HITACHI

SOFTWARE MANUAL

CPMS GENERAL DESCRIPTION AND MACRO SPECIFICATIONS

S10V
Programmable Controller

SVE-3-201（A）

First Edition, August 2003, SVE-3-201(A)

Printed in Japan.

BI-NR-MM<IC-NS> (FL-MW20, AI8.0)

SAFETY PRECAUTIONS

- Read this manual thoroughly and follow all the safety precautions and instructions given in this manual before operations such as system configuration and program creation.
- Keep this manual handy so that you can refer to it any time you want.
- If you have any question concerning any part of this manual, contact your nearest Hitachi branch office or service engineer.
- Hitachi will not be responsible for any accident or failure resulting from your operation in any manner not described in this manual.
- Hitachi will not be responsible for any accident or failure resulting from modification of software provided by Hitachi.
- Hitachi will not be responsible for reliability of software not provided by Hitachi.
- Make it a rule to back up every file. Any trouble on the file unit, power failure during file access or incorrect operation may destroy some of the files you have stored. To prevent data destruction and loss, make file backup a routine task.
- Furnish protective circuits externally and make a system design in a way that ensures safety in system operations and provides adequate safeguards to prevent personal injury and death and serious property damage even if the product should become faulty or malfunction or if an employed program is defective.
- If an emergency stop circuit, interlock circuit, or similar circuit is to be formulated, it must be positioned external to the programmable controller. If you do not observe this precaution, equipment damage or accident may occur when the programmable controller becomes defective.
- Before changing the program, generating a forced output, or performing the RUN, STOP, or like procedure during an operation, thoroughly verify the safety because the use of an incorrect procedure may cause equipment damage or other accident.

THIS PAGE INTENTIONALLY LEFT BLANK.

PREFACE

This manual describes the Compact Process Monitor System (CPMS), which is an operating system designed for real-time control of the S10V CMU, centered around the functions of the operating system and the linkage of macro calls. This manual is intended for those who design or develop real-time control programs on the Hitachi 04/R600 System. This manual assumes that readers have a basic knowledge of ordinary operating systems.

<Manual organization>

PART 1 GENERAL DESCRIPTION

CHAPTER 1 OVERVIEW
This chapter describes the configuration and basic function specifications of the CPMS.

CHAPTER 2 TASK MANAGEMENT
This chapter describes the configuration and linkage of tasks and other task functions required in creating real-time control programs.

CHAPTER 3 MEMORY MANAGEMENT
This chapter describes memory management functions such as for allocation and protection of main memory.

CHAPTER 4 TIMER MANAGEMENT
This chapter explains how to manage the time of day and the length of time.

CHAPTER 5 SHARED RESOURCE MANAGEMENT
This chapter describes exclusive control of resources shared by tasks.

CHAPTER 6 I/O DEVICE MANAGEMENT
This chapter explains how to identify I/O devices.

CHAPTER 7 SYSTEM MANAGEMENT
This chapter explains how to start up the system.

CHAPTER 8 TASK ERROR HANDLING
This chapter describes built-in subroutines executed in case of task errors.

CHAPTER 9 SYSTEM SERVICES
This chapter describes functions used to fetch information on the operation status of the system and tasks.

PART 2 MACRO SPECIFICATIONS
This part describes the functions and linkage of CPMS macro calls.

PART 3 LIBRARIES
This part describes the functions and linkage of libraries such as for arithmetic operations.

<Related Manual>

- SOFTWARE MANUAL OPERATION RPDP/S10V for windows® (Manual number SVE-3-133)

Note for storage capacity calculations:

- Memory capacities and requirements, file size and storage requirements, etc. must be calculated according to the formula 2^n. The following examples show the results of such calculations by 2^n (to the right of the equals signs):
  1 KB (kilobyte) = 1024 bytes
  1 MB (megabyte) = 1,048,576 bytes
  1 GB (gigabyte) = 1,073,741,824 bytes
- As for disk capacities, they must be calculated using the formula 10^n. Listed below are the results of calculating the above example capacities using 10^n in place of 2^n:
  1 KB (kilobyte) = 1000 bytes
  1 MB (megabyte) = 1000^2 bytes
  1 GB (gigabyte) = 1000^3 bytes

CONTENTS

PART 1 GENERAL DESCRIPTION

FIGURES

TABLES

PART 1 GENERAL DESCRIPTION

CHAPTER 1 OVERVIEW

1.1 CPMS Functions

The CPMS (Compact Process Monitor System) is the nucleus of the real-time operating system.
The CPMS has the following functions.

- Task management
  Up to 255 tasks can be controlled.
- Memory management
  Memory address conversion and memory protection are controlled.
- Timer management
  The time of day and the length of time maintained by the system are controlled.
- Shared resource management
  Resources shared by tasks are exclusively controlled.
- I/O device management
  A variety of I/O devices are controlled, and I/O drivers are incorporated into the system.
- System management
  System initialization as well as the status and configuration of the system are controlled.
- System services
  Information in the system and system services are offered.

1.2 CPMS Specifications

Table 1-1 shows CPMS specifications (system parameters).

Table 1-1 CPMS Specifications

Item	Value	Remarks
Number tasks	Up to 255	Assign task numbers as follows: 1 to 224: User tasks 225 to 255: System tasks 230 to 255: OS tasks
Task priority	32 levels	Users: 4 to 27 System: 0 to 31
Number of timers	320	Used by the TIMER, DELAY, WAKE macros.
Number of concurrently allocable resources	Up to 16	Used by the RSERV and PRSRV macros.
DHP buffer	128 KB	12 to 36 bytes per case
Error log buffer	32 KB	1 KB per case
Built-in subroutine	10 points	4 entries at each point

1.3 CPMS Structure

The CPMS consists of an exception processing program, dispatcher and system tasks as shown in Figure 1-1.

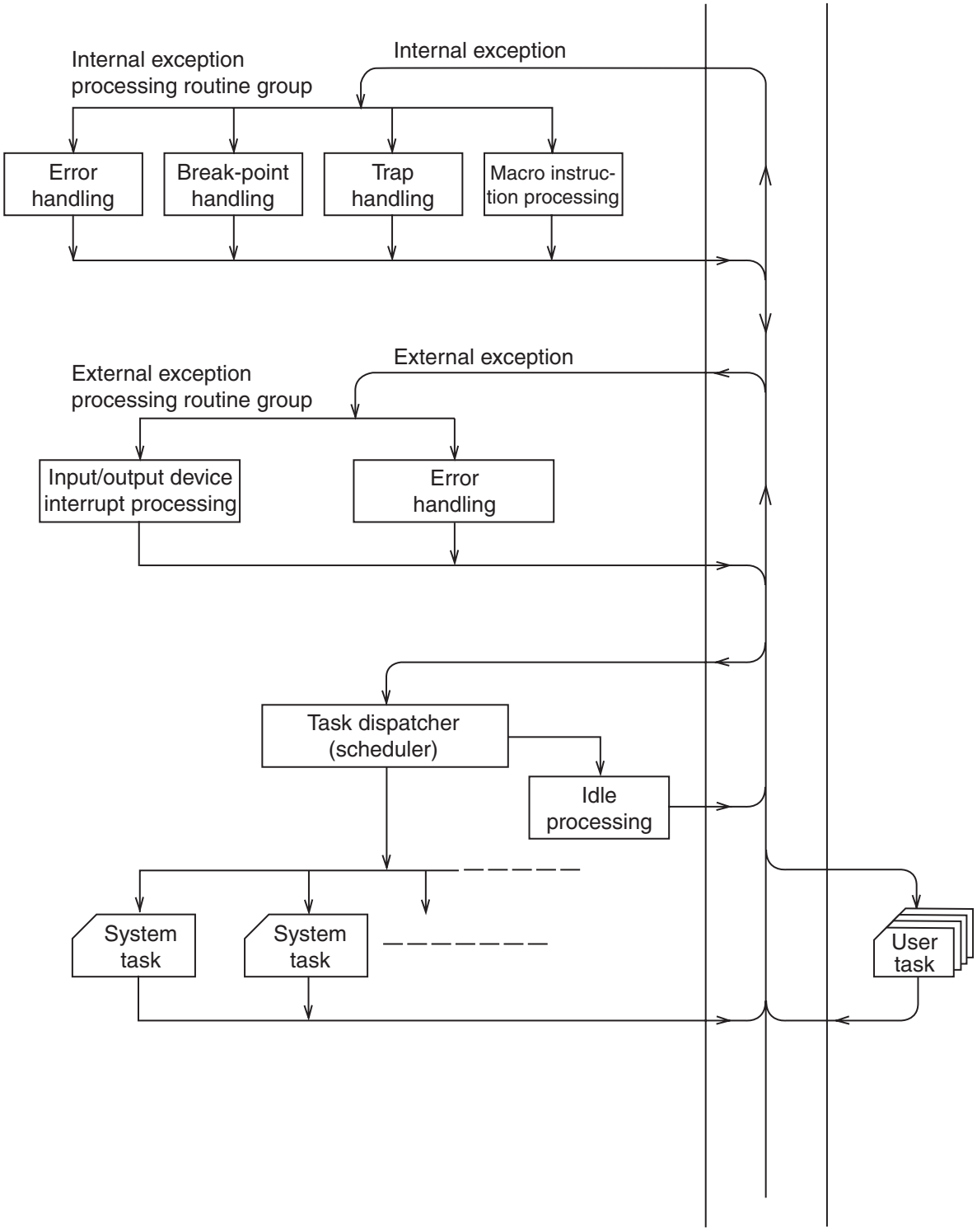


Figure 1-1 CPMS Structure

1.4 CPMS and Hardware

Figure 1-2 shows the relation between the S10V CMU configuration and the CPMS.

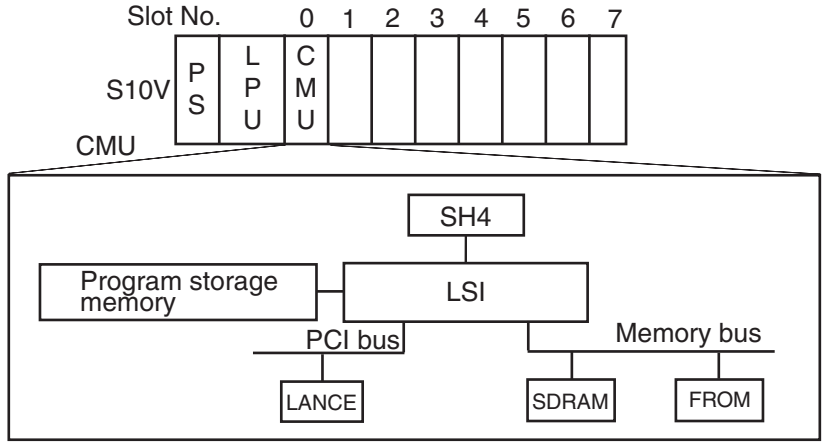


Figure 1-2 Relation between the Hardware Configuration and the CPMS

- Processor (SH4)
  The control program task is operated.
- LSI
  Controls memory access and bus access from the processor.
- Memory bus and memory
  The main memory (SDRAM) and the FROM reside on the memory bus.
  SDRAM: Main memory of the CMU. The OS and program are operated.
  The contents are deleted by turning off the power supply or resetting.
  FROM: Programs such as OS reside.
  Program storage memory: Flash memory that stores the RPDP execution environment, tasks, and HI-FLOW program. At a startup, data is copied from the program storage memory into the SDRAM.

1.5 Interface between the CPMS and Users

The interface between the CPMS and users effects such interactions as operations from the Real-time Program Development Package (RPDP), linking to built-in subroutines, and issuing by user tasks of macro instructions.
The RPDP provides a creating environment for tasks and built-in subroutines.

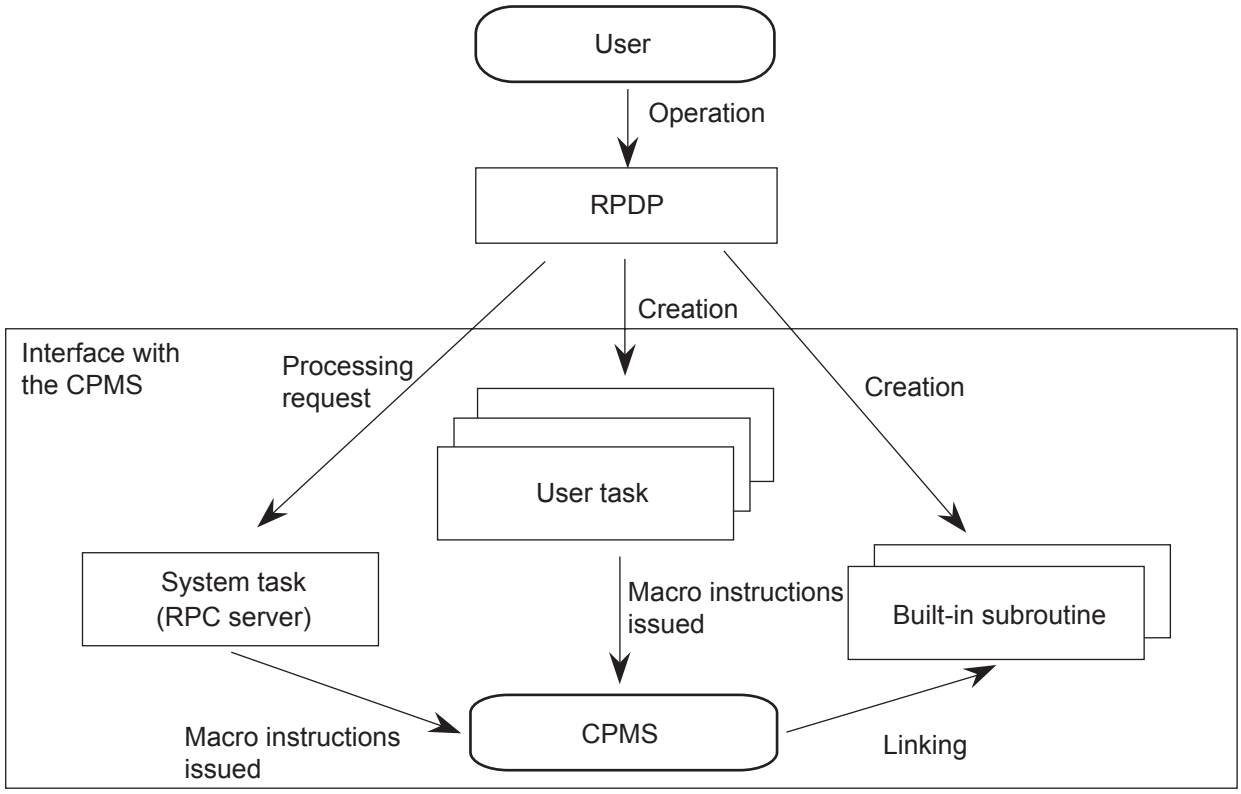


Figure 1-3 Interface between the CPMS and Users

CHAPTER 2 TASK MANAGEMENT

2.1 Task

A task is a unit of work done by the system for program execution. On a task-by-task basis, the CPMS manages program execution and allocates resources.

(1) Task number

The task number (TN) is used to identify a task. CPMS can manage up to 255 tasks. The user can assign task numbers 1 to 224 to user tasks. The task numbers 225 to 229 are reserved for system tasks. And task numbers 230 to 255 are reserved for the OS. CPMS starts up the tasks having task number 1 as initial start tasks.

(2) Task structure

A task consists of the TEXT, DATA, BSS, STACK, and OS work areas. TEXT is a part to be executed in the program. DATA is data with initial values. BSS is data without initial data. STACK is the work area used for program execution, starting at the largest address and moving toward the lowest address in the area. TEXT and DATA are write-protected. The OS work is an operation data part that is used for the CPMS to execute macro instructions.

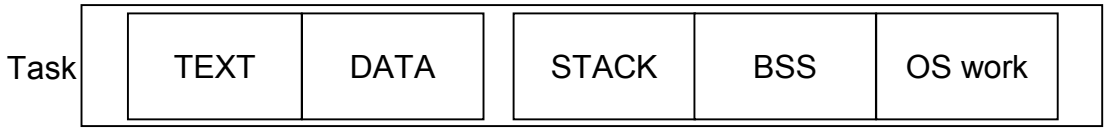


Figure 1-4 Task Structure

Multitasks that share TEXT, DATA, and BSS can be created. In this case, STACK is allocated for each member task, but BSS is shared by the members.

(3) Task types

Two types of tasks are used: user tasks created by the user and system tasks provided by the system. Tasks having the task numbers 225 to 255, reserved for the system, are system tasks. User tasks can be assigned the task numbers 1 to 224.

(4) Initial start task

The task having the task number 1 is the user initial start task (UIST). The user is responsible for creation of user tasks in such a way that they are started up by the user initial start task.

(5) Task priority level

When multiple tasks make a request for use to shared resources (CPU and memory) in the system, a task to which the right of use is to be given is determined by the processing priority associated with each task. This processing priority is called "priority level" or "level." Each level is represented by a numeric value of 0 to 31. The smaller the value, the higher the priority. The levels available for the user are 4 to 27. When registering a task, the level of the task is specified. This level is called the original level of the task. Usually, when a task is activated, this original level becomes the task operating level (current execution level).

The order of resource assignment is determined according to this current execution level.

When a task is registered, its priority level is specified.

Figure 1-5 shows the relationship for assigning levels to system and user tasks.

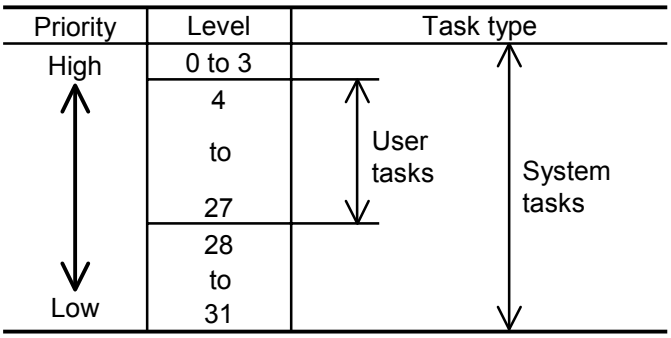
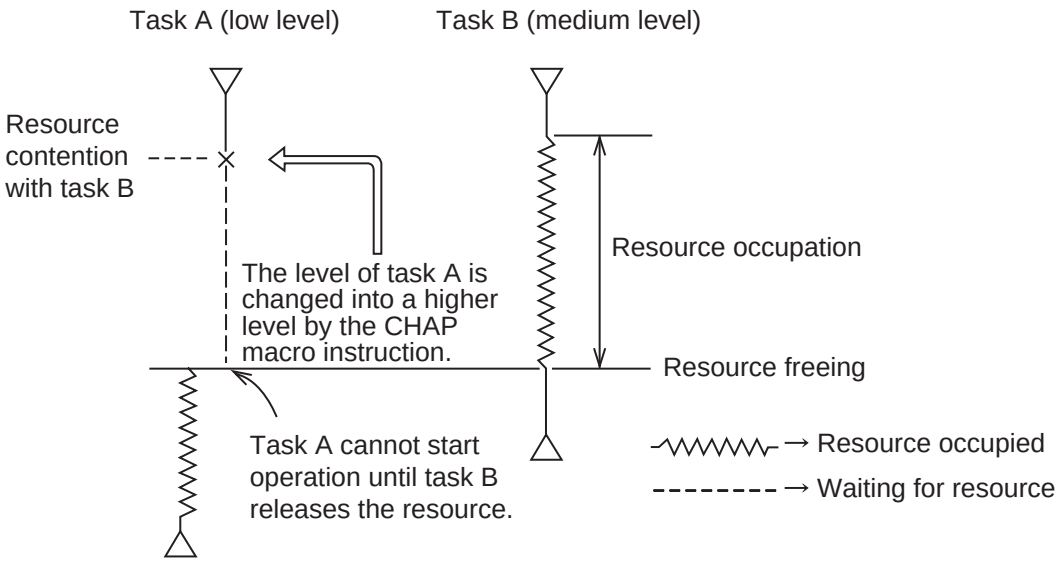


Figure 1-5 Relationship between Tasks and Levels

(6) Changing priority level

The "CHAP" macro instruction can change the level of the task being executed. The effect of the CHAP macro instruction continues from the start of the task, whose level was changed, to an end of the processing. When the task operation ends, the original level becomes the level of the task. If a level is changed by the CHAP macro instruction before a task starts its operation, the new level given by the CHAP macro instruction becomes the priority level during the operation. However, if the task operation is aborted in the period between the level change and the start of operation, the effect of the macro instruction is lost.

The CHAP macro instruction changes the priority level being a standard for resource assignment, but does not relinguish a resource already assigned to another task forcedly to a task having a changed higher priority level. This is shown in Figure 1-6.



Note: Even if the priority level of task A is raised by the CHAP macro instruction when task A and task B are in the wait state under resource contention, the resource occupied by task B is not given to task A.

Figure 1-6 Change of Priority Level and Resource

2.2 Task Scheduling

(1) Scheduling Algorithm

When multiple tasks in the system are asking for the right of use for the CPU. Only one task can always receive the right because there is only one CPU in the system. To select a task from among multiple tasks is called “dispatch.” The method of dispatching between tasks is called task scheduling.

Among different scheduling algorithms, the CPMS adopts a fixed priority scheduling method. According to this method, among the same level of tasks, the FCFS (First Come First Served) algorithm is employed.

In the FCFS, activation requesting tasks are linked to the CPU queue in the order of received activation requests. As shown in Figure 1-7, A task is linked to a CPU queue; a block of memory, called TCB (Task Control Block), is registered in a CPU queue. A TCB is always assigned to a task.

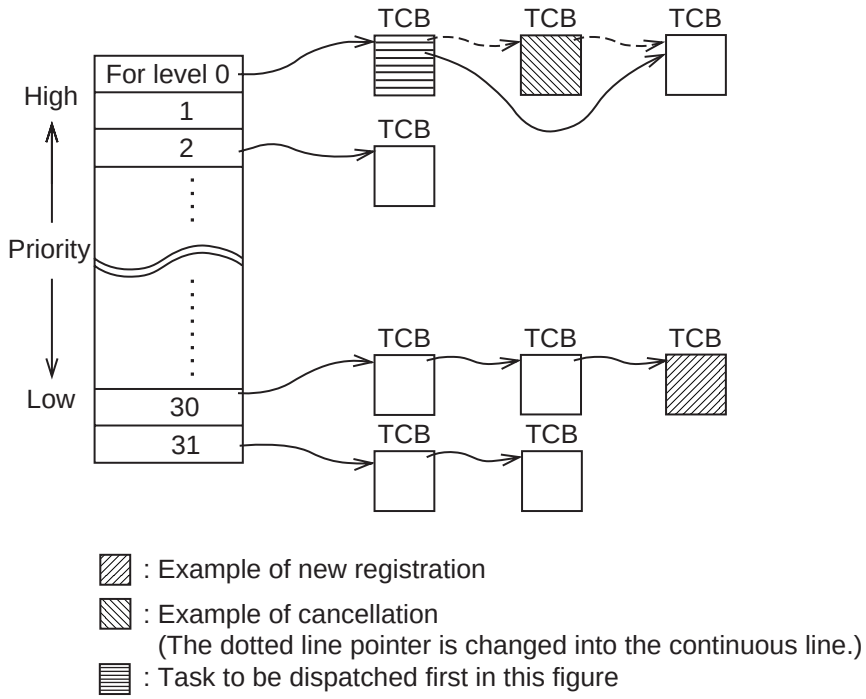
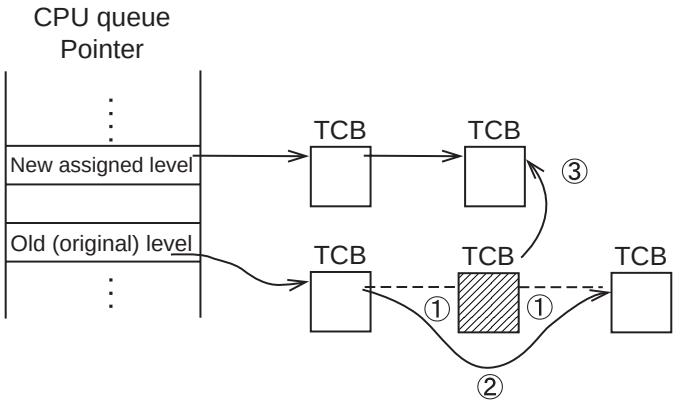


Figure 1-7 CPU Queue

In the following 3 cases, a task is released from the CPU queue:

- when the task issued the EXIT macro instruction
- when the task was aborted by the ABORT macro instruction another task issued
- when the task became abnormal (for example, when a task tries to access privileged data a protection error occurs). When a task results in an error, the task is aborted by the CPMS.

(2) CPU queue's behavior when changing priority level
Figure 1-8 shows how the TCB of a target task is managed in the CPU queue, when the CHAP macro instruction is issued from one task to another via CPMS, the former task is, hereafter, called as an issuing task and the latter task is called as a target task.



<Procedure>
① The specified TCB is released from the old level queue.
② The old level queue linkage is changed.
③ The released TCB is linked to the tail of the newly assigned level queue.

Note: When the level is changed by the CHAP macro instruction, the specified TCB is linked to the tail of the newly assigned level queue in the FCFS algorithm.

Figure 1-8 Level Change

(3) Multitasking
In task management, processing is performed so that the CPU may be usefully used. For example, when a task in process cannot proceed its processing, for some reason, the next task in the CPU queue is dispatched at once. The dispatched task starts its operation. During the execution of this task, if the cause of the suspended task is eliminated, the dispatcher dispatches the suspended task again as shown in Figure 1-9. From a broad point of view, this looks like as if two tasks were processed simultaneously although only one task is processed actually. This concurrent task processing increases the rate of CPU utilization.

