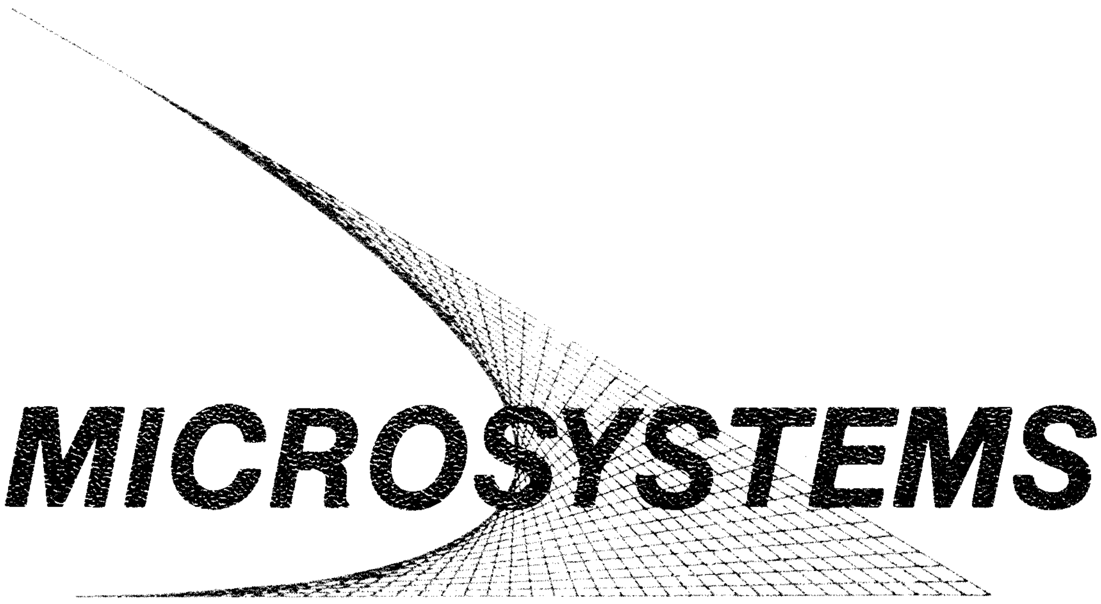
MOTOROLA

M68KRMS68K/D9

M68000 Family Real-Time Multitasking Software User's Manual



QUALITY • PEOPLE • PERFORMANCE

MOTOROLA

M68KRMS68K/D9

MARCH 1985

M68000-FAMILY
REAL-TIME MULTITASKING SOFTWARE
USER'S MANUAL

Any addendums to previous revisions of this manual have been incorporated in this revision.

Ninth Edition

Copyright 1985 by Motorola Inc.

Eighth Edition March 1984

MICROSYSTEMS

REVISION RECORD

M68KRMS68K/D9 - March 1985. Reflects the following software levels: VERSAdos 4.4 and Link 1.8. Adds support of the MC68020, VM04, VME/12, MVME115, MVME120, MVME121, MVME122, MVME123, and MVME128. New features: Supports the concept of real-time tasks to process events asynchronously and supports a priority-driven pre-emptive dispatcher. Several new directives have been added to support real-time tasks. Bit 7 of the attributes mask has been assigned as the real-time task attribute bit. All functions have been rewritten to increase processing speed.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont'd)

TABLE OF CONTENTS (cont'd)

TABLE OF CONTENTS (cont'd)

TABLE OF CONTENTS (cont'd)

CHAPTER 1

GENERAL INFORMATION

1.1 INTRODUCTION

The Motorola M68000 Family Real-Time Multitasking Software is a full-functioned kernel designed for use in a real-time multitasking environment based on a Motorola MC68000, MC68010, or MC68020 microprocessor. This software consists of a priority-driven scheduler/dispatcher to support the real-time multitasking environment and eight resource managers. Each resource manager supports a particular function such as, memory management, event management, and timer management. For the remainder of this manual, the M68000 Family Real-Time Multitasking Software will be referred to as RMS68K or the Executive and unless otherwise noted, MC68000 also applies to the MC68010 and MC68020.

This manual describes the functions of RMS68K and provides examples of how they may be used within an application. Chapter 1 explains some basic system concepts, such as multitasking, real-time, and priority-driven, as well as an overview of the functional partitioning of the Executive into resource managers. The remaining chapters describe each resource manager in detail including the theory of operation, data structures, directives, and examples of typical uses.

It is recommended that Chapter 1 be read to become familiar with the basic functionality provided by RMS68K. Thereafter, the user can go directly to the chapters describing the resource managers required by the application.

1.2 SYSTEM CONCEPTS

As stated in the introduction, RMS68K is a real-time multitasking, priority-driven kernel (or Executive). The following paragraphs will clarify the basic system concepts behind such words as "real-time", "multitasking", and "priority-driven", and show how the design of RMS68K supports these concepts.

A real-time system must respond to external events as they occur. An external event may be a user pressing a key on a keyboard, a thermometer indicating that the temperature within an observed environment has reached or surpassed a predefined threshold, or a camera indicating that an object has entered its field of vision. Because it is not known in advance at what time an external event will occur, a real-time system is said to execute asynchronously.

Unlike a batch system, where one operation is completed before a new operation is started, a real-time system may delay the completion of one operation so that another operation may be started, continued, or completed. When one operation is suspended so that another may execute, the first operation is said to have been pre-empted by the requirement to execute the second operation. This mechanism, where more than one operation may be in progress at any given time, is called concurrent processing. Even though only one operation can be executing at a given time, using a single central microprocessing unit, the concurrent processing mechanism of a real-time system makes it appear as though several operations are executing simultaneously.

A real-time application system can be broken down into several tasks. A task is a function (or operation) that can execute concurrently with other functions. A task can be written to process a single type of event, or it may process more than one type of event. A kernel that supports more than one task executing concurrently is called a multitasking kernel.

A task consists of some code representing the task's knowledge of how to do something (e.g., process keystrokes from a terminal), and some data representing the task's world view (e.g., a collection of characters since the last carriage return). This code (procedural knowledge) and data (declarative knowledge) is usually all the knowledge that a task needs to do its function. However, the multitasking kernel needs to maintain a collection of data on each task under its control, consisting of such elements as: the name of the task, how much memory currently belongs to this task, and the state of this tasks registers and program counter the last time it was running on the processor. This data is collected into a piece of memory called the Task Control Block (TCB), and is used by the kernel to control the execution of the task. The knowledge contained within the TCB is always available to the kernel, but it is usually not available to the task.

One piece of knowledge contained within the TCB that is of particular importance to a real-time system, is the priority of the task. The task's priority is a measure of the tasks importance relative to all the other tasks within the system and represents its "need to run" in a multitasking system where many tasks may be "ready to run" at any moment. Thus, in a multitasking system where multiple tasks are "ready to run", the kernel runs the highest priority task. This type of system is known as a priority-driven multitasking system.

One aspect of priority that is crucial to the behavior of a real-time system is the ability of the kernel to pre-empt a lower priority task when a higher priority task becomes ready. This is known as a priority-driven pre-emptive dispatch. For example, consider a system where a task of priority nine is currently running on the processor and two tasks of priority six and nineteen are both waiting on their respective events. If an event occurs that the priority six task was waiting on, it will be made ready but the Executive will return control to the priority nine task that was previously executing; the priority six task will not run until the priority nine task relinquishes control of the processor. However, if an event occurs that the priority nineteen task was waiting on, the Executive will pre-empt the priority nine task by saving its registers and program counter within its TCB, marking that task as being in the "ready to run" state and turning control of the processor over to the priority nineteen task.

RMS68K supports all the real-time system concepts previously introduced and therefore, may be described as a real-time multitasking kernel that supports the asynchronous processing of events and a priority-driven pre-emptive dispatcher. The way in which RMS68K supports these concepts is explained in paragraph 1.3.

1.3 OVERVIEW OF RMS68K

RMS68K consists of an inner kernel (or nucleus) that supports the priority-driven, multitasking environment, and eight resource managers to provide services such as memory management, event management, and semaphore management. Refer to paragraph 1.3.7 for a detailed description of the functional partitioning of RMS68K.

1.3.1 Resource Managers

Each resource manager consists of data structures and about five to seven RMS68K directives, each providing a specific service from the resource manager. One example of a resource manager is the RMS68K memory manager. Some of the data structures controlled by the memory manager are the free memory list that describes all memory not currently allocated to any task, and the task segment table that describes all memory currently assigned to a particular task. Some of the directives contained within the memory manager are:

Get me a segment of memory
Release a segment of memory back to the system
Declare a segment to be shareable
Attach me to a previously-declared shareable segment

1.3.2 Directives

An RMS68K directive consists of a name for identification by the user, a number for machine recognition, and a body of code to provide the service associated with that directive. In addition, most RMS68K directives require that the task provide additional information about the requested service and select between one or more options. This information is collected into a block of data known as a parameter block. For example, the "Get me a segment of memory" directive requires some additional information: the name of the segment, the logical address of the segment, and the type of memory requested (RAM or ROM). Some of the options tell the directive what action to take if the memory is not immediately available: return an error code, put the task into a WAIT state until the memory becomes available, or return the next largest piece of memory currently available.

The interface between tasks and RMS68K directives is the TRAP #1 instruction. Before executing the TRAP #1 instruction, the task loads the directive number into data register D0, and the address of the parameter block, if required, into address register A0. If the directive executed successfully, control returns to the task at the instruction following the TRAP #1 instruction with the Z bit of the condition codes set and data register D0 equal to 0. If the directive failed, execution returns to the instruction following the TRAP #1 with the Z bit clear and data register D0 containing an error code of 08DDEEEE; where 08 indicates that this error code is from the Executive, DD contains the original directive number, and EEEE contains a code describing the error condition. An example of this interface is:

```
MOVE.W    #DIRECTIVE_NUMBER,D0
LEA       PARAMETER_BLOCK,A0
TRAP      #1
BNE       DECODE_ERROR_IN_D0
```

Note that the condition codes returned by RMS68K support the BNE instruction to an error routine. It is strongly recommended that users follow all TRAP #1 instructions with a BNE instruction to an error routine, even if the error routine merely aborts the task (this makes debugging easier).

Some directives return information in register A0 or in registers A0 and A1. Except for SR, D0, A0, and A1, all registers are returned as they were before the directive call. When A0 and A1 are not used as return parameters, they are also preserved.

WARNING

ANY UNUSED OPTIONAL FIELD IN AN EXECUTIVE DIRECTIVE PARAMETER BLOCK SHOULD BE CLEARED TO ZERO BY THE ISSUING TASK BEFORE ISSUING THAT EXECUTIVE DIRECTIVE.

1.3.3 Sessions

RMS68K supports a concept called Sessions that is a boundary around a group of tasks. This boundary works in two ways:

a. It limits the scope of the tasks in its bounds.

b. It protects them from the activities of tasks in alien sessions.

Within the boundaries of a session, tasks may communicate freely and share resources (such as memory). Tasks may also create other tasks within the session, start them, stop them, or terminate them. They may affect the state of other tasks within the session by waking them up, changing their priority, or reading or writing to their memory. A task may monitor exceptions (within other tasks), or customize the session environment by adding directives to the Executive that only affect that session. However, these activities are not allowed between tasks belonging in different sessions.

The session protection mechanism can be applied to any system where tasks work in groups to achieve common sub-goals. Sessions are used to protect different users from each other in multiple user systems.

1.3.4 System and User Tasks

To coordinate the activities of different sessions and provide system wide resources, a class of super-tasks is required that can break through or cross over the session boundaries. Under RMS68K this class of super-tasks is called system tasks. Tasks bound within the perimeters of a session are called user tasks.

A system task can communicate with another task in any session, affect its state, read or write to its memory, or monitor its exceptions. A user task is restricted from influencing a task within another session.

A system task can only be created by another system task. To create a system task under VERSAdos, link the task with the "S" option and load it under User 0.

This two level system defines a hierarchy around which various software architectures can be designed. Both system and user tasks execute within the user hardware state of the M68000 Family microprocessor.

Session number $0000 provides a totally independent session from the remainder of the system. System tasks within session $0000 can affect tasks within other sessions. Session $0000 tasks are protected from system or user tasks within any other session; this allows a development system in which the online, real-time control to be totally protected from software development functions. Tasks are placed in session $0000 by including them within the boot file at SYSGEN time.

1.3.5 Real-Time Domain

To fulfill the performance requirements of real-time programming, RMS68K supports two domains of execution: real-time and non real-time. Tasks executing within the real-time domain are called real-time tasks and are subject to the following constraints:

a. The entire address space must be mapped so that all logical addresses are equivalent to their corresponding physical address (no Memory Management Unit (MMU) address translation).

b. They must use the internally generated 8-byte code as a target task interface (refer to paragraph 1.3.6), when executing RMS68K directives that refer to target tasks.

Other than these two constraints, the two domains are identical. All RMS68K directives function the same in either domain, except that RMS68K services directives faster in the real-time domain. Also, the domain of execution is totally independent from the system/user session constraint. Four combinations of the session/domain constraints exist:

DOMAIN CONSTRAINT	SESSION CONSTRAINT	TASK TYPE
Real-time	System	Real-time system task.
Real-time	User	Real-time user task.
Non real-time	System	Non real-time system task.
Non real-time	User	Non real-time user task.

There are two ways to create real-time tasks:

a. Describing the task to SYSGEN as a real-time task and including it in the boot file.

```
ATTRIB = 'RTIM'
```

b. Writing a position-independent task and describing it to the linker (using both the "P" and "R" attributes) as being both position-independent and real-time.

```
LINK
IN        TASK
ATTR      PR
END
```

1.3.6 Target Task Interface

About one third of RMS68K directives allow a task to refer to another task, called the target, by encoding an 8-byte task identifier (or task_id) into the target task field of the parameter block. The format of the task_id differs depending on whether the requesting task is a real-time task or not. However, it is not dependent on whether the target is a real-time task or a non real-time task.

The task_id format for a non real-time requesting task is a 4-byte taskname followed by a 4-byte session number. This target task interface protocol supports the session boundary constraint, so it is important to understand the rules of that protocol.

Target task interface protocol rules:

a. A taskname of 0 indicates the requesting task.

b. A session number of 0 indicates the home session for system tasks.

c. The session number is ignored for user tasks.

These rules are summed up in the following algorithm:

```
IF
(taskname = 0)
THEN ACCESS
     (name = requesting_task)
     (session = home_session);

ELSE IF
(session = 0) or
(requesting_task = user_task)
THEN ACCESS
     (name = taskname)
     (session = home_session);

ELSE ACCESS
     (name = taskname)
     (session = session_number);
```

The format of the task_id for a real-time requesting task is an 8-byte code generated by the Get_Task_ID directive. The input to Get_Task_ID is the taskname and session number of the target and the output is the target task's task_id in the proper format for the requestor. Thus, if a non real-time task calls Get_Task_ID, the output is identical to the input but for a real-time task, Get_Task_ID translates the taskname and session number into the 8-byte code.

The behavior of Get_Task_ID supports the migration of tasks between the real-time and non real-time domains. A task that calls Get_Task_ID to translate a target task's taskname and session number into the appropriate task_id format before using it within the target task field of an RMS68K directive, executes properly in either the real-time or non real-time domain. Also, because Get_Task_ID supports the session boundary constraint, real-time user tasks are restricted to accessing target tasks within their own sessions the same as non real-time user tasks.

In either domain, a task_id of 0 indicates that the target task is the requesting task.

1

1.3.7 RMS68K Functional Partitioning

The functional partitioning of RMS68K is:

- Nucleus
- Resource Managers
  - Event Manager
  - Memory Manager
  - Task Manager
  - Time Manager
  - Semaphore Manager
  - Trap Server Manager
  - Exception Monitor Manager
  - Exception Manager

Figure 1-1 describes the RMS68K partitioning. An overview of each functional partition within RMS68K is given in the following paragraphs.

```
User
Level               ------         ------         ------
                   | Task |       | Task |       | Task |
                   |  A   |       |  B   |       |  C   |
                    ------         ------         ------
                      \              |              /
                       \             |             /
                        \            |            /
                         \           |           /
Supervisor                \          |          /
Level                      \         |         /
                         -----------------------------
         ---------      |                             |      ---------
        | Event   |     |          NUCLEUS            |     |Exception|
        | Manager |-----|                             |-----|Manager  |
        |         |     |                             |     |         |
         ---------     / -----------------------------  \    ---------
                      / |      |      |        |        \
                     /  |      |      |        |         \
                    /   |      |      |        |          \
           ---------    |  ---------  |   ---------     ---------
          | Memory  |   | | Time    | |  |Trap     |   |Exception|
          | Manager |   | | Manager | |  |Server   |   |Monitor  |
          |         |   | |         | |  |Manager  |   |Manager  |
           ---------    |  ---------  |   ---------     ---------
                        |             |
                 ----------        ---------
                | Task     |      |Semaphore|
                | Manager  |      |Manager  |
                |          |      |         |
                 ----------        ---------
```

FIGURE 1-1. RMS68K Partitioning

1.3.7.1 Nucleus. The nucleus is the collection of modules required to support the real-time multitasking environment and contains the priority-driven pre-emptive dispatcher, the ready module to schedule tasks to run, the TRAP #1 handler to enable tasks to request services from resource managers, and the TRAP #0 handler to provide a similar interface to device drivers.

The nucleus also contains a module to schedule and run background jobs for drivers. A background job is a driver activity that executes with interrupts enabled to do post-interrupt processing of data latched during an Interrupt Service Routine (ISR). Background activity supports the principle that masking interrupts degrades the ability of a real-time system to respond to external stimuli and should be minimized. (For more information on background processing refer to the Guide to Writing Device Drivers for VERSAdos manual.)

1.3.7.2 Event Manager. The event manager provides sophisticated inter-task communication. An event is a message passed from one task to another and the mailbox mechanism is called the Asynchronous Service Queue (ASQ). The event manager supports the passing of variable length messages up to a maximum of 254 bytes and either the passing of a pointer to a message or the moving of the message from the sender's address space to the receiver's space.

The ASQ supports the receipt of events in two domains:

a. Synchronous

b. Asynchronous

In the synchronous domain, the task informs the event manager when it is ready to receive an event using the GET EVENT directive. If there is an event pending in the ASQ, the event manager moves the event to the task's receiving buffer and returns to the task. Otherwise, the task is put into a WAIT state until an event arrives.

In the asynchronous domain, the task may execute with an Asynchronous Service Routine (ASR) enabled. (This is analogous to a task connected to an ISR running with interrupts enabled.) When an event is sent to a task whose ASR is enabled, the task is interrupted out of its normal user level code and dispatched to its ASR. Once done executing the ASR, it returns to the user level code at the point at which it was interrupted.

Some of the services provided by the event manager are:

a. Get an Asynchronous Service Queue (ASQ).

b. Queue an event to a target task's ASQ.

c. Get an event in the synchronous mode.

d. Enable or disable asynchronous event processing.

e. Return to user level code from the asynchronous event processing.

1.3.7.3 Memory Manager. RMS68K supports the concept of segmented (non-paged) memory. A segment of memory must be contiguous, but the logical address visible to the task is not necessarily equal to the physical address visible to the hardware (MMU systems). A segment may be restricted to one task, visible to any task within a session, or it may be visible to any task within the system. A particular task may be attached to zero to four segments at any moment.

Segments may be RAM, ROM, or memory mapped I/O (hardware devices). The memory manager organizes all RAM and ROM memory into partitions, each with a partition number and a partition type. Partitions support different types of memory boards within a system. The partition number names the partitions for reference purposes and the partition type groups similar partitions of memory into classes. This partition number and partition type convention allows a task to request a segment from a particular partition number, from any partition of a specific type, or from the default number or type specified at initialization.

Some of the services provided by the memory manager are:

a. Get a segment from a partition.

b. Transfer a segment to a target task.

c. Detach from a segment and return it to its partition.

d. Declare a segment shareable.

e. Attach to a shareable segment.

1.3.7.4 Task Manager. The task manager provides the following services:

a. Create a task.

b. Start a task.

c. Stop a task.

d. Terminate or abort a task.

e. Task synchronization

f. Task queries

The task manager provides primitive inter-task synchronization through the SUSPEND/RESUME and WAIT/WAKEUP pairs of directives. These two pairs of directives do the same basic function, i.e., allow one task to wait for a signal from another task or device driver. The difference is that the WAIT/WAKEUP pair supports a one deep buffer for that signal, so that the signalling task can send the signal (wakeup the target) before the waiting task indicates its desire to wait. Whereas the SUSPEND/RESUME pair requires that the SUSPEND directive (wait on the signal) precede the RESUME directive (signal the target).

The RELINQUISH directive allows a task to yield voluntarily its control of the processor to another task of equal or slightly lower priority.

The task query directives allow a task to access information about another task within the system. This function is supported by four directives:

a.	Get Task Attributes	Returns the user number of the target task and its attributes (system task, position independent task, task has claimed its own exception vectors).
b.	Get Task Information	Returns a copy of the target task's TCB.
c.	Get Task_ID	Translates a taskname and session number into a task_id.
d.	Get Taskname	Translates a task_id into a taskname and session number.

1.3.7.5 Time Manager. Another resource that RMS68K manages for the system is time. The time manager notifies tasks of the expiration of pre-selected time intervals and supports two types of time:

a. Elapsed time (wake me up in fifteen minutes)

b. Calendar time (current time is 4:34 PM, January 4th, 1985)

The two directives supporting elapsed time are:

a. Delay

b. Request Periodic Activation

Both directives support the concept of "Wake me up in 100 milliseconds". The main difference is that the DELAY directive is a one time request, whereas the REQUEST PERIODIC ACTIVATION directive usually involves a request such as, "Wake me up every 100 milliseconds from now on".

The two directives supporting calendar time are:

a. Set Date and Time

b. Get Date and Time

1.3.7.6 Semaphore Manager. RMS68K provides sophisticated task synchronization via the Semaphore Manager. This manager supports three types of semaphores: a binary semaphore, a counting semaphore, and a broadcast semaphore. Most of the differences are concerned with such details as, "who can initialize the semaphore count" and "what happens when someone tries to attach to this semaphore before it is created". These details support the use of semaphores in specific environments; one task controlling a resource pool or two or more tasks sharing a critical section of code.

Some of the services provided by the semaphore manager are:

a. Create a semaphore
b. Attach to an existing semaphore
c. Wait on a semaphore
d. Signal a semaphore
e. Detach from a semaphore

1.3.7.7 Trap Server Manager. The Trap Server Manager supports the special requirements of protected and privileged operating system tasks such as, an I/O or file management system, and may be used by application programs to add common functions or operating system extensions (networking, GKS graphics server).

The trap server manager is a layer of additional functionality immediately above the event manager. This manager converts a TRAP #N instruction executed by a task, into an event sent to the trap server associated with that particular trap number. In addition, the server manager allows the trap server to exercise a certain amount of control over those tasks requesting its service.

One aspect of this control is the ability to regulate when the events generated by tasks requesting service are allowed to enter its ASQ. Typically, when a trap server is servicing one request, its ASQ will be disabled to additional trap requests while remaining enabled for communication with other operating system tasks or device drivers. Any task requesting service during this period, will be placed on a queue awaiting an indication from the trap server that it is ready to handle another request.

In addition to being able to indicate when it is ready to service requests, the trap server manager also allows the server to regulate when a requesting task is released from the server's control and in what state it should be on release. After a task executes a TRAP #N instruction, the trap manager places it into a state of waiting for acknowledgement from the server. It will not run until the server releases it via the ACKNOWLEDGE SERVICE REQUEST directive.

Some of the services provided by the trap server manager are:

a. Establish a trap server.

b. Acknowledge a request and release the task from server control.

c. Enable the server to handle additional trap requests even though a previous one has not been acknowledged.

d. Deallocate a trap server.

1.3.7.8 Exception Monitor Manager. The Exception Monitor Manager allows one task to observe and control the behavior of one or more target tasks by declaring itself to be an exception monitor for those tasks and to indicate which exceptions (bus errors, divide by zero faults, trap instructions), it is interested in observing. If a target task causes one of these exceptions to occur, the exception monitor manager will freeze the state of the target task and queue an event to the exception monitor describing the exception.

On notification that the target task has caused an exception, the exception monitor can read the state of the target task (registers, status register, program counter), change the state of the target task, read or write to the target task's code or data space, and enable the task to run, either freely or in a trace mode (execute one instruction, run until a memory location is changed, or run until a memory location equals a specific value).

Exception monitor tasks may be used to debug other tasks or to increase system security by observing and reporting on unusual task behavior.

Some of the services provided by the exception monitor manager are:

a. Establish a connection between an exception monitor and a target task.

b. Allow the exception monitor to indicate which exceptions it wants to monitor.

c. Run the target task under the control of the exception monitor.

d. Read the state of the target task.

e. Change the state of the target task.

f. Detach a target task from its exception monitor.

1.3.7.9 Exception Manager. The exception manager provides a set of directives for managing exceptions. It allows a user task to dynamically configure an ISR and to simulate a hardware interrupt.

It also allows the user task to claim any of the trap vectors not already used by RMS68K (TRAPS 0 and 1), or to claim any of the exception vectors (bus error, divide by zero, privilege error). If a task causes a claimed exception to occur, or executes a claimed trap instruction, the exception manager dispatches the task to its exception handler with information about the exception on the user stack. The default for unclaimed exceptions is to abort the task and queue an event to the task's monitor indicating the exception that caused the exception manager to take this action.

1.4 DATA STRUCTURES

The data structures controlled by RMS68K are:

ASQ	Asynchronous Service Queue Mailbox mechanism supporting the queuing of events between tasks.
FML	Free Memory List Doubly linked list of nodes describing current status of free memory within a RAM partition.
GST	Global Segment Table Array of segment descriptors for all currently defined shareable segments within the system.
IOV	I/O Vector Table Array of descriptors for all tasks currently claiming interrupts via the CISR directive.
MEMMAP	Memory Map Table Array of partition descriptors for all RAM or ROM partitions within the system.
PAT	Periodic Activation Table Linked list of nodes describing all current demands for elapsed time notification.
SYSPAR	System Parameter Table Unstructured list of miscellaneous system parameters.
TCB	Task Control Block Unstructured list of miscellaneous task state information.

TIOT	Trap Instruction Owner Table Array of descriptors for currently defined trap servers indexed by trap number.
TRC	System Trace Table Circular queue of traced events describing system history.
TST	Task Segment Table Array of segment descriptors for all segments currently accessible to a task.
UDR	User Defined Directive Table Array of descriptors indexed by directive number for all directives currently defined by tasks via the CDIR directive.
UST	User Semaphore Table Array of semaphore descriptors indexed by semaphore key for all currently defined semaphores created by the CRSEM or ATSEM directives.

1.5 HARDWARE REQUIREMENTS

The hardware requirements for an RMS68K-based application system are:

M68000 Family microprocessor
Adequate memory
Optional real-time clock

The amount of memory required varies from one system to another, depending on the system environment, user-supplied code and data, and the RMS68K functions configured in the user system. The maximum memory requirement for the entire RMS68K is:

Kernel	18.0Kb code 3.0Kb data
Channel management routines	2.0Kb code .25Kb data per channel

If the system is to use the time manager functions, a timer "tick" function must be provided. This can be done with a software tick mechanism that the user configures as part of RMS68K, or with a hardware timer device such as the Motorola MC6840 Programmable Timer Module.

The requirements listed above can be satisfied through use of the Motorola VERSAmodule Monoboard Microcomputer, VMEmodule Monoboard Microcomputer, VMC68/C, VME/10, VME/12, EXORmacs, or a user-designed M68000 Family microprocessor based board.

1.6 RELATED DOCUMENTATION

The following publications may provide additional helpful information. If not shipped with this product, they may be obtained from Motorola's Literature Distribution Center, 616 West 24th Street, Tempe, Az 85282; telephone (602) 994-6561.

DOCUMENT TITLE	MOTOROLA PUBLICATION NUMBER
System Generation Facility User's Manual	M68KSYSGEN
VERSAdos Messages Reference Manual	M68KVMSG
M68000 Family VERSAdos System Facilities Reference Manual	M68KVSF
VERSAdos to VME Hardware and Software Configuration User's Manual	MVMEVDOS
VERSAdos Data Management Services and Program Loader User's Manual	RMS68KIO
M68000 Family Linkage Editor User's Manual	M68KLINK
Guide to Writing Device Drivers for VERSAdos	M68KDRVGD
M68000 16/32-Bit Microprocessor Programmer's Reference Manual	M68000UM

CHAPTER 2

EVENT MANAGER

2.1 OVERVIEW

The Event Manager is the facility within RMS68K that enables tasks to communicate declarative knowledge to other tasks. The event manager may also be known as: inter-task communication, inter-process communication, or message passing. Declarative knowledge has two facets: factual knowledge (the sky is blue), and temporal or event knowledge (the sun just went down). The RMS68K event manager supports the communication of both types of declarative knowledge.

Other features of the event manager are:

a. Operates in either a synchronous or an asynchronous mode.

b. Supports two modes to communicate declarative knowledge:

   1. Move the data.

   2. Pass a pointer to the data.

c. Supports variable length messages.

d. Automatically queues messages if the receiving task is not ready to receive.

e. Supports task to task communication or driver to task communication.

f. High performance.

2.2 THEORY OF OPERATION

2.2.1 Events

An Event is a string of bytes used to communicate knowledge of a fact or occurrence between tasks. An event has three fields:

a. 1 byte event length field (N)	Supports variable length events.
b. 1 byte event code field	Differentiates types of events.
c. Optional event content field (N-2 bytes)	Communicates additional factual knowledge.

The following examples show the power and flexibility of events for communicating declarative knowledge.

EXAMPLE 1: A task controlling a camera communicating knowledge to a task that is controlling a robot arm.

The camera scans a conveyor belt for crates. It knows how to recognize three types of crates: large crates, medium crates, and small crates. When the camera observes a crate moving down the conveyor belt, it must determine the type of crate and communicate that knowledge to the robot arm. The tasks could establish an agreement that a small crate is signalled by a code $20 event, a medium crate by a code $30 event, and a large one by a code $40 event. Thus, a simple event consisting of only the length and code field would fulfill the requirements of the system.

EXAMPLE 2: The same system at a later stage of development where the camera not only recognizes crates, but can also determine the exact dimensions.

For this system, a single byte is not enough to differentiate between crates. The two tasks would need to define an event such as:

Code $20 means a crate is coming down the conveyor belt.

The event is 8 bytes long and the 6 bytes of the content field include three 2-byte sub-fields indicating the length, width, and depth of the crate in centimeters.

If the system needed to recognize other objects with different attributes, more event codes would be defined. The code field would differentiate between objects, and the content field would describe the objects attributes.

EXAMPLE 3: A vision system designed to recognize printed text.

The task controlling the camera would convert a bit map representing the page of text into a variable length string of ASCII characters (maximum length 5000). This string would be passed to another task that analyzes or processes the ASCII string. To support the requirement to pass large amounts of data via events, the camera task would assemble the ASCII string in a shared data segment and send an event containing a pointer to the ASCII string within the data segment.

Several of the event codes are used by RMS68K for specific purposes such as: notifying a server that a task has executed a trap instruction (code - $07), or notifying an exception monitor that a task under its control has taken an exception (code = $08). The only event codes a task cannot use are the server event code ($07) and event codes in the range $80 to $FF. However, to remain compatible with future enhancements of RMS68K, it is recommended to use event codes between $20 and $7F inclusive.

A code $03 event is treated in a special way by RMS68K; the task_id of the sending task is inserted immediately after the length field to inform the receiver of the sender's identity. This 8-byte task identification is in the proper format for the receiver to use, i.e., if the receiver is a real-time task, RMS68K inserts the internal 8-byte code, otherwise it inserts the taskname and session number.

2.2.2 Asynchronous Service Queue (ASQ)

The data structure supporting the queueing of events between tasks is called the ASQ. The ASQ consists of a control portion, a variable length data portion for temporary storage of events, and an overflow portion to support the high performance storage and retrieval of events. In other systems, the function of the ASQ is done by a data structure called a mailbox.

A task must request the event manager to allocate it an ASQ before it can receive events. The task can specify options such as: the size of the longest event it will receive, how much storage is necessary for the temporary storage of events, and whether the task intends to process events in a synchronous or asynchronous mode. The established ASQ is identified with the task and supports a many to one communication path; many tasks may queue events to an ASQ, but only one task may receive those events.

Even though the ASQ is identified with a particular task, it is not directly accessible by the task. In other words, the task does not process an event within the ASQ; the event must be moved from the ASQ into a receiving buffer within the task's address space. The task may choose to define a default receiving buffer when the ASQ is allocated. The knowledge of a default receiving buffer enables the system under most circumstances, to move the event directly from the sender's domain into the receiver's, bypassing the ASQ entirely.

The event manager contains a directive, SETASQ, that allows a task to dynamically enable or disable various facets of its ASQ. Thus, a task may inform the event manager that it does not want to receive events at the current time so the event manager rejects any such messages, informing the sender that the receiver's ASQ is disabled. Later the task may inform the event manager that it is now ready to receive events. The task may also enable or disable its default receive buffer or inform the event manager that it is ready or not ready to receive messages asynchronously.

2.2.3 Synchronous and Asynchronous Modes

The Synchronous Mode of inter-task communication is characterized by the ability of the receiving task to control the timing of event processing. In the Asynchronous Mode, the timing of event processing is determined by the behavior of the sender. In other words, in the synchronous mode the receiving task informs the event manager when it is ready to process an event, but in the asynchronous mode the receiver must be ready at any time to get "interrupted" to process an event. RMS68K supports inter-task communication in both the synchronous and asynchronous modes.

2.2.3.1 Synchronous Mode. The Synchronous Mode of inter-task communication is supported by two RMS68K directives, GTEVNT (Get an Event) and RDEVNT (Read an Event). Both directives eventually return control to the calling program at the instruction following the directive call. However, RDEVNT returns immediately, regardless of whether an event was present within the ASQ, whereas GTEVNT waits on an empty ASQ until an event arrives before returning control to the task. A task executing a GTEVNT directive is telling the event manager "I am ready to receive an event and I am willing to wait until one arrives" while a task executing a RDEVNT is saying "If there is an event in my ASQ, give it to me; if not, let me know". The first case is analogous to calling a get-character subroutine, while the latter case is analogous to polling an I/O port for an incoming character.

After executing a GTEVNT call, the event is present within the task's receiving buffer. The RDEVNT directive has two return conditions:

a. If there was at least one event present within the ASQ before the RDEVNT call, the oldest event within the ASQ will have been moved to the receive buffer.

b. If the ASQ was empty, the receive buffer contains a "null" event of length = \$00 and code = \$00, indicating an empty ASQ.

2.2.3.2 Asynchronous Mode. Any system that supports the Asynchronous Mode of inter-task communication must be able to maintain at least two "levels" of task execution. Level 1 is the level at which the task is executing its normal inline code, and level 2 is the task's event handling code. The level 2 code is dispatched asynchronously on event arrival, and since the level 2 code must eventually "return" to the level 1 code, the entire level 1 state must be saved before dispatching the level 2 code. (This is analogous to a processor that automatically saves the entire processor state before responding to an interrupt by dispatching an ISR.)

The typical requirements for supporting asynchronous inter-task communication are:

a. A task should be executing its normal or level 1 code.

b. A second task queues an event to this task.

c. The level 1 state of this task is saved somewhere (registers, status register, program counter), and the task is dispatched to its level 2 event handler routine.

d. The event handler routine processes the event, possibly passing some information about the event back to the level 1 state.

e. The event handler signals that it is finished processing the event.

f. The level 1 state is restored and level 1 execution is resumed at the point it was interrupted by the event arrival.

A more sophisticated variation on this scenario involves the level 2 event handler getting interrupted by the arrival of another event; its state is saved and a level 3 event handler is dispatched. This nesting of event handlers could continue to an arbitrary level "n". RMS68K's event manager supports both asynchronous scenarios.

Within RSM68K the event handler is called an Asynchronous Service Routine (ASR) and is declared when the asynchronous service queue (ASQ) is allocated. The ASR may be disabled or enabled at any time via the SETASQ directive. Asynchronous event processing requirements are fulfilled by the RMS68K event manager in the following way:

a. The task is executing its normal level 1 code and its ASR is enabled.

b. A second task queues an event to this task.

c. This task's level 1 state is saved on its stack and the task is dispatched to its level 2 ASR.

d. The ASR processes the event possibly passing some information about the event back to the level 1 state on the stack.

e. The event handler signals that it is finished processing the event via a RTEVNT (return from event) directive call to the event manager.

f. The level 1 state is restored by the event manager and the level 1 execution is restored at the point it was interrupted by the event.

On dispatch to the ASR, the ASR is automatically disabled by the event manager to guard against unwanted nesting of event interrupts. However, an ASR that is prepared for nesting can enable its ASR via the GETASQ directive. Thus, RMS68K also supports the more sophisticated case where ASRs may be nested to an arbitrary level "n".

If the default receive buffer was enabled, the event is present within the default receive buffer on dispatch to the ASR. Otherwise, the ASR needs to read the event into a buffer via the RDEVNT directive. On dispatch to the ASR, the state of the previous level of the task is present on the task's stack. Figure 2-1 shows the user stack format on entering an ASR.

```
                    =====================================
USP ---------›          D0 at moment of interrupt        USP
                        D1 at moment of interrupt        USP  +4
                                    .
                                    .
                                    .
                        D6 at moment of interrupt        USP  +24
                        D7 at moment of interrupt        USP  +28
                        A0 at moment of interrupt        USP  +32
                        A1 at moment of interrupt        USP  +36
                                    .
                                    .
                                    .
                        A5 at moment of interrupt        USP  +52
                        A6 at moment of interrupt        USP  +56
                        SR at moment of interrupt        USP  +60
                        PC at moment of interrupt        USP  +62
                    =====================================
```

FIGURE 2-1. User Stack on Entering an ASR (Register Stacking Feature Enabled)

The state of the previous level is also present within the ASR's registers D0 to D7 and A0 to A6. If the ASR needs to communicate with the previous level, it can change the state of one or more of that level's registers on the stack. When the ASR executes the RTEVNT trap call, the state of the ASR is lost and the state of the previous level is restored. Note that the ASR must ensure that its stack pointer is equal to the value it possessed immediately after ASR dispatch for proper functioning of the RTEVNT directive.

An optional mode of dispatch to the ASR can be selected via the GTASQ directive called "register stacking disabled". In this mode the previous level's state (D0 to D7 and A0 to A6) is present within the ASR's registers on ASR dispatch, but are not present on the task's stack.

```
                    =====================================
USP --------›       SR at moment of interrupt            USP
                    PC at moment of interrupt            USP +2
                    =====================================
```

FIGURE 2-2. User Stack on Entering an ASR (Register Stacking Feature Disabled)

In this mode the ASR must be careful not to destroy the state of the previous level contained within its registers. An RTR instruction restores the state of the previous level from the status register and program counter on the stack.

A problem occurs when a task is processing events in the asynchronous mode and the normal level 1 code runs out of things to do. It could go into a loop waiting for a signal from its ASR that another event has been processed. This approach, while feasible, wastes processing resources in a multitasking environment. The event manager solves this problem with the WTEVNT (Wait For Event) directive.

The WTEVNT directive is similar to the synchronous mode GTEVNT directive; it informs the event manager that the task cannot proceed until an event arrives. However, where the synchronous mode GTEVNT directive causes the task to be dispatched to the instruction immediately following the GTEVNT call, the asynchronous mode WTEVNT directive causes dispatch to the ASR on event arrival at which point the asynchronous processing of events starts up again.

One more aspect of asynchronous event processing is alternate ASRs. So far the ASR was assumed to be the default ASR declared via the GTASQ directive. However, a task may have an unlimited number of ASRs in addition to the default one, called alternate ASRs. A possible use for alternate ASRs would be to establish a different ASR for each type of event the task can process. To dispatch a task to an alternate ASR, the sending task must know the logical address of the alternate ASR. This address is then included in the QEVNT (Queue Event) parameter block and a bit is set indicating that this event should be dispatched to the alternate ASR at that address.

The asynchronous mode of event handling is powerful and flexible and many sophisticated asynchronous systems can be built around these primitives. However, many systems are synchronous by nature and those systems can be built faster and easier with the GTEVNT directive. The system designer should carefully consider the trade-offs between a synchronous and an asynchronous design.

2.2.4 Default Receive Buffer

The Default Receive Buffer is a buffer for the receipt of events that a task can declare via the GTASQ directive and may disable or enable via the SETASQ directive. The default receive buffer increases the performance of the event manager in the following way:

a. The event manager can translate the logical address of the default buffer into its corresponding physical address once, at initialization time, instead of every time a GTEVNT or RDEVNT directive is executed.

b. The presence of an enabled default buffer eliminates the requirement for an ASR to execute a RDEVNT because the event manager automatically moves the event into the default receive buffer before dispatching the asynchronous task to its ASR.

c. Usually, the presence of an enabled default receive buffer allows the event manager to move the event directly from the senders domain to the receivers, bypassing the ASQ's temporary storage buffer entirely. Thus, the Executive only needs to move the event once, instead of twice.

The event manager never writes a new event to the default receive buffer before the task has finished processing the current event. The user must notify the event manager that the buffer is no longer being used by:

a. Executing a GTEVNT directive.

b. Executing a RDEVNT directive.

c. Enabling the ASR by:

   1. Executing a SETASQ directive with the enable ASR bit set.

   2. Executing a WTEVNT directive.

   3. Executing a RTEVNT directive with the enable ASR bit set

2.2.5 Directive Summary

The directives contained within the event manager are:

GTASQ	Allocate an ASQ for the target task.
SETASQ	A task enables or disables its ASQ, ASR, and/or default receive buffer.
QEVNT	An event is sent to the target task.
GTEVNT	A task moves itself into the GETTING EVENT state until an event arrives at which time the task is dispatched to the instruction following the GTEVNT directive call.
RDEVNT	The next event in the requesting task's ASQ is moved into the task's receive buffer.
WTEVNT	A task moves itself into the WAIT FOR EVENT state until an event arrives at which time the task is dispatched to its ASR.
RTEVNT	Return to the point of task interruption on completion of that task's ASR processing.
DEASQ	Deallocate the requesting task's ASQ.

2.3 DATA STRUCTURES

The only data structure managed by the event manager is the ASQ. The ASQ is a task's mailbox and contains a control portion, a circular queue for the temporary storage of events, and an overflow area.

The fields within the control portion are:

ASQ (4 bytes)	Block ID Each ASQ begins with '!ASQ" to allow consistency checking and ease of dump reading.
ASQASR (4 bytes)	Default service address Contains the logical address of the default asynchronous service routine.
ASQXFR (4 bytes)	Current service address When a dispatch to an ASR is pending, this field contains the ASR's logical address. (This could be the default ASR defined in the GTASQ directive, or an alternate ASR specified in the QEVNT directive.)
ASQDBUF (4 bytes)	Physical address of default receive area defined via GTASQ directive.
ASQAOBUF (4 bytes)	Physical address of buffer pointed to by A0 if task does a GTEVNT when the ASQ is empty and default buffer is not enabled.
ASQBOT (4 bytes)	Queue beginning address Contains the physical address of the beginning of the ASQ event storage area.
ASQTOP (4 bytes)	Queue ending address Contains the physical address of the end of the ASQ event storage area (beginning of overflow area).
ASQGET (4 bytes)	Get pointer Contains the physical address of the oldest event within the ASQ.
ASQPUT (4 bytes)	Put pointer Contains the physical address of the next available byte for an incoming event.
ASQSTATE (2 bytes)	Contains the state variable for the ASQ and the switching mode variables. Bits 15-6 Reserved.

Bits 5-3 contain the state variable. Six ASQ states are defined:

BITS 5-3	ASQ STATE	MEANING
000	RQ_DIS	ASR and ASQ are both disabled.
001	Q_EN	ASR is disabled and ASQ is enabled.
010	R_EN	ASR is enabled and ASQ is disabled.
011	RQ_EN	Both the ASR and ASQ are enabled.
100	WT_EN	Task executed a WTEVNT directive when the ASQ was empty. Task is waiting for an event.
101	GT_EN	Task executed a GTEVNT directive when the ASQ was empty. Task is getting an event.

