

SIEMENS

RMOS3

RMOS3 real-time operating system RMOS3 V3.50 User Manual




Operating Manual

Things worth knowing about your RMOS3 documentation	1
RMOS3 introduction	2
Necessary hardware and software	3
Installing RMOS3	4
Human Interface of RMOS3	5
Practical section: Creating an RMOS3 task	6
Practical section: Testing an RMOS3 task	7
Properties of the RMOS3 operating system	8
RMOS3 system configuration	9
Abbreviations/Glossary	A

Legal information

Warning notice system

This manual contains notices you have to observe in order to ensure your personal safety, as well as to prevent damage to property. The notices referring to your personal safety are highlighted in the manual by a safety alert symbol, notices referring only to property damage have no safety alert symbol. These notices shown below are graded according to the degree of danger.

 DANGER
indicates that death or severe personal injury will result if proper precautions are not taken.
 WARNING
indicates that death or severe personal injury may result if proper precautions are not taken.
 CAUTION
indicates that minor personal injury can result if proper precautions are not taken.
NOTICE
indicates that property damage can result if proper precautions are not taken.


If more than one degree of danger is present, the warning notice representing the highest degree of danger will be used. A notice warning of injury to persons with a safety alert symbol may also include a warning relating to property damage.

Qualified Personnel

The product/system described in this documentation may be operated only by **personnel qualified** for the specific task in accordance with the relevant documentation, in particular its warning notices and safety instructions. Qualified personnel are those who, based on their training and experience, are capable of identifying risks and avoiding potential hazards when working with these products/systems.

Proper use of Siemens products

Note the following:

 WARNING
Siemens products may only be used for the applications described in the catalog and in the relevant technical documentation. If products and components from other manufacturers are used, these must be recommended or approved by Siemens. Proper transport, storage, installation, assembly, commissioning, operation and maintenance are required to ensure that the products operate safely and without any problems. The permissible ambient conditions must be complied with. The information in the relevant documentation must be observed.

Trademarks

All names identified by ® are registered trademarks of Siemens AG. The remaining trademarks in this publication may be trademarks whose use by third parties for their own purposes could violate the rights of the owner.

Disclaimer of Liability

We have reviewed the contents of this publication to ensure consistency with the hardware and software described. Since variance cannot be precluded entirely, we cannot guarantee full consistency. However, the information in this publication is reviewed regularly and any necessary corrections are included in subsequent editions.

Table of contents

1	Things worth knowing about your RMOS3 documentation	9
1.1	About this document	10
1.2	Notations	11
2	RMOS3 introduction	15
2.1	Process control with separate HMI	17
2.2	RMOS3 performance features	19
2.2.1	Multitasking and multiprocessing mechanisms	20
2.2.2	Real-time behavior and suitability for industrial applications	23
2.2.3	Configurable nucleus	26
2.2.4	Boot sequence	27
2.2.5	Performance scope of the RMOS3 nucleus	27
2.2.6	Hardware requirements and development environment	30
2.2.7	Test strategies	30
2.2.8	Product overview	32
2.2.9	General fields of application for RMOS3	34
2.3	Summary	35
3	Necessary hardware and software	39
3.1	RMOS3 target system	39
3.2	RMOS3 development system	40
3.3	Programming and debugging tools for task development	40
3.4	Programming and debugging tools for system generation	41
4	Installing RMOS3	43
4.1	Installing the development environment	43
4.1.1	Installing the RMOS3 development environment	43
4.1.1.1	Update	44
4.1.1.2	New installation	44
4.1.2	Installation of the compilers, IDE, and debugger	47
4.1.3	Installation of additional products, Board Support Packages	47
4.1.4	Notes on using the RMOS3 utilities	48
4.2	Commissioning the target system	48
4.2.1	Booting from mass storage media	49
4.2.1.1	New installation of the target system using a USB Flash Drive that supports booting in DOS	49
4.2.1.2	New installation of the target systems from a USB Flash Drive that supports booting from Windows PE	52
4.2.1.3	Updating the target system	54

5	Human Interface of RMOS3.....	57
5.1	Starting the RMOS3 operating system	57
5.2	Operating system I/Os on the PC	57
5.3	Command line interpreter CLI	59
5.4	RMOS3 debugger	60
6	Practical section: Creating an RMOS3 task.....	61
6.1	Creating an RMOS3 task using the GNU tools.....	61
6.2	Notes on programming and loading tasks	61
7	Practical section: Testing an RMOS3 task.....	63
7.1	Testing using the GNU tools	63
7.2	SVC exception handler, status messages and test output	63
7.3	Testing with RMOS3 Debugger	64
8	Properties of the RMOS3 operating system.....	67
8.1	Memory protection	71
8.2	Multiprocessing in RMOS3.....	75
8.2.1	Startup.....	75
8.2.2	Task properties	76
8.2.3	Task management	80
8.2.4	RMOS3 API for task management.....	85
8.2.5	Creating and deleting tasks.....	85
8.2.6	Binding tasks.....	86
8.2.7	Starting tasks	86
8.2.7.1	Task start through unexpected input.....	87
8.2.7.2	Parameter passing at task start	87
8.2.8	Task priority.....	88
8.2.8.1	Priority changes by SVCs	88
8.2.8.2	Priority change triggered by timeout	88
8.2.8.3	Automatic priority change by means of semaphore possession.....	89
8.2.9	Memory distribution.....	89
8.2.10	I/O request.....	89
8.2.11	SYSTEM HALTED	90
8.2.12	Performance.....	90
8.2.13	Notes on porting.....	91
8.3	Interrupt processing in RMOS3.....	92
8.3.1	Basics of interrupt processing	93
8.3.2	What is an interrupt handler?	97
8.3.3	DI state.....	98
8.3.4	I state.....	99
8.3.5	S state	101
8.3.6	A state	103
8.3.7	Overview of the operating states and RMOS3-SVCs for interrupt processing	103
8.4	Task communication/coordination/synchronization	106
8.4.1	Communication and coordination by starting a task	108
8.4.2	Communication and coordination by means of semaphores.....	108

8.4.3	Communication and coordination by means of spinlocks.....	110
8.4.4	Communication by means of event flags.....	111
8.4.5	Communication via local mailboxes.....	113
8.4.6	Message communication.....	114
8.4.7	Communication by means of shared data areas.....	115
8.5	Resource management.....	116
8.5.1	Resource catalog.....	117
8.5.2	Time management.....	118
8.5.2.1	Time-related system calls.....	119
8.5.3	Memory management.....	122
8.5.4	Descriptor management.....	123
8.5.5	Driver I/O management.....	125
8.5.6	System calls.....	126
8.6	DEBUG strategies in RMOS3.....	127
8.6.1	Test strategies.....	128
8.6.2	Test tools.....	130
8.6.2.1	RMOS3 debugger and resource reporter.....	130
8.6.2.2	Host debugger.....	130
8.6.2.3	Hardware emulator.....	131
8.6.3	System startup messages.....	132
8.6.4	Error messages/error logger task.....	133
8.6.5	Exception interrupt handler.....	134
8.6.6	SVC exception handler.....	136
8.6.7	Debugging with the RMOS3 debugger.....	137
8.6.8	Testing with the RMOS3 profiler.....	138
8.6.8.1	Practical exercise with the RMOS3 profiler.....	138
8.6.9	Debugging at source code level.....	147
8.7	Basic I/O system of RMOS3.....	148
8.8	Arithmetic co-processor.....	150
8.9	C Runtime library CRUN.....	151
8.9.1	Conditions for using the C library.....	151
8.9.2	Mathematical functions.....	152
8.9.3	Special features of the C Runtime library.....	156
8.9.4	Example programs for using the C library.....	157
9	RMOS3 system configuration.....	159
9.1	Overview.....	159
9.2	Boot sequence and memory allocation.....	159
9.3	System properties of the RMOS3 nucleus.....	163
9.3.1	Allocation of the device units.....	163
9.3.2	Software configuration.....	165
9.3.3	HD integration.....	166
9.3.4	Warm restart.....	166
A	Abbreviations/Glossary.....	167
	Index.....	173

Tables

Table 1- 1	Commands	12
Table 1- 2	Abbreviations	12
Table 1- 3	General data types	13
Table 1- 4	Data types of the RMOS3 API	13
Table 2- 1	RMOS3 system utilities, resources overview	28
Table 5- 1	Keyboard layouts	58
Table 8- 1	Privilege levels of system tasks	72
Table 8- 2	Access rights	72
Table 8- 3	RMOS3 API for task management.....	85
Table 8- 4	RMOS3 SVCs for interrupt handlers	104
Table 8- 5	Task communication/coordination/synchronization	107
Table 8- 6	Resource management.....	116
Table 8- 7	Overview of resources	117
Table 8- 8	Performance features of the test tools	129

Figures

Figure 2-1	Process control with separate HMI	18
Figure 2-2	Process control with RMOS3	18
Figure 2-3	Multitasking principle in RMOS3	20
Figure 2-4	RMOS3 task status graph	22
Figure 2-5	Operating states of real-time interrupts.....	25
Figure 2-6	Host-/target-oriented debugging concept.....	31
Figure 2-7	Operating system structure of RMOS3	32
Figure 4-1	Installation preparation.....	44
Figure 4-2	Start screen	45
Figure 4-3	Selecting the target directory	45
Figure 4-4	Setting the environment variables.....	46
Figure 4-5	Select a directory for the documentation	46
Figure 4-6	End of the installation.....	47
Figure 8-1	Operating system structure	68
Figure 8-2	Overview of privilege levels.....	71
Figure 8-3	Tasks from the point of view of user programs	77
Figure 8-4	RMOS3 data structures for task management.....	78

Figure 8-5	Segmentation of an application in tasks	79
Figure 8-6	Task state and transitions	81
Figure 8-7	Scheduling mechanism, based on the example of a system with two cores.....	82
Figure 8-8	Ready Task Queue	83
Figure 8-9	Interrupt processing on the 80x86 processor	93
Figure 8-10	Cascading the interrupt controllers	94
Figure 8-11	Operating states of real-time interrupts	103
Figure 8-12	Using semaphores for task communication and co-ordination.....	109
Figure 8-13	Using spinlocks for task communication and co-ordination.....	111
Figure 8-14	Communication by means of event flags.....	112
Figure 8-15	Communication via local mailboxes.....	113
Figure 8-16	Communication by means of message queues.....	114
Figure 8-17	Communication by means of shared memory	115
Figure 8-18	Concept of the four privilege levels in the 80386 architecture.....	123
Figure 8-19	Loading a descriptor	124
Figure 8-20	Debugger communication	131
Figure 8-21	F_TASK extended with a WHILE loop	139
Figure 8-22	Structure of drivers and devices	149
Figure 9-1	Memory mapping in a typical PC architecture	160
Figure 9-2	Basic structure of RMOS3 on PC	162

Things worth knowing about your RMOS3 documentation

1

Manuals

The RMOS3 documentation consists of the User Manual, Reference Manuals, and the System Manual.

The user manual provides you with information pertaining to the handling of the product and, partially in the form of a text book, gets you familiarized with the RMOS3 properties and options.

The system manual describes the operating system and its configuration, including the programming interfaces.

Readership of the reference manuals are system operators and programmers.

What do you want to do?	User manual	Reference manual	System manual
Getting started with RMOS3	X		
Operating RMOS3		Part I	
Programming tasks		Part II and III	
Configuring the operating system		Part IV	X
Writing drivers		Part IV	X

Finding your way through the documentation

Your approach to the documentation will differ, depending on whether you are a newcomer to RMOS3, or whether you have already collected knowledge in programming the RMOS3 system. In this context we would like to offer you some advice. We assume that you are familiar with the DOS commands of your PC. Knowledge of the C programming language is required.

RMOS3 newcomers

If not already familiar with RMOS3, read this user manual carefully, working step by step. It is advisable to study at least chapters 2 through 8 in succession. Chapter "RMOS3 system configuration (Page 159)" deepens your insight with regard to the structure of the operating system nucleus in RMOS3.

Information related to the RMOS3 utility programs and on the usage of the RMOS3 HMI is available in part I of the reference manual.

Expert RMOS3 programmers

Experienced RMOS3 programmers will find programming support in parts II and III of the reference manual, which list the C functions available in RMOS3, the RMOS3 SVCs, and other RMOS3 language elements.

If you intend to program your own drivers, you will find the necessary information in part IV of the reference manual and in the system manual.

RMOS3 system programmers

The system manual provides you with information you need to configure the operating system, or to add, edit or remove elements. The manual also describes all RMOS3 components.

1.1 About this document

Overview

The user manual provides a product overview and shows you how to install RMOS3. It explains how to create, execute and debug RMOS3 programs. The topics covered in the manual also include communication between RMOS3 tasks and the RMOS3 properties. The chapters toward the end of this documentation provide information about the structure of the RMOS3 nucleus.

Chapter 1

"Things worth knowing about your RMOS3 documentation (Page 9)" offers you a documentation guideline and explains the notations used in the manuals.

Chapter 2

"RMOS3 introduction (Page 15)" highlights essential terminology and relations of the RMOS3 operating system variant.

Chapter 3

"Necessary hardware and software (Page 39)" describes the conditions for installing and running RMOS3 on a PC. Moreover, it provides you with the details on requirements for developing RMOS3 programs and configurations on your PC. We show you exactly which programming tools you may use for this purpose, and which compilers you must use to compile RMOS3 programs.

Chapter 4

"Installing RMOS3 (Page 43)" shows you how to install RMOS3 on your computer.

Chapter 5

"Human Interface of RMOS3 (Page 57)" helps you to launch RMOS3 on your computer and explains the different consoles that you can use to operate RMOS3. The document also introduces the RMOS3 Command Line Interpreter and the RMOS3 Debugger. Demo programs provide you with a first impression of the possibilities RMOS3 offers.

Chapters 6 and 7

"Practical section: Creating an RMOS3 task (Page 61)" and "Practical section: Testing an RMOS3 task (Page 63)" show you, based on a single example, how to create or start executable programs in RMOS3, and how to debug these with the help of the RMOS3 debugging tools.

Chapter 8

"Properties of the RMOS3 operating system (Page 67)" describes the performance features of RMOS3, for example, memory protection, multitasking mechanisms, real-time behavior, boot sequence and memory allocation, as well as task communication, resource management and debug strategies.

Chapter 9

"RMOS3 system configuration (Page 159)" outlines the structure of an RMOS3 nucleus.

1.2 Notations

We use the following notation to enhance legibility of the RMOS3 documentation:

Commands

Commands control program sequences in dialog or batch mode. The Courier font is used in the text to highlight the commands. A command consists of at least one element. Elements are constants or variables. They are comprised of letters, numbers and special characters (e.g. `#*?`).

The Courier font is also used to present program listings. They are printed in compliance with the characters and do not follow the notation for commands. The programming language C, for example, distinguishes between uppercase and lowercase notation.

1.2 Notations

Table 1- 1 Commands

Representation	Property	Comment
UPPERCASE	Constant	Elements in uppercase notation represent constants. Entries are made in accordance with the character notation with support of uppercase and lowercase letters.
1847	Constant	Numbers are always constants.
x	Variable	Elements in lowercase notation represent variables that can be replaced by current elements.
()\; >	Special characters	Special characters and whitespaces serve as delimiters between elements and always need to be entered.
The use of elements is described by meta characters. Meta characters are not entered.		
Representation	Property	Comment
< >	Delimiters	Variables are enclosed in angled brackets
[]	Optional	Elements in angled brackets are optional.
$\left. \begin{matrix} a \\ b \\ c \end{matrix} \right\} a b c $	Selection	An element can be selected if several elements are arranged vertically in braces or horizontally between vertical lines.
...	Repetition	Ellipses indicate a possible repetition of the previous element.

Abbreviations

A number of abbreviations and short names apply to the entire RMOS3 documentation:

Table 1- 2 Abbreviations

Abbreviation	Meaning	Text passage
cfg	configurable	System calls
uintmax	maximum unsigned integer (FFFF FFFFH)	System calls
dp	data pointer (sel:off) 48-bit pointer	System calls

Data types

The following data types may be used for the C compilers approved for RMOS3:

Table 1- 3 General data types

Data type	Data length
char	8 bit
short	16 bit
int	32 bit
long	32 bit
long long ¹	64 bit
void _FAR * ²	48 bit
void _NEAR * ²	32 bit
enum	32 bit
float	32 bit
double	64 bit

1 Only available for GNU programs.

2 Pointers in flat programs always have a length of 32 bit. No distinction is made between NEAR and FAR.

Table 1- 4 Data types of the RMOS3 API

Name	Data type	Data length
uchar	unsigned char	8 bit
ushort	unsigned short	16 bit
uint	unsigned int	32 bit
ulong	unsigned long	32 bit
rmfarproc ¹	Pointer to function type void _FAR f(void)	48 bit
rmnearproc ¹	Pointer to function type void _NEAR f(void)	32 bit
rmproc ¹	Pointer to function type void _NEAR f(void)	32 bit

1 Pointers in flat programs always have a length of 32 bit. No distinction is made between NEAR and FAR.

RMOS3 introduction

Optimizing software development

It is a commonly accepted fact that software costs represent a key factor with regard to the costs incurred for the development, expansion, and maintenance of microprocessor controlled products. The increasing complexity of the machinery or plant control systems to be implemented, as well as the continuous cut on time for development for exactly the projects impose additional pressure on developers. The decision as to whether to handle all developments in-house or whether to use turnkey modules plays a key role.

In many situations, the only chance that you have in terms of fast and cost-effective implementation of your application is to optimize the use of standardized hardware and software modules.

RMOS3 provides you with a fundamental, basic software module. Thanks to the diversity of this **Realtime Multitasking Operating System (= RMOS)**, the development, servicing and maintenance tasks for your software on PC/AT compatible systems are significantly accelerated and simplified. The fundamental properties of RMOS3 can be derived directly from the term "real-time multitasking operating system".

Real-time

In the following, the term real-time should be interpreted as the capability of a system to complete specific functions at all conditions and within a defined period.

The automation of technical processes, for example, imposes typical demands on the real-time behavior of the computer system.

- Many tasks such as digital control algorithms demand cyclic processing, whereby certain algorithms show a highly sensitive response to cycle time fluctuation.
- A defined reaction to asynchronous events is indispensable for processing sequential controls and the reception of limit/position feedback messages.
- Certain tasks must be at absolute times or require absolute times (e.g., trend and logging information).

These requirements can only be fulfilled by a real-time operating system that never blocks the handling of external events for a duration longer than absolutely necessary (microseconds range).

Multitasking

Multitasking capability represents a further important requirement to be fulfilled by an operating system. Most technical processes can be split into multiple, linked subprocesses running in parallel. This makes it extremely difficult and, in certain scenarios even impossible, to maintain a clear of overview of these processes and to control them using a sequential program. A practical solution for implementing multitasking is to use a greater number of programs (self-contained programs = **Tasks**) that run in parallel in a mono-processor or multiprocessor environment.

Operating system

The following technical definition of the tasks of an operating system is derived from external references:

"The operating system provides the utilities that are necessary for the programming and user interfaces, which do not directly exist on the computer. Compared to the actual computer, it offers an interface that is more suitable for usage and utilization."

These are satisfactory basic properties that enable you to realize important tasks, for example, for measuring, controlling, regulating, or parameter assignment. However, because machine control systems and industrial production processes have gained complexity, they are more sensitive to incorrect operations that pose the risk of incurring very high costs through production downtimes.

The operating and monitoring functions play a key role under this aspect. A convenient human machine interface (as of herewith **HMI**) with optimized, precise process visualization offers far-reaching opportunities for guiding operators and helps to avoid incorrect operations. Moreover, the HMI provides optimum support for operator in normal operation, because it is a known fact that one picture is worth more than 1000 figures, for example, when it comes to the qualitative assessment of pressure or temperature profiles.

PC compatible hardware platforms running on operating systems with graphical user interfaces, e.g., Microsoft Windows (as of herewith **Windows**), have become widely accepted.

Restrictions of Windows in real-time mode

However, the following substantial restrictions clearly speak against using Windows for HMI **and** process control tasks.

- The multitasking control mechanism of Windows is based on a message-controlled allocation process between the various programs (tasks). This method blocks the assignment of defined reaction times, because Windows does not support premature cancellation of a task, for example, to execute a different task that takes higher priority due to an external event.
- The poor real-time capability in Windows, which means sluggish interrupt reaction times that are subject to substantial fluctuation prevents the control software from maintaining synchronism with the process runtime speed at **any** given time. The inevitably consequence are malfunctions and costly downtimes in the production process.

RMOS3

RMOS3 offers the following solutions to bypass these restrictions:

1. Process control and the HMI are implemented **on the same PC**. You need the separately available product RMOS3-GRAPHX to implement the graphical user interface.
2. Process control and the HMI are implemented on **separate PCs** that are interconnected by means of suitable communication equipment. RMOS3 is employed on the process control PC to control all hardware resources by means of application tasks. The separate PC for the HMI operates on a Windows platform.
Section "Process control with separate HMI (Page 17)" outlines the fundamental properties of this solution.

Section "RMOS3 performance features (Page 19)" outlines the essential performance features.

This section is concluded with a presentation of the general fields of application for RMOS3, an overview of RMOS3 expansion components, and a summary.

2.1 Process control with separate HMI

Separate PCs

The figure "Process control with separate HMI" shows the hardware or software structure of a typical automation task with the process control system and HMI being implemented on **separate PCs**.

Process control side

The alarm clock on the process control side characterizes the key requirement for this computer, namely the necessary **on-time** reaction of the entire software (operating system, including control programs) in all operating states.

Figuratively speaking, this means that we always need to set the alarm clock so that we get up on time, depending on whether we have hard (take-off time at the airport) or soft (floating working hours) scheduled real-time requirements.

2.1 Process control with separate HMI

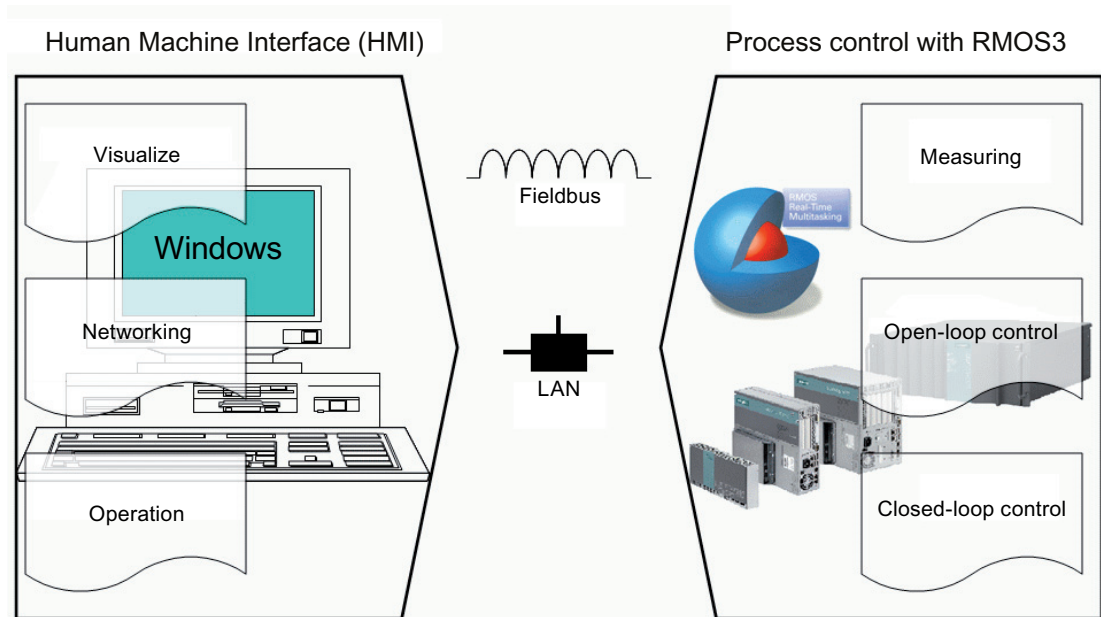


Figure 2-1 Process control with separate HMI

The yardstick for the demanded real-time capability is therefore formed by the automation task and not by the operating system. A good example is a car's automatic braking system that has substantially greater demands on immediate system reaction compared to the automatic air-conditioning system.

Communication process HMI

The process control system and HMI units implemented on separate computers need to communicate as a consequence. The following figure outlines this requirement based on the fieldbus or LAN (Local Area Network) components.

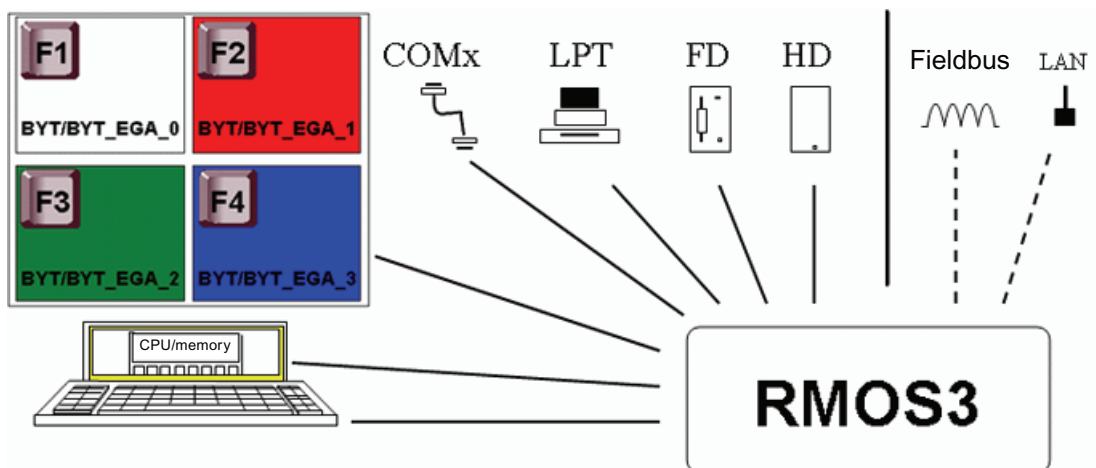


Figure 2-2 Process control with RMOS3

Systems without operator intervention

The figure "Process control with RMOS3" emphasizes that operation on the RMOS3 platform involves process control

- of all hardware resources of the PC by means of the real-time multitasking operating system or application tasks, and
- that additionally required hardware resources (e.g., networking) can only be used with software components that are executable in RMOS3.

On systems without operator intervention or less complex demands on the HMI, you can implement all tasks in RMOS3 and dispense with the separate PC for HMI functions. A side effect is that this can save you additional license fees for DOS/Windows operating systems or programs.

The PC screen is operated in RMOS3 in semi-graphics mode, which means character-oriented. Tasks can only achieve full graphic visualization by writing the corresponding data directly to the memory of the PC screen.

Reaction times

A multitasking control mechanism with defined reaction times and successful handling of "most stringent" real-time demands with interrupt blocking times of max. 2 microseconds (CPU Intel Core™ 2, 2.166 GHz) always ensures that program execution is in line with the process.

Networking (TCP/IP)

RMOS3 already contains drivers for networking over TCP/IP. Provided the CPU supports such functionality, it is also possible to implement remote control tasks for accessing distributed computers and transferring data over FTP (File Transfer Protocol) using the Telnet and Telnet Daemon utility programs.

2.2 RMOS3 performance features

This chapter will now go into more detail with respect to the RMOS3 performance features.

The real-time multitasking mechanisms of the RMOS3 nucleus are explained in this chapter (sections "Multitasking and multiprocessing mechanisms (Page 20)", "Real-time behavior and suitability for industrial applications (Page 23)", "Boot sequence (Page 27)", and "Performance scope of the RMOS3 nucleus (Page 27)").

It also elaborates on hardware requirements, development environments (section "Hardware requirements and development environment (Page 30)"), on test strategies (section "Test strategies (Page 30)"), and on the general overview of the RMOS3 product (section "Product overview (Page 32)").

For information on the RMOS3 variants included with your package and on general fields of application for RMOS3 refer to sections "Configurable nucleus (Page 26)" and "General fields of application for RMOS3 (Page 34)".

2.2.1 Multitasking and multiprocessing mechanisms

Several programs running simultaneously

Multitasking means that a system containing a single CPU is capable of running two or more programs (tasks) in quasi-parallel mode. This scenario is controlled in RMOS3 by means of a task change mechanism (as of herewith **scheduler**) that is oriented on an absolute ID number, namely the **priority**. All tasks are started with the priority you specified, with priority ranging from 0 (lowest) to 255 (top priority). These settings can be adjusted dynamically during runtime.

Tasks are capable of running in true simultaneous mode if the system contains several processor cores (multiprocessing system). The scheduler allocates the tasks to the different cores. A system containing only one processor core represents a special case of a multiprocessing system. This chapter only presents this special case in order to simplify matters.

The following figure shows a simplified diagram of different tasks having different priorities.

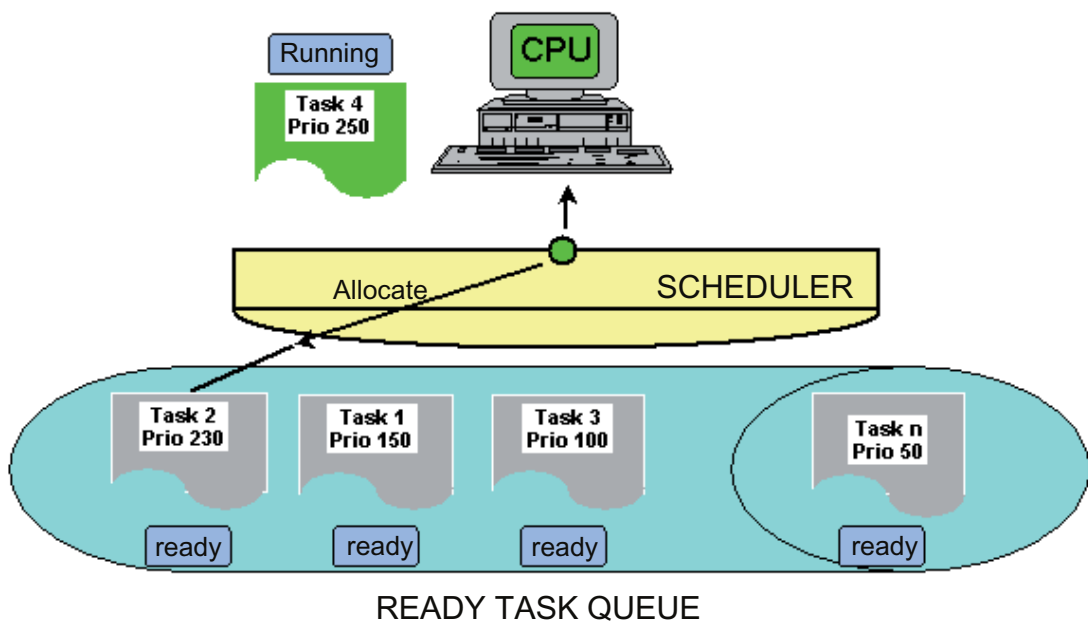


Figure 2-3 Multitasking principle in RMOS3

Scheduling

The operating system (scheduler) will now allocate the CPU to the respective task that is ready for computing and assigned the highest priority of all tasks in ready state. In multiprocessing systems, the scheduler allocates a core to the task that is ready for computing and assigned the highest priority of all tasks that are in ready state and capable of running on this core.

In comparison to programs in the Windows environment that can only be interrupted after they **themselves** have passed control to the CPU, RMOS3 also supports the premature (preemptive) interruption of a currently active task:

- By a task that has just entered the ready state and has higher priority
- By I/O interrupts
- By device driver programs

This property is a fundamental prerequisite for completing a process control task, which may also be implemented with several tasks within a defined period and at any condition.

Task states

A good multitasking mechanism is characterized by its capability of being controlled in a very fast and convenient way. The RMOS3 task organization shown in the following figure conforms to this characteristic.

Because it is not possible to process all tasks simultaneously, each task is always set to one of the following five states.

- **RUNNING**
The CPU was allocated to this task; the program code is executed.
- **READY**
The task is ready for computing and waits for allocation of the CPU.
- **BLOCKED**
The task is awaiting an event or point in time. The operating system resets the task to the READY state after the event has been triggered or the waiting time has expired.
- **DORMANT**
The task is in dormant state. Its existence is known within the operating system. It can be started by another task.
- **NON-EXISTENT**
The existence of the task is unknown within the operating system. The program code of the task may already exist in the random access memory, or may be loaded as file from a mass storage medium.

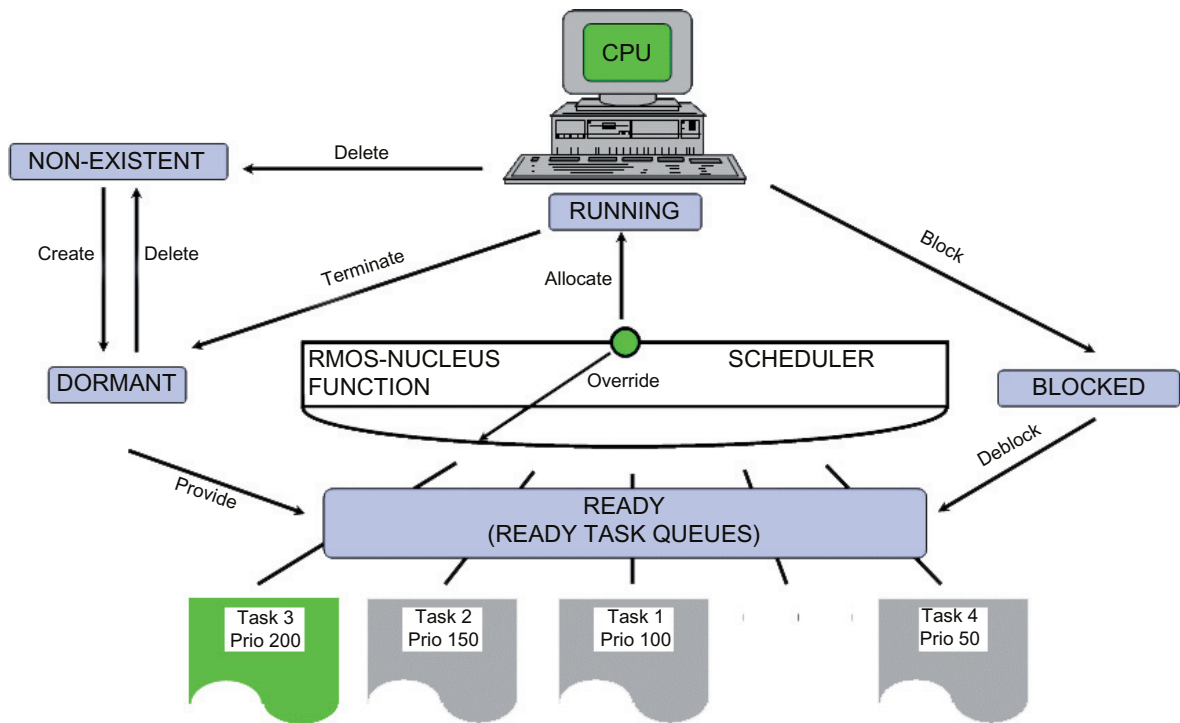


Figure 2-4 RMOS3 task status graph

State change

The status transitions are initiated by explicit system calls, external events (interrupts) or by the scheduler.

Tasks can be redefined or deleted during system runtime with the status transitions **Create** and **Delete**. You can always load tasks to enhance system maintenance by activating diagnostics or update software as file from a mass storage medium.

The **Providing** status transition is executed by

- the automatic start of your specified initialization task after operating system startup
- a "Start task" system call
- a start request triggered by an interrupt.

The RMOS3 scheduler controls the **Allocate** or **Override** status transitions. The CPU is allocated to the top priority task on completion of all operating system activities.

An override of the computing task is triggered, for example,

- after a system call of the task
- if a task of higher priority enters the "ready" state
- after an interrupt routine for operation of an I/O device has switched to the so-called "S state (figure "Operating states of real-time interrupts").

Typical reasons for the **Blocking** transition are the output of system calls with wait condition, for example, "Wait for completion of an I/O function", by the computing task, or "Wait for expiration of a **pause**". Likewise, the operating system executes the **Deblock** status transition on expiration of the wait condition.

A computing task can use the system calls

- to end a task
- to end a task and restart on expiration of a time interval
- initiate the **terminate** status transition.

2.2.2 Real-time behavior and suitability for industrial applications

Fast reaction to events

In most microprocessor applications, high-speed reaction to external events, which are mapped to hardware signals (interrupts) of the processor in a real-time system, is attached greatest importance. Primarily its lacking real-time capability speaks against the use of Windows in process-oriented automation engineering.

It is fact, for example, that disk operations or mouse movements in Windows can block the control system for a duration of several seconds. Although the interrupt requests will be registered at system level, it takes a long time for these to reach the control program in the aforementioned scenario.

Interrupt handling

RMOS3 is responsible for handling the interrupt control functions. All interrupts are managed by means of a central monitoring routine that calls the RMOS3 interrupt routine, depending on the cause of the interrupt. The respective RMOS3 interrupt routines are stored in an interrupt descriptor table and executed on receipt of an interrupt.

Interrupt control in four operating states

The figure "Operating states of real-time interrupts" emphasizes the operating states that are available for handling interrupts.

DI state	Disabled Interrupt state	All interrupts are disabled.
I state	Interrupt state	Interrupts of lower priority compared to the currently processed are blocked. SVCs can be called.
S state	System state	All interrupts are enabled and it is possible to output non-blocking operating system calls.
A state	Application state	All interrupts are enabled; operating system calls can be output.

The internal `x_elipse`, `x_systate` and `x_xel` system routines automatically execute status transitions.

SVC calls during interrupt handling

SVCs may also be called in the I and S states. Observe the following aspects:

The SVCs called may not lead to the BLOCKED state, which means it is not permitted to wait for messages, or test and set semaphores. Task ID 0 is output if an SVC exception was triggered by the call of the SVC from an interrupt handler.

Note that in **I state** an SVC is not executed until the I state is terminated. This means that the result of an SVC is not available. You should therefore refrain from using SVCs other than those which report certain operations to the operating system, for example, start task, set flag, or reset semaphore. Moreover, the stack may only contain the SVC parameters. This means that it is not possible to write a message to the stack and then pass a pointer to this message. In I state, it is not permitted to use `SVCRmIO` to call drivers.

In **S state**, the use of `SVCRmIO` is only permitted without wait function.

