SIEMENS

SIMATIC

System Software for M7-300/400 Writing Loadable Drivers

Manual

Part of M7-SYS RT V4.0

Edition 1, April 1998

Safety Guidelines

This manual contains notices which you should observe to ensure your own personal safety, as well as to protect the product and connected equipment. These notices are highlighted in the manual by a warning triangle and are marked as follows according to the level of danger:

Danger

indicates that death, severe personal injury or substantial property damage **will** result if proper precautions are not taken.

Warning

indicates that death, severe personal injury or substantial property damage **can** result if proper precautions are not taken.

Caution

indicates that minor personal injury or property damage can result if proper precautions are not taken.

Note

draws your attention to particularly important information on the product, handling the product, or to a particular part of the documentation.

Qualified Personnel

Only **qualified personnel** should be allowed to install and work on this equipment. Qualified persons are defined as persons who are authorized to commission, to ground, and to tag circuits, equipment, and sys–tems in accordance with established safety practices and standards.

Correct Usage

Note the following:

Warning

This device and its components may only be used for the applications described in the catalog or the technical description, and only in connection with devices or components from other manufacturers which have been approved or recommended by Siemens.

This product can only function correctly and safetly if it is transported, stored, set up and installed cor-rectly, and operated and maintained as recommended.

Trademarks

SIMATIC®, SIMATIC HMI® and SIMATIC NET® are registered trademarks of SIEMENS AG.

Some of the other designations used in these documents are also registered trademarks; the owner's rights may be violated if they are used by third parties for their own purposes.

Disclaimer of Liability

We have checked the contents of this manual for agreement with the hard-ware and software described. Since deviations cannot be precluded entirely, we cannot guarantee full agreement. However, the data in this manual are reviewed regularly and any necessary corrections included in subsequent editions. Suggestions for improvement are welcomed.

Siemens AG
Automation and Drives Group
Industrial Automation Systems
P.O.Box 4848, D- 90327 Nuremberg

Preface

Purpose of the Manual

This manual supports you in writing your own loadable drivers for the M7-300/400 automation system. It describes the structure and operating principle of loadable drivers under M7 RMOS32 and the methods used to program them.

The manual is intended both as a user manual and as a reference document for the necessary function calls and data structures.

Audience

This manual is aimed at system programmers who develop loadable drivers for hardware components of the M7-300/400 automation system.

You will need to be familiar with the M7-SYS RT system software, the STEP 7 standard software, and the programming language C. You should also be familiar with the hardware for which you intend to program a driver.

Validity of the Manual

This manual is valid for the M7-SYS RT system software, version V4.0.

Documentation Required

The following manuals contain additional information about the M7-SYS RT system software and the development of application programs for SIMATIC M7-300/400 automation systems, and are available from your regional Siemens office.

Documentation Package System Software for M7-300/400 Order No. 6ES7802-0FA14-8BA0		
Manual	**General Contents**	**Contents Relating to Loadable Drivers**
Installation and Operation, User Manual	Installation and operation of M7–300/400 automation computers	Loading of drivers
Program Design, Programming Manual	Design and development of C programs.	Communication with loadable drivers from the user program
System and Standard Functions, Reference Manual	Detailed information on programming with M7–SYS RT	Function calls and data structures for communication with loadable drivers

Manual and Online Help

This manual is only available in electronic format as part of the M7-SYS RT system software V4.0. The reference section of the manual, which describes the function calls and data structures, is also available in the online help file M7RLDRVB.HLP in the S7BIN directory of STEP 7. You can include this file in the search range of the OpenHelp function of the Borland IDE for context–sensitive support during programming. The procedure used to include the file is the same as for the other help files of M7-SYS RT and M7 ProC/C++.

Feedback

We need your help to enable us to provide you and future M7-SYS RT users with optimum documentation. If you have any questions or comments on this *manual* or the *online help*, please fill in the remarks form at the end of the manual and return it to the address shown on the form. We would be grateful if you could also take the time to answer the questions giving your personal opinion of the manual.

Literature References /.../

References to other manuals are shown using the part number of the literature between slashes /.../. Using these numbers you can find out the exact title of the manual from the literature list at the end of this manual.

SIMATIC Training Center

Siemens also offers a number of training courses to introduce you to the SIMATIC S7 and M7 automation systems. Please contact your regional training center or the central training center in Nuremberg, Germany for details:

D-90327 Nuremberg, Tel. (+49) (911) 895 3154.

SIMATIC Customer Support Hotline

Contactable worldwide round the clock:

Nuremberg

SIMATIC BASIC Hotline

Local time: Mo.-Fr. 8:00 to 18:00

Phone: +49 (911) 895-7000

Fax: +49 (911) 895-7002

E-Mail: simatic.support@nbgm.siemens.de

SIMATIC Premium Hotline

(Calls billed, only with SIMATIC Card)

Time: Mo.-Fr. 0:00 to 24:00

Phone: +49 (911) 895-7777

Fax: +49 (911) 895-7001

Johnson City

SIMATIC BASIC Hotline

Local time: Mo.-Fr. 8:00 to 17:00

Phone: +1 423 461-2522

Fax: +1 423 461-2231

E-Mail: simatic.hotline@sea.siemens.com

Singapore

SIMATIC BASIC Hotline

Local time: Mo.-Fr. 8:30 to 17:30

Phone: +65 740-7000

Fax: +65 740-7001

E-Mail: simatic@singnet.com.sg

SIMATIC Customer Support Online Services

The SIMATIC Customer Support team provides you with comprehensive additional information on SIMATIC products via its online services:

- You can obtain general current information:
  - On the **Internet** at `http://www.ad.siemens.de/simatic`
  - Using **fax polling** no. 08765-93 02 77 95 00
- Current Product Information leaflets and downloads which you may find useful for your product are available:
  - On the **Internet** at `http://www.ad.siemens.de/support/html–00/`
  - Via the **Bulletin Board System** (BBS) in Nuremberg (*SIMATIC Customer Support Mailbox)* at the number +49 (911) 895-7100.

    To access the mailbox, use a modem with up to V.34 (28.8 kbps), whose parameters you should set as follows: 8, N, 1, ANSI, or dial in using ISDN (x.75, 64 kbps).

Further Support

If you have any further questions about SIMATIC products, please contact your Siemens partner at your local Siemens representative's or regional office. You will find the addresses in our catalogs and in Compuserve (`go autforum`).

Contents

Programming Loadable Drivers

1

This chapter describes the structure and operating principle of the loadable drivers for M7-300/400 and the methods used to program them.

1.1 General Information About Drivers

Introduction

The operating system provides a range of different device drivers (also referred to simply as drivers) for controlling hardware components such as I/O devices or mass storage units. A driver is responsible for activating and deactivating a device; for passing the correct hardware parameters to the device; for ensuring that data can be exchanged between the device and the user program; and for handling device errors.

A second type of driver exists in addition to the drivers which are permanently installed in the operating system (such as the hard disk driver). These “reloadable” drivers can be loaded into the work memory on demand. Examples of this type of driver include the **3964** and **ser8250** drivers used to access the serial interface.

With Version 4.0 (or higher) of the M7-SYS RT system software you can write your own loadable drivers. In addition to the **RmIOxxx** call interface for the use of existing loadable drivers, further calls are now also available for initializing drivers, processing user requests, and handling interrupts and timeouts.

Unlike normal user programs, device drivers are part of the operating system. For this reason, it is important that they behave “correctly” at all times. Since the kernel places very few restrictions on their actions, the drivers you write must never adversely affect the stability of the overall system.

Character Devices and Block Devices

We distinguish between two types of device according to the format of the data processed:

- Character Device

  The data transmitted from the device are unformatted. Data which are read must be interpreted by the device driver or by the user program itself.

- Block Device

  The data transmitted from the device have a defined format and can be addressed in blocks.

It is also possible to use hybrid character and block devices, however the driver must provide two separate access modes for hybrid devices.

Devices and Units

The terms **device** and **unit** appear repeatedly in the paragraphs below.

The device is the driver for a hardware component (for example SER8250 for serial interfaces) or for a virtual device (for example a virtual console).

A unit is a device unit (management unit) of the driver and is responsible for one interface of the driver. In a driver for serial interfaces, for example, one unit is responsible for the I/O operations of one serial interface (for example COM1). Another unit is responsible for a further serial interface (for example COM2).

Both the driver (the device) and the unit are resources. They are entered in the resource catalog whenever the driver is loaded or a unit is generated.

Maximum Number of Drivers and Units

The resource management system in M7 RMOS32 supports the use of up to 256 drivers. The identifiers are allocated as follows:

- 0 to 63 for permanently loaded drivers
- 64 to 255 for reloadable drivers

Up to 256 units can be generated per driver, hardware permitting.

Example Programs

You will find example programs for loadable drivers on the programming device/PC in STEP 7 in the directories

- \m7sys4.00\EXAMPLES\DRV8250 – sample driver DRV8250 for the serial interface
- \m7sys4.00\EXAMPLES\DMYDRV – dummy driver DMYDRV

What You Need to Know About the Hardware

If you are writing a loadable driver for a hardware component, you will need to be familiar with the hardware specification. In particular, you will need the following information:

- Interrupts
- I/O addresses
- Memory allocation for memory-mapped I/O or dual–port RAM.

You will find this information in the hardware documentation.

Features of System Tasks

The tasks of a device driver run as “system tasks”. This applies to initialization and device tasks as well as interrupt and timeout handlers. System tasks can only be generated by other system tasks and the tasks they generate themselves are also system tasks.

System tasks differ from user tasks in respect of the following features:

- **Memory Protection**

  Unlike user tasks, which run at user level, system tasks have system–level permissions, which means that they are not governed by memory protection features (see also ***/280/*** Chapter 4.).

- **Priorities**

  System tasks can run with higher priority than user tasks. In addition to priorities 0 to 255, which can also be set for user tasks, a system task can also be assigned the higher priority RM_HIGHPRI(511). However, this is only possible with the RMOS-API calls ***RmCreateDeviceUnit(..)*** and ***RmCreateDevice(..)***. Values other than 0 to 255 or RM_HIGHPRI(511) are illegal.

- **Scheduler Disables**

  System tasks are not governed by the scheduler disables issued by user tasks. If ***RmDisableScheduler(..)*** is called by a user task, the schedule disable only acts on user tasks. System tasks continue to be managed by the scheduler.

