in the directory indicated by the environment variable "LIB30".

cfg100 Error : Can't open default configuration file
The default configuration file cannot be accessed. "default.cfg" is needed in the current directory or directory "LIB100" specifying.

cfg100 Error : Can't open configuration file <xxxx.cfg>
The configuration file cannot be accessed. Check that the file name has been properly designated.

cfg100 Error : illegal XXXX --> <xx> near line xxx (xxxx.cfg)
The value or ID number in definition item XXXX is incorrect. Check the valid range of definition.

cfg100 Error : Unknown XXXX --> <xx> near line xx (xxxx.cfg)
The symbol definition in definition item XXXX is incorrect. Check the valid range of definition.

cfg100 Error : XXXX's ID number is too large.--> <xxx> (xxxx.cfg)
A value is set to the ID number in XXXX definition that exceeds the total number of objects defined.The ID number must be smaller than the total number of objects.

cfg100 Error : Task[x]'s priority is too large.--> <xxx> near line xxx (xxxx.cfg)
The initial priority in task definition of ID number x exceeds the priority in system definition.

cfg100 Error : clock.IPL is too large.--> <xxx> near line xxx (xxxx.cfg)
The system clock interrupt priority level for system clock definition item exceeds the value of IPL within service call of system definition item.

cfg100 Error : System timer's vector <x>conflict near line xxx

A different vector is defined for the system clock timer interrupt vector. Confirm the vector No.x for interrupt vector definition.

cfg100 Error : XXXX is not defined (xxxx.cfg)

"XXXX" item must be set in your configuration file.

cfg100 Error : System's default is not defined

These items must be set int the default configuration file.

cfg100 Error : <XXXX> is already defined near line xxx (xxxx.cfg)

XXXX is already defined. Check and delete the extra definition.

cfg100 Error : XXXX[x] is already defined near line xxx (default.cfg)
cfg100 Error : XXXX[x] is already defined near line xxx (xxxx.cfg)

The ID number in item XXXX is already registered. Modify the ID number or delete the extra definition.

cfg100 Error : XXXX must be defined near line xxx (xxxx.cfg)

XXXX cannot be omitted.

cfg100 Error : SYMBOL must be defined near line xxx (xxxxcfg)

This symbol cannot be omitted.

cfg100 Error : Zero divide error near line xxx (xxxx.cfg)

A zero divide operation occurred in some arithmetic expression.

cfg100 Error : task[X].stack_size must set XX or more near line xxx (xxxx.cfg)

You must set more than XX bytes.in task[x].stack_size.

cfg100 Error : "R2R0" must be contained in task[x].context near line xxxx (xxxx.cfg)

You must select R2R0 register in task[x].context.

cfg100 Error : Can't specify B or F switch when os_int=YES. (xxxx.cfg)

"/B" and "/F" switch cannot be specified to a kernel interrupt handler.

cfg100 Error : Can't specify B and E switch at a time when os_int=NO. (xxxx.cfg)

"/B" and "/E" switch cannot be specified to the non-kernel interrupt handler at a time.

cfg100 Error : interrupt_vector[%ld].os_int must be YES. (xxxx.cfg)

When a kernel interrupt mask level is 7, an interrupt handler must be kernel interrupt handler..

cfg100 Error : system_IPL should be 7. (xxxx.cfg)

When "-Eipl" is specified as the command option of configurator, the value of sysrem_IPL of a system definition must be 7.

cfg100 Error : Timer counter value is overflow. (xxxx.cfg)

Overflow occurred in the operation of a timer count. A timer cannot be initialized with the time tick cycle and peripheral clock which were specified. Please initialize the timer and sets clock.timer to “OTHER”.

Warning messages

The following message are a warning. A warning can be ignored providing that its content is understood.

cfg100 Warning : system is not defined (xxxx.cfg)

cfg100 Warning : system.XXXX is not defined (xxxx.cfg)

System definition or system definition item XXXX is omitted in the configuration file.

cfg100 Warning : task[x].XXXX is not defined near line xxx (xxxx.cfg)

The task definition item XXXX in ID number is omitted.

cfg100 Warning : Already definition XXXX near line xxx (xxxx.cfg)

XXXX has already been defined.The defined content is ignored, check to delete the extra definition.

cfg100 Warning : interrupt_vector[x]'s default is not defined (default.cfg)

The interrupt vector definition of vector number x in the default configuration file is missing.

cfg100 Warning : interrupt_vector[x]'s default is not defined near line xxx (xxxx.cfg)

The interrupt vector of vector number x in the configuration file is not defined in the default configuration file.

cfg100 Warning : Initial start task is not defined

The task of task ID number 1 was defined as the initial startup task because no initial startup task is defined in the configuration file.

cfg100 Warning : system.stack_size is an uneven number near line xxx
cfg100 Warning : task[x].stack_size is an uneven number near line xxx

Please set even size in system.stack_size or task[x].stack_size.

cfg100 Warning : system_IPL should be 7

When "-Wipl" is specified as the command option of KONFIGYURETA, you should make the value of sysrem_IPL of a system definition 7.

cfg100 Warning : Timer counter value is less than your settimg time

The error occurred in the operation of a timer count. Please check whether an error is permitted.

cfg100 Warning : XXXX is specified as YYYY.

XXXX is specified as YYYY.

9. Sample Program Description

9.1 Overview of Sample Program

As an example application of MR100, the following shows a program that outputs a string to the standard output device from one task and another alternately.

Table 9.1 Functions in the Sample Program

Function Name	Type	ID No.	Priority	Description
main()	Task	1	1	Starts task1 and task2.
task1()	Task	2	2	Outputs "task1 running."
task2()	Task	3	3	Outputs "task2 running."
cyh1()	Handler	1		Wakes up task1().

The content of processing is described below.

- The main task starts task1, task2, and cyh1, and then terminates itself.
- task1 operates in order of the following.
  1. Gets a semaphore.
  2. Goes to a wakeup wait state.
  3. Outputs "task1 running."
  4. Frees the semaphore.
- task2 operates in order of the following.
  1. Gets a semaphore.
  2. Outputs "task2 running."
  3. Frees the semaphore.

cyh1 starts every 100 ms to wake up task1.