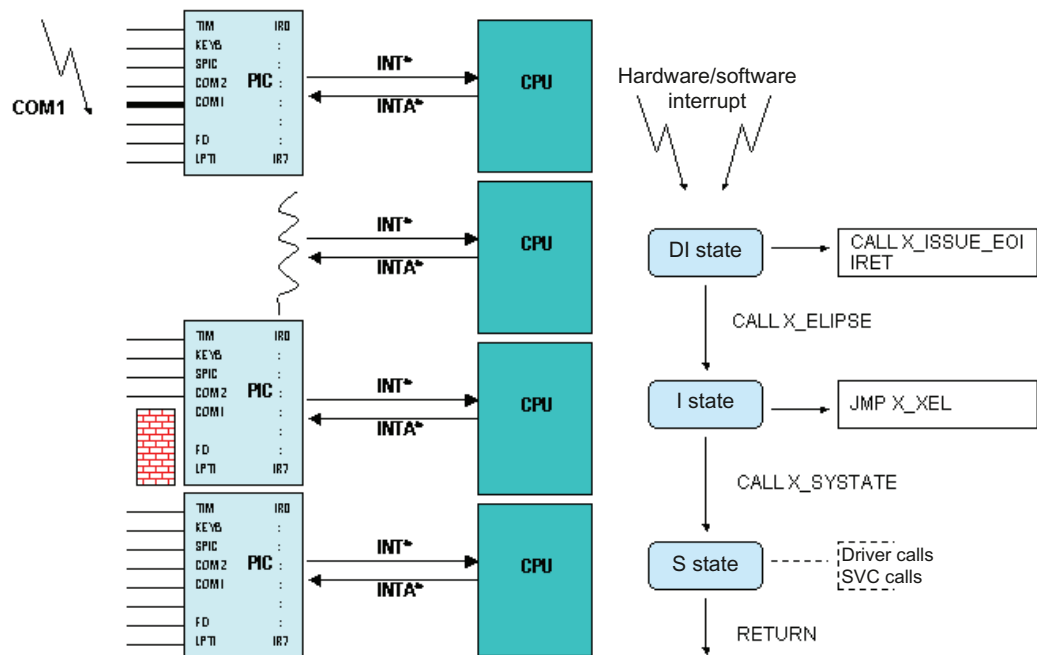


Figure 2-5 Operating states of real-time interrupts

On occurrence of a hardware or software interrupt, the routine will first branch to the **disable interrupt state** or to the **interrupt state** (I or DI state). In these states, the CPU executes driver programs or interrupt service routines of the application.

On exit of the DI or I state, the system resumes the program (task, system call, or interrupted ISR) that was active at the time of interrupt triggering.

A status transition from the DI to I state or from the I to S state can be initiated by calling interface routines.

The Interrupt Service Routines (ISR) are coded directly in the high-level language (e.g., in C) and may be installed, activated or replaced dynamically by means of procedure declaration without problems. It is therefore generally not necessary to write a special driver that conforms to RMOS3 for handling the real-time interrupt.

For interrupts taking a slightly longer time slice, you can also choose between the

- "start task by means of hardware/software interrupt" and
- "send a message by means of hardware/software interrupt"

services.

In **System** state (S state), the CPU executes RMOS3 system calls, programs for operating I/O devices (drivers), or interrupt service routines of the application. Completion of all activities in S state is conditional for the status transition to A state.

The program code of a task is executed in **Application** state (A state).

Protective mechanisms in RMOS3

A vital aspect of RMOS3 **suitability for industrial applications** is the support of the 32-bit protected mode (as of 80386). In this mode, all tasks are protected against each other. A task violating the data, stack, or program area triggers an error detection by the operating system and output of a corresponding error message that indicates the nature and cause of the detected processor exception.

Memory protection for GNU programs is realized by two different privilege levels.

2.2.3 Configurable nucleus

The RMOS3 nucleus (PC_CNUC) that can be configured by means of ASCII file is included by default in the scope of delivery. This nucleus is completely pre-configured for SIMATIC PCs. It offers the following functions:

- VGA interface
- Keyboard
- Floppy disk drives and HDs
- 2 MB RAM disk
- Management of up to 16 MB RAM
- Two serial interfaces with pre-configured BYT driver
- Extended detection of RAM exceeding 16 MB
- LAN support
- Support for hard disks on the secondary IDE controller
- Completely configurable using the ASCII file RMOS.INI
- Support for project-specific identification of the nucleus
- Automatic start of reloadable applications
- Automatic start of batch files
- Logging of startup messages with backup of the old LOG file
- Logging of exception messages of the nucleus to a LOG file
- System flag group for service purposes
- Configurable interfaces (BYT driver, 3964(R) driver)
- Configurable SRAM disk
- Automatically activated PCI interrupt sharing
- Configurable LAN controllers
- Support for LAN services, such as FTP daemons or Telnet daemons
- Programming of the CPU LEDs
- Version management for the RMOS3 nucleus and configuration task CNFTSK.386

This nucleus therefore offers a wide range of application options and is immediately available for use on the target system without recompilation of the RMOS3 system. The respective up-to-date version of the nucleus is stored in directory RM3DEV\SYSTEM\PC_CNUC. Older versions are stored in the directories RM3DEV\SYSTEM\PC_CNUC<version>. These directories also contain installation scripts for commissioning the nucleus on the target system.

The nucleus is available in two boot variants (see chapter "Boot sequence and memory allocation (Page 159)"):

- Bootstrapping of the RMOS3 nucleus by means of second stage boot loader RMLDR
- Loading of the RMOS3 nucleus in MS-DOS by means of LOADX.

2.2.4 Boot sequence

Start-up of the RMOS3 system

The RMOS3 nucleus can be loaded directly from hard disk or Compact Flash Card. On completion of the operating system start-up, all hardware resources of the PC will be available under control of the RMOS3 system or application tasks.

The PC screen is operated in RMOS3 in semi-graphics mode, which means character-oriented.

The RMOS3 file management system is capable of multitasking, manages the mass storage media resources in DOS format and therefore supports the convenient exchange of data volumes between RMOS3 and DOS systems.

in contrast to the DOS file management system, RMOS3 also supports quasi-parallel file operations, for example, simultaneous read/write access to files on the floppy/hard disk. The advantage is here, for example, that it is possible to run simultaneous file operations of real-time tasks on different mass storage media (hard disk, Compact Flash Card) while formatting a floppy disk.

2.2.5 Performance scope of the RMOS3 nucleus

RMOS3 nucleus

In addition to the previously mentioned multitasking, multiprocessing, real-time and communication mechanisms, the real-time multitasking RMOS3 nucleus offers a manageable (appropriate) number of system services for task and resource management, as well as for task coordination and communication.

The use of the configurable RMOS3 nucleus allows you to rely on tested operating system variants and to integrate, configure, and activate the necessary drivers by means of a configuration file (RMOS.INI) without you having to generate a nucleus. This means that you can immediately launch application development.

The network support included in the standard scope of delivery offers you diverse options, for example, the exchange of files with the target system by means of File Transfer Protocol (FTP), or remote debugging and operation of the RMOS3 target system via Telnet.

Resources

All resources may be declared statically and/or dynamically, which means that they are defined dynamically at runtime or statically in your configuration. Dynamically defined resources can also be deleted.

The following table shows an overview of the system utilities and resources provided.

Table 2- 1 RMOS3 system utilities, resources overview

TASK management				
TASK	CREATE DELETE LOAD	TASK	START/END STOP/RELEASE START BY INTERRUPT	TASK PRIORITY CONTROL BIND TASK
		STOP/ENABLE SCHEDULING		TASK STATUS QUERY
TASK COORDINATION AND COMMUNICATION				
LOCAL MESSAGE TRAFFIC	EVENT FLAGS	SEMAPHORES	MESSAGES	SPINLOCKS
RESOURCE MANAGEMENT				
TIME MANAGEMENT	I/O MANAGEMENT	MEMORY MANAGEMENT	MANAGEMENT OF LOGICAL RESOURCE NAMES	

Task management

Task management offers you in essence the following utilities:

- Create, delete, load, start, and end tasks, and cyclic start of tasks.
- Explicit revocation of the pause state of a different task.
- Explicit stop or release of the RMOS3 scheduler.
- Start the task on detection of a hardware or software interrupt.
- Dynamic change of priorities (internal or external task).
- Status request (internal or external task).

Task communication and coordination

RMOS3 facilitates task communication based on local message traffic and event flags. Message traffic is based on so-called mailboxes that can be used for computing tasks to send or receive messages. This means that the tasks can select whether to wait until their transmitted message has been received or whether to wait for an incoming message when receiving.

A message queue can be generated for each task. It is a FIFO memory to which other tasks can copy message pointers. Each message requires at least one 16-bit parameter (the message number or code) and a message-specific 32-bit parameter (for example, to be used as pointer to a message block).

Event flags are implemented by means of logical bit units and are collected in event flag groups. Computing tasks can

- immediately or on expiration of a specific time, set
- reset, or
- test

one or several event flags. It is also possible to wait until one of several event flags have been set.

Coordination of mutual exclusion of tasks with simultaneous access to shared resources (data structures, I/O blocks, etc.) is controlled by means of semaphore and spinlock calls.

Resource management

Resource management provides RMOS3 system calls were the management of time, I/O devices, and memory space.

Time management is based on the clock signal of a timer block. The system clock rate can be set in millisecond steps starting at 1 ms. In addition to time related system utilities (cyclic restart, pause, timeout monitoring, etc.), RMOS3 maintains an internal software clock that implements the date and time functions (resolution in ms). You may use custom system calls to set and query the software clock, as well as the hardware clock on your PCs.

RMOS3 controls the I/O functions for many distributed devices by means of device drivers. A task can execute a system call to transfer data input/output (texts) to a driver. Along with the call, the task can decide whether to wait for completion of the job, or whether to wait for a done confirmation returned by means of an event flag.

In memory management, tasks can request/return contiguous memory space from/to memory pools. This enables tasks to dynamically request memory space they need, for example, to record or analyze measured values. An infrequently executed task that is otherwise forced to statically reserve the entire data area could possibly have a negative impact on memory utilization.

It is also possible to employ commonly used memory space, e.g., shared memory.

Logical names of resources

The management of logical resource names facilitates porting your application software to other systems. For this purpose, you may assign a string with a length of up to 15 characters to each resource (e.g., task, mailbox, event flag, etc.). You can execute a system call to write these strings, including the associated resource IDs, to a directory that is also known as resource catalog. Before other tasks use these resources, they query the existence of the requested resource and its current ID in the system based on these character strings.

2.2.6 Hardware requirements and development environment

Development system

You may run the development system on any HW platform with Windows operating system. You may optionally employ TCP/IP for data transmission over FTP or for remote diagnostics and operation of the RMOS3 system via Telnet.

You need a CD-ROM drive to install the RMOS3 development environment.

Target system

Hardware requirement for the target system of RMOS3 is a PC compatible SIMATIC hardware with Pentium III or later generation processor.

The recommended memory configuration depends on the operating system variant used and should not be less than 2 MB.

The configurable nucleus offers an operating system with command line-oriented user interface for the control and visualization software, as the RMOS3 tasks can be loaded/started dynamically and all resources can be generated at runtime as mentioned earlier.

Applications

The examples included demonstrate the operating principle and performance scope of RMOS3 and usually form the basis for custom applications.

RMOS3 tasks are developed on the Windows platform using the GNU Crosscompiler `rm-gcc` and loaded dynamically at runtime by means of RMOS3 task loader (dynamic tasks).

2.2.7 Test strategies

Debugger

The HOST/TARGET-oriented debugger concept (see figure below) was implemented for the RMOS3 tasks based on the following organization:

- Low-level (LL) debugger for the target system
- High-level language (HLL) user interface for debugging on the development system (Windows supports the `rm-gdb` from the RMOS3 GNU software package)
- Communication between LL and HLL debuggers over TCP/IP.

The low-level debugger is capable of performing the following actions, for example:

- Handling the sequential control and checking the status of all tasks working in RMOS3
- Interactive output of system calls
- Verification and modification of memory contents
- Setting breakpoints in user tasks.

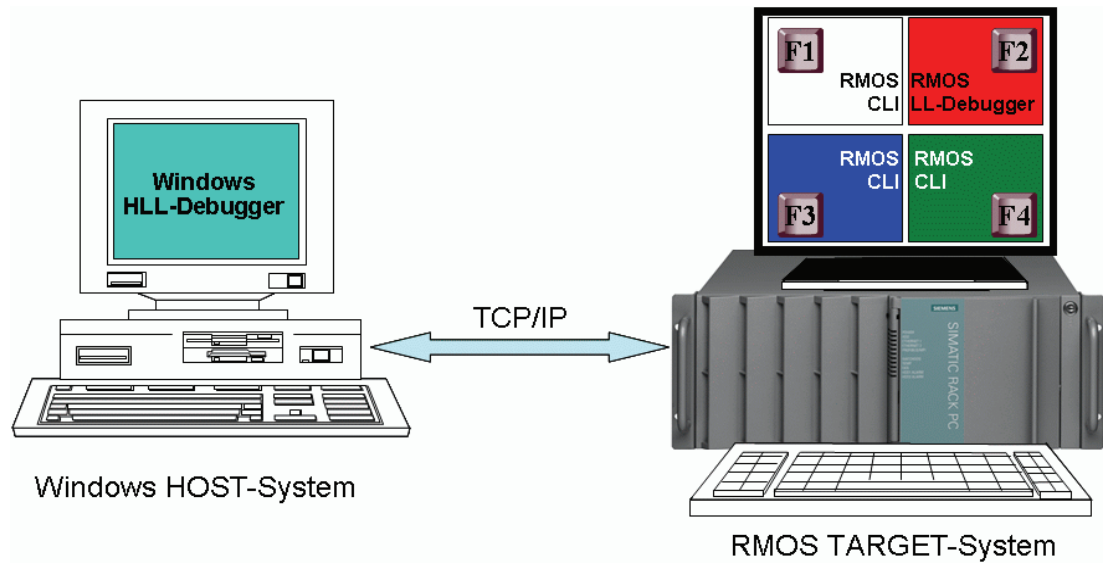


Figure 2-6 Host-/target-oriented debugging concept

Features of the HLL debugger

Essential features of the HLL debug user interface include:

- Window-oriented user interface with pull-down menus, mouse operation, and macro support.
- Debugging on high-language level based on program listings.
- Access to all program symbols and data structures (including stack-based) such as multi-dimensional arrays, structures, or bit arrays.
- Loading and symbolic debugging of relocatable programs.

Resource reporter

Resource reporter is a useful task that supplements the debugger in the testing phase. With the help of this task it is possible to output snapshots of RMOS3 data structures and resources to the screen.

The utility comprises functions for evaluation of tasks, devices drivers, memory pools, semaphores, local and global event flags, message queues, and mailboxes. You may select the short or detailed version for visualization of the data.

2.2.8 Product overview

The following figure shows the aforementioned development tools in the operating system structure of RMOS3.

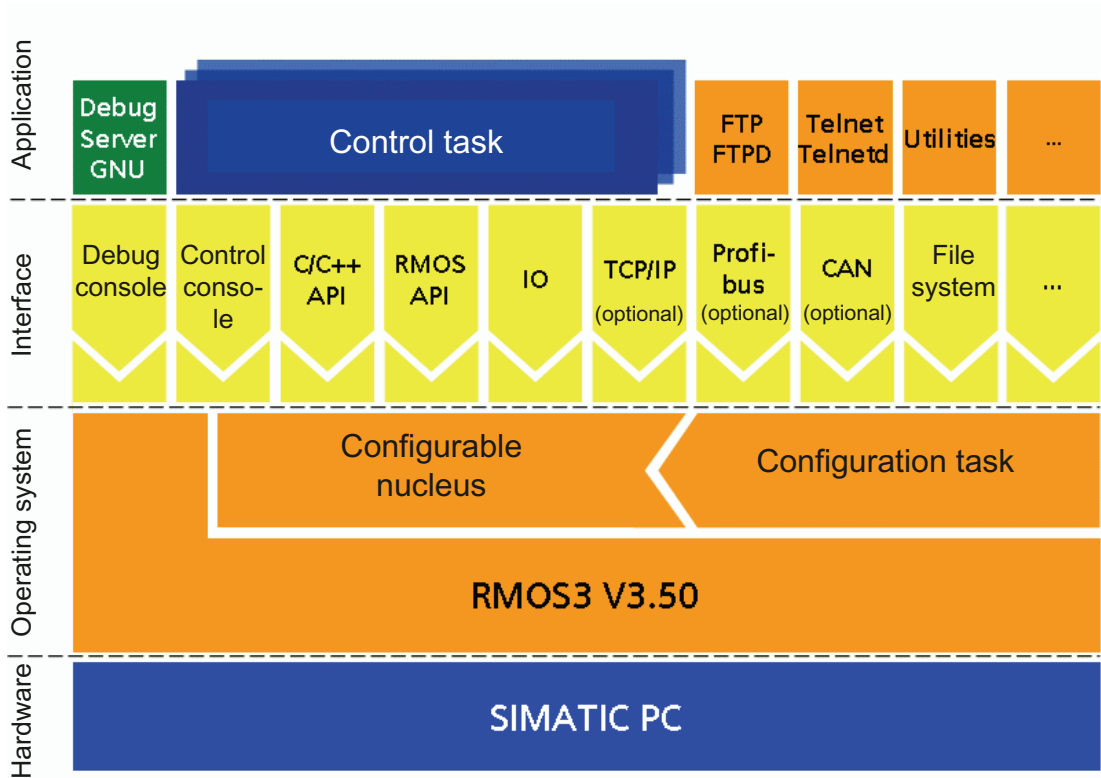


Figure 2-7 Operating system structure of RMOS3

The following components emphasize the innovative character of RMOS3, for example:

- Symmetrical multiprocessing,
- Memory protection
- ANSI-C Runtime Library
- Command Line Interpreter (CLI),
- RMOS3 Profiler for runtime analysis of the RMOS3 system
- LAN interface with TCP/IP stack
- FTP and Telnet utilities
- PCI interface with interrupt sharing
- Advanced Programmable Interrupt Controller (APIC)
- High Precision Event Timer (HPET)
- File manager

ANSI-C Runtime Library

The C runtime library **RMOS3-CRUN** provides all C functions in the RMOS3 multitasking environment in accordance with ANSI Draft Standard ISO/IEC DIS 9899 (published 1990). The runtime library can be used as "shared library" and includes the following performance features.

- Character management, string and memory operations
- I/O operations, e.g., HD, terminal, printer
- Memory space allocation
- Mathematical functions, e.g., `sin`, `cos`, etc.
- Control functions and error handling.

Command line interpreter CLI

The CLI represents a user interface for the RMOS3 operating system. The CLI can be used to execute commands and to start and interactively operate reloadable programs.

The CLI consists in essence of the following components:

- CLI task (shell); you can activate it dynamically multiple times
- Program manager for active CLI commands
- In-line commands, e.g., `path`, `prompt`, `cd`, etc.
- Loadable commands, e.g., `dir`, `rename`, etc.
- Runtime environment for reloadable tasks; e.g., definition of `stdin`, `stdout`, `stderr` and pass of the call parameters
- CLI startup batch file(s) similar to `AUTOEXEC.BAT`

The command syntax is kept widely compatible with DOS and a corresponding `HELP` command helps PC users to overcome any operating problem.

RMOS3 profiler

The RMOS3 profiler is a utility that provides support for the following activities:

- Displaying system parameters
- Calculating load distribution
- Analyzing task activities.

You may use the results of the load distribution calculation to optimize system runtime. It is helpful for error handling and runtime optimization to log the task activities. It may also be used to measure SVC execution times and specific program units.

LAN and TCP/IP

The configurable RMOS3 nucleus comes prepared for operation on a TCP/IP network. The necessary LAN drivers and TCP/IP stack are loaded dynamically at start-up and can be customized or replaced conveniently. IP addresses can be entered in configuration file RMOS.INI.

FTP and Telnet

The FTP/FTPD and Telnet/TelnetD utilities included with your scope of delivery facilitate file transfer via network. You can immediately run software updates on the target systems and generate backup copies of the measurement data from distributed systems. Remote maintenance is also facilitated via Telnet.

PCI interface and interrupt sharing

State-of-the-art bus systems, such as the PCI bus, are supported via RMOS3 API. The API facilitates explicit scanning of the PCI bus to identify inserted I/O modules, even across bridges. The parameters of the PCI modules found are written to special data structures for further processing.

Moreover, the multiple use of interrupts on the PCI bus is also supported. This will become necessary if several modules and devices on the PCI bus share the same interrupt.

File manager

The file manager presents a tree structure of the file system on a semi-graphical user interface. Files can be copied, moved, deleted and edited. It is also possible to create and delete directories.

2.2.9 General fields of application for RMOS3

The modular structure of RMOS3 makes it possible to customize the performance scope and therefore memory requirements to suit customer-specific requirements.

RMOS3 provides a flexible real-time platform for PC-compatible architectures that include small embedded applications and ultra-complex automation systems.

Reaction time

The standard execution time for an operating system call (including the re-scheduling time) is less than one microsecond with Intel Core™ 2, 2.166 GHz.

2.3 Summary

Controlled allocation and co-ordination of resources

The introduction of high-level programming languages has made it possible to design structured programs. However, the controlled allocation and corresponding co-ordination of resources such as memory, processor time, or I/O units is impossible without a real-time operating system that is suitable for the respective area of application.

General advantages in using a real-time operating system

- Segmentation of an application to form individual comprehensive program elements (= tasks) can simplify implementation, testing and maintenance of the application system.
- The modular approach supports reuse of the software, which means the "porting" of tasks to different projects.
- Support for the "quasi parallel" sequence of tasks.
- Deterministic behavior through priority-controlled multitasking and defined interrupt reaction times.
- Because the operating system itself provides solutions for the special requirements of a real-time multitasking runtime environment, it is possible to substantially reduce the time (and, of course, costs) required for developing and testing the application.

Optimizing software development

It is a commonly accepted fact that **software costs** represent a key factor with regard to the costs incurred for the development, expansion, and maintenance of microprocessor controlled products.

The increasing complexity of the machinery or plant control systems to be realized, as well as the continuous cut on time for development for exactly the projects impose additional pressure on developers.

The decision as to whether to handle all developments in-house or whether to use turnkey modules plays a key role.

In many situations, the only chance that you have in terms of fast and cost-effective implementation of your application is to optimize the use of standardized hardware and software modules.

RMOS3 = fundamental basic software block

RMOS3 provides an **elementary basic software block** in the form of a real-time multitasking operating system, featuring a diversity that considerably accelerates and simplifies development, servicing and maintenance of the application software.

Motivation for using RMOS3

- Avoidance of high recurrent costs incurred by the development and management of software elements that are not directly relevant to the actual application.
- Simple portability of existing, fully tested software modules to other projects. Hardware-dependent code can be stored in separate modules. Only these need to be adapted for porting.

What is the meaning of the term RMOS3?

"RMOS3" denotes the
"Realtime Multitasking Operating System
for x86 processors as of 80386

Realtime

Real-time

Multitasking

Multitasking

Operating System

Operating system

for 80386 platforms

for x86 processors as of the 80386 series

Embedded systems

Embedded systems are characterized by an overall system containing computers that are not necessarily identified as typical computers (PCs) by external viewers.

There are significant differences in embedded system with regard to the performance of computer components used.

SIMATIC PC

A typical example of high-performance **PC-compatible** embedded systems (automation computers) that are suitable for industrial applications is the SIMATIC Microbox PC that features high-end CPUs and several megabyte RAM, external storage media, network integration, etc.

However, along with the minimization of hardware costs, the focus must be set on the deployment of **efficient methods for software development** that are especially tailored for hardware resources.

Software costs have increasingly evolved into being the decisive factor

Because compact embedded systems need to handle tasks under the aspect of rising costs and complexity, it is a common feature of both systems that the costs of development, expansion and maintenance are determined primarily by the **software costs**.

By **using the real-time operating systems RMOS3**, you can expect the following benefits

- significant gains in productivity
- clearly improved maintenance capability
- higher reliability
- a clear structure of the application software
- support for portability and reuse of the applications software.

Performance features

The performance features include in essence

- Multiprocessing
- Memory protection
- Preemptive multitasking
- Real-time capability
- Suitability for industrial applications
- Communication and coordination mechanisms

RMOS3 combines different standards to form an optimum operating system base for many tasks in automation engineering.

Support for portability

Support for portability and reuse of the application software by means of:

- Use of the operating system interfaces, which are as abstract as possible and independent of the application
- Provisions for a user interface that is independent of the hardware and network
- Efficient architecture design: RMOS3 provides scalable and configurable functions for optimizing the architecture to meet the special application requirements.

Advantages

- Substantial savings in software costs and development time.
- Enhancement of software quality
- Standardized software interfaces for applications with different hardware configuration.

Necessary hardware and software

Overview

The hardware and software components you need for working in RMOS3 depend on whether you are going to develop tasks on a development system or whether you want to run RMOS3 on the target system.

This chapter describes the minimum configuration for the pre-configured operating system variant delivered to you, as well as for the development system.

Section "Programming and debugging tools for task development (Page 40)" also informs you of the tools you need for RMOS3 programming.

Development system

You can use different tools with different properties for task and system development.

Target system

The demands of an RMOS3 runtime system on the hardware differ greatly. RMOS3 can be adapted to different hardware in a very flexible way. Altogether, you need the following minimum hardware configuration:

- Processor of the x86 series as of 80386
- Timer block that triggers an interrupt at the processor at cyclic intervals
- Interrupt control block.

3.1 RMOS3 target system

The demands of an RMOS3 runtime system on the hardware differ greatly. RMOS3 can be adapted to different hardware in a very flexible way.

Altogether, you need the following minimum hardware configuration:

- Processor of the x86 series as of 80386
- Timer block that triggers an interrupt at the processor at cyclic intervals
- Interrupt control block.
- 4 MB RAM

Configurable nucleus for RMOS3

The configurable nucleus supports PC systems of the SIMATIC PC product range that meet the following minimum hardware requirements, for example:

- IBM AT/02 or 03 compatible keyboard
- EGA/VGA compatible graphic adapter
- 1.44 MB floppy disk drive
- EIDE hard disk

Additional interfaces, such as USB, PROFINET, or PROFIBUS, are supported by add-on Board Support packages (e.g., BSP-SIMATIC PC).

The configurable nucleus is available in development directory RM3DEV\SYSTEM\PC_CNUC. Chapter "Booting from mass storage media (Page 49)" shows you how to boot RMOS3 from a mass storage medium (setup of the mass storage medium, copying the system data).

3.2 RMOS3 development system

Development system

An RMOS3 development system requires a PC/AT compatible system

- with approximately 40 MB of free space on a mass storage medium for the RMOS3 development system, as well as
- a CD-ROM drive for installation.

You also need storage space for the software packages mentioned in section "Programming and debugging tools for task development (Page 40)".

You may use the SETUP program that is specified in chapter "Installing the development environment (Page 43)" to install the RMOS3 development system on your system.

3.3 Programming and debugging tools for task development

The following tools are suitable for use in task development.

RMOS3 GNU

Development package that consists of

- Assembler
- C/C++ Compiler
- Linker
- Debugger
- Integrated user interface for development

For information on using this development package, refer to the user manual included with your RMOS3 GNU package; this is not covered in this documentation.

3.4 Programming and debugging tools for system generation

Generation of the RMOS3 system is no longer required thanks to the use of the configurable nucleus. For this reason, you should no longer use the tools listed below to generate a system in new developments.

Your selection of a development tool has a direct impact on the properties of the RMOS3 system. In contrast to the programming in memory model Flat, memory protection is obtained in RMOS3 using the hardware of the 80386 processor and a segmented memory model of the compiler.

CAD-UL

Compiler package V401 04 or V415 01 that includes:

- CC386, ANSI-C Cross Optimizing Compiler
- AS386 Structured Macro Cross Assembler
- LINK386 Cross Linker and System Builder
- LIB386 Library Manager

Intel

- C Compiler iC386 V4.5
- Assembler ASM386 V4.0
- Binder BND386 V1.5
- Builder BLD386 V1.6
- Library Manager LIB386 V1.3

Installing RMOS3

Overview

This chapter outlines the steps you need to complete to install the development environment for RMOS3 on a development computer. It includes step-by-step instructions for commissioning an RMOS3 target system. Once you completed this chapter, you will be able to create applications on the development system, to boot a target system with RMOS3, and to transfer applications to the target system and test them there.

4.1 Installing the development environment

Preparation

The development environment is installed on a Windows development system.

A CD-ROM drive is required for the installation.

On successful completion of installation, the development computer contains all files that you need to create applications and generate RMOS3 systems. Sources for adaptation to special hardware, sample applications and programs, utilities, and the RMOS3 documentation are also made available.

4.1.1 Installing the RMOS3 development environment

Step-by-step installation

The RMOS3 development environment is installed on the development computer using a Windows InstallShield. The following chapters describe the various installation steps.

Observe chapter "Update (Page 44)" before you run an update from an earlier version of RMOS3.

You may start directly with chapter "New installation (Page 44)" if installing a new system.

4.1.1.1 Update

Backing up the development environment

You need to specify a new target directory during installation. This step corresponds to a new installation. In this case, you have to adjust your project data accordingly.

Installation

Complete the steps listed in chapter "New installation (Page 44)".

If necessary, copy your project files once again to the updated RMOS3 development tree, while observing the notes in chapters "Installation of the compilers, IDE, and debugger (Page 47)", "Installation of additional products, Board Support Packages (Page 47)", and "Notes on using the RMOS3 utilities (Page 48)".

4.1.1.2 New installation

Step 1

Run SETUP.EXE from the installation CD to start the InstallShield Wizard. The Wizard guides through the Setup of RMOS3.

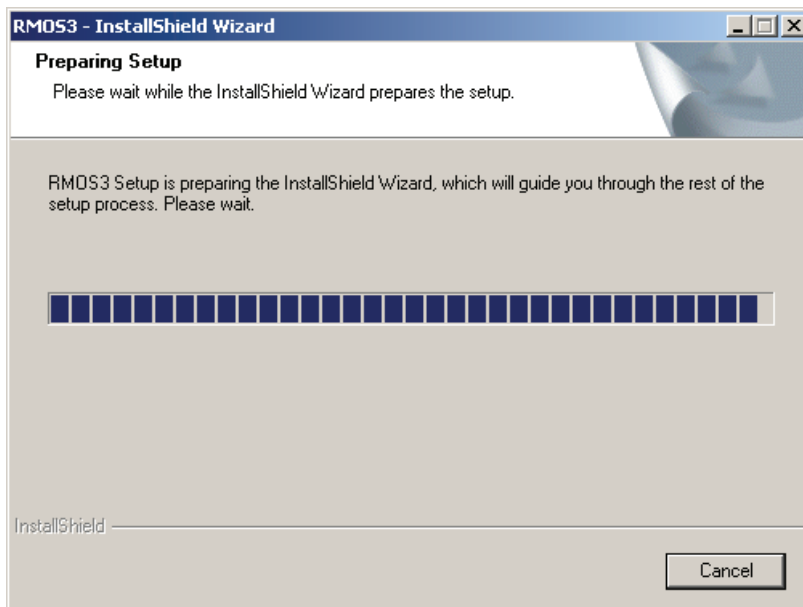


Figure 4-1 Installation preparation

Step 2

Click "Next" to open the page for installation of the RMOS3 development tree.

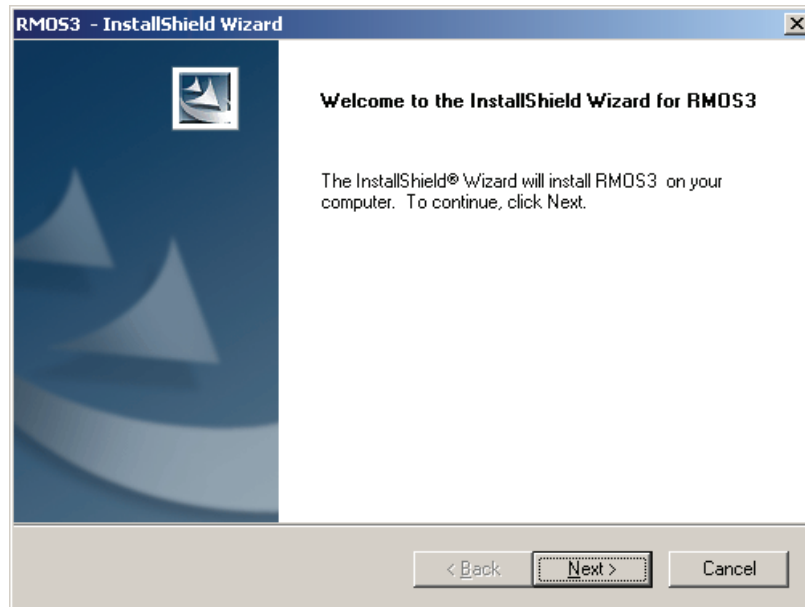


Figure 4-2 Start screen

Step 3

Select the target directory for installing the development environment. Directory C:\RM3DEV is proposed as the default. Click "Browse" if you want to select an alternative directory.

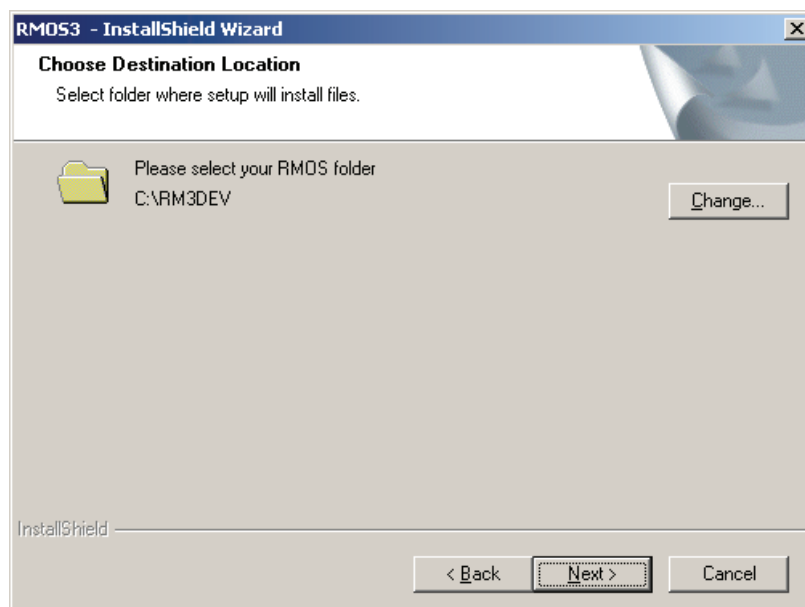


Figure 4-3 Selecting the target directory

Step 4

You are now prompted to set the necessary environment variables. These variables are important for installation of further RMOS3 products.

If you confirm with "Yes", environment variable RMOS_HOME is set. The variable points to the root directory of your RMOS3 installation.

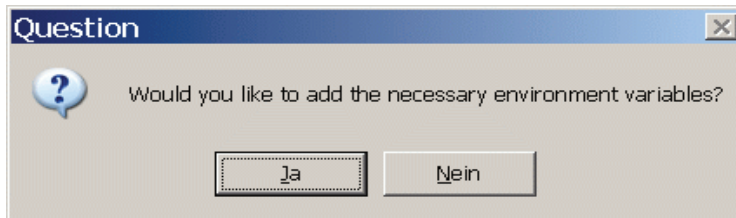


Figure 4-4 Setting the environment variables

Step 5

You are now prompted to select a directory for the start menu to which the links for the RMOS3 documents are saved. Default is SICOMP RMOS3. If you have already installed an older version of RMOS3 and want to prevent this from being overwritten, select a different directory.

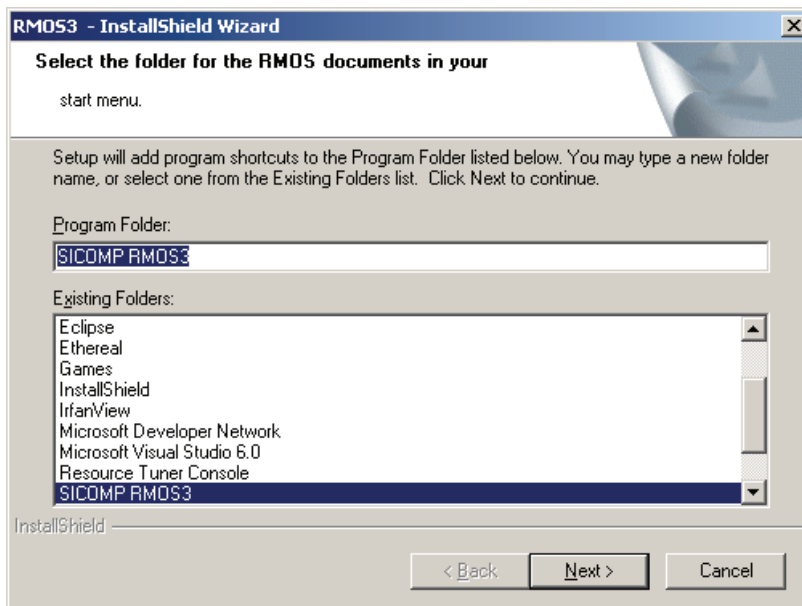


Figure 4-5 Select a directory for the documentation

Step 6

On successful installation of all files, click "Finish" to close the InstallShield Wizard. If the "Show Readme" selection box is activated, an information page is displayed with notes related to the scope of your RMOS3 installation and important information on using RMOS3.

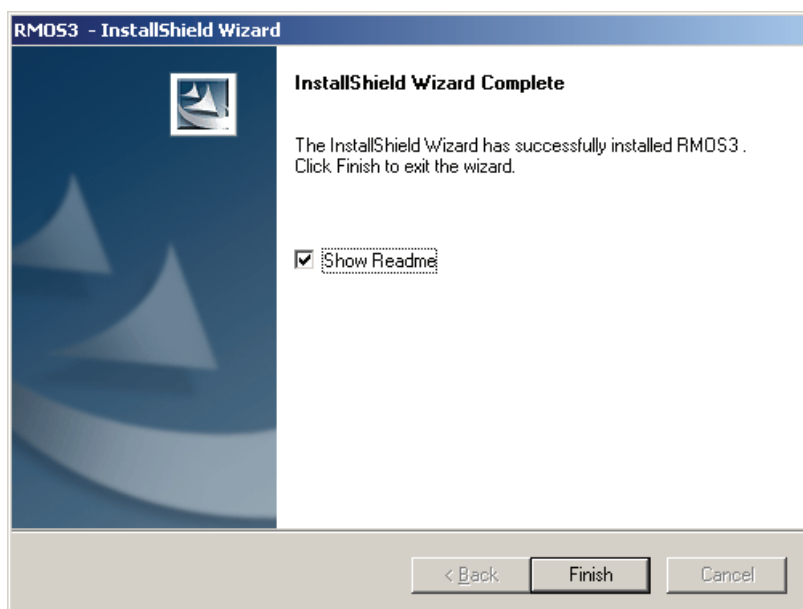


Figure 4-6 End of the installation

You have successfully completed installation of the RMOS3 development environment. You can now install additional Board Support Packages of RMOS3, the compiler, and a graphical user interface for development (IDE).