  If ***RmDisableScheduler(..)*** is called by another system task, however, only this task remains active, while all other tasks (user and system tasks) are excluded from scheduling.

  In all other respects, the scheduling behavior is the same as for user tasks, that is, the task with the highest priority which is ready to run is active. Where two or more tasks have the same priority, the processor time is allocated equally among the tasks in time slices.

Note

RM_HIGHPRI(511) is the highest possible priority for system tasks. A device task which runs with this priority takes precedence over all other system tasks, including the system server. If you program a task with RM_HIGHPRI(511) you must make sure that it does not impair the function of the scheduler or the system server.

Plan the priorities of the individual tasks carefully and exercise caution when using scheduler disables, in order to prevent a malfunction in your multitasking program from being caused by an incorrect priority allocation. (Please see ***/280/*** Section 4.5. for more information)

1.2 Structure of a Loadable Driver

Important Components of a Loadable Driver

Loadable drivers have the same basic structure as normal loadable M7 RMOS32 applications and consist of the following main components:

- An initialization task.

  This initializes the driver once it has been loaded and started.

- One or more units

  A unit consists of the following components:

  – Device task with message queue for processing the I/O requests

  – (If necessary) one or more interrupt handlers

  – (If necessary) one or more timeout handlers

We distinguish between a parallel and serial driver structure, according to the way in which the I/O requests are processed.

Parallel Structure

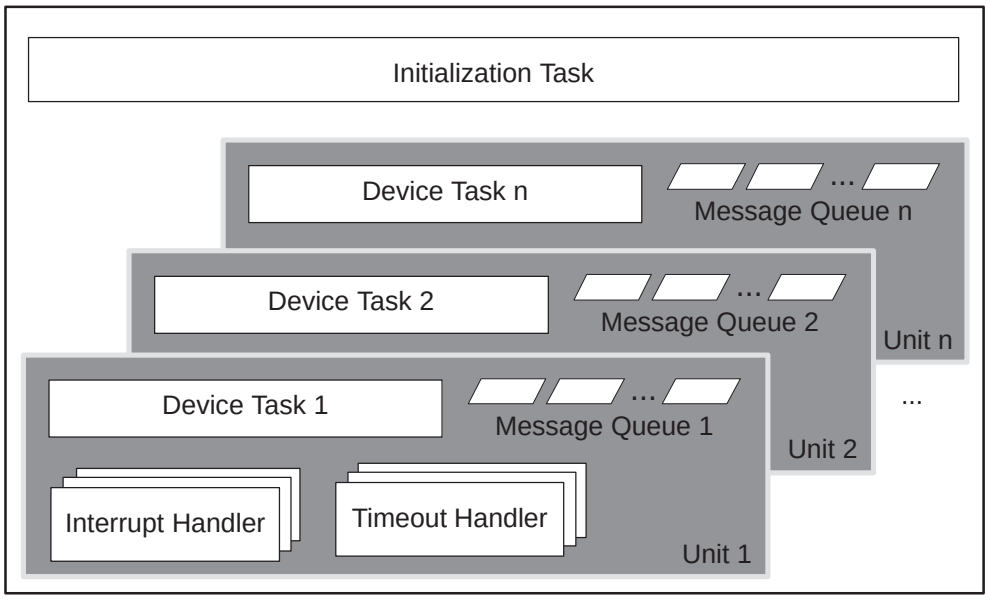

Figure 1-1 Driver with Parallel Structure

A device task is generally assigned to a unit, that is, the units can be operated in parallel. Figure 1-1 shows the structure of a parallel driver.

Serial Structure

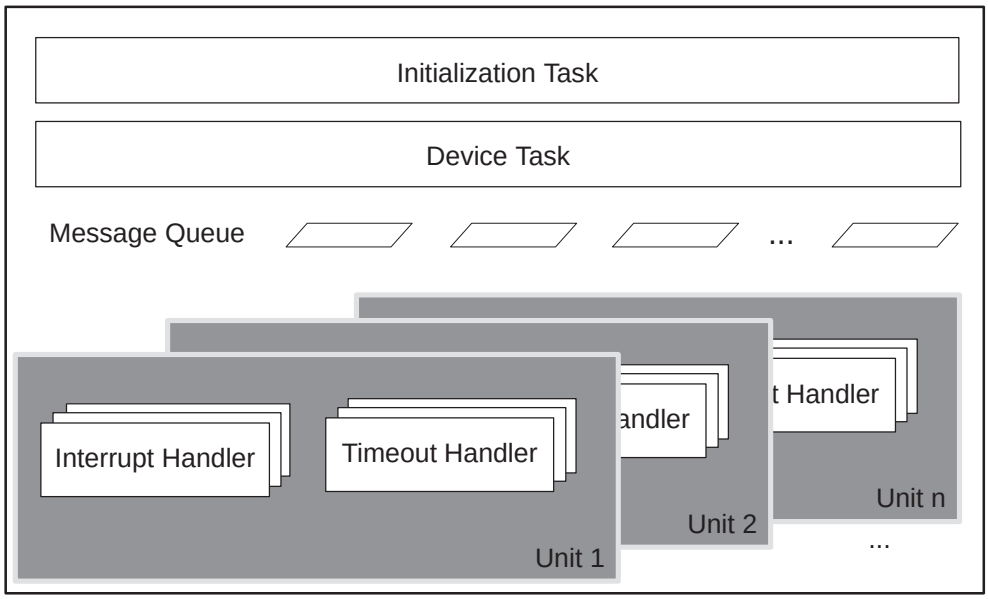

Figure 1-2 Driver with Serial Structure

If the device does not support parallel operation, it is alternatively possible to structure drivers “serially” with only one device task for all units. Serial drivers are used for devices in which only one I/O request can be processed at a time (more units can exist but only one of them can be active). Figure 1-2 shows the structure of a serial driver.

1.3 Operating Principle of a Loadable Driver

This section describes the interaction between a loadable driver and the user. The user can address the driver either via the user program or using the DEVICE command (in the CLI or in the RMOS.INI file).

In the following section you will learn:

- How to load a loadable driver and what happens when it is loaded
- How to issue requests to a loadable driver

Loading the Driver

The user decides when and how the driver is loaded. The driver can be loaded in various ways:

- On system startup via a DEVICE entry in the configuration file RMOS.INI
- In the CLI command interpreter with the DEVICE command from the command line or via an entry in the file CLISTART.BAT
- In the user program with the call ***RmLoadDevice(..)***.

When the driver is loaded, the operating system starts the initialization task. It registers the driver with the operating system, causing an entry to be made in the resource catalog. One or more units are generated, as required, during the initialization phase.

With parallel drivers, a separate device task is started and a message queue is set up for each unit. With serial drivers, a shared device task is started and a message queue is set up for all units. The device task initializes the unit(s) and generates interrupt and timeout handlers as required.

Requests to the Driver

The device task receives the I/O and control requests from the user tasks via the message queue. The requests can be issued in the user program using the following calls:

- RMOS-API calls: ***RmIOOpen(..)***, ***RmIOClose(..)***, ***RmIORead(..)***, ***RmIOWrite(..)*** and ***RmIOControl(..)***
- Functions of the C runtime library: ***open()***, ***close()***, ***ioctl()***, ***read()***, ***write()***, ***lseek()***. Other ANSI-C I/O functions, such as ***fopen()***, ***fclose()***, ***fread()***, ***fwrite()***, ***fgets()***, ***fputs()***, ***fgetc()***, ***fputc()*** etc., can also be used on units of loadable drivers.

  These calls are mapped internally onto the RMOS-API calls listed above, and are therefore not referenced again in this manual. For example, in cases where the manual describes the procedure for opening a unit, we refer exclusively to ***RmIOOpen(..)*** although it would be equally possible to issue the request with ***open()*** or ***fopen()***.

The device task must provide dedicated routines for processing the I/O requests and must acknowledge the processing of requests.

1.4 Programming an Initialization Task

Functions of the Initialization Task

The initialization task is the main entry point of the driver. It is started by the operating system when the driver is loaded for the first time. The initialization task must usually do the following:

1. Register the driver with the operating system

   The driver is entered in the resource catalog under a user–definable name, and a driver–specific data structure is created.

2. Generate the unit(s)

   One or more units are generated as required. The unit–specific data structures are created and the device task(s) started.

3. Terminate the initialization task

The flow of data between the initialization task, user task and operating system is illustrated in Figure 1-3.

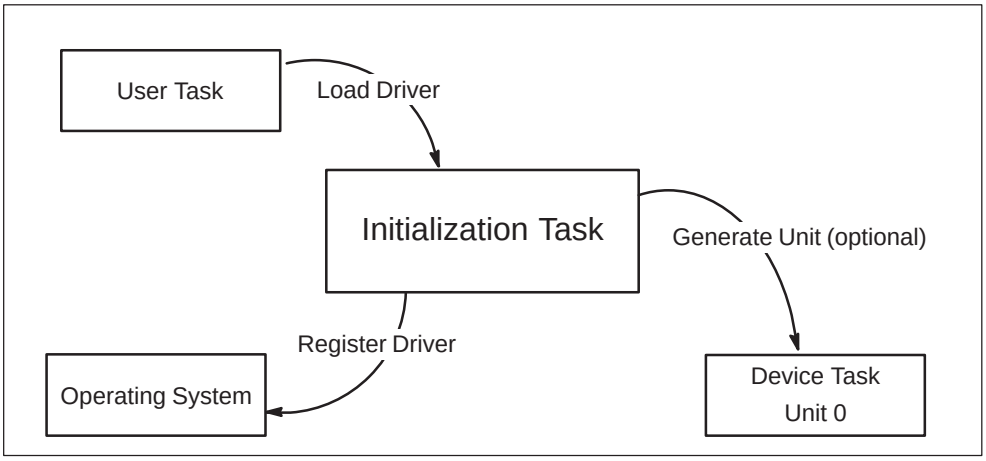

Figure 1-3 Initialization Task

Registering the Driver with the Operating System

Use the following call to register the driver with the operating system:

***RmCreateDevice(*pDeviceName, TaskEntry, Priority, StackSize, ...)*;**

You must pass parameters for the device task to this call: ***TaskEntry*** is the entry address, ***Priority*** is the priority of the device task and ***StackSize*** is the stack size of the device task. The device task is not generated until a unit is generated. ***RmCreateDevice(..)*** enters the driver in the resource catalog with the name specified in ***pDeviceName*** and creates a device structure of the type **RmDeviceStruct**.