Bits 2 and 1 contain the ASQ switching mode bits (bits defining ASQ parameters that are critical to real-time performance of the event manager).

Bit 2	DBUF_EN	Defines whether the default buffer is enabled (1) or disabled (0).
Bit 1	ASQ_MT	Defines whether the ASQ is empty (1) or not (0).
Bit 0	Reserved	

ASQSTMOD (2 bytes) Contains the static mode variables for the ASQ (bits defining ASQ parameters not critical to real-time performance of the event manager).

Bits 15-6	Reserved	
Bit 5	ASQS_RNV	ASR is not valid. If set, this bit indicates that an ASR was not defined when the ASQ was allocated. Otherwise, the ASR was defined.

Bit 4	ASQS_DBV	Default receive buffer is valid. If set, this bit indicates that a default receive buffer was defined when the ASQ was allocated. Otherwise, the default buffer was not defined.
Bit 3	ASQSK_DIS	Register stacking is disabled if this bit is set. Otherwise, registers are pushed on the user's stack before ASR dispatch.
Bits 2-0	Reserved	

The circular queue follows the control portion. Its starting and ending addresses are contained in ASQBOT and ASQTOP, respectively.

The last portion of the ASQ is the overflow area that starts at the location pointed to by ASQTOP. The length of the overflow area is equal to the contents of ASQML plus 4 bytes.

ASQML (2 bytes)	Maximum message length accepted.
ASQCNT (2 bytes)	Count of events currently stored within the ASQ.

2.4 EVENT MANAGER DIRECTIVES

The event manager directives used by a task for event queueing and event servicing are described on the following pages:

ALLOCATE ASYNCHRONOUS SERVICE QUEUE (ASQ) GTASQ

Directive Number: 31

Parameter: Logical address of parameter block defining the ASQ

Target Task (8 bytes)	Task_id of task to receive ASQ. (Refer to target task interface paragraph 1.3.6.)	
ASQ Status (1 byte)	Initial status of new ASQ and associated functions.	
	Bit 5=1	ASR service vector is NOT valid. User does not want an ASR. (NOTE)
	=0	ASR service vector is valid. User requires an ASR.
	Bit 4=1	Receiving buffer is valid. User is defining a default receiving buffer. (NOTE)
	=0	Receiving buffer is not valid. User is not defining a default receiving buffer.
	Bit 3=1	Disable register stacking at event interrupt.
	=0	Enable register stacking at event interrupt.
	Bit 2=1	ASR enabled.
	=0	ASR disabled.
	Bit 1=1	Enable the default input buffer.
	=0	Disable the default input buffer.
	Bit 0=1	ASQ enabled.
	=0	ASQ disabled
Maximum Message Length (1 byte)	Maximum number of bytes from an event to be transferred to the receive buffer.	
Queue length (4 bytes)	Number of bytes reserved for events.	
ASR Service Vector (4 bytes)	Logical address of target task's ASR. This is the default service vector. This address is valid only if bit 5 of the ASQ status byte is set to 0.	

Receiving Buffer (4 bytes)	Logical address of target task's default input buffer. This address is valid only if bit 4 of the ASQ status byte is set to 1. This buffer should be as long as the maximum message length.

NOTE: These two bits are opposite, i.e., the ASR service vector address is valid when its valid bit is 0, but the receive buffer address is valid when its valid bit is 1.

Detailed Description:

RMS68K allocates memory for the target task's ASQ. The ASQ consists of a fixed length ASQ control block, the area for receiving messages, whose length is specified in the ASQ parameter block, and an overflow area. RMS68K records the default service vector (ASR), the default receive buffer and the status of the ASQ, ASR, the default receive buffer, and the register stacking parameter.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	ASQ was successfully allocated.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.
6/$06	ASQ already exists for target task.
8/$08	Memory space is not available.
12/$0C	Message buffer not in caller's address space.
15/$0F	Invalid options for this directive. Task has attempted to enable its ASR or default buffer without declaring them valid.

EXAMPLE:

A user task, TSKA, wants to allocate an ASQ for itself. The ASQ is to serve messages up to 20 bytes in length and is to accommodate up to four messages. The ASR of the task is located at address AASR, and its default input buffer is at address INBUFF.

```
TSKA:     MOVE.L   #31,D0         Load GTASQ directive number 31.
          LEA      PRMBLK,A0      Load parameter block address.
          TRAP     #1
          BNE      FAULT          Branch, if error.
            .
            .
            .
PRMBLK:   DC.L     0              Task to receive ASQ.
          DC.L     0              Session number.
          DC.B     %00011111      ASR address is valid.
                                  Receive buffer address is valid.
                                  Register stacking is disabled.
                                  ASR is enabled.
                                  Receive buffer is enabled.
                                  ASQ is enabled.
          DC.B     20             Maximum message length.
          DC.L     4*20           Reserve room for four messages.
          DC.L     AASR           Address of ASR.
          DC.L     INBUFF         Address of input buffer.

INBUFF:   DS.B     20             Default receive buffer.
```

SET ASQ/ASR STATUS SETASQ

Directive Number:	33	
Parameter:	New ASQ Status	
	Bits 31-3	Reserved
	Bit 2=1	Enable ASR
	=0	Disable ASR
	Bit 1=1	Enable default receive buffer
	=0	Disable default receive buffer
	Bit 0=1	Enable ASQ
	=0	Disable ASQ

Detailed Description:

RMS68K replaces the requesting task's current ASQ, ASR, and default receive buffer status with the requested status. If the new status enables the ASR and there is an event in the ASQ, an ASR interrupt occurs and the ASR is disabled. However, if RMS68K detects any error within this directive, no update occurs.

When an ASQ is disabled, requests to queue an event to that ASQ are rejected, but the events already in the ASQ remain. When an ASR is disabled, the event manager will not dispatch the task to the ASR even if these are events in the associated ASQ. Table 2-1 summarizes the effect of enabling and disabling the ASQ and ASR on the behavior of the event manager.

TABLE 2-1. Effect of Enabling and Disabling ASQ and ASR on Event Manager

ASQ	ASR	ACTION
Enabled	Enabled	Events accepted into ASQ and ASR-processed.
Enabled	Disabled	Events accepted into ASQ, but not ASR-processed.
Disabled	Enabled	New events not accepted into ASQ, but existing events ASR-processed.
Disabled	Disabled	New events not accepted into ASQ, and existing events not ASR-processed.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Status was changed Successfully.
4/$04	Requestor has no ASQ.
7/$07	An ASR was not previously defined.
10/$0A	Event interrupt is appropriate but not possible because of ASR address, service vector, or USP pointing outside requestor's address space.
11/$0B	A default receiving buffer was not previously defined.

EXAMPLE:

TSKA wants to service all events in its ASQ before accepting new events, so it disables its ASQ, rejecting new events. It wants to process the events asynchronously, so it enables its ASR. To speed up the processing, the task enables its default buffer so that the event will have already been moved to the buffer when the ASR is entered.

```
TSKA:    MOVE.L   #33,D0           Load SETASQ directive number 33.
         MOVE.W   #%00000110,A0    ASQ disabled.
                                   AUTOBUF enabled.
                                   ASR enabled.
         TRAP     #1
         BNE      FAULT            Branch, if error.
```

QUEUE EVENT TO TASK — QEVNT

Directive Number: 35

Parameter: Event Block Address

Event Block:

Target Task (8 bytes)	Task_id of task to receive event. (Refer to target task interface, paragraph 1.3.6.)
Directive Options (2 bytes)	Bit 15=1 Alternate service vector is supplied for this event.
	=0 Default ASR service vector to be used for processing this event.
	Bits 14-0 Reserved
Event Address (4 bytes)	Logical address of event being queued. Must be on a word boundary.
Alternate Service Vector (4 bytes)	Supplied only if option bit 15=1. When this event causes an ASR interrupt, control is transferred to this logical address. However, if the receiving task is processing events in the synchronous mode, this field will be effectively ignored.

Detailed Description:

If the ASQ is enabled, RMS68K places the specified event in the ASQ of the target task or moves the event directly into the target's default buffer. The requesting task can queue an event to itself or another task. The event at the location specified in the event address field of the event block must conform to the message event format defined in paragraph 2.7.

The QEVNT directive causes the target task to undergo an ASR interrupt if the receiving task's ASQ and ASR are both enabled or if it is in the WAITING ON EVENT state.

WARNING

THE ASQ REQUIRES THAT THE ODD LENGTH EVENTS BE "ROUNDED UP" BY 1 BYTE. THE "ROUNDING UP" CAUSES AN EVENT OF LENGTH 2N + 1 TO BE REJECTED AS "EXCEEDING MAXIMUM MESSAGE LENGTH" WHEN SENT TO A TASK WHOSE ASQ WAS DECLARED AS ACCEPTING EVENTS OF LENGTH 2N + 1. ALWAYS DECLARE THE MAXIMUM MESSAGE LENGTH TO BE AN EVEN NUMBER OF BYTES TO AVOID THIS PROBLEM.

Sending a code $03 event causes the length of the event to increase by 8 bytes; sending an event to a task's alternate service vector causes the length to increase by 4 bytes if the event must be stored temporarily in the receiving task's ASQ. If a task is designed to receive either of these types of events, its maximum message length should be set to account for the extra bytes.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.
4/$04	Target task has no ASQ.
5/$05	Target task's ASQ is full.
10/$0A	ASR address, service address, or USP of target task points outside target task's address space.
12/$0C	Event address not in requestor's address space.
14/$0E	Target task's ASQ not enabled.
16/$10	Message length greater than target task's maximum message length allowed, message length is less than 4, or message not on word boundary.

EXAMPLE:

TSKA wants to queue an event to TSKB that is serviced by TSKB's ASR at the default service address.

```
    TSKA:         .
                  .
                  .
          MOVE.L    #35,D0              Load QEVNT directive number 35.
          LEA       EVTBLK,A0           Load parameter block address.
          MOVE.L    #EVNT,EVTADR        Modify parameter block.
          TRAP      #1
          BNE       FAULT               Branch, if error.
                  .
                  .
                  .
    EVTBLK:   DC.L      'TSKB'              Target taskname.
              DC.L        0                 Same session.
              DC.W        0                 Use default ASR service vector.
    EVTADR:   DC.L        0                 Address of event.
              DC.L        0                 Alternative ASR.
                  .
                  .
                  .
    EVNT:     DC.B        6                 Length of event.
              DC.B       $03                Event code.
              DC.L        0                 Message text.
                  .
                  .
```

GET AN EVENT GTEVNT

Directive Number: 38

Parameter: Receive Buffer Address

Detailed description:

RMS68K moves the oldest event sent to the task to the receive buffer. If the task's default receive buffer is enabled, the requested receive buffer is ignored, and the event is moved to the default receive buffer.

The requesting task should ensure that the receiving buffer is large enough to accommodate the longest event its ASQ can accept.

If no event exists in the task's ASQ, it is put into a getting an event state until an event is sent to its ASQ. When this occurs, the task is made ready and dispatched to the instruction immediately following the directive call with the event available in the appropriate buffer.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	The operation was successful and an event is present either in the receiving buffer or in the default buffer.
4/$04	ASQ does not exist for requestor.
10/$0A	Requestor's ASR is enabled, or its ASQ is disabled.
12/$0C	Receiving buffer not in requestor's address space.

EXAMPLE 1:

A task, TSKA, wants to process the next event in the synchronous mode. TSKA's default receive buffer is currently disabled.

```
TSKA:          MOVE.L   #38,D0      Load GTEVNT directive number 38.
               LEA      RCVBUF,A0   Load receive buffer address.
               TRAP     #1
               BNE      FAULT       Branch, if error.
                 .                  Process the event.
                 .
                 .
RCVBUF:        DS.B     MAXMSG      Receive buffer.
```

EXAMPLE 2:

A task wants to process the next event in the synchronous mode and has its default buffer INBUFF enabled.

```
TSKA:          MOVE.L   #38,D0      Load GTEVNT directive number 38.
               TRAP     #1
               BNE      FAULT       Branch, if error.
                 .                  Process the event.
                 .
                 .
INBUFF:        DS.B     MAXMSG      Default receive buffer.
```

READ EVENT RDEVNT

Directive Number: 34

Parameter: Receive Buffer Address.

Detailed Description:

RMS68K moves the oldest event sent to the task to the receive buffer. If the task's default input buffer is enabled, the requested receiving buffer is ignored and the event is moved to the default receive buffer.

The requesting task should ensure that the receiving buffer is large enough to accommodate the longest event its ASQ can accept.

If no event exists in the caller's ASQ, the first 2 bytes of the receive buffer or default buffer are set to 0.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	The operation was successful and an event is present either in the receiving buffer, or in the default buffer.
4/$04	Requestor has no ASQ.
12/$0C	Receiving buffer not contained in requestor's address space.

EXAMPLE 1:

An ASR, beginning at location ASRTN, wants to process the next event in its ASQ after an ASR interrupt has occurred. The default receive buffer is disabled.

```
ASRTN:    MOVE.L   #34,D0            Load RDEVNT directive number 34.
          LEA      RCVBUF,A0         Load receive buffer address.
          TRAP     #1
          BNE      FAULT             Branch, if error.
            .
            .
            .
RCVBUF:   DS.B     MAXMSG            Receive buffer.
```

EXAMPLE 2:

A task wants to process the next event in its ASQ and has its default buffer INBUFF enabled.

```
    TSKA:     MOVE.L    #34,D0              Load RDEVNT directive number 34.
              TRAP      #1
              BNE       FAULT               Branch, if error.
                 .
                 .
                 .
    INBUFF:   DS.B      MAXMSG              Default receive buffer.
```

WAIT FOR EVENT WTEVNT

Directive Number: 36

Parameter: None

Detailed Description:

RMS68K ensures that the ASQ and ASR of the requesting task are enabled and places the task in the WAIT FOR EVENT state.

If the ASQ is not empty, the ASR is entered immediately; otherwise the next event sent to the task causes an ASR interrupt. In either case, if the default receive buffer was enabled, the event will be present in the default receive buffer.

When the ASR is entered, the previous status register and program counter are pushed on the task's stack. If the register stacking feature is enabled, the task's registers are also pushed.

The Executive returns control to the instruction immediately following the WTEVNT directive call when the ASR returns from event service by issuing an RTEVNT directive or an RTR instruction.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
4/$04	Requestor has no ASQ.
7/$07	An ASR was not previously defined.
10/$0A	Event interrupt is due but not possible because of ASR address, service vector, or USP pointing outside requestor's address space.

EXAMPLE:

TSKA wants to stop current execution and wait until it has received an event.

```
    TSKA:        .
                 .
                 .
                MOVE.L   #36,D0          Load WTEVNT directive number 36.
                TRAP     #1
                BNE      FAULT           Branch, if error.
                 .
                 .
                 .
```

RETURN FROM EVENT SERVICE RTEVNT

Directive Number:	37	
Parameter:	A0 = ASR Status	
	Bits 31-1	Reserved
	Bit 0=1	Enable ASR.
	=0	Current status of ASR left unchanged.

Detailed Description:

If register stacking was enabled in the original GTASQ directive call, RMS68K restores the contents of data registers D0-D7 and the address registers A0-A6. In either case (register stacking enabled or not), RMS68K restores the status register and the program counter and then returns control to the point where the event interrupt occurred.

The ASR can re-enable itself by setting A0 appropriately and issuing the RTEVNT directive. If the ASR is enabled and an event is present in the ASQ, RMS68K generates an event interrupt and control is transferred to the ASR. RMS68K returns an error code, if a request to enable the ASR is encountered and no ASR was declared.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

7/$07	An ASR was not previously defined.

EXAMPLE:

An ASR, named ASRTN, after processing an event, re-enables itself, and returns to the point of task interruption.

```
ASRTN:    Process event
            .
            .
            .
          MOVE.L   #37,D0          Load RTEVNT directive number 37. (Issue
                                   a RTEVNT call)
          MOVE.L   #0001,A0        Re-enable ASR.
          TRAP     #1
          BNE      FAULT           Branch, if error.
```

DEALLOCATE ASYNCHRONOUS SERVICE QUEUE (ASQ) DEASQ

Directive Number: 32

Parameter: None

Detailed description:

The memory dedicated to the requestor's ASQ is freed. Any unserviced events in the ASQ are lost. If the requestor has no ASQ, the directive is ignored. D0 is cleared to 0.

Error Codes (returned in bits 15-0 of D0): None

EXAMPLE:

A user task, TSKA, wants to delete its ASQ.

```
    TSKA:         .
                  .
                  .
             MOVE.L   #32,D0         Load DEASQ directive number 32.
             TRAP     #1
             BNE      FAULT          Branch, if error.
                  .
                  .
                  .
```

2.5 EXAMPLES

EXAMPLE 1: TSKA wants to process events in a synchronous mode.

```
GTASQPB:        DC.L      0               Task_id
                DC.L      0
                DC.B      %00110011       ASR Address      <== Invalid
                                          Receive Buffer Address
                                                           <== Valid
                                          Register Stacking
                                                           <== Enabled
                                          ASR              <== Disabled
                                          Default Receive Buffer
                                                           <== Enabled
                                          ASQ              <== Enabled
                DC.B      20              Maximum Message Length
                DC.L      80              Allow 4 Messages
                DC.L      0               ASR Address
                DC.L      INBUFF          Input Buffer Address

INBUFF:         DS.B      20

TSKA:           MOVE.L    #31,D0          Load  GTASQ  directive  number  31
                                          (allocate an ASQ).
                LEA       GTASQPB,A0      Point  A0  to  the GTASQ parameter
                                          block.
                TRAP      #1
                BNE       FAULT           Branch, if error.

GTEVNT:         MOVE.L    #38,D0          Load  GTEVNT  directive  number 38
                                          (Get the Event).
                TRAP      #1
                BNE       FAULT           Branch, if error.

PROCESS:        .
                .
                .

GOFORMOR:       BRA       GTEVNT          Go back to Get Next Event.

FAULT:          .
                .
                .
```

EXAMPLE 2:   TSKB wants to process events in an asynchronous mode.

```
GTASQPB:        DC.L      0               Task_id
                DC.L      0
                DC.B      %00011111       ASR address     <== Valid
                                          Receive Buffer Address
                                                          <== Valid
                                          Stack Registers
                                                          <== Disabled
                                          ASR             <== Enabled
                                          Receive Buffer  <== Enabled
                                          ASQ             <== Enabled
                DC.B      20              Maximum message length
                DC.L      80              Allow 4 messages
                DC.L      TSKBASR         ASR Address
                DC.L      INBUFF          Input Buffer Address

INBUFF:         DS.B      20

TSKB:           MOVE.L    #31,D0          Load  GETASQ  directive  number 31
                                          (allocate an ASQ).
                LEA       GETASQPB,A0     Point  A0  to the GETASQ parameter
                                          block.
                TRAP      #1
                BNE       FAULT           Branch, if error.

DOWORK:           .                       Other  work is done until an event
                  .                       is received, at which time TSKBASR
                  .                       is entered to process the event.
                  .
TSKBASR:          .                       Process the event
                  .                       (In the INBUFF).
                  .
                  .
                  .

                MOVE.L    #37,D0          Load RTEVNT directive number 37.
                MOVE.W    #$0001,A0       Enable the ASR.
                TRAP      #1              Return to Main Code.

FAULT:            .
                  .
                  .
```

EXAMPLE 3: TSKC wants to send events to TSKD.

```
GTTIDPB:     DC.L     'TSKD'          Target Task
             DC.L     0               Session number

QEVNTPB:     DC.L     0               Task_id
             DC.L     0
             DC.W     0               Options
             DC.L     OUTBUF          Event address
             DC.L     0               Alternate service address (none)

OUTBUF:      DS.B     20

TSKC:        MOVE.L   #10,D0          Load  GTTASKID directive number 10
                                      (Get target task's task_id.)
             LEA      GTTIDPB,A0      Point A0 to the GTTASKID parameter
                                      block.
             TRAP     #1
             BNE      FAULT           Branch, if error.

             MOVE.L   A0,QEVNTPB      Save target task task_id.
             MOVE.L   A1,QEVNTPB+4

LOOP:         .                       Do work and create event.
              .
              .

             MOVE.L   #35,D0          Load  QEVNT  directive  number  35
                                      (QEVNT to target task).
             LEA      QEVNTPB,A0      Point   to   the  QEVNT  parameter
                                      block.
             TRAP     #1
             BEQ      LOOP

FAULT:        .
              .
              .
```

2.5.1 Recommended Use

The use of a default receive buffer is recommended as stated in paragraph 2.2.4.

The synchronous mode is recommended using the GTEVNT directive because:

a. Dispatching to the ASR is slower than dispatching inline.

b. Dispatching inline eliminates the requirement to return to inline code via the RTEVNT directive (or RTR instruction).

c. Eliminates the requirement to possess an ASR if asynchronous event processing is not necessary.

d. This method is a subroutine-like interface, that is a well-known and understood concept that is easier to use, especially for first time users.

2.6 EVENT MESSAGE FORMATS

The detailed format of each type of RMS68K defined event message that a task can receive in its ASQ is described on the following pages. Most of the event messages shown originate in RMS68K, however, event code = $03 originates in a user task and is sent to a user task via RMS68K. In the latter case, RMS68K adds some fields to the event message so this format is different for the sender and receiver.

Code $01 Events -- I/O Completion Interrupt or Message from I/O Handler

This event is returned to the task attached to an I/O Channel by an I/O handler when an I/O function has been completed. The event appears in the task's ASQ as:

1 byte	Length
1 byte	Event code = $01
1 byte	Event type =
	$70 Normal $71 Halt/abort $FF Unsolicited channel
1 byte	User generated channel identification key.
4 bytes	User supplied identification:
	Event types $70 or $71 — Usually the DCB address.
	Event type $FF — The channel mnemonic.
2 bytes	Status of the request

```
$70   0          --> Successful
      Non-zero   --> Unsuccessful - value is error code

$71   0          --> No I/O to halt
      Non-zero   --> I/O halt successful

$FF   0          --> Channel has been reset
      Non-zero   --> Channel down
```

Event type:

$80 Unsolicited device status whose status value is $00.

1 byte	Length
1 byte	Event code = $01
1 byte	Event type = $80
1 byte	User-generated channel identification key.
4 bytes	User-supplied identification, usually the DCB address.
2 bytes	Status value = $00
2 bytes	Device status

Event type:

$80 Unsolicited device status whose status value is $01.

1 byte	Length
1 byte	Event code = $01
1 byte	Event type = $80
1 byte	User-generated channel identification key
4 bytes	Channel mnemonic
2 bytes	Status value = $01
2 bytes	Device status
1 byte	Device number
1 byte	Reserved

Device status - Byte 1:

Terminal	Bit	Meaning if set
	7	Ready
	6-1	Available for use
	0	Break condition
Printer	7	Ready
	6-0	Available for use
Disk	7	Ready
	6	Available for use
	5	Write protected
	4-0	Available for use

Device status - Byte 2:

Bit	Meaning
7-4	Available for use
3-0	Type of device

Value	Device
1	Floppy diskette
2	Rigid disk
3	Terminal
4	Printer
5	Magnetic tape

Code $02 Events -- Task ISR Events

This event is queued to a task when one of its ISRs has executed the RTE directive with D0 = 2 (queue an event on return from ISR).

1 byte	Length = $06
1 byte	Code = $02
4 bytes	MSG = contents of D2 when RTE directive was issued

This event is returned to a task when one of its ISRs has encountered an exception (bus error, etc.).

1 byte	Length = $0A
1 byte	Event Code = $02
4 bytes	Error Flag = $FFFFFxxx

where: xxx is type of exception):

010 = bus error	016 = privilege violation
011 = address error	017 = unimplemented instruction (1010 opcode pattern)
012 = illegal instruction	018 = unimplemented instruction (1111 opcode pattern)
013 = zero divide	
014 = CHK instruction	
015 = TRAPV instruction	

4 bytes	Program Counter at time of exception

Code $03 Event -- User Task Events

This event originates in a user task and is sent to a user task. The text of the message can be any format that has been agreed on by the sending and receiving tasks.

Event sent:

1 byte	Length = N
1 byte	Event code = $03
N - 2 bytes	Message text

Event received:

1 byte	Length = N + 8
1 byte	Event code = $03
8 bytes	Task_id of sending task in format appropriate for use by receiving task
N - 2 bytes	Message text

Code $04 Event -- Timeout

This event originates in RMS68K when a task is to receive an event as the result of the previously issued RQSTPA directive.

If no Request ID was supplied:

1 byte	Length = $0A
1 byte	Event Code = $04
4 bytes	Current System Date
4 bytes	Current System Time

If request ID was supplied:

1 byte	Length = $10
1 byte	Event code = $04
4 bytes	Current system date
4 bytes	Current system time
4 bytes	Activation request ID (Usually the DCB address)
2 bytes	Activation count; incremented by one each time an event is queued.

Code $05 Event -- Subtask Termination

This event originates in RMS68K when a subtask (or monitored task) terminates and is sent to the subtask's monitor.

1 byte	Length = $18	
1 byte	Event code = $05	
8 bytes	Task_id of subtask	
8 bytes	Task_id of the task that initiated the termination of the subtask	
1 byte	Termination code -	1 = normal termination 2 = abnormal termination
1 byte	Not used	
2 bytes	Abort code	Bit 15 = 0 implies the subtask aborted itself. Bit 15 = 1 implies RMS68K aborted the subtask. This is the contents of the lower 2 bytes in register A0.

If the subtask aborts itself, the abort code is as supplied in the ABORT directive. If RMS68K aborts the subtask, the abort code corresponds to the exception causing the abort as:

Abort Code	Exception
$8010	bus error
$8011	address error
$8012	illegal instruction
$8013	divide by zero
$8014	CHK instruction
$8015	TRAPV instruction
$8016	privilege violation
$8017	line 1010 emulator
$8018	line 1111 emulator

2 bytes	Upper 2 bytes of register D0. If the subtask terminates itself, this code is as supplied in the ABORT or TERM directive. If RMS68K aborts the subtask, this code is 0.

Code $07 Event -- Task Server

This event originates in RMS68K when a task (the requesting task) requests the services of a server task with a trap instruction. RMS68K notifies the server task of the request with this event.

1 byte Length = $18

1 byte Event code = $07

1 byte Trap instruction number.

Bit 7	= 1	requesting task is a system task.
	= 0	requesting task is a user task.
Bits 6-5	= 00	requesting task is requesting service.
	= 10	requesting task is terminating.
	= 11	requesting task is terminating and is the only task in that session.
Bits 3-0		trap instruction number used to invoke server task.

1 byte Current priority of requesting task.

8 bytes Task_id of requesting task.

2 bytes User generated I.D. (refer to CRTCB directive).

4 bytes Value of requesting task's register D0.

4 bytes	Value of requesting task's register A0.
1 byte	Parameter block status.

Value	Meaning
0	Total parameter block moved.
1	Parameter block partially moved.
2	Bad parameter block was specified; parameter block not moved.
3	Parameter block move was not requested.

1 byte	Parameter block size indicating number of bytes of the parameter block that follows.

NOTE: If the server task specified the option for parameter block move when the task established itself as a server task, the parameter block immediately follows the event message in the server task's buffer.

Code $08 Event -- Exception Monitor

This event originates in RMS68K when a target task is attached or detached from an exception monitor task, or when a target task under the control of an exception monitor is halted. The event is sent to the exception monitor task.

1 byte	Length = $0C
1 byte	Event code = $08
8 bytes	Task_id of target task
1 byte	Exception code If the exception type field has value $01 or $02, this field is 0.

If the exception type field has value $03, the exception code and its meaning are:

CODE	MEANING	CODE	MEANING
$00	Reserved	$10	Bus error
$01	TRAP #1	$11	Address error
$02	TRAP #2	$12	Illegal instruction
$03	TRAP #3	$13	Zero divide
$04	TRAP #4	$14	CHK instruction
$05	TRAP #5	$15	TRAPV
$06	TRAP #6	$16	Privilege violation
$07	TRAP #7	$17	Line 1010 emulator
$08	TRAP #8	$18	Line 1111 emulator
$09	TRAP #9	$19	Not used
$0A	TRAP #10	$1A	Not used
$0B	TRAP #11	$1B	Maximum count reached
$0C	TRAP #12	$1C	Traced 1 instruction
$0D	TRAP #13	$1D	Value change occurred
$0E	TRAP #14	$1E	Value equal occurred
$0F	TRAP #15		

1 byte Exception event type with a value of:

$01 - Target task was attached to exception monitor.
$02 - Target task was detached from exception monitor.
$03 - Exception code indicates reason for exception event.

THIS PAGE INTENTIONALLY LEFT BLANK.

CHAPTER 3

MEMORY MANAGER

3.1 THEORY OF OPERATION

The Memory Manager within RMS68K consists of those data structures and directives that support the concept of memory as a resource that can be allocated to a task, shared between two or more tasks, transferred from one task to another, and deallocated or returned to the system. The unit of memory manipulated by the memory manager is the segment.

3.1.1 Segments

The Segment is a block of memory consisting of an integral number of contiguous pages of physical memory. The page size can be set at initialization to any value 2**n, where $8 \leq n \leq 16$. A typical page size is 2**8 or 256 bytes. Segments are described by data structures called segment descriptors consisting of attributes:

Name
Logical address (address visible to the task)
Physical address (address visible to hardware)
Length
Protection attributes (read-only or read-write).

Another attribute of a memory segment is the type. RMS68K supports three types of memory:

Random Access Memory (RAM)
Read Only Memory (ROM)
Memory-Mapped I/O

A memory mapped I/O segment is usually a hardware device containing a set of registers visible to software. However, it may consist of any portion of the address space not described at initialization time as being part of a RAM or ROM partition (refer to paragraph 3.1.4).

Another important attribute of a segment is its scope. The scope of a segment determines whether the segment is:

Private (visible to only one task in the system)
Locally shareable (visible to any task in a particular session)
Globally shareable (visible to any task in the system).

The use of shareable segments can reduce the memory requirements and/or the message traffic within a system. An example of reducing memory requirements would be to allow multiple copies of a task, (e.g., text editor), to share a common code segment. Two or more tasks that frequently communicate long messages can reduce this traffic by composing messages in a shared data segment and passing pointers to them instead of their text.

Another segment attribute is permanence. Permanence affects whether a segment is automatically released back to the system on task termination and applies only to segments that were previously declared shareable. Since a private (non-shareable) segment may not be made permanent, all private segments are automatically released to the system on task termination. This relieves the task of the responsibility to deallocate private segments before termination. However, a locally shareable permanent segment will survive the termination of the task that created it and can only be released either by an explicit call to the deallocate segment directive (with the remove permanent status bit set), or by the termination of the last task within the session (typically the session manager in a multi-user system). Finally, a globally shareable permanent segment will survive the termination of the session that created it as long as at least one task within another session is attached to it.

Every task within the system has a Task Segment Table (TST) that contains descriptors for all segments currently attached to the task. In addition, there is a Global Segment Table (GST) within the system that contains descriptors for all globally shareable or locally shareable segments. Thus, declaring a segment shareable causes the system to create a descriptor for that segment within the GST in addition to the descriptor within the task's TST. At any moment a shareable segment is described by one descriptor within the global segment table and zero to "n" descriptors within task segment tables representing the zero to "n" tasks that are currently attached to that segment.

3.1.2 Segment Operations

The memory manager has seven directives for manipulating segments that may be grouped into two classes, and a third class that does not operate on segments. The first class contains those directives that operate on all segments within the system, the second class contains those directives applicable only to shareable segments, and the third class that provides "utility" services such as copying a portion of a memory segment from one task's address space to another and flushing all user mode entries from the cache memory. The directives are summarized in Table 3-1. (Refer to paragraph 3.5 for detailed descriptions.)

TABLE 3-1. Segment Directives

DIRECTIVE	DESCRIPTION
	<u>Class 1 Segment Directives</u>
GTSEG	Allocate a new code or data segment to the target task by placing it within the task's address space.
TRSEG	Remove a segment from the requesting task's address space and place it within the address space of another task.
RCVSA	Return a description of the specified segment to the requesting task.
DESEG	Delete a code or data segment from the target task's address space.
	<u>Class 2 Shareable Segment Directives</u>
DCLSHR	Make a segment available for shared access.
ATTSEG	Place an existing shareable segment within the requesting task's address space.
SHRSEG	Place an existing shareable segment within another task's address space.
	<u>Class 3 Utility Memory Directives</u>
MOVELL	A task requests that data be copied from the logical address space of one task to the logical address space of another task.
MOVEPL	A system task requests that data be copied from any physical address to a logical address within a target task's address space.
FLUSHC	Flushes all user mode entries from all caches known to the Executive.

3.1.3 Partitions

In addition to segments, RSM68K supports another level of memory organization known as Partitions. A partition is a large piece of RAM or ROM (usually one or more boards), managed by the memory manager. All RAM or ROM segments are allocated from and returned to specific partitions so segment boundaries cannot overlap partition boundaries.

Partitions are useful in differentiating between types of memory within a system such as, onboard and offboard RAM, or in reserving large pieces of memory for specific purposes (operating system, task, graphics).

The address space described by a partition must be contiguous, although the real physical memory contained within the partition may contain gaps or holes. Each partition is described by a data structure known as a partition descriptor, consisting of:

a. Partition number
b. Partition type
c. Starting address
d. Ending address (ROM partitions) or
   Pointer to free memory list header (RAM partitions)

All the partition descriptors are grouped into a collected data structure known as the Memory Map Table (MEMMAP).

The only partition attributes a task references are the partition number and the partition type. The partition number is equivalent to a name and is used for identification. The partition type is used to group different physical partitions of memory into types or classes of memory possessing similar characteristics.

A system that contains some local memory on the processor board, a memory board connected via the VMX local memory bus, and a memory board connected via the VME system bus can be designed as a three partition system. The memory boards are described as:

MEMORY BOARD	PARTITION NUMBER	PARTITION TYPE
Onboard	0	local
VMXbus	1	local
VMEbus	2	global

If a task needs to get a segment for use as a stack segment, it can ask for memory as:

```
(partition_number = dont_care) and
(partition_type = local)
```

RMS68K looks at both the onboard RAM and the VMXbus RAM to satisfy the request. If however, the task needs to acquire a segment of memory on the VMXbus memory board, the type = local description is not enough to differentiate the onboard memory from the VMXbus memory. Here the partition number needs to be explicitly encoded as:

```
(partition_number = 1) and
(partition_type = dont_care)
```

Memory request precedence rules are:

a. A partition number or partition type of 0 is treated as a dont care.

b. An explicitly coded partition number (nonzero) takes precedence over an explicitly coded partition type.

c. The default partition numbers and types are set using the following SYSGEN parameters and are typically set to 0:

```
MTYPE$URO for user task read-only memory
MTYPE$URW for user task read-write memory
MTYPE$SRO for system task read-only memory
MTYPE$SRW for system task read-write memory
```

These precedence rules are summed up in the algorithm:

```
IF (partition_number <> 0)
THEN set_memory
   (number = partition number)
   (type = dont_care);

ELSE IF (partition_type <> 0)
THEN set_memory
   (number = dont_care)
   (type = partition_type);

ELSE get_memory
   (number = dont_care)
   (type = default_type as
           specified in SYSGEN);
```

There are two other SYSGEN parameters about default partition numbers and types for TCBs and ASQs that are typically set to 0.

```
MEMTYPA for the ASQ
MEMTYPT for the TCB
```

All other requests for system memory default to partition type 0. Thus, if the user wants to protect a memory partition from unanticipated system requests, the memory should be described as a type other than 0.

3.1.4 Free Memory Lists

RAM memory is organized into doubly linked lists of nodes called Free Memory Lists. Each node within the list contains a 16-byte control portion for bookkeeping purposes and one or more pages of free memory, followed by zero or more pages of used memory. When the last portion of free memory within a node (including the control portion) is allocated, the entire node is considered used and is concatenated with the used portion of the previous node within the partition. When a task frees a piece of memory, one of four results may occur depending on the position of the piece within the used portion of the node and whether the piece represents the entire used portion of the node or not:

a. The piece may increase the free portion of its current node.

b. The piece may increase the free portion of the next node.

c. The piece may cause a new node to be created between the current node and the next node (node creation).

d. The piece may cause the entire current node to become free and concatenated with the free portion of the next node within the list (node deletion).

In other words, the memory manager implements a policy of automatic "garbage collection" with the free memory list.

3.2 DATA STRUCTURES

The data structures in the memory manager are:

Memory Map Table	Array of partition descriptors.
Free Memory List	Doubly linked list of nodes within a single partition.
Task Segment Table	Array of segment descriptors describing all segments belonging to a particular task.
Global Segment Table	Array of segment descriptors describing all shareable segments within the system.
Segment Parameter Block	Parameter block used for requests to the memory manager.

3.2.1 Memory Map Table (MEMMAP)

The MEMMAP is an array of partition descriptors. Each descriptor is 10 bytes long and is composed of four fields. The MEMMAP contains one descriptor for each RAM or ROM partition designated by the user at initialization time. The end of the table is indicated by a sentinel descriptor consisting of 2 bytes of $FF. The MEMMAP table is pointed to by the MAPBEG variable within SYSPAR. Table 3-2 describes the MEMMAP entries.

TABLE 3-2. Memory Map Table Entry (MEMMAP)

ENTRY	DESCRIPTION
MAPMTYP (1 byte)	Memory type in bits 7-4. If MAPMTYP=$FF and MAPPART=0, this entry describes a ROM partition. The end of the MEMMAP is recognized when both MAPMTYP and MAPPART equal $FF.
MAPPART (1 byte)	Partition number in bits 3-0.
MAPSTRA (4 bytes)	Start of available memory for this partition.
MAPFMLP (4 bytes)	For RAM partitions this points to the free memory list header. For ROM partitions, this represents the top boundary (1 byte past the last byte within the partition).

3.2.2 Free Memory List

The Free Memory List consists of one header node, describing the entire partition, and a doubly linked list of nodes describing the dynamic state of the partition. The free memory list header either resides in the first or last page of the partition as indicated by the user at initialization. The entry describing a given node of free memory is found in the first 16 bytes of that node. Table 3-3 describes the free memory list.

TABLE 3-3. Free Memory List

ENTRY	DESCRIPTION
	Header Information
LOWFREE (4 bytes)	Points to the first (lowest address) entry in the free memory list.
STRAVAIL (4 bytes)	Start of available memory for the partition.
ENDAVAIL (4 bytes)	End of available memory for this partition.
MEMTYPE (1 byte)	Memory type and partition number.
(1 byte)	Unused.
SEMFRMEM (6 bytes)	Semaphore used to control access to the free memory list.
SEMWTMEM (6 bytes)	Semaphore used to queue tasks waiting for memory to become free.
	Doubly Linked List
The entry describing a given node of free memory is found in the first 16 bytes of that node.	
FMLFP (4 bytes)	Forward pointer to next node in list. If FMLFP=0, this is the last node in list.
FMLBP (4 bytes)	Backward pointer to previous node in list. If FMLBP=0, this is the first node in list.
FMLFREE (4 bytes)	The number of 256-byte pages that are free in this block.
FMLUSED (4 bytes)	The number of 256-byte pages that have been allocated.

3.2.3 Segment Descriptors

Segment descriptors exist within the Global Segment Table (GST) and the Task Segment Table (TST) although the format is slightly different. The basic difference is that the segment descriptors within the GST contain only those segment attributes that are global, such as name, session number, attributes, physical address, and length, while the descriptors within the TST contain local information about the logical address of the segment as seen by the task as well as global information.

3.2.4 Task Segment Table (TST)

The TST consists of a control portion followed by two parallel arrays containing the segment descriptors. The first array contains mapping and protection information for the MMU and the second contains naming and attribute information for the memory manager. The information was placed in two arrays to accelerate the loading of segment descriptors into the EXORmacs MMU. The TST is described in Table 3-4.

TABLE 3-4. Task Segment Table

ENTRY	DESCRIPTION
TST (4 bytes)	Block ID Each TST begins with '!TST' to allow consistency checking and ease of dump reading.
TSTNSEGS (1 byte)	Allowed segments Contains the maximum number of segments this task is allowed to address at any instant.
TSTCSEGS (1 byte)	Current segments Contains the number of segments currently included in this task's address space.
TSTLMMU (2 bytes)	Last MMU segment index Contains the offset to the last entry in the MMU load table.
TSTLATTR (2 bytes)	Last attribute index Contains the offset to the last entry in the segment attribute table.
TSTSTAT (2 bytes)	Reserved for future use.

TABLE 3-4. Task Segment Table (cont'd)

ENTRY	DESCRIPTION
TSTMMU (32 bytes)	MMU load table Contains the information required to allow the MMU to control address space access for this task. The number of entries in this table is equal to the number contained in TSTNSEGS. A TSTMMU entry is defined as: TSTLB (2 bytes) — Bits A23-A16 of the beginning logical address. TSTLE (2 bytes) — Bits A23-A16 of the ending logical address. TSTPO (2 bytes) — Bits A23-A16 of the physical offset. (logical address + physical offset = physical address) BTCTL (2 bytes) — Control byte: 1 = read/write segment 3 = read only segment
TSTATTR (32 bytes)	Segment attribute table Contains segment information not relevant to the MMU. A TSTATTR entry is defined as: TSTANAME (4 bytes) — Segment name. TSTAATTR (2 bytes) — Segment attribute field. Bits definitions: 15 = Segment entry in use. 14 = Segment is read only. 13 = Segment is locally shareable. 12 = Segment is globally shareable. 11 = Segment is memory mapped I/O space. 10 = Segment is physical ROM. TSTAR1 (1 byte) — Segment type. TSTAUCNT (1 byte) — Segment use count.

3.2.5 Global Segment Table (GST)

The GST is used to describe all shareable segments within the system. It consists of a control portion followed by an array of segment descriptors. Table 3-5 describes the GST.

TABLE 3-5. Global Segment Table

ENTRY	DESCRIPTION
GST (4 bytes)	Block ID Each GST segment begins with '!GST' to allow consistency checking and ease of dump reading.
GSTNEXT (4 bytes)	Reserved for future use.
GSTNSEG (2 bytes)	Reserved for future use.
GSTNPAGE (2 bytes)	GST segment size Contains the number of 256-byte pages comprising this GST segment.
GSTMENT (2 bytes)	Maximum entry count Contains the maximum number of GST entries allowable in this GST segment.
GSTLENT (2 bytes)	Last entry Contains the last entry number currently in use.
GSTFENT (4 bytes)	First entry address Points to the first GST entry.
A GST entry is defined as:	
GSTSESSN (4 bytes)	Originating task's session number.
GSTNAME (4 bytes)	Segment name.
GSTATTR (2 bytes)	Segment attributes. Bit definitions are the same as the TSTATTR field (refer to paragraph 3.2.4).
GSTCNT (2 bytes)	Use count. Number of tasks currently attached to this segment.
GSTPA (4 bytes)	Physical starting address.
GSTNP (2 bytes)	Segment size in 256-byte pages.

3.2.6 Segment Parameter Block

Several of the directives require a memory segment block that describes the request to RMS68K in detail. The memory segment block must begin at an even address and the general format is:

8 bytes	Target task
2 bytes	Directive options
2 bytes	Segment attributes
4 bytes	Segment name
4 bytes	Logical address
4 bytes	Segment length

Target Task — Task_id for accessing a target task.

Directive options — Options will vary, depending on the particular directive. A description of the relevant options are included in the detailed directive description in paragraph 3.5.

Segment attributes — The segment attributes are relevant only to particular directives. If required, the format is:

Bit 15	Reserved
Bit 14 = 0	Segment is read-write
= 1	Segment is read-only
Bit 13 = 0	Segment is not locally shareable
= 1	Segment is locally shareable
Bit 12 = 0	Segment is not globally shareable
= 1	Segment is globally shareable
Bit 11 = 0	Segment is not memory mapped I/O space
= 1	Segment is memory mapped I/O space
Bit 10 = 0	Segment is not physical ROM
= 1	Segment is physical ROM
Bits 9-0	Reserved

Segment name — Specifies a particular segment to be referenced. Any 32-bit combination is a valid segment name.

Logical address — The address of the segment as viewed by the target task.

Segment length — Specifies the length, in bytes, of the segment.