Installation log file

The installation directory will now contain a log file that contains the name of the installed product and the date of installation.

4.1.2 Installation of the compilers, IDE, and debugger

When installing the compilers, the graphical user interface for your development environment (IDE) and the debugger, observe the instructions for the corresponding products.

4.1.3 Installation of additional products, Board Support Packages

You may need to install further RMOS3 products or Board Support Packages, depending on the target hardware used. You may install these on completion of your installation of the RMOS3 development environment in accordance with the instructions in the respective manuals.

4.1.4 Notes on using the RMOS3 utilities

The RMOS3 development environment includes diverse utilities and the RMOS3 documentation. Observe the following notes when using these utilities:

Documentation

The RM3DEVDOC directory contains the entire RMOS3 documentation in six PDF format manuals. You can use the Start menu entries to access these electronic manuals.

4.2 Commissioning the target system

Preparation

You need a USB Flash Drive that is bootable in DOS or Windows PE.

Boot variants

RMOS3 can generally be started by means of two different variants:

- **Start with RMOS3 boot loader**

The RMOS3 nucleus is loaded directly or by means of RMOS3 boot loader.

An RMOS3 nucleus (RM3_PC1.SYS) of a length less than 608 KB (max. 608 K RAM is available for code) can be loaded below the adapter gap and booted from there.

To load an extensive RMOS3 nucleus of a length greater than 608 KB, use the second stage boot loader RMLDR to load and start the operating system code above the 1 MB gap.

- **Start with MS-DOS boot loader**

First boots MS-DOS and then loads the RMOS3 nucleus by means of special boot loader (LOADX.EXE). Once this routine has been completed, MS-DOS is no longer available.

Booting from different boot media on the target system

You can launch RMOS3 from different boot media:

- Hard disk
- Solid State Disk
- Compact Flash Disk

The following section describes the installation routine for starting from HD, Solid State Disk, or Compact Flash Disk. For more information on commissioning, refer to the corresponding technical specifications.

4.2.1 Booting from mass storage media

Preparing the USB Flash Drive

When installing the RMOS3 runtime environment on a HD or Compact Flash Disk of the target system, you need

- a USB Flash Drive that supports booting in DOS or Windows PE

Partition size

- A FAT16 partition has a maximum size of 2 GB. FAT16 supports up to 4 partitions.
- Partitions in FAT32 must have a minimum size of 4 GB. A storage unit of a physical size exceeding 32 GB must be partitioned to obtain two or several partitions (maximum 4) of which none exceeds the size of 32 GB.

4.2.1.1 New installation of the target system using a USB Flash Drive that supports booting in DOS

Preparing for installation of RMOS3 from a USB Flash Drive

Call of INSTUSB

The USB Flash Drive used to run the RMOS3 setup must support booting from DOS (e.g., SIMATIC PC USB Flash Drive; the FreeDOS operating system installed on the Mini CD can be used to set up the drive for booting from FreeDOS. Order no. 6ES7648-0DC40-0AA0). The necessary setup files are transferred from the development system to the USB Flash Drive by running INSTUSB.BAT from directory RM3DEV\SYSTEM\PC_CNUC.

At the DOS Command Prompt on the development system, call the

```
INSTUSB <ini> <loader> <target_dir>
```

file.

<ini> denotes the file extension of *RMOS.<ini>*- that is to be transferred to the target system, for example, INI for a standard-RMOS.INI file or 427B for the target system with Microbox PC 427B. On the target system this file is then named *RMOS.INI*.

<loader> determines the boot variant:

- **Bootung RMOS3 with RMOS3 boot loader**, parameter "1". this is the default variant.
- **Booten von RMOS3 mit DOS-Bootloader**, Parameter "2".

<target_dir> represents the target directory on the USB Flash Drive for storing the installation files.

The RMOS3 installation files (e.g., RMOS3 or DOS boot loader, RMOS3 kernel, configuration file RMOS.INI, the configuration task, and the runtime environment files) are copied to the USB Flash Drive.

Example

```
INSTUSB INI 1 F:\INSTALL
```

Preparing the mass storage medium

Partitioning and formatting the mass storage medium

The mass storage medium on the target system must be partitioned and formatted. For this purpose, boot MS-DOS from floppy disk, or from the USB Flash Drive and execute the FDISK and FORMAT utilities provided in DOS.

Note

Partition the volume in FAT16 or FAT32 format.

The boot partition must be in active state.

Do not label the volume after having formatted it. The RMOS3 boot loader must be the first file of the file system to enable booting of RMOS3. A label is interpreted as file in RMOS3 and may cause problems in the boot process.

Setup for booting RMOS3 using the DOS boot loader

You need to setup MS-DOS at this point if you boot the RMOS3 nucleus using the MS-DOS boot loader LOADX. Follow the setup instructions of MS-DOS.

It is best practice for the automatic start of RMOS3 to run the boot loader LOADX from AUTOEXEC.BAT in MS-DOS. Append the following line to your AUTOEXEC.BAT:

```
C:\LOADX C:\RM3_PC1.LOC
```

Observe the corresponding description of the DOS boot loader in chapter "LOADX.EXE" of the RMOS3 system manual.

Running RMOS3 setup on the mass storage medium from USB Flash Drive

Installation in DOS

You install the RMOS3 runtime environment in DOS. After having booted from USB Flash Drive and prepared the mass storage medium (see chapter "Preparing the mass storage medium (Page 50)"), run your installation by executing

```
INSTALL <drive>
```

<drive> represents the drive on the target system on which RMOS3 is to be installed.

Note

The USB Flash Drive from which you booted is assigned drive letter C:

INSTALL identifies the RMOS3 boot variant to be installed based on an entry on the USB Flash Drive:

With variant "**Boot RMOS3 with RMOS3 boot loader**", the boot sector is set up for the second stage boot loader RMLDR. The routine then copies RMDLR to the mass storage medium and transfers the RMOS3 nucleus RM3_PC1.SYS. This step is followed by the installation of RMOS.INI and CLISTART.BAT.

With variant "**Boot RMOS3 with DOS boot loader**", the routine transfers the DOS boot loader LOADX, the RMOS3 nucleus RM3_PC1.LOC, the RMOS.INI, and CLISTART.BAT.

In both variants, the routine then installs configuration task CNFTSK.386, the runtime environment files in directory <drive>:\RM3RUN, as well as utility files such as PASSWD, or HOSTS, which you need to run FTP and Telnet.

Rebooting the system

Remove the USB Flash Drive on successful completion of your setup. Reboot the systems to run RMOS3 from the mass storage medium.

4.2.1.2 New installation of the target systems from a USB Flash Drive that supports booting from Windows PE

Preparing for installation of RMOS3 from a USB Flash Drive

Call of INSTUSBPE

The USB Flash Drive used to run the RMOS3 setup must support booting from Windows PE (e.g., SIMATIC PC USB Flash Drive with preinstalled Windows PE operating system. Order no. 6ES7648-0DC50-0AA0). The necessary setup files are transferred from the development system to the USB Flash Drive by running INSTUSBPE.BAT from directory RM3DEV\SYSTEMPC_CNUC.

At the DOS Command Prompt on the development system, call the

```
INSTUSBPE <ini> <target_dir>
```

file.

<ini> denotes the file extension of *RMOS.<ini>*- that is to be transferred to the target system, for example, INI for a standard-RMOS.INI file or 427B for the target system with Microbox PC 427B. On the target system, this file is then named *RMOS.INI*.

<target_dir> represents the target directory on the USB Flash Drive for storing the installation files.

The RMOS3 installation files (e.g., RMOS3 boot loader, RMOS3 kernel, configuration file RMOS.INI, the configuration task, and the runtime environment files) are copied to the USB Flash Drive.

Example

```
INSTUSBPE INI F:\INSTALL
```

Preparing the mass storage medium

Partitioning and formatting the mass storage medium

The mass storage medium on the target system must be partitioned and formatted. For this purpose, boot Windows PE from the USB Flash Drive and execute the DISKPART and FORMAT utilities provided in Windows PE.

Note

Partition the volume in FAT16 or FAT32 format.

The boot partition must be in active state.

Do not label the volume after having formatted it. The RMOS3 boot loader must be the first file of the file system to enable booting of RMOS3. A label is interpreted as file in RMOS3 and may cause problems in the boot process.

Running RMOS3 setup on the mass storage medium from USB Flash Drive

Installation in Windows PE

You install the RMOS3 runtime environment in Windows PE. After having booted from USB Flash Drive and prepared the mass storage medium (see chapter "Preparing the mass storage medium (Page 53)"), run your installation by executing

```
INSTALLPE <drive>
```

<drive> denotes the drive on the target system on which RMOS3 is to be installed.

Note

The USB Flash Drive from which you booted is assigned drive letter C:

The routine now sets up the boot sector for the second stage boot loader RMLDR, copies RMDLR to the mass storage medium, and transfers the RMOS3 nucleus RM3_PC1.SYS. This step is followed by the installation of RMOS.INI and CLISTART.BAT.

The routine then installs configuration task CNFTSK.386, the runtime environment files in directory <drive>:\RM3RUN, as well as utility files such as PASSWD, or HOSTS, which you need to run FTP and Telnet.

Rebooting the system

Remove the USB Flash Drive on successful completion of your setup. Reboot the systems to run RMOS3 from the mass storage medium.

4.2.1.3 Updating the target system

Depending on the type of USB Flash Drive and boot variant used, you need to observe certain conditions when updating an existing RMOS3 nucleus on the target system.

Note

All system files must be replaced during the update to prevent unexpected effects. When updating a target system with configurable nucleus, verify that the RMOS3 nucleus and the configuration task CNFTSK.386 have the same version.

Update from USB Flash Drive that supports booting from DOS

Booting RMOS3 using the DOS boot loader LOADX

The RMOS3 nucleus RM3_PC1.LOC can be stored at any position on the data volume, which means you can run the update by copying the new RMOS3 nucleus to the volume.

Further installation of the nucleus is handled by means of USB Flash Drive, variant "Booting RMOS3 with DOS boot loader", as explained in chapter "New installation of the target system using a USB Flash Drive that supports booting in DOS (Page 49)".

Booting RMOS3 using the RMOS3 boot loader RMLDR

The RMOS3 nucleus RM3_PC1.SYS must be located at the second position in the root directory and may not be fragmented. Based on these conditions, you have several options of running the update:

- Replacing RMOS3 nucleus RM3_PC1.SYS, the configuration files, and the reloadable programs by copying the new versions to the volume. The SYSCOPY tool (see the system manual, chapter 5.9) must be used to prevent the RMOS3 nucleus from being saved in fragmented state.
- Backup of all customer-specific data (e.g. to second partition D:) and new installation in accordance with chapter "New installation of the target system using a USB Flash Drive that supports booting in DOS (Page 49) ", variant "Booting RMOS3 with RMOS3 boot loader".

Update from USB Flash Drive that supports booting from Windows PE

The RMOS3 nucleus RM3_PC1.SYS must be located at the second position in the root directory and may not be fragmented. Based on these conditions, you have several options of running the update:

- Replacing RMOS3 nucleus RM3_PC1.SYS, the configuration files, and the reloadable programs by copying the new versions to the volume. The SYSCOPY32 tool (see the system manual, chapter 5.10) must be used to prevent the RMOS3 nucleus from being saved in fragmented state.
- Backup of all customer-specific data (e.g. to second partition D:) and re-installation in accordance with chapter "New installation of the target systems from a USB Flash Drive that supports booting from Windows PE (Page 52)".

Human Interface of RMOS3

Overview

Based on corresponding examples, this section outlines the steps in commissioning the operating system on completion of its installation. The section provides basic knowledge of the operator interface and of the I/O of the operating system. It is intended to incite you to test the operating system, its reactions and possibilities.

5.1 Starting the RMOS3 operating system

Booting

You can now boot the RMOS3 system as specified in chapter "Commissioning the target system (Page 48)" to get started.

Press <CTRL>-<R> to launch the RMOS3 command line interpreter CLI on a white screen background.

5.2 Operating system I/Os on the PC

Units and devices

RMOS3 handles all input/output activities by means of drivers (devices) and their subunits, namely the units. During configuration of the OS, you specify the drivers and units it has to contain and the corresponding ID number (device ID, unit ID) with which they are used. In RMOS3, screen output and keyboard input are handled by means of the BYT driver. Usually, an RMOS3 system encompasses further drivers and units. A terminal, for example, is operated in serial interface COM2. The assignment of the interfaces (COM1, ..) to the device and units must be queried using the RMOS3 resource catalog. The RMOS3 programs may contain the definition of the device and unit to be used for data input/output. In many situations, I/O requests are defined by the runtime environment CRUN, or in the calling program.

RMOS3 provides four RMOS3 consoles for data input/output. Usually, these are the units 0 to 3 of the BYT driver. You can use the <F1> to <F4> keys to switch between the units. The four units are visualized on different background colors to obtain a clear differentiation.

Start screen for RMOS3 on PC

On successful completion of system startup, the "System is ready" message is output to the VGA screen of the PC.

Starting the CLI

Press <CTRL>+<R> and then <RETURN> to launch the CLI.

Changing from unit to unit

The screen changes to the red background color if you press <F2>. The keyboard and screen are now assigned to unit 1 in both operating systems. A prompt is not output because you have not yet started a program at this unit.

At this unit, for example you can launch a second session of the CLI by entering the <CTRL>+<R> shortcut key. Both CLIs run independently.

In RMOS3 you can now use the <F5> to <F8> keys to assign the screen as output to a different unit; the keyboard remains assigned to the unit it was assigned by means of the <F1> to <F4> keys. These are special functions of the BYT driver.

Example of the BYT driver function in RMOS3:

If the screen and keyboard are still assigned to Unit 1 (by pressing <F2>; red screen is output) and you press <F4>, the screen changes to Unit 0 (white). The keyboard input "dir <Return>" has no effect on Unit 0, because the keyboard is still assigned to Unit 1. Press <F2> to return to unit 1. This is necessary to display your keyboard input and the program response.

Exit the CLI on Unit 1 by entering the "exit <Return>" command and return to Unit 0 by pressing <F1>.

Country-specific keyboard layout

In the following example, note that the RMOS3 is factory set to the US keyboard layout that is commonly used on development systems. You may convert to a German keyboard by specifying the keyboard layout in configuration file RMOS.INI or by entering corresponding commands.

Changing the keyboard layout by means of CLI command input: Enter the command for changing the keyboard layout, followed by the specification of the selected keyboard driver, for example, for a German keyboard:

```
chgkbd kbdger
```

The following parameters are valid:

Table 5- 1 Keyboard layouts

Country	Keyboard layout
Germany	KBDGER
France	KBDFRE
Italy	KBDITA
Spain	KBDSPA
England	KBDUKE
USA	KBDUSE

You may also specify the selected keyboard layout in configuration file RMOS.INI.

USB keyboards are only provided support for the US and German keyboard layout.

5.3 Command line interpreter CLI

Function

The CLI serves to launch RMOS3 programs and display directories; in short, to form a user interface for RMOS3.

Operation and environment

The CLI is operated similar to the DOS command line interpreter. CLI entries are not case sensitive. To the CLI, "DIR" is equivalent to "dir". The CLI always starts by executing the CLISTART.BAT command file. In the factory state of the system, this file is used to set the cursor shape (prompt) and the search path that the CLI scans for programs. In factory state, the search path is set so that the CLI will find the included RMOS3 example programs in directory C:\RM3RUN (DIR.386, TIME.386, etc). Note that the CLI comprises both in-line and reloadable commands. In-line commands do not need additional files, while reloadable commands represent separate programs that must be available via the search path. For this reason, it must always be possible to reach at least the programs in directory C:\RM3RUN via the search path. For example, execution of the PATH command but not of the HELP command is an indication of an incorrectly set search path.

Most important commands

For comprehensive information on all CLI commands, refer to the Reference Manual Part I.

Start the CLI on your selected console by entering the <CTRL>+<R> shortcut key.

The essential commands are explained in the following section:

HELP	<p>is an external command that must be made available to the CLI via the search path.</p> <p>If you call <code>HELP</code> without parameters, a short description of the commands is output to the screen.</p> <p>If you call <code>HELP</code> and include the command name as parameter (e.g., <code>help dir</code>), a short description of the command and of its syntax is output.</p>
PATH	<p>You can use this command to view or edit the search path.</p> <p>The search path is displayed if you enter <code>PATH</code> without parameters.</p> <p>In contrast, the "<code>set path=c:\c:\RM3RUN</code>" entry sets the search path to volume C: and subdirectory \RM3RUN.</p>
DIR	<p>is an external command for displaying a directory.</p> <p>Specify the files to display at the first parameter; it is permitted to use the *, ? wildcard characters. The "/w" or "/p" parameters are also supported to set the screen to multiple column or page output. Parameter "/s" is particularly useful to extend the search for files to display to the subdirectories of the current directory.</p>
SYSTAT	<p>displays the CLI state. It informs you of currently active jobs (programs that the CLI controls) and indicates the device and unit that you are currently working with. Systate has no parameters.</p>

SESSION	<p>launches a program on a different unit (or device). The <code>SESSION</code> is called by specifying or interactively querying the device, unit and command parameters.</p> <p>Example of command usage: First you call <code>SYSTAT</code> to indicate the device on which the CLI is active. Use the same device number for this call, but a unit number that is higher by the count of 1. At the call of <code>SYSTAT</code> for example, the CLI has Device 1, Unit 1 on console F2. The command <code>session 1 2 dir a:</code> assigns the display of the directory to Device 1, Unit 2. Unit 2 can be reached by pressing <F3>.</p>
EXIT	The <code>EXIT</code> command terminates the CLI.

The CLI can be used to run the following program files:

- RMOS3 programs with extension "386" (however, no Windows drivers)
- RMOS3 programs with extension "EXE" (however, no DOS programs)
- Batch files with extension "BAT". These files contain command strings in ASCII code.
 Example: CLISTART.BAT

5.4 RMOS3 debugger

Debugger functions

RMOS3 Debugger is a program for debugging tasks and reading system information. The RMOS3 debugger can be used to obtain an overview of system resources and of the operating system state, and explicitly change system states.

Start the RMOS3 debugger on your selected console by entering the <CTRL>+<D> shortcut key.

HELP	<p>If called without parameters, it outputs an overview of the commands and tells you whether these are available.</p> <p>The expected command syntax is displayed if you call help with parameters.</p>
LOADTASK	<p>A command for loading the program file that is available in OMF386 or EXE format. <code>LOADTASK</code> reports the task ID that can be used to run the program as task.</p> <p>For example, load <code>EXP_CRUN.386</code>¹ using the following command: <code>loadtask a:\bin\exp_crun.386</code> Observe the task number <ID> that is returned.</p>
REP TASK	Outputs an overview of the currently cataloged tasks
START	Starts a task. You must specify the task ID (identification number) as parameter. This ID is displayed, for example, when loading with <code>loadtask</code> .
EXIT	Closes the debugger.

¹ `EXP_CRUN.386` is not included automatically in the BIN path on the boot floppy. You will first have to install the program on the floppy disk.

Practical section: Creating an RMOS3 task

6.1 Creating an RMOS3 task using the GNU tools

Creation using the GNU tools

For information on creating an RMOS3 task using the GNU tools, refer to the RMOS3-GNU manual, chapter 7 "Example of application generation".

6.2 Notes on programming and loading tasks

Analysis of the automation task

The first step should always involve the precise analysis of the automation task. The organization of the various tasks and the way they communicate is derived from this analysis. This phase also includes the assignment of different priorities to the tasks.

Scheduling

The significance of task communication resources (e.g., of a specific flag or mailbox) and the number of participating tasks must be planned carefully. All task communication resources, except local flags, can be addressed by all tasks by means of SVCs. This approach demands particular discipline from the programmer. Because all resources are identified by an ID, you need to know the respective ID to parameterize an SVC during task implementation. It is best practice to assign the function and ID number as follows:

A resource catalog is provided in RMOS3 to compensate for the fact that the IDs of dynamic resources remain unknown until runtime of the task that generates the resources by means of SVC (SVC `RmCreateTask`, `RmCreateBinSemaphore`, etc.). This catalog serves to make the dynamic resources known to other tasks as well. You could also see this catalog as small database. The catalog consists of entries that have the following structure:

name (max. 15 char) – resource type – ID No.

New entries can be generated by means of SVCs (SVC `RmCatalog`) and old entries can be deleted (SVC `RmUncatalog`). You can rely on search calls (SVC `RmGetEntry`) that include the respective name or type as parameters. The catalog can be accessed by all tasks. You may also enter static resources in the catalog. A particular advantage of the catalog is the fact that you only need to know the name and type of the resource for task programming, but not necessarily the ID. This approach represents the method recommended for RMOS3.

Getting started with examples

Use the tasks of the application examples to get started with task programming. These example tasks contain a number of particularly useful routines. Wherever possible, you should rely on these routines for your software development.

Application as dynamic task

Along with the aforementioned method, you may also employ the CLI, or the relocatable task loader, or the debugger to load a task (as file) dynamically at runtime. Once the task was loaded, it is capable of generating further dynamic tasks, similar to a static task.

The TCD block for a dynamic task is generated at runtime by means of system call and can be deleted by executing a second system call. The task ID is also assigned or released in this process. There is no other difference between dynamic and static tasks. The task can be identified and named based on its name entry in the catalog. Identification of a task based on its ID is made possible for reasons of compatibility.

General considerations

You need to observe certain aspects of task configuration:

1. When creating a TCD in the configuration, you need to define all start data of the static tasks (constants); this includes the task entry point, the start values of specific registers, and the values that define priorities. Symbols belonging to these values must be made accessible in the binding process.
2. All HLL tasks must contain a `_FAR` task entry point. The task entry point marks the start of a PUBLIC procedure. At the task entry point, a jump is executed to the procedure named as `UserTaskEntry` in the function call. At the start, two uint parameters are passed on the stack to the procedure for optional evaluation. The parameters are derived from the `TCD.EAX` and `TCD.EBX` components in the TCD block.

Stack length of a task

Each task is allocated a separate stack at runtime. The length of this stack must be calculated separately for each task and specified at the `RcInitUserTask` or `RmCreateTaskEx` function call. You also need to observe this rule when creating dynamic tasks. The length of a task stack is calculated based on four variables:

1. Stack requirements of a task:

The internal stack requirements of a task encompass memory space that is required for general purposes (automatic variables), for calling subprograms, and for passing the arguments. Calculation of stack requirements is no trivial matter when it comes to using recursive or deeply nested subprogram calls. The setup of a stack of inappropriate length is likely to lead to apparently inexplicable system crashes. You are well advised to configure a safety reserve.
2. Stack requirements of the operating system per task:

Upon status transitions of a task, RMOS3 retrieves the processor registers from the internal stack of the task. This action requires 40 bytes.
3. If the task uses CRUN functions (e.g. `printf`, `scanf`), additional 4 KB stack must be provided.
4. Additional stack for interrupt handlers in DI state. The system interrupt handler that needs the largest stack in DI state is decisive, because the interrupt handlers in DI state use the current task stack.

Practical section: Testing an RMOS3 task

7.1 Testing using the GNU tools

For information on procedures for testing RMOS3 tasks by means of the GNU tools, refer to the RMOS3 GNU Manual, chapter "Testing an RMOS3 application".

7.2 SVC exception handler, status messages and test output

SVC exception handler

The RMOS3 system provides an SVC exception handler. It is intended to react at lowest level to program errors that cannot be managed in the user software. Each SVC that returns a return code greater than zero is assigned an error message and output by the error logger task.

The SVC exception handler can be configured for a system. The default exception handler (SVCEXC.C) is available in the RM3BAS.LIB library. For information on its configuration, refer to the System Manual, chapter "Configuring the RMOS3 nucleus". This document will only deal with the library version. This version outputs all SVCs having a status unequal zero and that are not listed in exception table `ignore[]` to the system console.

Status evaluation

An SVC with return code greater than zero is not necessarily faulty. A timeout error will be reported, for example, if SVC `RmGetFlag` is assigned a waiting time and the flag is not set within this time. The program may be instructed to ignore such errors. You can prevent the output of unnecessary messages by dispensing with the configuration of the exception handler or by suppressing the message to the error logger task. For this reason, the user program should always rely on the status evaluation and initiate corresponding error handling routines. The function `svc_sts` shows an example.

Listing "Default reactions to SVC return status unequal 0"

```

/***/ void svc_sts(unsigned int index, char *svc_txt, int status)/***/
{
char buf[RM_MAXDECODELEN];
  if (status >0)
  {
    printf("\nERROR : %s INDEX: %i CODE: %i", svc_txt, index, status);
    if (RmDecode(RM_SVCERROR, status, buf) == RM_OK)
      printf(" = %s\n", buf);
    else printf("\n");
  }
}

```

If an error occurred, the error code is converted into plain text using SVC `RmDecode`.

Example: A call of the `svc_sts` function

```
svc_sts(1,"RmCreateFlagGrp", status);
```

triggers generation of an error message that displays the triggering SVC (`RmCreateFlagGrp`), an index number as cross-reference to the program code and for orientation (e. g. `INDEX = 1`), and the corresponding return status (as code or plain text). This measure initiates an error handling routine at user level when the program runs. This makes it easier to locate errors in the program.

7.3 Testing with RMOS3 Debugger

Start

Start your computer and launch a CLI session for the following demo of the test mechanisms.

On completion of the startup, start the `S_TASK` example task by entering

```
C:\>c:\<PATH>\s_task <Return>
```

The first exception

Following the initial message, the task is supposed to report the flag group at which it is waiting. Instead, the initial message is followed by the output of the following messages on the system console:

```
*** nuc-0: 09-MAR-1995 09:50:35, svc RmCreateFlagGrp from task: CLI_JOB_1 id: 0x28
failed: 37 (Invalid string)
ERROR: RmCreateFlagGrp INDEX: 1 CODE: 37 = Invalid string
```

and the `S_TASK` then signals the flag group.

The message was generated by the integrated exception handler of the system.

Locating the error

The exception message above is a typical example of an SVC exception handler reporting a faulty SVC. It indicates the name and ID of the task that has called the SVC (28H), the SVC name (`RmCreateFlagGrp`), the return code (numerical (37) and in plain text) of the SVC, and the time of occurrence of the exception. The string to be entered contains invalid characters or exceeds the maximum length.

Output of the test message begins with `ERROR` and is initiated by the user-specific `svc_sts` function. This function does not have to rely on the availability of a configured SVC exception handler in the system. The message is evaluated similar to the exception message.

Identifying the error

If you check the line containing the `RmCreateFlagGrp` entry, you will see that the following line is obviously faulty:

```
Status = RmCreateFlagGrp("THIS_IS_THE_SECOND_FLAG", &FlagId);
```


The exception was triggered due to the string having exceeded the valid length of 16 bytes. Therefore, we need to correct the line as follows:

```
Status = RmCreateFlagGrp("SECOND_FLAG", &FlagId);
```

Eliminate this error in the source code and then recompile the program. Restart the task as shown earlier. The task will now run without this error message and wait for the flag to be set (SECOND_FLAG).

Change to a different RMOS3 console, start the debugger, and enter

```
>svc rmsetflag <Task-ID> 1 <Return>
```

to set the flag.

Using the debugger for troubleshooting

After having returned to the CLI session, your search for the completion message will be fruitless. This means that the task was interrupted before the message was output. To find the reason for this situation, return to the debugger and execute the

```
>dir <Return>
```

command to view the catalog entries. The S_TASK was started as CLI_JOB_1. Note down the task ID and enter

```
>rep task <Return>
```

to view the task evaluation. The `event flag` status is still marked for the S_TASK (CLI_JOB_1), which means that the S_TASK is still waiting for the event flag. Enter

```
>rep flag lo <Return>
```

to view the flag report in detail (**long**). For the flag group reported by the S_TASK, the report shows that flag 1 is set and that a priority 40 task (S_TASK) is waiting at the flag group. The test mask of the flags is set to **3** (binary 0011), and the logic type is **AND**. This means that the task is waiting until both flags are set, which means flags 1 **and** 2. This setup was actually intended, which is why we changed call parameter **RM_TEST_ONE** of SVC `RmGetFlag` to **RM_TEST_ALL** in the corresponding source code line and also defined **FLAG2**. The error is therefore caused by incorrect operation of the debugger. You need to set both flags in the debugger to continue the task:

```
>svc rmsetflag <Flag-ID> 3 <Return>
```

Exception: SVC error

The task resumes operation after this command was entered. However, a new exception message is triggered:

```
*** nuc-0: 09-MAR-1995 09:50:35,  
svc RmGetBinSemaphore from task: CLI_JOB_1 id: 0x28  
failed: 36 (Invalid ID)  
ERROR: RmGetFlag INDEX: 2 CODE: 36 = Invalid ID
```

This time, **SVC RmGetBinSemaphore failed**. Status 36 shows you that the passed ID was invalid. If you now compare the source code, you will see that the lines

```
Status = RmGetBinSemaphore(RM_WAIT, SemaId);  
svc_sts(3, "RmGetBinSemaphore", Status);  
  
Status = RmCreateBinSemaphore("S_SEMA", &SemaId);  
svc_sts(4, "RmCreateBinSemaphore", Status);
```

were entered in the wrong order. In this case, an attempt is made to reserve the semaphore with **SVC RmGetBinSemaphore**, before it was created with **RmCreateBinSemaphore**. Source code with corrected order:

```
Status = RmCreateBinSemaphore("S_SEMA", &SemaId);  
svc_sts(3, "RmCreateBinSemaphore", Status);  
  
Status = RmGetBinSemaphore(RM_WAIT, SemaId);  
svc_sts(4, "RmGetBinSemaphore", Status);
```

The task continues until completed, regardless of the incorrect semaphore reservation. The done message is output on the CLI console and the task ends automatically 30 seconds later.

Verify proper completion as usual by using the debugger to check on system resources.

You should then eliminate the second error in the source code, recompile it and once again test the program.

Properties of the RMOS3 operating system

RMOS3 structure

This chapter provides a global description of the RMOS3 structure and application options.

Note for the reader

The RMOS3 structure is independent of the hardware. This chapter offers a comprehensive overview of the terminology you need to understand the operating system.

It is surely worthwhile for newcomers to get an initial overview of this complex chapter and then go into the detail offered in the following comprehensive sections.

RMOS3 tasks

Roughly simplified, we can say that it is possible to split software into two groups:

- system programs that control a computer and its I/O devices and
- application or user programs.

In general terms, computers consist of one or several processors, memory, network interfaces, mass storage media, and I/O modules.

An operating system relieves users from direct operation of the hardware by handling the control and management of the hardware. Instead of a hardware interface (e.g., registers and interrupts), the operating system provides the application program with a software interface. This interface is implemented by means of operating system calls (SVCs, also known as system calls or supervisor calls).

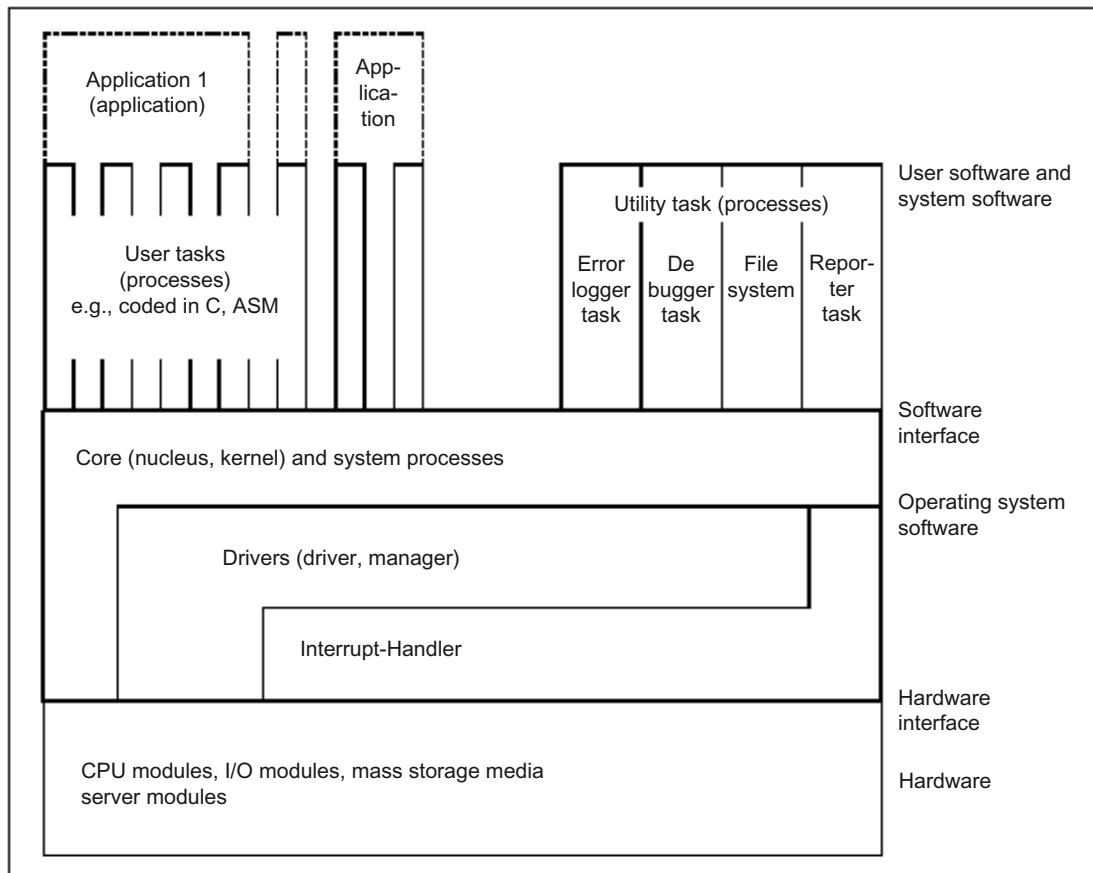


Figure 8-1 Operating system structure

Performance features for real-time applications

The scope and performance provided by the RMOS3 operating system for real-time applications can also be highlighted based on the following criteria:

Memory protection:

Protection against the overwriting of operating system code and data by user tasks, as well as against the overwriting of application code by user tasks.

Multiprocessing:

Capability of simultaneous execution of several tasks on several processor cores

Multitasking:

Capability of managing asynchronous (concurrent, quasi-parallel) independent tasks (processes). The operating system needs to provide the necessary system calls to this effect.

Real-time behavior:

Reaction to events within a specified (guaranteed) period. An event in this context could be a software or hardware interrupt. Critical aspects are delayed processing of the event and the possibility of assigning priorities for event processing. The reaction times and duration of event processing by RMOS3 depends decisively on the following conditions:

- Processor type used and its clock rate
- Priority of the hardware interrupt
- Priority level of the tasks

RMOS3 guarantees that processing of an interrupt of highest priority is initiated within a maximum of 2 μ s (Intel Core™ 2, 2.166 GHz).

Communication, synchronization, and coordination:

Making operating system calls (SVCs) available to enable synchronization or message exchange between independent tasks. An application (e.g., a control system) can then be implemented by several tasks that are synchronized if necessary. RMOS3 offers a number of SVCs that support these features.

Interprocess communication:

SVCs that support message exchange (communication) between tasks running in self-contained RMOS3 operating system configurations. This link can be realized by means of dual-port RAM, shared memory, or network.

Testing and debugging:

An interactive debugger is available for testing the user tasks. Further utilities include the monitoring of authorizations to access the stack and data areas and the logging of corrupted SVCs at task runtime. The resource reporter provided is a utility that can be used to trace task synchronization and communication and to analyze the synchronization and communication mechanisms (you can also use the debugger for these analyses). You can use the RMOS3 debugger in monitor mode, or an emulator, to perform complete driver tests.

In addition, symbolic debugging by means of HLL is also possible using the rm-gdb of RMOS3 GNU (separate product).

RMOS3 elements

The RMOS3 operating system is organized based on the following elements: kernel, drivers, interrupt handlers, and system processes. The nucleus (also core or kernel) contains all procedures that are necessary to execute the multiprocessing functions. The scheduler has key function in this context. The scheduler allocates the cores to the tasks, depending on task priority.

Nucleus

The communication and synchronization mechanisms of user tasks, as well as memory and time management are allocated to the nucleus. Last but not least, the nucleus also handles the system calls of the software interface.

Drivers

In RMOS3, distributed I/O functions are controlled and managed by corresponding device-dependent software modules. This software is implemented by drivers. Each driver controls and manages an distributed I/O function or a class of distributed I/O functions of the same type. An example of distributed I/O functions of the same type is the control and management of character output to different terminals. All drivers have a uniform interface to the nucleus. The drivers and hardware are synchronized by means of hardware interrupts and the associated interrupt routines, also known as interrupt handlers. The nucleus provides a uniform system call `RmIO` that is valid for all drivers and that can be employed by user tasks to address the drivers. Drivers are configured in two phases. Firstly, the driver must be known to the operating system kernel. Secondly, the driver and its interrupt routines need to be adapted to the hardware interface.

Utility tasks

Utility tasks are program running on the RMOS3 operating system similar to standard user tasks. RMOS3 utility tasks include, for example, the HSFS file system, the debugger, and the resource reporter.

Applications

Applications or user programs consist of one or several tasks, which are capable of interactive communication and synchronization. All tasks are based on RMOS3 system calls. The tasks may be programmed in Assembler or in a high-level language.

8.1 Memory protection

Description

In addition to protected memory for segmented programs, RMOS3 provides memory protection for flat programs. This protection prevents user programs written in the flat memory model from overwriting operating system code and data, as well as the code of other user programs.

Memory protection is realized by using the paging and privilege level mechanism of the 80386 architecture. Moreover, heap management data is saved to a special area that cannot be overwritten by user programs.

Privilege level mechanism

User programs are always assigned privilege level 3 (PL3) that is lower than privilege level 0 (PL0) of the kernel and system tasks.