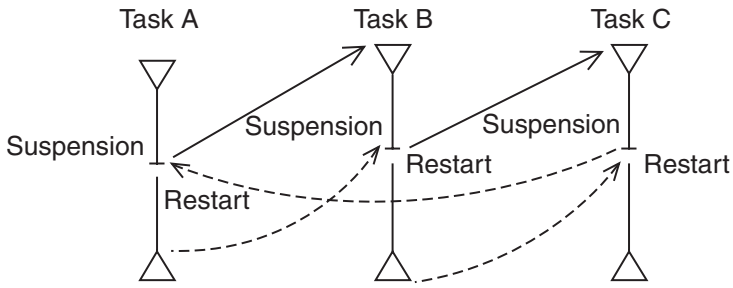


Figure 1-9 Concurrent Task Processing (Multitasking)

2.3 Task Operations

Generally, tasks have a life cycle. That is, tasks are generated, activated, executed, suspended, restarted, terminated and become extinct. However, for real-time tasks, the overhead from issuing an activation request till task execution is an important factor to determine responsiveness. Accordingly, to minimize generation and extinction of a task is necessary. Therefore, when receiving an activation request, task generation should not be performed in the real-time system. Tasks are previously generated and registered to the CPMS. That is, when activating a real-time task, the target task does not need to be generated anew and only an activation request (issuing the QUEUE macro instruction) is issued. Besides, after termination of its operation, it is not caused to be extinct.

The events that motivate task activation are shown in Table 1-2. Task execution conditions (initial activation) are shown in Table 1-3. After a task is activated, it is executed when all the conditions shown in Table 1-3 are satisfied.

An executed task continues its operation until the task cannot continue its processing by some reasons or an interrupt occurs and a task with a higher priority level must be operated. When all necessary processing is completed (termination of program execution), the task operation terminates. This is called task suspension and termination (abortion).

Table 1-4 shows factors of task suspension. A suspended task restarts its operation when the factor of suspension is eliminated and a higher-priority task or a same-priority task earlier activated cannot operate. This is called task restart.
Table 1-5 shows task restart conditions. Table 1-6 shows task termination and abortion conditions.

Table 1-2 Factors of Task Activation

EVENT		Explanation
Internal factor	Issue of QUEUE macro instruction	When the QUEUE instruction is issued from a task, the task specified in its parameter is activated.
	Lapse of certain time or at a fixed time of day	When the TIMER macro instruction is already issued, the task specified in its parameter is activated at the specified time or time of day.
External factor	Attention interrupt from I/O device	A task that is registered in built-in subroutine is activated by an attention interrupt from the I/O device.

Table 1-3 Task Execution Conditions (Initial Activation)

Condition	Explanation
All the higher-priority tasks or same-priority tasks earlier activated cannot operate.	When a higher-priority task is operable, it is executed.
The Task's main program is loaded on the main memory.	Unless the program is loaded on the main memory, it cannot operate.
The execution of the task itself is not inhibited.	When the execution is inhibited by the SUSP, ASUSP macro instruction, the task is not executed.

When all the conditions shown in Table 1-3 are satisfied, the task is executed.

Table 1-4 Task Suspension Conditions

Condition	Explanation
A higher priority task is activated.	When a high-priority task is activated by an interrupt (process interrupt or timer) and this task is operable, control is transferred to the task.
The inhibition of execution for a higher-priority task is canceled.	When a higher-priority task (whose execution has been inhibited) becomes operable, control is transferred to the task.
The execution is suspended by the task itself.	When the execution is suspended by the task itself, for example, for synchronization, control is transferred to another task.

When one of the conditions shown in Table 1-4 is satisfied, the task is suspended.

Table 1-5 Task Restart Conditions

Condition	Explanation
The inhibition of execution by another task is canceled.	The inhibition of execution by the SUSP, ASUSP macro instruction is canceled.
When the system waits for occurrence of an event, this event occurs.	An event that eliminates the factor of suspension of the task itself (DELAY or WAIT) occurs.
A higher-priority task or a same-priority task earlier activated terminates or is suspended.	As far as a higher-priority task or a same-priority task earlier activated is operable, processor service cannot be rendered to the task.

When one of the conditions shown in Table 1-5 is satisfied, the task starts its operation.

Table 1-6 Task Termination Conditions

Condition	Explanation
The EXIT macro instruction is issued.	Usually, task processing is terminated by the EXIT macro instruction.
The task becomes a target of the ABORT macro instruction.	Processing aborted by the ABORT macro instruction.
An unprocessable condition is caused by a program error.	The CPMS performs ABORT processing automatically for an error originating task.

When one of the conditions shown in Table 1-6 is satisfied, the task terminates its operation.

2.4 Task State Transition

In the CPMS system, multiple real-time tasks are organically linked and operated to perform the functions of the whole system. Therefore, individual tasks continue their operations mutually repeating activation, suspension, restart and termination, as described in Section 2.3, in close connection with one another.

Between tasks, data exchange is performed by using the GLB (Global Data Area) that is a data area common to tasks.
Between tasks, control exchange is performed by using the macro instructions prepared by task management.

The task management macro instruction controls a task operation by causing task state transition. Design system and programs with a correct understanding of how task state transition is occurred and what macro instructions cause state transition.

Table 1-7 shows the states of a task.
Figure 1-10 shows the relationship between macro instructions and task states.
Note that in the states shown in Figure 1-10, the Running state is not always an executing state of the task (including the suspended state) in a strict sense.
The state of the task produced by the macro instruction is only an example but does not represent all cases.

Table 1-7 States of Task

State	Designation	Explanation
Execution is in progress.	RUNNING	The CPU is locked for task execution.
Execution is being awaited.	RUNNABLE	The task is waiting for the CPU to be unlocked.
Execution is suspended.	SUSPENDED	Execution of the task is suspended.
An event is being awaited.	WAIT	The task is waiting for an event.
Startup is being awaited.	IDLE	The task is waiting to be executed.
Startup is suspended.	DORMANT	Startup of the task is suspended.
Non-registered	NON-EXISTENT	The task is not registered in CPMS.

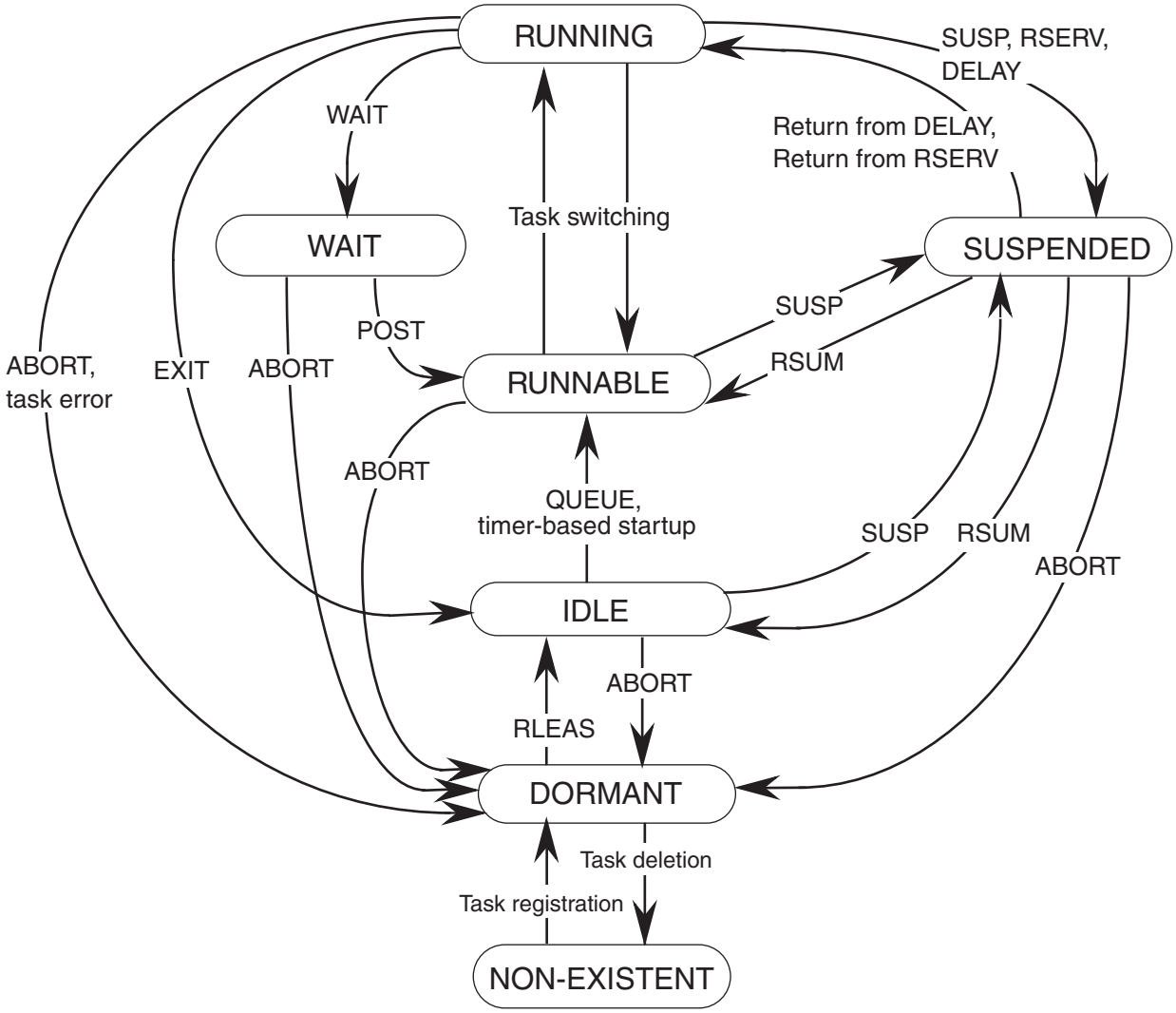


Figure 1-10 Task State Transitions

(Supplement) It a susp/rsum macro is issued to a non-gueued task that is in the IDLE state, it is executed but the state of that task remains unchanged, and the suspended state is indicated in the tc-flag field of the task control table (TCB).

2.5 Task Control

The task control method will be explained by giving examples below.

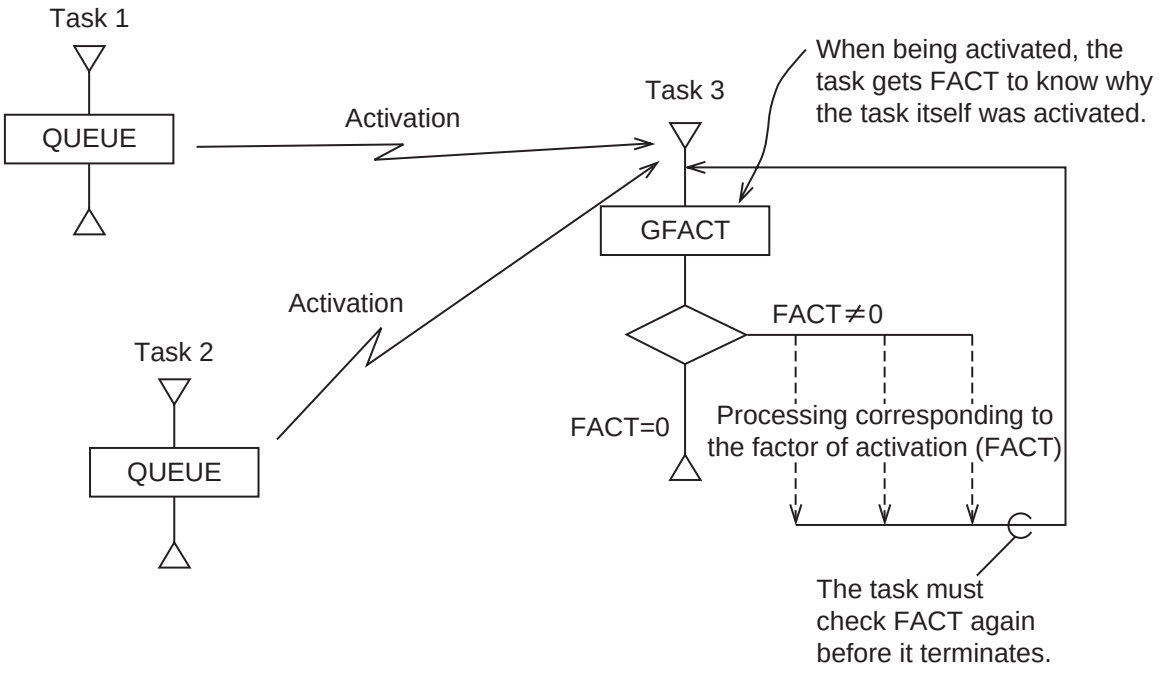
2.5.1 Initial state

When the system is started (when the power is on and the processor starts to operate), all user tasks except an initial start task are in the DORMANT state.
The initial start task is automatically activated by the CPMS when the system is started. The initial start task puts the tasks required for job execution into the IDLE state by the RLEAS macro instruction. (This is called "to release the task.") This state is ready for receiving an activation request.

2.5.2 Task activation

● QUEUE macro instruction

Tasks are activated by the QUEUE macro instruction. An activated task can get the factor of activation (FACT) by the GFACT macro instruction to know what factor activated the task itself. Figure 1-11 shows this relationship.



By the GFACT macro instruction, task 3 can know by what task (task 1 or task 2) the task 3 is activated. That is, if the FACT by which task 1 activates task 3 is specified to make a difference from the FACT by which task 2 activates task 3, it is possible to know which task activates task 3.

Figure 1-11 Task Activation

In Figure 1-11, the GFACT macro instruction reads out factors of activations one after another. For example, supposing that 4 factors of activation (integers of 1 to 32) are set as “1”, “5”, “10” and “11” the GFACT macro instruction reads them out in sequence from the younger number.

At the first issue of the GFACT macro instruction, FACT = 1 is read out. At the second issue of the GFACT macro instruction, FACT = 5 is read out. The read out FACT is cleared by the GFACT macro instruction. Accordingly, in the above example, FACT = 1 is not read out again even if the GFACT macro instruction is issued after FACT = 1 is once read out.

Such a FACT can also be set by the SFACT macro instruction. Figure 1-12 shows this example.

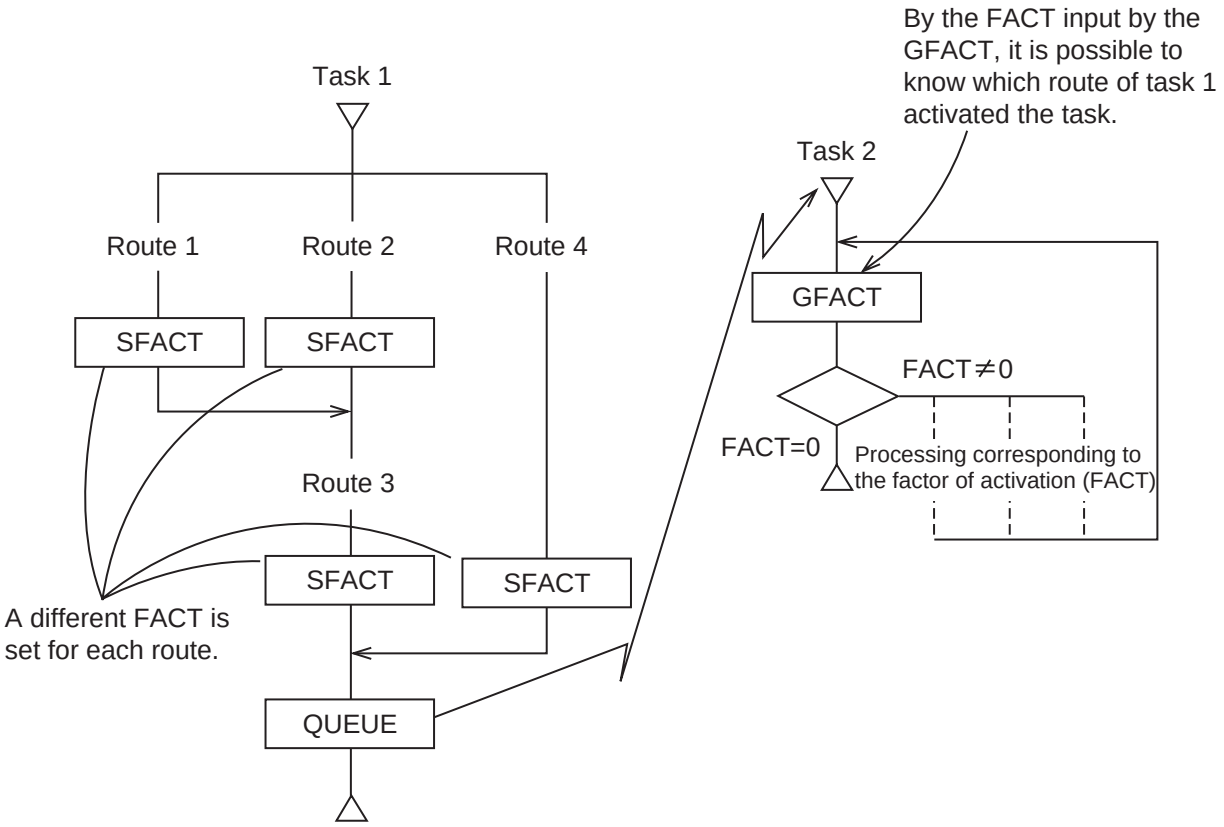
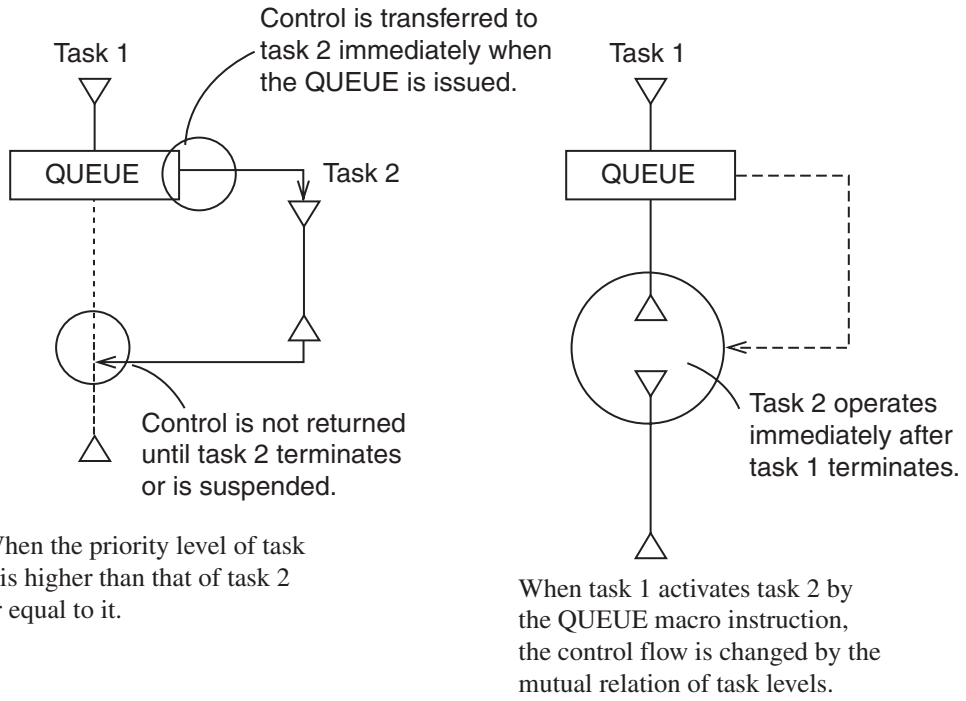


Figure 1-12 SFACT Macro Instruction

In the CPMS, the “first come first served” rule is applied for among same-level tasks for high-efficiency real-time control. Accordingly, the task execution order flow is changed by the mutual relation of task activation levels as shown in Figure 1-13.



When task 1 activates task 2 by the QUEUE macro instruction, the control flow is changed by the mutual relation of task levels.

Figure 1-13 QUEUE Macro Instruction and Task Execution Order

- TIMER macro instruction

  As seen in Figure 1-13, task activation by the QUEUE macro instruction is usually performed at once. In some cases, however, a task must be activated after the lapse of certain time or at certain time of day. In these cases, use the TIMER macro. This macro instruction can activate a task at the time of day or after the lapse of certain time as specified in a parameter. At this time, the factor of activation (FACT) is transferred to the activated task like the FACT using the QUEUE macro instruction.

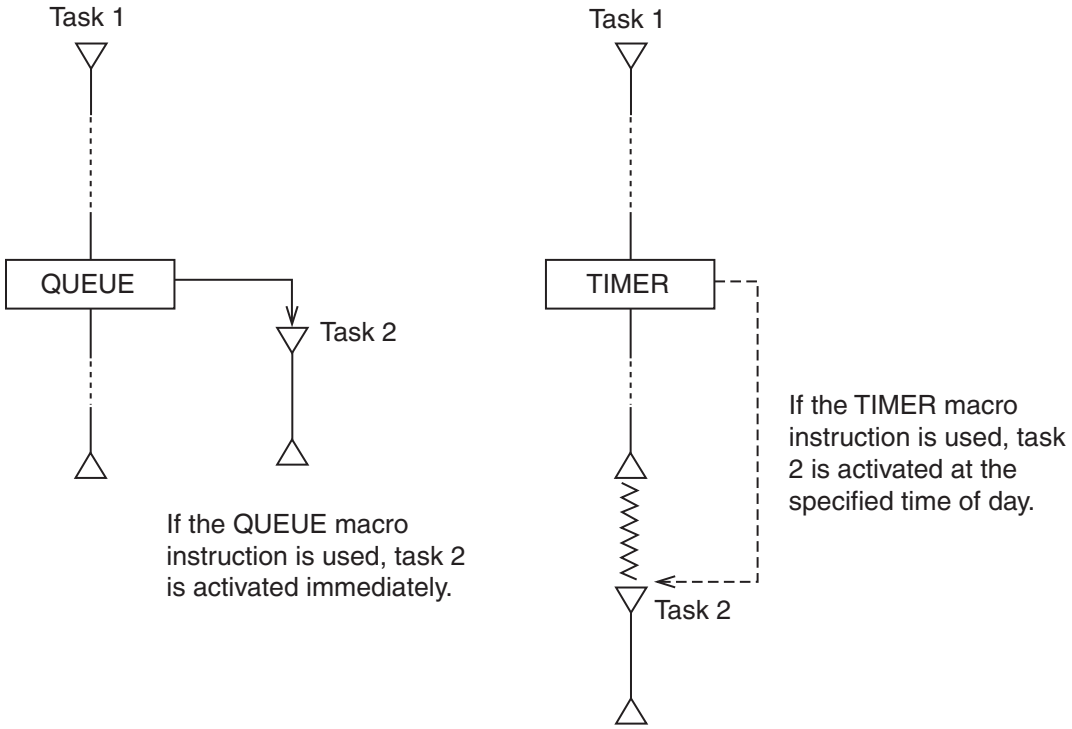


Figure 1-14 Difference in Task Activation between QUEUE Macro Instruction and TIMER Macro Instruction

2.5.3 Task termination

A task terminates processing by issuing the EXIT macro instruction. In the CPMS, tasks are allowed to issue the EXIT macro instruction when returning from the main routine.

2.5.4 Task execution inhibition

● DELAY macro instruction

The TIMER macro instruction is used to activate another task after the lapse of certain time. To activate the task itself, this can be attained by the DELAY macro instruction.

The TIMER macro instruction can also be issued to the task itself. If the DELAY macro instruction is used, the environment (e.g., BSS, STACK value) provided when the DELAY macro instruction was issued can be kept when control is returned to the task itself after the lapse of the time specified in a parameter. When the TIMER macro instruction is used, operation is started from the beginning of the task but the environment is not kept. For this reason, to start the operation again after the suspension of certain time, the DELAY macro instruction is used.

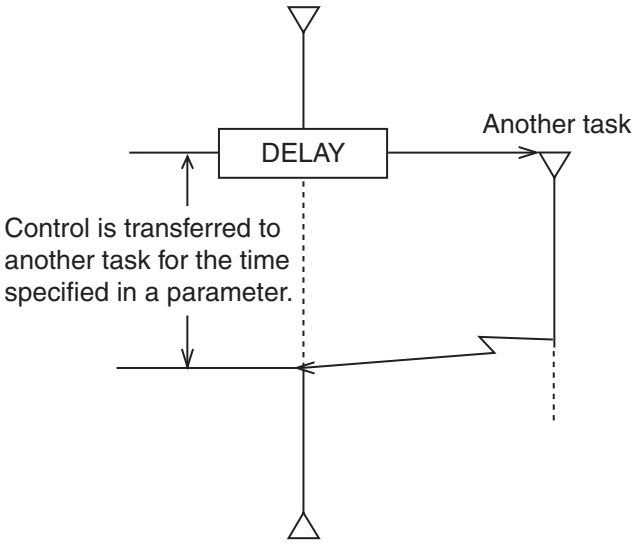


Figure 1-15 DELAY Macro Instruction

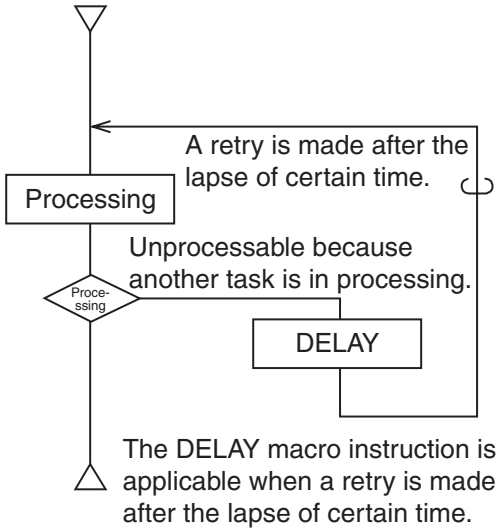
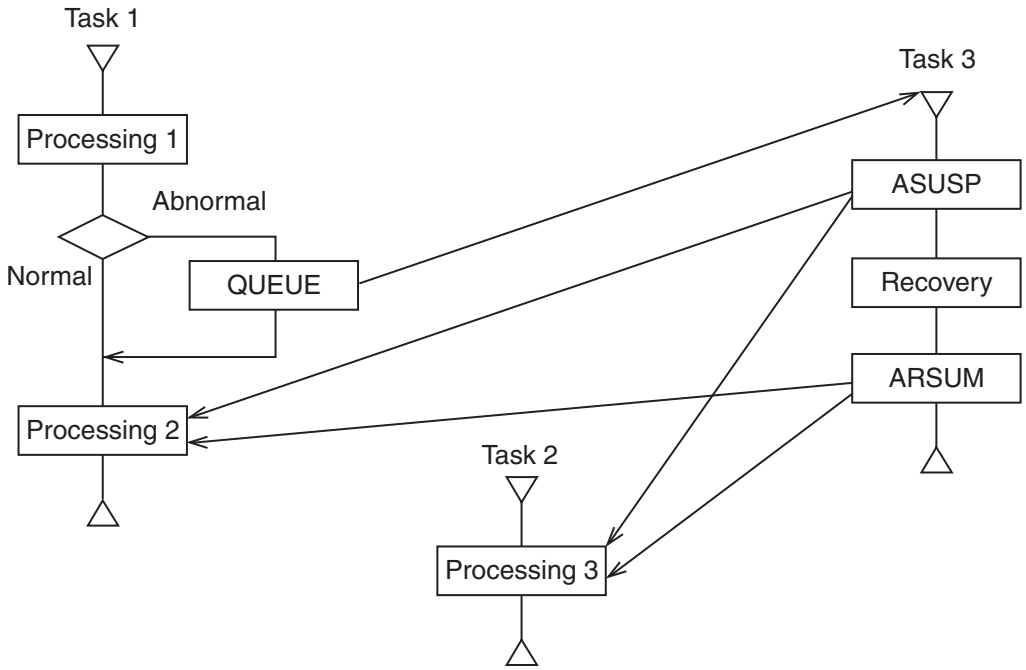


Figure 1-16 Application of DELAY Macro Instruction

- ASUSP macro instruction
  To inhibit the execution of all other tasks including higher-priority tasks, the ASUSP macro instruction is used.