9.2 Program Source Listing

```
/************************************************************************
*                        MR100  smaple program
*
* COPYRIGHT(C) 2003(2005) RENESAS TECHNOLOGY CORPORATION
* AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
*
*
*     $Id: demo.c,v 1.2 2005/06/15 05:29:02 inui Exp $
************************************************************************/

#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
#include <stdio.h>


void main( VP_INT stacd )
{
     sta_tsk(ID_task1,0);
     sta_tsk(ID_task2,0);
     sta_cyc(ID_cyh1);
}
void task1( VP_INT stacd )
{
     while(1){
             wai_sem(ID_sem1);
             slp_tsk();
             printf("task1 running\n");
             sig_sem(ID_sem1);
     }
}

void task2( VP_INT stacd )
{
     while(1){
             wai_sem(ID_sem1);
             printf("task2 running\n");
             sig_sem(ID_sem1);
     }
}

void cyh1( VP_INT exinf )
{
     iwup_tsk(ID_task1);
}
```

9.3 Configuration File

```
//*************************************************************************
//
//   COPYRIGHT(C) 2003,2005 RENESAS TECHNOLOGY CORPORATION
//   AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
//
//     MR100 System Configuration File.
//       "$Id: smp.cfg,v 1.5 2005/06/15 05:41:54 inui Exp $"
//
//*************************************************************************

// System Definition
system{
       stack_size     = 1024;
     priority        = 10;
       system_IPL     = 4;
     tic_nume        = 1;
     tic_deno        = 1;
     message_pri     = 255;
};
//System Clock Definition
clock{
     mpu_clock               = 20MHz;
       timer                 = A0;
       IPL                   = 4;
};
//Task Definition
//
task[]{
     entry_address   = main();
     name            = ID_main;
       stack_size     = 100;
     priority        = 1;
     initial_start   = ON;
};
task[]{
     entry_address   = task1();
     name            = ID_task1;
       stack_size     = 500;
     priority        = 2;
};
task[]{
     entry_address   = task2();
     name            = ID_task2;
     stack_size      = 500;
     priority        = 3;
};

semaphore[]{
     name            = ID_sem1;
     max_count       = 1;
     initial_count   = 1;
     wait_queue      = TA_TPRI;
};



cyclic_hand [1] {
     name                    = ID_cyh1;
     interval_counter        = 100;
     start                   = OFF;
     phsatr                  = OFF;
     phs_counter             = 0;
     entry_address           = cyh1();
     exinf                   = 1;
};
```

10. Stack Size Calculation Method

10.1 Stack Size Calculation Method

The MR100 provides two kinds of stacks: the system stack and the user stack. The stack size calculation method differ between the stacks.

- User stack

  This stack is provided for each task. Therefore, writing an application by using the MR100 requires to allocate the stack area for each stack.

- System stack

  This stack is used inside the MR100 or during the execution of the handler.

  When a task issues a service call, the MR100 switches the user stack to the system stack. (See Figure 10.1:System Stack and User Stack)

  The system stack uses interrupt stack(ISP).

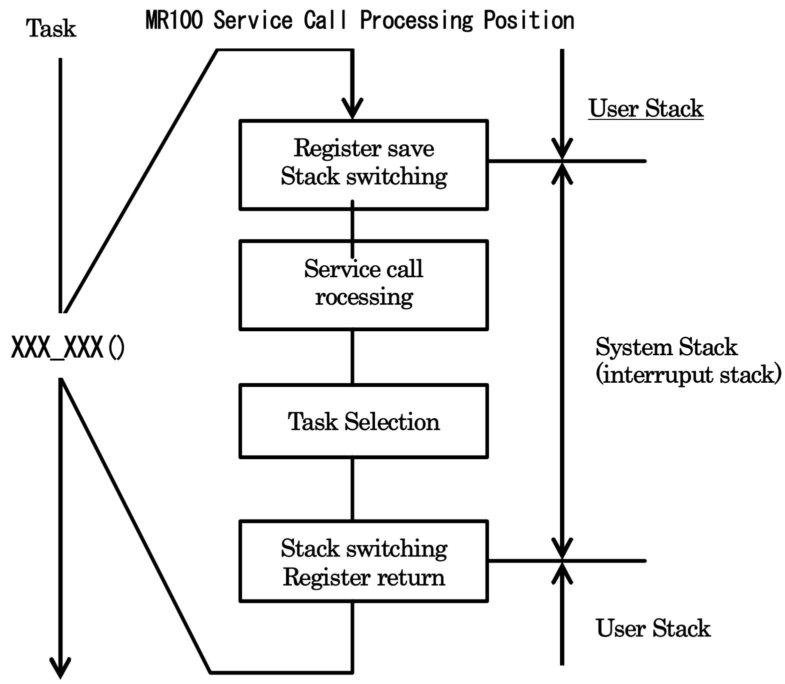

Figure 10.1:System Stack and User Stack

The sections of the system stack and user stack each are located in the manner shown below. However, the diagram shown below applies to the case where the stack areas for all tasks are located in the stack section during configuration.

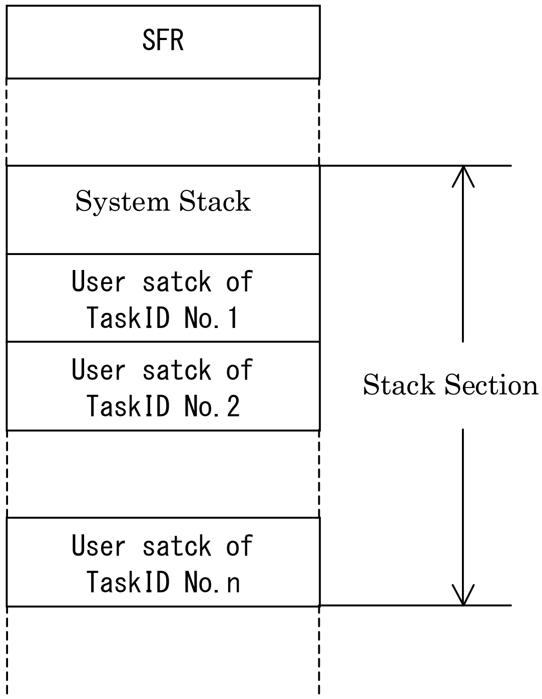

Figure 10.2: Layout of Stacks

10.1.1 User Stack Calculation Method

User stacks must be calculated for each task. The following shows an example for calculating user stacks in cases when an application is written in the C language and when an application is written in the assembly language.