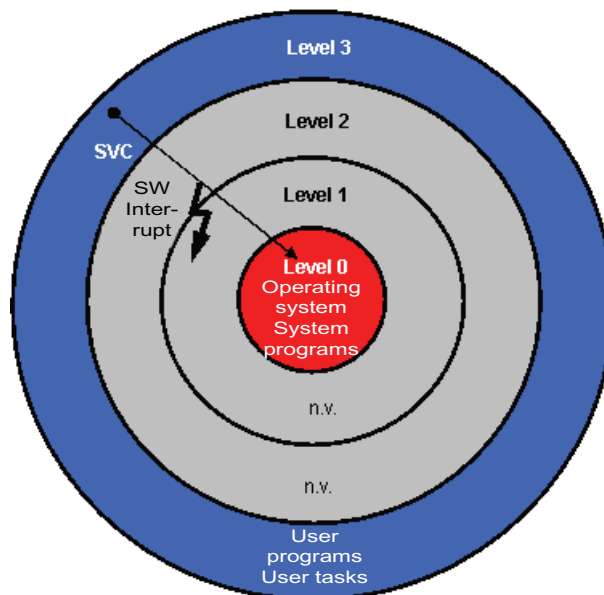


Figure 8-2 Overview of privilege levels

Tasks that are started from RMOS.INI run as system tasks with PL0.

The PL0.386 program can be used to start tasks at privilege level 0.

The RPROF.386 and POOL.386 programs are executed automatically at privilege level 0.

Table 8- 1 Privilege levels of system tasks

Task	Privilege level
Busy	0
Debugger	0
Debugger dispatcher	0
Error logger	0
Exception task	0
File system	0
Reporter	0
Remote	0
VGA task	0
Init task (rmconf.c)	0/3
CLI	3
CLI dispatcher	3

Paging

The entire memory is emulated in a shared linear address space by means of paging.

Access rights

All unused (free) memory pages are marked as read only. Memory space requested by user programs (e.g., `m_malloc`, `RmAlloc`) is marked for read/write access. User programs can only write access these especially released pages and cannot corrupt the remaining protected memory. Write access to all pages is enabled for the kernel and system tasks at privilege level 0.

User programs with privilege level 3 cannot write access operating system code and data or the code of user programs. However, read access is possible. CRUN and CLI data form the exception, as these are enabled for read/write access at privilege level 0 and 3.

The following table highlights the access rights for the different memory models. In the case of the flat memory model, you must distinguish between the Debug and Release version under GNU.

Table 8- 2 Access rights

	Operating system code and data	Code of user programs	User Const range
PL0 Flat (GNU / BORLAND)	R/W	R/W	R/W
PL3 Flat Debug (GNU / BORLAND)	R	R/(W) ¹ (GNU) R (BORLAND)	R/W
PL3 Flat Release (GNU / BORLAND)	R	R	R (GNU) R/W (BORLAND)
PL0 Segm. (CAD-UL)	R	R/W	R/W
PL3 Segm. (CAD-UL)	R	R/W	R/W

1 In normal situations, the major part of the code area is protected. However, there is an overlap area with read/write attribute.

Page Fault Exception

Attempts to write access memory areas that are read-only for the respective privilege level will lead to a page fault exception. Errors can be analyzed using the GNU tools.

NULL pointer

Access to the lowest 4 KB range at privilege level 3 is impossible as a matter of principle. Access to a non-initialized pointer will therefore lead to a page fault exception. Read access to the lowest 4K range is possible at privilege level 0.

User stacks

Each user application requires a user stack. User stacks are usually allocated from the standard heap, where they are read-only. The stack pages are always mapped with read/write attribute to a higher linear memory area. The pages before and after a stack are locked (guarded pages, each one 4 KB). With this setup, a page fault exception is triggered on stack overflow or stack underflow. However, incorrect access to a valid stack of a different task beyond the locked page cannot be detected.

All user programs are assigned a user stack and a system stack. Memory space requested from the user stack is rounded to 4K. System stack always has a length of 3072 bytes (0x0c00) it is used when operating system calls are processed. Only the user stack is visible to users. The user stack can be set up for segmented and flat applications (see the RMOS3 GNU description).

User stacks must be generated using `RmCreateTask` and `RmAlloc` or `RmMemPoolAlloc` with parameter `RM_STACKALLOC`.

Memory allocated with `malloc` for use as stack is not protected.

Existing programs

Existing user programs can be executed in the segmented and flat memory model without changes. Restrictions are listed below.

Access to physical memory

Write access to physical memory must be enabled in GNU and CAD-UL with `RmMapMemory`. Otherwise, you will receive a page fault exception. The call of `RmCreateDescriptor` is not sufficient in CAD-UL.

The system enables the memory space from 0xB8000 to 0xBBFFF for read/write access. Likewise, the entire memory space from 0xC0000 to 0xEFFFF is enabled for access to modules.

The PCI Shared Interrupt Server automatically enables all address spaces of the PCI cards.

The MAPMEM program is available for enabling the memory areas of ISA cards (see Reference Manual Part 1, chapter 2.2.20).

The configurable nucleus automatically enables the address spaces of configured modules.

I/O access

I/O access is compatible with CAD-UL (e.g., inbyte, outbyte, etc.) and can be executed without restrictions at PL0 and PL3.

RMOS3 SVCs

SVC calls are executed at system level (PL0). An implicit change from PL3 to PL0 is executed for user programs.

CRUN calls

CRUN calls are possible at any privilege level.

Restrictions

- All network drivers and the USB driver must run at PL0. For this reason, the drivers must be started in RMOS.INI.
- Syslogd and Telnetd must run at PL0.
- With segmented programs, interrupts can only be processed at PL0.
- Descriptors in segmented programs (CAD-UL), which point to memory areas of the nucleus, cannot (no longer) be used for write access.
- Programs that use fast timer ticks:
 - Existing GNU programs run without changes in PIC mode at PL0 and PL3. These programs must be recompiled for the APIC mode.
 - Existing CAD-UL programs run without changes in PIC mode at PL0. These programs must be recompiled for the APIC mode. CAD-UL programs with fast timer ticks no longer work at PL3.
- If the `RmMapMemory` is called at PL0, the memory release is only valid for PL0. Access to the memory is not possible at PL3.
- The `RmSetIntDIHandler` cannot be used with CAD-UL at PL3 to install any interrupt handler.

8.2 Multiprocessing in RMOS3

Processing concurrent operations

RMOS3 is a symmetrical multiprocessing (SMP) operating system that supports multicore architectures. Operation in single-core mode represents a special case of operation in multicore mode. In single-core mode, RMOS3 works only on one processor core, while using several processor cores in multicore mode.

Symmetrical multiprocessing is characterized by two or several identical processors sharing the same address space. This means that each processor with the same (physical) address will address the same memory cells or I/O register.

System load is automatically distributed to the processor cores. This is based on the condition that the application is segmented in several tasks that are capable of simultaneous operation.

Each individual program consists only of sequential program instructions. When changing to a different program, the operating system needs to retrieve the current state of the program. This applies only to interruptible programs that will resume execution at a later time. These interruptible programs are so-called tasks or processes, which must be managed by the operating system.

Note

Four cores are currently supported.

8.2.1 Startup

Starting the processor cores

The boot processor core (core 0) starts the BIOS and boots RMOS3. The other processor cores (core 1, etc.) are so-called application processors. These processors are initialized in the BIOS. While the PCIAPIC.DRV is loaded, the further processor cores are initialized, and the scheduler is executed.

Note

The APIC driver supports multi-core processors. Multiprocessing can be activated by setting the corresponding parameter (`PCIAPIC.DRV -M`).

8.2.2 Task properties

What is a task?

The term tasks can denote user tasks, utility tasks, or system processes for executing I/O requests. Switching between the different tasks is usually independent of the currently active task (asynchronous) and is triggered by an interrupt event. Each task has

- one or several code areas (segments)
- one or several data areas (segments), or none
- one stack area (segment)

The code area of a task consists of functions, procedures and constants that are written in Assembler or in a HLL.

In RMOS3, tasks and processes represent the same matter (in other operating systems, tasks may represent internal processes of the operating system that are used, for example, to execute driver functions, while user programs are realized by means of processes).

In RMOS3, the internal processes of the operating system are known as system processes, or programs running in S state (e.g., specific parts of driver programs).

Managing tasks

The RMOS3 nucleus manages all tasks (processes, programs), except the system processes, in two tables to support the correct concurrent sequence of all tasks. For this purpose, RMOS3 creates two data structures for each task. These data structures are referred to as process control blocks (TCD (**t**ask **c**ontrol **d**ata), TCB (**t**ask **c**ontrol **b**lock) that contain the start conditions (constants) for each task, as well as the variable data for resuming interrupted processing.

The following constants for a task start by the nucleus are also stored in the TCD:

- Start address of the task
- Address of the top of the stack
- Start priority of the task

The following variables are stored in the TCB:

- Current register set if the task was replaced
- Current priority

For information on the precise structure of TCD and TCB, refer to the RMTYPES.H file in the RMOS3 directory INC.

The operating system identifies a task based on the entry of an identification number (ID, or task ID) in the resource catalog. This corresponds to the index in both task management tables. The task ID can have a decimal value between 0 and 32767, which also defines the maximum number of tasks that RMOS3 is capable of managing.

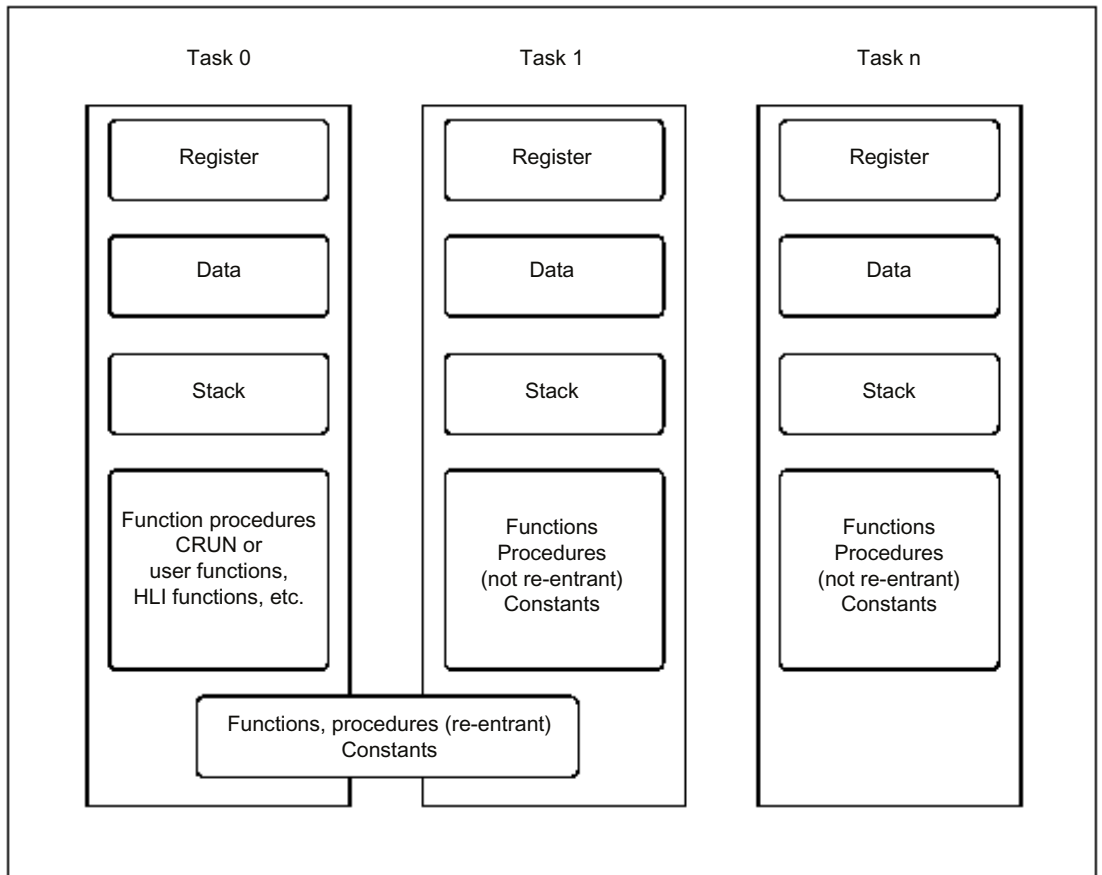


Figure 8-3 Tasks from the point of view of user programs

The SVCs `RmCreateTask`, `RmCreateChildTask` and `RmCreateTaskEx` request RMOS3 to enter the start data of a new task in the task management table. The new table entry consists of the parameters that were passed at the SVC. When deleting tasks, RMOS3 removes the entry from the task management table.

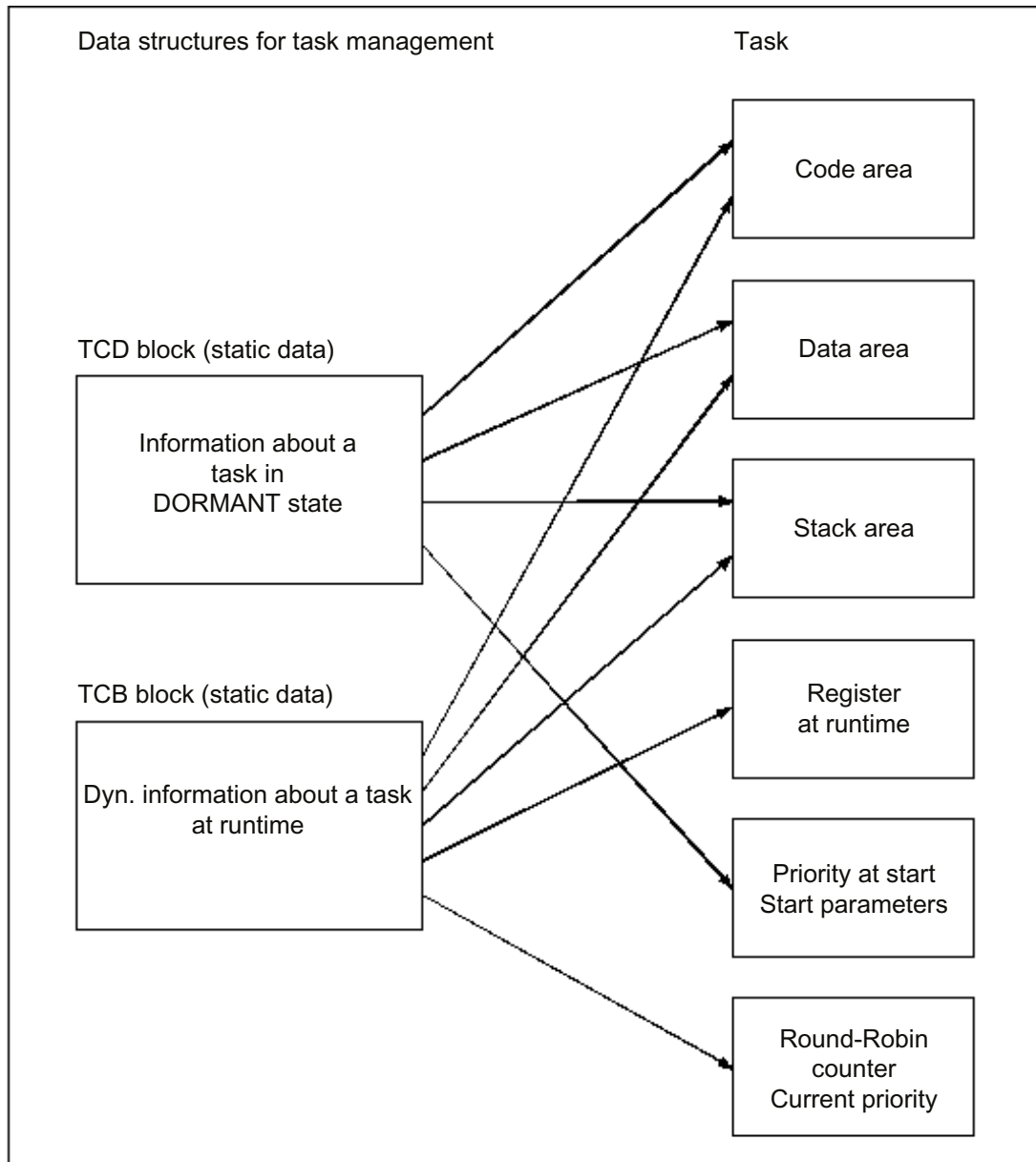


Figure 8-4 RMOS3 data structures for task management

Tasks allow for certain flexibility when you configure the operating system, because it is not necessary to specify all details in your initial configuration. For the purpose of testing, for example, it is possible to dynamically create and start different tasks without manipulation of the operating system configuration.

RMOS3 needs at least one task that it has already started at system startup. This task is specified in the configuration data. This task is therefore referred to as start task or init task and is capable of starting all other tasks by means of SVCs. Tasks are generated at runtime of a user program by corresponding SVCs.

SRB

RMOS3 provides internal dynamic management data structures, namely the system request blocks (SRBs), for system processes for runtime reasons.

System calls

An application can be realized by one or several tasks. The RMOS3 system calls are available to all tasks that have been programmed in Assembler or in an HLL. In RMOS3, system calls are also known as SVCs (supervisor calls). The SVCs form the interface between RMOS3 and the tasks. SVCs are triggered by tasks and processed by the RMOS3 system. SVCs are used, for example, to manipulate task states, allocate memory space, or transfer messages to other tasks. The operating system controls all necessary data exchange between tasks, as well as task synchronization.

Segmentation of an application in tasks

Segmentation of an application or user program (e.g. a conveyor control system) in tasks, as well as the communication and synchronization processes between tasks must be planned with due care. The following criteria for segmentation of an application and implementation of the segments as task must be taken into account:

- Simultaneous (quasi-parallel) processes are segmented to form different tasks.
- Processes of different priority are assigned to different tasks.
- Processes that fulfill a clearly defined function are implemented as task.

Moreover, application segmentation in tasks is governed decisively by factors such as program modularity and maintenance capabilities, as well as work distribution in the program development phase.

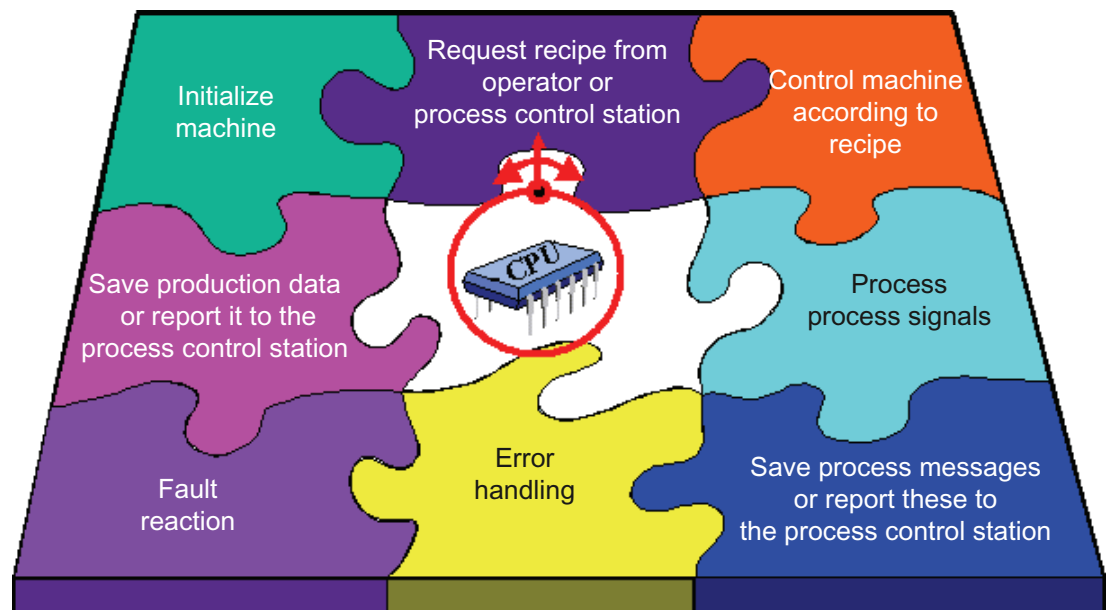


Figure 8-5 Segmentation of an application in tasks

Reentrant capable code

Code elements (e.g., functions and procedures) that can be used by several tasks are referred to as being reentrant. Code that is written with reentrant property saves all data and variables to the stack and may not automatically modify itself. This code can then be used simultaneously by several tasks, as each reentrant routine works on the stack of the calling task (this property is of importance, for example, for library routines). Reentrant code elements may enhance the maintenance of applications, as a procedure that was already tested can be used several times.

8.2.3 Task management

Five task states

Task management is a key function of the operating system core. Because it is not possible to process all tasks simultaneously, each task is always set to one of the following five states.

RUNNING

The task was allocated a core; the program code is executed. However, internal CPU requests (see below) of the operating system may always override the task.

READY

The task is ready for computing and waits to be allocated a core.

WAITING (BLOCKED)

The task waits for an event or point in time. Typical reasons for the BLOCKED state are the SVCs that are transferred with a WAIT parameter (waiting for completion of an I/O function, waiting for expiration of a pause, and waiting for a message). The operating system restores the task (in passive wait state) to READY state after the event was triggered, or the waiting time has expired.

DORMANT

The task is known to the operating system, including all necessary elements such as code areas, or stack, which means it was entered in the task management table of RMOS3. The task is not yet started by an SVC. The task transition from DORMANT to READY state is referred to as the task start and is initiated by an SVC. This SVC may originate from another task, from an interrupt routine, or from a driver. The task occupies code/data areas, and a stack area.

NON-EXISTENT

The task is not entered in the task management tables of RMOS3. This task may be available as file, for example, on a mass storage medium, or was already written to PROM. Only dynamic tasks can be in this state. The transition from NON-EXISTENT to DORMANT state, which means entry in the task management table, is referred to as creation or generation of the task and is triggered by an SVC. The reverse operation is referred to as destruction, which is also triggered by an SVC.

The following figure lists all task states and possible status transitions. The status transitions are initiated by explicit SVCs and events (interrupts), and by the scheduler.

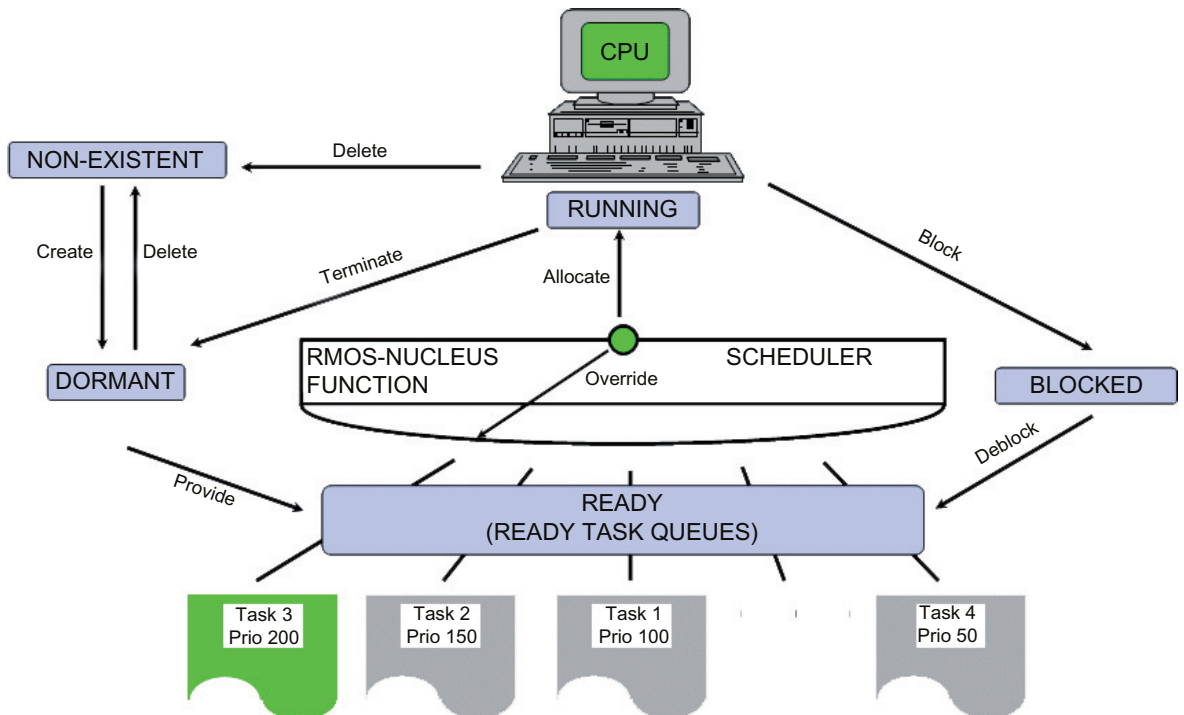


Figure 8-6 Task state and transitions

Allocating a core

If task execution is to be restricted to a single core, the task is declared as local task. It is then bound to this core. This setup ensures that the task will only run on a single core. A global task can run on any core.

By default, tasks are identified with global attribute, all generated tasks of an already existing software can always run as global tasks on any core.

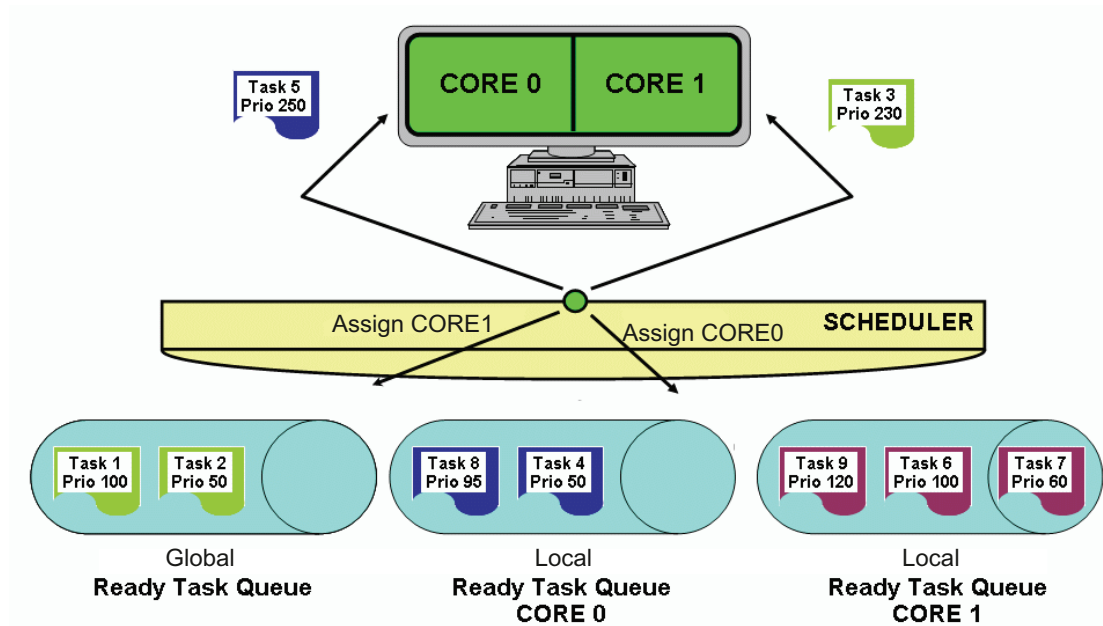


Figure 8-7 Scheduling mechanism, based on the example of a system with two cores

Ready Task Queue (RTQ)

RMOS3 maintains a queue for each core, which consists of all local tasks in "READY" state, as well as a global queue consisting of all global tasks in "READY" state. These queues are also known as RTQs (Ready Task Queues).

A task started in DORMANT state is entered in the RTQ. The scheduler is the operating system software in the nucleus for managing computing time slices. The scheduler allocates computing time slices (CPU times) to tasks.

An RUNNING state task that requests temporary interruption of its execution, or waits for completion of an SVC, is removed from the RTQ and set to BLOCKED state. On expiration of the pause or completion of the SVC, the operating system returns the task to the RTQ.

Tasks are entered in the RTQ based on their priority. This enables allocation of the task having highest priority in the RTQ to the CPU upon task changes. Tasks are entered in the RTQ based on the FIFO principle, which means "tasks of the same priority are entered in chronological order".

On changes to the global RTQ on a core, a rescheduling process is initiated on the other cores by means of an interrupt. This always ensures that the tasks with highest priority are active.

Note

In versions previous to RMOS3 V3.11, a task of the same priority was entered **before** the already existing task.



Figure 8-8 Ready Task Queue

Allocating time slices

When processing tasks, RMOS3 distinguishes between internal CPU requests of the operating system and external computing time requests from tasks.

Internal requests include interrupt processing, requests to drivers, data structure updates, time management and similar. All internal processes of the operating system are given priority over the processing of external computing time requests.

The internal RMOS3 scheduler allocates the computing time.

Allocation of computing time slices by the scheduler

When allocating computing time slices, the scheduler is oriented on the current priority of the tasks in READY state. Priorities are assigned with ascending ranking from 0 to 255 (255 is top priority). All tasks generated are assigned a priority. The priority can be changed by the task itself, by other tasks, or by the operating system in special situations. Such changes are initiated by means of SVCs. The scheduler always allocates the task that is of highest priority and in READY state to the CPU, which that the task enters the RUNNING state.

Tasks of the same priority in the local and global queue are handled in alternating mode.

A task having higher priority compared to the currently active task immediately overrides this active task when it enters the READY state.

Tasks of the same priority are handled based on a Round-Robin algorithm.

Round-Robin algorithm

With Round-Robin algorithm, each task is allocated a time interval, or time slice; the next task of the same priority will enter the RUNNING state on expiration of this time slice. If the task terminates itself prematurely (set to DORMANT state), or is blocked by means of SVC (set to BLOCKED state), the next task is set immediately to RUNNING state.

At each transition from READY to RUNNING state, the task is allocated a full time slice and the Round-Robin counter is once again initialized with the specific time interval. The length of the time interval can be configured separately for each task. This facilitates timesharing on the CPU for tasks of the same priority.

Times are managed only on processor core 0. This means that timer block interrupts are always triggered on processor core 0. Processor core 0 increments the internal X_ABS_TIME on each interrupt and executes expired timer jobs (timeouts). At each timer interrupt on processor core 0, this core also triggers a timer interrupt on the other processor cores by means of APIC. The other processor cores will now verify the Round-Robin counter and execute the scheduler upon expiration of the Round-Robin counter.

Note

For example, a task that initiates the internal RMOS3 routine "add task to RTQ" by means of SVC `RmStartTask` is itself scheduled. This could lead to a premature task change (< Round Robin time interval).

The previously scheduled task returning to the RUNNING state starts with a new, **complete** Round Robin time interval.

Round Robin in RMOS3 does not mean "realization of a time slice", but rather "the task cannot be RUNNING beyond the duration of the Round-Robin time interval".

Disabling task scheduling

SVC `RmDisableScheduler` disables execution of the scheduler for all local and global tasks and on all cores. Driver tasks (e.g. VGA task) are excluded from this lock.

8.2.4 RMOS3 API for task management

Overview of functions

RMOS3 provides an extensive API for task management. The following chapters outline the most important calls. For more information on the calls, refer to Reference Manual Part III.

Table 8- 3 RMOS3 API for task management

RMOS3 API for task management		
<ul style="list-style-type: none"> • Create task • Delete task 	<ul style="list-style-type: none"> • Start/stop task • Disable/enable task • Disable/enable task scheduling 	<ul style="list-style-type: none"> • Task priority control • Task status query • Bind task
<ul style="list-style-type: none"> • RmCreateTask • RmCreateChildTask • RmCreateTaskEx • RmDeleteTask 	<ul style="list-style-type: none"> • RmStartTask • RmQueueStartTask • RmEndTask • RmRestartTask • RmKillTask • RmPauseTask • RmSuspendTask • RmActivateTask • RmResumeTask • RmDisableScheduler • RmEnableScheduler 	<ul style="list-style-type: none"> • RmSetTaskPriority • RmGetTaskId • RmGetTaskInfo • RmGetTaskPriority • RmGetTaskState • RmBindTask • RmGetBindTaskInfo

8.2.5 Creating and deleting tasks

Creating a task

A new task is defined using `RmCreateTask`. RMOS3 accordingly requests two system memory blocks (SMR) for the TCD and TCB task control blocks, which are fetched from the internal RMOS3 memory pool. Parameter `TCD_PTR` of the SVC points to a memory area that contains the task control data to be copied to the requested SMR.

The structure of the TCD block is defined in the system configuration.

Deleting a task

`RmDeleteTask` is used to delete tasks. `RmDeleteTask` terminates a static or dynamic task, provided the queue does not contain any further start requests for this task. The task is set to NON-EXISTENT state.

8.2.6 Binding tasks

RmBindTask

The `RmBindTask` is used to specify whether to bind a (local) task to a specific core or whether it may run on all cores.

RmGetBindTaskInfo

Use the `RmGetBindTaskInfo` call to determine the core binding of a task.

Bound tasks

- Ethernet driver
- Debugger
- VGA task
- HD0 task
- Exception task (SYSDEMON)
- Telnetd

8.2.7 Starting tasks

By means of call, interrupt

A task can be started in different ways. The `RmStartTask` and `RmQueueStartTask` calls are available for direct task starts. In your configuration, you may also specify the task to start automatically at system initialization. Last but not least, a task can be started by an interrupt that was triggered by a hardware or software interrupt.

At system start

All static tasks in the system are in DORMANT state after system startup. RMOS3 automatically starts the initialization task you have configured. Although this task is not subject to specific restrictions, it is expected that it initializes the application-specific environment (data structures, etc.) and then starts further tasks that are necessary for the application.

`RmStartTask` is used by tasks to start another task. The addressed target task is merely switched to READY state if it was in DORMANT state; otherwise, the call has no further effect.

Instead of `RmStartTask`, `RmQueueStartTask` is used to start a task once per call. This call is similar to `RmStartTask`, except that the call is queued if the task to start is already in READY state.

8.2.7.1 Task start through unexpected input

Unexpected input

We speak of unexpected input whenever an distributed input device such as a terminal triggers an interrupt and no input request of a task is pending for this device. This situation arises, for example, if an operator presses the BREAK key on the terminal.

Two characters per device

RMOS3 supports the definition of a task that processes unexpected input from this device. The corresponding task is defined in the configuration of the respective device/driver. Unexpected input is ignored if no task was configured to process it. Unexpected input can be set up for two characters per device. After a task start was triggered by unexpected input of a task, the driver ID is set at register AL, the device ID at register AH, and the input byte at register BL; register BH is reserved (null).

An example of unexpected input is the start of the CLI on a VGA console. In this case, a CLI task (CLI_JOB_x) is started with <CTRL> + <R>.

8.2.7.2 Parameter passing at task start

Two data words as parameters

At the start (transition from DORMANT to READY state), each task receives two data words as parameters in the EAX and EBX registers. Only tasks written in Assembler can directly access these parameters. Tasks coded in an HLL can evaluate these parameters using the `getdword`, `getparm` and `get2ndparm` procedures.

Procedures `getdword`, `getparm`, `get2ndparm`

`getdword` returns an unsigned long value that corresponds with the EAX register.

`getparm` returns a pointer with the content of the EAX register as return value.

`getparm` or `getdword` must be the first statement in the program code, because programming languages use the EAX and EBX registers and therefore corrupt the original values. In addition, two parameters are transferred from the respective TCD block to the stack.

`get2ndparm` returns the EBX of the task as return value. This overwrites the EAX register. As a result, it is no longer possible to use the `getdword` and `getparm` functions.

8.2.8 Task priority

Degree of urgency

The priority of a task reflects the degree of urgency for task execution in RMOS3. The priority of a task is set in the task definition and may always be changed to respond to system requests.

8.2.8.1 Priority changes by SVCs

RmStartTask, RmQueueStartTask, RmSetTaskPriority.

SVCs capable of manipulating a configured priority include `RmStartTask`, `RmQueueStartTask`, and `RmSetTaskPriority`.

It is possible to change the priority on a temporary basis to accelerate task execution in response to situations demanding task processing speed that exceeds the originally intended speed. The first call of `RmSetTaskPriority` increases the priority level, while a second call of `RmSetTaskPriority` on completion of task processing restores the original priority.

8.2.8.2 Priority change triggered by timeout

RmSetTaskPriority, RmRestartTask

RMOS3 provides a mechanism for changing the current task priority if no `RmEndTask` (end task) or `RmRestartTask` (end task and restart on expiration of the time interval) was output within the specified time interval. You can rely on this facility to meet situations in which system load fluctuates across a wide range, and it is necessary to execute a cyclic task that is usually assigned low priority within a specific period. RMOS3 monitors completion of the task within this period; if not completed, RMOS3 raises the task priority. You specify the time interval, increase of the priority, and the priority high limit in the TCD block when creating the task (`RmCreateTaskEx`). The time interval must be shorter than the actual task execution time.

The specified time interval is reset and execution restarts at the original priority level at each task start that was triggered with `RmStartTask`, `RmQueueStartTask`, or `RmRestartTask`. On expiration of the period, the priority is increased, the time interval is reloaded, and the down count is restarted.

This process is not terminated until the task has called `RmEndTask` or `RmRestartTask`, or reached the maximum priority level. The original task priority can also be restored to original level by calling `RmSetTaskPriority`.

8.2.8.3 Automatic priority change by means of semaphore possession

Semaphore possession

To prevent a lower priority task that is in possession of a semaphore from blocking a task that requests a semaphore, the task in possession and the requesting task are assigned the same priority until the semaphore is reset. Once it has returned the semaphore, the task's raised priority level will be restored to the original priority level.

The original priority level is not restored if a priority other than affected by the priority change is pending at the time of reset of the semaphore. In this case, it is assumed that an explicit change in priority was executed with `RmSetTaskPriority`.

8.2.9 Memory distribution

Local and global areas

Operating system data is split into local and global areas. Both areas must conform to page limits (4 KB). The global area contains all data that is the same for all processor cores. Each processor core is assigned a separate local area. This is achieved by entering different physical pages for the local area in the page table of each processor core.

8.2.10 I/O request

RmIO

I/O requests with `RmIO` call are always executed on processor core 0, which means that the driver and interrupt handler are only executed on processor core 0. This ensures that the existing drivers (BYT, CRT, 3964, FD0, HD0, RAM, EMEM, USBMEM) will run without changes. An I/O request that is executed on processor core 1 is added to the SRB queue (S state) of processor core 0 and processor core 0 is informed by means of an interrupt. Processor core 0 executes the I/O request immediately if no interrupts and other S states are pending processing.

This means that processor core 0 always executes all HW interrupts. The same applies to the reloadable drivers (e.g., LAN drivers).