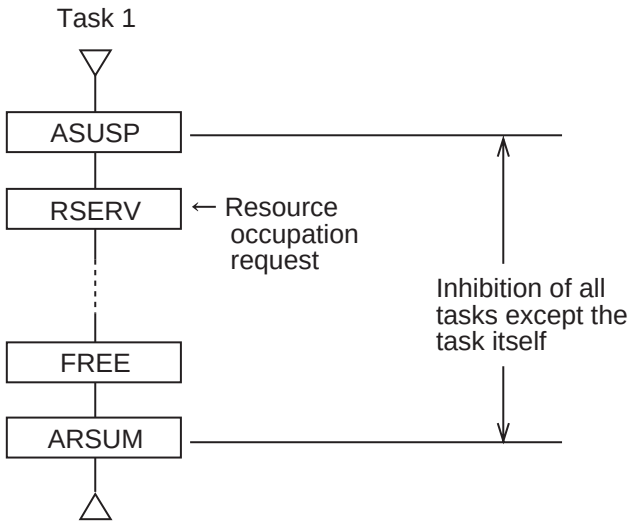
  The tasks whose execution was inhibited by the ASUSP macro instruction is freed from the inhibited state by the ARSUM macro instruction. However, these instructions are used to inhibit the execution of other tasks. If the use of them is not limited, a deadlock may be caused.

  To avoid such a deadlock, any processing that requires a system resource must not be performed between the issue of the ASUSP macro instruction and the issue of the ARSUM macro instruction.



Task 3 which controls processings 1, 2 and 3 inhibit the execution of tasks 1 and 2 by the ASUSP macro instruction when processing 1 is abnormal. Task 3 makes processing 2 and 3 suspended by the ASUSP macro instruction until the recovery processing ends. After inhibition, recovery processing is performed so that processing 2 and 3 may be performed normally, and then the inhibition of task 1 and 2 execution is canceled by the ARSUM macro instruction.

Figure 1-17 Inhibition of Execution by ASUSP Macro Instruction



When a resource occupation request is issued while the execution of other tasks is inhibited, a deadlock is caused if the inhibited tasks occupy the resource.

Figure 1-18 Example of Deadlock by ASUSP Macro Instruction

2.5.5 Task abortion

- ABORT macro instruction

  To abort task execution and put the task into the execution inhibit state, the ABORT macro instruction is used.

  The ABORT macro instruction aborts a task in execution (or in the wait state), forcedly frees the resource occupied by the task, and put the task into the DORMANT state.

2.5.6 Synchronization between tasks

For synchronization between tasks (to perform another task processing after termination of a task processing), the WAIT macro instruction and POST macro instruction are available. This synchronization is controlled by an "event." To synchronize with another task, a task informs an area called ECB (Event Control Block) that it waits for an event occurrence, and enters the WAIT state. This ECB is defined for each event.

A task that notifies occurrence of an event references the ECB and checks who waits for occurrence of the event, then informs the waiting task of an event occurrence and releases it from the WAIT state. See Figure 1-19. This processing is performed by the WAIT macro instruction and POST macro instruction, respectively.

One ECB is assigned to one event. The same ECB must not be shared with multiple events and multiple tasks must not share the same ECB. Through the ECB, detailed event information can be exchanged between tasks. This is called POST code.

In the WAIT/POST macro instruction, there is no limitation on the relation of issuing order. This is shown in Figure 1-20.

To prevent a deadlock, the effect of the ASUSP macro instruction is lost if the WAIT macro instruction is issued after the ASUSP macro instruction is issued.
Figure 1-21 shows ECB state transition.

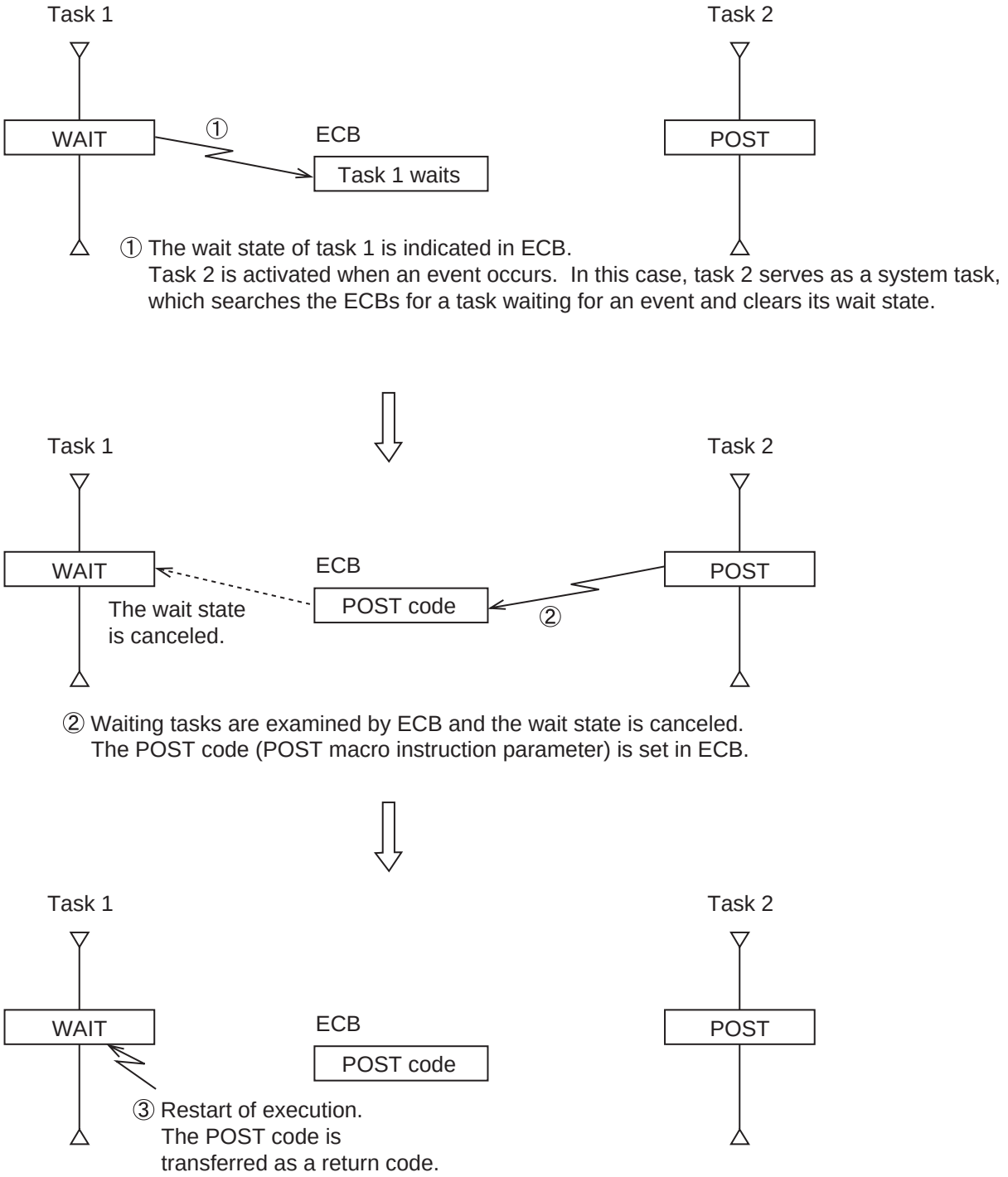
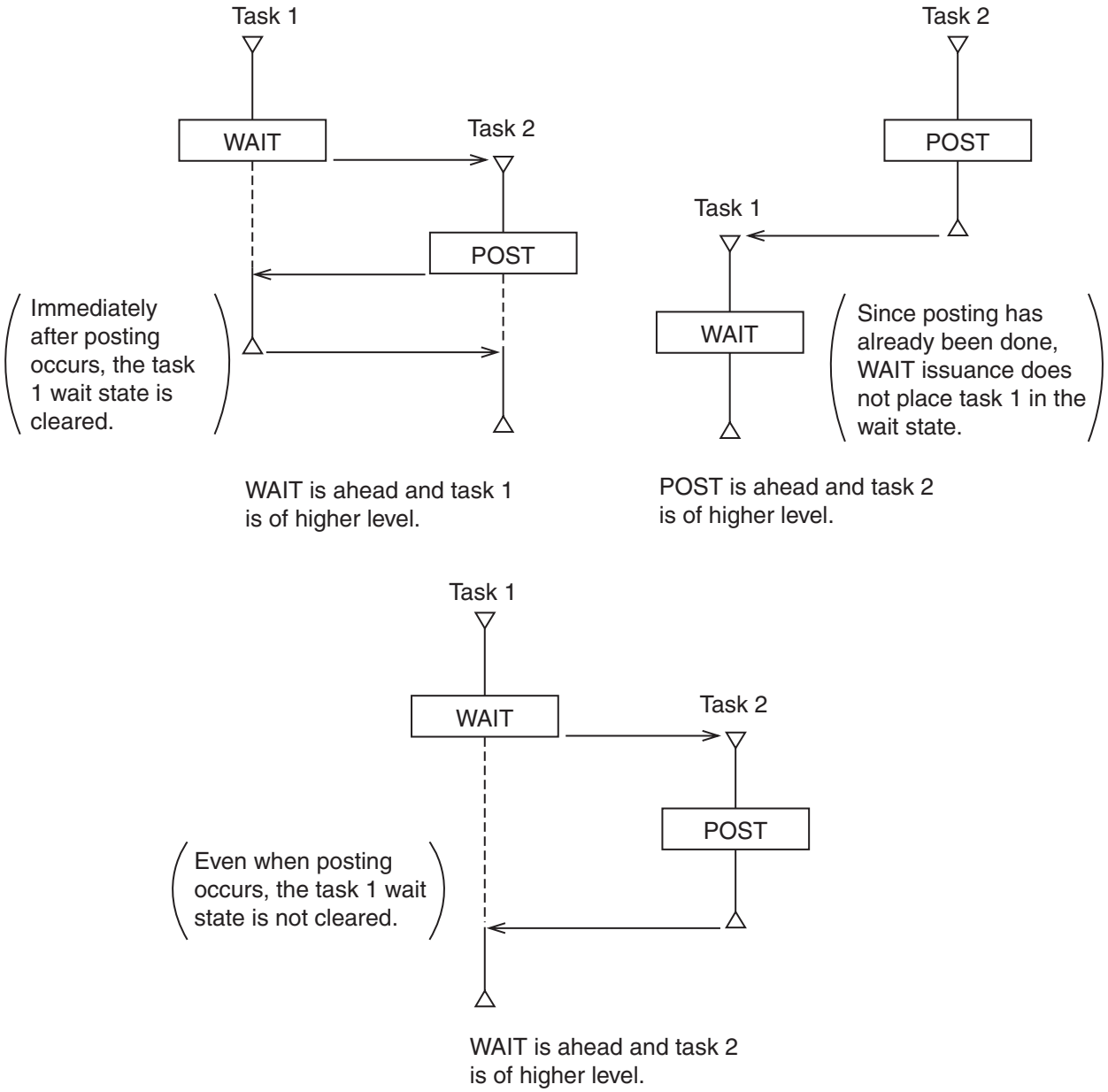
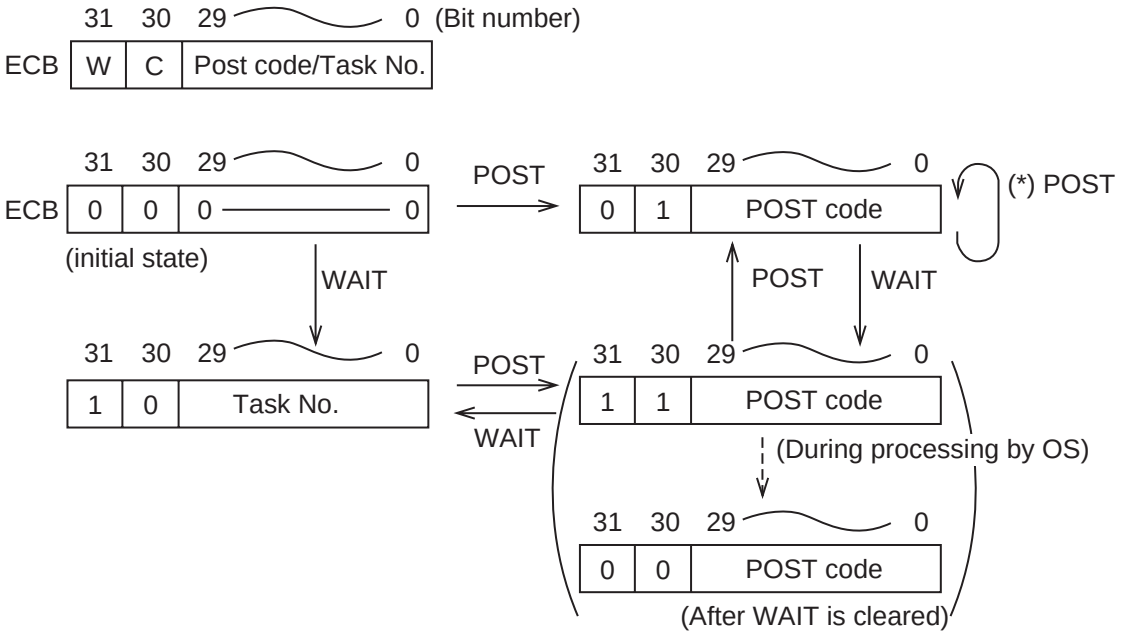


Figure 1-19 Synchronization between Tasks by WAIT/POST



The control flow between tasks varies depending on difference of levels between tasks and issuing order of WAIT and POST macros. The dotted line indicates that task execution is inhibited (in the WAIT state).

Figure 1-20 Control Flow Using WAIT/POST



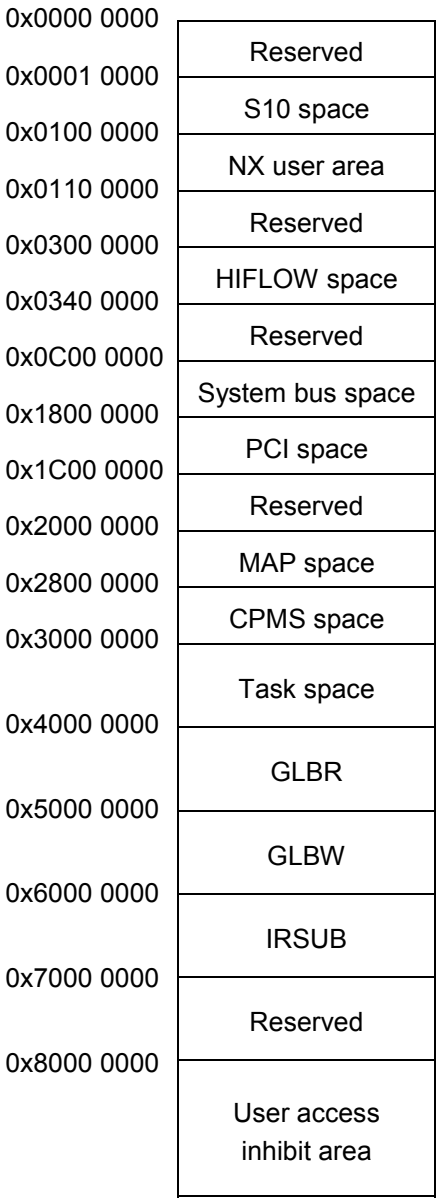
ECB bit numbers 31 and 30 are called W (Wait) bit and C (Complete) bit respectively. The POST code marked (*) is overwritten.

Figure 1-21 ECB State Transition

CHAPTER 3 MEMORY MANAGEMENT

3.1 Logical Space

CPMS runs all tasks in a single logical space. CPMS also manages conversion between logical addresses and physical addresses.



Reserved: Reserved for the CPMS.
S10 space: The LPU memory is allocated.
NX user area: The buffer area to be used by the NX is allocated.
HIFLOW space: The HIFLOW program is allocated.
System bus space: The high-speed bus I/O and memory are allocated.
PCI bus memory space: The PCI bus I/O and memory are allocated. The built-in LANCE uses this space.
MAP space: The tasks and IRSUB to be used by the CPMS and the built-in subroutine control table are arranged.
CPMS space: Space exclusively used by the CPMS.
Task space: The TEXT, DATA, BSS, STACK, and OS works of tasks are allocated.
GLBR: The shared memory (read only) among tasks in the PU is allocated.
GLBW: Shared memory (read and write allowed) among tasks in the PU is allocated.
IRSUB: The shared indirect link subprogram among tasks is allocated.
User access inhibit area: The area after 0x80000000 cannot be accessed by task. Access will result in a program error.

Figure 1-22 Logical Address Map

3.2 Memory Protection

CPMS manages memory write-protection for pages each consisting of 4 KB. Table 1-8 lists the memory access rights.

Tasks can write to the following memory spaces. The other spaces are write-protected.

- BSS and STACK for the local task (When a multitask is used, BSS is shared.)
- Area in the GLBW and CM where the logical space and the physical memory are mapped.
- PI/O space and cyclically transferred memory in system bus memory space.

The CPMS wrtmem macro allows user programming tasks to rewrite programs and protected data. This macro can be used to write to write-protected main memory.

Table 1-8 Memory Access Rights

Space / Accessed by / Accessed mode	CPMS System	Task User	Remarks
Task space (In the user space)			
Text for the local task	R-X	R-X	
Data for the local task	R-X	R-X	
Stack for the local task	RWX	RWX	
BSS for the local task	RWX	RWX	
Text for other tasks	R-X	R-X	
Data for other tasks	R-X	R-X	
Stack for other tasks	R-X	R-X	
BSS for other tasks	R-X	R-X	RWX when a multitask is used
User space (except the task space)			
NX user area	RWX	RWX	
HIFLOW space	RWX	RWX	
GLBW	RWX	RWX	
GLBR	R-X	R-X	
IRSUB	R-X	R-X	
MAP	R-X	R-X	
System bus space (for user)	RWX	RWX	
System bus space (for system)	R-X	R-X	Memory for the OS subsystem (driver)
CPMS space	R-X	R-X	
PCI space (for user)	RWX	RWX	
PCI space (for system)	R-X	R-X	
LPU space	RWX	RWX	
Kernel space			
Space with V = R specified in main memory	RWX	−−−	Text and data of CPMS are included.
I/O register space	RWX	−−−	Only the kernel and driver can access it.
KROM	R-X	R-X	

R: Readable, W: Writable, X: Executable,

−−− : Inaccessible (When a task executes this access, it will be aborted.)

3.3 Error Handling during Memory Access

● Memory error
When a multi-bit error of the memory with the ECC feature occurs, the system stops.

● Memory single-bit error
A single-bit error in memory provided with the ECC feature is corrected, and data is read correctly. Therefore, a single-bit error is not handled as an error. When a single-bit error is encountered during memory patrol, the data is rewritten and the error is corrected. If the single-bit error persists, it is handled as a solid failure and an alarm report is logged as an error.

● System bus access error
Unless the system bus connection I/O is mounted, mapping is not performed to the system bus memory space. An attempt to access a non-mapped address results in a program error. However, even if the address has been mapped, an attempt to access the address in the event of a hardware failure may result in a system bus error. Such an error is called a target abort error. A target abort error is not handled as a program error. System behavior against a target abort error is described below.
- If the target abort error is detected during a read access, all-1s data is read.
- If the target abort error is detected during a write access, the program continues running as if data was written.
- In response to the target abort error, an interrupt is generated to the PU, resulting in a module failure.

● Write-protect error
A write may occur at a write-protected address due to a software failure. This results in a programming error, aborting the task.

3.4 Procedure for Checking Access to the System Bus

The cyclically transferred memory of the system bus connection I/O is directly accessed from user programs as the bus memory. To allow for errors during access to bus memory, the procedure given below is required.

Have the user program issue a CHKBMEM macro to check whether the bus memory in the specified slot is accessible to the user program. The CHKBMEM macro returns information that indicates whether (1) bus memory is mounted in the specified slot, and (2) the hardware is not accessible due to a target abort error. When an error is detected by the CHKBMEM macro, do not allow the user program to access the bus memory in the slot.

Upon completion of access to bus memory, make sure that a CHKTAER macro has been issued to check that the hardware is functioning normally. This is because even if the hardware is faulty, the task continues processing without being aborted as if the operation required had been done normally.

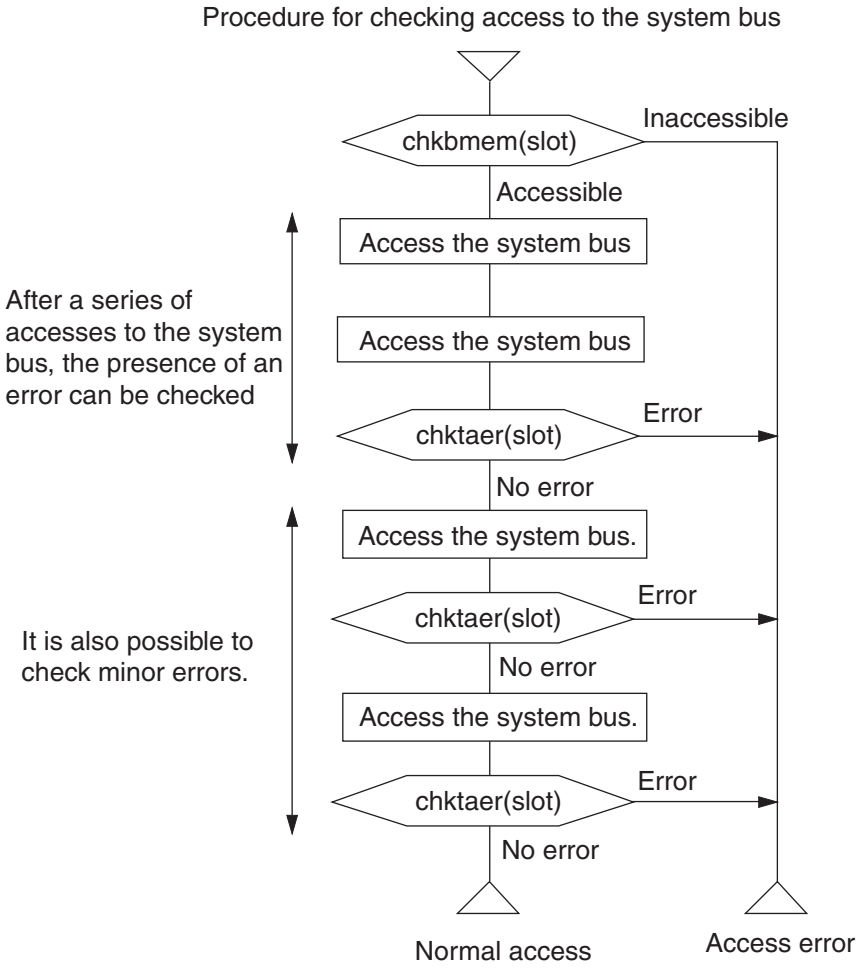


Figure 1-23 Procedure for Checking Access to the System Bus

CHAPTER 4 TIMER MANAGEMENT

4.1 Length of Time and Time of Day

CPMS manages the length of time and the time of day. The time is represented as Gregorian year, month, day, hour, minute, and second. Years are valid within the range of 1970 to 2069. The length of time is in milliseconds.

Tasks can use the GTIME macro to get the time of day under management by CPMS. The STIME macro can also be used to change the time managed by CPMS.

The CMU does not includes a real-time clock (RTC) that is backed up by battery in case of a power failure. However, the LPU includes an RTC. When CPMS is started up, it reads the year, month, day, hours, minutes and seconds from the RTC and sets that time as the point from which the time of day starts to be measured. During operation, CPMS manages the length of time and the time of day by using the internal timer based on clocks supplied to the processor. Since the RTC and internal timer operate based on different clocks, an error may be caused after a long time elapsed. To allow for this, CPMS sets the time of day based on the internal timer in the RTC once per day to eliminate any error.

4.2 Time-Based Task Control

Tasks can use the DELAY macro to suspend task execution for a specified duration. The TIMER macro can be used to create a timer that starts up a task at a specified time or after the elapse of a specified length of time and then restart the task at fixed intervals. The timer can be deleted with the CTIME macro, if necessary. The timer created by the TIMER macro accepts a time of day only within 24 hours after the TIMER macro is issued, as the initial value.

4.3 Changing the Time

If the time is changed by the STMIE macro, this has an effect on the operation of the timer in which a task start is set by the time start in the TIMER macro. If the scheduled time is jumped over by advancing the time, the first scheduled start time may elapse and the start timing may be lost. In this case, the timer is started when it is changed. In the case of cyclic time specification, the scheduled start time is shifted so that the time resulting from adding the cyclic time to the first scheduled start time may be behind the changed time. The timer that started a task at the scheduled time does not perform re-registration for a start at the scheduled time even if the time is set backward.
For the time-specified timer, the start time is not changed even if the time is changed.

4.4 Matching the Times between the CMU and LPU

The LPU mounts an RTC but the CMU does not mount any RTC. Accordingly, the CMU time is set to the current time of the LPU according to the following timing.

- Issue of the STIME macro by CMU
- CMU time matching (00 hour, 00 minute, 30 seconds)

Note that when the current time is set in the basic system, the CMU time becomes discontinuous.