3.3 MEMORY INITIALIZATION

During initialization, a table of partition descriptors defined by the user with the SYSGEN utility is input to RMS68K. RMS68K verifies that these descriptors conform to its set of rules for partitions, zeros out all memory within RAM partitions to insure good parity, creates the free memory lists for RAM partitions (marking any holes within RAM partitions as used memory), and creates the memory map table of partition descriptors.

The rules the memory initializer uses are:

a. Partitions may not overlap.

b. Partition numbers must be assigned in order of increasing address, e.g., a partition consisting of addresses $100000 to $200000 must have a lower partition number than one consisting of addresses $400000 to $500000.

Because RMS68K requires a portion of low memory for interrupt vectors, system stack, and system parameters, partition 0 has two special rules:

a. If RMS68K is within partition 0, free memory starts immediately after RMS68K or after the last task included with RMS68K in the load module, whichever is higher.

b. If RMS68K is NOT within partition 0, free memory starts immediately after the system parameter area, usually less than $1000.

3.4 MEMORY DIRECTIVES

The following pages contain detailed descriptions of the memory manager directives.

ALLOCATE A SEGMENT GTSEG

Directive Number: 1

Parameter: Segment Block Address

Segment Block (refer to paragraph 3.2.6):

Target Task (8 bytes)	Task_id of task to receive segment. (Refer to target task interface, paragraph 1.3.6.)	
Directive Options (2 bytes)	Bits 15,14	Reserved
	Bit 13=1	RMS68K sets the logical address equal to the physical address. If a physical memory allocation is requested (bit 8 = 1), logical address is equal to physical address, regardless of how this bit is set.
	=0	An address is specified in address field below.
	Bits 12,11	Reserved
	Bit 10=1	If memory is not available, do not return, wait until memory is available.
	=0	If memory is not available, take action as directed by option bit 9.
	Bit 9=1	If entire memory space requested is not available, allocate largest block that is available.
	=0	If entire memory space requested is not available, reject the directive.
	Bit 8=1	RMS68K attempts to allocate the segment at the physical address specified in address field below. If this space is non-existent, an error code is returned. If this space is already allocated, the action taken is determined by option bits 10 and 9.

	Bit 7=1	RMS68K looks at option bits 6-0 to determine which memory partitions can be used to satisfy this allocation request.
	7=0	RMS68K uses the defaults set by SYSGEN to determine which memory partitions can be used to satisfy this memory request. Separate defaults can be set for: a. System Task, Read-only b. System Task, Read-write c. User Task, Read-only d. User Task, Read-write
	Bits 6-4	Type number can be 0 to 7.
	Bits 3-0	Partition number can be 0 to 15. If partition number is 0, then allocation can be in any partition that has been assigned the specified type number. If partition number is not 0, then allocation can be only within the partition specified; type number is ignored.
Segment Attributes (2 bytes)	Bit 15	Reserved
	Bit 14=1	Segment is to be Read-only.
	=0	Segment is to be Read-write.
	Bits 13,12	Reserved
	11=1	Segment is to be memory mapped I/O space. The address given in the address field must be a physical address that is not within the limits of allocatable RAM. If this bit is set, none of the options is applicable, and the segment is allocated as a shared segment.

NOTE

Memory mapped I/O space is not intended to be used for code. A program loaded into memory mapped I/O space cannot make RMS68K directive calls.

	=0	Segment is not to be memory mapped I/O space.
	10=1	Segment is to be physical ROM. The address given in the address field must be a physical address that is within the limits of a memory partition defined as physical ROM. If this bit is set, none of the options is applicable.
	=0	Segment is not to be physical ROM.
	Bits 9-0	Reserved
Segment Name (4 bytes)		Name of new segment.
Address (4 bytes)		If options bit 8=1 or if attributes bit 11=1 or bit 10=1, then this field specifies the physical address of the new segment. For all other cases, this field specifies the logical address of the new segment. Not applicable if option bit 13 = 1.
Segment Length (4 bytes)		Length of new segment, in bytes. (Maximum length is $FFFF00. If maximum length is exceeded, results are unpredictable.)

Detailed Description:

RMS68K allocates the smallest number of memory pages that satisfies the specified length. Page size is determined via SYSGEN for a particular hardware environment. The task to receive the new segment can be the requesting task or another task. In the latter case, the receiving task must be in the DORMANT state.

GTSEG

The GTSEG directive allows a task to get a named segment of memory for itself or for another task from either ROM, RAM or memory mapped I/O space. There are also options pertaining only to the acquisition of RAM segments that allow the user to specify whether to acquire the segment from a specific partition number, from a specific type of partition, or to default to the type set up at initialization. The options also allow the user to specify the segments logical beginning address, to instruct the memory manager to make the logical beginning address equal to the physical beginning address and return that address to the task, or to allocate RAM memory at a specific physical address and make the logical address equal to the physical address. (This option overrides the specification of a partition number or type.)

Finally, the options allow the user to inform the memory manager what action to take if the RAM is not immediately available. These options include, return an error code, return the next largest piece of RAM, or wait until the memory becomes available. The following paragraphs explain this directive in more detail.

The name field allows the task to name a segment. The segment inherits the session number of the task that created it. Therefore, a segment has a segment name and session number in the same way that a task has a taskname and session number. The target task field of the GTSEG parameter block enables a task to obtain a segment for either itself or another task according to the target task interface rules. The length field tells the memory manager the segment size. If the segment length is a fraction of a page, the memory manager rounds the length up to the next page boundary.

Within the attributes field there are two bits defining the segment as either ROM or memory mapped I/O. If either of these bits is set, the memory manager allocates the memory from either ROM or memory mapped I/O at the physical address specified within the address field. It also makes the logical address of that segment identical to the specified physical address. If the ROM or memory mapped I/O attribute bit is set, none of the bits within the options field have any affect on the memory allocation.

If the ROM or memory mapped I/O attribute bit is not set, the segment is defined by default to be a RAM segment, and the entire range of options are in effect.

The GTSEG directive allocates RAM segments from memory partitions as described in paragraph 3.1. Bits 7 through 0 of the options field allow the user to specify a partition number or type (0 in either field indicates a don't care; an explicitly designated partition number takes precedence over a specifically designated partition type). Bit 7 of the option field must be set to specify a partition number or type. Usually it is enough to allow the system to allocate the memory from the default type designated at initialization.

The task may also designate the beginning logical address for the RAM segment using bit 13 of the options field. If bit 13 is 0, the segments beginning logical address is set to the value within the address field. Otherwise, the memory manager makes the beginning logical address equal to the beginning physical address and returns that value in register A0.

A task may also request a segment of RAM memory at a specific physical address by setting bit 8 of the options field. If this option is set, the GTSEG call behaves as if it was allocating a ROM or memory mapped I/O segment; it treats the address within the address field as a physical address, attempts to allocate a segment at that physical address, and if successful, sets the logical address equal to that physical address. The only difference between allocating a RAM segment at a physical address and allocating a ROM or memory mapped I/O segment is the response of the memory manager if the memory is not available. For ROM or memory mapped I/O, the memory manager returns an error, whereas for RAM, it responds in one of three ways as described below.

The last option of the GTSEG directive selects the action the memory manager should take when a request to allocate a RAM segment cannot be immediately satisfied because all or part of the requested memory is currently allocated to another segment. The task may select one of three options using bits 10 and 9 of the options field:

OPTIONS BITS 10/9	ACTION
00	Return an error code to the task and return size of largest available block in register A1.
01	Allocate largest available block and return size in register A1.
10	Put the task into a WAIT state until the memory becomes available.
11	Not defined.

Return Parameters:

Register A0	Physical address of the new segment.
Register A1	If options bit 9 = 1, then A1 contains the actual number of bytes allocated. Otherwise, A1 is returned unchanged.

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block is not in requestor's address space.
3/$03	Target task does not exist.
5/$05	Target task already has full segment allocation.
6/$06	Target task already has segment with specified name.
7/$07	Memory requested does not exist.
8/$08	Physical memory not available.
10/$0A	When requestor is not receiving task, receiving task is not in dormant state.
11/$0B	Logical address conflicts with existing segments.

EXAMPLE:

A user task, TSKA, wants to get a RAM segment, SEG1, of length $1000 bytes at logical address $20000. If the memory is not immediately available, TSKA indicates it is willing to wait.

```
TSKA:     MOVE.L    #1,D0          Load GTSEG directive number 1.
          LEA       PRMBLK,A0      Load parameter block address.
          TRAP      #1
          BNE       FAULT          Branch, if error.

PRMBLK:   DC.L      0              Task_id  of 0 means that segment is for
          DC.L      0              the calling task.
          DC.W      $0400          Wait until memory is available.
          DC.W      0              Get RAM memory.
          DC.L      'SEG1'         Call Segment "SEG1".
          DC.L      $20000         Logical address is $20000.
          DC.L      $1000          Segment length is $1000.
```

DETACH A SEGMENT — DESEG

Directive Number: 2

Parameter: Segment Block Address

Segment Block (refer to paragraph 3.2.6)

Target Task (8 bytes)	Task_id of task to lose segment. (Refer to target task interface, paragraph 1.3.6.)	
Directive Option (2 bytes)	Bits 15-12	Reserved
	Bit 11=1	Remove permanent status of a shareable segment.
	Bits 10-0	Reserved
Segment Attributes (2 bytes)	N/A	
Segment Name (4 bytes)	Segment to be detached.	
Logical Address (4 bytes)	N/A	
Segment Length (4 bytes)	N/A	

Detailed Description:

RMS68K deletes the specified segment from the target task's address space. If options bit 11=1, the permanent status is removed if the segment is shareable. If the segment is currently not shared by any other tasks and its status is not permanent, the memory is added back into the free memory list.

A task cannot delete a segment from its own address space if the User Stack Pointer (USP), which is stored in the task's ASQ, points within the segment to be detached. A task can detach another task's segment only if the other task is in the DORMANT state.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.
7/$07	Segment does not exist.
9/$09	When requestor is target, USP points within segment.
10/$0A	When requestor is not target, target is not in DORMANT state.

EXAMPLE:

A non real-time user task, TSKA, wants to deallocate a segment from another task, called TSKB. The name of the segment is SEGA. The segment is not shareable so options bit 11 is not applicable.

```
TSKA:           .
                .
                .
          MOVE.L   #2,D0          Load DESEG directive number 2.
          LEA      PRMBLK,A0      Load parameter block address.
          TRAP     #1
          BNE      FAULT          Branch, if error.
                .
                .
                .
PRMBLK:   DC.L     'TSKB'         Task to lose segment.
          DC.L       0            N/A; TSKA is a user task.
          DC.W       0            N/A; segment not shareable.
          DC.W       0            N/A
          DC.L     'SEGA'         Segment name in task to detach.
          DC.L       0            N/A
          DC.L       0            N/A
```

DECLARE A SEGMENT SHAREABLE — DCLSHR

Directive Number: 7

Parameter: Segment Block Address

Segment Block (refer to paragraph 3.2.6)

Target Task (8 bytes)	N/A	
Directive Options (2 bytes)	Bit 15=1	All segment attributes are specified (bits 14-12).
	=0	Only shareable attributes are specified (bits 13-12).
	Bits 14,13	Reserved
	Bit 12=1	Make shareable segment permanent.
	Bits 11-0	Reserved
Segment Attributes (2 bytes)	Bit 15	Reserved
	Bit 14=1	Segment is Read-only.
	=0	Segment is Read-write.
		Segment attribute bit 14 is applicable only if options bit 15=1.
	Bit 13=1	Segment is locally shareable.
	=0	Segment is not locally shareable.
	Bit 12=1	Segment is globally shareable.
	=0	Segment is not globally shareable.
	Bits 11-0	Reserved
		Either bit 12 or bit 13, but not both, must be set equal to one.
Segment Name (4 bytes)	Name of segment to be shareable.	

Logical Address (4 bytes)	N/A
Segment Length (4 bytes)	N/A

Detailed Description:

The DCLSHR directive is used to make a non-shareable segment, contained within the address space of the requesting task, into a shareable segment so that more than one task may attach to it. There are four types of shareable segments:

a.	Locally shareable	Not permanent
b.	Locally shareable	Permanent
c.	Globally shareable	Not permanent
d.	Globally shareable	Permanent

The DCLSHR directive can transform a non-shareable segment into any of the four types, but only a system task may declare a segment globally shareable.

In addition, DCLSHR can set the read-only, read-write status of the shareable segment, or propagate the current status. Once established by DCLSHR, all tasks that attach to the shareable segment are restricted to those permissions. Thus, the originating task can get the segment with read-write permissions and then declare it shareable with read-only permissions. The originating task may then read or write to that segment but any attached tasks may only read from it.

The segment that is to be made shareable must be in the address space of the requesting task and its name is specified in the name field of the parameter block. Any one of four types of shareable segments may be declared by the following combinations of options bit 12 and attributes bits 13 and 12.

OPTIONS BIT 12	ATTRIBUTES BITS 13/12	ACTION
0	01	Globally shareable/non permanent
0	10	Locally shareable/non permanent
1	01	Globally shareable/permanent
1	10	Locally shareable/permanent

(All other combinations are undefined.)

The read-only/read-write status within the global segment table may be set by the following combination of options bit 15 and attributes bit 14:

OPTIONS BIT 15	ATTRIBUTE BIT 14	ACTION
0	1 or 0	Same as current status within TST.
1	0	Segment is Read-write.
1	1	Segment is Read-only.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
5/$05	Global segment table is full.
6/$06	Segment name conflicts with segment that already exists in target task's address space.
7/$07	Segment does not exist.
9/$09	User task attempted to make segment globally shareable.
15/$0F	Attributes specify both global and local sharing, or neither.

EXAMPLE:

A user task, TSKA, wants to make a data segment, called SEGD, locally shareable to other tasks with the same session number. The segment is to be made permanent.

```
TSKA:          .
               .
               .
          MOVE.L   #7,D0          Load DCLSHR directive number 7.
          LEA      PRMBLK,A0      Load parameter block address.
          TRAP     #1
          BNE      FAULT          Branch, if error.
               .
               .
               .
PRMBLK:   DC.L       0            N/A
          DC.L       0            N/A
          DC.W     $1000          Make shareable segment permanent; segment
                                  is read/write.
          DC.W     $2000          Locally shareable.
          DC.L     'SEGD'         Name of shareable segment.
          DC.L       0            N/A
          DC.L       0            N/A
```

3

ATTACH A SHAREABLE SEGMENT — ATTSEG

Directive Number: 4

Parameter: Segment Block Address

Segment Block (refer to paragraph 3.2.6)

Target Task (8 bytes)	N/A	
Directive Options (2 bytes)	Bits 15,14	Reserved
	Bit 13=1	RMS68K supplies logical address of segment equal to physical address of segment.
	Bits 12,11	Reserved
	Bit 10=1	Length of segment to be attached is specified in the segment length field.
	Bits 9-0	Reserved
Segment Attributes (2 bytes)	Bits 15,14	Reserved
	Bit 13=1	Segment to attach is a locally shareable segment.
	Bit 12=1	Segment to attach is a globally shareable segment.
		Bits 11-0 Reserved
	Either bit 12 or bit 13 (but not both) must be set equal to one.	
Segment Name (4 bytes)	Name of the desired segment.	
Logical Address (4 bytes)	Logical address of segment within task's address space. Not applicable if options bit 13=1.	
Segment Length (4 bytes)	Length of segment to be attached. Applicable only if options bit 10=1. The value specified must be less than or equal to the actual length of the segment. If less, the first x bytes are attached.	

Detailed Description:

ATTSEG allows the requesting task control over the logical beginning address of the segment via options bit 13 and the address field. If option bit 13 is set, then the logical beginning address is equal to the physical address; otherwise, it is equal to the value of the address field within the parameter block.

ATTSEG allows the task to specify whether it needs access to all the shareable segment or some fraction starting from the beginning, via options bit 10 and the length field. If option bit 10 is 0, then ATTSEG gives the task access to the entire segment. Otherwise, ATTSEG creates a sub-segment of that segment starting at the beginning address and ending at the beginning address plus the value of the length field and gives the task access to that sub-segment.

Return Parameters:

Register A0 - physical address of segment.

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
5/$05	Requestor already has full segment allocation.
6/$06	Requestor already has segment with specified name.
7/$07	Segment not shareable or does not exist.
11/$0B	Logical address conflicts with requestor's address space.
16/$10	Invalid length field in parameter block.

EXAMPLE:

A user task, called TSKA, wants to add a globally shareable segment called SEGS, into its address space. The logical address of the segment is to be the same as the physical address.

```
TSKA:           .
                .
                .
          MOVE.L   #4,D0              Load ATTSEG directive number 4.
          LEA      PRMBLK, A0         Load parameter block address.
          TRAP     #1                 Branch, if error.
          BNE      FAULT
                .
                .
                .
PRMBLK:   DC.L     0                  N/A
          DC.L     0                  N/A
          DC.W     $2000              Logical address = physical address.
          DC.W     $1000              Segment to be globally shareable.
          DC.L     'SEGS'             Segment name.
          DC.L     0                  N/A; bit 13 set.
          DC.L     0                  N/A; bit 10 not set.
```

GRANT SHARED ACCESS TO ANOTHER TASK SHRSEG

Directive Number: 5

Parameter: Segment Block Address

Segment Block (refer to paragraph 3.2.6)

Target Task (8 bytes)	Task_id of task to receive segment. (Refer to target task interface, paragraph 1.3.6.)	
Directive Options (2 bytes)	Bits 15,14	Reserved
	Bit 13=1	RMS68K supplies logical address = physical address.
	Bits 12,11	Reserved
	Bit 10=1	Length of segment is specified in segment block.
	Bits 9-0	Reserved
Segment Attributes (2 bytes)	Bits 15,14	Reserved
	Bit 13=1	Segment to be attached is a locally shareable segment.
	Bit 12=1	Segment to be attached is a globally shareable segment.
	Either bit 12 or bit 13, but not both, must be set equal to one.	
	Bits 11-0	Reserved
Segment Name (4 bytes)	Segment to be attached.	
Logical Address (4 bytes)	Logical address of segment within target task's address space. Not applicable if options bit 13=1.	
Segment Length (4 bytes)	Length of segment to be attached. Applicable only if options bit 10=1. The value specified must be less than or equal to the actual length of the segment.	

Detailed Description:

SHRSEG allows the requesting task control over the logical beginning address of the segment via options bit 13 and the address field. If option bit 13 is set, then the logical beginning address is equal to the physical address; otherwise, it is equal to the value of the address field within the parameter block.

SHRSEG allows the task to specify whether it needs access to all the shareable segment or some fraction starting from the beginning, via options bit 10 and the length field. If option bit 10 is 0, then SHRSEG gives the task access to the entire segment. Otherwise, SHRSEG creates a sub-segment of that segment starting at the beginning address and ending at the beginning address plus the value of the length field and gives the task access to that sub-segment.

Return Parameters:

Register A0 - physical address of segment

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exit.
5/$05	Target task already has full segment allocation.
6/$06	Target task already has segment with specified name.
7/$07	Segment not shareable or does not exist.
9/$09	Requestor is specified as target task.
11/$0B	Logical address conflicts with target task's address space.

EXAMPLE:

A user task, TSKA, wants to place a shareable data segment in the address space of another task, TSKB. The segment, called SEGD, is a locally shareable segment, and is to have a logical address equal to its physical address.

```
TSKA:     MOVE.L   #5,D0          Load SHRSEG directive number 5.
          LEA      BLKADR,A0      Load parameter block address.
          TRAP     #1
          BNE      FAULT          Branch, if error.
            .
            .
            .
BLKADR:   DC.L     'TSKB'         Target task to receive segment.
          DC.L       0            N/A; user task.
          DC.W     $2000          Logical address=physical address; length
                                  not specified.
          DC.W     $2000          Locally shareable segment.
          DC.L     'SEGD'         Segment name to attach.
          DC.L       0            N/A; option bit 13 = 1.
          DC.L       0            N/A; option bit 10 = 0.
```

TRANSFER A SEGMENT — TRSEG

Directive Number: 3

Parameter: Segment Block Address

Segment Block (refer to paragraph 3.2.6)

Target Task (8 bytes)	Task_id of task to receive segment. (Refer to target task interface, paragraph 1.3.6.)	
Directive Options (2 bytes)	Bit 15=1	The attributes of the segment are changed according to the segment attribute field.
	Bit 14=1	Logical address supplied by requestor in segment block.
	Bit 13=1	RMS68K supplies logical address equal to physical address.
	If option bits 13 and 14 both are equal to zero, the logical address of the segment is the same as currently assigned in the requestor's address space.	
	Bits 12-0	Reserved
Segment Attributes (2 bytes)	Bit 15	Reserved
	Bit 14=1	Segment is to be Read-only.
	=0	Segment is to be Read-write.
	Bits 13-0	Reserved
	Shareable attributes cannot be changed.	
Segment Name (4 bytes)	Name of segment to be transferred.	
Logical Address (4 bytes)	New logical address needed if options bit 14=1.	
Segment Length (4 bytes)	N/A	

Detailed Description:

The TRSEG directive transfers a segment from the address space of the requesting task to the address space of the target task. There are two options, one allows the task to change the read-only/read-write attribute of the segment and the other allows the task to change the segments logical address. When the TRSEG directive is completed, the segment is no longer accessible to the requesting task.

The segment to be moved is named within the segment name field and the target task is designated by the target task identification field. If option bit 15 is set, then the read-only or read-write attribute of the segment is updated according to bit 14 of the segment attribute field. Bits 14 and 13 of the options field determine the new logical address:

OPTIONS BITS 14/13	NEW LOGICAL ADDRESS
00	No change to logical address.
01	New logical address equals physical address.
10	New logical address equals value of address field.
11	Not defined.

Return Parameters:

Register A0 - physical address of segment

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.
5/$05	Target task already has full segment allocation.
6/$06	Target task already has segment with specified name.
7/$07	Segment does not exist.
9/$09	Requestor's USP points within segment.
11/$0B	Logical address conflicts with target task's address space.

EXAMPLE:

A non real-time user task, TSKA, wants to transfer a data segment, called SEG1, into the address space of TSKB. The logical address is to be provided by RMS68K, and the segment is to be changed to a read-only segment.

```
TSKA:          .
               .
               .
          MOVE.L    #3,D0              Load TRSEG directive number 3.
          LEA       BLKADR,A0          Load parameter block address.
          TRAP      #1
          BNE       FAULT              Branch, if error.
               .
               .
               .
BLKADR:   DC.L      'TSKB'             Target task to receive segment.
          DC.L      0                  N/A; user task.
          DC.W      $A000              Segment   attributes   defined   below,
                                       logical address = physical address.
          DC.W      $4000              Segment to be Read-only.
          DC.L      'SEG1'             Segment name.
          DC.L      0                  N/A; options bit 14 = 0.
          DC.L      0                  N/A
```

RECEIVE SEGMENT ATTRIBUTES RCVSA

Directive Number: 9

Parameter: Segment Block Address

Segment block (refer to paragraph 3.2.6)

Target Task (8 bytes)	Task_id of task in whose address space the segment resides. (Refer to target task interface, paragraph 1.3.6.)	
	Bit 15	Reserved
Directive Option (2 bytes)	Bit 14=1	Segment identified by logical address in logical address field.
	=0	Segment identified by segment name.
	Bit 13=1	No information is returned in the user's buffer. The logical address of the segment referenced will be returned in Address Register 0.
	=0	All information about segment is returned in caller's buffer.
	Bits 12-0	Reserved
Directive Attributes (2 bytes)	N/A	
Segment Name (4 bytes)	Segment name. Applicable only if options bit 14=0.	
Logical Address (4 bytes)	Logical address within segment. Applicable only if options bit 14=1.	
Segment Length (4 bytes)	N/A	
Buffer Address (4 bytes)	Pointer to buffer where segment description is to be returned. The buffer must be 18 bytes in length.	

Detailed Description:

A task can obtain information about a segment using the RCVSA directive. The requesting task can request partial information (logical beginning address returned in a register), or total information (segment descriptor in a receive buffer), option bit 13. There are two ways of indicating the segment: by name or address.

The system "address" for the segment in question consists of two parts. First the target task identification fields point to a target task within the system. Then, depending on the status of option bit 14, RCVSA looks for a segment by name or logical address. If bit 14 equals 0, RCVSA looks for a segment whose name matches the value within the segment name field. Otherwise, it looks for one whose logical address range encompasses the logical address within the address field.

If the requesting task requires only the beginning logical address of the segment, it can set option bit 13 and the address is returned in register A0. Otherwise, a segment descriptor consisting of a name, attributes, beginning and ending logical address, and beginning physical address is returned in a buffer pointed to by the buffer address field in the format:

Segment Name (4 bytes)

Segment Attributes (2 bytes)

Bit 15	Reserved
Bit 14=1	Segment is Read only.
=0	Segment is Read-write.
Bit 13=1	Segment is locally shareable.
=0	Segment is not locally shareable.
Bit 12=1	Segment is globally shareable.
=0	Segment is not globally shareable.
Bit 11=1	Segment is memory mapped I/O space.
=0	Segment is not memory mapped I/O space.
Bit 10=1	Segment is physical ROM.
=0	Segment is not physical ROM.
Bits 9-0	Reserved

Beginning Logical Address (4 bytes)

Ending Logical Address (4 bytes)

Physical Address (4 bytes)

Return Parameters:

Segment information is described above at location specified in segment block.

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.
7/$07	Segment does not exist.
12/$0C	Receiving buffer not in requestor's address space.

EXAMPLE:

A non real-time user task, TSKA, wants to determine the attributes of segment SEG1, as it resides in the address space of TSKB. The information is to be given to TSKA starting at location RCVBUF.

```
TSKA:        .
             .
             .
            MOVE.L   #9,D0           Load RCVSA directive number 9.
            LEA      PRMBLK,A0       Load parameter block address.
            MOVE.L   #RCVBUF,BUFADR  Modify parameter block buffer address.
            TRAP     #1
            BNE      FAULT           Branch, if error.
             .
             .
             .
PRMBLK:     DC.L     'TSKB'          Target task segment information from.
            DC.L      0              N/A; user task.
            DC.W     $0000           Segment identified by name;
                                     information returned in buffer.
            DC.W      0              N/A
            DC.L     'SEG1'          Segment name.
            DC.L      0              N/A; bit 14-0.
            DC.L      0              N/A
BUFADR:     DC.L      0              Pointer to buffer.
RCVBUF:     DS.B     18              Segment description return buffer.
```

MOVE FROM LOGICAL ADDRESS — MOVELL

Directive Number: 6

Parameter: Parameter Block Address

Parameter Block:

Source Task (8 bytes)	Task_id of task that contains address from which data is being moved. Requesting task is assumed if this field is 0. (Refer to paragraph 1.3.6.)
Source Logical Address (4 bytes)	Logical address within source task's address space where data to be moved resides.
Target Task (8 bytes)	Task_id of task that contains address to which data is being moved. Requesting task is assumed if this field is 0.
Destination Logical Address (4 bytes)	Logical address within destination task's address space where data is to be moved.
Length of Data Block (4 bytes)	The number of bytes in the data block to be moved.

Detailed Description:

A block of data is moved from one logical address to another. A user task may only move data to other tasks within its own session, and cannot move data to a system task's address space.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block is not in requestor's address space.
3/$03	Source task does not exist.
7/$07	Destination task does not exist.
9/$09	User task cannot move data to a system task.
11/$0B	Trying to move from an odd address to an even address or from an even address to an odd address.
12/$0C	Source logical address not in address space of source task.
13/$0D	Destination logical address not in address space of destination task.

EXAMPLE:

A non real-time user task, TSKA, wants to move an 8-byte block of data residing at location SRCDAT into the address space of TSKB at location DSTDAT. Both TSKA and TSKB are in the same session.

```
TSKA:       .
            .
            .
            MOVE.L   #6,D0        Load MOVELL directive number 6.
            LEA      PRMBLK,A0    Load parameter block address.
            TRAP     #1
            BNE      FAULT        Branch, if error.

            .
            .
            .
            DC.L     0            Source taskname (requesting task).
            DC.L     0            N/A; user task.
            DC.L     SRCDAT       Address from which to start data
                                  transfer.
            DC.L     'TSKB'       Destination task.
            DC.L     0            N/A; user task.
            DC.L     DSTDAT       Where to place data.
            DC.L     8            Length of data block in bytes.
```

MOVE FROM PHYSICAL ADDRESS — MOVEPL

Directive Number: 72

Parameter: Parameter Block Address

Parameter Block:

Not Used (8 bytes)	Reserved
Source Physical Address (4 bytes)	The actual physical address at which the data to be moved resides.
Target Task_id (8 bytes)	Task_id of task that contains address to which data is being moved. (Refer to paragraph 1.3.6.)
Destination Logical Address (4 bytes)	Logical address within destination task's address space where data is to be moved.
Length of Data Block (4 bytes)	The number of bytes in the data block to be moved.

Detailed Description:

A block of data is copied from a physical address to a logical address within the destination task's address space.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block is not in requestor's address space.
7/$07	Destination task not found.
11/$0B	Trying to move from an odd address to an even address or from an even address to an odd address.
12/$0C	A bus error occurred in trying to access the physical address specified.
13/$0D	Destination logical address not in the address space of the destination task.

MOVEPL

EXAMPLE:

A task wants to copy a 256-byte block of data from physical address $900 into a buffer labeled SYSBUF within its own address space.

```
TSKA:           .
                .
                .
               MOVE.L   #72,D0         Load MOVEPL directive number 72.
               LEA      PRMBLK,A0      Load parameter block address.
               TRAP     #1
               BNE      FAULT          Branch, if error.
                .
                .
                .
PRMBLK:        DC.L     0,0            Reserved
               DC.L     $900           Address data location.
               DC.L     0,0            Requesting task, requesting session.
               DC.L     SYSBUF         Local address in which to store data.
               DC.L     256            Number of bytes to move.

SYSBUF:        DC.B     256
```

FLUSH USER CACHE FLUSHC

Directive Number: 75

Parameter: None

Detailed Description:

RMS68K flushes user mode entries from all caches known to it. This includes instruction caching and data caching.

Return Parameters: None

Error Codes: None

EXAMPLE:

TSKA wants to modify its own code space and execute it. Because the code has already been executed, the cache must be cleared in case it contains entries corresponding to the overwritten addresses.

```
TSKA:        BSR     SOME_ROUTINE     Call the routine, possibly caching
                                      entries.
              .
              .
              .                       (Change the code in SOME_ROUTINE.)

              .
              .
              .
             MOVE.L  #75              Clear the cahe of all user mode
                                      entries.
             TRAP     #1
             BSR     SOME_ROUTINE     Call the routine again without fear of
                                      getting invalidated entries from cache.
```

CHAPTER 4

TASK MANAGER

4.1 OVERVIEW

The Task Manager consists of those directives and data structures that support the concept of tasks as resources that can be created, terminated, started, stopped, temporarily halted, reawakened, and inquired about. The task manager maintains a list of all tasks currently known to the system and the TCBs that contain information describing the current state and resources allocated to each task. The task manager imposes no limit on the number of tasks within a system; the limiting factors are typically the amount of available memory and the application's performance requirements.

The task manager functions are divided into three categories and are described in detail in this chapter:

a. Task Initialization and Termination

b. Task Synchronization

c. Task Query

4.2 THEORY OF OPERATION

4.2.1 Tasks

A task is a function that can execute concurrently with other functions within a multitasking environment. A task typically accepts one or more inputs, performs some processing function based on the input, and responds with one or more outputs.

Two typical classes of tasks are data processing and control tasks. The input and output to a data processing task is data, usually as events. A control task accepts input from the external word as signals, external interrupts, or events, decides on an appropriate response in light of its world view and the recent behavior of the system, and responds, usually by manipulating a hardware device that causes some change to occur within the external world. These are only two examples of tasks. There are many more types that combine elements of control and data processing.

4.2.1.1 <u>Task Structure</u>. The minimum configuration of a task is a TCB containing the system's knowledge of the task, and one program code memory segment containing the task's procedural knowledge of how to do the function. Optionally, a task can possess up to three additional memory segments (program code or data) and one ASQ.

Task Control Block (TCB)

Associated with each task currently in the system is a TCB. The TCB contains information about the task that allows RMS68K to maintain control of the task's execution, account for resources allocated to the task, and ensure task protection. The TCB remains associated with one task throughout the task's existence.

All RMS68K's resource managers maintain their own specific fields within the TCB. The task manager is responsible for creating and initializing the TCB and purging it from the system on task termination.

Program Code Segment

A Program Code Segment contains instructions used during execution and is normally marked read-only, although it can be marked read-write if it also contains data. A task's program code segment can be divided into independent sections to provide an atmosphere conducive to event processing. Although RMS68K is not aware of section boundaries, it can be advised of the entry points of the various sections; this allows RMS68K to exercise control over the dispatching function in an asynchronous environment.

Task Level Program Code

The task level code is subdivided into categories:

a. Main Code

The basic element of a program code segment. The main line code is executed from a task's initial entry point when the task is started.

b. Trap Handling Code

Allows a task to respond to its own trap instruction. A task can specify that trap instructions 2 through 15 be handled by normal default processing or by trap handling code starting at a specific location within the task.

c. Exception Handling Code

Similar to trap handling code, except that a task can elect to process many of its own exceptions.

d. Interrupt Service Routine (ISR)

An external interrupt activates the ISR. Although the ISR code is part of a task and totally shares its address space, the ISR executes independently of the task. Therefore, an ISR can run concurrently with the task and functions like a user-mode extension of the Executive.

When RMS68K enters an ISR, the ISR should do a minimum amount of processing and then exit. On exit, the ISR can activate (or reactivate) the task in which it is included, usually for the purpose of processing results in a background mode.

Directives cannot be issued during interrupt handling except when returning from ISR processing. Any exception that occurs during ISR processing ends ISR execution.

e. Asynchronous Service Routine (ASR)

The ASR is the service routine for a software interrupt, analogous to the ISR that is the service routine for a hardware interrupt. The ISR is executed in response to an external hardware event (interrupt), whereas the ASR is executed in response to an external software event (some other task queued an event to the task).

f. Directive Handling Code

The Directive Handling Code allows a task to respond to its own or other task's TRAP #1 instructions. A task's Directive Handling Code executes in supervisor mode without MMU protection and allows a task to dynamically extend RMS68K's directive set.

Data Segment

A Data Segment is used by a task for working storage, for passing bulk data to other tasks, and for two or more tasks to share a common data area.

One data segment is usually allocated to a task during task initiation, however, additional segments can be allocated by various memory management directives.

Asynchronous Service Queue (ASQ)

The ASQ holds events that are waiting to be processed by the task. Refer to paragraph 2.2.2 for details.

4.2.1.2 Task Identification. Associated with each task is a taskname and a session number. The taskname can be any 4-byte value except 0, and the session number can be in the range $0000 to $FFFF, inclusive. (Refer to paragraph 1.3.6 for details on how the taskname and session number are used within the target task interface protocol.)

4.2.1.3 Task Priority. A task's priority is a measure of the task's importance relative to all other tasks within the system and indicates its "need to run" in a multitasking system where many tasks may be "ready to run" at any moment.

When a task is created, it is given an initial current priority and a limit priority. The current priority can be changed at any time to a value less than or equal to that task's limit priority via the SETPRI directive. The priority is any value 0 to 255 (inclusive) with 0 being the lowest priority. Tasks in the READY state are dispatched for execution during a dispatch cycle of RMS68K. The task with the highest priority residing in the READY state is selected for dispatch. If more than one task has the highest priority, the task that has been in the READY state the longest is selected.

4.2.1.4 Monitor and Subtasks. A monitor task can be set up to automatically receive notification of the termination of another task, referred to as a subtask of the monitor. A monitor task can monitor any number of subtasks and does not require the same session number as its subtask.

When a task is created or started, options specify which task, if any, is its monitor. The monitor can be assigned as the task requesting the creation or start, the requestor's monitor, or a third task. When a subtask terminates, the task manager places an event in the monitor task's ASQ, specifying the subtask identification, the task that initiated the termination, and a normal or abnormal termination indicator.

A monitor task should not be confused with an exception monitor that receives notification when its target task causes an exception, such as divide by zero or a bus error.

4.2.2 Task Initialization and Termination

The Task Initialization and Termination functions within the task manager provide services to support the most significant events within a task's life. A task does not exist in the system until a TCB is created. Once the TCB is created, other tasks can refer to it and act on its behalf in allocating and deallocating resources, but the task cannot function on its own. Once a task has been started, it can execute its function, vie with other tasks for processor time according to its relative priority, allocate and deallocate resources for itself or other tasks, and exercise all rights and privileges according to its rank and station within the system (system/user task, real-time/non real-time task, session zero/not session zero).

If a task is stopped, it remains in the system but is incapable of acting on its own behalf. When a task is terminated or aborted, all resources currently allocated to it (including its TCB) are released back to the system, all knowledge of its existence is purged from the system, and any further directives referring to it are rejected with an error code of $03 (target task does not exist).

The directives supporting task initialization and termination are:

CRTCB	Create a TCB for a new task, initialize the name, session number, monitor, priority, attributes and entry point fields, and place the task in the DORMANT state.
START	Move the target task from the DORMANT to the READY state. This directive can also initialize the task's registers and specify its monitor.
SETPRI	A task changes its own priority or the priority of another task to the specified value according to the restrictions described in task priority (refer to paragraph 4.2.1.3).
STOP	Move the target task to the DORMANT state.
TERMT	Terminate a target task by releasing resources and deleting the TCB of the target task so that the task no longer exists in the system.
TERM	Terminate the requesting task by releasing all resources and deleting its TCB so that the task no longer exists in the system.
ABORT	Initiate an abnormal termination of the requesting task; its TCB is deleted and the task no longer exists in the system.

4.2.3 Task Synchronization

A "primitive level" of task synchronization based on the "Wait for a signal from another task" concept is provided by the WAIT/WAKEUP and SUSPEND/RESUME directives.

WAIT	A task moves itself into the WAIT state.
WAKEUP	Move the target task from the WAIT state to the READY state.
SUSPND	A task moves itself into the SUSPEND state.
RESUME	Move the target task from the SUSPEND state to the READY state.
RELINQ	A task moves itself to the READY state forcing RMS68K to execute a dispatch cycle.

The WAIT/WAKEUP pair of directives support a one-deep buffer for the signal, allowing the signalling task to send the signal (WAKEUP the target) before the waiting task indicates its intention to wait on that signal. This state is known as WAKE-UP PENDING; a task that executes a WAIT directive while in the WAKE-UP PENDING state continues executing immediately after the WAIT.

The SUSPEND/RESUME pair does not buffer the signal, instead the signalling task must issue the RESUME directive after the waiting task executes the SUSPEND for the waiting task to be moved from the SUSPEND state back to the READY state.

The RELINQ directive allows a task to relinquish control of the processor to tasks of equal or slightly lower priority. A task of priority $NM where N and M are both between 0 and $F (inclusive) that executes a RELINQ directive, is placed back on the READY list at priority $N0 and runs again after all other READY tasks of priority $N0 or greater execute. (The RELINQ directive does not affect a task's current priority.)

4.2.4 Task Query

The task manager supports four task query directives:

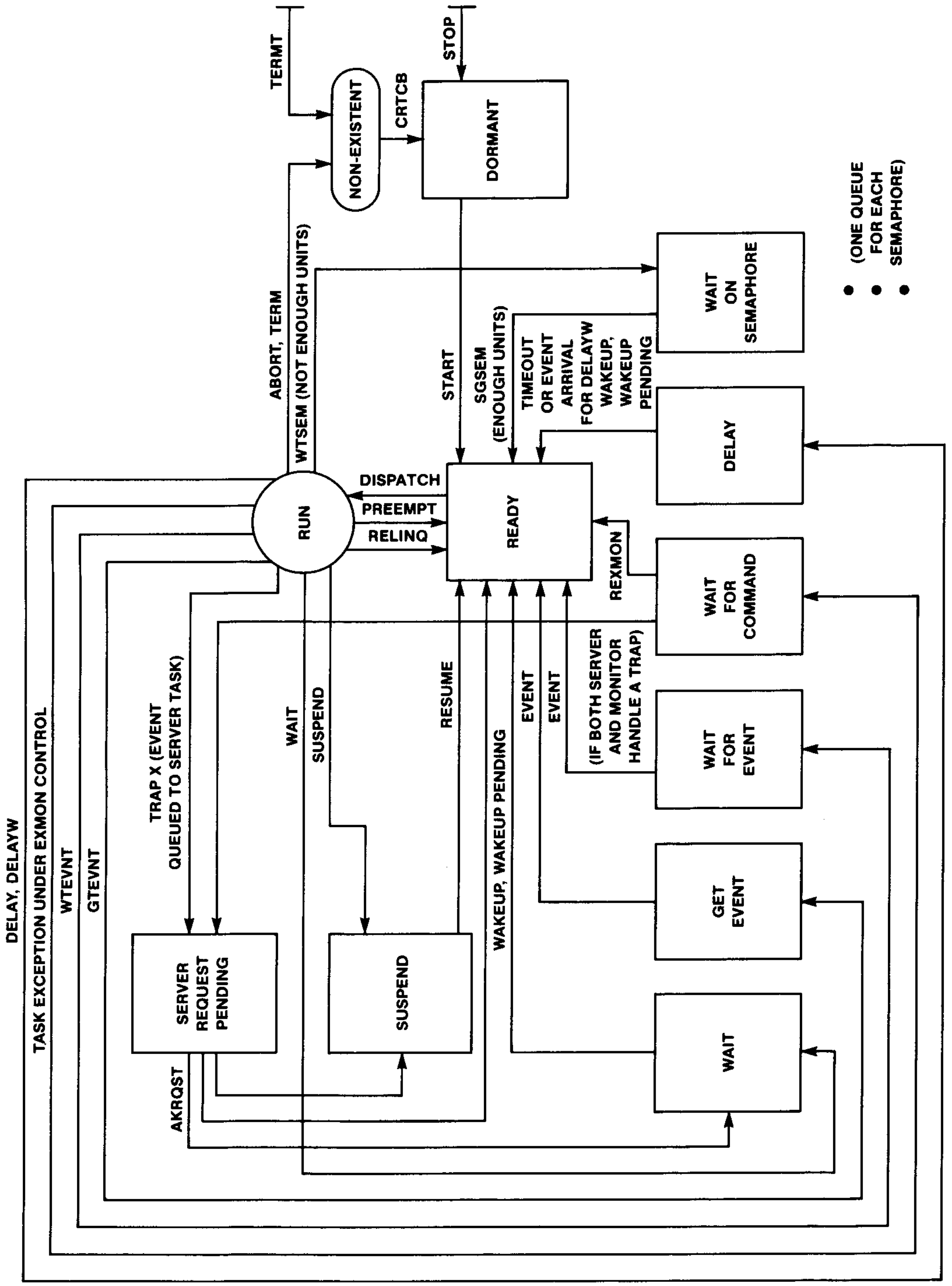
TSKATTR	A task receives the user number and attributes of a target task.
TSKINFO	A system task requests a copy of a target task's TCB.
GTTASKID	A task translates a target task's taskname and session number into its task_id.
GTTASKNM	A task translates a target task's task_id into its taskname and session number.

The TSKATTR and TSKINFO directives are self-explanatory. GTTASKID supports the target task interface in either the real-time or non real-time domain of execution. The input to GTTASKID is the target task's taskname and session number and the output is the task_id appropriate for the requesting task's domain of execution (i.e., if the requesting task is a non real-time task, the task_id is the original taskname and session number; otherwise, it is an internal 8-byte code required for real-time tasks to access target tasks).

The GTTASKNM directive converts a target task's task_id into a taskname and session number.

4.2.5 Task State Transitions

Figure 4-1 is a diagram of all the task state transitions caused by any resource manager.