8.2.11 SYSTEM HALTED

Running out of SMRs or SRBs

If one processor core runs out of SMRs or SRBs, all processor cores will be stopped.

8.2.12 Performance

Speed gain

Even with multi-thread applications, the existence of two processor cores does not necessarily imply that an application will run at double speed. A rule manifested by Gene Amdahl in 1967 describes the restrictions of performance gain on multi-processor systems by threads that cannot be executed in parallel. According to Amdahl, the speed of a multi-threaded application never increases in linear proportion to the number of processors. Further performance losses are incurred by limited system resources, such as the bandwidth of shared RAM used in SMP systems.

Programs at which efforts for synchronization of threads and data exceeds the time gain through parallel processing are unsuitable for multi-threading. Their execution would always be slower compared to the single-threaded variant - even with several processors. In comparison, multi-threading is particularly simple and effective whenever it is possible to distribute large data quantities to independent segments (segmentation).

Execution of single-threaded applications in a multi-processor system is not faster compared to a single-processor system, because the processors continuously communicate and, for example, synchronize their cache contents. This overhead for synchronization will always reduce the speed of the respective application.

Cache line

Performance is substantially reduced if several processors access the same cache line (e.g., 64 bytes). It makes no difference whether this is a read or a write operation. One would expect that performance losses only occur during write access to the cache line, because the write operation would make it necessary for the first processor core to continuously refresh the cache of the second processor core.

This finding shows that a gap of 64 bytes should be set for the individual variables. Also, all variables should be aligned to a length of 16 bytes.

Example:

```
struct _NowRead
{
    volatile char dummy0[64]; /* optimize data fetch */
    volatile uint cpu0;
    volatile char dummy1[62]; /* optimize data fetch */
    volatile uint cpu1;
    volatile char dummy2[62]; /* optimize data fetch */
};
typedef struct _NowRead NowRead;
```

8.2.13 Notes on porting

Interrupt disable

In a multi-processing system, it is possible that programs safeguarding themselves against interruptions by means of interrupt disable (`cli` or `disable`) will no longer be executed.

An interrupt disable may possibly prevent interruption by an interrupt on the same processor, but not the interruption by the other processor.

Interrupt disables can no longer prevent a variable from being accessed by a different task in the multi-processing system.

You have three solutions for programs with interrupt disable:

- Binding the program to a core
- Replacing interrupt disables with spinlocks
- At task level: using `RmDisableScheduler`

Tasks

All generated tasks of an existing software always run as global tasks that can be executed on any core. If synchronization problems develop when you run the software, you can rely on the following solutions for binding the application to a core:

- Binding the application with `CORE`
- Binding selected tasks with `CORETASK`
- Binding selected tasks with `RmBindTask`

Fast Timer Ticks

Programs using the `FastTimerTick` functions must be regenerated in RMOS3 V3.50.

8.3 Interrupt processing in RMOS3

Interrupts

Interrupts serve for synchronization of the driver states and hardware, meaning that the hardware reports a specific state to the software (e.g., that a character can be read). Each driver that is synchronized by means of interrupts contains at least one and possibly several entry points for jumps from an interrupt routine. It is necessary to provide several different entry points, for example, if the driver operates several controllers of different types (e.g. timer and V.24 controllers). The nucleus loads the UCB address to the EBX register prior to the start of interrupt handling by the driver.

Interrupts are processed without forcing the RUNNING task into the READY state. Only the necessary CPU registers are retrieved from the stack of the RUNNING task. Execution of the RUNNING task is resumed on return from the interrupt routine. It is also possible for extensive operations to replace the currently RUNNING task and transform the interrupt handler into a system process.

Note

Segmented programs that use interrupts must be executed at PL0.

Expected and unexpected interrupts

Depending on the status of the driver or unit, a distinction is made between expected and unexpected interrupts.

Expected interrupts, for example, signal termination of the processing step by the hardware and initiate the next processing step or a status transition at the driver.

An unexpected interrupt (e.g., no request pending for this unit) is processed according to the requirements of the unit that caused the interrupt. If the unexpected interrupt does not make sense for the unit, an error may have occurred to which the driver has to respond (e.g., simply by ignoring the interrupt). However, if an unexpected interrupt does make sense, a typical reaction of the driver may be to save a character to an internal buffer or to start a task for processing the event.

8.3.1 Basics of interrupt processing

Interrupt in 80x86

All 80x86 processors provide two interrupt inputs, namely the

- INTR, Interrupt Request
- NMI, Non-Maskable Interrupt.

Only the interrupt at input INTR can be disabled or enabled by setting an interrupt flag.

Input INTR is enabled by executing the `STI` command (in Assembler) or the `enable()` function in "C".

The reverse `CLI` (Assembler) or `disable()` ("C") commands disable the interrupt input.

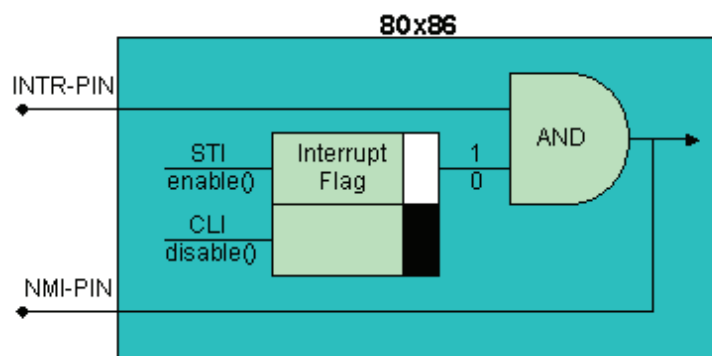


Figure 8-9 Interrupt processing on the 80x86 processor

Interrupt processing by the interrupt controller

Hardware interrupt requests are managed by the interrupt controller. This controller can be operated in PIC mode (Programmable Interrupt Controller), or in APIC mode (Advanced Programmable Interrupt Controller).

PIC mode

On a PC architecture with PIC, two interrupt controllers of the type 8259A are usually cascaded in series. These are the

- Master Programmable Interrupt Controller (MPIC) and
- Slave Programmable Interrupt Controller (SPIC),

whereby the SPIC is permanently connected to interrupt input IR2 of the MPIC.

The inputs of the interrupt controllers are assigned a PIC-internal, fixed priority:

- IR0 has top priority
- IR7 has lowest priority.

This opens the following scenario for cascading the interrupt controllers:

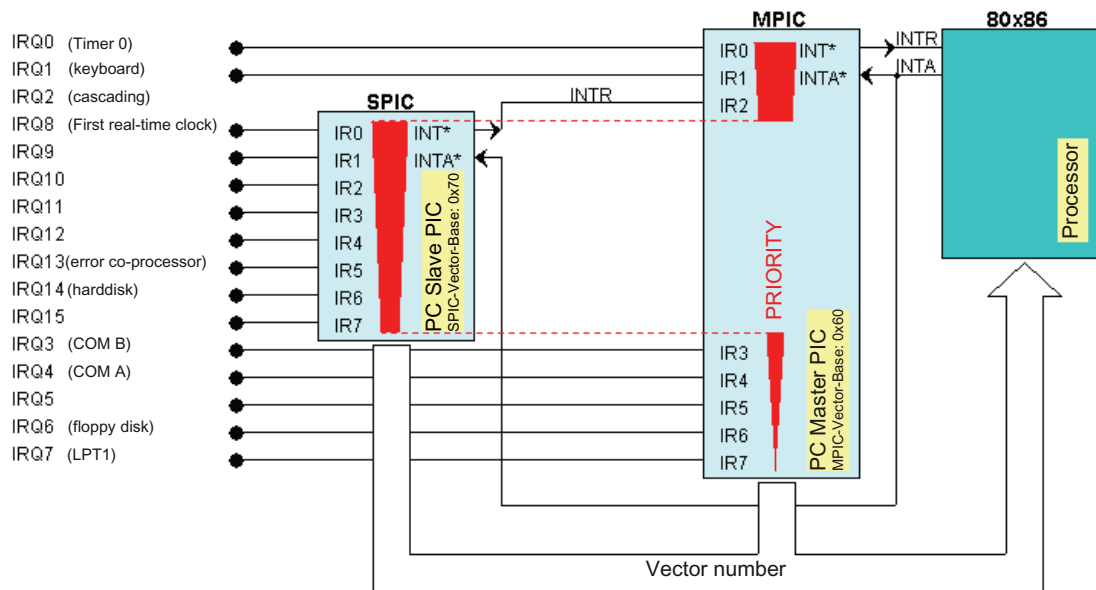


Figure 8-10 Cascading the interrupt controllers

APIC mode

PC architectures of the latest generation support 24 interrupt inputs. The allocation of the 16 lines from IRQ0 to IRQ15 remains as shown in Fig. 8-10. The IRQ16 to IRQ23 inputs are used for on-board-PCI devices of the CPU (e.g., USB or LAN) or for PCI devices on the PCI bus.

Individual masking of inputs

In addition to the interrupt disable or enable of the processor, the inputs of the interrupt controllers can be masked and re-enabled separately. This is made possible by means of the interrupt mask register of the interrupt controller and handled automatically by the RMOS3 operating system during installation of interrupt routines and drivers.

Interrupt processing in PIC mode

Based on the example of the MPIC, interrupts are processed in the following steps:

1. An interrupt set at the **IR** inputs of the PIC is transferred to the CPU (provided it is not disabled) by triggering the **INTR** signal.
2. With enabled INTR pin, the CPU completes execution of the current command and responds to the PIC request with the **first INTA# pulse**.
3. The PIC then transfers the programmed **vector number** that corresponds to the IR pin to the data bus (**D7 to D0**).
4. The CPU reads this vector number at the **second INTA# pulse**.

5. The CPU reads the address of the Interrupt Service Routine from the interrupt vector table (or **interrupt descriptor table**) and then executes this routine.
6. At the end of the corresponding Interrupt Service Routine (no later than at the I to S state transition) the CPU must return an **EOI command** (End Of Interrupt) to the PIC. In RMOS3 this is controlled and executed automatically using the "X_ISSUE_EOI" routine.

Interrupt processing in APIC mode

Interrupt processing is similar to that in PIC mode, but significantly faster.

Interrupts on the ISA bus

In RMOS3, too, interrupts on the ISA bus are processed in **edge-triggered mode**. This means that the interrupt is detected by the interrupt controller at the negative signal edge on the ISA bus and processed as explained earlier.

This operating mode does not support the operation of several different interrupt sources on the same interrupt input. If interrupts are input within a short time before the PIC has received the EOI command, the interrupts received at a later time are possibly not detected and therefore be lost.

Interrupts on the PCI bus

The introduction of the PCI bus made it necessary to integrate the **level-triggered mode** of the PIC, because the PCI bus merely provides the four interrupt lines INTA#, INTB#, INTC# and INTD# for operations. As a consequence, the modules connected to the PCI bus must "share" the interrupt lines (shared interrupts). The RMOS3 Shared Interrupt Server handles the automatic change from edge- to level-triggered mode.

Example of the processing of interrupts on the PCI bus:

If two different modules sharing the same interrupt, e.g., INTA#, output the interrupt request to the CPU in rapid succession, the CPU first identifies the vector number, and therefore the interrupt routine, at the second INTA# pulse. Within this interrupt routine it is then necessary to identify the source of the interrupt. This is done by reading the module registers of the connected PCI modules. Once the interrupt source was found, the interrupt request of the corresponding PCI module must be canceled. Then, the CPU outputs the EOI command to the interrupt controller. This completes interrupt processing for the first module.

However, the interrupt request from the second module is still pending. The INTR signal is active again immediately, the CPU once again identifies the vector number, and then branches to the Interrupt Service Routine.

This mechanism is repeated until all interrupt requests are completed.

Interrupt routing on the PCI bus

PCI interrupts INTA#, INTB#, INTC# and INTD# are mapped in the CPU-PCI bridge to the interrupt requests IRQx of the AT architecture.

In APIC mode, the PCI interrupts are assigned permanently to interrupts IRQ16 to IRQ19, or IRQ20 to IRQ23.

Handling corrupted interrupt signals in PIC mode

- Each IR signal up to the negative edge of the first INTA# pulse (from the CPU) must remain "active" (low) both in edge-triggered and level-triggered PIC mode!
- If input IR is set to inactive (high) state before that, the PIC automatically generates a "spurious interrupt" at interrupt input IR7 after the CPU has responded to the interrupt request with the first INTA# pulse.
- This property can be used as "safety measure" to detect and handle corrupted interrupt signals (malfunctions).
- In the simplest case, a corresponding IR7 Interrupt Service Routine is installed, which acknowledges (ignores) the interrupt with EOI without taking further actions.
- If IR7 is already in use as standard interrupt input, the "spurious interrupt" interrupt can still be detected and processed separately by the corresponding Interrupt Service Routine:

If the IR7 interrupt originates from a connected device, the corresponding bit is set at the "In Service Register" (ISR) of the PIC, but not for a "spurious interrupt".

- If a "spurious interrupt" is triggered **while** processing the IR7 interrupt of the connected device, the corresponding bit will still be set at the "In Service Register" (ISR) of the PIC.

In this case, the IR7 Interrupt Service Routine must "remember" its currently active state, which means that the new IR7 request can only be a "spurious interrupt".

Handling corrupted interrupt signals in APIC mode

- Spurious interrupts do not exist in APIC.

8.3.2 What is an interrupt handler?

An interrupt handler is a procedure to which the system immediately branches to execute the handler after a hardware or software interrupt was triggered at the processor (hardware interrupts must be enabled accordingly).

In RMOS3, interrupt handlers may always be customized to suit special hardware requirements (e.g., reading characters in a buffer). Execution of an interrupt handler may not impair the integrity of the system. This concerns, for example, stack requirements, the duration of processing sequences, and proper conclusion of the handler.

An interrupt handler may branch to a system process that is provided task communication resources for event flags, local mailboxes, and task starts in the form of subprogram calls.

Calls triggering a wait state are not available. This means that you have to rely on the communication option in the direction interrupt handler to application task.

An interrupt handler may be implemented independent of RMOS3 data structures (e.g., for drivers). However, this requires a corresponding application task that is tuned to this interrupt handler and e.g. reads a specific buffer after an event flag was set.

Internal states of the operating system

Each code currently processed by the processor is always in one out of four operating states:

- Application state (A state)
- System state (S state)
- Interrupt state (I state)
- Disabled Interrupt state (DI state)

The status of the currently running code is linked directly to the interruptibility of code execution by other events. Each state is also linked to specific support by the operating system.

The following section shows you the meaning of each state, its operating system support, and the type of state transition.

8.3.3 DI state

DI state

Disabled Interrupt **State** of the CPU. No task or interrupt can interrupt this state. This status is set automatically for all software or hardware interrupts.

Not interruptible

At each interrupt, the processor backs up the flag register (program status word) and the return jump address to the current stack and resets the interrupt enable flag for processing further interrupts (disable interrupt).

This automatically sets the active code, namely the interrupt handler, to the DI state. The DI state cannot be interrupted. All critical timing requirements must be processed in this state. Time-critical in this context are all necessary processing steps that must be completed before the actual interrupt handler can be interrupted by a higher-priority interrupt.

Maximum of approximately 25 Assembler commands

The duration of these processing sequences in DI state may not exceed approximately 25 Assembler commands.

If feasible and useful with regard to the application to fully implement an interrupt handler based on approximately 25 commands, the handler can be terminated by means of IRET command on completion of processing by the interrupt controller. In the case of a hardware interrupt, it is necessary to re-enable the input of the interrupt controller before this command is executed (nonspecific end of interrupt). This can be achieved by calling subprogram `x_call_eoi`. This setup will now prevent the DI state from affecting any other RMOS3 state.

Exiting as soon as possible

Usually, an interrupt handler should exit the DI state as soon as possible. The maximum processing time in this state depends on the processor frequency and may not exceed 5 µs on 80486DX2/66 processors.

If an interrupt handler is unable to complete interrupt processing in DI state, it is best practice to process only the most important parts and continue processing in I state. Usually, this change is triggered immediately at the start of the handler, because sufficient time is usually available for operating the controller, even if there is an interruption by higher-priority interrupts (specified by the input at the PIC (Programmable Interrupt Controller)).

The transition to the I state is executed by a call of subprogram `x_eclipse` (see also HLL interrupt handlers). `x_eclipse` saves all registers on the stack of the interrupted program and switches to the RMOS3 system stack if this stack was not already active. Prior to exiting the DI state, it is therefore necessary to restore the registers to the contents that were valid at the start of interrupt handling.

SVC RmSetIntDIHandler

RMOS3 provides SVC `RmSetIntDIHandler` that can be used to write custom interrupt routines in DI state.

8.3.4 I state

I state

Interrupt state. Interrupts of higher priority than that of the currently processed are capable of interrupting active processing. In I state, all registers are available to the CPU. The RMOS3 system stack is made available for these interrupt handlers for processing local variables.

The transition to I state must be executed if it takes more than approximately 25 Assembler commands to process an interrupt. Time-critical operations can then be completed before the handler exits the DI state (prior to the call of subprogram `x_eclipse`). This is not the usual procedure because RMOS3 drivers immediately call `x_eclipse`. `x_eclipse` also re-enables the interrupt input of the CPU (`STI` command).

The time-based priority of interrupt handling is determined by the wiring at the interrupt inputs of the interrupt controller.

Interrupts of higher priority are re-enabled in I state

Interrupts of higher priority are enabled in I state, while interrupts of lower or equal priority are disabled (see also Programming the interrupt controller in the hardware configuration).

Maximum of approximately 100 Assembler commands

Processing of the code can always be interrupted by an interrupt of higher priority. Take into account that the duration of code processing in I state may not exceed approximately 100 Assembler commands in order to prevent a negative impact on real-time capability, which means the capability of reacting to interrupts of lower priority.

If feasible and useful with regard to the application to terminate processing in I state, the handler can be terminated by means of a jump (`JMP`) to subprogram `x_xe1`. This subprogram controls the interrupt controller and ends the interrupt routine.

The I state is already available for interrupt handlers programmed in C. Typical sequences in this state include:

- Operation of controllers (e.g., serial interfaces)
- Writing characters read in DI state to the buffers
- Internal status transitions and consistency checks. For example, the refresh of buffers, pointers, and status bits that indicate the validity of specific buffers. Check for buffer overflows, completion of a job, or checksum calculation.

In I state, all processor registers can be used. The original register values are backed up on completion of the `x_ellipse` call. The RMOS3 system stack can be used for automatic variables (the stack length is specified in the configuration). However, the stack must be cleaned before the program exits the state. The interrupt handler can also access and manipulate all control data tables and data structures. **It may not manipulate system variables of the nucleus.**

If it takes more than 100 commands to complete interrupt processing, it is necessary to switch to the S state. A transition to the S state is also necessary for the call of subprograms that may only be called in S state (e.g., to complete a job that was passed by an `RmIO` SVC). The transition to S state is executed and the interrupt handler is converted into a system process by the call of subprogram `x_systate`.

SCV calls during interrupt handling

In I state, it is possible to call any SVC if the following rules are observed:

- The SVCs called may not cause the BLOCKED state, which means that it is not permitted to wait for a message or `RmGetBinSemaphore`. "in i-state" is output in response to an SVC exception if the SVC was called from an interrupt handler.
- An SVC is not executed until the I state is exited. This means that the result of an SVC is not available.
- You should therefore refrain from using SVCs other than those which report certain operations to the operating system, for example, start task, set flag, or reset semaphore.
- Moreover, the stack may only contain the SVC parameters. This means that it is not possible to write a message to the stack and then pass a pointer to this message.
- SVC `RmIO` is not permitted in I state.

SVC `RmSetIntIHandler`

The programmer may use SVC `RmSetIntIHandler` to launch interrupt routines in I state. The SVC also enables immediate branching to the S state or to exit the I state on completion of processing. These actions are initiated internally by a call of the `x_systate` routine or by a jump to the `x_xel` label, to enable repeated processing of the same or lower-priority interrupt. Execution of the system processes or of a still RUNNING task is resumed at the jump to `x_xel`.

The call of the `x_systate` routine initiates conversion of the interrupt handler to a system process that is executed at a later time.

RMOS3 drivers

The RMOS3 standard drivers initiate the processing of an interrupt in I state. In I state, it is possible to call SVCs that are processed after exit from the I state.

8.3.5 S state

S state

System state. These represent interrupt handlers that were converted into system processes. On completion of the conversion, these system processes are transferred to a queue. All system processes take higher priority compared to user tasks. The queue of system processes is processed based on the FIFO principle (First In, First Out). The system's capability of real-time reaction to events (interrupts) is only ensured in this state, or while user tasks are being processed, because this is the only state in which all interrupt levels are enabled.

Scheduler manages the CPU

Code execution in S state differs considerably in comparison with the two interrupt states mentioned earlier. In S state it is the RMOS3 nucleus (scheduler) that manages the CPU. The nucleus sets up a self-contained data structure (SRB block) for interrupt handlers branching to the S state (by means of the call of `x_systate` or `RmSetIntISHandler`) and transfers this to a queue.

The SRB block contains all data that supports resuming of the processing tasks controlled by the scheduler at a later time. Handling of the event is then resumed by means of a system process (execution of the code in S state).

Queue of system processes

The queue of system processes is handled based on the FIFO method. This means that the first queued system process will become active. System processes in RUNNING state occupy the CPU until terminated automatically. As a consequence, all successive system processes are blocked. It follows that system processes have no priority over each other.

All interrupts are enabled

All interrupts are enabled in S state. Sequences of the RMOS3 nucleus, as well as large parts of the driver programs should be executed in S state. System processes always terminate themselves by means of the subprogram calls provided by the nucleus or by a RETURN instruction. Completion of a system process always involves a transfer of the control flow to the nucleus (scheduler). The scheduler initiates the next processing step at this point.

All other system processes (FIFO) and tasks are blocked

Processing time in S state is not unlimited, as all other system processes (FIFO processing) and all application tasks are blocked. The length of the queue of system processes is limited by the (configurable) maximum number of available SRB blocks. If the interrupt was triggered while a task was in active state, the task is reset from the RUNNING state to the READY state at the transition from the I to S state because processing of the system process always takes higher priority.

System processes can use local variables (RMOS3 system stack), system variables and a number of subprogram calls of the nucleus to conclude event processing. In this context, system variables (*XGEN*, *XUCB*, etc.) serve as parameters for the subprogram calls.

With the help of subprogram calls, it is possible to manipulate event flags, dispatch messages, start application tasks, and allocate memory from a memory pool. However, the stack must be cleaned before the system process is concluded.

SVC calls in S state

In S state, it is possible to call SVCs if the following rules are observed:

- The SVCs called may not lead to the BLOCKED state, which means it is not permitted to wait for messages or to test and set a semaphore (*RmGetBinSemaphore*).
- Task ID 0 is output if an SVC exception was triggered by the call of the SVC from an interrupt handler.
- In S state, the use of SVC *RmIO* is only permitted without wait function.

SVC *RmSetIntIHandler*

SVC *RmSetIntIHandler* supports integration of an interrupt routine in S state. This interrupt routine can also be executed immediately in S state after an interrupt was received, provided the SVC was called in I state without call of an interrupt routine.

8.3.6 A state

A state

In applications state (A state), the CPU executes the programs of the user tasks and all interrupts are enabled. The RMOS3 nucleus (scheduler) allocates CPU control to the user tasks based on ascending priority. It can never be guaranteed that another command will be executed on completion of the previous command at the next CPU cycle, because every interrupt will immediately interrupt the programs in application state.

The RMOS3 scheduler initiates task processing and enables transitions from the S to the A state (task switch). An interrupt handler or system process cannot enter the A state.

An application task is permitted its own stack. The full scope of the SVCs is available for this task. The tasks can always intercommunicate by means of calls to the operating system.

8.3.7 Overview of the operating states and RMOS3-SVCs for interrupt processing

Overview

The following figure once again highlights the relation between the states and their transitions:

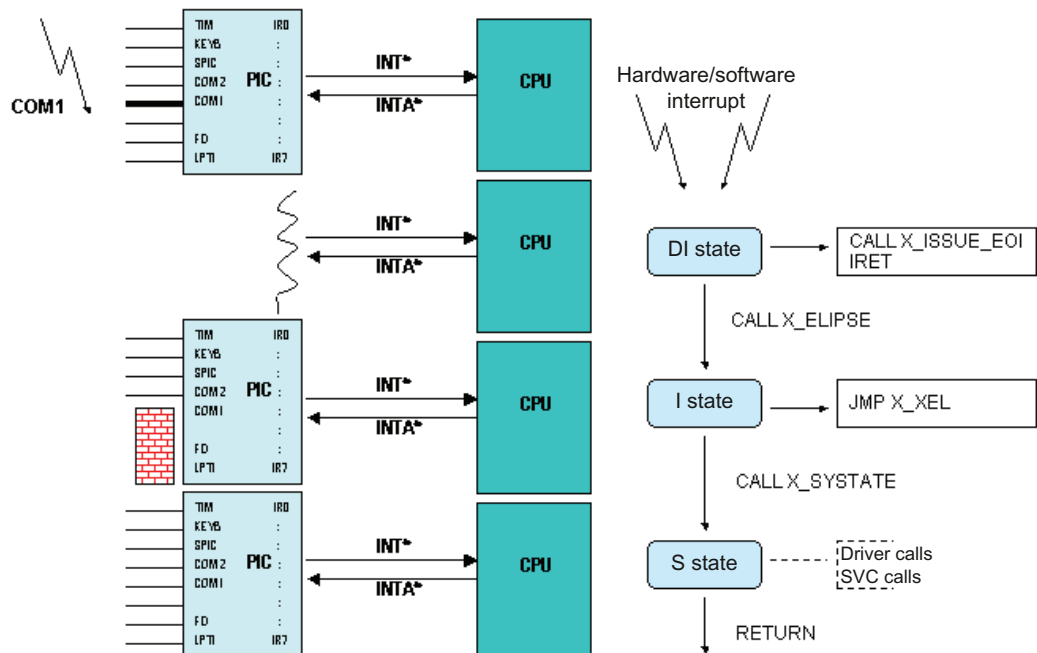


Figure 8-11 Operating states of real-time interrupts

Table 8- 4 RMOS3 SVCs for interrupt handlers

Interrupt handler	
<ul style="list-style-type: none"> • Reserve interrupt handler • Read interrupt handler • Create interrupt handler • Install interrupt handler • Restore interrupt handler 	
<ul style="list-style-type: none"> • RmReserveInterrupt • RmGetIntHandler • RmSetDeviceHandler • RmSetIntDefHandler • RmSetIntDIHandler • RmSetIntISHandler • RmSetIntTaskHandler • RmSetIntMailboxHandler • RmRestoreIntHandler 	<ul style="list-style-type: none"> • RmIntShSrv • RmInitShIntClient • RmSetShIntISHandler1 • RmSetShIntISHandler2 • RmClrShIntIsHandler

Interrupt processing on the PCI bus

Interrupt processing on the PCI bus demands three services:

- The Shared Interrupt Server contains all functions for sharing and managing interrupts.
- The Shared Interrupt Client acts as interface between the user application and the Shared Interrupt Server. The client provides the option of implementing and deleting interrupt service routines in the server.
- The PCI scanner enables identification of individual or all modules on the PCI bus.

Shared Interrupt Server

The RMISHSRV.DRV driver initializes the PCI Shared Interrupt Server, enters the driver in the RMOS3 catalog, and outputs access information for the client in a mailbox.

The Interrupt Server is loaded at the startup of the nucleus:

- `RMISHSRV.DRV` system program for dynamic call

The system may only contain one Interrupt Server.

Shared Interrupt Client

The functions of the Shared Interrupt Client enable access of an RMOS3 task to the API of the Shared Interrupt Server. `RMISHCLI.LIB` must be linked to each RMOS3 task that uses this library.

The Shared Interrupt Client is initialized by the call of `RmInitShIntClient`. This initiates the transfer of access information from a defined mailbox to the server. Further client functions cannot be executed until this transfer is completed.

We differentiate between two interrupt handling modes:

- In mode 1 (compatibility mode, `RmSetShIntISHandler1`) in addition to the already available I and S handler routines, the transfer includes a user-defined `GetIntReqState` routine that forms the basis for making the decision of whether or not the handlers should process active interrupts. This allows you to reuse existing I and S handler routines.
- In the faster mode 2 (Fast Mode, `RmSetShIntISHandler2`), only the I and S handler routines are transferred to the RMOS3 operating system, similar to the standard RMOS3 function `RmSetIntISHandler` for edge-triggered interrupts. Whether or not the active interrupt addresses the handler must be queried automatically by the I handler routine.

`RmClrShIntISHandler` calls a server function that deletes the corresponding interrupt handler entry from the server, regardless of the entry mode. Once the last entry for this interrupt has been removed, the default RMOS3 interrupt handler of RMOS3 will be installed.

PCI scanner

The PCI scanner functions (`RmPciSearchFunction` and `RmPciSearchSubFunction`) enable an RMOS3 task to scan the bus for all connected PCI devices, or to run an explicit search for PCI devices with known vendor ID and device ID, or vendor-ID, device ID, subsystem VendorID and subsystem ID (refer to the technical description of the respective module).

The configuration spaces of these devices or functions (with regard to multifunctional devices) are read and saved to a `PCI_CONFIG` structure. The order of entries conforms to the PCI standard.

The interrupt client needs the interrupt numbers from the `PCI_CONFIG` structure, which are assigned to the devices.

8.4 Task communication/coordination/synchronization

Term

The term task communication denotes all mechanisms available for tasks to explicitly manipulate one or several other tasks. Task communication can also be divided into:

- Means for communication
- Means for synchronization and coordination

In this context, the term communication denotes the transfer of messages to other tasks.

The means for task synchronization and coordination are usually referred to as measures that manipulate the status of tasks.

The necessary implementation and type of task communication are derived from the application to be realized.

An intelligent machine control system, for example, could be realized by two tasks. One task controls the terminal for the operator and processes the inputs of this operator (Human Interface), while the second task controls the machine.

Both tasks need to exchange their information. The control task needs specifications for machine control, while the HMI task needs information on the machine state.

The split into two tasks is useful for the following two reasons:

- The control process and regulating process may not be influenced by inputs or processing of the HMI task; both tasks must be capable of processing their jobs asynchronously.
- the second reason is found in the maintenance friendliness of the software: One task only contains the program for machine control and the other only the program for processing the HMI, while both intercommunicate via a precisely defined interface. This also means that the tasks may be created by different programmers.

Task coordination and synchronization may prove necessary if several tasks directly and without driver support access a hardware port, for example, a V.24 interface. In order to prevent output of an illegible scramble of data (theoretically, the scheduler could be activated after each character output), the hardware port must be reserved for the duration of the current output. The reservation is revoked and the port is released for use by other tasks on completion of the output procedure.

The period between reservation and release of the hardware port, during which one of the tasks is exclusive user, is referred to as "critical range".

Mechanisms

RMOS3 provides an extensive range of SVC mechanisms for task communication, which differ in terms of the length of information, speed, complexity and purpose, while partially overlapping.

Task communication is frequently associated with waiting for the partner task. Depending on whether this waiting period expires in RUNNING/READY state (depending on the scheduler), or in BLOCKED state, we speak of an active or passive waiting. Active waiting, or polling, is of disadvantage in multitasking operating systems due to the load on computing time that reduces system throughput.

The communication means provided in RMOS3 are outlined in the following section.

Table 8- 5 Task communication/coordination/synchronization

Task communication/coordination/synchronization				
Event flags	Semaphores	Local message traffic	Messages	SpinLocks
<ul style="list-style-type: none"> • Create, delete event flag group • Set event flags; set on expiration of a time interval • Test event flags • Reset event flags 	<ul style="list-style-type: none"> • Create, delete semaphores • Test and set semaphores • Release semaphores 	<ul style="list-style-type: none"> • Create, delete mailbox • Send mail; send, send with delay, receive • Cancel delayed send job • Specify limit for mailbox entries 	<ul style="list-style-type: none"> • Create, delete message queue • Send, receive message • Set limit for message entries 	<ul style="list-style-type: none"> • Initialize SpinLock • Set SpinLock (with/without interrupt disable) • Release SpinLock (with/without interrupt enable)
RmCreateFlagGrp RmDeleteFlagGrp RmSetFlag RmSetLocalFlag RmSetFlagDelayed RmGetFlag RmResetFlag RmResetLocalFlag	RmCreateBin Semaphore RmDeleteBin Semaphore RmGetBin Semaphore RmReleaseBin Semaphore	RmCreateMailbox RmDeleteMailbox RmSendMail RmSendMailDelayed RmReceiveMail RmSendMailCancel RmSetMailboxSize	RmCreateMessageQueue RmDeleteMessageQueue RmSendMessage RmReadMessage RmSetMessageQueueSize RmGetStatus MessageQueue	RmInitSpinLock RmGetSpinLock RmGetSpinLockIRQ RmReleaseSpinLock RmReleaseSpinLock IRQ

8.4.1 Communication and coordination by starting a task

Start of a task

This means of communication is closely related to task management. The message consists of the task start by another task by means of SVCs (`RmStartTask`, `RmQueueStartTask`) and, in the best case, contains a predefined meaning for the started task.

If the task to start is not in DORMANT state, it is also possible to add the start request to a queue (`RmQueueStartTask`).

All three SVCs provide a wait option that blocks the calling task until the started task is completed. This also enables interactive notification of the tasks.

Communication based on the start of a different task is time consuming compared to the information content of the message. Usually, the task start SVCs are therefore used only for a single start of the tasks.

Transition to READY state

Communication through task transition from BLOCKED to READY state

A task that has used a time-related SVC (`RmRestartTask` and `RmPauseTask`; `RmRestartTask`: end task and restart on expiration of the time interval) to set itself to the BLOCKED state may be returned to the READY state by the SVC `RmResumeTask` of a different task. The calls of `RmPauseTask` or `RmRestartTask` on the one hand, and `RmRestartTask` on the other, are requisite for activation of an information flow.

8.4.2 Communication and coordination by means of semaphores

Semaphores

Semaphores represent resources that are used to implement reservations of critical areas (e.g., shared data areas).

A critical area denotes an action that may only be executed by a single task.

This action has a precisely defined start and end. A task that never exits this area will block access to this area by other tasks. Reservation of devices, or the control of access to shared data structures by tasks represent such examples.

Let us assume that a lower-priority task reserves a device and is continuously blocked by a higher-priority task. This means that the lower-priority task is not allocated any computing time and the device will not be available for any other task.

The beginning of the critical area is marked by the reservation of the distributed I/O device. The end of the critical area is marked by the cancellation of the reservation.

RMOS3 supports binary semaphores. The semaphore is a data structure for which only the `RmGetBinSemaphore` and `RmReleaseBinSemaphore` operations are approved.

`SVC RmGetBinSemaphore` verifies and disables access to the critical area by other tasks. Once the critical area is released, the call immediately returns and the task can access the critical area; otherwise, the task is set to the BLOCKED state until the critical area is released.

`SVC RmReleaseBinSemaphore` releases the critical area. If binary semaphores are used, the critical area cannot be entered by more than one task.

Note

The consequence is that if a task currently present in the critical area requests an `SVC RmGetBinSemaphore` for the same semaphore, it will be disabled permanently.

The following figure demonstrates communication of two tasks by means of shared memory or access to a shared device.

In the state shown, only task 2 is permitted access to the device or shared memory, because it was allocated the semaphore with `RmGetBinSemaphore`. Task 1 has previously released the device or shared memory with `RmReleaseBinSemaphore`.

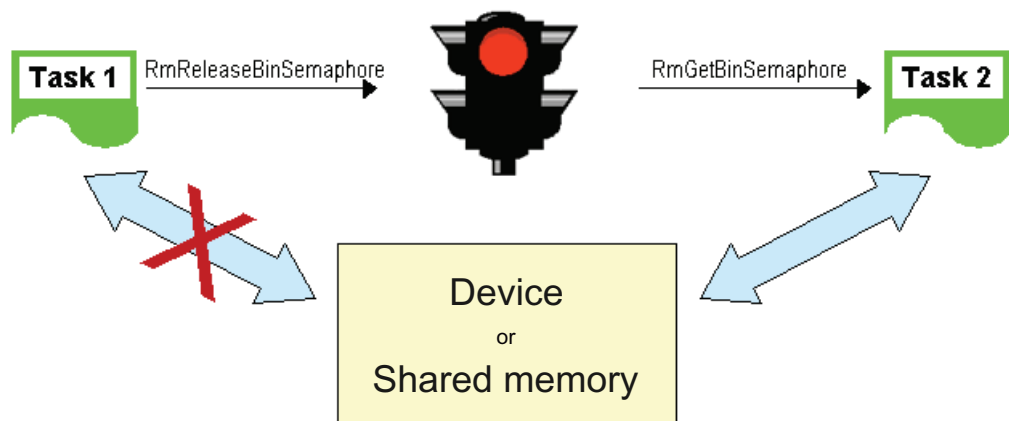


Figure 8-12 Using semaphores for task communication and co-ordination

8.4.3 Communication and coordination by means of spinlocks

Spinlocks

A spinlock is a programming technique for the synchronization of tasks or program elements running simultaneously on several cores.

Spinlocks and semaphores are quite similar with regard to their functionality. Semaphores wait in passive mode for the release of the resource. This means that the task that requested the semaphore changes to the BLOCKED state, and the scheduler can then assign the core a task from the ready task queue; this task will change to the RUNNING state.

Spinlocks wait in active mode for the release of a requested spinlock. The check is performed at cyclic intervals using processor command "pause". Spinlocks will therefore consume computing time. A core that is not in possession of the spinlock is halted by 100%, which means that it is no longer possible to execute any task on this core. The core currently in possession of the spinlock is also decelerated by approximately 20% (with two cores).

Compared to semaphores, Spinlocks are considerably easier to handle for the operating system.

Spinlocks are useful in the following situations:

- Semaphores are not available (e.g., in interrupt handlers).
- Execution of the program element to protect takes only a few nanoseconds.
- Performance of the program element to protect is to be increased (spinlock: 5 commands (approximately 5 ns), semaphores 500 commands (approximately 500 ns)).
- The sequence to protect contains no SVC calls.
- A code sequence previously protected with `enable/disable` is to be protected against interruption by another core.
- Access to an IO module is to be protected.
- Global variables are to be protected against shared access by tasks running on different cores.

Interrupts should be disabled prior to the call of the spinlock to prevent extension of the dwell time by an interrupt after the spinlock was received. If using the `RmGetSpinLock` function, you need to disable the interrupts by yourself (with `disable`). The alternative `RmGetSpinLockIRQ` function supports automatic disabling of the interrupts.