CHAPTER 5 SHARED RESOURCE MANAGEMENT

5.1 Shared Resources

As resources that are shared among tasks, the main memory, CPU, I/O and data area (GLB) can be mentioned. Out of these resources, the main memory, CPU and I/O are exclusively controlled on the system side. However, the GLB must be exclusively controlled on the user side.

Figure 1-24 shows the necessity of this exclusive control. Figure 1-25 shows the prevention against a fault due to resource contention by exclusive control.

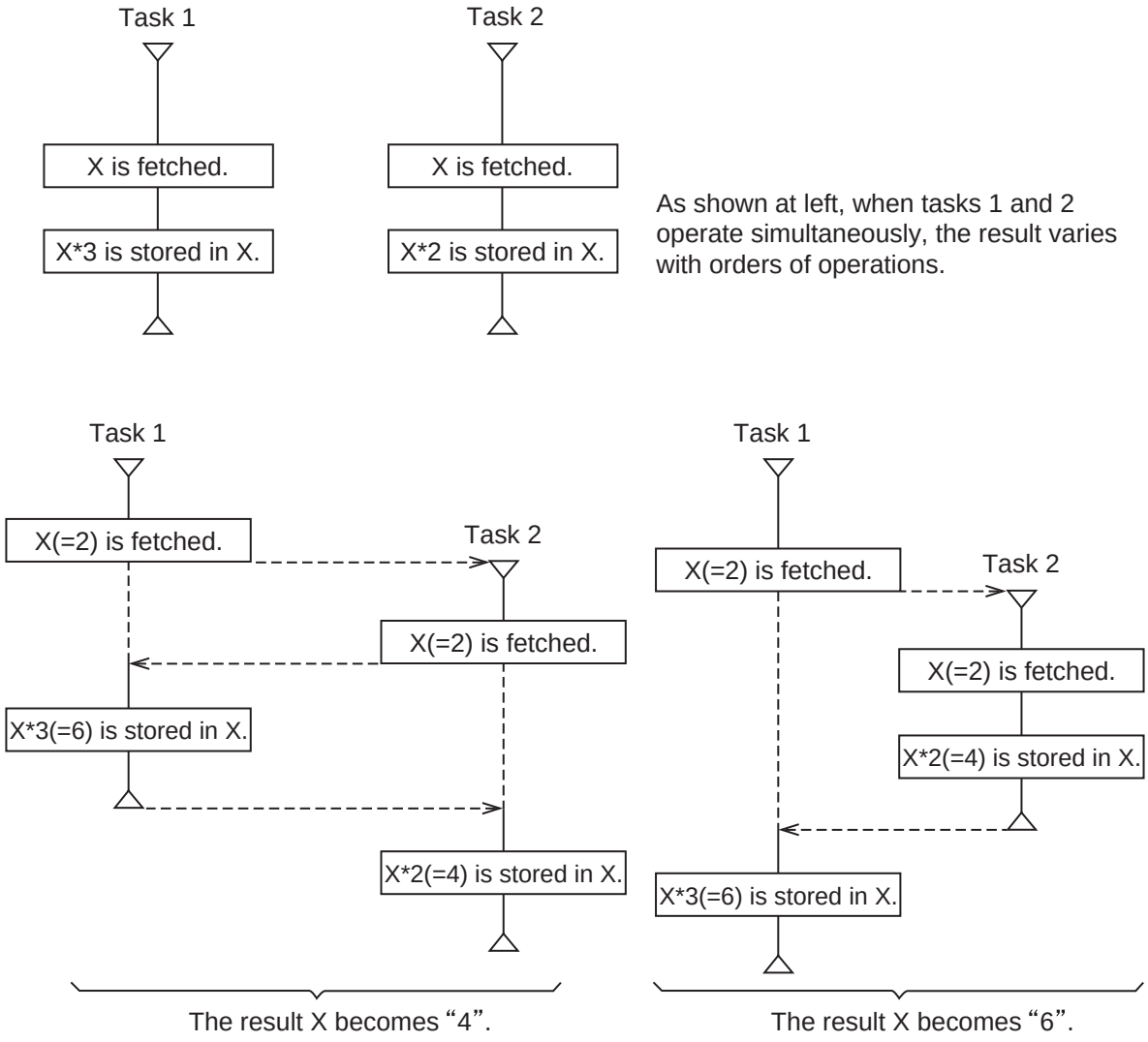
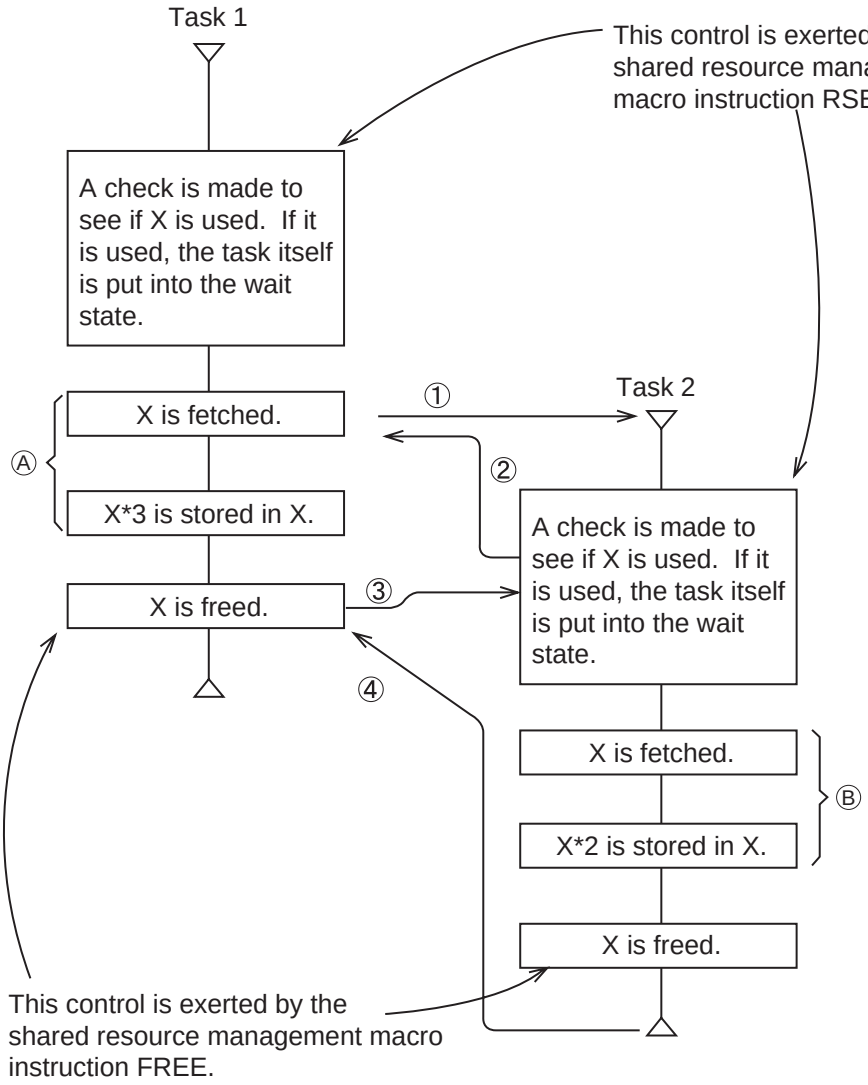


Figure 1-24 Fault Occurring When No Exclusive Control is Exerted



Control flows in the order of ①, ②, ③ and ④ to prevent both Ⓐ and Ⓑ from being operated at the same time.

Figure 1-25 Exclusive Control by Shared Resource Management Macro Instructions

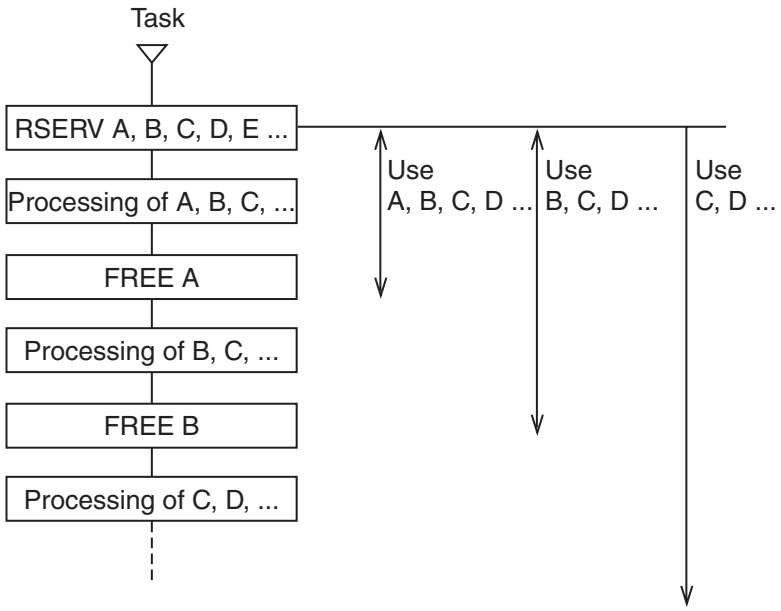
5.2 Shared Resource Management Method

Regarding the GLB being a shared resource among tasks, the physical resource itself can be occupied. That is, each time the GLB address and size are registered in the system table that manages shared resources and an occupation request is sent by the RSERV macro instruction, this system table is referenced and a check is made to see if the target GLB is already occupied or not. If the target GLB is occupied, the requesting task is put into the SUSPENDED state by the RSERV macro instruction until the resource is freed. The SUSPENDED state of this task is released when the requesting resource is freed and becomes usable.

When multiple tasks waits for freedom of a resource, this resource will be assigned to the highest-level task among them. However, this principle is not applicable when the highest-level task cannot operate for another reason.

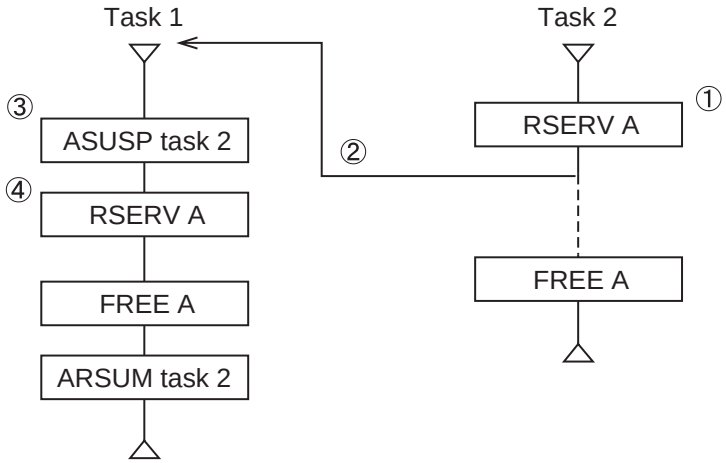
As a rule, all the resources required by a task are supposed to be occupied at a finite time to avoid a deadlock. Therefore, the RSERV macro does not permit multiple issue, that means a task already occupying a requesting resource cannot issue this RSERV instruction again. As shown in Figure 1-26, A task occupies all necessary resources at the beginning and starts processing. When these resources are selected, and performed their roles, they are freed by the FREE macro instruction at once.

Figure 1-27 shows an example of deadlock. As shown in this example, the RSERV macro instruction must not be issued after such a macro instruction that inhibits execution of another task as the SUSP macro instruction.



- All the resources to be requested by a task are occupied at a time.
  After they are used, they are freed in sequence by the FREE macro instruction.
- A single FREE macro instruction can free multiple resources at a time.

Figure 1-26 Usage of RSERV/FREE



① After task 2 occupies resource A
② Control is transferred from task 2 to task 1 before task 2 frees resources A.
③ Task 1 performs SUSP processing for task 2. (Consequently, task 2 becomes inoperable and cannot free resource A.)
④ Task 1 attempts to occupy resource A but this resource has already occupied by task 2. Task 1 has to be put into the wait state and cannot perform RSUM processing for task 2.
⑤ With this happening, both task 1 and task 2 become unexecutable.

Figure 1-27 Example of Deadlock

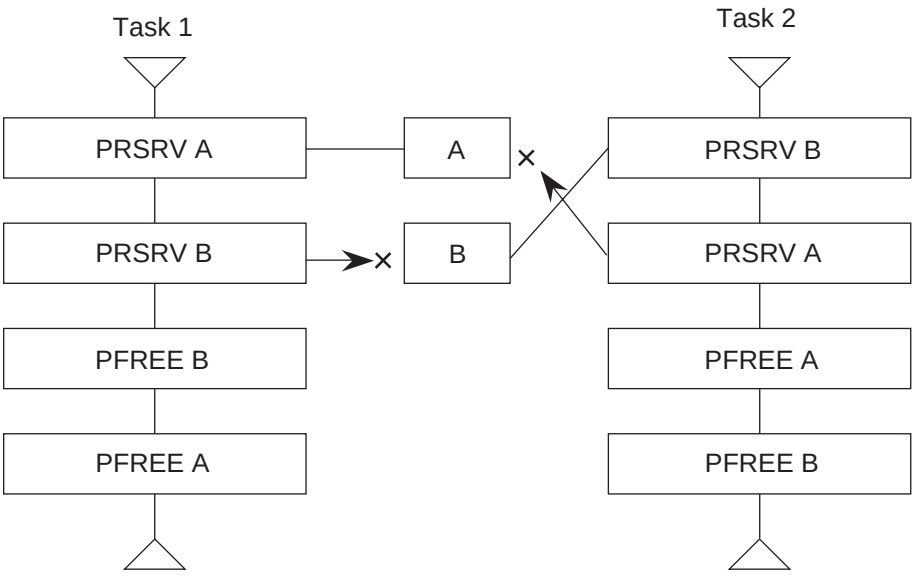
5.3 Exclusive Control of Shared Resources by the PRSRV and PFREE Macros

The PRSRV and PFREE macros can be used for exclusive control of shared resources between tasks more precisely than using the RSERV and FREE macros.

To start locking: Issue a PRSRV macro in which SAREA in GLB and the locking range are specified.

To terminate locking: Issue a PFREE macro in which SAREA in GLB and the locking range are specified.

When the specified GLB area cannot be locked, control is not returned to the task that issued a PRSRV macro until the shared resource is freed. Tasks can issue the PRSRV macro as many times as necessary. Therefore, a task can gradually share multiple resources using more than one PRSRV macro rather than getting them at one time. This would reduce the number of waits in locking resources. However, the order in which shared resources are locked should be clarified to prevent a deadlock.



In the example above, task 1 is locking resource A and task 2 is locking resource B. Task 1 waits for resource B to be freed, while task 2 waits for resource A to be freed. Each task is waiting for the other task to free the locked resource, resulting in a deadlock. A solution to this problem is to lock the same resource in the same order.

Figure 1-28 Sample Deadlock Caused by PRSRV

CHAPTER 6 I/O DEVICE MANAGEMENT

6.1 Structure of the I/O Device Management Feature

CPMS provides subsystems (I/O drivers) responsible for controlling I/O devices with the basic functions of I/O Device Management. Perform input/output operations using the interfaces of individual subsystems.

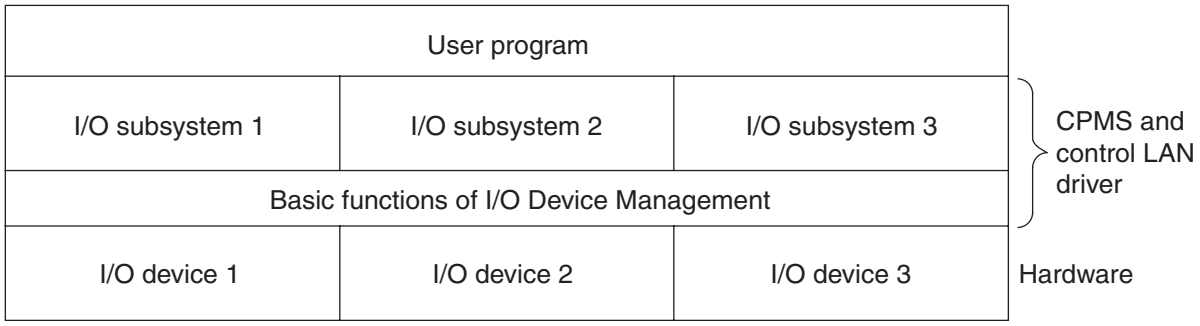


Figure 1-29 Structure of the I/O Device Management Feature

6.2 I/O Unit Number

The CPMS identifies the system bus connection I/O as the I/O target (device) by unit number (abbreviated as UNO). A number resulting from adding 4 to the connected slot number is allocated to each unit number.

6.3 Device Number

The device number identifies a logical device and the driver that controls it.

The logical device is used to define the purposes of a device. More than one logical device can be defined for one single device.

The device number consists of a major number and minor number. The major number identifies the subsystem that controls the device. The minor number specifies the location where the device is connected and its purposes. The device-dependent field is defined for each subsystem.

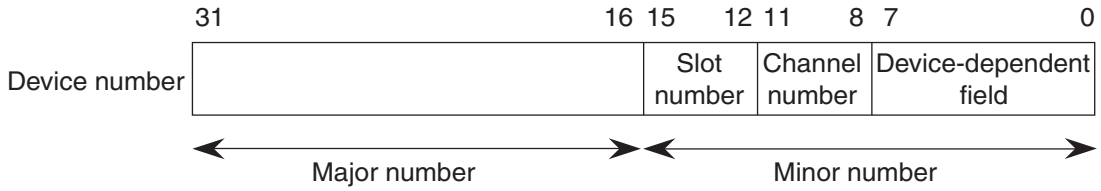


Figure 1-30 Device Number

CHAPTER 7 SYSTEM MANAGEMENT

7.1 Starting Up and Stopping CPMS

7.1.1 Status changes at startup and stop

Figure 1-31 shows how the status changes when CPMS is started up and stopped. Table 1-9 explains each state, while Table 1-10 explains events.

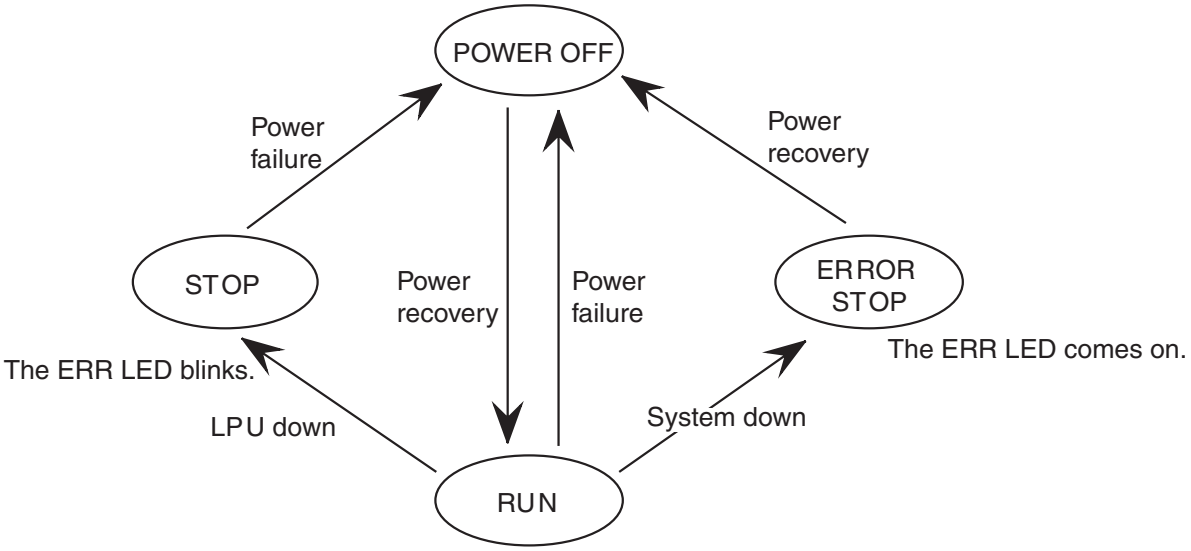


Figure 1-31 Status Changes when CPMS is Started Up and Stopped

Table 1-9 States on Startup and Stop

State	Explanation
POWER OFF	The power supply is OFF.
STOP	The LPU has gone down.
ERROR STOP	The system program is stopped due to a system error.
RUN	The system program is running.

Table 1-10 Events on Startup and Stop

Event	Explanation
Power recovery	Power is turned on.
Power failure	Power is turned off.
System down	The system program stops due to an error.

7.1.2 Startup

The first hardware status is a power OFF (memory erased) status. When a power recovery is made from this status, the CPMS is started into a RUN status. If data is downloaded into the ROM card, the programs and data on the program storage memory are copied according to this timing.

7.1.3 Stop

Turning off the power supply stops the system.

7.2 INS Built-in Subroutine and Initial Start Tasks

As the final processing during the start of the operating system, CPMS carries out the procedure below.

(1) Link to the INS built-in subroutine.
(2) Start up the system initial start task (SIST having task number 255).
(3) Start up the user initial start task (UIST having task number 1).

CPMS passes the numbers corresponding to the start factors listed in Table 1-11, as parameters in the INS built-in subroutine and the start factors of the initial start tasks.

Table 1-11 Start Factors

Number	Start factor	Explanation
1	IPL start	This factor the initial start task always started up.

7.3 Watchdog Timer

7.3.1 Functions

CPMS uses a watchdog timer to monitor tasks to prevent them from going into an infinite loop. The watchdog timer can detect the fact that a task took too much execution time and could not serve for plant control. When a WDT timeout occurs, a link is provided to the built-in subroutine WDTES. The user can register error handling programs in WDTES. The user can also force to stop the CPU using the value returned by WDTES. When the watchdog timer expires with no error handling programs registered in WDTES, CPMS does not abort the task, nor does it stop the CPU itself.

7.3.2 How to use the watchdog timer

When using the watchdog timer, make sure that one of the tasks for which execution time is monitored issues WDT control macros (WDTSETs) at an interval shorter than the time set in the watchdog timer. When the execution time of the macro-issuing task or tasks having higher priorities than that macro extends beyond the expected value, the setting of the watchdog timer is no longer updated by the task. As a result, the watchdog timer causes a time-out.

On the startup of the initial start task, the watchdog timer is not yet started. It starts when the user program issues the first WDTSET macro. The watchdog timer set to 0 by the WDTSET macro stops without a time-out.

CPMS uses only one watchdog timer, so only one task at a time is subjected to execution time monitoring based on the watchdog timer. Set a task that monitors the tasks involved in plant control, and monitor the monitoring task with the watchdog timer.

CHAPTER 8 TASK ERROR HANDLING

The basic concept of error handling during task execution is as follows:

- When a task error is encountered, the execution of the task is canceled. A method of continuing task execution is available when the task returns to its recovery point. (See "8.5 Recovery from Program Errors.")
- In the case of a hardware error that does not affect task execution, the task continues its execution. The task can get hardware error information to perform error handling.
- A built-in subroutine handles task errors. The task number of the erroneous task is returned with an input parameter of the built-in subroutine.

8.1 Repertory of Built-in Subroutines

CPMS imposes some rules on built-in subroutines so that the user can create part of system processing.

A total of four entries are allowed for each built-in subroutine: two for middleware and the operating system and two for the user.

Entry numbers 1 and 2 are assigned for middleware and the operating system, while the entry numbers 3 and 4 are for the user. Entry are linked in the ascending order of entry numbers like 1 → 2 → 3 → 4.

Table 1-12 Repertory of Built-in Subroutines

Built-in subroutine name	When to link	Input information	Output information	Macro issuing	Number of entries
CPES	A programming error occurs.	PRGEB	Available	Possible	4
IES	An I/O error occurs.	IOERB	Available	Possible	4
EAS	An error is logged.	ADB	Available	Possible	4
INS	Before start of IST	Start factor	None	Not possible	4
EXS	A task exits.	Task number	None	Possible	4
ABS	A task aborts.	Task number	None	Possible	4
PCKS	A macro parameter error occurs.	SVCEB	Available	Possible	4
MODES	A module error occurs.	HARDEB	Available	Possible	4
WDTES	The watchdog timer expires.	None	Available	Possible	4
ADTS	An ADT exception occurs.	Break information	None	Possible	4

8.2 Execution Environment of Built-in Subroutines

CPMS executes built-in subroutines in a system mode in which interrupts are inhibited. The execution priorities of all built-in subroutines are higher than those of any tasks. The following restrictions are imposed on the execution environment of built-in subroutines:

- The user must estimate that the size of the stack area used by built-in subroutines is only up to 1 KB. When the stack area overflows, the CPU stops.
- Built-in subroutines are intended for event logging, access to GLB, and start and stop of other tasks. Do not perform processing by which built-in subroutines being executed are made to wait or are stopped.
- To limit the time during which interrupts are inhibited, make sure that a built-in subroutine is executed within one millisecond.
- Only the RLEAS, QUEUE, and ABORT macros can be called by built-in subroutines.
- Floating-point arithmetic operations cannot be used in built-in subroutines. Such an attempt stops the CPU.
- A programming error in a built-in subroutine stops the CPU.

8.3 Processing to Link Built-in Subroutines

Figure 1-32 shows how the INS, ABS, EXS, CPES, PCKS, and WDTES built-in subroutines are linked to EAS.