- When an application is written in the C language

  Using the stack size calculation utility of NC100, calculate the stack size of each task. The necessary stack size of a task is the sum of the stack size output by the stack size calculation utility plus a context storage area of 48 bytes[54]

- When an application is written in the assembly language

  - **Sections used in user program**

    The necessary stack size of a task is the sum of the stack size used by the task in subroutine call plus the size used to save registers to a stack in that task.

  - **Sections used in MR100**

    The sections used in MR100 refer to a stack size that is used for the service calls issued.

    MR100 requires that if you issue only the service calls that can be issued from tasks, 8bytes of area be allocated for storing the PC and FLG registers. Also, if you issue the service calls that can be issued from both tasks and handlers, see the stack sizes listed in Table 10.2 Stack Sizes Used by Service Calls Issued from Handlers (in bytes) to ensure that the necessary stack area is allocated.

    Furthermore, when issuing multiple service calls, include the maximum value of the stack sizes used by those service calls as the sections used by MR100 as you calculate the necessary stack size.

Therefore,

User stack size =
Sections used in user program + registers used + Sections used in MR100

(registers used is total size of used registers.)

Figure 2.3:Example of Use Stack Size Calculation shows an example for calculating a user stack. In the example below, the registers used by the task are R2R0, R3R1, and A0.

[54] If written in the C language, this size is fixed.

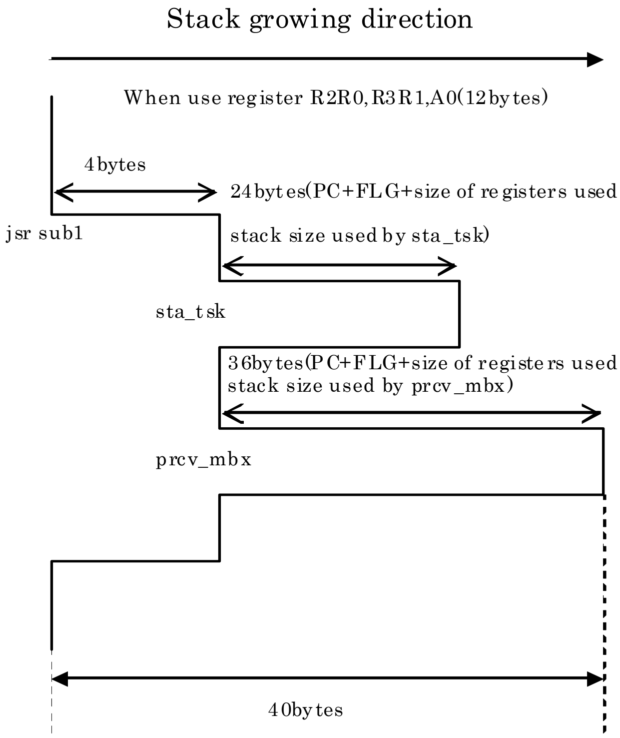

Figure 2.3:Example of Use Stack Size Calculation

10.1.2 System Stack Calculation Method

The system stack is most often consumed when an interrupt occurs during service call processing followed by the occurrence of multiple interrupts.[55] The necessary size (the maximum size) of the system stack can be obtained from the following relation:

Necessary size of the system stack = $\alpha \Sigma \beta i(\gamma)$

- α

  The maximum system stack size among the service calls to be used.[56].

  When sta_tsk, ext_tsk, and dly_tsk are used for example, according to the Table 10.1 Stack Sizes Used by Service Calls Issued from Tasks (in bytes),each of system stack size is the following.

Service Call name	System Stack Size
sta_tsk	4 bytes
ext_tsk	4 bytes
slp_tsk	4 bytes
dly_tsk	8 bytes

  Therefore,the maximum system stack size among the service calls to be used is the 8 bytes of dly_tsk.

- βi

  The stack size to be used by the interrupt handler.[57] The details will be described later.

- γ

  Stack size used by the system clock interrupt handler. This is detailed later.

[55] After switchover from user stack to system stack
[56] Refer from Table 10.1 Stack Sizes Used by Service Calls Issued from Tasks (in bytes) to Table 10.3 Stack Sizes Used by Service Calls Issued from Tasks and Handlers (in bytes) for the system stack size used for each individual service call.
[57] Kernel interrupt handler (not including the system clock interrupt handler here) and non-kernel interrupt handler.