TERMT
NON-EXISTENT
CRTCB
STOP
DORMANT
START
ABORT, TERM
WTSEM (NOT ENOUGH UNITS)
SGSEM (ENOUGH UNITS)
TIMEOUT OR EVENT ARRIVAL FOR DELAYW WAKEUP, WAKEUP PENDING
WAIT ON SEMAPHORE
(ONE QUEUE FOR EACH SEMAPHORE)
DELAY
DISPATCH
PREEMPT
RELINQ
RUN
READY
REXMON
WAIT FOR COMMAND
(IF BOTH SERVER AND MONITOR HANDLE A TRAP)
EVENT
EVENT
WAIT FOR EVENT
GET EVENT
WAIT
RESUME
WAKEUP, WAKEUP PENDING
WAIT
SUSPEND
TRAP X (EVENT QUEUED TO SERVER TASK)
DELAY, DELAYW
TASK EXCEPTION UNDER EXMON CONTROL
WTEVNT
GTEVNT
SERVER REQUEST PENDING
SUSPEND
AKRQST

The dispatch cycle of RMS68K is entered any time a task is removed from the RUN state. There are many reasons for a task being removed from the RUN state, several of which are:

a. A task relinquishes execution.

b. A task directly changes the task state of itself or any other task.

c. An event is placed in any task's ASQ (because of direct request for queuing, exception monitor event, physical I/O return event, etc.).

d. A task performs a semaphore wait operation.

e. Task execution time of a task exceeds the maximum timeslice allowed (if timeslicing is installed).

f. A higher priority task becomes READY (pre-emptive dispatch).

When a task is removed from the RUN state for any reason other than a STOP, ABORT, TERMT, or TERM directive, the task resumes execution at the next instruction following the last instruction that was executed in that task. The first time a task is executed via a START directive, execution begins at the task entry point.

4.3 DATA STRUCTURE

The TCB is allocated, initialized, manipulated, and de-allocated by the task manager. Other resource managers also manipulate fields within the TCB.

4.3.1 Task Control Block (TCB)

The TCB controls the execution of the relevant task.

TCB	(4 bytes)	Block ID Each TCB begins with '!TCB' to allow consistency checking and ease of dump reading.
TCBALL	(4 bytes)	TCB list link Points to the next TCB in the singly-linked list of all TCBs. Zero represents end of list.
TCBGROUP	(4 bytes)	Reserved for future use.

TCBREADY	(4 bytes)	Ready list link Points to the next ready-to-execute TCB in the singly-linked ready list. Zero represents end of list.
TCBNAME	(4 bytes)	Taskname The name of the task represented by this TCB.
TCBSESSN	(4 bytes)	Session code The session code of the task represented by this TCB.
TCBMON	(8 bytes)	Monitor ID The taskname and session code of this task's monitor. Zero indicates no monitor.
TCBSEM	(4 bytes)	Semaphore wait link If this task is blocked on a semaphore wait, this field points to the next TCB blocked on the same semaphore. Zero represents end-of-wait list.
TCBCPRI	(1 byte)	Current priority The software priority of this task.
TCBLPRI	(1 byte)	Limit priority The highest software priority assignable to this task.
TCBRPRI	(1 byte)	Ready list priority Sometimes, the Executive temporarily alters the priority of a task; this field places a task on the READY list.
TCBIOCNT	(1 byte)	Pending I/O count This field is incremented with each initiation of an input or output operation that transfers data to or from this task's address space; it is decremented with each completion.

TCBATTR	(2 bytes)	Task attributes See the TSKATTR directive in paragraph 4.4 for attributes definition.
TCBABORT	(2 bytes)	Abort code TCBABORT indicates the reason this task is aborting.
TCBSTATE	(4 bytes)	Current task state The state of a task is saved in the high order 2 bytes of the TCBSTATE field. If the task is stopped by a STOP directive, it is moved to the low order 2 bytes.

State bit definitions:

15=1	Task is DORMANT. It has not been started or it has been stopped.
14=1	Task is waiting. It can be reactivated by a WAKEUP.
13=1	Task is in a semaphore wait list.
12=1	Task is waiting for an event.
11=1	Task is waiting for an acknowledgement from a server task.
10=1	Task is waiting to be reactivated by an exception monitor.
9=1	Task is suspended. It can be reactivated by a RESUME.
8=1	Not defined.
7=1	Task is being terminated.
6=1	Task returns to Executive when next dispatched.
5=1	Dispatch task to ASR routine.
4=1	Task is on READY list.
3=1	Task has a pending WAKEUP.
2=1	Termination message sent to server task.
1	Reserved.
0	Reserved.

TCBTSTSM	(6 bytes)	TST semaphore Regulates access to this task's TST.
TCBTST	(4 bytes)	TST pointer Points to this task's TST.
TCBASQSM	(6 bytes)	ASQ semaphore Regulates access to this task's ASQ.
TCBASQ	(4 bytes)	ASQ pointer Points to this task's ASQ. Zero indicates no ASQ in existence for this task.
TCBCHAN	(4 bytes)	Channel Control Block (CCB) list head Points to the first CCB attached to this task. Zero indicates no CCBs attached.
TCBEVECT	(4 bytes)	Exception vector pointer Contains the logical address of this task's exception vector table. Zero indicates no own exception handling.
TCBTVECT	(4 bytes)	Trap Vector Pointer Contains the logical address of this task's trap vector table. Zero indicates no own trap handling.
	(8 bytes)	Reserved for future use.
TCBDLAY	(4 bytes)	Address of delay entry in PAT.
	(2 bytes)	Reserved for future use.
TCBISRS	(2 bytes)	ISR error code Used as return code for WAKEUP following a user interrupt.
	(12 bytes)	Reserved for future use.

TCBENTRY	(4 bytes)	Entry point Contains the logical address of this task's entry point.
TCBUSER	(2 bytes)	User number Contains the user number under which this task is executing.
TCBSSP	(1 byte)	Super stack depth If this task is in the "Return to Executive" state, this field contains the depth of the supervisor stack save area.
TCBUTRP	(1 byte)	User trap The user trap instruction number (not including 0 or 1) currently being processed for this task. Field remains until another trap is processed.
TCBXREGS	(60 bytes)	Executive registers If this task is in the "Return to Executive" state, TCBXREGS contains the registers content to be restored (D0 to D7 and A0 to A6).
TCBATSK	(4 bytes)	Terminator task If this task is terminating, TCBATSK contains the name of the task that initiated termination.
TCBASES	(4 bytes)	Terminator session The session code of the task that initiated termination.
TCBBERR	(8 bytes)	Error status If this task executed a bus error or address error exception, TCBBERR contains the error status information from the super stack.
	(64 bytes)	Supervisor stack save area

TCBD0	(64 bytes)	User Registers TCBD0 through TCBUSP contain the task registers (D0 to D7 and A0 to A7).
TCBSR	(2 bytes)	User SR Contains the task's status register.
TCBPC	(4 bytes)	User PC Contains the task's program counter.
TCBVOR	(2 bytes)	User Vector Offset Register (VOR) Contains the task's VOR.
TCBRR0	(2 bytes)	Error code Error code returned to task from user ISR.
	(80 bytes)	Task Segment Table (TST)
	(48 bytes)	Reserved for future use.
TCBEXM	(4 bytes)	Exception monitor taskname
TCBEXMS	(4 bytes)	Exception monitor session number
TCBEMMSK	(4 bytes)	Exception monitor mask
TCBEVMSK	(4 bytes)	Exception monitor value mask
TCBEVLOC	(4 bytes)	Exception monitor value address
TCBEVALU	(4 bytes)	Exception monitor value content
TCBECNT	(4 bytes)	Exception monitor maximum number of instructions
	(4 bytes)	Reserved for future use.

4.4 TASK MANAGER DIRECTIVES

The following pages contain detailed descriptions and examples of the task manager directives.

CREATE TASK CONTROL BLOCK (TCB) CRTCB

Directive Number: 11

Parameter: TCB Block Address

TCB Block:

Taskname (4 bytes)	Name of new task.
Session (4 bytes)	N/A if requestor is a user task.
Directive Options (2 bytes)	Bit 15=1 New task's monitor is specified in monitor fields of TCB block. Bit 14=1 New task's monitor is requesting task's monitor. Bits 13-0 Reserved If both bits 14 and 15 are set to 1, RMS68K chooses the monitor-specified option (bit 15=1). If both bits 14 and 15 are reset to 0, the target task does not have a monitor task.
Monitor Taskname (4 bytes)	This field is used only when options bit 15=1. If this field is 0, the new task's monitor is the requesting task. Otherwise, the monitor is the task specified in this field. WARNING SUBTASK TERMINATION EVENTS (CODE = $05) ARE 24 BYTES LONG. IF THE MONITOR HAS AN ASQ, THEN THE ASQ'S MAXIMUM MESSAGE LENGTH MUST BE AT LEAST 24 BYTES.
Monitor Session (4 bytes)	This field is used only when options bit 15=1 and the requesting task is a system task. If the field has the value 0, the session of the new task's monitor is the requestor's session. Otherwise, it is assigned to the value of this field.
Initial Priority (1 byte)	Initial priority to be assigned to the new task; 0 to 255 (0 = lowest).
Limit Priority (1 byte)	Highest priority that can be assigned to the new task; 0 to 255 (0 = lowest). The maximum limit is the priority of the calling task.

Task Attributes (2 bytes)	Bit 15=1	New task is a system task.
	Bit 14	Reserved
	Bit 13=1	Crash system if new task terminates abnormally.
	Bit 12=1	Task dump if new task terminates abnormally.
	Bit 11=1	Relocatable task running with no MMU. Entry address is adjusted when task is started.
	Bits 10-0	Reserved
Task Entry Point (4 bytes)	Task level code logical address to which control is transferred when new task is executed.	
User Generated I.D. (2 bytes)	This field is not used by RMS68K; it is for the user's information only. It appears in the event message to a server task when this task makes a request to the server task.	

Detailed Description:

RMS68K allocates 512 bytes of memory for the TCB of the new task. The TCB is initialized according to the information in the parameter block. A monitor task for the new task can be assigned, and initial and limit priorities for the new task are also assigned at this time. A user task cannot specify the new task to be a system task. The new task is in the DORMANT state.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
6/$06	Taskname already exists.
8/$08	Memory not available.

EXAMPLE:

A system task, TSKA, wants to create a new system task, TSKB, within the same session. The first thing TSKA must do is to create the TCB of TSKB. TSKA is to be TSKB's monitor, and TSKB is to have an initial priority of 100 with a limit priority of 150. The entry point of TSKB is to be at logical location BSTRT.

```
TSKA:          .
               .
               .
          MOVE.L    #11,D0        Load CRTCB directive number 11.
          LEA       PRMBLK,A0     Load parameter block address.
          TRAP      #1
          BNE       FAULT         Branch, if error.
               .
               .
               .
PRMBLK:   DC.L      'TSKB'        Name of new task.
          DC.L      0             New  task  to have same session as this
                                  system task.
          DC.W      $8000         Bit 15 set; monitor specified below.
          DC.L      0             New  task  monitor  is  this requesting
                                  task.
          DC.L      0             Session at new task is this session.
          DC.B      100           Initial priority.
          DC.B      150           Limit priority.
          DC.W      $8000         New  task  is  a  system  task;  do not
                                  crash, no dump, MMU.
          DC.L      BSTRT         Entry address of new task.
          DC.W      0             User-generated ID.
```

START TASK | START

Directive Number: 13

Parameter: Parameter Block Address

Parameter Block:

Target Task (8 bytes)	Task_id of target task to be executed. If the field is 0, the START directive looks for a task to re-start. It can also re-start a task that has been STOPped.
Directive Options (2 bytes)	Bit 15=1 The monitor of the target task is specified in the monitor fields of the parameter block. Bit 14=1 The monitor of the target task is the requesting task's monitor. If both bits 14 and 15 are set to 1, RMS68K honors the monitor-specified option (bit 15=1). If both bits 14 and 15 are reset to 0, the monitor task of the task being started remains unchanged. Bit 13=1 The registers of the task being started are to be initialized to the values in the registers field of the parameter block. Bits 12-0 Reserved
Monitor Taskname (4 bytes)	This field is used only when options bit 15=1. If this field is 0, the monitor of the task being started is the requesting task; otherwise, the monitor is the task specified in this field.

WARNING

SUBTASK TERMINATION EVENTS (CODE = $05) ARE 24 BYTES LONG. IF THE MONITOR HAS AN ASQ, THEN THE ASQ'S MAXIMUM MESSAGE LENGTH MUST BE AT LEAST 24 BYTES.

Monitor Session (4 bytes)	This field is used only when options bit 15=1 and the requesting task is a system task. If this field has the value 0, the session of the monitor of the task being started is the requestor's session; otherwise, the monitor's session is specified by the value in this field.
Registers (60 bytes)	Used only if options bit 13=1. This field contains the initial values of registers D0 to D7 and A0 to A6 to be assigned to the registers of the task being started.

Detailed Description:

RMS68K puts the target task into the READY state, based on its current priority, to wait for execution. A task can be started only from the DORMANT state. The monitor task can be assigned or changed and the initial values of the target task's registers can be assigned.

When the START directive is used to re-start a task that has been stopped, the task's monitor is not changed.

The START directive, with taskname set equal to 0, can be called repeatedly to re-start all tasks in the caller's session stopped by the STOP directive.

Return Parameters:

If START was called with taskname = 0, the taskname of the started task is returned in A0.

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requesting task's address space.
3/$03	Task does not exist.
6/$06	Duplicate request; task started.
9/$09	User task cannot start system task.

EXAMPLE:

A non real-time user task, TSKA, wants to start the execution of a second task, TSKB. TSKB is to have a monitor task, called TSKC, and all TSKB's registers are to be initialized to 0.

```
TSKA:       .
            .
            .
            MOVE.L    #13,D0         Load START directive number 13.
            LEA       PRMBLK,A0      Load parameter block address.
            TRAP      #1
            BNE       FAULT          Branch, if error.
            .
            .
            .
PRMBLK:     DC.L      'TSKB'         Target task to START.
            DC.L      0              N/A; user task.
            DC.W      $A000          Monitor  of task specified; registers to
                                     be set.
            DC.L      'TSKC'         Monitor taskname.
            DC.L      0              N/A; requesting task is a user task.
RD0:        DC.L      0              Initial value of D0.
RD1:        DC.L      0              Initial value of D1.
 .
 .
 .
RD7:        DC.L      0              Initial value of D7.
RA0:        DC.L      0              Initial value of A0.
RA1:        DC.L      0              Initial value of A1.
            .                              .
            .                              .
            .                              .
RA6:        DC.L      0              Initial value of A6.
```

NOTE

The user stack pointer (A7) must be initialized by the executing task. If the target task has a current priority greater than the limit priority of the requesting task, the target task is started at the requesting task's limit priority.

SET PRIORITY SETPRI

Directive Number: 24

Parameter: Parameter Block Address

Parameter Block:

Target Task (8 bytes)	Task_id of target task with changing priority. Requesting task is assumed if this field is 0.
New Priority (1 byte)	New current priority.

Detailed Description:

RMS68K changes the current priority of the target task to the value specified. The new priority must be less than or equal to the limit priority of the target task priority set at the time of TCB creation. A task can change its own priority or the priority of another task. A user task cannot alter the priority of a system task.

Return Parameters:

If error code 10/$A is returned in D0, then the target task's limit priority is returned in A0.

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.
9/$09	User task cannot alter priority of system task.
10/$0A	Specified new priority greater than limit priority.

EXAMPLE:

Supervisor task, TSKA, wants to change its own priority to the value 5.

```
TSKA:        .
             .
             .
        MOVE.L   #24,D0              Load SETPRI directive number 24.
        LEA      PRMBLK,A0           Load parameter block address.
        TRAP     #1
        BNE      FAULT               Branch, if error.
             .
             .
             .
PRMBLK: DC.L       0                 Requesting task is target task.
        DC.L       0                 Same session as requesting task.
        DC.B       5                 New current priority.
```

STOP TASK — STOP

Directive Number: 25

Parameter: Parameter Block Address

Parameter Block

Target Task (8 bytes)	Task_id of target task to be halted.

Detailed Description:

RMS68K stops execution of the target task and moves it to the DORMANT state with all resources still attached. The task remains in memory. A user task cannot stop a system task.

This directive operates in two modes:

a. Stop Single Task

b. Stop Session

Stop Single Task Mode

The requestor must specify the task_id of the task to be stopped. System tasks are only stopped in the stop single task mode.

Stop Session Mode

The stop session mode is used by the requesting task to stop all user tasks within a given session. A user task can only stop tasks within its own session. For this mode, the requestor does not specify the task_id (task_id = 0). RMS68K selects one task from the relevant session and places it in the DORMANT state. Thus, a task could stop all user tasks in a session by issuing the STOP directive repeatedly until it is the only task remaining in the READY state (the requestor is immune to the STOP directive).

Return Parameters:

A0	Name of stopped task

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.
6/$06	Specified task already in DORMANT state.
9/$09	User task cannot stop system task.

EXAMPLE:

A user task, TSKA, wants to stop all user tasks within its session.

```
TSKA:      .
           .
           .
STP:       MOVE.L   #25,D0        Load STOP directive number 25.
           LEA      PRMBLK,A0     Load parameter block address.
           TRAP     #1
           BEQ      STP           Branch, if successful, to STP; do it
                                  again.
           CMP.L    #3,D0         Test if all gone.
           BNE      FAULT         Branch, if error.
           .
           .
           .
PRMBLK:    DC.L     0             Any task.
           DC.L     0             N/A; user task.
```

TERMINATE SELF TERM

Directive Number: 15

Parameter: None

Detailed Description:

RMS68K halts execution of the requesting task and removes the task from memory. This results in the normal termination of task execution. If the requesting task has a monitor, a normal termination message is given to the monitor by an event with event code $05.

Return Parameters: None

Error codes: None

EXAMPLE:

A user task, TSKA, has completed its processing and wants to terminate.

```
    TSKA:         .
                  .
                  .
                 MOVE.L   #15,D0          Load TERM directive number 15.
                 TRAP     #1
```

TERMINATE TARGET TASK — TERMT

Directive Number: 16

Parameter: Parameter Block Address

Parameter Block:

Target Task (8 bytes)	Task_id of target task to be terminated.
Abort code (2 bytes)	Abort code sent to task's monitor.

Detailed Description:

RMS68K halts execution of the target task and removes the task from memory. A user task cannot terminate a system task.

This directive operates in two modes:

a. Terminate single task

b. Terminate session

Terminate Single Task Mode

The requestor must specify the task_id of the task to be terminated. A system task can be terminated only in the single task mode.

Terminate Session Mode

This mode is used by the requesting task to terminate all user tasks within a given session. A user task can only terminate tasks within its own session. For this mode, the user does not specify the task_id (task_id = 0). RMS68K selects one user task from the relevant session and terminates it. Thus, a task could terminate all user tasks in a session by issuing the TERMT directive repeatedly until it is the only task remaining in the READY state (the requestor is immune to the TERMT directive).

The TERMT directive may require several milliseconds before the target TCB is eliminated from the system. Therefore, a task trying to recreate the same TCB may temporarily get an error from the CRTCB directive ($06 Taskname already exists). The TERMT call starts termination processing for the target task, but this processing may not be complete when the caller returns from the TERMT directive. If the calling task needs to be sure that the target task has been flushed out of the system, the caller may want to DELAY or make a call that references the target task until the Executive says "There is no such task".

Return Parameters:

A0	Name of terminated task

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.
6/$06	Target task already in termination.
9/$09	Invalid target task (user task attempting to terminate system task or target = requestor).

EXAMPLE:

A user task, TSKA, wants to terminate all tasks within its session. An abort code of $8888 is sent to the terminating task's monitor.

```
TSKA:      .
           .
           .
TRMT:      MOVE.L   #16,D0         Load TERMT directive number 16.
           LEA      PRMBLK,A0      Load parameter block address.
           TRAP     #1
           BEQ      TRMT           Branch, if successful, to TERMT; do it
                                   again.
           CMP.L    #3,D0          Test all terminations.
           BNE      FAULT          Branch, if error.
           .
           .
           .
PRMBLK:    DC.L     0              Any task.
           DC.L     0              N/A; user task.
           DC.W     $8888          Abort code to send to monitor.
```

ABORT SELF ABORT

Directive Number: 14

Parameter: Abort Code

Abort Code:	If the task initiating its own abnormal termination has a monitor task, the abort code is passed to the monitor via an event queued to the ASQ of the monitor.

Detailed Description:

RMS68K halts the execution of the requesting task and removes the task from memory. The abort code (lower 2 bytes of register A0) is given to the task's monitor as an event (code $05). The upper 2 bytes of register D0 are also passed to the task's monitor in the event.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0): None

EXAMPLE:

A user task, TSKA, wants to terminate itself abnormally, and give its monitor an abort code indicating what caused the abort. For this example an abort code of 2 is used.

```
    TSKA:     .
              .
              .
          MOVE.L   #14,D0      Load ABORT directive number 14.
          MOVE.W   #2,A0       Return abort code to monitor.
          TRAP     #1
```

WAIT WAIT

Directive Number: 19

Parameter: None

Detailed Description:

RMS68K places the requesting task into the WAIT state until a WAKEUP directive is issued by another task. When the WAKEUP does take place, execution of the target task starts at the location following the wait, with D0 cleared to 0.

If another task has already issued a WAKEUP directive to the requestor, the requestor returns immediately to the instruction following the WAIT.

Return Parameters: None

Error Codes: None

EXAMPLE:

A user task, TSKA, wants to put itself into the WAIT state until a WAKEUP directive is issued by another task.

```
TSKA:          .
               .
               .
          MOVE.L   #19,D0         Load WAIT directive number 19.
          TRAP     #1
WKUP:          .
               .
               .
```

WAKEUP A TARGET TASK — WAKEUP

Directive Number: 20

Parameter: Parameter Block Address

Parameter block:

Target Task (8 bytes)	Task_id of target task to be awakened.

Detailed Description:

RMS68K moves the specified target task from the WAIT state to the READY state to await execution. If the target task is not currently in the WAIT state, a WAKEUP PENDING condition is set and takes effect the next time the task goes into the WAIT state. When the target task is awakened, D0 contains 0.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.

EXAMPLE:

A non real-time user task, TSKB, wants to resume execution of TSKA that is in the WAIT state because it issued a WAIT directive.

```
TSKB:          .
               .
               .
          MOVE.L   #20,D0         Load WAKEUP directive number 20.
          LEA      PRMBLK,A0      Load parameter block address.
          TRAP     #1
          BNE      FAULT          Branch, if error.
               .
               .
               .
PRMBLK:   DC.L     'TSKA'         Target task to issue WAKEUP.
          DC.L     0              N/A; user task.
```

SUSPEND SELF

SUSPND

Directive Number: 17

Parameter: None

Detailed Description:

RMS68K stops execution of the requesting task and moves it to the SUSPEND state. The execution of the task is started again only by a RESUME directive issued by another task. When the requesting task is resumed, DO will have been cleared to 0.

Return Parameters: None

Error Codes: None

EXAMPLE:

A user task, TSKA, wants to SUSPEND itself until another task issues a RESUME directive.

```
    TSKA:         .
                  .
                  .
                MOVE.L   #17,DO          Load SUSPND directive number 17.
                TRAP     #1
    RSM:          .
                  .
                  .
```

RESUME A TARGET TASK — RESUME

Directive Number: 18

Parameter: Parameter Block Address

Parameter Block:

Target Task (8 bytes)	Task_id of target task to be resumed.

Detailed Description:

RMS68K resumes execution of a previously suspended task. The target task is moved from the SUSPEND state to the READY state to await execution.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.
10/$0A	Target task is not in SUSPEND state.

EXAMPLE:

A non real-time user task, TSKB, wants to RESUME TSKA that had previously suspended itself.

```
TSKB:         .
              .
              .
          MOVE.L   #18,D0              Load RESUME directive number 18.
          LEA      PRMBLK,A0           Load parameter block address.
          TRAP     #1
          BNE      FAULT               Branch, if error.
              .
              .
              .
PRMBLK:   DC.L     'TSKA'              Target task to RESUME.
          DC.L       0                 N/A; user task.
```

RELINQUISH EXECUTION RELINQ

Directive Number: 22

Parameter: None

Detailed Description:

The RELINQ directive allows a task to relinquish control of the processor to tasks of equal or slightly lower priority. A task of priority $NM where N and M are both between 0 and $F (inclusive) that executes a RELINQ directive, is placed back on the READY list at priority $N0 and runs again after all other READY tasks of priority $N0 or greater execute. (The RELINQ directive does not affect a task's current priority.) Register D0 is cleared to 0 when the requestor continues executing.

Return Parameters: None

Error Codes: None

EXAMPLE:

TSKA wants to RELINQUISH execution so that RMS68K can enter a dispatch cycle.

```
    TSKA:         .
                  .
                  .
             MOVE.L   #22,D0          Load RELINQ directive number 22.
             TRAP     #1
    CONT:         .
                  .
                  .
```

TASK ATTRIBUTES — TSKATTR

Directive Number: 23

Parameter: Parameter Block Address

Parameter Block:

Target Task (8 Bytes)	Task_id of target task whose attributes are returned. Requesting task is assumed if this field is 0.

Detailed Description:

The target task's user number and attributes are returned to the requestor in register A0.

Task attribute bits are defined as:

Bits 31-16	User number.
Bit 15=1	Task is system task.
Bit 14=1	Not defined.
Bit 13=1	Task is critical to OS; crash system if this task aborts.
Bit 12=1	VERSAdos recognizes this bit as a request for a dump of task aborts.
Bit 11=1	Task can "run anywhere" in system with no MMU. Segment descriptions and the task's entry address are adjusted so that they describe physical addresses.
Bits 10-9	Reserved
Bit 8=1	Task has created user semaphore.
Bit 7=1	Task is a real-time task.
Bit 6=1	Task is controlled by exception monitor.
Bit 5=1	Task is an exception monitor.
Bit 4=1	Task has own exception vectors.

Bit 3=1	Task has own trap vectors.
Bit 2=1	Task is last task in session (set only when task is terminating).
Bit 1=1	Task was aborted.
Bit 0=1	Task has claimed a user vector.

Return Parameters:

Register A0	Bits 31-16	User number
	Bits 15-0	Task attributes

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.
10/$0A	Task is terminating - register A0 contains the return parameter.

EXAMPLE:

A system task, TSKA, wants to receive the user number and attributes of TSKB. TSKA and TSKB are in the same session.

```
TSKA:          .
               .
               .
          MOVE.L   #23,D0         Load TSKATTR directive number 23.
          LEA      PRMBLK,A0      Load parameter block address.
          TRAP     #1
          BNE      FAULT          Branch, if error.
               .
               .
               .
PARBLK:   DC.L     'TSKB'         Taskname of which to get attributes.
          DC.L     0              Same session of system task.
```

RETURN COPY OF TASK CONTROL BLOCK — TSKINFO

Directive Number: 28

Parameter: Parameter Block Address

Parameter Block:

Target Task (8 bytes)	Task_id of target task. Requesting task is assumed if this field is 0.	
Options (2 bytes)	Bit 15=1	Return copy of target task's TCB.
	Bit 15=0	Do not return copy of target task's TCB.
	Bits 14-0	Not used; available for future enhancements.
Buffer Address (4 bytes)	Starting address of a 512-byte buffer where a copy of the target task's TCB is moved.	

Detailed Description:

A copy of the target task's TCB is moved to the requestor's address space. The requesting task must be a system task.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.
9/$09	Requesting task is not a system task.
12/$0C	Buffer not in requestor's address space.
15/$0F	Options not recognized.

EXAMPLE:

A non real-time system task, TSKA, wants a copy of the TCB of task TSKB in session 0422.

```
     TSKA:        .
                  .
                  .
                MOVE.L   #28,D0         Load TSKINFO directive number 28.
                LEA      PRMBLK,A0      Load parameter block address.
                TRAP     #1
                BNE      FAULT          Branch, if error.
                  .
                  .
                  .
     PARBLK:    DC.L     'TSKB'         Target taskname.
                DC.L     '0422'         Target task session.
                DC.W     $8000          Return target task TCB.
                DC.L     TCBBUF

     TCBBUF:    DS.B     512            Buffer area for TCB information.
```

GET A TARGET TASK'S TASK_ID GTTASKID

Directive Number:	10
Parameter:	Logical address of the parameter block describing the target task.

Parameter Block:

Taskname (4 bytes)	TCB address is requested for target task. Requesting task is assumed if this field is 0.
Session (4 bytes)	Not available if requestor is user task.

Detailed Description:

The Executive returns the task_id of the target task in response to the input of a taskname and session number. If the requestor is a non real-time task, the task_id is the original taskname and session number. However, if the requestor is a real-time task, the Executive returns an internally generated 8-byte code.

For future reference to the target task, the value returned by this directive in register A0 should be placed in the old taskname field in the parameter block and the value returned in register A1 should be placed in the session number field. This is not required for non real-time tasks, but should be done for a later migration to the real-time domain transparent to the task.

Return Parameters:

If the requestor was a real-time task:

Registers A0/A1	8-byte code for fast access to a target task.

If the requestor was not a real-time task:

Register A0	Taskname.
Register A1	Session Number.

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block is not in requestor's address space.
3/$03	Target task does not exist.

EXAMPLE:

A real-time task, TSKA, needs to get TSKB's task_id in preparation for issuing a WAKEUP signal to TSKB.

```
TSKA:      MOVE.L    #10,D0          Load GTTASKID directive number 10.
           LEA       PARBLK1,A0      Load GTTASKID parameter block address.
           TRAP      #1
           BNE       FAULT           Branch, if error.

           MOVEM.L   A0-A1,PARBLK2   Save TASKB's task_id in QEVNT parameter
                                     block.
           MOVE.L    #20,D0          Load WAKEUP directive number 35.
           LEA       PARBLK2,A0      Load WAKEUP parameter block address.
           TRAP      #1
           BNE       FAULT           Branch, if error.
            .
            .
            .
PARBLK1:   DC.L      'TSKB'          Target taskname.
           DC.L      0               Same session.

PARBLK2:   DC.L      0               Target task task_id field.
           DC.L      0
```

GET A TARGET TASK'S TASKNAME AND SESSION NUMBER GTTASKNM

Directive Number:	12
Parameter:	Logical address of the parameter block describing the target task.

Parameter Block:

Target Task (8 bytes)	Task_id of target task.

Detailed Description:

In response to the input of a task_id, the Executive returns the taskname and session number of the target task.

Return Parameters:

Register A0	Taskname of the target task.
Register A1	Session number of the target task.

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block is not in requestor's address space.
3/$03	Target task does not exist.

EXAMPLE:

A real-time monitor task needs to get the taskname of the subtask that has just terminated. The subtask's task_id is contained within the subtask termination event.

```
    TSKA:       MOVE.L   #12,D0          Load the GTTASKNM directive number 12.
                LEA      SUBTASK_ID,A0   Point  A0  to the subtask task_id field
                                         within the subtask termination event.
                TRAP     #1
                BNE      FAULT           Branch, if error.
                  .
                  .                      A0 now contains the subtasks taskname
                  .                      A1 contains its session number.
    EVENT:      DC.B     0               Length field.
                DC.B     0               Event code field.
  SUBTASK_ID:   DC.L     0               Subtask task_id field.
                DC.L     0
                  .
                  .
                  .
    EVENT:      DC.B     0               Length field.
```

CHAPTER 5

TIME MANAGER

5.1 OVERVIEW

RMS68K's Time Manager supports two concepts of time: elapsed and calendar. The directives supporting elapsed time involve notifying a task when a quantum of time has expired. The directives supporting calendar time allow a task to inform the time manager of the current date and time (e.g., March 21, 1985; 12:04), or to ask the time manager for the current date and time.

The time manager can also determine when a tasks' timeslice has expired (if timeslicing was enabled at initialization). When a timeslice expires, the time manager places the running task back on the READY list after all tasks of the same priority and forces a dispatch cycle.

5.2 THEORY OF OPERATION

5.2.1 Basic Principles

RMS68K's time manager knows how to respond to "ticks" from a timer Interrupt Service Routine (ISR). A tick is defined at initialization to be some number of milliseconds and optionally, some additional number of microseconds. A timeslice is defined to be some integral number of timer ticks. Some typical values are:

a. Tick = 10 milliseconds

b. Timeslice = 2 ticks

The time manager maintains a data structure called the Periodic Activation Table (PAT) consisting of nodes representing requests for notification at specified times. They may be requests to notify a task that a time quantum has expired, to notify a driver, or to notify the time manager that some internal event has occurred (e.g., the date has changed at midnight). The PAT nodes are linked in order of increasing time from "now" and the time difference between two adjacent nodes is recorded in a delta field within the second node.

Each node has a 32-bit activation ID that allows a task to request notification of multiple intervals via the RQSTPA directive and to differentiate between them via the activation ID. For most applications, one interval is enough so the activation ID is usually 0.

The user does not need to understand the internal workings of the PAT to request service from the time manager.

5.2.2 Elapsed Time

The concept of elapsed time is supported by three RMS68K directives:

a.	DELAY	A task moves itself into the DELAY state for a specified period of time.
b.	DELAYW	A task moves itself into the DELAY state for a specified period of time and returns if the time expires, a WAKEUP occurs, or an event is queued.
c.	RQSTPA	A task requests the timed periodic activation of itself or another task.

The DELAY directive is a request to WAKE ME UP in "n" milliseconds, at which time the task is dispatched to the instruction following the DELAY directive call. The only exception to this rule involves a task that issues a DELAY directive when both its ASR and ASQ are enabled. If an event is sent to its ASQ, the DELAY is cancelled and the task is dispatched to its ASR. On completion of ASR processing, the RTEVNT directive dispatches the task to the instruction following the DELAY directive call.

The DELAYW directive is a combination of three directives:

a. DELAY

b. WTEVNT

c. WAIT

DELAYW tells RMS68K to wait until one of three events occur:

a. The DELAY interval expires.

b. An event is queued to this task.

c. A WAKEUP is issued to this task.

To support the WTEVNT function, DELAYW automatically enables the requesting task's ASQ and ASR. If an event is queued to this task, the DELAY is cancelled and the task is dispatched to its ASR. If either the DELAY expires or a WAKEUP is issued, the DELAY is cancelled and the task is dispatched to the instruction following the DELAYW directive call with the task's ASQ and ASR enabled.

This directive is often used to specify a background timeout on a WTEVNT or WAIT directive.

The RQSTPA directive differs from DELAY and DELAYW in two ways:

a. It can be used to activate the task on a periodic basis ("Wake me up every 100 milliseconds from now on").

b. It does not put the task into a WAIT state but returns immediately to the instruction following the RQSTPA directive call. The notification that the periodic interval has expired occurs asynchronously in relation to the task's normal execution.

The RQSTPA is used in an environment where a task wants to be activated on some periodic basis to poll the status of a non-interrupting hardware device. For example, consider the following task written in pseudo PL/1:

```
poll_keyboard: procedures;

     call initialize (keyboard);
     call RQSTPA (100-milliseconds, issue_resume);

     do forever;
          call suspend;
          call poll (keyboard)
          if (key_was_pressed)
          then call process (keystroke);
     end;

end poll_keyboard;
```

The call to RQSTPA tells the time manager to issue a RESUME signal to the poll_keyboard task every 100 milliseconds from now on. Poll_keyboard then loops forever, suspending itself until the time manager issues the 100 millisecond RESUME signal. On resumption, the task polls the keyboard to see if any key has been pressed and processes as required.

If the processing of the keystroke takes longer than 100 milliseconds, it is not a problem because the intervening RESUME signals are ignored until poll_keyboard issues the next SUSPEND directive. Then it is resumed at the next 100 milliseconds interval after it SUSPENDS itself.

RQSTPA can notify the task of the periodic interval's expiration in one of four ways:

a. Issue a RESUME signal to the task.

b. Issue a WAKEUP signal to the task.

c. Queue an event to the task's default ASR.

d. Queue an event to an alternate ASR.

It may also be used to schedule multiple activations at different intervals and the task can discriminate between these by either their activation IDs embedded within the timer event (refer to paragraph 2.7), or by using distinct alternate ASR addresses for each activation.

For example, an autonomous robot has the following requirements:

a. Must scan field of vision every 100 milliseconds.

b. Must re-evaluate current strategy every 10 seconds based on knowledge acquired since the last evaluation.

c. Must diagnose internal functions every minute:

   Temperature
   Fuel level
   Oil pressure

These requirements could be met by one task scheduling three activation intervals:

```
robot:  procedure;

     call initialize (hardware);
     call GTASQ (enable_asq, enable_asr);

     call RQSTPA (100_milliseconds, scan_vision_asr);
     call RQSTPA (10_seconds, evaluate_strategy_asr);
     call RQSTPA (1_minute, diagnose_internal_functions_asr);

     do forever;
          call WTEVNT;
     end;


scan_vision_asr:

     call scan_field_of_vision;
     if (object_in_view)
     then call react_to_object;
     call RTEVNT;

end scan_vision_asr;
```

```
evaluate_strategy_asr:

     call evaluate_strategy (current_strategy, new_knowledge);
     if (current_strategy_needs_revision)
     then current_strategy =
          generate_new_strategy (current_strategy, new_knowledge);
     RTEVNT;

end evaluate_strategy_asr;


diagnose_internal_functions_asr:

     call SETASQ (enable_asr);
     call diagnose_internal_functions;
     if (adjustment_is_necessary)
     then call adjust_internal_functions;

end diagnose_internal_functions_asr;

end robot;
```

Note that the diagnose_internal_function_asr enables the ASR before proceeding with its diagnostics. This implements the policy that diagnostics are less important than scanning the field of vision or evaluating the current strategy based on the newly acquired knowledge and therefore should be "interruptible" by those requirements.

The purpose of this example is to show the power and flexibility of the RQSTPA directive. Another way to implement the robot would be to design three tasks running at different priorities, each implementing one discrete ASR function.

5.2.3 Calendar Time

Calendar time is supported by two directives:

a.	STDTIM	A system task sets the system date and time.
b.	GTDTIM	A task obtains the current system date and time.

The format for the system date and time is two longwords where:

a. The first longword represents the number of days since December 31, 1979.

b. The second longword represents the number of milliseconds since midnight of the current day.

The VERSAdos real-time operating system includes utility subroutines (TIMECONV, ODATCONV, GDATCONV, DATEGO, and DATEOG) for converting binary format for system date and time required by the time manager and an equivalent ASCII format, consisting of fields for the day, month, year, hours, minute, and second.

5.3 DATA STRUCTURES

The time manager maintains the PAT and several SYSPAR parameters. An entry is placed in the PAT each time a task uses the DELAY, DELAYW, or RQSTPA directives.

5.3.1 Periodic Activation Table (PAT)

PAT	(4 bytes)	Block ID The PAT table begins with '!PAT' to allow consistency checking and ease of dump reading.
PATFHDR	(4 bytes)	Pointer to the first unused entry in the PAT.
PATHDR	(4 bytes)	Pointer to the first entry used in the PAT.
PATBABT	(4 bytes)	Background activation block used to schedule Executive routine that activates PAT nodes.

A PAT entry is defined as:

PATNEXT	(4 bytes)	Pointer to next entry in list.
PATTCB	(4 bytes)	TCB address of task that requested delay or periodic activation (0 if Executive activation node).
PATDELTA	(4 bytes)	Amount of time (ms) between activation of this node and previous node.
PATINTV	(4 bytes)	Activation interval.
PATASR	(4 bytes)	ASR address of message option chosen by caller. If this is Executive routine entry, starting address of Executive routine.

PATOPT	(2 bytes) Activation options. A DELAY request results in PATOPT = 0.
PATARID	(4 bytes) Activation request ID.
PATCNT	(2 bytes) Activation count.
PATILVL	(2 bytes) Interrupt level to be used by Executive routine.

5.4 TIME MANAGER DIRECTIVES

The following pages contain detailed descriptions and examples of the time manager directives.

DELAY SELF DELAY

Directive Number: 21

Parameter: Number of Milliseconds to Delay

Detailed Description:

RMS68K delays the execution of the requesting task until the specified amount of time has elapsed; execution resumes at the location following the DELAY directive.

This directive does not affect asynchronous event processing. If the ASR is enabled and an event arrives, the DELAY is considered to be satisfied and the event is processed.

This directive places an entry into the PAT and results in cancelling a periodic activation request with a request ID = 0, if such an entry exists for the calling task. It has no effect on periodic activation requests with nonzero request IDs.

Return Parameters: None

Error Codes: None

EXAMPLE:

TSKA wants to delay itself for 5 seconds (5000 milliseconds).

```
    TSKA:         .
                  .
                  .
                MOVE.L   #21,D0        Load DELAY directive number 21.
                MOVE.L   #5000,A0      Load number of milliseconds to delay.
                TRAP     #1
    CONT:         .
                  .
                  .
```

DELAY and WAIT — DELAYW

Directive Number: 30

Parameter: Number of Milliseconds to Delay

Detailed Description:

This directive functions as a combination of the DELAY, WTEVNT, and WAIT directives. If the calling task has an ASQ, RMS68K enables the calling task's ASR and ASQ and puts the calling task into a WAIT state. The task returns to the READY state as a result of any of the following occurrences:

a. The specified amount of time has elapsed. Execution resumes at the location following the DELAYW directive with the DELAY and WAIT cancelled.

b. An asynchronous event arrives or is already present in the calling task's ASQ. Both DELAY and WAIT are cancelled. Control is given to the calling task at its ASR address. When the ASR returns, execution resumes at the location following the DELAYW directive.

c. A WAKEUP is sent to the waiting task or the WAKEUP PENDING condition exists at the time the directive is called. The DELAY is cancelled and execution resumes at the location following the DELAYW directive.

This directive places an entry into the PAT and results in cancelling a periodic activation request with a request ID = 0. It has no effect on a periodic activation request with a nonzero request ID.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
5/$05	No room in PAT.

EXAMPLE:

TSKA wants to WAIT for an asynchronous event or a WAKEUP, but wants to return to normal processing in five seconds (5000 milliseconds) if neither occurs within that time.

```
    TSKA:       .
                .
                .
          MOVE.L   #30,D0          Load DELAYW directive number 30.
          MOVE.L   #5000,A0        Load number of milliseconds to delay.
          TRAP     #1
          BNE      FAULT           Branch, if error.
                .
                .
                .
```

REQUEST PERIODIC ACTIVATION RQSTPA

Directive Number: 29

Parameter: Parameter Block Address

Parameter Block:

Target Task (8 bytes)	Task_id of task to be activated. (Refer to target task interface, paragraph 1.3.6.)	
Directive Options (2 bytes)	Bit 15=1	Time of first activation is specified in initial time field.
	=0	Time of first activation is computed as the sum of the time that the directive is issued and the interval time.
	Bit 14=1	The interval is specified in the interval field. Task is activated at initial time, initial time plus interval, initial time plus 2*interval, etc.
	=0	Task is activated at initial time only.
	Bits 15-14 =00	This request cancels a currently active periodic activation.
	Bits 13-12	Activation method. 00 Issue RESUME. 01 Issue WAKEUP. 10 Queue timer event to default ASR service address. 11 Queue timer event to ASR service address specified in service address field.
	Bit 11=1	Task is activated only one time (if this bit is set, bit 14 is ignored).
	=0	Task is either activated once or continuously, as determined by option bit 14.

	Bit 10=1	An argument is supplied in the activation request ID field. This argument provides a unique ID for this request if multiple requests are outstanding.
	=0	Activation request ID of all zeros is assumed.
	Bit 9=1	Send an event to target task when the activation is cancelled. This option can be set with the request to activate the task.
	Bits 8-0	Reserved
Initial time (4 bytes)	Time of day, in milliseconds, for first activation. Applicable only if options bit 15=1.	
Interval (4 bytes)	Period of time, in milliseconds, between activations. Applicable only if options bit 14=1.	
Service Address (4 bytes)	ASR service address where timeout event is to be serviced. Applicable only if options bits 13-12 = 11.	
Activation Request ID (4 bytes)	A unique identification used to identify this request if task has multiple requests outstanding. Applicable only if options bit 10=1.	

Detailed Description:

RMS68K activates the target task at an initial time and at optional intervals. The task can be activated in four ways, one of which is specified when the directive is issued:

RESUME

RMS68K issues a RESUME signal to activate the task from the SUSPEND state. If the task is not in the SUSPEND state at that time, the RESUME has no effect.

WAKEUP

RMS68K issues a WAKEUP signal to activate the task from the WAIT state. If the task is not in the WAIT state at that time, the WAKEUP PENDING status is set for that task, and the WAKEUP occurs when the task goes to the WAIT state.