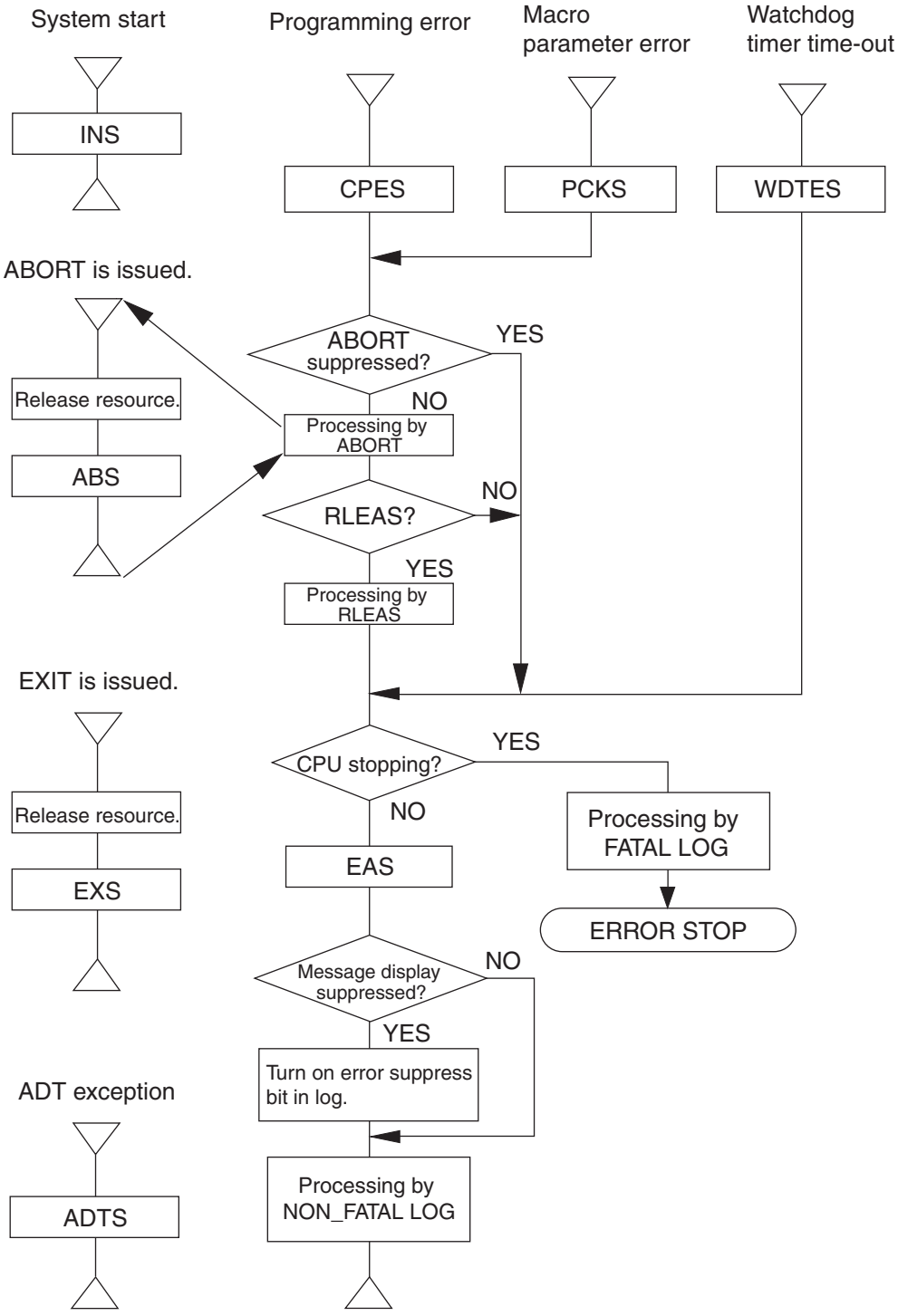


Figure 1-32 Processing to Link Built-in Subprograms (1)

Figure 1-33 shows how the IES, PIOS, and MODES built-in subroutines are linked to EAS.

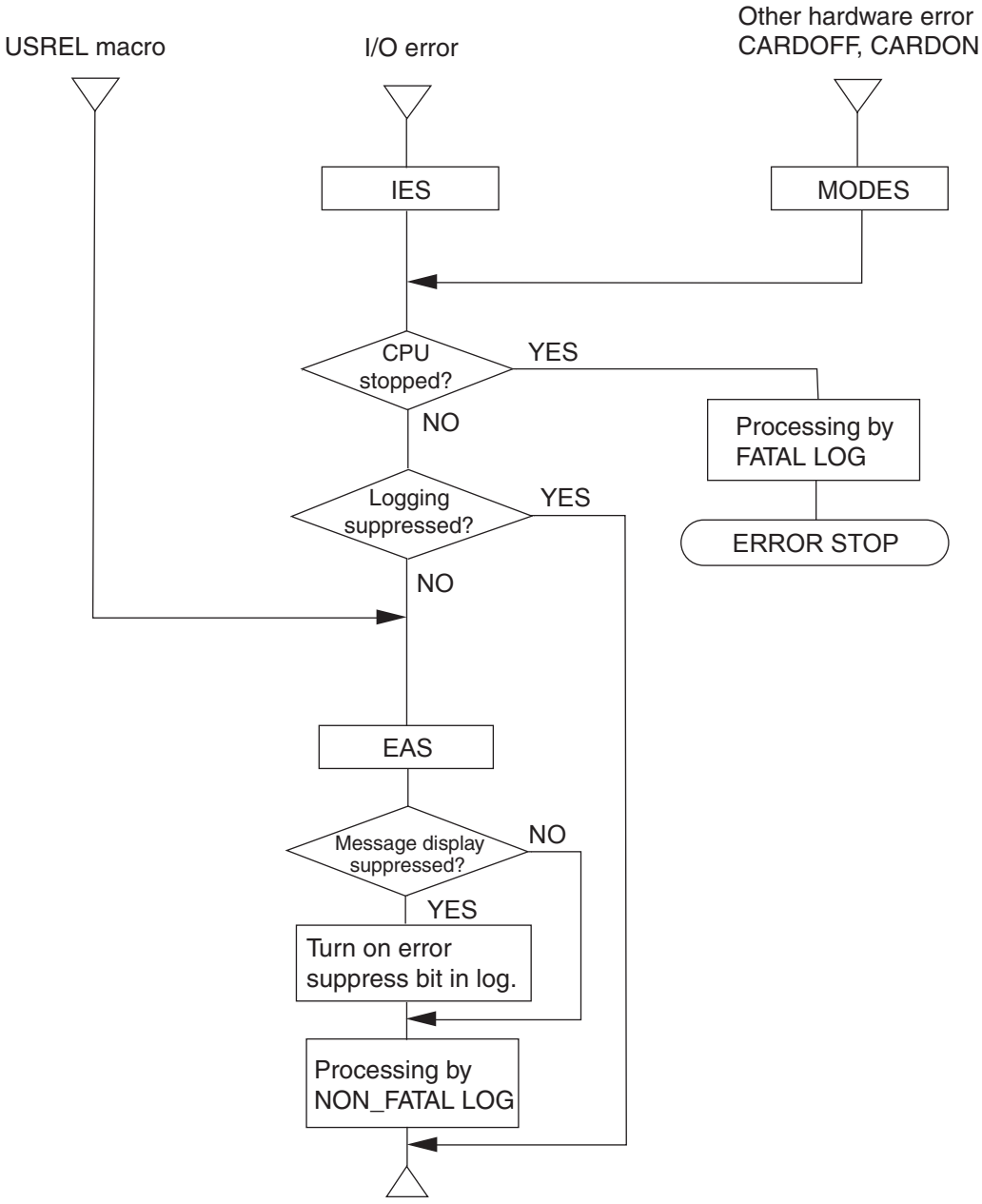


Figure 1-33 Processing to Link Built-in Subprograms (2)

8.4 Linkage of Built-in Subprograms

#include<cpms_ulsub.h>

- CPES - CPU Error Subroutine
  int cpes(prgeb)
  struct PRGEB *prgeb; /* Program Error Block */

- IES - I/O Error Subroutine
  int ies(ioerb)
  struct IOERB *ioerb; /* Program Error Block */

- EAS - Error Alert Subroutine
  int eas(adb)
  struct ADB *adb; /* Alert Data Block */

- INS - Initial Start Subroutine
  int ins(reset)
  long reset; /* System start factor */ See start factors in Table 1-11.

- EXS - Exit Subroutine
  int exs(tn)
  long tn; /* Task Number */

- ABS - Abort Subroutine
  int abs(tn)
  long tn; /* Task Number */

- PCKS - Parameter Check Subroutine
  int pcks(svceb)
  struct SVCEB *svceb; /* SVC Error Block */
  long piono, isw, idp;

- MODES - Module Error Subroutine
  int modes(hardeb)
  struct HARDEB *hardeb;

- WDTES - WDT Error Subroutine
  int wdtes( )

- ADTS - ADT Subroutine
  int adts(adtdb)
  struct ADTDB *adtdb;

NOTE 1: When CPES or PCKS is called, the task is aborted by default. To suppress task abort, turn on the ULSUB_OUT_ABORTSUPRES bit in output information.
NOTE 2: WDTES is called when the system watchdog timer expires. WDTES is not intended for monitoring termination of each task.
NOTE 3: In the case of a system task error, no link to either EXS or ABS occurs.

For input information, see “APPENDIX C BUILT-IN SUBROUTINE INPUT DATA.”
Output information (return values)

All output information items from built-in subroutines are in the common format. Their meanings depend on bits. Since there is more than one entry point, output information from each set-up data is ORed.

```
#define ULSUB_OUT_LOGSUPRES     0x00000010    /* Error logging is suppressed. */
#define ULSUB_OUT_MSGSUPRES     0x00000020    /* Error message display is suppressed. */
#define ULSUB_OUT_RLEAS         0x00000040    /* The task is released. */
#define ULSUB_OUT_ABORTSUPRES   0x00000080    /* Task abort is suppressed. */
#define ULSUB_OUT_CPUDOWN       0x00000100    /* The CPU goes down. */
```

These bits may or may not have an effect depending on the built-in subroutine type, as show in Table 1-13.

Table 1-13 Output Information from Built-in Subroutines

	CPES	IES	EAS	INS	EXS	ABS	PCKS	MODES	WDTES	ADTS
Availability of output information	Yes	Yes	Yes	No	No	No	Yes	Yes	Yes	No
ULSUB_OUT_ABORTSUPRES	√	iv	iv	iv	iv	iv	√	iv	iv	iv
ULSUB_OUT_RLEAS	√	iv	iv	iv	iv	iv	√	iv	iv	iv
ULSUB_OUT_LOGSUPRES	iv	√	iv	iv	iv	iv	iv	√	iv	iv
ULSUB_OUT_MSGSUPRES	iv	iv	√	iv	iv	iv	iv	iv	iv	iv
ULSUB_OUT_CPUDOWN	√	√	iv	iv	iv	iv	√	√	√	iv

√: Valid
iv: Invalid

(1) ULSUB_OUT_ABORTSUPRES
This bit suppresses the task specified by an argument from being aborted. The bit has an effect with the CPES and PCKS built-in subroutines. When the ULSUB_OUT_CPUDOWN bit is On but the ULSUB_OUT_ABORTSUPRES bit is Off, the task is aborted and then the CPU is stopped.

(2) ULSUB_OUT_RLEAS
This bit releases the task specified by an argument. The bit has an effect with the CPES and PCKS built-in subroutines. To abort or release a task, turn on this bit but turn off the ULSUB_OUT_ABORTSUPRES bit.

(3) ULSUB_OUT_LOGSUPRES
This bit has an effect with the MODES and IES built-in subroutines. When a module error or I/O access has been normally processed, turn on this bit. The On/Off condition of this bit is judged by the CPMS or I/O driver. When the bit is On, error logging is skipped.

(4) ULSUB_OUT_MSGSUPRES
This bit has an effect only with the EAS built-in subroutine. When the bit is On, the message suppress flag in error information is set to 1. Actual processing to suppress message display is entrusted to the display program. Make sure that the display program performs processing according to the message suppress flag in the error information. This bit does not affect error logging.

(5) ULSUB_OUT_CPUDOWN
This bit stops the CPU when an error occurs.

8.5 Recovery from Program Errors

To allow for program errors caused by a task, set a recovery point in advance. Returning to the recovery point enables the task to continue execution. The recovery point is effective for program errors caused in the routine including a recovery point or the subroutines called from that routine.

- Call save_env to save the execution environment data at the recovery point into GLB.
- Call resume_env from the CPES built-in subroutine to return control to the recovery point after CPES is executed.

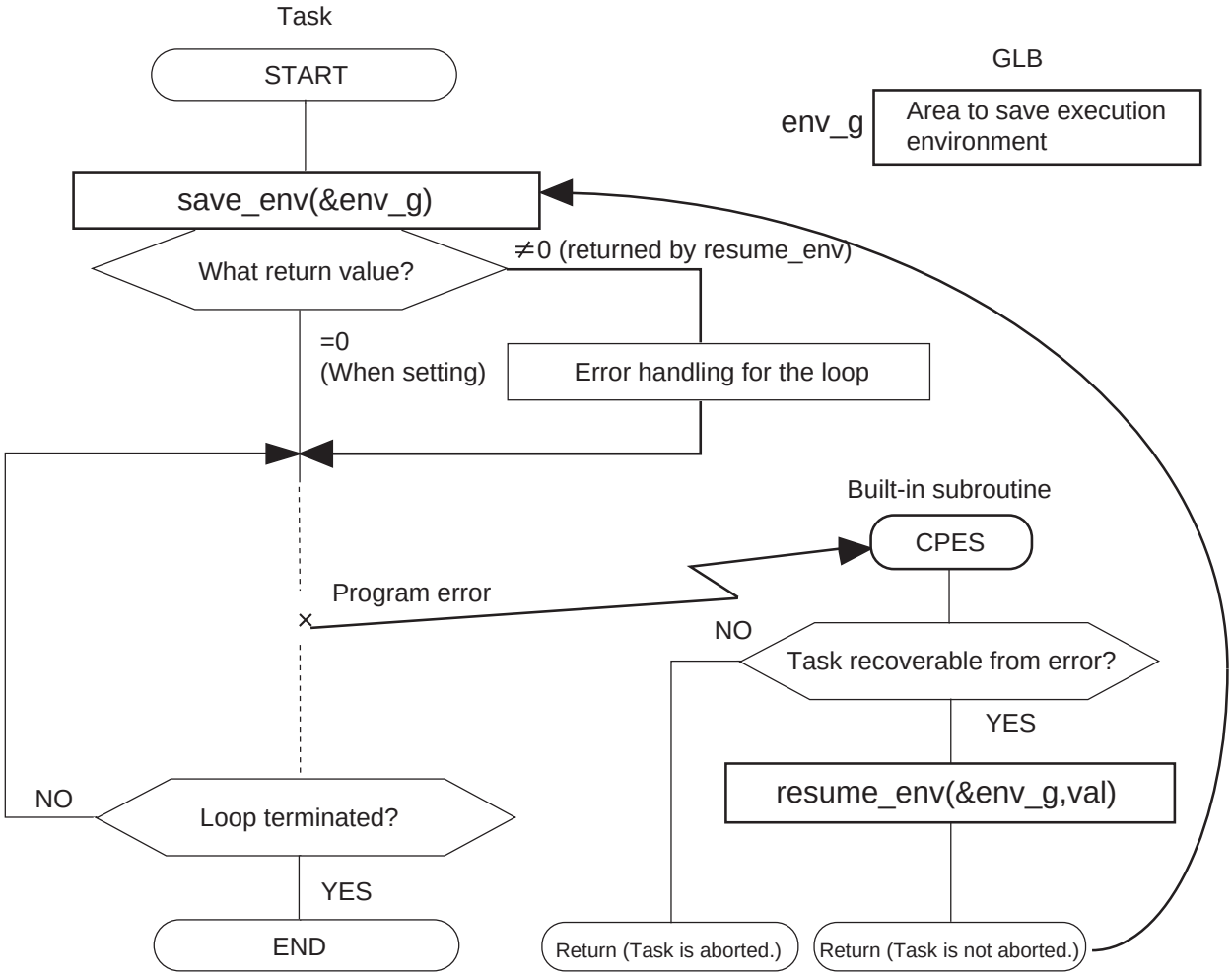


Figure 1-34 Recovery from Program Errors

(1) How to use

- Call save_env(&env_g) to save the execution environment at the error recovery point into env_g allocated in GLB. save_env will return 0.

- Make sure the task that caused the error is linked to the CPES built-in subroutine.

- When the task that caused an error is recoverable from the error, the task should not be aborted. To do this, set a user built-in subroutine registered in the CPES built-in subroutine in such a way that resume_env(&env_g, val) resumes the erroneous task from the error recovery point. Make sure that val is set to a non-0 value.

- When the task returns from the built-in subroutine without being aborted, the task resumes from the recovery point. At the same time, save_env returns val.

- From the code returned from save_env, the task can learn control has been returned from the CPES built-in subroutine, enabling the task to perform postprocessing for the error.

(2) Notes

- It is assumed that when an error occurs, the contents of the stack remain the same as when the recovery point was set. The task cannot recover from an error caused by a stack crash or program crash.

- When the task is recovered from an error by resume_env, external variables, static variables, or auto variables are not recovered. Since, for example, the number of loops in the figure above remains unchanged, the loop in error can be determined. Conversely, when these variables are corrupted due to an error, processing cannot be continued correctly even after the task returns to the error recovery point.

- When the CPES built-in subroutine attempts recovery from an unrecoverable error, an infinite loop may occur, repeating programming errors and recovery processing. Limit the programming errors from which the task in error should recover, and make sure that CPES checks whether the task is a recoverable task.

- Be sure to specify, in resume_env, the area (env_g) where the execution environment is saved by the task in error with save_env. Otherwise, the task cannot return to the recovery point correctly.

CHAPTER 9 SYSTEM SERVICES

9.1 DHP

Each time CPMS passes a predetermined processing point, it records that fact in a buffer in main memory. This record is called the debugging helper (DHP). The DHP buffer is allocated in the kernel work area. CPMS supports DHP processing.

(1) Recording point

DHPs are taken at the following points:

- In general, a DHP is taken after all CPMS macros are issued and the parameters are read. Upon completion of macro processing, the processing result (for example, start factor returned to the user with GFACT) may be recorded as a DHP.
- Before and after task switch processing
- Processing to start up or stop a task
- Processing to start up I/O or issue a termination interrupt
- Task error handling
- Error handling for the operating system or hardware
- The USRDHP macro can also be used to record user information.

(2) Data to be recorded

At each DHP point, the following data is recorded:

- Code representing a DHP point (4 bytes)
- DHP recording time (4 bytes)
- Task number and task priority (2 bytes each)
- Data necessary for analysis (variable length between 0 and 20 bytes)

(3) Recording mode

By default, recording is always working. The RPDP svdhp command can be used to stop or restart DHP processing.

(4) DHP buffer

The DHP buffer uses 128 KB in main memory.

(5) Output of the record

- The RPDP svdhp command can be used to read the current DHP data.
- Errors are logged together with the most recent DHP data.

9.2 CPU Load Ratio

The user can measure the CPU load ratio and the CPU execution time for each task.

(1) CPU load ratio

The SYS_IDLE function of the getsysinfo macro can be used to get the accumulated CPU idle time. The idle time is continually accumulated. Obtain the idle time by calculating the difference between the accumulated idle time when the previous SYS_IDLE is issued and the current accumulated idle time. The accumulated idle time overflows when it reaches 32 bits long. Take care when obtaining the idle time. If the current value is greater than the previous value, simply calculate the difference between the two. If the current value is smaller than the previous value, add the twos complement (one greater than the inverted value) of the previous value to the current value.
Make sure that the measurement time is 24 hours or less.

The CPU load ratio is calculated as follows:

$$\text{CPU load ratio} = \frac{\text{Measurement time} - \text{Difference between current and previous accumulated idle times}}{\text{Measurement time}}$$

THIS PAGE INTENTIONALLY LEFT BLANK.

PART 2 MACRO SPECIFICATIONS

CHAPTER 1 OVERVIEW

1.1 Macro Instructions

Macro instructions are used for a user program (task) to request the CPMS for processing. In the user program, macro instructions are described as subroutine calls. These subroutines are automatically expanded into trap instructions being CPMS calling instructions by the macro linkage library.
When a subroutine is run, the subroutine links to the CPMS by this trap instruction, so that requesting processing is executed.

1.2 CPMS Macro Linkage Library

The CPMS macro linkage library is a subroutine to expand a macro instruction described in the user program into a trap instruction when a CPMS macro instruction is used.
When the CPMS macro linkage library is called, it stores parameters into the user stack and/or registers in the order specified for each macro instruction, and issues a trap instruction.

This is shown in Figure 2-1.

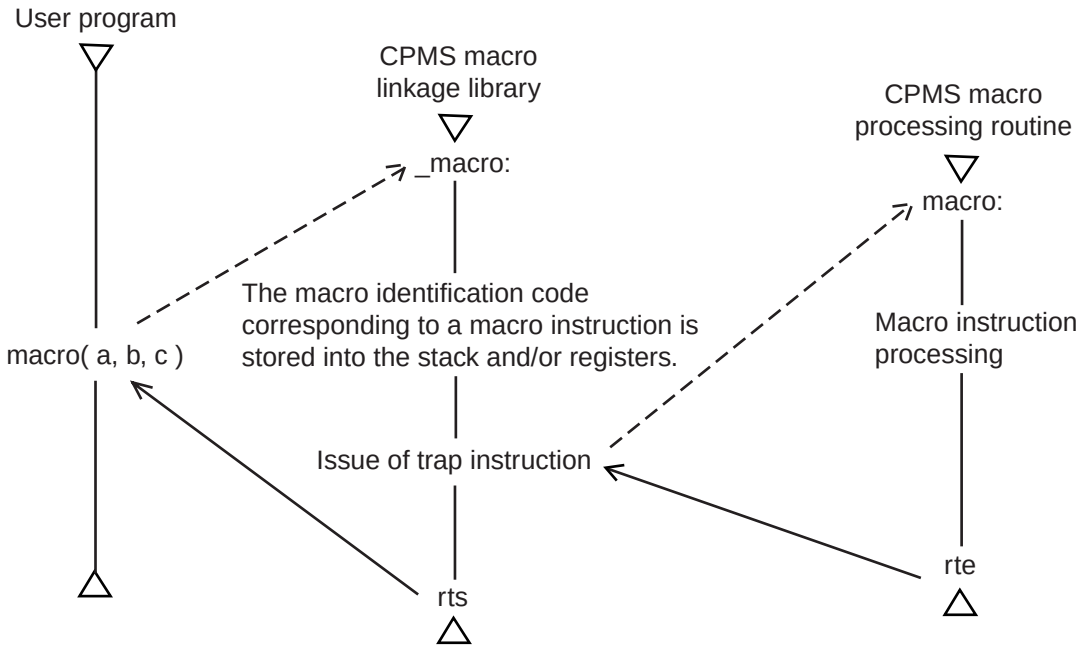


Figure 2-1 CPMS Macro Linkage Library Function

1.3 General Rule for Macro Instructions

(1) Transfer of parameters
When the CPMS macro linkage library is used, parameters are transferred in the form of address or contents. For example, when creating a user program in the C language, parameters are transferred as follows:

```
long tn ;
tn = 100 ;
abort (&tn) ;
```

When the ABORT macro instruction is used, the address containing tn (= 100) is described in an argument. (&tn indicates the pointer to tn or the address containing tn.)
This must not be written as abort (tn).
When the C language is used for writing programs, there are various description methods in addition to the above. Create a program by using a method that permits easy coding. A coding example is shown below.

● When parameters comprises the whole array:

```
long    x[n] ;
macro   (x) ;
```

● When parameters are a part of array elements:
(The lower 3 are equal.)

```
long    x [n] ;        long    x [n] ;        long   * x [i] ;
x [i]=100 ;            x [i]=100 ;            * x [i]=100 ;
macro (&x[i]) ;        macro (x+i) ;          macro (x[i]) ;
```

● When parameters are simple variables:
(The lower 2 are equal.)

```
long      x ;          long   * x ;
x=100 ;                * x=100 ;
macro (&x) ;           macro (x) ;
```

(2) Return code
The execution result of a macro instruction is returned as a return code from the macro processing module of the CPMS. When a macro instruction is used as a function, the processing result of the macro instruction can be judged by the return code as shown in below.

```
long macro (   ) , rtn ;
long * x ;

rtn = macro (x) ;
if (rtn)
  {
      ⋮
  }
```

Note: When macro instruction processing is normally performed, 0 is usually returned as a return code. However, for some macro instructions, a value other than 0 is returned as a return code that indicates normal processing.

1.4 Macro Instruction Parameter Check

Macro instructions are direct data exchange between a user program and the CPMS. If a parameter is wrong, this may cause a system malfunction or system failure.

For the CPMS macro instructions, a rationality check is made on their major parameters by software. If any parameter is regarded as irrational, it is reported as a macro parameter error and the macro instruction issuing task is aborted.

Figure 2-2 shows the relationship among TNs when making a parameter check.

In the parameter check for each macro, the maximum user task number is 224. A memory protect check is conducted in accordance with the information (accessibility/inaccessibility) supplied for individual tasks.

Table 2-1 Relationship among TNs at Parameter Check

Target TN / Issuing TN	User task 1 to 224	System task, ROM task 225 to 255	Error TN 256 to
1 to 224	Normally processed	Do not issue.	*
225 to 255	Normally processed	Normally processed	*

When a task with an issuing TN issues a parameter check instruction for a task with target TN, one of the following results is obtained.
Normally processed: Normal execution takes place for each.
Do not issue: Execution takes place for each, but the results is unpredictable. Do not issue the instruction.
*: Detected as a parameter error.
When target TN = 0, the CPMS does not regard it as a parameter error but performs no processing and returns 1.

1.5 CPMS Macros

(1) Task management
rleas
queue
exit
abort
wait
post
susp
rsum
asusp
arsum
chap
sfact
gfact

(2) Memory management
wrtmem
chkbmem
chktaer
mvmem
uspchk

(3) Timer management
timer
ctime
delay
stime
gtime
wake
cwake

(4) Shared resource management
rserv
prsrv
free
pfree

(5) System management
wdtset

(6) System services
getsysinfo
gettaskinfo
gtkmem
usrdhp
usrel
save_env
resume_env
gettimebase
TimebaseToSecs
atmswap
atmand
atmor
atmxor
atmadd
atmtas
atmcas

Notes on coding

CPMS has the following include files:

- cpms_types.h Defines variable types used by macros.
- cpms_macro.h Defines macro functions.
- cpms_errno.h Defines return codes from macros.
- cpms_table.h Defines the structures of the tables in CPMS.
- cpms_dhp.h Defines the codes used in the DHP.
- cpms_elog.h Defines the codes and structures used in error log.
- cpms_ulsub.h Defines the codes and structures used by built-in subroutines.

When using a system management macro of class (5) above for loading, specify the -lsysctl option in loadhr.

NAME

rleas - Make a task wait to be started up.

SYNOPSIS

int rleas(&tn)
long tn;

DESCRIPTION

The rleas macro checks whether the task specified by the tn parameter is in the DORMANT state. If so, the macro places the task in the IDLE state. When the task is not in the DORMANT state, the macro has no effect.
Specify the task number of the task in the tn parameter.

When the task is aborted during its I/O processing, the task enters the DORMANT state and is not actually aborted until it has the I/O processing completed. When an rleas macro is issued to a task in the DORMANT state which is performing I/O processing, rleas terminates normally as soon as possible, with the return code 0 returned. The task aborts when it has the I/O completed, and then enters the IDLE state.