Generating a Unit

Use the following call to generate a unit for the driver:

RmCreateDeviceUnit(*pDeviceName, *pUnitName, Priority,..)

You must specify the name of the driver in ***pDeviceName***, as in the ***RmCreateDevice(..)*** call. You can define a new priority for the device task in ***Priority*** or assign ***Priority*=DEVPRI** to retain the priority you defined with the ***RmCreateDevice(..)*** call.

The unit is entered in the resource catalog with the name specified in ***pUnitName*** and a unit structure of the type **RmUnitStruct** is created.

The device task is also started. With serial drivers (***RmCreateDevice*** called with *Type* RM_IO_TYPE_SERIAL), a device task is only generated and started when the first unit is generated. All other units use the same device task.

RmCreateDeviceUnit(..) can be called several times in succession in order to generate further units.

In the example drivers DMYDRV and DRV8250, the corresponding section is enclosed in `#if _CREATE_FIRST_UNIT_ / #endif` and can be activated, if necessary, by changing `#define _CREATE_FIRST_UNIT_` to a value not equal to 0.

The user can generate further units for a registered driver at any subsequent time by issuing the DEVICE command in the CLI or by calling ***RmLoadDevice(..)*** in the user program.

Terminating the Initialization Task

The initialization task must be terminated with ***RmEndTask(..)*** after successful initialization of the driver or by ***RmDeleteTask(RM_OWN_TASK)*** in the event of an error. When an initialization task is terminated by ***RmDeleteTask(RM_OWN_TASK)*** the driver is removed from the system again. In the other situation, the driver remains in the system and is now ready for operation. If several units have already been generated (that is, if device tasks were generated and/or interrupt or timeout handlers were installed), the initialization tasks must always be terminated with ***RmEndTask(..)***.

1.5 Programming a Device Task

A “device task”, responsible for processing the I/O requests of a unit, exists for each device unit of a driver. This device task is generated and started when the unit is generated. In the case of serial drivers (***RmCreateDevice(..)*** with Mode RM_IO_TYPE_SERIAL), a shared device task exists for all units of the driver.

Functions of the Device Task

The device task usually does the following:

1. Initialize the unit
2. Monitor the message queue
3. Process I/O and control requests

The flow of data between the device task, user task and operating system is illustrated in Figure 1-4.

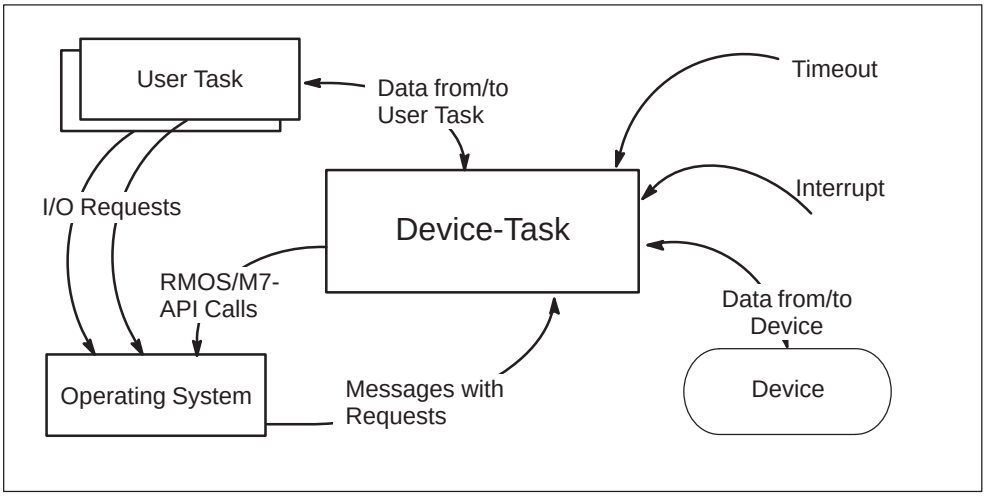

Figure 1-4 Device Task

Initializing a Unit

After a unit has been generated, a message of the type RM_IO_MSG_INIT is sent to the device task of this unit. When the message is received, the device task must perform the necessary unit initialization, for example initialization of the hardware. If necessary, interrupt and timeout handlers are installed during the initialization phase (see Sections 1.6 and 1.7).

No unit initialization is required in the example driver DMYDRV, which is why the corresponding section in the ***execute_request()*** function is blank. Example driver DRV8250 performs the unit initialization in the ***execute_request()*** function when it receives RM_IO_MSG_INIT.

Monitoring the Message Queue

A message queue is automatically generated for the device tasks. The operating system sends a message to this queue for each I/O request (read, write, etc.). The device task has to retrieve the messages with the RMOS-API call ***RmReadMessage(..)*** and process them accordingly.

In addition to the message queue, the device task can use two further queues to process the I/O requests:

- Busy queue if interrupts require processing and if I/O requests can be interrupted with CANCEL
- Exclusive queue if the unit can also be reserved by a task

The driver must manage these queues independently. Pointers for linking I/O requests in the busy and exclusive queues are contained in the unit structure type **RmUnitHeadStruct**.

For less complex drivers which do not have to process interrupts and which support neither RM_IOCTL_CANCEL nor –_RESERVE and –_RELEASE, sequential processing of the requests from the message queue is totally adequate.

Busy Queue

The busy queue usually contains references to I/O requests which occur while another I/O request is currently being processed by the unit.

Figure 1-5 shows a possible sequence in the driver with a message queue and busy queue. In this case, the device task waits for messages with ***RmReadMessage(..)***.

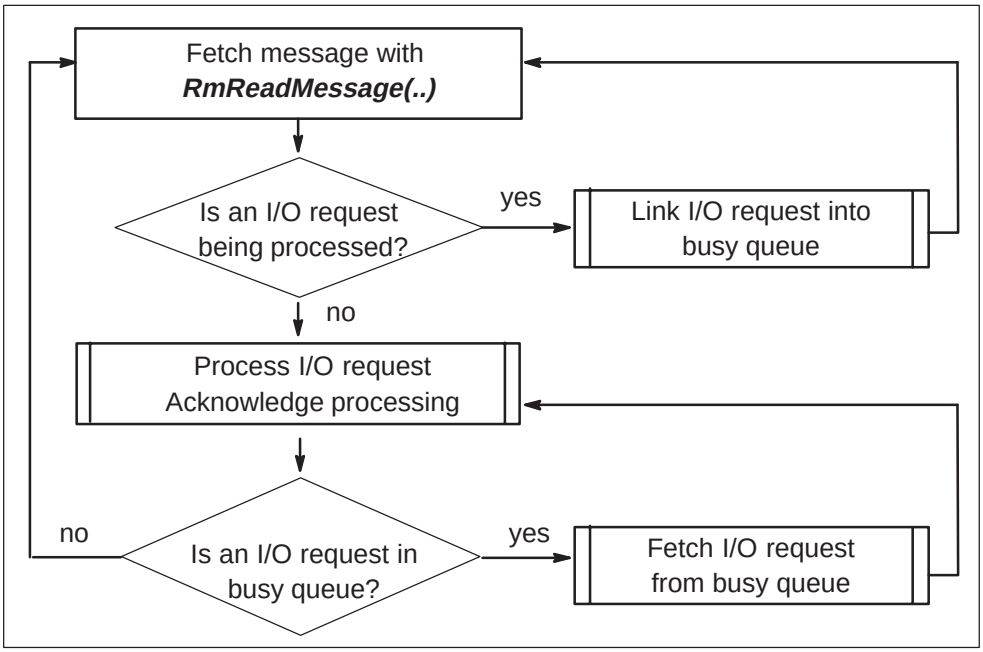

Figure 1-5 I/O Request Processing When the Unit is Not Reserved

If a message is received with the control function RM_IOCTL_CANCEL, the request which is currently being processed must be canceled.

Exclusive Queue

The exclusive queue is only required if the system allows tasks to reserve units with the IOCTL control function RM_IOCTL_RESERVE. In this case, only the I/O requests of the task which has reserved the unit are processed. Requests from other tasks must wait until the unit is released again with the IOCTL control function RM_IOCTL_RELEASE.

In this case, I/O requests which belong to the task which has reserved the unit and which arrive while another I/O request is currently being processed by the unit are linked into the busy queue.

The I/O requests of all other units are linked into the exclusive queue. These requests are not processed until the unit is released.

A possible solution with message queue, busy queue and exclusive queue is contained in example driver DRV8250.

Structure of the Messages for Device Tasks

Each message contains the information required in order to execute the I/O request. A message to the device task consists of a message ID and a parameter block. The message ID identifies the type of request; a pointer to the I/O request block type **RmIORStruct** is passed in the associated parameter block.

The macros for the message ID are stored in RMAPI.H. The values for the message ID and their meaning are defined as follows:

Message ID	Macro for Message ID	Meaning "Sent If..."
0	RM_IO_MSG_INIT	... the unit is generated with ***RmCreateDeviceUnit (..)***
1	RM_IO_MSG_OPEN*)	... the unit is to be opened with the call ***RmIOOpen(..)***
2	RM_IO_MSG_CLOSE*)	... the unit is to be closed with the call ***RmIOClose(..)***
3	RM_IO_MSG_READ	... data are to be read from the unit with ***RmIORead(..)***
4	RM_IO_MSG_WRITE	... data are to be written to the unit with ***RmIOWrite(..)***
5	RM_IO_MSG_CONTROL	... a control function is to be executed for the unit with ***RmIOControl(..)***
6		Reserved for future enhancements
7	RM_IO_MSG_TIMEOUT**)	... the specified timeout has expired**)
8	RM_IO_MSG_FINISH**)	... an I/O request is finished
9 ... 31		Reserved for future enhancements
32 ... 1023		User–defined
1024 ...		Reserved for future enhancements

*) With ***RmIOOpen(..)*** and ***RmIOClose(..)***, messages are only appended to the message queue of the associated device task if the driver requests this by calling ***RmCreateDevice(..)*** with RM_IO_TYPE_OPENMSG.

**) Messages with the IDs RM_IO_MSG_TIMEOUT or RM_IO_MSG_FINISH are not sent to loadable drivers by the operating system. However, they can, if necessary, be sent by a unit (device task, interrupt or timeout handler) to its own device task.