Queue Timer Event to Default ASR Service Address

When activation is to occur, RMS68K queues an event (code =$04) to the task's ASQ. The event is serviced at the default ASR service address. (Refer to paragraph 2.7 for the timer event format.)

WARNING

TIMER EVENTS (CODE = $04) ARE 16 BYTES LONG. IF A TASK REQUESTS THE RQSTPA DIRECTIVE ACTIVATION BY A TIMER EVENT, THE ASQ'S MAXIMUM MESSAGE LENGTH MUST BE AT LEAST 16 BYTES.

Queue Timer Event to Alternate ASR Service Address

When activation is to occur, RMS68K queues an event (code =$04) to the task's ASQ. The event is serviced at the ASR service address specified in the parameter block.

If a request to activate a task has the same activation request ID as a currently active request, the previous request is cancelled and the new request is scheduled.

Option bits 15, 14, and 11 interact to select between:

a. (cancelling a previous request) or
   (requesting an activation);

b. (activating a task one time) or
   (activating a task periodically);

c. Activating at initial time. Where:

   initial_time =

   (value of initial_time_field) or
   (now + value of interval_field);

These option bits are defined as (X implies a "don't care" where the value may be either 0 or 1):

OPTION BITS 15, 14, 11	ACTION OF RQSTPA
00X	Cancel (previous request);

010	Activate (periodically);
	initial_time = now + interval_field;

011	Activate (one_time);
	initial_time = now + interval_field;

(100) or
(1X1)
	Activate (one_time);
	initial_time = initial_time_field;

110	Activate (periodically);
	initial_time = initial_time_field;

To cancel all currently-active periodic activations, options bit 10 must be set and the activation request ID supplied must be all zeros.

The event sent to the target task when a request is cancelled (if option bit 9=1) is sent immediately instead of at the next scheduled interval time. The activation count field in the event queued has bit 15=1 to identify it as a cancel event.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.
5/$05	PAT is full.
7/$07	No entry found on cancel request.
16/$10	Interval supplied was zero or negative.

EXAMPLE:

A non real-time user task, TSKA, wants to activate TSKB every 500 milliseconds. TSKB is to be activated by an ASR interrupt at its default service address. An activation request ID is supplied.

```
TSKA:        .
             .
             .
        MOVE.L   #29,D0          Load RQSTPA directive number 29.
        LEA      PRMBLK,A0       Load parameter block address.
        TRAP     #1
        BNE      FAULT           Branch, if error.
             .
             .
             .
PRMBLK: DC.L     'TSKB'          Task to be activated.
        DC.L       0             N/A; user task.
        DC.W     $6400           Interval  in interval field, interval
                                 time  from  now, default ASR address,
                                 continual  activation,  activation ID
                                 is specified.
        DC.L       0             N/A; bit 15=0.
        DC.L      500            Interval in milliseconds.
        DC.L       0             N/A; bits 13-12 ≠ 11.
        DC.L     'ID#1'          Identification table.
```

SET SYSTEM DATE AND TIME STDTIM

Directive Number: 73

Parameter: Parameter Block Address

Parameter Block:

New System Date (4 bytes)	The date is expressed in ordinal day number with a base of January 1, 1980. January 1, 1980 = day number 1 January 2, 1980 = day number 2 January 1, 1981 = day number 367 Etc.
New System time (4 bytes)	The time is expressed in the number of milliseconds into the new day.

Detailed Description:

RMS68K updates the system date and time. Only a system task may use this directive.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
9/$09	Requestor is not a system task.

EXAMPLE:

TSKA, a system task, wants to reset the system date to March 3, 1980 and the time to 01:05:33.

```
TSKA:          .
               .
               .
          MOVE.L   #73,D0                Load STDTIM directive number 73.
          LEA      PRMBLK,A0             Load parameter block address.
          TRAP     #1
          BNE      FAULT                 Branch, if error.
               .
               .
               .
PRMBLK:   DC.L     63                    Set day to March 3, 1980.
          DC.L     $003C0348             Set time to 1:05:33.
```

GET SYSTEM DATE AND TIME GTDTIM

Directive Number: 74

Parameter: Return Parameter Block Address

Detailed Description:

RMS68K places the current system date and time into the specified return parameter block.

Return Parameter Block:

Current System Date (4 bytes)	The date is expressed in ordinal day number, with a base of January 1, 1980. January 1, 1980 = day number 1 January 2, 1980 = day number 2 January 1, 1981 = day number 367 Etc.
Current System Time (4 bytes)	The time is expressed in the number of milliseconds into the current day.

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.

EXAMPLE:

A user task, TSKA, wants to examine the current date and time.

```
TSKA:     .
          .
          .
          MOVE.L   #74,D0       Load GTDTIM directive number 74.
          LEA      PRMBLK,A0    Load parameter block address.
          TRAP     #1
          BNE      FAULT        Branch, if error.
          .
          .
          .
PRMBLK:   EQU       *
CURRDT:   DS.L      1            Current system date.
CURRTM:   DS.L      1            Current system time.
```

CHAPTER 6

SEMAPHORE MANAGER

6.1 OVERVIEW

The Semaphore Manager provides sophisticated synchronization primitives that are used to coordinate the activities of multiple tasks or to arbitrate access to shared resources.

Three types of semaphores are used to fulfill different sets of requirements:

a. A binary semaphore (type 1) arbitrates access to a single resource that is either available or not.

b. A counting semaphore (type 3) controls access to a pool of "n" resources where at any moment "m" of those resources are available ($0 \leq m \leq n$) and "n-m" are not.

c. A broadcast semaphore (type 2) allows one task to broadcast a signal that an event has occurred to one or more tasks waiting on the signal. The broadcast semaphore is often used to synchronize the execution of two or more asynchronous tasks.

6.2 THEORY OF OPERATION

6.2.1 Synchronization Requirements

To synchronize the execution of multiple tasks or to arbitrate access to shared resources, requires a simple form of an event called a signal. A signal indicates that a predefined event has occurred and contains no data describing the event. Sophisticated synchronization also requires a counter to record the number of signals sent but not yet received, and a list of tasks awaiting receipt of the signal.

The semaphore data structure fulfills all the previous requirements. A semaphore possesses a name to distinguish it from other semaphores within the system, a key to enable quick access to the semaphore, and the requisite signal count variable and linked list of waiting tasks. In addition to the signal count variable, a semaphore may also contain an initial count, used as an initial assignment value for the signal count or to determine when the semaphore is no longer required.

6.2.2 Synchronization Services

The semaphore manager provides synchronization services via the directives:

CRSEM	A task creates a new semaphore or resets the initial count of an existing semaphore.
ATSEM	A task attaches to a semaphore and acquires use of the semaphore. If the semaphore does not exist, it is created and given an initial signal count.
WTSEM	A task requests and, if necessary, waits for semaphore-controlled access.
SGSEM	A task signals release of a semaphore-controlled resource.
DESEM	A task detaches from a semaphore.
DESEMA	A task detaches from all semaphores to which it is attached.

6.2.3 Synchronization Rules

The semaphore manager supports three types of semaphores, each designed to solve a specific synchronization problem. The synchronization rules for all types of semaphores are:

a. When a task does a WAIT ON A SEMAPHORE operation, the signal count is decremented by one. The task continues execution if the count is then greater than or equal to zero. If the count is less than zero, the task is put on a waiting list for the semaphore.

b. The signal count is incremented by one when a task does a signal operation. If the count is less than or equal to zero, the first task in the semaphore waiting list is placed in the READY state. The signaling task always completes its execution.

6.2.4 Semaphore Types

The three types of semaphores supported by the semaphore manager are:

<u>Type 1 (Binary) Semaphore</u>

When multiple tasks require exclusive access to a single resource, a type 1 or binary semaphore is used. The signal indicates the available or not available status of the resource.

SIGNAL COUNT	STATUS
1	(resource is available);
0	(resource is not available) and (no tasks are waiting on the resource);
-n	(resource is not available) and ("n" tasks are waiting on the resource);

A task bids for the resource by performing a WAIT ON SEMAPHORE operation which decrements the signal count. If the signal count was previously one (resource available), the task is allowed to access the resource. Otherwise, the task is placed on a queue waiting for the resource.

When a task is finished with the resource, it indicates the resource is available by performing a SIGNAL SEMAPHORE operation that increments the signal count. If the signal count was previously zero (no tasks waiting), no other action is taken. Otherwise, ("n" tasks waiting) the first task on the wait queue is made ready and given access to the resource.

A binary semaphore is deleted when all tasks using it have detached. (A binary semaphore has a built in protection mechanism that does not allow the signal count to exceed one. This feature enforces the "binary" nature of the semaphore (available or not available), and is the main distinction between binary and counting semaphores.)

APPLICATION EXAMPLE:

A system has several tasks capable of outputting multiple line messages to a single console device. To avoid messages becoming interleaved with one another, access to the console should be arbitrated by a binary semaphore. Any task that wants to output a message must first wait on the binary semaphore. When it becomes available, the task may output its message, knowing that no other task is currently using the console. When the message is complete, the task must signal the binary semaphore to indicate that the console is now available for use by other tasks.

The VERSAdos I/O system handles all synchronization of I/O devices. Therefore, tasks using VERSAdos I/O do not need to arbitrate access to the system console via a binary semaphore.

Type 2 (Broadcast) Semaphore

A broadcast or type 2 semaphore simultaneously notifies "n" tasks that an event has occurred. These "n" tasks are called dependent tasks and the task that broadcasts the event is known as the primary task. Only the primary task may create the broadcast semaphore. All dependent tasks must attach to it before using it; it is valid to attach to a broadcast semaphore before it is created.

If a task attaches to a broadcast semaphore before it is created, the semaphore manager automatically creates it and gives it a signal count of zero. Any dependent tasks that do a WAIT operation on the semaphore before it is created are placed in its First-In First-Out (FIFO) WAIT queue and the signal count is decremented.

When the primary task does create the semaphore with an initial count of "m", the semaphore manager signals the semaphore "m" times and broadcasts the event to all waiting dependent tasks up to a total of "m". If any of the "m" dependent tasks wait on the broadcast semaphore after it is created, they are allowed to run.

Deletion of the broadcast semaphore occurs when the last task has detached from it and the current signal count value is equal to the initial count value.

APPLICATION EXAMPLE:

One use of the broadcast semaphore is to control the execution sequence of several tasks. Consider a problem in factory automation where one task monitors the flow of materials and must inform nine material handling tasks when the raw materials arrive. This requires a broadcast semaphore called "Goods" with an initial count of nine. As soon as the material handling tasks finish processing the previous batch, they can attach to and WAIT on Goods. When the primary task detects the arrival of a new batch of materials, it can create Goods with an initial count of nine and any dependent tasks that are waiting are released to run. Any dependent tasks that have not finished their previous job are allowed to run when they attach and WAIT on Goods. Finally, when the dependent tasks finish processing the materials, they signal and detach from Goods. When the last dependent task detaches from Goods and the signal count equals the initial signal count of nine, Goods is deleted from the system.

Type 3 (COUNTING) Semaphore

A type 3 or counting semaphore is used when one task controls a pool of resources that other tasks need to access. The counting semaphore is an extension of the binary semaphore where the signal count variable can assume any positive or negative value. When the signal count is positive, it indicates the number of items from the resource pool currently available; when it is negative, it indicates the number of tasks waiting on one of those items.

APPLICATION EXAMPLE:

A buffer management system can be implemented using a counting semaphore. If the count is positive, it indicates the number of buffers currently available; if negative, it indicates the number of tasks waiting for a buffer.

The traditional producer-consumer problem can be solved with two counting semaphores, full and empty. The producer waits on empty. When empty is available, it dequeues a buffer from the empty list, fills it, queues it on the full list and signals full. The consumer waits on full; dequeues the full buffer, consumes it, queues it back on the empty list, and signals empty.

This example is for demonstration purposes only; it does not address the problem of maintaining the integrity of the full and empty lists within the concurrent multitasking environment.

6.3 DATA STRUCTURES

Two data structures are used:

a. Semaphore Parameter Block	Describes a request to a semaphore manager directive.
b. User Semaphore Table (UST)	An array of semaphore descriptors indexed by semaphore key containing information on all semaphores known to the system.

6.3.1 Semaphore Parameter Block

Many of the task synchronization directives use a semaphore parameter block. The general format is:

4 bytes	Semaphore name
4 bytes	Semaphore key
1 byte	Initial count
1 byte	Semaphore type

Semaphore name	The name that any task can reference to use the semaphore. Any 32-bit combination is a valid semaphore name.
Semaphore key	Allows RMS68K to quickly access the semaphore. RMS68K creates the semaphore key and returns it to the user in register A0 when an ATSEM or CRSEM directive is issued.

Initial count	Supplied by a task that creates a type 2 or type 3 semaphore. Sometimes the initial count is used to initialize the semaphore's signal count or to control the deletion of a semaphore.
Semaphore type	Specifies the semaphore as a type 1, 2, or 3. The type defines the characteristics of the semaphore and how it is used:
00 - Error	
01 - Type 1	Used when a task requires exclusive access to a single resource.
10 - Type 2	Used to order execution of dependent tasks.
11 - Type 3	Used if a task controls a resource.

NOTE: Semaphores are always local to a session and cannot be shared with tasks in different sessions.

6.3.2 User Semaphore Table (UST)

The UST manages user semaphore activities.

UST	(4 bytes)	Block ID Each UST segment begins with '!UST' to allow consistency checking and ease of dump reading.
USTNEXT	(4 bytes)	Reserved for future use.
USTNSEG	(2 bytes)	Reserved for future use.
USTNPAGE	(2 bytes)	UST segment size Contains the number of 256-byte pages comprising this UST segment.
USTMENT	(2 bytes)	Maximum entry count Contains the maximum number of UST entries allowable in this UST segment.

USTCENT	(2 bytes)	Current entry count Contains the number of UST entries currently residing in this UST segment.
USTFENT	(4 bytes)	First entry address Points to the first UST entry.

A UST entry is defined as:

USTTNAME	(4 bytes)	Originating task's name.
USTSESSN	(4 bytes)	Originating task's session number.
USTSNAME	(4 bytes)	Semaphore name.
USTUCNT	(2 bytes)	Number of tasks attached to this semaphore or, if equal to -1, this entry is a pointer to a semaphore entry.
USTXCNT	(1 byte)	Initial count or count of waits and signals.
USTTYPE	(1 byte)	Semaphore type (1, 2, or 3).
USTSEM	(6 bytes)	Semaphore or pointer to semaphore if USTCNT = -1.

Every time a task creates or attaches to a semaphore, an entry is placed in the UST. If the task creates a semaphore, a semaphore entry is created with USTUCNT set to the number of tasks attached to the semaphore and USTSEM containing the semaphore count variable and the linked list of waiting tasks. If the task attaches to a semaphore, a pointer entry is created with USTUCNT set to -1 to indicate a pointer entry, and USTSEM containing a pointer to the "real" semaphore entry.

This scheme increases system security and power by encoding all knowledge about the state of the semaphore and all tasks currently attached to it, allowing the system to handle unexpected cases properly (such as tasks terminating target tasks waiting on semaphores; creators of type 3 semaphores detaching from the semaphore while other tasks are accessing the resource).

6.4 SEMAPHORE MANAGER DIRECTIVES

The directives a task can use to request services from the semaphore manager are described on the following pages.

CREATE A SEMAPHORE CRSEM

Directive Number:	45
Parameter:	Semaphore Block Address

Semaphore Block:

Semaphore Name (4 bytes)	Name of semaphore to create.
Semaphore Key (4 bytes)	N/A
Initial Count (1 byte)	Used for type 2 and type 3 semaphores. Must be non-negative value. See detailed description below.
Semaphore Type (1 byte)	Type 1, 2, or 3.

Detailed Description:

RMS68K creates or re-initializes the specified semaphore, and allows the requesting task to use it. The semaphore type determines the directive function.

Type 1 (Binary):	If the specified semaphore does not exist, it is created with an initial signal count of one. If it exists, no action is taken.
Type 2 (Broadcast):	If the specified semaphore does not exist, it is created with an initial signal count of the initial count in the parameter block. Any tasks in the WAIT state as a result of issuing an ATSEM directive followed by a WTSEM directive before the CRSEM directive creates the semaphore, are reactivated. If the semaphore exists, the initial count in the parameter block is saved.
Type 3 (Counting):	If the specified semaphore does not exist, it is created with an initial signal count of the initial count in the parameter block. Any tasks in the WAIT state as a result of issuing an ATSEM directive before the CRSEM directive creates the semaphore, are reactivated. If the semaphore exists, the CRSEM directive is rejected.

Return Parameters:

A0	Semaphore key

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
5/$05	No more semaphore space available in system.
6/$06	Duplicate request to create type 3 semaphore.
11/$0B	Semaphore type given conflicts with existing semaphore type.
15/$0F	Illegal semaphore type.
16/$10	Negative count field supplied.

EXAMPLE:

Refer to paragraph 6.5.

ATTACH TO SEMAPHORE ATSEM

Directive Number: 41

Parameter: Semaphore Block Address

Semaphore Block:

Semaphore Name (4 bytes)	Name of semaphore to which attaching.
Semaphore Key (4 bytes)	N/A
Initial Count (1 byte)	N/A
Semaphore Type (1 bytes)	Type 1, 2, or 3.

Detailed Description:

RMS68K allows the requesting task to use the specified semaphore. The semaphore type determines the directive functions.

Type 1 (Binary):	If the specified semaphore does not exist, it is created with an initial signal count of one. If it exists, no action is taken.
Type 2 (Broadcast):	If it does not exist, the specified semaphore is created with an initial signal count of zero. If it exists, no action is taken.
Type 3 (Counting):	If the specified semaphore does not exist, the requesting task is placed in a WAIT ON SEMAPHORE state until the CRSEM directive issued by another task creates the semaphore. If the semaphore exists, no action is taken.

Return Parameters:

A0	Semaphore Key

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
5/$05	No more semaphore space available in system.
6/$06	Duplicate request.
11/$0B	Semaphore type conflicts with existing semaphore type.
15/$0F	Illegal semaphore type specified (type 0).

EXAMPLE:

Refer to paragraph 6.5.

WAIT ON SEMAPHORE WTSEM

Directive Number: 42

Parameter: Semaphore Block Address

Semaphore Block:

Semaphore Name (4 bytes)	Semaphore on which requesting task is waiting.
Semaphore Key (4 bytes)	Assigned when a semaphore is created. A task should save the semaphore key when it first attaches to the semaphore.
Initial Count (1 byte)	N/A
Semaphore Type (1 byte)	N/A

Detailed Description:

The current signal count of the specified semaphore is decremented by one. If the count is zero or positive, the requesting task continues executing. If the count is negative, the requesting task is added to the semaphore waiting list (FIFO).

If the semaphore is type 1, a check is made that the WTSEM directive is issued before a SGSEM directive.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
7/$07	Semaphore not found.
9/$09	WTSEM and SGSEM out of sequence for type 1 semaphore.

EXAMPLE:

Refer to paragraph 6.5.

SIGNAL SEMAPHORE SGSEM

Directive Number: 43

Parameter: Semaphore Block Address

Semaphore Block:

Semaphore Name (4 bytes)	Semaphore requesting task is signaling.
Semaphore Key (4 bytes)	Assigned when a semaphore is created. A task should save the semaphore key when it first attaches to the semaphore.
Initial Count (1 byte)	N/A
Semaphore Type (1 byte)	N/A

Detailed description:

The current signal count of the specified semaphore is incremented by one. If the count is zero or negative, the first task in the semaphore waiting list is removed from the list and placed in the ready list to await execution. The requesting task continues executing (RMS68K does not enter its dispatch cycle).

If the semaphore is type 1, a check is made that a WTSEM directive is issued before the SGSEM directive and that the SGSEM follows a WTSEM.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
7/$07	Semaphore not found.
9/$09	WTSEM and SGSEM out of sequence for type 1 semaphore.

EXAMPLE:

Refer to paragraph 6.5.

DETACH FROM SEMAPHORE — DESEM

Directive Number: 44

Parameter: Semaphore Block Address

Semaphore Block:

Semaphore Name (4 bytes)	Semaphore from which detaching.
Semaphore Key (4 bytes)	Assigned when semaphore is created. A task should save the value when it first attaches to a semaphore.
Initial Count (1 byte)	N/A
Semaphore Type (1 byte)	N/A

Detailed Description:

RMS68K detaches the requesting task from the specified semaphore that can no longer use the semaphore until an ATSEM is issued. The physical removal of a semaphore from the system is based on the semaphore type.

Type 1 (Binary):	Physical removal of the semaphore from the system is done when the last user detaches.
Type 2 (Broadcast):	When the last user detaches, and the current signal count is equal to the initial count set by the CRSEM directive, the semaphore is physically removed from the system.
Type 3 (Counting):	Physical removal of the semaphore from the system is done when the task that created it with a CRSEM directive detaches.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
7/$07	Semaphore not found.

EXAMPLE:

Refer to paragraph 6.5.

DETACH ALL SEMAPHORES DESEMA

Directive Number: 46

Parameter: None

Detailed Description:

RMS68K detaches the requesting task from all semaphores. Rules for removing a semaphore from the system are explained under the DESEM directive.

Register D0 is always cleared to 0.

Return Parameters: None

Error Codes: None

EXAMPLE:

TSKA wants to detach from all semaphores that it is currently attached to.

```
     TSKA:         .
                   .
                   .
                MOVE.L    #46,D0         Load DESEMA directive number 46.
                TRAP      #1
                BNE       FAULT          Branch, if error.
                   .
                   .
                   .
```

6.5 SEMAPHORE USAGE AND CHARACTERISTICS

This paragraph contains a typical example for each of the three types of semaphores and the characteristics of each semaphore type. The macro used for the examples is:

```
ERQ    MACRO                       Macro name ERQ.
       MOVE.L    #\1,D0            Move directive number into D0.
       LEA       \2,A0             Move address of parameter block into A0.
       TRAP      #1                Do TRAP.
       ENDM                        End macro definition.
```

EXAMPLE: Type 1 Semaphore

A type 1 semaphore is used when more than one task requires exclusive access to a single resource. For this example, three tasks, TSKA, TSKB, and TSKC, all need exclusive access to some data area.

```
    TSKA                        TSKB                        TSKC

ERQ ATSEM,R1SEM             ERQ ATSEM,R1SEM             ERQ ATSEM,R1SEM
MOVE.L   A0,R1KEY           MOVE.L   A0,R1KEY           MOVE.L   A0,R1KEY
ERQ WTSEM,R1SEM             ERQ WTSEM,R1SEM             ERQ WTSEM,R1SEM
 .  (Use                     .  (Use                     .  (Use
 .     resource)             .     resource)             .     resource)
 .                           .                           .
ERQ SGSEM,R1SEM             ERQ SGSEM,R1SEM             ERQ SGSEM,R1SEM
```

Each of the tasks would define the semaphore parameter block as:

```
ATSEM     EQU     41          Attach semaphore directive.
WTSEM     EQU     42          Wait on semaphore directive.
SGSEM     EQU     43          Signal semaphore directive number.
R1SEM     DC.L   'SEMX'       Semaphore name.
R1KEY     DC.L    0           Semaphore key.
          DC.B    0           Initial count.
          DC.B    1           Semaphore type.
```

Type 1 semaphore characteristics:

a. The first time an ATSEM is issued, the semaphore is created with a signal count of one, and the semaphore key is returned in A0.

b. Later ATSEM directives cause the semaphore key to be returned in A0.

c. The CRSEM directive is identical to the ATSEM directive for type 1 semaphores.

d. Within any task, each WTSEM directive in a type 1 semaphore must be followed by an SGSEM directive. If a WTSEM or SGSEM directive is not in the proper sequence in a task, it is rejected. If a task detaches from a semaphore before issuing a SGSEM following the last WTSEM, the DESEM directive issues the necessary signal.

e. The semaphore is deleted when the last user detaches.

Figure 6-1 is a diagram of how the tasks are coordinated in time and assumes TSKB was executed first, followed by TSKA and TSKC.

EXAMPLE: Type 2 Semaphore

A type 2 semaphore controls the execution sequence of two or more tasks. For this example, TSKA must complete a process before TSKB can continue and TSKB must complete a process before TSKC can continue. Two type 2 semaphores are used. Macro ERQ is defined at the beginning of this paragraph.

TSKA

```
ERQ CRSEM,R2ASEM
MOVE.L   A0, R2AKEY
    (do processing)
 .
 .
 .
ERQ SGSEM,R2ASEM
ERQ DESEM,R2ASEM
 .
 .
 .
 .
R2ASEM DC.L 'SEMA'
R2AKEY DC.L    0
       DC.B    0
       DC.B    2
```

TSKB

```
ERQ ATSEM,R2ASEM
MOVE.L   A0, R2AKEY
ERQ CRSEM,R2BSEM
MOVE.L   A0, R2BKEY
ERQ WTSEM,R2ASEM
ERQ DESEM,R2ASEM
(do processing)
 .
 .
 .
ERQ SGSEM,R2BSEM
ERQ DESEM,R2BSEM
 .
 .
 .
R2ASEM DC.L 'SEMA'
R2AKEY DC.L    0
       DC.B    0
       DC.B    2
R2BSEM DC.L 'SEMB'
R2BKEY DC.L    0
       DC.B    0
       DC.B    2
```

TSKC

```
ERQ ATSEM,R2BSEM
MOVE.L   A0, R2BKEY
ERQ WTSEM,R2BSEM
ERQ DESEM,R2BSEM
 (do processing)
 .
 .
 .
 .
R2BSEM DC.L 'SEMB'
R2BKEY DC.L    0
       DC.B    0
       DC.B    2


Semaphore name
Semaphore key
Initial count
Semaphore type
```

```
       ACTION ------------------>
      ==============================================================
     |             |               |             |_______RISEM______|
     |             |               |             | INITIAL | SIGNAL |       |
TIME |    TSKA     |    MTSKB      |    TSKC     |  COUNT  |  COUNT | FIFO  |
 |   |==============================================================|
 |   |             | ATSEM         |             |   N/A   |   1    | EMPTY |
 |   |             | WTSEM         |             |         |   0    |       |
 |   |             | (USE RESOURCE)|             |         |        |       |
 |   |             | .             |             |         |        |       |
 |   |             | .             |             |         |        |       |
 |   |             | .             |             |         |        |       |
 V   |             | DELAY         |             |         |        |       |
     |-------------|---------------|             |         |        |       |
     |   ATSEM     |               |             |         |        |       |
     |   WTSEM     |               |             |         |  -1    | TSKA  |
     |-------------|               |-------------|         |        |       |
     |             |               |   ATSEM     |         |        |       |
     |             |               |   WTSEM     |         |  -2    | TSKA, |
     |             |---------------|-------------|         |        | TSKC  |
     |             | (USE RESOURCE)|             |         |        |       |
     |             | .             |             |         |        |       |
     |             | .             |             |         |        |       |
     |             | .             |             |         |        |       |
     |             | SGSEM         |             |         |  -1    | TSKC  |
     |-------------|---------------|             |         |        |       |
     |(USE RESOURCE)|              |             |         |        |       |
     |.            |               |             |         |        |       |
     |.            |               |             |         |        |       |
     |.            |               |             |         |        |       |
     |SGSEM        |               |             |         |   0    |       |
     |-------------|               |-------------|         |        |       |
     |             |               |(USE RESOURCE)|        |        | EMPTY |
     |             |               |.            |         |        |       |
     |             |               |.            |         |        |       |
     |             |               |.            |         |        |       |
     |             |               |SGSEM        |         |   1    |       |
     |==============================================================|
```

NOTE: A SGSEM must follow each WTSEM directive.

FIGURE 6-1. Type 1 Semaphore Usage

6

Type 2 semaphore characteristics:

a. The semaphore is created when the first ATSEM or CRSEM is issued. If an ATSEM is issued first, the signal count is set to 0. If a CRSEM is issued first, the signal count is set to the initial count in the semaphore parameter block. The semaphore key is returned in register A0.

b. Later ATSEM directives return the semaphore key in register A0.

c. Later CRSEM directives reset the initial count value (but not the current signal count), and return the semaphore key in register A0.

d. If the signal count is equal to the initial count set by the last CRSEM directive, the semaphore is deleted when the last user detaches.

e. The number or order of WTSEM and SGSEM directives issued by a given task is not checked.

Figure 6-2 shows how the tasks of the previous example are coordinated in time. Although the tasks can be started in any order, this example shows TSKB starting first, followed by TSKA and TSKC.

EXAMPLE: Type 3 Semaphore

A type 3 semaphore is used when one task has control over a resource that other tasks want to use. In this example, TSKA controls a message buffer (that can contain up to five entries), and processes any messages that have been placed in the buffer. TSKB and TSKC place messages in the buffer. TSKA creates two semaphores. The first, called BUFRDY, controls input to the buffer by its signal count field being equal to the number of buffer entries available. The second, called MSGSNT, is signaled each time a message is placed in the buffer. Using the semaphores in this way prevents TSKB and TSKC from putting messages in a full buffer, and prevents TSKA from taking a message out of an empty buffer. Macro ERQ is defined at the beginning of this paragraph.

```
          ACTION -------------->
      =============================================================================
      |             |            |            |________R2ASEM_______|________R2BSEM_______|
      |             |            |            |INITIAL| SIGNAL|      |INITIAL| SIGNAL|      |
      |    TSKA     |    TSKB    |    TSKC    | COUNT | COUNT | FIFO | COUNT | COUNT | FIFO |
      |=============================================================================|
TIME  |             | ATSEM (A)  |            |       |   0   | EMPTY |       |       | EMPTY |
 |    |             | CRSEM (B)  |            |       |       |       |   0   |   0   |       |
 |    |             | WTSEM (A)  |            |       |  -1   | TSKB  |       |       |       |
 |    |-------------|------------|            |       |       |       |       |       |       |
 |    |CRSEM (A)    |            |            |   0   |       |       |       |       |       |
 |    | (PROCESSING)|            |            |       |       |       |       |       |       |
 |    |.            |            |            |       |       |       |       |       |       |
 V    |.            |            |            |       |       |       |       |       |       |
      |.            |            |            |       |       |       |       |       |       |
      |SGSEM (A)    |            |            |       |   0   |       |       |       |       |
      |DESEM (A)    |            |            |       |       |       |       |       |       |
      |-------------|------------|            |       |       |       |       |       |       |
TDA   |             | DSEM (A)   |            |       |       | EMPTY |       |       |       |
      |             |(PROCESSING)|            |       |       |       |       |       |       |
      |             | .          |            |       |       |       |       |       |       |
      |             | .          |            |       |       |       |       |       |       |
      |             | .          |            |       |       |       |       |       |       |
      |             |SGSEM (B)   |            |       |       |       |       |   1   |       |
      |             |DESEM (B)   |            |       |       |       |       |       |       |
      |             |------------|------------|       |       |       |       |       |       |
      |             |            | ATSEM (B)  |       |       |       |       |       |       |
      |             |            | WTSEM (B)  |       |       |       |       |   0   |       |
TDB   |             |            | DESEM (B)  |       |       |       |       |       |       |
      |             |            |(PROCESSING)|       |       |       |       |       |       |
      |             |            | .          |       |       |       |       |       |       |
      |             |            | .          |       |       |       |       |       |       |
      |             |            | .          |       |       |       |       |       |       |
      |             |            |----------- |       |       |       |       |       |       |
      |             |            |            |       |       |       |       |       |       |
      |             |            |            |       |       |       |       |       |       |
      |             |            |            |       |       |       |       |       |       |
      |             |            |            |       |       |       |       |       |       |
      |             |            |            |       |       |       |       |       |       |
      =============================================================================

       TDA = Time of R2ASEM deletion
       TDB = Time of R2BSEM deletion
       Task B must wait until task A finishes
```

FIGURE 6-2. Type 2 Semaphore Usage

```
      TSKA                          TSKB                          TSKC

  ERQ CRSEM,BUFRDY             ERQ ATSEM,BUFRDY             ERQ ATSEM,BUFRDY
  MOVE.L     A0, BUFKEY        MOVE.L  A0, BUFKEY           MOVE.L A0,BUFKEY
  ERQ CRSEM,MSGSNT             ERQ ATSEM,MSGSNT             ERQ ATSEM,MSGSNT
  MOVE.L     A0, MSGKEY        MOVE.L  A0, MSGKEY           MOVE.L A0,MSGKEY
WT:ERQ WTSEM,MSGSNT          SD:ERQ WTSEM,BUFRDY          SD:ERQ WTSEM,BUFRDY
  (process message)            (put message                 (put message
   .                              into buffer)                 into buffer)
   .                            .                            .
   .                            .                            .
  ERQ SGSEM,BUFRDY              .                            .
  BRA WT                       ERQ SGSEM,MSGSNT             ERQ SGSEM,MSGSNT
   .                            .                            .
   .                            .                            .
   .                            .                            .
BUFRDY: DC.L 'BFRY'            BRA SD                       BRA SD
BUFKEY: DC.L   0                .                            .
        DC.B   5                .                            .
        DC.B   3                .                            .
MSGSNT: DC.L 'MSSG'          BUFRDY: DC.L 'BFRY'          BUFRDY: DC.L 'BFRY'
MSGKEY: DC.L   0             BUFKEY: DC.L   0             BUFKEY: DC.L   0
        DC.B   0                     DC.B   5                     DC.B   5
        DC.B   3                     DC.B   3                     DC.B   3
                             MSGSNT: DC.L 'MSSG'          MSGSNT: DC.L 'MSSG'
                             MSGKEY: DC.L   0             MSGKEY: DC.L   0
                                     DC.B   0                     DC.B   0
                                     DC.B   3                     DC.B   3
```

Type 3 semaphore characteristics:

a. Only a CRSEM directive can create a semaphore. The signal count is initialized to the value in the initial count field of the semaphore parameter block, and the semaphore key is returned in register A0.

b. Later CRSEM directives are rejected.

c. ATSEM directives issued before the semaphore is created causes the issuing task to be placed in a WAIT state. When another task creates the semaphore, the semaphore key is returned to each task that had attached to the semaphore.

d. The semaphore is deleted when the task that created the semaphore terminates or issues a DESEM directive. An error code is returned for any task in the wait list at the time the semaphore is deleted.

e. The number or order of WTSEM and SGSEM directives issued by a given task are not checked.

Figure 6-3 shows how the tasks of the above example can be coordinated in time. Although the tasks could be started in any order, the diagram shows TSKB starting first, followed by TSKA and TSKC.

ACTION -------------->

				BUFRDY			MSGSNT		
	TSKA	TSKB	TSKC	INITIAL COUNT	SIGNAL COUNT	FIFO	INITIAL COUNT	SIGNAL COUNT	FIFO
TIME		ATSEM (BUFRDY)				EMPTY			EMPTY
|	CRSEM (BUFRDY)			5	5		0	0	
|	CRSEM (MSGSNT)							-1	TSKA
|	WTSEM (MSGSNT)								
|		ATSEM (MSGSNT)							
V		WTSEM (BUFRDY)			4				
		(PLACE MESSAGE IN BUFFER)							
		SGSEM (MSGSNT)						0	EMPTY
	(PROCESS MESSAGE)								
	SGSEM (BUFRDY)				5				
	WTSEM (MSGSNT)							-1	TSKA
			ATSEM (BUFRDY)						
			ATSEM (MSGSNT)						
			WTSEM (BUFRDY)		4				
			(PLACE MESSAGE IN BUFFER)						
			SGSEM (MSGSNT)					0	
			WTSEM (BUFRDY)		3				
			(PLACE MESSAGE IN BUFFER)						
			SGSEM (MSGSNT)					1	EMPTY
	PROCESS MESSAGE								
	.								
	.								
	.								

Task A controls a resource which task B and task C will use.

FIGURE 6-3. Type 3 Semaphore Usage

THIS PAGE INTENTIONALLY LEFT BLANK.

CHAPTER 7

TRAP SERVER MANAGER

7.1 OVERVIEW

Most operating systems require a class of tasks that provide services and control resources on a system wide basis. These server tasks should be able to respond to a request from any task in the system, execute the requested service, and provide feedback to the client task about the success or failure of the request. A typical use for a server task would be to implement an input-output system, a file management system, or a database manager.

The RMS68K Trap Server Manager supports this class of server tasks by providing them with the following privileges and capabilities:

a. A server task can respond to a request from any task in any session.

b. The event that results from the request contains the client task's task_id, the directive number of the requested service, and a copy of the parameter block describing the request, if required.

c. The server can prohibit the client from running until the service is completed.

d. The server can provide feedback to the client by altering its condition codes and two of its registers.

e. The server can specify what state the client task should be in when released from the server's control.

f. The server can request notification of task termination to implement any necessary termination processing.

The VERSAdos real-time operating system is implemented as a collection of server tasks running under the control of RMS68K. Therefore, server tasks can be used to implement a special purpose operating system or to add domain specific extensions to VERSAdos.

7.2 THEORY OF OPERATION

A task becomes a server task by informing the trap server manager that it wants to service a particular trap instruction. Thereafter, any task can request its services by loading a directive number into D0, optionally pointing A0 to a parameter block, and executing the trap instruction. The trap server manager responds by placing the client task into a WAITING ON ACKNOWLEDGEMENT from the server state, and queuing an event to the server containing the trap number, clients task_id, D0 and A0 registers, and optionally, the entire parameter block.

Then the server task can process the event, either synchronously or asynchronously. When the request is satisfied, the server can report on the status of the request within the client's condition codes and D0 register, return a value in its A0 register, and enable it to run by acknowledging the request. This acknowledgement also informs the trap server manager that the server is ready to process another request for service.

This server task interface was designed to be identical to the interface to RMS68K. Therefore, server tasks appear to their clients as extensions of the Executive.

The preceding paragraphs describe one typical use of the trap server manager. There are several variations of this scheme that are explained later such as:

a. A server task that responds to more than one trap number.

b. A server task responding to task termination events.

c. A server informing the trap server manager that it is ready to process another request before acknowledging a previous one.

d. A server placing a task into a WAIT or SUSPEND state on acknowledgement.

7.2.1 Trap Server Manager Directives

The directives used for server task control are:

SERVER	A task establishes itself as a server task.
AKRQST	A server task acknowledges receipt of completion of a request by placing the requesting task into an appropriate state.
DERQST	A server task controls the receipt of requests for service.
DSERVE	A server task initiates orderly shutdown of service.

7.2.2 Server Tasks and Session Boundaries

A server task can either be a system task or a user task. Most servers are implemented as system tasks because they can respond to and acknowledge requests from tasks within all sessions, whereas user server tasks can only respond to and acknowledge requests from tasks within their own session. One use of user server tasks would be to customize different sessions or restrict access to a classified server task to tasks executing within a privileged session.

Because of the data structure format that associates a trap number with a server task, only one server can be associated with a specific trap instruction at any moment, regardless of whether the server is a system or user task. Therefore, two user server tasks in different sessions could not respond to the same trap number.

7.2.3 Server/Client Communication

A client task communicates to a server by executing a trap instruction that causes the trap server manager to build and queue an event describing the trap request to the server. If the server is implemented as a system task, it receives the event even if the client is a user task executing in an alien session. This represents an extension to the rules for queueing events because a user task can not queue an event across session boundaries, regardless of whether the recipient is a system or user task.

The event built by the trap server manager to describe the request consists of information describing the client:

a. Task_id
b. System/User task status
c. Real-time/Non Real-time status

and information describing the request:

a. Trap number
b. Directive number (D0)
c. Parameter (A0)
d. Optional Parameter Block pointed to by A0

and information about the status of the request:

a. The total parameter block was moved.
b. Part of the parameter block was moved.
c. A bad parameter block was specified so nothing was moved.

The server task can process this event in either the synchronous or asynchronous mode, depending on whether its Asynchronous Service Routine (ASR) is disabled or enabled. To process server events in the asynchronous mode, the server must inform the trap server manager where it wants to be dispatched when a server event arrives. This routine may or may not be equivalent to the servers default ASR, and a server that wants to handle multiple trap instructions can declare a different service routine for each trap number. Likewise, a server that wants to service multiple trap instructions in the synchronous mode can distinguish between them by decoding the trap number within the server event.

The server communicates status and returned values back to the client via the AKRQST directive. AKRQST allows the server to set the client's condition codes and thereby support the familiar interface to RMS68K:

```
    MOVE.W     #Directive_Number, D0
    LEA        Parameter_Block, A0
    TRAP       #Trap_Number
    BNE        Decode_Error
```

In addition, the server can return an error code in D0 and an optional returned value in A0.

AKRQST also allows the server to place the task into one of the following states:

READY
WAIT
SUSPEND

Normally, a server makes the client READY on acknowledgement. However, in systems where the server just initiates the service and another task or device driver completes it, the server can choose to acknowledge the request and leave the client in the WAIT or SUSPEND state to be re-activated by the task or driver on completion of the service.

7.2.4 Server Request Control

Another requirement of server tasks is that they be allowed to specify when they are ready to process trap requests. One trap server may dictate that it only process one request at any moment; another may be capable of simultaneously handling "n" requests. The trap server manager supports both implementations via the AKRQST and DERQST directives.

Basically the server request control mechanism is implemented as a gate, external to the server's Asynchronous Service Queue (ASQ). When the gate is open and a task executes a trap instruction, the trap server manager creates an event describing the request, queues it to the server's ASQ, and closes the gate. Any requests issued while the gate is closed are placed on a secondary queue waiting for the server task to tell the trap server manager to open the gate. Note that this gate is selective and only holds out server events generated in response to trap instructions or task termination. All other communications from drivers, dependent subtasks, or other servers proceed according to the normal rules for ASQ event processing. This gate protects the server's ASQ from being inundated with requests for service and therefore unable to communicate with other operating system entities.

If the trap server has completed servicing a request, it can release the client task and tell the trap server manager to open the gate by issuing the AKRQST directive. This is a typical implementation of a server that is limited to processing one request at a time.

Some servers can process "n" requests concurrently. They typically accept a request, do some validation and pre-processing, and pass the request on to another task or device driver. At this point they are capable of processing another request even though the subtask or driver is still working on the first request, so they execute the DERQST directive with the enable bit set. This tells the trap server manager to open the gate and allow the next request to enter the server's ASQ while keeping the previous client task in the WAITING ON ACKNOWLEDGEMENT state. When the subtask or driver queues an event indicating that the request is complete, the server can execute the AKRQST directive to release the client. The AKRQST that follows a DERQST does not cause the gate to open.

A third class of servers executes the request in parallel with the continued execution of the client. These servers might notify the client asynchronously when the request is complete or only notify the client if there was some problem with satisfying the request. One example of this type of client/server relationship would be a data acquisition system where the client was responsible for the real-time acquisition of the data and the server was responsible for logging the data to a low performance output device.

To implement this system, the client would collect and pre-process the data and then execute a trap instruction to request that the server task log the data. The server would verify the request and immediately acknowledge the client, perhaps returning a pointer in A0 to a free buffer for collecting the next piece of data. After releasing the client to collect more data, the server would begin the slower output operation. When the output is completed, the server would queue an event to the client describing the status of the request that the client could process synchronously or asynchronously. A variation on this scheme would be for the server and client to assume that the request would succeed, and the server would only have to notify the client if there was some problem with the request.

7.2.5 Termination Control

A server task can request that the trap server manager notify it when any task is terminating. Here, the trap server manager places the terminating task into the WAIT ON ACKNOWLEDGEMENT state and queues an event to the server. The server may then execute any required termination processing such as aborting any outstanding I/O, releasing any resources currently allocated to the task, or deleting all reference to the task from the server's internal data structures. When the server has completed its termination processing, it can release the task to finish its termination by executing the AKRQST directive. The task will not terminate or be purged from the system until the server executes the AKRQST in response to the termination event.

A monitor task also receives a termination event when any of its subtasks terminates; this event is for information only and does not stop the subtask from terminating or place it in any kind of WAIT state. Thus, if the monitor is running at a lower priority than the subtask, it may not be dispatched to process the termination event until after the subtask has been purged from the system.

A termination server task could be implemented that provides no services other than monitoring the system for task termination. Since in many real-time systems, tasks are never supposed to terminate, this termination server could play an important role in system reliability and security.