DIAGNOSTICS

Upon normal termination, a 0 is returned. Otherwise, one of the following values is returned:

1: The value of tn is 0.
3: The task specified by tn is not in the DORMANT state.
4: The task specified by tn is not registered.

PARAMETER CHECK

Parameter check is performed to see whether the following requirements are satisfied. If not, a parameter check error is returned.

- The task number specified by tn is within the range of 0 to the maximum task number.

NAME

queue - Start up a task.

SYNOPSIS

int queue(&tn, &fact)
long tn, fact;

DESCRIPTION

The queue macro makes the task specified by the tn parameter wait to be executed by the CPU if the task is in the IDLE state. Making a task wait for CPU execution is called starting up the task.

Tasks waiting for CPU execution are dispatched in their order of priority: the higher the priority, the faster the task is executed. When the specified task has a higher priority than the task that issued rleas, the specified task is dispatched. When the specified task has a lower priority, the task that issued rleas is executed.
Specify the task number of the task in the tn parameter.

When the task is made to wait for CPU execution, the setting of the fact parameter is set as the start factor. Up to 32 start factors can be set. Note that the same factor cannot be set repeatedly. When a start factor is fetched by a gfact macro, the factor is deleted.

When the value of the specified start factor does not fall in the range of 1 to 32, the task is processed, assuming that no start factor is specified.

Another queue macro can be issued to the task specified to wait for CPU execution. After the task terminates, it is made to wait for CPU execution once more. Multi-execution requests like this are stored up to two levels. This means that when the specified task is waiting for CPU execution, the task is queued in such a way that it is started up once more; when the task is not waiting for CPU execution, the task is queued in such a way that it can wait for CPU execution twice. When the execution of the task aborts, however, the second startup request queued for the task is canceled.

DIAGNOSTICS

Upon normal termination, a 0 is returned. Otherwise, one of the following values is returned:

1: The value of tn is 0.
2: The task specified by tn is already in the DORMANT state.
4: The task specified by tn is not registered.

PARAMETER CHECK

A parameter check is performed to see whether the following requirements are satisfied. If not, a parameter check error is returned.

- The task number specified by tn is within the range of 0 to the maximum task number.

NAME

exit - Terminate a task.

SYNOPSIS

exit()

DESCRIPTION

The exit macro terminates the execution of the task that issued the macro, that is, places the task in the IDLE state rather than making it wait to be executed by the CPU.

The task priority that was changed by a chap macro during task execution is restored to the value at task registration.

All shared resources having been locked by an rserv macro are released.

The monitoring of task termination is also canceled.

Where the EXS built-in subroutine is registered, the exit macro links to EXS.

When startup requests are queued, the task is made to wait for CPU execution again after processing to terminate the task is completed.

DIAGNOSTICS

Issuing an exit macro does not return, control to the task. Therefore, there is no return code.

NAME

abort - Forcibly terminate a task.

SYNOPSIS

int abort(&tn)
long tn;

DESCRIPTION

The abort macro forcibly terminates the task specified by the tn parameter and places it in the DORMANT state. When the execution of the task is inhibited or the task is waiting for an event, abort shifts the task from those states to the DORMANT state.

The task priority that was changed by a chap macro during task execution is restored to the value at task registration.

All shared resources having been locked by an rserv macro are released.

The start factors set for the task are cleared.

The monitoring of task termination is also canceled.

When the ABS built-in subroutine is registered, abort links to ABS.

The timer event registered by the specified task using a timer macro is not canceled. Timer event-based startup for the task is also not canceled unless the task is in the DORMANT state, in which case the startup is not handled normally.

DIAGNOSTICS

Upon normal termination, a 0 is returned. Otherwise, one of the following values is returned:

1: The value of tn is 0.
2: The task specified by tn is already in the DORMANT state.
4: The task specified by tn is not registered.

PARAMETER CHECK

Parameter check is performed to see whether the following requirement is satisfied. If not, a parameter check error is returned.

- The task number specified by tn is within the range of 0 to the maximum task number.

NAME

wait - Suppress task execution until an event is generated.

SYNOPSIS

int wait(&ecb_g)
long ecb_g;

DESCRIPTION

The wait macro makes the task that issues it wait for an event to be generated by a post macro. The event to be awaited can be specified by the ecb_g parameter. In the ecb_g parameter, set the address of the pointer to the event control block (ECB) allocated in GLB.

When the value of the ecb_g parameter specified in the post macro matches the value of the ecb_g parameter in the wait macro, the task that has issued the wait macro is released from the waiting state and waits to be executed by the CPU.

When the event to be awaited is already generated by a post macro, the task that issued the wait macro cannot wait for the event.

When an event is generated by a post macro, the post code specified by the pcode parameter in post is set in bits 29 to 0 (bit 0 is the LSB) in the ECB.

The post code is reported as the return code from the wait macro. For instance, suppose that the task that issued a post macro sets the factor as the post code, by which an event is generated. Then, after the task that has issued a wait macro is released from the event-waiting state, the task can learn the factor from the post code.

Define an ECB for each event.

DIAGNOSTICS

Upon normal termination, the post code is returned.

PARAMETER CHECK

A parameter check is performed to see whether the following requirement is satisfied. If not, a parameter check error is returned.

- The ECB specified by ecb_g already used by the wait macro issued by another task.

NOTE

Be sure to initialize the ECB allocated in GLB to 0 before using it.

NAME

post - Generate an event and restart the task.

SYNOPSIS

int post(&ecb_g, &pcode)
long ecb_g;
long pcode;

DESCRIPTION

The post macro releases from the wait state the task that issued a wait macro and is waiting for an event. Then, post passes the post code specified by the pcode parameter. The event to be generated can be specified by the ecb_g parameter. In the ecb_g parameter, set the address of the pointer to the event control block (ECB) allocated in GLB. The post code is set in bits 29 to 0 (bit 0 is the LSB) in the ECB.

When the task released from the wait state by a post macro has a higher priority than the task that issued post, control is passed to the task with higher priority.

When the value of the ecb_g parameter specified in post matches the value of the ecb_g parameter in the wait macro, the task that issued wait is released from the event waiting state and waits to be executed by the CPU.

When there is no task that issued a wait macro and is waiting for the event to be generated, the post code specified by the pcode parameter is set in bits 29 to 0 (bit 0 is the LSB) in the ECB. This means that the fact the event was already generated is recorded. The post code set like this is passed when a wait macro in which the event is set is issued.

DIAGNOSTICS

When there is no task that is waiting for the event specified by ecb_g, processing is terminated normally with the return code 3. Upon normal termination under other conditions, the value of the return code is 0. In other cases, the following value is returned:
2: The task waiting for the specified event is in the DORMANT state.

PARAMETER CHECK

A parameter check is performed to see whether the following requirement is satisfied. If not, a parameter check error is returned.
• The post code specified by pcode is set in bits 29 to 0 (bit 0 is the LSB) in the ECB.

NAME

susp - Suppress the execution of a task.

SYNOPSIS

int susp(&tn)
long tn;

DESCRIPTION

The susp macro suppresses the execution of the task specified by the tn parameter. The specified task must be waiting to be executed by the CPU or be in the IDLE state.

The specified task is not released from the execution-suppressed state until an rsum or arsum macro is issued or the task is forcibly terminated by an abort macro.

When two or more susp macros are issued for the same task, only one macro has an effect. Therefore, one rsum macro is enough to release the task from the execution-suppressed state.

DIAGNOSTICS

Upon normal termination, a 0 is returned. Otherwise, one of the following values is returned:

1: The value of tn is 0.
2: The task specified by tn is in the DORMANT state.
3: The task specified by tn is already in the execution-suppressed state.
4: The task specified by tn is not registered.

PARAMETER CHECK

A parameter check is performed to see whether the following requirement is satisfied. If not, a parameter check error is returned.

- The task number specified by tn is within the range of 0 to the maximum task number.

NAME

rsum - Release a task from the execution-suppressed state.

SYNOPSIS

int rsum(&tn)
long tn;

DESCRIPTION

The rsum macro releases the task specified by the tn parameter from the execution-suppressed state if the task is placed in that state by a susp macro.

When the task released from the execution-suppressed state has a higher priority than the task that issued rsum, control is passed to the task with higher priority.

The rsum macro has no effect in a certain case. For instance, when an rsum macro is issued to a task placed in the execution-suppressed state by an asusp macro, the task is not released from that state.

DIAGNOSTICS

Upon normal termination, a 0 is returned. Otherwise, one of the following values is returned:

1: The value of tn is 0.
2: The task specified by tn is in the DORMANT state.
3: The task specified by tn is not placed in the execution-suppressed state by a susp macro.
4: The task specified by tn is not registered.

PARAMETER CHECK

Parameter check is performed to see whether the following requirement is satisfied. If not, a parameter check error is returned.

- The task number specified by tn is within the range of 0 to the maximum task number.

NAME

asusp - Suppress the execution of multiple tasks.

SYNOPSIS

int asusp()

DESCRIPTION

The asusp macro suppresses the execution of all tasks other than the task that has issued the macro. CPMS records the number of times asusp is issued in the execution-suppressed counter. The counter is incremented by one each time asusp is issued, while the counter is decremented by one each time arsum is issued. When the counter is 0, it is no longer decremented. When the counter is greater than 0, all tasks other than the task that issued asusp are placed in the execution-suppressed state. Only one asusp macro is executed at a time.

When the task that issued asusp issues a wait or exit macro, the execution suppression counter is set to 0. The counter is also set to 0 when the task that issued asusp aborts.

The rsum macro cannot cancel the execution-suppressed state enabled by asusp. When an susp macro is issued to a task placed in the execution-suppressed state by asusp, the execution suppression counter remains unchanged. In this case, however, the fact that execution was suppressed by susp is recorded. To release the task from this state, the counter must be zero-cleared by arsum and the execution-suppressed state enabled by susp must be released by rsum.

DIAGNOSTICS

After asusp is executed, the value of the execution suppression counter is returned.

NOTES

The overhead of asusp increases as the number of tasks being executed increases.

Asusp is a macro that locks the CPU. However, it cannot be used to lock other resources.

When asusp causes a conflict with another task for the resource to be locked by a rserv macro, a deadlock occurs. After asusp is issued, do not issue any processing routine or macro that causes a conflict for resources.

Minimize the time during which asusp is active. Otherwise, the system operation may be affected adversely. Other macros should not be issued from when asusp is issued until arsum is issued.

NAME

arsum - Release multiple tasks from the execution-suppressed state.

SYNOPSIS

int arsum()

DESCRIPTION

The arsum macro clears the execution-suppressed state effected by an asusp macro. CPMS records the number of times asusp is issued in the execution suppression counter. The counter is incremented by one each time asusp is issued, while the counter is decremented by one each time arsum is issued. When the counter is 0, it is no longer decremented. When the counter is greater than 0, all tasks other than the task that issued asusp are placed in the execution-suppressed state.

When the task that issued asusp issues a wait or exit macro, the execution suppression counter is set to 0. The counter is also set to 0 when the task that issued asusp aborts.

The rsum macro cannot clear the execution-suppressed state effected by asusp in a certain case. For instance, when an arsum macro is issued to a task placed in the execution-suppressed state by a susp macro, that state is not cleared.

When the task released from the execution-suppressed state has a higher priority than the task that issued arsum, control is passed to the task with higher priority.

DIAGNOSTICS

After arsum is executed, the value of the execution suppression counter is returned.

0: The execution-suppressed state effected by asusp is already released.

n: To clear the execution-suppressed state effected by asusp, arsum must be issued n more times.

NAME

chap - Temporarily change a task priority.

SYNOPSIS

int chap(&tn, &chgp)
long tn, chgp;

DESCRIPTION

The chap macro temporarily changes the priority of the task specified by the tn parameter to the priority specified by the chgp parameter. After chap is executed, the priority of the specified task may be higher than that of the task that issued chap. In this case, control is passed to the specified task if the task is waiting to be executed by the CPU. When the specified task is the task that issued chap and its priority is lowered by chap, control is passed to a task with a higher priority.

The priority temporarily changed by chap remains in effect until the specified task terminates or aborts.

The chap macro can also be used to increase the priority of the task waiting for a resource to be freed. However, this does not mean that the resource is forcibly allocated to the task.

DIAGNOSTICS

Upon normal termination, a 0 is returned. Otherwise, one of the following values is returned:

1: The value of tn is 0.
2: The task specified by tn is in the DORMANT state.
4: The task specified by tn is not registered.

PARAMETER CHECK

A parameter check is performed to see whether the following requirements are satisfied. If not, a parameter check error is returned.

- The task number specified by tn is within the range of 0 to the maximum task number.
- When the specified task is a system task, the priority specified by chgp is within the range of 0 to 31.
- When the specified task is a user task, the priority specified by chgp is within the range of 4 to 27.

NAME

sfact - Set a start factor for a task.

SYNOPSIS

int sfact(&tn, &fact)
long tn, fact;

DESCRIPTION

The sfact macro sets the start factor specified by the fact parameter for the task specified by the tn parameter. When the value of the specified start factor does not fall in the range of 1 to 32, the task is processed, assuming that no start factor is specified.

The set start factor is read by a gfact macro and then cleared. Even when an attempt is made to set the same start factor twice or more for the same task, it is set only once. The start factor is cleared by a single gfact macro.

When the specified task is in the DORMANT state, no start factor is set. When the specified task aborts, all start factors set for the task are cleared.

DIAGNOSTICS

Upon normal termination, a 0 is returned. Otherwise, one of the following values is returned:

1: The value of tn is 0.
2: The task specified by tn is in the DORMANT state.
4: The task specified by tn is not registered.

PARAMETER CHECK

A parameter check is performed to see whether the following requirement is satisfied. If not, a parameter check error is returned.

• The task number specified by tn is within the range of 0 to the maximum task number.

NAME

gfact - Read a start factor of a task.

SYNOPSIS

```
int gfact(fact)
long *fact;
```

DESCRIPTION

The gfact macro reads a start factor set for the task that issued the macro. Only one start factor is read by a single gfact macro at the address specified by the fact parameter, in the ascending order of numerical values.

After the start factor is read, it is cleared. The remaining start factors can be read by issuing gfact again. When all start factors are read, gfact returns a 0 to the address specified by the fact parameter.

Make sure that when a task is started up, all its start factors are read by gfact.

DIAGNOSTICS

Upon normal termination, a 0 is returned.

NAME

wrtmem - Write to protected memory.

SYNOPSIS

```
int wrtmem(vaddr, dst, size)
long *vaddr;
long *dst;
int size;
```

DESCRIPTION

The wrtmem macro writes to protected memory and is used by programming tasks to write programs and data to such memory.

PARAMETERS

vaddr: First address of the transfer source. (Specify an address on a four-byte boundary.)

dst: First address of the transfer destination. (Specify an address on a four-byte boundary.)

size: Number of data items. (Specify data in bytes. The number of data items must be a multiple of four within 8,192.)

DIAGNOSTICS

Upon normal termination, a 0 is returned. Otherwise, one of the following values is returned:

1: Parameter error. (One of the vaddr, data, and size parameters is invalid.)

NOTES

Depending on the specified address, the wrtmem macro may destroy the program. CPMS cannot prevent the macro from destroying the program.

NAME

chkbmem - Check access to bus memory.

SYNOPSIS

int chkbmem(slot)
long slot;

DESCRIPTION

The chkbmem macro returns a value that indicates whether access to bus memory in the specified slot is possible.

PARAMETER

slot: Number of the slot where bus memory is mounted (0 to 7)

DIAGNOSTICS

Upon normal termination, a 0 is returned. Otherwise, combination of the following values is returned:
0x8000: Parameter error.
0x8001: Not mounted.
0x8004: Target abort error was detected. (Failure)

NAME

chktaer - Check for a target abort.

SYNOPSIS

int chktaer(slot)
long slot;

DESCRIPTION

The chktaer macro returns a value that indicates whether there is a target abort in bus memory in the specified slot.

PARAMETER

slot: Number of the slot where bus memory is mounted (0 to 7)

DIAGNOSTICS

Return codes
0: There was a target abort.
1: There was no target abort.

NAME

mvmem - Data transfer between specified areas

SYNOPSIS

```
int mvmem(wno, daddr, saddr)
long *wno;
long *daddr;
long *saddr;
```

DESCRIPTION

The mvmem macro transfers the specified number of bytes of data from the address specified in saddr to the address specified in daddr.

PARAMETER

wno: Number of transfer words
daddr: Beginning address of data transfer destination
saddr: Beginning address of data transfer source

DIAGNOSTICS

The return code is always 0.

NAME

uspchk - Check the number of bytes used for stack.

SYNOPSIS

int uspchk(usebyt, addr)
long *usebyt;
long *addr;

DESCRIPTION

The uspchk macro checks if the task having issued this macro instruction uses the stack exceeding the number of bytes specified in the parameter. It makes a check with stack size used when this macro is called.

PARAMETER

usebyt: Number of bytes of the stack area to be checked
addr: Mask of the status register to be changed

DIAGNOSTICS

Upon normal termination, a 0 is returned and the free capacity up to the number of bytes of the stack area specified in addr is returned.
Upon abnormal termination, a 1 is returned and the capacity exceeding the number of bytes of the stack area specified in addr.

REMARKS

- It is the most effective to execute this macro instruction at the deepest position of program nesting.
- After completion of debug, we recommend the user to delete this macro instruction from the program.
- The user must describe the error handling according to the judgement of a return code.

NAME

timer - Register a task that starts up based on a timer event.

SYNOPSIS

int timer(&id, &tn, &fact, &t, &cyt)
long id, tn, fact, t, cyt;

DESCRIPTION

The timer macro registers the task specified by the tn parameter so that it is started up based on the timer event specified by the id parameter. Four timer events are specifiable: length-of- time basis, time-of-day basis, length-of-time and cycle basis, and time-of-day and cycle basis. These events are explained in the table below.

When the time of day that already elapsed is specified to register a timer event, the time is registered as the same time on the next day. When the time of day is advanced by an stime macro and the timer event with the time of day specified is skipped, the time is registered as the same time on the next day.

The start factor specified by the fact parameter is passed to the task subject to timer event-based startup.

When the value of the specified start factor does not fall in the range of 1 to 32, the task is processed, assuming that no start factor is specified.

When a timer event is to be registered, the status of the specified task is not checked. When the specified task is in the DORMANT state on the occurrence of the timer event, the task does not start up.

To cancel the timer event, use a ctime macro. Even when a task for which a timer event is registered is aborted or deleted, the timer event is not canceled.

PARAMETERS

id: Timer event type (1 to 4)
tn: Task number of the task for which a timer event is to be registered
fact: Start factor to be passed to the task to be started up
t: Time of day for the first timer event or length of time relative to the current time (in milliseconds)
cyt: Duration of the cycle when events are to be generated cyclically (milliseconds). Specify a 0 when id is 1 or 2, or a value greater than 0 but smaller than or equal to 86,400,000 when id is 3 or 4.

Explanation of the id, t, and cyt parameters in the timer macro

Timer event	id	t	cyt	Explanation
Length-of-time basis	1	Relative time up to the start time, measured from the current time of day	0	After the length of time specified by the t parameter elapses, the task specified by the tn parameter is started up.
Time-of-day basis	2	Time of day at which the task is started up, measured from 00:00	0	The task specified by the tn parameter is started up at the time specified by the t parameter.
Length-of-time and cycle basis	3	Relative time up to the start time, measured from the current time of day (relative time up to the first startup)	Interval at which the task is started up cyclically after the first startup	After the length of time specified by the t parameter elapses, the task specified by the tn parameter is started up. Then, the task is started up cyclically at the interval specified by the cyt parameter.
Time-of-day and cycle basis	4	Time of day at which the task is started up, measured from 00:00 (time of day at the first startup)	Interval at which the task is started up cyclically after the first startup	The task specified by the tn parameter is started up at the time specified by the t parameter. Then, the task is started up cyclically at the interval specified by the cyt parameter.

DIAGNOSTICS

Upon normal termination, a 0 is returned. Otherwise, one of the following values is returned:

1: The value of tn is 0.

4: No more timer events can be registered because the system tables are full.

PARAMETER CHECK

Parameter check is performed to see whether the following requirements are satisfied. If not, a parameter check error is returned.

- The task number specified by tn is within the range of 0 to the maximum task number.
- The value of the type specified by id is within the range of 1 to 4.
- When the value of id is 1 or 3, the length of time specified by t is greater than 0 but does not exceed 86,400,000.
- When the value of id is 2 or 4, the time of day specified by t is greater than or equal to 0 but less than 86,400,000.
- When the value of id is 1 or 2, the value of cyt is 0.
- When the value of id is 3 or 4, the value of cyt is greater than 0 but does not exceed 86,400,000.

NAME

ctime - Cancel timer events for a task.

SYNOPSIS

int ctime(&tn, &fact)
long tn, fact;

DESCRIPTION

The ctime macro cancels timer events registered by a timer macro for the specified task.

Specifically, ctime searches for the timer event having the task number specified by the tn parameter and the start factor specified by the fact parameter, and deletes all occurrences of the timer event. When the value of the specified start factor does not fall into the range of 1 to 32, it is assumed that no start factor is specified.

The ctime macro cannot cancel the execution of a task that was already started up. When cyclic events to be generated after the current time are registered, however, these events are canceled.

DIAGNOSTICS

Upon normal termination, a 0 is returned. Otherwise, the following value is returned:
1: The timer event having the specified task number and start factor is not registered.

PARAMETER CHECK

A parameter check is performed to see whether the following requirement is satisfied. If not, a parameter check error is returned.

- The task number specified by tn is within the range of 0 to the maximum task number.

NAME

delay - Suppress task execution for a specified time.

SYNOPSIS

int delay(&t)
long t;

DESCRIPTION

The delay macro suppresses the execution of the task that issued the macro for the time specified by the t parameter.

In the t parameter, specify the time in milliseconds during which task execution is to be suppressed. During the suppression of task execution, another task has control. After the task execution has been suppressed for the specified time, control is returned to the task that issued delay if there are no other runnable tasks (that have higher priorities than the task that issued delay or have the same priority but have been already started up).

DIAGNOSTICS

Upon normal termination, a 0 is returned. Otherwise, the following value is returned:
4: Task execution could not be suppressed because of insufficient system tables.

PARAMETER CHECK

A parameter check is performed to see whether the following requirement is satisfied. If not, a parameter check error is returned.

- The time during which to suppress task execution specified by t is greater than 0 but does not exceed 86,400,000 (24 hours)

NOTE

For system operation reasons, the delay macro should not be issued while shared resources are being locked.

NAME

stime - Set the time of day.

SYNOPSIS

```
int stime(&time)
struct{
    short year;
    short month;
    short day;
    short week;
    long msec;
} time;
```

DESCRIPTION

The stime macro changes the time of day managed by CPMS to the time specified by the time parameter, and also sets the time in the TOD.

Specify the time parameter as follows:
year: Calendar year from 1970 to 2069
month: Month
day: Day
week: Set 0 because it is not used.
msec: Time in milliseconds measured from 00:00 within the range of 0 to 86,399,000 (23:59:59)

DIAGNOSTICS

Upon normal termination, a 0 is returned. Otherwise, the following value is returned:
1: The time specified by time is invalid.

PARAMETER CHECK

A parameter check is performed to see whether the following requirement is satisfied. If not, a parameter check error is returned.

- The year is from 1970 to 2069. The month is from 1 to 12. The day is from 1 to 31.
  The number of milliseconds is from 0 to 86,399,000.

NOTE

The timer events registered by the timer macro are affected at the time of their occurrence as described below.

Timer event type	When time is delayed	When time is advanced	Remarks
Length-of-time basis and length-of-time and cycle basis	The time of the timer event is not affected.	The time of the timer event is not affected.	After the length of time specified by the timer macro elapses, the timer event is generated.
Time-of-day basis and time-of-day and cycle basis	When the time to generate the timer event is delayed 24 hours or more, the time is registered as the same time on the next date.	The scheduled start time is shifted so that the time resulting from adding the cyclic time to the first scheduled start time may be behind the changed time. If the first scheduled start time has elapsed and the start timing is lost, the timer is started when it is changed.	

The timer events registered by the wake macro are affected at the time of their occurrence as described below.

Timer event type	When time is delayed	When time is advanced	Remarks
Time-of-day basis	(When Don't care is specified) When the start time is delayed 24 hours or more by delaying the time, the time is set to the same time on the same date after a change. (When the absolute time specified) The time is not affected by time change.	When the start time is skipped by advancing the time, the time is registered as the same time on the next date.	
Time-of-day basis and cycle basis		When the start time is skipped by advancing the time, the time is registered so that the time resulting from adding the cyclic time to the first start time may be behind the newly set time.	

NAME

gtime - Read the current time.

SYNOPSIS

```
int gtime(time)
struct{
        short year;
        short month;
        short day;
        short week;
        long msec;
}*time;
```

DESCRIPTION

The gtime macro stores the current time at the address specified by the time parameter.
The following values are returned to the address specified by the time parameter:
year: Calendar year from 1970 to 2069
month: Month
day: Day
week: Week (1 to 7 correspond to Sunday to Saturday.)
msec: Time in milliseconds measured from 00:00

DIAGNOSTICS

Upon normal termination, a 0 is returned.

NAME

wake - Start a task at the specified time

SYNOPSIS

```
int         wake(&id, &tn, &fact, &time, &cycle)
long id;
long tn;
long fact;
TIME time;
Long cycle;
```

DESCRIPTION

The wake macro starts a task at the start time specified in the parameter. When Cyclic start is specified, this macro starts a task at the start time and starts the task at each specified cyclic time. As a task start factor, fact specified in the parameter is transferred to the task.

PARAMETER

id: Start mode (0: Time start, 1: Cyclic start)
tn: Task No. of the task to be started
fact: Start factor that is transferred to the task to be started
time: Pointer to the TIME structure to set the start time
short sec: Specified in units of second supposing that 00:00 am is 0.
(0≦sec≦86399)
short day: Specified by day.
short month: Specified by month.
short year: Specified by Gregorian year (1970≦year≦2069)
long week: Set 0 because it is not used.
cycle: Specify the cyclic time. (1≦msec≦863400)

• The relation among id, time, and cycle is as follow.

id	time	cycle	Contents
0	Start time	0	The macro starts the specified task only once at the time specified in time.
1	First start time	Cyclic start time after the first start time	The macro starts the specified task at the time specified in time. After that, the macro starts the requested task in the cycle specified in cycle.

• The start time can be set as follows by using the Don't care code (= -1).