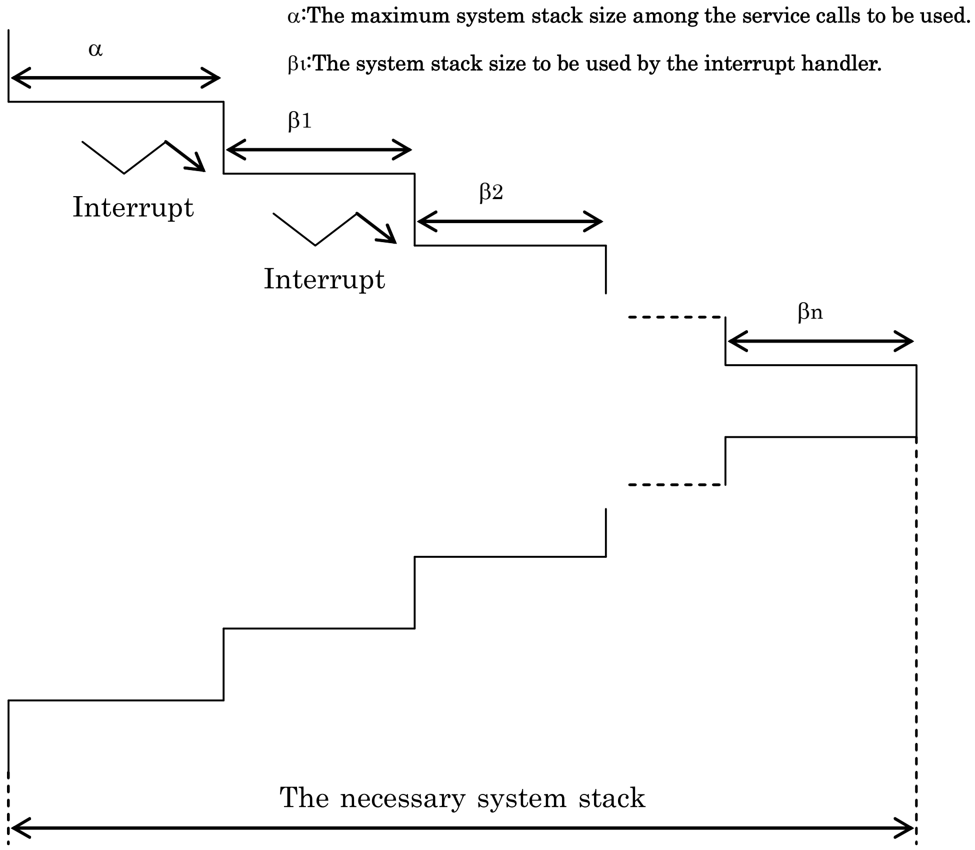

Figure 10.4: System Stack Calculation Method

[(Stack size βi used by interrupt handlers)]

The stack size used by an interrupt handler that is invoked during a service call can be calculated by the equation below.

The stack size βi used by an interrupt handler is shown below.

C language

Using the stack size calculation utility of NC100, calculate the stack size of each interrupt handler.

Refer to the manual of for the stack size calculation utility detailed use of it.

Assembly language

The stack size to be used by kernel interrupt handler
= register to be used + user size + stack size to be used by service call

The stack size to be used by non-kernel interrupt handler
= register to be used + user size

User size is the stack size of the area written by user.

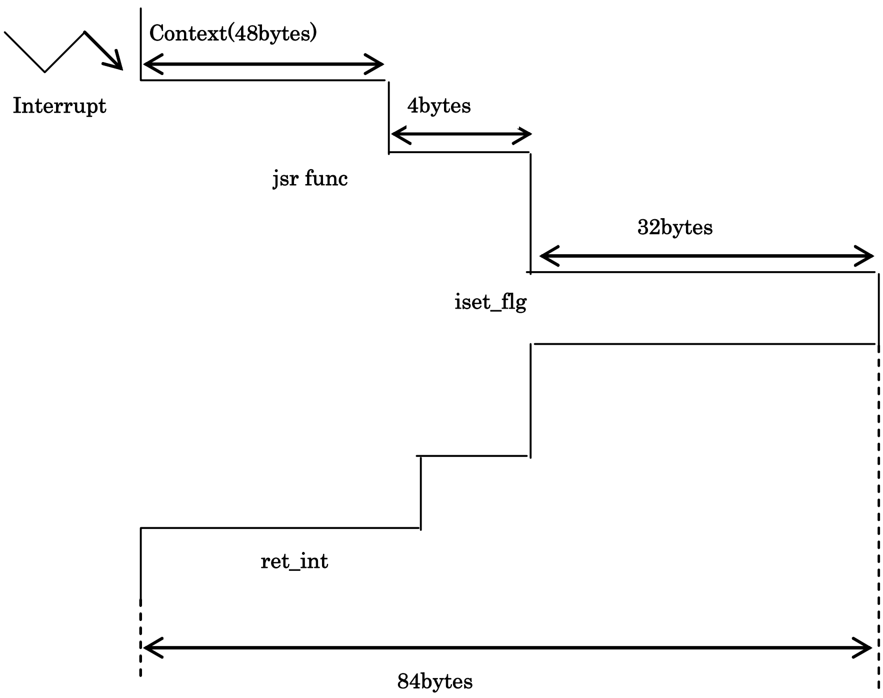

Figure 10.5: Stack size to be used by Kernel Interrupt Handler(Written in C language)

[(System stack size γ used by system clock interrupt handler)]

When you do not use a system timer, there is no need to add a system stack used by the system clock interrupt handler.

The system stack size γ used by the system clock interrupt handler is whichever larger of the two cases below:

48 + maximum size used by cyclic handler
48 + maximum size used by alarm handler
72 bytes

C language

Using the stack size calculation utility of NC100, calculate the stack size of each Alarm or Cyclic handler.

Refer to the manual of the stack size calculation utilityr for detailed use of it.

Assembly language

The stack size to be used by Alarm or Cyclic handler
= register to be used + user size + stack size to be used by service call

If neither cyclic handler nor alarm handler is used, then

γ = 72 bytes

When using the interrupt handler and system clock interrupt handler in combination, add the stack sizes used by both.

10.2 Necessary Stack Size

Table 10.1 Stack Sizes Used by Service Calls Issued from Tasks (in bytes) lists the stack sizes (system stack) used by service calls that can be issued from tasks.

Table 10.1 Stack Sizes Used by Service Calls Issued from Tasks (in bytes)