Processing I/O and Control Requests

The device must provide routines for processing the following I/O requests:

- Open unit
- Close unit
- Read data from unit
- Write data to unit
- IOCTL control functions

The processing of messages sent from the operating system to the driver must be acknowledged with the RMOS-API call ***RmQuitRequest(..)***.

In order to execute an I/O request, a device task can call any of the functions which are available to a user task. Special RMOS-API calls, which are only available to device tasks, exist in addition:

Function	Description
RmGetUnitData	Get address of unit structure
RmQuitRequest	Terminate I/O request
RmSetISUnitHandler	Install interrupt handler for driver unit
RmUnitTimeout	Install timeout handler for driver unit
RmUnitTimeoutCancel	Cancel timeout

Note

If calls of the M7-API are used by the device task, you must make sure that the system servers are initialized before the device task is started.

A driver of this type must not be loaded using the DEVICE command in the file RMOS.INI. It can be loaded either from the CLI with the DEVICE command or from a user program with the call ***RmLoadDevice(..)***.

Opening and Closing the Unit

The routines for opening and closing a unit only need to be available if this is explicitly requested in the ***RmCreateDevice(..)*** call by specifying the RM_IO_TYPE_OPENMSG parameter. If no additional operations need to be performed during the open and close routines, the driver has the operating system execute these I/O requests without sending a message to the unit.

Reading and Writing Data

The routines used to read and write data must initiate the following basic sequence of tasks:

1. Fetch the parameters for the task from the I/O request structure type **RmIORStruct**.
2. Start processing and, if necessary, set the processing status in the **io_status* field of the I/O request structure to RM_IO_IN_PROGRESS.

   We recommend you to set the processing status in the **io_status* field. This enables the user program to determine the processing status if the parameter ***wait= RM_CONTINUE*** (do not wait) was set when ***RmIORead(..)*** or ***RmIOWrite(..)*** was called.

   You can only write to **io_status* if *io_status != 0*.
3. Process the request, that is:
   - Fetch the read data from the device and write the data into the read buffer or
   - Fetch the write data from the write buffer and write the data to the device

   The read or write buffer can be found in the **buffer* field of the I/O request structure.
4. Enter the quantity of data read or written into the **io_count* field of the I/O request structure. The number of characters (bytes) is returned for a character device; the number of blocks is returned for a block device.

   You can only write to **io_count* if *io_count != 0*.
5. Terminate the request by calling ***RmQuitRequest(... Status)***. The execution status of the I/O request must be passed in ***Status***.

IOCTL Control Functions

The control functions which can be executed for the unit of a block or character device using the ***RmIOControl(..)*** call are classified into the following categories:

- Required control functions

  These control functions are mandatory for all drivers and their units. They must be supported, because they are used by the operating system (for example by the DEVICE command).

  If a function is (intentionally) not supported by a driver for a particular reason, the driver must still report the RM_OK status with the ***RmQuitRequest(..)*** call. This is only permitted, however, in the case of control functions which do not return data to the caller, for example RM_IOCTL_RESERVE and RM_IOCTL_RELEASE.

- Optional control functions

  Optional control functions can be supported where this is appropriate for the driver. If one of these functions is not supported, the error RM_EIO_INVALID_CONTROL can be reported or, where appropriate, RM_OK (as a dummy function).

  The purpose of the optional control functions is to allow different drivers to use identical control functions with identical syntax for identical tasks.

- Driver–specific control functions

  These control functions can be implemented on demand. If a function is not supported, the error RM_EIO_INVALID_CONTROL must be reported. The following codes can be used for driver–specific control functions: **RM_IOCTL_USER ... RM_IOCTL_USER + 4095**.

The *pBuffer* mentioned below is the parameter of the ***RmIOControl(..)*** call in which the parameter block is passed. The reference to *pBuffer* is stored in the *buffer* field of the I/O request structure.

Control Functions Required for Character Device and Block Device Drivers

RM_IOCTL_RESERVE
Reserve unit for calling task. I/O requests of other tasks are accepted, but are not executed until the unit is released. *pBuffer* is ignored.

RM_IOCTL_RELEASE
Release the unit. I/O requests which were blocked while the unit was reserved are now executed. *pBuffer* is ignored.

RM_IOCTL_INIT
Configure unit with new values. *pBuffer* points to buffer with a driver-specific structure, which is used to pass the configuration data.

RM_IOCTL_INIT_GET
Read in the current configuration of the unit. *pBuffer* points to a buffer with a driver-specific structure, which is used to pass the configuration data (same structure as with RM_IOCTL_INIT).

RM_IOCTL_INIT_ASCII
Configure unit with new values. The new configuration values are passed in the form of ASCII strings. *pBuffer* points to a driver-specific array of strings which contaun the configuration parameters. The last element of the array must be a NULL pointer.

RM_IOCTL_CANCEL
Cancel current I/O request. *pBuffer* is ignored.

RM_IOCTL_GET_PROPERTIES
Determine the function scope of the driver. *pBuffer* points to a structure of the type `RmIOCTLPropertiesStruct`.

RM_IOCTL_GET_VERSION
Find out version of the driver. *pBuffer* points to a structure of the type `RmIOCTLVersionStruct`.

Control Functions Required for Character Device Drivers

RM_IOCTL_MODE
Configure unit with new values for communication (e.g. baud rate). *pBuffer* points to a structure containing the configuration data in a driver-specific form. For example the drivers for serial interfaces require a structure type `RmIOCTLModeSerialStruct`. (see “Control functions for SER8250.DRV” or “Control functions for 3964.DRV” in the description of **RmIOControl** in the Reference Manual ***/281/***)

Control Functions Required for Block Device Drivers

RM_IOCTL_FORMAT
Format the unit. *pBuffer* is ignored.

Optional Control Functions for Character Device and Block Device Drivers

RM_IOCTL_LOCK
Lock the unit, i.e. disable access of all tasks. All subsequent requests to access the unit return the RM_EIO_LOCKED error code. Access to the unit is enabled again using the RM_IOCTL_UNLOCK control function. *pBuffer* is ignored.

RM_IOCTL_UNLOCK
Unlock, i.e. re-enable access to the unit. *pBuffer* is ignored.

RM_IOCTL_GET_STATUS
Get unit status. The status data are written in a driver-specific format to the address pointed to by *pBuffer*.

RM_IOCTL_VERIFY_ON
Activate the data verification. *pBuffer* is ignored.

RM_IOCTL_VERIFY_OFF
Deactivate the data verification. *pBuffer* is ignored.

RM_IOCTL_BUFFER_SETSIZE
Set the size of the background buffer. Data already stored in the background buffer are deleted. In the event of an error (e.g. not enough free memory), the background buffer remains unchanged. *pBuffer* points to a `ulong` which specifies the new buffer size in number of characters.

RM_IOCTL_BUFFER_GETSIZE
Find out the size of the background buffer. The buffer size in number of characters is written to a `ulong`, to which *pBuffer* points.

RM_IOCTL_BUFFER_FLUSH
Flush background buffer. *pBuffer* is ignored.

RM_IOCTL_BUFFER_USED
Determine the number of characters in the background buffer. The number is stored in a `ulong` to which *pBuffer* points.

RM_IOCTL_READ_MODE
Select the mode of ***RmIORead***. *pBuffer* points to a ulong in which either RM_WAIT or RM_CONTINUE is specified. When RM_WAIT is specified, a read request is not completed until the end condition (number of characters, stop character, time–out, ...) has been attained or an error occurs. When RM_CONTINUE is specified, the read request is terminated with RM_IO_NO_DATA when no data (including the end condition) are stored in the background buffer.

RM_IOCTL_READ_MODE_GET
Get the mode of ***RmIORead***. *pBuffer* points to a ulong into which the current read mode (either RM_WAIT or RM_CONTINUE) is written (see also RM_IOCTL_READ_MODE).

Optional Control Functions for Character Device Drivers

RM_IOCTL_LINEMODE_ON
Activate line-oriented reading (end of read request at CR). *pBuffer* is ignored.

RM_IOCTL_LINEMODE_OFF
Deactivate line-oriented reading (end of read request at CR). *pBuffer* is ignored.

RM_IOCTL_READTERM_ON
Define a terminator character that ends a read request. *pBuffer* must point to a `unchar` which contains the character (e.g. CTRL-Z).

RM_IOCTL_READTERM_OFF
Deactivate the terminator character for reading. *pBuffer* is ignored.

RM_IOCTL_WRITETERM_ON
Define a terminator character that ends a write request. *pBuffer* must point to a `unchar` which contains the character (e.g. 0).

RM_IOCTL_WRITETERM_OFF
Deactivate the terminator character for writing. *pBuffer* is ignored.

RM_IOCTL_READSTOP
Define which end condition is used for read requests. The stop character(s) is (are) not written to the user buffer. The end condition is defined by the `char` to which *pBuffer* points. The following values are permitted:

0	Do not use any end condition
1	Use stop character 1
3	Use stop characters 1 and 2, that is cancel when the 1st character is followed by the 2nd stop character.
4	Terminate read request when the number of characters defined by RM_IOCTL_READLEN have been read in.

Two or more conditions can be combined using OR logic.

RM_IOCTL_READSTOP1
Define stop character 1 that terminates the read request. Only valid when activated by RM_IOCTL_READSTOP. *pBuffer* must point to a `char` which contains the stop character.

RM_IOCTL_READSTOP2
Define stop character 2 that terminates the read request. Only valid when activated by RM_IOCTL_READSTOP. *pBuffer* must point to a `char` which contains the stop character.

RM_IOCTL_READSTOP_GET
Read in the end condition activated by RM_IOCTL_READSTOP and the entered stop character. *pBuffer* must point to an array with 3 char in which the current values of RM_IOCTL_READSTOP, RM_IOCTL_READSTOP1 and RM_IOCTL_READSTOP2 are entered.

RM_IOCTL_READLEN
Define the number of characters after which read requests are terminated automatically (only valid when activated by RM_IOCTL_READSTOP). *pBuffer* must point to a `ulong` which contains the number of characters.

RM_IOCTL_READLEN_GET
Read in the number of characters defined by RM_IOCTL_READLEN. The number of characters is written to the `ulong` to which *pBuffer* points.

RM_IOCTL_READTIMEOUT
Define a time span (in ms) specifying the maximum pause between two characters during read requests. If the pause is longer, the read request is terminated. Specifying RM_CONTINUE deactivates the time–out. *pBuffer* must point to a `ulong` which specifies the time span.