`RmGetSpinLock` can be nested in RMOS3. This means that the core in possession of the spinlock may call `RmGetSpinLock` several times without being blocked. Accordingly, `RmReleaseSpinLock` must then be called several times until the spinlock is released again. The spinlock structure has an internal counter that logs the number of calls of the spinlock on the same core (e.g., by subprogram calls also requesting spinlock). The counter has a maximum value of 255. An overflow to 0 is triggered once this value is exceeded. `RmReleaseSpinLock` has no effect in this case.

Spinlocks should not be used at task level. If nonetheless used, the task may not change to another core under any circumstances.

Spinlocks can only be used within a loaded application, because the code is linked directly. The code is linked once again for each loaded application.

Since spinlocks are no operating system calls, the profiler will not log the corresponding calls.

The status of the interrupt flag is saved, the interrupts are disabled, and the spinlock is requested at the call of `RmGetSpinLockIRQ`. The interrupt is only re-enabled with `RmReleaseSpinLockIRQ` on the condition that it was enabled prior to the call of `RmGetSpinLockIRQ`.

The spinlocks can also be used on a single-core system. However this does not make any sense, because the interrupt disable will prevent the processor from executing anything else.

The following figure demonstrates access to a shared variable.

In the state shown, only task 2 is allowed access to the variable, because it was allocated the spinlock with `RmGetSpinLock`. An attempt of task 1 to obtain possession of the same spinlock will fail.

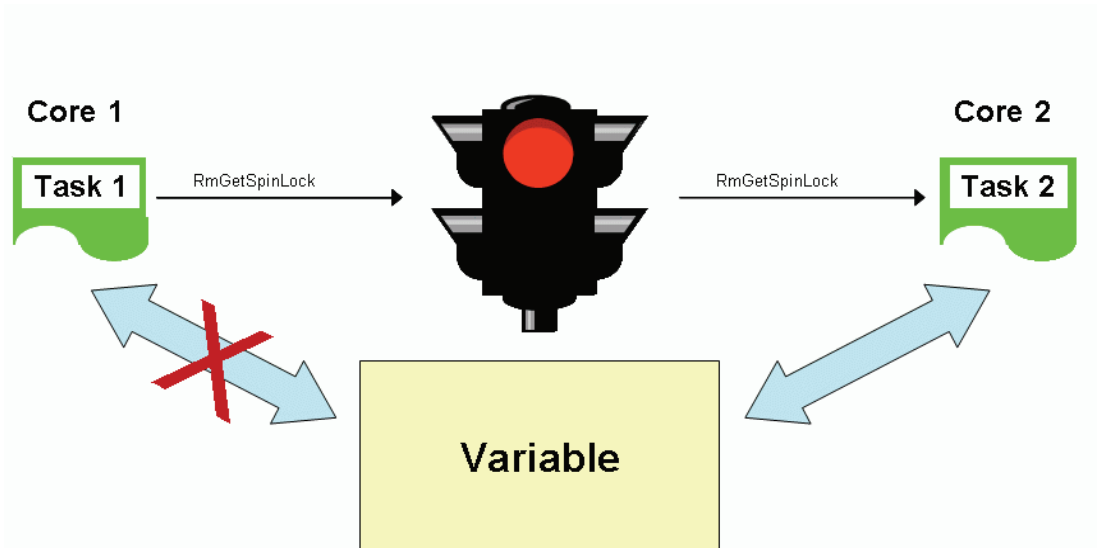


Figure 8-13 Using spinlocks for task communication and co-ordination

8.4.4 Communication by means of event flags

Event flags

An event flag, in short flag, is a data structure that is implemented as single bit in memory. All flags are collected in flag groups, each with a length of 32 bits and separate ID. The system provides local and global flag groups.

A single local flag group is generated as soon as you create a task. This local flag group is always assigned flag ID zero. Global flag groups must be configured in static mode, or be generated in dynamic mode, and are assigned flag IDs unequal zero.

Each task can access all global flag groups, for example, by means of SVCs `RmSetFlag` or `RmGetFlag` (flag ID unequal zero), as well as all local flag groups by means of SVCs `RmSetLocalFlag` or `RmGetLocalFlag` (using the corresponding task ID).

RMOS3 provides the following SVCs for flagged operations:

- Set event flag (`RmSetFlag`, `RmSetLocalFlag`)
- Set flag on expiration of a time interval (`RmSetFlagDelayed`)
- Reset event flag (`RmResetFlag`, `RmResetLocalFlag`)
- Testing event flag (`RmGetFlag`). Testing of the event flag can be linked with waiting for the flag. While waiting, the task is set to BLOCKED state (passive waiting). The waiting time can be limited to a certain extent. On return from the waiting for flag state, it has to be checked whether the flag was set, or whether the waiting time has expired.

A task usually sets an event flag if a specific condition is met, or influenced by a system process when an event (interrupt) was triggered (interrupt). In this context, a flag corresponds to a binary message with predefined meaning as specified by the application. Flags form an elegant and efficient solution for the exchange of binary messages. However, the SVCs will not be queued. An SVC `RmSetFlag` for a set flag has no effect, which means that the operating system will not store this request until the flag is reset. Tasks always communicate by means of two SVCs. An active SVC that manipulates the flag state (`RmSetFlag`, `RmResetFlag`, `RmSetFlagDelayed`), and a responding SVC (`RmGetFlag`).

The following figure shows how Task 1 generates a global flag group, which can be addresses using FLAG-ID 1. Task 3 waits for the event flags 8, 14 and 28 of the flag group to be set. Task 2 waits for the incoming flag 2.

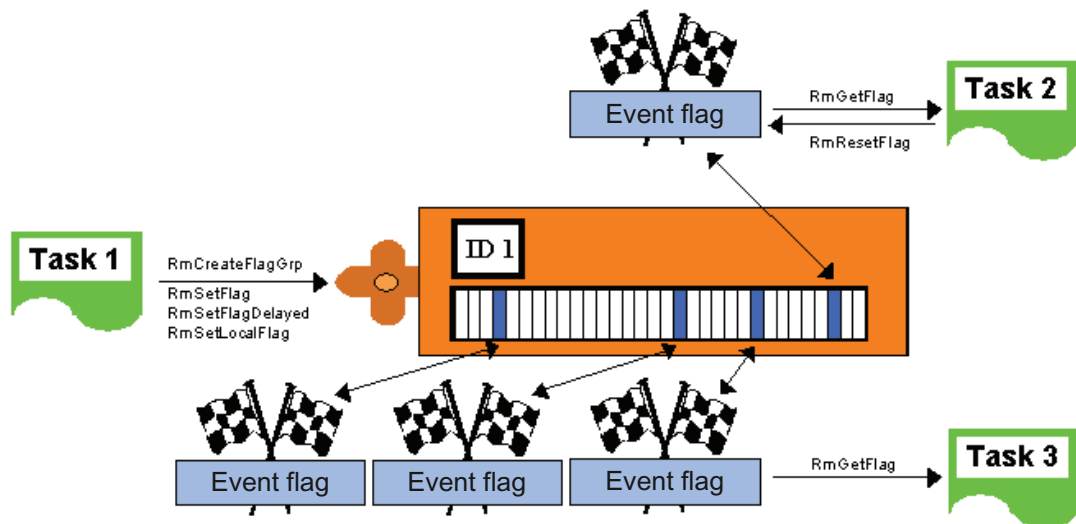


Figure 8-14 Communication by means of event flags

8.4.5 Communication via local mailboxes

Local mailboxes

A mailbox in RMOS3 represents a data structure that corresponds to a dynamic, linear list of the type message. A message always has a length of 3 words (12 bytes), with content determined by the application (e.g. a pointer and a length, or simply 12 characters). The queue has no static length (therefore dynamic) and is extended by `SVC RmSendMail`, or reduced by `SVC RmReceiveMail`. `SVC RmSendMail` copies a message of a length of 12 bytes to the mailbox. `SVC RmReceiveMail` copies the message from the mailbox to the memory of the calling task.

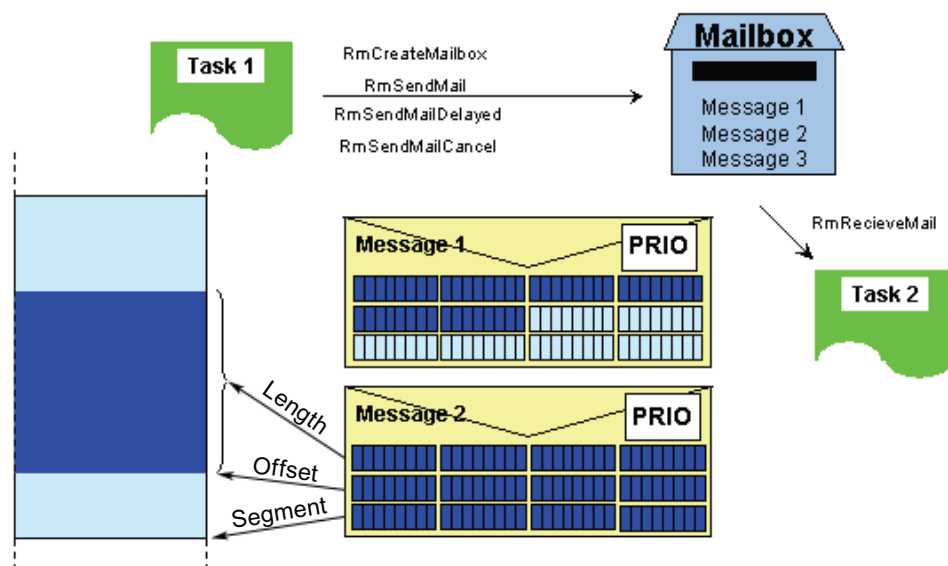


Figure 8-15 Communication via local mailboxes

All `RmSendMail` calls can be prioritized, which means that the message is written to the mailbox according to its priority. `SVC RmReceiveMail` always fetches the message having highest priority from the queue. Both SVCs can be called with the "wait" parameter.

Because all tasks are permitted access to the mailbox, which means execute `RmSendMail` and `RmReceiveMail` calls, the user task, or the application programmer are responsible for allocating a specific meaning as communication point to each mailbox.

Task 1 shown in the figure above, for example, could create a mailbox and send cyclic messages to this mailbox. Message 1 could contain a message with a length of six bytes, while message 2 contains a pointer to shared memory. Task 2 can use `SVC RmReceiveMail` to successively fetch and therefore remove the messages from the mailbox (starting with message 1).

8.4.6 Message communication

Messages

RMOS3 tasks can exchange information in the form of messages. A corresponding message queue can be generated for each task. The message queue is a FIFO memory to which other tasks may copy messages.

A message must contain at least a 32-bit parameters and a FAR pointer. The 32-bit parameter (message ID) specifies the message number or code. The FAR pointer contains additional message-specific parameters, such as a pointer to a parameter block.

SVC `RmSendMessage` can be prioritized, which means that the message is added to the message queue according to this priority. SVC `RmReadMessage` always fetches the message having highest priority from the queue. Both SVCs can be called with the "wait" parameter.

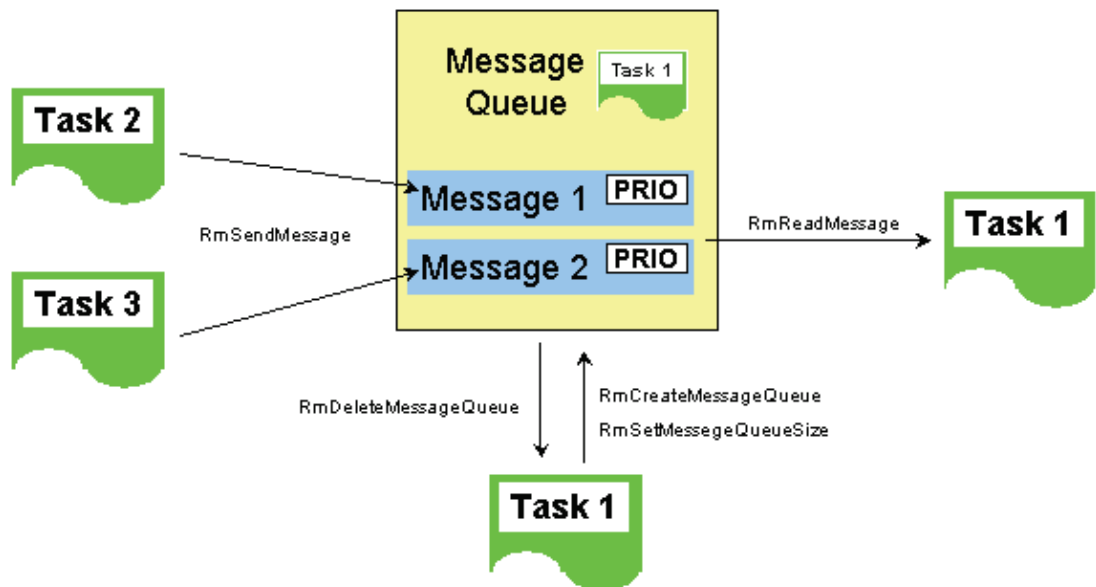


Figure 8-16 Communication by means of message queues

8.4.7 Communication by means of shared data areas

The seemingly simplest and most efficient solution for task communication is to use a shared data area (e.g. shared data arrays, buffers, shared memory) for two or several tasks.

RMOS3 always allows you to use this variant. However, this renders you responsible for specifying the valid areas and the period in which these are valid and released for read/write access, and for reporting these settings to the partner tasks. The respective programming procedure is always application-specific and never a trivial matter. This means that you will have to carefully consider whether or not to benefit from any gain in speed that may be expected.

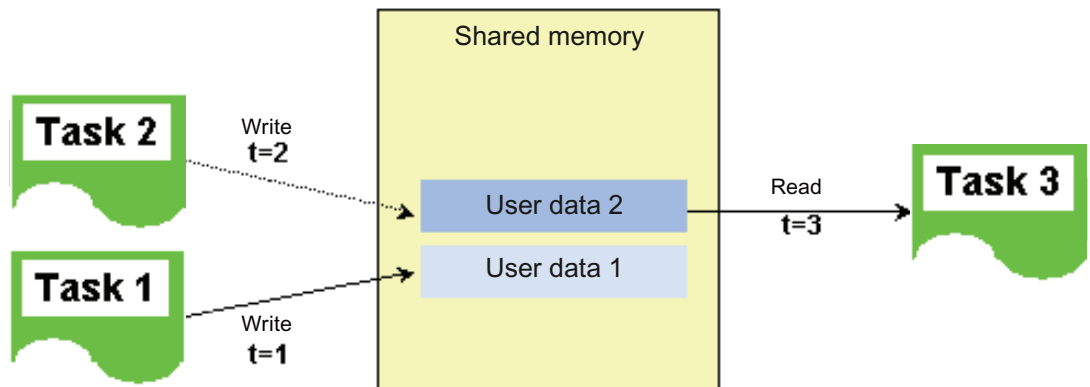


Figure 8-17 Communication by means of shared memory

8.5 Resource management

RMOS3 provides numerous system calls for resource management. The following overview lists the most important SVCs. For more information on calls, refer to Reference Manual Part III.

Table 8- 6 Resource management

Resource management					
Resource catalog	Time management	Memory management	Descriptor management	Drivers	System
<ul style="list-style-type: none"> Enter, find, delete cataloged resources List catalog entries 	<ul style="list-style-type: none"> Set, read the date/time of the software clock Read absolute time Set, read the date/time of the real-time clock Query system clock rate 	<ul style="list-style-type: none"> Create, delete memory pool Request memory space, release individually, release all Change memory space length Reserve memory space Query information about memory pool Determine memory space length Address physical memory 	<ul style="list-style-type: none"> Create, delete, edit descriptors Determine linear address Determine physical address 	<ul style="list-style-type: none"> Register, suspend drivers Mount unit Define new driver Request I/O operation 	<ul style="list-style-type: none"> Decode RMOS error message Read SRB Configure operating system
RmCatalog RmGetEntry RmGetName RmUnCatalog RmList	RmSetSystemTime RmGetSystemTime RmGetAbsTime RmSetHWClockTime RmGetHWClockTime RmGetSystemPeriod	RmCreateMemPool RmDeleteMemPool RmAlloc RmMemPoolAlloc RmReAlloc RmFree RmFreeAll RmExclude RmGetMemPoolInfo RmGetSize RmMapMemory	RmCreateDescriptor RmDeleteDescriptor RmChangeDescriptor RmChangeDescriptor Access RmGetLinAddress RmGetPhysAddress	RmCreateDriver RmSuspendDriver RmCreateUnit RmResumeDriver RmIO	RmDecode RmGetSysB RmSetOS

8.5.1 Resource catalog

All resources in RMOS3 are entered in the resource catalog. All tasks, drivers (devices), devices (units), semaphores, flags, etc. are cataloged with their name. The following resources are available:

Table 8- 7 Overview of resources

Resources		Resource ID	Max. number
Task	RM_CATALOG_TASK	0	2048
Device	RM_CATALOG_DEVICE	1	255
Pool	RM_CATALOG_POOL	2	8
Semaphore	RM_CATALOG_SEMAPHORE	3	4096
Event flag	RM_CATALOG_EVENTFLAG	4	256
Local mailbox	RM_CATALOG_LOCALMAILBOX	6	256
Miscellaneous	RM_CATALOG_MISC	7	65536
User-defined	RM_CATALOG_USER	8	65536
Unit	RM_CATALOG_UNIT	10	255
Message	RM_CATALOG_MESSAGE	11	2048

The resource catalog allows you to run explicit searches for entries without knowing the ID, for example, of a corresponding task or interface, because the RMOS3 system always assigns the IDs dynamically while retaining the names of catalog entries.

`RmGetEntry` or `RmGetName` can be used to determine the ID of a selected catalog entry for further processing. This solution, for example, enables the exchange of message between several tasks sharing the same mailbox, provided the mailbox name is known to all tasks.

`RmCatalog` or `RmUnCatalog` serves to create and remove a catalog entry. The MISC or USER resources are available for making user-specific entries. You can make such an entry, for example, to report a segment address to other tasks, by setting the type to USER, creating a catalog entry such as "DATA", and publishing the segment and offset address using resource ID "ID" and extended ID "IDEX".

`SVC RmList` is used to list all catalog entries in successive order and enter these in an available structure of the type `RmEntryStruct`. The procedure starts at the initial call of `RmList` with `pIndex = 0`. All further calls of `RmList` must be continued with the number of entries found with `pNumEntries` to obtain a complete catalog listing.

You may use this function, for example, to query the number of cataloged tasks.

8.5.2 Time management

Central, internal operating system clock rate

The internal operating system clock rate is generated by setting interrupts of a timer block at the input of the interrupt controller (IRQ0).

Set the time interval to the next interrupt as integer multiple of a millisecond in your configuration (minimum time interval). You can set a time interval between two system clock rates within the range from one millisecond to several hours (usually from 1 to 10 milliseconds).

If you select a short system clock rate, e.g., one millisecond, management load for processing the timer interrupt will amount to 1% to 5% of total execution time depending on the processor clock, and therefore curb the time slice available for task execution. A suitable system clock rate should be derived from the time-related requirements of the application. The system clock rate should amount to approximately 1/10 of the minimum time interval required by the application (e.g., for pauses and timeouts).

Assuming the application system demands a minimum pause of 100 ms, you should set a system clock rate of 10 ms ("fluctuation" of the actual pause will then amount to a maximum of 10 ms, which is equivalent to 10%).

Usage

Based on the system clock rate, RMOS3 generates the monitoring time for all `RmPauseTask` SVCs, the refresh time of the Round-Robin counter, and a software clock. An additional real-time clock in the system can be integrated in time management by means of further calls (cf. call of `RmInitOS`). In RMOS3, the time of a real-time clock is also referred to as global time (in contrast to the software clock). All SVCs containing a time parameter transfer a time interval, including the number of intervals. The shortest time interval amounts to one millisecond, while the longest amounts to one hour.

In RMOS3, a zero value at the parameter specifying the number of time intervals is interpreted as the minimum interval time. This period, simply said, corresponds to the waiting time for the next system clock rate.

At system startup (without real-time clock, global time), RMOS3 sets the software clock (local time) to 01-JAN-1995 00:00:00.

You can use SVCs to read and set the software and real-time clocks. Corresponding data and time functions are implemented in the SVCs. The resolution is set to one second. A year is interpreted as leap year, if the year value can be divided by 4 without remainder and the value is not an integer multiple of 100, or divisible by 400 without remainder.

8.5.2.1 Time-related system calls

Time-related system calls are handled in the following groups:

- Task processing
- Time-related task communication
- Timeout handling for system calls
- Timeout handling for devices

The section is concluded with advise on how to avoid system deadlocks.

Task processing

BLOCKED to restart

`RmRestartTask` (end task and restart on expiration of the time interval) triggers transition of the task from RUNNING to BLOCKED state. The task is returned to the READY state on expiration of the specified time interval for the restart with the parameters from TCD. You can specify the time interval that `RmRestartTask` uses as orientation in two different ways. The time is calculated based on the most recently specified start time or current time. The first setting is the preferred for a cyclic task that enters the RUNNING state at cyclic intervals. Take into account that the start time is equivalent to the time at which the task enters the READY state, but not to the time at which it was actually assigned the CPU.

Example (of a task to be restarted at cyclic intervals of 5 minutes):

The task was initially started at 11:00 h, but not completed until 11:07 h. The task is restarted immediately after it was completed, because the next restart was planned for execution at 11:05 h. If the task has completed processing before 11:10 h it must wait for its next restart. If completed after 11:10 h, the task is restarted immediately, and so forth. The system will attempt to compensate for delays in the start sequence.

READY for restart, fast timer ticks

In RMOS3, fast timer ticks functionality can be launched simultaneously for up to five tasks at a resolution of 10 microseconds. You can rely on this functionality to implement high-speed control algorithms. The common basic cycle of 10, 50, 100, 200, or 500 microseconds is set by calling `RmInitFastTick`, which installs a timer interrupt handler for a cyclic start of the tasks. The `RmInitFastTask` starts up to five tasks that were created previously with `RmCreateTask` or `RmCreateTaskEx`, and which are currently in DORMANT state, at the selected time interval. This task-specific time interval is an integer multiple of the set basic cycle. The time for cyclic start of the task is calculated based on the most recently specified start time at which the task enters the READY state.

BLOCKED to resume

RMOS3 can execute `RmPauseTask` to set a task for the duration of a specific period to the BLOCKED state. You may specify the pause length or permit RMOS3 to set a minimum time interval. The task is returned to the READY state and resumed on expiration of the time interval.

A task can call `RmResumeTask` (cancel pause) to return a BLOCKED task to the READY state. `RmResumeTask` has no effect on a task that is not in BLOCKED state.

Time-related task communication

Delayed messages

The following SVCs initiate delayed task communication:

`RmSetFlagDelayed`,
`RmSendMailDelayed`.

Timeout handling for system calls

Task-specific timeout values

You may set task-specific timeout values for the SVCs listed below. These SVCs are in particular:

`RmAlloc`,
`RmMemPoolAlloc`,
`RmGetEntry`,
`RmGetFlag`,
`RmSendMail`,
`RmReceiveMail`,
`RmSendMessage`,
`RmReadMessage`,
`RmGetBinSemaphore`.

Timeout settings

The SVCs are transferred with parameter `TimeOutValue`. This parameter may assume the following values:

Value	Range	Meaning
<code>RM_CONTINUE</code>		Do not wait. The function returns immediately.
<code>1 ... RM_MAXTIME</code>		Specifies the time interval in ms. The function returns event-triggered, or on timeout.
<code>RM_WAIT</code>		Wait until the event was triggered (no timeout)
<code>RM_MILLISECOND (ms)</code>	<code>1 ... 2^31</code>	Waits for <code>ms</code> milliseconds.
<code>RM_SECOND (sec)</code>	<code>1 ... 2^31 / 1000</code>	Waits for <code>sec</code> seconds
<code>RM_MINUTE (min)</code>	<code>1 ... 2^31 / 60 000</code>	Waits for <code>min</code> minutes.
<code>RM_HOUR (hour)</code>	<code>1 ... 2^31 / 3 600 000</code>	Waits for <code>hour</code> hours.

You may logically link the values for hours, minutes, and seconds:

`RM_HOUR (hour) + RM_MINUTE (min) + RM_SECOND (s)`.

Timeout handling for devices

Timeout monitoring for devices

In order to safeguard completion of a requested I/O operation and proper operation of the system on failure of a distributed I/O device, RMOS3 offers you the option of defining a time interval, which means device timeout, in all device software drivers, which represents the maximum permitted time for execution of an I/O operation. The timeout interval is decremented at the start of every I/O operation and, if this I/O operation was not completed within this time, the device driver takes control and returns a corresponding status message to the requesting task. You may set any timeout interval and customize it to suit requirements of the respective control block.

Handling deadlocks

Avoiding deadlocks as a result of timeout

It may so happen in certain scenarios that a (system) deadlock prevents a task from exiting the BLOCKED state. This interactive deadlock may occur, for example, if Task A is in possession of semaphore X and waiting for semaphore Y, while Task B is in possession of semaphore Y and waiting for semaphore X. You can avoid such scenarios by means of the timeout handling routine for SVC `RmGetBinSemaphore`.

8.5.3 Memory management

Differentiation based on functional aspects

Along with an organization by RAM, dual-port and EPROM area, you may also organize memory space based on functional aspects. In RMOS3, you can generally split the allocated memory of a computer system into three areas.

Operating system code and data structures:

This area is occupied by the operating system and contains the data structures and the operating system code that are necessary for operation.

Code, data and stack areas of tasks:

This area is occupied by user and utility tasks with the respective code, data and stack spaces. Firstly, these areas are determined by the compiler and length of the code and data segments, and secondly by the configuration (stack size for the individual tasks).

Memory areas for dynamic data structures, memory pools:

The RMOS3 system calculates the amount of free RAM resources at system startup. RMOS3 manages this RAM as HEAP pool. You may also define additional memory pools. The HEAP pool can always be addressed with ID -1, while it is always necessary to specify the ID for the other pools.

Free RAM serves to support the dynamic data structures of user and utility tasks of the application system.

RMOS3 provides SVCs for reserving and releasing dynamic memory structures. These reserving and releasing actions are also known as memory allocation and memory release. List elements, for example, are dynamic data structures. The entire free memory is managed in the heap. The advantage of a large pool is that because all user tasks access the same pool in a uniform way it is probably less complex to program the system.

8.5.4 Descriptor management

Protected mode and privilege levels

RMOS3 is a protected mode operating system, which means that the code, data and stack of the nucleus and tasks are protected against unauthorized access to each other. The Protected Mode mechanisms came along with the 80386 processor architecture, which enables you to check access of a program to data and code, and assign access authorizations at altogether four security levels (privilege levels).

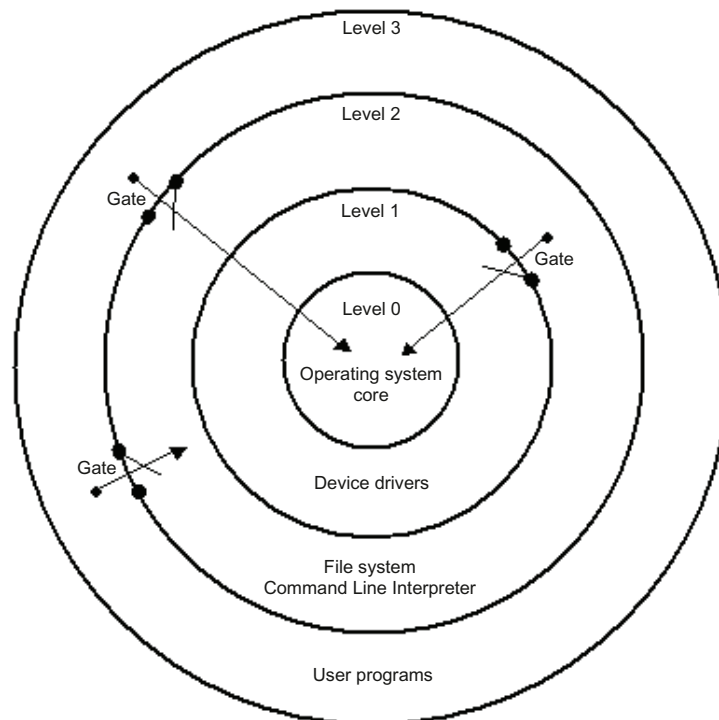


Figure 8-18 Concept of the four privilege levels in the 80386 architecture

The operating system nucleus is allocated the highest privilege level 0, while the application programs are assigned level 3 as lowest privilege level. This low level provides an excellent means for protecting the data and code of other programs and of the operating system.

Gates

RMOS3 provides so-called gates that enable the use of code and data of higher privilege levels. The gates facilitate controlled access of programs to external data and code, e.g., of the nucleus. They define gateways that can be employed by user programs, for example, to execute CRUN functions of the RMOS3 nucleus.

Addressing in protected mode

Instead of directly using the basic addresses of a selected data segment for addressing, this is handled using segment selectors in protected mode.

This segment selector is loaded into a 16-bit segment register, e.g. DS, and is used to address a descriptor, e.g. descriptor #3. This means that this segment selector selects a descriptor from its descriptor table (GDT, LDT).

This descriptor defines the properties of the selected data segment, i.e. the base address, limit and access rights, and transfers this data to the corresponding segment register cache (e.g., DS cache).

The processor validates the access rights and triggers a so-called exception interrupt if the result is negative.

The Segment Register Cache that contains the 32-bit base address of the data segment can now be used to address the base of the data segment.

In protected mode, the pointers to data segments represent FAR pointers (48-bit) that consist of a 16-bit selector and a 32-bit offset.

In addition, the Segment Register Cache returns information with regard to all segment properties the processor needs to execute its security function:

- The segment limit specifies the maximum valid offset (max. 4 GB).
- The access rights (access rights AR) specify the conditions for using a segment, for example, by tagging a segment with read-only attribute.

The physical address space and the maximum segment size are 4 GB.

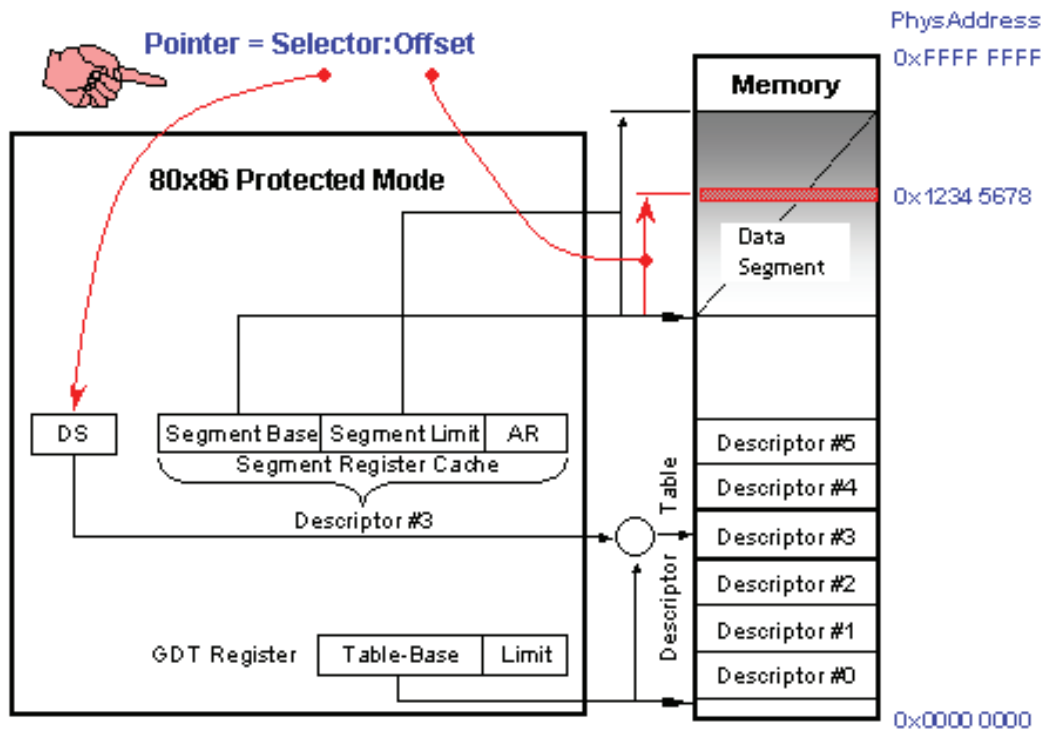


Figure 8-19 Loading a descriptor

Addressing data segments in RMOS3

In preparation to the addressing of a memory space, it is necessary to execute `RmCreateDescriptor` to generate a selector that points to the descriptor table. You can obtain the pointer to the relevant address using the function `buildpointer(selector, offset)`.

In addition, RMOS3 offers a number of other functions for descriptor management. Along with the aforementioned `RmCreateDescriptor` function, these include functions for deleting descriptors, for editing access rights, and for calculating linear and physical addresses (see the figure in section "Resource management (Page 116)").

8.5.5 Driver I/O management

I/O process

RMOS3 drivers provide support for the I/O operations of many distributed I/O devices by means of "Memory Mapped I/O" or "Direct I/O" functions.

Multiple devices

Usually, drivers are capable of operating multiple devices of the same type. User tasks initiate I/O processing by calling `SVC_RmIO`. `SVC_RmIO` contains parameters that include the driver ID (device ID) and the ID of the relevant device (unit ID).

A driver controls and manages the first of five data terminals as unit 0, the second as unit 1 and so forth, for example. The driver and device IDs depend on the configuration. RMOS3 supports up to 255 different drivers. Each driver is, in theory, capable of controlling and managing up to 255 devices.

RmIO initiates the I/O request

`SVC_RmIO` is used to initiate an I/O request and to transfer it to the queue of the corresponding distributed I/O device (device driver). The `SVC` contains the following parameters:

- Driver ID
- Device ID
- Function (read/write, etc.)
- Even flag ID/flag mask
- I/O status
- Pointer to parameter block

Once RMOS3 has accepted an I/O request (an added it to the queue), the specified event flags are reset and the I/O status is set to 0. Control is returned to the requesting task.

Synchronization with RmGetFlag

The task can synchronize its processing on completion of the I/O operation by calling an `SVC_RmGetFlag` with waiting for the flag parameter. Once the I/O request was completed (with/without error), the event flags are set so the waiting task is able to resume execution.

Return value of RmIO

The return value of `RmIO` provides information about the formal check of the call by the nucleus at the beginning of call processing. Verification of this return value alone is not sufficient to determine whether or not the I/O operation was completed successfully.

E/A status on completion of the I/O operation

The I/O status provides status information with regard to the executed I/O operation to the requesting task. The first byte represents the primary state that is reported back by the device driver, while the second byte, which represents the secondary status and provides information on the error cause, is driver-specific and/or device-specific. As agreed, a negative number in the primary status indicates incorrect completion of the I/O operation. For information on the cause of error, refer to the description of the specific device drivers.

8.5.6 System calls

RMOS3 provides system calls that can be used to decode RMOS3 error numbers and to read the System Control Block.

RmDecode

SVC `RmDecode` decodes the specified RMOS3 error number and outputs the corresponding message in plain text.

This SVC was already deployed in the `svc_sts` procedure in the examples shown in chapters "Practical section: Creating an RMOS3 task (Page 61)" and "Practical section: Testing an RMOS3 task (Page 63)" and covered in detail in chapter "SVC exception handler, status messages and test output (Page 63)".

RmGetSysB

SVC `RmGetSysB` serves to read the System Control Block. This block contains the most important system parameters, e.g the system type (RMOS2, RMOS3, ...), version, boot volume, system clock rate, etc. . Moreover, the addresses of the hardware functions are stored in this block (cf. `RmInitOS`).

For more information on the structure of the System Control Block, refer to the Reference Manual Part II, chapter 3.

RmSetOS

The `RmSetOS` call performs system configurations. These include the setting of the Round-Robin counter, output of SVC messages, and setup of the system console.

8.6 DEBUG strategies in RMOS3

Test tools

You always need to differentiate between tests of the drivers and user tasks:

You can test drivers using the low-level debugger that is integrated in RMOS3. However, this poses a risk to real-time processing if a breakpoint is triggered and causes a system standstill.

For testing and debugging user tasks, you also have the option of using the GNU debugger `rm-gdb` (via network), the resource reporter, and the RMOS3 profiler.

Debugger

The low-level debugger has the following properties:

- Sequential control (e.g., startup) and checking the status of all tasks running in RMOS3
- Option for verifying and editing memory contents
- Setting breakpoints in user tasks
- Option for verifying and editing the register contents of an interrupted task.
- Loading tasks

Resource reporter

The resource reporter task is a useful supplement for the debugger. Resource reporter facilitates the on-screen display of RMOS3 data structures and resources, or writing this information to a data volume. The reporter encompasses evaluations for tasks, device drivers, memory pools, semaphores, global event flags, programs with monitored access, message queues, and mailboxes.

Error logger task

You specify the static utility task (error logger task) in your configuration.

The error logger task transfers error messages of the operating system to the output device that is programmed in the task. The address of the output string is transferred to the task by means of EAX (offset) and EBX (segment/selector) and can be read using `getparm`. The error string has a maximum length of 255 characters and is terminated by a null character. For more information on the type of error messages, refer to chapter "Error messages/error logger task (Page 133)". The task is started by the internal operating system call `xerrlog` and can be customized to suit the respective system environment.

The error logger task is available in the `RM3BAS.LIB` library. An example that is identical with respect to functionality is available as `ERRLOG.C` file in the directory `SOURCE\ETC\CADUL`

RMOS3 profiler

The RMOS3 profiler is a utility that provides support for the following activities:

- Displaying system parameters
- Calculating load distribution
- Analyzing task activities.

You may use the results of the calculation of load distribution to optimize system runtime. It is helpful for error handling and runtime optimization to log the task activities. It may also be used to measure SVC execution times and specific program units.

For more information on using the RMOS3 profiler, refer to Reference Manual Part I, chapter 5.

Pool

Pool is a reloadable program that you can use to determine and visualize memory blocks and analyze memory distribution in the RMOS3 system.

The syntax for this program is described in the Reference Manual Part I, chapter 2.

Test and error message facilities

RMOS3 provides several test and error message facilities that are active throughout the runtime of an application.

Each essential internal initialization procedure of the system during system startup triggers the output of messages. These messages are an initial means of validating a configuration.