Service call	Stack size		Service call	Stack size	
	User stack	System stack		User stack	System stack
act_tsk	0(4)	4	rcv_mbx	4	28
can_act	0(12)	0	prcv_mbx	0(16)	0
sta_tsk	0(4)	4	trcv_mbx	0(4)	28
ext_tsk	0	4	ref_mbx	0(8)	0
ter_tsk	0(4)	16	get_mpf	4	28
chg_pri	0(4)	16	pget_mpf	0(20)	0
get_pri	0(12)	0	tget_mpf	4	32
ref_tsk	0(32)	0	rel_mpf	0(4)	16
ref_tst	0(12)	0	ref_mpf	0(8)	0
slp_tsk	0(4)	4	pget_mpl	4	74
tslp_tsk	0(4)	8	rel_mpl	0(4)	38
wup_tsk	0(4)	16	ref_mpl	0(20)	0
can_wup	0(12)	0	set_tim	0(8)	0
rel_wai	0(4)	16	get_tim	0(8)	0
sus_tsk	0(4)	4	sta_cyc	0(12)	0
rsm_tsk	0(4)	4	stp_cyc	0(8)	0
frsm_tsk	0(4)	4	ref_cyc	0(16)	0
dly_tsk	0(4)	8	sta_alm	0(12)	0
sig_sem	0(4)	16	stp_alm	0(12)	0
wai_sem	0(4)	28	ref_alm	0(16)	0
pol_sem	0(8)	0	rot_rdq	0(4)	0
twai_sem	0(4)	28	get_tid	0(8)	0
ref_sem	0(12)	0	loc_cpu	0	0
set_flg	0(4)	24	unl_cpu	0(4)	0
clr_flg	0(8)	0	ref_ver	0(12)	0
wai_flg	4	28	vsnd_dtq	0(4)	28
pol_flg	0(8)	0	vpsnd_dtq	0(4)	16
twai_flg	4	28	vtsnd_dtq	0(4)	28
ref_flg	0(8)	0	vfsnd_dtq	0(4)	16
snd_dtq	0(4)	28	vrcv_dtq	4	16
psnd_dtq	0(4)	16	vprcv_dtq	4	16
tsnd_dtq	0(4)	28	vtrcv_dtq	4	16
fsnd_dtq	0(4)	16	vref_dtq	0(8)	0
rcv_dtq	4	16	vrst_dtq	0(4)	48
prcv_dtq	4	16	vrst_vdtq	0(4)	48
trcv_dtq	4	16	vrst_mbx	0(8)	0
ref_dtq	0(8)	0	vrst_mpf	0(4)	48
snd_mbx	0(4)	12	vrst_mpl	0	28(68)
dis_dsp	0	0	ena_dsp	0(4)	0

(): Stack sizes used by service call in Assembly programs.

Table 10.2 Stack Sizes Used by Service Calls Issued from Handlers (in bytes) lists the stack sizes (system stack) used by service calls that can be issued from handlers.

Table 10.2 Stack Sizes Used by Service Calls Issued from Handlers (in bytes)

Service call	Stack size	Service call	Stack size
iact_tsk	12(40)	iprcv_mbx	16(28)
ican_act	12(24)	iref_mbx	12(20)
ista_tsk	12(40)	ipget_mpf	28(32)
ichg_pri	24(52)	irel_mpf	32(64)
iget_pri	16(24)	iref_mpf	12(20)
iref_tsk	12(44)	iset_tim	12(20)
iref_tst	12(24)	iget_tim	12(20)
iwup_tsk	24(56)	ista_cyc	12(24)
ican_wup	12(24)	istp_cyc	12(20)
irel_wai	24(56)	iref_cyc	12(28)
isus_tsk	12(32)	ista_alm	12(24)
irsm_tsk	12(40)	istp_alm	12(24)
ifrsm_tsk	12(40)	iref_alm	12(28)
isig_sem	28(60)	irot_rdq	12(24)
ipol_sem	12(20)	iget_tid	16(20)
iref_sem	12(20)	iloc_cpu	12
iset_flg	32(68)	iunl_cpu	12(20)
iclr_flg	12(20)	ret_int	16
ipol_flg	16(24)	iref_ver	12(24)
iref_flg	12(20)	vipsnd_dtq	32(64)
ipsnd_dtq	28(60)	vifsnd_dtq	32(64)
ifsnd_dtq	28(60)	viprcv_dtq	36(64)
iprcv_dtq	40(64)	viref_dtq	12(20)
iref_dtq	12(20)	isnd_mbx	24(52)
iref_mpl	12(20)		

(): Stack sizes used by service call in Assembly programs.

Table 10.3 Stack Sizes Used by Service Calls Issued from Tasks and Handlers (in bytes) lists the stack sizes (system stack) used by service calls that can be issued from both tasks and handlers. If the service call issued from task, system uses user stack. If the service call issued from handler, system uses system stack.

Table 10.3 Stack Sizes Used by Service Calls Issued from Tasks and Handlers (in bytes)

Service call	Stack size	Service call	Stack size
sns_ctx	12(20)	sns_loc	12(20)
sns_dsp	12(20)	sns_dpn	12(20)

(): Stack sizes used by service call in Assembly programs.

11. Note

11.1 The Use of INT Instruction

MR100 has INT instruction interrupt numbers reserved for issuing service calls as listed in Table 11.1 Interrupt Number Assignment. For this reason, when using software interrupts in a user application, do not use interrupt numbers 63 through 48 and be sure to use some other numbers.

Table 11.1 Interrupt Number Assignment

Interrupt No.	Service calls Used
249	Service calls that can be issued from only task context
250	Service calls that can be issued from only non-task context. Service calls that can be issued from both task context and non-task context.
251	ret_int service call
252	dis_dsp service call
253	loc_cpu, iloc_cpu service call
254	ext_tsk service call
255	Reserved for future extension

11.2 The Use of registers of bank

The registers of bank is 0, when a task starts on MR100.

MR100 does not change the registers of bank in processing kernel.

You must pay attention to the followings.

- Don’t change the regisers of bank in processing a task.
- If an interrupt handler with regisers of bank 1 have multiple interrupts of an interrupt handler with regisers of bank 1 , the program can not execute normally.

11.3 Regarding Delay Dispatching

MR100 has four service calls related to delay dispatching.

- dis_dsp
- ena_dsp
- loc_cpu
- unl_cpu

The following describes task handling when dispatch is temporarily delayed by using these service calls.

14. When the execution task in delay dispatching should be preempted

While dispatch is disabled, even under conditions where the task under execution should be preempted, no time is dispatched to new tasks that are in an executable state. Dispatching to the tasks to be executed is delayed until the dispatch disabled state is cleared. When dispatch is being delayed.

- Task under execution is in a RUN state and is linked to the ready queue
- Task to be executed after the dispatch disabled state is cleared is in a READY state and is linked to the highest priority ready queue (among the queued tasks).

15. isus_tsk,irsm_tsk during dispatch delay

In cases when isus_tsk is issued from an interrupt handler that has been invoked in a dispatch disabled state to the task under execution (a task to which dis_dsp was issued) to place it in a SUSPEND state. During delay dispatching.