RM_IOCTL_READTIMEOUT_GET
Read in the time span specified by RM_IOCTL_READTIMEOUT. The time span is written to the `ulong` to which *pBuffer* points.

RM_IOCTL_WRITESTOP
Define which end condition is used for write requests. The stop character(s) is (are) transferred in addition to the data sent by the user. The end condition is defined by the `char` to which *pBuffer* points. The following values are permitted:

0	Do not use any end condition
1	Use stop character 1
3	Use stop character 1 followed by stop character 2

Two or more conditions can be combined using OR logic.

RM_IOCTL_WRITESTOP1
Define stop character 1 for write requests. Only valid when activated by RM_IOCTL_WRITESTOP. *pBuffer* must point to a `char` which contains the stop character.

RM_IOCTL_WRITESTOP2
Define stop character 2 for write requests. Only valid when activated by RM_IOCTL_WRITESTOP. *pBuffer* must point to a `char` which contains the stop character.

RM_IOCTL_WRITESTOP_GET
Read in the end condition activated by RM_IOCTL_WRITESTOP and the entered stop character. *pBuffer* must point to an array with 3 char in which the current values of RM_IOCTL_WRITESTOP, RM_IOCTL_WRITESTOP1 and RM_IOCTL_WRITESTOP2 are entered.

RM_IOCTL_WRITEDELAY
Define a time span (in ms) specifying the minimum pause observed after transmission of the last character during write requests by the driver, before the request is terminated and a new request is processed. Specifying RM_CONTINUE deactivates the time–out. *pBuffer* must point to a `ulong` in which the time span is specified.

RM_IOCTL_WRITEDELAY_GET
Read in the time span specified by RM_IOCTL_WRITEDELAY. The time span is written to the `ulong` to which *pBuffer* points.

RM_IOCTL_ECHO_ON
Activate the echo of inputs for read requests. *pBuffer* is ignored.

RM_IOCTL_ECHO_OFF
Deactivate the echo of inputs for read requests. *pBuffer* is ignored.

RM_IOCTL_LINE_FEED
Execute a line feed. *pBuffer* is ignored.

RM_IOCTL_FORM_FEED
Execute a form feed (clear screen). *pBuffer* is ignored.

RM_IOCTL_ABORTCHAR_ON
Define abort character (e.g. CTRL-C) and enable checking. *pBuffer* must point to a `uchar` which contains the abort character.

RM_IOCTL_ABORTCHAR_OFF
Disable checking of abort character. *pBuffer* is ignored.

RM_IOCTL_ABORTCHAR_TEST
Find out wether the unit received an abort character since the last call of an ***RmIOxxx*** function. *pBuffer* must point to a `uchar` into which one of the following is written: 0xFF, if the abort character has been received at least once or 0, if no abort character has been received.

RM_IOCTL_ABORTCHAR_BYTE
Define the address of a `uchar` that is set to 0xFF when an abort character is received. *pBuffer* is the address of the `uchar`.*pBuffer*=NULL deactivates this function.

RM_IOCTL_ABORTCHAR_CALL
Define the address of a function to be invoked when an abort character is received. *pBuffer* is the address of the function. *pBuffer*=NULL deactivates this function. The function must have the following interface:

RM_IOCTL_TERMINAL_ON
Enable terminal mode (disable transparent mode).

The following characters are evaluated in terminal mode:

- Read terminator character (RM_IOCTL_READABORT_ON)
- Write terminator character (RM_IOCTL_WRITEABORT_ON)
- Abort character (RM_IOCTL_ABORTCHAR_ON)

pBuffer is ignored.

RM_IOCTL_TERMINAL_OFF
Disable terminal mode (enable transparent mode).

The following characters are ignored:

- Read terminator character (RM_IOCTL_READABORT_ON)
- Write terminator character (RM_IOCTL_WRITEABORT_ON)
- Abort character (RM_IOCTL_ABORTCHAR_ON)

pBuffer is ignored.

Optional Control Functions for Block Device Drivers

RM_IOCTL_FORMAT_TRACK
Format track of the unit. *pBuffer* must point to a an area containing first a `ulong` which references the desired track. The `ulong` must be followed by data needed by the driver in order to format the unit.

Restarting the Device Task

The device task is restarted by the operating system by calling ***RmIOControl(..)*** with the control function RM_IOCTL_RESET. In order to restart a device task you must first terminate all I/O requests currently in progress with the error message RM_EIO_UNIT_RESET. If necessary, the unit can be initialized again and any I/O requests which are in the driver's queue can be terminated with the error message RM_EIO_UNIT_RESET.

All units of the driver must be handled where a serial driver is involved.

Note

With the I/O control function RM_IOCTL_RESET, no message is sent to the device task (neither RM_IO_MSG_INIT nor RM_IO_MSG_CONTROL).

If the unit has to be initialized again when the device task is restarted, it is necessary to reset the unit structure. You can use the ***RmGetUnitData(..)*** call to determine the address of the unit structure which has to be reset.

Example:

Example drivers DMYDRV and DRV8250 call the function ***reset_unit_structure()*** when the device task is restarted. In DMYDRV, this function merely terminates the current request with an error message but does not initialize the unit. In DRV8250, the function terminates all requests being processed by the driver or stored in a queue with an error message; the unit is not initialized again.

1.6 Interrupt Handlers

Introduction

If the hardware controlled by the driver triggers interrupts, you must install one or more interrupt handlers for a unit of the loadable driver by calling ***RmSetISUnitHandler(..)***.

Important features of interrupt handlers for loadable drivers are:

- ***RmSetISUnitHandler(..)*** can only be used to install handlers for hardware interrupts. If necessary, you can install handlers for software interrupts with the RMOS-API calls ***RmSetIntDIHandler(..)*** and ***RmSetISHandler(..)*** for user tasks.
- Installation of the interrupt handlers is unit–specific. You therefore have access to the data structure of the unit. Interrupt handlers installed with ***RmSetIntDIHandler(..)*** or ***RmSetIntISHandler(..)*** have no unit assignment.
- An interrupt handler of a loadable driver consists of two components:
  - An I handler, which runs in interrupt (I) mode
  - An S handler, which runs in system (S) mode.

Where to Find Information About Interrupt Handlers

You will find information about interrupt handlers, for example operating states, processing sequences and RMOS-API functions in Sections 4.18 and 4.19 of **/280/**.

Installing an Interrupt Handler

The interrupt handler must be installed by the device task.

To install an interrupt handler, use the RMOS-API call

RmSetISUnitHandler(IntNum, DeviceID, UnitID, IHandlerEntry, SHandlerEntry)

The parameters passed to the function are the interrupt number ***IntNum***, the identifiers of the driver and the unit and the entry addresses for the two handlers. The identifiers ***DeviceID*** and ***UnitID*** are the output parameters of the calls ***RmCreateDevice(..)*** and ***RmCreateDeviceUnit(..)***.

Both handlers must always be installed. If you do not require one of them, you must install a dummy handler (with an empty function).

While a new handler is being installed, an interrupt must not be triggered for this handler. The DRV8250 example uses the functions ***disable_8250(..)*** and ***enable_8250(..)*** for this purpose.

During the initialization phase, the device task of a unit can install none, one or several interrupt handlers – depending on the hardware.

Calling and Running the Interrupt Handler

When a hardware interrupt is triggered, the specified I handler is invoked by the operating system. The return value of the I handler must be an “int” type. The S handler is called only when the return value of the I handler is not equal to 0. Otherwise, if the return value of the I handler is equal to 0 it is not called. The S handler is not allowed to have a return value. Processing routines within the I handler must be kept as short as possible to avoid compromising the real–time properties of the operating system. Longer processing routines should be implemented in the S handler.

Both handlers must have a Pascal interface. The address of the associated unit structure is passed as a parameter to both interrupt handlers.

While it is running, the I or S handler can, if necessary, send a message of type RM_IO_MSG_FINISH to the device task using the ***RmSendMessage(..)*** call.

An end-of-interrupt routine is not required for the interrupt controller; it is performed automatically by the operating system. The driver must run an EOI handling routine for the hardware if this is required by the hardware.

Note

No memory protection exists within an interrupt handler!

Example

Since example driver DMYDRV does not require an interrupt, no interrupt handler is installed in this driver. However, the *RmSetISUnitHandler()* call is included as a comment in the source code. Example driver DRV8250 uses an interrupt handler to transfer data via the serial interface.

1.7 Timeout Handlers

Introduction

A timeout handler is used to initiate certain actions after a specified length time elapses (for example to cancel an I/O request if the hardware does not respond to a command within the specified time).

Installing a Timeout Handler

The timeout handler must be installed by the device task.

To install a timeout handler, use the RMOS-API call ***RmUnitTimeout(TimeValue, HandlerEntry, pParameter, *pID)***

The parameters passed to the function are the length of time ***TimeValue*** after which the timeout handler is to be called, and the entry address ***HandlerEntry*** of the handler. The output parameter is an identifier stored in ****pID***. You need this identifier to deinstall the timeout handler prematurely (see below).

During the initialization phase, the device task of a unit can install none, one or several timeout handlers as required.

Calling and Running the Timeout Handler

The specified handler is invoked by the operating system when the specified length of time expires. The handler is deinstalled automatically.

Timeout handlers must have a Pascal interface. When a timeout handler is invoked by the operating system, the value ***pParameter***, which was specified when the handler was installed, is passed as a parameter.

While it is running, the timeout handler can send a message of the type RM_IO_MSG_TIMEOUT to the device task using the ***RmSendMessage(..)*** call.

Note

No memory protection exists within a timeout handler!

Premature Deinstallation of a Timeout Handler

Premature deinstallation of the handler is possible with the RMOS-API call ***RmUnitTimeoutCancel(ID)***. ***ID*** is the output parameter of the ***RmUnitTimeout(..)*** call used to install the handler.

1.8 Starting up a Loadable Driver

A loadable driver has the same structure as a normal user program, that is, it consists of an executable program file which can be written with Borland C++. The driver program must be downloaded onto the M7-300/400 and started there.

Preconditions

In order to start up a loadable driver which you have programmed, you must ensure that the following conditions have been met:

- The driver program must be stored in the program folder of the SIMATIC M7-300/400 station.
- The hardware components which are to be controlled by the driver must be installed in the SIMATIC M7-300/400 and entered in the hardware configuration of the project and in the BIOS setup.