RMOS3 can log all SVCs that were not transferred with correct parameters in a message that is output to a configurable terminal. This message is generated independent of any SVC error evaluation in the code of the calling task.

8.6.1 Test strategies

Test tools

Test strategies and tools can be distributed to test tools that are supported by and independent of the operating system. The test tools supported by the system output corresponding system calls (SVCs).

Test tools supported by the operating system include, for example:

1. RMOS3 debugger
2. Resource reporter
3. RMOS3 debugger (target) with high-level language (host) debugger
4. Error logger task
5. SVC exception handler
6. Exception interrupt handler

A hardware emulator (ICE), for example, is a test tool used independently of the operating system.

Testing user tasks

All testing tools support the test of user tasks. The operating system supports the testing of internal system routines (e.g., custom driver developments) by means of startup messages and internal error messages using the error logger task. You need a host debugger with RMOS3 target debugger or a hardware emulator to test drivers that are available in source code.

The `rm-gdb` from the RMOS3 GNU software package (for execution in Windows) is currently available as host debugger.

In addition to the independent test tools, you can rely on the debugger, resource reporter, error logger task and the SVC exception handler to test user tasks, provided that the operating system was properly configured. The term RMOS3 application used in the following sections relates to the configured operating system with all user tasks.

The following table provides an overview of the most important performance features of the test tools:

Table 8- 8 Performance features of the test tools

Test tools	Real-time	Trace memory	Symbol processing	Source code processing	Driver test	Task test
RMOS3 debugger						
Task mode	yes	no	no	no	no	yes
Monitor mode	no ¹	no	no	no	yes	no
Host debugger with RMOS3 debugger (target)						
Task mode	yes	no	yes	yes	no	yes
Monitor mode	no ¹	no	yes	yes	yes	yes
ICE	no ¹	yes	yes	yes	yes	yes

¹ If the task or driver to be tested is interrupted by a breakpoint, real-time response is no longer given for the remaining system.

8.6.2 Test tools

8.6.2.1 RMOS3 debugger and resource reporter

The RMOS3 debugger and resource reporter represent utility tasks of RMOS3 and are used to debug user tasks. Both tasks need the operating system for support.

Debugger

Debugger functionality include, for example:

- Sequential control (e.g., startup) and checking the status of all tasks running in RMOS3
- Output of system calls
- Verification and modification of memory contents
- Setting breakpoints in user tasks or drivers (only in monitor mode).

Resource reporter

The resource reporter task is a useful supplement for the debugger. With the help of the reporter, it is possible to output the RMOS3 data structures and resources. The utility comprises functions for evaluation of tasks, device drivers, memory pools, flags, and similar. For more information on both test tools, refer to chapter 4 in the System Manual. The chapter describes, for example, the configuration and all test options.

8.6.2.2 Host debugger

Connection

The host debugger on a PC can communicate via LAN interface with the RMOS3 debugger as target system.

rm-gdb of the GNU

Features of the GNU rm-gdb.exe:

- Program that is executable in Windows
- Different modes of command input: text-based, function keys, buttons, function icons, pull-down menus
- Direct task loading via LAN over TCP/IP
- Symbol management in source code
- Source code debugging with GNU Compiler

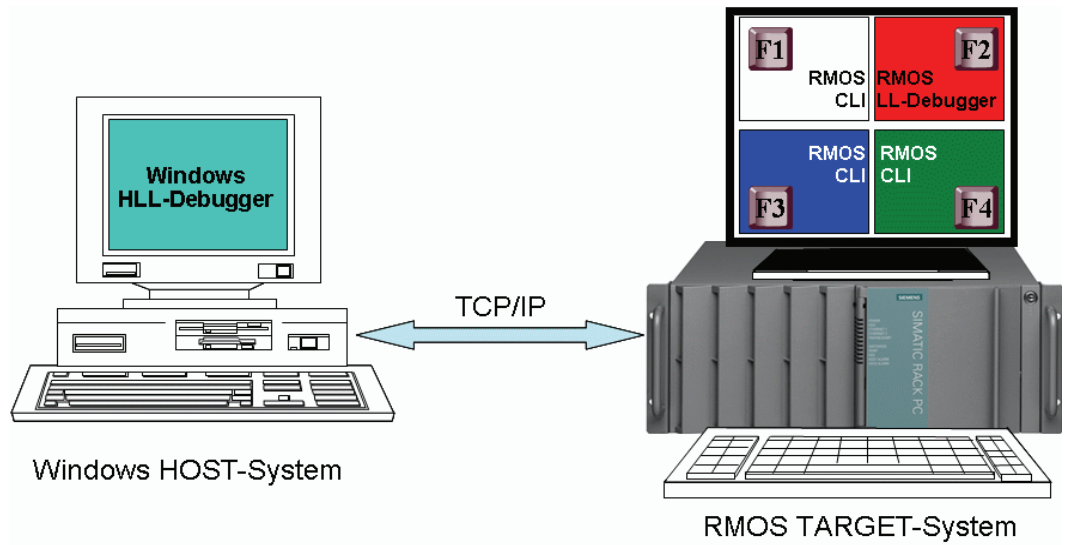


Figure 8-20 Debugger communication

8.6.2.3 Hardware emulator

A hardware emulator can be used for unrestricted testing of the entire RMOS3 application (including drivers). All details and test information related to the loading and testing of linked programs are available in the respective emulator description. Compared to the monitor, the emulator definitely represents a more convenient tool with higher performance (symbolic testing, trace buffer).

Note

Once the RMOS3 application was interrupted by a breakpoint, real-time processing is no longer possible because the entire system is frozen.

8.6.3 System startup messages

Boot messages

RMOS3 can output startup messages. For this purpose, an output routine that is referenced to a specific block is defined in configuration file RMCONF.C. This makes it possible for initialization routines to output strings using a `Putchar` function. The following messages are output at significant points of the startup routine:

Message	Meaning
NUC: init passpoint	Initializing the passpoint output.
NUC: init pic	Initializing the interrupt controller.
NUC: init OS	Creating system parameters.
NUC: config SVCs	Installing SVCs.
NUC: reserve interrupts	Allocating reserved interrupts.
NUC: init npx	Initializing the arithmetic co-processor.
NUC: create resources	Creating semaphores, event flags, and mailboxes.
NUC: create cli dispatcher	Creating CLI dispatcher.
NUC: init byt driver	Initializing BYT driver.
NUC: init ega	Initializing EGA units.
NUC: init com1	Initializing COM1 port (BYTE).
NUC: init com2	Initializing COM2 port (BYTE).
NUC: set system parameter	Defining the console for system output.
NUC: create reporter	Creating resource reporter.
NUC: config debugger	Installing debugger commands.
NUC: create error logger	Creating the error logger task.
NUC: init hsfS	Initializing the HSFS file management system.
NUC: create remote task	Creating the Remote task.
NUC: create init task system level	Creating initialization task at system level.
NUC: create init task user level	Creating initialization task at user level.
NUC: starting system	RMOS3 is running.
NUC: init fd0 driver	Initializing floppy driver.
NUC: drive A: (1.44 MB)	Drive A: 3.5" 1.44 MB.
NUC: drive B: not installed	Drive B: 5.25" 1.2 MB.
NUC: init ramdisk driver	Initializing RAM disk driver.
NUC: drive R: 2048 KB	Drive R: (xx kByte).
NUC: init hd0 driver	Initializing HDD driver.
NUC: harddisk 0	Drive C.

Meaning of the messages

The actual order of output depends on the configuration and hardware. For example, a corresponding DEVICE message is output prior to each driver initialization. This is followed by the respective UNIT messages for this driver. Not all of the strings listed above are output. Completion of the routine up to the message `NUC: starting system` indicates that the configuration does not contain any serious errors. Following a system crash during startup, you can derive the type of configuration error logger task of messages that were already output.

8.6.4 Error messages/error logger task

Two options

The operating system has two options of error message output:

1. The error logger task, which outputs error messages to the system console by means of a driver.
2. The `x_nucprintf` routine, which is available to the nucleus, for the direct output of messages to the system console.

Error logger task

The error logger task is a separate task used to output error messages to the system console. The error logger task is used, for example, by the CRUN. Drivers can also use the error logger task to generate error messages. The system configuration section (System Manual) describes the error logger task configuration and the definition of the system console; you can reproduce these in all RMOS3 applications included with your package.

Messages

If the system is lacking resources, the nucleus calls the `x_nucprintf` routine to output the following messages:

```
*** nuc: <date> <time> no SRBS, SYSTEM HALTED
```

No further system request blocks (SRBs) are available for the operating system.

```
*** nuc: <date> <time> no SMRS, SYSTEM HALTED
```

No further system memory blocks (SMBs) are available for the operating system (e.g. driver requests an SMR).

```
*** nuc: <date> <time> SMRS increased
```

The nucleus has added 50 system memory blocks (SMRs).

```
*** nuc: <date> <time> SMRS reached 0
```

Increase of the number of system memory blocks (SMRs) failed; the SVC was delayed. This status only develops if no memory space is available in the heap or the number of SMRs was limited by the call of `RmSetSMRCount`. Only the tasks requesting the SMRs are disabled, e.g., by means of SVCs. Other tasks, even those of lower priority, continue operations. Disabled tasks are resumed as soon as SMRs are available again.

8.6.5 Exception interrupt handler

Message content

The Exception Interrupt Handler (EIH as of herewith) is always called when the processor triggers an exception interrupt. It outputs the name of the triggered exception interrupt, the content of processor registers, as well as additional information such as the system status at the time of triggering of the exception interrupt to the screen. The modular EIH is split into an Assembler and a C section. This is a convenient setup for future extensions.

Note

The EIH impairs the real-time capability of RMOS3 as it employs the polling method for data output to the screen. The `x_nucprintf` routine will **not** explicitly disable interrupts.

On-screen output is handled by means of the `x_nucprintf` output routine of the RMOS3 nucleus. The EIH visualizes all numerical output data in hexadecimal format, except the date and time. The text that is output depends on the specific exception interrupt and on the specific system state (I, S or A) at which the interrupt was triggered. Four different scenarios may develop in this case:

Trigger of the processor exception

Scenario	Exception triggered by	System state
1)	Task	A
2)	SVC	S
3)	Interrupt handling routine	I
4)	Interrupt handling routine	S

The EIH outputs data with the following syntax, with "x" representing any character (no blank character) and "y" representing any string. The `error code` line is only output if the corresponding exception interrupt returns a CPU error code.

Task in A state

If an exception interrupt was triggered by a task in state A, for example, the data is visualized on-screen as follows:

```
*** nuc-<CoreID>: <date> <time> <exception text>: xxxx:xxxxxxxx
xxxx:yyyyyyyy ZZ...command
error code: y
caused by task <name> id: 0xXX tcb at address: xxxx:xxxxxxxx
eax: xxxxxxxx ebx: xxxxxxxx ecx: xxxxxxxx edx: xxxxxxxx
esi: xxxxxxxx edi: xxxxxxxx ebp: xxxxxxxx esp: xxxxxxxx
ss: xxxx ds: xxxx es: xxxx fs: xxxx gs: xxxx
cr0: xxxxxxxx, cr2: xxxxxxxx, cr3: xxxxxxxx
eflag: xxxxxxxx <(decoded flags)>
```

Interrupt routine in I state

If the exception interrupt was triggered by an interrupt handling routine in I state, the third line changes and is output as follows:

```
caused by interrupt handler in I state, SYSTEM HALTED
```

Interrupt routine in S state

If the exception interrupt was triggered by an interrupt handling routine in S state, the third line is output as follows:

```
caused by interrupt handler in S state, SYSTEM HALTED
```

The exception interrupt handler will stop the system in both of these situations.

<exception text> depends on the exception interrupt and is representative for the following strings:

INT-NUM	<exception text>
INT 0:	DIVIDE ERROR AT ADDRESS:
INT 1:	DEBUG EXCEPTION NEAR ADDRESS:
INT 3:	BREAKPOINT EXCEPTION NEAR ADDRESS:
INT 4:	OVERFLOW EXCEPTION NEAR ADDRESS:
INT 5:	BOUNDS CHECK NEAR ADDRESS:
INT 6:	INVALID OPCODE AT ADDRESS:
INT 7:	NO COPROCESSOR AVAILABLE AT ADDRESS:
INT 8:	DOUBLE FAULT EXCEPTION AT ADDRESS:
INT 9:	NPX SEGMENT OVERRUN NEAR ADDRESS:
INT 10:	INVALID TSS AT ADDRESS:
INT 11:	SEGMENT NOT PRESENT AT ADDRESS:
INT 12:	STACK FAULT AT ADDRESS:
INT 13:	GENERAL PROTECTION AT ADDRESS:
INT 14:	PAGE FAULT AT ADDRESS:
INT 16:	FLOATING-POINT ERROR NEAR ADDRESS:
INT 17:	ALIGNMENT CHECK NEAR ADDRESS:

You may use the `rm-gdb` RMOS3 GNU software package to analyze the exceptions (see the RMOS3 GNU manual).

The EIH outputs AT ADDRESS or NEAR ADDRESS, depending on whether or not the EIP register contains the address of the triggering or next command following the exception interrupt.

Example of on-screen output:

```
*** nuc-0: 02-JAN-2003 10:39:44, GENERAL PROTECTION AT ADDRESS: 0270:0000027A
0270:0000027A 64C60000 MOV BYTE PTR FS:[EAX],00
error code: 0
caused by task id: 0x21: `exep prot'
eax: FFFFFFFF, ebx: 00000000, ecx: 00000280, edx: 00000068
esi: AA55AA55, edi: 000002B8, ebp: FFFFFFF78, esp: FFFFFFF64
ss: 0278, ds: 0280, es: 0280, fs: 0000, gs: 0228
cr0: 7FFFFFFE3, cr2: 00000000, cr3: 0000C000
eflag: 00010282 ( SIGN INTERRUPT IOPL(0) RESUME )***
```

The example demonstrates on-screen output of the EIH on a GENERAL PROTECTION exception interrupt that was triggered by the `exep prot` task with ID 21.

NMI

The NMI-INTERRUPT (INT 2) is only processed by the assembler section of the EIH. The following strings are output to the screen:

```
*** nuc-0: <date> <time> NMI INTERRUPT
```

The exception interrupt handler is stored as MISCIN.ASM file in the directory EXAMPLES\ETC. For more information on using the EIH, refer to chapter 4.4 "Exception Interrupt Handler" in the System Manual.

8.6.6 SVC exception handler

Error returns from SVCs

The SVC exception handler is a subprogram that is called in S state. The source code for these routines is stored in the SVCEXC.C file in the directory SOURCE\ETC\CADUL\ . These routines support users who do not always consequently evaluate the error data returned by the SVCs. Each SVC with return status unequal 0 (0=RmOK) is output with a corresponding error message to the system console. These messages are independent of the programmed code of the user tasks and help you to locate irregularities in a system. These include, for example, return codes that may have an impact on program execution and that were not yet evaluated. The error message are output as follows

```
*** nuc-0: <date> <time>, svc <name> <status text>
failed: <error number> (<error text>)
```


Meaning of the representatives:

<code><name></code>	Decoded SVC name, e.g., <code>RmGetFlag</code>
<code><status text></code>	The following text is inserted, depending on the active system state at the call of the SVC exception handler. from task: <code><name></code> id: <code>0xXX</code> during system startup in monitor mode in s-state in i-state
<code>failed</code>	Decoded error text (cf. <code>RmDecode</code>). The error message is output as follows if the SVC was called via the old interface: <code>0xYY <error number> (<decoded error flags>)</code> .

Example

```
*** nuc-1: 14-FEB-1995 16:20:57, svc RmGetEntry from task: RUN id: 0x29
failed: 36 (invalid ID)
```

You may customize the SVC exception handler (refer to chapter 2.4.2 "Configuration file RMCONF.C" in the System Manual) that is also available as object in the RM3BAS.LIB library. The library version outputs all SVCs with status unequal zero and which are not listed in the exception table to the system console.

Suppressing unwanted messages

An SVC returning a status unequal zero is not necessarily faulty. You can also prevent unwanted SVC exception messages by suppressing messages to the error logger task. To this effect enter the relevant unwanted message, the precise status, and the SVC number in the corresponding table that is available in the SVCEXC.C file. The assignments of SVC numbers to SVC names are listed in the NUCIF.ASM file in the directory SOURCE\HLI\CADUL. Compile the file and link it in front of the RM3BAS.LIB file. SVC messages can be suppressed at runtime by calling `RmSetOS`.

8.6.7 Debugging with the RMOS3 debugger

The RMOS3 low-level debugger is a fixed component in a standard RMOS3 system (you may remove it from the RMOS3 system for special applications, for example, to reduce the size of the operating system nucleus). This program is used to debug task and to read system information. The RMOS3 debugger can be used to obtain an overview of system resources and of the operating system state, and explicitly change system states.

You can launch an RMOS3 debugger session on a selected console by entering the <CTRL>+<D> shortcut key. The utility opens with the following message:

```
RMOS3 DYNAMIC DEBUGGER, Vx.y  
>
```

For information on additional properties and operation of the RMOS3 debugger refer to chapter 4.2 "Debugger" in the System Manual or to chapter 3 "Reference for debugger commands" in Reference Manual Part I.

Chapter "Testing with RMOS3 Debugger (Page 64)" in this manual describes the practical use of the debugger.

8.6.8 Testing with the RMOS3 profiler

The RMOS3 profiler is a utility that provides support for the following activities

- Displaying system parameters
- Calculating load distribution
- Analyzing task activities.

You can use the results of the load distribution calculation to optimize system runtime. It is helpful for error handling and runtime optimization to log the task activities. It can also be used to measure SVC execution times and specific program units.

The following steps offer a short introduction to the practical use of the RMOS3 profiler. A comprehensive documentation of the RMOS3 profiler is available in chapter 5, "Reference for RMOS3 profiler commands", Reference Manual Part I.

8.6.8.1 Practical exercise with the RMOS3 profiler

You should start the following two small user programs that enable you to perform simple analyses on the RMOS3 system:

- LAST.386
- F_TASK.386.

LAST.386 is a load program that generates a controlled system load by means of two call parameters and then enters the DORMANT state for the duration of a specific pause. System load is generated by continuously reading the time. This program is available in BIN\LAST.

Expanding the F_TASK example

F_TASK.C is stored in the directory \EXAMPLES\F_TASK. This example is to be expanded by adding a WHILE loop.

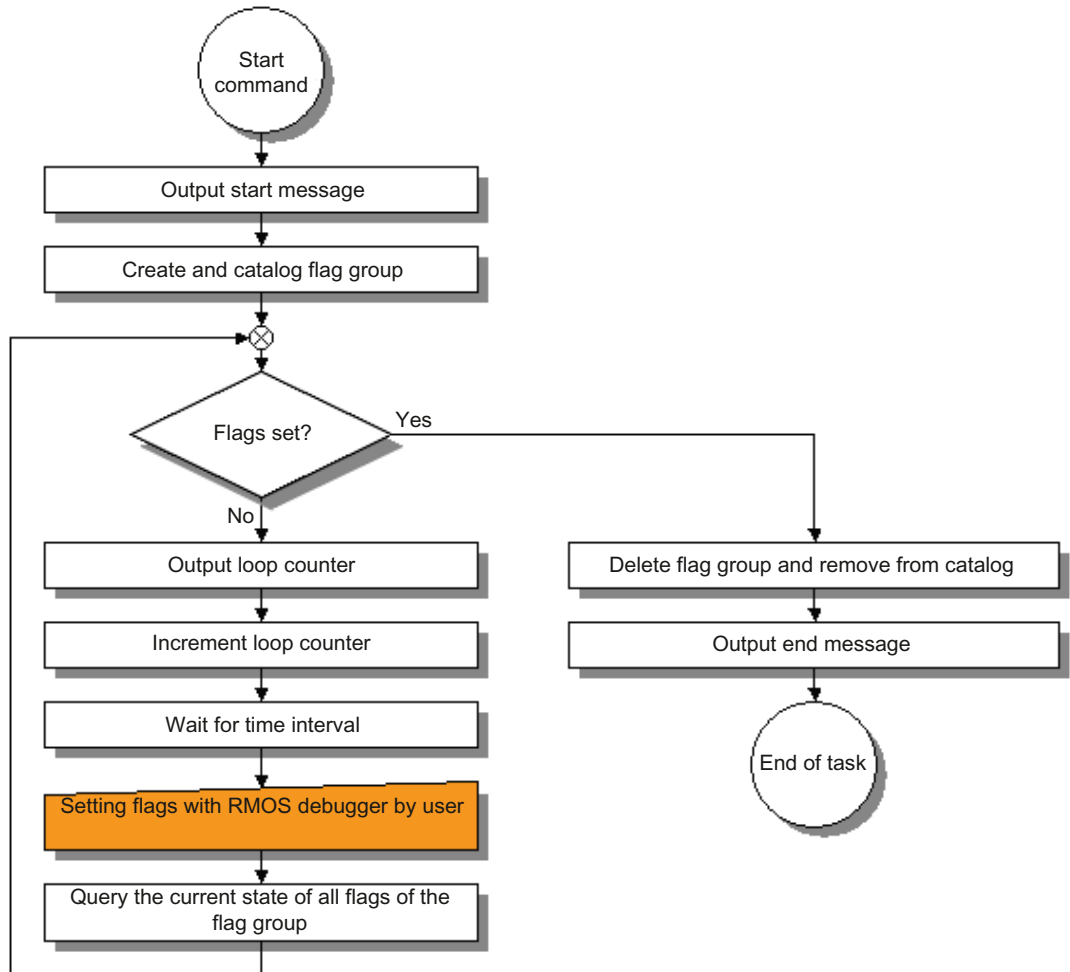


Figure 8-21 F_TASK extended with a WHILE loop

The following listing shows the expansions in bold. On completion of these changes, compile the program and transfer it to the target system.

Listing source code of the expanded F_TASK program

```
#include <rmapi.h>
#include <stdio.h>
#include <stdlib.h>
#define FLAG1 0x1

void svc_sts (unsigned int, char *, int);

void _FAR main (void)
{
    int Status;
    unsigned int FlagId, CurrentFlag, Counter;

    printf ("FIRST_TASK is running\n\n\r");
    Status = RmCreateFlagGrp("FIRST_FLAG", &FlagId);
    svc_sts (1, "RmCreateFlagGrp", Status);

    printf ("FIRST_TASK stops with flag %XH on Flaggroup %XH\n\n\r", FLAG1, FlagId);
    CurrentFlag = 0;
    Counter = 0;

    while (CurrentFlag != FLAG1)
    {
        /* output of current loop counter */
        printf ("Loop counter = %d\n\r",Counter);

        /* increment loop counter */
        Counter ++;

        /* wait a moment */
        Status = RmPauseTask(RM_SECOND(1));
        svc_sts (2, "RmPauseTask", Status);

        Status = RmGetFlag(RM_CONTINUE, RM_TEST_ONE, FlagId, FLAG1, &CurrentFlag);
    }

    printf ("FIRST_TASK will terminate\n\n\r");
    Status = RmDeleteFlagGrp(FlagId);
    svc_sts (3, "RmDeleteFlagGrp", Status);

    exit(0);
}

void svc_sts (unsigned int index, char *svc_txt, int status)
{
    char buf[RM_MAXDECODELEN];
    if (status >0)
    {
        printf("\nERROR : %s INDEX: %I CODE: %I", svc_txt, index, status);
        if (RmDecode(RM_SVCERROR, status, buf) == RM_OK)
            printf (" = %s\n", buf);
        else
            printf ("\n");
    }
}
```

Preparations on the target system

Start RMOS3 on the target system. If not automated, start the CLI on the white screen by entering <CTRL-R>.

```
CLI logon task
Enter name of startup file : <ENTER>
RMOS3 Command Line Interpreter V1.4
C:\>
```

First, run F_TASK as CLI background job with

```
C:\>start f_task
CLI Job 3 started
```

Launch the LAST load program. Press <F3> to change to the blue console and start a second CLI with <CTRL-R>

```
CLI logon task
Enter name of startup file : <ENTER>
RMOS3 Command Line Interpreter V1.4
C:\>
```

and start LAST

```
C:\>last 1000 10000
Creating an RMOS load: WAIT=1000, LOOP=10000
```

```
Start LOOP
Ende LOOP
```

Overview of resources

Press <F2> to return to the red screen, start the RMOS3 debugger and get an overview of the cataloged tasks with

```
>dir task
Cataloged resources
Symbolic-name  kind  id          Symbolic-name  kind  id
BU_COUNT      TASK  0010H      BU_COUNT-1    TASK  0011H
CLI_CLEAN_UP  TASK  000EH      CLI_DPAT      TASK  0000H
CLI_JOB_0     TASK  0059H      CLI_JOB_1     TASK  004EH
CLI_JOB_2     TASK  0054H      CLI_JOB_3     TASK  0056H
CLI_JOB_5     TASK  005BH      CLI_JOB_6     TASK  005DH
CLI_JOB_7     TASK  005CH      CLI_PL0_SERV1 TASK  0006H
CLI_PL0_SERV2 TASK  0007H      CLI_PL0_SERV3 TASK  0008H
CLI_PL0_SERV4 TASK  0009H      CLI_TASK_PL0  TASK  0005H
...
```

Of course, you also want to know which tasks the various CLI jobs have to complete. For this purpose, press <F1> to return to the white window. The "systat" command displays all active CLI jobs:

```
>systat
Tsk Pjob Job Pri Command          stdout          Start time
059h      0  64 CLI C:\CLISTART.BAT Device=0 Unit=0 12-JUN-2008 13:18:28
04Eh      1  63 C:\RM3RUN/ftpd.386 -n Device=0 Unit=4 12-JUN-2008 13:18:20
054h      2  60 C:\RM3RUN/telnetd.386 Device=0 Unit=4 12-JUN-2008 13:18:21
056h      3  60 C:\RM3RUN/telnetd.386 Device=0 Unit=4 12-JUN-2008 13:18:21
05Ah      4  64 CLI C:\CLISTART.BAT Device=0 Unit=4 12-JUN-2008 13:18:34
05Ah      4  64 systat          Device=0 Unit=4 12-JUN-2008 13:19:12
05Bh      5  64 CLI C:\CLISTART.BAT Device=0 Unit=2 12-JUN-2008 13:18:42
05Dh      5  64 last 1000 10000 Device=0 Unit=2 12-JUN-2008 13:19:07
05Ch      7  63 f_task          NUL             12-JUN-2008 13:19:0
```

Starting the RMOS3 profiler

You can now launch the RMOS3 profiler on the white console:

Main menu "Select measurement mode"

```
C:\>rprof

Rprof, RMOS Profiler V2.1.0 running on RMOS3 Version V03.50.00

Select measurement mode
0 - Determine load distribution
1 - Determine task activity
2 - Activate screen paging
3 - Deactivate screen paging
4 - Terminate program
5 - Show system parameters
6 - Control a memory
7 - Select fast timer tick
8 - check tasks (pri, npx)
Input:
```

Determining the general system parameters

To determine the general system parameters you now need to select the "5 - Show system parameters" item:

```
Input: 5
```

"System parameters" menu

```
System parameters
System clock:          1 ms (HPET)
RMOS3 Version:        V03.50.00
Measurement priority: 250
Interrupt latency (APIC): 5 us
Max. timer latency:   7 us
Min. timer latency:   1 us
Addresses which caused the interrupt latency (hex):
latency      address  core
  0      50:00006c33    0
  1      50:000035a8    1
  3      30:000236d0    0
  5      50:00003469    1
```

After you acknowledged the following two final messages

```
Reset interrupt latency measurement ? (Y/N)n
Set max. interrupt latency trigger ? (Y/N)n
```

Press "N" to return to the main menu.

Logging load distribution

The

"0 - Determine load distribution" command of the main menu returns an overview of computing load distribution.

Input: 0

"Determine load distribution" menu

```
Determine load distribution
0 - Start measurement
1 - Stop measurement
2 - Start measurement of specified duration
3 - Output measurement
4 - Return to main menu
5 - Release memory
6 - Start cyclic measurement
7 - Output cyclic measurement
Input:
```

Now you need to start logging for a specific period. You may select the period manually by executing the "0 - Start measurement" or "1 - Stop measurement" menu commands, or set a default period by executing the "2 - Start measurement of specified duration" menu command.

Select the first variant and start the measurement with

Input: 0

run a 2 second measurement and then stop it again with

Input: 1

The data is output by executing menu command 3

Input: 3

8.6 DEBUG strategies in RMOS3

```
Start time of measurement:      00:01:27:782 (h:min:s:ms)
Stop time of measurement:      00:01:30:746 (h:min:s:ms)
Duration of measurement:       2964 ms
Load distribution for tasks(core0):      3 ms ( 0.10%)
Load distribution for s-state(core0):    2 ms ( 0.06%)
Idle time(core0):              2959 ms ( 99.83%)
Load distribution for tasks(core1):      0 ms ( 0.00%)
Load distribution for s-state(core1):    2 ms ( 0.06%)
Idle time(core1):              2962 ms ( 99.93%)
Task computing load           ID           Task
0 ms ( 0.00%)                34          rhpollT0
0 ms ( 0.00%)                36          rhpollT1
0 ms ( 0.00%)                38          rhpollT2
0 ms ( 0.00%)                3a          rhpollT3
0 ms ( 0.00%)                3b          memDriverT
0 ms ( 0.00%)                3c          memPollT
0 ms ( 0.00%)                3d          HSF_U0
0 ms ( 0.00%)                3e          HSF_U1
0 ms ( 0.00%)                3f          HSF_U2
0 ms ( 0.00%)                40          HSF_U3
0 ms ( 0.00%)                45          TCP/IP-Timer
0 ms ( 0.00%)                46          keyrepeatT
0 ms ( 0.00%)                5c          CLI_JOB_7
1 ms ( 0.03%)                5d          CLI_JOB_6
0 ms ( 0.00%)                62          CLI_JOB_9
```

```
Determine load distribution
0 - Start measurement
1 - Stop measurement
2 - Start measurement of specified duration
3 - Output measurement
4 - Return to main menu
5 - Release memory
6 - Start cyclic measurement
7 - Output cyclic measurement
Input:
```

You can read the duration of measurement, load distribution for the cores of tasks and interrupt routines in S state, and the IDLE time from this evaluation. It also provides a tabular list of the active tasks and their slices of system load.

Select the

"4 - Return to main menu" menu command to return to the main menu:

Input: 4

Logging task activities

In a conclusive step you should list and evaluate the task activities. Select the corresponding "1 - Determine task activity" command from the main menu.

Input: 1

"Determine task activity" menu

```
Determine task activity
0 - Start measurement
1 - Stop measurement
2 - Start measurement of specified duration
3 - Output measurement
4 - Return to main menu
5 - Release memory
6 - Output measurement to file RPROF.SAV
Input:
```

This function, too, allows you to determine the measuring period in manual mode or by specifying a precise default period.

Select the "0 - Start measurement" menu command

```
Input: 0
```

to open a submenu in which you can select special tasks and status transitions for logging:

"SVC selection" menu

```
SVC selection
0 - All SVCs including special states (TASKIN, TASKOUT...)
1 - All SVCs excluding special states (TASKIN, TASKOUT...)
2 - Only specified SVCs
3 - Old SVC selection
4 - Abort input
Input:
```

Select the following command to log all tasks and status transitions

```
""0 - All SVCs including special states (TASKIN, TASKOUT...)""
```

```
Input: 0
```

and, in the next query "0ffff = all tasks" query, to record all tasks:

```
Selection of task ID (hex., max. 5, 0ffff = all tasks): 0ffff
```

The log is written to a linear buffer that is allocated from the heap; select the buffer length as follows:

```
Buffer size
0 - 1 kByte
1 - 4 kByte
2 - 8 kByte
3 - 16 kByte
4 - 32 kByte
5 - 64 kByte
6 - Special buffer size
7 - Abort input
Input:
```

After you executed menu command

```
"6 - Special buffer size"
```

and specified a buffer length, the measurement is started and written to the buffer of the specified size as soon as logging data is input:

```
Input: 6
```

Terminate the measurement after four seconds with

```
Input: 1
```

Log analysis

You now save the log as RPROF.SAV file for further processing to the directory from which RPROF was started or display the data on your screen:

```

Determine task activity
0 - Start measurement
1 - Stop measurement
2 - Start measurement of specified duration
3 - Output measurement
4 - Return to main menu
5 - Release memory
6 - Output measurement to file RPROF.SAV
Input:

```

Select menu command "3 - Output measurement"

Input: 3

to display the following list on your screen (here is printed an extract):

```

Duration of measurement:      4918ms
Buffer size (1024000 Bytes) was sufficient

```

Time [ms]	Counter [usec]	Action	TASK ID	DATA	CORE
1192	578	RmNucData	5c	1	0
1192	580	RmGetSysB	5c		0
1192	582	RmGetTaskID	5c		0
1192	584	RmGetTaskID	5c		0
1192	586	RmNucData	5c	1	0
1192	588	RmNucData	5c	2	0
1192	589	RmNucData	5c	1	0
1192	591	RmGetSysB	5c		0
1192	593	RmGetTaskID	5c		0
1192	595	RmGetTaskID	5c		0
1192	597	RmNucData	5c	100	0
1192	598	RmNucData	5c	101	0
1192	600	RmNucData	5c	1	0
1192	602	RmGetSysB	5c		0
1192	604	RmGetTaskID	5c		0
1192	605	RmGetTaskID	5c		0
1192	607	RmNucData	5c	1	0
1192	609	RmNucData	5c	2	0
1192	611	RmPauseTask	5c		0
1192	612	TASKOUT	5c		0
1192	614	TASKIN	10		0
1193	534	DI-STATE	0	d1	0
1193	535	DI-STATE	1	d1	1
1194	534	DI-STATE	0	d1	0
1194	536	DI-STATE	2	d1	1
1195	534	DI-STATE	0	d1	0

Log structure

The two columns on the left side list the time stamps in microseconds, while the third column (Action) indicates the SVC that is currently executed by the task that is listed in the same line. The fourth column (TASK ID) lists the task IDs of all logged tasks or the interrupt vector of the target interrupt routines. The fifth column (DATA) displays any additional information about the executed SVCs. For support on this topic, refer to chapter 5 "Reference for the RMOS3 profiler commands" in Reference Manual Part I. The sixth column (CORE) contains the CoreID of the processor core in which the task was executed. The last column lists the task names and the names of the interrupt routines, for example, of the timer tick (RMOS3 system clock rate) in DI state and also an interrupt processing in S state, namely execution of SVC `RmPauseTask`.

Log analysis

It is quite apparent that F_TASK is the most frequently executed program.

A status transition is initiated when F_TASK executes SVCs `RmPauseTask`. F_TASK terminates itself with TASKOUT and enters the DORMANT state. The only task in the Ready Task queue, namely BU_COUNT (Busy task) enters the RUNNING state at the transition TASKIN.

Closing the profiler

Select menu command

```
"4 - Return to main menu"
```

```
Input: 4
```

to return to the main menu. Close RPROF by selecting menu command

```
"4 - Terminate program"
```

```
Input: 4
```

8.6.9 Debugging at source code level

Debugging at source code level is possible using the GNU tools.

GNU debugger

For information on using the GNU tools for debugging, refer to the RMOS3 GNU Manual, chapter 5 "Debugging an RMOS3 application".

8.7 Basic I/O system of RMOS3

Controlling devices in RMOS3

In RMOS3, distributed I/O devices are controlled and managed by means of suitable device-dependent software. The entire device-dependent software is assigned to the I/O area. I/O software can be implemented by means of user tasks, relatively simple interrupt handlers, and operating system software (drivers, see below). I/O not supported by a driver can be implemented in a user task with corresponding interrupt handler. The user task is assigned a priority that suits I/O processing requirements.

The advantage of this solution is that it saves you from having to program custom drivers or from getting involved with the internal data structures of RMOS3. Instead, you can rely on SVCs and a relatively simple interrupt handler to meet an extensive range of requirements. For example, you can implement an interrupt routine that triggers the start of a task (`RmQueueStartTask`) that handles processing.

Drivers

The device-specific software components of the operating system are known as drivers. The terms "manager", "driver", and "device driver" relate to the same matter. Each driver controls and manages one or several distributed I/O devices of the same type (e.g., output transactions via one or several V.24 interfaces). In RMOS3, all distributed I/O devices that are controlled and managed by a single driver are known as "devices" or "units" of this driver. You can roughly split the drivers into two categories: Character-oriented and block-oriented drivers. You can see the following differences between the character-oriented and block-oriented drivers:

- A block-oriented driver reads and writes blocks that are assigned a fixed number of characters. Each block can be addressed independently of other blocks (floppy drives are block-oriented devices).
- Character-oriented drivers read a stream of characters without structure. These characters cannot be addressed individually. Terminals, printers, and mouse pointer devices, or similar represent character-oriented devices.

Serial and parallel drivers

A further feature that allows differentiation is to determine whether or not it is possible to control the devices of a driver separately. Floppy drives, for example, usually have a single controller. For this reason, I/O requests to one or several floppy drives must be controlled successively (serial model).

In contrast, it is possible to control and manage several terminals independently (in parallel), because each terminal is assigned its own V.24 controller.

Depending on whether the driver uses one controller to control several devices or several controllers to control one device per controller, the corresponding driver is interpreted as serial or parallel driver.

A driver usually consists of one or several programs running primarily as system processes, with interface to the nucleus that is realized by means of subprogram calls.