- The task under execution is handled inside the OS as having had its delay dispatching cleared. For this reason, in isus_tsk that has been issued to the task under execution, the task is removed from the ready queue and placed in a SUSPEND state. Error code E_OK is returned. Then, when irsm_tsk is issued to the task under execution, the task is linked to the ready queue and error code E_OK is returned. However, tasks are not switched over until delay dispatching is cleared.
- The task to be executed after disabled dispatching is re-enabled is linked to the ready queue.

16. rot_rdq, irot_rdq during dispatch delay

When rot_rdq (TPRI_RUN = 0) is issued during dispatch delay, the ready queue of the own task's priority is rotated. Also, when irot_rdq (TPRI_RUN = 0) is issued, the ready queue of the executed task's priority is rotated. In this case, the task under execution may not always be linked to the ready queue. (Such as when isus_tsk is issued to the executed task during dispatch delay.)

17. Precautions

- No service call (e.g., slp_tsk, wai_sem) can be issued that may place the own task in a wait state while in a state where dispatch is disabled by dis_dsp or loc_cpu.
- ena_dsp and dis_dsp cannot be issued while in a state where interrupts and dispatch are disabled by loc_cpu.
- Disabled dispatch is re-enabled by issuing ena_dsp once after issuing dis_dsp several times.

  The above status transition can be summarized in Table 3.3.

11.4 Regarding Initially Activated Task

MR100 allows you to specify a task that starts from a READY state at system startup. This specification is made by setting the configuration file.

Refer to 8.1.2 for details on how to set.

12. Appendix

12.1 Data Type

```
typedef    signed char       B;          /* Signed 8-bit integer */
typedef    signed short      H;          /* Signed 16-bit integer */
typedef    signed long       W;          /* Signed 32-bit integer */
typedef    unsigned char     UB;         /* Unsigned 8-bit integer */
typedef    unsigned short    UH;         /* Unsigned 16-bit integer */
typedef    unsigned long     UW;         /* Unsigned 32-bit integer */
typedef    char              VB          /* 8-bit value with unknown data type */
typedef    short             VH;         /* 16-bit value with unknown data type */
typedef    long              VW;         /* 32-bit value with unknown data type */
typedef    void              *VP;        /* Pointer to unknown data type */
typedef    void              (*FP)();    /* Pointer to a function */
typedef    W                 INT         /* Signed 32-bit integer */
typedef    UW                UINT;       /* Unsigned 32-bit integer */
typedef    H                 ID;         /* Object ID number */
typedef    H                 PRI;        /* Priority */
typedef    W                 TMO;        /* Timeout */
typedef    H                 ER;         /* Error code(Signed integer) */
typedef    UH                ATR;        /* Object attribute(Unsigned integer) */
typedef    UH                STAT;       /* Task status */
typedef    UH                MODE;       /* Service call operation mode */
typedef    UW                SIZE;       /* Memory area size */
typedef    UW                RELTIM      /* Relative time */
typedef    W                 VP_INT;     /* Pointer to an unknown data type, or a signed in-
                                            teger for the processor */
typedef    struct            systim{     /* System time */
           UH                utime;      /* Upper16bit of the system time */
           UW                ltimer;     /* Lower32bit of the system time */
}    SYS-
TIM;
typedef    W                 ER_UINT;    /* Error code or unsigned integer */
```

12.2 Common Constants and Packet Format of Structure

```
----Common formats----
TRUE          1                 /* True */
FALSE         0                 /* False */
----Formats related to task management----
TSK_SELF      0                 /* Specifies the issuing task itself */
TPRI_RUN      0                 /* Specifies priority of task being executed then */
typedef struct t_rtsk {
    STAT      tskstat;          /* Task status */
    PRI       tskpri;           /* Current priority of task */
    PRI       tskbpri;          /* Base priority of task */
    STAT      tskwait;          /* Reason for which task is kept waiting */
    ID        wid;              /* Object ID for which task is kept waiting */
    TMO       tskatr;           /* Remaining time before task times out */
    UINT      actcnt;           /* Number of activation requests */
    UINT      wupcnt;           /* Number of wakeup requests */
    UINT      suscnt;           /* Number of suspension requests */
} T_RTSK;
typedef struct t_rtst {
    STAT      tskstat;          /* Task status */
    STAT      tskwait;          /* Reason for which task is kept waiting */
} T_RTST;
----Formats related to semaphore----
typedef struct t_rsem {
    ID        wtskid;           /* ID number of task at the top of waiting queue */
    INT       semcnt;           /* Current semaphore count value */
} T_RSEM;
----Formats related to eventflag----
wfmod:
TWF_ANDW      H'0000            /* AND wait */
TWF_ORW       H'0002            /* OR wait */
typedef struct t_rflg {
    ID        wtskid;           /* ID number of task at the top of waiting queue */
    UINT      flgptn;           /* Current bit pattern of eventflag */
} T_RFLG;
----Formats related to data queue and short data queue----
typedef struct t_rdtq {
        ID    stskid;           /* ID number of task at the top of transmission waiting queue */
        ID    rtskid;           /* ID number of task at the top of reception waiting queue */
        UINT  sdtqcnt;          /* Number of data bytes contained in data queue */
} T_RDTQ;
----Formats related to mailbox----
typedef struct    t_msg   {
    VP  msghead;                /* Message header */
} T_MSG;
typedef struct t_msg_pri {
    T_MSG     msgque;           /* Message header */
    PRI       msgpri;           /* Message priority */
} T_MSG_PRI;
typedef struct t_mbx {
    ID        wtskid;           /* ID number of task at the top of waiting queue */
    T_MSG     *pk_msg;          /* Next message to be received */
} T_RMBX;
----Formats related to fixed-size memory pool----
typedef struct t_rmpf {
    ID        wtskid;           /* ID number of task at the top of memory acquisition waiting queue
    */
    UINT      frbcnt;           /* Number of memory blocks */
} T_RMPF;
```

```
----Formats related to Variable-size Memory pool----
typedef struct t_rmpl {
      ID          wtskid;       /* ID number of task at the top of memory acquisition waiting queue */
      SIZE        fmplsz;       /* Total size of free areas */
      UINT        fblksz;       /* Maximum memory block size that can be acquired immediately */
} T_RMPL;
----Formats related to cyclic handler----
typedef struct t_rcyc {
      STAT        cycstat;      /* Operating status of cyclic handler */
      RELTIM      lefttim;      /* Remaining time before cyclic handler starts */
} T_RCYC;
----Formats related to alarm handler----
typedef struct t_ralm {
      STAT        almstat;      /* Operating status of alarm handler */
      RELTIM      lefttim;      /* Remaining time before alarm handler starts */
} T_RALM;
----Formats related to system management----
typedef struct t_rver {
      UH          maker;        /* Maker */
      UH          prid;         /* Type number */
      UH          spver;        /* Specification version */
      UH          prver;        /* Product version */
      UH          prno[4];      /* Product management information */
} T_RVER;
```

12.3 Assembly Language Interface

When issuing a service call in the assembly language, you need to use macros prepared for invoking service calls.