Procedure

To start a loadable driver on the M7-300/400, proceed as follows:

1. In the SIMATIC Manager activate the menu item **PLC > Manage M7 System**.
1. In the “Programs” tab, download the driver program and all its components onto the M7-300/400. You will find information about this procedure in Section 4.3 of the User Manual or in the online help of the SIMATIC Manager.
2. If you want to use an entry in a configuration file to load the driver, follow the instructions below (if not, continue with step 5.):

   In the “Configure Op. System” tab, load the file onto the programming device on which the driver is to be loaded and make the appropriate entry (see also Section 3.10 in **/282/**):

   – The DEVICE command in the file RMOS.INI or CLISTART.BAT or

   – The path name of the user program in the file INITTAB
3. Load the modified file onto the M7-300/400.
4. Perform one of the following actions, depending on which configuration file you have modified:

   – If you have modified the file RMOS.INI or INITTAB, start the M7-300/400 again. The driver is loaded on system startup.

   – If you have modified the file CLISTART.BAT, start the CLI. The driver is loaded.
5. Start the CLI and enter the DEVICE command. The driver is loaded immediately.
6. If necessary, generate further units with the DEVICE command.
7. Start the user programs which use the loadable driver.

2 Function Calls

Overview

The RMOS API provides the following function calls for writing loadable drivers:

Function	Description	Page
RmCreateDevice	Register a driver with the operating system	2-2
RmCreateDeviceUnit	Generate a driver unit	2-4
RmGetUnitData	Get the address of the unit structure	2-6
RmQuitRequest	Terminate an I/O request	2-7
RmSetISUnitHandler	Install an interrupt handler for a driver unit	2-8
RmUnitTimeout	Install a timeout handler for a driver unit	2-11
RmUnitTimeoutCancel	Cancel a timeout	2-13

RmCreateDevice

Function

Register a driver with the operating system

Syntax

```
#include <rmapi.h>
int   RmCreateDevice (
        const char *pDeviceName,
        uint Type,
        rmfarproc TaskEntry,
        uint Priority,
        ulong StackSize,
        uint *pDeviceID);
```

Parameter Name	Meaning
pDeviceName	Name of the driver, up to 15 characters
Type	Type of the driver. The following values are permitted: Either RM_IO_TYPE_CHAR or RM_IO_TYPE_BLOCK must be specified. RM_IO_TYPE_SERIAL and RM_IO_TYPE_OPENMSG can also be added optionally using OR logic
TaskEntry	Start address for the device task of the driver
Priority	Priority for the device task; 0 to 255 or RM_HIGHPRI(511)
StackSize	Stack size of the device task in 32–bit words
pDeviceID	Output parameter: returns the ID of the registered driver

Description

`RmCreateDevice` registers a driver with the operating system. A device structure of the type `RmDeviceStruct` is created.

pDeviceName specifies the name by which the driver is entered in the resource catalog. This name is used to identify the driver when generating a unit.

Type specifies the type of driver. Either RM_IO_TYPE_CHAR (character device driver) or RM_IO_TYPE_BLOCK (block device driver) must be specified. If you want to inform a driver unit by generating a message on `RmIOOpen` and `RmIOClose`, you can also specify RM_IO_TYPE_OPENMSG here using a logical OR operation.

A “device task” is normally generated for each unit. This behavior is the same as a parallel driver, since requests to different units are handled concurrently by the different device tasks. The additional specification of RM_IO_TYPE_SERIAL (using OR logic) causes the generation of a serial driver which has only one device task for all units. When a serial driver is used, all requests, even those issued to different units, are handled successively.

TaskEntry is the entry address of the device task used for `RmCreateDeviceUnit`.

Priority specifies the priority for the device task. Values from 0 to 255 and RM_HIGHPRI(511) are permitted. The latter is a macro, the only permissible argument for which is 511. It sets the priority of the device task to a value which is higher than the possible priority of any user task.

The device task is allocated a stack of size *StackSize* in 32–bit words.

Up to 256 drivers can be registered under M7 RMOS32, of which 192 are reloadable.

Return Values

Code	Meaning
RM_OK	Function successfully executed
RM_INVALID_ID	The priority specified in *Priority* is invalid
RM_INVALID_POINTER	Invalid pointer
RM_INVALID_STRING	The length of *pDeviceName* is illegal. It is either 0 or greater than 15
RM_INVALID_TYPE	Type is invalid
RM_IS_ALREADY_CAT-ALOGED	The driver cannot be cataloged, because *pDeviceName* is already in use
RM_MAX_DE-VICES_REACHED	The maximum number of loadable drivers has already been reached; no more drivers could be generated
RM_OUT_OF_MEMORY	Insufficient free memory is available in the heap in order to generate the driver

See Also

RmCreateDeviceUnit, RmGetUnitData, RmDeviceStruct

RmCreateDeviceUnit

Function

Generate a driver unit

Syntax

```
#include <rmapi.h>
int   RmCreateDeviceUnit (
        const char *pDeviceName,
        const char *pUnitName,
        uint Priority,
        uint *pUnitID);
```

Parameter Name	Meaning
pDeviceName	Name of the driver, up to 15 characters
pUnitName	Name of the unit, up to 15 characters
Priority	Priority of the device task; The following values are permitted: • 0 to 255 or RM_HIGHPRI(511) or • RM_DEVPRI = take priority from `RmCreateDevice`
pUnitID	Output parameter: returns the ID of the new unit

Description

A unit with the name *pUnitName* is created for the driver *pDeviceName*. The device task is generated with the entry address specified for `RmCreateDevice` in *TaskEntry*. The priority of the device task is allocated according to *Priority*. Values from 0 to 255, RM_HIGHPRI(511) and RM_DEVPRI are permitted. RM_HIGHPRI(511) sets the priority of the device task to a value which is higher than the possible priority of any user task; RM_DEVPRI accepts the priority specified in the `RmCreateDevice` call.

When serial drivers are used (`RmCreateDevice` called with *Type* RM_IO_TYPE_SERIAL), a device task is created and started only when the first unit is generated. All further units use the same device task.

The address of the device structure for the driver is passed to the device task when it is started. An initialization message (RM_IO_MSG_INIT) is then sent to it.

`RmCreateDeviceUnit` generates a unit structure of the type `RmUnitStruct`.

Up to 256 units can be generated per driver.

Return Values

Code	Meaning
RM_OK	Function successfully executed
RM_INVALID_DEVICE	The driver specified by *pDeviceName* is not a loadable driver
RM_INVALID_ID	The priority specified in *Priority* is invalid
RM_INVALID_POINTER	Invalid pointer
RM_INVALID_STRING	The length of *pDeviceName* or *pUnitName* is illegal. It is either zero or greater than 15
RM_INVALID_TASK_ENTRY	`RmCreateDeviceUnit` was called not by a system task, but by a user task
RM_IS_ALREADY_ CATALOGED	The unit cannot be cataloged, because *pUnitName* is already in use
RM_IS_NOT_CATALOGED	The driver is not cataloged with *pDeviceName*
RM_MAX_UNITS_REACHED	The maximum number of units has already been reached; no more units could be generated
RM_OUT_OF_MEMORY	Insufficient free memory is available in the heap in order to generate the unit

Note

`RmCreateDeviceUnit` cannot be called by user tasks, but only by system tasks (e.g. an initialization task or the device task of a loadable driver).

See Also

RmCreateDevice, RmGetUnitData, RmUnitHeadStruct, RmUnitStruct, RmLoadDevice (in /281/)

RmGetUnitData

Function

Get the address of the unit structure

Syntax

```
#include <rmapi.h>
int   RmGetUnitData  (
        uint TaskID,
        void *pUnitData );
```

Parameter Name	Meaning
TaskID	ID of the device task (RM_OWN_TASK = calling task) for which the associated unit structure is to be determined
pUnitData	Address of a pointer into which the address of the unit structure is to be written

Description

`RmGetUnitData` determines the address of the unit structure of the type `RmUnitStruct` of the device task specified by *TaskID* (RM_OWN_TASK = calling task). The address is written into the pointer to which *pUnitData* points.

When serial drivers are used (RM_IO_TYPE_SERIAL specified with `RmCreateDevice`), a shared device task exists for all units. In this case, the function determines the address of the structure of the first unit.

It is only permissible to call `RmGetUnitData` from system tasks.

Return Values

Code	Meaning
RM_OK	Function successfully executed
RM_INVALID_ID	Invalid task ID or not the ID of a device task
RM_INVALID_POINTER	Invalid pointer
RM_INVALID_TASK_PL	The call was initiated not by a system task, but by a user task

See Also

RmCreateDeviceUnit, RmUnitHeadStruct, RmUnitStruct

RmQuitRequest

Function

Terminate an I/O request

Syntax

```
#include <rmapi.h>
int   RmQuitRequest  (
        RmIORStruct *pIOR,
        int Status );
```

Parameter Name	Meaning
pIOR	Pointer to **RmIORStruct**
Status	Status of the I/O request (RM_OK, RM_IO_xxx, RM_EIO_xxx)

Description

`RmQuitRequest` terminates the I/O request (received by `RmReadMessage`) specified by *pIOR* with the status specified by *Status*. The device task can then handle the next request.

Return Values

Code	Meaning
RM_OK	Function successfully executed
RM_INVALID_POINTER	Invalid pointer

See Also

RmIORStruct, RmReadMessage (in /281/)

RmSetISUnitHandler

Function

Install an interrupt handler for a driver unit

Syntax

```
#include <rmapi.h>
int   RmSetISUnitHandler (
        uint IntNum,
        uint DeviceID,
        uint UnitID,
        rmfarproc IHandlerEntry,
        rmfarproc SHandlerEntry );
```

Parameter Name	Meaning
IntNum	Interrupt number The following hardware interrupts are permitted: IRQx (x=0 to 63) or IRQ(n) (n=0 to 63) The hardware interrupts on the M7-300/400 are 0 to 15.
DeviceID	ID of the driver for whose unit the interrupt handlers are to be installed
UnitID	ID of the unit for which the interrupt handlers are to be installed
IHandlerEntry	Entry address of the I interrupt handler.
SHandlerEntry	Entry address of the S interrupt handler.

Description

`RmSetISUnitHandler` installs an I and S interrupt handler (entries *IHandlerEntry* and *SHandlerEntry*) for the interrupt specified by *IntNum*. Only one hardware interrupt can be specified for *IntNum*. *DeviceID* and *UnitID* specify the unit for which the interrupt handlers are to be installed.