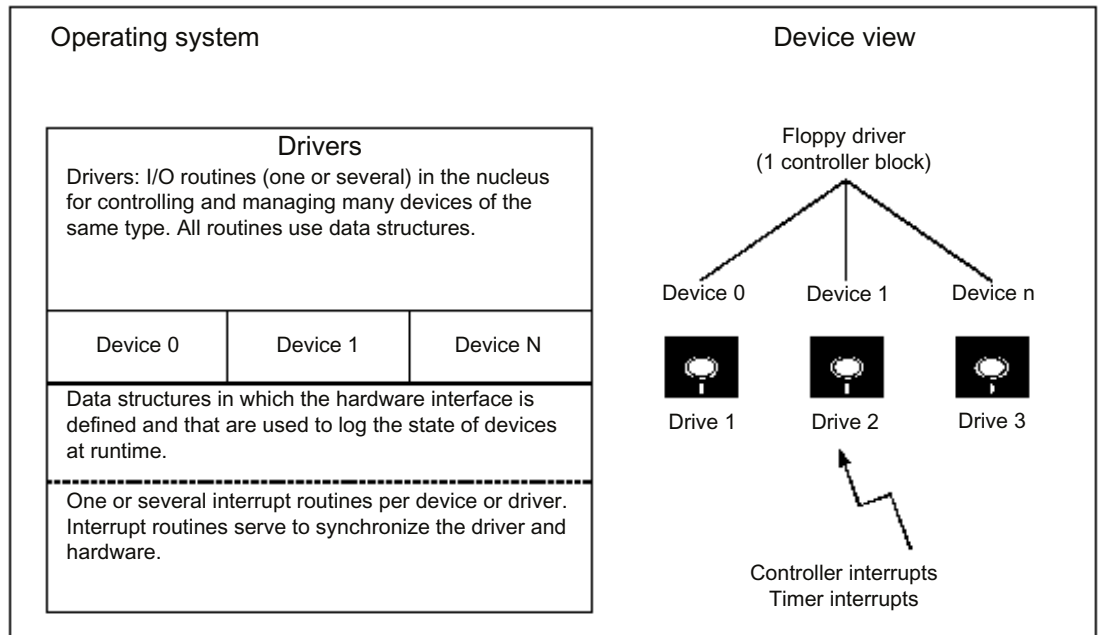


Figure 8-22 Structure of drivers and devices

Processing priority

System processes do not have a priority attribute; all internal computing requests of the operating system are handled based on the FIFO principle ("First In First Out"). Synchronization between drivers and hardware or between user tasks and hardware is handled using hardware interrupts and corresponding interrupt handlers that can branch to system processes (to the S state).

The drivers can be addressed from user tasks using a uniform SVC_{RmIO} provided by the nucleus. In the SVC, the driver type is always passed by means of its ID (device ID) and the corresponding device ID (unit ID) as parameters. Each SVC can be output to a driver with or without wait parameter. If the "with waiting" parameter is passed in SVC_{RmIO}, the nucleus sets the task to the BLOCKED state until the driver has completed processing of the request.

Default drivers

The most important default drivers are:

BYT driver:

This driver supports central control of all character-oriented devices of an application (e.g., terminals or printers). The driver supports the duplex mode and XON/XOFF, buffers characters and provides support for emergency entries, interpretation of terminal control characters.

Floppy driver:

Driver for controlling physical access to the floppy drive for recording data on 3.5" and 5.25" floppy disks.

HD driver:

Driver for operating devices with EIDE interface

DMA driver:

Driver for controlling DMA controllers (e.g., 82258, 82380). This driver serves primarily for the support of floppy and Winchester drivers and may also be addressed by tasks.

8.8 Arithmetic co-processor

For several tasks

An arithmetic co-processor (e.g., 80387), if available, can be used by tasks for complex mathematical operations. If a task that has accessed the co-processor is blocked, RMOS3 swaps the registers of the co-processor. The content of the registers is backed up by means of request of corresponding dedicated memory space and is referred to as NPX context (NPX = "Numeric Processor eXtension").

A task using the arithmetic processor is identified automatically.

8.9 C Runtime library CRUN

8.9.1 Conditions for using the C library

Size of the area for code and constants

Memory requirements for code and constants, without floating point functions, amount to approximately 110 KB. The optimized library for the 80387 co-processor and compatibles occupies approximately 6 KB.

Size of the data area

The data area has a size of approximately 4 KB. This area contains the static structures for file management and the Runtime environment.

Stack size for each task

Stack requirements of the C library functions amount to approximately 4 KB.

Requests to memory management

For each task requesting C Runtime support, the following memory requirements must be fulfilled:

- Approximately 2 KB at the call of the `xinitc` function. This memory is also requested implicitly if a task does not call `xinitc` but uses C functions nevertheless (see `xinitc`).
- Approximately 1 KB for each opened stream, if the size of the stream buffer for this stream was not adjusted by means of the `setvbuf` or `setbuf` functions.

Segment names

The C library (without floating point functions) is stored in subsystem CRUN; while the floating point library is stored in subsystem CRUN_NUM.

Initializing C Runtime support

To initialize C Runtime support, you first need to call the `xinitc` function once. In order to initialize the task-specific data, you also need to call the `xinitc` function at the start of every task. These steps must be completed to make the actual CRUN functions available.

`xinitc` is called by default by the initialization task (in RMCONF.C). The RMOS3 CLI initializes all tasks it starts (`main`) with `xinitc`.

Note

If the calls `xinitc` and `xinitc` are missing, initialization is completed automatically (see `xinitc` and `xinitc`).

8.9.2 Mathematical functions

The numerical functions declared in header file MATH.H perform floating point operations and are defined for double arguments (64-bit).

The remaining mathematical functions can only be applied to integer values. These are declared in header file STDLIB.H.

Note

The calculation result returned by the functions in header file MATH.H is always an approximation having a precision that is limited by the numerical co-processor used.

General error analysis

Errors are handled in the numerical part of the CRUN library in two phases.

1. An error in a numerical function first triggers the call of the user-specific `matherr` function that is to be programmed.
2. The function generates a return value that indicates whether or not the error was rectified. If yes, the numerical function returns the corrected value to the user program.

If `matherr` has not corrected the error, further procedures depend on whether the associated exception is enabled or disabled in the co-processor.

If enabled (e.g., OVERFLOW is enabled), the corresponding bit in the co-processor status register is set to call the signal handler for SIGFPE.

Otherwise, a default value is returned (e.g., `+HUGE_VAL` for `exp(5000)`). Variable `errno` is set to a corresponding error value as well.

Error handling functionality by means of `matherr` is not included in the ANSI standard and is derived from the UNIX SYSTEM V. However, the procedure for setting variable `errno` and returning a special value conforms to ANSI standard.

The numerical functions support the handling of two types of error:

A DOMAIN ERROR is triggered if the argument exceeds the definition range of the function. In this case, variable `errno` is set to the EDOM value and an implementation dependent value is returned. For the CRUN library, this is the IND value that is specified in the ISO/IEC DIS 9899 standard. This value is a negative NaN (Not a Number). This NaN is a special number format of the 8087 processor. A NaN is output, for example, by the `printf` function as "+NaN" or "-NaN" string.

A RANGE ERROR is triggered if the result exceeds the range that can be visualized. On range overflow, the function returns the `HUGE_VAL` value, and on range underflow a 0 value. `errno` is set to ERANGE in both situations. Whether or not the co-processor triggers an overflow or underflow exception by means of the signal handler depends on the enable status of these exceptions in the co-processor. `errno` is not set for trigonometric `sin`, `cos` and `tan` functions.

Handling special 8087 number formats

All special number formats are handled as arguments of numerical functions as follows:

NaNs (Not a Number)

The function returns an NaN that is set as argument in a numerical function without changes. This procedure conforms to the ISO/IEC DIS 9899 standard. An NaN, for example, is the return value of the square root function, which is derived from the square root of a negative number. With NaN as argument, the function neither calls `matherr`, nor sets `errno`.

INFs (infinities)

INFs (infinities) are special infinite numbers in ISO/IEC DIS 9899 format for representation of $+\infty$ and $-\infty$. +INF is derived, for example, from a division of 1 by 0 with disabled NULL-Divide-Exception in the co-processor. This value can then be passed as an argument to a numerical library function. INFs are handled in accordance with the definition ranges. Example: `atan(+INF) = $\pi/2$` .

Denormals

Denormals represent very small figures that can no longer be represented with full precision on the co-processor (due to leading zeros in the mantissa). The CRUN library interprets these numbers as zero value.

Unnormals

Unnormals are numbers that are derived from further calculations with a denormal number. They only appear on 8087 or 80287 architectures. These numbers can be represented in the mantissa without leading zeros, but the numerical co-processor continues calculations with leading zeros. Unnormals are also generated whenever the co-processor is loaded with a denormal.

The following procedure is selected in the CRUN library:

An argument representing an unnormal $< 2^{-63}$ is treated as zero. An unnormal $> 2^{-63}$ is normalized and `matherr` is called with DOMAIN. Large unnormals indicate that the user program contains errors.

Note

Denormals and unnormals $< 2^{-63}$ are rather infrequent and handled in the CRUN library as zeros. Unnormals $> 2^{-63}$ are extremely rare and should not appear in normal situations. NaNs and INFs indicate that previous calculations returned incorrect results.

Initializing the 80x87 co-processor

The function for initialization of the 80x87 co-processor is designed by default for PC-compatible hardware. The SWCCF87.ASM file contains the functions listed below, which possibly need to be customized to suit the special features of the hardware and user software. The separate SWCCF87.ASM init file that was realized for handling hardware that is not PC-compatible must be linked ahead of the RM3BAS.LIB library.

```
far void x_cr_initcopr()
```

This function initializes the 80x87 co-processor. The function is called in the initialization routine by means of `xinitt`, if this is a task with numeric flag set in the TCD. This function can be used to enable selected exceptions.

Co-processor settings

During initialization, the following basic settings are activated at the co-processor:

- Rounding mode is **Round to Nearest**
- Precision Control is **80 bits** (CRUN calculates internally at maximum precision)
- The high-affinity range of numbers is selected (in compliance with ISO/IEC DIS 9899)
- All exceptions are disabled.

The following section describes the various exceptions and their handling:

Precision exception

Always triggered on the loss of calculation precision. This scenario is always given, for example, when computing sine and cosine functions. This exception must always be disabled within the CRUN.

Underflow exception

Upon underflow in an arithmetical operation, the value with reduced precision (denormal or 0) is used for further computing.

Enabling this exception renders you responsible for the proper handling of the underflow exception. CRUN does not support this procedure.

Overflow exception

An overflow (e.g., at `exp(3000)`) triggers the corresponding co-processor interrupt for the overflow exception, provided this interrupt is enabled.

If the overflow exception is disabled, the default value defined for the various CRUN functions is returned. The co-processor always returns the corresponding MASKED RESPONSE (INF) if the overflow exception is disabled. The CRUN functions return the HUGE_VAL value that is specified in the ANSI standard.

Zero Divide Exception

If Zero Divide Exception is disabled, the value defined by the MASKED RESPONSE of the co-processor is returned.

Denormalized operand

A denormal that is nonetheless derived as interim result will be used for further computing. The resultant loss of precision is tolerated.

By enabling the denormal exception, you are responsible for ensuring that the value is normalized in the exception routine. A normalization is not executed in the CRUN.

Invalid operation

This exception is triggered by invalid arithmetical operations (e.g., calculation of the square root of a negative number). If this exception is triggered, the corresponding exception handler is called.

If you disable this exception, the respective CRUN function returns a special value. Information on this value is available in the description of the respective functions and of the 8087 architecture (for operations that are not completed within CRUN).

The following initialization sequence should be used in `x_cr_initcopr` (the example initialization is programmed in Assembler):

```
controlword dw      0001001100111111B
             FINIT
             FLDCW   controlword
             FWAIT
```

This initiates execution of the aforementioned standard initialization. Once exceptions are enabled, a task-specific signal handler is called if an exception condition is met. The respective task is terminated by default. The signal handler can be exited with a Return command, for example, when an underflow exception is being processed. It is also possible to exit the signal handler with `longjmp` if an error condition is triggered. In this case, you need to install a custom signal handler.

```
far struct env87 void x_cr_issue_eoi_copr(void)
```

This function is used to process 80x87 interrupts. The function is called in RMOS3 DI state. The function writes the 80x87 environment to the stack (14 words).

```
far int x_cr_init_copr_hw()
```

This function returns the number of the interrupt vector that is used to trigger co-processor exceptions (PC/AT: external interrupt 13, 80x86 architecture: Interrupt 16). The interrupt vector of the PC depends on the programming of the 8259 interrupt controller. Interrupt 13 is the interrupt vector used in DOS. This function is also used for hardware initialization of the co-processor. This includes, for example, programming of the 80486 to set the co-processor to a PC-compatible mode (NE bit in control register CR0). `x_cr_init_copr_hw` is called only once by `xinitc`.

The CRUN routines for PC-compatible co-processor connections are available by default. These are used if `SWCCF87.OBJ` is not linked.

8.9.3 Special features of the C Runtime library

Comments related to the HSFS file system

Write or read/write access to files is controlled by means of lock option. This means that only one task can access a file at any given time.

It is not permitted to pass a file pointer (`FILE *`) to a different task for further use. The `fopen` or `fduopen` functions return file pointers. The same rule is valid for file descriptors (return value from `open`).

Error handling

If an error number (`errno`) unequal zero was returned as a result of a function call, you can possibly analyze the error in detail. This affects all functions that execute I/O operations by means of the RMOS3 file system or RMOS3 BYT driver. In addition, an `errno2` variable will be initialized in this case. This variable contains the original status values of the called driver or of the HSFS. Variable `errno2` only contains useful values if `errno` is unequal zero.

Example

```
#include <stdio.h>
#include <errno.h>

errno = 0;
errno2 = 0;
fclose(fp) ;
if (errno != 0)
{
    printf("errno: %d, errno2: %d\n", errno, errno2);
}
```

`errno2` is not included in the ANSI specification and is an RMOS3 extension instead. This error variable serves to support troubleshooting. You should not use `errno2` when writing portable programs.

Data output by means of BYT driver

Each string is passed unchanged to the BYT driver. The `'\n'`, `'\t'`, or `'\r'` arguments are not handled separately. You must customize the configuration of the BYT driver to suit the terminal type. Binary characters are passed forward without changes. The BYT driver is possibly capable of interpreting these characters. Only for streams opened in text mode, the entries are converted from `'\n'` to `'\r\n'` for write operations and from `'\r\n'` to `'\n'` for read operations

If the BYT driver is used to read strings, the respective last character received is validated. The `'\r'` character is converted to `'\n'` in this case.

Note

The respective device of the driver is not reserved for input or output operations. This means that all I/O jobs will be processed in chronological order. You must take this into account when using the C library.

8.9.4 Example programs for using the C library

You need to complete the following steps for commissioning:

1. Global call of `xinitc`
2. Call of `xinitt` in the initialization task.
3. Starting other tasks.
4. Calling `xinitt` at the start of each further task.
5. Exiting tasks with `exit`.

Example program

```
#include <stdio.h>
#include <task.h>
#include <io.h>
#include <errno.h>
/* ... other includes */

/* This task has to be started first */
void INITIALIZATION_TASK (void)
{
    /* Use memory pool -1 (Heap), generate temporary files on drive C: in directory TEMP
    */
    xinitc (-1, 0, 0, 2, "C:\\\"TEMP");

    /* xinitt: stdin, stdout and stderr on device 0, unit 0 */
    xinitt (0,0, 0,0, 0,0, NULL);

    /* ... Do the rest */

    /* go away */
    exit (0);
}

void TASK1 (void)
{
    xinitt (0,0, 0,0, 0,0, NULL);
    /* ... Do the rest */
    exit (0);
}

/* ... */

void TASKx (void)
{
    xinitt (0,1, 0,1, 0,1, NULL);
    /* ... Do the rest */
    exit (0);
}
```


RMOS3 system configuration

9.1 Overview

RMOS3 is supplied by default with the configurable RMOS3 nucleus (PC_CNUC) . The nucleus is parameterized using configuration file RMOS.INI. It is not necessary to generate the system.

For information on the RMOS3 system, refer to the System Manual, chapter 2 "Configuring the RMOS3 nucleus".

9.2 Boot sequence and memory allocation

Basic boot sequence

The RMOS3 boot sequence conforms to DOS conventions, which means that after a system reset the BIOS in EPROM loads the RMOS3 boot sector (512 bytes) from the corresponding boot medium (HDD, Compact Flash Card) to system memory as of address 0:0x7C00.

The code of the RMOS3 boot sector, namely the RMOS3 boot loader, loads and starts the RMOS3 system that is available as memory image. The RMOS3 system must be stored at the first position in the main directory of the boot medium.

Because the boot sector code is executed almost exclusively in real mode, the RMOS3 systems can only be loaded to the first MB gap without special boot loader. This restricts the maximum size of the RMOS3 nucleus to 608 KB.

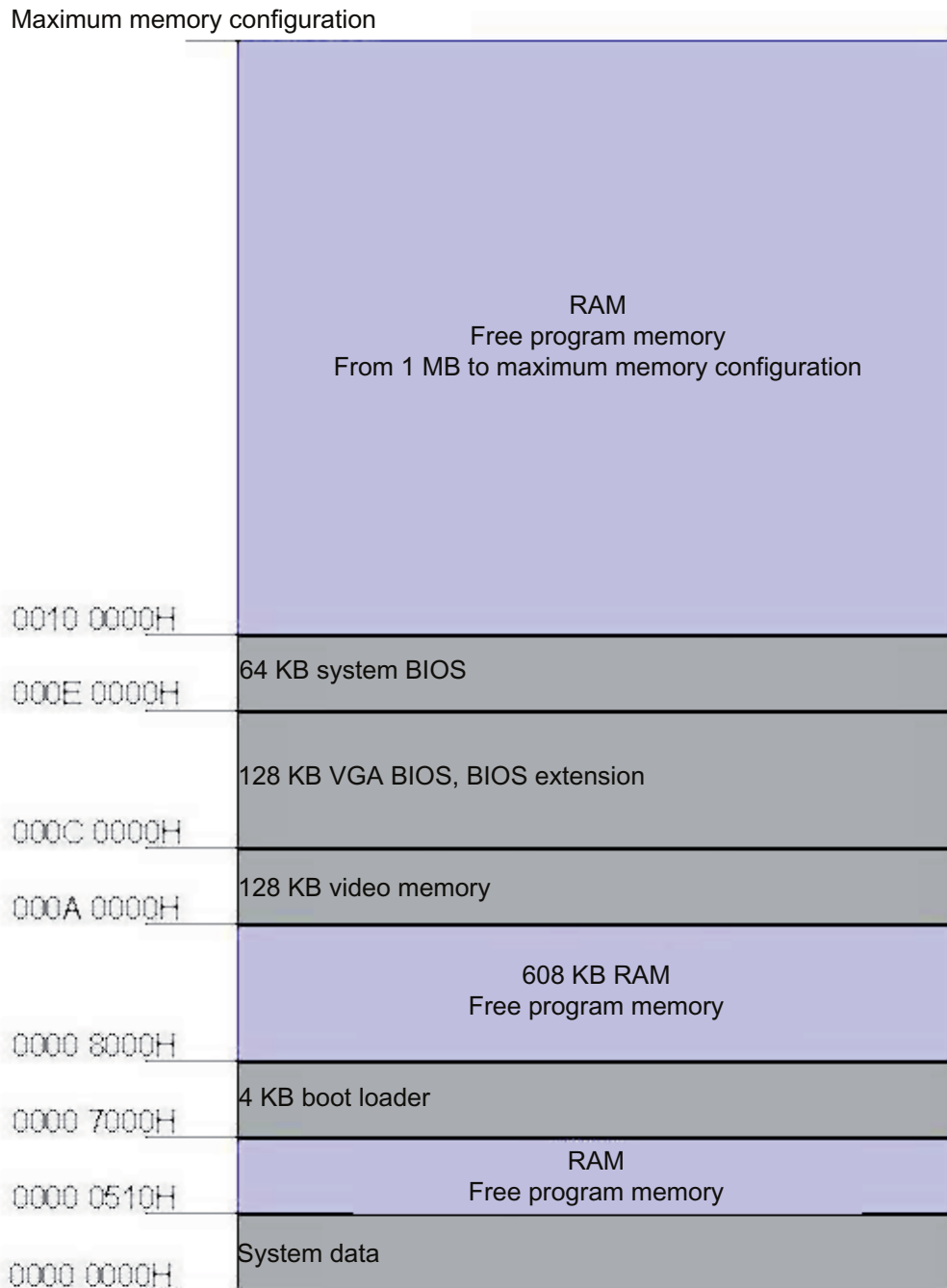


Figure 9-1 Memory mapping in a typical PC architecture

Before the boot sector code starts an RMOS3 system by means of task switch, the GDT, IDT and task registers (with GDT entry 5) are loaded and switched to protected mode. The selection of the boot medium depends on the BIOS.

An RMOS3 boot sector is set up on HDDs, Compact Flash Cards, floppy disks and RAM/ROM-Disks (floppy image) using the RDISK.EXE utility.

Operating system variants and their boot options

Depending on the size of the RMOS3 nucleus RM3_PC1.SYS, you can load the operating system directly with the RMOS3 boot loader (only possible with file sizes less than 608 KB), or using the RMLDR second stage boot loader.

You may also boot MS-DOS and then run the DOS boot loader LOADX.EXE to load the RMOS3 nucleus.

Based on the memory mapping of the typical PC architecture, various areas are reserved for the BIOS, graphic and system data as shown in the figure above. The range from 000A 0000H to 000F FFFF is usually referred to as adapter gap.

RMOS3 boot loader

You can load the RMOS3 nucleus RM3_PC1.SYS to the range from 0000 8000H to 0009 FFFFH to optimize memory utilization. However, this is based on the condition that the nucleus code does not exceed a length of 608 K, which is not the case in terms of the configurable nucleus.

Second stage boot loader RMLDR

You will have to use the RMLDR second stage boot loader if the size of the nucleus exceeds this limit. Instead of the RMOS3 nucleus, you first load the RMLDR that boots the code of the operating system nucleus RM3_PC1.SYS. The code is loaded directly to the addresses above 0010 0000H, which means that the code of the nucleus is now located above the adapter gap and is no longer restricted in terms of its length.

DOS boot loader LOADX

You can also rely on the DOS boot loader to load the operating system code above the adapter gap. First, you need to boot MS-DOS, but without loading the memory manager. In MS-DOS, the DOS boot loader LOADX.EXE then loads the RMOS3 nucleus RM3_PC1.LOC. In this variant it is not necessary that the memory image of the nucleus is relocated.

Memory distribution

In all three variants, the contiguous memory image RM3_PC1.SYS or RM3_PC1.LOC of the nucleus is loaded to CPU RAM within the boot sequence and then processed. RAM is split accordingly into a memory area for the nucleus code (code segment, or local ROM), as well as the static variables, data structures (data segment, local RAM), and heap (RAM) that the nucleus needs. The following diagram highlights the corresponding memory images.

Maximum memory configuration

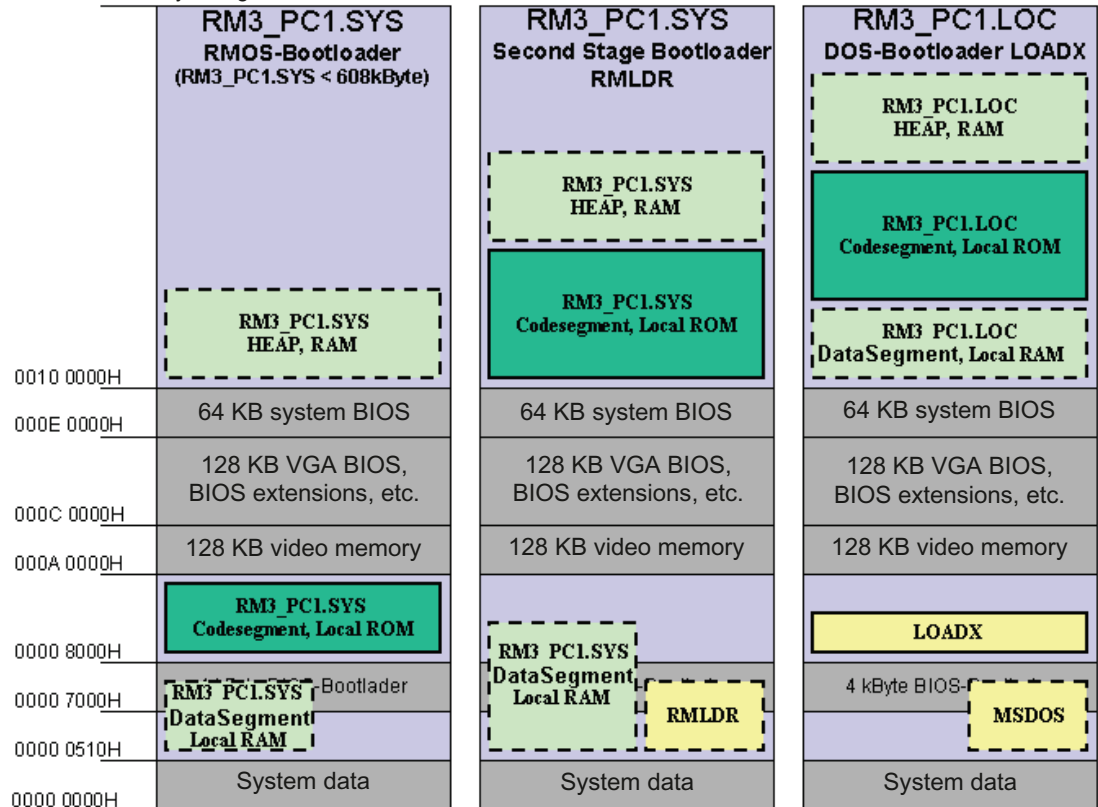


Figure 9-2 Basic structure of RMOS3 on PC

9.3 System properties of the RMOS3 nucleus

Properties of the configurable nucleus

Once the RMOS3 system has completed its startup, the operating system elements listed in the following sections will be available in the configurable nucleus.

9.3.1 Allocation of the device units

File management system

RMOS3 contains the HSFS file management system. HSFS is capable of managing DOS 12-bit FAT (File Allocation Table), DOS 16-Bit FAT, BIGDOS, VFAT16, FAT32, and VFAT32 partitions.

A: Floppy drive A:	The parameters are now read from CMOS.
C: Hard disk drive(s)	The partitions on the first HD are integrated automatically at system startup. The system detects DOS12, DOS16, BIGDOS, VFAT16, FAT32, or VFAT32 partitions. Beginning with C, the drive letters are assigned consecutively: C: Partition 1 D: Partition 2
R:	RAM disk, set by default to 2 MB
R0:	SRAM disk, battery-backed RAM disk. Optional, integrated in PC_CNUC

EGA/VGA configuration

For visual differentiation, the following colors were assigned in the pre-configured software:

RMOS3 system console	Unit 0: white
RMOS3 console 1	Unit 1: red
RMOS3 console 2	Unit 2: blue
RMOS3 console 3	Unit 3: green

The RMOS3 system console is activated at system startup. The F1, F2, F3, or F4 function keys are used to select the units.

A debugger can be activated at the unit by pressing <CTRL>+<D>, and the command line interpreter CLI by pressing <CTRL>+<R>.

Interrupt allocation

The following interrupt routines are installed: The IRQs may differ to these in APIC mode, depending on the chipset.

Interrupt	Vector	Allocation
IRQ0	60H	Timer block
IRQ1	61H	Keyboard input
IRQ2	62H	Cascading
IRQ3	63H	Serial interface 2 (COM2)
IRQ4	64H	Serial interface 1 (COM1)
IRQ5	65H	Free (Unexpected Interrupt Handler)
IRQ6	66H	Floppy controller
IRQ7	67H	Printer (Centronics)
IRQ8	70H	Free (Unexpected Interrupt Handler)
IRQ9	71H	Free (Unexpected Interrupt Handler)
IRQ10	72H	Free (Unexpected Interrupt Handler)
IRQ11	73H	Free (Unexpected Interrupt Handler)
IRQ12	74H	Free (Unexpected Interrupt Handler)
IRQ13	75H	Co-processor exception interrupt
IRQ14	76H	HD controller
IRQ15	77H	Free (Unexpected Interrupt Handler)

DMA allocation

Channel	Allocation
0 and 1	Floppy control

Operating system clock rate

The operating system clock rate is set to 1 ms, which means that 1 ms is the shortest time interval for the `RmPauseTask` and `RmRestartTask` calls.

9.3.2 Software configuration

CRUN

The configured C Runtime support provides all functions. It conforms to ANSI Standard Draft International Standard ISO/IEC DIS 9899 (published 1990). The functions are listed individually in Reference Manual Part III.

System tasks

The following RMOS3 system tasks are configured:

Catalog entry	Name	Meaning
CLI_DPAT	CLI distribution task	You start a debugger by entering <CTRL>+<D> on this console. If the console was not yet assigned a debugger, a new debugger named DEB_1, DEB_2, etc., is generated. Enter <CTRL>+<R> to start the CLI logon task that will run the CLI.
REP	Resource reporter task	Output to unit 0. Call from the debugger with START 2 or using the REP command.
DEB_0	Debugger	Activation with <CTRL>+<D>.
ERRLOG	Error logger task	
REMOTE	REMOTE task	Task for loading via serial interface
HSF_A HSF_B HSF_R	HSFS tasks for floppy drives and the RAM disk	HSFS starts a task for each data volume.
RMCONF	RMOS3 startup task	

The order of the following tasks depends on the started RMOS3 system:

Catalog entry	Name	Meaning
BU_COUNT	Busy task	Measures system load
CLI_CLEANUP	CLI task for background jobs	Clears memory on completion of the background job
CLI_JOB_0	Basic task of the CLI	CLI prompt
HSF_C	Task for HD partition C:	A HSFS task is started for each HD partition.

RMOS3 drivers

The following RMOS3 drivers are configured:

Driver	Device ID	UnitID	Meaning
BYT drivers:	0	0	System console (EGA/VGA unit 0), keyboard (input)
		1	EGA/VGA Unit 1
		2	EGA/VGA Unit 2
		3	EGA/VGA Unit 3
		4	Serial interface (COM1, 19200 bit/s)
		5	Serial interface (COM2, 19200 bit/s)
		6	Printer (LPT1: Centronics)
FD0 driver	1	0 1	Floppy drives A: and B:
RAM disk driver	2	0	Virtual drive R: In the RAM
HD0 driver	3	0	HD volume C: and D:
		1	

9.3.3 HD integration

HDINIT

The hard disks are integrated into the HSFS at system start. Module HDINIT is included in the RMOS3 linking process and executed during initialization.

The HD0 driver is initialized in the first phase. For this purpose, the data of the BIOS ROM table is used which the interrupt vectors 41H and 46H point to. On the completion of initialization of the driver for Unit 0 (drive C) and Unit 1 (drive D), the file management system is initialized in the second phase. A "file volume name" is assigned to each DOS partition in the process.

Integration is completed by calling the `x_hd_init` function in the initialization task. For more information on the SVC, refer to Reference Manual Part II.

9.3.4 Warm restart

<Ctrl>+<Alt>+

You may trigger a warm restart of RMOS33 by pressing <Ctrl>+<Alt>+.

A

Abbreviations/Glossary

API

Application Programming Interface

APIC

Advanced Programmable Interrupt Controller

BSP

Board Support Package.

CAD-UL

Computer Aided Design Ulm, compiler manufacturer

CLI

Command Line Interpreter, user interface to the operating system.

Client

Consumer or client of a utility, sometimes the term denotes a role behavior in the relationship between two co-operating processes in which the client takes over the active, requesting role.

Configuration Space

All modules on the PCI bus must provide a block of 64 dwords for their configuration data. The first 16 dwords are defined by the PCI specification.

CRUN

ANSI C runtime library for RMOS3. Provides all C functions in accordance with ANSI Draft standard ISO/IEC DIS 9899 (published 1990) in the RMOS3 multitasking environment.

DCB

Driver Control Block; table that lists the current dynamic data of the driver.

DCD

Driver Control Data; table that contains the default configuration values.

Device

Driver program. RMOS3 handles I/O operations by means of special programs, or in short, drivers and their units. The drivers to be made available to the operating system are specified during system configuration. The operating system identifies drivers based on their number, namely the device ID.

DMA

Direct Memory Addressing

Driver

Program module in the operating system, which operates or controls an I/O block

EIDE

Enhanced Integrated Drive Electronics

EOI

End Of Interrupt

EWB

Event Flag Wait Block, management block for waiting for an event flag

FTP

File Transfer Protocol

GDT

Global Descriptor Table

HLL debugger

High Level Language debugger

HSFS

High Speed File System

IDT	Interrupt Descriptor Table
IRB	I/O Request Block transferred to a driver.
Job	Programs and commands started from the CLI (RMOS3 command line interpreter).
LAN	Local Area Network
Message	A message is interpreted in RMOS3 as content of a 3-word buffer
MMB	Mailed Message Block; management block for waiting to fetch a message.
MPIC	Master Programmable Interrupt Controller
NMI	Non-Maskable Interrupt
NPX	Numeric Processor Extension; numerical co-processor
PCI	Peripheral Component Interconnect; data bus system of 32-bit width
PIC	Programmable Interrupt Controller
PIT	Programmable Interrupt Timer

Placeholder character

Also known as wildcard or joker character. Special characters as ellipsis:

"*" represents a group of letters or numbers

"?" "?" represents a single alphanumerical character

PWB

Pool Wait Block; management block for waiting to allocate memory

RCB

Restart Control Block; management block for start requests of tasks

RMB

Receive Message Block; management block for waiting to receive messages.

RMOS

Real-time Multitasking Operating System

Root directory

In HSFS, files are stored in directories. In HSFS, directories form a hierarchic structure. The root directory represents the top level directory of a volume. Only one root directory is available per volume (partition).

RTQ

Ready Task Queue, internal data structure of the nucleus for managing all tasks that are in READY state.

SDB

Supervisor Descriptor Block; used for the transfer of parameters at system calls.

Server

Service-providing partner in the relationship between two cooperating processes

SMR

System Memory Resource; system memory block set up by the nucleus for internal management. Generic term for TMB, SWB, RMB, RCB, PWB, MMB, IRB, EWB

SPIC

Slave Programmable Interrupt Controller

SRAM

Shadow Random Access Memory

SRB

System Request Block; supervisor request block, structure for storing status transitions

SSPIC

Soft Slave Programmable Interrupt Controller

Subdirectory

Denotes all directories on a volume that are not root. Subdirectories must be assigned a name that is unique in the directory in which they were created.

SVC

Supervisor Call; system call

SWB

Semaphore Wait Block; data structure that the nucleus sets up for waiting for semaphore.

TCB

Task Control Block; table that lists the current dynamic data for controlling a task.

TCD

Task Control Data, table of default values; created for static tasks in the configuration, for dynamic tasks by the loader.

TCP

Transmission Control Protocol

TCP/IP

TCP Internet Protocol

Telnet

Telecommunications Network, dialog service via TCP/IP networks

TIB

Timer Control Block

TMB

Time Monitor Block; set up by the nucleus for managing time-related calls

UCB

Unit Control Block; table that lists the current dynamic data for controlling a unit

UCD

Unit Control Data; table listing the default configuration values

Unit

I/O device. One or several units are addressed by drivers. The units that a driver may address are specified in the driver configuration data. The operating system identifies units based on their number, namely the unit ID.

Index

3

32-bit protected mode, 26

A

A state, 135

ANSI-C Runtime Library, 32

APIC, 93

Application state, 103

B

BLOCKED, 80

Block-oriented drivers, 148

Board Support Packages, 47

Boot sector, 159

Boot sequence, 159

Booting RMOS3 using the DOS boot loader, 49

Booting RMOS3 using the RMOS3 boot loader, 49

Busy task, 165

BYT driver, 166

C

C Runtime Library, 33

Cascading, 94

Character-oriented drivers, 148

CLI, 165

CLI task, 165

Command line interpreter, 59

Communication, 69

Configurable RMOS3 nucleus, 159

Configuring tasks, 62

Coordination, 69

Co-processor, 150

D

Data area, 151

Data types, 13

Debugger, 165

Descriptor, 124

Descriptor table, 124

Development environment, 43

Development system, 40

Device control, 148

DI state, 25

Disabled Interrupt state, 98

DMA driver, 150

DORMANT, 80

Driver, 166

Drivers, 148

E

Edge-triggered mode, 95

Embedded systems, 36

Error logger task, 165

Event flags, 111

Exception interrupt handler, 134

Expected interrupt, 92

F

FAR pointer, 124

FD0 driver, 166

File Allocation Table, 163

File management system, 163

File manager, 34

File Transfer Protocol, 27

Flag, 111

Floppy driver, 150

FTP, 34

H

HD driver, 150

HD0 driver, 166

HEAP, 122

Host debugger, 130

HSFS, 163

HSFS tasks, 165

Human Machine Interface (HMI), 16

I

I state, 135

IND, 152

Infinite, 153
Input and Output, 29
Installation, 43
Interprocess communication, 69
Interrupt, 92
Interrupt control, 23
Interrupt controller, 93
Interrupt descriptor table, 23
Interrupt handler, 97
Interrupt routines, 164
Interrupt Service Routine, 95
Interrupt sharing, 34
Interrupt state, 99
Interrupt Vector Table, 95

K

Keyboard drivers, 58
Keyboard layouts, 58

L

Level-triggered mode, 95
LOADX.EXE,
Logical names of resources, 29
Low-level debugger, 137

M

Mailbox, 113
Memory management, 122
Memory protection, 68
Memory requirements, 151
Messages, 114
Multiprocessing, 75
Multitasking, 68
Multitasking mechanism, 21

N

NMI, 136
NON-EXISTENT, 81
Nucleus, 69

O

Operating system calls, 67
Operating system clock rate, 164
Operating system structure, 32

P

PCI bus, 95
PCI interface, 34
PCI scanner, 104
Performance features, 37
PIC, 93
Pool, 128
Preemptive multitasking, 37
Priority, 88
Process control, 17
Protected mode, 124
Protective mechanisms, 26

R

RAM disk, 166
RDISK.EXE, 160
Reaction time, 34
READY, 80
Ready Task Queue, 82
Real-time, 15
Real-time behavior, 68
Real-time clock, 118
reentrant, 80
REMOTE task, 165
Resource catalog, 117
Resource management, 29
Resource reporter, 165
Resources, 117
rm-gdb, 130
RMLDR,
RMOS3 profiler, 142
Round-Robin, 84
Round-Robin counter, 118
RUNNING, 80

S

S state, 135
Scheduler, 101
Scope of performance, 34
Segment selectors, 124
Semaphores, 108
Shared Interrupt Client, 104
Shared Interrupt Server, 104
Shared interrupts, 95
SMR, 133
Software clock, 118
Spinlock, 110
SRB, 133
Stack, 62

State change, 22
Status transitions, 22
SVC, 79
SVC calls, 24
SVC exception handler, 136
Synchronization, 69
System calls, 126
System console, 163
System Control Block, 126
System startup, 132
System state, 101
System tasks, 165
System utilities, 28

T

Target system, 48
Task, 77
Task communication, 106
Task ID, 77
Task management, 28
Task organization, 21
Task processing, 119
Task stack, 62
Task states, 21
TCB, 85
TCD, 85
TCP/IP, 34
Telnet, 34
Telnet Daemon, 19
Time management, 29

U

Unexpected input, 87
Unexpected interrupt, 92
Utility tasks, 70

X

x_cr_init_copr_hw, 155
x_cr_initcopr(), 154
x_cr_issue_eoi_copr, 155