No.	year	month	day	sec	Contents
1	1990	1	10	36610	Start at 10 seconds, 10 minutes, 10 hours, 10th day, January, 1990 (The absolute hour is specified.)
2	-1 Don't care	1	10	36610	Start at 10 seconds, 10 minutes, 10 hours, 10th day, January, this year or next year (*2)
3	(*1)	-1 Don't care	10	36610	Start at 10 seconds, 10 minutes, 10 hours, 10th day, this month or next month (*2)
4	(*1)	(*1)	-1 Don't care	36610	Start at 10 seconds, 10minutes, 10 hours, today or tomorrow (*2)

(*1) Higher-order data than the Don't care code is ignored.
(*2) When the time is ahead of the present time, the task is started next year, next month or next day. If the time is behind the present time, the task is started this year, this month or today.

DIAGNOSTICS

Upon normal termination, a 0 is returned.
1: tn = 0
4: Cannot be registered because of an insufficient system table.

NAME

cwake - Cancel the task start request registered by wake.

SYNOPSIS

```
int         cwake(&tn, &fact)
long tn;
long fact;
```

DESCRIPTION

The cwake macro cancels the start request registered by wake.
This macro cancels a task from the start request when the specified tn matches fact.

PARAMETER

tn: Task No. of the task to be started
fact: Start factor to be transferred to started

DIAGNOSTICS

Upon normal termination, a 0 is returned.
1: Any task that matches the specified one is not found in the system table.

NAME

rserv - Lock shared resources in a centralized manner.

SYNOPSIS

#include <cpms/cpms-_rsrv.h>

int rserv(&n, &para1, &para2, …)
long n;
cpms_rserv_t para1, para2, …;

DESCRIPTION

The rserv macro locks the shared resources specified by parameters para1, para2, and so on. The rserv macro returns with an error when the task that issued the macro has been locking shared resources with another rserv macro. Since shared resources are locked in a centralized manner like this, a deadlock is prevented.

The rserv macro does not perform exclusive control for shared resources between rserv and prsrv.

When the specified task does not yet lock shared resources with another rserv macro, the issued rserv macro checks whether the shared resources specified by parameters para1, para2, and so on are being locked by other tasks. If all specified shared resources are free, the specified task locks them. If some of the specified shared resources are locked by other tasks, control is not returned from rserv, suppressing the execution of the specified task.

When the resource-locking tasks release all shared resources with free macros, the specified task in the execution-suppressed state can lock them and return from rserv.

The shared resources are freed when a free macro is issued or when the task that is locking them terminates or aborts.

The shared resources being locked with a rserv macro cannot be freed with a pfree macro.

The task that has issued rserv is also placed in the execution-suppressed state when system tables are insufficient. This is true even when the specified resources are not being locked by other tasks. System tables have available space when other tasks free the shared resources with free macros. At this time, the task locks the shared resources and returns from the rserv macro. The number of system tables used for shared resource management is defined in CPMS. Be sure to specify shared resources within the range of the defined number of system tables.

Do not issue rserv after issuing a susp or asusp macro that suppresses the execution of an other task(s). If an execution-suppressed task is locking a shared resource, a deadlock occurs.

A shared resource is represented as an area in SAREA of GLB.

Specify the n parameter and cpms_rserv_t structure as shown below.

n: Number of resources to be locked (1 to 5)

```
typedef struct cpms_rserv{
        long type;
        long addr;
        long top;
        long last;
}cpms_rserv_t;
```

type: This parameter has no meaning and is always set to 0.
addr: Address of SAREA including the shared resource to be locked
top: Starting address of the resource to be locked, relative to the beginning of SAREA
last: End address of the resource to be locked, relative to the beginning of SAREA

DIAGNOSTICS

When all specified resources are locked, a 0 is returned.
Otherwise, the following value is returned:
2: The shared resource is already being locked with rserv or prsrv.

PARAMETER CHECK

A parameter check is performed to see whether the following requirements are satisfied.
If not, a parameter check error is returned.

- The value of addr is valid.
- The value of top or last is valid.
- The number of resources specified by n is within the range of 1 to 5.

NOTE

When resources are being locked with prsrv, no resource can be locked with rserv.

NAME

free - Free the shared resources being locked by rserv.

SYNOPSIS

```
#include <cpms/cpms_rsrv.h>

int free(&n, &para1, &para2, …)
long n;
cpms_rserv_t para1, para2, …;
```

DESCRIPTION

The free macro frees the resources being locked by an rserv macro. Of the shared resources specified by parameters para1, para2, and so on, the free macro frees the shared resources being locked, if any.

The tasks waiting for shared resources to be freed are released from the execution-suppressed state when the shared resources are freed.

When the specified shard resources include a resource not being locked, the free macro gives the return code 1. Even in this case, the shared resources being locked are freed.

The shared resources being locked with a prsrv macro cannot be freed with a free macro.

A shared resources is represented as an area in SAREA of GLB.

Specify the n parameter and cpms_rserv_t structure as shown below.

n: Number of resources to be freed (1 to 5)

```
typedef struct cpms_rserv{
        long type;
        long addr;
        long top;
        long last;
}cpms_rserv_t;
```

type: This parameter has no meaning and is always set to 0.
addr: Address of SAREA including the shared resource to be freed
top: Starting address of the resource to be freed, relative to the beginning of SAREA
last: End address of the resource to be freed, relative to the beginning of SAREA

DIAGNOSTICS

When the shared resources are freed, a 0 or 1 is returned. Otherwise, the following value is returned:
2: All specified resources were already freed.

PARAMETER CHECK

A parameter check is performed to see whether the following requirements are satisfied. If not, a parameter check error is returned.

- The value of addr was valid.
- The value of top or last was valid.
- The number of resources specified by n was within the range of 1 to 5.

NAME

prsrv - Lock shared resources.

SYNOPSIS

```
#include <cpms/cpms_rsrv.h>

int prsrv(&n, &para1, &para2, …)
long n;
cpms_rserv_t para1, para2, …;
```

DESCRIPTION

The prsrv macro locks the shared resources specified by parameters para1, para2, and so on. When the task that issued prsrv is already locking shared resources with another prsrv macro, the prsrv macro can be used to lock other resources.

The prsrv macro does not perform exclusive control for shared resources between prsrv and rserv.

The prsrv macro checks whether the shared resources specified by parameters para1, para2, and so on are being locked by other tasks. If all specified shared resources are free, the specified task locks them. If some of the specified shared resources are being locked by other tasks, control is not returned from prsrv, suppressing the execution of the task that issued that prsrv macro.

When a specified shared resource is being locked by the prsrv macro issued by the task itself, processing is performed, assuming that the shared resource is now locked. Note that the same resource may be locked more than once by the same task with prsrv. To free such a shared resource, pfree must be issued as many times as prsrv was issued.

When the resource-locking tasks release all shared resources with pfree macros, the task in the execution-suppressed state can lock them and return from prsrv.

The shared resources are freed when a pfree macro is issued or the task that is locking them terminates or aborts.

The shared resources being locked with a prsrv macro cannot be freed by a free macro.

The task that issued prsrv is also placed in the execution-suppressed state when system tables are insufficient. This is true even when the specified resources are not being locked by other tasks. System tables have available spaces when other tasks free shared resources with pfree macros. At this time, the task locks the shared resources and returns from the prsrv macro. The number of system tables used for shared resource management is defined in CPMS. Be sure to specify shared resources within the defined number of system tables.

A shared resource is represented as an area in SAREA of GLB.

Specify the n parameter and cpms_rserv_t structure as shown below.

n: Number of resources to be locked (1 to 5)

```
typedef struct cpms_rserv{
        long type;
        long addr;
        long top;
        long last;
}cpms_rserv_t;
```

type: This parameter has no meaning and is always set to 0.
addr: Address of SAREA including the shared resource to be locked
top: Starting address of the resource to be locked, relative to the beginning of SAREA
last: End address of the resource to be locked, relative to the beginning of SAREA

DIAGNOSTICS

When all specified resources are locked, a 0 is returned.
Otherwise, the following value is returned:
2: The number of shared resources sharable by a single task was exceeded.

PARAMETER CHECK

A parameter check is performed to see whether the following requirements are satisfied.
If not, a parameter check error is returned.

- The value of addr was valid.
- The value of top or last was valid.
- The number of resources specified by n was within the range of 1 to 5.

NAME

pfree - Free the shared resources being locked by prsrv.

SYNOPSIS

```
#include <cpms/cpms_rsrv.h>

int pfree(&n, &para1, &para2, …)
long n;
cmps_rserv_t para1, para2, …;
```

DESCRIPTION

The pfree macro frees the resources being locked by a prsrv macro. Of the shared resources specified by parameters para1, para2, and so on, the pfree macro frees the shared resources being locked, if any.

The tasks waiting for shared resources to be freed are released from the execution-suppressed state when the shared resources are freed.

When the specified shard resources include a resource not being locked, the pfree macro gives the return code 1. Even in this case, the shared resources being locked are freed.

The shared resources being locked by an rserv macro cannot be freed by a pfree macro.

A shared resource is represented as an area in SAREA of GLB.

Specify the n parameter and cpms_rserv_t structure as shown below.
n: Number of resources to be freed (1 to 5)

```
typedef struct cpms_rserv{
        long type;
        long addr;
        long top;
        long last;
}cpms_rserv_t;
```

type: This parameter has no meaning and is always set to 0.
addr: Address of SAREA including the shared resource to be freed
top: Starting address of the resource to be freed, relative to the beginning of SAREA
last: End address of the resource to be freed, relative to the beginning of SAREA

DIAGNOSTICS

When the shared resources are freed, a 0 or 1 is returned. Otherwise, the following value is returned:
2: All specified resources were already freed.

PARAMETER CHECK

A parameter check is performed to see whether the following requirements are satisfied. If not, a parameter check error is returned.

- The value of addr was valid.
- The value of top or last was valid.
- The number of resources specified by n was within the range of 1 to 5.

NAME

wdtset - Start or stop the watchdog timer.

SYNOPSIS

int wdtset(&msec)
long msec;

DESCRIPTION

The wdtset macro starts or stops the watchdog timer.

When the watchdog timer expires, the macro links to the WDTES built-in subroutine.

Have WDTES perform processing upon time-out of the watchdog timer.

PARAMETER

msec: Time to be set in the watchdog timer (0 to 65,535) in milliseconds.
When a value from 1 to 65,535 is set, the watchdog timer starts.
When a 0 is set, the watchdog timer stops immediately.

DIAGNOSTICS

0: Normal termination
1: Parameter error

NAME

getsysinfo - Get system status information.

SYNOPSIS

```
int getsysinfo(type, addr)
int type;
char *addr;
```

DESCRIPTION

The getsysinfo macro returns the system information identified by the type parameter to the address specified by the addr parameter.

In the type parameter, specify one of the following values:

- SYS_IDLE
  The accumulated idle time is returned.

```
struct sys_idle {
        unsigned int idle_sec;    /*Measured in seconds*/
        int long idle_nsec;    /*Measured in nanoseconds*/
};
```

The idle time is accumulated, assuming the time of CPMS start is 0. Obtain the idle time by calculating the difference between the accumulated idle time when the previous SYS_IDLE was issued and the current accumulated idle time.

Make sure that the measurement time is 24 hours or less.

- SYS_CPMS
  The version number of CPMS is returned.

```
int cpms_ver;
```

- SYS_PROC
  The processor number is returned.

```
int proc_no;
```

DIAGNOSTICS

Upon normal termination, the size in bytes of the information is returned as the return code. Otherwise, one of the following values is returned:
0: The system information identified by type was not intended for use in processing.
-1: System information could not be fetched correctly.

NAME

gettaskinfo - Return task status information.

SYNOPSIS

```
int gettaskinfo(type, tn, addr)
int type, tn;
char *addr;
```

DESCRIPTION

The gettaskinfo macro returns task information identified by the type parameter to the address specified by the addr parameter. Specify the task for which information is to be fetched in the tn parameter. When fetching information on the task that issued gettaskinfo, specify 0 in tn.

In the type parameter, specify one of the following values:

- TASK_TN
  The task number of the task that issued gettaskinfo is returned. Be sure to specify 0 in tn.

  int task_tn;

- TASK_PRI
  The priority of the task specified by tn is returned.

  int task_pri;

- TASK_STAT
  The current status of the task specified by tn is returned.

  int task_stat;

  0: Not registered, 1: DORMANT, 2: IDLE, 3: READY, 4: SUSPENDED, 5: WAIT

DIAGNOSTICS

Upon normal termination, the size in bytes of the information is returned as the return code. Otherwise, one of the following values is returned:

0: The task information identified by type was not intended for use in processing. Or, the task specified by tn was not registered.

-1: Task information could not be fetched correctly.

NAME

gtkmem - Read a table managed by CPMS.

SYNOPSIS

```
int gtkmem(tblno, caseno, offset, size, buf)
int tblno, caseno, offset, size;
char *buf;
```

DESCRIPTION

The gtkmem macro reads data from a table managed by CPMS.

PARAMETERS

tblno: Table number to be acted on:

- 1: OSCB table
- 2: SYSCB table
- 3: TCB table
- 4: TMCB table
- 5: RSCB table
- 6: RSVB table
- 7: UCB table
- 8: TRB table

caseno: Relative case number in the table. When the OSCB or SYSCB, TMCB or RSCB table is specified, specify 0 in caseno.

offset: Relative address in the case of the data to be read

size: Size in bytes of the data to be read

buf: Address of memory to be read

DIAGNOSTICS

Upon normal termination, a 0 is returned. Otherwise, one of the following values is returned:

1: The table specified by tblno was not intended for use in processing.

2: Data could not be read correctly from the table.

NAME

usrdhp - Write to the DHP record.

SYNOPSIS

```
#include <cpms_dhp.h>

int usrdhp(code, data, ndata)
unsigned long code;
long *data;
long ndata;
```

DESCRIPTION

The usrdhp macro records user-defined events in the kernel operation trace (DHP).

PARAMETERS

code: Trace code. Specify one of DHP_USR0 to DHP_USR7.
data: Pointer to the array where trace data is to be stored
ndata: Number of elements in the array (0 to 5; one case consists of four bytes).

DIAGNOSTICS

0: Normal termination
1: Parameter error

NAME

usrel - Write user error log.

SYNOPSIS

```
#include <cpms_elog.h>

int usrel(type, class, retcode, errtype, erb)
long type;
long class;
long retcode;
long errtype;
long *erb;
```

DESCRIPTION

After linking to the EAS built-in subroutine, the usrel macro writes the error information specified by arguments to the error log buffer area in the operating system.

PARAMETERS

type: Specifies one of the following severity types:

LOG_TYPE_NONFATAL
Errors that do not cause the system to go down. When degrading some of features, use this type. Errors of this type include program errors and I/O errors.
LOG_TYPE_WARNING
Warning errors. Use this type for recoverable errors. Errors of this type include those caused by insufficient resources such as temporarily insufficient memory.
LOG_TYPE_NOTE
Messages to provide the user with information

class: Specifies either of the following error message classes (provided for subsystem identification): To assign meanings to these classes is a task for the user.

LOG_CLASS_MSOFT1 to LOG_CLASS_MSOFT16 (for middleware)
LOG_CLASS_USER1 to LOG_CLASS_USER16 (for applications)

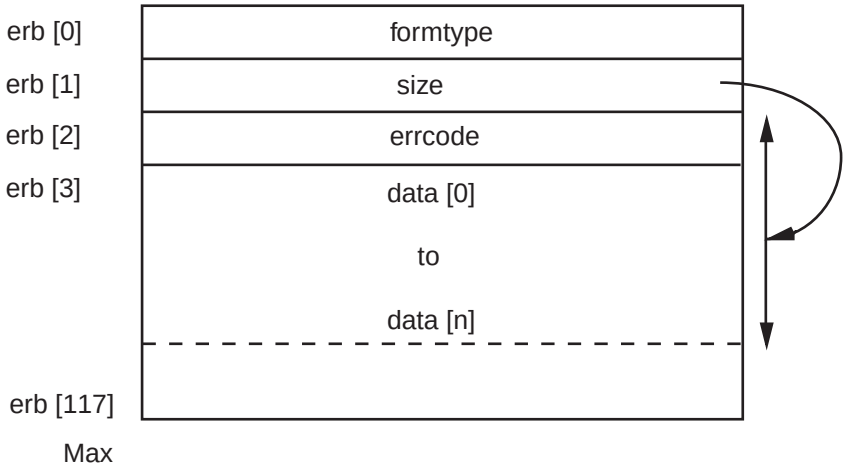
retcode: Specifies the return value by which a function was called immediately before error detection. When no function was called, set retcode to 0.

errtype: Specifies either of the following possible failure causes:

LOG_ERRTYPE_HARD (hardware)
LOG_ERRTYPE_SOFT (software)

erb: Specifies the pointer to the error block. The error block format is shown below.

● Error block format



formtype: Specifies a format type of error messages.

The following values are usable as format types:

LOG_FORM_MSOFT1 to LOG_FORM_MSOFT16 (for middleware)
LOG_FORM_USER1 to LOG_FORM_USER16 (for applications)
LOG_FORM_PIOERR (PI/O errors)
LOG_FORM_MODULERR (module errors)

To assign meanings to the format types for middleware and applications is a task for the user.

The formats for PI/O errors and module errors are defined by the operating system.

size: Specifies the size in bytes of the effective data after the errcode field. (4 to 464)
errcode: Specifies an error code:
0x08000000 to 0x08FFFFFF (for middleware)
0x09000000 to 0x09FFFFFF (for applications)

To assign meanings to these error codes is a task for the user. However, use the high-order 16 bits as the major part to specify an error type, and the low-order 16 bits as the minor part to specify a detail factor. It is also possible to use the error codes of PI/O errors and module errors defined by the operating system.

data: Detailed data on the error. Make sure that the format of this data matches the format specified in the formtype field.

DIAGNOSTICS

0: Normal termination
1: Parameter error

NAME

save_env - Save the execution environment of a task.

SYNOPSIS

```
#include <cpms_table.h>
int save_env(env)
struct task_env *env;
```

DESCRIPTION

The save_env macro saves the execution environment information for the task at the time save_env is issued. The address at which to save the data can be specified in the env parameter. The saved data is used by the resume_env macro.

In the env parameter, specify the address of the task_env structure in which the execution environment info for the task is to be saved.

The task_env structure is configured as shown below. It requires a 424-byte area.

```
struct task_env {
        struct basic_regs env_basic_regs;
        struct float_regs env_float_regs;
};
```

Allocate the area in which to save the execution environment info for the task in GLB.

The save_env macro issued from a built-in subroutine does not perform any processing.

DIAGNOSTICS

When the execution environment info for the task is saved successfully by save_env, a 0 is returned.

When control is returned from save_env as the result of issuing a resume_env macro, the value specified by val in resume_env is given as the return code. When a 0 is specified by val, a 1 is returned.

NOTE

When control is returned by resume_env to the point where save_env was issued, the contents of the user stack or the BSS or GLB data related to task control may differ from those at the time save_env was issued. If this happens, resume_env may not assure the same processing as before.

NAME

resume_env - Restore the execution environment of a task.

SYNOPSIS

```
#include <cpms_table.h>
void resume_env(env, val)
struct task_env *env;
int val;
```

DESCRIPTION

The resume_env macro restores the execution environment of the task specified in the parameter env. The execution environment of the task to be restored is only the register, so the contents of the user task and BSS are not restored.

Make sure that resume_env is issued from the CPES built-in subroutine. resume_env is effective as long as neither a task abort nor a CPU down occurs according to the execution result of the CPES built-in subroutine. The task execution environment is restored not immediately after resume_env is issued, but after all entries of CPES are executed. Then, control is passed to the return address specified by the save_env macro corresponding to the env parameter.

Upon normal restoration of the task execution environment, control is returned from save_env that saved the execution environment of the specified task. At the same time, save_env returns the val parameter as the return code. When a 0 is specified by val, save_env returns a 1.

When resume_env is issued more than once from multiple entries in the CPES built-in subroutine, the parameters in the last resume_env macro are in effect.

DIAGNOSTICS

The resume_env macro does not give any return code.

NOTES

The resume_env macro must be issued from the CPES built-in subroutine.

When resume_env is issued from other than CPES, it does not perform any processing.

When control is returned by resume_env to the point where save_env was issued, the contents of the user stack or the BSS or GLB data related to task control may differ from those at the time save_env was issued. If this happens, resume_env may not assure the same processing as before.

If the env parameter is specified incorrectly, the CPU may go down.

NAME

gettimebase - Read the time base.

SYNOPSIS

void gettimebase(timebase)
unsigned long timebase [2];

DESCRIPTION

The gettimebase macro reads and returns the 64-bit time base. The time base is incremented every four bus clocks. Since the bus clock of the CMU operates at 39.996 MHz, the time base is incremented at a rate of 9.999 MHz. Division of the time base by 9,999,000 results in the number of seconds measured from January 1, 1970, 00:00:00.

PARAMETERS

timebase[0]: High-order 32 bits of the time base register (TBU)
timebase[1]: Low-order 32 bits of the time base register (TBL)

NOTES

The time base depends on the model. In the future, the time base is likely to be handled in a different way depending on the model and operating frequency.

For the CMU, the contents of the area are 39996000 (0x02624a60).

NAME

TimebaseToSecs - Convert the time base value into seconds or nano seconds.

SYNOPSIS

```
void        TimebaseToSecs(timebase, tval)
unsigned long timebase [2];
struct tval{
            unsined int tv_sec;
            int         tv_nsec;
} tval;
```

DESCRIPTION

The TimebaseToSecs macro converts the 64-bit time base value into relative seconds from the year 1970 or nano seconds that are less than a second.

NAME

atmswap, and other macros - Atomic operation libraries

SYNOPSIS

```
long atmswap(addr, data)
long *addr, data;

long atmand(addr, data)
long *addr, data;

long atmor(addr, data)
long *addr, data;

long atmxor(addr, data)
long *addr, data;

long atmadd(addr, data)
long *addr, data;

long atmtas(addr, data)
long *addr, data;

long atmcas(addr, data1, data2)
long *addr, data1, data2;
```

DESCRIPTION

These libraries prevent another task or interrupt processing from rewriting the memory while the memory is read, changed or written, thus guaranteeing exclusive read, change and write operations. This permits exclusive control.

Each library handles only 32-bit integers (long int) as data. The return value in olddata is the value (addr) in memory before operation. The symbol "→addr" indicates that data is to be stored at the address specified by addr.

olddata=atmswap(addr, data):	data→addr
olddata=atmand(addr, data):	(addr) AND data→addr
olddata=atmor(addr, data):	(addr) OR data→addr
olddata=atmxor(addr, data):	(addr) XOR data→addr
olddata=atmadd(addr, data):	(addr) + data→addr
olddata=atmtas(addr, data):	Test And Swap if (addr) = 0 then data→addr
olddata=atmcas(addr, data1, data2):	Compare And Swap if (addr) = data1 then data2→addr

NOTE

Exclusive control achieved by these libraries takes effect between processing programs within the local processor. They cannot be used for exclusive control with another processor or I/O DMA.

PART 3 LIBRARIES

CHAPTER 1 OVERVIEW

1.1 Programming Requirements

When using library subroutines in a program, link the library by specifying the -l option in the svload command. Meet the following requirements when linking libraries:

- When using subroutines in libcrs.a
  Specify "-lcrs" in the svload command.

1.2 Order of Libraries Specified

Meet the following requirements when specifying libraries in the svload command:

- If the same name exists in the specified libraries, specify first the library containing the option file to be linked.

1.3 Names Defined in Libraries

The names defined in libraries are given below. Code a program so that a name will not be duplicated. When using a duplicated name, specify the library file after the object file to be linked. This prevents the library files from being linked first.

- libcrs.a

fpcheck	fpchecko	fpsetmask	fpgetmask
fpsetround	fpgetround	fpsetsticky	fpgetsticky

IEEE floating-point processing environment control subroutines

NAME

fpgetround, fpsetround, fpgetmask, fpsetmask, fpgetsticky, fpsetsticky - IEEE floating-point processing environment control

SYNOPSIS

```
#include <ieeefp.h>

typedef enum {
        FP_RN =0,    /*  round nearest    */
        FP_RZ =1     /*  round zero (truncate)    */
    } fp_rnd;

#define fp_except int
#define FP_X_INV        0x10    /* invalid operation exception  */
#define FP_X_OFL        0x04    /* overflow exception           */
#define FP_X_UFL        0x02    /* underflow exception          */
#define FP_X_DZ         0x08    /* devided by zeros exception   */
#define FP_X_IMP        0x01    /* imprecise(loss of precision) */

fp_rnd      fpgetround(void);
fp_rnd      fpsetround(fp_rnd rnd_dir);
fp_except   fpgetmask(void);
fp_except   fpsetmask(fp_except mask);
fp_except   fpgetsticky(void);
fp_except   fpsetsticky(fp_except sticky);
```

DESCRIPTION

These macros control floating-point rounding and floating-point exception occurrence.

(1) Rounding

Rounding is divided into 2 modes and controlled by fpgetround() and fpswtround().

FP_RN: Rounding to nearest

FP_RZ: Rounding to zero

The initial value of rounding is FP_RN.

(2) Floating-point exceptions

The floating-point exceptions that may occur in the CMU are as follows.

- FPU error (E): When FPSCR.DN=0 and a non-normalized value is input (*)
- Invalid operation (V): Invalid operation such as a NaN input
- Division by 0 (Z): Division by divisor 0
- Overflow (O): An operation result overflows.
- Underflow (U): An operation result underflows.
- Imprecise exception (I): An overflow, underflow or rounding occurs.

(*) In the CMU, FPSCR.DN=1 is set, so a non-normalized value is handled as 0 and no FPU error occurs.

An floating-point exception occurs when 1 set in the enable bit corresponding to the exception of the floating-point control register (FPSCR).
When a floating-point exception occurs, the corresponding bit in the FPU exception factor field of the floating-point control register (FPSCR) is set to 1 and 1 is stacked in the bit corresponding to the FPU exception flag field. When no FPU exception occurs, the corresponding bit in the FPU exception factor field is cleared to 0 and the bit corresponding to the FPU exception flag field is not changed.
The initial value of the floating-point exception enable bit is as follows:

- Invalid operation (V): Valid
- Division by 0 (Z): Valid
- Overflow (O): Valid
- Underflow (U): Invalid
- Imprecise exception (I): Invalid

Floating-point exception control is exerted by fpgetmask(), fpsetmask(), fpgetsticky(), and fpsetsticky().