Both handlers must be installed. If one of the two handlers is not required, a dummy handler (with no function) must be installed.

The I handler must have a return value of type int. The S handler is called only when the return value of the I handler is not equal to 0. Otherwise, if the return value of the I handler is equal to 0 it is not called. The S handler must not have a return value. Both handlers must have a Pascal interface. The address of the associated unit structure is passed as a parameter to both interrupt handlers.

Example

```
int _FIXED i_handler (RmUnitStruct *Unit)
{
            ....
}

void _FIXED s_handler (RmUnitStruct *Unit)
{
            ....
}
```

Note

Processing within the I handler must be as short as possible to avoid compromising the real–time properties of the operating system. The S handler should be used for longer handling routines.

An interrupt must not occur for a new interrupt handler while it is being enabled.

An end of interrupt (EOI) routine is not required for the interrupt controller. This is handled automatically by the operating system. However, the driver must perform an EOI routine for the hardware it is controlling.

`RmSetISUnitHandler` must only be called by the device task of the specified unit.

No memory protection exists within the interrupt handler!

Return Values

Code	Meaning
RM_OK	Function successfully executed
RM_INVALID_ACCESS	`RmSetISUnitHandler` was not called by the device task of the unit specified by *DeviceID* and *UnitID*
RM_INVALID_DEVICE	*DeviceID* is invalid or is not the ID of a loadable driver
RM_INVALID_IRQ_NUMBER	The specified interrupt number is invalid (e.g. not the number of a hardware interrupt)
RM_INVALID_POINTER	Invalid pointer
RM_INVALID_UNIT	*UnitID* is invalid
RM_OUT_OF_MEMORY	Insufficient free memory available in the heap

See Also

RmCreateDevice, RmCreateDeviceUnit, RmGetUnitData

RmUnitTimeout

Function

Install a timeout handler for a driver unit

Syntax

```
#include <rmapi.h>
int   RmUnitTimeout  (
        ulong  TimeValue,
        rmfarproc  HandlerEntry,
        void  *pParameter,
        uint  *pID );
```

Parameter Name	Meaning
TimeValue	Time span in ms after which the handler is called
HandlerEntry	Entry address of the timeout handler
pParameter	Parameter to be passed to the timeout handler
pID	Output parameter: returns an identifier which allows the timeout to be canceled

Description

`RmUnitTimeout` installs a timeout handler (entry *HandlerEntry*) which is called after the time span specified by *TimeValue*. The handler is called in the S state.

After execution, *pID contains an ID identifying the timeout. The timeout can be canceled prematurely by calling `RmUnitTimeoutCancel` with this ID.

The timeout handler must have a Pascal interface. The value specified in *pParameter* is passed as a parameter when `RmUnitTimeout` is called.

Example

```
void _FIXED t_handler (void *pParameter)

{

            ....

}
```

Note

`RmUnitTimeout` must only be called by system tasks (e.g. by the device task of a loadable driver).

No memory protection exists within a timeout handler!

Return Values

Code	Meaning
RM_OK	Function successfully executed
RM_INVALID_POINTER	Invalid pointer
RM_INVALID_TASK_PL	Function called not by a system task, but by a user task
RM_OUT_OF_MEMORY	Insufficient free memory available in the heap

See Also

RmUnitTimeoutCancel

RmUnitTimeoutCancel

Function

Cancel a timeout

Syntax

#include <rmapi.h>
int RmUnitTimeoutCancel (uint *ID* **);**

Parameter Name	Meaning
ID	ID of a timeout which was started previously and which is to be canceled.

Description

`RmUnitTimeoutCancel` cancels a timeout started with `RmUnitTimeout`. *ID* (output parameter of `RmUnitTimeout`) is used to specify which timeout is to be canceled.

`RmUnitTimeoutCancel` must only be called by system tasks (e.g. by the device task of a loadable driver).

Return Values

Code	Meaning
RM_OK	Function successfully executed
RM_INVALID_ID	ID is invalid
RM_INVALID_TASK_PL	Function called not by a system task, but by a user task

See Also

RmUnitTimeout

3 Data Structures and Error Codes

Overview

RmDeviceStruct

Syntax

```
#include <rmapi.h>
typedef struct      _RmDeviceStruct
{
   RmDevice         Struct *f_link;
   RmDevice         Struct *b_link;
   uint             id;
   uint             type;
   uint             nuc_flags;
   rmfarproc        task_entry;
   uint             task_priority;
   ulong            stack_size;
   uint             status;
   uint             unit_count;
   RmUnitStruct     *first_unit;
   RmUnitStruct     *last_unit;
   RmTCBStruct      *init_task;
   uint             device_task;
   uint             user[6];
   uint             reserved[6];
} RmDeviceStruct;
```

Description

Device structure `RmDeviceStruct` is used to manage a loadable driver. The internal format of the structure is identical for all loadable drivers. The device structure is generated by the operating system when `RmCreateDevice` is called.

The meaning of the structure elements is as follows:

Field	Type	Meaning
f_link	RmDeviceStruct*	Pointer for linking device structures.
b_link	RmDeviceStruct*	Pointer for linking device structures.
id	uint	Driver identification (= device ID).
type	uint	Type of driver which was specified with `RmCreateDevice`.
nuc_flags	uint	Flags for driver management.
task_entry	rmfarproc	Start address of device task.
task_priority	uint	Default priority for device task.
stack_size	ulong	Stack size for device task.
status	uint	Global status of driver.
unit_count	uint	Number of units generated for this driver.

Field	Type	Meaning
first_unit	RmUnitStruct*	Address of structure `RmUnitStruct` for the first unit of this driver.
last_unit	RmUnitStruct*	Address of structure `RmUnitStruct` for the last unit of this driver.
init_task	RmTCBStruct*	Address of the management structure for the initialization task of the driver.
device_task	uint	Identifies the device task of the driver (only for serial drivers). Only valid when the first unit has been generated.
user	array of 6 uints	Available for use by the driver.
reserved	array of 6 uints	Reserved for future enhancements.

Note

All fields with the exception of **status** and **user** are reserved for use by the operating system and must not be modified.

See Also

RmCreateDevice, RmCreateDeviceUnit, RmUnitHeadStruct, RmUnitStruct

RmIORStruct

Syntax

```
#include <rmapi.h>
typedef struct      _RmIORStruct
{
    RmIORStruct     *f_link;
    RmIORStruct     *b_link;
    RmTCBStruct     *tcb;
    uint            task_id;
    RmUnitStruct    *unit;
    uint            nuc_flags;
    uint            reserved[6];
    uint            user[6];
    ushort          priority;
    ushort          function;
    uint            control;
    uint            mode;
    uint            flagmask;
    RmIOHandle      handle;
    ulong           length;
    void            *buffer;
    ulong           block_addr;
    ulong           *io_count;
    int             *io_status;
}RmIORStruct;
```

Description

The I/O request to be executed is passed to the device task of a driver via a pointer to a request structure `RmIORStruct`. The request structure contains all the information which was passed when the I/O function (`RmIORead`, `RmIOWrite`, `RmIOControl`, `RmIOOpen` or `RmIOClose`) was called.

I/O requests must be terminated with `RmQuitRequest` when they have been completed.

Messages with the IDs RM_IO_MSG_TIMEOUT or RM_IO_MSG_FINISH are not sent by the operating system to loadable drivers, however, if necessary, they can be sent by a unit to its own device task.

The meaning of the structure elements for RM_IO_MSG_TIMEOUT (message from timeout handler to device task: timeout expired) and RM_IO_MSG_FINISH (message from interrupt handler to device task: request processing finished) is driver–dependent.

The meaning of the structure elements is as follows:

Field	Type	Meaning
f_link	RmIORStruct*	Pointer for linking requests which are not executed immediately by the driver. The operating system initializes *f_link* to 0
b_link	RmIORStruct*	Pointer for linking requests which are not executed immediately by the driver. The operating system initializes *b_link* to 0
tcb	RmTCBStruct*	Address of the management structure of the calling task. Reserved for the operating system; must not be changed!
task_id	uint	Identifier of the calling task Must not be changed!
unit	RmUnitStruct*	Address of the unit structure RmUnitStruct of the unit which was passed to this request. Must not be changed!
nuc_flags	uint	Flags for unit management. Reserved for the operating system; must not be changed!
reserved	array of 6 uints	Reserved for future enhancements
user	array of 6 uints	Available for use by the driver
priority	ushort	Priority of the request
function	ushort	Code of the I/O request, identical to the message ID (see below). Must not be changed!
control	uint	Depends on I/O request *function* (see below)
mode	uint	Depends on I/O request *function* (see below)
flagmask	uint	Depends on I/O request *function* (see below)
handle	RmIOHandle	Depends on I/O request *function* (see below)
length	ulong	Depends on I/O request *function* (see below)
buffer	void*	Depends on I/O request *function* (see below)
block_addr	ulong	Depends on I/O request *function* (see below)
io_count	ulong*	Depends on I/O request *function* (see below)
io_status	int*	Depends on I/O request *function* (see below)

The *function* field is identical with the message ID. It can contain one of the following values (code of the I/O request) and must not be changed.