7.3 DATA STRUCTURES

The Trap Instruction Owner Table (TIOT) is an array of descriptors indexed by trap number describing each trap server and the state of the gate controlling access to its ASQ for server events. The TIOT is contained within the SYSPAR system parameter area and consists of one 22-byte entry for each of the 16 trap instructions. An entry is defined as:

TIOTTCB	(4 bytes)	Server task TCB address
TIOTSESS	(4 bytes)	Server task sessions number
TIOTSEM	(6 bytes)	Semaphore used to limit access to the server task's ASQ
TIOTADDR	(4 bytes)	Server task's ASR address
TIOTMCNT	(2 bytes)	Count of unacknowledged messages
TIOTSTAT	(1 byte)	Status

Bit 15=1	Server function enabled
Bit 14=1	Server wants termination notification
Bit 13=1	Server wants parameter block moved with message
Bit 12=1	Message sent to server, ACK pending
Bit 11=1	DERQST called while ACK pending
Bits 10-0	Reserved

TIOTPBSZ	(1 byte)	Parameter block size

7.4 TRAP SERVER MANAGER DIRECTIVES

The detailed descriptions contained within the trap server manager are described on the following pages.

ESTABLISH SERVER SERVER

Directive Number: 51

Parameter: Server Block Address

Server Block:

Request Service Address (4 bytes)	Address at which specified trap instruction is to be serviced. If this field is 0, then the default ASR address is used.	
Trap Instruction Identifier (1 byte)	Bit 7	Reserved.
	Bit 6=0	Receive event only when task executes a trap instruction.
	=1	Task elects to also receive an event each time a task terminates. (This option is only available to system tasks.)
	Bit 5=1	Server wants a parameter block with the event.
	Bit 4	Reserved.
	Bits 3-0	Specify the number of the trap instruction that the requesting task serves at the above request service address. Valid values are 2 through 15.
Parameter Block Size (1 byte)	Required if trap instruction identifier bit 5=1. Specifies size of parameter block that is attached to the end of the event.	

Detailed Description:

RMS68K establishes the requesting task as a server task of the trap instruction specified in the parameter block and/or terminations. Any task can request the services of the server task by executing the appropriate trap instruction. If a task has elected local processing of a specific trap instruction via the TRPVCT directive, the local processing overrides the server task processing of the trap instruction.

A request for service manifests itself to the server task as an event message with event code = $07. Auxiliary information can be passed to the server task through a parameter block. The address of the parameter block would be contained in the register A0 field of the server event (event code = $07). The server task can elect to automatically receive the parameter block in its event receiving area by setting trap instruction identifier bit 5 = 1. If this method is elected, the parameter block immediately follows the event text. A server task could also elect to obtain the parameter block on its own by issuing a MOVELL directive.

A server task can process requests synchronously or asynchronously. If asynchronous processing is desired, the request service address in the parameter block specifies the beginning address of the routine to be activated when a request event causes an ASR interrupt. This request service routine can be dedicated to servicing a request or the same routine that processes all other events.

There can only be one server task for a given trap number in the entire system. If the server is a system task, then tasks in any session can issue that trap and the server will receive it. If the server is a user task, then only tasks in his session can issue that trap. Tasks in other sessions receive an error if they attempt the trap call.

WARNING

SERVER TASK EVENTS (CODE = $07) ARE 24 BYTES LONG. A SERVER TASK'S ASQ MUST HAVE A MAXIMUM MESSAGE LENGTH OF AT LEAST 24 BYTES.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
4/$04	Requestor has no ASQ.
6/$06	Specified trap instruction not available (TRAP #0, TRAP #1, or some other server has trap).
12/$0C	Request service address not in requestor's address space.

EXAMPLE:

TSKA wants to establish itself as the server for TRAP #3 and TRAP #7 instructions and for all terminations within its session.

```
TSKA:       .
            .
            .
          MOVE.L   #51,D0         Load SERVER directive number 51.
          LEA      SVB1,A0        Load parameter block address.
          TRAP     #1
          BNE      FAULT          Branch, if error.

          MOVE.L   #51,D0         Load SERVER directive number 51.
          LEA      SVB2,A0        Load parameter block address.
          TRAP     #1
          BNE      FAULT          Branch, if error.
            .
            .
            .
SVB1:     DC.L     REQ3           Address at which TRAP #3 is served.
          DC.B     $43            Receive an event on terminate, no
                                  parameter block, TRAP #3.
          DC.B       0            N/A; no parameter block.

SVB2:     DC.L     REQ7           Address at which TRAP #7 is served.
          DC.B       7            Event only when task requests, no
                                  parameter block, TRAP #7.
          DC.B       0            N/A; no parameter block.
```

ACKNOWLEDGE SERVICE REQUEST AKRQST

Directive Number: 54

Parameter: Parameter Block Address

Parameter Block:

Target Task (8 bytes)	Task_id of task whose request is being acknowledged.	
Directive Option (2 bytes)		
	Bit 15	Reserved.
	Bit 14=1	Set target task's condition codes in status register as specified.
	Bit 13=1	Set target task's register D0 to value specified.
	Bit 12=1	Set target task's register A0 to value specified.
	Bits 11-9 = 1xx	Reactivate target task.
	= 01x	Place target task in WAIT state.
	= 001	Place target task in SUSPEND state.
	Bits 8-0	Reserved.
Trap Number (1 byte)	Trap number being acknowledged.	
Condition Codes (1 byte)	Supplied if option bit 14=1.	
Register D0 (4 bytes)	Supplied if option bit 13=1.	
Register A0 (4 bytes)	Supplied if option bit 12=1.	

Detailed Description:

A task that has issued a trap handled by a trap server is in the WAITING ON ACKNOWLEDGEMENT state and cannot execute until another task does an AKRQST to release it. The task that does the AKRQST must be a system task or in the same session as the trap server; it does not have to be the server itself.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.
7/$07	Specified trap not dedicated to acknowledging task's session.
10/$0A	Target task is not waiting to be served for specified trap.

EXAMPLE:

A non real-time server task, TSKA, has finished processing a TRAP #5 request from TSKB, and wants to reactivate TSKB. The condition codes of the SR, register D0, and register A0 are all left unaltered.

```
TSKA:           .
                .
                .
               MOVE.L   #54,D0        Load AKRQST directive number 54.
               LEA      PRMBLK,A0     Load parameter block address.
               TRAP     #1            Reactivate target task; SR. D0, A0 unaltered.
               BNE      FAULT         Branch, if error.
                .
                .
                .
PRMBLK:        DC.L     'TSKB'        Target taskname.
               DC.L     1             Session number of target task.
               DC.W     $0800
               DC.B     5             Trap number acknowledged.
               DC.B     0             N/A; bit 14 is clear.
               DC.L     0,0           N/A; bit 13 and 12 are clear.
```

SET USER/SERVER REQUEST STATUS — DERQST

Directive Number: 53

Parameter: Trap Number and Status

Bits 31-8	Reserved
Bit 7=1	Enable request receipt.
=0	Disable request receipt.
Bits 6-4	Reserved
Bits 3-0	Trap number of interest (values 2 through 15).

Detailed Description:

RMS68K enables or disables the server request control mechanism according to the parameter.

When a request caused by a given trap instruction is queued to a server task, the server's ASQ is disabled for further requests from that trap instruction, (that are queued), until the server indicates that it is again ready to process those trap requests. The DERQST directive with the enable bit set, enables the server's ASQ to receive those trap requests.

Unless the request receipt was disabled by the DERQST directive, request receipt for a given request type is automatically re-enabled when an acknowledgement is made for a request of that type.

Return Parameter: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
7/$07	Specified trap not assigned to task issuing directive.

EXAMPLE:

After reading an event for a TRAP #5 instruction, the server task, TSKA wants to re-enable receipt of TRAP #5 requests into its ASQ.

```
     TSKA:       .
                 .
                 .
            MOVE.L    #53,D0          Load DERQST directive number 53.
            MOVE.W    #$0085,A0       Enable request receipt, TRAP #5
            TRAP      #1
            BNE       FAULT           Branch, if error.
                 .
                 .
                 .
```

DEALLOCATE SERVER FUNCTION — DSERVE

Directive Number:	52	
Parameter:	Bits 31-4	Reserved
	Bits 3-0	Relevant Trap Instruction Number

Detailed Description:

A server task initiates orderly shutdown of service. Any request events for the specified trap instruction already in the ASQ but not serviced, continue to assert themselves in order. Any pending requests not yet in the ASQ are treated as if the server never existed. The trap instruction is detached from the requesting task.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
7/$07	Specified trap instruction not dedicated to the task issuing the directive.

EXAMPLE:

TSKA no longer wants to be the server task for TRAP #6 instructions.

```
    TSKA:          .
                   .
                   .
                 MOVE.L    #52,D0           Load DSERVE directive number 52.
                 MOVE.W    #6,A0            TRAP number 6.
                 TRAP      #1
                 BNE       FAULT            Branch, if error.
                   .
                   .
                   .
```

CHAPTER 8

EXCEPTION MONITOR MANAGER

8.1 OVERVIEW

The Exception Monitor Manager provides services to a class of tasks called exception monitors. An exception monitor is a task that can indirectly observe and control the execution of a target task. The events that an exception monitor is capable of observing are those events (called exceptions) that cause the processor to switch from executing user mode code to supervisor mode code. These exceptions include all the TRAP instructions and error conditions such as bus error, illegal instruction, and divide by zero.

The exception monitor can control the target task by reading or writing to its memory or registers, setting breakpoints, or tracing through the program in one of three trace modes.

A typical use for exception monitors is to implement symbolic debuggers or timing analysis programs. Another use would be to increase system security by observing and reporting on erratic task behavior.

8.1.1 Services

Listed below is a summary of the directives contained within the exception monitor manager.

EXMON	A target task becomes associated with an exception monitor task.
EXMMSK	Events of interest to an exception monitor task are specified.
REXMON	A target task executes as directed by an exception monitor task.
RSTATE	An exception monitor task receives the current state of one of its target tasks.
PSTATE	An exception monitor task sets the current state of one of its target tasks.
DEXMON	A target task detaches from an exception monitor task.

8.2 THEORY OF OPERATION

A target task is attached to an exception monitor when some task, either the target, the exception monitor, or a third task executes an EXMON directive. In response to the EXMON directive, the exception monitor manager places the target into the WAIT FOR COMMAND state, establishes the connection between the target task and the exception monitor and queues an event describing this relationship to the exception monitor.

Once attached, the exception monitor can inform the exception monitor manager which exceptions it wants to observe by setting bits in the exception monitor mask corresponding to those exceptions and then executing the EXMMSK directive.

The exception monitor can now start the target task executing under the control of the exception monitor via the REXMON directive. REXMON supports four modes of target task execution:

1. Run until an enabled exception occurs.
2. Trace one instruction.
3. Run until a location within the target task's memory changes value or an enabled exception occurs.
4. Run until a location within the target task's memory equals a specified value or an enabled exception occurs.

For modes 3 and 4 the exception monitor may also specify a maximum number of instructions. If the target executes this number of instructions before the location changes or equals the specified value, or an enabled exception occurs, the exception monitor is notified by an appropriate event.

If an enabled exception occurs, or one of the trace conditions is satisfied, the exception monitor manager places the target task into the WAIT FOR COMMAND state, builds an event describing the exception or trace condition, and queues the event to the exception monitor. The exception monitor can service this event synchronously or asynchronously depending on whether its ASR is disabled or enabled.

The exception monitor can query the state of the target task at any time, even while it is running, via the RSTATE and MOVELL directives. RSTATE returns a copy of the task's registers, status register, program counter, task state, and variables associated with REXMON's trace modes. The exception monitor can read the state of the target task's program or data memory via the MOVELL directive.

Likewise, the exception monitor can update the state of the target task via the PSTATE and MOVELL directives. PSTATE allows the exception monitor to write into the target task's registers, status register, or program counter, and update the mask of enabled exceptions. MOVELL can be used to write into the target task's program or data space.

One use of MOVELL is to create breakpoints within the target task's program space. For example, to create a breakpoint at location $1000, the exception monitor reads location $1000 via MOVELL, saves the current instruction, and uses MOVELL to place an instruction in $1000 that would cause an exception (such as an illegal instruction). If the target task executes the illegal instruction, the exception monitor is notified and replaces the original instruction via MOVELL.

After any querying or updating of the target task's state, the exception monitor can restart the target via the REXMON directive, either from the point at which the exception occurred, or from the location written into the target's program counter via the PSTATE directive. This process continues until the exception monitor detaches from the target via the DEXMON directive.

8.3 DATA STRUCTURES

The exception monitor manager maintains the following fields within the target task's TCB.

TCBEXM	(4 bytes)	Exception monitor taskname.
TCBEXMS	(4 bytes)	Exception monitor session number.
TCBEMMSK	(4 bytes)	Exception monitor mask.
TCBEVMSK	(4 bytes)	Exception monitor value mask.
TCBEVLOC	(4 bytes)	Exception monitor value address.
TCBEVALU	(4 bytes)	Exception monitor value content.
TCBECNT	(4 bytes)	Exception monitor maximum number of instructions.

8.4 EXCEPTION MONITOR MANAGER DIRECTIVES

The exception monitor manager directives are described in detail on the following pages.

ATTACH EXCEPTION MONITOR EXMON

Directive Number: 64

Parameter: Parameter Block Address

Parameter Block:

Target Task (8 bytes)	Task_id of target task being attached to exception monitor.
Exception Monitor (8 bytes)	Task_id of exception monitor to attach to.

Detailed Description:

RMS68K attaches the target task to the exception monitor task and places the target task in the WAIT FOR COMMAND state. An event with event code = $08, indicating the attach, is queued to the ASQ of the exception monitor. Refer to Chapter 2 for event format.

This directive can be issued by the target task, the exception monitor task, or a third task. If the target task does not issue the directive, it must be in the DORMANT state.

WARNING

EXCEPTION MONITOR EVENTS (CODE = $08) ARE 12 BYTES LONG. A MONITOR'S ASQ MUST HAVE A MAXIMUM MESSAGE LENGTH OF AT LEAST 12 BYTES.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.
5/$05	ASQ of exception monitor is full or not enabled.
6/$06	Target task is already attached to an exception monitor.
7/$07	Exception monitor does not exist.
9/$09	Target task is a system task or is not in DORMANT state.

EXAMPLE:

TSKA, a non real-time user task, wants to attach TSKC to the exception monitor task, TSKB.

```
TSKA:          .
               .
               .
          MOVE.L   #64,D0         Load EXMON directive number 64.
          LEA      PRMBLK,A0      Load parameter block address.
          TRAP     #1
          BNE      FAULT          Branch, if error.
               .
               .
               .
PRMBLK:   DC.L     'TSKC'         Task to be attached to exception
                                  monitor.
          DC.L       0            N/A; user task.
          DC.L     'TSKB'         Exception monitor taskname.
          DC.L       0            N/A; user task.
```

8

SET EXCEPTION MONITOR MASK EXMMSK

Directive Number: 66

Parameter: Parameter Block Address

Parameter Block:

Target Task (8 bytes)	Task_id of target task receiving mask.

Exception Monitor Mask (4 bytes)

An exception monitor mask is associated with a target task that is to be controlled by an exception monitor task. This mask specifies which exceptions are to cause the execution of the target task to cease and notification to be sent to the exception monitor.

Each bit of the mask corresponds to a particular exception. If a bit is set, the associated exception is enabled.

The bits and associated exceptions are:

BIT	EXCEPTION	BIT	EXCEPTION
0	Reserved	16	Bus Error
1	TRAP 1*	17	Address Error
2	TRAP 2	18	Illegal Instruction
3	TRAP 3	19	Zero Divide
4	TRAP 4	20	CHK Instruction
5	TRAP 5	21	TRAPV
6	TRAP 6	22	Privilege Violation
7	TRAP 7	23	Line 1010 Emulator
8	TRAP 8	24	Line 1111 Emulator
9	TRAP 9	25	Reserved
10	TRAP 10	26	Reserved
11	TRAP 11		
12	TRAP 12		
13	TRAP 13		
14	TRAP 14		
15	TRAP 15		

Bits 27-31 are used by RMS68K for execution control events.

*The exception monitor is not notified of a target task executing a TRAP #1 instruction if the target is executing within the real-time domain.

Detailed Description:

The specified mask is attached to the target task. When an enabled exception occurs within the target task, the target task is placed in the WAIT FOR COMMAND state and an appropriate message is queued to the target task's exception monitor.

It is not required that the target task be attached to an exception monitor when this directive is issued; the mask has no effect (no exception monitor action takes place).

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.

EXAMPLE:

A user task, TSKA, wants TRAP #2 and bus error exceptions of TSKB to be relevant to the exception monitor task of TSKB.

```
TSKA:         .
              .
              .
          MOVE.L    #66,D0        Load EXMMSK directive number 66.
          LEA       PRMBLK,A0     Load parameter block address.
          TRAP      #1
          BNE       FAULT         Branch, if error.
              .
              .
              .
PRMBLK:   DC.L      'TSKB'        Task to receive mask.
          DC.L        0           N/A; user task.
          DC.L      $00010004     Mask, TRAP #2, bus error.
```

RUN TASK UNDER EXCEPTION MONITOR CONTROL — REXMON

Directive Number: 69

Parameter: Parameter Block Address

Parameter Block:

Target Task (8 bytes)	Task_id of target task to be run.
Buffer Address (4 bytes)	Buffer containing execution control information.

Buffer Contents:

Execution Options (2 bytes)	Bits 15-12 = 0000	Normal execution.
	= 0001	Execute 1 instruction.
	= 0010	Value change trace.
	= 0011	Value equal trace.
	Bit 11 = 1	Maximum instruction count is specified.
	Bits 10-0	Reserved
Value Location (4 bytes)	Required when options bits 15-12 equal 0010 or 0011.	
Value (4 bytes)	Required when options bits 15-12 equal 0011.	
Value Mask (4 bytes)	Required when options bits 15-12 equal 0010 or 0011.	
Maximum Instruction (4 bytes) Count	Required when options bit 11 = 1.	

Detailed Description:

An exception monitor task specifies how a target task is to be executed. There are four modes of operation that can be selected in the options field; the remainder of information in the buffer depends on the mode selected. Following is a description of each of the four modes and the information required:

Normal Execution:

If this mode is selected, no other information is required in the buffer. The target task executes until the occurrence of an exception that has been enabled by the target task's exception monitor mask. The target task then goes to the WAIT FOR COMMAND state and an appropriate event is queued to the exception monitor task's ASQ.

Execute One Instruction:

If this mode is selected, no other information is required in the buffer. The target task executes one instruction. The target task then goes into the WAIT FOR COMMAND state, and an appropriate event is queued to the exception monitor task's ASQ.

Value Change Trace:

If this mode is selected, value location, value mask, and optionally maximum instruction count must be provided in the buffer. The target task runs until whichever of the following occurs first:

a. The location specified in the value location changes value.

b. An exception monitor mask exception occurs.

c. If options bit 11=1, the number of instructions specified by maximum instruction count have executed.

The value checked in a. above is a 4-byte field beginning at the specified even value location. The value mask indicates which bits are to be included in the check. For example, a value mask of $FF0000FF causes execution to stop when either the first or last byte changes. When any of the above three incidents occurs, the target task goes to the WAIT FOR COMMAND state and an appropriate event is queued to the exception monitor task's ASQ.

Value Equal Trace:

If this mode is selected, value location, value, value mask, and optionally maximum instruction count must be provided in the buffer. The target task runs until whichever of the following comes first:

a. The contents of the location specified in value location equals the specified value.

b. An exception monitor mask exception occurs.

c. If options bit 11=1, the number of instructions specified by maximum instruction count have executed.

The value checked in a. is a 4-byte field beginning at the specified even value location. The value mask indicates which bits are to be included in the check. For example, if value = $12345678 and value mask = $FF0000FF, execution stops when the contents of the specified location are equal to $12xxxx78. When any of the previous three incidents occurs, the target task goes to the WAIT FOR COMMAND state and an appropriate event is queued to the exception monitor task's ASQ.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist or privilege denied.
10/$0A	Target task not attached to requesting exception monitor.
12/$0C	Buffer not in requestor's address space.
15/$0F	Value location not in requestor's address space.

EXAMPLE:

A user task, TSKA, wants to run task TSKB under monitor control until the value located at address STAT changes.

```
TSKA:      .
           .
           .
           MOVE.L   #69,D0          Load REXMON directive number 69.
           LEA      PRMBLK,A0       Load parameter block address.
           TRAP     #1
           BNE      FAULT           Branch, if error.
           .
           .
           .
PRMBLK:    DC.L     'TSKB'          Target task to run.
           DC.L       0             N/A; user task.
           DC.L     EXBUF           Pointer to buffer with execution
                                    information.
EXBUF:     DC.W     $2000           Value change trace; no maximum
                                    count.
           DC.L     STAT            Value location.
           DC.L       0             N/A; value change trace.
           DC.L     $FFFFFFFF       Mask value.
           DC.L       0             N/A; no mask count.
```

RECEIVE TASK STATE — RSTATE

Directive Number: 67

Parameter: Parameter Block Address

Parameter Block:

Target Task (8 bytes)	Task_id of target task.
Buffer Address (4 bytes)	Logical address of buffer to receive task state of target task.

Buffer Contents:

```
D0       (4 bytes)
D1       (4 bytes)
.
.
.
D7       (4 bytes)
A0       (4 bytes)
A1       (4 bytes)
.
.
.
A7(USP)  (4 bytes)
PC       (4 bytes)
SR       (2 bytes)
```

Exception Monitor Mask (4 bytes)

Task Status (4 bytes) — Current task state is indicated by bits 31-18, each set bit corresponding to the following task state:

BIT	TASK STATE
31	Task is in DORMANT state
30	Task is in WAIT state
29	Task is in WAIT ON SEMAPHORE state
28	Task is in WAIT FOR EVENT state
27	Task is in WAIT FOR SERVICE REQUEST ACKNOWLEDGMENT state
26	Task is in WAIT FOR COMMAND state
25	Task is in SUSPEND state
24	Reserved
23	Task has pending termination
22	Task will return to RMS68K
21	Task is headed for ASR

20	Task is in READY state
19	Task has PENDING WAKEUP
18	Termination message sent to server while acknowledgement is outstanding
17-0	Reserved

Execution Options (2 bytes)	Refer to REXMON directive.
Value Location (4 bytes)	Refer to REXMON directive.
Value (4 bytes)	Refer to REXMON directive.
Value Mask (4 bytes)	Refer to REXMON directive.
Maximum Instruction (4 bytes) Count	Refer to REXMON directive.

Detailed Description:

An exception monitor receives the current state of a target task. This current state information includes the data registers, address registers, user stack pointer, program counter, status register, exception monitor mask, task state, and execution control fields. The entire information uses 96 bytes of space.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist or privilege denied.
10/$0A	Target task not attached to the requesting exception monitor.
12/$0C	Buffer not in requestor's address space.

EXAMPLE:

A non real-time system task, TSKA, is the exception monitor for TSKB. TSKA wants to examine the current state of TSKB.

```
TSKA:         .
              .
              .
              MOVE.L   #67,D0            Load RSTATE directive number 67.
              LEA      PRMBLK,A0         Load parameter block address.
              TRAP     #1
              BNE      FAULT             Branch, if error.
              .
              .
              .
PRMBLK:       DC.L     'TSKB'            Target taskname.
              DC.L       0               Same session; supervisor task.
              DC.L     STBUF             Address in which to store data.
STBUF:        DS.B      96               Receiving buffer is 96 bytes.
```

PUT TASK STATE — PSTATE

Directive Number: 68

Parameter: Parameter Block Address

Parameter Block:

Target Task (8 bytes)	Task_id of target task.
Buffer Address (4 bytes)	Pointer to new state information buffer.

Buffer Contents:

```
D0        (4 bytes)
D1        (4 bytes)
.
.
.
D7        (4 bytes)
A0        (4 bytes)
A1        (4 bytes)
.
.
.
A7(USP)   (4 bytes)
PC        (4 bytes)
SR        (2 bytes)

Exception Monitor Mask (4 bytes)
```

8

Detailed Description:

An exception monitor can change the state of a target task by changing the values of the target task's data registers, address registers, user stack pointer, program counter, status register, and exception monitor task.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist or privilege denied.
10/$0A	Target task is not attached to this exception monitor.
12/$0C	Buffer is not in requestor's address space.

EXAMPLE:

TSKA is the exception monitor of TSKB. TSKA wants to change the exception monitor mask of TSKB so that only bus errors are relevant. TSKA must first get the current state information of TSKB and then make the necessary changes.

```
TSKA:      .
           .
           .
           MOVE.L   #68,D0              Load PSTATE directive number 68.
           LEA      PRMBLK,A0           Load parameter block address.
           TRAP     #1
           BNE      FAULT               Branch, if error.
           .
           .
           .
PRMBLK:    DC.L     'TSKB'              Target taskname.
           DC.L       0                 Same session number.
           DC.L     STBUF               Pointer to new state information.
STBUF:     DS.B      70                 Buffer contains D0-D7, A0-A7.
MASK:      DS.L       1                 Exception monitor mask.
```

DETACH EXCEPTION MONITOR DEXMON

Directive Number: 65

Parameter: Parameter Block Address

Parameter Block:

Target Task (8 bytes)	Task_id of target task being detached from exception monitor.
Exception Monitor Taskname (4 bytes)	N/A
Exception Monitor Session (4 bytes)	N/A

Detailed Description:

RMS68K detaches the target task from its exception monitor and the target task then resumes normal activity according to its current state. A detach message is queued to the ASQ of the exception monitor task.

Return Parameters: None

Error codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.
10/$0A	Target task is not attached to an exception monitor.

EXAMPLE:

TSKA wants to detach itself from its exception monitor.

```
    TSKA:          .
                   .
                   .
          MOVE.L   #65,D0        Load DEXMON directive number 65.
          LEA      PRMBLK,A0     Load parameter block address.
          TRAP     #1
          BNE      FAULT         Branch, if error.
                   .
                   .
                   .
    PRMBLK:  DC.L  0,0,0,0       Detach requesting task from monitor.
```

CHAPTER 9

EXCEPTION MANAGER

9.1 OVERVIEW

Exceptions are those conditions that cause the M68000 Family processor to switch from executing in the user mode to the supervisor mode. When an exception occurs the current status register, program counter, and the optional vector offset register are pushed on the supervisor stack and an exception vector number is generated, either internally by the processor or externally by the interrupting device. This vector number is within the range $0 to $FF and is multiplied by four to index into the longword vector table to access the address of the routine that handles that exception.

These exceptions may be broken down into two classes:

a. Interrupts that are caused by a request for service from some external device.

b. Program exceptions resulting from the program causing the processor to enter a state that requires supervisor mode processing.

Program exceptions are further broken down into two sub-classes:

a. Error exceptions such as bus errors or divide by zero faults that indicate error conditions within the program.

b. Trap instructions indicating that the user program is requesting service from some supervisor mode routine.

The Exception Manager provides services to support all these exception conditions via the directives:

CISR	A task configures a portion of its code to act as an Interrupt Service Routine (ISR) in response to a particular exception.
SINT	A task simulates an exception (interrupt).
RTE	A task returns from an exception. (Do not confuse the RMS68K RTE direcitve with the M68000 RTE instruction.)
EXPVCT	A task announces the handling of its own exceptions.
TRPVCT	A task announces the handling of its own trap instructions.

CDIR	A task configures a portion of its code to act as an extension of RMS68K's set of directives.
SNPTRC	The contents of the System Trace Table are copied into a buffer in the calling task's address space.

9.2 THEORY OF OPERATION

9.2.1 Interrupt Handling

RMS68K provides two modes for handling interrupts:

a. Task-level ISRs

   These ISRs run in the user mode of the processor, are relatively easy to use, but are slower to respond because of the requirement to save the previous state of the processor and load the MMU before dispatching the ISR.

b. Driver-level ISRs

   This level of ISR runs in the supervisor mode, respond faster because only a portion of the previous state is saved, but require more discipline to write because they run in supervisor mode with access to all memory.

The following pages provide a detailed description of the task-level ISRs. Driver-level ISRs are described in the manual Guide to Writing Device Drivers for VERSAdos.

9.2.1.1 Task-Level ISRs. A task can configure one or more routines activated as the result of an external interrupt (or exception), and executed in the M68000 Family microprocessor user hardware state at the interrupt priority level via the CISR directive. This routine is called an ISR and is useful when creating task-level I/O device drivers.

A task that wants to do an I/O function can do so by acquiring a segment that includes the memory mapped I/O space for a particular device. An ISR within the task can clear interrupts, read-to or write-from device I/O registers, and do other such activities. On exit from the ISR, the task can be activated to process data. This mechanism allows quick response to an interrupt in the user mode and allows the time-consuming function of data manipulation to be executed when time is available.

When an ISR is active, all task-level activities in the entire system are disabled, but an ISR can be interrupted in favor of an ISR with a higher priority level. Therefore, it is important that an ISR uses a minimum amount of execution time to avoid system performance degradation and lost interrupts.

When an external interrupt occurs, RMS68K saves the pre-interrupt state of the processor and invokes the ISR at the appropriate processor priority level. When control is passed to the ISR, register A0 contains the vector number in the low order 16 bits, and register A1 is set equal to the value of the argument provided in the CISR directive parameter block. The contents of all other registers are completely unreliable. When a normal exit is made from the ISR, no state information is saved; RMS68K returns the processor to the pre-interrupt state.

RMS68K distinguishes between the task-level execution and ISR execution of a task. During ISR execution, only one executive directive is allowed; the RTE directive. This directive is not available for use during task-level execution.

The RTE directive can activate the task containing the ISR by issuing it a WAKEUP signal or queueing it an event.

9.2.1.2 Simulating Interrupts. In system development it is often necessary to design the hardware and software components simultaneously. Most of the software can be checked on a development system by inserting stubs for routines that directly access the hardware. Once the high-level functionality is checked out in this way, a primitive hardware simulator may be built to verify some of the lower level functions.

The Simulate Interrupt (SINT) directive allows a task to emulate the behavior of an external device interrupting the processor at a specific interrupt level and vector. Thus, a task that delays for a time and then executes a SINT directive can be an effective component within a hardware simulator.

9.2.2 Exception Handling

The TRPVCT and EXPVCT directives allow a task to announce to the exception manager that it wants to claim trap exceptions or error exceptions. If an exception occurs that has been previously claimed by TRPVCT or EXPVCT, the exception manager will push the status register and program counter on the task's stack and dispatch the task to the address specified in the TRPVCT or EXPVCT parameter block. Then the task's exception handling routine can determine what action to take and return to the point at which the exception occurred by executing the RTR instruction.

For bus and address error exceptions, EXPVCT pushes 8 additional bytes of descriptive information on the user's stack after pushing the status register and program counter. This information includes the attempted access address that caused the fault, the instruction register at the time of the fault, and one word describing the faulted bus cycle. This information is in the format of the MC68000 long bus error stack frame, regardless of whether the processor is an M68000, M68010, or M68020, and is described in detail in the M68000 16/32-Bit Microprocessor Programmer's Reference Manual.

The stack frames pushed by the exception manager when dispatching a task to its internal exception handler are described in Figures 9-1 and 9-2.

```
                   ========================================
USP ------------>   SR  at moment of exception                 Loc.
                   PC  at moment of exception                 Loc. +2
                   ========================================
```

FIGURE 9-1. User Stack on Entering Exception Handler for all Exceptions except Bus or Address Errors

```
                   ======================================
USP ------------>   Internal information                        Loc.
                   Fault address                               Loc. +2
                   Instruction register                        Loc. +6
                   SR                                          Loc. +8
                   PC                                          Loc. +10
                   ======================================
```

FIGURE 9-2. User Stack on Entering a Bus Error or Address Error Exception Handler

Claiming exception conditions is sometimes necessary when implementing high level language constructs such as PL/1 "on" conditions, where the program describes what action to task when certain programming conditions occur such as divide by zero faults.

The TRPVCT and EXPVCT can work with exception monitors. Where both the exception monitor and the task want to be notified of an exception, and the exception occurs, the task is placed in the WAIT FOR COMMAND state and the exception monitor is notified first. If the exception monitor decides to allow the task to keep running and executes the REXMON directive, the task is dispatched to its exception handling routine.

9.2.3 Defaults

The default condition for interrupts that were not claimed by any task or driver is to return from the interrupt as if nothing happened. For TRAP instructions that were not claimed by the task itself or by any server, the default condition returns an error code. The default condition for error exceptions not claimed by the task itself or by an exception monitor is to abort the task with an abort code describing the error exception. This error format is:

$8010	Bus Error
$8011	Address Error
$8012	Illegal instruction
$8013	Zero divide
$8014	CHK instruction
$8015	TRAPV instruction
$8016	Privilege violation
$8017	Unimplemented instruction ($AXXX)
$8018	Unimplemented instruction ($FXXX)

The exception manager contains a stub for the ACFAIL exception so the user can insert the code to implement their policy for handling this catastrophic condition. When a SYSFAIL exception occurs, the exception manager notifies all device drivers so they can poll their devices to see if any device is asserting SYSFAIL.

9.2.4 Extending RMS68K

There are two techniques for adding extensions to RMS68K.

a. Writing a directive and linking it into the Executive as described in Chapter 10.

b. Using the CONFIGURE (CDIR) directive.

CDIR allows a task to configure a portion of its code as an RMS68K directive to run in supervisor mode as the result of some task loading its directive number into register D0 and executing a TRAP #1 instruction. When the user directive is called, it possesses a pointer to the client task's TCB in A6 and a copy of the task's A0 register in A0. Because user directives run in supervisor mode with all logical addresses equal to their corresponding physical addresses, they should be written in a position-independent way.

Also, if the client task's A0 is a pointer to a parameter block, the directive must first convert it into a physical address by calling the TRAP #0 routine, LOGPHY, as described in the manual Guide to Writing Device Drivers for VERSAdos.

When the directive processing is complete, it can return to RMS68K by executing an RTE instruction.

9.2.5 System Trace Facility

RMS68K provides a facility for tracing some system-level events such as:

a. Dispatching a task.
b. Interrupt occurred.
c. Exception occurred.

These events are enabled by a Trace Flag (TRCFLG) set-up at intialization time. They are traced on a system-wide basis; any task that is dispatched or causes an exception is traced if the appropriate bit is enabled within the TRCFLG. All enabled events are recorded in a circular queue called the TRC. This table can be read either via the monitor debugger or by a task executing the SNPTRC directive. SNPTRC causes a copy of the TRC to be moved into the task's memory with all pointers adjusted as required.

Note that enabling the system trace facility degrades system performance and should only be used within a debug environment. Also, to improve the performance of real-time tasks, TRAP #1 instructions are not traced for tasks executing within the real-time domain.

9.3 DATA STRUCTURES

Three data structures are supported by the exception manager:

a. I/O Vector Table (IOV)

b. User Directive Table (UDR)

c. System Trace Table (TRC)

9.3.1 I/O Vector Table (IOV)

The IOV contains an entry for each vector claimed by a task using the CISR directive.

The table header is defined as:

IOVHDR	(4 bytes)	Block ID Each IOVAT begins with '!IOV' to allow consistency checking and ease of dump reading.
IOVEND	(4 bytes)	Address of end of table.

Each entry is defined as:

IOVJSR	(6 bytes)	A JSR instruction to the common interrupt routine. When a vector is allocated, it is changed to point to IOVJSR.
IOVVECT	(2 bytes)	Vector number.
IOVTCB	(4 bytes)	TCB address of task that owns this vector.
IOVADR	(4 bytes)	ISR address.
IOVARG	(4 bytes)	Argument to be placed in register A1 when ISR is entered.

9.3.2 User Directive Table (UDR)

An entry is placed in the UDR each time a directive is configured by the CDIR directive.

UDR	(4 bytes)	Block ID.
UDRCNT	(2 bytes)	Number of entries in table.

A UDR entry is defined as:

UDRSESS	(4 bytes)	Originating task's session number.
UDROPT	(1 byte)	Options.
UDREXIT	(1 byte)	Exit number.
UDRADDR	(4 bytes)	Directive entry address.

9.3.3 System Trace Table (TRC)

Entries are built in the TRC when various events occur. The setting of the SYSGEN parameter TRCFLG determines which events cause an entry to be built.

BIT NUMBER IN TRCFLG	EVENT	TRACE CODE
15	TRAP #1	$FF15
14	I/O Interrupt not claimed by user task	$EE14
13	Timer interrupt	$FF13
12	User TRAP (#2-#15)	$AA12
11	Exception	$AA11
10	Dispatch	$FD10
9	I/O interrupt claimed by user task	$EE09
8	Return from LOADMMU	$DD08
7	Simulated interrupt	$DD07
6	SYSFAIL interrupt	$EE07

The TRC header information is:

TRCPTR	(4 bytes)	Pointer to address where next entry will be stored.
TRCLNG	(4 bytes)	Pointer to the highest address in table.

A TRC entry is defined as:

TRCCODE	(2 bytes)	Trace code described above.
TRCSR	(2 bytes)	Status register.
TRCPC	(4 bytes)	Program counter.
TRCA0	(4 bytes)	Contents of A0 or second program counter if tracing I/O interrupt.
TRCA6	(4 bytes)	Contents of A6.
TRCD0	(4 bytes)	Contents of D0.
TRCTIME	(4 bytes)	Time-of-day (in milliseconds).
TRCTIM2	(2 bytes)	Extension to time-of-day (in microseconds).

9.4 EXCEPTION MANAGER DIRECTIVES

The directive for the exception manager are described in detail on the following pages.

CONFIGURE INTERRUPT SERVICE ROUTINE CISR

Directive Number: 61

Parameter: Parameter Block Address

Parameter Block:

Target Task (8 bytes)	Task_id of target task.	
Directive Options (2 bytes)	Bits 15-13=000	Allocate exception vector to target task's ISR.
	=001	Dissolve an existing ISR vector connection. If this option is specified, only the vector number field must be supplied.
	=010	Switch an exception vector to new ISR.
	Bits 12-0	Reserved
Reserved (1 byte)	Must be zeros.	
Vector Number (1 byte)	The vector number of the exception vector being allocated, deallocated, or switched. Values can be any vector number $00 through $FF not already used by the system. Typically values in the user interrupt range $40 through $FF are used.	
ISR Address (4 bytes)	Logical address of target ISR.	
Argument (4 bytes)	A user-defined value that is loaded into address register one (A1) when an interrupt occurs.	

Detailed Description:

A task can allocate only those exception vectors not currently assigned to other functions (such as ISRs, server tasks, etc.).

A system task can dissolve any ISR vector connection. A user task can only dissolve the connection for itself and other tasks within its own session.

If the switch option is specified (option bits 15-13 = 010), the specified exception vector is connected to the specified ISR address within the specified target task. A system task can switch an exception vector from any task to any other task. A user task can switch an exception vector from itself or any task within its session to itself or any task within its session.

NOTE

> Any program exception occurring during ISR processing terminates ISR execution with an event describing the exception queued to the task. Breakpoints set by an exception processing task cause an exception interrupt that terminates ISR execution. On entry to the ISR, all registers, except A1, are in an undefined state (any stack, etc. assumed by the user must be set up within the ISR).

WARNING

> ISR EXCEPTION EVENTS (CODE = $02) ARE 10 BYTES LONG. IF A TASK HAS AN ASQ AND CONFIGURES AN ISR, THEN THE ASQ'S MAXIMUM MESSAGE LENGTH MUST BE AT LEAST 10 BYTES.

RTE is the only Executive directive that can be issued during ISR execution, and is unavailable for use in task-level execution.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
3/$03	Target task does not exist.
5/$05	No space in the IOV.
6/$06	Requested vector not available.
7/$07	Vector does not belong to caller.
12/$0C	ISR address not in user's address space.
15/$0F	Invalid options.

EXAMPLE:

A non real-time user task, TSKA, wants to redirect number 232 exception vector connection from itself to an ISR at location ISRADR within TSKB. The address of BEGDATA will be moved to address register one when an interrupt occurs.

```
TSKA:           .
                .
                .
        MOVE.L  #61,D0          Load CISR directive number 61.
        LEA     PRMBLK,A0       Load parameter block address.
        TRAP    #1
        BNE     FAULT           Branch, if error.
                .
                .
PRMBLK: DC.L    'TSKB'          Target taskname.
        DC.L    0               N/A; user task.
        DC.W    $4000           Switch one exception vector to new ISR.
        DC.B    0               Reserved
        DC.B    $E8             Vector 232/$E8.
        DC.L    ISRADR          New ISR address.
        DC.L    BEGDATA         Data to load into A1 on interrupt.
```

SIMULATE INTERRUPT SINT

Directive Number: 62

Parameter: Parameter Block Address

Parameter Block:

Not Used (2 bytes)	
Interrupt Priority (1 byte)	Hardware priority level of the exception to be generated.
Vector Number (1 byte)	The exception vector number of the exception to be generated. Value may be in the range $00 to $FF, inclusive.

Detailed Description:

The vector number specified must be currently connected to a task-level ISR. RMS68K activates the ISR as if the actual exception (interrupt) had occurred at the specified level.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
9/$09	Requesting task does not have permission to request this function.
14/$0E	Function is not enabled.

EXAMPLE:

TSKA wants to generate an interrupt internally with exception number 226 at hardware priority level 5.

```
    TSKA:          .
                   .
                   .
              MOVE.L    #62,D0          Load SINT directive number 62.
              LEA       PRMBLK,A0       Load parameter block address.
              TRAP      #1
              BNE       FAULT           Branch, if error.
                   .
                   .
                   .
    PRMBLK:   DC.W      0               Not used.
              DC.B      5               Hardware interrupt priority.
              DC.B      $E2             Exception vector number 226 ($E2).
```

RETURN FROM INTERRUPT SERVICE — RTE

Directive Number: None

Parameters:	Register D0=0	Simple return from ISR.
	=1	Return from ISR and issue WAKEUP to task.
	=2	Return from ISR and queue event to task's ASQ.
	Register D1	Needed only if D0=2. If nonzero, it specifies an alternate ASR service address. If zero, the default ASR service address assumed.
	Register D2	Needed only if D0=2. The 4 bytes of this register include the text of the event message queued to the task's ASQ.

Detailed Description:

This is the only Executive directive that can be issued during ISR execution, and is unavailable for use in task-level execution.

If D0=0, the processor is simply returned to the pre-interrupt state.

If D0=1, RMS68K moves the task that contains the ISR from the WAIT state to the READY state and returns the processor to the pre-interrupt state.

If D0=2, RMS68K creates an event with event code = $02, using the contents of register D2 as the message text. It then queues the event into the ASQ of the task containing the ISR. If an alternate ASR service vector is in register D1, this is queued along with the event. The processor is then returned to the pre-interrupt state.

Return Parameters: None

Error Codes: If an exception occurs in the ISR, an error event is sent to the ASQ of the task containing the ISR; a WAKEUP is also issued with error code in D0.

EXAMPLE:

An ISR has completed execution and wants to issue a WAKEUP to the task in which it is contained:

```
    ISRADR:     .
                .
                .
           MOVE.L   #1,D0          Issue WAKEUP to task on return.
           TRAP     #1             Return from interrupt service.
```

ANNOUNCE EXCEPTION VECTORS EXPVCT

Directive Number:	26
Parameter:	Exception Vector Table Address

Exception Vector Table:

The exception vector table consists of nine 4-byte entries, each of which is the transfer address for the appropriate exception. The applicable exceptions, in order, are:

Bus Error
Address Error
Illegal Instruction
Zero Divide
CHK Instruction
TRAPV Instruction
Privilege Violation
Line 1010 Emulator
Line 1111 Emulator

Detailed Description:

The user must have previously allocated a stack area for EXPVCT to work. RMS68K uses the exception vectors in the above table to handle exceptions that occur during execution of the issuing task. A value of zero in any table entry results in default processing of the exception. Any other value causes the following to take place when an exception is encountered:

a. The proper information is pushed on the stack, according to the MC68000 Family microprocessor exception processing sequence.

b. The task begins executing at the address specified by the vector table entry.

Bus and address errors cause 14 bytes to be pushed on the stack. Other exceptions cause 6 bytes to be pushed on the stack.