Processing in a service call invocation macro involves setting each parameter to registers and starting execution of a service call routine by a software interrupt. If you issue service calls directly without using a service call invocation macro, your program may not be guaranteed of compatibility with future versions of MR100.

The table below lists the assembly language interface parameters. The values set forth in μITRON specifications are not used for the function code.

Task Management Function

ServiceCall	INTNo.	Parameter					ReturnParameter	
		FuncCode A0	R1	R2	R3	A1	R0	R2
ista_tsk	250	8	stacd	tskid	stacd	-	ercd	-
sta_tsk	249	6	stacd	tskid	stacd	-	ercd	-
act_tsk	249	0	-	tskid	-	-	ercd	-
iact_tsk	250	2	-	tskid	-	-	ercd	-
ter_tsk	249	10	-	tskid	-	-	ercd	-
can_act	250	4	-	tskid	-	-	actcnt	-
ican_act	250	4	-	tskid	-	-	actcnt	-
chg_pri	250	12	-	tskid	tskpri	-	ercd	-
ichg_pri	250	14	-	tskid	tskpri	-	ercd	-
rel_wai	249	32	-	tskid	-	-	ercd	-
irel_wai	250	34	-	tskid	-	-	ercd	-
ref_tst	250	20	-	tskid	-	pk_rtst	ercd	-
iref_tst	250	20	-	tskid	-	pk_rtst	ercd	-
ref_tsk	250	18	-	tskid	-	pk_rtsk	ercd	-
iref_tsk	250	18	-	tskid	-	pk_rtsk	ercd	-
ext_tsk	137	106	-	-	-	-	-	-
get_pri	250	16	-	tskid	-	-	ercd	tskpri
iget_pri	250	16	-	tskid	-	-	ercd	tskpri

Task Dependent Synchronization Function

ServiceCall	INTNo.	Parameter			ReturnParameter
		FuncCode A0	R2	R6R4	R0
slp_tsk	249	22	-	-	ercd
wup_tsk	249	26	tskid	-	ercd
iwup_tsk	250	28	tskid	-	ercd
can_wup	250	30	tskid	-	wupcnt
ican_wup	250	30	tskid	-	wupcnt
tslp_tsk	249	24	-	tmout	ercd
sus_tsk	249	36	tskid	-	ercd
isus_tsk	250	38	tskid	-	ercd
rsm_tsk	249	40	tskid	-	ercd
irsm_tsk	250	42	tskid	-	ercd
frsm_tsk	249	40	tskid	-	ercd
ifrsm_tsk	250	42	tskid	-	ercd
dly_tsk	249	44	-	tmout	ercd
rel_wai	249	32	tskid	-	ercd
irel_wai	250	34	tskid	-	ercd

Synchronization & Communication Function

ServiceCall	INTNo.	Parameter						ReturnParameter	
		FuncCode A0	R0	R3R1	R2	R6R4	A1	R0	R3R1
wai_sem	249	50	-	-	semid	-	-	ercd	-
pol_sem	250	52	-	-	semid	-	-	ercd	-
ipol_sem	250	52	-	-	semid	-	-	ercd	-
sig_sem	249	46	-	-	semid	-	-	ercd	-
isig_sem	250	48	-	-	semid	-	-	ercd	-
twai_sem	249	54	-	-	semid	tmout	-	ercd	-
ref_sem	250	56	-	-	semid	-	pk_rsem	ercd	-
iref_sem	250	56	-	-	semid	-	pk_rsem	ercd	-
wai_flg	249	64	wfmode	waiptn	flgid	-	-	ercd	flgptn
twai_flg	249	92	wfmode	waiptn	flgid	tmout	-	ercd	fgptn
pol_flg	250	66	wfmode	waiptn	flgid	-	-	ercd	flgptn
ipol_flg	250	66	wfmode	waiptn	flgid	-	-	ercd	flgptn
set_flg	249	58	-	setptn	flgid	-	-	ercd	-
iset_flg	250	60	-	setptn	flgid	-	-	ercd	-
ref_flg	250	70	-	-	flgid	-	pk_rflg	ercd	-
iref_flg	250	70	-	-	flgid	-	pk_rflg	ercd	-
clr_flg	250	62	-	clrptn	flgid	-	-	ercd	-
iclr_flg	250	62	-	clrptn	flgid	-	-	ercd	-
snd_dtq	249	72	-	data	dtqid	-	-	ercd	-
psnd_dtq	249	74	-	data	dtqid	-	-	ercd	-
ipsnd_dtq	250	76	-	data	dtqid	-	-	ercd	-
fsnd_dtq	249	80	-	data	dtqid	-	-	ercd	-
ifsnd_dtq	250	82	-	data	dtqid	-	-	ercd	-
tsnd_dtq	249	104	-	data	dtqid	tmout	-	ercd	-

Synchronization & Communication Function

ServiceCall	INTNo.	Parameter					ReturnParameter		
		FuncCode A0	R3R1	R2	R6R4	A1	R0	R3R1	A1
rcv_dtq	249	84	-	dtqid	-	-	ercd	data	-
prcv_dtq	249	86	-	dtqid	-	-	ercd	data	-
iprcv_dtq	250	88	-	dtqid	-	-	ercd	data	-
trcv_dtq	249	90	-	dtqid	tmout	-	ercd	data	-
ref_dtq	250	92	-	dtqid	-	pk_rdtq	ercd	-	-
iref_dtq	250	92	-	dtqid	-	pk_rdtq	ercd	-	-
snd_mbx	249	94	-	mbxid	-	pk_msg	ercd	-	-
isnd_mbx	250	96	-	mbxid	-	pk_msg	ercd	-	-
rcv_mbx	249	98	-	mbxid	-	-	ercd	-	pk_msg
prcv_mbx	250	100	-	mbxid	-	-	ercd	-	pk_msg
iprcv_mbx	250	100	-	mbxid	-	-	ercd	-	pk_msg
trcv_mbx	249	102	-	mbxid	tmout	-	ercd	-	pk_msg
ref_mbx	250	104	-	mbxid	-	pk_rmbx	ercd	-	-
iref_mbx	250	104	-	mbxid	-	pk_rmbx	ercd	-	-