- fpgetround returns the current rounding mode.
  FP_RN: Rounding to nearest
  PF_RZ: Rounding to zero
- fpsetround() sets the rounding mode and returns the previous rounding mode.
- fpgetmask() returns the current FPSCR exception enable bit value.

The relation between the exception mask and the FPSCR enable bit is shown below.

Exception mask	FPSCR enable bit
FP_X_INV	Invalid operation (V)
FP_X_DZ	Division by 0 (Z)
FP_X_OFL	Overflow (O)
FP_X_UFL	Underflow (U)
FP_X_IMP	Imprecise exception (I)

- fpsetmask() sets the FPSCR exception enable bit according to the exception mask value and returns the previous set value.
  The relation between the exception mask and the FPSCR exception enable bit is the same as that of fpgetmask().
- fpgetsticky() returns the value of the FPU exception flag field.
  The relation between the sticky flag and the FPSCR FPU exception flag field is shown below.

sticky flag	FPSCR flag field
FP_X_INV	Invalid operation (V)
FP_X_DZ	Division by 0 (Z)
FP_X_OFL	Overflow (O)
FP_X_UFL	Underflow (U)
FP_X_IMP	Imprecise exception (I)

- fpsetsticky() sets the value in the FPU exception flag field according to the sticky flag value and returns the previous value.
  The relation between the sticky flag and the FPSCR FPU exception flag field is the same as that of fpgetsticky().

NOTES

fpsetsticky() changes the value of the FPU exception flag field corresponding to every the sticky flag.
fpsetmask() changes the exception enable bit corresponding to every exception mask value.
The following mode are not available for rounding control by fpgetround() and fpsetround().

- FP_RP: A negative value is truncated and a positive value is rounded up.
- FP_RM: A positive value is truncated and a negative value is rounded up.

NAME

fpcheck, fpchecko - Detect a floating-point exception.

SYNOPSIS

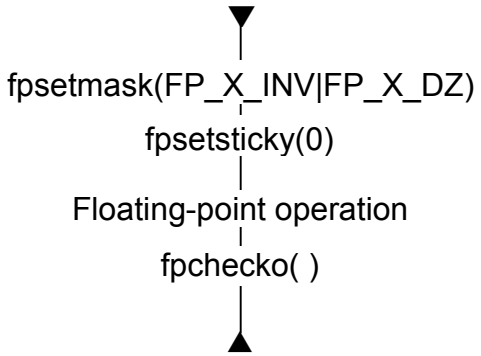
```
#include <ieeefp.h>

typedef enum {
        FP_RN =0,    /*   round nearest      */
        FP_RZ =1     /*   round zero (truncate)   */
    } fp_rnd;

#define fp_except int
#define FP_X_INV        0x10    /* invalid operation exception   */
#define FP_X_OFL        0x04    /* overflow exception            */
#define FP_X_UFL        0x02    /* underflow exception           */
#define FP_X_DZ         0x08    /* devided by zeros exception    */
#define FP_X_IMP        0x01    /* imprecise (loss of precision) */

void        fpcheck(fp_except flg);
void        fpchecko(void);
```

DESCRIPTION

These macros detect an occurrence status of a flowing-point exception that suppressing occurrence.
When the occurrence of a floating-point exception is detected, it results in a program error and the task is aborted.
fpchecko() detects whether an overflow occurs or not.
fpcheck() detects whether the exception specified in the parameter occurs or not. For detecting multiple exceptions simultaneously, specify the OR of exception factors.

EXAMPLE

The following is an example of suppressing the occurrence of an overflow and detecting the occurrence of an overflow after operations.



… Invalid operation and division by 0 are validated. (Overflow is invalidated.)

… Clear the factor flag bit.

… Check whether an overflow occurs or not. If an overflow occurs, this results in a program error.

THIS PAGE INTENTIONALLY LEFT BLANK.

APPENDIXES

APPENDIX A MACRO PARAMETERS

Macro name	Parameter 1	Parameter 2	Parameter 3	Parameter 4	Parameter 5	Parameter 6
rleas	tn					
queue	tn	fact				
exit						
abort	tn					
wait	ecb					
post	ecb	pcode				
susp	tn					
rsum	tn					
asusp						
arsum						
chap	tn	chgp				
sfact	tn	fact				
gfact	fact					
wrtmem	vaddr	data	size			
chkbmem	slot					
chktaer	slot					
mvmem	wno	daddr	saddr			
uspchk	usebyt	addr				
timer	id	tn	fact	t	cyt	
ctime	tn	fact				
delay	t					
stime	time					
gtime	time					
wake	id	tn	fact	time	cycle	
cwake	tn	fact				
rserv	n	para1	para2	para3	para4	para5
prsrv	n	para1	para2	para3	para4	para5
free	n	para1	para2	para3	para4	para5
pfree	n	para1	para2	para3	para4	para5
wdtset	msec					
getsysinfo	type	addr				
gettaskinfo	type	tn	addr			
gtkmem	tblno	caseno	offset	size	buf	
usrdhp	code	data	ndata			
usrel	type	class	retcode	errtype	erb	
save_env	env					
resume_env	env	val				
gettimebase	timebase					
TimebaseToSecs	timebase	tval				
atmswap	addr	data				
atmand	addr	data				
atmor	addr	data				
atmxor	addr	data				
atmadd	addr	data				
atmtas	addr	data				
atmcas	addr	data1	data2			

APPENDIX B CPMS ERROR HANDLING

(1/2)

No.	Error code	Error message	Error description	Fault category	Fault location	Termination	Recovery
1	03620000	Program error (Invalid Data Access)	Data access error	Software	TASK	TASK ABORT	Program correction
2	03660000	Program error (Data Access Protection)	Data access protect error	Software	TASK	TASK ABORT	Program correction
3	03600000	Program error (Data Page Fault)	Data access page fault	Software	TASK	TASK ABORT	Program correction
4	03420000	Program error (Invalid Inst. Access)	Instruction access error	Software	TASK	TASK ABORT	Program correction
5	03460000	Program error (Inst. Access Protection)	Instruction access protect error	Software	TASK	TASK ABORT	Program correction
6	03400000	Program error (Instruction Page Fault)	Instruction access page fault	Software	TASK	TASK ABORT	Program correction
7	03030000	Program error (Inst. Alignment Error)	Instruction alignment error	Software	TASK	TASK ABORT	Program correction
8	03080000	Program error (Privileged Instruction)	Privileged instruction error	Software	TASK	TASK ABORT	Program correction
9	03040000	Program error (Illegal Instruction)	Illegal instruction error	Software	TASK	TASK ABORT	Program correction
10	03390000	Program error (FP Program Error)	Floating-point calculation error	Software	TASK	TASK ABORT	Program correction
11	03470000	Program error (Data Alignment Error)	Data alignment error	Software	TASK	TASK ABORT	Program correction
12	05130000	Invalid macro	Undefined-macro issuance	Software	TASK	TASK ABORT	Program correction
13	05110000	Macro parameter error	Macro parameter abnormal	Software	TASK	TASK ABORT	Program correction
14	05C70000	WDT timeout error	Watchdog timer timeout	Software	TASK	—	Program correction
15	03B70000	Module error (Bus Target Abort)	Bus target abort	Hardware	I/O	—	Hardware replacement or program correction
16	05000000	Module error (Invalid Interrupt)	Invalid interrupt	Hardware	CMU	—	Hardware replacement
17	05000001	Module error (Undefined Invalid Interrupt)	Undefined invalid interrupt	Hardware	CMU	—	Hardware replacement
18	05000002	Module error (INTEVT Invalid Interrupt)	INTEVT invalid interrupt	Hardware	CMU	—	Battery replacement
19	0500F001	Module error (HERST Invalid Interrupt)	Serious fault invalid interrupt	Hardware	CMU	—	Battery replacement
20	0500F002	Module error (HERST2 Invalid Interrupt)	Serious fault invalid interrupt 2	Hardware	CMU	—	Hardware replacement
21	0500F003	Module error (BUERRSTAT Invalid Interrupt)	Bus error serious fault interrupt status invalid	Hardware	CMU	—	Hardware replacement
22	0500F006	Module error (MHPMCLG Invalid Interrupt)	Memory serious fault interrupt status invalid	Hardware	CMU	—	Hardware replacement
23	0500F007	Module error (ECC 2bit Master Invalid Interrupt)	Memory ECC 2-bit error serious fault status invalid	Hardware	CMU	—	Hardware replacement
24	0500F008	Module error (RERRMST Invalid Interrupt)	RERR interrupt status invalid	Hardware	CMU	—	Hardware replacement
25	0500C001	Module error (NINTR Invalid Interrupt)	NINT status invalid	Hardware	CMU	—	Hardware replacement
26	0500B001	Module error (PUINTR Invalid Interrupt)	PUINT status invalid	Hardware	CMU	—	Hardware replacement
27	05008001	Module error (PUINTC Invalid Interrupt)	PUINTC status invalid	Hardware	CMU, LPU	—	Hardware replacement
28	05005001	Module error (RINTR Invalid Interrupt)	RINT status invalid	Hardware	CMU	—	Hardware replacement
29	05003001	Module error (LV3 INTST Invalid Interrupt)	Level 3 interrupt status invalid	Hardware	CMU	—	Hardware replacement
30	05003002	Module error (RQI6 INF Invalid Interrupt)	RQI6 status invalid	Hardware	CMU	—	Hardware replacement
31	05001001	Module error (RQI3 INT Invalid Interrupt)	RQI3 status invalid	Hardware	CMU	—	Hardware replacement
32	05001002	Module error (RQI3 Link Invalid Interrupt)	RQI3 link status invalid	Hardware	CMU	—	Hardware replacement
33	05001003	Module error (RQI3 Module Invalid Interrupt)	RQI3 module status invalid	Hardware	CMU	—	Hardware replacement
34	0D010000	Module error (Memory Alarm)	Memory 1-bit error (solid)	Hardware	CMU	—	Hardware replacement
35	0D320000	Module error (Memory Error)	Memory error	Hardware	CMU, I/O	—	Hardware replacement
36	0D330000	Module error (Hardware WDT Timeout)	Hardware WDT timeout	Hardware	CMU, I/O	—	Hardware replacement
37	0D340000	Module error (Software WDT Timeout)	Software WDT timeout	Hardware	CMU, I/O	—	Hardware replacement or program correction
38	0D350000	Module error (RAM Sum Check Error)	RAM checksum error	Hardware	CMU, I/O	—	Hardware replacement or program correction
39	0D360000	Module error (ROM Sum Check Error)	ROM checksum error	Hardware	CMU, I/O	—	Hardware replacement
40	0D370000	Module error (Clock Stop Error)	Clock stop error	Hardware	CMU, I/O	—	Hardware replacement
41	0D380000	Module error (OS Clear Error)	OS clear error	Hardware	CMU, I/O	—	Program load

(2/2)

No.	Error code	Error message	Error description	Fault category	Fault location	Termination	Recovery
42	0D800000	Module error (TOD Error)	Backup clock error	Hardware	CMU, LPU	—	Hardware replacement
43	05A00000	Kernel warning	Kernel warning	Software	—	—	—
44	05A00001	Clock synchronization	Kernel warning	Software	—	—	—
45	05D00000	Kernel information	Kernel information	Software	—	—	—
46	0D810000	System down (BPU Error)	BPU error	Hardware	CMU	CMU STOP	Hardware replacement
47	03820000	System down (Memory Error)	Memory error	Hardware	CMU	CMU STOP	Hardware replacement
48	038A0000	System down (Memory Access Error)	Memory access error	Hardware	CMU	CMU STOP	Hardware replacement
49	038B0000	System down (Internal Bus Parity)	Internal bus parity error	Hardware	CMU	CMU STOP	Hardware replacement
50	038C0000	System down (System Bus Parity)	System bus parity error	Hardware	CMU	CMU STOP	Hardware replacement
51	038F0000	System down (Undefined Machine Check)	Undefined-machine check error	Hardware	CMU	CMU STOP	Hardware replacement
52	03620000	System down (Invalid Data Access)	Data access error	Software	CPMS	CMU STOP	Program correction
53	03660000	System down (Data Access Protection)	Data access protect error	Software	CPMS	CMU STOP	Program correction
54	03600000	System down (Data Page Fault)	Data access page fault	Software	CPMS	CMU STOP	Program correction
55	03420000	System down (Invalid Inst. Access)	Instruction access error	Software	CPMS	CMU STOP	Program correction
56	03460000	System down (Inst. Access Protection)	Instruction access protect error	Software	CPMS	CMU STOP	Program correction
57	03400000	System down (Instruction Page Fault)	Instruction access page fault	Software	CPMS	CMU STOP	Program correction
58	03030000	System down (Inst. Alignment Error)	Instruction alignment error	Software	CPMS	CMU STOP	Program correction
59	03040000	System down (Illegal Instruction)	Illegal instruction error	Software	CPMS	CMU STOP	Program correction
60	03380000	System down (FP Unavailable)	Floating-point unavailable exception	Software	CPMS	CMU STOP	Program correction
61	03390000	System down (FP System Down)	Floating-point calculation error	Software	CPMS	CMU STOP	Program correction
62	03470000	System down (Data Alignment Error)	Data alignment error	Software	CPMS	CMU STOP	Program correction
63	030F0000	System down (Illegal Exception)	Illegal exception	Software	CPMS	CMU STOP	Program correction
64	05700000	System down (System Error)	System failure (system error)	Software	CPMS	CMU STOP	Program correction
65	05800000	System down (Kernel Trap)	System failure (kernel trap)	Software	CPMS	CMU STOP	Program correction
66	03620000	ULSUB down (Invalid Data Access)	Data access error	Software	ULSUB	CMU STOP	Program correction
67	03660000	ULSUB down (Data Access Protection)	Data access protect error	Software	ULSUB	CMU STOP	Program correction
68	03600000	ULSUB down (Data Page Fault)	Data access page default	Software	ULSUB	CMU STOP	Program correction
69	03420000	ULSUB down (Invalid Inst. Access)	Instruction access error	Software	ULSUB	CMU STOP	Program correction
70	03460000	ULSUB down (Inst. Access Protection)	Instruction access protect error	Software	ULSUB	CMU STOP	Program correction
71	03400000	ULSUB down (Instruction Page Fault)	Instruction access page fault	Software	ULSUB	CMU STOP	Program correction
72	03030000	ULSUB down (Inst. Alignment Error)	Instruction alignment error	Software	ULSUB	CMU STOP	Program correction
73	03080000	ULSUB down (Privileged Instruction)	Privileged instruction error	Software	ULSUB	CMU STOP	Program correction
74	03040000	ULSUB down (Illegal Instruction)	Illegal instruction error	Software	ULSUB	CMU STOP	Program correction
75	03380000	ULSUB down (FP Unavailable)	Floating-point unavailable exception	Software	ULSUB	CMU STOP	Program correction
76	03390000	ULSUB down (FP System down)	Floating-point calculation error	Software	ULSUB	CMU STOP	Program correction
77	03470000	ULSUB down (Data Alignment Error)	Data alignment error	Software	ULSUB	CMU STOP	Program correction
78	030F0000	ULSUB down (Illegal Exception)	Illegal exception	Software	ULSUB	CMU STOP	Program correction
79	05140000	System down (ULSUB Stop)	System failure (built-in sub-stop)	Software	ULSUB	CMU STOP	Program correction
80	05F00000	Program error (ADT Error)	Memory access detection	Software	TASK	Log	Program correction
81	00000201	Message frame error	Message frame error	Software	NXACP	—	—
82	00000401	Buffer status	Buffer status report	Software	NXACP	—	—
83	00000501	Socket error	Socket error	Software	NXACP	—	—
84	00000601	Transfer memory address error	Transfer area duplication error	Software	TASK	—	Program correction

APPENDIX C BUILT-IN SUBROUTINE INPUT DATA

(1) CPES input data format (PRGEB)

	Name	Description
0	pge_form	LOG_FORM_PRGERR
4	pge_frsz	Data size after pge_ecd (in bytes)
8	pge_ecd	Error code
12	pge_tn	Task number
16	pge_gr0_b0	General register R0_BANK0
20	pge_gr1_b0	General register R0_BANK0
24	pge_gr2_b0	General register R0_BANK0
28	pge_gr3_b0	General register R0_BANK0
32	pge_gr4_b0	General register R0_BANK0
36	pge_gr5_b0	General register R0_BANK0
40	pge_gr6_b0	General register R0_BANK0
44	pge_gr7_b0	General register R0_BANK0
48	pge_gr8	General register R8
52	pge_gr9	General register R9
56	pge_gr10	General register R10
60	pge_gr11	General register R11
64	pge_gr12	General register R12
68	pge_gr13	General register R13
72	pge_gr14	General register R14
76	pge_gr15	General register R15
80	pge_pc	Program counter
84	pge_sr	Status register
88	pge_pr	Procedure register
92	pge_gbr	Global base register
96	pge_mach	Sum-of-products upper register
100	pge_macl	Sum-of-products lower register
104	pge_expevt	expevt register
108	pge_fadr	Fault Address
112	pge_fr0	Floating-point register FPR0_BANK0
116	pge_fr1	Floating-point register FPR1_BANK0
120	pge_fr2	Floating-point register FPR2_BANK0
124	pge_fr3	Floating-point register FPR3_BANK0
128	pge_fr4	Floating-point register FPR4_BANK0
132	pge_fr5	Floating-point register FPR5_BANK0
136	pge_fr6	Floating-point register FPR6_BANK0
140	pge_fr7	Floating-point register FPR7_BANK0
144	pge_fr8	Floating-point register FPR8_BANK0
148	pge_fr9	Floating-point register FPR9_BANK0
152	pge_fr10	Floating-point register FPR10_BANK0
156	pge_fr11	Floating-point register FPR11_BANK0
160	pge_fr12	Floating-point register FPR12_BANK0
164	pge_fr13	Floating-point register FPR13_BANK0
168	pge_fr14	Floating-point register FPR14_BANK0
172	pge_fr15	Floating-point register FPR15_BANK0
176	pge_fr16	Floating-point register FPR0_BANK1
180	pge_fr17	Floating-point register FPR1_BANK1
184	pge_fr18	Floating-point register FPR2_BANK1
188	pge_fr19	Floating-point register FPR3_BANK1
192	pge_fr20	Floating-point register FPR4_BANK1
196	pge_fr21	Floating-point register FPR5_BANK1
200	pge_fr22	Floating-point register FPR6_BANK1
204	pge_fr23	Floating-point register FPR7_BANK1
208	pge_fr24	Floating-point register FPR8_BANK1
212	pge_fr25	Floating-point register FPR9_BANK1
216	pge_fr26	Floating-point register FPR10_BANK1
220	pge_fr27	Floating-point register FPR11_BANK1
224	pge_fr28	Floating-point register FPR12_BANK1
228	pge_fr29	Floating-point register FPR13_BANK1
232	pge_fr30	Floating-point register FPR14_BANK1
236	pge_fr31	Floating-point register FPR15_BANK1
240	pge_fpscr	Floating-point status control register
244	pge_fpul	Floating-point communication register
248	pge_iarvn9	Contents of address 36 indicated by program counter
252	pge_iarvn8	Contents of address 32 indicated by program counter
256	pge_iarvn7	Contents of address 28 indicated by program counter
260	pge_iarvn6	Contents of address 24 indicated by program counter
264	pge_iarvn5	Contents of address 20 indicated by program counter
268	pge_iarvn4	Contents of address 16 indicated by program counter
272	pge_iarvn3	Contents of address 12 indicated by program counter
276	pge_iarvn2	Contents of address 8 indicated by program counter
280	pge_iarvn1	Contents of address 4 indicated by program counter
284	pge_iarv0	Contents of address indicated program counter
288	pge_iarv1	Contents of address +4 indicated by program counter

(2) IES input data format (IOERB)

	Name	Description
0	ioe_form	Format type (in this case, LOG_FORM_IOERR)
4	ioe_frsz	Data size after ioe_ecd (bytes)
8	ioe_ecd	Error code
12	ioe_uno	Unit No.
16	ioe_dev	Device No.
20	ioe_dva	Device address
24	ioe_ioec	Detailed error code
28	ioe_tn	Task No. (-1 is used if the task No. is invalid.)
32	ioe_data[110]	Detailed information of I/O error. (Detailed information depends on each I/O.)
472		

(3) EAS input data format (ADB)

	Name	Description
0	adb_logno	Error log No.
4	adb_timestamp	Time (host clock value)
8	adb_type	Significance type
12	adb_class	Failure detecting component class
16	adb_retcode	Return code from function when failure is detected
18	adb_errtype	Failure type (hardware/CPMS/other)
20	adb_flag	Error message flag (i.e., display suppression)
24	adb_site[16]	Site name
40	erb[118]	Error block (failure report data). The area size is fixed at 472 bytes but the effective data size depends on the format type. For details, see “Error Log Format.”
512	adb_dhpbuf[128]	DHP data (512 bytes)
1024		

(4) PCKS input data format (SVCEB)

	Name	Description
0	sve_form	Format type (in this case, LOG_FORM_PARAMERR)
4	sve_frsz	Data size after sve_ecd (bytes)
8	sve_ecd	Error code
12	sve_tn	Task No.
16	sve_svc	Macro ID
20	sve_epn	Error parameter No.
24	sve_p1	Macro instruction parameter 1
28	sve_p2	Macro instruction parameter 2
32	sve_p3	Macro instruction parameter 3
36	sve_p4	Macro instruction parameter 4
40	sve_p5	Macro instruction parameter 5
44	sve_p6	Macro instruction parameter 6
48	sve_p7	Macro instruction parameter 7

(5) MODES input data format (HARDEB)

	Name	Description
0	mde_form	Format type (in this case, LOG_FORM_MODULERR)
4	mde_frsz	Data size after mde_ecd (bytes)
8	mde_ecd	Error code
12	mde_slot	Slot No.
16	mde_msw0	Module status word 0 (-1 is used if this word is invalid.)
20	mde_msw1	Module status word 1 (-1 is used if this word is invalid.)
24	mde_data[112]	Detailed format of module error
472		

(6) ADTS input data format (ADTDB)

	Name	Description
0	adt_form	LOG_FORM_ADTERR
4	adt_frsz	Data size after adt_ecd (in bytes)
8	adt_ecd	Error code
12	adt_tn	Task number
16	adt_gr0	General register R0_BANK0
20	adt_gr1	General register R1_BANK0
24	adt_gr2	General register R2_BANK0
28	adt_gr3	General register R3_BANK0
32	adt_gr4	General register R4_BANK0
36	adt_gr5	General register R5_BANK0
40	adt_gr6	General register R6_BANK0
44	adt_gr7	General register R7_BANK0
48	adt_gr8	General register R8
52	adt_gr9	General register R9
56	adt_gr10	General register R10
60	adt_gr11	General register R11
64	adt_gr12	General register R12
68	adt_gr13	General register R13
72	adt_gr14	General register R14
76	adt_gr15	General register R15
80	adt_pc	Program counter
84	adt_sr	Status register
88	adt_pr	Procedure register
92	adt_gbr	Global base register
96	adt_mach	Sum-of-products upper register
100	adt_macl	Sum-of-products lower register
104	adt_expevt	expevt register
108	adt_fadr1	Fault Address1
112	adt_fadr2	Fault Address2
116	adt_bara	Break address register A
120	adt_bamra	Break address mask register A
124	adt_bbra	Break bus cycle register A
128	adt_basra	Break A SID register A
132	adt_bamrb	Break address register B
136	adt_bamrb	Break address mask register B
140	adt_bbrb	Break bus cycle register B
144	adt_basrb	Break A SID register B
148	adt_brcr	Break control register
152	adt_fr0	Floating-point register FPR0_BANK0
156	adt_fr1	Floating-point register FPR1_BANK0
160	adt_fr2	Floating-point register FPR2_BANK0
164	adt_fr3	Floating-point register FPR3_BANK0
168	adt_fr4	Floating-point register FPR4_BANK0
172	adt_fr5	Floating-point register FPR5_BANK0
176	adt_fr6	Floating-point register FPR6_BANK0
180	adt_fr7	Floating-point register FPR7_BANK0
184	adt_fr8	Floating-point register FPR8_BANK0
188	adt_fr9	Floating-point register FPR9_BANK0
192	adt_fr10	Floating-point register FPR10_BANK0
196	adt_fr11	Floating-point register FPR11_BANK0
200	adt_fr12	Floating-point register FPR12_BANK0
204	adt_fr13	Floating-point register FPR13_BANK0
208	adt_fr14	Floating-point register FPR14_BANK0
212	adt_fr15	Floating-point register FPR15_BANK0
216	adt_fr16	Floating-point register FPR0_BANK1
220	adt_fr17	Floating-point register FPR1_BANK1
224	adt_fr18	Floating-point register FPR2_BANK1
228	adt_fr19	Floating-point register FPR3_BANK1
232	adt_fr20	Floating-point register FPR4_BANK1
236	adt_fr21	Floating-point register FPR5_BANK1
240	adt_fr22	Floating-point register FPR6_BANK1
244	adt_fr23	Floating-point register FPR7_BANK1
248	adt_fr24	Floating-point register FPR8_BANK1
252	adt_fr25	Floating-point register FPR9_BANK1
256	adt_fr26	Floating-point register FPR10_BANK1
260	adt_fr27	Floating-point register FPR11_BANK1
264	adt_fr28	Floating-point register FPR12_BANK1
268	adt_fr29	Floating-point register FPR13_BANK1
272	adt_fr30	Floating-point register FPR14_BANK1
276	adt_fr31	Floating-point register FPR15_BANK1
280	adt_fpscr	Floating-point status control register
284	adt_fpul	Floating-point communication register
288	adt_iarvn9	Contents of address -36 indicated by program counter
292	adt_iarvn8	Contents of address -32 indicated by program counter
296	adt_iarvn7	Contents of address -28 indicated by program counter
300	adt_iarvn6	Contents of address -24 indicated by program counter
304	adt_iarvn5	Contents of address -20 indicated by program counter
308	adt_iarvn4	Contents of address -16 indicated by program counter
312	adt_iarvn3	Contents of address -12 indicated by program counter
316	adt_iarvn2	Contents of address -8 indicated by program counter
320	adt_iarvn1	Contents of address -4 indicated by program counter
324	adt_iarv0	Contents of address indicated program counter
328	adt_iarv1	Contents of address +4 indicated by program counter