After the EXPVCT directive has been executed, the requestor can dynamically alter exception processing by swapping values in specific table entries, without re-issuing an EXPVCT directive.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Exception vector table not in requestor's address space.

EXAMPLE:

TSKA wants to handle its own zero divide exception at location ZERDVD. All other exceptions are handled by default processing.

```
TSKA:         .
              .
              .
        MOVE.L   #26,D0          Load EXPVCT directive number 26.
        LEA      VECTBL,A0       Load parameter block address.
        TRAP     #1
        BNE      FAULT           Branch, if error.
              .
              .
              .
ZERDVD:       .
              .                  Handle zero divide exception.
              .
        RTR                      Return from exception.
              .
VECTBL: DC.L      0,0,0          Exception address for ZERDVD.
        DC.L      ZERDVD
        DC.L      0,0,0,0,0
```

ANNOUNCE TRAP VECTORS TRPVCT

Directive Number: 27

Parameter: Trap Vector Table Address

Trap Vector Table:

The trap vector table consists of fourteen 4-byte entries, each of which is the transfer address for the appropriate trap instruction. The table covers TRAP #2 through TRAP #15.

Detailed Description:

The user must have previously allocated a stack area for TRPVCT to work. A task can indicate to RMS68K that it will handle its own TRAP instructions, which takes precedence over a server task responding to the TRAP instructions, (i.e., if a server task is established to service TRAP #9 instructions, and a task claims TRAP #9 instructions via TRPVCT, the task will be notified and the server task will not.)

RMS68K uses the trap vectors in the above table to handle trap instructions that occur during the execution of the issuing task. A value of zero in any table entry results in default processing of the corresponding trap instruction. Any other value causes the following to take place when a trap instruction is encountered:

a. SR and PC pushed onto user stack (6 bytes).

b. Task begins executing at location specified by vector table entry.

After the TRPVCT directive has been executed, the requestor can dynamically alter trap instruction processing by swapping values in the trap vector table, without re-issuing a TRPVCT directive.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Trap vector table not in requestor's address space.

EXAMPLE:

TSKA wants to handle its own TRAP #4 instruction. All other TRAP # instructions are to be handled by default processing.

```
TSKA:     .
          .
          .
        MOVE.L  #27,D0                  Load TRPVCT directive number 27.
        LEA     VCTTBL,A0               Load parameter block address.
        TRAP    #1
        BNE     FAULT                   Branch, if error.
          .
TRP4:     .                             Service TRAP #4 instruction.
          .
        RTR                             Return from TRAP exception.
          .
VCTTBL: DC.L    0,0                     Default serve of TRAPS #2, #3.
        DC.L    TRP4                    Address to serve own TRAP #4s.
        DC.L    0,0,0,0,0,0,0,0,0,0,0   Default serve of TRAPS #5 to #15.
```

CONFIGURE DIRECTIVE

CDIR

Directive Number: 58

Parameter: Directive Block Address

Directive Block:

Directive Number (2 bytes)	This is a negative number. Its range is -1 to -n, with n determined by a SYSGEN parameter.	
Options (2 bytes)	Bits 7,6	Reserved
	Bit 5=0	Enable the directive number specified.
	=1	Disable the directive number specified.
	Bit 4=0	Directive can only be called by tasks within this session.
	=1	Directive can be called by any task in system (caller must be a system task).
	Bits 3,2	Reserved
	Bit 1-0=00	When directive exits, return control to requesting task.
	=01	When directive exits, go to dispatcher. Do not put requesting task on READY list.
	=10	When directive exits, put requesting task on READY list and then go to dispatcher.
	=11	Reserved
Directive Routine Entry Address (4 bytes)	Start address (within calling task's address space) of directive routine.	

Detailed Description:

The CDIR directive provides a way to create new Executive directives for use by specific applications.

Executive directive routines are executed in supervisor mode with no memory mapping or protection provided. It is the responsibility of the user of this directive to ensure that the integrity of the system stack pointer is preserved, that the contents of memory are not inadvertently destroyed, that interrupts are not masked for excessive periods of time, and that control is returned to the Executive through the normal TRAP #1 exit processing.

Executive directive routines are not tasks and therefore may not request services from server tasks or from the Executive via the TRAP #1 interface. This could cause the system to crash. These directive routines may request services from the Executive via the TRAP #0 driver interface as described in the Guide to Writing Device Drivers for VERSAdos manual.

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
2/$02	Parameter block not in requestor's address space.
4/$04	User Directive Table (UDR) does not exist.
5/$05	Directive number is out of range allowed in the user directive table.
6/$06	Duplicate user directive number.
9/$09	User task cannot disable a directive created by another session.
12/$0C	Address of directive routine is not within caller's address space.

When a TRAP #1 directive routine is entered:

A7 = System Stack Pointer

The status register contents and program counter needed to return to the TRAP #1 exit routine are already loaded on the stack and should not be changed. Data can be saved on the stack during the processing of the directive, but must be removed before the exit.

A6 = Pointer to requesting task's TCB

This register must not be changed by the directive routine. The state of the requesting task has been saved in the TCB.

A0 = Requesting task's A0

The contents of all other registers is unpredictable.

Returning error codes:

By convention, error information is returned to the requesting task as a code number in D0. A directive routine provides a code to be returned in D0 by moving the code to the 2-byte word at TCBRTCD (A6). The requesting task's condition code on return is set according to the content of TCBRTCD (A6) when the directive exits. (The equate for TCBRTCD is in the file 9995.&.TCB.EQ.)

Return to the TRAP #1 exit processing is achieved by executing a M68000 RTE instruction (not an RMS68K RTE directive).

EXAMPLE:

A system task, TSKA, configures and executes a directive GTBUF. GTBUF is accessible to any task within the system and returns directly to the requesting task (subroutine return).

```
TSKA:     .
          .
          .
          MOVE.L   #58,D0          Load CDIR directive number 58.
          LEA      PRMBLK,A0       Load parameter block address.
          TRAP     #1
          BNE      FAULT           Branch, if error.
          .
          .
          .
          MOVE.L   #-1,D0          Load GTBUF directive number -1.
          TRAP     #1
          BNE      FAULT           Branch, if error.
          .                        A0 now contains pointer to buffer.
          .
          .
PRMBLK:   DC.W     -1              GTBUF directive number is -1.
          DC.W     $10             Enable directive. Directive available
                                   to any task within system. Return
                                   control to task.
          DC.L     GTBUF           New directive address.

GTBUF:    .
          .
          .
          MOVE.L   BUFPTR,TCBA0(A6)     Return a pointer to requesting
                                        task.
          RTE                           Return from directive processing.
```

SNAPSHOT OF SYSTEM TRACE SNPTRC

Directive Number: 8

Parameter: Address of buffer where the TRC is copied.

Detailed Description:

The contents of the TRC are copied into the buffer provided within the address space of the requesting task.

The pointers within the trace table, to the next free entry and to the end of the table, are adjusted to point to equivalent addresses within the requestor's own buffer.

Return Parameters: None

Error Codes (returned in bits 15-0 of D0):

0/$00	Successful.
12/$0C	Buffer not in requestor's address space or buffer not large enough to contain trace table.

EXAMPLE:

TSKA wants to move the contents of the TRC into a buffer labeled TRACBUF.

```
     TSKA:      .
                .
                .
                MOVE.L   #8,D0          Load SNPTRC directive number 8.
                LEA      TRACBUF,A0     Load buffer block address.
                TRAP     #1
                BNE      FAULT          Branch, if error.
                .
                .
                .
     TRACBUF:   DS.L     34             Buffer to hold trace table information.
```

CHAPTER 10

BUILDING A SYSTEM

10.1 INTRODUCTION

The software development of a real-time application system is broken into four phases:

a. Analysis	Used to determine the potential value and need of a system, the system's functional requirements, and the impact to the hardware environment.
b. Design	Defines the data, establishes interfaces, design functional components, and write the code for user-provided modules.
c. Implementation	Modifiable system parameters are specified, source modules are assembled or compiled, load modules are created, and the system is configured.
d. Test and debug	Testing the performance and reliability of the entire system, including individual modules, and groups of modules.

These four phases are not necessarily four distinct phases carried out in a particular sequence; instead they overlap each other. For example, the test and debug phase usually begins during the implementation phase. Also, it cannot be assumed that system functional requirements, hardware design, and interfaces, will remain constant during system development. There are many factors that influence system development throughout all phases.

The following paragraphs describe each of these phases as a stand-alone entity, only because it is difficult to talk about them as an integrated package. These paragraphs bring out general system development concepts and RMS68K system specific development procedures. The general system development guidelines in the following paragraphs may not be appropriate for every application system; the user must tailor the techniques for the needs of a particular application system being developed.

10.2 ANALYZING YOUR SYSTEM

The analysis phase of system development is also the "what to build" phase. Therefore, this phase begins with the realization of the need for a real-time system. Many types of systems are built using real-time operating system concepts, including industrial process control systems, operations control systems, data acquisition systems, management information systems, and development systems.

During the analysis phase, system designers should consider some general questions:

a. What are the basic functional requirements of the system?

b. What type of real-time system can satisfy these requirements?

c. What hardware and software components are needed to satisfy these requirements?

Before any other questions are answered, the basic functional requirements must be defined. During the analysis phase questions arise that do not have an obvious answer; they are answered through an integrated approach requiring decisions and compromise.

10.2.1 Hardware Environment

The user system can be designed and built to operate in a variety of hardware environments. Although this is not a complete list, some of the more common environments are described here. RMS68K assumes RAM exists for the system vectors, addresses $8 through $3FF, inclusive, and that the SYSPAR variables exist within the sort-addressable space ($FF8000 - $7FFF).

Complete Bootstrap Loadable System

The entire user system is initially located in non-resident memory or a peripheral mass storage device in this environment. At system startup time, RMS68K, a system initializer, and resident user tasks are loaded into system RAM. The system initializer executes first and performs such duties as:

a. Memory sizing
b. System parameter and table initialization
c. Task state configuration

When the system initializer completes its execution, the RMS68K dispatcher is invoked and the system is functional. Removing the system initializer from RAM frees the space for future use.

ROM Resident RMS68K System

In this environment, only RMS68K and a simple system initializer are located in resident memory at system startup time. The system initializer executes first, constructing a list of free memory and performing a load operation to load a task into system RAM that could complete the system initialization. This is the first task dispatched by RMS68K; remaining resident user tasks are loaded into system RAM. When the task has completed execution, removing the task from system memory makes the area availàble for future use. The system is then functional.

Complete ROM Resident System

RMS68K, a system initializer, and resident user tasks are all located in system ROM at system startup time in a complete ROM resident system.

10.2.2 Functional Requirements

The functional requirements of a system must be clearly defined before beginning the design process. A few items the user may want to consider when defining the functional requirements are:

Input rate
Types of input
Device responses
System response time
Task priorities
Output characteristics
Reliability

10.2.3 Basic Software Components

The complete user application system consists of four major components:

a. Application tailored RMS68K
b. System initializer
c. Memory-resident user tasks
d. Non-memory-resident user tasks

Tailored RMS68K

The complete RMS68K package provided to the user is extensive and offers a wide variety of tools that the system designer can incorporate into the application system. Because of the extensive nature of the complete RMS68K package, it is unlikely that a given application would require the full set of RMS68K capabilities. The user can reduce the complete RMS68K package into a target RMS68K tailored to fit the needs of the application system. The system is tailored by choosing the set of Executive directives required to provide the desired functions, and building the reduced RMS68K to contain that executive directive subset. Appendix C discusses the tailoring method.

System Initializer

The system initializer is normally executed first, following a system startup. Its main functions are to create and initialize system control data, and then give control to the task dispatcher of RMS68K so that system operation can begin. Typical duties of the system initializer include:

a. Initialize exception vectors.
b. Build a list of free memory.
c. Initialize system parameters and tables used by RMS68K.
d. Configure resources into given states.
e. Make resident tasks known to RMS68K.
f. Place resident tasks to be executed on the READY list.

User Tasks

Some user tasks are executed frequently throughout the operation of the application system and must always reside in system memory. They are known to RMS68K from system initialization time and are called memory resident user tasks.

Other tasks are only needed occasionally during the operation of the application system. These tasks do not reside in system memory, but are stored in nonresident memory or peripheral mass storage devices. The task is created and loaded into system RAM memory when it is needed and removed from the system RAM memory area when it has completed its execution. Therefore, a nonmemory resident task is known to RMS68K only when it is in the system RAM memory.

10.3 DESIGN CONSIDERATIONS

The components needed to satisfy the functional requirements are defined during the design or "how to build" phase of system development. The following paragraphs give background information that can help determine the best approach to a particular design problem. Sometimes, it is possible to research existing systems that are similar to the one being designed and can often provide valuable insight into the best approach for a new system, as well as point out problem areas to avoid during system development.

10.3.1 Defining Tasks

Defining the tasks that make up an application system is made easier by using a top down structured method when defining the system functions. Modules are grouped together to form tasks according to functional binding after determining the detailed level of functional modularity. This binding is reflected by the amount of data passed between tasks. An attempt should be made to minimize the data or messages passed between tasks; more messages means added overhead for the management of those messages, and often indicate poor functional partitioning of the system.

The designer should realize, however, that there is a danger in carrying the ideas of modularity or functional binding too far. If many small tasks are created to obtain modularity, the overhead used by RMS68K in managing all the small tasks can damage system performance. If a few large tasks are created so that little data needs to be passed between tasks, it reduces the overhead used by RMS68K in task and data management but destroys the multitasking concept used by RMS68K.

It is also helpful to consider tasks in relation to the design organization. If one or a few tasks are assigned to individual designers, tasks could exist as truly independent modules. Each designer only has to be aware of the function and interfaces of other tasks; this helps prevent too much interaction between tasks.

One other concern in defining task boundaries is the ease of maintenance of the system. If changes and enhancements are kept in mind while defining tasks, the job of implementing future changes and enhancements is minimized by grouping affected areas together.

Relative task priorities should also be considered at the time the tasks are defined. Tasks with higher priorities, that are considered prime-time tasks, should operate quickly and efficiently. Lower priority tasks, or spare-time tasks, can do longer operations without degrading system performance.

10.3.2 Device Drivers

There are two methods for a user to provide device drivers in an RMS68K application system and are described below to help determine the best method to use.

Interrupt Service Routines (ISRs)

The ISR mechanism provided by RMS68K is also used to provide device drivers. Chapter 9 explains the ISR mechanism. This method provides operation in the user hardware mode of the M68000 Family microprocessor.

With this method the user must provide the synchronization needed in dealing with the device. Some overhead still exists, however, because interrupts must still be treated by RMS68K and dispatched to the proper task for processing.

I/O Handlers

This method described in detail in the Guide to Writing Device Drivers for VERSAdos manual, uses the optional Channel Management Routines (CMR) within RMS68K. It is a method of extending the Executive with I/O personality.

The user has complete control over the device and must provide synchronization needed in dealing with the device. This method runs in supervisor mode and is the most efficient method of doing I/O.

10.3.3 Exception Monitor

There are two reasons for the exception monitor mechanism:

a. To provide a debugging capability.

b. To provide the user with the capability of handling alarm situations within the system.

For example, all tasks within a system could run under the control of one exception monitor. The exception monitor is notified any time an error condition occurs and relays the error condition information to a console to request human operator intervention.

10.4 IMPLEMENTING YOUR SYSTEM

Once the user tasks are coded and relocatable object modules are obtained the final system is ready to be created. There are three main steps:

a. Build the tailored RMS68K load module.
b. Build the application load modules.
c. Create file suitable for system bootload operation or ROM creation.

The following paragraphs provide background information for these steps. The step-by-step procedure for creating the application system is in Appendix C.

10.4.1 Building RMS68K Load Module

As mentioned earlier, a subset of all RMS68K functions can be chosen for inclusion in an application system. Keeping this in mind, four steps are necessary to build the RMS68K load module:

a. Select appropriate RMS68K modules that provide the desired function.

b. Change any supplied RMS68K source modules to reflect the modules chosen in step a.

c. Assemble source modules.

d. Link object modules to produce final load modules.

Directive Selection

The user determines the directives based on the implementation requirements brought out in the system design phase or the functions required for a similar system. A simpler way to determine the directives is to test the system using the complete RMS68K discarding those not used.

Group or individual directives can be selected. A group of directives contains all directives dealing with a particular concept, such as memory allocation and management, or exception monitors. Within a group, only particular directives may be required. As an example, the entire server directive group could be discarded if a system is not going to use the concept of server tasks. However, only the DSERVE directive would be discarded if the system is going to use the server task concept but not going to dynamically deallocate server functions. Appendix G contains a summary of directives by function.

Frequently, certain entities (such as semaphores) are created during the system build procedure; the directive that creates the entity dynamically need not be included. Likewise, the directive that deletes the entity is excluded, if an entity does not need to be dynamically deleted during system operation.

The modules that support these directives are chosen once the directives are determined. Sometimes, there is only one module corresponding to one directive. In other cases, if a directive has options, one module represents each version of the directive. In yet other cases, several partial modules are provided for one directive; a particular combination of these partial modules must be selected to provide the directive with the desired options. Details of modules corresponding to directives are given in Appendix A.

Source Module Modification

The directive table, TABLE1, should be changed by the user to reflect the directives chosen for inclusion in the system. The RMS68K package includes a source module (TRAP #1) containing TABLE1. Each entry in the table corresponds to one directive. The entries are numbered from 0 to the highest numbered directive in the system. The entries for all excluded directives should be modified (refer to Appendix C).

Assembling and Linking

Chain files are supplied in the RMS68K package that assemble the modified source modules and link the selected object modules. The user modifies the chain files to include only the required modules in the assembly or link process.

10.4.2 Building the Application Load Modules

The application load modules contain code consisting of memory resident tasks, system initializer, and data areas. There are two steps for building the application load module:

a. Produce relocatable or absolute modules for all resident user supplied tasks and data.

b. Produce a relocatable or absolute module for the system initializer.

User-Supplied Tasks and Data

Each system task, user task, and associated data that is to be ROM resident or bootloaded is assembled and linked to form an independent load module. They are identified later to RMS68K.

System Initializer

The RMS68K package includes a system initializer in source form. The system initializer is used as is, modified, or completely rewritten, as the user deems necessary. It is assembled using the chain file provided with the RMS68K package.

10.4.3 Create Bootload or ROM File

The final step in creating an application system is to make a file suitable for bootload or ROM creation out of the RMS68K load module and the application load module that has been built. This is done by using the SYSGEN utility. Appendix C contains the steps for building this bootload or ROM file.

10.5 TESTING AND DEBUGGING YOUR SYSTEM

The RMS68K-based application system can be tested and debugged on the MC68000-Family microprocessor-based system, using the appropriate firmware debug monitor or the SYMbug symbolic debugging utility. Refer to the corresponding debug reference manual.

APPENDIX A

LIST OF PARTS SUPPLIED IN RMS68K PACKAGE

A.1 INTRODUCTION

RMS68K is distributed in object-only form as well as source/object. The object-only product contains all relocatable object modules necessary to generate a real-time multitasking Executive for all of Motorola's supported M68000 Family of processors. There are specific system generation chain files that allow the user to generate an RMS68K Executive for each of the following systems or single board microcomputers.

EXORmacs
VME/10
VME/12
VM01
VM02
VM03
VM04
VMC 68/2
VME101
VME110
VME115
VME120
VME121
VME122
VME123
VME128

A.2 CONTENTS OF VOLUME REAL-TIME MULTITASKING

The RMS68K package is made up of a number of modules that support various processor and module configurations. To easily identify which configuration a particular module supports, some naming conventions have been applied to the various files that make up the RMS68K package. A catalog name has been added to all RMS68K modules. These are broken into four classes; processor type, system type, MMU type, and timer chip type. The following table describes the catalog names.

<u>Processor Type</u>

M68XXX	These modules may be used on all systems regardless of processor type.
M68000	These modules may be used only on MC68000 based configurations.
M68010	These modules may be used on MC68010 based configurations. Some of these modules are also used on MC68020 configurations.
M68020	These modules may be used only on MC68020 based configurations.

System Type

EXORmacs	These modules apply to a specific system configuration of the EXORmacs.
VM04	These modules apply to a specific system configuration of the VM04.
VME101	These modules apply to a specific system configuration of the MVME101.
VME110	These modules apply to a specific system configuration of the MVME110.
VME120	These modules apply to a specific system configuration of the MVME120/MVME121.
VME122	These modules apply to a specific system configuration of the MVME122/MVME123.
VME128	These modules apply to a specific system configuration of the MVME128.

MMU and CACHE Type

M68451	These modules apply to a system using the MC68451 MMU.
NOMMU	These modules apply to a system using no MMU.
NOCACHE	These modules apply to a system using no CACHE.
NOMMUC	These modules apply to a system using no MMU but a CACHE (MVME122).

Timer Chip

M6840	These modules apply to a system using the MC6840 timer chip.
M146818	These modules apply to a system using the MC146818 timer chip.
M68901	These modules apply to a system using the MC68901 timer chip.
M68230	These modules apply to a system using the MC68230 timer chip.
Z8036	These modules apply to a system using the Z8036 timer chip.

Some examples of modules with their catalog names are:

EXORMACS.LOADMMU.RO	Module that manipulates the MMU of the EXORMACS.MPU board.
M68XXX.DELAY.RO	Module that handles time delay calls for all processor types.
M68010.DISPATCH.RO	Module that handles task dispatching for MC68010 based systems/modules.
M146818.RDTIMER.RO	Module specific to the MC146818 real-time clock chip.

A.2.1 RMS68K Directive Modules

A list of relocatable object modules followed by the name of the directive(s) contained in the module follows. For example, module ASQALOC.RO supports the directive GTASQ. These module names also contain an appropriate catalog describing the chip type support (refer to Appendix A.1 for description of catalog naming conventions used).

AKRQST.RO	AKRQST
ASQALOC.RO	GTASQ
ASQEVENT.RO	QEVNT
ASQFREE.RO	DEASQ
ASQGET.RO	GTEVNT
ASQREAD.RO	RDEVNT
ASQSTATS.RO	SETASQ
ATSEM.RO	ATSEM and CRSEM
CACHE.RO	FLUSHC
CDIR.RO	CDIR
CISR.RO	CISR
CMR.RO	CMR
DCLSHAR.RO	DCLSHR
DELAY.RO	DELAY
DEMON.RO	DEXMON

DERQST.RO	DERQST
DESEM.RO	DESEM, DESEMA
DSERVE.RO	DSERVE
EXMMSK.RO	EXMMSK
EXMON.RO	EXMON
GTDTIM.RO	GTDTIM
GTTASKID.RO	GTTASKID
GTTNAME.RO	GTTASKNM
PSTATE.RO	PSTATE. Uses EXMONVR.
RCVSA.RO	RCVSA
RELINQ.RO	RELINQ
RESUME.RO	RESUME
REXMON.RO	REXMON
RQSTPA.RO	RQSTPA
RSTATE.RO	RSTATE. Uses EXMONVR.
RTEVENT.RO	RTEVNT
SEGALOC.RO	GTSEG
SEGDEAL.RO	DESEG
SEGSHAR.RO	ATTSEG and SHRSEG
SERVE.RO	SERVER
SETPRI.RO	SETPRI
SGSEM.RO	SGSEM and WTSEM
SINT.RO	SINT
SNAPTRAC.RO	SNPTRC
STDTIM.RO	STDTIM
SUPER.RO	SUPER
SUSPEND.RO	SUSPND

TERM.RO	ABORT, TERM and TERMT. Uses DSEGX, DSRVX, EXQEVENT, and PAUSE.
TFRSEG.RO	TRSEG
TSKATTR.RO	TSKATTR
TSKBORN.RO	CRTCB
TSKINFO.RO	TSKINF
TSKMOVE.RO	MOVELL
TSKSTART.RO	START and STOP
TSKWAIT.RO	WAIT
USERVECT.RO	EXCVCT and TRPVCT
WAKEUP.RO	WAKEUP
WTEVENT.RO	WTEVNT

A.2.2 Real-Time Multitasking Subroutines

The following modules function as subroutines and are called by other real-time multitasking modules.

ASRINT	Provides an event pseudo-interrupt if appropriate.
BKG	Schedules and dispatches any background routines on the background queue.
CKDELAY	Processes satisfied DELAY or periodic activation.
CKEXPAT	Processes satisfied Executive periodic activation.
DSEGX	Deletes segments from terminating tasks.
DSEMX	Detaches terminating task from semaphores.
DSRVX	Frees trap instructions belonging to terminating tasks.
EQDQ	Holds server requests until the server is ready.
EXABRT	Provides real-time multitasking-initiated task aborts.
EXMONVR	Used by some exception monitor directives.
EXQEVENT	Used to queue an real-time multitasking-initiated event to a task.

EXRQPA	Used to request Executive periodic activation.
FNDGSEG	Locates a segment descriptor in the GST.
FNDTSEG	Locates a segment descriptor in a TST.
FNDUSEM	Locates a user semaphore.
GETTCB	Locates a TCB.
KILLER	Provides a controlled system halt.
LOGPHY	Translates a logical address to a physical address and verifies that a logical address range is within a task's address space.
PAGEALOC	Performs physical memory allocation.
PAGEFREE	Frees physical memory.
PAUSE	Waits for I/O to complete during termination of a task.
PVSEM	Performs semaphore signals and WAITS.
RDTIMER	Reads the system timer.
READY	Places a task in the READY list.
TRACER	Inserts an entry into the trace table.

A.2.3 Real-Time Multiasking Special Modules

The following .RO modules provide special functions.

COMINT	Common interrupt handler.
DISPATCH	Sends a task into execution.
EXCEPT	Determines appropriate response to traps and exceptions.
EXIT	Implements specific exit policy from directive processing.
POWRFAIL	Provides stub for user-defined power failure ISR.
RMS	Contains entry point to RMS.
RMSPATCH	Patch area for RMS.
SELFTEST	Provides stub for user-defined self-test routine.
SPURINT	Processes spurious interrupts.
SYSPAR	Contains XDEFs for system parameters.

TIMEINT	Responds to timer interrupts.
TRAP0	Handles TRAP #0 real-time multitasking calls.
TRAP1	Handles directive calls and external interrupts.
VECTTBL	Refer to Appendix C, paragraph C.3 for description.

A.2.4 Equate Source Files: 9995

The following assembler source files are included to allow the user to assemble modified or original RMS68K source modules.

STR.EQ	General purpose equates
TCB.EQ	Task Control Block (TCB)
TST.EQ	Task Segment Table (TST)
SEG.EQ	Used by some segment-oriented modules
ASQ.EQ	Asynchronous Service Queue (ASQ)
GST.EQ	Global Segment Table (GST)
UST.EQ	User Semaphore Table (UST)
SRVR.EQ	Server equates
TIOT.EQ	Trap instruction owner equates
TACK.EQ	Server acknowledge options
IOV.EQ	User interrupt vector equates
ENV.EQ	Chip-oriented equates
TRACE.EQ	Trace table
CCB.EQ	Channel Control Block (CCB)
PANEL.EQ	EXORmacs front panel
MAP.EQ	Memory Map Tables (MEMMAP)
PAT.EQ	Periodic Activation Table (PAT)
UDR.EQ	User Directive Table (UDR)
BAB.EQ	Equates relating to background activation blocks.
TR1RTCD.EQ	Equates for error return codes from RMS and EXIT macro.

A.2.5 RMS68K Module Source Files

The following assembler source files are included within the object release to allow the user to modify modules for tailoring purposes. Each source file name contains a ‹catalog› that specifies which system or processor module is supported. These files appear in various catalogs under 9999. Many appear in more than one catalog. The link chain file for the target system specifies which version (catalog) of each module is used.

SYSPAR.AG	Two versions of the system parameter modules are supplied.
RMS.SA	RMS68K initial entry point following system startup. RMS.SA also contains entry points for null routines that can be used if functions are deleted from RMS68K.
VECTTBL.AG	Table used to initialize the system exception vectors.
KILLER.SA	Saves registers and halts execution when unrecoverable error occurs.
TRAP1.SA	Contains TRAP #1 handler and TABLE1 that specifies which RMS68K directives can be invoked by tasks through TRAP #1 instructions executed in user mode.
READY.SA	The READY module is called by other supervisor functions to place a task on the READY list. Multiple entry points are provided so that task priority can be modified before placing a task on the READY list.
VECTORS.SA	This module is used in an EXORmacs-based system to allow room for the hardware vectors.
SELFTEST.SA	This module provides an entry point for a self-test routine. The self-test routine must be added if needed.
POWRFAIL.SA	This module provides an entry point for a power fail or sysfail routine. The power fail or sysfail routine must be added if needed.
TIMEINT.SA	Timer interrupt handler.
‹catalog›RDTIMER.SA	Two versions are provided to support MC6840 or MC146818. The RDTIMER module is called by other RMS68K routines when a time-of-day value is required. It may have to be modified for a system with a timer other than the MC6840 or MC146818.

M68451.LOADMMU.SA EXORMACS.LOADMMU.SA EXORMACS.FAKEMMU.SA NOMMU.LOADMMU.SA NOMMUC›LOADMMU.SA	These routines handle the loading of the appropriate MMU when a task is to be dispatched. The M68451 and EXORMACS modules load the MC68451 or EXORMACS MMU from the TST. The NOMMU module is for systems without an MMU and does not load any registers. NOMMUC is for systems with CACHE but without an MMU. EXORMACS. FAKEMMU.SA is provided so that a system that will ultimately run with no MMU can be tested on an EXORmacs. It loads the MMU with the entire address range.

A.2.6 RMS68K Assembly/Instruction Files

Chain and instruction files are provided to assemble those files identified as RMS68K source files. A default module name of ‹module name›.LS is assumed on each assembly if no output file/device is specified. The format of all assembly chain file names is ‹module name›.AF, while the instruction filename format is ASMNEWS.XX.

A.2.7 RMS68K Initializer Files

INIT.SA	Assembler source for initializer program code.
‹catalog›.INITIO1.AG	Assembler source for the subroutine INITIO called by INIT. These source files allow specific I/O devices to be initialized during system boot.
INITDAT.AG	Assembler source for initializer data. Includes substitution parameters for the system generation process.

THIS PAGE INTENTIONALLY LEFT BLANK.

APPENDIX B

SYSTEM PARAMETER AREA (SYSPAR)

The SYSPAR module defines the RMS68K system parameter area, which is an area in RAM that functions as the RMS68K work space. It contains configuration parameters and parameters required to manage Executive resources.

SYSPAR is assembled at the physical memory address where the system parameters will reside in a running system. It also contains an equate defining the physical memory address of an area where RMS68K can save registers in the event of a system crash (CRASHSAV). If it is necessary to change these addresses, SYSPAR.AG must be modified and re-assembled using the chain file SYSPAR.AF.

The RMS68K parameters are defined within SYSPAR and initialized during system startup.

MAPBEG	(4 bytes)	Points to the beginning of memory map table.
BKG_FLAG	(1 byte)	Set when driver schedules a background job.
PREEMPT_FLAG	(1 byte)	Set when Executive detects a preempt.
NULL	(2 bytes)	
EXCSTACK	(4 bytes)	Contains address of top of supervisor stack.
RUNNER	(4 bytes)	Points to the Task Control Block (TCB) of the currently executing task.
TCBHD	(4 bytes)	Points to the first TCB in the list of all TCBs. Zero indicates no TCBs exist.
READYHD	(4 bytes)	Points to the first TCB in the list of ready-to-execute tasks. Zero indicates no tasks ready.
CCBHD	(4 bytes)	Points to the first Channel Control Block (CCB) in the list of all CCBs. Zero indicates no CCBs exist.
MMUHERE	(4 bytes)	Contains the memory-mapped address of the Memory Management Unit (MMU). Zero indicates no MMU being used.
GSTBEG	(4 bytes)	Points to the Global Segment Table (GST). Zero indicates no GST exists.
USTBEG	(4 bytes)	Points to the User Semaphore Table (UST). Zero indicates no UST exists.

UDRBEG	(4 bytes)	Points to the User Directive Table (UDR). Zero indicates no UDR exists.
PATBEG	(4 bytes)	Points to the Periodic Activation Table (PAT). Zero indicates that no PAT exists.
TRACEBEG	(4 bytes)	Points to the system trace table.
TRACFLAG	(2 bytes)	System trace flags.
MACSTRC	(4 bytes)	Points to the resident debug monitor trace routine.
PANEL	(4 bytes)	Contains the memory-mapped address of the EXORmacs front panel registers. Contains a dummy address if no front panel exists.

<u>TIMER PARAMETERS</u>

DATE	(4 bytes)	Current date.
PTMADDR	(4 bytes)	Contains the memory-mapped address of the timer device.
TIMEOUT	(2 bytes)	Counter of the number of timer interrupts since the last dispatch.
TIMESLIC	(2 bytes)	Number of timer interrupts allowed in a timeslice.
NSE	(4 bytes)	Absolute time in milliseconds of next significant event (when next periodic activation node is due to be scheduled).
TIME_LEFT	(4 bytes)	Amount of time in milliseconds until next periodic activation node is due to be scheduled.
MIDNIGHT	(4 bytes)	Absolute time in milliseconds of previous midnight.
TIMINTV	(2 bytes)	Number of milliseconds between timer interrupts.
TIMINTV4	(2 bytes)	Value used to set the timer to the time interval specified.
TIMINTR	(4 bytes)	Holds microsecond remainder for odd clock rates.
TINTFLAG	(1 byte)	Set to indicate a timer interrupt has occurred.
TMSGFLAG	(1 byte)	Not used.

<u>SPURIOUS INTERRUPT VARIABLES</u>

SPURCNT	(2 bytes)	Count of the number of spurious interrupts that have occurred. Any nonzero value is an indication of a possible hardware problem.

SPURTIME	(4 bytes)	The time_of_day of the first spurious interrupt occurrence.
MMULOAD	(4 bytes)	Points to the task segment table from which the MMU was last loaded.
VCTUBGN	(4 bytes)	Points to the start of the vector use table.
IOVCTBGN	(4 bytes)	Points to the start of the vector assignment table.

DEFAULT PARTITION NUMBERS AND TYPES

ADEFTYP	(1 byte)	Memory type and/or partition number used when allocating space for an ASQ.
TDEFTYP	(1 byte)	Memory type and/or partition number used when allocating space for a TCB.
SDEFTYP	(2 bytes)	Default memory type and/or partition number used when allocating space for a system task.
UDEFTYP	(2 bytes)	Default memory type and/or partition number used when allocating space for a user task.
SLFTSTA7	(4 bytes)	A7 saved if self-test called.

EXECUTIVE SEMAPHORES

SEMTCB	(6 bytes)	Semaphore protecting the TCB list.
SEMGST	(6 bytes)	Semaphore protecting the GST.
SEMUST	(6 bytes)	Semaphore protecting the UST.
SEMCCB	(6 bytes)	Semaphore protecting the CCB list.
SEMTIOT	(6 bytes)	Semaphore protecting the Trap Instruction Owner Table (TIOT).

TRAP INSTRUCTION ASSIGNMENT TABLE

TIAT (16 bytes) This table consists of one byte for each of the 16 trap instructions. The contents of each byte are:

$00	trap is unassigned
$01	trap is reserved for RMS68K
$02	trap is assigned to server task

TRAP INSTRUCTION OWNER TABLE

TIOT — This table consists of one 22-byte entry for each of the 16 trap instructions. An entry is defined as:

TIOTTCB	(4 bytes)	Server task TCB address
TIOTSESS	(4 bytes)	Server task sessions number
TIOTSEM	(6 bytes)	Semaphore used to limit access to the server task's ASQ
TIOTADDR	(4 bytes)	Server task's ASR address
TIOTMCNT	(2 bytes)	Count of unacknowledged messages
TIOTSTAT	(1 byte)	Status
	Bit 15=1	Server function enabled
	Bit 14=1	Server wants termination notification
	Bit 13=1	Server wants parameter block moved with message
	Bit 12=1	Message sent to server, ACK pending
	Bit 11=1	DERQST called while ACK pending
TIOTPBSZ	(1 byte)	Parameter block size

BACKGROUND PARAMETERS

BKG_HEAD (4 bytes) Points to the first entry in the background queue. Zero indicates that no background jobs are present.

BKG_TAIL (4 bytes) Points to the last entry in the background queue. Points to BKG_HEAD if the queue is empty.

BKG_ACTIVE (1 byte)This flag is true (nonzero) when the background is running.

CURR_ASN (1 byte) Current address space number within the MC68451 MMU.

POINTERS USED BY SDLC AND NETWORK SERVICES

FREEQHD (4 bytes) Free buffer queue head.

DBUFSZ (2 bytes) Size of data area in buffer.

FQLWM (2 bytes) Free queue low water mark.

FQBCNT	(2 bytes)	Free queue current buffer count.
USERQHD	(4 bytes)	User buffer queue head.
USERQND	(4 bytes)	User buffer queue end.
SDLCPCB	(4 bytes)	Pointer to primary control block.
NNTBEG	(4 bytes)	Pointer to network name table.
NATBEG	(4 bytes)	Pointer to network address table.
LCTBEG	(4 bytes)	Pointer to logical connect table.
NWPSEG	(4 bytes)	Limits of network procedure segment.
NWTSEG	(4 bytes)	Limits of network table segment.
NWDQHD	(4 bytes)	Disconnect (task terminated) queue head.
NWSTATUS	(4 bytes)	Network status (-1 = dead).
V2RQHD	(4 bytes)	Requests for action by VM02 system.
MEMOFF	(4 bytes)	VM02 board memory offset.
SYSPOFF	(4 bytes)	VM02 SYSPAR area offset.

<u>POINTERS USED BY IO DRIVERS, ETC.</u>

CTRLREG	(4 bytes)	Pointer to VM02 control register.
DPRVAO	(4 bytes)	Dual-ported RAM VERSAdos address offset.
RAD1TBL	(4 bytes)	Pointer to table used by RAD1 driver.
RIOTBL	(4 bytes)	Pointer to RIO1 driver table.
DCOTBL	(4 bytes)	Pointer to DCO driver table.
ACOTBL	(4 bytes)	Pointer to ACO driver table.
INPTBL	(4 bytes)	Address of interrupt queue control table for the MVME-610/620 driver.
DACTBL	(4 bytes)	MVME605 driver table address.

<u>SDLC/ TS FREE QUEUE END</u>

FREEQND	(4 bytes)	Pointer to end of free queue.

PARAMETERS RELATING TO ADDRESS SPACE

ASNTBL	(4 bytes)	Points to table of task address space numbers.
NOTLAM	(4 bytes)	(Page size - 1) for segment allocation.
LAM	(4 bytes)	(MC68451 logical address mask) *256.
FRST451	(4 bytes)	Address of first MC68451.
LAST451	(4 bytes)	Address of last MC68451.
CURR451	(4 bytes)	Address of MC68451 to next check for swapping.
CURRSD	(4 bytes)	Segment descriptor in CURR451 to next check.

PARAMETERS FOR FLUSHING CACHE

(Applies only to VME120, VME121, VME122, VME123, and VME128 based systems.)

CFLUSH (4 bytes) Address for flushing cache.

```
If       ((CFLUSH) = F_BANK1)
Then     (flush bank 1 only);
Else If  ((CLFUSH) = F_BANK2)
Then     (flush bank 2 only);
Else If  ((CFLUSH) = F_ALL)
Then     (flush banks 1 and 2);
```

LAST_MMU_INT_LEVEL

(2 bytes) On systems using the MC68451 MMU, this contains the interrupt level of the last bus error that resulted in the load of a segment descriptor.

APPENDIX C

RMS68K CONFIGURATION

C.1 INTRODUCTION

This appendix details the steps to tailor RMS68K to the user environment.

C.2 ADDING OR DELETING DIRECTIVES

When a TRAP #1 is executed by a task, the Executive responds to the exception by entering the TRAP #1 routine. This routine saves the state of the task that executed the TRAP #1, and then references a table (TABLE1) that contains an entry for each directive.

TABLE1 can be modified and then re-assembled by invoking the chain file TRAP1.AF.

C.2.1 TABLE1 Entry Format

An entry in TABLE1 is created by the SETUP1 Macro. The syntax for SETUP1 is:

```
    SETUP1  <directive_number>,<directive_name>,<pb>,<len(pb)>,<tt>
```

where:

<directive_number>	is the number of the directive or label that has been equated to the directive number. NOTE: TABLE1 must be created in order of increasing directive number with no gaps.
<directive_name>	label of first instruction in directive code.
<pb>	"PB" indicates this directive requires a parameter block; anything else indicates no parameter block is required.
<len(pb)>	number of bytes in parameter block. Only applicable if <pb> = "PB".
<tt>	"TT" in this field indicates this directive may access a target task. Anything else indicates this directive does not access a target task.

C.2.2 Register Use in TRAP #1 Directive Routines

When a TRAP #1 directive processing routine is entered, the following registers are set:

A7	Supervisor stack.
A6	TCB address of calling task. This register must not be changed by the processing routine.
A5	TCB address of target task if the "TT" option was specified in the options field of the TABLE1 entry.
A4	Absolute physical address of the calling task's parameter block if the "PB" option was specified in the options field of the TABLE1 entry.
A0	If the "PB" option was not specified in the TABLE1 entry, A0 contains the value it had when the TRAP #1 was executed by the calling task.

C.2.3 Return from Directive Processing Routine

A set of exit macros are contained in the file 9995.&.TR1RTCD.EQ. This file should be included in any module containing an RMS68K directive.

If the directive did not cause the task to enter a WAIT state, the directive should call the EXIT macro with the SUB argument which causes a subroutine exit from the Executive.

EXAMPLE: EXIT SUB

If the directive caused the task to enter a WAIT state, the directive should call the EXIT MACRO with the POST argument that tells the Executive to postempt the task (i.e., leave it off the READY list and go through a dispatch cycle).

EXAMPLE: EXIT POST

If the directive encountered an error, the directive should call the EXIT MACRO with the appropriate error code as an argument. (The equates for error codes are also contained in 995.&.TR1RTCD.EQ.)

EXAMPLE: EXIT RTCDOPT Exit and signal an invalid option (error code = $0F).

C.3 EXCEPTION VECTORS

The hardware vector addresses are initialized during system startup. The module VECTTBL.AG consists of a table describing how the vectors are initialized.

A specific VECTTBL.AG is supplied for each of the supported configurations identified by ‹catalog›.

Vectors fall into two classes:

a. Assigned to a specific Executive function.
b. Unassigned, point to COMINT, the common interrupt module.

The assigned vectors are aimed at specific portions of the Executive. As examples, the vector number $02 points to the Executive's bus error handler; the vector number $21 points to the TRAP #1 module.

The exception vectors point to the exception pseudo-vectors within the EXCEPT module. Most trap instruction vectors point to the trap pseudo-vectors within EXCEPT. If the EXCEPT module is not included in the target Executive, the vectors should be redirected.

Most external interrupt vectors point to the COMINT module.

VECTTBL header:

```
        DC.L   '!VCT'
```

System startup searches for this table header, so it must be included as the first 4 bytes of the table.

A macro is provided to create table entries:

```
        VECTOR ‹vector number›,‹vector address›
```

If the vector address specified in the table is 0, the address assigned to that vector is a pointer to COMINT. These vectors are initially unassigned and are handled by the common interrupt handler.

If the vector address specified in the table is 1, this vector is skipped during initialization. Normally the address left at this vector location is the address used by the firmware debugger. Typical cases of vectors that might be skipped during initialization are the software abort vector, or the illegal instruction vector.

C.4 RMS68K INITIALIZER

Three modules comprise the RMS68K initializer:

INIT.SA	Contains initialization code common to all systems.
INITIO1.AG	Contains initialization code specific to a particular system.
INITDAT.AG	Contains initialization data.

Ordinarily the user never changes INIT.SA or INITDAT.AG. The user may change INITIO1.AG source; it is reassembled during SYSGEN operation.

C.4.1 SYSGEN Parameters

The supplied initializer contains a data area (INITDAT.AG), that receives SYSGEN parameters. Some typical parameters are described in the following list. The SYSGEN parameter names are slightly different from the names for the corresponding variables in INITDAT.