Interrupt Management Functions

ServiceCall	INTNo.	Parameter	ReturnParameter
		FuncCode A0	R0
ret_int	251	--	--

System State Management Functions

ServiceCall	INTNo.	Parameter		ReturnParameter	
		FuncCode A0	R3	R0	R2
rot_rdq	249	140	tskpri	ercd	-
irot_rdq	250	142	tskpri	ercd	-
get_tid	250	144	-	ercd	tskid
iget_tid	250	144	-	ercd	tskid
loc_cpu	253	198	-	ercd	-
iloc_cpu	253	200	-	ercd	-
dis_dsp	252	206	-	ercd	-
ena_dsp	249	150	-	ercd	-
unl_cpu	249	146	-	ercd	-
iunl_cpu	250	148	-	ercd	-
sns_ctx	250	152	-	ercd	-
sns_loc	250	154	-	ercd	-
sns_dsp	250	156	-	ercd	-
sns_dpn	250	158	-	ercd	-

Memorypool Management Functions

Service-Call	INT-No.	Parameter						ReturnParam-eter	
		Func-Code A0	R1	R2	R3	R6R4	A1	R0	R3R1
get_mpf	249	108	-	mpfid	-	-	-	ercd	p_blk
pget_mpf	250	106	-	mpfid	-	-	-	ercd	p_blk
ipget_mpf	250	106	-	mpfid	-	-	-	ercd	p_blk
tget_mpf	249	110	-	mpfid	-	tmout	-	ercd	p_blk
rel_mpf	249	112	blk	mpfid	blk	-	-	ercd	-
irel_mpf	250	114	blk	mpfid	blk	-	-	ercd	-
ref_mpf	250	116	-	mpfid	-	-	pk_rmpf	ercd	-
iref_mpf	250	116	-	mpfid	-	-	pk_rmpf	ercd	-
pget_mpl	249	118	-	mplid	-	-	-	ercd	p_blk
rel_mpl	249	120	blk	mplid	blk	-		ercd	-
ref_mpl	250	122	-	mplid	-	-	pk_rmpl	ercd	-
iref_mpl	250	122	-	mplid	-	-	pk_rmpl	ercd	-

Time Management Functions

ServiceCall	INTNo.	Parameter				ReturnParameter
		FuncCode A0	R2	R6R4	A1	R0
set_tim	250	124	-	-	p_systim	ercd
iset_tim	250	124	-	-	p_systim	ercd
get_tim	250	126	-	-	p_systim	ercd
iget_tim	250	126	-	-	p_systim	ercd
sta_cyc	250	128	cycid		-	ercd
ista_cyc	250	128	cycid		-	ercd
stp_cyc	250	130	cycid		-	ercd
istp_cyc	250	130	cycid		-	ercd
ref_cyc	250	132	cycid		pk_rcyc	ercd
iref_cyc	250	132	cycid		pk_rcyc	ercd
sta_alm	250	134	almid	almtim	-	ercd
ista_alm	250	134	almid	almtim	-	ercd
stp_alm	250	136	almid		-	ercd
istp_alm	250	136	almid		-	ercd
ref_alm	250	138	almid		pk_ralm	ercd
iref_alm	250	138	almid		pk_ralm	ercd

System Configuration Management Functions

ServiceCall	INTNo.	Parameter		ReturnParameter
		FuncCode A0	A1	R0
ref_ver	250	160	pk_rver	ercd
iref_ver	250	160	pk_rver	ercd

Extenden Function(Reset Function)

ServiceCall	INTNo.	Parameter		ReturnParameter
		FuncCode A0	R2	R0
vrst_vdtq	249	192	vdtqid	ercd
vrst_dtq	249	184	dtqid	ercd
vrst_mbx	250	186	mbxid	ercd
vrst_mpf	249	188	mpfid	ercd
vrst_mpl	250	190	mplid	ercd
vrst_mbf	249	218	mbfid	ercd

Extenden Function(Short Data Queue Function)

ServiceCall	INTNo.	Parameter					ReturnParameter	
		FuncCode A0	R1	R2	R6R4	A1	R0	R1
vsnd_dtq	249	162	data	vdtqid	-	-	ercd	-
vpsnd_dtq	249	164	data	vdtqid	-	-	ercd	-
vipsnd_dtq	250	166	data	vdtqid	-	-	ercd	-
vfsnd_dtq	249	170	data	vdtqid	-	-	ercd	-
vifsnd_dtq	250	172	data	vdtqid	-	-	ercd	-
vtsnd_dtq	249	228	data	vdtqid	tmout	-	ercd	-
vrcv_dtq	249	174	-	vdtqid	-	-	ercd	data
vprcv_dtq	249	176	-	vdtqid	-	-	ercd	data
viprcv_dtq	250	178	-	vdtqid	-	-	ercd	data
vtrcv_dtq	249	180	-	vdtqid	tmout	-	ercd	data
vref_dtq	250	182	-	vdtqid	-	pk_rdtq	ercd	-
viref_dtq	250	182	-	vdtqid	-	pk_rdtq	ercd	-

Real-time OS for R32C/100 Series
M3T-MR100/4 User's Manual

Publication Date: September. 16, 2007 Rev.1.00

Published by: Sales Strategic Planning Div.
Renesas Technology Corp.

Edited by: Application Engineering Department 1
Renesas Solutions Corp.

M3T-MR100/4
User's Manual

Renesas Technology Corp.
2-6-2, Ote-machi, Chiyoda-ku, Tokyo, 100-0004, Japan