Code	I/O Request	Message ID
0	Unit initialization	(RM_IO_MSG_INIT)
1	`RmIOOpen`	(RM_IO_MSG_OPEN)
2	`RmIOClose`	(RM_IO_MSG_CLOSE)
3	`RmIORead`	(RM_IO_MSG_READ)
4	`RmIOWrite`	(RM_IO_MSG_WRITE)
5	`RmIOControl`	(RM_IO_MSG_CONTROL)
6	Reserved for future enhancements	
7	Timeout	(RM_IO_MSG_TIMEOUT)
8	Request processing finished	(RM_IO_MSG_FINISH)
9 ... 31	Reserved for future enhancements	
32 ... 1023	User–defined	
1024 ...	Reserved for future enhancements	

The meaning of the following structure elements depends on the I/O request (*function* field). These elements must not be changed:

	Function					
Field	**0 Init**	**1 Open**	**2 Close**	**3 Read**	**4 Write**	**5 Control**
control	reserved	reserved	reserved	reserved	reserved	*Control* parameter of `RmIOControl`
mode	reserved	*Mode* parameter of `RmIOOpen`	reserved	*Wait* parameter of `RmIORead`	*Wait* parameter of `RmIOWrite`	*Wait* parameter of `RmIOControl`
flagmask	reserved	reserved	reserved	*FlagMask* parameter of `RmIORead`	*FlagMask* parameter of `RmIOWrite`	*FlagMask* parameter of `RmIOControl`
handle	reserved	Descriptor passed from `RmIOOpen` to the user	*Handle* parameter of `RmIO-Close`	*Handle* parameter of `RmIORead`	*Handle* parameter of `RmIOWrite`	*Handle* parameter of `RmIOControl`
length	reserved	reserved	reserved	*Length* parameter of `RmIORead`	*Length* parameter of `RmIOWrite`	reserved
buffer	reserved	*pUnitName* parameter of `RmIOOpen`	reserved	*pBuffer* parameter of `RmIORead`	*pBuffer* parameter of `RmIOWrite`	*pBuffer* parameter of `RmIOControl`
block_ addr	reserved	reserved	reserved	*BlockAddress* parameter of `RmIORead`	*BlockAddress* parameter of `RmIOWrite`	reserved
io_count	reserved	reserved	reserved	No. of bytes*)	No. of bytes*)	reserved

*) Pointer to a ulong into which the unit must write the number of transferred bytes (character device) or blocks (block device) when the I/O request has finished.

Field	0 Init	1 Open	2 Close	3 Read	4 Write	5 Control
io_status	reserved	reserved	reserved	*pIOStatus* parameter of `RmIORead`	*pIOStatus* parameter of `RmIOWrite`	*pIOStatus* parameter of `RmIOControl`

*) Pointer to a ulong into which the unit must write the number of transferred bytes (character device) or blocks (block device) when the I/O request has finished.

See Also

RmQuitRequest, RmDeviceStruct, RmUnitHeadStruct, RmUnitStruct

RmUnitHeadStruct

Syntax

```
#include <rmapi.h>
typedef struct          _RmUnitHeadStruct
{
    RmUnitStruct      *f_link;
    RmUnitStruct      *b_link;
    RmDeviceStruct    *device;
    uint              nuc_flags;
    uint              id;
    uint              device_task;
    RmIORStruct       *cur_request;
    RmIORStruct       *read_request;
    RmIORStruct       *write_request;
    RmIORStruct       *busy_queue;
    uint              exclusive_task;
    RmIORStruct       *exclusive_queue;
    uint              reserved[6];
} RmUnitHeadStruct;
```

Description

`RmUnitHeadStruct` is part of the unit structure `RmUnitStruct` which is identical for all units (independent of the driver) and which is used to manage the unit of a driver.

The meaning of the structure elements is as follows:

Field	Type	Meaning
f_link	RmUnitStruct*	Pointer for linking unit structures. Reserved for the operating system; must not be changed!
b_link	RmUnitStruct*	Pointer for linking unit structures. Reserved for the operating system; must not be changed!
device	RmDeviceStruct*	Pointer to device structure `RmDeviceStruct` of the driver. Must not be changed!
nuc_flags	uint	Flags for unit management. Reserved for the operating system; must not be changed!
id	uint	Identifier of this unit. Must not be changed!
device_task	uint	Identifier for device task of unit. Must not be changed!

Field	Type	Meaning
cur_request	RmIORStruct*	Pointer to request structure `RmIORStruct` of the I/O request which is currently being processed. *cur_request* is initialized to 0 by the operating system when the unit is generated
read_request	RmIORStruct*	Pointer to request structure `RmIORStruct` of the current read request (in cases where read and write requests are processed concurrently). *read_request* is initialized to 0 by the operating system when the unit is generated
write_request	RmIORStruct*	Pointer to request structure `RmIORStruct` of the current write request (in cases where read and write requests are processed concurrently). *write_request* is initialized to 0 by the operating system when the unit is generated
busy_queue	RmIORStruct*	Pointer for linking I/O requests (`RmIORStruct`) which have been fetched from the message queue of the device task by the RMOS API call `RmReadMessage`, but which have not yet been executed because an I/O request is still being processed. *busy_queue* is initialized to 0 by the operating system when the unit is generated
exclusive_task	uint	Identifier of the task for which the associated unit is currently reserved by RM_IOCTL_RESERVE. *exclusive_task* is initialized to –1 by the operating system when the unit is generated. The driver must store the identifier of the calling task in *exclusive_task* when the RM_IOCTL_RESERVE operation is executed. When the unit is released by RM_IOCTL_RELEASE, *exclusive_task* must be reset to -1
exclusive_queue	RmIORStruct*	Pointer for linking I/O requests (`RmIORStruct`) which could not be executed immediately because the unit was reserved. *exclusive_queue* is initialized to 0 by the operating system when the unit is generated
reserved	array of 6 uints	Reserved for future enhancements

See Also

RmCreateDevice, RmCreateDeviceUnit, RmGetUnitData, RmDeviceStruct, RmUnitStruct

RmUnitStruct

Syntax

```
#include <rmapi.h>
typedef struct          _RmUnitStruct
{
   RmUnitHeadStruct     head;
   uchar                data[256];
} RmUnitStruct;
```

Description

Unit structure `RmUnitStruct` is used by the driver and the operating system for unit management. The structure consists of the header `RmUnitHeadStruct`, the internal format of which is identical for all units (independent of the driver), and a 256–byte array of unit–specific data. The unit structure is generated by the operating system when the unit is created.

The meaning of the structure elements is as follows:

Field	Type	Meaning
head	RmUnit-HeadStruct	Header of the unit structure with standard format. The meaning of the elements is described under `RmUnitHeadStruct`
data	array of 256 uchars	Unit-specific data. The driver can use this field to store different data for the various units (e.g. I/O addresses)

See Also

RmCreateDevice, RmCreateDeviceUnit, RmGetUnitData, RmDeviceStruct, RmUnitHeadStruct

Error Codes

This section describes the error codes which can be returned by the calls for loadable drivers. The corresponding numeric value and a brief explanation is provided in addition to definition.

The following error codes can occur with all loadable drivers

Error Code	Value	Explanation
RM_EIO_PARAMETER	0×0401	Parameter error
RM_EIO_INVALID_CONTROL	0×0402	The specified control function is not supported
RM_EIO_INVALID_ACCESS	0×0403	Descriptor is not open for type of access used (Read/Write)
RM_EIO_UNIT_RESERVED	0×0404	Unit is already reserved or unit was not reserved by the calling task
RM_EIO_CANCEL	0×0405	Request was canceled by RM_IOCTL_CANCEL
RM_EIO_LOCKED	0×0406	The unit has been locked by RM_IOCTL_LOCK
RM_EIO_IO_ERROR	0×0407	Request canceled due to I/O error
RM_EIO_PARITY_ERROR	0×0408	Request canceled due to parity error
RM_EIO_OVERRUN_ERROR	0×0409	Request canceled due to overrun error
RM_EIO_TIMEOUT	0×040A	Request canceled with timeout
RM_EIO_INVALID_STATE	0×040B	An error has occurred during status check of the controller (e.g. parity)
RM_EIO_NO_HARDWARE	0×040C	Hardware does not exist or is defective
RM_EIO_INIT_FAILED	0×040D	Initialization of the unit was not possible
RM_EIO_UNIT_RESET	0×040E	Request canceled by RM_IOCTL_RESET

Error codes 0x040F to 0x047F are reserved for further enhancements.

Driver-Specific Error Codes

Error codes 0x0480 to 0x04FF can be given driver-specific assignments, e.g. for the 3964(R) driver.

Messages

The following messages can occur as return values

Define	Value	Explanation
RM_IO_QUEUED	–1024	Request was appended to queue
RM_IO_IN_PROGRESS	–1025	Request is currently being processed
RM_IO_NO_DATA	–1026	No data exist

Literature List

A

/70/ Manual: S7-300 Programmable Controller,
Hardware and Installation

/71/ Reference Manual: S7-300 and M7-300 Programmable Controllers,
Module Specifications

/77/ Manual: Application Module FM 356,
Hardware and Installation

/80/ Manual: M7-300 Programmable Controller,
Hardware and Installation

/100/ Manual: S7-400/M7-400 Programmable Controllers,
Hardware and Installation

/101/ Reference Manual: S7-400/M7-400 Programmable Controllers,
Module Specifications

/106/ Manual: Application Module FM 456,
Hardware and Installation

/231/ User Manual: Standard Software for S7 and M7,
STEP 7

/234/ Programming Manual: System Software for S7-300 and S7-400,
Program Design

/235/ Reference Manual: System Software for S7-300 and S7-400,
System and Standard Functions

/254/ Manual: CFC Continuous Function Charts,
Volume 1

/280/ Programming Manual: System Software for M7-300 and M7-400,
Program Design

/281/ Reference Manual: System Software for M7-300 and M7-400,
System and Standard Functions

/282/ User Manual: System Software for M7-300 and M7-400,
Installation and Operation

/290/ User Manual: ProC/C++ for M7-300 and M7-400,
Writing C Programs

/291/ User Manual: ProC/C++ for M7-300 and M7-400,
Debugging C Programs

Index

T

U

Siemens AG
A&D ASE 48
Postfach 4848
D-90327 Nürnberg
Federal Republic of Germany

From:
Your Name: _
Your Title: _
Company Name: _
Street: _
City, Zip Code _
Country: _
Phone: _

Please check any industry that applies to you:

- ❒ Automotive
- ❒ Chemical
- ❒ Electrical Machinery
- ❒ Food
- ❒ Instrument and Control
- ❒ Nonelectrical Machinery
- ❒ Petrochemical
- ❒ Pharmaceutical
- ❒ Plastic
- ❒ Pulp and Paper
- ❒ Textiles
- ❒ Transportation
- ❒ Other _ _ _ _ _ _ _ _ _ _ _

Remarks Form

Your comments and recommendations will help us to improve the quality and usefulness of our publications. Please take the first available opportunity to fill out this questionnaire and return it to Siemens.

Please give each of the following questions your own personal mark within the range from 1 (very good) to 5 (poor).

1. Do the contents meet your requirements? ☐
2. Is the information you need easy to find? ☐
3. Is the text easy to understand? ☐
4. Does the level of technical detail meet your requirements? ☐
5. Please rate the quality of the graphics/tables: ☐
6. ☐
7. ☐
8. ☐

☐

Additional comments:

_ _
_ _
_ _
_ _
_ _
_ _
_ _
_ _
_ _
_ _
_ _
_ _