MEMBEND1	Top of partition zero. (Address after last byte in partition zero.)
MEMEND2	Bottom of partition one. (Address of first byte in partition one.)
MEMEND3	Top of partition one. (Address after last byte in partition one.)
STACK	The address of the desired Executive stack area. For example, to have a stack that consumes $8FF and downward, specify $900.
MMU	If an MMU is to be used, this specifies its address in the memory map. Zero specifies no MMU.
PANEL	The address of the EXORmacs front panel, if appropriate. Zero specifies no panel.
ASN	The number of address spaces, currently 0 or 127 (M68451).
GST	The number of 256-byte pages to be in the GST. Zero specifies no GST. Each page can accommodate about 14 entries. This table is required if any task uses locally or globally shared memory segments.
UST	The number of 256-byte pages to be in the user semaphore table. Zero specifies no UST. Each page can accommodate about 11 entries.
UDR	The number of 256-byte pages to be in the UDR. Zero specifies no UDR. Each page can accommodate about 25 entries.

PAT	The number of 256-byte pages to be in the PAT. Zero specifies no PAT. Each page can accommodate about eight entries. This table is required if any task uses DELAY or RQSTPA directives.
IOV	The number of 256-byte pages to be in the IOV. Zero specifies no IOV. Each page can accommodate about 12 entries.
TRACE	The number of 256-byte pages to be in the trace table. Each page can accommodate about 10 entries. This table is required if any trace flags (TRCFLAG) are set.
TRCFLG	This is a 2-byte field in which each bit describes a type of occurrence that should be traced while the system is running:

Bit 15=1	Set to trace TRAP #1
Bit 14=1	Set to trace interrupts
Bit 13=1	Set to trace timer interrupts
Bit 12=1	Set to trace user TRAP #2-#15
Bit 11=1	Set to trace exceptions
Bit 10=1	Set to trace dispatches
Bit 9=1	Set to trace user claimed interrupts
Bit 8=1	Set to trace return from LOADMMU
Bit 7=1	Set to trace simulated interrupt
Bit 6=1	Set to trace SYSFAIL interrupt

TIMER	The address of the timer device. Zero specifies no timer.
CLOCKFRQ	The clock frequency is the number of ticks that occur in one millisecond.
TIMINTV	The time interval (in milliseconds) between timer interrupts. Normally, the time interval is 10 milliseconds. The maximum value is 64. (This parameter is not used on the VME/10. The actual value used for the VME/10 is 15.625 milliseconds.)
TIMSLIC	The number of timer interrupts allowed before a task is forced to relinquish the processor. In the released system, this variable is set to 2.
STARTRMS	The address of the RMS68K entry point. When finished, the initializer jumps to this location.
WHERLOAD	The address at which RMS68K is actually loaded, if it must be moved at system startup time. If booting from disk on a VERSAmodule 01 system, the boot file must be loaded into offboard memory and then moved to onboard memory. If WHERLOAD = 0, no move takes place.

C.4.2 Memory Map Table

The memory map table, which is part of the initializer data module, must be modified to describe what memory is available and how it is divided into memory partitions.

A macro is provided to build entries in the table MEMTABL:

```
        MTENTRY ROM,<low limit>,<high limit>


        MTENTRY RAM,<low limit>,<high limit>,<memory type>*16,<partition>
        TOP|BOTTOM
```

where:

MTENTRY RAM	indicates whether this entry describes RAM or ROM.
<low limit> <high limit>	describes the address range of this partition.

If the entry describes ROM, no other fields are required.

If the entry describes RAM, memory type and a partition number must be specified.

<memory type>*16	any value from 0-7.
<partition>	any value from 0-15. Must be unique for each partition described. Any number of partitions can have the same type value assigned.
TOP|BOTTOM	describes whether the memory partition header information should be placed at the top or bottom of the partition.

The total available RAM in a system can be divided into partitions so that a given allocation request can be limited to a specific address range.

C.4.3 Memory Allocation Default Values

All memory allocation requests are made for a specific memory type and/or partition number. The default values for the various kinds of allocation are defined in the Initializer data segment.

Each default value is 1 byte containing ‹memory type›*16+‹partition›.

The fields included in the data segment are:

MEMTYPA	(1 byte)	Default type and/or partition number used when allocating ASQs.
MEMTYPT	(1 byte)	Default type and/or partition number used when allocating TCBs.
MEMTYPS	(2 bytes)	The first byte is the default type and/or partition number used when a system task allocates a read-only segment. The second byte is the default type and/or partition number used when a system task allocates a read/write segment.
MEMTYPU	(2 bytes)	The first byte is the default type and/or partition number used when a user task allocates a read-only segment. The second byte is the default type and/or partition number used when a user task allocates a read/write segment.

C.5 RMS68K LOAD MODULE

Command files have been provided for the user to create a fully functional RMS68K load module for each of the support configurations listed in the table below. These command files are used as input to the SYSGEN utility that processes the commands to produce the desired RMS68K load.

EXORMACS.RMS.CD	generates	EXORMACS.RMS.LO
VMES10.RMS.CD	generates	VMES10.RMS.LO
VME101.RMS.CD	generates	VME101.RMS.LO
VME110.RMS.CD	generates	VME110.RMS.LO
VME115.RMS.CD	generates	VME115.RMS.LO
VME120.RMS.CD	generates	VME120.RMS.LO
VME122.RMS.CD	generates	VME122.RMS.LO
VME128.RMS.CD	generates	VME128.RMS.LO
VM01.RMS.CD	generates	VM01.RMS.LO
VM02.RMS.CD	generates	VM02.RMS.LO
VM03.RMS.CD	generates	VM03.RMS.LO
VM04.RMS.CD	generates	VM04.RMS.LO

C.6 PROCEDURE TO BUILD AN RMS68K LOAD MODULE

The following procedure summarizes the steps necessary to generate an RMS68K load module.

1. Log on to VERSAdos as user :9999.

2. Execute the command line:

   ```
   =RMSGEN.CF <arg1>,<arg2>
   ```

   where:

   ```
   <arg1> = mnemonic for system configuration
   <arg2> = output file/device (optional listing)
   ```

   RMSGEN.CF performs the RMS SYSGEN associated with the specified mnemonic.

   The mnemonic <arg1> is one of the following:

   ```
   EXORMACS
   VM01
   VM02
   VM03
   VM04
   VME101
   VME110
   VME115
   VME120
   VME122
   VME128
   VMES10
   ```

3. On completion, this procedure has generated a load module with the appropriate catalog, along with a list (LL) file containing a listing of SYSPAR, VECTTBL, and the link listing for RMS68K. The newly created file can now be used for further system generation.

APPENDIX D

RMS68K ERROR CODE SUMMARY

Error codes appear in the low order byte of D0 if a call to RMS68K is unsuccessful.

ERROR CODE	CAUSE OF ERROR
Issued by TRAP Handler	
$01	Directive number given in register D0 is not valid if the trap is a TRAP #1. Otherwise, there is no server for this trap or the service is unreachable from this session.
Issued by RMS68k Directives.	
$02	Parameter block address is not in requesting task's address space.
$03	Target task does not exist.
$04	Required table does not exist.
$05	Table is full; insufficient space for new entry.
$06	Duplicate request; function cannot be performed again.
$07	Entry not found in table or list.
$08	Memory space is not available.
$09	Requesting task does not have permission to request this function.
$0A	State of the target task is not valid for this directive.
$0B	Request conflicts with existing table entries.
$0C	Address of some parameter is not in requesting task's address space.
$0D	Address of some parameter is not in requesting task's address space.
$0E	Function is not enabled.
$0F	Invalid options specified in parameter block.
$10	Invalid count or length field specified in parameter block.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E

CRASH ANALYSIS GUIDE FOR VERSAdos

This appendix provides procedures for analyzing a VERSAdos system crash on VERSAmodules, VMEmodules, and on EXORmacs.

CRASH ANALYSIS GUIDE FOR VERSAdos

There is no fail indicator on some VERSAmodules or the VMEmodules so the only indication that the running system may have crashed is that there is no response to operator input. When this happens, memory must be examined to determine the cause of the system crash.

Tools that may be required when analyzing memory:

1. The output listing for the version of RMSGEN that is included in the running system. This listing contains an assembled system parameter listing.

2. The SYSGEN output produced when the running system was created.

The steps to follow when a system crash is suspected are:

1. Press the SOFTWARE ABORT button on the VERSAmodule, VMEmodule board, VME/10, VME/12, or EXORmacs. This returns control to the resident firmware debugger so that memory can be examined using the command MD ‹addr› ‹bytes›.

2. Display the CRASHSAV area in memory.

   Look up the address of CRASHSAV in the SYSPAR assembly listing. (The address of CRASHSAV may have been changed if the module ‹catalog›.SYSPARV.AG was modified.)

   EXAMPLE:

```
V*MD A00

000A00  00 00 19 C8 00 00 20 00  00 00 01 00 00 00 4E EE  ......'.......Nn
        -----PC-----      -SR--

V*MD A08 40

         ---D0----   ---D1----    ---D2----   ---D3----
000A0E  0C 00 01 00 00 00 4E EE  00 05 F7 F6 60 70 42 00  ......Nn..wv'p8.
000A1E  31 FC F0 0F 00 2E 31 7C  00 00 00 00 48 E0 00 28  1|P...1|....Km.(
000A2E  00 00 0C 00 00 FE 00 00  00 05 F6 08 00 01 5E 00  ......~....V...^.
000A3E  00 01 5E 00 00 01 5E 00  00 01 3E 00 00 00 08 D2  ..^...^...v....r
         ---A4----   ---A5----    ---A6----   ---A7----
```

If the system detected an error condition and called its crash procedure (subroutine KILLER in RMS68K), the registers are saved in the CRASHSAV area as indicated above.

If the Program Counter (PC) and Status Register (SR) displayed at CRASHSAV are 0, then the system did not crash; go to step 7.

3. Compare the PC displayed in the CRASHSAV area with the RMS68K link map to determine which module called the crash procedure. If the PC displayed is greater than the limits of RMS68K, check the SYSGEN output to see if it is within the limits of the System Initializer process (INIT.LO).

MODULE	CAUSE OF SYSTEM CRASH	ACTION
EXCEPT	An exception condition (bus error, address error, etc.) occurred while running in supervisor mode.	Go to step 4
TERM	A system task that is critical to the operating system has aborted.	Go to step 5
System Initializer INIT.LO	The system is unable to complete its initialization procedures.	Go to step 6

4. Exception condition in supervisor mode

If an exception condition is the cause of a system crash, the system stack area provides more information about the cause of the exception condition.

Register A7 (displayed in the CRASHSAV area) contained the system stack pointer when the crash happened.

Display $20 bytes of system stack.

EXAMPLE:

```
V*MD 8D2 20
        ---PC----   ---PC2---    -FC-   --ADDR---   -OP-
0008D2  00 00 19 C8 00 00 17 7A  38 65 00 05 58 71 38 6E   ................
0008E2  20 04 00 00 20 82 20 00  00 00 10 A  1C 80 00 00   ................
        -SR-   ---PC3---
```

A7+$00 --› PC   This is the same PC address saved in the CRASHSAV area.

A7+$04 --› PC2 This address points 2 bytes beyond the exception vector address for the type of exception that caused the crash.

The exception vector addresses can be found in the RMS68K link map.

SYMBOL	EXCEPTION TYPE	SYMBOL	EXCEPTION TYPE
PROGINT2	Bus error	PROGINT3	Address error
PROGINT4	Illegal instruction	PROGINT5	Zero divide
PROGINT6	CHK instruction	PROGINT7	TRAPV
PROGINT8	Priv. violation	PROGINT9	Trace
PROGINTA	Line 1010	PROGINTB	Line 1111

A7+$08 --› FC ADDR OP

These 8 bytes are placed on the stack only by the bus error and address error exceptions.

The FC field (2 bytes) contains address reference function code flags.

The ADDR field (4 bytes) contains the address that could not be accessed.

The OP field (2 bytes) contains the opcode of the instruction being executed.

A7+$10 --› SR PC3

This is the SR and PC saved when the exception occurred. If FC ADDR OP do not exist, then SR PC3 is found at A7+$08.

Knowing where the exception occurred and the address (if bus or address error) that could not be accessed can provide a clue about which SYSGEN parameter needs to be changed to run successfully.

5. SYSTEM TASK ABORT

When a critical operating system task aborts, the next step is to display the aborting task's TCB. The starting address of the aborting task's TCB was in A6 when the crash happened and was saved in the CRASHSAV area.

To interpret the contents of a TCB refer to paragraph 4.3.

Some of the important fields are shown below:

```
TENbug  2.0  > MD FA00 50
00FA00     21 54 43 42 00 00 DE 00  00 00 00 00 00 00 00 00  !TCB..^.........
00FA10     2E 49 4F 53 00 00 00 01  00 00 00 00 00 00 00 00  .IOS............
00FA20     00 00 00 00 D1 D1 D1 00  A0 82 80 10 00 80 00 80  ....QQQ.........
00FA30     00 01 00 00 00 00 00 00  FB 60 00 01 00 00 00 00  ........{`......
00FA40     00 02 AC 00 00 02 EF 00  00 00 00 00 00 00 00 00  ..,...o.........

MD FA80
00FA80     45 58 45 43 20 20 20 20  11 01 00 A0 00 00 00 FE  EXEC    .......~
```

TCB+$00		The characters '!TCB' appear in the first 4 bytes of every TCB.
TCB+$04	TCBALL	Pointer to next TCB in linked list of all TCBs.
TCB+$10	TCBNAME	Taskname.
TCB+$14	TCBSESSN	Task session number.
TCB+$2A	TCBABORT	Abort code (Flagged by -AB-- above).
TCB+$40	TCBASQ	The starting address of this task's ASQ.
TCB+$B0	TCBATSK	The taskname and session of the task that initiated the abort or Executive if RMS68K initiated the abort due to an exception condition.
TCB+$B8	TCBBERR	If a bus error or address error caused the abort, this field contains the 8 bytes of information saved on the supervisor stack when the exception occurred. The 4 bytes at TCB+$BC contain the address that could not be referenced.
TCB+$100	TCBD0	This is a save area for all 16 data and address registers as they were the last time an Executive directive was called or the task was interrupted.

```
MD FB00 40
00FB00     00 00 00 00 FA E7 00 00  00 00 00 00 00 00 00 00  ....zg..........
00FB10     00 00 00 01 00 01 B5 C0  00 01 AF B0 00 01 AF A0  ......5@../0../
00FB20     00 00 00 82 00 01 AD 28  00 02 AD 00 00 02 AD 5E  ......-(..-...-^
00FB30     00 02 AD 9C 00 02 AD B4  00 00 00 00 00 02 AE E6  ..-...-4.......f
```

TCB+$142	TCBPC	The last PC saved when task entered Executive.

```
MD FB40
00FB40     00 04 00 00 E0 CE 00 84  00 00 00 00 00 00 00 00  ....`N..........
```

A7 is the user stack pointer. If a system task aborted itself, it may have saved the error code returned by an earlier Executive directive call on its own stack; it may be useful to display the area pointed at by the task's stack pointer.

6. SYSTEM INITIALIZER ABORT

Look at the assembly output for the system initializer data segment that is part of the SYSGEN output (assembly of module INITDAT.AG). Check to see that the memory partitions were defined correctly and that other parameters were assigned reasonable values.

A list of possible initializer errors is:

a. There is no memory partition 0 starting at memory address 0.

b. Memory partitions have overlapping addresses.

c. The initializer could not find a vector table. The module ‹catalog›.VECTABLV.AG must be included within the first 512 bytes of RMS68K.

d. Memory is not available to build system tables. There may have been an error when memory limits were defined.

e. The initializer is unable to find any TCB defined.

f. Error returned by RMS68K when initializer tried to create TCBs. This can be caused by a duplicate taskname or by no memory available.

g. A bus error occurred at an address that is not a page boundary when the initializer was clearing memory. Register A3 contains the address of the memory location that caused the bus error; a probable cause of this error is a bad RAM board.

Most system initializer aborts are the result of problems defining memory limits. To display the contents of the MEMMAP, look up the address of the symbol MAPBEG in the RMS68K link map. The 4-byte address found at MAPBEG is a pointer to MEMMAP, a table defining the limits of each memory partition. Refer to paragraph 3.2.1 for a description of MEMMAP, as well as the free memory list associated with each memory partition.

EXAMPLE:

Display MAPBEG:

```
    V*MD C0U

    000C0C   0C 11 FE 1A 00 00 00 00  00 00 0C 00 00 05 EF 00  ................
```

Display MEMMAP:

```
    V*MD 11FE1A 40

    11FE1/   0C 00 00 00 58 00 00 11  FE 00 10 01 00 00 58 00  ....X..........X.
    11FE2/   0C 11 FE 00 20 02 00 00  58 00 00 11 FE 00 30 03  .... ...X.....0.
    11FE3/   0C 00 58 00 00 11 FE 00  FF FF 00 00 00 00 00 00  ..X.............
    11FE4/   0C 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

Offset $6 from the start of each RAM partition entry is a pointer to the free memory list header:

```
    V*MD 11FEC0

    11FE0C   00 01 78 00 00 00 58 00  00 18 00 00 00 00 00 01  ................
```

At offset $0 is a pointer to the first free memory list entry:

```
    V*MD 17800

    01780C   00 11 18 00 00 00 00 00  00 00 0E 88 00 00 00 E5  ................
```

7. SYSTEM DID NOT CRASH (nothing saved at CRASHSAV)

   Display the SYSPAR area in memory. Look up the address of SYSPAR in the RMS68K Linkage Editor map. The symbol RUNNER is at offset $C from the start of SYSPAR. This is the current running task's TCB address at the time of software abort. For a complete description of system parameters refer to Appendix B.

EXAMPLE:

```
    V*MD C00 40

    000C0C   0C 11 FE 1A 00 00 00 00 00 00 0C 00 00 05 EF 00  ................
    000C1C   00 01 3E 00 00 00 00 00 00 11 F0 00 00 00 00 00  ................
    000C2C   0C 11 FA 00 00 11 F9 00 00 11 F3 00 00 11 F1 00  ................
    000C3C   C0 00 00 FE 88 08 00 FE 00 00 00 00 02 E9 01 BC  ................
```

Next, display the contents of the running task's TCB. Refer to step 5 for explanation. If the running task is an operating system task (.IOS, .FHS, .TTY), display the contents of the task's ASQ (offset $40). The ASQ structure is defined in paragraph 2.3. By checking the ASQs of these tasks, a probable cause for no response at the terminal may be determined.

EXAMPLE:

```
    V*MD 5EF00 30

    005EFC0  21 41 53 51 07 30 00 01 52 86 00 00 00 00 00 05  !ASQ.0..R.......
    005EF10  EF 52 00 01 52 CC 00 05 EF 28 00 05 F0 00 00 05  .R..R....(......
    005EF20  EF 78 00 05 EF 78 00 00 00 00 00 00 00 00 00 00  .x...x..........
```

The following describes the sequence of events that takes place when the BREAK key is depressed.

a. Operator depresses BREAK key.

b. Channel I/O driver reads serial port, recognizes break, and queues an event to the terminal driver.

c. Terminal driver reads event, recognizes break code, and queues an event to the break claimer. A task qualifies as break claimer if it has issued an I/O request to claim all unclaimed breaks, or an I/O option for break service. On the released operating system, the break claimer is the command processor (&EET).

d. &EET reads the event and initiates the logon sequence.

To find the TCBs of all tasks known to the operating system, look up the address of TCBHD in the RMS68K link map. TCBHD contains a pointer to the most recently created TCB now known to the system and is the start of a linked list of all TCBs. In VERSAdos release 4.4, TCBHD can be found at offset $10 from the start of the SYSPAR area. The pointer to the next TCB in the linked list is found at offset $4 (symbol TCBALL) in each TCB. Refer to paragraph 4.3 for a complete description of a TCB and step 5 for a description of some of the important fields.

SOME COMMON CAUSES OF SYSTEM HANG-UP (no response at terminal)

a. DCB or CCB parameters specified incorrectly at SYSGEN. Refer to output of SYSGEN, specifically the listing of the assembly of IOC.‹driver_name›.AG (e.g., IOC.MPSCDRV.AG).

b. Terminal hardware problem.

c. Hardware interrupt of channel is not enabled; normally, interrupt level 5 for local terminals.

d. Spurious interrupts caused by a hardware configuration problem. Look up the symbol SPURCNT in the RMS68K link map. SPURCNT is the address of a 2-byte field that contains a count of spurious interrupts. If this value is not 0, check the hardware configuration.

e. A user program has modified memory outside the limits of its own address space. With no MMU, it is possible for a user task to crash other tasks or the operating system.

8. SYSTEM TRACE TABLE (TRC)

The TRC can provide information about the most recent events that have occurred while the system was running. Refer to paragraph 9.3.3 for a description of the TRC.

EXAMPLE:

Look at the RMS68K link map to find the address of the symbol TRACEBEG. At this address is a pointer to the start of the trace table.

```
V*MD E 30

000E 30 00 05 F1 00 C0 00 00 FE  88 08 00 FE 00 00 00 00  ................
```

Display the beginning of the trace table:

```
V*MD 5F100

05F10C  00 05 F2 F6 00 05 F2 F6  FF 15 00 00 00 00 7·
          --NEXT---   ---TOP---
```

Display part of the trace table containing recent entries:

```
V*MD 5F280 80

05F28C  00 00 97 00 00 00 00 22  02 E8 E8 80 02 F7 FF 15  ......."........
05F29C  00 10 00 00 89 5C 00 00  8E 4E 00 00 97 00 00 00  .....\...N......
05F2AC  00 4A 02 E8 E8 81 02 F8  FF 15 00 10 00 00 88 F4  .J..............
05F2BC  00 00 80 FE 00 00 97 00  00 00 00 3C 02 E8 E8 82  ...........<....
05F2CC  02 6F FF 15 00 00 00 00  86 42 00 00 80 FE 00 00  .o.......B......
05F2DC  97 00 00 00 00 24 02 E8  E8 85 02 E0 FF 15 00 00  .....$..........
05F2EC  00 00 75 6A 00 00 00 82  00 00 7C 00 00 00 00 35  ..uj...........5
05F2FC  02 E8 E8 86 01 87 00 00  00 00 00 00 00 00 00 00   ................
```

The most recent entry in the above table is:

```
TRCCODE TRCSR    TRCPC     TRCA0     TRCA6     TRCD0   TRCTIME   TRCTIM2
 FF15    0000 00C0756A 00000082 00007C00 00000035 02E8E886     0187
```

The code FF15 indicates that a TRAP #1 was executed.

The address at TRCA6 is the address of the TCB of the task that executed the TRAP #1.

The calling task's D0=$35. This is the directive number for the DERQST directive.

TRCSR and TRCPC contain the SR and PC of the task that executed the trap.

To learn more about the task that was responsible for any entry, examine that task's TCB as described in step 5.

When examining the TRC, it may also be useful to look for similar entries that are frequently repeated indicating a loop in some program, or a table that is completely full of I/O Interrupt entries (TRCCODE=$EE14) indicating that some interrupt may not be getting cleared.

APPENDIX F

GLOSSARY OF RMS68K TERMS

Asynchronous Mode

A mode of processing events where the task is dispatched to its ASR to handle the incoming event.

Asynchronous Service Queue (ASQ)

A FIFO queue used for management of event messages between tasks or between RMS68K and a task. The ASQ can be used by a task to process events synchronously or asynchronously.

Asynchronous Service Routine (ASR)

A part of a task's program code that asynchronously processes event messages in the task's ASQ. The ASR operates in a software interrupt mode.

Autovector

MC68000 Family microprocessor interrupt-caused exception vectors. There are seven autovectors corresponding to seven levels of interrupt priority.

Channel Control Block (CCB)

A block of data containing variables used to control an I/O Channel.

Channel Management Routine (CMR)

CMRs are an optional layer of RMS68K functionality for managing I/O Channels and I/O Requests.

CRASHSAV

Area of memory containing information describing system crashes such as program counter, status registers, and registers D0 to D7 and A0 to A7 at time of system crash. (Refer to Appendix E for format.)

Concurrent Processing

An operation mode where more than one process seems to be in progress at a given time even though the processes are actually sharing the processor.

Device Control Block (DCB)

A block of data containing variables and static data used to control an I/O device.

Exception Monitor Task

A task that can monitor one or more other tasks and be notified of any exceptions that occur within those tasks.

Exception Vector

A memory location from which the M68000 Family microprocessor fetches the address of a routine that handles an exception.

Executive Directive

A request issued by a task for services of RMS68K.

Free Memory List (FML)

Doubly linked list of nodes describing current status of free memory within a RAM partition.

Global Segment Table (GST)

Array of segment descriptors for all currently defined shareable segments within the system.

Interrupt Service Routine (ISR)

A part of a task's program code that handles interrupts. The ISR operates in an asynchronous mode with the task.

I/O Vector Table (IOV)

Array of descriptors for all tasks currently claiming interrupts via the CISR directive.

Memory Management Unit (MMU)

Hardware device that provides mapping of logical memory addresses to physical addresses and protects one task's address space from unauthorized accesses from other tasks.

Memory Map Table (MEMMAP)

Array of partition descriptors for all RAM or ROM partitions within the system.

Monitor Task

A task that receives automatic notification on the termination of one or more other tasks, called subtasks of the monitor task.

Multitasking

An operation mode where more than one functionally bound task is being processed concurrently.

Non Real-time Task

Task that does not execute under severe time constraints. These tasks may have any mapping of address space and use the taskname, session number identification when referring to a target task.

Periodic Activation Table (PAT)

Linked list of nodes describing all current demands for elapsed time notification.

Program Counter

Internal processor register that contains the address of the instruction currently being fetched by the processor.

Real-time Task

Task that executes under severe performance constraints. To meet those constraints, real-time tasks must be mapped with logical addresses equal to physical addresses and must use the internally generated code output by the GTTASKID directive when referring to a target task.

Segment

A block of memory that can be used for data or program code. Every task can consist of up to four segments. Segments can be shared by more than one task.

Semaphore

A unit representing a count of signals used for synchronizing task activity or controlling the use of resources.

Server Task

A task that operates like an extension of RMS68K, and can provide a service to any task in the system on request.

Session

A group of related tasks, identified by a session number.

Status Register (SR)

Internal processor register that contains status information such as supervisor/user node, trace bit, interrupt mask level, and condition codes.

Supervisor Hardware State

A privileged state of M68000 Family microprocessor execution. No restrictions are placed on operations. RMS68K operates in the supervisor hardware state.

Synchronous Mode

The mode of operation where the task processes the event inline, i.e., the task is not sent off to its ASR but dispatched to the instruction immediately following the GTEVNT or RDEVNT call in the main line code of the task.

System Parameter Table (SYSPAR)

Unstructured list of miscellaneous system parameters.

System Task

A task that operates in the user hardware state of the M68000 Family microprocessor and can affect other tasks executing within alien sessions.

System Trace Table (TRC)

Circular queue of traced events describing system history.

Task

A functionally bound group of one or more modules that can operate concurrently with other tasks.

Task_ID

An 8-byte code for referencing a target task. For real-time requesting tasks, it is an internally generated code; for non real-time requesting tasks, it is the taskname and session number of the target task. Note that the format of the task_id is dependent on the real-time/non real-time type of the requestor and is not dependent on the type of the target.

Task Control Block (TCB)

Unstructured list of miscellaneous task state information.

Taskname

A means of identifying a particular task within a session.

Task Priority

A relative level of importance given to a task. Tasks that are more urgent are assigned higher priorities.

Task Segment Table

Array of segment descriptors for all segments currently accessible to a task.

Trap Instruction Assignment Table (TIAT)

Byte array indexed by trap number that defines each trap instruction as being unassigned, reserved for RMS68K, or assigned to a server task.

Trap Instruction Owner Table (TIOT)

Array of descriptors for currently defined trap servers indexed by trap number.

Trap Vector

A particular type of M68000 Family microprocessor exception vector, corresponding to M68000 Family TRAP instructions.

User Directive Table (UDR)

Array of descriptors indexed by directive number for all directives currently defined by tasks via the CDIR directive.

User Hardware State

The normal state of execution of the M68000 Family microprocessor. There are restrictions on the operations allowed. User tasks and system tasks execute in this state.

User Semaphore Table (UST)

Array of semaphore descriptors indexed by semaphore key for all currently defined semaphores created by the CRSEM or ATSEM directives.

User Stack Pointer (USP)

Stack pointer in use when processor is executing in the user hardware mode.

User Task

A task that operates in the user hardware state of the M68000 Family microprocessor and can only affect other tasks executing within its own session.

User Vector

A particular type of M68000 Family microprocessor exception vector. They can be assigned by the user.

XDEF

Assembler construct for declaring labels to be accessible to external modules.

APPENDIX G

SUMMARY OF DIRECTIVES

B.1 ALPHABETICAL SUMMARY OF RMS68K DIRECTIVES

Following are the directives that tasks can issue to RMS68K by using the TRAP #1 instruction.

DIRECTIVE NAME	DIRECTIVE MEANING	DIRECTIVE NUMBER	PARAMETER	RETURN PARAMETER	PAGE
ABORT*	Task aborts itself	14	A0	-	124
AKRQST	Server acknowledge request	54	A0	-	186
ATSEM	Attach to semaphore	41	A0	A0	164
ATTSEG	Attach a shareable segment	4	A0	A0	80
CDIR	Configure a new directive	58	A0	-	225
CISR	Configure ISR	61	A0	-	215
CRSEM	Create semaphore	45	A0	A0	162
CRTCB*	Create TCB	11	A0	-	111
DCLSHR	Declare a segment shareable	7	A0	-	76
DEASQ*	Detach ASQ	32	-	-	42
DELAY*	Task moves itself to DELAY state	21	A0	-	144
DELAYW	DELAY, WTEVNT, and WAIT functions are performed	30	A0	-	145
DERQST	Set user/server request status	53	A0	-	188
DESEG*	Detach a segment	2	A0	-	74
DESEM	Detach from semaphore	44	A0	-	167
DESEMA	Detach from all semaphores	46	-	-	168

* Required for RSM68K; cannot be removed from the system.

DIRECTIVE NAME	DIRECTIVE MEANING	DIRECTIVE NUMBER	PARAMETER	RETURN PARAMETER	PAGE
DEXMON	Detach exception monitor	65	A0	-	206
DSERVE	Detach server function	52	A0	-	190
EXMMSK	Set exception monitor mask	66	A0	-	196
EXMON	Attach exception monitor	64	A0	-	194
EXPVCT	Announce exception vectors	26	A0	-	221
FLUSHC	Flush user cache	75	-	-	96
GTASQ	Allocate ASQ	31	A0	-	28
GTDTIM	Get date and time	74	A0	Buffer	154
GTEVNT	Get an event	38	A0	-	36
GTSEG	Allocate a segment	1	A0	A0,A1	68
GTTASKID	Get a target task's task_id	10	A0	A0,A1	134
GTTASKNM	Get a target task's taskname and session number	12	A0	A0,A1	136
MOVELL	Move from logical address	6	A0	Buffer	92
MOVEPL	Move from physical address	72	A0	Buffer	94
PSTATE	Modify task state	68	A0	-	204
QEVNT	Queue event to task's ASQ	35	A0	-	33
RCVSA	Receive segment attributes	9	A0	Buffer	89
RDEVNT	Task reads event from its ASQ	34	A0	Buffer	38
RELINQ	Task moves itself from RUN state to READY state	22	-	-	129
RESUME	Target task goes to READY state from SUSPEND state	18	A0	-	128
REXMON	Run task under exception monitor control	69	A0	-	198
RQSTPA	Task is set up to be periodically activated	29	A0	-	147

DIRECTIVE NAME	DIRECTIVE MEANING	DIRECTIVE NUMBER	PARAMETER	RETURN PARAMETER	PAGE
RSTATE	Receive task state	67	A0	Buffer	201
RTE	Return from ISR execution	-	D0,D1,D2	-	220
RTEVNT	ASR returns after event servicing	37	A0	-	41
SERVER	Task is made a server task	51	A0	-	183
SETASQ	Task changes its ASQ/ASR status	33	A0	-	31
SETPRI	Change priority of a task	24	A0	-	117
SGSEM	Signal semaphore	43	A0	-	166
SHRSEG	Grant shared segment access	5	A0	A0	83
SINT	Simulate interrupt	62	A0	-	218
SNPTRC	Snapshot of system trace	8	A0	Buffer	228
START	Target task goes to READY state from DORMANT state	13	A0	-	114
STDTIM	Set date and time	73	A0	-	152
STOP	Target task goes to DORMANT state from any state	25	A0	A0	119
SUSPND	Task moves itself to SUSPEND state	17	-	-	127
TERM*	Task terminates itself	15	-	-	121
TERMT*	Target task is terminated from any state	16	A0	A0	122
TRPVCT	Announce trap vectors	27	A0	-	223
TRSEG	Transfer a segment	3	A0	A0	86
TSKATTR	Receive task user number and attributes	23	A0	A0	130
TSKINFO	Receive copy of TCB	28	A0	Buffer	132

* Required for RMS68K; cannot be removed from system.

DIRECTIVE NAME	DIRECTIVE MEANING	DIRECTIVE NUMBER	PARAMETER	RETURN PARAMETER	PAGE
WAIT	Task moves itself to WAIT state	19	-	-	125
WAKEUP*	Target task goes to READY state from WAIT state	20	A0	-	126
WTEVNT	Task moves itself to WAIT FOR EVENT state	36	-	-	40
WTSEM	Wait on semaphore	42	A0	-	165

* Required for RMS68K; cannot be removed from system.

B.2 NUMERIC SUMMARY OF RMS68K DIRECTIVES

Following are the directives that tasks can issue to RMS68K by using the TRAP #1 instruction.

DIRECTIVE NAME	DIRECTIVE MEANING	DIRECTIVE NUMBER	PARAMETER	RETURN PARAMETER	PAGE
RTE	Return from ISR execution	-	D0,D1,D2	-	220
GTSEG	Allocate a segment	1	A0	A0,A1	68
DESEG*	Detach a segment	2	A0	-	74
TRSEG	Transfer a segment	3	A0	A0	86
ATTSEG	Attach a shareable segment	4	A0	A0	80
SHRSEG	Grant shared segment access	5	A0	A0	83
MOVELL	Move from logical address	6	A0	Buffer	92
DCLSHR	Declare a segment shareable	7	A0	-	76
SNPTRC	Snapshot of system trace	8	A0	Buffer	228
RCVSA	Receive segment attributes	9	A0	Buffer	89
GTTASKID	Get target task's task_id	10	A0	A0,A1	134
CRTCB*	Create TCB	11	A0	-	111
GTTASKNM	Get a target task's taskname and session number	12	A0	A0,A1	136
START	Target task goes to READY state from DORMANT state	13	A0	-	114
ABORT*	Task aborts itself	14	A0	-	124
TERM*	Task terminates itself	15	-	-	121
TERMT*	Target task is terminated from any state	16	A0	A0	122
SUSPND	Task moves itself to SUSPEND state	17	-	-	127
RESUME	Target task goes to READY state from SUSPEND state	18	A0	-	128

* Required for RSM68K; cannot be removed from the system.

DIRECTIVE NAME	DIRECTIVE MEANING	DIRECTIVE NUMBER	PARAMETER	RETURN PARAMETER	PAGE
WAIT	Task moves itself to WAIT state	19	-	-	125
WAKEUP*	Target task goes to READY state from WAIT state	20	A0	-	126
DELAY*	Task moves itself to DELAY state	21	A0	-	144
RELINQ	Task moves itself from RUN state to READY state	22	-	-	129
TSKATTR	Receive task user number and attributes	23	A0	A0	130
SETPRI	Change priority of a task	24	A0	-	117
STOP	Target task goes to DORMANT state from any state	25	A0	A0	119
EXPVCT	Announce exception vectors	26	A0	-	221
TRPVCT	Announce trap vectors	27	A0	-	223
TSKINFO	Receive copy of TCB	28	A0	Buffer	132
RQSTPA	Task is set up to be periodically activated	29	A0	-	147
DELAYW	DELAY, WTEVNT, and WAIT functions are performed	30	A0	-	145
GTASQ	Allocate ASQ	31	A0	-	28
DEASQ*	Detach ASQ	32	-	-	42
SETASQ	Task changes its ASQ/ASR status	33	A0	-	31
RDEVNT	Task reads event from its ASQ	34	A0	Buffer	38
QEVNT	Queue event to task's ASQ	35	A0	-	33
WTEVNT	Task moves itself to WAIT FOR EVENT state	36	-	-	40
RTEVNT	ASR returns after event servicing	37	A0	-	41

* Required for RMS68K; cannot be removed from the system.

DIRECTIVE NAME	DIRECTIVE MEANING	DIRECTIVE NUMBER	PARAMETER	RETURN PARAMETER	PAGE
GTEVNT	Get an event	38	A0	-	36
ATSEM	Attach to semaphore	41	A0	A0	164
WTSEM	Wait on semaphore	42	A0	-	165
SGSEM	Signal semaphore	43	A0	-	166
DESEM	Detach from semaphore	44	A0	-	167
CRSEM	Create semaphore	45	A0	A0	162
DESEMA	Detach from all semaphores	46	-	-	168
SERVER	Task is made a server task	51	A0	-	183
DSERVE	Detach server function	52	A0	-	190
DERQST	Set user/server request status	53	A0	-	188
AKRQST	Server acknowledge request	54	A0	-	186
CDIR	Configure a new directive	58	A0	-	225
CISR	Configure ISR	61	A0	-	215
SINT	Simulate interrupt	62	A0	-	218
EXMON	Attach exception monitor	64	A0	-	194
DEXMON	Detach exception monitor	65	A0	-	206
EXMMSK	Set exception monitor mask	66	A0	-	196
RSTATE	Receive task state	67	A0	Buffer	201
PSTATE	Modify task state	68	A0	-	204
REXMON	Run task under exception monitor control	69	A0	-	198
MOVEPL	Move from physical address	72	A0	Buffer	94
STDTIM	Set date and time	73	A0	-	152
GTDTIM	Get date and time	74	A0	Buffer	154
FLUSHC	Flush user cache	75	-	-	96

G

B.3 SUMMARY OF RMS68K DIRECTIVES BY FUNCTION

Following are the directives that tasks can issue to RMS68K by using the TRAP #1 instruction.

DIRECTIVE NAME	DIRECTIVE MEANING	DIRECTIVE NUMBER	PARAMETER	RETURN PARAMETER	PAGE
Event Manager					
QEVNT	Queue event to task's ASQ	35	A0	-	33
RDEVNT	Task reads event from its ASQ	34	A0	Buffer	38
RTEVNT	ASR returns after event servicing	37	A0	-	41
GTEVNT	Get an event	38	A0	-	
GTASQ	Allocate ASQ	32	A0	-	28
DEASQ*	Detach ASQ	32	-	-	42
SETASQ	Task changes its ASQ/ASR status	33	A0	-	31
WTEVNT	Task moves itself to WAIT FOR EVENT state	36	-	-	40
Memory Manager					
GTSEG	Allocate a segment	1	A0	A0,A1	68
DESEG*	Detach a segment	2	A0	-	74
DCLSHR	Declare a segment shareable	7	A0	-	76
ATTSEG	Attach a shareable segment	4	A0	A0	80
SHRSEG	Grant shared segment access	5	A0	A0	83
TRSEG	Transfer a segment	3	A0	A0	86
RCVSA	Receive segment attributes	9	A0	Buffer	89
MOVELL	Move from logical address	6	A0	Buffer	92
MOVEPL	Move from physical address	72	A0	Buffer	94
FLUSHC	Flush user cache	75	-	-	96

* Required for RMS68K; cannot be removed from the system.

DIRECTIVE NAME	DIRECTIVE MEANING	DIRECTIVE NUMBER	PARAMETER	RETURN PARAMETER	PAGE
Task Manager					
CRTCB*	Create TCB	11	A0	-	111
START	Target task goes to READY state from DORMANT state	13	A0	-	114
SETPRI	Change priority of a task	24	A0	-	117
STOP	Target task goes to DORMANT state from any state	25	A0	A0	119
TERM*	Task terminates itself	15	-	-	121
TERMT*	Target task is terminated from any state	16	A0	A0	122
ABORT*	Task aborts itself	14	A0	-	124
WAIT	Task moves itself to WAIT state	19	-	-	125
WAKEUP*	Target task goes to READY state from WAIT state	20	A0	-	126
SUSPND	Task moves itself to SUSPEND state	17	-	-	127
RESUME	Target task goes to READY state from SUSPEND state	18	A0	-	128
RELINQ	Task moves itself from RUN state to READY state	22	-	-	129
TSKATTR	Receive task user number and attributes	23	A0	A0	130
TSKINFO	Receive copy of TCB	28	A0	Buffer	132
GTTASKID	Get a target task's task_id	10	A0	A0,A1	134
GTTASKNM	Get a target task's taskname and session number	12	A0	A0,A1	136

* Required for RMS68K; cannot be removed from system.

DIRECTIVE NAME	DIRECTIVE MEANING	DIRECTIVE NUMBER	PARAMETER	RETURN PARAMETER	PAGE
<u>Time Manager</u>					
DELAY*	Task moves itself to DELAY state	21	A0	-	144
DELAYW	DELAY, WTEVNT, and WAIT functions are performed	30	A0	-	145
RQSTPA	Task is set up to be periodically activated	29	A0	-	147
STDTIM	Set date and time	73	A0	-	152
GTDTIM	Get date and time	74	A0	Buffer	154
<u>Semaphore Manager</u>					
CRSEM	Create semaphore	45	A0	A0	162
ATSEM	Attach to semaphore	41	A0	A0	164
WTSEM	Wait on semaphore	42	A0	-	165
SGSEM	Signal semaphore	43	A0	-	166
DESEM	Detach from semaphore	44	A0	-	167
DESEMA	Detach from all semaphores	46	-	-	168
<u>Trap Server Manager</u>					
SERVER	Task is made a server task	51	A0	-	183
AKRQST	Server acknowledge request	54	A0	-	186
DERQST	Set user/server request status	53	A0	-	188
DSERVE	Detach server function	52	A0	-	190

* Required for RMS68K; may not be removed from system.

DIRECTIVE NAME	DIRECTIVE MEANING	DIRECTIVE NUMBER	PARAMETER	RETURN PARAMETER	PAGE
Exception Monitor Manager					
EXMON	Attach exception monitor	64	A0	-	194
EXMMSK	Set exception monitor mask	66	A0	-	196
REXMON	Run task under exception monitor control	69	A0	-	198
RSTATE	Receive task state	67	A0	Buffer	201
PSTATE	Modify task state	68	A0	-	204
DEXMON	Detach exception monitor	65	A0	-	206
Exception Manager					
CISR	Configure ISR	61	A0	-	215
SINT	Simulate interrupt	62	A0	-	218
RTE	Return from ISR execution	-	D0,D1,D2	-	220
EXPVCT	Announce exception vectors	26	A0	-	221
TRPVCT	Announce trap vectors	27	A0	-	223
CDIR	Configure a new directive	58	A0	-	225
SNPTRC	Snapshot of system trace	8	A0	Buffer	228

G

THIS PAGE INTENTIONALLY LEFT BLANK

SUGGESTION/PROBLEM REPORT

MICROSYSTEMS
QUALITY • PEOPLE • PERFORMANCE

Motorola welcomes your comments on its products and publications. Please use this form.

To: Motorola Inc.
Microsystems
2900 S. Diablo Way
Tempe, Arizona 85282
Attention: Publications Manager
Maildrop DW164

Product: ______________________ Manual: ______________________

COMMENTS: ______________________

Please Print

Name ______________________ Title ______________________

Company ______________________ Division ______________________

Street ______________________ Mail Drop ____________ Phone ____________

City ______________________ State ____________ Zip ____________

For Additional Motorola Publications
Literature Distribution Center
616 West 24th Street
Tempe, AZ 85282
(602) 994-6561

Four Phase/Motorola Customer Support, Tempe Operations
(800) 528-1908
(602) 438-3100

MOTOROLA

MOTOROLA *Semiconductor Products Inc.*

P.O. BOX 20912 • PHOENIX, ARIZONA 85036 • A SUBSIDIARY OF MOTOROLA INC.

17343-1 PRINTED IN USA (1/86) MESSENGER 1500